INF 213 – Estrutura de dados

Prof. Marcus V. A. Andrade

Complexidade de algoritmos

Prof. Marcus V. A. Andrade

Adaptado do material do Prof. José Elias Arroyo

Análise de Algoritmos

- Avaliar um algoritmo consiste em:
  - Verificar se o algoritmo está correto:

    O algoritmo fornece uma solução válida para o problema?

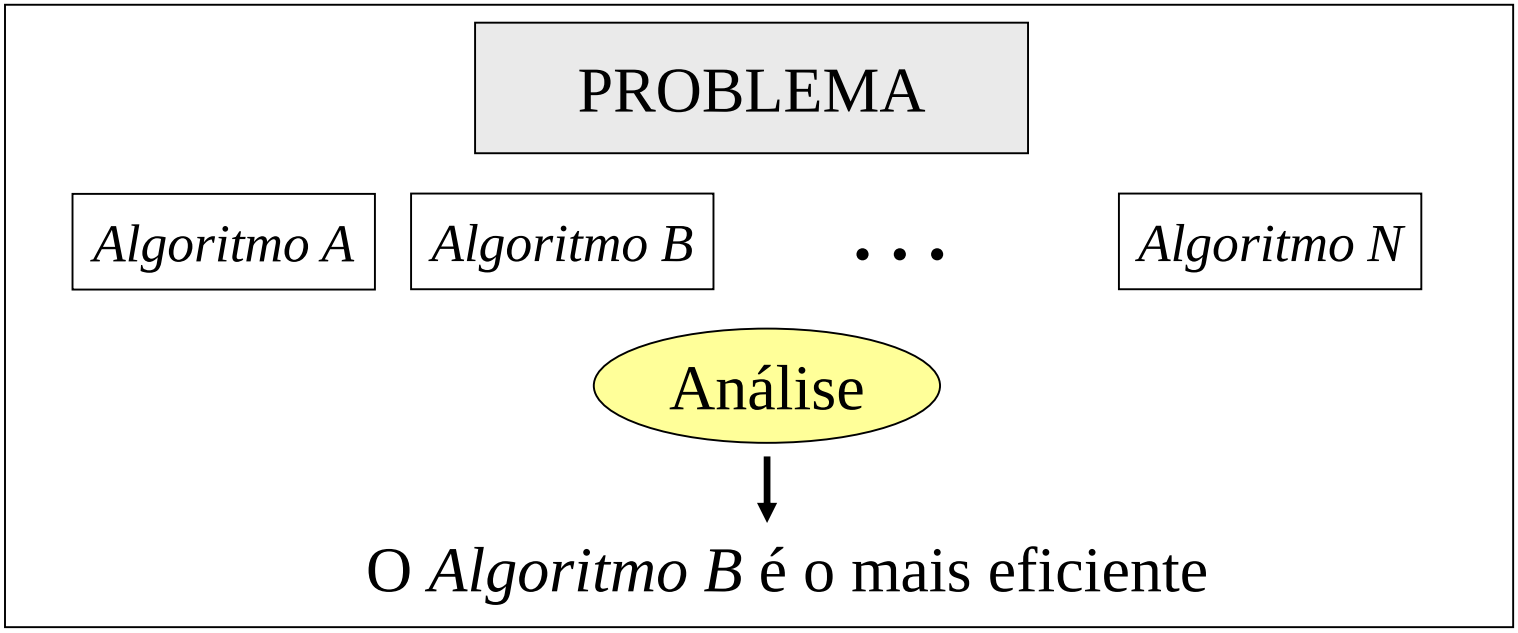
  - Verificar sua eficiência:

    Quanto tempo gasta?

    Quanto de memória usa?

Análise de Algoritmos

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



Análise de Algoritmos

- Como analisar a eficiência de um algoritmo?

- Existem dois tipos de eficiência:

✓ **Eficiência de tempo**: indica quão rápido um algoritmo é executado

✓ **Eficiência de espaço**: está relacionado com o espaço de memória que o algoritmo necessita.

Análise de Algoritmos

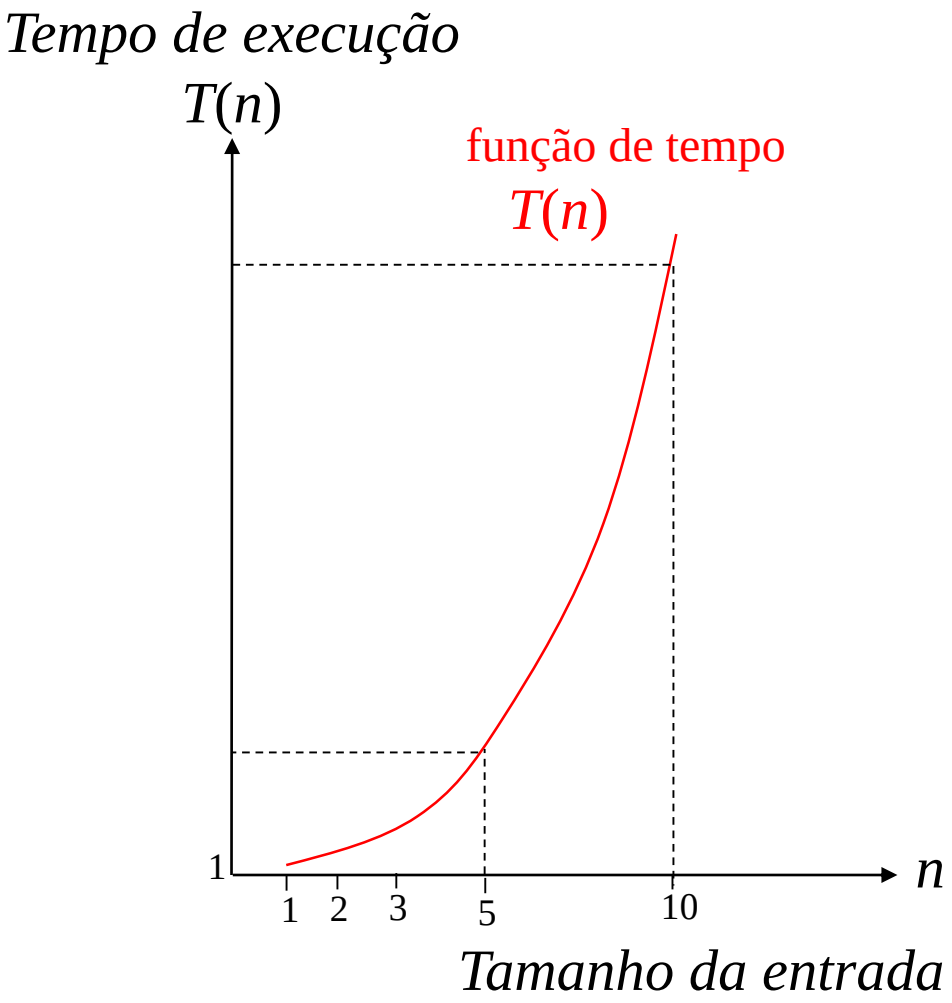
- Inicialmente, ambos tipos de eficiência eram importantes.
- Depois, a quantidade de espaço requerido por um algoritmo deixou de ser tão importante.
- Atualmente, voltou a ser importante devido ao enorme volume de dados que vem sendo produzido por diversas fontes: gps, celulares, satélites
- O famoso problema de processamento de *Big Data*

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 do algoritmo maior será a tempo de execução do algoritmo"
- Assim, a eficiência de um algoritmo é determinada em função do tamanho da entrada (um parâmetro *n*).
- Ou seja, se *n* é o tamanho da entrada, a eficiência de tempo do algoritmo é dada por uma função *T(n)*.

Complexidade de Tempo

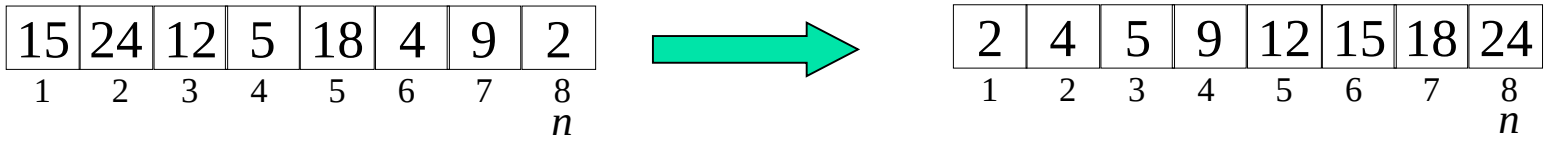
- O tempo necessário para resolver um problema cresce a medida que o tamanho da entrada *n* cresce.
- Existem algoritmos cujo tempo de execução cresce exponencialmente enquanto o tamanho do problema cresce apenas linearmente.



Tamanho da Entrada

- O tamanho da entrada de um algoritmo, geralmente, é um parâmetro inteiro ***n*** que denota a *quantidade de dados de entrada* do problema que esta sendo resolvido.
- Em alguns algoritmos o tamanho da suas entradas são definidas por mais de um parâmetro.

Exemplo 1. Ordenar uma lista $L = \{a_1, a_2, ...a_n\}$ com *n* números.



$\Rightarrow$ tamanho da entrada é *n* (quantidade total dos elementos).

Exemplo 2. Calcular o fatorial de um número inteiro *n*.
O único dado deste problema é *n* $\Rightarrow$ O cálculo do fatorial de *n* consiste em multiplicar os números de 1 até *n* $\Rightarrow$ o *tamanho da entrada* de qualquer algoritmo para resolver o problema será n.

Tamanho da Entrada

- *Exemplo 3*: Multiplicar duas matrizes $A_{n \times m}$ e $B_{m \times p}$.

  $\Rightarrow$ o *tamanho da entrada* é *n, m* e *p* $\Rightarrow$ a eficiência do algoritmo será dada por uma função com 3 variáveis: $T(n, m, p)$.

- *Exemplo 4*: Multiplicar duas matrizes quadradas $A_{n \times n}$ e $B_{n \times n}$

  $\Rightarrow$ o tamanho da entrada é n.

- *Exemplo 5*: Calcular x^n.

  $\Rightarrow$ o *tamanho da entrada* é *n.*

Unidades para medir o tempo de execução de um algoritmo

- **Unidade (física) de tempo**: segundo, milissegundo, etc.

  Neste caso, o tempo de execução depende do *computador* (*hardware*), *compilador*, *sistema operacional*, etc.

- **Unidade que não depende de fatores externos:** n° de operações

  - ✓ Número *total de operações* executadas pelo algoritmo.

    Identificar e contar todas as operações executadas pelo algoritmo é difícil e é desnecessário.

  - ✓ Número de vezes que a operação mais *importante* (*operação básica*) do algoritmo é executada.

    Basta identificar a operação que mais contribui para o tempo de execução total e contar o número de vezes que esta operação é executada.

Exemplos de operações básicas

- Problema da busca de uma chave em uma lista de *n* itens

  $\Rightarrow$ Operação básica: comparação de chaves

- Problema da ordenação de uma lista de *n* itens

  $\Rightarrow$ Operação básica: comparação de chaves

- Multiplicação de duas matrizes

  $\Rightarrow$ Operação básica: multiplicação de elementos

- Cálculo de x^n

  $\Rightarrow$ Operação básica: multiplicação da base *x*

- Cálculo do fatorial de *n*

  $\Rightarrow$ Operação básica: multiplicação de números (de 1 a *n*).

Eficiência de Algoritmos

- Para determinar a eficiência (tempo) de um algoritmo, basta determinar o número de vezes que a operação básica é executada. Este número é determinado em função do tamanho da entrada *n*.

  T(n) = número de vezes que a operação básica é executada

- Exemplo: $T(n) = 2n^2 + n + 4$ vezes.

- Geralmente, a operação básica (operação que consome mais tempo) está no laço (loop) mais interno

- *T(n)* é uma *estimativa* (aproximação) do *número total de operações* executadas pelo algoritmo.

Eficiência de Algoritmos

- Em alguns algoritmos, a eficiência (tempo) depende não apenas do tamanho da entrada, mas também da forma da entrada, ou seja, da instância do problema a ser resolvido.

- Exemplo: Problema de busca de uma chave *x* numa lista com n itens.

  Entradas (instâncias) de tamanho *n* = 6 e *x* = 5:

  L_1 = {**5**, 9, 1, 20, 4, 7}
  L_2 = {2, **5**, 14, 8, 22, 10}
  L_3 = {6, 9, 1, 20, **5**, 7}
  L_4 = {3, 7, 1, 10, 4, **5**}

- Note que o tempo do algoritmo de busca não é o mesmo para as diferentes entradas

Melhor Caso, Pior Caso e Caso Médio

- Seja um algoritmo para resolver um certo problema e sejam $E_1, E_2,..., E_m$, as m possíveis entradas de tamanho n.
- Suponha que $T_1(n), T_2(n), ..., T_m(n)$ sejam, respectivamente, os tempos para resolver cada uma das entradas.
- Daí, a eficiência (tempo) de:
  - Melhor Caso: $T_B(n) = \min \{T_i(n), i=1,...,m\}$
  - Pior Caso: $T_W(n) = \max \{Ti(n), i=1,...,m\}$
  - Caso Médio:

    $T_M(n) = p_1 \times T_1(n) + p_2 \times T_2(n) + ...+ p_m \times T_m(n)$ onde p_i é a probabilidade da entrada E_i ocorrer.

Melhor Caso, Pior Caso e Caso Médio

- *Exemplo 1*: Algoritmo de busca seqüencial para encontrar uma chave *x* em uma lista de *n* itens.

```
int BuscaSequencial(int L[ ], int n, int x){
int i = 0;
while( i<n && L[i] != x )
  i++;
if (i < n)
   return i; //retorna o índice do 1° elemento x encontrado
else return -1;  //retorna -1 caso x não seja encontrado
}
```

- A operação básica do algoritmo é L[i]!= x.

  $T_B(n) = 1$ (ou seja, no melhor caso, será executada 1 vez)

  $T_W(n) = n$ (ou seja, no pior caso, será executada *n* vezes)

  TM(n) = ???

Melhor Caso, Pior Caso e Caso Médio

- Calculando o caso médio:
  - Para o problema da busca podemos ter ($n+1$) entradas distintas:

    L_1,...., L_n, L_{n+1} (todas a listas de tamanho n)

  - Probabilidade p_i de cada lista entrada ocorrer:

    $p_i = 1/(n+1)$, $\forall i = 1,..., n+1$

  - Tempo do algoritmo para cada entrada (número de comparações):

    $T_i(n) = i$, $\forall i = 1,...,n+1$.

Melhor Caso, Pior Caso e Caso Médio

- Calculando o caso médio:

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

$$= \frac{1}{(n+1)} \times \sum_{i=1}^{n+1} \mathrm{i} = \frac{1}{(n+1)} \times \frac{(n+1)(n+2)}{2} = \frac{n+2}{2}$$

Melhor Caso, Pior Caso e Caso Médio

- *Exemplo2*: Determinar o maior e o menor elemento de uma lista L com n elementos inteiros.

```
void MaxMin1(int L[], int n,
             int &min, int &max)
{
   max = L[0]; min = L[0];
   for(int i=1; i<n; i++){
     if(L[i] > max) max=L[i];
     if(L[i] < min) min=L[i];
   }
}
```

Obs. `min` e `max` são retornados por referência.

```
void MaxMin2(int L[], int n,
             int &min, int &max)
{
 max = L[0]; min = L[0];
 for(int i=1; i<n; i++){
   if(L[i] > max) max=L[i];
   else
     if(L[i] < min) min=L[i];
 }
}
```

- Operação básica: comparações entre elementos da lista

L[i] > max ou L[i] < min

Melhor Caso, Pior Caso e Caso Médio

- No algoritmo *MaxMin1*: para qualquer entrada de tamanho *n*, a operação básica será executada *2(n-1)* vezes. Ou seja,

$$T_B(n) = T_W(n) = T_M(n) = 2(n-1), \text{ para } n>0.$$

- No algoritmo *MaxMin2*:
  - O *melhor caso* ocorre quando os elemento de L estão em ordem crescente. Neste caso, $T_B(n) =$ `n - 1`.
  - O *pior caso* ocorre quando os elementos de *L* estão em ordem decrescente. Neste caso, $T_W(n) =$ `2(n - 1)`.
  - No *caso médio*, podemos supor que `L[i]` é maior do que `max` a metade de vezes. Neste caso,

$$T_M(n) = (n-1) + (n-1)/2 = 3n/2-3/2,$$

Melhor Caso, Pior Caso e Caso Médio

- *Exemplo 3*: Dada *n* elementos, ordená-los em ordem crescente.

- Algoritmo de Ordenação por Inserção:
  - A lista é "dividida" em duas partes: *parte ordenada* e *parte não-ordenada.*
  - No início, a *parte ordenada* é formada somente pelo primeiro elemento da lista. A parte *não-ordenada* é formada pelos outros elementos (a partir da segunda posição).
  - Um a um os elementos da parte *não-ordenada* são inseridos na posição correta da parte *ordenada*, fazendo o deslocamento necessário de elementos.
  - O algoritmo termina quando não há mais elementos na parte *não-ordenada*.

Melhor Caso, Pior Caso e Caso Médio

```
void Isertion_Sort ( T L[],  int n )
    {
   T x;  int i;
    for (int j = 1; j<n ; j++) {  x =
   L[j]; i = j - 1;
       while (i >= 0 && L[i] > x ){
                L[i+1] = L[i];
                 i--;
       }//fim while
       L[i+1] = x;
     }//fim for
}
```

Operação básica:

`L[i] > x`

- Melhor Caso: ocorre quando a lista já está em ordem crescente (lista ordenada). Neste caso, $T_B(n) = n$.
- Pior Caso: ocorre quando os elemento da lista estão em ordem decrescente. Neste caso, $T_W(n) = n(n-1)/2$.

Ordem de Crescimento

- O processo de análise da eficiência de algoritmos leva em conta apenas um número pequeno de operações básicas
- Ou seja, estabelece apenas uma *medida aproximada* que deve ser usada de forma comparativa
- Neste processo, as constantes multiplicativas, aditivas e termos de mais baixa ordem são ignorados.
- O objetivo central é determinar apenas a ordem ou taxa de crescimento da operação básica
- Por exemplo: $T(n) = 2n^2 + n + 4 \Rightarrow T(n) \approx n^2$

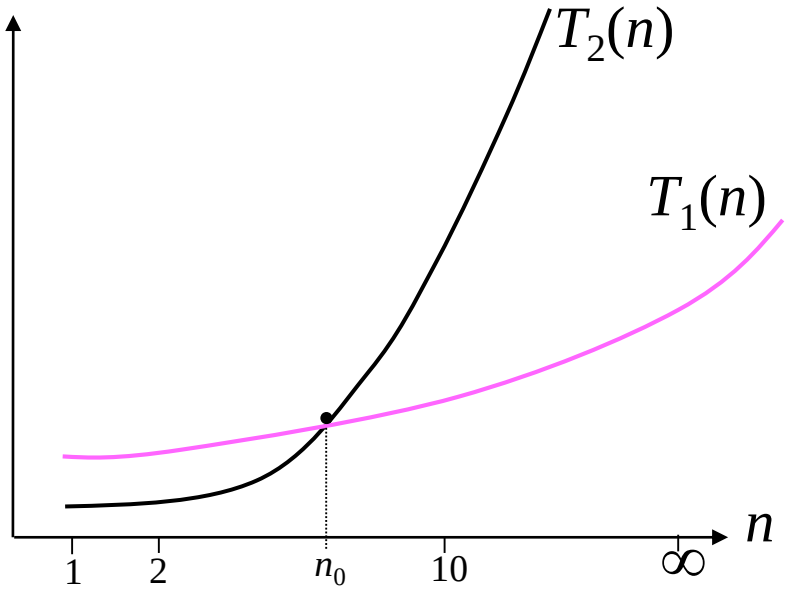
Ordem de Crescimento

- O tempo de um algoritmo aumenta com o tamanho da entrada *n*.
- Para valores pequenos de *n*, em geral, os algoritmos executam em pouco tempo
- Assim, a análise de algoritmos é realizada considerando valores grandes de *n*
- Ou seja, é analisado apenas a taxa de crescimento assintótico das funções de tempo.

Crescimento assintótico de funções

- Considere dois algoritmos:
  - `A1`: $T1(n) = 3n^2 + 4n + 50$
  - `A2`: $T_2(n) = n^3$

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



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

- Para n suficientemente grande o algoritmo `A1` é assintoticamente mais eficiente do que o algoritmo `A2`.

Notações Assintóticas

- Para comparar e classificar a ordem de crescimento de funções são utilizadas as notações assintóticas: O, Ω e Θ

- *Definições informais:*

  - $O(f(n))$ é o conjunto de todas as funções com ordem de crescimento **menor ou igual** a $f(n)$.

  - Exemplos:

    $n \in O(n^2)$, $100n + 5 \in O(n^2)$, $1/2n(n-1) \in O(n^2)$

    $n^3 \notin O(n^2)$, $0{,}00001n^3 \notin O(n^2)$, $n^4+n+1 \notin O(n^2)$

Notações Assintóticas

- $\Omega(f(n))$ é o conjunto de todas as funções com ordem de crescimento maior ou igual a $f(n)$.
  - *Exemplos:*

    $n^3 \in \Omega(n^2)$, $1/2n(n-1) \in \Omega(n^2)$, $100n + 5 \notin \Omega(n^2)$

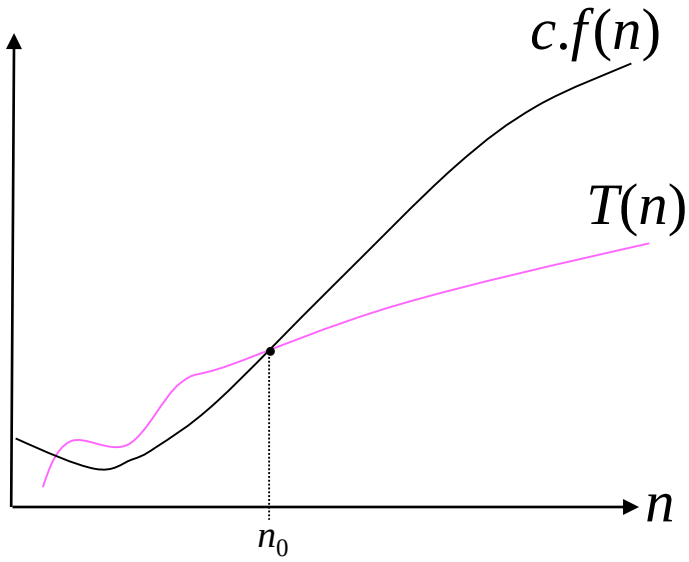
- *$\Theta(f(n))$ é o conjunto de todas as funções com ordem de crescimento igual a f(n).*
  - *Exemplo: $an^2 + bn + c \in \Theta(n^2)$, $a>0$*

Notação Assintótica O

- **Definição** (***Limite Superior***)

  Uma função $T(n) \in O(f(n))$ se e somente se existem constantes positivas c e n_0 tais que

  $$0 \leq T(n) \leq c.f(n), \forall\ n \geq n_0.$$



- Ou seja, a partir de n_0, $f(n)$ "majora" $T(n)$

- $f(n)$ é um ***limitante assintótico superior*** para $T(n)$.

Notação Assintótica *O*

- **Exemplo**: $T(n) = 3n^2 + 4n + 50 \in O(n^2)$

  Dem: Basta exibir duas constantes c e n_0 tais que

  $$T(n) \leq c.\ n^2\ ,\ \forall n \geq n_0.$$

  Se fizermos $c = 57$, temos

  $$3n^2 + 4n + 50 \leq 57n^2,\ \forall n \geq 1 = n_0.$$

- Pode-se mostrar também que

  $$T(n) = 3n^2 + 4n + 50 \in O(n^3) \text{ ou } O(n^4)$$

  entretanto estamos interessados no *menor limite superior* possível.

Notação Assintótica Ω

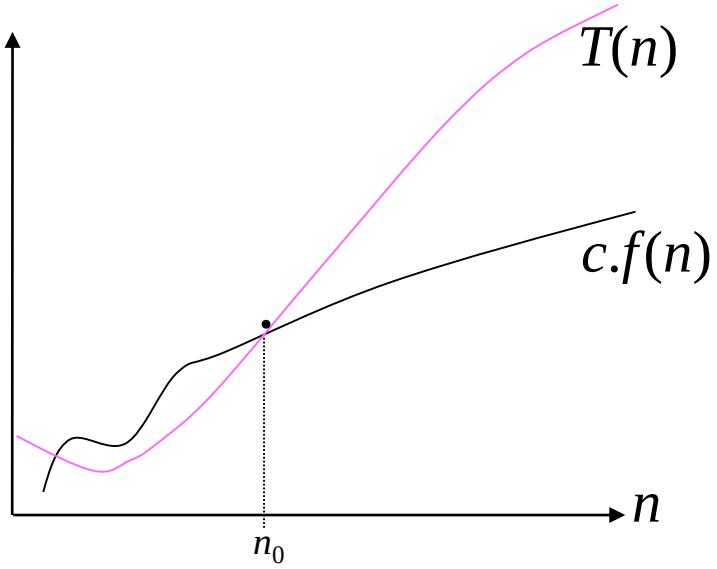
- **Definição** (***Limite Inferior***)

  Uma função $T(n) \in \Omega(f(n))$ se e somente se existem constantes positivas c e n_0 tais que

  $$T(n) \geq c.f(n), \forall\ n \geq n_0.$$

- Ou seja, a partir de n_0, $f(n)$ "minora" $T(n)$

- $f(n)$ é um ***limitante assintótico inferior*** para $T(n)$.



Notação Assintótica Ω

- **Exemplo**: $T(n) = n^4 - n \in \Omega(n^4)$

  Dem: Basta exibir duas constantes c e n_0 tais

  $$n^4 - 8n \geq c.\, n^4 , \forall n \geq n_0.$$

  Para $c = 1/2$, temos

  $$n^4 - 8n \geq 1/2 n^4 , \forall n \geq 3 = n_0.$$

- Pode-se mostrar também que

  $$T(n) = n^4 - n \in \Omega(n^3) \text{ ou } \Omega(n^2)$$

  entretanto estamos interessados no *maior limite inferior*.

Notação Assintótica Θ

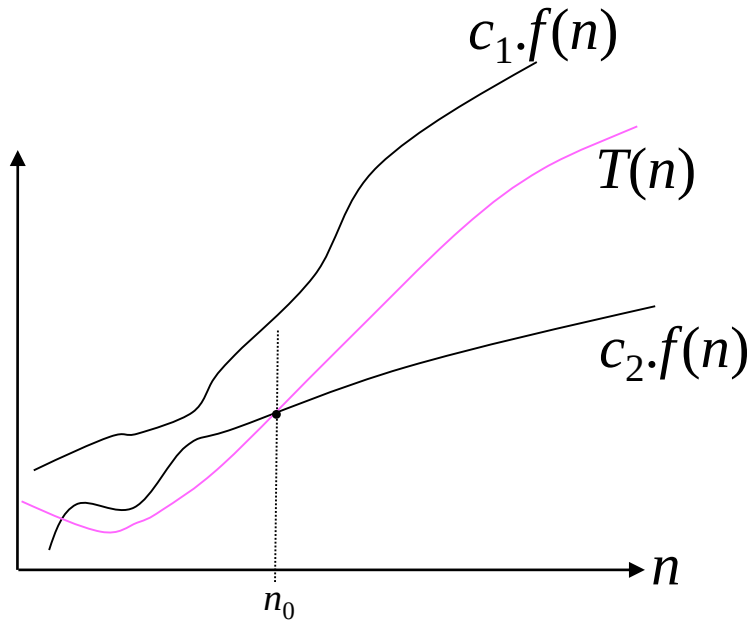
- **Definição** (***Limite Exato***)

  Uma função $T(n) \in \theta(f(n))$ se e somente se

  $T(n) \in \Omega(f(n))$ e $T(n) \in O(f(n))$.

- Ou seja, $T(n) \in \theta(f(n))$ se e somente se existem existem constantes c_1, c_2 e n_0 tais que

  $c_2.f(n) \leq T(n) \leq c_1.f(n), \forall\ n \geq n_0$.



- *f(n)* é um limite inferior e superior para *T(n)* apenas alterando as constantes.

- *T(n)* e *f(n)* possuem a mesma ordem de crescimento.

Notação Assintótica Θ

- **Exemplo**: $T(n) = n^2 + n \in \theta(n^2)$

  Dem: Pois, tomando $c_1 = 2$, $c_2 = 1$, $n_0 = 1$ temos

  $n^2 + n \leq \mathbf{2}.n^2$, $\forall n \geq 1 \Rightarrow n^2 + n \in O(n^2)$

  $n^2 + n \geq \mathbf{1}.n^2$, $\forall n \geq \mathbf{1} \Rightarrow n^2 + n \in \Omega(n^2)$

- **Abuso de notação**:
  - É muito comum se escrever

    $T(n) = O(f(n))$

    $T(n) = \Omega(f(n))$

    $T(n) = \theta(f(n))$

    Para indicar que $T(n)$ pertence a uma das classes de comportamento assintótico

Usando Limites para Comparar Ordens de Crescimento de funções

- Suponha que o limite $\lim_{n \to \infty} T(n)/f(n)$ exista. Assim,
  - Se $\lim_{n \to \infty} T(n)/f(n) = 0$, então $T(n)$ possui ordem de crescimento menor do que $f(n) \Rightarrow T(n) \in O(f(n))$.
  - Se $\lim_{n \to \infty} T(n)/f(n) = \infty$, então $T(n)$ possui ordem de crescimento > que $f(n) \Rightarrow T(n) \in \Omega(f(n))$
  - Se $\lim n \to \infty\ T(n)/f(n) = c > 0$ (constante), então $T(n)$ possui o mesmo ordem de crescimento que $f(n) \Rightarrow$ $T(n) \in O(f(n))$, $T(n) \in \Omega(f(n))$ ou $T(n) \in \Theta(f(n))$.

Usando Limites para Comparar Ordens de Crescimento de funções

- **Exemplo 1:** Provar que *T(n) = 1/2n(n-1)* e *f(n) = n^2* possuem a mesma ordem de crescimento.

$$\lim_{n\to\infty} \frac{1/2n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty} \frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}(1-\frac{1}{n}) = \frac{1}{2}$$

Como o valor do limite é uma constante >0, conclui-se que *1/2n(n-1)* $\in \Theta(n^2)$.

- **Exemplo 2:** Compare a ordem de crescimento de $log_2 n$ e $\sqrt{n}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{\sqrt{n}}{n} = 0$$

Daí, temos que $log_2 n \in O(\sqrt{n})$

Uso das Notações Assintóticas em Análise de Algoritmos

- A notação *O* é usada frequentemente para descrever o limite superior do tempo de execução do pior caso de um algoritmo.

- Este limite superior (descrito pela notação *O*) é calculado analisando apenas a estrutura global de um algoritmo.

- Por exemplo, no trecho de código, a estrutura do loop aninhado produz um limite superior $O(n^2)$ sobre o tempo de pior caso.

- Para cada valor de *i=1, ..., n-1,* a operação *L[j-1] > L[j]* é executada *j* vezes sendo que *j=n, n-1, ...,i+1*

```
for (i=1; i<n-1; i++)
   for (j=n; i>=i+1; i--)
        if(L[j-1] > L[j]){
            temp = L[j-1]
            L[j-1] = L[j]
            L[j] = temp;
        }
```

Uso das Notações Assintóticas em Análise de Algoritmos

- Quando dizemos que um algoritmo, no pior caso, é *O(f(n))* estamos dizendo que o tempo de execução do algoritmo nunca gasta mais do que *O(f(n))*
- Mas é importante mostrar que este limite superior não é (muito) folgado
- Uma forma de fazer isto é mostrar que existe pelo menos um caso que exige que o algoritmo atinja aquele limite.
- Assim, podemos dizer que o algoritmo nunca gasta mais do que *O(f(n))* e também que não podemos abaixar este limite superior.
- Logo, podemos dizer que o algoritmo é $\Theta(f(n))$.

Uso das Notações Assintóticas em Análise de Algoritmos

- Por exemplo, o algoritmo de ordenação por inserção é $O(n^2)$.
- Se a entrada estiver em ordem decrescente serão realizadas $n(n-1)/2$ comparações.
- Portanto, o algoritmo de ordenação por inserção, no pior caso, é $\Theta(n^2)$.

Propriedades das Notação Assintótica

- **Regra das Somas**

Se $T_1(n) \in O(f(n))$ e $T_2(n) \in O(g(n))$ então

$T_1(n) + T_2(n) \in O(\max\{f(n), g(n)\})$.

Dem:

Se $T_1(n) \in O(f(n))$, então existem c_1 e n_1 : $T_1(n) \leq c_1 f(n)$, $\forall\ n \geq n_1$ (1)

Se $T_2(n) \in O(g(n))$, então existem c_2 e n_2 : $T_2(n) \leq c_2 g(n)$, $\forall\ n \geq n_2$ (2)

Queremos provar que existem c_3 e n_3 tais que

$T_1(n) + T_2(n) \leq c_3 \max\{f(n), g(n)\}$, $\forall\ n \geq n_3$.

Sejam $c_3 = c_1 + c_2$, $n_3 = \max\{n_1, n_2\}$ e $h(n) = \max\{f(n), g(n)\}$

Para todo $n \geq n_3$, temos $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n) \leq c_1 h(n) + c_2 h(n)$

$= (c_1 + c_2) h(n) = c_3 \max\{f(n), g(n)\}$.

Logo, $T_1(n) + T_2(n) \in O(\max\{f(n), g(n)\})$.

Propriedades das Notação Assintótica

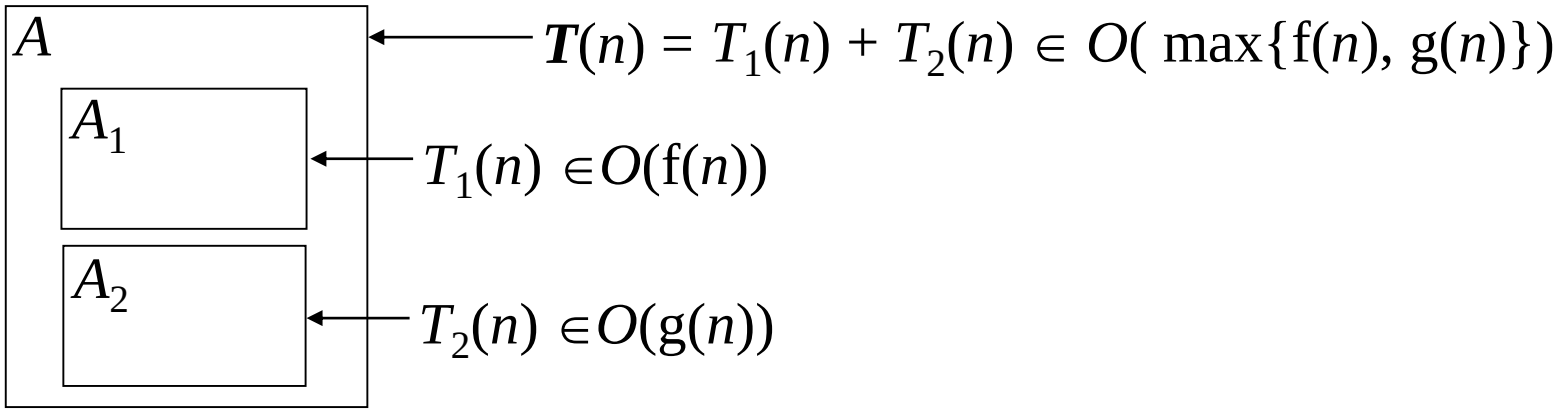
- **Regra do Produto**

  Se $T_1(n) \in O(f(n))$ e $T_2(n) \in O(g(n))$ então $T_1(n).T_2(n) \in O(f(n).g(n))$

  Dem: Exercício. Considere $c_3 = c_1.c_2$ e $n_3 = \max\{n_1, n_2\}$

- Em análise de algoritmos, a *regra das somas* é utilizada da seguinte maneira:

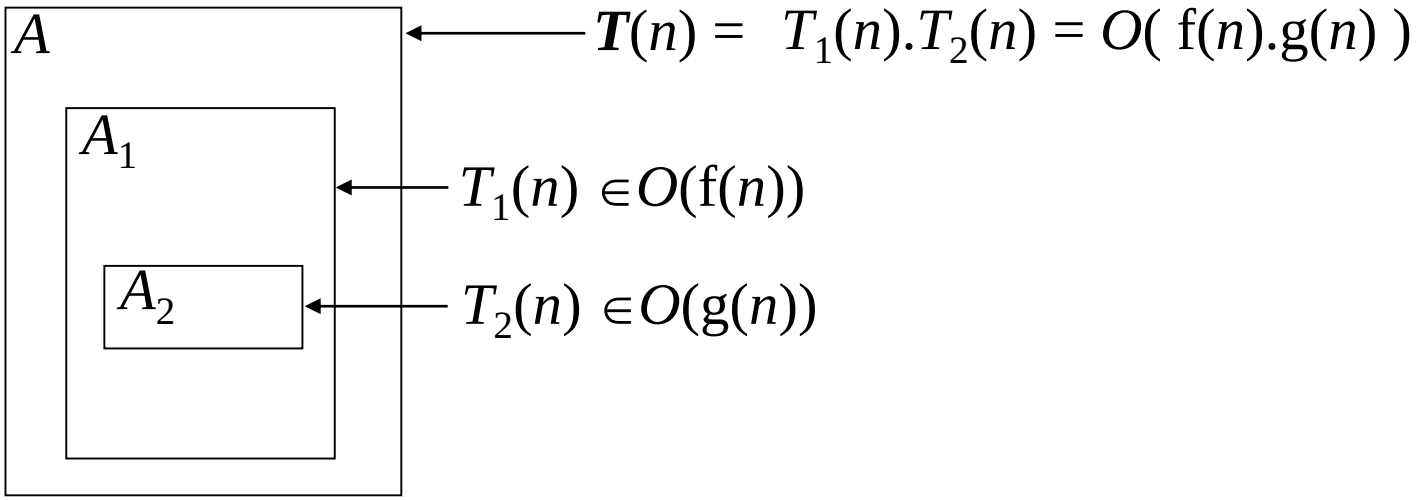
  Se um algoritmo ***A*** se divide em duas partes independentes $\boldsymbol{A}_1$ e $\boldsymbol{A}_2$, onde o tempo de $\boldsymbol{A}_1$ é $T_1(n) \in O(f(n))$ e o de $\boldsymbol{A}_2$ é $T_2(n) \in O(g(n))$ então o tempo de ***A*** é $T_1(n) + T_2(n) \in O(\max\{f(n), g(n)\})$.



Propriedades das Notação Assintótica

- A *regra do produto* é utilizada da seguinte maneira:

  Se um algoritmo ***A*** contém dois "aninhamentos" $\boldsymbol{A}_1$ e $\boldsymbol{A}_2$, onde o tempo de $\boldsymbol{A}_1$ é $T_1(n) \in O(\mathrm{f}(n))$ e o tempo de $\boldsymbol{A}_2$ é $T_2(n) \in O(\mathrm{g}(n))$ então, o tempo de ***A*** será $T_1(n).T_2(n) \in O(\mathrm{f}(n).\mathrm{g}(n))$



Propriedades das Notação Assintótica

- **Outras propriedades:**
  - $f(n) \in O(f(n))$ (reflexiva). Também válidas para Ω e Θ.
  - *Se* $f(n) \in O(g(n))$ *e* $g(n) \in O(h(n))$, *então* $f(n) \in O(h(n))$ *(transitiva). Também válidas para* Ω *e* Θ.
  - $f(n) \in \Theta(g(n))$ *se e somente se* $g(n) \in \Theta(f(n))$ *(simetria)*
  - $f(n) \in O(g(n))$ *se e somente se* $g(n) \in \Omega(f(n))$
  - *Se* $f(n) \in O(g(n))$ *então* $f(n)+g(n) \in O(g(n))$
  - $c.O(f(n)) = O(f(n))$, c = *constante*
  - $O(f(n)) + O(f(n)) = O(f(n))$
  - $O(O(f(n))) = O(f(n))$
  - $f(n).O(g(n)) = O(f(n).g(n))$

Funções Assintóticas Básicas

- A eficiência (tempo) de um grande número de algoritmos recaem nas seguintes classes de funções:

1	Constante*
$\log n$	logarítmica
n	linear
$n \log n$	$n \log n$
n^2	quadrática
n^3	cúbica
$2^{n,}$	exponencial
$n!$	Fatorial
n^n	exponencial

*Um algoritmo é $O(1)$ se ele contém apenas um número constante de "comandos simples".

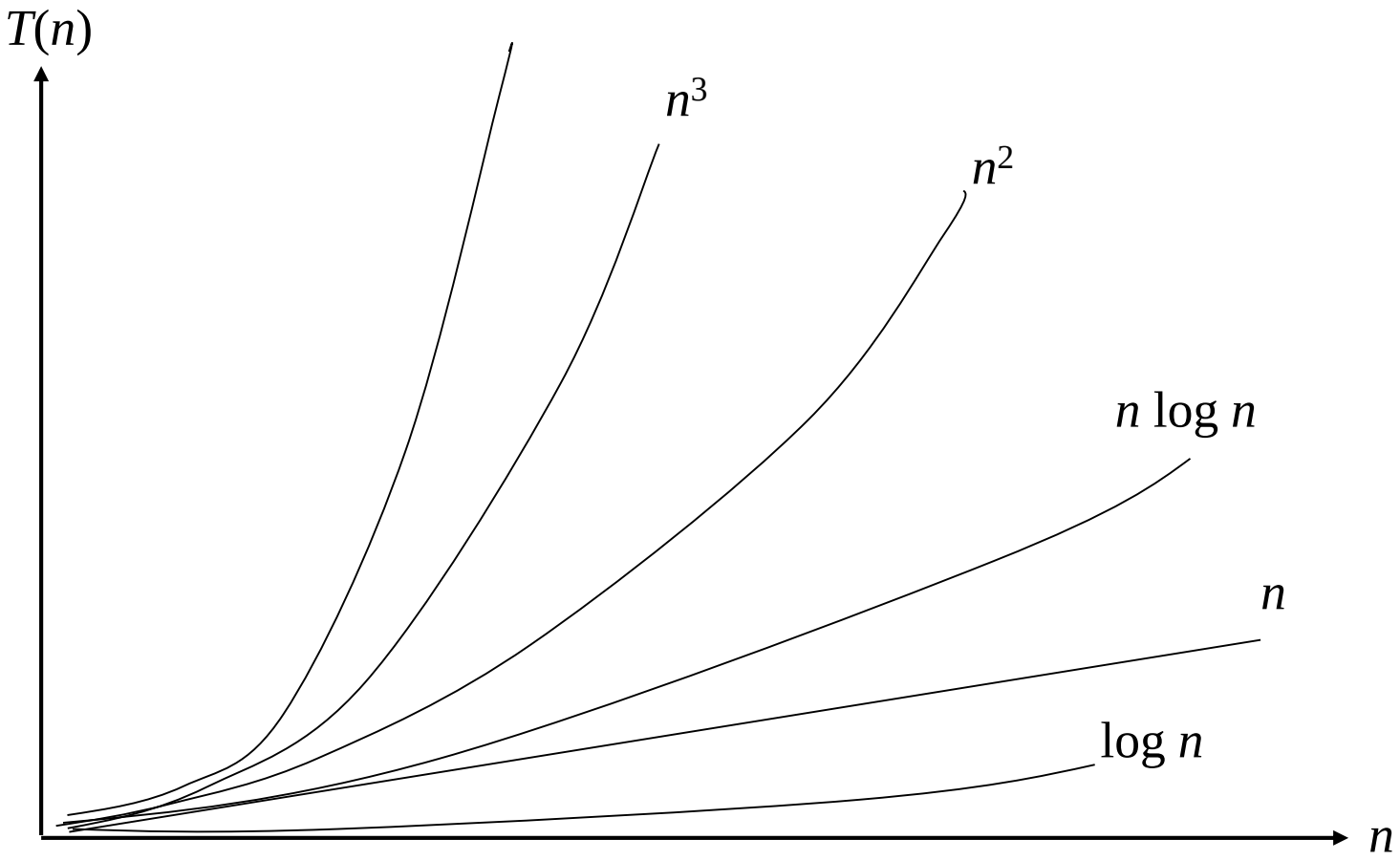
Crescimento de funções

- Valores de várias funções importantes em análise de algoritmos

n	$Log_2\, n$	n	$n\ log_2 n$	n^2	n^3	2^n	$n!$	n^n
10	3,3	10	$3{,}3\times10$	10^2	10^3	10^3	$3{,}6\times10^6$	10^{10}
10^2	6,6	10^2	$6{,}6\times10^2$	10^4	10^6	$1{,}3\times10^{30}$	$9{,}3\times10^{157}$	10^{200}
10^3	10	10^3	$1{,}0\times10^4$	10^6	10^9			
10^4	13	10^4	$1{,}3\times10^5$	10^8	10^{12}			
10^5	17	10^5	$1{,}7\times10^6$	10^{10}	10^{15}			
10^6	20	10^6	$2{,}0\times10^7$	10^{12}	10^{18}			

- A função logarítmica $log_2\, n$ cresce mais lentamente
- As funções exponencial (2^n e n^n) e fatorial ($n!$) crescem rapidamente

Crescimento de funções



Crescimento de funções

Exemplo de Tempos num computador Atual

Algoritmo	Tempo	Tamanho da entrada		
		$n = 20$	$n = 40$	$n = 60$
A1	n	0,0002 seg	0,0004 seg	0,0006 seg
A2	$n.\log n$	0,0009 seg	0,0021 seg	0,0035 seg
A3	n^2	0,004 seg	0,016 seg	0,036 seg
A4	n^3	0,08 seg	0,64 seg	2,16 seg
A5	2^n	10 seg	127 dias	3660 séculos

Uma operação é executado em 0,00001seg (10 micro-segundos

Crescimento de funções

- Suponha que temos dois computadores novos X e Y.
  Computador X é 100 vezes mais rápido do que o computador atual
  Computador Y é 1000 vezes mais rápido do que o computador atual

Aumento do tamanho da entrada

Algoritmo	Tempo	Maior problema solucionável em **1 hora**		
		Computador Atual	Computador X	Computador Y
A1	n	N_1	$100N_1$	$1000N_1$
A2	$n.\log n$	N_2	$22{,}5N_2$	$140{,}2N_2$
A3	n^2	N_3	$10N_3$	$31{,}62N_3$
A4	n^3	N_4	$4{,}64N_4$	$10N_4$
A5	2^n	N_5	$N_5 + 4$	$N_5 + 10$

Conceitos e Fórmulas Matemáticas usados em Análise de Algoritmos

- **Somas**

- Progressões aritméticas

$$1+2+3+\ldots+n=\sum_{i=1}^{n} i=\frac{n(n+1)}{2}$$

$$\sum_{i=k}^{n} 1=n-k+1, \quad (k \leq n)$$

$$\sum_{i=1}^{n} 1=n$$

$$a_1+a_2+a_3+\ldots+a_n=\frac{a_1(q^n-1)}{q-1},$$

onde q é o fator, ou seja $a_i.q=a_{i+1}$

- Progressões geométricas

$$c^0+c^1+c^2+\ldots+c^n=\sum_{i=0}^{n} c^i=\frac{c^{n+1}-1}{c-1}, \quad c \neq 1$$

$$\sum_{i=0}^{n} 2^i=2^{n+1}-1$$

Conceitos e Fórmulas Matemáticas usados em Análise de Algoritmos

- **Outras fórmulas**

$$\sum_{i=0}^{n} i^k = 1^k + 2^k + 3^k ... + n^k \approx \frac{1}{k+1} n^{k+1}$$

$$\sum_{i=1}^{n} i2^i = 1.2 + 2.2^2 + ... + n2^n = (n-1)2^{n+1} + 2$$

$$\sum_{i=1}^{n} \log i \approx n \log n$$

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \quad \text{quando } n \to \infty$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 ... + n^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + ... + \frac{1}{n} \approx \ln n + \gamma$$

($\gamma \approx 0{,}5772...$ Constante de Euler)

$$\sum_{i=k}^{n} ca_i = c\sum_{i=k}^{n} a_i$$

$$\sum_{i=k}^{n} (a_i \pm b_i) = \sum_{i=k}^{n} a_i \pm \sum_{i=k}^{n} b_i$$

$$\sum_{i=k}^{n} a_i = \sum_{i=k}^{m} a_i + \sum_{i=m+1}^{n} a_i$$

onde, $k \leq m \leq n$

$$\sum_{i=k}^{n} (a_i - a_{i-1}) = a_n - a_{k-1}$$

Conceitos e Fórmulas Matemáticas usados em Análise de Algoritmos

- **Logaritmos**

Definição: $\text{Log}_b\, n = x \Leftrightarrow b^x = n$

$$b^{Log_b n} = n$$

$$n = m \Leftrightarrow Log_b n = Log_b m$$

Logaritmo de um produto: $Log_b(n.m) = Log_b n + Log_b m$

Logaritmo de uma divisão: $Log_b(\frac{n}{m}) = Log_b n - Log_b m$

Logaritmo de uma potência: $Log_b n^x = x.Log_b n$

Troca de base: $Log_b n = \frac{Log_a n}{Log_a b}$

Análise de Algoritmos Não-Recursivos

- A estratégia principal da análise de algoritmos não-recursivos é estabelecer um *somatório* que representa o tempo de execução.

Passos para Analisar um Algoritmo Não-Recursivo

1. Escolha o parâmetro *n* indicando o tamanho da entrada;
2. Identifique a *operação básica* do algoritmo
3. Verifique se o tempo do algoritmo (número de vezes que a operação básica é executada) é depende somente de *n*. Ou seja, verifique se existem tempos de melhor, pior e médio caso;
4. Estabeleça um *somatório* expressando o número de vezes que a operação básica é executada;
5. Utilize fórmulas e regras para manipulação de somas, encontre uma fórmula final (função $T(n)$) e estabeleça sua ordem de crescimento.

Análise de Algoritmos Não-Recursivos

- **Exemplo 1**: Encontrar o maior elemento de uma lista (arranjo) com n elementos.

```
T MaxElement(T L[],int n ){
   T max = L[0];
    for (int i = 1; i<n; i++)
       if( L[i] > max )
         max = L[i];
    return max;
}
```

1. Tamanho da entrada: *n*
2. Operação básica: *L[i]>max*
3. O número de execuções da operação básica será o mesmo para qualquer entrada (lista de tamanho *n*), ou seja, não existem casos: melhor, pior e médio.

4. Seja T(n) o número de vezes que a operação básica é executada. Note que a operação básica é executada 1 vez em cada execução do loop for. Ou seja, para cada valor da variável i, a operação básica é executado 1 vez (i = 1, ..., n-1). Portanto, temos o seguinte somatório: $T(n) = \sum_{i=1}^{n-1} 1 = n-1$

5. Tempo do algoritmo: $T(n) = n\text{-}1 = \Theta(n)$

Análise de Algoritmos Não-Recursivos

- **Exemplo 2**: Verificar se todos os elementos em uma lista são diferentes ou não.

```
bool MaxElement(T L[],  int n
    ){
    for (int i = 0; i<n-1;
        i++)
        for (int j = i+1; j<n;
            j++)
            if( L[i] == L[j] )
                return false;
    return true;
}
```

Se dois elementos são iguais, o algoritmo retorna *false*. Caso contrário, se todos os elementos são diferentes, retorna *true*.

A operação básica *L[i]==L[j]* será executada 1 vez para cada par de valores de *i* e *j* (no pior caso). Assim,

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} \left[(n-1) - (i+1) + 1 \right] = \sum_{i=0}^{n-2} \left[(n-1) - i \right] = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i$$

$$T(n) = (n-1) \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i = (n-1)(n-1) - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} = \Theta(n^2)$$

Análise de Algoritmos Não-Recursivos

- **Exemplo 3**: Calcular a multiplicação de duas matrizes A e B, ambos de ordem `n×n`.

```
void MultiplicaM(T **A,  T **B, T **C, int n){
    for (int i = 0; i<n; i++)
        for (int j = 0; j<n; j++){
            C[i][j] = 0;
            for (int k = 0; k<n; k++)
              C[i][j] += A[i][k]*B[k][j];
}
```

Operação básica *A[i][k]*B[k][j]* será executada 1 vez para cada combinação de valores de *i*, *j* e *k* (para qualquer caso).

$$T(n) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3 \in \Theta(n^3)$$

Análise de Algoritmos Recursivos

- A análise deste tipo de algoritmos consiste em determinar um tipo especial de equação chamada *relação de recorrência*.

- **Relações de Recorrências**

  Uma relação de recorrência ou simplesmente *recorrência* é uma forma de definir uma função através de uma expressão que contém a mesma função.

  Exemplos:

  - $f(n) = 2f(n-1) + 1$
  - $f(n) = f(n-1) + f(n-2)$
  - $f(n) = f(n/2) + 1$
  - $f(n) = n.f(n-1)$
  - $f(n) = n + f(n-1)$

  - $f(n) = \begin{cases} 1 & se\ n = 1 \\ 2f(n\text{-}1) + 1 & se\ n > 1 \end{cases}$

  - $f(n) = \begin{cases} 2 & se\ n = 1 \\ 2f(n/2) + 3n + 2 & se\ n > 1 \end{cases}$

Análise de Algoritmos Recursivos

Passos para Analisar um Algoritmo Recursivo

1. Escolha o parâmetro *n* indicando o tamanho da entrada;

2. Identifique a *operação básica* do algoritmo;

3. Verifique se o tempo do algoritmo (número de vezes que a operação básica é executada) pode variar para diferentes entradas do mesmo tamanho. Caso o tempo dependa da entrada, calcule separadamente os tempos de melhor caso, pior caso e caso médio;

4. Estabeleça uma *relação de recorrência* para o número de vezes que a operação fundamental é executada. Identifique o *caso base* ou *condição de parada* do algoritmo (tempo para o menor valor de *n*);

5. Resolva a relação de recorrência obtendo o tempo $T(n)$ do algoritmo ou encontrar a ordem de crescimento de $T(n)$.

Análise de Algoritmos Recursivos

- **Exemplo 1**: Calcular o fatorial de um número *n*.

```
int FAT(int n){
      if (n == 0) return 1;
      else {
       return n * FAT(n - 1);
      }
}
```

1. Tamanho da entrada: *n*
2. Operação básica: multiplicação *n**FAT(*n*–1)
3. Existe só um caso para o tempo de execução. Isto é, não há melhor caso ou caso médio

4. Seja *T*(*n*) a complexidade do algoritmo para uma entrada de tamanho n. Para n>0, o algoritmo executará a operação básica 1 vez e em seguida fará a chamada (recursiva) para n-1. Assim,

$$T(n) = 1 + T(n-1) \quad n > 0$$
$$T(0) = 0$$

Análise de Algoritmos Recursivos

5. Resolvendo a relação de recorrência

Método da Substituição:

$T(n) = 1 + T(n-1)$, substituir $T(n-1) = 1 + T(n-2)$

$T(n) = 1 + 1 + T(n-2)$, substituir $T(n-2) = 1 + T(n-3)$

$T(n) = 1 + 1 + 1 + T(n-3)$

Generalizando (depois de k passos):

$T(n) = 1+1+ \ldots.+1 + T(n-k) = k + T(n-k)$ .....(*)

Condição de parada: $T(0) = 0$,

Para finalizar fazemos $n - k = 0 \Rightarrow k = n$

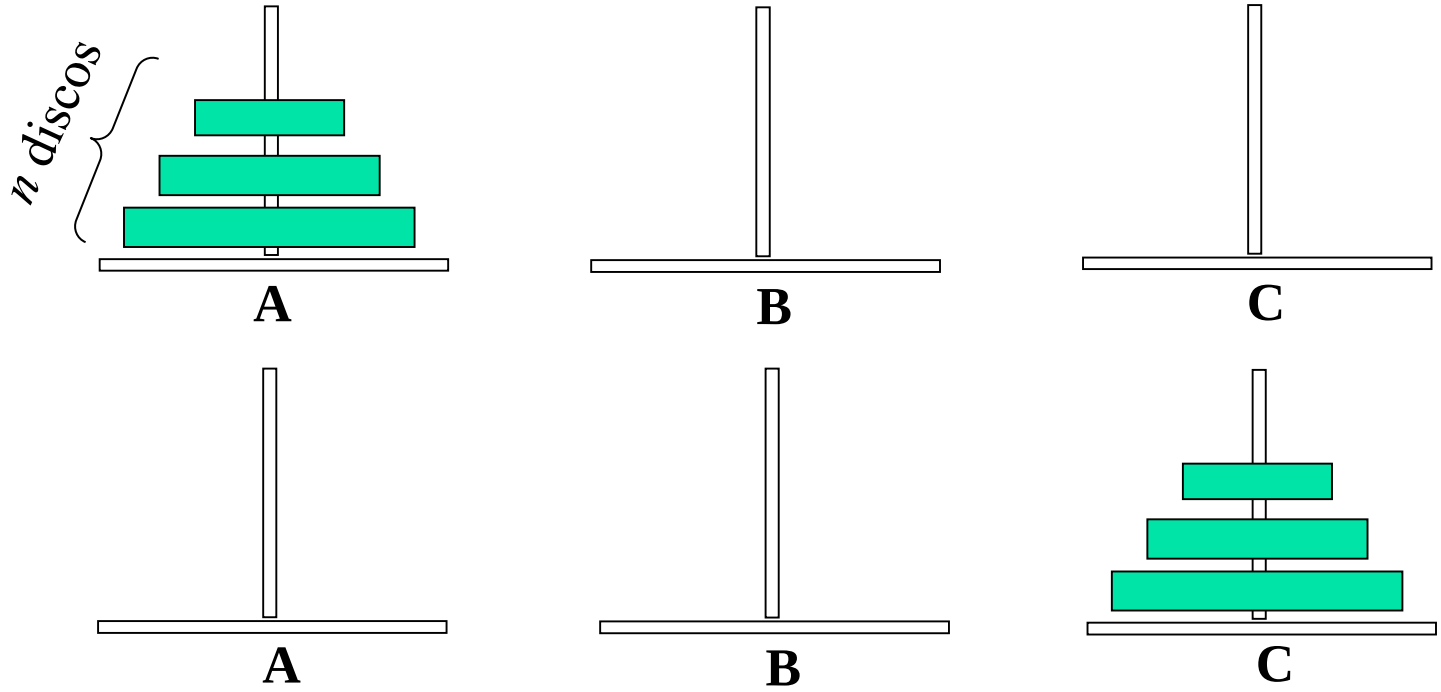
Substituindo k em (*): $T(n) = n + T(0) = n \in \Theta(n)$.

Análise de Algoritmos Recursivos

- **<u>Exemplo 2</u>: Problema das Torres de Hanói**

  Deslocar *n* discos do pino **A** para o pino **C** usando o pino **B**.

  - Só um disco pode ser movimentado de cada vez;
  - Um disco maior no pode ser colocado sobre um disco menor;
  - Realizar o menor número de movimentos.



Análise de Algoritmos Recursivos

```
void HANOI (int n, char A, char B, char C ) {
    if ( n == 1)
      cout<<“ mova disco”<< n <<“de”<<A<<“para”<< C<<endl;
    else {
        HANOI ( n-1, A, C, B );
        cout<<“ mova disco”<< n <<“de”<<A<<“para”<< C<<endl;
        HANOI ( n-1, B, A, C);
}
```

- A *operação básica* corresponde a um movimento realizado.

Seja $T(n)$ o tempo de execução do algoritmo.

Para o caso base (n=1) é feito 1 movimento (ou seja, $T(1) = 1$).

Para n>1, são realizadas duas chamadas recursivas para n-1 e além disso é feito 1 movimento. Assim, temos a seguinte recorrência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1 \\ 2.T(n-1)+1, & \text{se } n > 1 \end{cases}$$

Análise de Algoritmos Recursivos

- **Resolvendo a recorrência (método da substituição):**

$T(n) = 2.T(n-1) + 1$, Substituir $T(n-1) = 2.T(n-2) + 1$

$T(n) = 2(\ 2.T(n-2) + 1\ ) + 1$,

$T(n) = 2^2.T(n-2) + 2 + 1$, Substituir $T(n-2) = 2.T(n-3) + 1$

$T(n) = 2^3.T(n-3) + 2^2 + 2 + 1$

.....

Generalizando (depois de k passos):

$T(n) = 2^k.T(n-k) + 2^{k-1} + 2^{k-2} + ... + 2 + 1$ (*)

Usando a condição parada: $n - k = 1 \Rightarrow k = n - 1$

Substituindo o valor de k em (*):

$T(n) = 2^{n-1}T(1) + 2^{n-2} + ... + 2^1 + 2^0 = \sum_{i=0}^{n-1} 2^i = 2^n - 1$

$T(n) = 2^n - 1 \in \Theta(2^n)$

Análise de Algoritmos Recursivos

- **Exemplo 3**: Algoritmo recursivo para determinar os dígitos da representação binária de um numero decimal inteiro *n*>0.

```
void Binario(int n){
    if (n == 1) cout<<1;
    else {
     Binario(n/2);
     cout<<n%2; //imprime o resto
    }
}
```

- A *operação básica* corresponde à impressão de um dígito.

  Seja *T*(*n*) a complexidade do algoritmo. Para o caso base (*n*=1), *T*(1) = 1 .

  Para *n*>1, é feito uma chamada recursiva para *n*-1 e em seguida é feito 1 impressão. O tempo de cada chamada recursiva é *T*(*n*-1).

  Temos a seguinte recorrência:

$$T(n) = \begin{cases} 1, & \text{se } n = 1 \\ T(\lfloor n/2 \rfloor) + 1, & \text{se } n > 1 \end{cases}$$

Análise de Algoritmos Recursivos

- **Resolvendo a recorrência (método da substituição):**

  Sem perda de generalidade, podemos supor que n é potência de 2 e portanto $\lfloor n/2 \rfloor = n/2$

  $T(n) = T(n/2) + 1$

  $T(n) = T(n/4) + 1 + 1 \quad = T(n/2^2) + 2$

  $T(n) = T(n/8) + 1 + 1 + 1 = T(n/2^3) + 3$

  Após de k passos (generalizando):

  $T(n) = T(n/2^k) + k$ ..... (*)

  Condição parada: $n/2^k = 1 \Rightarrow 2^k = n$

  Aplicando $\log_2$: $\log_2 2^k = \log_2 n \Rightarrow k = \log_2 n$

  Substituindo k em (*):

  $T(n) = T(n/2^k) + k = T(1) + \log_2 n = 1 + \log_2 n \in \Theta(\log_2 n)$.

  Se $T(n) \in \Theta(\log_2 n)$ para n potencia de 2 então, $T(n) \in \Theta(\log_2 n)$, $\forall n > 1$.

Resolução de Relações de Recorrências

- Resolver a seguinte relação de recorrência:

$T(n) = n^2 + T(n-1), \quad T(1) = 1$

$T(n) = n^2 + (n-1)^2 + \underline{T(n-2)}$

$T(n) = n^2 + (n-1)^2 + (n-2)^2 + \underline{T(n-3)}$

Generalizando:

$T(n) = n^2 + (n-1)^2 + (n-2)^2 + \ldots + T(n-k)$

Utilizando a condição de parada, fazemos $n - k = 1 \Rightarrow \quad k = n - 1$

$T(n) = n^2 + (n-1)^2 + (n-2)^2 + \ldots + T(1)$

$T(n) = n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2 \quad \Rightarrow T(n) = \sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$

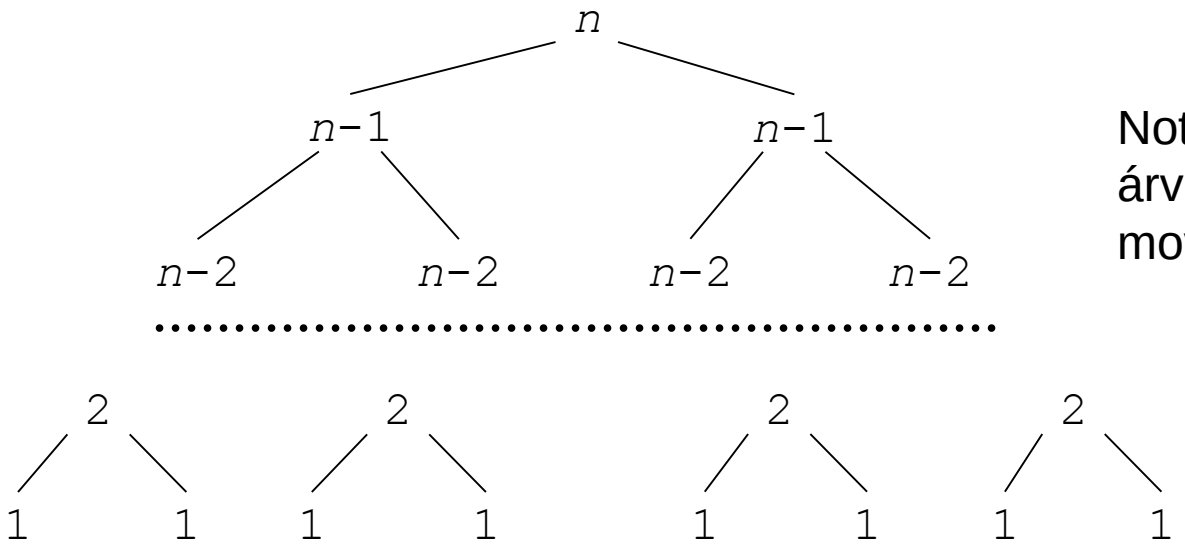
$T(n) \in \Theta(n^3)$

Resolução de Relações de Recorrências

- **Método da Árvore de Recursão**

Exemplo1 : $T(n) = 2T(n-1) + 1$, $T(1) = 1$ (Algoritmo HANOI)

Devemos mostrar a árvore das chamadas realizadas pelo algoritmo recursivo. No algoritmo HANOI, na chamada para *n* (chamada principal) são feitas 2 chamadas para *n*-1, depois 4 chamadas para *n*-2, daí 8 chamadas para *n*-3, etc. Para *n* = 1 não é feita nenhuma chamada. A cada chamada é executado 1 movimento (operação básica).



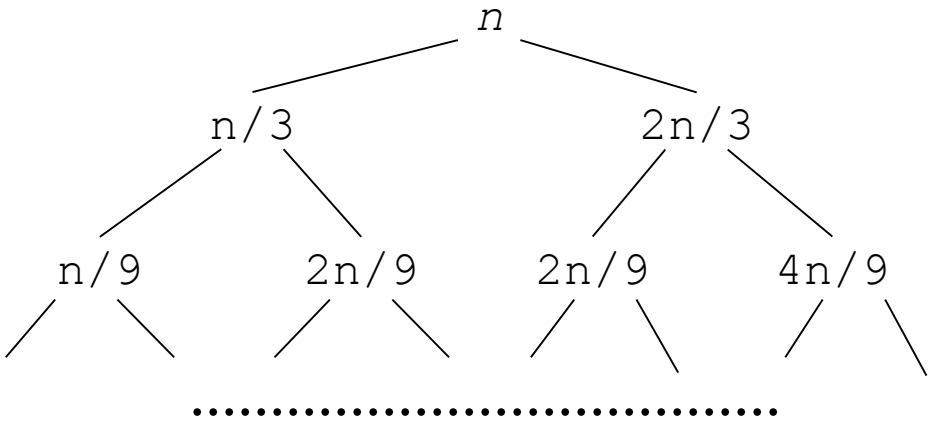
Note que para cada nó da árvore é executado 1 movimento (ou 1 chamada).

Resolução de Relações de Recorrências

- **Exemplo 2:** Resolver $T(n) = n + T(n/3) + T(2n/3), \quad n \geq 1$

  Nesta recorrência, a primeira chamada executa n operações e são feitas duas chamadas para entradas de tamanhos $n/3$ e $2n/3$.

  Na chamada para $n/3$ serão executadas $n/3$ operações e serão feitas duas chamadas para entradas de tamanhos $n/9$ e $2n/9$ ($T(n/3) = n/3 + T(n/9) + T(2n/9)$). E assim sucessivamente.



A cada nível da árvore de recursão são executadas n operações.

Queremos saber qual é o número de níveis da árvore. Para isto analisemos o caminho mais longo da raiz até uma folha:

$n, (2/3)n, (2/3)^2n, ..., (2/3)^kn, ..., 1.$

Deve-se determinar o valor de k tal que $(2/3)^k n = 1 \Leftrightarrow (3/2)^k = n$.

Aplicando $\log_{3/2}$: $k = \log_{3/2} n \Rightarrow$ O número de níveis da árvore é: $\log_{3/2} n$

Logo número total de operações é $T(n) = (\log_{3/2} n)\, n \in O(n \lg n)$

Resolução de Relações de Recorrências

- **Método Mestre**

Usado para determinar um limitante assintótico de recorrências da forma: $T(n) = aT(n/b) + f(n)$.

Teorema Mestre: Seja a recorrência

$T(n) = aT(n/b) + f(n)$, (com n potencia de b)

$T(1) = c \geq 0$

onde: $a \geq 1$, $b \geq 2$ e $f(n)$ é uma função positiva tal que $f(n) \in \Theta(n^d)$ com $d \geq 0$. Então,

1) Se $a < b^d$ então, $T(n) \in \Theta(n^d)$

2) Se $a = b^d$ então, $T(n) \in \Theta(n^d \log_b n)$

3) Se $a > b^d$ então, $T(n) \in \Theta(n^{\log_b a})$

Resolução de Relações de Recorrências

1) Encontre um limite assintótico para

$T(n) = 9T(n/3) + n, \quad T(1) = 1.$

$a = 9$, $b = 3$ e $f(n) = n \in \Theta(n^1)$, $d = 1$

Como $a > b^d$ então aplica-se o caso 3 do teorema mestre:

$T(n) \in \Theta(n^{\log_3 9}) = \Theta(n^2)$

2) Resolver $T(n) = T(n/4) + 1$, $T(1) = 1$.

$a = 1$, $b = 4$ e $f(n) = 1 \in \Theta(n^0)$, $d=0$

Como $a = b^d$ $(1 = 4^0)$, então $T(n) \in \Theta(n^0 \log_4 n) = \Theta(\log_4 n)$

3) Resolver $T(n) = 2T(n/2) + n.\log n, \quad T(1) = 1$.

$a = 2$, $b = 2$ e $f(n) = n.\log n \in \Theta(n^{\log_n(n.\log n)})$, $d = \log_n(n.\log n)$

$a < b^d$, então $T(n) \in \Theta(n^{\log_n(n.\log n)}) = \Theta(n.\log n)$.