Construção e Análise de Algoritmos

aula 05: O problema da ordenação

1 Introdução

Nas últimas três aulas nós obtivemos três algoritmos diferentes para o problema da ordenação, todos eles com tempo de execução $O(n \log n)$.

Nesse ponto, é natural imaginar que isso é o melhor possível.

Para responder essa pergunta, considere a seguinte historinha.

Imagine que você tem a tarefa de colocar uma coleção de números em chinês em ordem crescente.

Você não sabe chinês, mas você tem um amigo chinês.

Você pode apresentar pares de números para ele e perguntar qual é o maior.

Quantas perguntas você precisa fazer? (no pior caso)

2 Limite inferior para a ordenação

Os algoritmos de ordenação que vimos nas aulas anteriores são todos diferentes.

Isto é, todos eles aplicam a técnica da divisão de uma maneira diferente.

Mas, apesar disso, existe uma coisa comum entre eles.

Isto é, todos são baseados na seguinte operação

- comparar elementos de duas posições diferentes, e mover o menor para frente e o maior para trás (se necessário)

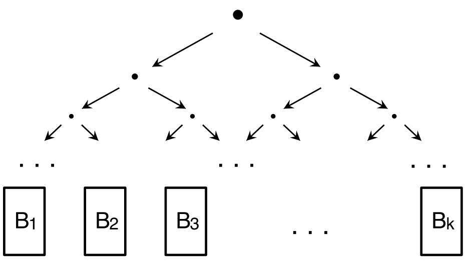
É nesse sentido que nós dizemos que todos eles são algoritmos de ordenação baseados em comparações.

A seguir, nós vamos argumentar que os algoritmos desse tipo precisam realizar um número mínimo de comparações da ordem de $n \log n$ para funcionar corretamente.

Para ver isso claramente, é útil separar a comparação dos elementos de sua movimentação na lista.

Isto é, nós podemos imaginar que o algoritmo primeiro faz todas as comparações necessárias, e só depois reorganiza os elementos na lista, com base nos resultados das comparações.

Um algoritmo desse tipo tem a seguinte estrutura:



A ideia aqui é que cada bloco de código B_i reorganiza os elementos da lista de uma maneira fixa, sem examinar os seus valores.

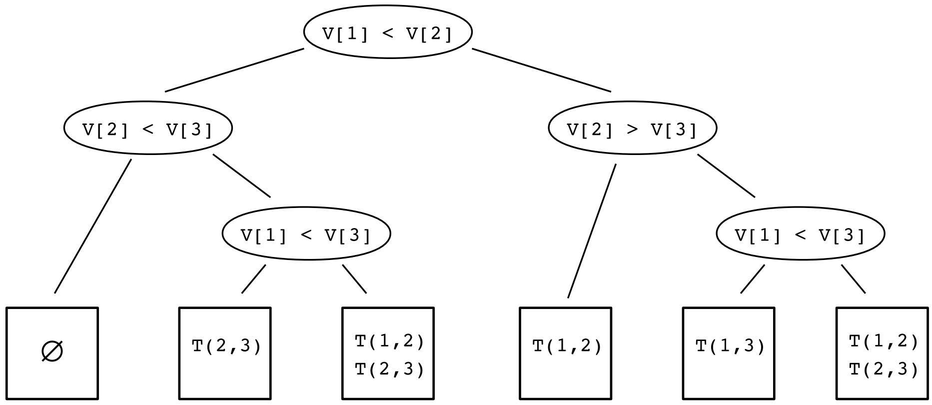
E as comparações que acontecem na primeira etapa servem justamente para determinar qual a reorganização que deve ser aplicada na lista para colocá-la em ordem crescente.

Vejamos o exemplo concreto de um algoritmo que ordena uma lista de tamanho 3.

```
Procedimento  ord-Comp3 ( V[1..3] )
 {
     Se ( V[1] < V[2] )
      {
        Se ( V[2] < V[3] )
         {
            // a lista já está em ordem crescente, não há nada a fazer
         }
        Senão  // V[2] é o maior de todos
         {
           Se ( V[1] < V[3] )
            {
               Troca (2,3)
            }
           Senão  // e V[3] é o menor de todos
            {
               Troca (2,3);   Troca (1,2)
      }  }  }
     Senão
      {
        Se ( V[2] > V[3] )
         {
            // a lista está em ordem decrescente, basta invertê-la
            Troca (1,3)
         }
        Senão  // V[2] é o menor de todos
         {
           Se ( V[1] < V[3] )
            {
```

```
                Troca (1,2)
             }
            Senão  // e V[1] é o maior de todos
             {
                Troca (1,2);   Troca (2,3)
}    }  }  }
```

Abaixo nós temos a representação esquemática desse algoritmo



A primeira observação importante aqui é que não é uma coincidência que existam 6 blocos de código na parte de baixo do esquema.

Uma lista de tamanho 3 pode estar em 6 configurações diferentes

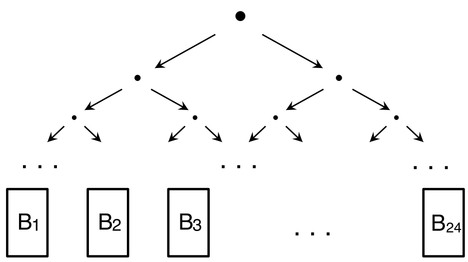
```
< 1, 2, 3 >        < 1, 3, 2 >        < 2, 3, 1 >

< 2, 1, 3 >        < 3, 1, 2 >        < 3, 2, 1 >
```

E, para cada uma dessas configurações, existe uma reorganização diferente que deve ser aplicada na lista para colocá-la em ordem.

Uma lista de tamanho 4 pode estar em $4 \times 6 = 24$ configurações diferentes

- qualquer um dos 4 elementos pode estar na primeira posição
- e os outros 3 elementos podem estar em 6 configurações diferentes

Portanto, um algoritmo que ordena uma lista de tamanho 4 deve ter 24 blocos de código na parte de baixo.

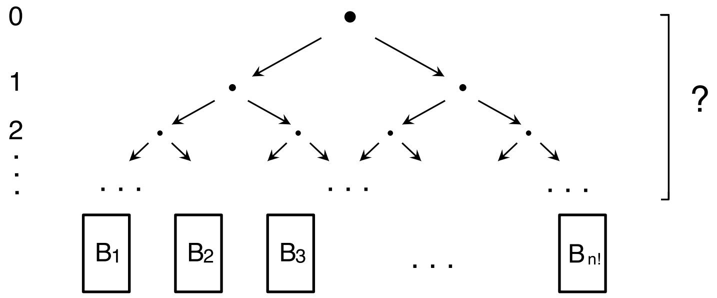


Em geral, uma lista de tamanho n pode estar em $n!$ configurações diferentes.

Logo, um algoritmo que ordena uma lista de tamanho n deve ter $n!$ blocos de código na parte de baixo.

A questão que nos interessa agora é a seguinte:

- *que altura deve ter a árvore de comparações para que a parte de baixo do esquema possa ter $n!$ blocos de código diferentes?*



A resposta é simples.

Observe que

- o nível 1 da árvore tem $2^1 = 2$ nós
- o nível 2 da árvore tem $2^2 = 4$ nós
- o nível 3 da árvore tem $2^3 = 8$ nós

e assim por diante ...

Em geral, o nível k da árvore tem 2^k nós.

Isso significa que, se a árvore tem altura k, então o algoritmo pode ter no máximo 2^k blocos de código (pois cada bloco de código está associado a um nó diferente da árvore).

Mas, nós já sabemos que serão necessários $n!$ blocos de código.

Então, k deve ser grande o suficiente de modo que

$$2^k \quad \geq \quad n!$$

o que é equivalente a

$$k \quad \geq \quad \log n!$$

A seguir, nós precisamos estimar o valor de $\log n!$.

$$\begin{aligned} \log n! \quad &= \quad \log\big(n \cdot (n-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1\big) \\ &= \quad \log n \ + \ \log(n-1) \ + \ \ldots \ + \ \log 3 \ + \ \log 2 \ + \ \log 1 \end{aligned}$$

A observação chave a seguir é que

$$\log n/2 \quad = \quad \log n \ - \ 1$$

Ou seja, os primeiros $n/2$ termos da soma acima são basicamente iguais a $\log n$ (com uma diferença de no máximo 1).

Isso nos permite obter a seguinte aproximação

$$\log n! \quad \geq \quad \frac{n}{2} \cdot \log n$$

Mas, isso significa que nós devemos ter

$$k \quad \geq \quad \log n! \quad \geq \quad \frac{n}{2} \cdot \log n \quad = \quad O(n \log n)$$

Isto é, qualquer algoritmo de ordenação baseado em comparações executa em tempo ao menos $O(n \log n)$.

3 Algoritmos de ordenação super-eficientes

Existem alguns casos particulares, no entanto, onde é possível ordenar a lista em tempo menor que $n \log n$.

Considere, por exemplo, a situação em que os elementos de uma lista de tamanho n são exatamente os números $1, 2, \ldots, n$ (em uma ordem qualquer).

Nesse caso, basta examinar o valor de um elemento para saber qual é a sua posição correta na lista — isto é, não é preciso fazer nenhuma comparação.

Abaixo, nós temos um algoritmo que ordena uma lista desse tipo:

```
Procedimento ord-Permutação ( V[1..n] )
 {
     k ← 1
     Enquanto ( k < n )
      {
        Se ( V[k] ≠ k )       // V[k] está fora de posição
         {
           p ← V[k]
           Troca (k,p)
         }
        Senão
           k++
      }
 }
```

Qual o tempo de execução desse algoritmo?

Bom, a cada iteração do laço pode ocorrer uma das seguintes coisas:

- o valor de `k` é incrementado
- um elemento da lista é trocado de posição

É fácil ver que o valor de `k` só pode ser incrementado n vezes.

Por outro lado, sempre que um elemento é trocado de posição ele vai para a sua posição correta, e nunca mais sai de lá.

Isso significa que podem ocorrer no máximo n trocas durante a execução do algoritmo.

Essas duas observações nos permitem concluir que o laço executa no máximo

$$n \ + \ n \quad = \quad 2n \text{ iterações}$$

Portanto, esse algoritmo de ordenação executa em tempo $O(n)$.

3.1 Algoritmo de ordenação *Radix*

Considere agora a situação em que nós sabemos que todos os números da lista são, digamos, menores do que 1000.

321	786	110	050	488	346	555	487	128	551	329	323	486

A ideia do algoritmo radix consiste em ordenar a lista examinando apenas um dígito dos números de cada vez.

Por exemplo, nós sabemos que `7xy` é menor do que `3wz` sem precisar saber quem são `x,y,w,z`.

Ordenando a lista acima de acordo com o primeiro dígito, nós obtemos

050	110	128	321	346	329	323	488	487	486	555	551	786

A seguir, a lista é reordenada de acordo com o segundo dígito.

Mas, é preciso cuidado para não desfazer o trabalho anterior!

Isto é, os números serão ordenados apenas nas faixas correspondentes ao mesmo primeiro dígito.

050	110	128	321	346	329	323	488	487	486	555	551	786

050	110	128	321	329	323	346	488	487	486	555	551	786

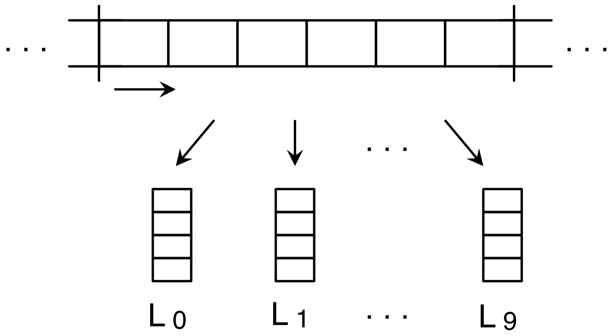
Finalmente, ordenando mais uma vez a lista de acordo com o terceiro dígito (tomando cuidado para não desfazer o trabalho anterior), nós obtemos

050	110	128	321	323	329	346	486	487	488	551	555	786

Qual o tempo de execução desse algoritmo?

A ordenação baseada em dígitos é um procedimento muito rápido.

Basta percorrer a lista uma vez da esquerda para a direita (ou a faixa que se quer ordenar), movendo os números para listas auxiliares $\mathtt{L}_0, \ldots, \mathtt{L}_9$, de acordo com o dígito correspondente



A seguir, basta percorrer as listas $\mathtt{L}_0, \ldots, \mathtt{L}_9$ (nessa ordem), movendo os números de volta para a lista (ou a faixa).

Utilizando esse procedimento, a ordenação da lista de acordo com um dígito (ou de todas as faixas separadamente, de acordo com esse dígito) leva tempo $O(n)$.

Logo, assumindo que o maior número da lista possui k dígitos, o algoritmo radix executa em tempo $O(kn)$.

Isto é, em qualquer situação em que $k < \log n$, o algoritmo radix pode ser uma boa opção.

Mas, ainda há um pequeno inconveniente no algoritmo que nós acabamos de apresentar:

- manter as faixas de valores já ordenadas pelos dígitos anteriores pode ser confuso e difícil de implementar na prática

Isso nos dá a oportunidade de apresentar uma ideia legal!

Suponha que, ao invés de ordenar a lista a partir do dígito mais significativo (como fizemos acima), nós começamos com o dígito menos significativo.

Aplicando essa ideia ao exemplo acima, nós obtemos o seguinte

321	786	110	050	488	346	555	487	128	551	329	323	486

110	050	321	551	323	555	786	346	486	487	488	128	329

110	321	323	128	329	346	050	551	555	786	486	487	488

050	110	128	321	323	329	346	486	487	488	551	555	786

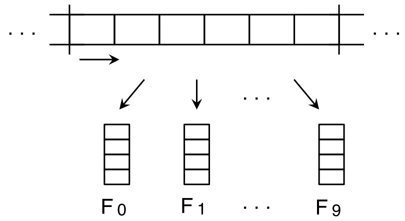
A coisa funciona!

É claro, aqui também é preciso cuidado para que cada ordenação não desfaça o trabalho realizado pelas anteriores.

Mas, nesse caso, o procedimento é bem mais simples

- sempre que houver empate entre dois elementos durante a ordenação por um certo dígito, a ordem entre os dois elementos produzida pelas ordenações anteriores deve ser preservada

Uma maneira simples de garantir isso consiste em armazenar os elementos em filas auxiliares $\mathtt{F}_0, \ldots, \mathtt{F}_9$ durante o processo de varredura.



Finalmente, ao invés de trabalhar com os dígitos $\mathtt{0}, \ldots, \mathtt{9}$ da base decimal, nós podemos trabalhar com os bits `0` e `1` da base binária.

Fazendo isso, nós só precisamos de duas filas auxiliares: $\mathtt{F}_0$ e $\mathtt{F}_1$.

Abaixo nós temos o pseudo-código do algoritmo de ordenação radix que utiliza essa ideia.

```
Procedimento ord-Radix ( V[1..n] )
 {
      F0,F1 ←  filas auxiliares vazias
      L ←  número de bits no maior elemento de V
      Para k ← 1 Até L
       {
          Para j ← 1 Até n
           {
              b ← getBit ( V[j], k )
              Se ( b = 0 )   insere V[j] em F0
              Senão          insere V[j] em F1
           }
          j ← 1
          Enquanto ( F0 ≠ vazio ) {   V[j] ← getFirst(F0);   j++   }
          Enquanto ( F1 ≠ vazio ) {   V[j] ← getFirst(F1);   j++   }
       }
 }
```

Exercícios

1. Considere uma lista de tamanho n onde os elementos são números inteiros entre 1 e $2n$.

   Você consegue construir um algoritmo que ordena essa lista em tempo $O(n)$?

2. Considere uma lista de registros onde todos os elementos possuem a chave -1 ou 1.
   O problema consiste em ordenar essa lista de acordo com o valor da chave obedecendo a seguinte restrição:

   - *a ordem relativa dos elementos com a mesma chave não pode ser alterada pelo processo de ordenação*

   a) Apresente um algoritmo que realiza essa tarefa em tempo $O(n)$.

   b) Você consegue realizar a tarefa em tempo menor que $O(n \log n)$ sem utilizar uma lista auxiliar?

   (CONTINUA ...)