AULA 6

ORDENAÇÃO POR INTERCALAÇÃO

Na aula 5 revimos os métodos de ordenação mais básicos, que são todos iterativos, simples e têm tempo de execução de pior caso proporcional a n^2, onde n é o tamanho da entrada. Métodos mais eficientes de ordenação são baseados em recursão, técnica introduzida na aula 1. Nesta aula, estudamos o método da ordenação por intercalação, conhecido como *mergesort*.

A ordenação por intercalação, que veremos nesta aula, e a ordenação por separação, que veremos na aula 7, são métodos eficientes baseados na técnica recursiva chamada **dividir para conquistar**, onde quebramos o problema em vários subproblemas de menor tamanho que são similares ao problema original, resolvemos esses subproblemas recursivamente e então combinamos essas soluções para produzir uma solução para o problema original. Esa aula é baseada no livro de P. Feofiloff [2] e no livro de Cormen et. al [1].

6.1 Dividir para conquistar

A técnica de dividir para conquistar é uma técnica geral de construção de algoritmos e programas, tendo a recursão como base, que envolve três passos em cada nível da recursão:

Dividir o problema em um número de subproblemas;

Conquistar os subproblemas solucionando-os recursivamente. No entanto, se os tamanhos dos subproblemas são suficientemente pequenos, resolva os subproblemas de uma maneira simples;

Combinar as soluções dos subproblemas na solução do problema original.

Como mencionamos na aula 4, o algoritmo da busca binária é um método de busca que usa a técnica de dividir para conquistar na solução do problema da busca. O método de ordenação por intercalação, que veremos nesta aula, e o método da ordenação por separação, que veremos na aula 7, também são algoritmos baseados nessa técnica.

6.2 Problema da intercalação

Antes de apresentar o método da ordenação por intercalação, precisamos resolver um problema anterior, que auxilia esse método, chamado de problema da intercalação. O problema da intercalação pode ser descrito de forma mais geral como a seguir: dados dois conjuntos crescentes A e B, com m e n elementos respectivamente, obter um conjunto crescente C a partir

de A e B. Variantes sutis desse problema geral podem ser descritas como no caso em que se permite ou não elementos iguais nos dois conjuntos de entrada, isto é, conjuntos de entrada A e B tais que $A \cap B \neq \emptyset$ ou $A \cap B = \emptyset$.

O problema da intercalação que queremos resolver aqui é mais específico e pode ser assim descrito: dados dois vetores crescentes $v[p..q-1]$ e $v[q..r-1]$, rearranjar $v[p..r-1]$ em ordem crescente. Isso significa que queremos de alguma forma intercalar os vetores $v[0..q-1]$ e $v[q..r-1]$. Nesse caso, à primeira vista parece que os vetores de entrada podem ter elementos em comum. Entretanto, este não é o caso, já que estamos considerando o mesmo conjunto inicial de elementos armazenados no vetor v. Uma maneira fácil de resolver o problema da intercalação é usar um dos métodos de ordenação da aula 5 tendo como entrada o vetor $v[p..r-1]$. Essa solução, no entanto, tem consumo de tempo de pior caso proporcional ao quadrado do número de elementos do vetor e é ineficiente por desconsiderar as características dos vetores $v[p..q-1]$ e $v[q..r-1]$. Uma solução mais eficiente, que usa um vetor auxiliar, é mostrada a seguir.

```
/* Recebe os vetores crescentes v[p..q-1] e v[q..r-1]
   e rearranja v[p..r-1] em ordem crescente */
void intercala(int p, int q, int r, int v[MAX])
{
   int i, j, k, w[MAX];

   i = p;
   j = q;
   k = 0;
   while (i < q && j < r) {
      if (v[i] < v[j]) {
         w[k] = v[i];
         i++;
      }
      else {
         w[k] = v[j];
         j++;
      }
      k++;
   }
   while (i < q) {
      w[k] = v[i];
      i++;
      k++;
   }
   while (j < r) {
      w[k] = v[j];
      j++;
      k++;
   }
   for (i = p; i < r; i++)
      v[i] = w[i-p];
}
```

A função `intercala` tem tempo de execução de pior caso proporcional ao número de comparações entre os elementos do vetor, isto é, $r - p$. Assim, podemos dizer que o consumo de tempo no pior caso da função `intercala` é proporcional ao número de elementos do vetor de entrada.

6.3 Ordenação por intercalação

Com o problema da intercalação resolvido, podemos agora descrever uma função que implementa o método da ordenação por intercalação. Nesse método, dividimos ao meio um vetor v com $r - p$ elementos, ordenamos recursivamente essas duas metades de v e então as intercalamos. A função `mergesort` a seguir é recursiva e a base da recursão ocorre quando $p \geqslant r - 1$, quando não é necessário qualquer processamento.

```
/* Recebe um vetor v[p..r-1] e o rearranja em ordem crescente */
void mergesort(int p, int r, int v[MAX])
{
   int q;

   if (p < r - 1) {
      q = (p + r) / 2;
      mergesort(p, q, v);
      mergesort(q, r, v);
      intercala(p, q, r, v);
   }
}
```

Como a expressão $(p + q)/2$ da função `mergesort` é do tipo inteiro, observe que seu resultado é, na verdade, avaliado como $\lfloor \frac{p+q}{2} \rfloor$.

Observe também que para ordenar um vetor $v[0..n-1]$ basta chamar a função `mergesort` com os seguintes argumentos:

```
mergesort(0, n, v);
```

Vejamos um exemplo de execução da função `mergesort` na figura 6.1, para um vetor de entrada $v[0..7] = \{4, 6, 7, 3, 5, 1, 2, 8\}$ e chamada

```
mergesort(0, 8, v);
```

Observe que as chamadas recursivas são realizadas até a linha divisória imaginária ilustrada na figura, quando $p \geqslant r - 1$. A partir desse ponto, a cada volta de um nível de recursão, uma intercalação é realizada. No final, uma última intercalação é realizada e o vetor original torna-se então um vetor crescente com os mesmos elementos de entrada.

Qual o desempenho da função `mergesort` quando queremos ordenar um vetor $v[0..n-1]$? Suponha, para efeito de simplificação, que n é uma potência de 2. Se esse não é o caso, podemos examinar duas potências de 2 consecutivas, justamente aquelas tais que $2^{k-1} < n \leqslant 2^k$, para algum $k \geqslant 0$. Observe então que o número de elementos do vetor é diminuído a aproximadamente metade a cada chamada da função `mergesort`. Ou seja, o número aproximado de chamadas é proporcional a $\log_2 n$. Na primeira vez, o problema original é reduzido a dois subproblemas onde é necessário ordenar os vetores $v[0..\frac{n}{2} - 1]$ e $v[\frac{n}{2}..n - 1]$. Na segunda vez, cada

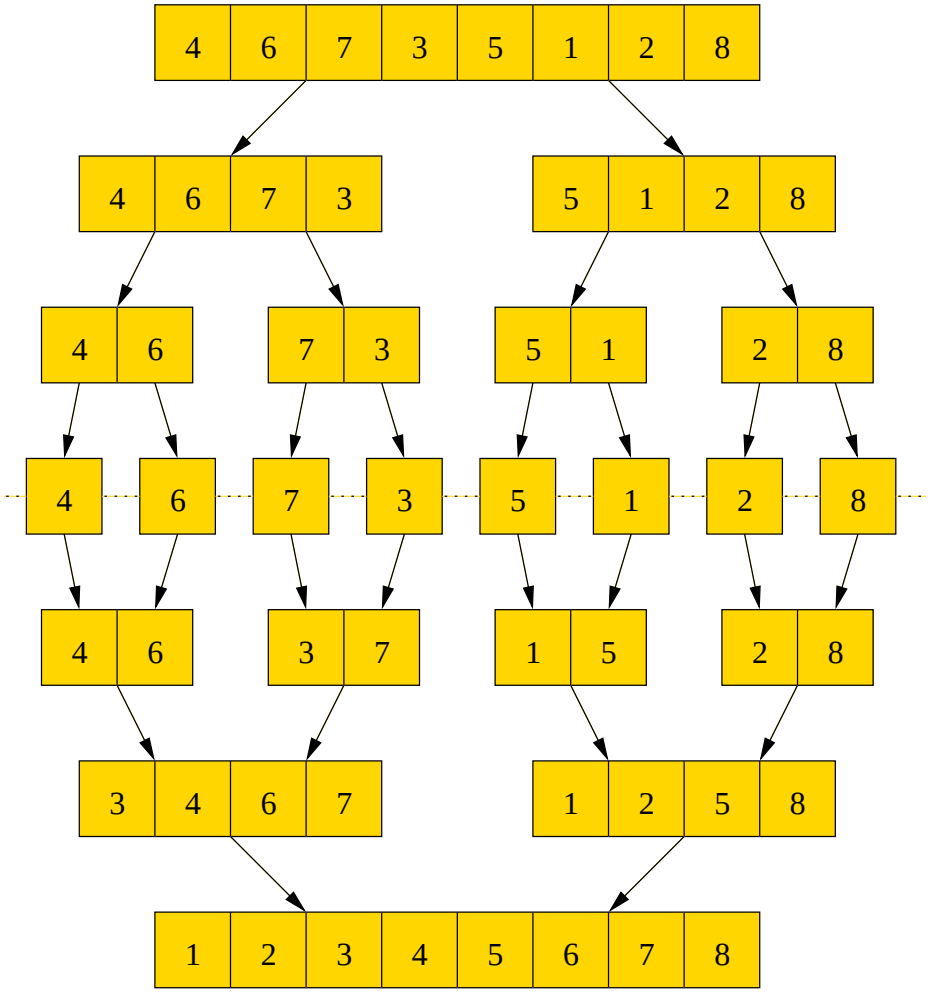

Figura 6.1: Exemplo de execução da ordenação por intercalação.

um dos subproblemas são ainda divididos em mais dois subproblemas cada, gerando quatro subproblemas no total, onde é necessário ordenar os vetores $v[0..\frac{n}{4}-1]$, $v[\frac{n}{4}..\frac{n}{2}-1]$, $v[\frac{n}{2}..\frac{3n}{4}-1]$ e $v[\frac{3n}{4}..n-1]$. E assim por diante. Além disso, como já vimos, o tempo total que a função `intercala` gasta é proporcional ao número de elementos do vetor v, isto é, $r-p$. Portanto, a função `mergesort` consome tempo proporcional a $n \log_2 n$.

Exercícios

6.1 Simule detalhadamente a execução da função `mergesort` sobre o vetor de entrada $v[0..7] = \{3, 41, 52, 26, 38, 57, 9, 49\}$.

6.2 A função `intercala` está correta nos casos extremos $p = q$ e $q = r$?

6.3 Um algoritmo de intercalação é **estável** se não altera a posição relativa dos elementos que têm um mesmo valor. Por exemplo, se o vetor tiver dois elementos de valor 222, um algoritmo de intercalação estável manterá o primeiro 222 antes do segundo. A função

`intercala` é estável? Se a comparação `v[i] < v[j]` for trocada por `v[i] <= v[j]` a função fica estável?

6.4 O que acontece se trocarmos `(p + r)/2` por `(p + r - 1)/2` no código da função `mergesort`? Que acontece se trocarmos `(p + r)/2` por `(p + r + 1)/2`?

6.5 Escreva uma versão da ordenação por intercalação que rearranje um vetor $v[p..r-1]$ em ordem decrescente.

6.6 Escreva uma função eficiente que receba um conjunto S de n números reais e um número real x e determine se existe um par de elementos em S cuja soma é exatamente X.

6.7 Agora que você aprendeu o método da ordenação por intercalação, o problema a seguir, que já vimos na aula 2, exercício 2.10, fica bem mais fácil de ser resolvido.

Seja A um vetor de n números inteiros distintos. Se $i < j$ e $A[i] > A[j]$ então o par (i, j) é chamado uma **inversão** de A.

(a) Liste as cinco inversões do vetor $\{2, 3, 8, 6, 1\}$.

(b) Qual vetor com elementos do conjunto $\{1, 2, \ldots, n\}$ tem o maior número de inversões? Quantas são?

(c) Qual a relação entre o tempo de execução da ordenação por inserção e o número de inversões em um vetor de entrada? Justifique sua resposta.

(d) Modificando a ordenação por intercalação, escreva uma função eficiente, com tempo de execução $O(n \log n)$, que determine o número de inversões em uma permutação de n elementos.

6.8 Escreva um programa para comparar experimentalmente o desempenho da função `mergesort` com o das funções `trocas_sucessivas`, `selecao` e `insercao` da aula 5. Use um vetor com números (pseudo-)aleatórios para fazer os testes.

6.9 Veja animações dos métodos de ordenação que já vimos nas seguintes páginas:

- Sort Animation de R. Mohammadi;
- Sorting Algorithms de J. Harrison;
- Sorting Algorithms de P. Morin;
- Sorting Algorithms Animations de D. R. Martin.