Projeto e Análise de Algoritmos

Flávio L. C. de Moura

28 de março de 2023

1 Introdução e motivação AULAS01E02

O objetivo deste curso é o estudo de técnicas de complexidade e projeto de algoritmos. No que tange à complexidade, estamos interessados no tempo demandado pelo algoritmo para processar uma determinada entrada (complexidade de tempo), assim como no espaço utilizado durante este processamento (complexidade de espaço). As técnicas de projeto de algoritmos estão relacionadas ao trabalho da construção de algoritmos a partir de um dado problema. Neste contexto, a técnica de *divisão e conquista* resolve um problema a partir da sua divisão em problemas menores seguida de uma combinação adequada das soluções dos subproblemas para assim construir uma solução do problema original. Outras técnicas que serão estudadas incluem a chamada *programação dinâmica* e os *algoritmos gulosos*. Uma vez construído um algoritmo, precisamos mostrar que ele funciona da forma esperada. Em outras palavras, antes de qualquer coisa precisamos mostrar que o algoritmo é correto, e só depois faremos a análise da sua complexidade. Intuitivamente, dizemos que um algoritmo A é correto quando sempre fornece respostas corretas para qualquer entrada possível. Por exemplo, se A é um algoritmo de ordenação de inteiros, então espera-se que para qualquer lista de inteiros l, A retorne uma permutação de l 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.

Uma ferramenta que será utilizada frequentemente nas provas de correção de algoritmos é a indução matemática. Ao analisarmos a eficiência (ou complexidade) dos algoritmos também faremos o uso de diversas ferramentas matemáticas (somatórios, conjuntos, funções, matrizes, etc). O apêndice VIII do livro [3, 4] pode ser usado para revisar estes temas.

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 os algoritmos? Inicialmente porque precisamos garantir que são corretos. 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. Mais ainda, é a análise da complexidade quem vai nos mostrar como podemos melhor aproveitar a capacidade de processamento

de um computador como veremos a seguir.

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.

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.

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[4, 6, 5, 1, 2]. As instruções no modelo RAM são as encontradas em computadores reais:

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

- 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 como o estudo da complexidade é fundamental para que possamos utilizar melhor os recursos computacionais disponíveis. Considere, por exemplo, duas máquinas A e B de forma que B seja 10 vezes mais rápida do que A. Um algoritmo linear vai processar uma entrada de tamanho n em tempo proporcional a n, digamos $c.n$, onde c é uma constante positiva. Na máquina B esta mesma entrada seria processada em tempo $\frac{c.n}{10}$, e portanto a máquina B é capaz de processar um problema 10 vezes maior no mesmo tempo que a máquina A! Aumentar em 10 vezes o tamanho do problema a ser processado é bastante significativo. No caso de um algoritmo exponencial, em que uma entrada de tamanho n seja processada em tempo proporcional a 2^n, o incremento do poder de processamento da máquina A para a máquina B praticamente não é sentido. De fato, uma instância de tamanho n seria processada em tempo $\frac{2^n}{10}$ na máquina B. O tamanho n' do problema que a máquina B pode processar no tempo 2^n é dado por $\frac{2^{n'}}{10} = 2^n \Rightarrow n' = n + \lg 10 = n + 3,3$. Como exercício, complete a tabela a seguir[2], onde um algoritmo hipotético que tem a complexidade de tempo dada na coluna da esquerda resolve, num dado tempo t, uma instância de tamanho máximo na máquina A dado na segunda coluna

Complexidade	tamanho máximo em A	tamanho máximo em B
$\lg n$	x_0	
n	x_1	$10.x_1$
n^2	x_3	
n^3	x_4	
2^n	x_5	$x_5 + 3,3$
3^n	x_6	

Alternativamente, podemos determinar o tamanho máximo de uma instância que pode ser resolvida por um algoritmo de uma dada complexidade conhecendo o poder computacional da máquina. Para cada função $f(n)$ e tempo t na tabela a seguir, determine o maior tamanho n de um problema que pode ser resolvido em tempo t, assumindo que o algoritmo resolve o problema em $f(n)$ microssegundos (adaptada de [3]).

$f(n)$	1 segundo	1 minuto	1 hora	1 dia	1 mês	1 ano	1 século
$\lg n$							
n	10^6						
n^2							
n^3							
2^n							
3^n							

[2]Adaptada de [8].

1.1 Algoritmos corretos

Após a construção de um programa é natural 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. No caso de uma resposta 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 construir provas matemáticas das propriedades esperadas do programa. Vejamos um exemplo: considere o pseudocódigo a seguir que recebe um vetor $A[0..n-1]$ contendo n números naturais e retorna o maior elemento deste vetor:

```
1 max ← A[0];
2 for i = 1 to n − 1 do
3 |   if max < A[i] then
4 |   |   max ← A[i];
5 |   end
6 end
7 return max;
```

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

Como mostrar que este algoritmo é correto? O primeiro passo é expressar o significado de correção neste caso por meio de um teorema, por exemplo. Em seguida construir uma prova.

Na aula, sugerimos provar a correção deste algoritmo por meio da seguinte invariante:

Antes da i-ésima iteração, a variável max contém o maior elemento do subvetor $A[0..i-1]$.

Construa uma prova para esta invariante de laço (Veja seção 2.1 de [3] para mais informações sobre invariantes de laço).

Agora vamos reescrever este algoritmo utilizando a estrutura de listas encadeadas, que possuem a seguinte gramática:

$$l ::= nil | h :: l$$

Adicionalmente, consideraremos duas funções sobre esta estrutura: A função hd, ou "head", retorna a cabeça da lista, ou seja, o primeiro elemento da lista, e a função tail, ou "cauda", retorna a lista sem o primeiro elemento. Utilizaremos a notação $hd.l$ (resp. $tail.l$) para denotar o primeiro elemento (resp. a cauda) da lista l.

```
1 max ← h;
2 l ← tl;
3 while l ≠ nil do
4 |   if max < hd.l then
5 |   |   max ← hd.l;
6 |   end
7 |   l ← tail.l;
8 end
9 return max;
```

Algorithm 2: MaxListElt($h :: tl$)

Prove que o algoritmo $MaxListElt$ é correto. Por fim, vejamos uma versão recursiva deste algoritmo, que agora recebe dois argumentos: uma lista, e um valor *default* que será retornado quando a lista for vazia. A ideia é armazenar o maior elemento no segundo argumento.

```
1 if l = h :: tl then
2 |   if max < h then
3 |   |   MaxListEltRec(tl, h)
4 |   else
5 |   |   MaxListEltRec(tl, max)
6 |   end
7 else
8 |   return max
9 end
```

Algorithm 3: MaxListEltRec(l, max)

Prove a correção deste algoritmo. Provas manuais, *i.e.* feitas em papel e lápis podem conter erros? Certamente, e uma forma de aumentar a confiança sobre uma prova consiste em fazê-la, por exemplo, em um assistente de provas. Durante a aula faremos uma prova mecânica da correção do algoritmo MaxListEltRec no assistente de provas Coq[7]. Segue o código Coq:

```
Fixpoint elt_max (l: list nat) (max:nat) :=
 match l with
 | nil => max
 | h::tl => if max <? h then (elt_max tl h) else (elt_max tl max)
 end.
```

A palavra reservada `Fixpoint` mostra que a função é recursiva. Observe a semelhança entre MaxListEltRec e `elt_max`. Na próxima aula, mostraremos a correção da função `elt_max` no Coq.

Referências

[1] 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.

[2] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., USA, 1996.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, fourth edition, April 2022.

[5] Jon M. Kleinberg and Éva Tardos. *Algorithm Design*. Addison-Wesley, 2006.

[6] A. V. Levitin. *Introduction to the Design and Analysis of Algorithms, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2012.

[7] The Coq Development Team. The Coq Proof Assistant. Zenodo, October 2021.

[8] Laira V. Toscani and Paulo A. S. Veloso. *Complexidade de Algoritmos: Volume 13 Da Série Livros Didáticos Informática UFRGS*, volume 13. Artmed Editora, 2012.