Algoritmos e Estruturas de Dados I

Análise de Complexidade de Algoritmos

versão 5.4

Fabiano Oliveira
fabiano.oliveira@ime.uerj.br

Complexidade

Análise de Complexidade

- A análise de ***complexidade*** de algoritmos descreve a ***eficiência*** dos algoritmos

- Como ***medir*** a eficiência de algoritmos?

Análise de Complexidade

- Compare com outras grandezas:

  "a **massa** do objeto é (igual a) **10 kg**"
  "a **altura** do menino é (igual a) **1,40 m**"

- Queremos dizer:

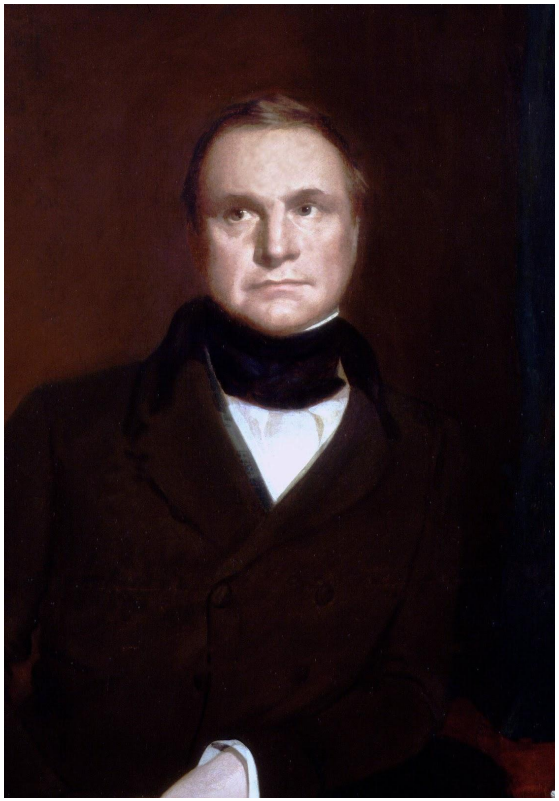
  "a **complexidade** do algoritmo é **???**"

Análise de Complexidade

- A preocupação com a complexidade (isto é, a eficiência) é antiga — em verdade, nasceu conjuntamente com a própria noção de computar por meios mecânicos:

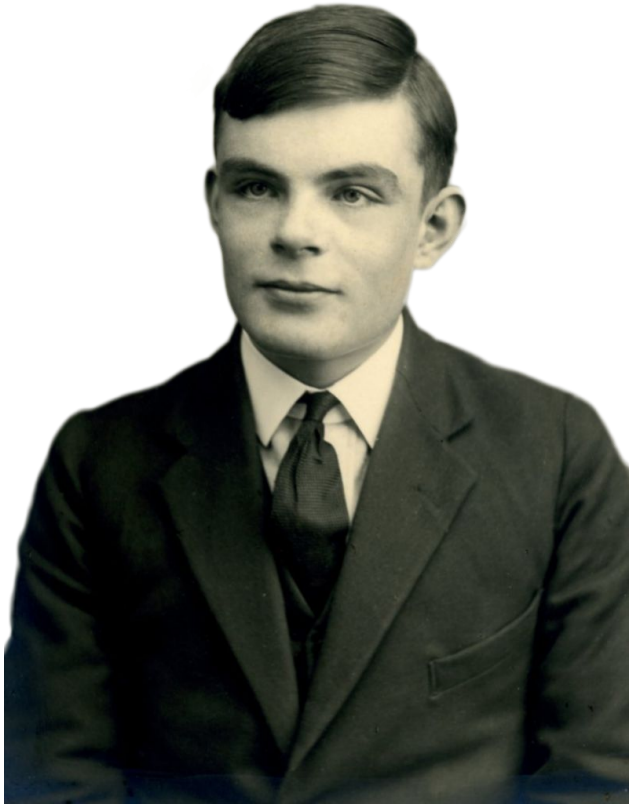
Análise de Complexidade

"(....) As soon as analytic engine exists, it will necessarily guide the future course of science. Whenever any result is sought by its aid, the question will raise – By what means of calculation can these results be arrived at by this machine in the shortest time? (...)". Charles Babbage



Análise de Complexidade

"(....) it's convenient to have a measure of the amount of work involved in a computing process, even if it has to be a very crude one. We may count up the number of things that various times at various elementary operations are applied in the whole process. (...)". Alan Turing



Análise de Complexidade

- ***O quê*** medir?

Análise de Complexidade

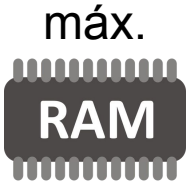
- ***O quê*** medir?
  - **Tempo**: tempo de execução do algoritmo
  - **Espaço**: quantidade de memória máxima empregada durante execução

Análise de Complexidade

Complexidade

de tempo:

de espaço:



Análise de Complexidade

- ***Como*** medir?

Análise de Complexidade

- ***Como*** medir?
  - Empiricamente
  - Analiticamente

Análise de Complexidade

- **Empiricamente**:
  - Organizar um conjunto de entradas para o algoritmo, cada uma exigindo um nível diferente de consumo do recurso sendo estudado
  - Executar e medir do consumo do recurso
  - Apresentar de forma tabular e por meio de gráficos os resultados do experimento
  - (*Opcional*) Sugerir a função que descreve os pontos do gráfico (uma reta? uma parábola? outra?)

Análise de Complexidade

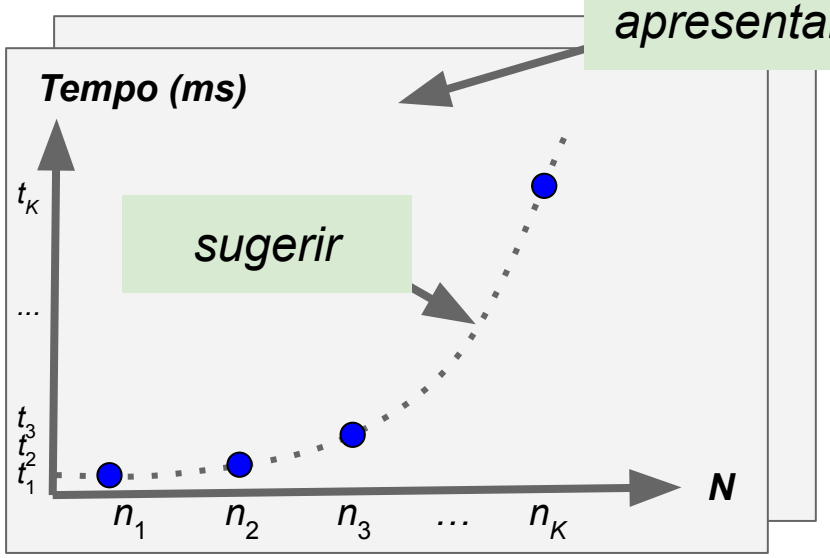
- **Exemplo: ordenação de vetores**
  - V_1 = […]
    V_2 = [……...]
    V_3 = [……...]
    …
    V_K = [……..……..]

N elementos

organizar

executar & medir

apresentar



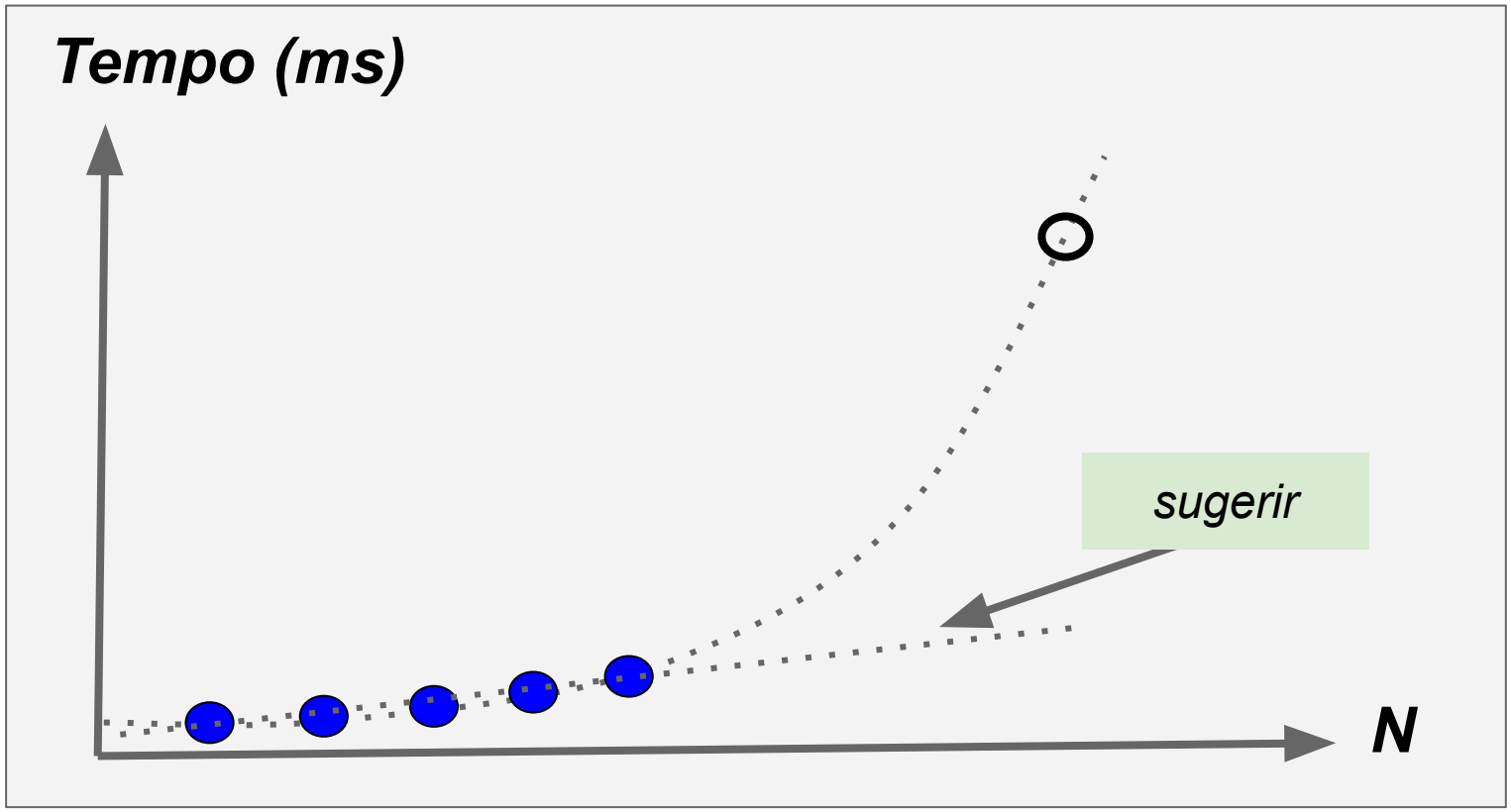
#	N	Tempo (em ms)	Espaço (em MB)
1	n_1	t_1	...
2	n_2	t_2	...
3	n_3	t_3	...
...	....	...	...
K	n_K	t_K	...

Análise de Complexidade

- **Desvantagens**
  - Qualidade da análise **altamente dependente** da amostra de entradas. Para ser relevante, deve ser muito bem **justificada** a escolha das entradas
  - **Impossibilidade** de assegurar que a função sugerida corresponde à real

Análise de Complexidade

- **Desvantagens**



Análise de Complexidade

- **Esta disciplina NÃO empregará análise empírica**

Análise de Complexidade

- **Analiticamente**:
  - Estudar o algoritmo
  - Contabilizar o **número de passos** (complexidade de tempo) e de **células de memória** (complexidade de espaço) que o algoritmo requer para sua execução
    - Note que, normalmente, tal contabilidade deve resultar em uma função das variáveis de entrada

Análise da Complexidade de Tempo

Análise de Complexidade

- Considere o seguinte algoritmo:

```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro

    para i ← 1 até N faça
        para j ← 1 até i faça
            t ← t + j

    retornar t
```

Análise de Complexidade

- Considere o seguinte algoritmo:

```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro

    para i ← 1 até N faça
        para j ← 1 até i faça
            t ← t + j                    = 1 passo

    retornar t
```

Análise de Complexidade

- Considere o seguinte algoritmo:

```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro

    para i ← 1 até N faça
        para j ← 1 até i faça
            t ← t + j

    retornar t
```

= *i* passos

= 1 passo

Análise de Complexidade

- Considere o seguinte algoritmo:

```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro

    para i ← 1 até N faça          = 1+2+...+N = N(N+1)/2 passos
        para j ← 1 até i faça      = i passos
            t ← t + j              = 1 passo

    retornar t
```

Análise de Complexidade

- Considere o seguinte algoritmo:

```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro
                                                    = N(N+1)/2 + 1 passos
    para i ← 1 até N faça                = 1+2+...+N = N(N+1)/2 passos
        para j ← 1 até i faça                        = i passos
            t ← t + j                                = 1 passo

    retornar t
```

Análise de Complexidade

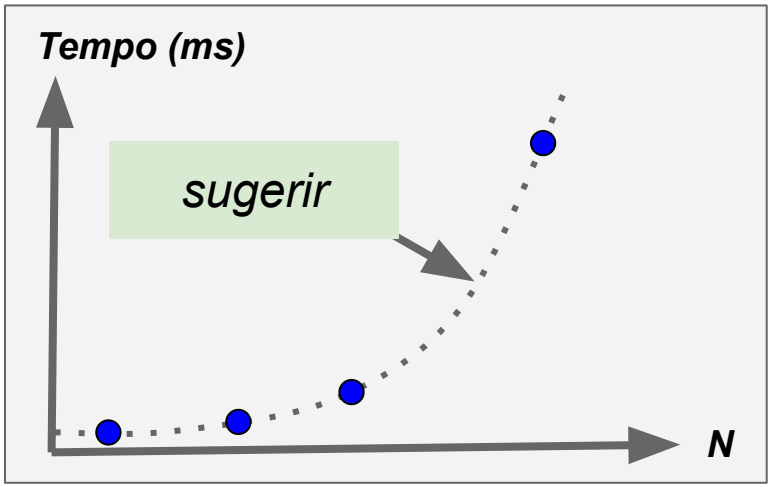
- Considere o seguinte algoritmo:

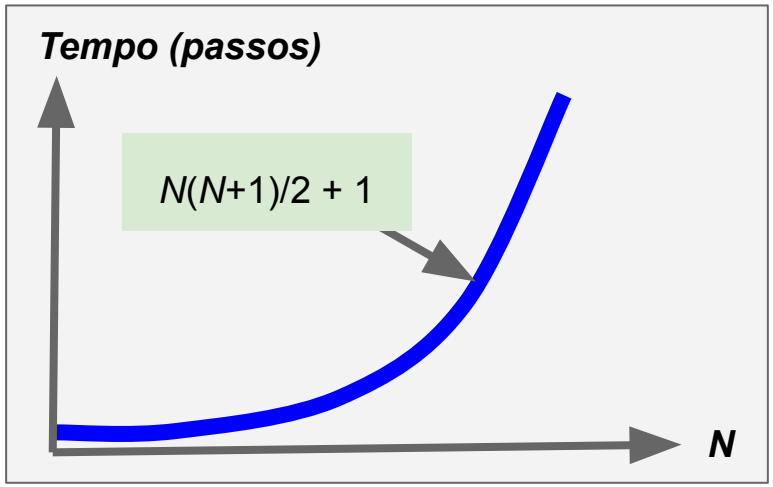
```
função Calcular(B[], N: Inteiro): Inteiro
    var i, j, t: Inteiro
                                                    = N(N+1)/2 + 1 passos
    para i ← 1 até N faça                = 1+2+...+N = N(N+1)/2 passos
        para j ← 1 até i faça                        = i passos
            t ← t + j                                = 1 passo

    retornar t
```

- ∴ a complexidade de tempo é $N(N+1)/2 + 1$

Análise de Complexidade

- Compare as complexidades:



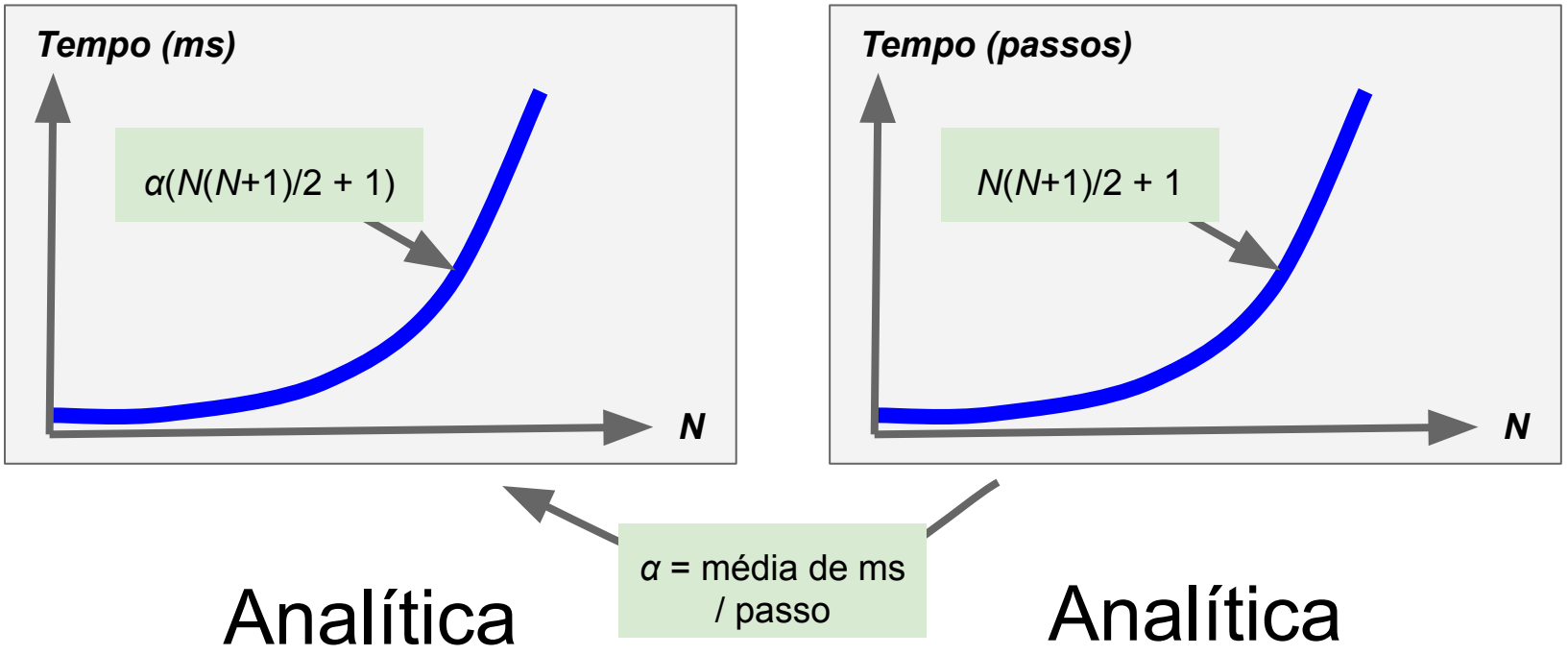


Empírica

Analítica

Análise de Complexidade

- Compare as complexidades:



Análise de Complexidade

- **Desvantagens**
  - É necessário ter acesso à descrição do algoritmo (seu "código-fonte")
  - **Potencialmente** de difícil obtenção
    - Existem algoritmos para os quais sua complexidade de tempo exata não é conhecida

Complexidade Assintótica

Análise de Complexidade

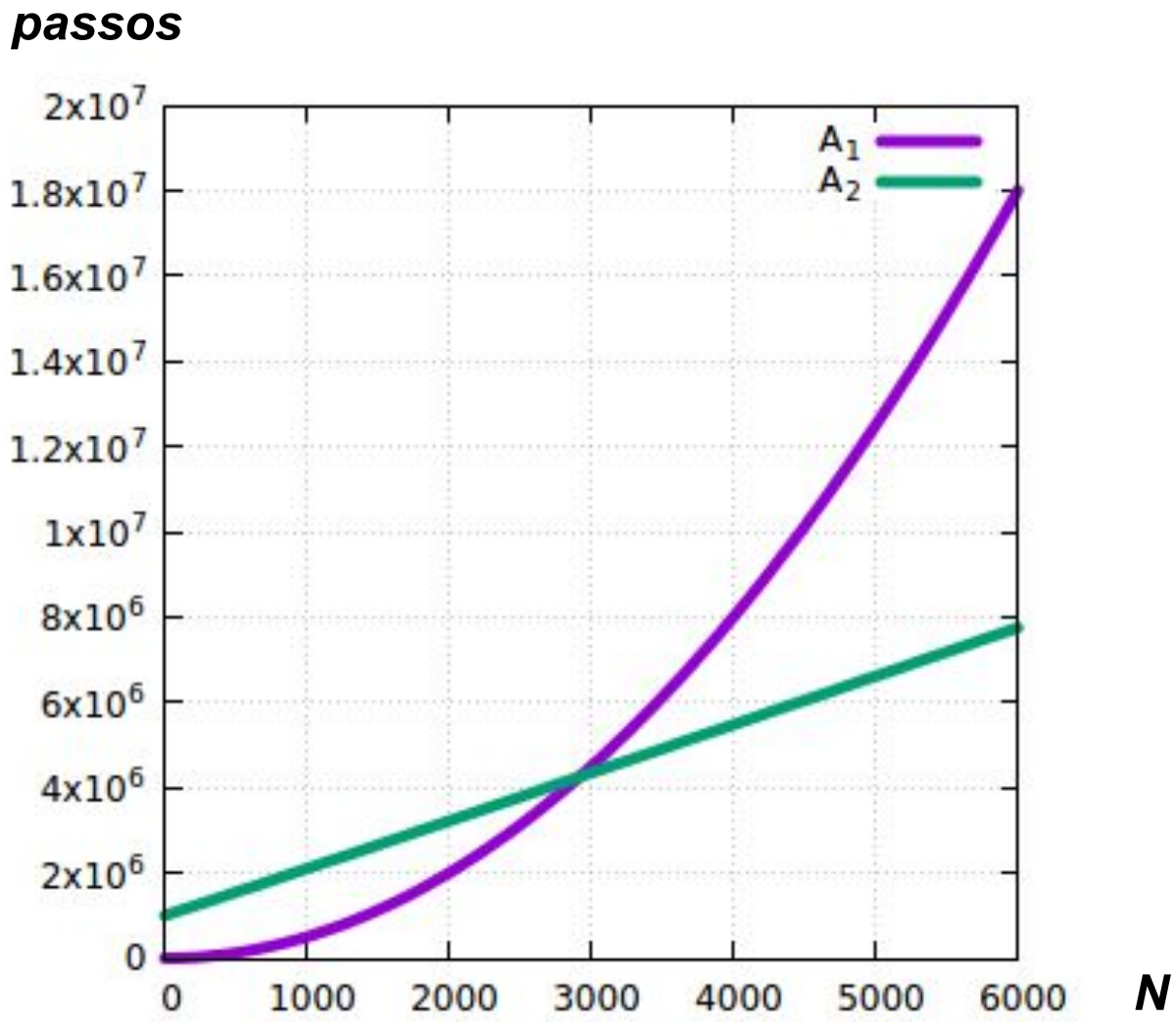
- Considere a existência de outro algoritmo A_2 que é equivalente ao algoritmo anterior A_1

- Suponha que a complexidade de tempo de A_2 seja determinada como sendo

  $10N \log N + 1000N + 1000000$

- Qual dos algoritmos é mais eficiente em tempo?

Análise de Complexidade



Análise de Complexidade

- O grau do polinômio é o que determina o valor do polinômio a medida que a variável cresce!

- Esta **complexidade** é chamada de **assintótica**

- Ela é um dos critérios mais utilizados para comparação de eficiência de algoritmos

Análise de Complexidade

- Motivado por isto, existe uma notação especial que evidencia a classe de crescimento de uma função —
  a **notação assintótica**

Análise de Complexidade

- **Intuição** da notação assintótica**:**

  Suponha uma função T(N) desconhecida.

Termo de maior crescimento	Notação	Funções T(N) que satisfazem notação	Funções T(N) que NÃO satisfazem notação
Igual a N^2	$\theta(N^2)$	$\frac{1}{2} N^2 + 10N$; $100N^2$	$10N$; $N^3 + 2N^2$; $N \log N$
No máximo N^2	$O(N^2)$	$100N^2$; 100; $N \log N$	$N^3 + 2N^2$; $N^2 \lg N$
No mínimo N^2	$\Omega(N^2)$	$100N^2$; $N^3 + 2$; $N^2 \lg N$	$10N$; 100; $N \log N$
Menor que N^2	$o(N^2)$	$10N$; 100; $N \log N$	$100N^2$; $N^3 + 2N^2$; $N^2 \lg N$
Maior que N^2	$\omega(N^2)$	$N^3 + 2N^2$; $N^2 \lg N$	$100N^2$; $10N$; 100; $N \log N$

- **Formalidade** da notação assintótica, a seguir

Análise de Complexidade

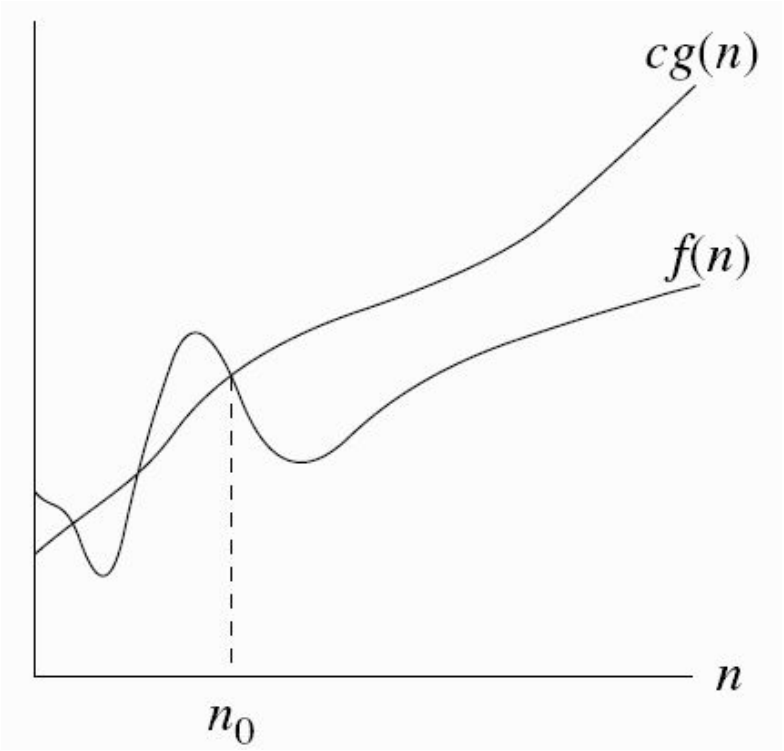
- Notação O

  $O(g(n)) = \{ h(n) \mid \exists\ c, n_0$ tais que
  $0 \le h(n) \le cg(n), \ \forall\ n \ge n_0 \}$

  Pode-se denotar
  $f(n) \in O(g(n))$ por
  $f(n) = O(g(n))$

  Se f(n)/g(n) admitir limite,

  $\lim_{n\to\infty} f(n)/g(n) \neq \infty \Rightarrow$
  $f(n) = O(g(n))$



Análise de Complexidade

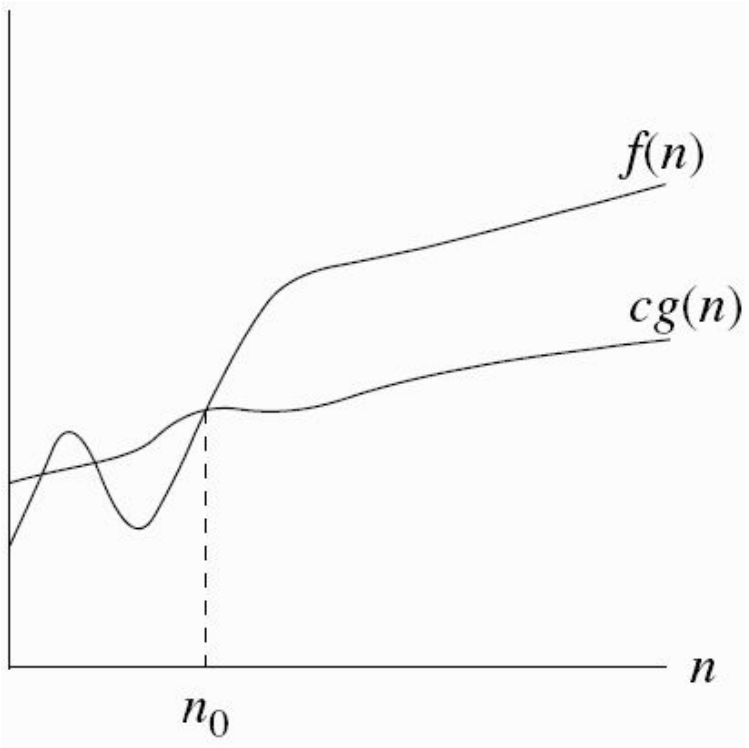
- Notação Ω

  $\Omega(g(n)) = \{ h(n) \mid \exists\ c > 0, n_0$ tais que
  $0 \le cg(n) \le h(n), \ \forall\ n \ge n_0 \}$

  Pode-se denotar
  $f(n) \in \Omega(g(n))$ por
  $f(n) = \Omega(g(n))$

  Se f(n)/g(n) admitir limite,

  $\lim_{n\to\infty} f(n)/g(n) \neq 0 \Rightarrow$
  $f(n) = \Omega(g(n))$



Análise de Complexidade

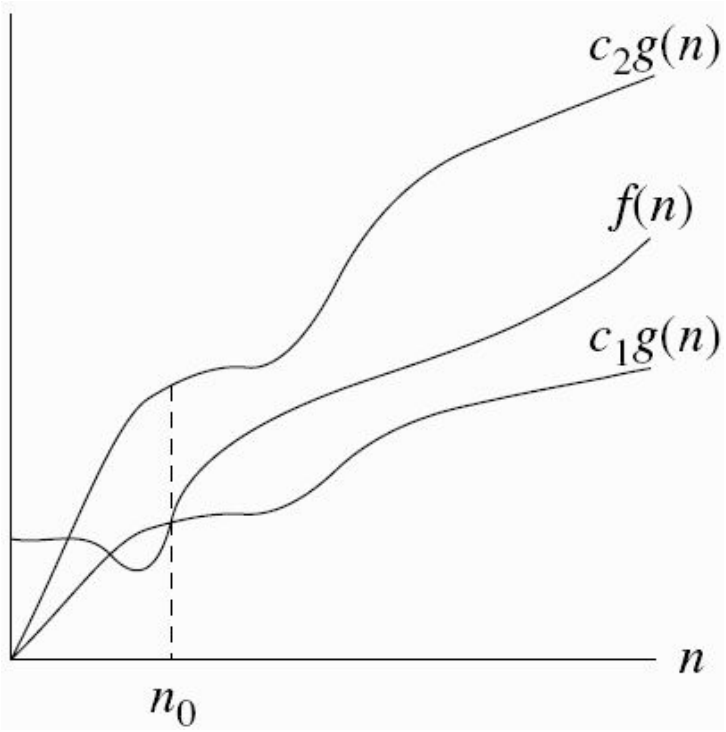
- Notação θ

  $\theta(g(n)) = \{ h(n) \mid \exists\ c_1 > 0, c_2, n_0$ tais que
  $0 \le c_1 g(n) \le h(n) \le c_2 g(n), \ \forall\ n \ge n_0 \}$

  Pode-se denotar
  $f(n) \in \theta(g(n))$ por
  $f(n) = \theta(g(n))$

  Se f(n)/g(n) admitir limite,

  $\lim_{n \to \infty} f(n)/g(n) \neq 0, \infty \Rightarrow$
  $f(n) = \theta(g(n))$



Análise de Complexidade

- Notação o

  $o(g(n)) = \{ h(n) \mid \forall\ c > 0,\ \exists\ n_0$ tal que
  $0 \leq h(n) < cg(n),\ \forall\ n \geq n_0 \}$

  Pode-se denotar $f(n) \in o(g(n))$ por $f(n) = o(g(n))$

  Se f(n)/g(n) admitir limite,

  $\lim_{n \to \infty} f(n)/g(n) = 0 \quad \Rightarrow$
  $f(n) = o(g(n))$

Análise de Complexidade

- Notação ω

  $\omega(g(n)) = \{ h(n) \mid \forall\ c > 0,\ \exists\ n_0$ tal que
  $0 \leq cg(n) < h(n),\ \forall\ n \geq n_0 \}$

  Pode-se denotar $f(n) \in \omega(g(n))$ por $f(n) = \omega(g(n))$

  Se f(n)/g(n) admitir limite,

  $\lim_{n \to \infty} f(n)/g(n) = \infty \Rightarrow$
  $f(n) = \omega(g(n))$

Análise de Complexidade

- Assim, temos que
  - o algoritmo A_1 possui complexidade de tempo de $N\,(N + 1)/2 + 1 = \boldsymbol{\theta(N^2)}$
  - o algoritmo A_2 possui complexidade de tempo de $10N \log N + 1000N + 1000000 = \boldsymbol{\theta(N \log N)}$

- Portanto, como $\boldsymbol{N \log N = o(N^2)}$ o algoritmo A_2 possui uma melhor **complexidade assintótica**

Análise de Complexidade

- Contar as instruções pode ser um processo laborioso. Por outro lado...
- o que normalmente importa em relação ao tempo de execução de um algoritmo é seu crescimento assintótico (determinado pelo termo de maior grau do polinômio)

- **Pergunta natural**: há um processo prático para se chegar à complexidade sem determinar a função que fornece o número exato de passos?

Processo Prático

(para algoritmos não-recursivos)

Análise de Complexidade

- **Passo 1**: Separe o programa em blocos. Um bloco é um trecho do código que atende a alguma das definições abaixo:
  - Um comando
  - Um comando condicional incluindo os blocos correspondentes aos trechos *então*, *senão se*, e *senão*
  - Uma sequência de blocos
  - Uma repetição, incluindo o bloco de seu corpo

Análise de Complexidade

```
procedimento EscreverMochilaMáxima(val w[], v[], N, M: Inteiro)
    var MM[0..N, 0..M]: Inteiro, Escolha[1..N, 1..M]: Lógico
    MM[0, m] ← 0, m ∈ {0,...,M}
    para i ← 1 até N faça
        para m ← 1 até M faça
            se w[i] > m então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão se MM[i-1, m] > MM[i-1, m-w[i]] + v[i] então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão
                MM[i, m], Escolha[i, m] ← MM[i-1, m-w[i]] + v[i], V
    m ← M
    para i ← N até 1 passo -1 faça
        se Escolha[i, m] então
            escrever [i]
            m ← m - w[i]
```

Análise de Complexidade

```
procedimento EscreverMochilaMáxima(val w[], v[], N, M: Inteiro)
    var MM[0..N, 0..M]: Inteiro, Escolha[1..N, 1..M]: Lógico
    MM[0, m] ← 0, m ∈ {0,...,M}
    para i ← 1 até N faça
        para m ← 1 até M faça
            se w[i] > m então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão se MM[i-1, m] > MM[i-1, m-w[i]] + v[i] então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão
                MM[i, m], Escolha[i, m] ← MM[i-1, m-w[i]] + v[i], V
    m ← M
    para i ← N até 1 passo -1 faça
        se Escolha[i, m] então
            escrever [i]
            m ← m - w[i]
```

Análise de Complexidade

- **Passo 2**: Determinação "bottom-up" das complexidades dos blocos

  Seja B um bloco cuja complexidade falta determinar e tal que todas as complexidades dos blocos contidos em B já foram determinadas.

  A complexidade de B é determinada conforme a natureza do bloco:

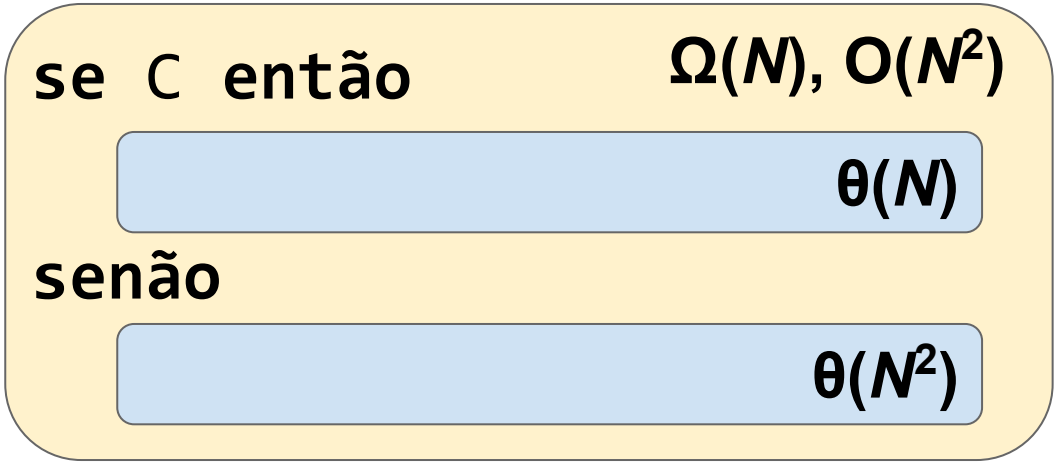
Análise de Complexidade

- **Um comando:**

Comando	Complexidade de Tempo
`x ← x + 1`	θ(1)
`x ← x + y`	θ(1)
`A[i] ← 1`	θ(1)
`A[1..N] ← 1`	θ(*N*)
`A[i] ← máx{ A[j] | 1 ≤ j ≤ N }`	θ(*N*)
`A[i,j] ← 0,` `  para todo 1 ≤ i,j ≤ N`	θ(N^2)
`m ← Calcular(A,N)`	complexidade de tempo de `Calcular(A,N)`

Análise de Complexidade

- **Um condicional:**

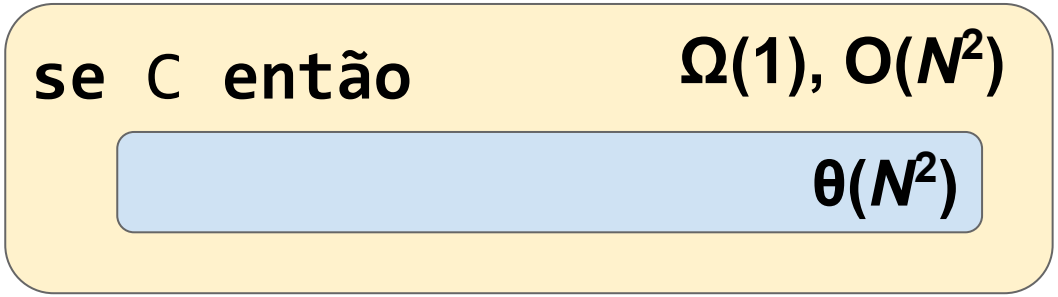


Análise de Complexidade

- **Um condicional:**

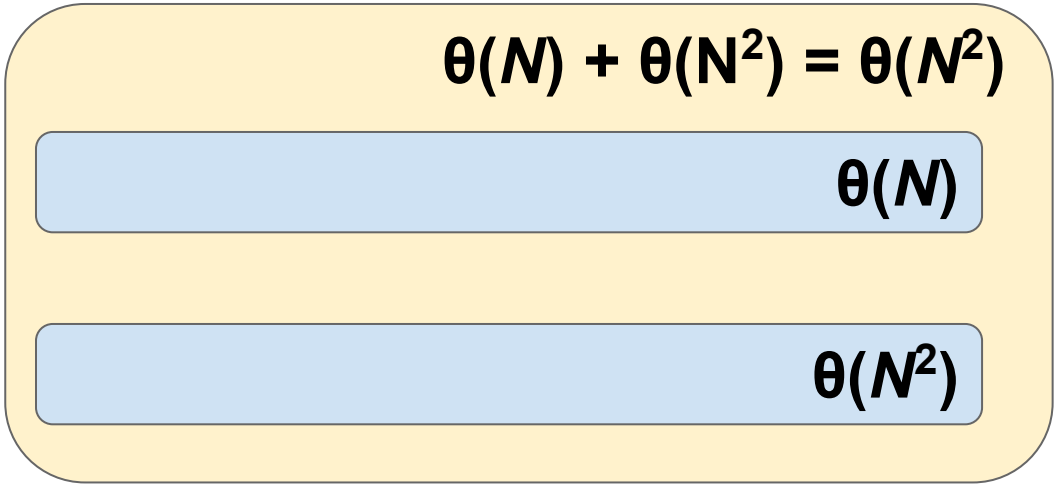
Análise de Complexidade

- **Um condicional:**



Análise de Complexidade

- **Uma sequência:**



Análise de Complexidade

- **Uma sequência:**

$O(N^3) + \theta(N^2) = \Omega(N^2), O(N^3)$

$O(N^3)$

$\theta(N^2)$

Análise de Complexidade

- **Uma iteração:**

```
para i ← 1 até N faça                    θ(N³)
    ...                                  θ(N²)
```

Análise de Complexidade

- **Uma iteração:**

```
para i ← 1 até N faça
```

$\Omega(N^2)$, $O(N^3)$

$\Omega(N)$, $O(N^2)$

Análise de Complexidade

- **Uma iteração:**

```
para i ← 1 até N faça
```

$\theta(1+2+...+N) = \theta(N^2)$

$\theta(i)$

Análise de Complexidade

- **Uma iteração:**

```
para i ← 1 até N faça                    Ω(N²)
    θ(i)
```

No mínimo, a complexidade de:

```
para i ← N/2 até N faça    θ(N/2 × N/2) = θ(N²)
    θ(N/2)
```

Análise de Complexidade

- **Uma iteração:**

```
para i ← 1 até N faça                    O(N²)
    θ(i)
```

No máximo, a complexidade de:

```
para i ← 1 até N faça                    θ(N × N) = θ(N²)
    θ(N)
```

Análise de Complexidade

- **Uma iteração:**

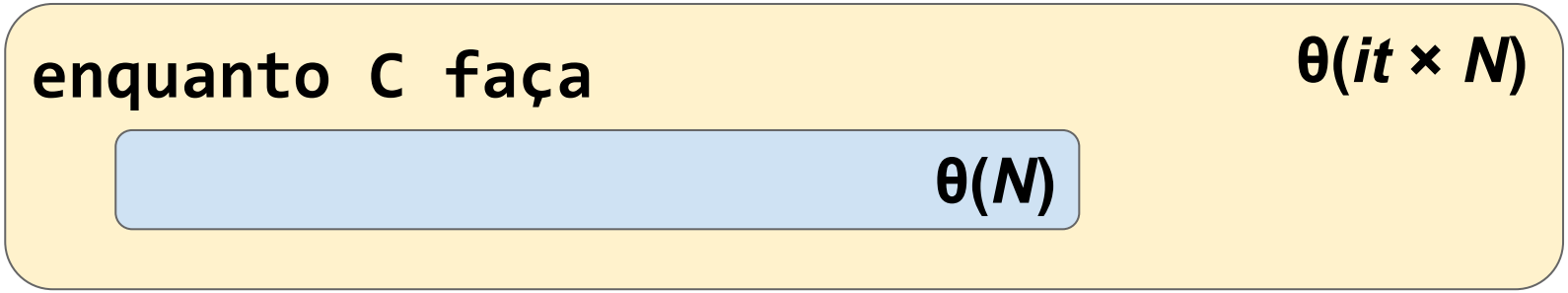
```
para i ← 1 até N faça
```

$\Omega(N^2)$, $O(N^2) = \theta(N^2)$

$\theta(i)$

Análise de Complexidade

- **Uma iteração:**



onde *it* é o número de iterações necessárias até que C se torne falso. Tal valor deve ser determinado analisando-se o algoritmo.

Análise de Complexidade

- **Uma iteração:**

```
i ← 1                                          θ(N²)
enquanto i ≤ N faça
    [                                  ]  θ(N)
    i ← i + 1
```

it = *N*

Análise de Complexidade

- **Uma iteração:**

```
i ← 1                                          θ(N²)
enquanto i ≤ N faça
    [                                  ]  θ(N)
    i ← i + 2
```

it = N/2

Análise de Complexidade

- **Uma iteração:**

```
i ← 1                                          θ(N log N)
enquanto i ≤ N faça
    <bloco>                                    θ(N)
    i ← i * 2
```

$it = \log N$

Análise de Complexidade

```
procedimento EscreverMochilaMáxima(val w[], v[], N, M: Inteiro)
    var MM[0..N, 0..M]: Inteiro, Escolha[1..N, 1..M]: Lógico
    MM[0, m] ← 0, m ∈ {0,...,M}
    para i ← 1 até N faça
        para m ← 1 até M faça
            se w[i] > m então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão se MM[i-1, m] > MM[i-1, m-w[i]] + v[i] então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F
            senão
                MM[i, m], Escolha[i, m] ← MM[i-1, m-w[i]] + v[i], V
    m ← M
    para i ← N até 1 passo -1 faça
        se Escolha[i, m] então
            escrever [i]
            m ← m - w[i]
```

Análise de Complexidade

θ(MN)

```
procedimento EscreverMochilaMáxima(val w[], v[], N, M: Inteiro)
    var MM[0..N, 0..M]: Inteiro, Escolha[1..N, 1..M]: Lógico
    MM[0, m] ← 0, m ∈ {0,...,M}                                   θ(M)
    para i ← 1 até N faça                                          θ(MN)
        para m ← 1 até M faça                                      θ(M)
            se w[i] > m então                                      θ(1)
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F            θ(1)
            senão se MM[i-1, m] > MM[i-1, m-w[i]] + v[i] então
                MM[i, m], Escolha[i, m] ← MM[i-1, m], F            θ(1)
            senão
                MM[i, m], Escolha[i, m] ← MM[i-1, m-w[i]] + v[i], V   θ(1)
    m ← M                                                          θ(1)
    para i ← N até 1 passo -1 faça                                 θ(N)
        se Escolha[i, m] então                                     θ(1)
            escrever [i]                                           θ(1)
            m ← m - w[i]                                           θ(1)
```

Análise de Complexidade

- Exercício: analisar a complexidade de tempo

```
procedimento Ordenar(B[], N: Inteiro)
    var i, j, t: Inteiro
    para i ← N até 1 passo -1 faça
        para j ← 1 até i-1 faça
            se B[j] > B[j+1] então
                t ← B[j]
                B[j] ← B[j+1]
                B[j+1] ← t
```

Complexidade de Tempo de Pior Caso, Melhor Caso e Caso Médio

Análise de Complexidade

- Suponha que *N* seja a variável que determina o tempo de execução de um algoritmo *A*. Seja *U* o conjunto de entradas para algum valor de *N* e T(*E*) o número de passos em que o algoritmo executa com entrada $E \in U$. Classificamos as análises de complexidade como
  - **de pior caso:** $\max \{ T(E) : E \in U \}$
  - **de melhor caso:** $\min \{ T(E) : E \in U \}$
  - **de caso médio:** $\sum \{ p(E)T(E) : E \in U \}$
    onde p(*E*) é a probabilidade de ocorrência da entrada E
  - **amortizada:** número máximo de passos em *n* execuções consecutivas de A dividido por *n*

Análise da Complexidade de Espaço

Análise de Complexidade

- Como medimos complexidade de espaço?
  - **Espaço Auxiliar**: memória utilizada internamente ao algoritmo, adicional à entrada do algoritmo
  - **Espaço Total**: memória auxiliar mais a memória para guardar a entrada
  - Como todo algoritmo precisa armazenar a entrada, usualmente é empregado o espaço auxiliar como critério de eficiência de espaço entre algoritmos

Análise de Complexidade

- Como medimos complexidade de espaço?

Tipo de Variável	Complexidade de Espaço
Escalar Primitiva (Inteiro, Real, Lógico, DataHora, Caracter, etc.)	$\theta(1)$
Vetor/Matriz A, cada elemento de tipo X	$\theta(\|A\|) \cdot$ <Espaço de X>
Estrutura E, com campos de tipos $T_1, T_2, \ldots, T_K$	$\sum$ { <Espaço de T_i> : i = 1,...,K }
Ponteiro para tipo qualquer tipo (Nota: Uma variável passada por referência para uma rotina conta, no escopo desta rotina, como ponteiro!)	$\theta(1)$

Análise de Complexidade

- Análise de Espaço:

```
procedimento Imprime(ref L[]: Tipo)
```

O vetor L está sendo passado por referência e, portanto, foi alocado fora da rotina Imprime. Logo, o espaço de L deve ser contabilizado na rotina que faz sua alocação. O espaço criado para passar L por parâmetro é o tamanho apenas de um ponteiro, que é de espaço constante (não depende de |L|).

```
procedimento Imprime(L[]: Tipo)
```
, equivalente à
```
procedimento Imprime(val L[]: Tipo)
```

L está sendo passado por valor e, portanto, o vetor original está sendo copiado para o procedimento Imprime. Logo, devemos contabilizar $\theta(|L|) \cdot$ <Espaço Tipo> em espaço para a entrada.

Análise de Complexidade

- Análise de Espaço:

  Alguns problemas envolvendo espaço se preocupam **além do aspecto assintótico**, precisando que se leve em consideração **valores específicos para o espaço** ocupado. Neste caso, as premissas empregadas neste material, normalmente válidas para as linguagens de programação modernas, serão as seguintes:

Análise de Complexidade

- Análise de Espaço:

1. 1 B (byte) = 8 bits

2. 1 valor "Lógico" = 1 bit; 1 valor "Inteiro" = 4 B; 1 valor caractere = 1 B;

3. Como 4 B = 32 bits, e como com tais 32 bits é possível enumerar 2^{32} configurações distintas, pode-se representar inteiros num intervalo contíguo de $\mathbb{Z}$ com ≈ 4 bilhões de elementos. Em geral, assumimos que este intervalo é aquele entre 0 e 2^{32}-1 (≈ 4 bilhões) ou ainda entre -2^{31} (≈ -2 bilhões) e 2^{31}-1 (≈ 2 bilhões).

4. 1 KB = $2^{10} \approx 10^3$ bytes; 1 MB = $2^{20} \approx 10^6$ bytes; 1 GB = $2^{30} \approx 10^9$ bytes

Análise de Complexidade de Tempo em Algoritmos Recursivos

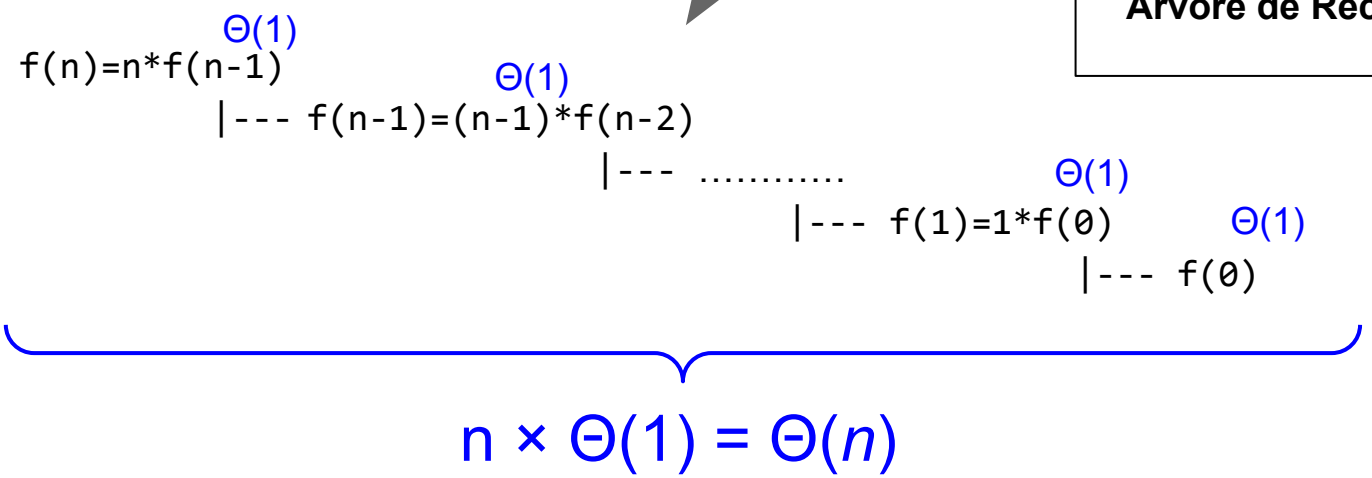
Método #1:
Árvore de Recursão

Recursão

```
função f(n: Inteiro): Inteiro
    //retorna n!
    se n = 0 então
        retornar 1
    senão
        retornar n*f(n-1)
```

Complexidade de Tempo: soma do número de passos de cada nó

Árvore de Recursão



$$n \times \Theta(1) = \Theta(n)$$

Método #2:
Relações de Recorrência

Recursão

- Seja $T(n)$ a complexidade de tempo da função recursiva:

$$T(n) = T(n_1) + g(n), \text{ com } n_1 < n$$

- Como $T(n_1) = T(n_2) + g(n_1)$, com $n_2 < n_1$,

$$T(n) = T(n_2) + g(n_1) + g(n)$$

Recursão

- Como $T(n_2) = T(n_3) + g(n_2)$, com $n_3 < n_2$,

$$T(n) = T(n_3) + g(n_2) + g(n_1) + g(n)$$

- Agora, tentamos generalizar a expressão da função da *i*-ésima recorrência,

$$T(n) = T(n_i) + g(n_{i-1}) + \ldots + g(n_1) + g(n)$$

Recursão

- Em seguida, descobrimos para qual valor de i vale que n_i é um caso base. Digamos que tal valor seja para $i = b$. Finalmente, determinamos a forma fechada do somatório

$$T(n) = 1 + g(n_{b-1}) + \ldots + g(n_1) + g(n)$$

Método #3:
Teorema Mestre

Recursão

- Quando a função de recorrência é

  $T(n) = a\,T(n/b) + f(n)$, com a,b constantes

  aplicar ***Teorema-Mestre***

Análise de Complexidade

```
procedimento AlgRecursivo(Dados[], N: Inteiro)
    se N = 0 então   // ou N = 1, ou …
        ....
    senão
        ...
        AlgRecursivo(SubDados1, N1)   // N1 < N
        ...
        AlgRecursivo(SubDados2, N2)   // N2 < N
        ...
        AlgRecursivo(SubDadosa, Na)   // Na < N
```

Caso Base

Tamanho do Problema

Caso Geral

Análise de Complexidade

```
procedimento AlgRecursivo(Dados[], N: Inteiro)
    se N = 0 então
        ....
    senão
        ...
        AlgRecursivo(SubDados1, N1) // N1 < N
        ...
        AlgRecursivo(SubDados2, N2) // N2 < N
        ...
        AlgRecursivo(SubDadosa, Na) // Na < N
```

θ(1)

f(*N*)

T(*N*)

Contagem de número de passos em vermelho

Análise de Complexidade

- Análise de Complexidade de Tempo:

$$T(N) = \begin{cases} \theta(1), \text{ se } N = 0 \text{ // ou N = 1, ou ...} \\ \sum \{ T(N_i) : 1 \leq i \leq a \} + f(N), \text{ c.c.} \end{cases}$$

Resolver uma equação de recorrência!

Análise de Complexidade

- Exemplos:
  - Ordenação BubbleSort
  - Ordenação InsertionSort
  - Ordenação SelectionSort

Análise de Complexidade

- Caso particular importante:

Todos os problemas possuem o mesmo tamanho!

Análise de Complexidade

```
procedimento AlgRecursivo(Dados[], N: Inteiro)
    se N = 0 então
        ....
    senão
        ...
        AlgRecursivo(SubDados1, N/b)
        ...
        AlgRecursivo(SubDados2, N/b)
        ...
        AlgRecursivo(SubDadosa, N/b)
        ...
```

θ(1)

f(*N*)

T(*N*)

Contagem de número de passos em vermelho

Análise de Complexidade

- Análise de Complexidade de Tempo:

$$T(N) = \begin{cases} \theta(1), \text{ se } N = 0 \\ a\,T(N/b) + f(N), \text{ se } N > 0 \end{cases}$$

Análise de Complexidade

Teorema ("Teorema Mestre"):
Na recorrência $T(N) = a\,T(N/b) + f(N)$:

1) Caso $f(N) = O(N^c)$ e $c < \log_b a$:
   $T(N) = \theta(N^{\log_b a})$

2) Caso $f(N) = \theta(N^c \log^k N)$, $k \geq 0$ e $c = \log_b a$:
   $T(N) = \theta(N^c \log^{k+1} N)$

3) Caso $f(N) = \Omega(N^c)$, $f(N) = O(N^k)$ e $c > \log_b a$:
   $T(N) = \theta(f(N))$

Análise de Complexidade

- Exemplo 1:

$T(N) = 5\,T(N/2) + \theta(N^2)$

Como $f(N) = O(N^2)$ e $2 < \log_2 5$,
Caso 1 é aplicado, resultando em
$T(N) = \theta(N^{\log_2 5})$

Análise de Complexidade

- Exemplo 2:

$T(N) = 27\ T(N/3) + \theta(N^3 \log N)$

Como $f(N) = \theta(N^3 \lg N)$ e $3 = \log_3 27$,
Caso 2 é aplicado, resultando em
$T(N) = \theta(N^3 \log^2 N)$

Análise de Complexidade

- Exemplo 3:

$T(N) = 5\,T(N/2) + \theta(N^3)$

Como $f(N) = \Omega(N^3)$, $3 > \log_2 5$ e $f(N) = O(N^3)$, **Caso 3** é aplicado, resultando em

$T(N) = \theta(N^3)$

Análise de Complexidade

- Exemplo 4:

$T(N) = 27\ T(N/3) + \theta(N^3 / \log N)$

Nenhum caso pode ser aplicado. Esta recorrência não pode ser resolvida com o Teorema Mestre.

(felizmente, casos como este são menos comuns…)

Análise de Complexidade

- Outros Exemplos:
  - Pesquisa Binária
  - Ordenação MergeSort

Exemplo Prático

Análise de Complexidade

- **Problema**:

Dados um vetor `A[1..N]` de naturais, com `N` ≈ 1 bilhão e cada natural ≤ 2 bilhões, e um natural `K` ≤ 2 bilhões, determinar se há dois elementos deste vetor cujo produto resulta em `K`. Ex.:

```
N=8, A=[10,16,7,60,5,3,24,4], K=48 ⇒ SIM
N=8, A=[10,16,7,60,5,3,24,4], K=9  ⇒ SIM
N=8, A=[10,16,7,60,5,3,24,4], K=45 ⇒ NÃO
```

Análise de Complexidade

- **Solução #1**:

```
função Prod(A[],N,K: Inteiro): Lógico
   para i ← 1 até N faça
      para j ← 1 até N faça
         se A[i]*A[j]=K então
            retornar V
   retornar F
```

Tempo:
≈ 31 mil anos
(10^6 passos/s)

Tempo:
$O(N^2)$

Análise de Complexidade

- **Solução #2**:

```
função Prod(A[],N,K: Inteiro): Lógico
   para i ← 1 até N faça
      para j ← i até N faça
         se A[i]*A[j]=K então
            retornar V
   retornar F
```

Tempo:
$O(N^2)$

Análise de Complexidade

- **Solução #3**:

```
função Prod(A[],N,K: Inteiro): Lógico
   Ordenar(A,N) //por BubbleSort,
                // InsertionSort, ou SelectionSort
   para i ← 1 até N faça
      pos ← BuscaBinaria(A,N,K/A[i])
      se pos ≥ 1 então
         retornar V
   retornar F
```

Tempo:
$O(N^2)$

Análise de Complexidade

- **Solução #4**:

```
função Prod(A[],N,K: Inteiro): Lógico
   Ordenar(A,N) //por QuickSort
   para i ← 1 até N faça
      pos ← BuscaBinaria(A,N,K/A[i])
      se pos ≥ 1 então
         retornar V
   retornar F
```

Tempo:
≈ 8,3 horas

Tempo:
$O(N^2)$

Caso Médio:
$\theta(N \log N)$

Análise de Complexidade

- **Solução #5**:

```
função Prod(A[],N,K: Inteiro): Lógico
   Ordenar(A,N) //por MergeSort
   para i ← 1 até N faça
      pos ← BuscaBinaria(A,N,K/A[i])
      se pos ≥ 1 então
         retornar V
   retornar F
```

Tempo:
≈ 8,3 horas

Tempo:
$\theta(N \log N)$

Análise de Complexidade

- **Solução #6**:

```
função Prod(A[],N,K: Inteiro): Lógico
    Ordenar(A,N) //por MergeSort
    i,j ← 1,N
    enquanto i ≤ j faça
        se A[i]*A[j]=K então
            retornar V
        senão se A[i]*A[j]>K então
            j ← j-1
        senão
            i ← i+1
    retornar F
```

Tempo:
≈ 17 minutos

Tempo:
$\theta(N \log N)$

ou

$\theta(N)$
se `A[1..N]` já estiver ordenado

Análise de Complexidade

- **Solução #7**:

```
função Prod(A[],N,K: Inteiro): Lógico
    Ordenar(A,N) //por CountingSort
    i,j ← 1,N
    enquanto i ≤ j faça
        se A[i]*A[j]=K então
            retornar V
        senão se A[i]*A[j]>K então
            j ← j-1
        senão
            i ← i+1
    retornar F
```

Tempo:
≈ 17 minutos

Tempo:
$\theta(N)$

Análise de Complexidade

- **Solução #8**:

```
função Prod(A[],N,K: Inteiro): Lógico
    Ordenar(A,N) //por CountingSort

    i ← 1
    enquanto A[i] ≤ √(K) faça
        pos ← BuscaBinaria(A,N,K/A[i])
        se pos ≥ 1 então
            retornar V
        i ← i+1
    retornar F
```

Tempo:
≈ 1 segundo

Tempo:
$\theta(N)$

ou

$\theta(\sqrt{(N)} \log N)$
se `A[1..N]` já estiver ordenado
e seus elementos distribuídos uniformemente

Exercícios

Exercícios

1. Preencha com V ou F (preencha “V” na linha “100” coluna “O(n)”, por exemplo, se 100 = O(n) ou “F” caso contrário)

	$O(n)$	$\Omega(n)$	$\theta(n)$	$O(n^2)$	$\Omega(n^2)$	$\theta(n^2)$
100						
$3n + 3$						
$10n^2 + n$						
$2n^3$						

	$o(n)$	$\omega(n)$	$o(n^2)$	$\omega(n^2)$
100				
$3n + 3$				
$10n^2 + n$				
$2n^3$				

Exercícios

2. Prove ou refute (**necessário usar a definição de cada notação**):
   a. $n^{2,7} = o(n^3)$
   b. $2^{n+1} = \theta(2^n)$
   c. $2^{2n} = O(2^n)$
   d. $\log n = \theta(\lg n)$
   e. $\lg (n!) = O(n \lg n)$
   f. $\lg^k n = O(n)$, para todo $k \geq 1$

   Sejam f(n) e g(n) funções assintoticamente positivas.

   h. $f(n) = O(g(n))$ implica $g(n) = O(f(n))$
   i. $f(n) = O(g(n))$ implica $\lg(f(n)) = O(\lg(g(n)))$, onde $\lg(g(n)) \geq 1$ e $f(n) \geq 1$ para todo n suficientemente grande
   j. $f(n) = O(g(n))$ implica $2^{f(n)} = O(2^{g(n)})$
   k. $f(n) = O((f(n))^2)$
   l. $f(n) = O(g(n))$ implica $g(n) = \Omega(f(n))$
   m. $f(n) = \theta(f(n/2))$
   n. $f(n) + o(f(n)) = \theta(f(n))$

Exercícios

3. Determine a complexidade de tempo do algoritmo InsertionSort:

```
procedimento Ordenar(B[], N: Inteiro)
    var i, j, t: Inteiro
    para i ← 2 até N faça
        t ← B[i]
        j ← i - 1
        enquanto j > 0 e B[j] > t faça
            B[j+1] ← B[j]
            j ← j - 1
        B[j+1] ← t
```

Exercícios

4. Determine a complexidade de tempo dos algoritmos abaixo:

a)
```
para i ← 1 até 10 faça
    escrever(i)
```

b)
```
para i ← 1 até 10 faça
    escrever(A[1..N])
```

c)
```
para i ← 1 até N faça
    escrever(A[1..N])
```

d)
```
para i ← 1 até N faça
    escrever(A[1..i])
```

e)
```
para i ← 1 até N*N faça
    escrever(i*i)
```

f)
```
i ← 1
enquanto i < N/4 faça
    i ← i*2
```

h)
```
i ← 1
enquanto i < N*N faça
    i ← i*2
```

i)
```
i ← N
enquanto i > 1 faça
    i ← i div 2
```

j)
```
i ← 2
enquanto i < N faça
    i ← i * i
```

Exercícios

5. Determine a complexidades de tempo do algoritmo abaixo:

```
procedimento Imprime(ref B[], N: Inteiro)
    var i, j: Inteiro
    i ← 1
    enquanto i < N faça
        escrever( B[i] )
        i ← i * 2
    se B[1] > 0 então
        para i ← 1 até N/2 faça
            j ← i+1
            enquanto ( j < N ) E (B[j] > B[i] ) faça
                escrever ( B[j] )
                j ← j + 1
```

Exercícios

6. Determine as complexidades de tempo dos algoritmos abaixo:

```
procedimento Imprime(ref B[], N: Inteiro)
    var i: Inteiro ← 1
    enquanto i < N faça
        escrever( B[i] )
        i ← i * 4
        Calcula(B, N)

procedimento Calcula(ref B[], N: Inteiro)
    var i, j, z, x: Inteiro; i, x ← 1, 0
    enquanto B[i] < 0 e i < N-1 faça
        i ← i+1
    para j ← i até N faça
        para z ← N até i passo -1 faça
            x ← x + B[j]*z
    escrever ( x )
```

Exercícios

7. Determine as complexidades de tempo do algoritmos abaixo:

```
função ExisteValor(ref M[], L1,L2,C1,C2,x: Inteiro): Lógico
    se L1 ≤ L2 e C1 ≤ C2 então
        var m1, m2: Inteiro
        m1, m2 ← (L1+L2) div 2, (C1+C2) div 2
        se M[m1, m2] = x então
            retornar V
        senão se M[m1, m2] < x então
            retornar ExisteValor(M, m1+1, L2, m2+1, C2, x)
        senão
            retornar ExisteValor(M, L1, m1-1, C1, m2-1, x)
    senão
        retornar F
```

Exercícios

8. Marque (**V**)erdadeiro ou (**F**)also e justifique.
   a. ( ) Um algoritmo A resolve o problema P em tempo $O(n^2)$ e um algoritmo B resolve P em tempo $\Omega(n^2)$. Portanto, P é resolvível em tempo $\Theta(n^2)$.
   b. ( ) Um algoritmo que executa em tempo $\Theta(n \lg n)$ é sempre preferível a outro que resolve o mesmo problema em tempo $O(n^4)$ para n suficientemente grande.
   c. ( ) Prova-se que, sob um conjunto restrito das entradas, um algoritmo A executa exatamente n^2+n-50 passos. Logo, a complexidade de tempo de A é $O(n^2)$.
   d. ( ) Prova-se que, sob um conjunto restrito das entradas, um algoritmo A executa exatamente n^2+n-50 passos. Logo, a complexidade de tempo de pior caso de A é $\Omega(n^2)$.
   e. ( ) Devido aos valores de N, entrada de certo algoritmo A, serem muito elevados em certa aplicação, nenhum algoritmo de tempo $\omega(N\sqrt{N})$ é aceitável. Um algoritmo que resolve o mesmo problema em tempo de pior caso $\Theta(n^{1.4})$ estaria portanto dentro do desejável.

Exercícios

9. O famoso algoritmo de Euclides para calcular o MDC entre os números x > y é dado abaixo.
   a. Mostre que sua complexidade é O(lg x). (Dica: se a,b,c,d são uma subsequência de restos produzidos pelo algoritmo, mostre que 2(c+d) < a+b e elabore com tal desigualdade um limite superior no número de iterações do algoritmo.)
   b. Mostre que sua complexidade de pior caso é θ(lg x). (Dica: mostre que se x = F(n) e y = F(n-1), onde F(n) representa o n-ésimo número de Fibonacci, então a sequência de restos é a série de Fibonacci F(1),F(2),...,F(n-1),F(n). O número de Fibonnaci F(n) é definido como F(n)=n se n=1 ou n=2 e F(n) = F(n-1)+F(n-2) se n>2.)

```
função MDC(x, y: Inteiro): Inteiro
    enquanto y ≠ 0 faça
        x,  y  ← y,  x mod y
    retornar x
```