

Apostila Introdutória de Algoritmos

Guilherme D. da Fonseca
Celina M. H. de Figueiredo

Versão rascunho para o curso de Análise de Algoritmos da Unirio
2009.

18 de Setembro de 2009

CAPÍTULO 1

Introdução

Segundo o dicionário Aurélio, um algoritmo é um “*processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, regras formais para obtenção do resultado ou da solução do problema*”. Embora os algoritmos não sejam necessariamente executados por computadores, este é o tipo de algoritmo que trataremos neste livro. O propósito deste livro é que o leitor não só conheça e entenda diversos algoritmos para problemas variados, como também que seja capaz de desenvolver por si próprio algoritmos eficientes.

As sessões deste livro, em sua maioria, explicam cinco itens:

- Problema: a explicação de que problema está sendo resolvido na sessão.
- Algoritmo: o método computacional para a resolução do problema.
- Prova de corretude: a argumentação de que o algoritmo apresentado resolve corretamente o problema.
- Complexidade: o tempo que o algoritmo leva para resolver o problema.
- Análise de complexidade: o cálculo deste tempo.

Não necessariamente os itens são explicados nesta ordem, ou de modo completamente separado. Muitas vezes, a prova de corretude é apresentada junto com a explicação do algoritmo, justificando o modo como ele é desenvolvido e facilitando seu entendimento.

Nesta introdução, falamos destes cinco itens, fornecendo a base necessária para o entendimento dos demais capítulos do livro.

1.1. Os Problemas

Problemas precisam ser resolvidos constantemente, em todas as áreas do conhecimento humano. Muitos problemas, principalmente de áreas sociais, humanas ou artísticas, não podem ser resolvidos por um computador. Porém, a maioria dos problemas das áreas chamadas de ciências exatas podem ser resolvidos de modo mais eficaz com o auxílio dos computadores. Este livro visa fornecer conhecimentos necessários para programar um computador de modo a resolver problemas não triviais eficientemente. Antes disso, devemos formalizar o que é um problema.

Todo o problema tem uma entrada, também chamada de instância. Nos problemas que estudamos, existem infinitas entradas possíveis. A entrada pode ser bastante simples como no problema cuja entrada é um número inteiro e desejamos descobrir se ele é primo. Em outros problemas, a entrada pode ser bastante complexa, tendo vários elementos relacionados, como grafos, vértices especiais dos grafos, particionamentos dos vértices etc.

Além da entrada, todo problema tem uma saída correspondente, que é a resposta do problema. Os algoritmos devem ser capazes de manipular a entrada para obter a saída.

O tipo de problema mais elementar é o chamado problema de decisão. Neste tipo de problema, formula-se uma pergunta cuja resposta é sim ou não. Vejamos alguns exemplos de problemas de decisão:

- Dado um número inteiro, dizer se este número é primo.
- Dado um conjunto, dizer se um elemento x pertence a este conjunto.
- Dado um conjunto de segmentos no plano, dizer se dois segmentos se interceptam.
- Dado um grafo, dizer se o grafo possui ciclos.

Embora a resposta para um problema de decisão seja sim ou não, é natural formular a chamada versão de construção de alguns desses problemas. Em um problema de construção,

não se deseja apenas saber se uma estrutura existe ou não, mas construir a estrutura que satisfaça algumas propriedades. As versões de construção dos dois últimos problemas de decisão apresentados é:

- Dado um conjunto de segmentos no plano, encontrar dois segmentos que se interceptam, se existirem.
- Dado um grafo, exibir um ciclo deste grafo, se existir.

Em outros problemas de construção, não há uma versão de decisão relacionada. Nos exemplos abaixo, não há dúvida que a estrutura exista, a única dificuldade é exibi-la:

- Dados dois números inteiros, calcular seu produto.
- Dado um conjunto de números reais, ordenar seus elementos.
- Dado um conjunto de pontos não colineares no plano, encontrar 3 pontos que formem um triângulo sem nenhum outro ponto em seu interior.
- Dada uma árvore, encontrar seu centro.

Um tipo especial de problema de construção é chamado de problema de otimização. Nestes problemas, não queremos construir uma solução qualquer, mas sim aquela que maximize ou minimize algum parâmetro. Vejamos alguns exemplos:

- Dados dois números inteiros, calcular seu maior divisor comum.
- Dado um conjunto de números reais, encontrar o menor.
- Dado um conjunto de pontos não colineares no plano, encontrar os 3 pontos que formem um triângulo sem nenhum outro ponto em seu interior que tenha perímetro mínimo.
- Dado um grafo, encontrar sua árvore geradora mínima.

A diferença entre esses problemas e os problemas de construção é sutil, e nem sempre precisamente definida. Por exemplo, o problema de construção onde se deseja encontrar o centro de uma árvore é um problema de otimização, pois o centro de uma árvore é o conjunto dos vértices cuja distância ao vértice mais distante é mínima. Ainda assim, é útil diferenciar estes tipos básicos de problemas, pois algumas técnicas que apresentaremos, se mostram especialmente eficientes para determinado tipo de problema.

Existem outros tipos de problemas que não resolveremos neste livro. Os problemas de enumeração são um exemplo. Nestes problemas deseja-se listar todas as estruturas que satisfazem uma propriedade. Associado a todo o problema de enumeração, existe um problema de contagem. No problema de contagem, não se está interessado em listar todas as soluções, mas apenas descobrir quantas soluções distintas existem. Alguns exemplos destes dois tipos de problema são:

- Dado um número inteiro, listar todos os seus fatores (primos ou não).
- Dado um conjunto, contar o número de sub-conjuntos com determinado número de elementos.
- Dado um conjunto de segmentos no plano, calcular o número de interseções entre os segmentos.
- Dado um grafo, exibir todos os seus ciclos.

1.2. Algoritmos e Paradigmas

Um algoritmo é uma maneira sistemática de resolver um problema. Algoritmos podem ser usados diretamente por seres humanos para diversas tarefas. Ao fazer uma conta de dividir sem usar calculadora, por exemplo, estamos executando um algoritmo. Porém, os algoritmos ganharam importância muito maior com os computadores. Vários problemas cuja solução era praticamente inviável sem um computador passaram a poder ser resolvidos em poucos segundos. Mas tudo depende de um bom algoritmo para resolver o problema.

Ao recebermos um problema, como fazemos para desenvolver um bom algoritmo para resolvê-lo? Não há resposta simples para esta pergunta. Todo este livro visa preparar o leitor para este desenvolvimento. Sem dúvida, conhecer bons algoritmos para muitos problemas ajuda bastante no desenvolvimento de novos algoritmos. Por isso, praticamente todos os livros sobre o assunto

apresentam vários problemas, junto com suas soluções algorítmicas. Geralmente, os problemas são organizados de acordo com a área do conhecimento a que pertencem (teoria dos grafos, geometria computacional, seqüências, álgebra...). Neste livro fazemos diferente.

Embora não exista uma receita de bolo para projetar um algoritmo, existem algumas técnicas que freqüentemente conduzem a “bons” algoritmos. Este livro está organizado segundo estas técnicas, chamadas de paradigmas. Vejamos, de modo simplificado, dois exemplos de paradigmas: “construção incremental” e “divisão e conquista”.

- Construção incremental: Resolve-se o problema para uma entrada com apenas um elemento. A partir daí, acrescenta-se, um a um, novos elementos e atualiza-se a solução.
- Divisão e conquista: Quando a entrada tem apenas um elemento, resolve-se o problema diretamente. Quando é maior, divide-se a entrada em duas entradas de aproximadamente o mesmo tamanho, chamadas sub-problemas. Em seguida, resolvem-se os dois sub-problemas usando o mesmo método e combinam-se as duas soluções em uma solução para o problema maior.

Vamos exemplificar estes dois paradigmas no problema de ordenação:

PROBLEMA 1. *Dado um conjunto de números reais, ordene o conjunto do menor para o maior elemento.*

Neste problema, a entrada consiste de um conjunto de números reais e a saída é uma lista desses números, ordenada do menor para o maior. Nos dois paradigmas, precisamos saber resolver o caso em que a entrada possui apenas um elemento. Isto é fácil. Neste caso, a lista ordenada contém apenas o próprio elemento.

No paradigma de construção incremental, precisamos descobrir como acrescentar um novo elemento x em uma lista já ordenada. Para isto, podemos percorrer os elementos a partir do menor até encontrar um elemento que seja maior que x . Então, deslocamos todos os elementos maiores que x de uma posição, e colocamos o elemento x na posição que foi liberada. Este algoritmo é chamado de ordenação por inserção.

No paradigma de divisão e conquista, precisamos descobrir como combinar duas listas ordenadas L_1 e L_2 em uma única lista L . Podemos começar comparando o menor elemento de L_1 com o menor elemento de L_2 . O menor elemento dentre esses dois é certamente o menor elemento de L . Colocamos então este elemento na lista L e removemos o elemento de sua lista de origem, L_1 ou L_2 . Seguimos sempre comparando apenas o menor elemento de L_1 com o menor elemento de L_2 e colocando o menor elemento dentre esses dois no final da lista L , até que uma das listas L_1 ou L_2 se torne vazia. Quando uma das listas se tornar vazia, a outra lista é copiada integralmente para o final da lista L . Este algoritmo é chamado de *mergesort*.

Às vezes, explicar um algoritmo em parágrafos de texto pode ser confuso. Por isto, normalmente apresentamos também o chamado pseudo-código do algoritmo. Este pseudo-código é uma maneira estruturada de descrever o algoritmo e, de certa forma, se parece com sua implementação em uma linguagem de programação. O pseudo-código do algoritmo de ordenação por inserção está na figura 1.1. Há várias maneiras de escrever o pseudo-código para um mesmo algoritmo. Vejamos dois pseudo códigos diferentes para o algoritmo de divisão e conquista que acabamos de apresentar, escritos nas figuras 1.2 e 1.3.

O primeiro pseudo-código (figura 1.2) é mais curto e muito mais fácil de entender que o segundo (figura 1.3). Por outro lado, o segundo pseudo-código se parece mais com uma implementação real do algoritmo. Mas note que, mesmo o segundo pseudo-código ainda é bastante diferente de uma implementação real. Afinal, não nos preocupamos em definir os tipos de variáveis ou fazer as alocações de memória. Neste livro, quase sempre optaremos por um pseudo-código no estilo do primeiro, pois consideramos o entendimento do algoritmo mais importante que um pseudo-código “pronto para implementar”. Embora a implementação do primeiro pseudo-código não seja imediata, qualquer bom programador deve ser capaz de compreendê-lo e implementá-lo em um tempo relativamente pequeno.

Entrada:

S : Conjunto de números reais a serem ordenados armazenado em um vetor.

Saída:

L : Conjunto S ordenado do menor para o maior.

Ordenar(S)

```

  Para  $i$  de 1 até  $|S|$ 
     $x \leftarrow S[i]$ 
     $j \leftarrow 1$ 
    Enquanto  $j < i$  e  $L[j] < x$ 
       $j \leftarrow j + 1$ 
    Para  $j$  de  $j$  até  $i$ 
      Troque valores de  $L[j]$  e  $x$ 
  Retorne  $L$ 

```

FIGURA 1.1. Pseudo-código do algoritmo de ordenação por inserção.

Entrada:

S : Conjunto de números reais a serem ordenados armazenado em um vetor.

Saída:

L : Conjunto S ordenado do menor para o maior.

Ordenar(S)

```

  Se  $|S| = 1$ 
    Retorne  $S[1]$ 
  Divida  $S$  em  $S_1$  e  $S_2$  aproximadamente de mesmo tamanho
   $L_1 \leftarrow$  Ordenar( $S_1$ )
   $L_2 \leftarrow$  Ordenar( $S_2$ )
  Enquanto  $|L_1| \neq 0$  e  $|L_2| \neq 0$ 
    Se  $L_1[1] \leq L_2[1]$ 
      Coloque  $L_1[1]$  no final da lista  $L$ 
      Remova  $L_1[1]$  de  $L_1$ 
    Senão
      Coloque  $L_2[1]$  no final da lista  $L$ 
      Remova  $L_2[1]$  de  $L_2$ 
  Se  $|L_1| \neq 0$ 
    Coloque elementos de  $L_1$  no final de  $L$ , na mesma ordem
  Senão
    Coloque elementos de  $L_2$  no final de  $L$ , na mesma ordem
  Retorne  $L$ 

```

FIGURA 1.2. Primeiro pseudo-código do algoritmo *mergesort*.

1.3. Provas de Corretude

Em alguns algoritmos, como os algoritmos de ordenação que acabamos de ver, é bastante claro que o algoritmo resolve corretamente o problema. Porém, em muitos outros, não é tão óbvio que a resposta encontrada realmente está correta. De fato, a diferença entre um algoritmo que funciona corretamente e outro que fornece respostas erradas pode ser bastante sutil. Por isso, é essencial provarmos que o algoritmo funciona corretamente, ou seja, faz aquilo que se propõe a fazer.

Entrada:

S : Conjunto de números reais a serem ordenados armazenado em um vetor.

n : Tamanho de S .

Saída:

L : Conjunto S ordenado do menor para o maior.

Ordenar(S, n)

 Se $n = 1$

 Retorne S

 Para i de 1 até $\lfloor n/2 \rfloor$

$S_1[i] \leftarrow S[i]$

 Para i de $\lfloor n/2 \rfloor + 1$ até n

$S_2[i - \lfloor n/2 \rfloor] \leftarrow S[i]$

$L_1 \leftarrow$ Ordenar($S_1, \lfloor n/2 \rfloor$)

$L_2 \leftarrow$ Ordenar($S_2, \lceil n/2 \rceil$)

$i \leftarrow i_1 \leftarrow i_2 \leftarrow 1$

 Enquanto $i_1 \leq \lfloor n/2 \rfloor$ e $i_2 \leq \lceil n/2 \rceil$

 Se $L_1[i_1] \leq L_2[i_2]$

$L[i] \leftarrow L_1[i_1]$

$i_1 \leftarrow i_1 + 1$

 Senão

$L[i] \leftarrow L_2[i_2]$

$i_2 \leftarrow i_2 + 1$

$i \leftarrow i + 1$

 Se $i_1 \neq \lfloor n/2 \rfloor$

 Para i de i até n

$L[i] \leftarrow L_1[i_1]$

$i_1 \leftarrow i_1 + 1$

 Senão

 Para i de i até n

$L[i] \leftarrow L_2[i_2]$

$i_2 \leftarrow i_2 + 1$

 Retorne L

FIGURA 1.3. Segundo pseudo-código do algoritmo *mergesort*.

Um exemplo que demonstra como a diferença entre um algoritmo funcionar e não funcionar pode ser sutil é o problema do troco. Neste problema, deseja-se formar uma quantia x em dinheiro, usando o mínimo de moedas possível. Provar que um algoritmo para este problema está correto significa provar que a quantia fornecida pelo algoritmo é x e que o número de moedas usado é realmente mínimo.

O nosso algoritmo procede da seguinte maneira. Para formarmos a quantia x , pegamos a moeda de valor m máximo dentre as moedas com valores menores ou iguais a x . Esta moeda de valor m é fornecida como parte do troco. Para determinar o restante do troco, subtraímos m de x , e procedemos da mesma maneira.

Vamos examinar este mesmo algoritmo com dois conjuntos diferentes de valores de moedas disponíveis. Estes conjuntos não são considerados parte da entrada do problema, mas sim parte de sua definição. A entrada do problema consiste do valor que desejamos fornecer como troco. Vamos supor, para simplificar nossa argumentação, que existam quantidades ilimitadas de moedas de cada valor disponível.

Digamos que temos moedas com os valores 1, 10, 25 e 50 centavos, e desejamos fornecer um troco no valor de 30 centavos. O nosso algoritmo, fornecerá primeiro uma moeda de 25 centavos e, em seguida, 5 moedas de 1 centavo, totalizando 6 moedas. Claramente, podemos formar esta quantia, com apenas 3 moedas de 10 centavos. Portanto, o algoritmo não está correto para este problema.

Vamos considerar agora outro problema, em que temos apenas moedas de 1, 5, 10 e 50 centavos. Neste caso o algoritmo funciona? Sim. Vejamos a prova:

TEOREMA 1.1. *O algoritmo apresentado acima funciona corretamente.*

DEMONSTRAÇÃO. Claramente a quantia fornecida pelo algoritmo soma x . Precisamos provar que o número de moedas é mínimo. O algoritmo fornece as moedas do troco em ordem, da maior para a menor. Seja $S = (m_1, m_2, \dots, m_n)$ a seqüência de valores das moedas fornecidas pelo algoritmo. Suponha, para obter um absurdo, que $S' = (m'_1, m'_2, \dots, m'_{n'})$, com $n' < n$, seja uma seqüência de valores de moedas que some x , ordenada do maior para o menor, que use o mínimo possível de moedas. Seja i o menor valor tal que $m_i \neq m'_i$. Certamente, $m_i > m'_i$, pois m_i , a moeda escolhida pelo algoritmo, é a maior moeda que não excederia a quantia x . Como as seqüências estão ordenadas, vale que $m_i > m'_j$, para j de i até n' . Também é claramente verdade que a soma das moedas de m'_i até $m'_{n'}$ vale pelo menos m_i . Unindo estas informações ao fato de que todas as moedas disponíveis (1, 5, 10 e 50 centavos) são múltiplas das moedas menores, então há um subconjunto não unitário das moedas de m'_i até $m'_{n'}$ que soma exatamente m_i . É possível melhorar a solução S' , substituindo este subconjunto por uma moeda de valor m_i , o que contradiz a otimalidade de S' . \square

Neste caso, foi possível provar que o algoritmo está correto, porque o valor de toda moeda é um múltiplo dos valores das moedas menores. Isto não acontecia antes, porque a moeda de 25 centavos não é múltipla da moeda de 10 centavos.

Caso tenhamos moedas de 1, 5, 10, 25 e 50 centavos, o algoritmo funciona? Não vale a propriedade que toda moeda é múltipla das menores, porém, ainda assim, o algoritmo funciona corretamente. A condição de toda a moeda ser múltipla das menores é suficiente para o algoritmo funcionar, mas não é necessária. A prova que o algoritmo funciona corretamente neste último caso é mais trabalhosa e fica como exercício.

1.4. Complexidade de Tempo

Como podemos calcular o tempo gasto por um algoritmo resolvendo um determinado problema? Este tempo depende de diversos fatores, como a entrada do problema, a máquina que está executando o programa e de como foi feita a implementação do algoritmo. Por isso, determinar exatamente o tempo gasto por um algoritmo é um processo intrinsecamente experimental. Implementa-se o algoritmo, define-se uma entrada ou conjunto de entradas e executa-se o algoritmo para estas entradas em uma máquina específica, medindo os tempos. Esta abordagem experimental tem vantagens e desvantagens com relação a abordagem teórica que estudamos neste livro. Vamos apresentar primeiro alguns pontos fracos da abordagem experimental.

- Dependência da entrada: O tempo gasto por um algoritmo pode ser extremamente dependente de alguns detalhes sutis da entrada. Há, por exemplo, algoritmos de ordenação bastante eficientes quando a entrada está bem embaralhada, mas que são muito lentos quando a entrada já está quase completamente ordenada. Por outro lado, há algoritmos que são muito rápidos quando a entrada já está quase completamente ordenada, mas que são extremamente ineficientes na maioria dos casos. Muitas vezes, é difícil saber se as entradas escolhidas para o experimento representam bem as entradas com que o algoritmo será de fato usado.
- Dependência da máquina: Este caso é bem menos crítico que o anterior. De um modo geral, se um algoritmo a foi mais rápido que um algoritmo b em uma determinada máquina, o algoritmo a também será mais rápido que o algoritmo b em qualquer outra máquina. Mas há exceções. Por exemplo, uma máquina com operações de ponto

flutuante extremamente rápidas pode se beneficiar de algoritmos que usem fortemente ponto flutuante, enquanto outra máquina pode se beneficiar de algoritmos que façam menos operações de ponto flutuante. Em máquinas com um *cache* de memória pequeno, um algoritmo que acesse os dados com maior localidade pode ser preferível, enquanto em máquinas com um *cache* maior, ou sem nenhum *cache*, outro algoritmo pode ser preferível.

- Dependência da implementação: Digamos que você crie um algoritmo a e resolva escrever um artigo argumentando que seu algoritmo é mais rápido que o algoritmo b . Como criador do algoritmo a , você provavelmente conhece muito bem este algoritmo e é capaz de implementá-lo de modo extremamente eficiente. A sua implementação do algoritmo a será provavelmente muito melhor que a sua implementação do algoritmo b . Deste modo, a comparação é bastante injusta.
- Incomparabilidade: Digamos que alguém apresente o tempo que uma implementação de um determinado algoritmo levou em uma determinada máquina com uma entrada específica e outra pessoa apresente o tempo que outro algoritmo para o mesmo problema levou com outra entrada em outra máquina. É completamente impossível comparar estes dois resultados para determinar qual algoritmo será mais rápido no seu caso.
- Alto custo: Devido a impossibilidade de comparar execuções dos algoritmos com entradas diferentes ou em máquinas diferentes, é necessário implementar e testar diversos algoritmos para determinar qual é mais rápido no seu caso específico. O tempo e o custo dessas tarefas podem ser bastante elevados.

A seguir, vamos introduzir a complexidade de tempo assintótica de pior caso, que usamos para avaliar a eficiência dos algoritmos. Esta análise tem se mostrado extremamente útil por fornecer uma expressão simples que permite comparar facilmente dois algoritmos diferentes para o mesmo problema, independente da máquina, implementação ou da entrada.

1.5. Complexidade de Tempo de Pior Caso

Primeiro vamos explicar como fazemos a análise independe da entrada. Para isto, consideramos sempre a pior entrada possível, ou seja, a que leva mais tempo para ser processada. Como estamos lidando com entradas ilimitadamente grandes, precisamos fixar o tamanho da entrada, ou alguma outra propriedade dela. Por enquanto, não vamos considerar a dependência da máquina ou da implementação. Vamos considerar que estamos falando sempre de uma máquina previamente definida e de uma implementação específica.

Podemos falar, no problema de ordenação, da lista de n elementos que leva mais tempo para ser ordenada por um determinado algoritmo (com relação a todas as listas com n elementos). No problema de, dado um conjunto de n pontos no plano, determinar o par de pontos mais próximos, podemos expressar a complexidade de tempo em função do número n de pontos da entrada. No problema de, dado um conjunto de polígonos, dizer se dois polígonos se interceptam, *não* é razoável expressar a complexidade de tempo em função do número de polígonos da entrada. Afinal, um polígono pode ter qualquer número de vértices. Uma entrada com apenas 2 polígonos pode ser extremamente complexa se estes polígonos tiverem muitos vértices. Já uma entrada com vários triângulos pode ser bem mais simples. Por isso, neste problema, é razoável expressar a complexidade de tempo em função do número total de vértices dos polígonos.

Em todos estes casos, queremos definir uma função $T(n)$ que representa o tempo máximo que o algoritmo pode levar em uma entrada com n elementos. Às vezes, podemos expressar o tempo em função de vários parâmetros da entrada, simultaneamente. Quando a entrada é um grafo, por exemplo, podemos expressar a complexidade de tempo em função do número n de vértices e do número m de arestas do grafo. Assim, desejamos obter uma função $T(n, m)$. Por enquanto, porém, vamos desconsiderar este caso de várias variáveis.

Há outras alternativas para a complexidade de pior caso, mas, na maioria das situações, a complexidade de pior caso é considerada a melhor opção. Uma alternativa é a chamada complexidade de caso médio. Esta opção é motivada pela idéia que, se um algoritmo é rápido

para a esmagadora maioria das entradas, então pode ser aceitável que este algoritmo seja lento para algumas poucas entradas. Há algumas desvantagens da complexidade de caso médio. A primeira delas é que, na complexidade de caso médio, é necessário ter uma distribuição de probabilidade para as entradas. Outra desvantagem é que o cálculo da complexidade de caso médio pode ser extremamente complicado. Não adianta ter uma medida de complexidade que ninguém consegue calcular.

1.6. Complexidade Assintótica

Neste ponto, já definimos que a nossa função $T(n)$ corresponde ao tempo que uma determinada implementação do algoritmo leva em uma determinada máquina para a entrada de tamanho n mais demorada. Vamos agora nos livrar da dependência da máquina específica e dos detalhes de implementação. Para isto, lançamos mão da hierarquia assintótica, que explicamos nos próximos parágrafos.

Dizemos que $f(n) \preceq g(n)$ se existem constantes positivas c e n_0 tais que $f(n) \leq cg(n)$, para todo $n > n_0$. Analogamente, dizemos que $f(n) \succeq g(n)$ se existem constantes positivas c e n_0 tais que $f(n) \geq cg(n)$, para todo $n > n_0$.

Se $f(n) \preceq g(n)$ e $f(n) \succeq g(n)$, dizemos que $f(n) \asymp g(n)$. Se $f(n) \preceq g(n)$, mas não é verdade que $f(n) \asymp g(n)$, então dizemos que $f(n) \prec g(n)$. Analogamente, se $f(n) \succeq g(n)$, mas não é verdade que $f(n) \asymp g(n)$, então dizemos que $f(n) \succ g(n)$.

Vejam alguns exemplos com polinômios:

$$\begin{aligned} 3n^2 + 2n + 5 &\preceq n^2 \\ 3n^2 + 2n + 5 &\asymp n^2 \\ 3n^2 + 2n + 5 &\prec n^3 \\ 1 &\prec n \prec n^2 \prec n^3 \prec \dots \end{aligned}$$

Com algumas funções mais complexas, podemos escrever, por exemplo:

$$\begin{aligned} 1 &\prec \lg \lg n \prec \lg n \prec \lg^2 n \prec n^{1/3} \prec \sqrt{n} \prec n/\lg n \prec n \\ n &\prec n \lg n \prec n^2 \prec n^3 \prec 2^n \prec e^n \prec n! \prec n^n \end{aligned}$$

Esta notação assintótica que acabamos de apresentar, embora correta, é raramente utilizada em computação. No seu lugar, utiliza-se a comumente chamada notação O . Denota-se por $O(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \preceq g(n)$. Denota-se por $\Omega(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \succeq g(n)$. Denota-se por $\Theta(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \asymp g(n)$. Denota-se por $o(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \prec g(n)$. Denota-se por $\omega(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \succ g(n)$. Esta equivalência está resumida a seguir:

$$\begin{aligned} f(n) = O(g(n)) &\equiv f(n) \preceq g(n) \\ f(n) = \Omega(g(n)) &\equiv f(n) \succeq g(n) \\ f(n) = \Theta(g(n)) &\equiv f(n) \asymp g(n) \\ f(n) = o(g(n)) &\equiv f(n) \prec g(n) \\ f(n) = \omega(g(n)) &\equiv f(n) \succ g(n) \end{aligned}$$

Esta notação tem alguns aspectos extremamente práticos e outros extremamente confusos. Um ponto forte da notação O é que ela pode ser usada diretamente dentro de equações. Podemos dizer, por exemplo que $2n^4 + 3n^3 + 4n^2 + 5n + 6 = 2n^4 + 3n^3 + O(n^2)$. Um ponto negativo é que a notação O anula a reflexividade da igualdade. Podemos dizer que $n^2 = O(n^3)$, mas não podemos dizer que $n^3 = O(n^2)$.

Uma propriedade importante da notação O é que ela despreza constantes aditivas e multiplicativas. Sejam c_1 e c_2 constantes, então $c_1 f(n) + c_2 = \Theta(f(n))$. Desta propriedade seguem algumas simplificações como $\lg n^k = \Theta(\lg n)$ e $\log_k n = \Theta(\lg n)$, para qualquer constante k . Sempre que usamos um logaritmo dentro da notação O , optamos pela função $\lg n$, o logaritmo

de n na base 2. Afinal, como $\log_k n = \Theta(\lg n)$, qualquer logaritmo é equivalente nesse caso e o logaritmo na base 2 é o mais natural em computação.

Agora podemos terminar de definir o método que usamos para medir o tempo gasto por um algoritmo, independente da máquina. Certamente, uma máquina mais rápida está limitada a executar qualquer programa um número de vezes mais rápido que outra máquina. Assim, se expressarmos a função $T(n)$ usando notação O , não é necessário depender de uma máquina específica. Com isto, também não dependemos de muitos detalhes de implementação, embora alguns detalhes de implementação possam alterar a complexidade assintótica. Esta avaliação do algoritmo é chamada de complexidade de tempo assintótica de pior caso, mas muitas vezes nos referimos a ela apenas como complexidade de tempo, ou mesmo complexidade.

Como o próprio nome diz, a complexidade de tempo assintótica avalia o tempo gasto pelo algoritmo para entradas cujo tamanho tende a infinito. Se um algoritmo a tem complexidade de tempo $O(f(n))$ e outro algoritmo b tem complexidade de tempo $O(g(n))$, com $f(n) \prec g(n)$, então, certamente, a partir de algum valor de n o algoritmo a se torna mais rápido que o algoritmo b . Porém, pode ser verdade que o algoritmo a seja mais lento que o algoritmo b para entradas “pequenas”.

1.7. Análise de Complexidade

Vamos agora mostrar algumas técnicas usadas para analisar a complexidade de um algoritmo através de dois exemplos simples: os dois algoritmos de ordenação vistos anteriormente. Primeiro vamos analisar a ordenação por inserção, cujo pseudo-código está na figura 1.1.

Temos 3 *loops* neste algoritmo. O *loop* mais externo é repetido exatamente n vezes, onde n é o número de elementos da entrada. O número exato de repetições dos *loops* mais internos depende da entrada, porém é possível notar que o primeiro *loop* realiza no máximo $i-1$ repetições e o segundo *loop* realiza no máximo i repetições. De fato, o número de repetições dos dois *loops* internos somados é exatamente i , mas não precisamos entrar nesse nível de detalhes para obtermos um limite superior para a complexidade. O que importa é que os *loops* internos realizam $O(i)$ repetições e, dentro deles, só há operações cujo tempo independe do valor de n . Assim, a complexidade de tempo do algoritmo é

$$\sum_{i=1}^n O(i) = \sum_{i=1}^n O(n) = nO(n) = O(n^2).$$

Neste cálculo, substituímos $O(i)$ por $O(n)$, pois $i \leq n$. Claro que poderíamos estar perdendo precisão nesta substituição. Se quisermos fazer os cálculos justos, não podemos usar este truque e também precisamos garantir que há caso em que os *loops* internos realizam $\Omega(i)$ repetições, o que é verdade já que os dois *loops* somados realizam exatamente i repetições para qualquer entrada. Como $1 + 2 + \dots + n = n(n-1)/2 = \Theta(n^2)$, temos

$$\sum_{i=1}^n \Theta(i) = \Theta(n^2).$$

Deste modo, finalizamos a análise do algoritmo de ordenação por inserção. Outra análise que podemos fazer é a chamada complexidade de espaço, ou seja, a quantidade de memória necessária para a execução do algoritmo. No caso da ordenação por inserção, a complexidade de memória é claramente $\Theta(n)$, pois só temos 2 vetores com n elementos, além de um número constante de variáveis cujo tamanho independe de n .

A análise do algoritmo de ordenação por divisão e conquista é mais complicada. Este algoritmo divide a entrada em duas partes aproximadamente iguais, executa-se recursivamente para essas duas partes e depois combina as duas soluções. A fase de combinação das duas soluções leva tempo linear no tamanho da entrada. Com isso, podemos dizer que

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{para } n > 1 \\ O(1) & \text{para } n \leq 1 \end{cases}$$

Esta é uma relação de recorrência, pois $T(n)$ está expresso em função da própria função $T(\cdot)$. Usamos freqüentemente relações de recorrência para analisar a complexidade de tempo de algoritmos. Quando usamos relações de recorrência para este fim, podemos fazer algumas simplificações. A primeira delas é omitirmos o caso base (no caso, $n = 1$). Para qualquer algoritmo, o tempo que o algoritmo leva para entradas de tamanho constante é constante. Assim, usando notação assintótica, $T(k) = \Theta(1)$ para qualquer *constante* k . Por isso, o caso base $T(k) = \Theta(1)$ é sempre satisfeito e, para simplificarmos, podemos escrever a recorrência acima como:

$$T(n) = 2T(n/2) + \Theta(n).$$

Além disso, como estamos interessados apenas na complexidade assintótica de $T(n)$, podemos alterar livremente as constantes multiplicativas de funções não recorrentes de n , ou seja, podemos substituir, por exemplo, $\Theta(1)$ por 1, ou $n(n-1)/2$ por n^2 . Assim, podemos reescrever nossa recorrência como:

$$T(n) = 2T(n/2) + n.$$

Resolver relações de recorrência não é uma tarefa simples, de modo geral. Porém, se temos um chute da resposta, podemos prová-lo ou derrubá-lo usando indução. Para obtermos este chute, vamos imaginar a execução do algoritmo como uma árvore como na figura 1.4. Cada vértice representa uma execução do procedimento e o número indicado nele representa o número de elementos na entrada correspondente. Os dois filhos de um vértice correspondem as duas chamadas recursivas feitas a partir do vértice pai. O tempo gasto pelo algoritmo, conforme a relação de recorrência, é o número de elementos da entrada mais o tempo gasto em duas execuções recorrentes com metade dos elementos. Assim, desejamos obter a soma dos valores representados nos vértices da árvore. A soma dos vértices no último nível da árvore vale $\Theta(n)$, ou seja, o tempo gasto em todas as execuções com um elemento na entrada é $\Theta(n)$. O mesmo é válido para todas as execuções com 2 (ou 4, ou 8...) elementos na entrada, que correspondem a cada um dos níveis da árvore. Como a altura da árvore é $\Theta(\lg n)$, a soma das complexidades de tempo vale $\Theta(n \lg n)$.

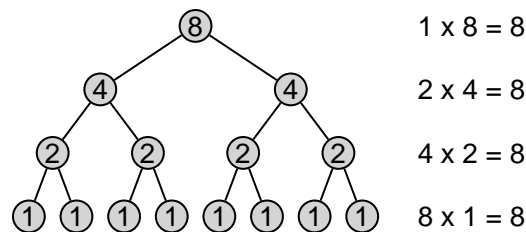


FIGURA 1.4. Árvore correspondente a execução do algoritmo de divisão e conquista em entrada de tamanho inicial 8.

Para provarmos que $T(n) \leq cn \lg n$ para alguma constante c , usando indução, fazemos:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leq 2cn/2 \lg(n/2) + n \\ &= cn \lg(n/2) + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n. \end{aligned}$$

Como não existe um procedimento simples para resolver qualquer relação de recorrência, foi criado um teorema para resolver as equações mais comumente encontradas em análise de

algoritmos. O teorema é chamado em teorema mestre e resolve equações de recorrência da forma geral

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

com a e b sendo constantes maiores ou iguais a 1. A solução se divide em três casos.

- 1) $f(n) = O(n^{\log_b a - \varepsilon})$, para alguma constante $\varepsilon > 0$. Neste caso temos

$$T(n) = \Theta(n^{\log_b a}).$$

- 2) $f(n) = \Theta(n^{\log_b a} \log^k n)$. Neste caso temos

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n).$$

- 3) $f(n) = \Omega(n^{\log_b a + \varepsilon})$, para alguma constante $\varepsilon > 0$ valendo também que $af(n/b) \leq cf(n)$ para alguma constante $c < 1$ e n suficientemente grande. Neste caso temos

$$T(n) = \Theta(f(n)).$$

1.8. Resumo e Observações Finais

Apresentamos três tipos de problemas que estudaremos nesse livro: problemas de decisão, problemas de construção e problemas de otimização. Todo problema possui uma entrada, ou instância, e uma saída desejada para cada entrada.

Um algoritmo é um método computacional para a solução do problema. Um paradigma é uma técnica usada para desenvolver algoritmos.

Quando desenvolvemos um algoritmo, precisamos provar que o algoritmo funciona, isto é, fornece a solução correta para o problema. Isto é chamado de prova de corretude. Algumas provas de corretude são bastante simples, enquanto outras são bastante complicadas.

Para compararmos a eficiência de algoritmos, precisamos definir o que chamamos de complexidade de tempo, pois uma medição de tempo na prática apresenta várias deficiências. A medida que mais usamos é chamada de complexidade de tempo assintótica de pior caso. O termo pior caso é usado porque sempre nos preocupamos com a entrada de um tamanho definido para a qual o algoritmo leva mais tempo. O termo assintótica é usado porque avaliamos quanto tempo o algoritmo leva para entradas grandes, com tamanho tendendo a infinito. Para expressarmos grandezas assintóticas, definimos a notação O .

Analisar a complexidade de tempo de um algoritmo nem sempre é uma tarefa simples. Muitas vezes, usamos relações de recorrência ou somatórios para esta tarefa.

Exercícios

- 1.1) Liste três problemas de cada um dos seguintes tipos: decisão, construção e otimização.
- 1.2) Descreva com pseudo-códigos os algoritmos usados normalmente para fazer adição e multiplicação de inteiros “na mão”. Analise a complexidade de tempo assintótica desses algoritmos, no pior caso, em função do número de algarismos dos dois operandos.
- 1.3) Realize as seguintes tarefas práticas com os dois algoritmos de ordenação descritos neste capítulo:
 - (a) Implemente corretamente os dois algoritmos da maneira mais eficiente que conseguir.
 - (b) Compare o tempo que cada um dos algoritmos gasta para ordenar listas aleatoriamente embaralhadas com tamanhos variados.
 - (c) Determine o tamanho k de lista para o qual o algoritmo de ordenação por inserção leva o mesmo tempo que o algoritmo de divisão e conquista.
 - (d) Modifique o algoritmo de divisão e conquista para, quando a lista possuir tamanho menor ou igual ao valor de k determinado no item anterior, executar o algoritmo de ordenação por inserção.
 - (e) Compare o tempo que esse novo algoritmo gasta para entradas de tamanhos variados.

- 1.4) Preencha a tabela abaixo com os valores de cada função. Em seguida, escreva cada função na forma mais simples usando notação Θ . Finalmente, coloque estas funções em ordem crescente segundo a hierarquia assintótica.

| | 2 | 3 | 5 | 10 | 30 | 100 |
|--------------------|---|---|---|----|----|-----|
| $7n + \sqrt{n}$ | | | | | | |
| $2^n/100$ | | | | | | |
| $n/\lg n$ | | | | | | |
| $\lg n^3$ | | | | | | |
| $2n^2$ | | | | | | |
| $n! - n^3$ | | | | | | |
| $(\lg \lg n)^2$ | | | | | | |
| $\lg n + \sqrt{n}$ | | | | | | |
| $\lg(n!)$ | | | | | | |

- 1.5) Considere a recorrência

$$T(n) = T(n/2) + 1.$$

A solução correta desta recorrência satisfaz $T(n) = \Theta(\lg n)$. Ache o erro na demonstração abaixo, que prova que $T(n) = O(\lg \lg n)$:

Vamos supor, para obter uma prova por indução, que $T(i) = O(\lg \lg i)$ para $i \leq n$. Vamos calcular $T(n+1)$. Temos: $T(n+1) = T(n/2) + 1 = O(\lg \lg(n/2)) + 1$. Como $\lg \lg(n/2) = O(\lg \lg(n+1))$ temos $T(n+1) = O(\lg \lg(n+1)) + 1 = O(\lg \lg(n+1))$, finalizando a indução.

- 1.6) Prove que a recorrência $T(n) = T(n/2) + 1$ satisfaz $T(n) = O(\lg n)$.

- *1.7) Prove que a recorrência abaixo satisfaz $f(n) = n$, considerando o caso base $f(1) = 1$:

$$f(n) = \sum_{i=0}^{n-2} \binom{n-2}{i} \frac{1}{2^{n-3}} f(i+1).$$