Capítulo 14

Métodos de Ordenação Avançados

"Análise de algoritmos: está é a área de computação onde as pessoas argumentam sobre programas e, ao mesmo tempo, provam teoremas sobre programas, ao invés de simplesmente escreverem e depurarem programas."

- **Routo Terada** -

Sumário:

14.1- Conceitos

Os métodos de ordenação vistos no capítulo anterior têm uma complexidade proporcional a n^2, ou seja, uma complexidade temporal $O(n^2)$. O objectivo deste capítulo é estudar métodos mais eficientes. Contudo iremos utilizar para esse estudo a estrutura de dados declarada no capítulo anterior.

14.2- Ordenação por Intercalação

Considere o seguinte problema. Dados dois vetores a e b de n e m elementos já ordenados, construir outro vetor c de m+n elementos também ordenado com os elementos de a e b.

Uma primeira solução, muito ingênua, seria colocar em c os elementos de a, seguidos dos elementos de b e depois ordenar o vetor c. Mas os vectores a e b já estão ordenados. Então vamos aproveitar essa propriedade, para intervalar esses vectores e obter um vector c também ordenado. Por exemplo:

```
a = {2, 5, 8, 9 }
b = {1, 3,  4 }
c = {1, 2, 3, 4, 5, 8, 9}
```

Para realizar essa intercalação, basta percorrer os dois vectores, e comparar o elemento do vector a com o elemento do vector b. Colocar no vector c o menor

elemento, e em seguida, incrementar uma unidade ao índice do vector a ou ao índice do vector b, e ao índice do vector c.

Devemos salientar que estamos a supor que os vectores a e b não possuem elementos repetidos. Mas se existirem temos duas alternativas possíveis: as ocorrências são eliminadas ou preservadas.

```
void intercalar ( TTable a, TTable b[ ], TTable  *c )
{
   int inda = 0, indb = 0, indc = 0;
   while ( indc < a.nElem + b.nElem )
     {
      if ( inda == a.nElem )
          c->item[indc++] = b.item[indb++] ;
       else
          if (j == b.item)
              c->item[indc++] = a.item[inda++];
          else
             if (a.item[inda].chave < b.item[indb].chave)
                 c->item[indc++] = a.item[inda++];
             else
                 c->item[indc++] = b.item[indb++];
     }
    c->nElem = indc-1;
 }
```

Uma outra forma de fazer essa intercalação é descrita pela seguinte função:

```
void intercalar (TTable a, TTable b, TTable *c )
  {
     int inda = 0, indb = 0, indc = 0;
     while (inda < a.nElem && indb < b.nElem )
       {
        if (a.item[inda].chave <= b.item[indbj].chave)
            c->item[indc++] = a.item[inda++];
         else
            c->[indc++] = b.item[indb++];
       }
     while ( inda < a.nElem )  c[indc++] = a.item[inda++];
     while ( indb < b.nElem ) c[indc++] = b.item[indb++];
     c->nElem = indc-1;
   }
```

Vamos optimizar as versões anteriores, com a implementação de um algoritmo compacto, que minimiza o número de comparações, e como consequência mais eficiente.

A estratégia consiste em colocar no fim de cada vector um sentinela. O valor desse sentinela depende do tipo de dados e da arquitectura do computador.

Como as chaves são do tipo de dados inteiro, e estamos a supor os vectores estão ordenados na ordem crescente, esse sentinela deverá possuir um valor igual ao maior inteiro que a linguagem C pode armazenar. Seja

```
#define MAXINT    2147483647
```

esse valor. Então o processamento termina quando os índices de varredura estiverem a apontar para esses elementos, e o vector c é contruído com a comparação dos elementos dos vectores a e b. A solução de descrevemos a seguir, implementa essa estratégia.

```
void intercalar (TTable a, TTable b, TTable *c )
 {
    int inda = 0, indb = 0, indc = 0;
    a.item[a.nElem].chave = b.item[b.nElem].chave  = MAXINT;
    while (inda < a.nElem || indb < b.nElem )
      {
        if (a.item[inda].chave <= b.item[indbj].chave)
            c->item[indc++] = a.item[inda++];
        else
            c->[indc++] = b.item[indb++];
      }
     c->nElem = indc-1;
  }
```

Para executar a intercalação de dois vectores ordenados, com n/2 elementos cada, este algoritmo realiza n/2 comparações e, no pior caso, n-1 comparações. Portanto o algoritmo tem uma complexidade linear, ou seja, de ordem 0(n).

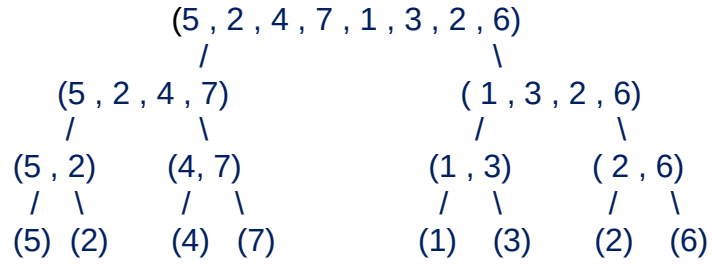
Agora, vamos tornar o problema mais complexo. Queremos intercalar os elementos do vector, utilizando o próprio vector. Para esse caso, a estratégia consiste em fragmentar o vector em dois subvectores, ordená-los separadamente, e depois intercala-los num único vector. Para intercalar os vectores já ordenados, basta utilizar o algoritmo de intercalação.

Pela descrição da solução, facilmente se percebe que a recursividade com estratégia de divisão e conquista é a melhor forma de resolver o problema. Em termos gerais, essa solução baseia-se em:

Passo1: dividir o vector em dois subvectores aproximadamente iguais;

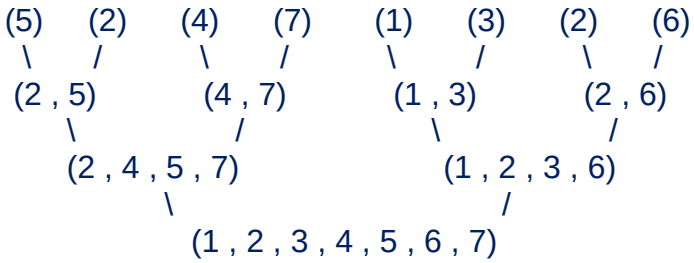
Passo2: se os subvectores não forem unitárias, cada subvector é submetido ao passo anterior. No caso contrário intercalar os seus elementos e propagar esse processo para os subvectores anteriores.

Vejamos um exemplo ilustractivo do processo de decomposição de um vector em subvectores.



Pelo diagrama anterior, observamos que o processo de divisão desse vector pode de ser obtido de forma eficiente, se escolhermos um elemento m que seja o ponto médio. Se repetirmos esse processo até a obtenção n subvectores unitários, uma parte do nosso problema estará resolvido. Agora, precisamos de descobrir um método eficiente para intercalar esses subvectores unitários.

Para termos uma noção mais precisa deste processo de intercalação, apresentamos o seguinte diagrama:



Fácilmente verifica-se que o problema de intercalação consiste em dados dois conjuntos adjacentes, T1=(t_0,t_1,… ,t_m) e T2=(t_{m+1},t_{m+2},…t_n) ordenadas na ordem crescente, intercalar os seus elementos de forma a constituir um conjunto T= (t_0,t_1,t_2,…,t_n) na mesma ordem.

A resolução deste problema é fácil, porém, não trivial. Vamos supor que temos um vector v, com dois segmentos contínuos ordenados, ou seja v = (v_0, v_1, …, v_{m-1}, v_m, … v_n) para $v_0 < v_1 <… < v_{m-1}$ e $v_m < v_{m+1} <… v_n$, mas o vector não está ordenado.

Para ordenar esse vector, temos de utilizar um vector auxiliar com o mesmo número de elementos, e na primeira chamada da função que implementa essa solução, considerar inicio = 0 , meio = (v.nElem-1/ 2 e fim = v.nElem – 1.

```
void intercalar ( TTable *vet,  int inicio, int meio, int fim)
 {
   TTable aux[max];
    int i = inicio, j= meio, k = inicio;
    while (i < meio && j < fim)
      {
          if ( vet.item[i].chave <= vet.item[j].chave )
               aux[k++] = v->item[i++];
         else
```

```
            aux[k++] = vet->item[j++];
        }
      while (i < meio) aux[k++] = vet->item[i++];
       while (j < fim) aux[k++] = vet->item[j++];
       for (i = inicio; i < fim; i++) vet->item[i] = aux[i];
       c->nElem i-1;
    }
```

Esta função uma complexidade linear para qualquer caso. Lembre-se que O(n+m) é o mesmo que O(2.n) que é o mesmo que O(n).

Agora, podemos utilizar o procedimento intercalar(), para desenvolver um algoritmo rápido e eficiente denominado por **(Merge Sort)**, proposto por John von Neumann em 1945 que ordena o vector v em ordem crescente.

Como a resolução do problema, consiste na aplicação da divisão sucessiva do intervalo ao meio, este processo gera estruturas com as mesmas propriedades. Então, a estratégia de divisão e conquista, é a técnica mais adequada para resolver o problema.

```
void mergeSort (TTable *vet, int inicio , int fim)
  {
     int meio;
     if ( inicio < fim )
       {
          meio = ( inicio + fim ) /2;
          mergeSort ( &vet , inicio , meio );
          mergeSort ( &vet , meio , fim );
          intercalar ( &vet , inicio , meio , fim );
       }
   }
```

Agora, vamos mostrar o funcionamento visual deste algoritmo. Suponhamos sem perda generalidade que v[1]= 3, v[2]= 1, v[3]= 2 e v[4]= 4. Temos:

mergeSort(1,4,v)	v = {3 1 4 2}	meio = 2
mergeSort(1,2,v)	v = {3 1}	meio = 1
mergeSort(1,1,v)	v = {3}	
mergeSort(2,2,v)	v = {1}	
Intercalar(1,1,2,v)	v = {1 3}	
mergeSort(3,4,v)	v = {4 2}	meio = 3
mergeSort(3,3,v)	v = {4}	
mergeSort(4,4,v)	v = {2}	
Intercalar(3,3,4,v)	v = {2 4}	
Intercalar(1,2,4,v)	v = {1 2 3 4 }	

À medida que o processo recursivo vai expandindo-se com as chamadas da função, uma cópia dos parâmetros e a variável local, é armazenada em memória.

O programa tem uma necessidade crescente de espaço em memória, mas à velocidade de processamento é inversamente proporcional a quantidade de memória reservada, logo o programa vai tornando-se cada vez mais lento à medida que o processamento vai sendo executado.

Vejamos em seguida o desempenho deste método. O tempo necessário para fazer o mergesort para um vector com n elementos é igual ao tempo necessário para fazer o merge de dois vectores com n/2 elementos mais o tempo necessário para se fazer a intercalação de um vector com n elementos.

Sem mostrar a fundamentação matemática, concluímos que função mergesort consome tempo proporcional a nlog2n, e isso é bem melhor do que o tempo proporcional a n^2 gasto pelos algoritmos elementares.

Na práctica, o MergeSort só é mais rápido do que os algoritmos anteriores, quando o volume de dados a ser ordenado for muito grande, nessa altura deve-se utilizar a versão iterativa.

A versão iterativa baseia-se na seguinte propriedade: em cada iteração, intercala-se dois vectores com b elementos que denominamos por blocos. O primeiro bloco intercala-se com o segundo, o terceiro com o quarto e assim por diante. A variável b assume os valores 1,2,4, …

```
void MergeSort(TTable *vet)
 {
        int inicio, meio, fim =1;
        while (fim < vet->nElem)
          {
             inicio = 0;
            while ( inicio + fim < T->nElem )
              {
                  meio = inicio + 2*fim;
                   if ( meio > T->nElem ) meio = T->nElem;
                   intercala(&vet, inicio, meio, fim );
                    inicio = inicio * 2 * meio;
                }
            fim = 2 * fim;
          }
   }
```

14.2- Ordenação por Troca e Partição

O método de ordenação por partição em três sublistas, denominado por ***Quick Sort***, é o algoritmo de ordenação interna mais rápido que se conhece. Foi inventado pelo matemático Britânico, Hoare C. A. R. em1960, quando visitava a Universidade de Moscovo e baseia-se no seguinte princípio: dado um conjunto finito com n elementos desordenado. Fragmenta-lo de forma sucessiva em três subconjuntos de tal forma que ao terminar o processo de fragmentação, o

conjunto está ordenado sem termos a necessidade de efectuar a sua intercalação.

A fragmentação do conjunto T em três subconjuntos pode ser facilmente obtida, se escolhermos um elemento t_d que pertence T, e em seguida, reorganizarmos os restantes elementos de T, de tal forma que os antecessores de t_d são menores ou iguais a t_d e os sucessores maiores ou iguais a t_d. Essa fragmentação dá-se o nome de partição de T segundo t_d.

Em termos gráficos:

$$\{ t_1,\ t_2,\ t_3,\ \ldots.\ ,\ t_n\}$$

$$/ \qquad | \qquad \backslash$$

$$\{ t_1, t_2, t_3, \ldots\ldots, t_{d-1} \} \quad \{ t_d \} \quad \{t_{d+1}, t_{d+2}, \qquad , t_n\}$$

Se repetirmos esse processo para os subconjuntos até obtermos n conjuntos unitários o problema estará resolvido.

A primeira estratégia para produzir essa partição baseia-se no seguinte princípio: seja t_1 o primeiro elemento do conjunto que denominamos por pivô. Percorrer o conjunto do início ao fim até encontrar o primeiro elemento maior do que o pivô que denominamos por t_i. Em seguida, percorrer o conjunto no sentido contrário até encontrar o primeiro elemento menor do que pivô que denominamos por t_j. Trocar os elementos. Continuar o processo até que todos os elementos tenham sido examinados. Quando o índice i for maior ou igual ao índice j todos os elementos do conjunto já foram examinados. Neste caso, o índice j aponta para o menor elemento que se encontra à direita do subconjunto formado pelos elementos menores do que o pivô, troca-se esse elemento com o pivô

Vamos ilustrar esta estratégia com um exemplo. Dado um conjunto:

T= { 5, 9, 6, 4, 3, 8, 7, 1 }

selecionámos para pivô o elemento t_1 = 5. O primeiro elemento maior do que o pivô é t_2 = 9 e o menor é t_8 = 1. Troca-se t_2 com t_8, obtendo desse modo, o conjunto

T = { 5, 1, 6, 4, 3, 8, 7 , 9 }

O segundo elemento maior é t_3 = 6 e o menor t_5= 3, troca-se t_3 com t_5 obtendo deste modo o seguinte conjunto

T = { 5, 1, 3, 4, 6, 8, 7, 9}.

O próximo elemento maior é t_5 = 6 e o menor é t_4= 4, mas 5 é maior do que 4. Então troca-se o elemento t_4 com o pivô, obtendo os subconjuntos:

{ 4, 1, 3 } { 5 } { 6, 8, 7, 9 }

Esta estratégia é descrita pela seguinte função:

```
int particao (int pri , int ult , TTable *vet)
 {
   int i = pri +1, j = ult;
  while ( i < j )
    {
        while (vet->item[i].chave < vet->item[pri].chave)  i++;
        while (vet->item[j].chave > vet->item[pri].chave)  j--;
        if ( i < j ) troca (&vet[i], &vet[j]);
    }
  troca (&vet[pri], &vet[j]);
  return j;
}
```

A segunda estratégia para produzir essa partição baseia-se no seguinte princípio: define-se dois índices. O índice i percorre o vector do início para o fim e o índice j percorre o vector no sentido contrário. Inicialmente, fixa-se o índice j ao último elemento do vector, e movimenta-se o índice i até encontrar um elemento que seja maior do que t_j. Troca-se os elementos. Em seguida, inverte-se o percurso, fixa-se o índice i e movimenta-se o índice j até encontrar um elemento menor do que t_i. Troca-se t_i com t_j. Volta-se a repetir o processo de inversão do percurso até que o índice i seja igual ao índice a j. Nessa altura todos os elementos cujos índices são menores do que i têm um conteúdo menor do que t_i e os índices cujo conteúdo são maiores do que i têm um conteúdo maior do que t_i.

Vejamos um exemplo ilustrativo. Dado o conjunto:

T = { 5, 3, 9, 7, 2, 8, 6 }

Fixa-se o índice j no último elemento do vector, j = 7 e t_7 = 6. Movimenta-se em seguida o índice i. Para i = 3, t_3 = 9, e t_3 é maior do que t_7, troca-se os elementos.

T = { 5, 3, 6, 7, 2, 8, 9 }

Inverte-se o percurso, ou seja, fixa-se o índice i, i = 3 e movimenta-se o índice j. Para j = 5, t_5 = 2, e esse valor é menor do que t_3 que é igual a 6. Troca-se os elementos.

T = { 5, 3, 2, 7, 6, 8, 9 }

Inverte-se outra vez o percurso, ou seja, fixa-se o índice j, j =5 e movimenta-se o índice i. Para i = 4, t_4 = 7, e esse valor é maior do que t_5 que é igual a 6.Troca-se os elementos

T = { 5, 3, 2, 6, 7, 8, 9 }

Inverte-se mais uma vez o percurso, ou seja, fixa-se o índice i, i = 4 e movimenta-se o índice j. Como o valor do índice j é igual ao valor do índice i, o conjunto T esta fragmentado em três subconjuntos ordenadas.

{ 5, 3, 2 } { 6 } { 7, 8, 9 }

Se continuarmos esse processo para os subconjuntos não unitários obteremos um conjunto ordenado.

Para efeito de consolidação da matéria, utilize esta estratégia para fragmentar o conjunto estudado na primeira estratégia e compare com base nessa fragmentação, qual o mais eficiente.

Esta estratégia é descrita pela seguinte função:

```
int partição (int ini , int fim TTable *vet)
{
   int i = ini , j = fim , dir =1;              /* fixa j e incrementa i */
   while (i < j)
     {
        if ( vet.item[i].chave > vet.item[j].chave )
          {
            troca(vet.item[i] , vet.item[j]);
            dir = - dir;                        /* inverte a direção */
           }
         if (dir == 1) i++; else j--;           /* incrementa i  ou decrementa j */
      }
  return i;
}
```

Como a resolução do problema, consiste na aplicação repetida do princípio de partição de T segundo d e a aplicação desse princípio, gera de estruturas com as mesmas propriedades, então, a estratégia de divisão e conquista, é a técnica mais adequada para resolver o problema. Mas, é necessário observar que primeiro ≤ ultimo é a condição necessária para existência de um subconjunto vazia ou unitária. Então, na chamada inicial, primeiro contém o índice do primeiro elemento do conjunto, neste caso 0, e ultimo o índice do último do conjunto, nesse caso nElem-1. Com esta observação, implementamos a versão básica deste método:

```
void quicksort (int primeiro, int ultimo, TTable *vet)
 {
    int pivo, d;
   if (primeiro < ultimo)
     {
       d = particao (primeiro, ultimo, vet);
       quicksort (primeiro, d-1, &vet);
       quicksort (d+1, ultimo, &vet);
     }
```

```
}
```

Para consolidar os conhecimentos, apresentamos em seguida a simulação do método para o conjunto {5,9,6,4,3,8,7,1} para a primeira estratégia de partição.

{ 5 , 9 , 6 , 4 , 3 , 8 , 7 , 1}

{4,1,3} {5} {6,8,7,9}

{3,1} {4} { } {5} {6,8,7} {9} { }

{1} {3} {} {6} {8,7}

{7} {8}

{1} {3} { } {4} { } {5} {6} {7} {8} {9} { }

Como se pode observar conjunto {1, 3, 4, 5, 6, 7, 8, 9} está ordenado.

Vejamos em seguida o desempenho do algoritmo. A função quickSort tem um esforço proporcional a $n\log_2 n$, ou seja, $0(n\log_2 n)$. Apesar de, no pior caso, o algoritmo ter um esforço proporcional a n^2, ele é considerado o melhor algoritmo de ordenação devido a sua notável eficiência média, por esse facto, este método de ordenação e a busca binária, estão disponíveis arquivo-cabeçalho **<stdlib.h>** da biblioteca ANSI

14.4- Recomendações Bibliográficas

Para o leitor aprofundar os seus conhecimentos sobre métodos de ordenação interna, recomendamos os seguintes livros:

Aho, A. V., Hopcroft, J. E., Ullman, J. D.;- *Data Structures and Algoritms*, Addison-Wesley, 1983.

Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C.;- *Algoritmos. Tradução da 2ª edição Americana Teoria e Prática*, Editora Campus, São Paulo, 2002.

Knuth D. E.;- *The Art of Computer Programming Vol 3: Sorting and Searching*, 3th Edition, Addison-Wesley, Reading, Mass, 1998.

Feofiloff P.;- *Algoritmos em linguagem C*, Editora Campus, São Paulo, Brasil, 2009.

Gonnet G. H., Baeza-Yates R.;- *Handbook of Algorithms and Data Structures*, Addison Weley, Reading, Mass. 1991

Sedgewick R. ;- *Algoritms in C*, Massachusetts, Addison-Wesley, Reading, Massachusetts,1990.

Ziviani N.;- *Projeto de Algoritmos Com implementações em Pascal e C*, 4ª Edição, Editora Pioneira, São Paulo, 1999.

14.4 - Exercícios

14.4.1-Simule detalhadamente a execução da função mergesort para o seguinte vector {3, 41, 52, 26, 38, 57, 9, 49}.

14.4.2-Um algoritmo de ordenação é estável se não altera a posição relativa dos elementos que têm um mesmo valor. Por exemplo, se o vector tiver dois elementos de valor 45, um algoritmo de ordenação estável manterá o primeiro 45 antes do segundo. A função intercalar() é estável? Se a comparação v[i] < v[j] for trocada por v[i] <= v[j] a função permanece estável?

14.4.3-O que acontece se trocarmos (p + r)/2 por (p + r - 1)/2 no código da função mergeSort ? O que acontece se trocarmos (p + r)/2 por (p + r + 1)/2 ?

14.4.4- Um vector L[p..r] está "arrumada" se existe um j tal que v[p..j-1] ≤ v[j] < v[j+1 .. r]. Escreva um algoritmo que decida se v[p..r] está arrumada. Em caso afirmativo, devolver o valor de j.

14.4.5-Discuta como a escolha do pivô pode influenciar o desempenho do QuickSort. Proponha estratégias para escolha do pivô com o objectivo de melhorar o seu desempenho.

14.4.6- Modifique o algoritmo de QuickSort de tal forma que dado um parâmetro ele executa a ordenação em ordem crescente ou decrescente.

14.4.7- Dada a função

```
int particao (int p, int u, TTable vet )
 {
   int i = p+1, j = u;
   while (1)
    {
       while ( i <= u && A[i].chave <= A[p].chave ) ++i;
       while ( vet[p].chave < vet[j].chave) --j;
       if ( i >= j ) break;
       troca ( &vet[i] , &vet[j] )
       i++;
       j--;
    }
  vet[p] = vet[j];
  vet[j] = vet[p];
  return j;
}
```

Mostre que ela devolve um j no conjunto p.. u tal que A[p..j-1] ≤ A[j] ≤ A[j+1..u].

14.4.8-Desenvolva uma função, chamada por Quick Find, baseada no algoritmo do Quick Sort para, retornar o k-ésimo menor elemento de um vector. Suponha que os elementos do conjunto S= { 7,1,3,10,17,2,21,9} estejam armazenados nessa ordem e desejamos obter o quinto maior elemento. Uma chamada QuickFind(S,0,7,5) deverá retornar o número 9, onde S é o nome do conjunto, 0 e 7 são a menor e a maior posição do conjunto e 5 indica que desejamos o quinto elemento. Obs: Não deve ordenar o conjunto depois devolver o k-ésimo elemento.

14.4.9- Implemente o método de ordenação por intercalação (MergeSort) que comtemple a situação da existência de chaves repetidas nos vectores e que nessa situação apenas copie um dos elementos repetidos para o vector ordenado.

14.4.10- Desenvolva uma versão da função quickSort para ordenar uma cadeia de caracteres em ordem lexicográfica.

14.4.11- O que acontece se trocarmos a expressão **if** (p < r) pela expressão **if** (p != r) no corpo da função quickSort ?

14.4.12- A função quickSort contém duas chamadas recursivas. Depois da chamada da separa , o sub-vector esquerdo é ordenado recursivamente e depois o sub-vector direito é ordenado recursivamente. A segunda chamada recursiva no corpo da função quickSort não é de facto necessária, ela pode ser evitada usando uma estrutura de controle iterativa. Essa técnica, chamada recursão de cauda, é fornecida automaticamente por bons compiladores. Considere a seguinte versão da ordenação por separação que simula a recursão caudal

```
void quickSort2(int p, int r, int v[ ])
 {
    while (p < r)
      {
        q = separar (p, r, v);
        quickSort2 (p, q, v);
        p = q + 1;
      }
 }
```

Ilustre a operação da função quicksort2() sobre o vetor v que contém os elementos {21, 7, 5, 11, 6, 42, 13, 2}. Use a chamada quicksort2 (0, n-1, v). Argumente que a função quicksort2 ordena corretamente o vector v.