**Fundamentos de programação**

**aula 06: Ordenação de listas**

## 1 Introdução

Na aula passada, nós vimos que a busca por um elemento em uma lista ordenada pode ser implementada de maneira muito, muito eficiente.

Mas, para tirar proveito desse fato, antes é preciso ordenar a lista, não é?

Na aula de hoje, nós vamos mostrar algumas maneiras diferentes de fazer isso.

## 2 Ordenando uma lista

Se você quer colocar os elementos de uma lista em ordem crescente, uma boa maneira de começar consiste em

- encontrar o maior elemento de todos, e movê-lo para a última posição

O seguinte trecho de programa faz exatamente isso

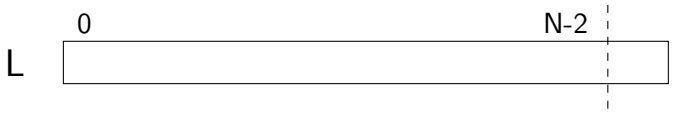
```
pM = 0;
for (i=1; i<N; i++)
   if ( L[i] > L[pM] )   pM = i;

aux = L[pM];   L[pM] = L[N-1];   L[N-1] = aux;
```

Quer dizer, primeiro nós localizamos a posição do maior elemento da lista.

E depois nós trocamos esse elemento de lugar com o último elemento.

Daí, veja só.

Se nós deixarmos o último elemento da lista de lado por um momento




então nós estamos novamente diante do mesmo problema: colocar a lista que vai de 0 até N-2 em ordem crescente.

E uma boa maneira de começar a fazer isso consiste em

- encontrar o maior elemento de todos, e movê-lo para a última posição

O seguinte trecho de programa faz exatamente isso

```
pM = 0;
for (i=1; i<N-1; i++)
   if ( L[i] > L[pM] )   pM = i;

aux = L[pM];   L[pM] = L[N-2];   L[N-2] = aux;
```

*Você já viu o que está acontecendo, não é?*

Quer dizer, executando esse trecho de programa de novo, de novo e de novo, nós colocamos a lista inteira em ordem.

O único cuidado é fazer com que ele vá um pouquinho menos longe na lista de cada vez — (porque o final já está em ordem).

O programa abaixo implementa essa ideia

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70};

int i, k, pM, aux;

int main()
 {
    for (k=N-1; k>=1; k--)
     {
        pM = 0;
        for (i=1; i<=k; i++)
           if ( L[i] > L[pM] )   pM = i;

        aux = L[pM];   L[pM] = L[k];   L[k] = aux;
 }   }
```

Note que o programa pára no momento em que o segundo menor elemento da lista é colocado na posição `L[1]`.

Isso faz sentido, porque nesso ponto o único elemento que pode estar na posição `L[0]` é o menor de todos.

### 2.1 **Análise de desempenho** (Opcional)

Certo.

Nós já temos um programa que ordena a lista corretamente.

Agora, nós podemos perguntar

- *Quanto tempo esse programa leva para fazer o seu trabalho?*

Bom, isso depende do computador em que ele executa.

Sim, é verdade.

Mas, se nós mudarmos a pergunta para

- *Quantas comparações esse programa realiza para ordenar a lista inteira?*

daí a resposta já não depende mais do computador.

Legal, então nós vamos calcular isso.

Nós começamos observando que o trecho de programa

```
pM = 0;
for (i=1; i<N; i++)
   if ( L[i] > L[pM] )   pM = i;

aux = L[pM];   L[pM] = L[N-1];   L[N-1] = aux;
```

realiza `N-1` comparações porque

- cada volta do comando `for` faz 1 comparação
- o comando `for` dá `N-1` voltas

Agora, é fácil ver que na hora em que esse trecho é executado outra vez, indo apenas até a posição `N-2`, serão realizadas `N-2` comparações.

E na próxima vez serão realizadas `N-3` comparações.

Daí que, o número total de comparações realizadas pelo programa é:

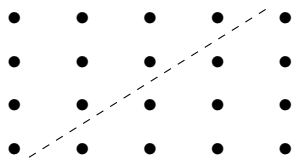
$$1 \ + \ 2 \ + \ 3 \ + \ \ldots \ + \ (\mathtt{N}-2) \ + \ (\mathtt{N}-1)$$

*Mas, o quanto é isso?*

Bom, existem muitas maneiras de calcular essa soma.

Nós vamos fazer a observação de que uma soma desse tipo é como um triângulo


E que dois triângulos são iguais a um retângulo



Daí observando que, no nosso caso, a base do retângulo é `(N-1)+1` e a sua altura é `N-1`, nós temos que o retângulo é igual a

$$\mathtt{N} \cdot (\mathtt{N}-1)$$

Mas, como a soma é um triângulo, o que é apenas meio retângulo, nós temos que

$$1 \ + \ 2 \ + \ 3 \ + \ \ldots \ + \ (\mathtt{N}-2) \ + \ (\mathtt{N}-1) \quad = \quad \frac{\mathtt{N} \cdot (\mathtt{N}-1)}{2}$$

Certo.

Agora nós já sabemos quantas comparações o nosso programa realiza.

A seguir, olhando outra vez para o trecho de programa

```
pM = 0;
for (i=1; i<N; i++)
   if ( L[i] > L[pM] )   pM = i;

aux = L[pM];   L[pM] = L[N-1];   L[N-1] = aux;
```

nós observamos que, além da comparação, cada volta do `for` também realiza

- o incremento `i++`
- o teste `i <= N-1`
- a instrução `pM = i` (se necessário)

Além disso, nós também temos 1 instrução antes e 3 instruções depois do comando `for`.

Somando tudo, nós vemos que esse trecho de programa pode executar até

$$4(N-1) \ + \ 4 \qquad \text{instruções}$$

(aproximadamente)

Agora, imagine que você quer ordenar uma lista com 10.000 elementos, em um computador que executa cada instrução em $1\mu$s.

Então, a coisa toda vai levar (aproximadamente)

$$4 \times \frac{10000 \times 9999}{2} \ + \ 4 \times 10000 \ \mu s \quad \simeq \quad 3,3 \ \text{min}$$

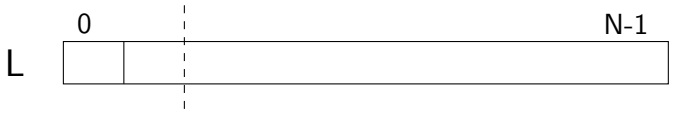
## 3 Ordenação por inserção

O programa que nós acabamos de ver ordena a lista do fim para o começo.

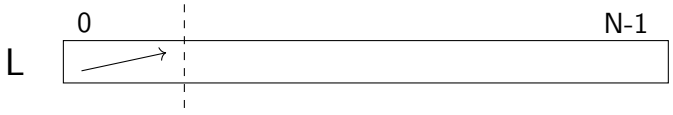
Agora nós vamos ver um programa que ordena a lista do começo para o fim.

A sua ideia é a seguinte.

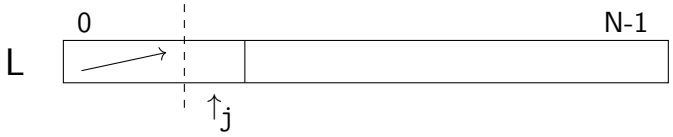
Esqueça tudo por um momento, menos os dois primeiros elementos




Daí, comparando os dois e colocando o menor na primeira posição, nós já temos esse comecinho da lista ordenado



Agora é a vez do terceiro elemento



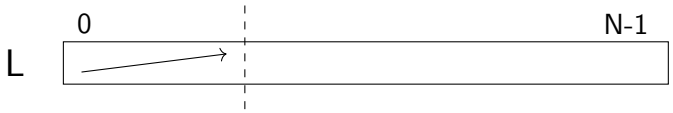

E a ideia é colocá-lo na posição correta dentro desse comecinho.

O seguinte trecho de programa faz exatamente isso

```
j = 2;
while ( j>0  &&  L[j] < L[j-1] )
 {
    aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
 }
```

Quer dizer, enquanto esse elemento é menor do que o anterior, ele vai sendo movido para a esquerda.

Daí que, após a execução desse trecho de programa, a situação que nós temos é a seguinte




E agora é a vez de fazer a mesma coisa com o elemento da quarta posição.

*Já deu para ver como a coisa funciona, não é?*

O programa abaixo implementa esse esquema de ordenação

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70};

int j, k, aux;

int main()
 {
    for (k=1; k<N; k++)
     {
        j = k;
        while ( j>0  &&  L[j] < L[j-1] )
         {
            aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
 }   }   }
```

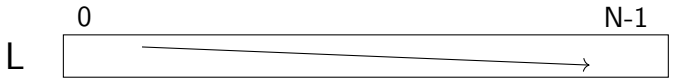
### 3.1 **Análise de desempenho** (Opcional)

Esse programa tem uma pequena vantagem com relação ao anterior

- Quando a lista já está ordenada

- o programa anterior ainda assim realiza as suas $\frac{N(N-1)}{2}$ comparações
- a ordenação por inserção só faz uma comparação para cada elemento da lista

Por outro lado, se a lista está organizada em ordem decrescente




então cada elemento é comparado com todos os anteriores, o que nós dá um total de

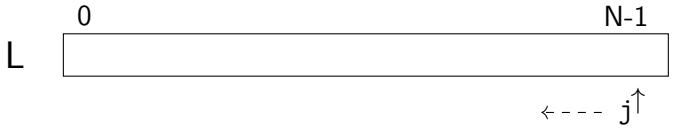
$$1 \ + \ 2 \ + \ 3 \ + \ \ldots \ + \ (N-2) \ + \ (N-1) \qquad \text{comparações}$$

## 4 O algoritmo da bolha

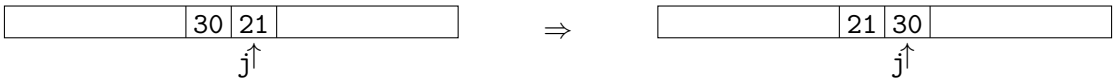
O algoritmo da bolha tem esse nome porque ele simula o comportamente das bolhas dentro d'água.

Quer dizer, aquilo que é mais leve tende a subir.

Daí que, a ideia do algoritmo é percorrer a lista do fim para o começo




trocando elementos consecutivos de posição que estejam na posição invertida (i.e., o maior à esquerda do menor)




trocando elementos consecutivos de posição que estejam na posição invertida (i.e., o maior à esquerda do menor)

O trecho de programa abaixo implementa esse procedimento de varredura

```
for (j=N-1; j>0; j--)
    if ( L[j] < L[j-1] )
     {
       aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
     }
```

Fazendo isso, os elementos menores vão sendo movidos para frente — (e por consequência, os maiores vão sendo levados para trás).

Mas, a observação importante é que, após esse procedimento de varredura, o menor elemento de todos vai estar na primeira posição.

E daí, a ideia é repetir essas varreduras até que a lista esteja completamente ordenada.

O programa abaixo faz exatamente isso

```
#include <stdlib.h>
#include <stdio.h>

#define  N  10

int L[N] = { 19, 27, 12, 31, 5, 28, 53, 39, 42, 70};

int j, k, aux;

int main()
 {
    for (k=0; k<N-1; k--)
     {
        for (j=N-1; j>k; j--)
         {
            if ( L[j] < L[j-1] )
             {
               aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
 }   }   }   }
```

### 4.1 **Análise de desempenho** (Opcional)

( . . . )

## 5 Uma pequena esperteza

Nesse ponto, talvez você esteja pensando que não é possível ordenar uma lista com `N` elementos fazendo menos do que $\frac{N(N-1)}{2}$ comparações (no pior caso).

( . . . )