

# ESD 2

Estrutura de  
Dados

## Complexidade de Algoritmos

Prof. Fernando Sambinelli

# Objetivo

---

- Por que analisar a complexidade dos algoritmos ?
  - É fundamental para projetar **algoritmos eficientes**
- Costuma-se medir um algoritmo em termos de **tempo** de execução ou o **espaço** (ou memória) usado



# Objetivo

---

- Como medir a eficiência?
  - Para o tempo, podemos considerar o tempo absoluto (em minutos, segundos, etc.). **Medir o tempo absoluto não é interessante por depender da máquina**
  - Em Análise de Algoritmos conta-se o número de operações consideradas relevantes realizadas pelo algoritmo e expressa-se esse número como uma função de  $n$ . Essas operações podem ser comparações, operações aritméticas, movimento de dados, etc



# Exemplo de Análise de Complexidade

```
int findMax(int A[], int n) {  
    int max= A[0];           ← 2 operações  
    int i= 1;                ← 1 operação  
    while (i <= n-1) {       ← n operações  
        if (A[i]>max)      ← 2 ops   n-1 vezes  
            max= A[i];       ← 2 ops  
        i= i+1;              ← 2 ops  
    }  
    return max;              ← 1 operação  
}
```

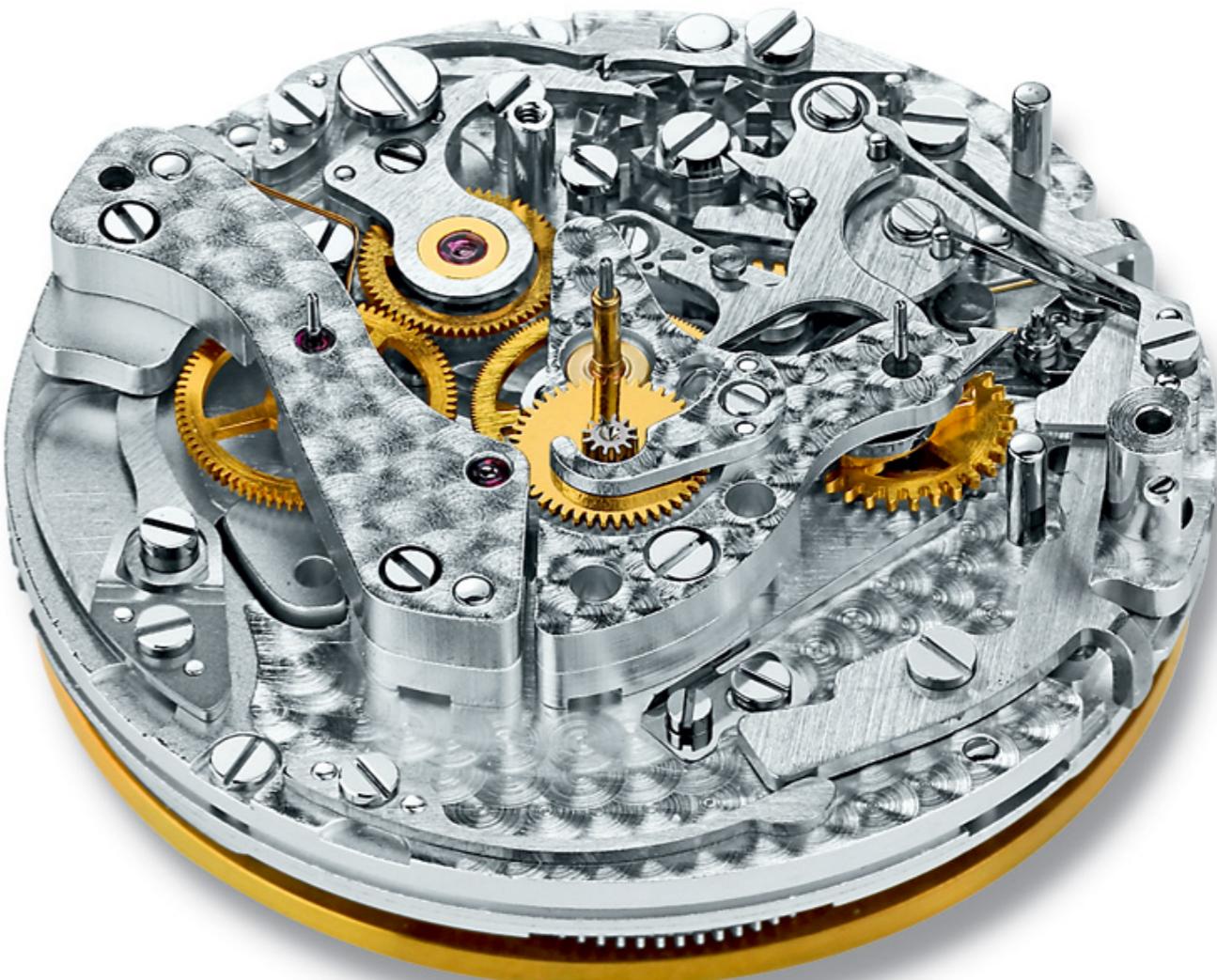
**Caso Mais Favorável (A[0] é o maior elemento):**  
 $t(n) = 2 + 1 + n + 4*(n - 1) + 1 = 5n$  operações

**Pior Caso:**  
 $t(n) = 2 + 1 + n + 6*(n - 1) + 1 = 7n - 2$  operações

# Complexidade de Tempo

---

- Como exemplo, considere o número de operações de cada um dos dois algoritmos que resolvem o mesmo problema, como função de  $n$
- **Algoritmo 1:**  $f_1(n) = 2n^2 + 5n$  operações
- **Algoritmo 2:**  $f_2(n) = 500n + 4000$  operações
- Dependendo do valor de  $n$ , o Algoritmo 1 pode requerer mais ou menos operações que o Algoritmo 2
- Obs: compare as duas funções para  $n = 10$  e  $n = 100$



# Comportamento Assintótico

- **Algoritmo 1:**  $f_1(n) = 2n^2 + 5n$  operações
- **Algoritmo 2:**  $f_2(n) = 500n + 4000$  operações
- Um caso de particular interesse é quando  $n$  tem valor muito grande ( $n \rightarrow \infty$ ), denominado **comportamento assintótico**
- Os termos inferiores e as constantes multiplicativas contribuem pouco na comparação e podem ser descartados
- O importante é observar que  $f_1(n)$  cresce com  $n^2$  ao passo que  $f_2(n)$  cresce com  $n$ .
- Um crescimento quadrático é considerado pior que um crescimento linear. Assim, vamos preferir o Algoritmo 2 ao Algoritmo 1



# Complexidade de Algoritmo - Melhor Caso

- Definido pela letra grega  $\Omega$  (Ômega)
- É o menor tempo de execução em uma entrada de tamanho N
- É pouco usado, por ter aplicação em poucos casos
- Exemplo: Se tivermos uma lista de N números e quisermos encontrar algum deles, assume-se que a complexidade no melhor caso é  $\Omega(1)$ , pois assume-se que o número estaria logo na cabeça da lista



# Complexidade de Algoritmo - Caso Médio

---

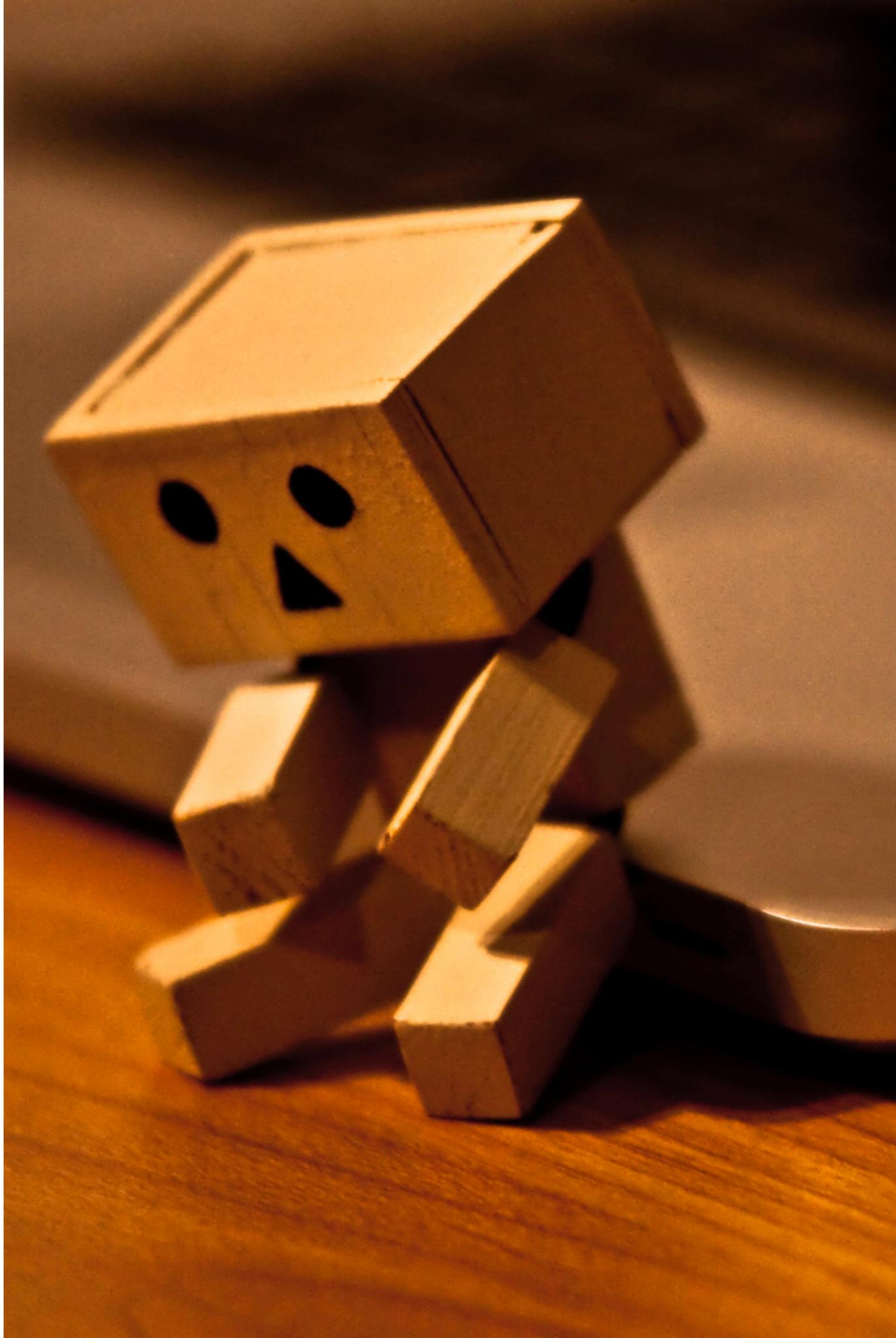
- Definido pela letra grega  $\theta$  (Theta)
- Dos três, é o mais difícil de se determinar
- Deve-se obter a média dos tempos de execução de todas as entradas de tamanho  $N$ , ou baseado em probabilidade de determinada condição ocorrer
- Exemplo: A complexidade média é  $P(1) + P(2) + \dots + P(N)$  Para calcular a complexidade média, é preciso conhecer as probabilidades de  $P_i$



# Complexidade de Algoritmo - Pior Caso

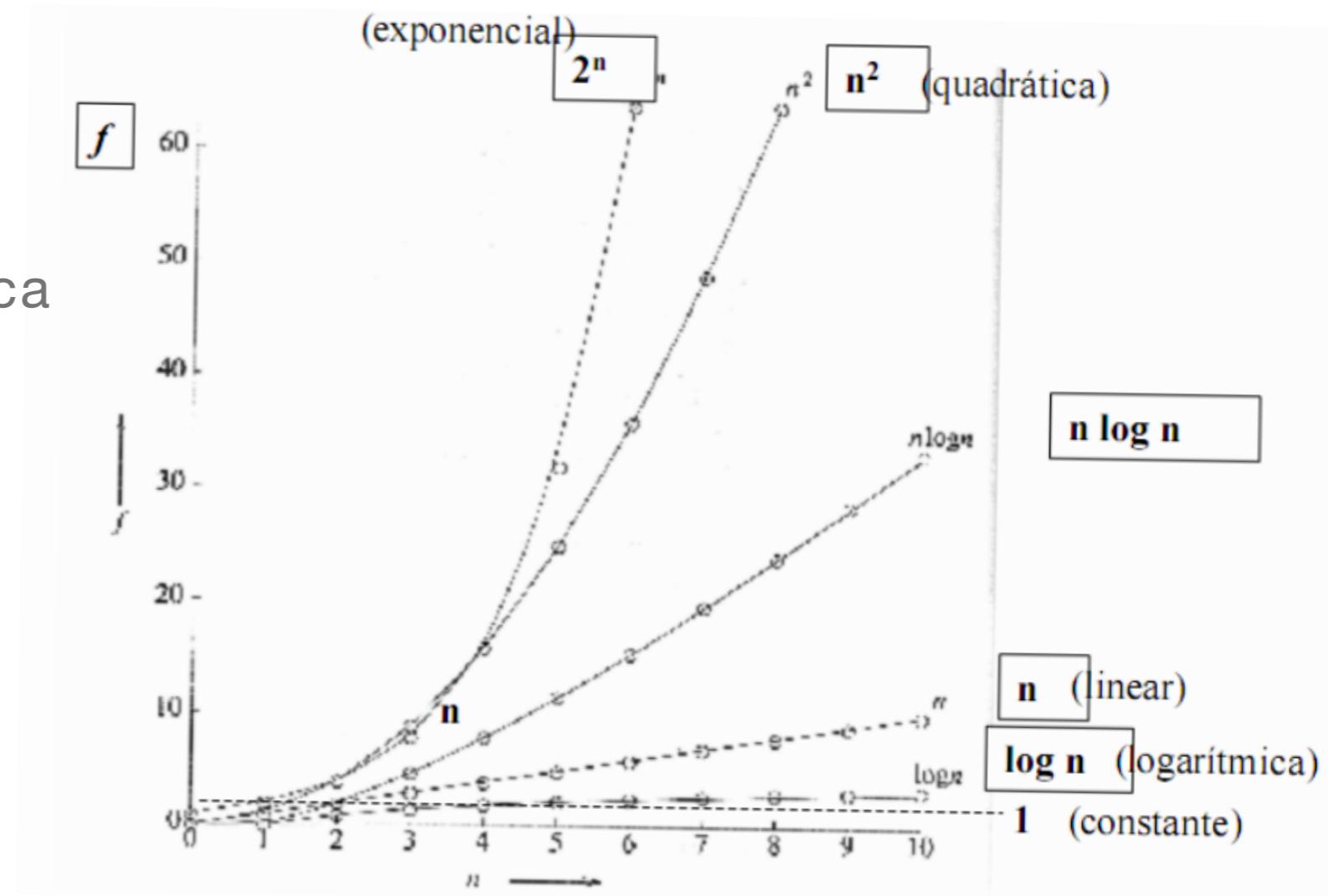
---

- Representado pela letra grega O (**O** maiúsculo)
- Muitas vezes chamado de notação **Big O**
- É o método mais fácil de se obter. Baseia-se no maior tempo de execução sobre todas as entradas de tamanho N
- Exemplo: Se tivermos uma lista de N números e quisermos encontrar algum deles, assume-se que a complexidade no pior caso é O (N), pois assume-se que o número estaria, no pior caso, no final da lista



# Classes de Problemas

- Podemos classificar os algoritmos baseados em suas complexidades:
  - Complexidade Constante
  - Complexidade Linear
  - Complexidade Logarítmica  $N.\log N$
  - Complexidade Quadrática
  - Complexidade Cúbica
  - Complexidade Exponencial



# Complexidade Constante

---

- São os algoritmos de complexidade  $O(1)$
- Independente do tamanho  $N$  de entradas
- É o único em que as instruções dos algoritmos são executadas num tamanho fixo de vezes
- Exemplo: uma busca em um hash “perfeito” seria  $O(1)$

```
Função Vazia(Lista: TipoLista) : Booleano;
Início
    Vazia := Lista.Primeiro = Lista.Ultimo;
Fim;
```

# Complexidade Linear

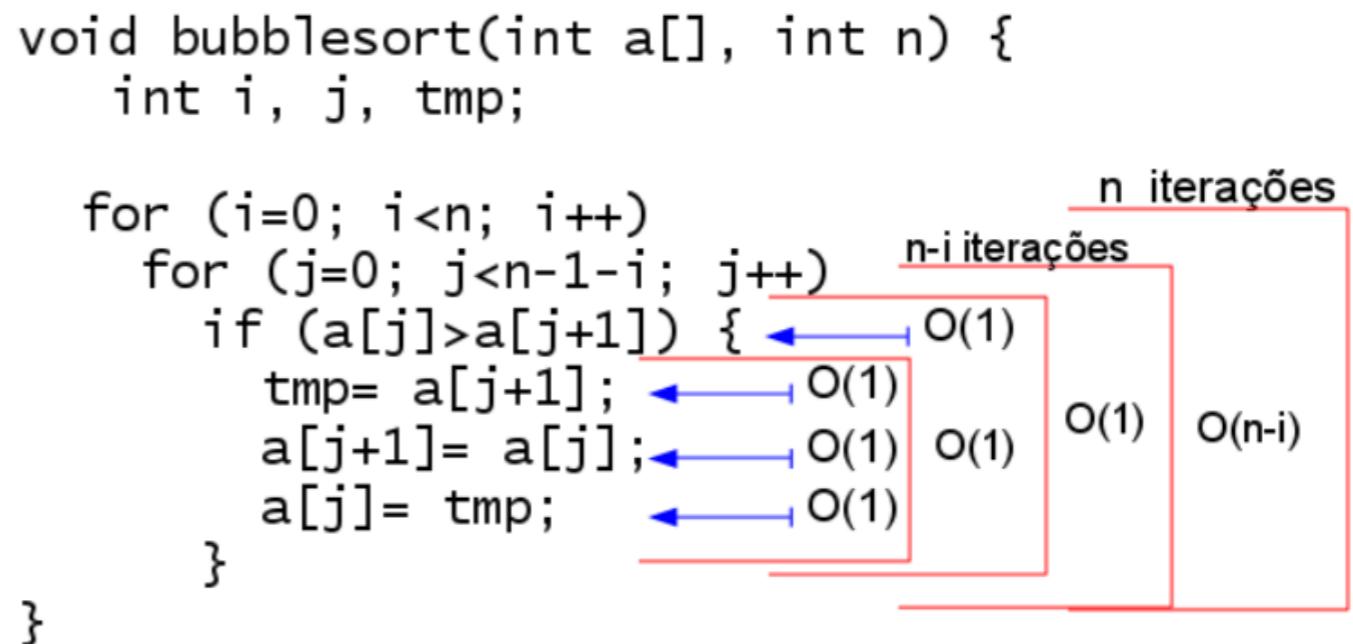
- São os algoritmos de complexidade  $O(N)$
- Uma operação é realizada em cada elemento de entrada, como por exemplo, uma pesquisa de elementos em uma lista

```
Procedimento Busca(Lista: TipoLista; x: TipoElem; Var pos:inteiro)
  i: inteiro;
  Início
    i:=1;
    enquanto (Lista.Elemento[i] <> x) faça
      i := i+1;
    se (i >= Lista.MaxTam) então
      pos := -1 // Elemento não encontrado
    senão
      pos := i; // Elemento encontrado na posição i
  Fim;
```

# Complexidade Quadrática

- São os algoritmos de complexidade  $O(N^2)$
- Itens são processados aos pares, geralmente com um loop dentro do outro

```
void bubblesort(int a[], int n) {  
    int i, j, tmp;  
  
    for (i=0; i<n; i++)  
        for (j=0; j<n-1-i; j++)  
            if (a[j]>a[j+1]) {  
                tmp= a[j+1];  
                a[j+1]= a[j];  
                a[j]= tmp;  
            }  
    }  
}
```



The diagram illustrates the time complexity of the bubble sort algorithm. The outer loop, labeled 'n iterações', runs from i=0 to n-1. The inner loop, labeled 'n-i iterações', runs from j=0 to n-1-i. Inside the inner loop, there are four assignments: tmp=a[j+1], a[j+1]=a[j], a[j]=tmp, and a[j]=tmp. Each of these assignments is labeled O(1). The total cost of the inner loop is O(n-i), and the total cost of the entire algorithm is O(n^2).

```
Procedimento SomaMatriz(Mat1, Mat2, MatRes: Matriz);  
i, j: inteiro;  
Inicio  
    for i:=1 to n do  
        for j:=1 to n do  
            MatRes[i,j] := Mat1[i, j] + Mat2[i,j];  
Fim;
```

# Complexidade Cúbica

- São os algoritmos de complexidade  $O(N^3)$
- Itens são processados três a três, geralmente com um *loop* dentro dos outros dois

```
Procedimento MultiplicaMatriz(Mat1, Mat2, MatRes: Matriz);
    i, j, k: inteiro;
    Início
        for i := 1 to M do
            for j := 1 to P do
                for k := 1 to N do
                    MatRes[i,j] :=(Mat1[i,k] * Mat2[k,j]) + MatRes[i,j];
    Fim;
```

# Complexidade Logarítmica

- São os algoritmos de complexidade  $O(\log N)$
- Ocorre tipicamente em algoritmos que dividem o problema em problemas menores .
- Quando  $n$  é um mil,  $\log n$  é aproximadamente 10

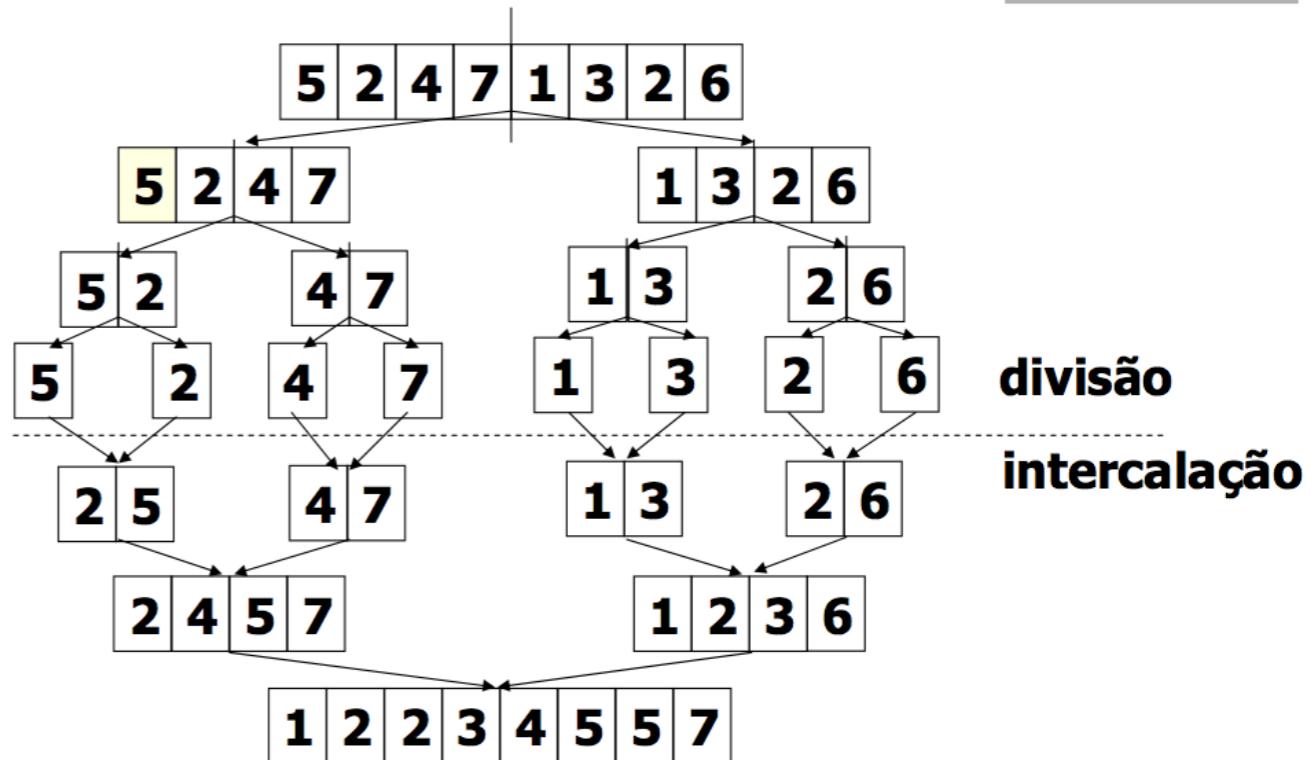
$$\begin{aligned}\log_2 N &= x \\ \log_2 1.024 &= x \\ 2^x &= 2^{10} \\ x &= 10\end{aligned}$$

- Ex: O algoritmo de Busca Binária

```
89 ⊕ public boolean containsKey(int p_valor) {  
90     return contains(raiz, p_valor);  
91 }  
92  
93 ⊕ private boolean contains(BTNode p_no, int p_valor)  
94     if (p_no == null) {  
95         return false;  
96     }  
97  
98     if (p_no.val == p_valor) {  
99         return true;  
100    }  
101  
102    if (p_valor < p_no.val) {  
103        return contains(p_no.esq, p_valor);  
104    } else {  
105        return contains(p_no.dir, p_valor);  
106    }  
107 }  
108 }
```

# Complexidade NLogN

- Como o próprio nome diz, são algoritmos que têm complexidade  $O(N\log N)$
- Ocorre tipicamente em algoritmos que dividem o problema em problemas menores, porém juntando posteriormente a solução dos problemas menores
- Ex: O algoritmo de ordenação Merge Sort



**divisão**  
**intercalação**

# Limites Inferiores e Superiores

---

- Essas ordens vistas definem o **Limite Superior (*Upper Bound*)** dos Algoritmos, ou seja, qualquer que seja o tamanho da entrada, a execução será aquela determinada pelo algoritmo. Algumas otimizações podem ser feitas para melhorar o limite superior
- Existem, porém, os **Limites Inferiores (*Lower Bound*)** dos Algoritmos, que são pontos em que não são mais possíveis otimizações



# Limites Inferiores e Superiores

- Às vezes é necessário mostrar que, para um dado problema, qualquer que seja o algoritmo a ser usado, requer um certo número de operações: o **Limite Inferior**
- Para o problema de multiplicação de matrizes de ordem  $n$ , apenas para ler os elementos das duas matrizes de entrada leva  $O(n^2)$ . Assim uma cota inferior trivial é  $\Omega(n^2)$
- Na analogia anterior, o limite inferior não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta. Um limite inferior trivial para os 100 metros seria o tempo que a velocidade da luz leva para percorrer 100 metros no vácuo
- Se um algoritmo tem uma complexidade que é igual ao limite inferior do problema então o algoritmo é ótimo.
  - O algoritmo de *CopperSmith* e *Winograd* é de  $O(n^{2.376})$  mas o limite inferior é de  $\Omega(n^2)$ . Portanto não é ótimo. Acredita-se que esse limite superior ainda pode ser melhorado

Vamos avaliar a complexidade  
e alguns métodos de  
ordenação ?



<http://goo.gl/3dUuXG>



Prof. Fernando Sambinelli  
[sambinelli@ifsp.edu.br](mailto:sambinelli@ifsp.edu.br)



$$i = \frac{p}{4\pi r^2}$$

