André C. P. L. F. de Carvalho

Tomasz Kowaltowski

Organizadores

Atualizações em Informática 2007

PUC Rio

Capítulo 1

Análise de Algoritmos

Paulo Feofiloff

Abstract

Analysis of Algorithms is a well-established subject, covered by many excellent textbooks (mostly in English). The present text is a short introduction to the subject. It briefly deals with some prerequisites (such as asymptotic notation and recurrences) and then goes on to analyse a few classical algorithms and discuss their efficiency. It also comments on the strategies (such as divide-and-conquer, dynamic programming, and primal-dual) used to develop the algorithms.

Resumo

A Análise de Algoritmos é uma disciplina bem-estabelecida, coberta por um bom número de excelentes livros (a maioria em inglês). O presente texto é uma introdução sucinta ao assunto. Depois de tratar brevemente de alguns pré-requisitos (como notação assintótica e resolução de recorrências), o texto analisa alguns algoritmos clássicos, discute sua eficiência, e chama a atenção para as estratégias (divisão e conquista, programação dinâmica, método primal-dual) que levaram à sua concepção.

1.1. Introdução

A Análise de Algoritmos[1] estuda certos problemas computacionais recorrentes, problemas que aparecem, sob diversos disfarces, em uma grande variedade de aplicações e de contextos. A análise de um algoritmo para um dado problema trata de

1. provar que o algoritmo está correto e
2. estimar o tempo que a execução do algoritmo consome.

(A estimativa do espaço de memória usado pelo algoritmo também é importante em muitos casos.) Dados dois algoritmos para um mesmo problema, a análise permite decidir qual dos dois é mais eficiente.

[1] A expressão foi cunhada por D. E. Knuth.

Pode-se dizer que a Análise de Algoritmos é uma disciplina de engenharia, pois ela 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 paradigmas de projeto de algoritmos (como a divisão e conquista, a programação dinâmica, o método primal-dual, etc.)

Este texto é uma breve introdução à Análise de Algoritmos, baseada nos excelentes livros de Cormen et al., Brassard et al., Bentley, Kleinberg et al. e Manber. Outros livros sobre o assunto estão relacionados na seção de referências bibliográficas.

No restante desta introdução, faremos uma rápida revisão de conceitos básicos e fixaremos a notação e a terminologia empregadas no texto.

1.1.1. Problemas e instâncias

Da mesma forma que distinguimos um algoritmo de sua aplicação a uma particular "entrada", convém distinguir problemas de suas instâncias. Todo problema computacional é uma coleção de **instâncias**.[2] Cada instância do problema é definida por um particular conjunto de dados.

Considere, por exemplo, o problema de encontrar a média dos elementos de um vetor $A[1..n]$ de números. Uma das instâncias deste problema consiste em encontrar a média dos elementos do vetor $(876, 145, 323, 112, 221)$.

Em discussões informais, é razoável confundir os conceitos e dizer "problema" quando "instância do problema" seria mais correto. Em outras ocasiões, é importante manter a distinção clara.

O **tamanho** de uma instância de um problema é a quantidade de dados necessária para descrever a instância. O tamanho de uma instância é descrito, em geral, por um só número natural, mas às vezes é conveniente usar dois ou até mais números. A ideia de tamanho permite dizer que uma instância é menor ou maior que outra.

No problema da média citado acima, por exemplo, é razoável dizer que o tamanho da instância $(876, 145, 323, 112, 221)$ é 5 e, em geral, que o tamanho de uma instância $A[1..n]$ é n. (Dependendo das circunstâncias, entretanto, pode ser mais apropriado adotar o número total de dígitos decimais de $A[1..n]$ como tamanho da instância. Nesse caso, a instância $(876, 145, 323, 112, 221)$ terá tamanho 15.)

1.1.2. Consumo de tempo de algoritmos

Dizemos que um algoritmo resolve um problema se, ao receber a descrição de qualquer instância do problema, devolve uma solução da instância (ou informa que a instância não tem solução). Para cada instância do problema, o

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

algoritmo consome uma quantidade de tempo diferente. A relação entre esse consumo de tempo e o tamanho das instâncias dá uma medida da eficiência do algoritmo.

Em geral, um problema tem muitas instâncias diferentes de um mesmo tamanho. Esse fato exige a introdução dos conceitos de "pior caso" e "melhor caso". Para cada n, escolha uma instância I_n do problema que tenha tamanho n. Considere a família $I_0, I_1, I_2, \ldots, I_n, \ldots$ de instâncias. Dado um algoritmo $\mathscr{A}$ para o problema, seja $T(n)$ o tempo que $\mathscr{A}$ consome para resolver I_n. Dizemos que T mede o

consumo de tempo de $\mathscr{A}$ **no pior caso**

se o algoritmo consome no máximo $T(n)$ unidades de tempo para processar qualquer instância de tamanho n. Dizemos que T mede o consumo de $\mathscr{A}$ **no melhor caso** se o algoritmo consome pelo menos $T(n)$ unidades de tempo para processar qualquer instância de tamanho n.

Por exemplo, um algoritmo pode consumir $200n$ unidades de tempo no melhor caso e $10n^2 + 100n$ unidades no pior. (Estou ignorando os valores pequenos de n, para os quais $200n$ é maior que $10n^2 + 100n$.)

Nas nossas análises, suporemos quase sempre (a Seção 1.5 é uma exceção) que uma execução de cada linha de pseudocódigo consome uma quantidade de tempo que não depende do tamanho da instância submetida ao algoritmo. (Para isso, será necessário descrever os algoritmos de maneira suficientemente detalhada.) Em particular, suporemos quase sempre que o consumo de tempo das operações aritméticas (adição, multiplicação, comparação, atribuição, etc.) não depende do tamanho dos números envolvidos.

1.1.3. Recursão e algoritmos recursivos

Muitos problemas computacionais têm a seguinte "estrutura recursiva": cada solução de uma instância do problema contém soluções de instâncias menores do mesmo problema. 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. Considere o problema de encontrar a soma dos elementos positivos de um vetor. O seguinte algoritmo recebe um vetor $A[1..n]$ de números

inteiros, com $n \geq 0$, e devolve a soma dos elementos positivos do vetor:[3]

```
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 $n > 0$. Podemos supor, por hipótese de indução, que SOMAPOSITIVOS com argumentos A e $n-1$ produz o resultado prometido. Portanto, no início da linha 4, s é a soma dos elementos positivos de $A[1..n-1]$. A partir de s, as linhas 4 a 6 calculam corretamente a soma dos elementos positivos de $A[1..n]$.

Exemplo 2. O seguinte algoritmo recursivo recebe um número natural $n \geq 1$ e devolve $\lfloor \lg n \rfloor$, ou seja, o piso do logaritmo de n na base 2:

```
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 $n > 1$ e seja x o número PISODELOG$(\lfloor n/2 \rfloor)$. Por hipótese de indução, $x = \lfloor \lg \lfloor n/2 \rfloor \rfloor$. Resta mostrar que $x + 1 = \lfloor \lg n \rfloor$. Se n é par então $\lfloor n/2 \rfloor = n/2$ e $x = \lfloor \lg \lfloor n/2 \rfloor \rfloor = \lfloor \lg \frac{n}{2} \rfloor = \lfloor \lg n - 1 \rfloor = \lfloor \lg n \rfloor - 1$, donde $x + 1 = \lfloor \lg n \rfloor$. Agora suponha que n é ímpar. Nesse caso, $\lfloor n/2 \rfloor = (n-1)/2$ e portanto $x = \lfloor \lg \frac{n-1}{2} \rfloor = \lfloor \lg (n-1) - 1 \rfloor = \lfloor \lg (n-1) \rfloor - 1$, donde $x + 1 = \lfloor \lg (n-1) \rfloor$. Seja k o único número natural tal que $2^k \leq n < 2^{k+1}$. Como n é ímpar, temos também $2^k \leq n - 1 < 2^{k+1}$. Logo, $k \leq \lg n < k + 1$ e $k \leq \lg (n-1) < k+1$, donde $\lfloor \lg n \rfloor$ e $\lfloor \lg (n-1) \rfloor$ são ambos iguais a k. Segue daí que $x + 1 = \lfloor \lg (n-1) \rfloor = k = \lfloor \lg n \rfloor$, como queríamos mostrar.

Exemplo 3. Uma **subsequência** é o que sobra de uma sequência quando alguns de seus termos são apagados. Por exemplo, $(2,3,5,8)$ é uma subsequência de $(1,2,3,4,5,6,7,8)$.

Considere o problema de imprimir todas as $2^n - 1$ subsequências não-vazias da sequência $(1,2,\ldots,n)$. (Isto equivale ao problema de imprimir uma lista de todos os subconjuntos não-vazios de $\{1,2,\ldots,n\}$.) O seguinte algoritmo resolve o problema:

[3] Usaremos a excelente notação do livro de Cormen et al. para descrever algoritmos.

```
TODASSUBSEQUÊNCIAS (n)
1 aloca vetor S[1..n]
2 SUBSEQSCOMPREFIXO (0, 1)
```

Esta "casca" aloca um vetor auxiliar $S[1..n]$ e invoca a rotina recursiva SUBSEQSCOMPREFIXO, que faz o serviço todo. Do ponto de vista da rotina, o vetor S é uma variável global. A rotina recebe k e m e imprime cada subsequência não-vazia de $(m,\ldots,n)$ precedida do prefixo $S[1..k]$:

```
SUBSEQSCOMPREFIXO (k, m)
3 se m ≤ n
4     então S[k+1] ← m
5           IMPRIMA (S[1..k+1])
6           SUBSEQSCOMPREFIXO (k+1, m+1)
7           SUBSEQSCOMPREFIXO (k, m+1)
```

A prova da correção da rotina é uma indução na diferença $n-m$ que mede o tamanho da instância do problema. Se $m > n$, a rotina não imprime nada, pois a sequência $(m,\ldots,n)$ é vazia. Agora tome $m \leq n$. Podemos supor, por hipótese de indução, que SUBSEQSCOMPREFIXO produz o resultado prometido quando invocada com segundo argumento $m+1$. Portanto, a rotina imprime três grupos de sequências, cada uma precedida do prefixo $S[1..k]$: primeiro, a única subsequência de $(m,\ldots,n)$ que começa com m e tem comprimento 1; depois, todas as subsequências de $(m,\ldots,n)$ que começam com m e têm comprimento maior que 1; finalmente, todas as subsequências não-vazias de $(m+1,\ldots,n)$. O primeiro grupo é gerado pela linha 5, o segundo pela linha 6 e o terceiro pela linha 7. Assim, a rotina produz o resultado que prometeu.

Em particular, ao ser invocada com argumentos $(0,1)$, a rotina imprime todas as subsequências não-vazias de $(1,2,\ldots,n)$.

1.1.4. Convenções de notação

Para concluir esta introdução, estabelecemos algumas convenções de notação. Denotaremos por $\mathbb{N}$ o conjunto $\{0,1,2,3,\ldots\}$ dos números naturais (que coincide com o dos inteiros não-negativos). O conjunto $\mathbb{N} - \{0\}$ será denotado por $\mathbb{N}^>$. O conjunto dos números reais será denotado por $\mathbb{R}$. O conjunto dos reais não-negativos e o dos reais estritamente positivos serão denotados por $\mathbb{R}^\geq$ e por $\mathbb{R}^>$ respectivamente.

Denotaremos por $\lg n$ o logaritmo de um número n na base 2. Portanto, $\lg n = \log_2 n$.

O **piso** de um número x em $\mathbb{R}^\geq$ 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.

1.2. 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 computador empregado. Para tornar isso 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 "notações assintóticas": uma com sabor de "$\leq$", outra com sabor de "$\geq$", e uma terceira com sabor de "$=$".

1.2.1. Notação O

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

$$F(n) \leq c \cdot G(n)$$

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

Exemplo 1: $100n$ está em $O(n^2)$. De fato, para todo $n \geq 100$ tem-se $100n \leq n \cdot n = n^2$.

Exemplo 2: $2n^3 + 100n$ está em $O(n^3)$. 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 $O(n^3)$: para todo $n \geq 100$ tem-se $2n^3 + 100n \leq 2n^3 + n \cdot n \cdot n = 3n^3$.

Exemplo 3: $n^{1.5}$ está em $O(n^2)$. De fato, para todo $n \geq 1$ tem-se $n^{1.5} < n^{0.5} n^{1.5} = n^2$.

Exemplo 4: $\frac{1}{10} n^2$ não está em $O(n)$. Prova: Tome qualquer c em $\mathbb{N}^>$. Para todo $n > 10c$ temos $n^2/10 = n \cdot n/10 > 10c \cdot n/10 = cn$. Em suma, não é verdade que $\frac{1}{10} n^2 \leq cn$ para todo n suficientemente grande.

Exemplo 5: $2n$ está em $O(2^n)$. 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 $n > 1$ 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 $O(G)$ então $F + F'$ também está em $O(G)$. Eis uma prova da regra: Por hipótese, existem números c e n_0 tais que $F(n) \leq cG(n)$ 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 $O(n^3)$, a função $2n^3 + 100n$ também está em $O(n^3)$.

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 $O(n^2)$, 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 $O(n)$. 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 $O(n^2)$. Segue imediatamente do Exemplo 8.

Exercícios

1. Critique a afirmação "$n^2 - 100n$ está em $O(n^2)$ para todo $n \geq 100$".
2. Mostre que $\lg n$ está em $O(n^{0.5})$.

1.2.2. 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

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

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 $O(F)$.

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 $n > 2k$ 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

3. Mostre que 2^{n-1} está em $\Omega(2^n)$.
4. Mostre que $\lg n$ não é $\Omega(n)$.

1.2.3. Notação Teta

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

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

para todo n suficientemente grande. Se F está em $\Theta(G)$, diremos que F é **proporcional** a G, ainda que isto seja um tanto incorreto.

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

Exercício

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

1.2.4. Consumo de tempo de algoritmos

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

"$\mathscr{A}$ consome $O(n^2)$ unidades de tempo no pior caso"

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

(Se $\mathscr{A}$ consome tempo $O(n^2)$ 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 "$\mathscr{A}$ consome tempo $\Omega(n^2)$ no pior caso".)

Linear, quadrático, ene-log-ene. Um algoritmo é **linear** se consome tempo $\Theta(n)$ 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 **ene-log-ene** se consome tempo $\Theta(n \lg n)$ 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 tempo $\Theta(n^2)$ 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.

Exercício

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

1.3. 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.

1.3.1. Exemplo 2F(n–1)+1

Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^>$ e suponha que F satisfaz a seguinte recorrência: $F(1) = 1$ e

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

para $n \geq 2$. Eis alguns valores da função: $F(2) = 3$, $F(3) = 7$, $F(4) = 15$.

É fácil "desenrolar" a recorrência: $F(n) = 2F(n-1) + 1 = 2(2F(n-2) + 1) + 1 = 4F(n-2) + 3 = 2^j F(n-j) + 2^j - 1 = 2^{n-1}F(1) + 2^{n-1} - 1$. Concluímos assim que

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

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

1.3.2. Exemplo F(n–1)+n

Seja F uma função que leva $\mathbb{N}$ em $\mathbb{R}^>$ e suponha que F satisfaz a seguinte recorrência: $F(1) = 1$ e

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

para $n > 1$. Em particular, $F(2) = 2F(1) + 2 = 3$ e $F(3) = 2F(2) + 3 = 6$.

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

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

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

Condições iniciais. O valor de $F(1)$ tem pouca influência sobre o resultado. Se tivéssemos $F(1) = 10$, por exemplo, teríamos $F(n) = \frac{1}{2}(n^2+n)+9$ no lugar de (4), e esta função também está em $\Theta(n^2)$.

O ponto a partir do qual (3) começa a valer também tem pouca influência sobre o resultado final. Se (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 (4), e esta função também está em $\Theta(n^2)$.

1.3.3. Exemplo 2F(n/2)+n

Seja F uma função que leva $\mathbb{N}$ em $\mathbb{R}^>$. Suponha que F satisfaz as seguintes condições: $F(1) = 1$ e

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

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

É mais fácil de entender (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

$$\begin{aligned}
F(2^j) &= 2F(2^{j-1})+2^j \\
&= 2^2F(2^{j-2})+2^12^{j-1}+2^j \\
&= 2^3F(2^{j-3})+2^22^{j-2}+2^12^{j-1}+2^j \\
&= 2^jF(2^0)+2^{j-1}2^1+\cdots+2^22^{j-2}+2^12^{j-1}+2^02^j \\
&= 2^j2^0+2^{j-1}2^1+\cdots+2^22^{j-2}+2^12^{j-1}+2^02^j \\
&= (j+1)2^j. \qquad (6)
\end{aligned}$$

Isto pode ser reescrito assim:

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

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 crescente, ou seja, que

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

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 $n > 1$. 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 (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 (5), $F(n) = 2F(\frac{n-1}{2})+n \leq 2F(\frac{n+1}{2})+n < 2F(\frac{n+1}{2})+n+1 = F(n+1)$. Mostramos assim que F é crescente.

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

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

para todo $n \geq 2$. Tome o único j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Em virtude de (8), $F(n) < F(2^{j+1})$. Em virtude de (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 (7): mostraremos que

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

para todo $n \geq 2$. Tome o único j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Em virtude de (8) e (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 (9) e (10), concluímos que F está em $\Theta(n\lg n)$.

Condições iniciais. O ponto n_0 a partir do qual (5) começa a valer, bem como os valores de $F(n)$ quando $n < n_0$, têm pouca influência sobre o resultado. Quaisquer que sejam esses valores iniciais, F estará em $\Theta(n\lg n)$. (Por exemplo, se (5) vale apenas para $n > 2$, teremos $n\lg n + \frac{F(2)-2}{2}n$ no lugar de (7).)

Exercícios

1. Seja F um função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que $F(n) = F(\lfloor n/2 \rfloor)+1$ para todo $n > 1$. Mostre que F está em $\Theta(\lg n)$.
2. Seja F um função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que $F(n) = 2F(\lfloor n/2 \rfloor)+1$ para todo $n > 1$. Mostre que F está em $\Theta(n)$.
3. Seja F um função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que $F(n) = 4F(\lfloor n/2 \rfloor)+n$ quando $n > 1$. 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.

1.3.4. Exemplo 3F(n/2)+n

Seja F um função de $\mathbb{N}$ em $\mathbb{R}^>$. Suponha que F satisfaz as seguintes condições: $F(1) = 1$ e

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

se $n > 1$. Em particular, $F(2) = 5$ e $F(3) = 6$. Quando $n = 2^j$, temos

$$\begin{aligned}
F(2^j) &= 3F(2^{j-1})+2^j \\
&= 3^2F(2^{j-2})+3^12^{j-1}+2^j \\
&= 3^3F(2^{j-3})+3^22^{j-2}+3^12^{j-1}+2^j \\
&= 3^jF(2^0)+3^{j-1}2^1+\cdots+3^22^{j-2}+3^12^{j-1}+3^02^j \\
&= 3^j2^0+3^{j-1}2^1+\cdots+3^22^{j-2}+3^12^{j-1}+3^02^j
\end{aligned}$$

$$
\begin{aligned}
&= 2^j\left((3/2)^j + (3/2)^{j-1} + \cdots + (3/2)^2 + (3/2)^1 + (3/2)^0\right) \\
&= 2^j \frac{(3/2)^{j+1} - 1}{3/2 - 1} \\
&= 2^{j+1}\left((3/2)^{j+1} - 1\right) \\
&= 3^{j+1} - 2^{j+1}. \qquad (12)
\end{aligned}
$$

Para ter certeza, podemos conferir esta fórmula por indução em j. Se $j = 0$ temos $F(2^j) = 1 = 3^{j+1} - 2^{j+1}$. Agora tome $j > 0$. De acordo com a recorrência (11), $F(2^j) = 3F(2^{j-1}) + 2^j$. Por hipótese de indução, (12) vale com "j" no papel de "$j+1$". Logo, $F(2^j) = 3F(2^{j-1}) + 2^j = 3\left(3^j - 2^j\right) + 2^j = 3^{j+1} - 2^{j+1}$, confirmando (12).

Como $3^j = (2^{\lg 3})^j = (2^j)^{\lg 3} = n^{\lg 3}$, a fórmula (12) pode ser reescrita assim:

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

para toda potência n de 2. (A título de curiosidade, $\lg 3$ fica entre 1.584 e 1.585.) Embora esta fórmula só valha quando n é potência 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 crescente:

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

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

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

para todo $n \geq 1$. Tome j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Como F é crescente, (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:

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

para todo $n \geq 1$. Tome j em $\mathbb{N}$ tal que $2^j \leq n < 2^{j+1}$. Como F é crescente, (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 (13) e (14) mostram que

$$F \text{ está em } \Theta(n^{\lg 3}).$$

Se alterarmos o problema de modo que (11) valha apenas para $n \geq n_0$ e adotarmos valores arbitrários para $F(1), F(2), \ldots, F(n_0 - 1)$, a função F continuará em $\Theta(n^{\lg 3})$.

1.3.5. Teorema mestre

O seguinte teorema dá a solução de muitas recorrências semelhantes às das Subseções 1.3.3 e 1.3.4. Sejam a, k e c números em $\mathbb{N}^>$, $\mathbb{N}$ e $\mathbb{R}^>$ respectivamente. Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que

$$F(n) = aF\left(\frac{n}{2}\right) + cn^k \tag{15}$$

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

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

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

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

(Recorrências como (15) aparecem na análise de algoritmos baseados na estratégia 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.)

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

Eis outra aplicação do teorema. Seja c um número em $\mathbb{R}^>$ e seja F uma função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que $F(n) = F(\lfloor n/2 \rfloor) + 2F(\lceil n/2 \rceil) + cn$ para todo $n > 1$. Nessas condições, F é não-decrescente e portanto (16) garante que F está em $\Theta(n^{\lg 3})$. (Compare com o resultado da Subseção 1.3.4.)

Generalização. O resultado (16-18) 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 $\mathbb{R}^>$. Seja F é uma função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que

$$F(n) = aF\left(\frac{n}{b}\right) + cn^k \tag{19}$$

para $n = n_0 b^1, n_0 b^2, n_0 b^3, \ldots$ Suponha ainda que F é assintoticamente não-decrescente. Nessas condições,

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

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

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

Exercício

4. Seja F um função de $\mathbb{N}$ em $\mathbb{R}^>$ tal que $F(n) = F(\lfloor n/2 \rfloor) + F(\lceil n/2 \rceil) + 1$ para todo n suficientemente grande. Mostre que F está em $\Theta(n)$.

1.3.6. Recorrências assintóticas

Seja F uma função de $\mathbb{N}$ em $\mathbb{R}^>$ e a um número em $\mathbb{N}^>$. Suponha que existe uma função f tal que $F(n) = aF(\lfloor n/2 \rfloor) + f(n)$ para todo n suficientemente grande. Se f está em $O(n^k)$, escrevemos simplesmente

$$F(n) = aF(\lfloor n/2 \rfloor) + O(n^k). \tag{23}$$

Nessas condições, se $\lg a > k$ e F é assintoticamente não-decrescente, podemos concluir que F está em $\Theta(n^{\lg a})$.

Algo análogo vale com "$\Theta(n^k)$" e "$\lg a = k$". Dizemos que $F(n) = aF(\lfloor n/2 \rfloor) + \Theta(n^k)$ se existe uma função f em $\Theta(n^k)$ tal que $F(n) = aF(\lfloor n/2 \rfloor) + f(n)$ para todo n suficientemente grande. Nesse caso, se $\lg a = k$ e F é assintoticamente não-decrescente, então F está em $\Theta(n^k \lg n)$.

Finalmente, dizemos que $F(n) = aF(\lfloor n/2 \rfloor) + \Omega(n^k)$ se existe uma função f em $\Omega(n^k)$ tal que $F(n) = aF(\lfloor n/2 \rfloor) + f(n)$ para todo n suficientemente grande. Se $\lg a < k$ e F é assintoticamente não-decrescente, então F está em $\Theta(n^k)$.

1.4. Ordenação de vetor

Um vetor $A[p..r-1]$ é **crescente** se $A[p] \leq A[p+1] \leq \cdots \leq A[r-1]$. A tarefa de colocar um vetor em ordem crescente é um dos problemas computacionais mais conhecidos e bem-estudados:

> Problema da Ordenação: Rearranjar um vetor $A[p..r-1]$ em ordem crescente.

1.4.1. Mergesort

O seguinte algoritmo é uma maneira eficiente de resolver o Problema da Ordenação. Ele supõe que $p \leq r$ e que o vetor está vazio se $p = r$:

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

A rotina INTERCALA rearranja o vetor $A[p..r-1]$ em ordem crescente supondo que os subvetores $A[p..q-1]$ e $A[q..r-1]$ são ambos crescentes.

O algoritmo está correto. Adote o número $n := r - p$ 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 $n > 1$. Nas linhas 3 e 4, o algoritmo é aplicado a instâncias do problema que têm tamanho estritamente menor que n (tamanho $\lfloor n/2 \rfloor$ na linha 3 e tamanho $\lceil n/2 \rceil$ na linha 4). Assim, podemos supor, por hipótese de indução, que no início da linha 5 os vetores $A[p..q-1]$ e $A[q..r-1]$ estão em ordem crescente. Portanto,

no fim da linha 5, $A[p..r-1]$ estará em ordem crescente graças à ação de INTERCALA.

Consumo de tempo. Seja $T(n)$ o tempo que o algoritmo consome, no pior caso, para processar uma instância de tamanho n. Então

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

As parcelas $T(\lfloor \frac{n}{2} \rfloor)$ e $T(\lceil \frac{n}{2} \rceil)$ 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. (Veja o Exercício 1 abaixo.) Como vimos nas Subseções 1.3.3 e 1.3.5, a função T está em

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

O consumo de tempo do algoritmo no melhor caso também está em $\Theta(n \lg n)$.

Exercício

1. Descreva em pseudocódigo o algoritmo INTERCALA mencionado acima. Mostre que o consumo de tempo do algoritmo é $\Theta(n)$.

1.4.2. 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.

A estratégia da divisão e conquista rende algoritmos eficientes para muitos problemas. Veja, por exemplo, as Seções 1.5 e 1.7, bem como o capítulo de Geometria Computacional deste volume.

1.5. Multiplicação de números naturais

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. No caso $m = n$, o consumo de tempo é proporcional a n^2.

É difícil fazer algo mais rápido se m e n forem pequenos (menores que 500 talvez). Mas existe um algoritmo que é bem mais rápido quando m e n são grandes (como em aplicações criptográficas, por exemplo).

1.5.1. O problema

Usaremos notação decimal para representar os números. (Poderíamos igualmente bem usar notação binária, ou notação base 2^{32}, por exemplo.) Diremos que um número natural u **tem** n **dígitos** 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 $u \cdot v$.

Convém lembrar que o produto de dois números com n dígitos cada tem $2n$ dígitos.

Você deve imaginar que cada número natural é representado por um vetor de dígitos (veja a Subseção 1.5.5). Mas nossa discussão será conduzida num nível de abstração alto, sem manipulação explícita desse vetor.

1.5.2. *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 passo 1 calcula todos os n produtos de u por um dígito de v (cada um desses produtos tem $n+1$ dígitos). O passo 2 do algoritmo desloca para a esquerda, de um número apropriado de casas, cada um dos n produtos obtidos no passo 1 e soma os n números resultantes.

O passo 1 do algoritmo consome tempo proporcional a n^2. O passo 2 também consome tempo proporcional a n^2. Assim, o consumo de tempo do algoritmo usual de multiplicação está em $\Theta(n^2)$.

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

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.

1.5.3. *Rascunho de um algoritmo mais eficiente*

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

$$u = a \cdot 10^{n/2} + b.$$

Defina c e d analogamente para v, de modo que $v = c \cdot 10^{n/2} + d$. Temos então

$$u \cdot v = a \cdot c \cdot 10^n + (a \cdot d + b \cdot c) \cdot 10^{n/2} + b \cdot d. \quad (24)$$

Esta expressão reduz a multiplicação de dois números com n dígitos cada a quatro multiplicações (a saber, a por c, a por d, b por c e b por d) de números com $n/2$ dígitos cada.

Se n for uma potência de 2, a redução pode ser aplicada 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, sendo n uma potência de 2, e devolve o produto $u \cdot v$:

```
RASCUNHO (u, v, n)
1  se n = 1
2      então devolva u · v
3      senão k ← n/2
4            a ← ⌊u/10^k⌋
5            b ← u mod 10^k
6            c ← ⌊v/10^k⌋
7            d ← v mod 10^k
8            ac ← RASCUNHO (a, c, k)
9            bd ← RASCUNHO (b, d, k)
10           ad ← RASCUNHO (a, d, k)
11           bc ← RASCUNHO (b, c, k)
12           x ← ac · 10^{2k} + (ad + bc) · 10^k + bd
13           devolva x
```

O algoritmo está correto. É claro que o algoritmo produz o resultado correto se $n = 1$. Suponha agora que $n > 1$. Como $k < n$, podemos supor, por hipótese de indução, que a linha 8 produz o resultado correto, ou seja, que $ac = a \cdot c$. Analogamente, podemos supor que $bd = b \cdot d$, $ad = a \cdot d$ e $bc = b \cdot c$ no início da linha 12. A linha 12 não faz mais que implementar (24), donde $x = u \cdot v$.

Consumo de tempo. 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^{2k} e 10^k na linha 12 também consomem no máximo n unidades de tempo, pois podem ser implementadas por deslocamento de casas decimais. As adições na linha 12 consomem tempo proporcional ao número de dígitos de ac, ad, bc e bd, que têm $2k = n$ dígitos cada.

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

$$T(n) = 4T(n/2) + n, \quad (25)$$

onde as quatro parcelas "$T(n/2)$" se referem às linhas 8 a 11 e a parcela "n" se refere ao consumo das demais linhas. Segue daí que T está em $\Theta(n^2)$, como

já vimos no Exercício 3 da Seção 1.3. Assim, o algoritmo RASCUNHO não é mais eficiente que o algoritmo usual de multiplicação.

```
   99998888    u
   77776666    v

    9999       a
       8888    b
    7777       c
       6666    d

77762223       ac
    59247408   bd
  135775310    ad+bc

7777580112347408   x
```

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

1.5.4. Algoritmo de Karatsuba e Ofman

Para tornar o algoritmo da seção anterior muito mais rápido, basta observar que os três números de que precisamos do lado direito de (24) — a saber $a \cdot c$, $(a \cdot d + b \cdot c)$ e $b \cdot d$ — podem ser obtidos com apenas três multiplicações. De fato,

$$a \cdot d + b \cdot c \;=\; (a+b) \cdot (c+d) - a \cdot c - b \cdot d$$

e portanto, se denotarmos $(a+b) \cdot (c+d)$ por y, a equação (24) será substituída por

$$u \cdot v \;=\; a \cdot c \cdot 10^n + (y - a \cdot c - b \cdot d) \cdot 10^{n/2} + b \cdot d\,. \qquad (26)$$

Para o caso em que n não é potência de 2, basta usar $\lceil n/2 \rceil$ no lugar de $n/2$. Temos então o seguinte algoritmo de Karatsuba e Ofman, que recebe números naturais u e v com n dígitos cada e devolve o produto $u \cdot v$:

```
KARATSUBA (u, v, n)
1  se n ≤ 3
2      então devolva u · v
3      senão k' ← ⌊n/2⌋
4            k ← ⌈n/2⌉
5            a ← ⌊u/10^k⌋
6            b ← u mod 10^k
7            c ← ⌊v/10^k⌋
8            d ← v mod 10^k
```

```
9     ac ← KARATSUBA (a, c, k')
10    bd ← KARATSUBA (b, d, k)
11    y ← KARATSUBA (a+b, c+d, k+1)
12    x ← ac · 10^{2k} + (y − ac − bd) · 10^k + bd
13    devolva x
```

O algoritmo está correto. A prova da correção do algoritmo é análoga à prova da correção de 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 11, a uma instância de tamanho $\lceil n/2 \rceil + 1$, e este número só é menor que n quando $n > 3$.

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

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + n. \tag{27}$$

As parcelas $T(\lfloor n/2 \rfloor)$, $T(\lceil n/2 \rceil)$ e $T(\lceil n/2 \rceil + 1)$ se referem às linhas 9, 10 e 11 respectivamente. A parcela n representa o consumo de tempo das demais linhas.

Como sugerimos na Subseção 1.3.5, a função T está na mesma ordem assintótica que a função T' que satisfaz a recorrência $T'(n) = 3T'(\lfloor n/2 \rfloor) + n$. De acordo com a Subseção 1.3.4, T' está em $\Theta(n^{\lg 3})$ e portanto

$$T \text{ está em } \Theta(n^{\lg 3}). \tag{28}$$

Como $\lg 3 < 1.6 < 2$, o algoritmo KARATSUBA é bem mais rápido que o algoritmo usual. (Observe que $n^{\lg 3}$ é apenas um pouco maior que $n\sqrt{n}$.) Mas esta superioridade só se manifesta quando n é maior que algumas centenas, em virtude da constante multiplicativa escondida sob a notação Θ.

1.5.5. Detalhes de implementação

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

O algoritmo KARATSUBA recebe os vetores $U[1..n]$ e $V[1..n]$ que representam u e v respectivamente e devolve o vetor $X[1..2n]$ que representa x. Nas linhas 5 e 6, a é representado por $U[k+1..n]$ e b é representado por $U[1..k]$. A implementação da linha 7 e 8 é análoga. Nas linhas 9 e 10, ac é representado por um vetor $AC[1..2k']$ e bd é representado por um vetor $BD[1..2k]$.

Na linha 11, para calcular $a+b$ basta submeter $U[k+1..n]$ e $U[1..k]$ ao algoritmo usual de adição, que produz um vetor $AB[1..k+1]$. Algo análogo vale para a subexpressão $c+d$. O número y é representado por um vetor $Y[1..2k+2]$.

Na linha 12, o valor de $y - ac - bd$ é calculado pelo algoritmo usual de subtração (não é preciso cuidar de números negativos pois $y - ac - bd \geq 0$). Para

999988888	u
077766666	v
07769223	ac
5925807408	bd
98887	$a+b$
67443	$c+d$
6669235941	y
735659310	$y-ac+bd$
077765801856807408	x

Multiplicação de u por v calculada pelo algoritmo KARATSUBA. Aqui, $n = 9$ e portanto $k' = 4$ e $k = 5$. O resultado da operação é x. Se u e v tivessem n dígitos cada, ac e bd teriam $2n$ dígitos, y teria $2n+2$ dígitos e x teria $2n$ dígitos.

calcular o valor da expressão $ac \cdot 10^{2k} + (y - ac - bd) \cdot 10^k + bd$, basta concatenar os vetores $AC[1..2k']$ e $BD[1..2k]$ que representam ac e bd respectivamente e somar o resultado com $y - ac - bd$ usando o algoritmo usual de adição.

1.5.6. Divisão e conquista

O algoritmo KARATSUBA é um bom exemplo de aplicação da estratégia de divisão e conquista, que já encontramos na Seção 1.4. O segredo do sucesso da estratégia está no fato de que o processo de divisão da instância original (linhas 5-8 do algoritmo) e o processo de combinação das soluções das instâncias menores (linha 12) consomem relativamente pouco tempo.

1.6. O problema da maioria

Imagine uma eleição com muitos candidatos e muitos eleitores. Como é possível determinar, em tempo proporcional ao número de eleitores, se algum dos candidatos obteve a maioria[4] dos votos?

1.6.1. O problema

Seja $A[1..n]$ um vetor de números naturais. Para qualquer número x, a cardinalidade do conjunto $\{i : 1 \leq i \leq n,\ A[i] = x\}$ é o **número de ocorrências** de x em $A[1..n]$. Diremos que x é um **valor majoritário** de $A[1..n]$ se o número de ocorrências de x em $A[1..n]$ é maior que $n/2$. Assim, um valor majoritário ocorre pelo menos $(n+2)/2$ vezes se n é par e pelo menos $(n+1)/2$ vezes se n é ímpar.

[4] A popular expressão "metade dos votos mais um" é incorreta pois o número de votos pode ser ímpar.

Problema da Maioria: Encontrar um valor majoritário em um dado vetor $A[1..n]$.

Nem todo vetor tem um valor majoritário. Mas se um valor majoritário existe, ele é único.

Poderíamos ordenar o vetor $A[1..n]$, contar o número de ocorrências de cada elemento, e verificar se alguma dessas contagens supera $n/2$. A ordenação do vetor consome $\Omega(n \lg n)$ unidades de tempo se usarmos o algoritmo MERGESORT (veja Seção 1.4).[5] As contagens no vetor ordenado consomem $\Omega(n)$. Portanto, o algoritmo todo consumirá $\Omega(n \lg n)$ unidades de tempo, À primeira vista, parece impossível resolver o problema em menos tempo.[6] Mas existe um algoritmo que consome apenas $O(n)$ unidades de tempo, como mostraremos a seguir.

1.6.2. Um algoritmo linear

Um algoritmo eficiente para o problema começa com a seguinte observação: se $A[1..n]$ tem um valor majoritário e se $A[n-1] \neq A[n]$ então $A[1..n-2]$ também tem um valor majoritário. (A observação se aplica igualmente a quaisquer dois elementos: se $A[i] \neq A[j]$ e A tem um valor majoritário então o vetor que se obtém pela eliminação das posições i e j também tem um valor majoritário.) Embora a intuição aceite esta observação como verdadeira, sua demonstração exige algum cuidado (veja a prova da correção do algoritmo).

A recíproca da observação não é verdadeira. Por exemplo, $(2,5,5,1,3)$ não tem valor majoritário, mas se retirarmos os dois últimos elementos então 5 passará a ser um valor majoritário.

Outra observação simples mas importante: para qualquer k no intervalo $1..n$, um número x é valor majoritário de $A[1..n]$ se e somente se x é majoritário em $A[1..k-1]$ ou em $A[k..n]$ ou em ambos. (É possível, entretanto, que $A[1..k-1]$ e $A[k..n]$ tenham valores majoritários sem que $A[1..n]$ tenha um valor majoritário.)

Usaremos as duas observações para obter um bom candidato a valor majoritário. Um **bom candidato** é um número x com a seguinte propriedade: se o vetor tem um valor majoritário então x é esse valor. Se tivermos um bom cadidato, será fácil verificar se ele é de fato majoritário. (Basta percorrer o vetor e contar o número de ocorrências do candidato.) A dificuldade está em obter um bom candidato em tempo $O(n)$.

O seguinte algoritmo recebe um vetor $A[1..n]$ de números naturais, com $n > 0$, e devolve um valor majoritário se tal existir. Se o vetor não tem valor majoritário, devolve -1 (note que o vetor não tem elementos negativos):

---

[5] Demonstra-se que todo algoritmo baseado em comparações consome tempo $\Omega(n \lg n)$.

[6] Se o número de possíveis valores de $A[1..n]$ fosse pequeno, conhecido a priori e independente de n, poderíamos usar um algoritmo do tipo bucket sort, que consome $O(n)$ unidades de tempo.

```
MAJORITÁRIO (A, n)
1  x ← A[1]
2  y ← 1

3  para m ← 2 até n faça
4     se y > 0
5        então se A[m] = x
6                 então y ← y + 1
7                 senão y ← y − 1
8        senão x ← A[m]
9              y ← 1
10 c ← 0
11 para m ← 1 até n faça
12    se A[m] = x
13       então c ← c + 1
14 se c > n/2
15    então devolva x
16    senão devolva −1
```

O algoritmo está correto. A cada passagem pela linha 3, imediatamente antes da comparação de m com n, temos os seguintes invariantes:

i. $y \geq 0$,

ii. existe k no intervalo $1..m-1$ tal que
$A[1..k-1]$ não tem valor majoritário e
x ocorre exatamente $\frac{1}{2}(m-k+y)$ vezes em $A[k..m-1]$.

(Note que $m-k$ é o número de elementos de $A[k..m-1]$.) A validade de i é óbvia. A validade de ii pode ser verificada como segue. No início da primeira iteração, o invariante ii vale com $k = 1$. Suponha agora que ii vale no início de uma iteração qualquer. Seja k um índice com as propriedades asseguradas por ii. Se $y > 0$ então o invariante vale no início da próxima iteração com o mesmo k da iteração corrente, quer $A[m]$ seja igual a x, quer seja diferente. Suponha agora que $y = 0$. Como o número de ocorrências de x em $A[k..m]$ é exatamente $\frac{1}{2}(m-k)$, o vetor $A[k..m]$ não tem valor majoritário. Como $A[1..k-1]$ também não tem valor majoritário, $A[1..m]$ não tem valor majoritário. Assim, no início da próxima iteração, o invariante ii valerá com $k = m-1$.

Na última passagem pela linha 3 temos $m = n+1$ e portanto existe k no intervalo $1..n$ tal que $A[1..k-1]$ não tem valor majoritário e x ocorre exatamente $\frac{1}{2}(n-k+1+y)$ vezes em $A[k..n]$. Suponha agora que $A[1..n]$ tem um valor majoritário a. Como $A[1..k-1]$ não tem valor majoritário, a é majoritário em $A[k..n]$. Portanto, $a = x$ e $y > 0$. Concluímos assim que, no começo da linha 10,

x é um bom candidato.

As linhas 10 a 13 verificam se x é de fato majoritário.

Consumo de tempo. É razoável supor que o consumo de tempo de uma execução de qualquer das linhas do algoritmo é constante, ou seja, não depende de n. Assim, podemos estimar o consumo de tempo total contando o número de execuções das diversas linhas.

O bloco de linhas 4-9 é executado $n-1$ vezes. O bloco de linhas 12-13 é executado n vezes. As linhas 1 e 2 e o bloco de linhas 14-16 é executado uma só vez. Portanto, o consumo de tempo total do algoritmo está em

$$\Theta(n)$$

(tanto no melhor quanto no pior caso). O algoritmo é linear.

A análise do consumo de tempo do algoritmo é muito simples neste caso. A dificuldade do problema está em inventar um algoritmo que seja mais rápido que o óbvio.

1.7. Segmento de soma máxima

Imagine uma grandeza que evolui com o tempo, aumentando ou diminuindo uma vez por dia, de maneira irregular. Dado o registro de valores desta grandeza ao longo de um ano, queremos encontrar um intervalo de tempo em que a variação acumulada tenha sido máxima. Esta seção (inspirada no livro de Bentley) estuda três algoritmos para o problema. Cada algoritmo é mais eficiente que o anterior. Cada algoritmo usa uma estratégia diferente.

1.7.1. O problema

Um **segmento** de um vetor $A[p..r]$ é qualquer subvetor da forma $A[i..k]$, com $p \le i \le k \le r$. A condição "$i \le k$" garante que o segmento não é vazio.[7] A **soma** de um segmento $A[i..k]$ é o número $A[i]+A[i+1]+\cdots+A[k]$.

A **solidez** de um vetor $A[p..r]$ é a soma de um segmento de soma máxima. (O termo "solidez" é apenas uma conveniência local e não será usado fora desta seção.)

> Problema do Segmento de Soma Máxima: Calcular a solidez de um vetor $A[p..r]$ de números inteiros.

Se o vetor não tem elementos negativos, sua solidez é a soma de todos os elementos. Por outro lado, se todos os elementos são negativos então a solidez do vetor é o valor de seu elemento menos negativo.

1.7.2. Primeiro algoritmo

Se aplicarmos cegamente a definição de solidez, teremos que examinar todos os pares ordenados (i, j) tais que $p \le i \le j \le r$. Esse algoritmo consumiria $\Theta(n^3)$ unidades de tempo. Mas é possivel fazer algo mais eficiente:

[7] Teria sido mais "limpo" e natural aceitar segmentos vazios. Mas a discussão fica ligeiramente mais simples se nos limitarmos aos segmentos não-vazios.

20 −30 15 −10 30 −20 −30 30

A cor mais clara destaca um segmento de soma máxima.
A solidez do vetor é 35.

```
SOLIDEZ1 (A, p, r)
1  x ← A[r]
2  para q ← r − 1 decrescendo até p faça
3      s ← 0
4      para j ← q até r 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 p,

$$x \text{ é a solidez do vetor } A[q+1\,..\,r].$$

Este invariante mostra que o algoritmo está correto, pois na última passagem pela linha 2 teremos $q = p-1$ e então x será a solidez do $A[p\,..\,r]$.

Consumo de tempo. O consumo de tempo do algoritmo será medido em relação ao tamanho $n := r-p+1$ do vetor $A[p\,..\,r]$. Podemos supor que uma execução de qualquer das linhas do algoritmo consome uma quantidade de tempo que não depende de n. Para cada valor fixo de q, o bloco de linhas 5-6 é repetido $r-q+1$ vezes. Como q decresce de $r-1$ até p, o número total de repetições do bloco de linhas 5-6 é

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

Portanto, o consumo de tempo está em $\Theta(n^2)$.

O algoritmo é ineficiente porque a soma de cada segmento é recalculada muitas vezes. Por exemplo, a soma de $A[10\,..\,r]$ é refeita durante o cálculo das somas de $A[9\,..\,r]$, $A[8\,..\,r]$, etc. O algoritmo seguinte procura remediar esta ineficiência.

Exercícios

1. Modifique o algoritmo SOLIDEZ1 de modo que ele devolva um par (i,k) de índices tal que a soma $A[i]+\cdots+A[k]$ é a solidez de $A[p\,..\,r]$.
2. Escreva um algoritmo análogo a SOLIDEZ1 para tratar da variante do problema que admite segmentos vazios. Nessa variante, um segmento $A[i\,..\,k]$ é **vazio** se $i = k+1$. A **soma** de um segmento vazio é 0.

1.7.3. Segundo algoritmo: divisão e conquista

Nosso segundo algoritmo usa a estratégia da divisão e conquista, que consiste em dividir a instância dada ao meio, resolver cada uma das metades e finalmente juntar as duas soluções. (Veja a Subseção 1.4.2.) O algoritmo devolve a solidez do vetor $A[p..r]$, supondo $p \leq r$:

```
SOLIDEZII (A, p, r)
 1  se p = r
 2     então devolva A[p]
 3     senão q ← ⌊(p+r)/2⌋
 4           x' ← SOLIDEZII (A, p, q)
 5           x'' ← SOLIDEZII (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. Seja $n := r - p + 1$ o tamanho do vetor $A[p..r]$. O algoritmo dá a resposta correta se $n = 1$. Suponha agora que $n > 1$. Depois da linha 4, por hipótese de indução, x' é a solidez de $A[p..q]$. Depois da linha 5, x'' é a solidez de $A[q+1..r]$. Depois do bloco de linhas 6-13, $y' + y''$ é a maior soma da forma $A[i] + \cdots + A[j]$ com $i \leq q < j$. (Veja o Exercício 3.)

Em suma, $y' + y''$ 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 solidez de $A[p..r]$.

Consumo de tempo. Seja $T(n)$ o consumo de tempo do algoritmo no pior caso quando aplicado a um vetor de tamanho n. O bloco de linhas 6 a 13 examina todos os elementos de $A[p..r]$ 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 e a parcela $T(\lfloor n/2 \rfloor)$ descreve o consumo da linha 5. Esta recorrência já foi discutida nas Subseções 1.3.3 e 1.3.5, onde mostramos que T está em

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

O algoritmo SOLIDEZII é mais eficiente que SOLIDEZI pois $n \lg n$ é $O(n^2)$ mas n^2 não é $O(n \lg n)$. Ainda assim, SOLIDEZII tem ineficiências, pois refaz

alguns cálculos diversas vezes (por exemplo, a soma de $A[i..q]$ nas linhas 6-9 já foi calculada durante a execução da linha 4). A seguir, procuraremos eliminar esta ineficiência.

Exercício

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

1.7.4. Terceiro algoritmo: programação dinâmica

Nosso terceiro algoritmo usa a técnica da programação dinâmica, que consiste em armazenar os resultados de cálculos intermediários numa tabela para evitar que eles sejam repetidos.

Mas a técnica não pode ser aplicada diretamente ao nosso problema. Será preciso tratar do problema auxiliar que restringe a atenção aos segmentos terminais do vetor: dado um vetor $A[p..r]$, encontrar a maior soma da forma

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

com $p \le i \le r$. Para facilitar a discussão, diremos que a maior soma desta forma é a **firmeza** do vetor $A[p..r]$. A solidez do vetor é o máximo das firmezas de $A[p..r]$, $A[p..r-1]$, $A[p..r-2]$, etc.

A firmeza do vetor tem uma propriedade recursiva que a solidez não tem. Suponha que $A[i]+\cdots+A[r]$ é a firmeza de $A[p..r]$. Se $i < r$ então $A[i]+\cdots+A[r-1]$ é a firmeza de $A[p..r-1]$. (De fato, se $A[p..r-1]$ tivesse firmeza maior então existiria $h \le r-1$ tal que $A[h]+\cdots+A[r-1] > A[i]+\cdots+A[r-1]$ e portanto teríamos $A[h]+\cdots+A[r] > A[i]+\cdots+A[r]$, o que é impossível.)

Se denotarmos a firmeza de $A[p..q]$ por $F[q]$, podemos resumir a propriedade recursiva por meio de uma recorrência: para qualquer vetor $A[p..r]$,

$$F[r] = \max\left(F[r-1]+A[r]\,,\,A[r]\right). \qquad (29)$$

Em outras palavras, se $F[r-1] > 0$ então $F[r] = F[r-1]+A[r]$, senão $F[r] = A[r]$.

A recorrência (29) serve de base para o seguinte algoritmo, que calcula a solidez de $A[p..r]$ supondo $p \le r$:

```
SOLIDEZIII (A, p, r)
1  F[p] ← A[p]
2  para q ← p+1 até r faça
3      se F[q−1] > 0
4          então F[q] ← F[q−1]+A[q]
5          senão F[q] ← A[q]
6  x ← F[p]
7  para q ← p+1 até r faça
8      se F[q] > x então x ← F[q]
9  devolva x
```

O algoritmo está correto. A cada passagem pela linha 2, imediatamente antes que q seja comparado com r, $F[q-1]$ é a firmeza de $A[p..q-1]$. Mais que isso,

$$F[j] \text{ é a firmeza de } A[p..j] \tag{30}$$

para todo j no intervalo $p..q-1$. Logo, no início da linha 6, (30) vale para todo j no intervalo $p..r$.

O bloco de linhas 6-8 escolhe a maior das firmezas. Portanto, no fim do algoritmo, x é a solidez de $A[p..r]$.

Consumo de tempo. O algoritmo consome tempo proporcional ao tamanho $n := r-p+1$ do vetor. O consumo de tempo do algoritmo está em

$$\Theta(n)$$

e o algoritmo é, portanto, linear.

Exercícios

4. Os dois processos iterativos de SOLIDEZIII podem ser fundidos num só, evitando-se assim o uso do vetor $F[p..r]$. Uma variável f pode fazer o papel de um elemento genérico do vetor F. Implemente essa ideia.
5. Suponha dada uma matriz $A[1..n, 1..n]$ de números inteiros (nem todos não-negativos). Encontrar uma submatriz quadrada de A que tenha soma máxima. (Veja o livro de Bentley.)

1.7.5. Observações sobre programação dinâmica

O algoritmo SOLIDEZIII é um exemplo de **programação dinâmica**.[8] A técnica 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 $F[p..r]$.) O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.

As Seções 1.8, 1.9 e 1.10 exibem outros exemplos de programação dinâmica.

1.8. As linhas de um texto

Queremos imprimir uma sequência de palavras numa folha de papel, ocupando uma ou mais linhas consecutivas, de modo que cada linha tenha no máximo M caracteres. Para que a margem direita fique razoavelmente uniforme, queremos distribuir as palavras pelas linhas de modo a minimizar a soma dos

[8] 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 usados para calcular a solução do problema.

```
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.
```

Duas decomposições do mesmo texto. As linhas têm capacidade $M = 30$ 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 $4^3 + 1^3 = 65$.

cubos dos espaços em branco que sobram no fim de todas as linhas exceto a última.

1.8.1. O problema

Um **texto** é uma sequência $P_1, P_2, \ldots, P_n$ de palavras. Cada palavra P_i tem l_i caracteres. Queremos distribuir estas palavras por linhas que comportam M caracteres no máximo. Nessa distribuição, as palavras não podem ser quebradas entre linhas e cada duas palavras consecutivas devem ser separadas por um espaço. Diremos que M é a **capacidade** de uma linha e suporemos que $1 \leq l_i \leq M$ para cada i.

Para todo $i \leq k$, denotaremos por $P[i,k]$ o segmento $P_i, P_{i+1}, \ldots, P_k$ do texto. O **comprimento** desse segmento é o número

$$l[i,k] := l_i + 1 + l_{i+1} + 1 + \cdots + 1 + l_k .$$

Diremos que um segmento $P[i,k]$ **é curto** se $l[i,k] \leq M$. O **defeito** de um segmento curto $P[i,k]$ é o número

$$(M - l[i,k])^3 .$$

Uma **decomposição** do texto $P_1, P_2, \ldots, P_n$ é uma subdivisão do texto em segmentos curtos. Podemos representar a decomposição por uma sequência $\langle j_1, j_2, \ldots, j_q \rangle$ de índices tal que $1 = j_1 < j_2 < \cdots < j_q \leq n$ e os segmentos $P[j_1, j_2-1]$, $P[j_2, j_3-1]$, ..., $P[j_q, n]$ são curtos. O **defeito** de uma decomposição é a soma dos defeitos de todos os segmentos da decomposição exceto o último. Uma decomposição é **ótima** se tiver defeito mínimo.

Problema da Quebra de Linhas: Encontrar uma decomposição ótima de um texto $P_1, P_2, \ldots, P_n$.

Exercícios

1. Tente explicar por que a definição de defeito usa a terceira potência e não a primeira ou a segunda.
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.
3. Suponha que $l_i = 1$ para $i = 1, \ldots, n$. Mostre que o defeito de uma decomposição ótima é no máximo $\lfloor 2n/M \rfloor$.

1.8.2. A estrutura recursiva do problema

Convém reformular o problema de modo que o índice da primeira palavra do texto seja uma variável i e não a constante 1. Nosso problema passa a ser o de encontrar uma decomposição ótima de um texto $P_i, P_{i+1}, \ldots, P_n$.

```
                                    aaa bbb--
ccc                 bbb ccc         ccc
```

Texto P_1, P_2, P_3 em que cada palavra tem 3 caracteres. A capacidade das linhas é $M = 9$. À esquerda temos uma decomposição ótima do texto P_3. No centro, uma decomposição ótima do texto P_2, P_3. À direita, uma decomposição ótima de P_1, P_2, P_3. Os caracteres "–" representam os espaços em branco que contribuem para o defeito.

O problema tem caráter recursivo, como passamos a mostrar. Suponha que $P[i,k]$ é o primeiro segmento de uma decomposição ótima do texto $P_i, P_{i+1}, \ldots, P_n$. Se $k < n$ então

a correspondente decomposição de $P_{k+1}, P_{k+2}, \ldots, P_n$ é ótima.

A prova desta propriedade é simples. Seja x o defeito da decomposição de $P_{k+1}, \ldots, P_n$ induzida pela decomposição ótima de $P_i, \ldots, P_n$. Suponha agora, por um momento, que o texto $P_{k+1}, \ldots, P_n$ tem uma decomposição com defeito menor que x. Então a combinação desta decomposição com o segmento $P[i,k]$ produz uma decomposição de $P_i, \ldots, P_n$ que tem defeito menor que o mínimo, o que é impossível.

A propriedade recursiva pode ser traduzida em uma relação de recorrência. Seja $X(i)$ o defeito de uma decomposição ótima do texto $P_i, P_{i+1}, \ldots, P_n$. Se $l[i,n] \leq M$ então $X(i) = 0$; senão,

$$X(i) = \min_{l[i,k] \leq M} \left((M - l[i,k])^3 + X(k+1) \right), \tag{31}$$

sendo o mínimo tomado sobre todos os segmentos curtos $P[i,k]$ com $i \leq k$.

A recorrência pode ser transformada facilmente num algoritmo recursivo. Mas o cálculo do min em (31) levaria o algoritmo a resolver as mesmas subinstâncias várias vezes, o que é muito ineficiente.

1.8.3. Um algoritmo de programação dinâmica

Para usar a recorrência (31) de maneira eficiente, basta interpretar X como uma tabela e calcular $X[n]$, depois $X[n-1]$, e assim por diante. (Veja a Subseção 1.7.5.) O seguinte algoritmo aplica esta ideia e devolve o defeito mínimo do texto $P_1, P_2, \ldots, P_n$, supondo $n \geq 1$ e linhas de capacidade M:

```
DEFEITOMÍNIMO (l_1,..,l_n, M)
1  para i ← n decrescendo até 1 faça
2      X[i] ← ∞
3      k ← i
4      L ← l_k
5      enquanto k < n e L ≤ M faça
6          X' ← (M − L)^3 + X[k+1]
7          se X' < X[i] então X[i] ← X'
8          k ← k + 1
9          L ← L + 1 + l_k
10     se L ≤ M então X[i] ← 0
11 devolva X[1]
```

As primeiras execuções do bloco de linhas 5-9 terminam tipicamente com $k = n$ e assim colocam $P[i,k]$ na última linha. As execuções subsequentes do mesmo bloco terminam com $k < n$ e portanto com $L > M$. A propósito, o valor da variável L é $l[i,k]$.

O algoritmo está correto. Na linha 1, imediatamente antes da comparação de i com 1, a parte $X[i+1\,..\,n]$ da tabela já foi calculada e temos o seguinte invariante: para $j = i+1, i+2, \ldots, n$,

$$X[j] \text{ é o defeito mínimo do texto } P_j, \ldots, P_n .$$

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 texto $P_i, P_{i+1}, \ldots, P_n$ usando a recorrência (31).

No fim do processo iterativo temos $i = 0$ e portanto $X[1]$ é o defeito mínimo do texto $P_1, \ldots, P_n$.

Consumo de tempo. Adote n como tamanho da instância $(l_1, .., l_n, M)$ do problema. Podemos supor que uma execução de qualquer linha do código 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 i, o bloco de linhas 6-9 é executado no máximo $n-i$ vezes. Como i decresce, no pior caso, de n a 1, o número total de execuções deste bloco de linhas 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 $l_i \geq \lceil M/2 \rceil$ para cada i), o algoritmo consome $\Theta(n)$ unidades de tempo.

Exercício

4. Modifique o algoritmo DEFEITOMÍNIMO para que ele devolva uma decomposição ótima $\langle j_1, \ldots, j_q \rangle$ de $P_1, \ldots, P_n$ e não apenas o defeito da decomposição.

1.9. Escalonamento de intervalos

Imagine que vários eventos querem usar um certo centro de convenções. Cada evento gostaria de usar o centro durante o intervalo de tempo que começa numa data a e termina numa data b. Dois eventos são considerados compatíveis se os seus intervalos de tempo são disjuntos. Cada evento tem um certo valor e a administração do centro precisa selecionar um conjunto de eventos mutuamente compatíveis que tenha o maior valor total possível.

1.9.1. O problema

Sejam $a_1, a_2, \ldots, a_n$ e $b_1, b_2, \ldots, b_n$ números naturais tais que $a_i \leq b_i$ para cada i. Cada índice i representa o intervalo que tem origem a_i e término b_i, ou seja, o conjunto $\{a_i, a_i+1, \ldots, b_i-1, b_i\}$.

Dois intervalos distintos i e j são **compatíveis** se $b_i < a_j$ ou $b_j < a_i$. Um conjunto de intervalos é **viável** se seus elementos são compatíveis dois a dois.

A cada intervalo i está associado um número natural v_i que representa o **valor** do intervalo. O **valor** de um conjunto S de intervalos é o número $v(S) := \sum_{i \in S} v_i$. Um conjunto viável S **tem valor máximo** se $v(S) \geq v(S')$ para todo conjunto S' de intervalos que seja viável.

> Problema dos Intervalos Compatíveis de Valor Máximo: Dados números naturais $a_1, \ldots, a_n$, $b_1, \ldots, b_n$ e $v_1, \ldots, v_n$ tais que $a_i \leq b_i$ para cada i, encontrar um subconjunto viável de $\{1, \ldots, n\}$ que tenha valor máximo.

A primeira ideia que vem à mente é a de um algoritmo guloso[9] que escolhe os intervalos em ordem decrescente de valor. Mas isso não resolve o problema.

[9] Um algoritmo guloso constrói uma solução escolhendo, a cada passo, o objeto mais "saboroso" de acordo com algum critério estabelecido a priori.

Exercício

1. Coloque os intervalos em ordem decrescente de valor, ou seja, suponha que $v_1 \geq v_2 \geq \cdots \geq v_n$. A m-ésima iteração do algoritmo começa com um subconjunto viável S de $\{1,\ldots,m-1\}$. Se $S \cup \{m\}$ for viável, comece nova iteração com $S \cup \{m\}$ no lugar de S. Senão, comece nova iteração sem alterar S. Mostre que este algoritmo guloso não resolve o problema.

1.9.2. A estrutura recursiva do problema

Adote a abreviatura **svvm** para a expressão "subconjunto viável de valor máximo" e seja S um svvm do conjunto de intervalos $\{1,\ldots,n\}$. Se S não contém n então S também é um svvm de $\{1,\ldots,n-1\}$. Se S contém n então $S-\{n\}$ é um svvm do conjunto de todos os intervalos que são compatíveis com n.

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

Para simplificar a descrição do conjunto de intervalos compatíveis com n, convém supor que os intervalos estão em ordem crescente de términos, ou seja, que

$$b_1 \leq b_2 \leq \cdots \leq b_n . \tag{32}$$

Sob esta hipótese, o conjunto dos intervalos compatíveis com n é simplesmente $\{1,\ldots,j\}$, sendo j o maior índice tal que $b_j < a_n$. Se denotarmos por $X(n)$ o valor de um svvm de $\{1,\ldots,n\}$, a estrutura recursiva do problema garante que

$$X(n) = \max\big(X(n-1),\, X(j)+v_n\big), \tag{33}$$

sendo j o maior índice tal que $b_j < a_n$. Se tal j não existe então (33) se reduz a $X(n) = \max\big(X(n-1),\, v_n\big)$. Esta recorrência pode ser facilmente transformada em um algoritmo recursivo. Mas o algoritmo é muito ineficiente, pois resolve cada subinstância repetidas vezes.

Exercício

2. Escreva um algoritmo recursivo baseado em (33). Mostre que o consumo de tempo do algoritmo no pior caso está em $\Omega(2^n)$. (Sugestão: Considere um conjunto com n intervalos compatíveis dois a dois, todos com valor 1. Mostre que o tempo $T(n)$ que o algoritmo consome para processar o conjunto satisfaz a recorrência $T(n) = T(n-1)+T(n-1)+1$.)

1.9.3. Algoritmo de programação dinâmica

Para tirar bom proveito da recorrência (33), é preciso recorrer à técnica da programação dinâmica, que consiste em guardar as soluções das subinstâncias numa tabela à medida que elas forem sendo resolvidas. (Veja a Subseção 1.7.5.) Com isso, cada subinstância será resolvida uma só vez.

O seguinte algoritmo recebe n intervalos que satisfazem (32) e devolve o valor de um svvm de $\{1,\ldots,n\}$:

```
INTERVALOSCOMPATÍVEIS (a_1,..,a_n, b_1,..,b_n, v_1,..,v_n)
1  X[0] ← 0
2  para m ← 1 até n faça
3      j ← m − 1
4      enquanto j ≥ 1 e b_j ≥ a_m faça
5          j ← j − 1
6      se X[m − 1] > X[j] + v_m
7          então X[m] ← X[m − 1]
8          senão X[m] ← X[j] + v_m
9  devolva X[n]
```

O algoritmo está correto. Na linha 2, imediatamente antes da comparação de m com n, $X[m-1]$ é o valor de um svvm de $\{1,\ldots,m-1\}$. Mais que isso, para todo i entre 0 e $m-1$,

$$X[i] \text{ é o valor de um svvm de } \{1,\ldots,i\}.$$

(Se $i = 0$, o conjunto é vazio.) Este invariante certamente vale no início da primeira iteração, quando $m = 1$. Suponha agora que ele vale no início de uma iteração qualquer. Para mostrar que continua valendo no início da iteração seguinte, observe que, no começo da linha 6, $\{1,\ldots,j\}$ é o conjunto dos intervalos compatíveis com m. Assim, o valor atribuído a $X[m]$ nas linhas 7 e 8 está de acordo com a recorrência (33).

Na última passagem pela linha 2 temos $m = n+1$ e portanto $X[n]$ é o valor de um svvm de $\{1,\ldots,n\}$. Assim, o algoritmo cumpre o que prometeu.

Consumo de tempo. Uma execução de qualquer linha do algoritmo INTERVALOSCOMPATÍVEIS consome uma quantidade de tempo que não depende de n. Logo, o consumo de tempo total do algoritmo pode ser medido contando o número de execuções das diversas linhas.

Para cada valor fixo de m, a linha 5 é executada no máximo $m-1$ vezes. Como m varia de 1 a n e $\sum_{m=1}^{n}(m-1) < n^2$, a linha 5 é executada menos que n^2 vezes. Todas as demais linhas são executadas no máximo $n-1$ vezes. Logo, o consumo de tempo total do algoritmo está em

$$O(n^2)$$

no pior caso. Uma análise um pouco mais cuidadosa mostra que o consumo está em $\Theta(n^2)$ no pior caso, e portanto o algoritmo é quadrático. (Mas veja o Exercício 3).

Não levamos em conta ainda o pré-processamento que rearranja os intervalos em ordem crescente de término. Como esse pré-processamento pode ser feito em tempo $O(n \lg n)$ (veja Seção 1.4) e $n \lg n$ está em $O(n^2)$, o consumo de tempo total é $\Theta(n^2)$. (Mas veja o Exercício 3.)

Exercícios

3. Mostre que o algoritmo INTERVALOSCOMPATÍVEIS pode ser implementado de modo a consumir apenas $O(n)$ unidades de tempo. Sugestão: construa uma tabela auxiliar que produza, em tempo constante, efeito equivalente ao das linhas 4-5.
4. Escreva um algoritmo de pós-processamento que extraia da tabela $X[1..n]$ produzida pelo algoritmo INTERVALOSCOMPATÍVEIS um svvm de $\{1,\ldots,n\}$. O seu algoritmo deve consumir $O(n)$ unidades de tempo.

1.10. 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, mas que não ultrapasse 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.

1.10.1. 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 M, um subconjunto S de $\{1,\ldots,n\}$ é **viável** $p(S) \leq M$. Um subconjunto S de $\{1,\ldots,n\}$ é **ótimo** se é viável e tem valor máximo, ou seja, se não existe conjunto viável S' tal que $v(S') > v(S)$.

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

Uma **solução** da instância (n, p, M, 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 tempo $\Omega(2^n)$.

1.10.2. A estrutura recursiva do problema

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

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 $X(n,M)$ o valor de um solução de (n, p, M, v), a estrutura recursiva do problema leva à seguinte recorrência:

$$X(n,M) = \max\left(X(n-1,M),\, X(n-1,M-p_n)+v_n\right). \tag{34}$$

O segundo termo de max só faz sentido se $M - p_n \geq 0$. Logo,

$$X(n,M) = X(n-1,M) \tag{35}$$

quando $p_n > M$. Podemos calcular $X(n,M)$ recursivamente. Mas o algoritmo resultante é muito ineficiente, pois resolve cada subinstância um grande número de vezes.

Exercício

1. Escreva um algoritmo recursivo baseado na recorrência (34-35). Calcule o consumo de tempo do seu algoritmo no pior caso. (Sugestão: para cada n, tome a instância em que $M = n$ 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 $T(n) = 2T(n-1) + 1$.)

1.10.3. Algoritmo de programação dinâmica

Para obter um algoritmo eficiente a partir da recorrência (34-35), é preciso armazenar as soluções das subinstâncias numa tabela à medida que elas forem sendo obtidas, evitando assim que elas sejam recalculadas. (Veja a Subseção 1.7.5.)

Como M e todos os p_i são números naturais, podemos tratar X como uma tabela com linhas indexadas por $0..n$ e colunas indexadas por $0..M$. Para cada i entre 0 e n e cada L entre 0 e M, a casa $X[i,L]$ da tabela será o valor de uma solução da instância (i, p, L, 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 recebe uma instância (n, p, M, v) do problema e devolve o valor de uma solução da instância:

```
MochilaPD (n, p, M, v)
1  para L ← 0 até M faça
2      X[0,L] ← 0
3      para m ← 1 até n faça
4          a ← X[m−1,L]
5          se L − p_m ≥ 0
6              então b ← X[m−1, L − p_m] + v_m
7                  se a < b então a ← b
8          X[m,L] ← a
9  devolva X[n,M]
```

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

$$X[i,L'] \text{ é o valor de uma solução da instância } (i, p, L', v) \tag{36}$$

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

Uma instância do Problema da Mochila com $n = 4$ e $M = 5$. A figura mostra a tabela $X[0..n, 0..M]$ calculada pelo algoritmo MOCHILAPD.

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

$$X[i,L] \text{ é o valor de uma solução da instância } (i, p, L, v) \tag{37}$$

para todo i no intervalo $0..m-1$. Se o invariante (37) vale no início de uma iteração, ele continua valendo no início da iteração seguinte em virtude da recorrência (34-35). Isto prova que (36) de fato vale a cada passagem pela linha 1.

Na última passagem pela linha 1 temos $L = M + 1$ e portanto (36) garante que $X[n,M]$ é o valor de uma solução da instância (n, p, M, v).

Consumo de tempo. É razoável supor que uma execução de qualquer das linhas do código consome uma quantidade de tempo que não depende de n nem de M. Portanto, basta contar o número de execuções das diversas linhas.

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

$$\Theta(nM)$$

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 M. Imagine, por exemplo, que M 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 $\Theta(nM)$, estamos implicitamente adotando o par de números (n, M) como tamanho da instância (n, p, M, v). No entanto, é mais razoável dizer que o tamanho da instância é $(n, \lceil \lg M \rceil)$, pois M pode ser escrito com apenas $\lceil \lg M \rceil$ bits. É mais razoável, portanto, dizer que o algoritmo consome

$$\Theta(n2^{\lg M})$$

unidades de tempo. Isto torna claro que o consumo de tempo cresce explosivamente em função de $\lg M$.

Gostaríamos de ter um algoritmo cujo consumo de tempo não dependesse do valor de M, 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.[10]

Exercícios

2. É verdade que $X[i,0] = 0$ para todo i depois da execução do algoritmo MOCHILAPD?
3. Por que nosso enunciado do problema inclui instâncias em que M vale 0? Por que inclui instâncias com objetos de peso nulo?
4. Mostre como extrair da tabela $X[0..n,0..M]$ produzida pelo algoritmo MOCHILAPD um subconjunto ótimo de $\{1,\ldots,n\}$.
5. Faça uma mudança de escala para transformar o conjunto $\{0,1,\ldots,M\}$ 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?

1.10.4. Instâncias especiais do problema da mochila

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 M 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. Esta coleção de instâncias pode ser resolvida por um algoritmo guloso óbvio que é muito mais eficiente que MOCHILAPD. (Você pode imaginar que $p_1, \ldots, p_n$ são os tamanhos de n arquivos digitais que você gostaria de gravar num CD de capacidade M. Quantos arquivos cabem no CD?)

1.11. Solução aproximada da mochila

O algoritmo de programação dinâmica para o Problema da Mochila (veja a Seção 1.10) é lento. Dada a dificuldade de encontrar um algoritmo mais rápido,

[10] O Problema da Mochila é NP-difícil. Veja o livro de Cormen et al.

faz sentido procurar por um algoritmo veloz que produza uma solução "aproximada" do problema. Um tal algoritmo deveria calcular um conjunto viável de valor maior que uma fração predeterminada (digamos 50%) do valor máximo.

1.11.1. O problema

Convém repetir algumas das definições da Subseção 1.10.1. Dados números naturais $p_1, \ldots, p_n$, M 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** $p(S) \leq M$. O **valor** de S é o número $v(S) := \sum_{i\in S} v_i$. Um conjunto viável S é **ótimo** se tem valor máximo.

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

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

$$1 \leq p_i \leq M \tag{38}$$

para todo i.

1.11.2. Algoritmo de aproximação

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

$$\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n}. \tag{39}$$

O algoritmo recebe uma instância do problema que satisfaz as condições (38) e (39) e devolve o valor de um subconjunto viável X de $\{1,\ldots,n\}$ tal que

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

sendo X^* é um subconjunto ótimo:

```
MochilaAprox (n, p, M, v)
1  s ← x ← 0
2  i ← 1
3  enquanto i ≤ n e s + p_i ≤ M faça
4      s ← s + p_i
5      x ← x + v_i
6      i ← i + 1
7  k ← 1
8  para i ← 2 até n faça
9      se v_i > v_k então k ← i
10 devolva max(x, v_k)
```

O algoritmo está correto. No bloco de linhas 1-6, o algoritmo escolhe, implicitamente, o maior m tal que $\{1,\ldots,m\}$ é viável. Seja $X := \{1,\ldots,m\}$ e observe que $v(X) = x$.

Nas linhas 7-9, o algoritmo escolhe um índice k tal que $v_k \geq v_i$ para todo i. Graças à hipótese (38), o conjunto $\{k\}$ é viável. Na linha 10, o algoritmo devolve o maior dentre $v(X)$ e v_k. Resta mostrar que

$$\max(v(X), v_k) \;>\; \tfrac{1}{2}\, v(S) \tag{40}$$

para todo conjunto viável S. Se $m = n$, a desigualdade (40) é óbvia. Suponha então que $m < n$ e seja $Y := X \cup \{m+1\}$. Observe que

$$\max(v(X), v_k) \;\geq\; \tfrac{1}{2}(v(X) + v_k) \;\geq\; \tfrac{1}{2}(v(X) + v_{m+1}) \;=\; \tfrac{1}{2}\, v(Y).$$

Resta mostrar que $v(Y) > v(S)$ para qualquer conjunto viável S. Isto pode ser feito assim:

$$\begin{aligned}
v(Y) - v(S) &= v(Y - S) - v(S - Y) \\
&= \sum_{i \in Y - S} \frac{v_i}{p_i}\, p_i - \sum_{i \in S - Y} \frac{v_i}{p_i}\, p_i \\
&\geq \frac{v_{m+1}}{p_{m+1}}\, p(Y - S) \;-\; \frac{v_{m+1}}{p_{m+1}}\, p(S - Y) \qquad (41) \\
&= \frac{v_{m+1}}{p_{m+1}} \big(p(Y) - p(S)\big) \\
&> \frac{v_{m+1}}{p_{m+1}} \big(M - M\big) \qquad (42) \\
&= 0.
\end{aligned}$$

A relação (41) vale porque $v_i/p_i \geq v_{m+1}/p_{m+1}$ para todo i em Y e $v_i/p_i \leq v_{m+1}/p_{m+1}$ para todo i no complemento de Y. Já (42) vale porque $p(Y) > M$ e $p(S) \leq M$.

Consumo de tempo. Adote n (poderia igualmente bem adotar $2n+1$) como medida do tamanho da instância (n, p, M, v) do problema. Uma execução de qualquer das linhas do algoritmo consome tempo independente de n. O bloco de linhas 4-6 é executado no máximo n vezes e a linha 9 é executada $n - 1$ vezes. Assim, o algoritmo todo consome

$$\Theta(n)$$

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

O pré-processamento necessário para fazer valer (38) e (39) pode ser feito em tempo $\Theta(n \lg n)$ (veja a Seção 1.4). Portanto, o consumo de tempo do processo todo é

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

Exercício

1. Construa uma instância do Problema da Mochila para a qual o número x calculado por MOCHILAAPROX é estritamente menor que $\frac{1}{2}v(X^*)$, sendo X^* um conjunto ótimo.

1.11.3. Observações sobre algoritmos de aproximação

Um algoritmo rápido cujo resultado é uma fração predeterminada da solução ótima é conhecido como **algoritmo de aproximação**. O resultado do algoritmo MOCHILAAPROX, 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., 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 é baseado numa variante de MOCHILAPD em que os papéis de p e v são trocados. (Veja o Exercício 5 na Seção 1.10.)

1.12. 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. Este é um célebre problema de otimização em grafos.

1.12.1. 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.

1.12.2. 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 em $\mathbb{N}$, 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 con-

sumo 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.[11]

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 Subseção 1.11.3.)

Uma instância do Problema da Cobertura. Encontre uma cobertura mínima, supondo que todos os vértices têm custo 1.

1.12.3. 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      x_i ← 0
3  para cada ij em A faça
4      y_ij ← 0
5  para cada pq em A faça
6      e ← min(c_p − x_p , c_q − x_q)
7      y_pq ← y_pq + e
8      x_p ← x_p + e
9      x_q ← x_q + e
10 X ← ∅
11 para cada i em V faça
12     se x_i = c_i então X ← X ∪ {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 $\sum_q y_{pq} \geq c_p$, sendo a soma feita sobre todas as arestas que têm ponta p. A regra do jogo é nunca pagar mais que o necessário. Portanto, $\sum_q y_{pq} \leq c_p$ para todo vértice p.

[11] O Problema da Cobertura é NP-difícil. Veja o livro de Cormen et al.

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. $x_i \leq c_i$ 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 $e \leq c_p - x_p$ e $e \leq c_q - x_q$. 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

$$\sum_{ij \in A} y_{ij} \leq c(Z) \tag{43}$$

para qualquer cobertura Z. Para mostrar isso, basta observar que $\sum_{ij \in A} y_{ij} \leq \sum_{i \in Z} \sum_j y_{ij} = \sum_{i \in Z} x_i \leq \sum_{i \in Z} c_i$. 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

$$c(X) \leq 2 \sum_{ij \in A} y_{ij}. \tag{44}$$

De fato, como $x_i = c_i$ para todo i em X, temos $\sum_{i \in X} c_i = \sum_{i \in X} x_i = \sum_{i \in X} \sum_j y_{ij} \leq 2 \sum_{ij \in A} y_{ij}$. 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 (44) e (43) que, depois do bloco de linhas 10-12, para qualquer cobertura Z,

$$c(X) \leq 2c(Z).$$

Isto vale, em particular, se Z é uma cobertura de custo mínimo. Mas X é uma cobertura, em virtude do invariante iii. Portanto, ao devolver X, 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.

Consumo de tempo. Podemos supor que o grafo é dado da maneira mais crua possível: os vértices são 1, 2, ..., 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 algoritmo 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.

1.12.4. 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 (43) é uma manifestação da dualidade de programação linear.

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.

Exercício

1. 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.

1.12.5. 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 o livro de Cormen et al., 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.

1.13. 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 na Seção 1.9: encontrar um conjunto máximo de intervalos dois a dois compatíveis.

1.13.1. 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 Seção 1.12). 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 (como o que discutimos na Subseção 1.12.3) 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 fração (ou seja, 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.

1.13.2. 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 $0 \leq 2m \leq n^2 - n$. 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	$n/(2n-2)$

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

- $|I| \geq n/2$ se $m \leq n/2$ e
- $\mathbb{E}[|I|] \geq n^2/4m$ 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: $X[i] = 1$ se e somente se i pertence ao conjunto independente.

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 pseudo-aleatórios.)

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 $n^2/4m$.

Consumo de tempo. O algoritmo é linear. É razoável supor que cada execução da rotina RANDOM consome tempo independente de n e 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.

1.13.3. Comentários sobre algoritmos probabilísticos

Diz-se que um algoritmo é **probabilístico** 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, obtêm resultados interessantes.

1.14. Busca em largura num grafo

Considere a relação "amigo de" entre os usuários de uma rede social (como o Orkut, por exemplo). 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 Subseção 1.12.1).

1.14.1. 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: 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.

1.14.2. *Busca em largura: versão preliminar*

Usaremos a estratégia 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 deve 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 es-

tão 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.

Consumo de tempo. 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

1. Um grafo é **conexo** se seus vértices estão ligados dois a dois. Escreva um algoritmo que decida se um grafo é conexo.
2. Escreva um algoritmo que calcule o número de componentes de um grafo.

1.14.3. 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 $1, 2, \ldots, n$. As vizinhanças dos vértices serão representadas por uma matriz Z com linhas indexadas pelos vértices e colunas indexadas por $1, 2, \ldots, n-1$. Os vizinhos de um vértice i serão

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

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 subseção 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 $1, 2, \ldots, n$, 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 código não faz mais que implementar o algoritmo COMPONENTEPRELIM, que já discutimos.

Consumo de tempo. 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 algoritmo 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.)

Exercício

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.)

1.15. Busca em profundidade num grafo

Trataremos agora de um segundo algoritmo para o problema estudado na seção anterior.

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

O algoritmo que discutiremos a seguir usa a estratégia 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.

1.15.1. 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 Subseção 1.14.2.

```
COMPONENTE (n, Z, v)
1  para p ← 1 até n faça
2      cor[p] ← 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.[12]

[12] Ou seja, a rotina pode consultar e alterar o valor das variáveis.

BUSCAEMPROFUNDIDADE (p)
5 $cor[p] \leftarrow 2$
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 Subseção 1.14.1) 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 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

$$\sum_{x \in X} g(x), \tag{45}$$

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,

$$\text{o conjunto dos vértices brancos é } B - \big(\{p\} \cup Y_1 \cup \cdots \cup Y_{j-1}\big). \qquad (46)$$

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 (46) continua valendo no início da próxima iteração.

No fim do processo iterativo, o invariante (46) 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.

Consumo de tempo. 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 m de arestas do grafo, podemos dizer que a linha 3 de COMPONENTE

$$O(m)$$

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

$$\Theta(n+m)$$

no pior caso.

Exercício

1. Escreva e analise uma versão iterativa do algoritmo BUSCAEMPROFUNDIDADE.

1.16. Conclusões

O presente 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 o texto 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.

Agradecimentos

Cristina Gomes Fernandes e José Coelho de Pina Jr. (ambos do Departamento de Ciência da Computação do Instituto de Matemática e Estatística da USP) contribuíram de muitas maneiras com o presente texto, fornecendo material, ideias e valiosas sugestões. A eles o meu muito obrigado!

Agradeço também aos dois revisores(as) anônimos(as) pelas valiosas sugestões e pela correção de diversos erros (alguns graves...).

Referências bibliográficas

[Bentley 2000] Bentley, J. L. (2000). Programming Pearls. ACM Press, second edition.

[Brassard and Bratley 1996] Brassard, G. and Bratley, P. (1996). Fundamentals of Algorithmics. Prentice Hall.

[Cormen et al. 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). Introduction to Algorithms. MIT Press and McGraw-Hill, second edition.

[Dasgupta et al. 2006] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2006). Algorithms. McGraw-Hill.

[Feofiloff 2009b] Feofiloff, P. (1999–2009b). Análise de Algoritmos. Internet: `www.ime.usp.br/~pf/analise_de_algoritmos/`.

[Feofiloff 2009a] Feofiloff, P. (2009a). Algoritmos em linguagem C. Campus/Elsevier.

[Kleinberg and Tardos 2005] Kleinberg, J. and Tardos, E. (2005). Algorithm Design. Addison-Wesley.

[Knuth 1973] Knuth, D. E. (1973). The Art of Computer Programming. Addison-Wesley. Several volumes.

[Manber 1989] Manber, U. (1989). Introduction to Algorithms: A Creative Approach. Addison-Wesley.

[Mitzenmacher and Upfal 2005] Mitzenmacher, M. and Upfal, E. (2005). Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press.

[Neapolitan and Naimipour 1998] Neapolitan, R. and Naimipour, K. (1998). Foundations of Algorithms. Jones and Bartlett, second edition.

[Parberry 1995] Parberry, I. (1995). Problems on Algorithms. Prentice Hall.

[Parberry and Gasarch 2002] Parberry, I. and Gasarch, W. (2002). Problems on Algorithms. Internet: `www.eng.unt.edu/ian/books/free/`.

[Szwarcfiter e Markenzon 1994] Szwarcfiter, J. e Markenzon, L. (1994). Estruturas de Dados e seus Algoritmos. Livros Técnicos e Científicos, segunda edição.

[Ziviani 2004] Ziviani, N. (2004). Projeto de Algoritmos. Thomson, segunda edição.