

Análise e Desenho de Algoritmos

Ano Letivo 2022/23

Margarida Mamede
DI – FCT NOVA

Apresentação e Avaliação

Enquadramento na LEI e no MIEI

IP — **Introdução à Programação**

POO — **Programação Orientada pelos Objetos**

MD — **Matemática Discreta**

AED — **Algoritmos e Estruturas de Dados**

ADA — **Análise e Desenho de Algoritmos**

Descrição Geral

Estudo de algoritmos **eficientes** para resolver problemas fundamentais de **grafos**.

Aplicação de três **técnicas de desenho de algoritmos** (estratégias *greedy*, programação dinâmica e transformação-e-conquista).

Compreensão dos conceitos básicos da **Teoria da Complexidade**.

Programa (1)

- Programação dinâmica.
- Introdução ao estudo de grafos. Definições fundamentais. Tipos abstratos de dados grafo não orientado e grafo orientado. Implementações de grafos.
- Algoritmos elementares de grafos. Percursos em profundidade e em largura. Ordenação topológica.
- Árvores mínimas de cobertura. Algoritmo de Kruskal. Tipo abstrato de dados partição.
- Complexidade amortizada. Método do potencial.

Programa (2)

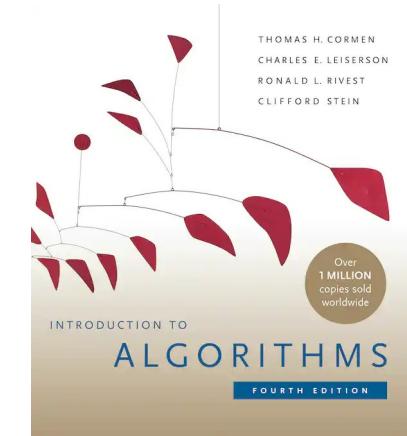
- Algoritmo de Prim. Tipo abstrato de dados fila com prioridade adaptável.
- Redes de fluxos. Fluxos máximos. Método de Ford-Fulkerson. Algoritmo de Edmonds-Karp. Emparelhamentos máximos em grafos bipartidos. Cortes mínimos.
- Caminhos mais curtos. Algoritmos de Dijkstra, Bellman-Ford e Floyd-Warshall.
- Introdução à Teoria da Complexidade. As classes P, NP, PSPACE e EXPTIME. Os sufixos difícil e completo. Redução de problemas. Alguns problemas em aberto.

Bibliografia Principal

Thomas H. Cormen, Charles E. Leiserson,
Ronald L. Rivest, and Clifford Stein.

Introduction to Algorithms (4th edition).

The MIT Press, 2022.

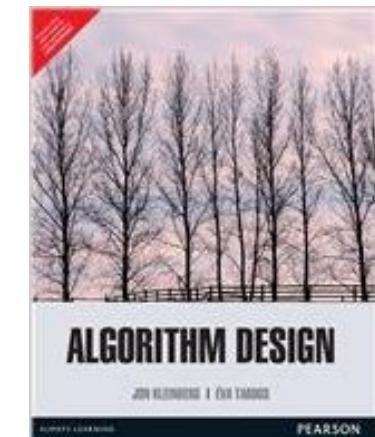
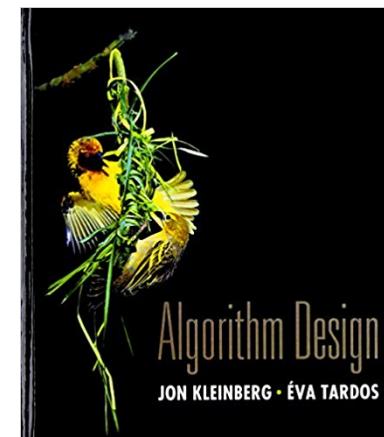


Jon Kleinberg and Éva Tardos.

Algorithm Design.

Pearson (hardcover), 2005.

Pearson (paperback), 2013.



Componentes de Avaliação

- A avaliação é constituída por duas componentes: a componente laboratorial e a componente teórico-prática.

Componente Laboratorial (1)

- A componente laboratorial é composta por três trabalhos.
- Cada trabalho consiste no desenho, na análise e na implementação (em Java 17) de um algoritmo para resolver um problema de um concurso de programação, na elaboração de um relatório e na realização de uma discussão.

Componente Laboratorial (2)

- As duas primeiras partes do trabalho (programa e relatório) são realizadas em grupo de dois alunos. Quem quiser fazer um trabalho sozinho tem de **pedir autorização** a um docente, justificando o tratamento excepcional.
- Há duas semanas para realizar (o programa de) cada trabalho. A segunda aula prática de cada turno é de apoio ao trabalho.
- Um grupo **entrega** um trabalho se o seu programa for **aceite** pelo Mooshak e o respetivo relatório (em PDF) for entregue no Moodle dentro do prazo.

Componente Laboratorial (3)

- Devem ser usadas as EDs da biblioteca de Java (pacote Java.util) sempre que forem as adequadas.
- Não são fornecidos testes de treino.
- Os relatórios devem ter a estrutura indicada no Moodle.
- A avaliação incidirá sobre todos os aspectos: correção e eficiência da solução, estrutura e documentação do código e correção do relatório.
- As notas dos trabalhos entregues variam entre 10 e 20 valores.

Componente Laboratorial (4)

- As discussões são para aferir o conhecimento que cada aluno tem sobre os trabalhos que entregou. São **obrigatórias** e **individuais**. Em princípio, decorrerão logo a seguir ao segundo teste.
- A nota de um aluno num trabalho depende:
 - do trabalho que entregou,
 - do seu desempenho na discussão e
 - do desempenho do seu colega de grupo na discussão, caso os desempenhos de ambos sejam significativamente desequilibrados.

Consequentemente, as notas dos dois elementos do grupo podem ser diferentes. Essas notas variam entre 0 e a nota do trabalho.

Componente Laboratorial (5)

- A nota da componente laboratorial (CompL) é a média das notas do aluno nos três trabalhos (P1, P2 e P3):

$$\text{CompL} = (P1 + P2 + P3)/3.$$

- Para obter **frequência**, é necessário que $\text{CompL} \geq 7.0$.
- As discussões dos trabalhos serão efetuadas presencialmente no fim do semestre, apenas com os alunos que puderem ter frequência (i.e. que tiverem entregado pelo menos dois trabalhos).

Componente Laboratorial (6)

Datas Provisórias

	Trabalho 1	Trabalho 2	Trabalho 3
Enunciado	18 Mar	29 Abr	13 Mai
Mooshak	20 Mar – 1 Abr	1 – 13 Mai	15 – 27 Mai
Aulas de apoio	27 e 28 Mar	8 e 9 Mai	22 e 23 Mai
Relatório	2 Abr	14 Mai	28 Mai
Discussões	12 de Junho (segunda-feira) às 17h		

Componente Teórico-Prática (1)

- A componente teórico-prática é composta por dois testes (no período de aulas) ou por um exame (na Época de Recurso).
- As três provas são presenciais, individuais, escritas e com consulta. Cada aluno pode consultar todo o material que quiser, desde que esteja em papel e o tenha levado para a prova. Durante a prova, não o pode emprestar, nem pedi-lo emprestado. Não são permitidos dispositivos eletrónicos (e.g. calculadoras, telemóveis, *tablets*, *smartwatches* e portáteis).
- É necessário apresentar um documento de identificação com fotografia.

Componente Teórico-Prática (2)

- A nota da componente teórico-prática (**CompTP**) é a média das notas dos testes (T1 e T2) ou a nota do exame (Ex):

$$\text{CompTP} = (T1 + T2)/2 \quad \text{ou} \quad \text{CompTP} = \text{Ex.}$$

- Para obter **aprovação**, é necessário que **CompTP** ≥ 9.5 .

Datas Provisórias

Teste 1	2 de Maio (terça-feira) às 18h30
Teste 2	12 de Junho (segunda-feira) às 14h
Exame	a ser marcado pelo Conselho Pedagógico

Nota Final

- (Não há nota final quando os alunos não têm frequência.)
- A nota final (**F**) dos alunos com frequência é:

$$F = \begin{cases} \text{CompTP}, & \text{se } \text{CompTP} < 9.5; \\ 0.3 \text{CompL} + 0.7 \text{CompTP}, & \text{se } \text{CompTP} \geq 9.5. \end{cases}$$

- Todas as notas (P1, P2, P3, T1, T2, Ex, **CompL** e **CompTP**) são arredondadas às décimas, exceto a nota final (**F**) que é arredondada às unidades.
- Qualquer aluno envolvido numa fraude (num trabalho, num teste ou no exame) reprova na disciplina.

Frequência e Classificações Anteriores

- Os alunos que obtiveram frequência em **2020/21** ou **2021/22** têm automaticamente frequência. No cálculo da nota final,

$$\text{CompL} = \max(\text{CompL-20/21}, \text{CompL-21/22}, \text{CompL-22/23}).$$

- A pauta com as classificações anteriores “válidas” está no Moodle.

Docentes dos Turnos Práticos

P1, P3 André Rijo

P2, P4 David Semedo

P5 Margarida Mamede

P6, P7 Mário Pereira

Início das Aulas Práticas: **Hoje**

Página de ADA

no **Moodle** (password: **adaPos**) e no CLIP

URL do Mooshak

<http://mooshak.di.fct.unl.pt/~mooshak/>

Erros Frequentes nos Trabalhos de ADA



Estrutura do Programa

- Constantes e Identificadores
- Tipos e Interfaces
- Entrada, Saída e Cálculo
- Encapsulamento
- *Enhanced for-loop*

Constantes e Identificadores

Definam constantes

Escolham nomes apropriados para as variáveis,
parâmetros, métodos, etc.

Tipos e Interfaces (1)

Usem as interfaces para declarar os tipos dos objetos

Exemplo: Declarar e criar um vetor de listas ligadas de inteiros, com comprimento *vecLen* (= 4), e criar as *vecLen* listas.

- **Declaração da variável:** um vetor de listas de inteiros.

- `LinkedList<Integer>[] vec;` ERRO
- `List<Integer>[] vec;` CORRETO

- **Criação do vetor:**

- `vec = new LinkedList<>[vecLen];` ERRADO
- `vec = new List<>[vecLen];` QUASE CORRETO

Tipos e Interfaces (2)

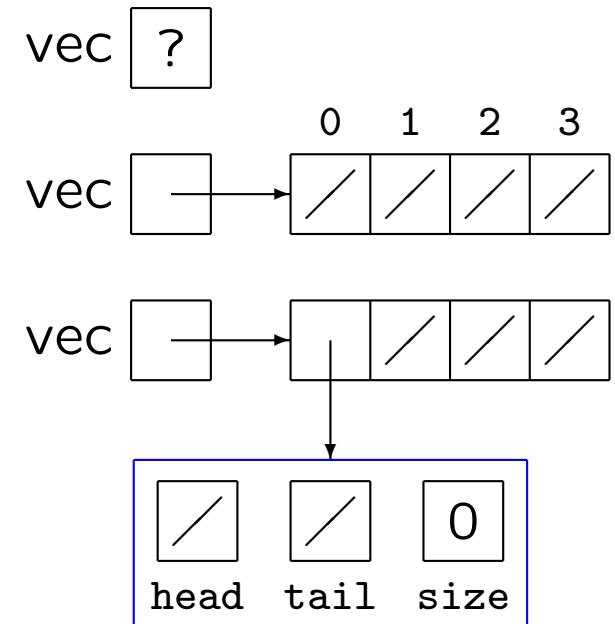
```
List<Integer>[] vec;
```

```
vec = new List<>[vecLen]; ERRADO
```

```
vec[0] = new LinkedList<>();
```

```
@SuppressWarnings("unchecked")
```

```
void createDataStructure( int vecLength ) {  
    vec = new List[vecLength];  
    for ( int i = 0; i < vec.length; i++ )  
        vec[i] = new LinkedList<>();  
}
```



CORRETO

Entrada, Saída e Cálculo (1)

A funcionalidade deve ser independente da sintaxe de entrada dos dados e da sintaxe de saída dos resultados

Exemplo: A entrada e a saída são compostas por dois inteiros, separados por um espaço.

```
public static void main( String[] args ) throws IOException {  
    BufferedReader input =  
        new BufferedReader( new InputStreamReader(System.in) );  
    String line = input.readLine();  
    Problem prob = new Problem(line);  
    String ans = prob.answer();  
    System.out.println(ans);  
}
```

2 ERROS

Entrada, Saída e Cálculo (2)

A funcionalidade deve ser independente da sintaxe de entrada dos dados e da sintaxe de saída dos resultados

```
public static void main( String[] args ) throws IOException {  
    BufferedReader input =  
        new BufferedReader( new InputStreamReader(System.in) );  
    String[] tokens = input.readLine().split(" ");  
    int n1 = Integer.parseInt(tokens[0]);  
    int n2 = Integer.parseInt(tokens[1]);  
    Problem prob = new Problem(n1, n2);  
    int[] ans = prob.answer();  
    System.out.println(ans[0] + " " + ans[1]);  
}
```

CORRETO

Encapsulamento

Os métodos não devem retornar membros de dados alteráveis do exterior

```
public class Graph {  
    private List<Integer>[] edges;  
  
    public List<Integer> adjacentNodes( int node ) {  
        return edges[node];  
    }  
}
```

ERRO

A correção deste erro tem de ser encontrada caso a caso.

Enhanced for-loop

Prefiram-no para iterar

```
Iterator<ElemType> iter = struct.iterator();
while ( iter.hasNext() ) {
    ElemType elem = iter.next();
    // PROCESS elem
}
```

MUITO LIXO

```
for ( ElemType elem : struct ) {
    // PROCESS elem
}
```

MAIS LEGÍVEL

Eficiência

- Criação de Objetos
 - Objetos Desnecessários
 - EDs Pseudodinâmicas
 - Acesso aos Caracteres das Strings
- Repetição de Cálculos
 - Repetição Visível
 - *containsKey + get* em Dicionários
 - *get* em Listas Ligadas

Criação de Objetos Desnecessários

Não criem objetos que nunca serão usados

```
int capacity = 5000;
```

```
Map<String, List<Integer>> map = new HashMap<>(capacity);  
map = this.createMap(capacity);
```

ERRO

```
Map<String, List<Integer>> map;  
map = this.createMap(capacity);
```

CORRETO

Criação de EDs Pseudodinâmicas

Não criem EDs **estáticas** com a capacidade por omissão

```
int capacity = 5000;      // Maximum size of the list.
```

```
List<T> list = new ArrayList<>();
```

ERRO

Inicialmente o vetor tem capacidade **10**,

ao inserir o 11º elem., copia-se a informação para um vetor com capacidade **20**,

ao inserir o 21º elem., copia-se a informação para um vetor com capacidade **40**,

e assim sucessivamente, para as capacidades **80, 160, 320, 640, 1280, 2560 e 5120**.

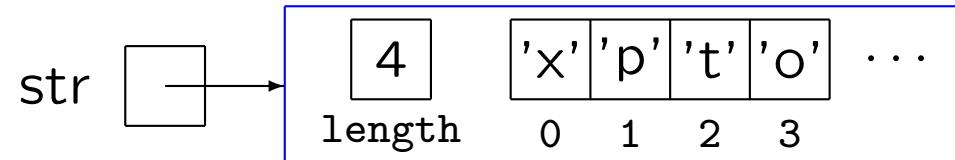
```
List<T> list = new ArrayList<>(capacity);
```

CORRETO

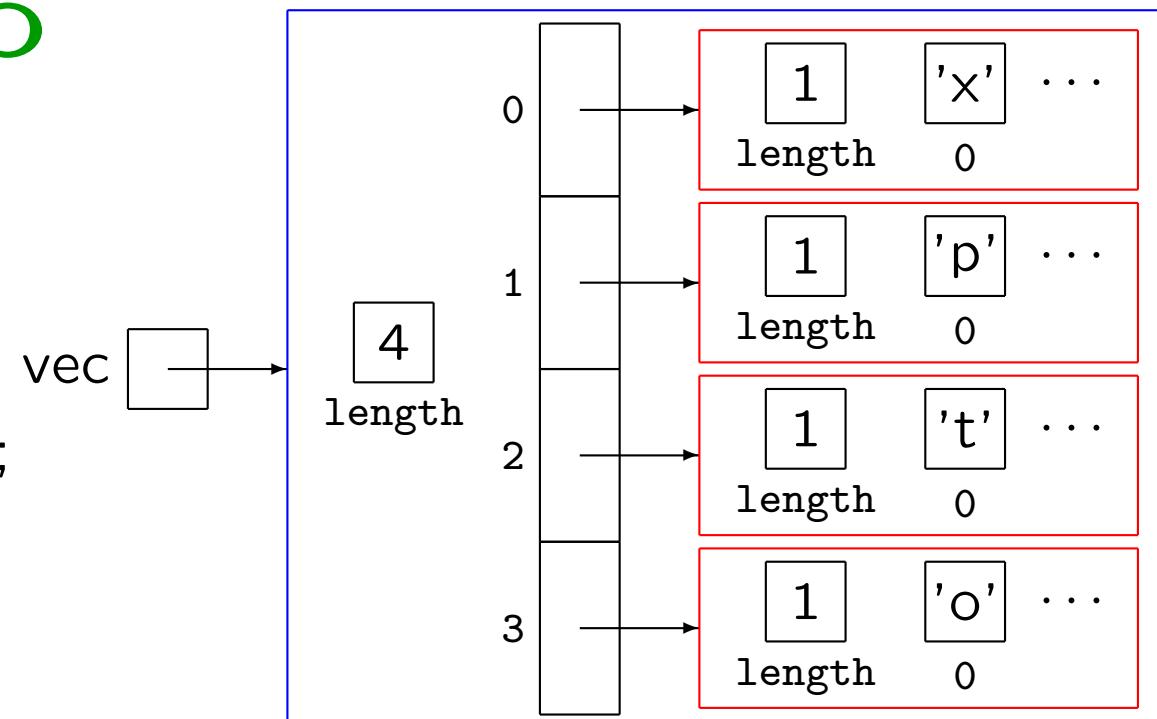
Acesso aos Caracteres das Strings

Usem o método *charAt*

```
String str = "xpto";  
char firstC = str.charAt(0);  
if ( firstC == 'y' )  
...  
CORRETO
```



```
String[] vec = str.split("");  
if ( vec[0].equals("y") )  
...  
ERRO
```



Repetição de Cálculos Visível

Não repitam cálculos para poupar variáveis

```
String[] vec = str.split(" "); // The first element is an integer.  
int res;
```

```
if ( Integer.parseInt(vec[0]) >= 0 )  
    res = Integer.parseInt(vec[0]) + 1;      ERRO  
else  
    res = Integer.parseInt(vec[0]) / 2;
```

```
int num = Integer.parseInt(vec[0]);  
if ( num >= 0 )  
    res = num + 1;                          CORRETO  
else  
    res = num / 2;
```

containsKey + get em Dicionários

se chave não existe, insere entrada; obtém sempre valor

```
Map<String, Integer> map = new ... ;
```

```
int value;  
if ( !map.containsKey(key) )  
    value = this.computeValue(key);  
    map.put(key, value);  
else  
    value = map.get(key);
```

ERRO

2 pesquisas, se chave existe

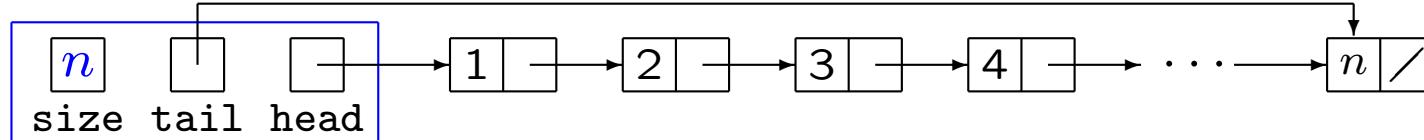
```
Integer value = map.get(key);  
if ( value == null )  
    value = this.computeValue(key);  
    map.put(key, value);
```

CORRETO

1 pesquisa, se chave existe

get em Listas Ligadas

Percorram uma lista ligada iterando-a



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

```
List<Integer> list = new LinkedList<>();
```

```
for ( int i = 0; i < list.size(); i++ ) {
    int elem = list.get(i);
    // PROCESS elem
}
```

ERRO

Complexidade $\Theta(n^2)$ †

```
for ( int elem : list ) {
    // PROCESS elem
}
```

CORRETO

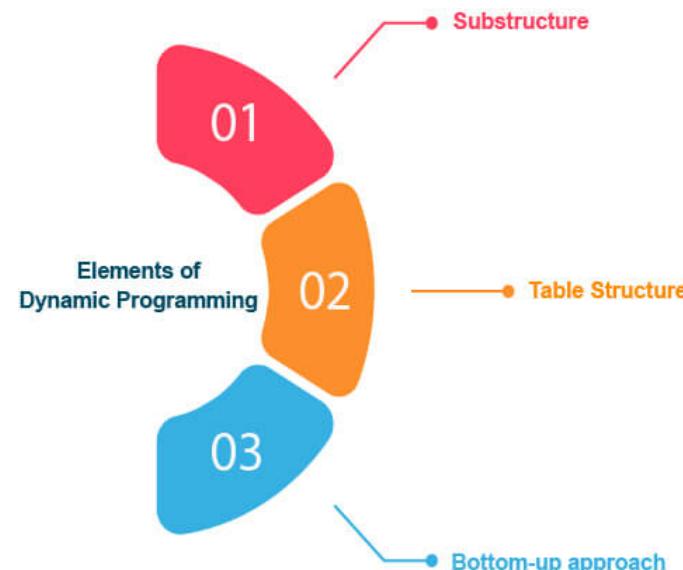
Complexidade $\Theta(n)$ †

† Assume-se que o custo de processar cada elemento da lista é constante.

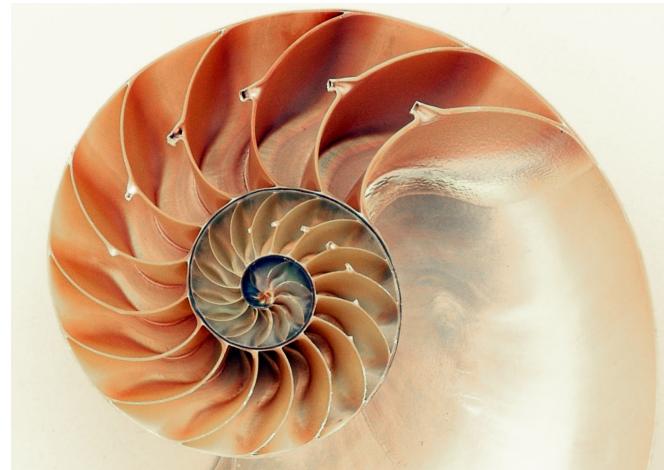
Capítulo I

Programação Dinâmica

(Dynamic Programming)



Problema dos Números de Fibonacci



Números de Fibonacci

Qual é o valor de $\mathcal{F}(7)$?

$$\mathcal{F}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathcal{F}(n - 1) + \mathcal{F}(n - 2) & n \geq 2 \end{cases}$$

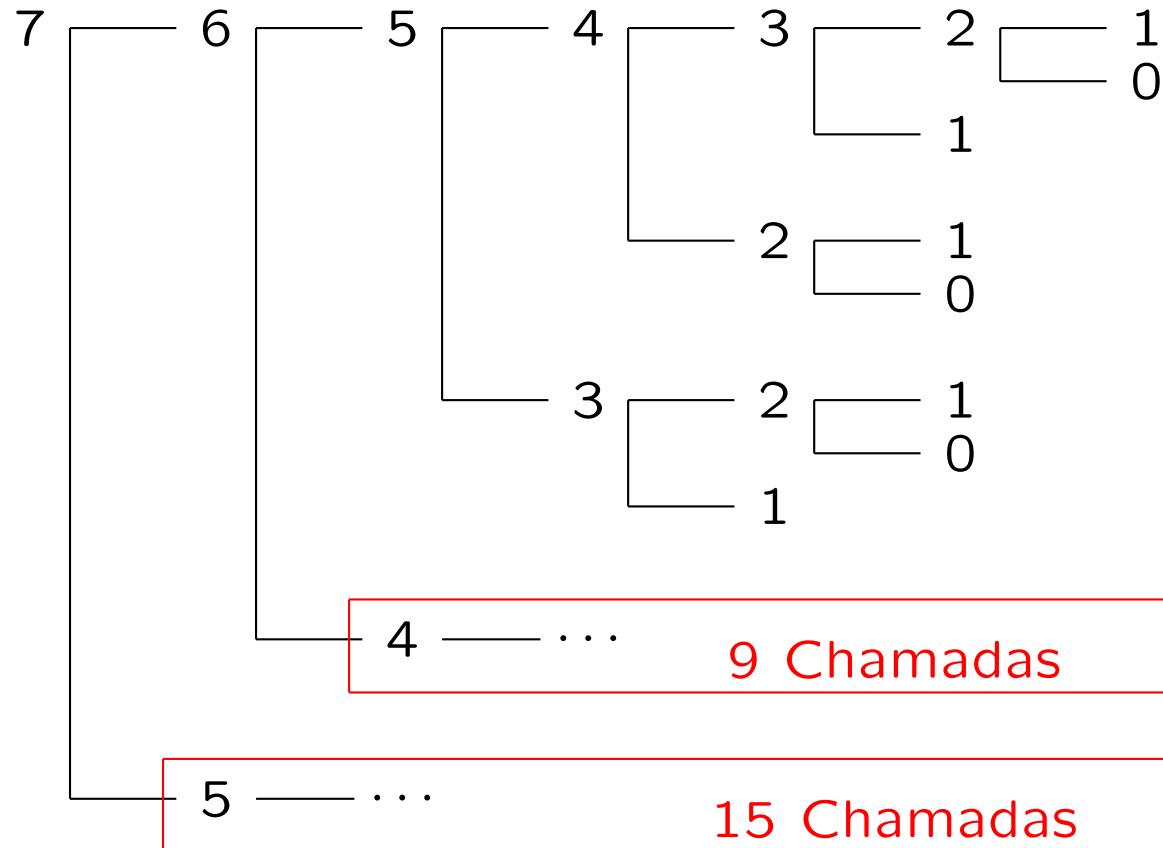
Solução Recursiva por Força Bruta

```
long fibRec( int n ) {  
    int res;  
    if ( n == 0 || n == 1 )  
        res = n;  
    else  
        res = fibRec(n - 1) + fibRec(n - 2);  
    return res;  
}
```

Qual é a complexidade temporal de $\text{fibRec}(n)$?

Quantas chamadas recursivas, $n\text{CR}(n)$, são efetuadas durante a execução de $\text{fibRec}(n)$?

Complexidade da Solução Força Bruta



Facto: $n\text{CR}(n)$ é $\Theta(\phi^n)$, onde $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ é o número de ouro.

Portanto, a complexidade temporal de `fibRec(n)` é exponencial.

Característica do Problema

O número de **sub-problemas distintos** a resolver é polinomial. Para calcular o valor de $\mathcal{F}(n)$ só precisamos dos valores de:

$$\mathcal{F}(n - 1), \mathcal{F}(n - 2), \dots, \mathcal{F}(1), \mathcal{F}(0)$$

A função `fibRec(n)` executa **chamadas recursivas repetidas**.

Técnicas de Desenho de Algoritmos Aplicáveis

- **Função-Memória** (*memoization*): algoritmo recursivo (*top-down*)
- **Programação Dinâmica**: algoritmo iterativo (*bottom-up*)

Em ambas, cada sub-problema é resolvido, no máximo, uma vez, guardando-se a solução numa “tabela”.

Solução com Função-Memória

```
long fibMem( int n ) {  
    long[] tab = new long[n + 1];  
    for ( int i = 0; i ≤ n; i++ )  
        tab[i] = -1;  
    return fibMem(tab, n);  
}
```

```
long fibMem( long[] tab, int n ) {  
    if ( tab[n] == -1 ) {  
        if ( n == 0 || n == 1 )  
            tab[n] = n;  
        else  
            tab[n] = fibMem(tab, n - 1) + fibMem(tab, n - 2);  
    }  
    return tab[n];  
}
```

Complexidade temporal:
 $\text{fibMem}(n) = \Theta(n)$

Complexidade espacial:
 $\text{fibMem}(n) = \Theta(n)$

Solução com Programação Dinâmica

Aplicação Direta da Técnica

```
long fibDP( int n ) {  
    long[] tab = new long[n + 1];  
    // Position 0: Base case.  
    tab[0] = 0;  
    // Position 1: Base case.  
    tab[1] = 1;  
    // Recursive case.  
    for ( int i = 2; i <= n; i++ )  
        tab[i] = tab[i - 1] + tab[i - 2];  
    return tab[n];  
}
```

Complexidade temporal:

$$\text{fibDP}(n) = \Theta(n)$$

Complexidade espacial:

$$\text{fibDP}(n) = \Theta(n)$$

tab	0	1	1	2	3	5	8	13
	0	1	2	3	4	5	6	7

Solução com Programação Dinâmica

Versão Otimizada

```
long fibIter( int n ) {  
    if ( n == 0 || n == 1 )  
        return n;  
  
    long penultimate;  
    long last = 0;  
    long current = 1;  
    for ( int i = 2; i ≤ n; i++ ) {  
        penultimate = last;  
        last = current;  
        current = penultimate + last;  
    }  
    return current;  
}
```

Complexidade temporal:

$$\text{fibIter}(n) = \Theta(n)$$

Complexidade espacial:

$$\text{fibIter}(n) = \Theta(1)$$

Problema dos Caixotes de Morangos

Lucro Máximo



Caixotes de Morangos — Lucro (1)

O dono de uma pequena cadeia de ($L \geq 1$) mercearias adquiriu ($C \geq 1$) caixotes de morangos e quer saber como deve **distribuir os caixotes pelas lojas** de forma a **maximizar o lucro** que pode obter.

Devido às características de cada loja (localização, capacidade de armazenamento, número médio de clientes, etc.), o lucro esperado com a venda dos morangos varia, não só de loja para loja, como também consoante o número de caixotes enviados para cada loja. Mas o dono sabe estimar o **lucro** obtido com o envio de um número de caixotes para uma determinada loja (para qualquer número de caixotes entre 1 e C e qualquer loja). Estas estimativas estão na tabela **Lucros**.

Por razões administrativas, cada caixote é indivisível (i.e., o seu conteúdo não pode ser repartido por várias lojas). Não é necessário enviar caixotes para todas as lojas.

Caixotes de Morangos — Lucro (2)

Assuma que há três lojas ($L = 3$), cinco caixotes de morangos ($C = 5$) e que a tabela **Lucros** tem os seguintes valores (em euros):

Lucros	Número de Caixotes					
	0	1	2	3	4	5
Loja 0	0.00	1.50	3.50	4.50	6.00	6.50
Loja 1	0.00	2.50	5.00	5.50	5.50	5.50
Loja 2	0.00	2.00	3.00	5.50	6.00	6.00

Se se enviassem **três** caixotes para a Loja 0, **zero** caixotes para a Loja 1 e **dois** caixotes para a Loja 2, o lucro seria:

$$4.50 + 0.00 + 3.00 = 7.50 \text{ euros.}$$

Qual é o lucro máximo?

Ideia da Abordagem Recursiva (1)

$\mathcal{L}(\{0, 1, 2\}, 5)$ é o **lucro máximo** com o envio para as lojas **0, 1, 2** de 5 caixotes:

- **(UM passo de cada vez)**
- **(TODAS as alternativas são consideradas)**
- **(Só queremos a MELHOR alternativa)**

Ideia da Abordagem Recursiva (2)

$\mathcal{L}(\{0, 1, 2\}, 5)$ é o **lucro máximo** com o envio para as lojas **0, 1, 2** de 5 caixotes:

- (**UM passo de cada vez**) Quantos caixotes vão para a última loja?
- (**TODAS as alternativas são consideradas**) 0, 1, 2, 3, 4 ou 5
- (**Só queremos a MELHOR alternativa**)
 - (0 caixotes para a loja 2) $\text{Lucros}[2][0] + \mathcal{L}(\{0, 1\}, 5)$
 - (1 caixotes para a loja 2) $\text{Lucros}[2][1] + \mathcal{L}(\{0, 1\}, 4)$
 - (2 caixotes para a loja 2) $\text{Lucros}[2][2] + \mathcal{L}(\{0, 1\}, 3)$
 - (3 caixotes para a loja 2) $\text{Lucros}[2][3] + \mathcal{L}(\{0, 1\}, 2)$
 - (4 caixotes para a loja 2) $\text{Lucros}[2][4] + \mathcal{L}(\{0, 1\}, 1)$
 - (5 caixotes para a loja 2) $\text{Lucros}[2][5] + \mathcal{L}(\{0, 1\}, 0)$

Ideia da Abordagem Recursiva (3)

$\mathcal{L}(\{0, 1, \dots, i\}, j)$ é o **lucro máximo** com o envio para as lojas $0, 1, \dots, i$ de j caixotes:

- (**UM passo de cada vez**) Quantos caixotes vão para a última loja?
- (**TODAS as alternativas são consideradas**) $0, 1, \dots, j - 1$ ou j
- (**Só queremos a MELHOR alternativa**)
 - (0 caixotes para a loja i) $\text{Lucros}[i][0] + \mathcal{L}(\{0, \dots, i - 1\}, j)$
 - (1 caixotes para a loja i) $\text{Lucros}[i][1] + \mathcal{L}(\{0, \dots, i - 1\}, j - 1)$
 - (2 caixotes para a loja i) $\text{Lucros}[i][2] + \mathcal{L}(\{0, \dots, i - 1\}, j - 2)$
 -
 - (j caixotes para a loja i) $\text{Lucros}[i][j] + \mathcal{L}(\{0, \dots, i - 1\}, 0)$

Resolução do Problema (1)

Identificação das lojas: $0, 1, \dots, L - 1$

Número de caixotes a distribuir: C

Lucro com o envio para a Loja i de j caixotes: **Lucros**[i][j]

(com $i = 0, \dots, L - 1$ e $j = 0, \dots, C$)

$\mathcal{L}(i, j)$ é o **lucro máximo com o envio para as lojas $0, 1, \dots, i$ de j caixotes**:

-
-
-

Resolução do Problema (2)

Identificação das lojas: $0, 1, \dots, L - 1$

Número de caixotes a distribuir: C

Lucro com o envio para a Loja i de j caixotes: $\text{Lucros}[i][j]$

(com $i = 0, \dots, L - 1$ e $j = 0, \dots, C$)

$\mathcal{L}(i, j)$ é o **lucro máximo com o envio para as lojas $0, 1, \dots, i$ de j caixotes**:

- **(0 caixotes)** Se $j = 0$, $\mathcal{L}(i, j) = 0$;
-
-

Resolução do Problema (3)

Identificação das lojas: $0, 1, \dots, L - 1$

Número de caixotes a distribuir: C

Lucro com o envio para a Loja i de j caixotes: $\text{Lucros}[i][j]$

(com $i = 0, \dots, L - 1$ e $j = 0, \dots, C$)

$\mathcal{L}(i, j)$ é o **lucro máximo com o envio para as lojas $0, 1, \dots, i$ de j caixotes**:

- **(0 caixotes)** Se $j = 0$, $\mathcal{L}(i, j) = 0$;
- **(1 só loja)** Se $i = 0$ e $j \geq 1$, $\mathcal{L}(i, j) = \text{Lucros}[i][j]$;
-

Resolução do Problema (4)

Identificação das lojas: $0, 1, \dots, L - 1$

Número de caixotes a distribuir: C

Lucro com o envio para a Loja i de j caixotes: $\text{Lucros}[i][j]$
(com $i = 0, \dots, L - 1$ e $j = 0, \dots, C$)

$\mathcal{L}(i, j)$ é o **lucro máximo com o envio para as lojas $0, 1, \dots, i$ de j caixotes**:

- (**0 caixotes**) Se $j = 0$, $\mathcal{L}(i, j) = 0$;
- (**1 só loja**) Se $i = 0$ e $j \geq 1$, $\mathcal{L}(i, j) = \text{Lucros}[i][j]$;
- (**Caso geral**) Se $i \geq 1$ e $j \geq 1$, enviam-se k caixotes para a Loja i , para algum $k = 0, 1, \dots, j$, e os restantes para as lojas $0, 1, \dots, i - 1$.

Portanto:

$$\mathcal{L}(i, j) = \max_{0 \leq k \leq j} \left(\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k) \right).$$

Resolução do Problema (5)

Qual é o valor de $\mathcal{L}(L - 1, C)$?

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \textcolor{green}{\text{Lucros}}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\textcolor{green}{\text{Lucros}}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

No caso do exemplo, qual é o valor de $\mathcal{L}(2, 5)$?

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

Programação Dinâmica da Função \mathcal{L} (1)

Passo 1: Obter memória para tabelar a função

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L}	0	1	2	3	4	5
0						
1						
2						

Programação Dinâmica da Função \mathcal{L} (2)

Passo 2: Tratar a(s) base(s) da recursividade
(Caso $j = 0$, primeira coluna)

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L}	0	1	2	3	4	5
0	0.0					
1	0.0					
2	0.0					

Programação Dinâmica da Função \mathcal{L} (3)

Passo 2: Tratar a(s) base(s) da recursividade
(Caso $i = 0$ e $j \geq 1$, restante primeira linha)

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L}	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0					
2	0.0					

Programação Dinâmica da Função \mathcal{L} (4)

Passo 3: Tratar o caso geral

(Caso $i \geq 1$ e $j \geq 1$, tabelação por linhas)

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L}	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	6.5	8.5	9.5
2	0.0					

Programação Dinâmica da Função \mathcal{L} (5)

Passo 3: Tratar o caso geral

(Caso $i \geq 1$ e $j \geq 1$, tabelação por linhas)

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Luc	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	5.5	5.5	5.5
2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L}	0	1	2	3	4	5
0	0.0	1.5	3.5	4.5	6.0	6.5
1	0.0	2.5	5.0	6.5	8.5	9.5
2	0.0	2.5	5.0	7.0	8.5	10.5

Programação Dinâmica da Função \mathcal{L} (6)

$$\begin{aligned}\mathcal{L}[2][4] &= \max(\frac{\text{Lucros}[2][0] + \mathcal{L}[1][4]}{\text{Lucros}[2][2] + \mathcal{L}[1][2]}, \frac{\text{Lucros}[2][1] + \mathcal{L}[1][3]}{\text{Lucros}[2][3] + \mathcal{L}[1][1]}, \\ &\quad \text{Lucros}[2][4] + \mathcal{L}[1][0]) \\ &= \max(\frac{0.00 + 8.50}{3.00 + 5.00}, \frac{2.00 + 6.50}{5.50 + 2.50}, \\ &\quad \frac{6.00 + 0.00}{}) \\ &= \max(\frac{8.50}{8.00}, \frac{8.50}{8.00}, \\ &\quad \frac{6.00}{})\end{aligned}$$

```
// The values in profit first column (0 crates) are ignored.  
  
double strawberriesP( double[][] profit ) {  
  
    int nShops = profit.length;  
  
    int nCols = profit[0].length;  
  
    double[][] maxProf = new double[nShops][nCols];  
  
    // Column 0 — 0 crates.  
  
    for ( int i = 0; i < nShops; i++ )  
        maxProf[i][0] = 0;  
  
    // Row 0 — Only one shop.  
  
    for ( int j = 1; j < nCols; j++ )  
        maxProf[0][j] = profit[0][j];  
  
    // Remaining cells, filled by rows.  
  
    .....  
}
```

```
// Remaining cells, filled by rows.

for ( int i = 1; i < nShops i++ )

    for ( int j = 1; j < nCols; j++ ) {

        maxProf[i][j] = maxProf[i - 1][j];

        for ( int k = 1; k <= j; k++ ) {

            double value = profit[i][k] + maxProf[i - 1][j - k];

            if ( value > maxProf[i][j] )

                maxProf[i][j] = value;

        }

    }

return maxProf[nShops - 1][nCols - 1];

}
```

Complexidade Temp. de strawberriesP

($L = \text{profit.length}$, $C = \text{profit}[0].length - 1$)

Primeiro Ciclo $\Theta(L)$

Segundo Ciclo $\Theta(C)$

Terceiro Ciclo

$$\sum_{i=1}^{L-1} \sum_{j=1}^C \sum_{k=1}^j 1 = \sum_{i=1}^{L-1} \sum_{j=1}^C j = \sum_{i=1}^{L-1} \frac{C(C+1)}{2}$$

$$= (L-1) \frac{C(C+1)}{2} = \Theta(L C^2)$$

Total $\Theta(L C^2)$

Complexidades

$(L = \text{profit.length}, C = \text{profit}[0].length - 1)$

Temporal

strawberriesP

$\Theta(L C^2)$

Espacial

strawberriesP

$\Theta(L C)$

Matriz maxProf tem L linhas e $C + 1$ colunas.

(É fácil diminuir a complexidade espacial?)

Problema dos Caixotes de Morangos

Distribuição Ótima



Caixotes de Morangos — Distr.Ot. (1)

O dono de uma pequena cadeia de ($L \geq 1$) mercearias adquiriu ($C \geq 1$) caixotes de morangos e quer saber como deve **distribuir os caixotes pelas lojas** de forma a **maximizar o lucro** que pode obter.

Devido às características de cada loja (localização, capacidade de armazenamento, número médio de clientes, etc.), o lucro esperado com a venda dos morangos varia, não só de loja para loja, como também consoante o número de caixotes enviados para cada loja. Mas o dono sabe estimar o **lucro** obtido com o envio de um número de caixotes para uma determinada loja (para qualquer número de caixotes entre 1 e C e qualquer loja). Estas estimativas estão na tabela **Lucros**.

Por razões administrativas, cada caixote é indivisível (i.e., o seu conteúdo não pode ser repartido por várias lojas). Não é necessário enviar caixotes para todas as lojas.

Caixotes de Morangos — Distr.Ot. (2)

Assuma que há três lojas ($L = 3$), cinco caixotes de morangos ($C = 5$) e que a tabela **Lucros** tem os seguintes valores (em euros):

Lucros	Número de Caixotes					
	0	1	2	3	4	5
Loja 0	0.00	1.50	3.50	4.50	6.00	6.50
Loja 1	0.00	2.50	5.00	5.50	5.50	5.50
Loja 2	0.00	2.00	3.00	5.50	6.00	6.00

Se se enviassem **três** caixotes para a Loja 0, **zero** caixotes para a Loja 1 e **dois** caixotes para a Loja 2, o lucro seria:

$$4.50 + 0.00 + 3.00 = 7.50 \text{ euros.}$$

Como encontrar uma distribuição ótima (i.e. que maximize o lucro)?

Resolução do Problema

Identificação das lojas: $0, 1, \dots, L - 1$

Número de caixotes a distribuir: C

Lucro com o envio para a Loja i de j caixotes: $\text{Lucros}[i][j]$
(com $i = 0, \dots, L - 1$ e $j = 0, \dots, C$)

$\mathcal{L}(i, j)$ é o **lucro máximo com o envio para as lojas $0, 1, \dots, i$ de j caixotes**:

- **(0 caixotes)** Se $j = 0$, $\mathcal{L}(i, j) = 0$;
- **(1 só loja)** Se $i = 0$ e $j \geq 1$, $\mathcal{L}(i, j) = \text{Lucros}[i][j]$;
- **(Caso geral)** Se $i \geq 1$ e $j \geq 1$, enviam-se k caixotes para a Loja i , para algum $k = 0, 1, \dots, j$, e os restantes para as lojas $0, 1, \dots, i - 1$.

Portanto:

$$\mathcal{L}(i, j) = \max_{0 \leq k \leq j} \left(\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k) \right).$$

Programação Dinâmica da Função \mathcal{L} (1)

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

Lucros 0 1 2 3 4 5 (Número de Caixotes)

Loja 0	0.0	1.5	3.5	4.5	6.0	6.5
Loja 1	0.0	2.5	5.0	5.5	5.5	5.5
Loja 2	0.0	2.0	3.0	5.5	6.0	6.0

\mathcal{L} 0 1 2 3 4 5 (Número de Caixotes)

lojas 0	0.0	1.5	3.5	4.5	6.0	6.5
lojas 0,1	0.0	2.5	5.0	6.5	8.5	9.5
lojas 0,1,2	0.0	2.5	5.0	7.0	8.5	10.5

Programação Dinâmica da Função \mathcal{L} (2)

$$\begin{aligned}\mathcal{L}[2][4] &= \max(\frac{\text{Lucros}[2][0] + \mathcal{L}[1][4]}{\text{Lucros}[2][2] + \mathcal{L}[1][2]}, \frac{\text{Lucros}[2][1] + \mathcal{L}[1][3]}{\text{Lucros}[2][3] + \mathcal{L}[1][1]}, \\ &\quad \text{Lucros}[2][4] + \mathcal{L}[1][0]) \\ &= \max(\frac{0.00 + 8.50}{3.00 + 5.00}, \frac{2.00 + 6.50}{5.50 + 2.50}, \\ &\quad \frac{6.00 + 0.00}{}) \\ &= \max(\frac{8.50}{8.00}, \frac{8.50}{8.00}, \\ &\quad \frac{6.00}{})\end{aligned}$$

O que deve ser guardado
para se poder construir uma distribuição ótima?

$$\begin{aligned}
 \mathcal{L}[2][4] &= \max(\frac{\text{Lucros}[2][0] + \mathcal{L}[1][4]}{\text{Lucros}[2][2] + \mathcal{L}[1][2]}, \frac{\text{Lucros}[2][1] + \mathcal{L}[1][3]}{\text{Lucros}[2][3] + \mathcal{L}[1][1]}, \\
 &\quad \frac{\text{Lucros}[2][4] + \mathcal{L}[1][0]}{}) \\
 &= \max(\frac{0.00 + 8.50}{3.00 + 5.00}, \frac{2.00 + 6.50}{5.50 + 2.50}, \\
 &\quad \frac{6.00 + 0.00}{}) \\
 &= \max(\frac{8.50}{8.00}, \frac{8.50}{8.00}, \\
 &\quad \frac{6.00}{})
 \end{aligned}$$

$$\mathcal{D}[2][4] = 0$$

$\mathcal{D}[2][4]$ é o número de caixotes a enviar para a **Loja 2**,
quando há **4** caixotes para distribuir pelas **lojas 0, 1, 2**.
(É o valor do “ k que maximiza”.)

O que deve ser guardado para se poder construir uma distribuição ótima?

$$\mathcal{L}(i, j) = \begin{cases} 0 & j = 0 \\ \text{Lucros}[i][j] & i = 0 \text{ e } j \geq 1 \\ \max_{0 \leq k \leq j} (\text{Lucros}[i][k] + \mathcal{L}(i - 1, j - k)) & i \geq 1 \text{ e } j \geq 1 \end{cases}$$

$\mathcal{D}(i, j)$ é o número de caixotes a enviar para a Loja i ,
quando há j caixotes para distribuir pelas lojas $0, 1, \dots, i$.

- **(0 caixotes)** Se $j = 0$, $\mathcal{D}(i, j) = 0$;
- **(1 só loja)** Se $i = 0$ e $j \geq 1$, $\mathcal{D}(i, j) = j$;
- **(Caso geral)** Se $i \geq 1$ e $j \geq 1$, $\mathcal{D}(i, j) = k'$, onde k' é qualquer valor em $\{0, 1, \dots, j\}$ tal que $\text{Lucros}[i][k'] + \mathcal{L}(i - 1, j - k') = \mathcal{L}(i, j)$.

O que deve ser guardado
para se poder construir uma distribuição ótima?

\mathcal{L}	0	1	2	3	4	5	(Número de Caixotes)
lojas 0	0.0	1.5	3.5	4.5	6.0	6.5	
lojas 0,1	0.0	2.5	5.0	6.5	8.5	9.5	
lojas 0,1,2	0.0	2.5	5.0	7.0	8.5	10.5	

\mathcal{D}	0	1	2	3	4	5	(Número de Caixotes)
lojas 0	0	1	2	3	4	5	
lojas 0,1	0	1	2	2	2	2	
lojas 0,1,2	0	0	0	1	0	1	

Construção da Distribuição Ótima

\mathcal{D}	0	1	2	3	4	5	(Número de Caixotes)
lojas 0	0	1	2	3	4	5	
lojas 0,1	0	1	2	2	2	2	
lojas 0,1,2	0	0	0	1	0	1	

Problema	\mathcal{D}	Distribuição				
		<table border="1"><tr><td></td><td></td><td></td></tr></table>				
(2, 5)	$\mathcal{D}[2][5] = 1$	<table border="1"><tr><td></td><td></td><td>1</td></tr></table>			1	Restam $5 - 1 = 4$ caixotes
		1				
(1, 4)	$\mathcal{D}[1][4] = 2$	<table border="1"><tr><td></td><td>2</td><td>1</td></tr></table>		2	1	Restam $4 - 2 = 2$ caixotes
	2	1				
(0, 2)	$\mathcal{D}[0][2] = 2$	<table border="1"><tr><td>2</td><td>2</td><td>1</td></tr></table>	2	2	1	Restam $2 - 2 = 0$ caixotes
2	2	1				
(-1, 0)						

```
// The values in profit first column (0 crates) are ignored.  
Pair<Double,int[]> strawberriesD( double[][] profit ) {  
    int nShops = profit.length;  
    int nCols = profit[0].length;  
    double[][] maxProfTab = new double[nShops][nCols];  
    int[][] distrTab = new int[nShops][nCols];  
    compMaxProfit(profit, maxProfTab, distrTab);  
  
    double maxProfit = maxProfTab[nShops - 1][nCols - 1];  
  
    int[] optDistr = new int[nShops];  
    compDistr(distrTab, nShops - 1, nCols - 1, optDistr);  
  
    return new PairClass<Double,int[]>(maxProfit, optDistr);  
}
```

```

void compMaxProfit( double[][] profit, double[][] maxProfit,
int[][] distr ) {

    int nShops = profit.length;
    int nCols = profit[0].length;
    // Column 0 — 0 crates.
    for ( int i = 0; i < nShops; i++ ) {
        maxProfit[i][0] = 0;
        distr[i][0] = 0;
    }
    // Row 0 — Only one shop.
    for ( int j = 1; j < nCols; j++ ) {
        maxProfit[0][j] = profit[0][j];
        distr[0][j] = j;
    }
    // Remaining cells, filled by rows.
    .....

```

```
// Remaining cells, filled by rows.

for ( int i = 1; i < nShops i++ )

    for ( int j = 1; j < nCols; j++ ) {

        maxProfit[i][j] = maxProfit[i - 1][j];

        distr[i][j] = 0;

        for ( int k = 1; k <= j; k++ ) {

            double value = profit[i][k] + maxProfit[i - 1][j - k];

            if ( value > maxProfit[i][j] ) {

                maxProfit[i][j] = value;

                distr[i][j] = k;

            }

        }

    }

}
```

Complexidade Temp. de `compMaxProfit`

($L = \text{profit.length}$, $C = \text{profit[0].length} - 1$)

Primeiro Ciclo $\Theta(L)$

Segundo Ciclo $\Theta(C)$

Terceiro Ciclo

$$\sum_{i=1}^{L-1} \sum_{j=1}^C \sum_{k=1}^j 1 = \sum_{i=1}^{L-1} \sum_{j=1}^C j = \sum_{i=1}^{L-1} \frac{C(C+1)}{2}$$

$$= (L-1) \frac{C(C+1)}{2} = \Theta(L C^2)$$

Total $\Theta(L C^2)$

```
void compDistr( int[][] distrTab, int shop, int crates, int[] optDistr ) {  
    if ( shop >= 0 ) {  
        int cratesToShop = distrTab[shop][crates];  
        optDistr[shop] = cratesToShop;  
        compDistr(distrTab, shop - 1, crates - cratesToShop, optDistr);  
    }  
}
```

```
void compDistr( int[][] distrTab, int shop, int crates, int[] optDistr ) {  
    if ( shop >= 0 ) {  
        int cratesToShop = distrTab[shop][crates];  
        optDistr[shop] = cratesToShop;  
        compDistr(distrTab, shop - 1, crates - cratesToShop, optDistr);  
    }  
}
```

```
void compDistr( int[][] distrTab, int shop, int crates, int[] optDistr ) {  
    while ( shop >= 0 ) {  
        int cratesToShop = distrTab[shop][crates];  
        optDistr[shop] = cratesToShop;  
        shop -- ;  
        crates -= cratesToShop;  
    }  
}
```

Complexidades

$(L = \text{profit.length}, C = \text{profit}[0].length - 1)$

Temporal

`compMaxProfit` $\Theta(L C^2)$

`compDistr` (Chamada inicial $(_, L - 1, _, _)$) $\Theta(L)$

$\Theta(L)$ chamadas/passos, cada uma/um $\Theta(1)$.

`strawberriesD` $\Theta(L C^2)$

Espacial

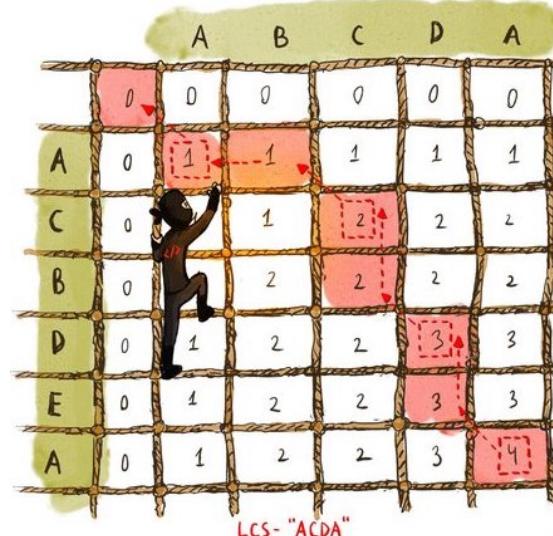
`strawberriesD` $\Theta(L C)$

Matriz `maxProfTab` tem L linhas e $C + 1$ colunas;

Matriz `distrTab` tem L linhas e $C + 1$ colunas.

(É fácil diminuir a complexidade espacial?)

Problema da Maior Subsequência Comum (Longest Common Subsequence)



Maior Subsequência Comum

b d c a b a

Subsequência: a b a

a b c b d a b

Comprimento: 3

b d c a b a

Subsequência: b c a b

a b c b d a b

Comprimento: 4

Dadas duas sequências:

$$X = (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1)$$

$$Y = (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1),$$

calcular o comprimento das maiores subsequências comuns a X e Y .

Ideia da Abordagem Recursiva (1)

$\mathcal{C}(X,Y)$ é o **comprimento** das maiores subsequências comuns a X e Y .

Qual é a forma $S(X,Y)$ de uma maior subsequência comum a X e Y ?

- Se X e Y terminam com o mesmo elemento:

$$X = \underbrace{\text{b d c a b}}_{X'} \text{ c}$$

$$Y = \underbrace{\text{a b c b d a}}_{Y'} \text{ c}$$

Ideia da Abordagem Recursiva (2)

$\mathcal{C}(X,Y)$ é o **comprimento** das maiores subsequências comuns a X e Y .

Qual é a forma $\mathcal{S}(X,Y)$ de uma maior subsequência comum a X e Y ?

- Se X e Y terminam com o mesmo elemento:

$$X = \underbrace{\text{b d c a b}}_{X'} \text{ c}$$

$$Y = \underbrace{\text{a b c b d a}}_{Y'} \text{ c}$$

$$\mathcal{S}(X,Y) = \mathcal{S}(X',Y') \text{ c}$$

$$\mathcal{C}(X,Y) = \mathcal{C}(X',Y') + 1$$

Ideia da Abordagem Recursiva (3)

$\mathcal{C}(X,Y)$ é o **comprimento** das maiores subsequências comuns a X e Y .

Qual é a forma $S(X,Y)$ de uma maior subsequência comum a X e Y ?

- Se X e Y terminam com elementos diferentes:

$$X = \underbrace{\text{b d c a b}}_{X'} \text{ a}$$

$$Y = \underbrace{\text{a b c b d a}}_{Y'} \text{ b}$$

Ideia da Abordagem Recursiva (4)

$\mathcal{C}(X,Y)$ é o **comprimento** das maiores subsequências comuns a X e Y .

Qual é a forma $\mathcal{S}(X,Y)$ de uma maior subsequência comum a X e Y ?

- Se X e Y terminam com elementos diferentes:

$$X = \underbrace{\mathbf{b} \mathbf{d} \mathbf{c} \mathbf{a} \mathbf{b}}_{X'} \mathbf{a}$$

$$Y = \underbrace{\mathbf{a} \mathbf{b} \mathbf{c} \mathbf{b} \mathbf{d} \mathbf{a}}_{Y'} \mathbf{b}$$

$$\mathcal{S}(X,Y) = \mathcal{S}(X,Y') \text{ ou } \mathcal{S}(X',Y)$$

$$\mathcal{C}(X,Y) = \max(\mathcal{C}(X,Y'), \mathcal{C}(X',Y))$$

Resolução do Problema (1)

$$\begin{aligned} X &= (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1) \\ Y &= (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1) \end{aligned}$$

$\mathcal{C}(i, j)$ é o **comprimento** das maiores subsequências comuns a X_i e Y_j :

$$\begin{aligned} X_i &= (x_1 \ x_2 \ x_3 \ \cdots \ x_i) \quad (i \geq 0) \\ Y_j &= (y_1 \ y_2 \ y_3 \ \cdots \ y_j) \quad (j \geq 0) \end{aligned}$$

Resolução do Problema (2)

$$X = (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1)$$

$$Y = (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1)$$

$\mathcal{C}(i, j)$ é o **comprimento** das maiores subsequências comuns a X_i

e Y_j :

$$X_i = (x_1 \ x_2 \ x_3 \ \cdots \ x_i) \quad (i \geq 0)$$

$$Y_j = (y_1 \ y_2 \ y_3 \ \cdots \ y_j) \quad (j \geq 0)$$

- Se $i = 0$ ou $j = 0$,
- Se $i \geq 1$, $j \geq 1$ e $x_i = y_j$,
- Se $i \geq 1$, $j \geq 1$ e $x_i \neq y_j$,

Resolução do Problema (3)

$$\begin{aligned} X &= (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1) \\ Y &= (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1) \end{aligned}$$

$\mathcal{C}(i, j)$ é o **comprimento** das maiores subsequências comuns a X_i

e Y_j :

$$\begin{aligned} X_i &= (x_1 \ x_2 \ x_3 \ \cdots \ x_i) \quad (i \geq 0) \\ Y_j &= (y_1 \ y_2 \ y_3 \ \cdots \ y_j) \quad (j \geq 0) \end{aligned}$$

- Se $i = 0$ ou $j = 0$, $\mathcal{C}(i, j) = 0$;
- Se $i \geq 1$, $j \geq 1$ e $x_i = y_j$,
- Se $i \geq 1$, $j \geq 1$ e $x_i \neq y_j$,

Resolução do Problema (4)

$$\begin{aligned} X &= (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1) \\ Y &= (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1) \end{aligned}$$

$\mathcal{C}(i, j)$ é o **comprimento** das maiores subsequências comuns a X_i e Y_j :

$$\begin{aligned} X_i &= (x_1 \ x_2 \ x_3 \ \cdots \ x_i) \quad (i \geq 0) \\ Y_j &= (y_1 \ y_2 \ y_3 \ \cdots \ y_j) \quad (j \geq 0) \end{aligned}$$

- Se $i = 0$ ou $j = 0$, $\mathcal{C}(i, j) = 0$;
- Se $i \geq 1$, $j \geq 1$ e $x_i = y_j$, $\mathcal{C}(i, j) = \mathcal{C}(i - 1, j - 1) + 1$;
- Se $i \geq 1$, $j \geq 1$ e $x_i \neq y_j$,

Resolução do Problema (5)

$$\begin{aligned} X &= (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1) \\ Y &= (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1) \end{aligned}$$

$\mathcal{C}(i, j)$ é o **comprimento** das maiores subsequências comuns a X_i e Y_j :

$$\begin{aligned} X_i &= (x_1 \ x_2 \ x_3 \ \cdots \ x_i) \quad (i \geq 0) \\ Y_j &= (y_1 \ y_2 \ y_3 \ \cdots \ y_j) \quad (j \geq 0) \end{aligned}$$

- Se $i = 0$ ou $j = 0$, $\mathcal{C}(i, j) = 0$;
- Se $i \geq 1$, $j \geq 1$ e $x_i = y_j$, $\mathcal{C}(i, j) = \mathcal{C}(i - 1, j - 1) + 1$;
- Se $i \geq 1$, $j \geq 1$ e $x_i \neq y_j$, $\mathcal{C}(i, j) = \max(\mathcal{C}(i - 1, j), \mathcal{C}(i, j - 1))$.

Resolução do Problema (6)

$$X = (x_1 \ x_2 \ x_3 \ \cdots \ x_m) \quad (m \geq 1)$$

$$Y = (y_1 \ y_2 \ y_3 \ \cdots \ y_n) \quad (n \geq 1)$$

Qual é o valor de $\mathcal{C}(m, n)$?

$$\mathcal{C}(i, j) = \begin{cases} 0 & i = 0 \text{ ou } j = 0 \\ 1 + \mathcal{C}(i - 1, j - 1) & i \geq 1, j \geq 1 \text{ e } x_i = y_j \\ \max(\mathcal{C}(i - 1, j), \mathcal{C}(i, j - 1)) & i \geq 1, j \geq 1 \text{ e } x_i \neq y_j \end{cases}$$

No caso do exemplo, qual é o valor de $\mathcal{C}(6, 7)$?

$$X = (\textcolor{blue}{b} \ d \ c \ a \ \textcolor{blue}{b} \ a)$$

$$Y = (\textcolor{blue}{a} \ b \ c \ b \ d \ a \ b)$$

Programação Dinâmica da Função \mathcal{C}

$$\mathcal{C}(i, j) = \begin{cases} 0 & i = 0 \text{ ou } j = 0 \\ 1 + \mathcal{C}(i - 1, j - 1) & i \geq 1, j \geq 1 \text{ e } x_i = y_j \\ \max(\mathcal{C}(i - 1, j), \mathcal{C}(i, j - 1)) & i \geq 1, j \geq 1 \text{ e } x_i \neq y_j \end{cases}$$

		a	b	c	b	d	a	b
\mathcal{C}	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
b	1	0	0	1	1	1	1	1
d	2	0	0	1	1	1	2	2
c	3	0	0	1	2	2	2	2
a	4	0	1	1	2	2	3	3
b	5	0	1	2	2	3	3	4
a	6	0	1	2	2	3	3	4

```
// The values in seqX[0] and seqY[0] are ignored.  
  
int LCS( char[] seqX, char[] seqY ) {  
    int[][] maxLength = new int[seqX.length][seqY.length];  
    // Row 0 — seqX is empty.  
    for ( int j = 0; j < seqY.length; j++ )  
        maxLength[0][j] = 0;  
    // Column 0 — seqY is empty.  
    for ( int i = 1; i < seqX.length; i++ )  
        maxLength[i][0] = 0;  
    // Remaining cells, filled by rows.  
    .....  
}
```

```
// Remaining cells, filled by rows.

for ( int i = 1; i < seqX.length; i++ )

    for ( int j = 1; j < seqY.length; j++ )

        if ( seqX[i] == seqY[j] )

            maxLength[i][j] = 1 + maxLength[i - 1][j - 1];

        else if ( maxLength[i - 1][j] >= maxLength[i][j - 1] )

            maxLength[i][j] = maxLength[i - 1][j];

        else

            maxLength[i][j] = maxLength[i][j - 1];

return maxLength[seqX.length - 1][seqY.length - 1];

}
```

Complexidades

$(m = \text{seqX.length} - 1, n = \text{seqY.length} - 1)$

Temporal

LCS

$\Theta(m n)$

Primeiro ciclo: $\Theta(n)$

Segundo ciclo: $\Theta(m)$

Terceiro ciclo: $\Theta(m n)$

Espacial

LCS

$\Theta(m n)$

Matrix maxLength tem $m + 1$ linhas e $n + 1$ colunas.

(É fácil diminuir a complexidade espacial?)

Problema da Multiplicação de uma Cadeia de Matrizes (Matrix-Chain Multiplication)



Multiplicação de Duas Matrizes

$$\begin{bmatrix} a_{i1} & a_{i2} & \cdots & a_{ik} \end{bmatrix} \begin{bmatrix} b_{1j} \\ b_{2j} \\ \vdots \\ \vdots \\ b_{kj} \end{bmatrix} = \begin{bmatrix} c_{ij} \end{bmatrix}$$

$m \times k$ $k \times n$ $m \times n$

$$c_{ij} = \sum_{h=1}^k a_{ih} b_{hj}$$

Multiplicação2Matrizes(m, k, n) = $\Theta(m \times n \times k)$

Multiplicação de Cadeia de Matrizes

$$\begin{array}{cccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 \times 50 & & 50 \times 100 & & 100 \times 200 & & 200 \times 30 \end{array}$$

$$((M_1 \times M_2) \times M_3) \times M_4 \longrightarrow 62\,000 \text{ produtos}$$

$$(M_1 \times (M_2 \times M_3)) \times M_4 \longrightarrow 1\,032\,000 \text{ produtos}$$

$$(M_1 \times M_2) \times (M_3 \times M_4) \longrightarrow 616\,000 \text{ produtos}$$

$$M_1 \times ((M_2 \times M_3) \times M_4) \longrightarrow 1\,303\,000 \text{ produtos}$$

$$M_1 \times (M_2 \times (M_3 \times M_4)) \longrightarrow 753\,000 \text{ produtos}$$

Dadas as dimensões das matrizes:

$$\begin{array}{ccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

qual é o número mínimo de produtos para calcular $M_1 \times \cdots \times M_n$?

Ideia da Abordagem Recursiva (1)

$$\begin{array}{ccccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 & & 50 & & 100 & & 200 & & 30 \end{array}$$

$\mathcal{P}((1, 2, 3, 4))$ é o **número mínimo de produtos** para calcular (a matriz)

$M_1 \times M_2 \times M_3 \times M_4$:

- (**UM passo de cada vez**)
- (**TODAS as alternativas são consideradas**)
- (**Só queremos a MELHOR alternativa**)

Ideia da Abordagem Recursiva (2)

$$\begin{array}{ccccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 & & 50 & & 100 & & 200 & & 30 \end{array}$$

$\mathcal{P}((1, 2, 3, 4))$ é o **número mínimo de produtos** para calcular (a matriz) $M_1 \times M_2 \times M_3 \times M_4$:

- (**UM passo de cada vez**) Qual é a última multiplicação efetuada?

- (**TODAS as alternativas são consideradas**)

$$\begin{array}{ll} (M_1) \times (M_2 \times M_3 \times M_4) & (M_1 \times M_2) \times (M_3 \times M_4) \\ (M_1 \times M_2 \times M_3) \times (M_4) & \end{array}$$

- (**Só queremos a MELHOR alternativa**)

$$(M_1 \text{ é a última da esquerda}) \quad \mathcal{P}((1)) + \mathcal{P}((2, 3, 4)) + 2 * 50 * 30$$

$$(M_2 \text{ é a última da esquerda}) \quad \mathcal{P}((1, 2)) + \mathcal{P}((3, 4)) + 2 * 100 * 30$$

$$(M_3 \text{ é a última da esquerda}) \quad \mathcal{P}((1, 2, 3)) + \mathcal{P}((4)) + 2 * 200 * 30$$

Ideia da Abordagem Recursiva (3)

$$\begin{array}{cccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

$\mathcal{P}(i, j)$ é o número mínimo de produtos para calcular $M_i \times \cdots \times M_j$:

- (**UM passo de cada vez**) Qual é a última multiplicação efetuada?
- (**TODAS as alternativas são consideradas**)

$$\begin{array}{ll} (M_i) \times (M_{i+1} \times \cdots \times M_j) & (M_i \times M_{i+1}) \times (M_{i+2} \times \cdots \times M_j) \\ \dots \dots \dots \dots \dots \dots \dots & (M_i \times \cdots \times M_{j-1}) \times (M_j) \end{array}$$

- (**Só queremos a MELHOR alternativa**)

$$\begin{array}{ll} (M_i \text{ é a última da esq.}) & \mathcal{P}(i, i) + \mathcal{P}(i+1, j) + d_{i-1}d_id_j \\ (M_{i+1} \text{ é a última da esq.}) & \mathcal{P}(i, i+1) + \mathcal{P}(i+2, j) + d_{i-1}d_{i+1}d_j \\ \dots \dots \dots \dots \dots \dots \dots & \\ (M_{j-1} \text{ é a última da esq.}) & \mathcal{P}(i, j-1) + \mathcal{P}(j, j) + d_{i-1}d_{j-1}d_j \end{array}$$

Resolução do Problema (1)

$$\begin{array}{ccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

$\mathcal{P}(i, j)$ é o **número mínimo de produtos para calcular** $M_i \times \cdots \times M_j$
(com $i \leq j$):

Resolução do Problema (2)

$$\begin{array}{cccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

$\mathcal{P}(i, j)$ é o **número mínimo de produtos para calcular** $M_i \times \cdots \times M_j$ (com $i \leq j$):

- Se $i = j$,
- Se $i < j$,

Resolução do Problema (3)

$$\begin{array}{ccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

$\mathcal{P}(i, j)$ é o **número mínimo de produtos para calcular** $M_i \times \cdots \times M_j$ (com $i \leq j$):

- Se $i = j$, $\mathcal{P}(i, j) = \mathcal{P}(i, i) = 0$;
- Se $i < j$,

Resolução do Problema (4)

$$\begin{array}{ccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

$\mathcal{P}(i, j)$ é o **número mínimo de produtos para calcular** $M_i \times \cdots \times M_j$ (com $i \leq j$):

- Se $i = j$, $\mathcal{P}(i, j) = \mathcal{P}(i, i) = 0$;
- Se $i < j$, a última operação é $(M_i \times \cdots \times M_k) \times (M_{k+1} \times \cdots \times M_j)$, para algum $k = i, i+1, \dots, j-1$, efetuando-se $d_{i-1}d_jd_k$ produtos nessa última operação.

Portanto:

$$\mathcal{P}(i, j) = \min_{i \leq k < j} \left(\mathcal{P}(i, k) + \mathcal{P}(k+1, j) + d_{i-1}d_jd_k \right)$$

Resolução do Problema (5)

$$\begin{array}{cccccccccc} M_1 & \times & M_2 & \times & M_3 & \times & \cdots & \times & M_n \\ d_0 & & d_1 & & d_2 & & d_3 & \cdots & d_{n-1} & & d_n \end{array}$$

Qual é o valor de $\mathcal{P}(1, n)$?

$$\mathcal{P}(i, j) = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} (\mathcal{P}(i, k) + \mathcal{P}(k + 1, j) + d_{i-1}d_jd_k) & i < j \end{cases}$$

No caso do exemplo, qual é o valor de $\mathcal{P}(1, 4)$?

$$\begin{array}{cccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 & & 50 & & 100 & & 200 & & 30 \end{array}$$

Programação Dinâmica da Função \mathcal{P} (1)

$$\mathcal{P}(i, j) = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} (\mathcal{P}(i, k) + \mathcal{P}(k + 1, j) + d_{i-1}d_jd_k) & i < j \end{cases}$$

$$\begin{array}{ccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 & & 50 & & 100 & & 200 & & 30 \end{array}$$

$$\begin{aligned} \mathcal{P}[1][3] &= \min(\mathcal{P}[1][1] + \mathcal{P}[2][3] + 2 \times 200 \times 50, \\ &\quad \underline{\mathcal{P}[1][2] + \mathcal{P}[3][3] + 2 \times 200 \times 100}) \\ &= \min(1\,020\,000, \underline{50\,000}) \end{aligned}$$

Programação Dinâmica da Função \mathcal{P} (2)

$$\mathcal{P}(i, j) = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} (\mathcal{P}(i, k) + \mathcal{P}(k + 1, j) + d_{i-1}d_jd_k) & i < j \end{cases}$$

$$\begin{array}{ccccc} M_1 & \times & M_2 & \times & M_3 & \times & M_4 \\ 2 & & 50 & & 100 & & 200 & & 30 \end{array}$$

$\mathcal{P}_{11} = 0$	$\mathcal{P}_{22} = 0$	$\mathcal{P}_{33} = 0$	$\mathcal{P}_{44} = 0$
$\mathcal{P}_{12} = 10\,000$	$\mathcal{P}_{23} = 1\,000\,000$	$\mathcal{P}_{34} = 600\,000$	
$\mathcal{P}_{13} = 50\,000$	$\mathcal{P}_{24} = 750\,000$		
$\mathcal{P}_{14} = 62\,000$			

```
// Matrix i has dims[i – 1] rows and dims[i] columns.  
long matrixChainMult( int[] dims ) {  
    long[][] minProd = new long[dims.length][dims.length];  
    // Base case: 1 matrix.  
    for ( int i = 1; i < dims.length; i++ )  
        minProd[i][i] = 0;  
    // Recursive case: d is the difference between the indices.  
    for ( int d = 1; d < dims.length – 1; d++ )  
        // i is the left index.  
        for ( int i = 1; i < dims.length – d; i++ ) {  
            // j is the right index.  
            int j = i + d;  
            .....  
        }
```

```

// Recursive case: d is the difference between the indices.

for ( int d = 1; d < dims.length - 1; d++ ) {
    // i is the left index.

    for ( int i = 1; i < dims.length - d; i++ ) {
        int j = i + d; // j is the right index
        int fixedFacts = dims[i - 1] * dims[j];
        minProd[i][j] = minProd[i + 1][j] + fixedFacts * dims[i];
        for ( int k = i + 1; k < j; k++ ) {
            long value = minProd[i][k] + minProd[k + 1][j] +
                fixedFacts * dims[k];
            if ( value < minProd[i][j] )
                minProd[i][j] = value;
        }
    }
}

return minProd[1][dims.length - 1];
}

```

Complex. Temp. de matrixChainMult

($n = \text{dims.length} - 1$)

Primeiro Ciclo (caso base) $\Theta(n)$

Segundo Ciclo (caso geral)

$$\begin{aligned} \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} \sum_{k=i}^{j-1} 1 &= \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} (j-1-i+1) = \sum_{d=1}^{n-1} \sum_{i=1}^{n-d} d \\ &= \sum_{d=1}^{n-1} (n-d)d = \sum_{d=1}^{n-1} (nd - d^2) = n \sum_{d=1}^{n-1} d - \sum_{d=1}^{n-1} d^2 \\ &= n \frac{(n-1)n}{2} - \frac{(n-1)n(2n-1)}{6} \\ &= \left(\frac{1}{2}n^3 + \dots\right) - \left(\frac{1}{3}n^3 + \dots\right) = \frac{1}{6}n^3 + \dots = \Theta(n^3) \end{aligned}$$

Complexidades

$$(n = \text{dims.length} - 1)$$

Temporal

matrixChainMult

$\Theta(n^3)$

Espacial

matrixChainMult

$\Theta(n^2)$

Matrix minLength tem $n + 1$ linhas e $n + 1$ colunas.

(É fácil diminuir a complexidade espacial?)

Problema do Robotruck



Robotruck (1)

There is a robotic truck that distributes mail packages to several locations in a factory. The robot sits at the end of a conveyer at the mail office and waits for packages to be loaded into its cargo area.

The robot has a **maximum load capacity**, which means that it may have to perform several round trips to complete its task. Provided that the maximum capacity is not exceeded, the robot can stop the conveyer at any time and start a round trip distributing the already collected packages. The **packages must be delivered in the incoming order**.

The **distance** of a round trip is **computed in a grid** by measuring the number of robot moves from the mail office, at location $(0, 0)$, to the location of delivery of the first package, the number of moves between package delivery locations, until the last package, and then the number of moves from the last location back to the mail office. The robot moves a cell at a time either horizontally or vertically in the factory plant grid.

Robotruck (2)

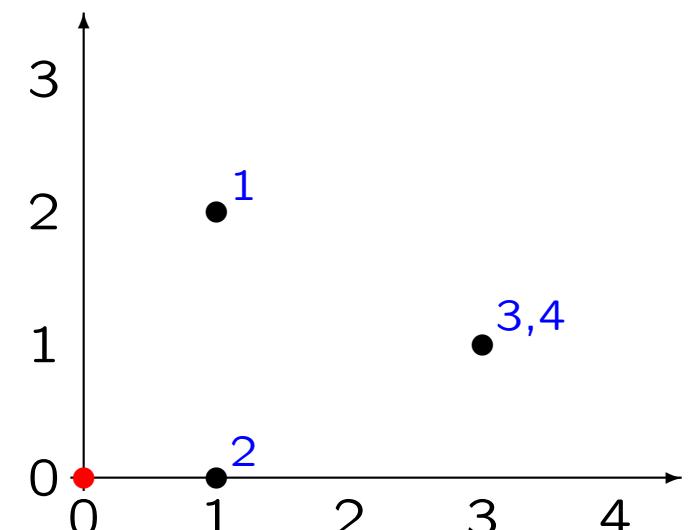
For example, consider four packages, to be delivered at the locations:

$$(1, 2), (1, 0), (3, 1), (3, 1).$$

By dividing these packages into 2 round trips of 2 packages each, the number of moves is $6 + 8 = 14$:

$$3 + 2 + 1 = 6 \quad (\text{first trip})$$

$$4 + 0 + 4 = 8 \quad (\text{second trip})$$



Notice that the two last packages are delivered at the same location and thus the number of moves between them is 0.

Robotruck (3)

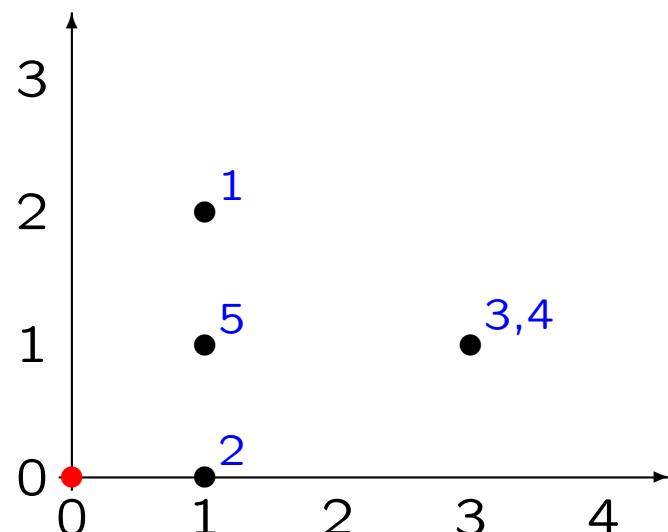
Given:

the robot maximum load capacity: C

the sequence of delivery locations: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

the package weights: w_1, w_2, \dots, w_n

$(n \geq 1, x_i, y_i \geq 0$ and $1 \leq w_i \leq C$, for every $i = 1, \dots, n$) compute the
minimum distance the robot must travel to deliver all packages.



$$C = 10$$

Locais: $(1, 2), (1, 0), (3, 1), (3, 1), (1, 1)$

Pesos: 3, 2, 4, 4, 3

Resolução do Problema (1)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

Distância entre Locais

$d((x, y), (x', y'))$ é a **distância entre os locais** (x, y) e (x', y') :

$$d((x, y), (x', y')) = |x - x'| + |y - y'|$$

Resolução do Problema (2)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

Distância de *round trip*

$r(i, j)$ é a **distância da viagem completa, da base até à base, para entregar os pacotes $i, i + 1, \dots, j$** (com $i \leq j$):

$$\begin{aligned} r(i, j) &= d((0, 0), (x_i, y_i)) \\ &\quad + \sum_{k=i}^{j-1} d((x_k, y_k), (x_{k+1}, y_{k+1})) \\ &\quad + d((x_j, y_j), (0, 0)) \end{aligned}$$

Resolução do Problema (3)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

Resolução do Problema (4)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $i = 1$, $\mathcal{D}(i) = r(1, 1);$
- Se $i = 2$ e $w_1 + w_2 \leq C$,

Resolução do Problema (5)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $i = 1$, $\mathcal{D}(i) = r(1, 1);$
- Se $i = 2$ e $w_1 + w_2 \leq C$, $\mathcal{D}(i) = r(1, 2);$
- Se $i = 3$ e $w_1 + w_2 + w_3 \leq C$,

Resolução do Problema (6)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $i = 1$, $\mathcal{D}(i) = r(1, 1);$
- Se $i = 2$ e $w_1 + w_2 \leq C$, $\mathcal{D}(i) = r(1, 2);$
- Se $i = 3$ e $w_1 + w_2 + w_3 \leq C$, $\mathcal{D}(i) = r(1, 3);$
- Se $\left(\sum_{k=1}^i w_k \right) \leq C$,

Resolução do Problema (7)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $i = 1$, $\mathcal{D}(i) = r(1, 1);$
- Se $i = 2$ e $w_1 + w_2 \leq C$, $\mathcal{D}(i) = r(1, 2);$
- Se $i = 3$ e $w_1 + w_2 + w_3 \leq C$, $\mathcal{D}(i) = r(1, 3);$
- Se $\left(\sum_{k=1}^i w_k \right) \leq C$, $\mathcal{D}(i) = r(1, i);$

Resolução do Problema (8)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?

Resolução do Problema (9)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima:

Resolução do Problema (10)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima: $r(i, i) + \mathcal{D}(i - 1)$

Resolução do Problema (11)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima: $r(i, i) + \mathcal{D}(i - 1)$
 - os 2 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima:

Resolução do Problema (12)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima: $r(i, i) + \mathcal{D}(i - 1)$
 - os 2 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima: $r(i - 1, i) + \mathcal{D}(i - 2)$

Resolução do Problema (13)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima** para entregar os pacotes $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima: $r(i, i) + \mathcal{D}(i - 1)$
 - os 2 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima: $r(i - 1, i) + \mathcal{D}(i - 2)$
 - os 3 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima: $r(i - 2, i) + \mathcal{D}(i - 3)$

Resolução do Problema (14)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima para entregar os pacotes** $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, quais os pacotes entregues na última viagem?
 - apenas o último pacote (uma alternativa que existe sempre)
Distância mínima: $r(i, i) + \mathcal{D}(i - 1)$
 - os 2 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima: $r(i - 1, i) + \mathcal{D}(i - 2)$
 - os 3 últimos pacotes, se a soma dos seus pesos não exceder C
Distância mínima: $r(i - 2, i) + \mathcal{D}(i - 3)$
 - e assim sucessivamente. Qual é a melhor alternativa?

Resolução do Problema (15)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

$\mathcal{D}(i)$ é a **distância mínima para entregar os pacotes** $1, \dots, i$ ($i \geq 1$):

- Se $\left(\sum_{k=1}^i w_k \right) > C$, a última viagem leva os pacotes $j, j+1, \dots, i-1, i$, para algum j (entre 2 e i) tal que $w_j + w_{j+1} + \dots + w_i \leq C$:

$$\begin{aligned}\mathcal{D}(i) = \min \quad & (\quad r(i, i) \quad + \quad \mathcal{D}(i-1) \quad , \\ & \quad r(i-1, i) \quad + \quad \mathcal{D}(i-2) \quad , \\ & \quad r(i-2, i) \quad + \quad \mathcal{D}(i-3) \quad , \\ & \quad \dots \quad , \\ & \quad r(j, i) \quad + \quad \mathcal{D}(j-1) \quad)\end{aligned}$$

Resolução do Problema (16)

Capacidade do robot: C

Locais das entregas: $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$

Pesos dos pacotes: w_1, w_2, \dots, w_n

Qual é o valor de $\mathcal{D}(n)$?

$$\mathcal{D}(i) = \begin{cases} r(1, i) & \left(\sum_{k=1}^i w_k \right) \leq C \\ \min_{\{j \mid 1 < j \leq i \text{ e } \left(\sum_{k=j}^i w_k \right) \leq C\}} (r(j, i) + \mathcal{D}(j - 1)) & \text{caso contrário} \end{cases}$$

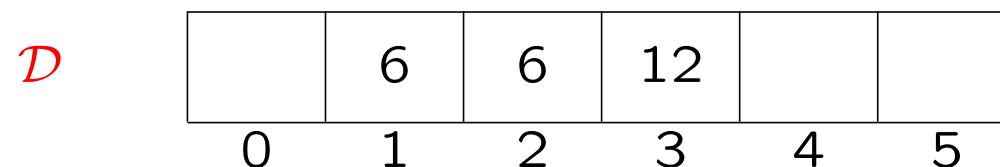
Para o exemplo seguinte, qual é o valor de $\mathcal{D}(5)$?

$C = 10$ Locais $(1, 2), (1, 0), (3, 1), (3, 1), (1, 1)$ Pesos $3, 2, 4, 4, 3$

Programação Dinâmica da Função \mathcal{D} (1)

$$\mathcal{D}(i) = \begin{cases} r(1, i) & \left(\sum_{k=1}^i w_k \right) \leq C \\ \min_{\{j \mid 1 < j \leq i \text{ e } \left(\sum_{k=j}^i w_k \right) \leq C\}} (r(j, i) + \mathcal{D}(j - 1)) & \text{caso contrário} \end{cases}$$

$C = 10$ Locais $(1,2), (1,0), (3,1), (3,1), (1,1)$ Pesos 3, 2, 4, 4, 3



$$\mathcal{D}[1] = r(1, 1) = 3 + 3 = 6$$

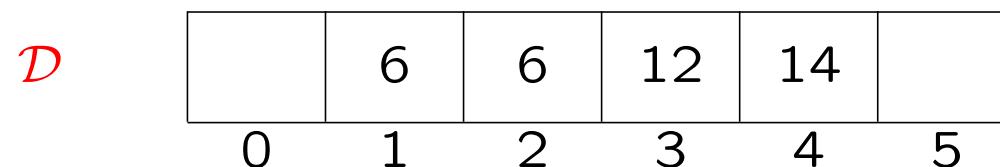
$$\mathcal{D}[2] = r(1, 2) = 3 + 2 + 1 = 6$$

$$\mathcal{D}[3] = r(1, 3) = 3 + 2 + 3 + 4 = 12$$

Programação Dinâmica da Função \mathcal{D} (2)

$$\mathcal{D}(i) = \begin{cases} r(1, i) & \left(\sum_{k=1}^i w_k \right) \leq C \\ \min_{\{j \mid 1 < j \leq i \text{ e } \left(\sum_{k=j}^i w_k \right) \leq C\}} (r(j, i) + \mathcal{D}(j-1)) & \text{caso contrário} \end{cases}$$

$C = 10$ Locais $(1,2), (1,0), (3,1), (3,1), (1,1)$ Pesos $3, 2, 4, 4, 3$

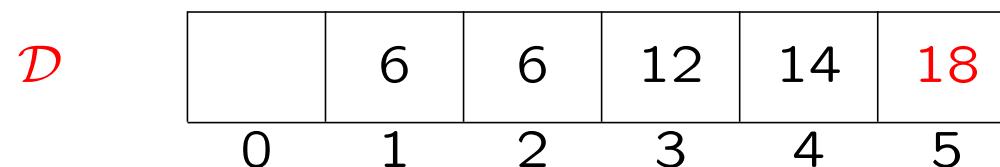


$$\begin{aligned} \mathcal{D}[4] &= \min(\underline{r(4,4) + \mathcal{D}[3]}, \underline{r(3,4) + \mathcal{D}[2]}, \underline{r(2,4) + \mathcal{D}[1]}) \\ &= \min(8 + 12, \underline{8 + 6}, \underline{8 + 6}) \end{aligned}$$

Programação Dinâmica da Função \mathcal{D} (3)

$$\mathcal{D}(i) = \begin{cases} r(1, i) & \left(\sum_{k=1}^i w_k \right) \leq C \\ \min_{\{j \mid 1 < j \leq i \text{ e } \left(\sum_{k=j}^i w_k \right) \leq C\}} (r(j, i) + \mathcal{D}(j-1)) & \text{caso contrário} \end{cases}$$

$C = 10$ Locais $(1,2), (1,0), (3,1), (3,1), (1,1)$ Pesos $3, 2, 4, 4, 3$



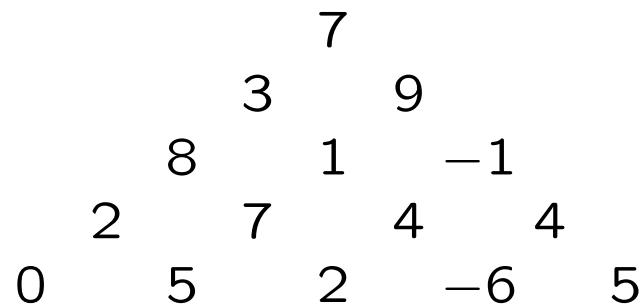
$$\begin{aligned} \mathcal{D}[5] &= \min(\underline{r(5,5) + \mathcal{D}[4]}, r(4,5) + \mathcal{D}[3]) \\ &= \min(\underline{4 + 14}, 8 + 12) \end{aligned}$$

Problema do Jogo da Pirâmide



Jogo da Pirâmide (1)

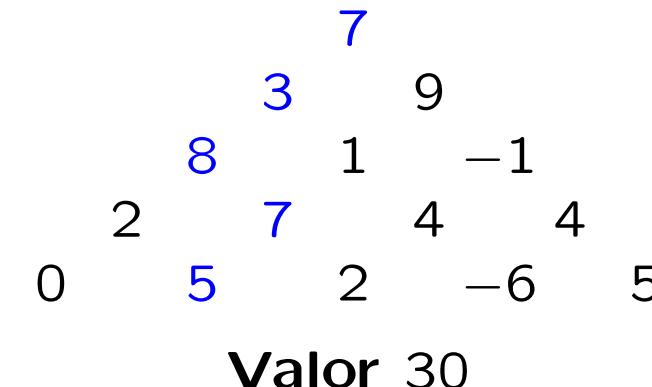
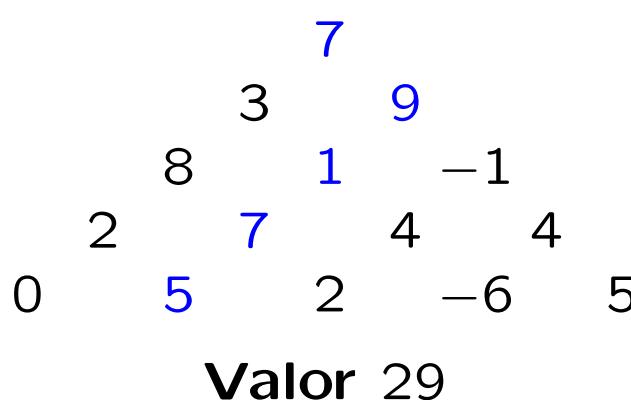
No jogo da Pirâmide, o jogador recebe uma pirâmide de números inteiros. O objetivo é efetuar um **percurso do topo** da pirâmide **até à base** que **maximize a soma** dos números do percurso.



O percurso começa no topo da pirâmide. Em cada ponto do percurso (que não pertença à base da pirâmide), o jogador é obrigado a descer para o número que se encontra imediatamente à esquerda ou imediatamente à direita do ponto onde se encontra. O percurso termina assim que se chega à base da pirâmide.

Jogo da Pirâmide (2)

Considere os dois seguintes percursos na pirâmide (e os seus valores):



Se a pirâmide tiver n níveis ($n \geq 1$),
está guardada numa tabela com n
linhas e n colunas, como se ilustra
na figura da direita.

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

Dada uma pirâmide, qual é o **valor máximo** dos percursos do topo até à base?

Resolução do Problema (1)

Pirâmide com $n \geq 1$ níveis: $\text{int}[n][n]$ P

$V(i, j)$ é o **valor máximo** dos percursos desde a posição (i, j) até à **base** (com $i = 0, 1, \dots, n - 1$ e $j = 0, 1, \dots, i$):

Resolução do Problema (2)

Pirâmide com $n \geq 1$ níveis: $\text{int}[n][n]$ P

$\mathcal{V}(i, j)$ é o **valor máximo** dos percursos desde a posição (i, j) até à **base** (com $i = 0, 1, \dots, n - 1$ e $j = 0, 1, \dots, i$):

- (**Posição pertence à base**) Se $i = n - 1$, $\mathcal{V}(i, j) = P[i][j]$;

Resolução do Problema (3)

Pirâmide com $n \geq 1$ níveis: $\text{int}[n][n]$ P

$\mathcal{V}(i, j)$ é o **valor máximo** dos percursos desde a posição (i, j) até à **base** (com $i = 0, 1, \dots, n - 1$ e $j = 0, 1, \dots, i$):

- (**Posição pertence à base**) Se $i = n - 1$, $\mathcal{V}(i, j) = P[i][j]$;
- (**Caso geral**) Se $i < n - 1$, desce-se pela esquerda ou pela direita.

Portanto:

$$\begin{aligned}\mathcal{V}(i, j) &= \max \left(P[i][j] + \mathcal{V}(i + 1, j), P[i][j] + \mathcal{V}(i + 1, j + 1) \right) \\ &= P[i][j] + \max \left(\mathcal{V}(i + 1, j), \mathcal{V}(i + 1, j + 1) \right).\end{aligned}$$

Resolução do Problema (4)

Pirâmide com $n \geq 1$ níveis: $\text{int}[n][n]$ P

Qual é o valor de $\mathcal{V}(0, 0)$?

$$\mathcal{V}(i, j) = \begin{cases} P[i][j] & i = n - 1 \\ & \text{e } 0 \leq j \leq i \\ P[i][j] + \max(\mathcal{V}(i + 1, j), \mathcal{V}(i + 1, j + 1)) & 0 \leq i \leq n - 2 \\ & \text{e } 0 \leq j \leq i \end{cases}$$

Programação Dinâmica da Função \mathcal{V} (1)

$$\mathcal{V}(i, j) = \begin{cases} P[i][j] & i = n - 1 \\ & \text{e } 0 \leq j \leq i \\ P[i][j] + \max(\mathcal{V}(i + 1, j), \mathcal{V}(i + 1, j + 1)) & 0 \leq i \leq n - 2 \\ & \text{e } 0 \leq j \leq i \end{cases}$$

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

\mathcal{V}	0	1	2	3	4
0	0				
1	1				
2	2				
3	3				
4	0	5	2	-6	5

$$\mathcal{V}[4][1] = P[4][1]$$

Programação Dinâmica da Função \mathcal{V} (2)

$$\mathcal{V}(i, j) = \begin{cases} P[i][j] & i = n - 1 \\ & \text{e } 0 \leq j \leq i \\ P[i][j] + \max(\mathcal{V}(i + 1, j), \mathcal{V}(i + 1, j + 1)) & 0 \leq i \leq n - 2 \\ & \text{e } 0 \leq j \leq i \end{cases}$$

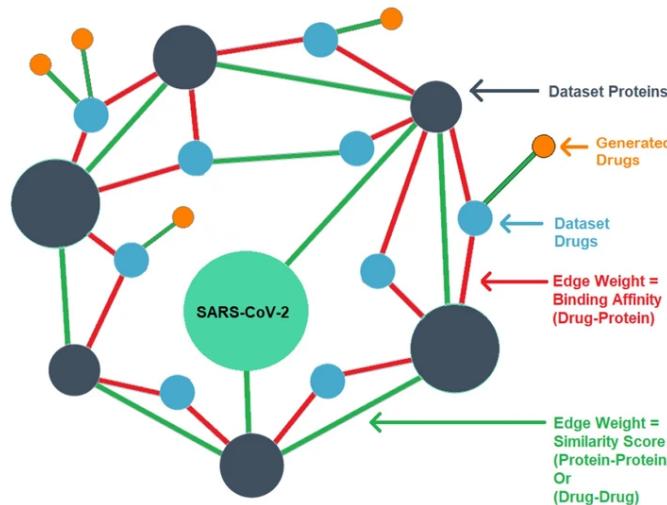
P	0	1	2	3	4
0	7				
1	3	9			
2	8	1	-1		
3	2	7	4	4	
4	0	5	2	-6	5

\mathcal{V}	0	1	2	3	4
0	30				
1	23	22			
2	20	13	8		
3	7	12	6	9	
4	0	5	2	-6	5

$$\begin{aligned} \mathcal{V}[2][1] &= P[2][1] + \max(\underline{\mathcal{V}[3][1]}, \mathcal{V}[3][2]) \\ &= 1 + \max(\underline{12}, 6) \end{aligned}$$

Capítulo II

Noções Básicas de Grafos

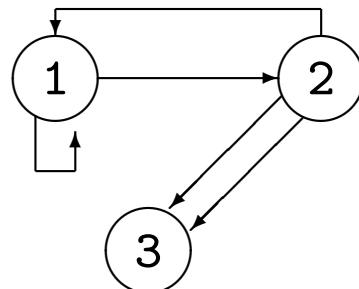


Grafo $G = (V, A)$

V conjunto de **vértices** ou **nós**

A coleção de **arcos** ou **arestas**

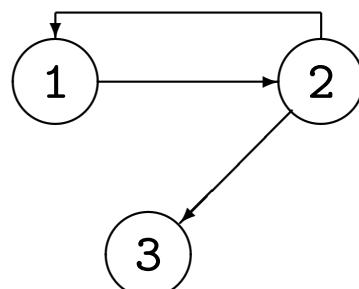
Grafo Genérico — Com arcos paralelos ou com lacetes.



$$V = \{1, 2, 3\}$$

$$A = <(1, 1), (1, 2), (2, 1), (2, 3), (2, 3)>$$

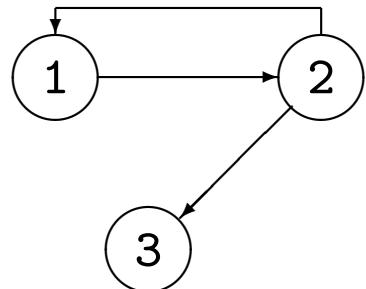
Grafo Simples — $A \subseteq V \times V$ e sem lacetes.



$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 1), (2, 3)\}$$

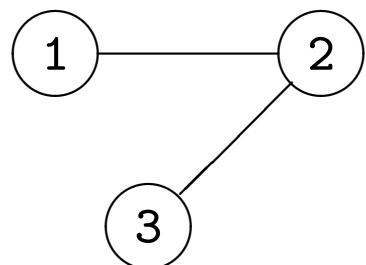
Grafo Orientado — Os arcos têm sentido.



$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 1), (2, 3)\}$$

Grafo Não Orientado — Os arcos não têm sentido único.



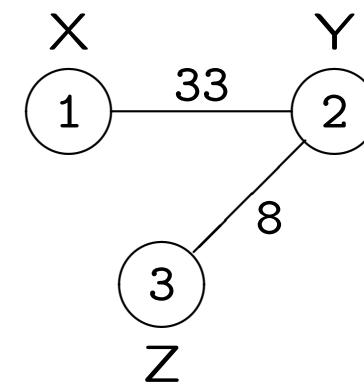
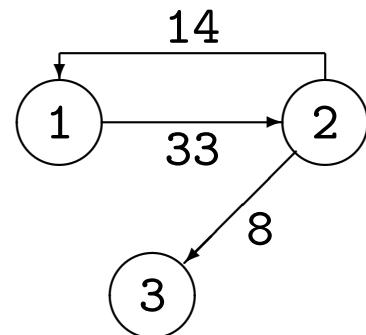
$$V = \{1, 2, 3\}$$

$$A = \{(1, 2), (2, 3)\}$$

Considera-se que $(\forall v, w \in V) (v, w) = (w, v)$.

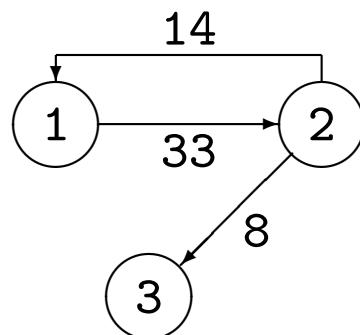
Grafo Etiquetado (ou Pesado)

Os vértices, os arcos ou ambos têm
uma **etiqueta**, um **peso** ou um **custo**.



Caminho

É uma sequência não vazia de vértices v_1, v_2, \dots, v_n (com $n \geq 1$), tal que, para qualquer $i = 1, 2, \dots, n - 1$: $(v_i, v_{i+1}) \in A$.



Caminho: 2, 1, 2, 3

Comprimento: 3

Comprimento Pesado: 55

Comprimento do Caminho: o número de arcos do caminho ($n - 1$).

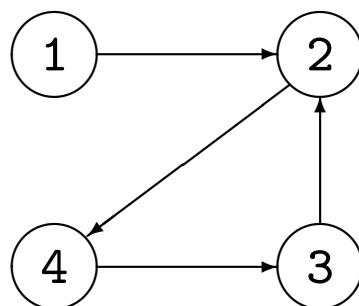
Comprimento Pesado ou Custo do Caminho: a soma dos pesos (numéricos) dos arcos do caminho (num grafo pesado).

Caminho Simples: um caminho cujos vértices são todos diferentes, exceto, possivelmente, o primeiro e o último.

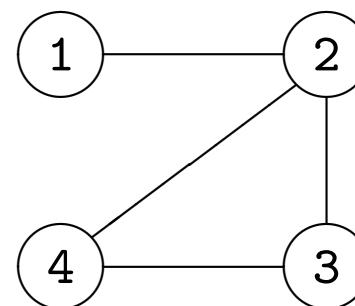
Ciclo ou Circuito

Num Grafo Orientado: um caminho de comprimento positivo onde o primeiro e o último vértices são iguais.

Num Grafo Não Orientado: um caminho de comprimento positivo, onde o primeiro e o último vértices são iguais, que não passa 2 vezes pelo mesmo arco.



Ciclo:
2, 4, 3, 2
Não é ciclo:
1



Ciclo:
2, 3, 4, 2
Não é ciclo:
1, 2, 1

Grafo Cíclico / Acíclico: um grafo com / sem ciclos.

Conectividade

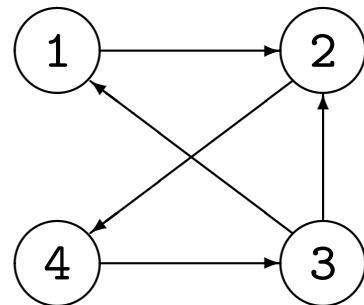
Grafo Fortemente Conexo: um **grafo orientado** tal que:

$(\forall v, w \in V)$ existe um caminho de v para w .

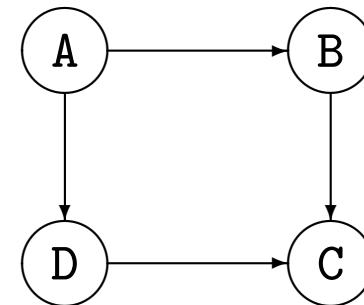
Grafo Fracamente Conexo: um **grafo orientado** tal que, ignorando o sentido dos arcos:

$(\forall v, w \in V)$ existe um caminho de v para w .

Grafo
fortemente
conexo



Grafo
fracamente
conexo



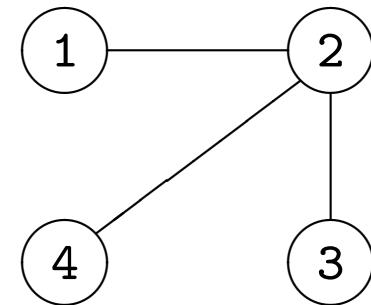
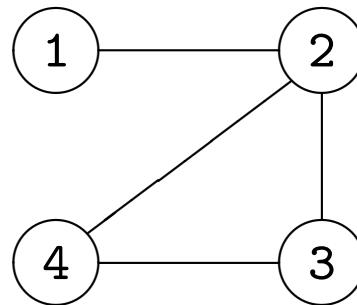
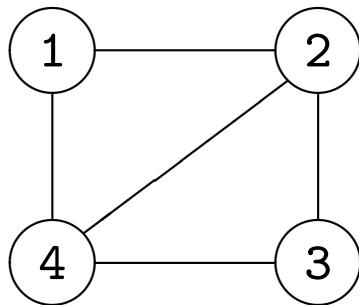
Grafo Conexo: um **grafo não orientado** tal que:

$(\forall v, w \in V)$ existe um caminho de v para w .

Sub-grafos e Árvores

Sub-grafo de (V, A) : um grafo (V', A') tal que $V' \subseteq V$ e $A' \subseteq A$.

Sub-grafo de Cobertura de (V, A) : um sub-grafo (V', A') de (V, A) , com $V' = V$.



Árvore (livre): um **grafo não orientado**, conexo e acíclico.

Árvore de Cobertura de (V, A) : Um sub-grafo de cobertura de (V, A) que é árvore.

Tipos Abstratos de Dados

Vértice, Arco, Grafo Não Orientado e Grafo Orientado

- Os cinco slides que se seguem introduzem os métodos usados nos slides das aulas teóricas.
- Por questões de eficiência, **não implementem** as interfaces correspondentes.
- Por exemplo, em geral, um vértice é um número inteiro (entre zero e número-total-de-vértices – 1), não havendo interface nem classe para os vértices.

ADT Node

// In practice, a node is an integer.

ADT Edge<L> (whose label is of type L)

// Returns the edge label.

L **label**();

// Returns the first endpoint of the edge,

// which is its origin if the edge is directed.

Node **firstNode**();

// Returns the second endpoint of the edge,

// which is its destination if the edge is directed.

Node **secondNode**();

// Returns the edge endpoint that is distinct from the specified node.

Node **oppositeNode**(Node node);

ADT AnyGraph<L>

```
// Returns the number of nodes.          // Returns the nodes.  
int numNodes( );           Iterable<Node> nodes( );  
  
// Returns the number of edges.          // Returns the edges.  
int numEdges( );           Iterable<Edge<L>> edges( );  
  
// Returns an arbitrary node.  
Node aNode( );  
  
// Inserts edge (node1, node2) whose label is the specified label.  
void addEdge( Node node1, Node node2, L label );  
  
// Returns true iff there is an edge of the form (node1, node2).  
boolean edgeExists( Node node1, Node node2 );
```

ADT UndiGraph<L> (extends AnyGraph<L>)

// Returns the degree of the specified node.

int degree(Node node);

// Returns the nodes adjacent to the specified node.

Iterable<Node> adjacentNodes(Node node);

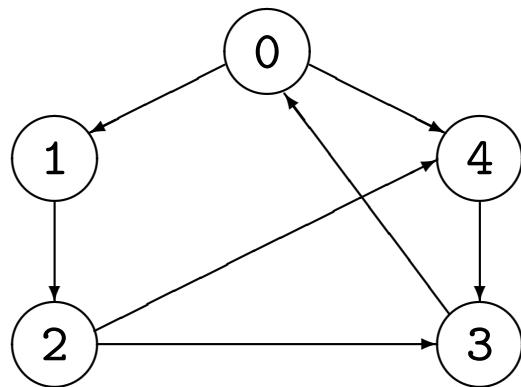
// Returns the edges incident on the specified node.

Iterable<Edge<L>> incidentEdges(Node node);

ADT Digraph<L> (extends AnyGraph<L>)

```
// Returns the in-degree of the specified node.  
int inDegree( Node node );  
  
// Returns the out-degree of the specified node.  
int outDegree( Node node );  
  
// Returns the nodes adjacent to the specified node along incoming  
// edges to it.  
Iterable<Node> inAdjacentNodes( Node node );  
  
// Returns the nodes adjacent to the specified node along outgoing  
// edges from it.  
Iterable<Node> outAdjacentNodes( Node node );  
  
// Returns the incoming edges to the specified node.  
Iterable<Edge<L>> inIncidentEdges( Node node );  
  
// Returns the outgoing edges from the specified node.  
Iterable<Edge<L>> outIncidentEdges( Node node );
```

Matriz de Adjacências



	0	1	2	3	4
0	0	1	0	0	1
1	0	0	1	0	0
2	0	0	0	1	1
3	1	0	0	0	0
4	0	0	0	1	0

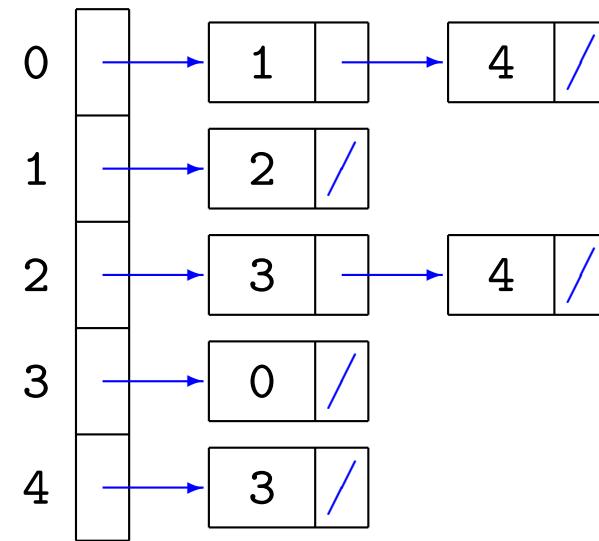
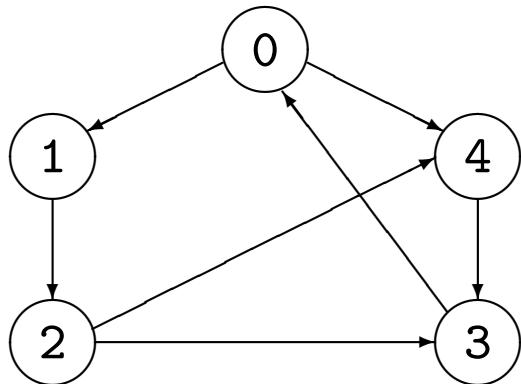
Pesquisar Arco (v_1, v_2) $\Theta(1)$

Obter Sucessores (diretos) v $\Theta(|V|)$

Antecessores (diretos) v $\Theta(|V|)$

Memória Requerida $\Theta(|V|^2)$

Listas Ligadas de Adjacências de Sucessores (diretos)



Pesquisar Arco (v_1, v_2)

$O(|\text{Suc}(v_1)|)$

Obter Sucessores (diretos) v

$\Theta(|\text{Suc}(v)|)$

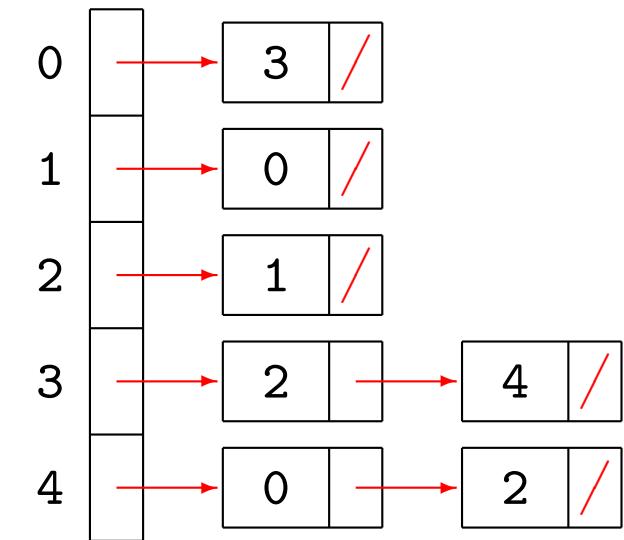
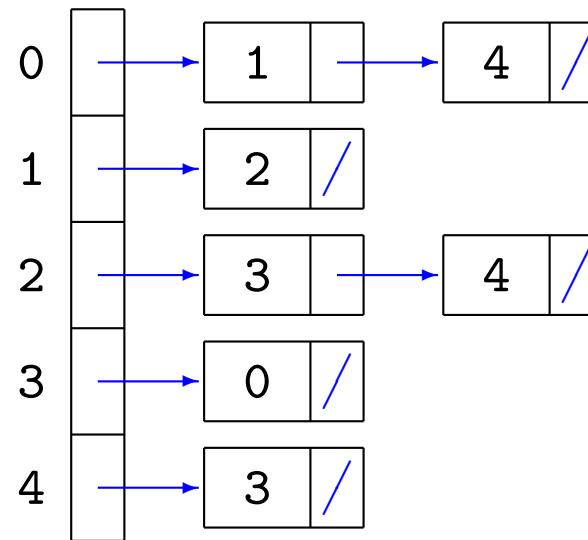
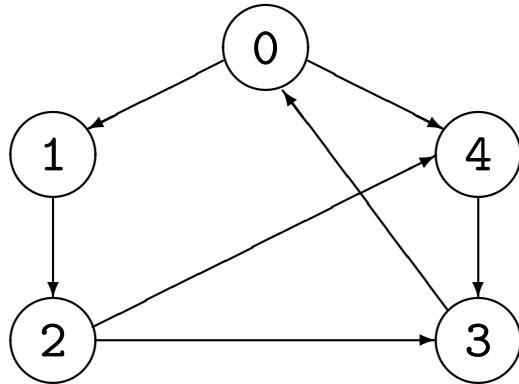
Antecessores (diretos) v

$O(|V| + |A|)$

Memória Requerida

$\Theta(|V| + |A|)$

Listas Ligadas de Adjacências de Sucessores e de Antecessores



Pesquisar Arco (v_1, v_2) $O(\min(|\text{Suc}(v_1)|, |\text{Ant}(v_2)|))$

Obter Sucessores v $\Theta(|\text{Suc}(v)|)$

Antecessores v $\Theta(|\text{Ant}(v)|)$

Memória Requerida $\Theta(|V| + |A|)$

Exemplo de Tradução do Pseudo-código (1)

```
int algorithm( Digraph graph, Node source ) {  
    for every Node v in graph.nodes()  
        ( ... )  
    for every Node v in graph.outAdjacentNodes(source)  
        ( ... )  
    return ...  
}
```

- **Quais são as operações sobre o grafo?** Percorrem-se os nós e percorrem-se os sucessores (diretos) de um nó arbitrário.
- **Quantos nós e quantos arcos pode ter o grafo?** Entre 2 e 100 000 nós; entre 1 e 500 000 arcos.
- **Como se obtém a informação sobre os nós e os arcos do grafo?** Inicialmente, sabe-se o número de nós; depois, conhecem-se os arcos por uma ordem qualquer.
- **DECIDIR como guardar o grafo e TRADUZIR o pseudo-código de acordo com essa implementação.**

Exemplo de Tradução do Pseudo-código (2)

```
int algorithm( Digraph graph, Node source ) {
    for every Node v in graph.nodes()
        ( ... )
    for every Node v in graph.outAdjacentNodes(source)
        ( ... )
    return ...
}
```

Grafo
implementado
em vetor
de listas ligadas
de adjacências
de sucessores

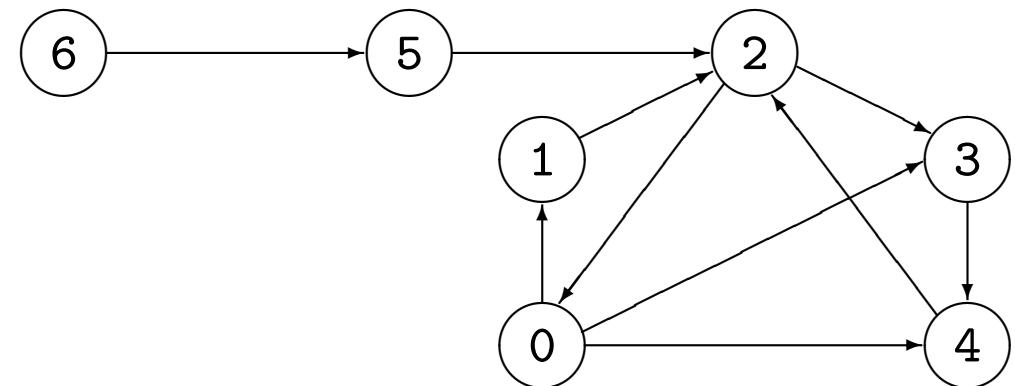
```
class Problem {
    private int numNodes;
    private List<Integer>[] edges;

    public int algorithm( int source ) {
        for ( int v = 0; v < numNodes; v++ )
            ( ... )
        for ( int v : edges[source] )
            ( ... )
        return ...
    }
}
```

Capítulo III

Percorso em Profundidade
(num grafo orientado ou não orientado)

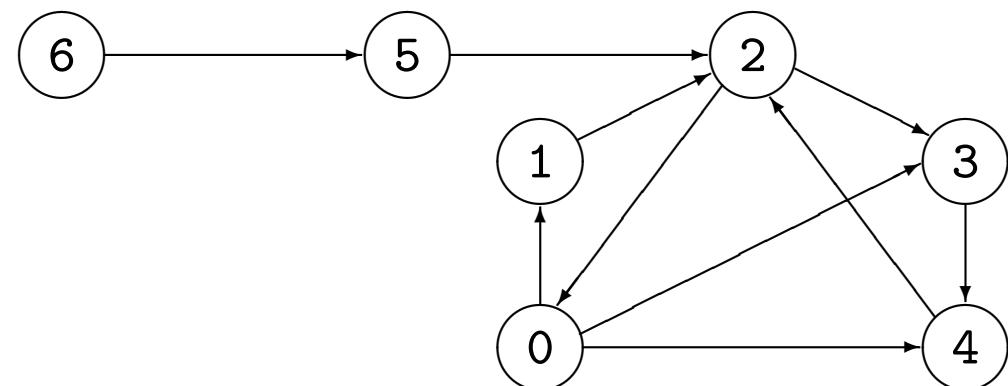
Percorso em Profundidade



Percorso em Profundidade

0

Ordem:

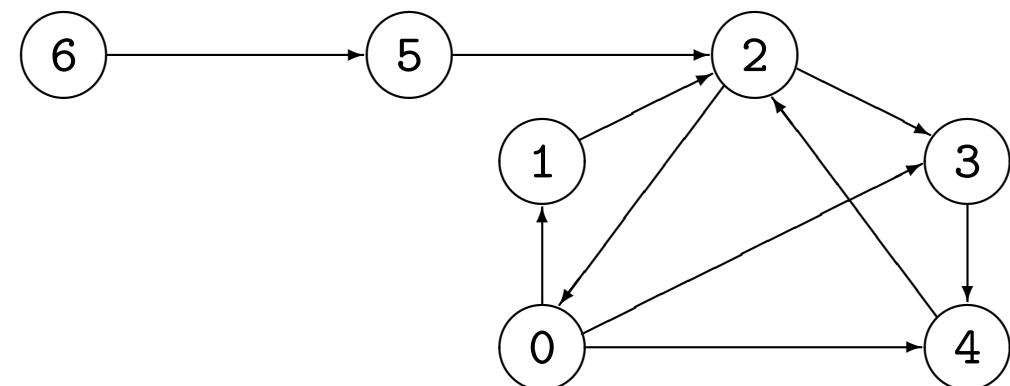


Percorso em Profundidade

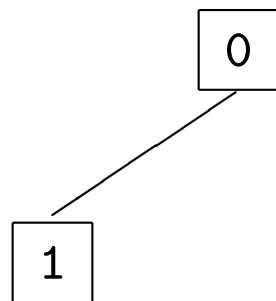
0

Ordem:

0

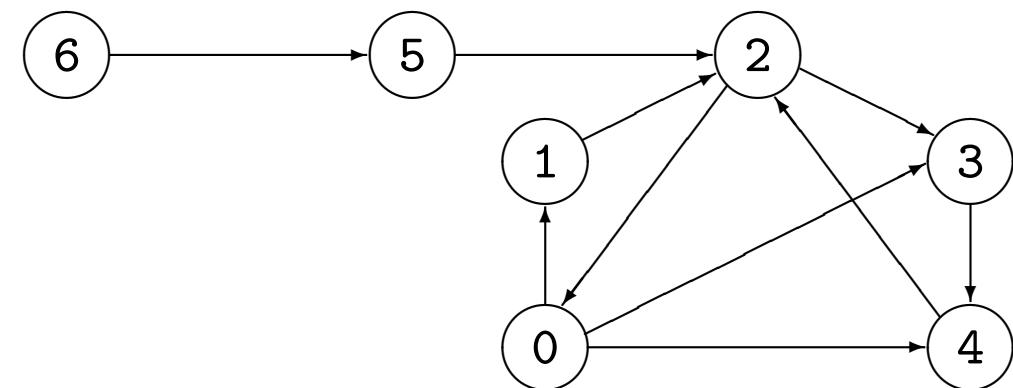


Percorso em Profundidade

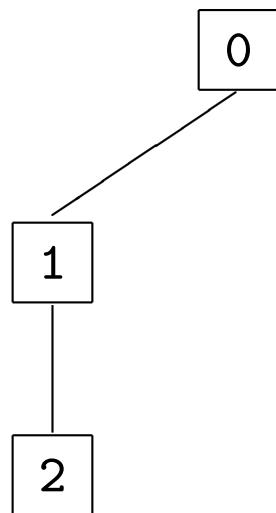


Ordem:

0 1

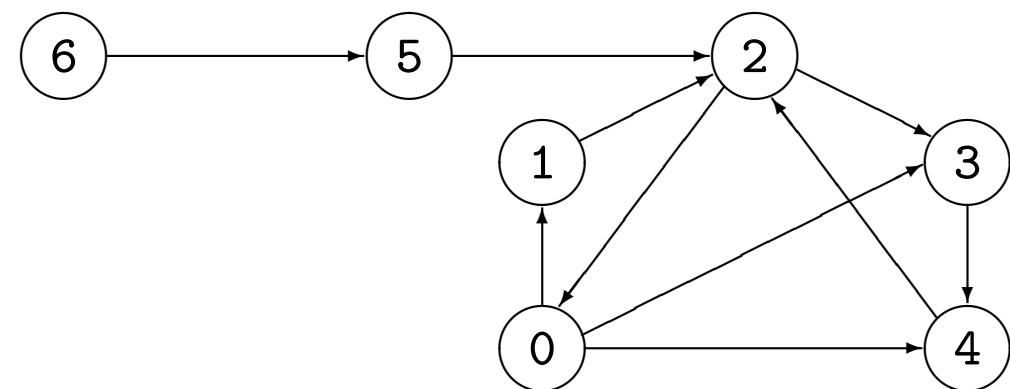


Percorso em Profundidade

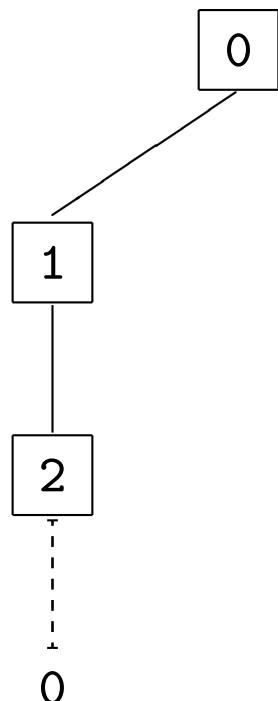


Ordem:

0 1 2

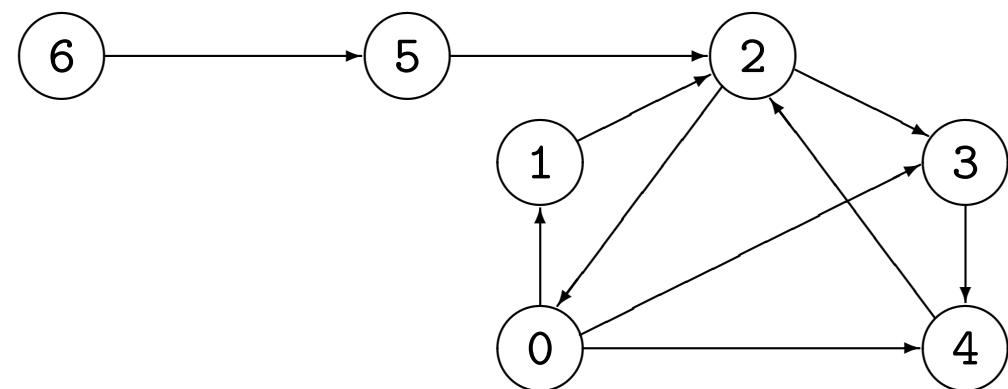


Percorso em Profundidade

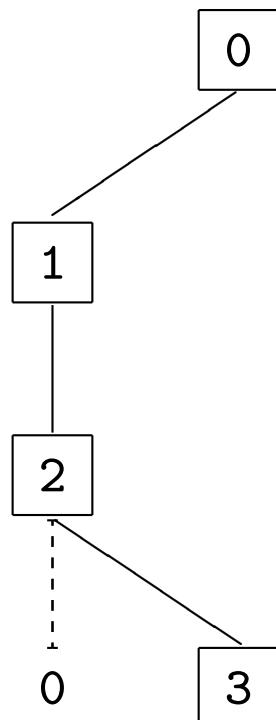


Ordem:

0 1 2

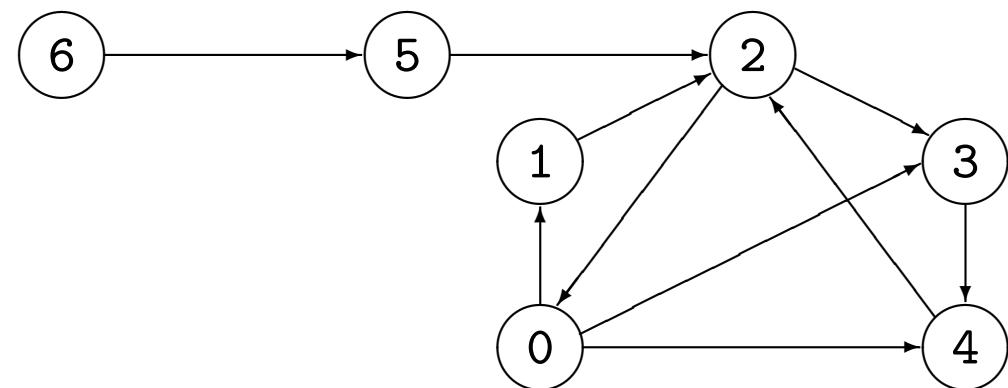


Percorso em Profundidade

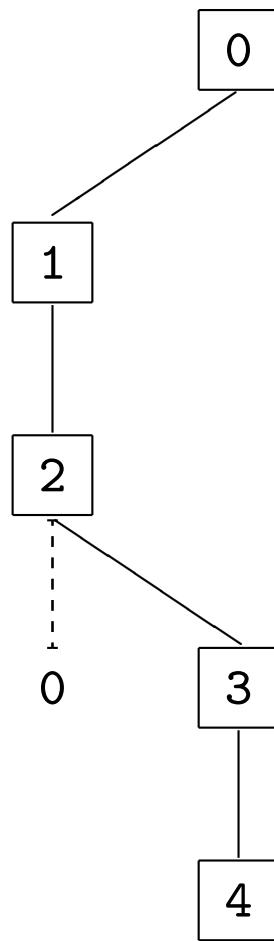


Ordem:

0 1 2 3

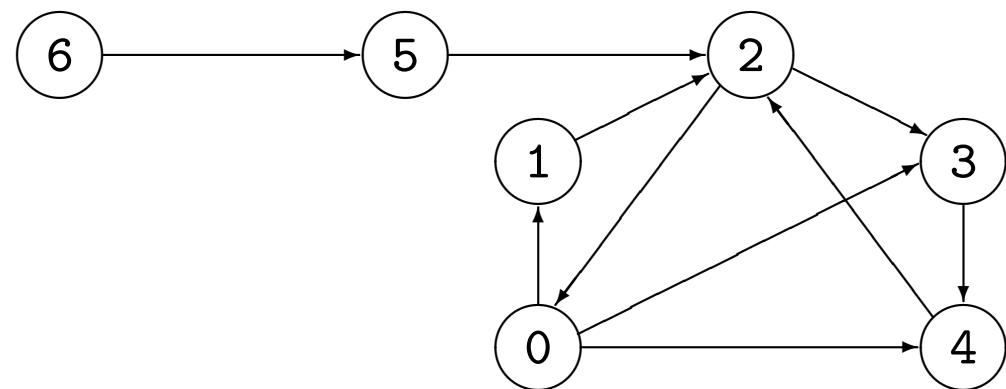


Percorso em Profundidade

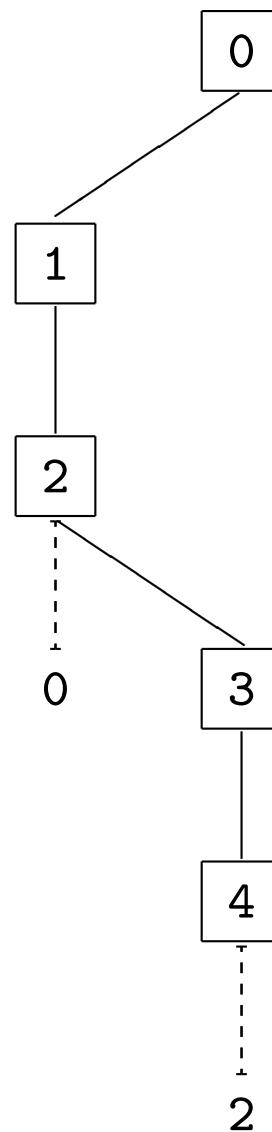


Ordem:

0 1 2 3 4

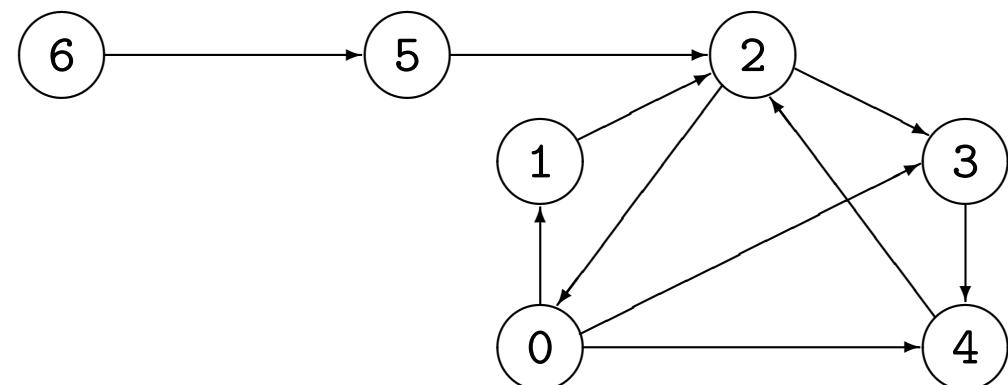


Percorso em Profundidade

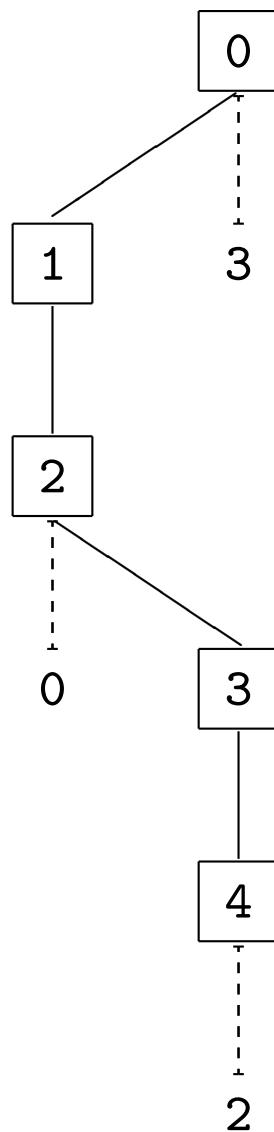


Ordem:

0 1 2 3 4

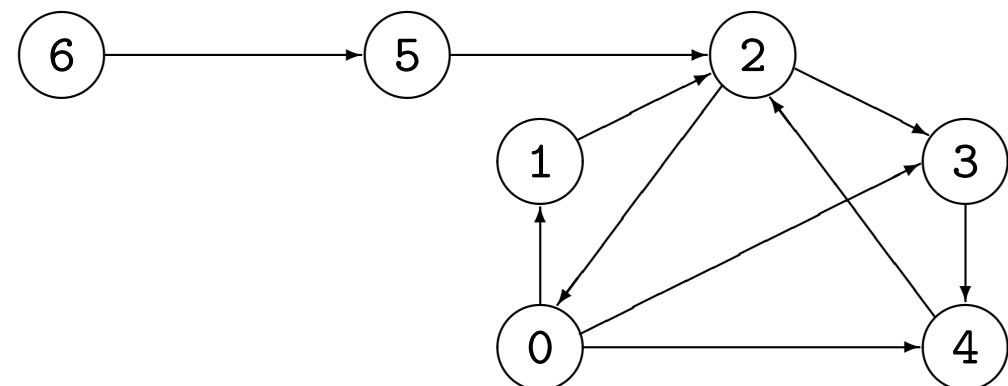


Percorso em Profundidade

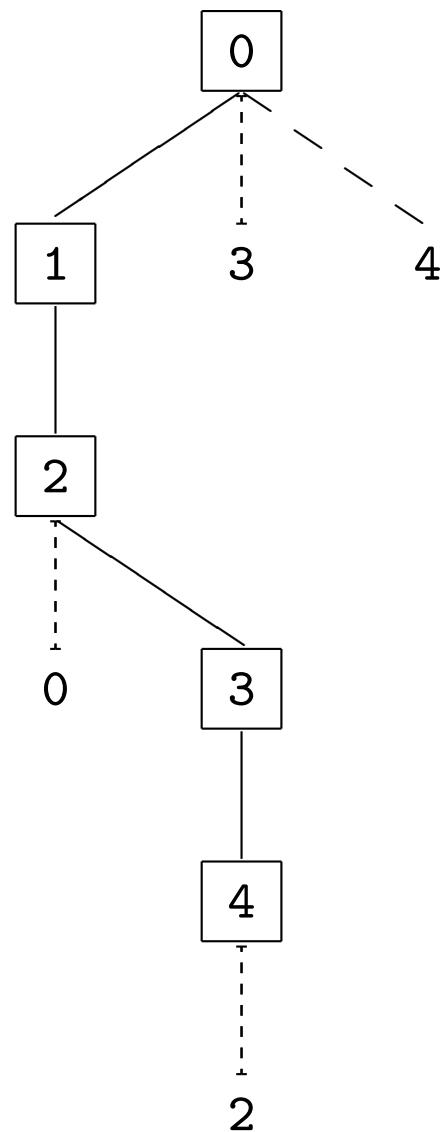


Ordem:

0 1 2 3 4

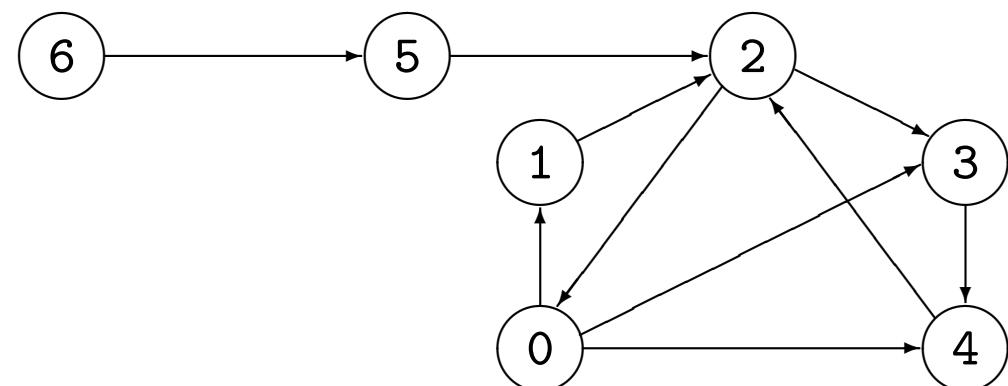


Percorso em Profundidade

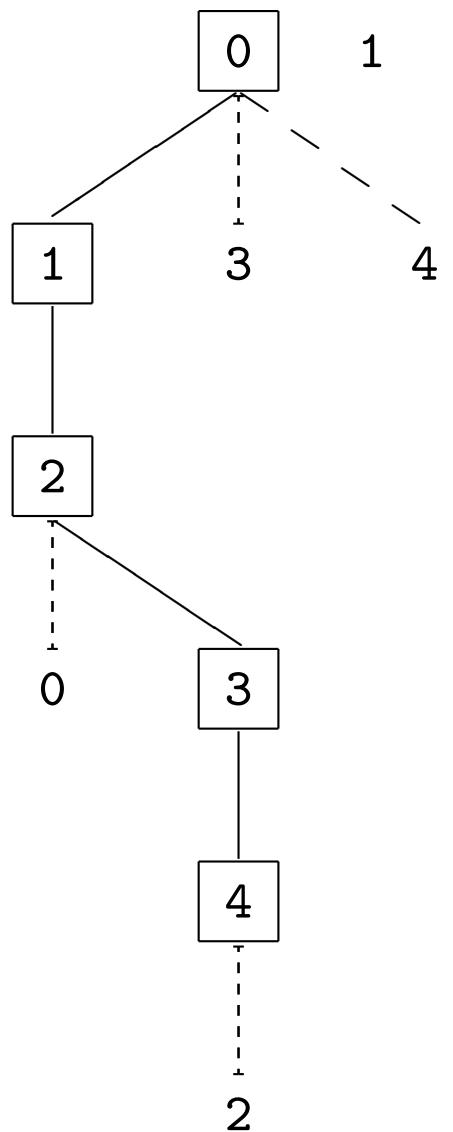


Ordem:

0 1 2 3 4

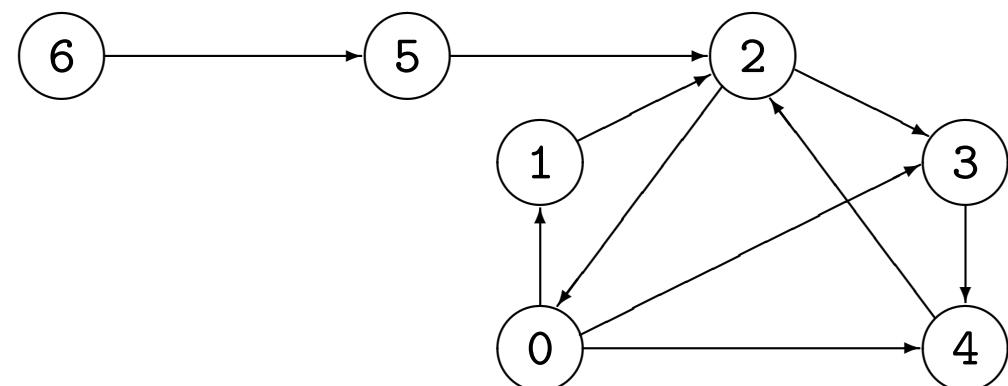


Percorso em Profundidade

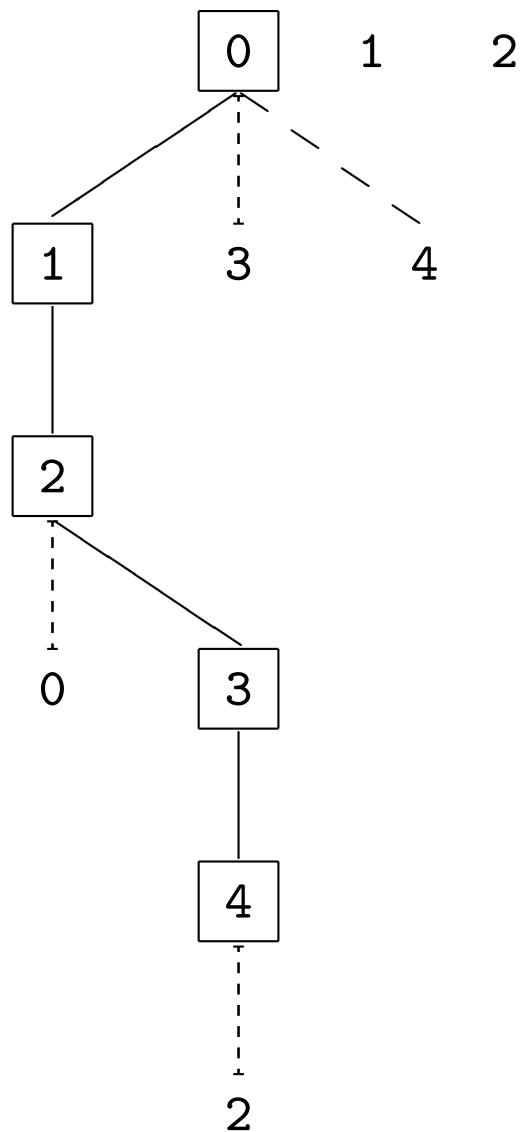


Ordem:

0 1 2 3 4

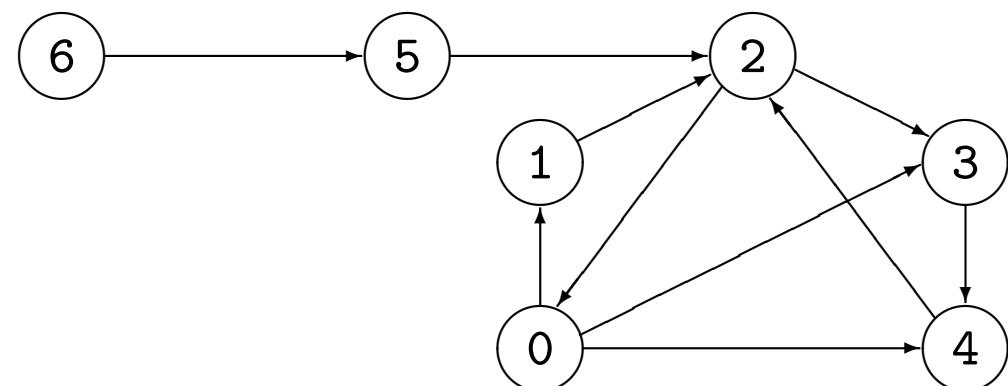


Percorso em Profundidade

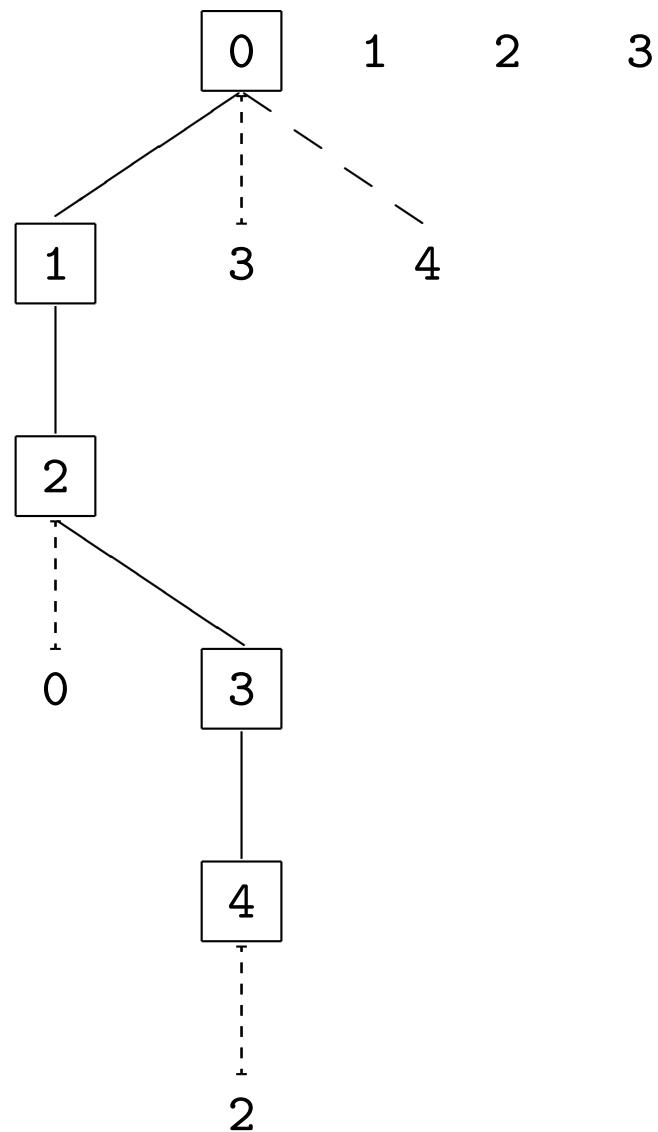


Ordem:

0 1 2 3 4

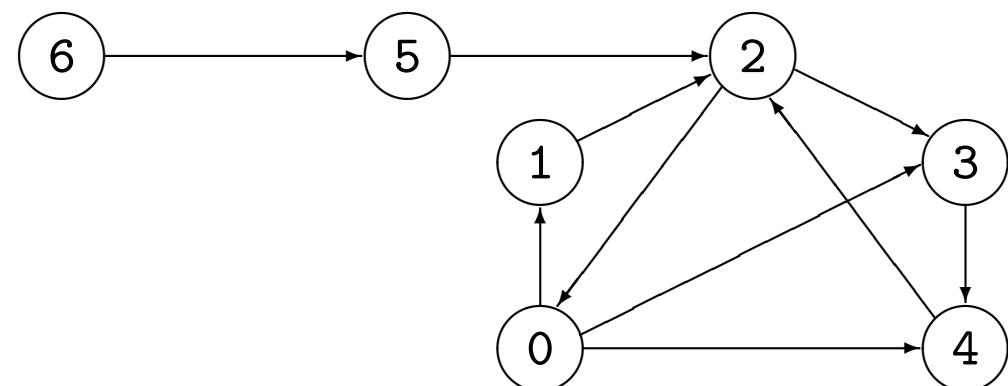


Percorso em Profundidade

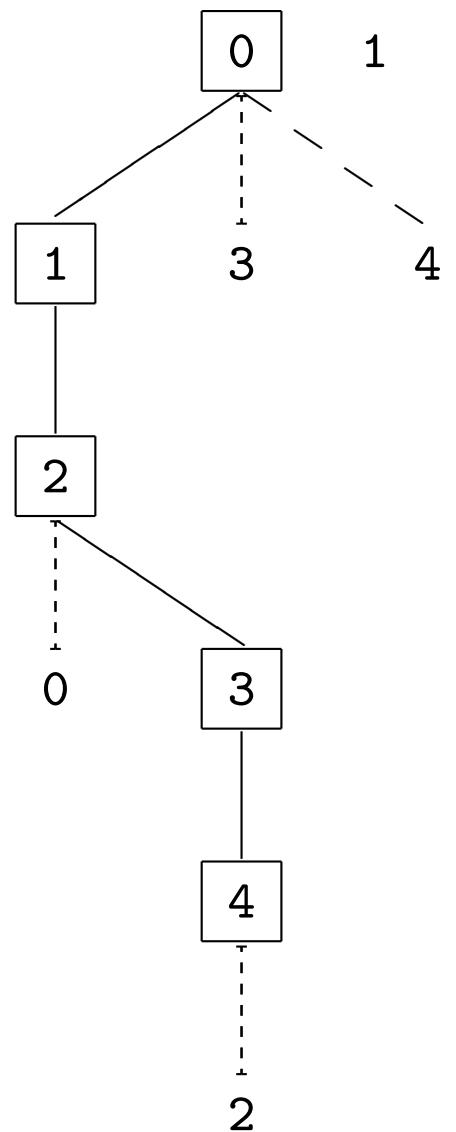


Ordem:

0 1 2 3 4

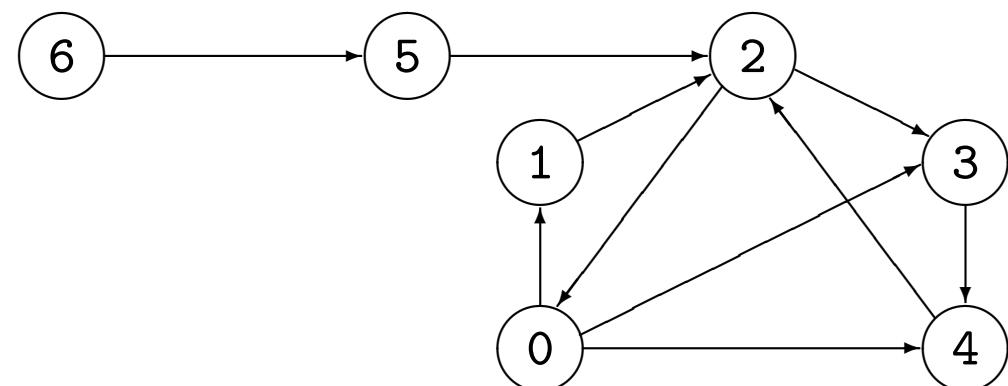


Percorso em Profundidade

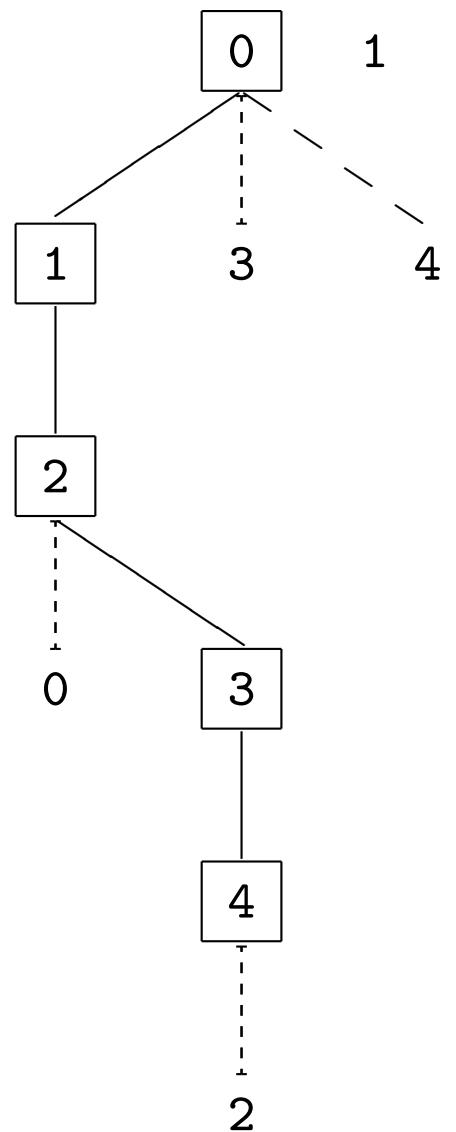


Ordem:

0 1 2 3 4

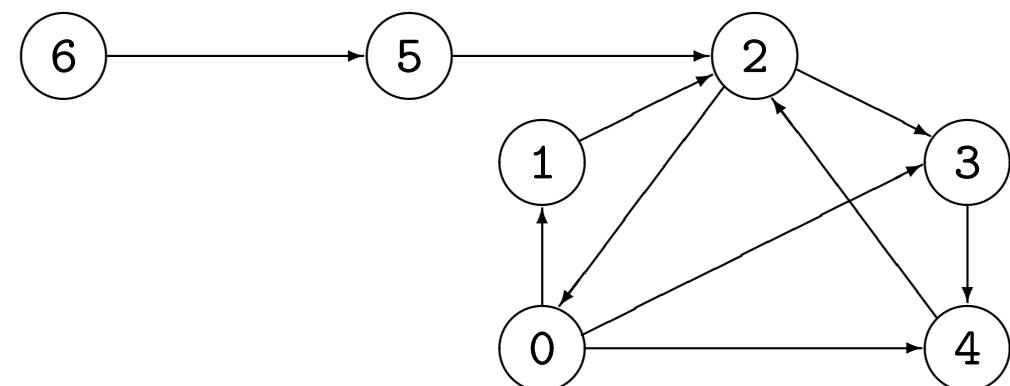


Percorso em Profundidade

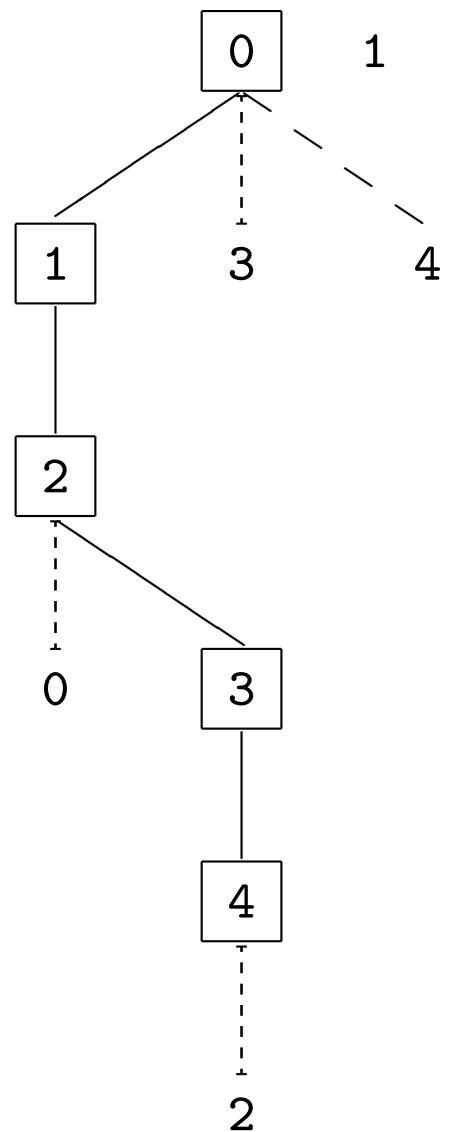


Ordem:

0 1 2 3 4

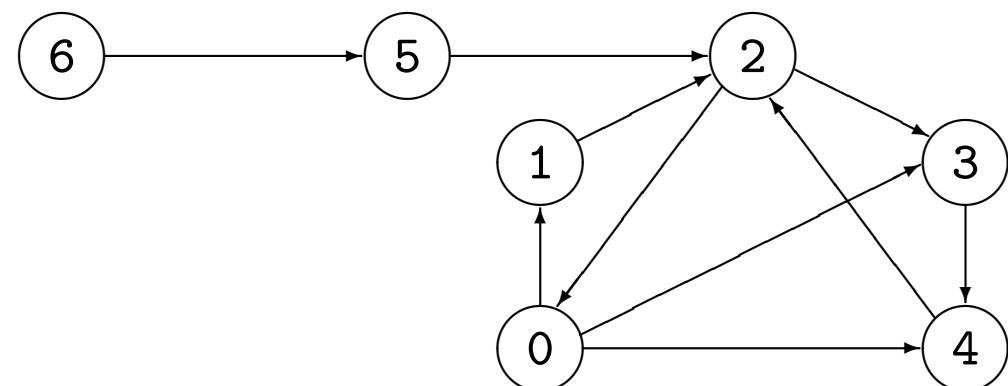


Percorso em Profundidade

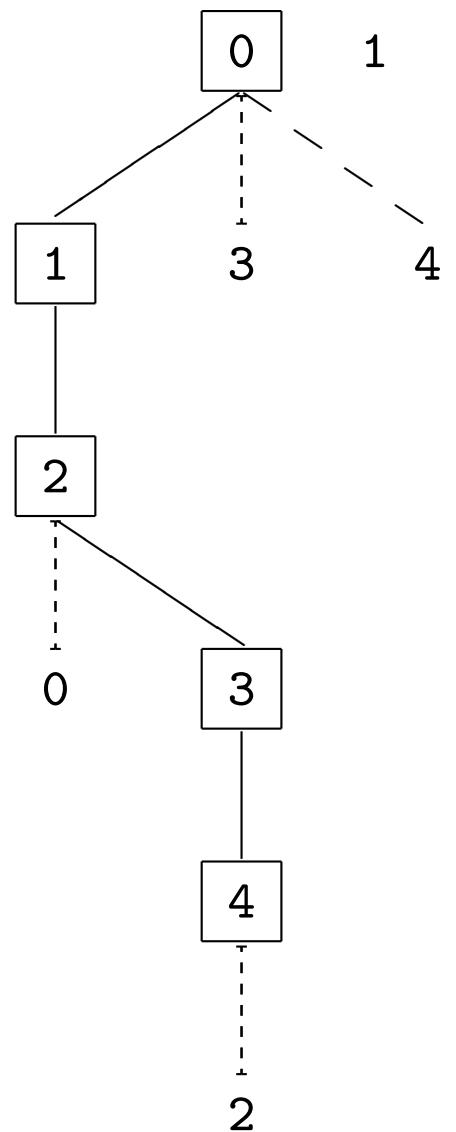


Ordem:

0 1 2 3 4 5

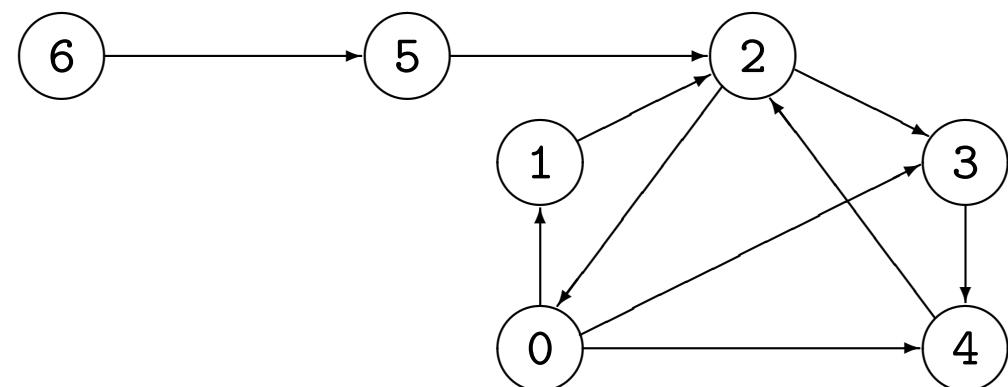


Percorso em Profundidade

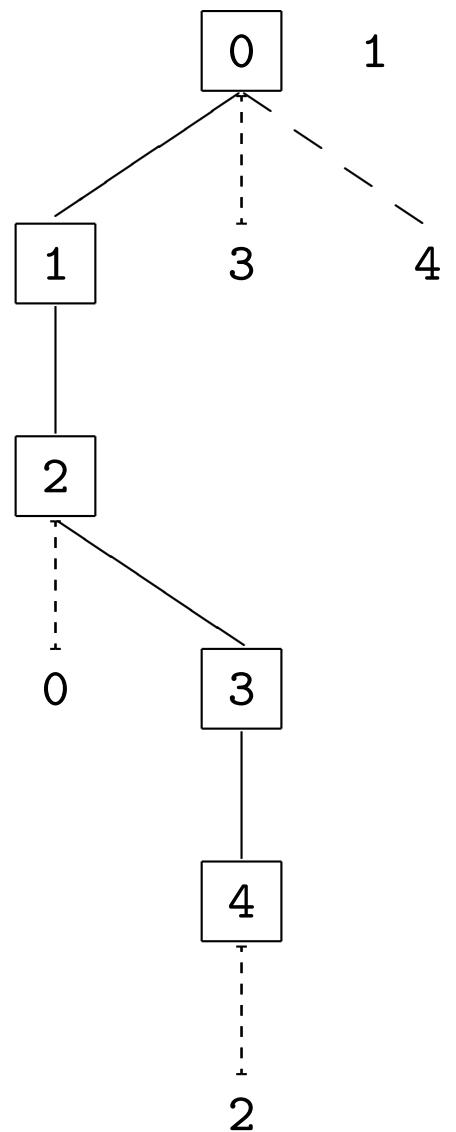


Ordem:

0 1 2 3 4 5

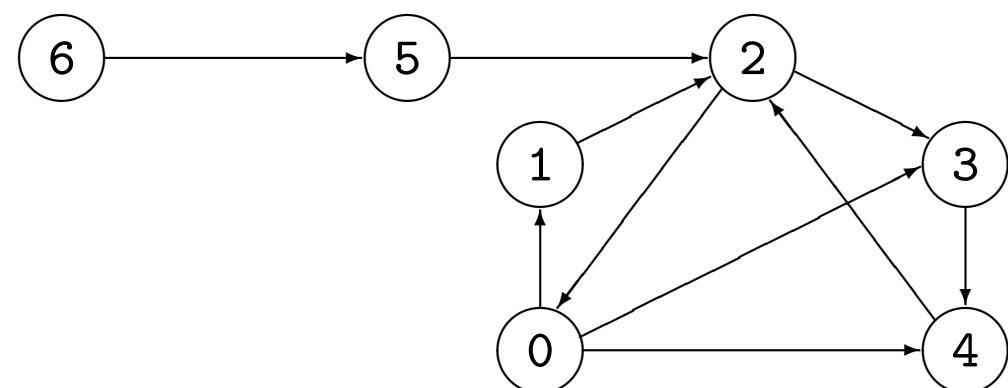


Percorso em Profundidade

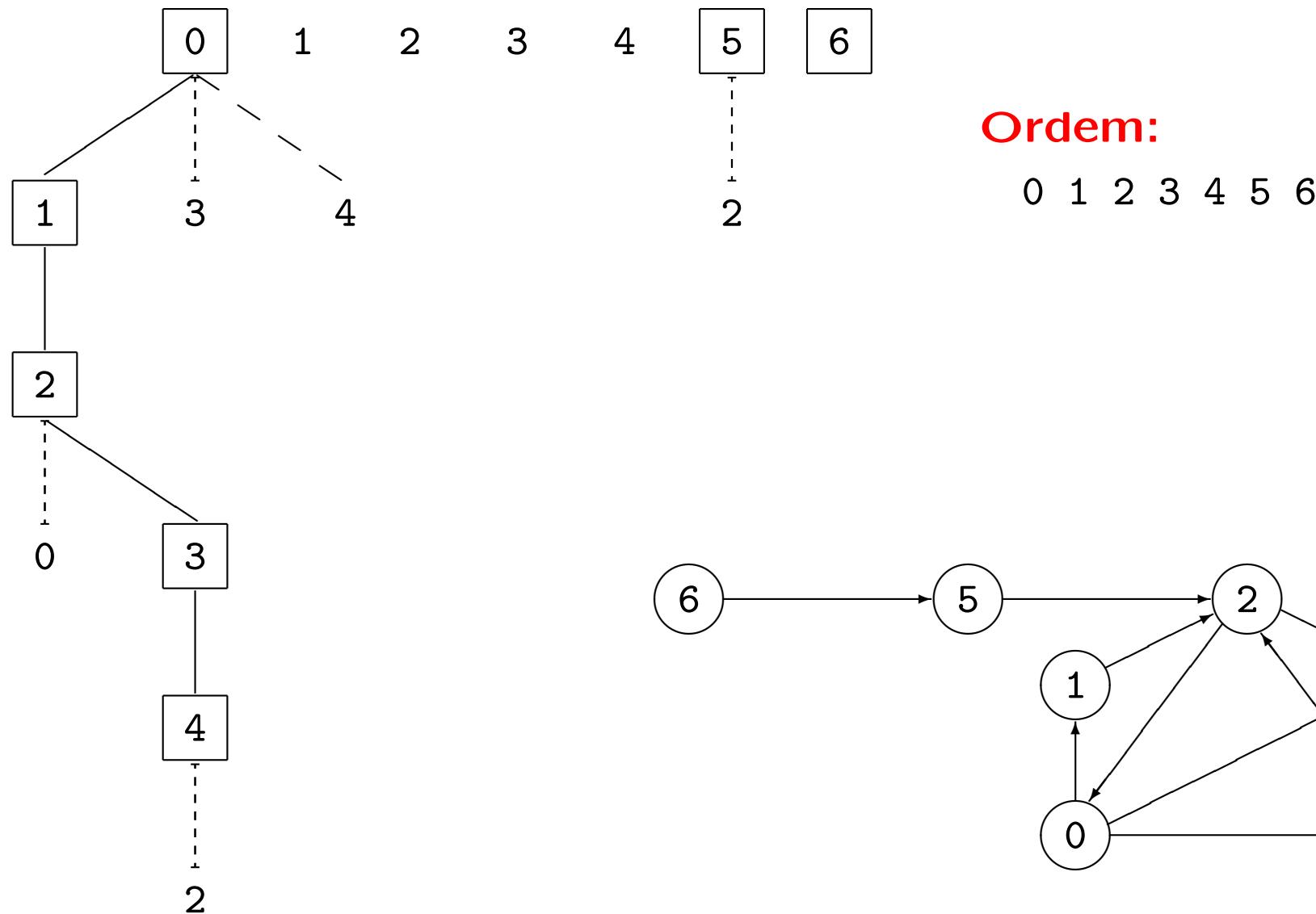


Ordem:

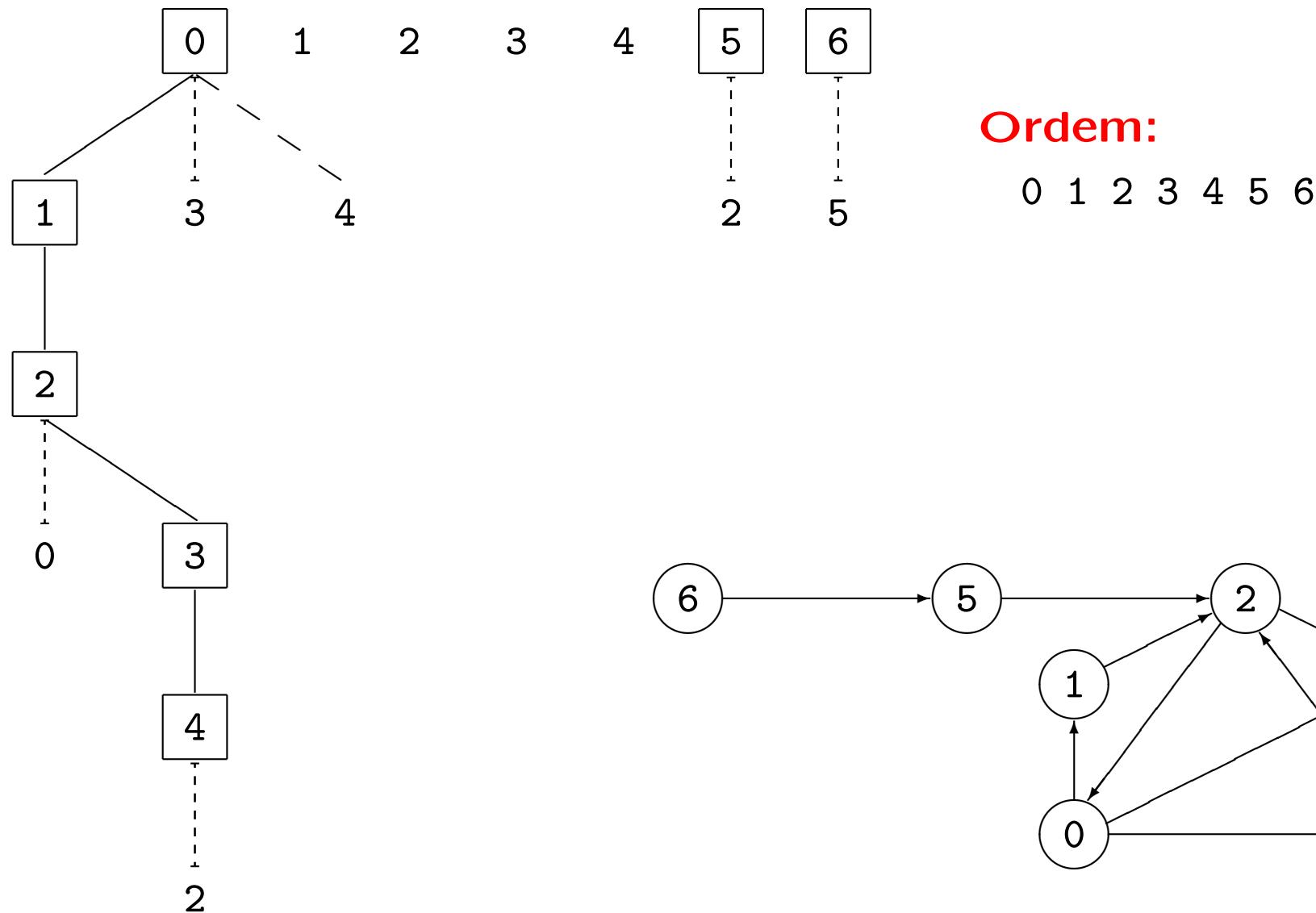
0 1 2 3 4 5



Percorso em Profundidade



Percorso em Profundidade



Percorso em Profundidade

(Depth-First Search Traversal)

```
void dfsTraversal( Digraph graph ) {  
  
    boolean[] processed = new boolean[ graph.numNodes() ];  
  
    for every Node v in graph.nodes()  
        processed[v] = false;  
  
    for every Node v in graph.nodes()  
        if ( !processed[v] )  
            dfsExplore(graph, processed, v);  
}
```

Árvore em Profundidade (recursivo)

```
void dfsExplore( Digraph graph, boolean[] processed, Node root ) {  
    // PROCESS(root)  
    processed[root] = true;  
  
    for every Node v in graph.outAdjacentNodes(root)  
        if ( !processed[v] )  
            dfsExplore(graph, processed, v);  
}
```

Complexidade de **dfsExplore** (recursivo)

- **Custo da uma chamada (vértice w)**

- Se $\text{PROCESS}(w)$ não for constante, considerar o seu custo
- Altera-se o booleano $\Theta(1)$
- Iteram-se os sucessores de w
 - * Grafo em matriz de adjacências $\Theta(|V|)$
 - * Grafo em vetor de listas de adjacências (suc.) $\Theta(|\text{Suc}(w)|)$

- **Custo de todas as chamadas (uma por cada vértice)**

- Grafo em matriz de adjacências $\Theta(|V|^2)$
- Grafo em vetor de listas de adjacências (suc.) $\Theta(|V| + |A|)$
 - Num grafo orientado, $\sum_{w \in V} |\text{Suc}(w)| = |A|$.
 - Num grafo não orientado, $\sum_{w \in V} |\text{Suc}(w)| = 2 \times |A|$.

Complexidade Temporal de **dfsTraversal** (rec)

Grafo em matriz de adjacências

Criação do vetor processed	$\Theta(1)$
1º ciclo (inicialização do vetor processed)	$\Theta(V)$
2º ciclo (ignorando execuções de dfsExplore)	$\Theta(V)$
Execuções de dfsExplore	$\Theta(V ^2)$
TOTAL	$\Theta(V ^2)$

Grafo em vetor de listas de adjacências (suc.)

Criação do vetor processed	$\Theta(1)$
1º ciclo (inicialização do vetor processed)	$\Theta(V)$
2º ciclo (ignorando execuções de dfsExplore)	$\Theta(V)$
Execuções de dfsExplore	$\Theta(V + A)$
TOTAL	$\Theta(V + A)$

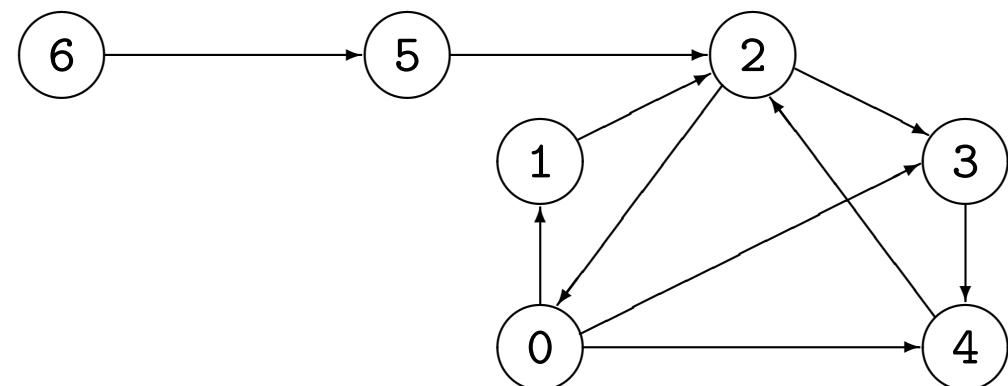
Complexidade Espacial de **dfsTraversal** (rec)

Vetor processed	$\Theta(V)$
Pilha de chamadas recursivas	$O(V)$
TOTAL	$\Theta(V)$

Percorso em Profundidade

0

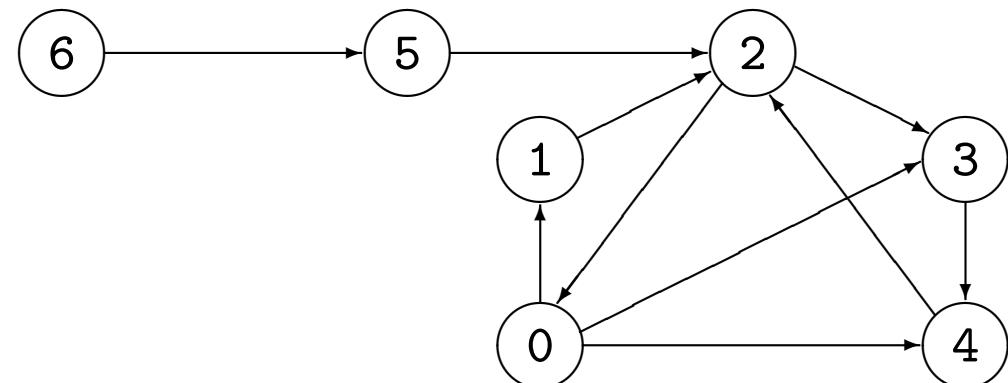
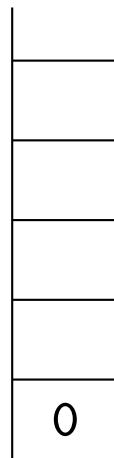
Ordem:



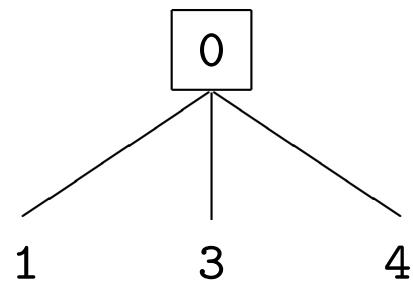
Percorso em Profundidade

0

Ordem:



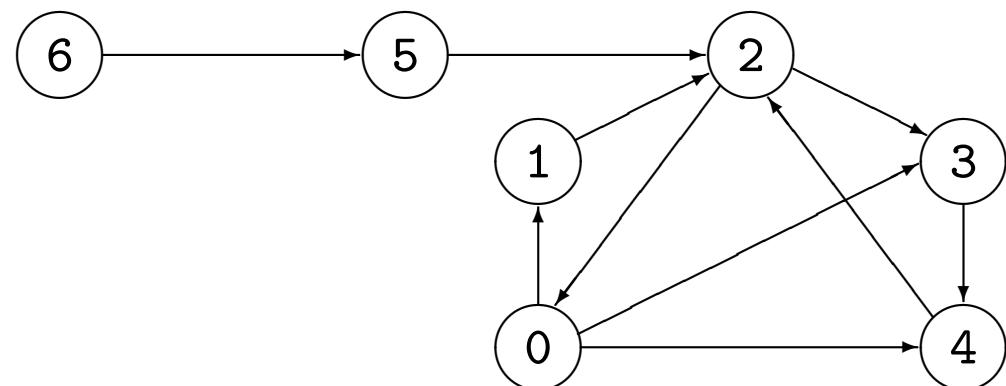
Percorso em Profundidade



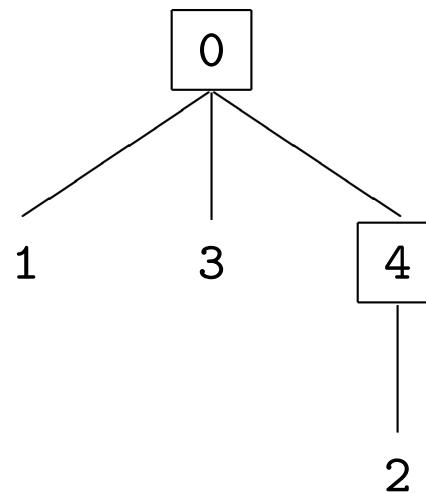
Ordem:

0

4
3
1

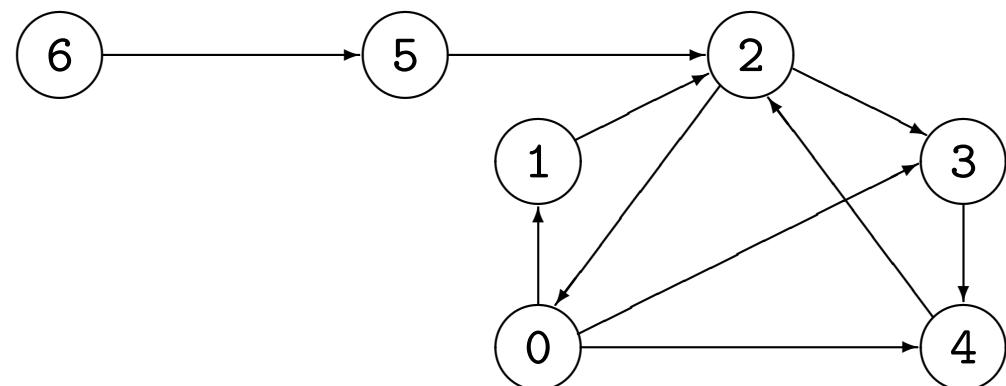


Percorso em Profundidade

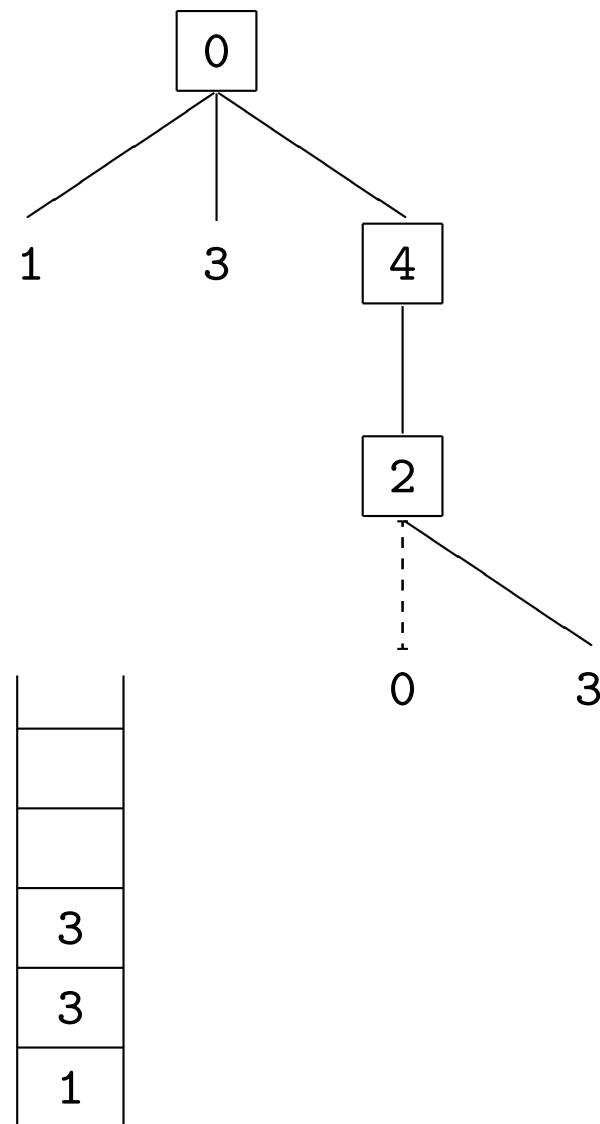


Ordem:

0 4

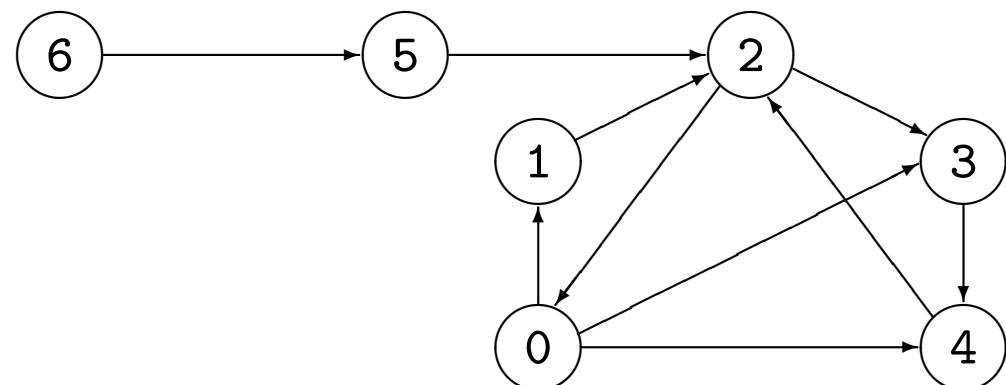


Percorso em Profundidade

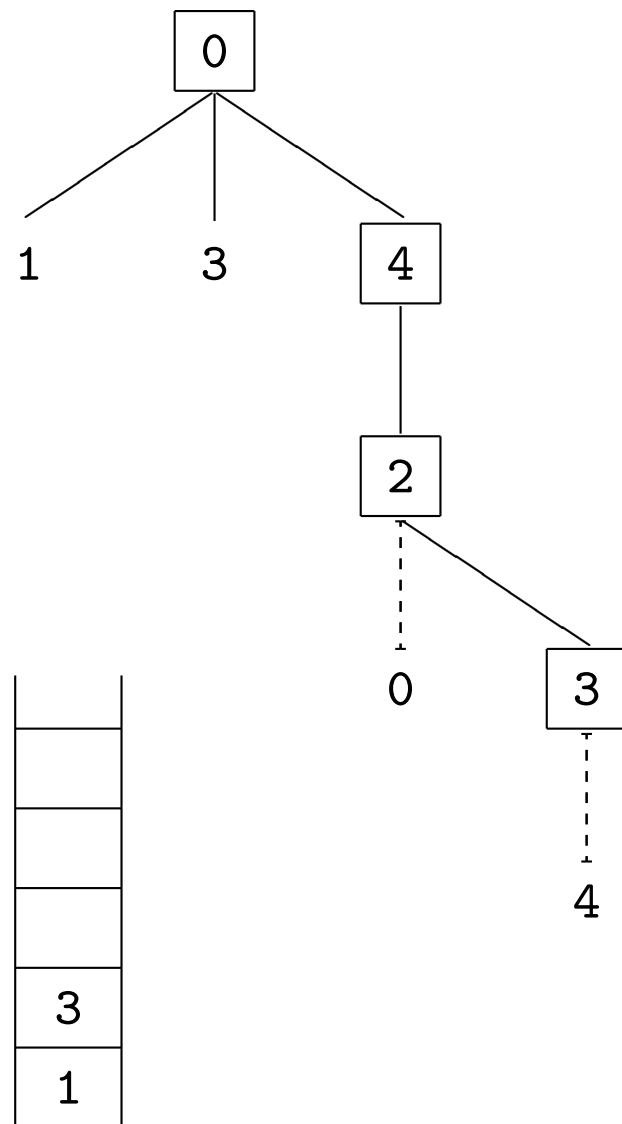


Ordem:

0 4 2

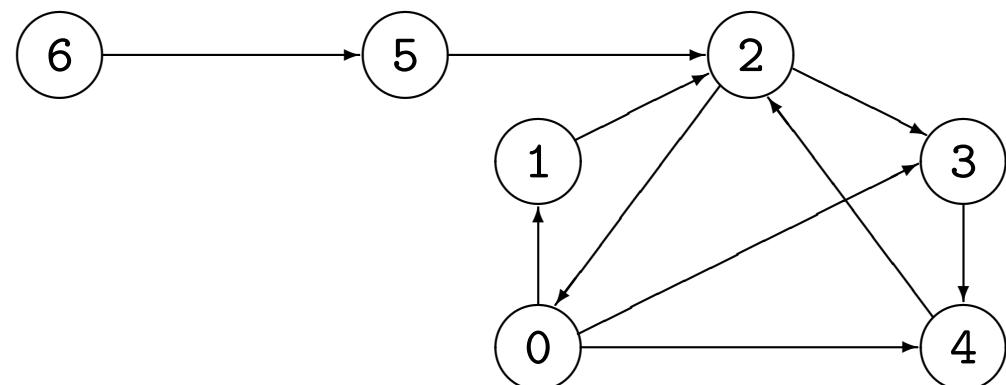


Percorso em Profundidade

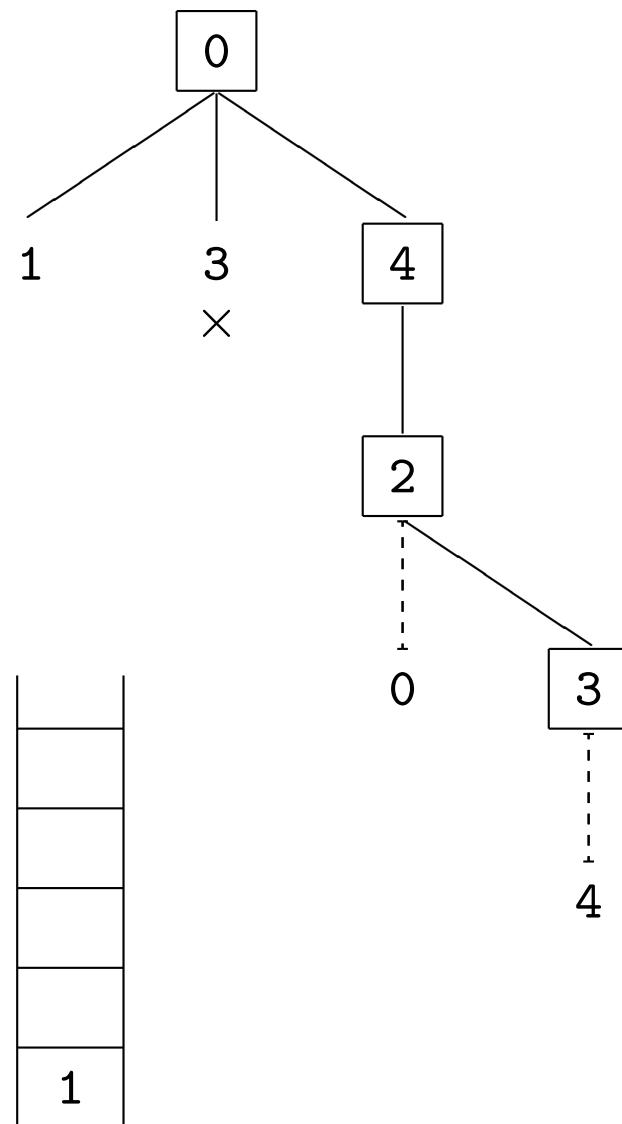


Ordem:

0 4 2 3

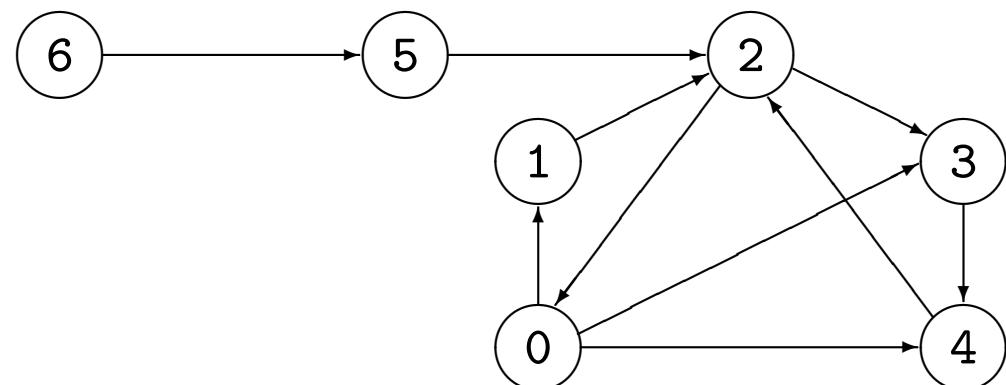


Percorso em Profundidade

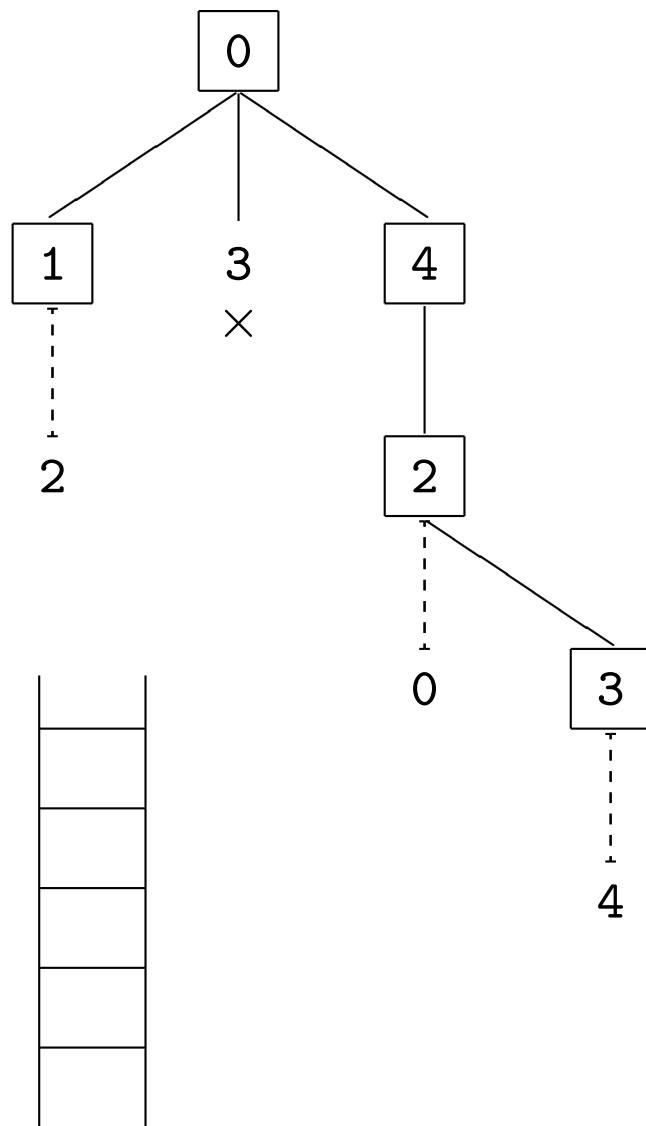


Ordem:

0 4 2 3

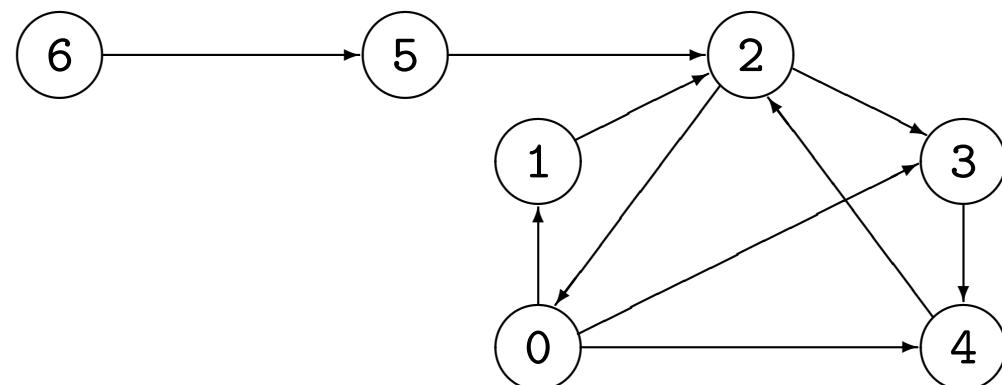


Percorso em Profundidade

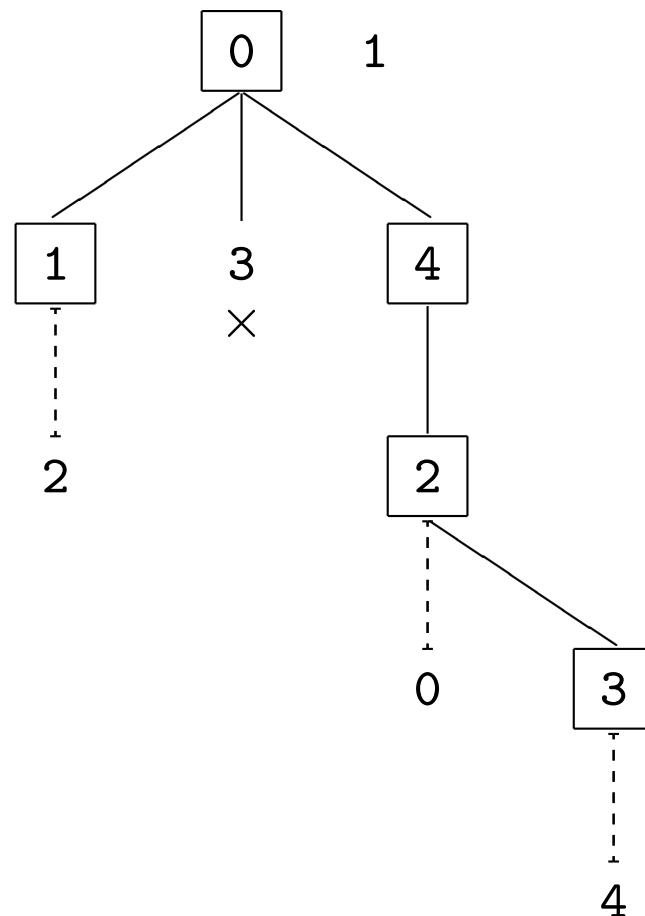


Ordem:

0 4 2 3 1

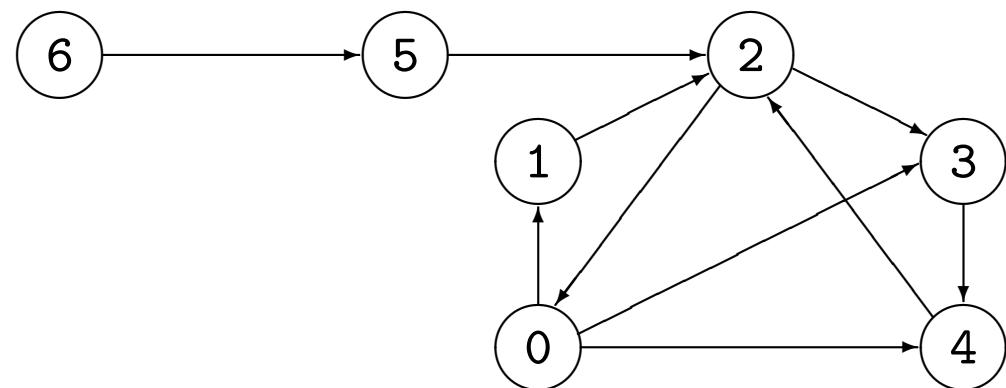


Percorso em Profundidade

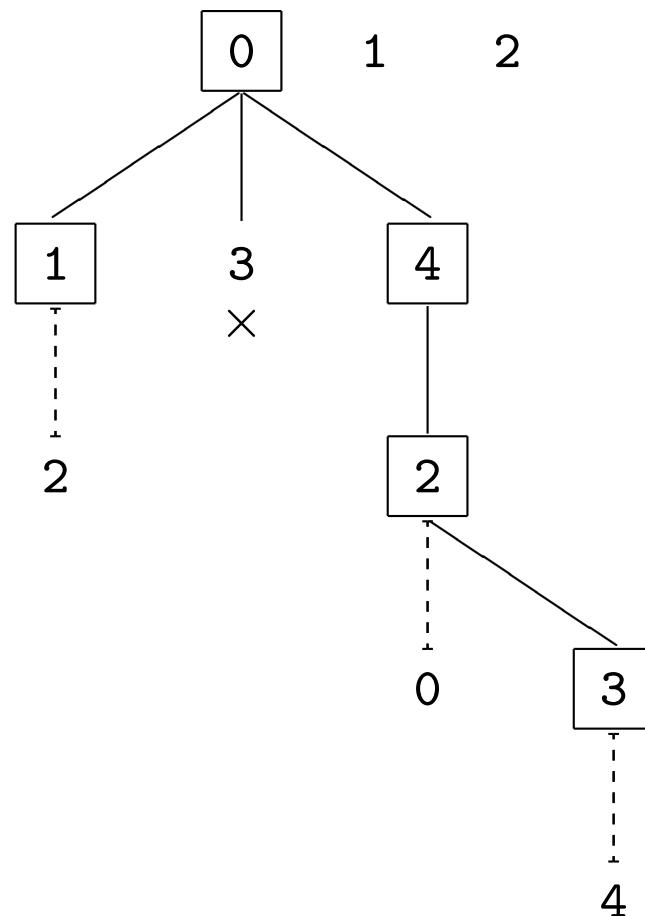


Ordem:

0 4 2 3 1

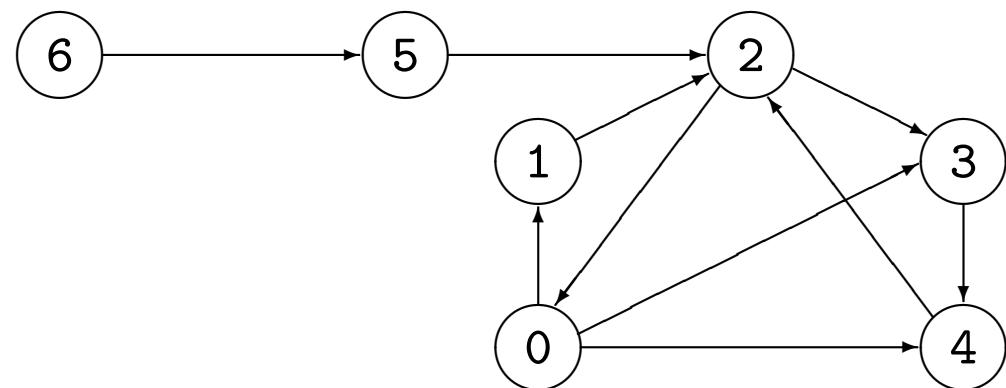


Percorso em Profundidade

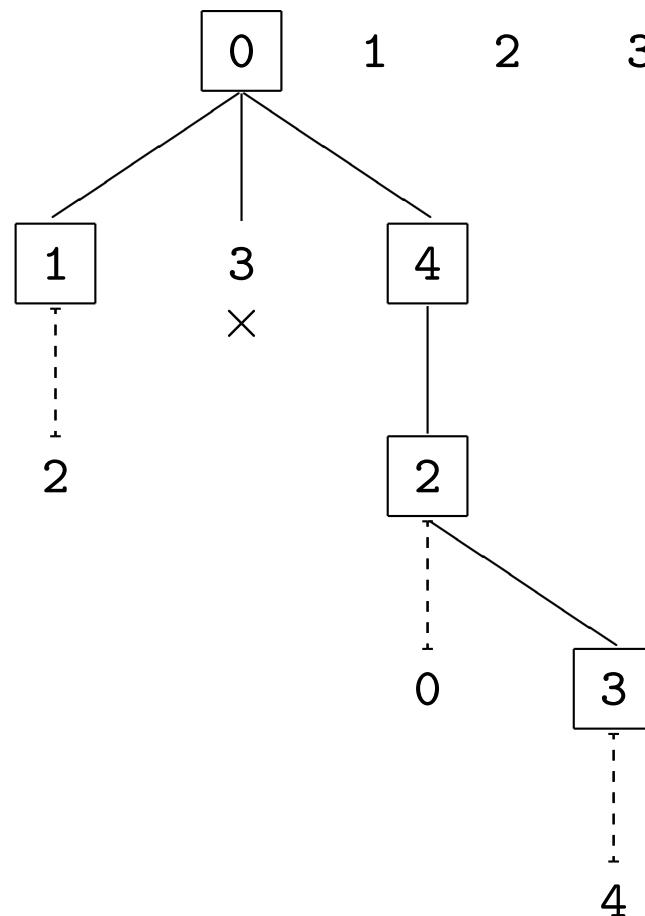


Ordem:

0 4 2 3 1

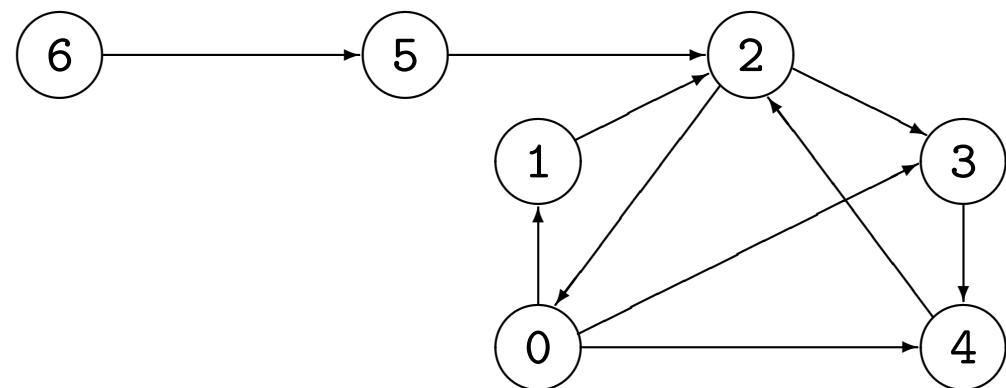


Percorso em Profundidade

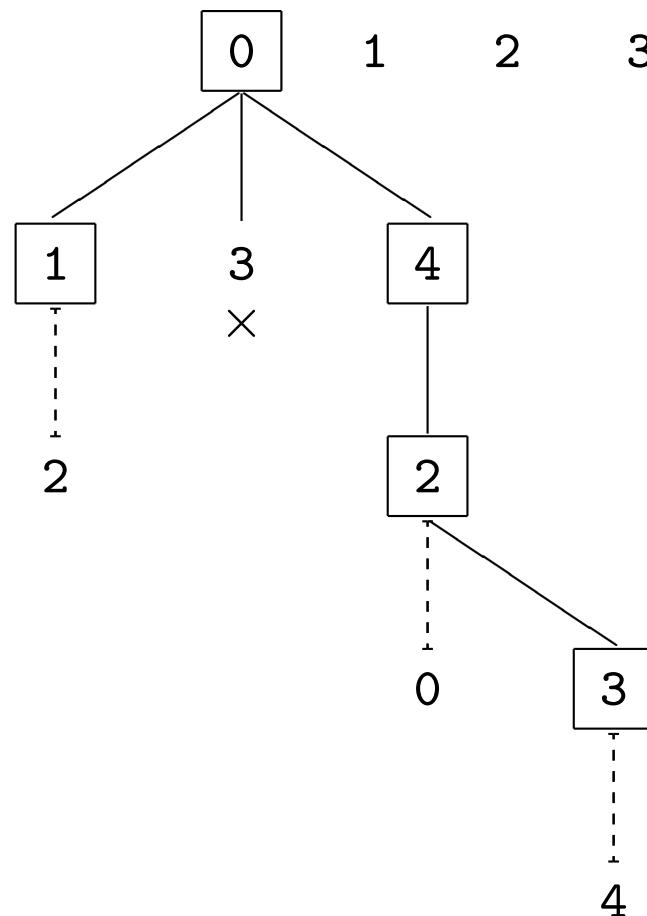


Ordem:

0 4 2 3 1

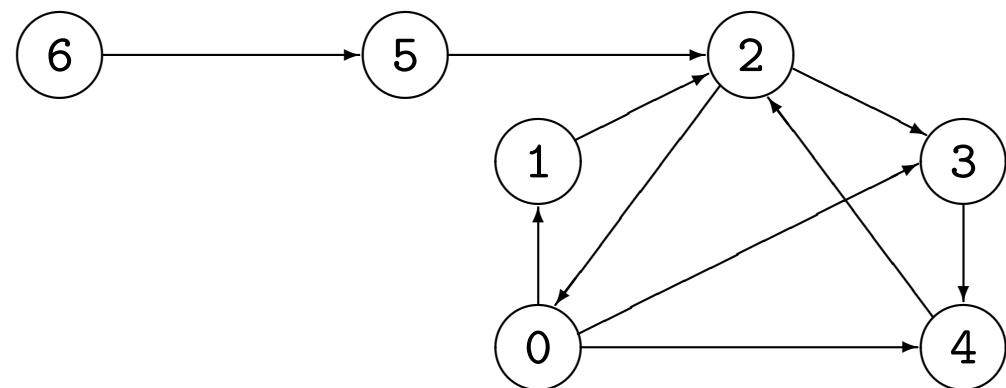


Percorso em Profundidade

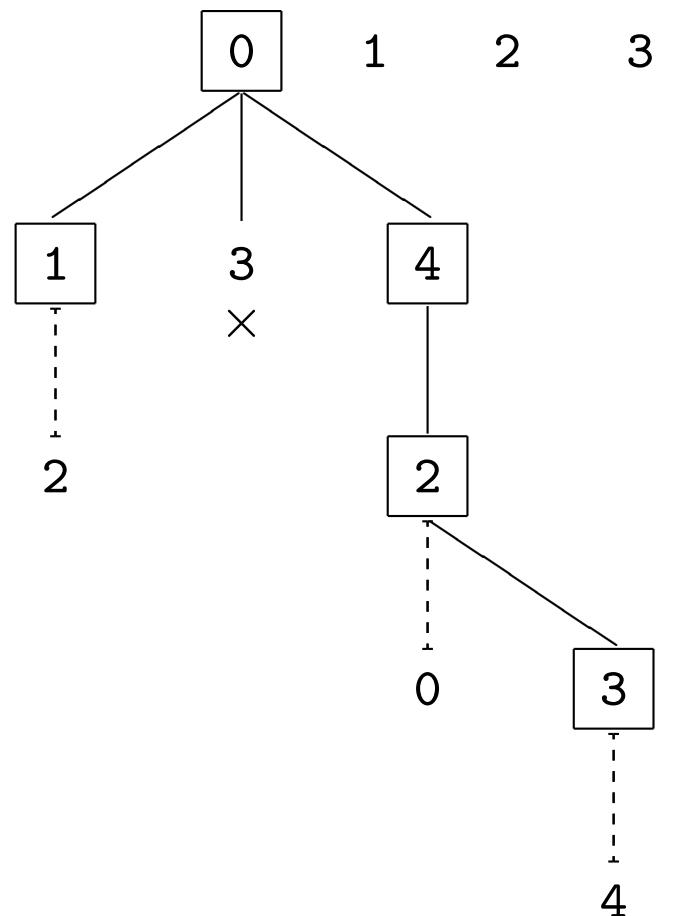


Ordem:

0 4 2 3 1

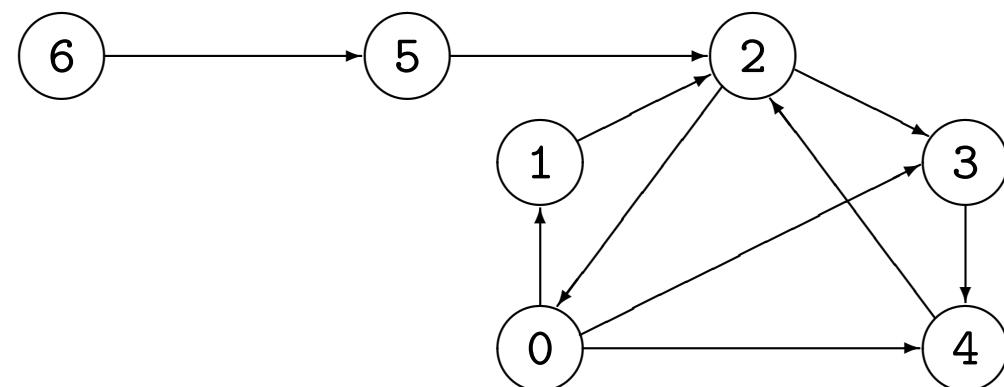


Percorso em Profundidade

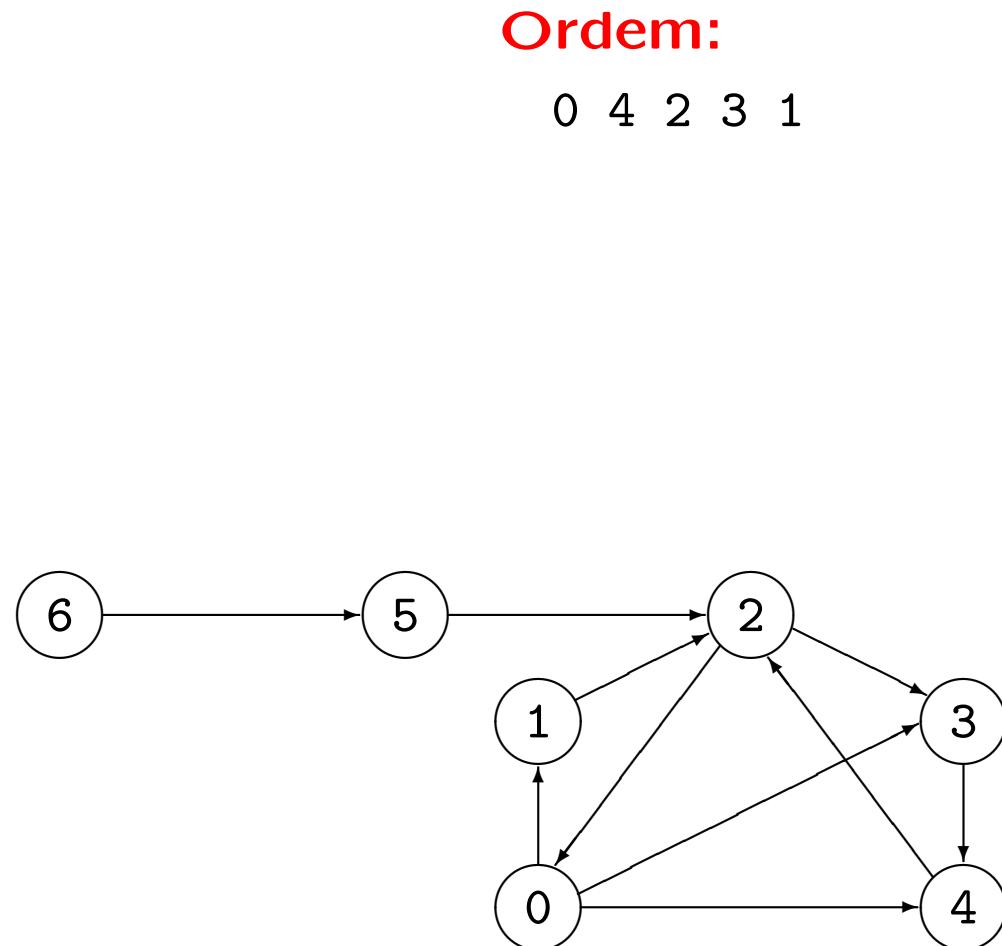
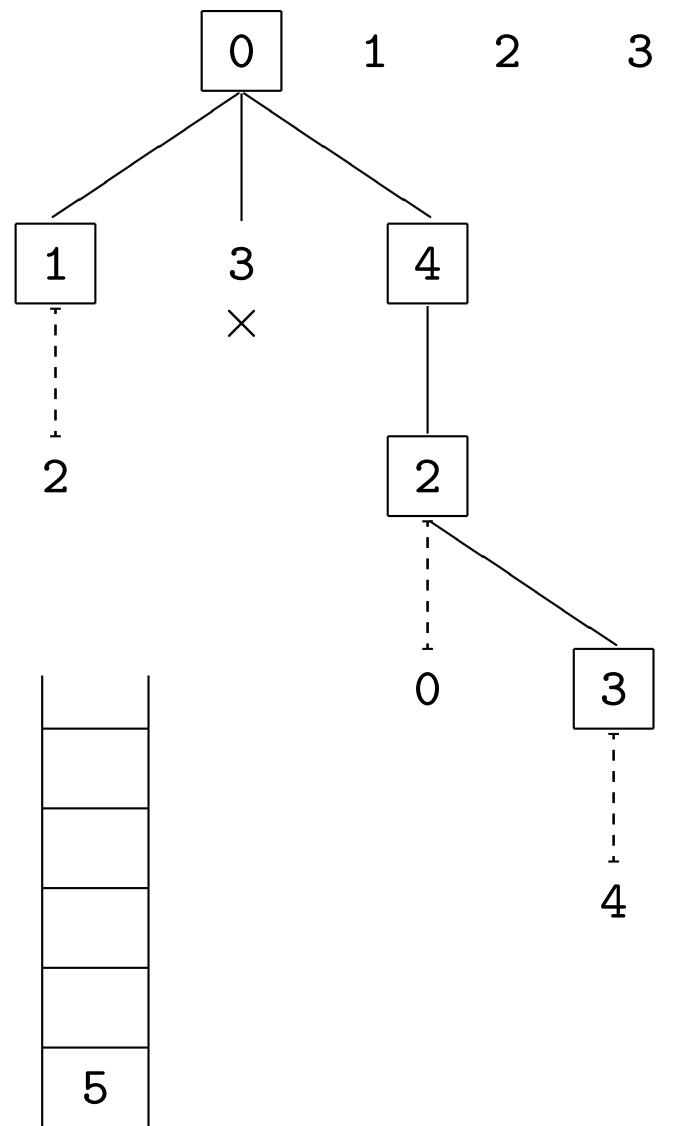


Ordem:

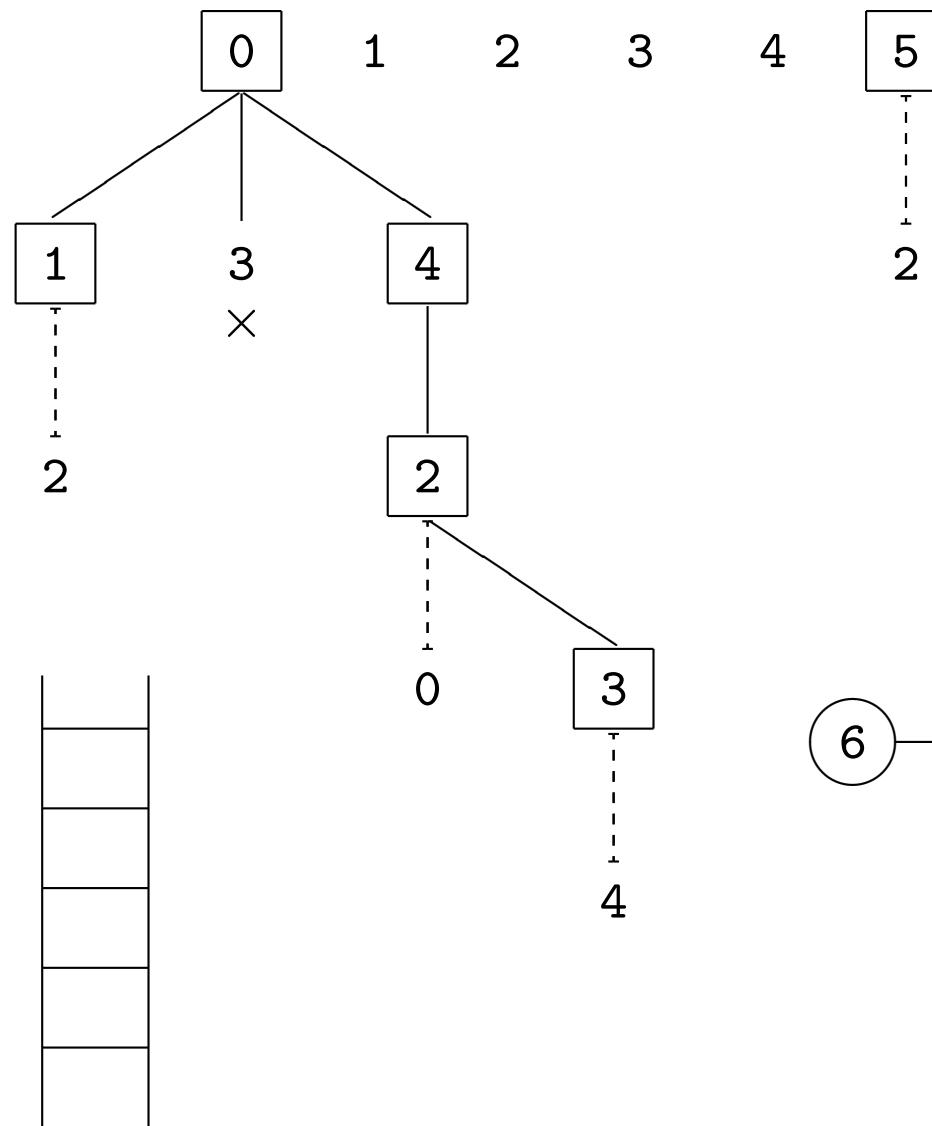
0 4 2 3 1



Percorso em Profundidade

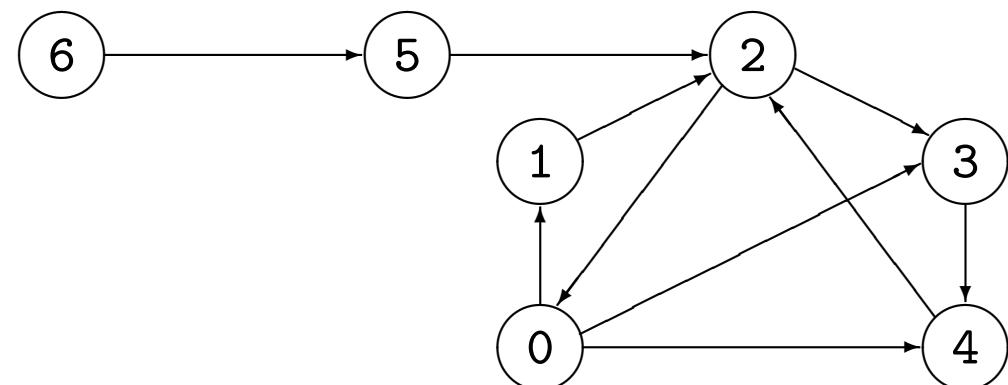


Percorso em Profundidade

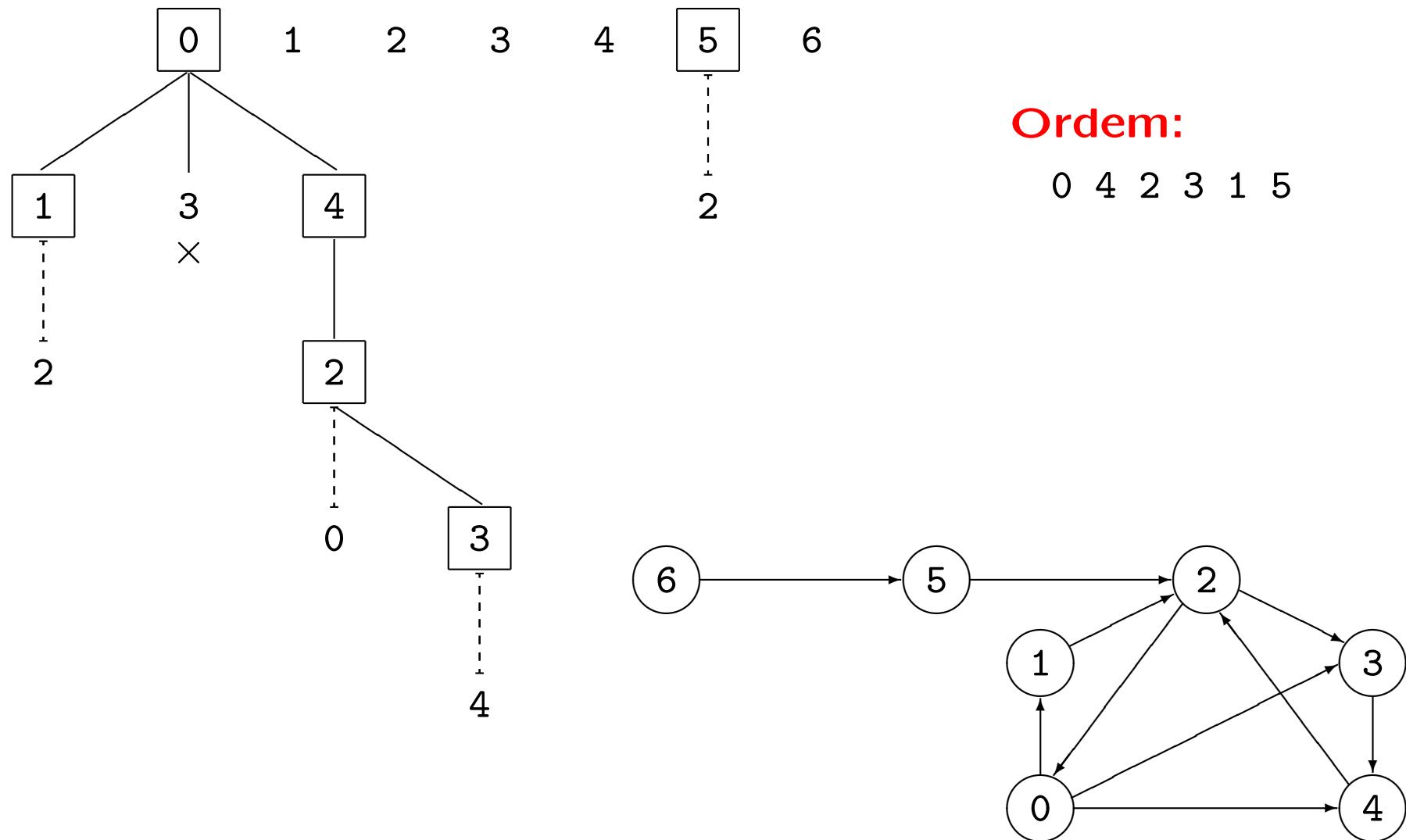


Ordem:

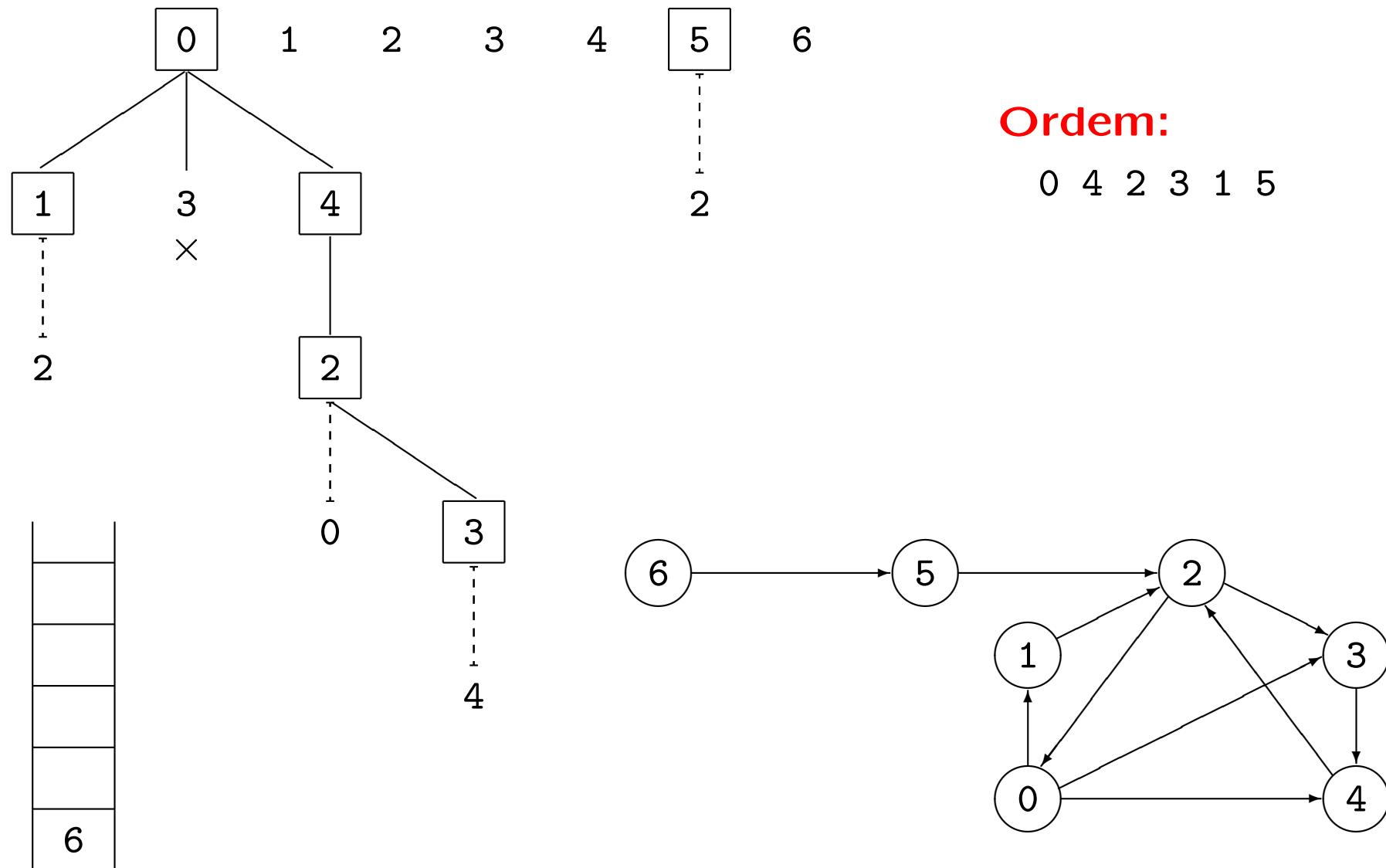
0 4 2 3 1 5



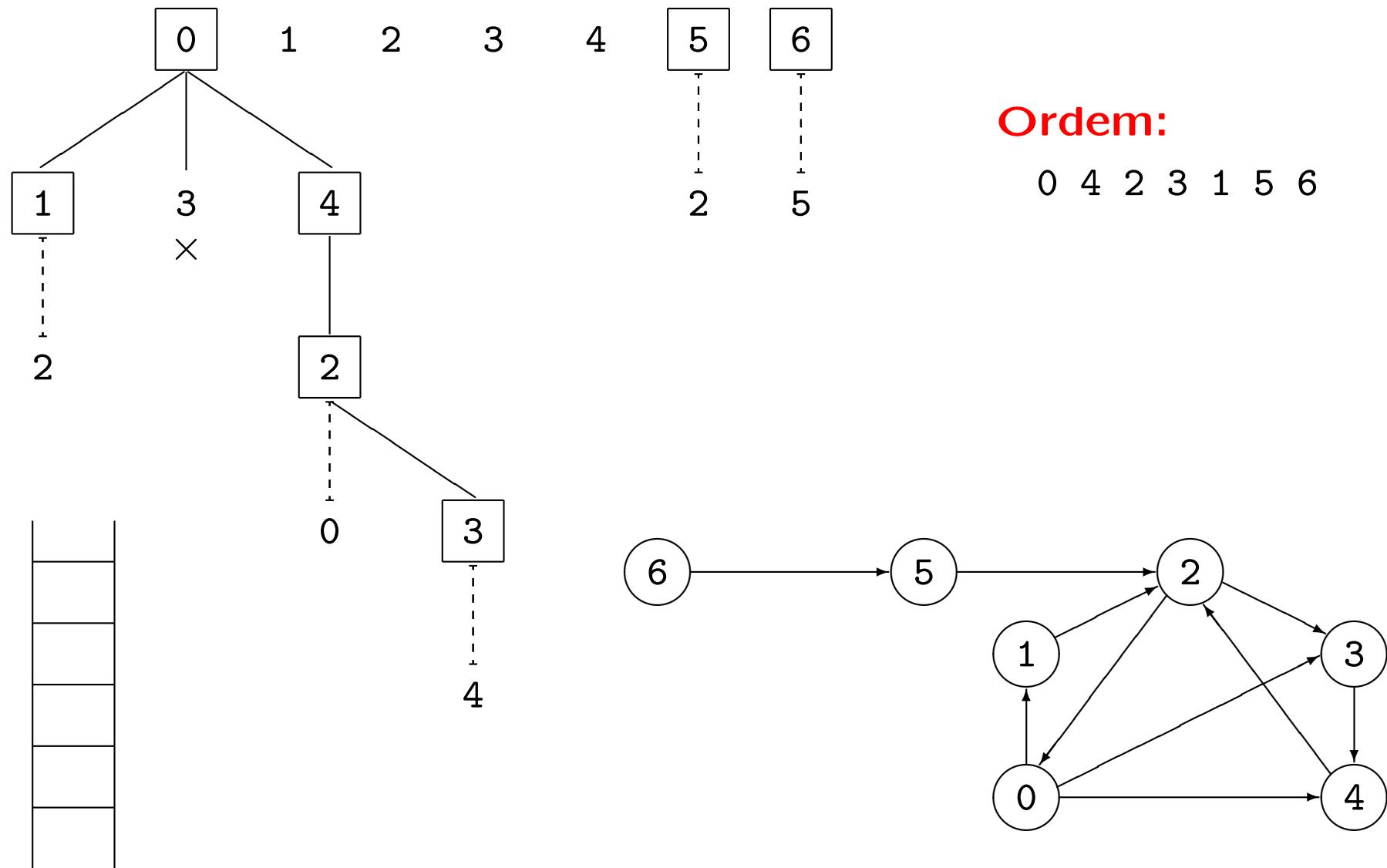
Percorso em Profundidade



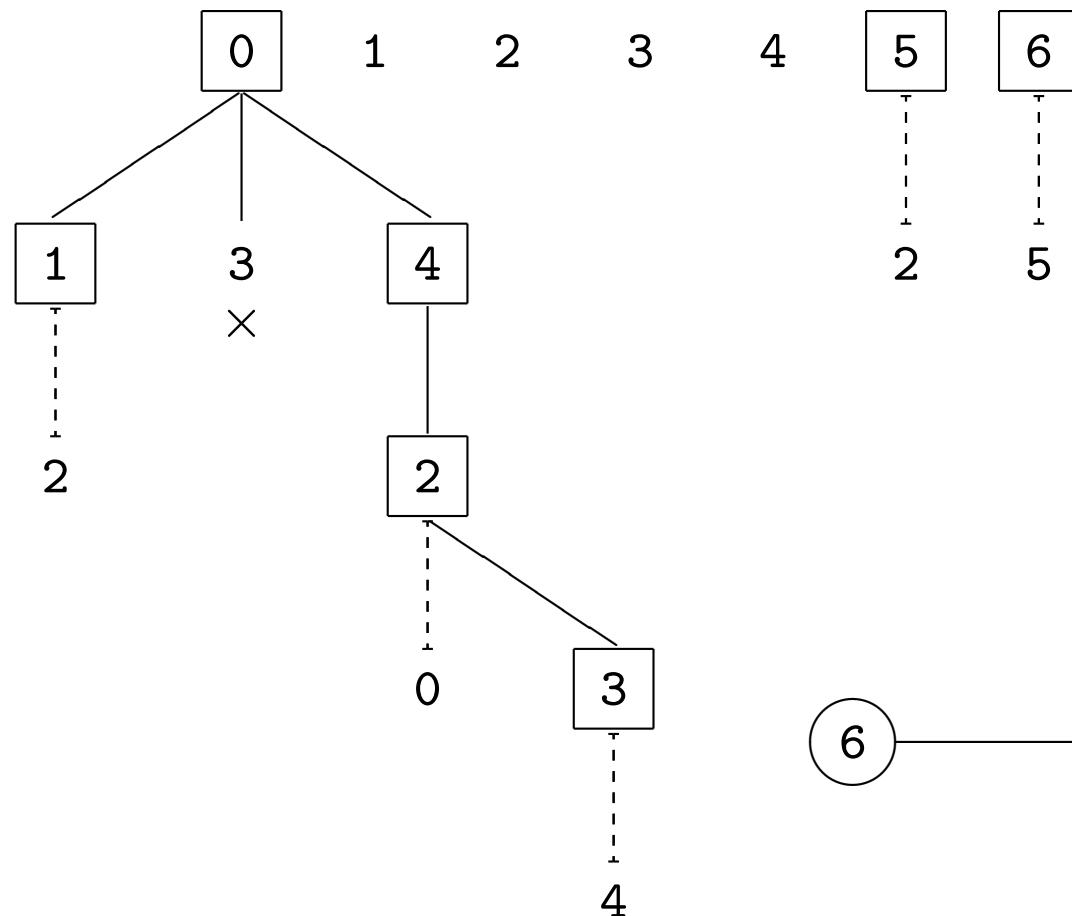
Percorso em Profundidade



Percorso em Profundidade

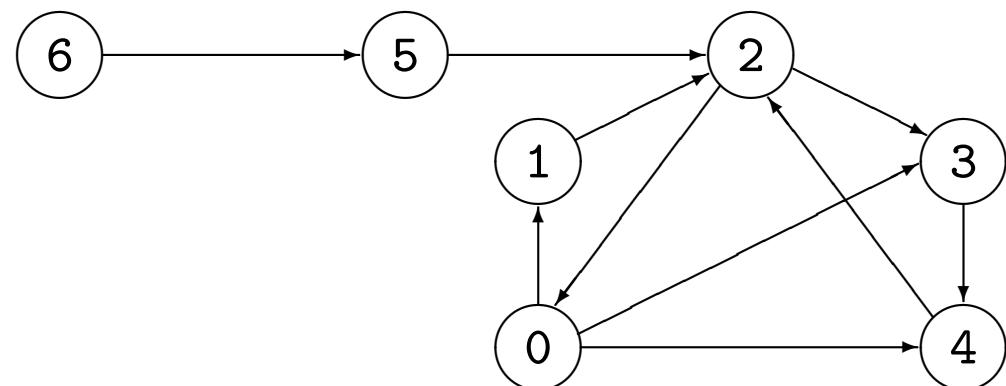


Percorso em Profundidade



Ordem:

0 4 2 3 1 5 6



Percorso em Profundidade

(Depth-First Search Traversal)

```
void dfsTraversal( Digraph graph ) {  
  
    boolean[] processed = new boolean[ graph.numNodes() ];  
  
    for every Node v in graph.nodes()  
        processed[v] = false;  
  
    for every Node v in graph.nodes()  
        if ( !processed[v] )  
            dfsExplore(graph, processed, v);  
}
```

Árvore em Profundidade (iterativo)

```
void dfsExplore( Digraph graph, boolean[] processed, Node root ) {  
    Stack<Node> foundUnprocessed = new StackIn...<>( ? );  
    foundUnprocessed.push(root);  
    do {  
        Node node = foundUnprocessed.pop();  
        if ( !processed[node] ) {  
            // PROCESS(node)  
            processed[node] = true;  
            for every Node v in graph.outAdjacentNodes(node)  
                if ( !processed[v] )  
                    foundUnprocessed.push(v);  
        }  
    }  
    while ( !foundUnprocessed.isEmpty() );  
}
```

Complexidade de **dfsExplore** (iterativo)

- **Por cada vértice (w)**
 - Se $\text{PROCESS}(w)$ não for constante, considerar o seu custo
 - Altera-se o booleano $\Theta(1)$
 - Iteram-se os sucessores de w
 - * Grafo em matriz de adjacências $\Theta(|V|)$
 - * Grafo em vetor de listas de adjacências (suc.) $\Theta(|\text{Suc}(w)|)$
 - Empilham-se os sucessores não processados de w $O(|\text{Suc}(w)|)$
- **Custo de todas as execuções ($\#\text{POP} = \#\text{PUSH}$)**
 - Grafo em matriz de adjacências $\Theta(|V|^2)$
 - Grafo em vetor de listas de adjacências (suc.) $\Theta(|V| + |A|)$
 - Num grafo orientado, $\sum_{w \in V} |\text{Suc}(w)| = |A|$.
 - Num grafo não orientado, $\sum_{w \in V} |\text{Suc}(w)| = 2 \times |A|$.

Complexidade Temporal de **dfsTraversal** (iter)

(se push, pop e isEmpty de Stack forem $\Theta(1)$)

Grafo em matriz de adjacências

Criação do vetor processed	$\Theta(1)$
1º ciclo (inicialização do vetor processed)	$\Theta(V)$
2º ciclo (ignorando execuções de dfsExplore)	$\Theta(V)$
Execuções de dfsExplore	$\Theta(V ^2)$
TOTAL	$\Theta(V ^2)$

Grafo em vetor de listas de adjacências (suc.)

Criação do vetor processed	$\Theta(1)$
1º ciclo (inicialização do vetor processed)	$\Theta(V)$
2º ciclo (ignorando execuções de dfsExplore)	$\Theta(V)$
Execuções de dfsExplore	$\Theta(V + A)$
TOTAL	$\Theta(V + A)$

Complexidade Espacial de **dfsTraversal** (iter)

Vetor processed

$\Theta(|V|)$

Pilha foundUnprocessed

$O(|A|)$

TOTAL

$O(|V| + |A|)$

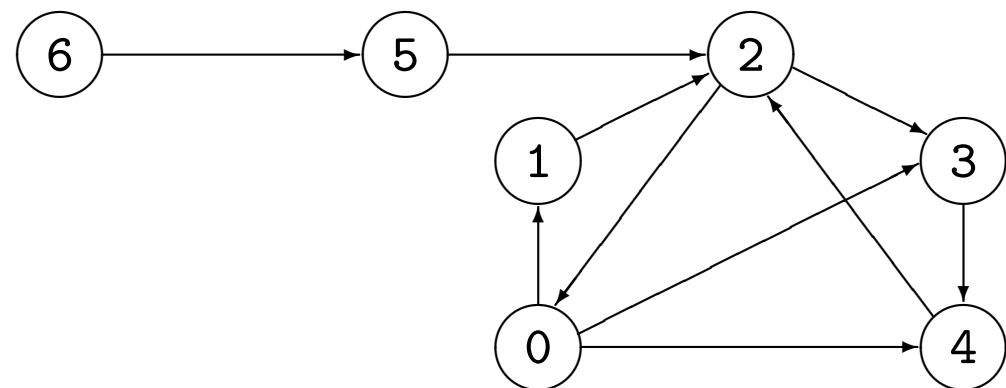
Capítulo IV

Percorso em Largura (num grafo orientado ou não orientado)

Percorso em Largura

0

Ordem:



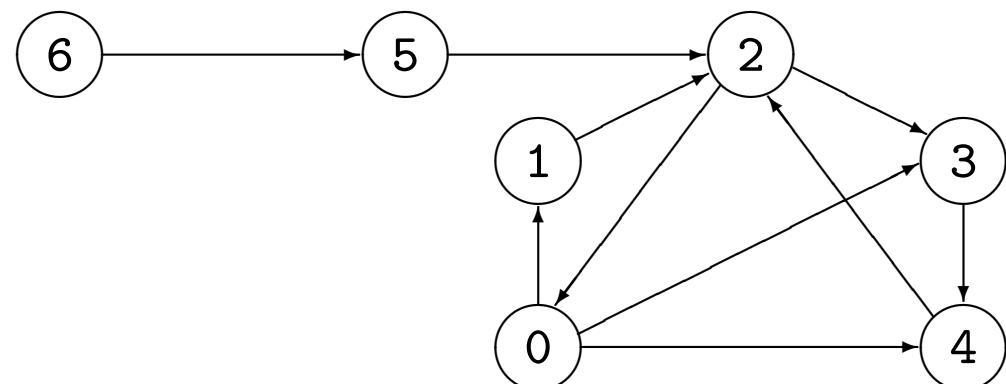
Percorso em Largura

0

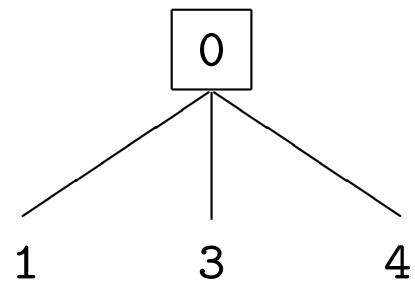
Ordem:

FIFO:

0



Percorso em Largura

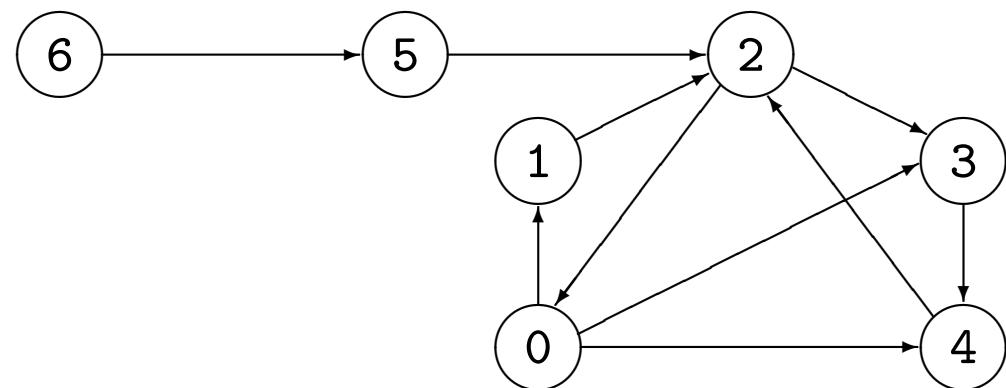


Ordem:

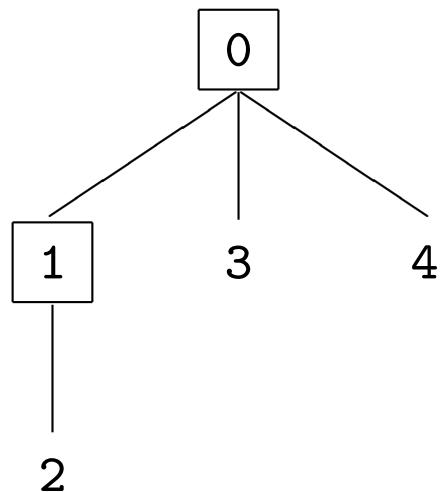
0

FIFO:

1 3 4



Percorso em Largura

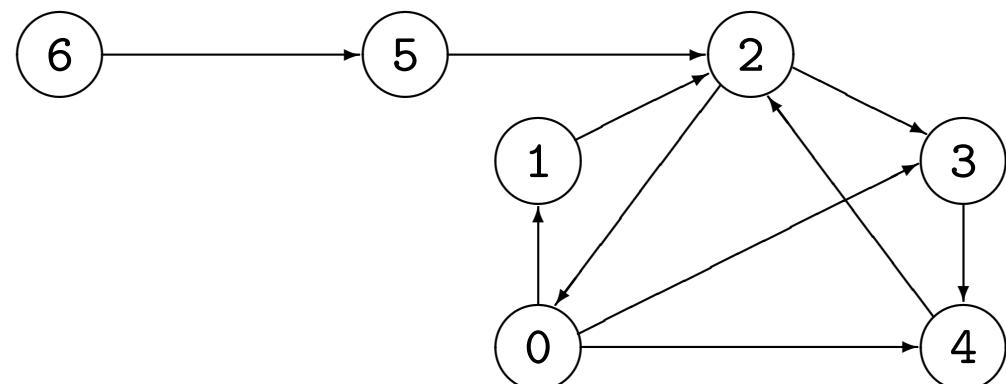


Ordem:

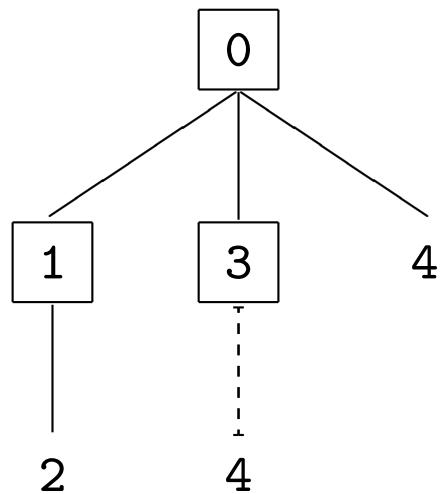
0 1

FIFO:

3 4 2



Percorso em Largura

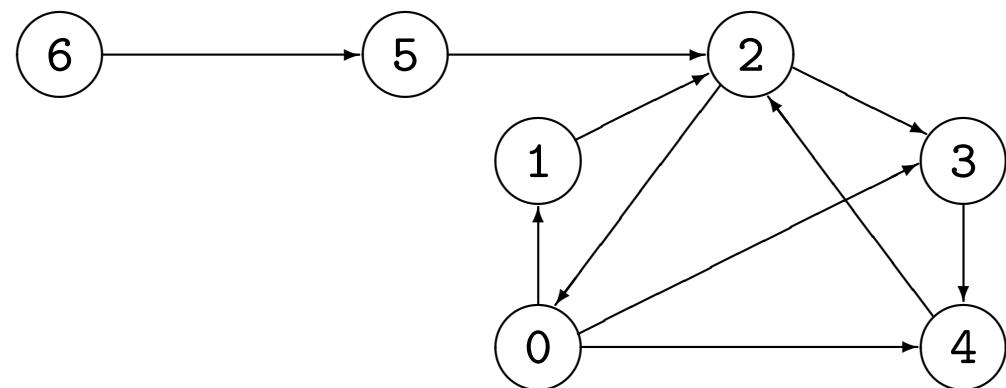


Ordem:

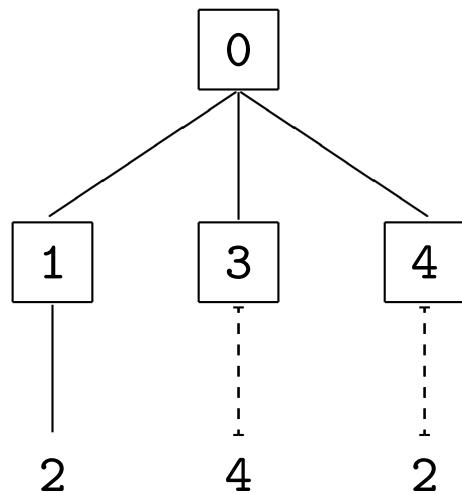
0 1 3

FIFO:

4 2



Percorso em Largura

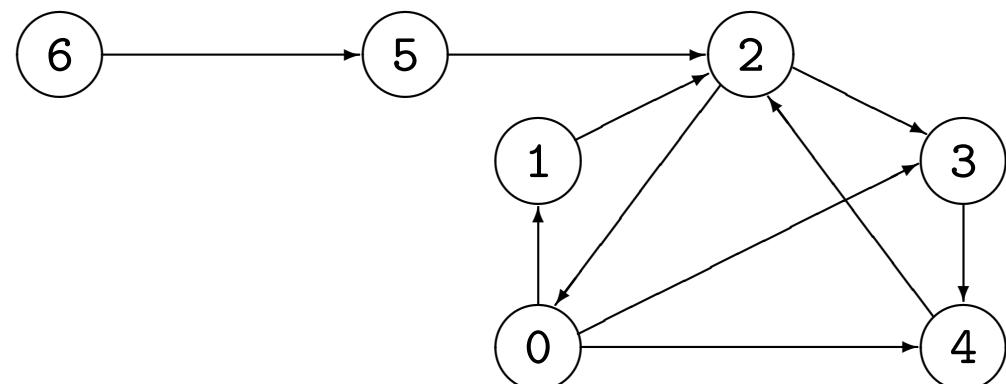


Ordem:

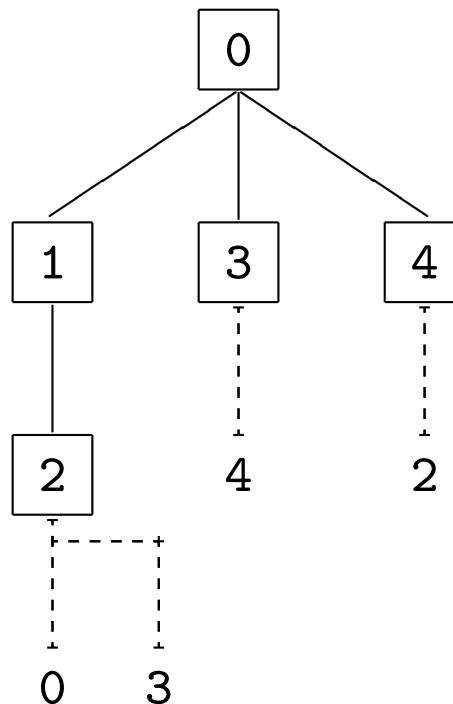
0 1 3 4

FIFO:

2



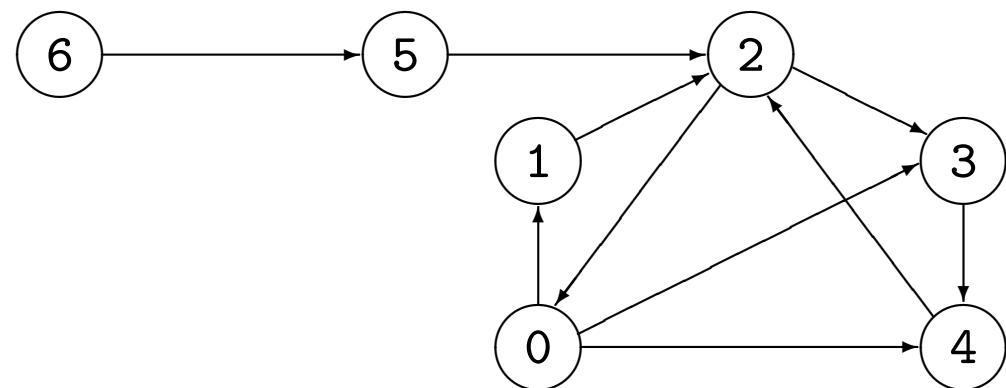
Percorso em Largura



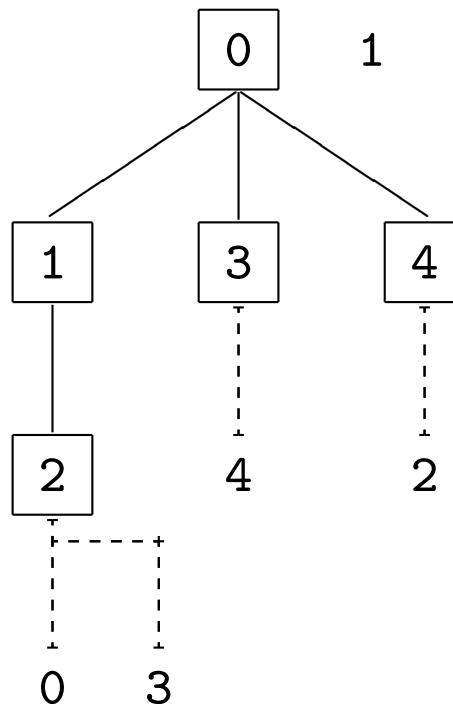
Ordem:

0 1 3 4 2

FIFO:

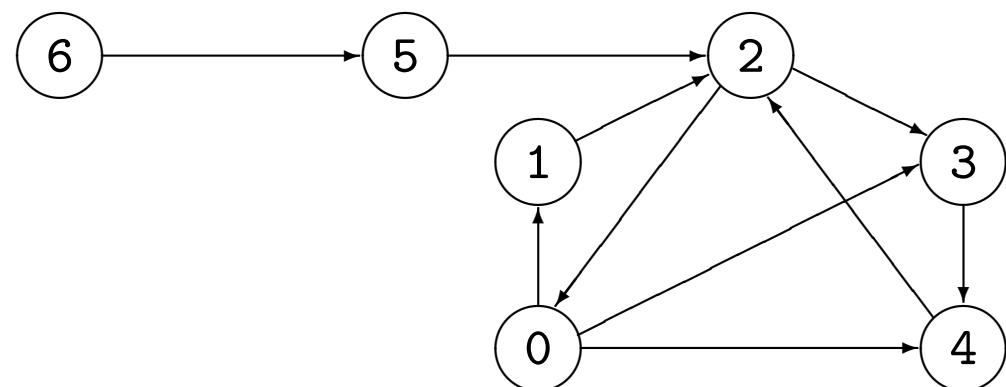


Percorso em Largura

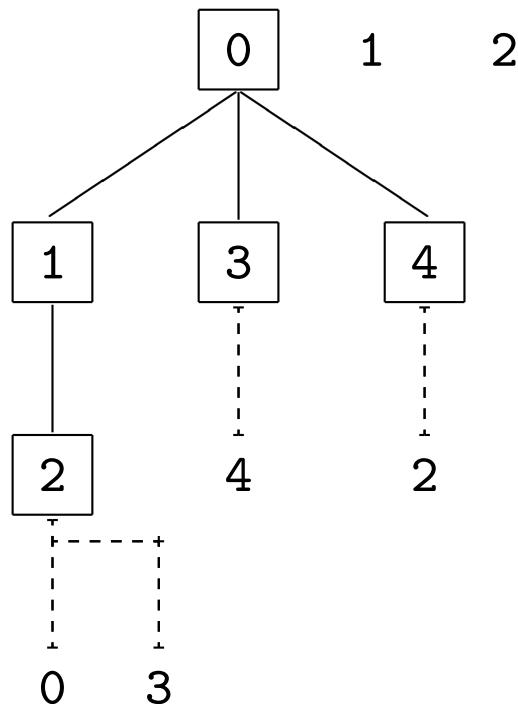


Ordem:

0 1 3 4 2

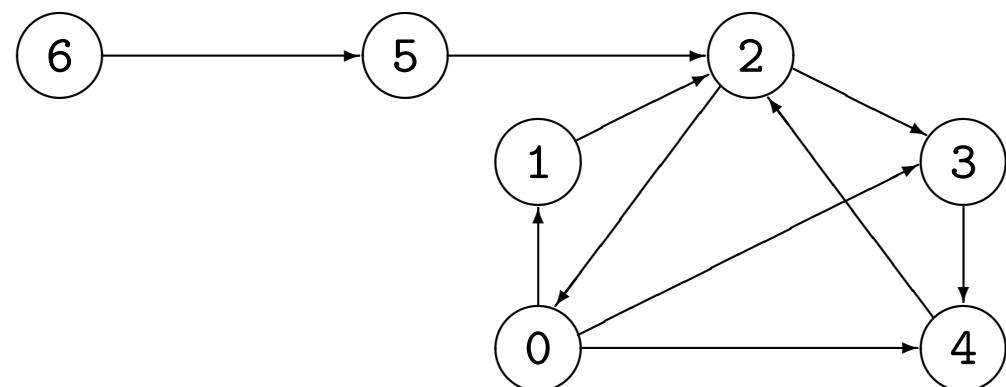


Percorso em Largura

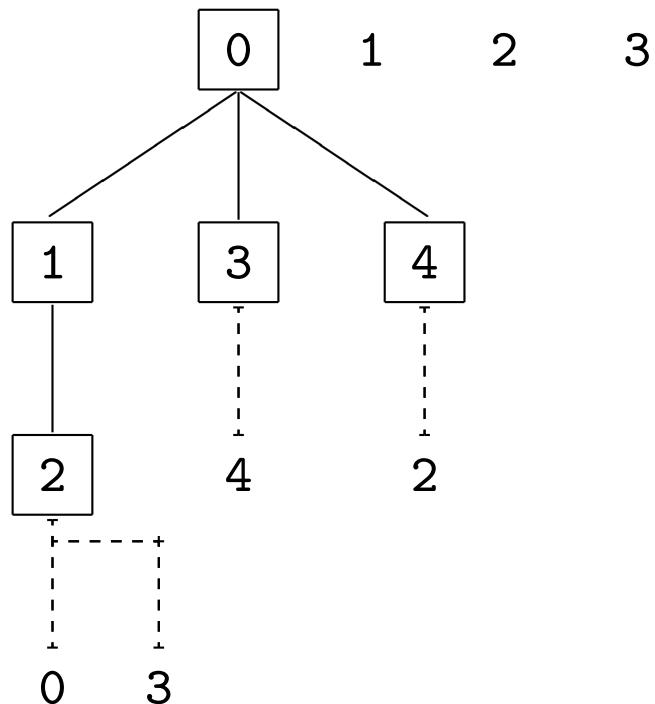


Ordem:

0 1 3 4 2

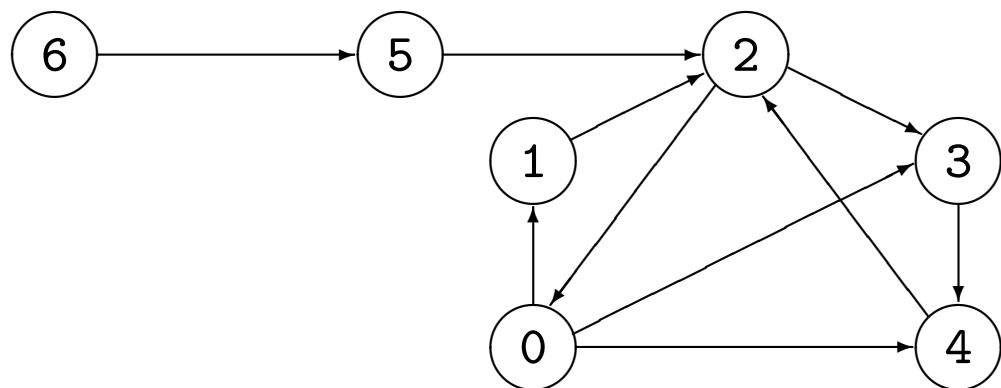


Percorso em Largura

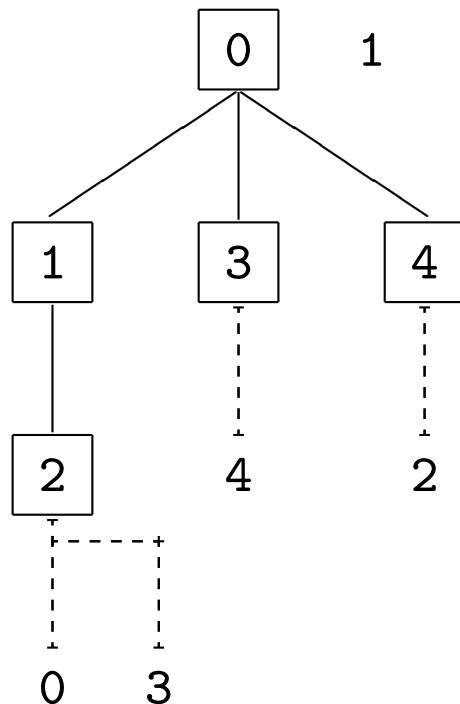


Ordem:

0 1 3 4 2

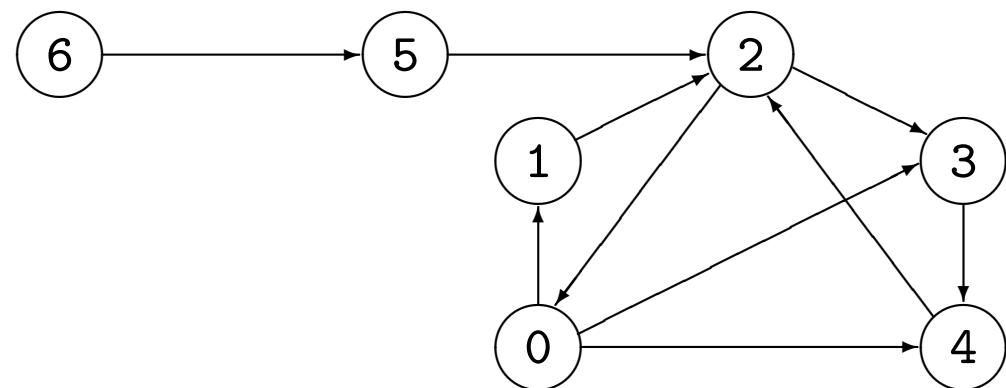


Percorso em Largura

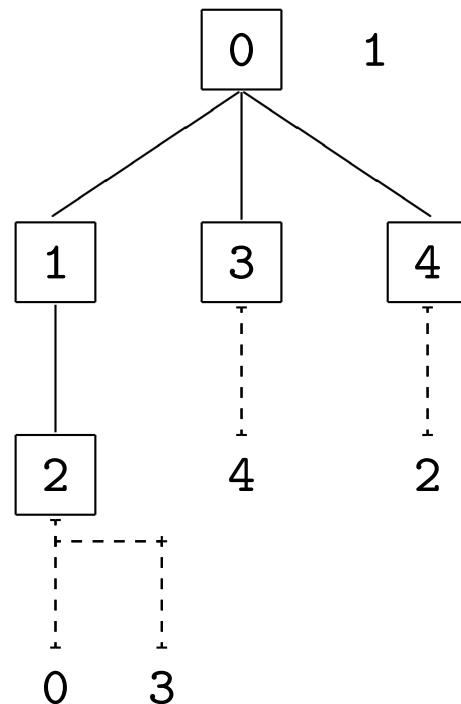


Ordem:

0 1 3 4 2

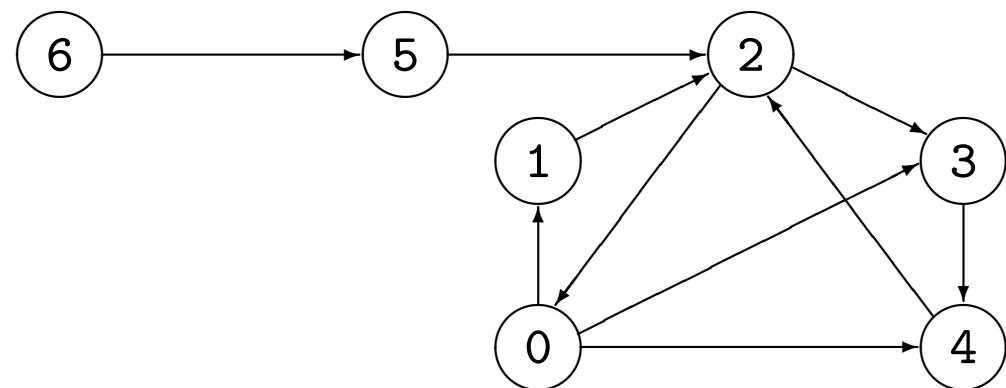


Percorso em Largura

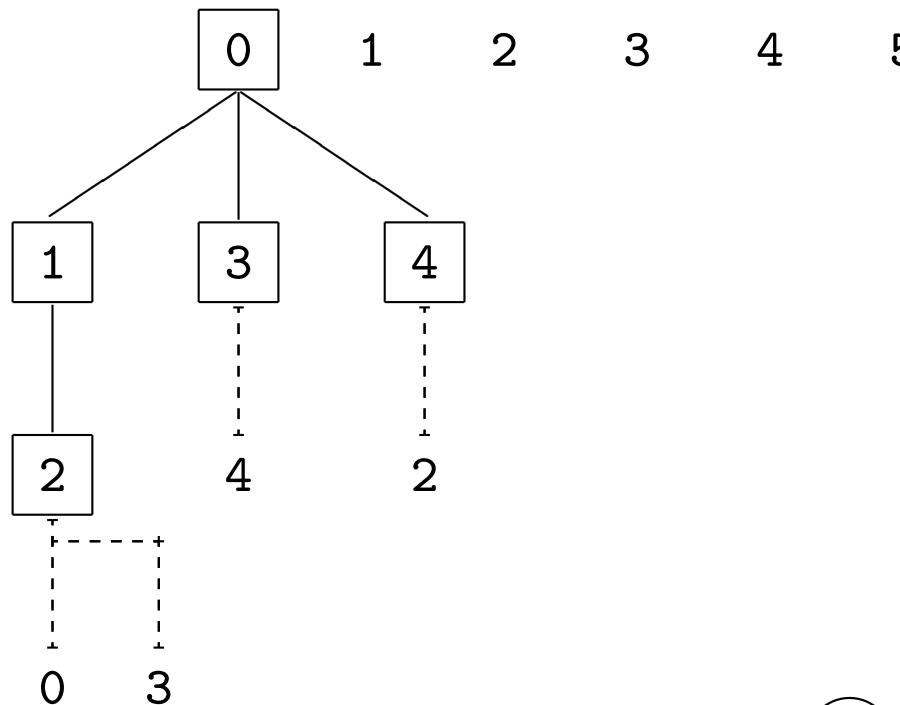


Ordem:

0 1 3 4 2



Percorso em Largura

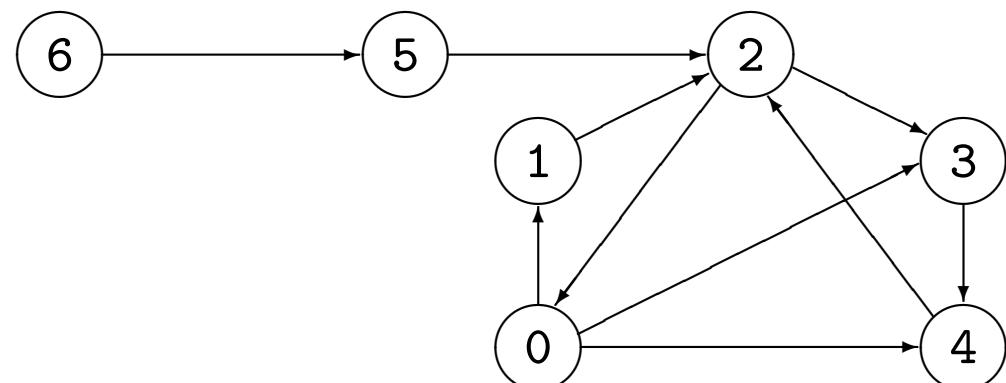


Ordem:

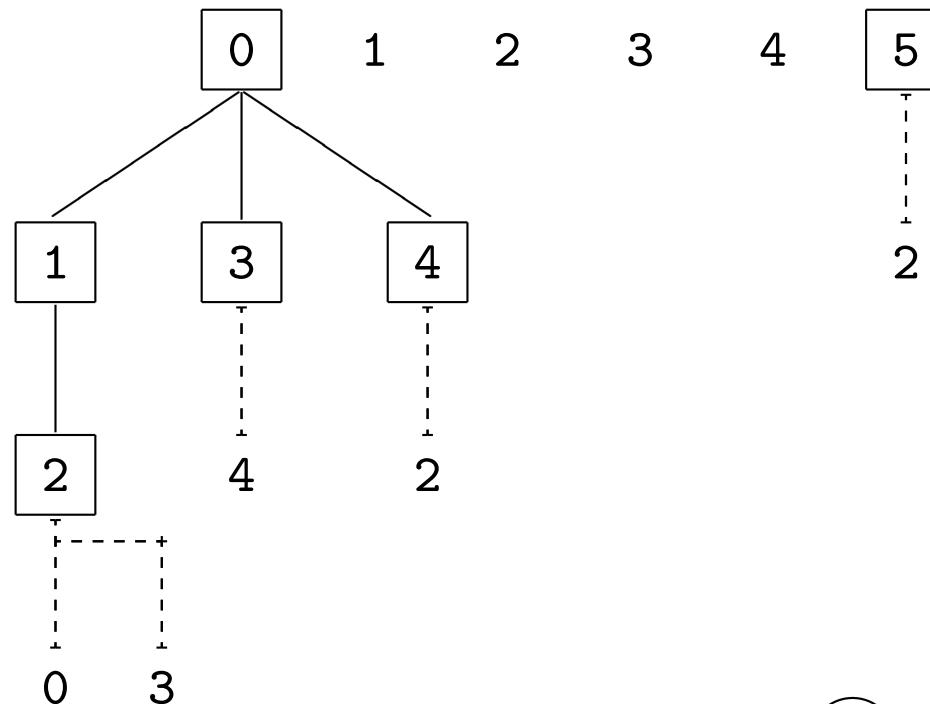
0 1 3 4 2

FIFO:

5



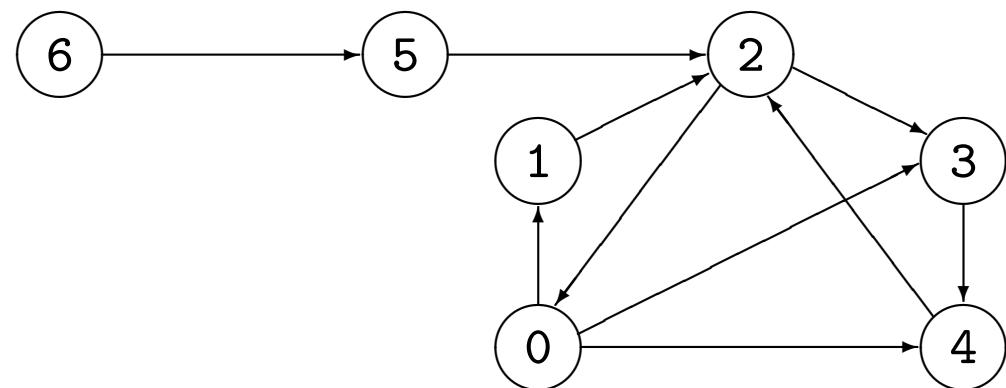
Percorso em Largura



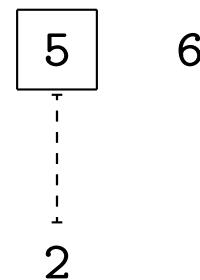
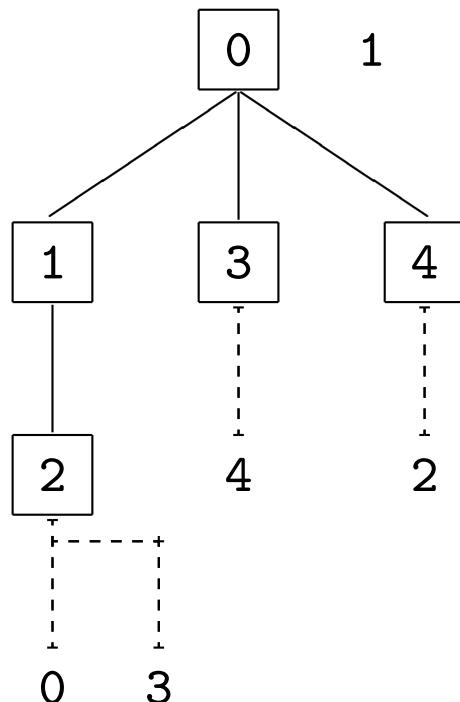
Ordem:

0 1 3 4 2 5

FIFO:

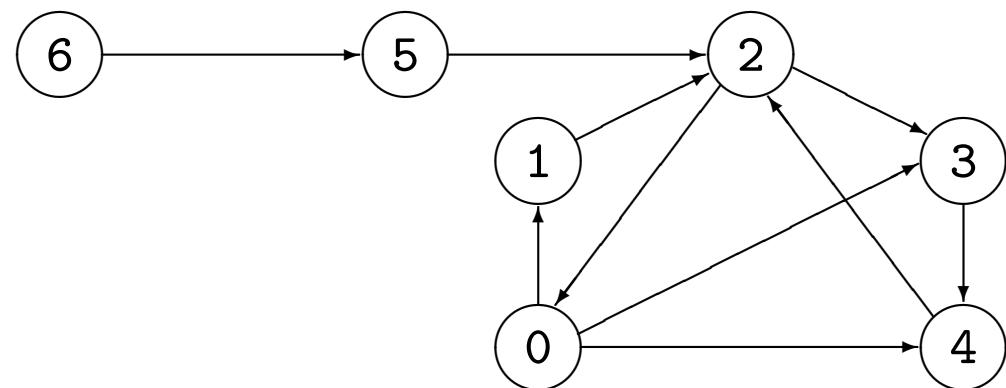


Percorso em Largura

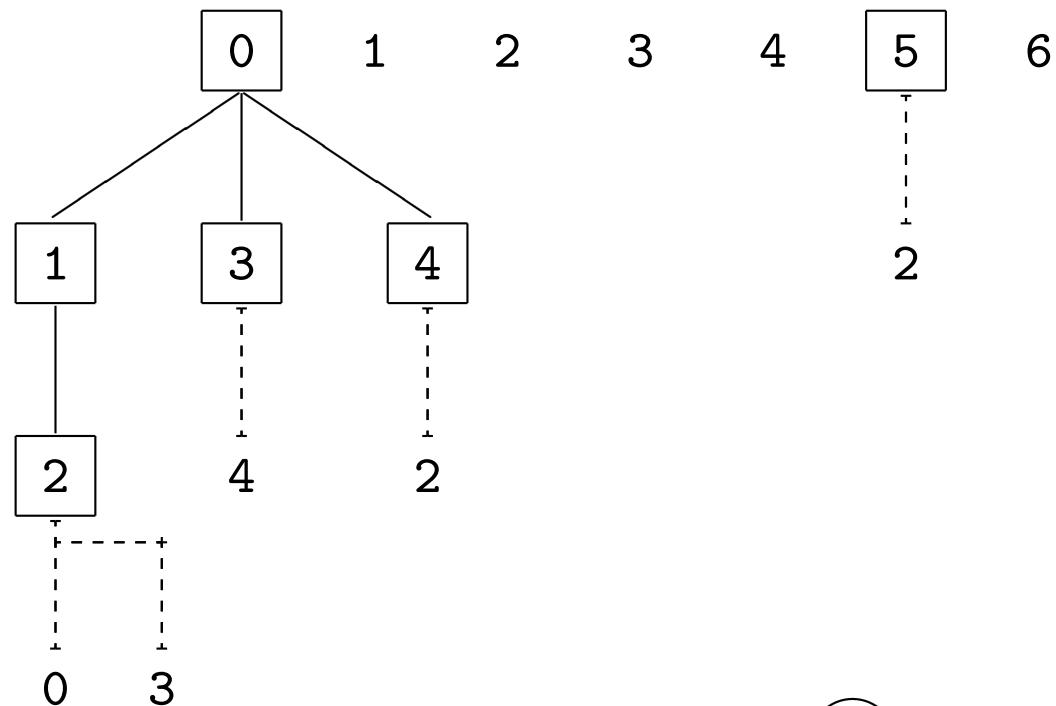


Ordem:

0 1 3 4 2 5



Percorso em Largura

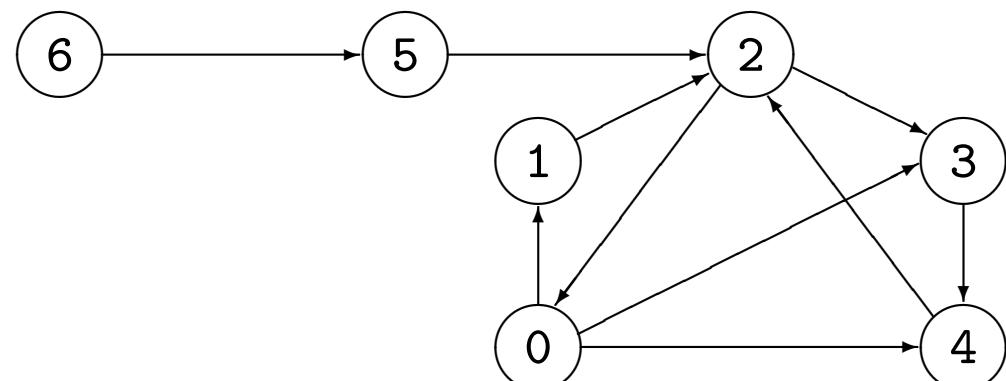


Ordem:

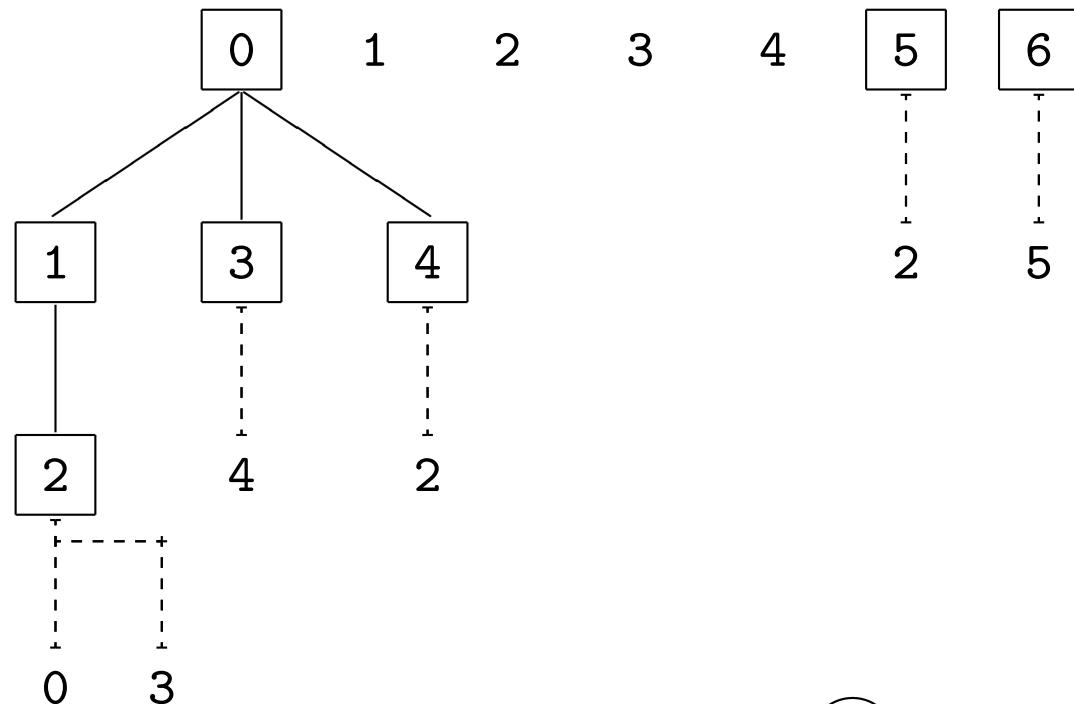
0 1 3 4 2 5

FIFO:

6



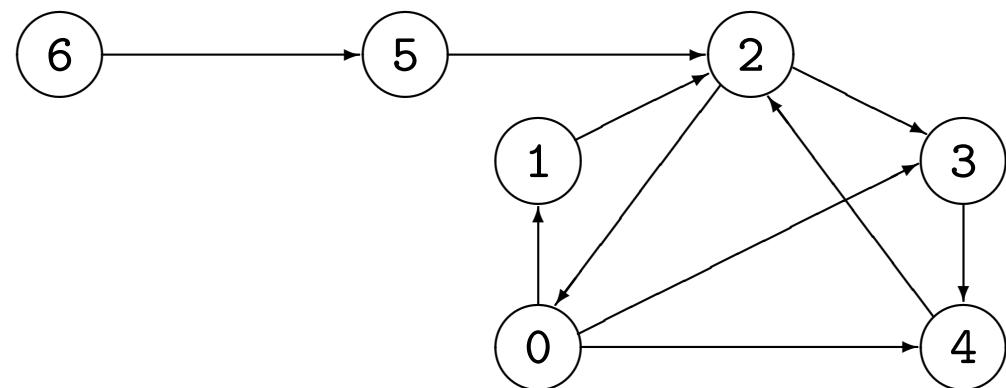
Percorso em Largura



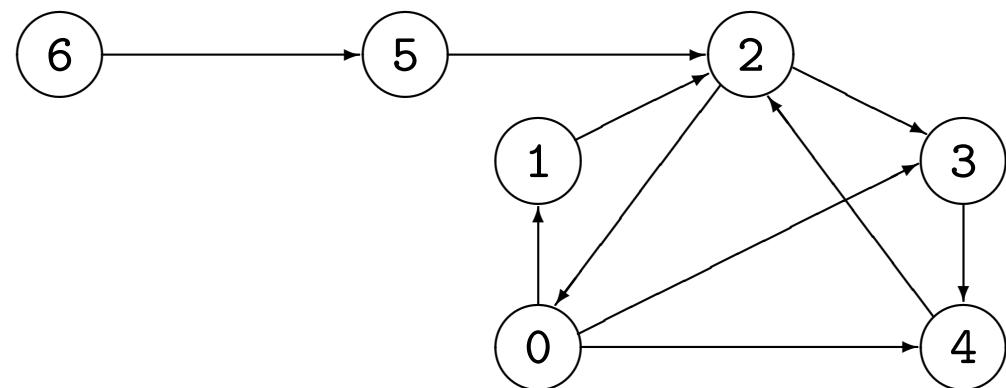
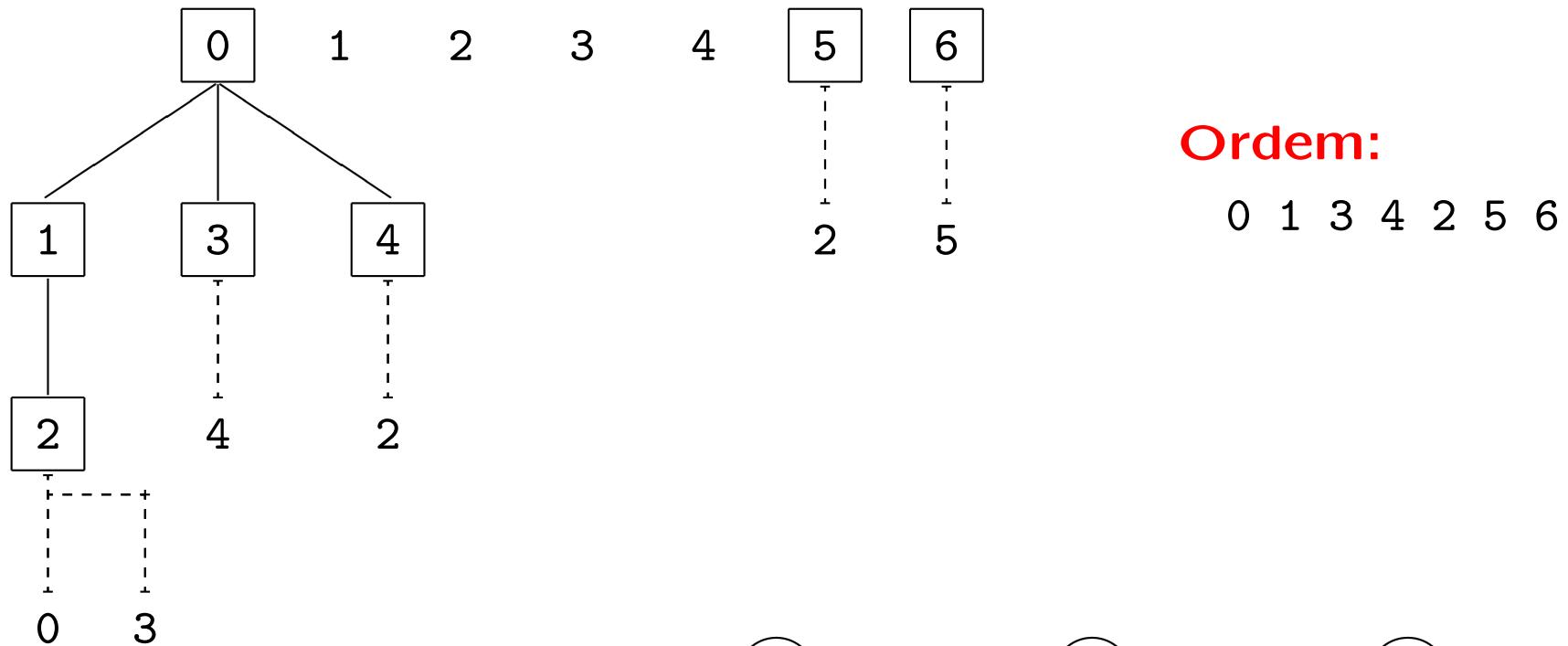
Ordem:

0 1 3 4 2 5 6

FIFO:



Percorso em Largura



Percurso em Largura

(Breadth-First Search Traversal)

```
void bfsTraversal( Digraph graph ) {  
    boolean[] found = new boolean[ graph.numNodes() ];  
  
    for every Node v in graph.nodes()  
        found[v] = false;  
  
    for every Node v in graph.nodes()  
        if ( !found[v] )  
            bfsExplore(graph, found, v);  
}
```

Árvore em Largura (iterativo)

```
void bfsExplore( Digraph graph, boolean[] found, Node root ) {  
    Queue<Node> waiting = new QueueIn...<>(?);  
    waiting.enqueue(root);  
    found[root] = true;  
    do {  
        Node node = waiting.dequeue();  
        // PROCESS(node)  
        for every Node v in graph.outAdjacentNodes(node)  
            if ( !found[v] ) {  
                waiting.enqueue(v);  
                found[v] = true;  
            }  
    }  
    while ( !waiting.isEmpty() );  
}
```

Complexidade de **bfsExplore**

- **Por cada vértice (w)**

- Remove-se w da fila $\Theta(1)$
- Se $\text{PROCESS}(w)$ não for constante, considerar o seu custo
- Iteram-se os sucessores de w
 - * Grafo em matriz de adjacências $\Theta(|V|)$
 - * Grafo em vetor de listas de adjacências (suc.) $\Theta(|\text{Suc}(w)|)$
- Inserem-se os suc. nunca inseridos de w na fila $O(|\text{Suc}(w)|)$

- **Custo de todas as execuções ($\#\text{DEQUEUE} = \#\text{ENQUEUE}$)**

- Grafo em matriz de adjacências $\Theta(|V|^2)$
- Grafo em vetor de listas de adjacências (suc.) $\Theta(|V| + |A|)$
 - Num grafo orientado, $\sum_{w \in V} |\text{Suc}(w)| = |A|$.
 - Num grafo não orientado, $\sum_{w \in V} |\text{Suc}(w)| = 2 \times |A|$.

Complexidade Temporal de **bfsTraversal** (se enqueue, dequeue e isEmpty de Queue forem $\Theta(1)$)

Grafo em matriz de adjacências

Criação do vetor found	$\Theta(1)$
1º ciclo (inicialização do vetor found)	$\Theta(V)$
2º ciclo (ignorando execuções de bfsExplore)	$\Theta(V)$
Execuções de bfsExplore	$\Theta(V ^2)$
TOTAL	$\Theta(V ^2)$

Grafo em vetor de listas de adjacências (suc.)

Criação do vetor found	$\Theta(1)$
1º ciclo (inicialização do vetor found)	$\Theta(V)$
2º ciclo (ignorando execuções de bfsExplore)	$\Theta(V)$
Execuções de bfsExplore	$\Theta(V + A)$
TOTAL	$\Theta(V + A)$

Complexidade Espacial de **bfsTraversal**

Vetor found	$\Theta(V)$
Fila waiting	$O(V)$
TOTAL	$\Theta(V)$

Capítulo V

Ordenação Topológica
(num grafo orientado e acíclico)

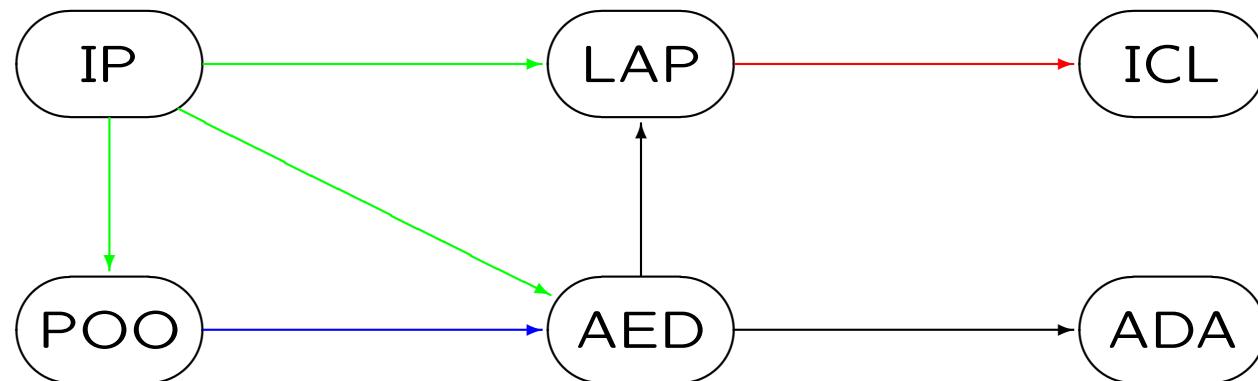
&

Teste à Aciclicidade
(num grafo orientado)

Ordenação Topológica

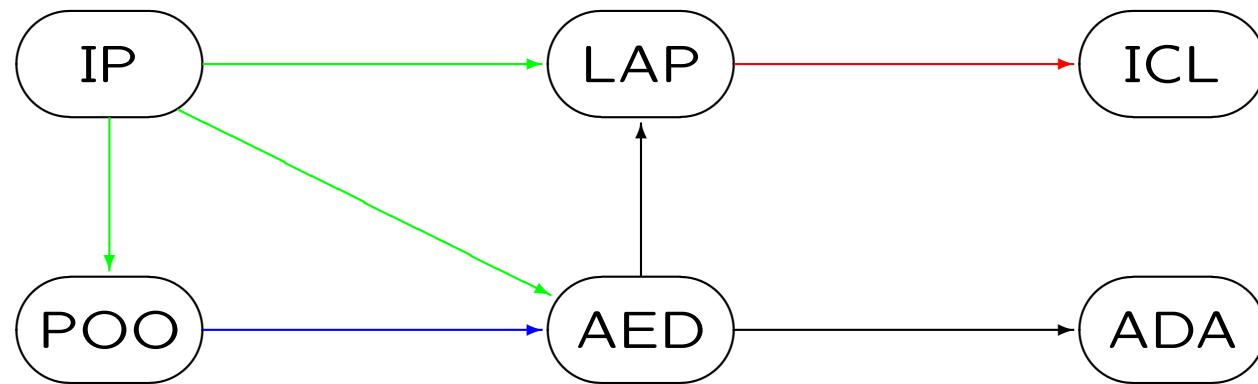
Dado um grafo $G = (V, A)$, **orientado e acíclico**, uma **ordenação topológica** de G é uma permutação de V tal que:

$$\forall(x, y) \in A \quad x \text{ precede } y.$$

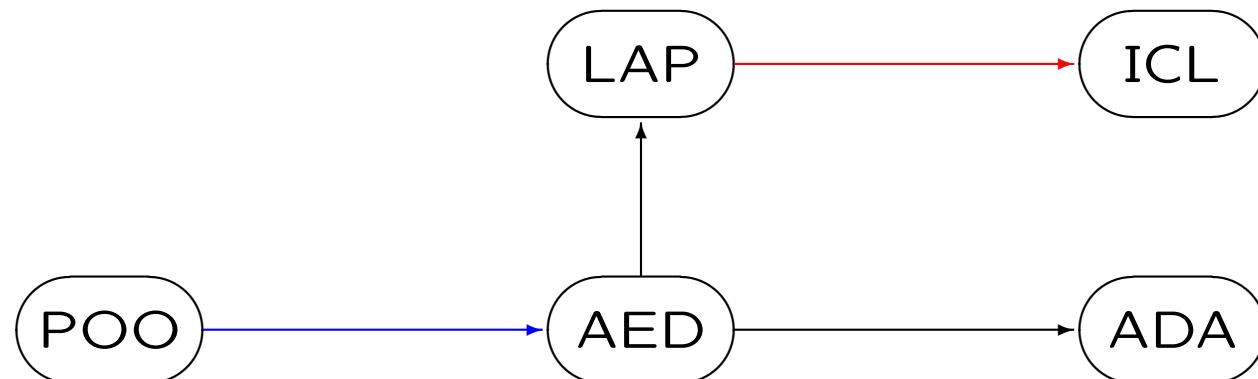


IP	POO	AED	LAP	ICL	ADA
IP	POO	AED	LAP	ADA	ICL
IP	POO	AED	ADA	LAP	ICL

Exemplo (1)

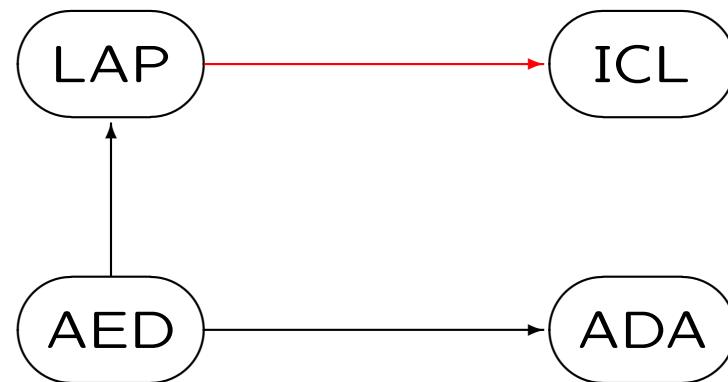


Ordenação: IP

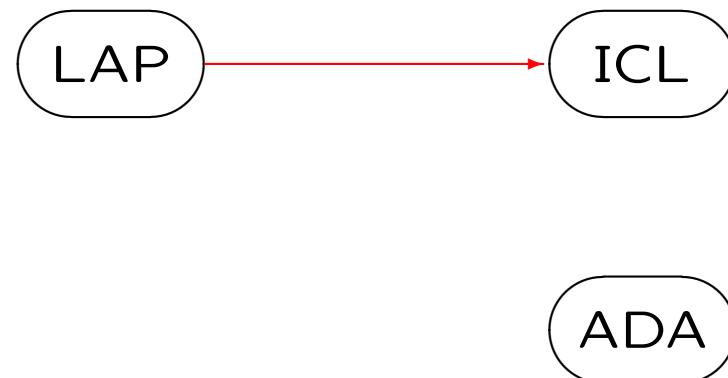


Ordenação: IP POO

Exemplo (2)



Ordenação: IP POO AED



Ordenação: IP POO AED LAP (uma das alternativas)

Exemplo (3)

ICL

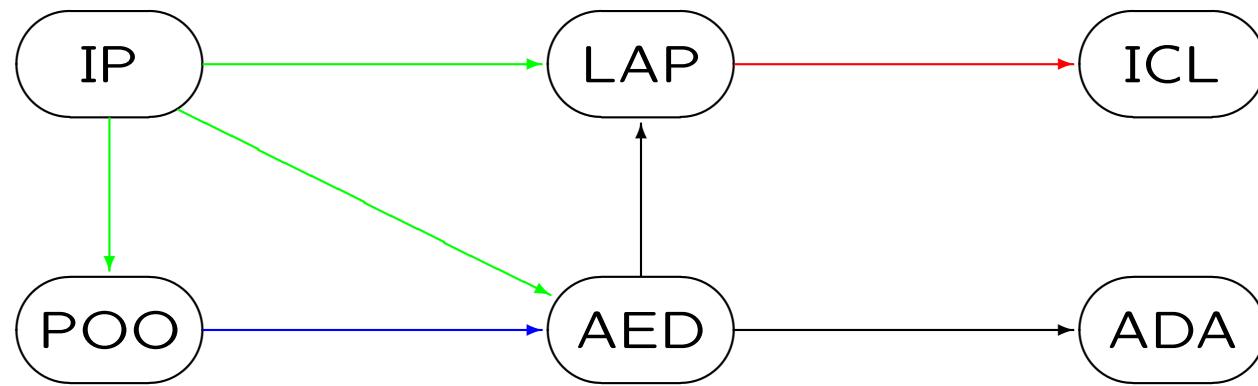
ADA

Ordenação: IP POO AED LAP ICL (uma das alternativas)

ADA

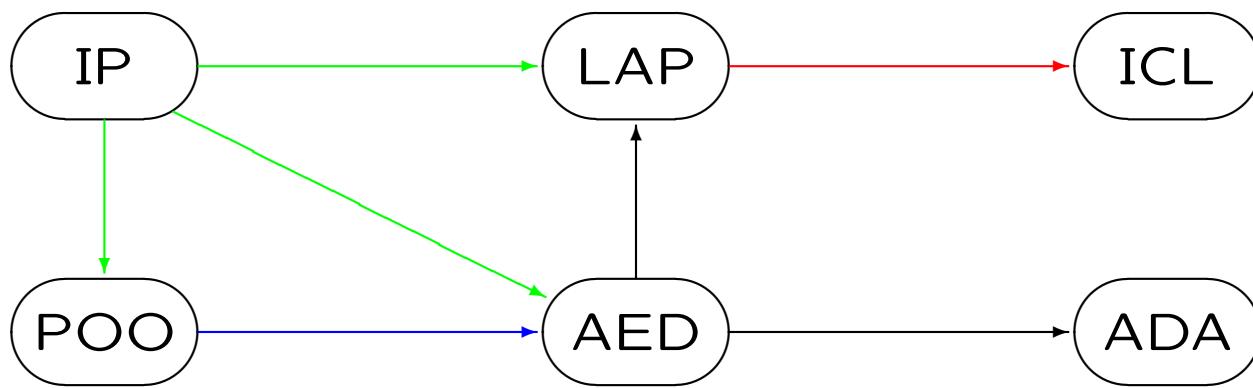
Ordenação: IP POO AED LAP ICL ADA

Número de Antecessores



LAP	2	1	1	0	—	—	—
ICL	1	1	1	1	0	—	—
AED	2	1	0	—	—	—	—
ADA	1	1	1	0	0	0	—
IP	0	—	—	—	—	—	—
POO	1	0	—	—	—	—	—
Permutação:	IP	POO	AED	LAP	ICL	ADA	

Número de Antecessores e Saco



LAP	2	1	1	0	0	0	0
ICL	1	1	1	1	0	0	0
AED	2	1	0	0	0	0	0
ADA	1	1	1	0	0	0	0
IP	0	0	0	0	0	0	0
POO	1	0	0	0	0	0	0
Saco:	IP	POO	AED	ADA, LAP	ADA, ICL	ADA	\emptyset
Perm:		IP	POO	AED	LAP	ICL	ADA

Ordenação Topológica (1)

(Topological Sorting)

```
Node[] topologicalSort( Digraph graph ) {  
    Node[] permutation = new Node[ graph.numNodes() ];  
    int permSize = 0;  
    Bag<Node> ready = new BagIn...<>(?);  
    int[] inDegree = new int[ graph.numNodes() ];  
  
    for every Node v in graph.nodes() {  
        inDegree[v] = graph.inDegree(v);  
        if ( inDegree[v] == 0 )  
            ready.add(v);  
    }  
}
```

Ordenação Topológica (2)

```
do {
    Node node = ready.remove();
    permutation[ permSize++ ] = node;
    for every Node v in graph.outAdjacentNodes(node) {
        inDegree[v]--;
        if ( inDegree[v] == 0 )
            ready.add(v);
    }
}
while ( !ready.isEmpty() );

return permutation;
}
```

Complexidade Temporal de **topologicalSort**

Grafo em matriz de adjacências (suc.)

(se criação, add, remove e isEmpty de Bag forem $\Theta(1)$)

Criação de permutation, ready e inDegree	$\Theta(1)$
1º ciclo (executado $\forall w \in V$)	$\Theta(V ^2)$
• inicialização de $\text{inDegree}[w]$: $\Theta(V)$	
• teste e possível inserção de w em ready: $\Theta(1)$	
2º ciclo (executado $\forall w \in V$)	$\Theta(V ^2)$
• remoção de w de ready: $\Theta(1)$	
• inserção de w em permutation: $\Theta(1)$	
• iteração dos sucessores de w : $\Theta(V)$	
• tratamento de cada sucessor de w : $\Theta(1)$	
TOTAL	$\Theta(V ^2)$

Complexidade Temporal de **topologicalSort**

Grafo em vetor de listas de adjacências (suc.) e vetor de inteiros (com os graus de entrada)
(se criação, add, remove e isEmpty de Bag forem $\Theta(1)$)

Criação de permutation, ready e inDegree	$\Theta(1)$
1º ciclo (executado $\forall w \in V$)	$\Theta(V)$
• inicialização de $\text{inDegree}[w]$: $\Theta(1)$	
• teste e possível inserção de w em ready: $\Theta(1)$	
2º ciclo (executado $\forall w \in V$)	$\Theta(V + A)$
• remoção de w de ready: $\Theta(1)$	
• inserção de w em permutation: $\Theta(1)$	
• iteração dos sucessores de w : $\Theta(\text{Suc}(w))$	
• tratamento de cada sucessor de w : $\Theta(1)$	
TOTAL	$\Theta(V + A)$

Complexidade Espacial de **topologicalSort**

Vetor permutation	$\Theta(V)$
Vetor inDegree	$\Theta(V)$
Saco ready	$O(V)$
TOTAL	$\Theta(V)$

Teste à Aciclicidade (1)

(Acyclicity Checking)

```
boolean isAcyclic( Digraph graph ) {  
    int numProcNodes = 0;  
    Bag<Node> ready = new BagIn...<>(?);  
    int[] inDegree = new int[ graph.numNodes() ];  
  
    for every Node v in graph.nodes() {  
        inDegree[v] = graph.inDegree(v);  
        if ( inDegree[v] == 0 )  
            ready.add(v);  
    }  
}
```

Teste à Aciclicidade (2)

```
while ( !ready.isEmpty() ) {  
    Node node = ready.remove();  
    numProcNodes++;  
    for every Node v in graph.outAdjacentNodes(node) {  
        inDegree[v]--;  
        if ( inDegree[v] == 0 )  
            ready.add(v);  
    }  
}  
  
return numProcNodes == graph.numNodes();  
}
```

Complexidade Temporal de **isAcyclic**

Grafo em matriz de adjacências (suc.)

(se criação, add, remove e isEmpty de Bag forem $\Theta(1)$)

Criação de ready e inDegree	$\Theta(1)$
1º ciclo (executado $\forall w \in V$)	$\Theta(V ^2)$
• inicialização de $\text{inDegree}[w]$: $\Theta(V)$	
• teste e possível inserção de w em ready: $\Theta(1)$	
2º ciclo (executado no máximo $\forall w \in V$)	$O(V ^2)$
• remoção de w de ready: $\Theta(1)$	
• iteração dos sucessores de w : $\Theta(V)$	
• tratamento de cada sucessor de w : $\Theta(1)$	
TOTAL	$\Theta(V ^2)$

Complexidade Temporal de **isAcyclic**

Grafo em vetor de listas de adjacências (suc.) e vetor de inteiros (com os graus de entrada)
(se criação, add, remove e isEmpty de Bag forem $\Theta(1)$)

Criação de ready e inDegree	$\Theta(1)$
1º ciclo (executado $\forall w \in V$)	$\Theta(V)$
• inicialização de $\text{inDegree}[w]$: $\Theta(1)$	
• teste e possível inserção de w em ready: $\Theta(1)$	
2º ciclo (executado no máximo $\forall w \in V$)	$O(V + A)$
• remoção de w de ready: $\Theta(1)$	
• iteração dos sucessores de w : $\Theta(\text{Suc}(w))$	
• tratamento de cada sucessor de w : $\Theta(1)$	
TOTAL	$O(V + A)$

Complexidade Espacial de **isAyclic**

Vetor inDegree $\Theta(|V|)$

Saco ready $O(|V|)$

TOTAL $\Theta(|V|)$

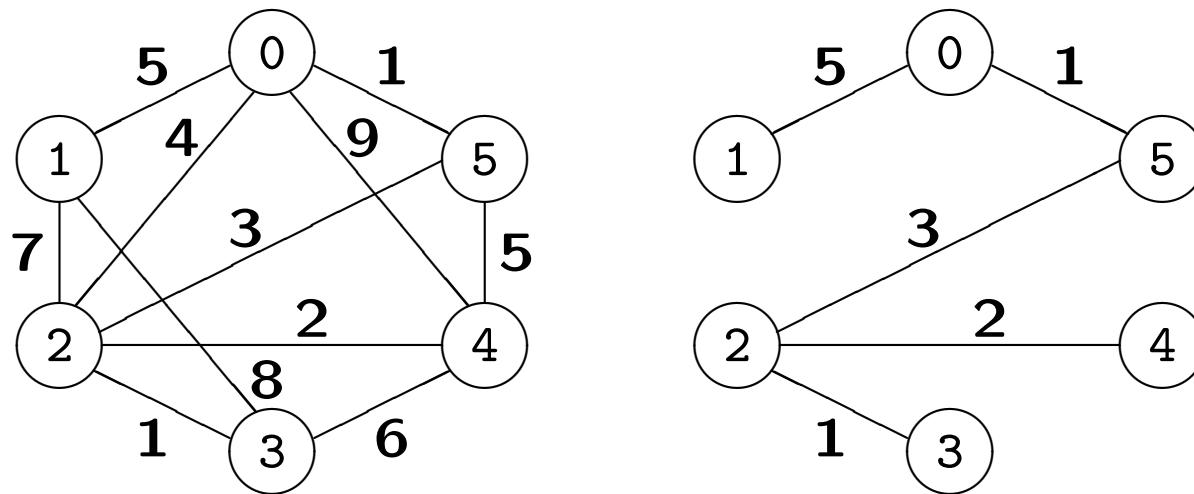
Capítulo VI

Árvore Mínima de Cobertura
(num grafo não orientado, conexo e pesado)

Algoritmo de Kruskal

Problema

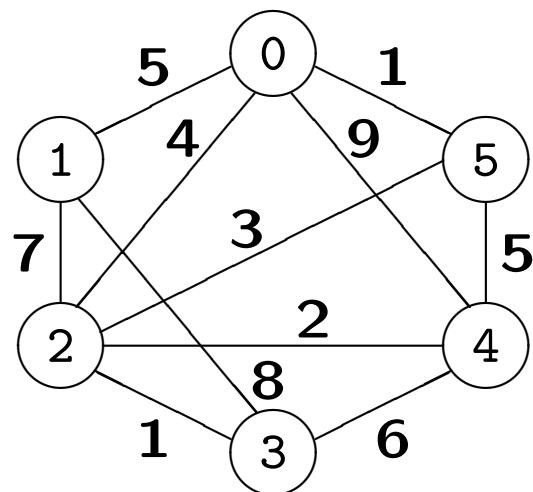
Como ligar um dado equipamento, minimizando a soma dos comprimentos das ligações?



Árvore de Cobertura (sub-grafo acíclico e conexo com todos os vértices) de custo Mínimo (nenhuma árvore de cobertura tem custo menor).

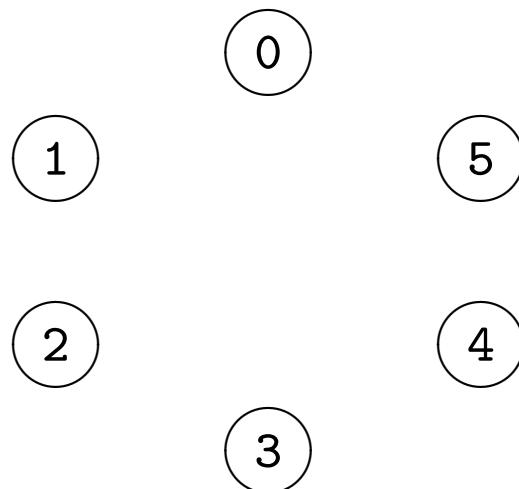
Dado um grafo **não orientado, conexo e pesado**, como obter uma **Árvore Mínima de Cobertura**?

Algoritmo de Kruskal [1956]



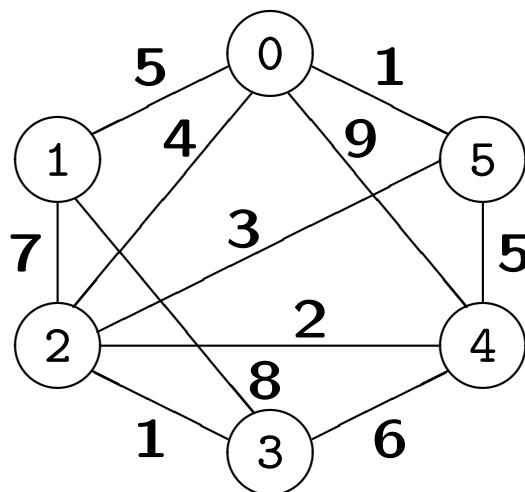
1 0 \longleftrightarrow 5
1 2 \longleftrightarrow 3
2 2 \longleftrightarrow 4
3 2 \longleftrightarrow 5
4 0 \longleftrightarrow 2
5 0 \longleftrightarrow 1

5 4 \longleftrightarrow 5
6 3 \longleftrightarrow 4
7 1 \longleftrightarrow 2
8 1 \longleftrightarrow 3
9 0 \longleftrightarrow 4



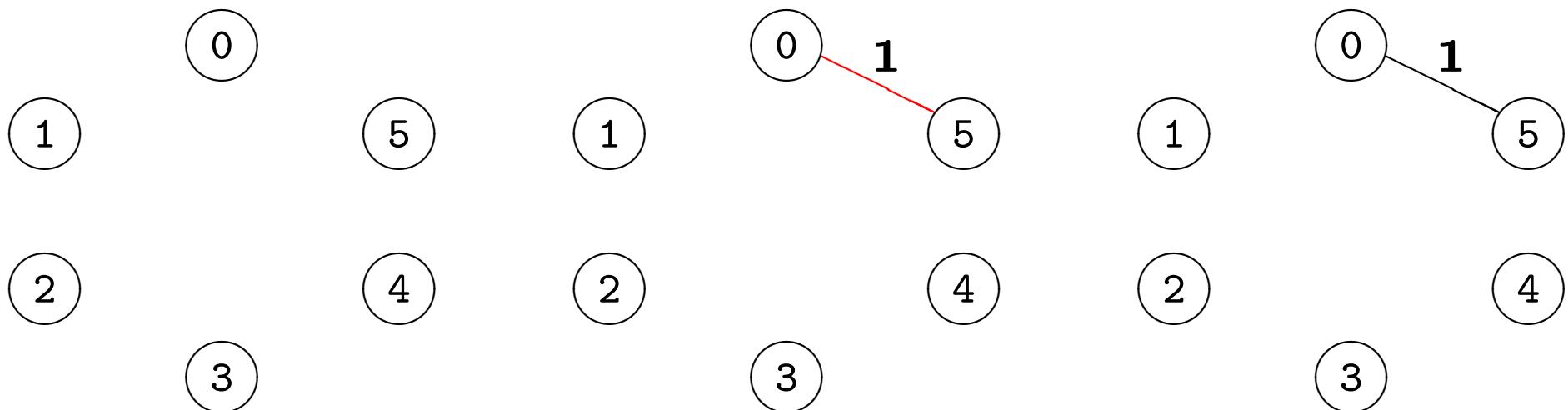
Algoritmo Greedy

Algoritmo de Kruskal (1)

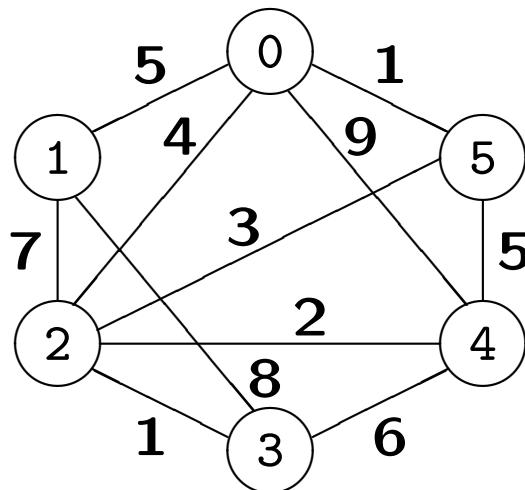


1 0 \longleftrightarrow 5
1 2 \longleftrightarrow 3
2 2 \longleftrightarrow 4
3 2 \longleftrightarrow 5
4 0 \longleftrightarrow 2
5 0 \longleftrightarrow 1

5 4 \longleftrightarrow 5
6 3 \longleftrightarrow 4
7 1 \longleftrightarrow 2
8 1 \longleftrightarrow 3
9 0 \longleftrightarrow 4

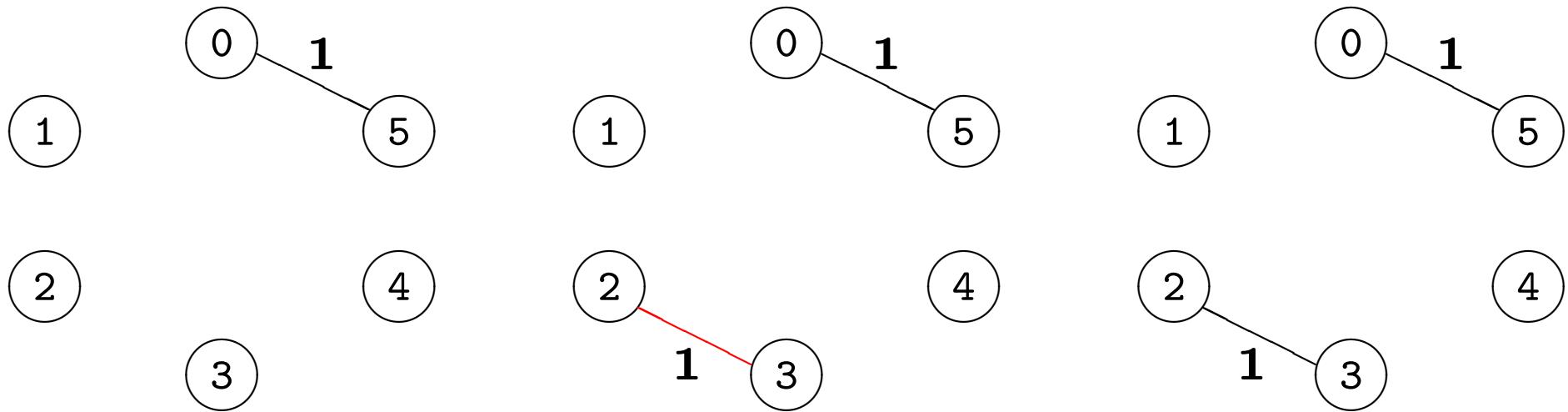


Algoritmo de Kruskal (2)

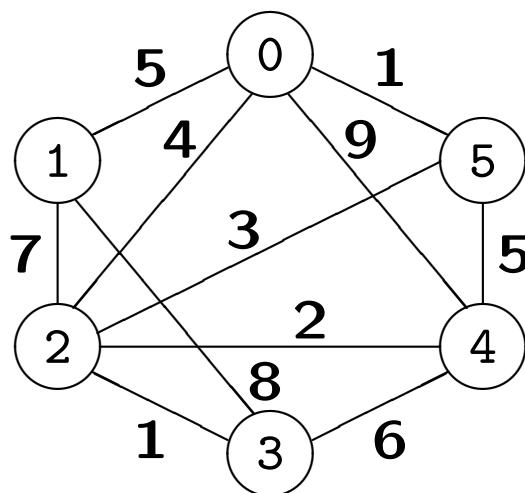


1 0 \longleftrightarrow 5 ✓
1 2 \longleftrightarrow 3
2 2 \longleftrightarrow 4
3 2 \longleftrightarrow 5
4 0 \longleftrightarrow 2
5 0 \longleftrightarrow 1

5 4 \longleftrightarrow 5
6 3 \longleftrightarrow 4
7 1 \longleftrightarrow 2
8 1 \longleftrightarrow 3
9 0 \longleftrightarrow 4

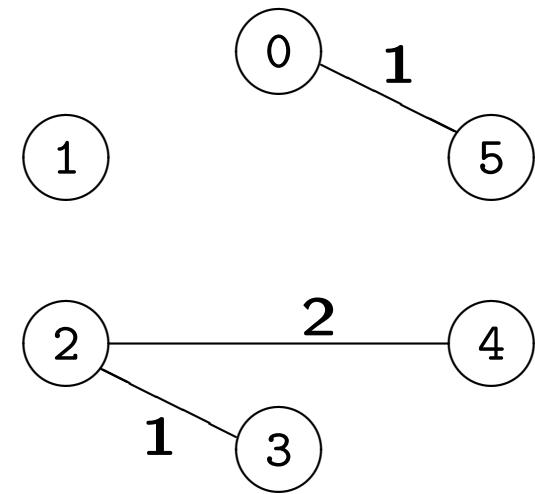
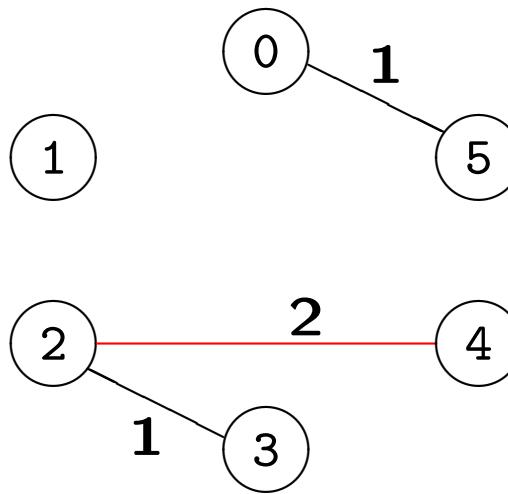
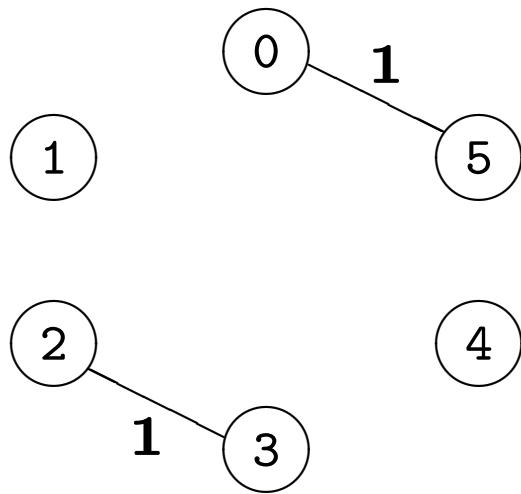


Algoritmo de Kruskal (3)

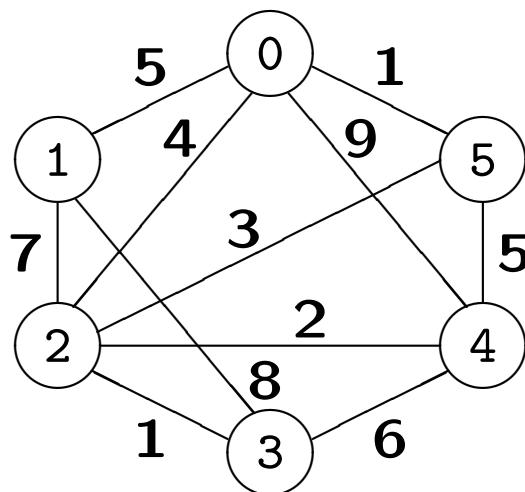


1 0 ↔ 5 ✓
1 2 ↔ 3 ✓
2 2 ↔ 4
3 2 ↔ 5
4 0 ↔ 2
5 0 ↔ 1

5 4 ↔ 5
6 3 ↔ 4
7 1 ↔ 2
8 1 ↔ 3
9 0 ↔ 4

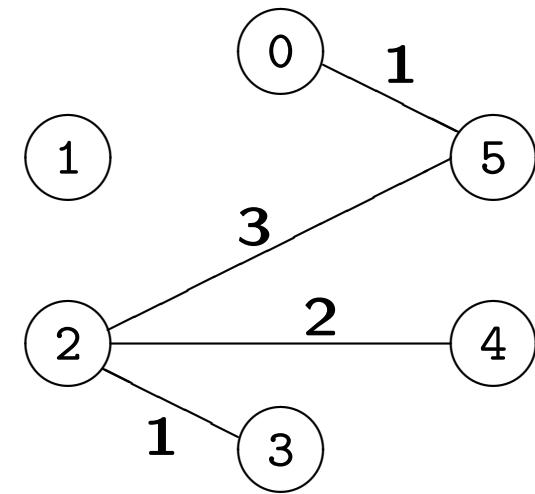
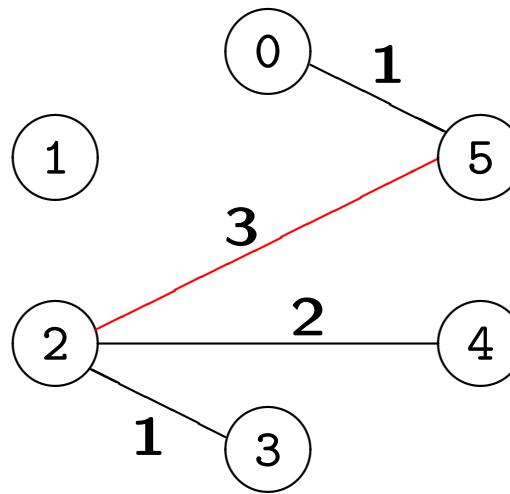
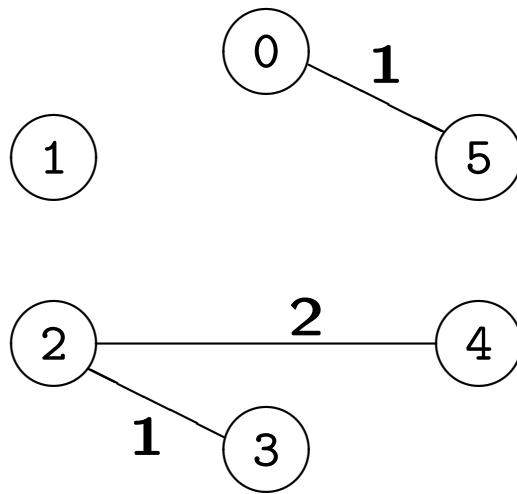


Algoritmo de Kruskal (4)

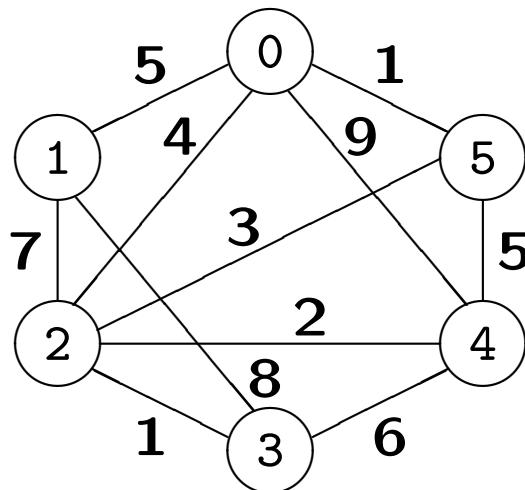


1	0	\longleftrightarrow	5	✓
1	2	\longleftrightarrow	3	✓
2	2	\longleftrightarrow	4	✓
3	2	\longleftrightarrow	5	
4	0	\longleftrightarrow	2	
5	0	\longleftrightarrow	1	

5	4	\longleftrightarrow	5
6	3	\longleftrightarrow	4
7	1	\longleftrightarrow	2
8	1	\longleftrightarrow	3
9	0	\longleftrightarrow	4

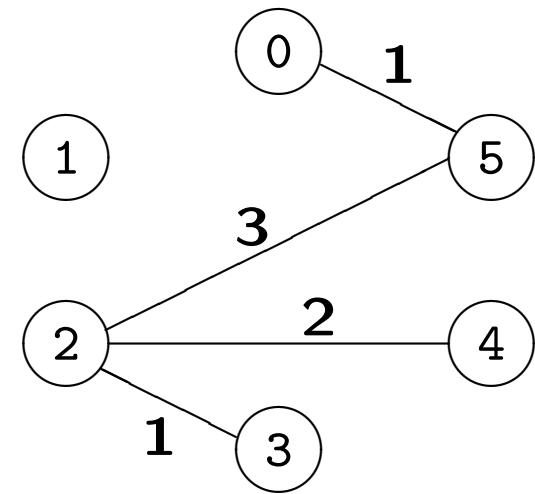
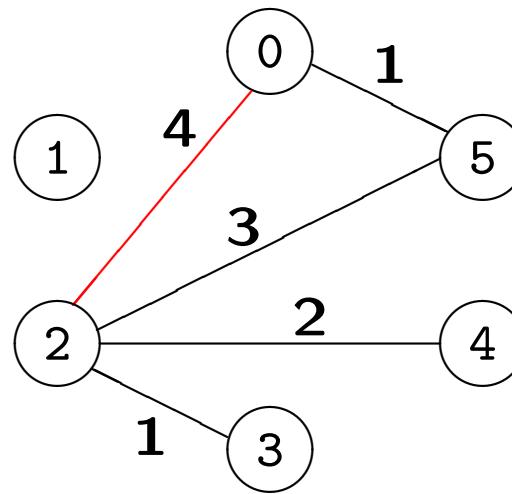
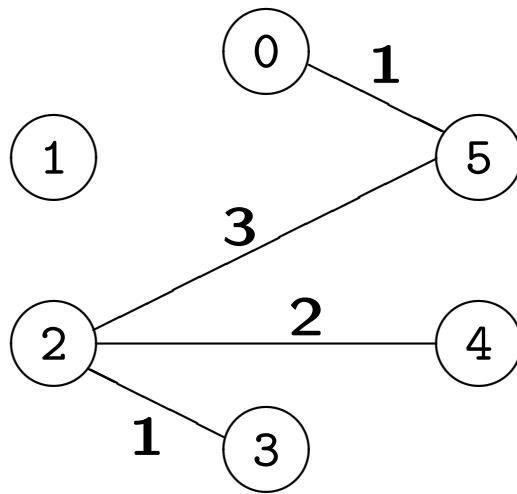


Algoritmo de Kruskal (5)

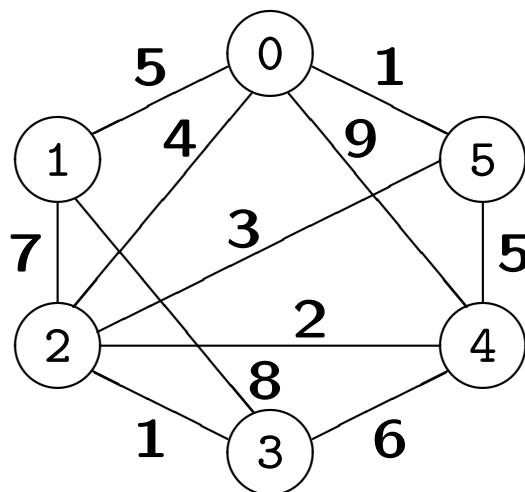


1 0 \longleftrightarrow 5 ✓
1 2 \longleftrightarrow 3 ✓
2 2 \longleftrightarrow 4 ✓
3 2 \longleftrightarrow 5 ✓
4 0 \longleftrightarrow 2
5 0 \longleftrightarrow 1

5 4 \longleftrightarrow 5
6 3 \longleftrightarrow 4
7 1 \longleftrightarrow 2
8 1 \longleftrightarrow 3
9 0 \longleftrightarrow 4

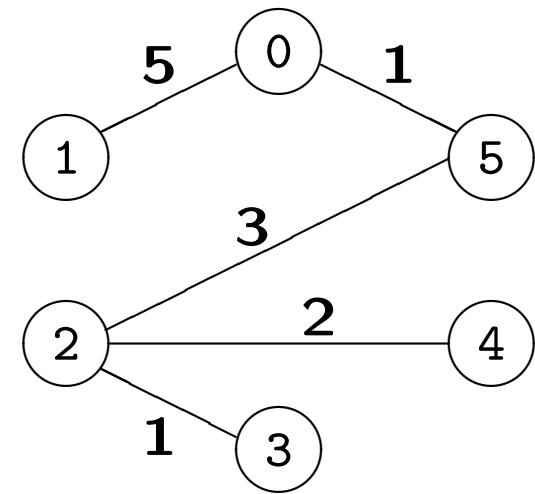
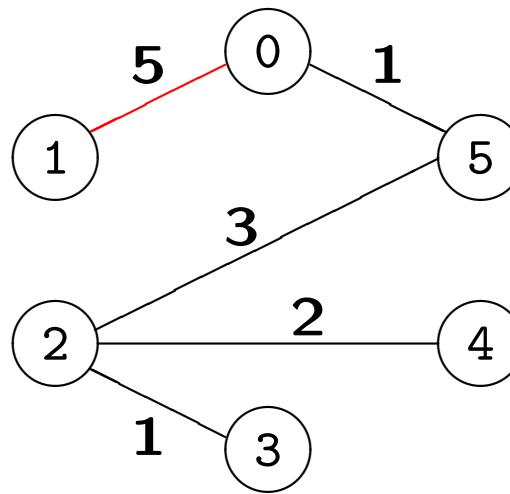
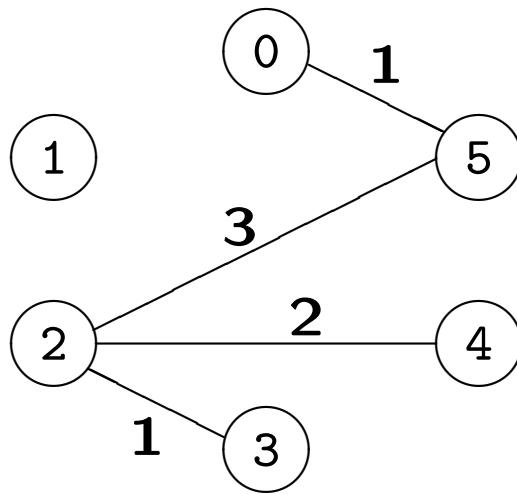


Algoritmo de Kruskal (6)

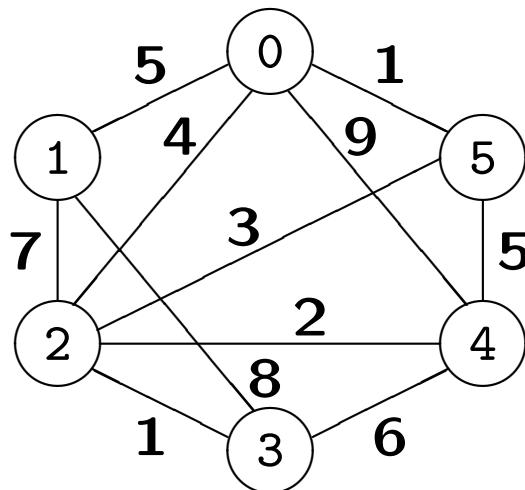


1 0 \longleftrightarrow 5 ✓
1 2 \longleftrightarrow 3 ✓
2 2 \longleftrightarrow 4 ✓
3 2 \longleftrightarrow 5 ✓
4 0 \longleftrightarrow 2
5 0 \longleftrightarrow 1

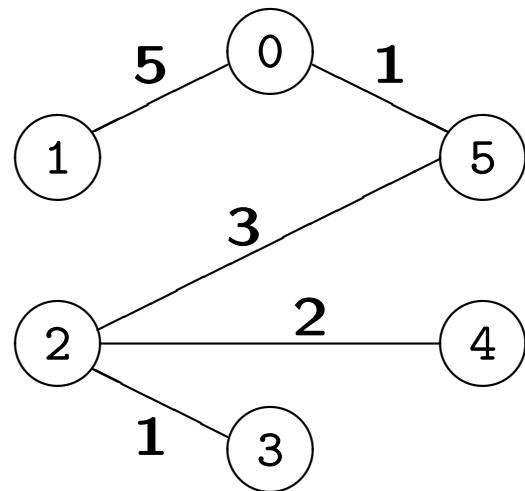
5 4 \longleftrightarrow 5
6 3 \longleftrightarrow 4
7 1 \longleftrightarrow 2
8 1 \longleftrightarrow 3
9 0 \longleftrightarrow 4



Algoritmo de Kruskal (7)



1	0 ←→ 5	✓	5	4 ←→ 5
1	2 ←→ 3	✓	6	3 ←→ 4
2	2 ←→ 4	✓	7	1 ←→ 2
3	2 ←→ 5	✓	8	1 ←→ 3
4	0 ←→ 2		9	0 ←→ 4
5	0 ←→ 1	✓		



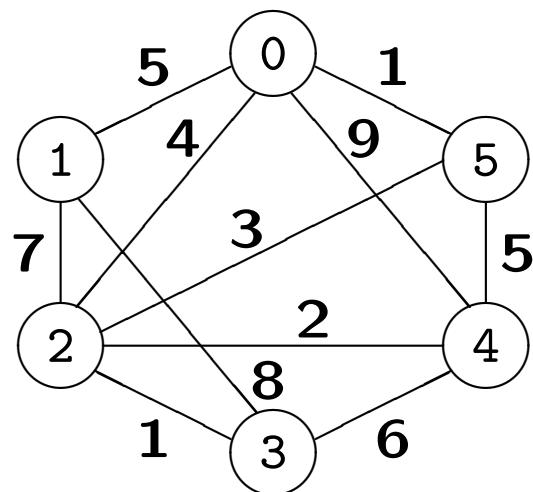
Custo da Árvore:

$$1 + 1 + 2 + 3 + 5 = 12$$

Como obter os arcos por ordem?

- Ordenam-se os arcos com um algoritmo de ordenação: $O(|A| \log |A|)$
E se se analisarem apenas k arcos, com $k \ll |A|$?
- Colocam-se os arcos todos numa ED e, em cada passo, remove-se um arco com peso mínimo.
 - ED é vetor (desordenado) ou lista ligada: $O(k \times |A|)$
 $O(|A|^2)$
 - ED é AVL ou red-black: $O(|A| \log |A|)$
 - ED é heap binário (construído com A): $O(|A| + k \log |A|)$
 $O(|A| \log |A|)$

Se inserir “este” arco, haverá ciclos? (1)

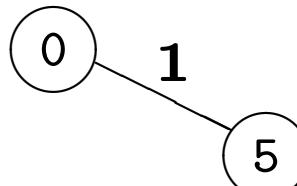


(0,5)?



{ {0}, {1}, {2}, {3}, {4}, {5} }

(2,3)?



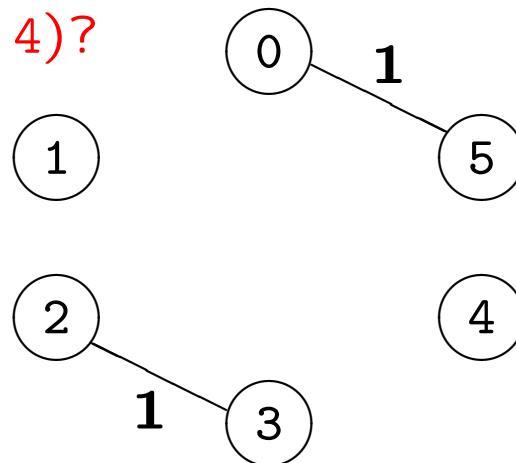
{ {0,5}, {1}, {2}, {3}, {4} }



{ {0,5}, {1}, {2,3}, {4} }

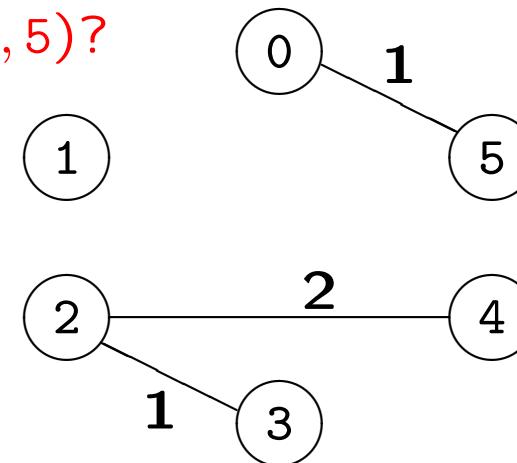
Se inserir “este” arco, haverá ciclos? (2)

(2, 4)?



{ {0, 5}, {1}, {2, 3}, {4} }

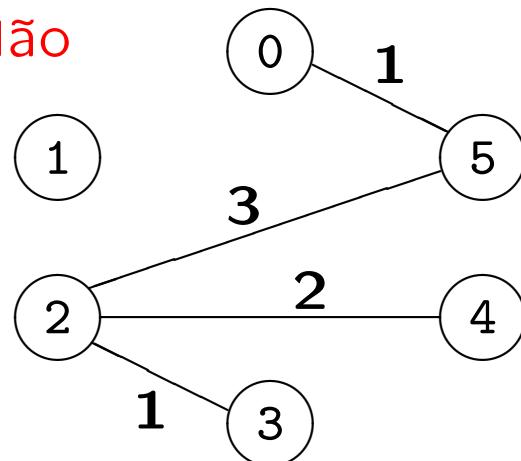
(2, 5)?



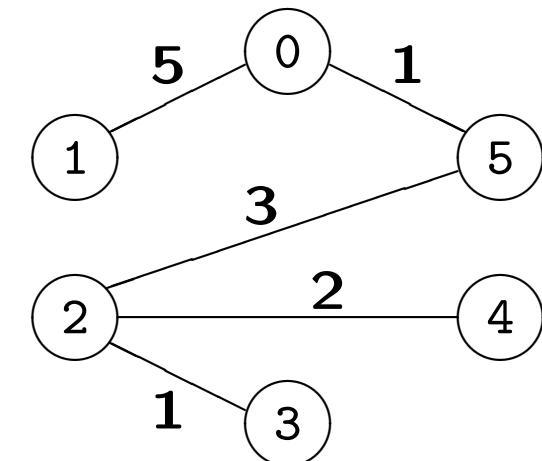
{ {0, 5}, {1}, {2, 3, 4} }

(0, 2)? Não

(0, 1)?



{ {0, 5, 2, 3, 4}, {1} }



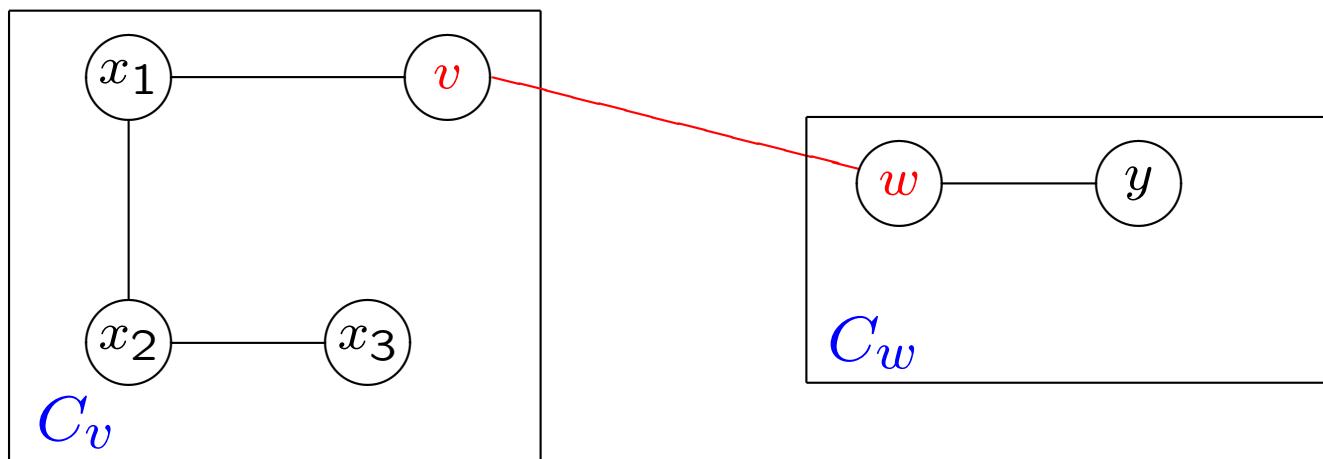
{ {0, 1, 2, 3, 4, 5} }

Caminhos & Partição

- **No início:**

- O grafo não tem arcos; só há caminhos de comprimento 0
- A partição é $\{\{v_1\}, \{v_2\}, \dots, \{v_k\}\}$, se $V = \{v_1, v_2, \dots, v_k\}$

- **Inserção do arco (v, w) :** (apenas quando $C_v \neq C_w$)



- No grafo: há caminho entre quaisquer dois vértices de $C_v \cup C_w$
- Na partição: substitui-se C_v e C_w por $C_v \cup C_w$

TAD Partição (com n elementos)

Os elementos dos conjuntos são $0, 1, 2, \dots, n - 1$. Cada conjunto é identificado por um dos seus elementos, denominado **o representante** do conjunto.

$$\text{Domínio} = \{0, 1, \dots, n - 1\}$$

// Cria a partição $\{\{0\}, \{1\}, \dots, \{n - 1\}\}$.

Partição **cria**(int n);

// Devolve o representante do conjunto ao qual e pertence.

Domínio **representante**(Domínio e);

// Substitui os conjuntos C_e e C_f , cujos representantes são e e f ,

// respetivamente, pelo conjunto $C_e \cup C_f$.

// **Pré-condição:** $e \neq f$ (ou seja, $C_e \neq C_f$).

void reunião(Domínio e , Domínio f);

Interface Partição (com n elementos)

```
public interface UnionFind {  
  
    // Creates the partition {{0}, {1}, ..., {domainSize - 1}}.  
    // UnionFind( int domainSize );  
  
    // Returns the representative of the set that contains  
    // the specified element.  
    int find( int element ) throws InvalidElementException;  
  
    // Removes the two distinct sets  $S_1$  and  $S_2$  whose representatives  
    // are the specified elements, and inserts the set  $S_1 \cup S_2$ .  
    // The representative of the new set  $S_1 \cup S_2$  can be any of  
    // its members.  
    void union( int representative1, int representative2 ) throws  
        InvalidElementException, NotRepresentativeException,  
        EqualSetsException;  
}
```

Construir a Fila com Prioridade de Arcos (Java)

```
@SuppressWarnings( "unchecked" )
```

```
MinPriorityQueue<L, Edge<L>> buildQueue( UndiGraph<L> graph ) {  
  
    Entry<L, Edge<L>>[] auxArray = new Entry[ graph.numEdges() ];  
  
    int pos = 0;  
    for every Edge<L> e in graph.edges()  
        auxArray[pos++] = new EntryClass<>(e.label(), e);  
  
    MinPriorityQueue<L, Edge<L>> priQueue =  
        new MinHeap<>(auxArray);  
  
    return priQueue;  
}
```

Árvore Mínima de Cobertura (1)

(Minimum Spanning Tree)

```
Edge<L>[] mstKruskal( UndiGraph<L> graph ) {  
  
    MinPriorityQueue<L, Edge<L>> allEdges = buildQueue(graph);  
  
    UnionFind nodesPartition =  
        new UnionFindInArray( graph.numNodes() );  
  
    int mstFinalSize = graph.numNodes() - 1;  
  
    Edge<L>[] mst = new Edge<>[ mstFinalSize ];  
  
    int mstSize = 0;
```

Árvore Mínima de Cobertura (2)

```
while ( mstSize < mstFinalSize ) {  
    Edge<L> edge = allEdges.removeMin().getValue();  
    int rep1 = nodesPartition.find( edge.firstNode() );  
    int rep2 = nodesPartition.find( edge.secondNode() );  
    if ( rep1 != rep2 ) {  
        mst[ mstSize++ ] = edge;  
        nodesPartition.union(rep1, rep2);  
    }  
}  
  
return mst;
```

}

Complexidade

Identificação das Operações

criação do heap	$\Theta(A)$
criação da partição	?
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	
1 remoção do mínimo	$O(\log A)$
2 representante	?
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	?

Complexidade do Algoritmo de Kruskal

Reunião sem Estratégia

Representante sem Efeitos Laterais

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times V)$

Complexidade do Algoritmo de Kruskal

Reunião sem Estratégia

Representante sem Efeitos Laterais

Complexidade do Primeiro Ciclo

$$O(|A| \times \log |A| + |A| \times (2R))$$

$$O(|A| \times \underbrace{\log |A|}_{\log |A| < 2 \log |V| \text{ porque } |A| < |V|^2} + |A| \times |V|)$$

$$O(|A| \times \log |V| + |A| \times |V|)$$

$$O(|A| \times |V|)$$

Complexidade do Algoritmo de Kruskal

Reunião por Altura ou por Tamanho

Representante sem Efeitos Laterais

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(\log V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times \log V)$

Complexidade do Algoritmo de Kruskal

Reunião por Altura ou por Tamanho

Representante sem Efeitos Laterais

Complexidade do Primeiro Ciclo

$$O(|A| \times \log |A| + |A| \times (2\mathbf{R}))$$

$$O(|A| \times \log |V| + |A| \times \log |V|)$$

$$O(|A| \times \log |V|)$$

Complexidade do Algoritmo de Kruskal

Reunião por Nível ou por Tamanho

Representante com Compressão do Caminho

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(\log V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times \log V)$

Complexidade do Algoritmo de Kruskal

Reunião por Nível ou por Tamanho

Representante com Compressão do Caminho

Complexidade do Primeiro Ciclo

$$O(|A| \times \log |A| + \underbrace{|A| \times (2\mathbf{R})}_{2|A| \geq 2(|V|-1) \geq |V|})$$

$$O(|A| \times \log |V| + (2|A|) \underbrace{\alpha(2|A|, |V|)}_{\leq 4})$$

$$O(|A| \times \log |V| + |A|)$$

$$O(|A| \times \log |V|)$$

Capítulo VII

Tipo Abstrato de Dados Partição

(Union-Find, Disjoint Sets or Partition ADT)

TAD Partição (com n elementos)

Os elementos dos conjuntos são $0, 1, 2, \dots, n - 1$. Cada conjunto é identificado por um dos seus elementos, denominado **o representante** do conjunto.

$$\text{Domínio} = \{0, 1, \dots, n - 1\}$$

// Cria a partição $\{\{0\}, \{1\}, \dots, \{n - 1\}\}$.

Partição **cria**(int n);

// Devolve o representante do conjunto ao qual e pertence.

Domínio **representante**(Domínio e);

// Substitui os conjuntos C_e e C_f , cujos representantes são e e f ,

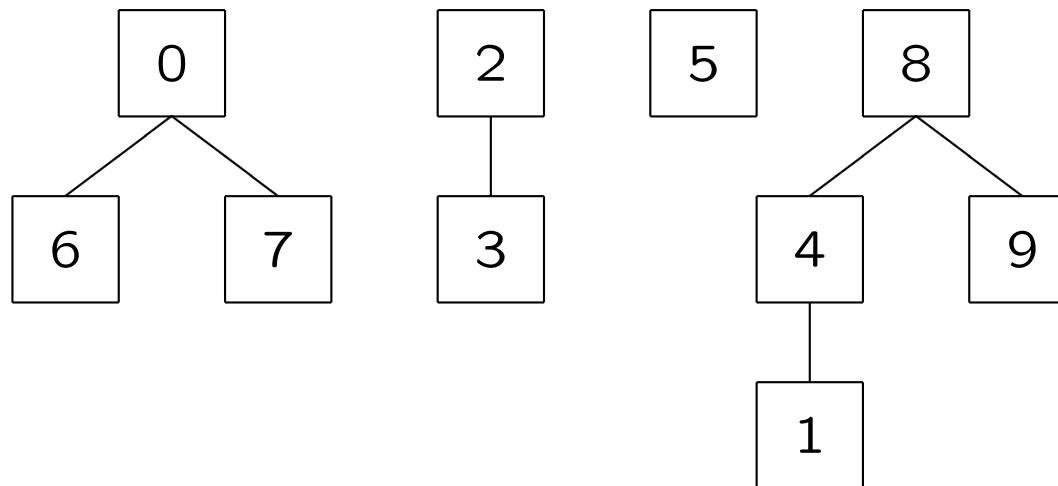
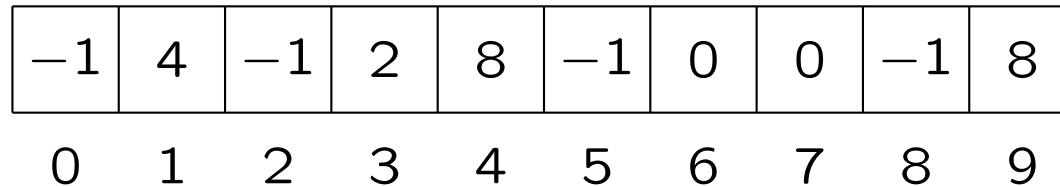
// respetivamente, pelo conjunto $C_e \cup C_f$.

// **Pré-condição:** $e \neq f$ (ou seja, $C_e \neq C_f$).

void reunião(Domínio e , Domínio f);

Implementação em Floresta

(implementada em vetor)



Complexidade (com n elementos)

criação $\Theta(?)$

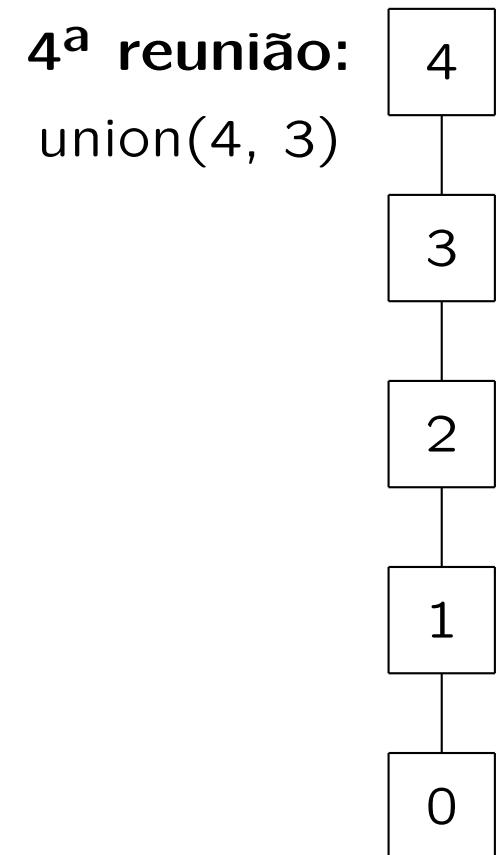
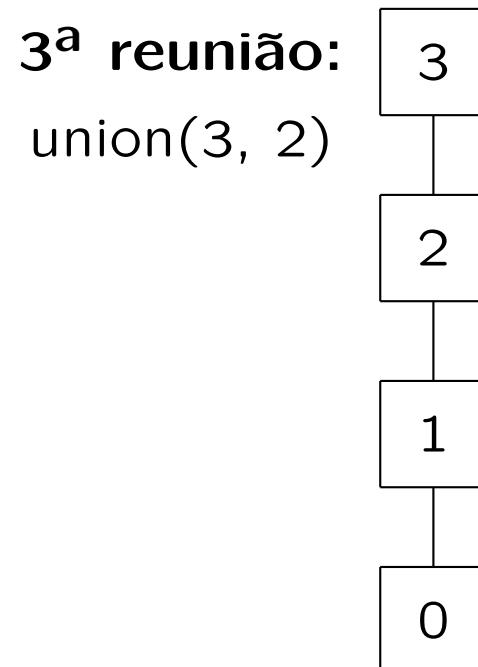
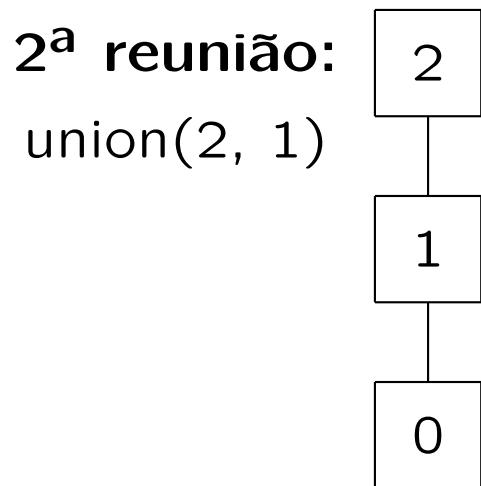
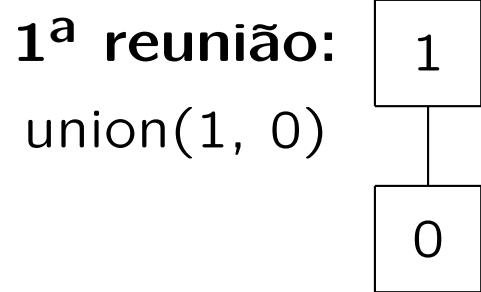
representante $O(?)$

reunião $\Theta(?)$

(no pior caso)

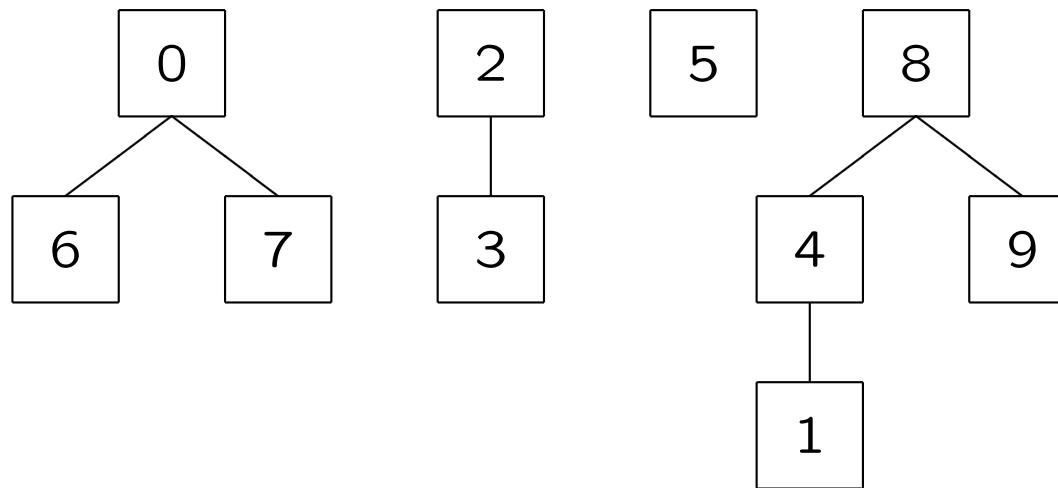
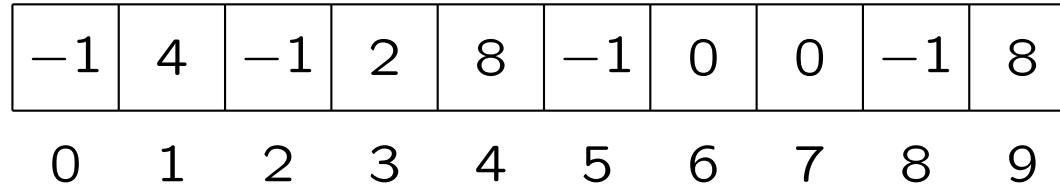
Complexidade do Representante com Reunião **sem Estratégia**

Raiz da nova árvore: primeiro representante



Implementação em Floresta

(implementada em vetor)



Complexidade (com n elementos)

criação $\Theta(n)$

representante $O(n)$

reunião $\Theta(1)$

(no pior caso)

Complexidade do Algoritmo de Kruskal

Reunião sem Estratégia

Representante sem Efeitos Laterais

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times V)$

Complexidade do Algoritmo de Kruskal

Reunião sem Estratégia

Representante sem Efeitos Laterais

Complexidade do Primeiro Ciclo

$$O(|A| \times \log |A| + |A| \times (2R))$$

$$O(|A| \times \underbrace{\log |A|}_{\log |A| < 2 \log |V| \text{ porque } |A| < |V|^2} + |A| \times |V|)$$

$$O(|A| \times \log |V| + |A| \times |V|)$$

$$O(|A| \times |V|)$$

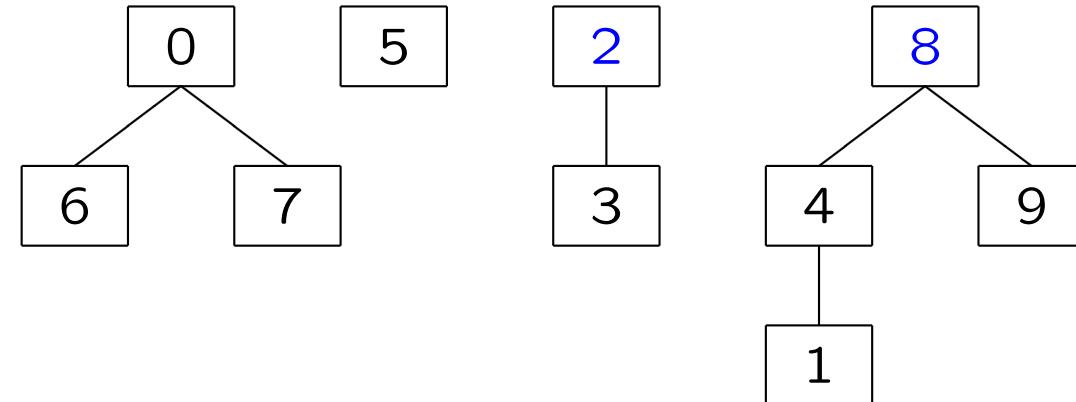
Reunião

por

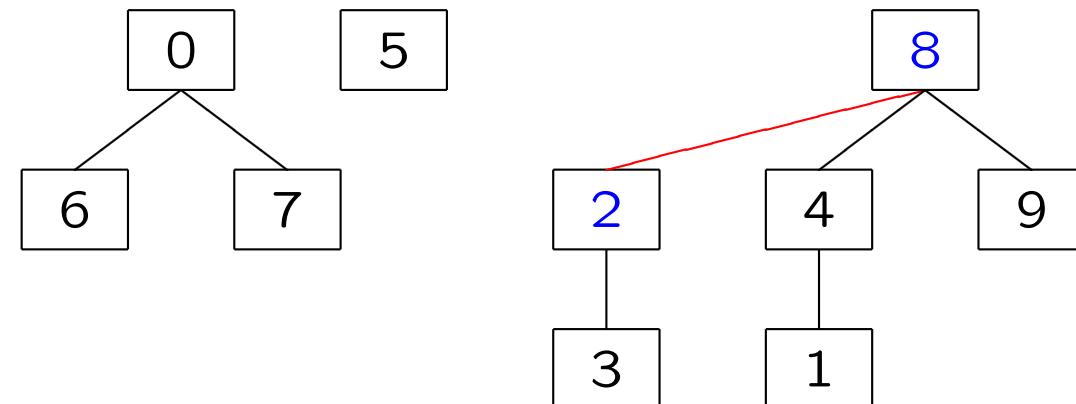
Altura

(2, 8)

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



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



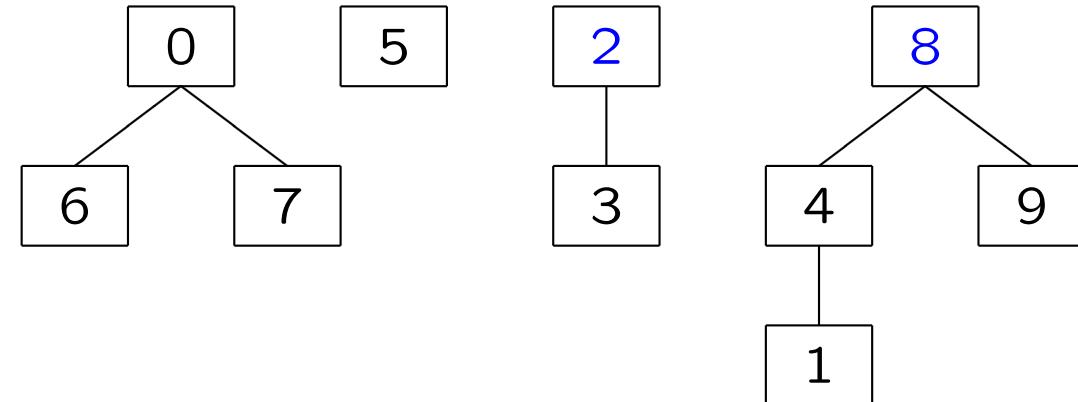
Reunião

por

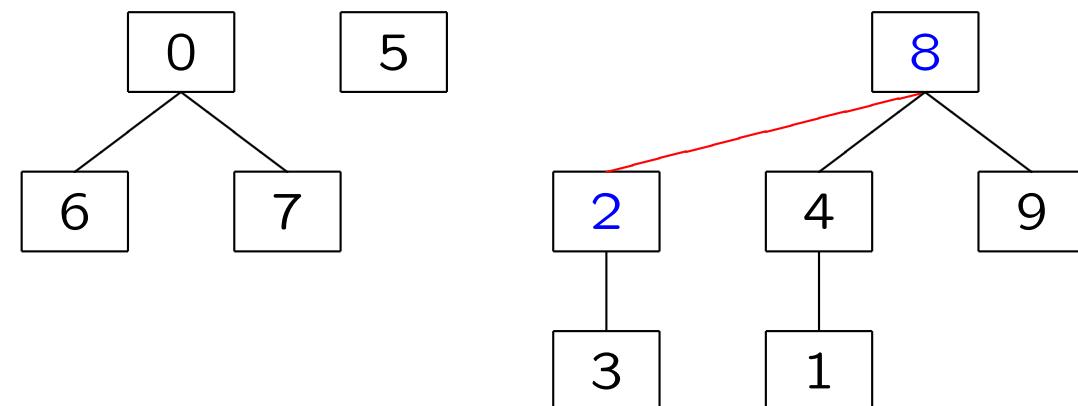
Tamanho

(2, 8)

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



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



Interface Partição (com n elementos)

```
public interface UnionFind {  
  
    // Creates the partition {{0}, {1}, ..., {domainSize - 1}}.  
    // UnionFind( int domainSize );  
  
    // Returns the representative of the set that contains  
    // the specified element.  
    int find( int element ) throws InvalidElementException;  
  
    // Removes the two distinct sets  $S_1$  and  $S_2$  whose representatives  
    // are the specified elements, and inserts the set  $S_1 \cup S_2$ .  
    // The representative of the new set  $S_1 \cup S_2$  can be any of  
    // its members.  
    void union( int representative1, int representative2 ) throws  
        InvalidElementException, NotRepresentativeException,  
        EqualSetsException;  
}
```

Classe Partição em Vetor

```
public class UnionFindInArray implements UnionFind {  
  
    // The partition is a forest implemented in an array.  
    protected int[] partition;  
  
    // Definition of the range of valid elements.  
    protected String validRangeMsg;
```

Criar a Partição

```
// Creates the partition {{0}, {1}, ..., {domainSize - 1}}.  
public UnionFindInArray( int domainSize ) {  
    partition = new int[domainSize];  
    for ( int i = 0; i < domainSize; i++ )  
        partition[i] = -1;  
  
    int lastElement = domainSize - 1;  
    validRangeMsg =  
        "Range of valid elements: 0, 1, ..., " + lastElement;  
}
```

Métodos de Validação

```
protected boolean isInTheDomain( int number ) {  
    return ( number >= 0 ) && ( number < partition.length );  
}
```

// Pre-condition: $0 \leq element < partition.length$.

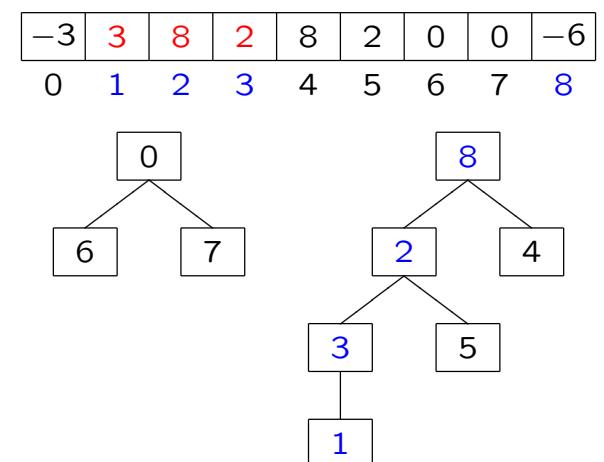
```
protected boolean isRepresentative( int element ) {  
    return partition[element] < 0;  
}
```

Representante — Recursivo

```
public int find( int element ) throws InvalidElementException {  
    if ( !this.isInTheDomain(element) )  
        throw new InvalidElementException(validRangeMsg);  
  
    return this.findRec(element);  
}
```

// Pre-condition: $0 \leq \text{element} < \text{partition.length}$.

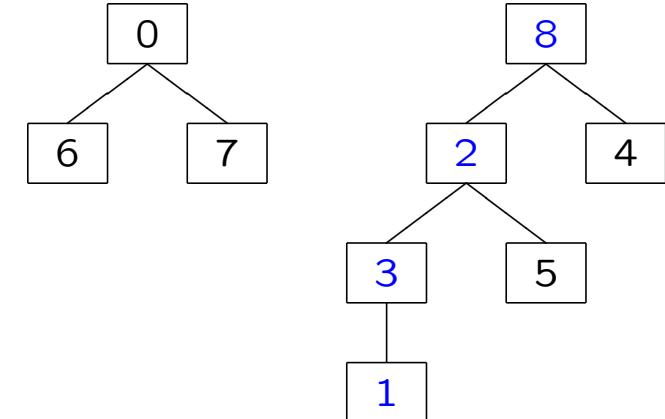
```
protected int findRec( int element ) {  
    if ( partition[element] < 0 )  
        return element;  
  
    return this.findRec( partition[element] );  
}
```



Representante — Iterativo

```
public int find( int element ) throws InvalidElementException {  
    if ( !this.isInTheDomain(element) )  
        throw new InvalidElementException(validRangeMsg);  
  
    int node = element;  
  
    while ( partition[node] >= 0 )  
        node = partition[node];  
  
    return node;  
}
```

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

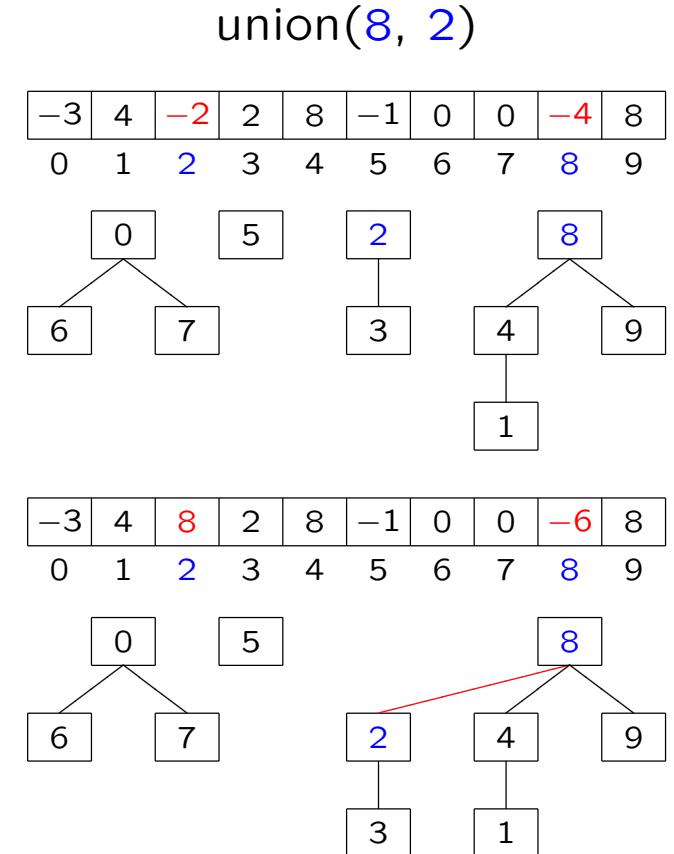


Reunião por Tamanho (*Union by Size*)

```
public void union( int rep1, int rep2 ) throws  
    InvalidElementException, NotRepresentativeException,  
    EqualSetsException {  
    if ( !this.isInTheDomain(rep1) || !this.isInTheDomain(rep2) )  
        throw new InvalidElementException(validRangeMsg);  
    if ( !this.isRepresentative(rep1) )  
        throw new NotRepresentativeException("First argument");  
    if ( !this.isRepresentative(rep2) )  
        throw new NotRepresentativeException("Second argument");  
    if ( rep1 == rep2 )  
        throw new EqualSetsException("The two arguments are equal");
```

Reunião por Tamanho

```
if ( partition[rep1] <= partition[rep2] ) {  
    // Size(S1) >= Size(S2).  
    partition[rep1] += partition[rep2];  
    partition[rep2] = rep1;  
}  
  
else {  
    // Size(S1) < Size(S2).  
    partition[rep2] += partition[rep1];  
    partition[rep1] = rep2;  
}  
}
```



Reunião por Altura (*Union by Height*)

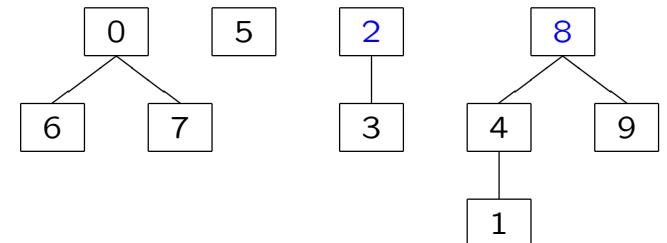
```
public void union( int rep1, int rep2 ) throws  
    InvalidElementException, NotRepresentativeException,  
    EqualSetsException {  
    if ( !this.isInTheDomain(rep1) || !this.isInTheDomain(rep2) )  
        throw new InvalidElementException(validRangeMsg);  
    if ( !this.isRepresentative(rep1) )  
        throw new NotRepresentativeException("First argument");  
    if ( !this.isRepresentative(rep2) )  
        throw new NotRepresentativeException("Second argument");  
    if ( rep1 == rep2 )  
        throw new EqualSetsException("The two arguments are equal");
```

Reunião por Altura

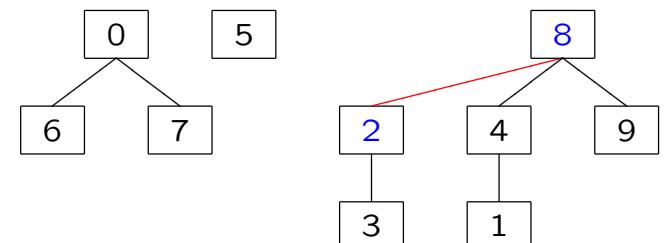
```
if ( partition[rep1] <= partition[rep2] ) {  
    // Height(S1) >= Height(S2).  
  
    if ( partition[rep1] == partition[rep2] )  
        partition[rep1]--;  
        partition[rep2] = rep1;  
    }  
  
else  
    // Height(S1) < Height(S2).  
    partition[rep1] = rep2;  
}
```

union(8, 2)

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

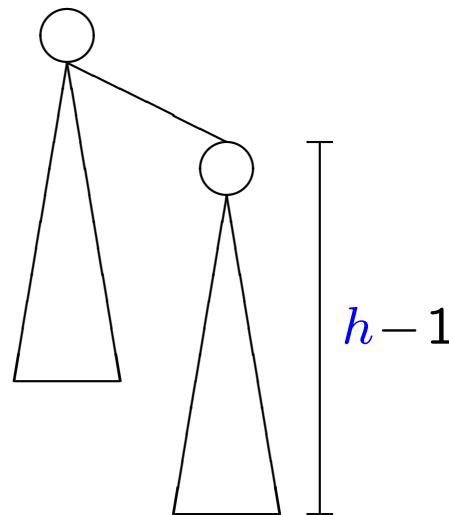


-2	4	8	2	8	-1	0	0	-3	8
0	1	2	3	4	5	6	7	8	9



Complexidade do Representante com Reunião por **Altura**

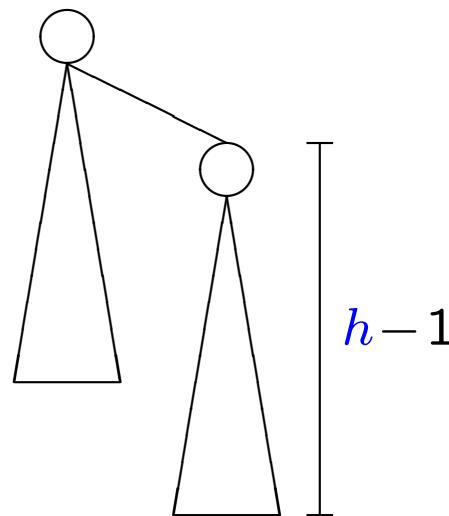
Número Mínimo de Nós
de uma Árvore com Altura h



$$\begin{aligned}N(1) &= 1 & = 2^0 \\N(2) &= 2 & = 2^1 \\N(h) &= 2 \times N(h-1) \stackrel{H.I.}{=} 2 \times 2^{h-2} = 2^{h-1} \\&& (\text{para } h \geq 2)\end{aligned}$$

Complexidade do Representante com Reunião por **Tamanho**

Número Mínimo de Nós
de uma Árvore com Altura h

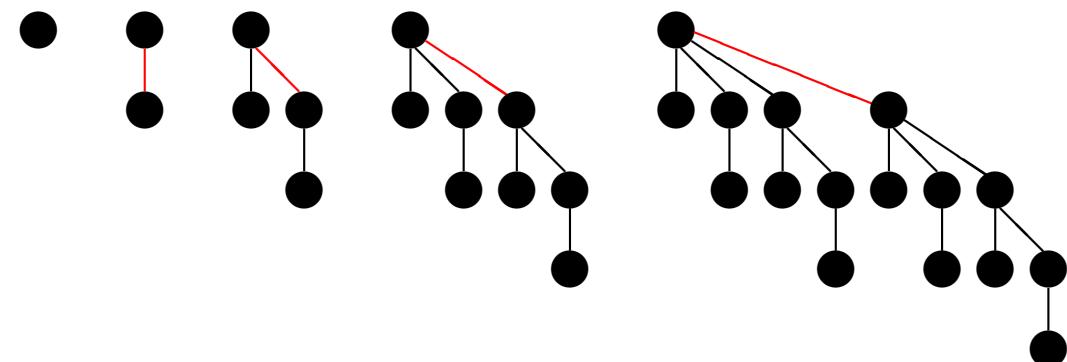


$$\begin{aligned}N(1) &= 1 & = 2^0 \\N(2) &= 2 & = 2^1 \\N(h) &= 2 \times N(h-1) \stackrel{H.I.}{=} 2 \times 2^{h-2} = 2^{h-1} \\&& (\text{para } h \geq 2)\end{aligned}$$

Complexidade do Representante com Reunião por **Altura** ou por **Tamanho**

Número Mínimo de Nós
de uma Árvore com Altura h

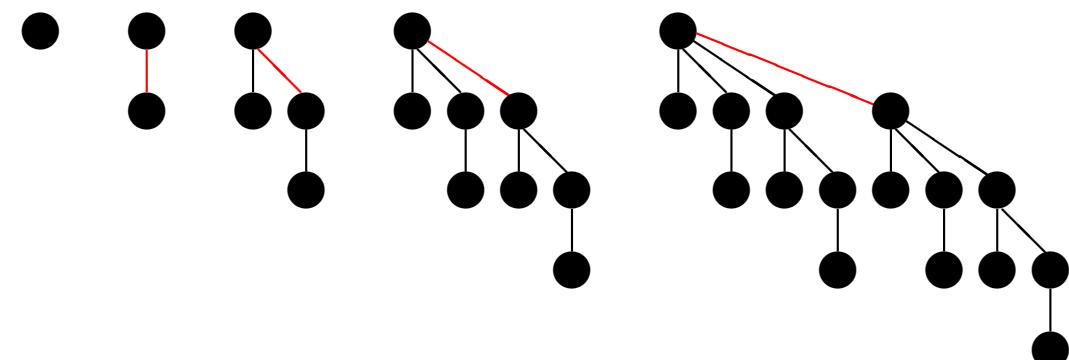
Altura h	Nº mín. nós
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
5	$16 = 2^4$
6	$32 = 2^5$
7	$64 = 2^6$



Complexidade do Representante com Reunião por **Altura** ou por **Tamanho**

Altura Máxima de uma Árvore com n nós

Altura h	Nº mín. nós
1	$1 = 2^0$
2	$2 = 2^1$
3	$4 = 2^2$
4	$8 = 2^3$
5	$16 = 2^4$
6	$32 = 2^5$
7	$64 = 2^6$



Qual é a maior altura de uma árvore, se o domínio tiver $n = 20$ nós?

Complexidade do Representante com Reunião por **Altura** ou por **Tamanho**

Altura Máxima de uma Árvore com n nós

Dado n (o número total de nós), existe h tal que:

$$\underbrace{2^{h-1}}_{\begin{array}{l} h \text{ é a maior altura} \\ \text{com } n \text{ nós} \end{array}} \leq n < \underbrace{2^h}_{\begin{array}{l} \text{número mínimo de nós} \\ \text{para altura } h + 1 \end{array}}$$

$$2^{h-1} \leq n$$

$$h - 1 \leq \log(n)$$

$$h \leq 1 + \log(n)$$

Complexidade do Algoritmo de Kruskal

Reunião por Altura ou por Tamanho

Representante sem Efeitos Laterais

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(\log V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times \log V)$

Complexidade do Algoritmo de Kruskal

Reunião por Altura ou por Tamanho

Representante sem Efeitos Laterais

Complexidade do Primeiro Ciclo

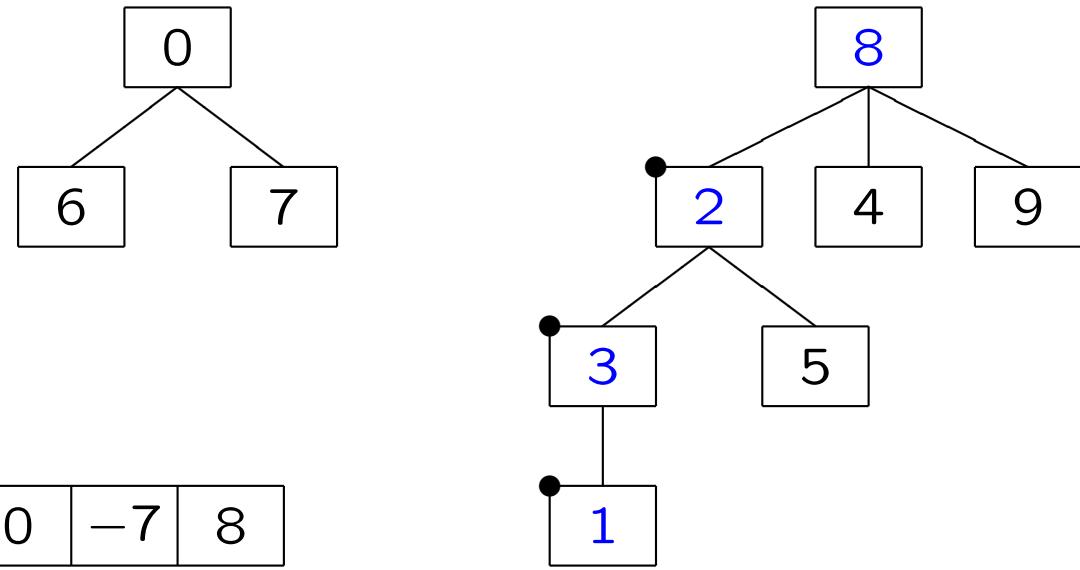
$$O(|A| \times \log |A| + |A| \times (2\mathbf{R}))$$

$$O(|A| \times \log |V| + |A| \times \log |V|)$$

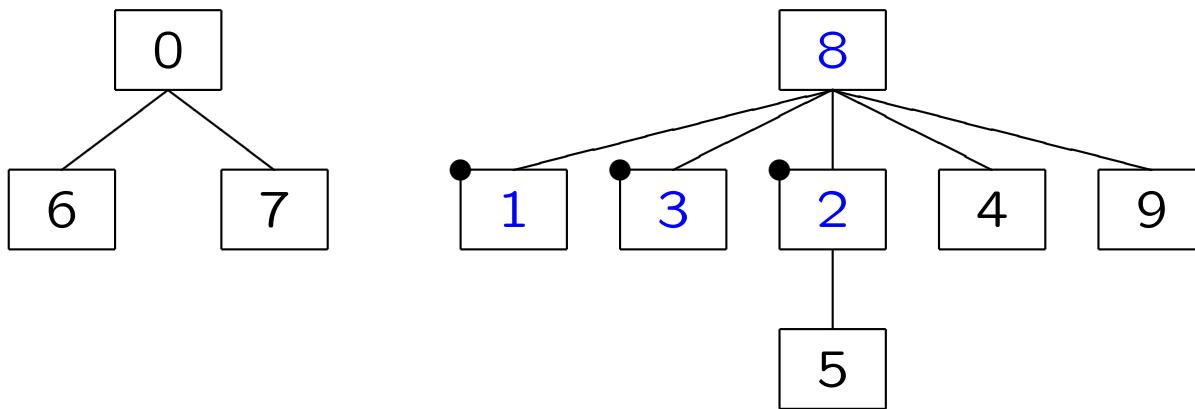
$$O(|A| \times \log |V|)$$

Compressão do Caminho **find(1)**

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



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



Representante com Compressão do Caminho

```
public int find( int element ) throws InvalidElementException {  
    if ( !this.isInTheDomain(element) )  
        throw new InvalidElementException(validRangeMsg);  
  
    return this.findPathCompr(element);  
}  
  
protected int findPathCompr( int element ) {  
    if ( partition[element] < 0 )  
        return element;  
  
    int root = this.findPathCompr( partition[element] );  
    partition[element] = root;  
  
    return root;  
}
```

Complexidade no PIOR CASO de U operações de **reunião** e R operações de **representante** (com n elementos)

Reunião sem Estratégia	$\Theta(1)$	$O(U + Rn)$
Representante sem Efeitos Laterais	$O(n)$	

Reunião por Altura ou Tamanho	$\Theta(1)$	$O(U + R \log n)$
Representante sem Efeitos Laterais	$O(\log n)$	

Reunião por Nível (código Altura) ou Tamanho	$\Theta(1)$	$O(k \alpha(k, n))$
Representante com Compressão do Caminho	$O(\log n)$	

se $k = U + R \geq n$ [Tarjan 75].

Valor de $\alpha(k,n)$ ($k \geq n \geq 1$)

$$2^2$$

$$4$$

$$2^{2^2}$$

$$2^4$$

$$16$$

$$2^{2^{2^2}}$$

$$2^{16}$$

$$65536$$

$$2^{2^{2^{2^2}}}\}^4$$

$$2^{65536}$$

$\approx 20\,000$ algarismos

Na prática,

$$\alpha(k,n) \leq 4$$

porque

$$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}^{16} > \log n.$$

Complexidade do Algoritmo de Kruskal

Reunião por Nível ou por Tamanho

Representante com Compressão do Caminho

criação do heap	$\Theta(A)$
criação da partição	$\Theta(V)$
criação do vetor resultado	$\Theta(1)$
Ciclo (executado entre $ V - 1$ e $ A $ vezes)	(ver próximo slide)
1 remoção do mínimo	$O(\log A)$
2 representante	$O(\log V)$
Ciclo (executado $ V - 1$ vezes)	
1 inserção no vetor	$\Theta(1)$
1 reunião	$\Theta(1)$
TOTAL	$O(A \times \log V)$

Complexidade do Algoritmo de Kruskal

Reunião por Nível ou por Tamanho

Representante com Compressão do Caminho

Complexidade do Primeiro Ciclo

$$O(|A| \times \log |A| + \underbrace{|A| \times (2\mathbf{R})}_{2|A| \geq 2(|V|-1) \geq |V|})$$

$$O(|A| \times \log |V| + (2|A|) \underbrace{\alpha(2|A|, |V|)}_{\leq 4})$$

$$O(|A| \times \log |V| + |A|)$$

$$O(|A| \times \log |V|)$$

Capítulo VIII

Complexidade Amortizada (Amortized Analysis)

Complexidade Amortizada

Objetivo: analisar o **custo de uma sequência** de operações numa estrutura de dados (ED), **no pior caso**.

Interesse: mostrar que, embora alguma operação possa ser “cara”, o custo total da sequência de operações é “baixo”.

Não é a complexidade no caso esperado (que indica o custo médio, considerando todas as distribuições da entrada).

Não envolve probabilidades.

Pilha com MultiDesempilha

```
void push( E element );                                // Pior caso: Θ(1).  
  
E pop( );                                              // Pior caso: Θ(1).  
  
void multiPop( int k ) {                             // Pior caso: Θ(min(s, k))  
    while ( !this.isEmpty() && k > 0 ) {           // onde s é o número de  
        E element = this.pop();                     // elementos na pilha.  
        k--;  
    }  
}
```

Qual é a complexidade (no pior caso) de uma sequência de n operações de `push`, `pop` e `multiPop`, numa pilha inicialmente vazia?

Contador Binário

```
void increment( int[] counter ) { // Pior caso:  $\Theta(c)$ , onde
    int pos = 0; // c é a capacidade do vetor.
    while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;
        pos++;
    }
    if ( pos < counter.length )
        counter[pos] = 1;
}
```

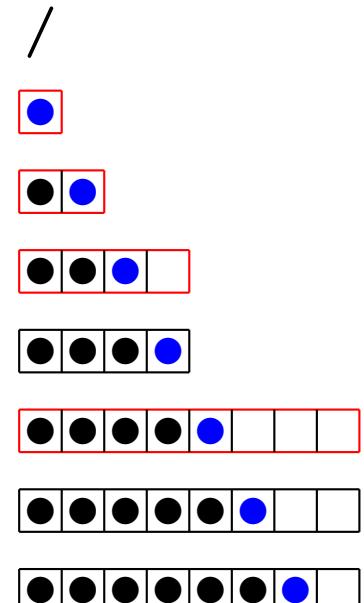
Qual é a complexidade (no pior caso) de uma sequência de n operações de `increment`, num contador inicialmente a zero?

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
.....
1	1	1	1
0	0	0	0

Tabela Dinâmica

```
// int currentSize, E[] table (preenchida de 0 a currentSize – 1).  
  
void insert( E element ) { // Pior caso:  $\Theta(s)$ , onde  
    if ( table == null ) // s é o número de elementos na tabela.  
        table = new E[1];  
    else if ( currentSize == table.length ) {  
        E[] newTable = new E[ 2 * currentSize ];  
        System.arraycopy(table, 0, newTable, 0, currentSize);  
        table = newTable;  
    }  
    table[ currentSize++ ] = element;  
}
```

Qual é a complexidade (no pior caso) de uma sequência de n operações de `insert`, numa tabela inicialmente vazia?



Métodos Existentes

- Há três métodos (com algumas variantes).
 - Agregação.
 - Contabilidade.
 - Potencial: é o mais versátil e o único que será estudado.
- Em todos os métodos, calcula-se um **majorante do custo total** da sequência de operações.
A esse majorante, chama-se **custo total amortizado**.
- Ao verdadeiro custo total, chama-se **custo total real**.
- **O custo total amortizado nunca é inferior ao custo total real.**

Ideia Geral do Método do Potencial

- Define-se uma função potencial Φ , que atribui a cada ED D um número real $\Phi(D)$.
- Prova-se que a função potencial Φ verifica algumas propriedades.
- Sejam:
 - D uma estrutura de dados,
 - c o custo real da operação efetuada sobre D e
 - D' a estrutura de dados resultante.

O custo amortizado da operação, que se denota por \hat{c} , é:

$$\hat{c} = c + \Phi(D') - \Phi(D).$$

- O custo amortizado de uma operação pode ser superior, igual ou inferior ao custo real da operação.

Justificação do Método do Potencial (1)

- Para cada $i = 1, 2, \dots, n$, sejam:
 - D_0 a ED inicial;
 - D_i a ED depois da operação i ; e
 - c_i o custo real da operação i .

Então:

- o **custo amortizado da operação i** é

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1});$$

- o **custo total amortizado** (da sequência de n operações) é

$$\begin{aligned}\sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= (\sum_{i=1}^n c_i) + \Phi(D_n) - \Phi(D_0).\end{aligned}$$

Justificação do Método do Potencial (2)

- É necessário garantir que **o custo total amortizado nunca é inferior ao custo total real**. Como o custo total amortizado é

$$\sum_{i=1}^n \hat{c}_i = \left(\sum_{i=1}^n c_i \right) + \Phi(D_n) - \Phi(D_0),$$

basta assegurar que, para qualquer $i = 1, 2, \dots, n$:

$$\Phi(D_i) \geq \Phi(D_0).$$

- Na prática, define-se Φ de forma a que:

$$\text{(P1)} \quad \Phi(D_0) = 0; \text{ e}$$

$$\text{(P2)} \quad \Phi(D_i) \geq 0, \text{ para qualquer } i \geq 1.$$

E diz-se que Φ é uma função potencial **válida**.

Aplicação do Método do Potencial

- Define-se uma função potencial Φ , que atribui a cada ED D um número real $\Phi(D)$.
- Prova-se que a função potencial Φ é **válida**, i.e., que Φ verifica as propriedades:
 - (P1) $\Phi(D_0) = 0$, onde D_0 é a ED inicial (acabada de criar); e
 - (P2) $\Phi(D) \geq 0$, para qualquer ED D (podendo-se excluir D_0).
- Calcula-se o custo amortizado \hat{c} de cada operação com a equação:

$$\hat{c} = c + \Phi(D') - \Phi(D)$$

onde c é o custo real da operação, D é a estrutura de dados **antes** da operação e D' é a estrutura de dados **depois** da operação.

Pilha com MultiDesempilha

```
void push( E element );                                // Pior caso: Θ(1).  
  
E pop( );                                              // Pior caso: Θ(1).  
  
void multiPop( int k ) {                            // Pior caso: Θ(min(s, k))  
    while ( !this.isEmpty() && k > 0 ) { // onde s é o número de  
        E element = this.pop();           // elementos na pilha.  
        k--;  
    }  
}
```

Qual é a complexidade (no pior caso) de uma sequência de n operações de `push`, `pop` e `multiPop`, numa pilha inicialmente vazia?

Potencial — Pilha (1)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

A função Φ é **válida**:

(P1) $\Phi(P_0) = 0$, onde P_0 é a pilha inicial,

porque P_0 é uma pilha vazia (tem zero elementos).

(P2) $\Phi(P) \geq 0$, para qualquer pilha P ,

porque o número de elementos em P não pode ser negativo.

Portanto, o custo total amortizado nunca será inferior ao custo total real.

O custo amortizado de uma operação é $\hat{c} = c + \Delta\Phi$.

Potencial — Pilha (2)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
	c	$\Phi(P') - \Phi(P)$	$\hat{c} = c + \Delta\Phi$
push	1		
pop	1		
multiPop k	$\min(k, s)$		

Notação: s antes / s' depois da operação

Potencial — Pilha (3)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

Operação	Custo Real c	Dif. de Potencial $\Phi(P') - \Phi(P)$	Custo Amortizado $\hat{c} = c + \Delta\Phi$
push	1	1	$1 + 1 = 2$ $O(1)$
pop	1		
multiPop k	$\min(k, s)$		

Notação: s antes / s' depois da operação

Diferença de Potencial de push:

$$s' - s = (s + 1) - s = 1$$

Potencial — Pilha (4)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

Operação	Custo Real c	Dif. de Potencial $\Phi(P') - \Phi(P)$	Custo Amortizado $\hat{c} = c + \Delta\Phi$
push	1	1	$1 + 1 = 2$ $O(1)$
pop	1	-1	$1 - 1 = 0$ $O(1)$
multiPop k	$\min(k, s)$		

Notação: s antes / s' depois da operação

Diferença de Potencial de pop:

$$s' - s = (s - 1) - s = -1$$

Potencial — Pilha (5)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

Operação	Custo Real c	Dif. de Potencial $\Phi(P') - \Phi(P)$	Custo Amortizado $\hat{c} = c + \Delta\Phi$
push	1	1	$1 + 1 = 2$ $O(1)$
pop	1	-1	$1 - 1 = 0$ $O(1)$
multiPop k	$\min(k, s)$	$-\min(k, s)$	0 $O(1)$

Notação: s antes / s' depois da operação

Diferença de Potencial de **multiPop k :**

$$s' - s = (s - \min(k, s)) - s = -\min(k, s)$$

Potencial — Pilha (6)

Seja P uma pilha qualquer e seja s o número de elementos em P .

$$\Phi(P) = s.$$

Operação	Custo Real c	Dif. de Potencial $\Phi(P') - \Phi(P)$	Custo Amortizado $\hat{c} = c + \Delta\Phi$
push	1	1	2 $O(1)$
pop	1	-1	0 $O(1)$
multiPop k	$\min(k, s)$	$-\min(k, s)$	0 $O(1)$

Notação: s antes / s' depois da operação

Conclusões: A complexidade amortizada do push, do pop e do multiPop é $O(1)$. A complexidade de uma sequência de n operações de push, pop e multiPop, numa pilha inicialmente vazia, é $O(n)$.

Pilha — Exemplo

	Estado da Pilha	Custo Real da Operação	Total	Custo Amortizado da Operação	Total
Criação					
push e_1	e_1	1	1	2	2
push e_2	$e_1 e_2$	1	2	2	4
push e_3	$e_1 e_2 e_3$	1	3	2	6
pop	$e_1 e_2$	1	4	0	6
push e_4	$e_1 e_2 e_4$	1	5	2	8
push e_5	$e_1 e_2 e_4 e_5$	1	6	2	10
mPop 4		4	10	0	10
push e_6	e_6	1	11	2	12
mPop 3		1	12	0	12
push e_7	e_7	1	13	2	14

O **custo total amortizado** nunca é inferior ao **custo total real**.

Contador Binário

```
void increment( int[] counter ) { // Pior caso:  $\Theta(c)$ , onde
    int pos = 0; // c é a capacidade do vetor.
    while ( pos < counter.length && counter[pos] == 1 ) {
        counter[pos] = 0;
        pos++;
    }
    if ( pos < counter.length )
        counter[pos] = 1;
}
```

Qual é a complexidade (no pior caso) de uma sequência de n operações de `increment`, num contador inicialmente a zero?

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
.....
1	1	1	1
0	0	0	0

Potencial — Contador (1)

Seja C um contador qualquer e seja u o número de UNS em C .

$$\Phi(C) = u.$$

A função Φ é **válida**:

- (P1) $\Phi(C_0) = 0$, onde C_0 é o contador inicial,
porque C_0 só tem ZEROS.
- (P2) $\Phi(C) \geq 0$, para qualquer contador C ,
porque o número de UNS em C não pode ser negativo.

Portanto, o custo total amortizado nunca será inferior ao custo total real.

O custo amortizado de uma operação é $\hat{c} = c + \Delta\Phi$.

Potencial — Contador (2)

Seja C um contador qualquer e seja u o número de UNS em C .

$$\Phi(C) = u.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
increment	c	$\Phi(C') - \Phi(C)$	$\hat{c} = c + \Delta\Phi$
incrementa	$k + 1$		
anula	k		

k é o número de UNS que passam a ZERO

Notação: u antes / u' depois da operação

Potencial — Contador (3)

Seja C um contador qualquer e seja u o número de UNS em C .

$$\Phi(C) = u.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
increment	c	$\Phi(C') - \Phi(C)$	$\hat{c} = c + \Delta\Phi$
incrementa	$k + 1$	$-k + 1$	2
anula	k		

k é o número de UNS que passam a ZERO

Notação: u antes / u' depois da operação

Diferença de Potencial quando incrementa:

$$u' - u = (u - k + 1) - u = -k + 1$$

Potencial — Contador (4)

Seja C um contador qualquer e seja u o número de UNS em C .

$$\Phi(C) = u.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
increment	c	$\Phi(C') - \Phi(C)$	$\hat{c} = c + \Delta\Phi$
incrementa	$k + 1$	$-k + 1$	2
anula	k	$-k$	0

k é o número de UNS que passam a ZERO

Notação: u antes / u' depois da operação

Diferença de Potencial quando anula:

$$u' - u = (u - k) - u = -k$$

Potencial — Contador (5)

Seja C um contador qualquer e seja u o número de UNS em C .

$$\Phi(C) = u.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
increment	c	$\Phi(C') - \Phi(C)$	$\hat{c} = c + \Delta\Phi$
incrementa	$k + 1$	$-k + 1$	2
anula	k	$-k$	0

k é o número de UNS que passam a ZERO

Notação: u antes / u' depois da operação

Conclusões: A complexidade amortizada do increment é $O(1)$. A complexidade de uma sequência de n operações de increment, num contador inicialmente a zero, é $O(n)$.

Contador — Exemplo

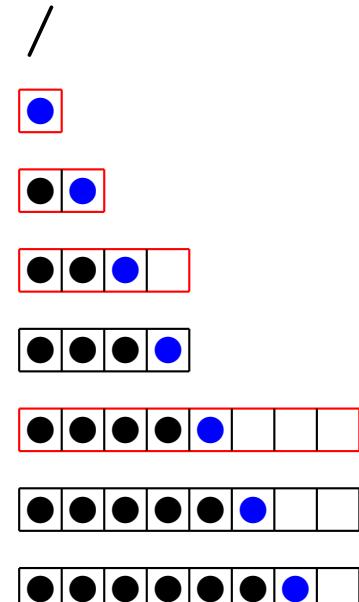
	Estado do Contador	Custo Real da Operação	Total	Custo Amortizado da Operação	Total
Criação	000				
incr.	001	1	1	2	2
incr.	010	2	3	2	4
incr.	011	1	4	2	6
incr.	100	3	7	2	8
incr.	101	1	8	2	10
incr.	110	2	10	2	12
incr.	111	1	11	2	14
incr.	000	3	14	0	14
incr.	001	1	15	2	16

O **custo total amortizado** nunca é inferior ao **custo total real**.

Tabela Dinâmica

```
// int currentSize, E[] table (preenchida de 0 a currentSize – 1).  
  
void insert( E element ) { // Pior caso:  $\Theta(s)$ , onde  
    if ( table == null ) // s é o número de elementos na tabela.  
        table = new E[1];  
    else if ( currentSize == table.length ) {  
        E[] newTable = new E[ 2 * currentSize ];  
        System.arraycopy(table, 0, newTable, 0, currentSize);  
        table = newTable;  
    }  
    table[ currentSize++ ] = element;  
}
```

Qual é a complexidade (no pior caso) de uma sequência de n operações de `insert`, numa tabela inicialmente vazia?



Potencial — Tabela (1)

Seja T uma tabela qualquer e sejam s o número de elementos em T e l a capacidade de T .

$$\Phi(T) = 2s - l.$$

A função Φ é **válida**:

(P1) $\Phi(T_0) = 0$, onde T_0 é a tabela inicial,
porque T_0 tem 0 elementos e capacidade 0.

(P2) $\Phi(T) \geq 0$, para qualquer tabela T exceto a inicial.
Como o fator de ocupação da tabela é superior a 0.5,

$$s > \frac{1}{2}l$$

$$2s > l$$

$$\Phi(T) > 0.$$

Potencial — Tabela (2)

Seja T uma tabela qualquer e sejam s o número de elementos em T e l a capacidade de T .

$$\Phi(T) = 2s - l.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
insert	c	$\Phi(T') - \Phi(T)$	$\hat{c} = c + \Delta\Phi$
não expande	1		
expande	$s + 1$		

Notação: (s, l) antes / (s', l') depois da operação

Potencial — Tabela (3)

Seja T uma tabela qualquer e sejam s o número de elementos em T e l a capacidade de T .

$$\Phi(T) = 2s - l.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
insert	c	$\Phi(T') - \Phi(T)$	$\hat{c} = c + \Delta\Phi$
não expande	1	2	3
expande	$s + 1$		

Notação: (s, l) antes / (s', l') depois da operação

Diferença de Potencial quando não expande:

$$\begin{aligned}(2s' - l') - (2s - l) &= (2(s + 1) - l) - (2s - l) \\ &= 2s + 2 - l - 2s + l = 2\end{aligned}$$

Potencial — Tabela (4)

Seja T uma tabela qualquer e sejam s o número de elementos em T e l a capacidade de T .

$$\Phi(T) = 2s - l.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
insert	c	$\Phi(T') - \Phi(T)$	$\hat{c} = c + \Delta\Phi$
não expande	1	2	3
expande	$s + 1$	$2 - s$	3

Notação: (s, l) antes / (s', l') depois da operação

Diferença de Potencial quando expande (e $s = l$):

$$\begin{aligned}
 (2s' - l') - (2s - l) &= (2(s + 1) - 2l) - (2s - l) \\
 &= 2s + 2 - 2l - 2s + l = 2 - l = 2 - s
 \end{aligned}$$

Potencial — Tabela (5)

Seja T uma tabela qualquer e sejam s o número de elementos em T e l a capacidade de T .

$$\Phi(T) = 2s - l.$$

Operação	Custo Real	Dif. de Potencial	Custo Amortizado
insert	c	$\Phi(T') - \Phi(T)$	$\hat{c} = c + \Delta\Phi$
não expande	1	2	3
expande	$s + 1$	$2 - s$	3

Notação: (s, l) antes / (s', l') depois da operação

Conclusões: A complexidade amortizada do insert é $O(1)$. A complexidade de uma sequência de n operações de insert, numa tabela inicialmente vazia, é $O(n)$.

Tabela — Exemplo

	Estado da Tabela		Custo Real da Operação	Total	Custo Amortizado da Operação	Total
	length	size				
Inicial	0	0				
ins. e_1	1	1	1	1	3	3
ins. e_2	2	2	2	3	3	6
ins. e_3	4	3	3	6	3	9
ins. e_4	4	4	1	7	3	12
ins. e_5	8	5	5	12	3	15
ins. e_6	8	6	1	13	3	18
ins. e_7	8	7	1	14	3	21
ins. e_8	8	8	1	15	3	24
ins. e_9	16	9	9	24	3	27

O **custo total amortizado** nunca é inferior ao **custo total real**.

Complexidade no PIOR CASO de U operações de **reunião** e R operações de **representante** (com n elementos)

Reunião por **Nível** (código Altura) ou Tamanho $\Theta(1)$
Representante com Compressão do Caminho $O(\log n)$ }
se $k = U + R \geq n$ [Tarjan 75]. } $O(k \alpha(k, n))$

Resultados: a **complexidade amortizada** do **find** (com compressão do caminho) e do **union** (por nível ou por tamanho) é $O(\alpha(k, n))$, se se realizarem $k \geq n$ operações.

A complexidade de uma sequência de k operações de **find** (com compressão do caminho) e **union** (por nível ou por tamanho), numa partição com n elementos, acabada de criar, é $O(k \alpha(k, n))$, se $k \geq n$.

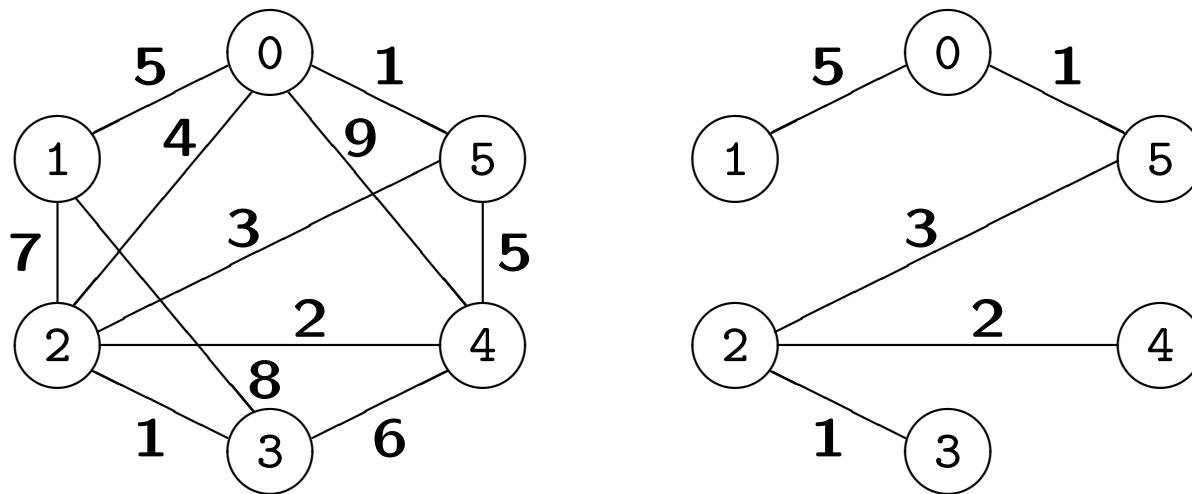
Capítulo IX

Árvore Mínima de Cobertura
(num grafo não orientado, conexo e pesado)

Algoritmo de Prim

Problema

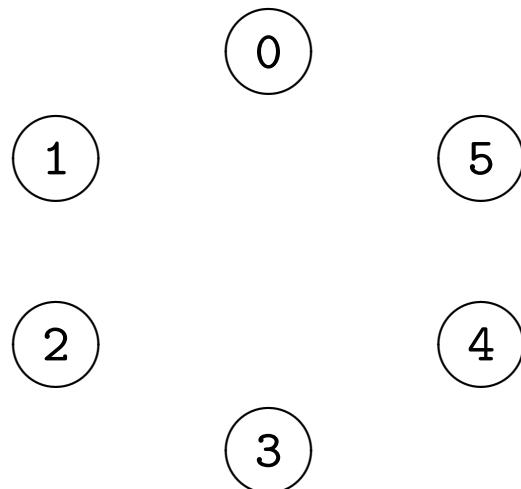
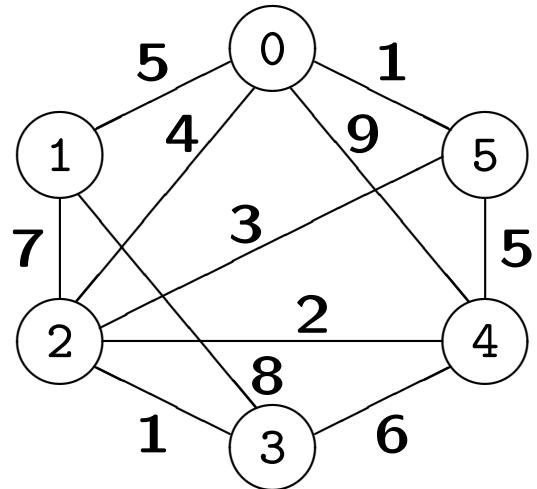
Como ligar um dado equipamento, minimizando a soma dos comprimentos das ligações?



Árvore de Cobertura (sub-grafo acíclico e conexo com todos os vértices) de custo Mínimo (nenhuma árvore de cobertura tem custo menor).

Dado um grafo **não orientado, conexo e pesado**, como obter uma **Árvore Mínima de Cobertura**?

Algoritmo de Prim [1957]

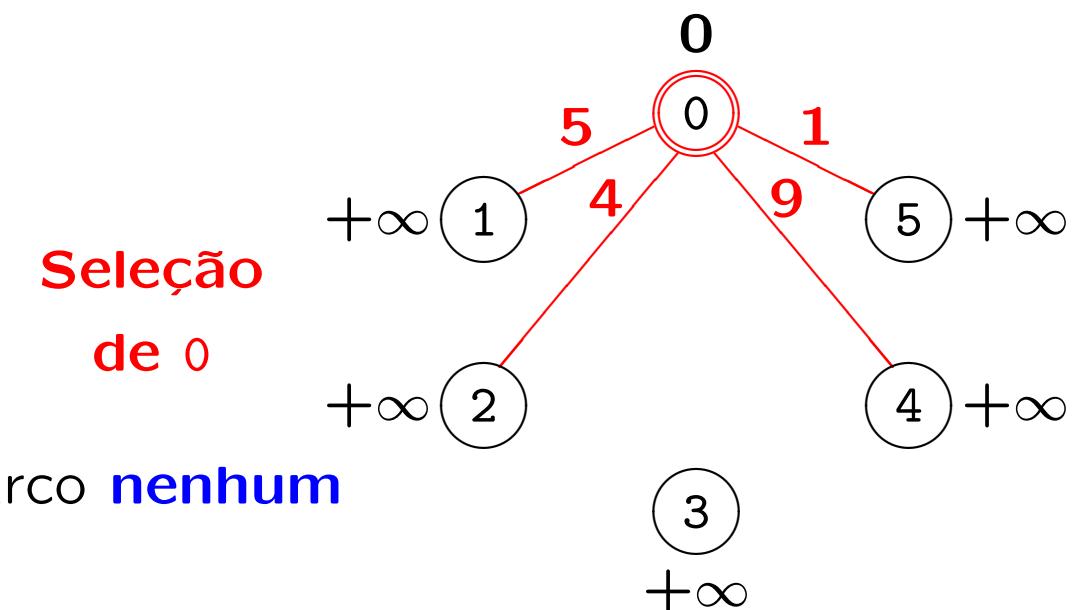
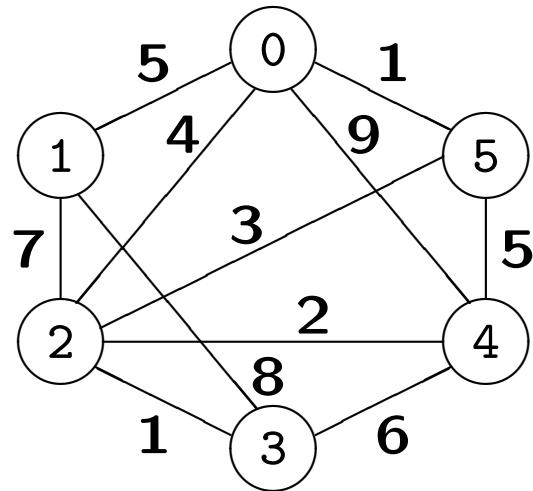


Algoritmo Greedy

Construir a árvore,
partindo de um vértice origem qualquer o
e selecionando, em cada passo:

- um novo vértice v e
- um arco incidente em v , se $v \neq o$.

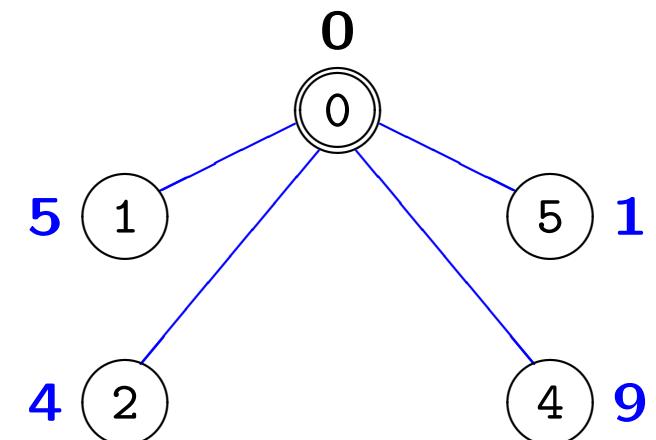
Algoritmo de Prim (1)



5 $+\infty$ origem 0

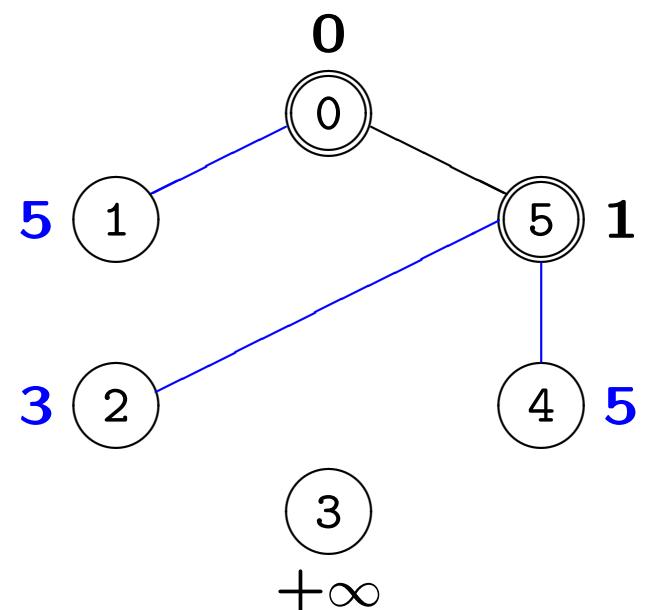
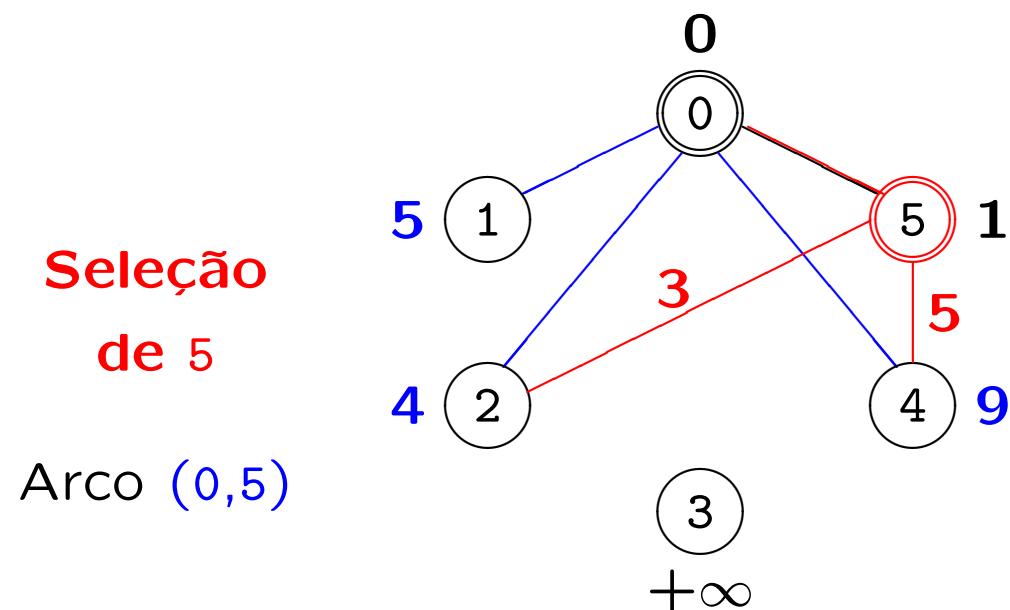
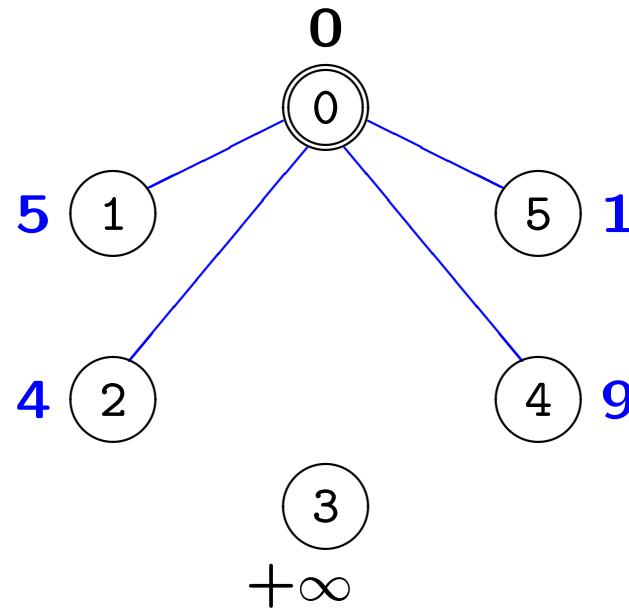
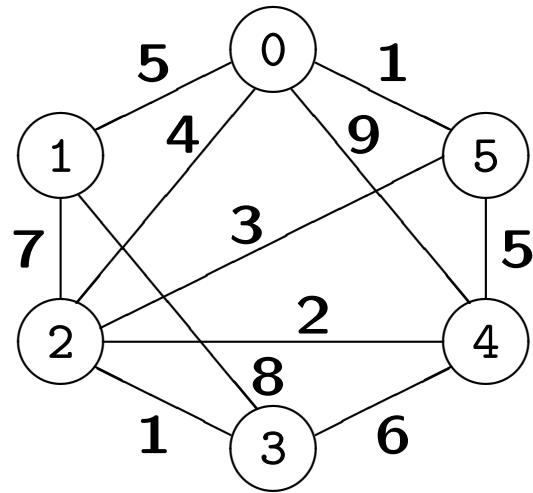
4 $+\infty$

3 $+\infty$

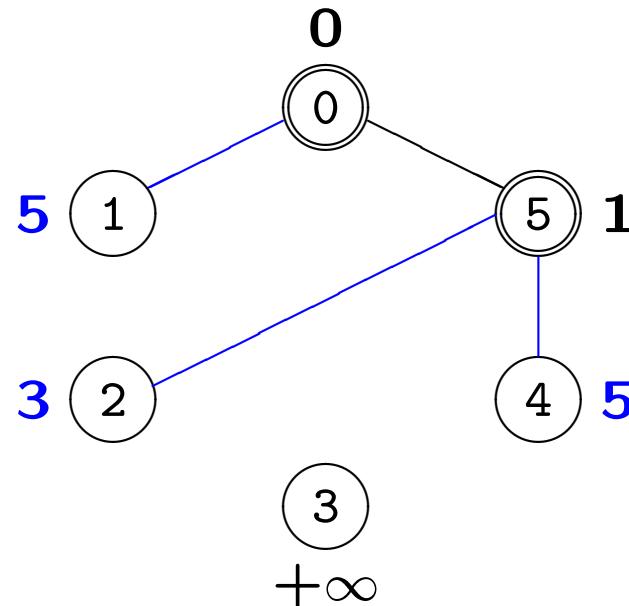
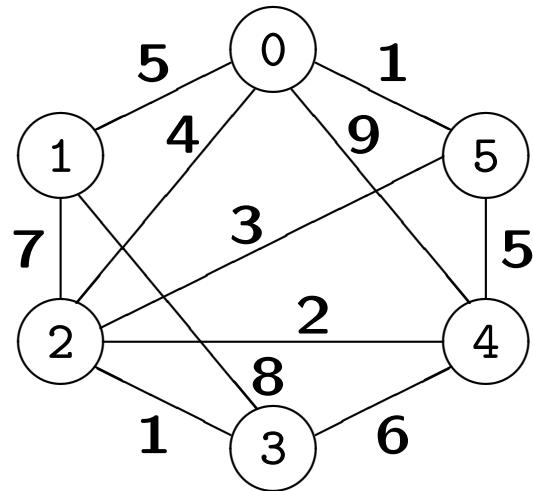


3
 $+\infty$

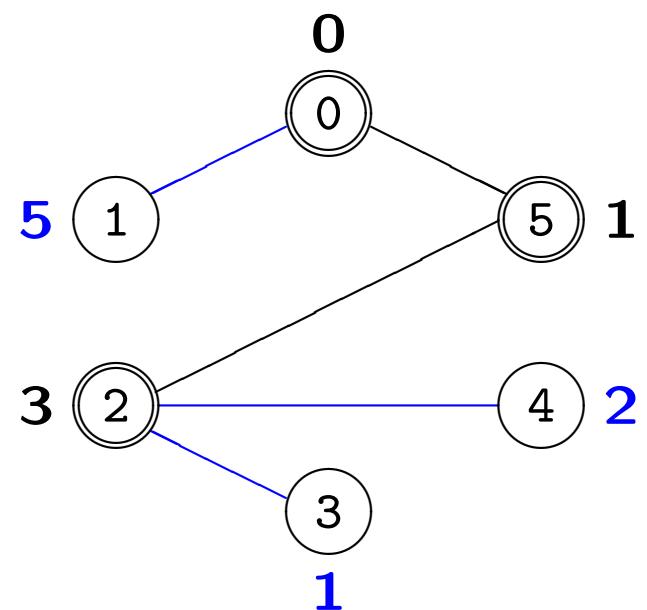
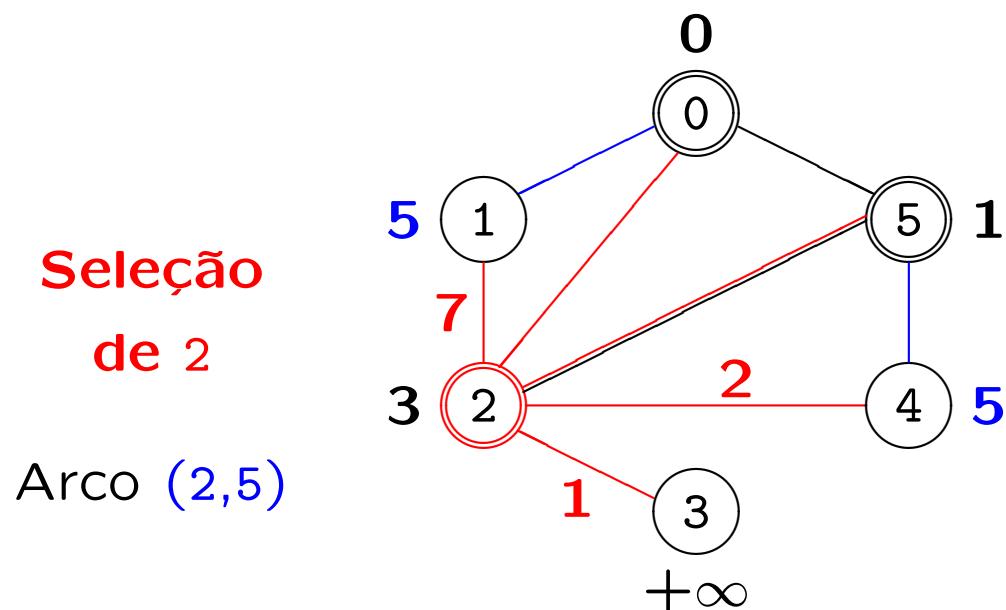
Algoritmo de Prim (2)



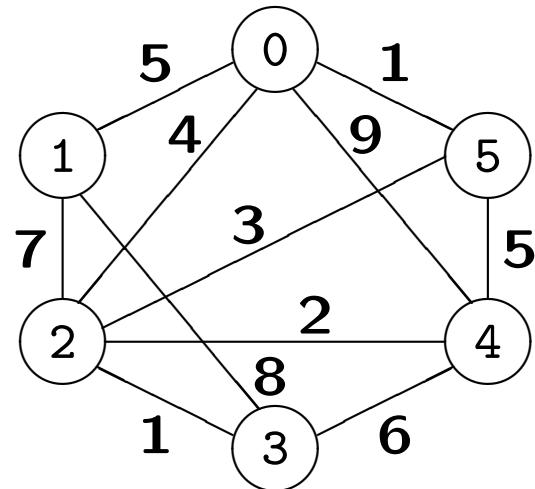
Algoritmo de Prim (3)



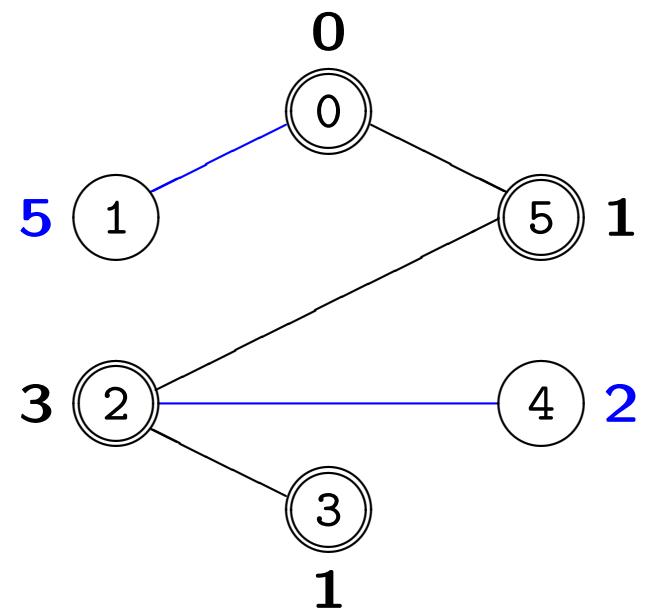
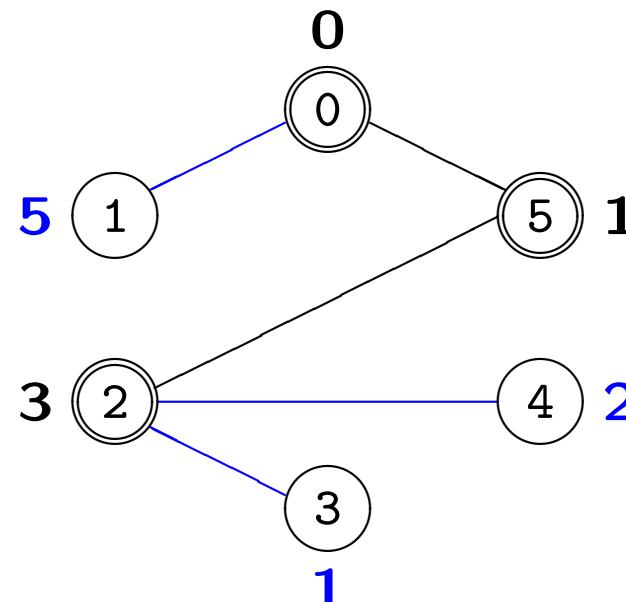
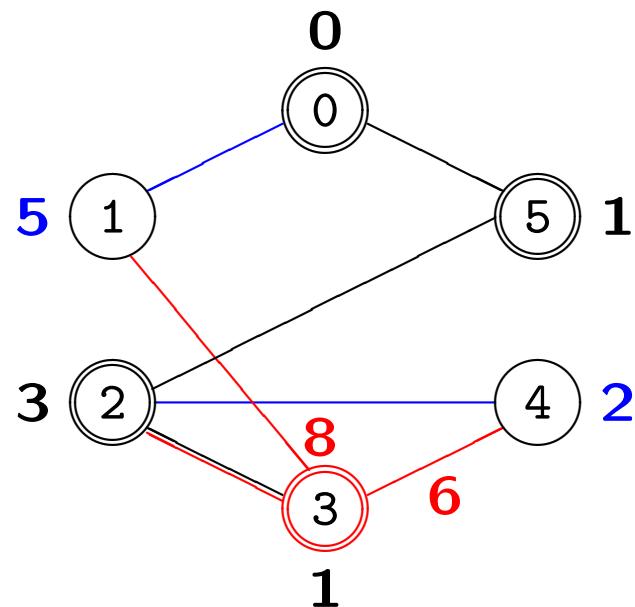
Situação Corrente



Algoritmo de Prim (4)

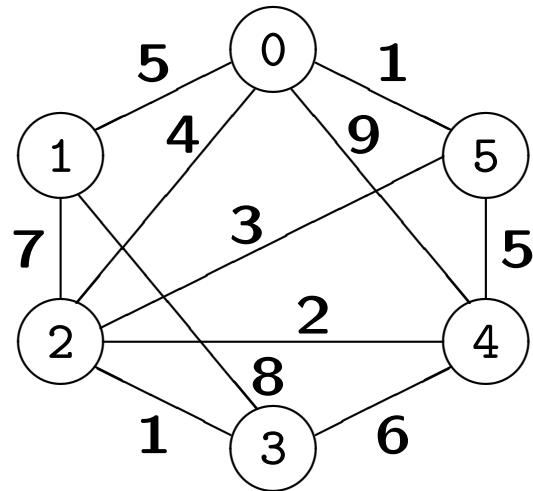


**Seleção
de 3**
Arco (2,3)

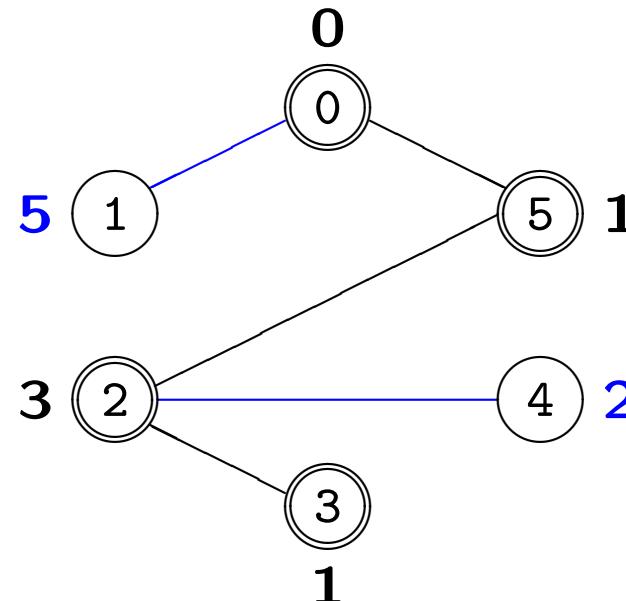
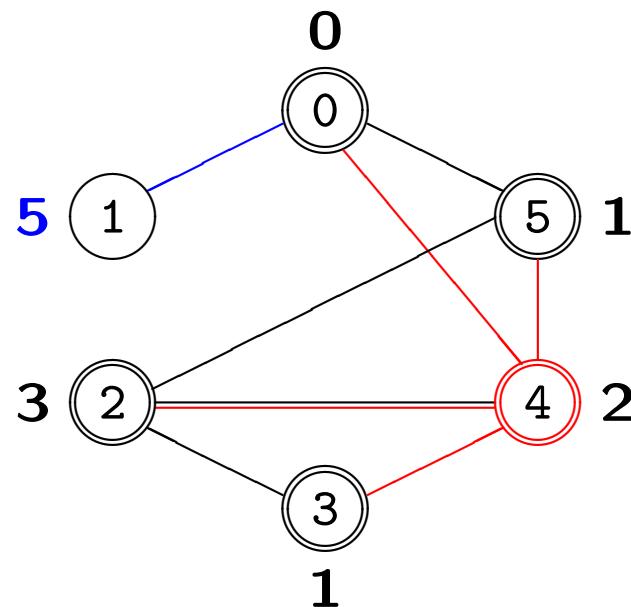


**Situação
Corrente**

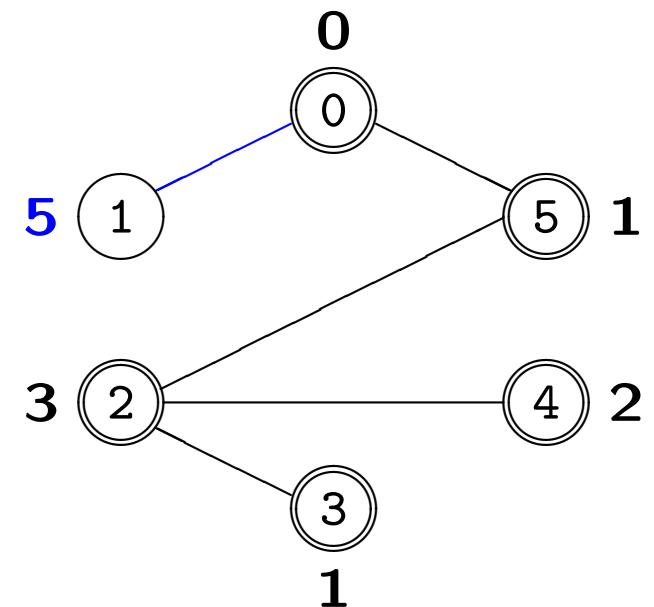
Algoritmo de Prim (5)



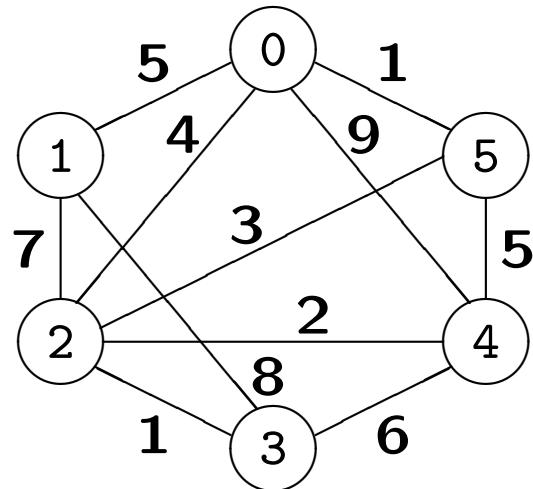
**Seleção
de 4**
Arco (2,4)



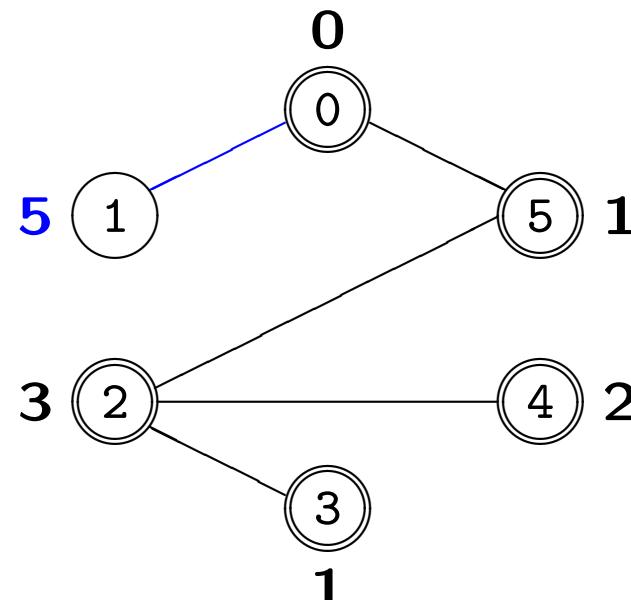
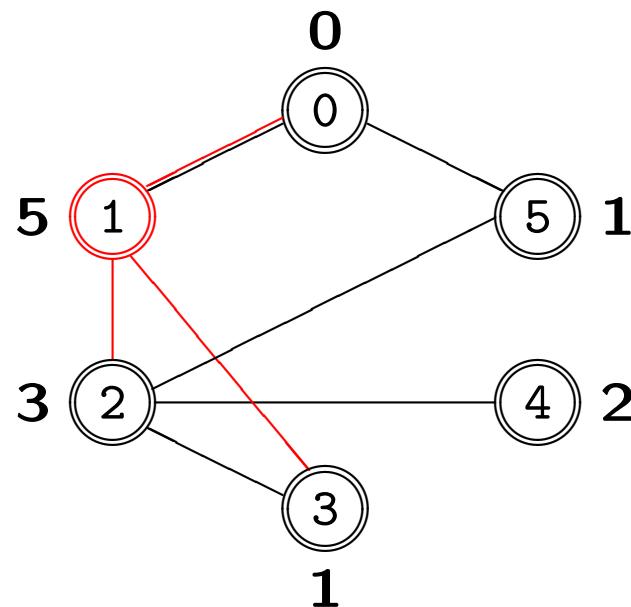
**Situação
Corrente**



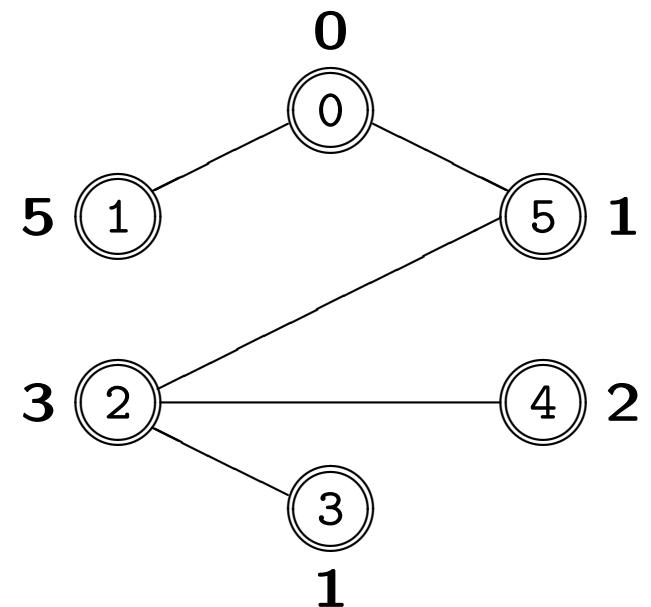
Algoritmo de Prim (6)



**Seleção
de 1**
Arco (0,1)



**Situação
Corrente**



Informação Necessária

Global: ligados

Conjunto dos vértices nunca selecionados para os quais já há caminho a partir de o .

Por cada vértice x :

- **boolean** selecionado[x]

Indica se x já foi selecionado, i.e., se já “faz parte da árvore final”.

- $\mathbb{R}_0^+ \cup \{+\infty\}$ custo[x]

– **ou** é $+\infty$, quando ainda não há caminho de o para x ;
– **ou** é o peso do arco via[x], no caso contrário.

- Edge via[x]

Se estiver definido, via[x] é um arco de peso mínimo (até ao momento) que liga x à árvore.

Situação Inicial

Global: $\text{ligados} = \{o\}$.

Informação para o vértice o :

- $\text{selecionado}[o] = \text{false}$;
- $\text{custo}[o] = 0$;
- $\text{via}[o]$ não está definido.

Informação para todos os vértices $x \in V \setminus \{o\}$:

- $\text{selecionado}[x] = \text{false}$;
- $\text{custo}[x] = +\infty$;
- $\text{via}[x]$ não está definido.

Em Cada Iteração

Seleciona-se um vértice x de ligados tal que $\text{custo}[x]$ é mínimo.

Árvore Mínima de Cobertura (1)

(Minimum Spanning Tree)

```
Edge<L>[] mstPrim( UndiGraph<L> graph ) {  
  
    Edge<L>[] mst = new Edge[ graph.numNodes() - 1 ];  
  
    int mstSize = 0;  
  
    boolean[] selected = new boolean[ graph.numNodes() ];  
  
    L[] cost = new L[ graph.numNodes() ];  
  
    Edge<L>[] via = new Edge[ graph.numNodes() ];  
  
    AdaptMinPriQueue<L, Node> connected =  
        new AdaptMinHeap<>( graph.numNodes() );
```

Árvore Mínima de Cobertura (2)

```
for every Node v in graph.nodes() {  
    selected[v] = false;  
    cost[v] = +∞;  
}  
  
Node origin = graph.aNode();  
cost[origin] = 0;  
connected.insert(0, origin);
```

Árvore Mínima de Cobertura (3)

```
do {  
    Node node = connected.removeMin().getValue();  
    selected[node] = true;  
    if ( node != origin )  
        mst[ mstSize++ ] = via[node];  
    exploreNode(graph, node, selected, cost, via, connected);  
}  
  
while ( !connected.isEmpty() );  
  
return mst;  
}
```

```

void exploreNode( UndiGraph<L> graph, Node source,
boolean[] selected, L[] cost, Edge<L>[] via,
AdaptMinPriQueue<L, Node> connected ) {
for every Edge<L> e in graph.incidentEdges(source) {
    Node node = e.oppositeNode(source);
    if ( !selected[node] && e.label() < cost[node] ) {
        boolean nodeIsInQueue = cost[node] < +∞;
        cost[node] = e.label();
        via[node] = e;
        if ( nodeIsInQueue )
            connected.decreaseKey(node, cost[node]);
        else
            connected.insert(cost[node], node);
    }
}
}

```

Complexidade do Algoritmo de Prim

Grafo em vetor de listas de “incidências”

Identificação das Operações

criação de 4 vetores	$\Theta(1)$
criação da fila com prioridade	?
inicialização de 2 vetores (selected e cost)	$\Theta(V)$
inserção da origem na fila	?
$ V $ remoção do mínimo da fila	?
$ V - 1$ inserção no vetor resultado	$\Theta(V)$
$ V $ obtenção dos arcos incidentes	$\Theta(A)$
$\leq A $ inserção ou decremento da chave na fila	?

TAD Fila com Prioridade por Mínimos (K,V)

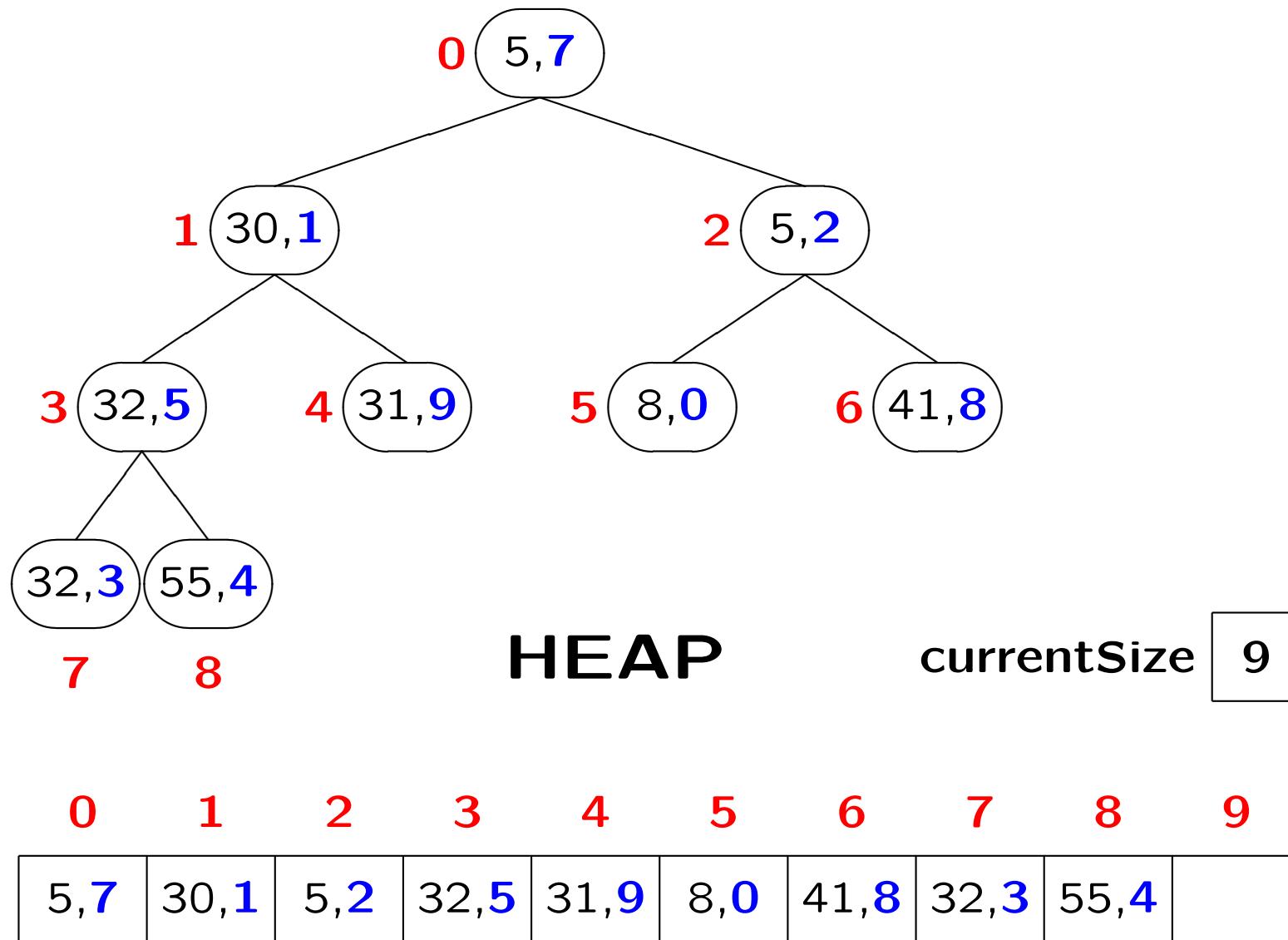
```
public interface MinPriorityQueue<
    K extends Comparable<? super K>, V>
extends Serializable {
    // Returns true iff the priority queue contains no entries.
    boolean isEmpty();
    // Returns the number of entries in the priority queue.
    int size();
    // Returns an entry with the smallest key in the priority queue.
    Entry<K,V> minEntry() throws EmptyQueueException;
    // Inserts the entry (key, value) in the priority queue.
    void insert( K key, V value );
    // Removes an entry with the smallest key from the priority queue
    // and returns that entry.
    Entry<K,V> removeMin() throws EmptyQueueException;
}
```

TAD Fila com Prioridade Adaptável por Mínimos (K,V)

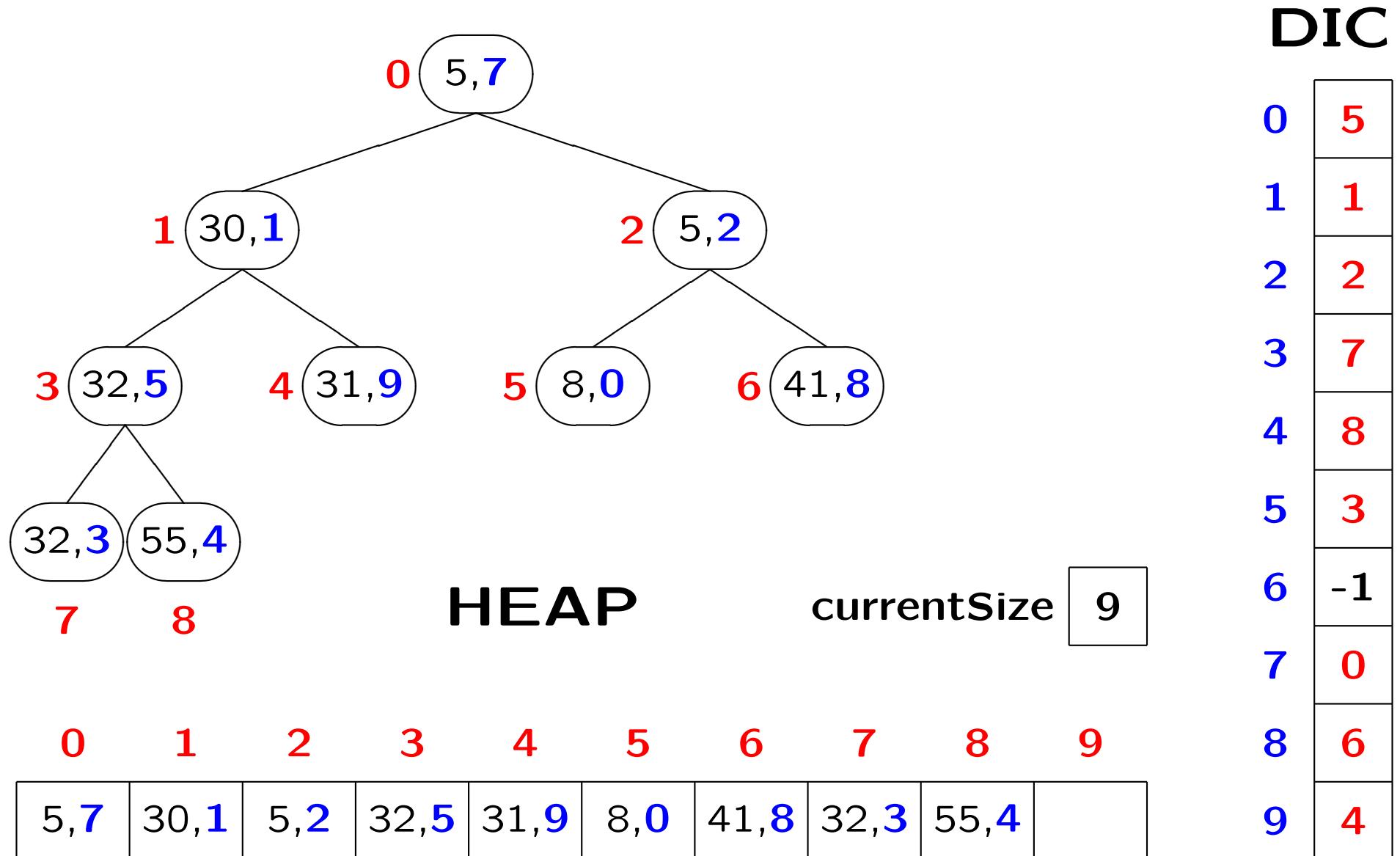
```
public interface AdaptMinPriQueue<  
    K extends Comparable<? super K>, V>  
extends MinPriorityQueue<K,V> {  
  
    // If the priority queue contains an entry with the specified value,  
    // returns the associated key and replaces it by the specified key  
    // (which is less than the old one). Otherwise, returns null.  
    K decreaseKey( V value, K newKey ) throws InvalidKeyException;  
}
```

Nota: Por questões de eficiência, é usual exigir-se que os valores sejam todos distintos. Nesses casos, o método **insert** levanta uma exceção quando se tenta inserir uma entrada cujo valor já existe na fila.

Implementação com Heap (K, V)



Implementação com Heap e Vetor (K,V)



Descrição das Operações com Sucesso (1)

- **void insert(K key, V value) throws InvalidValueException**

Dicionário: Pesquisa-se o valor, para descobrir se já existe.

Heap: Coloca-se a nova entrada no fim do heap e executa-se o borbulhar ascendente a partir dessa posição (a última ocupada).

- **Entry<K,V> removeMin() throws EmptyQueueException**

Heap: Retorna-se a primeira entrada do heap (posição zero). Coloca-se a última entrada no início do heap e executa-se o borbulhar descendente a partir da posição zero.

Altera-se o dicionário sempre que uma entrada é inserida, é removida ou muda de posição no heap.

Descrição das Operações com Sucesso (2)

- K **decreaseKey**(V value, K newKey) **throws** InvalidKeyExcept.

Dicionário: Pesquisa-se o valor, para descobrir a posição da entrada no heap.

Heap: Executa-se o borbulhar ascendente a partir dessa posição.

Altera-se o dicionário sempre que uma entrada muda de posição no heap.

Complexidades da Fila com Prioridade Adaptável com **Heap** e **Vetor** (*n* entradas)

	Melhor Caso	Pior Caso	Caso Esperado
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
minEntry	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(1)$	$O(\log n)$	$O(\log n)$
removeMin	$\Theta(1)$	$O(\log n)$	$O(\log n)$
decreaseKey	$\Theta(1)$	$O(\log n)$	$O(\log n)$

Complexidades da Fila com Prioridade Adaptável com **Heap** e **Tabela de Dispersão** (n entradas)

	Melhor Caso	Pior Caso	Caso Esperado
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
minEntry	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(1)$	$O(n \log n)$	$O(\log n)$
removeMin	$\Theta(1)$	$O(n \log n)$	$O(\log n)$
decreaseKey	$\Theta(1)$	$O(n \log n)$	$O(\log n)$

Complexidade do Algoritmo de Prim

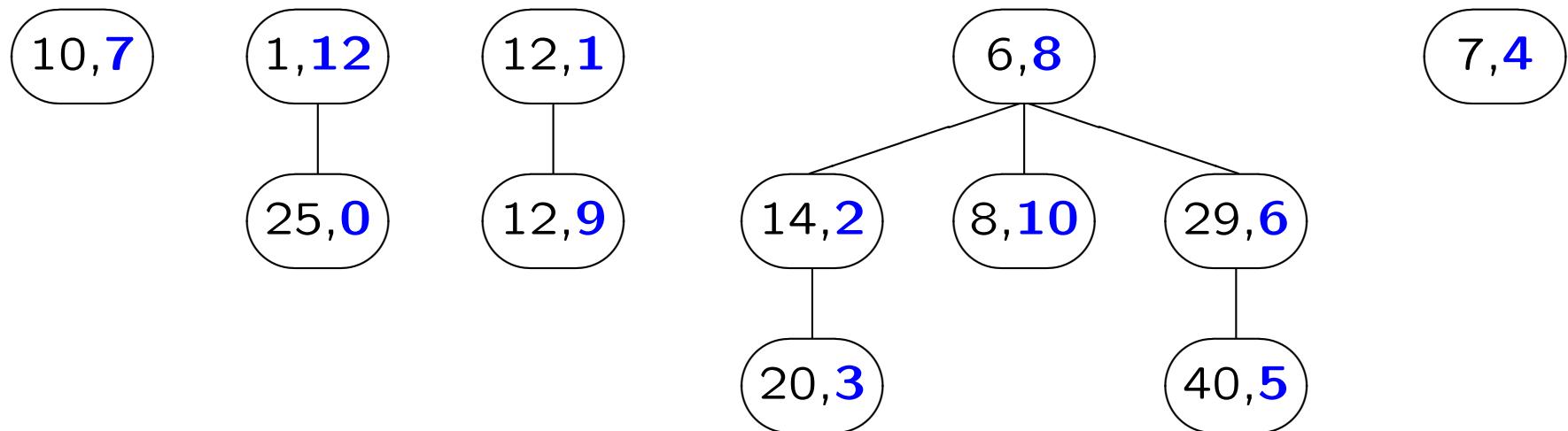
Grafo em vetor de listas de “incidências”

Fila implementada com Heap e Vetor

criação de 4 vetores	$\Theta(1)$
criação da fila com prioridade	$\Theta(V)$
inicialização de 2 vetores (selected e cost)	$\Theta(V)$
inserção da origem na fila	$\Theta(1)$
$ V $ remoção do mínimo da fila	$O(V \times \log V)$
$ V - 1$ inserção no vetor resultado	$\Theta(V)$
$ V $ obtenção dos arcos incidentes	$\Theta(A)$
$\leq A $ inserção ou decremento da chave na fila	$O(A \times \log V)$
TOTAL	$O(A \times \log V)$

Fila de Fibonacci [Fredman e Tarjan 87]

Uma **fila de Fibonacci** (K, \mathbf{V}) é uma floresta de árvores com prioridade.
(Qualquer nó tem uma chave inferior ou igual às chaves que se encontram nas suas sub-árvores.)



Tal como no Heap, é conveniente ter um dicionário que associe a cada valor o nó que tem esse valor.

Complexidades da Fila com Prioridade Adaptável com **Heap/Fila de Fibonacci** e **Vetor** (n entradas)

	Heap (Pior Caso)	Fila de Fibonacci (Pior Caso)	Fila de Fibonacci Amortizada
isEmpty	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
size	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
minEntry	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$O(\log n)$	$\Theta(1)$	$\Theta(1)$
removeMin	$O(\log n)$	$O(n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(n)$	$\Theta(1)$

Complexidade do Algoritmo de Prim

Grafo em vetor de listas de “incidências”

Fila implem. com Fila de Fibonacci e Vetor

criação de 4 vetores	$\Theta(1)$
criação da fila com prioridade	$\Theta(V)$
inicialização de 2 vetores (selected e cost)	$\Theta(V)$
inserção da origem na fila	$\Theta(1)$
$ V $ remoção do mínimo da fila	$O(V \times \log V)$
$ V - 1$ inserção no vetor resultado	$\Theta(V)$
$ V $ obtenção dos arcos incidentes	$\Theta(A)$
$\leq A $ inserção ou decremento da chave na fila	$O(A \times 1)$
TOTAL	$O(A + V \times \log V)$

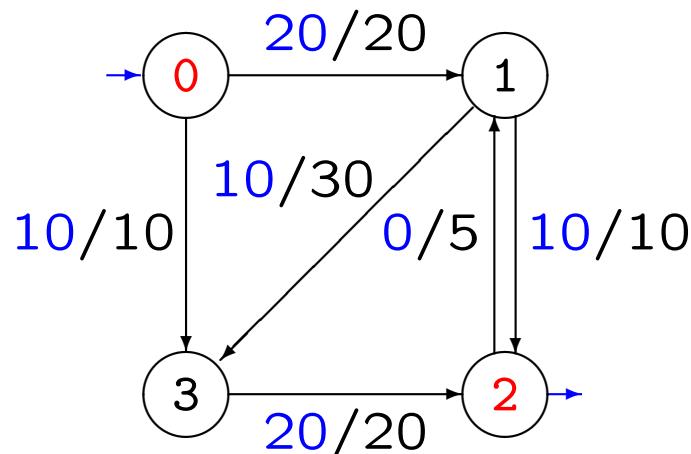
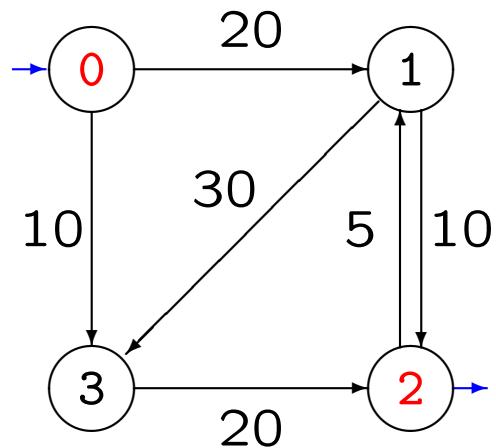
Capítulo X

Fluxo Máximo entre Dois Vértices (num grafo orientado e pesado)

Algoritmo de Edmonds-Karp

Problema

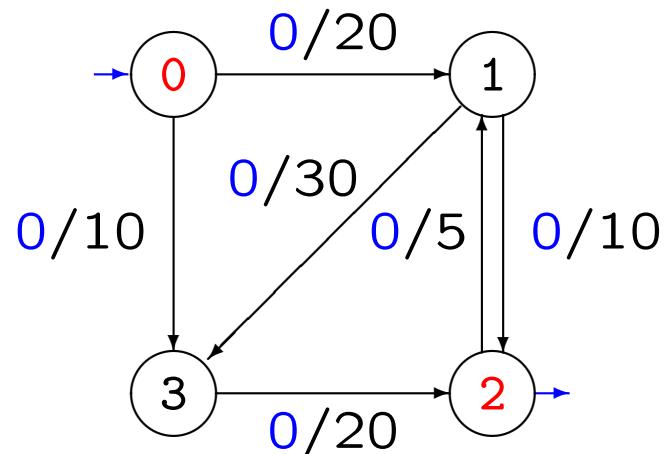
Dado um grafo **orientado** e **pesado** (com pesos não negativos) e dois vértices f e d , como encontrar um **fluxo máximo de f para d** ?



Valor de um Fluxo Máximo de 0 para 2: 30

Assume-se que qualquer vértice pertence a um caminho de f para d .
Portanto, o grafo é **simplesmente conexo** e $|A| \geq |V| - 1$.

Ideia Geral (de 0 para 2)



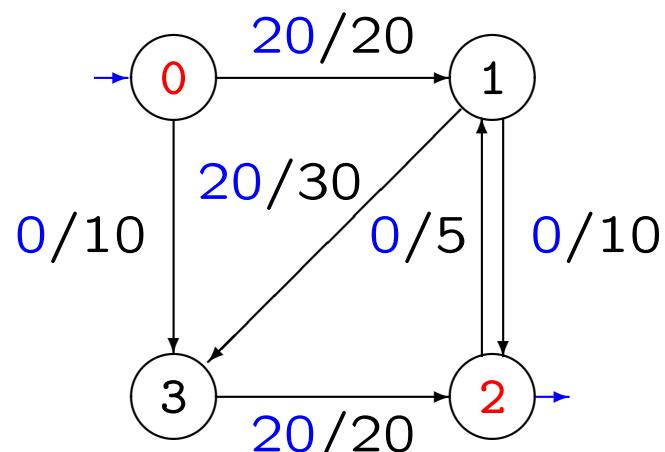
Descobrir um caminho para drenar:

0, 1, 3, 2.

Incremento máximo permitido:

20.

Atualizar o fluxo.



PROBLEMA

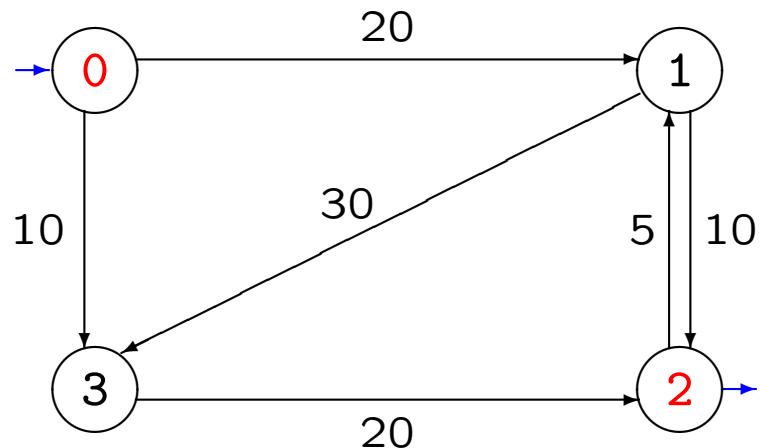
Já não há caminho para drenar e o valor de um fluxo máximo é > 20.

SOLUÇÃO

“Cancelar 10 unidades de fluxo do arco (1, 3), desviando-as para (1, 2)”

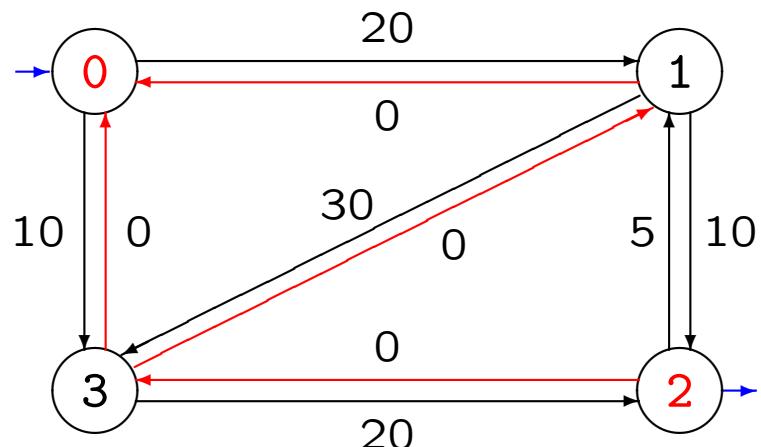
Método de Ford-Fulkerson [1962]

Grafo Original

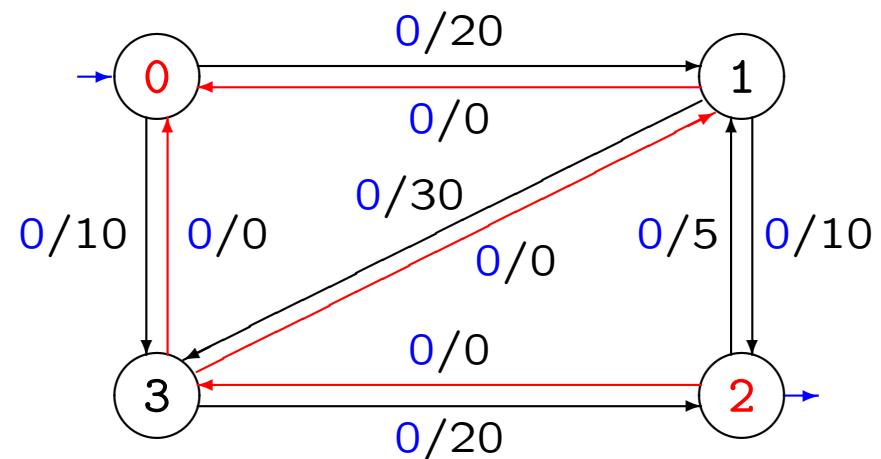


Trabalhar na Rede de Fluxos
(todos os arcos têm inverso)

Rede de Fluxos

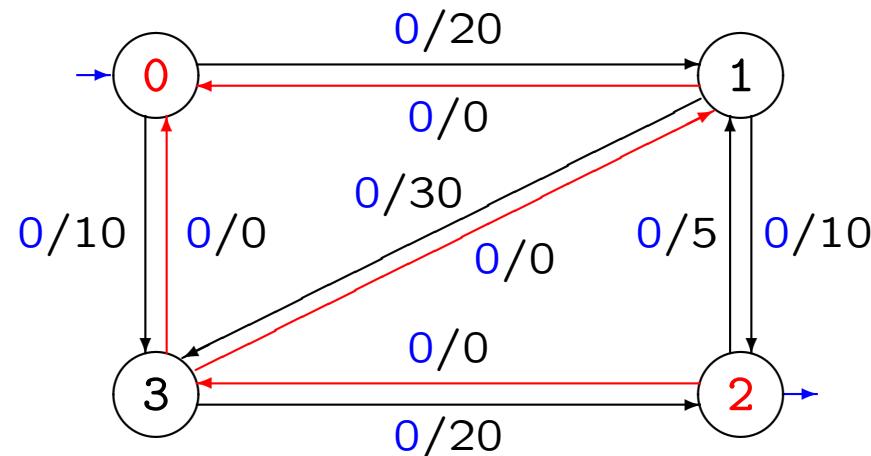


Fluxo Inicial

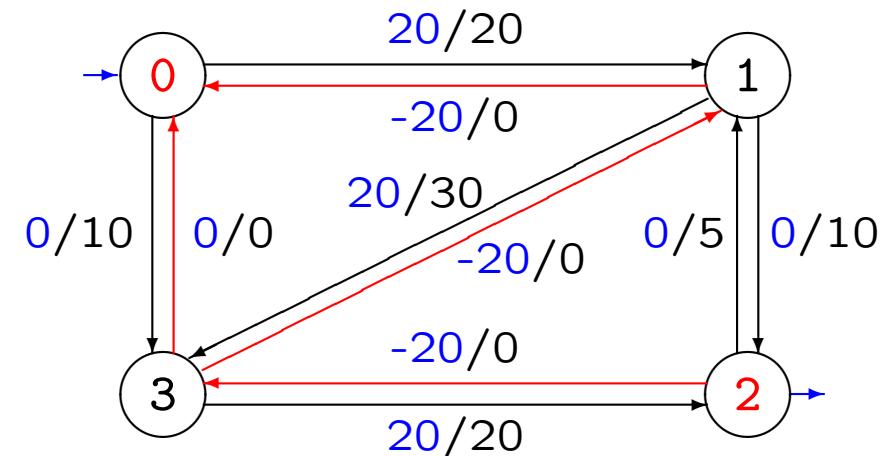


Método de Ford-Fulkerson (1)

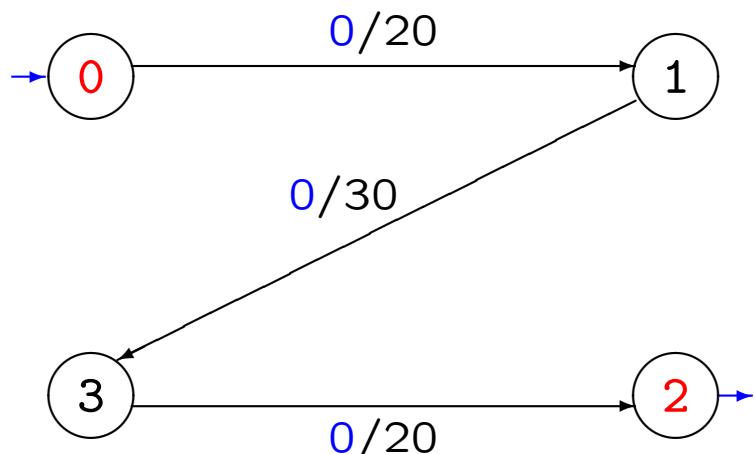
Estado Inicial



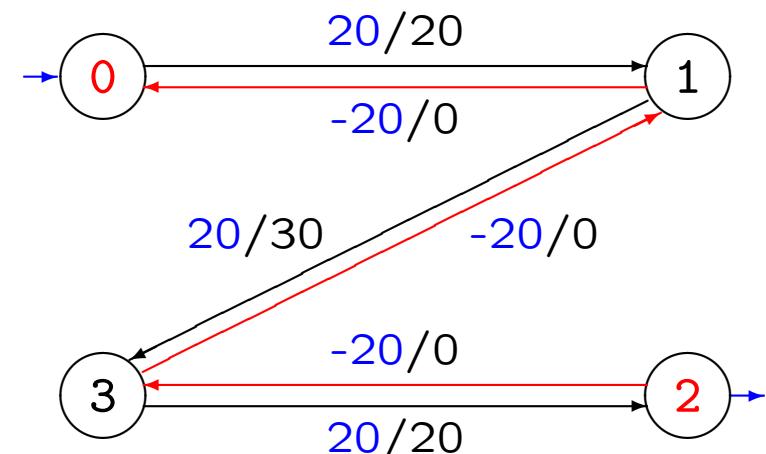
Estado Seguinte



Caminho: 0, 1, 3, 2

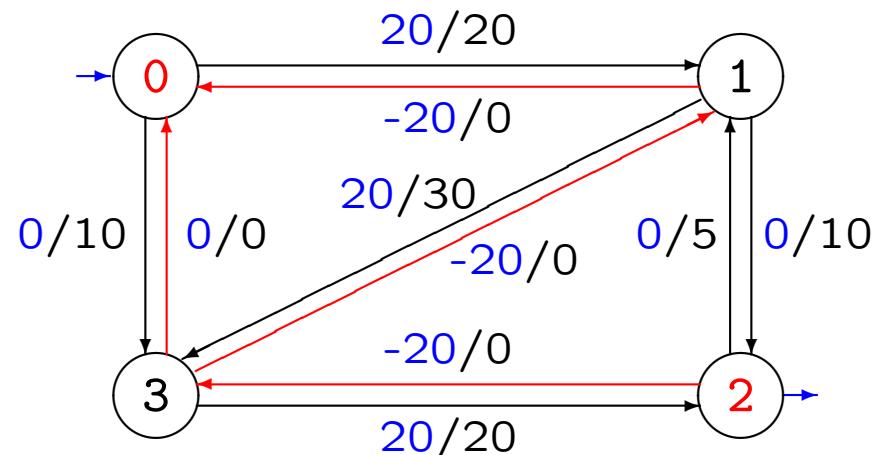


Incremento: 20

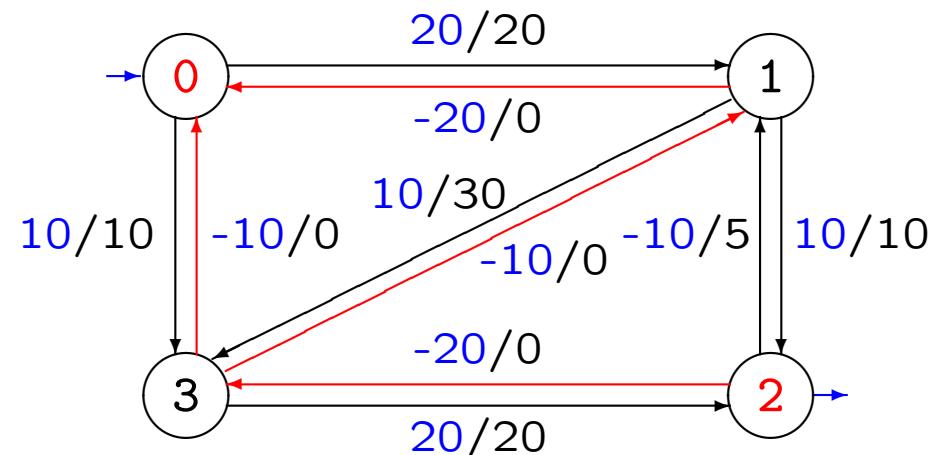


Método de Ford-Fulkerson (2)

Estado Corrente

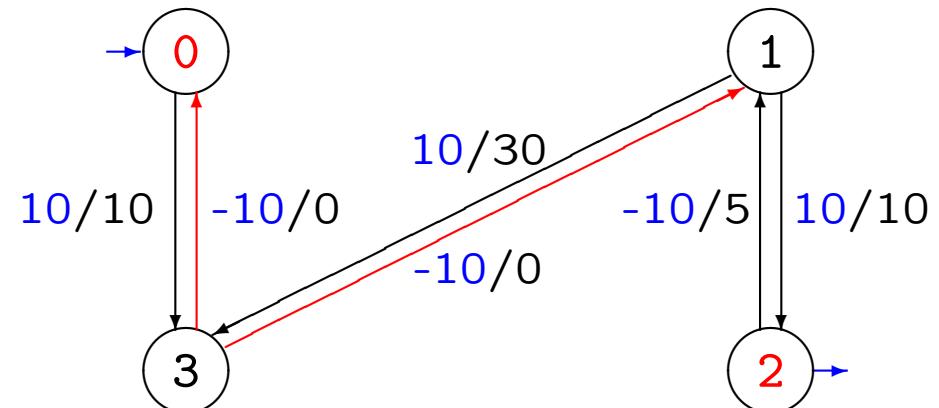
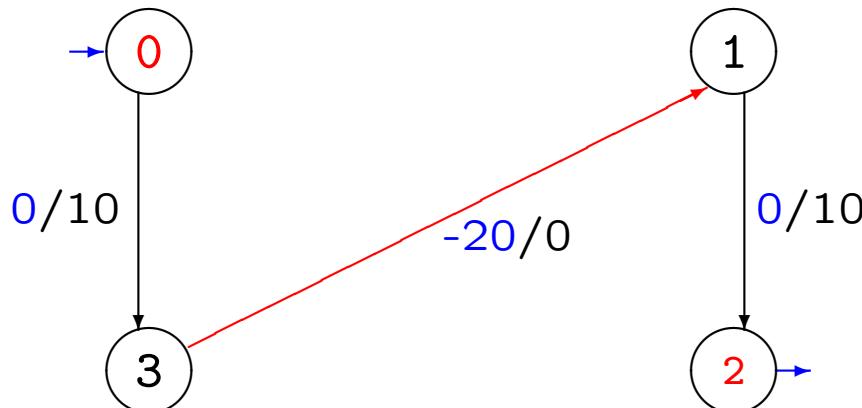


Estado Seguinte



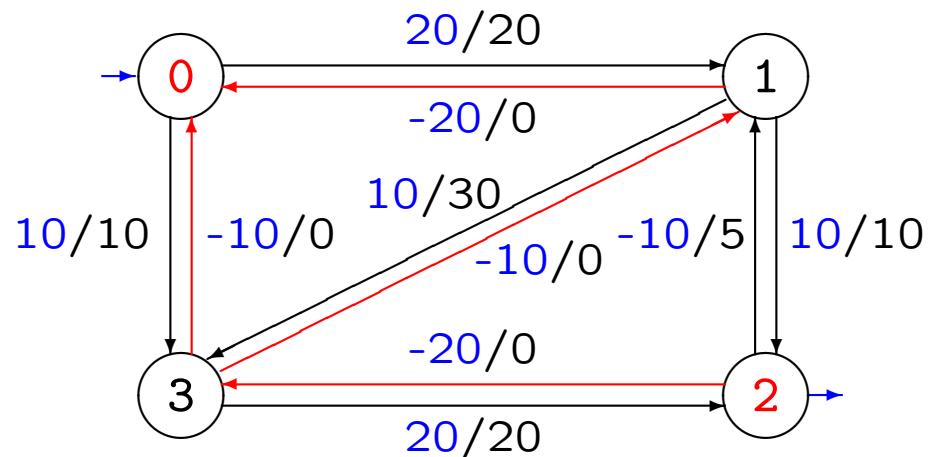
Caminho: 0, 3, 1, 2

Incremento: 10

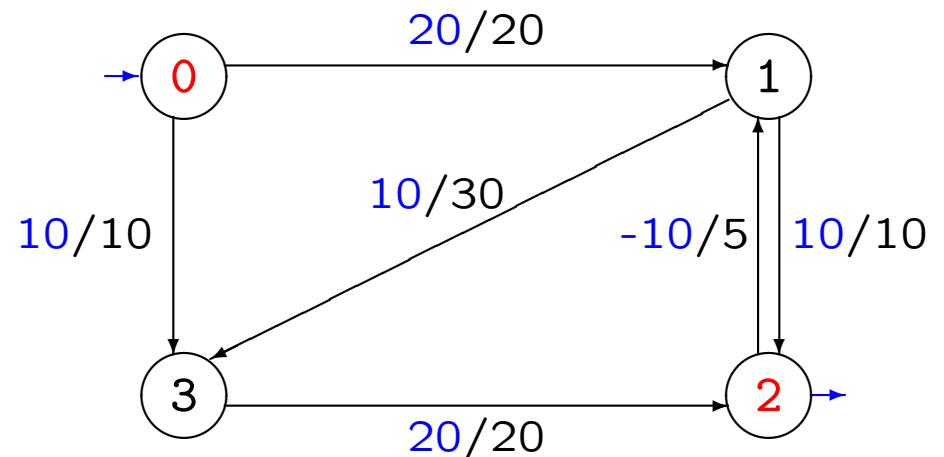


Método de Ford-Fulkerson (3)

Estado Corrente



Resultado



Como não há caminho de 0 para 2 na **rede residual** (i.e. quando se ignoram os arcos com um fluxo igual à capacidade), o fluxo é máximo (e o seu valor é 30).

Rede de Fluxos (*Flow Network*)

Seja $G = (V, A)$ um grafo orientado e pesado, cujos arcos têm um peso não negativo (que representa a **capacidade** do arco). A **rede de fluxos** de G é o grafo $R = (V, A')$, orientado e pesado, tal que:

- $A \subseteq A'$ (a rede tem todos os arcos de G);
- Se $(v, w) \in A$ e $(w, v) \notin A$, a rede tem o arco (w, v) com peso zero.

