

ORDEM E BUSCA

No mundo da computação, talvez nenhuma outra tarefa seja mais fundamental ou tão extensivamente analisada como estas de ordenação e busca. Estas rotinas são utilizadas em praticamente todos os programas de banco de dados, bem como em compiladores, interpretadores e sistemas operacionais. Neste capítulo são apresentados os conceitos básicos de ordenação e busca.

Pelo fato da ordenação de dados geralmente fazer uma busca mais fácil e rápida, a ordenação é discutida primeiro.

ORDEM

Ordenação é o processo de arranjar um conjunto de peças parecidas de informação numa ordem crescente e decrescente. Dada uma lista ordenada i de n elementos,

$$i_1 \leq i_2 \leq \dots \leq i_n$$

Existem duas categorias de algoritmos de classificação: a ordenação de vetores, ambos na memória ou em acessos randômicos a arquivos em disco, e a ordenação de arquivos em fita ou em disco. Este capítulo focalizará a primeira categoria porque é de maior interesse dos usuários de microcomputadores. No entanto, o método geral de ordenação seqüencial de arquivos também será comentado.

A principal diferença entre ordenação de vetores e ordenação seqüencial de arquivos é que cada elemento de um vetor está sempre disponível. Isto é, qualquer elemento pode ser comparado ou trocado com qualquer outro elemento a qualquer ins-

tante. Em um arquivo seqüencial, porém, somente um elemento está disponível num determinado momento.

Devido a esta diferença, técnicas de ordenação diferem enormemente entre as duas categorias.

Geralmente quando a informação é ordenada, uma pequena porção desta informação é usada como *chave de ordenação* com a qual as comparações são baseadas. Quando uma troca deve ser feita, toda a estrutura de dados é transferida. Em listagem de endereços, por exemplo, o campo de código de área (CEP) pode ser usado como chave, e o nome completo e endereço acompanham o CEP quando uma troca é feita. Com o objetivo de simplificar, exemplos de vários métodos de ordenação apresentados aqui são enfocados em ordenação de vetores caracteres. Mais tarde, você aprenderá como adaptar qualquer um destes métodos a qualquer tipo de estrutura de dados.

CLASSE DE ALGORITMOS DE ORDENAÇÃO

Existem três métodos gerais que podem ser usados para ordenar vetores:

- por troca
- por seleção
- por inserção

Imagine as cartas de um baralho. Para classificá-las *por troca*, você as espalhará, voltadas para cima, numa mesa, e então procederá a troca das cartas fora de ordem até que todo baralho esteja em ordem.

Para ordenação por *seleção*, você espalhará as cartas na mesa e escolherá a de menor valor, e a retirará do baralho. Então, das cartas restantes na mesa, você selecionará a menor, colocando-a atrás da já existente na sua mão. Você sempre escolhe a de menor valor restante na mesa; quando o processo for completado, as cartas em sua mão estarão ordenadas.

Na ordenação por *inserção*, você seguraria as cartas em sua mão, tirando uma por vez. Conforme você tirar as cartas do maço, as colocará em um novo maço sobre a mesa, inserindo-as sempre na posição correta. O maço estará ordenado quando não tiver mais cartas na mão.

UMA AVALIAÇÃO DOS ALGORITMOS DE ORDENAÇÃO

Existem muitos algoritmos para cada um dos três métodos de ordenação. Cada um deles tem seus méritos, mas, de uma forma geral, a avaliação de um algoritmo de ordenação está baseada nas respostas às seguintes perguntas:

- O quanto pode ser rápido, em média, um algoritmo de ordenação?
- Qual a velocidade de seu melhor e pior casos?
- Este algoritmo apresenta um comportamento *natural* ou *não-natural*?
- Ele rearranja elementos com chaves iguais?

Com que velocidade um algoritmo ordena para ter um ótimo desempenho? A velocidade na qual a ordenação de um vetor seja diretamente relacionada ao número de comparações e ao número de trocas exigidas, com as trocas exigindo mais tempo. Mais adiante neste capítulo você verá que algumas ordenações variam o tempo de ordenação de um elemento de forma exponencial, e algumas de forma logarítmica.

Os tempos de processamento no pior e melhor caso são importantes se você desejar saber regularmente quais as situações de melhor e pior caso. Frequentemente uma ordenação terá um bom caso médio mas um terrível pior caso, ou vice-versa.

Diz-se que uma ordenação tem um comportamento *natural* se ela trabalha o menos possível quando a lista já está ordenada, e quanto mais desordenada estiver a lista mais trabalho terá o algoritmo, e trabalhará o maior tempo quando a lista estiver em ordem inversa. O maior trabalho de uma ordenação é o número de comparações e movimentos que ele deve executar.

Para entender a importância de rearranjar elementos com chaves iguais, imagine um banco de dados que é ordenado de acordo com uma chave principal e uma subchave – por exemplo, uma lista postal com o código CEP como chave principal e o último nome com o mesmo código CEP como subchave. Quando um novo endereço for acrescentado à lista e a lista for novamente ordenada, você não desejará que as subchaves sejam novamente rearranjadas. Para garantir isto, a ordenação não poderá trocar chaves principais de mesmo valor.

Nas seções seguintes, são analisados e avaliados conforme sua eficiência cada um dos algoritmos de ordenação.

A Ordenação Bolha A mais conhecida (e mais difamada) ordenação é a *ordenação bolha*. Sua popularidade vem de seu nome fácil e de sua simplicidade. Porém, é uma das piores ordenações concebidas.

A *ordenação bolha* usa o método de ordenação por trocas. Faz repetidas comparações e, se necessário, troca elementos adjacentes. Seu nome provém da semelhança com a maneira como bolhas de água movimentam-se num tanque, onde cada bolha procura seu nível próprio. Neste mais simples exemplo de *ordenação bolha*

```

bubble(item,count)      /* ordenacao bolha */
char *item;
int count;
{
    register int a,b;
    register char t;

    for (a=1; a < count; ++a)
        for (b=count-1; b>=a; --b)
        {
            if (item [b-1] > item [b])
            {
                /* trocando os elementos */
                t = item [b-1];
                item [b-1] = item [b];
                item [b] = t;
            }
        }
}

```

item é um ponteiro a um vetor de caracteres a ser ordenado e **count** é o número de elementos no vetor.

A *ordenação bolha* é feita através de dois laços (loops). Uma vez que existem **count** elementos no vetor, o laço mais externo faz o vetor ser varrido **count-1** vezes. Isto assegura que, no pior caso, todos os elementos estarão na sua posição correta quando a função estiver terminada. O loop mais interno faz as comparações e trocas.

Esta versão de *ordenação bolha* pode ser usada para ordenar um vetor de caracteres em ordem ascendente. Por exemplo, este programa ordena uma seqüência de caracteres digitada:

```

main()    /* ordena caracteres digitados através do teclado */
{
    char s[80];
    int count;

    printf("entre um caracter:");
    gets(s);
    count=strlen(s);
    bubble(s,count);
    printf("cadeia de caracteres ordenados: %s",s);
}

```

Para ilustrar o funcionamento da *ordenação bolha*, aqui estão os passos para se ordenar o vetor **dcab**

início	d c a b
passo 1	a d c b
passo 2	a b d c
passo 3	a b c d

Na avaliação de qualquer ordenação, você deve determinar quantas comparações serão feitas, para o melhor, médio e pior caso. Para a *ordenação bolha*, o número de comparações é sempre o mesmo porque os dois laços (**for**) serão repetidos um número determinado de vezes, estando ou não a lista inicialmente ordenada. Isto significa que a *ordenação bolha* executará sempre $1/2(n^2-n)$ comparações, onde n é igual ao número de elementos a ser ordenado. A fórmula é derivada do fato de o laço mais externo da *ordenação bolha* ser executado $n-1$ vezes e o laço mais interno $n-2$ vezes. Multiplicando-se um pelo outro obtemos a fórmula.

O número de trocas é 0 para o melhor caso – para uma lista já ordenada. Os números são: para o caso médio $3/4(n^2-n)$ e para o pior caso $3/2(n^2-n)$. Está fora do objetivo deste livro explicar a origem destes casos; você pode observar, porém, que à medida que a lista torna-se menos ordenada, o número de elementos fora de ordem aproxima-se do número de comparações. (Existem três trocas para cada elemento fora de ordem na *ordenação bolha*.) Ela é um *algoritmo n-quadrado* pois seu tempo de execução é um múltiplo do quadrado do número de elementos. A *ordenação bolha* é muito ruim para um número grande de elementos porque o tempo de execução é diretamente proporcional ao número de comparações e trocas.

Por exemplo, se você ignorar o tempo que leva para trocar um elemento qualquer fora de ordem, verá que cada comparação leva 0.001 segundos, então para ordenar 10 elementos levará 0.05 segundos, ordenar 100 elementos levará 5 segundos, e ordenar 1000 elementos levará 500 segundos. Uma ordenação de 100.000 elementos, o tamanho de uma pequena lista telefônica, levaria em torno de 5.000.000 de segundos, ou 1:400 horas, por volta de dois meses de ordenação contínua! O gráfico da Figura 2.1 mostra como o tempo de execução aumenta em relação ao tamanho do vetor.

Você pode fazer umas melhorias para que a *ordenação bolha* fique mais rápida – e melhore um pouco a sua imagem. Por exemplo, a *ordenação bolha* tem uma peculiaridade: um elemento fora-de-ordem no “final longo”, tal como o **a** no vetor **dcab** dado como exemplo, irá para sua posição correta em um passo, mas um elemento desordenado no “final curto”, como o **d**, subirá vagarosamente para seu lugar apropriado. Em vez de sempre ler o vetor numa mesma direção, podemos alternar a direção entre passos consecutivos. A maior parte dos elementos fora-de-ordem irá mais rapidamente para suas posições corretas. É mostrada aqui uma versão de *ordenação bolha* chamada *ordenação chacoalhadeira*, por causa de seu movimento chacoalhador sobre o vetor:

```
shaker(item,count) /* ordenacao chacoalhadeira */
char *item;
int count;
{
    register int a,b,c,d;
    char t;

    c=1;
    b=count-1; d=count-1;

    do {
        for(a=d; a>=c; --a) {
            if(item[a-1]>item[a]) {
                t=item[a-1];
                item[a-1]=item[a];
                item[a]=t;
                b=a;
            }
        }
        c=b+1;
        for(a=c;a<d+1;++a) {
            if(item[a-1]>item[a]) {
```

```

        t=item[a-1];
        item[a-1]=item[a];
        item[a]=t;
        b=a;
    }
}
d=b-1;
} while (c<=d);
}

```

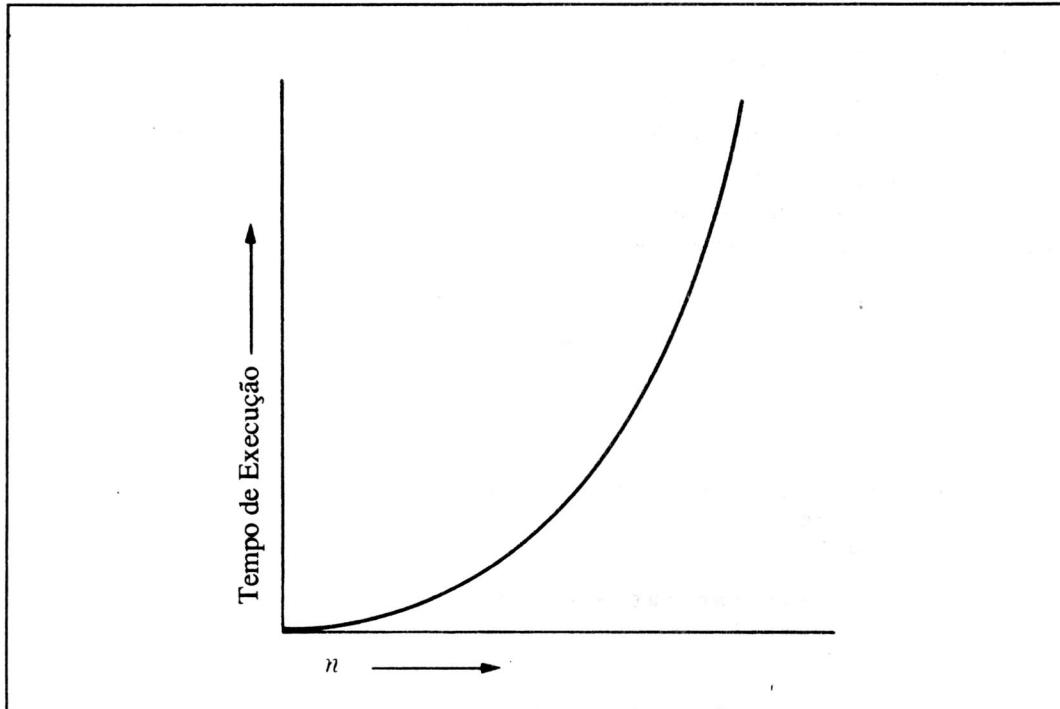


Figura 2.1. Tempo de processamento em relação ao tamanho do vetor numa ordenação n^2 .

Embora a ordenação “chacoalhadeira” melhore a *ordenação bolha*, ela ainda é executada na ordem de n^2 , porque o número de comparações não é alterado e porque o número de trocas foi reduzido de uma pequena constante.

Ordenação por Seleção A *ordenação por seleção* seleciona o elemento de menor valor e troca-o como primeiro elemento. Então, para os $n-1$ elementos restantes, o

elemento com o menor valor é encontrado e trocado com o segundo elemento, e assim por diante até os dois últimos. Por exemplo, se o método de seleção tivesse sido usado no vetor **bdac**, cada passo se apresentaria como:

início	b d a c
passo 1	a d b c
passo 2	a b d c
passo 3	a b c d

Uma ordenação por seleção simples é vista aqui:

```
select(item,count) /* ordenacao por selecao */
char *item;
int count;
{
    register int a,b,c;
    char t;

    for(a=0;a<count-1;++a) {
        c=a;
        t=item[a];
        for(b=a+1; b<count; ++b) {
            if(item[b]<t) {
                c=b;
                t=item[b];
            }
        }
        item[c]=item[a];
        item[a]=t;
    }
}
```

Infelizmente, como na *ordenação bolha*, o laço mais externo é executado $n-1$ vezes e o laço mais interno $1/2(n)$ vezes. Isto significa que a *ordenação por seleção* requer $1/2(n^2-n)$ comparações, que a tornam muito lenta para um número grande de itens. O número de trocas para o melhor caso é $3(n-1)$ e para o pior caso $n^2/4 + 3(n-1)$.

Para o melhor caso (a lista está ordenada) somente $n-1$ elementos precisam ser movimentados, e cada movimento requer três trocas. O pior caso aproxima-se do número de comparações. Não obstante o desenvolvimento do cálculo do caso médio esteja fora do objetivo deste livro, é igual a $n(\ln n + y)$, onde y é a constante de Euler, aproximadamente 0,577216. Isto significa que embora o número de comparações para a *ordenação bolha* e para a *ordenação por seleção* seja o mesmo, o número de trocas no caso médio é muito menor para a *ordenação por inserção*.

Ordenação por Inserção A *ordenação por inserção* é o último dos algoritmos simples. Inicialmente ela ordena os dois primeiros membros no vetor. A seguir, o algoritmo insere o terceiro membro na sua posição ordenada em relação aos dois primeiros membros. Então, o quarto elemento é inserido na lista de três elementos. O processo continua até que todos elementos tenham sido ordenados. Por exemplo, no vetor **dca b**, cada passo da *ordenação por inserção* seria como:

início	d c a b
passo 1	c d a b
passo 2	a c d b
passo 3	a b c d

A versão da *ordenação por inserção* é vista aqui:

```
insert(item,count) /* ordenacao por insercao */
char *item;
int count;
{
    register int a,b;
    char t;

    for(a=1; a<count; ++a) {
        t=item[a];
        b=a-1;
        while(b>=0 && t<item[b] ) {
            item[b+1]=item[b];
            b--;
        }
        item[b+1]=t;
    }
}
```

Ao contrário da *ordenação bolha* e da *ordenação por seleção*, o número de comparações que ocorrem enquanto a *ordenação por inserção* é usada depende a rigor de como a lista está inicialmente ordenada. Se a lista estiver em ordem, então o número de comparações é $n-1$. Se a lista estiver fora de ordem, então o número de comparações é $1/2(n^2+n)-1$, enquanto sua média é $1/4(n^2+n-2)$.

O número de trocas para cada caso é o seguinte:

melhor	$2(n-1)$
médio	$1/4(n^2+9n-10)$
pior	$1/2(n^2+3n-4)$

Como se vê, o número para o pior caso é tão ruim quanto aqueles para as ordenações *bolha* e *por seleção*, e este número para o caso médio é somente um pouco melhor.

A *ordenação por inserção* tem certamente duas vantagens. Em primeiro lugar comporta-se com *naturalidade*: trabalha menos quando o vetor já está ordenado e o máximo quando o vetor está na ordem inversa. Isto faz com que a *ordenação por inserção* seja útil para listas que estão quase ordenadas. Em segundo, não rearranja elementos de mesma chave: se uma lista é ordenada com duas chaves, mantém-se ordenada para ambas as chaves após uma *ordenação por inserção*.

ORDENAÇÕES MELHORES

Os algoritmos vistos até agora tiveram a grave falha de serem processados numa grandeza de tempo n^2 . Para um grande número de dados, a ordenação será lenta – em certos casos, muito lenta para sua utilização. Todo programador já ouviu ou contou a estória da “ordenação que levou três dias”. Infelizmente, estas estórias freqüentemente são verdadeiras.

Quando uma ordenação demora tanto tempo, deve ser uma falha do algoritmo básico. Porém, o pior é que a primeira idéia freqüentemente é “vamos escrevê-lo em assembler”. Embora o assembler aumente quase sempre a velocidade de uma constante, se o algoritmo básico for ruim, a ordenação ainda será lenta, não importando o quão bom seja o código. Lembre-se: quando o tempo de uma rotina é n^2 , aumentar a velocidade do código ou do computador apenas causa uma melhoria marginal, pois a razão na qual o tempo de execução aumenta é alterada exponencialmente (o gráfico da Figura 2.1 é deslocado para a direita, levemente, mas a curva continua a mesma). Tenha em mente que se alguma coisa não for rápida o bastante em C, não o será em assembler. A solução é usar um algoritmo de ordenação melhor.

Ordenação Shell A *ordenação Shell* é assim chamada por causa do seu inventor, D.L.Shell. Porém, o nome parece ter sido apropriado porque seu método de operação lembra as conchas do mar empilhadas umas sobre as outras.

O método geral, derivado da ordenação por inserção, está baseado em *diminishing increments*. A Figura 2.2 mostra o diagrama da *ordenação Shell* no vetor **fdacbe**. Primeiro, todos os elementos que estão três posições afastados são ordenados. Então todos os elementos que estão duas posições afastados são ordenados. Finalmente, todos aqueles adjacentes um ao outro são ordenados.

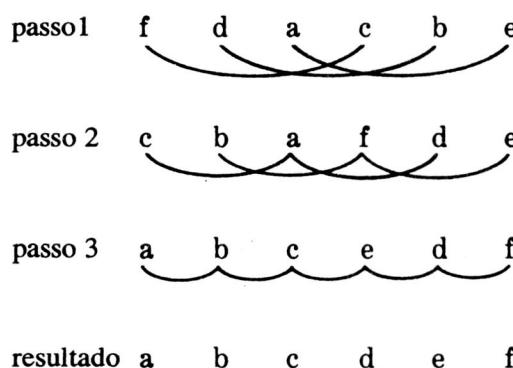


Figura 2.2. A ordenação Shell.

Não parece óbvio que esse método leve a bons resultados, ou mesmo que irá ordenar um vetor, mas ele executa ambas as funções. Este algoritmo é eficiente porque cada passo da ordenação envolve, ou relativamente poucos elementos, ou elementos que já estão em razoável ordem; contudo, cada passo aumenta a ordenação dos dados.

A seqüência exata para os incrementos pode ser alterada. A única regra é que o último incremento deve ser um. Por exemplo, a seqüência 9, 5, 3, 1 funciona e é usada na ordenação Shell mostrada aqui. Evite seqüências de potências de 2 porque, por razões matemáticas complexas, elas reduzem a eficiência do algoritmo de ordenação. (Contudo, mesmo se você usá-las, a ordenação ainda funcionará.)

```

shell(item,count)
char *item;
int count;
{
    register int i,j,k,s,w;
    char x, a[5];
    a[0]=9; a[1]=5; a[2]=3; a[3]=3; a[4]=1;

    for(w=0; w<5; w++) {
        k=a[w]; s=-k;
        for(i=k; i<count; ++i) {
            x=item[i];
            j=i-k;
            if(s==0) { s=-k;
                        s++;
                        item[s]=x;
            }
            while(x<item[j] && j>=0 && j<=count) {
                item[j+k]=item[j];
                j=j-k;
            }
            item[j+k]=x;
        }
    }
}

```

Você deve ter observado que o laço interno **while** contém três condições de teste. O **x<item[j]** é a comparação necessária para o processo de ordenação. Os testes **j>=0** e **j<=count** são usados para evitar que os limites do array **item** sejam ultrapassados. Estas checagens extras degradarão até certo ponto a performance da ordenação Shell. Versões um pouco diferentes de ordenação Shell empregam elementos especiais, vetores chamados *sentinelas*, que não fazem parte do vetor a ser ordenado. Sentinelas carregam uma terminação particular que indica os elementos, menor e maior possível. Desta maneira, as checagens dos limites são desnecessárias. No entanto, usar sentinelas requer um conhecimento dos dados, o que limita a generalização da função de ordenação. A análise da ordenação Shell apresenta problemas matemáticos difíceis que estão fora dos objetivos deste livro. O tempo de processamento é proporcional a $n^{1.2}$ para uma ordenação de n elementos. Esta é uma redução significante sobre as ordenações n^2 da seção anterior; observe a Figura 2.3 que mostra o gráfico da curva n^2 junto com a curva $n^{1.2}$. Porém, antes que você se decida a usar a *ordenação Shell*, deverá saber que a *ordenação Quicksort* é ainda melhor.

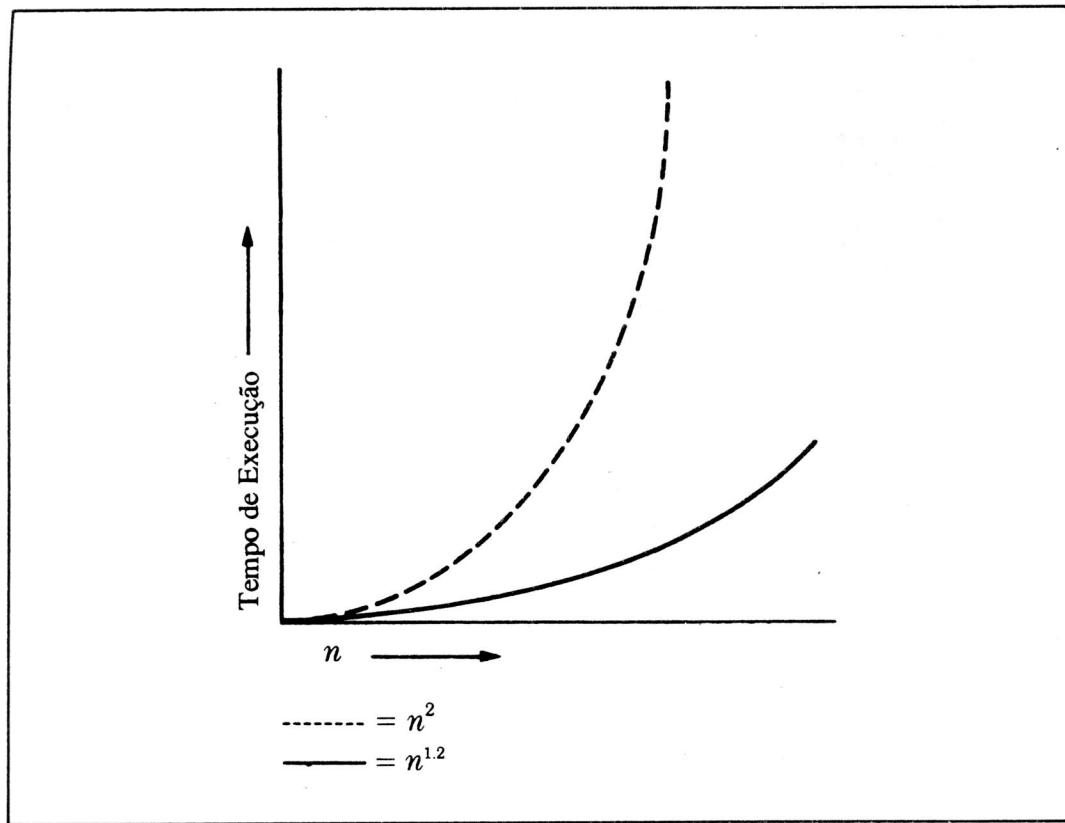


Figura 2.3. As curvas n^2 e $n^{1.2}$.

Ordenação Rápida – Quicksort A *Quicksort*, criada e denominada por C.A.R.Hoare, é geralmente considerada o melhor algoritmo de ordenação, normalmente avaliado. Está baseada no método de ordenação por trocas. Isto é surpreendente se você considerar a péssima performance da *ordenação bolha*, que também está baseada no método de trocas.

A *Quicksort* é construída sobre a idéia das partições. O procedimento mais usual seleciona um valor chamado de *comparador* e então divide o vetor em duas partes, com todos os elementos maiores e iguais ao valor de partição do outro. Esse processo é repetido para cada parte restante até que o vetor é ordenado. Por exemplo, dado o vetor **fedacb** e o valor **d** para partição, o primeiro passo da *Quicksort* rearranjará o vetor como segue:

início	f e d a c b
passo 1	b c a d e f

Este processo é repetido para cada parte (**bca** e **def**). O processo é essencialmente recursivo; certamente, as mais claras implementações da *Quicksort* são algoritmos recursivos.

A escolha do valor do comparador mediano pode ser feita de duas maneiras. O valor pode ou ser escolhido randomicamente ou por média dos valores de um pequeno conjunto retirado do vetor. Para uma ordenação ideal é melhor escolher o valor que é precisamente o meio do escopo de valores. Porém, isto não é fácil para a maioria dos conjuntos de dados. Mesmo no pior caso – o valor escolhido é uma das extremidades – a *Quicksort* ainda assim tem um rendimento razoavelmente bom.

A versão de *Quicksort* a seguir seleciona o elemento médio do vetor. Embora esta forma nem sempre resulte numa boa escolha, é uma técnica simples e rápida, e funciona corretamente.

```
quick(item,count) /* quick sort */
char *item;
int count;
{
    qs(item,0,count-1);
}

qs(item,left,right)
char *item;
int left,right;
{
    register int i,j;
    char x,y;

    i=left; j=right;
    x=item[(left+right)/2];

    do {
        while(item[i]<x && i<right) i++;
        while(x<item[j] && j>left) j--;

        if(i<=j) {
            y=item[i];
            item[i]=item[j];
            item[j]=y;
            i++; j--;
        }
    } while(i<j);
}
```

```

    }
} while(i<=j);

if(left<j)  qs(item,left,j);
if(i<right)  qs(item,i,right);
}

```

Aqui `quick()` executa a chamada para a função principal de ordenação, chamada `gs()`. Enquanto isto mantém a mesma interface comum a `item` e `count`, não é essencial, porque `qs()` poderia ter sido chamada diretamente, usando-se três argumentos.

A demonstração de como o número de comparações e o número de trocas que a *Quicksort* executa requer um ambiente matemático fora do escopo deste livro. Porém, você pode assumir que o número de comparações é $n \log n$, e que o número de trocas é aproximadamente $n/6 \log n$. Eles são significativamente melhores do que qualquer uma das ordenações vistas até agora.

A equação

$$N = a^x$$

pode ser reescrita como

$$x = \log_a N$$

Isto quer dizer, por exemplo, que se 100 elementos fossem ordenados, *Quicksort* iria precisar em média de 100^2 , ou 200, comparações porque $\log 100$ é 2. Comparando com a média de 990 comparações da *ordenação bolha*, este número é muito bom.

No entanto, existe um estranho aspecto em *Quicksort* sobre o qual você deve ser advertido. Se acontecer de o valor do comparador para cada partição ser o valor máximo, então a *Quicksort* degenera-se em uma ordenação lenta com tempo n^2 . Geralmente, porém, isto não acontece.

Você deve tomar cuidado ao escolher um método para determinar o valor do comparador. Freqüentemente o valor é determinado pelo valor de um dado real que está sendo ordenado. Em grandes listas postais onde a ordenação é freqüentemente feita através dos códigos CEP, a seleção é simples, porque os códigos são distribuídos ordenadamente e uma função algébrica pode determinar o melhor comparador. Porém, em determinados bancos de dados, as chaves de ordenação possuem valores tão próximos, com muitos tendo o mesmo valor, que a seleção randômica é a melhor disponível. Um método co-

mum e satisfatoriamente eficaz é amostrar três elementos da partição e retirar o de valor médio.

ORDENANDO OUTRAS ESTRUTURAS DE DADOS

Até agora, as ordenações só têm sido aplicadas para vetores de caracteres, tornando mais fácil a apresentação de cada rotina de ordenação. Obviamente, vetores de qualquer dos tipos de dados intrínsecos podem ser ordenados simplesmente trocando-se os tipos de dados dos parâmetros e variáveis na função de ordenação. No entanto, são geralmente tipos de dados complexos como *strings* ou estruturas que precisam ser ordenados. (A maior parte das ordenações envolvem uma chave e informação enlaçadas em uma chave.) Para adaptar os algoritmos para ordenar outras estruturas de dados, você precisa trocar ou a parte da comparação ou a parte das trocas, ou ambas. O algoritmo permanecerá basicamente inalterado.

A ordenação *Quicksort* será usada em nossos exemplos por ser considerada uma das melhores rotinas de propósito geral disponíveis até o momento. As mesmas técnicas, no entanto, se aplicarão para qualquer uma das ordenações descritas anteriormente.

Ordenação de Seqüência de Caracteres A maneira mais fácil de ordenar seqüência de caracteres é criar um vetor de ponteiros a caractere para essas seqüências, possibilitando uma indexação fácil, e mantendo a base do algoritmo Quicksort inalterada. A versão para seqüências de caracteres da Quicksort mostrada aqui contém um vetor de ponteiros a caractere que apontam para as seqüências a serem ordenadas. A ordenação rearranja os ponteiros às seqüências, não as seqüências reais na memória. Esta versão ordena seqüências de caracteres em ordem alfabética.

O passo da comparação foi alterado para o uso da função `strcmp()`, que retorna um número negativo se a primeira seqüência de caracteres for lexograficamente menor que a segunda, 0 se as seqüências forem iguais, ou um número positivo se a primeira seqüência for lexograficamente maior que a segunda. A parte da troca da rotina foi deixada inalterada porque só os ponteiros são trocados – não as seqüências reais. Para trocar as seqüências reais, você teria de usar a função `strcpy()`.

```
quick_string(item, count)
char *item[];
int count;
{
```

```

    qs(item,0,count-1);

}

qs(item,left,right)
char *item[];
int left,right;
{
    register int i, j;
    char *x, *y;

    i=left; j=right;
    x=item[(left+right)/2];

    do {
        while(strcmp(item[i],x)<0 && i<right) i++;
        while(strcmp(item[j],x)>0 && j>left) j--;

        if(i<=j) {
            y=item[i];
            item[i]=item[j];
            item[j]=y;
            i++; j--;
        }
    } while(i<=j);

    if(left<j) qs(item,left,j);
    if(i<right) qs(item,i,right);
}

```

O uso da **strcmp()** retarda a ordenação por duas razões. Primeiro ela envolve uma função de chamada, que sempre leva tempo; segundo, a **strcmp** executa várias (e algumas vezes muitas) comparações para determinar a relação entre duas seqüências. Se a velocidade é realmente crítica, o código para **strcmp()** pode ser reescrito através de uma rotina particular mais rápida. No entanto, não existe nenhuma maneira de evitar a comparação entre seqüências de caracteres, desde que esta seja a definição que envolva a tarefa.

Ordenação de Estruturas A maioria dos programas de aplicação que requer ordenação precisará ter um grupo de dados ordenados. Uma lista postal é um excelente exemplo porque o nome, rua, cidade, Estado e código CEP estão todos ligados. Quando esta unidade conglomerada de dados é ordenada, uma chave de ordenação é utilizada, mas a estrutura toda é trocada. Para entender esse processo, você primeiro precisa criar uma estrutura. Para uma lista postal, por exemplo, uma estrutura conveniente é

```

struct address {
    char name[40];
    char street[40];
    char city[20];
    char state[3];
    char zip[10];
};

```

state tem extensão de três caracteres e o código CEP tem extensão de 10 caracteres porque um vetor de caracteres precisa ter extensão de um caractere a mais do que o comprimento máximo de qualquer seqüência de caracteres no sentido de armazenar uma terminação nula.

Assim é razoável arranjar uma mala postal como um vetor de estrutura; assuma para este exemplo que a rotina ordena um vetor de estruturas do tipo **address** pelo código ZIP, como mostrado aqui:

```

quick_structure(item,count) /* ordenacao de estruturas */

struct address item[];
int count;
{
    qs(item,0,count-1);
}

qs(item,left,right)
struct address item[];
int left,right;
{
    register int i,j;
    char *x,*y;

    i=left; j=right;
    x=item[(left+right)/2].zip;

    do {
        while(strcmp(item[i].zip,x)<0 && i<right) i++;
        while(strcmp(item[j].zip,x)>0 && j>left) j--;
        if(i<=j) {
            swap_all_fields(item,i,j);
            i++; j--;
        }
    }
}

```

```
    }while(i<=j);

    if(left<j) qs(item,left,j);
    if(i<right) qs(item,i,right);

}
```

Observe que tanto o código de comparação como o código de troca precisaram ser alterados. Em razão de muitos campos precisarem ser alterados, uma função separada chamada **swap-all-fields()**, mostrada mais adiante, foi criada.

ORDENAÇÃO DE ARQUIVOS EM DISCOS

Existem dois tipos de arquivos em disco: *seqüenciais* e *de acesso randômico*. Se um arquivo em disco é pequeno o bastante, ele deve ser lido na memória, desta forma as rotinas de ordenação de vetores apresentadas anteriormente podem ordenar esses arquivos mais eficientemente. No entanto, muitos arquivos em disco são muito grandes para serem ordenados facilmente na memória e requerem técnicas especiais.

Ordenação de Arquivos em Disco por Acesso Randômico Utilizado pela maioria das aplicações de banco de dados em microcomputadores, arquivos em disco de acesso randômico têm duas vantagens sobre arquivos em disco seqüenciais. Primeiro, eles são de fácil manutenção, você pode atualizar informações sem ter que copiar toda a lista. Segundo, arquivos em disco de acesso randômico podem ser tratados como um grande vetor no disco, que simplifica bem a ordenação. Aplicar esse método significa que você pode usar a base da Quicksort com modificações para procurar diferentes registros no disco, em vez de ter que indexar um vetor. Diferente de ordenar arquivos em disco seqüenciais, ordenar um arquivo randomicamente significa que um disco cheio não precisa ter espaço para arquivos ordenados e desordenados.

Cada situação de ordenação difere na estrutura de dados exata que é ordenada e na chave que é usada. Todavia, o conceito geral de ordenação de arquivos em disco de acesso randômico pode ser compreendido pelo desenvolvimento de programas de ordenação que ordenam uma estrutura de lista postal, chamada **address**, que foi definida anteriormente. Este programa exemplo assume que o número de elementos é fixado em 100. Mas em aplicações reais, o contador de registros teria de ser dinamicamente mantido. Novamente, a chave de ordenação é o campo de código CEP.

```
#include "stdio.h"
#define NUM_ELEMENTS 100           /* este numero é arbitrario,
                                o operador podera alterar de
                                acordo com a sua conveniencia */

struct address {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    char zip[10];
}

main()
{
    FILE *fp;
    int t;

    if((fp=fopen("mplist","r+"))==0)  [
        printf("arquivo inacessivel para escrita e leitura\n");
        exit(0);
    }

    quick_disk(fp,NUM_ELEMENTS);

    fclose(fp);

    printf("Lista ordenada\n");
}

quick_disk(fp,count)      /* quick sort para arquivos randomicos */
FILE *fp;                  /* chamada */
long int count;
{
    qs_disk(fp,0,(long)count-1);
}

qs_disk(fp,left,right)   /* quick sort para arquivos randomicos */
FILE *fp;
long int left,right;
{
    long int i,j;
    char x[100],*y,*get_zip();

    i=left; j=right;

    strcpy(x,get_zip(fp,(long)(i+j)/2)); /* media */

    do  {
        while(strcmp(get_zip(fp,i),x)<0 && i<right) i++;
        while(strcmp(get_zip(fp,j),x)>0 && j>left) j--;
    }
}
```

```
        if(i<=j) {
            swap_all_fields(fp,i,j);
            i++; j--;
        }
    } while(i<=j);

    if(left<j) qs_disk(fp,left,j);
    if(i<right) qs_disk(fp,i,right);
}

swap_all_friends(fp,i,j)
FILE *fp;
long int i,j;
{
    char a[sizeof(ainfo)],b[sizeof(ainfo)];
    register int t;

    /* primeira leitura */
    fseek(fp,sizeof(ainfo)*i,0);
    for(t=0;t<sizeof(ainfo);++t) a[t]=getc(fp);

    fseek(fp,sizeof(ainfo)*j,0);
    for(t=0;t<sizeof(ainfo);++t) b[t]=getc(fp);

    /* escrita */
    fseek(fp,sizeof(ainfo)*j,0);
    for(t=0;t<sizeof(ainfo);++t) putc(a[t],fp);

    fseek(fp,sizeof(ainfo)*i,0);
    for(t=0;t<sizeof(ainfo);++t) putc(b[t],fp);
}

char *get_zip(fp,rec)
FILE *fp;
long int rec;
{
    char *p;
    register int t;

    p=&ainfo;

    fseek(fp,rec*sizeof(ainfo),0);
    for(t=0;t<sizeof(ainfo);++t) *p++=getc(fp);

    return ainfo.zip;
}
```

Muitas funções de suporte têm sido escritas para ordenar os registros de endereços. Na parte da comparação da ordenação, **get-zip** retorna um ponteiro para o código ZIP do comparador e o registro é conferido. A função **swap-all-fields()** executa a troca real de dados. Para a maioria dos sistemas operacionais, os pedidos de leitura e escrita causam um grande impacto sobre a velocidade de ordenação. O código, como ele é escrito, força uma busca ao registro **i** e então ao **j**. Enquanto o controlador do acionador de disco está ainda posicionado em **j**, o dado de **i** é escrito. Isto significa que não é necessário para o controlador movimentar-se em grandes distâncias. O dado de **i** terá sido escrito primeiro, e, então, uma nova busca se fará necessária.

Ordenação de Arquivos Seqüenciais Ao contrário dos arquivos de acesso randômico, arquivos seqüenciais geralmente não têm tamanho fixo para registros e podem ser organizados em periféricos de armazenagem que não permitem acesso randômico fácil. Portanto, arquivos em disco seqüenciais são comuns porque em uma aplicação específica adaptam-se melhor a comprimento de registros variáveis ou porque o periférico de armazenamento é seqüencial por natureza. Por exemplo, a maioria dos arquivos de texto é seqüencial.

Embora ordenar arquivos em disco como se eles fossem vetores tenha muitas vantagens, este método não pode ser utilizado com arquivos seqüenciais – não existe maneira de se ter um acesso rápido a um elemento arbitrário qualquer. Por exemplo, não existe nenhuma maneira rápida de buscar registros arbitrários de um arquivo seqüencial gravado em uma fita. Portanto, seria difícil apresentar qualquer dos algoritmos de ordenação de vetores anteriormente descritos para os arquivos seqüenciais.

Existem dois métodos de ordenação de arquivos seqüenciais. O primeiro lê a informação na memória e ordena-a com um dos algoritmos usuais de ordenação de vetores. Embora este seja um método rápido, a memória limita o tamanho do arquivo que pode ser ordenado.

O segundo método, chamado de *ordenação agrupada*, divide o arquivo a ser ordenado em dois arquivos de mesmo comprimento. Usando esses arquivos, a ordenação lê um elemento de cada arquivo, ordena esse par e escreve os elementos em terceiro arquivo em disco. Esse novo arquivo é então dividido, e as duplas ordenadas são unidas em quádruplos ordenados. O novo arquivo é dividido novamente, e o mesmo procedimento prossegue até que a lista esteja ordenada. Por razões históricas, esta ordenação conjunta é chamada de *three-tape merge* porque requer três arquivos (acionadores de fita) cada vez que é utilizado.

Para entender como a *ordenação agrupada* trabalha, considere a seguinte seqüência:

1 4 3 8 6 7 2 5

A primeira divisão produz

1 4 3 8
6 7 2 5

O primeiro agrupamento fornece

1 6 – 4 7 – 2 3 – 5 8

Dividindo-se novamente torna-se

1 6 – 4 7
2 3 – 5 8

O próximo agrupamento dá

1 2 3 6 – 4 5 7 8

A divisão final é

1 2 3 6
4 5 7 8

que resulta

1 2 3 4 5 6 7 8

Como você já deve ter imaginado, o “agrupamento de três fitas” (*tree-tape merge*) requer que cada arquivo seja acessado $\log_2 n$ vezes, onde n é o número total de elementos a serem ordenados.

Temos aqui uma versão simples de ordenação por agrupamento. Esta ordenação assume que o arquivo de entrada é um vetor de caracteres, tal qual um arquivo texto, e que o arquivo possui comprimento de potência par. Você pode facilmente alterar esta versão para ordenar qualquer tipo de arquivo de dados

```
#include "stdio.h"

#define LENGTH 16 /* arbitrario */

main(argc,argv)
int argc;
char *argv[];
{

    FILE *fp1,*fp2,*fp3;

    if((fp1=fopen(argv[1],"rw"))==0) {
        printf("arquivo 1 inacessivel %s\n",argv[1]);
        exit(0);
    }

    if((fp2=fopen("sort1","rw"))==0) {
        printf("arquivo 2 inacessivel \n");
        exit(0);
    }

    if((fp3=fopen("sort2","rw"))==0) {
        printf("arquivo 3 inacessivel \n");
        exit(0);
    }

    merge(fp1,fp2,fp3,LENGTH);

    fclose(fp1);  fclose(fp2);  fclose(fp3);
}

merge(fp1,fp2,fp3,count)
FILE *fp1,*fp2,*fp3;
int count;
{
    register int t,n,j,k,q;
    char x,y;

    for(n=1;n<count;n=n*2) {

        for(t=0;t<count/2;++t) putc(getc(fp1),fp2);
        for(;t<count;++t) putc(getc(fp1),fp3);

        rewind(fp1,fp2,fp3);

        for(q=0;q<count/2;q+=n) {
            x=getc(fp2);
            y=getc(fp3);
            for(j=k=0;;) {
                if(x<y) {
```

```
        putc(x,fp1);
        j++;
        if(j<n) x=getc(fp2);
        else break;
    }
    else {
        putc(y,fp1);
        k++;
        if(k<n) y=getc(fp3);
        else break;
    }
}
if(j<n) {
    putc(x,fp1);
    j++;
}
if(k<n) {
    putc(y,fp1);
    k++;
}
for(;j<n;++j) putc(getc(fp2),fp1);
for(;k<n;++k) putc(getc(fp3),fp1);
}
rewind(fp1,fp2,fp3);
}

rewind(fp1,fp2,fp3)
FILE *fp1,*fp2,*fp3;
{
    fseek(fp1,(long)0,0);
    fseek(fp2,(long)0,0);
    fseek(fp3,(long)0,0);
}
```

Os três arquivos foram abertos para modo **leitura/escrita**, e **rewind()** foi criada para desmontar os arquivos a cada vez. A atribuição **long** é usada com **fseek** porque esta função geralmente requer um inteiro longo para o deslocamento do arquivo.

BUSCA

Bancos de dados funcionam normalmente da seguinte forma: de tempos em tempos, o usuário pode localizar e utilizar o dado de um registro qualquer, tão logo a chave do registro seja conhecida. Existe somente um método de encontrar informação em um arquivo ou vetor desordenado e em um arquivo ou vetor ordenado.

MÉTODOS DE BUSCA

Encontrar informações em um vetor desordenado requer uma busca seqüencial, começando pelo primeiro elemento e parando ou quando o elemento procurado é encontrado ou quando o fim do vetor é alcançado.

Este método deve ser usado em dados desordenados mas também pode ser aplicado para dados ordenados. Se o dado tiver sido ordenado, então a busca binária pode ser usada, o que irá aumentar a velocidade de qualquer busca.

A Busca Seqüencial A busca seqüencial é fácil de ser codificada. A função a seguir faz uma busca em um vetor de caracteres de comprimento conhecido até encontrar o elemento procurado a partir de uma chave específica:

```
sequential_search(item, cout, key)
char *item;
int count;
char key;
{
    register int t;

    for(t=0; t<count; ++t)
        if(key==item[t]) return t;

    return -1;
}
```

Esta função retorna ou o número de indexação da entrada encontrada se existir um, ou -1 se não houver.

Uma ordenação seqüencial clássica testará em média $1/2n$ elementos. No melhor caso, testará somente um elemento e, no pior caso, n elementos. Se a informação está armazenada em disco, o tempo de busca pode ser muito longo. Mas se os dados estiverem desordenados, a busca seqüencial é o único método disponível.

Busca Binária Se o dado a ser encontrado estiver colocado de forma ordenada, então um método superior, chamado *busca binária*, poderá ser usado para encontrar o elemento procurado. O método utiliza “divisão e conferência”. Ele primeiro testa o elemento médio; se o elemento é maior do que a chave, ele testa o elemento mediano da primeira metade; caso contrário, ele testa o elemento médio da segunda metade. Esse processo é repetido até que o elemento procurado seja encontrado ou até que não haja mais elementos para testar. Por exemplo, para encontrar o número 4 no vetor **1 2 3 4 5 6 7 8 9**, a busca binária testaria primeiro o elemento mediano, que é o **5**. Como este elemento é maior do que 4, a busca continuaria com a primeira metade ou

1 2 3 4 5

Neste exemplo, o elemento médio é o **3**, e é menor do que 4, então a primeira metade é descartada, e a busca continua com

4 5

Neste momento, o elemento procurado é encontrado.

Na busca binária, o número de comparações no pior caso é $\log_2 n$. Para o caso médio, o número é um pouco melhor; no melhor caso o número é 1.

Você pode usar a seguinte função de busca binária para vetores caracteres para encontrar qualquer estrutura de dados arbitrária, trocando a parte de comparação da rotina:

```
bsearch(item,count,key) /* busca binaria */
char *item;
int count;
char key;
{
    int low,high,mid;
    low=0; high=count-1;

    while(low<=high) {
        mid=(low+high)/2;
        if(key<item[mid]) high=mid-1 ;
        else if(key>item[mid]) low=mid+1 ;
        else return mid; /* achou */

    }
    return -1 ;
}
```

O próximo capítulo explora diferentes métodos de armazenamento e recuperação de dados, que, em alguns casos, podem tornar as tarefas de busca e ordenação muito mais fáceis.