

# Algoritmos e Estruturas de Dados II

2º Período Engenharia da Computação

Prof. Edwaldo Soares Rodrigues  
Email: [edwaldo.rodrigues@uemg.br](mailto:edwaldo.rodrigues@uemg.br)

# Algoritmos

- Um algoritmo é um procedimento de passos para realizar uma determinada tarefa
  - Este procedimento é composto de instruções que definem uma função
- Até o momento, vimos apenas códigos em C para descrever ideias
  - Já um algoritmo pode descrever ideias para pessoas que programem em outras linguagens de programação

# Algoritmos - Exemplo

## Código em C

```
int max(int[] a, int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

## Algoritmo

Entrada: Conjunto de números inteiros  $A$

Saída: Maior elemento deste conjunto

$m \leftarrow$  primeiro elemento de  $A$

Para todos os outros elementos  $A_i$  de  $A$

se o elemento  $A_i$  for maior que  $m$

$m \leftarrow A_i$

retorne o valor de  $m$

# Definição

- Apesar de ser um conjunto de regras que define uma sequência, não existe uma definição formal e geral de algoritmo
  - Podem ser incluídos algoritmos que não fazem cálculos
  - O algoritmo precisa sempre parar?
  - Um programa que não para é um algoritmo?
  - Podem ser incluídos programas que só param com intervenção do usuário?
  - Programas que dependem do usuário são algoritmos?
  - Algoritmos podem ter componentes aleatórios?

# Expressando Algoritmos

- Há também diversas maneiras de se expressar um algoritmo:
  - Linguagem natural (Português)
  - Pseudo-código (Listagem das operações matemáticas)
  - Diagramas (Quadrados e setas)
  - As próprias linguagens de programação

# Algoritmos

- Estamos mesmo assim interessados em estudar algoritmos em vez de códigos em linguagens de programação
- Isso porque queremos conclusões amplas sobre a análises que fazemos dos algoritmos
- Em programação, esta análise mais ampla permite ajudar pessoas que utilizam diferentes linguagens

# Modelos de comparação

- Nem sempre dois algoritmos que resolvem o mesmo problema são igualmente eficientes
- Então, como comparar algoritmos?
  - Alguns são mais rápidos
  - Outros gastam mais memória
  - Outros obtém soluções de melhor qualidade

# Medida de custo real

- Usualmente inadequada
  - Depende muito do sistema em geral
    - Hardware, compilador, quantidade disponível de memória
- Pode ser vantajosa quando dois algoritmos têm comportamento similar
  - Os custos reais de operação são então considerados



# Medida por modelos matemáticos

- Especificamos um conjunto de operações e seus custos
  - Usualmente, apenas as operações mais importantes do algoritmo são consideradas
    - Comparações
    - Atribuições
    - Chamadas de função

# Complexidade de Algoritmos

- Para se medir o custo de um algoritmo, definimos uma **função de custo** ou **função de complexidade  $f$** 
  - $f(n)$  pode medir o tempo necessário para executar um algoritmo
  - $f(n)$  pode medir o tempo determinando quantas vezes uma operação relevante será executada
  - $f(n)$  pode medir a memória gasta para executar um algoritmo


# Complexidade de Algoritmos

- Considere o algoritmo abaixo:

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

# Complexidade de Algoritmos

- Para achar o maior elemento, ele guarda o valor do primeiro elemento do arranjo

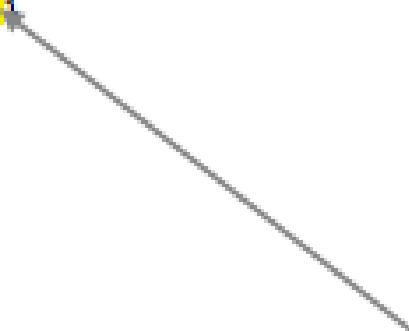


```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

# Complexidade de Algoritmos

- Para achar o maior elemento, ele guarda o valor do primeiro elemento do arranjo

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```



- e o compara com todos os outros elementos do arranjo

# Complexidade de Algoritmos

- Para um arranjo de  $n$  elementos e uma função  $f(n)$  do número de comparações temos que o custo do algoritmo é:

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

$$f(n) = n-1$$

# Complexidade de Algoritmos

- Note que a operação relevante deste algoritmo é o número de comparações.

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

- No caso geral, não é interessante medir o número de operações de atribuição pois elas só ocorrem quando a comparação retorna true

# Melhor caso, pior caso e caso médio

- Suponha agora um computador no qual as operações de atribuição são muito custosas

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

- Estamos então interessados no número de operações de atribuição como medida de tempo



# Melhor caso, pior caso e caso médio

- O melhor caso em relação a tempo é quando a comparação é sempre false pois temos apenas uma operação de atribuição

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

→ false

- Isso ocorre quando o primeiro elemento já é o maior

# Melhor caso, pior caso e caso médio

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

→ *false*

- Assim, pode-se dizer que o melhor caso é igual a  $f(n) = 1$ , pois sempre teremos apenas uma operação de atribuição;

# Melhor caso, pior caso e caso médio


- Na situação contrária, se a comparação é sempre verdadeira, temos nosso pior caso em relação ao número de atribuições

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

→ *true*

- Isso ocorre quando os elementos do arranjo estão em ordem crescente;

# Melhor caso, pior caso e caso médio

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  true  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

- Assim, pode-se dizer que o pior caso é igual a  $f(n) = n$ , pois sempre teremos uma operação de atribuição para cada elemento do arranjo

# Melhor caso, pior caso e caso médio

- Já o caso médio, ou o caso esperado, é a média de todos os casos possíveis. O caso médio é muito mais difícil de ser obtido pois depende da nossa estimativa de probabilidade de cada entrada

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

→ *true / false*

# Melhor caso, pior caso e caso médio

- Se supormos que a probabilidade de uma comparação ser true é de 50%, podemos dizer que o caso médio é  $f(n) = 1 + (n-1)/2$ , porém está é uma suposição pouco provável na prática

```
int max(int a[], int n){  
    int i, temp;  
    temp = a[0];  
    for (i = 1; i < n; i++){  
        if (temp < a[i]){  
            temp = a[i];  
        }  
    }  
    return temp;  
}
```

→ *true / false*

# 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.
- (Compare as duas funções para  $n = 10$  e  $n = 100$ .)

# Complexidade de tempo

- 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.

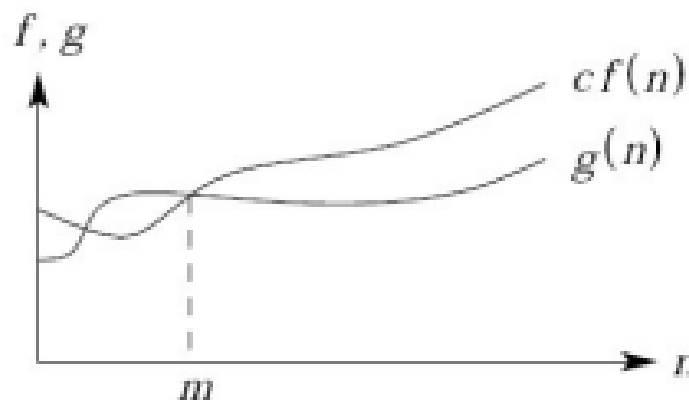


# Notação O

- A notação O é utilizada para descrever a tendência de uma função
- Ela é muito utilizada em computação para classificar algoritmos de acordo com a taxa de crescimento de suas funções

# Dominação assintótica

- Uma função  $f(n)$  *domina* assintoticamente outra função  $g(n)$  se existem duas constantes positivas  $c$  e  $m$  tais que, para  $n \geq m$ , temos  $|g(n)| \leq c \times |f(n)|$
- Ou seja, para números grandes  $cf(n)$  será sempre maior que  $g(n)$ , para alguma constante  $c$



# Notação O para dominação assintótica

- Quando uma função  $g(n) = O(f(n))$ , dizemos que:
  - $g(n)$  é O de  $f(n)$   $\rightarrow$   $g(n)$  é da ordem de no máximo  $f(n)$
  - $f(n)$  domina  $g(n)$  assintoticamente
- Ex: Quando dizemos que o tempo de execução  $T(n)$  de um programa é  $O(n^2)$ , significa que existem constantes **c** e **m** tais que, para valores de  $n \geq m$ ,  $T(n) \leq cn^2$ .
- A notação é importante para comparar algoritmos pois não estamos interessados em funções exatas de custo mas sim, do comportamento da função

# Exemplo 1

- Dadas as funções:
  - $g(n) = (n+1)^2$
  - $f(n) = n^2$
- As duas funções se dominam assintoticamente pois:
  - $g(n) = O(f(n))$  pois  $g(n) \leq 4f(n)$  para  $n > 1$
  - $f(n) = O(g(n))$  pois  $f(n) \leq g(n)$

# Exemplo 2

- Dadas as funções:
  - $g(n) = (n+1)^2$
  - $f(n) = n^3$
- A segunda função domina a primeira
  - $g(n) = O(f(n))$  pois  $g(n) \leq cf(n)$  para um  $c$  coerente e a partir de  $n > 1$
- Porém a primeira não domina a segunda:
  - Pois  $f(n) \leq cg(n)$  seria uma afirmação falsa para qualquer  $c$  e a partir de qualquer  $n$

# Limites fortes

- $g(n) = 3n^3 + 2n^2 + n$  é  $O(n^3)$ 
  - Pois  $g(n) \leq 6n^3$  para  $n \geq 0$
- É claro que também podemos dizer que  $g(n) = O(n^4)$ , porém esta seria uma afirmação fraca;
  - Para analisar comportamentos de algoritmos, estamos interessados em afirmações fortes

# Operações com notação O

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

$$c \times O(f(n)) = O(f(n)) \quad c = \text{constante}$$

$$O(f(n)) + O(f(n)) = O(f(n))$$

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

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

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

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

# Operações com notação O

- Uma função sempre se domina pois basta a multiplicar por uma constante para que seja maior que ela mesmo

$$\begin{aligned} f(n) &= O(f(n)) \\ c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\ O(f(n)) + O(f(n)) &= O(f(n)) \\ O(O(f(n))) &= O(f(n)) \\ O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\ O(f(n))O(g(n)) &= O(f(n)g(n)) \\ f(n)O(g(n)) &= O(f(n)g(n)) \end{aligned}$$



# Operações com notação O

- O mesmo vale para esta função  $O(f(n))$  multiplicada por uma constante, pois basta multiplicarmos  $f(n)$  por uma constante ainda maior para que seja dominada

$$\begin{aligned}f(n) &= O(f(n)) \\c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\O(f(n)) + O(f(n)) &= O(f(n)) \\O(O(f(n))) &= O(f(n)) \\O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\O(f(n))O(g(n)) &= O(f(n)g(n)) \\f(n)O(g(n)) &= O(f(n)g(n))\end{aligned}$$

# Operações com notação O

- A soma de duas funções dominadas por  $f(n)$  é ainda dominada por  $f(n)$  pois esta diferença pode ainda ser compensada por uma constante

$$\begin{aligned}f(n) &= O(f(n)) \\c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\O(f(n)) + O(f(n)) &= O(f(n)) \\O(O(f(n))) &= O(f(n)) \\O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\O(f(n))O(g(n)) &= O(f(n)g(n)) \\f(n)O(g(n)) &= O(f(n)g(n))\end{aligned}$$

# Operações com notação O

- Se uma função é dominada por uma função dominada por  $f(n)$ , a primeira função é também dominada por  $f(n)$

$$\begin{aligned}f(n) &= O(f(n)) \\c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\O(f(n)) + O(f(n)) &= O(f(n)) \\O(O(f(n))) &= O(f(n)) \\O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\O(f(n))O(g(n)) &= O(f(n)g(n)) \\f(n)O(g(n)) &= O(f(n)g(n))\end{aligned}$$

# Operações com notação O

- A soma de duas funções será dominada pela maior função que as domina

$$\begin{aligned}f(n) &= O(f(n)) \\c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\O(f(n)) + O(f(n)) &= O(f(n)) \\O(O(f(n))) &= O(f(n)) \\O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\O(f(n))O(g(n)) &= O(f(n)g(n)) \\f(n)O(g(n)) &= O(f(n)g(n))\end{aligned}$$

# Operações com notação O

- A multiplicação de duas funções será dominada pela multiplicação das funções que as dominavam

$$\begin{aligned}f(n) &= O(f(n)) \\c \times O(f(n)) &= O(f(n)) \quad c = \text{constante} \\O(f(n)) + O(f(n)) &= O(f(n)) \\O(O(f(n))) &= O(f(n)) \\O(f(n)) + O(g(n)) &= O(\max(f(n), g(n))) \\O(f(n))O(g(n)) &= O(f(n)g(n)) \\f(n)O(g(n)) &= O(f(n)g(n))\end{aligned}$$

# Classes de Algoritmos

- Com as funções de complexidade de cada programa, podemos compará-los
- No caso geral, para valores grandes de  $n$ , um programa que leva tempo  $O(n)$  é melhor que um programa que leva tempo  $O(n^2)$

$$f(n) = O(1)$$

- Dizemos que algoritmos  $O(1)$  têm complexidade constante
- Eles têm o mesmo desempenho independente do valor de  $n$
- Estes são algoritmos onde as instruções são executadas um número fixo de vezes
- Ex: determinar se um número é par ou ímpar

$$f(n) = O(\log n)$$

- Dizemos que algoritmos  $O(\log n)$  têm complexidade logarítmica
- Algoritmos típicos desta classe são os que dividem o problema em outros menores na forma de divisão e conquista
- Tempo de execução pode ser considerado menor do que uma constante grande
  - Quando  $n$  é 1000000,  $\log n$  é aproximadamente 20
  - A base do logaritmo tem impacto pequeno
- Ex: Busca binária



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

- Dizemos que algoritmos  $O(n)$  têm complexidade linear
- Nestes algoritmos, se dobrarmos  $n$ , o tempo de resposta também dobra
- Algoritmos típicos desta classe são os que realizam um pequeno trabalho sobre cada um dos  $n$  elementos da entrada
- Ex: Busca sequencial, calcular fatorial

$$f(n) = O(n \log n)$$

- Algoritmos típicos desta classe são os que quebram o problema em problema menores, fazendo uma operação em cada um dos elementos e depois combinam as soluções
- Se  $n$  é 1 milhão,  $n \log_2 n$  é cerca de 20 milhões
- Se  $n$  é 2 milhões,  $n \log_2 n$  é cerca de 42 milhões
- Ou seja,  $f(n)$  é pouco mais que o dobro ao dobrar o tamanho de  $n$
- Ex: ordenação(eficiente)

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

- Dizemos que algoritmos  $O(n^2)$  têm complexidade quadrática
- Nestes algoritmos, se dobramos  $n$ , o tempo de resposta se multiplica por 4
  - São úteis para problemas relativamente pequenos
- Algoritmos típicos desta classe, são os que processam elementos par a par
- Ex: ordenação (ineficiente), imprimir uma matriz

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

- Dizemos que algoritmos  $O(n^3)$  têm complexidade cúbica
- Nestes algoritmos, se dobramos  $n$ , o tempo de resposta se multiplica por 8
  - São úteis apenas para problemas muito pequenos
- Ex: Multiplicação de matrizes

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

- Dizemos que algoritmos  $O(2^n)$  têm complexidade exponencial
- Quando  $n=20$ , o tempo de execução já é cerca de 1 milhão
  - Não são algoritmos muito úteis na prática
- Algoritmos típicos desta classe são os que usam força-bruta para resolver problemas que envolvem combinações
  - São testadas todas as soluções possíveis até que se encontre a certa

$$f(n) = O(n!)$$

- Dizemos que algoritmos  $O(n!)$  também têm complexidade exponencial, apesar de serem muito piores que  $O(2^n)$
- Não são úteis na prática
- Algoritmos típicos desta classe são os que usam força-bruta para resolver problemas que envolvam permutações
  - O computador levaria séculos para resolver problemas pequenos

# Comparação de Classes de Complexidade

Função de custo	Tamanho $n$					
	10	20	30	40	50	60
$n$	0,00001 s	0,00002 s	0,00003 s	0,00004 s	0,00005 s	0,00006 s
$n^2$	0,0001 s	0,0004 s	0,0009 s	0,0016 s	0,0.35 s	0,0036 s
$n^3$	0,001 s	0,008 s	0,027 s	0,64 s	0,125 s	0.316 s
$n^5$	0,1 s	3,2 s	24,3 s	1,7 min	5,2 min	13 min
$2^n$	0,001 s	1 s	17,9 min	12,7 dias	35,7 anos	366 séc.
$3^n$	0,059 s	58 min	6,5 anos	3855 séc.	$10^8$ séc.	$10^{13}$ séc.

# Limite superior (Upper bound)

- Seja dado um problema, por exemplo, multiplicação de duas matrizes quadradas  $n \times n$ .
- Conhecemos um algoritmo para resolver este problema (pelo método trivial) de complexidade  $O(n^3)$ .
- Sabemos assim que a complexidade deste problema não deve superar  $O(n^3)$ , uma vez que existe um algoritmo que o resolve com esta complexidade.
- Uma cota superior ou limite superior (upper bound) deste problema é  $O(n^3)$ .

$O(n^3)$

---

- A cota superior de um problema pode mudar se alguém descobrir um outro algoritmo melhor.

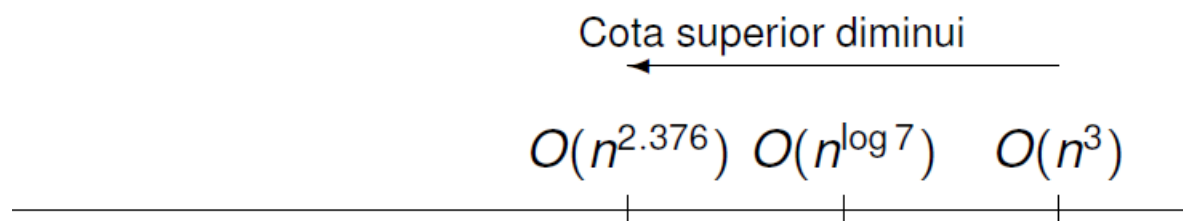


# Limite superior (Upper bound)

- O Algoritmo de Strassen reduziu a complexidade para  $O(n^{\log 7})$ . Assim a cota superior do problema de multiplicação de matrizes passou a ser  $O(n^{\log 7})$ .



- Coppersmith e Winograd melhoraram ainda para  $O(n^{2.376})$ .



- Note que a cota superior de um problema depende do algoritmo. Pode diminuir quando aparece um algoritmo melhor.

# Analogia com record mundial

- A cota superior para resolver um problema é análoga ao record mundial de uma modalidade de atletismo. Ele é estabelecido pelo melhor atleta (algoritmo) do momento. Assim como o record mundial, a cota superior pode ser melhorada por um algoritmo (atleta) mais veloz.

“Cota superior” da corrida de 100 metros rasos:

Ano	Atleta (Algoritmo)	Tempo
1988	Carl Lewis	9s92
1993	Linford Christie	9s87
1993	Carl Lewis	9s86
1994	Leroy Burrell	9s84
1996	Donovan Bailey	9s84
1999	Maurice Greene	9s79
2002	Tim Montgomery	9s78
2007	Asafa Powell	9s74
2008	Usain Bolt	9s72
2008	Usain Bolt	9s69
2009	Usain Bolt	9s58

# Sequência de Fibonacci

- Para projetar um algoritmo eficiente, é fundamental preocupar-se com a sua complexidade. Como exemplo: considere a sequência de Fibonacci.
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, :::

- A sequência pode ser definida recursivamente:

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

- Dado o valor de  $n$ , queremos obter o  $n$ -ésimo elemento da sequência.
- Vamos apresentar dois algoritmos e analisar sua complexidade.

# Sequência de Fibonacci

- Seja a função  $\text{fibonacci}(n)$  que calcula o  $n$ -ésimo elemento da sequência de Fibonacci.

**Input:** Valor de  $n$

**Output:** O  $n$ -ésimo elemento da sequência de Fibonacci

Function  $\text{fibonacci}(n)$

```
1: if  $n = 0$  then
2:   return 0
3: else
4:   if  $n = 1$  then
5:     return 1
6:   else
7:     return  $\text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$ 
8:   end if
9: end if
```

- Experimente rodar este algoritmo para  $n = 100$  :-) A complexidade é  $O(2^n)$ .
- (Mesmo se uma operação levasse um picosegundo,  $2^{100}$  operações levariam  $3 \times 10^{13}$  anos = 30.000.000.000.000 anos.)

# Sequência de Fibonacci

Function *fibonacci2*(*n*)

```
1: if n = 0 then  
2:   return 0  
3: else  
4:   if n = 1 then  
5:     return 1  
6:   else  
7:     penultimo ← 0  
8:     ultimo ← 1  
9:     for i ← 2 until n do  
10:      atual ← penultimo + ultimo  
11:      penultimo ← ultimo  
12:      ultimo ← atual  
13:    end for  
14:    return atual  
15:  end if  
16: end if
```

- A complexidade agora passou de  $O(2^n)$  para  $O(n)$ .

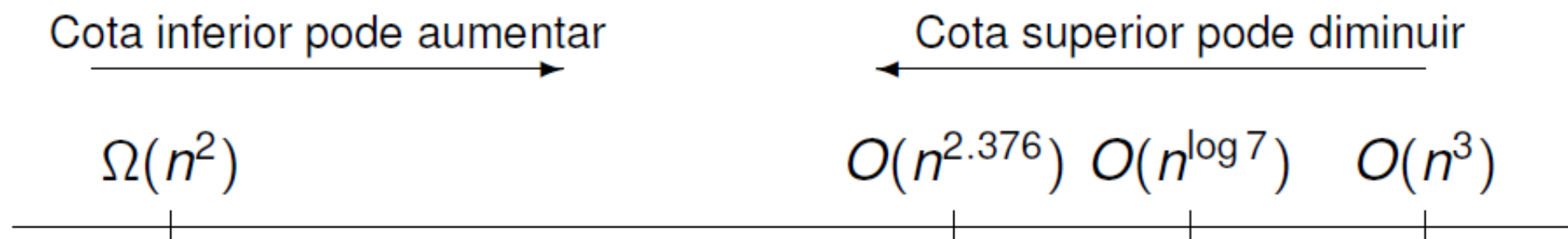
# Limite inferior (lower bound)

- As vezes é possível demonstrar que, para um dado problema, qualquer que seja o algoritmo a ser usado, o problema requer pelo menos um certo número de operações.
- Essa complexidade é chamada cota inferior (lower bound) do problema.
- Note que a cota inferior depende do problema mas não do particular algoritmo.

# Limite inferior para multiplicação de matrizes

- Para o problema de multiplicação de matrizes quadradas  $n \times n$ , apenas para ler os elementos das duas matrizes de entrada ou para produzir os elementos da matriz produto leva tempo  $O(n^2)$ . Assim uma cota inferior trivial é  $\Omega(n^2)$ .
- Na analogia anterior, uma cota inferior de uma modalidade de atletismo não dependeria mais do atleta. Seria algum tempo mínimo que a modalidade exige, qualquer que seja o atleta. Uma cota inferior trivial para os 100 metros rasos seria o tempo que a velocidade da luz leva para percorrer 100 metros no vácuo.

# Aproximando os dois limites



- Se um algoritmo tem uma complexidade que é igual à cota inferior do problema, então ele é assintoticamente ótimo;
- O algoritmo de Coppersmith e Winograd é de  $O(n^{2.376})$  mas a cota inferior (conhecida até hoje) é de  $(n^2)$ . Portanto não podemos dizer que ele é ótimo;
- Pode ser que esta cota superior possa ainda ser melhorada. Pode também ser que a cota inferior de  $(n^2)$  possa ser melhorada (isto é “aumentada”). Para muitos problemas interessantes, pesquisadores dedicam seu tempo tentando encurtar o intervalo (“gap”) até encostar as duas cotas;



# Importância

- Considere 5 algoritmos com as complexidades de tempo. Suponhamos que uma operação leve 1 ms:

$n$	$f_1(n) = n$	$f_2(n) = n \log n$	$f_3(n) = n^2$	$f_4(n) = n^3$	$f_5(n) = 2^n$
16	0.016s	0.064s	0.256s	4s	1m 4s
32	0.032s	0.16s	1s	33s	46 dias
512	0.512s	9s	4m 22s	1 dia 13h	$10^{137}$ séculos

- Verifique se resolveria usar uma máquina mais rápida onde uma operação leve 1 ps (pico segundo) ao invés de 1 ms: ao invés de  $10^{137}$  séculos seriam  $10^{128}$  séculos;
- Podemos muitas vezes melhorar o tempo de execução de um programa otimizando o código (ex: usar  $x + x$  ao invés de  $2x$ , evitar re-cálculo de expressões já calculadas, etc.);
- Entretanto, melhorias muito mais substanciais podem ser obtidas se usarmos um algoritmo diferente, com outra complexidade de tempo, ex: obter um algoritmo de  $O(n \log n)$  ao invés de  $O(n^2)$ ;

# Algoritmos e Estruturas de Dados II

- Bibliografia:

- Básica:

- CORMEN, Thomas, RIVEST, Ronald, STEIN, Clifford, LEISERSON, Charles. Algoritmos. Rio de Janeiro: Elsevier, 2002.
    - EDELWEISS, Nina, GALANTE, Renata. Estruturas de dados. Porto Alegre: Bookman. 2009. (Série livros didáticos informática UFRGS,18).
    - ZIVIANI, Nívio. Projeto de algoritmos com implementação em Pascal e C. São Paulo: Cengage Learning, 2010.

- Complementar:

- ASCENCIO, Ana C. G. Estrutura de dados. São Paulo: Pearson, 2011. ISBN: 9788576058816.
    - PINTO, W.S. Introdução ao desenvolvimento de algoritmos e estrutura de dados. São Paulo: Érica, 1990.
    - PREISS, Bruno. Estruturas de dados e algoritmos. Rio de Janeiro: Campus, 2000.
    - TENEMBAUM. Aaron M. Estruturas de dados usando C. São Paulo: Makron Books. 1995. 884 p. ISBN: 8534603480.
    - VELOSO, Paulo A. S. Complexidade de algoritmos: análise, projeto e métodos. Porto Alegre, RS: Sagra Luzzatto, 2001

# Algoritmos e Estruturas de Dados II

