CAPÍTULO

8

ORDENAÇÃO E BUSCA

Serão apresentados neste capítulo os principais algoritmos para ordenar uma coleção de dados e para buscar um dado específico numa coleção de dados armazenada num vetor.

8.1 Fundamentos

Ordenação e *busca* são problemas recorrentes em programação. Frequentemente, em várias aplicações, precisamos ordenar coleções de dados ou buscar dados em coleções. Nesse capítulo, apresentamos os principais métodos de ordenação e busca em vetores e comentamos sobre a eficiência de cada um deles.

8.1.1 Análise de algoritmos

Quando há vários algoritmos que resolvem um mesmo problema, para escolher um deles, precisamos ter uma forma de compará-los. Essa comparação pode ser baseada em diferentes aspectos como, por exemplo:

- Clareza da lógica.
- Concisão da lógica.
- Consumo de tempo.
- Consumo de memória.

Embora todos esses aspectos sejam importantes, a *análise de algoritmos* tem como foco o *consumo de tempo*, ou seja, algoritmos mais rápidos

são considerados melhores. Porém, como um mesmo algoritmo pode ser codificado em diversas linguagens e ser executado em diferentes computadores, com diferentes sistemas operacionais, determinar o consumo de tempo real de um algoritmo é uma tarefa impossível. Então, para viabilizar a análise de um algoritmo, assumimos que: (*a*) apenas os *passos* mais importantes do algoritmo consomem tempo relevante, (*b*) cada passo é executado em *uma* única unidade de tempo, (*c*) em cada unidade de tempo, um *único* passo pode ser executado e (*d*) o consumo de tempo de um algoritmo deve ser expresso em função do *tamanho* de sua entrada.

Por exemplo, considere a função definida na Figura 8.1, que calcula a soma dos itens de um vetor de tamanho n. Nessa figura, para cada passo, está anotada a quantidade de vezes que ele é executado. A soma dessas quantidades, indicada por $T(n) = 3n + 4$, representa o consumo de tempo desse algoritmo, em função do tamanho de sua entrada, que é n.

```
int soma(int v[], int n) {                          // vezes
   int s = 0;                                       //     1
   int i = 0;                                       //     1
   while( i<n ) {                                   //   n+1
      s += v[i];                                    //     n
      i++;                                          //     n
   }
   return s;                                        //     1
}
```

Figura 8.1 | Função que calcula a soma dos itens de um vetor.

8.1.2 Notação O

Para entradas pequenas idênticas, a diferença entre os tempos consumidos por diferentes algoritmos costuma ser pequena. Por isso, o objetivo da análise de algoritmos não é calcular o tempo exato consumido por um algoritmo, mas sim descobrir a *ordem de grandeza* desse tempo, para entradas *grandes*. Essa ordem de grandeza, denominada *complexidade de tempo*, é indicada em notação O.

Formalmente, uma função $f(n)$ é $O(g(n))$ se e só se existem constantes positivas c e n_0, tais que $f(n) \leq c \cdot g(n)$, para todo $n \geq n_0$. Esse conceito é ilustrado na Figura 8.2. Intuitivamente, se $f(n)$ é $O(g(n))$, significa que $g(n)$ é um *limite superior* para $f(n)$ ou, então, que a função $f(n)$ não cresce mais rapidamente que a função $g(n)$, a menos de um fator constante c.

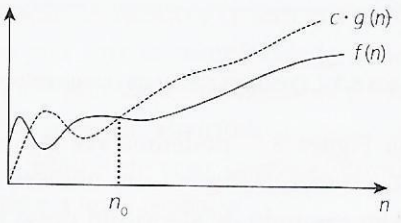


Figura 8.2 | A função $f(n)$ é $O(g(n))$.

Por exemplo, a função $T(n) = 3n + 4$ é $O(n)$, pois para $c = 4$ e $n_0 = 4$, temos $3n + 4 \leqslant 4n$, para todo $n \geqslant 4$, como exemplificado na Figura 8.3.

$n!$	$3n + 4$	$4n$
1	7	4
2	10	8
3	13	12
4	16	16
5	19	20
6	22	24
7	25	28
8	28	32
9	31	36

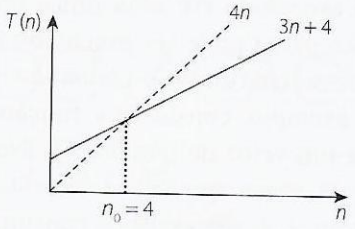


Figura 8.3 | Crescimento das funções $3n + 4$ e $4n$.

Usando a notação O, podemos separar os algoritmos em infinitas *classes de complexidade*, sendo que as principais classes são as seguintes:

- **Constante:** $O(1)$.
- **Logarítmica:** $O(\lg n)$.
- **Linear:** $O(n)$.
- **Linear-logarítmica:** $O(n \lg n)$.
- **Quadrática:** $O(n^2)$.
- **Cúbica:** $O(n^3)$.
- **Exponencial:** $O(2^n)$.

A relação entre essas classes de complexidade é ilustrada na Figura 8.4.

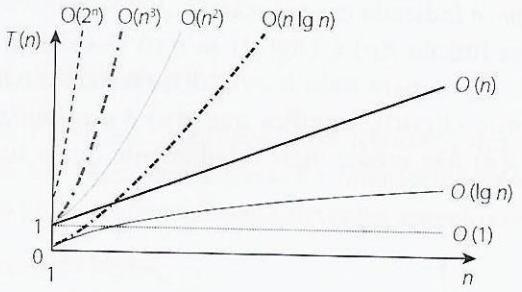


Figura 8.4 | Principais classes de complexidade.

Analisando as curvas na Figura 8.4, podemos ver que algoritmos da classe $O(1)$ consomem tempo *constante*, ou seja, executam um número fixo de passos, para qualquer tamanho de entrada. Um exemplo de algoritmo dessa classe é um algoritmo que

devolve o menor item de um vetor ordenado. A classe $O(1)$ é a classe dos algoritmos mais eficientes que podem ser criados. Por outro lado, algoritmos da classe $O(2^n)$ consomem tempo *exponencial*, ou seja, se o tamanho da entrada de um algoritmo exponencial aumenta uma unidade, seu consumo de tempo é *multiplicado* por uma constante maior que 1. Um exemplo de algoritmo dessa classe é aquele que resolve o problema das Torres de Hanói. A classe $O(2^n)$ é uma das classes de algoritmos menos eficientes que podem ser criados.

8.1.3 Análise de pior caso

Em muitos casos, o consumo de tempo de um algoritmo depende não somente do tamanho de sua entrada, mas também da configuração particular dessa entrada. Nesses casos, a análise de complexidade do algoritmo deve ser baseada na configuração de entrada que o faz consumir o maior tempo possível. Esse tipo de análise, chamada *análise de pior caso*, determina um limite superior para o consumo de tempo de um algoritmo, para *qualquer* configuração de sua entrada.

Por exemplo, considere a função definida na Figura 8.5, que verifica se um vetor está em ordem crescente.

```
int crescente(int v[], int n) {                    //  vezes
   int i = 0;                                      //  =    1
   while( i<n-1 ) {                                //  <=   n
      if( v[i] > v[i+1] )                          //  <=   n-1
         return 0;                                 //  <=   1
      i++;                                         //  <=   n-1
   }
   return 1;                                       //  <=   1
}
```

Figura 8.5 | Função que verifica se um vetor está em ordem crescente.

Claramente, se essa função recebe como entrada um vetor cujo primeiro par de itens está fora de ordem, independentemente do tamanho desse vetor, após a primeira comparação feita pelo `if`, a função devolve resposta `0`. Esse é o *melhor caso* possível, pois nele o número de passos executados é *constante* (isto é, 4 passos). Então, no *melhor caso*, a complexidade de tempo dessa função é $O(1)$.

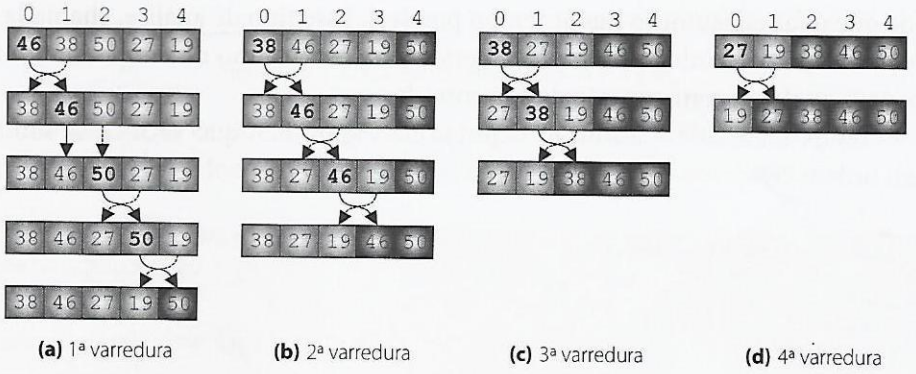
O problema da análise de melhor caso é que ela não revela o limite superior de consumo de tempo do algoritmo. Particularmente, no pior caso, que ocorre quando a entrada está ordenada, a função `crescente()` tem que executar $3n$ passos para devolver a resposta `1`. Assim, no *pior caso*, a complexidade de tempo dessa função é $O(n)$. Além das análises de melhor e pior casos, há também uma análise de *caso médio*, que revela o comportamento *típico* de um algoritmo para a maioria das configurações de entrada possíveis; porém, a análise de caso médio costuma ser mais complicada. Na prática, a análise de pior caso é a mais usada.

8.2 Métodos de ordenação

Dado um vetor v contendo n itens em ordem aleatória, um *método de ordenação* deve reorganizar os itens de v, de modo que tenhamos $v_0 \leqslant v_1 \leqslant v_2 \leqslant ... \leqslant v_{n-1}$. Por exemplo, após a ordenação de $v = \{4, 5, 3, 1, 4, 2\}$, devemos ter $v = \{1, 2, 3, 4, 4, 5\}$.

8.2.1 Ordenação por trocas

O modo mais simples de ordenar um vetor v com n itens é comparar itens adjacentes em v e, se eles estiverem fora de ordem, trocá-los de posição. Assim, após a primeira varredura de v, seu maior item estará na posição $n - 1$. A Figura 8.6 exemplifica o uso dessa estratégia para ordenar o vetor $v = \{46, 38, 50, 27, 19\}$.



(a) 1ª varredura **(b)** 2ª varredura **(c)** 3ª varredura **(d)** 4ª varredura

Figura 8.6 | Funcionamento da ordenação por troca.

De forma geral, após a i-ésima varredura de v, o item na posição $n - i$ estará corretamente posicionado. Como cada varredura desloca um item para sua posição definitiva em v, são necessárias $n - 1$ varreduras para ordenar os n itens. A *ordenação por trocas*, também chamada *bubble sort*, é implementada na Figura 8.7.

```
#define troca(a,b) { int x=a; a=b; b=x; }

void bubble_sort(int v[], int n) {
   for(int i=1; i<=n-1; i++)
      for(int j=0; j<n-i; j++)
         if( v[j]>v[j+1] )
            troca(v[j],v[j+1]);
}
```

Figura 8.7 | Função para ordenação por trocas.

A ordenação por trocas faz $n - 1$ varreduras no vetor e, na i-ésima varredura, ela faz $n - i$ comparações (que, no pior caso, resultam em $n - i$ trocas). Ou seja, na primeira varredura, ela faz $n - 1$ comparações; na segunda varredura, ela faz $n - 2$ comparações;

na terceira, $n - 3$ comparações e assim por diante. Portanto, o total de comparações feitas é $(n - 1) + (n - 2) + (n - 3) + ... + 2 + 1 = (n^2 - n)/2$. Logo, a complexidade de tempo da ordenação por trocas, no pior caso, é $O(n^2)$.

8.2.2 Ordenação por seleção

Para ordenar um vetor v com n itens, a *ordenação por seleção* seleciona um índice $i \in [0..n-1]$, tal que v_i seja máximo em v, e troca os itens v_i e v_{n-1}. Após essa troca, o item na posição $n - 1$ de v estará corretamente posicionado. Então, esse passo é repetido, supondo que o vetor tem um item a menos. A cada passo, um novo item é movido para sua posição definitiva em v. Assim, após $n - 1$ passos, o vetor v estará completamente ordenado. A *ordenação por seleção*, também chamada *selection sort*, é ilustrada na Figura 8.8 e implementada na Figura 8.9.

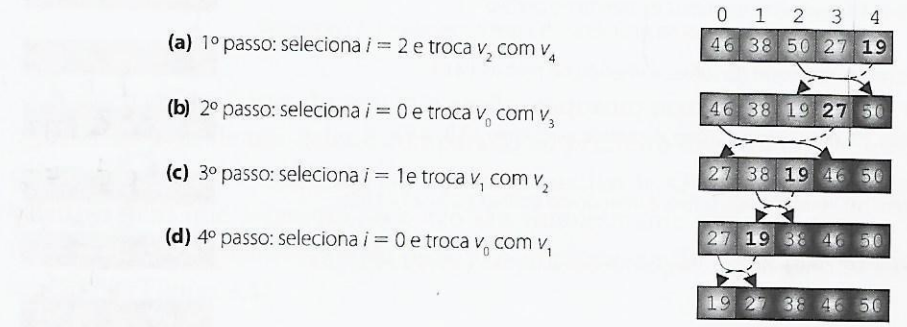


Figura 8.8 | Funcionamento da ordenação por seleção.

```
int seleciona(int v[], int n) {
   int i = 0;
   for(int j=1; j<n; j++)
      if( v[i]<v[j] ) i = j;
   return i;
}

void selection_sort(int v[], int n) {
   while( n>1 ) {
      troca( v[seleciona(v,n)], v[n-1] );
      n--;
   }
}
```

Figura 8.9 | Função para ordenação por seleção.

A ordenação por seleção chama a *função de seleção* $n - 1$ vezes. Na primeira vez, essa função faz $n - 1$ comparações para encontrar o índice do item máximo em v; na segunda, ela faz $n - 2$; na terceira, $n - 3$ e assim por diante. Portanto, o total de comparações feitas é $(n - 1) + (n - 2) + (n - 3) + ... + 2 + 1 = (n^2 - n)/2$. Logo, a complexidade de tempo da ordenação por seleção, no pior caso, é $O(n^2)$. Porém,

como a ordenação por seleção faz apenas $n - 1$ trocas, para itens *grandes* (por exemplo, cadeias e registros), ela é mais rápida que a ordenação por trocas.

8.2.3 Ordenação por inserção

Para ordenar um vetor v com n itens, a *ordenação por inserção* assume que v tem um *prefixo* ordenado e um *sufixo* desordenado. Inicialmente, o prefixo está vazio e o sufixo contém todos os itens de v. Em cada passo, um item é removido do sufixo e *inserido em ordem* no prefixo. Assim, após cada passo, o prefixo aumenta e o sufixo diminui. Quando o sufixo fica vazio, o prefixo ordenado contém todos os itens de v. A *ordenação por inserção*, também chamada *insertion sort*, é ilustrada na Figura 8.10 e implementada na Figura 8.11.

(a) 1º passo: o item 46, removido do sufixo, é inserido no prefixo {}

(b) 2º passo: o item 19, removido do sufixo, é inserido no prefixo {46}

(c) 3º passo: o item 27, removido do sufixo, é inserido no prefixo {19,46}

(d) 4º passo: o item 50, removido do sufixo, é inserido no prefixo {19,27,46}

(e) 5º passo: o item 38, removido do sufixo, é inserido no prefixo {19,27,46,50}

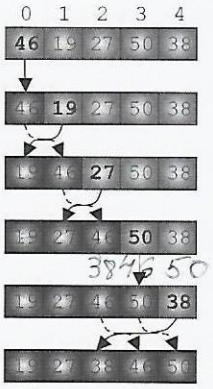


Figura 8.10 | Funcionamento da ordenação por inserção.

```
void insere(int x, int v[], int n) {
   while( n>0 && x<v[n-1] ) {
      v[n] = v[n-1];
      n--;
   }
   v[n] = x;
}
void insertion_sort(int v[], int n) {
   for(int i=0; i<n; i++)
      insere(v[i],v,i);
}
```

Figura 8.11 | Função para ordenação por inserção.

A ordenação por inserção chama a *função de inserção* n vezes. Na primeira vez, essa função não faz nenhuma comparação de itens; na segunda, ela faz no máximo uma comparação; na terceira, no máximo duas e assim por diante. Portanto, o máximo de comparações é $1 + 2 + ... + (n - 2) + (n - 1) = (n^2 - n)/2$. Logo, a complexidade de tempo da ordenação por inserção, no pior caso, é $O(n^2)$. Mas, como mover um item é mais rápido que trocar a

posição de dois itens no vetor, a ordenação por inserção é mais rápida que a ordenação por trocas.

8.2.4 Ordenação por intercalação

Intercalação é uma operação que combina dois subvetores ordenados para obter um único vetor ordenado. Isto é, dado um vetor v, indexado de p até u, tal que os subvetores $v_p, ..., v_m$ e $v_{m+1}, ..., v_u$ estejam ordenados, a intercalação combina os itens desses dois subvetores, de modo que o vetor v inteiro fique ordenado.

A Figura 8.12 ilustra o funcionamento da operação de intercalação.

Figura 8.12 | Funcionamento da operação de intercalação.

A intercalação é feita do seguinte modo: enquanto nenhum subvetor de v está vazio, o primeiro item de um deles é comparado ao primeiro item do outro, sendo o menor deles removido de v e inserido em um vetor auxiliar w. Quando um dos subvetores fica vazio, os itens que sobraram no outro são simplesmente copiados para w. No final, os itens de w são copiados de volta para v. A implementação desse procedimento é apresentada na Figura 8.13.

```
void intercala(int v[], int p, int m, int u) {
   int *w = malloc((u-p+1)*sizeof(int));
   if( w == NULL ) abort();
   int i = p, j = m+1, k = 0;
   while( i<=m && j<=u )
      if( v[i] < v[j] ) w[k++] = v[i++];
      else w[k++] = v[j++];
   while( i<=m ) w[k++] = v[i++];
   while( j<=u ) w[k++] = v[j++];
   for(k=0; k<=u-p; k++) v[p+k] = w[k];
   free(w);
}
```

Figura 8.13 | Função para intercalação.

A ordenação por intercalação de um vetor v, indexado de p até u, é feita *recursivamente* da seguinte maneira:

- **Base:** Se $p = u$, então v tem um único item e, portanto, já está ordenado.
- **Passo:** Se $p < u$, então v tem pelo menos dois itens e pode ser dividido em dois subvetores indexados, respectivamente, de p até m e de $m + 1$ até u. Ordenando esses subvetores, recursivamente, e intercalando os resultados obtidos, obtemos v ordenado.

Essa estratégia recursiva é ilustrada na Figura 8.14.

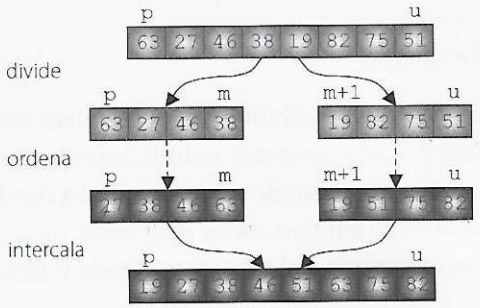


Figura 8.14 | Estratégia recursiva para ordenação por intercalação.

A *ordenação por intercalação*, também chamada *merge sort*, é ilustrada na Figura 8.15 e implementada na Figura 8.16.

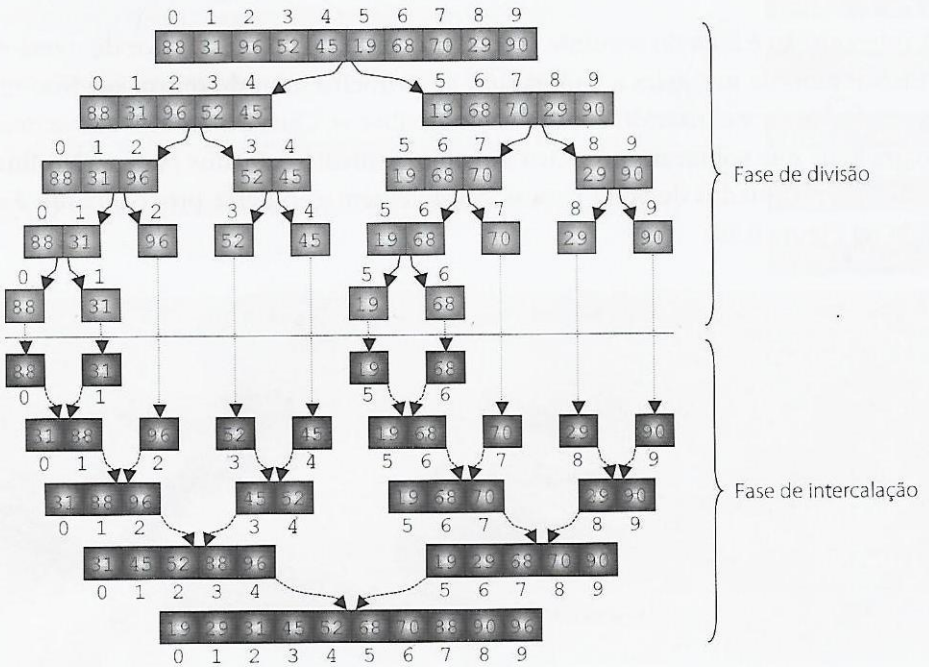


Figura 8.15 | Funcionamento da ordenação por intercalação.

```
void merge_sort(int v[], int p, int u) {
   if( p == u ) return;
   int m = (p+u)/2;
   merge_sort(v,p,m);
   merge_sort(v,m+1,u);
   intercala(v,p,m,u);
}
```

Figura 8.16 | Função para ordenação por intercalação.

Seja $n = u - p + 1$ o tamanho do vetor. A cada divisão, esse tamanho é reduzido à metade. Então, começando com n itens, após a primeira divisão, teremos subvetores com cerca de $n/2$ itens; após a segunda, $n/2^2$; após a terceira, $n/2^3$; e assim por diante, até que, após a k-ésima divisão, teremos $n/2^k$ itens. Suponha que, após a k-ésima divisão, reste apenas um item em cada subvetor (nesse caso, eles já estarão ordenados). Mas, se $n/2^k = 1$, segue que $2^k = n$, ou seja, $k = \lg n$. Então, o número máximo de níveis na fase de divisão (ou intercalação) é $O(\lg n)$. Como, em cada nível de intercalação são intercalados, no total, n itens, a complexidade de tempo da ordenação por intercalação, no pior caso, é $O(n \lg n)$.

8.3 Métodos de busca

Dados um item x e um vetor v, um *método de busca* deve informar se x é um dos n itens em v. A seguir, são apresentados os dois métodos de busca mais usados.

8.3.1 Busca linear

Para verificar se um item x está num vetor v, a *busca linear* examina cada item de v, desde v_0, até encontrar um que seja igual a x ou, então, até que todos eles tenham sido examinados. Esse método de busca, também chamado *linear search*, é implementado na Figura 8.17.

```
int linear_search(int x, int v[], int n) {
   for(int i=0; i<n; i++)
      if( x == v[i] )
         return 1;
   return 0;
}
```

Figura 8.17 | Função para busca linear.

A complexidade de tempo da busca linear, no pior caso, é $O(n)$. A vantagem desse método é que ele funciona mesmo quando o vetor não está ordenado.

8.3.2 Busca binária

Seja v um vetor *ordenado*, cujo primeiro item é v_p e cujo último item é v_u. Para verificar se um item x está em v, a *busca binária* examina o item v_m que está no *meio* de v. Então, se $x = v_m$, a busca termina com sucesso; senão, ela continua no *subvetor* v_p, ..., v_{m-1} (se $x < v_m$) ou no *subvetor* v_{m+1}, ..., v_u (se $x > v_m$). A busca termina com fracasso quando o subvetor considerado está *vazio* (isto é, quando o índice p desse subvetor é maior que o seu índice u). Esse método de busca, também chamado *binary search*, é implementado na Figura 8.18.

```
int binary_search(int x, int v[], int n) {
   int p = 0;
   int u = n-1;
   while( p <= u ) {
      int m = (p+u)/2;
      if( x == v[m] ) return 1;
      if( x < v[m] ) u = m-1;
      else p = m+1;
   }
   return 0;
}
```

Figura 8.18 | Função para busca binária.

A cada comparação, a busca binária reduz o total de itens à metade. Então, começando com n itens, após a primeira comparação, teremos $n/2$ itens; após a segunda, $n/2^2$; após a terceira, $n/2^3$; e assim por diante, até que, após a k-ésima comparação, teremos $n/2^k$ itens. Suponha que, após k-ésima comparação, reste apenas um item (nesse caso, uma última comparação é suficiente para concluir a busca). Mas, se $n/2^k = 1$, segue que $2^k = n$, ou seja, $k = \lg n$. Então, o número máximo de comparações que podem feitas num vetor com n itens é $\lg n + 1$. Logo, a complexidade de tempo da busca binária, no pior caso, é $O(\lg n)$.

Exercícios

8.1 Altere `bubble_sort()`, para ordenar vetores de números reais.

8.2 Altere `insertion_sort()`, para ordenar vetores de cadeias. *Dica*: (*a*) use as funções `strcmp()` e `strcpy()`, declaradas em `string.h`, para comparar e copiar cadeias e (*b*) copie `v[i]` para um vetor de caracteres `aux` e chame `insere(aux,v,i)`.

8.3 Usando a função `empurra()`, que move o item máximo de um vetor para sua última posição, crie a função recursiva `bubbleSort(v,n)`, que ordena um vetor `v` com `n` números inteiros.

```
void empurra(int v[], int n) {
   for(int i=0; i<n; i++)
      if( v[i]>v[i+1] )
         troca(v[i],v[i+1]);
}
```

8.4 Usando a função `seleciona()`, definida na Figura 8.9, crie a função recursiva `selectionSort(v,n)`, que ordena um vetor `v` com `n` números inteiros.

8.5 Usando a função `insere()`, definida na Figura 8.11, crie a função recursiva `insertionSort(v,n)`, que ordena um vetor `v` com `n` números inteiros.

8.6 Crie a função recursiva `linearSearch(x,v,n)`, que faz uma busca linear para verificar se o item `x` está no vetor `v`, que contém `n` números inteiros.

8.7 Crie a função recursiva `binarySearch(x,v,p,u)`, que faz uma busca binária para verificar se o número inteiro `x` está no vetor `v`, indexado de `p` até `u`.

8.8 A *ordenação por partição*, também chamada *quick sort*, é considerada um dos mais eficientes algoritmos de ordenação. Dado um vetor v, indexado de p até u, esse algoritmo usa o item $x = v_p$ como *pivô* para particionar o vetor v em dois subvetores indexados de p até m e de $m + 1$ até u, respectivamente, de modo que todo item com índice no intervalo $[p..m]$ seja menor ou igual a x e todo item com índice no intervalo $[m + 1 ..u]$ seja maior ou igual a x. Em seguida, cada subvetor é ordenado recursivamente, como indicado a seguir:

```
int particiona(int v[], int p, int u) {
   int x = v[p];
   p--;
   u++;
   while( p<u ) {
      do u--; while( v[u]>x );
      do p++; while( v[p]<x );
      if( p<u ) troca(v[p],v[u]);
   }
   return u;
}
void quick_sort(int v[], int p, int u) {
   if( p>=u ) return;
   int m = particiona(v,p,u);
   quick_sort(v,p,m);
   quick_sort(v,m+1,u);
}
```

(a) Simule a execução de `quick_sort()` para `v = {72,69,51,46,33,28,15}`.
(b) Simule a execução de `quick_sort()` para `v = {46,28,51,33,72,15,69}`.
(c) Mostre que, no *pior caso*, a complexidade de `quick_sort()` é $O(n^2)$.
(d) Mostre que, no *melhor caso*, a complexidade de `quick_sort()` é $O(n \lg n)$.