Análise de Algoritmos: Introdução

INF 332 - Projeto e Análise de Algoritmos

Departamento de Informática
Universidade Federal de Viçosa

INF 332 - 2022/2

Outline

Analisar/avaliar um algoritmo consiste em:

- Verificar se o algoritmo está **correto**:
  - O algoritmo fornece uma solução válida para o problema?
- Verificar sua **eficiência**:
  - Quanto **tempo** gasta?
  - Quanto de **memória** usa?

- Eficiência de **tempo** (complexidade de tempo)
- Eficiência de **espaço** (complexidade de espaço)

Iremos focar a **complexidade de tempo.**

Análise de Algoritmos

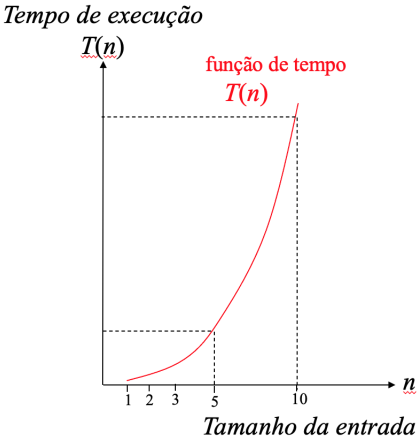
- Para resolver um problema podem ser projetados **diferentes algoritmos** .
- O fato de um algoritmo resolver (teoricamente) o problema não significa que seja aceitável na prática.
- Através da análise de algoritmos, podemos determinar o algoritmo mais eficiente (o **melhor algoritmo**).

Complexidade de Tempo

- Analisar um algoritmo com relação ao tempo consiste em calcular o "**tempo de execução**" sem implementá-lo em uma plataforma específica;
- O "tempo de execução" de um algoritmo depende do **tamanho da entrada** do algoritmo, ou seja, do **tamanho do problema** resolvido pelo algoritmo.
- "Quanto **maior o tamanho da entrada**, **maior** será a **tempo de execução** do algoritmo"

Medindo o Tamanho da Entrada

- $\Rightarrow$ Vamos determinar o **tempo do algoritmo** em função do **tamanho da entrada** (um parâmetro *n*).



Medindo o Tamanho da Entrada

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	
Multiplicação de duas matrizes	
Problemas típico de grafos	
Primalidade de um inteiro n	
Calcular a^n	

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	Número de itens na lista, i.e., n
Multiplicação de duas matrizes	
Problemas típico de grafos	
Primalidade de um inteiro n	
Calcular a^n	

Medindo o Tamanho da Entrada

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	Número de itens na lista, i.e., n
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elementos
Problemas típico de grafos	
Primalidade de um inteiro n	
Calcular a^n	

Medindo o Tamanho da Entrada

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	Número de itens na lista, i.e., n
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elementos
Problemas típico de grafos	Número de vértices e/ou arestas
Primalidade de um inteiro n	
Calcular a^n	

Medindo o Tamanho da Entrada

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	Número de itens na lista, i.e., n
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elementos
Problemas típico de grafos	Número de vértices e/ou arestas
Primalidade de um inteiro n	Valor de n
Calcular a^n	

Medindo o Tamanho da Entrada

Exemplos de tamanhos de entrada

Problema	**Medida do Tamanho de Entrada**
Procurar uma chave em uma lista com n itens	Número de itens na lista, i.e., n
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elementos
Problemas típico de grafos	Número de vértices e/ou arestas
Primalidade de um inteiro n	Valor de n
Calcular a^n	Valor de n

Unidades para Medir Tempo

- Utilizar **medida padrão de tempo** como **segundos** ou **milissegundos** (por exemplo, para medir o tempo de execução de um programa que implementa um algoritmo).

- Desvantagens:

Unidades para Medir Tempo

- Utilizar **medida padrão de tempo** como **segundos** ou **milissegundos** (por exemplo, para medir o tempo de execução de um programa que implementa um algoritmo).

- Desvantagens:
  - Dependente da *velocidade do computador* (hardware)
  - Depende da *linguagem de programação* e do *compilador* usado para geração do código de máquina

Unidades para Medir Tempo

- **Alternativa**: contar o número de vezes que cada operação do algoritmo é executada.

  Esta unidade não depende de fatores externos.

- **Problema**: Pode ser difícil de calcular e muitas vezes é desnecessário considerar todas as operações.

Unidades para Medir Tempo

- $\Rightarrow$ Identificar a operação básica (a operação que mais contribui para o tempo de execução do algoritmo).
- **Contar** quantas vezes essa operação básica é executada.
- Geralmente essa operação está no *laço mais interno* do algoritmo.

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com *n* itens	Número de itens na lista, i.e., *n*	
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	
Problema de grafos	Número de vértices e/ou arestas	
Primalidade de um inteiro *n*	Valor de *n*	
Calcular a^n	Valor de *n*	

Unidades para Medir Tempo

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com n itens	Número de itens na lista, i.e., n	Comparação de chaves
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	
Problema de grafos	Número de vértices e/ou arestas	
Primalidade de um inteiro n	Valor de n	
Calcular a^n	Valor de n	

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com *n* itens	Número de itens na lista, i.e., *n*	Comparação de chaves
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	Multiplicação de dois números
Problema de grafos	Número de vértices e/ou arestas	
Primalidade de um inteiro *n*	Valor de *n*	
Calcular a^n	Valor de *n*	

Unidades para Medir Tempo

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com n itens	Número de itens na lista, i.e., n	Comparação de chaves
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	Multiplicação de dois números
Problema de grafos	Número de vértices e/ou arestas	Visitar um vértice ou uma aresta
Primalidade de um inteiro n	Valor de n	
Calcular a^n	Valor de n	

Unidades para Medir Tempo

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com *n* itens	Número de itens na lista, i.e., *n*	Comparação de chaves
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	Multiplicação de dois números
Problema de grafos	Número de vértices e/ou arestas	Visitar um vértice ou uma aresta
Primalidade de um inteiro *n*	Valor de *n*	Divisão
Calcular a^n	Valor de *n*	

Unidades para Medir Tempo

Exemplos de **tamanhos de entrada** e **operações básicas**

Problema	**Tamanho de Entrada**	**Operação Básica**
Procurar uma chave x em uma lista com n itens	Número de itens na lista, i.e., n	Comparação de chaves
Multiplicação de duas matrizes	Dimensões das matrizes ou número total de elem.	Multiplicação de dois números
Problema de grafos	Número de vértices e/ou arestas	Visitar um vértice ou uma aresta
Primalidade de um inteiro n	Valor de n	Divisão
Calcular a^n	Valor de n	Multiplicação

Análise Teórica da Eficiência de Tempo

- A **eficiência de tempo** de um algoritmo é analisada calculando o número de vezes que a operação básica é executada.
  Este número é determinado em **função** do tamanho da entrada n.

Notação:

- n: tamanho da entrada
- $T(n)$: número de vezes que a operação básica é executada.

Análise Teórica da Eficiência de Tempo

$$Tempo(n) \approx t_{op}.T(n)$$

- $Tempo(n)$: tempo de execução aproximado do algoritmo (em minutos, segundos, milisegundos).
- t_{op}: tempo de uma execução da operação básica (em minutos, segundos, milisegundos).
- $T(n)$: número de vezes que a operação básica é executada.

Análise Teórica da Eficiência de Tempo

$$Tempo(n) \approx t_{op}.T(n)$$

- Esta fórmula fornece uma aproximação do tempo de execução do algoritmo (tempo de relógio)
- Permite responder perguntas úteis como a seguinte...

Análise Teórica da Eficiência de Tempo

Se $Tempo(n) \approx t_{op}.T(n)$,

Quão mais rápido será um algoritmo se ele é executado em uma máquina 20 vezes mais rápida?

Se $Tempo(n) \approx t_{op}.T(n)$,

Quão mais rápido será um algoritmo se ele é executado em uma máquina 20 vezes mais rápida?

- 20 vezes mais rápido

Análise Teórica da Eficiência de Tempo

$$Tempo(n) \approx t_{op}.T(n)$$

Suponha que $T(n) = \frac{1}{2}n(n-1)$,
Qual será o **aumento** no tempo de processamento se dobrarmos o tamanho da entrada?

$$Tempo(n) \approx t_{op}.T(n)$$

Suponha que $T(n) = \frac{1}{2}n(n-1)$,
Qual será o **aumento** no tempo de processamento se dobrarmos o tamanho da entrada?

$$T(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$Tempo(n) \approx t_{op}.T(n)$$

Suponha que $T(n) = \frac{1}{2}n(n-1)$,
Qual será o **aumento** no tempo de processamento se dobrarmos o tamanho da entrada?

$$T(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$\frac{Tempo(2n)}{Tempo(n)} \approx \frac{t_{op}.T(2n)}{t_{op}.T(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

Análise Teórica da Eficiência de Tempo

$$Tempo(n) \approx t_{op}.T(n)$$

Suponha que $T(n) = \frac{1}{2}n(n-1)$,
Qual será o **aumento** no tempo de processamento se dobrarmos o tamanho da entrada?

$$T(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2$$

$$\frac{Tempo(2n)}{Tempo(n)} \approx \frac{t_{op}.T(2n)}{t_{op}.T(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

$\Rightarrow$ O tempo aumentará em *aproximadamente* 4 vezes

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador *A* é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador *B*, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador *A*: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador *B*: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador *B*, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Para um determinado algoritmo, suponha que $T(n) = n^2$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
Suponha que temos um novo computador B, 100 vezes mais rápido que o computador A.
No mesmo tempo t, qual será o **tamanho da entrada** do problema que pode ser resolvido pelo novo computador?

Computador A: $Tempo(10^4) = t_{op}.(10^4)^2 = t_{op}.10^8$ seg
Computador B: $Tempo(n) = t_{op}.n^2/100$ seg,
$t_{op}.n^2/100 = t_{op}.10^8$,
$n^2 = 10^2 10^8$,
$n^2 = 10^{10}$
$n = 10^5 = 100.000$
$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema $\times 10$ maior.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Análise Teórica da Eficiência de Tempo

Suponha que o algoritmo tem $T(n) = 2^n$.
Num computador A é possível resolver um problema com tamanho de entrada $n = 10^4 = 10.000$ em t segundos.
No computador B, qual o tamanho do problema que pode ser resolvido no mesmo tempo?

Computador A: $Tempo(10^4) = t_{op}.2^{10.000}$ seg
Computador B: $Tempo(n) = t_{op}.2^n/100$ seg,
$2^n/100 = 2^{10.000}$,
$2^n = 100 \times 2^{10.000}$,
$n = log_2 10^2 + log_2 2^{10.000}$
$n = 2log_2(2 \times 5) + 10.000 = 2 + 2log_2 5 + 10.000$
$n = 2 + 2(2,32) + 10.000 = 10.000 + 6,64$

$\Rightarrow$ No computador B, no mesmo tempo, pode ser resolvido um problema quase do mesmo tamanho. O incremento no tamanho da entrada será de apenas 6 unidades.

Ordem de Grandeza (Ordem de Crescimento)

- No cálculo de $T(n)$ (número de vezes que a operacao básica é executada), podem ser ignoradas os **termos de menor grau** e as **constantes** aditivas e multiplicativas (**coeficientes**).
- A eficiência de um algoritmo depende do **termo de maior grau** (ou seja, o termo de maior grau determina a **ordem de crescimento** da função $T(n)$).

Ordem de Grandeza (Ordem de Crescimento)

- No cálculo de $T(n)$ (número de vezes que a operacao básica é executada), podem ser ignoradas os **termos de menor grau** e as **constantes** aditivas e multiplicativas (**coeficientes**).
- A eficiência de um algoritmo depende do **termo de maior grau** (ou seja, o termo de maior grau determina a **ordem de crescimento** da função $T(n)$).
- Exemplo: $T(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx n^2$ (estimativa do número de operacões executadas).

Ordem de Grandeza (Ordem de Crescimento)

- No cálculo de $T(n)$ (número de vezes que a operacao básica é executada), podem ser ignoradas os **termos de menor grau** e as **constantes** aditivas e multiplicativas (**coeficientes**).
- A eficiência de um algoritmo depende do **termo de maior grau** (ou seja, o termo de maior grau determina a **ordem de crescimento** da função $T(n)$).
- Exemplo: $T(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx n^2$ (estimativa do número de operacões executadas).
- A eficiência do algoritmo deve ser analisada para entradas suficientemente grandes ($n \longrightarrow \infty$).

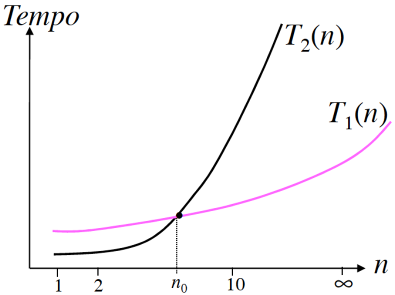
Eficiência Assintótica

Algoritmo A1: $T_1(n) = 3n^2 + 4n + 50$

Algoritmo A2: $T_2(n) = n^3$

	$n = 1$	$n = 2$	$n = 10$
T_1(n)	57	70	390
T_2(n)	1	8	1000

Para n suficientemente grande o algoritmo A1 é *assintóticamente* mais eficiente do que o algoritmo A2.



$T_1(n) \leq T_2(n), \ \forall n \geq n_0$

Aproximação de **valores numéricos** para **funções importantes** na análise de algoritmos:

n	$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n	$n!$
10	3	10	3.3×10^1	10^2	10^3	1.0×10^3	3.6×10^6
10^2	7	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	1.0×10^4	10^6	10^9	1.1×10^{301}	–
10^4	13	10^4	1.3×10^5	10^8	10^{12}	–	–
10^5	17	10^5	1.7×10^6	10^{10}	10^{15}	–	–
10^6	20	10^6	2.0×10^7	10^{12}	10^{18}	–	–

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	
20	
40	
50	
60	

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.000001024 segundos
20	
40	
50	
60	

Ordem de Grandeza

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.000001024 segundos
20	0.0010486 segundos
40	
50	
60	

Ordem de Grandeza

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.000001024 segundos
20	0.0010486 segundos
40	18.3 minutos
50	
60	

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.000001024 segundos
20	0.0010486 segundos
40	18.3 minutos
50	13 dias
60	

Ordem de Grandeza

- Assuma que a operação básica gasta 10^{-9} segundos para ser executada. Considere um programa que implementa um algoritmo em que a operação básica é executada 2^n vezes ($T(n) = 2^n$). Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.000001024 segundos
20	0.0010486 segundos
40	18.3 minutos
50	13 dias
60	36.5 anos

- A operação básica gasta 10^{-9} segundos para ser executada. A operação básica é executada $n!$ vezes ($T(n) = n!$).
  Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	
20	
21	

Ordem de Grandeza

- A operação básica gasta 10^{-9} segundos para ser executada. A operação básica é executada $n!$ vezes ($T(n) = n!$).
Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.0036288 segundos
20	
21	

- A operação básica gasta 10^{-9} segundos para ser executada. A operação básica é executada $n!$ vezes ($T(n) = n!$).
  Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.0036288 segundos
20	77 anos
21	

Ordem de Grandeza

- A operação básica gasta 10^{-9} segundos para ser executada. A operação básica é executada $n!$ vezes ($T(n) = n!$).
  Então teremos:

Tamanho da entrada: n	Tempo de Execução ($Tempo(n) = t_{op}.T(n)$)
10	0.0036288 segundos
20	77 anos
21	1620 anos

Pior Caso, Melhor Caso e Caso Médio

- A complexidade de um algoritmo não somente depende do **tamanho da entrada**, mas também da instância do problema a ser resolvido (uma **entrada particular**).

Pior Caso, Melhor Caso e Caso Médio

- A complexidade de um algoritmo não somente depende do **tamanho da entrada**, mas também da instância do problema a ser resolvido (uma **entrada particular**).

Exemplo: **Problema da busca.** Para o Algoritmo de Busca Sequencial, podemos ter $n + 1$ instâncias de tamanho n para procurar uma chave K.

Para $n = 6$ e chave $K = 5$, temos 7 entradas:

E1 = {**5**, 9, 1, 20, 4, 7} $\Rightarrow$ Melhor Entrada
E2 = {2, **5**, 1, 8, 7, 10}
E3 = {4, 2, **5**, 7, 1, 11}
E4 = {4, 2, 11, **5**, 1, 9}
E5 = {6, 9, 1, 20, **5**, 7}
E6 = {3, 7, 1, 10, 4, **5**} $\Rightarrow$ Pior Entrada
E7 = {2, 8, 4, 20, 3, 7} $\Rightarrow$ Pior Entrada

Pior Caso, Melhor Caso e Caso Médio

- **Pior caso:** $T_{pior}(n)$ - é o maior tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o maior número de vezes?

- **Melhor caso:** $T_{melhor}(n)$ - é o menor tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o menor número de vezes?

- **Caso médio** (caso esperado): $T_{medio}(n)$ - é a média dos tempos de execução de todas as entradas possíveis de tamanho n, assumindo uma distribuição probabilística.

Pior Caso, Melhor Caso e Caso Médio

- **Pior caso:** $T_{pior}(n)$ - é o maior tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o maior número de vezes?

- **Melhor caso:** $T_{melhor}(n)$ - é o menor tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o menor número de vezes?

- **Caso médio** (caso esperado): $T_{medio}(n)$ - é a média dos tempos de execução de todas as entradas possíveis de tamanho n, assumindo uma distribuição probabilística.

Pior Caso, Melhor Caso e Caso Médio

- **Pior caso:** $T_{pior}(n)$ - é o maior tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o maior número de vezes?

- **Melhor caso:** $T_{melhor}(n)$ - é o menor tempo de execução sobre todas as possíveis entradas de tamanho n.

  Para qual entrada a operacão básica é executada o menor número de vezes?

- **Caso médio** (caso esperado): $T_{medio}(n)$ - é a média dos tempos de execução de todas as entradas possíveis de tamanho n, assumindo uma distribuição probabilística.

Pior Caso, Melhor Caso e Caso Médio

Busca sequencial

Busca ($A[1..n], K$)

```
//Buscar a chave K no array A
for i ← 1 to n do
    if A[i] = K
        return i    //successful, A[i] matches K
return −1           //unsuccessful, no matching
```

Operação básica do algoritmo: *A[i] = K* (comparações)

- Tempo de **Pior caso**:
- Tempo de **Melhor caso**:
- Tempo de **Caso médio**:

Pior Caso, Melhor Caso e Caso Médio

Busca sequencial

Busca $(A[1..n], K)$

```
//Buscar a chave K no array A
for i ← 1 to n do
    if A[i] = K
        return i   //successful, A[i] matches K
return −1          //unsuccessful, no matching
```

Operação básica do algoritmo: *A[i] = K* (comparações)

- Tempo de **Pior caso**: $T_{pior}(n) = n$ (a operação básica será executada n vezes).
- Tempo de **Melhor caso**:
- Tempo de **Caso médio**:

Busca sequencial

```
Busca (A[1..n], K)
    //Buscar a chave K no array A
    for i ← 1 to n do
        if A[i] = K
            return i    //successful, A[i] matches K
    return −1           //unsuccessful, no matching
```

Operação básica do algoritmo: $A[i] = K$ (comparações)

- Tempo de **Pior caso**: $T_{pior}(n) = n$ (a operação básica será executada n vezes).
- Tempo de **Melhor caso**: $T_{melhor}(n) = 1$ (a operação básica será executada 1 vez).
- Tempo de **Caso médio**:

Pior Caso, Melhor Caso e Caso Médio

Busca sequencial

Busca $(A[1..n], K)$

```
//Buscar a chave K no array A
for i ← 1 to n do
    if A[i] = K
        return i    //successful, A[i] matches K
return −1           //unsuccessful, no matching
```

Operação básica do algoritmo: *A[i] = K* (comparações)

- Tempo de **Pior caso**: $T_{pior}(n) = n$ (a operação básica será executada n vezes).
- Tempo de **Melhor caso**: $T_{melhor}(n) = 1$ (a operação básica será executada 1 vez).
- Tempo de **Caso médio**: $T_{medio}(n) = \ldots$

Pior Caso, Melhor Caso e Caso Médio

Tempo de **caso médio** da *Busca Sequencial*:

- O elemento procurado pode estar na posição i ($i = 1...n$) ou não está no array. Ou seja, há $n+1$ possíveis entradas.
- *Probabilidade* de cada entrada ocorrer: $p_i = \dfrac{1}{n+1}, \forall i = 1...n+1$
- Número de comparações realizadas por cada entrada: $T_i(n) = i, \forall i = 1...n, \quad T_{n+1}(n) = n$
- $T_{medio}(n) = \sum_{i=1}^{n+1} p_i \times T_i(n)$

$$T_{medio}(n) = \sum_{i=1}^{n} \frac{1}{n+1} \times i + \frac{1}{n+1} \times n$$

$$T_{medio}(n) = \frac{1}{n+1} \sum_{i=1}^{n} i + \frac{n}{n+1}$$

$$T_{medio}(n) = \frac{1}{n+1} \frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

Pior Caso, Melhor Caso e Caso Médio

Tempo de **caso médio** da *Busca Sequencial*:

- O elemento procurado pode estar na posição i ($i = 1...n$) ou não está no array. Ou seja, há $n+1$ possíveis entradas.
- *Probabilidade* de cada entrada ocorrer: $p_i = \dfrac{1}{n+1}, \forall i = 1...n+1$
- Número de comparações realizadas por cada entrada:
  $T_i(n) = i, \forall i = 1...n, \quad T_{n+1}(n) = n$
- $T_{medio}(n) = \sum_{i=1}^{n+1} p_i \times T_i(n)$

$$T_{medio}(n) = \sum_{i=1}^{n} \frac{1}{n+1} \times i + \frac{1}{n+1} \times n$$

$$T_{medio}(n) = \frac{1}{n+1} \sum_{i=1}^{n} i + \frac{n}{n+1}$$

$$T_{medio}(n) = \frac{1}{n+1} \frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

Pior Caso, Melhor Caso e Caso Médio

Tempo de **caso médio** da *Busca Sequencial*:

- O elemento procurado pode estar na posição i ($i = 1...n$) ou não está no array. Ou seja, há $n+1$ possíveis entradas.
- *Probabilidade* de cada entrada ocorrer: $p_i = \dfrac{1}{n+1}, \forall i = 1...n+1$
- Número de comparações realizadas por cada entrada:
  $T_i(n) = i, \forall i = 1...n, \quad T_{n+1}(n) = n$
- $T_{medio}(n) = \sum_{i=1}^{n+1} p_i \times T_i(n)$

  $T_{medio}(n) = \sum_{i=1}^{n} \dfrac{1}{n+1} \times i + \dfrac{1}{n+1} \times n$

  $T_{medio}(n) = \dfrac{1}{n+1} \sum_{i=1}^{n} i + \dfrac{n}{n+1}$

  $T_{medio}(n) = \dfrac{1}{n+1} \dfrac{n(n+1)}{2} + \dfrac{n}{n+1} = \dfrac{n}{2} + \dfrac{n}{n+1}$

Pior Caso, Melhor Caso e Caso Médio

Tempo de **caso médio** da *Busca Sequencial*:

- O elemento procurado pode estar na posição i ($i = 1...n$) ou não está no array. Ou seja, há $n + 1$ possíveis entradas.
- *Probabilidade* de cada entrada ocorrer: $p_i = \dfrac{1}{n+1}, \forall i = 1...n+1$
- Número de comparações realizadas por cada entrada: $T_i(n) = i, \forall i = 1...n, \quad T_{n+1}(n) = n$
- $T_{medio}(n) = \sum_{i=1}^{n+1} p_i \times T_i(n)$

$$T_{medio}(n) = \sum_{i=1}^{n} \frac{1}{n+1} \times i + \frac{1}{n+1} \times n$$

$$T_{medio}(n) = \frac{1}{n+1} \sum_{i=1}^{n} i + \frac{n}{n+1}$$

$$T_{medio}(n) = \frac{1}{n+1} \frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

Pior Caso, Melhor Caso e Caso Médio

Tempo de **caso médio** da *Busca Sequencial*:

- O elemento procurado pode estar na posição i ($i = 1 ... n$) ou não está no array. Ou seja, há $n + 1$ possíveis entradas.
- *Probabilidade* de cada entrada ocorrer: $p_i = \frac{1}{n+1}, \forall i = 1 ... n+1$
- Número de comparações realizadas por cada entrada: $T_i(n) = i, \forall i = 1 ... n, \quad T_{n+1}(n) = n$
- $T_{medio}(n) = \sum_{i=1}^{n+1} p_i \times T_i(n)$

  $T_{medio}(n) = \sum_{i=1}^{n} \frac{1}{n+1} \times i + \frac{1}{n+1} \times n$

  $T_{medio}(n) = \frac{1}{n+1} \sum_{i=1}^{n} i + \frac{n}{n+1}$

  $T_{medio}(n) = \frac{1}{n+1} \frac{n(n+1)}{2} + \frac{n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$

Pior Caso, Melhor Caso e Caso Médio

Outra maneira de calcular o tempo de **caso médio** da *Busca Sequencial* [1]:

- Assuma que:
  - p: é a probabilidade de uma busca bem sucedida.
  - $\frac{p}{n}$: é probabilidade de encontrar o elemento K na i-ésima posição do arranjo, para todo $i = 1, ..., n$.

[1] A. Levitin, Introduction to the Design and Analysis of Algorithms

Pior Caso, Melhor Caso e Caso Médio

Outra maneira de calcular o tempo de **caso médio** da *Busca Sequencial* [1]:

- Assuma que:
  - p: é a probabilidade de uma busca bem sucedida.
  - $\frac{p}{n}$: é probabilidade de encontrar o elemento K na i-ésima posição do arranjo, para todo $i = 1, ..., n$.
- Não encontrar o elemento K ocorre com probabilidade $1 - p$ após n comparações

[1] A. Levitin, Introduction to the Design and Analysis of Algorithms

Pior Caso, Melhor Caso e Caso Médio

O número médio de comparações feitos pelo algoritmo de busca sequencial é:

$$T_{medio}(n) = \sum_{i=1}^{n+1} T_i(n) \times p_i$$

$$T_{medio}(n) = (1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}) + n(1 - p)$$

$$T_{medio}(n) = \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1 - p)$$

$$T_{medio}(n) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1 - p)$$

$$T_{medio}(n) = \frac{p(n+1)}{2} + n(1 - p)$$

Pior Caso, Melhor Caso e Caso Médio

O número médio de comparações feitos pelo algoritmo de busca sequencial é:

$$T_{medio}(n) = \sum_{i=1}^{n+1} T_i(n) \times p_i$$

$$T_{medio}(n) = (1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p)$$

$$T_{medio}(n) = \frac{p(n+1)}{2} + n(1-p)$$

Pior Caso, Melhor Caso e Caso Médio

O número médio de comparações feitos pelo algoritmo de busca sequencial é:

$$T_{medio}(n) = \sum_{i=1}^{n+1} T_i(n) \times p_i$$

$$T_{medio}(n) = (1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p)$$

$$T_{medio}(n) = \frac{p(n+1)}{2} + n(1-p)$$

Pior Caso, Melhor Caso e Caso Médio

O número médio de comparações feitos pelo algoritmo de busca sequencial é:

$$T_{medio}(n) = \sum_{i=1}^{n+1} T_i(n) \times p_i$$

$$T_{medio}(n) = (1\frac{p}{n} + 2\frac{p}{n} + \cdots + i\frac{p}{n} + \cdots + n\frac{p}{n}) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n}(1 + 2 + \cdots + i + \cdots + n) + n(1-p)$$

$$T_{medio}(n) = \frac{p}{n} \times \frac{n(n+1)}{2} + n(1-p)$$

$$T_{medio}(n) = \frac{p(n+1)}{2} + n(1-p)$$

Pior Caso, Melhor Caso e Caso Médio

A busca sequencial faz em média $\frac{p(n+1)}{2} + n(1-p)$ comparações.

- Se $p = 1$, (i.e., a busca é bem sucedida), a média de comparações é: $T_{medio}(n) = (n+1)/2$
- Se $p = 0$, (i.e., busca mal sucedida), o número médio de comparações é: $T_{medio}(n) = n$.

Plano para Analisar Algoritmos Não-Recursivos

- Determinar o **tamanho da entrada** n.
- Identificar a **operação básica** do algoritmo.
- Determinar o **número de vezes que a operação básica é executada**
- Investigar se existe os **casos**: pior, melhor e médio.
- Formular o **somatório** que determina o número de vezes que a operação básica é executada.
- **Simplificar o somatório** (usando fórmulas e regras) para obter uma fórmula fechada.
- Finalmente, indicar a classe de eficiência do algoritmo, através das notações assintóticas.

Exemplo 1: Elementos únicos

Algoritmo que verifica se os elementos de um arranjo são únicos (ou diferentes):

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow i+1$ **to** $n-1$ **do**
    if $A[i] = A[j]$ **return false**
return true

- **Tamanho da entrada:** n (número de elementos no arranjo).
- **Operação básica:** comparação de elementos
- **Número de comparações** não depende apenas de n, depende também da entrada. $\Rightarrow$ tempos de **melhor caso** e **pior caso**.

Exemplo 1: Elementos únicos

Algoritmo que verifica se os elementos de um arranjo são únicos (ou diferentes):

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

- **Tamanho da entrada:** n (número de elementos no arranjo).
- **Operação básica:** comparação de elementos
- **Número de comparações** não depende apenas de n, depende também da entrada. $\Rightarrow$ tempos de **melhor caso** e **pior caso**.

Exemplo 1: Elementos únicos

Algoritmo que verifica se os elementos de uma lista são únicos (diferentes)

ALGORITHM *UniqueElements*$(A[0..n-1])$

//Determines whether all the elements in a given array are distinct
//Input: An array $A[0..n-1]$
//Output: Returns "true" if all the elements in A are distinct
//   and "false" otherwise
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] = A[j]$ **return false**
return true

- Análise de **pior caso** (todos os elementos são diferentes).

$$T_{pior}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \, .$$

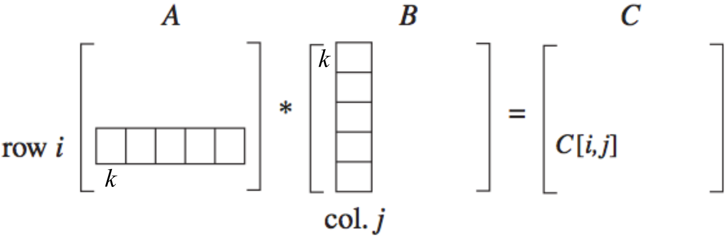
Exemplo 1: Elementos únicos

Simplificando o somatório:

$$
\begin{aligned}
T_{pior}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} \{(n-1) - (i+1) + 1\} = \sum_{i=0}^{n-2} \{n - i - 1\} \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{2(n-1)^2 - (n-2)(n-1)}{2} \\
&= \frac{(n-1)(2n-2-n+2)}{2} = \frac{n(n-1)}{2}.
\end{aligned}
$$

Exemplo 2: Multiplicação de matrizes

Considerar matrizes de ordem $n \times n$:



Exemplo 2: Multiplicação de matrizes

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

```
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two n × n matrices A and B
//Output: Matrix C = AB
for i ← 0 to n − 1 do
    for j ← 0 to n − 1 do
        C[i, j] ← 0.0
        for k ← 0 to n − 1 do
            C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j]
return C
```

- **Tamanho da entrada:** ordem n da matriz.
- **Operação básica:** multiplicação ($A[i,k] * B[k,j]$).
- Número de multiplicações: depende apenas de n.

Exemplo 2: Multiplicação de matrizes

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$
//Multiplies two square matrices of order n by the definition-based algorithm
//Input: Two $n \times n$ matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
  for $j \leftarrow 0$ **to** $n-1$ **do**
    $C[i, j] \leftarrow 0.0$
    for $k \leftarrow 0$ **to** $n-1$ **do**
      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

- **Tamanho da entrada:** ordem n da matriz.
- **Operação básica:** multiplicação $(A[i, k] * B[k, j])$.
- Número de multiplicações: depende apenas de n.

Exemplo 2: Multiplicação de matrizes

O algoritmo executa $\sum_{k=0}^{n-1} 1$ multiplicações para cada par (i, j). Portanto, o número total de multiplicações é dado por,

$$
\begin{aligned}
Tn) &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 \\
&= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = n \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 \\
&= n^2 \sum_{i=0}^{n-1} 1 \\
&= n^3 .
\end{aligned}
$$