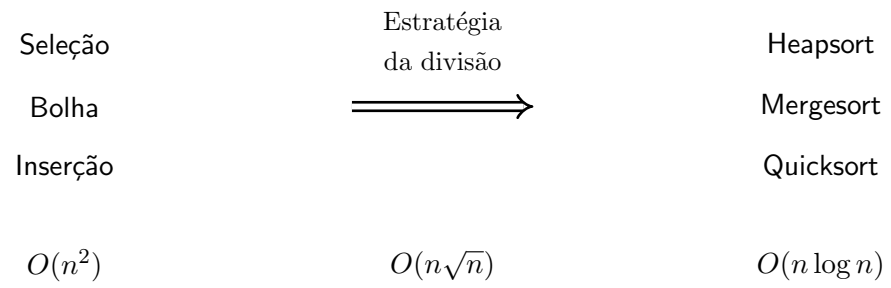
Construção e Análise de Algoritmos

aula 04: O problema da ordenação

1 Introdução

Depois de 3 aulas estudando algoritmos de ordenação, a situação que nós temos é a seguinte



E nesse ponto, a gente pode perguntar

- *Porque? porque? porque?*

Quer dizer, essa pergunta sempre faz sentido — (*porque?*)

E na aula de hoje, nós vamos produzir respostas para essas 3 perguntas.

2 Ordenação por comparação

Bom, a primeira pergunta pode ser reformulada assim

- *Porque a ordenação da bolha e a ordenação por inserção são uma porcaria ?*

Quer dizer, nós já vimos onde está a ineficiência da ordenação por seleção.

E já vimos como ela pode ser eliminada.

Agora, está na hora de entender melhor a ordenação da bolha e a ordenação por inserção.

E para fazer isso, a gente pode perguntar

- *O que há em comum entre esses dois algoritmos ?*

E a resposta é que os dois funcionam com base na seguinte operação

```
- comparar dois vizinhos na lista,
      e trocá-los de posição — (se necessário ...)
```

Certo.

Mas, porque isso não é uma boa ideia?

Bom, a resposta é que essa operação realiza muito pouco trabalho.

Daí que, para ordenar uma lista completamente ela precisa ser realizada muitas vezes.

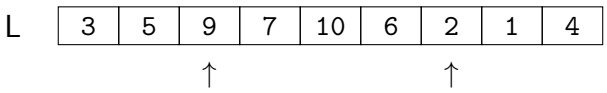
Daí que, qualquer algoritmo baseado nessa operação vai ser uma porcaria.

Legal.

Para entender isso ainda com mais precisão, nós podemos introduzir a noção de *inversão*.

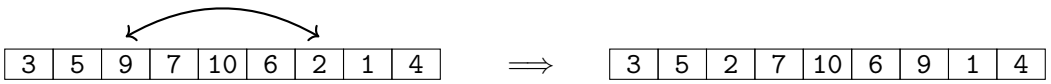
Quer dizer, uma inversão corresponde basicamente a um par de elementos que está fora de ordem.

Por exemplo,



É bem fácil ver que, para ordenar a lista completamente, todas as inversões precisam ser eliminadas.

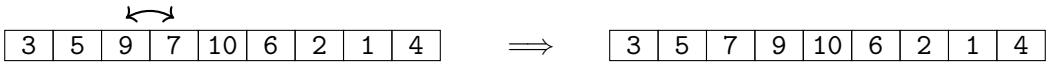
E a boa notícia é que a troca de um único par de elementos pode eliminar um monte de inversões



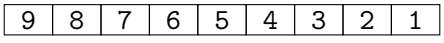
$\Rightarrow$ inversões eliminadas

- `(9,2)`
- `(9,7)`
- `(9,6)`
- `(7,2)`
- `(10,2)`
- `(6,2)`

Mas, a má notícia é que a troca de um par de vizinhos só pode eliminar uma única inversão.



E para piorar as coisas, a gente observa que uma lista de tamanho n pode ter $O(n^2)$ inversões



— (*i.e., todo par de elementos forma uma inversão ...*)

2.1 O algoritmo Shellsort

Legal.

Uma vez que a gente entendeu isso, a gente pode tentar remover a ineficiência do algoritmo.

(. . .)

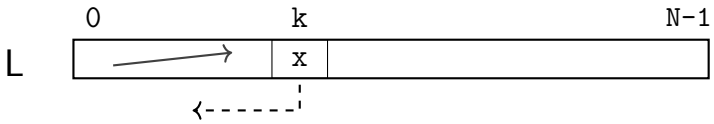
2.2 A ordenação da árvore

Certo.

A ideia que a gente acabou de ver remove a ineficiência do algoritmo da bolha.

Mas, a gente pode ter uma ideia diferente para a ordenação por inserção.

Quer dizer, relembre como ela funciona




```
- dado que a porção L[0..k-1] da lista já está ordenada
    a gente insere o elemento L[k] no lugar correto,
      para obter a porção L[0..k] ordenada
```

Daí, a primeira observação legal é a seguinte

→ *Como a porção* `L[0..k-1]` *já está ordenada,*
a gente pode fazer a busca binária para encontrar o lugar de `L[k]`

Isso reduz tremendamente o número de comparações realizadas pelo algoritmo.

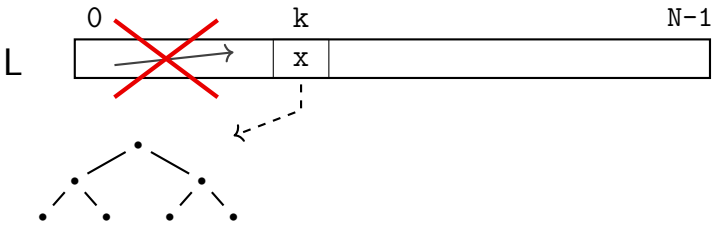
Mas, ainda é preciso deslocar os elementos para fazer a inserção de `L[k]`.

E por isso, o algoritmo continua sendo $O(n^2)$.

Sim.

Mas, agora que a gente entendeu, a gente pode fazer diferente.

Quer dizer, a esperteza consiste em usar uma árvore binária de busca para armazenar os elementos na porção ordenada



— (*ou, uma árvore binária balanceada ...*)

Porque daí, cada inserção leva tempo $O(\log n)$.

E a coisa toda leva tempo $O(n \log n)$.

Legal, não é?

3 A estratégia da divisão

O segundo *porque?* da Introdução pode ser reformulado assim

- *Porque a estratégia da divisão acelera o algoritmo da bolha ?*

Quer dizer, na aula 02 a gente viu que isso acontece.

Mas, a gente não perguntou: *porque?*

Só que, agora a gente perguntou.

E agora, a gente pode entender melhor as coisas.

Bom, a gente já sabe que a ordenação da bolha executa em tempo $O(n^2)$.

Daí, porque o tempo é quadrático, ordenar duas metades é mais rápido que ordenar a coisa inteira

$$\frac{n^2}{4} \quad + \quad \frac{n^2}{4} \quad = \quad \frac{n^2}{2} \quad < \quad n^2$$

Mas, a coisa também funcionaria se o algoritmo executasse em tempo $O(n\sqrt{n})$.

Porque

$$\frac{n}{2} \cdot \sqrt{n/2} \quad + \quad \frac{n}{2} \cdot \sqrt{n/2} \quad = \quad \frac{n \cdot \sqrt{n}}{\sqrt{2}} \quad < \quad n\sqrt{n}$$

Pensando desse modo, a gente vê que a estratégia da divisão sempre vale a pena.

A menos que o algoritmo já execute em tempo linear — (*i.e.,* $O(n)$ ...)

Porque daí

$$\frac{n}{2} \quad + \quad \frac{n}{2} \quad = \quad n$$

Mas, a gente está esquecendo um pequeno detalhe.

Quer dizer, a ordenação das duas metades ainda não produz uma lista completamente ordenada.

Para fazer isso, é preciso realizar a intercalação.

E daí, o tempo de ordenação por metades fica assim

$$\frac{n^2}{4} \quad + \quad \frac{n^2}{4} \quad + \quad n$$

Mas, o ganho quadrático ainda é maior do que o custo linear da intercalação.

Daí que, a coisa vale a pena.

E a coisa também valeria a pena se o algoritmo de base executasse em tempo $O(n\sqrt{n})$.

Veja só,

$$\begin{aligned}
&\tfrac{n}{2} \cdot \sqrt{n/2} \quad + \quad \tfrac{n}{2} \cdot \sqrt{n/2} \quad + \quad n \\
&= \quad \tfrac{n \cdot \sqrt{n}}{\sqrt{2}} \quad + \quad n \\
&= \quad n \cdot \left(\tfrac{\sqrt{n}}{\sqrt{2}} + 1 \right) \quad < \quad n\sqrt{n}
\end{aligned}$$

Mas, quando o algoritmo de base já executa em tempo $O(n \log n)$, a estratégia da divisão já não é mais efetiva.

Veja só,

$$
\begin{aligned}
&\tfrac{n}{2} \cdot \log\big(n/2\big) \quad + \quad \tfrac{n}{2} \cdot \log\big(n/2\big) \quad + \quad n \\
&= \quad n \cdot \log\big(n/2\big) \quad + \quad n \\
&= \quad n \cdot \Big(\log n \; - \; \log 2 \Big) \quad + \quad n \\
&= \quad n \log n \quad - \quad n \quad + \quad n \\
&= \quad n \log n
\end{aligned}
$$

Legal, não é?

4 Algoritmos ótimos de ordenação

Sim.

E isso nos leva à próxima pergunta

- *Será que existe um algoritmo de ordenação que executa em tempo menor que* $O(n \log n)$ *?*

E a resposta é: *Não!*

Só que, agora é preciso explicar porque.

Quer dizer, a gente vai argumentar que

→ *Não existe algoritmo de ordenação baseado em comparações que executa em tempo menor que* $O(n \log n)$

Porque?

Bom, porque a comparação de dois elementos gera muito pouca informação.

Daí que, é preciso fazer um monte de comparações para ordenar a lista completamente.

Daí que, qualquer algoritmo de ordenação baseado em comparações vai levar um boca de tempo.

Certo, isso faz sentido.

Mas, porque $O(n \log n)$ *?*

Bom, fixe o tamanho n da lista.

E imagine que a lista contém os elementos $1, 2, 3, \ldots, n$ — (*em uma ordem qualquer*)

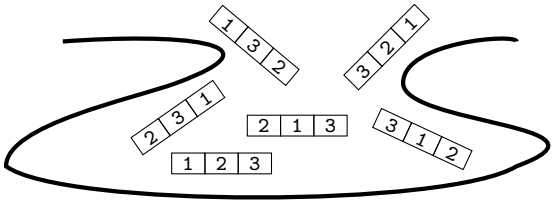
Quer dizer, os valores dos elementos realmente não importam.

Porque a única coisa que o algoritmo faz é comparar pares de elementos.

Certo.

Antes da coisa começar, o algoritmo ainda não sabe nada sobre a ordem em que os elementos estão — (*e por isso, ele não pode fazer nada ainda ...*)

A gente representa esse fato como um saco, onde se encontram todas as configurações possíveis da lista



Note que cada configuração da lista corresponde a uma permutação dos números de 1 a n.

Logo, existem $n!$ configurações no saco.

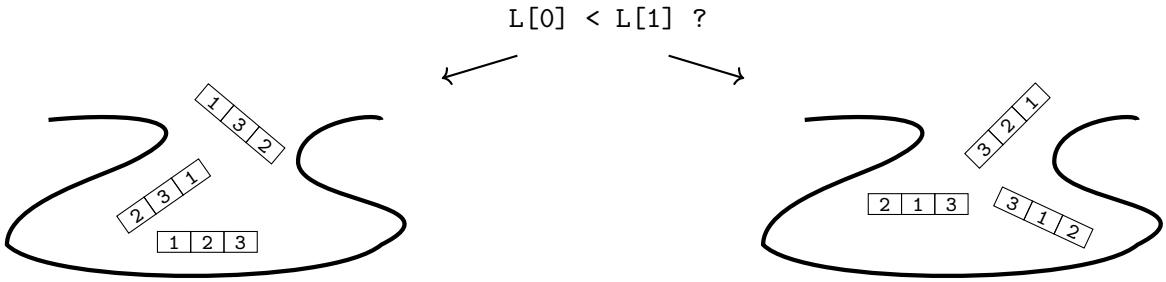
Legal.

Agora, imagine que o algoritmo compara os dois primeiros elementos da lista.

O que acontece?

Bom, isso depende do resultado da comparação.

E abaixo, nós temos os dois resultados possíveis



Quer dizer, a coisa importante é que o número de configurações dentro do saco foi reduzido à metade — (*porque?*)

E agora, a gente pode imaginar que uma nova comparação reduz o número de configurações à metade outra vez.

Bom, nem sempre é isso o que acontece — (*porque?*)

Mas, isso é o que acontece no melhor caso — (*porque?*)

E agora, a gente pode pensar sobre o que acontece quando todas as comparações correspondem ao melhor caso.

Bom, o que acontece é que o saco vai sendo reduzido sucessivamente à metade.

Até que reste apenas uma configuração lá dentro.

Nesse ponto, o algoritmo já sabe exatamente qual é a configuração em que os elementos se encontram.

E daí, ele pode aplicar as trocas necessárias para ordenar a lista completamente
— (*sem a necessidade de nenhuma comparação adicional ...*)

Certo.

Mas, quantas comparações são necessárias para chegar nesse ponto?

Bom, a resposta é

$$\log_2 n!$$

Sim, mas quanto é isso?

Bom, para descobrir isso, a gente reescreve a coisa assim

$$\log_2 \Big(1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n \Big)$$

e depois assim

$$\log_2 1 \;+\; \log_2 2 \;+\; \log_2 3 \;+\; \ldots \;+\; \log_2 n$$

E agora?

Bom, agora é a hora para uma pequena esperteza.

Quer dizer, primeiro a gente joga metade dos termos fora

$$\cancel{\log_2 1 \;+\; \log_2 2} \;+\; \log_2 3 \;+\; \ldots \;+\; \log_2 \tfrac{n}{2} \;+\; \ldots \;+\; \log_2 n$$

Depois, a gente observa que isso é maior que

$$\log_2 \frac{n}{2} \;+\; \log_2 \frac{n}{2} \;+\; \ldots \;+\; \log_2 \frac{n}{2} \qquad \text{(n/2 vezes)}$$

E depois, a gente nota que

$$\frac{n}{2} \cdot \log_2 \frac{n}{2} \quad = \quad O(n \log n)$$

Quer dizer, para ordenar a lista completamente, é preciso fazer ao menos $\frac{n}{2} \cdot \log n$ comparações.

E isso leva tempo $O(n \log n)$.

Legal, não é?

5 Ordenação em tempo linear

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

5.1 Ordenação de permutações

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 $O(n)$.

5.2 Algoritmo de ordenação do balde

A ideia que nós acabamos de ver também pode ser adaptada para outros casos.

Por exemplo, suponha que o vetor `V[1..n]` contém apenas números inteiros positivos (todos distintos), e nós sabemos que o maior número em `V` é igual a $m > n$.

Então, nós podemos ordenar o vetor `V` com o auxílio de uma lista auxiliar `L`, através do seguinte procedimento

1. inicializar todas as posições de `L` com o valor 0
2. percorrer o vetor `V`, colocadando cada elemento k na k-ésima posição de `L`
3. percorrer a lista `L`, movendo os seus elementos maiores do que 0 de volta para o vetor `V`

Não é difícil ver que esse algoritmo executa em tempo

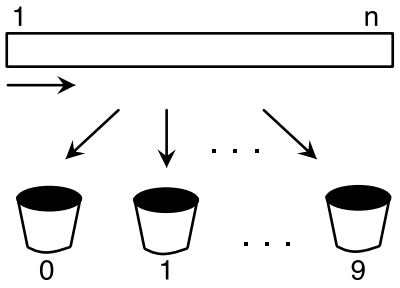
$$O(m) \quad + \quad O(n) \quad + \quad O(m) \qquad = \qquad O(n+m)$$

Logo, ele pode ser uma boa opção, quando $m < n \log n$.

Agora, considere o caso em que o vetor `V[1..n]` possui muitos elementos repetidos.

Por exemplo, suponha que os únicos números que aparecem em `V` são: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Nesse caso, nós podemos adaptar o algoritmo acima como indicado na figura abaixo



Quer dizer, ao invés da lista auxiliar `L`, nós vamos utilizar uma estrutura de dados que funciona como uma coleção de baldes.

A ideia então é percorrer o vetor `V` colocando cada elemento no balde correspondente.

Uma vez que isso tenha sido feito, basta mover os elementos do primeiro balde de volta para o vetor `V`, em seguida os elementos do segundo balde, e assim por diante.

Qual é o tempo de execução desse algoritmo?

Bom, fazendo a suposição de que a inserção de um elemento na estrutura de dados dos baldes leva tempo $O(1)$, não é difícil ver que a primeira etapa executa em tempo $O(n)$.

E, fazendo a suposição de que a remoção de um elemento de um balde leva tempo $O(1)$, nós vemos que a segunda etapa também leva tempo $O(n)$.

Portanto, o algoritmo como um todo executa em tempo

$$O(n) \quad + \quad O(n) \qquad = \qquad O(n)$$

Ou seja, melhor impossível!

5.3 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

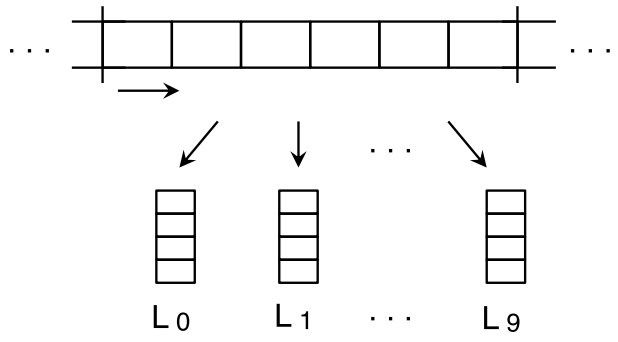
Finalmente, ordenando mais uma vez a lista de acordo com o terceiro dígito (tomando cuidado para não desfazer o trabalho anterior), nós obtemos

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.

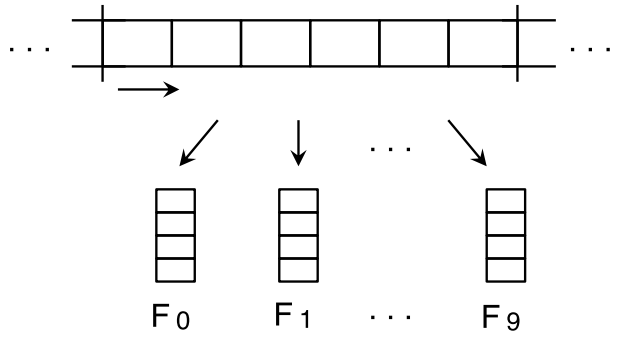
Mas, nesse caso, o procedimento é bem mais simples

- 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 $\texttt{F}_0, \ldots, \texttt{F}_9$ durante o processo de varredura.
Finalmente, ao invés de trabalhar com os dígitos $\texttt{0}, \ldots, \texttt{9}$ da base decimal, nós podemos trabalhar com os bits `0` e `1` da base binária.

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

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++   }
      }
 }
```