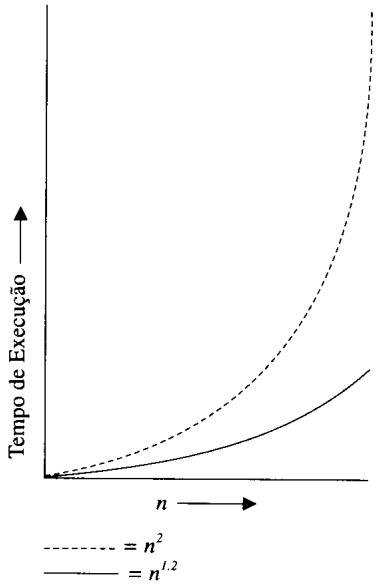

Figura 19.3 As curvas n^2 e $n^{1.2}$.

Quicksort

A quicksort, inventada e denominada por C.A.R. Hoare, é superior a todas as outras ordenações deste livro, e geralmente é considerada o melhor algoritmo de ordenação de propósito geral atualmente disponível. É baseada no método de ordenação por trocas. Isso é surpreendente, quando se considera o terrível desempenho da ordenação bolha!

A quicksort é baseada na idéia de partições. O procedimento geral é selecionar um valor, chamado de *comparando*, e, então, fazer a partição da matriz em duas seções, com todos os elementos maiores ou iguais ao valor da partição de um lado e os menores do outro. Esse processo é repetido para cada seção restante até que a matriz esteja ordenada. Por exemplo, dada a matriz **fedacb** e usando o valor **d** para a partição, o primeiro passo da quicksort rearranja a matriz como segue:

início	f e d a c b
passo1	b c a d e f

Esse processo é, então, repetido para cada seção — isto é, **bca** e **def**. Como você pode ver, o processo é essencialmente recursivo por natureza e, certamente, as implementações mais claras da quicksort são algoritmos recursivos.

Você pode selecionar o valor do comparando intermediário de duas formas. Você pode escolhê-lo aleatoriamente ou selecioná-lo fazendo a média de um pequeno conjunto de valores retirado da matriz. Para uma ordenação ótima, você deveria selecionar um valor que estivesse precisamente no centro da faixa de valores. Porém, isso não é fácil para a maioria dos conjuntos de dados. No pior caso, o valor escolhido está em uma extremidade e, mesmo nesse caso, quicksort ainda tem um bom rendimento. A versão seguinte da quicksort seleciona o elemento intermediário da matriz. Embora isso nem sempre resulte em uma boa escolha, a ordenação ainda é efetuada corretamente.

```
/* Função de inicialização da Quicksort. */
void quick(char *item, int count)
{
  qs(item, 0, count-1)
}

/* A Quicksort. */
void qs(char *item, int left, int 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);

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

Nessa versão, a função **quick()** executa a chamada à função da ordenação principal **qs()**. Isso permite manter a interface comum com **item** e **count**, mas não é

essencial porque **qs()** poderia ter sido chamada diretamente, usando-se três argumentos.

A dedução do número de comparações e de trocas que quicksort realiza requer uma matemática fora do âmbito deste livo. Porém, o número médio de comparações é

$n \log n$

e o número médio de trocas é aproximadamente

$n/6 \log n$

Esses números são significativamente menores do que aqueles vistos até agora para qualquer ordenação.

No entanto, existe um aspecto particularmente problemático de quicksort sobre o qual você deve ser advertido. Se o valor do comparando, para cada partição, for o maior valor, então a quicksort se degenerará em uma ordenação lenta com um tempo de procesamento n. Geralmente, porém, isso não acontece.

Você deve escolher com cuidado um método para definir o valor do comparando. O método é freqüentemente determinado pelo tipo de dado que está sendo ordenado. Em listas postais muito grandes, onde a ordenação é freqüentemente feita por meio do código CEP, a seleção é simples, porque os códigos são razoavelmente distribuídos — e uma simples função algébrica pode determinar um comparando adequado. Porém, em certos bancos de dados, as chaves podem ser iguais ou muito próximas em valor e uma seleção randômica é freqüentemente a melhor. Um método comum e satisfatório é tomar uma amostra com três elementos de uma partição e utilizar o valor médio.

Escolhendo uma Ordenação

Geralmente, quicksort é a melhor ordenação em virtude da sua velocidade. Porém, quando apenas listas muito pequenas de dados devem ser ordenadas (menos que 100), o tempo extra criado pelas chamadas recursivas do quicksort pode compensar os benefícios de um algoritmo superior. Em casos muito raros como esse, uma das ordenações mais simples — talvez até mesmo a ordenação bolha — pode ser mais rápida.