

Introdução

Estruturas de Dados
Prof. Otávio Alcântara

Sumário

— — —

- Tipos de Dados
- Estrutura de Dados
- Tipos Abstratos de Dados
- Análise de Algoritmos
- Taxa de Crescimento
- Notação Assintótica
- Análise de Algoritmos Simples
- Recursividade
- Conclusão

Objetivos

— — —

- Entender os conceitos básicos de estruturas de dados e tipos abstratos de dados
- Compreender noções de análise de algoritmos
- Elencar taxas de crescimento comumente usadas
- Entender conceitos básicos de notação assintótica
- Aplicar notação assintótica em algoritmos simples
- Revisar conceitos de recursividade
- Encontrar relações de recorrência de funções simples
- Resolver problemas simples com recursividade

Tipos de Dados

— — —

- Conjunto de dados com valores pré-definidos
- Tipos primitivos
 - int, char, string
 - tamanho em bytes
 - intervalo
- Tipos definidos pelo usuário
 - estruturas
 - classes

Estruturas de Dados

— — —

- Formas de armazenar e organizar dados
 - Exemplos
 - Vetores, Arquivos, Listas, Pilhas, Filas, Árvores, Grafos, dentre outros
 - Linear
 - Listas encadeadas, pilhas e filas
 - Não lineares
 - Árvores e Grafos

Tipos Abstratos de Dados

— — —

- Especificação de uma estrutura de dados e um conjunto de operações sobre esta estrutura
 - Listas encadeadas, Filas, Filas de Prioridade, Árvores Binárias, Dicionários, Conjuntos Disjuntos, Tabelas de Hash, Grafos, dentre outros
- Exemplo
 - Pilha (LIFO)
 - criar pilha
 - inserir/remover elemento
 - determinar o topo da pilha
 - determinar o número de elementos

Análise de Algoritmos

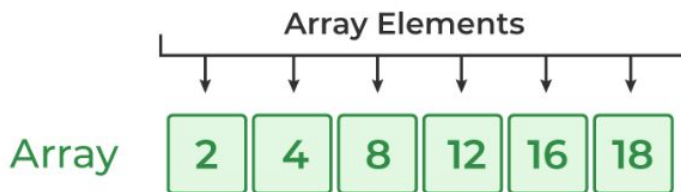
— — —

- Algoritmo
 - Sequência de instruções para solução de um problema
 - Corretude
 - Encontrar uma solução em um número finito de passos
 - Eficiência
 - Em termos de memória e tempo
- Análise de Algoritmos
 - Fornece ferramentas para determinar a eficiência de um algoritmo
 - Permite comparar diferentes algoritmos para o mesmo problema

Análise do Tempo de Execução

— — —

- Determinar como o tempo de execução varia de acordo com o tamanho do problema (entrada)
 - Variação da entrada
 - **Tamanho do vetor**
 - Grau de um polinômio
 - Número de elementos em uma matriz
 - Número de bits em uma imagem
 - Vértices e nós de um grafo



Análise do Tempo de Execução

— — —

- Determinar como o tempo de execução varia de acordo com o tamanho do problema (entrada)
 - Variação da entrada
 - Tamanho do vetor
 - **Grau de um polinômio**
 - Número de elementos em uma matriz
 - Número de bits em uma imagem
 - Vértices e nós de um grafo

$$y = 3x^4 - 4x^3 + 5x^2 - 6x + 7$$

Análise do Tempo de Execução

— — —

- Determinar como o tempo de execução varia de acordo com o tamanho do problema (entrada)
 - Variação da entrada
 - Tamanho do vetor
 - Grau de um polinômio
 - **Número de elementos em uma matriz**
 - Número de bits em uma imagem
 - Vértices e nós de um grafo

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

Análise do Tempo de Execução

— — —

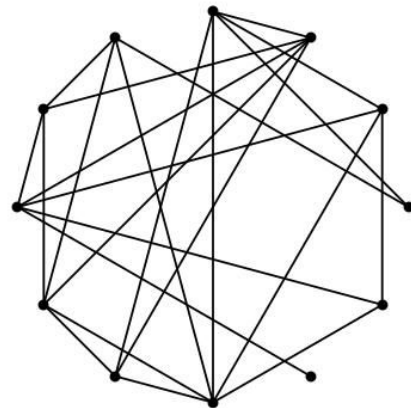
- Determinar como o tempo de execução varia de acordo com o tamanho do problema (entrada)
 - Variação da entrada
 - Tamanho do vetor
 - Grau de um polinômio
 - Número de elementos em uma matriz
 - **Número de bits em uma imagem**
 - Vértices e nós de um grafo



Análise do Tempo de Execução

— — —

- Determinar como o tempo de execução varia de acordo com o tamanho do problema (entrada)
 - Variação da entrada
 - Tamanho do vetor
 - Grau de um polinômio
 - Número de elementos em uma matriz
 - Número de bits em uma imagem
 - **Vértices e nós de um grafo**



Análise do Tempo de Execução

— — —

- Comparando algoritmos

Tempo de execução medido em um computador	Opção ruim!
Número de sentenças do programa	Opção ruim!
Expressar o tempo de execução como função do tamanho da entrada: $f(n)$	Boa!

Análise do Tempo de Execução

— — —

- Nós precisamos
 - de uma forma sistemática para descrever um algoritmo e das propriedades de suas estruturas de dados associadas
 - fazer isso de maneira independente do hardware
- Para tal, precisamos
 - Dos símbolos de Landau e de sua análise assintótica

Taxa de Crescimento

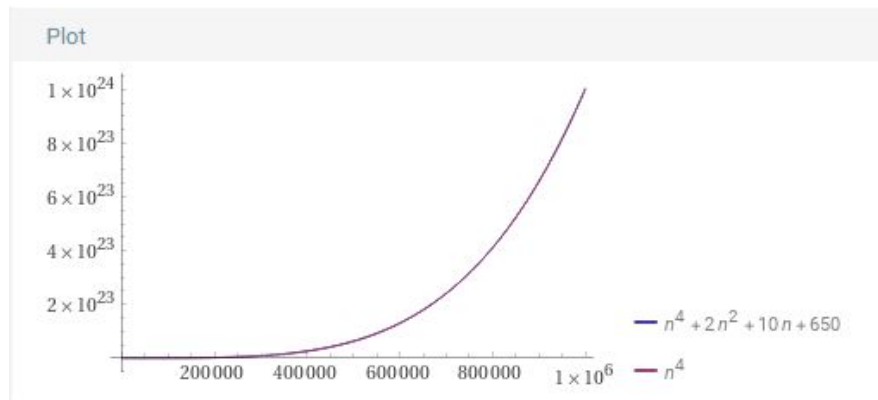
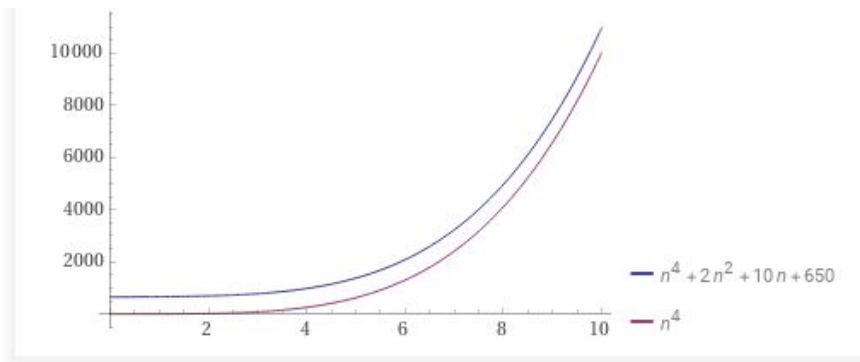
- É a taxa na qual o tempo de execução cresce em função da entrada
 - Exemplo
 - Após análise de um algoritmo, encontramos uma função que representa o tempo de execução
 - Para valores muito grandes de n , os termos de menor ordem influenciam pouco no valor de $T(n)$

$$T(n) = n^4 + 2n^2 + 10n + 650 \approx n^4, \text{ para } n \text{ muito grande}$$

Taxa de Crescimento

- É a taxa na qual o tempo de execução cresce em função da entrada
 - Exemplo

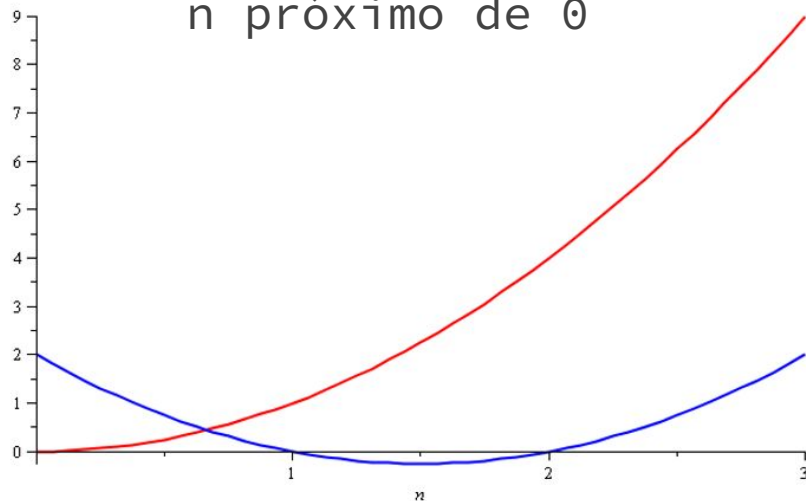
$$T(n) = n^4 + 2n^2 + 10n + 650 \approx n^4, \text{ para } n \text{ muito grande}$$



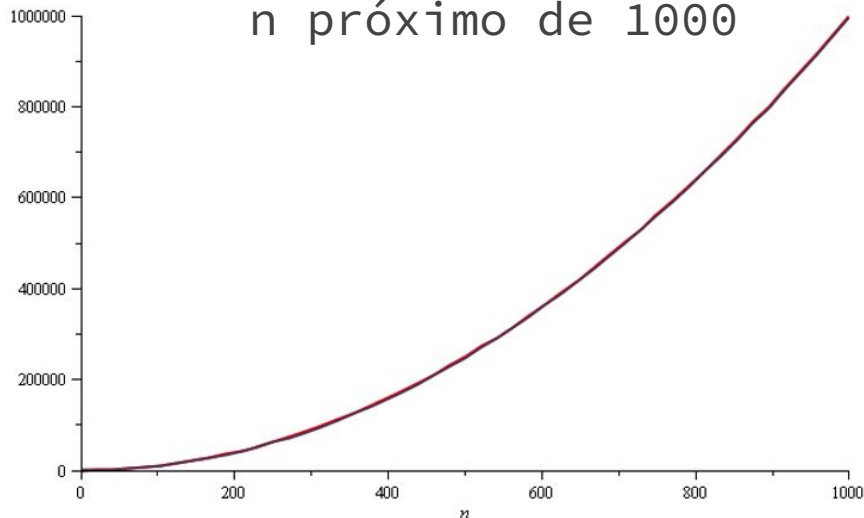
Crescimento Quadrático

- Considere as duas funções: $f(n) = n^2$ e $g(n) = n^2 - 3n + 2$

n próximo de 0



n próximo de 1000



Crescimento Quadrático

— — —

- A diferença absoluta entre as duas funções é grande em $n=1000$

$$\begin{aligned}f(1000) &= 1000000 \\g(1000) &= 997002\end{aligned}$$

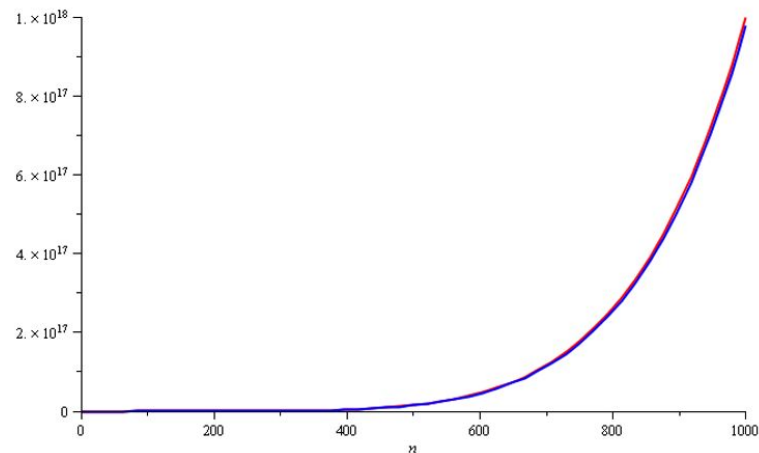
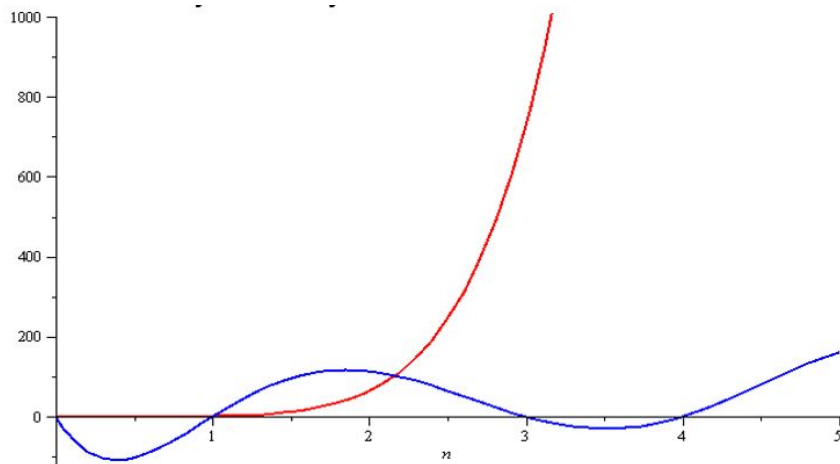
- A diferença relativa é pequena

$$\left| \frac{f(1000) - g(1000)}{f(1000)} \right| < 3\%$$

Se os coeficientes dos termos quadráticos fossem diferentes, as funções teriam a mesma taxa de crescimento, mas uma seria proporcionalmente maior!

Crescimento Polinomial

- Um outro exemplo: $f(n) = n^6$ e $g(n) = n^6 - 25n^5 + 193n^4 - 729n^3 + 120n^2 + 648n$



Diferença relativa para $n = 1000$ é menor que 3%, tende a 0 conforme n aumenta

Taxa de Crescimento

- Taxas de crescimento comumente usadas

$$2^{2^n} \rightarrow n! \rightarrow 4^n \rightarrow 2^n \rightarrow n^2 \rightarrow n \log n \rightarrow \log(n!) \rightarrow n \rightarrow 2^{\log n} \rightarrow \log^2 n \rightarrow \sqrt{\log n} \rightarrow \log \log n \rightarrow 1$$

Taxa de Crescimento

— — —

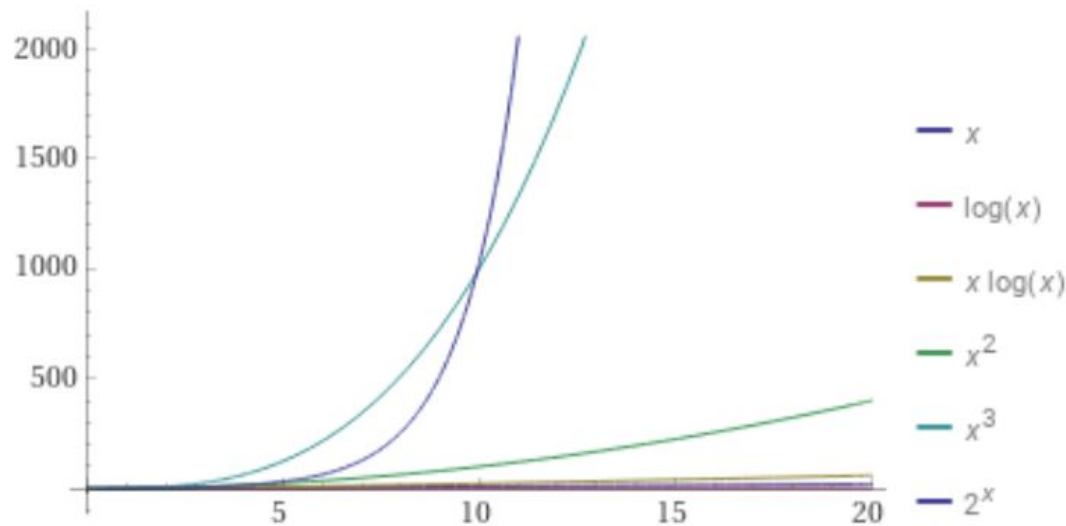
- Taxas de crescimento comumente usadas

Complexidade	Nome	Exemplo
1	Constante	Inserir elemento na frente de uma lista
$\log n$	Logarítmica	Encontrar elemento em vetor ordenado
n	Linear	Encontrar elemento em vetor desordenado
$n \log n$	Linear Logarítmica	Ordenar n elementos usando Mergesort
n^2	Quadrática	Menor caminho entre dois nós em um grafo
n^3	Cúbica	Multiplicação de matrizes
2^n	Exponencial	Torres de Hanoi

Taxa de Crescimento

- Taxas de crescimento comumente usadas

Complexidade	Nome
1	Constante
$\log n$	Logarítmica
n	Linear
$n \log n$	Linear Logarítmica
n^2	Quadrática
n^3	Cúbica
2^n	Exponencial

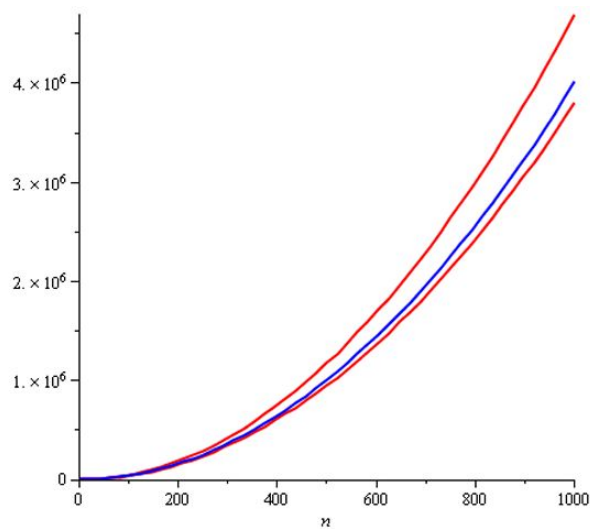
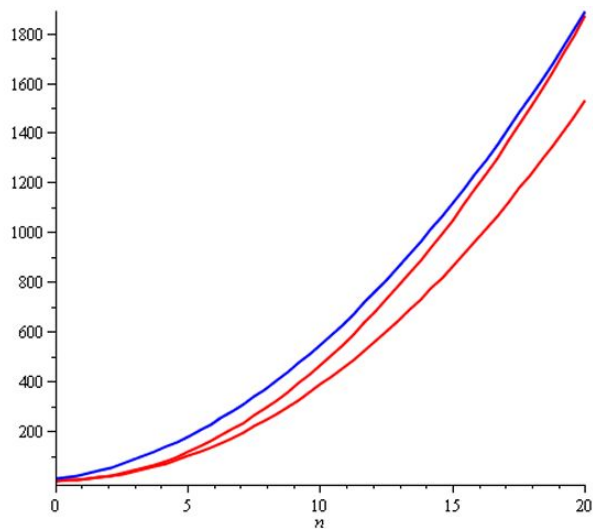


Exemplo de Análise

Selection Sort $s(n) = 4n^2 + 8n + 6$

Bubble Sort melhor caso $b_{best} = 4,7n^2 + 0,5n + 5$

pior caso $b_{worst} = 3,8n^2 - 0,5n + 5$



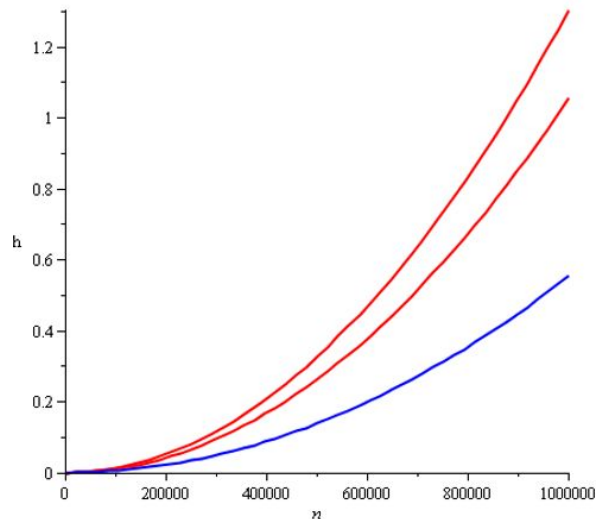
Executar o
Selection Sort em
um computador
mais rápido faria
que $s(n) <$
 $b(n)$???

Exemplo de Análise

Selection Sort $s(n) = 4n^2 + 8n + 6$

Bubble Sort melhor caso $b_{best} = 4,7n^2 + 0,5n + 5$

pior caso $b_{worst} = 3,8n^2 - 0,5n + 5$



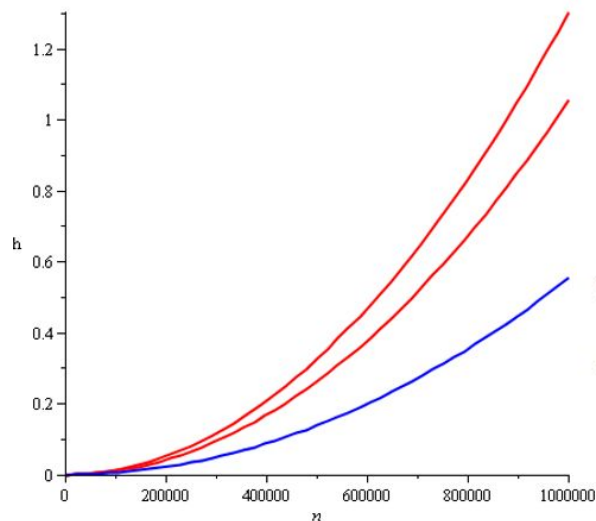
Para valores grandes de n , em um computador mais rápido, Selection Sort sempre será mais rápido do que o Bubble sort.

Exemplo de Análise

Selection Sort $s(n) = 4n^2 + 8n + 6$

Bubble Sort melhor caso $b_{best} = 4,7n^2 + 0,5n + 5$

pior caso $b_{worst} = 3,8n^2 - 0,5n + 5$



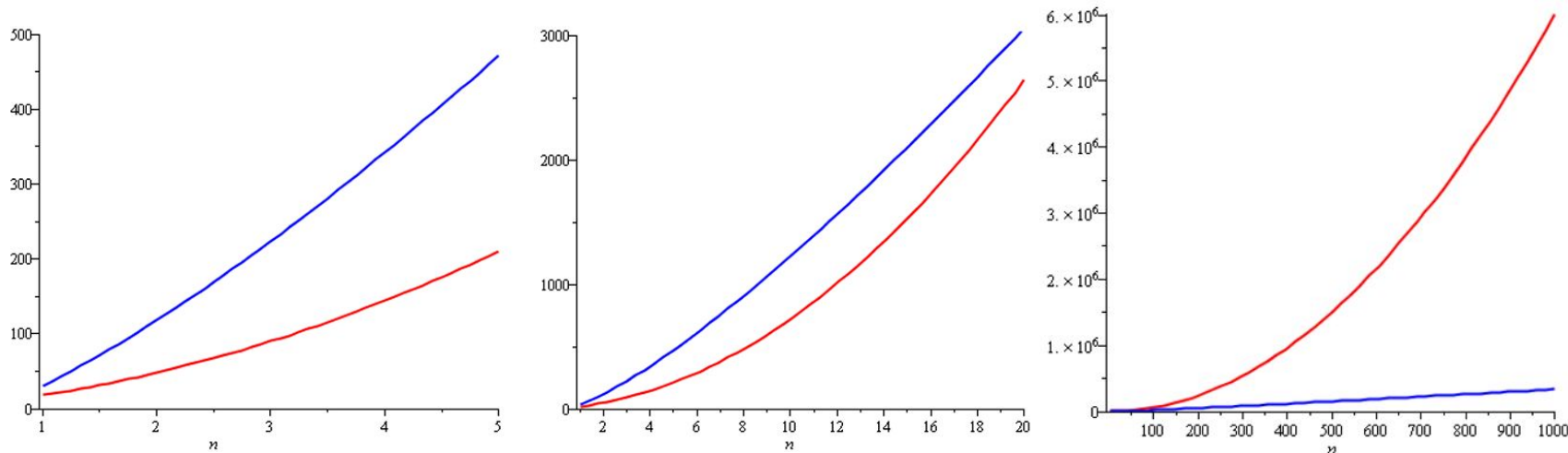
Justificativa?

- Basta encontrar M tal que $s(n) < M b(n)$

se $f(n) = a_k n^k + \dots$ e $g(n) = b_k n^k + \dots$, para n muito grande,
 $f(n) < M g(n)$, $M = a_k / b_k + 1$

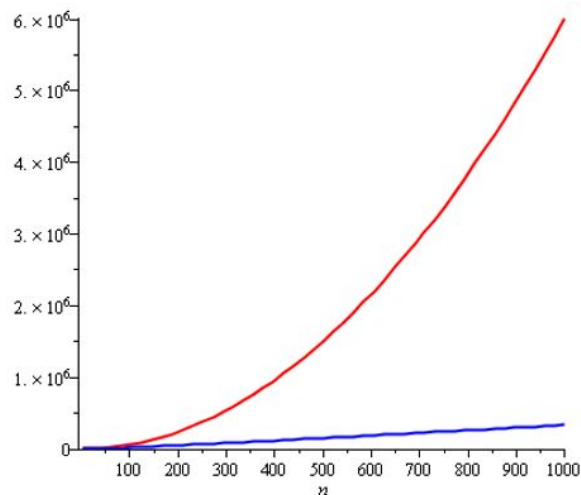
Exemplo de Análise

- Insertion Sort (vermelho) e Quicksort (azul)



Exemplo de Análise

- Insertion Sort (vermelho) e Quicksort (azul)



Comprar um computador mais rápido faria o Insertion Sort melhor que o Quicksort para n muito grande?

Ordenação Fraca

- Considere as seguintes definições

$$f \sim g \text{ se } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, 0 < c < \infty$$

$$f < g \text{ se } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

- Se $f \sim g$, então é possível melhorar a performance comprando um computador melhor
- Se $f < g$, então **não** é possível melhorar a performance comprando um computador melhor

Tipos de Análise

— — —

- **Pior caso**
 - A entrada que o algoritmo executa mais lento
- **Melhor caso**
 - A entrada que o algoritmo executa mais rápido
- **Caso médio**
 - Executa diversas vezes com diferentes entradas aleatórias
 - Calcula a média do tempo de execução

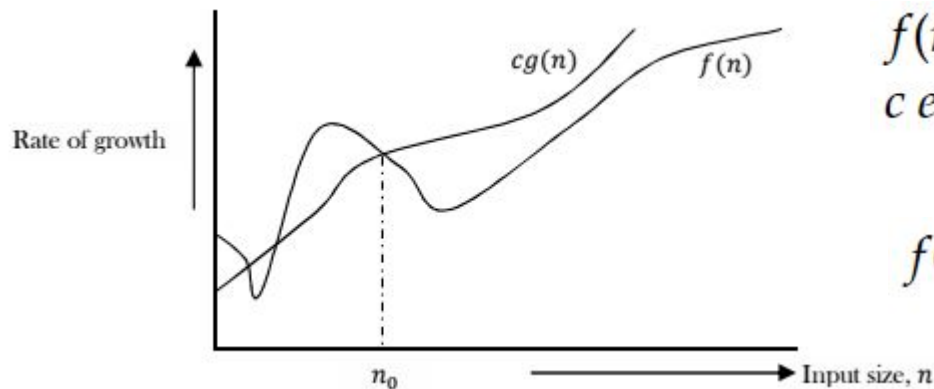
Notação Assintótica

— — —

- Tendo as funções do tempo de execução
 - Melhor, pior e caso médio
- É necessário analisar seu comportamento para valores muito grandes da entrada
 - Comportamento assintótico
- Determinar os limites do tempo de execução
 - Diferentes entradas
 - Pior, melhor e caso médio

Notação Big-O

- Big-O determina um limite superior “apertado”
 - Nós queremos encontrar a menor taxa de crescimento tal que $g(n) \geq f(n)$



$f(n) = O(g(n)) \implies f(n) \leq cg(n), \forall n \geq n_0,$
 c e n_0 são constantes

$$f(n) = n^4 + 2n^2 + 10n + 650 = O(n^4)$$

A taxa de crescimento de $f(n)$ não é maior do que $g(n)$

Notação Big-O

- Visualização do Big-O

$O(1)$: 100, 1000, 200, 1, 20, etc.

$O(n \log n)$: $5n \log n$, $3n - 100$, $2n - 1$, 100, $100n$, etc.

$O(n)$: $3n + 100$, $100n$, $2n - 1$, 3, etc.

$O(n^2)$: n^2 , $5n - 10$, 100, $n^2 - 2n + 1$, 5, etc.

Notação Big-O

- Exemplos

Encontre o limite superior para as funções abaixo :

i) $f_0(n) = 410$

ii) $f_1(n) = n$

iii) $f_2(n) = n^2 + 100n + 50$

iv) $f_3(n) = 2n^3 - 2n^2$

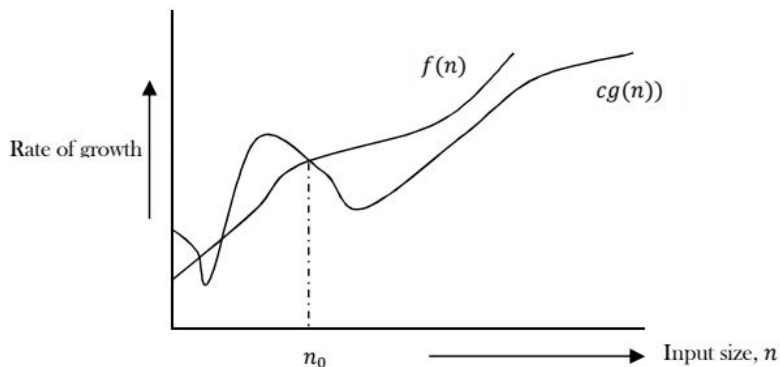
$f(n) = n^4 + 100n^2 + 50$

Solução : $n^4 + 100n^2 + 50 \leq 2n^4, \forall n \geq 50$

$n^4 + 100n^2 + 50 = O(n^4)$ com $c = 2$ e $n_0 = 50$

Notação Ômega

- Determina o limite inferior “apertado”



$$f(n) = \Omega(g(n)) \implies 0 \leq cg(n) \leq f(n), \forall n \geq n_0,$$

c e n_0 são constantes

Exemplo: Se $f(n) = 100n^3 + 100n^2 + 50$, $g(n)$ é $\Omega(n^3)$

Notação Ômega

- Exemplos

Encontre o limite inferior para as funções abaixo :

i) $f_0(n) = 5n^2$

ii) $f_1(n) = 100n + 5$

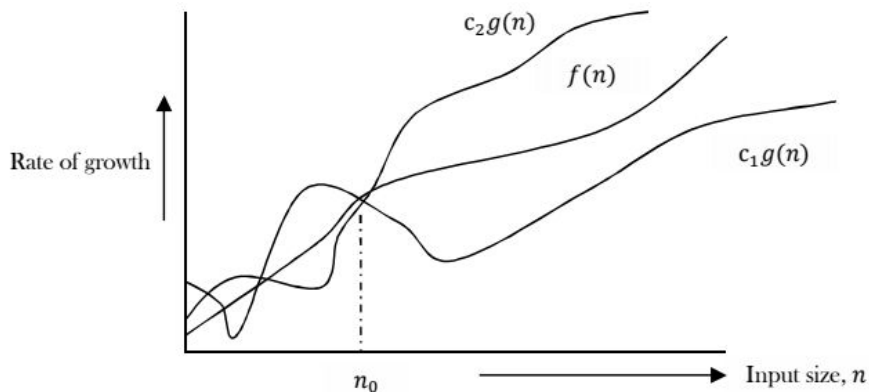
iii) $f_2(n) = 2n$

iv) $f_3(n) = n^3$

v) $f_4(n) = \log n$

Notação Teta

- Determina se o limite inferior e o limite superior de um dado algoritmo é o mesmo



$$f(n) = \Theta(g(n)) \implies c_1g(n) \leq f(n) \leq c_2g(n), \forall n \geq n_0,$$

c_1, c_2 e n_0 são constantes

$$f(n) = \frac{n^2}{2} - \frac{n}{2} = \Theta(g(n))$$

$$\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2, \text{ para todo } n \geq 2$$

$$\frac{n^2}{2} - \frac{n}{2} = \Theta(n^2) \text{ com } c_1 = \frac{1}{5}, c_2 = 1, n_0 = 2$$

Símbolos de Landau

— — —

Símbolo de Landau	Limite	Descrição	Operador Relacional Análogo
$f(n) = \omega(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$	f cresce muito mais rápido do que g	$>$
$f(n) = \Omega(g(n))$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < c$	f cresce na mesma taxa ou maior que g	\geq
$f(n) = \theta(g(n))$	$0 < \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$	f cresce na mesma taxa g	$=$
$f(n) = O(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$	f cresce na mesma taxa ou maior que g	\leq
$f(n) = o(g(n))$	$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$	f cresce bem menos rápido que g	$<$

f e g são combinações lineares de r^n e $r^n \ln(n)$, $n \in \mathbb{R}$, $n > 0$.

Orientações para Notação Assintótica

— — —

- A notação pode ser usada para o pior, melhor ou caso médio
- Normalmente, estamos interessados em encontrar o limite superior do pior caso usando Big-O
- Nas situações nas quais os limites inferior e superior forem iguais, podemos usar a notação Teta

Orientações para análise assintótica

— — —

- Sentenças simples
 - Executam em $\Theta(1)$

```
arr = ['manga', 'maçã', 'graviola']  
n=100  
i = n + 2  
print(arr[0])
```

Orientações para análise assintótica

— — —

- Laços
 - O tempo de execução é o produto do número de iterações pelo tempo de execução das instruções do laço

```
#execute n times
for i in range(0,n):
    print("Hello %d"%i)
```

$$T(n) = cn = \Theta(n), \text{ } c \text{ é uma constante}$$

Orientações para análise assintótica

- Laços aninhados
 - Tempo de execução é o total do produto dos tempos dos laços

```
#laço externo executa n vezes
for i in range(0,n):
    for j in range(0,n):
        print("Valor de i: %d, valor de j:%d"%(i,j))
```

$$T(n) = c * n * n = cn^2 = \Theta(n^2), c \text{ é uma constante.}$$

Orientações para análise assintótica

— — —

- Laços logarítmicos
 - Considere que $n=2^m$, Nesse caso i seria $2^m, 2^{m-1}, 2^{m-2}, \dots, 2, 1, 0$
 - Tempo de execução logarítmico
 - Dobre a entrada do problema, o tempo de execução aumenta por uma constante

```
i = n
while (i>0):
    i/=2
    print("Valor de i %d"%i)
```

$m = \lg(n)$, portanto $\Theta(\lg(n))$

Lembre – se que os logaritmos são múltiplos escalares uns dos outros, logo $\lg(n) = \Theta(\ln(n))$.

Orientações para análise assintótica

- Laços logarítmicos
 - Um algoritmo é $O(\log n)$ quando leva um tempo constante para dividir um problema por uma fração (normalmente 2)

```
def func(n):  
    i=1  
    while i <= n:  
        i = i*2  
        print(i)  
func(40)
```

$$i = 2, 2^2, 2^3, 2^4, \dots, 2^k$$

São realizadas k iterações

Na última iteração, $n \geq 2^k$

Aplicando log na base 2

$$\log n \geq k$$

$$T(n) = O(\log n)$$

Orientações para análise assintótica

- Laços iterativos
 - Determinar se um valor está armazenado em um arranjo de inteiros ordenado
 - O laço pode executar em $\Theta(1)$ no melhor caso, como pode executar n vezes no pior
 - Como o laço não executa sempre n vezes, dizemos que é $O(n)$
 - Pode rodar em n , mas se tivermos sorte roda em menos

```
def linear_search(value, arr, n):  
    for i in range(0,n):  
        if arr[i]==value:  
            return True
```

Orientações para análise assintótica

— — —

- $\Theta(n) \times O(n)$

```
def find_max(arr):  
    max = arr[0]  
    for e in arr:  
        if e > max:  
            max = e  
    return max
```

```
def linear_search(value, arr, n):  
    for i in range(0,n):  
        if arr[i]==value:  
            return True
```

Orientações para análise assintótica

- Laços dependentes de variáveis
 - O laço interno executa $\Theta(i)$, mas i muda a cada iteração!
 - Logo, temos que calcular quantas operações são feitas

```
for i in range(0,n):  
    for j in range(0,i):  
        print("Valor de i%d e valor de j %d"%(i,j))
```

Valor de I	1	2	3	..	n	Total
Número de Operações	1	2	3	..	n	1+2+..+n

$$\Theta\left(\sum_{i=1}^n i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Orientações para análise assintótica

- Sentenças consecutivas
 - Tempo de execução é a soma de cada sentença

```
#laço executa n vezes
for k in range(0,n):
    print("H %d"%k)

#laço externo executa n vezes
for i in range(0,n):
    for j in range(0,n):#laço interno executa n vezes
        print("Valor de i: %d, valor de j:%d"%(i,j))
```

$$T(n) = c_0n + c_1n^2 = \Theta(n^2)$$

Orientações para análise assintótica

- Sentenças de controle
 - Tempo de execução é o tempo do teste mais o tempo da condição mais lenta (análise de pior caso)

```
if n == 1:  
    print("Valor errado")  
else:  
    for i in range(0,n):#laço externo executa n vezes  
        for j in range(0,n):#laço interno executa n vezes  
            print("Valor de i: %d, valor de j:%d"%(i,j))
```

$$T(n) = c_0 + c_1n^2 = \Theta(n^2)$$

Orientações para análise assintótica

— — —

- Sentenças de controle

- Suponha que a condição do if tem 50% de chances de ser verdadeira
 - Metade das vezes o segundo laço executa
 - Metade das vezes o segundo laço não executa

```
for i in range(0,n):  
    if (condicao_logica):  
        for j in range(0,i):  
            print("Valor de i %d e valor de j %d"%(i,j))
```

$$\Theta\left(\frac{n}{2}n + \frac{n}{2} \cdot 1\right) = \Theta\left(\frac{n^2}{2} + \frac{n}{2}\right) = \Theta(n^2)$$

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    i=s=1  
    while s <= n:  
        print(i)  
        print(s)  
        i = i + 1  
        s = s + i  
  
func(40)
```

- Encontre a taxa de crescimento de i e s
- Determine a relação entre o número de iterações e n

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    i=s=1  
    while s <= n:  
        print(i)  
        print(s)  
        i = i + 1  
        s = s + i
```

```
func(40)
```

$$i = 1, 2, 3, 4, 5, 6, 7, \dots, k$$

$$s = 1, 3, 6, 10, 15, 21, 28, \dots, \frac{k(k+1)}{2}$$

s_j é a soma dos primeiros j inteiros

$$\frac{k^2}{2} + \frac{k}{2} > n, k \approx \sqrt{n} \implies k = O(\sqrt{n})$$

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    i=1  
    count = 0  
    while i*i <= n:  
        print(i)  
        print(count)  
        i = i + 1  
        count = count + 1  
  
func(40)
```

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    i=1  
    count = 0  
    while i*i <= n:  
        print(i)  
        print(count)  
        i = i + 1  
        count = count + 1
```

```
func(40)
```

$$\begin{aligned} i &= 1, 2, 3, 4, 5, 6, 7, \dots, k \\ s &= 1, 4, 9, 16, 25, 36, 49, \dots, k^2 \\ k^2 > n &\implies T(n) = \sqrt{n} \end{aligned}$$

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    count=0  
    for i in range(n/2,n):  
        j=1  
        while j+n/2<=n:  
            k=1  
            while k<=n:  
                count=count+1  
                k=k*2  
            j=j+1  
    print(count)
```

- Determine o número de iterações de cada laço em relação a n

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    count=0  
    for i in range(n/2,n):  
        j=1  
        while j+n/2<=n:  
            k=1  
            while k<=n:  
                count=count+1  
                k=k*2  
            j=j+1  
        print(count)
```

Primeiro laço faz $n/2$ iterações

Segundo laço faz $n/2$ iterações

Terceiro laço

$k = 1, 2, 4, 8, \dots, 2^M$

$n \geq 2^M$, logo o número de iterações é $\log n$
na base 2

Então, $T(n) = \frac{n}{2} * \frac{n}{2} * \log n = O(n^2 \log n)$

Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    count=0  
    for i in range(n/2,n):  
        j=1  
        while j+n/2<=n:  
            break  
        j=j*2  
        print(count)  
  
func(40)
```


Problemas e Soluções

- Qual é o tempo de execução da função abaixo?

```
def func(n):  
    count=0  
    for i in range(n/2,n):  
        j=1  
        while j+n/2<=n:  
            break  
        j=j*2  
        print(count)
```

func(40)

- O laço exterior executa $n/2$
- O laço interior possui um break e só executa uma vez
- $O(n)$

Recursividade

— — —

- O que é recursividade?
- Exemplo de algoritmos recursivos

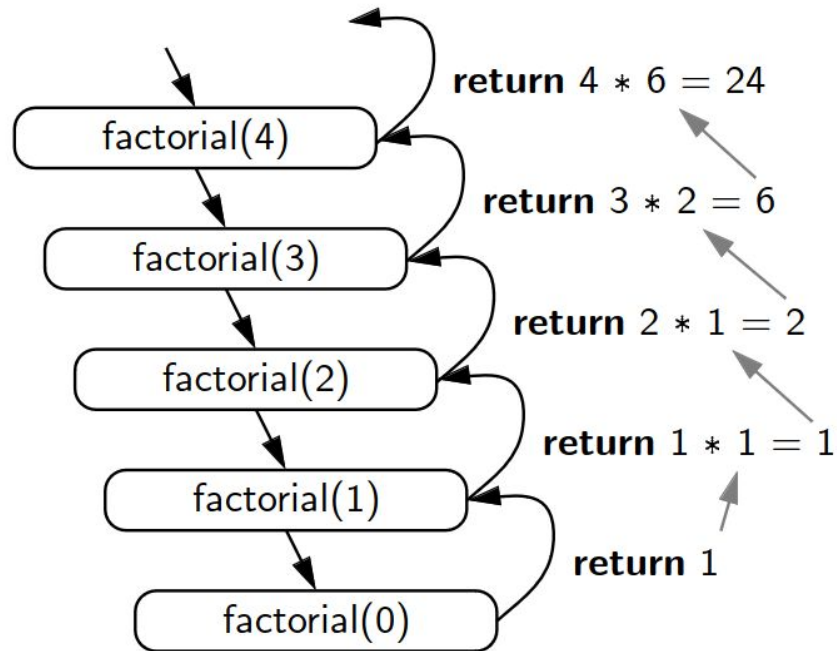
```
def fatorial(n):  
    if n==0: return 1  
    return n*fatorial(n-1)  
  
print(fatorial(10))
```

Recursividade	Iteratividade
Termina no caso base	Termina na condição falsa
Usa mais pilha	Não requer memória extra
Pode ocasionar estouro de pilha se mal escrita	Laço infinito se mal escrita
Mais fácil de resolver alguns problemas	Nem sempre tem uma solução óbvia

Recursividade

- É uma forma elegante de realizar tarefas repetitivas

```
def fatorial(n):  
    if n==0: return 1  
    return n*fatorial(n-1)  
  
print(fatorial(10))
```



Recursividade

— — —

- Tempo de Execução

```
def fatorial(n):  
    if n==0: return 1  
    return n*fatorial(n-1)  
  
print(fatorial(10))
```

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + \Theta(1) & n > 1 \end{cases}$$

A função recursiva reduz o trabalho em 1 e faz um trabalho constante

Recursividade

— — —

- Tempo de Execução

```
def fatorial(n):  
    if n==0: return 1  
    return n*fatorial(n-1)  
  
print(fatorial(10))
```

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + \Theta(1) & n > 1 \end{cases}$$

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

$$T(n-1) = T(n-2) + \Theta(1)$$

.

.

.

$$T(1) = T(0) + \Theta(1)$$

Somando todas as parcelas e cancelando os termos, temos:

$$T(n) = \Theta(1) + \dots + \Theta(1) = \Theta(n)$$

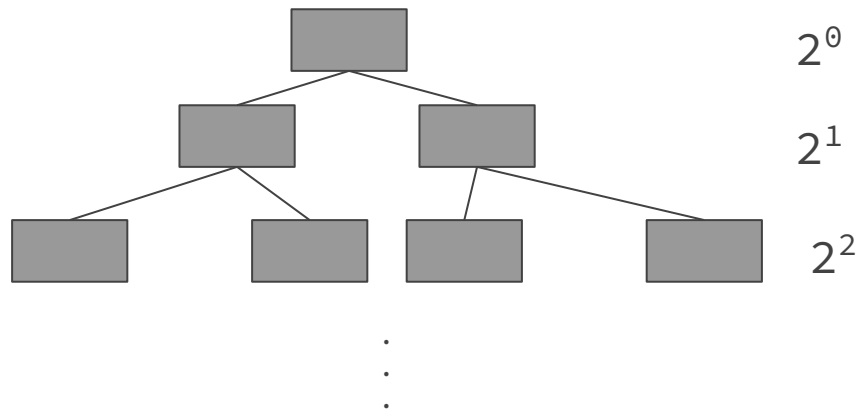
Recursividade

- Tempo de Execução

```
def fib(n):  
    if n <= 1:  
        return n  
    else:  
        return fib(n-1)+fib(n-2)
```

A cada chamada recursiva o problema é dividido em dois subproblemas que não se sobrepõem

$$T(n) = \begin{cases} \Theta(1) & n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) & n > 1 \end{cases}$$



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

$$T(n) = \Theta(2^n)$$

Recursividade


- Tempo de Execução

```
def selection_sort(arr,n,i):  
    if n < 1: return  
    min = i  
    for k in range(i+1,len(arr)):  
        if arr[k] < arr[min]:  
            min = k  
    arr[i], arr[min] = arr[min], arr[i]  
    selection_sort(arr,n-1,i+1)
```

Em cada chamada a função faz uma busca linear $n-1$ pelo menor valor

$$\begin{aligned}T(n) &= T(n) + T(n-1) + T(n-2) + \dots + 1 \\T(n) &= \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \\T(n) &= \Theta(n^2)\end{aligned}$$

Problemas e Soluções

- Escreva um programa para resolver o problema das Torres de Hanoi
- 
- Mover os discos da torre 1 para torre 3
 - Apenas um disco pode ser movido por vez
 - Nenhum disco pode ser posto sobre um disco menor
 - Mova $n-1$ discos da torre fonte para torre auxiliar
 - Mova o n -ésimo disco da fonte para o destino
 - Mova $n-1$ discos da torre auxiliar para torre destino

Problemas e Soluções

- Escreva um programa para resolver o problema das Torres de Hanoi

```
def towersOfHanoi( numberOfDisks, startPeg=1, endPeg=3):  
    if numberOfDisks :  
        towersOfHanoi( numberOfDisks-1, startPeg, 6-startPeg-endPeg)  
        print("Mova disco %d da torre %d para torre %d" %(numberOfDisks, startPeg, endPeg))  
        towersOfHanoi( numberOfDisks-1, 6-startPeg-endPeg, endPeg )  
  
towersOfHanoi(numberOfDisks=4)
```

Problemas e Soluções

— — —

- Imprima todas as strings binárias com n bits. Considere $A[0\dots n-1]$ é um vetor de tamanho n .
 - Exemplo
 - $n=2$
 - 00,01,10,11
 - $n=3$
 - 000,001,010,011,100,101,110,111

Problemas e Soluções

— — —

- Imprima todas as strings binárias com n bits. Considere $A[0..n-1]$ é um vetor de tamanho n .
 - Algoritmo
 - if $n == 0$: return []
 - if $n == 1$: return ["0", "1"]
 - else: return insereBit("0", bitstring($n-1$))+insereBit("1", bitstring($n-1$))

Problemas e Soluções

- Imprima todas as strings binárias com n bits. Considere $A[0\dots n-1]$ é um vetor de tamanho n .

```
def appendAtFront(x,L):  
    return [x + element for element in L]  
  
def bitStrings(n):  
    if n == 0 : return []  
    if n == 1 : return ["0", "1"]  
    else:  
        return (appendAtFront("0",bitStrings(n-1))+appendAtFront("1",bitStrings(n-1)))  
  
print(bitStrings(2))
```

Problemas e Soluções

— — —

- Encontre a relação de recorrência da função abaixo

```
def f(n):  
    count = 0  
    if n <= 0:  
        return  
    for i in range(0,n):  
        for j in range(0,n):  
            count = count + 1  
    f(n-3)  
    print(count)
```

Problemas e Soluções

- Encontre a relação de recorrência da função abaixo

```
def f(n):  
    count = 0  
    if n <= 0:  
        return  
    for i in range(0,n):  
        for j in range(0,n):  
            count = count + 1  
    f(n-3)  
    print(count)
```

- 1° laço
 - n vezes
- 2° laço
 - n vezes
- Chamada recursiva

$$T(n) = cn^2 + T(n-3)$$

Problemas e Soluções

— — —

- Encontre a relação de recorrência do programa abaixo

```
def f(n):  
    if n==0: return 0  
    elif n==1: return 1  
    else: return f(n-1) + f(n-2)  
  
print(f(3))
```

Problemas e Soluções

- Encontre a relação de recorrência do programa abaixo

```
def f(n):  
    if n==0: return 0  
    elif n==1: return 1  
    else: return f(n-1) + f(n-2)  
  
print(f(3))
```

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

- O código faz duas chamadas recursivas
- Desenhe a árvore de recursão para essa função
 - Determine o número de níveis
 - Determine o número de folhas por nível
 - Qual é a complexidade?

Problemas e Soluções

Escreva as funções recursivas listadas a seguir, todas recebem um inteiro e devolvem um inteiro

- maiorDigito
 - retorna o maior dígito do inteiro
- menorDigito
 - retorna o menor dígito do inteiro
- contaDigito
 - retorna a quantidade de dígitos do inteiro
- somaDigito
 - retorna a soma dos dígitos do inteiro

Problemas e Soluções

— — —

Escreva as funções recursivas listadas a seguir, todas recebem um inteiro e devolvem um inteiro

- `zeraPares`
 - retorna um inteiro com os dígitos pares em zero
- `zeraImpares`
 - retorna um inteiro com os dígitos ímpares em zero
- `removePares`
 - remove os dígitos pares do inteiro
- `removeImpares`
 - remove os dígitos ímpares do inteiro

Problemas e Soluções

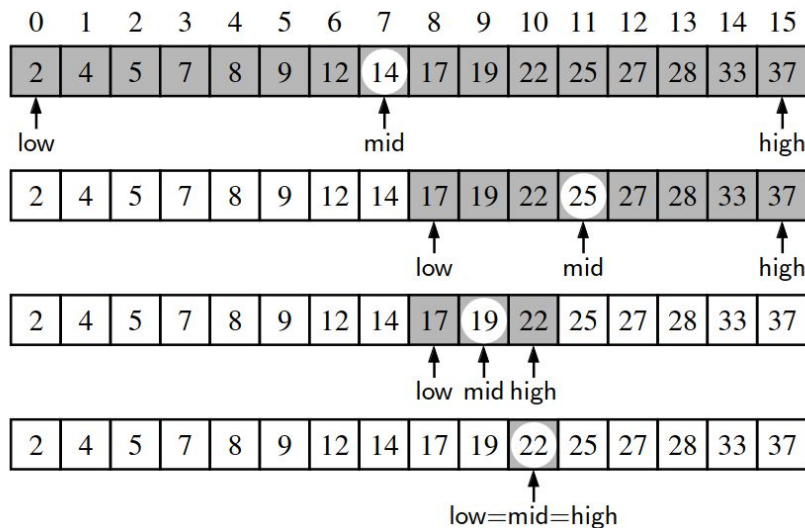
Escreva uma função recursiva que inverte os dígitos de um número inteiro

- `inverte(n, inverso)`

Problemas e Soluções

Busca binária

- Localizar o elemento 22 em uma lista ordenada



Problemas e Soluções

Busca binária

- Localizar um elemento em uma lista ordenada
- Para qualquer índice j
 - todos os valores armazenados $< j$ tem valores $\leq \text{data}[j]$
 - todos os valores armazenados $> j$ tem valores $\geq \text{data}[j]$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Problemas e Soluções

Busca binária

- Localizar um elemento em uma lista ordenada

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

`binary_search(data, target, low, high)`

- `mid = (low + high)/2`
- `if target < data[mid]`
 - `binary_search(data, target, low, mid-1)`
- `else`
 - `binary_search(data, target, mid+1, high)`

Exercícios de Recursividade

— — —

- Crie uma função recursiva que determina se uma string `s` tem mais vogais do que consoantes
- Dado um array `S` não ordenado de inteiros e um inteiro `k`, crie um algoritmo recursivo para reorganizar os elementos de `S` tal que todos os elementos menores ou iguais a `K` apareçam antes do que os elementos maiores.

Exercícios de Recursividade

— — —

- Dados um vetor de inteiros distintos e ordenados de maneira crescente e um inteiro target, crie um algoritmo recursivo que determine se existem dois inteiros no vetor que a soma seja igual a target.

Recursividade

— — —

- Teorema Mestre

- Ferramenta para análise de recorrências, que surgem em algoritmos de divisão e conquista

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

a – número de subproblemas

b – fator de redução do problema

f(n) – custo de dividir e recombinar as soluções dos subproblemas

Recursividade

— — —

- Teorema Mestre

- Ferramenta para análise de recorrências, que surgem em algoritmos de divisão e conquista

Casos do Teorema Mestre

Caso 1: se $f(n) = O(n^c)$ onde $c < \log_b a$, então $T(n) = O(n^{\log_b a})$.

Caso 2: se $f(n) = \Theta(n^{\log_b a} \log^k n)$, então $T(n) = O(n^{\log_b a} \log^{k+1} n)$.

Caso 3: se $f(n) = \Omega(n^c)$ onde $c > \log_b a$ e $af\left(\frac{n}{b}\right) \leq kf(n)$ para $k < 1$ e n suficientemente grande, então $T(n) = O(f(n))$.