Algoritmos e Estruturas de Dados
Aula 2 :: Complexidade

Filipe Cordeiro (filipe.rolim@ufrpe.br)

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Crédito slides: Prof. Francisco Simões (francisco.simoes@ufrpe.br)

O que é um algoritmo?

Sequência de passos para realizar uma tarefa.

Exemplos de classes de algoritmos básicos:

Ordenação, Busca, Remoção...

Definição de Algoritmo

- Um algoritmo é dito **correto** se, para cada instância de entrada existente, o mesmo encerra sua execução com um valor de saída correto.

- O que mais o algoritmo precisa ser?

Uso de Algoritmos

- Em computação, os algoritmos são utilizados para solucionar os mais diversos tipos de problemas práticos, em áreas como:
  - Biologia;
  - Medicina;
  - Artes;
  - Ciências Sociais;
  - ...

Estruturas de Dados

São estruturas utilizadas para guardar/organizar dados.

A estrutura certa pode facilitar a execução de algoritmos.

Exemplos:

Vetor, Lista, Pilha, Fila, Árvores, Tabelas Hash...

Análise de Complexidade

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Complexidade

- Como comparar eficiência de algoritmos?

Como saber se o algoritmo A é **mais rápido que** o algoritmo B (considerando sua execução numa mesma máquina)?

Complexidade

- Como comparar eficiência de algoritmos?

Como saber se o algoritmo A é **mais rápido que** o algoritmo B (considerando sua execução numa mesma máquina)?

- Como estimar **o quão mais lento o algoritmo executa quando aumentamos** a quantidade de dados?

Ordenar 10 números *vs* ordenar 10.000.000?

Complexidade

- Como comparar eficiência de algoritmos?

Como saber se o algoritmo A é **mais rápido que** o algoritmo B (considerando sua execução numa mesma máquina)?

- Como estimar **o quão mais lento o algoritmo executa quando aumentamos** a quantidade de dados?

Ordenar 10 números *vs* ordenar 10.000.000?

Caso médio? **Pior caso?** Melhor caso?
É super importante a análise do pior caso! Por quê?

O que é eficiência em computação?

https://www.youtube.com/watch?v=iZuvuJq-58w

Crédito: Vídeo do prof. Rodrigo Nonamor do DC-UFRPE.

Como medir a complexidade de um código (empiricamente)?

Vamos construir a versão iterativa e a versão recursiva de um fibonacci.

Vamos começar a aprender como olhar para o código e pensar sobre complexidade.

Para ler a respeito de Fibonacci e sua análise:

Capítulo 0, Seção 0.1 e 0.2 deUmesh, DASGUPTA, Sanjoy ; PAPADIMITRIOU, Christos ; V. **Algoritmos**. Grupo A, 2011.

Link direto para o livro: https://integrada.minhabiblioteca.com.br/books/9788563308535

Procurar no portal Minha Biblioteca da biblioteca UFRPE. Se quiser seguir o link, tem que estar com login ativo no Minha Biblioteca.

Atividades

1. Implementar o algoritmo de Fibonacci de duas formas:
   a. recursiva
   b. iterativa
2. Adicione ao seu código uma variável global chamada NOP para calcular a quantidade de operações do Fibonacci, seja iterativo ou recursivo.
3. Realize a chamada da função para calcular o Fibonacci de alguns números como no exemplo sobre complexidade do prof. Rodrigo.
4. Compare e discuta a quantidade de operações da chamada recursiva e iterativa.

Gere um gráfico comparando os NOPS dos dois modelos.

Análise de Complexidade de Algoritmos

- Tenta prever os recursos requeridos por um algoritmo durante sua execução;
- Ex: Memória, alocação de banda para comunicação, dispositivo de hardware, tempo de processamento.

Análise de Complexidade

- Por que é importante analisar a complexidade de algoritmos?
- A complexidade é um dos principais fatores durante o projeto de novos algoritmos.
    - Outros fatores: corretude, eficiência, tolerância à falhas, etc.

Análise de Complexidade

- Por que é importante analisar a complexidade de algoritmos?
- A complexidade é um dos principais fatores durante o projeto de novos algoritmos.
    - Outros fatores: corretude, eficiência, tolerância à falhas, etc.
- A análise de complexidade fornece uma medida para que se possa decidir qual técnica é a mais adequada para o problema em mãos.

Análise de algoritmos

- Como analisar a eficiência de um algoritmo?
- Possível solução: estimar o tempo experimentalmente...
  - Problema: nem sempre isso é possível/conveniente
- É esperado que ocorram variações
  - Em função do hardware da máquina e uso do sistema operacional, principalmente
  - Tempo pode variar muito
    - Configurações da máquina
    - Uso do sistema operacional
    - Tamanho dos dados de entrada

Análise de algoritmos

- Tempo de execução depende de
  1. Custo de execução de cada instrução
     - Dependente da arquitetura
  2. Frequência de execução de cada instrução
     - Dependente do algoritmo
     - Foco se dá nos trechos de código mais frequentes

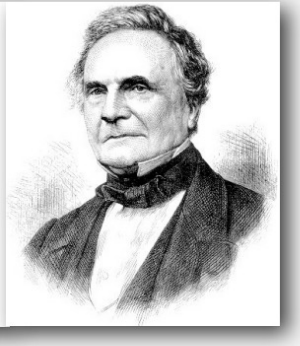
Foco da análise de algoritmos!

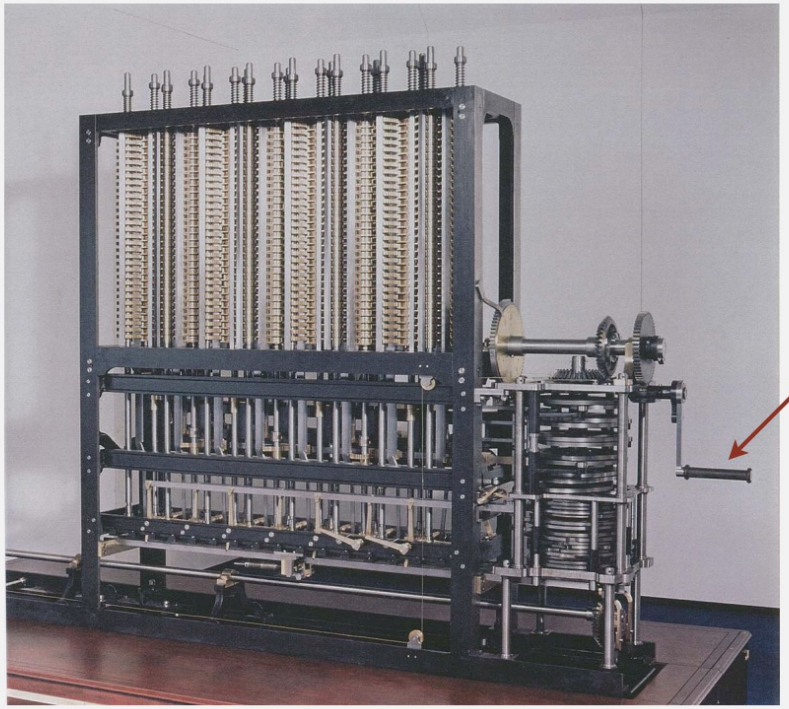
Análise de Complexidade

- A análise é feita levando em consideração **passos considerados relevantes** realizados pelo algoritmo durante sua execução sobre uma entrada genérica de tamanho n.

- O tempo de execução **T** de um algoritmo passa a ser visto como a soma total do tempo de execução de cada uma de suas etapas.

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





how many times do you have to turn the crank?

Analytic Engine

Exemplo

- Criar um algoritmo que encontre o máximo valor em um vetor. Calcule sua complexidade.
- Em que situação o custo do algoritmo é menor?

Insertion-Sort(A)

```
                                                    //custo   tempo
1.  Para j = 2 até tamanho(A) faça                  //c1
2.      chave = A[j]                                //c2
3.      i = j – 1                                   //c3
4.      Enquanto i > 0 e A[i] > chave faça          //c4
5.          A[i+1] = A[i]                           //c5
6.          i = i – 1                               //c6
7.      Fim_Enquanto
8.      A[i+1] = chave                              //c7
9.  Fim_Para
```

Insertion-Sort(A)

		//custo	tempo
1.	**Para** $j = 2$ até *tamanho*(A) **faça**	//c_1	n
2.	$chave = A[j]$	//c_2	n-1
3.	$i = j - 1$	//c_3	n-1
4.	**Enquanto** $i > 0$ e $A[i] > chave$ **faça**	//c_4	$\sum_{j=2}^{n} t_j$
5.	$A[i+1] = A[i]$	//c_5	$\sum_{j=2}^{n} (t_j - 1)$
6.	$i = i - 1$	//c_6	$\sum_{j=2}^{n} (t_j - 1)$
7.	**Fim_Enquanto**		
8.	$A[i+1] = chave$	//c_7	n-1
9.	**Fim_Para**		

Pior Caso, Caso Médio e Melhor Caso

- Tempo de execução do Insertion-Sort:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j$$

$$+c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

- No melhor caso, $t_j = 1$
- No pior caso, $t_j = j$

Pior Caso, Caso Médio e Melhor Caso

- Por quê fazer a análise do pior caso é mais interessante?

Pior Caso, Caso Médio e Melhor Caso

- Por quê fazer a análise do pior caso é mais interessante?
  - O tempo de execução no pior caso nos dá uma noção do tempo máximo necessário para que o algoritmo execute com qualquer entrada de tamanho n.
  - A frequência de ocorrência do pior caso é muito grande para determinados algoritmos. Ex.: na busca por uma entrada inexistente em um banco de dados.

Pior Caso, Caso Médio e Melhor Caso

- Complexidade de caso médio:

Pior Caso, Caso Médio e Melhor Caso

- Complexidade de caso médio:
  - É bastante usado quando há uma suposição inicial da distribuição dos dados de entrada.
  - Ex.: Análise probabilística.
  - Quando existe algum fator aleatório em alguma etapa do algoritmo, também é comum o uso da análise do caso médio para a obtenção do tempo de execução esperado.

Análise de Complexidade

- Considere que três algoritmos sejam capazes de resolver o mesmo problema. Como avaliar qual o mais custoso?

- $f_1(n) = 5n^2 + 10n$ operações
- $f_2(n) = 9n^2 + 50n + 300(n - 1)$ operações
- $f_3(n) = 300n + 5000 + 500.000n^2$ operações

Análise Assintótica

UNIVERSIDADE
FEDERAL RURAL
DE PERNAMBUCO

Análise Assintótica

A análise assintótica estuda a curva de crescimento da complexidade de um algoritmo em decorrência do tamanho da entrada fornecida para o mesmo, ou seja, o quanto o tempo de execução aumenta com o tamanho da entrada.

Análise Assintótica

- **Apenas o termo de maior ordem é considerado, sem seus coeficientes constantes**, tendo em vista que quando a entrada for muito grande os termos de menor ordem tornam-se irrelevantes no custo computacional final.
    - Ex: $f(n) = 2n^9 + 500n^4 + 1000n^2 + 89n = O(n^9)$
- Um método é considerado melhor que outro se sua curva de crescimento é menor de acordo com o aumento do tamanho da entrada fornecida.

Análise Assintótica

- Notação O
- Na notação O, é oferecido apenas um **limitante superior** à função.
- Dada uma função $g(n)$, denota-se por $O(g(n))$ ao conjunto de funções que:
  - $\{f(n)$: existem constantes positivas c e n_0 tais que $0 \leq f(n) \leq cg(n)$, para todo $n \geq n_0\}$
- Diz-se que $f(n) = O(g(n))$ ou $f(n) \in O(g(n))$

Busca Sequencial e consumo de tempo

- Qual a complexidade do algoritmo de busca sequencial?

```
1 int busca(int *v, int n, int x) {
2   int i;
3   for (i = 0; i < n; i++)
4     if (v[i] == x)
5       return i;
6   return -1;
7 }
```

- Qual o melhor caso?
- Qual o pior caso

Busca sequencial e consume de tempo

Consumo de tempo por linha no pior caso:
- Linha 2: tempo c_2 (alocação de variável)
- Linha 3: tempo c_3 (atribuições, acessos e comparação)
  - no pior caso, essa linha é executada *n+1* vezes
- Linha 4: tempo c_4 (acessos, comparação e *if*)
  - no pior caso, essa linha é executada *n* vezes
- Linha 5: tempo c_5 (acesso e *return*)
- Linha 6: tempo c_6 (*return*)

O tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6$$

```
1 int busca(int *v, int n, int x) {
2   int i;
3   for (i = 0; i < n; i++)
4     if (v[i] == x)
5       return i;
6   return -1;
7 }
```

Busca sequencial e consume de tempo

O tempo de execução é menor ou igual a $c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6$

Cada c_i não depende de n, depende apenas do computador

- Leva um tempo constante

Sejam $a = c_2 + c_3 + c_5 + c_6$, $b = c_3 + c_4$ e $d = a + b$

Se $n \geq 1$, temos que o tempo de execução é menor ou igual a

$$c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6 = c_2 + c_3 + c_5 + c_6 + (c_3 + c_4) \cdot n$$
$$= a + b \cdot n \leq a \cdot n + b \cdot n = d \cdot n$$

Isto é, o crescimento do tempo é linear em n

- Se n dobra, o tempo de execução praticamente dobra.

Notação Assintótica

Como vimos antes, existe uma constante **d** tal que, para $n \geq 1$,

$$c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6 \leq dn$$

d não interessa tanto, depende apenas do computador…
Estamos preocupados em estimar

O tempo do algoritmo é da **ordem de *n***
- A **ordem de crescimento** do tempo é igual a **f(n)=n**

Dizemos que

$$c_2 + c_3 \cdot (n+1) + c_4 \cdot n + c_5 + c_6 = O(n)$$

Outros Exemplos

$$1 = O(1)$$
$$1.000.000 = O(1)$$
$$5n + 2 = O(n)$$
$$5n^2 + 5n + 2 = O(n^2)$$
$$\log_2 n = O(\log_{10} n)$$
$$\log_{10} n = O(\log_2 n)$$

Nomenclatura e Consumo de Tempo

$O(1)$: tempo constante

- não depende de n

Ex: - atribuição e leitura de uma variável;
- operações artiméticas: +, -, x, /
- comparações (<, <=, ==, >=, ||, |, !)
- acesso a uma posição de um vetor

Nomenclatura e Consumo de Tempo

$O(1)$: tempo constante

- não depende de n

Ex: - atribuição e leitura de uma variável;
- operações artiméticas: +, -, x, /
- comparações (<, <=, ==, >=, ||, |, !)
- acesso a uma posição de um vetor

$O(\lg n)$: logarítmico

- lg indica $\log_2$
- quando ***n*** dobra, o tempo aumenta em **1**

Ex: Busca binária

Nomenclatura e consume de tempo

$O(n)$: linear

- quando **n** dobra, o tempo dobra
- Ex: busca linear
- Ex: encontrar o máximo/mínimo de um vetor
- Ex: Produto interno de dois vetores

Nomenclatura e consume de tempo

$O(n)$: linear

- quando **n** dobra, o tempo dobra
- Ex: busca linear
- Ex: encontrar o máximo/mínimo de um vetor
- Ex: Produto interno de dois vetores

$O(n \lg n)$:

- quando n dobra, o tempo mais que dobra
- Ex: algoritmos de ordenação que veremos

Nomenclatura e consume de tempo

$O(n)$: linear
- quando **n** dobra, o tempo dobra
- Ex: busca linear
- Ex: encontrar o máximo/mínimo de um vetor
- Ex: Produto interno de dois vetores

$O(n \lg n)$:
- quando n dobra, o tempo mais que dobra
- Ex: algoritmos de ordenação que veremos

$O(n^2)$: quadrático
- quando **n** dobra, o tempo quadriplica
- Ex: BubbleSort, SelectionSort, InsertionSort

Nomenclatura e consume de tempo

$O(n)$: linear
- quando **n** dobra, o tempo dobra
- Ex: busca linear
- Ex: encontrar o máximo/mínimo de um vetor
- Ex: Produto interno de dois vetores

$O(n \lg n)$:
- quando n dobra, o tempo mais que dobra
- Ex: algoritmos de ordenação que veremos

$O(n^2)$: quadrático
- quando **n** dobra, o tempo quadriplica
- Ex: BubbleSort, SelectionSort, InsertionSort

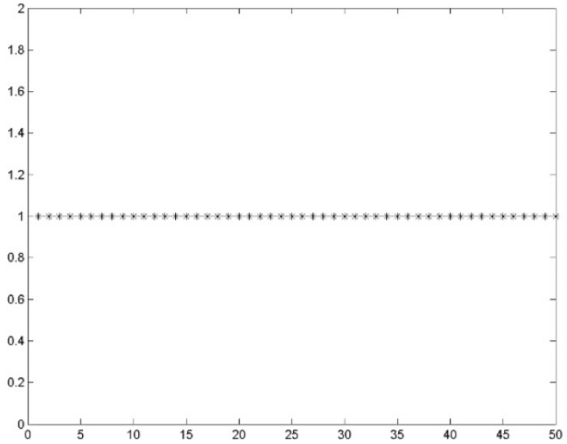
$O(n^3)$: cúbico
- quando **n** dobra, o tempo octuplica
- Ex: multiplicação de matrizes $n \times n$

Análise Assintótica

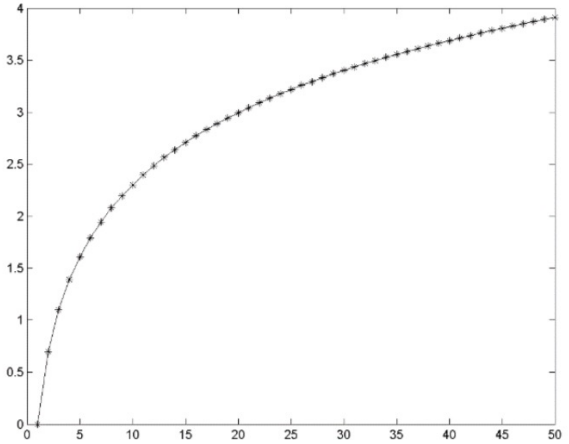
CLASSE	NOME
$O(1)$	Constante
$O(\log n)$	Logarítmica
$O(n)$	Linear
$O(n \log n)$	$n \log n$
$O(n^2)$	Quadrática
$O(n^3)$	Cúbica
$O(n^k), k \geq 1$	Polinomial
$O(2^n)$	Exponencial
$O(a^n), a > 1$	Exponencial

Análise Assintótica

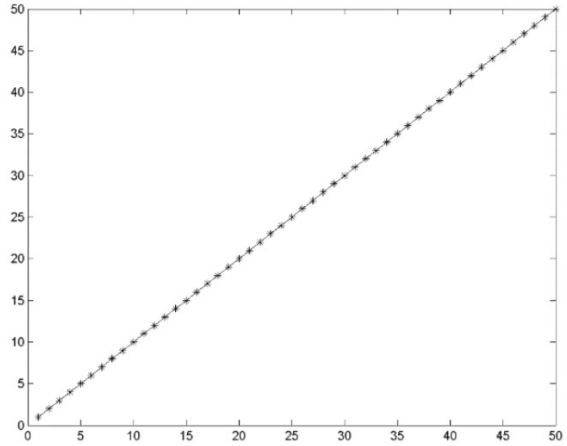
$O(1)$



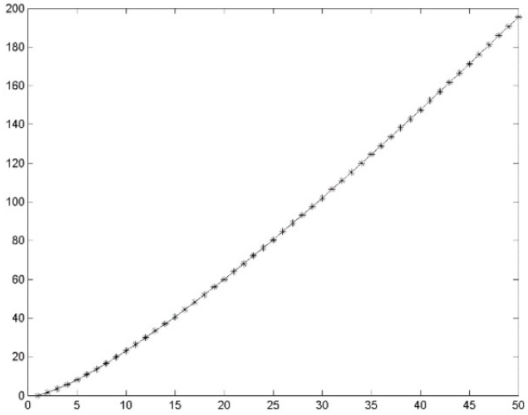
$O(\log n)$



$O(n)$

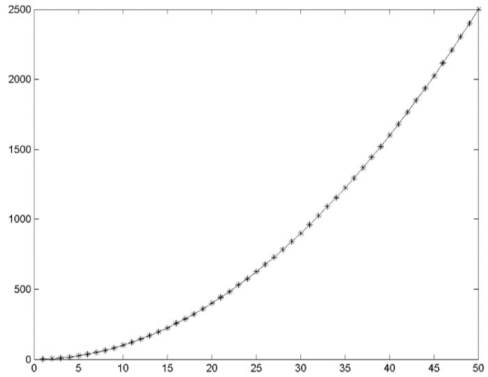


$O(n \log n)$

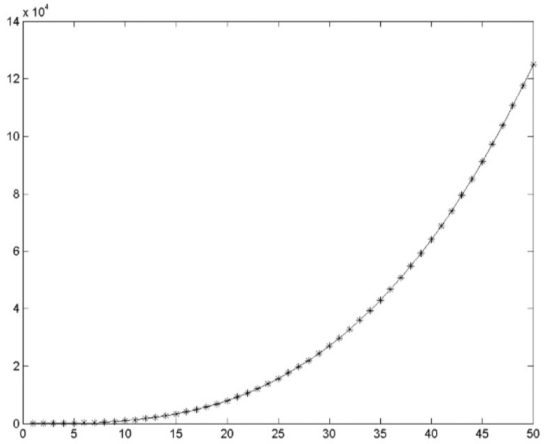


Análise Assintótica

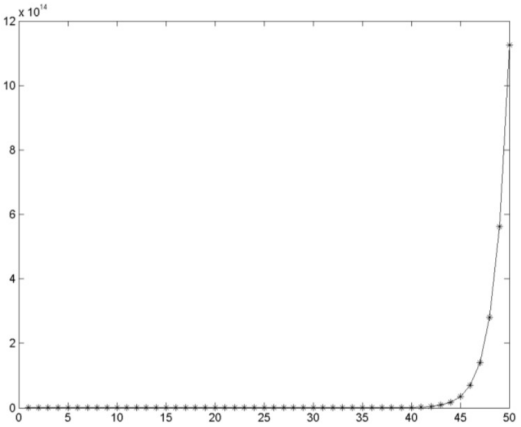
$O(n^2)$



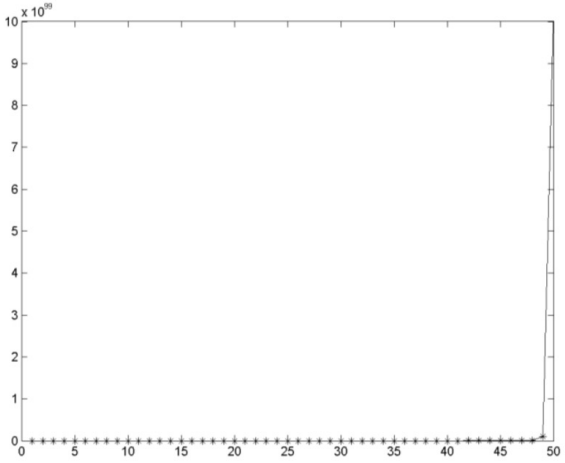
$O(n^3)$



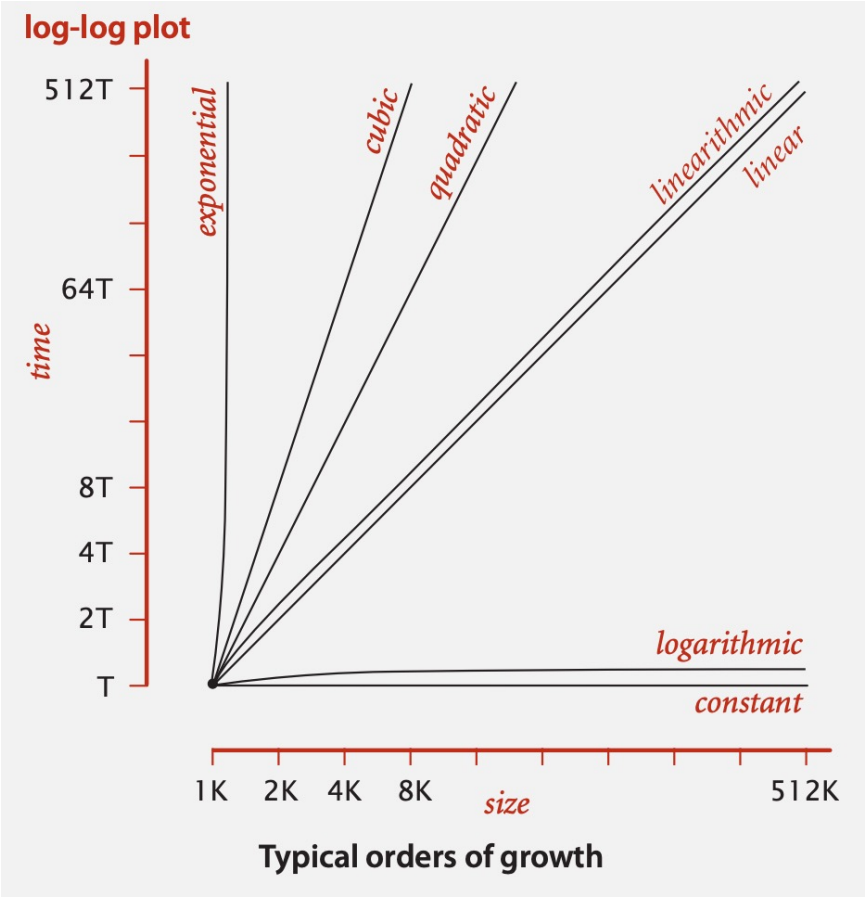
$O(2^n)$



$O(a^n)$



Análise Assintótica

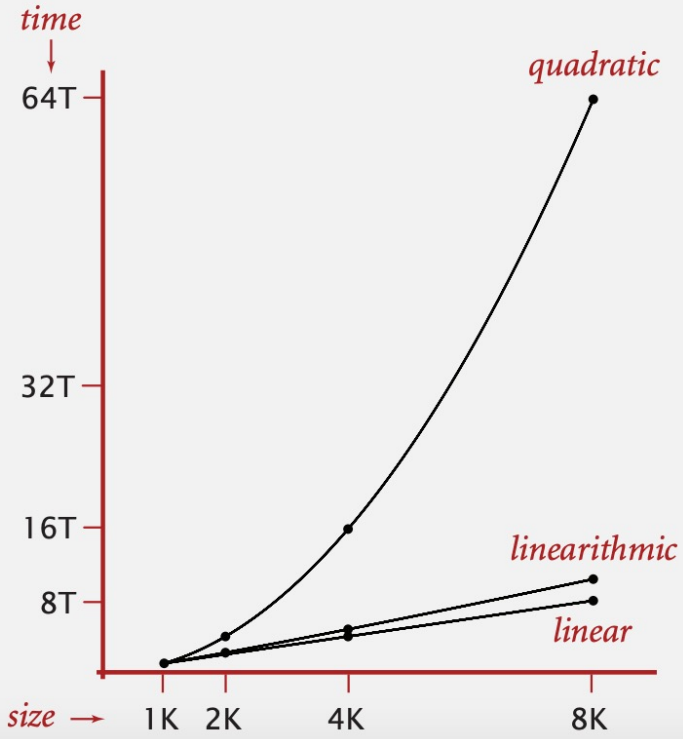


order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

Discrete Fourier transform.

- Break down waveform of N samples into periodic components.
- Applications: DVD, JPEG, MRI, astrophysics, ...
- Brute force: N^2 steps.
- FFT algorithm: $N \log N$ steps, enables new technology.

Friedrich Gauss
1805



growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

Exercício

- Que tipo de crescimento melhor caracteriza cada uma dessas funções? (constante, linear, polynomial, exponencial)
- $\left(\frac{3}{2}\right)^2$
- 1
- $\left(\frac{3}{2}\right)n$
- $2n^3$
- $2n^2$
- $3n^2$
- 1000
- $3n$

Exercícios

1. Escreva um algoritmo que receba como entrada a matriz A e obtenha sua matriz transposta.
2. Escreva um algoritmo que receba o vetor de dados x n-dimensional e retorne o resultado da seguinte função:

$$f(x) = \sum_{i=1}^{n} x^i$$

- Obs: Calcule a complexidade assintótica dos algoritmos dos problemas acima.
- Obs: usar laços de repetição e comandos de decisão, além de operações aritméticas fundamentais (suponha que a linguagem não possui a função para potenciação).

Exercícios - Extra

- Calcule o número de operações e complexidade assintótica do algoritmo de fatorial abaixo:

```
int n = input;
int fat = 1;
for (int i = n; i > 0; i--)
{
      fat *= i;
}
```

Exercícios - Extra

- Exemplo: Considere os seguintes trechos do código abaixo.
- Quantas vezes cada um deles é executado? Analise a complexidade assintótica.

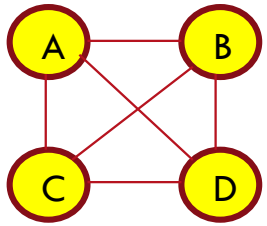
```
int contaTriplas(int vetor[], int n)
{
  int contagem = 0;
    for (int i = 0; i < n; i++)
      for (int j = i + 1; j < n; j++)
        for (int k = j + 1; k < n; k++)
          if (vetor[i] + vetor[j] + vetor[k] == 0)
            contagem++;

  return contagem;
}
```

- `if (vetor[i] + vetor[j] + vetor[k] == 0)`

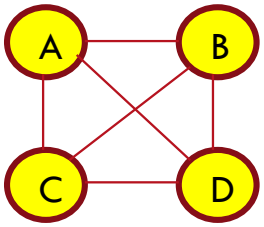
- `int contagem = 0;`

Exercícios- Extra



- Problema do caixeiro viajante
  - Suponha que um caixeiro deve visitar n cidades, encerrando a viagem na cidade de origem
  - De cada cidade, pode-se ir para todas as outras
  - Para se ir de uma cidade para outra existe um custo
    - Tempo
    - Pedágio
    - Custo de gasolina, etc..
      - Ou ainda uma combinação de vários fatores

Exercícios - Extra

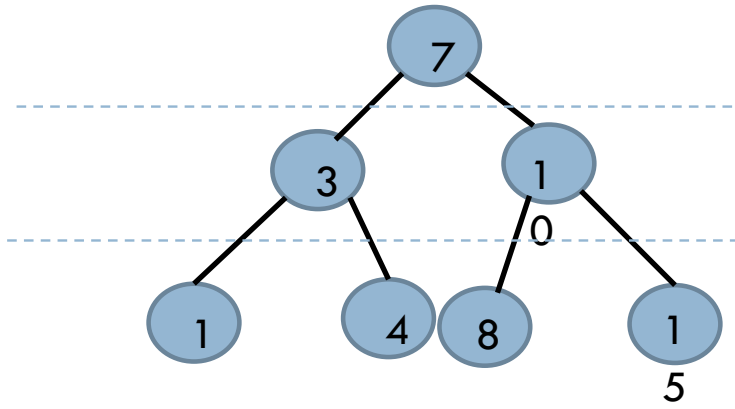


- Problema do caixeiro viajante
  - Sentido da viagem na importa
    - custo(a,b) = custo(b,a)
- Problema: encontrar a melhor rota
- Solução força-bruta: calcular todas as rotas possíveis

- Qual a complexidade assintótica do algoritmo para encontrar a melhor rota?

Exercícios - Extra

- Busca em um árvore binária de busca (completa)
- Custo da busca: altura da árvore
- *Qual a complexidade assintótica da busca em uma árvore binária ordenada?*



Número de nós até o nível

$2^{0+1} - 1 = 1\ nó$

$2^{1+1} - 1 = 3\ nós$

$2^{2+1} - 1 = 7\ nós$

Exercícios - Extra

Análise de funções `busca`, `maximo` e `twosum`

```c
//Busca o maior número e retorna seu valor
int maximo (int n, int v[]) {
   int k, m = v[0];
   NOP = 0;
   for(k = 1; k < n; k++) {
       NOP++;
       if(v[k] > m)
           m = v[k];
   }
   return m;
}
```

```c
//Faz uma busca do índice onde o número x gerado aleatoriamente está no vetor.
int busca (int x, int n, int v[]) {
   int k;
   k = n - 1;
   NOP = 0;
   while (k >= 0 && v[k] != x) {
       k -= 1;
       NOP++;
   }
   return k;
}
```

```c
//Verifica se a soma de dois números dentro do vetor é igual ao valor k
int twosum (int k, int n, int v[]) {
   int i, j, r;
   NOP = 0;
   r = 0;
   for(i = 0; i < n; i++)
       for(j = 0; j < n; j++) {
           NOP++;
           if(v[i] + v[j] == k)
               r = 1;
       }
   return r;
}
```