Estruturas de Dados

aula 02: Análise de algoritmos

1 Introdução

Agora que os nossos algoritmos já foram escritos, está na hora de perguntar

- *Será que o meu algoritmo é bom ?*

Mas, antes disso, o que é um algoritmo bom?

Bom, um algoritmo é bom se ele está correto e se ele é rápido.

É bem difícil ter a certeza absoluta de que o nosso algoritmo funciona corretamente
— (*em todos os casos possíveis ...*)

Então, nós vamos deixar essa questão de lado nesse momento.

E vamos passar para o outro lado da questão

- *O que é um algoritmo rápido ?*

Bom, um algoritmo é rápido se ele não fica dando voltas por todo lado.

Ou melhor, se ele não dá mais voltas do que o necessário para resolver o problema.

Quer dizer, no final das contas, um algoritmo é só uma coisa que fica dando voltas
— (*i.e., uma estrutura de repetição com alguma lógica dentro ...*)

Então, para saber se o algoritmo é rápido, basta contar quantas voltas ele dá.

Vejamos como a coisa funciona.

2 Análise de algoritmos

Considere novamente a tarefa

→ *Encontrar o maior elemento de uma lista*

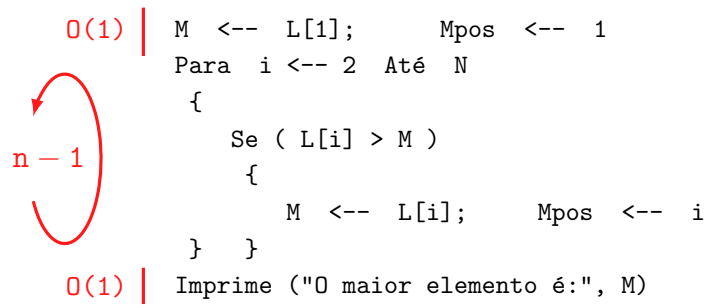
Na primeira aula nós apresentamos o seguinte algoritmo para esse problema

```
M  <--  L[1];      Mpos  <--  1
Para  i <-- 2  Até  N
 {
    Se ( L[i] > M )
     {
        M  <--  L[i];     Mpos  <--  i
 }   }
Imprime ("O maior elemento é:", M)
```

Daí, examinando o código nós fazemos duas observações simples

- o laço sempre dá $n - 1$ voltas
- antes e depois do laço existem instruções que só são executadas uma vez

Nós podemos representar essa informação no código da seguinte maneira

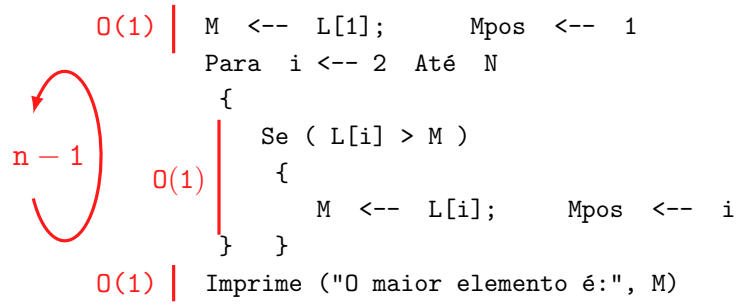


Certo.

Agora está na hora de examinar o que acontece dentro do laço.

E o que nós vemos ali é um trecho de código que é executado apenas uma vez — (*a cada volta do laço*)

Essa informação pode ser representada da seguinte maneira



Pronto.

Com base nessas anotações, nós podemos escrever a seguinte expressão para o tempo de execução do algoritmo

$$O(1) \quad + \quad (n-1) \times O(1) \quad + \quad O(1)$$

Finalmente, descartando os termos menores e concentrando a atenção no termo de maior ordem, nós chegamos ao seguinte

$$\cancel{O(1)} \quad + \quad \underbrace{(n-1) \times O(1)}_{O(n)} \quad + \quad \cancel{O(1)} \quad = \quad O(n)$$

Esse resultado está dizendo que o tempo de execução do algoritmo é da ordem de n
— (*ou mais ou menos igual a* n)

E isso faz sentido porque

→ *percorrer a lista examinando cada um dos seus elementos*
leva tempo proporcional ao tamanho da lista — (mais ou menos ...)

Legal, não é?

A seguir, nós vamos ver mais exemplos de análise de algoritmos.

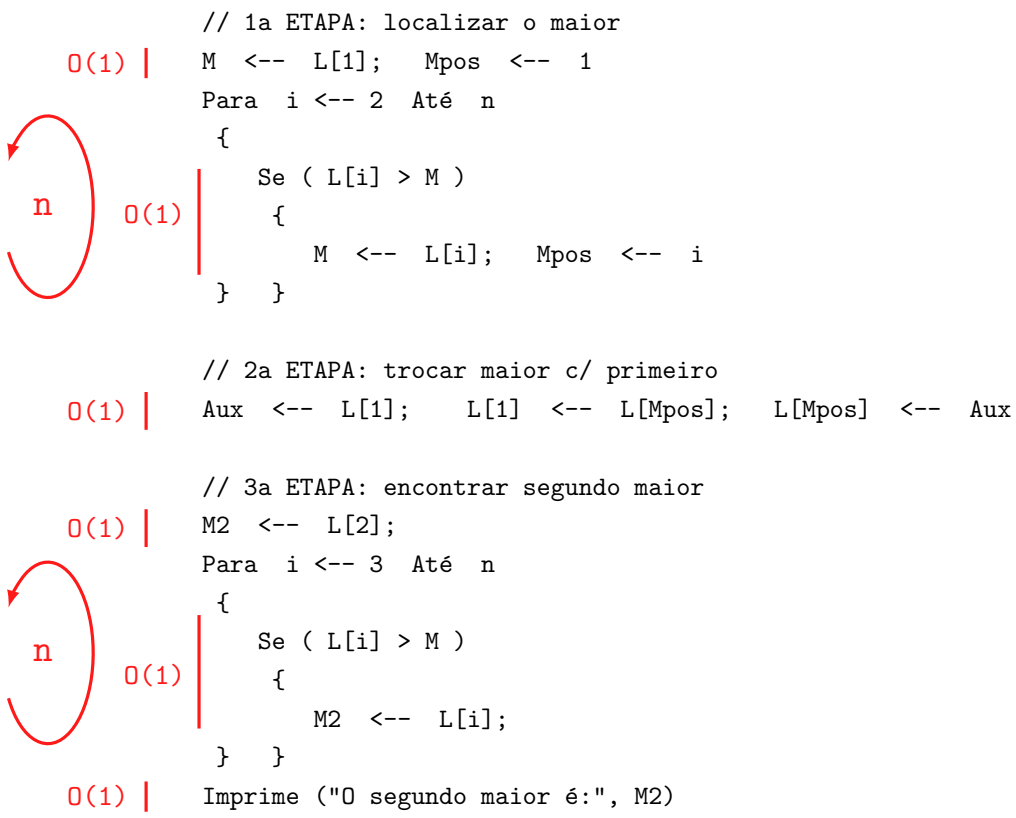
Exemplos

a. O segundo maior

Considere a seguinte tarefa

→ *Encontrar o segundo maior elemento da lista* L

Abaixo nós temos o primeiro algoritmo apresentado na aula 00.1 para esse problema, já anotado



Observe que, dessa vez, coisas como $n-1$ e $n-2$ foram anotadas simplesmente como n — (*porque a análise é só mais ou menos ...*)

Agora com base nas anotações, nós obtemos a seguinte expressão para o tempo de execução do algoritmo

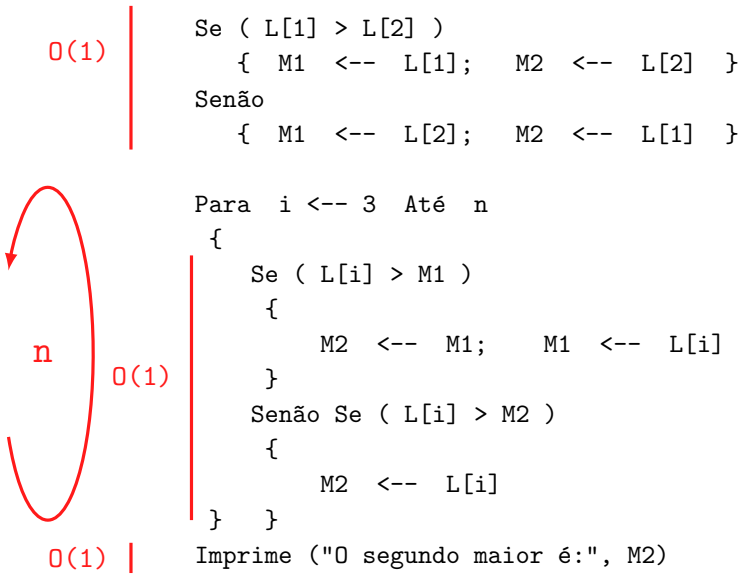
$$O(1) \quad + \quad n \times O(1) \quad + \quad O(1) \quad + \quad O(1) \quad + \quad n \times O(1) \quad + \quad O(1)$$

Daí, descartando os termos menores nós chegamos ao seguinte resultado

$$\cancel{O(1)} \quad + \quad n \times O(1) \quad + \quad \cancel{O(1)} \quad + \quad \cancel{O(1)} \quad + \quad n \times O(1) \quad + \quad \cancel{O(1)} \quad = \quad 2 \cdot O(n)$$

Legal.

Agora nós vamos analisar a segunda versão do algoritmo



Daí com base nas anotações, nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

$$O(1) \quad + \quad n \times O(1) \quad + \quad O(1) \qquad = \qquad O(n)$$

Quer dizer, as análises nos deram os seguintes resultados

- `versão 1:` $2 \cdot O(n)$
- `versão 2:` $O(n)$

E isso parece indicar que a segunda versão é mais eficiente do que a primeira.

Mas aqui é preciso cuidado.

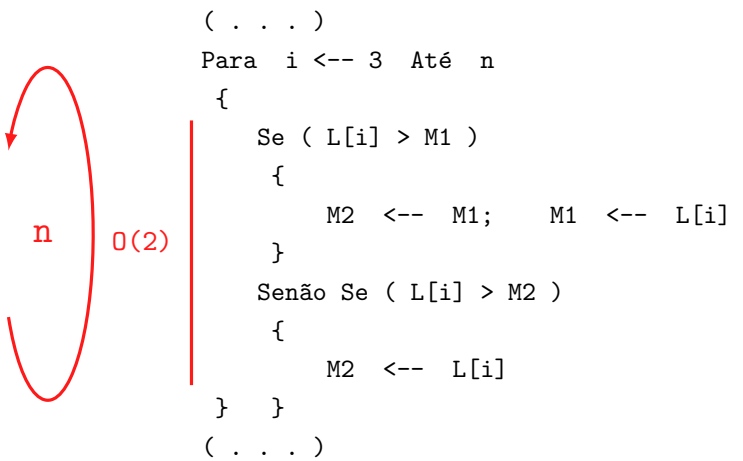
Porque, examinando outra vez os códigos, nós vemos que

→ a `versão 1` contém dois laços, com um teste dentro de cada um

→ a `versão 2` contém um laço, com dois testes dentro dele

E isso é basicamente a mesma coisa.

Quer dizer, para levar esse detalhe em conta, nós deveríamos anotar a `versão 2` assim



Mas, para não complicar as coisas, nós vamos assumir que coisas como $2O(n)$ e $3O(n)$ são a mesma coisa que $O(n)$.

Dessa maneira, o resultado final da análise é

- `versão 1:` $O(n)$
- `versão 2:` $O(n)$

De modo que os dois algoritmos são igualmente eficientes — (*do ponto de vista da nossa análise mais ou menos*)

Isso faz sentido, porque os dois algoritmos de fato executam em tempo proporcional ao tamanho da lista.

◇

b. Busca

Considere a seguinte tarefa

→ *Verificar se o número x está na lista* L *ou não*

Na aula 00.1, nós resolvemos esse problema para o caso ordenado e o caso não-ordenado.

E aqui as coisas se complicam um pouquinho outra vez.

Quer dizer, os dois algoritmos utilizam o comando `Break`.

E isso significa que nós não sabemos quantas voltas os laços vão dar


```
// Caso não ordenado

Aux  <--  0
Para  i <-  1  Até  n
 {
    Se ( L[i] = x )
     {
        Aux  <--  i
        Break
 }   }
Se ( Aux != 0 )   Imprime("Sim");
Senão             Imprime("NÃo");
```


```
// Caso ordenado

Aux  <--  0
Para  i <-- 1  Até  n
 {
    Se ( L[i] = x )
     {
        Aux  <--  i;    Break
     }
    Se ( L[i] > x )   Break
 }
Se ( Aux != 0 )   Imprime("Sim");
Senão             Imprime("NÃo");
```

E agora?

Bom, agora a gente lembra que a nossa análise é apenas mais ou menos.

Daí que a gente pode considerar aquilo que acontece no pior caso.

No pior caso, os laços dois dois algoritmo realizam n voltas — (*porque?*)

```
// Caso não ordenado
O(1) | Aux  <--  0
     | Para  i <-  1  Até  n
     |  {
n    |    | Se ( L[i] = x )
     |    |  {
O(1) |    |     Aux  <--  i
     |    |     Break
     |     }   }
O(1) | Se ( Aux != 0 )   Imprime("Sim");
     | Senão             Imprime("NÃo");
```

```
// Caso ordenado
O(1) | Aux  <--  0
     | Para  i <-- 1  Até  n
     |  {
n    |    | Se ( L[i] = x )
     |    |  {
O(1) |    |     Aux  <--  i;    Break
     |    |  }
     |    | Se ( L[i] > x )   Break
     |  }
O(1) | Se ( Aux != 0 )   Imprime("Sim");
     | Senão             Imprime("NÃo");
```

Com bases nas anotações, nós chegamos ao seguinte resultado

- caso não ordenado: $O(n)$
- caso ordenado: $O(n)$

Quer dizer, o ganho de eficiência que a gente obteve no caso ordenado não aparece na análise.

Por um lado, a gente pode interpretar isso como uma limitação do nosso método de análise.

Por outro lado, a gente pode pensar que os ganhos de eficiência que a gente obtém só são significativos se eles aparecem na análise.

E isso também faz sentido.

◇

c. Elemento repetido

Considere a tarefa

→ *Verificar se a lista contém algum elemento repetido*

Na primeira aula nós apresentamos um algoritmo para esse problema.

E abaixo nós já temos esse código anotado

```
     Para  k <-- 1  Até  n-1
      {
O(1) | x  <--  L[k];   Aux  <--  0
     | Para  i <-- k+1  Até  n
     |  {
n    |    | Se ( L[i] = x )
     |    |  {
n-k  |    |     Aux  <--  i;  Break
O(1) |    |  }   }
O(1) | Se ( Aux != 0 )  {  Imprime ("Sim");  Retorna  }
     |  }
O(1) | Imprime ("Não")
```

Aqui pela primeira vez nós temos um algoritmo com um laço dentro de um laço.

E para complicar as coisas, o número de voltas do laço mais interno depende da volta em que a gente está no laço mais externo.

Mas a gente pode simplificar as coisas assumindo que o laço mais interno sempre dá n voltas.

Nesse caso, a gente obtém a seguinte estimativa para o tempo de execução do algoritmo

$$\begin{aligned} & n \times \Big(O(1) \;+\; n \times O(1) \;+\; O(1) \Big) \;+\; O(1) \\ = \;& n \times O(1) \;+\; n^2 \times O(1) \;+\; n \times O(1) \;+\; O(1) \\ = \;& \cancel{O(n)} \;+\; O(n^2) \;+\; \cancel{O(n)} \;+\; \cancel{O(1)} \\ = \;& O(n^2) \end{aligned}$$

Note que dessa vez os termos $O(n)$ também foram descartados, porque na presença de $O(n^2)$ eles são termos menores.

Portanto a análise nos diz que o algoritmo executa em tempo $O(n^2)$.

Mas nesse ponto alguém pode perguntar

→ *Sera que ao reescrever $n - k$ como n a gente não acabou obtendo um resultado maior do que devia ?*

Bom, vamos raciocinar com mais precisão então.

Quer dizer, nós podemos observar que

- na primeira vez, o laço mais interno dá $n - 1$ voltas
- na segunda vez, o laço mais interno dá $n - 2$ voltas
- na terceira vez, o laço mais interno dá $n - 3$ voltas
- e assim por diante ...

Daí que o número total de voltas realizadas pelo laço mais interno pode ser calculado assim

$$(n-1) \;+\; (n-2) \;+\; (n-3) \;+\; \ldots \;+\; 3 \;+\; 2 \;+\; 1$$

O truque mais conhecido para calcular essa soma é o seguinte

$$\begin{array}{rccccccccccc} & (n-1) & + & (n-2) & + & \ldots & + & 2 & + & 1 & & \\ + & 1 & + & 2 & + & \ldots & + & (n-2) & + & (n-1) & & \\ \hline & n & + & n & + & \ldots & + & n & + & n & = & n^2 \end{array}$$

Quer dizer, duas vezes a soma é igual a n^2.

Logo a soma é igual a $n^2/2$.

Isso significa que a contribuição do laço mais interno para o tempo de execução do algoritmo é $O(n^2/2)$.

E isso já nos permite concluir que o algoritmo executa em tempo $O(n^2)$

— (*porque* $O(n^2/2)$ *é a mesma coisa que* $O(n^2)$)

Legal, não é?

◇

d. Par de elementos com soma k

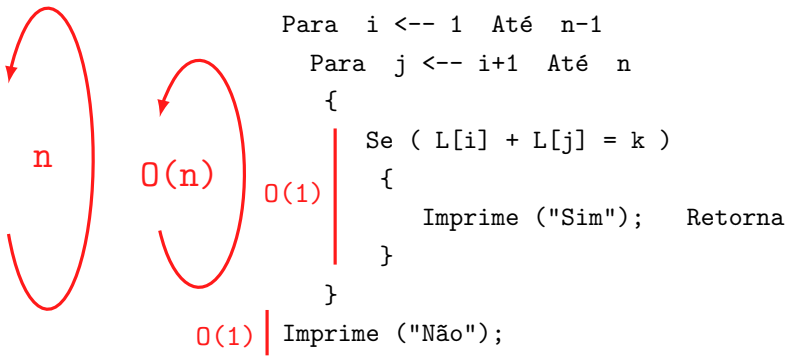
Agora considere a seguinte tarefa

→ *Verificar se existem dois elementos na lista cuja soma é igual a* k

Na primeira aula nós apresentamos dois algoritmos para esse problema: um para o caso não ordenado, e um para o caso ordenado.

A análise do algoritmo para o caso não ordenado é quase imediata.

Quer dizer, assumindo que o laço interno dá n voltas, nós obtemos a seguinte versão anotada do algoritmo



(*Nós anotamos o laço mais interno como* $O(n)$ *para indicar que isso é uma aproximação*)

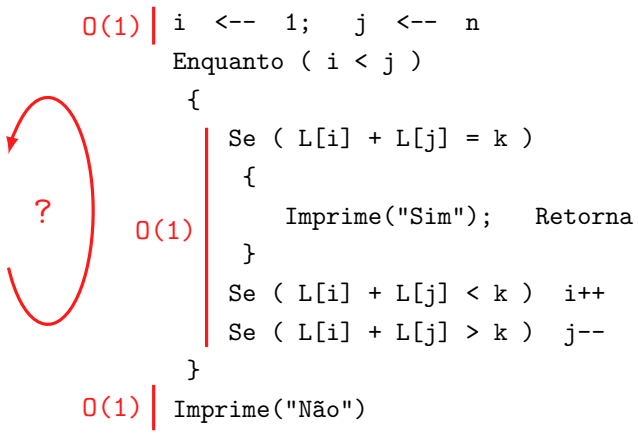
Agora com base nessas anotações nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

$$O(1) \quad + \quad n \times O(n) \times O(1) \quad + \quad O(1) \qquad = \qquad O(n^2)$$

Certo.

Mas a análise do algoritmo para o caso ordenado não é tão imediata assim.

Quer dizer, esse algoirtmo apresenta mais uma novidade: o laço `Enquanto`, que não possui um número fixo, pré-determinado de voltas.



Daí que, parte da análise consiste em estimar o número de voltas que o laço dá.

Nesse caso, nós podemos fazer as seguintes observações

- no início as variáveis `i`, `j` são inicializadas como

  ```
  i <-- 1;    j <-- n
  ```

  de modo que a diferença entre elas é $(\texttt{j} - \texttt{i}) = \texttt{n} - 1$.
- daí, a cada volta do laço
  - ou a variável `i` é incrementada
  - ou a variável `j` é decrementada

  de modo que a diferença sempre diminui de 1

Logo, o laço termina no máximo após n voltas.

Portanto, o algoritmo do caso ordenado executa em tempo

$$O(1) \quad + \quad O(n) \times O(1) \quad + \quad O(1) \qquad = \qquad O(n)$$

Abaixo nós temos os resultados das duas análises

- `caso não ordenado`: $O(n^2)$
- `caso ordenado`: $O(n)$

E aqui nós temos um primeiro exemplo onde o algoritmo para o caso ordenado é significativamente mais eficiente do que o algoritmo para o caso não ordenado.

◇

e. Trabalhando com duas listas

Imagine que L_1 e L_2 são duas listas de tamanho n.

E considere a tarefa

→ *Contar quantos elementos de* L_1 *são maiores que todos os elementos de* L_2

Na primeira aula nós vimos o seguinte algoritmo para esse problema

```
O(1) | cont  <--  0
       Para  k <-- 1  Até  n
        {
O(1) | x  <--  L1[k];   Aux  <--  0
          Para  i <-- 1  Até  n
           {
O(1) |       Se ( L2[i] >= x )
              {
                 Aux  <--  1;   Break
           }  }
O(1) | Se ( Aux = 0 )   cont ++
        }
O(1) | Imprime ("A resposta é:", cont)
```

E com base nas anotações nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

$$O(1) \quad + \quad n \cdot \Big(O(1) \quad + \quad n \cdot O(1) \quad + \quad O(1) \Big) \quad + \quad O(1) \qquad = \qquad O(n^2)$$

Mas nós também vimos um algoritmo que resolve o problema de maneira mais inteligente

```
          // 1a ETAPA: encontrar o maior elemento de L2

O(1) | M  <--  L2[1]
       Para  i <-- 1  Até  n
        {
O(1) |  Se ( L2[i] > M )    M  <--  L2[i]
        }

          // 2a ETAPA: contar elementos de L1 maiores do que M
O(1) | cont  <--  0
       Para  k <-- 1  Até  n
        {
O(1) |  Se ( L1[k] > M )   cont ++
        }
O(1) | Imprime ("A resposta é:", cont)
```

Com base nas anotações nós obtemos a seguinte estimativa para o tempo de execução do algoritmo

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

Quer dizer, as análises nos deram os seguintes resultados

- `primeira versão:` $O(n^2)$
- `versão inteligente:` $O(n)$

de modo que a versão inteligente é realmente bem mais eficiente.

◇