# Análise de complexidade

## Introdução

➔ <u>Algoritmo</u>: sequência de instruções necessárias para a resolução de um problema bem formulado (passíveis de implementação em computador)

➔ Estratégia:

- especificar (definir propriedades)
- arquitectura (algoritmo e estruturas de dados)
- análise de complexidade (tempo de execução e memória)
- implementar (numa linguagem de programação)
- testar (submeter entradas e verificar observância das propriedades especificadas)

# Análise de complexidade

## Análise de algoritmos

- ➔ Provar que um algoritmo está correcto
- ➔ Determinar recursos exigidos por um algoritmo (tempo, espaço, etc.)
  - — comparar os recursos exigidos por diferentes algoritmos que resolvem o mesmo problema (um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema)
  - — prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresce

# Análise de complexidade

## Análise de algoritmos

- Que dados usar ?
  - dados reais: verdadeira medida do custo de execução
  - dados aleatórios: assegura-nos que as experiências testam o algoritmo e não apenas os dados específicos
    - Caso médio
  - dados perversos: mostram que o algoritmo funciona com qualquer tipo de dados
    - Pior caso!
  - dados benéficos:
    - Melhor caso

# Análise de complexidade

## Complexidade espacial e temporal

- ➔ Complexidade espacial de um programa ou algoritmo: espaço de memória que necessita para executar até ao fim

  ***S(n)*** : espaço de memória exigido em função do tamanho ***n*** da entrada

- ➔ Complexidade temporal de um programa ou algoritmo: tempo que demora a executar (tempo de execução)

  ***T(n)*** : tempo de execução em função do tamanho ***n*** da entrada

- ➔ Complexidade ↑ vs. Eficiência ↓
- ➔ Por vezes estima-se a complexidade para o "melhor caso" (pouco útil), o "pior caso" (mais útil) e o "caso médio" (igualmente útil)

# Análise de complexidade

## Complexidade temporal

➔ Análise precisa é uma tarefa complicada

- algoritmo é implementado numa dada linguagem
- a linguagem é compilada e o programa é executado num dado computador
- difícil prever tempos de execução de cada instrução e antever optimizações
- muitos algoritmos são “sensíveis” aos dados de entrada
- muitos algoritmos não são bem compreendidos

➔ Para prever o tempo de execução de um programa

- apenas é necessário um pequeno conjunto de ferramentas matemáticas

# Análise de complexidade

## Crescimento de funções

➔ O tempo de execução geralmente dependente de um único parâmetro ***n***

- ordem de um polinómio
- tamanho de um ficheiro a ser processado, ordenado, etc.
- ou medida abstracta do tamanho do problema a considerar (usualmente relacionado com o número de dados a processar)

➔ Quando há mais do que um parâmetro

- procura-se exprimir todos os parâmetros em função de um só
- faz-se uma análise em separado para cada parâmetro

# Análise de complexidade

## Ordens de complexidade mais comuns

➔ Os Algoritmos têm tempo de execução proporcional a

- ***1*** : muitas instruções são executadas uma só vez ou poucas vezes (se isto acontecer para todo o programa diz-se que o seu tempo de execução é constante)
- ***log n :*** tempo de execução é logarítmico (cresce ligeiramente à medida que ***n*** cresce; quando ***n*** duplica ***log n*** aumenta mas muito pouco; apenas duplica quando ***n*** aumenta para $n^2$)
- ***n :*** tempo de execução é linear (típico quando algum processamento é feito para cada dado de entrada; situação óptima quando é necessário processar ***n*** dados de entrada, ou produzir ***n*** dados na saída)

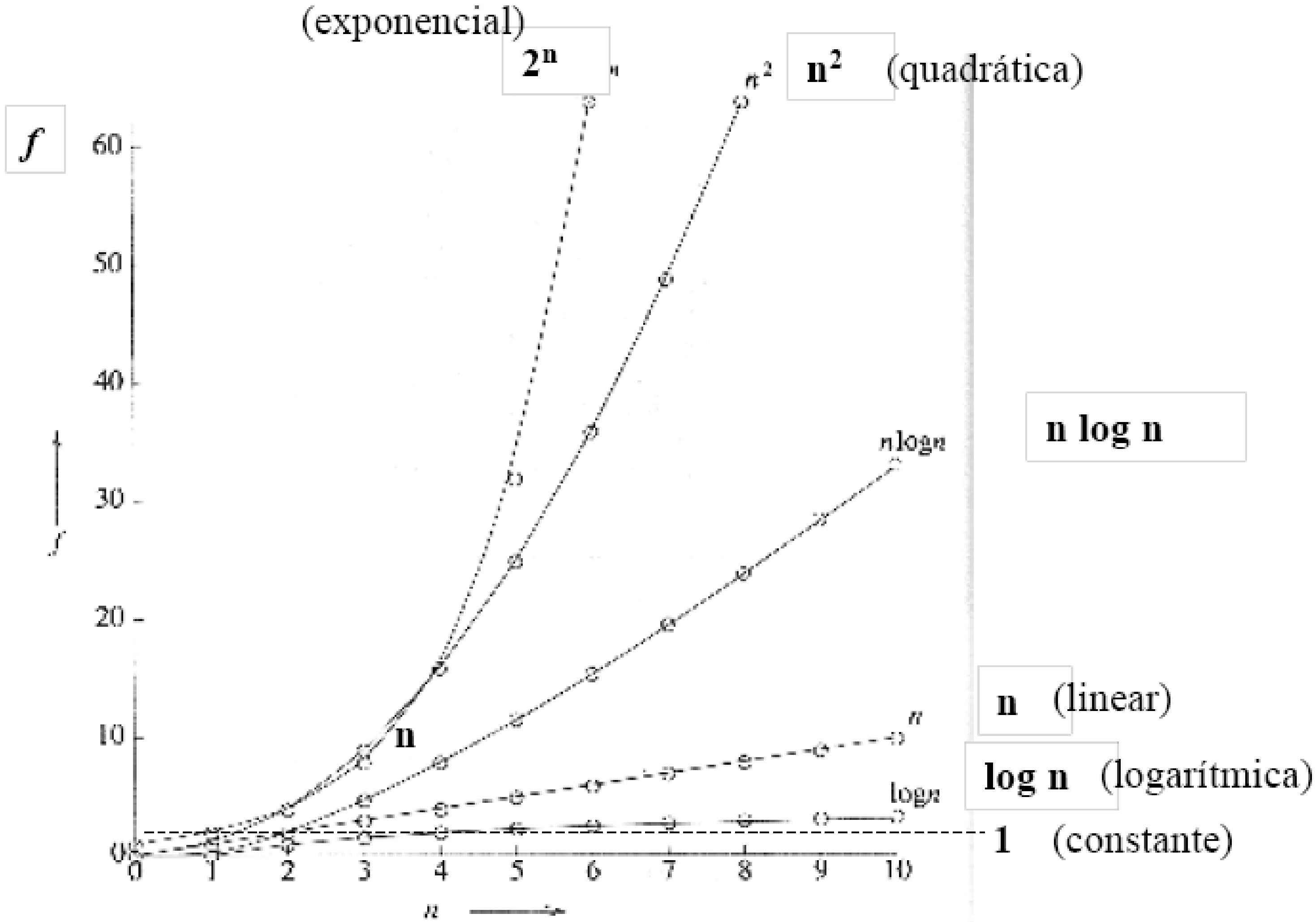
# Análise de complexidade

## Ordens de complexidade mais comuns

- ***n log n*** : típico quando se reduz um problema em subproblemas, se resolve estes separadamente e se combinam as soluções (se ***n*** é igual a 1 milhão, ***n log n*** é perto de 20 milhões)
- $n^2$ **:** tempo de execução quadrático (típico quando é necessário processar todos os pares de dados de entrada; prático apenas em pequenos problemas, ex: produto de vectores)
- $n^3$ **:** tempo de execução cúbico (para ***n*** = 100, $n^3$ = 1 milhão, ex: produto de matrizes)
- $2^n$ **:** tempo de execução exponencial (provavelmente de pouca aplicação prática; típico em soluções de força bruta; para ***n*** = 20, $2^n$ = 1 milhão; se ***n*** duplica, o tempo passa a ser o quadrado)

# Análise de complexidade

## Ordens de complexidade mais comuns



# Análise de complexidade

## Notação de "O grande"

➔ Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa

- Para obter o tempo a menos de:
  - constantes multiplicativas (normalmente estas constantes são tempos de execução de operações atómicas)
  - parcelas menos significativas para valores grandes de ***n***
- Identificam-se as operações dominantes (mais frequentes ou muito mais demoradas) e determina-se o número de vezes que são executadas (e não o tempo de cada execução, que seria uma constante multiplicativa)
- Exprime-se o resultado com a notação de "O grande"

# Análise de complexidade

## Notação de “O grande”

- Definição: $T(n) = O(f(n))$ (lê-se: $T(n)$ é de ordem $f(n)$) se e só se existem constantes positivas $c$ e $n_0$ tal que $T(n) \leq c.f(n)$ para todo o $n > n_0$
- Esta notação é usada com três objectivos:
  - limitar o erro que é feito ao ignorar os termos menores nas fórmulas matemáticas
  - limitar o erro que é feito na análise ao desprezar parte do programa que contribui de forma mínima para o custo/complexidade total
  - permitir-nos classificar algoritmos de acordo com limites superiores no seu tempo de execução

# Análise de complexidade

## Notação de "O grande"

➔ Exemplos:

– $c_k\, n^k + c_{k-1}\, n^{k-1} + \ldots + c_0 = O(n^k)$

  ($c_i$ - constantes)

– $log_2\, n = O(log\, n)$

  (não se indica a base porque mudar de base é multiplicar por uma constante)

– $4 = O(1)$

  (usa-se 1 para ordem constante)

# Análise de complexidade

## Metodologia para determinar a complexidade

➔ Considere-se o seguinte código:

```
for (i = 0; i < n; i++)
{
   Instruções;
}
```

➔ A contabilização do número de instruções é simples:

***n*** iterações e, em cada uma, são executadas um número constante de instruções $\Rightarrow$ ***O(n)***

# Análise de complexidade

## Metodologia para determinar a complexidade

➔ Considere-se o seguinte código:

```
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
    {
        Instruções;
    }
```

➔ A contabilização do número de instruções é ainda simples:

o ciclo interno (for j) é ***O(n)*** e é executado ***n*** vezes $\Rightarrow$ ***O(n²)***

# Análise de complexidade

## Eficiência da Pesquisa Sequencial

```
int  PesquisaSequencial  (int  x,  int  V[],  int  n)
{
   int  i = 0, k = -1;      //  k = posição onde se encontra x em V
   while ( (i < n)  &&  (k == -1) )
      if (x == V[i])
         k = i;
      else
         if (V[i] < x)
            i = i + 1;
         else
            k = -2;
   return (k);
}
```

# Análise de complexidade

## Eficiência da Pesquisa Sequencial

➔ Eficiência temporal:

- A operação realizada mais vezes é o teste da condição de continuidade do ciclo **while**, que é no máximo ***n+1*** vezes (no caso de não encontrar **x** — o pior caso)
- Se **x** existir no array, o teste é realizado aproximadamente
  - ***n/2*** vezes (no caso médio) e,
  - ***1*** vez (no melhor caso)
- Logo, **T(n) = O(n) (linear)** no pior caso e no caso médio

# Análise de complexidade

## Eficiência da Pesquisa Sequencial

➔ Eficiência espacial:

- Gasta o espaço das variáveis locais (incluindo argumentos)
- Como os arrays são passados "por referência" (na linguagem C, o que é passado é o endereço do primeiro elemento do array), o espaço ocupado pelas variáveis locais é constante e independente do tamanho do array
- Logo, **S(n) = O(1) (constante)** em qualquer caso

# Análise de complexidade

## Eficiência da Pesquisa Binária

```
int PesquisaBinaria (int x, int V[], int n)
{
   int inicio = 0, fim = n – 1, meio, k = -1;
   while ( (inicio <= fim) && (k == -1) )
   {
      meio = (inicio + fim) / 2;
      if (x == V[meio])
         k = meio;
      else
         if (x < V[meio])
            fim = meio - 1;
         else
            inicio = meio + 1;
   }
   return (k);
}
```

# Análise de complexidade

## Eficiência da Pesquisa Binária

➔ Eficiência temporal:

- Em cada iteração, o tamanho do sub-array a analisar é dividido por um factor de aproximadamente igual a **2**
- Ao fim de **k** iterações, o tamanho do sub-array a analisar é aproximadamente igual a $\mathbf{n/2^k}$
- Se não existir no array o valor procurado, o ciclo só termina quando $\mathbf{n/2^k \approx 1 \Leftrightarrow \log_2 n - k \approx 0 \Leftrightarrow k \approx \log_2 n}$
- No pior caso, o número de iterações é aproximadamente igual a $\mathbf{\log_2 n}$. Logo, $\mathbf{T(n) = O(\log n)}$ (logarítmico)

# Análise de complexidade

## Eficiência da Ordenação por Borbulhagem

```
void Ordenar_Borbulagem  (int  V[],  int  n) {
    int  k, Num_trocas, aux;
    do{
        Num_trocas = 0;
        for (k = 0; k < n-1; k++)
            if (V[k] > V[k+1])
            {
                aux = V[k];
                V[k] = V[k+1];
                V[k+1] = aux;
                Num_trocas++;
            }
    }while (Num_trocas != 0);
}
```

# Análise de complexidade

## Eficiência da Ordenação por Borbulhagem

➔ No melhor caso:

- Apenas 1 execução do ciclo **for** : **n-1** comparações e **0** trocas
- No total : **(n-1)** operações $\Rightarrow$ **O(n)**

➔ No pior caso:

- 1ª passagem pelo ciclo **for** : **n–1** comparações e **n–1** trocas
- 2ª passagem pelo ciclo **for** : **n–1** comparações e **n–2** trocas
- . . .
- (n-1)-ésima passagem pelo ciclo for: **n-1** comparações e **1** troca
- Total de comparações : $(n-1)x(n-1) = n^2 - 2n + 1$
- Total de trocas : $(n-1) + (n-2) + ... + 1 = n(n-1)/2 = (n^2 - n)/2$
- Total de operações : $(n^2 - 2n + 1) + ((n^2 - n)/2) \Rightarrow$ $\mathbf{O(n^2)}$

# Análise de complexidade

## Eficiência da Ordenação por Borbulhagem

```
void Ordenar_Borbulagem_Opt  (int  V[],  int  n) {
    int  k, kk, fim = n-1, aux;
    do{
        kk = 0;
        for (k = 0; k < fim; k++)
            if (V[k] > V[k+1]) {
                aux = V[k];
                V[k] = V[k+1];
                V[k+1] = aux;
                kk = k;
            }
        fim = kk;
    }while (kk != 0);
}
```

# Análise de complexidade

## Eficiência da Ordenação por Borbulhagem

➔ No melhor caso:

- Apenas 1 execução do ciclo **for** : **n-1** comparações e **0** trocas
- No total : **(n-1)** operações $\Rightarrow$ **O(n)**

➔ No pior caso:

- 1ª passagem pelo ciclo **for** : **n–1** comparações e **n–1** trocas
- 2ª passagem pelo ciclo **for** : **n–2** comparações e **n–2** trocas
- . . .
- (n-1)-ésima passagem pelo ciclo for: **1** comparação e **1** troca
- Total de comparações : $(n-1) + (n-2) + ... + 1 = n(n-1)/2$
- Total de trocas : $(n-1) + (n-2) + ... + 1 = n(n-1)/2$
- Total de operações : $2x(n(n-1)/2) = n(n-1) = (n^2-n) \Rightarrow$ **O(n²)**