Projeto e Análise de Algoritmos

Flávio L. C. de Moura*

1 Introdução e Motivação

O objetivo deste curso é o estudo de técnicas para analisar e projetar de algoritmos. As técnicas utilizadas para analisar algoritmos, nos permitirão estudar o tempo e o espaço demandado pelo algoritmo para processar uma determinada entrada (complexidade de tempo e 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, a primeira preocupação que precisamos ter em relação a um algoritmo é se ele funciona da forma esperada. Em outras palavras, antes de qualquer coisa precisamos saber se 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 [2, 3] 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 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 aproveitar melhor a capacidade de processamento de um computador como veremos ao longo do curso.

2 A correção de algoritmos

Como primeiro exemplo, considere o problema de ordenar uma lista de inteiros. Existem diversos algoritmos que resolvem este problema, e uma abordagem simples consiste em inicialmente inserir elementos em uma lista ordenada. A função $insert\ x\ l$ definida a seguir, insere o elemento x na lista l:

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

*flaviomoura@unb.br

A inserção é feita de forma que se l está ordenada então a lista resultante também está ordenada. Podemos enunciar esta propriedade como um lema:

Lema 2.1. *Se l é uma lista ordenada e x é um inteiro, então $(insert\ x\ l)$ é uma lista ordenada.*

Exercício 2.2. *Prove o Lema 2.1.*

Na aula de *[2024-03-18 seg]*, iniciamos a prova do Lema 2.1 utilizando indução na estrutura da lista l. A base da indução corresponde ao caso em que a lista l é vazia. Neste caso, temos que provar que $insert\ x\ nil$ é uma lista ordenada, assumindo que a lista vazia está ordenada. Mas isto é trivial porque, pela definição de *insert*, $insert\ x\ nil$ é a lista unitária $x :: nil$ que está ordenada por definição.
O passo indutivo ocorre quando l tem a forma $h :: tl$. Neste caso, temos que provar que $insert\ x\ (h :: tl)$ é uma lista ordenada, assumindo que a lista $h :: tl$ está ordenada. A definição de *insert* nos dá dois subcasos:

1. $x \leq h$: Neste caso, $insert\ x(h :: tl)$ retorna $x :: h :: tl$ que está ordenada porque o elemento inserido x é menor ou igual ao primeiro elemento de uma lista ordenada.
2. $x > h$: Na aula de *[2024-03-20 qua]*, iniciamos com a escrita da hipótese de indução:

   > Se tl é uma lista ordenada e x é um inteiro, então $(insert\ x\ tl)$ é uma lista ordenada.

Temos como hipótese (do problema) que $h :: tl$ é uma lista ordenada, e queremos mostrar que a lista $insert\ x\ (h :: tl)$ é uma lista ordenada. Pela definição da função *insert*, temos que $insert\ x\ (h :: tl) = h :: (insert\ x\ tl)$. Agora observe que, por hipótese de indução, a lista $insert\ x\ tl$ está ordenada. Além disto, temos que $h < x$ e h é menor ou igual a todo elemento da lista tl já que $h :: tl$ está ordenada. Então h é menor ou igual a todo elemento de $(insert\ x\ tl)$, e portanto a lista $h :: (insert\ x\ tl)$ está ordenada. □

Agora vamos refazer esta prova mecanicamente utilizando o assistente de provas Coq. Primeiramente, vamos definir o predicado que caracteriza uma lista ordenada.

```
Inductive ordenada : list nat -> Prop :=
  | nil_ord : ordenada nil
  | one_ord : forall n:nat, ordenada (n::nil)
  | cons_ord : forall (n m:nat) (l:list nat), n <= m ->
               ordenada (m::l) -> ordenada (n::m::l).
```

A função *insert* pode ser definida em Coq como segue:

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

Agora podemos enunciar e provar o lema 2.1 em Coq:

```
Lemma insert_ord : forall (x:nat) (l:list nat), ordenada l -> ordenada (insert x l).
```

Exercício 2.3. *Prove o lema* `insert_ord` *no Coq.*

A resolução do exercício anterior nos dá uma boa ideia do que consiste o processo de formalização (ou mecanização) de um algoritmo, ainda que bastante simples. A função principal do algoritmo de ordenação por inserção é dada a seguir:

$$insertion\_sort\ l := \begin{cases} l, & \text{se } l = nil \\ insert\ x\ (insertion\_sort\ tl), & \text{se } l = h :: tl \end{cases}$$

Exercício 2.4. *O algoritmo* `insertion_sort` *é correto? Como você responderia a esta pergunta com as devidas justificativas em papel e lápis? E no Coq?*

Tente resolver os exercícios apresentados, e traga as suas dúvidas para discutirmos na próxima aula. O arquivo Coq está disponível em http://flaviomoura.info/files/paa_2024_1_introducao.v

Exercício 2.5. *Considere a seguinte função:*

$$f\ n := \begin{cases} 0, & \text{se } n = 0 \\ 2.n - 1 + f\ (n-1), & \text{se } n > 0 \end{cases}$$

Mostre que a função f computa o quadrado do seu argumento, i.e. $f\ n = n^2$.

Exercício 2.6. *Considere a seguinte função:*

$$seq\_search\ x\ l := \begin{cases} FALSE, & \text{se } l = nil; \\ TRUE, & \text{se } l = h :: l' \text{ e } h = x; \\ seq\_search\ x\ l', & \text{se } l = h :: l' \text{ e } h \neq x; \end{cases}$$

Prove que se x for um elemento de l então seq_search x l retorna TRUE, e retorna FALSE, quando x não ocorre em l.

Exercício 2.7. *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:*

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

```
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
```

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.

Prove as seguintes propriedades:

1. *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.*

2. *Seja $A[1..n]$ um vetor ordenado de inteiros distintos. Mostre que se a chave key ocorre em $A[1..n]$ na posição $1 \leq j \leq n$, então BinarySearch($A[1..n], 1, n,$ key) retorna o valor j.*

2.1 A correção de algoritmos não-recursivos

Como primeiro exemplo, considere o problema de buscar um elemento em um vetor de números naturais. O pseudocódigo a seguir recebe como argumentos o vetor $A[0..n-1]$ contendo n números naturais e o natural x procurado, e faz a busca 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.

Algorithm 2: SequentialSearch($A[0..n-1], x$)

```
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
```

Parece bastante intuitivo dizer que este algoritmo é correto, mas como **provar** isto? Inicialmente temos que expressar a noção de correção como uma propriedade, e em seguida, precisamos provar esta propriedade. Em programas contendo laços utilizamos as *invariantes de laço*, que são propriedades satisfeitas durante toda a execução de um laço. A prova de uma invariante de laço consiste em três etapas, e é análoga a uma prova indutiva:

1. **Inicialização**: Nesta etapa mostramos que a propriedade é satisfeita antes da primeira execução do laço. Esta etapa é equivalente à base da indução em uma prova indutiva;
2. **Manutenção**: Esta é a etapa mais delicada da prova porque corresponde ao passo indutivo. Aqui assumimos por hipótese (de indução) que a invariante vale antes de uma iteração arbitrária do laço (depois da primeira) e mostramos que a invariante continua válida antes da próxima iteração. Ou seja, assumimos que a invariante vale antes da k-ésima iteração ($k > 0$) e mostramos que ao final desta iteração, isto é, antes da $(k+1)$-ésima iteração, a invariante continua satisfeita;
3. **Finalização**: Nesta etapa concluímos que a invariante é satisfeita durante toda a execução do laço, e esta informação é utilizada para estabelecer a correção do algoritmo.

Com estas informações, construiremos uma invariante para o laço **while** (linhas 2-4) que nos permita concluir a correção ao final da execução do algoritmo. A construção da invariante costuma ser a etapa mais difícil porque precisamos pensar em uma propriedade que seja verdadeira antes da execução do laço (inicialização), permaneça verdadeira durante toda a execução do laço (manutenção) e tal que sua validade ao final da execução do laço nos permita concluir sua correção (finalização).

No caso da busca sequencial 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:

Proof. 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 consideraremos novamente o algoritmo de ordenação por inserção, mas sua versão não-recursiva. Sabemos que o algoritmo *insertion sort* (ordenação por inserção)e tem como etapa principal inserir um elemento em um vetor ordenado. Assim, para ordenarmos $n > 0$ números naturais em ordem crescente vamos supor que estes números estejam armazenados no vetor $A[0..n-1]$. 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 $1 \leq i < n$. O algoritmo de ordenação por inserção (*insertion sort*) é dado pelo pseudocódigo a seguir([2]):

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

```
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
```

A prova da correção de algoritmos iterativos que possuem laços utilizam as *invariantes de laço*, *i.e.* uma propriedade que é preservada durante a execução do laço. 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**:

1. **Inicialização**: Antes da primeira iteração do **while** temos que $i + 1 = j = k$, e como $key = A[j]$ a invariante está satisfeita.
2. **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.
3. **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 2.8. *Prove que o algoritmo BubbleSort a seguir é correto.*

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

```
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
```

Exercício 2.9. *Prove que o algoritmo BubbleSort2 [3] a seguir é correto.*

Algorithm 5: BubbleSort2($A[0..n - 1]$)

```
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
```

Exercício 2.10. *Prove que o algoritmo SelectionSort a seguir é correto.*

Algorithm 6: SelectionSort($A[0..n - 1]$)

```
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
```

3 Leitura complementar:

- [2] (Capítulos 1 e 2)

- [3] (Capítulos 1 e 2)
- [4] (Capítulos 1 e 3)
- [1] (Capítulos 1 e 2)

Referências

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

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

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

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