Introdução ao Estudo de Algoritmos

Algoritmos e Estruturas de Dados

Prof. Dr. Luciano Demétrio Santos Pacífico
{luciano.pacifico@ufrpe.br}

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Conteúdo

- **Introdução**
- **Notação Algorítmica (Pseudolinguagem)**
- **Análise de Complexidade**
- **Análise Assintótica**

Introdução

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Definição de Algoritmo

- Um **algoritmo** pode ser definido como uma sequência bem definida de passos ordenados, que recebe um valor ou conjunto de valores (dados) como **entrada**, e tem como resultado um valor ou conjunto de valores de **saída**.

- Um algoritmo é dito **correto** se, para cada instância de entrada existente, o mesmo encerra sua execução com um valor de saída correto.

Exemplo

- Desenvolver um algoritmo para fritar um ovo.
- Entradas: ovo não-frito, fogão, frigideira, ...
- Saídas: ovo frito.
  - Untar frigideira;
  - Ligar o fogão;
  - Quebrar casca do ovo não-frito;
  - ...

Exemplo

- O exemplo anterior mostra que alguns problemas podem ser resolvidos por sequências de passos diferentes
  - Pode optar-se por ligar o fogão antes de untar a frigideira.

- Em alguns problemas, o intercâmbio de duas ou mais etapas de sua solução pode gerar **incorretude** do algoritmo, ou soluções **não ótimas**.
  - Ligar o fogão antes de untar a frigideira pode levar ao mesmo resultado (ovo frito), mas se o consumo de gás for levado em consideração no projeto do algoritmo, o algoritmo gerado por esta solução é não ótimo.

Uso de Algoritmos

- Em computação, os algoritmos são utilizados para solucionar os mais diversos tipos de problemas práticos, em áreas como:
  - Biologia (ex.: Análise de sequências de DNA);
  - Medicina (ex.: Detecção de tumores em imagens);
  - Artes (ex.: Efeitos visuais);
  - Ciências Sociais (ex.: Análise de dados populacionais);
  - ...
- Os algoritmos (computacionais ou não) estão em todos os lugares.

Notação Algorítmica (Pseudolinguagem)

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO

Notação Algorítmica (Pseudolinguagem)

- Para o estudo de algoritmos, precisamos definir uma forma de **notação básica**.

- Essa notação deve contemplar apenas **aspectos de raciocínio lógico**, não se limitando a uma **linguagem de programação** específica.

- Porém, essa linguagem tem de ser **facilmente adaptável** à codificação em linguagens de programação.

- Geralmente, usa-se uma linguagem próxima à linguagem natural (**pseudolinguagem**), porém sem as ambiguidades presentes em uma linguagem natural.

- A Pseudolinguagem deve ser próxima o suficiente de uma **codificação em uma linguagem de programação real**... Mas nem tanto, e, ao mesmo tempo, **facilmente compreensível por um ser humano**, como uma linguagem natural... Mas nem tanto.

Visão Geral

- A pseudolinguagem adotada na disciplina se aproximará de linguagens de programação como as linguagens **C**, **Java** e **Python**.

- Uma visão geral da pseudolinguagem será apresentada nos próximos slides, porém as aulas futuras poderão trazer alguma representação nova para a pseudolingaugem.

- Não há necessidade de decorar a pseudolinguagem proposta, pois haverá flexibilidade.

- **Cuidado**: Não confundir **flexibilidade** com **ambiguidade**.
  - Vamos ver como isso vai funcionar, mas em caso de dúvidas, entrar em contato com o professor da disciplina.
  - Ambiguidades serão consideradas **erro**.

Tipos e Variáveis

- A pseudolinguagem será **fracamente tipada**, ou seja, não há necessidade da declaração do tipo de variáveis e constantes, como na linguagem **Python**.

- Porém, deve-se ter em mente que ao se atribuir um valor a uma variável/constante, a mesma terá um **tipo implicitamente definido**.

- **Tipos primitivos** da pseudolinguagem: **inteiro**, **caractere**, **real** e **booleano**.

- A pseudolinguagem será ***Case Sensitive*** (diferencia letras maiúsculas de minúsculas na declaração de nomes).

- O símbolo "**=**" representará o comando de **atribuição**.

Tipos e Variáveis

- Nos exemplos presentes nos slides de aula, as linhas de código estarão numeradas, no intuito de facilitar a leitura e entendimento, assim como para possíveis referências ao longo da explicação dos códigos.

- Exemplo:

```
1. contador = 1
```

- Após a execução do código acima, a variável `contador` armazenará o valor “1”, sendo considerada do tipo **inteiro**.

Tipos e Variáveis

- Exemplos de **Flexibilização (permitido fazer)**:

  - Finalizar comandos com o símbolo "**;**" (como em **C**, **Java**, etc.):

```
1. contador = 1;
```

  - **Declarar tipo ao declarar variável/constante** (como em **C**, **Java**, etc.)...

```
1. inteiro contador = 1
```

  - ... Ou ainda, com o uso do símbolo ":" e tipo após o nome da variável (como em **Kotlin**)...

```
1. contador:inteiro = 1
```

  - ... Ou ainda algumas combinações:

```
1. contador:inteiro = 1;
```

    - Ou

```
1. inteiro contador = 1;
```

Tipos e Variáveis

- **Ambiguidades (Proibido)**:
  - Adotar mais de um estilo em um mesmo código (exemplo: finalizar algumas linhas com ";" e outras não; declarar variáveis com <nome>:<tipo>, e em seguida com <tipo> <nome>).
  - Adotar o estilo de declaração de tipo na criação da variável/constante, e em seguida tentar mudar o tipo da variável.
    - Se estiver usando o estilo de atribuição de tipo ao declarar a variável ou constante, **não será permitido mudar o tipo da variável em seguida** (como em **linguagens fortemente tipadas**).
    - Variáveis que **não tenham tido um tipo atribuído em suas declarações** poderão **mudar de tipo** normalmente.
- Dica: Mantenha um mesmo estilo em todos os códigos que escrever ao longo da disciplina.

Expressões

- A pseudolinguagem conterá os mesmos tipos de expressões comumente encontrados em outras linguagens de alto nível.

- Os símbolos serão os mesmos.

- Tipos:
  - Expressões Aritméticas: +, -, *, /, % (resto inteiro);
  - Expressões Lógico-Relacionais: **e**, **ou**, **xor** (para “ou exclusivo”), ! (negação/inversão), >, >=, <, <=, == (comparação de igualdade), != (comparação de diferença).

- **Flexibilização**:
  - Pode-se optar por usar “&&”, “||”, “^”, no lugar de **e**, **ou**, ou **xor**, respectivamente.

Estruturas Condicionais

- Alteram o fluxo de execução dos comandos.
- Decidem quais comandos serão executados a partir de uma condição estabelecida.
- Condição representada por **expressão lógica**.
- Na pseudolinguagem:

se <condição>
    <comandos_1>
senão
    <comandos_2>

se <condição>
    <comandos_1>
senão se <condição>
    <comandos_2>

- Blocos de código serão delimitados por **indentação** (com em **Python**).
- **Flexibilização**:
  - Pode-se usar **chaves** “{ }” para delimitar blocos de código.

Estruturas de Repetição

- Sequência de comandos executadas repetidamente até que a condição de interrupção seja satisfeita.

- Número **indefinido** de repetições:

```
enquanto <condição>
    <comandos_1>
//Fora do corpo do laço de repetição
<comandos_2>
```

- Quando o número de repetições é **definido**, usa-se o comando abaixo:

```
para <cont> = <v-ini> até <v-fim> passo <d>
    <comandos_1>
//Fora do corpo do laço
<comandos_2>
```

Sub-Rotinas

- As **sub-rotinas** (funções, métodos, procedimentos, etc.) serão declaradas pelo uso da palavra reservada **procedimento**.

Ex.:

```
1. procedimento ola_mundo()
2.     imprimir(“Olá, Mundo!”)
```

- Variáveis declaradas em procedimentos terão **escopo local**.

- Variáveis declaradas foram de procedimentos terão **escopo global**.

Sub-Rotinas – Parâmetros de Entrada e Valores de Saída

- Um procedimento pode ter **nenhum** (como a função `ola_mundo`), **um** (como a função `imprimir`) ou **vários parâmetros de entrada**.

- Parâmetros podem ser declarados com **valores *default*** (com em **Python**).

- Parâmetros aos quais **não são declarados valores *default*** precisam necessariamente ter **valores atribuídos** quando o procedimento for chamado.

- Um procedimento pode retornar **no máximo um valor**, indicado pelo uso da palavra reservara **`retorne`**.

Estruturas de Dados Homogêneas: Arrays

- As Estruturas de Dados e Algoritmos mais simples que serão vistos na disciplina poderão ser representados através de **Arranjos Homogêneas** (ou seja, que armazenam **um único tipo de dado**), que poderão ter **uma ou múltiplas dimensões**.

- Os arranjos poderão ser declarados através do uso da palavra reservada **`Array`**.

- Os arranjos serão **Estruturas de Dados Estáticas**, cujo **tamanho** (quantidade **máxima** de elementos) deve ser fornecido no momento de sua declaração.

- A **indexação** de arranjos será dada **de 1 até *n***, sendo *n* o tamanho do arranho.

- Ex.:

```
1. //Declaração de Arranjo com capacidade 10
2. lista_numeros = Array[10]
```

Estruturas de Dados Homogêneas: Arrays

- Um tipo especial de arranjo será o de **cadeia de caracteres**, declarado pela palavra reservada **`String`**.

- Arranjos podem ter várias dimensões.

```
1. //Declaração de Arranjo bidimensional
2. matriz = Array[10][5]
3. //Declaração de Arranjo tridimensional
4. imagem_colorida = Array[128][128][3]
```

- É possível declarar um tipo para o arranjo.

```
1. //Declaração de Arranjo tipado
2. matriz_numerica = Array<real>[10][5]
```

Estruturas de Dados Heterogêneas

- Registros (Classes, Estruturas) serão declarados através da palavra-chave **`registro`**, seguida de suas propriedades, separadas por ‘,’ (**vírgula**).

```
registro Aluno
      matricula,
      nome:String,
      cpf
```

- Propriedades de um registro são acessadas através da **sintaxe de ponto**.

```
//Construtor default não recebe parâmetros
aluno = Aluno()
aluno.nome = "Pedro"
```

Exemplo

$\|\vec{x}\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$ (norma-2 de um vetor numérico)

```
//x -> vetor, n -> tamanho do vetor
procedimento norma2(x, n)
    soma = 0
    para i = 1 até n //passo pode ser omitido quando for 1
        soma = soma + x[i]*x[i]
    retorne sqrt(soma) //sqrt é uma função pré-definida (raiz quadrada)
```

Análise de Complexidade

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Análise de Complexidade

- Análise de Complexidade de Algoritmos
  - Tenta prever os recursos requeridos por um algoritmo durante sua execução.
  - Ex.: Memória, alocação de banda para comunicação, dispositivos de hardware, tempo de processamento.

Análise de Complexidade

- Por que é importante analisar a complexidade de algoritmos?

  - A complexidade é um dos principais fatores durante o projeto de novos algoritmos.

    - Outros fatores: corretude, eficiência, tolerância à falhas, etc.

  - Quando um problema pode ser resolvido por mais de uma técnica, a análise de complexidade fornece uma medida para que se possa decidir qual técnica é a mais adequada para o problema em mãos.

Análise de Complexidade

- A complexidade algorítmica pode ser dividida em **complexidade de tempo** e **complexidade de espaço de memória** (ou **complexidade de espaço**).

  - Pode-se usar como medida de complexidade de tempo o tempo absoluto de execução (em segundos, minutos, etc.), porém tal atitude torna a análise dependente da máquina na qual os experimentos são executados.
    - Ex.: Comparar o tempo absoluto de uma técnica que foi executada em um computador com processador de 512Ghz, memória RAM de 1TB com outra técnica executada em um computador com processador de 4Mhz e 64KB de memória RAM não parece exatamente justo...

Análise de Complexidade

- Para evitar tal restrição, a análise é feita levando em consideração o número de operações primitivas ou passos considerados relevantes realizados pelo algoritmo durante sua execução sobre uma entrada genérica de tamanho *n*.

- O tempo de execução *T* de um algoritmo passa a ser visto como a soma total do tempo de execução de cada uma de suas etapas.

Análise de Complexidade

- Exemplo: Supondo que três algoritmos sejam capazes de resolver o mesmo problema, avaliar qual o mais custoso:

  - $f_1(n) = 5n^2 + 10n$ operações.
  - $f_2(n) = \frac{2^n}{500} + \frac{1000}{n}$ operações.
  - $f_3(n) = 300n + 5000$ operações.

    - Qual é o mais custoso?

Pior Caso, Caso Médio e Melhor Caso

- O tempo de execução de um algoritmo está fortemente relacionado à forma de apresentação da instância do problema fornecida como entrada para o mesmo.
  - Um algoritmo de ordenação crescente vai executar bem mais rápido se a entrada já estiver em ordem crescente ☺ (**melhor caso**)

- Geralmente, investiga-se o desempenho do algoritmo em relação ao seu **pior caso** (**análise de pior caso**).
  - Um algoritmo de ordenação crescente recebe como entrada um vetor de dados ordenado de forma decrescente ☹

Insertion-Sort

- Vamos analisar a execução do procedimento abaixo:

`//A -> vetor, n -> tamanho do vetor`		
procedimento `insertion_sort(A, n)`	// custo	execuções
para `j = 2` **até** `n`	// c_1	n
`chave = A[j]`	// c_2	$n-1$
`i = j - 1`	// c_3	$n-1$
enquanto `(i > 0)` **e** `(A[i] > chave)`	// c_4	$\sum_{j=2}^{n} t_j$
`A[i+1] = A[i]`	// c_5	$\sum_{j=2}^{n} (t_j - 1)$
`i = i - 1`	// c_6	$\sum_{j=2}^{n} (t_j - 1)$
`A[i + 1] = chave`	// c_7	$n-1$

Exemplo *Insertion-Sort*

- A = [5, 2, 4, 6, 1, 3]

A	1	2	3	4	5	6
Nº trocas = 1	5	**2***	4	6	1	3
Nº trocas = 1	2	5	**4***	6	1	3
Nº trocas = 0	2	4	5	**6***	1	3
Nº trocas = 4	2	4	5	6	**1***	3
Nº trocas = 3	1	2	4	5	6	**3***
Nº trocas = 0	1	2	3	4	5	6

Pior Caso, Caso Médio e Melhor Caso

- Tempo de execução do *Insertion-Sort*:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j$$

$$+c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

- No pior caso, $t_j = j$:

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

Pior Caso, Caso Médio e Melhor Caso

- Tempo de execução do *Insertion-Sort* (cont.):

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1)$$

$$+c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1)$$

$$T(n) = \left(\frac{c_4+c_5+c_6}{2}\right)n^2 + (c_1 + c_2 + c_3 + c_7 + \frac{c_4-c_5-c_6}{2})n\text{-}(c_1 + c_3 + c_4 + c_7)$$

- Concluindo: o tempo de execução do *Insertion-Sort* no pior caso é da ordem de $\Theta(n^2)$.

Pior Caso, Caso Médio e Melhor Caso

- Por quê fazer a análise do pior caso é mais interessante?

  - O tempo de execução no pior caso nos dá uma noção do tempo máximo necessário para que o algoritmo execute com qualquer entrada de tamanho *n*.

  - A frequência de ocorrência do pior caso é muito grande para determinados algoritmos. Ex.: na busca por uma entrada inexistente em um banco de dados.

Pior Caso, Caso Médio e Melhor Caso

- Complexidade de caso médio:
  - É bastante usado quando há uma suposição inicial da distribuição dos dados de entrada.
    - Ex.: Análise probabilística.
  - Quando existe algum fator aleatório em alguma etapa do algoritmo, também é comum o uso da análise do caso médio para a obtenção do tempo de execução esperado.

Pior Caso, Caso Médio e Melhor Caso

- Observação: geralmente, o caso médio pode ser tão custoso quanto o pior caso.

  - Ex.: no *Insertion-Sort*, se metade dos dados estivessem ordenados em ordem crescente e outra metade em ordem decrescente, teríamos $t_j = j/2$, o que nos daria a mesma complexidade de pior caso.

Análise Assintótica

Análise Assintótica

- Taxa de crescimento ou ordem de crescimento do tempo de execução: dá uma caracterização simplificada da eficiência de um algoritmo.

- A análise assintótica estuda a curva de crescimento da complexidade de um algoritmo em decorrência do tamanho da entrada fornecida para o mesmo, ou seja, o quanto o tempo de execução aumenta com o tamanho da entrada.

Análise Assintótica

- Apenas o termo de maior ordem é considerado, sem seus coeficientes constantes, tendo em vista que quando a entrada for muito grande os termos de menor ordem tornam-se irrelevantes no custo computacional final.
  - Ex.: $f(n) = 2n^9 + 500n^4 + 1000n^2 + 89n = O(n^9)$.
- Um método é considerado melhor que outro se sua curva de crescimento é menor de acordo com o aumento do tamanho da entrada fornecida.

Análise Assintótica

- Notação Θ

- Dada uma função *g*(*n*), denota-se por Θ(*g*(*n*)) ao conjunto de funções que:
  - {*f*(*n*): existem constantes positivas c_1, c_2 e n_0 tais que 0 ≤c_1*g*(*n*)≤*f*(*n*) ≤c_2*g*(*n*), para todo *n* ≥ n_0}

- Ou seja, para valores grandes de *n*, *f*(*n*) vai ter seus valores limitados entre c_1 *g*(*n*) e c_2 *g*(*n*), c_1 e c_2 positivos.

- Neste caso, diz-se que *f*(*n*) = Θ(*g*(*n*)) ou $f(n) \in \Theta(g(n))$.
  - Ex.: $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

Análise Assintótica

- Notação O

- A **notação Θ** limita assintoticamente uma função **inferiormente e superiormente**.

- Na **notação O**, é oferecido apenas um **limitante superior** à função.

- Dada uma função $g(n)$, denota-se por $O(g(n))$ ao conjunto de funções que:
  - $\{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$

- Diz-se que $f(n) = O(g(n))$ ou $f(n) \in O(g(n))$.

Análise Assintótica

- Notação Ω

- Oferece um **limite inferior** assintótico para a função.

- Dada uma função *g*(*n*), denota-se por Ω(*g*(*n*)) ao conjunto de funções que:
  - {*f*(*n*): existem constantes positivas *c* e n_0 tais que 0≤*cg*(*n*)≤*f*(*n*), para todo *n* ≥n_0}

- Diz-se que *f*(*n*) = Ω(*g*(*n*)) ou $f(n) \in \Omega(g(n))$.

Análise Assintótica

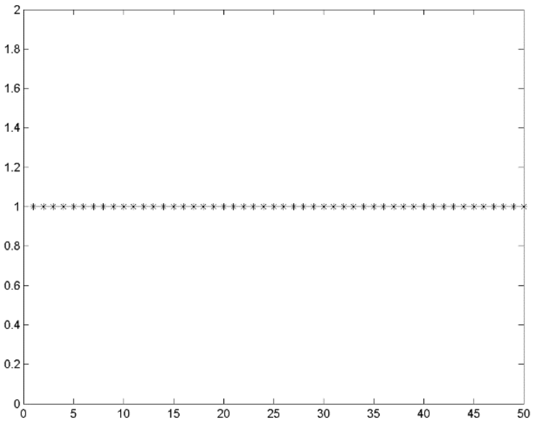
Para duas funções $f(n)$ e $g(n)$, temos que $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$

Análise Assintótica

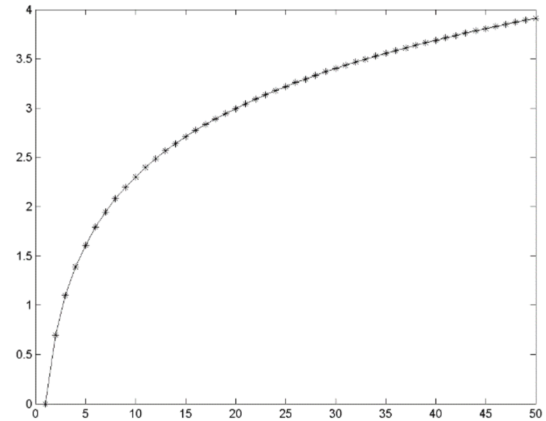
CLASSE	NOME
$O(1)$	Constante
$O(\log n)$	Logarítmica
$O(n)$	Linear
$O(n \log n)$	$n \log n$
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(n^k)$, $k \geq 1$	Polinomial
$O(2^n)$	Exponencial
$O(a^n)$, $a > 1$	Exponencial

Análise Assintótica

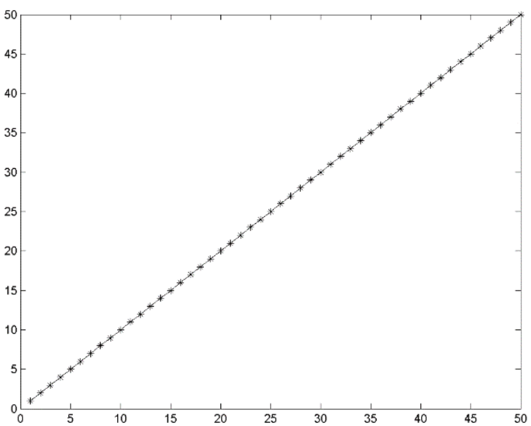
O(1)



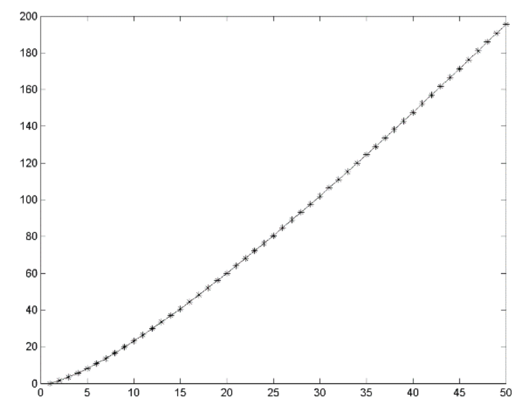
O(log *n*)



O(*n*)

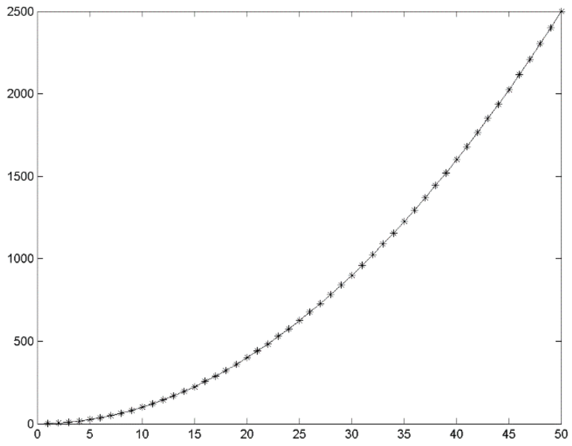


O(*n* log *n*)

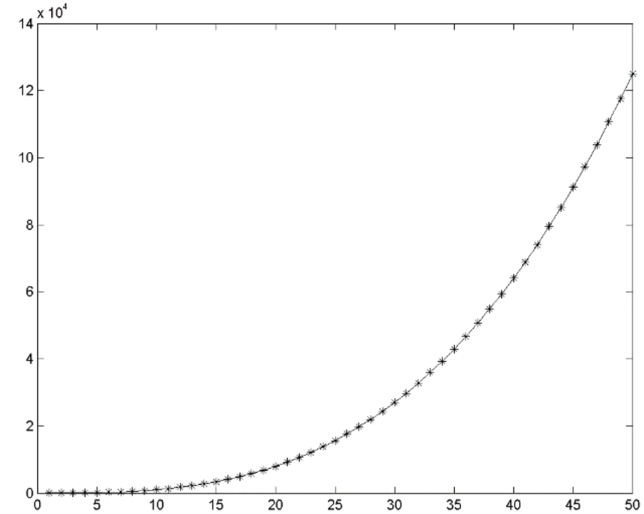


Análise Assintótica

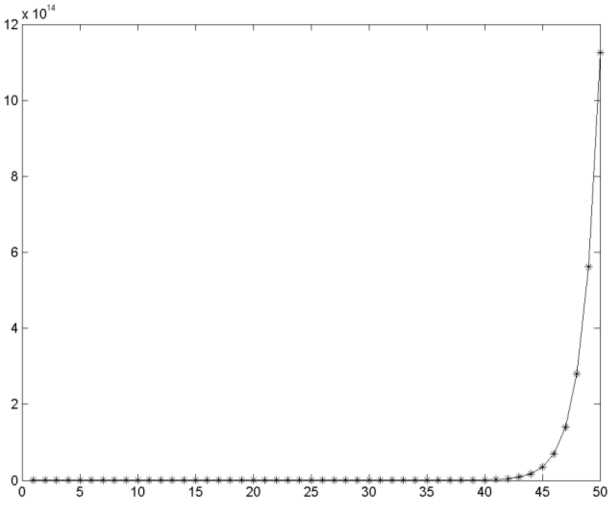
$O(n^2)$



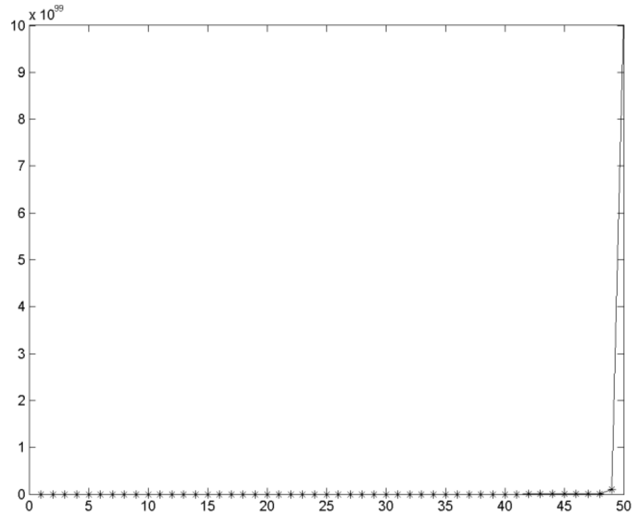
$O(n^3)$



$O(2^n)$



$O(a^n)$



Referências

- CORMEN, H. T.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Introduction to Algorithms, 3rd ed., *Boston: MIT Press*, 2009.

- FEOFILOFF, Paulo. Algoritmos em Linguagem C. Editora Campus/Elsevier, 2009.

Introdução ao Estudo de Algoritmos

Algoritmos e Estruturas de Dados

Prof. Dr. Luciano Demétrio Santos Pacífico
{luciano.pacifico@ufrpe.br}

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO