



UNIÃO DE TECNOLOGIA E ESCOLAS DE SANTA CATARINA
Tecnologia em Processamento de Dados

MÉTODOS DE ORDENAÇÃO

Tecnologia em Processamento de Dados
(Pascal)

Jurandir Steffens
Acadêmico

Glauco Vinicius Scheffel
Professor

Joinville, Novembro de 1998.

INTRODUÇÃO

Algumas tarefas de programação são tão comuns, que, com o passar dos anos, foram desenvolvidos algoritmos altamente eficientes e padronizados para se encarregar dessas tarefas. Ordenação é uma das mais comuns e, esta pesquisa descreve alguns algoritmos e mostra como eles são utilizados no Turbo Pascal.

Métodos de Ordenação

Embora tenham sido desenvolvidos vários algoritmos de ordenação com o passar dos anos, três deles são usados com maior frequência – a ordenação por troca, a ordenação shell e a ordenação por segmentação.

A ordenação por troca é fácil de gravar, mas terrivelmente lenta. A ordenação shell é relativamente rápida, mas faz uso excessivo dos recursos de memória. A ordenação por segmentação, a mais rápida das três, exige extenso espaço da área de armazenamento para as chamadas recursivas. Conhecer todos os três algoritmos e perceber por que um é melhor que o outro é importante e ilustra as sutilezas da boa programação.

Princípios Gerais da Ordenação

Os algoritmos de ordenação apresentados nesta seção comparam um elemento de um array com outro, e, se os dois elementos estiverem desordenados, os algoritmos trocam a ordem deles dentro do array. Este processo é ilustrado pelo seguinte segmento de código:

```
If  a[i] > a[i+1] Then Begin  
    temp := a[i];  
    a[i] := a[i+1];  
    a[i+1] := temp;  
End;
```

A primeira linha do código verifica se os dois elementos do array estão desordenados. Em geral, os arrays encontram-se em ordem quando o elemento atual é menor que o elemento seguinte. Se os elementos não estiverem adequadamente ordenados, ou seja, se o elemento atual for maior que o elemento seguinte, sua ordem será trocada. A troca exige uma variável de armazenamento temporário do mesmo tipo que a dos elementos do array que está sendo ordenado.

A diferença principal entre os três algoritmos de ordenação é o método pelo qual os elementos do array são selecionados para comparação. O método de comparação tem grande impacto sobre a eficiência da ordenação. Por exemplo, uma ordenação por troca, que compara apenas elementos adjacentes do array, pode exigir meio milhão de comparações para ordenar um array, enquanto a ordenação por segmentação exige somente três ou quatro mil comparações.

Ordenação por Troca

Para os programadores de computador, existem os bons métodos, existem os métodos ruins, e os programas caseiros. Estes últimos em geral funcionam, mas lenta e ineficientemente. A ordenação por troca é um bom exemplo de um programa caseiro – com tempo suficiente, ele ordenará seus dados, mas você pode ter que esperar um dia ou dois.

O algoritmo da ordenação por troca é simples – ele começa no final do array a ser ordenado e executa a ordenação na direção do início do array. A rotina compara cada elemento com seu precedente. Se os elementos estiverem desordenados, serão trocados. A rotina continua até alcançar o início do arquivo.

Como a ordenação funciona de trás para frente através do array, comparando cada par de elementos adjacentes, o elemento de menor valor sempre “flutuará” no topo depois da primeira passagem. Após a segunda passagem, o segundo elemento de menor valor “flutuará” para a segunda posição do array, e assim por diante, até que o algoritmo tenha verificado uma vez cada elemento do array.

O código a seguir mostra esse processo no Turbo Pascal:

```
For i := 2 to n Do
  For j := n DownTo i Do
    If a[j-1] > a[j] Then
      Switch(a[j], a[j-1]);
```

Como você pode perceber, o algoritmo da ordenação por troca é compacto. Na verdade, ele é uma única instrução do Turbo Pascal. A ordenação por troca recebe duas entradas – **a**, o array a ser ordenado, e **n**, a quantidade de elementos do array. O loop interno, controlado pela instrução

```
For j := n DownTo i Do
```

efetua todas as comparações em cada verificação do array. O loop externo, controlado pela instrução

```
For i := 2 to n Do
```

determina a quantidade de passagens a serem executadas. Observe que **j** promove a execução do loop a partir do final do array (**n**) até **i** e que **i** diminui após cada passagem. Dessa forma, cada verificação do array se torna menor à medida que a ordenação por troca é executada.

Um exemplo de como a ordenação por troca funciona é apresentado na **Figura 11.2**. Um array de 10 inteiros é ordenado de modo crescente. Os elementos de array são listados no final de cada passagem. Uma passagem consiste em uma execução completa do loop **For-Do** interno.

A ordem do array original é apresentada na linha identifica como início. Os valores variam de 0 a 92, e são distribuídos aleatoriamente em todo o array. A primeira passagem pelo array posiciona o valor mais baixo (0) na primeira posição do array, e o número 91 é deslocado da primeira posição para a segunda. Os outros elementos ainda permaneceram relativamente desordenados.

| Passagem | Posição no Array | | | | | | | | | |
|----------------|------------------|----|----|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Início: | 91 | 6 | 59 | 0 | 75 | 0 | 48 | 92 | 30 | 83 |
| 1 | 0 | 91 | 6 | 59 | 0 | 75 | 30 | 48 | 92 | 83 |
| 2 | 0 | 0 | 91 | 6 | 59 | 30 | 75 | 48 | 83 | 92 |
| 3 | 0 | 0 | 6 | 91 | 30 | 59 | 48 | 75 | 83 | 92 |
| 4 | 0 | 0 | 6 | 30 | 91 | 48 | 59 | 75 | 83 | 92 |
| 5 | 0 | 0 | 6 | 30 | 48 | 91 | 59 | 75 | 83 | 92 |
| 6 | 0 | 0 | 6 | 30 | 48 | 59 | 91 | 75 | 83 | 92 |
| 7 | 0 | 0 | 6 | 30 | 48 | 59 | 75 | 91 | 83 | 92 |
| 8 | 0 | 0 | 6 | 30 | 48 | 59 | 75 | 83 | 91 | 92 |
| 9 | 0 | 0 | 6 | 30 | 48 | 59 | 75 | 83 | 91 | 92 |

Figura 11.2 – Ordenar um array de inteiros com o algoritmo de ordenação por troca.

A cada passo da ordenação por troca, o próximo número mais baixo assume o lugar apropriado dentro do array, e os números maiores são deslocados para a direita. No final da oitava passagem, o array estará inteiramente ordenado, ainda que a ordenação continue a fazer uma passagem pelo array.

O programa a seguir contém o algoritmo de ordenação por troca, que considera como parâmetros um array de inteiros e a quantidade de elementos desse array.

```

Program BubbleTest;
Type Int_Arr = Array [1..10] Of Integer;
Var i : Integer;

```

```

    a : Int_Arr;

    (*****)

    Procedure Bubble(Var a : Int_Arr;
                    n : Integer);
    Var i,j : Integer;

    (*****)

    Procedure Switch(Var a,b : Integer);
    Var c := Integer;
    Begin
        c := a;
        a := b;
        b := c;
    End;

    (*****)

    Begin
        For i := 2 to n Do
            For j := n DownTo i Do
                If a[j-1] > a[j] Then
                    Switch(a[j],a[j-1]);
            End;
        End;

        (*****)

    Begin
        a[1] := 91;
        a[2] := 06;
        a[3] := 59;
        a[4] := 00;
        a[5] := 75;
        a[6] := 00;
        a[7] := 48;
        a[8] := 92;
        a[9] := 30;
        a[10] := 83;

        For i := 1 To 10 Do
            Write(a[i]:4);
        WriteLn;

        Bubble(a,10);

        For i := 1 To 10 Do
            Write(a[i]:4);
        WriteLn;
    End;

```

```
Write('Pressione ENTER...');  
ReadLn;  
End.
```

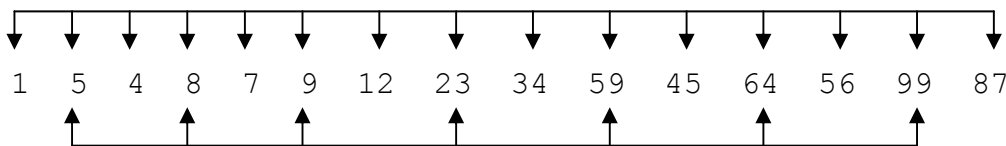
O programa começa atribuindo valores aleatórios ao array **a**, e mostra os valores na tela. A rotina **Bubble** ordena o array. Ao terminar a ordenação, o array é novamente apresentado.

O ponto fraco da ordenação por troca é que ela compara apenas elementos adjacentes do array. Se o algoritmo de ordenação comparasse primeiro elementos separados por um amplo intervalo e, depois, se concentrasse nos intervalos progressivamente menores, o processo seria mais eficiente. Essa seqüência de idéias levou ao desenvolvimento dos algoritmos de ordenação shell e da ordenação por segmentação.

Ordenação Shell

A ordenação shell é bem mais eficiente que a ordenação por troca. Primeiro, ela dispõe os elementos aproximadamente onde ficarão na ordenação final e determina, em seguida, seu exato posicionamento. O valor do algoritmo está no método que ele utiliza para fazer uma estimativa da posição final aproximada de um elemento.

O conceito-chave da ordenação shell é o intervalo (gap), que representa a distância entre os elementos comparados. Se o intervalo for 5, o primeiro elemento será comparado com o sexto elemento, o segundo com o sétimo, e assim por diante. Em uma única passagem pelo array, todos os elementos contidos dentro do intervalo serão ordenados. Por exemplo, os elementos do array a seguir encontram-se ordenados, considerando-se um intervalo de 2.



Como você pode perceber, o array já está praticamente ordenado antes de o algoritmo verificar os elementos adjacentes. Na próxima passagem por esse array, o intervalo será reduzido para 1, que resultará em um array inteiramente ordenado. O valor inicial do intervalo é arbitrário, embora seja comum defini-lo com a metade da quantidade de elementos do array (ou seja, $n \text{ div } 2$).

As muitas versões da ordenação shell variam em complexidade e eficiência. A versão apresentada neste capítulo é extremamente eficiente, exigindo apenas algumas passagens para concluir a ordenação.

Infelizmente, não existe uma maneira simples de descrever como funciona esse algoritmo de ordenação shell. Os algoritmos eficientes tendem a ser mais complexos do que ineficientes, e são, portanto, mais difíceis de serem expressos em palavras. É por isso que os algoritmos pobres são utilizados com tanta frequência. A **Figura 11.3** contém o código fundamental para o algoritmo de ordenação shell. Reveja esse código ao ler a explicação.

A primeira linha da rotina define o intervalo como $n \text{ div } 2$. O loop externo da ordenação shell, controlado pela instrução

While (gap > 0) Do

determina a quantidade de passagens feitas pelo array. Após cada passagem pelo array, o intervalo é reduzido à metade, em cada passagem, até que o intervalo chegue a 0. Por exemplo, se existissem dez elementos no array, o primeiro intervalo seria 5, seguido por 2, 1 e 0. Como é usada a divisão de inteiros, $1 \text{ div } 2$ resulta em 0.

Em cada passagem, três variáveis determinam os elementos que serão comparados – **i**, **j** e **k**. A variável **i** aponta para o elemento distante, a **j** aponta para o elemento próximo. Por exemplo, se o intervalo for 5, **i** será igual a 6 e **j** será igual a 1. Antes da comparação, **k** é igual a **j + gap**, que, para a primeira comparação, é igual a **i**.

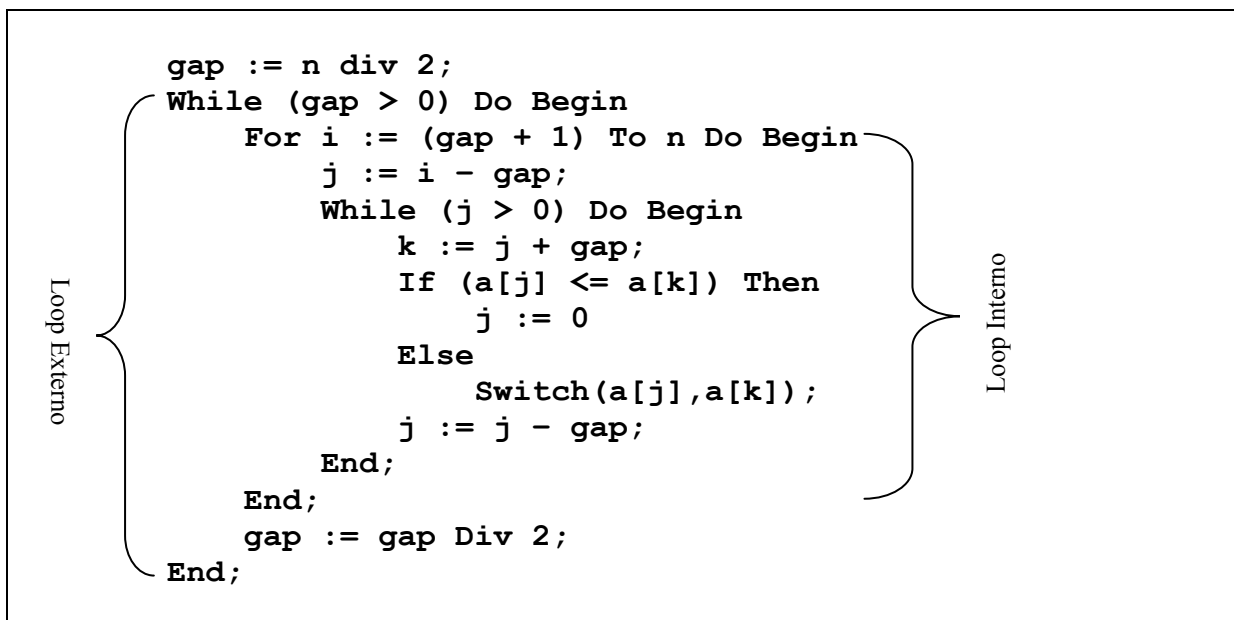


Figura 11.3 – Os principais loops do algoritmo de ordenação shell.

A comparação usa **k** em vez de **i** porque pode ser necessário repassar o array. Para repassar, o intervalo é subtraído de **j** e **k** também é alterado para que um novo par de elementos seja comparado. Como **i** controla o loop interno, não deve ser alterado nesse processo de repassagem.

Considere o exemplo mostrado na **Figura 11.4**, onde está sendo ordenado um array de dez elementos. No passo 1, **gap** é igual a 2, **j** é igual a 6, e **k** é igual a 8. Dessa forma, o sexto e o oitavo elementos do array serão comparados. Como o elemento 6 é 85 e o elemento 8 é 49, os dois devem ser trocados, como é mostrado no passo 2.

A seguir, o algoritmo define j como igual a $j - \text{gap}$. Nesse caso, 4. Como j é maior que zero, o loop interno é novamente executado. Como 2 foi subtraído de j , o quarto e sexto elementos são comparados. Novamente, os elementos estão desordenados e precisam ser trocados. Como antes, j está definido como $j - \text{gap}$, ou 2, levando ao passo 3.

Os elementos 2 e 4 do array encontram-se na ordem correta. Assim, em vez de trocar os elementos, o programa define j como zero. Agora, o resultado de $j - \text{gap}$ é o número 2 negativo. Como esse número é menor que zero, o loop interno termina e i é incrementado. Continuando a tarefa, j é definido como $i - \text{gap}$, ou 7, e k é definido como igual a $j + \text{gap}$, ou 9.

| Passo | Posição no Array | | | | | | | | | |
|-------|------------------|----------|----|----------|----|----------|----------|----------|----------|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1 | 19 | 9 | 32 | 63 | 86 | 85 | 87 | 49 | 35 | 86 |
| | | | | | | ↑ | | ↑ | | |
| | | | | | | j | | k | | |
| 2 | 19 | 9 | 32 | 63 | 86 | 49 | 87 | 85 | 35 | 86 |
| | | | | ↑ | | ↑ | | | | |
| | | | | j | | k | | | | |
| 3 | 19 | 9 | 32 | 49 | 86 | 63 | 87 | 85 | 35 | 86 |
| | | ↑ | | ↑ | | | | | | |
| | | j | | k | | | | | | |
| 4 | 19 | 9 | 32 | 49 | 86 | 63 | 87 | 85 | 35 | 86 |
| | | | | | | | ↑ | | ↑ | |
| | | | | | | | j | | k | |

Figura 11.4 – Ordenar um array de inteiros com o algoritmo de ordenação shell.

Em resumo, i rastreia o fluxo global do algoritmo, enquanto k faz uma repassagem, quando necessário. Esse pequeno truque de lógica aumenta a eficiência em cerca de 300% sobre a ordenação shell mais simples.

O exemplo de programa a seguir apresenta a rotina **Shell**, que contém o algoritmo da ordenação shell.

```

Program ShellTest;
Uses CRT;
Type Int_Arr = Array [1..10] Of Integer;
Var i : Integer;
    a : Int_Arr;

(*****)

Procedure Shell(Var a : Int_arr;

```

```

                n : Integer);
Var gap,i,j,k,x : Integer;

(*****)

Procedure Switch(var a,b : Integer);
Var c : Integer;
Begin
    c := a;
    a := b;
    b := c;
End;

(*****)

Begin
    gap := n div 2;
    While (gap > 0) Do Begin
        For i := (gap + 1) To n Do Begin
            j := i - gap;
            While (j > 0) Do Begin
                k := j + gap;
                If (a[j] <= a[k]) Then
                    j := 0
                Else
                    Switch(a[j],a[k]);
                j := j - gap;
            End;
        End;
        gap := gap Div 2;
    End;
End;

(*****)

Begin
    ClrScr;
    a[1] := 19;
    a[2] := 09;
    a[3] := 32;
    a[4] := 63;
    a[5] := 86;
    a[6] := 85;
    a[7] := 87;
    a[8] := 49;
    a[9] := 35;
    a[10] := 86;

    For i := 1 To 10 Do
        Write(a[i]:4);

```

```
WriteLn;

Shell(a,10);

For i := 1 To 10 Do
    Write(a[i]:4);
WriteLn;
Write('Pressione ENTER...');
ReadLn;
End.
```

A rotina **Shell** aceita o array a ser ordenado, incluído a quantidade de elementos do array, e depois retorna em sua forma ordenada. Se você comparar esse programa com a ordenação por troca considerando um array de 1.000 elementos, encontrará uma sensível diferença no tempo exigido para ordenar o array. A ordenação por segmentação é duas ou três vezes mais eficiente que a ordenação shell.

Ordenação por Segmentação (Quick)

A rainha de todos os algoritmos de ordenação é a ordenação por segmentação. Esse algoritmo é amplamente aceito como o método mais rápido de ordenação de propósito geral disponível.

Um dos aspectos interessantes da ordenação por segmentação é que ela ordena as coisas quase da mesma maneira como fazem as pessoas. Primeiro, ela cria grandes “pilhas”, e depois, as ordena em pilhas cada vez menores, terminando finalmente com um array inteiramente ordenado.

O algoritmo de ordenação por segmentação começa estimando um valor de intervalo médio para o array. Se o array for composto de números de 1 a 10, o ponto médio poderia ser 5 ou 6. O valor exato do ponto médio não é fundamental. O algoritmo vai trabalhar com um ponto médio de qualquer valor. Entretanto, quanto mais próximo o ponto médio estimado estiver do ponto médio real do array, mais rápida será a ordenação.

A rotina calcula um ponto médio considerando o primeiro e o último elemento da parte do array que está sendo ordenada. Depois que a rotina seleciona um ponto médio, ela coloca todos os elementos menores que o ponto médio na parte inferior do array, e todos os elementos maiores na parte superior. Observe a ilustração da **Figura 11.5**.

| Ponto Médio | Posição no Array | | | | | | | | | |
|---------------------|------------------|-----|----|----|----|----|----|----|----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Passo 1: 55 | 86 | 3 | 10 | 23 | 12 | 67 | 59 | 47 | 31 | 24 |
| Passo 2: 35 | 24 | 3 | 10 | 23 | 12 | 31 | 47 | 59 | 67 | 86 |
| Passo 3: 27 | 24 | 3 | 10 | 23 | 12 | 31 | 47 | 59 | 67 | 86 |
| Passo 4: 18 | 24 | 3 | 10 | 23 | 12 | 31 | 47 | 59 | 67 | 86 |
| Passo 5: 11 | 12 | 3 | 10 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |
| Passo 6: 06 | 10 | 3 | 12 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |
| Passo 7: 23 | 3 | 10 | 12 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |
| Passo 8: 72 | 3 | 10 | 12 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |
| Passo 9: 63 | 3 | 10 | 12 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |
| Ordem final: | 1 | 310 | 12 | 23 | 24 | 31 | 47 | 59 | 67 | 86 |

Figura 11.5 – O algoritmo da ordenação por segmentação.

No passo 1, o ponto médio é 55, que representa a média entre 86 e 24. No passo 2, o segmento que está sendo ordenado compreende 24 e 47, levando a um ponto médio igual a 35. Observe que os elementos contidos no segmento raramente se separam equilibradamente

ao redor do ponto médio. Isso não tem efeito sobre o algoritmo, mas realmente diminui de alguma forma sua eficiência.

Em cada passo de processo, a ordenação por segmentação promove a ordenação dos elementos de um segmento do array em torno do valor do ponto médio. À medida que os segmentos se tornam cada vez menores, o array se aproxima da ordenação completa.

No programa a seguir, a rotina **Quick** contém o algoritmo da ordenação por segmentação.

```
Program QuickTest;
Uses CRT;
Type Int_Arr = Array [1..10] Of Integer;
Var InFile : Text;
    i : Integer;
    a : Int_Arr;

(*****)

Procedure Quick(Var item  : Int_arr;
                count : Integer);

(*****)

Procedure PartialSort(left, right : Integer;
                    Var a : Int_Arr);

(*****)

Procedure Switch(var a,b : Integer);
Var c : Integer;
Begin
    If  a <> b Then Begin
        c := a;
        a := b;
        b := c;
    End;
End;

(*****)

Begin
    k := (a[left] + a[right]) Div 2;
    i := left;
    j := right;
    Repeat
        While a[i] < k Do
```

```

        Inc(i,1);
    While k < a[j] Do
        Dec(j,1);
    If i <= j Then Begin
        Switch(a[i],a[j]);
        Inc(i,1);
        Dec(j,1);
    End;
Until i > j;
If left < j Then
    PartialSort(left,j,a);
If i < right Then
    PartialSort(i,right,a);
End;

(*****

Begin
    PartialSort(1,count,item);
End;

(*****

Begin
    ClrScr;
    a[1] := 86;
    a[2] := 03;
    a[3] := 10;
    a[4] := 23;
    a[5] := 12;
    a[6] := 67;
    a[7] := 59;
    a[8] := 47;
    a[9] := 31;
    a[10] := 24;

    For i := 1 To 10 Do
        Write(a[i]:4);
    WriteLn;

    Quick(a,10);

    For i := 1 To 10 Do
        Write(a[i]:4);
    WriteLn;
    Write('Pressione ENTER...');
    ReadLn;
End.

```

Essa rotina começa chamando a sub-rotina **PartialSort**, que considera três parâmetros – o limite inferior do segmento do array, o limite superior, e o próprio array. Ao ser chamado pela primeira vez, o limite inferior especificado para o **PartialSort** é 1, e o limite superior é a quantidade de elementos do array.

A sub-rotina **PartialSort** calcula o ponto médio e ordena os elementos contidos no segmento do array concomitantemente. Em seguida, ela chama a si mesma, especificando novos limites inferior e superior, concentrando-se, dessa forma, nos segmentos progressivamente menores do array. Ao chegar ao nível mais baixo do array, a recursividade e a rotina enviam o array ordenado de volta para o programa.

Comparação dos Algoritmos de Ordenação

A quantidade de comparações exigidas para ordenar uma lista é a medida universal pela qual todos os algoritmos de ordenação são avaliados. A quantidade de comparações é expressa como um múltiplo da quantidade de elementos da lista. Por exemplo, se você estiver ordenando um array de n elementos com a ordenação por troca, o programa terá que executar $1/2(n^2-n)$ comparações. Se n for igual a 100, a quantidade de comparações será 4.950.

Essa marca de referência é adequada para as comparações com uma curvatura teórica, mas a maioria dos programadores considera mais fácil comparar métodos de ordenação considerando o tempo gasto por cada método para ordenar o mesmo array. A **Tabela 11.1** mostra os resultados dos testes realizados utilizando os algoritmos da ordenação por troca, da ordenação shell, e da ordenação por segmentação para arrays com 100, 500 e 1.000 números aleatórios. Como mostra a tabela, a ordenação por troca é um algoritmo pobre comparado com a ordenação shell e com a ordenação por segmentação, levando um tempo de 6 a 68 vezes maior para ordenar um array. Entre a ordenação shell e a ordenação por segmentação, a diferença em termos de tempo também é significativa. A ordenação shell leva um tempo duas vezes maior que a ordenação por segmentação e quatro vezes maior em muitas comparações.

Tabela 11.1 – Eficiência relativa de diferentes métodos de ordenação.

| <i>N</i> | <i>Por Troca</i> | | <i>Shell</i> | | <i>Por Segmentação</i> | |
|-------------|------------------|--------------------|--------------|--------------------|------------------------|--------------------|
| | <i>Tempo</i> | <i>Comparações</i> | <i>Tempo</i> | <i>Comparações</i> | <i>Tempo</i> | <i>Comparações</i> |
| 100 | 0.66 | 4.950 | 0.11 | 849 | 0.06 | 232 |
| 500 | 15.88 | 124.750 | 0.77 | 5.682 | 0.44 | 1.473 |
| 1000 | 63.66 | 499.500 | 1.87 | 13.437 | 0.93 | 3.254 |

A única desvantagem em relação à ordenação por segmentação é a quantidade de espaço que ela exige na área de armazenamento. Como a ordenação por segmentação é uma rotina recursiva, o espaço na área de armazenamento deve ser alocado sempre que a rotina chamar a si mesma. Se você estiver preocupado com o espaço na área de armazenamento, deverá usar a ordenação shell. Caso contrário, deve usar a ordenação por segmentação.

CONCLUSÃO

Com a necessidade de produtividade e qualidade, os programadores tiveram a necessidade de desenvolver funções como as explanadas neste trabalho.

Tentou-se abordar o assunto de uma forma simples, mas completa, objetivando servir de uma ferramenta de apoio aos programadores.

Conclui-se que as rotinas de ordenação podem ser de grande utilidade e economia, desde que empregadas da forma correta.

BIBLIOGRAFIA

Livro : Turbo Pascal 6 – Completo e Total

Autor : Stephen O'Brien

Editora : Makron Books

Sites na Internet

Foram consultados diversos sites na Internet.

