15. Ordenação

W. Celes e J. L. Rangel

Em diversas aplicações, os dados devem ser armazenados obedecendo uma determinada ordem. Alguns algoritmos podem explorar a ordenação dos dados para operar de maneira mais eficiente, do ponto de vista de desempenho computacional. Para obtermos os dados ordenados, temos basicamente duas alternativas: ou inserimos os elementos na estrutura de dados respeitando a ordenação (dizemos que a ordenação é garantida por construção), ou, a partir de um conjunto de dados já criado, aplicamos um algoritmo para ordenar seus elementos. Neste capítulo, vamos discutir dois algoritmos de ordenação que podem ser empregados em aplicações computacionais.

Devido ao seu uso muito freqüente, é importante ter à disposição algoritmos de ordenação (*sorting*) eficientes tanto em termos de tempo (devem ser rápidos) como em termos de espaço (devem ocupar pouca memória durante a execução). Vamos descrever os algoritmos de ordenação considerando o seguinte cenário:

- a entrada é um vetor cujos elementos precisam ser ordenados;
- a saída é o mesmo vetor com seus elementos na ordem especificada;
- o espaço que pode ser utilizado é apenas o espaço do próprio vetor.

Portanto, vamos discutir *ordenação de vetores*. Como veremos, os algoritmos de ordenação podem ser aplicados a qualquer informação, desde que exista uma ordem definida entre os elementos. Podemos, por exemplo, ordenar um vetor de valores inteiros, adotando uma ordem crescente ou decrescente. Podemos também aplicar algoritmos de ordenação em vetores que guardam informações mais complexas, por exemplo um vetor que guarda os dados relativos a alunos de uma turma, com nome, número de matrícula, etc. Nesse caso, a ordem entre os elementos tem que ser definida usando uma das informações do aluno como chave da ordenação: alunos ordenados pelo nome, alunos ordenados pelo número de matrícula, etc.

Nos casos em que a informação é complexa, raramente se encontra toda a informação relevante sobre os elementos do vetor no próprio vetor; em vez disso, cada componente do vetor pode conter apenas um ponteiro para a informação propriamente dita, que pode ficar em outra posição na memória. Assim, a ordenação pode ser feita sem necessidade de mover grandes quantidades de informação, para re-arrumar as componentes do vetor na sua ordem correta. Para trocar a ordem entre dois elementos, apenas os ponteiros são trocados. Em muitos casos, devido ao grande volume, a informação pode ficar em um arquivo de disco, e o elemento do vetor ser apenas uma referência para a posição da informação nesse arquivo.

Neste capítulo, examinaremos os algoritmos de ordenação conhecidos como "ordenação bolha" (*bubble sort*) e "ordenação rápida" (*quick sort*), ou, mais precisamente, versões simplificadas desses algoritmos.

15.1. Ordenação bolha

O algortimo de "ordenação bolha", ou "*bubble sort*", recebeu este nome pela imagem pitoresca usada para descrevê-lo: os elementos maiores são mais leves, e sobem como bolhas até suas posições corretas. A idéia fundamental é fazer uma série de comparações entre os elementos do vetor. Quando dois elementos estão fora de ordem,

há uma inversão e esses dois elementos são trocados de posição, ficando em ordem correta. Assim, o primeiro elemento é comparado com o segundo. Se uma inversão for encontrada, a troca é feita. Em seguida, independente se houve ou não troca após a primeira comparação, o segundo elemento é comparado com o terceiro, e, caso uma inversão seja encontrada, a troca é feita. O processo continua até que o penúltimo elemento seja comparado com o último. Com este processo, garante-se que o elemento de maior valor do vetor será levado para a última posição. A ordenação continua, posicionando o segundo maior elemento, o terceiro, etc., até que todo o vetor esteja ordenado.

Para exemplificar, vamos considerar que os elementos do vetor que queremos ordenar são valores inteiros. Assim, consideremos a ordenação do seguinte vetor:

```
25 48 37 12 57 86 33 92
```

Seguimos os passos indicados:

```
25 48 37 12 57 86 33 92       25x48
25 48 37 12 57 86 33 92       48x37 troca
25 37 48 12 57 86 33 92       48x12 troca
25 37 12 48 57 86 33 92       48x57
25 37 12 48 57 86 33 92       57x86
25 37 12 48 57 86 33 92       86x33 troca
25 37 12 48 57 33 86 92       86x92
25 37 12 48 57 33 86 92       final da primeira passada
```

Neste ponto, o maior elemento, `92`, já está na sua posição final.

```
25 37 12 48 57 33 86 92       25x37
25 37 12 48 57 33 86 92       37x12 troca
25 12 37 48 57 33 86 92       37x48
25 12 37 48 57 33 86 92       48x57
25 12 37 48 57 33 86 92       57x33 troca
25 12 37 48 33 57 86 92       57x86
25 12 37 48 33 57 86 92       final da segunda passada
```

Neste ponto, o segundo maior elemento, `86`, já está na sua posição final.

```
25 12 37 48 33 57 86 92       25x12 troca
12 25 37 48 33 57 86 92       25x37
12 25 37 48 33 57 86 92       37x48
12 25 37 48 33 57 86 92       48x33 troca
12 25 37 33 48 57 86 92       48x57
12 25 37 33 48 57 86 92       final da terceira passada
```

Idem para `57`.

```
12 25 37 33 48 57 86 92       12x25
12 25 37 33 48 57 86 92       25x37
12 25 37 33 48 57 86 92       37x33 troca
12 25 33 37 48 57 86 92       37x48
12 25 33 37 48 57 86 92       final da quarta passada
```

Idem para `48`.

```
12 25 33 37 48 57 86 92       12x25
12 25 33 37 48 57 86 92       25x33
12 25 33 37 48 57 86 92       33x37
12 25 33 37 48 57 86 92       final da quinta passada
```

Idem para `37`.

```
12 25 33 37 48 57 86 92       12x25
12 25 33 37 48 57 86 92       25x33
12 25 33 37 48 57 86 92       final da sexta passada
```

Idem para `33`.

```
12 25 33 37 48 57 86 92       12x25
12 25 33 37 48 57 86 92       final da sétima passada
```

Idem para `25` e, consequentemente, `12`.

```
12 25 33 37 48 57 86 92       final da ordenação
```

A parte sabidamente já ordenada do vetor está sublinhada. Na realidade, após a troca de `37x33`, o vetor se encontra totalmente ordenado, mas esse fato não é levado em consideração por esta versão do algoritmo.

Uma função que implementa esse algoritmo é apresentada a seguir. A função recebe como parâmetros o número de elementos e o ponteiro do primeiro elemento do vetor que se deseja ordenar. Vamos considerar o ordenação de um vetor de valores inteiros.

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

Uma função cliente para testar esse algoritmo pode ser dada por:

```
/* Testa algoritmo de ordenação bolha */
#include <stdio.h>

int main (void)
{
   int i;
   int v[8] = {25,48,37,12,57,86,33,92};
   bolha(8,v);
   printf("Vetor ordenado: ");
   for (i=0; i<8; i++)
      printf("%d ",v[i]);
   printf("\n");
   return 0;
}
```

Para evitar que o processo continue mesmo depois de o vetor estar ordenado, podemos interromper o processo quando houver uma passagem inteira sem trocas, usando uma variante do algoritmo apresentado acima:

```
/* Ordenação bolha (2a. versão) */
void bolha2 (int n, int* v)
{
   int i, j;
   for (i=n-1; i>0; i--) {
      int troca = 0;
      for (j=0; j<i; j++)
         if (v[j]>v[j+1]) { /* troca */
            int temp = v[j];
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
         }
      if (troca == 0)   /* nao houve troca */
         return;
   }
}
```

A variável `troca` guarda o valor `0` (falso) quando uma passada do vetor (no `for` interno) se faz sem nenhuma troca.

O esforço computacional despendido pela ordenação de um vetor por este procedimento é fácil de se determinar, pelo número de comparações, que serve também para estimar o número máximo de trocas que podem ser realizadas. Na primeira passada, fazemos `n-1` comparações; na segunda, `n-2`; na terceira `n-3`; e assim por diante. Logo, o tempo total gasto pelo algoritmo é proporcional a `(n-1) + (n-2) + ... + 2 + 1`. A soma desses termos é proporcional ao quadrado de `n`. Dizemos que o algoritmo é de ordem quadrática e representamos isso escrevendo $O(n^2)$.

Implementação recursiva

Analisando a forma com que a ordenação bolha funciona, verificamos que o algoritmo procura resolver o problema da ordenação por partes. Inicialmente, o algoritmo coloca em sua posição (no final do vetor) o maior elemento, e o problema restante é semelhante ao inicial, só que com um vetor com menos elementos, formado pelos elementos `v[0],…,v[n-2]`.

Baseado nessa observação, é fácil implementar um algoritmo de ordenação bolha recursivamente. Embora não seja a forma mais adequada de implementarmos esse algoritmo, o estudo dessa recursão nos ajudará a entender a idéia por trás do próximo algoritmo de ordenação que veremos mais adiante.

O algoritmo recursivo de ordenação bolha posiciona o elemento de maior valor e chama, recursivamente, o algoritmo para ordenar o vetor restante, com `n-1` elementos.

```
/* Ordenação bolha recursiva */
void bolha_rec (int n, int* v)
{
   int j;
   int troca = 0;
   for (j=0; j<n-1; j++)
      if (v[j]>v[j+1]) { /* troca */
         int temp = v[j];
         v[j] = v[j+1];
         v[j+1] = temp;
         troca = 1;
      }
   if (troca != 0)   /* houve troca */
      bolha_rec(n-1,v);
}
```

Algoritmo genérico**

Esse mesmo algoritmo pode ser aplicado a vetores que guardam outras informações. O código escrito acima pode ser reaproveitado, a menos de alguns detalhes. Primeiro, a assinatura da função deve ser alterada, pois deixamos de ter um vetor de inteiros; segundo, a forma de comparação entre os elementos também deve ser alterada, pois não podemos, por exemplo, comparar duas cadeias de caracteres usando simplesmente o operador relacional "maior que" (>).

Para aumentar o potencial de reuso do nosso código, podemos re-escrever o algoritmo de ordenação apresentado acima tornando-o independente da informação armazenada no vetor. Vamos inicialmente discutir como podemos abstrair a função de comparação. O mesmo algoritmo para ordenação de inteiros apresentado acima pode ser re-escrito usando-se uma função auxiliar que faz a comparação. Em vez de compararmos diretamente dois elementos com o operador "maior que", usamos uma função auxiliar que, dados dois elementos, verifica se o primeiro é maior que o segundo.

```
/* Função auxiliar de comparação */
int compara (int a, int b)
{
   if (a > b)
      return 1;
   else
      return 0;
}
/* Ordenação bolha (3a. versão) */
void bolha (int n, int* v)
{
   int i, j;
   for (i=n-1; i>0; i--) {
      int troca = 0;
      for (j=0; j<i; j++)
         if (compara(v[j],v[j+1])) { /* troca */
            int temp = v[j];
            v[j] = v[j+1];
            v[j+1] = temp;
            troca = 1;
         }
      if (troca == 0)   /* nao houve troca */
         return;
   }
}
```

Desta forma, já aumentamos o potencial de reuso do algoritmo. Podemos, por exemplo, arrumar os elementos em ordem decrescente simplesmente re-escrevendo a função compara. A idéia fundamental é escrever uma função de comparação que recebe dois

elementos e verifica se há uma inversão de ordem entre o primeiro e o segundo. Assim, se tivéssemos um vetor de cadeia de caracteres para ordenar, poderíamos usar a seguinte função de ordenação.

```
int compara (char* a, char* b)
{
   if (strcmp(a,b) > 0)
      return 1;
   else
      return 0;
}
```

Consideremos agora um vetor de ponteiros para a estrutura `Aluno`:

```
struct aluno {
   char nome[81];
   char mat[8];
   char turma;
   char email[41];
};
```

Uma função de comparação, neste caso, receberia como parâmetros dois ponteiros para a estrutura que representa um aluno e, considerando uma ordenação que usa o nome do aluno como chave de comparação, poderia ter a seguinte implementação:

```
int compara (Aluno* a, Aluno* b)
{
   if (strcmp(a->nome,b->nome) > 0)
      return 1;
   else
      return 0;
}
```

Portanto, o uso de uma função auxiliar para realizar a comparação entre os elementos ajuda para a obtenção de um código reusável. No entanto, isto só não é suficiente. Para o mesmo código poder ser aplicado a qualquer tipo de informação armazenada no vetor, precisamos tornar a implementação independente do tipo do elemento, isto é, precisamos tornar tanto a própria função de ordenação (`bolha`) quanto a função de comparação (`compara`) independentes do tipo do elemento.

Em C, a forma de generalizar o tipo é usar o tipo `void*`. Escreveremos o código de ordenação considerando que temos um ponteiro de qualquer tipo e passaremos para a função de comparação dois ponteiros genéricos, um para cada elemento que se deseja comparar. A função de ordenação, no entanto, precisa percorrer o vetor e para tanto precisamos passar para a função uma informação adicional–o tamanho, em número de bytes, de cada elemento. A assinatura da função de ordenação poderia então ser dada por:

```
void buble (int n, void* v, int tam);
```

A função de ordenação por sua vez, recebe dois ponteiros genéricos:

```
int compara (void* a, void* b);
```

Assim, se estamos ordenando vetores de inteiros, escrevemos a nossa função de comparação convertendo o ponteiro genérico para um ponteiro de inteiro e fazendo o teste apropriado:

```
/* função de comparação para inteiros */
int compara (void* a, void* b)
{
   int* p1 = (int*) a;
   int* p2 = (int*) b;
   int i1 = *p1;
   int i2 = *p2;
   if (i1 > i2)
      return 1;
   else
      return 0;
}
```

Se os elementos do vetor fossem ponteiros para a estrutura aluno, a função de comparação poderia ser:

```
/* função de comparação para ponteiros de alunos */
int compara (void* a, void* b)
{
   Aluno** p1 = (Aluno**) a;
   Aluno** p2 = (Aluno**) b;
   Aluno* i1 = *p1;
   Aluno* i2 = *p2;
   if (strcmp(i1->nome,i2->nome) > 0)
      return 1;
   else
      return 0;
}
```

O código da função de ordenação necessita percorrer os elementos do vetor. O acesso a um determinado elemento `i` do vetor não pode mais ser feito diretamente por `v[i]`. Dado o endereço do primeiro elemento do vetor, devemos incrementar este endereço de `i*tam` bytes para termos o endereço do elemento `i`. Podemos então escrever uma função auxiliar que faz esse incremento de endereço. Essa função recebe como parâmetros o endereço inicial do vetor, o índice do elemento cujo endereço se quer alcançar e o tamanho (em bytes) de cada elemento. A função retorna o endereço do elemento especificado. Uma parte sutil, porém necessária, dessa função é que para incrementar o endereço genérico de um determinado número de bytes, precisamos antes, temporariamente, converter esse ponteiro para ponteiro para caractere (pois um caractere ocupa um byte). O código dessa função auxiliar pode ser dado por:

```
void* acessa (void* v, int i, int tam)
{
   char* t = (char*)v;
   t += tam*i;
   return (void*)t;
}
```

A função de ordenação identifica a ocorrência de inversões entre elementos e realiza uma troca entre os valores. O código que realiza a troca também tem que ser pensado de forma genérica, pois, como não sabemos o tipo de cada elemento, não temos como declarar a variável temporária para poder realizar a troca. Uma alternativa é fazer a troca dos valores byte a byte (ou caractere a caractere). Para tanto, podemos definir uma outra função auxiliar que recebe os ponteiros genéricos dos dois elementos que devem ter seus valores trocados, além do tamanho de cada elemento.

```
void troca (void* a, void* b, int tam)
{
   char* v1 = (char*) a;
   char* v2 = (char*) b;
   int i;
   for (i=0; i<tam; i++) {
      char temp = v1[i];
      v1[i] = v2[i];
      v2[i] = temp;
   }
}
```

Assim, podemos escrever o código da nossa função de ordenação genérica. Falta, no entanto, um último detalhe. As funções auxiliares acessa e troca são realmente genéricas, e independem da informação efetivamente armazenada no vetor. Porém, a função de comparação deve ser especializada para cada tipo de informação, conforme ilustramos acima. A assinatura dessa função é genérica, mas a sua implementação deve, naturalmente, levar em conta a informação armazenada para que a comparação tenha sentido. Portanto, para generalizar a implementação da função de ordenação, não podemos chamar uma função de comparação específica. A solução é passar, via parâmetro, qual função de ordenação deve ser chamada. Para tanto, temos que introduzir o conceito de *ponteiro para função*. O nome de uma função representa o endereço dessa função. A nossa função de comparação tem a assinatura:

```
int compara (void*, void*);
```

Uma variável ponteiro para armazenar o endereço dessa função é declarada como:

```
int(*cmp)(void*,void*);
```

onde `cmp` representa a variável do tipo ponteiro para a função em questão.

Agora sim, podemos escrever nossa função de ordenação genérica, recebendo como parâmetro adicional o ponteiro da função de comparação:

```
/* Ordenação bolha (genérica) */
void bolha_gen (int n, void* v, int tam, int(*cmp)(void*,void*))
{
   int i, j;
   for (i=n-1; i>0; i--) {
      int fez_troca = 0;
      for (j=0; j<i; j++) {
         void* p1 = acessa(v,j,tam);
         void* p2 = acessa(v,j+1,tam);
         if (cmp(p1,p2))
            troca(p1,p2,tam);
            fez_troca = 1;
         }
      if (fez_troca == 0)   /* nao houve troca */
         return;
   }
}
```

Esse código genérico pode ser usado para ordenar vetores com qualquer informação. Para exemplificar, vamos usá-lo para ordenar um vetor de números reais. Para isso, temos que escrever o código da função que faz a comparação, agora especializada para número reais:

```
int compara_reais (void* a, void* b)
{
   float* p1 = (float*) a;
   float* p2 = (float*) b;
   float f1 = *p1;
   float f2 = *p2;
   if (f1 > f2)
      return 1;
   else
      return 0;
}
```

Podemos, então, chamar a função para ordenar um vetor `v` de `n` números reais:

```
   ...
   buble_gen(n,v,sizeof(float),compara_reais);
   ...
```

15.2. Ordenação Rápida

Assim como o algoritmo anterior, o algoritmo "ordenação rápida", "*quick sort"*, que iremos discutir agora, procura resolver o problema da ordenação por partes. No entanto, enquanto o algoritmo de ordenação bolha coloca em sua posição (no final do vetor) o maior elemento, a ordenação rápida faz isso com um elemento arbitrário `x`, chamado de pivô. Por exemplo, podemos escolher como pivô o primeiro elemento do vetor, e posicionar esse elemento em sua correta posição numa primeira passada.

Suponha que este elemento, `x`, deva ocupar a posição `i` do vetor, de acordo com a ordenação, ou seja, que essa seja a sua posição definitiva no vetor. Sem ordenar o vetor completamente, este fato pode ser reconhecido quando todos os elementos `v[0]`, … `v[i-1]` são menores que `x`, e todos os elementos `v[i+1]`, …, `v[n-1]` são maiores que `x`. Supondo que `x` já está na sua posição correta, com índice `i`, há dois problemas menores para serem resolvidos: ordenar os (sub-) vetores formados por `v[0]`, … `v[i-1]` e por `v[i+1]`, …, `v[n-1]`. Esses sub-problemas são resolvidos (recursivamente) de forma semelhante, cada vez com vetores menores, e o processo continua até que os vetores que devem ser ordenados tenham zero ou um elementos, caso em que sua ordenação já está concluída.

A grande vantagem desse algoritmo é que ele pode ser muito eficiente. O melhor caso ocorre quando o elemento pivô representa o valor mediano do conjunto dos elementos do vetor. Se isto acontece, após o posicionamento do pivô em sua posição, restará dois sub-vetores para serem ordenados, ambos com o número de elementos reduzido a metade, em relação ao vetor original. Pode-se mostrar que, neste melhor caso, o esforço computacional do algoritmo é proporcional a `n log(n)`, e dizemos que o algoritmo é O(n log(n)). Um desempenho muito superior ao $O(n^2)$ apresentado pelo algoritmo de ordenação bolha. Infelizmente, não temos como garantir que o pivô seja o mediano. No pior caso, o pivô pode sempre ser, por exemplo, o maior elemento, e recaímos no algoritmo de ordenação bolha. No entanto, mostra-se que o algoritmo *quicksort* ainda apresenta, no caso médio, um desempenho O(n log(n)).

A versão do "*quicksort*" que vamos apresentar aqui usa `x=v[0]` como primeiro elemento a ser colocado em sua posição correta. O processo compara os elementos `v[1]`, `v[2]`, … até encontrar um elemento `v[a]>x`. Então, a partir do final do vetor, compara os elementos `v[n-1]`, `v[n-2]`, … até encontrar um elemento `v[b]<=x`.

Neste ponto, `v[a]` e `v[b]` são trocados, e a busca continua, para cima a partir de `v[a+1]`, e para baixo, a partir de `v[b-1]`. Em algum momento, a busca termina, porque os pontos de busca se encontrarão. Neste momento, a posição correta de `x` está definida, e os valores `v[0]` e `v[a]` são trocados.

Vamos usar o mesmo exemplo da seção anterior:

```
(0-7) 25 48 37 12 57 86 33 92
```

onde indicamos através de `(0-7)` que se trata do vetor inteiro, de `v[0]` a `v[7]`. Podemos começar a executar o algoritmo procurando determinar a posição correta de `x=v[0]=25`. Partindo do início do vetor, já temos, na primeira comparação, `48>25 (a=1)`. Partindo do final do vetor, na direção oposta, temos `25<92`, `25<33`, `25<86`, `25<57` e finalmente, `12<=25 (b=3)`.

```
(0-7) 25 48 37 12 57 86 33 92
        a↑    b↑
```

Trocamos então `v[a]=48` e `v[b]=12`, incrementando `a` de uma unidade e decrementando `b` de uma unidade. Os elementos do vetor ficam com a seguinte disposição:

```
(0-7) 25 12 37 48 57 86 33 92
         a,b↑
```

Na continuação, temos `37>25 (a=2)`. Pelo outro lado, chegamos também a `37` e temos `37>25` e `12<=25`. Neste ponto, verificamos que os índices `a` e `b` se cruzaram, agora com `b<a`.

```
(0-7) 25 12 37 48 57 86 33 92
        b↑ a↑
```

Assim, todos os elementos de `37` (inclusive) em diante são maiores que `25`, e todos os elementos de `12` (inclusive) para trás são menores que `25`. Com exceção do próprio `25`, é claro. A próxima etapa troca o pivô, `v[0]=25`, com o último dos valores menores que `25` encontrado: `v[b]=12`. Temos:

```
(0-7)12 25 37 48 57 86 33 92
```

com `25` em sua posição correta, e dois vetores menores para ordenar. Valores menores que `25`:

```
(0-0) 12
```

E valores maiores:

```
(2-7) 37 48 57 86 33 92
```

Neste caso, em particular, o primeiro vetor (com apenas um elemento: `(0-0)`) já se encontra ordenado. O segundo vetor `(2-7)` pode ser ordenado de forma semelhante:

```
(2-7) 37 48 57 86 33 92
```

Devemos decidir qual a posição correta de `37`. Para isso identificamos o primeiro elemento maior que `37`, ou seja, `48`, e o último menor que `37`, ou seja, `33`.

```
(2-7) 37 48 57 86 33 92
        a↑       b↑
```

Trocamos os elementos e atualizamos os índices:

```
(2-7) 37 33 57 86 48 92
          a↑ b↑
```

Continuando o processo, verificamos que `37<57` e `37<86`, `37<57`, mas `37>=33`. Identificamos novamente que `a` e `b` se cruzaram.

```
(2-7) 37 33 57 86 48 92
        b↑ a↑
```

Assim, a posição correta de `37` é a posição ocupada por `v[b]`, e os dois elementos devem ser trocados:

```
(2-7) 33 37 57 86 48 92
```

restando os vetores

```
(2-2) 33
```

e

```
(4-7) 57 86 48 92
```

para serem ordenados.

O processo continua até que o vetor original esteja totalmente ordenado.

```
(0-7) 12 25 33 37 48 57 86 92
```

A implementação do *quick sort* é normalmente recursiva, para facilitar a ordenação dos dois vetores menores encontrados. A seguir, apresentamos uma possível implementação do algoritmo, adotando como pivô o primeiro elemento.

```
/* Ordenação rápida */
void rapida (int n, int* v)
{
   if (n <= 1)
      return;
   else {
      int x = v[0];
      int a = 1;
      int b = n-1;
      do {
         while (a < n && v[a] <= x) a++;
         while (v[b] > x) b--;
         if (a < b) {          /* faz troca */
            int temp = v[a];
            v[a] = v[b];
            v[b] = temp;
            a++; b--;
         }
      } while (a <= b);

      /* troca pivô */
      v[0] = v[b];
      v[b] = x;

      /* ordena sub-vetores restantes */
      rapida(b,v);
      rapida(n-a,&v[a]);
   }
}
```

Devemos observar que para deslocar o índice `a` para a direita, fizemos o teste:

```
while (a < n && v[a] <= x)
```

enquanto que para deslocar o índice `b` para a esquerda, fizemos apenas:

```
while (v[b] > x)
```

O teste adicional no deslocamento para a direita é necessário porque o pivô pode ser o elemento de maior valor, nunca ocorrendo a situação `v[a]<=x`, o que nos faria acessar posições além dos limites do vetor. No deslocamento para a esquerda, um teste adicional tipo `b>=0` não é necessário, pois, na nossa implementação, `v[0]` é o pivô, impedindo que `b` assuma valores negativos (teremos, pelo menos, `x>=v[0]`).

Algoritmo genérico da biblioteca padrão

O *quicksort* é o algoritmo de ordenação mais utilizado no desenvolvimento de aplicações. Mesmo quando temos os dados organizados em listas encadeadas, e precisamos colocá-los de forma ordenada, em geral, optamos por criar um vetor temporário com ponteiros para os nós da lista, fazer a ordenação usando *quicksort* e re-encadear os nós montando a lista ordenada.

Devido a sua grande utilidade, a biblioteca padrão de C disponibiliza, via a interface `stdlib.h`, uma função que ordena vetores usando esse algoritmo. A função disponibilizada pela biblioteca independe do tipo de informação armazenada no vetor. A implementação dessa função genérica segue os princípios discutidos na implementação do algoritmo de ordenação bolha genérico. O protótipo da função disponibilizada pela biblioteca é:

```
void qsort (void *v, int n, int tam, int (*cmp)(const void*, const
void*));
```

Os parâmetros de entrada dessa função são:

- `v`: o ponteiro para o primeiro elemento do vetor que se deseja ordenar. Como não se sabe, a priori, o tipo dos elementos do vetor, temos um ponteiro genérico – `void*`.
- `n`: o número de elementos do vetor.
- `tam`: o tamanho, em bytes, de cada elemento do vetor.
- `cmp`: o ponteiro para a função responsável por comparar dois elementos do vetor. Em C, o nome de uma função representa o *ponteiro da função*. Esse ponteiro pode ser armazenado numa variável, possibilitando chamar a função indiretamente. Como era de se esperar, a biblioteca não sabe comparar dois elementos do vetor (ela desconhece o tipo desses elementos). Fica a cargo do cliente da função de ordenação escrever a função de comparação. Essa função de comparação tem que ter o seguinte protótipo:

  ```
  int nome (const void*, const void*);
  ```

  O parâmetro `cmp` recebido pela função `qsort` é um ponteiro para uma função com esse protótipo. Assim, para usarmos a função de ordenação da biblioteca temos que escrever uma função que receba dois ponteiros genéricos, `void*`, os quais representam ponteiros para os dois elementos que se deseja comparar. O modificador de tipo `const` aparece no protótipo apenas para garantir que essa função não modificará os valores dos elementos (devem ser tratados como valores constantes). Essa função deve ter como valor de retorno `-1`, `0`, ou `1`, dependendo se o primeiro elemento for menor, igual, ou maior que o segundo, respectivamente.

Para ilustrar a utilização da função `qsort` vamos considerar alguns exemplos. O código a seguir ilustra a utilização da função para ordenar valores reais. Neste caso, os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `float`.

```
/* Ilustra uso do algoritmo qsort */
#include <stdio.h>
#include <stdlib.h>

/* função de comparação de reais */
int comp_reais (const void* p1, const void* p2)
{
   /* converte ponteiros genéricos para ponteiros de float */
   float *f1 = (float*)p1;
   float *f2 = (float*)p2;
   /* dados os ponteiros de float, faz a comparação */
   if (*f1 < *f2) return -1;
   else if (*f1 > * f2) return 1;
   else return 0;
}
```

```
/* programa que faz a ordenação de um vetor */
int main (void)
{
   int i;
   float v[8] = {25.6,48.3,37.7,12.1,57.4,86.6,33.3,92.8};

   qsort(v,8,sizeof(float),comp_reais);

   printf("Vetor ordenado: ");
   for (i=0; i<8; i++)
      printf("%g ",v[i]);
   printf("\n");
   return 0;
}
```

Vamos agora considerar que temos um vetor de alunos e que desejamos ordenar o vetor usando o nome do aluno como chave de comparação. A estrutura que representa um aluno pode ser dada por:

```
struct aluno {
   char nome[81];
   char mat[8];
   char turma;
   char email[41];
};
typedef struct aluno Aluno;
```

Vamos analisar duas situações. Na primeira, consideraremos a existência de um vetor da estrutura (por exemplo, `Aluno vet[N];`). Neste caso, cada elemento do vetor é do tipo `Aluno` e os dois ponteiros genéricos passados para a função de comparação representam ponteiros para `Aluno`. Essa função de comparação pode ser dada por:

```
/* Função de comparação: elemento é do tipo Aluno */
int comp_alunos (const void* p1, const void* p2)
   /* converte ponteiros genéricos para ponteiros de Aluno */
   Aluno *a1 = (Aluno*)p1;
   Aluno *a2 = (Aluno*)p2;
   /* dados os ponteiros de Aluno, faz a comparação */
   return strcmp(a1->nome,a2->nome);
}
```

Numa segunda situação, podemos considerar que temos um vetor de ponteiros para a estrutura aluno (por exemplo, `Aluno* vet[N];`). Agora, cada elemento do vetor é um ponteiro para o tipo `Aluno` e a função de comparação tem que tratar uma indireção a mais. Aqui, os dois ponteiros genéricos passados para a função de comparação representam ponteiros de ponteiros para `Aluno`.

```
/* Função de comparação: elemento é do tipo Aluno* */
int comp_alunos (const void* p1, const void* p2)
    /* converte ponteiros genéricos para ponteiros de ponteiros de Aluno
*/
   Aluno **pa1 = (Aluno**)p1;
   Aluno **pa2 = (Aluno**)p2;
   /* acessa ponteiro de Aluno */
   Aluno *a1 = *p1;
   Aluno *a2 = *p2;
   /* dados os ponteiros de Aluno, faz a comparação */
   return strcmp(a1->nome,a2->nome);
}
```