Algoritmos e Estrutura de Dados
Noções de análise de complexidade Parte: *Introdução*

Professor Alexandre Magno de Sousa

Departamento de Computação e Sistemas

Sumário

- Introdução
- Exemplos de análise de algoritmos
- Crescimento assintótico das funções
- Classes de complexidade
- Classes de algoritmos e medidas de tempo
- Funções típicas e estimativas *O-grande*
- Referências bibliográficas

Introdução

- O projeto de algoritmos é fortemente influenciado pelo estudo de seus **comportamentos**.
- Depois que decisões de projetos são feitas o algoritmo deve ser **implementado**.
- Vários algoritmos podem ser utilizados e deve-se estudar aspectos de ***tempo de execução e espaço***.
- Muitos destes algoritmos são encontrados em áreas como: *pesquisa operacional*, *otimização*, *teoria dos grafos*, *estatística*, *probabilidade*, *entre outras*.

Introdução

Existem dois problemas bem distintos:

- **Análise de um algoritmos em particular:**

Qual é o custo de usar um dado algoritmo para resolver um problema específico?

- **Análise de uma classe de algoritmos:**

Qual é o algoritmo de menor custo para resolver um problema particular?

Introdução

- Determinando o custo, tem-se a medida de ***dificuldade para resolver*** um dado problema.
- Se o custo de um dado algoritmo é o menor, pode-se concluir que o algoritmo é **ótimo**.
- Podem existir vários algoritmos para resolver o mesmo problema e ***é necessário escolher o melhor***.
- Se a mesma **medida de custo** é aplicada, é possível **compará-los** e escolher o mais adequado.

Introdução

- O custo de execução de um algoritmo pode ser medido de várias maneiras.
- Uma delas é a medição do tempo de execução real de um algoritmo em um computador.

Objeções:

(i) resultados dependem do ***compilador***.

(ii) resultados dependem do ***hardware***.

(iii) resultados dependem do tamanho da ***memória***.

Introdução

- Para medir o custo de execução é comum definir uma ***função de custo*** ou *função de complexidade* ***f***.
- ***f(n)*** é a medida de tempo necessário para executar um algoritmo para um problema de tamanho *n*.
- Se ***f(n)*** é a medida da quantidade da memória necessária para executar um algoritmo, então é chamada de ***função de complexidade de espaço***.

Análise de algoritmos: exemplo

- Considere um algoritmo para encontrar o maior elemento de um vetor de inteiros.

1	2	3	4	5	6	7	8	9	10

Análise de algoritmos: exemplo

seja *f(n)* o custo do o número de comparações

```
void max( int v[ ], int n, int *max ){
    *max = v[0], i;
    for( i = 1; i < n; i++ )
        if( *max < v[i] )
            *max = v[i];
}
```

Análise de algoritmos: exemplo

seja *f(n)* o custo do o número de comparações

```
void max( int v[ ], int n, int *max ){
   *max = v[0], i;
   for( i = 1; i < n; i++ )
      if( *max < v[i] )
         *max = v[i];
}
```

$$f(n) = \sum_{1}^{n-1} 1 = (n-1) \times 1 = n-1$$

Análise de algoritmos: exemplo

seja *f(n)* o custo do o número de comparações

```
void max( int v[ ], int n, int *max ){
    *max = v[0], i;
    for( i = 1; i < n; i++ )
        if( *max < v[i] )
            *max = v[i];
}
```

$$f(n) = n - 1, n > 0$$

$$f(n) = \sum_{1}^{n-1} 1 = (n-1) \times 1 = n - 1$$

Análise de algoritmos

- **Melhor caso**: menor tempo de execução.
- **Pior caso**: maior tempo de execução.
- **Caso médio ou caso esperado**: média dos tempos de execução de todas as entradas de tamanho *n*.
  - Uma distribuição de probabilidades sobre o conjunto de entradas de tamanho *n* é suposta e custo é obtido com base nessa distribuição.

Análise: pesquisa sequencial

seja *f(n)* o custo do o número de comparações

- **Pesquisa sequencial**: examina registros na ordem em que eles aparecem no arquivo.

Caso	Custo
Melhor caso	$f(n) = 1$
Pior caso	$f(n) = n$
Caso médio	$f(n) = (n + 1)/2$

Análise de algoritmos:

seja *f(n)* o custo do o número de comparações

```
void maxMin1( int v[ ], int n, int *max, int *min ){
    *max = v[0]; *min = v[0];
    int i;
    for( i = 1; i < n; i++ ){
        if( v[i] > *max )
            *max = v[i];
        if( v[i] < *min )
            *min = v[i];
    }
}
```

Análise de algoritmos:

seja *f(n)* o custo do o número de comparações

```
void maxMin1( int v[ ], int n, int *max, int *min ){
    *max = v[0]; *min = v[0];
    int i;
    for( i = 1; i < n; i++ ){
        if( v[i] > *max )
            *max = v[i];
        if( v[i] < *min )
            *min = v[i];
    }
}
```

$$f(n) = \sum_{1}^{n-1} 2 = (n-1) \times 2 = 2n - 2$$

Análise de algoritmos:

seja *f(n)* o custo do o número de comparações

```
void maxMin1( int v[ ], int n, int *max, int *min ){
    *max = v[0]; *min = v[0];
    int i;
    for( i = 1; i < n; i++ ){
        if( v[i] > *max )
            *max = v[i];
        if( v[i] < *min )
            *min = v[i];
    }
}
```

$$f(n) = 2(n-1), n > 0$$

$$f(n) = \sum_{1}^{n-1} 2 = (n-1) \times 2 = 2n - 2$$

Análise de algoritmos:

seja *f(n)* o custo do o número de comparações

```
void maxMin2( int v[ ], int n, int *max, int *min ){
    *max = v[0]; *min = v[0];
    int i;
    for( i = 1; i < n; i++ ){
        if( v[i] > *max )
            *max = v[i];
        else if( v[i] < *min )
            *min = v[i];
    }
}
```

Caso	Custo
Melhor caso	$f(n) = n - 1$
Pior caso	$f(n) = 2(n - 1)$
Caso médio	$f(n) = 3n/2-3/2$

Análise de algoritmos:

seja *f(n)* o custo do o número de comparações

Para *maxMin2*, segue os seguintes cálculos:

- **Melhor caso:** vetor em ordem crescente.

$$f(n) = \sum_{1}^{n-1} 1 = (n-1) \times 1 = n-1$$

- **Pior caso**: vetor em ordem decrescente.

$$f(n) = \sum_{1}^{n-1} 2 = (n-1) \times 2 = 2n-2$$

- **Caso médio**: $f(n) = (n-1) - \frac{(n-1)}{2} = \frac{3n}{2} - \frac{3}{2}$

```
void maxMin3( int v[ ], int n, int *max, int *min ){
    int i = 2, fimDoLaco;
    if( (n % 2) > 0){ v[n] = v[n - 1]; fimDoLaco = n; }
    else fimDoLaco = n – 1;
    if( v[0] > v[1] ){ *max = v[0]; *min = v[1]; }
    else { *max = v[1]; *min = v[0] }
    while( i < fimDoLaco){
        if( v[i] > v[i + 1]){
            if( v[i] > *max) *max = v[i];
            if( v[i + 1] < *min) *min = v[i + 1];
        } else{
            if( v[i] < *min ) *min = v[i];
            if( v[i + 1] > *max ) *max = v[i + 1];
        }
        i =  i + 2
    }
}
```

$$f(n) = \left(\sum_{i=2}^{n-2} 3 \right) + 1 = \left(\sum_{i=1}^{\frac{n-2}{2}} 3 \right) + 1 = \frac{3n-6}{2} + 1 = \frac{3n}{2} - 2$$

```
void maxMin3( int v[ ], int n, int *max, int *min ){
    int i = 2, fimDoLaco;
    if( (n % 2) > 0){ v[n] = v[n - 1]; fimDoLaco = n; }
    else fimDoLaco = n – 1;
    if( v[0] > v[1] ){ *max = v[0]; *min = v[1]; }
    else { *max = v[1]; *min = v[0] }
    while( i < fimDoLaco){
        if( v[i] > v[i + 1]){
            if( v[i] > *max) *max = v[i];
            if( v[i + 1] < *min) *min = v[i + 1];
        } else{
            if( v[i] < *min ) *min = v[i];
            if( v[i + 1] > *max ) *max = v[i + 1];
        }
        i =  i + 2
    }
}
```

1 comparação

1 comparação
+
2 comparações

ou

2 comparações

3 comparações

$$f(n) = \left(\sum_{i=2}^{n-2} 3 \right) + 1 \Rightarrow \left(\sum_{i=1}^{\frac{n-2}{2}} 3 \right) + 1 = \frac{3n-6}{2} + 1 = \frac{3n}{2} - 2$$

Análise de algoritmos:
comparação do custo de *f(n)* dos algoritmos

Algoritmos	Melhor caso	Pior caso	Caso médio
maxMin1	*2(n – 1)*	*2(n – 1)*	*2(n – 1)*
maxMin2	*n – 1*	*2(n – 1)*	*3n/2 – 3/2*
maxMin3	*3n/2 – 2*	*3n/2 – 2*	*3n/2 – 2*

Séries e somatórios

$$\sum_{j=i}^{n} c = (n-i+1) \times c$$

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

$$\sum_{j=0}^{\infty} \left(\frac{1}{a}\right)^j = \frac{1}{1-(1/a)}$$

$$\sum_{j=0}^{n} \left(\frac{1}{a}\right)^j = \frac{1-(1/a)^n}{1-(1/a)}$$

$$\sum_{j=0}^{n} 2^n = 2^{n+1} - 1$$

$$\sum_{j=0}^{n} \frac{1}{2^j} = 2 - \frac{1}{2^n}$$

$$\sum_{j=0}^{n} a^j = \frac{a^{n+1}-1}{a-1} \quad (a \neq 1)$$

$$\sum_{j=1}^{n} j^2 = \frac{n(n+1)(2n+1)}{6}$$

Notações Auxiliares

Notações Padrão: exponenciais

$$a^0 = 1$$

$$a^1 = a$$

$$a^{-1} = \frac{1}{a}$$

$$(a^m)^n = a^{mn}$$

$$(a^n)^m = a^{nm}$$

$$a^m a^n = a^{m+n}$$

Notações padrão: logaritmos

$\lg n = \log_2 n$	logaritmo binário
$\ln n = \log_e n$	logaritmo natural
$\lg^k n = (\lg n)^k$	exponenciação
$\lg\lg n = \lg(\lg n)$	composição

Notações padrão: logaritmos

$$a = b^{\log_b a},$$

$$\log_c(ab) = \log_c a + \log_c b,$$

$$\log_b a^n = n \log_b a,$$

$$\log_b a = \frac{\log_c a}{\log_c b},$$

Notações padrão: logaritmos

$$\log_b(1/a) = -\log_b a,$$

$$\log_b a = \frac{1}{\log_a b},$$

$$a^{\log_b c} = c^{\log_b a},$$

$$\sqrt[b]{n^a} = b^{a/b \times \log_b n} = n^{a/b \times \log_b b} = n^{a/b \times 1} = n^{a/b}$$

Exercícios Dirigidos

Análise de algoritmos:

seja *f(n)* o custo do n.º de atribuições

```
for( i = soma = 0; i < n; i++ )
    soma += a[i];
```

Análise de algoritmos:

seja *f(n)* o custo do n.º de atribuições

```
for( i = 0; i < n; i++){
   for( j = 1, soma = a[0]; j <= i; j++)
       soma += a[j];
   printf("Soma de 0 até %d é %d", i, soma);
}
```

Análise de algoritmos:

seja *f(n)* o custo do n.º de atribuições

```
for( i = 4; i < n; i++ ){
   for( j = i – 3, soma = a[ i – 4 ]; j <= i; j++ )
        soma += a[ j ];
   printf("Soma de %d até %d é %d", (i-4), i, soma);
}
```

Análise de algoritmos:

seja *f(n)* o custo do n.º de atribuições

```
for( i = 0, comprimento = 1; i < n – 1; i++ ){
   for( i1=i2=k=i; k < n – 1 && a[k] < a[k+1]; k++,i2++);
      if( comprimento < i2 – i1 + 1)
         comprimento = i2 – i1 +1;
}
```

Análise de algoritmos:

seja *f(n)* o custo do n.º de atribuições

```
int buscaBinaria( int arr[], int arrTamanho, int chave){
    int baixo = 0, meio, alto = arrTamanho – 1;
    while( baixo <= alto ){
        meio = ( baixo + alto )/2;
        if( chave < arr[ meio ] )
            alto = meio – 1;
        else if( arr[ meio ] < chave )
            baixo = meio + 1;
        else return meio; //retorna o índice da chave encontrada
    }
    return -1; // caso não encontrado, retorna -1 indicando falha
}
```

Análise Assintótica de Funções

Notação *O*-Grande

- O tempo exato de execução de um algoritmo é uma expressão complexa, assim ele é apenas estimado

- A análise assintótica é uma estimativa utilizada para entender o tempo de execução quando executado sobre entradas grandes

Notação O-Grande

- Seja a função $f(n) = 6n^3 + 2n^2 + 20n + 45$
- O termo de mais alta ordem é: $6n^3$
- Desconsiderando o coeficiente 6, dizemos que f é assintoticamente, no máximo, n^3
- A notação ***assintótica*** ou ***O-grande*** para descrever esse relacionamento é:

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

Notação O-Grande

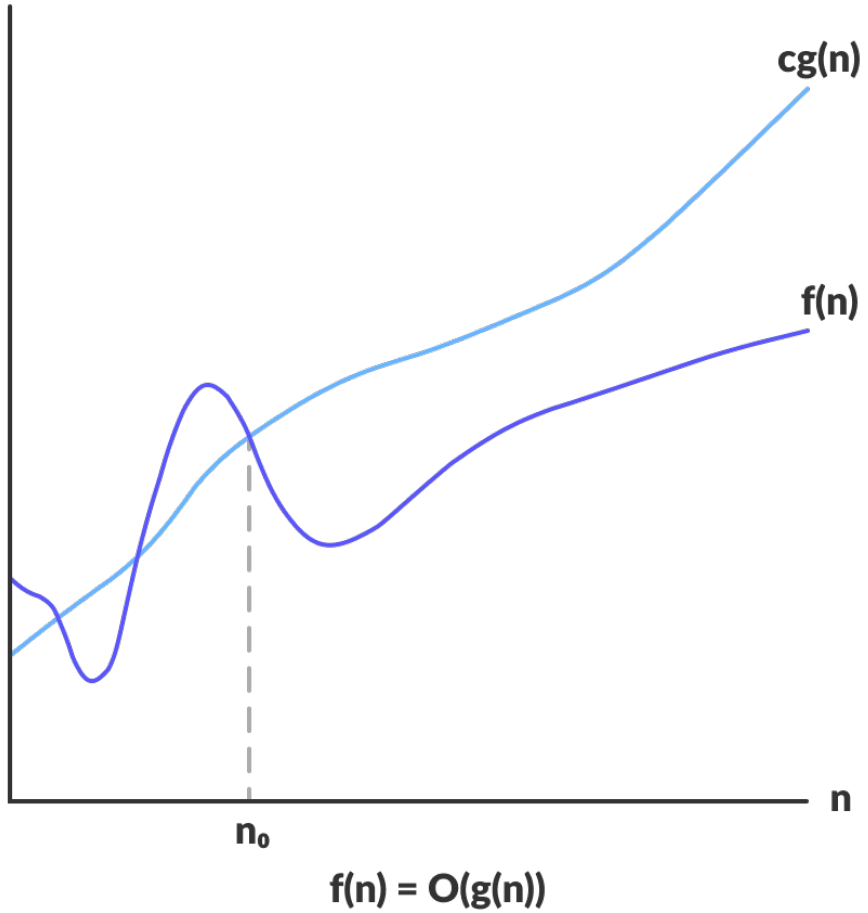
DEFINIÇÃO

Sejam f e g funções, f,g: N → R+

Digamos que f(n) = O(g(n)) se existem inteiros positivos c e n_0 tais que para todo inteiro $n \geq n_0$:

$$f(n) \leq c * g(n)$$

Comportamento assintótico



Notação O-Grande

- Quando $f(n) = O(g(n))$ dizemos que $g(n)$:
  - é um limitante superior para $f(n)$

    ou
  - é limitante superior assintótico para $f(n)$
- Isso enfatiza que estamos suprimindo fatores constantes.

Notação *O*-Grande: exemplo

- Seja $f(n) = 5n^3 + 2n^2 + 22n + 6$
- Termo de mais alta ordem: $5n^3$
- Desconsiderando seu coeficiente, tem-se que: $f(n) = O(n^3)$

Notação *O*-Grande: exemplo

- Verificação da definição formal:

Tornando $c = 6$ e $n_0 = 10$

Então, $5n^3 + 2n^2 + 22n + 6 \leq 6n^3$ *para todo* $n \geq 10$

Notação O-Grande

- $f(n) = O(n^4)$ porque $n^4 > n^3$, portanto, ainda é um limitante superior assintótico sobre f.

- $f(n)$ não é $O(n^2)$, porque independente dos valores que sejam atribuídos a c e n_0, a definição permanece insatisfeita.

Como definir c e n_0 para f(n) e O-grande

Notação O-Grande

- Seja a função: $f(n) = 2n^2 + 3n + 1$

$$f(n) = O(n^2), \text{ onde } g(n) = n^2$$

- Então, o valor de n_0 e c é obtido resolvendo a desigualdade:

$$2n^2 + 3n + 1 \leqslant cn^2$$

$$\frac{2n^2}{n^2} + \frac{3n}{n^2} + \frac{1}{n^2} \leqslant c$$

$$2 + \frac{3}{n} + \frac{1}{n^2} \leqslant c$$

Desigualdade com duas incógnitas, diferentes pares de constantes n_0 e c para a mesma função podem ser encontradas!

Notação O-Grande

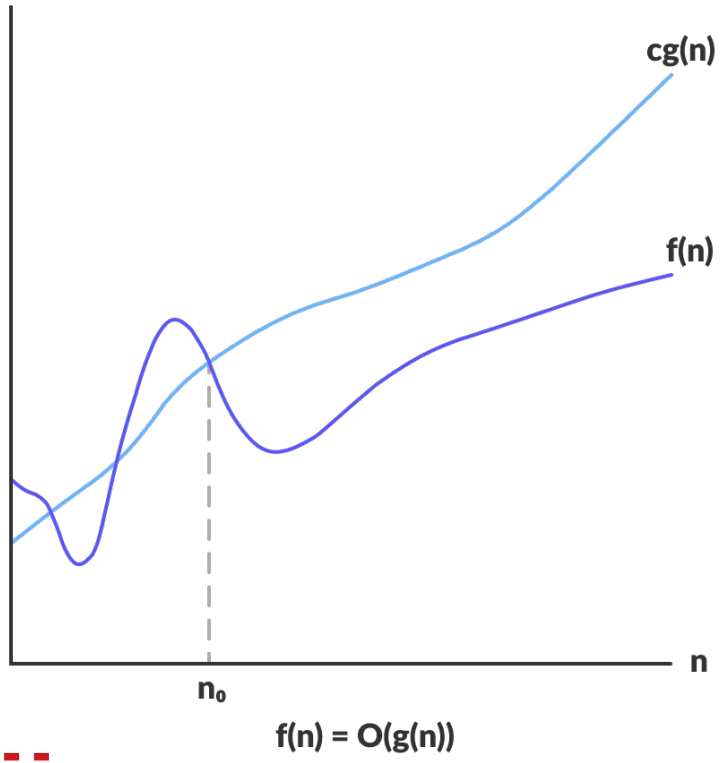
- Para escolher os melhores valores de n_0 e c:
  - *deve ser determinado para qual valor de n_0 um certo termo em f(n) se torna o maior e permanece dessa forma!*
- Os dois únicos candidatos são $2n^2$ e 3n:
  - $2n^2 > 3n$, que é válida para n > 1.
  - Assim, $n_0 = 2$ e substituindo:

$$2+\frac{3}{n}+\frac{1}{n^2}\leqslant c$$

$$2+\frac{3}{2}+\frac{1}{2^2}\leqslant c$$

$$3.75\leqslant c$$

n0 = 2 e c = 3.75



Notação O-Grande

Tabela: pares de valores de c e n_0 calculados a partir de f(n) e O-grande.

c	6	3,75	3,11	2,81	2,64	2,53	2,45	…	2
n_0	1	2	3	4	5	6	7	...	∞

- Qual o significado prático dos pares de c e n_0 listados acima?
  - Para um g(n) fixo, existe um ***número infinito de pares*** pode ser identificado.
  - A definição estabelece que ***quase sempre*** g(n) > f(n): para todos n não menores do que n_0!
  - ***O valor de c depende do n_0 escolhido!***

Notação *o*-pequeno

- A notação **O-grande** diz que uma função é assintoticamente ***não mais que*** outra.

- Pode-se dizer que uma função é assintoticamente ***menor que*** outra, por meio da notação ***o*-pequeno.**

- A diferença entre as notações ***O-grande*** é ***o-pequeno*** é análoga àquela entre **≤** e **<**.

Notação *o*-pequeno

DEFINIÇÃO

Sejam f e g funções, $f,g: N \rightarrow R^+$
Digamos que $f(n) = o(g(n))$ se *se existem inteiros positivos c e* n_0 *tais que para todo inteiro* $n \geq n_0$:

$$f(n) < c * g(n)$$

e na medida em que *n* se aproxima do infinito:

$$\lim_{n \rightarrow \infty} f(n)/g(n) = 0$$

Notação *o*-pequeno

- Em outras palavras, $f(n) = o(g(n))$ significa que, para qualquer número real $c > 0$ e n_0, onde $f(n) < c\ g(n)$ para todo $n \geq n_0$.

Notação *o*-pequeno: exemplos

1. $\sqrt{n} = o(n)$
2. $n = o(n \log \log n)$
3. $n \log \log n = o(n \log n)$
4. $n \log n = o(n^2)$
5. $n^2 = o(n^3)$
6. $f(n)$ nunca é $o(f(n))$

Notação Ω-grande

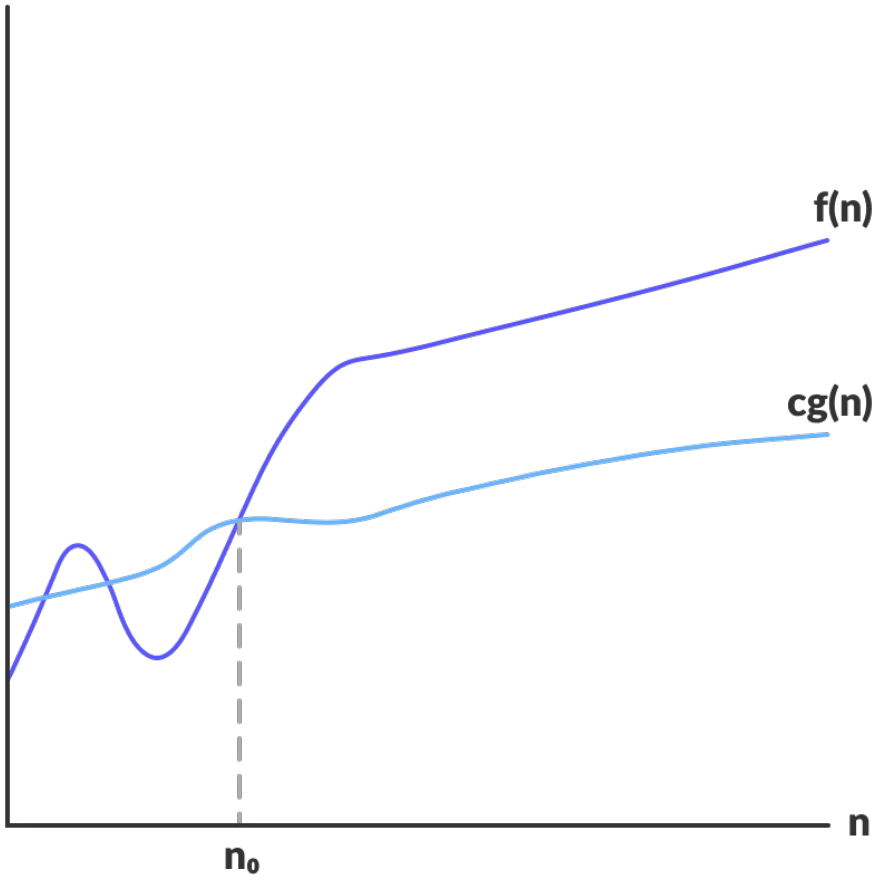
DEFINIÇÃO

Sejam f e g funções, $f,g: N \rightarrow R^+$

Digamos que $f(n) = \Omega(g(n))$ se existem inteiros positivos c e n0 tais que para todo inteiro n ≥ n0:

$$0 \leq c * g(n) \leq f(n)$$

Comportamento assintótico



$f(n) = \Omega(g(n))$

Notação ω-pequeno

DEFINIÇÃO

Sejam f e g funções, $f,g: N \rightarrow R^+$

Digamos que $f(n) = \omega(g(n))$ se existem inteiros positivos c e n_0 tais que para todo inteiro $n \geq n_0$:

$$0 \leq c * g(n) < f(n)$$

e na medida em que n se aproxima do infinito:

$$\lim_{n \rightarrow \infty} f(n)/g(n) = \infty$$

Notação Θ-grande

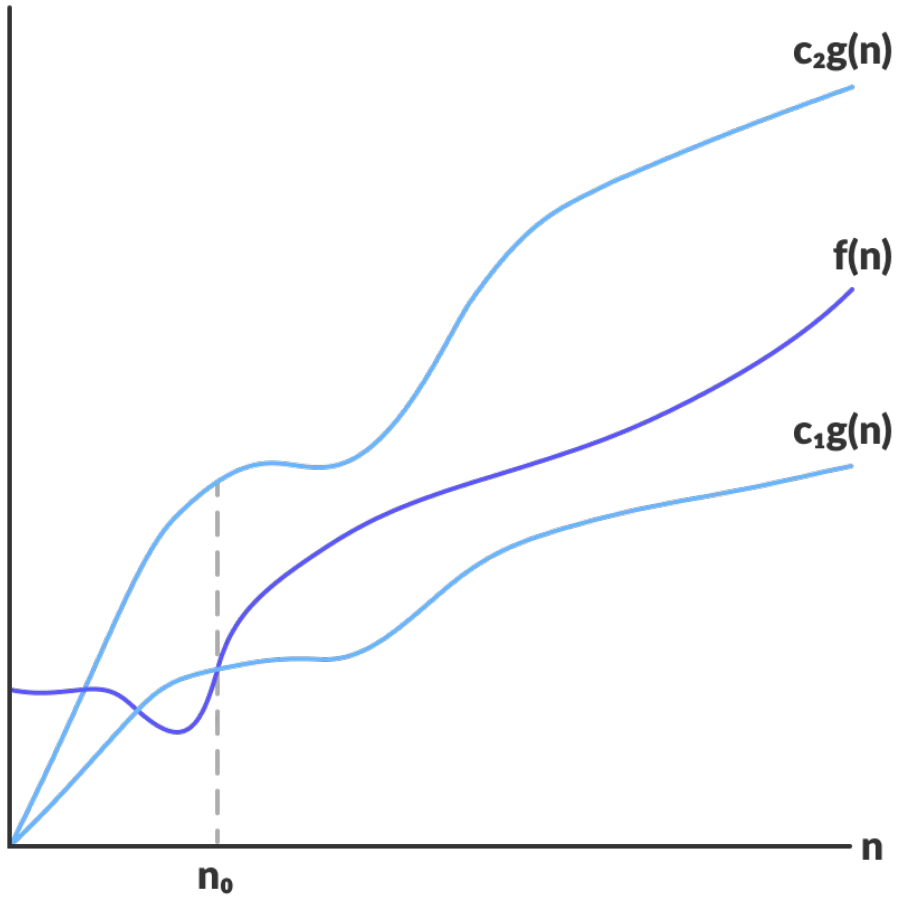
DEFINIÇÃO

Sejam f e g funções, $f,g: N \rightarrow R^+$

Digamos que $f(n) = \Theta(g(n))$ se existem inteiros positivos c_1, c_2 e n_0 tais que para todo inteiro $n \geq n_0$:

$$0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

Comportamento assintótico



$f(n) = \Theta(g(n))$

Analogia: comparação das funções f e g com número reais a e b

$f(n) = O(g(n))$ $a \leq b$

$f(n) = o(g(n))$ $a < b$

$f(n) = \Omega(g(n))$ $a \geq b$

$f(n) = \omega(g(n))$ $a > b$

$f(n) = \Theta(g(n))$ $a = b$

Operações com a notação *O-grande*

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

$$c \; X \; O(f(n)) = O(f(n)), \qquad c = constante$$

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

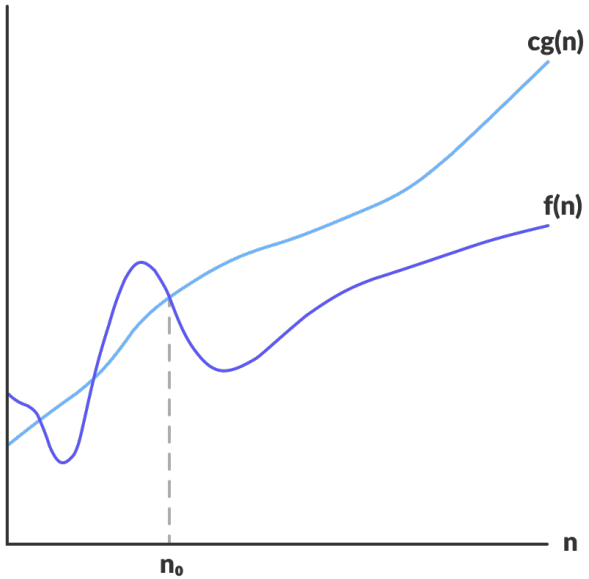
$$O(O(f(n))) = O(f(n))$$

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

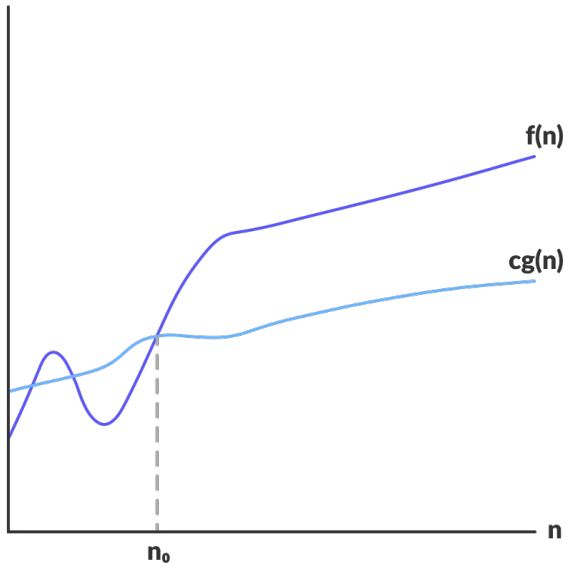
$$O(f(n))\, O(g(n)) = O(f(n)g(n))$$

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

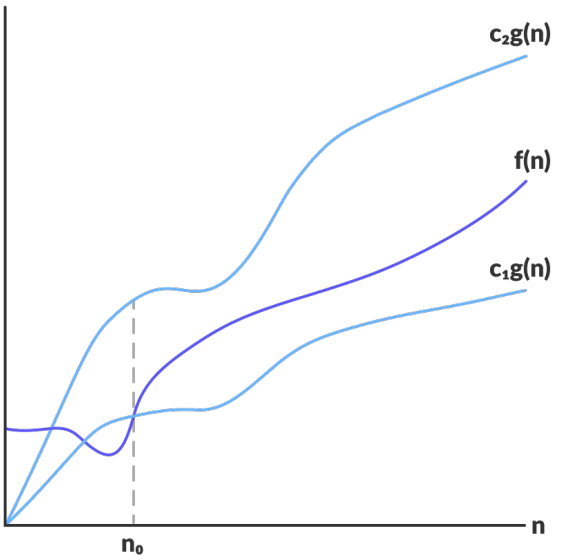
Comportamento assintótico



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



f(n) = Ω(g(n))



f(n) = Θ(g(n))

Técnicas de análise de algoritmos

Comando	Custo
Atribuição, leitura, escrita, condição	*O(1)*
Sequência de comandos	*Determinado pelo maior tempo na sequência*
Decisão	*Tempo da sequência dentro do comando mais a condição*

Técnicas de análise de algoritmos

Comando	Custo
Laços de repetição	*Soma do tempo da sequencia dentro do laço mais a condição de parada Multiplicado pelo número de iterações*
Procedimentos não recursivos	*Deve ser computado separadamente de acordo com as avaliações anteriores*

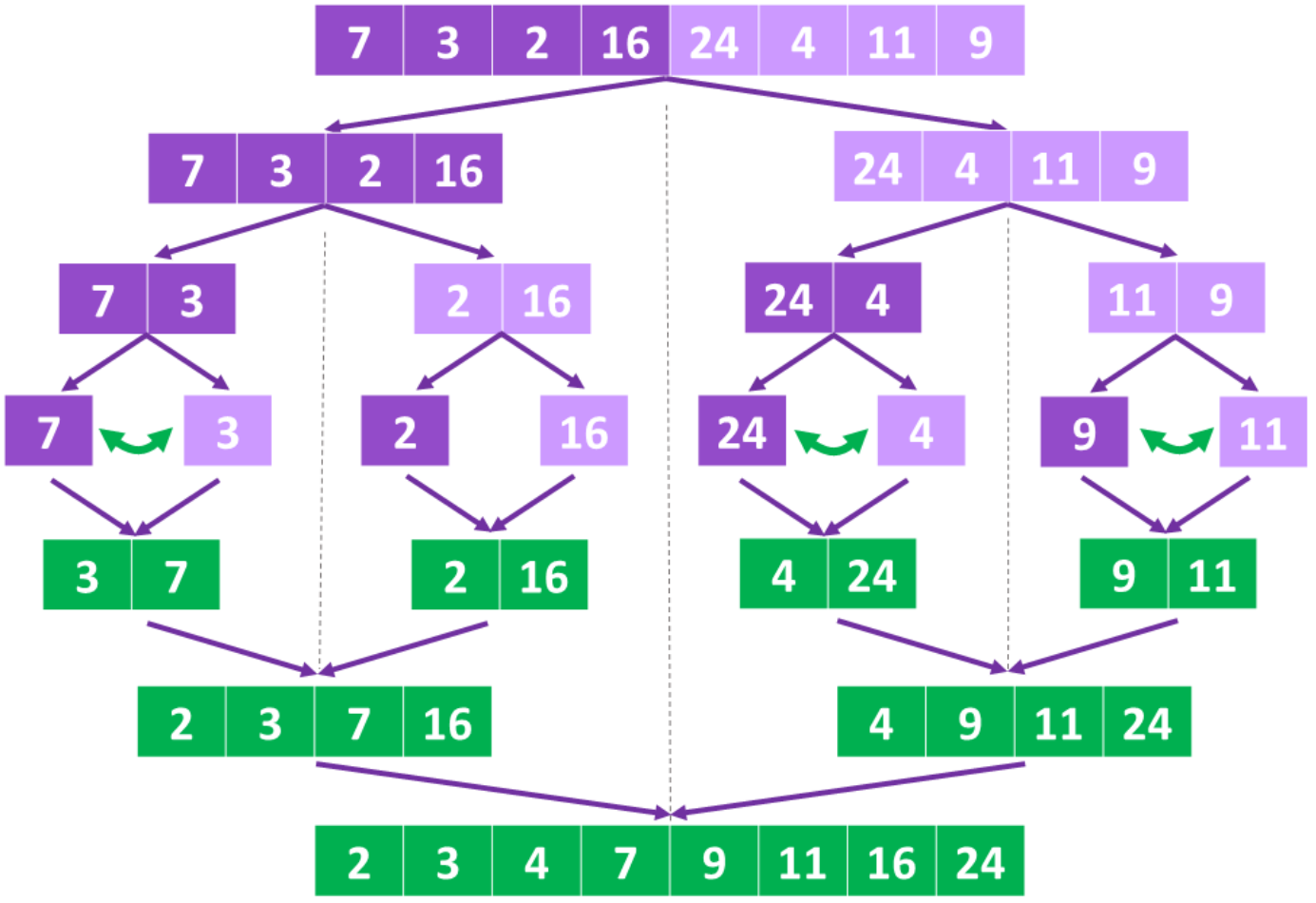
Técnicas de análise de algoritmos

Comando	Custo
Laços de repetição	*Soma do tempo da sequencia dentro do laço mais a condição de parada Multiplicado pelo número de iterações*
Procedimentos não recursivos	*Deve ser computado separadamente de acordo com as avaliações anteriores*

Classes de Problemas

Classe	Nome	Situação
$f(n) = O(1)$	*Constante*	As instruções do algoritmo são executadas um n.º fixo de vezes.
$f(n) = O(\log n)$	*Logarítmica*	Algoritmos que resolvem um problema transformando-os em tempos menores.
$f(n) = O(n)$	*Linear*	Em geral, um pequeno trabalho é realizado sobre cada elemento da entrada.
$f(n) = O(n \log n)$	*n log n*	Um problema é transformado em problemas menores, depois, cada um é resolvido independentemente e, depois, juntam-se as soluções.
$f(n) = O(n^2)$	*Quadrática*	Os itens de dados são processados aos pares, em geral, um laço dentro do outro.
$f(n) = O(n^3)$	*Cúbica*	Úteis para resolver problemas pequenos.
$f(n) = O(2^n)$	*Exponencial*	Uso de força bruta aplicado a um problema.
$f(n) = O(n!)$	*Fatorial*	Uso de força bruta aplicado a um problema.

Ordenação de um vetor: Mergesort

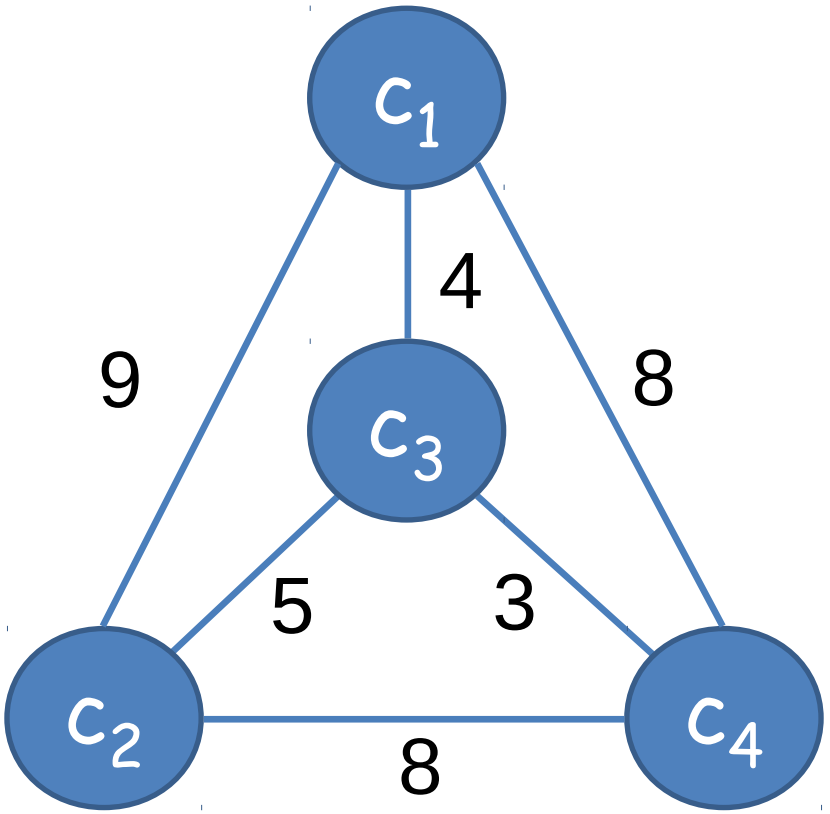


Divida sublistas em 2 até alcançar pares de valores

Ordene os pares de valores se necessário

Junte e ordene as sublistas e repita o processo até juntar a lista completa

Problema do Caixeiro Viajante



- Visitar ***n*** cidades iniciando e terminando em uma mesma cidade, e cada cidade deve ser visitada apenas uma vez.

- **menor percurso:**

  c_1, c_3, c_4, c_2, c_1

- **E se fossem 50 cidades?**
  São 50! possibilidades, que é aproximadamente 10^{64}

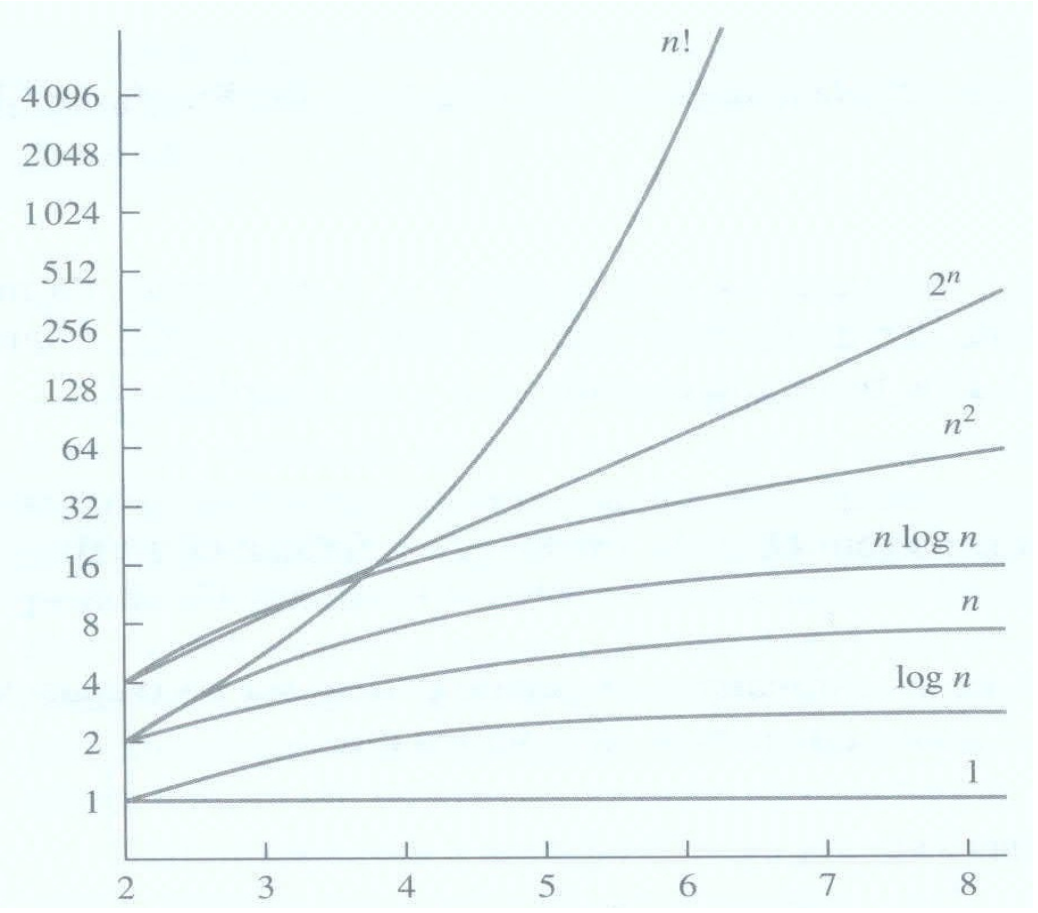
Classes de algoritmos e medidas de tempo

Classe		**N.º de Complexidade de Operações e Tempos de Execução (1 instr/µseg)**					
n		**10**		**10^2**		**10^3**	
Constante	*O(1)*	*1*	*1 µseg*	*1*	*1 µseg*	*1*	*1 µseg*
Logarítimo	*O(log n)*	*3,32*	*3 µseg*	*6,64*	*7 µseg*	*9,97*	*10 µseg*
Linear	*O(n)*	*10*	*10 µseg*	*10^2*	*100 µseg*	*10^3*	*1 mseg*
n log n	*O(n log n)*	*33,2*	*33 µseg*	*664*	*664 µseg*	*9970*	*10 mseg*
Quadrático	*O(n^2)*	*10^2*	*100 µseg*	*10^4*	*10 mseg*	*10^6*	*1 seg*
Cúbico	*O(n^3)*	*10^3*	*1 mseg*	*10^6*	*1 seg*	*10^9*	*16,7 min*
Exponencial	*O(2^n)*	*1024*	*10 mseg*	*10^{30}*	*3,17 * 10^{17} anos*	*10^{301}*	∞

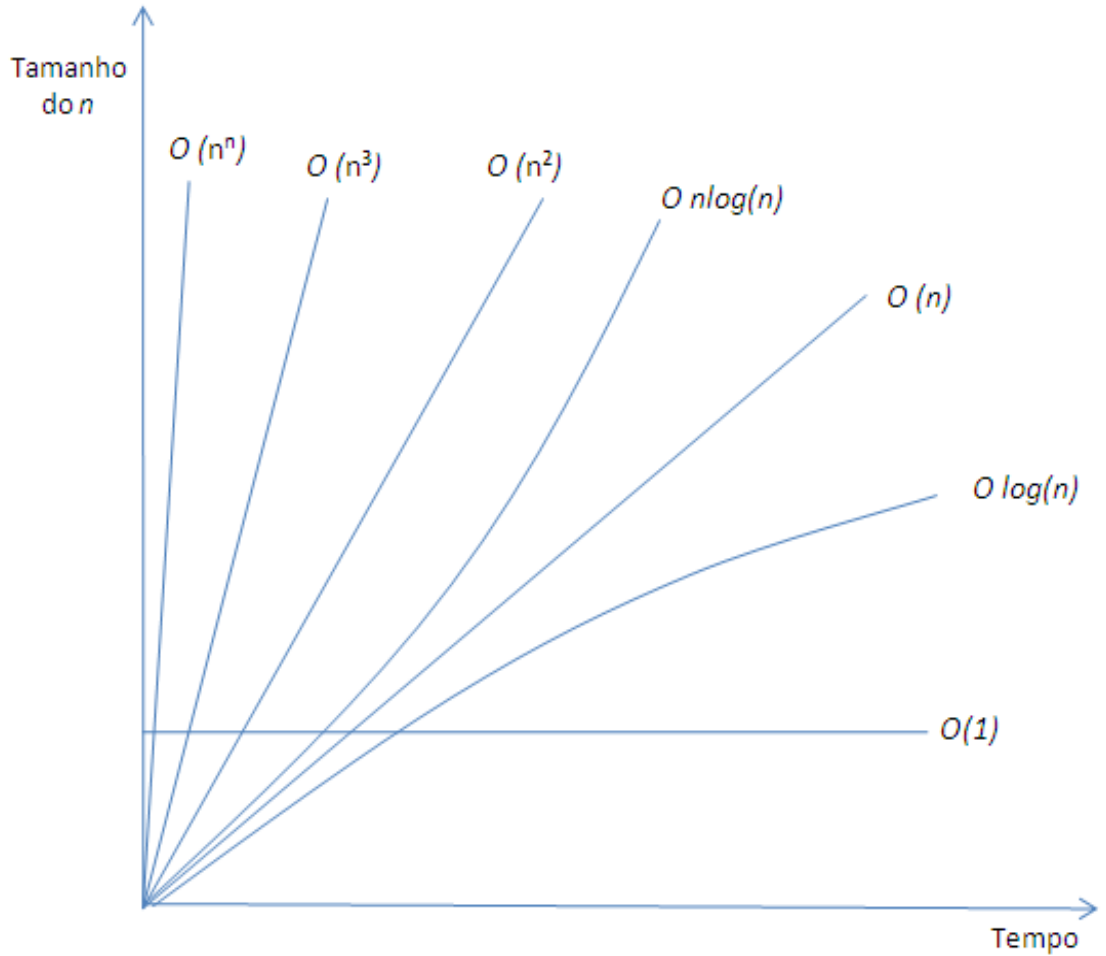
Classes de algoritmos e medidas de tempo

Classe		**N.º de Complexidade de Operações e Tempos de Execução (1 instr/μseg)**					
n		**10^4**		**10^5**		**10^6**	
Constante	*O(1)*	*1*	*1 μseg*	*1*	*1 μseg*	*1*	*1 μseg*
Logarítimo	*O(log n)*	*13,3*	*13 μseg*	*16,6*	*7 μseg*	*19,93*	*20 μseg*
Linear	*O(n)*	*10^4*	*10 mseg*	*10^5*	*0,1 seg*	*10^6*	*1 seg*
n log n	*O(n log n)*	*$133*10^3$*	*133 mseg*	*$166*10^4$*	*1,6 seg*	*$199,3*10^5$*	*20 seg*
Quadrático	*O(n^2)*	*10^8*	*1,7 min*	*10^{10}*	*16,7 min*	*10^{12}*	*11,6 dias*
Cúbico	*O(n^3)*	*10^{12}*	*11,6 dias*	*10^{15}*	*31,7 anos*	*10^{18}*	*31,709 anos*
Exponencial	*O(2^n)*	*10^{3010}*	∞	*10^{30103}*	∞	*10^{301030}*	∞

Funções Típicas e estimativas O-grande



Funções Típicas e estimativas *O*-grande



Referências bibliográficas

Observação: o conteúdo dos slides foram extraídos e montados a partir das referências bibliográficas.

- SIPSER, Michael. **Introdução a teoria da computação**. São Paulo: Editora Thomson, 2007.

  Capítulo 7: introdução à complexidade de tempo –pág. 261 – 268.

- ZIVIANI, Nívio. **Projeto de algoritmos**: com implementação em Java e C++. 2a. ed. São Paulo: Pioneira, 2006.

  Capítulo 1: medida de tempo de execução, comportamento assintótico, classes assintóticas, técnicas de análise de algoritmos.

- CORMEN, Thomas H. et al. **Algoritmos: Teoria e Prática**. Rio de Janeiro: Campus, 2002.

  Capítulo 3 e 4: crescimento de funções (pág. 32) e recorrências (pág. 50)

- DROZDEK, Adam. **Estrutura de Dados e Algoritmos em C++**. São Paulo: Cengage, 2002.

  Capítulo 2: exemplos de análise de algoritmos – páginas 54 – 57.

- ROSEN, Kenneth H. **Matemática discreta e suas aplicações**. 6a. ed. Porto Alegre: McGraw-Hill, 2009.

  Capítulo 3: revisão sobre somatórios – páginas 149 – 158.