Aula 8
Análise de Complexidade de Algoritmos
(continuação da aula anterior)

Algoritmos e Estruturas de Dados

Análise

por modelos matemáticos

Análise por modelos matemáticos

- Objetivo
  - Encontrar um modelo matemático que me aproxime a complexidade de um algoritmo, em função do tamanho/complexidade do problema de entrada n.

Análise p/ modelos matemáticos

- Tentar estimar a ordem de crescimento de forma analítica
  - Custo operação * frequência
- Começar por determinar o custo de operações básicas

Instrução	Exemplo	Tempo
Declaração de variáveis	int a	c1
atribuição	a = b	c2
Comparação de inteiros	a < b	c3
acesso a um elemento de um array	a[i]	c4
tamanho de um array	a.length	c5
alocação de array	new int[n]	c6 n
alocação de array bidimensional	new int[n][n]	c7 n^2

Exemplo: 1-sum

- Contar o número de 0s num array

```
int count = 0;
for(int i = 0; i < n; i++)
{
    if(a[i] == 0)
        count++;
}
```

Instrução	frequência	Custo	Tempo estimado
declaração de variáveis	2	c1	2c1
atribuição	2	c2	2c2
comparação <	n+1	c3	(n+1)c3
Comparação ==	n	c4	n c4
acesso array	n	c5	n c5
incremento	n a 2n	c6	n c6 a 2n c6

Exemplo: 1-sum

- Contar o número de 0s

```
int count = 0;
for(int i = 0; i < n; i+
{
    if(a[i] == 0)
        count++;
}
```

Observação:
Na realidade eu não estou interessado em saber o tempo exato, mas sim saber como é que o tempo aumenta em função de n.

Portanto posso ignorar todos os custos constantes que não dependam de n

Instrução	frequência	Custo	Tempo estimado
~~declaração de variáveis~~	~~2~~	~~c1~~	2c1
~~atribuição~~	~~2~~	~~c2~~	2c2
comparação <	n+1	c3	(n+1)c3
Comparação ==	n	c4	n c4
acesso array	n	c5	n c5
incremento	n a 2n	c6	n c6 a 2n c6

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

??

Instrução	frequência
declaração de variáveis	$n + 2$
atribuição	$n + 2$
comparação <	$(n + 1) + ?$
Comparação ==	?
acesso array	2 x ?
incremento	?

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

- Número de execuções do 2.º for
  - n-1 + n-2 + … + 2 + 1 + 0

Soma dos n primeiros termos de uma progressão aritmética

$$S_n = \frac{n(a_1 + a_n)}{2}$$

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

- Número de execuções do corpo do 2.º for
  - n-1 + n-2 + … + 2 + 1 + 0
  - $= \frac{n((n-1)+0)}{2}$ = ½ n (n – 1)

Soma dos n primeiros termos de uma progressão aritmética

$$S_n = \frac{n(a_1 + a_n)}{2}$$

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

- Número de execuções do 2.º for
  - n-1 + n-2 + … + 2 + 1 + 0
  - = ½ n (n – 1)

Instrução	frequência
declaração de variáveis	$n + 2$
atribuição	$n + 2$
comparação <	$(n + 1) + \underline{\tfrac{1}{2}\, n\, (n + 1)}$
Comparação ==	$\tfrac{1}{2}\, n\, (n - 1)$
acesso array	$2 \times \tfrac{1}{2}\, n\, (n - 1)$
incremento	de $\tfrac{1}{2}\, n\, (n - 1)$ a $n\, (n - 1)$

Esta expressão é ligeiramente diferente pq o número de comparações é dado por n + n-1 + … + 1

Notação tilde

Notação tilde

- Problema:
  - Não é nada prático ter de fazer estes cálculos todos para estimar a complexidade temporal de um algoritmo
  - Na realidade eu não estou interessado em saber o tempo exato, mas sim saber como é que o tempo aumenta em função de n.
  - Será que não conseguimos simplificar ainda mais o processo?

Instrução	frequência
declaração de variáveis	$n + 2$
atribuição	$n + 2$
comparação <	$(n + 1) + ½\, n\, (n + 1)$
Comparação ==	$½\, n\, (n - 1)$
acesso array	2 x ½ n (n – 1)
incremento	de $½\, n\, (n - 1)$ a $n\, (n - 1)$

Notação tilde

- Ideia:
  - As expressões têm termos dominantes e não dominantes
  - Vamos ignorar termos de menor magnitude numa expressão

- Porquê?
  - quando a complexidade de *n* é grande

    erro é negligenciável

  - quando a complexidade de *n* é pequena

    não queremos saber (não estamos preocupados com a eficiência)

Ex: $\frac{1}{6}n^3 + 20\,n + 16$

Notação tilde

- Ideia:
  - As expressões têm termos dominantes e não dominantes
  - Vamos ignorar termos de menor magnitude numa expressão

- Porquê?
  - quando a complexidade de *n* é grande

    erro é negligenciável

  - quando a complexidade de *n* é pequena

    não queremos saber (não estamos preocupados com a eficiência)

Ex: $\frac{1}{6}n^3 + 20\,n + 16 \;\rightarrow \sim \frac{1}{6}n^3$

Para n=1000, o erro devido a esta simplificação é apenas de 0.002%!

Notação tilde

- Definição matemática para aproximação tilde

Def:
uma função f é aproximada por uma função g

$$f(n) \sim g(n)$$

Se e só se

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 1$$

Ex: $\frac{1}{6}n^3 + 20\,n + 16 \rightarrow \sim \frac{1}{6}n^3$

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

Número de execuções do corpo do 2.º for

n-1 + n-2 + … + 2 + 1 + 0

= ½ n (n – 1) = ½ n^2 – ½ n

Instrução	frequência	notação tilde
declaração de variáveis	$n + 2$	~n
atribuição	$n + 2$	~n
comparação <	$(n + 1) + ½ (n + 1) (n - 1)$	~½ n^2
Comparação ==	½ $n (n - 1)$	~½ n^2
acesso array	2 x ½ n (n – 1)	~n^2
incremento	½ $n (n - 1)$ a $n (n - 1)$	~½ n^2 a ~n^2

Análise

assimptótica

Análise assimptótica

- Análise da eficiência/complexidade assimptótica de algoritmos
  - Estamos apenas preocupados em estudar a complexidade do algoritmo

    Quando o tamanho/complexidade do input ***aumenta de forma ilimitada***

  - Só a ordem de crescimento é relevante

- Ou, como eu gosto de lhe chamar

  "análise para preguiçosos"

  "TLDR de análise de complexidade"

Notação O (Oh-grande)

- **Notação Oh-Grande *(Big O notation)***
  - *Notação matemática usada para descrever o comportamento de uma função quando o seu argumento tende para infinito*

- $f(n) = O(g(n))$, quando $n \rightarrow \infty$

Para abreviar, costuma-se escrever:
f(n) = O(g(n))

- *O(g(n))* diz-se **o limite superior** para a ordem de crescimento de uma função *f*

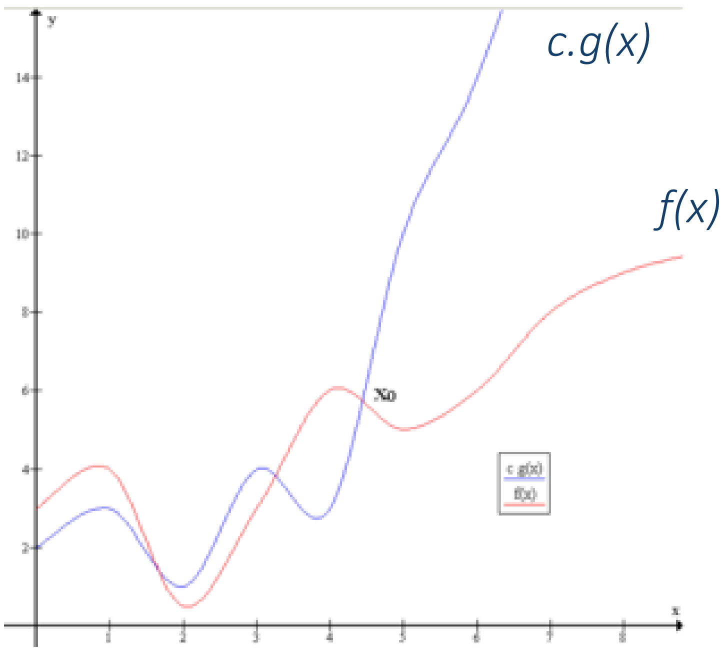
- Diz-se que *f* tem ordem de crescimento no **pior caso** de *O(g(n))*

Notação O (Oh-grande)

- **Definição matemática**

- f(n) = <u>O(g(n))</u>
  - sse existem constantes positivas **c** e **n_0** tais que
  - $0 \leq f(n) \leq c.g(n)$
  - para todo $n \geq n_0$

- Diz-se que f tem ordem de crescimento no **pior caso** de O(g(n))

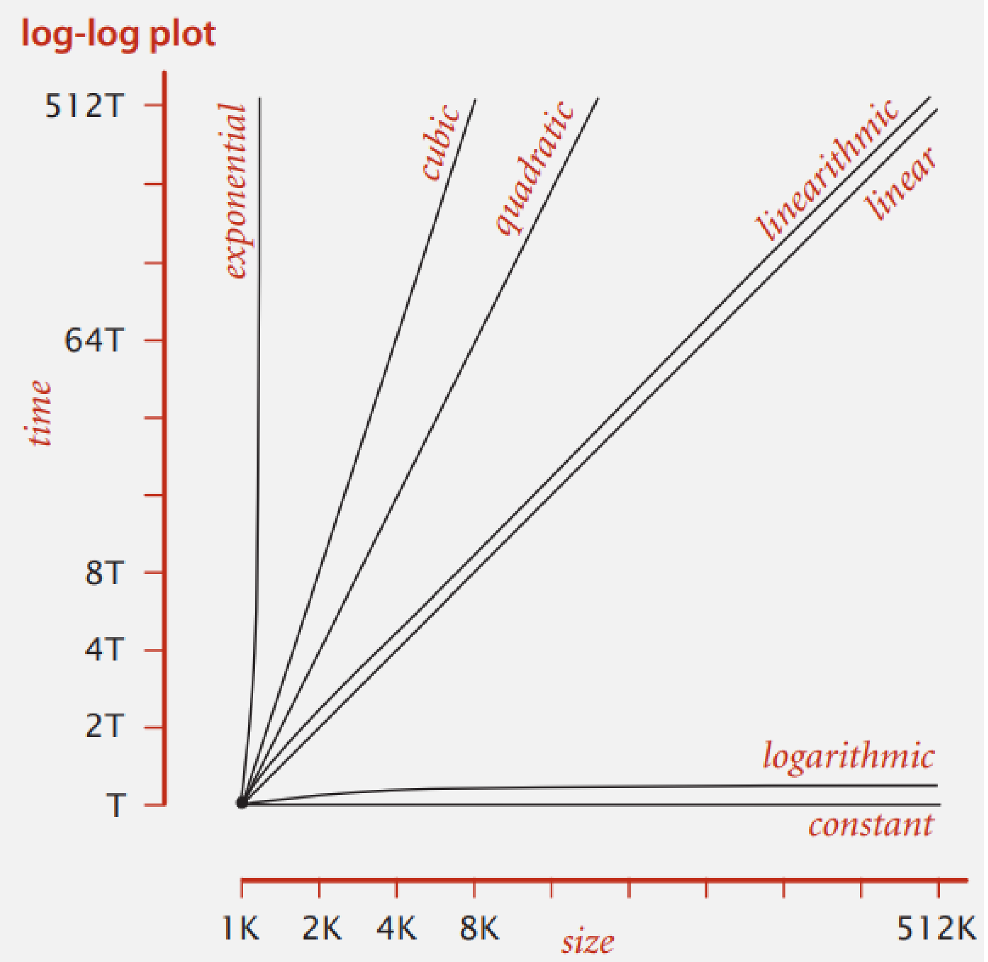


Existe um valor **n_0** a partir do qual, a função f(x) tem sempre complexidade inferior a c.g(x)

Ordens de crescimento típicas

- $g(n) = 1$
- $g(n) = \log n$
- $g(n) = n \log n$
- $g(n) = n$
- $g(n) = n^2$
- $g(n) = n^3$
- $g(n) = 2^n$

Permitem-nos estudar a ordem de crescimento da maior parte dos algoritmos



Typical orders of growth

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < n; i++)
{
    for(int j = i+1; j < n; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

Instrução	frequência	complexidade assimptótica
declaração de variáveis	$n + 2$	O(n)
atribuição	$n + 2$	O(n)
comparação <	$(n + 1) + ½ (n + 1) (n - 1)$	$O(n^2)$
Comparação ==	$½ n (n - 1)$	$O(n^2)$
acesso array	2 x ½ n (n – 1)	$O(n^2)$
incremento	$½ n (n - 1)$ a $n (n - 1)$	$O(n^2)$

Exemplo: 2-sum

```
int count = 0;
for(int i = 0; i < N; i++)
{
    for(int j = i+1; j < N; j++)
    {
        if(a[i] + a[j] == 0)
            count++;
    }
}
```

Tal como fizemos na notação tilde, numa soma de instruções, podemos ignorar os termos não dominantes

Abuso de notação mas...

O(n) +
O(n) +
$O(n^2)$ +
$O(n^2)$ +
$O(n^2)$ +
$O(n^2)$
$= \sim 4n^2 = O(n^2)$

Ao contrário da notação tilde, quando usamos notação assimptótica podemos ignorar o 4, pois pela própria definição de O:
$f(n) = O(4n^2) \rightarrow f(n) = O(n^2)$

Para provar isto, basta escolher a constante c2 = 4 c1

Exemplo 3-sum

```
public static int threeSum(int[] a)
{
   int n = a.length;
   int count = 0;

   for(int i = 0; i < n; i++)
   {

      for(int j = i+1; j < n; j++)
      {

         for(int k = j+1; k < n; k++)
         {

            if(a[i]+a[j]+a[k] == 0)
               count++;
         }
      }
   }
   return count;
}
```

O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)

O(1)
O(1)
)
)
)
O(1)

Exemplo 3-sum

Dica:
podemos ignorar tudo o que seja O(1)
pois nunca irá afectar a ordem de crescimento

```java
public static int threeSum(int[] a)
{
    int n = a.length;
    int count = 0;

    for(int i = 0; i < n; i++)
    {

        for(int j = i+1; j < n; j++)
        {

            for(int k = j+1; k < n; k++)
            {

                if(a[i]+a[j]+a[k] == 0)
                    count++;
            }
        }
    }
    return count;
}
```

O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)
O(1)
O(n) x (
O(1)
O(1)
O(1)
O(1)
)
)
)
O(1)

Exemplo 3-sum

Dica:
podemos ignorar tudo o que seja O(1)
pois nunca irá afectar a ordem de crescimento

```
public static int threeSum(int[] a)
{
   int n = a.length;
   int count = 0;

   for(int i = 0; i < n; i++)              → O(n) x (
   {

      for(int j = i+1; j < n; j++)         → O(n) x (
      {

         for(int k = j+1; k < n; k++)      → O(n) x (
         {

            if(a[i]+a[j]+a[k] == 0)
               count++;
         }                                            )
      }                                          )
   }                                        )
   return count;
}
```

$\boxed{= O(n^3)}$

Exemplos de ordens de crescimento

description	order of growth	typical code framework	description	example
constant	1	`a = b + c;`	*statement*	*add two numbers*
logarithmic	$\log N$	[*see page 47*]	*divide in half*	*binary search*

linear	N	`double max = a[0];`
		`for (int i = 1; i < N; i++)`
		`   if (a[i] > max) max = a[i];`
			loop	*find the maximum*

linearithmic	$N \log N$	[*see* ALGORITHM 2.4]	*divide and conquer*	*mergesort*

quadratic	N^2	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      if (a[i] + a[j] == 0)`
		`         cnt++;`
			double loop	*check all pairs*

cubic	N^3	`for (int i = 0; i < N; i++)`
		`   for (int j = i+1; j < N; j++)`
		`      for (int k = j+1; k < N; k++)`
		`         if (a[i] + a[j] + a[k] == 0)`
		`            cnt++;`
			triple loop	*check all triples*

exponential	2^N	[*see* CHAPTER 6]	*exhaustive search*	*check all subsets*

Notação Ω (Omega-grande)

- **limite inferior** para a ordem de crescimento de uma função f

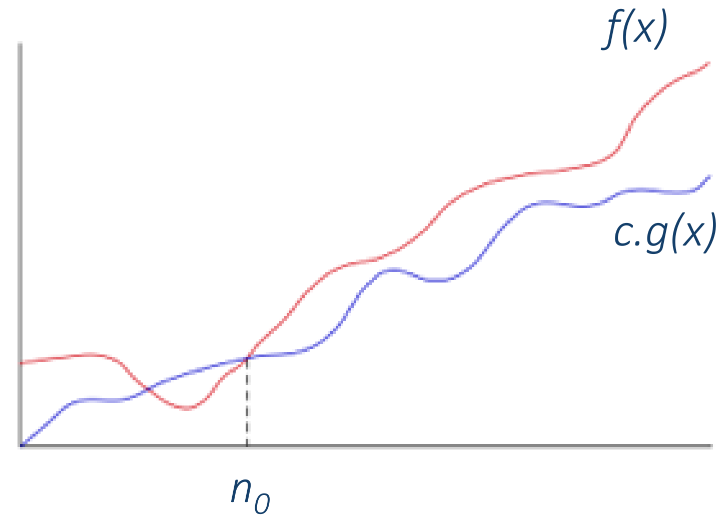
- f(n) = <u>Ω(g(n))</u>

  sse existem constantes positivas c e n_0 tais que

  $$0 \leq c.g(n) \leq f(n)$$

  para todo $n \geq n_0$

- Diz-se que f tem ordem de crescimento no **melhor caso** de Ω(g(n))



Existe um valor n_0 a partir do qual, a função f(x) tem sempre complexidade superior a c.g(x)

Notação Θ (Theta-grande)

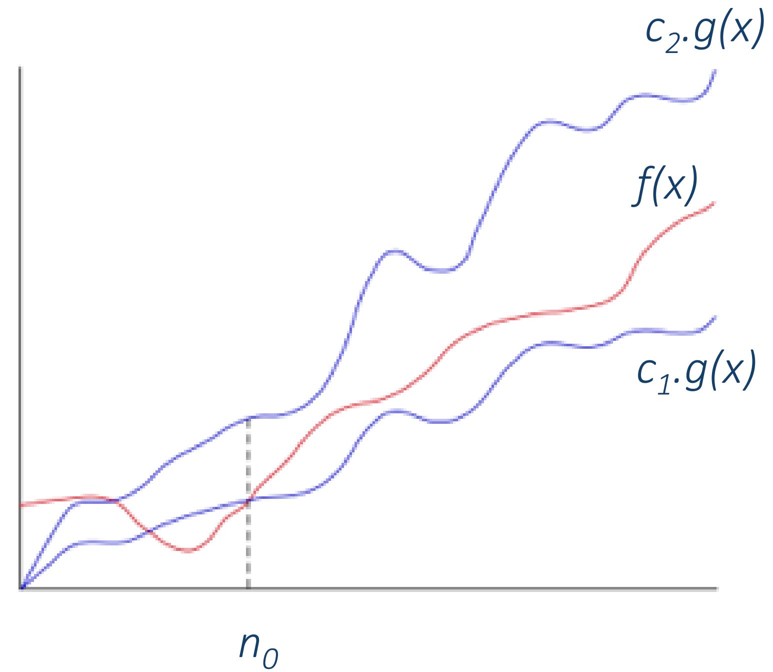
- **limite assimptóticamente restrito** para a ordem de crescimento de uma função f

- f(n) = <u>Θ(g(n))</u>

  sse existem constantes positivas c1 c2 e n_0 tais que

  $$0 \leq c_1 . g(n) \leq f(n) \leq c_2 . g(n)$$

  para todo $n \geq n_0$



Existe um valor n_0 a partir do qual, a função f(x) tem sempre complexidade entre $c_1.g(x)$ e $c_2.g(x)$

Relações entre notações

- $f(n) = \Theta(g(n))$

  (=)

  $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

- Grande-Theta implica g(n) é ao mesmo tempo um limite superior e inferior

Utilização de notações

	Exemplo	usado para...
Grande-Theta	Θ(n)	Classificar algoritmos
Grande-Oh	O(n)	definir limites superiores, i.e. no pior caso
Grande-Omega	Ω(n)	definir limites inferiores, i.e. no melhor caso

Devido a

- Ω(n) ser pouco relevante na prática
- Θ(n) -> O(n)
- O(n) ser o mais relevante para um eng. informático

Muito frequentemente usa-se a notação O(n) para análise de algoritmos

- Mesmo em situações onde o mais correcto seria Θ(n)

Importância

Análise assimptótica

Importância análise assimptótica

- Muito importante na análise e comparação de algoritmos
  - Um algoritmo com maior ordem de crescimento é sempre pior

    A não ser para alguns casos com n pequeno

- Heurística simples e fácil de usar para
  - Juntamente com notação tilde
  - Ter uma ideia de alto nível da eficiência de um programa
  - Evitar erros catastróficos de eficiência

Erro vs erro catastrófico

Legal action taken over €18.5m cost of fixing Barrow bridge 'design errors'

Alleged errors led to 10-month delay in completion of N25 link between Kilkenny and Wexford

Mon, Jun 8, 2020, 19:55

Mary Carolan

The 887m Barrow Bridge, also known as the Rose Fitzgerald Kennedy bridge, ispart of the N25 New Ross bypass project.

EPIC FAIL!!

colapso da Tacoma bridge

Erro não catastrófico

```
public int compareTo(Jogador j2) {
    List<String> posicoes = new ArrayList<>(Arrays.asList("Guarda-redes", "Defesa", "Medio",
"Avancado"));
    int comparePosicoes = posicoes.indexOf(getPosicao()) - posicoes.indexOf(j2.getPosicao());
    int compareGolos =j2.getGolos() - getGolos();
    int compareInternacionalizacoes =j2.getInternacionalizacoes() - getInternacionalizacoes();
    int comparenomeClube = getNome().compareTo(j2.getNome());
    if(comparePosicoes != 0)
        return comparePosicoes;
    if(compareGolos != 0)
        return compareGolos;
    if(compareInternacionalizacoes != 0)
        return compareInternacionalizacoes;
    else
        return comparenomeClube;
}
```

- Embora a ordem de crescimento não seja má O(1)
  - Podemos evitar algumas operações de custo não trivial
  - $4\ c_1 + 4c_2 + 4c_2$ na comparação de posições

Erro vs erro catastrófico

$O(n^2)$

```
static List<String> listaNomesClubes(List<Jogador> jogadores){
    List<String> result = new ArrayList<>();
    int i = 0;
    int iguais = 0;
    while(i < jogadores.size()) { O(n)
        for(int j = 0; j < result.size(); j++) { O(n)
            if(jogadores.get(i).getClube().compareTo(result.get(j)) != 0)
                iguais++;
        }
        if(iguais == result.size())
            result.add(jogadores.get(i).getClube());
        iguais = 0;
        i++;
    }
    return result;
}
```

- Este método não tem nenhum erro grave
  - Embora possa ser tornado + eficiente

- Importante: no pior caso, $O(n^2)$

Erro catastrófico

```
static List<Clube> calculaListaClubes(List<Jogador> jogadores){
    List<Clube> result = new ArrayList<>();

    for(int i = 0; i < listaNomesClubes(jogadores).size(); i++) {

        List<Jogador> jogadoresdoClube = filtraPorClube(jogadores, listaNomesClubes(jogadores).get(i));
        Clube clubeEinter = new Clube(listaNomesClubes(jogadores).get(i), totalIntern(jogadoresdoClube));
        result.add(clubeEinter);
    }
    return result;
}
```

UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Erro catastrófico

Esta chamada devia ser feita apenas 1 vez, antes do *for*

$O(n^2)$ $O(n)$ $O(n)$ $O(n^2)$

```
static List<Clube> calculaListaClubes(List<Jogador> jogadores){
    List<Clube> result = new ArrayList<>();

    for(int i = 0; i < listaNomesClubes(jogadores).size(); i++) {

        List<Jogador> jogadoresdoClube = filtraPorClube(jogadores, listaNomesClubes(jogadores).get(i));
        Clube clubeEinter = new Clube(listaNomesClubes(jogadores).get(i), totalIntern(jogadoresdoClube));
        result.add(clubeEinter);
    }
    return result;
}
```

$O(n^2)$ $O(n)$

Erro catastrófico

Esta chamada devia ser feita apenas 1 vez, antes do *for*

```
static List<Clube> calculaListaClubes(List<Jogador> jogadores){
    List<Clube> result = new ArrayList<>();

    for(int i = 0; i < listaNomesClubes(jogadores).size(); i++) {

        List<Jogador> jogadoresdoClube = filtraPorClube(jogadores, listaNomesClubes(jogadores).get(i));
        Clube clubeEinter = new Clube(listaNomesClubes(jogadores).get(i), totalIntern(jogadoresdoClube));
        result.add(clubeEinter);
    }
    return result;
}
```

$O(n)$ — for

$O(n^2)$ — listaNomesClubes(jogadores).size()

$O(n)$ — filtraPorClube

$O(n^2)$ — listaNomesClubes(jogadores).get(i)

$O(n^2)$ — listaNomesClubes(jogadores).get(i)

$O(n)$ — totalIntern

- Versão eficiente
  - T(n)

    $= O(n^2) + O(n) \times (O(n) + O(n))$

    $= O(n^2) + O(n^2)$

    $= O(n^2)$

- Esta versão
  - T(n)

    $= O(n) \times (O(n^2)+O(n)+O(n^2)+O(n^2) +O(n))$

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

    $= O(n^3)$!!!!

Consequências de um erro catastrófico

- Quão mau é este erro?
- Para n = 100, $O(n^3)$ é 100x pior que $O(n^2)$

  0.1segundos → 10segundos

- Para n = 1000, $O(n^3)$ é 1000x pior que $O(n^2)$

  10segundos → 2h46m

- Para n = 10 000, $O(n^3)$ é 10 000x pior que $O(n^2)$

  16minutos → 115 dias....