Minicurso de Análise de Algoritmos

https://www.ime.usp.br/~pf/livrinho-AA/

Paulo Feofiloff
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

6 de agosto de 2019

Sumário

1 Introdução **9**

1.1 Problemas e instâncias 9
1.2 Consumo de tempo de algoritmos 10
1.3 Notações O, Ômega e Teta 11
1.4 Convenções de notação e terminologia 12

2 Recursão e recorrências **13**

2.1 Algoritmos recursivos 13
2.2 Recorrências 14

3 Ordenação de vetor **17**

3.1 O problema da ordenação 17
3.2 Algoritmo Mergesort 18
3.3 Desempenho 19
3.4 Observações sobre divisão e conquista 21

4 Segmento de soma máxima **23**

4.1 O problema 23
4.2 Primeiro algoritmo 24
4.3 Segundo algoritmo 25
4.4 Terceiro algoritmo: divisão e conquista 27
4.5 Quarto algoritmo: programação dinâmica 28
4.6 Observações sobre programação dinâmica 30

5 O problema da maioria **31**

5.1 O problema 31
5.2 A estrutura recursiva do problema 32
5.3 Um algoritmo linear 33

6 Multiplicação de grandes números 37
6.1 O problema 38
6.2 O algoritmo usual 38
6.3 Observações sobre o método de divisão e conquista 38
6.4 Algoritmo de Karatsuba 40
6.5 Detalhes de implementação 43
6.6 Observações sobre divisão e conquista 44

7 Intervalos disjuntos 45
7.1 Formalização do problema 45
7.2 A propriedade gulosa do problema 46
7.3 Um algoritmo guloso 47
7.4 Implemementação do algoritmo guloso 48
7.5 Observações sobre algoritmos gulosos 49

8 As linhas de um parágrafo 51
8.1 O problema 51
8.2 A estrutura recursiva do problema 52
8.3 Um algoritmo de programação dinâmica 53

9 Mochila de valor máximo 55
9.1 O problema 55
9.2 A estrutura recursiva do problema 55
9.3 Algoritmo de programação dinâmica 56
9.4 Instâncias especiais do problema 58

10 Mochila de valor quase máximo 59
10.1 O problema 59
10.2 Um algoritmo de aproximação 59
10.3 Observações sobre algoritmos de aproximação 61

11 A cobertura de um grafo 63
11.1 Grafos 63
11.2 O problema da cobertura 63
11.3 Um algoritmo de aproximação 64
11.4 Comentários sobre o método primal-dual 66
11.5 Instâncias especiais do problema 66

12 Conjuntos independentes em grafos 67
12.1 O problema . . . 67
12.2 Um algoritmo probabilístico . . . 68
12.3 Comentários sobre algoritmos probabilísticos . . . 70

13 Busca em largura num grafo 71
13.1 O problema do componente . . . 71
13.2 Busca em largura: versão preliminar . . . 71
13.3 Busca em largura: versão mais concreta . . . 73

14 Busca em profundidade num grafo 75
14.1 Busca em profundidade . . . 75

A Comparação assintótica de funções 79
A.1 Notação O . . . 79
A.2 Notação Ômega . . . 80
A.3 Notação Teta . . . 81
A.4 Consumo de tempo de algoritmos . . . 81

B Solução de recorrências 83
B.1 Exemplo 1 . . . 83
B.2 Exemplo 2 . . . 84
B.3 Exemplo 3 . . . 84
B.4 Exemplo 4 . . . 86
B.5 Teorema mestre . . . 88

Posfácio 90

Bibliografia 93

Índice remissivo 95

Prefácio

A Análise de Algoritmos é uma disciplina bem-estabelecida (veja o verbete *Analysis of Algorithms* na Wikipedia) e coberta por um bom número de excelentes livros, como os de Cormen, Leiserson, Rivest e Stein [3], Brassard e Bratley [2], Bentley [1], Kleinberg e Tardos [10], e Manber [14].

Este livrinho faz uma modesta introdução ao assunto. O texto analisa alguns algoritmos clássicos, discute sua eficiência, e chama a atenção para as estratégias (divisão e conquista, programação dinâmica, gula, primal-dual, busca em profundidade, probabilística, aproximação, etc.) que levaram à sua concepção.

A análise de algoritmos requer uma certa familiaridade com duas ferramentas matemáticas: a notação assintótica e a resolução de recorrências. Esses assuntos são apresentados no capítulo introdutório e depois detalhados nos apêndices.

O livrinho é uma versão corrigida do minicurso que ministrei nas Jornadas de Atualização em Informática (JAI) da Sociedade Brasileira de Computação (SBC), ocorridas em Bento Gonçalves, RS, em julho de 2009. O minicurso foi publicado no livro organizado por André P. L. F. de Carvalho e Tomasz Kowaltowski [5].

O texto deve muito a Cristina Gomes Fernandes e José Coelho de Pina Jr., ambos do Departamento de Ciência da Computação do Instituto de Matemática e Estatística da USP, que contribuíram com material, ideias e valiosas sugestões.

IME–USP, São Paulo, abril de 2015
P. F.

Capítulo 1

Introdução

Análise de Algoritmos[1] é o estudo da quantidade de recursos computacionais — tempo e memória — que algoritmos consomem. Pode-se dizer que a Análise de Algoritmos é uma disciplina de engenharia, pois procura *prever* o comportamento de um algoritmo antes que ele seja efetivamente implementado e colocado "em produção".

Num nível mais abstrato, a Análise de Algoritmos procura identificar aspectos estruturais comuns a algoritmos diferentes e estudar métodos e paradigmas de projeto de algoritmos (como divisão e conquista, programação dinâmica, gula, primal-dual, busca em profundidade, probabilística, aproximação, etc.).

Este texto é uma breve introdução à Análise de Algoritmos. Ele analisa algoritmos para alguns problemas computacionais básicos (em geral de natureza combinatória) que aparecem em uma grande variedade de contextos. Para cada problema e algoritmo, a análise

1. prova que o algoritmo está correto e
2. estima o tempo que a execução do algoritmo consome.

Essa análise permite comparar dois algoritmos diferentes para um mesmo problema e decidir qual dos dois é mais eficiente.

No restante deste capítulo introdutório, faremos uma rápida revisão de conceitos básicos e fixaremos a notação e a terminologia empregadas no texto.

1.1 Problemas e instâncias

Em geral, problemas computacionais têm um ou mais parâmetros, ou "dados de entrada". Quando os valores desses parâmetros são especificados, temos uma **instância** do problema.[2] Assim, todo problema computacional é uma coleção (em geral infinita) de instâncias.

Considere, por exemplo, o problema da equação do segundo grau: encontrar um número x tal que $ax^2 + bx + c = 0$. Os números a, b e c são os parâmetros do problema.

[1] A expressão *analysis of algorithms* foi cunhada por D. E. Knuth [12].

[2] A palavra *instância* é um neologismo importado do inglês. Ela está sendo empregada aqui no sentido de *caso particular, exemplo, espécime, amostra*.

Uma das instâncias desse problema consiste em encontrar um número x tal que $11x^2 - 22x + 33 = 0$.

Outro problema: encontrar a média dos elementos de um vetor $A[1\,..\,n]$ de números. O parâmetro desse problema é o vetor $A[1\,..\,n]$. Uma das instâncias do problema consiste em encontrar a média dos elementos do vetor $(876, -145, 323, 112, 221)$.

Em discussões informais, pode ser aceitável dizer "problema" quando "instância do problema" seria mais correto. Em geral, entretanto, é importante manter clara a distinção entre os dois conceitos.

O **tamanho** de uma instância de um problema é a quantidade de dados necessária para descrever a instância. A ideia de tamanho permite dizer que uma instância é *menor* ou *maior* que outra.

No problema da média citado acima, é razoável dizer que o tamanho da instância $(876, -145, 323, 112, 221)$ é 5 e que o tamanho de uma instância $A[1\,..\,n]$ é n. Já no problema da equação do segundo grau, parece mais razoável dizer que o tamanho de uma instância do problema é o número total de dígitos decimais em (a, b, c). A instância $(11, -22, 33)$, por exemplo, tem tamanho 6 (ou 7, se o leitor contar o sinal negativo).

1.2 Consumo de tempo de algoritmos

Dizemos que um algoritmo **resolve** um problema se, ao receber a descrição de uma instância do problema, devolve uma solução da instância (ou informa que a instância não tem solução).[3]

Para cada instância do problema, o algoritmo consome uma quantidade de tempo diferente. Digamos que o algoritmo consome $T(I)$ unidades de tempo para resolver a instância I. A relação entre $T(I)$ e o tamanho de I dá uma medida da eficiência do algoritmo.

Em geral, um problema tem muitas instâncias diferentes de um mesmo tamanho. Dado um algoritmo $\mathcal{A}$ para um problema e um número natural n, denotaremos por $T^*(n)$ o máximo de $T(I)$ para todas as instâncias I que têm tamanho n.[4] Diremos que

T^* mede o desempenho de $\mathcal{A}$ **no pior caso**.

De modo análogo, denotaremos por $T_*(n)$ o mínimo de $T(I)$ para todas as instância I de tamanho n e diremos que T_* mede o desempenho de $\mathcal{A}$ **no melhor caso**. (Mas, em geral, usaremos a palavra *desempenho* apenas em referência ao pior caso.)

Por exemplo, um algoritmo pode consumir $200n$ unidades de tempo no melhor caso e $10n^2 + 100n$ unidades no pior. (À primeira vista, pode parecer que há algo errado com esse exemplo pois $200n$ é *maior* que $10n^2 + 100n$ para valores pequenos de n. Isso chama a atenção para um detalhe importante: só estamos interessados nos valores *grandes* de n.)

[3] Em geral, nem todas as instâncias de um problema têm solução. A tarefa de decidir se uma dada instância tem solução faz parte do problema.

[4] Na verdade, as coisas não são tão simples assim, pois o máximo pode não estar definido.

$\log n$	2	3	4	5	6
n	100	1000	10000	100000	1000000
$n \log n$	200	3000	40000	500000	6000000
n^2	10000	1000000	100000000	10000000000	1000000000000
n^3	1000000	1000000000	1000000000000	1000000000000000	1000000000000000000

Figura 1.1: Valores das funções $\log n$, n, $n \log n$, n^2 e n^3 para n entre 100 e 1 milhão. Cada função domina a anterior, e esse efeito aumenta com o valor de n. (Para simplificar, usamos o logaritmo na base 10; o efeito é conceitualmente o mesmo se usarmos logaritmo na base 2.)

1.3 Notações O, Ômega e Teta

A análise de algoritmos trata principalmente do desempenho de *pior* caso. Felizmente, não é preciso determinar uma fórmula exata para $T^*(n)$: basta entender o comportamento *assintótico* da função, ou seja, o comportamento quando n tende a infinito.

Para valores grandes de n, o termo dominante das fórmulas é muito maior que os demais e portanto os termos de ordem inferior podem ser desprezados (veja a figura 1.1). Na fórmula $10n^2 + 100n$, por exemplo, o segundo termo pode ser despezado porque é dominado pelo primeiro. Assim, a fórmula se reduz a $10n^2$.

Além disso, as constantes multiplicativas — como o "10" em "$10n^2$" — podem ser ignoradas pois seu efeito será anulado se o computador for substituído por outro mais rápido. Com tudo isso, em vez de dizer que $T^*(n) = 10n^2 + 100n$, é mais apropriado dizer que

$$T^*(n) \text{ está em } \mathrm{O}(n^2),$$

onde o "O" serve para esconder os termos de ordem inferior e as constantes.

Na verdade, a notação O( ) esconde mais que as constantes e os termos de ordem inferior, pois ela tem o sabor de "$\leq$". Ao dizer que $T^*(n)$ está em $\mathrm{O}(n^2)$, estamos afirmando apenas que $T^*(n) \leq cn^2$ para alguma constante c (e todo n suficientemente grande). Assim, por exemplo, é perfeitamente correto dizer que $55n+66$ está em $\mathrm{O}(n^2)$. (Veja a definição precisa da notação O( ) no Apêndice A.)

Apesar do sabor "$\leq$" de O( ), tornou-se hábito escrever "$T^*(n) = \mathrm{O}(n^2)$" no lugar de "$T^*(n)$ está em $\mathrm{O}(n^2)$", o que pode ser um tanto confuso ...

Há uma notação análoga com sabor de "$\geq$": dizemos que $T^*(n)$ está em $\Omega(n^2)$, ou que $T^*(n) = \Omega(n^2)$, se $T^*(n) \geq dn^2$ para alguma constante d (e todo n suficientemente grande).

Finalmente, dizemos que $T^*(n)$ está em $\Theta(n^2)$, ou que $T^*(n) = \Theta(n^2)$, se $T^*(n)$ está em $\mathrm{O}(n^2)$ e também em $\Omega(n^2)$. Assim, a notação $\Theta(\,)$ tem sabor de "$=$".

Algoritmos lineares, linearítmicos e quadráticos. Um algoritmo é **linear** se seu desempenho no pior caso está em $\Theta(n)$. É fácil entender o comportamento de um tal algoritmo: quando o tamanho de uma instância do problema dobra, o algorimo consome duas vezes mais tempo. Algoritmos lineares são considerados muito rápidos pois o tempo que consomem é essencialmente igual ao necessário para a leitura dos dados de entrada.

Um algoritmo é **linearítmico** (ou **ene-log-ene**) se seu desempenho no pior caso está em $\Theta(n \log n)$. Se o tamanho n da instância do problema dobra, o consumo de tempo dobra e é acrescido de $2n$.

Um algoritmo é **quadrático** se consome $\Theta(n^2)$ unidades de tempo no pior caso. Se o tamanho da instância dobra, o consumo quadruplica.

Exercícios

1.1 O que é um algoritmo *de tempo constante*? Como ele se comporta quando o tamanho da instância do problema dobra?

1.2 O que é um algoritmo *logarítmico*? Como ele se comporta quando o tamanho da instância do problema dobra? Em quanto o tamanho da instância precisa aumentar para que o consumo de tempo dobre?

1.3 O que é um algoritmo *cúbico*? Como um algoritmo cúbico se comporta quando o tamanho da instância do problema dobra?

1.4 Convenções de notação e terminologia

Diremos que um número x é **positivo** se $x \geq 0$ e **estritamente** positivo se $x > 0$. Analogamente, x é **negativo** se $x \leq 0$ e **estritamente** negativo se $x < 0$.

Um vetor $A[p\,..\,r]$ é **crescente** se $A[p] \leq A[p+1] \leq \cdots \leq A[r]$ e **estritamente** crescente se $A[p] < A[p+1] < \cdots < A[r]$. As definições de **decrescente** e **estritamente** decrescente são análogas.

Os elementos de um vetor serão, em geral, indexados a partir de 1 e não a partir de 0, como se faz em muitas linguagens de programação.

$\mathbb{Z}$ $\mathbb{N}$ Denotaremos por $\mathbb{Z}$ o conjunto $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ dos números inteiros. Denotaremos por $\mathbb{N}$ o conjunto $\{0, 1, 2, 3, \ldots\}$ dos números naturais. (que coincide com o dos inteiros positivos). O conjunto $\mathbb{N} - \{0\}$ será denotado por $\mathbb{N}^>$.

O **piso** de um número real positivo x é o único número i em $\mathbb{N}$ tal que $i \leq x < i+1$. O **teto** de x é o único número j em $\mathbb{N}$ tal que $j - 1 < x \leq j$. O piso e o teto de x são denotados por $\lfloor x \rfloor$ e $\lceil x \rceil$ respectivamente.

$\lfloor x \rfloor$ $\lceil x \rceil$ Expressões como "$n/2$" representam um quociente exato (e não o piso do quociente, como acontece em muitas linguagens de programação). Assim, por exemplo, $15/2$ vale 7.5 e não 7.

$\lg n$ Para qualquer n inteiro estritamente positivo, denotaremos por $\lg n$ o número $\log_2 n$, ou seja, o logaritmo de n na base 2 (que é aproximadamente igual ao número de bits na representação binária de n). É claro que $\lfloor \lg n \rfloor$ é o único número natural j tal que $2^j \leq n < 2^{j+1}$.

Exercícios

1.4 Mostre que $\log_{10} n$ está em $\Theta(\lg n)$. Mostre que $\lg n$ está em $\Theta(\log_{10} n)$. (Portanto, $\log_{10} n$ e $\lg n$ são assintoticamente equivalentes.)

Capítulo 2

Recursão e recorrências

Algoritmos recursivos resolvem problemas dotados de estrutura recursiva. A análise de algoritmos recursivos envolve a solução de recorrências. Este capítulo faz uma breve introdução à recursão e à solução de recorrências. O Apêndice B estuda recorrências de maneira mais detalhada.

2.1 Algoritmos recursivos

Muitos problemas computacionais têm a seguinte propriedade: cada solução de uma instância do problema contém soluções de instâncias menores do mesmo problema. Dizemos que tais problemas têm *estrutura recursiva*. Para resolver um problema desse tipo, aplique o seguinte método: se a instância dada é pequena, resolva-a diretamente (use força bruta se necessário); se a instância é grande,

1. reduza-a a uma instância menor,
2. encontre uma solução S da instância menor,
3. use S para construir uma solução da instância original.

A aplicação desse método produz um algoritmo *recursivo*.

Para provar que um algoritmo recursivo está correto, basta fazer uma indução matemática no tamanho das instâncias. Antes de empreender a prova, é essencial colocar no papel, de maneira precisa e completa, a relação entre o que o algoritmo recebe e o que devolve.

Exemplo 1: soma dos elementos positivos de um vetor. O seguinte algoritmo recursivo recebe um vetor $A[1\,..\,n]$ de números inteiros, com $n \geq 0$, e devolve a soma dos elementos positivos do vetor:[1]

[1] Usaremos a excelente notação do livro de Cormen *et al.* [3] para descrever algoritmos. Nessa notação, expressões como "$x = y$" e "$x \leftarrow y$" correspondem, respectivamente, às expressões "`x == y`" e "`x = y`" em linguagens de programação como C e Java.

```
SOMAPOSITIVOS (A, n)
1  se n = 0
2     então  devolva 0
3     senão  s ← SOMAPOSITIVOS (A, n − 1)
4            se A[n] > 0
5               então  devolva s + A[n]
6               senão  devolva s
```

A prova da correção do algoritmo é uma indução em n. Se $n = 0$, é evidente que o algoritmo dá a resposta correta. Agora tome $n \geq 1$ e suponha — essa é nossa hipótese de indução — que SOMAPOSITIVOS $(A, n-1)$ produz o resultado prometido, ou seja, a soma dos elementos positivos de $A[1\,..\,n-1]$. Então, as linhas 4–6 calculam corretamente a soma dos elementos positivos de $A[1\,..\,n]$.

Exemplo 2: piso do logaritmo. O seguinte algoritmo recursivo recebe um número natural $n \geq 1$ e devolve $\lfloor \lg n \rfloor$:

```
PISODELOG (n)
1  se n = 1
2     então  devolva 0
3     senão  devolva PISODELOG (⌊n/2⌋) + 1
```

Prova da correção do algoritmo: Se $n = 1$, é evidente que o algoritmo devolve o valor correto de $\lfloor \lg n \rfloor$. Agora tome $n \geq 2$ e seja k o número $\lfloor \lg n \rfloor$. Queremos mostrar que o algoritmo devolve k.

É claro que $2^k \leq n < 2^{k+1}$, e portanto $2^{k-1} \leq n/2 < 2^k$. Como 2^{k-1} é um número natural, temos $2^{k-1} \leq \lfloor n/2 \rfloor < 2^k$ e portanto $\lfloor \lg \lfloor n/2 \rfloor \rfloor = k - 1$. Como $\lfloor n/2 \rfloor < n$, podemos supor, por hipótese de indução, que o valor da expressão PISODELOG $(\lfloor n/2 \rfloor)$ é $k - 1$. Portanto, o algoritmo devolve $k - 1 + 1 = k$, como queríamos provar.

2.2 Recorrências

Para analisar algoritmos recursivos é necessário resolver recorrências. Uma *recorrência* é uma fórmula que define uma função, digamos T, em termos dela mesma. Mais precisamente, a recorrência define $T(n)$ em termos de $T(n-1)$, $T(n-2)$, $T(n-3)$, etc. Uma *solução* de uma recorrência é uma fórmula que exprime $T(n)$ em termos de n apenas.

Considere, por exemplo, o algoritmo SOMAPOSITIVOS da Seção 2.1. Digamos que $T(n)$ é a função que dá o consumo de tempo no pior caso. Se a execução de qualquer das linhas do pseudocódigo consome uma unidade de tempo, então $T(0) = 2$ e $T(n)$ satisfaz a recorrência

$$T(n) = T(n-1) + 3\,. \tag{2.1}$$

(O termo $T(n-1)$ corresponde à linha 3 e o termo 3 é o consumo de tempo das linhas 1 e linhas 4–6.) Assim, $T(1) = 2 + 3 = 5$, $T(2) = 5 + 3 = 8$, $T(3) = 8 + 3 = 11$, etc.

Para resolver a recorrência, basta "desenrolá-la" como segue:

$$\begin{aligned} T(n) &= T(n-1) + 3 \\ &= (T(n-2) + 3) + 3 \\ &= T(n-2) + 6 \\ &= T(n-3) + 9 \\ &= \cdots \\ &= T(n-j) + 3j \\ &= T(0) + 3n \\ &= 2 + 3n\,. \end{aligned}$$

Portanto, $3n + 2$ é a solução da recorrência (2.1).

Em geral, resolver uma recorrência não é tão fácil quanto nesse exemplo. Felizmente, uma fórmula exata para $T(n)$ não é realmente necessária. Em geral, basta obter uma boa cota superior que seja válida a partir de algum valor de n. No exemplo acima, é fácil mostrar, por indução em n, que

$$T(n) \le 4n \tag{2.2}$$

a partir de $n = 2$. Prova: A desigualdade evidentemente vale quando $n = 2$. Já quando $n \ge 3$, podemos supor que $T(n-1) \le 4(n-1)$ por hipótese de indução e assim $T(n) = T(n-1) + 3 \le 4(n-1) + 3 = 4n - 1 < 4n$, como queríamos provar.

É igualmente fácil provar a cota inferior

$$T(n) \ge 3n$$

a partir de $n = 1$. Essas duas cotas mostram que $T(n)$ está em $\Theta(n)$. Portanto, o algoritmo é linear (veja a Seção 1.3).

Exercícios

2.1 Considere a função $T(n)$ definida pela recorrência (2.1). Mostre que $T(n) \le \frac{7}{2}n$ para $n = 4, 5, 6, \ldots$

2.2 Seja $T(n)$ o consumo de tempo do PISODELOG da Seção 2.1. Escreva a recorrência que $T(n)$ satisfaz. Mostre que $T(n)$ está em $\Theta(\lg n)$.

Capítulo 3

Ordenação de vetor

A tarefa de rearranjar um vetor em ordem crescente é um dos problemas computacionais mais conhecidos e bem-estudados.

111	333	999	555	777	888	222	444	777	999	555
111	222	333	444	555	555	777	777	888	999	999

Figura 3.1: Na primeira linha, o vetor original. Na segunda, o vetor rearranjado em ordem crescente.

Este capítulo discute o célebre algoritmo Mergesort para o problema. A discussão serve para mostrar (1) como se analisa um algoritmo recursivo e (2) como se resolve uma recorrência. Também serve de introdução ao método de divisão e conquista no projeto de algoritmos.

3.1 O problema da ordenação

Suporemos que os elementos do vetor a ordenar são números inteiros, mas a discussão se aplica igualmente bem à ordenação de quaisquer objetos comparáveis, como números reais, cadeias de caracteres, etc. Basta que possamos dizer se um elemento do vetor é maior, igual, ou menor que outro.

Problema da ordenação: Rearranjar um vetor $A[p \mathrel{.\,.} r]$ de inteiros em ordem crescente.

Exercícios

3.1 Todas as instâncias do problema da ordenação têm solução? O problema faz sentido se o vetor é vazio? Qual a solução das instâncias em que o vetor tem um só elemento?

3.2 O problema faz sentido se trocarmos "ordem crescente" por "ordem estritamente crescente"?

3.2 Algoritmo Mergesort

O seguinte algoritmo é uma solução eficiente do problema da ordenação:

```
MERGESORT (A, p, r)
1  se p < r
2     então  q ← ⌊(p + r)/2⌋
3            MERGESORT (A, p, q)
4            MERGESORT (A, q+1, r)
5            INTERCALA (A, p, q, r)
```

A rotina INTERCALA rearranja o vetor $A[p\,..\,r]$ em ordem crescente se os subvetores $A[p\,..\,q]$ e $A[q{+}1\,..\,r]$ estiverem ambos em ordem crescente.

O algoritmo está correto. Adote o número $n := r - p + 1$ como tamanho da instância (A, p, r) do problema. Se $n \leq 1$, o vetor tem no máximo 1 elemento e portanto não fazer nada é a ação correta. Suponha agora que $n \geq 2$. Nas linhas 3 e 4, o algoritmo é aplicado a instâncias do problema que têm tamanho estritamente menor que n. Podemos supor então, por hipótese de indução, que os vetores $A[p\,..\,q]$ e $A[q{+}1\,..\,r]$ estão em ordem crescente no início da linha 5. No fim da linha 5, graças à ação de INTERCALA, o vetor $A[p\,..\,r]$ estará em ordem crescente.

Figura 3.2: Vetor A no início da linha 5 do algoritmo.

Exercícios

3.3 Descreva o algoritmo INTERCALA em pseudocódigo. Mostre que ele consome no máximo $\Theta(n)$ unidades de tempo.

3.3 Desempenho

Seja $T(n)$ o tempo que o algoritmo MERGESORT consome, no pior caso, para processar uma instância de tamanho n. Se contarmos o tempo pelo número de linhas de pseudocódigo executadas, então $T(0) = T(1) = 1$ e a função $T(n)$ satisfaz a recorrência

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \tag{3.1}$$

(veja a Seção 2.2). As parcelas $T(\lceil n/2 \rceil)$ e $T(\lfloor n/2 \rfloor)$ correspondem às linhas 3 e 4. A parcela n representa o consumo de tempo da linha 5, uma vez que o algoritmo INTERCALA pode ser implementado de maneira a consumir tempo proporcional a n.

Felizmente, não precisamos de uma solução exata da recorrência (3.1). Ficaremos muito satisfeitos com uma fórmula assintótica, como $T(n) = \mathrm{O}(n \lg n)$ por exemplo (veja a Seção 1.3). O primeiro passo nessa direção é resolver uma recorrência semelhante mais simples.

Restrição às potências de 2. A restrição da recorrência às potências de 2 tem a vantagem de eliminar os incômodos "$\lfloor\ \rfloor$" e "$\lceil\ \rceil$": para $N = 2^1, 2^2, 2^3, \ldots$, temos simplesmente

$$T(N) = 2\,T(N/2) + N\,. \tag{3.2}$$

Essa recorrência pode ser "desenrolada" da seguinte maneira:

$$\begin{aligned}
T(N) &= 2\,T(N/2) + N \\
&= 2(2\,T(N/4) + N/2) + N \\
&= 2^2 T(N/2^2) + 2N \\
&= 2^3 T(N/2^3) + 3N \\
&= 2^j T(N/2^j) + jN \\
&= N T(N/N) + N \lg N \qquad (3.3)\\
&= N + N \lg N. \qquad (3.4)
\end{aligned}$$

Na linha (3.3) tomamos $j = \lg N$ e na linha (3.4) usamos o valor inicial[1] $T(1) = 1$.

É interessante conferir esse resultado por indução em N. A fórmula (3.4) está evidentemente correta quando $N = 1$. Suponha agora que $N \geq 2$ e que a fórmula está correta se trocarmos "N" por "$N/2$". Então,

$$\begin{aligned}
T(N) &= 2T(\tfrac{N}{2}) + N \\
&= 2(\tfrac{N}{2} \lg \tfrac{N}{2} + \tfrac{N}{2}) + N \\
&= N \lg \tfrac{N}{2} + N + N \\
&= N(\lg N - 1) + 2N \\
&= N \lg N + N,
\end{aligned}$$

[1] A intuição sugere (e a experiência confirma) que os valores iniciais de uma recorrência — ou seja, os valores de $T(n)$ quando n é pequeno — não têm influência sobre a forma geral de $T(n)$. Podemos então adotar quaisquer valores convenientes.

confirmando (3.4).

Embora a fórmula $N \lg N + N$ esteja correta apenas quando N é potência de 2, é seguro supor que o termo dominante $N \lg N$ descreve corretamente o comportamento assintótico de $T(N)$ para todos os demais valores de N (exceto os muito pequenos). Poderíamos, então, aceitar com base na intuição que a solução da recorrência (3.1) está em $\Theta(n \lg n)$. Mas é interessante confirmar essa conclusão calculando uma cota superior para T.

Cota superior para todo n. Usaremos a fórmula $N \lg N + N$ para mostrar que a solução $T(n)$ da recorrência (3.1) satisfaz a desigualdade

$$T(n) \leq 4n \lg n \tag{3.5}$$

para todo n a partir de 4.

Prova: Seja $j = \lceil \lg n \rceil$ e $N = 2^j$. Então $2^{j-1} < n \leq 2^j$ e portanto $N/2 < n \leq N$. Como T é crescente (veja o exercício 3.6), temos $T(n) \leq T(N)$. Portanto,

$$\begin{aligned} T(n) &\leq N \lg N + N \\ &= N(1 + \lg N) \\ &< 2n\,(1 + \lg 2n) \\ &= 2n\,(1 + 1 + \lg n) \\ &\leq 2n\,(2 \lg n) \qquad (3.6) \\ &= 4n \lg n\,. \end{aligned}$$

O passo (3.6) está correto pois $4 \leq n$, e portanto $2 \leq \lg n$.

O algoritmo é linearítmico. A cota superior (3.5) mostra (veja a Seção 1.3) que $T(n)$ está em $\mathrm{O}(n \lg n)$. Cálculos semelhantes mostram que $T(n) \geq \frac{1}{2} n \lg n$ para todo $n \geq 2$, donde $T(n)$ está em $\Omega(n \lg n)$. Segue daí que

$$T(n) \text{ está em } \Theta(n \lg n).$$

Portanto, o algoritmo MERGESORT é linearítmico.

Exercícios

3.4 Calcule os valores de $T(n)$ determinados pela recorrência (3.1) para $n = 2, 3, \ldots, 10$.

3.5 Calcule os valores de $T(n)$ determinados pela recorrência (3.1) para $n = 3, 4, \ldots, 10$ supondo que $T(1) = 2$ e $T(2) = 3$.

3.6 Mostre que a solução T da recorrência (3.1) é estritamente crescente, ou seja, que $T(n) \leq T(n+1)$ para todo n.

3.7 Mostre que $T(n) \geq \frac{1}{2} n \lg n$ para todo $n \geq 2$.

3.8 Mostre que o consumo de tempo do algoritmo MERGESORT no *melhor* caso está em $\Theta(n \lg n)$.

3.9 NÚMERO DE COMPARAÇÕES. A operação elementar mais significativa do algoritmo MERGESORT é a comparação entre elementos do vetor. Ocorrem no máximo n dessas operações (veja o exercício 3.3) e todas estão no algoritmo auxiliar INTERCALA. Seja $T(n)$ o número de comparações entre elementos do vetor que o algoritmo executa, no pior caso, ao processar um vetor com n elementos. Mostre que $T(n)$ satisfaz uma recorrência essencialmente igual a (3.1). Mostre que $T(n)$ está em $\Theta(n \lg n)$.

3.4 Observações sobre divisão e conquista

O algoritmo MERGESORT emprega a estratégia da divisão e conquista: a instância original do problema é dividida em duas instâncias menores, essas instâncias são resolvidas recursivamente, e as soluções são combinadas para produzir uma solução da instância original.

O segredo do sucesso está no fato de que o processo de divisão (que está na linha 2, nesse caso) e o processo da combinação (que acontece na linha 5) consomem pouco tempo.

O método da divisão e conquista rende algoritmos eficientes para muitos problemas. Veremos exemplos nos capítulos seguintes.

Capítulo 4

Segmento de soma máxima

Dado o registro da variação diária de temperatura num determinado lugar ao longo de uma década, queremos encontrar uma sequência de dias em que a variação acumulada tenha sido máxima.

$$20 \quad -30 \quad 15 \quad -10 \quad 30 \quad -20 \quad -30 \quad 30$$

Figura 4.1: Variação da temperatura (em décimos de grau) em relação ao dia anterior ao longo de 8 dias. A variação acumulada foi máxima no período que começou entre os dias 2 e 3 e terminou entre os dias 5 e 6.

Este capítulo estuda quatro algoritmos para o problema. Cada algoritmo é mais eficiente que o anterior; cada algoritmo usa uma estratégia diferente.

O capítulo contém lições de caráter geral sobre (1) a estimativa do desempenho de algoritmos iterativos, (2) o método da divisão e conquista, (3) o método de programação dinâmica, (4) a importância de entender bem o problema antes de tentar inventar um algoritmo, (5) o perigo de ficar satisfeito com o algoritmo óbvio, (6) a eventual necessidade de generalizar o problema para chegar a um algoritmo eficiente.

4.1 O problema

Um **segmento** de um vetor $A[1\,..\,n]$ é qualquer subvetor da forma $A[i\,..\,k]$, com $1 \leq i \leq k \leq n$. A condição $i \leq k$ garante que segmentos não são vazios.[1] A **soma** de um segmento $A[i\,..\,k]$ é o número $A[i] + A[i+1] + \cdots + A[k]$.

A **altura** de um vetor $A[1\,..\,n]$ é a soma de um segmento de soma máxima. (Por exemplo, a altura do vetor na Figura 4.1 é $15 - 10 + 30 = 35$.)

---

[1] Teria sido mais "limpo" e natural aceitar segmentos vazios. Mas os segmentos vazios poderiam tornar a discussão sutil demais em algumas ocasiões.

Problema do segmento de soma máxima: Calcular a altura de um vetor $A[1\,..\,n]$ de números inteiros.

É importante entender muito bem as sutilezas do problema antes de começar a propor soluções. É verdade, por exemplo, que todas as instâncias do problema têm solução? Em virtude da maneira como o conceito de segmento foi definido, a única instância sem solução é aquela em que o vetor $A[1\,..\,n]$ é vazio. Por isso, vamos supor, implicitamente, que $n \geq 1$ no enunciado do problema.

Os exercícios abaixo propõem outras perguntas que ajudam a entender as sutilezas do problema. Todas essas perguntas poderiam ter sido feitas espontaneamente pelo leitor.

Exercícios

4.1 Que acontece se todos os elementos do vetor têm valor 1? Que acontece se todos os elementos do vetor são positivos? Que acontece se todos os elementos do vetor são negativos? Que acontece se os elementos do vetor são, alternadamente, $+1$ e -1?

4.2 Que aconteceria se permitíssemos segmentos vazios?

4.3 Digamos que um segmento $A[i\,..\,k]$ de $A[1\,..\,n]$ *tem soma maximal* se $A[i] + \cdots + A[k] \geq A[i'] + \cdots + A[k']$ sempre que $1 \leq i' \leq i \leq k \leq k' \leq n$. Mostre como encontrar um segmento de soma maximal.

4.2 Primeiro algoritmo

O algoritmo óbvio para o problema do segmento de soma máxima examina, sistematicamente, todos os segmentos de $A[1\,..\,n]$ e escolhe o que tiver maior soma.

```
ALTURAI (A, n)
1   x ← A[1]
2   para i ← 1 até n faça
3       para k ← i até n faça
4           s ← 0
5           para j ← i até k faça
6               s ← s + A[j]
7           se s > x então x ← s
8   devolva x
```

O algoritmo está correto. No início de cada execução da linha 7, s é a soma do segmento $A[i\,..\,k]$. Como i varia de 1 até n e k varia de i até n, o valor de x na linha 8 é a altura do vetor $A[1\,..\,n]$.

Desempenho. Vamos supor que cada execução de qualquer das linhas do pseudocódigo consome uma unidade de tempo. Assim, o consumo total de tempo é proporcional à soma, sobre todas as linhas, do número de execuções de cada linha.

Comecemos a análise pela linha 6, pois ela é a mais executada. Para cada valor fixo de i e k, a linha 6 é executada $k - i + 1$ vezes. Para cada i fixo, o número de execuções da linha é

$$\sum_{k=i}^{n} (k - i + 1) = 1 + 2 + 3 + \cdots + (n - i + 1) = (n - i + 1)(n - i + 2)/2\,.$$

Como i varia de 1 a n, o número total de execuções da linha é a metade de

$$\begin{aligned}\sum_{i=1}^{n} (n - i + 1)(n - i + 2) &= \sum_{m=1}^{n} m(m + 1) \\ &= \sum_{m=1}^{n} m^2 + \sum_{m=1}^{n} m \\ &= n(n + 1)(2n + 1)/6 + n(n + 1)/2 \\ &= an^3 + bn^2 + cn + d\,,\end{aligned}$$

sendo a, b, c e d constantes e $a > 0$.

Quanto às demais linhas, o número total de execuções de cada uma delas não passa de n^2. Por exemplo, o número total de execuções da linha 7 é $\sum_{i=1}^{n} (n - i + 1) = n(n + 1)/2$.

Concluímos assim que n^3 é o termo dominante no consumo de tempo total do algoritmo. Portanto, para valores grandes de n, o consumo de tempo é praticamente proporcional a n^3 no pior caso. Logo (veja a Seção 1.3), o desempenho do algoritmo está em

$$\Theta(n^3)$$

no pior caso. O algoritmo é, portanto, cúbico. (O desempenho no melhor caso também está em $\Theta(n^3)$.)

O algoritmo ALTURAI é muito ineficiente pois a soma de cada segmento é recalculada muitas vezes. Por exemplo, a soma de $A[100\,..\,200]$ é refeita durante o cálculo da soma de todos os segmentos $A[i\,..\,k]$ com $i \leq 100$ e $k \geq 200$. O algoritmo seguinte procura evitar essa fonte de ineficiência.

Exercícios

4.4 Onde o algoritmo ALTURAI usa a hipótese $n \geq 1$?

4.5 Na linha 1 do o algoritmo ALTURAI, que acontece se trocarmos "$x \leftarrow A[1]$" por "$x \leftarrow 0$" ou por "$x \leftarrow A[2]$" ou por "$x \leftarrow A[n]$"?

4.6 Prove, por indução em n, que $\sum_{m=1}^{n} m^2 = n(n + 1)(2n + 1)/6$.

4.3 Segundo algoritmo

Nosso segundo algoritmo procura não calcular a soma de alguns dos segmentos mais de uma vez:

```
ALTURAII (A, n)
1   x ← A[1]
2   para q ← 2 até n faça
3       s ← 0
4       para j ← q decrescendo até 1 faça
5           s ← s + A[j]
6           se s > x então x ← s
7   devolva x
```

O algoritmo está correto. A cada passagem pela linha 2, imediatamente antes da comparação de q com n,

$$x \text{ é a altura do vetor } A[1\,..\,q{-}1]. \tag{4.1}$$

É claro que essa propriedade vale quando $q = 2$. Suponha agora que a propriedade vale no início de uma iteração qualquer e observe que o bloco de linhas 3–6 examina todos os segmentos que terminam no índice q e atualiza o valor de x de acordo. Portanto, a propriedade continua valendo no início da iteração seguinte. Isso prova que (4.1) é um invariante.

O invariante (4.1) mostra que o algoritmo está correto, pois na última passagem pela linha 2 teremos $q = n + 1$ e então x será a altura do $A[1\,..\,n]$.

Desempenho. Suponhamos que cada execução de qualquer das linhas do pseudocódigo consome uma unidade de tempo. Então o consumo total de tempo é proporcional à soma dos números de execuções de todas as linhas.

Para cada valor fixo de q, o bloco de linhas 5–6 é executado q vezes. Como q cresce de 2 até n, o número total de execuções do bloco de linhas 5–6 é

$$\sum_{q=2}^{n} q \;=\; 2 + 3 + \cdots + n \;=\; \tfrac{1}{2}(n^2 + n - 2)\,.$$

O número total de execuções de cada uma das demais linhas não passa de n. (Por exemplo, o número total de execuções da linha 3 é $n - 1$ e o número total de execuções da linha 2 é n.) Portanto, o termo dominante no consumo total de tempo do algoritmo é n^2. Logo (veja a Seção 1.3), o desempenho no pior caso está em

$$\Theta(n^2)\,.$$

O algoritmo é, portanto, quadrático. (No melhor caso, o desempenho também está em $\Theta(n^2)$.)

O algoritmo ALTURAII é ineficiente porque a soma de alguns segmentos é refeita várias vezes. Por exemplo, a soma de $A[1\,..\,100]$ é refeita durante o cálculo das somas de $A[1\,..\,101]$, $A[1\,..\,102]$, etc. O algoritmo da próxima seção procura remediar essa ineficiência.

Exercícios

4.7 Que acontece se a linha 4 for trocada por "para $j \leftarrow 1$ até q faça"?

4.8 Modifique o algoritmo ALTURAII de modo que ele devolva um par (i, k) de índices tal que a soma $A[i] + \cdots + A[k]$ é a altura de $A[1 \mathinner{..} n]$.

4.9 Escreva um algoritmo análogo a ALTURAII para tratar da variante do problema que admite segmentos vazios. Nessa variante, um segmento $A[i \mathinner{..} k]$ é **vazio** se $i = k + 1$. A **soma** de um segmento vazio é 0.

4.4 Terceiro algoritmo: divisão e conquista

Nosso terceiro algoritmo usa o método da divisão e conquista (veja a Seção 3.4): divide a instância dada ao meio, resolve cada uma das metades e finalmente junta as duas soluções.

Para descrever o algoritmo, é preciso que o índice do primeiro elemento do vetor A seja variável. Portanto, o enunciado do problema precisa ser generalizado como segue: calcular a altura de um vetor $A[p \mathinner{..} r]$ de números inteiros.[2]

```
ALTURAIII (A, p, r)
1   se p = r
2       então devolva A[p]
3       senão q ← ⌊(p + r)/2⌋
4             x′ ← ALTURAIII (A, p, q)
5             x″ ← ALTURAIII (A, q + 1, r)
6             y′ ← s ← A[q]
7             para i ← q − 1 decrescendo até p faça
8                 s ← A[i] + s
9                 se s > y′ então y′ ← s
10            y″ ← s ← A[q + 1]
11            para j ← q + 2 até r faça
12                s ← s + A[j]
13                se s > y″ então y″ ← s
14            x ← max (x′, y′ + y″, x″)
15            devolva x
```

O algoritmo está correto. Se $p = r$, é claro que o algoritmo dá a resposta correta. Suponha agora que $p < r$. Depois da linha 4, por hipótese de indução, x' é a altura de $A[p \mathinner{..} q]$. Depois da linha 5, x'' é a altura de $A[q{+}1 \mathinner{..} r]$. Depois do bloco de linhas 6–13, $y' + y''$ é a maior soma da forma $A[i] + \cdots + A[j]$ com $i \le q < j$. (Veja o Exercício 4.10.)

Resumindo, $y' + y''$ cuida dos segmentos que contêm $A[q]$ e $A[q{+}1]$, enquanto x' e x'' cuidam de todos os demais segmentos. Concluímos assim que o número x calculado na linha 14 é a altura correta de $A[p \mathinner{..} r]$.

[2] Assim, o problema original, enunciado na Seção 4.1, é uma coleção de instâncias do problema generalizado.

p			q	$q{+}1$			r
20	−30	15	−10	30	−20	−30	30

Figura 4.2: Início da linha 14 do algoritmo ALTURAIII.
Nesse ponto, $x' = 20$, $x'' = 30$, $y' = 5$ e $y'' = 30$.

Desempenho. Seja $T(n)$ o consumo de tempo do algoritmo no pior caso quando aplicado a um vetor de tamanho $n := r - p + 1$. O bloco de linhas 6–13 examina cada elemento de $A[p\,..\,r]$ uma só vez e portanto consome tempo proporcional a n. Temos então

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n \,.$$

A parcela $T(\lceil n/2 \rceil)$ descreve o consumo da linha 4, a parcela $T(\lfloor n/2 \rfloor)$ descreve o consumo da linha 5, e a parcela n corresponde às linhas 6–15. Essa recorrência já foi discutida na Seção 3.3, onde mostramos que T está em $\Theta(n \lg n)$.

Concluímos assim que o algoritmo é linearítmico: seu desempenho no pior caso está em

$$\Theta(n \lg n) \,.$$

O algoritmo ALTURAIII é mais eficiente que ALTURAII. Mas ALTURAIII tem suas próprias ineficiências, pois refaz alguns cálculos várias vezes (por exemplo, a soma de $A[i\,..\,q]$ nas linhas 6–9 já foi calculada durante a execução da linha 4). A próxima seção procura eliminar essas ineficiências.

Exercícios

4.10 Prove que depois do bloco de linhas 6–13 do algoritmo ALTURAIII, $y' + y''$ é a maior soma dentre todas as que têm a forma $A[i] + \cdots + A[j]$ com $i \leq q < j$.

4.11 Por que não trocar "$q + 1$" por "q" nas linhas 5 e 10 do algoritmo ALTURAIII?

4.12 Mostre que o desempenho do algoritmo ALTURAIII no *melhor* caso está em $\Theta(n \lg n)$.

4.5 Quarto algoritmo: programação dinâmica

Nosso quarto algoritmo usa o método da programação dinâmica, que consiste em armazenar os resultados de cálculos intermediários numa tabela e assim evitar que eles sejam refeitos.

Para aplicar o método, será preciso introduzir o seguinte problema auxiliar, que restringe a atenção aos segmentos *terminais* do vetor: dado um vetor $A[1\,..\,q]$, encontrar a maior soma da forma

$$A[i] + \cdots + A[q] \,,$$

com $1 \le i \le q$. Para facilitar a discussão, diremos que a maior soma dessa forma é a **semialtura** do vetor $A[1\,..\,q]$. A altura do vetor $A[1\,..\,n]$ é o máximo das semialturas de $A[1\,..\,n]$, $A[1\,..\,n{-}1]$, $A[1\,..\,n{-}2]$, etc.

A semialtura de um vetor tem uma propriedade recursiva que a altura não tem: se a soma de $A[i\,..\,q]$ é a semialtura de $A[1\,..\,q]$ e $i < q$ então a soma de $A[i\,..\,q{-}1]$ é a semialtura de $A[1\,..\,q{-}1]$. (De fato, se a semialtura de $A[1\,..\,q{-}1]$ fosse maior então existiria $h \le q-1$ tal que $A[h] + \cdots + A[q{-}1] > A[i] + \cdots + A[q{-}1]$ e portanto teríamos $A[h] + \cdots + A[q] > A[i] + \cdots + A[q]$, o que é impossível.)

Se denotarmos a semialtura de $A[1\,..\,q]$ por $S[q]$, podemos resumir a propriedade recursiva por meio de uma recorrência: para qualquer vetor $A[1\,..\,q]$,

$$S[q] = \max\big(\, S[q{-}1] + A[q]\ ,\ A[q]\,\big)\,. \tag{4.2}$$

Em outras palavras, $S[q] = S[q-1] + A[q]$ a menos que $S[q-1]$ seja estritamente negativo, caso em que $S[q] = A[q]$.

A recorrência (4.2) serve de base para o seguinte algoritmo, que calcula a altura de $A[1\,..\,n]$:

```
ALTURAIV (A, n)
1  S[1] ← A[1]
2  para q ← 2 até n faça
3       se S[q − 1] ≥ 0
4            então S[q] ← S[q−1] + A[q]
5            senão S[q] ← A[q]
6  x ← S[1]
7  para q ← 2 até n faça
8       se S[q] > x então x ← S[q]
9  devolva x
```

O algoritmo está correto. A cada passagem pela linha 2, imediatamente antes que q seja comparado com n, $S[q-1]$ é a semialtura de $A[1\,..\,q{-}1]$. Mais que isso,

$$S[j] \text{ é a semialtura de } A[1\,..\,j] \tag{4.3}$$

para $j = 1, 2, \ldots, q{-}1$.

Em particular, no início da linha 6, o fato (4.3) vale para $j = 1, 2, \ldots, n$. O bloco de linhas 6–8 escolhe a maior das semialturas e portanto, no fim do algoritmo, x é a altura correta de $A[1\,..\,n]$.

Desempenho. A estimativa do consumo de tempo do algoritmo é muito fácil. Cada linha do algoritmo é executada no máximo n vezes, e portanto o algoritmo consome tempo proporcional a n. Em outras palavras, o desempenho do algoritmo no pior caso está em

$$\Theta(n)$$

e portanto o algoritmo é linear. (O desempenho no melhor caso também está em $\Theta(n)$.)

	p							r
A	20	−30	15	−10	30	−20	−30	30
S	20	−10	15	5	35	15	−15	30

Figura 4.3: Vetores A e S no fim da execução do algoritmo ALTURAIV.

Exercícios

4.13 Os dois processos iterativos de ALTURAIV podem ser fundidos num só, evitando-se assim o uso do vetor $S[1 \mathinner{..} n]$. Uma variável s pode fazer o papel de um elemento genérico do vetor S. Implemente essa ideia.

4.14 Suponha dada uma matriz quadrada $A[1 \mathinner{..} n, 1 \mathinner{..} n]$ de números inteiros (nem todos positivos). Encontrar uma submatriz quadrada de A que tenha soma máxima. (Veja o livro de Bentley [1, p.84].)

4.15 Na definição de semialtura, todos os segmentos $A[i \mathinner{..} q]$ são não vazios. Onde essa observação é usado no algoritmo ALTURAIV?

4.6 Observações sobre programação dinâmica

O algoritmo ALTURAIV é um exemplo de **programação dinâmica**.[3] O método só se aplica a problemas dotados de estrutura recursiva: qualquer solução de uma instância deve conter soluções de instâncias menores. Assim, o ponto de partida de qualquer algoritmo de programação dinâmica é uma recorrência.

A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias da instância original. (No nosso caso, trata-se da tabela $S[1 \mathinner{..} n]$.) O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

Os Capítulos 8 e 9 exibem outros exemplos de programação dinâmica.

Notas bibliográficas

Este capítulo foi baseado no livro de Bentley [1].

[3] Aqui, a palavra *programação* não tem relação direta com programação de computadores. Ela significa *planejamento* e refere-se à construção da tabela que armazena os resultados intermediários.

Capítulo 5

O problema da maioria

Imagine uma eleição com muitos candidatos e muitos eleitores. Queremos determinar, no menor tempo possível, se algum dos candidatos obteve a maioria dos votos.

D	C	A	B	C	B	B	C	B	B	B	B
D	C	A	B	C	B	B	C	B	B	B	F

Figura 5.1: Dois exemplos com 12 eleitores e candidatos A, B, $\ldots$, Z. No primeiro exemplo, B tem maioria. No segundo, nenhum candidato tem maioria.

Este capítulo discute um algoritmo surpreendentemente rápido para o problema. A análise de desempenho do algoritmo é muito fácil, mas a prova da correção é sutil e depende de um invariante não óbvio.

5.1 O problema

Seja $A[1\,..\,n]$ um vetor de objetos de algum tipo, como números naturais, cadeias de caracteres, ou figuras, por exemplo. (Você pode imaginar que os índices 1, 2, $\ldots$, n são os eleitores e o eleitor i vota no candidato $A[i]$.) Sobre os objetos que compõem o vetor, vamos supor que podemos decidir se dois objetos são iguais ou diferentes, mas as relações "maior que" e "menor que" entre objetos não estão necessariamente definidas.

Diremos que um objeto x **é majoritário** se o número de ocorrências de x em $A[1\,..\,n]$ é (estritamente) maior que $n/2$. Em outras palavras, x é majoritário se mais da metade[1] dos elementos do vetor é igual a x, ou seja, se o número de elementos iguais a x é maior que o número de elementos diferentes de x.

> *Problema da maioria:* Encontrar um elemento majoritário em um dado vetor $A[1\,..\,n]$ de objetos.

[1] A popular expressão "metade dos votos mais um" está errada pois a metade de um número ímpar não é um número inteiro.

Nem todo vetor tem um elemento majoritário. Mas se um elemento majoritário existe ele é único. Assim, toda instância do problema tem uma única solução ou não tem solução alguma. Faz parte do problema decidir se a instância dada tem solução.

Algoritmos óbvios. Adotaremos n como tamanho da instância $A[1 \mathinner{..} n]$ do problema. Com essa definição, há um algoritmo quadrático óbvio para o problema: basta comparar cada elemento do vetor com todos os demais.

Se a relação "menor que" estiver definida entre os elementos de $A[1 \mathinner{..} n]$, podemos ordenar o vetor e depois contar o número de ocorrências de cada objeto. Esse algoritmo é linearítmico: a ordenação é linearítmica (veja o algoritmo MERGESORT no Capítulo 3) e as contagens no vetor ordenado consomem tempo linear.

À primeira vista, parece ímpossível resolver o problema em menos tempo. Mas existe um algoritmo que resolve o problema em tempo linear, como mostraremos a seguir. (Em termos do exemplo na introdução do capítulo, o algoritmo determina o vencedor da eleição em tempo proporcional ao número de eleitores.)

Exercícios

5.1 Escreva um algoritmo que verifique se um dado objeto x é ou não majoritário em $A[1 \mathinner{..} n]$.

5.2 Suponha que n é par e um objeto x ocorre exatamente $n/2$ vezes em $A[1 \mathinner{..} n]$. Mostre que o vetor não tem objeto majoritário.

5.3 Escreva o algoritmo linearítmico sugerido acima supondo que $A[1 \mathinner{..} n]$ é um vetor de inteiros.

5.2 A estrutura recursiva do problema

No que segue, o conceito de objeto majoritária será aplicado a subvetores arbitrários de $A[1 \mathinner{..} n]$. Assim, a expressão "x é majoritário em $A[i \mathinner{..} j]$" significa que o número de ocorrências de x em $A[i \mathinner{..} j]$ é maior que $(j - i + 1)/2$.

Para resolver o problema da maioria em tempo linear, é preciso entender a seguinte propriedade, que descreve a estrutura recursiva do problema:

> Se x é um objeto majoritário em $A[1 \mathinner{..} n]$ então, para qualquer k entre 1 e $n{+}1$, x é majoritário em $A[1 \mathinner{..} k{-}1]$ ou em $A[k \mathinner{..} n]$ ou em ambos. (5.1)

Prova: Sejam p e q os números de ocorrências de x em $A[1 \mathinner{..} k{-}1]$ e $A[k \mathinner{..} n]$ respectivamente. Se x não é majoritário em nenhum desses dois subvetores então $p \le (k-1)/2$ e $q \le (n-k+1)/2$. Portanto, o número de ocorrências de x em $A[1 \mathinner{..} n]$ é $p + q \le (k - 1 + n - k + 1)/2 = n/2$. Logo, x não é majoritário em $A[1 \mathinner{..} n]$, como queríamos provar.

É importante observar que a recíproca da propriedade (5.1) não é verdadeira: é possível que x seja majoritário em $A[1 \mathinner{..} k{-}1]$ ou em $A[k \mathinner{..} n]$ mas não seja majoritário em $A[1 \mathinner{..} n]$.

Exercícios

5.4 Mostre que a recíproca da propriedade (5.1) não é verdadeira.

5.5 Escreva um algoritmo (linearítmico) de divisão e conquista para o problema da maioria baseado na propriedade (5.1).

5.3 Um algoritmo linear

O seguinte algoritmo resolve as instâncias não vazias do problema da maioria em tempo linear: ele recebe um vetor $A[1\,..\,n]$ de objetos, com $n \geq 1$, e devolve o objeto majoritário se tal existir; senão, devolve um objeto NIL que não possa ser confundido com um elemento do vetor.

```
MAIORIA (A, n)
1   x ← A[1]
2   v ← 1
3   k ← 1
4   para m ← 2 até n faça
5       se v ≥ 1
6           então  se A[m] = x
7                      então v ← v + 1
8                      senão v ← v − 1
9           senão  x ← A[m]
10                 v ← 1
11                 k ← m
12  c ← 0
13  para m ← 1 até n faça
14      se A[m] = x
15          então c ← c + 1
16  se c > n/2
17      então devolva x
18      senão devolva NIL
```

(A variável k é supérflua, mas vai nos ajudar a mostrar que o algoritmo está correto.) O algoritmo tem duas fases. A primeira (linhas 1–11) encontra um bom candidato a solução x e a segunda (linhas 12–18) verifica se x é, de fato, solução.[2] Um objeto x é um **bom candidato** se tem a seguinte propriedade: ou x é majoritário em $A[1\,..\,n]$ ou o vetor não tem elemento majoritário algum.

O algoritmo está correto. Para mostrar que o algoritmo está correto, observe as seguintes propriedades invariantes: a cada passagem pela linha 4, imediatamente antes da comparação de m com n,

[2] A segunda fase é necessária porque a recíproca da propriedade (5.1) não é verdadeira.

i. v é a vantagem de x em $A[k\,..\,m{-}1]$,

ii. $v \geq 0$ e

iii. $A[1\,..\,k{-}1]$ não tem elemento majoritário.

A **vantagem** de um objeto x **em** um vetor $A[i\,..\,j]$ é a diferença entre o número de elementos de $A[i\,..\,j]$ iguais a x e o número de elementos de $A[i\,..\,j]$ diferentes de x. (O problema da maioria poderia ser formulado assim: encontrar um objeto cuja vantagem em $A[1\,..\,n]$ é estritamente positiva.)

Prova dos invariantes: A validade de ii é evidente. A validade de i e iii exige verificação mais cuidadosa. No início da primeira iteração, é claro que as propriedades i e iii estão satisfeitas. Suponha agora que i e iii valem no início de uma iteração qualquer que não a última. Se $v \geq 1$ então é fácil deduzir que i e iii valem no início da próxima iteração quer $A[m]$ seja igual a x, quer seja diferente (note que o valor de k não se altera nesse caso).

Suponha agora que $v = 0$. De acordo com i, a vantagem de x em $A[k\,..\,m{-}1]$ é nula e portanto não existe objeto majoritário em $A[k\,..\,m{-}1]$. De acordo com iii, $A[1\,..\,k{-}1]$ também não tem objeto majoritário. A propriedade (5.1) garante então que

$A[1\,..\,m{-}1]$ não tem elemento majoritário.

Além disso, no fim da iteração corrente, depois que x, v e k assumem os valores $A[m]$, 1 e m respectivamente (e antes que m seja incrementado), é evidente que a vantagem de x em $A[k\,..\,m]$ é v. Portanto, no início da próxima iteração (depois que m for incrementado), será verdade que

$A[1\,..\,m{-}2]$ não tem elemento majoritário e
a vantagem de x em $A[k\,..\,m{-}1]$ é v.

Assim, as propriedades i e iii valerão no início da próxima iteração. Isso encerra a prova dos invariantes.

Resta mostrar como os invariantes i a iii garantem que o algoritmo está correto. No início da segunda fase do algoritmo, na linha 12, temos $m = n + 1$ e portanto

$A[1\,..\,k{-}1]$ não tem elemento majoritário e
pelo menos metade dos elementos de $A[k\,..\,n]$ é igual a x.

Suponha agora que $A[1\,..\,n]$ tem um elemento majoritário, digamos a. Como $A[1\,..\,k{-}1]$ não tem elemento majoritário, a propriedade (5.1) garante que a é majoritário em $A[k\,..\,n]$. Como metade dos elementos de $A[k\,..\,n]$ é igual a x, temos necessariamente $a = x$. Isso mostra que x é um bom candidato. As linhas 12–15 do algoritmo não fazem mais que verificar se x é majoritário em $A[1\,..\,n]$.

Desempenho. A análise de desempenho do algoritmo MAIORIA é muito simples. Suponhamos que uma execução de qualquer das linhas do pseudocódigo consome 1 unidade de tempo; assim, podemos estimar o consumo de tempo total contando o número de execuções das várias linhas.

O bloco de linhas 5–11 é executado $n-1$ vezes. O bloco de linhas 14–15 é executado n vezes. As linhas 1 a 3 e o bloco de linhas 16–18 é executado uma só vez. Portanto, o

algoritmo consome

$$\Theta(n)$$

unidades de tempo no pior caso. Portanto, o algoritmo é linear. (O consumo de tempo no melhor caso também está em $\Theta(n)$.)

Exercícios

5.6 Prove a seguinte afirmação: a vantagem de x em $A[i \,..\, j]$ é v se e somente se o número de ocorrências de x em $A[i \,..\, j]$ é exatamente $(j - i + 1 + v)/2$.

Notas bibliográficas

Este capítulo foi baseado nos livros de Bentley [1] e Manber [14].

Capítulo 6

Multiplicação de grandes números

Quanto tempo é necessário para multiplicar dois números naturais? Se os números têm m e n dígitos respectivamente, o algoritmo usual consome tempo proporcional a mn. Se $m = n$, por exemplo, o consumo de tempo é proporcional a n^2. Provavelmente não existe algoritmo mais rápido se m e n forem pequenos (menores que duas ou três centenas, digamos). Mas existe um algoritmo que é bem mais rápido quando m e n são grandes (como acontece em criptografia, por exemplo).

```
    9999    A
    7777    B
--------
   69993    C
  69993     D
 69993      E
69993       F
--------
77762223    G
```

Figura 6.1: Algoritmo usual de multiplicação de dois números naturais. Aqui estamos multiplicando 9999 por 7777 para obter o produto 77762223. A linha C é o resultado da multiplicação da linha A pelo dígito menos significativo da linha B. As linhas D, E e F são definidas de maneira semelhante. A linha G é o resultado da soma das linhas C a F.

Este capítulo ensina algumas lições de caráter geral: (1) para muitos problemas, existem algoritmos surpreendentes mais rápidos que o algoritmo usual; (2) às vezes, a superioridade do algoritmo mais sofisticado só se manifesta nas instâncias muito grandes do problema; (3) a técnica de resolução de recorrências é capaz de fazer uma análise muito precisa do desempenho de algoritmos recursivos.

6.1 O problema

Usaremos notação decimal para representar números naturais. (Poderíamos igualmente bem usar notação binária, ou notação base 2^{32}, por exemplo.) Diremos que um **dígito** é qualquer número menor que 10. Diremos que um número natural u **tem** n **dígitos**[1] se $u < 10^n$. Com esta convenção, podemos nos restringir, sem perder generalidade, ao caso em que os dois multiplicandos têm o mesmo número de dígitos:

> *Problema da multiplicação:* Dados números naturais u e v com n dígitos cada, calcular o produto $u \times v$.

Você deve imaginar que cada número natural é representado por um vetor de dígitos. Mas nossa discussão será conduzida num nível de abstração alto, sem manipulação explícita desse vetor. (Veja a Seção 6.5.)

6.2 O algoritmo usual

Preliminarmente, considere a operação de *adição*. Sejam u e v dois números naturais com n dígitos cada. A soma $u + v$ tem $n + 1$ dígitos e o algoritmo usual de adição calcula $u + v$ em tempo proporcional a n.

Considere agora o algoritmo usual de *multiplicação* de dois números u e v com n dígitos cada. O algoritmo tem dois passos. O primeiro passo calcula todos os n produtos de u por um dígito de v (cada um desses produtos tem $n+1$ dígitos). O segundo passo do algoritmo desloca para a esquerda, de um número apropriado de casas, cada um dos n produtos obtidos no primeiro passo e soma os n números resultantes. O produto $u \times v$ tem $2n$ dígitos.

O primeiro passo do algoritmo consome tempo proporcional a n^2. O segundo passo também consome tempo proporcional a n^2. Assim, o consumo de tempo do algoritmo usual de multiplicação está em $\Theta(n^2)$. O algoritmo é, portanto, quadrático.

Exercícios

6.1 Descreva o algoritmo usual de adição em pseudocódigo. O algoritmo deve receber números u e v com n dígitos cada e devolver a soma $u + v$.

6.2 Descreva o algoritmo usual de multiplicação em pseudocódigo. O algoritmo deve receber números u e v com n dígitos cada e devolver o produto $u \times v$.

6.3 Observações sobre o método de divisão e conquista

Sejam u e v dois números com n dígitos cada. Suponha, por enquanto, que n é par. Seja p o número formado pelos $n/2$ primeiros dígitos de u e seja q o número formado pelos

[1] Teria sido melhor dizer "tem no máximo n dígitos".

$n/2$ últimos dígitos de u. Assim,

$$u = p \times 10^{n/2} + q\,.$$

Defina r e s analogamente para v, de modo que $v = r \times 10^{n/2} + s$. Teremos então

$$u \times v \;=\; p \times r \times 10^{n} + (p \times s + q \times r) \times 10^{n/2} + q \times s\,. \tag{6.1}$$

Esta expressão reduz a multiplicação de dois números com n dígitos cada a quatro multiplicações (a saber, p por r, p por s, q por r e q por s) de números com $n/2$ dígitos cada.[2]

Se n for uma potência de 2, o truque pode ser aplicado recursivamente a cada uma das quatro multiplicações menores. O resultado é o seguinte algoritmo recursivo, que recebe números naturais u e v com n dígitos cada e devolve o produto $u \times v$:

```
RASCUNHO (u, v, n)    ▷ n é potência de 2
1   se n = 1
2      então devolva u × v
3      senão m ← n/2
4            p ← ⌊u/10^m⌋
5            q ← u mod 10^m
6            r ← ⌊v/10^m⌋
7            s ← v mod 10^m
8            pr ← RASCUNHO (p, r, m)
9            qs ← RASCUNHO (q, s, m)
10           ps ← RASCUNHO (p, s, m)
11           qr ← RASCUNHO (q, r, m)
12           uv ← pr × 10^(2m) + (ps + qr) × 10^m + qs
13           devolva uv
```

O algoritmo está correto. É claro que o algoritmo produz o resultado correto se $n = 1$. Agora tome $n \geq 2$. Como $m < n$, podemos supor, por hipótese de indução, que a linha 8 produz o resultado correto, ou seja, que $pr = p \times r$. Analogamente, podemos supor que $qs = q \times s$, $ps = p \times s$ e $qr = q \times r$ no início da linha 12. A linha 12 não faz mais que implementar (6.1). Portanto, $uv = u \times v$.

Desempenho. As linhas 4 a 7 envolvem apenas o deslocamento de casas decimais, podendo portanto ser executadas em não mais que n unidades de tempo. As multiplicações por 10^{2m} e 10^{m} na linha 12 também consomem no máximo n unidades de tempo, pois podem ser executadas com um mero deslocamento de casas decimais. As adições na linha 12 consomem tempo proporcional ao número de dígitos de uv, igual a $2n$.

[2] Minha calculadora de bolso não é capaz de exibir/armazenar números que tenham mais que n dígitos. Para calcular o produto de dois números com n dígitos cada, posso recorrer à equação (6.1): uso a máquina para calcular os quatro produtos e depois uso lápis e papel para fazer as três adições.

Portanto, se denotarmos por $T(n)$ o consumo de tempo do algoritmo no pior caso, teremos

$$T(n) = 4\,T(n/2) + n\,. \tag{6.2}$$

As quatro parcelas $T(n/2)$ referem-se às linhas 8 a 11 e a parcela n refere-se ao consumo das demais linhas. Segue daí (exercício 6.3) que

$$T(n) \text{ está em } \Theta(n^2).$$

Assim, o algoritmo RASCUNHO não é mais eficiente que o algoritmo usual de multiplicação.

99998888	u
77776666	v
9999	p
8888	q
7777	r
6666	s
77762223	pr
59247408	qs
135775310	$ps + qr$
7777580112347408	uv

Figura 6.2: Multiplicação de u por v calculada pelo algoritmo RASCUNHO. Neste exemplo temos $n = 8$. O resultado da operação é uv.

Exercícios

6.3 Mostre que a solução $T(n)$ da recorrência (6.2) está em $\Theta(n^2)$. (Sugestão: veja a seção 3.3.)

6.4 Algoritmo de Karatsuba

Para tornar o algoritmo da seção anterior muito mais rápido, basta observar que os três números de que precisamos do lado direito de (6.1) — a saber, $p \times r$, $(p \times s + q \times r)$ e $q \times s$ — podem ser obtidos com apenas *três* multiplicações. De fato,

$$p \times s + q \times r \;=\; y - p \times r - q \times s\,,$$

sendo $y = (p + q) \times (r + s)$. Assim, a equação (6.1) pode ser substituída por

$$u \times v \;=\; p \times r \times 10^n + (y - p \times r - q \times s) \times 10^{n/2} + q \times s\,. \tag{6.3}$$

(É bem verdade que (6.3) envolve duas adições e duas subtrações adicionais, mas essas operações consomem muito menos tempo que as multiplicações.) Se n não é par, basta trocar $n/2$ por $\lceil n/2 \rceil$: teremos $u = p \times 10^{\lceil n/2 \rceil} + q$ e $v = r \times 10^{\lceil n/2 \rceil} + s$ e portanto

$$u \times v \;=\; p \times r \times 10^{2\lceil n/2 \rceil} + (y - p \times r - q \times s) \times 10^{\lceil n/2 \rceil} + q \times s\,.$$

Essa é a ideia do algoritmo proposto por A. Karatsuba. Ele recebe números naturais u e v com n dígitos cada e devolve o produto $u \times v$:

KARATSUBA $(u,\ v,\ n)$
1 se $n \le 3$
2 então devolva $u \times v$
3 senão $m \leftarrow \lceil n/2 \rceil$
4 $p \leftarrow \lfloor u/10^m \rfloor$
5 $q \leftarrow u \bmod 10^m$
6 $r \leftarrow \lfloor v/10^m \rfloor$
7 $s \leftarrow v \bmod 10^m$
8 $pr \leftarrow$ KARATSUBA $(p,\ r,\ m)$
9 $qs \leftarrow$ KARATSUBA $(q,\ s,\ m)$
10 $y \leftarrow$ KARATSUBA $(p+q,\ r+s,\ m{+}1)$
11 $uv \leftarrow pr \times 10^{2m} + (y - pr - qs) \times 10^m + qs$
12 devolva uv

O algoritmo está correto. A prova da correção do algoritmo é análoga à prova da correção do algoritmo RASCUNHO. As instâncias em que n vale 1, 2 ou 3 devem ser tratadas na base da recursão porque o algoritmo é aplicado, na linha 10, a uma instância de tamanho $\lceil n/2 \rceil + 1$, e este número só é menor que n quando $n > 3$.

999988888	u
077766666	v
07769223	pr
5925807408	qs
98887	$p+q$
67443	$r+s$
6669235941	y
735659310	$y - pr - qs$
077765801856807408	uv

Figura 6.3: Multiplicação de u por v calculada pelo algoritmo KARATSUBA. O resultado da operação é uv. Se u e v tivessem n dígitos cada, pr e qs teriam $n+1$ dígitos cada, y teria $n+2$ (mais precisamente, só $n+1$) dígitos e uv teria $2n$ dígitos.

Desempenho. Seja $T(n)$ o consumo de tempo do algoritmo KARATSUBA no pior caso. Então

$$T(n) = 2\,T(\lceil n/2\rceil) + T(\lceil n/2\rceil + 1) + n\,. \tag{6.4}$$

As duas parcelas $T(\lceil n/2\rceil)$ e a parcela $T(\lceil n/2\rceil+1)$ se referem às linhas 8, 9 e 10 respectivamente. A parcela n representa o consumo de tempo das demais linhas. Se tomarmos $T(1) = 1$, $T(2) = 2$ e $T(3) = 3$, teremos $T(4) = 11$, $T(5) = 22$, etc.

Felizmente não precisamos de uma solução exata da recorrência (6.4). Para obter uma solução aproximada, comecemos por restringir n às potências de 2 (o que elimina os "$\lfloor\ \rfloor$" e "$\lceil\ \rceil$") e a remover o incômodo "+1". Assim, obtemos a recorrência

$$T(N) = 3\,T(N/2) + N \tag{6.5}$$

para $N = 2^1, 2^2, 2^3, \ldots$ Essa recorrência pode ser "desenrolada" da seguinte maneira:

$$\begin{aligned}
T(N) &= 3\,T(N/2) + N \\
&= 3\,(3\,T(N/4) + N/2) + N \\
&= 3^2\,T(N/2^2) + 3N/2 + N \\
&= 3^3\,T(N/2^3) + 3^2N/2^2 + 3^1N/2^1 + 3^0N/2^0 \\
&= 3^j\,T(N/2^j) + 3^{j-1}N/2^{j-1} + 3^{j-2}N/2^{j-2} + \cdots + 3^0N/2^0 \\
&= 3^j\,(1) + 3^{j-1}N/2^{j-1} + 3^{j-2}N/2^{j-2} + \cdots + 3^0N/2^0 && (6.6)\\
&= 3^jN/2^j + 3^{j-1}N/2^{j-1} + 3^{j-2}N/2^{j-2} + \cdots + 3^0N/2^0 && (6.7)\\
&= N\,\big((3/2)^j + (3/2)^{j-1} + (3/2)^{j-2} + \cdots + (3/2)^0\big) \\
&= N\,\big((3/2)^{j+1} - 1\big)/\big((3/2) - 1\big) && (6.8)\\
&= 2N\,((3/2)^{j+1} - 1) \\
&= 3^{j+1} - 2N \\
&= 3N^{\lg 3} - 2N && (6.9)
\end{aligned}$$

Na linha (6.6), tomamos $j = \lg N$ e $T(1) = 1$. Na linha (6.7), lembramos que $2^j = N$. Na linha (6.8), usamos a fórmula da soma de uma progressão geométrica (exercício 6.4). Na linha (6.9), observamos que

$$3^j = (2^{\lg 3})^j = (2^j)^{\lg 3} = N^{\lg 3}\,.$$

O número $\lg 3$ fica entre 1.584 e 1.585 e portanto $N^{\lg 3}$ fica entre $N\sqrt{N}$ e N^2.

É interessante conferir (6.9) por indução em N. A fórmula está evidentemente correta quando $N = 1$. Suponha agora que $N \geq 2$ e que a fórmula está correta se trocarmos "N" por "$N/2$". Então,

$$T(N) = 3\big(3(\tfrac{N}{2})^{\lg 3} - 2\tfrac{N}{2}\big) + N = 3^2\,\frac{N^{\lg 3}}{3} - 2N = 3N^{\lg 3} - 2N,$$

confirmando (6.9).

Embora a fórmula $3N^{\lg 3} - 2N$ esteja correta apenas quando N é potência de 2, a experiência mostra que o termo dominante $N^{\lg 3}$ descreve corretamente o comportamento

assintótico de $T(N)$ para todos os demais valores de N (exceto os muito pequenos). Podemos, então, aceitar como fato que a solução $T(n)$ da recorrência (6.4) está em $\Theta(n^{\lg 3})$. (Veja o exercício 6.5.) Portanto, o consumo de tempo do algoritmo KARATSUBA no pior caso está em

$$\Theta(n^{\lg 3}) .$$

Como $\lg 3 < 1.6$, o algoritmo KARATSUBA é assintoticamente mais rápido que o algoritmo quadrático usual. Mas, em virtude da constante multiplicativa escondida sob a notação Θ, essa superioridade só se manifesta quando n é maior que algumas centenas.

Exercícios

6.4 Verifique, por indução em j, a fórmula da soma de uma progressão geométrica: $r^0 + r^1 + \cdots + r^j = (r^{j+1} - 1)/(r - 1)$.

6.5 Mostre que a solução $T(n)$ da recorrência (6.4) está em $\Theta(n^{\lg 3})$. (Sugestão: inspire-se na Seção 3.3.) Siga o seguinte roteiro: (1) Mostre, por indução em n, que $T(n)$ é crescente, ou seja, que $T(n) \leq T(n+1)$ para todo n. (2) Mostre, a partir de (1) e (6.9), que $T(n) \leq 9n^{\lg 3}$ para todo $n \geq 2$. (3) Mostre, a partir de (1) e (6.9), que $T(n) \geq \frac{1}{3}\, n^{\lg 3}$ para todo $n \geq 2$. (Esse método de cálculo da classe assintótica de uma recorrência poderia ser chamado "método da interpolação".)

6.6 Compare os valores de $n\sqrt{n}$, $n^{\lg 3}$ e n^2 para $n = 100$ e 1000.

6.5 Detalhes de implementação

Para implementar os algoritmos que discutimos acima, é preciso representar cada número por um vetor de dígitos, ou seja, um vetor $U[1\mathinner{..}n]$ cujos componentes pertencem ao conjunto $\{0, 1, \ldots, 9\}$. Um tal vetor representa o número natural $U[n] \times 10^{n-1} + U[n{-}1] \times 10^{n-2} + \cdots + U[2] \times 10 + U[1]$.

9	8	7	6	5	4	3	2	1
9	9	9	9	8	8	8	8	8

Figura 6.4: Vetor $U[1\mathinner{..}9]$ que representa o número 999988888.

O algoritmo KARATSUBA recebe os vetores $U[1\mathinner{..}n]$ e $V[1\mathinner{..}n]$ que representam os números u e v respectivamente e devolve o vetor $X[1\mathinner{..}2n]$ que representa uv. Nas linhas 4 e 5 do algoritmo, p é representado por $U[m{+}1\mathinner{..}n]$ e q é representado por $U[1\mathinner{..}m]$. A implementação das linhas 6 e 7 é análoga. Nas linhas 8 e 9, pr é representado por um vetor $PR[1\mathinner{..}2m]$ e qs é representado por um vetor $QS[1\mathinner{..}2m]$.

Na linha 10, para calcular $p + q$ basta submeter $U[m{+}1\mathinner{..}n]$ e $U[1\mathinner{..}m]$ ao algoritmo usual de adição, que produz um vetor $A[1\mathinner{..}m{+}1]$. Algo análogo vale para a subexpressão $r + s$. O número y é representado por um vetor $Y[1\mathinner{..}2m{+}1]$.

Na linha 11, o valor de $y - pr - qs$ é calculado pelo algoritmo usual de subtração (não é preciso cuidar de números negativos pois $y - pr - qs \geq 0$). Para calcular o valor da expressão $pr \times 10^{2m} + (y - pr - qs) \times 10^m + qs$, basta concatenar os vetores $PR[1\,..\,2m]$ e $QS[1\,..\,2m]$ que representam pr e qs respectivamente e somar o resultado com $y - pr - qs$ (deslocado de m posições) usando o algoritmo usual de adição.

6.6 Observações sobre divisão e conquista

O algoritmo KARATSUBA é um bom exemplo do método da divisão e conquista, que já encontramos nos Capítulos 3 e 4. O segredo do sucesso do método neste caso está no fato de que o processo de divisão da instância original (linhas 3 a 7 do algoritmo) e o processo de combinação das soluções das instâncias menores (linha 11) consomem relativamente pouco tempo.

Notas bibliográficas

Este capítulo foi baseada na Seção 7.1 do livro de Brassard e Bratley [2]. O assunto também é discutido nos livros de Knuth [11], Dasgupta, Papadimitriou e Vazirani [4] e Kleinberg e Tardos [10].

O algoritmo KARATSUBA foi publicado em 1962 pelos matemáticos russos Anatolii Karatsuba e Yurii Ofman. Veja o verbetes *Multiplication algorithm* e *Karatsuba algorithm* na Wikipedia.

O célebre algoritmo de Strassen para multiplicação de matrizes usa as mesmas ideias básicas do algoritmo de Karatsuba.

Capítulo 7

Intervalos disjuntos

Imagine que vários eventos (feiras, congressos, etc.) querem usar um certo centro de convenções. Cada evento gostaria de usar o centro durante um intervalo de tempo que começa num dia a e termina num dia b. Dois eventos são compatíveis se os seus intervalos de tempo são disjuntos. A administração do centro quer atender o maior número possível de eventos que sejam mutuamente compatíveis.

A figura 7.1 mostra um exemplo com oito eventos. Há três eventos mutuamente compatíveis, mas não há quatro.

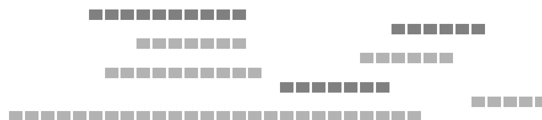

Figura 7.1: Cada linha horizontal representa o intervalo de tempo de um evento (o menor tem 5 dias e o maior tem 26 dias). A cor mais escura identifica um conjunto de eventos mutuamente compatíveis.

Este capítulo ilustra várias ideias e conceitos importantes: algoritmos gulosos, pré-processamento, descrição de algoritmos em nível alto de abstração.

7.1 Formalização do problema

Um **intervalo** é um par (a, b) de números naturais tal que $a \leq b$. Esse intervalo representa o conjunto $\{a, a+1, \ldots, b-1, b\}$. Diremos que a é a **origem** e b o **término** do intervalo.

A origem e o término de um intervalo s serão denotados por $a(s)$ e $b(s)$ respectivamente. Dois intervalos s e s' são **compatíveis** se forem disjuntos, ou seja, se

$$b(s) < a(s') \quad \text{ou} \quad b(s') < a(s)\,.$$

Diremos que um conjunto de intervalos é **viável** se os intervalos são compatíveis dois a dois.

Problema dos intervalos disjuntos: Dado um conjunto S de intervalos, encontrar um subconjunto viável máximo de S.

Um subconjunto viável X de S é **máximo** se não existe outro maior, ou seja, se $|X| \geq |X'|$ para todo subconjunto viável X' de S.

Exercícios

7.1 Qual a solução das instâncias do problema dos intervalos disjuntos em que o conjunto S de intervalos é vazio ou unitário?

7.2 Verifique que dois intervalos s e s' são incompatíveis se e somente se $b(s) \geq a(s')$ e $b(s') \geq a(s)$. Mostre também que s e s' são incompatíveis se e somente se existe um número t tal que $a(s) \leq t \leq b(s)$ e $a(s') \leq t \leq b(s')$.

7.3 Considere o seguinte algoritmo iterativo: enquanto o conjunto S de intervalos não estiver vazio, escolha qualquer intervalo s em S, elimine de S todos os intervalos incompatíveis com s, e elimine s de S. Descreva esse algoritmo em pseudocódigo. O algoritmo resolve o problema dos intervalos disjuntos?

7.4 Suponha que um subconjunto viável X de S tem a seguinte propriedade: nenhum dos superconjuntos próprios de X é viável. É verdade que X é um subconjunto viável máximo?

7.5 É verdade que todo subconjunto viável máximo de um conjunto de intervalos contém um intervalo que minimiza a diferença $b - a$? É verdade que contém um intervalo de término mínimo? É verdade que contém um intervalo de origem mínima?

7.2 A propriedade gulosa do problema

O problema dos intervalos disjuntos tem uma propriedade fundamental que chamaremos *gulosa* (pelas razões indicadas abaixo). Para discutir a propriedade, precisamos do seguinte conceito: um intervalo z tem *término mínimo em* um conjunto Z de intervalos se $z \in Z$ e $b(z) \leq b(z')$ para todo z' em Z. Eis a propriedade:

Todo intervalo de término mínimo num conjunto S de intervalos pertence a algum subconjunto viável máximo de S.

Prova da propriedade: Seja y um intervalo de término mínimo em S, seja Z um subconjunto viável máximo de S, e seja z um intervalo de término mínimo em Z. É claro que $b(y) \leq b(z)$ e $b(z) < a(z')$ para todo z' em $Z - \{z\}$. Logo, y é compatível com todos os elementos de $Z - \{z\}$ e portanto o conjunto

$$Y := (Z - \{z\}) \cup \{y\}$$

é viável. Suponha agora, por um momento, que y está em $Z - \{z\}$. Então, por um lado, $b(y) = b(z)$, e, por outro, $b(y) < a(z) \leq b(z)$ pois y e z são compatíves. Essa contradição mostra que y não está em $Z - \{z\}$. Concluímos assim que Y tem a mesma cardinalidade que Z e portanto é um subconjunto viável máximo de S. Isso prova a propriedade, uma vez que Y contém y.

7.3 Um algoritmo guloso

A propriedade gulosa do problema dos intervalos disjuntos leva a um algoritmo surpreendentemente simples:

```
INT-GULOSOR (S)
1  se S = ∅
2     então devolva ∅
3     senão seja y um intervalo de término mínimo em S
4           S' ← {s ∈ S : a(s) > b(y)}
5           X' ← INT-GULOSOR (S')
6           devolva X' ∪ {y}
```

Esse algoritmo é *guloso* porque escolhe, recursivamente, os intervalos mais "apetitosos" sem parecer se importar com o objetivo maior de maximizar um subconjunto viável.

O algoritmo está correto. É claro que o algoritmo produz a solução correta quando S é vazio. Suponha agora que S não é vazio e que o algoritmo produz uma solução correta ao receber qualquer subconjunto próprio de S. Como S' é um subconjunto próprio de S (pois não contém y), podemos supor que, no fim da linha 5, X' é um subconjunto viável máximo de S'.

Todos os intervalos em S' são compatíveis com y. Assim, $X' \cup \{y\}$ é viável. Para mostrar que esse conjunto é máximo em S, basta mostrar que $|X' \cup \{y\}| \geq |Y|$ para algum subconjunto viável máximo Y de S.

De acordo com a propriedade gulosa, podemos supor que Y contém y. Como veremos a seguir,

$$Y - \{y\} \subseteq S'. \tag{7.1}$$

Para mostrar isso, tomemos um elemento arbitrário s de $Y - \{y\}$ e verifiquemos que s está em S'. Como y e s são compatíveis (uma vez que ambos pertencem ao conjunto viável Y) temos $b(s) < a(y)$ ou $b(y) < a(s)$. A primeira alternativa não vale pois $a(y) \leq b(y)$ e $b(y) \leq b(s)$ em virtude da maneira como y foi escolhido. Portanto, vale a segunda alternativa, ou seja, $b(y) < a(s)$. Isso mostra que $s \in S'$ e assim encerra a prova de (7.1).

Segue daí e da maximalidade de X' em S' que $|Y - \{y\}| \leq |X'|$. Logo, $|Y| \leq |X' \cup \{y\}|$, como queríamos mostrar. Portanto, $X' \cup \{y\}$ é, de fato, um subconjunto viável máximo de S.

Versão iterativa. O algoritmo recursivo INT-GULOSOR pode ser reescrito em modo iterativo como segue:

INT-GULOSO (S)
0 $S' \leftarrow S$
1 $X \leftarrow \emptyset$
2 enquanto $S' \neq \emptyset$ faça
3 seja y um intervalo de término mínimo em S'
4 $X \leftarrow X \cup \{y\}$
5 $S' \leftarrow \{s \in S' : a(s) > b(y)\}$
6 devolva X

O algoritmo INT-GULOSO está correto porque não passa de uma reformulação de INT-GULOSOR, que já provamos correto.

Exercícios

7.6 Resolva a seguinte instância do problema dos intervalos disjuntos (a mesma da Tabela 7.1):

6	25	9	23	7	18	30	1
15	30	15	28	16	24	34	26

A primeira linha dá as origens dos intervalos e a segunda dá os términos.

7.7 Dê uma prova da correção do algoritmo iterativo INT-GULOSO que não depende da correção da versão recursiva INT-GULOSOR. (Sugestão: mostre que no início de cada iteração valem os seguintes invariantes: (1) $b(x) < a(s)$ para todo x em X e todo s em S' e (2) X é um subconjunto viável máximo de $S - S'$.)

7.4 Implemementação do algoritmo guloso

O algoritmo INT-GULOSO da seção anterior está escrito num nível de abstração um tanto alto. Assim, não fica claro como implementar as linhas 3 e 5 de maneira eficiente.

Uma implementação eficiente de INT-GULOSO começa com um pré-processamento que coloca o conjunto S de intervalos em *ordem crescente de términos*. Depois desse pré-processamento, podemos supor que $S = \{s_1, \ldots, s_n\}$ e

$$b(s_1) \leq b(s_2) \leq \cdots \leq b(s_n) . \tag{7.2}$$

(Veja a figura 7.2.) Agora podemos reescrever o algoritmo como segue:

INTERVALOS-GULOSO $(s_1, \ldots, s_n)$ ▷ $n \geq 1$ e vale (7.2)
1 $j \leftarrow 1$
2 $I \leftarrow \{1\}$
3 para $k \leftarrow 2$ até n faça
4 se $a(s_k) > b(s_j)$
5 então $I \leftarrow I \cup \{k\}$
6 $j \leftarrow k$
7 devolva $\{s_i : i \in I\}$

(O algoritmo pode se restringir às instâncias $n \geq 1$ pois a instância $n = 0$ é trivial.)

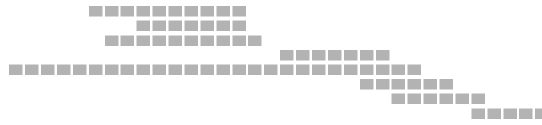

Figura 7.2: Intervalos da figura 7.1 em ordem crescente de término.

Desempenho. A análise de desempenho de INTERVALOS-GULOSO é muito simples: o algoritmo gasta tempo essencialmente constante com cada um dos n intervalos e portanto consome $\mathrm{O}(n)$ unidades de tempo.

É preciso levar em conta ainda o pré-processamento que estabelece a propriedade (7.2). Rearranjar os intervalos em ordem crescente de término consome $\Theta(n \lg n)$ unidades de tempo (veja o Capítulo 3, por exemplo). Podemos dizer, portanto, que a solução completa do problema dos intervalos disjuntos consome

$$\Theta(n \lg n)$$

unidades de tempo.

Exercícios

7.8 Analise a seguinte versão alternativa do algoritmo INTERVALOS-GULOSO:

```
INTERVALOS-GULOSO (s_1, ..., s_n)
1  I ← ∅
2  j ← 1
3  enquanto j ≤ n faça
4      I ← I ∪ {j}
5      k ← j + 1
6      enquanto k ≤ n e a(s_k) ≤ b(s_j) faça
7          k ← k + 1
8      j ← k
9  devolva {s_i : i ∈ I}
```

7.9 Reescreva o algoritmo dos intervalos disjuntos que os intervalos estão em ordem crescente de origens.

7.5 Observações sobre algoritmos gulosos

Para explicar o termo genérico "algoritmo guloso", é preciso dizer antes o que é um problema de otimização. Um problema *de otimização* busca um conjunto de objetos[1] de um dado tipo (intervalos, por exemplo) que seja *ótimo* num certo sentido (viável máximo, por exemplo). Assim, uma solução ótima[2] de uma instância do problema é um conjunto ótimo de objetos.

[1] Aqui, "objeto" é apenas um sinônimo de "coisa".

[2] A expressão "solução ótima" é redundante; é mais correto dizer apenas "solução". Estou usando a expressão redundante apenas para dar ênfase.

Um *algoritmo guloso* (= *greedy algorithm*) para um problema de otimização constrói uma solução gradualmente, escolhendo um novo objeto a cada passo. O objeto escolhido é o melhor disponível naquele passo, de acordo com algum critério estabelecido a priori. Assim, uma solução é construída com objetos *localmente* ótimos.

A estratégia de um algoritmo guloso é semelhante ao de um montanhista que escolhe sempre uma trilha ascendente na esperança de chegar ao pico mais alto da cadeia de montanhas.

Problemas de otimização que podem ser resolvidos por algoritmos gulosos são raros. Um problema desse tipo precisa ter a seguinte "propriedade gulosa": qualquer objeto que satisfaz o critério de escolha local pertence a alguma solução ótima. Cada instância de um problema desse tipo tem, tipicamente, muitas soluções ótimas diferentes (todas igualmente boas).

Eis algumas características típicas de um algoritmo guloso:

- o algoritmo constrói uma solução ótima global a partir de objetos escolhidos por um critério local;
- o algoritmo é míope: enxerga apenas as informações disponíveis no passo corrente;
- o algoritmo nunca se arrepende nem volta atrás: cada um dos objetos escolhidos fará parte da solução final;
- o algoritmo tem aparência simples e inocente;
- o algoritmo é muito rápido;
- é difícil entender *por que* a solução produzida pelo algoritmo é ótima.

Notas bibliográficas

O problema dos intervalos disjuntos é discutido no livro de Kleinberg e Tardos [10], no livro de Cormen *et al*. [3], e no verbete *Interval scheduling* da Wikipedia.

Capítulo 8

As linhas de um parágrafo

Um parágrafo de texto é uma sequência de palavras, sendo cada palavra uma sequência de caracteres. Queremos imprimir um parágrafo em uma ou mais linhas consecutivas de uma folha de papel de tal modo que cada linha tenha no máximo L caracteres. As palavras do parágrafo não devem ser quebradas entre linhas e cada duas palavras consecutivas numa linha devem ser separadas por um espaço.

Para que a margem direita fique razoavelmente uniforme, queremos distribuir as palavras pelas linhas de modo a minimizar a soma dos cubos dos espaços em branco que sobram no fim de todas as linhas exceto a última.

8.1 O problema

Para simplificar a discussão do problema, convém numerar as palavras do parágrafo em *ordem inversa* e confundir as palavras com os seus comprimentos. Diremos, pois, que um **parágrafo** é uma sequência $c_n, c_{n-1}, \ldots, c_2, c_1$ de números naturais. Diremos também que cada c_i é uma **palavra**: c_n é a primeira palavra do parágrafo e c_1 é a última.

Um **trecho** de um parágrafo $c_n, c_{n-1}, \ldots, c_1$ é qualquer sequência da forma $c_m, c_{m-1}, \ldots, c_k$ com $n \geq m \geq k \geq 1$. Este trecho será denotado por (m, k). O **comprimento** do trecho (m, k) é o número

$$c(m,k) \ := \ c_m + 1 + c_{m-1} + 1 + \cdots + 1 + c_k \,.$$

Os próximos conceitos envolvem um número natural L que chamaremos **capacidade de uma linha**. Um trecho (m, k) é **curto** se $c(m, k) \leq L$. O **defeito** de um trecho curto (m, k) é o número

$$(L - c(m,k))^3 \,.$$

Uma L-**decomposição** do parágrafo $c_n, c_{n-1}, \ldots, c_1$ é uma subdivisão do parágrafo em trechos curtos. Mais precisamente, uma L-decomposição é uma sequência

$$(n, k_q),\, (k_q - 1, k_{q-1}),\, (k_{q-1} - 1, k_{q-2}),\, \ldots,\, (k_2 - 1, k_1)\ (k_1 - 1, 1)$$

de trechos curtos em que $n \geq k_q > k_{q-1} > \cdots > k_2 > k_1 > 1$. O **defeito** de uma decomposição é a soma dos defeitos de todos os trechos da decomposição exceto o último. Uma decomposição é **ótima** se tem defeito mínimo.

```
Aaaa bb cccc d eee ff gggg iii
jj kkk. Llll mmm nn ooooo-----
ppppp qq r sss ttt uu.
```

```
Aaaa bb cccc d eee ff gggg----
iii jj kkk. Llll mmm nn ooooo-
ppppp qq r sss ttt uu.
```

Figura 8.1: Duas decomposições do mesmo parágrafo. As linhas têm capacidade $L = 30$ e os caracteres "–" representam os espaços em branco que contribuem para o defeito. O defeito da primeira decomposição é $0^3 + 5^3 = 125$ e o da segunda é $4^3 + 1^3 = 65$.

```

ccc
```

```

bbb ccc
```

```
aaa bbb--
ccc
```

Figura 8.2: À direita temos uma decomposição ótima de um parágrafo c_3, c_2, c_1. A capacidade das linhas é $L = 9$. Os caracteres "–" representam os espaços em branco que contribuem para o defeito. No centro, temos uma decomposição ótima do parágrafo c_2, c_1. À esquerda, uma decomposição ótima do parágrafo c_1.

Problema da decomposição de um parágrafo: Dado um parágrafo $c_n, c_{n-1}, \ldots, c_1$ e um número natural L, encontrar uma L-decomposição ótima do parágrafo.

É claro que as instâncias do problema em que $c_i > L$ para algum i não têm solução.

Exercícios

8.1 Tente explicar por que a definição de defeito usa a terceira potência e não a primeira ou a segunda.

8.2 Considere a seguinte heurística "gulosa": preencha a primeira linha até onde for possível, depois preencha o máximo possível da segunda linha, e assim por diante. Mostre que esta heurística não resolve o problema.

8.3 Suponha que $c_i = 1$ para $i = n, \ldots, 1$. Mostre que o defeito de uma decomposição ótima é no máximo $\lfloor 2n/L \rfloor$.

8.2 A estrutura recursiva do problema

O problema da decomposição de parágrafos tem caráter recursivo, como passamos a mostrar. Suponha que (n, k) é o primeiro trecho de uma decomposição ótima do

parágrafo $c_n, c_{n-1}, \ldots, c_1$. Se $k \geq 2$ então

a correspondente decomposição de $c_{k-1}, c_{k-2}, \ldots, c_1$ é ótima.

Eis a prova desta propriedade: Seja x o defeito da decomposição de $c_{k-1}, \ldots, c_1$ induzida pela decomposição ótima de $c_n, \ldots, c_1$. Suponha agora, por um momento, que o parágrafo $c_{k-1}, \ldots, c_1$ tem uma decomposição com defeito menor que x. Então a combinação desta decomposição com o trecho (n, k) é uma decomposição de $c_n, \ldots, c_1$ que tem defeito menor que o mínimo, o que é impossível.

A propriedade recursiva pode ser representada por uma relação de recorrência. Seja $X(n)$ o defeito de uma decomposição ótima do parágrafo $c_n, c_{n-1}, \ldots, c_1$. Se o trecho $(n, 1)$ é curto então $X(n) = 0$. Caso contrário,

$$X(n) \;=\; \min_{c(n,k)\leq L} \left((L - c(n,k))^3 + X(k-1) \right), \tag{8.1}$$

sendo o mínimo tomado sobre todos os trechos curtos da forma (n, k).

A recorrência poderia ser transformada facilmente num algoritmo recursivo. Mas o cálculo do $\min$ em (8.1) levaria o algoritmo a resolver as mesmas subinstâncias várias vezes, o que é muito ineficiente.

8.3 Um algoritmo de programação dinâmica

Para usar a recorrência (8.1) de maneira eficiente, basta interpretar X como uma tabela $X[1\,..\,n]$ e calcular $X[n]$, depois $X[n{-}1]$, e assim por diante. Esta é a técnica da programação dinâmica. O seguinte algoritmo implementa esta ideia. Ele devolve o defeito mínimo de um parágrafo $c_n, c_{n-1}, \ldots, c_1$ supondo $n \geq 1$, linhas de capacidade L e $c_i \leq L$ para todo i:

```
DefeitoMínimo (c_n, ..., c_1, L)
 1  para m ← 1 até n faça
 2      X[m] ← ∞
 3      k ← m
 4      s ← c_m
 5      enquanto k > 1 e s ≤ L faça
 6          X' ← (L − s)^3 + X[k−1]
 7          se X' < X[m] então X[m] ← X'
 8          k ← k − 1
 9          s ← s + 1 + c_k
10      se s ≤ L então X[m] ← 0
11  devolva X[n]
```

(Não é difícil modificar o algoritmo de modo que ele produza uma decomposição ótima e não apenas o seu defeito.)

No início de cada iteração do bloco de linhas 6–9, o valor da variável s é $c(m, k)$. As primeiras execuções do processo iterativo descrito nas linhas 5–9 terminam tipicamente com $k = 1$ e (m, k) é o único trecho da decomposição. As execuções subsequentes do mesmo bloco terminam com $k \geq 2$ e portanto com $s > L$.

O algoritmo está correto. Na linha 1, imediatamente antes da comparação de m com n, temos o seguinte invariante:

$$\begin{gathered} X[m-1] \text{ é o defeito mínimo do parágrafo } c_{m-1}, \ldots, c_1\,, \\ X[m-2] \text{ é o defeito mínimo do parágrafo } c_{m-2}, \ldots, c_1\,, \\ \ldots\,, \\ X[1] \text{ é o defeito mínimo do parágrafo } c_1\,. \end{gathered}$$

Este invariante vale vacuamente na primeira passagem pela linha 1. Suponha agora que o invariante vale no início de uma iteração qualquer. A propriedade continua valendo no início da iteração seguinte, pois o bloco de linhas 2–10 calcula o defeito de uma decomposição ótima de parágrafo $c_m, c_{m-1}, \ldots, c_1$ usando a recorrência (8.1).

No fim do processo iterativo temos $m = n + 1$ e portanto $X[n]$ é o defeito mínimo do parágrafo $c_n, \ldots, c_1$.

Desempenho. Adote n como tamanho da instância $(c_n, \ldots, c_1, L)$ do problema. Podemos supor que uma execução de qualquer linha do pseudocódigo consome uma quantidade de tempo que não depende de n. Assim, o consumo de tempo é determinado pelo número de execuções das várias linhas.

Para cada valor fixo de m, o bloco de linhas 6–9 é executado no máximo $m - 1$ vezes. Como m assume os valores $1, 2, \ldots, n$, o número total de execuções do bloco de linhas 6–9 não passa de $\frac{1}{2}n(n-1)$. Portanto, o algoritmo consome

$$\Theta(n^2)$$

unidades de tempo no pior caso. No melhor caso (que acontece, por exemplo, quando $c_i \geq \lceil L/2 \rceil$ para cada i), o algoritmo consome $\Theta(n)$ unidades de tempo.

Exercícios

8.4 Modifique o algoritmo DEFEITOMÍNIMO para que ele devolva a descrição $\langle k_q, k_{q-1}, \ldots, k_1 \rangle$ de uma decomposição ótima do parágrafo $c_n, \ldots, c_1$ e não apenas o defeito da decomposição.

Notas bibliográficas

O problema que discutimos neste capítulo é proposto como exercício nos livros de Cormen *et al.* [3], Parberry [17] e Parberry e Gasarch [18].

Capítulo 9

Mochila de valor máximo

Suponha dado um conjunto de objetos e uma mochila. Cada objeto tem um certo peso e um certo valor. Queremos escolher um conjunto de objetos que tenha o maior valor possível sem ultrapassar a capacidade (ou seja, o limite de peso) da mochila. Este célebre problema aparece em muitos contextos e faz parte de muitos problemas mais elaborados.

9.1 O problema

Suponha dados números naturais $p_1, \ldots, p_n$ e $v_1, \ldots, v_n$. Diremos que p_i é o **peso** e v_i é o **valor** de i. Para qualquer subconjunto S de $\{1, 2, \ldots, n\}$, sejam $p(S)$ e $v(S)$ os números $\sum_{i \in S} p_i$ e $\sum_{i \in S} v_i$ respectivamente. Diremos que $p(S)$ é o **peso** e $v(S)$ é o **valor** de S.

Em relação a um número natural c, um subconjunto S de $\{1, \ldots, n\}$ é **viável** se $p(S) \leq c$. Um subconjunto viável S^* de $\{1, \ldots, n\}$ é **ótimo** se $v(S^*) \geq v(S)$ para todo conjunto viável S.

> *Problema da mochila:* Dados números naturais $p_1, \ldots, p_n$, c, $v_1, \ldots, v_n$, encontrar um subconjunto ótimo de $\{1, \ldots, n\}$.

Uma **solução** da instância $(n,\ p,\ c,\ v)$ do problema é qualquer subconjunto ótimo de $\{1, \ldots, n\}$. Poderíamos ser tentados a encontrar uma solução examinando todos os subconjuntos $\{1, \ldots, n\}$. Mas esse algoritmo é inviável, pois consome $\Omega(2^n)$ unidades de tempo.

9.2 A estrutura recursiva do problema

Seja S uma solução da instância $(n,\ p,\ c,\ v)$. Temos duas possibilidades, conforme n esteja ou não em S. Se $n \notin S$ então S também é um solução da instância $(n-1,\ p,\ c,\ v)$. Se $n \in S$ então $S - \{n\}$ é solução de $(n-1,\ p,\ c - p_n,\ v)$.

Podemos resumir isso dizendo que o problema tem estrutura recursiva: toda solução de uma instância do problema contém soluções de instâncias menores.

Se denotarmos por $X(n,c)$ o valor de um solução de $(n,\, p,\, c,\, v)$, a estrutura recursiva do problema pode ser representada pela seguinte recorrência:

$$X(n,c) \;=\; \max\big(\, X(n-1,\, c) \;,\; X(n-1,\, c-p_n) + v_n \,\big)\;. \tag{9.1}$$

O segundo termo de $\max$ só faz sentido se $c - p_n \geq 0$, e portanto

$$X(n,c) \;=\; X(n-1,c) \tag{9.2}$$

quando $p_n > c$.

Poderíamos calcular $X(n,c)$ recursivamente. Mas o algoritmo resultante é muito ineficiente, pois resolve cada subinstância um grande número de vezes.

Exercícios

9.1 Escreva um algoritmo recursivo baseado na recorrência (9.1-9.2). Calcule o consumo de tempo do seu algoritmo no pior caso. (Sugestão: para cada n, tome a instância em que $c = n$ e $p_i = v_i = 1$ para cada i. Mostre que o consumo de tempo $T(n)$ para essa família de instâncias satisfaz a recorrência $T(n) = 2T(n-1) + 1$.)

9.3 Algoritmo de programação dinâmica

Para obter um algoritmo eficiente a partir da recorrência (9.1-9.2), é preciso armazenar as soluções das subinstâncias numa tabela à medida que elas forem sendo obtidas, evitando assim que elas sejam recalculadas. Esta é a técnica da programação dinâmica, que já encontramos em capítulos anteriores.

Como c e todos os p_i são números naturais, podemos tratar X como uma tabela com linhas indexadas por $0\,..\,n$ e colunas indexadas por $0\,..\,c$. Para cada i entre 0 e n e cada b entre 0 e c, a casa $X[i,b]$ da tabela será o valor de uma solução da instância $(i,\, p,\, b,\, v)$. As casas da tabela X precisam ser preenchidas na ordem certa, de modo que toda vez que uma casa for requisitada o seu valor já tenha sido calculado.

O algoritmo abaixo recebe uma instância $(n,\, p,\, c,\, v)$ do problema e devolve o valor de uma solução da instância:[1]

```
MOCHILAPD (n, p, c, v)
1  para b ← 0 até c faça
2       X[0, b] ← 0
3       para j ← 1 até n faça
4            x ← X[j − 1, b]
5            se b − p_j ≥ 0
6                 então  y ← X[j − 1, b − p_j] + v_j
7                            se x < y então x ← y
8            X[j, b] ← x
9  devolva X[n, c]
```

[1] É fácil extrair da tabela X um subconjunto viável de $\{1,\ldots,n\}$ que tenha valor $X[n,c]$. Veja o Exercício 9.4.

p	v
4	500
2	400
1	300
3	450

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	500	500
2	0	0	400	400	500	500
3	0	300	400	700	700	800
4	0	300	400	700	750	850

Figura 9.1: Uma instância do problema da mochila com $n = 4$ e $c = 5$. A figura mostra a tabela $X[0\mathinner{.\,.} n, 0\mathinner{.\,.} c]$ calculada pelo algoritmo MOCHILAPD.

O algoritmo está correto. A cada passagem pela linha 1, imediatamente antes das comparação de b com c, as b primeiras colunas da tabela estão corretas, ou seja,

$$X[i, a] \text{ é o valor de uma solução da instância } (i,\ p,\ a,\ v) \tag{9.3}$$

para todo i no intervalo $0\mathinner{.\,.} n$ e todo a no intervalo $0\mathinner{.\,.} b{-}1$. Para provar este invariante, basta entender o processo iterativo nas linhas 3 a 8. No início de cada iteração desse processo,

$$X[i, b] \text{ é o valor de uma solução da instância } (i,\ p,\ b,\ v) \tag{9.4}$$

para todo i no intervalo $0\mathinner{.\,.} j{-}1$. Se o invariante (9.4) vale no início de uma iteração, ele continua valendo no início da iteração seguinte em virtude da recorrência (9.1-9.2). Isto prova que (9.3) de fato vale a cada passagem pela linha 1.

Na última passagem pela linha 1 temos $b = c{+}1$ e portanto (9.3) garante que $X[n, c]$ é o valor de uma solução da instância $(n,\ p,\ c,\ v)$.

Desempenho. É razoável supor que uma execução de qualquer das linhas do pseudocódigo consome uma quantidade de tempo que não depende de n nem de c. Portanto, basta contar o número de execuções das várias linhas.

Para cada valor fixo de b, o bloco de linhas 4–8 é executado n vezes. Como b varia de 0 a c, o número total de execuções do bloco de linhas 4–8 é $(c + 1)n$. As demais linhas são executadas no máximo $c + 1$ vezes. Esta discussão mostra que o algoritmo consome

$$\Theta(n\,c)$$

unidades de tempo. (Poderíamos ter chegado à mesma conclusão observando que o consumo de tempo do algoritmo é proporcional ao número de casas da tabela X.)

O consumo de tempo de MOCHILAPD é muito sensível às variações de c. Imagine, por exemplo, que c e os elementos de p são todos multiplicados por 100. Como isto constitui uma mera mudança de escala, a nova instância do problema é conceitualmente idêntica à original. No entanto, nosso algoritmo consumirá 100 vezes mais tempo para resolver a nova instância.

Observações sobre o consumo de tempo. Ao dizer que o consumo de tempo do algoritmo é $\Theta(nc)$, estamos implicitamente adotando o par de números (n, c) como tamanho da instância $(n,\, p,\, c,\, v)$. No entanto, é mais razoável dizer que o tamanho da instância é $(n, \lceil \lg c \rceil)$, pois c pode ser escrito com apenas $\lceil \lg c \rceil$ bits. É mais razoável, portanto, dizer que o algoritmo consome

$$\Theta(n\, 2^{\lg c})$$

unidades de tempo. Isto torna claro que o consumo de tempo cresce explosivamente com $\lg c$.

Gostaríamos de ter um algoritmo cujo consumo de tempo não dependesse do valor de c, um algoritmo cujo consumo de tempo estivesse em $O(n^2)$, ou $O(n^{10})$, ou $O(n^{100})$. (Um tal algoritmo seria considerado "fortemente polinomial".) Acredita-se, porém, que um tal algoritmo não existe.[2]

Exercícios

9.2 É verdade que $X[i, 0] = 0$ para todo i depois da execução do algoritmo MOCHILAPD?

9.3 Por que nosso enunciado do problema inclui instâncias em que c vale 0? Por que inclui instâncias com objetos de peso nulo?

9.4 Mostre como extrair da tabela $X[0\,..\,n, 0\,..\,c]$ produzida pelo algoritmo MOCHILAPD um subconjunto ótimo de $\{1, \ldots, n\}$.

9.5 Faça uma mudança de escala para transformar o conjunto $\{0, 1, \ldots, c\}$ no conjunto de números racionais entre 0 e n. Aplique o mesmo fator de escala aos pesos p_i. O algoritmo MOCHILAPD pode ser aplicado a essa nova instância do problema?

9.4 Instâncias especiais do problema

Algumas coleções de instâncias do problema da mochila têm especial interesse prático. Se $v_i = p_i$ para todo i, temos o célebre problema *subset sum*, que pode ser apresentado assim: imagine que você emitiu cheques de valores $v_1, \ldots, v_n$ durante o mês; se o banco debitou um total de c na sua conta no fim do mês, quais podem ter sido os cheques debitados?

Considere agora as instâncias do problema da mochila em que $v_i = 1$ para todo i. (Você pode imaginar que $p_1, \ldots, p_n$ são os tamanhos de n arquivos digitais. Quantos desses arquivos cabem num *pen drive* de capacidade c?) Esta coleção de instâncias pode ser resolvida por um algoritmo "guloso" óbvio que é muito mais eficiente que MOCHILAPD.

Notas bibliográficas

O problema da mochila é discutido na Seção 16.2 do livro de Cormen *et al.* [3] e na Seção 8.4 do livro de Brassard e Bratley [2], por exemplo.

[2] O problema da mochila é NP-difícil. Veja o livro de Cormen *et al.* [3].

Capítulo 10

Mochila de valor quase máximo

O algoritmo de programação dinâmica para o problema da mochila (veja o Capítulo 9) é lento. Dada a dificuldade de encontrar um algoritmo mais rápido,[1] faz sentido reduzir nossas exigências e procurar por uma solução *quase* ótima. Um tal algoritmo deve calcular um conjunto viável de valor maior que uma fração predeterminada (digamos 50%) do valor máximo.

10.1 O problema

Convém repetir algumas das definições da Seção 9.1. Dados números naturais $p_1, \ldots, p_n$, c e $v_1, \ldots, v_n$, diremos que p_i é o **peso** e v_i é o **valor** de i. Para qualquer subconjunto S de $\{1, \ldots, n\}$, denotaremos por $p(S)$ a soma $\sum_{i \in S} p_i$ e diremos que S é **viável** se $p(S) \leq c$. O **valor** de S é o número $v(S) := \sum_{i \in S} v_i$. Um conjunto viável S^* é **ótimo** se $v(S^*) \geq v(S)$ para todo conjunto viável S.

> *Problema da mochila:* Dados números naturais $p_1, \ldots, p_n$, c, $v_1, \ldots, v_n$, encontrar um subconjunto viável ótimo de $\{1, \ldots, n\}$.

Todo objeto de peso maior que c pode ser ignorado e qualquer objeto de peso nulo pode ser incluído em todos os conjuntos viáveis. Suporemos então, daqui em diante, que

$$1 \leq p_i \leq c \tag{10.1}$$

para todo i.

10.2 Um algoritmo de aproximação

O algoritmo que descreveremos a seguir tem caráter "guloso" e dá preferência aos objetos de maior valor específico. O **valor específico** de um objeto i é o número v_i/p_i. Para simplificar a descrição do algoritmo, suporemos que os objetos são dados em ordem

[1] O problema é NP-difícil. Veja o livro de Cormen *et al.* [3].

decrescente de valor específico, ou seja, que

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n} \,. \tag{10.2}$$

Nosso algoritmo recebe uma instância do problema que satisfaz as condições (10.1) e (10.2) e devolve um subconjunto viável X de $\{1, \ldots, n\}$ tal que

$$v(X) \geq \tfrac{1}{2} v(S^*) \,,$$

sendo S^* é um subconjunto viável ótimo:

```
MochilaQuaseÓtima (n, p, c, v)
1   X ← ∅
2   s ← x ← 0
3   k ← 1
4   enquanto k ≤ n e s + p_k ≤ c faça
5       X ← X ∪ {k}
6       s ← s + p_k
7       x ← x + v_k
8       k ← k + 1
9   se k > n ou x ≥ v_k
10      então devolva X
11      senão devolva {k}
```

O bloco de linhas 4–8 determina o maior k tal que $p_1 + \cdots + p_{k-1} \leq c$. No início da linha 9, $X = \{1, \ldots, k-1\}$, $s = p(X)$ e $x = v(X)$.

O algoritmo está correto. No início da linha 9, é claro que X é viável. Se $k > n$ então $X = \{1, \ldots, n\}$ e o algoritmo adota X como solução. Nesse caso, é evidente que $v(X) \geq \frac{1}{2}\, v(S)$ para todo conjunto viável S, como prometido.

Suponha agora que $k \leq n$ no início da linha 9. Graças à hipótese (10.1), o conjunto $\{k\}$ é viável e o algoritmo adota como solução o mais valioso dentre X e $\{k\}$. Resta mostrar que

$$\max\,(v(X), v_k) \;\geq\; \tfrac{1}{2}\, v(S)$$

para todo conjunto viável S. O primeiro passo da demonstração consiste na seguinte observação:

$$\max\,(v(X), v_k) \;\geq\; \tfrac{1}{2}(v(X) + v_k) \;=\; \tfrac{1}{2}\, v(R) \,,$$

sendo $R := X \cup \{k\}$. O segundo passo mostra que $v(R) > v(S)$ qualquer que seja o conjunto viável S:

$$\begin{aligned} v(R) - v(S) &= v(R - S) - v(S - R) \\ &= \sum\nolimits_{i \in R-S} v_i - \sum\nolimits_{i \in S-R} v_i \\ &= \sum\nolimits_{i \in R-S} \frac{v_i}{p_i}\, p_i - \sum\nolimits_{i \in S-R} \frac{v_i}{p_i}\, p_i \end{aligned}$$

$$\begin{aligned}
&\geq \frac{v_k}{p_k}\,p(R-S) \;-\; \frac{v_k}{p_k}\,p(S-R) && (10.3)\\
&= \frac{v_k}{p_k}\big(p(R)-p(S)\big)\\
&> \frac{v_k}{p_k}\big(c-c\big) && (10.4)\\
&= 0\,.
\end{aligned}$$

A relação (10.3) vale porque $v_i/p_i \geq v_k/p_k$ para todo i em R e $v_i/p_i \leq v_k/p_k$ para todo i no complemento de R. Já (10.4) vale porque $p(R) > c$ e $p(S) \leq c$.

Desempenho. Adote n (poderia igualmente ter adotado $2n+1$) como medida do tamanho da instância $(n,\, p,\, c,\, v)$ do problema. Uma execução de qualquer das linhas do pseudocódigo consome uma quantidade de tempo que não depende de n. O bloco de linhas 5–8 é executado no máximo n vezes. Assim, o algoritmo todo consome

$$\Theta(n)$$

unidades de tempo, tanto no pior quanto no melhor caso. O algoritmo propriamente dito é, portanto, linear.

O pré-processamento necessário para fazer valer (10.1) e (10.2) pode ser feito em tempo $\Theta(n \lg n)$ (veja o Capítulo 3). Portanto, o consumo de tempo do processo todo é

$$\Theta(n \lg n)\,.$$

Exercícios

10.1 Construa uma instância do problema da mochila para a qual o algoritmo devolve $\{k\}$ na linha 11.

10.3 Observações sobre algoritmos de aproximação

Um algoritmo rápido cujo resultado é uma fração predeterminadada solução ótima é conhecido como **algoritmo de aproximação**. O resultado do algoritmo MOCHILA-QUASEÓTIMA, por exemplo, é melhor que 50% do ótimo. Esse fator de aproximação pode parecer grosseiro, mas é suficiente para algumas aplicações que precisam de um algoritmo muito rápido.

Outros algoritmos de aproximação para o problema da mochila têm fator de aproximação bem melhor que 50%. O algoritmo de Ibarra e Kim (veja o livro de Fernandes *et al.* [8], por exemplo) garante um fator de aproximação tão próximo de 100% quanto o usuário desejar. Como seria de se esperar, o consumo de tempo do algoritmo é tanto maior quanto maior o fator de aproximação solicitado. O algoritmo de Ibarra e Kim é baseado numa variante de MOCHILAPD em que os papéis de p e v são trocados. (Veja o Exercício 9.5.)

Notas bibliográficas

Algoritmos de aproximação para o problema da mochila são discutidos na Seção 13.2 do livro de Brassard e Bratley [2].

Capítulo 11

A cobertura de um grafo

Imagine um conjunto de salas interligadas por túneis. Um guarda postado numa sala é capaz de vigiar todos os túneis que convergem sobre a sala. Queremos determinar o número mínimo de guardas suficiente para vigiar todos os túneis.

Se houver um custo associado com cada sala (este é o custo de manter um guarda na sala), queremos determinar o conjunto mais barato de guardas capaz de manter todos os túneis sob vigilância.

11.1 Grafos

Um **grafo** é um par (V, A) de conjuntos tal que $A \subseteq \binom{V}{2}$. Cada elemento de A é, portanto, um par não ordenado de elementos de V. Os elementos de V são chamados **vértices** e os de A são chamados **arestas**.

Uma aresta $\{i, j\}$ é usualmente denotada por ij. Os vértices i e j são as **pontas** desta aresta.

11.2 O problema da cobertura

Uma **cobertura** de um grafo é um conjunto X de vértices que contém pelo menos uma das pontas de cada aresta. Se cada vértice i tem um **custo** c_i, o **custo** de uma cobertura X é o número $c(X) := \sum_{i \in X} c_i$.

> *Problema da cobertura:* Dado um grafo cujos vértices têm custos (que são números naturais), encontrar uma cobertura de custo mínimo.

Poderíamos resolver o problema examinando todos os subconjuntos do conjunto de vértices, mas um tal algoritmo é explosivamente lento: seu consumo de tempo cresce exponencialmente com o número de vértices. Infelizmente, acredita-se que não existem algoritmos substancialmente mais rápidos para o problema, nem mesmo quando todos os vértices têm custo unitário.[1]

[1] O problema da cobertura (de grafos) é NP-difícil. Veja o livro de Cormen *et al.* [3].

Faz sentido, portanto, procurar por um bom algoritmo de aproximação, um algoritmo rápido que encontre uma cobertura cujo custo seja limitado por um múltiplo predeterminado do ótimo. (Veja a Seção 10.3.)

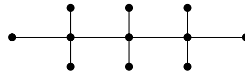

Figura 11.1: Uma instância do problema da cobertura. Encontre uma cobertura mínima, supondo que todos os vértices têm custo 1.

Exercícios

11.1 Seja (V, A) um grafo. Ignore os custos dos vértices do grafo. Uma cobertura X desse grafo é **mínima** se não existe uma cobertura X' tal que $|X'| < |X|$. Uma cobertura X é **minimal** se não existe uma cobertura X' tal que $X' \subset X$. Mostre que uma cobertura minimal pode ser arbitrariamente maior que uma cobertura mínima.

11.3 Um algoritmo de aproximação

O seguinte algoritmo recebe um grafo (V, A) e um número natural c_i para cada vértice i e devolve uma cobertura X tal que $c(X) \leq 2 \cdot c(X_*)$, sendo X_* uma cobertura de custo mínimo:

CoberturaBarata $(V,\ A,\ c)$
1 para cada i em V faça
2 $\quad x_i \leftarrow 0$
3 para cada ij em A faça
4 $\quad y_{ij} \leftarrow 0$
5 para cada pq em A faça
6 $\quad e \leftarrow \min\,(c_p - x_p\ ,\ c_q - x_q)$
7 $\quad y_{pq} \leftarrow y_{pq} + e$
8 $\quad x_p \leftarrow x_p + e$
9 $\quad x_q \leftarrow x_q + e$
10 $X \leftarrow \emptyset$
11 para cada i em V faça
12 $\quad$ se $x_i = c_i$ então $X \leftarrow X \cup \{i\}$
13 devolva X

O algoritmo atribui números naturais y às arestas. Você pode imaginar que y_{pq} é a quantia que a aresta pq paga a cada uma de suas pontas para convencer uma delas

a fazer parte da cobertura. Um vértice p aceita participar da cobertura se $\sum_q y_{pq} \ge c_p$, sendo a soma feita sobre todas as arestas que têm ponta p. A regra do jogo é nunca pagar mais que o necessário. Portanto, $\sum_q y_{pq} \le c_p$ para todo vértice p.

O algoritmo está correto. No início de cada iteração do bloco de linhas 6–9, temos os seguintes invariantes:

i. $x_i = \sum_j y_{ij}$ para todo i em V,

ii. $x_i \le c_i$ para todo i em V,

iii. para toda aresta ij já examinada tem-se $x_i = c_i$ ou $x_j = c_j$.

Estas propriedades são obviamente verdadeiras no início da primeira iteração. No início de cada iteração subsequente, o invariante i continua valendo, pois y_{pq}, x_p e x_q são acrescidos da mesma quantidade e. O invariante ii continua valendo, pois $e \le c_p - x_p$ e $e \le c_q - x_q$. O invariante iii continua valendo, pois $e = c_p - x_p$ ou $e = c_q - x_q$.

Os invariantes i e ii têm a seguinte consequência: no início de cada iteração do bloco de linhas 6–9, temos

$$\sum_{ij \in A} y_{ij} \;\le\; c(Z) \tag{11.1}$$

para qualquer cobertura Z. Para mostrar isso, basta observar que $\sum_{ij \in A} y_{ij} \le \sum_{i \in Z} \sum_j y_{ij} = \sum_{i \in Z} x_i \le \sum_{i \in Z} c_i$. A primeira desigualdade vale porque toda aresta tem pelo menos uma de suas pontas em Z. A igualdade seguinte vale em virtude do invariante i. A última desigualdade decorre do invariante ii.

Agora observe que no fim do bloco de linhas 10–12 temos

$$c(X) \;\le\; 2 \sum_{ij \in A} y_{ij}\,. \tag{11.2}$$

De fato, como $x_i = c_i$ para todo i em X, temos $\sum_{i \in X} c_i = \sum_{i \in X} x_i = \sum_{i \in X} \sum_j y_{ij} \le 2 \sum_{ij \in A} y_{ij}$. A segunda igualdade vale em virtude do invariante i. A última desigualdade é verdadeira pois cada aresta tem no máximo duas pontas em X.

Segue de (11.2) e (11.1) que, depois do bloco de linhas 10–12, para qualquer cobertura Z,

$$c(X) \le 2\, c(Z)\,.$$

Isto vale, em particular, se Z é uma cobertura de custo mínimo. Como X é uma cobertura (em virtude do invariante iii), o algoritmo cumpre o que prometeu.

O fator de aproximação 2 pode parecer ser grosseiro, mas o fato é que ninguém descobriu ainda um algoritmo eficiente com fator de aproximação 1.9 por exemplo.

Desempenho. Podemos supor que o grafo é dado da maneira mais crua possível: os vértices são $1, 2, \ldots, n$ e as arestas são listadas em ordem arbitrária.

Seja m o número de arestas do grafo e adote o par (n, m) como tamanho de uma instância do problema. Podemos supor que uma execução de qualquer das linhas do pseudocódigo consome tempo que não depende de n nem de m. A linha 2 é executada n vezes. A linha 4 é executada m vezes. A linha 12 é executada n vezes. O bloco de linhas 6–9 é repetido m vezes. Assim, o consumo de tempo total do algoritmo está em

$$\Theta(n + m)\,,$$

tanto no pior quanto no melhor caso. O algoritmo é, portanto, linear.

11.4 Comentários sobre o método primal-dual

O algoritmo COBERTURABARATA é o resultado da aplicação do **método primal-dual** de concepção de algoritmos. O primeiro passo do método (omitido acima) é escrever um programa linear que represente o problema. As variáveis duais do programa linear são as variáveis y do nosso algoritmo. A relação (11.1) é uma manifestação da dualidade de programação linear. (Veja o meu material [6] sobre o assunto.)

Os algoritmos do tipo primal-dual têm um certo caráter "guloso". No algoritmo COBERTURABARATA, por exemplo, as arestas são examinadas em ordem arbitrária e as linhas 8–9 do algoritmo aumentam o valor de x_p e x_q — o que torna mais provável a inclusão de p e q em X — o máximo possível sem se preocupar com o objetivo global de minimizar o custo da cobertura X.

Exercícios

11.2 Considere as instâncias do problema da cobertura em que todos os vértices têm o mesmo custo. Dê um algoritmo de aproximação para essas instâncias que seja mais simples que COBERTURABARATA. O seu algoritmo deve produzir uma cobertura de tamanho não superior ao dobro do ótimo.

11.5 Instâncias especiais do problema

Um grafo é **bipartido** se seu conjunto de vértices admite uma bipartição (U, W) tal que toda aresta tem uma ponta em U e outra em W. (Portanto, U e W são coberturas.)

Existe um algoritmo muito elegante e eficiente para o problema da cobertura restrito a grafos bipartidos. (Veja a Seção 22.4 do livro de Sedgewick [19], por exemplo.) O algoritmo consome $\Theta(nm)$ unidades de tempo no pior caso, sendo n o número de vértices e m o número de arestas do grafo.

Notas bibliográficas

Este capítulo foi baseado no livro de Kleinberg e Tardos [10].

Capítulo 12

Conjuntos independentes em grafos

Considere a relação amigo-de entre os usuários de um *site* de relacionamentos. Queremos encontrar um conjunto máximo de usuários que sejam amigos dois a dois.

Um problema da mesma natureza (mas computacionalmente mais fácil) já foi estudado no Capítulo 7: encontrar um conjunto máximo de intervalos dois a dois disjuntos.

12.1 O problema

Um conjunto I de vértices de um grafo é **independente** se seus elementos são dois a dois não vizinhos, ou seja, se nenhuma aresta do grafo tem ambas as pontas em I. Um conjunto independente I é **máximo** se não existe um conjunto independente I' tal que $|I'| > |I|$.

> *Problema do conjunto independente:* Encontrar um conjunto independente máximo num grafo dado.

Conjuntos independentes são os complementos das coberturas (veja o Capítulo 11). De fato, em qualquer grafo (V, A), se I é um conjunto independente então $V - I$ é uma cobertura. Reciprocamente, se C é uma cobertura então $V - C$ é um conjunto independente. Portanto, encontrar um conjunto independente máximo é computacionalmente equivalente a encontrar uma cobertura mínima. Não se conhecem bons algoritmos para esses problemas.

Apesar da relação de complementaridade, algoritmos de aproximação para o problema da cobertura de grafos (como o que discutimos na Seção 11.3) não podem ser convertidos em algoritmos de aproximação para o problema do conjunto independente. De fato, se uma cobertura mínima num grafo de n vértices tiver apenas 100 vértices e se nosso algoritmo de aproximação obtiver uma cobertura com 199 vértices, o correspondente conjunto independente terá $n - 199$ vértices. Mas este número não é menor que uma percentagem fixa do tamanho de um conjunto independente máximo.

Discutiremos a seguir um algoritmo probabilístico muito simples, que fornece um conjunto independente de tamanho *esperado* razoavelmente grande, embora modesto.

Exercícios

12.1 Mostre que o problema dos intervalos disjuntos discutido no Capítulo 7 é um caso especial (ou seja, uma coleção de instâncias) do problema do conjunto independente.

12.2 Um conjunto independente I em um grafo é **maximal** se não existe um conjunto independente I' no grafo tal que $I' \supset I$. Mostre que um conjunto independente maximal pode ser arbitrariamente menor que um conjunto independente máximo.

12.2 Um algoritmo probabilístico

Convém lembrar que todo grafo com n vértices tem entre 0 e $\binom{n}{2}$ arestas. Portanto, se m é o número de arestas então $0 \le 2m \le n^2 - n$. O algoritmo que discutiremos a seguir produz um conjunto independente I cujo tamanho esperado é

$$n^2/4m\,.$$

O resultado é intuitivamente razoável: quanto mais denso o grafo, ou seja, quanto mais próximo m de n^2, menor o tamanho esperado de I.

m	$n/2$	n	$2n$	$10n$	$n\sqrt{n}/4$	$n^2/4$	$(n^2-n)/2$
$n^2/4m$	$n/2$	$n/4$	$n/8$	$n/40$	$\sqrt{n}$	1	$n/(2n-2)$

Figura 12.1: Comparação entre o número m de arestas e o tamanho esperado do conjunto independente produzido pelo algoritmo CONJINDEPENDENTE.

O algoritmo recebe um grafo com vértices $1, 2, \ldots, n$ e conjunto A de arestas e devolve um conjunto independente I tal que

- $|I| \ge n/2$ se $m \le n/2$ e
- $\mathbb{E}[|I|] \ge n^2/4m$ se $m > n/2$,

sendo $m := |A|$. A expressão $\mathbb{E}[x]$ denota o valor esperado de x.

```
CONJINDEPENDENTE (n, A)
1  para i ← 1 até n faça
2      X[i] ← 1
3  m ← |A|
4  se 2m > n
5      então  para i ← 1 até n faça
6                 r ← RANDOM (2m − 1)
7                 se r < 2m − n
8                     então X[i] ← 0
```

```
 9  para cada ij em A faça
10      se X[i] = 1 e X[j] = 1
11          então X[i] ← 0
12  devolva X[1 . . n]
```

O conjunto independente que o algoritmo devolve é representado pelo vetor booleano X indexado pelos vértices: i pertence ao conjunto independente se e somente se $X[i] = 1$.

A rotina RANDOM é um gerador de números aleatórios. Com argumento U, ela produz um número natural uniformemente distribuído no conjunto $\{0, 1, 2 \ldots, U\}$. (É muito difícil obter números verdadeiramente aleatórios, mas existem bons algoritmos para gerar números pseudoaleatórios.)

O algoritmo está correto. O algoritmo tem duas fases. Na primeira fase (linhas 1–8), o algoritmo elimina vértices até que sobre um conjunto W. Se $m \leq n/2$, W é o conjunto de todos os vértices. Caso contrário, os vértices são eliminados com probabilidade $1 - n/2m$, ou seja, permanecem em W na proporção de n em cada $2m$. (Se $m = 10n$, por exemplo, sobra 1 vértice em cada 20. Se $m = \frac{1}{2}n\sqrt{n}$, sobra 1 em cada $\sqrt{n}$. Se $m = n^2/4$, sobram cerca de 2 vértices apenas.)

É fácil determinar o tamanho esperado de W uma vez que cada um dos n vértices fica em W com probabilidade $n/2m$. Se $w := |W|$ então

$$\mathbb{E}[w] = n\,\frac{n}{2m} = \frac{n^2}{2m}\,.$$

Considere agora o conjunto B das arestas que têm ambas as pontas em W. Como o grafo tem m arestas e a probabilidade de uma ponta de aresta ficar em W é $n/2m$, temos

$$\mathbb{E}[b] = m\left(\frac{n}{2m}\right)^2 = \frac{n^2}{4m}\,,$$

sendo $b := |B|$.

Na segunda fase (linhas 9–11), o algoritmo elimina vértices de W de modo que o conjunto restante I seja independente. Esta segunda fase do algoritmo remove no máximo b vértices e portanto o tamanho esperado de I é

$$\mathbb{E}[|I|] \;\geq\; \mathbb{E}[w] - \mathbb{E}[b] \;=\; \frac{n^2}{2m} - \frac{n^2}{4m} \;=\; \frac{n^2}{4m}\,.$$

Se executarmos o algoritmo um bom número de vezes e escolhermos o maior dos conjuntos independentes que resultar, temos uma boa chance de obter um conjunto independente de tamanho não menor que $n^2/4m$.

Desempenho. O algoritmo é linear. É razoável supor que cada execução da rotina RANDOM consome uma quantidade de tempo que não depende de n nem de m. Assim, o algoritmo consome

$$\Theta(n+m)$$

unidades de tempo, sendo $\Theta(n)$ unidades na primeira fase e $\Theta(m)$ unidades na segunda.

12.3 Comentários sobre algoritmos probabilísticos

Diz-se que um algoritmo é **probabilístico** ou **aleatorizado** se usa números aleatórios. Cada execução de um tal algoritmo dá uma resposta diferente e o algoritmo promete que o valor esperado da resposta é suficientemente bom. Em geral, o algoritmo é executado muitas vezes e o usuário escolhe a melhor das respostas.

Alguns algoritmos (não é o caso do algoritmo CONJINDEPENDENTE acima) prometem também que a resposta fornecida está próxima do valor esperado com alta probabilidade.

Os algoritmos probabilísticos ignoram, em geral, a estrutura e as peculiaridades da instância que estão resolvendo. Mesmo assim, surpreendentemente, muitos algoritmos probabilísticos produzem resultados bastante úteis.

Notas bibliográficas

O algoritmo deste capítulo é descrito, por exemplo, no livro de Mitzenmacher e Upfal [15]. Sobre algoritmos aleatorizados em geral, veja o verbete *Randomized algorithm* na Wikipedia.

Capítulo 13

Busca em largura num grafo

Considere a relação amigo-de entre os usuários de uma rede social. Digamos que dois usuários A e B estão ligados se existe uma sucessão de amigos que leva de A a B.

Suponha que queremos determinar todos os usuários ligados a um dado usuário. A melhor maneira de resolver esse problema é fazer uma busca num grafo (veja a Seção 11.1).

13.1 O problema do componente

Um **caminho** em um grafo é qualquer sequência $(i_1, i_2, \ldots, i_q)$ de vértices tal que cada i_p é vizinho de i_{p-1} para todo $p \geq 2$. Dizemos que um vértice está **ligado** a outro se existe um caminho que começa no primeiro e termina no segundo. O conjunto de todos os vértices ligados a um vértice v é o **componente** (do grafo) **que contém** v.

> *Problema do componente de grafo:* Dado um vértice v de um grafo, encontrar o conjunto de todos os vértices ligados a v.
>
> Este é um dos mais básicos e corriqueiros problemas sobre grafos.

13.2 Busca em largura: versão preliminar

Usaremos o método da "busca em largura" para resolver o problema. Começaremos por escrever uma versão preliminar do algoritmo, em alto nível de abstração.

Dizemos que dois vértices i e j são **vizinhos** se ij é uma aresta. A **vizinhança** de um vértice i é o conjunto de todos os vizinhos de i e será denotada por $Z(i)$.

O seguinte algoritmo recebe um vértice v de um grafo representado por seu conjunto V de vértices e suas vizinhanças $Z(i)$, $i \in V$, e devolve o conjunto de todos os vértices ligados a v.

```
COMPONENTEPRELIM (V, Z, v)
1   P ← ∅
2   C ← {v}
3   enquanto C ≠ ∅ faça
4       seja i um elemento de C
5       para cada j em Z(i) faça
6           se j ∉ C ∪ P
7               então C ← C ∪ {j}
8       C ← C − {i}
9       P ← P ∪ {i}
10  devolva P
```

Você pode imaginar que os vértices em P são pretos, os vértices em C são cinza e todos os demais são brancos. Um vértice é branco enquanto não tiver sido visitado. Ao ser visitado, o vértice fica cinza e assim permanece enquanto todos os seus vizinhos não tiverem sido visitados. Quando todos os vizinhos forem visitados, o vértice fica preto.

O algoritmo está correto. Considere o processo iterativo no bloco de linhas 3–9. A cada passagem pela linha 3, imediatamente antes da comparação de C com $\emptyset$, temos os seguintes invariantes:

i. $P \cap C = \emptyset$,

ii. $v \in P \cup C$,

iii. todo vértice em $P \cup C$ está ligado a v e

iv. toda aresta com uma ponta em P tem a outra ponta em $P \cup C$.

É evidente que estas propriedades valem no início da primeira iteração. Suponha agora que elas valem no início de uma iteração qualquer. É claro que i e ii continuam valendo no início da próxima iteração. No caso do invariante iii, basta verificar que os vértices acrescentados a $P \cup C$ durante esta iteração estão todos ligados a v. Para isso, é suficiente observar que i está ligado a v (invariante iii) e portanto todos os vértices em $Z(i)$ também estão ligados a v.

Finalmente, considere o invariante iv. Por um lado, o único vértice acrescentado a P durante a iteração corrente é i. Por outro lado, ao final da iteração, todos os vizinhos de i estão em $P \cup C$. Assim, o invariante iv continua válido no início da próxima iteração.

No fim do processo iterativo, C está vazio. De acordo com os invariantes ii e iv, todos os vértices de qualquer caminho que começa em v estão em P. Reciprocamente, todo vértice em P está ligado a v, de acordo com invariante iii. Portanto, P é o conjunto de vértices ligados a v. Assim, ao devolver P, o algoritmo cumpre o que prometeu.

Desempenho. Não há como discutir o consumo de tempo do algoritmo de maneira precisa, pois não especificamos estruturas de dados que representem as vizinhanças $Z(i)$ e os conjuntos P e C.

Podemos garantir, entretanto, que a execução do algoritmo termina depois de não mais que $|V|$ iterações do bloco de linhas 4–9. De fato, no decorrer da execução do algoritmo, cada vértice entra em C (linha 7) no máximo uma vez, faz o papel de i na linha 4 no máximo uma vez, é transferido de C para P (linhas 8 e 9) e portanto nunca mais entra em C. Assim, o número de iterações não passa de $|V|$.

Exercícios

13.1 Um grafo é **conexo** se seus vértices estão ligados dois a dois. Escreva um algoritmo que decida se um grafo é conexo.

13.2 Escreva um algoritmo que calcule o número de componentes de um grafo.

13.3 Busca em largura: versão mais concreta

Podemos tratar agora de uma versão mais concreta do algoritmo COMPONENTEPRELIM, que adota estruturas de dados apropriadas para representar os vários conjuntos de vértices.

Suponha que os vértices do grafo são $1, 2, \dots, n$. As vizinhanças dos vértices serão representadas por uma matriz Z com linhas indexadas pelos vértices e colunas indexadas por $1, 2, \dots, n-1$. Os vizinhos de um vértice i serão

$$Z[i,1],\ Z[i,2],\ \dots\ ,\ Z[i,g[i]]\ ,$$

onde $g[i]$ é o **grau** de i, ou seja, o número de vizinhos de i. (Na prática, usam-se listas encadeadas para representar essas vizinhanças.) Observe que cada aresta ij aparece exatamente duas vezes nesta representação: uma vez na linha i da matriz Z e outra vez na linha j.

Os conjuntos P e C da seção anterior serão representados por um vetor cor indexado pelos vértices: $cor[i]$ valerá 2 se $i \in P$, valerá 1 se $i \in C$ e valerá 0 nos demais casos. O conjunto C terá também uma segunda representação: seus elementos ficarão armazenados num vetor $F[a\,..\,b]$, que funcionará como uma fila com início a e fim b. Isso permitirá implementar de maneira eficiente o teste "$C \neq \emptyset$" na linha 3 de COMPONENTEPRELIM, bem como a escolha de i em C na linha 4.

O algoritmo recebe um grafo com vértices $1, 2, \dots, n$, representado por seu vetor de graus g e sua matriz de vizinhos Z e recebe um vértice v. O algoritmo devolve um vetor $cor[1\,..\,n]$ que representa o conjunto de vértices ligados a v (os vértices do componente são os que têm cor 2).

```
COMPONENTE (n, g, Z, v)
1  para i ← 1 até n faça
2       cor[i] ← 0
3  cor[v] ← 1
4  a ← b ← 1
5  F[b] ← v
```

```
 6  enquanto a ≤ b faça
 7      i ← F[a]
 8      para h ← 1 até g[i] faça
 9          j ← Z[i, h]
10          se cor[j] = 0
11              então  cor[j] ← 1
12                     b ← b + 1
13                     F[b] ← j
14      cor[i] ← 2
15      a ← a + 1
16  devolva cor[1 . . n]
```

O algoritmo COMPONENTE está correto pois o pseudocódigo não faz mais que implementar o algoritmo COMPONENTEPRELIM, que já discutimos.

Desempenho. Seja m o número de arestas do grafo. (Em termos dos parâmetros do algoritmo, m é $\frac{1}{2}\sum_{i=1}^{n} g[i]$.) Adotaremos o par (n, m) como medida do tamanho de uma instância do problema.

Podemos supor que uma execução de qualquer das linhas do pseudocódigo consome uma quantidade de tempo que não depende de n nem de m. A linha 7 é executada n vezes no pior caso, pois cada vértice do grafo faz o papel de i na linha 7 uma só vez ao longo da execução do algoritmo. (Segue daí, em particular, que $b \leq n$.) Para cada valor fixo de i, o bloco de linhas 9–13 é executado $g[i]$ vezes. Segue dessas duas observações que, no pior caso, o processo iterativo nas linhas 6–15 consome tempo proporcional à soma

$$\sum_{i=1}^{n} g[i] \,,$$

que vale $2m$. O consumo desse processo iterativo está, portanto, em $\Theta(m)$ no pior caso.

Como o consumo de tempo das linhas 1–5 está em $\Theta(n)$, o consumo de tempo total do algoritmo está em

$$\Theta(n + m)$$

no pior caso. (No melhor caso, o vértice v não tem vizinhos e portanto o algoritmo consome apenas $\Theta(n)$ unidades de tempo.)

Exercícios

13.3 Escreva um algoritmo que calcule um caminho de comprimento mínimo dentre os que ligam dois vértices dados de um grafo. (O comprimento de um caminho é o número de arestas do caminho.)

Capítulo 14

Busca em profundidade num grafo

Este capítulo trata de um segundo algoritmo para o problema estudado no capítulo anterior.

> *Problema do componente de grafo:* Dado um vértice v de um grafo, encontrar o conjunto de todos os vértices ligados a v.

O algoritmo que discutiremos a seguir usa o método da "busca em profundidade". A importância do algoritmo transcende em muito o problema do componente. O algoritmo serve de modelo para soluções eficientes de vários problemas mais complexos, como o de calcular os componentes biconexos de um grafo, por exemplo.

14.1 Busca em profundidade

O seguinte algoritmo recebe um vértice v de um grafo e devolve o conjunto de todos os vértices ligados a v. O conjunto de vértices do grafo é $\{1, \ldots, n\}$ e o conjunto de arestas é representado pelas vizinhanças $Z(p)$, já definidas na Seção 13.2.

COMPONENTE $(n,\ Z,\ v)$
1 para $p \leftarrow 1$ até n faça
2 $\quad$ $cor[p] \leftarrow 0$
3 BUSCAEMPROFUNDIDADE (v)
4 devolva $cor[1\,..\,n]$

O vetor cor, indexado pelo vértices, representa a solução: um vértice p está ligado a v se e somente se $cor[p] = 2$.

O algoritmo COMPONENTE é apenas uma "casca" que repassa o serviço para a rotina recursiva BUSCAEMPROFUNDIDADE. Do ponto de vista desta rotina, o grafo e o vetor cor são variáveis globais (a rotina pode, portanto, consultar e alterar o valor das variáveis).

```
BUSCAEMPROFUNDIDADE (p)
5  cor[p] ← 2
6  para cada q em Z(p) faça
7       se cor[q] = 0
8            então BUSCAEMPROFUNDIDADE (q)
```

O vetor cor só tem dois valores: 0 e 2. Diremos que um vértice p é branco se $cor[p] = 0$ e preto se $cor[p] = 2$. Diremos também que um caminho (veja Seção 13.1) é branco se todos os seus vértices são brancos. A rotina BUSCAEMPROFUNDIDADE pinta de preto alguns dos vértices que eram brancos. Assim, um caminho que era branco quando a rotina foi invocada pode deixar de ser branco durante a execução da rotina.

Podemos dizer agora o que a rotina BUSCAEMPROFUNDIDADE faz. Ela recebe um vértice branco p — que é vizinho de um vértice preto a menos que seja igual a v — e pinta de preto todos os vértices que estejam ligados a p por um caminho branco. (A rotina não altera as cores dos demais vértices.)

Agora que deixamos claro o problema que a rotina BUSCAEMPROFUNDIDADE resolve, é importante adotar uma boa definição de tamanho das instâncias desse problema. O número de vértices e arestas do grafo não dá uma boa medida do tamanho da instância, pois a rotina ignora boa parte do grafo. É bem mais apropriado dizer que o tamanho da instância é o número

$$\sum_{x \in X} g(x) \,, \tag{14.1}$$

onde X é o conjunto dos vértices ligados a p por caminhos brancos e $g(x) := |Z(x)|$ é o grau do vértice x.

A rotina está correta. A prova da correção da rotina BUSCAEMPROFUNDIDADE é uma indução no tamanho da instância. Se o tamanho for 0, o vértice p não tem vizinhos. Se o tamanho for 1, o único vértice em $Z(p)$ é preto. Em qualquer desses casos, a rotina cumpre o que prometeu.

Suponha agora que o tamanho da instância é maior que 1. Seja B o conjunto de vértices brancos no início da execução da rotina e seja X o conjunto dos vértices ligados a p em B. Queremos mostrar que no fim do processo iterativo descrito nas linhas 6–8 o conjunto dos vértices brancos é $B - X$.

Sejam $q_1, q_2, \ldots, q_k$ os elementos de $Z(p)$ na ordem em que eles serão examinados na linha 6. Para $j = 1, 2, \ldots, k$, seja Y_j o conjunto dos vértices ligados a q_j em $B - \{p\}$. É claro que $X = \{p\} \cup Y_1 \cup \cdots \cup Y_k$.

Se $Y_i \cap Y_j \neq \emptyset$ então $Y_i = Y_j$. De fato, se Y_i e Y_j têm um vértice em comum então existe um caminho em $B - \{p\}$ com origem q_i e término em Y_j. Portanto também existe um caminho em $B - \{p\}$ de q_i a q_j, donde $q_j \in Y_i$. Segue daí que $Y_j \subseteq Y_i$. Um raciocínio análogo mostra que $Y_i \subseteq Y_j$.

O processo iterativo descrito nas linhas 6–8 pode ser reescrito como

```
6  para j ← 1 até k faça
7       se cor[q_j] = 0
8            então BUSCAEMPROFUNDIDADE (q_j)
```

e isso permite formular o seguinte invariante: a cada passagem pela linha 6,

$$\text{o conjunto dos vértices brancos é } B - \big(\{p\} \cup Y_1 \cup \cdots \cup Y_{j-1}\big)\,. \tag{14.2}$$

Esta propriedade certamente vale no início da primeira iteração. Suponha agora que ela vale no início de uma iteração qualquer. Se tivermos $Y_j = Y_i$ para algum i entre 1 e $j-1$ então, no início desta iteração, todos os vértices em Y_j já são pretos e a linha 8 não é executada. Caso contrário, Y_j é disjunto de $Y_1 \cup \cdots \cup Y_{j-1}$ e portanto todos os vértices em Y_j estão ligados a q_j por caminhos em $B - \big(\{p\} \cup Y_1 \cup \cdots \cup Y_{j-1}\big)$. Como $\sum_{y \in Y_j} g(y)$ é menor que $\sum_{x \in X} g(x)$, podemos supor, por hipótese de indução, que a invocação de BUSCAEMPROFUNDIDADE na linha 8 com argumento q_j produz o resultado prometido. Assim, no fim desta iteração, o conjunto dos vértices brancos é $B - \big(\{p\} \cup Y_1 \cup \cdots \cup Y_j\big)$. Isto prova que (14.2) continua valendo no início da próxima iteração.

No fim do processo iterativo, o invariante (14.2) garante que o conjunto de vértices brancos é $B - \big(\,\{p\}) \cup Y_1 \cup \cdots \cup Y_k\,\big)$, ou seja, $B - X$. Logo, BUSCAEMPROFUNDIDADE cumpre o que prometeu.

Desempenho. A análise da correção da rotina BUSCAEMPROFUNDIDADE permite concluir que o consumo de tempo da rotina é proporcional ao tamanho, $\sum_{x \in X} g(x)$, da instância. Em particular, o consumo de tempo da linha 3 de COMPONENTE não passa de $\sum_{x \in V} g(x)$. Como esta soma é igual ao dobro do número de arestas do grafo, m, podemos dizer que a linha 3 de COMPONENTE consome

$$\mathrm{O}(m)$$

unidades de tempo. No pior caso, o consumo é $\Theta(m)$.

Se levarmos em conta o consumo de tempo das linhas 1–2 de COMPONENTE, teremos um consumo total de

$$\Theta(n + m)$$

unidades de tempo no pior caso.

Exercícios

14.1 Escreva e analise uma versão não recursiva do algoritmo BUSCAEMPROFUNDIDADE.

Apêndice A

Comparação assintótica de funções

É desejável exprimir o consumo de tempo de um algoritmo de uma maneira que não dependa da linguagem de programação, nem dos detalhes de implementação, nem do sistema operacional, nem do computador empregado. Para tornar isto possível, é preciso introduzir um modo grosseiro de comparar funções. (Estou me referindo às funções — no sentido matemático da palavra — que exprimem a dependência entre o consumo de tempo de um algoritmo e o tamanho de sua "entrada".)

Essa comparação grosseira só leva em conta a "velocidade de crescimento" das funções. Assim, ela despreza fatores multiplicativos (pois a função $2n^2$, por exemplo, cresce tão rápido quanto $10n^2$) e despreza valores pequenos do argumento (a função n^2 cresce mais rápido que $100n$, embora n^2 seja menor que $100n$ quando n é pequeno). Dizemos que esta maneira de comparar funções é **assintótica**.

Há três tipos de comparação assintótica: uma com sabor de "$\leq$", outra com sabor de "$\geq$", e uma terceira com sabor de "$=$".

Neste apêndice, o conjunto dos números reais será denotado por $\mathbb{R}$ e o conjunto dos reais positivos será denotado por $\mathbb{R}^{\geq}$.

$\mathbb{R}$
$\mathbb{R}^{\geq}$

A.1 Notação O

Dadas funções F e G de $\mathbb{N}$ em $\mathbb{R}^{\geq}$, dizemos que F **está em** $\mathrm{O}(G)$ se existem c e n_0 em $\mathbb{N}^{>}$ tais que

$$F(n) \;\leq\; c\, G(n)$$

para todo $n \geq n_0$. A mesma coisa pode ser dita assim: existe c em $\mathbb{N}^{>}$ tal que $F(n) \leq c\, G(n)$ para todo n suficientemente grande. Em lugar de "F está em $\mathrm{O}(G)$", podemos dizer "F é $\mathrm{O}(G)$", "$F \in \mathrm{O}(G)$" e até "$F = \mathrm{O}(G)$".

Exemplo 1: $100n$ está em $\mathrm{O}(n^2)$, pois $100n \leq n \cdot n = n^2$ para todo $n \geq 100$.

Exemplo 2: $2n^3+100n$ está em $\mathrm{O}(n^3)$. De fato, para todo $n \geq 1$ temos $2n^3+100n \leq 2n^3 + 100n^3 \leq 102n^3$. Eis outra maneira de provar que $2n^3 + 100n$ está em $\mathrm{O}(n^3)$: para todo $n \geq 10$ tem-se $2n^3 + 100n \leq 2n^3 + n \cdot n \cdot n = 3n^3$.

Exemplo 3: $n^{1.5}$ está em $\mathrm{O}(n^2)$, pois $n^{1.5} \leq n^{0.5} n^{1.5} = n^2$ para todo $n \geq 1$.

Exemplo 4: $n^2/10$ não está em $\mathrm{O}(n)$. Eis uma prova deste fato. Tome qualquer c em $\mathbb{N}^>$. Para todo $n > 10c$ temos $n^2/10 = n \cdot n/10 > 10c \cdot n/10 = cn$. Logo, não é verdade que $n^2/10 \leq cn$ para todo n suficientemente grande.

Exemplo 5: $2n$ está em $\mathrm{O}(2^n)$. De fato, $2n \leq 2^n$ para todo $n \geq 1$, como provaremos por indução em n. Prova: Se $n = 1$, a afirmação é verdadeira pois $2 \cdot 1 = 2^1$. Agora tome $n \geq 2$ e suponha, a título de hipótese de indução, que $2(n-1) \leq 2^{n-1}$. Então $2n \leq 2(n + n - 2) = 2(n-1) + 2(n-1) \leq 2^{n-1} + 2^{n-1} = 2^n$, como queríamos demonstrar.

A seguinte **regra da soma** é útil: se F e F' estão ambas em $\mathrm{O}(G)$ então $F + F'$ também está em $\mathrm{O}(G)$. Eis uma prova da regra. Por hipótese, existem números c e n_0 tais que $F(n) \leq c\,G(n)$ para todo $n \geq n_0$. Analogamente, existem c' e n'_0 tais que $F'(n) \leq c'\,G(n)$ para todo $n \geq n'_0$. Então, para todo $n \geq \max(n_0, n'_0)$, tem-se $F(n) + F'(n) \leq (c + c')\,G(n)$.

Exemplo 6: Como $2n^3$ e $100n$ estão ambas em $\mathrm{O}(n^3)$, a função $2n^3 + 100n$ também está em $\mathrm{O}(n^3)$.

Nossa definição da notação O foi formulada para funções de $\mathbb{N}$ em $\mathbb{R}^{\geq}$, mas ela pode ser estendida, da maneira óbvia, a funções que não estão definidas, ou são negativas, para valores pequenos do argumento.

Exemplo 7: Embora $n^2 - 100n$ não esteja em $\mathbb{R}^{\geq}$ quando $n < 100$, podemos dizer que $n^2 - 100n$ é $\mathrm{O}(n^2)$, pois $n^2 - 100n \leq n^2$ para todo $n \geq 100$.

Exemplo 8: Embora $\lg n$ não esteja definida quando $n = 0$, podemos dizer que $\lg n$ está em $\mathrm{O}(n)$. De fato, se tomarmos o logaritmo da desigualdade $n \leq 2^n$ do Exemplo 5, veremos que $\lg n \leq n$ para todo $n \geq 1$.

Exemplo 9: $n \lg n$ está em $\mathrm{O}(n^2)$. Segue imediatamente do Exemplo 8.

Exercícios

A.1 Critique a afirmação "$n^2 - 100n$ está em $\mathrm{O}(n^2)$ para todo $n \geq 100$".

A.2 Critique a afirmação "$2n^3 + 100n$ está em $\mathrm{O}(n^3)$ para $c = 3$ e todo $n \geq 100$".

A.3 Mostre que $\lg n$ está em $\mathrm{O}(n^{0.5})$.

A.2 Notação Ômega

Dadas funções F e G de $\mathbb{N}$ em $\mathbb{R}^{\geq}$, dizemos que F **está em** $\Omega(G)$ se existe c em $\mathbb{N}^>$ tal que

$$F(n) \;\geq\; \frac{1}{c}\,G(n)$$

para todo n suficientemente grande. É claro que Ω é "inversa" de O, ou seja, uma função F está em $\Omega(G)$ se e somente se G está em $\mathrm{O}(F)$.

Exemplo 10: $n^3 + 100n$ está em $\Omega(n^3)$, pois $n^3 + 100n \geq n^3$ para todo n.

A definição da notação Ω pode ser estendida, da maneira óbvia, a funções F ou G que estão definidas e são não negativas apenas para valores grandes de n.

Exemplo 11: $n^2 - 2n$ é $\Omega(n^2)$. De fato, temos $2n \leq \frac{1}{2}n^2$ para todo $n \geq 4$ e portanto $n^2 - 2n \geq n^2 - \frac{1}{2}n^2 = \frac{1}{2}n^2$.

Exemplo 12: $n \lg n$ está em $\Omega(n)$. De fato, para todo $n \geq 2$ tem-se $\lg n \geq 1$ e portanto $n \lg n \geq n$.

Exemplo 13: $100n$ não está em $\Omega(n^2)$. Tome qualquer c em $\mathbb{N}^>$. Para todo $n > 100c$, tem-se $100 < \frac{1}{c}n$ e portanto $100n < \frac{1}{c}n^2$.

Exemplo 14: n não é $\Omega(2^n)$. Basta mostrar que para qualquer c em $\mathbb{N}^>$ tem-se $n < \frac{1}{c}2^n$ para todo n suficientemente grande. Como todo c é menor que alguma potência de 2, basta mostrar que, para qualquer k em $\mathbb{N}^>$, tem-se $n \leq 2^{-k}2^n$ para todo n suficientemente grande. Especificamente, mostraremos que $n \leq 2^{n-k}$ para todo $n \geq 2k$.

A prova é por indução em n. Se $n = 2k$, temos $k \leq 2^{k-1}$ conforme o Exemplo 5 com k no papel de n, e portanto $n = 2k \leq 2 \cdot 2^{k-1} = 2^k = 2^{2k-k} = 2^{n-k}$. Agora tome $n \geq 2k + 1$ e suponha, a título de hipótese de indução, que $n - 1 \leq 2^{n-1-k}$. Então $n \leq n + n - 2 = n - 1 + n - 1 = 2 \cdot (n - 1) \leq 2 \cdot 2^{n-1-k} = 2^{n-k}$, como queríamos demonstrar.

Exercícios

A.4 Mostre que 2^{n-1} está em $\Omega(2^n)$.

A.5 Mostre que $\lg n$ não é $\Omega(n)$.

A.3 Notação Teta

Dizemos que F **está em** $\Theta(G)$ se F está em $\mathrm{O}(G)$ e também em $\Omega(G)$, ou seja, se existem c e c' em $\mathbb{N}^>$ tais que

$$\frac{1}{c}\, G(n) \;\leq\; F(n) \;\leq\; c'\, G(n)$$

para todo n suficientemente grande. Se F está em $\Theta(G)$, diremos que F é **proporcional** a G, ainda que isto seja um abuso do sentido usual de "proporcional."

Exemplo 15: Para qualquer a em $\mathbb{R}^{\geq}$ e qualquer b em $\mathbb{R}$, a função $an^2 + bn$ está em $\Theta(n^2)$.

Exercícios

A.6 Mostre que $n(n + 1)/2$ e $n(n - 1)/2$ estão em $\Theta(n^2)$.

A.4 Consumo de tempo de algoritmos

Seja $\mathcal{A}$ um algoritmo para um problema cujas instâncias têm tamanho n. Se a função que mede o consumo de tempo de $\mathcal{A}$ no pior caso está em $\mathrm{O}(n^2)$, por exemplo, podemos

usar expressões como

$$\text{“}\mathcal{A} \text{ consome } O(n^2) \text{ unidades de tempo no pior caso”}$$

e “$\mathcal{A}$ consome tempo $O(n^2)$ no pior caso”. Podemos usar expressões semelhantes com Ω ou Θ no lugar de O e com “melhor caso” no lugar de “pior caso”.

(Se $\mathcal{A}$ consome $O(n^2)$ unidades de tempo no pior caso, também consome $O(n^2)$ em qualquer caso. Analogamente, se consome $\Omega(n^2)$ no melhor caso, também consome $\Omega(n^2)$ em qualquer caso. Mas não podemos dispensar a cláusula “no pior caso” em uma afirmação como “$\mathcal{A}$ consome $\Omega(n^2)$ unidades de tempo no pior caso”.)

Linear, linearítmico, quadrático. Um algoritmo é **linear** se consome $\Theta(n)$ unidades de tempo no pior caso. É fácil entender o comportamento de um tal algoritmo: quando o tamanho da “entrada” dobra, o algorimo consome duas vezes mais tempo; quando o tamanho é multiplicado por uma constante c, o consumo de tempo também é multiplicado por c. Algoritmos lineares são considerados muito rápidos.

Um algoritmo é **linearítmico** (ou **ene-log-ene**) se consome $\Theta(n \lg n)$ unidades de tempo no pior caso. Se o tamanho da “entrada” dobra, o consumo dobra e é acrescido de $2n$; se o tamanho é multiplicado por uma constante c, o consumo é multiplicado por c e acrescido de um pouco mais que cn.

Um algoritmo é **quadrático** se consome $\Theta(n^2)$ unidades de tempo no pior caso. Se o tamanho da “entrada” dobra, o consumo quadruplica; se o tamanho é multiplicado por uma constante c, o consumo é multiplicado por c^2.

Polinomial e exponencial. Um algoritmo é **polinomial** se consome $O(n^k)$ unidades de tempo no pior caso, sendo k um número natural. Por exemplo, é polinomial qualquer algoritmo que consome $O(n^{100})$ unidades de tempo. Apesar desse exemplo, algoritmos polinomiais são considerados rápidos.

Um algoritmo é **exponencial** se consome $\Omega(a^n)$ unidades de tempo no pior caso, sendo a um número real maior que 1. Por exemplo, é exponencial qualquer algoritmo que consome $\Omega(1.1^n)$ unidades de tempo. Se n é multiplicado por 10, por exemplo, o consumo de tempo de um tal algoritmo é elevado à potência 10. Algoritmos exponenciais não são polinomiais.

Exercícios

A.7 Suponha que dois algoritmos, $\mathcal{A}$ e $\mathcal{B}$, resolvem um mesmo problema. Suponha que o tamanho das instâncias do problema é medido por um parâmetro n. Em quais dos seguintes cenários podemos afirmar que $\mathcal{A}$ é mais rápido que $\mathcal{B}$? Cenário 1: No pior caso, $\mathcal{A}$ consome $O(\lg n)$ unidades de tempo e $\mathcal{B}$ consome $O(n^3)$ unidades de tempo. Cenário 2: No pior caso, $\mathcal{A}$ consome $O(n^2 \lg n)$ unidades de tempo e $\mathcal{B}$ consome $\Omega(n^2)$ unidades de tempo. Cenário 3: No pior caso, $\mathcal{A}$ consome $O(n^2)$ unidades de tempo e $\mathcal{B}$ consome $\Omega(n^2 \lg n)$ unidades de tempo.

Apêndice B

Solução de recorrências

A habilidade de resolver recorrências é importante para a análise do consumo de tempo de algoritmos recursivos. Uma *recorrência* é uma fórmula que define uma função, digamos F, em termos dela mesma. Mais precisamente, define $F(n)$ em termos de $F(n-1)$, $F(n-2)$, $F(n-3)$, etc.

Uma *solução* de uma recorrência é uma fórmula que exprime $F(n)$ diretamente em termos de n. Para as aplicações à análise de algoritmos, uma fórmula exata para $F(n)$ não é necessária: basta mostrar que F está em $\Theta(G)$ para alguma função G definida explicitamente em termos de n.

Neste apêndice, $\mathbb{R}$ denota o conjunto dos números reais e $\mathbb{R}^{\geq}$. denota o conjunto dos reais positivos.

$\mathbb{R}$
$\mathbb{R}^{\geq}$

B.1 Exemplo 1

øconjunto dos números reais será denotado por $\mathbb{R}$. labelsec:recurrences:example10

$\mathbb{R}$

Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^{\geq}$. Suponha que $F(1) = 1$ e F satisfaz a recorrência

$$F(n) = 2F(n-1) + 1 \tag{B.1}$$

para todo $n \geq 2$. Eis alguns valores da função: $F(2) = 3$, $F(3) = 7$, $F(4) = 15$. É fácil "desenrolar" a recorrência:

$$\begin{aligned} F(n) &= 2F(n-1) + 1 \\ &= 2(2F(n-2) + 1) + 1 \\ &= 4F(n-2) + 3 \\ &= \cdots \\ &= 2^j F(n-j) + 2^j - 1 \\ &= 2^{n-1} F(1) + 2^{n-1} - 1 \,. \end{aligned}$$

Concluímos assim que

$$F(n) = 2^n - 1 \tag{B.2}$$

para todo $n \geq 1$. (Não temos informações sobre o valor de $F(0)$.) Portanto, F está em $\Theta(2^n)$.

Exercícios

B.1 Confira a fórmula (B.2) por indução em n.

B.2 Sejam a, b e n_0 três números naturais e suponha que $F(n_0) = a$ e $F(n) = 2F(n-1) + b$ para todo $n > n_0$. Mostre que F está em $\Theta(2^n)$.

B.2 Exemplo 2

Suponha que uma função F de $\mathbb{N}$ em $\mathbb{R}^{\geq}$ satisfaz a seguinte recorrência:

$$F(n) = F(n-1) + n \tag{B.3}$$

para todo $n \geq 2$. Suponha ainda que $F(1) = 1$. Teremos, em particular, $F(2) = F(1) + 2 = 3$ e $F(3) = F(2) + 3 = 6$.

A recorrência pode ser facilmente "desenrolada": $F(n) = F(n-1) + n = F(n-2) + (n-1) + n = \cdots = F(1) + 2 + 3 + \cdots + (n-1) + n = 1 + 2 + 3 + \cdots + (n-1) + n$. Concluímos assim que

$$F(n) = \tfrac{1}{2}(n^2 + n) \tag{B.4}$$

para todo $n \geq 1$. (Não temos informações sobre o valor de $F(0)$.) Portanto, F está em $\Theta(n^2)$.

Recorrências semelhantes. O valor de $F(1)$ tem pouca influência sobre o resultado. Se tivéssemos $F(1) = 10$, por exemplo, a fórmula (B.4) seria substituída por $F(n) = \frac{1}{2}(n^2 + n) + 9$, e esta função também está em $\Theta(n^2)$.

O ponto a partir do qual (B.3) começa a valer também tem pouca influência sobre a solução da recorrência. Se (B.3) valesse apenas a partir de $n = 5$, por exemplo, teríamos $F(n) = \frac{1}{2}(n^2 + n) + F(4) - 10$ no lugar de (B.4), e esta função também está em $\Theta(n^2)$.

Exercícios

B.3 Confira (B.4) por indução em n.

B.4 Suponha que F satisfaz a recorrência $F(n) = F(n-1) + 3n + 2$ para $n = 2, 3, 4, \ldots$ Mostre que se $F(1) = 1$ então $F(n) = 3n^2/2 + 7n/2 - 4$ para todo $n \geq 2$. Conclua que F está em $\Theta(n^2)$.

B.5 Sejam a, b e c três números naturais e suponha que F satisfaz as seguintes condições: $F(n_0) = a$ e $F(n) = F(n-1) + bn + c$ para todo $n > n_0$. Mostre que F está em $\Theta(n^2)$.

B.6 Suponha que $F(n) = F(n-1) + 7$ para $n = 2, 3, 4, \ldots$ Mostre que se $F(1) = 1$ então $F(n) = 7n - 6$ para todo $n \geq 1$. Conclua que F está em $\Theta(n)$.

B.3 Exemplo 3

Seja F uma função que leva $\mathbb{N}$ em $\mathbb{R}^{\geq}$. Suponha que F satisfaz a recorrência

$$F(n) = 2F(\lfloor n/2 \rfloor) + n \tag{B.5}$$

para todo $n \geq 2$. (Não faz sentido trocar "$\lfloor n/2 \rfloor$" por "$n/2$" pois $n/2$ não está em $\mathbb{N}$ quando n é ímpar.) Suponha ainda que $F(1) = 1$ (e portanto $F(2) = 4$, $F(3) = 5$, etc.)

É mais fácil entender (B.5) quando n é uma potência de 2, pois nesse caso $\lfloor n/2 \rfloor$ se reduz a $n/2$. Se $n = 2^j$, temos

$$\begin{aligned} F(2^j) &= 2F(2^{j-1}) + 2^j \\ &= 2^2 F(2^{j-2}) + 2^1 2^{j-1} + 2^j \\ &= 2^3 F(2^{j-3}) + 2^2 2^{j-2} + 2^1 2^{j-1} + 2^j \\ &= 2^j F(2^0) + 2^{j-1} 2^1 + \cdots + 2^2 2^{j-2} + 2^1 2^{j-1} + 2^0 2^j \\ &= 2^j 2^0 + 2^{j-1} 2^1 + \cdots + 2^2 2^{j-2} + 2^1 2^{j-1} + 2^0 2^j \\ &= (j+1) 2^j \\ &= j 2^j + 2^j \, . \end{aligned} \tag{B.6}$$

Para ter certeza, podemos conferir esta fórmula por indução em j. Se $j = 1$, temos $F(2^j) = 4 = (j+1)2^j$. Agora tome $j \geq 2$. De acordo com (B.5), $F(2^j) = 2F(2^{j-1}) + 2^j$. Por hipótese de indução, $F(2^{j-1}) = 2^{j-1} j$. Logo, $F(2^j) = 2 \cdot 2^{j-1} j + 2^j = 2^j j + 2^j$, como queríamos mostrar.

Como $2^j = n$, a expressão (B.6) pode ser reescrita assim:

$$F(n) = n \lg n + n \tag{B.7}$$

quando n é potência de 2. Embora esta fórmula só se aplique às potências de 2, podemos usá-la para obter cotas superior e inferior válidas para todo n suficientemente grande. O primeiro passo nessa direção é verificar que F é estritamente crescente, ou seja, que

$$F(n) < F(n+1) \tag{B.8}$$

para todo $n \geq 1$. A prova é uma indução em n. Se $n = 1$, temos $F(n) = 1 < 4 = F(n+1)$. Agora tome $n \geq 2$. Se n é par então $F(n) < F(n) + 1 = 2F(\frac{n}{2}) + n + 1 = 2F(\lfloor \frac{n+1}{2} \rfloor) + n + 1 = F(n+1)$, em virtude de (B.5). Agora suponha n ímpar. Por hipótese de indução, $F(\frac{n-1}{2}) \leq F(\frac{n-1}{2} + 1) = F(\frac{n+1}{2})$. Logo, por (B.5),

$$F(n) = 2F(\tfrac{n-1}{2}) + n \leq 2F(\tfrac{n+1}{2}) + n < 2F(\tfrac{n+1}{2}) + n + 1 = F(n+1) \, .$$

Mostramos assim que F é estritamente crescente.

Agora podemos calcular uma boa cota superior para F a partir de (B.7). Mostraremos que

$$F(n) \leq 6n \lg n \tag{B.9}$$

para todo $n \geq 2$. Tome o único j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Em virtude de (B.8), $F(n) < F(2^{j+1})$. Em virtude de (B.6), $F(2^{j+1}) = (j+2)2^{j+1} \leq 3j2^{j+1} = 6j2^j$. Como $j \leq \lg n$, temos $6j2^j \leq 6n \lg n$.

Uma boa cota inferior para F também pode ser calculada a partir de (B.7): mostraremos que

$$F(n) \geq \tfrac{1}{2} n \lg n \tag{B.10}$$

para todo $n \geq 2$. Tome o único j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Em virtude de (B.8) e (B.6), temos $F(n) \geq F(2^j) = (j+1)2^j = \frac{1}{2}(j+1)2^{j+1}$. Como $\lg n < j+1$, temos $\frac{1}{2}(j+1)2^{j+1} > \frac{1}{2} n \lg n$.

De (B.9) e (B.10), concluímos que F está em $\Theta(n \lg n)$.

Recorrências semelhantes. O ponto n_0 a partir do qual a recorrência (B.5) começa a valer, bem como os valores de $F(1)$, $F(2)$, ..., $F(n_0 - 1)$, têm pouca influência sobre a solução da recorrência: quaisquer que sejam esses valores iniciais, F estará em $\Theta(n \lg n)$. Se (B.5) vale apenas para $n \geq 3$, por exemplo, teremos $n \lg n + \frac{F(2)-2}{2} n$ no lugar de (B.7).

A parcela n na expressão $2F(\lfloor n/2 \rfloor) + n$ também pode ser ligeiramente alterada sem que F saia de $\Theta(n \lg n)$. Assim, quaisquer que sejam $c > 0$ e d, se $F(n) = 2F(\lfloor n/2 \rfloor) + cn + d$ então F está em $\Theta(n \lg n)$.

Finalmente, podemos trocar "$2F(\lfloor n/2 \rfloor)$" por "$2F(\lceil n/2 \rceil)$" ou até mesmo por "$F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$" sem que F saia de $\Theta(n \lg n)$.

Mas o fator 2 que multiplica "$F(\lfloor n/2 \rfloor)$" é crítico: se este fator for alterado, a função F deixará de pertencer a $\Theta(n \lg n)$.

Exercícios

B.7 Suponha que um função F de $\mathbb{N}$ em $\mathbb{R}^{\geq}$ satisfaz a recorrência $F(n) = F(\lfloor n/2 \rfloor) + 1$ para todo $n \geq 2$. Mostre que F está em $\Theta(\lg n)$.

B.8 Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^{\geq}$ tal que $F(n) = 2F(\lfloor n/2 \rfloor) + 1$ para todo $n \geq 2$. Mostre que F está em $\Theta(n)$.

B.9 Seja F um função de $\mathbb{N}$ em $\mathbb{R}^{\geq}$ tal que $F(n) = 4F(\lfloor n/2 \rfloor) + n$ quando $n \geq 2$. Mostre que F está em $\Theta(n^2)$. (Sugestão: mostre que $\frac{1}{4} n^2 \leq F(n) \leq 8n^2$ para todo n suficientemente grande.)

B.4 Exemplo 4

Seja F um função de $\mathbb{N}$ em $\mathbb{R}^{\geq}$. Suponha que

$$F(n) = 3F(\lfloor n/2 \rfloor) + n \tag{B.11}$$

para todo $n \geq 2$. Suponha ainda que $F(1) = 1$. Para começar a entender o comportamento de F, considere os valores de n que são potências de 2. Quando $n = 2^j$, temos

$$\begin{aligned}
F(2^j) &= 3F(2^{j-1}) + 2^j \\
&= 3^2 F(2^{j-2}) + 3^1 2^{j-1} + 2^j \\
&= 3^3 F(2^{j-3}) + 3^2 2^{j-2} + 3^1 2^{j-1} + 2^j \\
&= 3^j F(2^0) + 3^{j-1} 2^1 + \cdots + 3^2 2^{j-2} + 3^1 2^{j-1} + 3^0 2^j \\
&= 3^j 2^0 + 3^{j-1} 2^1 + \cdots + 3^2 2^{j-2} + 3^1 2^{j-1} + 3^0 2^j \\
&= 2^j \left((3/2)^j + (3/2)^{j-1} + \cdots + (3/2)^2 + (3/2)^1 + (3/2)^0 \right)
\end{aligned}$$

$$
\begin{aligned}
&= 2^j \, \frac{(3/2)^{j+1} - 1}{3/2 - 1} \\
&= 2^{j+1}\big((3/2)^{j+1} - 1\big) \\
&= 3^{j+1} - 2^{j+1} \,. \qquad \text{(B.12)}
\end{aligned}
$$

Observe que $3^j = (2^{\lg 3})^j = (2^j)^{\lg 3} = n^{\lg 3}$. (A título de curiosidade, $\lg 3$ fica entre 1.584 e 1.585.) Portanto, a fórmula (B.12) pode ser reescrita assim:

$$F(n) = 3n^{\lg 3} - 2n$$

para toda potência n de 2. Embora esta fórmula só valha para as potências de 2, podemos usá-la para obter cotas superior e inferior válidas para todo n suficientemente grande. A primeira providência nessa direção é certificar-se de que F é estritamente crescente:

$$F(n) < F(n+1)$$

para todo $n \geq 1$. A prova é análoga à de (B.8). Agora estamos em condições de calcular uma boa cota superior para F:

$$F(n) \leq 9n^{\lg 3} \qquad \text{(B.13)}$$

para todo $n \geq 1$. Tome j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Como F é estritamente crescente, (B.12) garante que $F(n) < F(2^{j+1}) = 3^{j+2} - 2^{j+2} \leq 3^{j+2} = 9 \cdot (2^j)^{\lg 3} \leq 9n^{\lg 3}$, como queríamos mostrar. Também podemos calcular uma boa cota inferior:

$$F(n) \geq \tfrac{1}{3} n^{\lg 3} \qquad \text{(B.14)}$$

para todo $n \geq 1$. Tome j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Como F é crescente, (B.12) garante que $F(n) \geq F(2^j) = 3^{j+1} - 2^{j+1} = 3^j + 2 \cdot 3^j - 2 \cdot 2^j \geq 3^j = \frac{1}{3} 3^{j+1} = \frac{1}{3}(2^{j+1})^{\lg 3} > \frac{1}{3} n^{\lg 3}$, como queríamos demonstrar.

As relações (B.13) e (B.14) mostram que F está em $\Theta(n^{\lg 3})$.

Recorrências semelhantes. Se (B.11) valesse apenas quando $n \geq n_0$, a função F continuaria em $\Theta(n^{\lg 3})$ quaisquer que fossem os valores de $F(\lfloor n_0/2 \rfloor)$, $F(\lfloor n_0/2 \rfloor + 1)$, $\ldots$, $F(n_0 - 1)$.

Além disso, F continuará em $\Theta(n^{\lg 3})$ se (B.11) for substituída por $F(n) = 3F(\lfloor n/2 \rfloor) + cn + d$, quaisquer que sejam $c > 0$ e d. Também continuará em $\Theta(n^{\lg 3})$ se "$3F(\lfloor n/2 \rfloor)$" for trocado por "$2F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil)$" ou até mesmo por "$2F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil + 1)$".

Exercícios

B.10 Confira a formula (B.12) por indução em j.

B.5 Teorema mestre

O seguinte teorema dá a solução de muitas recorrências semelhantes às das Seções B.3 e B.4. Sejam a, k e c números em $\mathbb{N}^>$, $\mathbb{N}$ e $\mathbb{R}^\geq$ respectivamente. Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^\geq$ tal que

$$F(n) = a\,F\big(\frac{n}{2}\big) + cn^k \tag{B.15}$$

para $n = 2^1, 2^2, 2^3, \ldots$ Suponha ainda que F é assintoticamente crescente, ou seja, que existe n_1 tal que $F(n) \leq F(n+1)$ para todo $n \geq n_1$. Nessas condições,

$$\text{se } \lg a > k \text{ então } F \text{ está em } \Theta(n^{\lg a}), \tag{B.16}$$

$$\text{se } \lg a = k \text{ então } F \text{ está em } \Theta(n^k \lg n), \tag{B.17}$$

$$\text{se } \lg a < k \text{ então } F \text{ está em } \Theta(n^k). \tag{B.18}$$

(Recorrências como (B.15) aparecem na análise de algoritmos baseados no método da divisão e conquista: dada uma instância de tamanho n, divida a instância em a partes de tamanho $n/2$, resolva essas subinstâncias, e combine as soluções em tempo limitado por cn^k.)

Generalização. O teorema pode ser generalizado como segue. Sejam $a \geq 1$, $b \geq 2$, $k \geq 0$ e $n_0 \geq 1$ números em $\mathbb{N}$ e seja c um número em $\mathbb{R}^\geq$. Seja F é uma função de $\mathbb{N}$ em $\mathbb{R}^\geq$ tal que

$$F(n) = a\,F\big(\frac{n}{b}\big) + cn^k \tag{B.19}$$

para $n = n_0b^1, n_0b^2, n_0b^3, \ldots$ Suponha ainda que F é assintoticamente crescente. Nessas condições,

$$\text{se } \lg a/\lg b > k \text{ então } F \text{ está em } \Theta(n^{\lg a/\lg b}), \tag{B.20}$$

$$\text{se } \lg a/\lg b = k \text{ então } F \text{ está em } \Theta(n^k \lg n), \tag{B.21}$$

$$\text{se } \lg a/\lg b < k \text{ então } F \text{ está em } \Theta(n^k). \tag{B.22}$$

A prova deste teorema pode ser encontrada na Seção 4.7 do livro de Brassard e Bratley [2] e na Seção 4.4 do livro de Cormen *et al.* [3].

Exemplo A. Seja c um número em $\mathbb{R}^\geq$ e seja F uma função de $\mathbb{N}$ em $\mathbb{R}^\geq$ tal que $F(n) = F(\lfloor n/2 \rfloor) + 2F(\lceil n/2 \rceil) + cn$ para todo $n \geq 2$. Nessas condições, F é crescente e portanto (B.16) garante que F está em $\Theta(n^{\lg 3})$. (Compare com o resultado da Seção B.4.)

Exemplo B. Sejam a e k números em $\mathbb{N}$ tais que $a > 2^k$ e seja c um número em $\mathbb{R}^\geq$. Seja F é uma função de $\mathbb{N}$ em $\mathbb{R}^\geq$ tal que $F(n) = aF\big(\lfloor n/2 \rfloor\big) + cn^k$ para todo $n \geq 1$. Nessas condições, F é crescente e portanto (B.16) garante que F está em $\Theta(n^{\lg a})$.

Notas bibliográficas

Este capítulo foi baseada na Seção 4.7 do livro de Brassard e Bratley [2]. Veja também o verbete *Master theorem* na Wikipedia.

Posfácio

O texto fez uma modesta introdução à Análise de Algoritmos usando material dos excelentes livros citados na bibliografia. Embora tenha tratado apenas de problemas e algoritmos muito simples, espero que este minicurso possa guiar o leitor que queira analisar os seus próprios algoritmos.

O tratamento de algoritmos probabilísticos foi muito superficial e muitos outros assuntos ficaram de fora por falta de espaço. Assim, não foi possível discutir a análise de tipos abstratos de dados, como filas de prioridades e estruturas *union/find*. A *análise amortizada* (muito útil para estimar o consumo de tempo dos algoritmos de fluxo em redes, por exemplo) foi igualmente ignorada. Finalmente, não foi possível mencionar a teoria da complexidade de problemas e a questão P=NP.

Bibliografia

[1] J.L. Bentley. *Programming Pearls*. ACM Press, second edition, 2000. 7, 30, 35

[2] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, 1996. 7, 44, 58, 62, 88, 89

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, second edition, 2001. 7, 13, 50, 54, 58, 59, 63, 88

[4] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani. *Algorithms*. McGraw-Hill, 2006. 44

[5] A.C.P.L. de Carvalho and T. Kowaltowski, editors. *Atualizações em Informática 2009*. SBC (Soc. Bras. de Computação). PUC-Rio, 2009. ISBN 978-85-87926-54-8. 7

[6] P. Feofiloff. Algoritmos de Programação Linear. Internet: https://www.ime.usp.br/~pf/prog-lin/ e http://arquivoescolar.org/handle/arquivo-e/72, 1999. 206 páginas. 66

[7] P. Feofiloff. Análise de Algoritmos. Internet: https://www.ime.usp.br/~pf/analise_de_algoritmos/, 2000–2011.

[8] C.G. Fernandes, F.K. Miyazawa, M. Cerioli, and P. Feofiloff, editors. *Uma Introdução Sucinta a Algoritmos de Aproximação*. XXIII Colóquio Brasileiro de Matemática. IMPA (Instituto de Matemática Pura e Aplicada), Rio de Janeiro, 2001. Internet: https://www.ime.usp.br/~cris/aprox/. 61

[9] J. Hromkovič. *Algorithmics for Hard Problems: Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics*. Springer, second edition, 2004.

[10] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley, 2005. 7, 44, 50, 66

[11] D.E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1997. 44

[12] D.E. Knuth. *Selected Papers on Analysis of Algorithms*. CSLI Lecture Notes, no. 102. Center for the Study of Language and Information (CSLI), Stanford, CA, 2000. 9

[13] D.E. Knuth. *The Art of Computer Programming*. Addison-Wesley, ca. 1973. Several volumes.

[14] U. Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, 1989. 7, 35

[15] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005. 70

[16] R. Neapolitan and K. Naimipour. *Foundations of Algorithms*. Jones and Bartlett, second edition, 1998.

[17] I. Parberry. *Problems on Algorithms*. Prentice Hall, 1995. 54

[18] I. Parberry and W. Gasarch. *Problems on Algorithms*. Internet: http://www.eng.unt.edu/ian/books/free/, second edition, 2002. 54

[19] R. Sedgewick. *Algorithms in C, Part5: Graph Algorithms*. Addison-Wesley, third edition, 2000. 66

[20] J.L. Szwarcfiter and L. Markenzon. *Estruturas de Dados e seus Algoritmos*. Livros Técnicos e Científicos, second edition, 1994.

[21] N. Ziviani. *Projeto de Algoritmos (com implementações em Pascal e C)*. Thomson, second edition, 2004.

Índice remissivo

$\lfloor x \rfloor$, 12
$\lceil x \rceil$, 12
O( ), 11, 79
Ω( ), 11, 80
Θ( ), 11, 81
ℕ, 12
ℝ, 79, 83
$\mathbb{R}^{\geq}$, 79, 83
ℤ, 12

algoritmo, 9
  aleatorizado, 70
  correto, 9
  cúbico, 12
  de aproximação, 61
  de Karatsuba, 41
  de Strassen, 44
  de tempo constante, 12
  ene-log-ene, 12, 82
  exponencial, 82
  guloso, 50, 58
  linear, 11, 82
  inearítmico, 12, 82
  logarítmico, 12
  Mergesort, 18
  polinomial, 82
  primal-dual, 66
  probabilístico, 70
  quadrático, 12, 82
  recursivo, 13, 18, 39, 53, 56, 75
algoritmo recursivo, 13
altura (de vetor), 23
análise assintótica, 79
aproximação (algoritmo de), 61
aresta (de grafo), 63
assinltoticamente crescente, 88
assintótico, 11, 79

Bentley, 7, 30, 35
BFS, 71
Brassard, 7, 44, 58, 62, 88, 89
Bratley, 7, 44, 58, 62, 88, 89
breadth-first search, 71
busca
  em largura, 71
  em profundidade, 75

C, 13
caminho (em grafo), 71
cobertura (de grafo), 63
compatíveis (intervalos), 45
componente (de grafo), 71
conexo (grafo), 73
conjunto independente (em grafo), 67
consumo de tempo, 10
Cormen, Leiserson, Rivest, Stein, 7, 13, 50, 54, 58, 59, 63, 88
correção (de algoritmo), 9
crescente, 12
crescente, 88

Dasgupta, 44
decrescente, 12
depth-first search, 75
desempenho, 10
DFS, 75
dígito, 38
dígitos (número de), 38
divisão e conquista, 21, 27, 44, 88
dominante (termo), 11

eficiência, 10
estritamente
  crescente, 12
  decrescente, 12
  negativo, 12
  positivo, 12
estrutura recursiva
  de um problema, 13, 29, 30, 52, 53, 55

fortemente polinomial, 58

Gasarch, 54
grafo, 63
    bipartido, 66
    conexo, 73
grau de vértice, 73
guloso, 48
guloso (algoritmo), 50, 58

indução matemática, 13, 14, 18, 80, 84
instância de problema, 9
intercalação de vetores, 18
intervalo, 45
intervalos disjuntos, 46
invariante, 26, 29, 33, 53, 54, 57, 65, 72, 77

Java, 13

Karatsuba, 41
Kleinberg, 7, 44, 50, 66
Knuth, 9, 44

lg n, 12

majoritário, 31
Manber, 7, 35
maximal, 68
máximo, 46
melhor caso, 10
Mergesort, 18
minimal, 64
mínimo, 64
Mitzenmacher, 70
mochila
    de valor máximo, 55
    de valor quase máximo, 59
método da interpolação, 43
multiplicação de números, 37

$\mathbb{N}$, 12
negativo, 12
notação
    O grande, 11, 79
    Ômega grande, 11, 80
    Teta grande, 11, 81
NP-difícil, 58, 59, 63
número de dígitos, 38
números
    naturais, 12
    reais, 79, 83

O( ), 11, 79
Ω( ), 11, 80

O grande, 11, 79
Ômega (Ω) grande, 11, 80
Ofman, 44

palavra, 51
Papadimitriou, 44
parágrafo, 51
Parberry, 54
pen drive, 58
pior caso, 10
piso de número, 12
pontas (de aresta), 63
positivo, 12
primal-dual, 66
problema
    da maioria, 31
    da ordenação, 17
    do segmento de soma máxima, 24
    dos intervalos disjuntoss, 46
programação dinâmica, 28, 30, 53, 56
programação linear, 66
proporcional, 81
prova por
    contradição, 80, 81
    indução matemática, 13, 14, 18, 80, 84

$\mathbb{R}$, 79, 83
$\mathbb{R}^{\geq}$, 79, 83
recorrência, 14, 83
recursão, 13
recursivo (algoritmo), 13, 18, 39, 53, 56, 75
resolve, 10

Sedgewick, 66
segmento (de vetor), 23
semialtura (de vetor), 29
solução de recorrência, 14
Strassen, 44
suficientemente grande, 79

Θ( ), 11, 81
tamanho de instância, 10
Tardos, 7, 44, 50, 66
tem n dígitos, 38
tempo (consumo de), 10
teorema mestre, 88
termo dominante, 11
Teta (Θ) grande, 11, 81
teto de número, 12

unidade de tempo, 10, 82
Upfal, 70

valor específico, 59
Vazirani, 44
vértice (de grafo), 63
vetor
  crescente, 12
  decrescente, 12
  estritamente crescente, 12
  estritamente decrescente, 12
viável, 45, 55, 59
vizinhos (vértices), 71

$\mathbb{Z}$, 12