Capítulo 4

Análise de algoritmos iterativos

4.1 Introdução

Neste capítulo, iremos retomar a discussão da seção 1.2 e iniciar nosso estudo sobre as técnicas de análise de algoritmos. O principal motivo para efetuar a análise de um algoritmo é prever suas necessidades de recursos (principalmente memória e tempo de execução) de forma a poder compará-lo com outros algoritmos que resolvem o mesmo problema. Este tipo de análise é importante porque existem diferenças entre algoritmos que nunca poderão ser superadas por melhorias no *hardware*.

A quantidade de recursos consumidos por um algoritmo é sua *complexidade computacional*. Em geral, o recurso que nos interessa é o tempo de execução. Quando for necessário tratar algum outro aspecto (por exemplo, o consumo de memória), isto estará explícito no texto. Lembre-se que a complexidade C é função do tamanho n da entrada:

$$C = f(n).$$

Como determinar a complexidade? Podemos pensar em uma abordagem bastante direta: implemente o algoritmo em alguma linguagem de programação, execute-o e cronometre o tempo de execução; faça isto para entradas de diferentes tamanhos e acompanhe a variação de C com n. Esta estratégia experimental é válida e será vista no capítulo 9. Entretanto, ela requer uma série de cuidados para fornecer resultados relevantes, e inicialmente iremos preferir uma estratégia que se concentre nas características essenciais do algoritmo e não dependa de uma implementação particular. Goodrich e Tamassia (2004, p. 22) listam as características desejáveis de uma metodologia analítica, contrastando-as com a abordagem experimental:

1. Todas as entradas possíveis para o algoritmo devem ser levadas em conta. Na análise experimental, obviamente, só se pode fazer experimentos para um número limitado de casos de teste.
2. A metodologia deve permitir que dois algoritmos possam ser comparados de forma independente do ambiente de execução. Experimentalmente, só podemos comparar algoritmos executando sob o mesmo *hardware* e *software*.
3. Deve ser possível realizar a análise a partir de uma descrição de alto nível do algoritmo, sem preocupação com detalhes de implementação; na análise experimental, detalhes desse tipo devem ser levados em conta.

Para atender a estas características, a metodologia utilizada deve ter os seguintes componentes (GOODRICH; TAMASSIA, 2004, p. 22):

1. Uma linguagem adequada para a descrição dos algoritmos, que seja suficientemente estruturada, porém não desça a detalhes excessivos. O pseudo-código descrito na seção 1.1 é adequado para os nossos propósitos.

2. Um modelo computacional onde o pseudo-código possa ser "executado", e que forneça a base necessária para o cálculo da complexidade.
3. Uma métrica para medir o tempo de execução dos algoritmos. Conforme veremos, uma métrica adequada ao modelo computacional acima não utiliza as unidades de tempo mais comuns (segundos, milissegundos, etc.), mas a *contagem de instruções* executadas pelo algoritmo.
4. Uma técnica para efetuar o cálculo dos tempos de execução. Esta técnica é o principal tema deste capítulo, embora, inicialmente, apenas algoritmos iterativos sejam discutidos. As técnicas para análise de algoritmos recursivos serão vistas no capítulo 7.

Com exceção do pseudo-código, já detalhado anteriormente, todos esses componentes serão discutidos a seguir.

O modelo computacional

Um modelo computacional é um modelo simplificado de um computador do mundo real, que nos livra dos detalhes de baixo nível de uma máquina real e permite que nos concentremos apenas nos aspectos relevantes para a análise de algoritmos. O modelo computacional resume aquilo que se assume sobre as capacidades do dispositivo que irá processar os algoritmos.

Um modelo muito utilizado é a máquina RAM (ver Goodrich e Tamassia (2004, p. 24–25) e Cormen et al. (2002, p. 16–17)). A sigla RAM significa *random-access machine* e não deve ser confundida com a memória RAM (*random-access memory*), mais conhecida. Neste modelo, o computador é constituído de uma máquina contendo uma UCP (unidade central de processamento) conectada a uma memória cujas células podem ser acessadas a qualquer momento em um único passo (daí seu nome)[1]. Outras características importantes são:

1. A máquina não realiza processamento concorrente de instruções.
2. Cada posição de memória armazena uma palavra, que pode conter um tipo de dados básico (número inteiro ou de ponto flutuante, caracter, endereço, etc.) Entretanto, não há limite para o tamanho dos números que podem ser armazenados em uma palavra.
3. Não existe hierarquia de memória (*cache*, principal, virtual, etc.)
4. As instruções primitivas são aquelas comumente encontradas em uma máquina real: instruções aritméticas (adição, subtração, multiplicação, divisão), lógicas (e, ou, não, ou exclusivo), movimentação de dados (armazenamento, cópia, etc.), controle (desvio incondicional e condicional, chamada e retorno de sub-rotinas), etc.
5. O tempo de execução das instruções primitivas é constante e independe do tamanho dos operandos.

Esta última característica é muito importante e nos permite chegar à seguinte afirmativa:

> *O número de operações primitivas que um algoritmo executa é proporcional a seu tempo de execução.*

As implicações deste fato levarão à métrica discutida na seção seguinte.

Não utilizaremos diretamente a máquina RAM como modelo computacional, mas adotaremos uma abordagem equivalente. Utilizaremos um conjunto de operações fundamentais de mais alto nível, correspondente ao conjunto de instruções de nosso pseudo-código, e consideraremos que o tempo de execução dessas instruções também é constante, de modo a poder estimar o tempo de execução de um algoritmo através de uma contagem de instruções tal como descrito a seguir.

[1] Este fato torna-se relevante quando constrastado com outros modelos computacionais, tais como as máquinas de Turing, nas quais as células de memória não são diretamente acessíveis: para acessar uma determinada célula, todas as intermediárias devem ser lidas em seqüência. Não iremos descer a detalhes porque isto nos levaria para fora do escopo deste trabalho.

Uma métrica para o tempo de execução

Da discussão da seção anterior, vimos que todas as instruções fundamentais do modelo computacional adotado são executadas em tempo constante, e que o tempo total é proporcional ao número de instruções executadas. Assim, podemos estimar o tempo de execução de um algoritmo apenas contando o número de instruções que ele executa[2].

Por exemplo, seja o algoritmo 1.2, analisado na seção 1.2. Vimos que sua complexidade de tempo é:

$$C = n(t_1 + t_2 + t_3) + (t_1 + t_3 + t_4),$$

onde t_1, t_2, t_3 e t_4 são os tempos das diversas instruções executadas pelo algoritmo. Suponha que a instrução fundamental mais rápida de nosso modelo seja executada em um tempo t_m, e a mais demorada em t_M. Assim, o tempo de execução do algoritmo é limitado por:

$$3t_m(n+1) \leqslant C \leqslant 3t_M(n+1).$$

Escolhendo o limite superior e adotando t_M como 1 temos:

$$C = 3(n+1)$$

que corresponde ao número de instruções executadas.

Isto significa que, ao medir a complexidade de um algoritmo, adotamos como medida de tempo não as unidades comuns utilizadas em Física (o *segundo* e seus múltiplos e submúltiplos), mas o número de instruções, e esta será a métrica temporal utilizada por nossa metodologia analítica. Em resumo:

> *Para calcular a complexidade de tempo de um algoritmo, conte o número de instruções executadas.*

Repare que, ao adotarmos esta métrica, estamos fazendo uma série de simplificações, e podemos nos perguntar se os resultados obtidos terão alguma relevância prática. A resposta a esta questão depende de nossos objetivos. Em geral, ao utilizarmos o método analítico, estamos interessados em obter estimativas do tempo de execução que nos permitam selecionar, dentre as várias alternativas, aquelas que merecem um esforço de codificação. Por exemplo, sejam os algoritmos a_1, a_2 e a_3 que resolvem de forma correta o mesmo problema. Após analisarmos todos eles, verificamos que a complexidade de a_3 é muito superior à dos demais, de modo que podemos eliminá-lo. Mas se as complexidades de a_1 e a_2 forem próximas, a melhor estratégia seria codificar ambos e realizar uma análise experimental que nos permitisse selecionar o mais conveniente. Além disso, como veremos mais adiante, freqüentemente estaremos mais interessados em comparar as taxas de crescimento das funções de complexidade que em seus valores exatos. Assim, vemos que a análise deve ser rápida e fácil de fazer, caso contrário será melhor dispender nossos esforços implementando diretamente os algoritmos envolvidos, e daí a adoção das simplificações mencionadas[3].

Podemos formalizar todos estes conceitos de forma a dar-lhes um grau maior de rigor matemático. Os leitores que não apreciam este tipo de tratamento podem pular diretamente para o início da seção seguinte. Sejam portanto[4]:

1. F: conjunto de todas as operações fundamentais utilizadas no modelo computacional adotado.
2. F*: conjunto de todas as seqüências de operações fundamentais (ou seja, conjunto de todas as seqüências de execução).

[2] Segundo Goodrich e Tamassia (2004, p. 28), cada passo em uma descrição de alto nível de um algoritmo (pseudo-código ou linguagem de programação) "corresponde a um pequeno número de operações primitivas que não depende do tamanho da entrada." Assim, a análise pela contagem das intruções de alto nível estima o número de instruções primitivas executadas, a menos de um fator constante.

[3] Sedgewick e Flajolet (1996) defendem objetivos mais ambiciosos para a metodologia analítica. Voltaremos a esta discussão mais adiante.

[4] O tratamento formal utilizado baseia-se em Toscani e Veloso (2002, p. 10–13).

3. A: conjunto de todos os algoritmos.
4. D: conjunto de todos os dados de entrada para os algoritmos.

A partir desses conjuntos, podemos fazer as seguintes definições:

- *Função execução* (e): a execução de um algoritmo é definida como uma função $e : A \times D \mapsto F^*$ tal que $e(a, d)$ é a seqüência de operações fundamentais efetuadas na execução do algoritmo $a \in A$ com entrada $d \in D$.
- *Função custo* (c): o custo de uma seqüência de instruções é uma função $c : F^* \mapsto \mathbb{N}$ tal que $c(s)$, $s \in F^*$, é o comprimento da seqüência s (ou seja, a quantidade de instruções que a compõem)[5].
- *Função desempenho* (p): o desempenho de um algoritmo $a \in A$ ao processar a entrada $d \in D$ é uma função $p : D \mapsto \mathbb{N}$ definida, para a fixo, como o custo do processamento de d por a:

$$p(d) = c(e(a, d)).$$

- *Função tamanho* (t): a função $t : D \mapsto \mathbb{N}$ fornece o tamanho de uma entrada $d \in D$ (tamanho de um vetor, número de vértices de um grafo, etc).

Finalmente, podemos definir a complexidade C_p como uma função $C_p : \mathbb{N} \mapsto \mathbb{N}$ que fornece o desempenho do algoritmo $a \in A$ para entradas de um determinado tamanho $n \in \mathbb{N}$:

$$C_p(n) = p(d) \text{ para } t(d) = n$$

ou, para colocar na notação que vínhamos utilizando,

$$C = C_p(n).$$

Entretanto, como veremos no capítulo 6, esta última definição precisará sofrer alguns refinamentos.

4.2 Contagem de instruções

Vamos estabelecer alguns princípios de contagem e, em seguida, aplicá-los à análise de algoritmos simples. A lista a seguir deve ser considerada como uma simples orientação, uma vez que não existem regras rígidas para a realização de uma análise. Em cada caso, indicaremos como contar as construções do pseudo-código de duas formas: uma mais rigorosa e outra simplificada.

1. Expressões

   Abordagem rigorosa: contar uma instrução para cada operador da expressão (seja qual for o tipo). Por exemplo, a expressão

   $$(t \neq -1) \textbf{ e } ((3 - t) \cdot (x/3)/(1 + t) > 5)$$

   possui 5 operadores aritméticos (uma subtração, uma multiplicação, duas divisões e uma adição), 2 operadores relacionais (um $\neq$ e um $>$) e 1 operador lógico (**e**), e deve ser contada como 8 instruções. Se a expressão envolver alguma chamada de procedimento, ver a regra 9 abaixo.

   Abordadem simplificada: a complexidade de execução de uma expressão não é levada em conta, não importando quão complexa. Esta complexidade será "embutida" no cálculo da instrução associada (atribuição, retorno de procedimento, etc). Entretanto, a complexidade de chamadas de procedimento ainda deve ser considerada.

[5] Se necessário, o cálculo de $c(s)$ pode levar em conta o peso de cada uma das instruções da seqüência.

2. Instruções simples

   Abordagem rigorosa: comandos simples tais como atribuições, comandos de entrada e saída e instruções de retorno de funções (**retorna**) contam como 1 mais a complexidade da expressão associada (contada como mostrado acima). Exemplos:

   $a \leftarrow b + 1$;
   retorna $(a \cdot b) + 1$;

   A primeira instrução tem complexidade 2 e a segunda 3.

   Abordagem simplificada: contar como 1, não importando a complexidade da expressão associada, a não ser que a mesma envolva chamadas de procedimentos (ver regra 9 abaixo).

3. Seqüências de instruções

   A complexidade de uma seqüência de instruções corresponde à soma das complexidades das instruções individuais da seqüência. Por exemplo:

   $f \leftarrow 1.8 \cdot c + 32$;
   retorna f;

   Esta seqüência tem complexidade 4 na abordagem rigorosa ou 2 na simplificada.

4. Condicionais simples

   Abordagem rigorosa: instruções do tipo

   se *teste* **então** corpo;

   contam como 1 mais a complexidade de *teste* mais a complexidade de *corpo* (que pode ser, inclusive, uma seqüência de instruções). Por exemplo, a complexidade da instrução a seguir é 4:

   se $i > 0$ **então** $i \leftarrow i + 1$;

   Abordagem simplificada: contar como 1 mais a complexidade de *corpo*, o que dá resultado 2 para a instrução acima.

5. Condicionais com **senão**

   Abordagem rigorosa: instruções do tipo

   se *teste* **então** *ramo-então* **senão** *ramo-senão*;

   contam como 1 mais a complexidade de *teste* mais a maior das complexidades entre *ramo-então* e *ramo-senão*. Por exemplo,

   se $x < 0$ **então** $x \leftarrow 0$ **senão** $x \leftarrow x + 1$;

   tem complexidade 4, pois a complexidade do ramo mais caro ($x \leftarrow x + 1$;) é levada em conta.

   Abordagem simplificada: contar como 1 mais a complexidade do ramo de maior complexidade.

6. Laços limitados

   Abordagem rigorosa: instruções do tipo:

   para $i = 1$ **até** k **faça** corpo;

   são contadas da seguinte forma:

   - A variável de controle i é inicializada uma única vez no início do laço: complexidade 1.
   - A variável de controle é incrementada $k + 1$ vezes (uma para cada iteração e uma para sair do laço): complexidade $k + 1$.[6]

[6] Os mais detalhistas poderiam argumentar que são necessárias duas operações para incrementar o valor da variável i (como na instrução $i \leftarrow i + 1$;). Considerando, porém, a existência de instruções de baixo nível que permitem fazer o incremento de uma variável inteira, contaremos apenas 1 para esta operação.

- O teste de terminação, da mesma forma, é executado $k+1$ vezes e tem complexidade $k+1$.
- O corpo do laço é executado k vezes: complexidade kc_p, onde c_p é a complexidade de *corpo*.

Portanto, a complexidade total é $k(c_p+2)+3$. Por exemplo, o laço a seguir calcula a soma dos elementos de um vetor de inteiros A:

para *i = 1* **até** *A.comprimento* **faça** soma ← soma + A[i];

Considerando *A.comprimento* como k, e verificando que o corpo do laço tem complexidade 2, a complexidade total é $4k+3$.

Abordagem simplificada: complexidade $k(c_p+1)$: complexidade de *corpo* mais complexidade do controle do laço (considerada como 1), multiplicadas pelo número de iterações. Para o laço acima, a complexidade seria $2k$.

7. Laços não limitados

Abordagem rigorosa: a complexidade de laços de pré-teste do tipo:

enquanto *teste* **faça** corpo;

é $kc_p+(k+1)c_t$, onde c_p é a complexidade do corpo do laço (executado uma vez por iteração), c_t a complexidade do teste de terminação (executado uma vez por iteração e outra na saída do laço) e k o número de iterações. Para laços de pós-teste:

repita corpo **até** *teste*;

a complexidade é $k(c_p+c_t)$, pois neste caso o corpo e o teste são executados o mesmo número de vezes. Para estes laços, como k não é conhecido de antemão, a nossa estratégia será adotar uma abordagem pessimista e considerar que o laço será executado o maior número possível de vezes. Por exemplo, o trecho de código a seguir percorre os elementos de um vetor de inteiros até encontrar o valor 5 ou o final do vetor:

enquanto $i \leqslant$ *A.comprimento* **e** $A[i] \neq 5$ **faça** $i \leftarrow i+1$;

Não sabemos se o vetor contém o valor 5 nem em que posição ele está. Assim assumimos que todos os elementos serão examinados. Chamando o tamanho de A de k, tem-se $c_p = 2$ e $c_t = 3$, logo a complexidade total é $5k+3$.

Abordagem simplificada: para ambos os casos (pré e pós-teste), considera-se complexidade 1 para o teste mais a complexidade do corpo multiplicadas pelo número de iterações: $k(c_p+1)$. Para determinar k, continuamos com a abordagem pessimista. A complexidade do laço acima é calculada como $2k$.

8. Laços aninhados

Para laços aninhados, começar a análise pelos laços mais internos e considerá-los como mais uma instrução no corpo nos laços mais externos. Exemplo:

para *i = 1* **até** k **faça** **para** *j = 1* **até** l **faça** $m \leftarrow m+1$;

Considerando primeiramente o laço interno, podemos ver que sua complexidade é $c_i = l(2+2)+3 = 4l+3$ (pois a complexidade da instrução $m \leftarrow m+1$ é 2). Levando este resultado para o cálculo da complexidade do laço externo vem $c_e = k[(4l+3)+2]+3 = 4kl+5k+3$, onde foi adotada a abordagem rigorosa. Para a abordagem simplificada, os cálculos seriam: $c_e = k(c_i+1) = k[l(1+1)+1] = 2kl+k$.

9. Funções, procedimentos e chamadas

As funções e procedimentos devem ser considerados como módulos separados e ter sua complexidade calculada seguindo as regras acima. É preferível começar a análise pelos procedimentos mais internos, ou seja, aqueles que não possuem chamadas para outros. A complexidade das chamadas de procedimentos é calculada como a seguir:

Instrução	Abordagem rigorosa	Abordagem simplificada
Expressão	1 para cada operador.	não considerar.
Comando simples	1 mais a complexidade da expressão associada.	1
Seqüência de comandos	Soma das complexidades dos comandos individuais.	Idem abordagem rigorosa.
Condicional simples	1 mais a complexidade do teste da condição mais a complexidade das instruções no corpo do condicional.	1 mais a complexidade das instruções no corpo do condicional.
Condicional	1 mais a complexidade do teste da condição mais a complexidade das instruções no ramo de maior complexidade.	1 mais a complexidade das instruções no ramo de maior complexidade.
Laço limitado	$k(c_p + 2) + 3$, onde k é o número de iterações e c_p a complexidade do corpo do laço.	$k(c_p + 1)$
Laço não limitado	Pré-teste: $kc_p + (k + 1)c_t$ Pós-teste: $k(c_p + c_t)$ k é o número de iterações, c_p a complexidade do corpo do laço e c_t o complexidade do teste terminação. Considerar que o número de iterações k é o maior possível.	$k(c_p + 1)$. Considerar que o número de iterações k é o maior possível.
Laços aninhados	Começar pelos laços mais internos e considerá-los como instruções comuns no corpo dos laços mais externos.	Idem abordagem rigorosa.
Chamadas de procedimentos e funções	1 mais a complexidade do procedimento ou função.	complexidade do procedimento ou função.

Tabela 4.1: Regras para contagem de instruções

Abordagem rigorosa: complexidade do procedimento chamado mais 1 para a chamada. Se os parâmetros passados envolverem algum tipo de expressão, a complexidade destas deve ser adicionada. Exemplo: a expressão abaixo tem complexidade igual a $c_p + 9$, onde c_p é a complexidade da função `Potencia`):

$$(t \neq -1) \texttt{ e } ((3 - t) \cdot (x/3)/(1 + t) > \texttt{Potencia(2,7)})$$

Abordagem simplificada: a complexidade da chamada é igual à do procedimento associado. Nesta abordagem, a complexidade da expressão acima seria igual a c_p.

A tabela 4.1 contém um resumo dos princípios acima. Vamos colocá-los em prática através da análise de um algoritmo simples.

Veja o algoritmo 4.1, que encontra o maior elemento de um vetor de inteiros. Ele trabalha da seguinte forma: inicialmente (linha 1), supõe-se que o maior elemento encontra-se na primeira posição do vetor. O laço das linhas 2–6 compara este elemento com os restantes, armazenando sempre o maior encontrado até então. Ao final, o maior elemento é retornado (linha 7)**To do** (5) .

Analizando este algoritmo (utilizando a abordagem rigorosa), vemos que ele é constituído, basicamente, por uma seqüência de comandos: a atribuição da linha 1, o laço das linhas 2–6 e a instrução de retorno da linha 7. Segundo a regra de cálculo para seqüências, a complexidade total será a soma das complexidades dos comandos individuais. Temos então:

$$C = C_a + C_l + C_r \tag{4.1}$$

onde:

Algoritmo 4.1: `Maximo(A)`

Entrada: Vetor de inteiros A.
Saída: Maior elemento de A.
Implementação: Programas A.12 e B.12.

```
1 maximo ← A[1];
2 para i = 2 até A.comprimento faça
3 |  se A[i] > maximo então
4 |  |  maximo ← A[i];
5 |  fim se
6 fim para
7 retorna maximo;
```

- C é a complexidade total do algoritmo;
- C_a é a complexidade do comando de atribuição;
- C_l é a complexidade do laço;
- C_r é a complexidade do comando de retorno.

O cálculo de C_a é simples: segundo a regra para atribuições, a complexidade é 1 mais a complexidade da expressão associada, que no caso é 0. Assim $C_a = 1$. Seguindo o mesmo raciocínio, é fácil ver que $C_r = 1$. Para calcular C_l, utilizamos a regra para laços limitados: $C_l = k(c_p + 2) + 3$, onde c_p é a complexidade do corpo do laço (linhas 3–5), e k é o número de iterações realizadas. Para um vetor de tamanho n, o número de iterações será $k = n - 1$ (porque o vetor é examinado de 2 até n). Assim, $C_l = (n - 1)(c_p + 2) + 3$. Levando estes resultados na equação 4.1 vem:

$$C = 1 + (n - 1)(c_p + 2) + 3 + 1 = (n - 1)(c_p + 2) + 5 \tag{4.2}$$

Para calcular c_p utilizamos a regra para condicionais simples: a complexidade é 1 mais a complexidade do teste "$A[i] >$ maximo" (que é 1) mais a complexidade da atribuição "maximo $\leftarrow A[i]$" (que também é 1). Assim, $c_p = 3$. Levando este valor em 4.2 temos $C = (n - 1)(3 + 2) + 5$ ou:

$$C = 5n. \tag{4.3}$$

Efetuando o mesmo cálculo, utilizando a abordagem simplificada, temos:

- Atribuição da linha 4: complexidade 1.
- Condicional simples das linhas 3–5: complexidade $1 + 1 = 2$.
- Laço limitado das linhas 2–6: complexidade $(n - 1)(2 + 1) = 3n - 3$.
- Atribuição da linha 1: complexidade 1.
- Instrução de retorno da linha 7: complexidade 1.
- Complexidade total: $C = 1 + (3n - 3) + 1$ ou:

$$C = 3n - 1. \tag{4.4}$$

Ou seja: a complexidade temporal do algoritmo 4.1, que encontra o maior elemento de um vetor de inteiros de tamanho n, é $5n$ ou, utilizando uma abordagem simplificada, $3n - 1$.

Antes de passarmos a outros exemplos, é necessário chamar a atenção para duas observações muito importantes. Em primeiro lugar, note que os princípios de contagem apresentados utilizam sempre uma abordagem pessimista. Ou seja: condicionais e laços optam sempre pelo pior caso, onde o corpo dos condicionais ou o ramo mais caro é sempre executado, o laço roda o maior número possível de vezes,

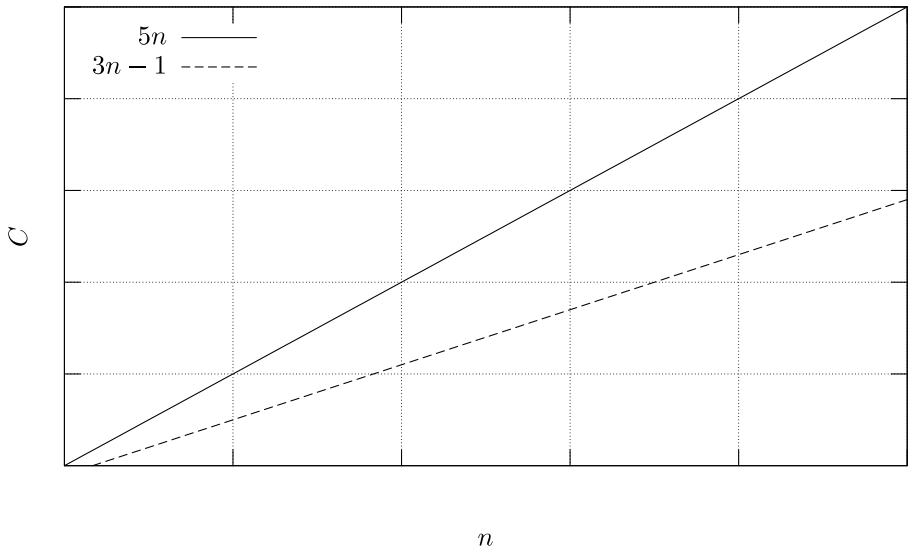

Figura 4.1: Complexidade do algoritmo 4.1

etc. Esta abordagem é denominada *análise de pior caso*. Existem também abordagens denominadas de *melhor caso* e *caso médio*, que serão estudadas no capítulo 6.

Em segundo lugar, repare que nossa análise forneceu dois resultados diferentes para a complexidade do algoritmo 4.1: um a partir da abordagem rigorosa ($C_r = 5n$) e outro a partir da abordagem simplificada ($C_s = 3n - 1$). Qual deles devemos utilizar? Obviamente, a abordagem rigorosa é mais precisa e, a princípio, preferível. Entretanto, iremos, pelo menos inicialmente, optar pela abordagem simplificada; na verdade, a partir da próxima seção, iremos adotar uma simplificação ainda maior.

O motivo é que nosso interesse, inicialmente, está não nos valores exatos, mas no comportamento da complexidade à medida que o tamanho n da entrada aumenta. A figura 4.1 mostra o gráfico das funções C_r e C_s em função de n. Repare que ambas as curvas têm a mesma "forma" (retas), indicando que as funções variam linearmente com o tamanho da entrada. Como $C_s \approx 3n$ (a constante desprezada torna-se cada vez menos significativa à medida que n cresce), ambas as funções nos dão o mesmo tipo de informação: se dobrarmos o tamanho da entrada, a complexidade dobra, se triplicarmos o tamanho da entrada, a complexidade triplica e, em geral, se multiplicarmos n por um determinado valor k, a complexidade será multiplicada pelo mesmo fator. Assim, podemos optar pela análise simplificada, uma vez que ela nos fornece a mesma informação que a análise mais rigorosa. No capítulo 5 todas estas idéias serão retomadas e justificadas com um grau maior de rigor matemático.

Sedgewick e Flajolet (1996, p. 2–3) procuram distingüir dois tipos de abordagens na análise de algoritmos:

1. O primeiro, que eles denominam *complexidade computacional*, coincide com a nossa abordagem, e tem como principal objetivo determinar o desempenho dos algoritmos no pior caso e descobrir algoritmos "ótimos" para um determinado problema[7].
2. O segundo tipo, chamado *análise de algoritmos*, procura determinar o desempenho no melhor caso, pior caso e caso médio, com o maior grau de precisão possível, de forma a permitir a previsão do desempenho real de um algoritmo em um computador particular. Isto pode envolver, inclusive, a determinação precisa do tempo requerido para a execução das instruções de baixo nível em um determinado sistema. O processo está esquematizado em Sedgewick e Flajolet (1996, p. 10–23).

Na verdade, estas duas abordagens podem ser vistas como complementares, com a complexidade computacional funcionando como um primeiro passo no sentido de desenvolver uma análise mais

[7] O conceito de algoritmo ótimo será desenvolvido no capítulo 10.

Algoritmo 4.2: `Pesquisa(A,x)`

Entrada: Vetor de inteiros A com elementos distintos e inteiro x.
Saída: Índice de x dentro de A, -1 se não encontrado.
Implementação: Programas A.13 e B.13.

```
1 para i = 1 até A.comprimento faça
2 |  se A[i] = x então
3 |  |  retorna i;
4 |  fim se
5 fim para
6 retorna −1;
```

acurada do algoritmo.

4.3 Alguns exemplos simples

Nesta seção, vamos realizar a análise de alguns algoritmos de modo a exemplificar o método de contagem de instruções. Entretanto, como mencionado no final da seção anterior, vamos adotar um procedimento simplificado e evitar a contagem linha a linha de todas as instruções do algoritmo. Seguindo um costume largamente adotado, vamos identificar, para cada algoritmo, a instrução mais significativa ou dominante, e contar apenas a execução dessa instrução. Por exemplo, na análise do algoritmo 4.1, poderíamos contar apenas as comparações da linha 3.

Nossos exemplos envolverão algoritmos alternativos para a solução de dois problemas: a pesquisa de um elemento dentro de um vetor e o cálculo da soma máxima das subseqüências de um vetor de inteiros.

Pesquisa de elementos em vetor

Suponha que temos um vetor de inteiros com elementos não repetidos e queremos determinar se um número dado encontra-se dentre os elementos deste vetor. A forma mais simples de resolver este problema é comparar o número procurado com cada um dos elementos do vetor, partindo do primeiro e indo até o último. Se o número for encontrado, a busca é interrompida. Este método denomina-se *pesquisa linear* ou *seqüencial* e está mostrado no algoritmo 4.2**To do** (6) . Se o número procurado fizer parte do vetor, o algoritmo retorna o índice correspondente; caso contrário, retorna -1.

A análise deste algoritmo é simples. Elegemos como instrução mais significativa a comparação do número procurado com os elementos do vetor (ou seja, queremos saber quantas comparações o algoritmo realiza). No algoritmo 4.2, este tipo de instrução ocorre apenas uma vez na linha 2. Considerando o pior caso, o corpo do laço das linhas 1–5 será executado n vezes (onde n é o tamanho do vetor), correspondendo à situação em que o elemento procurado não se encontra no vetor, forçando a realização do maior número possível de comparações. Assim, chamando de C_s a complexidade da pesquisa sequencial, temos:

$$C_s = n. \tag{4.5}$$

A pesquisa seqüêncial funciona independentemente da organização interna dos elementos do vetor. Suponha, agora, que sabemos que estes elementos estão ordenados de forma crescente, ou seja:

$$A_1 < A_2 < \cdots < A_n.$$

Podemos então adotar uma estratégia diferente. Inicialmente, comparamos o número procurado x com o elemento central A_m do vetor ($m = \lfloor (n+1)/2 \rfloor$). Se $x = A_m$, a busca termina. Caso contrário, devemos determinar em que região do vetor a pesquisa deve prosseguir. Se $x < A_m$, então certamente x não pode ser nenhum dos elementos A_{m+1}, A_{m+2}, $\cdots$, A_n. Assim, os elementos a pesquisar

Algoritmo 4.3: `Binaria(`A`,`x`)`

Entrada: Vetor de inteiros A com elementos distintos e inteiro x.
Saída: Índice de x dentro de A, -1 se não encontrado.
Implementação: Programas A.14 e B.14.

```
1  e ← 1; d ← A.comprimento;
2  enquanto e <= d faça
3  |  m ← ⌊(e + d)/2⌋;
4  |  se x = A[m] então
5  |  |  retorna m;
6  |  senão se x < A[m] então
7  |  |  d ← m − 1;
8  |  senão
9  |  |  e ← m + 1;
10 |  fim se
11 fim enqto
12 retorna −1;
```

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	$e = 1, d = 15, m = 8$
1	**2**	**3**	**4**	**5**	**6**	**7**	8	9	10	11	12	13	14	15	$e = 1, d = 7, m = 4$
1	2	3	4	**5**	**6**	**7**	8	9	10	11	12	13	14	15	$e = 5, d = 7, m = 8$
1	2	3	4	**5**	6	7	8	9	10	11	12	13	14	15	$e = 5, d = 5, m = 5$

Figura 4.2: Exemplo de pesquisa binária

reduziram-se à faixa de índices $1 \rightarrow m - 1$. Escolhemos o elemento central desta nova faixa e prosseguimos da mesma forma. Analogamente, se $x > A_m$, a nova faixa para busca é $m + 1 \rightarrow n$. Esta estratégia é denominada *pesquisa binária* e esta mostrada no algoritmo 4.3**To do** (7) . As variáveis e e d, representam, respectivamente, os limites esquerdo e direito da faixa de índices dos elementos a pesquisar e são inicializadas, na linha 1, para incluir todos os elementos do vetor. O índice do elemento central é calculado na linha 3. Se o número procurado não for encontrado pela comparação da linha 4, esta faixa será restringida por uma das instruções das linhas 7 ou 9, tal como foi descrito. A condição $d < e$ indica que os limites se cruzaram e, portanto, o número pesquisado não se encontra no vetor.

Por exemplo, seja um vetor com conteúdo:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15

e suponha que queremos procurar pelo número 5. A figura 4.2 mostra os passos da pesquisa, onde os números destacados indicam a faixa de elementos que ainda fazem parte da busca. O número 5 é encontrado na quinta posição do vetor.

Para calcular a complexidade temporal C_b deste algoritmo, primeiramente notamos que, a cada iteração do laço das linhas 2–11, são feitas duas comparações (linhas 4 e 6). Então,

$$C_b = 2c \tag{4.6}$$

onde c é o número de iterações do laço. Adotamos novamente a abordagem pessimista e assumimos a situação onde o número de iterações será o maior possível, correspondendo à procura de um número que não se encontra no vetor.

Em seguida, notamos que, a cada iteração, a quantidade de elementos a pesquisar é reduzida aproximadamente à metade. Por exemplo, para um vetor de tamanho $n = 100$, temos inicialmente $e = 1$ e $d = 100$, e o índice do elemento selecionado para comparação é $m = 50$. Assim, dependendo do resultado do teste da linha 6, restringiríamos nossa busca à faixa $e = 1$ a $d = 49$ (49 elementos) ou $e = 51$ a $d = 100$ (50 elementos), e assim sucessivamente.

Para simplificar os cálculos, vamos assumir que o tamanho n do vetor será sempre da forma:

$$n = 2^k - 1 \tag{4.7}$$

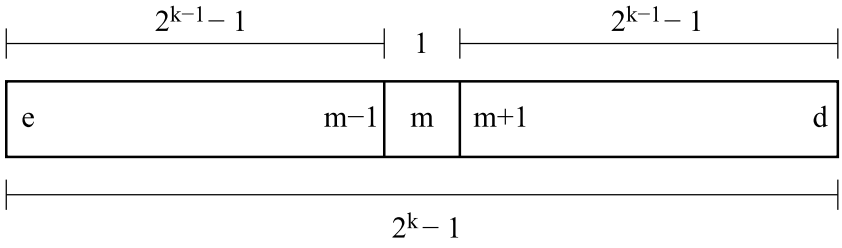

Figura 4.3: Pesquisa binária: comparação com elemento central do vetor

Iteração	Elementos restantes
1	$2^{k-1}-1$
2	$2^{k-2}-1$
3	$2^{k-3}-1$
...	...
i	$2^{k-i}-1$
...	...
k	$2^{k-k}-1=0$

Tabela 4.2: Pesquisa binária: número de elementos restantes após as iterações

onde $k \in \mathbb{N}^*$. Ou seja, nossos vetores terão tamanhos restritos aos valores 1, 3, 7, 15, 31, 63, 127, 255, ... (para k valendo 1, 2, 3, 4, 5, 6, 7, 8, ...). Se o tamanho real do vetor não está nesta lista, "aproximamos" este tamanho para o próximo elemento da lista que seja maior que ele. Por exemplo, se $n = 100$, iremos trabalhar assumindo que $n = 127$.

A vantagem de trabalhar com n da forma $2^k - 1$ pode ser vista na figura 4.3. Após a eliminação do elemento central de nossa pesquisa, ficaremos com $2^k - 2$ elementos divididos em duas faixas de tamanho idêntico $(2^k - 2)/2 = 2^{k-1} - 1$. Outra divisão levaria a $2^{k-2} - 1$, e assim sucessivamente. No capítulo 5 demostraremos que este tipo de aproximação é legítimo, e não compromete o resultado de nossa análise.

Desta forma, após a iteração número 1, restarão $2^{k-1} - 1$ elementos para análise, após a iteração número 2, $2^{k-2} - 1$ elementos e, em geral, após a iteração i, $2^{k-i} - 1$ elementos; estes fatos estão resumidos na tabela 4.2. Após k iterações, teremos eliminado todos os elementos e a execução se encerra.

Portanto, precisaremos de k iterações para fazer a pesquisa completa e, a partir da equação 4.6, temos:

$$C_b = 2k.$$

Precisamos colocar este último resultado em função do tamanho n do vetor. Mas, da equação 4.7, temos[8] $n = 2^k - 1 \therefore k = \lg(n + 1)$. Finalmente:

$$C_b = 2\lg(n + 1). \tag{4.8}$$

É interessante comparar os dois resultados obtidos (equações 4.5 e 4.8). Na tabela 4.3, temos alguns valores de C_s e C_b para diversos valores de n[9]. Repare que o crescimento de C_s é muito mais acentuado que o de C_b. Para pequenos valores (n inferior a 6) temos $C_s < C_b$. À medida que n cresce, entretanto, C_s torna-se muito maior, ou seja, o algoritmo seqüencial faz muito mais comparações para encontrar o elemento procurado. Assim, se temos um vetor previamente ordenado, a escolha do algoritmo de pesquisa binária torna-se óbvia. Examinando as duas funções de complexidade podemos compreender a causa disto:

- $C_s = n$ nos diz que, se dobrarmos o número de elementos de nosso vetor, o número de comparações também irá dobrar.

[8] A notação $\lg x$ indica logaritmo base 2: $\lg x = \log_2 x$. As notações matemáticas mais utilizadas estão na página xv.
[9] Os valores foram escolhidos para fornecer resultados inteiros

n	1	3	15	127	1.023	16.383	131.071	1.048.575
C_s	1	3	15	127	1.023	16.383	131.071	1.048.575
C_b	2	4	8	14	20	28	34	40

Tabela 4.3: Funções de complexidade C_s e C_b

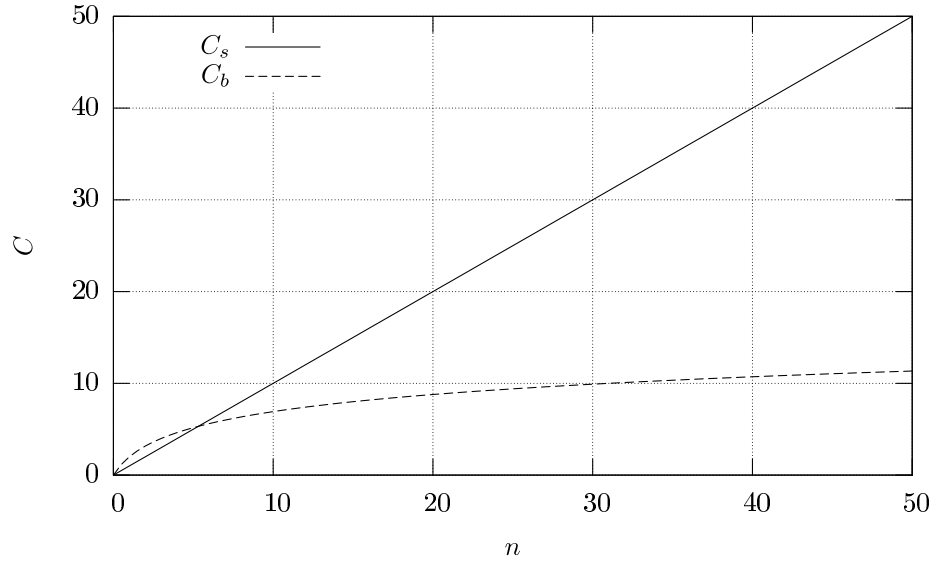

Figura 4.4: Complexidade das pesquisas seqüencial e binária

- $C_b = 2\lg(n+1)$ nos diz que, se dobrarmos o tamanho do vetor, precisaremos de apenas mais duas comparações em nossa pesquisa (ou seja, mais uma iteração do laço das linhas 2–11).

A figura 4.4 mostra representações gráficas dessas duas funções para que a diferença entre as taxas de crescimento se torne bem clara.

Soma máxima das subseqüências de um vetor

Dado um vetor A_1, A_2, ..., A_n, uma subseqüência é qualquer subvetor contendo elementos consecutivos A_i, A_{i+1}, ..., A_j, com $1 \leqslant i \leqslant j \leqslant n$. Se o vetor contém números inteiros, podemos falar da soma S_{ij} da subseqüência como o somatório de todos os seus elementos:

$$S_{ij} = \sum_{k=i}^{j} A_k.$$

Nosso problema consiste em, dado um vetor de inteiros, encontrar o maior valor de S_{ij}, chamado de soma máxima das subseqüências do vetor, definida como:

$$S_{\max} = \begin{cases} 0 & \text{se todos os elementos são negativos,} \\ \max\{S_{ij} \mid 1 \leqslant i \leqslant j \leqslant n\} & \text{caso contrário.} \end{cases}$$

Reparar que, de acordo com a definição, $S_{\max} \geqslant 0$. Alguns exemplos:

1. Vetor: 4, -3, 5, -2, -1, 2, 6, -2. $S_{\max} = 11$ (subseqüência 4, -3, 5, -2, -1, 2, 6).
2. Vetor: -1, 12, -5, 13, -4, 1. $S_{\max} = 20$ (subseqüência 12, -5, 13).
3. Vetor: 1, -2, 5, -2, -2, 7. $S_{\max} = 8$ (subseqüência 5, -2, -2, 7).

Algoritmo 4.4: `SubSeqMax1(`A`)`

Entrada: Vetor de inteiros A.
Saída: Soma máxima das subseqüências de A.
Implementação: Programas A.15 e B.15.

```
1  maximo ← 0;
2  para i = 1 até A.comprimento faça
3  |  para j = i até A.comprimento faça
4  |  |  soma ← 0;
5  |  |  para k = i até j faça
6  |  |  |  soma ← soma + A[k];
7  |  |  fim para
8  |  |  se soma > maximo então
9  |  |  |  maximo ← soma;
10 |  |  fim se
11 |  fim para
12 fim para
13 retorna maximo;
```

4. Vetor: -1, -2, -3, -4, -5, -6, -7. $S_{\max} = 0$ (subseqüência vazia).

A abordagem mais direta para resolver este problema consiste em examinar todas as possíveis subseqüências do vetor, calcular a soma de cada uma e ficar com a maior. Esta estratégia de força bruta está mostrada no algoritmo 4.4**To do** (8) . As variáveis i e j indicam, respectivamente, o início e o final de uma subseqüência, e os laços das linhas 2–12 e 3–11 percorrem todas as subseqüências existentes. Para cada uma delas, o laço das linhas 5–7 faz o cálculo da soma, que é comparada com a soma máxima nas linhas 8–10 e trocada com ela se necessário.

Para fazer a análise, vamos escolher a instrução dominante do algoritmo. Como estamos calculando somas de subseqüências, esta instrução é a adição de elementos do vetor, que ocorre apenas uma vez na linha 6. Repare que o laço mais externo (linhas 2–12) é executado n vezes (n é o tamanho do vetor) quando i varia de 1 até n. Em cada iteração, o laço intermediário (linhas 3–11) é completamente executado, com j variando de i até n. Por fim, a cada iteração do laço intermediário, o laço interno (linhas 5–7) é completamente executado, com k variando de i até j. Portanto, o número C_1 de execuções da linha 6 é:

$$C_1 = \sum_{i=1}^{n} \sum_{j=i}^{n} \sum_{k=i}^{j} 1 = \frac{n(n+1)(n+2)}{6} = \frac{n^3 + 3n^2 + 2n}{6} \tag{4.9}$$

O desenvolvimento completo dos somatórios da equação 4.9 é deixado como exercício para o leitor. Repare que o termo mais significativo desta equação é cúbico, ou seja, a complexidade varia (aproximadamente) com o cubo do tamanho da entrada, o que não é muito interessante.

Podemos obter uma melhoria observando que o laço das linhas 5–7 não é realmente necessário. A função deste laço é calcular a soma da subseqüência cujos limites (índices i e j) foram determinados pelo laço das linhas 3–11. Entretanto, é possível realizar simultaneamente (no mesmo laço) tanto a determinação dos limites quanto o cálculo, tal como mostrado no algoritmo 4.5**To do** (9) (verificar as linhas 3–9).

Efetuando a análise (de forma análoga ao que foi feito para o algoritmo 4.4), vemos que o número de vezes C_2 que a adição da linha 5 é executada é:

$$C_2 = \sum_{i=1}^{n} \sum_{j=i}^{n} 1 = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} \tag{4.10}$$

A parcela mais significativa da equação 4.10 é quadrática, de modo que o algoritmo 4.5 é mais eficiente que o algoritmo 4.4. Os detalhes do desenvolvimento da equação 4.10 são deixados como exercício.

Algoritmo 4.5: `SubSeqMax2(`A`)`

Entrada: Vetor de inteiros A.
Saída: Soma máxima das subseqüências de A.
Implementação: Programas A.16 e B.16.

```
1  maximo ← 0;
2  para i = 1 até A.comprimento faça
3  |  soma ← 0;
4  |  para j = i até A.comprimento faça
5  |  |  soma ← soma + A[j];
6  |  |  se soma > maximo então
7  |  |  |  maximo ← soma;
8  |  |  fim se
9  |  fim para
10 fim para
11 retorna maximo;
```

Algoritmo 4.6: `SubSeqMax3(`A`)`

Entrada: Vetor de inteiros A.
Saída: Soma máxima das subseqüências de A.
Implementação: Programas A.17 e B.17.

```
1  soma ← 0;
2  maximo ← 0;
3  para i = 1 até A.comprimento faça
4  |  soma ← soma + A[i];
5  |  se soma < 0 então
6  |  |  soma ← 0;
7  |  fim se
8  |  se soma > maximo então
9  |  |  maximo ← soma;
10 |  fim se
11 fim para
12 retorna maximo;
```

Temos dois algoritmos para resolver o problema da subseqüência de soma máxima de um vetor, com complexidades quadrática e cúbica. Podemos fazer ainda melhor: o algoritmo 4.6 **To do** (10) encontra o valor procurado fazendo apenas uma passagem pelos elementos do vetor (laço das linhas 3–11). Ele funciona da seguinte forma: considere uma determinada execução do corpo do laço, quando será feito o tratamento do elemento A_i. Logo antes da execução da linha 4, a variável *soma* armazena a soma de uma subseqüência que termina no elemento A_{i-1}. O elemento A_i é então adicionado (linha 4), fornecendo novo valor da soma da subseqüência. Se este valor se tornar negativo, *soma* recebe 0, indicando que a subseqüência atual não contem a soma máxima, e que o cálculo de uma nova subseqüência deve começar no elemento seguinte (linhas 5–7). De qualquer forma, o valor da maior subseqüência encontrada até então é sempre armazenado na variável *maximo* pelo teste das linhas 8–10.

A análise da complexidade C_3 é simples: uma soma por iteração, para um total de n iterações (onde n é o tamanho do vetor) leva a:

$$C_3 = n \tag{4.11}$$

Ou seja, temos um algoritmo de complexidade linear, cuja eficiência é superior à dos dois anteriores. A figura 4.5 mostra um gráfico que compara as complexidades calculadas nas equações 4.9, 4.10 e 4.11, mostrando que a taxa de crescimento da complexidade do algoritmo 4.6 é bem inferior à dos demais.

Os dois exemplos apresentados tiveram como objetivo demostrar a existência de diversos algorit-

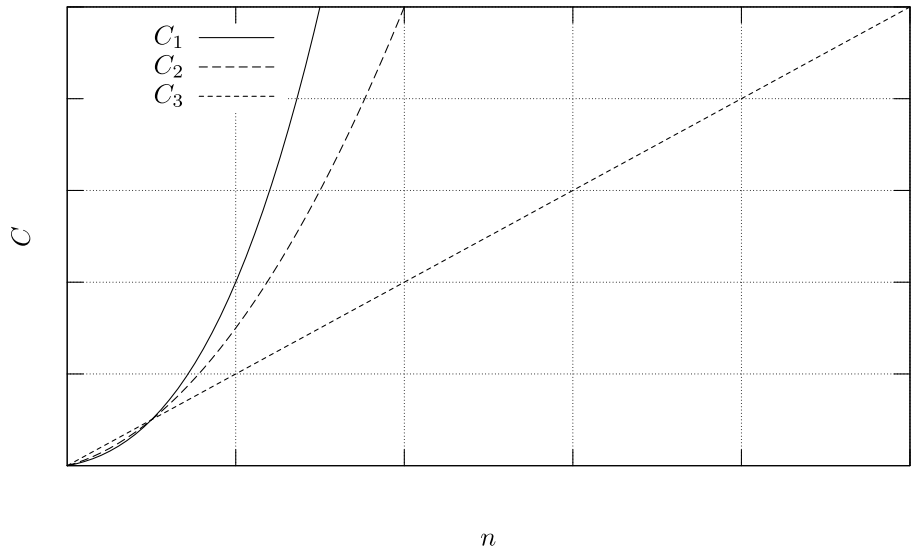

Figura 4.5: Complexidade dos algoritmos de maior subseqüência

mos distintos para a resolução do mesmo problema, por vezes com diferenças de desempenho bastante significativas, e como a análise de complexidade nos ajuda a optar pelo mais eficiente. Na seção seguinte, começaremos a aplicar este tipo de análise na discussão de um problema clássico: a ordenação de uma coleção de elementos.

4.4 Ordenação pelo método de seleção

Ordenar é rearranjar os objetos de uma coleção de modo que sua ordem obedeça a um determinado critério. Em geral, falamos de ordenar elementos de forma ascendente ou descendente. Em computação, o principal objetivo de ter uma coleção ordenada é conseguir localizar mais facilmente um elemento específico. Por exemplo, seria muito difícil encontrar o número telefônico de uma pessoa em um catálogo se os nomes não estivessem em ordem alfabética. Na seção 4.3, vimos também como a pesquisa binária nos permite encontrar um número dentro de um vetor de forma muito mais eficiente que uma busca seqüencial, desde que os elementos estejam previamente ordenados.

Em geral, os métodos de ordenação são classificados em *internos* ou *externos*. Na ordenação interna, considera-se que todos os elementos da coleção podem ser diretamente acessados, de forma rápida, a qualquer momento. Em termos de implementação, a coleção a ser ordenada tem um tamanho tal que permite que todos os objetos sejam trazidos para a memória principal ou "interna" do computador (daí o nome), possivelmente na forma de um vetor.

Na ordenação externa, os elementos não são todos diretamente acessíveis em um determinado instante, e devem ser tratados seqüencialmente ou em grandes grupos. Talvez o tamanho da coleção não permita que todos elementos sejam trazidos para a memória principal, ficando a maioria deles armazenados em dispositivos secundários tais como discos ou fitas magnéticas.

Wirth (1989, p. 50–51) faz a seguinte analogia: para ordenar as cartas de um baralho, podemos distribuir todas elas sobre a superfície de uma mesa. Todas são visíveis e podem ser diretamente manipuladas e movimentadas conforme desejado. Entretanto, se temos dez mil fichas para colocar em ordem, não podemos visualizar e manipular todas ao mesmo tempo. Talvez tenhamos que criar diversas pilhas contendo um número determinado de fichas e, em um determinado momento, só seremos capazes de acessar a ficha no topo de cada pilha. No primeiro caso, temos a ordenação interna e, no segundo, externa. Neste trabalho não iremos discutir os métodos de ordenação externa. Para maiores detalhes, consulte Knuth (1973b, p. 247–378), Wirth (1989, p. 76–107) e Ziviani (2004, p. 125–143).

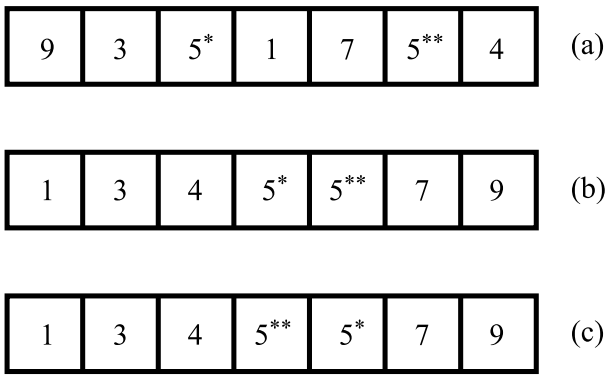

Figura 4.6: Estabilidade dos métodos de ordenação

Normalmente, os elementos a ordenar são registros que contêm diversos campos. Por exemplo, um registro que representa uma pessoa pode conter nome, endereço, número de documento, idade, estado civil, etc. Entretanto, a ordenação pode ser feita levando em conta apenas um desses campos, ou um pequeno conjunto deles. Talvez quiséssemos ordenar nossa coleção de pessoas por ordem de número de documento, ou por ordem alfabética de nomes. O campo que dirige a ordenação é considerado a *chave* do registro. A chave pode ser de qualquer tipo para o qual exista uma regra de ordenação bem definida (ou seja, podemos dizer se um determinado item é menor, igual ou maior que um outro). Para números inteiros, a regra de ordenação é fornecida pelas relações "menor" ($<$), "igual" ($=$) e maior ($>$); para cadeias de caracteres, temos a ordem lexicográfica (ordem do dicionário), e assim por diante. Por simplicidade, nossos algoritmos serão apresentados supondo que a chave é um número inteiro. Além disso, os demais campos serão desprezados, uma vez que não influenciam no processo de ordenação, mas apenas acompanham a chave. Assim, em todos os exemplos, estaremos sempre preocupados em ordenar vetores de inteiros.

Algumas observações antes de passarmos ao detalhamento do primeiro algoritmo de ordenação:

1. Um método de ordenação é dito *estável* se a ordem relativa de registros de mesma chave é mantida após o processo. Por exemplo, seja o vetor da figura 4.6(a). Repare que há dois elementos de mesma chave (5); os asteriscos ao lado de cada um servem apenas para distingüí-los visualmente, ou seja: o elemento 5* aparece, no vetor original, antes do elemento 5**. Se o método é estável, ao final do processo teremos a situação da figura 4.6(b), com o elemento 5* ainda antes do 5**. Se o método não é estável, nada se pode afirmar, e é possivel encontrarmos a situação da figura 4.6(c). É sempre possível forçar a estabilidade de um método, por exemplo, agregando um pequeno índice a cada chave (ZIVIANI, 2004, p. 96).

2. Vamos nos concentrar nos métodos que fazem a ordenação *in situ*, ou seja, aqueles que não necessitam de grandes áreas extras de memória para realizar a ordenação. A menos de algumas poucas variáveis auxiliares, o próprio vetor a ordenar será utilizado como área de trabalho. A única exceção a esta regra será o método denominado *mergesort*, estudado no capítulo 7.

3. Não pretendemos realizar um estudo exaustivo de todos os métodos de ordenação existentes. Vamos nos concentrar apenas em analisar os mais importantes. Os leitores interessados em outros métodos devem consultar Knuth (1973b, p. 1–388), Wirth (1989, p. 50–109) e Ziviani (2004, p. 95–152).

Iniciaremos com um método denominado *ordenação por seleção*. O método tem este nome porque, em cada etapa, um dos elementos (o menor dos ainda não analisados) é selecionado e colocado na posição correta. Explicando melhor: seja um vetor de n posições:

1. No primeiro passo, as posições 1 a n são examinadas para determinar onde se encontra o menor elemento. Este elemento é trocado com aquele que se encontra na primeira posição do vetor.

Vetor	Posições a examinar	Posição do menor elemento
5 4 9 1 3 2 8 6 7	1 a 9	4
1 4 9 5 3 2 8 6 7	2 a 9	6
1 2 9 5 3 4 8 6 7	3 a 9	5
1 2 3 5 9 4 8 6 7	4 a 9	6
1 2 3 4 9 5 8 6 7	5 a 9	6
1 2 3 4 5 9 8 6 7	6 a 9	8
1 2 3 4 5 6 8 9 7	7 a 9	9
1 2 3 4 5 6 7 9 8	8 a 9	9
1 2 3 4 5 6 7 8 9	—	—

Figura 4.7: Ordenação de vetor por seleção

2. Em seguida, as posições 2 a n são examinadas e o menor elemento é encontrado e trocado com o que está na segunda posição.

3. O processo prossegue desta forma até que todos os $n-1$ menores elementos tenham sido selecionados e colocados em suas posições (quando então, obviamente, o maior elemento já estará na última posição).

Por exemplo, seja o vetor:

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

No primeiro passo, o menor elemento (1, na posição 4) é selecionado e trocado com o elemento na primeira posição, levando a:

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

Em seguida, o mesmo é feito com o segundo menor elemento (2, na posição 6):

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

O processo prossegue da forma descrita até que o vetor esteja completamente ordenado; veja todos os passos executados na figura 4.7, onde os elementos destacados já estão em suas posições corretas.

O algoritmo 4.7 implementa o processo descrito[To do (11)] . A cada iteração do laço principal (linhas 1–9), o i-ésimo menor elemento é descoberto e trocado com o elemento na posição i (linha 8). O laço das linhas 3–7 encarrega-se de descobrir o índice do elemento procurado, supondo inicialmente que ele se encontra no início da faixa examinada (linha 2) e depois comparando-o com os demais e fazendo as trocas necessárias (linhas 4–6).

A análise deste algoritmo é simples. Primeiramente, precisamos determinar a instrução dominante. Para os algoritmos de ordenação, em geral queremos saber quantas comparações entre elementos são executadas até que a coleção esteja completamente ordenada. Seja n o tamanho do vetor. No algoritmo de seleção, as comparações são feitas apenas na linha 4. Esta linha é executada uma vez a cada iteração do laço das linhas 3–7. Este, por sua vez, é completamente executado (com j variando de $i+1$ até n) a cada iteração do laço das linhas 1-9, que é executado $n-1$ vezes (com i variando de 1 a $n-1$). Assim, chamando de C_s a complexidade da ordenação por seleção, temos:

$$C_s = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}. \tag{4.12}$$

Os detalhes do desenvolvimento são mais uma vez deixados para o leitor. Repare que o número de comparações cresce com o quadrado do tamanho do vetor. Existem diversos algoritmos de ordenação

Algoritmo 4.7: `Selecao(A)`

Entrada: Vetor de inteiros A.
Saída: Vetor A com elementos reordenados de forma que $A[1] < A[2] < \cdots < A[A.\text{comprimento}]$.
Implementação: Programas A.18 e B.18.

```
1 para i = 1 até A.comprimento − 1 faça
2 |   menor ← i;
3 |   para j = i + 1 até A.comprimento faça
4 |   |   se A[j] < A[menor] então
5 |   |   |   menor ← j;
6 |   |   fim se
7 |   fim para
8 |   A[i] ↔ A[menor];
9 fim para
```

com esta característica e, em geral, eles são chamados de *métodos simples* de ordenação (ZIVIANI, 2004, p. 97). Outros algoritmos exibem um comportamento melhor, e são chamados de *métodos eficientes*. Veremos mais alguns métodos de ordenação, tanto simples quanto eficientes, nos capítulos seguintes.

Pode-se argumentar que nem sempre a comparação entre elementos é a instrução mais significativa de um algoritmo de ordenação. Por exemplo, se os registros da coleção forem grandes, movimentá-los (trocá-los de lugar na coleção) pode levar muito mais tempo do que compará-los. Podemos, então, analisar a complexidade temporal da ordenação considerando o número de movimentações realizadas. Para o algoritmo de seleção, as movimentações ocorrem apenas na linha 8, que é executada uma vez a cada iteração do laço principal. Lembrando que a troca de elementos necessita, na verdade, de 3 movimentações[10], temos que o número total M_s de movimentações realizadas é:

$$M_s = \sum_{i=1}^{n-1} 3 = 3(n-1). \tag{4.13}$$

Repare que, para o algoritmo de seleção, o número de movimentações varia linearmente que o tamanho do vetor. Este fato é altamente significativo e torna o método de seleção extremamente interessante quando estamos manipulando coleções com grandes registros.

Uma última observação em relação à ordenação por seleção: não é difícil de perceber que o método não é estável: registros de mesma chave podem não ser deixados na mesma posição relativa.

4.5 Exercícios

Em todos os exercícios a seguir, sempre que for pedida uma implementação, você deverá desenvolver um algoritmo e implementá-lo em sua linguagem de programação predileta[**To do** (12)] .

Ex. 1 — Implemente uma função que receba um número inteiro não negativo m e retorne seu fatorial $m!$. Implemente também um algoritmo que receba dois vetores de inteiros A e B de mesmo tamanho, contendo elementos no intervalo $[-10, 10]$. Para cada elemento A_i, proceda da seguinte forma: se $A_i \geqslant 0$, calcule seu fatorial e guarde o resultado no elemento B_i. Se $A_i < 0$, faça $B_i = 0$. Utilize a primeira função para fazer os cálculos necessários. Faça a análise do algoritmo utilizando as abordagens rigorosa e simplificada descritas na tabela 4.1.

Ex. 2 — Analise, usando as abordagens rigorosa e simplificada da tabela 4.1, o algoritmo para multiplicação de números naturais mostrado abaixo.

[10] A instrução $a \leftrightarrow b$ é, na verdade, uma abreviação para $aux \leftarrow a$; $a \leftarrow b$; $b \leftarrow aux$; .

Algoritmo: `Multiplica(`x`,`y`)`
Entrada: Inteiros positivos x e y.
Saída: $x \cdot y$.
$z \leftarrow 0$;
enquanto $y > 0$ **faça**
  se $y \bmod 2 = 1$ **então**
    $z \leftarrow z + x$;
  fim se
  $x \leftarrow 2x$;
  $y \leftarrow \lfloor y/2 \rfloor$;
fim enqto
retorna z;

Ex. 3 — Implemente os algoritmos de cálculo da soma máxima das subseqüências de um vetor (4.4, 4.5 e 4.6) em sua linguagem de programação predileta e compare experimentalmente os desempenhos. Utilize vetores com tamanhos diversos (com até 50.000 elementos) e com conteúdo aleatório. Faça um gráfico dos resultados encontrados e verifique se a variação da complexidade com o tamanho do vetor está de acordo com os resultados obtidos na seção 4.3. Usuários de sistemas tipo Unix podem utilizar o comando *time* (ver TIME(1)).

Ex. 4 — Complete o desenvolvimento das equações 4.9, 4.10 e 4.12.

Ex. 5 — Observe o algoritmo abaixo. Qual é o valor retornado? Qual é a complexidade de tempo?

Algoritmo: Exercício
Entrada: Inteiro positivo n.
$r \leftarrow 0$;
para $i = 1$ **até** n **faça**
  para $j = 1$ **até** i **faça**
    para $k = j$ **até** $i + j$ **faça**
      $r \leftarrow r + 1$;
    fim para
  fim para
fim para
retorna r;

Ex. 6 — Analise os dois algoritmos para o cálculo de x^k, $x \in \mathbb{R}$, $k \in \mathbb{N}$, mostrados abaixo. Qual deles você escolheria? Justifique sua resposta.

Algoritmo: `Potencia1(`x`,`k`)`
Entrada: Números $x \in \mathbb{R}$ e $k \in \mathbb{N}$.
Saída: x^k .
$z \leftarrow 1$;
enquanto $k > 0$ **faça**
  $z \leftarrow z \cdot x$;
  $k \leftarrow k - 1$;
fim enqto
retorna z;

Algoritmo: `Potencia2(`x`,`k`)`
Entrada: Números $x \in \mathbb{R}$ e $k \in \mathbb{N}$.
Saída: x^k .
$z \leftarrow 1$;
enquanto $k > 0$ **faça**
  se $k \bmod 2 = 1$ **então**
    $z \leftarrow z \cdot x$;
  fim se
  $k \leftarrow \lfloor k/2 \rfloor$;
  $x \leftarrow x^2$;
fim enqto
retorna z;

Ex. 7 — Analise o algoritmo de reconhecimento de padrões abaixo. O algoritmo funciona da seguinte forma: a entrada consiste nos vetores $S[1\,..\,n]$ e $P[0\,..\,m-1]$, com $1 \leqslant m \leqslant n$. O algoritmo localiza a primeira ocorrência do padrão P na seqüência S e retorna p se $S[p\,..\,p+m-1] = P$ ou $(n-m+1)$ se P não ocorre em S. Quantas comparações entre os elementos de P e S são feitas?

Algoritmo: `Reconhece(`P`,`S`,`n`,`m`)`
Entrada: Vetores P e S e inteiros positivos n e m.
Saída: Posição do primeiro caracter de P em S se encontrado; $(n-m+1)$ caso contrário.
$l \leftarrow 0$; reconhecido $\leftarrow$ **falso**;
enquanto $(l \leqslant n-m)$ **e** (**não** *reconhecido*) **faça**
  $l \leftarrow l+1$;
  $r \leftarrow 0$; reconhecido $\leftarrow$ **verdadeiro**;
  enquanto $r < m$ **e** *reconhecido* **faça**
    reconhecido $\leftarrow$ reconhecido **e** $P[r] = S[l+r]$;
    $r \leftarrow r+1$;
  fim enqto
fim enqto
retorna l;

Ex. 8 — Seja um polinômio $p(x) = \sum_{i=0}^{n} a_i x^i$ (ou seja: $p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$).

1. Implemente um algoritmo que receba um valor para x e faça a avaliação do polinômio. Os coeficientes a_i devem estar armazenados em um vetor de tamanho n. Construa o algoritmo de forma direta e faça sua análise. Leve em conta o número de operações aritméticas executadas.

2. Implemente um novo algoritmo que utilize a regra de Horner para avaliar o polinômio:

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n)\cdots)))$$

   Analise este algoritmo e compare-o com o do item anterior.

Ex. 9 — Implemente um algoritmo para encontrar o maior e o menor elementos de um vetor de inteiros usando menos que $3n/2$ comparações, onde n é o tamanho do vetor. *Sugestão*: divida os elementos do vetor em um grupo de candidatos a maior e outro de candidatos a menor.

Ex. 10 — Seja uma matriz quadrada A de dimensão n. Todos os elementos dessa matriz assumem apenas os valores 0 ou 1 e, em uma linha, todos os valores 1 vêm antes dos valores 0. Implemente um algoritmo para encontrar a linha que contém a maior quantidade de valores 1. A complexidade temporal de seu algoritmo deve ser uma função linear (e não quadrática) de n.

Ex. 11 — Seja uma matriz quadrada A de dimensão n. Todos os elementos dessa matriz assumem apenas os valores 0 ou 1 e, em uma linha, todos os valores 1 vêm antes dos valores 0. Além disso, o número de valores 1 na linha i é sempre maior ou igual ao encontrado na linha $i+1$. Implemente um algoritmo que retorne o número de valores 1 em A. A complexidade temporal de seu algoritmo deve ser uma função linear (e não quadrática) de n.

Ex. 12 — Implemente um algoritmo para encontrar o tamanho da maior subseqüência crescente em uma seqüencia de números. Por exemplo, a seqüência $\langle 1, 5, 3, 2, 4\rangle$ tem a maior subseqüência crescente com tamanho **3** ($\langle 1, 3, 4\rangle$ ou $\langle 1, 2, 4\rangle$). Repare que, neste exercício, os elementos da subseqüência não precisam ser adjacentes. Analise seu algoritmo. A complexidade da solução deve ser, no máximo, uma função quadrática do tamanho da entrada.

Ex. 13 — No final da seção 4.4, mencionamos que a ordenação por seleção não é estável. Justifique esta afirmativa.

Ex. 14 — Suponha que você tenha um vetor de inteiros cujos elementos podem assumir apenas dois valores: 0 ou 1. Implemente um algoritmo para fazer a ordenação (*in situ*) dos elementos desse vetor. Analise seu algoritmo. A complexidade temporal deve ser uma função linear do tamanho da entrada.

Ex. 15 — Dado um vetor de inteiros distintos $A[1..n]$, $n \geqslant 2$, precisa-se de um algoritmo para encontrar o segundo maior elemento do vetor.

1. Construa um algoritmo de forma direta e faça sua análise. Leve em consideração o número de comparações. Você encontrou uma complexidade temporal de $2n - 3$?

2. Construa outro algoritmo para o mesmo problema cuja complexidade (no pior caso) seja $n + \lceil \lg n \rceil - 2$. *Sugestão*: imagine um torneio com n competidores se enfrentando pelo sistema de eliminatória simples (não se esqueça da repescagem ...). Comece considerando o tamanho n como uma potência de 2 ($n = 2^k$, $k \in \mathbb{N}^*$) e depois pense no que aconteceria se n não tivesse esta restrição.