Capítulo 13
Métodos de Ordenação Elementar

" A ciência de Computação tem tanto a ver com o computador, como a Astronomia tem com o telescópio, a Biologia com o microscópio ou a Química com os tubos d ensaio. A ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas "

- Edsgar Dijkstra -

Sumário:

13.1- Conceitos

A ordenação é o processo de organizar um conjunto de elementos segundo uma determinada ordem. Como já vimos anteriormente, se a informação estiver ordenada é possível utilizar algoritmos de pesquisa mais eficientes, como por exemplo a pesquisa binária ou a pesquisa sequencial em tabelas ordenadas. Isso mostra que a ordenação é uma tarefa muito importante no processamento de dados, e serve para facilitar o processo de pesquisa.

Ordenar, consiste em rearranjar as chaves de uma tabela numa determinada ordem. Sejam $R_0,R_1,R_2,...,R_n$ um conjunto finito de registros. Cada registro, Ri é formada por uma chave C_i e outros campos que não têm de momento qualquer importância.

A ordenação sobre esse conjunto de registros estabelece uma relação de ordem entre as suas chaves, ou seja: dada as posições no vector: $0 <1 < 2 <...< n$ a ordenação faz com que as chaves desse conjunto satisfaçam a seguinte propriedade: $C_0 < C_1 < C_2 < ... < C_n$.

Os métodos de ordenação (classificação) são utilizados para facilitar a consulta da informação. Imagine, por exemplo, como seria impossível utilizar uma lista telefónica, onde os assinantes estivessem descritos pela data de assinatura dos seus contratos, ou localizar um livro no catálogo de uma biblioteca pela data de sua aquisição.

Como exemplos de aplicações indirectas que utilizam algoritmos de pesquisa, temos: Optimizar sistemas de busca na Internet, manutenção de estruturas de bases de dados, etc.

Um método de ordenação é **estável** se ele preserva a ordem original dos registros com o mesmo valor de chave. Por exemplo, numa lista do tipo alfabético, se os nomes dos funcionários fossem ordenados pelo campo salário, um método estável produziria uma listagem onde os funcionários com o mesmo salário apareceriam na mesma ordem.

Os métodos de ordenação são classificados em dois grandes grupos: internos (***Sort Array***) quando a tabela a ser ordenada for pequena e couber na memória principal do computador. Neste caso, os registos são guardados numa estrutura de dados do tipo vector e, externos (***Sort File***) quando a tabela a ser ordenada for muito grande e não couber na memória do computador. Nesse cado os registros são armazenados em ficheiros em memória externa.

Os algoritmos de ordenação cujo processo de ordenação é feito na própria tabela (vector), enquadram-se numa das seguintes categorias: ordenação por selec-ção, ordenação por troca, e ordenação por inserção.

13.2- Estrutura de Dados

Vamos centrar o nosso estudo num vector de estruturas, denominado por vet, com MAX elementos. Cada elemento desse vector é uma estrutura (registro) que possui uma chave do tipo inteiro e outros campos que de momento não têm qualquer importância. Associado a esse vector temos uma variável, denominada por nElem, que dar-nos-á a qualquer instante o número de elementos inseridos.

13.3- Troca de dois elementos

Nos algoritmos de ordenação, a operação de troca entre dois elementos é executada com muita frequência. Vejamos algumas formas de realizar essa operação.

A primeira forma, a mais usual é desenvolver a seguinte função:

```
void troca (int * x , int * y)
{
  int aux = *x;
      *x = *y;
      *y = aux;
}
```

Para trocar o elemento vet[i] com o elemento vet[j] , basta invocar a função anterior, com a seguinte sintaxe:

```
troca ( &vet[i] , &vet[j] ).
```

A segunda forma para realizar esta operação, consiste em desenvolver a seguinte função:

```
void troca ( int *x[ ], int k1, int k2 )
 {
    int aux = *x[k1];
     *x[k1] = x[k2];
     *x[k2] = aux;
 }
```

Agora para trocar o elemento vet[i] com o elemento vet[j], basta invocar a função anterior com a seguinte sintaxe:

```
troca (vet , i , j ).
```

Para terminar, a maneira mais eficiente de realizar esta operação, consiste em utilizar as três atribuições no corpo do programa, sem usar uma função. Mas para tornar mais claro o código da aplicação, devemos utilizar uma macro com o comando **#define**:

```
#define troca (x, y) { int aux = x; x = y; y = aux; }
```

A utilização dessa macro será feita com a instrução:

```
troca (vet[i] , vet[j] );
```

13.4 - Ordenação Por Selecção

Os métodos de ordenação por seleção utilizam a pesquisa sequencial. Eles executam ciclos sucessivos sobre todos os elementos da tabela, e em cada ciclo, selecionam o elemento de menor valor de todos os elementos analisados, caso de se pretender ordenar em ordem crescente, ou o maior elemento, caso de se pretender ordenar em ordem decrescente. Em cada ciclo, um elemento entre os restantes elementos da tabela é colocado no lugar correcto.

Fazem parte desta estratégia, os métodos de ordenação sequencial (**Sequential Sort**) e o método de ordenação por seleção (**Selection Sort**).

O método de ordenação sequencial (**Sequential Sort)**) caracteriza-se por: numa primeira fase, comparamos o primeiro elemento com o segundo e procedemos a sua troca, se o primeiro for maior do que o segundo. Em seguida, executamos essa operação com o primeiro e o terceiro, depois com o primeiro com o quarto, e assim por diante. No fim dessa fase teremos o elemento de menor valor na primeira posição do vector. Na segunda fase, comparamos o segundo elemento com o terceiro e procedemos a sua troca se o segundo for maior do que o terceiro. Em seguida executamos essa operação com o segundo com o quarto e assim por diante. No fim dessa fase teremos o segundo elemento de menor

valor na segunda posição do vector. Repete-se essas fases para o terceiro, quarto e penúltimo.

```
void sequentialSort (TTable *vet )
 {
   int i, j;
   for ( i = 0 ; i < vet->nElem -1 ; i++)
      for ( j = i+1 ; j < vet->nElem ; j++)
         if ( vet->item[i].chave > vet->item[j].chave) troca (&vet[i] , &vet [j]);
 }
```

Para consolidar a matéria, faça uma simulação para deste algoritmo para o seguinte conjunto de dados:

59, 46, 32, 81, 46, 55, 87, 43, 70, 80

Qual é o esforço do algoritmo? Suponhamos sem perda da generalidade que o vector possui n elementos. Na primeira fase, comparamos o primeiro elemento com os n-1 restantes. Na segunda fase, comparamos o segundo elemento com os n-2 elementos. Na (n-1)-ésima, fase, comparamos o penúltimo elemento com o último. Logo o total de comparações que o algoritmo realiza é expresso pela fórmula:

$$C(n) = (n-1) + (n-2) + \ldots + 1$$
$$= n (n - 1)/2$$
$$= (n^2 – n)/2$$

Isso nos permite concluir que este algoritmo tem um esforço proporcional a n^2. Logo, o algoritmo tem uma complexidade computacional quadrática.

Agora, vamos analisar o número de trocas que o algoritmo realiza. Esse número depende do grau de desorganização dos elementos da tabela, no pior caso, o algoritmo executa:

$$T(n) = (n^2 – n)/2$$

A principal desvantagem desde algoritmo, reside no facto deste realizar muitas trocas desnecessárias. Uma das formas de reduzir o número de trocas em cada fase, consiste em encontrar o menor elemento e armazena-lo numa variável auxiliar. Depois de comparar esse elemento com os restantes proceder a sua troca. Esta é a essência do algoritmo que estudaremos a seguir.

O método de ordenação por selecção (**Selection Sort**) caracteriza-se por: selecionamos numa primeira fase, o primeiro elemento do vector e consideramos que esse elemento é o menor. Percorremos em seguida os restantes elementos e comparamos os seus conteúdos com o elemento que assumimos como menor. Se encontrarmos algum elemento cujo conteúdo é menor do que o elemento que assumimos como menor, efectuamos a sua troca. Com essa operação, temos todas às garantias, que o elemento que está na primeira posição é o menor.

Selecionamos, numa segunda fase, o segundo elemento e consideramos que esse elemento é o menor. Percorremos os restantes elementos do vector e comparamos os seus conteúdos com o elemento que assumimos como menor. Se encontrarmos algum elemento cujo conteúdo é menor do que o elemento que assumimos como menor, efectuamos a sua troca. Com essa operação os dois primeiros elementos do vector estão ordenados. Repetimos este processo, até que todos os elementos tenham sido ordenados.

```
void selectionSort (TTable *vet )
{
   int i , j , idMim;
   for ( i = 0 ; i < vet->nElem-1 ; i++)
     {
        idMim = i;
        for ( j = i+1 ; j < vet->nElem ; j++ )
          if ( vet->item[idMim].chave > vet->item[j].chave)  IdMim = j;
            if (idMim != i)  troca (&vet[i] , &vet[idMim]);
     }
 }
```

Apresentamos em seguida a simulação deste algoritmo para o seguinte conjunto de dados

59	46	32	81	46	55	87	43	70	80	
32	46	59	81	46	55	87	43	70	80	Troca (59 , 32)
32	43	59	81	46	55	87	46	70	80	Troca (46 , 43)
32	43	46	81	59	55	87	46	70	80	Troca (49 , 46)
32	43	46	46	59	55	87	81	70	80	Troca (81, 46)
32	43	46	46	55	59	87	81	70	80	Troca (59 , 55)
32	43	46	46	55	59	87	81	70	80	
32	43	46	46	55	59	70	81	87	80	Troca (87 , 70)
32	43	46	46	55	59	70	80	87	81	Troca (81 , 80)
32	43	46	46	55	59	70	80	81	87	Troca (87 , 81)

Com base nessa simulação, podemos concluir que, após o término de cada ciclo externo (fase), as seguintes condições são verdadeiras:

o subvector vet[0...i-1] está ordenado; e
o subvector vet[i..nElem-1] é uma permutação do vector original; e
vet[i-1] < vet[j] para qualquer j = i, i+1,….,nElem-1

Uma análise semelhante a que fizemos ao algoritmo sequêncial, mostra que o algoritmo de selecção realiza o mesmo número de comparações, ou seja:

$$C(n) = (n^2 - n)/2$$

Mas, o número de trocas necessárias para ordenar um vector com n elementos é no máximo:

$$T(n) = n - 1$$

Isso mostra que o seu esforço proporcional a n^2. Logo, a sua complexidade computacional quadrática. Contudo, o algoritmo de selecção é mais eficiente do que o algoritmo de ordenação sequencial.

13.5 - Ordenação Por Troca

Ao contrário dos algoritmos de selecção que encontram o elemento com menor valor e executam a sua troca, os algoritmos por troca, efectuam várias passagens pelo vector, e em cada passagem, comparam os elementos que se encontram à uma distância fixa. Se durante uma determinada passagem, não foi realizada qualquer troca, então os elementos que se encontram à distancia de comparação já foram trocados, e o processo de ordenação termina.

Fazem parte desta estratégia, o método de ordenação por bolha (**Bubble Sort**), e o método de ordenação de Shaker (**Shaker Sort**).

O método de ordenação por bolha (***Bubble Sort***) caracteriza-se pela comparação e troca de elementos consecutivos. Em termos mais concretos: percorrer o vector de forma sequencial várias vezes. Cada passagem (fase) consiste em comparar cada elemento com o seu sucessor e trocar os seus conteúdos se não estiverem na ordem desejada.

```
void  bubbleSort (TTable  *vet )
 {
      int i, j;
      for  (i = 1 ; i < vet->nElem ; i++)
        for (  j = i ;  j > 0 && vet->item.chave[j] < vet->item.chave[j-1] ; j--)
              troca ( vet[j] , vet[j-1] ) ;
   }
```

Vamos apresentar em seguida a simulação compactada deste algoritmo. Observe que os números a vermelho estão fora do padrão de ordenação e como consequência, sobem na estrutura até atingirem o lugar correcto.

59	46	32	81	46	55	87	43	70	80
46	32	59	46	55	81	43	70	80	87
32	46	46	55	59	43	70	80	81	87
32	46	46	55	43	59	70	80	81	87
32	46	46	43	55	59	70	80	81	87
32	46	43	46	55	59	70	80	81	87
32	43	46	46	55	59	70	80	81	87

Qual é o esforço do algoritmo? Para uma sequencia de n elementos, o número de comparações é análogo ao número de comparações do algoritmo de selecção.

$$C(n) = (n^2 - n)/2$$

A grande desvantagem deste método reside no número de trocas que é muito superior ao realizado pelo algoritmo de selecção. O número médio de trocas é expresso por:

$$T(n) = (n^2 - n)/4$$

Isso nos permite concluir que o algoritmo de bublle não é mais eficiente do que o algoritmo de selecção e do que o algoritmo sequencial.

Como o processo de ordenação deve terminar quando não houver mais trocas, podemos utilizar este facto, para melhorar a eficiência deste algoritmo. Para isso, basta utilizar uma variável do tipo inteira, com um funcionamento booleano, que deverá indicar se num determinado passo não foi efectuada nenhuma troca. Se essa condição for verdadeira, encerrar o processo de ordenação. Essa alteração é descrita pela seguinte função:

```
void  bubbleSort (TTable  *vet )
 {
    int i, j , trocou;
    for  (i = 1 ; i < vet->nElem ; i++)
     {
        trocou = 0;
        for (  j = i ;  j > 0 && vet->item.chave[j] < vet->item.chave[j-1] ; j--)
              troca ( vet[j] , vet[j-1] ) ;
        if (trocou == 0) break;
     }
  }
```

Numa analise superficial, parece que esta alteração proporciona um ganho substancial no desempenho do algoritmo, mas, na verdade, o número de trocas contínua a ser o mesmo. A única vantagem que obtivemos foi a eliminação do número de comparações redundantes.

Em termos gerais, podemos dizer que o algoritmo de bubble melhorado, apresenta poucas vantagens em relação a versão original.

13.6 - Ordenação Por Inserção

A ordenação por inserção é uma técnica de ordenação, que normalmente é utilizada para baralhar cartas. Ela consiste em determinar, para cada elemento da tabela, a posição de inserção correcta para que ele fique ordenado e inserir esse elemento nessa posição.

Fazem parte desta estratégia, o método de ordenação por inserção (**Insertion Sort**), e o método de ordenação por Shell (**Shell Sort**) na versão original.

O método de ordenação por inserção **(Insertion Sort)** caracteriza-se por: na primeira fase, compara-se os dois primeiros elementos e procede-se a troca se o primeiro for menor do que o segundo. Com isso temos a garantia que os dois primeiros elementos estão na ordem crescente. Na segunda fase, compara-se o terceiro elemento com o segundo e procede-se a troca se o terceiro for menor do que o segundo. Se houver trocas compara-se o segundo elemento com o primeiro e procede-se a troca se o segundo for menor do que o primeiro. Após a segunda fase, temos a garantia que os três primeiros elementos estão na ordem decrescente. Repetimos esse processo para os restantes elementos. Na última fase, compara-se o último elemento com o penúltimo e procede-se a troca se o último for menor do que o penúltimo. Se houver trocas, compara-se o penúltimo com o antepenúltimo e procede-se a troca se o penúltimo for menor do que o antepenúltimo. Repetimos este processo até que não haja mais trocas.

```
void insertionSort (TTable *vet )
 {
   int j , i ;
   TItem x;
   for ( j = 1 ; j < vet->nElem ; j++)
    {
      x = vet->item[j];
      for ( i = j-1 ; i >= 0 && vet->item[i].chave > x.chave ; j--)
         vet->item[i+1] = vet->item[i];
      vet->item[i+1] =  x;
    }
 }
```

Apresentamos em seguida a simulação deste algoritmo para o seguinte conjunto de dados

59	46	32	81	46	55	87	43	70	80
46	59	32	81	46	55	87	43	70	80
32	46	59	81	46	55	87	43	70	80
32	46	59	81	46	55	87	43	70	80
32	46	59	81	46	55	87	43	70	80
32	46	46	59	81	55	87	43	70	80
32	46	46	55	59	81	87	43	70	80
32	43	46	46	55	59	81	87	70	80
32	43	46	46	55	59	70	81	87	80
32	43	46	46	55	59	70	80	81	87

Com base nessa simulação, podemos concluir que, após o término de cada fase, as seguintes condições são verdadeiras:

O subvector vet[0..j-1] está ordenado; e
O subvector vet[j..nElem-1] é uma permutação do vector original.

Agora, vamos analisar o esforço deste algoritmo. Se o vector estiver inicialmente em ordem crescente, o número de comparações necessárias para ordena-lo é

igual a n-1. Mas se a vector está em ordem decrescente, o número de comparações necessárias para ordena-lo é igual (n^2 – n)/2. Portanto o esforço do algoritmo de inserção é aproximadamente igual à.

$$C(n) = (n^2-n)/2$$

Mas o número médio de trocas realizadas é expresso por:

$$T(n) = n-1$$

Logo, podemos concluir, que o algoritmo de inserção é tão eficiente quanto o algoritmo de selecção.

Como os algoritmos que estudamos têm uma eficiência proporcional a n^2, dizemos que esses algoritmos pertencem a classe $0(n^2)$.

13.7 – Referências 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.

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.

Wirt N;- *Algoritms and Data Structures*, oberon version: August 2004, disponível em: www.ethoberon.etz.ch/withPubli/AD.pdf

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

13.8 – Exercícios

13.8.1- Elabore uma versão recursiva para o método de ordenação por inserção direta.

13.8.2- Elabore uma versão recursiva para o método de ordenação sequencial.

13.8.3- Dado o conjunto T = {11, 6, 10, 12, 7, 5, 8, 3, 4, 1, 2}. Utilize esses dados para simular o funcionamento dos métodos de inserção e Selecção. Com base nessa simulação, diga qual o mais rápido.

13.8.4- Dado o conjunto T = {9, 25,10,18, 5,7,15, 3}. Utilize esses dados para mostrar o funcionamento do método BubbleSort.

13.8.5- Explique como seria possível melhorar o método BubbleSort, se armazenarmos a informação de troca e as posições onde ocorreram as trocas. Implemente essa modificação.

13.8.6- No método InsertionSort, a cada passo, o menor elemento é procurado para que seja inserido na sequência já ordenada. Essa procura pode ser feita sequencialmente ou por busca binária. Implemente essas modificações.

13.8.7- Dos algoritmos estudados qual é estável? Utilize os seguintes conjuntos de dados para justificar a sua afirmação:

T1 = { 8, 9, 7,9,3,2,3,8,4,6}

T2 = { 89, 79, 32, 38, 46, 26, 43, 38, 32, 79 }

T3= { 3, 19, 25, 24, 1, 8, 10, 7, 19, 12, 10 }

T4= { 2,4,6,8,10,12,11,9,4,5,3,1}

13.8.8- Quando a lista a ser ordenada possui muitos campos, o esforço computacional para a troca de registos é muito grande, tornando os métodos de ordenação inviáveis. Para esses casos, podemos desenvolver um algoritmo em duas etapas: A primeira etapa, consiste em definir uma lista do tipo inteiro com a mesma dimensão da lista que se pretende ordenar, denominada por índex "vector de índices". Inicialmente, definimos índex[i] = i para todos os elementos da lista que se pretende ordenar. Com essa operação, índex contêm as posições relativas dos registos da lista que se pretende ordenar. O processo de ordenação consiste em alterar os campos do vector de índices de tal forma que ao terminar o processo a lista ficará ordenada quando percorremos o vector de índices. Esse método dá-se o nome de vector indirecto de ordenação. Desenvolva um procedimento para esse método.

13.8.9- Dada o conjunto T = { 11, 9, 7, 10, 8, 6, 4, 3, 5, 1, 2}. Com base numa simulação, compare os métodos de ordenação de bublle e Shell.

13.8.10- Dado o segmento de código:

```
int i, indInicial = 1, indFinal = ultPos, uTroca = ultPos -1;
do {
        for ( i = indFinal; i >= IndInicial; i--)
          if ( A.item[i -1].chave > A.item[i],chave )
            {
              troca (&A[i] , &A[i -1]);
               uTroca = i;
            }
          indInicial = uTroca + 1;

       for ( i = indInicial; i <= IndFinal; i++)
         if ( A.item[i -1].chave > A.item[i].chave )
          {
             troca(&A[i] , &A[i -1]);
             uTroca = ind;
          }
       indFinal = uTroca - 1;
    }
while (indInicial < indFinal);
```

seja T = {9, 25,10,18, 5,7,15, 3} um conjunto de dados. Mostre o funcionamento deste segmento de código com esses dados que é denominado por ShakerSort.

13.8.11-Explique como seria possível melhorar o método BubbleSort, armazenando não apenas a informação da troca, mas também a posição na lista onde ocorreu a troca. Implemente essa modificação.