Análise de Complexidade

Algoritmos e Estrutura de Dados II

Professor: Kennedy R. Lopes

UFERSA

22 de fevereiro de 2022

Introdução

- Um algoritmo é um conjunto **finito** de passos.

Introdução

- Um algoritmo é um conjunto **finito** de passos.
- Entretanto, a existência de um algoritmo não garante que possa ser resolvido.

Introdução

- Um algoritmo é um conjunto **finito** de passos.
- Entretanto, a existência de um algoritmo não garante que possa ser resolvido.
- Condições de tempo e memória devem ser avaliadas.

Introdução

- Um algoritmo é um conjunto **finito** de passos.
- Entretanto, a existência de um algoritmo não garante que possa ser resolvido.
- Condições de tempo e memória devem ser avaliadas.
- Ex.: Resolução de um jogo de xadrez.
  - Tamanho do tabuleiro: 64.
  - Duração média do jogo: 80.

Introdução

- Um algoritmo é um conjunto **finito** de passos.
- Entretanto, a existência de um algoritmo não garante que possa ser resolvido.
- Condições de tempo e memória devem ser avaliadas.
- Ex.: Resolução de um jogo de xadrez.
  - Tamanho do tabuleiro: 64.
  - Duração média do jogo: 80.
  - Possíveis estados: (10^{47}).

Introdução

- Algoritmos demandam tempo de execução e recursos:
  - Memória
  - Espaço em disco
  - Dispositivos externos
  - Banda de rede
  - ...
- Um bom programador deve ter o atributo de **poupar** tempo e recursos.
- A principal desempenho avaliado é a *economia* do tempo necessário para o cálculo dos algoritmos.

Introdução

- A análise de complexidade é uma ferramenta que avalia como um algoritmo se comporta com diferentes dados de entrada.
- A depender da entrada, o algoritmo pode ter diferente tempo de execução.
- O problema a ser resolvido pode ser calculado pelo seu tamanho(*custo*) a ser processado, exemplo:
  - exec01: Calcular o *custo* em ordenar um vetor de inteiros com 100 posições;
  - exec02: Calcular o *custo* em ordenar um vetor de inteiros com 1000 posições;
- Aparentemente exec02 *demora* 10 vezes mais do que exec01. Entretanto iremos observar que isso nem sempre (quase sempre não é) verdade.

Introdução

O tempo da execução de um algoritmo T em função das n instruções existentes.

$$T(n) = \sum_{i=1}^{n} t_i n_i$$

Sendo t_i o tempo necessário para a excução da instrução i e n_i o número de vezes que a instrução i é executada.

Introdução

O tempo da execução de um algoritmo T em função das n instruções existentes.

$$T(n) = \sum_{i=1}^{n} t_i n_i$$

Sendo t_i o tempo necessário para a excução da instrução i e n_i o número de vezes que a instrução i é executada.
Obviamente o tempo t_i é de difícil obtenção, depende:

Introdução

O tempo da execução de um algoritmo T em função das n instruções existentes.

$$T(n) = \sum_{i=1}^{n} t_i n_i$$

Sendo t_i o tempo necessário para a excução da instrução i e n_i o número de vezes que a instrução i é executada.
Obviamente o tempo t_i é de difícil obtenção, depende:

- Memória disponível do computador;
- Desempenho do processador;
- Arquitetura e estado dos dispositivos naquele momento de execução.

Introdução

O tempo da execução de um algoritmo T em função das n instruções existentes.

$$T(n) = \sum_{i=1}^{n} t_i n_i$$

Sendo t_i o tempo necessário para a excução da instrução i e n_i o número de vezes que a instrução i é executada.

Obviamente o tempo t_i é de difícil obtenção, depende:

- Memória disponível do computador;
- Desempenho do processador;
- Arquitetura e estado dos dispositivos naquele momento de execução.

Além do mais, mesmo executando diversas vezes no mesmo pc, o algoritmo pode executar a mesma instrução t_i com valores distintos.

Introdução

O tempo da execução de um algoritmo T em função das n instruções existentes.

$$T(n) = \sum_{i=1}^{n} t_i n_i$$

Sendo t_i o tempo necessário para a excução da instrução i e n_i o número de vezes que a instrução i é executada.
Obviamente o tempo t_i é de difícil obtenção, depende:

- Memória disponível do computador;
- Desempenho do processador;
- Arquitetura e estado dos dispositivos naquele momento de execução.

Além do mais, mesmo executando diversas vezes no mesmo pc, o algoritmo pode executar a mesma instrução t_i com valores distintos.
Portanto, utiliza-se apenas a contagem de vezes que cada instrução é executada (frequência da instrução i).

Exemplos da contagem de frequência

```c
//f = 1
int main() {
    int x = 0;
    x = x + 1;
    return 0;
}
```

Exemplos da contagem de frequência

```c
//f = 1
int main() {
    int x = 0;
    x = x + 1;
    return 0;
}
```

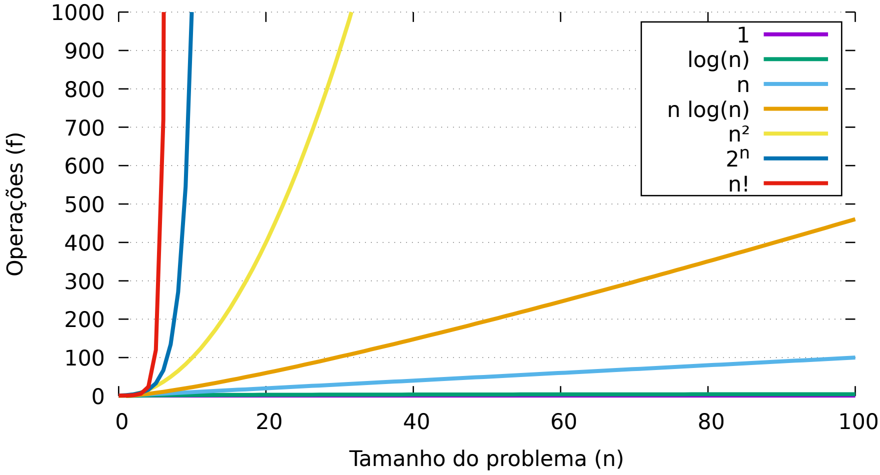
```c
//f = n
int main() {
    int x = 0, n=10;

    for (int i = 0; i < n; i++) {
        x = x + 1;
    }
    return 0;
}
```

Análise de Complexidade

- A maior frequência encontrada em um programaa é chamada de **ordem de grandeza** de crescimento de tempo do programa.
- A ordem de grandeza de um algoritmo é o principal parâmetro de análise do desempenho de sua execução.
- Seja N um parâmetro que caracteriza o tamanho de um problema, as ordens de grandeza mais comuns são:
  - $O(1) \rightarrow$ contante
  - $O(log_2 N) \rightarrow$ logaritmo
  - $O(N) \rightarrow$ linear
  - $O(N\ log_2 N)$
  - $O(N^2) \rightarrow$ quadrática
  - $O(N^3) \rightarrow$ cúbica
  - $O(2^N) \rightarrow$ exponencial
  - $O(N!) \rightarrow$ fatorial

Análise de Complexidades



Comportamento assintótico

Uma função $g(n)$ domina **assintoticamente** outra função $f(n)$ se existem duas constantes positivas c e n_0 tais que, para $n > n_0$, temos:

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

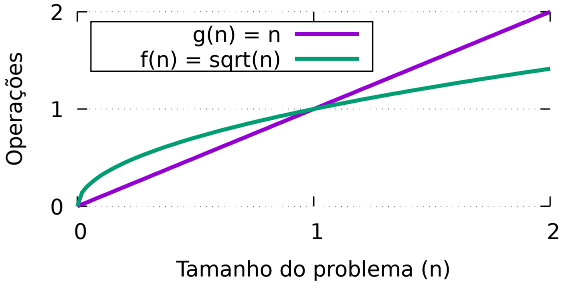
Se isso ocorre, dizemos que $g(n)$ atua como um limite superior para valores assintóticos da função $f(n)$, ou que $f = O(g)$.

Comportamento assintótico

Uma função $g(n)$ domina **assintoticamente** outra função $f(n)$ se existem duas constantes positivas c e n_0 tais que, para $n > n_0$, temos:

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

Se isso ocorre, dizemos que $g(n)$ atua como um limite superior para valores assintóticos da função $f(n)$, ou que $f = O(g)$.



Comportamento assintótico

Exemplo: Analisar se $g(n) = 0.2n^2$ domina assintoticamente $f(n) = 1000n$.

Comportamento assintótico

Exemplo: Analisar se $g(n) = 0.2n^2$ domina assintoticamente $f(n) = 1000n$.
Precisamos verificar então se existe um $c > 0$ e um $n_0 > 0$, com $n_0, c \in \mathbb{Z}$ de tal forma que

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

Comportamento assintótico

Exemplo: Analisar se $g(n) = 0.2n^2$ domina assintoticamente $f(n) = 1000n$. Precisamos verificar então se existe um $c > 0$ e um $n_0 > 0$, com $n_0, c \in \mathbb{Z}$ de tal forma que

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

Portanto:

$$\begin{aligned} |f(n)| &\leq |c\ g(n)| \\ |1000n| &\leq |c * 0.2n^2| \\ (c * 0.2n^2) - 1000n &\geq 0 \\ n(0.2cn - 1000) &\geq 0 \end{aligned}$$

Comportamento assintótico

Exemplo: Analisar se $g(n) = 0.2n^2$ domina assintoticamente $f(n) = 1000n$. Precisamos verificar então se existe um $c > 0$ e um $n_0 > 0$, com $n_0, c \in \mathbb{Z}$ de tal forma que

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

Portanto:

$$\begin{aligned} |f(n)| &\leq |c\ g(n)| \\ |1000n| &\leq |c * 0.2n^2| \\ (c * 0.2n^2) - 1000n \geq 0 \\ n(0.2cn - 1000) \geq 0 \end{aligned}$$

Nesta situação, a única condição possível é considerar o segundo termo $(0.2cn - 1000 > 0)$, já que desejamos escolher um $n_0 > 0$ para validar a situação. Pode-se escolher então $c = 5000$ para descobrir que a partir de $n = n_0 \geq 1$:

$$g(n) \geq f(n)$$

Comportamento assintótico

Exemplo 02: Prove que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$

Comportamento assintótico

Exemplo 02: Prove que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$
Demonstração:

$$\begin{aligned}
2n^3 + 3n^2 &\leq c * n^3 \\
2n^3 + 3n^2 - c * n^3 &\leq 0 \\
(2 - c)n^3 + 3n^2 &\leq 0 \\
n^2 \left[(2 - c)n + 3\right] &\leq 0 \\
(2 - c)n + 3 &\leq 0
\end{aligned}$$

A partir daqui podemos buscar valores para c e $n = n_0 \geq 0$. Se encontrarmos algum, podemos afirmar que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$.

Comportamento assintótico

Exemplo 02: Prove que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$
Demonstração:

$$\begin{aligned} 2n^3 + 3n^2 &\leq c * n^3 \\ 2n^3 + 3n^2 - c * n^3 &\leq 0 \\ (2-c)n^3 + 3n^2 &\leq 0 \\ n^2\left[(2-c)n + 3\right] &\leq 0 \\ (2-c)n + 3 &\leq 0 \end{aligned}$$

A partir daqui podemos buscar valores para c e $n = n_0 \geq 0$. Se encontrarmos algum, podemos afirmar que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$.

- Com $c = 2$, $\nexists n_0$ que satisfaz.

Comportamento assintótico

Exemplo 02: Prove que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$
Demonstração:

$$
\begin{aligned}
2n^3 + 3n^2 &\leq c * n^3 \\
2n^3 + 3n^2 - c * n^3 &\leq 0 \\
(2 - c)n^3 + 3n^2 &\leq 0 \\
n^2 \left[(2 - c)n + 3\right] &\leq 0 \\
(2 - c)n + 3 &\leq 0
\end{aligned}
$$

A partir daqui podemos buscar valores para c e $n = n_0 \geq 0$. Se encontrarmos algum, podemos afirmar que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$.

- Com $c = 2$, $\nexists n_0$ que satisfaz.
- Com $c < 2$, os valores possíveis são de $n_0 < 0$, que não são válidos.

Comportamento assintótico

Exemplo 02: Prove que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$
Demonstração:

$$
\begin{aligned}
2n^3 + 3n^2 &\le c * n^3 \\
2n^3 + 3n^2 - c * n^3 &\le 0 \\
(2 - c)n^3 + 3n^2 &\le 0 \\
n^2 \left[(2 - c)n + 3\right] &\le 0 \\
(2 - c)n + 3 &\le 0
\end{aligned}
$$

A partir daqui podemos buscar valores para c e $n = n_0 \geq 0$. Se encontrarmos algum, podemos afirmar que $f(n) = 2n^3 + 3n^2$ é $O(n^3)$.

- Com $c = 2$, $\nexists n_0$ que satisfaz.
- Com $c < 2$, os valores possíveis são de $n_0 < 0$, que não são válidos.
- Com $c > 2$, qualquer $n_0 > 0$ satisfaz!

Comportamento assintótico - Exercícios

Exercício 01: Prove ou apresente uma contraprova que

$$\sum_{i=1}^{n} i^2 \text{ é } O(n^2)$$

Exercício 02: Prove ou apresente uma contraprova que

$$\sum_{i=1}^{n} i^2 \text{ é } O(n^3)$$

Exercício 03: Prove ou apresente uma contraprova que

$$\frac{an^k}{log(n)} \text{ é } O(n^k) \text{ com } a \in \mathbb{Z}_+^*$$

.

Contagem de frequência

Exemplo: Conte as operações (contagem de frequência) realizadas na operação de uma soma geométrica definida por:

$$S = \sum_{i=0}^{n} x^i$$

Analise as instruções do algoritmo que o descreve logo a seguir:

Contagem de frequência

```
float soma(float x, int n) {
    int soma = 0;
    for (int i = 0; i <= n; i++) {
        int prod = 1;
        for (int j = 0; j < i; j++)
            prod = prod * x;
        soma + soma + prod;
    }
    return soma;
}
```

A soma de todos os processamentos pode ser visualizado linha a linha:

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n + 2$

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n+2$

L4) $n+1$

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n + 2$

L4) $n + 1$

L6) $\sum_{i=0}^{n} i$

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n+2$

L4) $n+1$

L6) $\sum_{i=0}^{n} i$

L7) $n+1$

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n+2$

L4) $n+1$

L6) $\sum_{i=0}^{n} i$

L7) $n+1$

L9) 1

Contagem de frequência

```
1  float soma(float x, int n) {
2      int soma = 0;
3      for (int i = 0; i <= n; i++) {
4          int prod = 1;
5          for (int j = 0; j < i; j++)
6              prod = prod * x;
7          soma + soma + prod;
8      }
9      return soma;
10 }
```

A soma de todos os processamentos pode ser visualizado linha a linha:

L2) 1

L3) $n+2$

L4) $n+1$

L6) $\sum_{i=0}^{n} i$

L7) $n+1$

L9) 1

Somando todos os tempos, temos o tempo total

$$T(n) = \frac{n^2}{2} + \frac{7n}{2} + 6$$

Logo T é $O(n^2)$.

Fórmula de *Horner*

Pode-se modificar o algoritmo para melhorar o tempo de execução utilizando o algoritmo de *Horner*.

$$
\begin{aligned}
S = \sum_{i=0}^{n} x^i &= 1 + x + x^2 + \ldots + x^n \\
&= 1 + x(1 + x + x^2 + \ldots + x^{n-1}) \\
&= 1 + x(1 + x(1 + x + x^2 + \ldots + x^{n-2})) \\
&= 1 + x(1 + x(1 + x(1 + \ldots (1 + x(1 + x))) \ldots))
\end{aligned}
$$

Fórmula fechada

```c
float somaHorner(float x, int n) {
    int i, soma = 0;
    for (int i = 0; i <= n; i++) {
        soma = soma * x + 1;
    }
    return soma;
}
```

O tempo de processamento terá como referência essa quantidade de operações:

Fórmula fechada

```c
1 float somaHorner(float x, int n) {
2     int i, soma = 0;
3     for (int i = 0; i <= n; i++) {
4         soma = soma * x + 1;
5     }
6     return soma;
7 }
```

O tempo de processamento terá como referência essa quantidade de operações:

L2) 1

Fórmula fechada

```
1 float somaHorner(float x, int n) {
2     int i, soma = 0;
3     for (int i = 0; i <= n; i++) {
4         soma = soma * x + 1;
5     }
6     return soma;
7 }
```

O tempo de processamento terá como referência essa quantidade de operações:

L2) 1

L3) n+2

Fórmula fechada

```
1 float somaHorner(float x, int n) {
2     int i, soma = 0;
3     for (int i = 0; i <= n; i++) {
4         soma = soma * x + 1;
5     }
6     return soma;
7 }
```

O tempo de processamento terá como referência essa quantidade de operações:

L2) 1

L3) n+2

L4) n+1

Fórmula fechada

```
1 float somaHorner(float x, int n) {
2     int i, soma = 0;
3     for (int i = 0; i <= n; i++) {
4         soma = soma * x + 1;
5     }
6     return soma;
7 }
```

O tempo de processamento terá como referência essa quantidade de operações:

- L2) 1
- L3) n+2
- L4) n+1
- L6) 1

Fórmula fechada

```
1 float somaHorner(float x, int n) {
2     int i, soma = 0;
3     for (int i = 0; i <= n; i++) {
4         soma = soma * x + 1;
5     }
6     return soma;
7 }
```

O tempo de processamento terá como referência essa quantidade de operações:

L2) 1

L3) n+2

L4) n+1

L6) 1

Portanto $T(n) = 1 + (n+2) + (n+1) + 1 = 2n + 5$ e este algoritmo é $O(n)$.

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$S = \sum_{i=0}^{n} i = 1 + x + x^2 + \ldots + x^n$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$S = \sum_{i=0}^{n} i = 1 + x + x^2 + \ldots + x^n$$

$$xS = x(1 + x + x^2 + \ldots + x^n)$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$S = \sum_{i=0}^{n} i = 1 + x + x^2 + \ldots + x^n$$

$$xS = x(1 + x + x^2 + \ldots + x^n)$$

$$xS = x + x^2 + \ldots + x^{n+1}$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$S = \sum_{i=0}^{n} i = 1 + x + x^2 + \ldots + x^n$$

$$xS = x(1 + x + x^2 + \ldots + x^n)$$

$$xS = x + x^2 + \ldots + x^{n+1}$$

$$xS + 1 = 1 + x + x^2 + \ldots + x^{n+1}$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$
\begin{aligned}
S = \sum_{i=0}^{n} i &= 1 + x + x^2 + \ldots + x^n \\
xS &= x(1 + x + x^2 + \ldots + x^n) \\
xS &= x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= 1 + x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= S + x^{n+1}
\end{aligned}
$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$
\begin{aligned}
S = \sum_{i=0}^{n} i &= 1 + x + x^2 + \ldots + x^n \\
xS &= x(1 + x + x^2 + \ldots + x^n) \\
xS &= x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= 1 + x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= S + x^{n+1} \\
xS - S &= x^{n+1} - 1
\end{aligned}
$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$
\begin{aligned}
S = \sum_{i=0}^{n} i &= 1 + x + x^2 + \ldots + x^n \\
xS &= x(1 + x + x^2 + \ldots + x^n) \\
xS &= x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= 1 + x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= S + x^{n+1} \\
xS - S &= x^{n+1} - 1 \\
(x-1)S &= x^{n+1} - 1
\end{aligned}
$$

Fórmula fechada

Uma terceira solução é encontrar a fórmula fechada:

$$
\begin{aligned}
S = \sum_{i=0}^{n} i &= 1 + x + x^2 + \ldots + x^n \\
xS &= x(1 + x + x^2 + \ldots + x^n) \\
xS &= x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= 1 + x + x^2 + \ldots + x^{n+1} \\
xS + 1 &= S + x^{n+1} \\
xS - S &= x^{n+1} - 1 \\
(x-1)S &= x^{n+1} - 1
\end{aligned}
$$

$$S = \frac{x^{n+1} - 1}{x - 1}$$

Fórmula fechada

Qual a complexidade desse algoritmo com a fórmula fechada?

Fórmula fechada

Qual a complexidade desse algoritmo com a fórmula fechada?

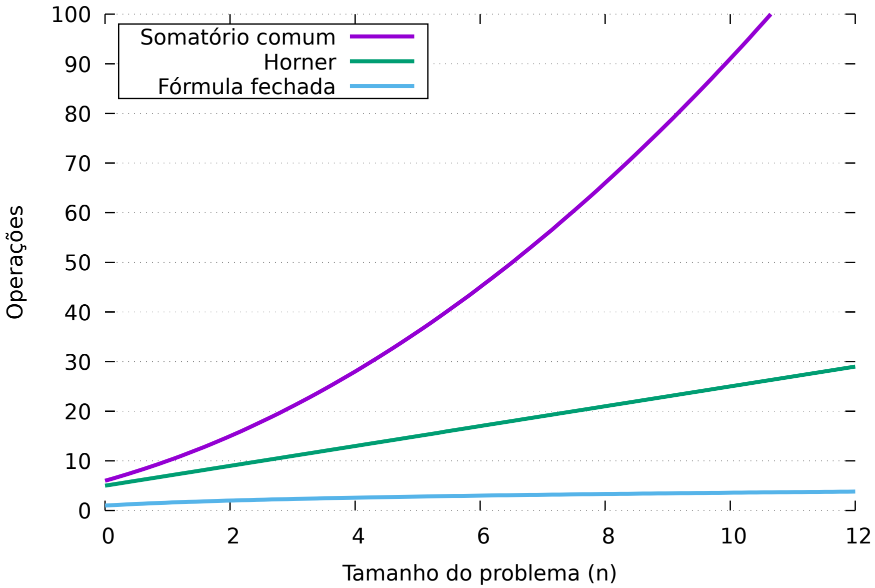
```c
#include <math.h>
float soma(int x, int n) {
    return pow(x, n + 1) / (x - 1);
}
```

Fórmula fechada

Qual a complexidade desse algoritmo com a fórmula fechada?

```c
#include <math.h>
float soma(int x, int n) {
    return pow(x, n + 1) / (x - 1);
}
```

Depende da complexidade do algoritmo potência. Considerando que tenha complexidade $T(n) = log_2(n+2)$, então essa também será a potência do algoritmo com a fórmula fechada.


Somatório comum
Horner
Fórmula fechada
Operações
Tamanho do problema (n)
0
10
20
30
40
50
60
70
80
90
100
0
2
4
6
8
10
12

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n			
$nlogn$			
n^2			
n^3			
2^n			
3^n			

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$			
n^2			
n^3			
2^n			
3^n			

[1]Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2			
n^3			
2^n			
3^n			

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3			
2^n			
3^n			

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n			
3^n			

[1]Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n	$10s$	$27dias$	
3^n			

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n	$10s$	$27dias$	3660 séculos
3^n			

[1]Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n	$10s$	$27dias$	3660 séculos
3^n	$580min$		

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n	$10s$	$27dias$	3660 séculos
3^n	$580min$	38550 séculos	

[1] Dura/custa: Termo de comparação dos algoritmos

Comportamento assintótico

Comparação numérica da complexidade: Considere que cada operação ($n = 1$) dura/custa[1] de $0.0001s = 100\mu s$.

$T(n)$	20	40	60
n	$200\mu s$	$400\mu s$	$600\mu s$
$nlogn$	$900\mu s$	$2.1ms$	$3.5ms$
n^2	$4ms$	$16ms$	$36ms$
n^3	$80ms$	$640ms$	$2.16s$
2^n	$10s$	$27dias$	3660 séculos
3^n	$580min$	38550 séculos	$1.3 * 10^{14}$ séculos[2]

[1] Dura/custa: Termo de comparação dos algoritmos

[2] O tempo do universo medido em séculos é de aproximadamente $13.8 * 10^7$ séculos

Exercício 04

Qual a complexidade do algoritmo abaixo?

```
int maior(int n, int v[]) {
    int m = v[0];
    for (int i = 1; i < n; i++) {
        if (v[i] >= m) {
            m = v[i];
        }
    }
    return m;
}
```

Exercício 05

Qual a Complexidade dos algoritmos *f*, *g* e *h*?

```c
int f(int n) {
    int i, soma = 0;
    for (i = 1; i <= n; ++i)
        soma += 1;
    return soma;
}
int g(int n) {
    int i, soma = 0;
    for (i = 1; i <= n; ++i)
        soma += i + f(i);
    return soma;
}
int h(int n) {
    return f(n) + g(n);
}
```

Notações de complexidade

- Notação Θ

Notações de complexidade

- Notação Θ
- Notação O

Notações de complexidade

- Notação Θ
- Notação O
- Notação Ω

Notações de complexidade

- Notação Θ
- Notação O
- Notação Ω
- Notação o

Notações de complexidade

- Notação Θ
- Notação O
- Notação Ω
- Notação o
- Notação ω

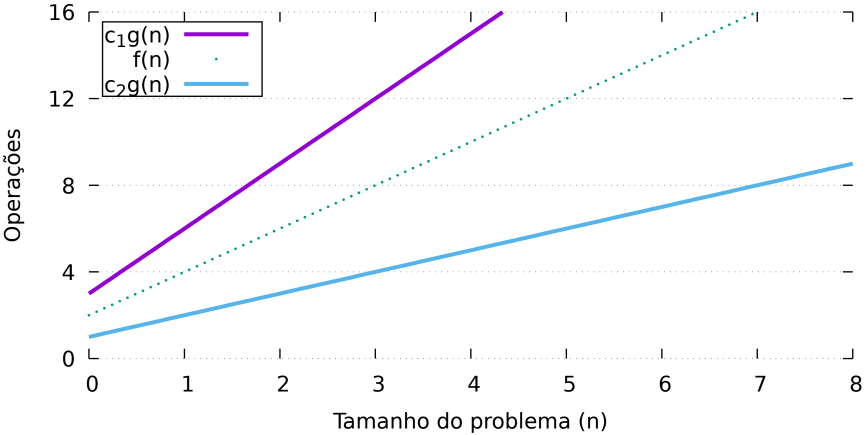
Notação Θ

Uma função $f(n)$ é dita como sendo $\Theta(g(n))$ se existem constantes positivas c_1, c_2 e n_0 para os quais:

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

para todo $n > n_0$.

Notação Θ



16
12
8
4
0
Operações
$c_1g(n)$
f(n)
$c_2g(n)$
0
1
2
3
4
5
6
7
8
Tamanho do problema (n)

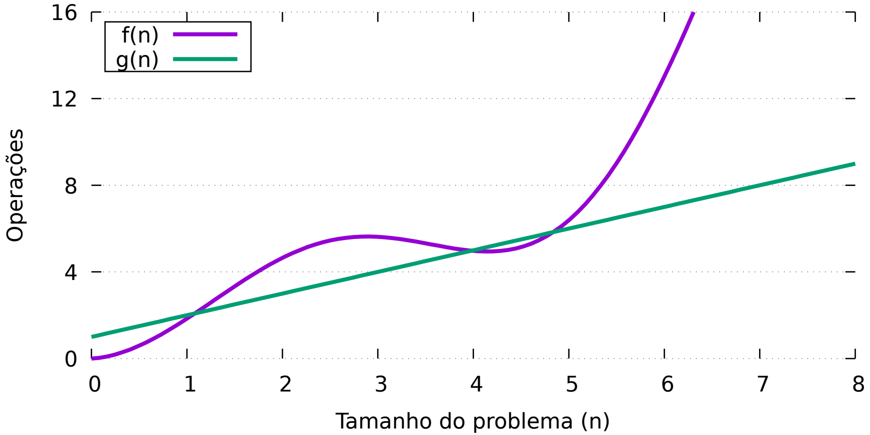
Notação Ω

Uma função $f(n)$ é dita como sendo $\Omega(g(n))$ se existem constantes positivas c e n_0 para os quais:

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

para todo $n > n_0$.

Notação Ω



Ordem de Complexidade

Retomando as ordens de complexidade:

- $O(1) \rightarrow$ contante
- $O(log_2 N) \rightarrow$ logaritmo
- $O(N) \rightarrow$ linear
- $O(N \ log_2 N)$
- $O(N^2) \rightarrow$ quadrática
- $O(N^3) \rightarrow$ cúbica
- $O(2^N) \rightarrow$ exponencial
- $O(N!) \rightarrow$ fatorial

Constante

- A complexidade independe do tamanho do problema;
- As instruções são executadas uma quantidade fixa de vezes.

Constante

- A complexidade independe do tamanho do problema;
- As instruções são executadas uma quantidade fixa de vezes.

```
void funcConstante(int *V) {
    if (V[0] == 0) {
        //Operacoes constantes
    } else {
        //Operacoes constantes
    }
}
```

Linear

- Alguma(s) operações fixas são executadas para cada elemento da entrada.

Linear

- Alguma(s) operações fixas são executadas para cada elemento da entrada.

```c
void funcLinear(int *V, int N) {
    for (int i = 0; i < N; i++) {
        //Operacoes constantes
    }
}
```

Logarítmico

- Surge em algoritmos nos quais o problema é subdividido em partes menores.

Logarítmico

- Surge em algoritmos nos quais o problema é subdividido em partes menores.

```
int pesquisaBinaria(int V[], int key, int N) {
    int inf = 0;
    int sup = N - 1;
    int meio;
    while (inf <= sup) {
        meio = (inf + sup / 2);
        if (key == V[meio])
            return meio;
        else {
            if (key < V[meio])
                sup = meio - 1;
            else
                inf = meio + 1;
        }
    }
    return -1;
}
```

Log Linear

- Acontece em algoritmos que são subdividido em problemas menores e depois é remontado.

Log Linear

- Acontece em algoritmos que são subdividido em problemas menores e depois é remontado.

```c
void merge(int inicio, int fim){
    if(inicio < fim){
        int meio = (inicio+fim)/2;
        merge(inicio, meio);
        merge(meio+1, fim);
        mesclar(inicio, meio,fim);
    }
}
```

Quadrático

- Normalmente as instruções são processados em duplos loops de tamanho proporcioanal ao problema.

Quadrático

- Normalmente as instruções são processados em duplos loops de tamanho proporcioanal ao problema.

```c
void ordBolha(int *a, int N) {
    for (int i = 0; i < N - 1; i++) {
        for (int j = 0; j < N - 1; j++) {
            if (a[j] > a[j + 1]) {
                swap(a, j, j + 1);
            }
        }
    }
}
```

Cúbico

Cúbico

- 3 loops.

Cúbico

- 3 loops.

```
void funcCub(int M, int N) {
    int dist[M][N];
    for (int k = 0; k < N; k++) {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                dist[i][j] =
                    min(dist[i][j],
                        dist[i][k] + dist[k][j]);
            }
        }
    }
```

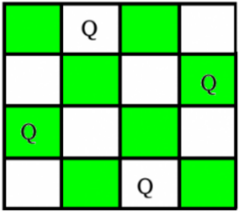
Exponencial

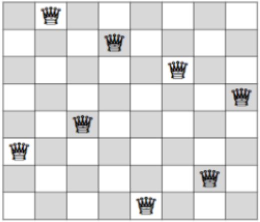
Exponencial

- Geralmente utilizado pelo algoritmo de **força bruta**.
- É a solução mais *trivial* do problema.
- A descoberta de uma chave criptográfica tem como solução inicial algoritmos dessa classe.

Exponencial

- Geralmente utilizado pelo algoritmo de **força bruta**.
- É a solução mais *trivial* do problema.
- A descoberta de uma chave criptográfica tem como solução inicial algoritmos dessa classe.





fatorial

[3] https://tspvis.com/

fatorial

- Também é complexidade para o algoritmo **força bruta**.
- Ocorre quando se testa todas as possíveis soluções das permutações de um problema.
- Ex.: Problema do Caixeiro Viajante.

[3] https://tspvis.com/

fatorial

- Também é complexidade para o algoritmo **força bruta**.
- Ocorre quando se testa todas as possíveis soluções das permutações de um problema.
- Ex.: Problema do Caixeiro Viajante.

Figura: Caixeiro Viajante[3]

[3] https://tspvis.com/

Pior, Melhor e Caso médio

Algoritmo simples de busca

```
#define N 1000000
int busca(int *A, int v) {
    for (int x = 0; x < N; x++) {
        if (A[x] == v) {
            return x;
        }
    }
    return -1;
}
```

Pior, Melhor e Caso médio

Algoritmo simples de busca

```
#define N 1000000
int busca(int *A, int v) {
    for (int x = 0; x < N; x++) {
        if (A[x] == v) {
            return x;
        }
    }
    return -1;
}
```

- Como avaliar esse algoritmo?

Pior, Melhor e Caso médio

Algoritmo simples de busca

```
#define N 1000000
int busca(int *A, int v) {
    for (int x = 0; x < N; x++) {
        if (A[x] == v) {
            return x;
        }
    }
    return -1;
}
```

- Como avaliar esse algoritmo?
- O desempenho depende apenas do tamanho de N?

Pior, Melhor e Caso médio

Algoritmo simples de busca

```
#define N 1000000
int busca(int *A, int v) {
    for (int x = 0; x < N; x++) {
        if (A[x] == v) {
            return x;
        }
    }
    return -1;
}
```

- Como avaliar esse algoritmo?
- O desempenho depende apenas do tamanho de N?
- O desempenho é modificado pelos valores de entrada?

Pior, Melhor e Caso médio

- **Pior caso** é a função que relaciona o tamanho da entrada n com o maior tempo possível para execução deste problema.
- **Melhor caso** é a função que relaciona o tamanho da entrada n com o menor tempo possível para execução deste problema.
- **Caso médio** é a função que relaciona o tamanho da entrada n com o tempo médio para execução deste problema. Para isso, é considerado uma *distribuição de probabilidade* das possíveis entradas.

Exemplo:

Qual o pior, melhor e o caso médio para a execução desse algoritmo? Quais suas complexidade?

Algoritmo simples de busca

```
#define N 1000000
int busca(int *A, int v) {
    for (int x = 0; x < N; x++) {
        if (A[x] == v) {
            return x;
        }
    }
    return -1;
}
```

Exemplo: Complexidade média

Considere:

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 * \frac{p}{n} + 2 * \frac{p}{n} + \ldots n * \frac{p}{n} = \sum_{i=1}^{n} i * \frac{p}{n} \tag{1}$$

- O custo do elemento não ser encontrado é:

$$(1-p)(n+1) \tag{2}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 * \frac{p}{n} + 2 * \frac{p}{n} + \ldots n * \frac{p}{n} = \sum_{i=1}^{n} i * \frac{p}{n} \tag{1}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 * \frac{p}{n} + 2 * \frac{p}{n} + \ldots n * \frac{p}{n} = \sum_{i=1}^{n} i * \frac{p}{n} \tag{1}$$

- O custo do elemento não ser encontrado é:

$$(1 - p)(n + 1) \tag{2}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 * \frac{p}{n} + 2 * \frac{p}{n} + \ldots n * \frac{p}{n} = \sum_{i=1}^{n} i * \frac{p}{n} \tag{1}$$

- O custo do elemento não ser encontrado é:

$$(n+1) \tag{2}$$

Exemplo: Complexidade média

Considere:

- $0 \leq p \leq 1$ é a probabilidade de v estar no vetor A;
- p/n é a probabilidade de v estar no vetor A na posição x.

Desta forma, podemos dizer que:

- O custo de encontrar um elemento é:

$$1 * \frac{p}{n} + 2 * \frac{p}{n} + \ldots n * \frac{p}{n} = \sum_{i=1}^{n} i * \frac{p}{n} \tag{1}$$

- O custo do elemento não ser encontrado é:

$$(1 - p)(n + 1) \tag{2}$$

Exemplo: Complexidade média

Somando 2 com 1, temos:

$$\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}$$

Exemplo: Complexidade média

Somando 2 com 1, temos:

$$
\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}
$$

- Considere $p = 1$ (busca bem sucedida), então o custo médio será de $\frac{n+1}{2}$.

Exemplo: Complexidade média

Somando 2 com 1, temos:

$$
\begin{aligned}
S_{medio} &= (1-p)(n+1) + \sum_{i=1}^{n} i * \frac{p}{n} \\
&= (1-p)(n+1) + \frac{p}{n}\sum_{i=1}^{n} i \\
&= (1-p)(n+1) + \frac{p}{n}\frac{n(n+1)}{2} \\
&= \frac{(n+1)(2-p)}{2}
\end{aligned}
$$

- Considere $p = 1$ (busca bem sucedida), então o custo médio será de $\frac{n+1}{2}$.
- Considere $p = 0$ (busca mal sucedida), então o custo médio será de $(n+1)$.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.
- A complexidade no **pior** caso é tão importante quanto por apresentar o pior cenário possível.

Considerações sobre complexidades

Pontos importantes:

- A complexidade **média** é de mais difícil obtenção: Dificuldade maior na análise.
- A complexidade no **pior** caso é tão importante quanto por apresentar o pior cenário possível.
- A complexidade no **melhor** caso não é tão relevante para análise de algoritmos.
- A complexidade do caso médio **não** é a média entre o pior caso e o melhor caso.