iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

iscte
TECNOLOGIAS
E ARQUITETURA

# Desenho e Análise de Algoritmos

## Notas - Aula TP 3 e 4
Análise de complexidade de algotitmos

**Ana Maria de Almeida**
DCTI
2022/2023

1

*"A person well-trained in computer science knows how to deal with algorithms: how to **construct** them, **manipulate** them, **understand** them, **analyze** them. This knowledge is preparation for much more than writing good computer programs; it is **a general-purpose mental tool** that will be a definite aid to the understanding of other subjects, whether they be chemistry, linguistics, or music, etc. The reason for this may be understood in the following way: It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not really understand something until after teaching it to a computer, i.e., expressing it as an algorithm . . . An **attempt to formalize things as algorithms leads to a much deeper understanding than if we simply try to comprehend things in the traditional way**."*

- **Donald Knuth**, "*Selected Papers on Computer Science*"

2

# Plano para esta aula

- Aumento de input e desempenho esperado
  – Fenómeno de Escala
- Introdução à Análise de Algoritmos:
  - análise de complexidade
  - limites assintóticos
  - análise no pior dos casos
  - estratégia *dividir-e-conquistar* recursiva



# Nas últimas aulas

- Revisão das noções de algoritmo e desempenho computacional
- Relembrar como podemos perceber o impacto do algoritmo em implementações (programas)
- e como esse impacto pode ser medido.
- Apresentação da estratégia geral “dividir e conquistar” e a mesma estratégia do ponto de vista recursivo.

- Estudámos algoritmos para efetuar o produto de inteiros, que só utilizavam a operação nativa para multiplicação em números com um único algarismo. Nomeadamente:
  - algoritmo da multiplicação do 1.º ciclo do Ensino Básico – $O(n^2)$
  - algoritmo recursivo do tipo “dividir e conquistar” usando 4 chamadas recursivas para n > 1 – $O(n^2)$
  - algoritmo de Karatsuba, também recursivo, também usa a estratégia “dividir e consquistar” – $O(n^{1,6})$

# Algoritmos de multiplicação: Podemos melhorar?

- **Toom-Cook** (1963): partição em 5 subproblemas com n/3 algarismos - $O(n^{1.465})$
- **Schönhage–Strassen** (1971): recorrendo a transformação polinómiais e usando multiplicação polinomial rápida (Fast Fourier Transform) - $O(n \log(n) \log \log(n))$
- **Furer** (2007): usando polinómios em $\mathbb{C}$- $O\left(n \log(n) \cdot 2^{O(\log^*(n))}\right)$
- **Harvey and van der Hoeven** (2019): usando reticulados vetoriais podemos multiplicar em $O(n \log(n))$
- **AlphaTensor** 2022: "AlphaTensor discovers" faster algorithms to multiply matrices" https://www.nature.com/articles/d41586-022-03023-w

- Para ler:
- https://en.wikipedia.org/wiki/Multiplication_algorithm#cite_note-22
- https://www.quantamagazine.org/mathematicians-discover-the-perfect-way-to-multiply-20190411/



# Dividir e Conquistar

- A **estratégia geral** dividir-e-conquistar resolve um problema em 3 passos:
  1. **Dividir** o problema em subproblemas .
  2. **Resolver** os subproblemas.
  3. **Conquistar** combinando as soluções dos subproblemas na solução do problema original.

- **Recursivamente**, esta estratégia recorre a 3 passos em cada nível recorrente:
  1. **Dividir** o problema em **k** subproblemas **iguais mas de menor dimensão**.
  2. **Enquanto não atingir** um **caso base,** resolver cada subproblema através da aplicação recursiva da divisão.
  3. **Combinar** as k soluções dos subproblemas na solução do problema da chamada anterior.

2022/2023

## Fibonacci

- **Sucessão de Fibonacci**: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- ou: $F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{se } n > 1 \end{cases}$

**Pseudo-código**

Em DAA, os algoritmos serão sempre apresentados em **pseudo-código** mantendo a independência da linguagem de programação.

**Algoritmo**

```
function fiboR(n):
   //Calcula o valor do n-ésimo número de Fibonacci
   //Input: inteiro não negativo, n
   //Output: valor do n-ésimo número de Fibonacci

   if n = 0 : return 0
   if n = 1 : return 1
   return fiboR(n-1) + fiboR(n-2)
```

**Solução Recursiva dividir para conquistar**

2022/2023



## Algoritmo: 3 questões essenciais

Algoritmo novo-> questionar sempre:

1. Está **correto**?
2. **Quanto tempo demora** a resolver em função do (tamanho) do input?
3. ***Que podemos fazer para melhorar***?

O melhor tempo(n) que se pode ter é constante: mas são raros os casos (ou muito simples). log(n) é melhor que n. No geral andamos nas ordens do: n log(n); e polinomial: n^k, sendo k>1.1, etc

2022/2023

Número de iterações num ciclo: (ultimo índice - primeiro índice) + 1 -> nº de operações é diferente da Ordem
Por exemplo com Fibonacci: (n-2) + 1



## A sucessão de Fibonacci - versão 2

$$F_n = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F_{n-1} + F_{n-2} & \text{se } n > 1 \end{cases}$$

*Medir o tempo em função da dimensão dos dados*

- $T(n) \leq 2$ se $n \leq 1$
- $T(n) = n - 1$ se $n > 1$
- Este algoritmo leva tempo linear no tamanho do input

**Algoritmo Fibonacci 2**

```
function fiboP(n):
    //Calcula o valor do n-ésimo número de Fibonacci
    //Input: inteiro não negativo, n
    //Output: valor do n-ésimo número de Fibonacci

    if n=0: return0
    if n=1: return1
    criar um array: f[0..n]
    f[0] ← 0,   f[1] ← 1
    for i=2,..., n:
        f[i] ← f[i-1] + f[i-2]
    return f[n];
```

**Programação Dinâmica**
(guardar cálculos de subproblemas)

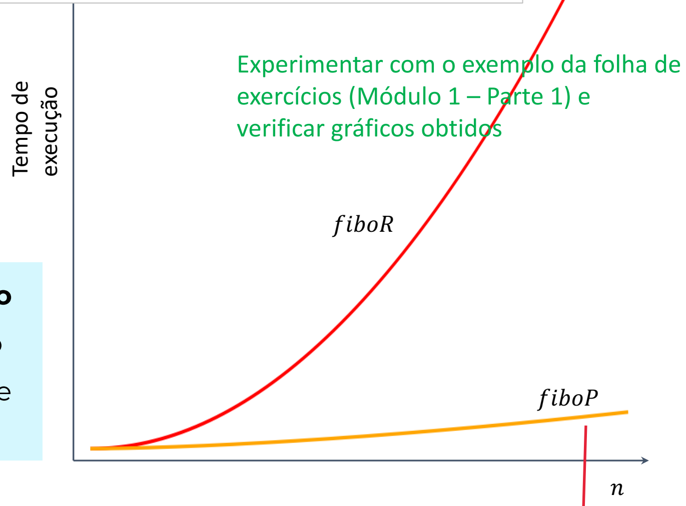


## Quão rápida é fiboP?

- O crescimento do tempo de execução de cada implementação será algo como:

- Mas como podemos medir o tempo para QUALQUER instanciação possível de input?

Resposta: fazendo a **análise do algoritmo** para caracterização do desempenho, qualquer que seja o input.




Experimentar com o exemplo da folha de exercícios (Módulo 1 – Parte 1) e verificar gráficos obtidos



valores já calculados são guardados podendo assim ser reutilizados, evitando repetir cálculos

iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

## Análise de Algoritmos é independente do hardware/software

- Caracteriza o tempo de execução de um algoritmo em função do input (de tamanho) *n*
- É uma análise concreta do tipo: "*Para qualquer input de tamanho n, o algoritmo executa, no máximo,* $1.62n^2 + 3.5n + 8$ *passos*"
- **Desvantagens**:
  - Um limite de eficiência tão preciso é uma atividade pesada e traz mais detalhe que aquele que é realmente necessário.

Necessitamos de classificar algoritmos numa **classe**/categoria, cuja diferença entre os elementos está apenas nas constantes envolvendo termos da mesma categoria (ou de menor grau).

  - E muito detalhe (baixo nível), na maior parte das vezes, nem sequer tem significado

O modo como cada passo elementar é implementado em passos básicos e daí em instruções máquina varia

A noção de passo, em cada nível de abstração, varia num factor constante



## Análise assimptótica

Quando estamos a definir o algoritmo há dois passos:

***Passo elementar***:
*Consiste em ações elementares (instruções) num (pseudo) código de mais alto nível, como por exemplo:*
- *uma comparação*
- *uma operação aritmética*
- ***…***

Note que **cada passo elementar** pode consistir **em um ou mais passos básicos**

***Passo básico*** *(ou* ***primitiva****)*:
*Consiste em ações (instruções) num (pseudo) código de baixo nível, como as de:*
- *atribuição de valor a uma variável*
- *acesso a uma entrada num array*
- *usar um ponteiro para seguir para um dado elemento*
- *uma operação aritmética com um inteiro de tamanho fixo*
- ***…***

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

## Ordem de Complexidade – Análise de Algoritmos

Estuda como o algoritmo escala com o aumento do n

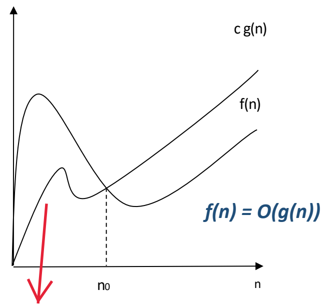
- **Análise de complexidade de um algoritmo:** concentra-se na **eficiência assintótica** (análise da taxa de crescimento quando $n \to \infty$)

**Notação *big-$\mathcal{O}$* (*O grande*):**

Sejam $f: \mathbb{N} \to \mathbb{R}$ e g: $\mathbb{N} \to \mathbb{R}$ duas funções. Diz-se que $f(n) = \mathcal{O}(g(n))$

(e lê-se *"f é da ordem de g"*) sse

$$\exists c \in \mathbb{R}^+, \exists n_0 \in \mathbb{N} : \forall n \geq n_0 |f(n)| \leq c\,|g(n)|$$




Para n's mais pequenos algumas funções podem ter comportamentos diferentes, por isso especifica-se a partir de que n é que válido

*Entende-se como "f não cresce mais rapidamente do que g" ou "o crescimento de f está limitado pelo de g"*

A notação *big-O* indica qual a tendencia de aumento de gasto de quantidade de recurso com o aumento da quantidade de dados.





iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

## *Definições*

No que se segue, define-se:

- **Problema**:
  - uma especificação de dados de entrada - input
  - uma especificação de resultados de saída - output (definidos em função do input)

- **Algoritmo:** uma solução para um **problema** constituída por:
  - instruções elementares, ordenadas, bem definidas e não ambíguas
  - capaz de produzir o output especificado para cada um dos potenciais inputs.

# Ordenação

- Necessidade de ordenar listas de símbolos (números, caracteres de um alfabeto ou mesmo strings) ou de registos (de alunos, de bibliotecas, de funcionários, etc.)
- Necessidade de tornar certas operações mais eficientes (rápidas), como as de: pesquisa(em dicionários ou mapeamentos, contactos telefónicos ou de mail, etc. ), compressão de dados, etc.
- Resposta ordenada (como a da *Google search* que exige um ordenamento (*ranking*) das pesquisas na Internet pela sua relevância de resposta)
- No caso de ordenação de registos, usa-se uma **chave** de ordenação (um atributo no objeto)

16

# Exercício – O problema da ordenação

- **Problema genérico da ordenação** (por ordem crescente):
  - *Dados*: uma sequência de objetos de valor ordenável: $a_1, a_2, \ldots, a_n$
  - *Resultados*: a permutação da sequência que os ordena por valor: $a_1 \leq a_2 \leq \ldots \leq a_n$

**Algoritmo ordena:**
//Input: dada uma sequência com n elementos ordenáveis
//Output: a sequência ordenada por ordem crescente

- inicializar "posição" à primeira da sequência
- enquanto a "posição" não esgotar a sequência:
  - procurar o menor dos elementos de "posição" para a frente
  - se os elementos são diferentes, então trocar entre as duas posições
  - avançar para a próxima "posição" na sequência

**Algoritmo descrito em linguagem corrente**

Descreve o algoritmo, mas deixa vários detalhes de implementação livres

# Exercício – O problema da ordenação

**Método da Seleção Linear (Selection Sort)**

**Algoritmo:**
//Input: dada uma sequência com n elementos ordenáveis
//Output: a sequência ordenada por ordem crescente
- inicializar "posição" à primeira da sequência
- enquanto a "posição" não esgotar a sequência:
  - procurar o menor dos elementos de "posição" para a frente
  - se os elementos são diferentes, então trocar entre as duas posições
  - avançar para a próxima "posição" na sequência

- Está correto?
  - em cada volta do ciclo é "inserido" um elemento numa lista ordenada e na sua posição final;
  - após, no máximo, n – 1 "inserções" (trocas), a lista contém os n elementos na ordem desejada
- Quanto "tempo" demora a ordenar?
  - Consoante a permutação inicial dos elementos na sequência, haverá maior ou menor peso computacional.

Depende de como o input que vamos tratar está. Por exemplo: já ordenada? na sequência inversa? Meio ordenada e meio desordenada?





# Exercício – cálculo analítico para a eficiência da seleção linear

```
Selectionsort(a, n):
  //Ordena uma sequência por ordem crescente
  //Input: sequência a ordenar  de tamanho n
  //Output: sequência ordenada

  for i = 1, …, n:
    minimo = i
    for j = i+1, …, n:
      if a[j] < a[minimo]
         minimo = j
    endfor
    if i ≠ minimo:
       troca(a[i], a[minimo])
  endfor
```

**Pseudo-código** para o Método da Seleção Linear

- **Se for dada uma sequência já ordenada?**

  o melhor que pode acontecer

- **E se estiver ordenada em ordem inversa da pedida?**
- **E se for "meio" ordenada e "meio" desordenada?**

# Análise do número de passos da Selecção Linear – versão 1

- O *for* exterior repete ***n – 1*** vezes e, em cada volta i há:
  - ----------------- do índice i a minimo
  - o ciclo interior corre **n – i** vezes:

    e, em cada volta j, há ------------- e haverá -------- ----------
  - para cada troca há ----------------

para cada troca há 3pk atribuições (com pa a indicar a probabilidade de a[k] != a[minimo])

E ainda existem as atribuições dos ciclos for a i e a j como iteradores dos ciclos.

```
Selectionsort(a, n):
  //Ordena uma sequência por ordem crescente
  //Input: sequência a ordenar de tamanho n
  //Output: sequência ordenada

  for i = 1, ..., n:
      minimo = i
    for j = i+1, ..., n:
      if a[j] < a[minimo]
          minimo = j
    endfor
    if i ≠ minimo:
      troca(a[i], a[minimo])
  endfor
```





# Análise do número de passos da Selecção Linear – versão 1

```
Selectionsort(a, n):
  //Ordena uma sequência por ordem crescente
  //Input: sequência a ordenar de tamanho n
  //Output: sequência ordenada

  for i = 1, ..., n:
    minimo = i
    for j = i+1, ..., n:
      if a[j] < a[minimo]
        minimo = j
    endfor
    if i ≠ minimo:
      troca(a[i], a[minimo])
  endfor
```

- No total, existem **n** objetos para comparar, logo, em cada ciclo do for exterior, se 2 elementos estiverem desordenados, fazem-se **1+pj+3pi** atribuições (assumimos que uma troca equivale a 3 atribuições e sem contar com as atribuições operacionais dos for ao contador i e ao j)
- Usando **K** (tempo de atribuição) e **C** (tempo de comparação) para denotar as constantes computacionais dependentes da máquina, o total é:

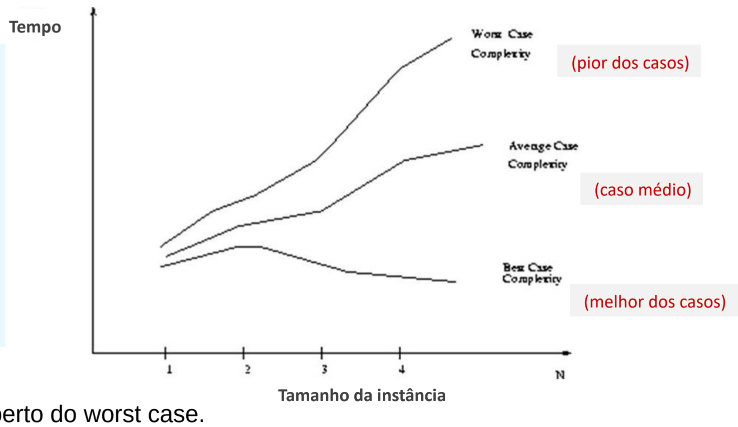
$$\sum_{i=1}^{n} K+(n-i)C+p_jK+3p_iK = \sum_{i=1}^{n}(n-i)C+(1+p_j+3p_i)K = \frac{n^2-n}{2}C+(1+p_j+3p_i)(n-1)K$$

iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

# Análise Assimptótica de um algoritmo

Medir (calcular) a eficiência computacional:

- no **melhor caso** (*best case*)
- no **caso médio** (*average case*)
- no **pior caso** (*worst case*)




Tipicamente o average case está mais perto do worst case.

2022/2023



# Análise da Selecção Linear - casos

Fórmula da análise geral:

$$\frac{n^2-n}{2}C+(1+p_j+3p_i)(n-1)K$$

- Melhor dos casos (*Best case*)– ***B(n)***:

  a sequência já está ordenada e não vão efetuadas trocas, ou seja, pi = pj = 0

$$B(n) = \frac{n^2-n}{2}C + (n-1)K$$

- Caso médio (*Average case*) – ***A(n)***:

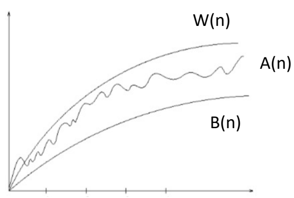
  se nada é dito, a distribuição escolhida é a uniforme, ou seja, pi = pj = 1/2

$$A(n) = C\frac{n^2-n}{2} + 3(n-1)K$$

- Pior dos casos (Worst case) – ***W(n)***:

  então, $p_i$ e $p_j$ acontecem sempre, i.e, pi = pj = 1

$$W(n) = C\frac{n^2-n}{2} + 6(n-1)K$$




2022/2023

Para relembrar fórmulas de cálculo usuais em Ciências da Computação
ver em Conteúdos > Materiais de auxílio ao estudo no Moodle

# O problema da ordenação: **Seleção linear**

- **Problema genérico da ordenação** (por ordem crescente):
  - Dados: uma sequência de objetos de valor ordenável: $a_1, a_2, \ldots, a_n$
  - Resultados: a permutação da sequência que os ordena por valor: $a_1 \leq a_2 \leq \ldots \leq a_n$

O **método da Seleção Linear**:

É O(n^2) em comparações, mas apenas O(n) em movimentos (atribuições para troca de elementos)

Falta mostrar que isto é verdade → exercício!

```
Selectionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

    for i = 1, …, n:
        minimo = i
       for j = i+1, …, n:
           if a[j] < a[minimo]
               minimo = j
       endfor
       if i ≠ minimo:
           troca(a[i], a[minimo])
   endfor
```

Seleção linear (Selection sort)



# Ordem de crescimento – factor de escala

- A diferença em tempos de execução **em inputs pequenos não permite distinguir** um algoritmo eficiente de um que não o é.
- Quando experimentamos com inputs (números, coleções, etc) grandes, é possível **começar a estimar** a eficiência (ou a falta dela).
- Para valores de input com dimensão n (realmente) muito grande, é a ordem de crescimento em função do n que realmente conta.

# Ordem de crescimento - valores

**+ Rapidez de desempenho -**

| $n$ | $\log_2 n$ | $n$ | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ | $n!$ |
|---|---|---|---|---|---|---|---|
| 10 | 3.3 | $10^1$ | $3.3 \cdot 10^1$ | $10^2$ | $10^3$ | $10^3$ | $3.6 \cdot 10^6$ |
| $10^2$ | 6.6 | $10^2$ | $6.6 \cdot 10^2$ | $10^4$ | $10^6$ | $1.3 \cdot 10^{30}$ | $9.3 \cdot 10^{157}$ |
| $10^3$ | 10 | $10^3$ | $1.0 \cdot 10^4$ | $10^6$ | $10^9$ | brutally large numbers!!! | |
| $10^4$ | 13 | $10^4$ | $1.3 \cdot 10^5$ | $10^8$ | $10^{12}$ | | |
| $10^5$ | 17 | $10^5$ | $1.7 \cdot 10^6$ | $10^{10}$ | $10^{15}$ | | |
| $10^6$ | 20 | $10^6$ | $2.0 \cdot 10^7$ | $10^{12}$ | $10^{18}$ | | |

Tabela de valores (alguns por aproximação) de crescimento com o aumento do valor de n

- Um sistema capaz de executar um bilião ($10^{12}$) de operações por segundo, levaria $4 \times 10^{10}$ anos a executar $2^{100}$ operations.
- Estima-se que o planeta Terra exista há cerca de 4.5 mil milhões de anos ($4,5 \times 10^9$) ...



# A notação O- grande (*Big-O*)

- A notação Big-O permite uma compreensão geral sobre a eficiência do algoritmo
- **Foco** em como cresce o tempo de execução (quantidade de operações) com o aumento de escala do (tamanho do) input - n

| Número de operações | Tempo assintótico de execução |
|---|---|
| $\frac{1}{10} \cdot n^2 + 100$ | $O(n^2)$ |
| $0.063 \cdot n^2 - .5\,n + 12.7$ | $O(n^2)$ |
| $100\, n^{1.5} - 10^{10000}\sqrt{n}$ | $O(n^{1.5})$ |
| $11\, n\log(n) + 1$ | $O(n\log(n))$ |

importa o factor com maior taxa de crescimento

este algoritmo é "assintoticamente mais rápido" que os restantes.

## Recapitulando os pontos principais:

- A eficiência temporal de um algoritmo é **medida** em termos de uma **função no tamanho do input**
- Para isso, devemos contar o número de vezes que o algoritmo executa operações elementares.
- As eficiências de diferentes algoritmos para o mesmo efeito podem diferir bastante para os mesmos inputs.
- Para perceber como escala um algoritmo, devemos estudar o que acontece no pior dos casos (worst-case), no caso médio (average-case) e no melhor dos casos (best-case).
- A análise de eficiência de um algoritmo é efetuada em termos da ordem de crescimento do "tempo de execução" do algoritmo quando o tamanho n do input aumenta (tende para o infinito)
- O estudo empírico permite perceber o desempenho em alguns casos mas não estabelece a verdadeira eficiência do algoritmo



## Limites assintóticos - exemplos

- O crescimento de $5n^2 + 2n + 20$ é dominado pelo termo em $n^2$ quando $n \to \infty$, logo, é $O(n^2)$

$$\exists c > 0, n_0 > 0, \forall n \geq n_0, f(n) \leq c\,g(n)$$
$$\Updownarrow$$
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

- De facto, a definição de "f é da ordem de g" significa que, no limite, f e g apenas diferem de uma constante: ***c***
- Ou seja, a notação indica um limite superior (= por excesso) :

  a ordem de crescimento de f não ultrapassa a de g

- No exemplo, temos:

$$\frac{5n^2 + 2n - 20}{n^2} = 5 + \frac{2}{n} - \frac{20}{n^2} \to 5$$

e

$$5n^2 + 2n - 20 = O(n^2)$$

Mas também temos:

$$\frac{n^2}{5n^2 + 2n - 20} = \frac{1}{5 + \frac{2}{n} - \frac{20}{n^2}} \to \frac{1}{5}$$

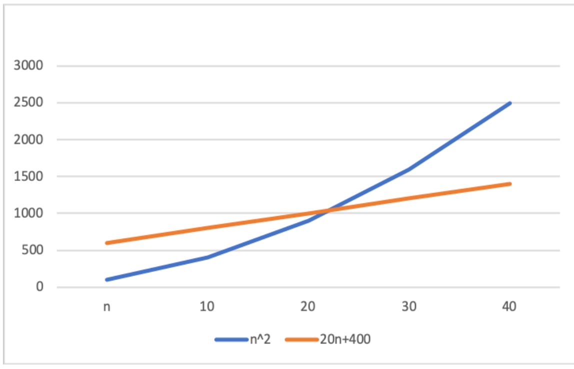
logo, $n^2 = O(5n^2 + 2n - 20)$

*"Ordem de"* determina *um conjunto de funções*

iscte
INSTITUTO UNIVERSITÁRIO DE LISBOA

# Limites assintóticos

- E quanto a $f_1 = n^2$ e $f_2 = 10n + 400$?
  - *A partir de $n_0 = 26$, $10n + 400$ é dominada por $n^2$, logo, $f_2 = O(f_1) = O(n^2)$*
  - *Mas, $f_1$ não é $O(10n + 400)$ $[= O(n)]$*




- E que dizer entre $f_2$ e $f_3 = n - 1$?
  - são ambas da mesma ordem
  - são ambas da ordem de n: O(n)
  - *são* ______________________

30

## Limite superior

- Diz-se que *f* é ***superiormente limitado da ordem de g*** e escreve-se $\boldsymbol{f(n) = O(g(n))}$ se e só se $\exists c > 0, n_0 > 0, \forall n \geq n_0, f(n) \leq c\, g(n)$

**Exemplos:**

- ordem constante
- ordem linear
- ordem polinomial
- ordem cúbica
- ordem quadrática
- ordem polilogarítmica

36

## Limite inferior

- Diz-se que ***f* é *inferiormente limitado da ordem de g*** e escreve-se $\boldsymbol{f(n) = \Omega(g(n))}$ se e só se $\exists c > 0, n_0 > 0, \forall n \geq n_0, f(n) \geq c\, g(n)$

Relembre que

$$f(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c > 0, n_0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$f(n) \leq cg(n)$$

**da ordem superior**

**da ordem inferior**



## Limite exato

- Diz-se que ***f* é *exatamente da ordem de g*** e escreve-se $\boldsymbol{f(n) = \Theta(g(n))}$ se e só se $f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$

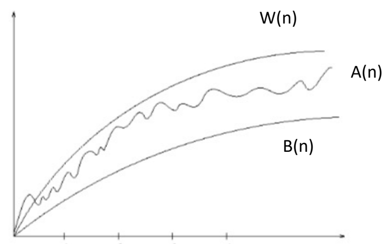
**da ordem superior**

**da ordem inferior**

**da ordem exata**

## A análise do pior dos casos (worst-case analysis)

- Garante que nada pior irá acontecer (pois estuda o pior caso possível), é um **limite superior** para qualquer tamanho de input
- Para alguns algoritmos o pior caso ocorre com frequência (exemplo: pesquisa de presença/pertença de elemento)
- Em geral, o **caso médio aproxima-se muito do pior dos casos**, ou seja, tende a comportar-se ligeiramente melhor mas é da mesma ordem em termos de limite superior.






# Algoritmos de ordenação

Recordar os ordenamentos (AED) e acrescentar informação

# Ordenação: aplicações

- Facilita a pesquisa: se a sequência estiver ordenada, a pesquisa é O(log n)
- Verificação do elemento mais semelhante – O(n)
- Que elemento é mais frequente (frequência ou moda) - O(n)
- Máximo, mínimo, mediana – O(1)
- Estabelecimento de um código de Huffman ([https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman](https://pt.wikipedia.org/wiki/Codifica%C3%A7%C3%A3o_de_Huffman))
- . . .



# Ordenação: factos

- Existem vários algoritmos, e **alguns deles** capazes de ordenar um array de tamanho n na ordem de **n log n** (polilogarítmica)
- Está provado este limite é o menor possível para um algoritmo que efetue comparações de chaves (i.e, nenhum algoritmo que faça comparações pode ser muito melhor que *O(n log n)*)

- Uns algoritmos são mais adequados para uma dada situação (ou tipo de dados) que outros
- Uns são relativamente simples mas lentos e outros mais complicados mas mais rápidos
- Uns trabalham melhor com inputs aleatórios e outros com listas quase ordenadas
- Uns podem residir em memória de processamento e outros são mais adaptados para ordenar ficheiros grandes armazenados em discos

# O problema da ordenação: **Seleção linear**

- **Problema genérico da ordenação** (por ordem crescente):
  - Dados: uma sequência de objetos de valor ordenável: $a_1, a_2, \ldots, a_n$
  - Resultados: a permutação da sequência que os ordena por valor: $a_1 \leq a_2 \leq \ldots \leq a_n$

```
Selectionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

    for i = 1, …, n:
        minimo = i
        for j = i+1, …, n:
             if a[j] < a[minimo]
                 minimo = j
        endfor
        if i ≠ minimo:
            troca(a[i], a[minimo])
   endfor
```

Seleção linear (Selection sort)

Atrás relembrámos o **método da Seleção Linear**:

**$O(n^2)$** em comparações

**$O(n)$** em movimentos (atribuições para troca de elementos)



# O problema da ordenação: **Inserção Linear**

```
Insertionsort(a, n):
    //Ordena uma sequência por ordem crescente
    //Input: sequência a ordenar e inteiro não negativo, n
    //Output: sequência ordenada

   for j = 2, …, n:
        este = a[j]
        i = j-1
       while i > 0 and a[i] > este:
           a[i+1] = a[i]
              i = i-1
       endwhile
       a[i+1] =  este
   endfor
```

**Inserção linear (Insertion sort)**

# Correção de um algoritmo

- Basta estar correto num input aleatório?
- Basta estar correcto num número (mais ou menos grande) de inputs (testes experimentais)?

- Temos de analisar o algoritmo sobre o ponto de vista do **pior dos casos**:
  - Um algoritmo tem de estar correto **para todos os inputs**.
  - Temos de avaliar qual será o tempo de execução para o **pior que pode acontecer**.

**worst-case analysis**



# Análise do Pior dos Casos

- Necessária para termos uma garantia (prova) que o nosso algoritmo vai sempre devolver uma resposta correta para qualquer input.
- Pode ser desafiante
- Mas é importante como medida de segurança *foolprof*.

**Como ?**

- Na maioria dos casos, podemos usar o método de indução, que nos permite generalizar um passo/conjunto de passos chave na construção do nosso algoritmo para qualquer input.
- Vamos ilustrar um modo de analisar um algoritmo e detetar o tal "passo chave" usando o método de Inserção Linear.

Source: Mary Wooters

# Generalizar

- Detetar uma propriedade (invariante) de construção da resposta final do algoritmo.
- Formular essa propriedade/invariante matematicamente.
- Usar uma técnica de prova construtiva para mostrar que, para qualquer input, a construção funciona corretamente.

No Moodle, vai surgir uma prova formal de como mostrar que a Inserção Linear está correta para qualquer input.



# Para estudar

- ***Introduction to Algorithms*. Cormen, Leiserson et al., 4th ed. MIT Press, 2022 - Cap. 1. e cap. 2 – secções 2.1 e 2.2**