Fundamentos da Análise de Algoritmos

Pesquisa, Ordenação e Técnicas de Armazenamento

Prof. Msc. Bruno de A. Iizuka Moritani

bruno.moritani@anhembi.br

Agenda

- Revisão
  - Recursividade
- Análise de Algoritmos
  - Definição
  - Tipos de funções
  - Notação Assintótica

Revisão - Recursividade

- Recursividade
  - 1. Que se pode repetir até ao infinito.
- Uma função é chamada de recursiva, quando ela invoca a si mesma, direta ou indiretamente, uma ou mais vezes para resolver subproblemas correlatos.

Revisão - Recursividade

- Dividir para conquistar;

- É necessário estabelecer pelo menos dois elementos:
  - <u>Uma condição de parada;</u>
  - <u>Uma mudança de estado a cada chamada.</u>

- Uso da pilha para armazenar os valores.

Revisão - Recursividade - Exemplo

```java
public class POTAAula02Exemplo01 {

    public static int soma(int n) {
        if (n == 1) {
            return 1;
        } else {
            return (n + soma(n - 1));
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n;
        System.out.println("Digite um inteiro positivo");
        n = scan.nextInt();
        System.out.println("Soma: " + soma(n));
    }

}
```

Fundamentos da Análise de Algoritmos

Análise de Algoritmos

- O que é analisar um algoritmo?
  - Prever os recursos de que o algoritmo necessitará;
  - Prever desempenho, comparar algoritmos e ajustar parâmetros;
  - Além de memória, largura de banda de comunicação e hardware, algo extremamente importante medir é o tempo de computação, identificando assim qual algoritmo é mais eficiente na resolução de um problema.

Análise de Algoritmos

- O tempo de execução pode ser afetado pelo:
  - Hardware
    - Processador, velocidade do clock, memória, disco...
  - Software
    - Sistema operacional, linguagem de programação, compilador, interpretador...
- Relacionamento entre o tempo de execução de um algoritmo e o tamanho da entrada.

Análise de Algoritmos

- Maneira geral de analisar os tempos de execução de algoritmos que:
  - considera todas as entradas possíveis;
  - permite que se avalie a eficiência relativa de quaisquer dois algoritmos de forma independente dos ambientes de hardware e software;
  - pode ser executada estudando-se descrições de alto nível de algoritmos sem ter de implementá-lo ou executar experimentos.

Análise de Algoritmos

- Modelo de tecnologia de implementação:
  - Um único processador;
  - Utilização da memória RAM;
  - Instruções executadas uma após a outra;
  - Sem operações concorrentes.

Funções

Análise de Algoritmos

- Associar, com cada algoritmo, uma função *f(n)* que caracteriza o tempo de execução como uma função do tamanho da entrada *n*.
- Funções típicas:
  - Função constante
  - Função logaritmo
  - Função linear
  - Função *n*-log-*n*
  - Função quadrática
  - Função cúbica e outras polinomiais
  - Função exponencial
  - Função factorial (completamente inútil)

Função Constante

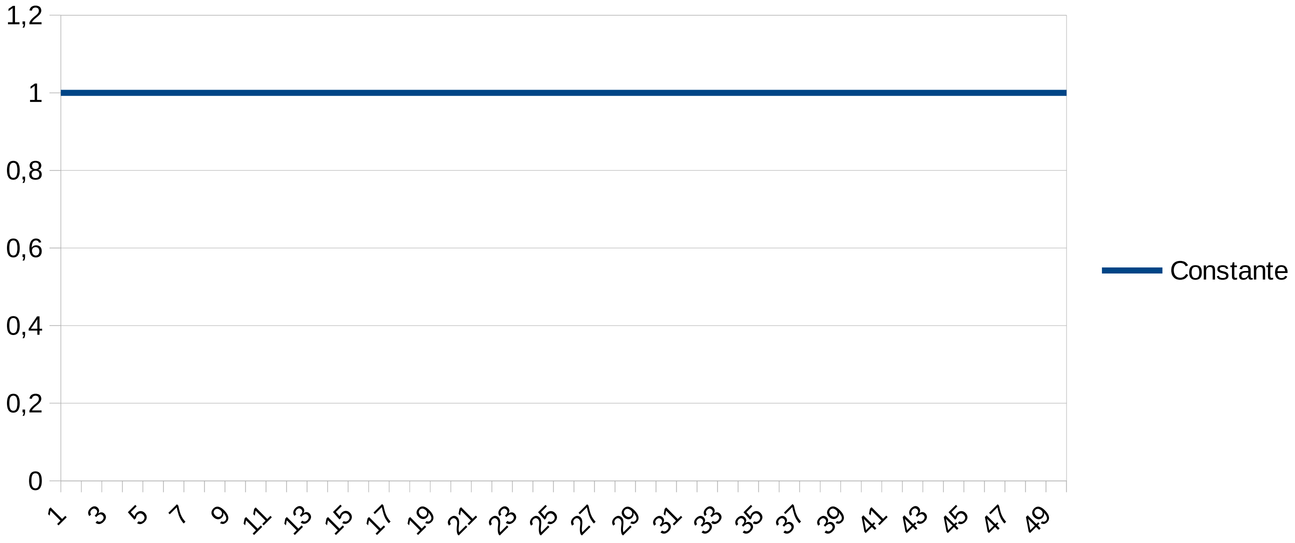
- A função mais simples:

$$f(n) = c$$

- Para qualquer argumento n, a função constante $f(n)$ atribui um valor c;
- $g(n) = 1 \rightarrow$ função constante mais fundamental;
- Qualquer outra função constante, $f(n) = c$, pode ser escrita como:

$$f(n) = cg(n)$$

Função Constante



Função Logaritmo

$$f(n) = \log_b n$$

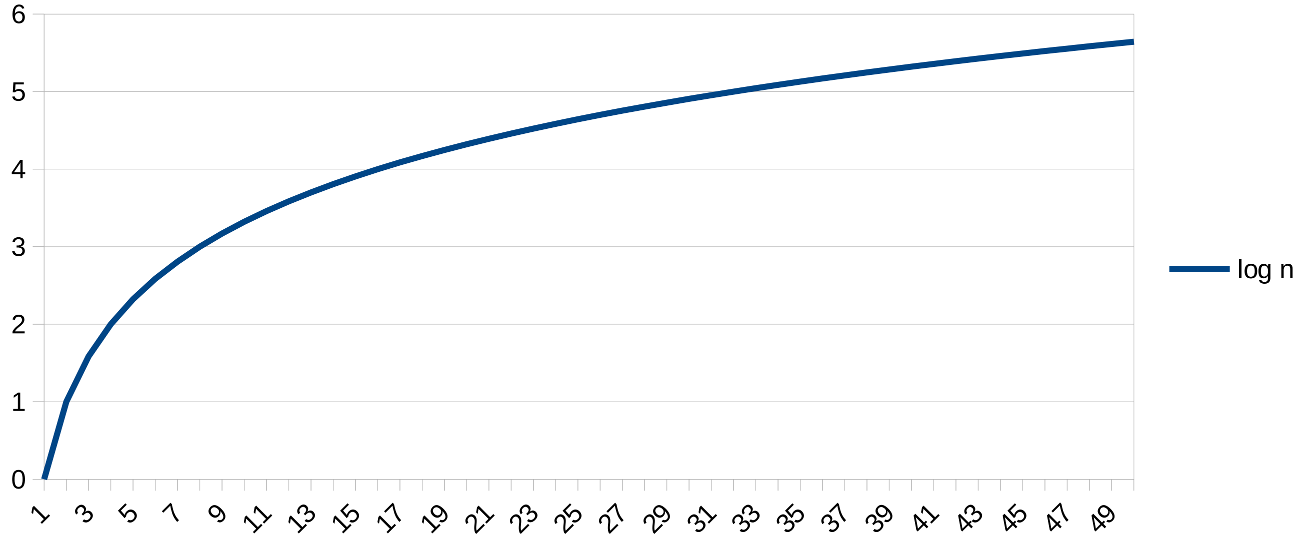
- para alguma constante $b > 1$. Esta função é definida como segue:

$$x = \log_b n \text{ se e somente se } b^x = n$$

- Por definição, $\log_b 1 = 0$.
  - O valor b é conhecido como base do logaritmo.
- A base mais comum para a função logaritmo em Ciência da Computação é 2.

$$\log n = \log_2 n$$

Função Logaritmo

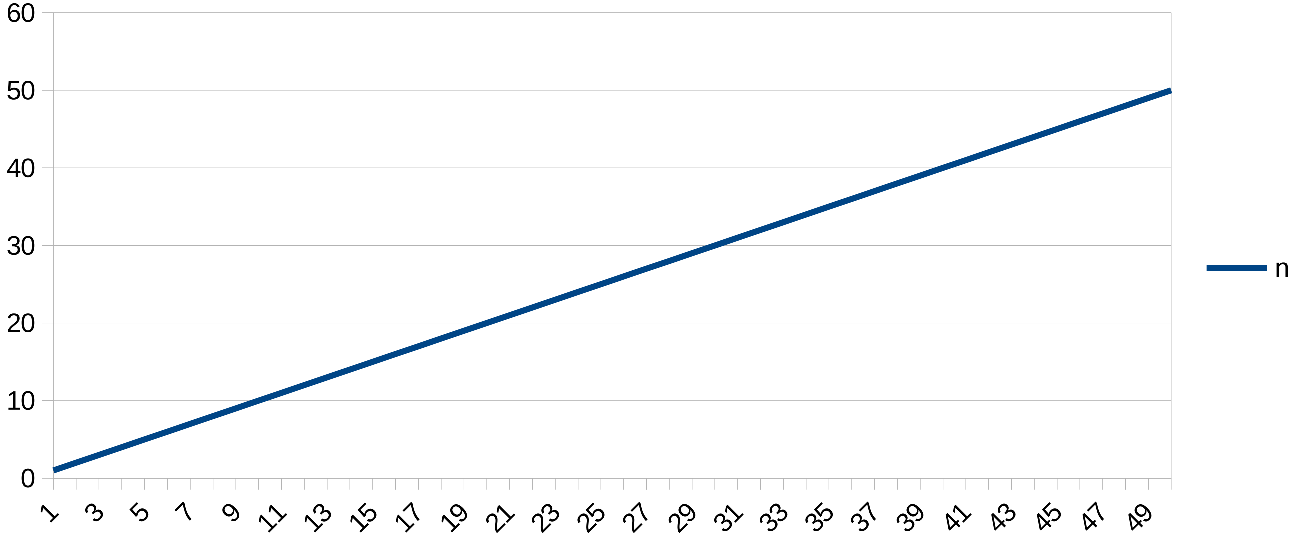


Função Linear

$$f(n) = n$$

- Dado um valor de entrada n, a função linear f atribui o valor n para si mesma.
- Execução em operações básicas sobre cada um de n elementos de um arranjo.

Função Linear

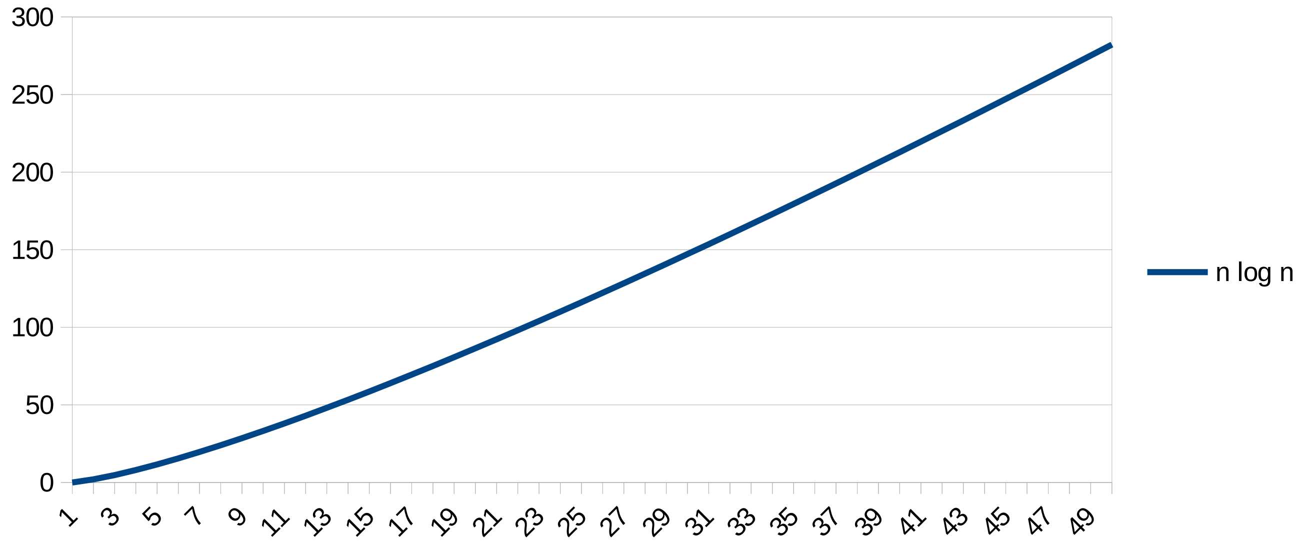


Função n log n

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

- Atribui para uma entrada n o valor de n multiplicado pelo logaritmo de n na base 2.
- Esta função cresce um pouco mais rápido que a função linear e muito mais devagar que a função quadrática.

Função *n* log *n*

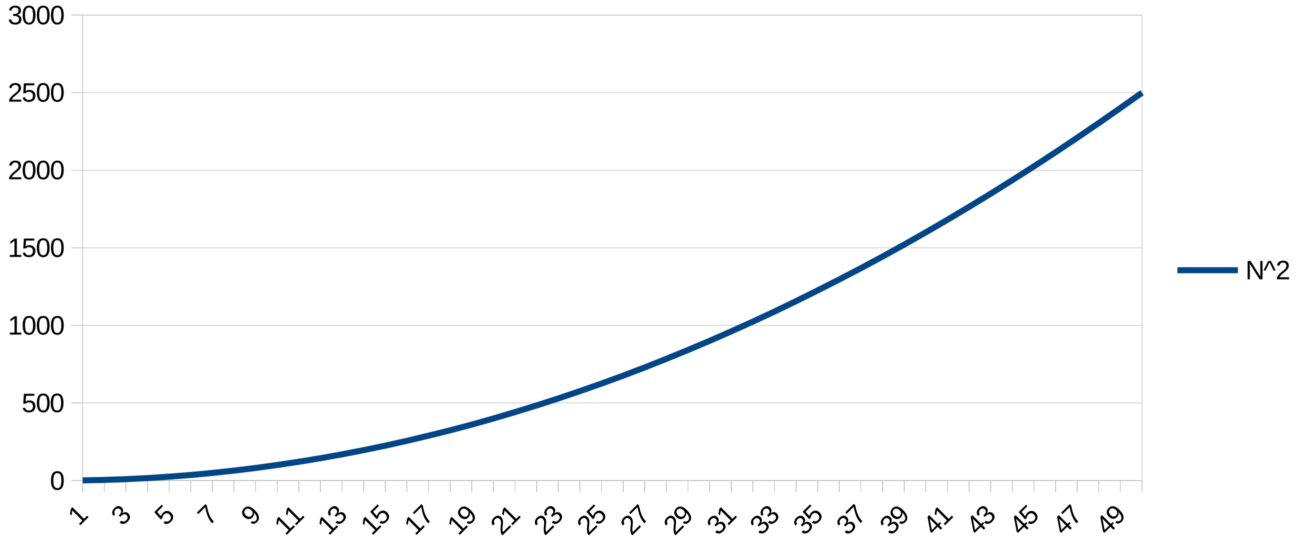


Função Quadrática

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

- Dado um valor de entrada n, atribui o produto de n por si mesmo (ou seja, n ao quadrado = n^2).
- Muito comum em algoritmos que possuem laços aninhados, com execuções de uma quantidade linear de operações em cada laço.

Função Quadrática

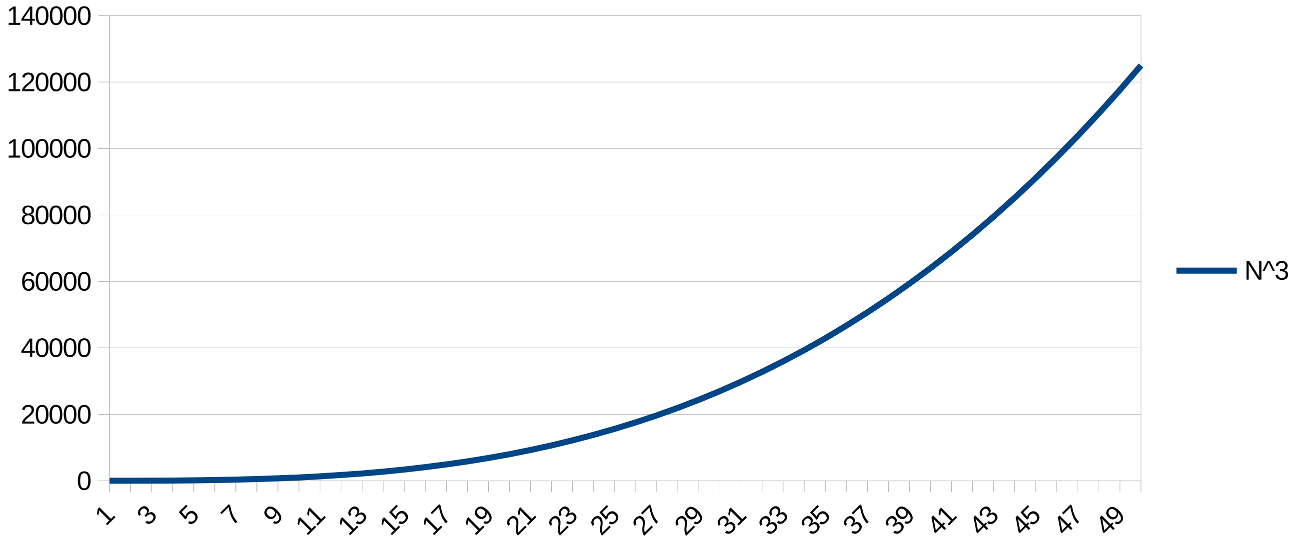


Função Cúbica e outros Polinômios

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

- Atribui um valor de entrada *n* o produto de *n* por ele mesmo três vezes (*n* ao cubo = n^3).
- Todas as funções apresentadas fazem parte da classe de polinômios
- $f\,(n) = \; a_i\, n^i = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_d n^d$

Função Cúbica e outros Polinômios

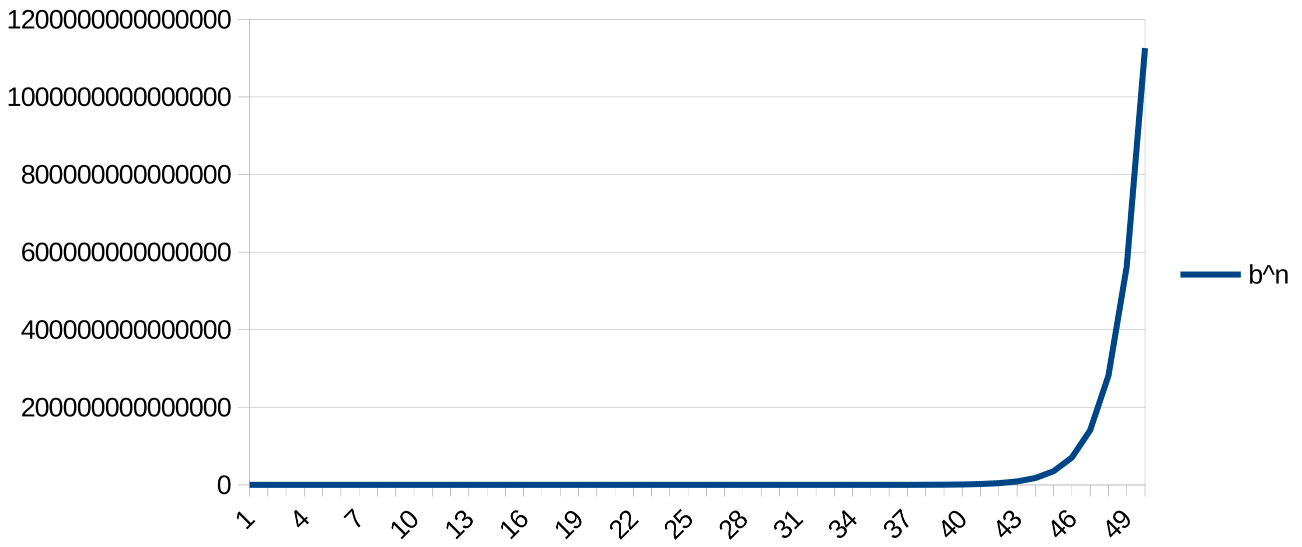


Função Exponencial

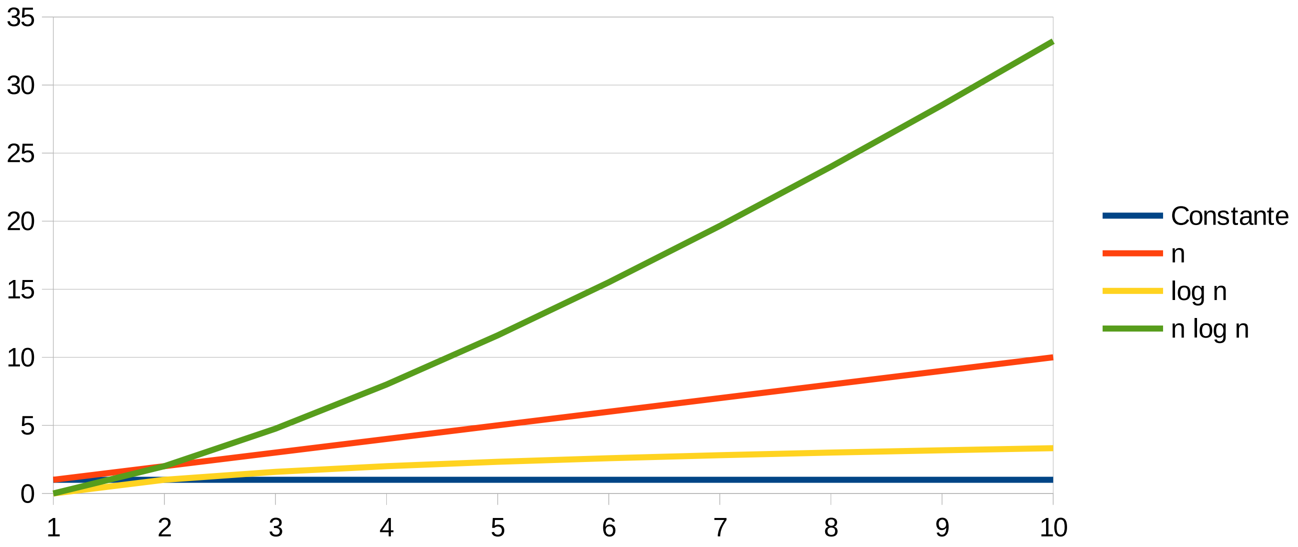
$$f(n) = b^n$$

- onde *b* é uma constante positiva, chamada base, e o argumento *n* é o expoente.
- Atribui ao argumento de entrada *n* o valor obtido pela multiplicação da base *b* por si mesma *n* vezes.
- Na análise de algoritmos, a base mais comum é $b = 2$.

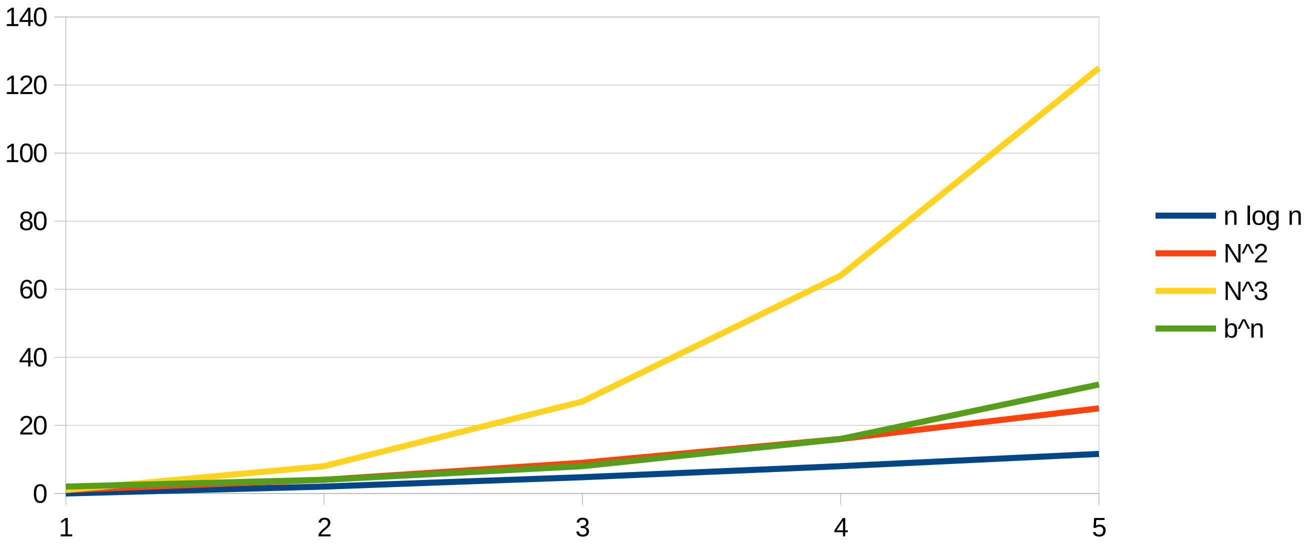
Função Exponencial



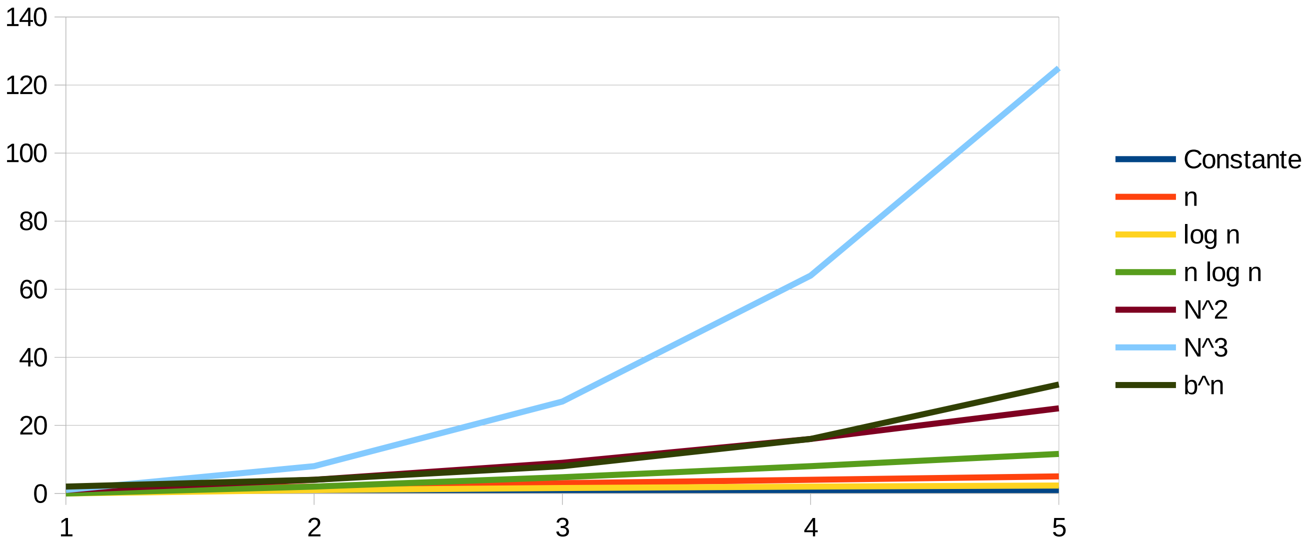
Comparativo entre Funções



Comparativo entre Funções



Comparativo entre Funções



Arredondamento

- A análise de algoritmos pode algumas vezes envolver o uso das funções de arredondamento para cima e arredondamento para baixo para poder expressar o tempo de execução.
  - $\lfloor x \rfloor \rightarrow$ o maior inteiro menor ou igual a x
    - $\lfloor 4,64 \rfloor = 4$
  - $\lceil x \rceil \rightarrow$ o menor inteiro maior ou igual a x
    - $\lceil 4,64 \rceil = 5$

Operações Primitivas

Operações Primitivas em Algoritmos

- Uma operação primitiva corresponde a uma instrução de baixo nível com um tempo de execução constante:
  - Atribuição de valores a variáveis;
    - Exemplo:
      - `variavel = 10;`
  - Chamadas de métodos;
    - Exemplo:
      - `MetodoCalculaSoma(10,20);`

Operações Primitivas em Algoritmos

- Operações aritméticas;
  - Exemplo:
    - `total = x + y;`
- Comparação entre dois números;
  - Exemplo:
    - `total == 10;`
- Acesso a um arranjo;
  - Exemplo:
    - `vetor[0] = 10;`

Operações Primitivas em Algoritmos

- Seguimento de uma referência para um objeto;
  - Exemplo:
    - `metodoVerificaNome (aluno);`
- Retorno de um método.
  - Exemplo:
    - `return total;`

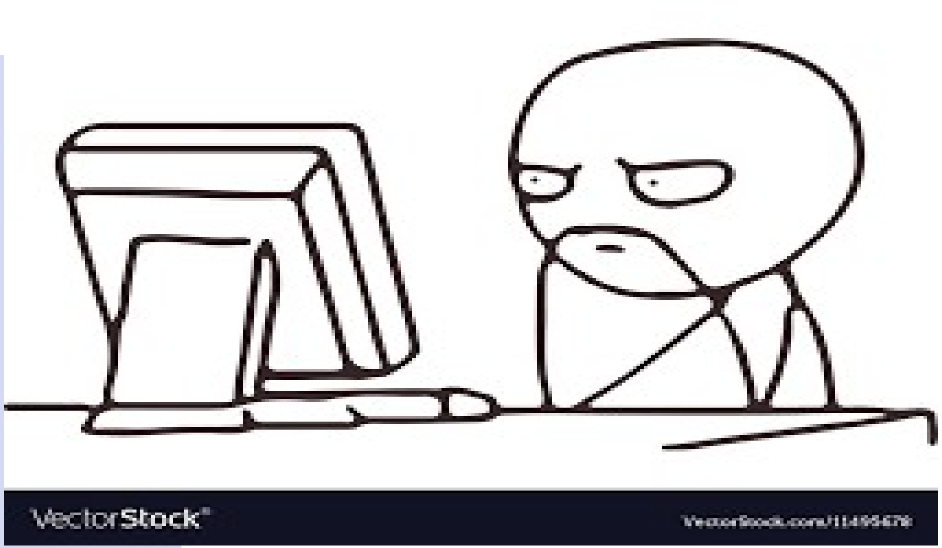
Contagem das Operações Primitivas

- Contagem de quantas operações primitivas são executadas;
- Usa este número *t* como uma estimativa do tempo de execução do algoritmo;
- Assume-se implicitamente que os tempos de execução de operações diferentes são similares.

Contagem das Operações Primitivas

- Um algoritmo pode executar mais rapidamente sobre algumas entradas do que sobre outras, mesmo sendo do mesmo tamanho.
- O ideal é obter o tempo de execução como uma função do tamanho de entrada obtido pela média de todas as possíveis entradas do mesmo tamanho.
  - Análise do caso médio é bastante desafiador.

Como realizar a contagem?



```java
//Código que faz parte do artigo para iniciantes em www.alexandregama.wordpress.com

//Primeira versão
import java.util.Scanner;

public class DivisaoDeNumeros {

	public static void main(String[] args) {
		//Criamos um objeto Scanner para capturar o que foi digitado
		Scanner input = new Scanner(System.in);
		//Imprime mensagem para a inserção do primeiro valor
		System.out.println("Insira o valor do dividendo: ");
		//Guarda o valor digitado pelo usuário na variável dividendo
		int dividendo = input.nextInt();
		//Imprime mensagem para a inserção do segundo valor
		System.out.println("Insira o valor do divisor: ");
		//Guarda o valor digitado pelo usuário na variável divisor
		int divisor = input.nextInt();

		//Verifica se o valor do divisor é igual a zero
		if (divisor == 0) {
			//Imprime o valor -1 caso o divisor seja zero
			System.out.println("-1");
		}
		//Verifica se o valor do cálculo da divisão é negativo
		else if ((dividendo / divisor < 0)) {
			//Imprime o valor 0 caso o resultdo da divisão seja negativo
			System.out.println("Valor encontrado: 0");
		}
		else {
			//Como o divisor não é zero e o cálculo não é negativo, imprime o resultado da divisão
			System.out.println("Valor calculado: " + dividendo / divisor);
		}
	}
}
```

Devemos analisar
<u>LINHA A LINHA</u>
do método desejado

E olhar qual o tipo de
operação primitiva
que o método realiza
na LINHA

```java
//Código que faz parte do artigo para iniciantes em www.alexandregama.wordpress.com

//Primeira versão
import java.util.Scanner;

public class DivisaoDeNumeros {

        public static void main(String[] args) {
                //Criamos um objeto Scanner para capturar o que foi digitado
                Scanner input = new Scanner(System.in);
                //Imprime mensagem para a inserção do primeiro valor
                System.out.println("Insira o valor do dividendo: ");
                //Guarda o valor digitado pelo usuário na variável dividendo
                int dividendo = input.nextInt();
                //Imprime mensagem para a inserção do segundo valor
                System.out.println("Insira o valor do divisor: ");
                //Guarda o valor digitado pelo usuário na variável divisor
                int divisor = input.nextInt();

                //Verifica se o valor do divisor é igual a zero
                if (divisor == 0) {
                        //Imprime o valor -1 caso o divisor seja zero
                        System.out.println("-1");
                }
                //Verifica se o valor do cálculo da divisão é negativo
                else if ((dividendo / divisor < 0)) {
                        //Imprime o valor 0 caso o resultdo da divisão seja negativo
                        System.out.println("Valor encontrado: 0");
                }
                else {
                        //Como o divisor não é zero e o cálculo não é negativo, imprime o resultado da div
                        System.out.println("Valor calculado: " + dividendo / divisor);
                }
        }
}
```

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

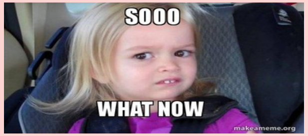
Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

Valor da
Op. Primitiva = 1

```java
//Código que faz parte do artigo para iniciantes em www.alexandregama.wordpress.com

//Primeira versão
import java.util.Scanner;

public class DivisaoDeNumeros {

        public static void main(String[] args) {
                //Criamos um objeto Scanner para capturar o que foi digitado
                Scanner input = new Scanner(System.in);
                //Imprime mensagem para a inserção do primeiro valor
                System.out.println("Insira o valor do dividendo: ");
                //Guarda o valor digitado pelo usuário na variável dividendo
                int dividendo = input.nextInt();
                //Imprime mensagem para a inserção do segundo valor
                System.out.println("Insira o valor do divisor: ");
                //Guarda o valor digitado pelo usuário na variável divisor
                int divisor = input.nextInt();

                //Verifica se o valor do divisor é igual a zero
                if (divisor == 0) {
                        //Imprime o valor -1 caso o divisor seja zero
                        System.out.println("-1");
                }
                //Verifica se o valor do cálculo da divisão é negativo
                else if ((dividendo / divisor < 0)) {
                        //Imprime o valor 0 caso o resultdo da divisão seja negativo
                        System.out.println("Valor encontrado: 0");
                }
                else {
                        //Como o divisor não é zero e o cálculo não é negativo, imprime o resultado
                        System.out.println("Valor calculado: " + dividendo / divisor);
                }
        }
}
```

E agora?



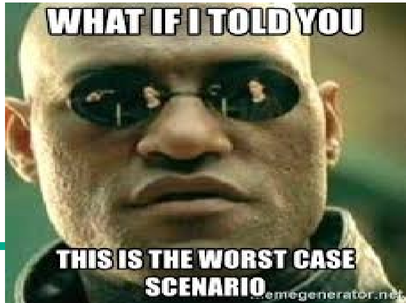
Somamos cada uma dessas operações

E temos…
O tempo aproximado que será gasto para rodar o algoritmo

Nesse caso
T(n) = 8 (ao calcular de maneira correta), ou
T(n) = constante

Focando no Pior Caso

- A análise do pior caso é mais fácil de ser realizada do que a análise do caso médio.
  - Requer apenas a identificação da entrada no pior caso possível.
  - Tipicamente, essa abordagem conduz a algoritmos melhores – se o algoritmo executa bem no pior caso, é necessário que ele execute melhor ainda para as demais entradas.



Notação Assintótica

Notação Assintótica

- Algoritmos tendem a perder desempenho a medida do crescimento da entrada de dados ou complexidade no seu processamento;
- Quando observamos tamanhos de entrada grandes o suficiente para tornar relevante apenas o crescimento do tempo de execução, estamos estudando a eficiência assintótica.

Notação Assintótica

- Em geral, um algoritmo que é assintoticamente mais eficiente será a melhor escolha para todas as entradas, exceto as muito pequenas.
- Para medir o custo de execução de um algoritmo é comum definir uma **função de complexidade *f***.

Notação Assintótica

- Onde *f*(*n*) é a medida do tempo necessário para executar um algoritmo para determinado problema *n*.
  - Se *f*(*n*) é uma medida da quantidade do tempo necessário para executar um algoritmo em um problema de tamanho *n*, então *f* é chamada **função de complexidade de tempo** do algoritmo;

  - Se *f*(*n*) é uma medida da quantidade de memória necessária para executar um algoritmo de tamanho *n*, então *f* é chamada **função de complexidade de espaço** do algoritmo.

Exemplo - Notação Assintótica

- Problema: acessar os registros de um arquivo através de um algoritmo de pesquisa sequencial.
- Seja *f* uma função de complexidade tal que *f*(*n*) é o número de vezes que a chave de consulta é comparada com a chave de cada registro. Os casos a considerar são:
  - Melhor caso:
    - O melhor caso ocorre quando o registro encontrado é o primeiro consultado.
    - *f*(*n*) = 1;

Registros	25	16	3	4	87	45	98	62	12	34
Índice	0	1	2	3	4	5	6	7	8	9

Queremos encontrar o número 25

Exemplo - Notação Assintótica

- Problema: acessar os registros de um arquivo através de um algoritmo de pesquisa sequencial.
- Seja *f* uma função de complexidade tal que $f(n)$ é o número de vezes que a chave de consulta é comparada com a chave de cada registro. Os casos a considerar são:
  - Pior caso:
    - O pior caso ocorre quando o registro encontrado é o último ou não existe no arquivo.
    - $f(n) = n$;

Registros	25	16	3	4	87	45	98	62	12	34
Índice	0	1	2	3	4	5	6	7	8	9

Queremos encontrar o número 76

Exemplo - Notação Assintótica

- Problema: acessar os registros de um arquivo através de um algoritmo de pesquisa sequencial.
- Seja *f* uma função de complexidade tal que $f(n)$ é o número de vezes que a chave de consulta é comparada com a chave de cada registro. Os casos a considerar são:
  - Médio caso:
    - $f(n) = (n + 1)/2$;

Registros	25	16	3	4	87	45	98	62	12	34
Índice	0	1	2	3	4	5	6	7	8	9

Queremos encontrar
o número 45

Notação Assintótica

- O comportamento assintótico de $f(n)$ representa o limite do comportamento do custo quando n cresce.
- Para uma dominação assintótica existem notações:
  - Notação O (ômicron) – conhecido como “Big O”
  - Notação Ω (ômega)
  - Notação Θ (theta)

Notação Assintótica

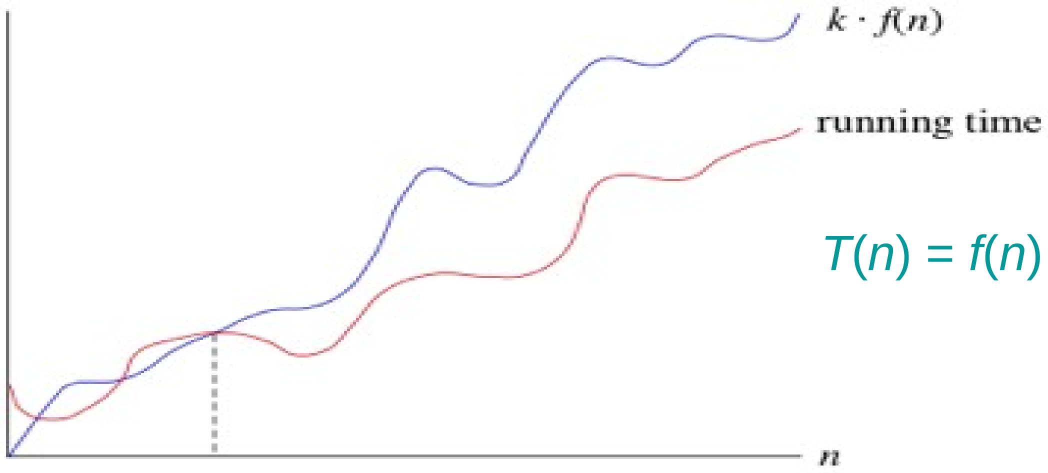
- Notação “Big O” para limites superiores
  - $T(n) = O(f(n))$ se $| T(n) / f(n) |$ é delimitada acima como $n \to \infty$
- Notação “Ômega” para limites inferiores
  - $T(n) = \Omega\ (f(n))$ se $| T(n) / f(n) |$ é delimitada abaixo como $n \to \infty$
- Notação “Theta” para ordem de crescimento com valor constante.
  - $T(n) = \Theta\ (f(n))$ se $T(n) = O(f(n))$ e $T(n) = \Omega\ (f(n))$

Notação Assintótica O

- Dizemos que $T(n)$ é $O(g(n))$ se existem constantes positivas c e n_0 tal que

$$T(n) <= c \, . \, g(n) \text{ para todo } n >= n_0$$

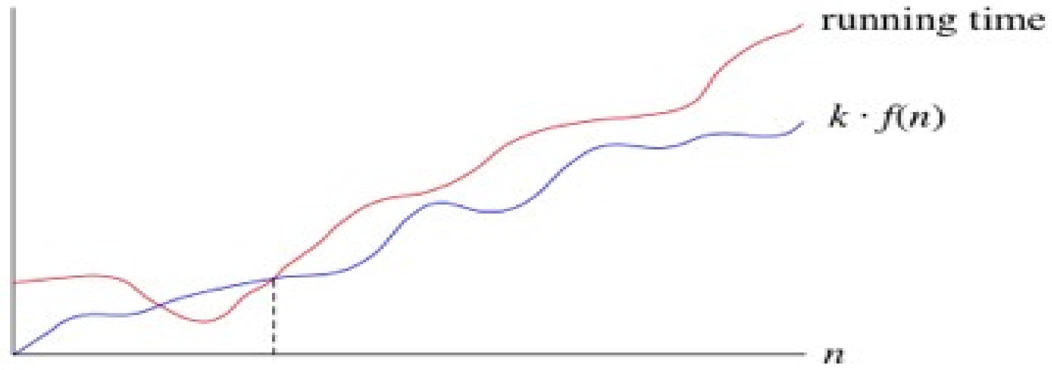
- Para n grande, $T(n)$ não cresce mais rápido que $g(n)$.
- $T(n)$ é $O(g(n))$ deve ser entendido como $T(n) \in O(g(n))$.



Notação Assintótica Ω

- Dizemos que $T(n)$ é $\Omega(g(n))$ se existem constantes positivas c e n_0 tal que

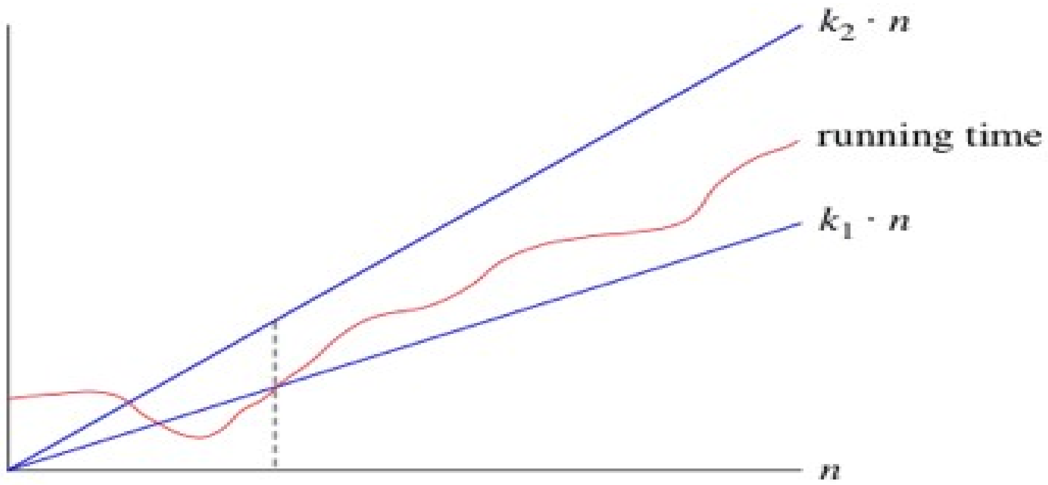
$$T(n) >= c \,.\, g(n) \text{ para todo } n >= n_0 \,.$$



$T(n) = f(n)$

Notação Assintótica Θ

- Θ define uma relação de equivalência.
- $T(n)$ é $\Theta(f(n))$ se $T(n) \in O(f(n))$ e $T(n) \in \Omega(f(n))$
- Limite assintótico justo.



$T(n) = f(n)$

Notação Assintótica

- A notação O possui sua importância, pois o programador conclui que seu algoritmo é no **máximo** tão complexo a uma função.
- Mas, no **mínimo** tão complexo como a notação Ω descreve, não é importante para conclusões práticas sobre algoritmos.

Notação Assintótica

- Essas notações ignoram os fatores constantes e os termos de menor ordem, mantendo o foco nos principais componentes da função que afetam seu crescimento.

$$\mathbf{5}\boldsymbol{n}^{4} + \mathbf{3}\boldsymbol{n}^{3} + \mathbf{2}\boldsymbol{n}^{2} + \mathbf{4}\boldsymbol{n}\mathbf{+1} \text{ é}$$

$$\mathbf{O}(\boldsymbol{n}^{4})$$

- Justificativa:

$$5n^4 + 3n^3 + 2n^2 + 4n+1 <= (5 + 3+ 2 + 4 +1)n^4 = cn^4$$

- para $c = 15$, quando $n >= n_0 = 1$

Comparação de Execução

- Para projetar um algoritmo eficiente, é fundamental preocupar-se com a sua complexidade. Como exemplo: considere a **sequência de Fibonacci**.

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

- 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.

Comparação de Execução

- Seja a função 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

```
private static int fibonacci(int n) {

    if (n <= 1) {
        vetor[n] = n;
        return n;
    } else {
        vetor[n] = fibonacci(n - 1) + fibonacci(n - 2);

        return vetor[n];
    }

}
```

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

Comparação de Execução

```java
private static void fibonacci(int numero) {
    int total = 0;
    int anterior = 0;
    int preAnterior = 0;
    int i = 0;

    while (i < numero) {
        if (i == 0) {
            total = 0;
            anterior = 0;
            preAnterior = 0;
        } else if (i == 1){
            total = 1;
            anterior = total;
            preAnterior = 0;
        } else {
            total = anterior + preAnterior;
            preAnterior = anterior;
            anterior = total;

        }
        i++;
        if (i == numero){
            System.out.println (total);
        } else {
            System.out.print(total + ", ");
        }
    }
}
```

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

Exemplo de Análise de Algoritmo

- Vamos considerar o seguinte algoritmo de busca. Como parâmetro, passaremos um valor de *k* que não se encontra no vetor (pior caso):

```
int busca(int[] vetor, int k) {
    int i = 1;
    while (i <= n) {

        if (vetor[i] == k) {

            return i;
        }
        i++;
    }
    return -1;
}
```

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23  
24          if (vetor[i] == k) {
25  
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

Neste caso, a atribuição da linha 21 será executada 1 vez.

- $T(n) = 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A comparação da linha 22 será executada para cada valor de i, de 1 a $n + 1$, ou seja, $n + 1$ vezes.

- $T(n) = 1 + (n + 1) + \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A comparação da linha 24 será executada para cada valor de 1 a *n*, ou seja, *n* vezes.

- $T(n) = 1 + (n + 1) + (n) + \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A linha 26 não será executada.

- $T(n) = 1 + (n + 1) + (n) + 0 + \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23  
24          if (vetor[i] == k) {
25  
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A linha 28 será executada *n* vez.

- $T(n) = 1 + (n + 1) + (n) + 0 + (n)$ ...

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A linha 30 será executada 1 vez.

- $T(n) = 1 + (n + 1) + (n) + 0 + (n) + 1$

Exemplo de Análise de Algoritmo

- Logo,

$$T(n) = c_1 + (n + 1)c_2 + nc_3 + nc_5 + c_6$$

$$= c_1 + nc_2 + c_2 + nc_3 + nc_5 + c_6$$

$$= (c_2 + c_3 + c_5)n + c_1 + c_2 + c_6$$

onde c_i é o custo de executar a linha i.

- Assim, definimos:

$$a = (c_2 + c_3 + c_5); \text{ e}$$

$$b = (c_1 + c_2 + c_6)$$

Exemplo de Análise de Algoritmo

- Temos que:

$$T\,(n) = an + b$$

- Concluímos que o tempo de execução do algoritmo no pior caso varia linearmente com o tamanho da entrada (*n*) à qual é submetido.
- Assim, determinamos um limite superior para todos os outros casos!
  - Não há caso que faça o algoritmo ser pior.
  - $T(n) \in O(n)$

Exemplo de Análise de Algoritmo

- E como seria a análise do mesmo algoritmo no melhor caso?

```
int busca(int[] vetor, int k) {
    int i = 1;
    while (i <= n) {

        if (vetor[i] == k) {

            return i;
        }
        i++;
    }
    return -1;
}
```

Exemplo de Análise de Algoritmo

```
20    int busca(int[] vetor, int k) {
21        int i = 1;
22        while (i <= n) {
23
24            if (vetor[i] == k) {
25
26                return i;
27            }
28            i++;
29        }
30        return -1;
31    }
```

Neste caso, a atribuição da linha 21 será executada 1 vez.

- $T(n) = 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A comparação linha 22 será executada 1 vez, pois o número que gostaríamos está na primeira posição.

- $T(n) = 1 + 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20 int busca(int[] vetor, int k) {
21     int i = 1;
22     while (i <= n) {
23 
24         if (vetor[i] == k) {
25 
26             return i;
27         }
28         i++;
29     }
30     return -1;
31 }
```

A comparação linha 24 será executada 1 vez, pois o número que gostaríamos está na primeira posição.

- $T(n) = 1 + 1 + 1 + \ldots$

Exemplo de Análise de Algoritmo

```
20  int busca(int[] vetor, int k) {
21      int i = 1;
22      while (i <= n) {
23
24          if (vetor[i] == k) {
25
26              return i;
27          }
28          i++;
29      }
30      return -1;
31  }
```

A comparação linha 26 será executada 1 vez.

- $T(n) = 1 + 1 + 1 + 1$

Análise de Algoritmos Recursivos

- Vamos considerar o algoritmo recursivo abaixo para calcular o fatorial de um número inteiro não-negativo (*n*).

```
private static int fatorial(int numero) {

    //Quando o numero for 0, o resultado do fatorial sempre vai ser 1
    if (numero == 0) {
        return 1;
    } else {
        return numero * fatorial(numero - 1);
    }
}
```

Análise de Algoritmos Recursivos

- Para fazer a análise desse algoritmo, é necessário dividi-lo em dois:
  - Caso a condicional seja verdadeira:
    - $T(n) = t_1 \rightarrow n = 0$
  - Caso a condicional seja falsa:
    - $T(n) = T(n - 1) + t_2 \rightarrow n > 0$
- Esse tipo de equação é chamado de “**relação de recorrência**”.

Análise de Algoritmos Recursivos

- Uma técnica para resolver uma relação de recorrência é a "substituição repetida";
- Dado que $T(n) = T(n - 1) + t_2$, podemos falar que $T(n - 1) = T(n - 2) + t_2$, para $n > 1$;
- Repetindo esse processo, temos:

$$T(n) = T(n - 1) + t_2$$

$$= (T(n - 2) + t_2) + t_2$$

$$= T(n - 2) + 2t_2$$

$$= (T(n - 3) + t_2) + 2t_2$$

$$= T(n - 3) + 3t_2$$

$$\ldots$$

Análise de Algoritmos Recursivos

- Identificando o padrão:

$$T(n) = T(n - k) + kt_2$$

- onde $1 \leq k \leq n$.
- Já que sabermos que $T(0) = t_1$, no padrão acima isso acontecerá quando n = k. Assim:

$$T(n) = T(n - k) + kt_2$$

$$= T(0) + nt_2$$

$$= t_1 + nt_2$$

Importância de Análise de Algoritmos

- Considere:
  - Um computador A mais rápido (10 bilhões de instruções por segundo - 10^{10}) que executa um algoritmo cujo tempo de execução para *n* valores cresce segundo n^2 ;
  - Um computador B mais lento (10 milhões de instruções por segundo - 10^{7}) que executa um algoritmo cujo tempo de execução para n valores cresce segundo *n* log *n*;
  - O computador A é mil vezes mais rápido do que o computador B em poder de computação bruto.
- Cada computador deverá operar sobre um conjunto de 10 milhões de elementos (números inteiros).

Importância de Análise de Algoritmos

- Considere também que:
  - o programador do computador A elaborou um programa que o código resultante exija $2n^2$ instruções para n números, feito em linguagem de máquina.
  - o programador do computador B, usando uma linguagem de alto nível com um compilador ineficiente, desenvolveu um código resultante que exija $50\ n \log n$ instruções.

Importância de Análise de Algoritmos

- Para 10 milhões de números, o computador A leva:

$$\frac{2 \times (10^7)^2 \text{ instruções}}{10^{10} \text{ instruções/segundo}} = 20000 \text{ segundos}$$

  - um pouco mais de 5 horas e meia.
- Já o computador B leva:

$$\frac{50 \text{ x } 10^7 \times \log 10^7 \text{ instruções}}{10^7 \text{ instruções/segundo}} = 1163 \text{ segundos}$$

  - que é menos do que 20 minutos.

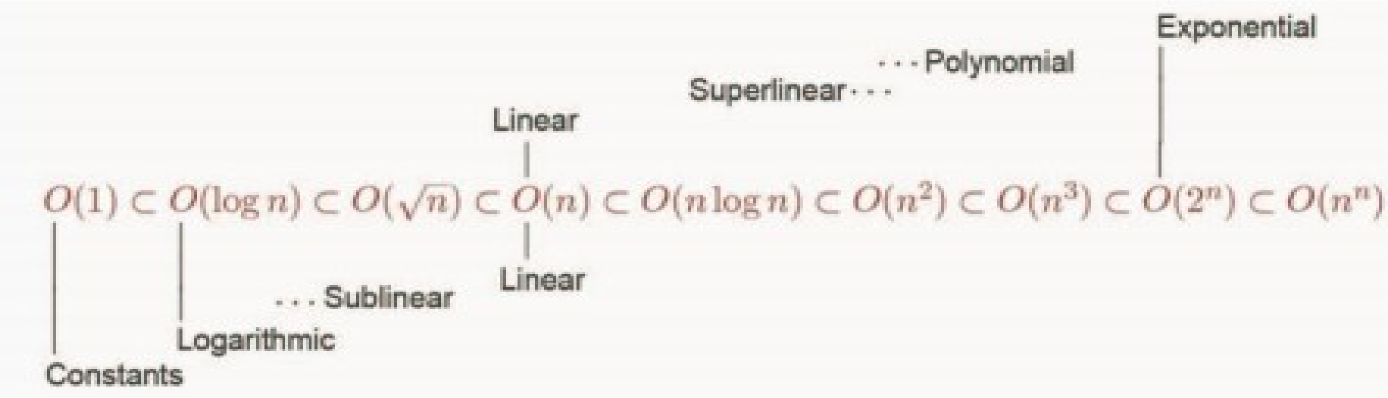
Importância de Análise de Algoritmos

- Usando um algoritmo cujo tempo de execução cresce mais lentamente, mesmo com um compilador ruim, o computador B executa 17 vezes mais rapidamente do que o computador A.
- Para 100 milhões de números, essa diferença se destaca ainda mais:
  – No computador A com o algoritmo de ordem n^2 a execução leva mais de 23 dias.
  – Já no algoritmo $n \log n$, no computador B, a execução leva menos de 4 horas.

Conclusões

- Mesmo com o avanço contínuo no hardware, o desempenho total do sistema depende da **escolha de algoritmos eficientes**, juntamente com a escolha de hardware rápido e sistemas operacionais eficientes.

Extra - Taxa de Crescimento

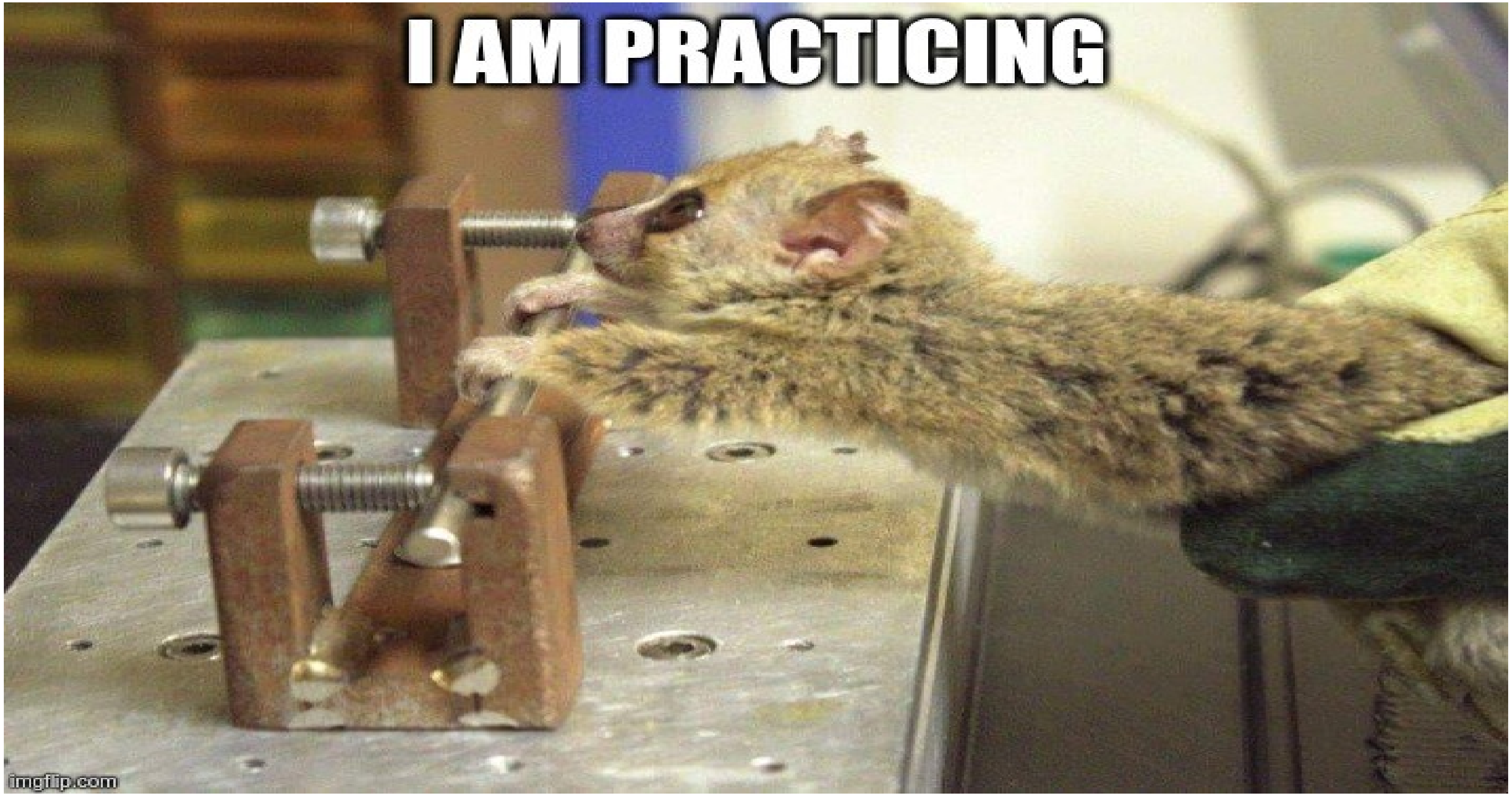


Dúvidas

Referência

- DOBRUSHKIN, Vladimir A. Métodos para Análise de Algoritmos. LTC, 03/2012

Exercícios



Exercício 01

- Analise o algoritmo abaixo expressando o tempo – T(n) – no melhor e no pior caso.
- Expresse o tempo também em notação assintótica.

```java
double[] mediaAcumulada(int V[]) {
    double[] M = new double[V.length];
    for (int i = 0; i < V.length; i++) {
        double soma = 0;
        for (int j = 0; j <= i; j++) {
            soma += V[j];
        }
        M[i] = soma / (i + 1);
    }
    return M;
}
```

Exercício 02

- Especifique um programa que calcule as médias acumuladas de um vetor *V* contendo *n* inteiros. As médias devem ser armazenadas em um vetor *M* com *n* reais assim como o exercício anterior, porém esse algoritmo deve ser no pior caso O(*n*).
  - Faça a análise de tempo no melhor e pior caso do algoritmo proposto.
  - Faça uma comparação de tempo entre os algoritmos dos exercícios 01 e 02.

Exercício 03

- Analise o algoritmo abaixo expressando o tempo – T(n) – no pior caso. Expresse o tempo também em notação assintótica.

```java
private void exercicio(int n) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            System.out.println(i * j * n);
        }
    }

}
```

Obrigado!
Bom Dia! Boa Tarde!
Boa Noite!

bruno.moritani@anhembi.br