Lógica Computacional e Algoritmos

Uma introdução assistida por computador

Flávio L. C. de Moura
Departamento de Ciência da Computação
Universidade de Brasília[1]

20 de dezembro de 2022

[1]flaviomoura@unb.br

Capítulo 1

Introdução

Este material está sendo desenvolvido para dar suporte aos alunos de graduação da Universidade de Brasília nas disciplinas de Lógica Computacional 1 e Projeto e Análise de Algoritmos. Normalmente, o público que cursa estas disciplinas pertence aos cursos de Computação, Matemática e Engenharias em geral, mas acreditamos que este material seja útil para todos que tenham interesse nos temas de lógica e algoritmos. A primeira parte deste material apresenta os conceitos básicos de lógica partindo da lógica proposicional e chegando na lógica de primeira ordem. A lógica de primeira ordem pode ser vista como a "lógica padrão"utilizada, ainda que informalmente, em Matemática e Computação. Este estudo inicial de lógica será útil para a segunda parte que trata da análise de algoritmos. Tanto o estudo de lógica quanto o de algoritmos dá especial atenção à construção de provas (matemáticas). Neste contexto, as provas são inicialmente feitas em papel e lápis (provas informais), e posteriormente em computador (provas formais). A construção de provas é um tema que costuma ser espinhoso, de forma que aqui tentaremos facilitar o processo de familiarização com este tema partindo de provas simples, para em seguida explorarmos situações mais complexas. Novas atividades serão incorporadas sempre que possível, de forma que este material está em constante atualização.

Linguagens naturais, como o Português por exemplo, são ambíguas por natureza, e para evitarmos possíveis dúvidas na leitura de fórmulas ou propriedades utilizaremos linguagens mais restritas. Iniciaremos com a linguagem da lógica proposicional (LP), que nos permitirá resolver diversos problemas interessantes. Estes problemas serão estudados também no contexto computacional. Com isto, queremos dizer que resolveremos problemas manualmente, isto é, em papel e lápis, e também no computador.

Apesar da LP possuir limitações de expressividade, ela será útil para que possamos entender a dinâmica da construção de provas, mas a lógica efetivamente usada no dia a dia do matemático ou do cientista da computação é a Lógica de Primeira Ordem (LPO), que nos permitirá expressar propriedades de algoritmos de forma mais natural. Durante esta caminhada, estudaremos um assunto fundamental que está presente em diversos contextos: *indução*. Intuitivamente, o conceito de indução é bastante simples, mas a sua aplicação em situações específicas costuma gerar muita dúvida.

A construção de provas mecânicas, ou seja, provas feitas em computador, é uma atividade que tem despertado interesse crescente nas últimas décadas em função da forma como a computação tem se infiltrado no nosso dia a dia. Mas aqui precisamos de uma pequena pausa para explicarmos o que queremos dizer com provas feitas por computador. Esta explicação se faz necessária porque existem pelo menos duas abordagens distintas no que se refere a este assunto: os provadores automáticos de teoremas por um lado, e os assistentes de prova por outro.

Um provador automático de teoremas é um programa munido de uma heurística que recebe um teorema como argumento e tenta, de forma automática, encontrar uma prova para o teorema dado [33, 21, 26]. Um assistente de provas por outro lado, consiste em um programa que requer a orientação/interação do usuário para poder construir uma prova. Ou seja, o usuário vai guiando o sistema na construção de prova, enquanto o sistema verifica se cada passo dado/sugerido pelo usuário está correto. São exemplos de assistentes de prova o PVS[29], o Isabelle/HOL[27], o Lean[14] e o Coq[37]. Neste

material trabalharemos com o assistente de provas Coq, que é um sistema de código aberto e que pode ser instalado em sistemas Linux, MacOs e Windows, e até mesmo ser executado via browser[1].

Existem materiais muito interessantes que servem como tutoriais do Coq, como por exemplo, [32], ou [8]. Aqui não pretendemos apresentar um tutorial do Coq. Nosso foco é o estudo da lógica proposicional e de primeira ordem, assim como a utilização delas no estudo de algoritmos. Para isto utilizaremos o Coq como ferramenta de apoio mostrando como um assistente de provas pode ser útil nesta caminhada. Aqui é importante observar também que um assistente de provas é basicamente uma linguagem de programação juntamente com uma linguagem de especificação, ou seja, além da linguagem de programação existem uma camada lógica adicional, chamada de linguagem de especificação, que nos permite expressar os lemas e teoremas, por exemplo. A camada lógica do Coq é baseada em um formalismo conhecido como *cálculo de construções indutivas* [30] que é muito mais expressivo do que a lógica de primeira ordem que estudaremos aqui. Neste sentido, utilizaremos apenas uma pequena parte do poder de computacional do Coq. Não assumimos nenhum conhecimento prévio de Coq. A ideia aqui é que você possa reproduzir os temas abordados em Coq a partir do zero: simplesmente abra o Coq com a interface de sua preferência e siga as orientações. Nem tudo que será abordado aqui terá uma versão correspondente em Coq, já que alguns temas serão puramente teóricos e foram pensados para serem feitos em papel e lápis. E mesmo a etapa de construção de provas em um assistente de provas deve ser precedida de um esboço em papel e lápis.

No contexto de algoritmos e desenvolvimento de *software* é comum a utilização de testes como método de validação. Ou seja, o programa (ou *software*) é executado com diversas entradas distintas, e se nenhum problema é encontrado, o programa é considerado bom o suficiente para ser utilizado. De fato, a primeira coisa que fazemos após implementar um algoritmo é testá-lo para diversas entradas, e caso alguma resposta seja incorreta, uma revisão da implementação é feita para corrigir o erro, e então novos testes são realizados. Este processo é repetido até que o programador sinta confiança na implementação, mas depois de todos estes testes é possível dizer que o programa é correto? Certamente não! Pensando no caso particular da implementação de um algoritmo de ordenação listas de naturais ou inteiros (ou qualquer estrutura munida de uma ordem total), sabemos que existe uma infinidade de listas de inteiros que podem ser utilizadas nos testes, e portanto não é possível testar todas elas. Em se tratando de programas utilizados em sistemas críticos (aviação, medicina, sistemas bancários, etc), por menores que sejam as chances de erros, falhas não são toleradas em sistemas críticos. O que fazer então para garantir a correção de um programa? Uma abordagem possível consiste em utilizar a lógica para **provar** a correção do programa! Uma prova de uma propriedade de um programa fornece a garantia de que o programa satisfaz a propriedade provada **sempre**! Esta é a abordagem que utilizaremos aqui, e que tem se mostrado cada vez mais importante para o desenvolvimento da Matemática[17, 15, 2, 3] e Computação[23, 31, 28]. Para concluir esta seção e começarmos a colocar a mão na massa, listamos três exemplos famosos de erros em sistemas computacionais:

1. **Therac-25**: Uma máquina de radioterapia controlada por computador causou a morte de pelo menos 6 pacientes entre 1985 e 1987 por overdose de radiação.

2. **Pentium FDIV**: Um erro na construção da unidade de ponto flutuante do processador Pentium da Intel causou um prejuízo de aproximadamente 500 milhões de dólares para a empresa que se viu forçada a substituir os processadores que já estavam no mercado em 1994.

3. **Ariane 5**: Um foguete que custou aproximadamente 7 bilhões de dólares para ser construído explodiu no seu primeiro voo em 1996 devido ao reuso sem verificação apropriada de partes do código do seu predecessor.

A *Lógica Computacional* (LC) tem por objetivo utilizar a lógica para raciocinar sobre Computação, ou seja, consiste na utilização da lógica para a resolução de problemas computacionais. Isto pode ser feito considerando a lógica como uma linguagem de programação [36], ou considerando uma mecanização

do raciocínio lógico de forma a permitir a resolução de exercícios no computador, ao invés da resolução usual em papel e lápis [20]. A abordagem que utilizaremos difere das anteriores, mas possui um pouco de cada uma delas como veremos a seguir.

Para explicar a nossa abordagem, suponha que você tenha um grande banco de dados com informações de uma determinada população, e que por alguma razão precise ordenar estas informações por idade em determinado momento; em outro momento, a ordenação que precisa ser feita é por nome ou outro critério qualquer. O que você faz? Uma alternativa é utilizar uma implementação já feita, mas precisamos então perguntar se a implementação utilizada é correta. A alternativa que utilizaremos consiste em construirmos uma implementação e provarmos que ela está correta. Em particular, estudaremos sistemas dedutivos que nos permitirão expressar e provar propriedades de programas[4].

Já deixamos claro que vamos **provar** muita coisa aqui. Mas o que é uma prova? Uma resposta possível "é um argumento feito para convencer alguém"[35]. O problema deste argumento é que pessoas diferentes podem ter compreensões distintas sobre o argumento, de forma que o argumento pode ser uma prova para uma pessoa, mas não para a outra... estranho, não? Uma definição geral e abstrata para a noção de prova não é uma tarefa fácil, mas forneceremos uma definição precisa em um contexto mais restrito, a saber, o da lógica simbólica[19, 39].

Este material está sendo construído a partir de notas de aulas utilizadas nas minhas turmas de Lógica Computacional 1 e Projeto e Análise de Algoritmos na Universidade de Brasília nos últimos anos, e portanto está em constante atualização.

Agradecimentos Este material foi escrito com o apoio do programa Aprendizagem para o Terceiro Milênio (A3M) coordenado pelo CEAD/UnB, que viabilizou o trabalho do estudante Rafael Monteiro Rodrigues na elaboração de diversas atividades. O estudante Gabriel Silva também fez contribuições importantes na elaboração das atividades, e em particular, na formalização do algoritmo *mergesort*. No entanto, todos os erros eventualmente existentes são de minha inteira responsabilidade. Críticas e/ou sugestões são muito bem-vindas e podem ser enviadas para `flaviomoura@unb.br`.

Parte I

Lógica

Capítulo 2

A Lógica Proposicional

Iniciaremos nosso estudo com a *lógica proposicional* (LP) que é uma lógica baseada na noção de **proposição**. Uma proposição, por sua vez, é uma sentença que pode ser qualificada como verdadeira ou falsa. São exemplos de proposição:

- 2+2 = 4.
- 1+3 < 0.
- 2 é um número primo.
- João tem 20 anos e Maria tem 22 anos.

Mas nem toda sentença é uma proposição. De fato, a sentença "Feche a porta!", ou ainda a pergunta "Qual é o seu nome?"não podem ser qualificadas como verdadeira ou falsa, e portanto não são proposições. Algumas proposições podem ser divididas em proposições menores. Por exemplo, a proposição "João tem 20 anos e Maria tem 22 anos"é composta da proposição "João tem 20 anos"e da proposição "Maria tem 22 anos", que por sua vez não podem mais serem divididas porque os pedaços menores não são mais qualificáveis como verdadeiro ou falso. Uma proposição que não pode ser dividida é um elemento básico utilizado na construção de proposições mais complexas, que chamaremos de *fórmula atômica*. Utilizaremos letras latinas minúsculas para representar fórmulas atômicas. Por exemplo, podemos utilizar a letra q para representar a proposição "Maria tem 22 anos", e a letra p para "João tem 20 anos". A proposição do exemplo acima é construída com a utilização do conectivo "E"(conjunção), que será representado pelo símbolo $\wedge$. Com esta simbologia, podemos codificar a proposição "João tem 20 anos e Maria tem 22 anos"pela fórmula $p \wedge q$. Vejamos então a gramática utilizada na construção das fórmulas da LP, que serão representadas por letras gregas minúsculas:

$$\varphi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \tag{2.1}$$

A gramática (2.1) define como as fórmulas da LP são construídas. Ela possui 6 construtores:

1. O primeiro denota uma variável proposicional, e caracteriza uma fórmula atômica, i.e. uma fórmula que não pode ser subdividida em fórmulas menores.
2. O segundo construtor é uma constante que denota o absurdo ($\bot$), que também é uma fórmula atômica. O absurdo será utilizado quando tivermos informações contraditórias em nossas provas. Isto ficará mais claro nos exemplos.

3. O terceiro construtor denota a negação.

4. O quarto construtor denota a conjunção.

5. O quinto construtor denota a disjunção.

6. O sexto construtor é a implicação.

Uma gramática como (2.1) nos fornece as regras sintáticas para a construção das fórmulas da LP. São quatro construtores recursivos (negação, conjunção, disjunção e implicação) também chamados de conectivos lógicos, e dois não recursivos. Apesar da gramática apresentada acima não incluir a bi-implicação, este é um conectivo bastante utilizado, e pode ser escrito em função dos outros conectivos: $\varphi \leftrightarrow \psi$ é o mesmo que $(\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$. Na verdade, a gramática apresentada possui redundâncias, isto é, conectivos que podem ser escritos em função de outros, mas veremos isto posteriormente.

2.1 A Lógica Proposicional Minimal

O sistema conhecido como *dedução natural* será utilizado para a construção das provas. Este sistema foi criado pelo lógico alemão Gerhard Gentzen (1909-1945), e consiste em um sistema lógico composto por um conjunto de regras de inferência que tenta capturar o raciocínio matemático da forma mais *natural* possível. Como veremos, estas regras nos permitem derivar novos fatos a partir das premissas. Os fatos a serem provados são representados por meio de fórmulas da LP. Neste contexto, o primeiro conceito importante que aparece é o de *sequente*. Formalmente, um sequente é um par cujo primeiro elemento é um conjunto finito de fórmulas (hipóteses), e o segundo elemento é uma fórmula (conclusão). Assim, se $\varphi_1, \varphi_2, \ldots, \varphi_n$ são as hipóteses de um dado problema, e se ψ é a sua conclusão, escrevemos $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ para representar o sequente que simboliza que ψ tem uma prova a partir das hipóteses $\varphi_1, \varphi_2, \ldots, \varphi_n$. O conjunto $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, isto é, a primeira componente do sequente $\varphi_1, \varphi_2, \ldots, \varphi_n \vdash \psi$ também será chamado de *contexto* ao longo do texto, e normalmente será escrito sem as chaves que usualmente são usadas para representar conjuntos. Este é um abuso de linguagem usado para não deixar a notação sobrecarregada. Assim, se Γ denota um conjunto finito de fórmulas, ao invés de $\Gamma \cup \{\varphi\} \vdash \psi$, escreveremos simplesmente $\Gamma, \varphi \vdash \psi$, onde Γ, φ deve então ser lido como o conjunto que contém a fórmula φ e todas as fórmulas de Γ. O conceito de prova agora será definido de forma mais precisa. Concretamente, uma prova (ou uma derivação) de um sequente da forma $\Gamma \vdash \psi$ é uma árvore finita onde cada nó corresponde a uma regra válida. A raiz da árvore é anotada com a conclusão, ou seja, o sequente que queremos provar, e as folhas são anotadas com axiomas. As regras são representadas por *regras de inferência*, onde cada premissa e a conclusão correspondem a um sequente:

$$\frac{\Gamma_1 \vdash \gamma_1 \ \Gamma_2 \vdash \gamma_2 \ldots \Gamma_k \vdash \gamma_k}{\Gamma \vdash \psi}$$

onde $k \geq 0$. Quando $k = 0$ e $\Gamma = \{\psi\}$ a regra corresponde a um *axioma*:

$$\frac{}{\psi \vdash \psi} \text{ (Ax)}$$

Uma prova (sequência de passos dedutivos) pode ser representada por meio de uma estrutura de árvore, onde os nós são anotados com sequentes. A raiz da árvore é anotada com o sequente que queremos provar, isto é, $\Gamma \vdash \psi$, e as folhas da árvore são axiomas.

Como veremos, a construção desta árvore deve obedecer alguns critérios que detalharemos ao longo deste capítulo, mas em linhas gerais, o principal critério consiste em obedecer as regras que definem o sistema de dedução natural. As regras são divididas em dois tipos: *regras de introdução* e *regras de eliminação* dos conectivos.

1. O fragmento implicacional da lógica proposicional

   Iniciaremos com a regra de *eliminação da implicação* que é bastante conhecida. Denotaremos esta regra, que também é conhecida como *modus ponens*, por $(\to_e)$:

$$\frac{\Gamma_1 \vdash \varphi \to \psi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \psi}\ (\to_e)$$

   onde Γ_1, Γ_2 é o mesmo que $\Gamma_1 \cup \Gamma_2$.

   A regra $(\to_e)$ nos diz que a partir de uma prova de $\Gamma_1 \vdash \varphi \to \psi$ e de outra prova de $\Gamma_2 \vdash \varphi$ podemos concluir que $\Gamma_1, \Gamma_2 \vdash \psi$, ou seja, uma prova de ψ a partir da união de Γ_1 com Γ_2. As regras de introdução são bastante intuitivas e, em certo sentido, podem ser vistas como uma definição do conectivo que estão introduzindo. A regra de *introdução da implicação*, denotada por $(\to_{\mathrm{i}})$, possui alguns detalhes importantes. Para construirmos uma prova de uma implicação, digamos do sequente $\Gamma \vdash \varphi \to \psi$, precisamos construir uma prova de ψ tendo φ como hipótese adicional ao contexto Γ. Em outras palavras, na leitura de baixo para cima, reduzimos o problema de provar $\Gamma \vdash \varphi \to \psi$ ao novo problema (possivelmente mais simples) de provar $\Gamma, \varphi \vdash \psi$:

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$$

   Também podemos observar esta regra de cima para baixo. Neste caso, ela nos permite passar uma fórmula do conjunto de hipóteses para o consequente como antecedente de uma implicação. Assim, a fórmula φ que era uma das hipóteses necessárias para provar ψ, deixa de ser hipótese, e passa a ser antecedente de uma implicação no consequente. Considerando o subconjunto das fórmulas da lógica proposicional construídas apenas com a implicação (, variáveis e a constante $\bot$) e as regras $(\to_e)$ e $(\to_{\mathrm{i}})$ temos o chamado *fragmento implicacional da lógica proposicional*. O interesse computacional deste fragmento está diretamente relacionado ao algoritmo de inferência de tipos em linguagens funcionais[18]. O fundamento teórico destas linguagens é o cálculo-λ[6] desenvolvido por Alonzo Church em 1936[9, 10]. Para mais detalhes veja o Capítulo 1 de [4]. Como primeiro exemplo, vamos considerar o sequente $\vdash (p \to q) \to (q \to r) \to p \to r$. A primeira observação a ser feita aqui é que a implicação é associativa à direita, ou seja, $\varphi \to \psi \to \gamma$ deve ser lido como $\varphi \to (\psi \to \gamma)$, e não como $(\varphi \to \psi) \to \gamma$. Portanto, o sequente que queremos provar deve ser lido como $\vdash (p \to q) \to ((q \to r) \to (p \to r))$. Utilizando inicialmente a regra $(\to_{\mathrm{i}})$, temos a seguinte situação:

$$\frac{p \to q \vdash (q \to r) \to (p \to r)}{\vdash (p \to q) \to ((q \to r) \to (p \to r))}\ (\to_{\mathrm{i}})$$

   Agora podemos aplicar novamente a regra $(\to_{\mathrm{i}})$:

$$\frac{\dfrac{p \to q, q \to r \vdash p \to r}{p \to q \vdash (q \to r) \to (p \to r)}\ (\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\ (\to_{\mathrm{i}})$$

   E mais uma vez, já que a conclusão do sequente é ainda uma implicação:

$$\dfrac{\dfrac{\dfrac{p \to q, q \to r, p \vdash r}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Agora não é mais possível utilizar a regra $(\to_{\mathrm{i}})$ porque a conclusão r não é uma implicação, mas podemos utilizar a hipótese $q \to r$ para obter r, desde que tenhamos q para utilizarmos $(\to_e)$. Neste ponto, a árvore é bifurcada em dois ramos e precisamos dividir o contexto de forma adequada em cada um dos ramos.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{p \to q, p \vdash q \qquad q \to r \vdash q \to r}{p \to q, q \to r, p \vdash r}\,(\to_e)}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Observe que o ramo da direita consiste em um axioma, e portanto consiste em uma folha da árvore. No ramo da esquerda podemos obter q por meio da regra $(\to_e)$ com as hipóteses $p \to q$ e p. A prova completa é dada a seguir:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,(\mathrm{Ax}) \quad \dfrac{}{p \to q \vdash p \to q}\,(\mathrm{Ax})}{p \to q, p \vdash q}\,(\to_e) \quad \dfrac{}{q \to r \vdash q \to r}\,(\mathrm{Ax})}{p \to q, q \to r, p \vdash r}\,(\to_e)}{p \to q, q \to r \vdash p \to r}\,(\to_{\mathrm{i}})}{p \to q \vdash (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})}{\vdash (p \to q) \to (q \to r) \to (p \to r)}\,(\to_{\mathrm{i}})$$

Exercício 1. *Prove o sequente* $\vdash (p \to p \to q) \to p \to q$.

Exercício 2. *Prove o sequente* $\vdash (p \to q) \to (p \to p \to q)$.

Exercício 3. *Prove o sequente* $\vdash (q \to r \to t) \to (p \to q) \to p \to r \to t$.

Exercício 4. *Prove o sequente* $\vdash (p \to q \to r) \to (p \to q) \to p \to r$.

Exercício 5. *Prove o sequente* $\vdash (p \to q \to r) \to (q \to p \to r)$.

Exercício 6. *Prove o sequente* $\vdash (p \to r) \to p \to q \to r$.

Exercício 7. *Prove o sequente* $\vdash (p \to q) \to (p \to r) \to (q \to r \to t) \to p \to t$.

As regras para a negação são muito similares às regras da implicação, e isto não ocorre por acaso. De fato, uma negação é uma implicação particular porque $\neg\varphi$ é definida como $\varphi \to \bot$. Considerando este fato, a analogia com as regras da implicação é direta.

$$\frac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i) \qquad\qquad \frac{\Gamma_1 \vdash \neg\varphi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \bot}\ (\neg_e)$$

Veremos posteriormente que apenas com a negação e implicação podemos expressar todos os outros conectivos apresentados na gramática (2.1), que portanto é uma gramática redundante. No entanto, esta redundância é interessante porque nos permite expressar fórmulas complexas de forma compacta.

Exercício 8. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\varphi \vdash \neg\neg\varphi$.*

O sequente do exercício anterior ocorre com alta frequência em provas, e pelo seu caráter especial, ele será promovido ao *status* de regra derivada. Nesta transformação, os antecedentes do sequente, neste caso a fórmula φ, são generalizados como premissas da regra de inferência:

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi}\ (\neg\neg_i)$$

Talvez você esteja esperando agora a derivação da eliminação da dupla negação para se juntar a regra anterior, mas infelizmente isto não é possível neste momento porque o poder de expressividade que temos até agora com as regras $(\to_e)$, $(\to_\mathrm{i})$, $(\neg_\mathrm{i})$ e $(\neg_e)$ não é suficiente para provarmos a eliminação da dupla negação. Mas para deixá-lo intrigado provaremos o seguinte sequente: $\neg\neg\neg\varphi \vdash \neg\varphi$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}\,(\mathrm{Ax})}{\varphi \vdash \neg\neg\varphi}\,(\neg\neg_i) \qquad \dfrac{}{\neg\neg\neg\varphi \vdash \neg\neg\neg\varphi}\,(\mathrm{Ax})}{\neg\neg\neg\varphi, \varphi \vdash \bot}\,(\neg_e)}{\neg\neg\neg\varphi \vdash \neg\varphi}\,(\neg_i)$$

O sequente anterior é a prova da eliminação da dupla negação de uma **fórmula negada**, e isto faz toda a diferença. Voltaremos a falar da eliminação da dupla negação na seção sobre lógica proposicional clássica.

Exercício 9. *Sejam φ e ψ fórmulas da lógica proposicional. Prove o sequente $\varphi \to \psi \vdash (\neg\neg\varphi) \to (\neg\neg\psi)$*

Exercício 10. *Sejam φ e ψ fórmulas da lógica proposicional. Prove o sequente $\neg\neg(\varphi \to \psi) \vdash (\neg\neg\varphi) \to (\neg\neg\psi)$*

A *regra de introdução da conjunção*, denotada por $(\wedge_i)$, nos diz o que precisamos fazer para construir uma prova de um sequente que possui uma conjunção na conclusão, isto é, um sequente da forma $\Gamma \vdash \varphi_1 \wedge \varphi_2$, onde Γ é um conjunto finito de fórmulas da LP, e φ_1 e φ_2 são fórmulas da LP.

A regra $(\wedge_i)$ é dada pela seguinte regra de inferência:

$$\frac{\Gamma_1 \vdash \varphi_1 \qquad \Gamma_2 \vdash \varphi_2}{\Gamma_1, \Gamma_2 \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i) \tag{2.2}$$

ou seja, uma prova de $\Gamma \vdash \varphi_1 \wedge \varphi_2$ é construída a partir de uma prova de $\Gamma \vdash \varphi_1$ e de uma prova de $\Gamma \vdash \varphi_2$.

Existem duas regras de eliminação para a conjunção já que podemos extrair qualquer uma das componentes de uma conjunção:

$$\frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_1}\ (\wedge_{e_1}) \qquad\qquad \frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_2}\ (\wedge_{e_2})$$

Estas duas regras podem ser representadas de forma mais concisa da seguinte forma:

$$\frac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i \in \{1,2\}}}\ (\wedge_e) \tag{2.3}$$

Usaremos o nome $(\wedge_e)$ para designar a utilização da regra de eliminação da conjunção quando não quisermos especificar qual das regras $(\wedge_{e_1})$ ou $(\wedge_{e_2})$ foi utilizada.

Com as regras da conjunção já podemos fazer um exercício interessante: provar a comutatividade da conjunção, isto é, queremos construir uma prova para o sequente $\varphi \wedge \psi \vdash \psi \wedge \varphi$, onde φ e ψ são fórmulas quaisquer da LP. A construção da prova é feita inicialmente de baixo para cima com a aplicação da regra $(\wedge_i)$:

$$\frac{\dfrac{?}{\varphi \wedge \psi \vdash \psi} \qquad \dfrac{?}{\varphi \wedge \psi \vdash \varphi}}{\varphi \wedge \psi \vdash \psi \wedge \varphi}\ (\wedge_i)$$

Concluímos com a regra de eliminação da conjunção e o axioma:

$$\frac{\dfrac{\dfrac{}{\varphi \wedge \psi \vdash \varphi \wedge \psi}\ (\text{Ax})}{\varphi \wedge \psi \vdash \psi}\ (\wedge_e) \qquad \dfrac{\dfrac{}{\varphi \wedge \psi \vdash \varphi \wedge \psi}\ (\text{Ax})}{\varphi \wedge \psi \vdash \varphi}\ (\wedge_e)}{\varphi \wedge \psi \vdash \psi \wedge \varphi}\ (\wedge_i)$$

Exercício 11. *Prove que a conjunção é associativa, isto é, prove o sequente* $(\varphi \wedge \psi) \wedge \rho \vdash \varphi \wedge (\psi \wedge \rho)$*, onde* φ*,* ψ *e* ρ *são fórmulas quaisquer da lógica proposicional.*

Vejamos agora as regras para a disjunção. A *regra de introdução da disjunção* nos permite construir a prova de uma disjunção a partir da prova de qualquer uma das suas componentes:

$$\frac{\Gamma \vdash \varphi_1}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_{i_1}) \qquad\qquad \frac{\Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_{i_2})$$

Como no caso da regra de eliminação da conjunção podemos representar estas duas regras de forma mais compacta:

$$\frac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$$

A *regra de eliminação da disjunção* nos permite construir a prova de uma fórmula, digamos γ, a partir de uma disjunção. Para isto, precisamos de duas provas distintas de γ, cada uma assumindo uma das componentes da disjunção separadamente:

$$\frac{\Gamma_1 \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma_2, \varphi_1 \vdash \gamma \qquad \Gamma_3, \varphi_2 \vdash \gamma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \gamma}\ (\vee_{\mathrm{e}})$$

Assim, para que tenhamos uma prova de γ a partir das fórmulas em $\Gamma_1, \Gamma_2, \Gamma_3$ (sequente $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \gamma$) precisamos de uma prova de γ a partir de φ_1 e das fórmulas de Γ_2 (sequente $\Gamma_2, \varphi_1 \vdash \gamma$) e de outra prova de γ a partir de φ_2 e das fórmulas de Γ_3 (sequente $\Gamma_3, \varphi_2 \vdash \gamma$). Observe como os contextos mudam em cada um dos sequentes que compõem esta regra.

Exemplo 12. *Vamos mostrar que a disjunção é comutativa, ou seja, queremos construir uma prova para o sequente $\varphi \vee \psi \vdash \psi \vee \varphi$. A ideia aqui é utilizarmos a regra $(\vee_{\mathrm{e}})$. Para isto podemos instanciar Γ com o conjunto unitário contendo a fórmula $\varphi \vee \psi$. Em função da estrutura da regra $(\vee_{\mathrm{e}})$, precisamos construir duas provas distintas de $\psi \vee \varphi$: uma a partir de φ, e outra a partir de ψ. Podemos fazer isto com a ajuda da regra $(\vee_i)$:*

$$\dfrac{(\mathrm{Ax})\ \dfrac{}{\varphi \vee \psi \vdash \varphi \vee \psi} \qquad \dfrac{\dfrac{\dfrac{}{\varphi \vdash \varphi}\ (\mathrm{Ax})}{\varphi \vdash \psi \vee \varphi}\ (\vee_i) \qquad \dfrac{\dfrac{}{\psi \vdash \psi}\ (\mathrm{Ax})}{\psi \vdash \psi \vee \varphi}\ (\vee_i)}{\varphi \vee \psi \vdash \psi \vee \varphi}\ (\vee_{\mathrm{e}})}{}$$

Exercício 13. *Sejam φ, ψ e ρ fórmulas quaisquer da lógica proposicional. Prove que a disjunção é associativa, isto é, prove o sequente $(\varphi \vee \psi) \vee \rho \vdash \varphi \vee (\psi \vee \rho)$.*

A Tabela 2.1 apresenta as regras vistas até aqui. Estas regras formam a chamada *lógica proposicional minimal*.

Agora vamos resolver mais alguns exercícios na lógica minimal.

Exemplo 14. *Considere o sequente $\varphi \to \psi, \neg\psi \vdash \neg\varphi$. Como a fórmula do consequente é uma negação, vamos aplicar a regra de introdução da negação na construção de uma prova de baixo para cima, isto é, da raiz para as folhas da árvore:*

$$\frac{\dfrac{?}{\varphi \to \psi, \neg\psi, \varphi \vdash \bot}}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\ (\neg_{\mathrm{i}})$$

	Regras de introdução	Regras de eliminação
1	$\dfrac{\Gamma_1 \vdash \varphi_1 \qquad \Gamma_2 \vdash \varphi_2}{\Gamma_1, \Gamma_2 \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$
2	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\Gamma_1 \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma_2, \varphi_1 \vdash \gamma \qquad \Gamma_3, \varphi_2 \vdash \gamma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \gamma}\ (\vee_e)$
3	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\Gamma_1 \vdash \varphi \to \psi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \psi}\ (\to_e)$
4	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\Gamma_1 \vdash \neg\varphi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \bot}\ (\neg_e)$

Tabela 2.1: Regras da Lógica Minimal

Agora, precisamos construir uma prova do absurdo, e portanto podemos tentar utilizar a regra $(\neg_e)$*. Para isto precisamos escolher uma fórmula do contexto para fazer o papel de* φ *da regra 8 da Tabela 2.1. A princípio temos três opções:* $\varphi \to \psi$*,* $\neg\psi$ *e* φ*. A boa escolha neste caso é* $\neg\psi$ *porque podemos facilmente provar* ψ *a partir deste contexto:*

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varphi \to \psi \vdash \varphi \to \psi}\,(\text{Ax}) \quad \dfrac{}{\varphi \vdash \varphi}\,(\text{Ax})}{\varphi \to \psi, \varphi \vdash \psi}\,(\to_e) \qquad \dfrac{}{\neg\psi \vdash \neg\psi}\,(\text{Ax})}{\varphi \to \psi, \neg\psi, \varphi \vdash \bot}\,(\neg_e)}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\,(\neg_i)$$

Depois de concluída a prova é fácil entender o que queríamos dizer com boa escolha *acima: Uma boa escolha é um caminho que vai nos permitir concluir uma prova. Mas como fazer uma boa escolha? Isto depende do problema a ser resolvido. Em alguns casos pode ser simples, mas em outros, bastante complicado. O ponto importante a compreender é que existem caminhos possíveis distintos na construção de provas da lógica proposicional, e muito deste processo depende da nossa criatividade.*

O sequente que acabamos de provar ocorre com certa frequência em outras provas, assim como a regra derivada $(\neg\neg_i)$. As regras que são obtidas a partir das regras da Tabela 2.1 são chamadas de *regras derivadas*. Este é o caso da regra conhecida como *modus tollens* (MT) obtida a partir do sequente do exemplo anterior, onde cada antecedente é generalizado como uma premissa da regra:

$$\dfrac{\Gamma_1 \vdash \varphi \to \psi \qquad \Gamma_2 \vdash \neg\psi}{\Gamma_1, \Gamma_2 \vdash \neg\varphi}\ (\text{MT})$$

Exemplo 15. *Considere o sequente* $\varphi \to \psi \vdash \neg\psi \to \neg\varphi$*. Inicialmente, devemos observar que a fórmula que queremos provar é uma implicação, e portanto, o mais natural é tentar aplicar a regra* $(\to_i)$*, e em seguida aplicar* (MT) *(na construção de baixo para cima) para poder completar a prova:*

$$\dfrac{\dfrac{\dfrac{}{\varphi \to \psi \vdash \varphi \to \psi}\,(\text{Ax}) \qquad \dfrac{}{\neg\psi \vdash \neg\psi}\,(\text{Ax})}{\varphi \to \psi, \neg\psi \vdash \neg\varphi}\,(\text{MT})}{\varphi \to \psi \vdash \neg\psi \to \neg\varphi}\,(\to_i)$$

O sequente que acabamos de provar é outro caso que aparece com frequência em provas, e corresponde a uma regra conhecida como *contrapositiva*:

$$\frac{\Gamma \vdash \varphi \to \psi}{\Gamma \vdash \neg\psi \to \neg\varphi}\ (\text{CP})$$

Exercício 16. *Sejam φ e ψ fórmulas da lógica proposicional. Prove o sequente $\varphi \to \neg\psi \vdash \psi \to \neg\varphi$*

É natural pensarmos que se é possível provar φ a partir do conjunto Γ então também é possível provar φ a partir do conjunto $\Gamma \cup \{\psi\}$. Formalmente esta ideia corresponde a regra de enfraquecimento (*weakening*):

$$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi}\ (w)$$

O teorema a seguir nos mostra que a regra de enfraquecimento pode ser derivada na lógica proposicional minimal:

Teorema 17. *Se $\Gamma \vdash \varphi$ então $\Gamma, \psi \vdash \varphi$ na lógica proposicional minimal, quaisquer que sejam o conjunto Γ, e as fórmulas φ e ψ.*

A prova deste lema é feita por indução na estrutura da derivação $\Gamma \vdash \varphi$. Retornaremos a ela no final da Seção 3.1.

Exercício 18. *Prove o sequente $\vdash ((((p \to q) \to p) \to p) \to q) \to q$.*

Considerando que o sistema dedutivo da lógica proposicional minimal apresentado nos permite derivar a regra de enfraquecimento, podemos reestruturar as regras de forma a não mais precisar dividir os contextos nas regras com bifurcação (de baixo para cima). O axioma pode então ser escrito da seguinte forma:

$$\frac{}{\Gamma, \varphi \vdash \varphi}\ (Ax)$$

E as regras de introdução e eliminação dos conectivos lógicos podem ser escritas como na Tabela 2.2.

Este é um bom momento para simplificarmos a notação que estamos usando, e tentaremos deixar clara a vantagem de nossa abordagem com a mudança de notação neste momento. Vamos retomar o Exercício 11 que consiste em provar que a conjunção é um conectivo que satisfaz a propriedade associativa. Neste ponto acreditamos que você já resolveu este exercício. Em caso negativo, resolva o exercício antes de prosseguir. Em seguida, compare sua solução com a que apresentamos a seguir, ok? Tentar resolver os exercícios antes de olhar qualquer solução é um passo muito importante para a sua evolução nos estudos de lógica. Considere a prova a seguir:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\phi \wedge \psi) \wedge \varphi \vdash (\phi \wedge \psi) \wedge \varphi}\,(\text{Ax})}{(\phi \wedge \psi) \wedge \varphi \vdash \phi \wedge \psi}\,(\wedge_e)}{(\phi \wedge \psi) \wedge \varphi \vdash \phi}\,(\wedge_e) \qquad \dfrac{\dfrac{\dfrac{}{(\phi \wedge \psi) \wedge \varphi \vdash (\phi \wedge \psi) \wedge \varphi}\,(\text{Ax})}{(\phi \wedge \psi) \wedge \varphi \vdash \phi \wedge \psi}\,(\wedge_e)}{(\phi \wedge \psi) \wedge \varphi \vdash \psi}\,(\wedge_e) \qquad \dfrac{\dfrac{}{(\phi \wedge \psi) \wedge \varphi \vdash (\phi \wedge \psi) \wedge \varphi}\,(\text{Ax})}{(\phi \wedge \psi) \wedge \varphi \vdash \varphi}\,(\wedge_e)}{(\phi \wedge \psi) \wedge \varphi \vdash (\psi \wedge \varphi)}\,(\wedge_i)}{(\phi \wedge \psi) \wedge \varphi \vdash \phi \wedge (\psi \wedge \varphi)}\,(\wedge_i)$$

	Regras de introdução	Regras de eliminação
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$
2	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma, \varphi_1 \vdash \gamma \qquad \Gamma, \varphi_2 \vdash \gamma}{\Gamma \vdash \gamma}\ (\vee_e)$
3	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$
4	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$

Tabela 2.2: Regras da Lógica Minimal

Observe que o contexto, isto é, o antecedente de cada um dos sequentes desta prova é o mesmo. De fato, o contexto em cada nó da árvore acima é o conjunto unitário contendo a fórmula $(\phi \wedge \psi) \wedge \varphi$. Como o que muda ao longo da prova é o consequente dos sequentes, é natural considerar que o foco, ou que a parte principal, desta prova é o consequente de cada sequente. Sabendo com qual contexto estamos trabalhando, podemos removê-lo da prova deixando-a mais limpa e compacta. Veja como fica a prova sem os contextos:

$$\dfrac{\dfrac{\dfrac{(\phi \wedge \psi) \wedge \varphi}{(\phi \wedge \psi)}\,(\wedge_e)}{\phi}\,(\wedge_e) \qquad \dfrac{\dfrac{\dfrac{(\phi \wedge \psi) \wedge \varphi}{\phi \wedge \psi}\,(\wedge_e)}{\psi} \qquad \dfrac{(\phi \wedge \psi) \wedge \varphi}{\varphi}\,(\wedge_e)}{(\psi \wedge \varphi)}\,(\wedge_i)}{\phi \wedge (\psi \wedge \varphi)}\,(\wedge_i)$$

Será que é possível sempre remover os contextos das provas de uma forma sistemática? Sim, e neste exemplo em particular, a situação é simples porque o contexto é o mesmo em toda a prova, mas este não será o caso em geral. Ainda considerando o exemplo anterior, se não soubéssemos qual o contexto que foi apagado, seria possível descobrí-lo? Sim, esta informação vem das folhas da árvore, que são as hipóteses do problema. Na notação com o contexto explícito, as folhas da árvore têm que ser axiomas, e portanto o contexto de todas as folhas é a fórmula $(\phi \wedge \psi) \wedge \varphi$ já que todas as folhas são iguais. Agora podemos consultar a Tabela 2.1 e observar que os contextos das regras $(\wedge_e)$ e $(\wedge_i)$ são os mesmos antes e depois da aplicação destas regras. Portanto, o contexto em cada nó da árvore é formado pelo conjunto unitário contendo a fórmula $(\phi \wedge \psi) \wedge \varphi$. Outro detalhe importante é que as folhas desta nova árvore não correspondem mais à regra (Ax), e portanto as folhas têm que ser fórmulas pertencentes ao contexto. As apresentações do sistema de dedução natural normalmente utilizam contextos implícitos [19, 13, 38, 4].

No Exemplo 14 construímos uma prova do sequente $\varphi \to \psi, \neg\psi \vdash \neg\varphi$ com contextos explícitos, a versão com contextos implícitos é dada pela seguinte árvore de derivação:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\,(\to_e) \qquad \neg\psi}{\bot}\,(\neg_e)}{\neg\varphi}\,(\neg_i)$$

Note que esta árvore possui três folhas, cada uma contendo uma fórmula distinta. O contexto inicial é o conjunto contendo todas as fórmulas que aparecem nas folhas, ou seja, nosso contexto inicial é o conjunto $\{\varphi \to \psi, \neg\psi, \varphi\}$. As regras $(\to_e)$ e $(\neg_e)$ preservam o contexto, e portanto o contexto da linha 3 é o conjunto $\{\varphi \to \psi, \neg\psi, \varphi\}$. De acordo com a Tabela 2.1, a regra $(\neg_i)$ adiciona uma fórmula ao contexto (leitura de baixo para cima), ou remove uma fórmula do contexto, se a leitura for feita de cima para baixo. Neste caso, a fórmula removida é φ, e portanto o contexto da fórmula $\neg\varphi$ (raiz da árvore) é o conjunto $\{\varphi \to \psi, \neg\psi\}$ como esperado. Na notação sem contexto,

as fórmulas que são removidas do contexto ao longo da prova, como a fórmula φ deste exemplo são colocadas entre colchetes para enfatizar que são hipóteses temporárias que em determinado momento serão removidas do contexto:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad [\varphi]}{\psi} (\to_e) \qquad \neg\psi}{\bot} (\neg_e)}{\neg\varphi} (\neg_{\mathrm{i}})$$

e agora fica claro que a fórmula φ não faz parte do contexto original do sequente a ser provado. Note que os colchetes são colocados **apenas nas folhas** que contêm fórmulas que não fazem parte do contexto dado pelo problema. Adicionalmente, como o contexto da raiz tem que ser o contexto dado pelo problema, caso contrário a prova não é uma prova do problema proposto, precisamos de um mecanismo para nos informar quando as fórmulas marcadas com os colchetes são **removidas** (ou **descartadas**) do contexto. No exemplo acima, isto ocorre ao aplicarmos a regra $(\neg_{\mathrm{i}})$. Então, utilizaremos uma letra para registrar este fato:

$$\dfrac{\dfrac{\dfrac{\varphi \to \psi \qquad [\varphi]^u}{\psi} (\to_e) \qquad \neg\psi}{\bot} (\neg_e)}{\neg\varphi} (\neg_i)\ u$$

Agora sabemos em que momento a fórmula φ foi introduzida (folha $[\varphi]^u$), e em que momento foi descartada (regra $(\neg_i)$ u) na árvore de derivação.

A seguir, veremos exemplos mais complexos onde fórmulas idênticas podem exigir marcas distintas, mas antes disto compare as regras de dedução natural para a lógica proposicional minimal com o contexto explícito e com o contexto implícito na Tabela 2.3, e veja como o mecanismo de descarte simula a mudança de contexto antes e depois de uma aplicação das regras $(\vee_{\mathrm{e}})$, $(\to_{\mathrm{i}})$ e $(\neg_{\mathrm{i}})$. Como a notação com contexto implícito é mais compacta, a partir deste momento não utilizaremos mais contextos explícitos. A intenção de iniciar esta apresentação utilizando contextos explícitos foi de permitir uma explicação mais fácil e natural para o descarte de hipóteses, que sempre gerou muitas dúvidas entre os alunos. Por exemplo, se a razão do descarte não está clara, é comum aparecerem árvores de derivação com descarte de hipóteses feito em regras como $(\wedge_i)$, $(\wedge_e)$, $(\vee_i)$, $(\to_e)$ e $(\neg_e)$. Se você acha que está tudo bem uma árvore de derivação conter descarte de hipóteses nas regras citadas na frase anterior, volte para o início deste capítulo e reinicie o estudo do sistema de dedução natural antes de prosseguir :-)

Exemplo 19. *Neste exemplo, veremos que é possível fazer a introdução de uma implicação sem precisar descartar uma hipótese, se tivermos uma prova do consequente da implicação que queremos construir. Ou seja, se temos uma prova de ψ então podemos construir uma prova de $\varphi \to \psi$, qualquer que seja a fórmula φ. Em outras palavras, queremos construir uma prova para o sequente $\psi \vdash \varphi \to \psi$. A ideia da prova neste caso é simples. Vamos assumir uma prova de φ, e transformá-la em uma prova de ψ que já temos como hipótese. Para isto basta introduzirmos e em seguida eliminarmos uma conjunção contendo ψ:*

$$\dfrac{\dfrac{\dfrac{[\varphi]^u \qquad \psi}{\varphi \wedge \psi} (\wedge_i)}{\psi} (\wedge_e)}{\varphi \to \psi} (\to_i)\ u$$

Como este raciocínio aparece com frequência nas provas, vamos colocá-lo como uma regra derivada:

$$\dfrac{\psi}{\varphi \to \psi} (\to_i)\ \emptyset$$

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma_1 \vdash \varphi_1 \qquad \Gamma_2 \vdash \varphi_2}{\Gamma_1, \Gamma_2 \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma_1 \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma_2, \varphi_1 \vdash \gamma \qquad \Gamma_3, \varphi_2 \vdash \gamma}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{matrix}[\varphi_1]^u \\ \vdots \\ \gamma\end{matrix} \qquad \begin{matrix}[\varphi_2]^v \\ \vdots \\ \gamma\end{matrix}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \psi\end{matrix}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma_1 \vdash \varphi \to \psi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{matrix}[\varphi]^u \\ \vdots \\ \bot\end{matrix}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma_1 \vdash \neg\varphi \qquad \Gamma_2 \vdash \varphi}{\Gamma_1, \Gamma_2 \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$

Tabela 2.3: Regras da Lógica Proposicional Minimal

$\dfrac{\Gamma \vdash \varphi}{\Gamma \vdash \neg\neg\varphi}\ (\neg\neg_i)$	$\dfrac{\Gamma_1 \vdash \varphi \to \psi \qquad \Gamma_2 \vdash \neg\psi}{\Gamma_1, \Gamma_2 \vdash \neg\varphi}$ (MT)	$\dfrac{\psi}{\varphi \to \psi}\ (\to_i)\ \emptyset$	$\dfrac{\Gamma \vdash \varphi \to \psi}{\Gamma \vdash \neg\psi \to \neg\varphi}$ (CP)

Tabela 2.4: Regras derivadas da Lógica Proposicional Minimal

Exercício 20. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $\varphi, \neg\varphi \vdash \neg\gamma$ na lógica proposicional minimal.*

Exercício 21. *Seja φ uma fórmula da lógica proposicional. Construa uma prova para o sequente $\neg\neg\neg\varphi \vdash \neg\varphi$ na lógica proposicional minimal.*

Exercício 22. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $\neg(\varphi \vee \psi) \vdash (\neg\varphi) \wedge (\neg\psi)$ na lógica proposicional minimal.*

Exercício 23. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\varphi) \wedge (\neg\psi) \vdash \neg(\varphi \vee \psi)$ na lógica proposicional minimal.*

Exercício 24. *Sejam φ, ψ e δ fórmulas da lógica proposicional. Construa uma prova para o sequente $\varphi \to \psi \vdash (\delta \vee \varphi) \to (\delta \vee \psi)$ na lógica proposicional minimal.*

Exercício 25. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $\varphi \to \psi \vdash \neg(\varphi \wedge \neg\psi)$ na lógica proposicional minimal.*

Exercício 26. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $\varphi \wedge \psi \vdash \neg(\neg\varphi \vee \neg\psi)$ na lógica proposicional minimal.*

Exercício 27. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para os sequentes $\neg(\varphi \vee \gamma) \vdash (\neg\varphi) \wedge (\neg\gamma)$ e $(\neg\varphi) \wedge (\neg\gamma) \vdash \neg(\varphi \vee \gamma)$ na lógica proposicional minimal.*

Exercício 28. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\varphi) \vee (\neg\gamma) \vdash \neg(\varphi \wedge \gamma)$ na lógica proposicional minimal.*

Exercício 29. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $\neg\neg(\varphi \wedge \gamma) \vdash (\neg\neg\varphi) \wedge (\neg\neg\gamma)$ na lógica proposicional minimal.*

Exercício 30. *Sejam φ e γ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\neg\varphi) \wedge (\neg\neg\gamma) \vdash \neg\neg(\varphi \wedge \gamma)$ na lógica proposicional minimal.*

Exercício 31. *Sejam φ, ψ e γ fórmulas da lógica proposicional. Prove o sequente $\varphi \vee (\psi \wedge \gamma) \vdash (\varphi \vee \psi) \wedge (\varphi \vee \gamma)$.*

Exercício 32. *Sejam φ, ψ e γ fórmulas da lógica proposicional. Prove o sequente $(\varphi \vee \psi) \wedge (\varphi \vee \gamma) \vdash \varphi \vee (\psi \wedge \gamma)$.*

Exercício 33. *Sejam φ, ψ e γ fórmulas da lógica proposicional. Prove o sequente $\varphi \wedge (\psi \vee \gamma) \vdash (\varphi \vee \psi) \wedge (\varphi \vee \gamma)$.*

Exercício 34. *Sejam φ, ψ e γ fórmulas da lógica proposicional. Prove o sequente $(\varphi \vee \psi) \wedge (\varphi \vee \gamma) \vdash \varphi \wedge (\psi \vee \gamma)$.*

Exercício 35. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\vdash \neg\neg(\varphi \vee \neg\varphi)$ na lógica proposicional minimal.*

Exercício 36. *Seja φ uma fórmula da lógica proposicional. Prove o sequente $\vdash \neg(\varphi \wedge \neg\varphi)$ na lógica proposicional minimal.*

2.2 O Assistente de provas Coq

1. Notação

   Utilizaremos algumas notações para facilitar a identificação de diferentes contextos, principalmente no que se refere ao assistente de provas Coq[37]. Os códigos do Coq são escritos em `verbatim`, mas uma sessão típica do Coq possui três janelas:

   A janela da esquerda é onde escrevemos as definições, lemas e as provas propriamente ditas; a do canto superior direito nos mostra o *status* atual da prova; e a do canto inferior direito nos mostra mensagens do sistema. Os textos da janela da esquerda aparecem em `verbatim`, enquanto que os da janela superior direita, isto é, o *status* das provas aparecem em

```
verbatim e dentro de uma caixa para facilitar a identificação.
```

   Assumimos que o Coq está instalado no seu computador. Estas notas estão sendo escritas usando a versão 8.15.1 do Coq.

2. A lógica proposicional minimal no assistente de provas Coq

   Apresentaremos as regras do sistema de dedução natural em paralelo com o assistente de provas Coq. Esta é uma forma de aprendermos a utilizar o sistema de uma forma progressiva e suave. O Coq implementa uma lógica de ordem superior baseado em dedução natural. Isto quer dizer que será possível fazer uma analogia entre o sistema de dedução natural que apresentamos aqui e o Coq, mas como veremos, esta analogia não é feita via uma correspondência direta entre as regras em dedução natural e as *regras* do Coq que são chamadas de *táticas*. De fato, as táticas são desenvolvidas para realizarem vários passos de prova de uma vez porque isto facilita o processo de construção de provas em sistemas mais complexos. Iniciaremos com a prova do exemplo anterior, que nos permitirá construir a prova de uma conjunção em Coq. Para simularmos a regra de introdução da conjunção vamos declarar duas variáveis `phi` e `psi`, e em seguida, criaremos uma seção que vai delimitar o escopo da prova. Chamaremos esta seção de `landi`, e então declaramos as hipóteses e o lema propriamente dito dentro da seção:

```
Variables phi psi: Prop.
```

```
Section landi.
 Hypothesis H1: phi.
 Hypothesis H2: psi.
 Lemma landi: phi /\ psi.

End landi.
```

Esta é uma forma de declarar o sequente `phi, psi` $\vdash$ `phi` $\wedge$ `psi` no Coq. De fato, na janela de prova temos o sequente como esperado:

```
  H1 : phi
  H2 : psi
  ============================
  phi /\ psi
```

Portanto uma prova deste sequente é o que vai corresponder a uma aplicação da regra $(\wedge_i)$. A prova é construída entre as palavras reservadas `Proof` (que denota o início da prova) e `Qed` (que indica que a prova foi finalizada). Utilizamos a tática `split` para dividirmos a prova da conjunção em subprovas das suas componentes (já que a construção em Coq é sempre feita de baixo para cima, isto é da raiz para as folhas da árvore) que por sua vez são hipóteses, e a prova de algo que já faz parte do conjunto de hipóteses pode ser concluída com a tática `assumption`:

```
Variables phi psi: Prop.

Section landi.
 Hypothesis H1: phi.
 Hypothesis H2: psi.
 Lemma landi: phi /\ psi.
 Proof.
   split.
   - assumption.
   - assumption.
 Qed.
End landi.
```

Logo, a regra $(\wedge_i)$ está relacionada com a tática `split` e o axioma com a tática `assumption`. Uma maneira de ver isto de forma mais explícita consiste em comparar a árvore de prova do sequente `phi, psi` $\vdash$ `phi` $\wedge$ `psi`:

$$\dfrac{\dfrac{}{\texttt{phi, psi} \vdash \texttt{phi}}\,(\text{Ax}) \qquad \dfrac{}{\texttt{phi, psi} \vdash \texttt{psi}}\,(\text{Ax})}{\texttt{phi, psi} \vdash \texttt{phi} \wedge \texttt{psi}}\,(\wedge_i) \qquad \dfrac{\dfrac{}{\texttt{phi, psi} \vdash \texttt{phi}}\,\texttt{assumption} \qquad \dfrac{}{\texttt{phi, psi} \vdash \texttt{psi}}\,\texttt{assumption}}{\texttt{phi, psi} \vdash \texttt{phi} \wedge \texttt{psi}}\,\texttt{split}$$

Neste caso, temos uma correspondência direta entre uma regra do sistema de dedução natural e o Coq, mas este não é sempre o caso. .

Como vimos, as provas em Coq são construídas de baixo para cima, isto é, partimos da raiz da árvore de dedução que é o sequente que queremos provar, e subimos até as folhas que são os axiomas. Em provas simples conseguimos seguir este caminho sem dificuldades, mas em provas mais complexas precisamos ter mais flexibilidade. Em papel e lápis, as provas podem ser construídas tanto de baixo para cima (da raiz para as folhas) quanto de cima para baixo, e na prática usamos

as duas estratégias porque dependendo do contexto, pode ser melhor uma estratégia ou outra. Veremos diversos exemplos para facilitar a compreensão desta ideia, e em Coq o mesmo acontece: a cada instante da construção de uma prova podemos tanto dar um passo de baixo para cima aplicando uma tática que altera o objetivo (raiz da árvore que estamos construindo) quanto uma tática que altera uma hipótese que corresponde a um passo de cima para baixo na construção da prova. Vejamos um exemplo de tática do Coq que altera uma hipótese. Para isto, considere o sequente que tem uma conjunção como hipótese, e uma das componentes desta conjunção como conclusão: `phi ∧ psi ⊢ phi`:

```
Variables phi psi: Prop.

Section landel.
 Hypothesis H: phi /\ psi.
 Lemma landel: phi.
 Proof.
```

Neste momento a janela de prova tem a seguinte forma:

```
1 subgoal (ID 1)

 H : phi /\ psi
 ============================
 phi
```

Podemos usar tática `inversion H` para decompor a hipótese deste sequente, e obtemos:

```
1 subgoal (ID 1)

 H : phi /\ psi
 H0 : phi
 H1 : psi
 ============================
 phi
```

E a prova pode ser concluída com a tática `assumption`. Não entraremos neste momento nos detalhes técnicos da tática `inversion`, mas grosso modo, ela gera as condições necessárias para a construção da hipótese onde ela está sendo aplicada. Para mais detalhes recomendamos que o leitor consulte o manual do usuário do Coq[1]. Existem outras táticas que podemos usar no lugar de `inversion` no exemplo anterior, como `destruct` e `elim`, mas elas não serão abordadas agora. No entanto, táticas diferentes podem ser usadas para construir provas diferentes de um mesmo sequente, assim como ocorre em papel e lápis.

Observe que no exercício anterior, solicitamos primeiro uma solução em papel e lápis para, somente depois, solicitar a prova no Coq. Este é um detalhe importante porque os assistentes de prova não são ferramentas para nos ajudar a construir provas, mas sim para verificar provas. A ideia é utilizar os assistentes de prova para mecanizarmos uma prova que já tenha sido feito em papel e lápis, ou uma prova que temos na cabeça (mesmo que apenas um esboço). Iniciar uma prova em um assistente de provas sem saber inicialmente que caminho seguir, tentando a sorte, em geral não é uma boa ideia.

[1] `https://coq.inria.fr/distrib/current/refman/`

Exercício 37. *Utilize os seus conhecimentos de Coq para provar que a conjunção é comutativa e associativa.*

Em Coq, o sequente deste exemplo pode ser escrito declarando duas variáveis `phi` e `psi`, e a hipótese `H: phi \/ psi.`:

```
Variables phi psi: Prop.

Section or_comm.
Hypothesis H: phi \/ psi.
Lemma or_comm: psi \/ phi.
Proof.

End or_comm.
```

Temos então o seguinte sequente para ser provado:

```
1 subgoal (ID 1)

 H : phi \/ psi
 ============================
 psi \/ phi
```

A tática `destruct H` vai dividir a prova em duas subprovas. A primeira nos pede para provar `psi \/ phi` a partir de `phi`:

```
 H : phi \/ psi
 H0 : phi
 ============================
 psi \/ phi
```

que consiste em usar a tática `right` seguida de `assumption`, enquanto que na segunda subprova precisamos provar `psi \/ phi` a partir de `psi`:

```
 H : phi \/ psi
 H0 : psi
 ============================
 psi \/ phi
```

que consiste em uma aplicação da tática `left` seguida de `assumption`.

Em Coq, esta regra é simulada por meio da tática `intro`.

```
 ============================
 phi -> psi
```

A tática `intro` vai mover o antecedente `phi` da implicação para as hipóteses:

```
H : phi
  ============================
  psi
```

Portanto a tática `intro` corresponde a uma aplicação da regra de introdução da implicação. A regra de *eliminação da implicação* é a mais famosa das regras que veremos, a ponto de possuir um nome próprio, a saber *modus ponens*. Esta regra nos diz como podemos usar a prova de uma implicação juntamente com uma prova do antecedente desta implicação:

$$\frac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$$

Quando lida de baixo para cima, esta regra corresponde a uma aplicação da regra do corte, que em Coq corresponde à tática `cut`. Assim, aplicando a tática `cut phi`, a prova se divide em dois ramos, ou em dois subobjetivos.

```
2 subgoals (ID 2)

 ============================
  phi -> psi

subgoal 2 (ID 3) is:
 phi
```

Exemplo 38. *Considere o sequente* $\Gamma, \varphi \to \psi, \varphi \vdash \psi$. *Podemos prová-lo usando a regra* $(\to_e)$ *da seguinte forma:*

$$\frac{\dfrac{}{\Gamma, \varphi \to \psi, \varphi \vdash \varphi \to \psi}\ (\text{Ax}) \qquad \dfrac{}{\Gamma, \varphi \to \psi, \varphi \vdash \varphi}\ (\text{Ax})}{\Gamma, \varphi \to \psi, \varphi \vdash \psi}\ (\to_e)$$

Agora faremos a prova acima em Coq considerando o conjunto Γ *como sendo o conjunto vazio. A declaração do sequente é feita como a seguir:*

```
Variables phi psi: Prop.

Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e: psi.
 Proof.
```

e o ambiente de prova corresponde ao sequente abaixo:

```
 H1 : phi -> psi
 H2 : phi
 ============================
 psi
```

Agora podemos aplicar a tática `cut phi` *como explicado acima, e concluir a prova com* `assumption`.

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e: psi.
 Proof.
   cut phi.
   - assumption.
   - assumption.
 Qed.
End imp_e.
```

Um outro caminho possível para provar este sequente é por meio da tática `apply H1` *seguida de* `assumption`. *Neste caso, o consequente da hipótese* `H1` *é confrontado com o objetivo a ser provado. Como eles coincidem, o novo objetivo gerado passa a ser* `phi`, *ou seja, o antecedente da hipótese* `H1`.

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e2: psi.
 Proof.
  apply H1.
  assumption.
Qed.
End imp_e.
```

Por fim, podemos também usar a tática `apply` *nas hipóteses. Neste caso, o antecedente da hipótese* `H1` *é confrontado com a fórmula* `phi` *da hipótese* `H2`. *Como estas fórmulas coincidem, a hipótese* `H2` *é convertida na fórmula* `psi` *e podemos concluir a prova com* `assumption`.

```
Section imp_e.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: phi.
 Lemma imp_e3: psi.
 Proof.
  apply H1 in H2.
  assumption.
 Qed.
End imp_e.
```

Exercício 39. *O símbolo da negação em Coq é* $\sim$. *Sabendo disto, refaça a prova acima no Coq.*

Exercício 40. *Prove MT e CP no Coq.*

Descarte de hipóteses: Observe como o Coq faz este trabalho de modificar o contexto da mesma forma que acabamos de descrever:

```
Variables phi psi: Prop.

Section mt.
```

```
Hypothesis H1: phi -> psi.
Hypothesis H2: ~psi.
Lemma mt: ~phi.
Proof.
```

Ao iniciarmos a prova do lema `mt`, temos a seguinte configuração:

```
  H1 : phi -> psi
  H2 : ~ psi
  ============================
  ~ phi
```

e aplicando a tática `intro`, a fórmula `phi` é introduzida no contexto com a marca `H`:

```
   H1 : phi -> psi
   H2 : ~ psi
   H : phi
   ============================
   False
```

Note que a marca `H` foi criada automaticamente pelo Coq, mas você pode colocar outra marca informando-a como parâmetro da tática `intro`, como por exemplo, `intro u`:

```
  H1 : phi -> psi
  H2 : ~ psi
  u : phi
  ============================
  False
```

Esta prova corresponde ao Exercício 39, cuja solução é dada a seguir:

```
Variables phi psi: Prop.

Section mt.
 Hypothesis H1: phi -> psi.
 Hypothesis H2: ~psi.
 Lemma mt: ~phi.
 Proof.
   intro u.
   apply H2.
   apply H1.
   assumption.
 Qed.

End mt.
```

Exercício 41. *Prove a introdução da implicação com descarte vazio no Coq.*

Os exercícios anteriores incluem diversos resultados importantes que podem ser provados na lógica proposicional minimal, e por isto, é importante que você resolva todos eles em papel e lápis e posteriormente no Coq para verificar se sua solução está correta.

Exercício 42. *Refaça os exercícios anteriores no Coq.*

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma', \varphi_1 \vdash \gamma \qquad \Gamma'', \varphi_2 \vdash \gamma}{\Gamma, \Gamma', \Gamma'' \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot_e)$	$\dfrac{\bot}{\varphi}\ (\bot_e)$

Tabela 2.5: Regras da Lógica Intuicionista

2.3 A Lógica Proposicional Intuicionista

Agora vamos estender a lógica proposicional minimal com uma nova regra chamada de *regra da explosão* ou *regra da eliminação do absurdo intuicionista*. Esta regra nos permite concluir qualquer fórmula a partir do absurdo:

$$\frac{\bot}{\varphi}\ (\bot_e)$$

A lógica obtida adicionando-se a regra da explosão à lógica proposicional minimal é denominada *lógica proposicional intuicionista*. Observe que a lógica proposicional minimal possui uma versão mais fraca de regra de explosão. De fato, podemos na lógica proposicional minimal concluir qualquer fórmula negada a partir do absurdo (veja o Exercício 20). A lógica proposicional intuicionista é conhecida por corresponder à noção de lógica construtiva que é particularmente interessante para a Computação. De forma simplificada, a lógica proposicional intuicionista pode ser vista como a lógica que rejeita a lei do terceiro excluído, ou seja, nesta lógica o sequente $\vdash \varphi \vee \neg\varphi$ não tem prova, quando φ é uma fórmula arbitrária.

2.3.

Vejamos um exemplo de prova na lógica proposicional intuicionista:

Exemplo 43. *Considere o seguinte sequente $\neg\varphi \vee \gamma \vdash \varphi \to \gamma$. Iniciando esta prova de baixo para cima, isto é, partindo do consequente, podemos aplicar a regra de introdução da implicação:*

$$\neg\varphi \vee \gamma \qquad [\varphi]^u$$

$$?$$

$$\frac{\gamma}{\varphi \to \gamma}\ (\to_i)\ u$$

Agora precisamos construir uma prova de γ tendo as fórmulas $\neg\varphi \vee \gamma$ e $[\varphi]^u$ como contexto. Uma ideia possível é usar a regra de eliminação da disjunção porque com o lado esquerdo, isto é, com $\neg\varphi$ e com $[\varphi]^u$ temos o absurdo, e com a regra da explosão podemos concluir γ como queríamos. O lado direito da disjunção já é igual a γ, e assim concluímos a prova:

$$\dfrac{\neg\varphi \vee \gamma \qquad \dfrac{\dfrac{[\neg\varphi]^v \qquad [\varphi]^u}{\bot}\ (\neg_e)}{\gamma}\ (\bot_e) \qquad [\gamma]^w}{\gamma}\ (\vee_e)\ v, w}{\varphi \to \gamma}\ (\to_i)\ u$$

Agora vamos refazer esta prova no Coq. Precisamos declarar duas variáveis, digamos `phi` e `psi`, e a hipótese `(~phi)\/psi`:

```
Variables phi psi: Prop.

Section or_to_imp.
 Hypothesis H: (~phi) \/ psi.
 Lemma or_to_imp: phi -> psi.
 Proof.
```

Neste momento estamos com a seguinte janela de prova:

```
  H : ~ phi \/ psi
  ============================
  phi -> psi
```

Reproduzindo a prova anterior (de baixo para cima), devemos iniciar com a tática `intro` que corresponde à regra $(\to_i)$, para em seguida dividirmos a prova em função da disjunção na hipótese `H` com a tática `destruct H`. O primeiro subcaso consiste em construir uma prova de `psi` tendo `phi` e `~phi` no contexto. Neste momento podemos utilizar a regra da explosão por meio da tática `contradiction`. O outro ramo é trivial:

```
Variables phi psi: Prop.

Section or_to_imp.
 Hypothesis H: (~phi) \/ psi.
 Lemma or_to_imp: phi -> psi.
 Proof.
   intro H'.
   destruct H.
   - contradiction.
   - assumption.
 Qed.
End or_to_imp.
```

Exercício 44. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $(\neg\neg\varphi) \to (\neg\neg\psi) \vdash \neg\neg(\varphi \to \psi)$ na lógica proposicional intuicionista.*

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma', \varphi_1 \vdash \gamma \qquad \Gamma'', \varphi_2 \vdash \gamma}{\Gamma, \Gamma', \Gamma'' \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_i)$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash \varphi}\ (\bot_e)$	$\dfrac{\bot}{\varphi}\ (\bot_e)$
10	$\dfrac{}{\vdash \varphi \vee \neg\varphi}\ (\text{LEM})$	$\dfrac{}{\varphi \vee \neg\varphi}\ (\text{LEM})$

Tabela 2.6: Regras da Lógica Clássica

Comparando o exercício anterior com o Exercício 10, podemos concluir que as fórmulas $(\neg\neg\varphi) \to (\neg\neg\psi)$ e $\neg\neg(\varphi \to \psi)$ podem ser provadas uma a partir da outra. Isto nos dá uma noção de equivalência que representaremos por $(\neg\neg\varphi) \to (\neg\neg\psi) \dashv\vdash \neg\neg(\varphi \to \psi)$. Podemos ainda escrever $(\neg\neg\varphi) \to (\neg\neg\psi) \dashv\vdash_i \neg\neg(\varphi \to \psi)$ para enfatizar que esta equivalência se dá na lógica intuicionista.

Exercício 45. *Seja φ uma fórmula da lógica proposicional. Construa uma prova para o sequente $\vdash \neg\neg(\neg\neg\varphi \to \varphi)$ na lógica proposicional intuicionista.*

Exercício 46. *Sejam φ e ψ fórmulas da lógica proposicional. Construa uma prova para o sequente $\vdash \neg\neg(((\varphi \to \psi) \to \varphi) \to \varphi)$ na lógica proposicional intuicionista.*

2.4 A Lógica Proposicional Clássica

Vamos caminhar na direção de mais uma extensão, agora da lógica intuicionista para a lógica clássica. Iniciamos com a lógica proposicional minimal, depois a estendemos para a lógica proposicional intuicionista, e agora vamos estendê-la com a lei do terceiro excluído, obtendo assim a lógica proposicional clássica. Na Tabela 2.6 apresentamos também as regras com contexto explícito para que tenhamos sempre em mente como os contextos mudam de acordo com a aplicação das regras.

Exemplo 47. *Neste exemplo, vamos construir uma prova de uma regra conhecida como* prova por contradição *(PBC). A ideia desta regra é negar o que se quer provar, e então gerar uma contradição. O sequente a ser provado é o seguinte* $(\neg\varphi) \to \bot \vdash \varphi$. *Veja que queremos provar* φ, *e para isto estamos assumindo que a negação de* φ *nos leva a uma contradição. Vamos então tomar uma instância da* (LEM), *e provar* φ *via a eliminação da disjunção:*

$$\dfrac{\text{(LEM)}\ \dfrac{}{\varphi \vee \neg\varphi} \qquad [\varphi]^u \qquad \dfrac{\dfrac{(\neg\varphi) \to \bot \qquad [\neg\varphi]^v}{\bot}\,(\to_e)}{\varphi}\,(\bot_e)}{\varphi}\,(\vee_e)\ u, v$$

A regra de prova por contradição é dada a seguir. Observe como o contexto muda por conta do descarte de hipóteses:

Contexto explícito	*Contexto implícito*
$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ (PBC)	$\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \dfrac{\bot}{\varphi}\ (PBC)\ u\end{array}$

Agora vamos construir esta prova em Coq, mas precisamos de alguns cuidados porque a lógica implementada no Coq é construtiva, e portanto não temos táticas que correspondam a uma aplicação da lei do terceiro excluído. Neste caso, vamos adicionar a lei do terceiro excluído como um axioma:

```
Section pbc.
 Variable phi: Prop.

 Axiom lem: phi \/ ~phi.

 Hypothesis H: ~phi -> False.
 Lemma pbc: phi.
 Proof.
```

e o contexto de prova correspondente é como a seguir:

```
 phi : Prop
 H : ~ phi -> False
 ============================
 phi
```

Podemos adicionar um axioma ou lema no contexto via a tática `pose proof`. Neste caso, usamos `pose proof lem` para obtermos o seguinte contexto:

```
 phi : Prop
 H : ~ phi -> False
 H0 : phi \/ ~ phi
 ============================
 phi
```

Agora podemos dividir a prova em duas subprovas com a tática `destruct H0`. A primeira subprova é trivial porque o que queremos provar está nas hipóteses. Na segunda subprova, temos pelo menos dois caminhos possíveis para seguir. O primeiro consiste em manipular as hipóteses (raciocínio de cima para baixo) para gerar o absurdo nas hipóteses por meio da tática `apply` no seguinte contexto:

```
  phi : Prop
   H : ~ phi -> False
   H0 : ~ phi
   ============================
   phi
```

Como resultado, temos o absurdo como hipótese e podemos provar qualquer coisa via a regra da explosão (tática `contradiction`):

```
   phi : Prop
   H : ~ phi -> False
   H0 : False
   ============================
   phi
```

A prova completa é dada a seguir:

```
Variable phi: Prop.

Axiom lem: phi \/ ~phi.

Hypothesis H: ~phi -> False.
Lemma pbc: phi.
Proof.
 pose proof lem.
 destruct H0.
 - assumption.
 - apply H in H0.
   contradiction.
Qed.
```

O segundo caminho consiste em gerar o absurdo como objetivo a ser provado, ou seja, aplicamos a regra da explosão de baixo para cima na prova. Isto pode ser feito com a tática `apply False_ind` que simplesmente troca o objetivo atual (qualquer que seja ele) pelo absurdo. Neste caso, podemos aplicar a hipótese H (com a tática `apply H`) e concluir a prova com `assumption`.

```
Variable phi: Prop.

Axiom lem: phi \/ ~phi.

Hypothesis H: ~phi -> False.
Lemma pbc: phi.
Proof.
 pose proof lem.
 destruct H0.
 - assumption.
 - apply False_ind.
   apply H.
   assumption.
Qed.
```

Exercício 48. *Acabamos de caracterizar a lógica proposicional clássica como sendo a lógica proposicional intuicionista juntamente com a lei do terceiro excluído (ver Tabela 2.6), mas outras caracterizações são possíveis. Por exemplo, a lógica minimal juntamente com a regra de prova por contradição (PBC) também nos dá a lógica proposicional clássica. Ou seja, a Tabela 2.7 nos dá outra caracterização da lógica*

	Contexto explícito	Contexto implícito
1	$\dfrac{\Gamma \vdash \varphi_1 \qquad \Gamma \vdash \varphi_2}{\Gamma \vdash \varphi_1 \wedge \varphi_2}\ (\wedge_i)$	$\dfrac{\varphi_1 \qquad \varphi_2}{\varphi_1 \wedge \varphi_2}\ (\wedge_i)$
2	$\dfrac{\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Gamma \vdash \varphi_{i\in\{1,2\}}}\ (\wedge_e)$	$\dfrac{\varphi_1 \wedge \varphi_2}{\varphi_{i\in\{1,2\}}}\ (\wedge_e)$
3	$\dfrac{\Gamma \vdash \varphi_{i\in\{1,2\}}}{\Gamma \vdash \varphi_1 \vee \varphi_2}\ (\vee_i)$	$\dfrac{\varphi_{i\in\{1,2\}}}{\varphi_1 \vee \varphi_2}\ (\vee_i)$
4	$\dfrac{\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Gamma', \varphi_1 \vdash \gamma \qquad \Gamma'', \varphi_2 \vdash \gamma}{\Gamma, \Gamma', \Gamma'' \vdash \gamma}\ (\vee_e)$	$\dfrac{\varphi_1 \vee \varphi_2 \qquad \begin{array}{c}[\varphi_1]^u \\ \vdots \\ \gamma\end{array} \qquad \begin{array}{c}[\varphi_2]^v \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\vee_e)\ u, v$
5	$\dfrac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}\ (\to_{\mathrm{i}})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \psi\end{array}}{\varphi \to \psi}\ (\to_i)\ u$
6	$\dfrac{\Gamma \vdash \varphi \to \psi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \psi}\ (\to_e)$	$\dfrac{\varphi \to \psi \qquad \varphi}{\psi}\ (\to_e)$
7	$\dfrac{\Gamma, \varphi \vdash \bot}{\Gamma \vdash \neg\varphi}\ (\neg_{\mathrm{i}})$	$\dfrac{\begin{array}{c}[\varphi]^u \\ \vdots \\ \bot\end{array}}{\neg\varphi}\ (\neg_i)\ u$
8	$\dfrac{\Gamma \vdash \neg\varphi \qquad \Gamma \vdash \varphi}{\Gamma \vdash \bot}\ (\neg_e)$	$\dfrac{\neg\varphi \qquad \varphi}{\bot}\ (\neg_e)$
9	$\dfrac{\Gamma, \neg\varphi \vdash \bot}{\Gamma \vdash \varphi}$ (PBC)	$\dfrac{\begin{array}{c}[\neg\varphi]^u \\ \vdots \\ \bot\end{array}}{\varphi}\ (PBC)\ u$

Tabela 2.7: Regras da Lógica Clássica (versão 2)

proposicional clássica. Para mostrarmos que esta é, de fato, uma caracterização da lógica proposicional clássica precisamos provar tanto a regra da explosão quanto a lei do terceiro excluído a partir das regras da Tabela 2.7. Sendo assim, prove os sequentes a seguir utilizando as regras da Tabela 2.7:

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \vee \neg\varphi$ *(lei do terceiro excluído)*
3. *Refaça estas duas provas no Coq.*

Uma terceira caracterização possível para a lógica proposicional clássica é com a regra de eliminação da dupla negação:

Contexto explícito	Contexto implícito
$\dfrac{\Gamma \vdash \neg\neg\varphi}{\Gamma \vdash \varphi}\ (\neg\neg_e)$	$\dfrac{\neg\neg\varphi}{\varphi}\ (\neg\neg_e)$

Exercício 49. *Substitua a regra 9 (PBC) na Tabela 2.7 pela regra* $(\neg\neg_e)$*, e prove os seguintes sequentes:*

1. $\bot \vdash \varphi$ *(regra da explosão)*
2. $\vdash \varphi \vee \neg\varphi$ *(lei do terceiro excluído)*
3. $\neg\varphi \to \bot \vdash \varphi$ *(prova por contradição)*

4. Refaça estas três provas no Coq.

Considerando que uma negação, digamos $\neg\varphi$, é o mesmo que $\varphi \to \bot$, é fácil ver que as regras de eliminação da dupla negação e prova por contradição são maneiras diferentes de escrever a mesma coisa (por que?). Uma outra caracterização possível da lógica proposicional clássica envolve a chamada *lei de Peirce* (LP), como detalhado no exemplo a seguir:

Contexto explícito	Contexto implícito
$\dfrac{}{\vdash ((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)	$\dfrac{}{((\varphi \to \psi) \to \varphi) \to \varphi}$ (LP)

Exercício 50. *Assuma a regra (LP) acima, e prove o sequente* $\vdash \varphi \vee \neg\varphi$ *utilizando as regras da Tabela 2.5*

Exercício 51. $\psi_1 \wedge \psi_2 \dashv\vdash \neg(\neg\psi_1 \vee \neg\psi_2)$

Exercício 52. $\psi_1 \to \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$

Exercício 53. $\varphi \leftrightarrow \neg\varphi \vdash \bot$[2]

Exemplo 54. *Considere o seguinte problema: Em uma ilha moram apenas dois tipos de pessoas: as honestas, que sempre falam a verdade; e as desonestas, que sempre mentem. Um viajante, ao passar por esta ilha encontra três moradores chamados A, B e C. O viajante pergunta para o morador A: "Você é honesto ou desonesto?" A responde algo incompreensível, e o viajante pergunta para B: "O que ele disse?" B então responde "Ele disse que é desonesto". Neste momento C se manifesta: "Não acredito nisto! Isto é uma mentira!". Questão: C é honesto ou desonesto?*

Para resolver este problema pense no que ocorre se um morador desta ilha, digamos X, disser "Eu sou desonesto"? Isto nos levaria a uma contradição! De fato, se X for honesto então ele disse a verdade, e portanto é desonesto. Por outro lado, se X é desonesto então ele mentiu, e portanto é honesto. Assim, como A não poderia ter dito que é desonesto, podemos concluir que B é desonesto! E portanto, C é honesto! Vamos construir uma prova de que este raciocínio está em correto usando a teoria que estudamos? O ponto de partida é construir um sequente que corresponda ao enunciado deste problema. Que variáveis proposicionais vamos precisar? Certamente precisamos de variáveis que nos permitam caracterizar quando um morador é ou não honesto. Assim, utilizaremos três variáveis proposicionais com a seguinte semântica:

- *a: o morador A é honesto;*
- *b: o morador B é honesto;*
- *c: o morador C é honesto.*

Desta forma, a negação de qualquer destas variáveis significa que o morador correspondente é desonesto. Agora precisamos representar o que foi dito por cada um dos moradores por meio de uma fórmula da lógica proposicional. Considere o que disse o morador B: "Ele disse que é desonesto", quer dizer, o morador B disse que o morador A disse que era desonesto. Como codificar este fato por meio de uma fórmula da LP? Vamos iniciar considerando uma situação geral e mais simples. Digamos que um morador X tenha dito Y, isto é, "X disse Y". Que fórmula da LP corresponde a este fato? Suponha que a variável x codifica a proposição "X é honesto". Então observe que, se X for honesto então o que ele disse é verdade, ou seja, tanto x quanto Y são verdade. Por outro lado, se X for desonesto então Y é falso, e tanto x quanto Y são falsos. Assim, podemos concluir que as variáveis x e Y são equivalentes, no sentido que ou ambas são verdadeiras, ou ambas são falsas. Assim, podemos representar a afirmação

[2]Note que a bi-implicação $\psi \leftrightarrow \gamma$ é apenas uma notação compacta para $(\psi \to \gamma) \wedge (\gamma \to \psi)$.

"X disse Y "pela fórmula $x \leftrightarrow Y$. Voltando então ao nosso problema original, podemos agora representar o fato de que o morador B disse que o morador A disse que era desonesto pela fórmula $b \leftrightarrow (a \leftrightarrow (\neg a))$. O morador C por sua vez, disse que B mentiu, o que corresponde a fórmula $c \leftrightarrow (\neg b)$. Com isto podemos montar o sequente a ser provado: $b \leftrightarrow (a \leftrightarrow (\neg a)), c \leftrightarrow (\neg b) \vdash c$.

Exercício 55. *Prove o sequente $b \leftrightarrow (a \leftrightarrow (\neg a)), c \leftrightarrow (\neg b) \vdash c$ construído no exemplo anterior. Em seguida refaça a prova em Coq.*

Exercício 56. *Considere uma ilha onde moram apenas dois tipos de pessoas: as honestas, e que portanto sempre falam a verdade; e as desonestas, que sempre mentem. Um viajante, ao passar por esta ilha encontra três moradores chamados A, B e C. O viajante pergunta para o morador A: "Quantos, dentre vocês três, são desonestos?" A responde algo incompreensível, e o viajante pergunta para B: "O que ele disse?" B então responde "Ele disse que exatamente dois de nós somos desonestos". Neste momento C se manifesta: "Não acredito nisto! Isto é uma mentira!". Questão: C é honesto ou desonesto?*

Exercício 57. *Em uma ilha moram apenas dois tipos de habitantes: os honestos, que sempre falam a verdade; e os desonestos, que sempre mentem. Você encontra dois habitantes desta ilha, digamos João e José. João diz que José é desonesto. José diz "Nem João nem eu somos desonestos". Você consegue determinar qual dos dois é honesto e qual é desonesto?*

No exemplo anterior, utilizamos a associação do valor de verdade (verdadeiro ou falso) de uma variável proposicional para resolver um problema. Esta abordagem está relacionado com a semântica da lógica proposicional clássica que nos fornece os meios para concluir quando uma fórmula é verdadeira ou falsa. A gramática (2.1) define como são as fórmulas da LP, a partir de seis construtores:

1. O primeiro denota uma variável proposicional, e caracteriza uma fórmula atômica, i.e. uma fórmula que não pode ser subdividida em uma fórmula menor.

2. O segundo construtor é uma constante que denota o absurdo ($\bot$), que também é uma fórmula atômica. O absurdo é utilizado para representar uma fórmula que tem valor de verdade "falso (F)". É importante observar que podemos associar a qualquer fórmula da LP apenas dois valores de verdade, a saber: verdadeiro (T) ou falso (F).

3. O terceiro construtor denota a negação e nos permite construir uma nova fórmula a partir de uma fórmula dada. Assim, dada uma fórmula φ, podemos construir a sua negação ($\neg\varphi$). A semântica da negação é a que conhecemos intuitivamente: se uma fórmula φ é verdadeira (T) então sua negação é falsa (F), e vice-versa. Normalmente, representamos este fato via a seguinte tabela:

φ	$(\neg\varphi)$
T	F
F	T

4. O quarto construtor denota a conjunção e nos permite construir uma nova fórmula a partir de duas fórmulas dadas. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua conjunção ($\varphi_1 \wedge \varphi_2$). A semântica da conjunção também é a usual, isto é, a conjunção ($\varphi_1 \wedge \varphi_2$) é verdadeira somente quando φ_1 e φ_2 são simultaneamente verdadeiras:

   Aqui é importante observar que a leitura da construção da conjunção na gramática 2.1 não diz que suas componentes são iguais (apesar da utilização do mesmo símbolo φ nas duas componenetes). Lembre-se que a leitura desta construção em 2.1 é: dadas duas fórmulas (não necessariamente

φ_1	φ_2	$(\varphi_1 \wedge \varphi_2)$
T	T	T
T	F	F
F	T	F
F	F	F

iguais!), podemos construir a sua conjunção. Alternativamente, poderíamos ter escrito a gramática 2.1 da seguinte forma equivalente:

$$\varphi, \psi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \psi) \mid (\varphi \vee \psi) \mid (\varphi \to \psi) \tag{2.4}$$

5. O quinto construtor denota a disjunção e, como no caso anterior, nos permite construir uma nova fórmula a partir de duas fórmulas dadas. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua disjunção $(\varphi_1 \vee \varphi_2)$, cuja semântica é dual à semântica da conjunção: a disjunção $(\varphi_1 \vee \varphi_2)$ é falsa somente quando φ_1 e φ_2 são simultaneamente falsas.

φ_1	φ_2	$(\varphi_1 \vee \varphi_2)$
T	T	T
T	F	T
F	T	T
F	F	F

6. O sexto construtor é a implicação. Assim, dadas duas fórmulas φ_1 e φ_2, podemos construir a sua implicação $(\varphi_1 \to \varphi_2)$ com a semântica dada na tabela abaixo.

φ_1	φ_2	$(\varphi_1 \to \varphi_2)$
T	T	T
T	F	F
F	T	T
F	F	T

O sentido usual da implicação assume implicitamente uma relação de causa e efeito, ou causa e consequência no sentido de que o antecedente φ_1 é o que gera o consequente φ_2 como em "Se eu não beber água então ficarei desidratado". No entanto, o sentido da implicação na lógica é um pouco diferente pois tem como fundamento a *preservação da verdade*, que não necessariamente possui uma relação de causa e efeito. Por exemplo, a proposição "Se 2+2=4 então o dia tem 24 horas"é verdadeira, mas não existe relação causal entre a igualdade 2+2=4 e o fato de o dia ter 24 horas de duração.

Uma gramática como 2.1 (ou 2.4) nos fornece as regras sintáticas para a construção das fórmulas da LP. São quatro construtores recursivos (negação, conjunção, disjunção e implicação) também chamados de conectivos lógicos, e dois não recursivos.

Apesar da gramática apresentada acima não incluir a bi-implicação, este é um conectivo bastante utilizado. De fato, a bi-implicação, já utilizada em exemplos anteriores, pode ser reescrita em usando a implicação e conjunção. Como exercício construa a tabela verdade da bi-implicação e observe que $\varphi \leftrightarrow \psi$ é verdadeira somente quando φ e ψ possuem o mesmo valor de verdade. Adicionalmente, dizemos que duas fórmulas φ e ψ são **equivalentes** quando a fórmula $\varphi \leftrightarrow \psi$ é uma tautologia:

Tautologia	Uma fórmula que é sempre verdadeira, independentemente dos valores de verdade associados às suas variáveis.
Contradição	Uma fórmula que é sempre falsa, independentemente dos valores de verdade associados às suas variáveis.
Contingência	Uma fórmula que pode ser tanto verdadeira quanto falsa dependendo dos valores de verdade associados às suas variáveis.

As tautologias e as contradições são particularmente importantes, e possuem símbolos especiais para representá-las. Nas gramáticas 2.1 e 2.4 já vimos que a constante $\bot$ é o representante das contradições. As tautologias, por sua vez, podem ser representadas pelo símbolo $\top$.

Agora chegamos em um momento chave do nosso estudo. Considere um sequente arbitrário, digamos $\Gamma \vdash \varphi$, onde Γ é um conjunto finito de fórmulas da LP. Podemos então perguntar: é possível provar este sequente? Ou em outras palavras, qualquer sequente possui uma prova? A resposta certamente é não. Se tudo pudesse ser provado então não teríamos razão para estudar a lógica proposicional. Como então é possível separar os sequentes que têm prova dos que não podem ser provados? Para responder esta pergunta precisamos inicialmente compreender a noção de **consequência lógica**. Dizemos que uma fórmula φ é consequência lógica da fórmula ψ, notação $\psi \models \varphi$, se φ for verdadeira sempre que ψ for verdadeira. Este conceito pode ser facilmente estendido para um conjunto Γ de fórmulas, de forma que $\Gamma \models \varphi$, isto é, φ é consequência lógica do conjunto Γ se φ for verdadeira sempre que as fórmulas em Γ forem verdadeiras. Agora podemos enunciar dois teoremas importantes que nos permitirão responder à questão anterior:

Teorema 58 (Correção da LP)**.** *Sejam Γ um conjunto, e φ uma fórmula da lógica proposicional. Se $\Gamma \vdash \varphi$ então $\Gamma \models \varphi$.*

A prova deste teorema é por indução em $\Gamma \vdash \varphi$. Não detalharemos aqui esta prova, que pode ser encontrada por exemplo em [4].

Teorema 59 (Completude da LP)**.** *Sejam Γ um conjunto, e φ uma fórmula da lógica proposicional. Se $\Gamma \models \varphi$ então $\Gamma \vdash \varphi$.*

A prova do teorema de completude da LP é um pouco mais complexa do que a prova de correção, e também pode ser encontrada em [4]. Note que este lema responde a nossa pergunta anterior: um sequente tem prova exatamente quando seu consequente for consequência lógica do seu antecedente.

A lógica proposicional nos permite resolver diversos problemas, e constitui a base de tudo o que faremos nos próximos capítulos. Apesar de muito importante como ponto de partida no estudo que estamos fazendo, a lógica proposicional possui limitações importantes, como a impossibilidade de quantificar de forma explícita sobre elementos de um conjunto. Por exemplo, podemos representar a sentença "Todo mundo gosta de Matemática"na LP via uma variável proposicional, mas esta representação não expressa a quantificação universal "Todo mundo"de forma explícita. O mesmo vale para uma sentença da forma "Existe um número natural que não é primo". O próprio princípio da indução, tão importante em Matemática e Computação, precisa de uma linguagem mais expressiva do que a proposicional. A lógica que nos permitirá expressar este tipo de quantificação (tanto existencial quanto universal) é conhecida como *Lógica de Primeira Ordem* (LPO), ou *Lógica de Predicados* que estudaremos no próximo capítulo.

Capítulo 3

A Lógica de Primeira Ordem

Nesta seção vamos em um certo sentido estender a Lógica Proposicional para ganhar em poder de expressividade. Como é a gramática da Lógica de Primeira Ordem (LPO)? Isto é, qual a linguagem que precisamos para conseguir expressar quantificação universal e existencial? Inicialmente, precisamos representar os elementos que podem ser quantificados. Assim, diferentemente do caso proposicional, temos duas classes de objetos na LPO: *termos* e *fórmulas*. Os termos são representados pela seguinte gramática:

$$t ::= x \mid f(t, \ldots, t) \tag{3.1}$$

ou seja, os termos são construídos a partir de variáveis (no sentido usual da palavra em Matemática) e, funções com uma certa aridade (i.e número de argumentos). Observe que os termos vão representar os elementos do conjunto sobre o qual podemos quantificar e caracterizar por meio de propriedades. Por exemplo, considere o conjunto dos números naturais $\mathbb{N}$. Neste caso, as variáveis representam números naturais, e exemplos de funções são: sucessor (aridade 1), soma (aridade 2), etc. O conjunto das variáveis de um termo t, notação (t), consiste no conjunto das variáveis que ocorrem em t, e pode ser definido indutivamente por:

Definição 60. *O conjunto $var(t)$ das variáveis que ocorrem no termo t é definido indutivamente como a seguir:*

1. $var(x) = \{x\}$;

2. $var(f(t_1, t_2, \ldots, t_n)) = var(t_1) \cup var(t_2) \cup \ldots \cup var(t_n)$.

Denotaremos por $t[[x/u]]$ o termo obtido ao se substituir todas as ocorrências da variável x pelo termo u no termo t.

As fórmulas da LPO utilizam os mesmos conectivos da LP e são definidas pela seguinte gramática:

$$\varphi ::= p(t, \ldots, t) \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \to \varphi) \mid \exists_x \varphi \mid \forall_x \varphi \tag{3.2}$$

onde o primeiro construtor representa uma fórmula atômica, e os dois últimos representam, respectivamente, a quantificação existencial e universal. Note que as fórmulas atômicas representam fórmulas que não podem ser decompostas, e que têm termos como argumentos. Em uma fórmula atômica da forma $p(t_1, \ldots, t_n)$, p é um *predicado* de aridade n, e t_1, ..., t_n são termos. A LPO é a lógica utilizada no dia a dia dos matemáticos, ainda que de maneira informal. Com os predicados podemos expressar propriedades dos termos. Por exemplo, ainda no conjunto dos números naturais, podemos expressar a propriedade de um número natural ser primo por meio de um predicado unário, digamos p. Desta forma, a fórmula $p(x)$ pode expressar o fato de x ser primo. Outros exemplos de fórmulas atômicas incluem os predicados $\leq$, $\geq$, $<$ e $>$ que normalmente usamos em notação infixa como em $2 \leq 5$, por exemplo.

Observe que agora existem dois tipos de variáveis na linguagem da Lógica de Primeira Ordem. Por exemplo, considere as fórmulas $q(x)$ e $\forall_x p(x)$. Em $\forall_x p(x)$ ocorrência da variável x em $p(x)$ está **ligada** ao quantificador universal, enquanto que na fórmula $q(x)$, a variável x está **livre**. De uma forma geral, dizemos que uma ocorrência de uma variável é ligada, se ela estiver no escopo de um quantificador (universal ou existencial), e livre, se a ocorrência não estiver no escopo de nenhum quantificador. Observe que uma variável pode ocorrer livre e ligada em uma mesma fórmula: $q(x) \vee \forall_x p(x)$. O conjunto das variáveis livres de uma fórmula φ, notação $FV(\varphi)$, é definido indutivamente como segue:

Definição 61. *Seja φ uma fórmula da Lógica de Primeira Ordem. O conjunto $FV(\varphi)$ das variáveis livres da fórmula φ é definido indutivamente na estrutura de φ por:*

1. $FV(p(t_1, t_2, \ldots, t_n)) = var(t_1) \cup var(t_2) \cup \ldots \cup var(t_n)$;
2. $FV(\bot) = \{\}$;
3. $FV(\neg\psi) = FV(\psi)$;
4. $FV(\psi \star \gamma) = FV(\psi) \cup FV(\gamma)$, *onde* $\star \in \{\wedge, \vee, \rightarrow\}$;
5. $FV(Q_x\psi) = FV(\psi)\backslash\{x\}$, *onde* $Q \in \{\forall, \exists\}$.

De maneira análoga podemos definir o conjunto das variáveis ligadas de uma fórmula:

Definição 62. *Seja φ uma fórmula da Lógica de Primeira Ordem. O conjunto $BV(\varphi)$ das variáveis ligadas da fórmula φ é definido indutivamente na estrutura de φ por:*

1. $BV(p(t_1, t_2, \ldots, t_n)) = \{\}$;
2. $BV(\bot) = \{\}$;
3. $BV(\neg\psi) = BV(\psi)$;
4. $BV(\psi \star \gamma) = BV(\psi) \cup BV(\gamma)$, *onde* $\star \in \{\wedge, \vee, \rightarrow\}$;
5. $BV(Q_x\psi) = BV(\psi) \cup \{x\}$, *onde* $Q \in \{\forall, \exists\}$.

Estas noções são importantes porque a operação de substituição na Lógica de Primeira Ordem é definida de tal forma a evitar captura de variáveis, diferentemente da substituição feita em termos vista anteriormente. Isto significa que, por exemplo, se quisermos substituir a ocorrência de y em $\forall_x p(x, y)$ por x, o resultado não pode ser $\forall_x p(x, x)$ já que neste caso a segunda ocorrência de x que era livre, passou a ser ligada depois da substituição, ou seja, a segunda ocorrência de x foi capturada. Para evitar este problema, podemos renomear as variáveis ligadas de uma fórmula sempre que necessário. De fato, observe que as fórmulas $\forall_x q(x)$, $\forall_y q(y)$ e $\forall_z q(z)$ têm todas a mesma semântica. Isto significa que o renomeamento de variáveis ligadas não muda o sentido, ou significado, de uma fórmula. Para enfatizarmos a operação de substituição que definiremos a seguir, denotaremos por $\varphi[x/t]$ o resultado de substituir todas as ocorrências livres de x na fórmula φ pelo termo t. Quando a variável a ser substituída não precisar

ser enfatizada (por exemplo, por poder ser facilmente obtida do contexto), escreveremos simplesmente $\varphi(t)$ ao invés de $\varphi[x/t]$.

Definição 63. *Seja φ uma fórmula da Lógica de Primeira Ordem. A operação de substituir todas as ocorrências livres da variável x pelo termo t em φ, notação $\varphi[x/t]$ é definida indutivamente na estrutura da fórmula φ da seguinte forma:*

1. $p(t_1, t_2, \ldots, t_n)[x/t] = p(t_1[[x/t]], t_2[[x/t]], \ldots, t_n[[x/t]])$;

2. $\bot[x/t] = \bot$;

3. $(\neg\psi)[x/t] = \neg(\psi[x/t])$;

4. $(\psi \star \gamma)[x/t] = (\psi[x/t]) \star (\gamma[x/t])$, *onde* $\star \in \{\vee, \wedge, \rightarrow\}$;

5. $(Q_y\psi)[x/t] = \begin{cases} Q_y\psi, & \text{se } x = y; \\ Q_y(\psi[x/t]), & \text{se } y \notin var(t); \\ Q_z(\psi[y/z][x/t]), & \text{c.c.} \end{cases}$
   onde z é uma variável nova, e $Q \in \{\forall, \exists\}$.

Observe que o primeiro caso do item 5 da definição anterior, a substituição não tem nenhum efeito sobre a fórmula quando a variável da substituição coincide com a variável do quantificador ($x = y$), e portanto variáveis ligadas não são substituídas. O caso em que $y \notin var(t)$ faz a propagação da substituição para dentro do corpo do quantificador já que não há possibilidade de captura de variável. Por fim, quando $x \neq y$ e $y \in var(t)$ a variável do quantificador é renomeada para um nome novo, no caso z, as ocorrências de y em ψ são renomeadas para z e então a substituição é propagada para dentro do corpo do quantificador.

O sistema de dedução natural na LPO possui as mesmas regras utilizadas no caso proposicional, mas agora aplicadas a fórmulas da LPO, e adicionalmente temos as regras de introdução e eliminação para os quantificadores que apresentamos a seguir.

A regra de introdução do quantificador universal permite a construção de uma prova de uma fórmula da forma $\forall_x\varphi(x)$, ou seja, queremos concluir que a propriedade φ é satisfeita por qualquer elemento x do domínio. Mas o que precisamos para garantir que todo elemento x do domínio tenha a propriedade φ? Uma maneira seria tentar a construção individual de cada uma destas provas, ou seja, suponha que o domínio seja o conjunto $\{x_0, x_1, x_2 \ldots\}$ que pode ser finito ou infinito, e considere uma prova de $\varphi(x_0)$, isto é, uma prova de que x_0 satisfaz a propriedade φ. Seria possível repetir esta prova para x_1, x_2, e assim sucessivamente? Se pudermos repetir a mesma prova para todos os elementos do domínio então certamente podemos concluir $\forall_x\varphi(x)$. Para que uma generalização desta forma seja possível precisamos que a prova de $\varphi(x_0)$ não dependa de hipótese que assuma alguma informação sobre x_0.

$$\frac{\varphi(x_0)}{\forall_x\varphi(x)}\ (\forall_i)$$ se a prova de $\varphi(x_0)$ não depende de hipótese não-descartada que contenha x_0.

A regra de eliminação do quantificador universal nos permite instanciar a variável quantificada universalmente x com qualquer elemento t do domínio.

$$\frac{\forall_x \varphi(x)}{\varphi(t)}\ (\forall_e)$$

A analogamente, a regra de introdução do quantificador existencial nos permite concluir que existe um elemento que satisfaz a propriedade φ a partir da prova de que algum elemento do domínio, digamos t, satisfaça a propriedade φ.

$$\frac{\varphi(t)}{\exists_x \varphi(x)}\ (\exists_i)$$

Por fim, a regra de eliminação do quantificador existencial é dada como a seguir:

$$\frac{\exists_x \varphi(x) \qquad \begin{array}{c}[\varphi(x_0)]^u \\ \vdots \\ \gamma\end{array}}{\gamma}\ (\exists_e)\ u$$

onde x_0 é uma variável nova que não ocorre em γ.

Nesta regra provamos γ a partir de uma prova de $\exists_x \varphi(x)$, e de uma prova de γ a partir da suposição $\varphi(x_0)$. Ou seja, como temos uma prova de $\exists_x \varphi(x)$, então temporariamente assumimos que x_0 (um novo elemento que, portanto, não pode ter sido utilizado antes) satisfaz a propriedade φ. Se a partir desta suposição pudermos provar uma fórmula, digamos γ, que não dependa de x_0 então podemos concluir γ após descartar a suposição $\varphi(x_0)$.

Exercício 64. *Apresente derivações em Dedução Natural para os sequentes $\forall_x \neg\varphi \dashv\vdash \neg\exists_x \varphi$ na lógica minimal.*

Exercício 65. *Apresente derivações em Dedução Natural para os sequentes $\neg\forall x\, \phi \dashv\vdash \exists x\, \neg\phi$, em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 66. *Apresente derivações em Dedução Natural para os sequentes $\forall x\, \phi \dashv\vdash \neg\exists x\, \neg\phi$, em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 67. *Apresente derivações em Dedução Natural para os sequentes $\exists x\, \phi \dashv\vdash \neg\forall x\, \neg\phi$, em seguida classifique cada prova como minimal, intuicionista ou clássica.*

Exercício 68. *Apresente derivações em Dedução Natural para os sequentes a seguir assumindo que x não ocorre livre em ψ, em seguida classifique cada prova como minimal, intuicionista ou clássica.*

1. $(\forall x\, \phi) \wedge \psi \vdash \forall x\, (\phi \wedge \psi)$
2. $(\exists x\, \phi) \wedge \psi \vdash \exists x\, (\phi \wedge \psi)$
3. $\forall x\, (\psi \to \phi) \vdash \psi \to \forall x\, \phi$
4. $\forall x\, (\phi \to \psi) \vdash (\exists x\, \phi) \to \psi$

Exercício 69. *Prove que não existe uma derivação intuicionista para os sequentes a seguir:*

1. $\neg\exists_x\neg\varphi \vdash \forall_x\varphi$

2. $\neg\forall_x\neg\varphi \vdash \exists_x\varphi$

3. $\varphi \to \psi \vdash (\neg\varphi) \vee \psi$

Assim como a lógica proposicional, a lógica de primeira ordem é correta e completa, mas estes resultados não serão provados aqui (Veja [4]).

3.1 Indução

Indução é uma técnica de prova muito poderosa que desempenha um papel fundamental tanto em Matemática quanto na construção de algoritmos. A ideia é bastante intuitiva: suponha que os elementos deste conjunto possam ser colocados um após o outro como peças de um dominó, de tal forma que, se uma peça qualquer for derrubada então a peça que está logo em seguida também é derrubada. Então podemos concluir, que se a primeira peça for derrubada então **todas** as outras serão derrubadas. Ou seja, voltando ao contexto de propriedades de elementos de um conjunto, a ideia é provar que se um elemento arbitrário do conjunto satisfaz a propriedade então o próximo elemento também satisfaz a propriedade. Se esta prova puder ser feita juntamente com a prova de que o primeiro elemento do conjunto também satisfaz a propriedade então podemos concluir que todos os elementos do conjunto satisfazem a propriedade.

Vamos iniciar este estudo sobre indução no contexto dos números naturais, onde esta noção de ordem é bem clara: o primeiro elemento é o 0, em seguida vem o 1, depois o 2, etc. De uma forma geral, depois de um número natural k vem o natural $S\ k$, o sucessor de k que também escrevemos como $k+1$. A indução no contexto dos números naturais é conhecida como *indução matemática*, e será explorada na próxima seção.

1. Indução Matemática

   O conjunto dos números naturais $\mathbb{N}$, que pode ser definido pela gramática a seguir:

$$n ::= 0 \mid S\ n \tag{3.3}$$

   A gramática (3.3) possui dois construtores: 0 e S. O primeiro diz que 0 é um número natural, e o segundo diz que a partir de um natural já construído, digamos n, podemos construir um outro natural, a saber, $S\ n$, ou seja, o sucessor de n. Muito bem, agora considere uma propriedade qualquer dos números naturais. Por exemplo, a que diz que a soma dos n primeiros números ímpares é igual a n^2. Como podemos provar esta propriedade? Isto mesmo, por indução! O que diz mesmo o princípio de indução para os números naturais? Diz que se uma propriedade P vale para 0 (base da indução), e se, supondo que P vale para um natural arbitrário k (hipótese de indução), podemos provar que ela vale também para $S\ k$ (o sucessor de k)[1] (passo indutivo) então podemos concluir que P vale para todos os números naturais. Esquematicamente, podemos apresentar este princípio, denominado *Princípio da Indução Matemática (PIM)*, como a seguir:

$$\frac{P\ 0 \qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}\ (PIM)$$

[1] Note que o sucessor de k pode ser escrito como $S\ k$ ou $k+1$.

Exemplo 70. *Queremos provar que a soma dos n primeiros números ímpares é igual a n^2. Esta propriedade vale trivialmente para o 0 (a soma dos 0 primeiros números ímpares é igual a 0^2). Agora suponha que a soma dos k primeiros números ímpares seja igual a k^2 (hipótese de indução). O $(k+1)$-ésimo número ímpar é igual a $2.k+1$ (por que?), e portanto a soma dos $k+1$ primeiros números ímpares é $k^2+2.k+1=(k+1)^2$, como queríamos provar.*

Uma outra forma de resolver este problema em um contexto mais formal pode ser feita a partir de uma definição formal da soma dos n primeiros números ímpares por meio do somatório $\sum_{i=1}^{n}(2.i-1)$, que por definição é igual a 0, se $n=0$. Queremos provar que $\sum_{i=1}^{n}(2.i-1)=n^2$, para todo número natural n. Aplicando o princípio da indução, teremos 2 casos para analisar:

- *(**Base da indução**): A base da indução se dá quando $n=0$, e é trivial porque o lado esquerdo da igualdade é igual a 0 por definição.*

- *(**Passo indutivo**): O passo indutivo é a parte interessante de qualquer prova por indução. Neste caso específico, vamos assumir que a propriedade que queremos provar vale para um número natural arbitrário, digamos k, e provaremos que esta propriedade continua valendo para o natural $k+1$. Ou seja, assumimos que $\sum_{i=1}^{k}(2.i-1)=k^2$, e vamos provar que $\sum_{i=1}^{k+1}(2.i-1)=(k+1)^2$. Partindo do lado esquerdo desta igualdade, podemos decompor o somatório da seguinte forma $\sum_{i=1}^{k+1}(2.i-1)=\sum_{i=1}^{k}(2.i-1)+(2.k+1)$, e agora podemos utilizar a hipótese de indução (h.i.) para assim chegarmos ao lado direito da igualdade: $\sum_{i=1}^{k+1}(2.i-1)=\sum_{i=1}^{k}(2.i-1)+(2.k+1) \overset{h.i.}{=} k^2+(2.k+1)=(k+1)^2$.*

Por fim, apresentamos esta prova na forma de árvore:

$$\dfrac{\dfrac{0=0}{\sum_{i=1}^{0}(2.i-1)=0^2} \qquad \dfrac{\dfrac{\dfrac{[\sum_{i=1}^{k}(2.i-1)=k^2]^u}{\sum_{i=1}^{k}(2.i-1)+(2.k+1)=(k+1)^2}}{\sum_{i=1}^{k+1}(2.i-1)=(k+1)^2}(\to_i)\,u}{\sum_{i=1}^{k}(2.i-1)=k^2 \to \sum_{i=1}^{k+1}(2.i-1)=(k+1)^2}}{\sum_{i=1}^{n}(2.i-1)=n^2}\,(\text{Ind. em } n)$$

Exercício 71. *Prove que a soma dos n primeiros números naturais é igual a $\frac{n(n+1)}{2}$. Ou seja, mostre que* $\sum_{i=1}^{n} i = \dfrac{n(n+1)}{2}$.

Exercício 72. *Prove que a soma dos n primeiros quadrados é igual a $\frac{n(n+1)(2n+1)}{6}$. Ou seja, mostre que* $\sum_{i=1}^{n} i^2 = \dfrac{n(n+1)(2n+1)}{6}$.

Exercício 73. *Prove que a soma dos n primeiros cubos é igual ao quadrado da soma de 1 até n, ou seja, que $1^3+2^3+\ldots+n^3=(1+2+\ldots+n)^2$.*

Agora vamos refazer estas provas no computador. Para isto utilizaremos o assistente de provas Coq (`https://coq.inria.fr`). Inicialmente, com o comando `Print nat`, podemos ver como os números naturais estão definidos:

```
Inductive nat : Set :=  O : nat | S : nat -> nat.
```

A linha acima nos diz que o tipo `nat` dos números naturais é definido indutivamente (palavra reservada `Inductive`), e que esta definição possui dois construtores: o `O` que tem tipo `nat`, ou seja, `O` é um número natural; e o `S` que é uma função (a função sucessor) que recebe um número natural como argumento e retorna outro natural como resultado. Esta é essencialmente a mesma informação dada pela gramática (3.3). Toda definição indutiva em Coq é capaz de gerar um princípio de indução de forma automática. No caso de `nat` podemos acessar este princípio pelo comando `Print nat_ind`:

```
nat_ind =
fun (P : nat -> Prop) (f : P 0) (f0 : forall n : nat, P n -> P (S n)) =>
fix F (n : nat) : P n := match n as n0 return (P n0) with
                         | 0 => f
                         | S n0 => f0 n0 (F n0)
                         end
:forall P: nat -> Prop, P 0 -> (forall n: nat, P n -> P (S n)) -> forall n: nat, P n
```

A parte essencial da resposta acima está na última linha, onde `P` é uma propriedade qualquer dos números naturais. A base de indução diz que `P 0`, e o passo indutivo corresponde ao trecho `(forall n :  nat, P n -> P (S n))`. A conclusão como esperado, diz que `forall n :  nat, P n`.

A seguir vamos refazer a prova dada no exemplo anterior de que a soma dos n primeiros números ímpares é igual a n^2. Inicialmente precisamos definir a função somatório. Antes disto carregamos a biblioteca `Lia`, que vai nos ajudar com a simplificação de expressões aritméticas nos inteiros.

```
Require Import Lia.

Fixpoint msum (n:nat) :=
 match n with
 | 0 => 0
 |S k => (msum k) + (2*k+1)
 end.
```

A palavra reservada `Fixpoint` é utilizada para definir funções recursivas. Note que $\sum_{i=1}^{n}(2.i-1)$ corresponde a `msum n`. Podemos fazer alguns testes com esta definição:

```
Eval compute in (msum 1).
Eval compute in (msum 2).
Eval compute in (msum 3).
Eval compute in (msum 4).
```

A primeira linha retorna 1, que é igual ao primeiro número ímpar. A segunda linha retorna 4, que corresponde a soma dos dois primeiros números ímpares, 1+3. De acordo com estes testes, nossa definição de somatório está funcionando como esperado, e portanto podemos definir o lema que queremos provar, a saber, que a soma dos `n` primeiros números naturais é igual a `n*n`:

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
```

A prova segue a mesma ideia da árvore dada no exemplo anterior. Iniciamos a prova por indução em `n` com a tática `induction n`. Teremos então dois casos para analisar. O primeiro caso é trivial, e a tática `reflexivity` é capaz de concluir que os lados esquerdo e direito da igualdade são iguais a 0. O segundo caso corresponde ao passo indutivo, onde `n` é um sucessor, digamos `(S k)`. Aplicamos a tática `simpl` para simplificar a expressão `msum (S k)` para que possamos usar a hipótese de indução via o comando `rewrite IHn`. A tática `rewrite` nos permite substituir o lado esquerdo pelo lado direito de uma igualdade, ou vice-versa com `rewrite <-`. A expressão resultante é uma igualdade envolvendo soma e multiplicação de números naturais. As simplificações algébricas necessárias para que possamos concluir que os lados esquerdo e direito da igualdade coincidem são feitas pela tática `lia`. Segue o código da prova completa:

```
Lemma msum_square: forall n, msum n = n*n.
Proof.
 induction n.
 - reflexivity.
 - simpl.
   rewrite IHn.
   lia.
Qed.
```

Exercício 74. *Refaça os exercícios 71, 72 e 73 no Coq.*

Existem propriedades que valem apenas para um subconjunto próprio dos números naturais:

Por exemplo, $2^n < n!$ só vale para $n \geq 4$. Para este tipo de problema utilizamos uma generalização do PIM onde a base de indução não precisa ser o 0. Chamaremos esta variação de *Princípio da Indução Generalizado (PIG)*:

$$\frac{P\ m \qquad\qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, n \geq m \Longrightarrow P\ n}\ (PIG)$$

Exemplo 75. *Assim, para provarmos que $2^n < n!, \forall n \geq 4$, precisamos:*

(a) (Base de indução) Mostrar que esta propriedade vale para $n = 4$, o que é trivial, e;

(b) (Passo indutivo) Mostrar que $2^{(S\ k)} < (S\ k)!$ assumindo que $2^k < k!, \forall k \geq 4$. De fato, temos que $2^{(S\ k)} = 2.2^k \overset{(h.i)}{<} 2.k! \overset{(*)}{<} (S\ k).k! = (S\ k)!$, onde a desigualdade (*) se justifica pelo fato de k ser maior ou igual a 4.

Exercício 76. *Faça a prova do exemplo anterior no Coq.*

Exercício 77. *Prove que $2^n - 1$ é múltiplo de 3, para todo número natural n par.*

Agora vamos mostrar que o PIM e o PIG são princípios equivalentes diretamente no Coq.

Exercício 78. *Complete a prova a seguir:*

```
Lemma PIG: forall (P : nat -> Prop) (k : nat), P k ->
              (forall n, n >= k -> P n -> P (S n)) ->
                       forall n : nat, n >= k -> P n.
Proof.
 intros P k H1 IH n H2.
 assert (H := nat_ind (fun n => n >= k -> P n)).
 Admitted.
```

Observe que a prova do exercício anterior utiliza o PIM via o comando `nat_ind`, e portanto temos uma prova de PIG via PIM. No outro sentido, vamos enunciar PIM como um lema:

Exercício 79. *Complete a prova a seguir:*

```
Lemma PIM : forall P: nat -> Prop,
  (P 0) ->
  (forall n, P n -> P (S n)) ->
  forall n, P n.
Proof.
 intros P H IH n.
 apply PIG with 0.
 Admitted.
```

Os dois exercícios anteriores estabelecem a equivalência entre PIM e PIG:

Exercício 80. *Refaça a prova apresentada no Exemplo 75 no Coq.*

Uma variação do PIM bastante útil é conhecida como *Princípio da Indução Forte (PIF)*:

$$\frac{\forall k, (\forall m, m < k \Longrightarrow P\ m) \Longrightarrow P\ k}{\forall n, P\ n}\ (PIF)$$

Exercício 81. *Prove que qualquer inteiro $n \geq 2$ é um número primo ou pode ser escrito como um produto de primos (não necessariamente distintos), i.e. na forma $n = p_1.p_2.\cdots.p_r$, onde os fatores p_i $(1 \leq i \leq r)$ são primos.*

Exercício 82. *Prove a equivalência entre PIM e PIF.*

2. Indução Estrutural

A gramática 2.1 nos diz como as fórmulas da LP podem ser construídas. Observe em particular os construtores recursivos destas gramáticas: por exemplo, a negação de uma fórmula é construída a partir de outra fórmula já construída; a conjunção, a disjunção ou a implicação se constroem a partir de duas fórmulas já construídas. As árvores de derivação das provas anteriormente também são definidas a partir de uma gramática recursiva: uma árvore é um nó, ou construímos uma nova árvore a partir de uma ou mais árvores já construídas. Nesta seção estudaremos como provar

propriedades de elementos de conjuntos definidos recursivamente, como as fórmulas da LP ou as árvores de derivação citadas anteriormente.

Como seria o princípio indutivo associado à gramática 2.1? Este princípio é análogo ao apresentado acima para os naturais considerando que temos uma base de indução para cada construtor não recursivo, e um passo indutivo para cada construtor recursivo. Como os naturais têm apenas um construtor não recursivo (zero), e um recursivo (sucessor), o princípio de indução tem apenas uma base de indução e um passo indutivo. Já a gramática 2.1 que define as fórmulas da LP, possui dois construtores não recursivos (variáveis proposicionais e a constante $\bot$) e quatro construtores recursivos (negação, conjunção, disjunção e implicação), e portanto o princípio indutivo correspondente terá a seguinte forma, considerando uma propriedade Q qualquer das fórmulas da LP:

$$\frac{(Q\ p) \qquad (Q\ \bot) \qquad (\forall \varphi_1, Q\ \varphi \Longrightarrow Q\ (\neg\varphi_1)) \qquad (\forall \varphi_1, Q\ \varphi_1 \wedge \forall \varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))}{\forall \varphi, Q\ \varphi}$$

onde $\star \in \{\wedge, \vee, \rightarrow\}$. Chamamos o princípio de indução construído a partir de uma gramática recursiva de *indução estrutural*.

No exemplo a seguir, vamos mostrar que a gramática acima possui redundâncias, isto é, que existem conectivos que podem ser escritos a partir de outros:

Exemplo 83. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\vee$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Dizemos que duas fórmulas φ e ψ da LP são equivalentes se $\varphi \leftrightarrow \psi$ é uma tautologia. Provaremos este exercício por indução estrutural, isto é, indução na estrutura de φ:

- *Se φ é uma variável proposicional ou a constante $\bot$ então tome $\varphi' = \varphi$.*
- *Se $\varphi = \neg\psi$ então, por hipótese de indução, existe uma fórmula ψ' equivalente a ψ construída apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em ψ. Neste caso, basta tomar $\varphi' = \neg\psi'$, e estamos prontos.*
- *Se $\varphi = \psi_1 \vee \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Neste caso, basta tomar $\varphi' = \psi_1' \vee \psi_2'$ e estamos prontos.*
- *Se $\varphi = \psi_1 \wedge \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Pelo exercício **??** sabemos que $\psi_1 \wedge \psi_2 \dashv\vdash \neg(\neg\psi_1 \vee \neg\psi_2)$. Então basta tomar $\varphi' = \neg(\neg\psi_1' \vee \neg\psi_2')$, e estamos prontos.*
- *Por fim, se $\varphi = \psi_1 \rightarrow \psi_2$ então, por hipótese de indução, existem fórmulas $\psi_i'(i = 1, 2)$, equivalentes respectivamente a $\psi_i(i = 1, 2)$, e construídas apenas com os conectivos $\vee$ e $\neg$, e os símbolos proposicionais que ocorrem em $\psi_i(i = 1, 2)$. Pelo exercício **??** da lista sabemos que $\psi_1 \rightarrow \psi_2 \dashv\vdash (\neg\psi_1) \vee \psi_2$. Então basta tomar $\varphi' = (\neg\psi_1') \vee \psi_2'$ e estamos prontos.*

Agora é a sua vez! Resolva o exercícios a seguir:

Exercício 84. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\rightarrow$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Exercício 85. *Prove, sem utilizar tabela de verdade, que para qualquer fórmula φ, existe uma fórmula φ' equivalente a φ construída apenas com os conectivos $\wedge$ e $\neg$, e com os símbolos proposicionais que ocorrem em φ.*

Baseado no que foi estudado sobre indução estrutural na LP, sabemos como gerar princípios de indução para gramáticas recursivas como 3.2. De fato, no caso dos números naturais temos a gramática:

$$n ::= 0 \mid S\ n$$

e o princípio de indução:

$$\frac{P\ 0 \qquad \forall k, P\ k \Longrightarrow P\ (S\ k)}{\forall n, P\ n}$$

Para a gramática da LP:

$$\varphi ::= p \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi)$$

o princípio gerado foi:

$$\frac{(Q\ p) \qquad (Q\ \bot) \qquad (\forall\varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (\forall\varphi_1, Q\ \varphi_1 \wedge \forall\varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))}{\forall\varphi, Q\ \varphi}$$

onde $\star \in \{\wedge, \vee, \rightarrow\}$.

Exemplo 86. *Considere a gramática da LPO:*

$$\varphi ::= p(t, \ldots, t) \mid \bot \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \exists_x\varphi \mid \forall_x\varphi$$

o princípio de indução correspondente é dado como a seguir:

$$\frac{(\forall t_1, \ldots, t_n, Q\ p(t_1, \ldots, t_n)) \qquad (Q\ \bot) \qquad (\forall\varphi, Q\ \varphi \Longrightarrow Q\ (\neg\varphi)) \qquad (*) \qquad (**)}{\forall\varphi, Q\ \varphi}$$

onde

$(*)$ *é igual a* $(\forall \varphi_1, Q\ \varphi_1 \wedge \forall \varphi_2, Q\ \varphi_2 \Longrightarrow Q\ (\varphi_1 \star \varphi_2))$, $\star \in \{\wedge, \vee, \rightarrow\}$;

$(**)$ *é igual a* $(\forall x, \varphi, Q\ \varphi(x) \Longrightarrow Q\ (R_x \varphi(x)))$, $R \in \{\exists, \forall\}$.

No próximo capítulo estudaremos diversos algoritmos que utilizam a estrutura de lista, definida pela seguinte gramática $l ::= nil \mid a :: l$, onde nil representa a lista vazia, e $a :: l$ representa a lista com primeiro elemento a e cauda l.

Exercício 87. *Escreva o princípio de indução para listas, e em seguida compare sua resposta com o princípio gerado em Coq via o comando* `Print list_ind`.

O comprimento de uma lista, isto é, o número de elementos que a lista possui, é definido recursivamente por:

$$|l| = \begin{cases} 0, & \text{se } l = nil \\ 1 + |l'|, & \text{se } l = a :: l' \end{cases}$$

Uma operação importante que nos permite construir uma nova lista a partir de duas listas já construídas é a concatenação. Podemos definir a concatenação de duas listas por meio da seguinte função recursiva:

$$l_1 \circ l_2 = \begin{cases} l_2, & \text{se } l_1 = nil \\ a :: (l' \circ l_2), & \text{se } l_1 = a :: l' \end{cases}$$

Por fim, o reverso de uma lista é definido recursivamente por:

$$rev(l) = \begin{cases} l, & \text{se } l = nil \\ (rev(l')) \circ (a :: nil), & \text{se } l = a :: l' \end{cases}$$

Os exercícios a seguir refletem diversas propriedades envolvendo estas operações. Resolva estes exercícios manualmente, e em seguida, no Coq.

Exercício 88. *Prove que* $|l_1 \circ l_2| = |l_1| + |l_2|$, *quaisquer que sejam as listas* l_1, l_2.

Exercício 89. *Prove que* $l \circ nil = l$, *qualquer que seja a lista* l.

Exercício 90. *Prove que a concatenação de listas é associativa, isto é,* $(l_1 \circ l_2) \circ l_3) = l_1 \circ (l_2 \circ l_3)$ *quaisquer que sejam as listas* l_1, l_2 *e* l_3.

Exercício 91. *Prove que* $|rev(l)| = |l|$, *qualquer que seja a lista* l.

Exercício 92. *Prove que* $rev(l_1 \circ l_2) = (rev(l_2)) \circ (rev(l_1))$, *quaisquer que sejam as listas* l_1, l_2.

Exercício 93. *Prove que* $rev(rev(l)) = l$, *qualquer que seja a lista* l.

Exercício 94. *Seja φ uma fórmula da lógica de predicados. Definimos a tradução negativa de Gödel-Gentzen de φ, denotada por φ^N, indutivamente por:*

$$\varphi^N = \begin{cases} \neg\neg\varphi & \text{se } \varphi \text{ é uma fórmula atômica, ou a constante } \bot \\ \neg(\psi^N) & \text{se } \varphi = \neg\psi \\ \varphi_1^N \wedge \varphi_2^N & \text{se } \varphi = \varphi_1 \wedge \varphi_2 \\ \neg(\neg(\varphi_1^N) \wedge \neg(\varphi_2^N)) & \text{se } \varphi = \varphi_1 \vee \varphi_2 \\ \varphi_1^N \to \varphi_2^N & \text{se } \varphi = \varphi_1 \to \varphi_2 \\ \forall_x(\psi^N) & \text{se } \varphi = \forall_x\psi \\ \neg(\forall_x\neg(\psi^N)) & \text{se } \varphi = \exists_x\psi \end{cases}$$

Construa uma prova intuicionista para o sequente a seguir: $\neg\neg(\varphi^N) \vdash_i \varphi^N$

Exercício 95. *Uma fórmula da lógica de predicados ϕ pertence ao* fragmento negativo *se ϕ pode ser construída a partir da seguinte gramática, onde $t_1, t_2, \ldots, t_n$ $(n > 0)$ são termos:*

$$\phi ::= \neg p(t_1, t_2, \ldots, t_n) \;||\; \bot \;||\; (\neg\phi) \;||\; (\phi \wedge \phi) \;||\; (\phi \to \phi) \;||\; (\forall_x\phi)$$

Prove na lógica minimal que $\neg\neg\theta \vdash_m \theta$ para qualquer fórmula θ pertencente ao fragmento negativo. Use indução na estrutura de θ.

Exercício 96. *O teorema de Glivenko diz que se $\Gamma \vdash_c \varphi$ então $\Gamma \vdash_i \neg\neg\varphi$ na lógica proposicional, ou seja, se φ tem uma prova clássica a partir de Γ, então $\neg\neg\varphi$ tem uma prova intuicionista a partir de Γ na lógica proposicional. Prove este teorema, isto é, prove que $\Gamma \vdash_c \varphi$ então $\Gamma \vdash_i \neg\neg\varphi$ quais quer que sejam Γ e φ.*

3. Indução na construção de algoritmos

   A construção de algoritmos e a construção de provas são processos similares [25].

3.2 Semântica da Lógica de Primeira Ordem

Estruturas aparecem em matemática rotineiramente. Por exemplo, um grupo é um conjunto não-vazio equipado com duas operações, sendo uma binária, uma unária e com um elemento neutro que satisfaz certas leis. A seguir definiremos formalmente o que são estruturas, mas iniciaremos com a definição de relação:

Definição 97. *Uma relação R n-ária $(n \geq 0)$ sobre um conjunto não-vazio A é um subconjunto de A^n.*

Definição 98. *Uma* estrutura $\mathfrak{A}$ *sobre o conjunto $S = (\mathcal{F}, \mathcal{P})$ (de símbolos de função e de relação, respectivamente) é um par $(A, \mathfrak{a})$ com as seguintes propriedades:*

1. *A é um conjunto não-vazio chamado de* universo *ou* domínio *de $\mathfrak{A}$;*

2. *$\mathfrak{a}$ é uma função definida sobre o conjunto S satisfazendo as seguintes propriedades:*

(a) para cada constante (função da aridade 0), $f \in \mathcal{F}$, associamos um elemento $\mathfrak{a}(f)$ de A;

(b) para cada símbolo de função $f \in \mathcal{F}$ de aridade $n > 0$, $\mathfrak{a}(f)$ é uma função A^n para A;

(c) para cada símbolo de relação (predicado) $p \in \mathcal{P}$ de aridade $n > 0$, $\mathfrak{a}(p)$ é uma relação n-ária em A, ou seja, um subconjunto de A^n.

Ao invés de $\mathfrak{a}(p)$, $\mathfrak{a}(f)$ escreveremos, respectivamente, $p^{\mathfrak{A}}$, $f^{\mathfrak{A}}$. Uma estrutura $\mathfrak{A} = (A, \mathfrak{a})$ sobre os conjuntos $\mathcal{F} = \{f_1, \ldots, f_n\}$ e $\mathcal{P} = \{p_1, \ldots, p_m\}$ será escrita na forma $\mathfrak{A} = (A, p_1^{\mathfrak{A}}, \ldots, p_m^{\mathfrak{A}}, f_1^{\mathfrak{A}}, \ldots, f_n^{\mathfrak{A}})$.

Exemplo 99. *1.* $(\mathbb{R}, +, \cdot, {}^{-1}, 0, 1)$ *- o corpo dos números reais;*

2. $(\mathbb{N}, \leq)$ *- o conjunto ordenado dos números naturais;*

3. $(\mathbb{Z}, <, +, -, 0)$ *- o grupo ordenado dos números inteiros (com a soma usual). Observe que, neste caso $-$ é uma função unária, enquanto que $+$ é binária;*

4. $(\mathbb{Q}, \cdot, {}^{-1}, 1)$ *- o grupo (abeliano) dos números racionais (com a multiplicação usual);*

5. $(\mathbb{Z}, +, \cdot, {}^{-1}, 0, 1)$ *- o anel dos números inteiros.*

Notação: $|\mathfrak{A}| = A$, o universo de $\mathfrak{A}$. A estrutura $\mathfrak{A}$ é dita (in)finita se o seu universo é (in)finito.

Definição 100. *Uma função de atribuição (ou designação) em uma estrutura $\mathfrak{A}$ é uma função $l : var \to A$. Denotaremos por $l[x \mapsto a]$ a função de atribuição que leva x em a e qualquer outro valor y é levado em $l(y)$.*

Definição 101. *Uma interpretação $\mathfrak{I}$ é um par $(\mathfrak{A}, i)$, onde $\mathfrak{A}$ é uma estrutura e i é uma designação.*

Exemplo 102. *Por exemplo, se a interpretação $\mathfrak{I} = (\mathfrak{A}, i)$ é tal que $\mathfrak{A} = (\mathbb{N}, <, +, \cdot, 0, 1)$ e $i(v_n) = 2n$ então a fórmula $v_2 \cdot (v_1 + v_2) = v_4$ é lida como $4 \cdot (2 + 4) = 8$. Já a fórmula $\forall v_0 \exists v_1 (v_0 < v_1)$ é lida como "para todo número natural v_0 existe um número natural v_1 maior do que v_0".*

1. Exercícios

   (a) Considere a interpretação $\mathfrak{I}$ dada no exemplo anterior. Como as fórmulas a seguir são lidas com esta interpretação?

   i. $\exists v_0 (v_0 + v_0 = v_1)$

   ii. $\exists v_0 (v_0 \cdot v_0 = v_1)$

   iii. $\exists v_1 (v_0 = v_1)$

   iv. $\forall v_0 \exists v_1 (v_0 = v_1)$

   v. $\forall v_0 \forall v_1 \exists v_2 (v_0 < v_1 \wedge v_2 < v_1)$

2. Modelos

A noção de satisfação que definiremos nesta seção tornará precisa a noção de quando uma fórmula é verdadeira sob uma dada interpretação. A definição a seguir mostra como termos são interpretados:

Definição 103. *Seja $\mathfrak{I} = (\mathfrak{A}, i)$ uma interpretação. Então:*

(a) para cada variável x, definimos $\mathfrak{I}(x) := i(x)$;

(b) para cada constante c, definimos $\mathfrak{I}(c) := c^{\mathfrak{A}}$;

(c) para cada símbolo de função f de aridade n, e termos $t_1, \ldots, t_n$, definimos $\mathfrak{I}(f(t_1, \ldots, t_n)) := f^{\mathfrak{A}}(\mathfrak{I}(t_1), \ldots, \mathfrak{I}(t_n))$.

Agora podemos definir recursivamente a relação "$\mathfrak{I}$ é um modelo da fórmula ϕ", onde $\mathfrak{I}$ é uma interpretação arbitrária.

Definição 104. *Para toda interpretação $\mathfrak{I} = (\mathfrak{A}, i)$, dizemos que $\mathfrak{I}$ satisfaz a fórmula ϕ (notação $\mathfrak{I} \models \phi$) da seguinte forma:*

- $\mathfrak{I} \models a = b$ *sse* $\mathfrak{I}(a)$ *e* $\mathfrak{I}(b)$ *representam o mesmo elemento do universo A de $\mathfrak{A}$;*
- $\mathfrak{I} \models R\, t_0 \ldots t_{n-1}$ *sse* $R^{\mathfrak{A}}\ \mathfrak{I}(t_0) \ldots \mathfrak{I}(t_{n-1})$, *i.e., se a n-upla* $(\mathfrak{I}(t_0), \ldots, \mathfrak{I}(t_{n-1}))$ *está no subconjunto de A^n que corresponde a interpretação de R;*
- $\mathfrak{I} \models \neg\phi$ *sse não é o caso que* $\mathfrak{I} \models \phi$;
- $\mathfrak{I} \models \phi \wedge \psi$ *sse* $\mathfrak{I} \models \phi$ *e* $\mathfrak{I} \models \psi$;
- $\mathfrak{I} \models \phi \vee \psi$ *sse* $\mathfrak{I} \models \phi$ *ou* $\mathfrak{I} \models \psi$;
- $\mathfrak{I} \models \phi \to \psi$ *sse se* $\mathfrak{I} \models \phi$ *então* $\mathfrak{I} \models \psi$;
- $\mathfrak{I} \models \phi \leftrightarrow \psi$ *sse* $\mathfrak{I} \models \phi$ *se, e somente se,* $\mathfrak{I} \models \psi$;
- $\mathfrak{I} \models \forall_x \phi$ *sse para todo $a \in A$ tal que* $\mathfrak{I}^x_a \models \phi$;
- $\mathfrak{I} \models \exists_x \phi$ *sse existe $a \in A$ tal que* $\mathfrak{I}^x_a \models \phi$.

Dado um conjunto S de fórmulas, dizemos que $\mathfrak{I}$ é um modelo de S (notação $\mathfrak{I} \models S$) se $\mathfrak{I} \models \phi$ para todo $\phi \in S$.

Definição 105. *Seja ϕ uma fórmula da lógica de primeira ordem. Dizemos que ϕ é:*

- satisfatível, *se existe uma interpretação $\mathcal{M}$ que é modelo de ϕ;*
- insatisfatível, *se não possui modelos;*

- válida, *se qualquer interpretação $\mathcal{M}$ é modelo de ϕ;*

- inválida, *se existe uma interpretação $\mathcal{M}$ que não é modelo de ϕ.*

Exemplo 106. *Seja $\mathcal{F} = \{e, \cdot\}$ e $\mathcal{P} = \{\leq\}$, onde e é uma constante, $\cdot$ é uma função binária e $\leq$ é um predicado binário. Considere o modelo $\mathcal{M}$, onde A é o conjunto de todas as* strings *(palavras) sobre o alfabeto $\{0, 1\}$ incluindo a palavra vazia que denotaremos por ϵ. A interpretação $e^{\mathcal{M}}$ corresponde a palavra vazia ϵ, enquanto que $\cdot^{\mathcal{M}}$ corresponde a concatenação de palavras. Por exemplo, $010111 \cdot^{\mathcal{M}} 1100$ é igual a 0101111100. Em geral, se $a_1a_2 \ldots a_k$ e $b_1b_2 \ldots b_r$ são palavras com $a_i, b_j \in \{0, 1\}$ então $a_1a_2 \ldots a_k \cdot^{\mathcal{M}} b_1b_2 \ldots b_r$ é igual a $a_1a_2 \ldots a_kb_1b_2 \ldots b_r$. Finalmente, interpretamos $\leq$ como a relação de prefixo sobre palavras, isto é, $x \leq y$ significa "x é prefixo de y".*

Dizemos que s_1 é um prefixo de s_2 se existir uma palavra s_3 tal que s_2 é igual a $s_1 \cdot^{\mathcal{M}} s_3$.

(a) Em nosso modelo, a fórmula $\forall_x((x \leq x \cdot e) \wedge (x \cdot e \leq x))$ diz que qualquer palavra é um prefixo dela mesma concatenado com a palavra vazia e vice-versa. Esta fórmula claramente é verdadeira em nosso modelo.

(b) Em nosso modelo, a fórmula $\exists_y\forall_x(y \leq x)$ diz que existe uma palavra s que é prefixo de todas as outras palavras. Esta fórmula é verdadeira porque podemos tomar ϵ como sendo a tal palavra. Note que, de fato, esta é a única escolha possível neste caso.

(c) Em nosso modelo, a fórmula $\forall_x\exists_y(y \leq x)$ diz que toda palavra possui um prefixo. Isto também é verdade e, em geral, existem muitas escolhas possíveis que dependem de x.

(d) Em nosso modelo, a fórmula $\forall_x\forall_y\forall_z((x \leq y) \rightarrow (x \cdot z \leq y \cdot z))$ diz que sempre que a palavra s_1 for um prefixo da palavra s_2 então a palavra s_1s tem que ser um prefixo da palavra s_2s para toda palavra s. Esta fórmula é falsa em nosso modelo: basta tomar $s_1 = 01$, $s_2 = 011$ e $s = 0$.

(e) Em nosso modelo a fórmula $\forall_y\exists_x((x \leq y) \rightarrow (y \leq x))$ diz que para qualquer palavra s existe uma palavra s' (que depende de s e) que é prefixo a s e tal que s' é também prefixo de s. Isto é verdadeiro, pois podemos considerar s' como sendo a própria s.

(f) Em nosso modelo a fórmula $\exists_x\forall_y((x \leq y) \rightarrow (y \leq x))$ diz que existe uma palavra s que seja prefixo de (toda) palavra s_1 é tal que s_1 é também prefixo de s. Isto é falso, pois a única palavra que é prefixo de todas as outras é a palavra vazia, no entanto apenas a palavra vazia é prefixo dela mesma.

Exemplo 107. *Seja $\mathcal{F} = \{\texttt{maria}\}$ e $\mathcal{P} = \{\texttt{ama}\}$, onde* ***maria*** *é uma constante e* ***ama*** *é um predicado binário. O modelo $\mathcal{M}$ que estamos construindo consiste de $A = \{a, b, c\}$, da função constante $\texttt{maria}^{\mathcal{M}} = a$ e do predicado $\texttt{ama}^{\mathcal{M}} = \{(a, a), (b, a), (c, a)\}$.*

Queremos verificar se $\mathcal{M}$ satisfaz a seguinte sentença:

> *Nenhuma das pessoas que amam as pessoas que amam* ***maria*** *ama* ***maria****.*

Inicialmente precisamos codificar esta sentença na LPO:

$$\forall_x\forall_y(ama(x, \texttt{maria}) \wedge ama(y, x) \rightarrow \neg ama(y, \texttt{maria}))$$

Escolhendo a para x e b para y é fácil ver que este modelo não satisfaz esta fórmula.

E se considerarmos o modelo $\mathcal{M}'$ onde A permanece inalterado e $\texttt{maria}^{\mathcal{M}'} = \texttt{maria}^{\mathcal{M}}$ e $\texttt{ama}^{\mathcal{M}'} = \{(b,a),(c,b)\}$? Neste caso, a sentença

$$\forall_x \forall_y (ama(x, \texttt{maria}) \wedge ama(y, x) \rightarrow \neg ama(y, \texttt{maria}))$$

é verdadeira em $\mathcal{M}'$.

(a) Exercícios

i. Seja P um símbolo de predicado unário e f uma função binária. Para cada uma das fórmulas $\forall_{v_1} f(v_0, v_1) = v_0$, $\exists_{v_0} \forall_{v_1} f(v_0, v_1) = v_1$ e $\exists_{v_0}(P(v_0) \wedge \forall_{v_1} P(f(v_0, v_1))$ encontre uma interpretação que satisfaz a fórmula e uma que não a satisfaz.

ii. Fórmulas que não contêm $\neg$, $\leftrightarrow$ ou $\rightarrow$ são chamadas de positivas. Mostre que toda fórmula positiva é satisfatível.

3. Consequência lógica

Na lógica proposicional dizemos que ψ é consequência lógica de $\phi_1, \ldots, \phi_n$, denotado por $\phi_1, \ldots, \phi_n \models \psi$, quando ψ é verdadeira sempre que $\phi_1, \ldots, \phi_n$ o forem. Como estender esta noção para a lógica de primeira ordem considerando que $\mathcal{M} \models \psi$?

Definição 108. *Seja Γ um conjunto (possivelmente infinito) de fórmulas da LPO e ψ uma fórmula da LPO.*

(a) A fórmula ψ é consequência lógica do conjunto Γ (notação $\Gamma \models \psi$) sse todo modelo de Γ é também modelo de ψ.

(b) O conjunto Γ é satisfatível (ou consistente) sse existe uma estrutura $\mathcal{M}$ que é modelo de Γ, i.e. $\mathcal{M} \models \Gamma$.

O símbolo $\models$ é utilizado aqui tanto para representar a noção de consequência lógica como para checagem de modelos. Computacionalmente estas duas noções são complicadas:

(a) Verificar que $\mathcal{M} \models \phi$ de forma mecânica (isto é, por uma máquina) pode se tornar muito difícil se o universo A de $\mathcal{M}$ for infinito. Neste caso, checar uma sentença da forma $\forall_x \psi$, onde x ocorre livre em ψ significa verificar $\mathcal{M} \models_{[x \mapsto a]} \psi$ para um número infinito de elementos.

(b) Verificar que $\phi_1, \ldots, \phi_n \models \psi$ vale é ainda mais complicado já que precisaríamos considerar *todos os possíveis modelos* que possuem a estrutura adequada. Lembre-se que na lógica proposicional isto é feito via tabelas-verdade.

Para alguns casos particulares podemos raciocinar sobre a noção de consequência lógica na LPO utilizando argumentos que não dependem de um modelo específico. Infelizmente isto só é possível para um número muito limitado de casos. Vejamos alguns exemplos:

Exemplo 109. *Considere* $\forall_x(p(x) \to q(x)) \models \forall_x p(x) \to \forall_x q(x)$. *Seja* $\mathcal{M}$ *um modelo que satisfaz a fórmula* $\forall_x(p(x) \to q(x))$. *Precisamos mostrar que* $\mathcal{M}$ *satisfaz* $\forall_x p(x) \to \forall_x q(x)$. *Se algum dos elementos de* $\mathcal{M}$ *não satisfaz* p *então* $\forall_x p(x)$ *é falsa e portanto* $\forall_x p(x) \to \forall_x q(x)$ *é verdadeiro, e não há nada a fazer. Caso contrário, isto é, se* $\mathcal{M}$ *satisfaz* $\forall_x p(x)$ *então* $\mathcal{M} \models_{l[x \mapsto a]} p(x)$ *para todo* $a \in A$, *e como* $\mathcal{M}$ *satisfaz* $\forall_x(p(x) \to q(x))$ *temos que* $\mathcal{M} \models_{l[x \mapsto a]} q(x)$ *para todo* $a \in A$. *Portanto* $\mathcal{M}$ *satisfaz* $\forall_x q(x)$. *Assim conseguimos mostrar que* $\forall_x(p(x) \to q(x)) \models \forall_x p(x) \to \forall_x q(x)$ *vale utilizando argumentos que independem do modelo* $\mathcal{M}$.

Exemplo 110. *E quanto a* $\forall_x p(x) \to \forall_x q(x) \models \forall_x(p(x) \to q(x))$? *Agora as coisas ficam muito mais complicadas... Suponha que* $\mathcal{M}'$ *é um modelo que satisfaz* $\forall_x p(x) \to \forall_x q(x)$. *Se* A' *é o universo associado a este modelo e* $p^{\mathcal{M}'}$ *e* $q^{\mathcal{M}'}$ *são as respectivas interpretações de* p *e* q *então* $\mathcal{M}' \models \forall_x p(x) \to \forall_x q(x)$ *simplesmente nos diz que se* $p^{\mathcal{M}'}$ *é igual ao conjunto* A' *então* $q^{\mathcal{M}'}$ *também é igual ao conjunto* A'. *Mas se este não é o caso, então o fato de a implicação* $\forall_x p(x) \to \forall_x q(x)$ *ser verdadeira não nos fornece nenhuma informação adicional. A partir destas observações podemos construir um contra-exemplo: seja* $A' = \{a, b\}$, $p^{\mathcal{M}'} = \{a\}$ *e* $q^{\mathcal{M}'} = \{b\}$. *Então* $\mathcal{M}' \models \forall_x p(x) \to \forall_x q(x)$ *vale, mas* $\mathcal{M}' \models \forall_x(p(x) \to q(x))$ *não vale.*

(a) Exercícios

i. Considere a fórmula $\phi = \forall_x \forall_y (q(g(x, y), g(y, y), z))$. Construa duas interpretações $\mathcal{M}$ e $\mathcal{M}'$ tais que $\mathcal{M} \models \phi$ e $\mathcal{M}' \not\models \phi$.

ii. Considere a sentença $\phi = \forall_x \exists_y \exists_z (p(x, y) \wedge p(z, y) \wedge (p(x, z) \to p(z, x)))$. Quais das seguintes interpretações dadas a seguir satisfazem ϕ?

A. A interpretação $\mathcal{M}$ possui como domínio o conjunto dos números naturais, e $p^{\mathcal{M}} = \{(m, n) \mid m < n\}$;

B. A interpretação $\mathcal{M}'$ possui como domínio o conjunto dos números naturais, e $p^{\mathcal{M}'} = \{(m, 2m) \mid m$ é um número natural$\}$;

C. A interpretação $\mathcal{M}''$ possui como domínio o conjunto dos números naturais com $p^{\mathcal{M}''} = \{(m, n) \mid m < n + 1\}$.

iii. Considere a sentença $\forall_x \neg p(x, x)$. Construa uma interpretação que satisfaz esta sentença e outra que não a
satisfaça.

iv. Considere as seguintes sentenças:

$$\begin{aligned} \phi_1 &= \forall_x p(x, x) \\ \phi_2 &= \forall_x \forall_y (p(x, y) \to p(y, x)) \\ \phi_3 &= \forall_x \forall_y \forall_z ((p(x, y) \wedge p(y, z)) \to p(x, z)) \end{aligned}$$

que expressam a reflexividade, simetria e transitividade do predicado p. Mostre que nenhuma destas sentenças é consequência lógica das outras duas escolhendo, para cada par de sentenças uma interpretação que satisfaça estas duas sentenças, mas não satisfaça a terceira. Essencialmente você deve encontrar três relações binárias onde cada uma satisfaz apenas duas destas propriedades.

v. Mostre que $\forall_x \neg \phi \models \neg \exists_x \phi$. Para isto considere uma interpretação genérica $\mathcal{M}$ e mostre que se $\mathcal{M} \models \forall_x \neg \phi$ então $\mathcal{M} \models \neg \exists_x \phi$ vale.

vi. Seja ϕ sentença $\forall_x \forall_y \exists_z (r(x, y) \to r(y, z))$.

A. Seja $\mathcal{M}$ uma interpretação onde $A = \{a, b, c, d\}$ e $r^{\mathcal{M}} = \{(b, c), (b, b), (b, a)\}$. É o caso que $\mathcal{M} \models \phi$?

B. Seja $\mathcal{M}'$ uma interpretação onde $A' = \{a, b, c\}$ e $r^{\mathcal{M}'} = \{(b, c), (a, b), (c, b)\}$. É o caso que $\mathcal{M}' \models \phi$?

vii. É o caso que $\forall_x(p(x) \vee q(x)) \models \forall_x p(x) \vee \forall_x q(x)$? Em caso afirmativo, construa uma prova, e em caso negativo justifique sua resposta.

viii. É o caso que $\exists_x(p(x) \vee q(x)) \models \exists_x p(x) \vee \exists_x q(x)$? Em caso afirmativo, construa uma prova, e em caso negativo justifique sua resposta.

ix. Considere as seguintes sentenças:

$$\forall_x(\exists_y(x + y = 0) \wedge \exists_z(z + x = 0))$$
$$\forall_x((x + 0 = x) \wedge (0 + x = x))$$
$$\forall_x\forall_y\forall_z(x + (y + z) = (x + y) + z)$$

Seja γ a conjunção destas três sentenças.

A. Mostre que γ é satisfatível.

B. Mostre que γ não é uma tautologia.

C. Mostre que a sentença $\forall_x\forall_y(x + y = y + x)$ não é consequência lógica de γ.

x. Determine se os sequentes a seguir são válidos ou não, e justifique sua resposta:

A. $\vdash (\forall_x\exists_y p(x, y)) \rightarrow (\exists_y\forall_x p(x, y))$

B. $\vdash (\exists_y\forall_x p(x, y)) \rightarrow (\forall_x\exists_y p(x, y))$

3.3 Indecidibilidade da LPO

Nesta seção provaremos que o ganho de expressividade obtido ao passarmos da lógica proposicional para a lógica de primeira ordem vem com um custo. Sabemos que, dada uma fórmula φ da lógica proposicional, podemos (pelo menos teoricamente) decidir se φ é válida ou não: basta construirmos a tabela verdade de φ. Ainda que este processo seja ineficiente, já que cresce exponencialmente de acordo com o número de variáveis proposicionais que ocorrem em φ, ele nos fornece um algoritmo para decidir a validade na lógica proposicional. Esta ideia não pode ser utilizada na LPO, e como veremos o custo a ser pago com o ganho de expressividade é que a validade na LPO passa a ser um problema indecidível.

A prova da indecidibilidade da validade na LPO, também conhecido como *indecidibilidade da LPO*, será feita reduzindo o problema da validade na LPO a outro problema que já é conhecidamente indecidível: O problema da correspondência de Post (PCP), que pode ser enunciado como a seguir.

Dada uma sequência finita de pares $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$ tal que todos os s_i's e t_i's são palavras binárias de comprimento positivo, existe uma sequência de índices $i_1, i_2, \ldots, i_n$ com $n \geq 1$ (que podem ser repetidos) tal que a concatenação das palavras $s_{i_1} s_{i_2} \ldots s_{i_n}$ é igual a $t_{i_1} t_{i_2} \ldots t_{i_n}$?

Um exemplo de uma instância deste problema é $(1, 101), (10, 00), (011, 11)$. Note que esta instância possui a sequência $(1, 3, 2, 3)$ como solução. Observe que nosso espaço de busca é infinito, e isto nos dá uma certa intuição da razão pela qual este problema não é solúvel em geral.

Assumindo então a indecidibilidade do PCP, provaremos que a noção de validade na LPO também é indecidível conforme [19]. Iniciaremos definindo a noção redutibilidade entre problemas:

Definição 111. *Um problema A é redutível para um problema B se existe uma função (total) computável $f : A \to B$ tal que:*

$$x \in A \iff f(x) \in B.$$

Nossa prova consiste em reduzir o PCP para o problema da validade na LPO, construindo uma função computável que a cada instância do PCP retorna uma fórmula da LPO. Desta forma, se validade na LPO fosse decidível poderíamos construir um algoritmo para decidir PCP da seguinte forma:

- para cada instância C de PCP construímos uma fórmula ϕ_C tal que ϕ_C é válida se, e somente se, C possui uma solução.

Teorema 112. *A LPO é indecidível.*

Demonstração. A prova consiste em dada uma instância C do problema da correspondência de Post, construir em espaço e tempo finitos uma fórmula ϕ_C da LPO tal que $\models \phi_C$ se, e somente se, a instância C tem uma solução. Seja C a sequência $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$. Consideramos como símbolos de função e, f_0, f_1 de aridade 0, 1, 1 respectivamente. Interpretamos e como sendo a palavra vazia, e f_0 e f_1 como sendo a concatenação com 0 e 1, respectivamente. Assim, se $b_1 b_2 \ldots b_l$ é uma palavra binária então podemos codificá-la como sendo o termo $f_{b_l}(f_{b_{l-1}} \ldots (f_{b_2}(f_{b_1}(e))) \ldots)$. Para facilitar a leitura das fórmulas abreviaremos um termo da forma $f_{b_l}(f_{b_{l-1}} \ldots (f_{b_2}(f_{b_1}(t))) \ldots)$ por $f_{b_1 b_2 \ldots b_l}(t)$. Também utilizaremos um predicado binário p, onde $p(s, t)$ significa "existe uma sequência de índices $(i_1, i_2, \ldots, i_m)$ tal que s é o termo representando $s_{i_1} s_{i_2} \ldots s_{i_m}$ e t é o termo representando $t_{i_1} t_{i_2} \ldots t_{i_m}$". Ou seja, s constrói uma palavra utilizando a mesma sequência de índices utilizada por t. Nossa fórmula ϕ_C possui a seguinte estrutura: $(\phi_1 \wedge \phi_2) \to \phi_3$, onde:

$$\begin{aligned}
\phi_1 &:= \bigwedge_{i=1}^{k} p(f_{s_i}(e), f_{t_i}(e)) \\
\phi_2 &:= \forall_v \forall_w (p(v, w) \to \bigwedge_{i=1}^{k} p(f_{s_i}(v), f_{t_i}(w))) \\
\phi_3 &:= \exists_z p(z, z)
\end{aligned}$$

Afirmação: $\models \phi_C$ sse a instância C do problema da correspondência de Post tem uma solução.

Inicialmente vamos assumir que $\models \phi_C$. Nossa estratégia é encontrar um modelo $\mathcal{M}$ para ϕ_C que nos diga que existe uma solução para a instância C por simples inspeção do significado da satisfatibilidade de ϕ_C para $\mathcal{M}$. O universo A de $\mathcal{M}$ é o conjunto de todas as palavras binárias finitas incluindo a palavra vazia (que denotaremos por ϵ), isto é $A = \{0, 1\}^*$. A interpretação $e^{\mathcal{M}}$ da constante e é a palavra vazia ϵ. A interpretação $f_0^{\mathcal{M}}$ de f_0 é a concatenação de 0 a uma dada palavra: $f_0^{\mathcal{M}}(s) = s0$. Analogamente, $f_1^{\mathcal{M}}(s) = s1$. Por fim,

$$\begin{aligned}
p^{\mathcal{M}} := \{&(s, t) \mid \text{ existe uma sequência de índices} \\
&(i_1, i_2, \ldots, i_m) \text{ tais que } s \text{ é igual a} \\
&s_{i_1} s_{i_2} \ldots s_{i_m} \text{ e } t \text{ é igual a } t_{i_1} t_{i_2} \ldots t_{i_m}\}
\end{aligned}$$

onde s e t são palavras binárias e s_i e t_i são dados de C.

Por hipótese temos que $\models \phi$, e portanto $\mathcal{M} \models \phi$. Mostraremos que $\mathcal{M} \models \phi_3$, de onde podemos concluir que a instância C possui uma solução.

Afirmação 1: $\mathcal{M} \models \phi_1$. De fato, a sequência unitária (i) é tal que $p(f_{s_i}(e), f_{t_i}(e))$ é verdadeiro para todo $i = 1, \ldots, k$.

Afirmação 2: $\mathcal{M} \models \phi_2$. De fato, $(s,t) \in p^{\mathcal{M}}$ implica que existe um sequência $(i_1, i_2, \ldots, i_m)$ tal que s é igual a $s_{i_1} s_{i_2} \ldots s_{i_m}$ e t é igual a $t_{i_1} t_{i_2} \ldots t_{i_m}$.

Escolhendo a sequência $(i_1, i_2, \ldots, i_m, i)$ temos que ss_i é igual a $s_{i_1} s_{i_2} \ldots s_{i_m} s_i$ e tt_i é igual a $t_{i_1} t_{i_2} \ldots t_{i_m} t_i$, e portanto $\mathcal{M} \models \phi_2$. Temos que $\mathcal{M} \models (\phi_1 \wedge \phi_2) \rightarrow \phi_3$ e $\mathcal{M} \models \phi_1 \wedge \phi_2$, e portanto $\mathcal{M} \models \phi_3$. Pela definição de ϕ_3 e $p^{\mathcal{M}}$, concluímos que existe uma solução para C.

Reciprocamente, suponha que a instância C dada do problema da correspondência de Post possui uma solução, a saber $(i_1, i_2, \ldots, i_n)$. Precisamos provar que se $\mathcal{M}'$ é um modelo qualquer contendo a constante $e^{\mathcal{M}'}$, duas funções unárias $f_0^{\mathcal{M}'}$ e $f_1^{\mathcal{M}'}$, e um predicado binário $p^{\mathcal{M}'}$ então $\mathcal{M}'$ satisfaz ϕ. Como ϕ é uma implicação, não há nada a fazer se $\mathcal{M}' \not\models \phi_1$ ou $\mathcal{M}' \not\models \phi_2$. A parte interessante é mostrar que $\mathcal{M}' \models \phi_3$ sempre que $\mathcal{M}' \models \phi_1 \wedge \phi_2$. Faremos isto interpretando palavras binárias finitas no domínio A' de $\mathcal{M}'$:

$$\begin{array}{lcl} i(\epsilon) & := & e^{\mathcal{M}'} \\ i(s0) & := & f_0^{\mathcal{M}'}(i(s)) \\ i(s1) & := & f_1^{\mathcal{M}'}(i(s)) \end{array}$$

Note que a função $i : \{0,1\}^* \rightarrow A'$ é definida indutivamente sobre o comprimento de s.

A função de interpretação i aplica as funções $f_0^{\mathcal{M}'}$ e $f_1^{\mathcal{M}'}$ de forma reversa. Por exemplo, a palavra 0100110 é interpretada como $f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_0^{\mathcal{M}'}(f_1^{\mathcal{M}'}(f_0^{\mathcal{M}'}(e^{\mathcal{M}'}))))))$. Note que $i(b_1 b_2 \ldots b_l) = f_{b_l}^{\mathcal{M}'}(f_{b_{l-1}}^{\mathcal{M}'}(\ldots(f_{b_1}^{\mathcal{M}'}(e^{\mathcal{M}'}))\ldots))$ corresponde ao significado dado para $f_s(e)$ em A', onde s corresponde a $b_1 b_2 \ldots b_l$. Sendo assim, e sabendo que $\mathcal{M}' \models \phi_1$ concluímos que $(i(s_i), i(t_i)) \in p^{\mathcal{M}'}$ para todo $i = 1, 2, \ldots, k$. Analogamente, como $\mathcal{M}' \models \phi_2$ concluímos que para todo $(s,t) \in p^{\mathcal{M}'}$ temos que $(i(ss_i), i(tt_i)) \in p^{\mathcal{M}'}$ para todo $i = 1, 2, \ldots, k$. Iniciando com $(s,t) = (s_{i_1}, t_{i_1})$ podemos utilizar o fato anterior repetidamente para obter que

$$(i(s_{i_1} s_{i_2} \ldots s_{i_n}), i(t_{i_1} t_{i_2} \ldots t_{i_n})) \in p^{\mathcal{M}'}.$$

Como as palavras $s_{i_1} s_{i_2} \ldots s_{i_n}$ e $t_{i_1} t_{i_2} \ldots t_{i_n}$ formam uma solução de C estas palavras são iguais. Sendo assim, $i(s_{i_1} s_{i_2} \ldots s_{i_n})$ e $i(t_{i_1} t_{i_2} \ldots t_{i_n})$ representam o mesmo elemento de A'. Logo $\mathcal{M}' \models \exists_z p(z,z)$, isto é, $\mathcal{M}' \models \phi_3$. Logo validade na LPO é um problema indecidível.

□

Parte II

Algoritmos

Nesta seção vamos utilizar os temas abordados nos capítulos anteriores para analisar algoritmos. Os computadores e seus algoritmos estão presentes na maioria das atividades quotidianas, utilizamos computadores para fazer compras, transações bancárias, etc. Os carros, aviões e equipamentos hospitalares modernos também possuem diversos sistemas embarcados. À medida que estes equipamentos se tornam mais comuns em nosso dia a dia, aumenta também o seu poder de processamento e de armazenamento. Diante desta realidade é natural perguntar: por que analisar algoritmos? Inicialmente porque precisamos garantir que são corretos. Mas o que significa dizer que um algoritmo é correto? Intuitivamente, significa que o algoritmo sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se AlgOrd é um algoritmo de ordenação de inteiros, então espera-se que para qualquer vetor de inteiros A dado, AlgOrd(A) retorne uma permutação de A que esteja ordenada. Isto significa que as respostas dadas pelo algoritmo são corretas para todas as entradas possíveis, e que estas respostas são geradas em tempo finito. Como veremos, em alguns casos a prova de correção é simples, mas em geral esta é uma tarefa complexa. Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática estudada no capítulo anterior.

Adicionalmente, precisamos analisar a eficiência destes algoritmos. Mas não poderíamos simplesmente migrar para um computador mais potente e com mais capacidade de armazenamento quando fosse necessário? A resposta é não. Ainda que tivéssemos uma capacidade infinita de processamento e/ou de armazenamento, a análise da eficiência seria necessária. De fato, não faz sentido ter que esperar horas para a finalização de um processamento se for possível fazê-lo de forma mais eficiente em apenas alguns segundos, ou utilizar uma quantidade gigantesca de memória sem necessidade. Estudaremos diversos problemas ao longo deste curso, e veremos situações em que uma abordagem ingênua (força bruta) requer um número exponencial de operações, enquanto que uma abordagem cuidadosa pode diminuir substancialmente o número de operações necessárias para resolver o mesmo problema. Ao analisarmos a eficiência dos algoritmos também faremos o uso de diversas ferramentas matemáticas (somatórios, conjuntos, funções, matrizes, etc). O apêndice VIII do livro [12] pode ser usado para revisar estes temas. Já a análise da complexidade de tempo e/ou complexidade de espaço de um algoritmo consiste no estudo sistemático do número de operações realizadas, ou espaço extra demandado, durante a execução do algoritmo. Assim, o resultado desta análise deve indicar o tempo de execução esperado para uma entrada particular. No entanto, não é possível listar o tempo esperado para todas as entradas possíveis a menos que o algoritmo seja muito simples, ou que o número de entradas possíveis seja finito. Para contornar este problema utilizaremos uma medida para a entrada, que chamaremos de *tamanho da entrada*, para fazer a análise. Como veremos, esta medida será de grande utilidade, ainda que um algoritmo possa ter comportamento diferente para entradas de mesmo tamanho. Também não definiremos uma noção geral de tamanho da entrada para todos os algoritmos porque estamos interessados em comparar diferentes algoritmos que resolvam o mesmo problema. De forma resumida, dado um problema e uma definição de tamanho para a entrada, queremos construir uma expressão que nos forneça o tempo de execução do algoritmo relativo ao tamanho da entrada. Como um algoritmo pode ter comportamento diferente para entradas de mesmo tamanho, precisamos escolher uma das possíveis entradas para expressar o custo de execução do algoritmo. Normalmente, a escolha é pela entrada que representa o pior caso, ou seja, o maior custo possível.

Capítulo 4

Análise Assintótica

O ponto de partida do estudo que faremos consiste em determinar a quantidade de recursos demandados por um algoritmo em função do tamanho das instâncias, e neste contexto, é natural esperar que quanto maior for a instância a ser resolvida, maior também será a quantidade de recursos demandados. Os recursos que estamos interessados em investigar são basicamente o tempo de computação e o espaço de armazenamento (ou memória). Assim, considerando novamente o contexto de ordenação de listas (ou vetores), quanto maior for a lista a ser ordenada (instância), maior será o tempo (recurso) demandado. Isto sugere então que para o problema de ordenação, o tamanho das instâncias seja justamente o número de elementos da lista (ou vetor) a ser ordenado.

O parâmetro n que denota o tamanho da entrada também precisa ser fornecido a partir de uma medida adequada para que possamos fazer uma análise concisa. Para o caso de ordenação de uma lista (ou vetor), vimos que o número de elementos da lista representa uma medida adequada, e a tabela abaixo apresenta outros exemplos:

Problema	**Tamanho da entrada**
Busca em um vetor	Tamanho do vetor
Multiplicação de duas matrizes	Dimensão das matrizes
Busca em grafos	Tamanho da representação do grafo[1]

O modelo computacional que utilizaremos é o de uma máquina de acesso aleatório (*random-access machine* - RAM) com um processador, e devemos lembrar que nossos algoritmos serão implementados como programas neste modelo, onde as instruções são executadas sequencialmente, sem operações concorrentes[12, 24, 22, 5, 7]. As instruções no modelo RAM são as encontradas em computadores reais:

- aritmética (soma, subtração, multiplicação, divisão, resto, piso e teto);
- movimento de dados (*load*, *store*, *copy*);
- controle (ramos condicionais e não-condicionais, chamadas a subprocedimentos e retorno).

Assumiremos que cada uma destas instruções é executada em tempo constante. Além disto, os tipos abstratos de dados deste modelo são os números inteiros e números em ponto flutuante.

Vejamos agora os fundamentos para a análise da eficiência de algoritmos. O que significa dizer que um algoritmo é eficiente? Podemos analisar a eficiência de um algoritmo de duas formas: eficiência temporal e eficiência espacial. No primeiro caso, estamos interessados no tempo de execução do algoritmo,

[4]O tamanho da representação normalmente é determinado a partir do número de vértices e número de arestas do grafo.

enquanto que no segundo caso, queremos analisar a quantidade de espaço extra (memória) que é utilizado pelo algoritmo durante sua execução. A forma de determinar a eficiência de um algoritmo deve permitir a comparação de algoritmos distintos que resolvam o mesmo problema. Inicialmente, poderíamos pensar em utilizar o tempo de execução de um programa que implementa um algoritmo, mas esta não é uma boa medida porque depende tanto do computador (*hardware*) quanto da implementação (*software*). Precisamos de um método que nos informe sobre a eficiência do algoritmo independentemente do computador em que ele venha a ser implementado, da linguagem de programação e do estilo de programação utilizados. O método deve ser preciso e geral de forma que possa ser utilizado para diversos algoritmos e aplicações. Uma ideia inicial é contar todas as operações realizadas pelo algoritmo, mas veremos que esta abordagem possui alguns problemas, além de ser muito trabalhosa.

4.1 Busca sequencial

Como primeiro exemplo, considere o problema de buscar um elemento em um vetor arbitrário. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n-1]$ contendo n elementos e o valor x procurado, e faz a buscar sequencial de x em A: se A possui uma ocorrência de x então o algoritmo retorna a posição $0 \leq i \leq n-1$ da primeira ocorrência encontrada. Caso contrário, o algoritmo retorna o valor -1.

```
1 i ← 0;
2 while i < n and A[i] ≠ x do
3 |   i ← i + 1;
4 end
5 if i < n then
6 |   return i;
7 else
8 |   return -1
9 end
```

Algorithm 1: SequentialSearch($A[0..n-1], x$)

Parece bastante intuitivo dizer que este algoritmo é correto, mas como **provar** isto? Inicialmente temos que expressar a noção de correção por meio de um teorema e em seguida, precisamos provar este teorema. Construiremos uma invariante para o laço **while** que nos permita concluir a correção ao final da execução do algoritmo. Uma invariante é uma propriedade que é verdadeira antes da execução do algoritmo (inicialização), permanece verdadeira durante a execução do algoritmo (manutenção) e tal que sua validade ao final da execução do algoritmo nos permite concluir sua correção (finalização). Queremos mostrar que ao final da execução, o algoritmo SequentialSearch retorna -1 se x não ocorre em A, e retorna $0 \leq i \leq n-1$ se a primeira ocorrência de x é na posição i do vetor A. Precisamos adaptar esta propriedade de forma que possa ser utilizada ao longo da execução do algoritmo, e não apenas ao final de sua execução. Sabemos que uma nova iteração do laço **while** só ocorrerá se o elemento x não foi ainda encontrado, pois caso contrário o algoritmo para. A partir destas observações, considere a seguinte invariante para o laço **while** do algoritmo SequentialSearch:

> Antes da i-ésima iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x.

A prova de uma invariante é construída pelos 3 passos citados acima:

Demonstração. A prova é dividida em 3 passos:

- **Inicialização**: Precisamos mostrar que antes da primeira iteração do laço **while**, o subvetor $A[0..i-1]$ não possui ocorrências de x. Este passo é trivial porque antes da primeira iteração, temos que $i = 0$, e portanto o subvetor $A[0..i-1]$ é vazio.

- **Manutenção**: Este é o passo que exige mais cuidado na prova. Observe que antes da primeira iteração o valor de i é 0. Considerando que as condições do laço sejam satisfeitas, i será incrementado, e portanto antes da segunda iteração o valor de i é 1, e assim sucessivamente. Logo, antes da k-ésima iteração $k > 1$, o valor de i é $k - 1$ e podemos assumir por hipótese que o subvetor $A[0..k-2]$ não possui ocorrências de x. Para que a próxima iteração ocorra, precisamos que $k < n$ e $A[k-1] \neq x$. Nestas condições, temos que o subvetor $A[0..k-1]$ não possui ocorrências de x preservando assim, a invariante.

- **Finalização**: Ao final da execução do laço, a condição "$i < n$ and $A[i] = x$ $(0 \leq i < n)$"não é mais satisfeita, e portanto temos que $i \geq n$ ou $A[i] = x$ $(0 \leq i < n)$. Se $i \geq n$ então o vetor A não possui ocorrências de x e o algoritmo retorna -1 de acordo com a linha 5. Se $A[i] = x$ $(0 \leq i < n)$ então a posição i é retornada, uma vez que o elemento procurado está na posição i do vetor A. Isto finaliza a prova de correção do algoritmo SequentialSearch.

□

Agora vamos analisar a complexidade em tempo e espaço deste algoritmo. Observe que a execução do algoritmo não demanda espaço adicional, ou seja, o espaço utilizado para a sua execução é o espaço alocado para armazenar o vetor A e nada mais. Neste caso, dizemos que a complexidade em espaço do algoritmo SequentialSearch é constante. Uma complexidade, seja em tempo ou espaço, constante é a melhor situação que podemos ter, ou seja, a mais eficiente possível. As classes básicas de eficiência que utilizaremos para analisar algoritmos são listadas a seguir em ordem crescente de complexidade em função do tamanho n da entrada:

Classe	**Nome**
1	constante
$\log n$	logarítmica
n	linear
$n.\log n$	linearítmica
n^2	quadrática
n^3	cúbica
n^k	polinomial ($k \geq 1$ e finito)
a^n	exponencial ($a \geq 2$)
$n!$	fatorial

A análise da complexidade de tempo do algoritmo SequentialSearch não é tão imediata quanto a análise feita para a complexidade de espaço, ainda que seja simples. Podemos começar com a seguinte pergunta: qual o custo de execução de cada linha do algoritmo SequentialSort? A linha 1 faz uma atribuição, cujo custo não depende do tamanho n do vetor A, e portanto é razoável dizer que este custo é constante, digamos c_1, uma constante positiva. Observe que esta constante não depende do parâmetro n, mas do computador e da linguagem de programação. As linhas 2-4 constituem um laço cujo corpo contém apenas uma atribuição. Ainda que o custo da linha 3 possa ser o mesmo da linha 1, vamos denotá-lo pela constante positiva c_3. Quantas vezes a linha 3 é executada? Isto depende tanto do vetor A quanto da chave x. De fato, se x ocorre na primeira posição de A, isto é, se $A[0]$ é igual a x então a condição do laço é executada uma única vez, mas a linha 3 não é executada nenhuma vez independente de existirem outras ocorrências de x em A. Esta é a situação constitui o melhor caso possível, e por isso é chamada de *análise do melhor caso*. Se $A[0] \neq x$ e x ocorre na segunda posição de A então a linha 3 é executada uma única vez, enquanto que a linha 2 é executada duas vezes. Em geral, observe que a linha que define um laço é sempre executada uma vez a mais do que as linhas que compõem o seu corpo. Por fim, se x não ocorre no vetor A então a linha 2 será executada $n + 1$ vezes enquanto que a linha 3 será executada n vezes. Esta situação vai configurar a *análise do pior caso*. Por fim, o condicional da linha 5 será executado uma única vez a um custo constante, digamos c_5, e apenas uma das linhas 6 ou 8 será executada uma única vez. Juntando todas estas informações podemos então dividir a análise em 2 casos:

1. Análise do melhor caso na busca sequencial

   Como vimos anteriormente, o melhor caso ocorre quando o elemento procurado ocorre na primeira posição do vetor A. Os custos associados por linha nesta situação são apresentados na seguinte tabela:

Linha	Custo	Observação
1	c_1	
2	c_2	
3	0	não é executada
5	c_5	
6	c_6	
8	0	não é executada
Total	$c_1 + c_2 + c_5 + c_6$	

   Denotando por $T_b(n)$ o custo no melhor caso (*best case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_b(n) = c_1 + c_2 + c_5 + c_6$. Neste caso dizemos que o custo da busca sequencial é constante em função do tamanho n da entrada.

2. Análise do pior caso na busca sequencial

   Agora vamos compilar as informações discutidas anteriormente considerando que o laço da linha 2 é executado o maior número de vezes possível, o que acontece quando o elemento procurado não ocorre no vetor:

Linha	Custo
1	c_1
2	$c_2.(n+1)$
3	$c_3.n$
5	c_5
6	0
8	c_8
Total	$c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$

   Denotando por $T_w(n)$ o custo no pior caso (*worst case*) para a busca sequencial considerando que o vetor A possui n elementos, temos que $T_w(n) = c_1 + c_2.(n+1) + c_3.n + c_5 + c_8$. Neste caso dizemos que o custo da busca sequencial é linear em função do tamanho n da entrada. Antes de refinarmos a análise e apresentarmos as definições precisas das análises de melhor e pior caso, vejamos um outro exemplo considerando agora o problema da ordenação de um vetor.

4.2 O algoritmo de ordenação por inserção (*insertion sort*)

Nesta seção estamos interessados em ordenar $n > 0$ números naturais em ordem crescente. Suponha que estes números estão armazenados no vetor $A[0..n-1]$. Então, ao final do processo queremos obter uma permutação de $A[0..n-1]$, digamos $A'[0..n-1]$ tal que $A'[i-1] \leq A'[i]$, para todo $1 \leq i < n$. Estudaremos diversas formas distintas de abordar este problema nas próximas seções, mas aqui vamos iniciar com o algoritmo de ordenação por inserção (*insertion sort*), cujo pseudocódigo é dado a seguir:

```
1 for  j = 1 to n − 1 do
2 |   key ← A[j];
3 |   i ← j − 1;
4 |   while  i ≥ 0 and A[i] > key do
5 |   |   A[i + 1] ← A[i];
6 |   |   i ← i − 1;
7 |   end
8 |   A[i + 1] ← key;
9 end
```

Algorithm 2: InsertionSort($A[0..n-1]$)

A primeira pergunta que precisamos responder é: este algoritmo é correto? Isto é, ele satisfaz as especificações do problema que propõe resolver?

1. A correção do algoritmo de ordenação por inserção

   Queremos ordenar os elementos de um vetor, assim esperamos que os elementos do vetor gerado após a execução do algoritmo coincidam com os elementos do vetor original, e que o vetor resultante esteja ordenado. Como vimos no exemplo anterior, em algoritmos iterativos utilizamos as invariantes de laço para estabelecer a correção do algoritmo. Dada a dinâmica do algoritmo InsertionSort, considere a seguinte invariante de laço:

   > Antes da j-ésima iteração do laço **for** (linhas 1-9), o subvetor $A[0..j-1]$ está ordenado e contém os mesmos elementos do vetor original $A[0..j-1]$.

   Assim, se esta propriedade for válida ao final da execução do laço **for**, *i.e.* antes da $n+1$-ésima iteração, teremos que o vetor gerado consiste dos elementos do vetor original $A[0..n-1]$ ordenado. Isto corresponde a dizer que InsertionSort é correto.

   Como então provar esta invariante para InsertionSort? A prova é por indução no número de iterações do laço **for**:

   - **Inicialização** (Base da indução):
     Antes da primeira iteração do laço **for**, temos que $j = 1$ (condição necessária para iniciar o laço), e portanto a invariante é trivial porque o subvetor unitário $A[0]$ está ordenado por definição.

   - **Manutenção** (Passo indutivo):
     Considere a k-ésima iteração, isto é, $j = k$ $(1 < k < n)$. Temos como hipótese que "Antes da k-ésima iteração do laço **for** o subvetor $A[0..k-1]$ é uma permutação que está ordenada do subvetor original $A[0..k-1]$."Assim, durante a k-ésima iteração, o laço **while** vai deslocar cada elemento maior do que $A[k]$, i.e. key, uma posição para a direita até encontrar a posição correta onde o elemento $A[k]$ deve ser inserido, de forma que neste momento o subvetor $A[0..k]$ está ordenado e possui os mesmos elementos do subvetor $A[0..k]$ original. A incrementarmos o valor de k para a próxima iteração, a invariante é reestabelecida. Informalmente estamos dizendo que o laço **while** encontra a posição correta para inserir $A[j]$ (que está armazenado na variável key). Provaremos este fato com a ajuda de uma invariante para o laço **while**:

     > Antes de cada iteração do laço **while**, o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key.

     A prova é também por indução no número de iterações do laço **while**:

(a) **Inicialização**: Antes da primeira iteração do **while** temos que $i+1 = j = k$, e como $key = A[j]$ a invariante está satisfeita.

(b) **Manutenção**: Por hipótese de indução temos que o subvetor $A[i+1..j]$ possui elementos que são maiores ou iguais a key. Durante uma iteração do laço, o elemento $A[i]$ é copiado na posição $i+1$ do vetor A, e portanto a invariante continua valendo.

(c) **Finalização**: Ao final da execução do laço, temos que i é, de fato, a posição correta para inserir o elemento $A[k]$ já que todos os elementos do subvetor $A[i+1..j]$ são maiores ou iguais a key. É importante observar que a inserção do elemento $A[k]$ na posição i não elimina nenhum elemento do vetor original porque o elemento que está na posição i foi copiado para a posição $i+1$, se o laço **while** foi executado pelo menos uma vez, ou ele é o próprio elemento armazenado em key, quando o laço não é executado.

- **Finalização**: Ao final da execução do laço **for**, temos $j = n$, e portanto a invariante corresponde a dizer que o vetor $A[0..n-1]$ obtido ao final da execução do algoritmo está ordenado, e é uma permutação do vetor original $A[0..n-1]$. Assim, concluímos a prova da correção do algoritmo InsertionSort.

Exercício 113. *Prove que o algoritmo BubbleSort a seguir é correto.*

```
1 for i = 0 to n − 2 do
2 |   for j = 0 to n − 2 − i do
3 |   |   if A[j + 1] < A[j] then
4 |   |   |   swap A[j] and A[j + 1];
5 |   |   end
6 |   end
7 end
```

Algorithm 3: BubbleSort($A[0..n-1]$)

Exercício 114. *Prove que o algoritmo BubbleSort2 [12] a seguir é correto.*

```
1 for i = 0 to n − 2 do
2 |   for j = n − 1 downto i + 1 do
3 |   |   if A[j] < A[j − 1] then
4 |   |   |   swap A[j] and A[j − 1];
5 |   |   end
6 |   end
7 end
```

Algorithm 4: BubbleSort2($A[0..n-1]$)

Exercício 115. *Prove que o algoritmo SelectionSort a seguir é correto.*

```
1 for i = 0 to n − 2 do
2 |   min ← i;
3 |   for j = i + 1 to n − 1 do
4 |   |   if A[j] < A[min] then
5 |   |   |   min ← j;
6 |   |   end
7 |   end
8 |   swap A[i] and A[min];
9 end
```

Algorithm 5: SelectionSort($A[0..n-1]$)

2. A complexidade do algoritmo de ordenação por inserção

Agora faremos uma análise da complexidade do algoritmo de ordenação por inserção semelhante análoga à feita para a busca sequencial. Certamente, ordenar um vetor com 1000 demanda mais tempo do que ordenar apenas 3 elementos, assim é usual descrever o tempo de execução de um algoritmo em função do tamanho da entrada que neste caso é o número n de elementos a serem ordenados. Novamente assumiremos que cada linha do pseudocódigo é executada em tempo constante, mas este tempo pode diferir de uma linha para outra. Assim, denotaremos por c_i a constante que corresponde ao tempo de execução da i-ésima linha do pseudocódigo. Vejamos, então, o custo de execução do algoritmo InsertionSort. O laço **for** da linha 1 é executado n vezes, enquanto que o corpo do laço é executado $n-1$ vezes, uma vez para cada $j = 1, \ldots, n-1$. Denotaremos por t_j o número de vezes que o teste do laço **while** da linha 4 é executado, de forma que temos o seguinte custo por linha:

Linha	Custo	Número de execuções	Custo total
1	c_1	n	$c_1.n$
2	c_2	$n-1$	$c_2.(n-1)$
3	c_3	$n-1$	$c_3.(n-1)$
4	c_4	$\sum_{j=1}^{n-1} t_j$	$c_4.\sum_{j=1}^{n-1} t_j$
5	c_5	$\sum_{j=1}^{n-1}(t_j-1)$	$c_5.\sum_{j=1}^{n-1}(t_j-1)$
6	c_6	$\sum_{j=1}^{n-1}(t_j-1)$	$c_6.\sum_{j=1}^{n-1}(t_j-1)$
8	c_8	$n-1$	$c_8.(n-1)$

Portanto, o custo total, que denotaremos por $T(n)$ é dado por:

$$T(n) = c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.\sum_{j=2}^{n} t_j + c_5.\sum_{j=2}^{n}(t_j-1) + c_6.\sum_{j=2}^{n}(t_j-1) + c_8.(n-1)$$

Agora note que, mesmo para entradas de mesmo tamanho, o tempo de execução pode mudar. De fato, um vetor que tenha mais elementos a serem reposicionados terá um custo maior para ser ordenado. Portanto, a análise do melhor caso se dá quando o vetor já estiver ordenado pois $t_j = 1$, para todo $2 \leq j \leq n$:

$$\begin{aligned} T_b(n) &= c_1.n + c_2.(n-1) + c_3.(n-1) + c_4.(n-1) + c_8.(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_8).n - (c_2 + c_3 + c_4 + c_8) \end{aligned}$$

ou seja, uma função linear de n. Por outro lado, a análise do pior caso se dá quando o vetor estiver ordenado decrescentemente pois $t_j = j$ (por que?), e portanto

$$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\tfrac{(n-1).n}{2}) + (c_5 + c_6).(\tfrac{(n-2).(n-1)}{2})$$

ou seja, uma função quadrática de n.

A forma de análise feita para InsertionSort acima, assim como para SequentialSearch na seção anterior, apresenta alguns problemas porque as constantes utilizadas podem mudar dependendo do computador, da linguagem de programação ou mesmo do estilo de programação utilizados. Uma maneira de ignorar estas especificidades, e fazer uma análise que seja independente destes aspectos, consiste na utilização de uma notação adequada, a *notação assintótica*, que considera o comportamento de funções no limite, isto é, para valores suficientemente grandes do parâmetro n. A ideia é que possamos pegar uma função como $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8$ que expressa o custo no pior caso do algoritmo de busca sequencial, e dizer que ela cresce como n, sem a necessidade

de considerar as constantes. Faremos isto considerando o conjunto das funções que são limitadas superiormente por um múltiplo constante de n. Observe que podemos facilmente construir uma cota superior para a função $T_w(n)$ da seguinte forma $T_w(n) = c_1 + c_2.n + c_3.(n-1) + c_6 + c_8 \leq c_1.n + c_2.n + c_3.n - c_3 + c_6.n + c_8.n \leq (c_1 + c_2 + c_3 + c_6 + c_8).n \leq c.n$ para qualquer constante $c \geq c_1 + c_2 + c_3 + c_6 + c_8$ e $n \geq 1$. Neste caso, dizemos que a função $T_w(n)$ é $O(n)$, ou seja, que $T_w(n)$ é de ordem n. Formalmente, temos a seguinte definição para o conjunto $O(g(n))$ que contém todas as funções que são da ordem de $g(n)$:

Definição 116. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $O(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $f(n) \leq c \cdot g(n), \forall n \geq n_0$. Alternativamente, $O(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0.\}$*

Observe que uma função $f(n)$ pode estar em $O(g(n))$ mesmo que $f(n) > g(n), \forall n$. O ponto importante é que $f(n)$ tem que ser limitada por um múltiplo constante de $g(n)$. A relação entre $f(n)$ e $g(n)$ para valores pequenos de n também é desconsiderada. Intuitivamente, os termos de menor ordem de uma função assintoticamente positiva podem ser ignorados na determinação da cota superior porque são insignificantes para valores grandes do parâmetro n. Assim, quando n é grande qualquer porção ou fração do termo de maior ordem é suficiente para dominar os termos de menor ordem.

Normalmente, escrevemos $T(n) \in O(n^2)$ para dizer que $T(n)$ é $O(n^2)$ já que $O(n^2)$ é um conjunto. No entanto, é comum encontrarmos o uso da igualdade $T(n) = O(n^2)$ ao invés de $T(n) \in O(n^2)$. A conveniência do uso da igualdade será vista posteriormente, mas o importante aqui é entender que esta igualdade é unidirecional, e portanto não pode ser confundida com a igualdade tradicional. Por exemplo, escrevemos $T(n) = O(n^2)$, mas $O(n^2) = T(n)$ não é correto. O número de funções anônimas em uma expressão é igual ao número de vezes que a notação assintótica aparece: por exemplo, na expressão $\sum_{i=1}^{n} O(i)$ contém apenas uma função anônima (a função que tem parâmetro i), e portanto esta expressão não é o mesmo que $O(1) + O(2) + \ldots + O(n)$ (que não possui uma interpretação clara). A notação assintótica também pode aparecer do lado esquerdo de uma equação: $2n^2 + O(n) = O(n^2)$. Neste caso, independentemente da forma como as funções anônimas são escolhidas do lado esquerdo da equação, existe uma forma de escolher funções anônimas do lado direito da equação de forma que a equação se verifique. No caso do exemplo acima, temos que para qualquer $f(n) = O(n)$, existe uma função $g(n) = O(n^2)$ tal que $2n^2 + f(n) = g(n), \forall n$.

Equações também podem ser encadeadas como em $2n^2 + 3n + 1 = 2n^2 + O(n) = O(n^2)$, e podem ser interpretadas separadamente de acordo com as regras anteriores. Assim, a primeira equação nos diz que existe alguma função $f(n) = O(n)$ para a qual a equação se verifica para todo n. A segunda equação nos diz que para toda função $g(n) = O(n)$, existe uma função $h(n) = O(n^2)$ tal que a equação se verifica para todo n. Este encadeamento é transitivo, ou seja, podemos concluir que $2n^2 + 3n + 1 = O(n^2)$.

O lema a seguir nos permite utilizar limites para utilizar a notação assintótica:

Lema 117. *Uma função $f(n) = O(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, incluindo o caso em que $c = 0$.*

Demonstração. Exercício.

□

Observe que a outra direção do lema anterior não vale: de fato, considere $f(n) = n$ e $g(n) = 2^{\lfloor \lg n \rfloor}$, onde $\lg n$ é o logaritmo de n na base 2. Temos que $f(n) = O(g(n))$ porque $f(n) = n = 2^{\lg n} \leq 2^{\lfloor \lg n \rfloor + 1} = 2.2^{\lfloor \lg n \rfloor} = 2.g(n), \forall n$. No entanto, o limite $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ não existe, já que o quociente $\frac{f(n)}{g(n)}$ oscila.

Teorema 118. *(a) Se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$*, onde* c *é uma constante real positiva, então* $f(n) = O(g(n))$ *e* $g(n) = O(f(n))$*;*

(b) Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ *então* $f(n) = O(g(n))$*, mas* $g(n) \neq O(f(n))$*;*

(c) Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *então* $f(n) \neq O(g(n))$*, mas* $g(n) = O(f(n))$*.*

Demonstração. Exercício.

□

No caso de InsertionSort, a análise do pior caso nos dá a função
$T_w(n) = c_1.n + (c_2 + c_3 + c_8).(n-1) + c_4.(\frac{(n-1).n}{2}) + (c_5 + c_6).(\frac{(n-2).(n-1)}{2})$ que é $O(n^2)$.

Como exercício, mostre em detalhes que a complexidade do pior caso de InsertionSort é $O(n^2)$.

Assim, considerando as expressões (ou polinômios) construídas(os) até agora, observamos que a classe de complexidade é obtida considerando-se o monômio de maior grau sem levar em conta o coeficiente. Portanto, a construção do polinômio a partir do custo de cada linha do algoritmo não é uma estratégia eficiente porque no final consideraremos apenas a parcela mais significativa, ou seja, o monômio de maior grau. Vamos então buscar diretamente a parte do algoritmo que nos dá este monômio de maior grau. Observando a Tabela 2 concluímos que o termo quadrático vem da linha 4, mais precisamente da comparação $A[i] > key$ que é executada em cada iteração do laço **for**. Então podemos fazer uma análise bem mais direta do que a feita anteriormente para chegarmos à mesma conclusão. Como durante a i-ésima iteração do laço **for**, a linha 4 é executada i vezes, temos:

$$T_w(n) = \sum_{i=1}^{n-1} i = \frac{(n-1).n}{2} = O(n^2).$$

A análise do melhor caso também pode ser feita da mesma forma considerando que a cada iteração do laço **for**, a linha 4 é executada uma única vez:

$$T_b(n) = \sum_{i=1}^{n-1} 1 = n - 1 = O(n).$$

Da mesma forma, na busca sequencial o custo linear do pior caso pode ser obtido calculando diretamente o número de comparações feitas na linha 2. A notação O nos dá uma cota superior para o custo de execução de algoritmos, mas ela também pode ser utilizada para estabelecer uma cota para a complexidade de espaço utilizado durante a execução de um algoritmo. Tanto a busca sequencial quanto o algoritmo de ordenação por inserção não necessitam de espaço adicional de armazenamento, e portanto, em ambos os casos a complexidade é constante, ou seja, é igual a $O(1)$. Dizemos que algoritmos de ordenação que não demandam espaço adicional fazem a ordenação *in place*. Posteriormente estudaremos algoritmos que necessitam de espaço adicional. Na tabela abaixo, resumimos as análises feitas até agora:

Algoritmo	tempo (melhor caso)	tempo (pior caso)	espaço
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$

Uma ferramenta bastante útil na análise assintótica é conhecida como *regra do máximo*:

$$O(f(n) + g(n)) = O(max(f(n), g(n))) \tag{4.1}$$

Depois de alguns exercícios, e de apresentarmos mais alguns detalhes sobre a notação assintótica, estudaremos um pouco da chamada análise do caso médio. A análise do melhor caso nos dá uma ideia de situações específicas em que o algoritmo tem a melhor performance possível, mas a análise do melhor caso não costuma ser muito informativa e normalmente não é relevante. A análise do pior caso, por outro lado, tem bastante relevância e será explorada exaustivamente nas próximas seções. Ela é importante porque nos fornece o pior cenário possível para o algoritmo. Com isto sabemos que o algoritmo não pode ter um comportamento menos eficiente do que o apresentado pela análise do pior caso. No entanto, esta análise pode ser excessivamente pessimista considerando uma situação mais realista. Por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A análise do caso médio pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis, o que não corresponde à média entre as análises do melhor e pior casos.

Por fim, é importante ter em mente que a notação assintótica nos permite analisar a taxa de crescimento, ou ordem de crescimento do tempo de execução de um algoritmo, e portanto as simplificações feitas na obtenção da cota superior não devem ser esquecidas em situações práticas. Por exemplo, considere dois algoritmos A e B com complexidades, respectivamente, iguais a $O(n^2)$ e $O(n^3)$. Qual dos dois algoritmos é mais eficiente? Para valores grandes de n certamente o algoritmo A é mais eficiente, mas devemos levar em consideração que no cálculo destas classes de complexidade diversas constantes foram ignoradas. Se soubéssemos, por exemplo, que o algoritmo A realiza $100.n^2$ operações, enquanto que o algoritmo B realiza $5.n^3$ operações para resolver o mesmo problema, então agora sabemos que para $n < 20$ o algoritmo B é mais eficiente.

Exercício 119. *Complete a tabela abaixo considerando os pseudocódigos apresentados nos exercícios 7 e 9.*

Algoritmo	*tempo (melhor caso)*	*tempo (pior caso)*	*espaço*
Sequential search	$O(1)$	$O(n)$	$O(1)$
Insertion sort	$O(n)$	$O(n^2)$	$O(1)$
Bubble sort			
Selection sort			

Exercício 120. *Mostre que* $n = O(n^2)$.

Exercício 121. *Mostre que* $100n + 5 = O(n^2)$.

Exercício 122. *Mostre que* $\frac{n(n-1)}{2} = O(n^2)$.

Exercício 123. *Mostre que* $n^3 \neq O(n^2)$.

Exercício 124. *Sejam* $f(n), g(n)$ *e* $h(n)$ *funções dos inteiros não-negativos nos reais positivos. Mostre que se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$.

Assim, como $O(g(n))$ estabelece uma cota superior para funções, o conjunto $\Omega(g(n))$ estabelece uma cota inferior para as funções:

Definição 125. *Seja $g(n)$ uma função dos inteiros não-negativos nos reais positivos. Então $\Omega(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem uma constante real $c > 0$ e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c \cdot g(n) \leq f(n), \forall n \geq n_0$. Alternativamente, $\Omega(g(n)) = \{f(n) :$ existem constantes positivas c e n_0 tais que $0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0.\}$*

Quando dizemos que o tempo de execução de um algoritmo é $\Omega(g(n))$, queremos dizer que independentemente da entrada de tamanho n, o tempo de execução desta entrada é pelo menos uma constante multiplicada por $g(n)$ para n suficientemente grande. Ou seja, estamos fornecendo uma cota inferior no melhor caso. Por exemplo, no melhor caso, o algoritmo InsertionSort é $\Omega(n)$, e portanto, o tempo de execução do algoritmo InsertionSort está entre $\Omega(n)$ e $O(n^2)$. A definição alternativa para o conjunto $\Omega(g(n))$ em termos de limites é dada pelo lema a seguir:

Lema 126. *Uma função $f(n) = \Omega(g)$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$, incluindo o caso em que o limite é igual a ∞.*

Demonstração. Exercício.

□

A forma mais precisa de expressar o comportamento assintótico de um algoritmo é fornecendo cotas superiores e inferiores ao mesmo tempo. No parágrafo anterior, apresentamos uma cota superior e uma cota inferior para o algoritmo InsertionSort. No entanto, estas cotas são de classes diferentes, o conjunto $\Theta(g(n))$, definido a seguir, é utilizado quando ambas as cotas são da mesma classe.

Definição 127. *Seja g uma função dos inteiros não-negativos nos reais positivos. Então $\Theta(g(n))$ é o conjunto das funções (também dos inteiros não-negativos nos reais positivos) tal que existem constantes reais positivas c_1 e c_2, e uma constante inteira $n_0 > 0$ satisfazendo a desigualdade $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0$. Alternativamente, $\Theta(g(n)) = \{f(n) :$ existem constantes positivas c_1, c_2 e n_0 tais que $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \forall n \geq n_0.\}$*

Como qualquer constante pode ser vista como um polinômio de grau 0, podemos representar funções constantes como $\Theta(n^0)$, ou simplesmente, $\Theta(1)$. O lema a seguir apresenta uma caracterização do conjunto $\Theta(g(n))$ em termos de limite:

Lema 128. *Uma função $f(n) = \Theta(g(n))$ se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$, para alguma constante $0 < c < \infty$.*

Demonstração. Exercício.

□

Teorema 129. *(a) Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty$, onde c é uma constante real positiva, então $f(n) = \Theta(g(n))$;*

(b) Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ então $f(n) = O(g(n))$, mas $f(n) \neq \Theta(g(n))$;

(c) Se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ *então* $f(n) = \Omega(g(n))$*, mas* $f(n) \neq O(g(n))$*.*

Demonstração. Exercício.

□

Exercício 130. *Prove que* $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$ *para qualquer inteiro* $k \geq 0$ *fixado.*

Teorema 131. *Dadas funções* $f(n)$ *e* $g(n)$*, temos que* $f(n) = \Theta(g(n))$ *se, e somente se,* $f(n) = O(g(n))$ *e* $f(n) = \Omega(g(n))$*.*

Demonstração. Exercício.

□

Lema 132. *(a)* $f(n) = O(g(n))$ *se, e somente se* $g(n) = \Omega(f(n))$*;*

(b) Se $f(n) = \Theta(g(n))$ *então* $g(n) = \Theta(f(n))$*;*

(c) Θ *define uma relação de equivalência sobre as funções. Cada conjunto* $\Theta(f(n))$ *é uma classe de equivalência que chamamos de classe de complexidade;*

(d) $\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$*;*

(e) $\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$*;*

Definição 133. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Definimos,* $o(g(n)) = \{f(n) :$ *para qualquer constante positiva* c*, existe uma constante positiva* n_0 *tal que* $0 \leq f(n) < c \cdot g(n), \forall n \geq n_0.\}$

Lema 134. *Uma função* $f(n) = o(g)$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$*.*

Definição 135. *Seja* $g(n)$ *uma função dos inteiros não-negativos nos reais positivos. Definimos,* $\omega(g(n)) = \{f(n) :$ *para qualquer constante positiva* c*, existe uma constante positiva* n_0 *tal que* $0 \leq c \cdot g(n) < f(n), \forall n \geq n_0.\}$

Lema 136. *Uma função* $f(n) = \omega(g(n))$ *se* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$*, se este limite existir.*

Lema 137. *Se* $f(n) = O(g(n))$ *e* $g(n) = O(h(n))$ *então* $f(n) = O(h(n))$*, ou seja, a notação* O *é transitiva. Também são transitivos* Ω, Θ, o *e* ω*.*

Teorema 138. $\lg n = o(n^\alpha), \forall \alpha > 0$*. Ou seja, a função logaritmo cresce mais lentamente do que qualquer potência de* n *(incluindo potências fracionárias)*

Teorema 139. $n^k = o(2^n), \forall k > 0$*. Ou seja, potências de* n *crescem mais lentamente que a função exponencial* 2^n*. Mais ainda, potências de* n *crescem mais lentamente do que qualquer função exponencial* $c^n, c > 1$*.*

Exercício 140. *Mostre que* $\frac{n^2}{2} - 3n = \Theta(n^2)$.

Exercício 141. *Mostre que* $6n^3 \neq \Theta(n^2)$.

Exercício 142. *Sejam* $f(n)$, $g(n)$ *e* $h(n)$ *funções não-negativas tais que* $f(n) = O(h(n))$ *e* $g(n) = O(h(n))$. *Prove que* $f(n) + g(n) = O(h(n))$.

4.3 Análise do caso médio

Nas sessões anteriores fizemos a análise do melhor caso e do pior caso para a busca sequencial, e para o algoritmo de ordenação por inserção. Nesta seção definiremos cada uma destas noções, assim como a análise do caso médio. A complexidade do pior caso consiste em considerar, dentre todas as entradas possíveis de um dado tamanho, aquela ou aquelas entradas para as quais o algoritmo em consideração executa o maior número de passos possível, ou seja, o maior número de operações básicas:

Definição 143. *Sejam* D_n *o conjunto das entradas de tamanho* n *para o algoritmo em questão, e* $I \in D_n$. *Seja* $t(I)$ *o número de operações básicas executadas pelo algoritmo na entrada* I. *Definimos a função* W, *que denota a complexidade do pior caso, por:*

$$W(n) = \max\{t(I) \mid I \in D_n\}$$

Analogamente, a complexidade do melhor caso é dada por:

Definição 144. *Sejam* D_n *o conjunto das entradas de tamanho* n *para o algoritmo em questão, e* $I \in D_n$. *Seja* $t(I)$ *o número de operações básicas executadas pelo algoritmo na entrada* I. *Definimos a função* B, *que denota a complexidade do melhor caso, por:*

$$B(n) = \min\{t(I) \mid I \in D_n\}$$

Como comentamos anteriormente, a análise do melhor caso não costuma ser muito informativa porque pode ocorrer de forma esporádica, e normalmente não é relevante. Em outras palavras, a análise do melhor caso faz pouco sentido porque sempre é possível construir um algoritmo que é rápido para uma entrada particular. Por outro lado, a análise do pior caso nos fornece uma cota superior para o tempo de execução do algoritmo que, por si só, já é bastante útil. No entanto, esta análise pode ser excessivamente pessimista: por exemplo, pode ser que o pior cenário só ocorra para uma ou duas entradas específicas dentre uma infinidade de possibilidades igualmente possíveis. A tentativa de fugir destes extremos nos leva à análise do caso médio, que pode nos fornecer uma ideia da eficiência do algoritmo considerando uma média dentre todos os tempos de execução possíveis. É importante observar que a análise do caso médio não corresponde à média entre as análises do melhor e pior casos. Formalmente, temos a seguinte definição:

Definição 145. *Sejam* D_n *o conjunto das entradas de tamanho* n *para o algoritmo em questão, e* $I \in D_n$. *Seja* $t(I)$ *o número de operações básicas executadas pelo algoritmo na entrada* I, *e* $p(I)$ *a probabilidade de ocorrência da entrada* I. *Definimos a complexidade do caso médio por:*

$$A(n) = \sum_{I \in D_n} p(I).t(I)$$

Portanto, a complexidade do caso médio consiste em uma espécie de média ponderada onde os pesos são probabilidades. Antes de fazermos um exemplo, vamos fazer algumas considerações sobre a complexidade do caso médio. Ainda que seja importante pelas razões apontadas no parágrafo anterior, a complexidade do caso médio não é tão comum quanto a complexidade do pior caso. A principal razão é que ela é em geral difícil de ser determinada. Sempre que possível incluiremos a complexidade do caso médio em nossas análises, mas na maioria dos casos ficaremos restritos à análise do pior caso. Como primeiro exemplo de análise do caso médio, vamos fazer esta análise para o algoritmo de busca sequencial. Para simplificar este primeiro exemplo, suponha que a chave procurada ocorra no vetor, e também assumiremos que a chave pode ocorrer com mesma probabilidade em qualquer uma das n posições do vetor A. Antes de prosseguirmos com este exemplo, vamos revisar algumas noções importantes de probabilidade.

Definição 146. *Um experimento é* determinístico *quando repetido em condições semelhantes conduz a resultados essencialmente idênticos. Já os experimentos que quando repetidos sob as mesmas condições produzem resultados geralmente diferentes são ditos* aleatórios*. Um* espaço amostral *é um conjunto que contém os resultados possíveis de um* experimento aleatório*. Os elementos de um espaço amostral são chamados de* eventos elementares*, enquanto que subconjuntos do espaço amostral são chamados de* eventos*.*

Por exemplo, considere o seguinte experimento aleatório: jogar um dado e observar o número mostrado na face de cima. O espaço amostral deste experimento é o conjunto $\Omega = \{1, 2, 3, 4, 5, 6\}$. Qualquer um dos elementos deste conjunto é um evento elementar, enquanto que o subconjunto $A = \{2, 4, 6\}$ corresponde ao evento que acontece se o número mostrado na face de cima é par. Os experimentos aleatórios que consideraremos aqui possuem as seguintes características:

1. Há um número finito de eventos elementares, e a união de todos os eventos elementares é o espaço amostral;

2. Os eventos elementares são igualmente prováveis;

3. Todo evento é a união de eventos elementares.

Definição 147. *Definimos a* probabilidade *de um evento A, notação $p(A)$, como sendo o quociente*

$$p(A) = \frac{\#(A)}{\#(\Omega)}$$

Como consequência desta definição temos as seguintes propriedades:

1. Para todo evento A, $0 \leq p(A) \leq 1$;

2. $p(\Omega) = 1$;

3. $p(\emptyset) = 0$;

4. Se $A \cap B = \emptyset$ então $p(A \cup B) = p(A) + p(B)$.

Voltando para o exemplo da busca sequencial, observe que ao assumirmos que a probabilidade da chave ocorrer em qualquer das n posições do vetor A, estamos dizendo que esta probabilidade é igual a $\frac{1}{n}$. O espaço amostral que estamos considerando é o conjunto D_n das entradas de tamanho n. Denotaremos por I_j o evento correspondente ao caso em que a chave ocorre na $j+1$-ésima $(0 \leq j \leq n-1)$ posição do vetor A, e portanto a união destes eventos é igual ao espaço amostral:

$$\bigcup_{j=0}^{n-1} I_j = D_n$$

O número de comparações realizadas para I_j é $t(I_j) = j + 1$, e portanto aplicando a definição da complexidade do caso médio, onde $p(I \mid succ)$ denota a probabilidade da entrada I para o caso de sucesso, temos:

$$A_s(n) = \sum_{I \in D_n} p(I \mid succ).t(I) = \sum_{j=0}^{n-1} p(I_j \mid succ).t(I_j) = \sum_{j=0}^{n-1} \frac{1}{n}.(j+1) = \sum_{j=1}^{n} \frac{1}{n}.j = \frac{1}{n}\frac{n.(n+1)}{2} = \frac{n+1}{2}.$$

Agora vejamos como a teoria da probabilidade pode nos ajudar a completar a análise do caso médio. A definição a seguir apresenta as noções de esperança e esperança condicional a partir de variáveis aleatórias que são variáveis reais que dependem do evento elementar que ocorreu. Em outras palavras, é uma função definida para eventos elementares, isto é, $X : \Omega \to \mathbb{R}$. No caso da busca sequencial, a função t que recebe um evento elementar de $I \in D_n$ e retorna o número de comparações feitas é uma variável aleatória. Ou seja, o nosso espaço amostral D_n contém as entradas de tamanho n, e cada possível entrada é um evento elementar.

Definição 148. *Seja $f(e)$ uma variável aleatória definida sobre um conjunto de eventos elementares $e \in U$. A* esperança *de f, denotada por $E(f)$, é definida por*

$$E(f) = \sum_{e \in U} f(e).p(e)$$

Normalmente, a esperança de f é chamada de valor médio de f. A esperança condicional *de f dado um evento S, denotada por $E(f \mid S)$, é definida por*

$$E(f \mid S) = \sum_{e \in U} f(e).p(e \mid S) = \sum_{e \in S} f(e).p(e \mid S)$$

onde a última igualdade se justifica pelo fato de que a probabilidade condicional de qualquer evento que não esteja em S é igual a 0.

Portanto a esperança de t (número de comparações) é igual a $A_s(n)$ calculado acima. Quando a chave não ocorre no vetor, n comparações são feitas, e denotaremos esta informação por $A_f(n) = n$. A análise do caso médio da busca sequencial é feita juntando estas informações de acordo com a lei da esperança condicional enunciada a seguir:

Lema 149. *Sejam $f(e)$ e $g(e)$ variáveis aleatórias definidas sobre um conjunto de eventos elementares $e \in U$, e S um evento qualquer. Então:*

1. $E(f+g) = E(f) + E(g)$

2. $E(f) = p(S).E(f \mid S) + p(\neg S).E(f \mid \neg S)$

Voltando ao nosso exemplo, consideraremos dois eventos: *succ* que consiste das instâncias de D_n (eventos elementares) que possuem a chave procurada, e $\neg \succ$ que contém as instâncias de D_n que não possuem a chave procurada. Assumindo que $p(succ) = q$, temos:

$$A(n) = p(succ).A_s(n) + p(\neg succ).A_f(n) = q.(\frac{n+1}{2}) + (1-q).n = n.(1 - \frac{q}{2}) + \frac{q}{2}.$$

Assim, se a probabilidade da chave ocorrer no vetor for a mesma de não ocorrer, i.e. se $q = \frac{1}{2}$ então $A(n) = \frac{3n+1}{4} = O(n)$, ou seja, a busca sequencial tem complexidade linear no caso médio.

O exemplo acima nos mostra como o conjunto D_n, das entradas de tamanho n, deve ser interpretado. Ao invés de considerar todos os possíveis vetores de tamanho n, identificamos as propriedades das entradas que afetam o comportamento do algoritmo. No exemplo, isto corresponde a considerar quando a chave é um elemento do vetor, e em que posições ocorre. Um elemento $I \in D_n$ pode ser visto como um conjunto (ou uma classe de equivalência) de todos os vetores e chaves que ocorrem em uma posição específica do vetor, ou que não ocorre no vetor. Então $t(I)$ é o número de operações realizadas para qualquer das entradas em I.

Exercício 150. *Considere o problema de busca em um vetor ordenado. Como o algoritmo de busca sequencial poderia ser melhorado para realizar a busca neste caso? Faça a análise do pior caso e caso médio para o seu algoritmo modificado.*

4.4 Casamento de padrões

Dados um texto t (sequência de n caracteres) e uma palavra p (*string*) (sequência de $m \leq n$ caracteres) que chamaremos de padrão, queremos encontrar a posição i da primeira ocorrência de p em t (*string matching*), ou retornar -1 caso não existam ocorrências de p em t. Mais precisamente, i corresponde à posição do caractere mais à esquerda da primeira ocorrência de p em t tal que $t_i = p_0$, ..., $t_{i+j} = p_j$, ..., $t_{i+m-1} = p_{m-1}$.

Caso outras possíveis ocorrências de p precisem ser encontradas, basta que o algoritmo continue até que todo o texto seja percorrido. Um algoritmo força-bruta para resolver este problema funciona da seguinte forma: alinhe o primeiro caractere da palavra p com o primeiro caractere do texto t e inicie a comparação entre os caracteres correspondentes. Quando os caracteres coincidem, dizemos que houve um casamento (*matching*), e prosseguimos para o caractere seguinte até, eventualmente parar depois do casamento do último caractere da palavra p. Quando durante este processo, um caractere do texto diverge do caractere correspondente da palavra, dizemos que ocorreu um *mismatching*, e neste caso, precisamos deslocar a palavra uma posição à direita da posição atual, para então reiniciarmos o processo. Observe que a última posição do texto t a partir da qual ainda pode o casamento com a palavra p é $n - m$ já que depois desta posição não temos mais caracteres suficientes para casar com a palavra p. A seguir, apresentamos o pseudocódigo que implementa esta ideia:

```
1  for  i = 0 to n − m do
2  |   j ← 0;
3  |   while j < m and P[j] = T[i + j] do
4  |   |   j ← j + 1;
5  |   end
6  |   if j = m then
7  |   |   return i;
8  |   end
9  end
10 return −1;
```

Algorithm 6: BFStringMatch($T[0..n-1], P[0..m-1]$)

Exercício 151. *Qual é a operação básica realizada por este algoritmo? Lembre-se que a operação básica é aquela que é mais relevante para a análise do custo do algoritmo.*

Exercício 152. *Qual é a complexidade deste algoritmo no melhor caso? Observe que o número de operações básicas a serem realizadas depende tanto do tamanho n do texto quanto do tamanho m da palavra, e portanto esta análise deve levar em conta os dois parâmetros.*

Exercício 153. *Qual é a complexidade deste algoritmo no pior caso? Observe que o número de operações básicas a serem realizadas depende tanto do tamanho n do texto quanto do tamanho m da palavra, e portanto esta análise deve levar em conta os dois parâmetros.*

Capítulo 5

Recursão

Recursão é um método para resolver problemas computacionais que depende das soluções de instâncias menores do mesmo problema[16] Nesta seção estudaremos alguns algoritmos recursivos, e iniciaremos pelas versões recursivas da busca sequencial e da ordenação por inserção. Como a utilização da recursão é muito natural no contexto funcional, isto é, da programação funcional, apresentaremos os algoritmos recursivos diretamente no assistente de provas Coq.

5.1 Busca sequencial recursiva

A busca sequencial pode ser definida recursivamente como $seq\_search\ l\ x := seq\_search\_aux\ l\ x\ 0$, onde:

Em Coq, temos:

```
Definition seq_search (l: list nat) (x: nat) := seq_search_aux l x 1.

Fixpoint seq_search_aux (l: list nat) (x pos: nat)  :=
 match l with
 | nil => 0
 | h::tl => if h =? x then pos else seq_search_aux tl x (S pos)
 end.
```

A função recursiva `seq_search` é definida em termos da função auxiliar `seq_search_aux` que recebe uma lista `l` de naturais, e os naturais `x` e `pos` como argumentos, e retorna 0 se `l` for a lista vazia. Ou seja, estamos assumindo que a primeira posição válida de uma lista é 1. Caso contrário, isto é, se `l` tem a forma `h::tl` então comparamos o primeiro elemento da lista `h` com o elemento procurado `x`, e se eles são iguais retornamos a posição atual `pos`. Caso contrário, continuamos recursivamente a busca de `x` na cauda `tl` da lista original depois de atualizar a variável `pos` em uma unidade.

1. **TODO** A correção do algoritmo de busca sequencial
2. **TODO** A complexidade do algoritmo de busca sequencial

5.2 TODO Busca binária

A busca binária é mais eficiente que a busca sequencial, mas este algoritmo assume que o vetor, onde a busca será realizada a busca, está ordenado. O pseudocódigo da busca binária em um vetor ordenado de inteiros com n elementos é dado a seguir:

```
1  if high < low then
2  |  return -1;
3  end
4  mid = ⌊(high + low)/2⌋;
5  if key > A[mid] then
6  |  return BinarySearch(A, mid + 1, high, key);
7  end
8  else
9  |  if key < A[mid] then
10 |  |  return BinarySearch(A, low, mid − 1, key);
11 |  end
12 |  else
13 |  |  return mid;
14 |  end
15 end
```

Algorithm 7: BinarySearch($A[1..n], low, high, key$)

Exercício 154. *Faça a análise da complexidade do melhor caso para este algoritmo.*

Exercício 155. *Faça a análise da complexidade do pior caso para este algoritmo.*

A correção deste algoritmo pode ser estabelecida em duas etapas. A primeira dela consiste em provar que se a chave *key* não ocorre no vetor $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor -1, e a segunda consiste em provar que o algoritmo retorna a posição correta do elemento procurado.

Exercício 156. *Prove o lema a seguir:*

> *Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key não ocorre em $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor -1.*

***Dica**: Indução (forte) sobre o tamanho n do vetor.*

Exercício 157. *Prove o lema a seguir:*

> *Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key ocorre em $A[1..n]$, então BinarySearch($A[1..n], 1, n, key$) retorna o valor mid tal que $A[mid] = key$.*

5.3 Ordenação por inserção recursivo

Nesta seção estudaremos o algoritmo de ordenação por inserção recursivo. A estrutura de dados utilizada é a de listas, e para simplificar trabalharemos com números naturais, mas as ideias são as mesmas para ordenarmos qualquer estrutura que possua uma ordem total. Vimos no capítulo anterior que as listas de naturais possuem dois construtores: *nil* para representar a lista vazia, e o operador :: que nos permite construir uma nova lista a partir de um número natural e de uma lista. Assim, a lista unitária contendo apenas o natural 5 é representada por $5 :: nil$, enquanto que a lista $1 :: (5 :: nil)$, ou simplesmente $1 :: 5 :: nil$, representa a lista que tem 1 como primeiro elemento, e a lista $5 :: nil$ como cauda.

A operação que dá nome ao algoritmo é a operação de inserção porque a cada passo queremos inserir um novo elemento em uma lista já ordenada. Suponha, por exemplo, que queiramos inserir o número 3 na lista $1 :: 5 :: nil$, isto é, o nosso objetivo final é obter a lista ordenada $1 :: 3 :: 5 :: nil$. Para isto,

precisamos inicialmente comparar o 3 com o primeiro elemento da lista, e o resultado desta comparação nos diz que o 3 deve ser inserido depois do 1, ou seja, em algum lugar da cauda da lista. Em seguida, comparamos o 3 com o primeiro elemento da cauda, ou seja, com 5, e como $3 < 5$, sabemos que ele deve ser inserido antes do 5. Esta ideia está implementada na função *insere* definida a seguir:

Definição 158. *Sejam x um número natural, e l uma lista de números naturais. A função $(insere\ x\ l)$ que insere o natural x na lista l é definida recursivamente como a seguir:*

$$insere\ x\ l = \begin{cases} x :: nil, & se\ l = nil \\ x :: l, & se\ l = h :: tl\ \ e\ x \leq h \\ h :: (insere\ x\ tl), & se\ l = h :: tl\ \ e\ x > h \end{cases}$$

O algoritmo de ordenação por inserção então consiste em recursivamente, dada uma lista não vazia $h :: tl$, inserir o primeiro elemento h na versão ordenada da cauda tl. Ou seja, o algoritmo de ordenação por inserção que será implementado pela função de *ord_insercao* vai receber como argumento uma lista l para ordenar. Se l for a lista vazia não há nada a fazer, e caso contrário, recursivamente ordenamos a cauda da lista para então inserir o novo elemento:

Definição 159. *Seja l uma lista de números naturais. A função ord_insercao é definida recursivamente como a seguir:*

$$ord\_insercao\ l = \begin{cases} nil, & se\ l = nil \\ insere\ h\ (ord\_insercao\ tl), & se\ l = h :: tl \end{cases}$$

Vejamos como este algoritmo funciona na prática. Suponha que queiramos ordenar a lista $3 :: 2 :: 1 :: nil$. Ao fornecermos esta lista como argumento para a função *ord_insercao*, temos:

$ord\_insercao\ (3 :: 2 :: 1 :: nil) =$ (def. *ord_insercao*)
$insere\ 3\ (ord\_insercao\ (2 :: 1 :: nil)) =$ (def. *ord_insercao*)
$insere\ 3\ (insere\ 2\ (ord\_insercao\ (1 :: nil))) =$ (def. *ord_insercao*)
$insere\ 3\ (insere\ 2\ (insere\ 1\ (ord\_insercao\ nil))) =$ (def. *ord_insercao*)
$insere\ 3\ (insere\ 2\ (insere\ 1\ nil)) =$ (def. *insere*)
$insere\ 3\ (insere\ 2\ (1 :: nil)) =$ (def. *insere*)
$insere\ 3\ (1 :: (insere\ 2\ nil)) =$ (def. *insere*)
$insere\ 3\ (1 :: 2 :: nil) =$ (def. *insere*)
$1 :: (insere\ 3\ (2 :: nil)) =$ (def. *insere*)
$1 :: 2 :: (insere\ 3\ nil) =$ (def. *insere*)
$1 :: 2 :: 3 :: nil$

Veja que o algoritmo ordenou corretamente a lista $3 :: 2 :: 1 :: nil$, mas será que ele ordena corretamente qualquer lista de números naturais? Para responder esta pergunta, vamos analisar se o algoritmo é correto ou não.

1. A correção do algoritmo de ordenação por inserção

   Nesta seção vamos provar que o algoritmo de ordenação por inserção apresentado na seção anterior é correto. Para isto precisaremos definir algumas noções que serão utilizadas também em outros algoritmos de ordenação. A primeira noção que precisamos definir formalmente é a de ordenação. Ou seja, o que significa dizer que uma lista está ordenada? A definição a seguir apresenta o predicado *sorted* que caracteriza a noção de lista ordenada:

   Definição 160. *Sejam x e y números naturais, e l uma lista de números naturais. O predicado sorted, que caracteriza o fato de uma lista estar ordenada, é definido por meio das seguintes regras de inferência:*

$$\frac{}{sorted\ nil}\ (sorted\_nil) \qquad \frac{}{sorted\ x :: nil}\ (sorted\_one)$$

$$\frac{x \leq y \qquad sorted\ y :: l}{sorted\ x :: y :: l}\ (sorted\_all)$$

A regra (*sorted_nil*) é um axioma que estabelece que a lista vazia está ordenada. A regra (*sorted_one*) também é um axioma que estabelece que listas unitárias estão ordenadas. A regra (*sorted_all*) possui duas condições para que uma lista da forma $x :: y :: l$ esteja ordenada: $x \leq y$ e a lista $y :: l$ tem que estar ordenada. Em outras palavras, a regra (*sorted_all*) diz que uma lista com pelo menos dois elementos está ordenada, se o primeiro elemento é menor ou igual ao segundo elemento, e a cauda da lista (ou seja, a lista do segundo elemento em diante) está ordenada. Note que as variáveis x, y e a lista l estão implicitamente quantificadas universalmente na Definição 160. Segundo esta definição, a lista $(1 :: 2 :: 3 :: nil)$ está ordenada. De fato, a prova deste fato é dada pela seguinte árvore de derivação:

$$\dfrac{1 \leq 2 \qquad \dfrac{2 \leq 3 \qquad \dfrac{}{sorted\ (3 :: nil)}\ (sorted\_one)}{sorted\ (2 :: 3 :: nil)}\ (sorted\_all)}{sorted\ (1 :: 2 :: 3 :: nil)}\ (sorted\_all)$$

Com esta definição em mãos, podemos provar uma propriedade da função *insere* que ficou implícita:

Lema 161. *Sejam x um número natural, e l uma lista de números naturais. Se l está ordenada então $(insere\ x\ l)$ também está ordenada.*

Demonstração. A prova é por indução na estrutura da lista l. Se l for a lista vazia então $(insere\ x\ l)$ é a lista unitária $x :: nil$ que está ordenada por definição (regra *sorted_one*). Se l é da forma $h :: tl$ então temos dois casos a considerar:

- $x \leq h$: Neste caso, $insere\ x\ (h :: tl)$ retorna a lista $x :: h :: tl$ que está ordenada já que $x \leq h$ e, por hipótese, a lista $h :: tl$ está ordenada:

$$\frac{x \leq h \qquad \frac{}{sorted\ h :: tl}\ (hip.)}{sorted\ x :: h :: tl}\ (sorted\_all)$$

- $x > h$: Neste caso, x será inserido na cauda tl, e por hipótese de indução temos que a lista $(insere\ x\ tl)$ está ordenada. Como a lista $h :: tl$ está ordenada, então h é menor ou igual a todo elemento de tl. Logo h é menor ou igual que todo elemento da lista $(insere\ x\ tl)$, e portanto a lista $h :: (insere\ x\ tl)$ está ordenada.

□

Agora vamos refazer esta prova no Coq. Note que uma das grandes vantagens da utilização de um assistente de provas é justamente a possibilidade de verificação de que uma prova feita manualmente está, de fato, correta. Isto é muito importante em provas mais complexas onde erros podem passar despercebidos. Iniciaremos definindo a função *insere* em Coq. Funções recursivas são definidas usando a palavra reservada `Fixpoint`:

```
Require Import List Arith.

Fixpoint insere x l :=
   match l with
    | nil => x::nil
    | h::tl => if x <=? h then x::l
               else h :: (insere x tl)
   end.
```

Na primeira linha importamos duas bibliotecas, a primeira chamada `List` nos permite usar a notação `x::l` para representar uma lista com cabeça `x` e cauda `l`. A segunda biblioteca, chamada `Arith`, nos permite usar a comparação booleana `<=?`. Observe que a definição acima é a mesma função da Definição 158: ambas retornam a lista unitária `x::nil` quando `l` é a lista vazia, e quando `l` tem a forma `h::tl`, a função retorna `x::l` quando `x` $\leq$ `h`, e `h::(insere x tl)`, caso contrário. O predicado *sorted* é definido em Coq utilizando a palavra reservada `Inductive`. Neste caso, cada regra da Definição 160 aparece como sendo um construtor da definição:

```
Inductive sorted: list nat -> Prop :=
| sorted_nil: sorted nil
| sorted_one: forall x, sorted (x::nil)
| sorted_all: forall x y l, x <= y -> sorted (y::l) -> sorted (x::y::l).
```

Agora estamos prontos para enunciar o Lema 161 em Coq, e iniciar a prova fazendo indução na estrutura da lista `l`. Observe que utilizamos o comando `induction l as [|h tl]` para utilizarmos os mesmos nomes que aparecem na prova do Lema 161, mas este detalhe não muda nada na estrutura da prova. Poderíamos ter utilizado apenas o comando `induction l`, e a única diferença é que o Coq utilizaria nomes de variáveis diferentes para se referir à cabeça e cauda da lista `l`.

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
```

Temos então dois casos a considerar:

```
  2 goals (ID 37)

   ============================
   forall x : nat, sorted nil -> sorted (insere x nil)

  goal 2 (ID 41) is:
  forall x : nat, sorted (h :: tl) -> sorted (insere x (h :: tl))
```

O primeiro caso se dá quando a lista `l` é a lista vazia. Então basta introduzirmos a variável `x` e a hipótese `sorted nil` no contexto, e aplicarmos a definição de `insere` via a tática `simpl` para concluirmos com a aplicação da regra `sorted_one`:

```
 Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
  induction l as [|h tl].
  - intros x H.
    simpl.
    apply sorted_one.
```

O segundo caso se dá quando a lista `l` tem a forma `(h::tl)`. Após introduzirmos a variável `x` e a hipótese `sorted (h::tl)`, precisamos provar que `sorted (insere x (h::tl))`:

```
  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  ============================
  sorted (insere x (h :: tl))
```

Observe a hipótese de indução `IHtl` que foi gerada pelo princípio de indução aplicado à estrutura da lista `l`: ela tem exatamente a mesma forma do lema (ou seja, expressa a mesma propriedade) aplicada à cauda `tl` da lista `l`. De acordo com a prova que fizemos para o Lema 161, neste ponto precisamos comparar `x` e `h` para decidir o que fazer de acordo com a definição da função `insere`. Podemos então aplicar a tática `simpl` (de simplificação) para que a definição de `insere` seja aplicada no objetivo atual:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
```

A janela de prova correspondente é:

```
  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  ============================
  sorted (if x <=? h then x :: h :: tl else h :: insere x tl)
```

Agora precisamos lidar com o condicional `if` para dividirmos a prova nos dois subcasos esperados. Para isto, utilizamos a tática `destruct` com `(x <=? h)` como argumento:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
```

Observe que a tática `destruct` foi utilizada com dois argumentos: `(x <=? h)` e `eqn:Hle`. O primeiro argumento divide a prova nos dois subcasos desejados, ou seja, $\texttt{x} \leq \texttt{h}$ e $\texttt{x} > \texttt{h}$. Já o argumento `eqn:Hle` mantém a informação dos casos que estão sendo analisados no contexto, isto é, na janela de prova:

```
2 goals (ID 61)

 h : nat
 tl : list nat
 IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
 x : nat
 H : sorted (h :: tl)
 Hle : (x <=? h) = true
 ============================
 sorted (x :: h :: tl)

goal 2 (ID 62) is:
sorted (h :: insere x tl)
```

Se a informação da hipótese `Hle` não fosse relevante para a prova, poderíamos ter utilizado a tática `destruct` apenas com o argumento `(x <=? h)`. Mas observe que para provarmos `sorted (x::h::tl)` precisamos, de acordo com a regra `sorted_all`, mostrar que `x` $\leq$ `h` e que `sorted (h::tl)`, e para isto precisaremos das hipóteses `Hle` e `H`, respectivamente.

No primeiro subcaso precisamos provar `sorted (x::h::tl)`, ou seja, que uma lista com pelo menos dois elementos está ordenada. Então, aplicamos a regra `sorted_all`. Isto vai dividir a prova em dois novos subcasos: no primeiro precisamos provar que `x <= h`, e no segundo, `sorted (h::tl)`. O segundo caso é imediato da hipótese `H`, então vamos focar no primeiro caso. Observe que `x <= h` é essencialmente o que diz a hipótese `Hle: (x <=? h) = true`, mas escrito de outra forma. Estas diferentes formas de escrever a mesma informação estão relacionadas com a teoria por trás do Coq e fogem do escopo deste livro. Então o que precisamos fazer é descobrir como passar de uma notação para outra. O Coq tem diversos comandos de busca, dentre os quais está o comando `Search`. Para resolver o nosso problema precisamos encontrar lemas que envolvam a relação `<=`. Podemos fazer esta busca com o comando `Search le` ou `Search "_ <= _"`. Em ambos os casos, o Coq vai exibir uma janela com o resultado da busca. A seguir aparecem três dos resultado listados que estão relacionados com os operadores `<=` e `<=?`:

```
leb_complete: forall m n : nat, (m <=? n) = true -> m <= n
leb_correct: forall m n : nat, m <= n -> (m <=? n) = true
Nat.leb_le: forall n m : nat, (n <=? m) = true <-> n <= m
```

Note que o lema `leb_correct` não serve para nossos propósitos porque não temos o operador `<=` nas hipóteses, e sim na conclusão. Mas tanto `leb_complete` quanto `Nat.leb_le` resolvem este caso, e ambos podem ser aplicados tanto na conclusão quanto na hipótese `Hle`. Por exemplo, para aplicar a tática `leb_complete` na conclusão utilizamos o comando `apply leb_complete`. Abaixo temos a prova com a aplicação do lema `leb_complete` na hipótese `Hle`:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
   + apply sorted_all.
     * apply leb_complete in Hle.
       assumption.
     * assumption.
```

Retornando para a prova do lema `insere_sorted`, precisamos analisar o caso em que `x` > `h`. Veja como esta condição aparece na hipótese `Hle` neste momento da prova:

```
h : nat
tl : list nat
IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
x : nat
H : sorted (h :: tl)
Hle : (x <=? h) = false
============================
sorted (h :: insere x tl)
```

Para utilizarmos aqui o mesmo argumento da prova do Lema 161, precisaremos encontrar uma forma de representar que um elemento é menor ou igual a todo elemento de uma lista, e relacionar este fato com o predicado `sorted`. Para isto, definimos o predicado `le_all`, de forma que `le_all x l` representa o fato de que `x` é menor ou igual a todo elemento de `l`, da seguinte forma:

```
Definition le_all x l := forall y, In y l -> x <= y.
```

Ou seja, `x` é menor ou igual a todo elemento de `l` se $x \leq y$, para todo `y` que seja elemento de `l`. O predicado `In` está definido em Coq, de forma que `In y l` significa que `y` é um elemento de `l`. A definição do predicado `In` pode ser vista com o comando `Print In`:

```
In =
fun A : Type =>
fix In (a : A) (l : list A) {struct l} : Prop :=
 match l with
 | nil => False
 | b :: m => b = a \/ In a m
 end
    : forall A : Type, A -> list A -> Prop
```

A palavra reservada `fix` denota que `In` é uma função recursiva (exatamente como fazemos com `Fixpoint`). Assim, se a lista dada como segundo argumento em `In a l` for a lista vazia a função retorna `False`, ou seja, `a` não é um elemento de `l`. Caso contrário, suponha que `l` tem a forma `b::m`, e neste caso verificamos se `b=a` ou então recursivamente continuamos a busca pelo elemento `a` na cauda `m`.

Agora podemos enunciar o argumento que precisamos para completar a prova do lema `insere_sorted`, ou seja, precisamos provar que se `l` é uma lista ordenada, e `a` é um natural menor ou igual a todo elemento de `l` então a lista `(a::l)` está ordenada:

```
Lemma le_all_sorted: forall l a, le_all a l -> sorted l -> sorted (a::l).
```

Este lema pode ser provado por análise de casos sobre a estrutura da lista `l`, isto é, basta inspecionarmos o que ocorre quando `l` é a lista vazia e quando é uma lista não vazia. Você pode estar se perguntando: mas não é isto que fazemos em uma prova por indução? Sim, a diferença é que na análise de casos não precisamos da hipótese de indução. Enquanto uma prova por indução é feita com a tática `induction`, a análise de casos é feita com a tática `case`:

```
Lemma le_all_sorted: forall l a, le_all a l -> sorted l -> sorted (a::l).
Proof.
 intro l.
 case l.
```

Neste ponto a prova é divida em dois subcasos, um quando `l` é a lista vazia, e outro quando `l` é uma lista não vazia:

```
2 goals (ID 42)

 l : list nat
 ============================
 forall a : nat, le_all a nil -> sorted nil -> sorted (a :: nil)

goal 2 (ID 43) is:
 forall (n : nat) (l0 : list nat) (a : nat),
 le_all a (n :: l0) -> sorted (n :: l0) -> sorted (a :: n :: l0)
```

O restante desta prova será deixado como exercício, e como dica fica a sugestão de dar uma olhada no lema `in_eq` que pode ser útil para completar a prova.

Exercício 162. *Complete a prova do lema `le_all_sorted`.*

Agora podemos retomar a prova do lema `insere_sorted` e aplicar o lema `le_all_sorted` que acabamos de provar. Isto vai dividir a prova em dois subcasos:

```
2 goals (ID 114)

 h : nat
 tl : list nat
 IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
 x : nat
 H : sorted (h :: tl)
 Hle : (x <=? h) = false
 ============================
 le_all h (insere x tl)

goal 2 (ID 115) is:
 sorted (insere x tl)
```

No primeiro subcaso precisamos provar que `h` é menor ou igual a todo elemento da lista `(insere x tl)`, e no segundo, que a lista `(insere x tl)` está ordenada. Como provar `le_all h (insere x tl)`? Isto é, como provar que `h` é menor ou igual a todo elemento da lista `(insere x tl)`? A hipótese `Hle` diz, usando a comparação booleana, que $\texttt{h} < \texttt{x}$. Adicionalmente, a hipótese `H` diz que a lista `(h::tl)` está ordenada, e portanto `h` tem que ser menor do que todo elemento em `tl`. Estes dois fatos nos permitem concluir informalmente o que queremos, mas como fazer isto em Coq? A ideia é novamente enunciar um lema auxiliar que será provado separadamente:

```
Lemma le_all_insere: forall l x y, y <= x -> le_all y l -> le_all y (insere x l).
Proof.
Admitted.
```

O lema `le_all_insere` expressa a propriedade que precisamos para continuar a prova do lema `insere_sorted`. Ao invés de provarmos este lema agora, vamos usá-lo para ver se realmente conseguimos avançar na prova de `insere_sorted`. Sua prova só será feita depois de verificarmos que o ele é realmente útil. Esta é uma estratégia importante no desenvolvimento de provas formais porque evita gastarmos energia na prova de um lema que eventualmente precise ser modificado,

ajustado ou mesmo eliminado em um momento posterior. O comando `Admitted` é utilizado nesta situação: permite que a utilização do lema ainda que ele não esteja provado. Ao aplicarmos o lema `le_all_insere` ao contexto atual, isto é, ao primeiro dos objetivos gerados na aplicação do lema `le_all_sorted`, obtemos uma nova bifurcação da prova:

```
  h : nat
  tl : list nat
  IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
  x : nat
  H : sorted (h :: tl)
  Hle : (x <=? h) = false
  ============================
  h <= x

goal 2 (ID 117) is:
le_all h tl
```

O primeiro subobjetivo, a saber `h <= x` pode ser provado a partir da hipótese `Hle`:

```
 Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
  induction l as [|h tl].
  - intros x H.
    simpl.
    apply sorted_one.
  - intros x H.
    simpl.
    destruct (x <=? h) eqn:Hle.
    + apply sorted_all.
      * apply leb_complete in Hle.
        assumption.
      * assumption.
    + apply le_all_sorted.
      * apply le_all_insere.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
```

No segundo ramo da prova precisamos provar `le_all h tl`, ou seja, que `h` é menor ou igual a todo elemento da lista `tl`. Precisamos então de uma propriedade semelhante ao lema `le_all_sorted`, mas na outra direção:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
Admitted.
```

Também deixaremos a prova deste lema para um momento posterior, mas é importante estar seguro de que todos os lemas deixados em aberto expressam propriedades corretas. Este é o caso do lema `sorted_le_all` porque se a lista `(a::l)` está ordenada então o primeiro elemento tem que ser menor ou igual a todos os elementos da cauda. Este segundo ramo é concluído de forma imediata com a ajuda deste lema:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
```

```
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
   simpl.
   destruct (x <=? h) eqn:Hle.
   + apply sorted_all.
     * apply leb_complete in Hle.
       assumption.
     * assumption.
   + apply le_all_sorted.
     * apply le_all_insere.
       ** apply leb_complete_conv in Hle.
          apply Nat.lt_le_incl in Hle.
          assumption.
       ** apply sorted_le_all.
          assumption.
```

O segundo caso gerado na aplicação do lema `le_all_sorted` consiste na prova de que a lista `(insere x tl)` está ordenada:

```
1 goal (ID 115)

 h : nat
 tl : list nat
 IHtl : forall x : nat, sorted tl -> sorted (insere x tl)
 x : nat
 H : sorted (h :: tl)
 Hle : (x <=? h) = false
 ============================
 sorted (insere x tl)
```

Note que podemos obter `sorted (insere x tl)` da hipótese de indução `IHtl` desde que a lista `tl` esteja ordenada, isto é, desde que tenhamos uma prova de `sorted tl`. Esta prova pode ser obtida da hipótese `H`, pois se a lista `(h::tl)` está ordenada então sua cauda `tl` também está ordenada. Apesar deste fato ser óbvio, precisamos provar mais este resultado auxiliar no Coq:

```
Lemma sorted_sublist: forall l a, sorted (a::l) -> sorted l.
```

A prova deste lema pode ser feita via análise de casos na estrutura da lista `l` e é deixada como exercício:

Exercício 163. *Prove o lema `sorted_sublist`.*

Agora podemos concluir a prova do lema `insere_sorted`:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
```

```
    simpl.
    apply sorted_one.
  - intros x H.
    simpl.
    destruct (x <=? h) eqn:Hle.
    + apply sorted_all.
      * apply leb_complete in Hle.
        assumption.
      * assumption.
    + apply le_all_sorted.
      * apply le_all_insere.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
        ** apply sorted_le_all.
           assumption.
      * apply IHtl.
        apply sorted_sublist in H.
        assumption.
Qed.
```

Agora que sabemos que os lemas `le_all_insere` e `sorted_le_all` são efetivamente úteis em nossa formalização, podemos prová-los.
Vamos iniciar com a prova do lema `sorted_le_all`, que é feita por indução na estrutura da lista `l`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l.
```

Quando a lista `l` é a lista vazia (base da indução), precisamos provar que o natural `a` é menor ou igual a todo elemento da lista vazia. Como a lista vazia não possui nenhum elemento, dizemos que este fato é verdadeiro por vacuidade, isto é, porque não existe nenhum elemento que o contradiz. Para ver como este tipo de situação ocorre em Coq, vamos abrir a definição de `le_all` com o comando `unfold le_all`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
```

O comando `unfold le_all` simplesmente substitui a expressão `le_all a l` pela expressão correspondente à definição de `le_all`:

```
 1 goal (ID 79)

  a : nat
  H : sorted (a :: nil)
  ============================
  forall y : nat, In y nil -> a <= y
```

Depois de fazermos as introduções possíveis, temos a hipótese `In y nil`. Como a lista vazia não possui elementos, a tática `inversion` nos permite concluir este ramo da prova.

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
   intros y Hnil.
   inversion Hnil.
```

No passo indutivo, a lista `l` tem a forma `h::tl`, e a janela de prova após fazermos as introduções possíveis é a seguinte:

```
  1 goal (ID 86)

   h : nat
   tl : list nat
   IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
   a' : nat
   H : sorted (a' :: h :: tl)
   ============================
   le_all a' (h :: tl)
```

Então precisamos provar que `a'` é menor ou igual a todo elemento da lista `(h::tl)`. Dividiremos esta tarefas em dois passos: primeiro mostraremos que `a'` é menor ou igual a `h`, e depois que `a'` é menor ou igual a todo elemento da lista `tl`. Para isto vamos enunciar mais um lema auxiliar cuja prova será deixada como exercício:

Exercício 164. *Prove o lema a seguir utilizando análise de casos na estrutura da lista* `l`*:*

```
Lemma le_le_all: forall l x y, y <= x -> le_all y l -> le_all y (x::l).
```

A aplicação do lema `le_le_all` divide a prova nos dois subcasos descritos acima. A prova de que `a'` $\leq$ `h` pode ser obtida da hipótese `H` porque a regra `sorted_all` diz que para uma lista com dois ou mais elementos estar ordenada, o primeiro elemento precisa ser menor ou igual ao segundo. Então utilizamos a tática `inversion` para que esta condição seja gerada a partir da hipótese `H`:

```
Lemma sorted_le_all: forall l a, sorted (a::l) -> le_all a l.
Proof.
 induction l as [|h tl].
 - intros a H.
   unfold le_all.
   intros y Hnil.
   inversion Hnil.
 - intros a' H.
   apply le_le_all.
   + inversion H; subst.
     assumption.
```

O segundo subcaso gerado consiste em provar `le_all a' tl`. Para isto podemos utilizar a hipótese de indução. Veja a janela de prova atual:

```
h : nat
tl : list nat
IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
a' : nat
H : sorted (a' :: h :: tl)
============================
le_all a' tl
```

Com o comando `apply IHtl` aplicamos a hipótese de indução ao objetivo atual, e o novo objetivo a ser provado passa a ser o antecedente da implicação que compõe a hipótese de indução considerando que a variável universal `a` de `IHtl` foi instanciada com `a'`:

```
h : nat
tl : list nat
IHtl : forall a : nat, sorted (a :: tl) -> le_all a tl
a' : nat
H : sorted (a' :: h :: tl)
============================
sorted (a' :: tl)
```

A prova de `sorted (a'::tl)` pode ser obtida a partir da hipótese `H`, se pudéssemos remover o segundo elemento da lista `(a'::h::tl)`, mas como fazer isto? Exatamente, através de um lema auxiliar já que a extração do segundo elemento de uma lista ordenada não decorre diretamente das definições que temos. Esta tarefa fica como exercício e pode ser feita por análise de casos:

Exercício 165. *Complete a prova do lema `sublist_sorted`:*

```
Lemma sublist_sorted: forall l a1 a2, sorted (a1 :: a2 :: l) -> sorted (a1 :: l).
Proof.
intro l; case l.
```

A última pendência em relação à prova do lema `insere_sorted` é o lema auxiliar `le_all_insere`. Esta prova será deixada como exercício já que sua prova pode ser feita com a ajuda dos lemas auxiliares já apresentados, ou seja, nenhum lema auxiliar adicional é necessário.

Exercício 166. *Prove o lema a seguir utilizando indução na estrutura da lista `l`:*

```
Lemma le_all_insere: forall l x y, y <= x -> le_all y l -> le_all y (insere x l).
```

Seguimos um longo caminho até completarmos uma versão formal (ou mecânica) da prova do Lema 161. Uma pergunta natural é: existe um caminho mais curto, ou em outras palavras, existe uma outra prova possível para este lema? A resposta é sim! A seguir apresentamos uma prova alternativa que não requer lemas auxiliares:

```
Lemma insere_sorted: forall l x, sorted l -> sorted (insere x l).
Proof.
 induction l as [|h tl].
 - intros x H.
   simpl.
   apply sorted_one.
 - intros x H.
```

```
    simpl.
    destruct (x <=? h) eqn:Hle.
    + apply sorted_all.
      * apply leb_complete in Hle.
        assumption.
      * assumption.
    + generalize dependent tl.
      intro tl; case tl.
      * intros IH H.
        simpl.
        apply sorted_all.
        ** apply leb_complete_conv in Hle.
           apply Nat.lt_le_incl in Hle.
           assumption.
        ** apply sorted_one.
      * intros n l IH H.
        simpl in *.
        destruct (x <=? n) eqn:H'.
        ** apply sorted_all.
           *** apply leb_complete_conv in Hle.
               apply Nat.lt_le_incl in Hle.
               assumption.
           *** apply sorted_all.
               **** apply leb_complete.
                    assumption.
               **** inversion H; subst.
                    assumption.
        ** inversion H; subst.
           apply sorted_all.
           *** assumption.
           *** specialize (IH x).
               apply IH in H4.
               rewrite H' in H4.
               assumption.
Qed.
```

Você compreendeu o que esta prova faz de diferente? Como exercício vamos fazer o inverso do que foi feito com o Lema 161.

Exercício 167. *Construa uma prova em linguagem natural que corresponda a estratégia utilizada na prova em Coq acima.*

Nosso próximo passo é provar que o algoritmo de ordenação por inserção efetivamente ordena qualquer lista de naturais dada como entrada:

Lema 168. *O algoritmo de ordenação por inserção da Definição 159 ao receber uma lista l de números naturais como argumento retorna uma lista ordenada. Em outras palavras, a lista $(ord\_insercao\ l)$ está ordenada, para qualquer lista l.*

Demonstração. A prova é por indução na estrutura da lista l. Se l é a lista vazia (base de indução) então, por definição temos que $ord\_insercao\ nil = nil$, e não há o que fazer porque a lista vazia está ordenada. No passo indutivo suponha que l tem a forma $h :: tl$. Temos $ord\_insercao\ (h :: tl) = insereh(ord\_insercao\ tl)$, e por hipótese de indução temos que a lista $(ord\_insercao\ tl)$.

Então, pelo Lema 161 concluímos que a lista $insereh(ord\_insercao\ tl)$ está ordenada, e portanto $ord\_insercao\ (h :: tl)$ está ordenada.

□

Exercício 169. *Refaça a prova acima utilizando a estrutura de árvore. Em outras palavras, prove o sequente* $\vdash sorted(ord\_insercao\ l)$.

Agora prove o Lemma 168 em Coq:

Exercício 170. `Lemma ord_insercao_ordena: forall l, sorted (ord_insercao l).`

A segunda parte da prova da correção de um algoritmo de ordenação consiste em mostrar que o algoritmo retorna uma lista que é uma permutação da lista de entrada. Assim, um algoritmo de ordenação será correto se, para qualquer lista l dada como entrada, a saída for uma permutação de l que esteja ordenada. Ou seja, a resposta do algoritmo tem que ser uma lista que contém exatamente os mesmos elementos da lista de entrada e que adicionalmente esteja ordenada.

Como então definir a noção de permutação? Temos pelo menos duas opções. A primeira é simplesmente contar o número de ocorrências de cada elemento e ver que qualquer elemento tem que ocorrer o mesmo número de vezes nas duas listas para que uma seja uma permutação da outra. De maneira mais precisa, podemos definir o número de ocorrências de x na lista l, notação $num\_oc\ x\ l$ da seguinte forma:

Definição 171. *Seja x um número natural, e l uma lista de números naturais. Definimos recursivamente o número de ocorrências de x em l por:*

$$num\_oc\ x\ l = \begin{cases} 0, & se\ l = nil \\ 1 + num\_oc\ x\ tl, & se\ l = x :: tl \\ num\_oc\ x\ tl, & caso\ contrário. \end{cases}$$

O predicado *perm*, que define quando duas lista, digamos l e l' são permutações uma da outra.

Definição 172. *Sejam l e l' listas de números naturais. Definimos o predicado perm em função de num_oc por* $perm\ l\ l' := \forall x, num\_oc\ x\ l = num\_oc\ x\ l'$.

De acordo com esta definição, a lista l' é uma permutação da lista l se o número de ocorrências de x em l é igual ao número de ocorrências de x em l'. Nosso objetivo agora é mostrar que o algoritmo de ordenação por inserção gera uma lista que é uma permutação da lista de entrada, ou seja, queremos provar o seguinte teorema:

Teorema 173. *Seja l uma lista de números naturais. O algoritmo de ordenação por inserção gera como saída uma lista que é permutação da lista de entrada, ou seja, o sequente* $\vdash perm\ l\ (ord\_insercao\ l)$ *é válido.*

Demonstração. A prova é por indução na estrutura da lista l. Quando l é a lista vazia (base da indução), temos que $num\_oc\ x\ (ord\_insercao\ nil) = num\_oc\ x\ nil$ para todo x, ou seja, nil é uma permutação de $(ord\_insercao\ nil)$. Suponha que l tenha a forma $h :: tl$ (passo indutivo). Precisamos provar que $(h :: tl)$ é uma permutação da lista $(ord\_insercao\ (h :: tl))$, que pela definição de $ord\_insercao$ é igual a $(insere\ h\ (ord\_insercao\ tl))$. Por hipótese de indução temos que tl é uma permutação da lista $(ord\_insercao\ tl)$. Considerando que a função $(insere\ h\ (ord\_insercao\ tl))$ apenas adiciona o elemento h à lista $(ord\_insercao\ tl)$, concluímos que a lista $(h :: tl)$ é uma permutação da lista $insere\ h\ (ord\_insercao\ tl)$, que por sua vez é igual a $(ord\_insercao\ (h :: tl))$, como queríamos demonstrar.

□

Como seria a representação desta prova em forma de árvore? O primeiro passo é aplicar o princípio de indução para listas (veja o Exercício 87):

$$\dfrac{(def.)\ \dfrac{(\text{Refl})\ \dfrac{}{perm\ nil\ nil}}{perm\ nil\ (ord\_insercao\ nil)} \qquad \dfrac{\dfrac{(*)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{perm\ (h :: tl)\ (ord\_insercao\ (h :: tl))}}{perm\ l\ (ord\_insercao\ l)}\ (ind.\ em\ l)$$

A folha do ramo esquerdo, não é nada mais do que a igualdade $\forall x, num\_oc\ x\ nil = \_oc\ x\ nil$, e portanto este ramo da prova está completo pelo axioma da reflexividade da igualdade apresentado abaixo, onde t é um termo qualquer:

$$\dfrac{}{t = t}\ (\text{Refl})$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\ (def.)}{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\ (\forall_e)}{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}}{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (def.)}{(*)}$$

A continuação da prova do ramo direito segue com a aplicação da definição de *perm* na leitura de baixo para cima. Em seguida aplicamos a definição de *num_oc*, e adicionamos a função sucessor em ambos os lados da igualdade já que h ocorre tanto do lado esquerdo quanto do lado direito. A regra utilizada acima sem nome (quarta linha de baixo para cima) corresponde à uma propriedade algébrica que não precisa ser detalhada aqui (injetividade da igualdade), mas é importante notar que a passagem da prova em linguagem natural (que normalmente chamamos de prova informal) para a prova em forma de árvore já exige uma disciplina maior porque cada regra aplicada na árvore exige uma justificativa mais detalhada. Por fim, aplicamos mais uma vez a definição de `perm` para chegarmos na hipótese de indução.

Agora vamos formalizar esta prova em Coq. Como sabemos, a disciplina exigida é ainda maior do que a que foi necessária para a construção da árvore de derivação. Iniciaremos com a definição da função *num_oc* em Coq:

```
Fixpoint num_oc n l  :=
 match l with
   | nil => 0
   | h :: tl =>
     if n =? h then S(num_oc n tl) else  num_oc n tl
 end.
```

Após se certificar que esta definição faz exatamente o mesmo que a Definição 171, podemos construir a versão em Coq da definição 172:

```
Definition perm l l' := forall n:nat, num_oc n l = num_oc n l'.
```

Agora vamos refazer a prova do Teorema 173 em Coq. A prova é por indução na estrutura da lista `l`, e a base de indução, como na prova informal (e na árvore de derivação), é imediata. Basta abrirmos a definição com a tática `unfold` e aplicarmos o axioma da reflexividade da igualdade (tática `reflexivity`).

```
 Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
  induction l as [|h tl].
  - simpl.
    unfold perm.
    reflexivity.
```

No passo indutivo, precisamos aplicar a definição de *perm* para que tenhamos o objetivo em função de *num_oc*, ou seja, precisamos aplicar a tática `unfold`. A janela de prova correspondente é mostrada a seguir:

```
  h : nat
  tl : list nat
  IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
  ============================
  forall n : nat,
  num_oc n (h :: tl) = num_oc n (insere h (ord_insercao tl))
```

Aqui é possível ver um problema na árvore de dedução acima. A aplicação da definição de *perm* (na primeira linha de baixo para cima) está **errada**! De fato, a definição de *perm* é feita sobre uma variável quantificada universalmente, enquanto que na árvore acima esta variável está instanciada como h, ou seja, é um caso particular da definição e portanto não serve como prova. Esta situação simples, mas serve para mostrar como uma formalização pode ajudar a corrigir erros de uma prova informal. A nossa estratégia será completar primeiro a prova em Coq, e a partir daí refazer a árvore de dedução. Após introduzirmos a variável universal `n`, precisamos comparar `n` com `h` para saber se o contador precisa ou não ser incrementado. Podemos, depois de `intro n`, usar a tática `simpl` para aplicar a definição de `num_oc` e gerar condicional que vai nos permitir dividir a prova em dois casos:

```
  h : nat
  tl : list nat
  IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
  n : nat
  ============================
  (if n =? h then S (num_oc n tl) else num_oc n tl) =
  num_oc n (insere h (ord_insercao tl))
```

Com o comando `destruct (n =?  h) eqn:H.` dividimos a prova em dois casos e guardamos a informação do caso em andamento na hipótese `H`. O primeiro caso é quando `n` é igual a `h`, e corresponde ao caso analisado em nossa árvore de dedução. No entanto, a árvore não analisou o caso em que `n` e `h` são distintos. Utilizaremos o lema `beq_nat_true` para transformar a comparação booleana em igualdade sintática, e assim substituir (tática `subst`) todas as ocorrências de `n` por `h`:

```
Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
   unfold perm in *.
   intro n.
   simpl.
   destruct (n =? h) eqn:H.
```

```
  + apply beq_nat_true in H.
    subst.
```

E a janela de prova correspondente é a seguinte:

```
h : nat
tl : list nat
IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
============================
S (num_oc h tl) = num_oc h (insere h (ord_insercao tl))
```

Agora precisamos que o Coq transformar `num_oc h (insere h (ord_insercao tl))` em `S (num_oc h (ord_insercao tl))`. No entanto, esta transformação não é trivial do ponto de vista formal porque não sabemos de antemão a posição da lista `(ord_insercao tl)` em que `h` será inserido. Ou seja, esta transformação não pode ser obtida de forma imediata das definições de `num_oc` e `insere`. Portanto, precisamos de um lema auxiliar que faça isto:

```
Lemma num_oc_insere: forall l x, num_oc x (insere x l) = S (num_oc x l).
```

A prova deste lema será deixada como exercício, e pode ser feita por indução na estrutura da lista `l`.

Exercício 174. *Prove o lema `num_oc_insere`.*

Como o lema `num_oc_insere` é uma igualdade então utilizamos a tática `rewrite`, e depois disto fechamos este ramo da prova com a hipótese de indução:

```
Theorem ord_insercao_perm: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
   unfold perm in *.
   intro n.
   simpl.
   destruct (n =? h) eqn:H.
   + apply beq_nat_true in H.
     subst.
     rewrite num_oc_insere.
     rewrite IHtl.
     reflexivity.
```

Por fim, podemos analisar o caso que faltou na nossa árvore de derivação, a saber, o caso em que `n` é diferente de `h`:

```
h : nat
tl : list nat
IHtl : forall n : nat, num_oc n tl = num_oc n (ord_insercao tl)
n : nat
H : (n =? h) = false
```

```
  ============================
  num_oc n tl = num_oc n (insere h (ord_insercao tl))
```

Este ramo pode ser provado facilmente, desde que consigamos transformar `num_oc n (insere h (ord_insercao tl))` em `num_oc n (ord_insercao tl)`, o que é verdade já que `n` é diferente de `h`. Novamente precisaremos de um resultado auxiliar cuja prova será deixada como exercício:

```
Lemma num_oc_insere_diff: forall l x y, (x =? y) = false -> num_oc x (insere y l) = num_oc x l.
```

Exercício 175. *Prove o lema* ***num_oc_insere_diff****.*

Com este lema e a hipótese de indução conseguimos completar a prova:

```
Theorem perm_ord_insercao: forall l, perm l (ord_insercao l).
Proof.
 induction l as [|h tl].
 - simpl.
   unfold perm.
   reflexivity.
 - simpl.
   unfold perm in *.
   intro n.
   simpl.
   destruct (n =? h) eqn:H.
   + apply beq_nat_true in H.
     subst.
     rewrite num_oc_insere.
     rewrite IHtl.
     reflexivity.
   + rewrite num_oc_insere_diff.
     * apply IHtl.
     * assumption.
Qed.
```

Agora podemos corrigir o ramo da árvore de dedução que corresponde ao passo indutivo. Note que a aplicação da definição de *perm* (primeira regra de baixo para cima) gera uma fórmula quantificada universalmente.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\,(h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(def.)}{num\_oc\ h\ tl = num\_oc\ h\ (ord\_insercao\ tl)}\,(\forall_e)}{S(num\_oc\ h\ tl) = S(num\_oc\ h\ (ord\_insercao\ tl))}}{num\_oc\ h\ (h :: tl) = num\_oc\ h\ (insere\ h\ (ord\_insercao\ tl))}\,(**)}{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\,(h = x \vee h \neq x)}{perm\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\,(def.)}{(*)}\,(h = x)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{perm\ tl\ (ord\_insercao\ tl)}\,(h.i.)}{\forall x, num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(def.)}{num\_oc\ x\ tl = num\_oc\ x\ (ord\_insercao\ tl)}\,(\forall_e)}{num\_oc\ x\ (h :: tl) = num\_oc\ x\ (insere\ h\ (ord\_insercao\ tl))}\,(h \neq x)}{(**)}$$

Os lemas *ord_insercao_ordena* e *ord_insercao$_p$erm* juntos caracterizam a correção do algoritmo de ordenação por inserção. Em Coq, temos:

```
Theorem ord_insercao_correcao: forall l, sorted (ord_insercao l) /\ perm l (ord_insercao l).
Proof.
 intro l. split.
 - apply ord_insercao_ordena.
 - apply ord_insercao_perm.
Qed.
```

Para finalizar esta seção, mostre que *perm* é uma relação de equivalência sobre a estrutura de listas, isto é, mostre que *perm* é reflexiva, simétrica e transitiva.

Exercício 176. *Mostre que o predicado perm da Definição 172 é uma relação de equivalência sobre a estrutura de listas, isto é, mostre:*

(a) perm l l, para qualquer lista l (reflexividade)

(b) Se perm l l′ então perm l′ l, quaisquer que sejam as listas l e l′ (simetria)

(c) Se perm l l′ e perm l′ l″ então perm l l″, quaisquer que sejam as listas l, l′ e l″ (transitividade)

(d) Refaça suas provas no Coq:

```
Lemma perm_refl: forall l, perm l l.

Lemma perm_sym: forall l l', perm l l' -> perm l' l.

Lemma perm_trans: forall l l' l'', perm l l' -> perm l' l'' -> perm l l''.
```

Existem formas distintas de definirmos o mesmo conceito, ou seja, existem formas distintas de escrever a mesma coisa. A consequência é um conjunto de provas diferentes que podem ser mais simples ou mais complexas. No contexto de provas informais, a mudança de uma definição pode não ter muito impacto, mas o contexto formal é muito mais sensível a este tipo de mudança. Além disto, a mudança ou mesmo um ajuste em uma definição ou teorema durante uma formalização normalmente implica em ter que refazer todas as provas que dependem daquela mudança. Por isto, um bom planejamento é fundamental antes de iniciar uma formalização. Para exemplificar como definições distintas podem impactar em uma formalização, apresentaremos uma definição indutiva da noção de permutação de listas.

Definição 177. *Sejam x e y números naturais, e l, l′ e l″ listas de números naturais. O predicado binário* `permutation` *é definido pelas regras de inferência seguintes:*

$$\frac{}{permutation\ nil\ nil}\ (permutation\_nil)$$

$$\frac{permutation\ l\ l'}{permutation\ (x :: l)\ (x :: l')}\ (permutation\_skip)$$

$$\frac{}{permutation\ (y :: x :: l)\ (x :: y :: l)}\ (permutation\_swap)$$

$$\frac{permutation\ l\ l' \qquad permutation\ l'\ l''}{permutation\ l\ l''}\ (permutation\_trans)$$

Você pode estar se perguntando se as definições *perm* e *permutation* são equivalentes. A resposta é sim, e a conclusão desta seção será justamente a prova desta equivalência. Isto significa que a utilização de uma ou outra não fará diferença do ponto de vista prático, mas pode fazer em relação à simplicidade ou complexidade das provas envolvidas. Vamos mostrar que o algoritmo de ordenação por inserção gera como saída uma lista que é uma permutação da lista de entrada segundo esta nova definição. Como a definição de *permutation* é feita via regras de inferência, é mais natural que a prova seja feita na forma de árvore:

Lema 178. *Seja l uma lista de números naturais. Então o sequente $\vdash permutation\ l\ (ord\_insercao\ l)$ é válido.*

Demonstração. A prova é por indução na estrutura da lista l. A base de indução é simples:

$$\frac{\dfrac{}{permutation\ nil\ nil}\ (permutation\_nil)}{permutation\ nil\ (ord\_insercao\ nil)}\ (def.)$$

Agora suponha que l tenha a forma $h :: tl$. Queremos construir uma prova para o sequente $\vdash permutation\ (h :: tl)\ (ord\_insercao\ (h :: tl))$. O primeiro passo é aplicar a definição de $ord\_insercao$:

$$\frac{\dfrac{(*)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}}{permutation\ (h :: tl)\ (ord\_insercao\ (h :: tl))}\ (def.)$$

□

E neste ponto precisamos de um resultado auxiliar porque nenhuma das regras pode ser aplicada já que não sabemos quem é(são) o(s) primeiro(s) elemento(s) da lista $(insere\ h\ (ord\_insercao\ tl))$. Na verdade, a utilização da regra *permutation_trans* é possível, mas precisaríamos de uma lista intermediária que nos permitisse avançar na prova. Vamos seguir o caminho do resultado auxiliar e provar a propriedade que corresponde ao objetivo atual:

Lema 179. *Sejam x um número natural, l e l' listas de números naturais. Se $(permutation\ l\ l')$ então $permutation\ (a :: l)\ (insere\ a\ l')$. Ou seja, se l' é uma permutação de l então $(insere\ a\ l')$ é uma permutação de $(a :: l)$.*

A prova deste lema é, sem dúvida a prova mais bonita que apresentaremos aqui, mas antes observe que com ele concluímos de forma imediata a prova do Lema 178:

$$\dfrac{\dfrac{\dfrac{}{permutation\ tl\ (ord\_insercao\ tl)}\ (h.i.)}{permutation\ (h :: tl)\ (insere\ h\ (ord\_insercao\ tl))}\ (\textsc{Lema}\ 70)}{(*)}$$

Observe que a aplicação do Lema 70 foi feita instanciando a com h, l com tl, e l' com $(ord\_insercao\ tl)$.

Como exercício, reproduza esta prova em Coq:

Exercício 180. `Theorem ord_insercao_permutation: forall l, permutation l (ord_insercao l).`

Agora vamos fazer a prova do Lema 179:

Demonstração. Observe que o lema consiste em uma implicação: temos como hipótese $(permutation\ l\ l')$ e queremos provar $permutation\ (a :: l)\ (insere\ a\ l')$. Adicionalmente, o predicado *permutation* é indutivo (assim como os números naturais), e portanto podemos fazer a prova por indução na hipótese $(permutation\ l\ l')$. Isto significa que teremos um caso para cada regra da Definição 177.

(a) O primeiro caso é o da regra $(permutation\_nil)$: para que a hipótese $(permutation\ l\ l')$ tenha sido gerada por esta regra é preciso que tanto l quanto l' sejam a lista vazia. Nesta situação, o que queremos provar é $permutation\ (a :: nil)\ (insere\ a\ nil)$. Esta prova pode ser feita como a seguir:

$$\dfrac{\dfrac{\dfrac{}{permutation\ nil\ nil}\ (permutation\_nil)}{permutation\ (a :: nil)\ (a :: nil)}\ (permutation\_skip)}{permutation\ (a :: nil)\ (insere\ a\ nil)}\ (def.)$$

(b) A segunda regra é $(permutation\_skip)$, e para que a hipótese $(permutation\ l\ l')$ tenha sido gerada por esta regra é preciso que as listas l e l' tenha a mesma cabeça. Assim, considerando que l (resp. l') tenha a forma $h :: tl$ (resp. $h :: tl'$) temos como hipótese $permutation\ tl\ tl'$ (que corresponde ao antecedente da regra neste caso), e temos que provar $permutation\ (a :: h :: tl)\ (insere\ a\ (h :: tl'))$. Adicionalmente, temos como hipótese de indução $permutation\ (a :: tl)\ (insere\ a\ tl')$. Iniciamos a prova aplicando a definição de *insere* que divide a prova em dois subcasos: o da esquerda se dá quando $a \leq h$, e o da direita quando $a > h$.

$$\dfrac{\dfrac{\dfrac{(hip.)\ \dfrac{}{permutation\ tl\ tl'}}{(permutation\_skip)\ \dfrac{}{permutation\ (h :: tl)\ (h :: tl')}}}{(permutation\_skip)\ \dfrac{}{permutation\ (a :: h :: tl)\ (a :: h :: tl')}}\quad\dfrac{}{(*)}}{permutation\ (a :: h :: tl)\ (insere\ a\ (h :: tl'))}\ (def.)$$

No caso em que $a > h$ usamos a regra da transitividade com a lista $(h :: a :: tl)$ para poder permutar a e h no primeiro argumento de *permutation* (lista da esquerda), e então poder aplicar $(permutation\_skip)$ para concluir com a hipótese de indução.

$$\dfrac{\dfrac{\dfrac{}{permutation\ (a :: h :: tl)\ (h :: a :: tl)}\ \Delta\quad\dfrac{\dfrac{}{permutation\ (a :: tl)\ (insere\ a\ tl')}\ (h.i.)}{permutation\ (h :: a :: tl)\ (h :: (insere\ a\ tl'))}\ \clubsuit}{permutation\ (a :: h :: tl)\ (h :: (insere\ a\ tl'))}\ \nabla}{(*)}$$

onde

- Δ corresponde à regra ($permutation\_swap$);
- $\clubsuit$ corresponde à regra ($permutation\_skip$);
- ∇ corresponde à regra ($permutation\_trans$).

(c) A terceira regra é ($permutation\_swap$), e para que a hipótese ($permutation\ l\ l'$) tenha sido gerada por esta regra é preciso que as listas l e l' tenham a forma $x :: y :: tl$ e $y :: x :: tl$, respectivamente. Neste caso, precisamos provar $permutation\ (a :: x :: y :: tl)\ (insere\ a\ (y :: x :: tl))$. Note que não existe hipótese de indução neste caso porque a regra ($permutation\_swap$) (assim como a regra ($permutation\_nil$)) é um axioma. O ponto chave aqui é utilizar a transitividade de $permutation$ com a lista $(a :: y :: x :: tl)$:

$$\clubsuit\ \dfrac{\Delta\ \dfrac{}{permutation\ (x :: y : tl)\ (y :: x :: tl)}}{\dfrac{permutation\ (a :: x :: y : tl)\ (a :: y :: x :: tl)}{permutation\ (a :: x :: y :: tl)\ (insere\ a\ (y :: x :: tl))}\ (*)}\ \nabla$$

onde

- Δ corresponde à regra ($permutation\_swap$);
- $\clubsuit$ corresponde à regra ($permutation\_skip$);
- ∇ corresponde à regra ($permutation\_trans$).

e o ramo da direita é como a seguir:

$$\dfrac{\dfrac{?}{permutation\ (a :: y :: x :: tl)\ (insere\ a\ (y :: x :: tl))}}{(*)}$$

Este ponto da prova é semelhante ao que ocorreu no caso 2, e foi resolvido com a hipótese de indução. Mas neste caso não temos hipótese de indução, já que a regra $permutation\_swap$ é um axioma! Nossa alternativa será utilizar um novo resultado auxiliar:

Lema 181. *Seja x um número natural, e l uma lista de naturais. Então $permutation\ (x :: l)\ (insere\ x\ l)$.*

Este lema nos permite fechar o ramo de prova atual de forma imediata. Ele pode ser provado por indução na estrutura da lista l, e será deixado como exercício.

$$\dfrac{\dfrac{}{permutation\ (a :: y :: x :: tl)\ (insere\ a\ (y :: x :: tl))}\ (Lema\ 72)}{(*)}$$

(d) A quarta e última regra é ($permutation\_trans$), e considerando que a hipótese ($permutation\ l\ l'$) tenha sido gerada por esta regra, temos por hipótese que ($permutation\ l\ l0$) e ($permutation\ l0\ l'$) para alguma lista $l0$. Além disto, temos duas hipóteses de indução:

i. Se $permutation\ l\ l0$ então $permutation\ (a :: l)\ (insere\ a\ l0)$;

ii. Se *permutation* $l0\ l'$ então *permutation* $(a :: l0)$ $(insere\ a\ l')$.

A prova de *permutation* $(a :: l)$ $(insere\ a\ l')$ é como a seguir:

$$\clubsuit\ \frac{\dfrac{(hip.)\ \overline{permutation\ l\ l0}}{permutation\ (a :: l)\ (a :: l0)} \qquad \overset{(**)}{permutation\ (a :: l0)\ (insere\ a\ l')}}{permutation\ (a :: l)\ (insere\ a\ l')}\ \nabla$$

onde

- $\clubsuit$ corresponde à regra (*permutation_skip*);
- ∇ corresponde à regra (*permutation_trans*).

E o ramo da direita é concluído com a hipótese de indução:

$$\frac{(hip.)\ \overline{permutation\ l0\ l'} \qquad \overline{permutation\ l0\ l' \to permutation\ (a :: l0)\ (insere\ a\ l')}\ (h.i.)}{(**)}\ (\to_e)$$

□

Exercício 182. *Prove o Lema 181 em papel e lápis, e em seguida reproduza a sua prova no Coq.*

```
Lemma permutation_insere: forall l a, permutation (a :: l) (insere a l).
```

Exercício 183. *Prove o Lema 179 no Coq.*

```
Lemma permutation_insere_diff: forall l l' a, permutation l l' ->
                               permutation (a :: l) (insere a l').
```

Exercício 184. *Mostre que o predicado permutatio da Definição 177 é uma relação de equivalência sobre a estrutura de listas, isto é, mostre:*

(a) permutation l l, para qualquer lista l (reflexividade)

(b) Se permutation l l′ então permutation l′ l, quaisquer que sejam as listas l e l′ (simetria)

(c) Se permutation l l′ e permutation l′ l″ então permutation l l″, quaisquer que sejam as listas l, l′ e l″ (transitividade)

(d) Refaça suas provas no Coq:

```
Lemma permutation_refl: forall l, permutation l l.

Lemma permutation_sym: forall l l', permutation l l' -> permutation l' l.

Lemma permutation_trans: forall l l' l'', permutation l l' ->
                      permutation l' l'' -> permutation l l''.
```

Temos duas provas distintas de que o algoritmo de ordenação por inserção gera uma permutação da lista de entrada, mas veja que a prova foi muito mais simples e elegante com a Definição 177. Em geral, definições indutivas facilitam o processo de construção de provas porque podemos usar o princípio de indução para estas definições. Concluiremos esta seção com a prova de que as definições 172 e 177 são equivalentes, isto é, *perm l l′* se, e somente se, *permutation l l′* quaisquer que sejam as listas l e l'. Esta prova será dividida em duas etapas, isto é, em dois teoremas:

Teorema 185. *Sejam l e l′ duas listas de números naturais. Se permutation l l′ então perm l l′.*

Teorema 186. *Sejam l e l′ duas listas de números naturais. Se perm l l′ então permutation l l′.*

A prova do Teorema 185 segue a mesma estrutura da prova do Lema 179, isto é, indução na hipótese (*permutation l l′*) e será deixado como exercício:

Exercício 187. *Prove o Teorema 185.*

Exercício 188. *Prove o Teorema 185 em Coq:*

```
Lemma permutation_to_perm: forall l l', permutation l l' -> perm l l'.
```

A prova do Teorema 186 é mais desafiadora porque a definição *perm* não é indutiva, e portanto, não podemos utilizar a mesma estratégia do lema anterior.

Demonstração. A prova é por indução na estrutura da lista `l`. Na base de indução, precisamos provar que, se *perm nil l′* então *permutation nil l′*. A ideia é concluir da hipótese *perm nil l′* que l' é a lista vazia, e daí, fecharmos este ramo da prova com a regra (*permutation_nil*). Para isto vamos utilizar o seguinte lema auxiliar, cuja prova é deixada como exercício:

Lema 189. *Seja l uma lista de números naturais. Se perm nil l então l = nil.*

Exercício 190. *Prove o Lema 189, e em seguida refaça esta prova em Coq.*

```
Lemma perm_nil: forall l, perm nil l -> l = nil.
```

No passo indutivo, vamos supor que l tem a forma $(h :: tl)$. Então precisamos provar que, se *perm* $(h :: tl)$ l' então *permutation* $(h :: tl)$ l'. Como l' é uma lista arbitrária, precisamos analisar sua estrutura. Se l' for a lista vazia então a hipótese *perm* $(h :: tl)$ *nil* corresponde ao absurdo, e concluímos este ramo da prova já que podemos provar qualquer coisa a partir do absurdo (regra da explosão). Agora suponha que l' tem a forma $h' :: tl'$. Então temos que provar que, se *perm* $(h :: tl)$ $(h' :: tl')$ então *permutation* $(h :: tl)$ $(h' :: tl')$, e como hipótese de indução temos que se (*perm tl l*0) então (*permutation tl l*0), qualquer que seja a lista $l0$. Agora podemos comparar h e h', pois se eles forem iguais então podemos concluir este ramo da prova com a regra (*permutation_skip*) e com a hipótese de indução. Se $h \neq h'$ então, pela hipótese *perm* $(h :: tl)$ $(h' :: tl')$, sabemos que h ocorre na lista tl', ou seja, existem listas $l1$ e $l2$ tais que $tl' = l1 + +(h :: l2)$, onde $++$ representa a operação de concatenação de listas. Podemos usar esta igualdade para substituir tl' na implicação que temos que provar: *perm* $(h :: tl)$ $(h' :: l1 + +(h :: l2))$ então *permutation* $(h :: tl)$ $(h' :: l1 + +(h :: l2))$. Agora podemos remover h destas listas, e concluir a prova utilizando a hipótese de indução.

□

Repetir esta prova em Coq exige alguns detalhes adicionais que aparecem nos exercícios a seguir. Por exemplo, o lema `perm_cons_num_oc` do exercício a seguir nos fornece uma forma de dizer que `n` ocorre na lista `l'`:

Exercício 191.
```
Lemma perm_cons_num_oc: forall n l l', perm (n :: l) l' ->
                              exists x, num_oc  n l' = S x.
```

O exercício a seguir, nos permite reescrever a lista `l` sabendo que o elemento `x` ocorre pelo menos uma vez em `l`:

Exercício 192.
```
Lemma num_occ_cons: forall l x n, num_oc x l = S n ->
 exists l1 l2, l = l1 ++ x :: l2 /\ num_oc x (l1 ++ l2) = n.
```

A reorganização de elementos em uma lista pode ser feita com um lema como o do exercício a seguir:

Exercício 193.
```
Lemma permutation_app_cons: forall l1 l2 a,
 permutation (a :: l1 ++ l2) (l1 ++ a :: l2).
```

Utilizando estes exercícios como dica, refaça a prova do Teorema 186 em Coq:

Exercício 194.
```
Theorem perm_to_permutation: forall l l', perm l l' ->
                                    permutation l l'.
```

2. A complexidade do algoritmo de ordenação por inserção

Em algoritmos de ordenação sobre listas, o tamanho da entrada consiste no tamanho da lista a ser ordenada. Vamos iniciar nossa análise com a função *insere* x l (Definição 158). A operação básica no caso do algoritmo de ordenação por inserção é a comparação entre chaves. Note que quando l é a lista vazia, a lista unitária $x :: nil$ é retornada, e nenhuma comparação é feita. Quando l é uma lista da forma $h :: tl$ então comparamos x com h, e quando $x \leq h$ a lista $x :: h :: tl$ é retornada e o algoritmo termina. Por outro lado, se $x > h$, o algoritmo continua buscando recursivamente a posição correta para inserir x. Denotaremos por $T_{insere}\ x\ l$ a função que computa o número de operações básicas realizadas pela função *insere* para inserir o elemento x na lista l:

$$T_{insere}\ x\ l = \begin{cases} 0, & \text{se } l = nil \\ 1, & \text{se } l = h :: tl \text{ e } x \leq h \\ 1 + (T_{insere}\ x\ tl), & \text{se } l = h :: tl \text{ e } x > h \end{cases}$$

Normalmente estamos interessados na análise do pior caso, e a função acima não computa necessariamente o número de comparações do pior caso para inserir um elemento qualquer em uma lista com n elementos. De fato, considerando $n = 3$, temos que $T_{insere}\ 1\ 2 :: 3 :: 4 :: nil = 1$, enquanto que $T_{insere}\ 4\ 1 :: 2 :: 3 :: nil = 3$. Assim, para inserirmos um elemento em uma lista com n elementos, no pior caso, precisamos comparar o elemento a ser inserido com todos os elementos da lista. A função $T^w_{insere}(n)$ modela esta situação ao receber o natural n como argumento (que corresponde ao tamanho da lista a ser ordenada) e faz o número máximo de comparações possíveis:

$$T^w_{insere}(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1 + T^w_{insere}(n-1), & \text{se } n > 0 \end{cases}$$

Assim, se a lista não possui elementos, nenhuma comparação é feita ($n = 0$), e caso contrário, uma comparação é feita para cada elemento da lista ($n > 0$). Observe que o número de comparações feitas pela função $T^w_{insere}(n)$ não pode ser maior do que o tamanho da lista:

Exercício 195. *Prove que $T^w_{insere}(n) = n$, para todo n.*

Assim, a relação entre as funções T_{insere} e T^w_{insere} é dada pelo lema a seguir:

Exercício 196. *Sejam x um número natural, e l uma lista de números naturais. Prove que $T_{insere}\ x\ l \leq T^w_{insere}(|l|)$, onde $|l|$ denota o tamanho da lista l.*

Os dois últimos exercícios nos permitem concluir que $T_{insere}\ x\ l \leq |l|$, ou seja, que a função *insere* tem complexidade linear. Vamos formalizar este resultado em Coq. A função recursiva T_insere é definida por:

```
Fixpoint T_insere (x: nat) (l: list nat) : nat :=
 match l with
 | nil => 0
 | h :: tl => if (x <=? h) then 1 else S (T_insere x tl)
 end.
```

Como exercício, prove que a função `T_insere` tem complexidade linear:

Exercício 197. `Lemma T_insere_linear: forall l x, T_insere x l <= length l.`

Qual é o número de comparações realizadas pelo algoritmo de ordenação por inserção, isto é, pela função *ord_insercao*, para ordenar uma lista l? Vamos denotar por $T_{is}()$ a função que faz esta contagem. Se l for a lista vazia então nenhuma comparação é feita, ou seja, $T_{is}(nil) = 0$. Se $l = h :: tl$ então é feita uma chamada à função *insere*, além da chamada recursiva à função *ord_insercao*:

$$T_{is}(l) = \begin{cases} 0, & \text{se } l = nil \\ T_{is}(tl) + T_{insere}\ h\ (ord\_insercao\ tl), & \text{se } l = h :: tl \end{cases}$$

Observe que, $T_{is}(1 :: 2 :: 3 :: nil) = 2$, $T_{is}(3 :: 2 :: 1 :: nil) = 3$, $T_{is}(1 :: 2 :: 3 :: 4 :: nil) = 3$ e $T_{is}(4 :: 3 :: 2 :: 1 :: nil) = 6$, etc. Portanto o número de comparações pode ser diferente para listas de mesmo tamanho, o que é esperado pelas chamadas feitas à função *insere*. Como então definir a função $T^w_{is}(n)$ que nos dá um limite superior para o número de comparações feitas pelo algoritmo de ordenação por inserção para uma lista qualquer de tamanho n. Em outras palavras, qual a complexidade do pior caso para o algoritmo de ordenação por inserção? Sabemos que quando $n = 0$, nenhuma comparação é feita. Quando $n > 0$, o algoritmo é aplicado recursivamente na cauda da lista, isto é, em uma lista de tamanho $n-1$, e é feita uma chamada à função *insere* cuja complexidade já conhecemos. Isto nos permite escrever a função $T^w_{is}(n)$ como a seguir:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + T^w_{insere}(n-1), & \text{se } n > 0 \end{cases}$$

que pode ser simplificada como a seguir, já que $T^w_{insere}(n) = n$:

$$T^w_{is}(n) = \begin{cases} 0, & \text{se } n = 0 \\ T^w_{is}(n-1) + (n-1), & \text{se } n > 0 \end{cases}$$

Podemos usar o método da substituição para encontrarmos uma solução para esta recorrência, e em seguida utilizar indução para verificarmos se a solução está correta. Pelo método da substituição, podemos ir aplicando a definição da recorrência, assumindo que $n > 0$:

$$\begin{aligned} T^w_{is}(n) &= T^w_{is}(n-1) + (n-1) \\ &= T^w_{is}(n-2) + (n-2) + (n-1) \\ &= T^w_{is}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \end{aligned}$$

Podemos continuar este processo de substituição até chegarmos em $T^w_{is}(1)$ que é igual a 0:

$$\begin{aligned} T^w_{is}(n) &= T^w_{is}(n-1) + (n-1) \\ &= T^w_{is}(n-2) + (n-2) + (n-1) \\ &= T^w_{is}(n-3) + (n-3) + (n-2) + (n-1) \\ &= \ldots \\ &= T^w_{is}(1) + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= 0 + 1 + 2 + \ldots + (n-3) + (n-2) + (n-1) \\ &= \textstyle\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \end{aligned}$$

Para finalizar, precisamos utilizar indução em n para provar que $T^w_{is}(n) = \frac{n(n-1)}{2}$. Se $n = 0$, o resultado é trivial. Se $n > 0$ então, por

definição, $T^w_{is}(n) = T^w_{is}(n-1)+(n-1)$. A hipótese de indução, nos dá que $T^w_{is}(n-1) = \frac{(n-1)(n-2)}{2}$, e portanto, $T^w_{is}(n) = T^w_{is}(n-1) + (n-1) \stackrel{h.i.}{=} \frac{(n-1)(n-2)}{2} + (n-1) = \frac{n(n-1)}{2}$.

Agora prove este lema em Coq:

Exercício 198.
```
Fixpoint T_is_w (n: nat) : nat :=
 match n with
 | 0 => 0
 | S k => (T_is_w k) + (T_insere_w k)
 end.

Lemma T_ord_insercao_w_teste: forall n, T_is_w (S n) = n * (S n)/2.
```

Nossa conclusão, portanto, é que o algoritmo de ordenação por inserção recursivo é correto, e sua complexidade no pior caso é quadrática, assim como na versão não-recursiva. Na próxima seção estudaremos um algoritmo mais eficiente do que a ordenação por inserção, mas antes vejamos um exercício clássico envolvendo recursão: a função de Fibonacci.

Exercício 199. *A função de Fibonacci é definida pela relação de recorrência*

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-1) + F(n-2) \end{aligned}$$

Seja $\mathcal{F}$ uma função geradora definida por

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} F(i) z^i$$

Siga os seguintes passos para resolver a relação de recorrência da função de Fibonacci.

(a) Demonstre que

$$\mathcal{F}(z) = \frac{z}{1-z-z^2} = \frac{z}{(1-\phi z)(1-\tilde{\phi} z)} = \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\tilde{\phi} z}\right),$$

onde

$$\phi = \frac{1+\sqrt{5}}{2} \quad e \quad \tilde{\phi} = \frac{1-\sqrt{5}}{2}$$

(b) Demonstre que

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}} (\phi^i - \tilde{\phi}^i) z^i$$

(c) Prove que $F(i) = \phi^i/\sqrt{5}$ para $i > 0$, aproximado ao inteiro mais próximo.

(d) Demonstre que $F(i+2) \geq \phi^i$, para $i \geq 0$.

(e) Conclua que $F(n) = \Theta(\phi^n)$; i.e., a função de Fibonacci é de complexidade exponencial.

5.4 O algoritmo *mergesort* (versão com *merge* iterativo)

Algoritmos recursivos desempenham um papel fundamental em Computação. O algoritmo de ordenação *mergesort* é um exemplo de algoritmo recursivo, que se caracteriza por dividir o problema original em subproblemas que, por sua vez, são resolvidos recursivamente. As soluções dos subproblema são então combinadas para gerar uma solução para o problema original. Este paradigma de projeto de algoritmo é conhecido com *divisão e conquista*. Este algoritmo foi inventado por J. von Newmann em 1945.

O algoritmo *mergesort* é um algoritmo de ordenação que utiliza a técnica de divisão e conquista, que consiste das seguintes etapas:

1. **Divisão**: O algoritmo divide a lista (ou vetor) l recebida como argumento ao meio, obtendo as listas l_1 e l_2;

2. **Conquista**: O algoritmo é aplicado recursivamente às listas l_1 e l_2 gerando, respectivamente, as listas ordenadas l'_1 e l'_2;

3. **Combinação**: O algoritmo combina as listas l'_1 e l'_2 através da função *merge* que então gera a saída do algoritmo.

Por exemplo, ao receber a lista $(4 :: 2 :: 1 :: 3 :: nil)$, este algoritmo inicialmente divide esta lista em duas sublistas, a saber $(4 :: 2 :: nil)$ e $(1 :: 3 :: nil)$. O algoritmo é aplicado recursivamente às duas sublistas para ordená-las, e ao final deste processo, teremos duas listas ordenadas $(2 :: 4 :: nil)$ e $(1 :: 3 :: nil)$. Estas listas são, então, combinadas para gerar a lista de saída $(1 :: 2 :: 3 :: 4 :: nil)$.

```
1 if p < r then
2 |   q = ⌊(p+r)/2⌋;
3 |   mergesort(A, p, q);
4 |   mergesort(A, q + 1, r);
5 |   merge(A, p, q, r);
6 end
```

Algorithm 8: mergesort(A, p, r)

A etapa de combinar dois vetores ordenados (algoritmo *merge*) é a etapa principal do algoritmo *mergesort*. O procedimento $merge(A, p, q, r)$ descrito a seguir recebe como argumentos o vetor A, e os índices p, q e r tais que $p \leq q < r$. O procedimento assume que os subvetores $A[p..q]$ e $A[q+1..r]$ estão ordenados.

```
1  n_1 = q − p + 1 ;                                   // Qtd. de elementos em A[p..q]
2  n_2 = r − q ;                                       // Qtd. de elementos em A[q + 1..r]
3  let L[1..n_1 + 1] and R[1..n_2 + 1] be new arrays;
4  for i = 1 to n_1 do
5  |  L[i] = A[p + i − 1];
6  end
7  for j = 1 to n_2 do
8  |  R[j] = A[q + j];
9  end
10 L[n_1 + 1] = ∞;
11 R[n_2 + 1] = ∞;
12 i = 1;
13 j = 1;
14 for k = p to r do
15 |  if L[i] ≤ R[j] then
16 |  |  A[k] = L[i];
17 |  |  i = i + 1;
18 |  end
19 |  else
20 |  |  A[k] = R[j];
21 |  |  j = j + 1;
22 |  end
23 end
```

Algorithm 9: merge(A, p, q, r)

Exercício 200. *Prove que o algoritmo merge é correto.*

Exercício 201. *Prove que o algoritmo mergesort é correto.*

Exercício 202. *Faça a análise assintótica do algoritmo merge.*

Exercício 203. *Faça a análise assintótica do algoritmo mergesort.*

5.5 O algoritmo *mergesort* (versão com *merge* recursivo)

A seguir apresentamos uma descrição do algoritmo *mergesort* diretamente em Coq:

```
Function mergesort (l: list nat) {measure length l}:=
 match l with
 | nil => nil
 | h::nil => l
 | h::tl =>
   let n := length(l) / 2 in
    let l1 := firstn n l in
    let l2 := skipn n l in
    let sorted_l1 := mergesort(l1) in
    let sorted_l2 := mergesort(l2) in
    merge (sorted_l1, sorted_l2)
 end.
```

A definição é baseada na estrutura da lista `l`, de forma que se `l` for a lista vazia ou uma lista unitária, o algoritmo retorna a própria lista `l`. Caso contrário `l` é dividida nas listas `l1` (contendo os elementos

da primeira metade de `l`), e `l2` (contendo os elementos restantes de `l`). Recursivamente, as listas `l1` e `l2` são ordenada para depois serem combinadas pela função `merge`:

```
Function merge (p: list nat * list nat) {measure len p} :=
 match p with
 | (nil, l2) => l2
 | (l1, nil) => l1
 | ((hd1 :: tl1) as l1, (hd2 :: tl2) as l2) =>
 if hd1 <=? hd2 then hd1 :: merge (tl1, l2)
 else hd2 :: merge (l1, tl2)
end.
```

A função `merge` recebe um par de listas ordenadas, e recursivamente gera uma lista ordenada contendo todos os elementos das listas dadas como argumento.

Exercício 204. *Prove que o algoritmo merge é correto.*

Exercício 205. *Prove que o algoritmo mergesort é correto.*

Exercício 206. *Faça a análise assintótica do algoritmo merge.*

Exercício 207. *Faça a análise assintótica do algoritmo mergesort.*

Agora provaremos a correção do algoritmo `mergesort` no Coq. Inicialmente mostrar que a função `merge` retorna uma lista ordenada, caso cada uma das listas do par dado como argumento também estejam ordenadas:

Exercício 208.
```
Lemma merge_sorts: forall p, (sorted (fst p) /\ sorted (snd p)) ->
     sorted (merge p).
```

Em seguida, podemos provar que a função `mergesort` efetivamente ordena e que gera uma permutação de qualquer lista recebida como argumento:

Exercício 209. `Theorem mergesort_sorts: forall l, sorted (mergesort l).`

Exercício 210. `Theorem mergesort_is_perm: forall l, perm l (mergesort l).`

Observe que podemos utilizar tanto `perm` como `permutation` no exercício anterior, já que estas definições são equivalentes. Feito isto, temos o teorema da correção do algoritmo `mergesort`:

```
Theorem mergesort_is_correct: forall l, perm l (mergesort l) /\ sorted (mergesort l).
Proof.
intro. split.
 - apply mergesort_is_perm.
 - apply mergesort_sorts.
Qed.
```

Assim, como fizemos para o algoritmo de ordenação por inserção, analisaremos a complexidade do algoritmo `mergesort` considerando o número de comparações realizadas pelo algoritmo durante o processo de ordenação. Para a função `merge`, chamaremos de `T_merge` a função que faz esta contagem:

```
Function T_merge (p: list nat * list nat) {measure len p} : nat :=
 match p with
 | (nil, l2) => 0
 | (l1, nil) => 0
 | ((hd1 :: tl1) as l1, (hd2 :: tl2) as l2) =>
   if hd1 <=? hd2 then S(T_merge (tl1, l2))
   else S(T_merge (l1, tl2))
 end.
```

Quando alguma das listas do par é a lista vazia, nenhuma comparação é feita, e portanto a função `T_merge` retorna 0. Caso contrário, o contador é incrementado e uma nova chamada de `T_merge` é feita. No exercício a seguir, prove que a função `merge` tem complexidade linear:

Exercício 211. `Lemma T_merge_is_linear: forall l1 l2, T_merge (l1,l2) <= (length l1 + length l2).`

Agora vamos contar o número de comparações feitas pela função `mergesort`. Quando a lista `l` tem no máximo um elemento, nenhuma comparação é feita. Quando `l` tem pelo menos dois elementos, a lista é divida em duas listas `l1` e `l2` e recursivamente contamos as comparações necessárias para ordená-las e também o número de comparações necessárias para juntar as versões ordenadas de `l1` e `l2`. Esta contagem está implementada na função `T_mergesort` a seguir:

```
Function T_mergesort (l: list nat) {measure length l} : nat :=
 match l with
 | nil => 0
 | hd :: nil => 0
 | hd :: tl =>
   let n := length(l) / 2 in
   let l1 := firstn n l in
   let l2 := skipn n l in
   T_mergesort(l1) + T_mergesort(l2) + T_merge (mergesort l1, mergesort l2)
 end.
```

Por fim, resolva o exercício a seguir que mostra que a complexidade do algoritmo `mergesort` é $O(\log_2 n)$, onde n é o tamanho da lista a ser ordenada.

Exercício 212. `Theorem T_mergesort_complexity: forall l k, length l = 2^k -> T_mergesort l <= k * 2^k.`

5.6 Equações de recorrência

Nesta seção estudaremos as equações de recorrência utilizadas no paradigma de divisão de conquista [24]:

Definição 213. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* eventualmente não-decrescente *se existir um número inteiro n_0 tal que $f(n)$ é não-decrescente no intervalo $[n_0, \infty)$, ou seja,*

$$f(n_1) \leq f(n_2), \forall n_2 > n_1 \geq n_0.$$

Definição 214. *Seja $f(n)$ uma função não-negativa definida no conjunto dos números naturais. Dizemos que $f(n)$ é* suave *se for eventualmente não-decrescente e*

$$f(2.n) = \Theta(f(n))$$

Teorema 215. *Sejam $f(n)$ uma função suave, e c e n_0 constantes positivas. Se $f(2n) \leq c.f(n), \forall n \geq n_0$ então $f(2^k n) \leq c^k.f(n), \forall n \geq n_0$ e $k \geq 1$.*

Teorema 216. *Seja $f(n)$ uma função suave. Então para qualquer $b \geq 2$ fixado,*

$$f(b.n) = \Theta(f(n))$$

O teorema a seguir é conhecido como *regra da suavização*

Teorema 217. *Seja $T(n)$ uma função eventualmente não-decrescente, e $f(n)$ uma função suave. Se $T(n) = \Theta(f(n))$ para valores de n que são potências de b ($b \geq 2$), então*

$$T(n) = \Theta(f(n)), \forall n.$$

A regra da suavização nos permite expandir a informação sobre a ordem de crescimento estabelecida para $T(n)$ de um subconjunto de valores (potências de b) para o domínio inteiro. O teorema a seguir é um resultado muito útil nesta direção conhecido como *teorema mestre*:

Teorema 218. *Seja $T(n)$ uma função eventualmente não-decrescente que satisfaz a recorrência*
$T(n) = a.T(n/b) + f(n), \quad$ *para* $n = b^k, k = 1, 2, 3, \ldots$
$T(1) = c$
onde $a \geq 1, b \geq 2$ e $c \geq 0$. Se $f(n) = \Theta(n^d)$, onde $d \geq 0$, então

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & se\ a > b^d \\ \Theta(n^d . \lg n), & se\ a = b^d \\ \Theta(n^d), & se\ a < b^d \end{cases}$$

Demonstração. Considere que $f(n) = n^d$. Aplicando o método da substituição para a recorrência do teorema, obtemos:

$$T(b^k) = a^k.[T(1) + \sum_{j=1}^{k} f(b^j)/a^j]$$

Como $a^k = a^{\log_b n} = n^{\log_b a}$, podemos reescrever a equação acima como:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} f(b^j)/a^j]$$

e para $f(n) = n^d$, temos:

$$T(n) = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^j)^d/a^j] = n^{\log_b a}.[T(1) + \sum_{j=1}^{\log_b n} (b^d/a)^j]$$

A soma acima forma uma série geométrica, e portanto:

$$\sum_{j=1}^{\log_b n} (b^d/a)^j = (b^d/a)\frac{(b^d/a)^{\log_b n} - 1}{(b^d/a) - 1}, \text{ se } b^d \neq a.$$

Quando $b^d \neq a$, temos que $\sum_{j=1}^{\log_b n} (b^d/a)^j = \log_b n$. Agora basta analisarmos cada um dos casos: $a < b^d$, $a > b^d$ e $a = b^d$.

□

Apresentaremos agora uma versão um pouco mais geral do teorema mestre[12]. Consideraremos como anteriormente uma recorrência da forma:

$$T(n) = a.T(n/b) + f(n)$$

on $a \geq 1$ e $b > 1$ são constantes, e $f(n)$ é uma função assintoticamente positiva.

Teorema 219. *Sejam $a \geq 1$ e $b \geq 2$ constantes, $f(n)$ uma função assintoticamente positiva, e $T(n)$ definida nos inteiros não-negativos pela recorrência $T(n) = a.T(n/b) + f(n)$, onde n/b deve ser interpretado como $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$. Então $T(n)$ tem as seguintes cotas assintóticas:*

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a}.\lg n)$;*

3. *Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para alguma constante $\epsilon > 0$, e se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$, então para todo n suficientemente grande, temos que $T(n) = \Theta(f(n))$.*

A prova será dividida em três lemas, onde inicialmente consideraremos que n é potência de b.

Lema 220. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. Defina $T(n)$ para potências de b pela recorrência:*

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1; \\ a.T(n/b) + f(n), & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j.f(n/b^j).$$

Demonstração. Analise a árvore de recorrência da equação dada.

□

Em termos da árvore de recorrência, os três casos do teorema mestre correspondem aos casos onde o custo total da árvore é:

1. dominado pelo custo das folhas;

2. uniformemente distribuído ao longo da árvore;

3. dominado pelo custo da raiz.

Lema 221. *Sejam $a \geq 1$ e $b > 1$ constantes, $f(n)$ uma função não-negativa definida para potências de b. A função $g(n)$ definida para potências de b por:*

$$g(n) = \sum_{j=0}^{\log_b n - 1} a^j . f(n/b^j).$$

tem as seguintes cotas assintóticas para potências de b:

1. *Se $f(n) = O(n^{\log_b a - \epsilon})$ para alguma constante $\epsilon > 0$, então $g(n) = O(n^{\log_b a})$;*

2. *Se $f(n) = \Theta(n^{\log_b a})$, então $g(n) = \Theta(n^{\log_b a} . \lg n)$;*

3. *Se $a.f(n/b) \leq c.f(n)$ para alguma constante $c < 1$ e para todo n suficientemente grande, então $g(n) = \Theta(f(n))$.*

Demonstração. Exercício.

□

Exercício 222. *Resolva as seguintes relações de recorrência:*

1. $T(1) = 1$, $T(n) = 3T(n/2) + n^2$, $n \geq 2$

2. $T(1) = 1$, $T(n) = 2T(n/2) + n$, $n \geq 2$

3. $T(1) \in \Theta(1)$, $T(n) = 3T(n/3 + 5) + n/2$

4. $T(1) = 1$, $T(n) = 2T(n-1) + 1$, $n \geq 2$

5. $T(1) \in \Theta(1)$, $T(n) = 9T(n/3) + n$

6. $T(1) \in \Theta(1)$, $T(n) = T(2n/3) + 1$

7. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + 1$

8. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + \sqrt{n}$

9. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + \sqrt{n}\lg^2 n$

10. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + n$

11. $T(1) \in \Theta(1)$, $T(n) = 2T(n/4) + n^2$

12. $T(1) \in \Theta(1)$, $T(n) = 3T(n/2) + nln(n)$

13. $T(1) \in \Theta(1)$, $T(n) = 3T(n/4) + nln(n)$

14. $T(1) \in \Theta(1)$, $T(n) = 2T(n/2) + nln(n)$

15. $T(1) \in \Theta(1)$, $T(n) = 2T(n/2) + n/ln(n)$

16. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + 1/n$

17. $T(1) \in \Theta(1)$, $T(n) = T(n-1) + ln(n)$

18. $T(1) \in \Theta(1)$, $T(n) = \sqrt{n}T(\sqrt{n}) + n$

19. $T(n) = 8T(n/2) + \Theta(n^2)$

20. $T(n) = 8T(n/2) + \Theta(1)$

21. $T(n) = 7T(n/2) + \Theta(n^2)$

Capítulo 6

Divisão e Conquista

6.1 O algoritmo *quicksort*

O algoritmo *quicksort*, assim como *merge sort*, utiliza o paradigma de divisão e conquista para ordenar um vetor $A[1..n]$, mas diferentemente de *mergesort*, seu foco está no particionamento e não na combinação das soluções. Este algoritmo, que foi inventado pelo cientista da computação britânico Charles Antony Richard Hoare em 1960, utiliza as seguintes etapas para ordenar um subvetor $A[p..r]$ $(1 \leq p \leq r \leq n)$:

1. **Dividir**: Esta etapa consiste em particionar o vetor $A[p..r]$ em dois subvetores (possivelmente vazios) $A[p..q-1]$ e $A[q+1..r]$ tais que cada elemento do subvetor $A[p..q-1]$ é menor ou igual a $A[q]$, que por sua vez também é menor ou igual do que cada elemento do subvetor $A[q+1..r]$. Esta etapa também computa o índice q do particionamento.

2. **Conquistar**: Esta etapa consiste em recursivamente ordenar os subvetores $A[p..q-1]$ e $A[q+1..r]$.

Assim, a ideia do algoritmo pode ser apresentada a partir do pseudocódigo a seguir:

```
1 if p < r then
2 |   q = partition(A, p, r);
3 |   quicksort(A, p, q − 1);
4 |   quicksort(A, q + 1, r);
5 end
```

Algorithm 10: $quicksort(A, p, r)$

A ordenação do vetor $A[1..n]$ é, então, obtida pela chamada $quicksort(A, 1, n)$.

O particionamento do vetor consiste na principal etapa do algoritmo *quicksort*:

```
1  x = A[r];
2  i = p − 1;
3  for j = p to r − 1 do
4  |   if A[j] ≤ x then
5  |   |   i = i + 1;
6  |   |   exchange A[i] com A[j];
7  |   end
8  end
9  exchange A[i + 1] with A[r];
10 return i + 1;
```

Algorithm 11: $partition(A, p, r)$

O algoritmo *partition* reorganiza o subvetor $A[p..r]$ *in place*, ou seja, sem alocar espaço adicional, e retorna o índice da posição que divide o subvetor $A[p..r]$ entre os elementos que são menores ou iguais

ao pivô, e maiores do que o pivô.

Exercício 223. *Prove a seguinte invariante de laço, e conclua que o algoritmo* partition *é correto:*

> *Antes de cada iteração do laço* ***for*** *(linhas 3-8), para todo k, temos:*
>
> 1. *Se $p \leq k \leq i$, então $A[k] \leq x$;*
> 2. *Se $i+1 \leq k \leq j-1$, então $A[k] > x$;*
> 3. *Se $k = r$, então $A[k] = x$.*

O número de comparações (linha 4) feitas por *partition* em um vetor com n elementos é $\Theta(n)$. Qual seria, então, o tempo de execução de *quicksort*? O pior caso de *quicksort* ocorre quando o particionamento gera um vetor vazio, e outro com $n-1$ elementos. Neste caso, dizemos que o particionamento é *desbalanceado*. Se assumirmos que este desbalanceamento ocorre em cada chamada recursiva, podemos modelar o tempo de execução $T_w(n)$ pela seguinte equação de recorrência:

$$T_w(n) = T_w(n-1) + T_w(0) + \Theta(n)$$

Como não há trabalho a fazer em um vetor vazio, temos que $T(0) = \Theta(1)$, e portanto a recorrência acima pode ser reescrita como:

$$T_w(n) = T_w(n-1) + \Theta(n) \tag{6.1}$$

Como exercício, mostre que $T_w(n) = \Theta(n^2)$, e observe que esta situação ocorre, por exemplo, quando o vetor dado como argumento já está ordenado.

A apresentação que acabamos de fazer é informal porque estamos o particionamento desbalanceado nos dá o pior caso para o algoritmo *quicksort*. Mas será que não pode existir uma combinação de particionamentos que seja pior do que quadrática? A seguir provaremos que não, isto é, mostraremos formalmente que a análise do pior caso para o algoritmo *quicksort* é, de fato, quadrática. Inicialmente estabeleceremos a cota superior. Para isto, considere a recorrência

$$T(n) \leq \max_{0 \leq q < n} (T(q) + T(n-q-1)) + \Theta(n) \tag{6.2}$$

onde $0 \leq q \leq n-1$.

A partir da solução vista anteriormente quando $q = 0$, parece razoável acreditar que $T(n) \leq c.n^2$ para alguma constante positiva c. Substituindo esta possível solução, temos

$$T(n) \leq \max_{0 \leq q < n} (c(q^2 + (n-q-1)^2) + \Theta(n) = c. \max_{0 \leq q < n} (q^2 + (n-q-1)^2) + \Theta(n)$$

e a expressão $q^2 + (n-q-1)^2$ assume valor máximo quando $q = 0$ ou $q = n-1$ (Exercício!), portanto

$$T(n) \leq c. \max_{0 \leq q < n} (q^2 + (n-q-1)^2) + \Theta(n) \leq (n-1)^2$$

Assim, podemos escrever $T(n) \leq c.n^2 - c.(2n-1) + \Theta(n) \leq c.n^2$, já que podemos tomar c grande o suficiente para dominar $\Theta(n)$. Ou seja,
$T(n) = O(n^2)$. A prova da cota inferior fica como exercício:

Exercício 224. *Mostre que a recorrência* $T(n) = \max_{0 \leq q < n} (T(q) + T(n-q-1)) + \Theta(n)$ *tem solução* $T(n) = \Theta(n^2)$. *Considerando o que já foi feito anteriormente, falta apenas mostrar que* $T(n) = \Omega(n^2)$.

Por outro lado, quando o particionamento é balanceado temos o melhor caso para *quicksort*. Agora, o particionamento divide o problema original em dois subproblemas sendo um de tamanho $\lfloor \frac{n}{2} \rfloor$, e outro de tamanho $\lceil \frac{n}{2} - 1 \rceil$, o que nos dá a recorrência:

$$T_b(n) = T_b(\lfloor \frac{n}{2} \rfloor) + T_b(\lceil \frac{n}{2} - 1 \rceil) + \Theta(n)$$

Se ignorarmos as funções de aproximação e a subtração por 1, temos a recorrência:

$$T_b(n) = 2.T_b(\frac{n}{2}) + \Theta(n)$$

que, pelo Teorema Mestre, tem solução $T_b(n) = \Theta(n. \lg n)$.

Assim, o tempo de execução de *quicksort* depende, por exemplo, do fato do particionamento estar balanceado ou não: Em caso afirmativo, *quicksort* é assintoticamente tão rápido quanto *mergesort*, mas se o particionamento não estiver balanceado, *quicksort* tem o mesmo comportamento assintótico de *Insertion sort* no pior caso.

O que ocorre se o particionamento é sempre da ordem de 9-1, *i.e.* 90% dos elementos são menores ou iguais ao pivô, e apenas 10% são maiores do que o pivô? (Exercício!) A resposta desta pergunta sugere que a complexidade do caso médio para *quicksort* está mais próxima do melhor caso do que do pior caso.

Exercício 225. *Mostre que a complexidade de* quicksort*, no melhor caso, é* $\Omega(n \lg n)$.

Exercício 226. *Mostre que o algoritmo* quicksort *é correto.*

Capítulo 7

Heapsort

Estudaremos um novo algoritmo de ordenação baseado em comparação de chaves, mas bem diferente dos algoritmos (*insertion sort* e *mergesort*) vistos anteriormente. Este novo algoritmo, conhecido como *heapsort*, possui tempo de execução $O(n.\log n)$, como *mergesort*, e o processo de ordenação é feito *in place* (como em *insertion sort*). Portanto *heapsort* combina as vantagens de *insertion sort* e *mergesort*. A estrutura de dados utilizada por este algoritmo é conhecida como *heap*:

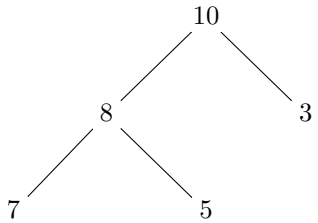
Definição 227. *Um* heap *(binário) T é uma estrutura de dados que corresponde a uma árvore binária com chaves associadas aos nós, sendo uma chave por nó, que satisfaz às seguintes condições:*

1. *T é uma árvore binária completa em todos os níveis, exceto possivelmente o último nível;*

2. *Todos os caminhos para uma folha do último nível estão à esquerda de todos os caminhos para uma folha do penúltimo nível;*

3. *A chave de cada nó é maior ou igual do que a chave dos seus filhos.*

Os itens 1 e 2 da definição acima caracterizam a chamada de **propriedade do corpo do *heap***. O item 3 corresponde a **propriedade de *heap***. Assim, em um *heap* o nó mais à direita pode ter apenas um filho à esquerda, mas não pode ter somente um filho à direita. Todos os outros nós internos possuem dois filhos.



A grande vantagem da estrutura de *heap* é que ela permite a implementação das operações de inserção de um novo elemento (ou uma nova chave), e extração do maior elemento (maior chave) em tempo logarítmico. Note que em um vetor (ou em uma lista) contendo n elementos, a inserção pode ser feita em tempo constante, mas a extração do maior elemento vai exigir, no pior caso, uma busca em todo o vetor (ou lista), o que tem custo linear. A estrutura de *heap* suporta simultaneamente as duas operações, e como veremos, de forma assintoticamente mais eficiente do que em listas ou vetores.

Um *heap* binário pode ser implementado como um subvetor de um vetor A, onde somente os elementos em $A[1..A.\text{heap-size}]$ $(0 \leq A.\text{heap-size} \leq A.length)$ são elementos válidos do *heap*. A raiz do *heap* é $A[1]$, e dado o índice i de um nó, o índice do filho à esquerda (resp. direita) é dado por $2i$ (resp. $2i+1$).

Por fim, o índice do nó correspondente ao pai do nó de índice i é dado por $\lfloor i/2 \rfloor$.

Alguns autores chamam a estrutura definida acima de *heap* de máximo (ou *max-heap*), isto é, um *heap* onde todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \geq A[i]$. Nesta linha, com um ajuste na propriedade de *heap* podemos definir um *heap* de mínimo (ou *min-heap*):

Em um *min-heap* todo nó i diferente da raiz é tal que $A[\lfloor i/2 \rfloor] \leq A[i]$.

Desta forma, o maior (resp. menor) elemento de um *max-heap* (resp. *min-heap*) é armazenado na raiz, e a subárvore com raiz em um determinado nó contém apenas valores que são menores ou iguais (resp. que são maiores ou iguais) ao valor deste nó. O algoritmo *heapsort* utiliza *max-heaps*, e portanto o primeiro passo do algoritmo será transformar o vetor A de entrada em um *max-heap*. Este trabalho é feito pelo algoritmo a seguir:

```
1  A.heap-size ← A.length;
2  for i = ⌊A.length/2⌋ downto 1 do
3  |  Max-Heapify(A,i);
4  end
```

Algorithm 12: Build-Max-Heap(A)

Exercício 228. *Mostre que na representação vetorial de um* heap *com n elementos, as folhas são os elementos do vetor com índices $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.*

O algoritmo Build-Max-Heap constrói o *max-heap* de baixo para cima a partir do primeiro vértice que não é uma folha, e o algoritmo Max-Heapify(A,i) reconstrói um *max-heap* a partir de uma árvore cuja raiz $A[i]$ seja o único elemento que precise ser reposicionado, ou seja, as subárvores com raiz $A[2i]$ e $A[2i+1]$ já são *max-heaps*:

```
1  l = 2i;
2  r = 2i + 1;
3  if l ≤ A.heap-size and A[l] > A[i] then
4  |  largest = l;
5  end
6  else
7  |  largest = i;
8  end
9  if r ≤ A.heap-size and A[r] > A[largest] then
10 |  largest = r;
11 end
12 if largest ≠ i then
13 |  exchange A[i] with A[largest];
14 |  Max-Heapify(A,largest);
15 end
```

Algorithm 13: Max-Heapify(A,i)

Observe que cada uma das subárvores com raiz nas posições $2i$ e $2i+1$ têm, no máximo, $2n/3$ elementos:

Exercício 229. *Mostre que, em um* heap *com n elementos e raiz $A[i]$, cada uma das subárvores com raiz em $2i$ e $2i+1$ têm, no máximo, $2n/3$ elementos.*

Portanto o tempo de execução de Max-Heapify é dado pela recorrência

$$T(n) \leq T(2n/3) + \Theta(1) \tag{7.1}$$

que, pelo teorema mestre, tem solução $O(\lg n)$.

Qual o tempo de execução do procedimento Build-Max-Heap(A)? Temos a seguinte cota superior, considerando um *heap* com n elementos: $\sum_{i=1}^{n/2} O(\lg n) = O(\lg n . \sum_{i=1}^{n/2} 1) = O(\lg n.(n/2) = O(n. \lg n)$. No entanto, esta cota, apesar de correta, não é a mais precisa que podemos encontrar. De fato, se observarmos que:

1. A altura do n-ésimo elemento de um *heap* é igual a $\lfloor \lg n \rfloor$.

2. Um *heap* com n elementos possui, no máximo, $\lceil n/2^{h+1} \rceil$ nós com altura h.

Então, observando que Max-Heapify tem complexidade $O(h)$ quando executado em um nó de altura h, concluímos que o tempo de execução de Build-Max-Heap(A), assumindo que A possui n elementos, tem a seguinte cota superior: $\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil n/2^{h+1} \rceil \cdot O(h) = O(n. \sum_{h=0}^{\lfloor \lg n \rfloor} \lceil h/2^{h+1} \rceil) = O(n)$, pois $\sum_{h=0}^{\lfloor \lg n \rfloor} h/2^{h+1} \leq \sum_{h=0}^{\infty} h/2^{h+1}$, que por sua vez converge. Assim, um *heap* pode ser construído em tempo linear.

Exercício 230. *Prove a seguinte invariante de laço, e conclua que o algoritmo Build-Max-Heap é correto:*

> *No início de cada iteração do laço* ***for*** *(linhas 2-4), cada nó nas posições $i+1$, $i+2$, ..., n é a raiz de um* max-heap.

O algoritmo *heapsort* recebe como argumento um vetor A qualquer contendo $n > 0$ elementos, e inicialmente o transforma em um *max-heap*. Neste momento, sabemos que a raiz do *heap* contém o maior elemento do vetor A, que pode então ser movido para sua posição correta. Em seguida, decrementamos o tamanho do *heap* em uma unidade, e repetimos o processo:

```
1 Build-Max-Heap(A);
2 for i = A.length downto 2 do
3 |   exchange A[1] with A[i];
4 |   A.heap-size = A.heap-size - 1;
5 |   Max-Heapify(A,1);
6 end
```

Algorithm 14: Heapsort(A)

A complexidade de Heapsort(A) no pior caso, se A é um vetor com $n > 0$ elementos, é $O(n) + \sum_{i=2}^{n} O(\lg n) = O(n) + O(\lg n . \sum_{i=2}^{n} 1) = O(n) + O((n-1). \lg n) = O(n. \lg n)$.

Exercício 231. *Mostre que a complexidade de tempo de Max-Heapify no pior caso é $\Omega(\lg n)$.*

Exercício 232. *Mostre que a complexidade de tempo de* Heapsort *no pior caso é $\Omega(n \lg n)$, e conclua que a complexidade de* Heapsort *é $\Theta(n. \lg n)$.*

Exercício 233. *Prove a correção do algoritmo* Heapsort *utilizando a seguinte invariante:*

No início de cada iteração do laço ***for*** *(linhas 2-6), o subvetor* $A[1..i]$ *é um* max-heap *que contém os* i *menores elementos do vetor* $A[1..n]$*, e o subvetor* $A[i+1..n]$ *está ordenado e contém os* $n-i$ *maiores elementos do vetor* $A[1..n]$*.*

Capítulo 8

Cota inferior para algoritmos baseados na comparação de chaves

Os algoritmos de ordenação vistos são baseados na comparação de chaves, ou seja, a operação básica destes algoritmos é a comparação entre elementos do vetor ou lista que se quer ordenar. Neste contexto, vimos que *Insertion Sort* e *Quicksort* têm, no pior caso, complexidade de tempo quadrática, *i.e.*, estão em $O(n^2)$. Já *Merge sort* é capaz de, no pior caso, ordenar um vetor com n elementos em tempo $O(n.\lg n)$. A dúvida que surge naturalmente agora é: seria possível construir um algoritmo baseado na comparação de chaves que consiga ordenar um vetor com n elementos, no pior caso, realizando menos do que $c.n.\lg n$ comparações (para alguma constante positiva c? Ou seja, seria possível melhorar a cota $O(n.\lg n)$ no pior caso?

Para responder esta pergunta, assumiremos que os n elementos $x_1, x_2, \ldots, x_n$ a serem ordenados são distintos. Além disto, o processo de ordenação será modelado por *árvores de decisão*, que são árvores binárias completas. Assim, para construirmos a árvore de decisão para um algoritmo A (baseado na comparação de chaves) que recebe como entrada um vetor com n elementos distintos, anotaremos seus nós internos com $i : j$, para $1 \leq i, j \leq n$, e cada folha com a permutação $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$. A execução do algoritmo corresponde a um caminho da raiz até uma folha. Cada nó interno corresponde a comparação $a_i \leq a_j$ (a rigor $a_i < a_j$ já que os elementos são distintos), e a subárvore à esquerda indica as comparações subsequentes. A subárvore à direita indica as comparações subsequentes quando $a_i > a_j$, e uma folha indica que o algoritmo ordenou o vetor de forma que $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi(n)}$. Como qualquer algoritmo de ordenação correto tem que ser capaz de produzir todas as permutações de uma entrada, cada uma das $n!$ possíveis permutações do vetor de tamanho n dado com entrada tem que aparecer como uma folha. O caminho mais longo da raiz de uma árvore de decisão até uma folha representa o número de comparações no pior caso do algoritmo em consideração. Ou seja, o número de comparações no pior caso de um algoritmo corresponde a altura de sua árvore de decisão. A cota inferior da altura de todas as possíveis árvores de decisão será então a cota inferior, no pior caso, para qualquer algoritmo de ordenaçãqo baseado na comparação de chaves.

Teorema 234. *Qualquer algoritmo baseado na comparação de chaves requer $\Omega(n.\lg n)$ comparações no pior caso.*

Demonstração. Precisamos determinar a altura de uma árvore de decisão onde cada permutação do vetor de entrada apareça como uma folha. Considere a árvore de decisão de altura h contendo l folhas de um algoritmo para ordenar n elementos distintos. Como cada uma das $n!$ permutações do vetor de entrada aparece como uma folha, temos que $n! \leq l$, e como uma árvore binária de altura h não possui mais do que 2^h folhas, temos que $n! \leq l \leq 2^h$. Portanto $h \geq \lg(n!) = \Omega(n.\lg n)$.

□

Exercício 235. *Sejam l o número de folhas em uma árvore binária, e h sua altura. Prove que $l \leq 2^h$.*

Exercício 236. *Prove que* $\lg(n!) = \Theta(n.\lg n)$.

Capítulo 9

Ordenação em Tempo Linear

Nesta seção veremos uma outra forma de ordenação baseada na ideia de contagem. O que precisamos fazer é contar, para cada elemento a ser ordenado, o número total de elementos que são menores do que este elemento e guardar este resultado em uma tabela. Os valores da tabela indicarão a posição dos elementos na lista ordenada. Por exemplo, se existem 6 elementos menores do que o elemento x, então x deve ser colocado na sétima posição do vetor ordenado. Desta forma podemos ordenar um vetor simplesmente deslocando os elementos para a posição correta. O pseudocódigo a seguir implementa esta ideia:

```
1  let C[0..n − 1] be a new array;
2  for i = 0 to n − 1 do
3  |  C[i] ← 0;
4  end
5  for i = 0 to n − 2 do
6  |  for j = i + 1 to n − 1 do
7  |  |  if A[i] < A[j] then
8  |  |  |  C[j] ← C[j] + 1;
9  |  |  end
10 |  |  else
11 |  |  |  C[i] ← C[i] + 1;
12 |  |  end
13 |  end
14 end
15 for i = 0 to n − 1 do
16 |  B[C[i]] ← A[i];
17 end
18 return B;
```

Algorithm 15: comparison-counting-sort($A[0..n-1]$)

Esta ideia não parece ser muito interessante já que a implementação acima tem complexidade quadrática e ainda utiliza espaço adicional, ou seja a complexidade de espaço é linear no tamanho n da entrada. No entanto, podemos utilizar a ideia da contagem de forma mais eficiente se os elementos a serem ordenados forem "conhecidos". Por exemplo, se o vetor a ser ordenado contém apenas 0s e 1s então podemos utilizar esta informação para fazer a ordenação sem a necessidade de fazer comparações porque com uma única passagem sobre o vetor podemos obter o número k de 0s, e assim retornar o vetor contendo 0s da posição 0 até $k-1$, e 1s da posição k em diante. De uma forma mais geral, se os elementos a serem ordenados são inteiros entre l e h então podemos computar a frequência de cada um destes valores, e armazená-las em um vetor, digamos $C[0..h-l]$, de forma que as primeiras $C[0]$ posições do vetor ordenado serão preenchidas com l, as $C[1]$ posições seguintes com $l+1$, e assim por diante. Observe que se os elementos do vetor original não puderem ser sobrescritos então precisaremos de um novo vetor (espaço adicional) para escrever o vetor ordenado.

O algoritmo counting-sort a seguir, assume que cada um dos n inteiros a serem ordenados estão no intervalo entre l e h:

```
1  let C[0..h − l] be a new array;
2  for i = 0 to h − l do
3  |  C[i] ← 0;
4  end
5  for i = 0 to n − 1 do
6  |  C[A[i] − l] ← C[A[i] − l] + 1;
7  end
8  for j = 1 to h − l do
9  |  C[j] ← C[j] + C[j − 1];
10 end
11 for i = n − 1 downto 0 do
12 |  j ← A[i] − l;
13 |  B[C[j] − 1] ← A[i];
14 |  C[j] ← C[j] − 1;
15 end
16 return B;
```

Algorithm 16: counting-sort$(A[0..n-1], l, h)$

Qual é a complexidade deste algoritmo? Por simplicidade denotaremos $k = h - l$, ou seja, k denota o tamanho do intervalo que contém os elementos a serem ordenado. Assim, o laço **for** das linhas 2-4 é executado em tempo $\Theta(k)$, o laço das linhas 5-7 em tempo $\Theta(n)$, o laço das linhas 8-10 em tempo $\Theta(k)$, e por fim o laço das linhas 11-14 em tempo $\Theta(n)$, o que perfaz um total de $\Theta(n + k)$. Assumindo que $k = O(n)$, temos que o tempo de execução de counting-sort é $\Theta(n)$.

Definição 237. *Um algoritmo de ordenação é dito* estável *se não altera a posição relativa dos elementos que têm o mesmo valor.*

Exercício 238. *Prove que o algoritmo counting-sort é estável.*

Exercício 239. *Prove que o algoritmo* mergesort *é estável.*

Exercício 240. *Prove que o algoritmo* insertion sort *é estável.*

9.1 *Radix Sort*

O algoritmo radix-sort ordena uma sequência de inteiros com d dígitos cada, em tempo linear. Ele utiliza um algoritmo auxiliar, que precisa ser estável, para ordenar a sequência de inteiros do dígito menos significativo para o mais significativo. Podemos utilizar counting-sort, por exemplo, como algoritmo auxiliar.

```
1 for i = 1 to d do
2 |  use a stable sort algorithm to sort array A on digit i;
3 end
```

Algorithm 17: radix-sort(A, d)

Lema 241. *Dados n números com d dígitos, que por sua vez podem assumir até k valores, radix-sort ordena corretamente estes números em tempo $\Theta(d.(n + k))$, se o algoritmo auxiliar estável tem complexidade de tempo $\Theta(n + k)$.*

Exercício 242. *Mostre como podemos ordenar n inteiros contidos no intervalo de 0 a* $n^2 - 1$ *em tempo linear, ou seja, em* $O(n)$*.*

Demonstração. Inicialmente iteramos pela lista dos números, convertendo cada um deles para a base n. Depois aplicamos o algoritmo radix-sort sobre a lista, utilizando o counting-sort como o algoritmo de ordenação dos dígitos da lista de números. Dada a nova base, cada número possuirá 2 dígitos, uma vez que $\log_n(n^2) = 2$, onde cada dígito estará em um intervalo entre 0 e $n - 1$. Sabendo disso, vemos que radix-sort ordenará n números de 2 dígitos, usando o counting-sort em um intervalo de 0 à $n - 1$, resultando em uma complexidade de $O(2(n + n)) = O(n)$.

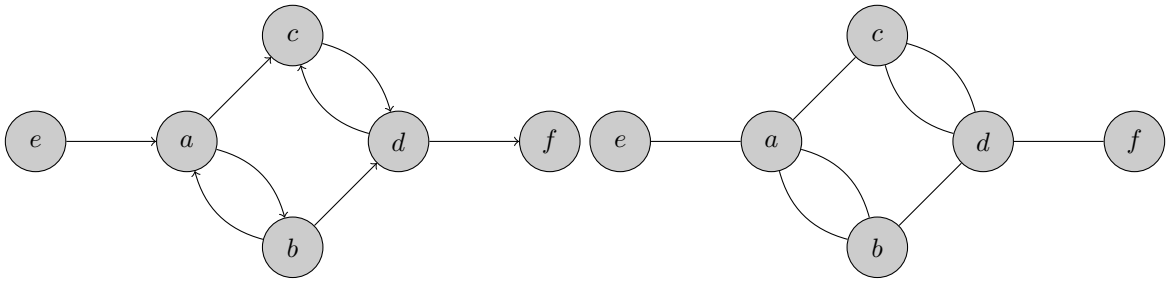
□

Exercício 243. *Mostre como podemos ordenar n inteiros contidos no intervalo de 0 a* $n^3 - 1$ *em tempo linear, ou seja, em tempo* $O(n)$*.*

Capítulo 10

Algoritmos em Grafos

Neste capítulo estudaremos algoritmos sobre grafos. Inicialmente, veremos como podemos representar um grafo, seja ele dirigido ou não. Informalmente, um grafo G é um par (V, E), onde V é o conjunto de vértices, e E é o conjunto de arestas. Quando as arestas do grafo são dirigidas, falamos em digrafo. A seguir, apresentamos um exemplo de um digrafo à esquerda, e um grafo à direita:



Existem duas formas bastante comuns para representar um (di)grafo $G = (V, E)$: matriz de adjacências ou listas de adjacências. As duas representações se aplicam a grafos e a digrafos.

Nesta seção estudaremos alguns algoritmos em grafos. Iniciaremos com uma sequência de definições para fixar a nomenclatura a ser utilizada.

Definição 244. *Um* grafo *(não dirigido) G é um par (V, E) onde V é um conjunto finito não-vazio, e E é um conjunto de pares não-ordenados de elementos de V. Em grafos não-dirigidos arestas de um vértice para ele mesmo (*auto-loop*) são proibidas, e portanto toda aresta liga dois vértices distintos.*

Definição 245. *Um* digrafo *(ou um grafo dirigido) G é um par (V, E) onde V é um conjunto finito não-vazio, e E é uma relação binária sobre V. Em digrafos* auto-loops *são permitidos.*

Dado um grafo $G = (V, E)$, a versão dirigida de G é o digrafo $G' = (V, E')$ onde $(u, v) \in E'$ se, e somente se $(u, v) \in E$, isto é, substituímos cada aresta $(u, v) \in E$ pelas duas arestas dirigidas (u, v) e (v, u) na versão dirigida.

Dado um digrafo $G = (V, E)$, a versão não-dirigida (ou o *grafo associado*) de G é o digrafo $G' = (V, E')$ onde $(u, v) \in E'$ se, e somente se $u \neq v$ e $(u, v) \in E$, ou seja, a versão não-dirigida de um grafo é construída removendo a direção das arestas e os auto-loops.

Definição 246. *A representação de um grafo $G = (V, E)$ por* listas de adjacências *consiste de um vetor Adj de $|V|$ listas, uma para cada vértice em V. Para cada $u \in V$, a lista de adjacências $Adj[u]$ contém*

todos os vértices v tais que $(u, v) \in E$, ou seja, contém todos os vértices adjacentes a u em G. Escreveremos $G.Adj[u]$ para se referir a lista $Adj[u]$ de G.

Se G é um digrafo então a soma dos comprimentos de todas as listas de adjacências é igual a $|E|$, já que uma aresta (u, v) é representada por uma ocorrência de v em $Adj[u]$. Se G for um grafo (não-dirigido) então a soma dos comprimentos de todas as listas de adjacências é igual a $2|E|$ pois uma aresta (u, v) é representada pela ocorrência de v em $Adj[u]$, e pela ocorrência de u em $Adj[v]$. Em ambos os casos, a representação por listas de adjacências utiliza espaço da ordem de $\Theta(V + E)$.

Uma desvantagem das listas de adjacências é que elas não fornecem uma forma rápida de determinar se a aresta (u, v) está ou não no (di)grafo. Para isto precisamos procurar por v em $Adj[u]$. A representação por matrizes de adjacências contorna este problema a um custo assintoticamente maior de espaço.

Definição 247. *A representação de um grafo $G = (V, E)$ por* matrizes de adjacências *assume uma enumeração (qualquer) $1, 2, \ldots, |V|$ dos vértices de G, e consiste de uma matriz $A = (a_{ij})$ de dimensão $|V| \times |V|$ tal que*

$$a_{ij} = \begin{cases} 1, & \text{se } (i, j) \in E \\ 0, & \text{caso contrário.} \end{cases} \tag{10.1}$$

A representação por matrizes de adjacências requer espaço da ordem de $\Theta(V^2)$, independentemente do número de arestas do grafo.

Diversas definições coincidem para grafos e digrafos, mas algumas diferenças podem ocorrer dependendo do contexto. Por exemplo, se (u, v) é uma aresta de um digrafo $G = (V, E)$ então dizemos que (u, v) sai de u, e entra (ou incide) em v. Já quando (u, v) é uma aresta de um grafo, dizemos que (u, v) incide em u e v.

Definição 248. *O* grau *de um vértice em um grafo é o número de arestas que incidem sobre ele. Um vértice de grau 0 é dito* isolado. *Em um digrafo, o* grau de saída *(resp.* grau de entrada*) de um vértice é o número de arestas que saem (resp. chegam) neste vérice. O* grau *de um vértice em um digrafo é a soma dos seus graus de saída e entrada.*

Se (u, v) é uma aresta do (di)grafo $G = (V, E)$ então dizemos que v é adjacente a u. Note que a relação de adjacência é simétrica em grafos, mas não em digrafos. De fato, se um digrafo $G = (V, E)$ possui a aresta (u, v), mas não possui a aresta (v, u) então v é adjacente a u, mas u não é adjacente a v.

Um grafo (não-dirigido) é dito *completo* se qualquer par de vértices é adjacente.

Definição 249. *Um* caminho *de comprimento k de um vértice u para um vértice v em um grafo $G = (V, E)$ é uma sequência $\langle v_0, v_1, \ldots, v_k \rangle$ de vértices tal que $v_0 = u$ e $v_k = v$, e $(v_{i-1}, v_i) \in E$ para $i = 1, 2, \ldots, k$. O comprimento de um caminho é o número de arestas deste caminho. Existe sempre um caminho de comprimento 0 de u para u, qualquer que seja o vértice u. Um* subcaminho *de um caminho $p = \langle v_0, v_1, \ldots, v_k \rangle$ é uma sequência contígua dos vértices de p, isto é, quaisquer que sejam $0 \leq i \leq j \leq k$, a subsequência de vértices $\langle v_i, v_{i+1}, \ldots, v_j \rangle$ é um subcaminho de p.*

Quando existe um caminho p de u para v, dizemos que v é alcançável a partir de u, o que normalmente é denotado por $u \overset{p}{\rightsquigarrow} v$, quando o grafo é dirigido.

Definição 250. *Um caminho é dito* simples *se todos os vértices no caminho são distintos.*

A definição de ciclos em grafos requer cuidado porque difere para grafos e digrafos. Em um digrafo, um *ciclo* é um caminho não-nulo, ou seja, de comprimento estritamente maior do que 0, tal que o primeiro e o último vértices são idênticos. Em um digrafo, um caminho $\langle v_0, v_1, \ldots, v_k \rangle$ forma um *ciclo* se $v_0 = v_k$, e este caminho possui pelo menos uma aresta. Dois caminhos $\langle v_0, v_1, \ldots, v_{k-1}, v_0 \rangle$ e $\langle v'_0, v'_1, \ldots, v'_{k-1}, v'_0 \rangle$ formam o mesmo ciclo se existir j tal que $v'_i = v_{(i+j) \mod k}$ para $i = 0, 1, \ldots, k-1$. Um auto-loop é um ciclo de comprimento 1, e um digrafo sem auto-loops é dito *simples*. Em um grafo, as definições são similares, mas existe um requerimento adicional de que se qualquer aresta aparece mais de uma vez, então ela aparece com a mesma orientação: em um caminho $\langle v_0, v_1, \ldots v_{k-1}, v_k \rangle$, se $v_i = x$ e $v_{i+1} = y$ para $0 \leq i < k$, então não pode existir j tal que $v_j = y$ e $v_{j+1} = x$. Um ciclo é dito *simples* se seus vértices são distintos. Um (di)grafo sem ciclos é dito *acíclico*.

Um grafo acíclico é chamado de *floresta* (não-dirigida), e se o grafo for conexo então é chamado de *árvore* (livre ou não-dirigida). Um digrafo acíclico é normalmente abreviado por DAG. Nenhuma condição de conectividade é assumida em DAGs.

Definição 251. *Um* caminho eulerianos *em um (di)grafo conexo G é um caminho que percorre cada aresta apenas uma vez, mas vértices podem ser visitados mais de uma vez. Um* caminho hamiltoniano *em um (di)grafo G é um caminho simples que contém cada vértice de G. Um* ciclo hamiltoniano *em um (di)grafo G é um ciclo simples que contém cada vértice de G.*

Note que em um ciclo hamiltoniano cada vértice do (di)grafo é visitado um única vez. Em um grafo (não dirigido) um caminho $\langle v_0, v_1, \ldots, v_k \rangle$ forma um ciclo se $k \geq 3$ e $v_0 = v_k$.

A definição de *conectividade* exige mais cuidado porque difere entre grafos e digrafos:

- Um grafo é dito *conexo* se para cada par de vértices v e w, existe um caminho entre v e w, ou seja, se qualquer vértice é alcançável a partir de todos os outros. As *componentes conexas* de um grafo são as classes de equivalência dos vértices sob a relação "é alcançável a partir de". Assim, um grafo é conexo se possui apenas uma componente conexa.

- A conectividade em digrafos é dividida em dois casos:

  - Um digrafo é *fortemente conexo* se o vértice u é alcançável a partir do vértice v, e vice-versa, quaisquer que sejam $u, v \in V$. As componentes fortemente conexas de um digrafo são as classes de equivalência dos vértices sob a relação "são mutuamente alcançáceis". Um digrafo é fortemente conexo se possui apenas uma componente fortemente conexa.

  - Um digrafo é *fracamente conexo* se o grafo associado é conexo, mas não é fortemente conexo.

Dois grafos $G = (V, E)$ e $G' = (V', E')$ são *isomorfos* se existir uma bijeção $f : V \to V'$ tal que $(u, v) \in E$ se, e somente se $(f(u), f(v)) \in E'$. Isto significa que podemos renomear os vértices de G como sendo os de G' mantendo as arestas correspondentes em G e G'. Dizemos que $G' = (V', E')$ é um *subgrafo* de $G = (V, E)$, notação $H \subseteq G$, se $V' \subseteq V$ e $E' \subseteq E$. Alguns subgrafos especiais:

- Um subgrafo H de um grafo G é dito *gerador* (*spanning*) se contém todos os vértices de G, isto é, se $H.V = G.V$ usando a notação de atributos[1].

[1] Sempre que for conveniente, utilizaremos a notação de atributos $G.V$ (resp. $G.E$) para denotar o conjunto dos vértices (resp. das arestas) do grafo G.

- Um subgrafo H de um grafo G é *próprio*, notação $H \subset G$, se for diferente de G, isto é, se $H.V < G.V$ ou $H.E < G.E$.

- Dado $X \subseteq G.V$, o subgrafo de G *induzido* por X, notação $G[X]$, é o grafo $G' = (X, E')$ onde $E' = \{(u, v) \in E : u, v \in X\}$.

- Dado $Y \subseteq G.E$, o subgrafo de G *induzido* por Y, notação $G[Y]$, é o grafo $G' = (V', Y)$ onde se $(u, v) \in Y$ então $u, v \in V'$.

Ao considerarmos o tempo de execução de um algoritmo sobre um (di)grafo $G = (V, E)$, normalmente consideramos tanto o número de vértices $|V|$, como o número de arestas $|E|$ como parâmetros para considerar o tamanho da entrada. É usual o abuso de notação $O(VE)$ quando se quer dizer $O(|V|.|E|)$, uma vez que a leitura se torna mais fácil e não há risco de ambiguidade.

10.1 Busca em Largura

Busca em largura (BFS) é um dos algoritmos mais simples para busca em grafos, além de ser utilizado em outros algoritmos sobre grafos. O algoritmo funciona tanto para grafos quanto para digrafos. Dado um grafo $G = (V, E)$ e um vértice $s \in V$ que chamaremos de *origem*, o algoritmo de busca em largura sistematicamente explora as arestas de G para descobrir todos os vértices que são alcançáceis a partir de s. O nome *busca em largura* se dá porque o algoritmo separa a fronteira entre os vértices que já foram descobertos dos que ainda não o foram a partir da sua distância até a origem s. Assim, o algoritmo descobre todos os vértices que estão a uma distância k da origem antes de descobrir qualquer vértice que esteja a uma distância $k + 1$ da origem. Este algoritmo nos permite responder dois tipos de questões:

1. Existe um caminho do vértice s para o vértice u?

2. Qual é o menor caminho do vértice s para o vértice u?

A seguir apresentamos o pseudocódigo do algoritmo BFS, que recebe como argumento o grafo G, e o vértice s a partir do qual a busca é iniciada.

```
1  for each vertex u ∈ G.V − {s} do
2  |  u.color ← WHITE;
3  end
4  s.color ← GRAY;
5  Q ← ∅;
6  enqueue(Q, s);
7  while Q ≠ ∅ do
8  |  u ← dequeue(Q);
9  |  for each v ∈ G.Adj[u] do
10 |  |  if v.color = WHITE then
11 |  |  |  v.color ← GRAY;
12 |  |  |  enqueue(Q, v);
13 |  |  end
14 |  end
15 end
```

Algorithm 18: BFS(G, s)

A inicialização (linhas 1-3) percorre todos os vértices, exceto s, e portanto tem tempo de execução limitado pelo número de vértices do grafo, i.e. $\Theta(V)$. As operações das linhas 4, 5 e 6 são executadas em

tempo constante. O *loop* das linhas 7-15 precisa de uma análise mais cuidadosa. Os vértices marcados com WHITE durante a inicialização correspondem aos vértices que ainda não foram visitados e, uma vez que um vértice é visitado, ele é imediatamente marcado com GRAY (linha 11) e inserido na fila Q (linha 12). Durante a primeira execução do *loop*, s é retirado da fila, e cada um dos vértices da lista de adjacências de s é marcado como visitado (GRAY), e então colocado na fila Q. Assim, o percorrimento do grafo inicia pelo vértice s, e em seguida são percorridos todos os vértices adjacentes a s perfazendo um total de $1 + |Adj[s]|$ vértices visitados nesta etapa. Na segunda execução do laço, um vértice adjacente a s, digamos u, é retirado da fila Q e todos os vértices adjacentes a u, que ainda não tenham sido visitados, serão marcados como visitados, e inseridos na fila, de forma que, no máximo, $|Adj[u]|$ vértices serão visitados porque alguns dos elementos em $Adj[u]$ já podem ter sido visitados: de fato, se um vértice v for simultaneamente adjacente aos vértices s e u, que por sua vez é também adjacente a s, então v será inserido na fila na primeira execução do laço, i.e. durante o percorrimento de $Adj[s]$, e será ignorado durante o percorrimento de $Adj[u]$. Portanto $1 + |Adj[s]| + |Adj[u]|$ é uma cota superior para o número de vértices visitados até este momento. Note que um vértice só é inserido na fila uma única vez. Mais ainda, para que um vértice seja inserido na fila Q é necessário que ele seja alcançável a partir de s, e portanto, o processo de enfileiramento e desenfileiramento tem custo limitado por $O(V)$. Cada um dos vértices enfileirados terá sua lista de adjacências percorrida, mas apenas alguns de seus elementos serão visitados. Assim, se $\{s, v_1, v_2, \ldots, v_k\}$ é o conjunto de todos os vértices alcançáveis a partir de s no grafo, então todos eles serão inseridos na fila Q logo após serem visitados, o que tem custo limitado por $O(V)$. Em seguida, para cada vértice $u \in \{s, v_1, v_2, \ldots, v_k\}$, os vértices de $Adj[u]$ que ainda não foram visitados são marcados com GRAY, o que tem custo $O(Adj[u])$. Somando este custo para cada um dos vértices do conjunto $\{s, v_1, v_2, \ldots, v_k\}$, temos custo limitado por $\sum_{u \in \{s, v_1, v_2, \ldots, v_k\}} |Adj[u]| \leq \sum_{u \in V} |Adj[u]| = \Theta(E)$. Assim, o custo total para percorrer o grafo é limitado por $O(V + E)$.

Mais sucintamente: depois da inicialização de BFS (linhas 1-5) nenhum vértice volta a ser marcado com WHITE. Então o teste da linha 13 garante que cada vértice é enfileirado apenas uma vez, e portanto desenfileirado apenas uma vez também. As operações de enfileiramento e desenfileiramento tomam tempo constante $\Theta(1)$, e portanto o tempo requerido pela operação de enfileiramento é da ordem de $O(V)$. Como BFS percorre a lista de adjacências somente quando o vértice é desenfileirado, concluímos que cada lista de adjacências é percorrida apenas uma vez. Como a soma dos comprimentos das listas de adjacências é $\Theta(E)$, o tempo total utilizado no percorrimento das listas de adjacências é limitado por $O(E)$. Portanto BFS possui tempo de execução limitado por $O(V + E)$, isto é, é linear no tamanho da representação de G. Observe que se $|E| \geq |V|$ então $|V| + |E| \leq |E| + |E| = 2.|E|$, e portanto neste caso, $O(V + E)$ significa $O(E)$. Analogamente, se $|E| < |V|$ então $O(V + E)$ significa $O(V)$. Em geral, $O(x + y)$ significa $O(max(x, y))$.

Exercício 252. *Considere uma enumeração qualquer* $1, 2, \ldots, |V|$ *dos vértices de* G. *A matriz de adjacências* $G.A$ *de dimensão* $|V| \times |V|$ *é dada por:*

$$G.A[i][j] = \begin{cases} 1, & \text{se } (i,j) \in G.E \\ 0, & \text{caso contrário.} \end{cases} \tag{10.2}$$

O pseudocódigo a seguir apresenta o algoritmo BFS onde o grafo G *é representado por sua matriz de adjacências:*

```
1  for i = 1 to |V| do
2  |  i.color ← WHITE;
3  end
4  s.color ← GRAY;
5  Q ← ∅;
6  enqueue(Q, s);
7  while Q ≠ ∅ do
8  |  u ← dequeue(Q);
9  |  for i = 1 to |V| do
10 |  |  if G.A[u][i] = 1 and i.color = WHITE then
11 |  |  |  i.color ← GRAY;
12 |  |  |  enqueue(Q, i);
13 |  |  end
14 |  end
15 end
```

Algorithm 19: BFS(G, s)

Qual é a complexidade de tempo de BFS neste caso?

1. Caminhos de comprimento mínimo

   O algoritmo BFS também computa a menor distância (menor número de arestas) de s até cada um dos vértices que são alcançáveis a partir de s. Por fim, o algoritmo constrói uma árvore com raiz s que contém todos os vértices alcançáveis a partir de s.

```
1  for each vertex u ∈ G.V − {s} do
2  |  u.color ← WHITE;
3  |  u.d ← ∞;
4  |  u.π ← NIL;
5  end
6  s.color ← GRAY;
7  s.d ← 0;
8  s.π ← NIL;
9  Q ← ∅;
10 enqueue(Q, s);
11 while Q ≠ ∅ do
12 |  u ← dequeue(Q);
13 |  for each v ∈ G.Adj[u] do
14 |  |  if v.color = WHITE then
15 |  |  |  v.color ← GRAY;
16 |  |  |  v.d ← u.d + 1;
17 |  |  |  v.π ← u;
18 |  |  |  enqueue(Q, v);
19 |  |  end
20 |  end
21 |  u.color ← BLACK;
22 end
```

Algorithm 20: BFS(G, s)

Denote por $\delta(s, v)$ o número de arestas do *caminho de comprimento mínimo* de s para v, isto é, o menor número de arestas de qualquer caminho de s para v, e $\delta(s, v) = \infty$ se não existe caminho de s para v.

Lema 253. *Seja $G = (V, E)$ um (di)grafo, e $s \in V$. Então para qualquer aresta $(u, v) \in E$, temos $\delta(s, v) \leq \delta(s, u) + 1$.*

Lema 254. *Seja $G = (V, E)$ um (di)grafo, e $s \in V$. Então após a execução de BFS(G, s), para cada vértice $v \in V$, temos que $v.d \geq \delta(s, v)$.*

Lema 255. *Suponha que durante a execução de BFS no grafo $G = (V, E)$, a fila Q contenha os vértices $v_1, v_2, \ldots, v_r$, i.e. $Q = \langle v_1, v_2, \ldots, v_r \rangle$, onde v_1 é o primeiro elemento, e v_r, o último. Então $v_r.d \leq v_1.d + 1$ e $v_i.d \leq v_{i+1}.d$ para todo $i = 1, 2, \ldots, r - 1$.*

Corolário 256. *Suponha que o vértice v_i tenha sido enfileirado antes do vértice v_j durante a execução do algoritmo BFS. Então $v_i.d \leq v_j.d$ no momento em que v_j é enfileirado.*

Por fim, o teorema a seguir estabelece a correção do algoritmo BFS:

Teorema 257. *Seja $G = (V, E)$ um (di)grafo, e considere a execução de BFS em G a partir da origem $s \in V$. Então, durante sua execução, BFS descobre cada vértice $v \in V$ que é alcançável a partir de s, e após o término de sua execução, $v.d = \delta(s, v), \forall v \in V$. Adicionalmente, para cada vértice $v \neq s$ alcançável a partir de s, um dos menores caminhos de s para v é o menor caminho de s para $v.\pi$ seguido da aresta $(v.\pi, v)$.*

2. Árvores BFS

   O algoritmo BFS constrói uma árvore, a chamada *árvore BFS*, à medida que percorre o grafo. A árvore é obtida a partir do atributo π de cada vértice. Para construirmos esta árvore consideraremos inicialmente o *subgrafo predecessor* de G como sendo o grafo $G_\pi = (V_\pi, E_\pi)$, onde

   - $V_\pi = \{v \in V : v.\pi \neq NIL\} \cup \{s\}$
   - $E_\pi = \{(v.\pi, v) : v \in V_\pi \backslash \{s\}\}$

   O subgrafo predecessor G_π é a árvore BFS de G se V_π consiste de todos os vértices alcançáveis de G a partir de s, e $\forall v \in V_\pi$ o subgrafo G_π contém um único caminho simples de s para v que é o menor caminho de s para v em G.

   Lema 258. *Quando aplicado ao (di)grafo $G = (V, E)$, o algoritmo BFS constrói π de forma que o subgrafo predecessor $G_\pi = (V_\pi, E_\pi)$ é uma árvore BFS.*

10.2 Busca em Profundidade

Nesta seção estudaremos outro algoritmo de busca em grafos, o algoritmo DFS:

```
1  for each vertex u ∈ G.V do
2  |  u.color = WHITE;
3  |  u.π = NIL;
4  end
5  time = 0;
6  for each vertex u ∈ G.V do
7  |  if u.color == WHITE then
8  |  |  DFS-Visit(G, u)
9  |  end
10 end
```

Algorithm 21: DFS(G)

```
1  time = time + 1;
2  u.d = time;
3  u.color = GRAY;
4  for each v ∈ G.Adj[u] do
5  |  if v.color == WHITE then
6  |  |  v.π = u;
7  |  |  DFS-Visit(G, v)
8  |  end
9  end
10 u.color = BLACK;
11 time = time + 1;
12 u.f = time;
```

Algorithm 22: DFS-Visit(G, u)

Qual o tempo de execução de DFS? O laço das linhas 1-4 é executado em tempo $\Theta(V)$. Já o laço das linhas 6-10 exige um pouco mais de atenção porque este laço faz uma chamada ao algoritmo DFS-visit. Então vamos determinar o custo de DFS-Visit primeiro. Em cada execução de DFS-Visit(G, v), o *loop* das linhas 4-9 é executado $|Adj[v]|$ vezes. Como

$$\sum_{v \in V} |Adj[v]| = \Theta(E)$$

o custo total de DFS-Visit é $\Theta(E)$, e portanto, o custo total de DFS é $\Theta(V + E)$.

Definição 259. *O* subgrafo predecessor *da busca em profundidade é definido por:*

- $G_\pi = (V, E_\pi)$*, onde* $E_\pi = \{(v.\pi, v) : v \in V \text{ and } v.\pi \neq NIL\}$

O subgrafo predecessor de uma busca em profundidade forma uma floresta.

Teorema 260. *Para dois vértices u e v quaisquer de um (di)grafo G, apenas uma das seguintes propriedades ocorre em uma busca em profundidade (DFS) em G, considerando que $u.d < v.d$:*

1. *$u.d < v.d < v.f < u.f$, e v é um descendente de u no subgrafo predecessor de G;*

2. *$u.d < u.f < v.d < v.f$, e u não é um descendente de v no subgrafo predecessor de G, ou vice-versa.*

Teorema 261. *Seja $G = (V, E)$ um grafo, e considere a floresta F obtida após a execução de DFS(G). As componentes de F são precisamente as componentes conexas de G.*

Corolário 262. *As componentes conexas de um grafo $G = (V, E)$ podem ser encontradas em tempo $\Theta(V + E)$.*

Capítulo 11

NP-completude

Praticamente todos os algoritmos estudados até aqui são polinomiais, i.e. o tempo de execução destes algoritmos no pior caso está em $O(p(n))$, onde $p(n)$ é um polinômio no tamanho da entrada n[11, 34]. Estes algoritmos formam a classe dos algoritmos que são considerados "eficientes". Os problemas que podem ser resolvidos por algoritmos polinomiais são chamados de "tratáveis". No entanto, vimos algoritmos cujos tempos de execução são exponenciais no tamanho da entrada, como por exemplo, busca em grafos utilizando força bruta. De fato, suponha que, dados um digrafo G e vértices u e v de G, queremos determinar quando existe um caminho de u para v em G. Utilizando um algoritmo força bruta, precisamos considerar todos os potenciais caminhos em G, e determinar quando algum deles é um caminho de u para v. Um potencial caminho é uma sequência de vértices de G de tamanho menor ou igual a $|V|$ (o número de vértices de G). O número de potenciais caminhos é $|V|^{|V|}$, e portanto exponencial.

Neste capítulo estudaremos uma classe de problemas para os quais as técnicas estudadas até aqui não se aplicam. Estes problemas não têm soluções polinomiais conhecidas, mas também não existe uma prova de que tais soluções não existam: esta é a famosa questão "P versus NP"que constitui um dos mais interessantes problemas em aberto na Computação. Formalmente, definimos um problema como a seguir:

Definição 263. *Um problema (abstrato) Q é uma relação binária que associa um conjunto I de instâncias a um conjunto S de soluções.*

Por exemplo, o problema PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho que contém os dois vértices. O problema SHORTEST-PATH é uma relação que associa cada instância de um digrafo e dois vértices com um caminho mínimo que contém os dois vértices. Como caminhos mínimos não são necessariamente únicos, uma instância de um problema pode ter mais de uma solução. Neste capítulo trabalharemos essencialmente com *problemas de decisão*, que são problemas cujas respostas são sempre *sim* ou *não*:

Definição 264. *Um problema de decisão é uma relação binária sobre um conjunto I de instâncias e um conjunto binário (sim ou não) de soluções.*

Assim, podemos ver um problema de decisão como sendo uma função que associa instâncias em I ao conjunto de soluções $\{0, 1\}$. Por exemplo, o problema PATH é uma relação que associa, cada instância de um digrafo e dois vértices, com um caminho que contém os dois vértices. Este problema pode ser visto como um problema de decisão: Dados um digrafo G, e vértices u e v de G, determinar se existe um caminho de u para v em G.

Seja $\mathcal{C}$ uma classe de problemas caracterizados por uma propriedade, como por exemplo, possuir solução em tempo polinomial. Estamos interessados em identificar os problemas mais difíceis em $\mathcal{C}$, de tal forma que uma solução eficiente para algum destes problemas difíceis resulte em soluções eficientes para todos os outros problemas de $\mathcal{C}$. Na próxima seção estudaremos a classe P dos problemas que podem

ser resolvidos deterministicamente em tempo polinomial.

11.1 A classe P

A classe P consiste dos problemas que podem ser resolvidos em tempo polinomial por um algoritmo determinístico.

Teorema 265. *PATH está em P*

Demonstração. Considere o seguinte algoritmo que recebe como entrada um digrafo G e vértices u e v.

1. Marque o vértice u

2. Enquanto existir aresta $(a, b) \in G$ com a marcado, e b não-marcado, marque b.

3. Se v está marcado retorne 1, caso contrário retorne 0.

   Análise do algoritmo: O *laço* (linha 2) pode ser executado segundo o algoritmo de busca em largura, por exemplo, que é linear no tamanho da representação do grafo G. As outras linhas do algoritmo são executadas apenas uma vez, portanto o algoritmo é polinomial.

□

1. Reduções em tempo polinomial

   A ideia de *redutibilidade* entre dois problemas é que se um possui uma solução eficiente, então o outro também pode ser resolvido de forma eficiente. Adicionalmente, também podemos dizer que um problema não é mais fácil, e nem mais difícil, que outro também se aplica quando ambos são problemas de decisão. Por exemplo, considere um problema A que queremos resolver em tempo polinomial. Suponha que sabemos como resolver um outro problema B em tempo polinomial, e que temos um procedimento que transforma instâncias do problema A em instâncias do problema B com as seguintes características:

   - A transformação é feita em tempo polinomial;
   - As respostas são as mesmas para ambas as instâncias.

   Chamamos este procedimento de *algoritmo de redução*, que nos fornece uma forma de resolver o problema A em tempo polinomial:

   - Dada uma instância α do problema A, usamos o algoritmo de redução para transformá-la em uma instância β do problema B;
   - Executamos o algoritmo polinomial para a instância β de B;
   - Usamos a resposta de β como resposta de α.

Como cada um destes passos é realizado em tempo polinomial, todo o algoritmo também é realizado em tempo polinomial, e portanto temos uma forma de decidir A em tempo polinomial.

2. Linguagem formal

Um problema de decisão pode ser visto como um problema de reconhecimento de linguagem: seja U o conjunto de todas as entradas possíveis para o problema de decisão. Seja $L \subseteq U$, o conjunto de todas as entradas para as quais a resposta do problema é *sim*. Chamamos L a *linguagem* correspondente ao problema, e utilizamos os termos *problema* e *linguagem* de forma intercambiável. O problema de decisão deve então reconhecer se uma dada entrada pertence ou não à linguagem L.

Definição 266. *Seja Σ um conjunto finito (de símbolos) que chamaremos de alfabeto. O conjunto de todas as palavras (*strings*) que podem ser formadas com os símbolos de Σ é denotado por Σ^*. Uma linguagem L sobre Σ é qualquer subconjunto de Σ^*. O alfabeto vazio é denotado por ϵ, enquanto que a linguagem vazia é denotada por $\emptyset$.*

Diversas operações podem ser realizadas sobre linguagens: união, interseção, complemento, concatenação e fechamento. Do ponto de vista da teoria de linguagens formais, o conjunto das instâncias de qualquer problema de decisão Q é simplesmente o conjunto Σ^*, onde $\Sigma = \{0, 1\}$. Como o problema Q é caracterizado pelas instâncias que produzem resposta 1 (*sim*), podemos ver a linguagem L sobre $\Sigma = \{0, 1\}$ como sendo $L = \{x \in \Sigma^* \mid Q(x) = 1\}$.

Definição 267. *Dizemos que um algoritmo A aceita a palavra $x \in \{0, 1\}^*$ se $A(x) = 1$, i.e. a resposta do algoritmo A ao receber a entrada x é igual a 1. A linguagem aceita pelo algoritmo A é o conjunto das palavras aceitas pelo algoritmo: $L = \{x \in \{0, 1\}^* \mid A(x) = 1\}$. Dizemos que o algoritmo A rejeita a palavra x, se $A(x) = 0$.*

Note que, mesmo que a linguagem L seja aceita pelo algoritmo A, isto não significa que A rejeita uma palavra $x \notin L$ porque, por exemplo, A pode entrar em *loop*.

Definição 268. *Uma linguagem L é decidida pelo algoritmo A, se A aceita toda palavra em L, e rejeita toda palavra que não está em L. Uma linguagem L é aceita em tempo polinomial pelo algoritmo A, se A aceita L, e se existe uma constante k tal que para qualquer palavra $x \in L$ de comprimento n, o algoritmo A aceita x em tempo $O(n^k)$, para algum $k \geq 0$. Uma linguagem L é decidida em tempo polinomial pelo algoritmo A se existir uma constante não-negativa k tal que para qualquer palavra $x \in \{0, 1\}^*$, o algoritmo decide em tempo $O(n^k)$ se $x \in L$.*

Assim, para aceitar uma linguagem, o algoritmo precisa apenas produzir uma resposta para toda palavra de L, mas para decidir uma linguagem, o algoritmo precisa aceitar ou rejeitar toda palavra em $\{0, 1\}^*$. Como exemplo, o problema de decisão PATH corresponde a seguinte linguagem:

PATH $= \{\langle G, u, v, k\rangle : G = (V, E)$ é um grafo (não dirigido), $u, v \in V, k \geq 0$ é um inteiro, e existe um caminho de u para v em G contendo, no máximo, k arestas$\}$. Assim, a linguagem PATH pode ser aceita em tempo polinomial pelo seguinte algoritmo: Dada uma instância (G, u, v), o algoritmo executa BFS para computar o menor caminho de u para v, e então compara o número de arestas deste menor caminho com k. Se o menor caminho possui até k arestas então o algoritmo retorna 1, e para. Caso contrário, o algoritmo entra em *loop*. Note que este algoritmo não decide PATH. Neste caso, para decidir PATH basta que o algoritmo retorne 0 e pare, ao invés de entrar em *loop*. Para outros problemas, como o *problema da parada para máquinas de Turing*, existem algoritmos de aceitação, mas não de decisão.

Por fim, definimos a classe P em termos de linguagens:

$P = \{L \subseteq \{0,1\}^* : \text{ existe um algoritmo } A \text{ que decide } L \text{ em tempo polinomial}\}$

Definição 269. *Dizemos que uma linguagem L_1 é redutível polinomialmente à linguagem L_2, notação $L_1 \leq_p L_2$, se existe uma função computável em tempo polinomial $f : \{0,1\}^* \to \{0,1\}^*$ tal que para todo $x \in \{0,1\}^*$, $x \in L_1$ se, e somente se, $f(x) \in L_2$. A função f é chamada de* função de redução*, e o algoritmo polinomial F que computa a função f é chamado de* algoritmo de redução*.*

Observe que ao reduzirmos uma linguagem (ou problema) L_1 para outra linguagem (ou problema) L_2, queremos que cada instância de L_1 seja transformada em uma instância de L_2, isto é, se $x \in L_1$ então $f(x) \in L_2$. Adicionalmente, precisamos que elementos que não sejam instância de L_1 não sejam levados em L_2: $x \notin L_1$ então $f(x) \notin L_2$, o que é equivalente por contraposição a se $f(x) \in L_2$ então $x \in L_1$. Juntando estas duas informações temos a equivalência "$x \in L_1$ se, e somente se, $f(x) \in L_2$"da definição acima.

Reduções polinomiais são uma ferramenta poderosa que nos permitem provar que outras linguagens estão em P:

Lema 270. *Sejam $L_1, L_2 \in \{0,1\}^*$ linguagens tais que $L_1 \leq_p L_2$, então $L_2 \in P$ implica que $L_1 \in P$.*

Demonstração. Seja A_2 um algoritmo polinomial que decide a linguagem L_2, e F um algoritmo de redução polinomial que computa a função de redução f. Construiremos um algoritmo A_1 que decide L_1 da seguinte forma: dado $x \in \{0,1\}^*$, o algoritmo A_1 inicialmente usa F para transformar x em $f(x)$, e então usa o algoritmo A_2 para responder. Note que A_1 é polinomial já que tanto A_2 quanto F são polinomiais.

□

Considere o problema 2-SAT, cujas instâncias são expressões lógicas formadas por conjunções de disjunções de dois literais, onde um literal é uma variável booleana ou a negação de uma variável booleana. Por exemplo, a expressão a seguir é uma instância de 2-SAT:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee x_2)$$

Uma solução da instância acima é uma designação de valores booleanos (0 ou 1) para as variáveis que satisfaçam a expressão, ou seja, que retornem 1. Por exemplo, a designação $x_1 = 1$, $x_2 = 1$ e $x_3 = 0$ satisfaz a expressão acima.

Teorema 271. *2-SAT $\in P$.*

11.2 A classe NP

A classe NP consiste dos problemas que podem ser resolvidos em tempo polinomial por um algoritmo não-determinístico. A ideia é que inicialmente, o algoritmo advinhe uma solução (fase não-determinística), e em seguida esta solução deve ser verificada em tempo polinomial deterministicamente. Assim, a forma mais usual de apresentar a classe NP, consiste em considerar os problemas que podem ser verificados em

tempo polinomial por um algoritmo determinístico [11, 34].

Como vimos, em alguns casos é possível evitar uma abordagem força bruta e encontrar soluções polinomiais para o problema em questão. Mas é fácil imaginar que isto nem sempre será possível. De fato, veremos que existem diversos problemas interessantes/importantes para os quais soluções polinomiais não foram encontradas até hoje, mas que ainda é possível verificar em tempo polinomial, dado um certificado.

Exemplo 272. *O problema de encontrar ciclos Hamiltonianos em (di)grafos tem sido estudado por muito tempo (mais de 100 anos!). Formalmente, um ciclo Hamiltoniano de um grafo $G = (V, E)$ é um ciclo simples que contém cada vértice de V, i.e. cada vértice de G é visitado uma única vez. Um (di)grafo que contém um ciclo Hamiltoniano é dito Hamiltoniano.*

Denotaremos por HAM-CYCLE o problema de encontrar ciclos Hamiltonianos em (di)grafos.

HAM-CYCLE $= \{\langle G\rangle : G$ *é um (di)grafo Hamiltoniano* $\}$

Podemos verificar uma possível solução em tempo polinomial: Suponha que um colega te diz que um (di)grafo G é Hamiltoniano, e como justificativa, fornece uma sequência de vértices na ordem que ele diz formar um caminho Hamiltoniano.

1. *Verifique que os vértices dados constituem o conjunto V dos vértices de G;*

2. *Verifique que cada par de vértices consecutivos da sequência dada corresponde a uma aresta de G.*

Como a verificação acima pode ser feita em tempo polinomial, temos que HAM-CYCLE $\in$ NP.

1. Algoritmos de Verificação

   Definição 273. *Um algoritmo de verificação é um algoritmo A que recebe dois argumentos x e y, e verifica se $A(x, y) = 1$. Uma linguagem verificada por um algoritmo de verificação A é*

   $$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* \text{ tal que } A(x, y) = 1\}$$

   Intuitivamente, o algoritmo A verifica a linguagem L se, para cada $x \in L$, existe um certificado y tal que A é usado para provar que $x \in L$. Formalmente, a classe NP é a classe das linguagens que podem ser verificadas em tempo polinomial.

   $$NP = \{L \subseteq \{0,1\}^* : \text{ existe um algoritmo determinístico } A \text{ que verifica } L \text{ em tempo polinomial}\}$$

   Mais precisamente, $L \in$ NP se, e somente se, existe um algoritmo polinomial A e uma constante c tais que $L = \{x \in \{0,1\}^* : \exists y, |y| = O(|x|^c)$ tal que $A(x, y) = 1\}$.

   Agora é fácil ver que HAM-CYCLE $\in$ NP. Basta checar que o caminho dado (sequência de vértices) é uma permutação de V, isto é, cada vértice ocorre apenas uma vez, e que existe uma aresta em G para cada par de vértices consecutivos do caminho dado, e entre o primeiro e último vértices.

Estamos interessados em algoritmos que verificam se uma instância está ou não em uma linguagem. Por exemplo, dada uma instância (G, u, v) do problema de decisão PATH, e um caminho p de u para v, podemos facilmente verificar se p é um caminho em G.

A classe NP consiste dos problemas que podem ser verificados em tempo polinomial, i.e. dado um certificado, podemos verificar em tempo polinomial que este certificado é correto. Por exemplo, considerando o problema dos ciclos hamiltonianos em um (di)grafo $G = (V, E)$ com $n = |V|$, um certificado pode ser uma sequência $\langle v_1, v_2, \ldots, v_n \rangle$ de vértices que pode ser verificada em tempo polinomial. Para o problema 3-SAT, um certificado pode ser uma designação de valores para as variáveis da fórmula que pode ser verificado (se satisfaz ou não a fórmula dada) em tempo polinomial.

Exemplo 274. *Um clique em um grafo (não dirigido) é um subgrafo onde dois vértices quaisquer estão ligados por uma aresta. Um k-clique é um clique que contém k vértices. O problema CLIQUE consiste em determinar se um grafo contém um clique de um tamanho especificado:*

$CLIQUE = \{(G, k) : G \text{ é um grafo com um } k\text{-clique}\}$

***Afirmação**: $CLIQUE \in NP$*

O clique é o certificado. Para a entrada $((G, k), c)$

(a) Verifique se c é um subconjunto de $G.V$ de tamanho k;

(b) Verifique se G contém todas as arestas que conectam vértices em c;

(c) Se ambas as verificações podem ser feitas então retorne 1, caso contrário, retorne 0.

Teorema 275. *$3\text{-}SAT \leq_P CLIQUE$*

Demonstração. Nos grafos a serem construídos, cliques de um tamanho específico correspondem a designações satisfatíveis da fórmula. Seja φ uma fórmula com k cláusulas

$$\varphi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots (a_k \vee b_k \vee c_k)$$

A redução f constrói a codificação $\langle G, k \rangle$ onde G é um grafo não dirigido dado por:

- Os vértices de G são organizados em k grupos de 3 vértices cada $t_1, t_2, \ldots, t_k$. Cada tripla t_i corresponde a uma das cláusulas de φ, e cada vértice na tripla corresponde a um literal da cláusula associada. Marque cada vértice de G com o literal correspondente em φ. As arestas de G conectam todos os vértices exceto:
  - vértices contraditórios, como x e $\neg x$;
  - vértices da mesma tripla.

Afirmação: φ é satisfatível se, e somente se, G possui um k-clique.

Suponha que φ é satisfatível, e portanto cada cláusula possui pelo menos um literal verdadeiro. Em cada tripla em G, selecionamos um vértice correspondente ao literal verdadeiro. Se mais de um literal for verdadeiro na mesma cláusula, escolhemos um deles aleatoriamente. Os vértices selecionados formam um k-clique: o número de vértices selecionados é k, cada par de vértices selecionado está ligado por uma aresta.

Suponha que G possui um k-clique. Nenhum par de vértices do clique ocorre na mesma tripla porque vértices da mesma tripla não são ligados por arestas. Portanto, cada tripla contém exatamente um dos vértices do k-clique. Designamos valores para as variáveis de φ de forma que cada literal que marca um vértice assume valor 1 (verdadeiro). Isto é possível porque vértices contraditórios não são ligados. Esta designação de variáveis satisfaz a fórmula φ porque cada tripla corresponde a um vértice do clique, e portanto cada cláusula de φ tem valor 1.

□

11.3 A classe NPC

Dizemos que um problema é *NP-completo*, i.e. que está na classe NPC, se está na classe NP e é tão difícil quanto qualquer problema em NP. Ao mostrarmos que um problema é NP-completo, não estamos tentando provar a existência de um algoritmo eficiente, mas concluir que a existência de um tal algoritmo é improvável. De fato, estamos concluindo sobre o quão difícil ele é, e não sobre quão fácil como fizemos até então. Portanto, um algoritmo eficiente provavelmente não existe para este problema.

Uma importante questão em aberto é quando P é ou não um subconjunto próprio de NP, o que corresponde ao problema "P vs NP"citado anteriormente. Problemas NP-completos normalmente são considerados intratáveis dada a grande quantidade de problemas NP-completos já estudados, e sem solução polinomial encontrada.

Propriedade interessante: Se algum problema NP-completo puder ser resolvido em tempo polinomial então qualquer problema em NP terá solução polinomial.

Seria uma surpresa encontrar uma solução polinomial para algum (e portanto para todos) destes problemas. Neste sentido, se um problema é NP-completo então isto pode ser visto como uma evidência da sua intratabilidade. Neste caso, algoritmos aproximados devem ser considerados, ao invés de soluções rápidas e exatas.

Por que até hoje ninguém conseguiu encontrar soluções polinomiais para estes problemas? Não sabemos, talvez porque elas simplesmente não existam, ou porque elas estejam baseadas em princípios ainda desconhecidos. Qualquer problema em P está também em NP porque pode ser verificada em tempo polinomial pelo mesmo algoritmo que a decide em P sem a necessidade de utilizar certificados.

Definição 276. *Uma linguagem $L \subseteq \{0,1\}^*$ é dita **NP-completa** se:*

1. *$L \in NP$;*

2. *$L' \leq_P L, \forall L' \in NP$.*

Denotamos por NPC a classe das linguagens NP-completas.

Se uma linguagem L satisfaz a propriedade 2 acima, mas não necessariamente a propriedade 1, dizemos que L é **NP-difícil**.

Como NP-completude consiste em mostrar quão difícil é um problema, utilizamos a redução polinomial na outra direção para mostrar que um problema é NP-completo:

Para mostrarmos que um problema B não possui solução polinomial, consideremos um problema A, para o qual sabemos não existir solução polinomial. Suponha também que temos um algoritmo de redução que transforma instâncias de A em instâncias de B em tempo polinomial. Agora, se existisse uma solução polinomial para B então poderíamos construir uma solução polinomial para A como acima, contradizendo a suposição de que A não possui solução polinomial.

Lema 277. *Se L é uma linguagem tal que $L' \leq_P L$ para algum $L' \in$ NPC, então L é NP-difícil. Se adicionalmente, $L \in$ NP então $L \in$ NPC.*

Demonstração. Como L' é NP-completo, então $\forall L'' \in$ NP, temos que $L'' \leq_P L'$, e por hipótese temos que $L' \leq_P L$. Logo. $L'' \leq_P L$, o que mostra que L é uma linguagem NP-difícil. Agora, se $L \in$ NP então, por definição temos que $L \in$ NPC.

□

O lema anterior nos dá um método para mostrar que uma linguagem L é NP-completa:

1. Prove que $L \in$ NP;
2. Escolha uma linguagem L' que seja NP-completa;
3. Descreva um algoritmo que computa uma função f que mapeia cada instância x de L' em uma instância $f(x) \in L$;
4. Prove que $x \in L'$, se e somente se, $f(x) \in L$;
5. Prove que o algoritmo que computa f é polinomial

Os passos de 2-5 mostram que L é NP-difícil.

Exemplo 278. *$SAT = \{\langle\varphi\rangle : \varphi$ é uma fórmula booleana satisfatível$\}$.*

Podemos determinar em tempo exponencial se uma dada fórmula booleana φ contendo n variáveis é satisfatível: basta checar cada uma das 2^n possíveis valorações para as variáveis de φ. Nenhum algoritmo polinomial é conhecido para SAT. O teorema a seguir, mostra que é muito improvável que tal algoritmo exista.

O teorema a seguir é conhecido como o *teorema de Cook-Levin*:

Teorema 279. *$SAT \in$ NPC.*

Teorema 280. *3-$SAT \in$ NPC.*

Demonstração. 1. 3-SAT $\in$ NP: Dada uma designação de valores para as variáveis de uma 3-FNC fórmula (certificado), o algoritmo de verificação substitui cada variável pelo valor dado e avalia a expressão. Se a expressão resulta em 1 então o certificado é válido e a fórmula é satisfatível.

2. SAT $\leq_P$ 3-SAT.

□

Teorema 281. *CLIQUE $\in$ NPC.*

Demonstração. 1. CLIQUE $\in$ NP;

2. 3-SAT $\leq_P$ CLIQUE

□

Uma cobertura de vértices de um grafo $G = (V, E)$ (não-dirigido) é um subconjunto $V' \subseteq V$ tal que $(u, v) \in E$ então $u \in V'$ ou $v \in V'$ (ou ambos!). O tamanho de uma cobertura de vértices é o seu número de vértices, ou seja, $|V'|$. O problema da cobertura de vértices consiste em encontrar uma cobertura de vértices de tamanho mínimo em um grafo. Reescrevendo este problema como um problema de decisão, queremos determinar se um grafo possui uma cobertura de vértices de tamanho dado k:

VERTEX-COVER = $\{\langle G, k\rangle : G$ possui uma cobertura de vértices de tamanho $k\}$

Teorema 282. *VERTEX-COVER $\in$ NPC.*

Demonstração. 1. VERTEX-COVER $\in$ NP;

2. CLIQUE $\leq_P$ VERTEX-COVER.

□

Um caminho Hamiltoniano em um digrafo G é um caminho de um vértice u para um vértice v que visita todos os vértices de G exatamente uma vez. O problema de decisão HAMPATH consiste em, dado um digrafo G, responder se G possui um caminho Hamiltoniano ou não.

HAMPATH = $\{\langle G, u, v\rangle : G$ é um digrafo com um caminho Hamiltoniano de u para $v\}$

Teorema 283. *HAMPATH $\in$ NPC.*

Demonstração. 1. HAMPATH $\in$ NP;

2. 3-SAT $\leq_P$ HAMPATH

□

Índice Remissivo

Referências Bibliográficas

[1] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards Hybrid Theorem Proving Interfaces. *Electronic Proceedings in Theoretical Computer Science*, 239:15–27, January 2017.

[2] Jeremy Avigad, Kevin Donnelly, David Gray, and Paul Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 9(1):2–es, December 2007.

[3] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, April 2014.

[4] M. Ayala-Rincón and F. L. C. de Moura. *Applied Logic for Computer Scientists - Computational Deduction and Formal Proofs*. UTCS. Springer, 2017.

[5] S. Baase and A. V. Gelder. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[6] H. P. Barendregt. *The Lambda Calculus : Its Syntax and Semantics (Revised Edition)*. North Holland, 1984.

[7] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

[8] A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2017.

[9] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[10] A. Church. A Formulation of the Simple Theory of Types. *journal of Symbolic Logic*, 5:56–68, 1940.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, fourth edition, April 2022.

[13] F. S. C. da Silva, A. C. V. de Melo, and M. Finger. *Lógica Para Computação*. THOMSON PIONEIRA, 2006.

[14] Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, Lecture Notes in Computer Science, pages 625–635, Cham, 2021. Springer International Publishing.

[15] G. Gonthier. A computer-checked proof of the Four Colour Theorem. Technical report, Microsoft Research Cambridge, 2008.

[16] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics : A Foundation for Computer Science*. Addison-Wesley, 2004.

[17] T. Hales, M. Adams, G. Bauer, D. Tat Dang, J. Harrison, T. Le Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. Tat Nguyen, T. Quang Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. Hoai Thi Ta, T. N. Tran, D. Thi Trieu, J. Urban, K. Khac Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *ArXiv e-prints*, January 2015.

[18] J. Roger Hindley. *Basic Simple Type Theory.* Number 42 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997.

[19] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge University Press, New York, NY, USA, 2004.

[20] Cezary Kaliszyk. Web Interfaces for Proof Assistants. *Electronic Notes in Theoretical Computer Science*, 174(2):49–61, 2007.

[21] Cezary Kaliszyk, Stephan Schulz, Josef Urban, and Jiří Vyskočil. System Description: E.T. 0.1. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, volume 9195, pages 389–398. Springer International Publishing, Cham, 2015.

[22] Jon M. Kleinberg and Éva Tardos. *Algorithm Design.* Addison-Wesley, 2006.

[23] Xavier Leroy. Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107, 2009.

[24] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[25] Udi Manber. *Introduction to Algorithms: A Creative Approach.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1989.

[26] W. McCune. Prover9 and mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005.

[27] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lncs.* Springer, 2002.

[28] R. B. Nogueira, A. C. A. Nascimento, F. L. C. de Moura, and M. Ayala-Rincón. Formalization of Security Proofs Using PVS in the Dolev-Yao Model. In *Booklet Proc. Computability in Europe - CiE*, 2010.

[29] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *CADE*, volume 607 of *Lnai*, pages 748–752. sv, 1992.

[30] C. Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. 2014.

[31] Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *J Autom Reasoning*, 55(1):1–37, 2015.

[32] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Catvalin Hriatcu, Vilhelm Sjoberg, and Brent Yorgey. *Software Foundations.* Electronic textbook, 2014.

[33] Alexandre Riazanov and Andrei Voronkov. The design and implementation of VAMPIRE. *AI Commun.*, 15(2-3):91–110, 2002.

[34] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.

[35] Raymond Smullyan. *Logical Labyrinths.* AK Peters, 2009.

[36] Leon Sterling and Ehud Y Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT press, 1994.

[37] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

[38] D. van Dalen. *Logic and Structure (4. Ed.).* Universitext. Springer, 2008.

[39] D. van Dalen. *Logic and Structure.* Universitext. Springer London, 2013.