

Construção e Análise de Algoritmos

aula 01: Algoritmos básicos de ordenação

1 Introdução

Imagine que L é uma lista que armazena números inteiros.

E imagine que a tarefa é

→ *Verificar se existe algum elemento repetido na lista*

Bom, se a lista está ordenada isso é moleza: basta examinar os pares de elementos consecutivos.

E isso é bem rápido também: $O(n)$ — (*porque?*)

Mas, para que a lista esteja ordenada, primeiro é preciso ordená-la, não é?

Esse é o problema que nós vamos estudar hoje.

2 Ordenação por seleção

A ideia mais natural para ordenar uma lista, provavelmente é a seguinte

- encontrar o maior elemento de todos, e colocá-lo na última posição
- encontrar o segundo maior elemento, e colocá-lo na penúltima posição
- encontrar o terceiro maior elemento, e colocá-lo na antepenúltima posição
- e por aí vai ...

A boa notícia é que é bem fácil encontrar o maior elemento da lista

```
M <-- L[1];    posM <-- 1

Para i <-- 2 Até N
{
    Se ( L[i] > M )
    {
        M <-- L[i];    posM <-- i
    }
}
```

E também é bem fácil colocar esse trecho de código dentro de uma função que encontra o maior elemento dentro de uma faixa da lista

```
func posMaior ( L, inicio, fim )
{
    M <-- L[inicio];    posM <-- inicio

    Para i <-- inicio + 1 Até fim
    {
        Se ( L[i] > M )
        {
            M <-- L[i];    posM <-- i
        }
    }
    Retorna ( posM )
}
```

Certo.

Agora imagine que nós também temos a função

```
func Troca ( L, i, j )
```

— (*que troca os elementos das posições i, j de lugar*)

Então, é bem fácil ver que o seguinte par de instruções coloca o maior elemento na última posição

```
k <-- posMaior ( L, 1, N )
Troca ( L, k, N )
```

E que depois disso, o seguinte par de instruções coloca o segundo maior elemento na penúltima posição

```
k <-- posMaior ( L, 1, N-1 )
Troca ( L, k, N-1 )
```

Daí que, para ordenar a lista completamente, basta botar esse negócio para repetir

```
Para j <-- N Até 2
{
  k <-- posMaior ( L, 1, j )
  Troca ( L, k, j )
}
```

Esse algoritmo é conhecido como a *ordenação por seleção*.

E agora, nós vamos analisar o seu tempo de execução.

Análise

Para fazer isso, o primeiro passo é analisar a função `posMaior()`.

E para isso, nós anotamos o seu código da seguinte maneira

```
func posMaior ( L, inicio, fim )
{
  0(1) | M <-- L[inicio];    posM <-- inicio

  Para i <-- inicio + 1 Até fim
  {
    0(1) | Se ( L[i] > M )
          {
            M <-- L[i];    posM <-- i
          }
  }
  0(1) | Retorna ( posM )
}
```



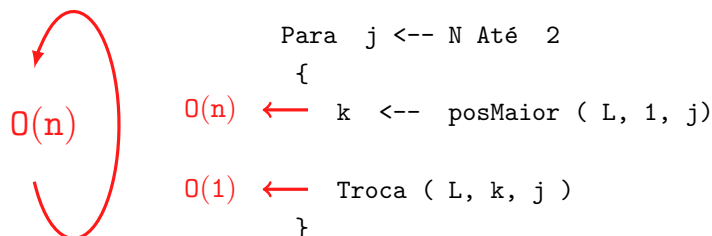
A observação fácil aqui é que, no pior caso, o laço dá $O(n)$ voltas — (*porque?*)

Daí que, no pior caso, a função executa em tempo

$$O(1) + O(n) \cdot O(1) + O(1) = O(n)$$

Certo.

Daí, sabendo disso, a gente anota o código principal assim



— (porque a função `Troca()` foi anotada com $O(1)$?)

Daí que, o tempo de execução da ordenação por seleção é

$$O(n) \cdot \left(O(n) + O(1) \right) = O(n^2)$$

Legal.

Mas daí, alguém poderia dizer que a nossa análise não foi muito precisa.

Quer dizer, a gente assumiu que a função `posMaior()` executa em tempo $O(n)$.

Mas, isso é apenas o tempo de execução no pior caso.

No melhor caso, a função executa em tempo $O(1)$ — (e é isso o que acontece nas últimas iterações do algoritmo)

Então, para levar isso em conta, nós vamos raciocinar com mais precisão.

Quer dizer, nós vamos considerar o tempo de execução de cada volta do laço em separado.

- na primeira volta, a chamada à função `posMaior()` leva em tempo n
- na segunda volta, a chamada à função `posMaior()` leva tempo $n - 1$
- na terceira volta, a chamada à função `posMaior()` leva tempo $n - 2$
- e por aí vai ...

Daí que, somando todos esses tempos a gente chega na expressão

$$n + (n - 1) + \dots + 3 + 2 + 1$$

Mas, todo mundo sabe que isso vale

$$\frac{n \cdot (n + 1)}{2}$$

E nós temos que

$$\frac{n \cdot (n + 1)}{2} = O(n^2)$$

— (porque?)

Quer dizer, no final das contas, a análise mais precisa não fez a menor diferença — (*ou só fez uma diferença menor ...*)

É por isso que, na maioria das vezes, a gente não faz uma análise precisa do tempo de execução de um algoritmo.

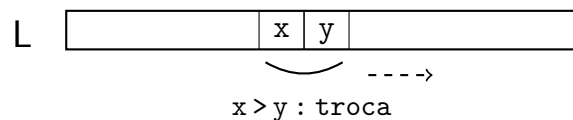
3 O algoritmo da bolha

Mas, essa não é a única maneira de fazer as coisas.

Quer dizer, a gente também pode ordenar a lista fazendo varreduras.

O procedimento de *varredura* consiste em

→ *Percorrer a lista da esquerda para a direita comparando elementos consecutivos, e fazendo a troca quando eles estão fora de ordem*



O efeito principal desse procedimento é levar o maior elemento de todos para a última posição — (*mas, quais são os seus efeitos colaterais?*)

Daí que, se a gente aplica varreduras sucessivas, a lista eventualmente fica completamente ordenada.

Para implementar essa ideia, o primeiro passo é escrever a função `Varredura()`

```
func Varredura ( L, inicio, fim )
{
  Para i <-- inicio Até fim - 1
  {
    Se ( L[i] > L[i+1] )
      Troca ( L, i, i+1 )
  }
}
```

Daí, a ordenação é feita botando o negócio para repetir

```
Para j <-- N Até 2
{
  Varredura ( L, 1, j )
}
```

Esse algoritmo é conhecido a ordenação da bolha — (*porque?*)

E agora, nós vamos analisar o seu tempo de execução.

Análise

Para fazer isso, o primeiro passo é analisar a função `Varredura()`.

E para isso, nós anotamos o seu código da seguinte maneira

```

func Varredura ( L, inicio, fim )
{
    Para i <-- inicio Até fim - 1
    {
        Se ( L[i] > L[i+1] )
            Troca ( L, i, i+1 )
    }
}

```



A observação fácil aqui é que o laço dá $O(n)$ voltas no pior caso — (*porque?*)

Daí que, no pior caso, a função executa em tempo

$$O(n) \cdot O(1) = O(n)$$

Certo.

Daí, sabendo disso, a gente anota o código principal assim

```

Para j <-- 1 Até N - 1
{
    Varredura ( L, 0, N-j )
}

```



Daí que, o tempo de execução da ordenação da bolha é

$$O(n) \cdot O(n) = O(n^2)$$

Exercício: Imagine que uma chamada à função `Varredura()` não realiza nenhuma troca.

Nesse caso, a ordenação da bolha já pode parar — (*porque?*)

Isso faz alguma diferença para o tempo de execução do algoritmo?

Bom, a resposta é: Sim e Não.

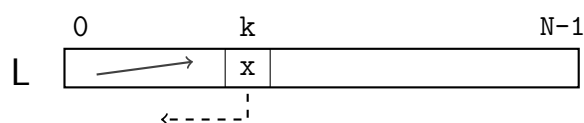
Porque?

4 Ordenação por inserção

O nosso próximo algoritmo vai ordenar a lista do começo para o fim.

Quer dizer, imagine que a porção $L[0..k-1]$ da lista já está ordenada.

E imagine que nós queremos inserir o elemento da posição k no lugar correto nessa ordem

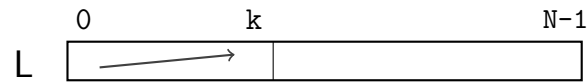


Como é que a gente faz isso?

Bom, uma ideia simples consiste em

→ *Trocar esse elemento de lugar com o elemento anterior
enquanto ele for menor do que o anterior*

Depois que a gente faz isso, a porção $L[0..k]$ da lista fica ordenada



E se a gente continuar fazendo isso, a lista eventualmente fica completamente ordenada.

Legal.

Para implementar essa ideia, o primeiro passo é escrever a função `Inserção()`

```
func Inserção ( L, k )
{
    i <-- k

    Enquanto ( i > 0 && L[i] < L[i-1] )
    {
        Troca ( L, i, i-1 )
        i--
    }
}
```

E daí, basta botar esse negócio para repetir

```
Para k <-- 2 Até N
{
    Inserção ( L, k )
}
```

Esse algoritmo é conhecido como a *ordenação por inserção*.

E agora, nós vamos analisar o seu tempo de execução.

Análise

Para fazer isso, o primeiro passo é analisar a função `Inserção()`.

E para isso, nós anotamos o seu código da seguinte maneira

```
func Inserção ( L, k )
{
    0(1) | i <-- k

    Enquanto ( i > 0 && L[i] < L[i-1] )
    {
        0(1) | Troca ( L, i, i-1 )
        i--
    }
}
```

A red circular arrow with a question mark inside is drawn to the left of the while loop, indicating a point of analysis or a question about the loop's complexity.

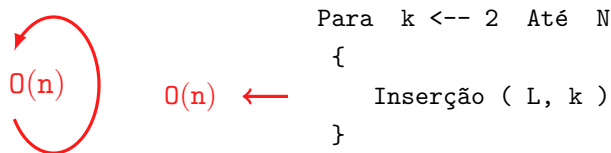
A observação fácil aqui é que, no pior caso, o laço dá $O(n)$ voltas — (*porque?*)

Daí que, no pior caso, a função executa em tempo

$$O(1) + O(n) \cdot O(1) = O(n)$$

Certo.

Daí, sabendo disso, a gente anota o código principal assim



Daí que, o tempo de execução da ordenação por inserção é

$$O(n) \cdot O(n) = O(n^2)$$

5 Quem é o mais rápido?

Nós acabamos de ver algoritmos de ordenação baseados em 3 ideias diferentes.

E nós também vimos que os 3 algoritmos executam em tempo $O(n^2)$.

Mas isso não significa que os 3 algoritmos executam exatamente no mesmo tempo.

Quer dizer, lembre que a nossa análise é aproximada.

Daí, faz sentido perguntar

- *Quem é o mais rápido ?*

Bom, uma maneira de decidir a questão consiste em escrever programas de computador que implementam os 3 algoritmos, e fazer uma bateria de testes.

Fazendo isso, a gente obtém o seguinte resultado

```
ordenação por seleção:  ????
```

```
ordenação da bolha:    ????
```

```
ordenação por inserção:  ????
```

Quer dizer, a ordenação por inserção é a mais rápida de todas.

E a ordenação da bolha é a mais lenta de todas.

Mas daí alguém pode perguntar: *porque?*

Faz sentido começar comparando a ordenação por seleção e a ordenação da bolha, porque esses algoritmos são baseados nas funções

- `Maior()`

- `Varredura()`

que percorrem a lista inteira da esquerda para a direita.

Daí a gente observa que

- enquanto a função `Maior()` faz comparações com todos os elementos, em busca do maior deles
- a função `Varredura()` faz comparações com todos os elementos, e também troca um monte deles de lugar

Isso já explica porque a ordenação da bolha é mais lenta do que a ordenação por seleção.

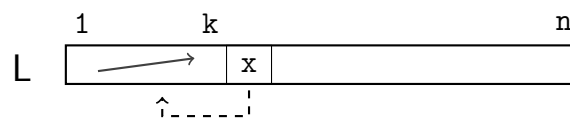
Certo.

Mas porque a ordenação por inserção é a mais rápida de todas?

A resposta é simples

→ *Porque a função `Inserção()` nem sempre percorre toda a porção ordenada da lista*

Quer dizer, se a posição correta do elemento da posição $k + 1$ na porção ordenada $L[1..k]$ é algum ponto no meio do caminho



então a função `Inserção()` só percorre essa parte do caminho.

E isso faz com que a ordenação por inserção seja a mais rápida de todas.

Legal, não é?