

## Algoritmo 1

### Sequência de Fibonacci

$$\left\{ \begin{array}{l} F(0) = 0 \\ F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \end{array} \right.$$

(não resolvendo)  $\rightarrow$  (0:n)  $\approx$  (términos) resolvendo n!

$$\left. \begin{array}{l} O = n \\ O = 1 \end{array} \right\} = n$$

↓ abstrato

### Método 1

(n) : (programa) 1.0; 0.0 → abstrato

fun fibonacci (n:Int): Int = if (n ≤ 1) n else fibonacci (n-1) + fibonacci (n-2)

- Complexidade temporal:  $O(2^n) \rightarrow \sum_{i=0}^{n-1} 2^i = 2^0 \times \frac{2^n - 1}{2 - 1} = 2^n$
  - Complexidade espacial:  $O(n)$
- ↓ soma dos n primeiros termos da sucessão geo.

### Método 2 → Pg Dinâmica - Técnica de Memorização

```
fun fibonacci (n:Int): Int {
    val f = IntArray (n+1)
    f[0] = 0
    f[1] = 1
    var i = 2
    while (i ≤ n) {
        f[i] = f[i-1] + f[i-2]
        i ++
    }
    return f[n]
}
```

+1 por causa de  $f[0] = 0$  e  $f[1] = 1$   
caso n=0  
 $(s(n), 0)$  resolvendo = 5 lev  
→ Não é necessário calcular valores repetidamente,  
simplesmente vai se memorizando

- Complexidade temporal/espacial:  $O(n)$

## Potência 1

$$a^n = \begin{cases} 1, & n=0 \\ a \cdot a^{n-1}, & n \neq 0 \end{cases}$$

## Método 1

fun power(a:Int, n:Int) = if(n==0) 1 else a \* power(a, n-1)

- Complexidade espacial/temporal:  $O(n)$

## Método 2

$$a^n = \begin{cases} 1, & \text{se } n=0 \\ a^{\frac{n}{2}} * a^{\frac{n}{2}}, & \text{se } n \text{ é par} \\ a^{\frac{n-1}{2}} * a^{\frac{n-1}{2}} * a, & \text{se } n \text{ é ímpar} \end{cases}$$

fun power(a:Int, n:Int): Int {  
 val z = power(a, n/2)  
 return if(a%2==0) z\*z  
 else z\*z\*a  
 else if(n==0) return 1}

- Complexidade temporal/espacial:  $O(\log_2 n)$

Nota: nem chamar o z e usar 2 vezes power(a, n/2) entra  
 Fica  $O(n)$

## SubArray Maximal

## Algorithm 2

↳ Maior subarray ( $a, l, r$ ) no array  $a$  tal que a soma dos seus elementos seja o máximo possível

$(a, \underbrace{l}_{\substack{\text{índice} \\ + à esq \\ \text{inclusive}}}, \underbrace{r}_{\substack{\text{índice} \\ + à direita \\ \text{inclusive}}})$

### Método 1

```
fun subarrayMax(array: DoubleArray, left: Int, right: Int) {
```

```
    var l = left
    var r = left - 1 ]→ caso não encontre sabemos q da erro
    var bestSum = 0.0
    var actualSum: Double
    for (i in left..right) {
        actualSum = 0.0
        for (j in i..right) {
            actualSum += array(j)
            if (bestSum < actualSum) {
                bestSum = actualSum
                r = j
                l = i
            }
        }
    }
}
```

→  $O(n^2)$

$n = \underline{right - left + 1}$

2 for loops

### Método 2 → Sliding Window (aula 1)

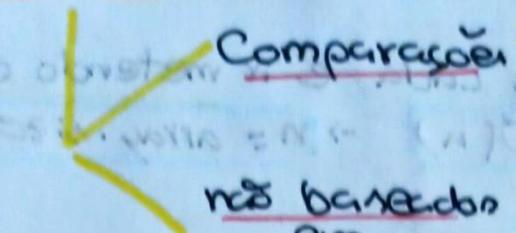
↳  $O(n)$

## Algoritmo 3

### Algoritmos de Ordenação

elementares

insertion  
selection  
bubble  
merge



não elementares

quick  
heap

não baseados  
em  
Comparação

quicksort  
counting

radix

### Insertion Sort

- inserir ordenadamente
- $n = \text{right} - \text{left} + 1$
- Melhor caso : estar ordenado de modo crescente  $\Omega(n)$
- Pior caso : ordenado de modo decrescente  $O(n^2)$

Complexidade  
temporal

### Selection Sort

- selecionar o  $i$ -ésimo menor elemento do array
- $n = \text{right} - \text{left} + 1$
- Complexidade temporal :  $\Theta(n^2)$

Nota:

$\Omega$  → limite inferior

$O$  → limite superior

$\Theta$  → limite inferior e igual ao superior

### Bubble Sort

- o menor elemento não ordenado "desce"
- $n = \text{right} - \text{left} + 1$
- Complexidade temporal :  $\Theta(n^2)$

Legenda:

- - estável
- - não estável

### Estabilidade

- um algoritmo de ordenação é estável se preserva a ordem relativa das chaves/elementos iguais

### Merge 3

↳ juntar arrays ordenados de modo crescente mantendo a ordenação;

↳ Comparar o 1º elemento de cada e ir metendo o menor.

↳ Complexidade temporal:  $\Theta(n)$  →  $n = \text{array.size}$

### Merge Sort 3

↳ dividir para conquistar

↳  $n = \text{right} - \text{left} + 1$

↳ dividir o array em partes e depois dar merge

↳ Complexidade temporal:  $\Theta(n \log_2 n)$

### Binary Search 3

↳ array tem de estar ordenado de modo crescente;

↳ dividir ao meio e ver se o elemento é igual, menor ou maior que o elemento do meio:

```

if (array[mid] == elem) return mid
else if (array[mid] < elem) return binarySearch(..., mid+1, right)
else return binarySearch(..., left, mid-1)
  
```

↳ Complexidade temporal:  $O(\log_2 n)$

Desafio → lower Bound - encontrar o primeiro elemento através da binary search

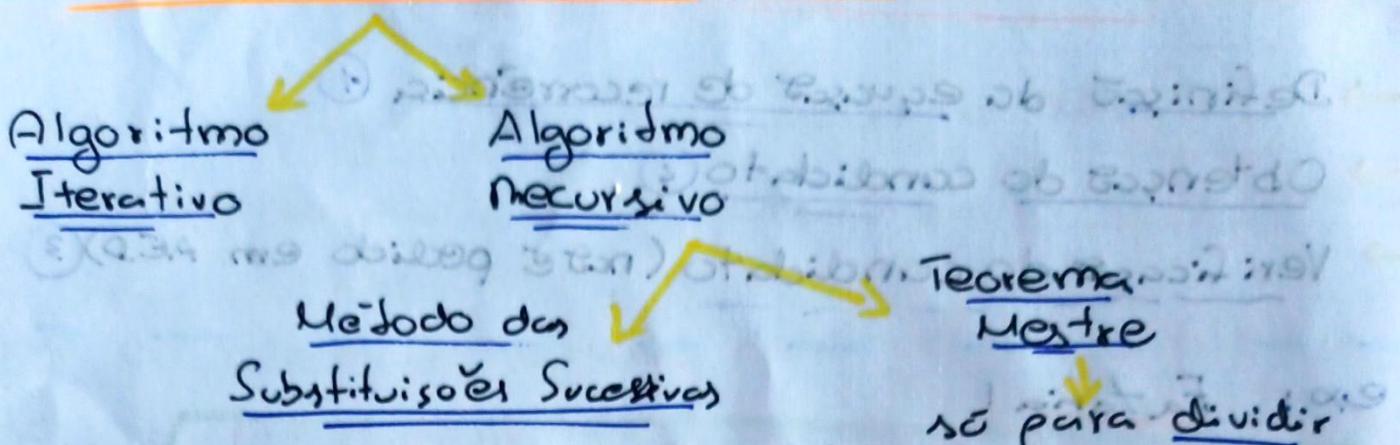
: abrigar  
leitura:  
lancar n:

$(\approx n)\Theta$  : interrogar gabinete:  $\Theta(n \log_2 n)$

s gabinete: dentro

avaliar se levanta: se não: o membro do contingente mu  
nicipal: estabelecer os autores: membro do

# Análise da Complexidade Assintótica (temporal)



## Notações

- $\mathcal{O}$  - limite assintótico superior
- $\Omega$  - limite assintótico inferior
- $\Theta$  - O e \Omega

## Propriedades da Notação O

Com  $f, g$  real positiva,  $C > 0$ :  $(n)O \times n = (n)O$

$$f = O(f)$$

$$C \cdot O(f) = O(C \cdot f) = O(f)$$

$$O(f) + O(g) = O(f+g) = O(\max(f, g))$$

$$O(f) \cdot O(g) = O(f \cdot g)$$

## Algoritmo iterativo

↳ fazer tabela

↳ obter e simplificar a expressão

$$(3)O = (n)O, (O \times n)O = (n)O, (3 + 2n)O = 3 + 2nO$$

$$(48n)O = (n)O, (3 + 48n)O = 3 + 48nO$$

## Método das substituições recursivas

→ Definição da equação de recorrência ①

→ Obtenção do candidato ②

→ Verificação do candidato (não é pedido em AED) ③

ex: Fatorial

```
fun factorial(n: Int): Int {
    return if (n == 0) 1 else n * factorial(n - 1)
}
```

①

$$C(n) = \begin{cases} O(1) & \text{se } n=0 \\ O(1) + C(n-1), & n>0 \end{cases}$$

Custo  
Fatorial

②

$$\begin{aligned} C(n) &= n \times O(1) + C(n-1) \\ &= O(n) + O(1) \\ &= O(n) \end{aligned}$$

Nota: Quando o algoritmo é dividir para conquistar tende-se a substituir  $n$  por  $x^m$ , sendo  $x$  o  $n$ : de partição (ex: mergeSort  $n \rightarrow 2^m$ )

Teorema Mestre (recorrências "dividir para conquistar")

$$C(n) = a \times C\left(\frac{n}{b}\right) + f(n), \quad a \geq 1, b > 1 \text{ e } f \text{ real}^+$$

Caso 1 →  $f = O(n^{\log_b a - \epsilon})$ , para um  $\epsilon > 0 \Rightarrow C(n) = \Theta(n^{\log_b a})$

Caso 2 →  $f = \Theta(n^{\log_b a})$ , para um  $\epsilon > 0 \Rightarrow C(n) = \Theta(n^{\log_b a} \log_2 n)$

Caso 3 →  $f = \Omega(n^{\log_b a + \epsilon})$ , para um  $\epsilon > 0 \Rightarrow C(n) = \Theta(f)$   
e  $a \times f\left(\frac{n}{b}\right) \leq c \times f(n)$ ,  $c > 1$

## Binary Heap - Amontoados binários

MaxHeap  $\rightarrow$  pai  $\geq$  filhos; usado no heap sort

MinHeap  $\rightarrow$  pai  $\leq$  filhos; usado em Priority Queue

L Todos os níveis da árvore devem estar completos exceto o último; o último se for incompleto tem de estar preenchido da esquerda para a direita.

L Funções left & right retornam o índice dos filhos

L Função parent retorna o índice do pai

## HeapSort

buildMaxHeap  $\rightarrow$  transformar o array num maxHeap

heapify  $\rightarrow$  voltar a transformar num maxHeap mas só 1 elemento está mal

Complexidade  $O(n \log n)$

heapify  $\rightarrow$   $O(\log n)$ , chamado  $n-1$  vezes

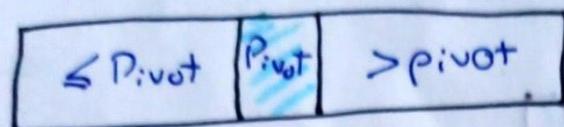
buildMaxHeap  $\rightarrow$   $O(n)$

## Priority Queue

- ↳ usa minHeap
- ↳ quem tá no topo do Heap tem prioridade
- ↳ Função offer (element) → adiciona no (~~em~~ do) Heap
  - ↳ decreaseKey → é usado no offer para colocar no index correto tendo em conta a prioridade
- ↳ Função peek () → retorna o heap[0] (tem prioridade)
- ↳ Função poll () → retorna o "sorteado", coloca o último no heap[0] e chama minheapi
- ↳ minheapi → coloca o valor em heap[0] na posição correta
- ↳ Função update (newStatus: Utente) atualizar o status do utente, chamando heapi ou decrease

## QuickSort

- ↳ escolher 1º pivot
- ↳ colocar o pivot na sua posição correta e final



3 - 0 : 1 2 3 4 5 6 7 8 9

- ↳ fazer o mesmo processo nas outras duas partições

Partition

Hoare

Lomuto

- ↳ não é estável

## Escolher o pivot

- ↳ escolher o meio à direita da esquerda
- ↳ escolher a mediana (left, mid, right)

## Complexidade

Partition  $\rightarrow \Theta(n)$

QuickSort  $\rightarrow$  Melhor caso  $\rightarrow \Omega(n \log_2 n)$   
 $\rightarrow$  Pior caso  $\rightarrow O(n^2)$

Pivot já na posição

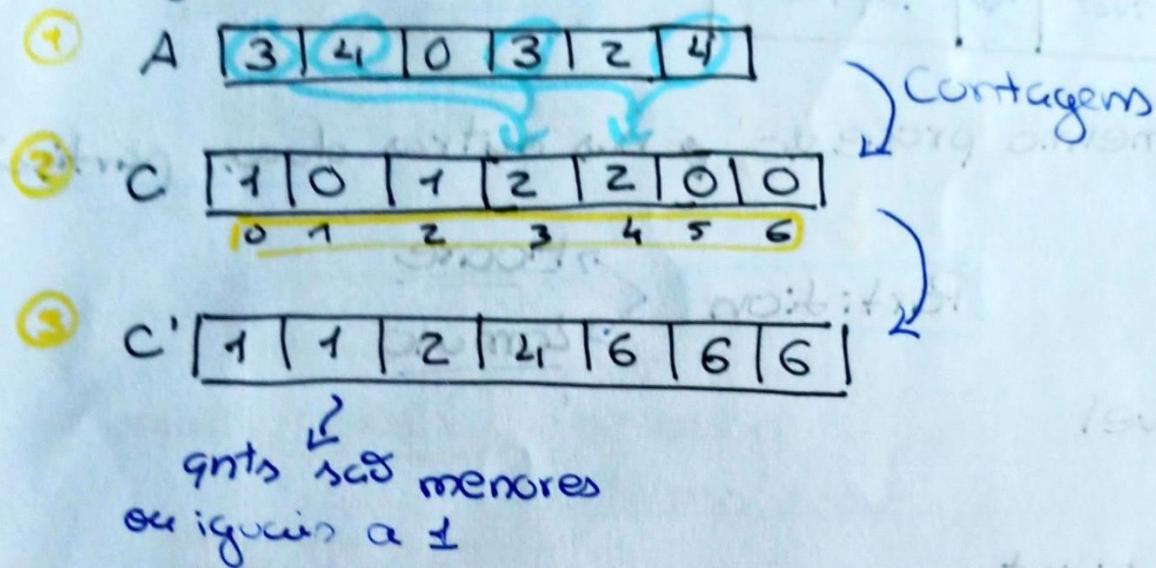
pivot < ou  
 que todos os elementos

(melhor) big-O: tempo gasto é proporcional ao logarítmico do número de elementos  
 (pior) big-O: tempo gasto é proporcional ao quadrado do número de elementos

## Counting Sort

- ↳ Conta o nº de aparições de cada número da gama do array

ex: gama: 0 - 6



- ④ Construir o B a partir de C e A

↳ É estável

↳ Gama =  $T = \lceil \max - \min + 1 \rceil = O(n)$

↳ O menor é  $\Theta(n)$ , ne o nº de elementos de A

## Radix Sort

- ↳ Para strings da mesma dimensão

↳ **MSD** → Most Significant Digit (+ esquerda)  
↳ **LSD** → Least Significant Digit (+ direita)

↳ Custo é  $O(n)$

↳ se for LSD é também  $O(n)$

## ADT - Abstract Data Type

tipos básinos

↳ especificações do conjunto de dados e operações que podem ser executadas sobre esses dados

### Interface

↳ contém declarações de métodos abstratos

```
interface Test {
    fun bar()
    fun foo(){}
    ...
}
```

```
class Child : Test Test {
    override fun bar() {
        ...
    }
}
```

Pode-se implementar  
n'interfaces,  
separadas  
por vírgula

↳ Definir a interface

↑ Implementar uma classe com a Interface

↳ Uma interface pode derivar de outras:

```
interface Person : Named {
    ...
}
```

// podemos dar override das funções  
// da interface Named

→ Interface  
Person deriva de  
Named

O mutar tanto == nicht

(ausreichend) > mutar tanto < nicht

(ausgeglichen) > mutar tanto > nicht

## Linked List

é só uma estrutura de dados

Simplemente Ligada (next)

Duplicamente Ligada (previous e next)

Circular

Não circular

Com sentinela → sentinelas são usadas para controlar o início e fim da lista, para não referenciar null

Sem sentinela

head: é o †: Node

## Comparable e Comparator

São interfaces utilizadas para soltar coleções de elementos

Comparable → compara por 1 elemento  
afeta a própria classe

Comparator → compara por múltiplos elementos  
não afeta a classe

O comparador funciona assim:

→ this == that - return 0

→ this > that - return 1 (Positivo)

→ this < that - return -1 (Negativo)

## Iterable e Iterator

- Classe implementados com Iterable podem ser representados por uma sequência de elementos que podem ser iterados
- Classe são iterados por um Iterator que implementa a interface Iterator

### Stack

- LIFO - Last In, First Out
- Array com var size e adiciona-se em size e remove-se em size
- Linked List simples com adicionar e remover à cabeça
- Complexidade < Temporal - O(1)  
Enpacial - O(n)

### Queue

- FIFO - First In, First Out
- Array:
  - size do array
  - tail - próxima posição a adicionar
  - head - próxima posição a removerCalcular nova tail  
$$\text{tail} = (\text{tail} + 1) \% \text{size}$$
- Linked List simples + adicionar no fim (tail) e remover no início (head)
- Complexidade < Enpacial - O(n)  
Temporal - O(1)

# Hash Tables

<u>Interfaces</u>	<u>Classes</u>	<u>Implementações</u>
• Set	• HashSet	• Tables de Dispersion (HashTables)
• MutableSet	• LinkedHashSet	
• Map	• HashMap	
• MutableMap	• LinkedHashMap	

Nota: Nos "Linked" o índice garante a ordem de inserção ou percorrer os elementos

Set - conjunto / coleção de elementos (s / repetição)

Map - tabela de associação

Map < key, value >

Hashtable → O(1)

↳ tabela (array)

↳ funções de dispersão (hashcode) →

↳ algoritmo de resolução de colisões

x.hashCode() % n

↳ retorna a posição a inserir o elemento

↳ encadeamento externo com listas duplamente ligadas

si == sz ⇒ si.hashCode() == sz.hashCode()

↳ mas não o oposto

(base) class or interface

## AED - Dicas

- Sliding Window -  $O(n)$
- Insertion Sort -  $O(n^2)$
- Selection Sort -  $O(n^2)$
- Bubble Sort -  $O(n^2)$
- Merge -  $O(n)$
- Merge Sort -  $O(n \log_2 n)$
- Binary Search -  $O(\log_2 n)$
- Heapify -  $O(\log_2 n)$
- build Heap -  $O(n)$
- Heap Sort -  $O(n \log_2 n)$
- Partition -  $\Theta(n)$
- QuickSort -  $O(n \log_2 n)$
- Counting Sort -  $O(n+k)$ ,  $k$  é range
- Radix Sort -  $O(n)$

Estabilidade - preservar ordem  
relativa dos elementos iguais

## Estruturas de dados

- Heap (Array)
  - $\min\text{Heap}$
  - $\max\text{Heap}$
- Priority Queue ( $\min\text{Heap}$ )
- Stack (LIFO)
- Queue (FIFO)

# Binary Search Tree (BST)

- filho da esquerda  $\leq$  ao pai
- filho da direita  $\geq$  ao pai

## variantes

- Red Black Tree

Tree Set: conjunto ordenado > java.util  
Tree Map: mapa ordenado

- AVL Tree

Tries > Árvores N-áreas de Pesquisa  
BTree

```
data class Node<E>(val item:E,
                     var left:E?
                     var right:E?
                     var parent:E?)
```

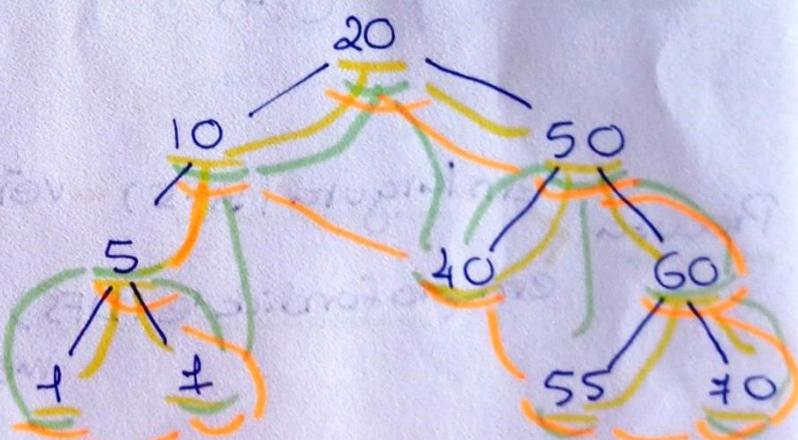
Não é muito usado

## Percursos

- Prefixo: visitar a root  
(Profundidade) depois a left  
depois a right

- Infixo: de modo crescente  
visitá vai a left  
visitá a root  
vai a right

- Sufixo: vai a left  
vai a right  
visitá a root



Prefixo : 20 10 5 1 50 40 ...

Infixo : 1 57 10 20 40 50 ...

Sufixo : 1 5 20 40 55 20 60 50

## Grafos

- Par de conjunto de vértices e arcos  $G = (V, E)$

- Orientação 

- Orientado
- Não orientado

- Custo 

- Pesoado
- Não Pesoado

 Denso → representar-se por matriz de adjacências  
Espesso → representar-se por listas de adjacências

	Matriz	Linha
<u>Criar</u>	$O(V^2)$	$O(V+E)$
<u>Inicializar</u>	$O(V^2)$	$O(V)$
<u>Inserir</u>	$O(1)$	$O(1)$
<u>Remover</u>	$O(1)$	$O(V)$
<u>Procurar</u>	$O(1)$	$O(V)$

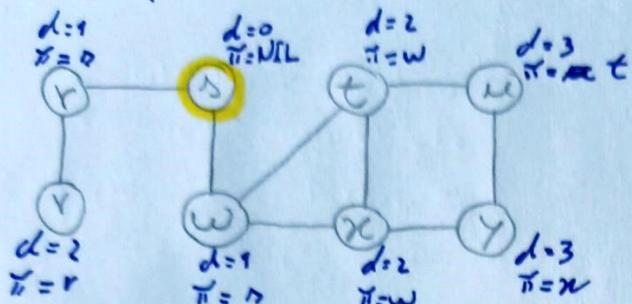
Procura 

- em largura (BFS) - vértices mais próximos primeiro
- em profundidade (DFS) - arcos dos vértices mais recentemente visitados

## Algoritmos de Procura

### Procura em largura (BFS)

- Visita por ordem de distância à origem

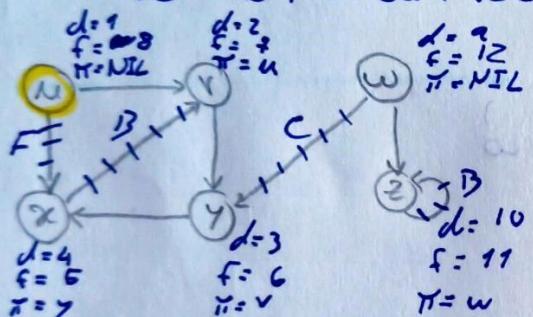


→ vértice inicial

Iniciaizações	Matriz	Cintos
Geral	$O(V^2)$	$O(V+E)$

### Procura em Profundidade (DFS)

- Prioridade com mais recentemente visitados

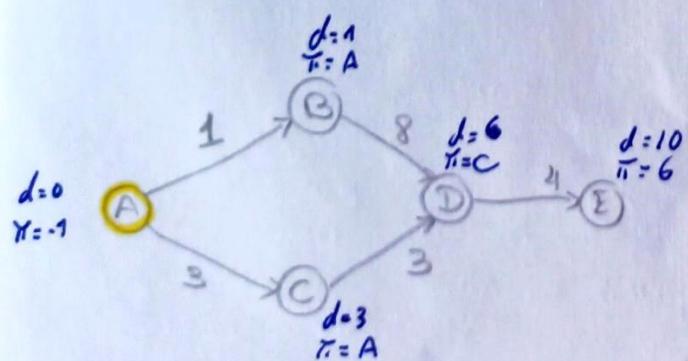


| B - back edges  
 | F - forward Edge  
 | C - cross edge

Complexidade:  $O(V+E)$

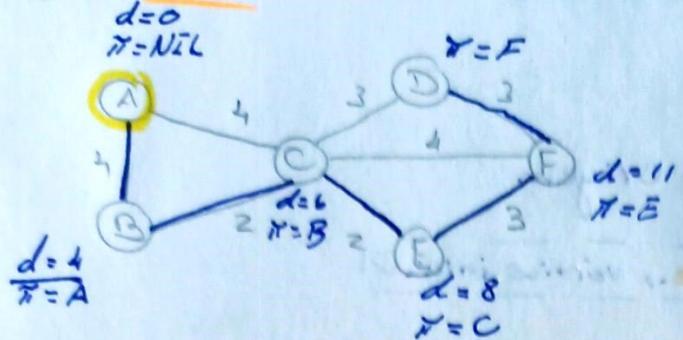
### Dijkstra

- caminho mais curto



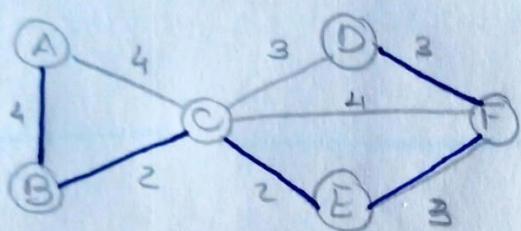
Complexidade:  $O(V+V \log V)$

## Algoritmo de Prim



• Complexidade:  $O(V^2)$

## Algoritmo de Kruskal



• Complexidade:  $O(E \log V)$

Ordenação Topológica  $\rightarrow$  grafo acíclico

1- invoca DFS

2- sempre que cada vértice está terminado (inverte-se f), inserir no inicio de uma lista ligada

3- retornar lista ligada

• Complexidade:  $O(V+E)$