UNIDADE 1 - ORDENAÇÃO E BUSCA

1.1 INTRODUÇÃO À ORDENAÇÃO

Entende-se a atividade de ordenação como sendo o processo de rearranjo de um certo conjunto de objetos de acordo com um critério (ordem) específico.

O objetivo da ordenação é facilitar a localização dos membros de um conjunto de dados. Assim sendo, é uma atividade fundamental e universalmente utilizada para a elaboração de algoritmos mais complexos.

A técnica da ordenação ilustra claramente como um dado problema pode ser resolvido por meio de diferentes algoritmos, cada qual apresentando vantagens e desvantagens, as quais devem ser consideradas sob o ângulo da particular aplicação a que se destina. A dependência da escolha de um algoritmo quanto a estrutura de dados a ser processada é tão forte no caso da ordenação que os métodos utilizados são em geral classificados em duas categorias:

- ordenação de vetores;
- ordenação de arquivos.

As duas classes são freqüentemente chamadas de ordenação interna e externa, porque os vetores são armazenados nas memórias "internas" dos computadores, de acesso aleatório e rápido, que apresentam alta velocidade, enquanto os arquivos são armazenados em memórias "externas" de acesso lento, porém com alta capacidade de armazenamento, baseadas em dispositivos mecânicos (discos/fitas).

1.2 ORDENAÇÃO DE VETORES

O registro mais importante a ser estabelecido em relação aos métodos de ordenação de vetores corresponde ao uso econômico da memória disponível. Isto implica que a permutação de elementos responsável por levar o elemento a ordem desejada deve ser efetuada "*in situ*", o que torna de menor interesse os métodos que efetuam o transporte físico dos elementos de um vetor A para um vetor resultante B.

Assim, restringindo-se a escolha dos métodos, de acordo com o critério de economia de memória, pode-se prover uma primeira classificação de acordo com a eficiência do mesmo em relação a economia de tempo.

Enquanto bons algoritmos de ordenação exigem cerca de ***n*log(n)** comparações, várias técnicas simples de ordenação chamadas métodos diretos, exigem n^2 comparações.

Podemos citar as seguintes vantagens em se iniciar os estudos pelos métodos diretos:

- São particularmente bem adequados para elucidar as características dos principais princípios de ordenação;
- Seus programas são curtos e de fácil compreensão;
- Embora métodos sofisticados exijam menor número de comparações, estas operações são, em geral, mais complexas em seus detalhes. Conseqüentemente, os métodos diretos são mais rápidos para os casos em que

n for suficientemente pequeno.

Os métodos de ordenação que ordenam os elementos "*in situ*" podem ser classificados em 3 principais categorias, de acordo com o método empregado em seu projeto:

- Ordenação por inserção;
- Ordenação por seleção;
- Ordenação por troca/permutação.

1.2.1 ORDENAÇÃO POR INSERÇÃO DIRETA

Neste método, considera-se o segmento ordenado como sendo formado pelo primeiro elemento do vetor de chaves. Os demais elementos, ou seja do 2º elemento ao último, pertencem ao segmento não ordenado.

A partir desta situação inicial, toma-se um a um dos elementos não ordenados, a partir do primeiro e, por busca seqüencial realizada da direita para a esquerda no segmento já ordenado, localiza-se a sua posição relativa correta.

À cada comparação realizada entre o elemento a ser inserido e os que lá já se encontram, podemos obter um dos seguintes resultados:

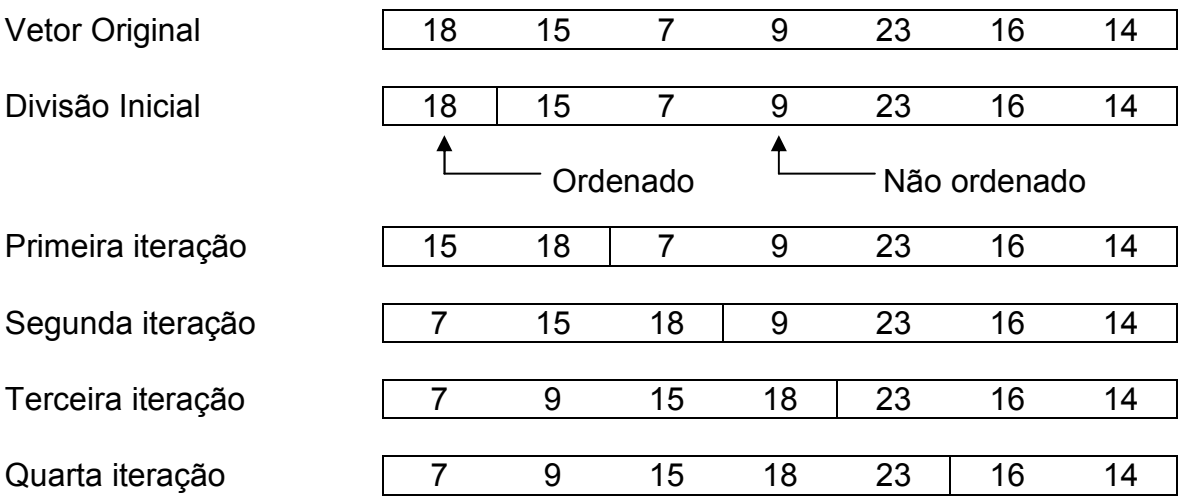
a) O elemento a ser inserido é menor do que aquele com que se está comparando. Nesse caso, este é movido uma posição para a direita, deixando, assim, vaga a posição que anteriormente ocupava.
b) O elemento a ser inserido é maior ou igual àquele com que se está comparando. Nesse caso, fazemos a inserção do elemento na posição vaga, a qual corresponde à sua posição relativa correta no segmento já ordenado.

Caso o elemento a ser inserido seja maior do que todos, a inserção corresponde a deixá-lo na posição que já ocupava.

Após a inserção, a fronteira entre os dois segmentos é deslocada uma posição para a direita, indicando, com isto, que o segmento ordenado ganhou um elemento e o não ordenado perdeu um.

O processo prossegue até que todos os elementos do segundo segmento tenham sido inseridos no primeiro.

Exemplo:



Quinta iteração	7	9	15	16	18	23	14
Sexta iteração (ordenado)	7	9	14	15	16	18	23

O algoritmo do método de ordenação por inserção direta é apresentado a seguir.

```
procedimento INSERCAO_DIRETA (var C: VETOR; N: inteiro);
{C = vetor a ser ordenado; N = número de elementos no vetor}
var
   I, J, K, CH: inteiro;

   para I = 2 até N faça {I = posição do elemento a inserir}
      K ← 1;             {K = início do primeiro segmento}
      J ← I - 1;         {J = fim do primeiro segmento}
      CH ← C[I];

      enquanto ((J >= 1) e (K = 1)) faça
         se (CH < C[J]) então
            C[J+1] ← C[J];
            J ← J - 1;
         senão
            K ← J + 1;  {posição de inserção}
         fim se;
      fim enquanto;

      C[K] ← CH;
   fim para;

fim procedimento INSERCAO_DIRETA;
```

1.2.2 ORDENAÇÃO PELO MÉTODO DOS INCREMENTOS DECRESCENTES – SHELLSORT

Um refinamento do método por inserção direta foi proposto por Shell em 1959. Primeiramente todos os elementos que estiverem a intervalos de quatro posições entre si na seqüência corrente são agrupados e ordenados separadamente. Este processo é chamado de ordenação de distância quatro. Após este primeiro passo, os elementos são agrupados em grupos cujo intervalo é de duas posições, sendo então ordenados novamente. Este processo é chamado ordenação de distância 2. Finalmente, num terceiro passo, todos os elementos são ordenados através de uma ordenação simples de distância 1.

Exemplo:

Convenção: os números acima de cada célula indicam os índices das chaves no vetor. Os números abaixo de cada célula indicam o número do segmento

ao qual cada célula pertence.

Primeiro passo: h = 4.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	25	49	12	18	23	45	38	53	42	27	13	11	28	10	14
1	2	3	4	1	2	3	4	1	2	3	4	1	2	3	4

Após classificar cada um dos quatro segmentos, o vetor terá o aspecto a seguir:

Segundo passo: h = 2.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
11	23	10	12	17	25	27	13	18	28	45	14	53	42	49	38
1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2

Após classificar cada um dos dois segmentos, o vetor terá o aspecto seguinte:

Terceiro (e último) passo: h = 1.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	12	11	13	17	14	18	23	27	25	45	28	49	38	53	42
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Após a execução do último passo, o vetor resultará classificado:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	11	12	13	14	17	18	23	25	27	28	38	42	45	49	53

O algoritmo do método de ordenação Shellsort é apresentado a seguir.

```
procedimento INSERC_DIR_SHELL(VETOR: var C: VETOR; N, F, H: inteiro);
{C = vetor a ser ordenado; N = número de elementos no vetor}
var
   I, K, J, CH: inteiro;

   I ← F + H;

   enquanto (I <= N) faça
      K ← F;
      J ← I - H;
      CH ← C[I]

      enquanto ((J >= F) e (K = F)) faça
         se (CH < C[J]) então
            C[J + H] ← C[J];
            J ← J - H;
```

```
            senão
               K ← J + H;
            fim se;
         fim enquanto;

         C[K] ← CH;
         I ← I + H;

      fim enquanto;

fim procedimento INSERC_DIR_SHELL;


procedimento SHELLSORT (var C: VETOR; N, NP: inteiro);
var
   P, J, H: inteiro;

   para P = NP até 1 faça {decrescente}
      H ← 2^(P-1);
      para J = 1 até H faça
         INSERC_DIR_SHELL (C, N, J, H);
      fim para;
   fim para;
fim procedimento SHELLSORT;
```

1.2.3 ORDENAÇÃO POR SELEÇÃO DIRETA

Neste método, a seleção da menor chave é feita por pesquisa seqüencial. Uma vez encontrada a menor chave, esta é permutada com a que ocupa a posição inicial do vetor, que fica, assim, reduzido de um elemento. O processo de seleção é repetido para a parte restante do vetor, sucessivamente, até que todas as chaves tenham sido selecionadas e colocadas em suas posições definitivas.

Exemplo:

Suponha que se deseja ordenar o vetor:

9	25	10	18	5	7	15	3

Segundo o princípio formulado, a ordenação desse vetor de 8 chaves demandará 7 iterações, conforme mostrado na tabela a seguir.

Observe que a última iteração (7ª) encerra a classificação, pois se as ***n–1*** menores chaves do vetor estão em suas posições definitivas corretas, então a maior (e última) automaticamente também estará.

Iteração	Vetor	Chave Selecionada	Permutação	Vetor ordenado até a posição
1	9 25 10 18 5 7 15 3	3	9 ↔ 3	
2	3 25 10 18 5 7 15 9	5	25 ↔ 5	1
3	3 5 10 18 25 7 15 9	7	10↔ 7	2
4	3 5 7 18 25 10 15 9	9	18 ↔ 9	3
5	3 5 7 9 25 10 15 18	10	10 ↔ 25	4
6	3 5 7 9 10 25 15 18	15	15 ↔ 25	5
7	3 5 7 9 10 15 25 18	18	25 ↔ 18	6
	3 5 7 9 10 15 18 25			8

O algoritmo do método de ordenação por seleção direta é apresentado a seguir.

```
procedimento SELECAO_DIRETA (var C: VETOR; N: inteiro);
{C = vetor a ser ordenado; N = número de elementos no vetor}
var
   I, MIN, J, CH: inteiro;

   para I = 1 até (N - 1) faça
      MIN ← I;

      para J = (i + 1) até N faça
         se (C[J] < C[MIN]) então
            MIN ← J;
         fim se;
      fim para;

      CH ← C[I];
      C[I] ← C[MIN];
      C[MIN] ← CH;
   fim para;

fim procedimento SELECAO_DIRETA;
```

1.2.4 ORDENAÇÃO PELO MÉTODO DA BOLHA – BUBBLESORT

Este método é um método típico de ordenação por troca/permutação. Neste método, o princípio geral é aplicado a todos os pares consecutivos de chaves. Este processo é executado enquanto houver pares consecutivos de chaves não ordenados. Quando não restarem mais pares não ordenados, o vetor estará classificado.

Exemplo:

Suponha que se deseje classificar em ordem crescente o seguinte vetor de chaves:

28	26	30	24	25

Comparamos todos os pares de chaves consecutivas, a partir do par mais à esquerda. Caso as chaves de um certo par se encontrem fora da ordem desejada, efetuamos a troca das mesmas. O processo de comparação dos ***n – 1*** pares de chaves é denominado de varredura. O método efetuará tantas varreduras no vetor quanto forem necessárias para que todos os pares consecutivos de chaves se apresentem na ordem desejada.

Primeira varredura:

28	26	30	24	25	Compara par (28, 26): troca.
26	28	30	24	25	Compara par (28, 30): não troca.
26	28	30	24	25	Compara par (30, 24): troca.
26	28	24	30	25	Compara par (30, 25): troca.
26	28	24	25	30	Fim da primeira varredura.

Como ainda existem pares não ordenados, reiniciamos o processo de comparações de pares de chaves, executando mais uma varredura. Observe, no entanto, que a primeira varredura sempre irá posicionar a chave de maior valor na sua posição definitiva correta (no final do vetor). Isto significa que na segunda varredura podemos desconsiderar a última posição do vetor, que portanto fica reduzido de um elemento. Esta situação se repetirá ao final de cada varredura efetuada.

Segunda varredura:

26	28	24	25	30	Compara par (26, 28): não troca.
26	28	24	25	30	Compara par (28, 24): troca.
26	24	28	25	30	Compara par (28, 25): troca.
26	24	25	28	30	Fim da segunda varredura.

Terceira varredura:

26	24	25	28	30	Compara par (26, 24): troca.
24	26	25	28	30	Compara par (26, 25): troca.
24	25	26	28	30	Fim da terceira varredura.

Como não restam mais pares não ordenados, o processo de classificação é dado por concluído.

Examinando-se o comportamento do método, vemos que eventualmente, dependendo da disposição das chaves, uma certa varredura pode colocar mais do que uma chave na sua posição definitiva. Isto ocorrerá sempre que houver blocos de chaves já previamente ordenadas, como no exemplo abaixo.

Vetor de chaves:

13	11	25	10	18	21	23

A primeira varredura produzirá o seguinte resultado:

11	13	10	18	21	23	25

Observe que, neste caso, as quatro últimas chaves já se encontram na

sua posição definitiva. Isto significa que a próxima varredura não precisa ignorar apenas a última chave. As quatro últimas podem ser ignoradas. A quantidade de chaves, a partir da última, que pode ser ignorada de uma varredura para outra é conhecida pela posição na qual ocorreu a última troca. A partir daquele ponto o vetor já se encontra ordenado.

A denominação deste método resulta da associação das chaves com bolhas dentro de um fluido. Cada bolha teria um diâmetro proporcional ao valor de uma chave. Assim, as bolhas maiores subiriam com velocidades maiores, o que faria com que, após um certo tempo, elas se arranjassem em ordem de tamanho.

O algoritmo do método de ordenação bubblesort é apresentado a seguir.

```
procedimento BUBBLESORT (var C: VETOR; N: inteiro);
{C = vetor a ser ordenado; N = número de elementos no vetor}
var
   I, M, K, CH: inteiro;
   TROCA : lógico;

   TROCA ← True;
   M ← N - 1;
   K ← 1;

   Enquanto (TROCA) faça
      TROCA ← False;

      para I = 1 até M faça
         se (C[I] > C[I + 1]) então
            CH ← C[I];
            C[I] ← C[I + 1];
            C[I + 1] ← CH;
            K ← I;          {posição da última troca}
            TROCA ← True;   {sinaliza ocorrência de troca}
         fim se;
      fim para;

      M ← K;                {vetor já ordenado de M+1 até N}
   fim enquanto;

fim procedimento BUBBLESORT;
```

1.2.5 ORDENAÇÃO POR PARTICIONAMENTO – QUICKSORT

A ordenação por particionamento é baseada no fato de que as permutações devem ser preferencialmente empregadas para pares de elementos que guardem entre si distâncias grandes, com a finalidade de se conseguir uma eficiência maior.

Admite-se que sejam fornecidos ***n*** elementos no ordem inversa de suas chaves. É possível ordená-los com apenas ***n/2*** permutações tomando-se primeiramente os elementos das extremidades e convergindo gradualmente para o

centro pelos dois lados.

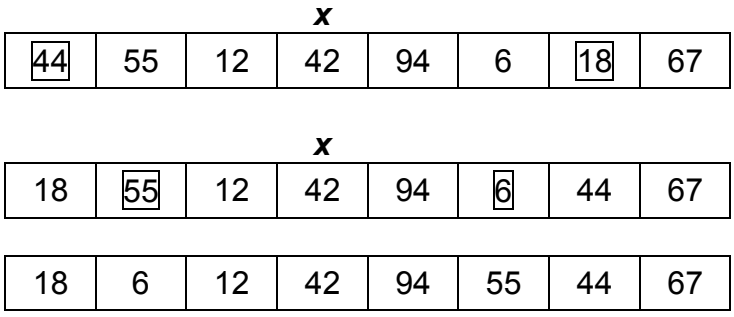
Obviamente, isto é possível se os elementos estiverem exatamente na ordem inversa. A metodologia empregada é a seguinte:

- escolhe-se arbitrariamente um elemento (denominado ***x***);
- o vetor é varrido da esquerda para a direita até que seja encontrado um elemento $a_i > x$, sendo então varrido da direita para a esquerda até que seja encontrado um elemento $a_j < x$;
- nesta ocasião os dois elementos serão permutados e este processo de varredura e permutação continua até que os dois deslocamentos se encontrem em algum ponto intermediário do vetor;
- nos casos em que somente uma das partições possui elemento a ser trocado este será trocado com o próprio elemento ***x***;
- o resultado dessa prática é um vetor particionado, onde a partição esquerda contém apenas chaves cujos valores são menores, ou iguais, que ***x*** e a partição direita o inverso.

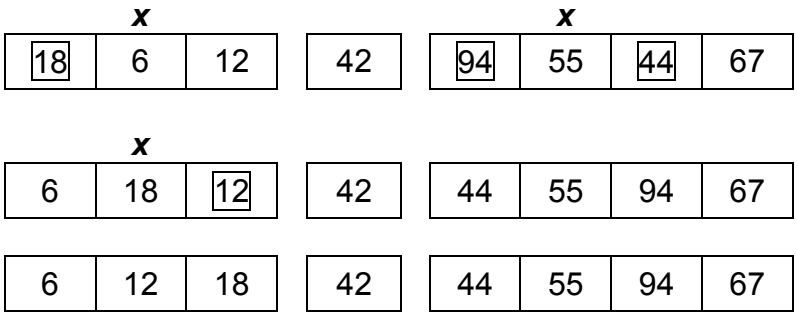
Após ter sido particionado o vetor, aplica-se o mesmo processo para ambas as partições e em seguida para as partições oriundas de cada partição obtida, e assim por diante, até que todas as partições consistam de apenas um único elemento.

Exemplo:

Seja o vetor descrito a seguir. Inicialmente escolhemos o elemento ***x*** que dividirá o vetor em duas partes e iniciamos o processo de ordenação .



Terminada a varredura nesta primeira etapa, particionamos o vetor acima em dois vetores menores e para cada segmento escolhemos um novo elemento ***x***.



O algoritmo do método de ordenação quicksort é apresentado a seguir.

```
procedimento QUICKSORT (INIC, FIM: inteiro ; var C: VETOR);
{INIC = primeiro elemento de C; FIM = último elemento de C}

   procedimento QSORT (INIC, FIM: inteiro);
   var
      I, J, K, TEMP: inteiro;

      K ← C[(INIC + FIM) div 2];{elemento particionador}
      I ← INIC;
      J ← FIM;

      repita

         enquanto (C[I] < K) faça
            I ← I + 1;
         fim enquanto;

         enquanto (K < C[J]) faça
            J ← J - 1;
         fim enquanto;

         se (I <= J) então
            TEMP ← C[I];
            C[I] ← C[J];
            C[J] ← TEMP;
            I ← I + 1;
            J ← J - 1;
         fim se;

      até que (I > J);

      se (INIC < J) então QSORT (INIC, J);
      se (I < FIM) então QSORT (I, FIM);

   fim procedimento QSORT;

   QSORT (INIC, FIM);

fim procedimento QUICKSORT;
```

1.2.6 CLASSIFICAÇÃO POR INTERCALAÇÃO SIMPLES – MERGESORT

A classificação por intercalação se baseia em um princípio de funcionamento bem simples. Consiste em dividir o vetor de chaves a ser classificado em dois ou mais, ordená-los separadamente e, depois, intercalá-los dois a dois, formando, cada par intercalado, novos segmentos ordenados, os quais serão

intercalados entre si, reiterando-se o processo até que resulte apenas um único segmento ordenado.

A Figura 1.1 ilustra este processo, supondo uma divisão inicial em quatro segmentos.

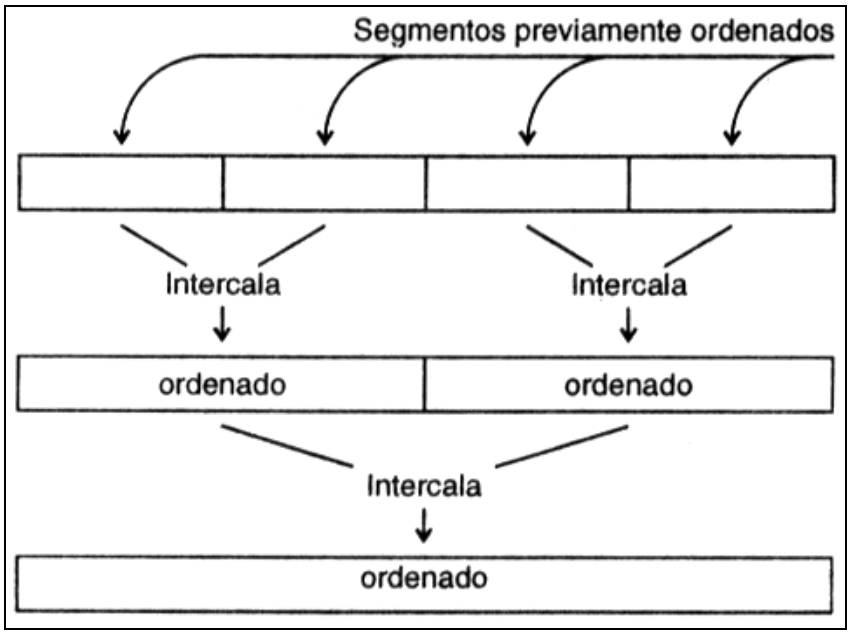


FIGURA 1.1 – Esquema do processo de intercalação

O algoritmo Mergesort inicia o processo considerando segmentos de comprimento 1, os quais, por conterem apenas um elemento cada, já estão ordenados. Dessa forma, consideramos o vetor de ***n*** elementos como sendo formado de ***n*** segmentos de 1 elemento cada, e aplicamos o processo reiterado de intercalação a partir daí, sem a necessidade de lançar mão de outros métodos de classificação para dar partida ao processo.

Exemplo:

A Figura 1.2 mostra um exemplo deste princípio sendo aplicado a um vetor de 8 chaves. Cada linha, a partir da segunda, corresponde ao resultado da intercalação de todos os pares de segmentos consecutivos da linha anterior.

23	17	8	15	9	12	19	7
17	23	8	15	9	12	7	19
8	15	17	23	7	9	12	19
7	8	9	12	15	17	19	23

FIGURA 1.2 – Classificação por intercalação de um vetor de 8 chaves

Quando o tamanho do vetor não for uma potência inteira de 2, sempre ocorrerá, em pelo menos uma iteração, que um segmento resulte sem um par para ser intercalado. Quando isto ocorrer, esse segmento, que obviamente sempre será o último do vetor, é simplesmente transcrito para a iteração subseqüente, como no caso ilustrado na Figura 1.3

23	17	8	15	9	12	19	7	14	10
17	23	8	15	9	12	7	19	10	14
8	15	17	23	7	9	12	19	10	14
7	8	9	12	15	17	19	23	10	14
7	8	9	10	12	14	15	17	19	23

FIGURA 1.3 – Intercalação com $n \neq 2^i$, i inteiro

Implementação:

A implementação deste método será feita em 3 níveis. No primeiro, denominado *simplemerge*, é definida a operação de *intercalação de um par* de segmentos ordenados, resultando um terceiro segmento também ordenado. O segundo nível, *mergepass*, fará a *intercalação de todos os pares* de segmentos ordenados nos quais o vetor está dividido. O terceiro e último nível, *mergesort*, efetuará toda a *seqüência de intercalações* necessária para que resulte apenas um segmento intercalado.

Esquematicamente, a implementação da operação *simplemerge* está representada na Figura 1.4. Os dois segmentos a serem intercalados ocupam posições contíguas no vetor **C**. O primeiro inicia na posição **is1** e termina na **fs1**. O segundo ocupa as posições de **is2** até **fs2**. O resultado da intercalação aparece no vetor **C'**, a partir da posição **r**.

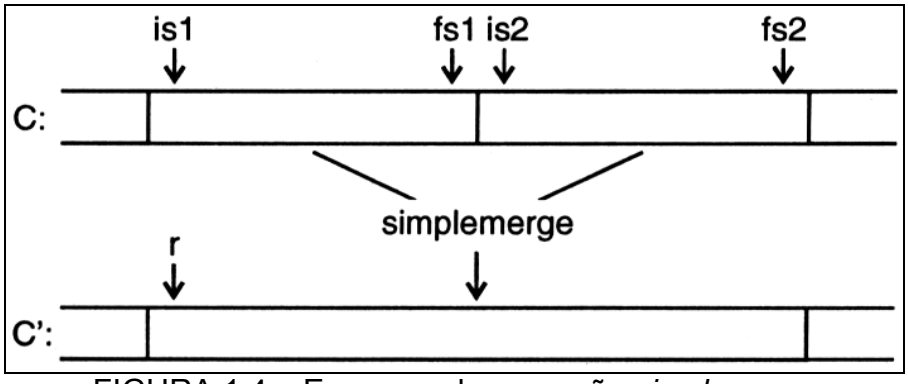


FIGURA 1.4 – Esquema da operação *simplemerge*

A operação *mergepass*, por sua vez, efetuará a intercalação de todos os pares consecutivos de segmentos de comprimento ℓ do vetor **C**, conforme o esquema da Figura 1.5.

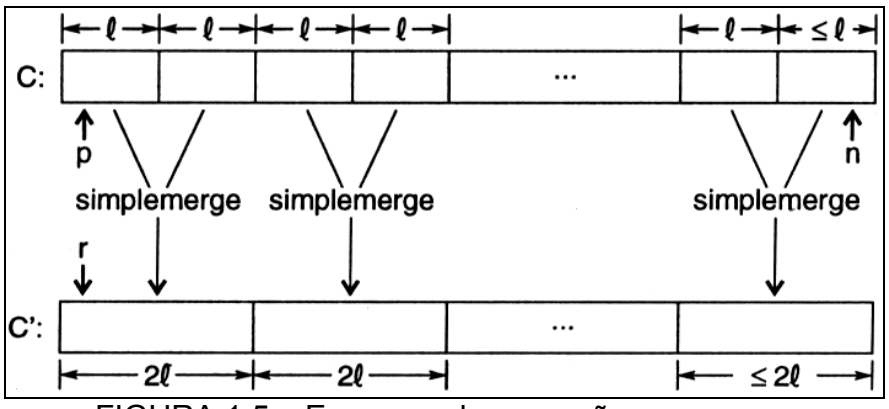


FIGURA 1.5 – Esquema da operação *mergepass*

```
procedimento SIMPLEMERGE (C, var C': VETOR;
                          IS1, FS1, IS2, FS2, R: inteiro);
var
   IS1', IS2', K, I: inteiro;

   IS1' ← IS1;
   IS2' ← IS2;
   K ← R;

   enquanto ((IS1'<= FS1) e (IS2' <= FS2)) faça

      se (C[IS1'] < C[IS2']) então
         C'[K] ← C[IS1'];
         IS1' ← IS1' + 1;
      senão
         C'[K] ← C[IS2'];
         IS2' ← IS2' + 1;
      fim se;

      K ← K + 1;
   fim enquanto;

   se(IS1' > FS1) então
      para I = IS2' até FS2 faça
         C'[K] ← C[I];
         K ← K + 1;
      fim para;
   senão
      para I = IS1' até FS1 faça
         C'[K] ← C[I];
         K ← K + 1;
      fim para;
   fim se;

fim procedimento SIMPLEMERGE;

procedimento MERGEPASS (C, var C': VETOR; N, ℓ: inteiro;
                        var E_PAR: lógico);
{ℓ = comprimento dos segmentos}
var
   P, Q, R, I: inteiro;

   E_PAR ← não E_PAR;
   P ← 1;      {P = início do primeiro segmento}
   Q ← P + ℓ;  {Q = inicio do segundo segmento}
   R ← 1;      {R = início do segmento resultante}
```

```
   enquanto (Q <= N) faça {enquanto houverem pares de segmentos}
      {intercala um par de segmentos}
      SIMPLEMERGE (C, C', P, Q - 1, Q, MIN(Q + ℓ - 1, N), R);
      R ← R + 2 * ℓ;
      P ← Q + ℓ;
      Q ← P + ℓ;
   fim enquanto;

   se (P <= N) então
      para I = P até N faça
         C'[I] ← C[I];
      fim para;
   fim se;

fim procedimento MERGEPASS;
```

Observe que o 6º parâmetro de chamada da operação *simplemerge*, que indica a posição final do 2º segmento, prevê a possibilidade de que o último deles tenha um comprimento menor do que ℓ, conforme mostrou a Figura 1.5.

O nível final de implementação, visível ao usuário, deverá efetuar todas as seqüências de intercalações, desde as dos segmentos de comprimento 1, até obter um segmento intercalado de comprimento **n**, dobrando, após cada execução da operação *mergepass*, o comprimento dos segmentos.

Devido ao fato da intercalação sempre ser feita em um vetor auxiliar, a operação *mergepass* deverá alternar sucessivamente os vetores de origem e destino, de acordo com o esquema mostrado na Figura 1.6. Dessa forma, as primeiras intercalações serão feitas do vetor **C** para o vetor **C'**. As segundas de **C'** para **C** e assim até o final. Ao concluir a última seqüência de intercalações, deve ser verificado em qual dos dois vetores, **C** ou **C'**, está o resultado final. Isto pode ser feito examinando-se a quantidade de iterações executadas. Se estiver em **C'**, este é copiado para **C**, uma vez que **C'** não é visível ao usuário.

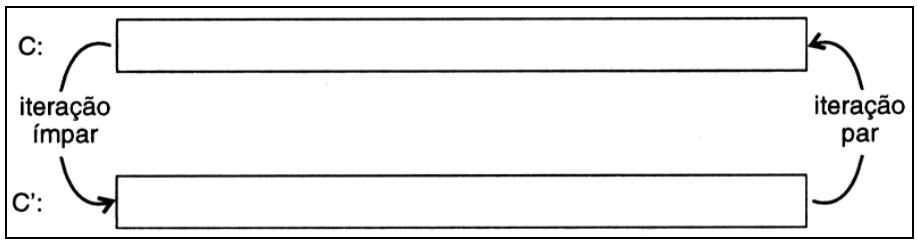


FIGURA 1.6 – Utilização do vetor auxiliar em cada iteração

```
procedimento MERGESORT (var C: VETOR; N: inteiro);
var
   C': VETOR;
   ℓ, I: inteiro;
   E_PAR: lógico;

   ℓ ← 1; {comprimento inicial dos segmentos}
```

```
    E_PAR ← Verdadeiro;
    enquanto (ℓ < N) faça {tamanho do menor segmento < N}

       se (E_PAR) então
          MERGEPASS (C, C', N, ℓ, E_PAR);
       senão
          MERGEPASS (C', C, N, ℓ, E_PAR);
       fim se;

       ℓ ← 2 * ℓ;

    fim enquanto;

    {se última intercalação resultou em C', copia C' em C}
    se (não E_PAR) então
       para I = 1 até N faça
          C[I] ← C'[I];
       fim para;
    fim se;

fim procedimento MERGESORT;
```

1.3 ORDENAÇÃO EXTERNA

Infelizmente, é inviável a aplicação dos algoritmos de ordenação apresentados anteriormente no caso de um conjunto de dados estar gravado em um dispositivo periférico de memória seqüencial, tal como uma fita magnética ou um disco. Neste caso, os dados são descritos na forma de arquivos (seqüenciais), cuja característica é que a cada instante é possível o acesso direto a apenas um e somente um de seus componentes. Isto é uma restrição severa, comparando com as possibilidades oferecidas pela estrutura de vetor. Devemos nestes casos, portanto, utilizar diferentes técnicas de ordenação.

1.3.1 CLASSIFICAÇÃO COM DISCOS – INTERCALAÇÃO DE ARQUIVOS

A classificação por intercalação é o método mais popular de classificação nos dispositivos de memória externa. Este método consiste, essencialmente, de duas fases distintas.

Na primeira fase os segmentos do arquivo de entrada são classificados usando um bom método de classificação interna. Esses segmentos classificados, conhecidos como “corridas” estão sendo gravados na memória externa na medida em que são gerados.

Na segunda fase, as “corridas” geradas na primeira fase são intercaladas seguindo a configuração da árvore de intercalação até esgotarem as corridas. O algoritmo de intercalação necessita que esteja em memória apenas o registro da frente das duas corridas que estão sendo intercaladas. Dessa forma, é possível intercalar grandes corridas. A Figura 1.7 mostra o esquema da classificação externa

por intercalação de arquivos.

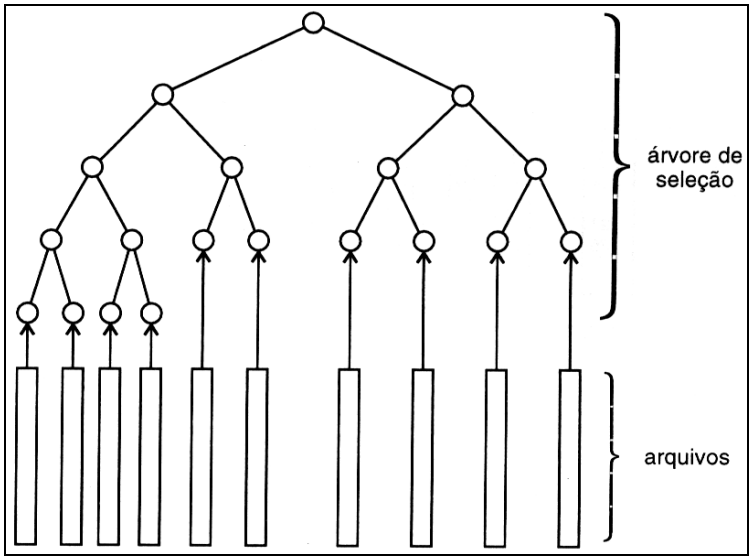


FIGURA 1.7 – Esquema da classificação externa por intercalação de arquivos

Exemplo:

Deve ser classificado um arquivo contendo 4500 registros num computador com memória interna com capacidade de classificar no máximo 750 registros. O arquivo de entrada está sendo mantido em um disco e dispõe-se de outro disco que pode ser utilizado como área auxiliar de armazenamento. Nada pode ser gravado no disco de entrada. A melhor maneira para conseguir a classificação usando o procedimento acima descrito será:

a) classifique internamente 3 blocos (B1 – B3) de registros com 1500 registros por bloco. Como a memória disponível permite trabalhar com apenas 750 registros por vez, divida os blocos de 1500 registros em 6 corridas de 250 registros cada. Ordene as corridas com 250 registros utilizando-se de um bom algoritmo de ordenação interna. Ao final desta etapa, cada bloco inicial com 1500 registros estará dividido em 6 corridas ordenadas (R1a – R6a) com 250 registros em cada uma delas.
b) intercale as corridas, duas a duas, gerando 3 corridas com 500 registros cada (R1b – R3b). Assim, cada bloco de 1500 registros estará dividido em 3 corridas com 500 registros.
c) intercale R1b com R2b gerando R1c, que contém 1000 registros e depois intercale R1c com R3b. Ao final deste passo teremos cada um dos blocos de 1500 registros já ordenados.
d) finalize intercalando os blocos B1 com B2 gerando um bloco com 3000 registros. Agora intercale este bloco com B3, obtendo o arquivo final ordenado.

1.3.2 CONSIDERAÇÕES SOBRE CLASSIFICAÇÃO EXTERNA

A classificação externa é extremamente influenciada pelas características do meio onde estão armazenados os dados a classificar, que é usualmente um disco magnético. Além disso, é usual que os arquivos apresentem registros com tamanho considerável (acima de 200 bytes), o que torna inviável proceder a ordenação

diretamente sobre o arquivo original. O procedimento adotado é o de formar um outro arquivo $\mathbf{A_1}$, consideravelmente menor do que o arquivo **A**, composto apenas por dois campos. A seguir $\mathbf{A_1}$ é classificado pelo valor do campo "*chave do registro*", obtendo-se ao final um vetor indireto de ordenação, dado pelo campo "*endereço do registro em **A***". Este vetor indireto de ordenação pode ser usado diretamente nas aplicações ou para a reordenação física do arquivo **A**. Como o arquivo $\mathbf{A_1}$ é usualmente maior que o espaço disponível em memória primária é necessário dividi-lo em segmentos, classificar cada um dos segmentos e posteriormente, por intercalação obter o arquivo $\mathbf{A_1}$ classificado.

1.4 MÉTODOS DE BUSCA

As buscas estão entre as tarefas mais freqüentes encontradas em programação de computadores. Há diversas variações básicas sobre o assunto e uma grande variedade de algoritmos foram desenvolvidos com esta finalidade. A hipótese básica no texto que segue é que a coleção de dados, entre os quais um determinado elemento deve ser procurado, é fixa. Admitiremos que este conjunto de ***N*** elementos seja representado através de um vetor, por exemplo como:

```
var
   A: vetor [1..N] de ITEM;
```

Tipicamente, o tipo ***item*** apresenta uma estrutura de registro, contendo um campo que atua como chave para a pesquisa. A tarefa consiste, então, em se encontrar um elemento de ***A*** cujo campo de chave seja igual a um argumento de busca ***X*** fornecido. O índice ***I*** resultante, que satisfaz à condição ***A[i].CHAVE = X*** permite então o acesso aos outros campos do elemento encontrado. Como o interesse básico deste estudo é exclusivamente o da busca, não sendo importantes os dados associados à chave, admitiremos que o tipo ***ITEM*** seja composto apenas do campo de chave, ou seja, o dado é a própria chave.

1.4.1 BUSCA LINEAR

Nos casos em que não se dispõe de informações adicionais sobre os dados a serem pesquisados, o procedimento mais óbvio é uma busca seqüencial por todo o vetor, aumentando-se desta maneira passo a passo o tamanho da região do vetor em que não figura o elemento desejado. Este procedimento é chamado ***busca linear***. Uma busca deste tipo termina quando for satisfeita uma das duas condições seguintes:

- O elemento é encontrado, isto é, $\boldsymbol{A_I = X}$;
- Todo o vetor foi analisado, mas o elemento não foi encontrado.

Isto resulta no algoritmo abaixo:

```
I ← 1;

enquanto ((I <= N) e (A[I] <> X)) faça
```

```
   I ← I + 1;
fim enquanto;
```

Quando ***I*** for maior que ***N***, isto significa que não foi encontrado o elemento.

Evidentemente, o término das repetições é garantido pois, em cada passo, ***I*** é incrementado e, portanto, certamente alcançará o limite ***N*** após um número finito de passos, caso não exista o elemento desejado no vetor.

Obviamente, cada passo requer o incremento do índice e a avaliação de uma expressão booleana. O problema seguinte consiste em verificar se esta tarefa pode ser simplificada, e portanto, a busca ser acelerada? A única possibilidade de se obter tal efeito consiste em se encontrar uma simplificação da expressão booleana, que depende certamente de dois fatores. Assim, é preciso estabelecer uma condição baseada em um único fator que implique nos dois fatores dos quais depende a expressão. Isto é possível desde que se possa garantir que tal elemento será encontrado. Para tanto introduz-se um elemento adicional, com valor ***X***, no final do vetor. A este elemento auxiliar dá-se o nome de ***sentinela***, porque ele evita que a busca avance o limite demarcado pelo índice. O vetor ***A*** será então declarado da seguinte forma:

```
var
   A: vetor [1..N+1] de ITEM;
```

e o algoritmo de busca linear com sentinela torna-se

```
A[N+1] ← X;
I ← 1;

enquanto (A[I] <> X) faça
   I ← I + 1;
fim enquanto;
```

Evidentemente, ***I = N+1*** implica que não foi encontrado o elemento desejado (exceto o correspondente à sentinela).

1.4.2 BUSCA BINÁRIA

É obvio que não existem meios de acelerar uma busca sem que se disponha de maiores informações acerca do elemento a ser localizado. Sabe-se também que uma busca pode ser mais eficiente se os dados estiverem ordenados. Suponha-se, por exemplo, uma lista telefônica em que os nomes não estejam listados em ordem alfabética. Uma coisa dessas seria completamente inútil. A seguir será apresentado, com base neste raciocínio, um algoritmo que explora o ato do vetor estar ordenado.

A idéia principal é a de testar um elemento sorteado aleatoriamente, por exemplo A_M, e compará-lo com um argumento de busca ***X***. Se tal elemento for igual a ***X***, a busca termina, se for menor do que ***X***, conclui-se que todos os elementos com índices menores ou iguais a ***M*** podem ser eliminados dos próximos testes, e se for

maior que ***X***, todos aqueles que possuem índices maior ou igual a ***M*** podem ser também eliminados. Isto resulta no algoritmo abaixo, denominado ***busca binária***; ele utiliza duas variáveis-índices ***L*** e ***R***, que marcam, respectivamente, os limites Esquerdo e Direito da região de ***A*** em que um elemento ainda pode ser encontrado.

```
L ← 1;
R ← N;
FOUND ← Falso;

enquanto ((L <= R) e não(FOUND)) faça
   M ← (L + R) div 2;

   se (A[M] = X) então
      FOUND ← Verdadeiro;
   senão
      se (A[M] < X) então
         L ← M + 1;
      senão
         R ← M – 1;
      fim se;
   fim se;
fim enquanto;
```

Embora a escolha de M possa ser arbitrária no sentido de que o algoritmo funciona independentemente dele, o valor desta variável influencia na eficiência do algoritmo. Torna-se absolutamente claro que a meta é eliminar, em cada passo, o maior número possível de elementos em futuras buscas, sem levar em consideração o resultado da comparação. A solução ótima é escolher a mediana dos elementos, porque esta escolha elimina, em qualquer caso, metade dos elementos do vetor. Como conseqüência, o número máximo de passos será $\boldsymbol{log_2 N}$, arredondado para cima até o número inteiro mais próximo. Com isso, este algoritmo permite uma drástica melhora do desempenho em relação ao método de busca linear, onde o número esperado de comparações é ***N/2***.

1.5 PROCESSAMENTO EM CADEIAS

1.5.1 INTRODUÇÃO

Entende-se por ***cadeia*** uma seqüência qualquer de elementos, denominados ***caracteres***. Os caracteres, por sua vez, são elementos escolhidos de um conjunto denominado ***alfabeto***. Por exemplo, 0110010 é uma cadeia com alfabeto {0, 1}.

As cadeias aparecem, em computação, no processamento de textos, palavras, mensagens, códigos, etc. com aplicações tão diversas quanto o tratamento computacional de dicionários, mensagens de correio eletrônico, seqüenciamento de DNA's em biologia computacional, criptografia de textos confidenciais e muitos outros. De fato, a importância do processamento de cadeias na computação vem crescendo consideravelmente, até mesmo para computadores de uso pessoal, nos quais a edição de textos tem sido a principal aplicação.

Nesta seção será abordado o problema do ***casamento de cadeias***. Dadas duas cadeias de caracteres, o problema consiste em verificar se a primeira delas contém a segunda, isto é, se os caracteres que compõem a segunda cadeia aparecem seqüencialmente também na primeira.

1.5.2 O PROBLEMA DO CASAMENTO DE CADEIAS

Já foi mencionado que uma cadeia é uma seqüência de caracteres escolhidos de um conjunto chamado alfabeto. O número de caracteres da cadeia é o seu ***comprimento***.

Sejam ***X***, ***Y*** duas cadeias de caracteres com comprimentos ***n***, ***m***, respectivamente, $n \geq m$. Supõe-se que ***X*** e ***Y*** sejam representadas por vetores com elementos x_i e y_j, $1 \leq i \leq n$ e $1 \leq j \leq m$. ***Y*** é uma subcadeia de ***X*** quando ***Y*** for uma subseqüência de elementos consecutivos de ***X***, isto é, quando existir algum inteiro $L \leq n - m$, tal que $x_{L+j} = y_j$, $1 \leq j \leq m$.

O problema do casamento de cadeias consiste em verificar se ***Y*** é subcadeia de ***X***. Em caso positivo, localizar ***Y*** em ***X***. Diz-se, então, que houve um casamento de ***Y*** com ***X*** na posição ***L + 1***.

Por exemplo, se ***X*** é a cadeia ***baaabea*** e ***Y*** é ***aabe***, então ***Y*** é subcadeia de ***X*** e a solução do problema do casamento de cadeias deve indicar que ***Y*** aparece em ***X*** a partir do terceiro caractere, como indica a Figura 1.8. Por outro lado, se ***Y*** é formada pela seqüência ***aeb***, então não é subcadeia de ***X***.

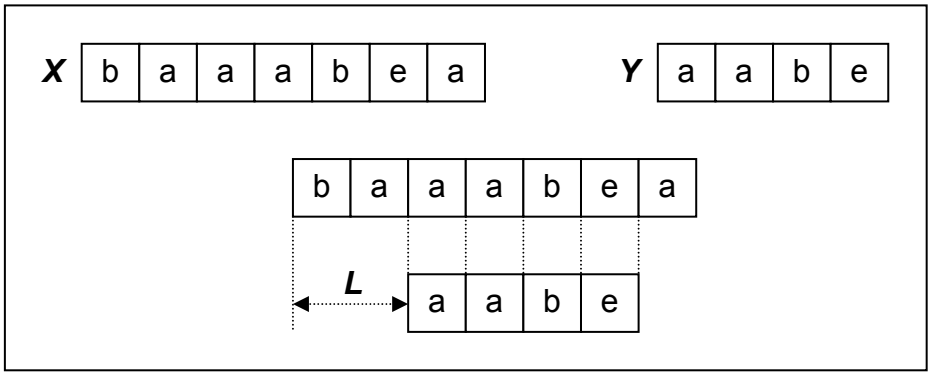


FIGURA 1.8 – O problema do casamento de cadeias

1.5.3 O ALGORITMO DA FORÇA BRUTA

Um método bastante simples para se verificar se ***Y*** é subcadeia de ***X*** consiste em examinar todas as possíveis situações. Se ***Y*** for subcadeia de ***X***, então ***Y*** se encontra em ***X***, deslocado de ***L*** posições à esquerda. A idéia consiste em comparar ***Y*** com a subcadeia de tamanho ***m*** de ***X***, que se inicia no caractere x_{L+1}, para todos os valores possíveis de ***L***. Ou seja, ***L = 0, 1, 2, ..., n – m***. A Figura 1.9 ilustra a aplicação do algoritmo da força bruta para as mesma cadeias ***X*** e ***Y*** da Figura 6.8. O algoritmo, de início, compararia a subcadeia X_1 = ***baaa***, iniciada em x_1, com ***Y***; em seguida, de início, a subcadeia X_2 = ***aaab***, iniciada em x_2, com ***Y***. Nessas duas situações, o teste seria negativo. Na iteração seguinte, em que a subcadeia X_3 = ***aabe*** é considerada, verifica-se o casamento.

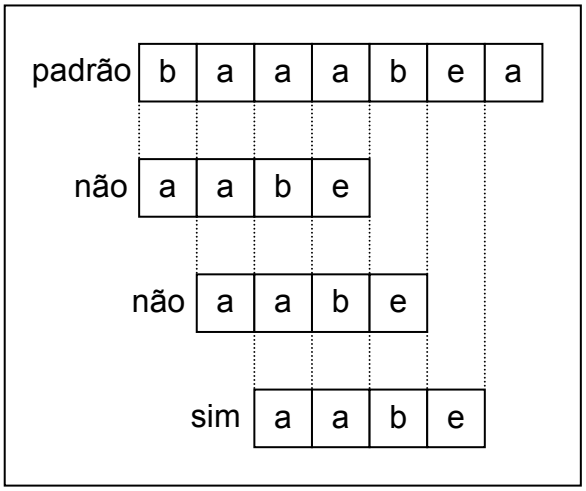


Figura 1.9 – O algoritmo da força bruta

O algoritmo a seguir descreve o processo. A variável ***L*** indica o número de caracteres, na cadeia ***X***, que antecedem cada subcadeia considerada. A variável lógica ***TESTE*** é utilizada para informar a ocorrência ou não de casamento. Para tornar o método mais eficiente, utilizou-se o princípio de abandonar a verificação de uma subcadeia no meio do processo, se for detectado que um casamento não é possível. Por exemplo, se o ***i-ésimo*** caractere da subcadeia considerada não coincide com o ***i-ésimo*** caractere de ***Y***, não há necessidade de comparar os caracteres de ordem ***i + 1, i + 2, ..., m***.

```
função CASAMENTO (X, Y : caractere): inteiro;
{retorna a posição onde ocorre o casamento e 0 caso não ocorra
casamento de cadeias de caracteres}
var
   L, N, M, I : inteiro;
   TESTE : lógico;

   N ← COMPRIMENTO(X); {COMPRIMENTO é função que retorna o}
   M ← COMPRIMENTO(Y); (comprimento da variável parâmetro}

   para L ← 0 até (N – M) faça
      I ← 1;
      TESTE ← Verdadeiro;
      enquanto ((I <= M) e (TESTE)) faça
         se (X[L + I] = Y[I]) então
            I ← I + 1;
         senão
            TESTE ← Falso;
         fim se;
      fim enquanto;

      se (TESTE) então
         CASAMENTO ← L + 1;
         exit;
```

```
      fim se;
   fim para;

   CASAMENTO ← 0;

fim função CASAMENTO;
```

1.5.4 O ALGORITMO DE KNUTH, MORRIS E PRATT

Por volta de 1970, D. E. Knuth, J. H. Morris e V. R. Pratt inventaram um algoritmo que exige, essencialmente, cerca de ***N*** comparações de caracteres, mesmo nos piores casos. O novo algoritmo é baseado no fato de que, no início de cada comparação de um novo padrão, informações valiosas podem estar sendo descartadas e, talvez, algumas destas informações poderão ser utilizadas para se agilizar a busca no texto da cadeia.

O exemplo a seguir auxiliará na compreensão do algoritmo. Dada uma cadeia ***X*** = BABABAEABAEABAEAABAEABAEAAEBA, desejamos saber se a subcadeia ***Y*** = ABAEABAEAAE está contida na cadeia ***X***.

A primeira etapa do algoritmo de Knuth, Morris e Pratt consiste em descobrir padrões de comportamento na subcadeia ***Y***. Estes padrões de comportamento poderão ser armazenados num vetor e depois utilizados para agilizar a busca. A Figura 1.10 mostra o processo de obtenção deste vetor.

O algoritmo que analisa a subcadeia ***Y*** armazenando os padrões de comportamento num vetor é apresentado a seguir.

```
const TAM = 11; {constante do tamanho da subcadeia Y}
tipo VET_PADROES = vetor [1..TAM] de inteiro;

procedimento FIND_PADROES (Y : caractere;
                           var PADROES: VET_PADROES);
var
   I, J, K, M    : inteiro;

   para I ← 1 até TAM faça
      PADROES[I] ← 0;
   fim para;
   J ← 0;
   K ← 1;
   M ← COMPRIMENTO(Y); {ou M ← TAM}

   enquanto (K < M) faça
      se (Y[K + 1] = Y[J + 1]) então
         K ← K + 1;
         J ← J + 1;
         PADROES[K] ← J;
      senão
         se (J = 0) então
            K ← K + 1;
```

```
            PADROES[K] ← 0;
         senão
            J ← PADROES[J];
         fim se;
      fim se;
   fim enquanto;

fim procedimento FIND_PADROES;
```

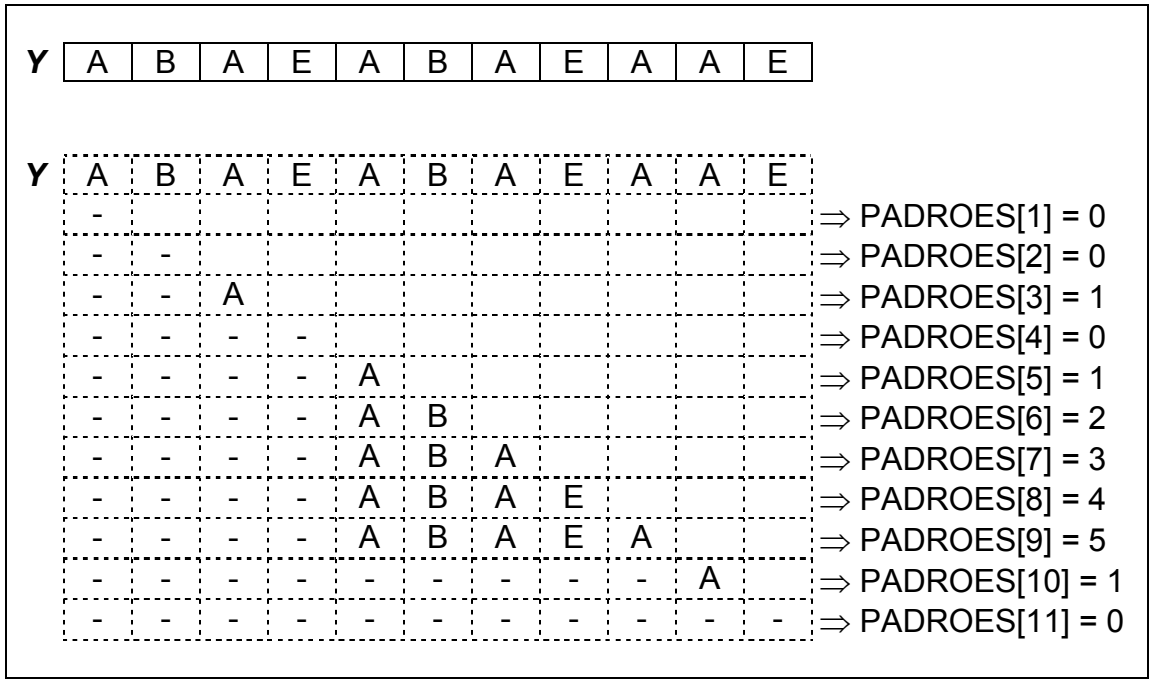


Figura 1.10 – Os valores de PADROES[K]

A segunda etapa do algoritmo consiste em utilizar-se do vetor PADROES para efetuar a busca da subcadeia propriamente dita. Neste processo nem todas as subcadeias de ***X*** com comprimento igual à subcadeia ***Y*** necessitam ser comparadas com ***Y***. A Figura 1.11 mostra as comparações efetuadas pelo algoritmo de Knuth, Morris e Pratt na busca da subacadeia ***Y*** em ***X***.

O algoritmo utilizado para verificar o casamento entre as cadeias ***X*** e ***Y*** é apresentado a seguir.

```
função CASAMENTO_KMP (X, Y : caractere): inteiro;
{retorna a posição onde ocorre o casamento e 0 caso não ocorra
casamento de cadeias de caracteres}

var
   PADROES      : VET_PADROES;
   N, M, I, J   : inteiro;
   TESTE        : lógico;

   N ← COMPRIMENTO(X);
   M ← COMPRIMENTO(Y);
```

```
    I ← 0;
    J ← 0;
    FIND_PADROES(Y, PADROES);

    enquanto ((I - J) <= (N - M)) faça
       TESTE ← Verdadeiro;

       enquanto ((J < M) e (TESTE)) faça
          se (X[I + 1] = Y[J + 1]) então
             I ← I + 1;
             J ← J + 1;
          senão
             TESTE ← Falso;
          fim se;
       fim enquanto;

       se (TESTE) então
          CASAMENTO_KMP ← I - M + 1;
          exit;
       fim se;

       se (J = 0) então
          I ← I + 1;
       senão
          J ← PADROES[J];
       fim se;

    fim enquanto;

    CASAMENTO_KMP ← 0;

fim função CASAMENTO_KMP;
```

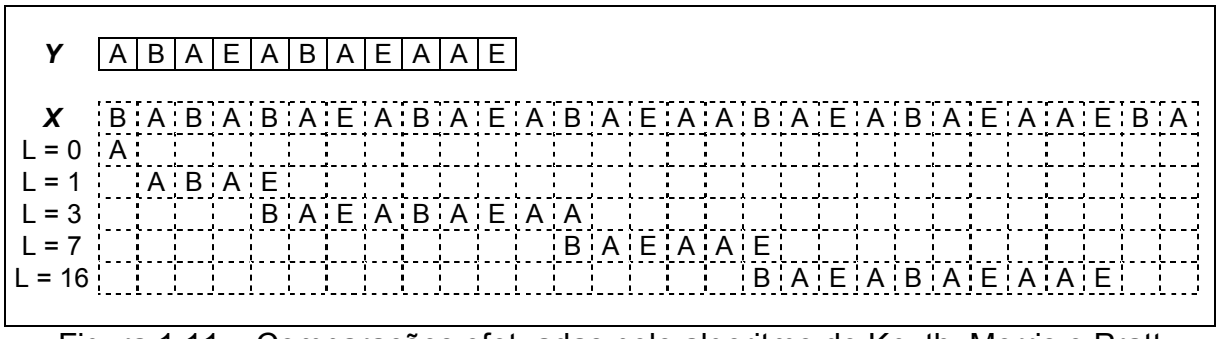


Figura 1.11 – Comparações efetuadas pelo algoritmo de Knuth, Morris e Pratt