ENGENHARIA INFORMÁTICA E DE COMPUTADORES

Algoritmos e Estruturas de Dados

(parte 5 – Notação Assintótica e Recorrências)

2º Semestre 2022/2023

Instituto Superior de Engenharia de Lisboa
Paula Graça

NOTAÇÃO GRANDE-O

- Os algoritmos têm tempos de execução proporcionais ao crescimento das funções:
  - 1 – tempo constante
    - O número de operações é o mesmo para qualquer dimensão da entrada (muitas instruções são executadas uma só vez ou poucas vezes)
  - lg N - tempo logarítmico
    - Cresce ligeiramente à medida que N cresce
    - Típico em algoritmos do tipo dividir para conquistar, quando se visita apenas cada uma das metades (ex: binary search)
  - N – tempo linear
    - Se n duplica, o número de operações também duplica
    - Típico quando é necessário processar N dados de entrada (ex: sequential search)

NOTAÇÃO GRANDE-O

- N lg N – tempo linear logaritmo
  - Típico em algoritmos do tipo dividir para conquistar, quando se visitam ambas as metades (merge sort)
- N^2 – tempo quadrático
  - Se n duplica, o número de operações quadruplica
  - Típico quando é preciso processar todos os pares de dados de entrada - ex: soma de matrizes
- N^3 – tempo cúbico
  - Para N = 100, N^3 = 1 milhão - ex: produto de matrizes
- 2^N – tempo exponencial
  - Provavelmente com pouca aplicação prática
  - Típico em soluções de força bruta (ex: torres de hanoi)
  - Para N = 20, 2^N = 1 milhão
- N! – tempo exponencial (ex: caixeiro viajante)

NOTAÇÃO GRANDE-O

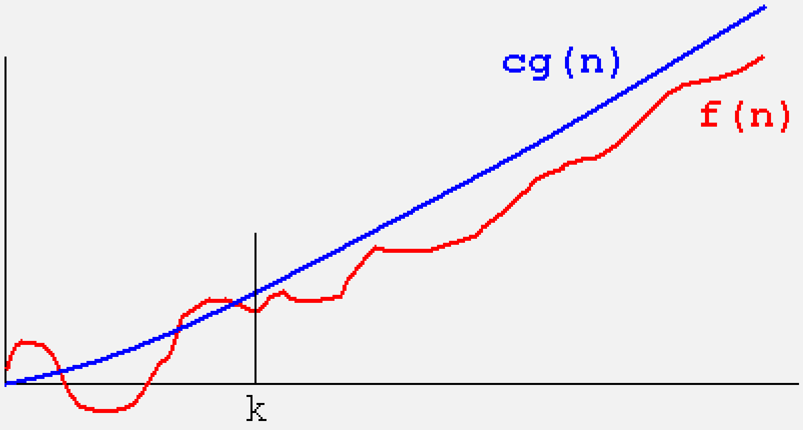
- A notação grande-*O*, é usada para descrever como é que a dimensão da entrada de um algoritmo afeta o seu grau de crescimento
- Permite classificar algoritmos de acordo com os limites superiores do seu tempo de execução

  - Definição:

  Uma função f(n) diz-se ser *O* (g(n)), ou seja, f(n) é superiormente limitada por g(n)), se existem duas constantes positivas c (ϵ R^+) e k (ϵ N_0) tais que

  f(n) <= c.g(n) para todo o n>= k

Logo, f = *O*(g) significa que f ∈ *O*(g), i.e., f pertence ao conjunto de funções limitadas superiormente por g a partir de certa ordem (n >= k)



NOTAÇÃO GRANDE-*O*

- Exemplo:

  $f(n) = 100n + 5$ é $O(n)$?

- Prova: dado $c = 101$ e $k = 5$

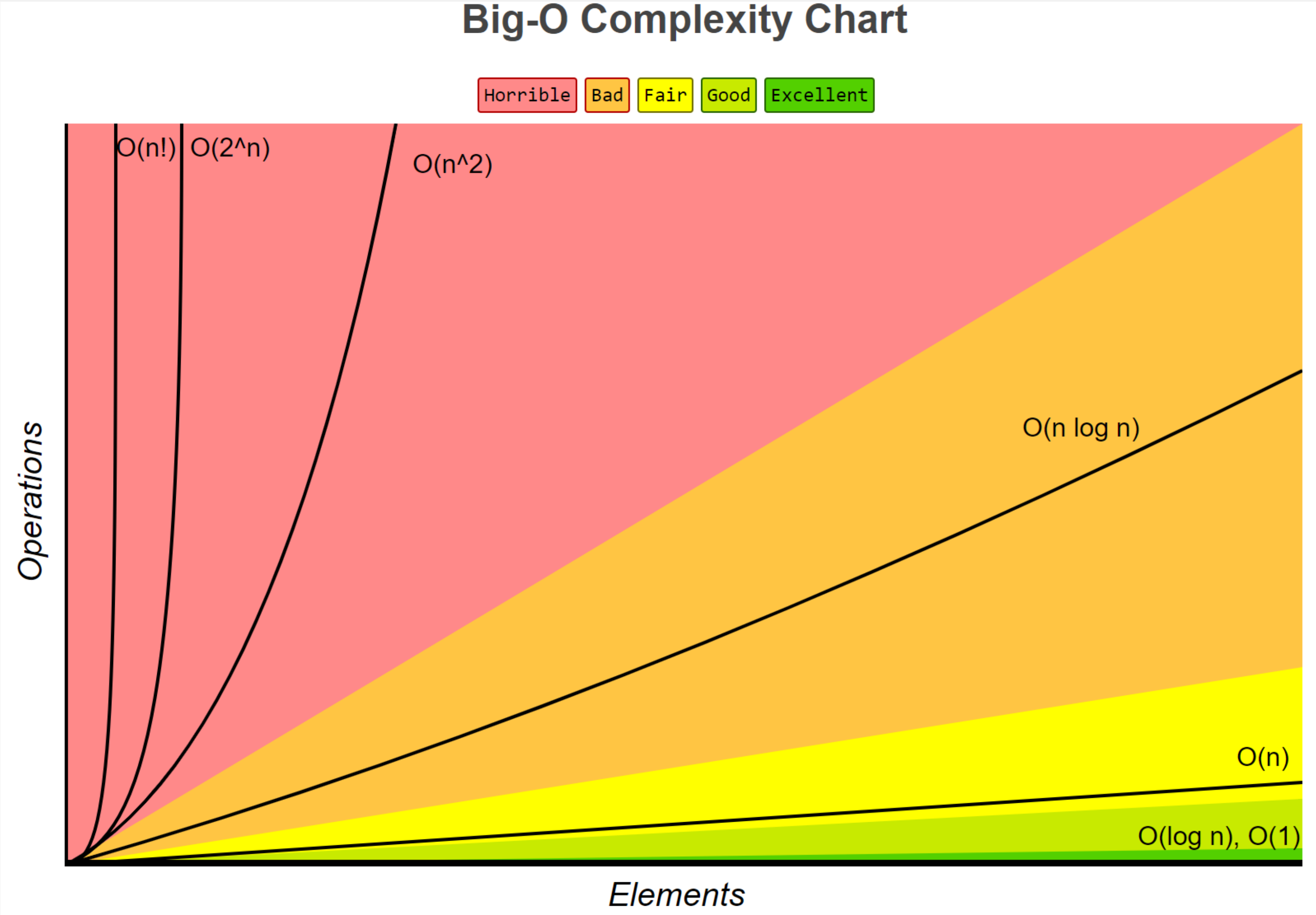
f(n)	<=	c.n
100 . 5+5 = 505	<=	101 . 5 = 505
100 . 6+5 = 605	<=	101 . 6 = 606
100 . 7+5 = 705	<=	101 . 7 = 707

Podemos dizer que $f(n)$ é $O(n)$ pois existe um $c = 101$ e $k = 5$ em que $100n + 5 <= c.n$ para $n >= k$

NOTAÇÃO GRANDE-*O*

- As Classes de Complexidade algorítmica são definidas de acordo com uma ordem de complexidade – notação grande *O*
  - Classes de complexidade ordenadas por ordem crescente de esforço: *O* (1)
    - *O* (1) Constante
    - *O* (lg n) Logarítmica
    - *O* (n) Linear
    - *O* (n lg n) Linear logarítmica
    - *O* (n^2) Quadrática
    - *O* (n^3) Cúbica
    - *O* (n^k) Polinomial
    - *O* (2^n) Exponencial
    - *O* (n!) Exponencial
  - Os "melhores" algoritmos do ponto de vista de tempo de execução, são os algoritmos de classe de complexidade constante ou logarítmica

NOTAÇÃO GRANDE-O



NOTAÇÃO GRANDE-*O*

- Sendo f(n) uma função que representa a variação do tempo de execução [C(n)$\Leftrightarrow$T(n)] de um algoritmo, então a afirmação

  "f(n) é *O* (g(n))" ou

  "f(n) pertence à classe de complexidade *O*(g(n))"

- É habitualmente escrita como

  $f(n) = O(g(n))$

- Significando que

  $f(n) \in O(g(n))$

dando uma noção da ordem de crescimento do tempo de execução f(n)

NOTAÇÃO GRANDE-O

- Propriedades

  - Constantes

$c \,.\, O\,(f(n)) = O\,(c \,.\, F(n)) = O\,(f(n))$ em que c>0

  - Soma

$O\,(f(n)) + O\,(g(n)) = O\,(f(n) + g(n)) = O\,(\max(f(n), g(n)))$

  - Multiplicação

$O\,(f(n)) \,.\, O\,(g(n)) = O\,(f(n) \,.\, g(n))$

NOTAÇÃO GRANDE-*O*

- Considerando o seguinte código

```
for (i in 1..N ) {
      instruções;
}
```

- Número de instruções:
  - N iterações
  - Em cada iteração são executadas um conjunto de instruções em tempo constante
  - *O* (N)   linear

NOTAÇÃO GRANDE-*O*

- Considerando o seguinte código

```
for (i in 1..N) {
	for (j in 1..N) {
		instruções;
	}
}
```

- Número de instruções:
  - O primeiro ciclo tem N iterações
  - Cada iteração é executada N vezes
  - $O(N^2)$ quadrática

NOTAÇÃO GRANDE-O

- Considerando o seguinte código

```
for (i in 1..N) {
	for (j in i..N) {
		instruções;
	}
}
```

- Número de instruções:
  - O primeiro ciclo tem N iterações, decrementando 1 cada vez que executa o ciclo mais interior

$$\underbrace{\hspace{10cm}}$$

$$N + (N-1) + (N-2) + \ldots + 3 + 2 + 1 = N(N+1)/2 \approx \tfrac{1}{2}.n^2$$

  - *O* (N^2) quadrática

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

NOTAÇÃO GRANDE-*O*

- A notação grande-*O* pode ser usada em conjunção com outros operadores aritméticos

$$C(n) = O(n^2) + 6n^2 + 2n + 1$$

- As fórmulas com termos contendo *O*(…) dizem-se expressões assintóticas
    - A palavra assintótico deriva do grego *asymptotos* que significa "não coincidente". Designa a recta que se aproxima indefinidamente de uma determinada curva, mas sem que ambas coincidam

NOTAÇÃO GRANDE-O

- Exemplo
  - Supondo um algoritmo com entrada de dimensão N, pretende-se saber qual o seu tempo de execução

    - O algoritmo inicialmente executa uma função de ordenação, conhecida com sendo de classe de complexidade $O(n^2)$
    - Depois o algoritmo executa uma função no tempo adicional de $6n^2+2n+1$ e termina
    - A classe de complexidade total do algoritmo pode ser expressa:

$$
\begin{aligned}
C(n) &= O(n^2) + 6n^2 + 2n + 1 \\
&= O(n^2) + O(6n^2) + O(2n) + O(1) \\
&= O(n^2) + O(n^2) + O(n) + O(1) \\
&= O(n^2)
\end{aligned}
$$

RECORRÊNCIAS BÁSICAS

- Considerar o seguinte algoritmo que determina o valor máximo de uma tabela

```
fun maxOfArray(a: IntArray, left: Int, right: Int, max: Int): Int {
    return if (left > right) max
        else if (a[left] > max) maxOfArray(a, left + 1, right, a[left])
        else maxOfArray(a, left + 1, right, max)
}
```

RECORRÊNCIAS BÁSICAS

- Na invocação do método *maxOfArray()*,
  - É executado um conjunto de instruções em tempo constante $O(1)$
  - O mesmo método *maxOfArray()*, é executado de novo com N-1 elementos

- Número total de instruções executadas

  $$C(N) = C(N-1) + O(1)$$

- Trata-se de uma relação de recorrência, ou simplesmente recorrência, pois é descrita em termos dela própria

RECORRÊNCIAS BÁSICAS

- Uma equação de recorrência expressa o valor de uma função f para um argumento n em termos de f (dela própria) para valores menores que n

$$f(n) = \begin{cases} 1 & \text{se } n = 0 \\ f(n-1) + 1 & \text{se } n >= 1 \end{cases}$$

- O tempo de execução de um algoritmo recursivo é expresso usando uma equação de recorrência ou recorrência, a qual descreve o tempo total de execução de um problema de dimensão n, descrita em termos do tempo de execução com entradas de dimensão menor que n

$$C(N) = \begin{cases} O(1) & \text{se } n = 0 \\ C(N-1) + O(1) & \text{se } n >= 1 \end{cases}$$

RECORRÊNCIAS BÁSICAS

- Numa relação de recorrência são descritas duas equações

Para n = 0 elementos, é resolvido em tempo constante

Padrão do caso geral

$$\begin{cases} C(0) = O(1) \\ C(N) = C(N-1) + O(1) \end{cases}$$

- A recorrência resolve-se por substituição das sucessivas instâncias da equação do caso geral, até se identificar um padrão
- Depois, utiliza-se a equação quando n = 0, para concluir a resolução

RECORRÊNCIAS BÁSICAS

- Resolução pelo método da substituição

$$
\begin{aligned}
C(N) &= C(N-1) + 1 \\
&= [C(N-2) + 1] + 1 \\
&= C(N-2) + 2 \\
&= [C(N-3) + 1] + 2 \\
&= C(N-3) + 3 \\
&= C(N-k) + k \\
&= C(0) + N \\
&= 1 + N \\
&= O(1) + O(N) \;=\; O(N)
\end{aligned}
$$

linear

Simplifica-se a expressão, substituindo $O(1)$ por 1, dado que é equivalente em notação assintótica

Padrão do caso geral

A derivação termina em C(0) quando N-k = 0, ou seja, k = N

Substituindo k = N
Sendo C(0) = 1

ANÁLISE DO MERGE SORT

```
fun mergeSort(table: DoubleArray, left: Int, right: Int) {
    if (left < right) {
        val mid = (right + left)/ 2
        val tableLeft = DoubleArray(mid - left + 1)
        val tableRight = DoubleArray(right - mid)
        //divide os elementos de table por tableLeft e tableRight
        divide(table, tableLeft, tableRight, left, mid, right)

        mergeSort(tableLeft, 0, mid)                  //repete para tableLeft
        mergeSort(tableRight, 0, right - mid - 1)     //repete para tableRight

        // junta ambas as metades em table, ordenando-as
        merge(table, tableLeft, tableRight, left, mid, right)
    }
}
```

$$C(n) = O(1) + O(divide) + C\left({}^{n}/_{2}\right) + C\left({}^{n}/_{2}\right) + O(merge)$$

ANÁLISE DO MERGE SORT

```
fun divide(t: DoubleArray, tLeft: DoubleArray, tRight: DoubleArray, left: Int,
                     mid: Int, right: Int) {
    for (i in left..mid)               // copia para tLeft
        tLeft[i] = t[i]
    for (i in mid + 1..right)
        tRight[i - mid - 1] = t[i] // copia para tRight
}
```

$$C(n) = O\left({}^{n}/_{2}\right) + O\left({}^{n}/_{2}\right) = O(n)$$

ANÁLISE DO MERGE SORT

```
fun merge(t: DoubleArray, tLeft: DoubleArray, tRight: DoubleArray, left: Int,
                    mid: Int, right: Int) {
   var i = 0
   var j = 0
   var k = left
   // faz o merge ordenado de tLeft com tRight
   while (i < tLeft.size && j < tRight.size)
      if (tLeft[i] < tRight[j])
         t[k++] = tLeft[i++]
      else t[k++] = tRight[j++]
   while (i < tLeft.size)        // copia os restantes elementos de tLeft
      t[k++] = tLeft[i++]
   while (j < tRight.size)       // copia os restantes elementos de tRight
      t[k++] = tRight[j++]
}
```

$$C(n) = O(n) + O\left({}^{n}/_{2}\right) + O\left({}^{n}/_{2}\right) = O(n)$$

ANÁLISE DO MERGE SORT

- Análise do algoritmo *Merge Sort* através da sua relação de recorrência

Quando n = 1 elemento, executa em tempo constante

$$\begin{cases} C(1) = O(1) \\ C(N) = O(1) + \underbrace{O(N)}_{\text{divide}} + 2C(N/2) + \underbrace{O(N)}_{\text{merge}} \end{cases}$$

Simplificando

$$\begin{cases} C(1) = O(1) \\ C(N) = 2C(N/2) + O(N) \end{cases}$$

RECORRÊNCIAS BÁSICAS

- Resolução por substituição

$C(N) = 2C(N/2) + N$

$= 2[2C(N/4) + N/2] + N$

$= 4C(N/4) + 2N$

$= 4[2C(N/8) + N/4] + 2N$

$= 8C(N/8) + 3N$

$= 2^k C(N/2^k) + kN$

Padrão do caso geral

A derivação termina em C(1) quando $N/2^k = 1$, ou seja, $N = 2^K$, $k = \lg N$

$= 2^{\lg N} C(1) + \lg N.N$

Substituindo $k = \lg N$
Sendo $C(1) = 1$

$= N + N \lg N$

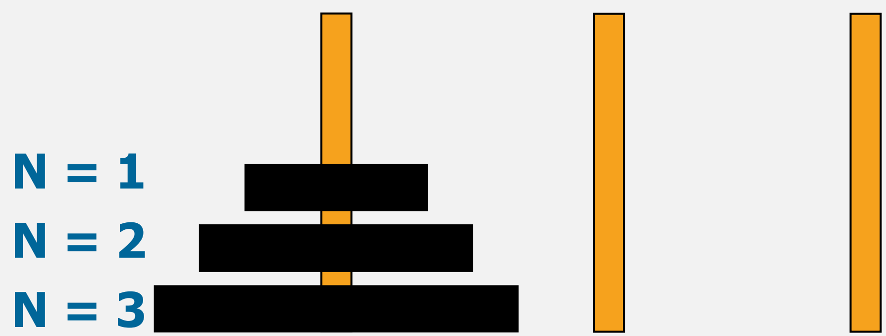
$= O(N) + O(N \lg N) = O(N \lg N)$ linear logaritmica

TORRES DE HANOI

- Algoritmo
  - N, número de discos
  - +d, movimento um pino à direita
  - -d, movimento um pino à esquerda

```
fun hanoi(n: Int, d: Int) {
    if (n == 1 ) move(n, d)
    else {
        hanoi(n - 1, -d)
        move(n, d)
        hanoi(n - 1, -d)
    }
}
```

Se n = 1 disco



ANÁLISE DAS TORRES DE HANOI

- Análise do algoritmo *Torres de Hanoi* através da sua relação de recorrência

Quando n = 1 disco, executa em tempo constante

$$\begin{cases} C(1) = O(1) \\ C(N) = C(N-1) + O(1) + C(N-1) \end{cases}$$

ANÁLISE DAS TORRES DE HANOI

- Resolução por substituição

$$C(N) = 2C(N-1) + 1$$
$$= 2[2C(N-2) + 1] + 1$$
$$= 4C(N-2) + 3$$
$$= 4\,[2C(N-3) + 1] + 3$$
$$= 8C(N-3) + 7$$
$$= 2^kC(N-k) + 2^k - 1$$

Padrão do caso geral

A derivação termina em C(1) quando N - k = 1, ou seja, k = N - 1

$$= 2^{N-1}C(1) + 2^{N-1} -1$$

Substituindo k = N - 1
Sendo C(1) = 1

$$= 2^{N-1} + 2^{N-1} -1$$
$$= 2^N - 1 \in \; O(2^N) \text{ exponencial}$$

TORRES DE HANOI

- Sendo N o número de discos, então

  - Para solucionar um Hanói de 4 discos, são necessários 15 movimentos
  - Para solucionar um Hanói de 7 discos, são necessários 127 movimentos
  - Para solucionar um Hanói de 15 discos, são necessários 32.767 movimentos
  - Para solucionar um Hanói de 64 discos, como diz a lenda, são necessários 18.446.744.073.709.551.615 movimentos.

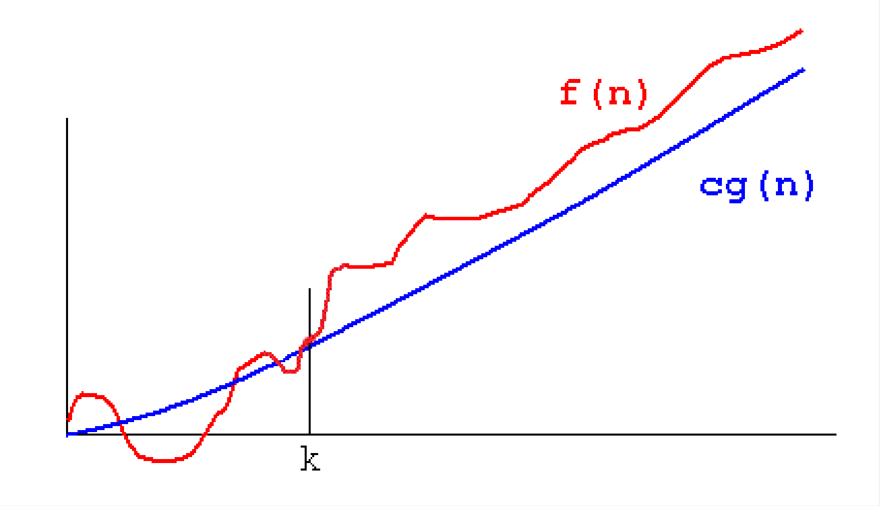
NOTAÇÃO Ω

- A notação Ω (omega) permite classificar algoritmos de acordo com os limites inferiores do seu tempo de execução.

  - Definição:

  Uma função f(n) diz-se ser Ω (g(n)), ou seja, f(n) é inferiormente limitada por g(n)), se existem duas constantes positivas c (ϵ R^+) e k (ϵ N_0) tais que

  f(n) >= c.g(n) para todo o n>= k



Logo, f = Ω(g) significa que f ∈ Ω(g), i.e., f pertence ao conjunto de funções limitadas inferiormente por g a partir de certa ordem

NOTAÇÃO Ω

- <u>Exemplo</u>
  - Provar que ½ n(n - 1) ϵ $\Omega(n^2)$

  - Prova-se o limite inferior:

    Existe c = ¼, k = 2

f(n)	>=	c . n^2
½ . 2(2 – 1) = 1	>=	¼ . 2^2 = 1
½ . 3(3 – 1) = 3	>=	¼ . 3^2 = 2.25
½ . 4(4 – 1) = 6	>=	¼ . 4^2 = 4

NOTAÇÃO Ω

- <u>Exemplo</u>
  - Provar que $f(n) = \frac{1}{2} n(n - 1) \in \Omega(n^2)$

- Prova-se o limite inferior:

  Existe $c = \frac{1}{4}$, $k = 2$

$f(n)$	$>=$	$c \cdot n^2$
$\frac{1}{2} \cdot 2(2 - 1) = 1$	$>=$	$\frac{1}{4} \cdot 2^2 = 1$
$\frac{1}{2} \cdot 3(3 - 1) = 3$	$>=$	$\frac{1}{4} \cdot 3^2 = 2.25$
$\frac{1}{2} \cdot 4(4 - 1) = 6$	$>=$	$\frac{1}{4} \cdot 4^2 = 4$

ANÁLISE DO BUBBLE SORT

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

- Bubble Sort adaptativo
  - N = right – left + 1
  - Pior caso: (N – 1) + (N – 2) + 3 + 2 + 1 = N(N – 1)/2 = $O(N^2)$
  - Melhor caso: (N – 1) = $\Omega(N)$

```
fun bubbleSortAdaptive(table: DoubleArray, left: Int, right: Int) {
    var trocas = false
    for (i in left..right) {
        for (j in right downTo i + 1) if (table[j] < table[j - 1]) {
            exchange(table, j, j - 1)
            trocas = true
        }
        trocas = if (trocas) false else break
    }
}
```

NOTAÇÃO Θ

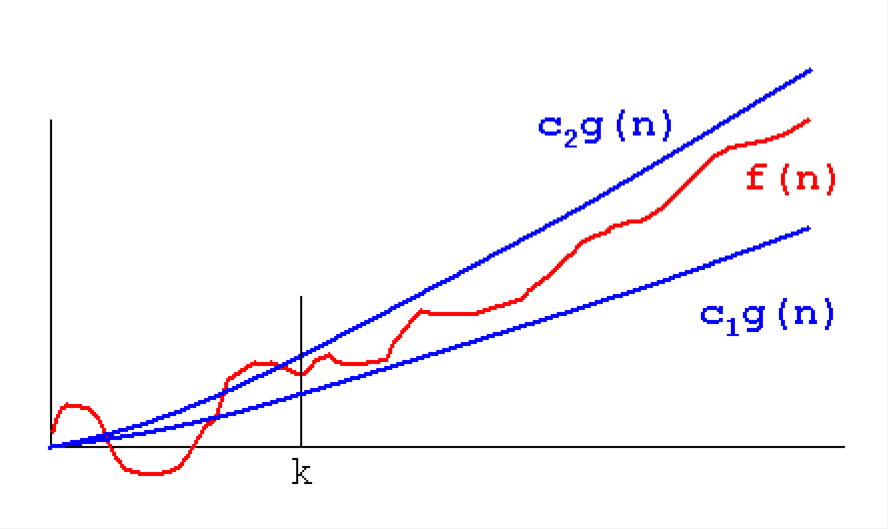
- A notação Θ (theta) permite classificar algoritmos de acordo com os limites inferiores e superiores do seu tempo de execução.

- <u>Definição</u>:

Uma função $f(n)$ diz-se ser $\theta(g(n))$, ou seja, $f(n)$ é inferiormente e superiormente limitada por $g(n)$), se existem duas constantes positivas c_1 e c_2 ($\in R^+$) e uma constante k ($\in N_0$) tais que

$$c_1.g(n) <= f(n) <= c_2.g(n)$$

para todo o $n >= k$



NOTAÇÃO Θ

- <u>Exemplo</u>
  - Provar que $f(n) = \frac{1}{2} n(n - 1) \in \Theta(n^2)$
  - Prova-se o limite superior para: $c_1 = \frac{1}{2}$, $k = 0$

$f(n)$	$<=$	$c_1 . n^2$
$\frac{1}{2} . 0(0 - 1) = 0$	$<=$	$\frac{1}{2} . 0^2 = 0$
$\frac{1}{2} . 1(1 - 1) = 0$	$<=$	$\frac{1}{2} . 1^2 = \frac{1}{2}$
$\frac{1}{2} . 2(2 - 1) = 1$	$<=$	$\frac{1}{2} . 2^2 = 2$

  - Prova-se o limite inferior para: $c_2 = \frac{1}{4}$, $k = 2$

$f(n)$	$>=$	$c_2 . n^2$
$\frac{1}{2} . 2(2 - 1) = 1$	$>=$	$\frac{1}{4} . 2^2 = 1$
$\frac{1}{2} . 3(3 - 1) = 3$	$>=$	$\frac{1}{4} . 3^2 = 2.25$
$\frac{1}{2} . 4(4 - 1) = 6$	$>=$	$\frac{1}{4} . 4^2 = 4$

NOTAÇÃO Θ

- Limites Justos
  - Se uma função C(n) tem como limites superior e inferior a mesma função f(n), ou seja
    - Se $C(n) \in \Theta(f(n))$ então f(n) é um limite justo de C(n)
  - Definição
    - $C(n) \in \Theta(f(n))$
      - sse $C(n) \in O(f(n))$
      - e $C(n) \in \Omega(f(n))$

ANÁLISE DO MERGE SORT

- Merge Sort $C(n) = O(1) + O(divide) + C\left(\frac{n}{2}\right) + C\left(\frac{n}{2}\right) + O(merge)$
  - N = right – left + 1
  - Pior caso: $O(N \lg N)$
  - Melhor caso: $\Omega(N \lg N)$

Θ(N lg N)

```
fun mergeSort(table: DoubleArray, left: Int, right: Int) {
    if (left < right) {
        val mid = (right + left)/ 2
        val tableLeft = DoubleArray(mid - left + 1)
        val tableRight = DoubleArray(right - mid)
        //divide os elementos de table por tableLeft e tableRight
        divide(table, tableLeft, tableRight, left, mid, right)

        mergeSort(tableLeft, 0, mid)                //repete para tableLeft
        mergeSort(tableRight, 0, right - mid - 1)   //repete para tableRight

        // junta ambas as metades em table, ordenando-as
        merge(table, tableLeft, tableRight, left, mid, right)
    }
}
```

NOTAÇÃO ASSINTÓTICA

- Teorema

Se

$$f_1(n) = O(g_1(n)) \text{ e } f_2(n) = O(g_2(n))$$

Então

$$f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n))$$

E

$$f_1(n).\ f_2(n) = O(g_1(n)\ .\ O(g_2(n))$$

É verdadeiro também para Ω e Θ

RESOLUÇÃO DE RECORRÊNCIAS

- Existem dois métodos de resolução de recorrências:
  - Método de Substituição
    - Chega-se a uma possível solução recorrendo à técnica de substituições
  - Teorema Mestre
    - Encontra facilmente a solução em recorrências na forma

      $T(n) = aT(n/b) + f(n)$

TEOREMA MESTRE

- A eficiência no tempo $T(n)$ em muitos algoritmos "dividir para conquistar", satisfaz a condição

$$T(n) = aT(n/b) + f(n)$$

$$a \geq 1, \quad b > 1$$

- Pode ser usado o *Master Theorem* (Teorema Mestre) para resolver este tipo de recorrências

TEOREMA MESTRE

Caso 1:

$$f(n) = O(n^{\log_b a-\varepsilon})$$

Se existir uma constante ε>0
f(n) <= c.g(n) para todo o n>= k

Solução: $T(n) = \theta(n^{\log_b a})$

Caso 2:

$$f(n) = \theta(n^{\log_b a})$$

Se existir c_1.g(n) <= f(n) <= c_2.g(n)
para todo o n>= k

Solução: $T(n) = \theta(n^{\log_b a} \lg n)$

Caso 3:

$$f(n) = \Omega(n^{\log_b a+\varepsilon})$$

Se existir uma constante ε>0 e
f(n) >= c.g(n) para todo o n>= k

Solução: $T(n) = \theta(f(n))$

a.f(n/b) ≤ c.f(n)

TEOREMA MESTRE

- Caso 1 – Exemplo

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

$$a = 8, b = 2, f(n) = 1000n^2$$

ε>0

f(n) = O (n$^{3-ε}$)

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

Solução: $T(n) = \Theta\left(n^{\log_b a}\right)$

$$T(n) = \Theta(\mathrm{n}^3)$$

TEOREMA MESTRE

- Caso 2 – Exemplo

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

$$a = 2, b = 2, f(n) = 10n$$

$$n^{log_b a} = n^{log_2 2} = n^1 = n$$

=

f(n) = O(n)

Solução: $T(n) = \Theta(n^{log_b a} \lg n)$

$$T(n) = \Theta(n \lg n)$$

TEOREMA MESTRE

- Caso 3 – Exemplo

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathrm{n}^2$$

$$a = 2, b = 2, f(n) = n^2$$

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

ε>0

f(n) = O(n$^{1+ε}$)

Solução: $T(n) = \Theta(f(n))$

$$T(n) = \Theta(\mathrm{n}^2)$$