Introdução aos Paradigmas de Projeto de Algoritmos

Rafael Lima
rafa@uft.edu.br
http://sites.google.com/site/rafaeluft

Dividir e conquistar

Paradigma dividir e conquistar

1 – Divide

2 – Conquer

3 - Combine

Paradigma dividir e conquistar

1 – Divide

- Divide o problema de p>= 1 partes, cada parte com tamanhos estritamente menores que n
- p= 2 é um valor comumente utilizado
- Se p=1, geralmente uma parte do problema é desprezada (caso da busca binária, p. ex.)
- p pode ser tão grande quanto logn como pode ser moderado como n^e, para $0 < e < 1$.

Paradigma dividir e conquistar

1 – Conquer

- Executa *p* chamadas recursivas, caso n > limiar de simplicidade
    - Ao n chegar em um tamanho relativamente pequeno um algoritmo iterativo pode entrar em ação
    - Geralmente encontrado empiricamente (requer uma análise do algoritmo)
    - Limiar grande → compromete o comportamento assintótico.

Paradigma dividir e conquistar

- Combine
  - As soluções das p chamadas recursivas devem ser combinadas
  - Merging, searching, finding the maximum or minimum, matrix addition, etc
  - É o passo crucial para definição de virtualmente todos os algoritmos deste paradigma.

Paradigma dividir e conquistar

Meta algoritmo:

(1) Se o I é simples, resolva com algum método direto, caso contrário vá para o próximo passo;

Base

(2) Divida a instância *I* em p partes I_1, I_2, ..., I_p de tamanhos aproximadamente iguais

(3) Recursivamente chame o método em cada I_j para obter p soluções parciais

Divisão

(4) Combine os resultados das p soluções para formar a solução da instância original *I*. Retorne a solução de *I*.

Conquista

D&C: Busca de min e max em um vetor

Considere o seguinte algoritmo:

$x \leftarrow A[1]; \quad y \leftarrow A[1]$

for $i \leftarrow 2$ **to** n

$\quad$ **if** $A[i] < x$ **then** $x \leftarrow A[i]$

$\quad$ **if** $A[i] > y$ **then** $y \leftarrow A[i]$

end for

return (x, y)

Podemos fazer melhor?

Claramente, o algoritmo gasta 2n-2 comparações.

D&C: Busca de min e max em um vetor

Aplicando a estratégia de divisão e conquista:

Ideia:

- divida o vetor de entrada em duas metades

$A[1...n/2] \quad A[n/2+1,...,n]$

- ache o maior e menor elementos de cada uma das metades
- retorne o mínimo de dois mínimos e o máximo de dois máximos

D&C: Busca de min e max em um vetor

Procedure $minmax(low, high)$

if $high - low = 1$ **then**
    if $A[low] < A[high]$ **then return** $(A[low], A[high])$
    else return $(A[high], A[low])$
    end if
else
    $mid \leftarrow \lfloor (low + high)/2 \rfloor$
    $(x_1, y_1) \leftarrow minmax(low, mid)$
    $(x_2, y_2) \leftarrow minmax(mid + 1, high)$
    $x \leftarrow \min\{x_1, x_2\}$
    $y \leftarrow \max\{y_1, y_2\}$
    return (x, y)
end if

E agora? Quantas comparações?

D&C: Busca de min e max em um vetor

$$C(n) = \begin{cases} 1 & \text{if } n = 2 \\ 2C(n/2) + 2 & \text{if } n > 2. \end{cases}$$

$$\begin{aligned} C(n) &= 2C(n/2) + 2 \\ &= 2(2C(n/4) + 2) + 2 \\ &= 4C(n/4) + 4 + 2 \\ &= 4(2C(n/8) + 2) + 4 + 2 \\ &= 8C(n/8) + 8 + 4 + 2 \\ &\vdots \\ &= (3n/2) - 2. \end{aligned}$$

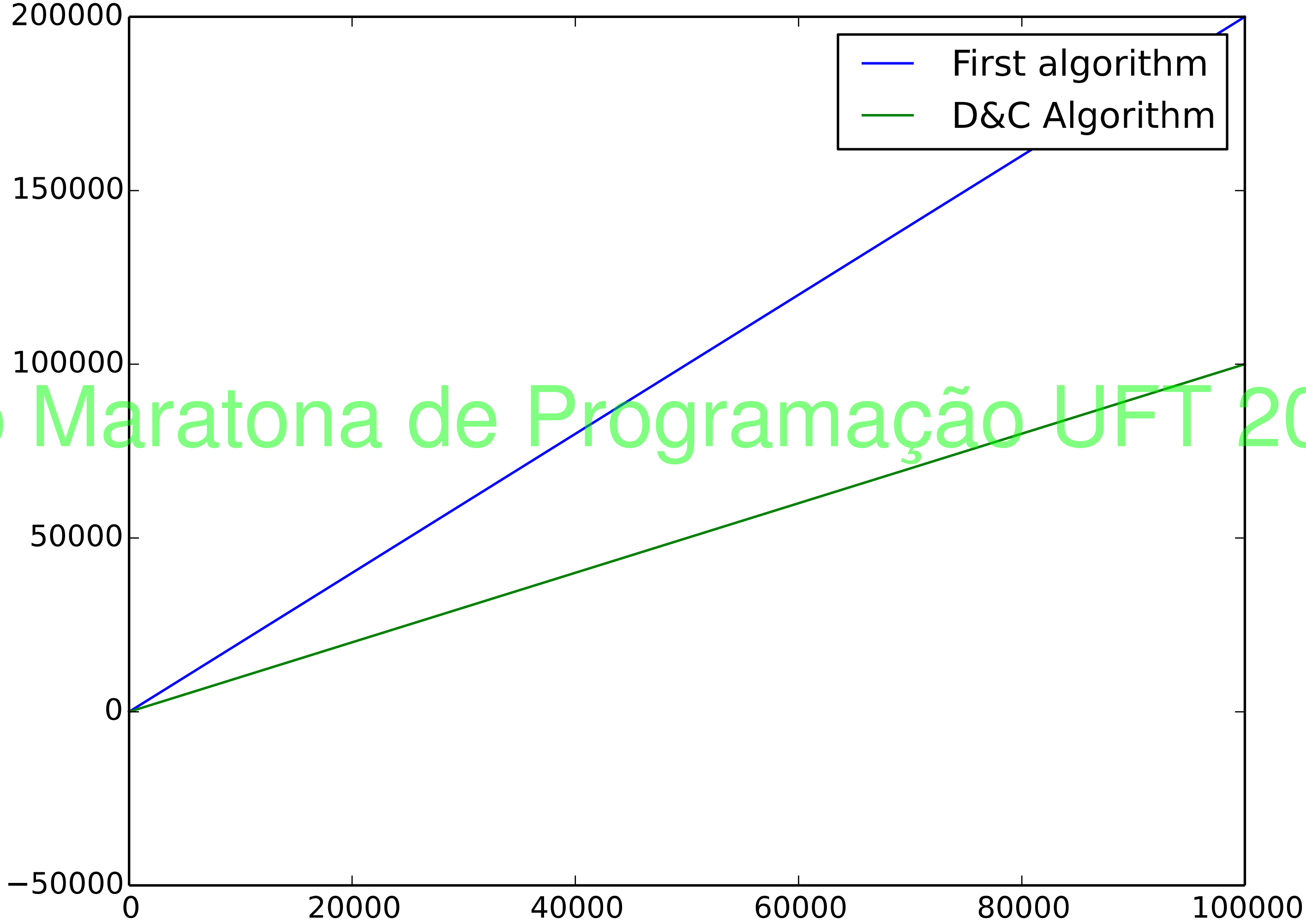
Mas, também não é O(n)?

D&C: Busca de min e max em um vetor

Mas, também

não é O(n)?

De olho nas constantes!



Programação Dinâmica

Discussão inicial

- Fibonacci: $$f(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ f(n-1) + f(n-2) & \text{if } n \geq 3. \end{cases}$$

- Implementação recursiva:

1. **procedure** $f(n)$
2. **if** $(n = 1)$ **or** $(n = 2)$ **then return** 1
3. **else return** $f(n-1) + f(n-2)$

- Custo: $$T(n) = \begin{cases} 1 & \text{if } n = 1 \text{ or } n = 2 \\ T(n-1) + T(n-2) & \text{if } n \geq 3. \end{cases}$$
- Solução é a própria função: $T(n) = f(n)$.

Discussão Inicial

- Sabe-se que: $f(n) \approx \Theta(\phi^n)$ $\phi = (1 + \sqrt{5})/2 \approx 1.61803$
- Conclusão: o tempo de execução para esta função é exponencial em n.

- Solução óbvia: iniciar com f_1, e f_2, atualizando a janela com os dois últimos valores para obter o terceiro.
  - Custo: $\Theta(n)$
  - Muito melhor que o exponencial!

The longest common subsequence Problem

Sejam A e B duas strings de tamanho n e m, respectivamente, definidas no alfabeto ∑, determine o ***tamanho da subsequência mais longa*** *que é comum a ambas A e B.*

Uma subsequência de $A = a_1 a_2, \ldots a_n$ *é uma string na forma* $a_{i_1} a_{i_2}, \ldots a_{i_k}$ *tal que* $1 \leq i_1 < i_2 < \ldots < i_k \leq n$

Exemplo: $\Sigma = \{x, y, z\}$ *A = zxyxyz e B = xyyzx*
Então: xyy é uma subsequência de tamanho 3 tanto de A quanto de B, mas não é a mais longa.

Encontrem uma sequência mais longa!

Que tal xyyz?

The longest common subsequence prob

- Qual seria a forma natural (brute force) de lidar com esse problema?
- Uma ideia (not so clever):

  - Enumere todas as 2^n subsequências de A
  - Para cada subsequência determine se também é uma subsequência de B em $\Theta(m)$
  - Total: $\Theta(m2^n)$
    - Exponencial

The longest common subsequence prob

- Seja $A = a_1a_2, \ldots a_n$ e $B = b_1b_2, \ldots b_m$ duas strings

- Seja *L[i,j]* o tamanho da maior subsequência mais longa de $a_1a_2, \ldots a_i$ e $b_1b_2, \ldots b_j$

- L[i,j] = 0 caso uma das duas strings tenha tamanho zero.

The longest common subsequence prob

- Suponha i> 0 e j>0:
  - If $a_i = b_j$, $L[i,j] = L[i-1, j-1] + 1$.
  - If $a_i \neq b_j$, $L[i,j] = \max\{L[i, j-1], L[i-1, j]\}$.

- Equação de recorrência:

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } i > 0, j > 0 \text{ and } a_i = b_j \\ \max\{L[i, j-1], L[i-1, j]\} & \text{if } i > 0, j > 0 \text{ and } a_i \neq b_j \end{cases}$$

The longest common subsequence prob

- Algoritmo
  - Usamos uma tabela (n+1)x(m+1) para computar os valores de L[i,j]
  - Cada entrada gasta O(1) para ser calculada
  - Logo: $\Theta(nm)$

- If $a_i = b_j$, $L[i,j] = L[i-1, j-1] + 1$.
- If $a_i \neq b_j$, $L[i,j] = \max\{L[i, j-1], L[i-1, j]\}$.

Elabore o algoritmo que preenche a tabela acima.

The longest common subsequence prob

Execute o algoritmo para as seguintes strings:
A =“xyxxzxyzxy” e B =“zxzyyzxxyxxz”.

The longest common subsequence prob

			z	x	z	y	y	z	x	x	y	x	x	z
		0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x	1	0	0	**1**	1	1	1	1	1	1	1	1	1	1
y	2	0	0	1	1	2	**2**	2	2	2	2	2	2	2
x	3	0	0	1	1	2	2	2	**3**	3	3	3	3	3
x	4	0	0	1	1	2	2	2	3	4	4	**4**	4	4
z	5	0	1	1	2	2	2	3	3	4	4	4	4	5
x	6	0	1	2	2	2	2	3	4	4	4	5	**5**	5
y	7	0	1	2	2	3	3	3	4	4	5	5	5	5
z	8	0	1	2	3	3	3	4	4	4	5	5	5	**6**
x	9	0	1	2	3	3	3	4	5	5	5	6	6	6
y	10	0	1	2	3	4	4	4	5	5	6	6	6	6

Programação dinâmica

- **Ideia principal**: salvar resultados para subproblemas de forma a evitar computá-los novamente.

- O algoritmo calcula uma solução ótima para cada subinstância da instância original do problema.
  - Todas as entradas da tabela representam soluções ótimas para subinstâncias consideradas pelo algoritmo.

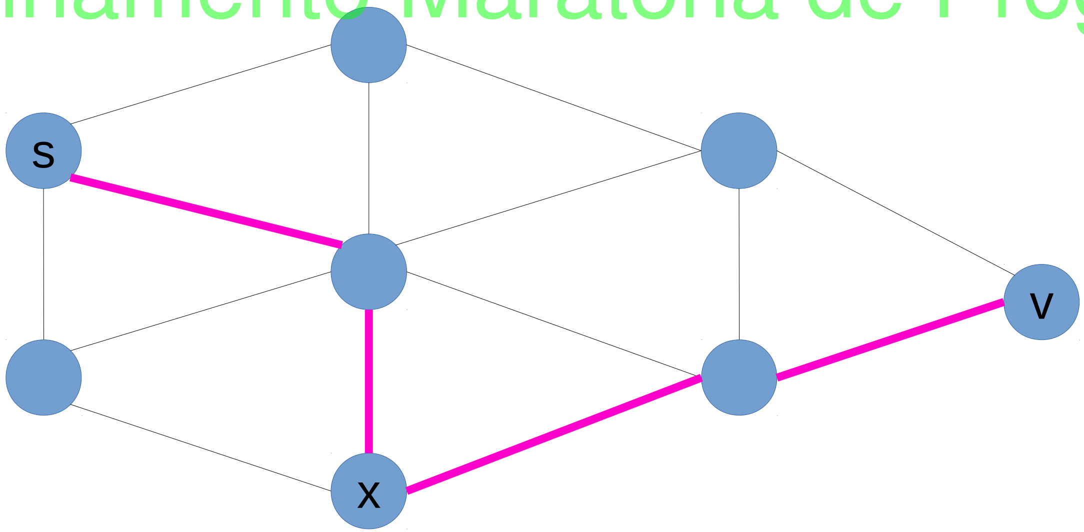
Programação Dinâmica

			z	x	z	y	y	z	x	x	y	x	x	z
		0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0	0	0	0	0	0	0	0	0	0	0	0
x	1	0	0	**1**	1	1	1	1	1	1	1	1	1	1
y	2	0	0	1	1	2	**2**	2	2	2	2	2	2	2
x	3	0	0	1	1	2	2	2	**3**	3	3	3	3	3
x	4	0	0	1	1	2	2	2	3	4	4	**4**	4	4
z	5	0	1	1	2	2	2	3	3	4	4	4	4	5
x	6	0	1	2	2	2	2	3	4	4	4	5	**5**	5
y	7	0	1	2	2	3	3	3	4	4	5	5	5	5
z	8	0	1	2	3	3	3	4	4	4	5	5	5	**6**
x	9	0	1	2	3	3	3	4	5	5	5	6	6	6
y	10	0	1	2	3	4	4	4	5	5	6	6	6	6

Representa a solução ótima se considerarmos a subinstância com as 3 primeiras letras!

Princípio de Otimalidade

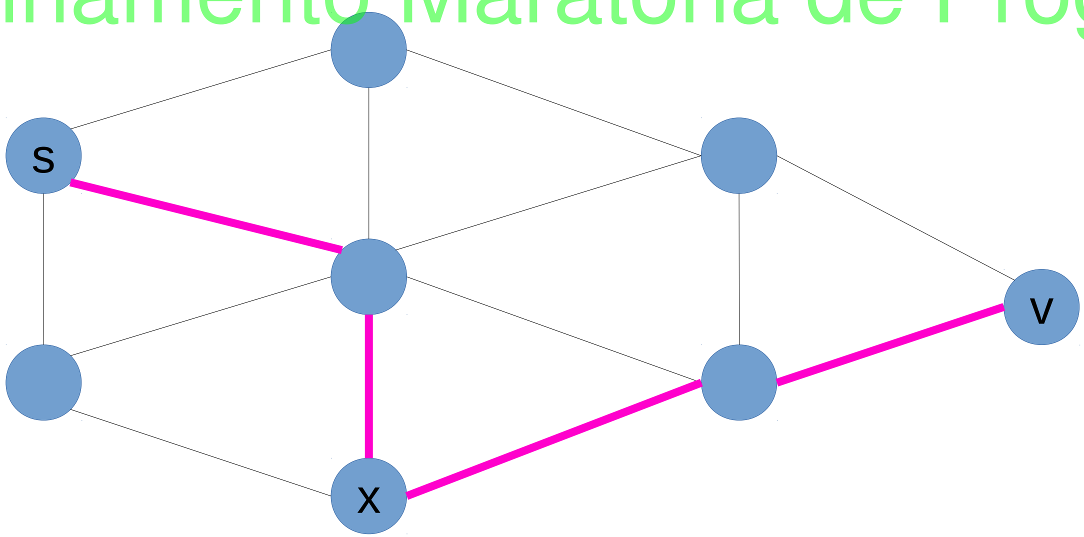
Dada uma sequência ótima de decisões, cada uma de suas subsequências deve também ser uma sequência ótima.



Se o caminho mais curto entre s e v incluir o vértice x, então o caminho mais curto entre s e x deve pertencer ao caminho mais curto entre s e v.
PD pode ser aplicada.

Princípio de Otimalidade

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.

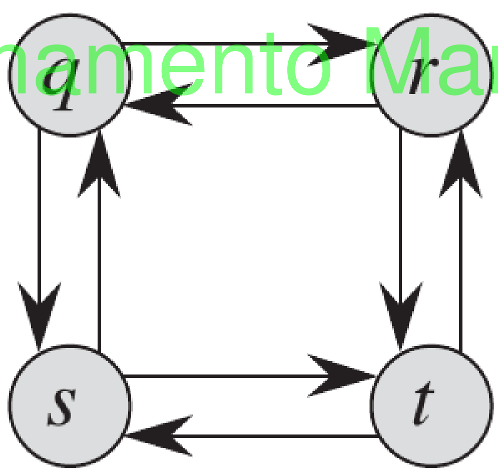


Simple paths: sem repetição de vértices.

Este problema, apresenta o princípio de otimalidade?

Princípio de Otimalidade

Unweighted longest simple path: Find a simple path from u to v consisting of the most edges. We need to include the requirement of simplicity because otherwise we can traverse a cycle as many times as we like to create paths with an arbitrarily large number of edges.



Simple paths sem repetição de vértices.

Este problema, apresenta o princípio de otimalidade?

Não.

q→ r→ t é um caminho mais longo de q para t

Porém de q para r: temos o caminho: q→ s→ t→ r

Na verdade é um problema NP-Completo.

The Knapsack Problem

Seja $U = \{u_1, u_2, \ldots, u_n\}$ um conjunto com n itens que devem ser acomodados em uma mochila de tamanho C.

Para $1 \leq j \leq n$ seja s_j e v_j o tamanho e o valor do j-ésimo item, respectivamente, onde C, s_j e v_j são todos inteiros positivos.

Objetivo: Encontrar um subconjunto $S \subseteq U$ tal que maximize a soma dos valores dos objetos respeitando-se a quantidade C.

$$\max \sum_{u_i \in S} v_i$$

s.t:

$$\sum_{u_i \in S} s_i \leq C$$

The Knapsack Problem

- V[i,j] → preenche uma mochila de tamanho j com os primeiros i itens $\{u_1, u_2, \ldots, u_i\}$ de maneira ótima
  - i vai de 0 até n e j de 0 até C
  - Nos interessa o valor de V[n,C]
  - V[0,j] = 0 → não há itens na mochila
  - V[i,0] = 0 → não cabe nada em uma mochila de tamanho zero.

The Knapsack Problem

- V[i,j] → é o máximo entre:
  - $V[i-1,j]$: The maximum value obtained by filling a knapsack of size j with items taken from $\{u_1, u_2, \ldots, u_{i-1}\}$ only *in an optimal way*.
  - $V[i-1,j-s_i]+v_i$: The maximum value obtained by filling a knapsack of size $j-s_i$ with items taken from $\{u_1, u_2, \ldots, u_{i-1}\}$ *in an optimal way* plus the value of item u_i. This case applies only if $j \geq s_i$ and it amounts to adding item u_i to the knapsack.

- Recorrência:

$$V[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1,j] & \text{if } j < s_i \\ \max\{V[i-1,j], V[i-1,j-s_i]+v_i\} & \text{if } i > 0 \text{ and } j \geq s_i. \end{cases}$$

Algorithm 7.4 KNAPSACK

Input: A set of items $U = \{u_1, u_2, \ldots, u_n\}$ with sizes $s_1, s_2, \ldots, s_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack capacity C.

Output: The maximum value of the function $\sum_{u_i \in S} v_i$ subject to $\sum_{u_i \in S} s_i \leq C$ for some subset of items $S \subseteq U$.

```
for i ← 0 to n
    V[i, 0] ← 0
end for
for j ← 0 to C
    V[0, j] ← 0
end for
for i ← 1 to n
    for j ← 1 to C
        V[i, j] ← V[i − 1, j]
        if s_i ≤ j then V[i, j] ← max{V[i, j], V[i − 1, j − s_i] + v_i}
    end for
end for
return V[n, C]
```

The Knapsack Exemplo

- Considere os seguintes itens: s=(2, 3, 4, 5) com os respectivos valores v=(3, 4, 5, 7). Resolva esta instância do problema informando que itens entrarão na mochila e seu valor objetivo.

The Knapsack - Exemplo

	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7	7
3	0	0	3	4	4	7	8	9	9	12
4	0	0	3	4	5	7	8	10	11	12

Rod-cutting problem

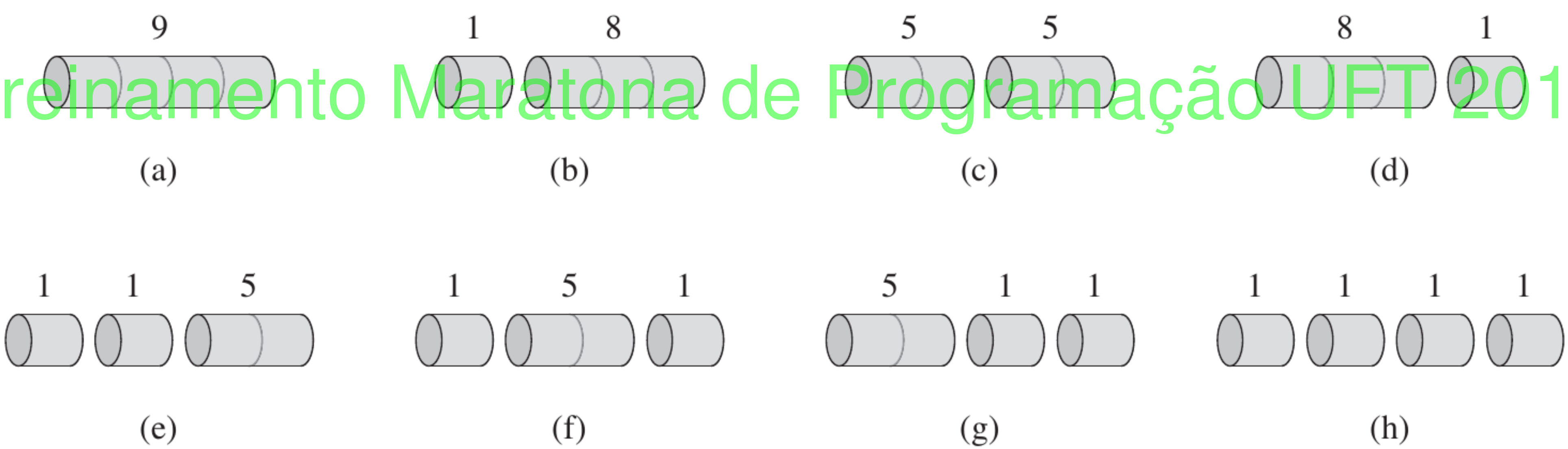
- A Pirassununga Enterprises compra longas barras, vendendo pedaços destas barras por um preço p_i

The ***rod-cutting problem*** is the following. Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod-cutting problem

- Uma barra de tamanho n pode ser cortada em 2^{n-1} maneiras diferentes.



length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Qual o corte ótimo?

Rod-cutting problem

- Decomposição como soma:
  - 7 = 2+2+3 → uma barra de tamanho 7 é cortada em três pedaços de tamanho 2, 2 e 3, respectivamente.
  - Se uma solução ótima corta a barra em k pedaços, para $1 \leq k \leq n$:
    - $n = i_1 + i_2 + \cdots + i_k$
  - Rende: $r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$

Rod-cutting problem

- Preencha a lista abaixo com base na tabela dada:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \text{ (no cuts)} \\
r_2 &= 5 && \text{from solution } 2 = 2 \text{ (no cuts)} \\
r_3 &= 8 && \text{from solution } 3 = 3 \text{ (no cuts)} \\
r_4 &= && \text{from solution } 4 = \\
r_5 &= && \text{from solution } 5 = \\
r_6 &= && \text{from solution } 6 = \\
r_7 &= && \text{from solution } 7 = \\
r_8 &= && \text{from solution } 8 = \\
r_9 &= && \text{from solution } 9 = \\
r_{10} &= && \text{from solution } 10 =
\end{aligned}
$$

Rod-cutting problem

- Preencha a lista abaixo com base na tabela dada:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} ,\\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} ,\\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} ,\\
r_4 &= 10 && \text{from solution } 4 = 4 = 2 + 2 ,\\
r_5 &= 13 && \text{from solution } 5 = 5 = 2 + 3 ,\\
r_6 &= 17 && \text{from solution } 6 = 6 = 6 \quad \text{(no cuts)} ,\\
r_7 &= 18 && \text{from solution } 7 = 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3 ,\\
r_8 &= 22 && \text{from solution } 8 = 8 = 2 + 6 ,\\
r_9 &= 25 && \text{from solution } 9 = 9 = 3 + 6 ,\\
r_{10} &= 30 && \text{from solution } 10 = 10 = 10 \quad \text{(no cuts)} .
\end{aligned}
$$

r_2+r_2

Rod-cutting problem

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

Não fazer cortes

Soma dos revenues entre cortar duas peças conforme r_i e r_{n-i}

$$\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)} , \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)} , \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)} , \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2 , \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3 , \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)} , \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \ \text{ or } \ 7 = 2 + 2 + 3 , \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6 , \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6 , \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)} .
\end{aligned}$$

Rod-cutting problem

$$r_n = \max\left(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1\right) .$$

Não fazer cortes

Soma dos revenues entre cortar duas peças conforme r_i e r_{n-i}

$$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i}\right)$$

Com r_0 = 0

```
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

Problema: muitas chamadas recursivas. N=40 já começa a ser um problema.

Rod-cutting problem – Solução usando PD

```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = -∞
    for i = 1 to j
        q = max(q, p[i] + r[j - i])
    r[j] = q
return r[n]
```


Crie uma versão que retorna onde estão os cortes.

Algoritmos Gulosos

Discussão inicial

- Em geral, aplicados a problemas de otimização
  - Deseja-se maximizar ou minimizar alguma quantidade
- Algoritmo iterativo: encontra a melhor solução local (*pode* ser o ótimo global)
- Constrói a solução passo a passo, dentro do horizonte presente de solução
- A cada passo, escolhe a solução que produz o melhor resultado imediato (enquanto mantém a *feasibility* da solução)

Um simples exemplo

Example 8.1 Consider the *fractional knapsack problem* defined as follows. Given n items of sizes $s_1, s_2, \ldots, s_n$, and values $v_1, v_2, \ldots, v_n$ and size C, the knapsack capacity, the objective is to find nonnegative *real numbers* $x_1, x_2, \ldots, x_n$ that maximize the sum

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

subject to the constraint

$$\sum_{i=1}^{n} x_i s_i \leq C.$$

Um simples exemplo

Example 8.1 Consider the *fractional knapsack problem* defined as follows. Given n items of sizes $s_1, s_2, \ldots, s_n$, and values $v_1, v_2, \ldots, v_n$ and size C, the knapsack capacity, the objective is to find nonnegative *real numbers* $x_1, x_2, \ldots, x_n$ that maximize the sum

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

subject to the constraint

$$\sum_{i=1}^{n} x_i s_i \leq C.$$

Ordene os valores por: e para cada item x_i, coloque o máximo de quantidade do elemento segundo y_i.

$$y_i = v_i / s_i$$

The Shortest Path Problem

- Seja G = (V,E) um grafo direcionado com pesos (não negativos) em suas arestas;
- Seja s um vértice (source) de v.

- Shortest Path Problem de uma fonte só (s) consiste em encontrar o caminho mais curto (em termos dos pesos) de s até qualquer outro vértice em G.
- Algoritmo guloso: Dijkstra's algorithm.

Dijkstra's algorithm

- Por simplicidade, vamos assumir:
  - S=1, V = {1, 2, 3, …, n}
  - V é dividido em dois conjuntos X = {1} e Y ={2, 3, …, n}
  - X→ vértices para qual a distância até o source foi determinada.

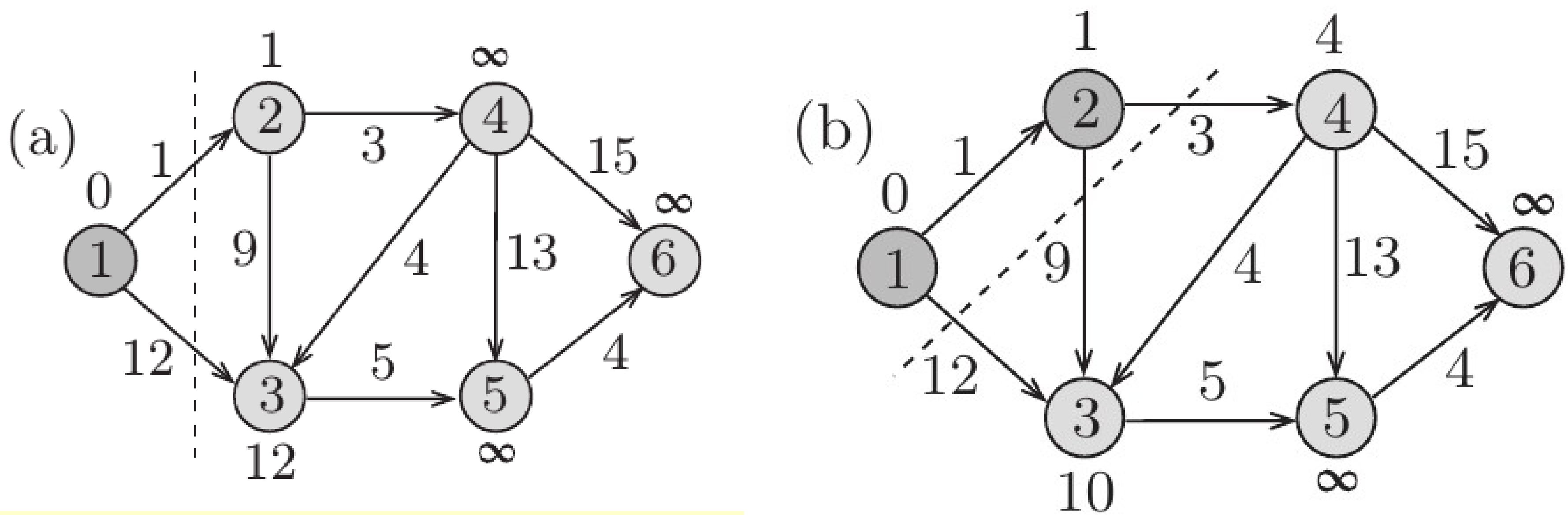
Ideia: A cada passo, selecionamos um vértice de Y para qual a distância até *s* for determinada (então o movemos para X).

Dijkstra's algorithm

- *Ideia*: A cada passo, selecionamos um vértice de Y para qual a distância até *s* for determinada (então o movemos para X).
- label[y] → tamanho do caminho mais curto que usa apenas vértices de X.
- *Quando um y é movido para X*: marcamos todo w adjacente a y, indicando que um caminho mais curto através de y foi descoberto, atualizando o peso virtual até *s*.
- delta[v] → distância de v para s.

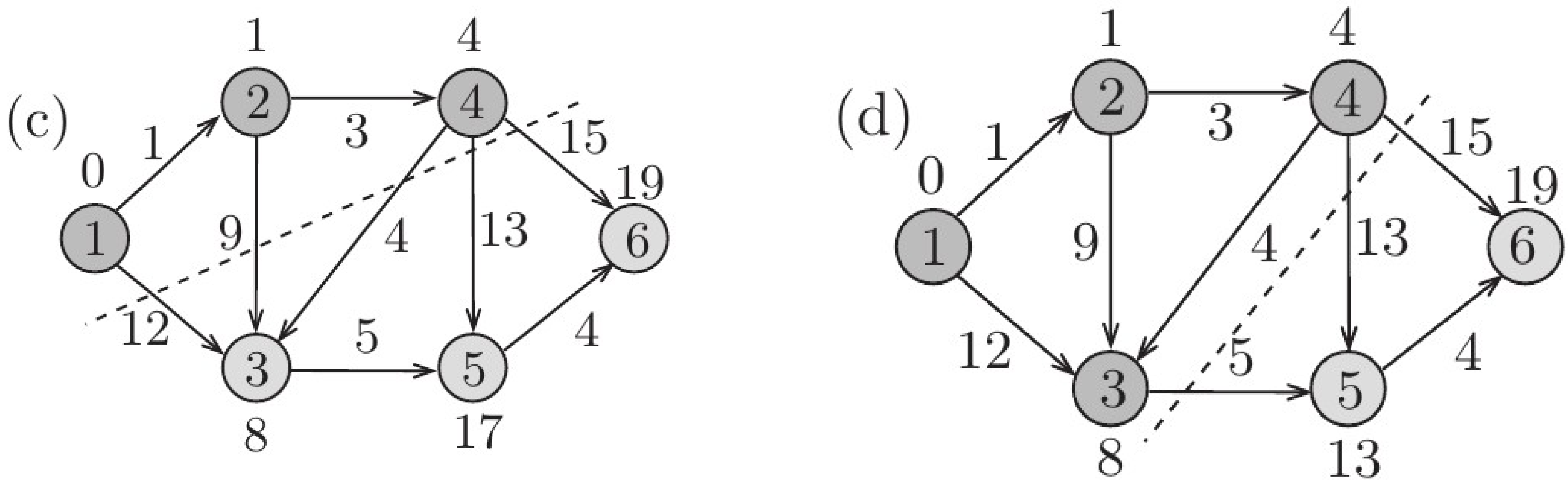
Dijkstra's algorithm

1. $X \leftarrow \{1\}; \quad Y \leftarrow V - \{1\}$
2. For each vertex $v \in Y$ if there is an edge from 1 to v then let $\lambda[v]$ (the label of v) be the length of that edge; otherwise let $\lambda[v] = \infty$. Let $\lambda[1] = 0$.
3. **while** $Y \neq \{\}$
4. Let $y \in Y$ be such that $\lambda[y]$ is minimum.
5. move y from Y to X.
6. update the labels of those vertices in Y that are adjacent to y.
7. **end while**



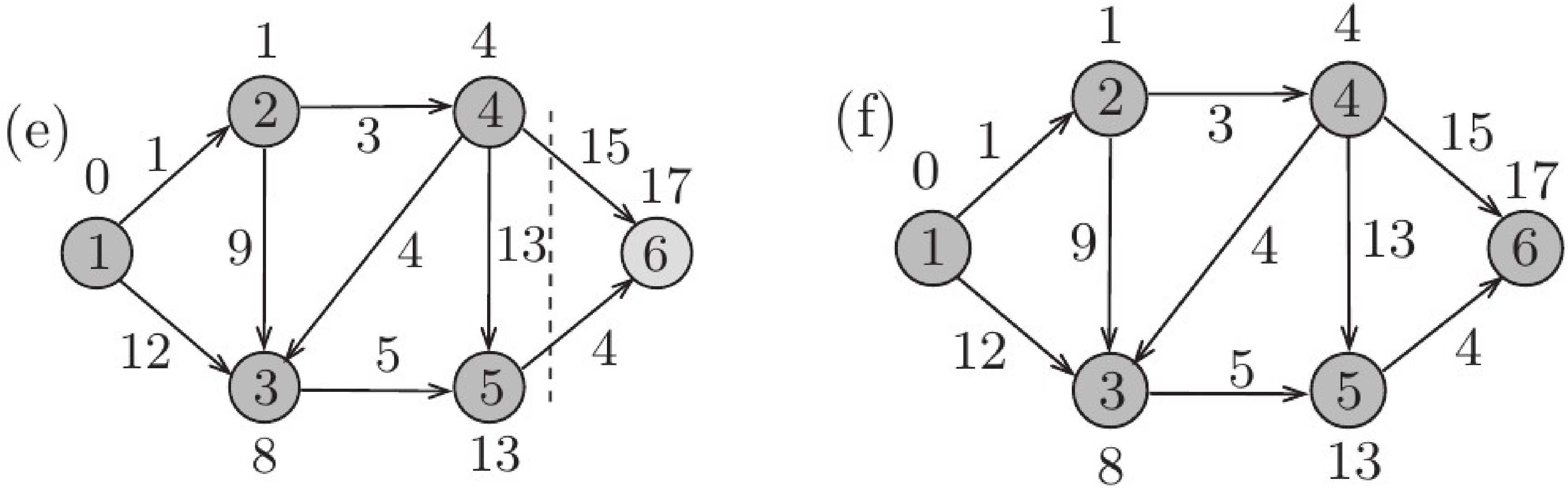
Dijkstra's algorithm

1. $X \leftarrow \{1\}$; $\quad Y \leftarrow V - \{1\}$
2. For each vertex $v \in Y$ if there is an edge from 1 to v then let $\lambda[v]$ (the label of v) be the length of that edge; otherwise let $\lambda[v] = \infty$. Let $\lambda[1] = 0$.
3. **while** $Y \neq \{\}$
4. Let $y \in Y$ be such that $\lambda[y]$ is minimum.
5. move y from Y to X.
6. update the labels of those vertices in Y that are adjacent to y.
7. **end while**

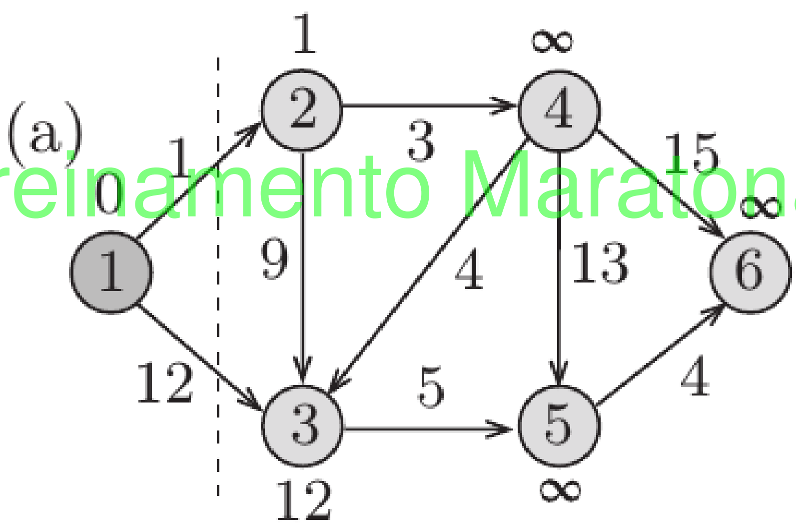


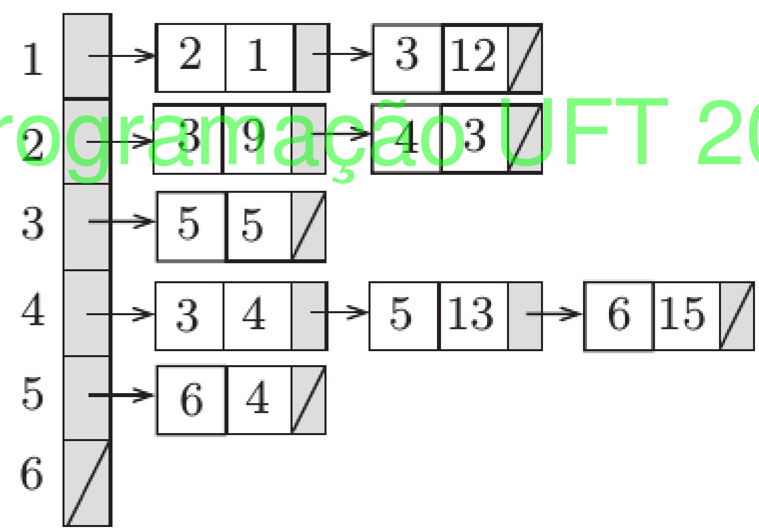
Dijkstra's algorithm

1. $X \leftarrow \{1\}; \quad Y \leftarrow V - \{1\}$
2. For each vertex $v \in Y$ if there is an edge from 1 to v then let $\lambda[v]$ (the label of v) be the length of that edge; otherwise let $\lambda[v] = \infty$. Let $\lambda[1] = 0$.
3. **while** $Y \neq \{\}$
4. Let $y \in Y$ be such that $\lambda[y]$ is minimum.
5. move y from Y to X.
6. update the labels of those vertices in Y that are adjacent to y.
7. **end while**



Dijkstra's algorithm – Detalhes de implementação





Dijkstra's Algorithm

1. $X \leftarrow \{1\}; \quad Y \leftarrow V - \{1\}$
2. For each vertex $v \in Y$ if there is an edge from 1 to v then let $\lambda[v]$ (the label of v) be the length of that edge; otherwise let $\lambda[v] = \infty$. Let $\lambda[1] = 0$.
3. **while** $Y \neq \{\}$
4. Let $y \in Y$ be such that $\lambda[y]$ is minimum.
5. move y from Y to X.
6. update the labels of those vertices in Y that are adjacent to y.
7. **end while**

Contabilizando o tempo, implementem este algoritmo. Os de mais conhecimento, liderem times com os de menos conhecimento.

Minimum Cost Spanning Trees

Definition 8.1 Let $G = (V, E)$ be a connected undirected graph with weights on its edges. A *spanning tree* (V, T) of G is a subgraph of G that is a tree. If G is weighted and the sum of the weights of the edges in T is minimum, then (V, T) is called a *minimum cost spanning tree* or simply a *minimum spanning tree*.

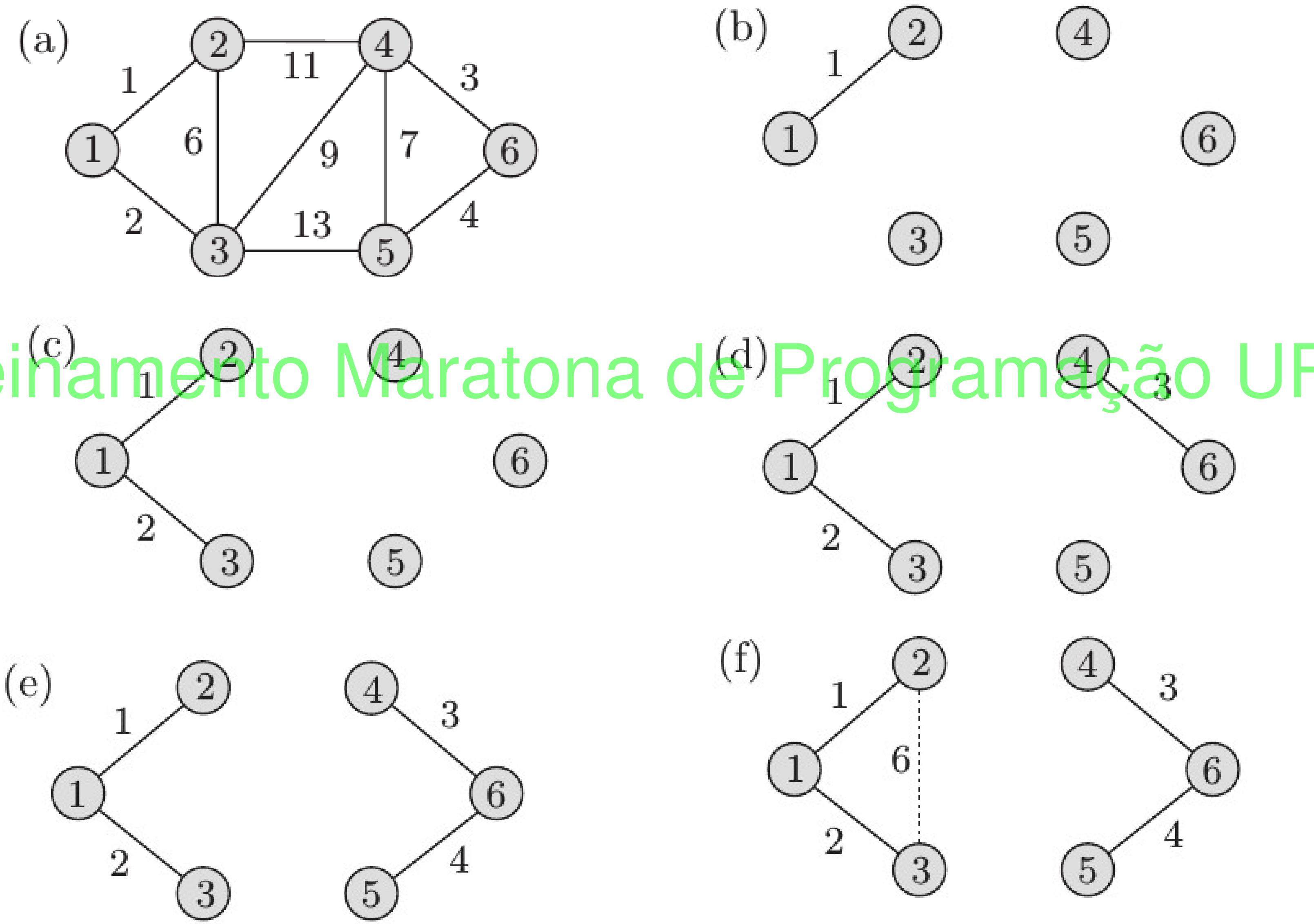
- Assumir que G é conexo;
- Algoritmo de Kruskal

Algoritmo de Kruskal

- Ordene as arestas em ordem decrescente de seus pesos
- Inicialize uma floresta (V,T), com T={}

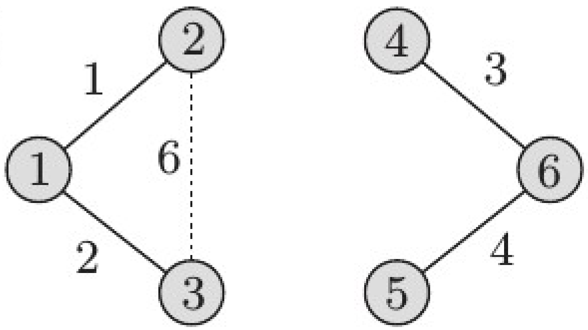
- Enquanto $|T| < n\text{-}1$
  - Seja e uma aresta pertencente a $E \setminus T$
  - Se adicionando e a T não cria um ciclo, então adicione, caso contrário descarte e

Algoritmo de Kruskal

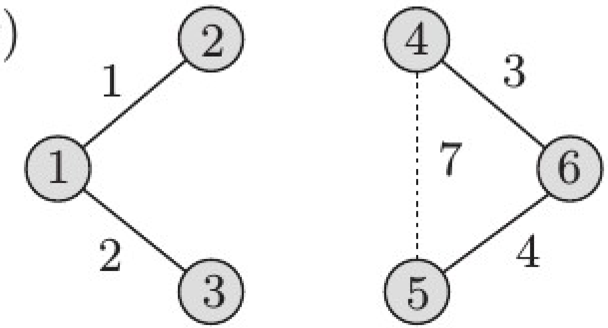


Algoritmo de Kruskal

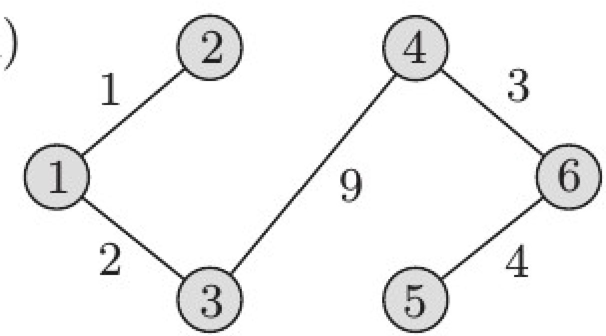
(f)



(g)



(h)



Contabilizando o tempo, implementem este algoritmo. Os de mais conhecimento, liderem times com os de menos conhecimento.

Algoritmo de Kruskal - Implementação

Algorithm 8.3 KRUSKAL

Input: A weighted connected undirected graph $G = (V, E)$ with n vertices.

Output: The set of edges T of a minimum cost spanning tree for G.

Sort the edges in E by nondecreasing weight.
for each vertex $v \in V$
    MAKESET($\{v\}$)
end for
$T = \{\}$
while $|T| < n - 1$
    Let (x, y) be the next edge in E.
    if FIND(x) $\neq$ FIND(y) **then**
        Add (x, y) to T
        UNION(x, y)
    end if
end while

FIND(x) retorna o *id* do conjunto a qual x pertence.

Disjoint Sets Data Structures

- Suponha S um conjunto com elementos distintos
  - Os elementos podem ser particionados em conjuntos disjuntos;
  - Inicialmente, consideraremos que cada elemento pertence a um conjunto separado.
  - Número de operações possíveis de unions: (máximo de n-1)

Disjoint Sets Data Structures

- Em cada subconjunto um elemento distinto pode ser usado para identificar aquele subconjunto
  - *Representante do conjunto* ou *nome do conjunto*.
  - Ex.: Seja S = {1, 2, … , 11} e exista quatro subconjuntos: {1, 7, 10, 11}, {2, 3, 5, 6}, {4,8} e {9}
  - Identificadores: 1, 3, 8 e 9, por exemplo.

Disjoint Sets Data Structures

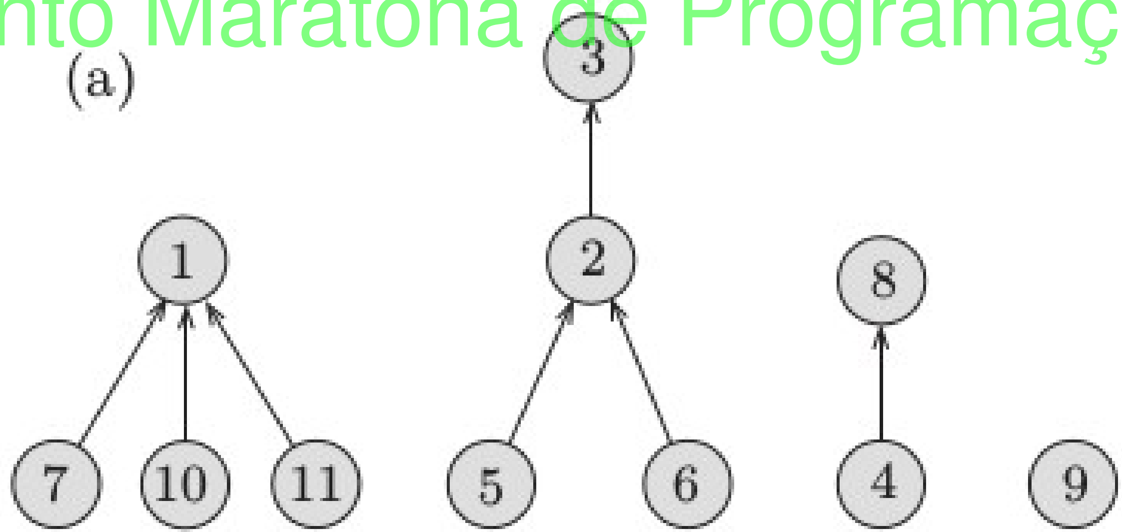
- Duas operações:
  - FIND(x) → retorna o *nome do conjunto* contendo x.
  - UNION(x,y)→ substitui os conjuntos onde x e y pertencem pela união de seus conjuntos.

    - O nome do conjunto resultante pode ser tanto o de x quanto o de y.
- O que precisamos: uma DS que seja simples e permita implementar eficientemente as duas operações acima.
  - Que tal uma árvore com elementos armazenados em seus nós?

Disjoint Sets Data Structures

- Cada nó x, tem um parent(x). A raiz do conjunto tem um parent =null.

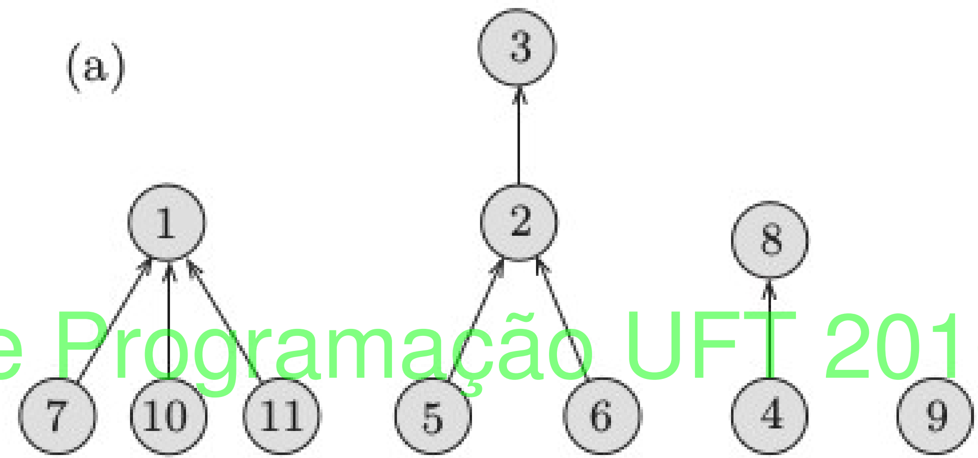
  Seja S = {1, 2, … , 11} e exista quatro subconjuntos: {1, 7, 10, 11}, {2, 3, 5, 6}, {4,8} e {9}



Disjoint Sets Data Structures

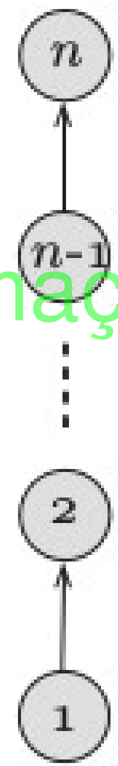
- FIND(x) : siga o caminho de parent em parent até chegar na raiz, então retorne a raiz.

- UNION(x,y)→ faça root(x) ser parent(y) ou vice-versa.



Disjoint Sets Data Structures

- Desvantagens da abordagem anterior:
  - Suponha os conjuntos: {1}, {2}, …, {n}
  - Execute:
    - union(1, 2), union(2, 3), . . . , union(n – 1, n),
    - find(1), find(2), . . . , find(n) → proporcional a n^2 visitas aos nós.



(a)

Disjoint Sets Data Structures

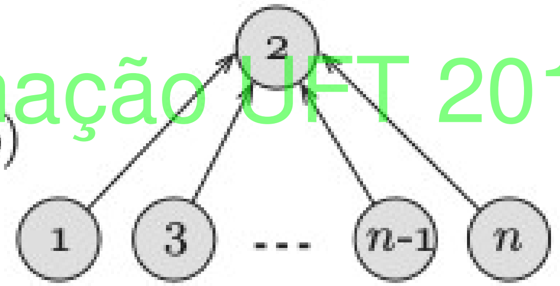
- Union by rank heuristic:
  - Associa a cada nó um número não negativo (rank que é a altura do nó na árvore).
  - Os nós iniciam com rank=0

  - Union(x,y):
    - if (rank(x) < rank(y) parent(x) = y
    - if (rank(x) > rank(y) parent(y) = x
    - Otherwise:
      - parent(x)
      - rank(y) = rank(y)+1

Disjoint Sets Data Structures

- Union(x,y):
    - if (rank(x) < rank(y) parent(x) = y
    - if (rank(x) > rank(y) parent(y) =
    - Otherwise:
        - parent(x)
        - rank(y) = rank(y)+1
  - Suponha os conjuntos: {1}, {2}, …, {n}
  - Execute:
    - union(1, 2), union(2, 3), . . . , union(n – 1, n),

(b)

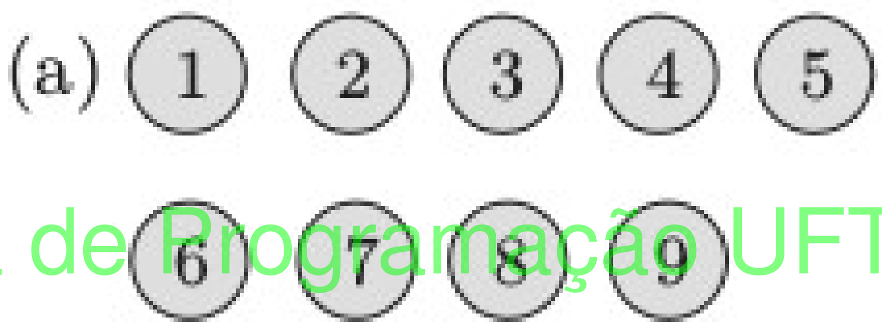


Disjoint Sets Data Structures

- Melhorar a performance da operação find: *path compression heuristic*
- Find(x) após encontrar a raiz (digamos y)→ percorremos o caminho de x a y novamente, trocando o parent de cada nó no caminho para y.

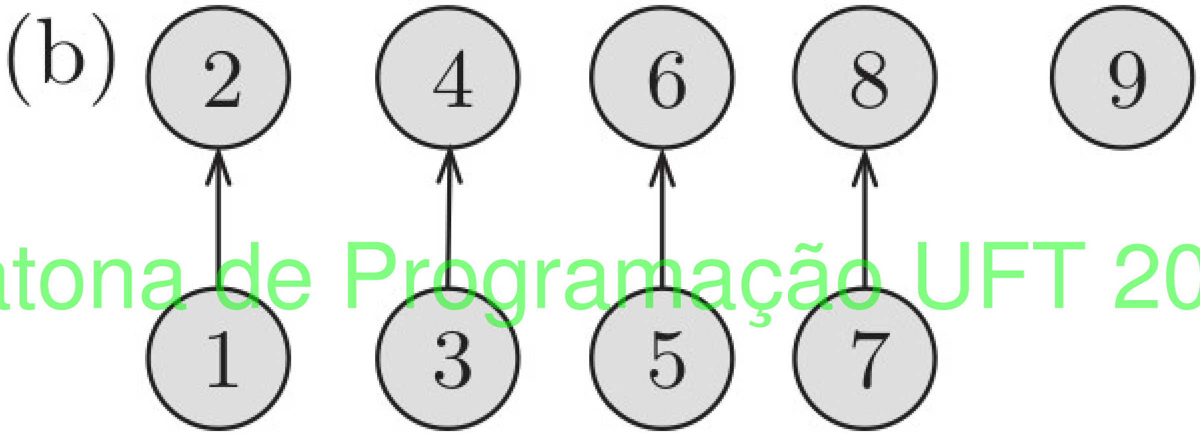
Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - union(1, 2),
  - union(3, 4),
  - union(5, 6),
  - union(7, 8),
  - union(2, 4),
  - union(8, 9),
  - union(6, 8),
  - find(5),
  - union(4, 8),
  - find(1)



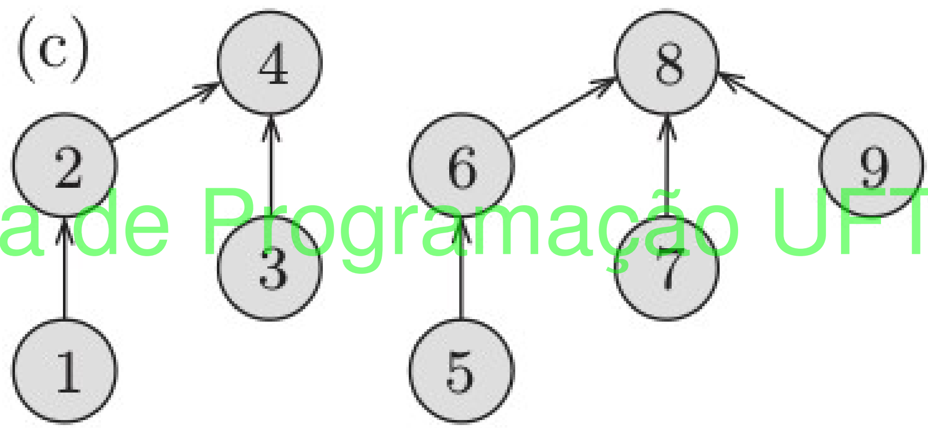
Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - ~~union(1, 2),~~
  - ~~union(3, 4),~~
  - ~~union(5, 6),~~
  - ~~union(7, 8),~~
  - union(2, 4),
  - union(8, 9),
  - union(6, 8),
  - find(5),
  - union(4, 8),
  - find(1)



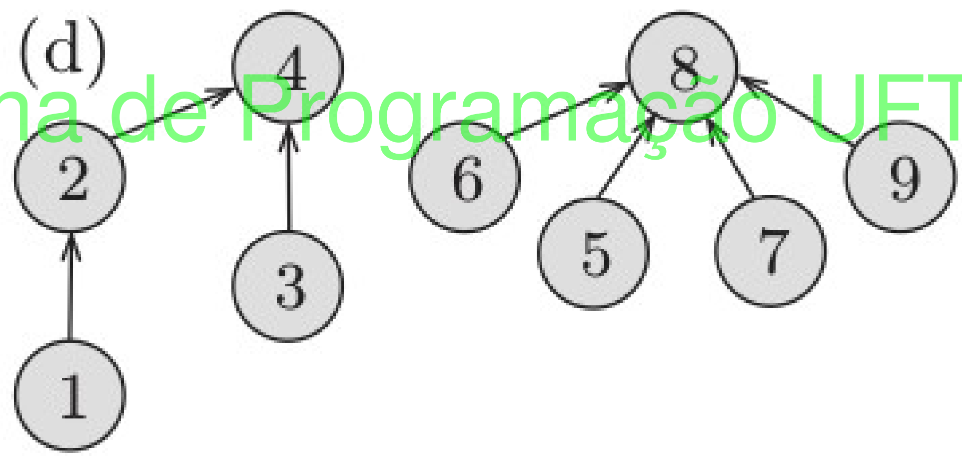
Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - ~~union(1, 2),~~
  - ~~union(3, 4),~~
  - ~~union(5, 6),~~
  - ~~union(7, 8),~~
  - ~~union(2, 4),~~
  - ~~union(8, 9),~~
  - ~~union(6, 8)~~,
  - find(5),
  - union(4, 8),
  - find(1)



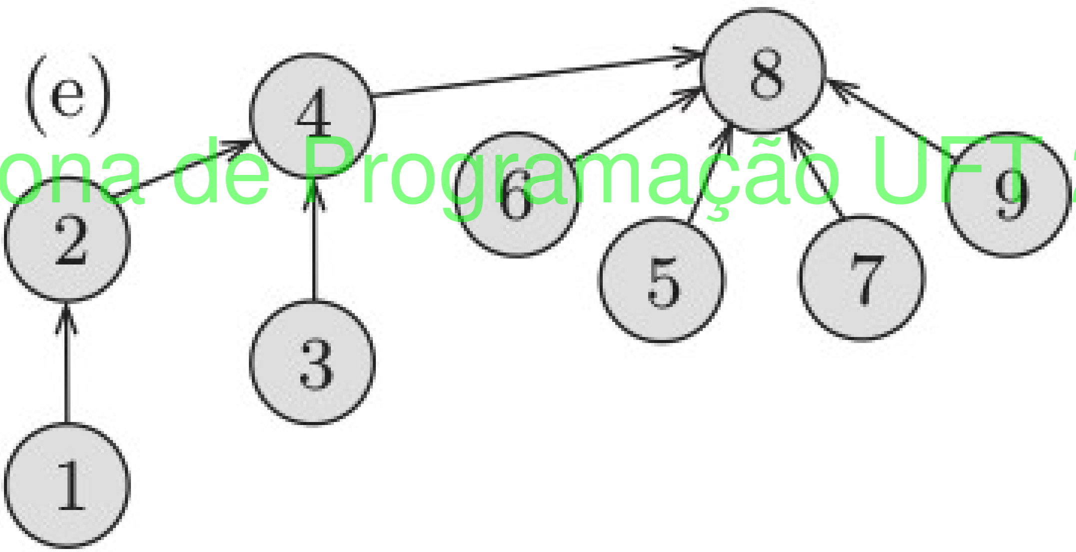
Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - ~~union(1, 2),~~
  - ~~union(3, 4),~~
  - ~~union(5, 6),~~
  - ~~union(7, 8),~~
  - ~~union(2, 4),~~
  - ~~union(8, 9),~~
  - ~~union(6, 8)~~,
  - ~~find(5),~~
  - union(4, 8),
  - find(1)



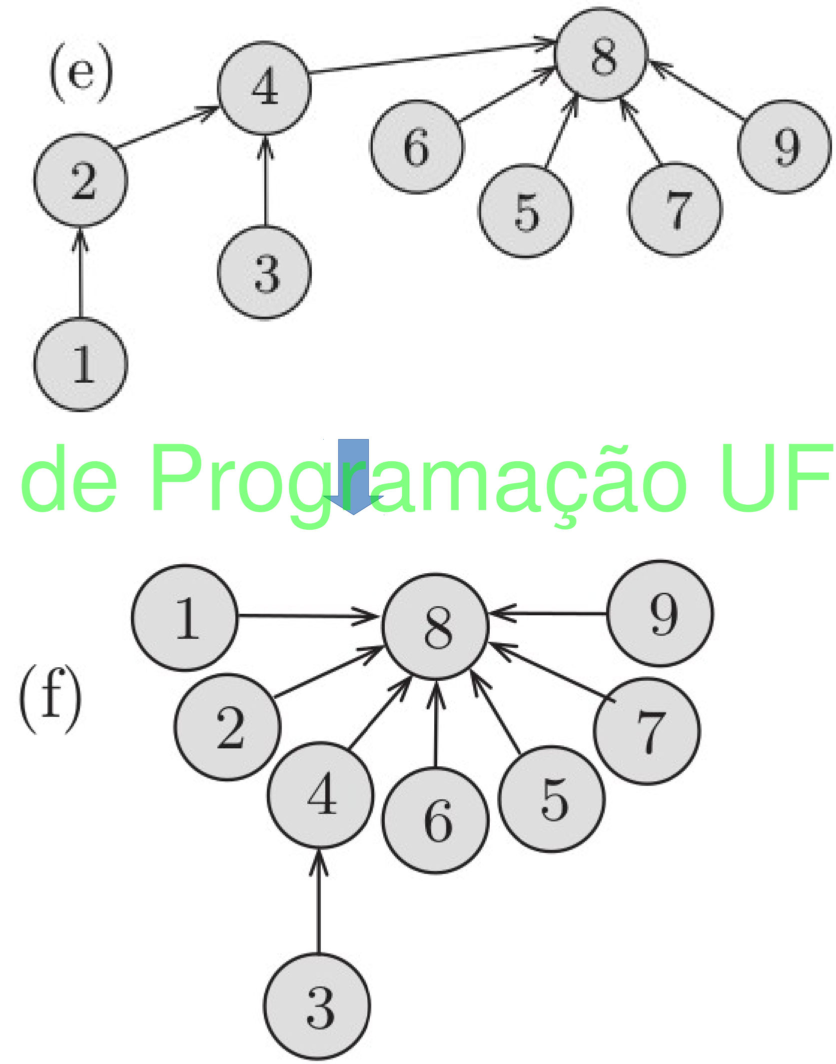
Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - ~~union(1, 2),~~
  - ~~union(3, 4),~~
  - ~~union(5, 6),~~
  - ~~union(7, 8),~~
  - ~~union(2, 4),~~
  - ~~union(8, 9),~~
  - ~~union(6, 8),~~
  - ~~find(5),~~
  - ~~union(4, 8),~~
  - find(1)



Disjoint Sets Data Structures

- Seja S = {1, 2, …, 9}
  - ~~union(1, 2),~~
  - ~~union(3, 4),~~
  - ~~union(5, 6),~~
  - ~~union(7, 8),~~
  - ~~union(2, 4),~~
  - ~~union(8, 9),~~
  - ~~union(6, 8),~~
  - ~~find(5),~~
  - ~~union(4, 8),~~
  - ~~find(1)~~



Algoritmo de Kruskal - Implementação

Algorithm 8.3 KRUSKAL

Input: A weighted connected undirected graph $G = (V, E)$ with n vertices.

Output: The set of edges T of a minimum cost spanning tree for G.

```
Sort the edges in E by nondecreasing weight.
for each vertex v ∈ V
    MAKESET({v})
end for
T = {}
while |T| < n − 1
    Let (x, y) be the next edge in E.
    if FIND(x) ≠ FIND(y) then
        Add (x, y) to T
        UNION(x, y)
    end if
end while
```

FIND(x) retorna o *id* do conjunto a qual x pertence.

Backtracking

Discussão inicial

- Suponha que você tenha que posicionar N rainhas em um tabuleiro de xadrez (NxN),de forma que nenhuma rainha ameace a outra.
- Como atacar este problema?

Backtracking

- Uma maneira sistemática de iterar por todas as possíveis configurações de um espaço de busca.
  - Permutações de objetos
  - Todos os subconjuntos
  - Enumerar todas as árvores geradoras de um grafo
  - Todos os caminhos entre dois vértices
  - etc...

Backtracking

- Comum a estes problemas: gerar cada configuração possível apenas uma vez!
- Backtracking permite definir uma ordem sistemática para evitar repetições e configurações errôneas!
- Solução de busca combinatorial como um vetor $a = (a1, a2, \ldots, an)$ onde ai é selecionado de um conjunto finito Si.
  - Representar um arranjo onde ai contem o i-ésimo elemento da permutação;
  - Representar um dado subconjunto onde ai é verdadeiro ou falso dizendo se o i-ésimo elemento do conjunto está presente ou não.
  - Uma sequência de movimentos em um jogo
  - Um caminho em um grafo.

Backtracking

- A cada passo do algoritmo tentamos avançar uma solução a = (a1, a2, … , ak) por adicionar outro elemento ao final do vetor
  - Caso seja uma solução→ reportamos ou contamos
  - Caso não seja, verificamos se a subsolução é ainda explorável
- Backtracking constrói uma árvore de soluções parciais
  - Cada vértice representa uma solução parcial
  - A aresta (x,y) existe se y foi criado avançando a solução x
  - O processo consiste em realizar uma busca em profundidade na árvore gerada pelo backtracking.

Voltando ao problema das rainhas

- Suponha que você tenha que posicionar N rainhas em um tabuleiro de xadrez (NxN),de forma que nenhuma rainha ataque a outra.

	Q		
			Q
Q			
		Q	

Voltando ao problema das rainhas

- Suponha que você tenha que posicionar N rainhas em um tabuleiro de xadrez (NxN),de forma que nenhuma rainha ataque a outra.

0	1	2	3
	X	X	X
X	X	**Q**	X
	X	X	x
X		X	

Q = (1,2)

Linha	Coluna	Diag1	Diag2
(1,0)	(0,2)	(0,1)	(3,0)
(1,1)	(2,2)	(2,3)	(2,1)
(1,3)	(3,2)		(0,3)

Qual o padrão?

Voltando ao problema das rainhas

- Suponha que você tenha que posicionar N rainhas em um tabuleiro de xadrez (NxN),de forma que nenhuma rainha ataque a outra.

0	1	2	3
	X	X	X
X	X	**Q**	X
	X	X	x
X		X	

Diag1

Q = (1,2) **1-2 = -1 (padrão da Diag1)**

Linha	Coluna	Diag1	Diag2
(1,0)	(0,2)	**(0,1)**	(3,0)
(1,1)	(2,2)	**(2,3)**	(2,1)
(1,3)	(3,2)		(0,3)

Voltando ao problema das rainhas

- Suponha que você tenha que posicionar N rainhas em um tabuleiro de xadrez (NxN),de forma que nenhuma rainha ataque a outra.

0	1	2	3
	X	X	X
X	X	**Q**	X
	X	X	x
X		X	

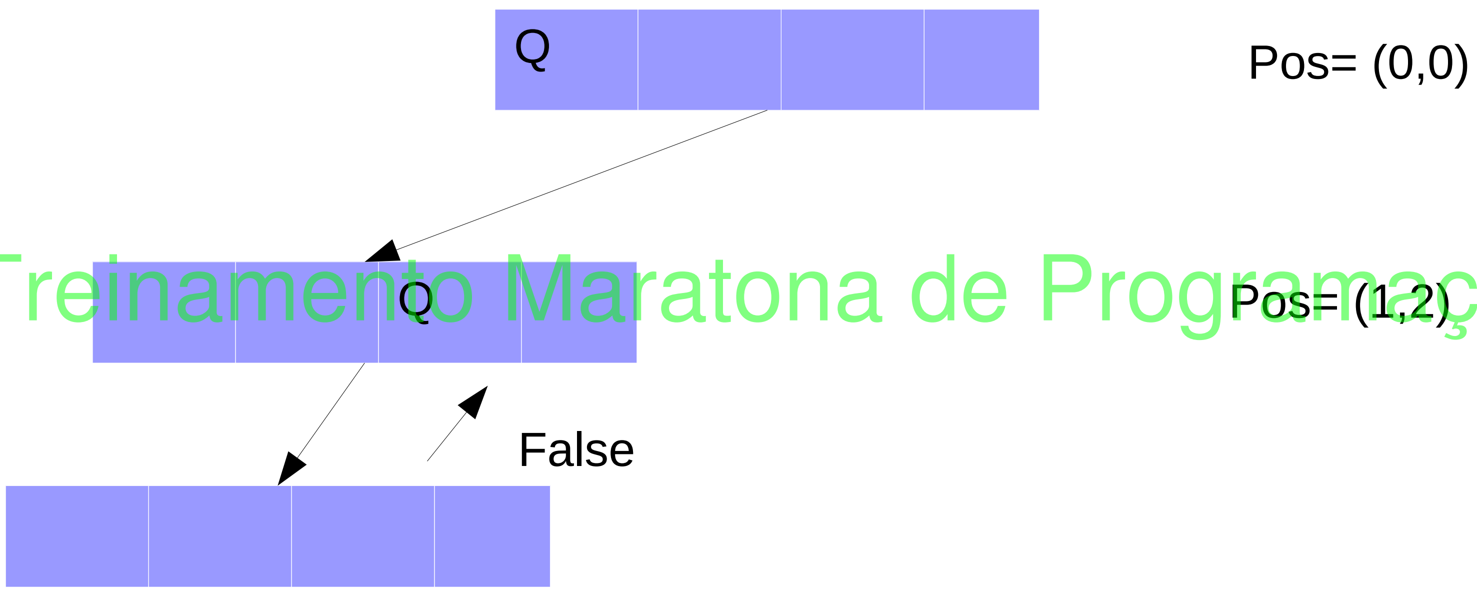
Diag2 Padrão (row+column)

Q = (1,2) **1+2 = 3 (padrão da Diag2)**

Linha	Coluna	Diag1	Diag2
(1,0)	(0,2)	(0,1)	**(3,0)**
(1,1)	(2,2)	(2,3)	**(2,1)**
(1,3)	(3,2)		**(0,3)**

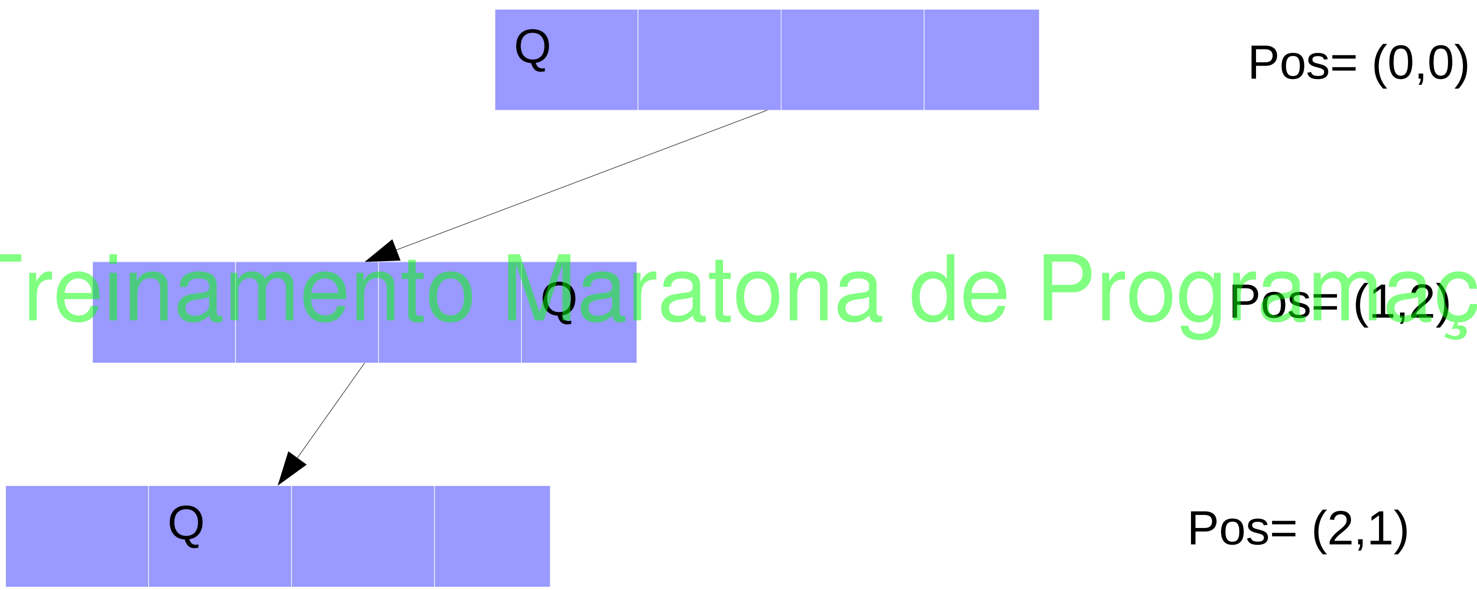
Voltando ao problema das rainhas

- Usando backtracking vamos estipular o posicionamento das rainhas, linha a linha.



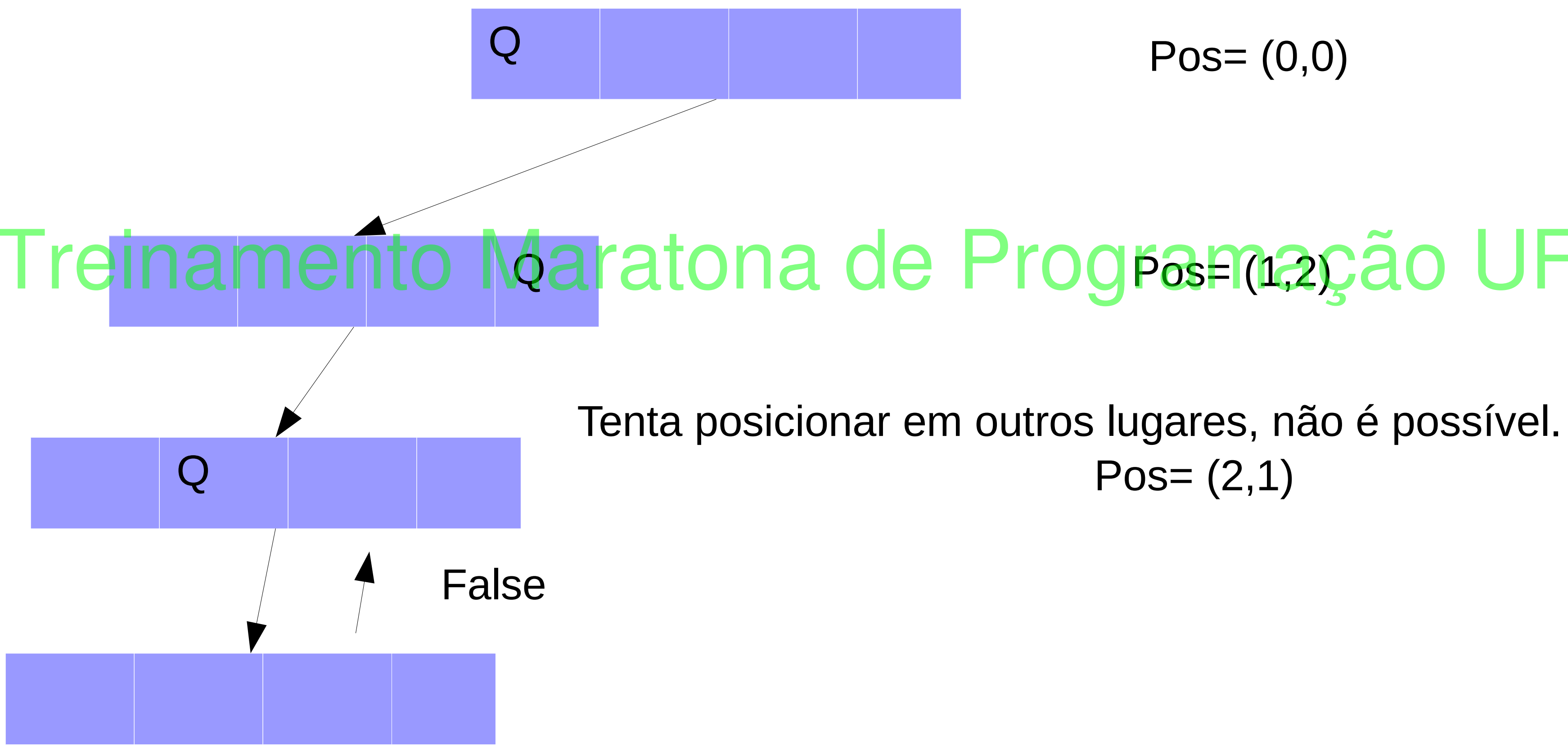
Voltando ao problema das rainhas

- Usando backtracking vamos estipular o posicionamento das rainhas, linha a linha.



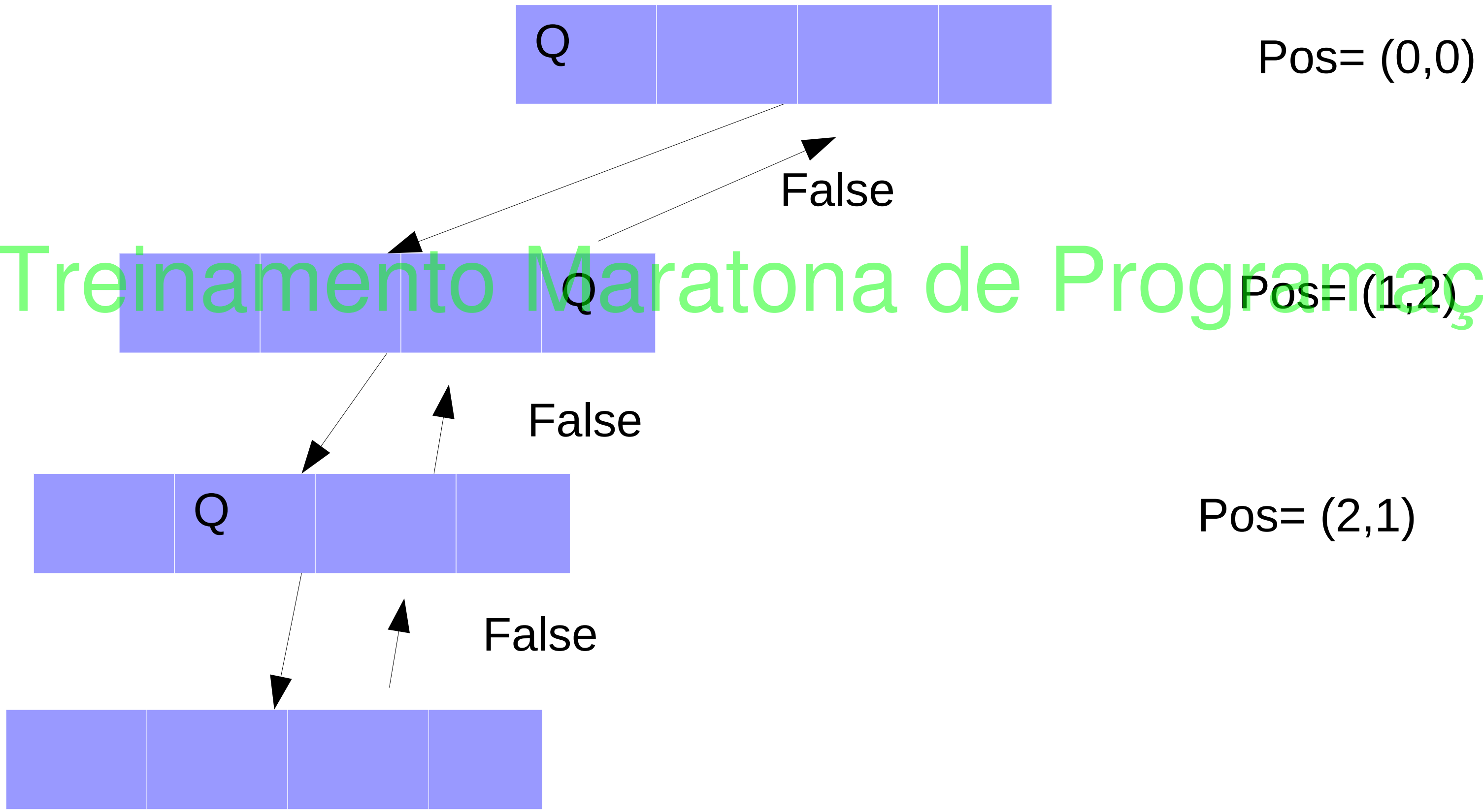
Voltando ao problema das rainhas

- Usando backtracking vamos estipular o posicionamento das rainhas, linha a linha.



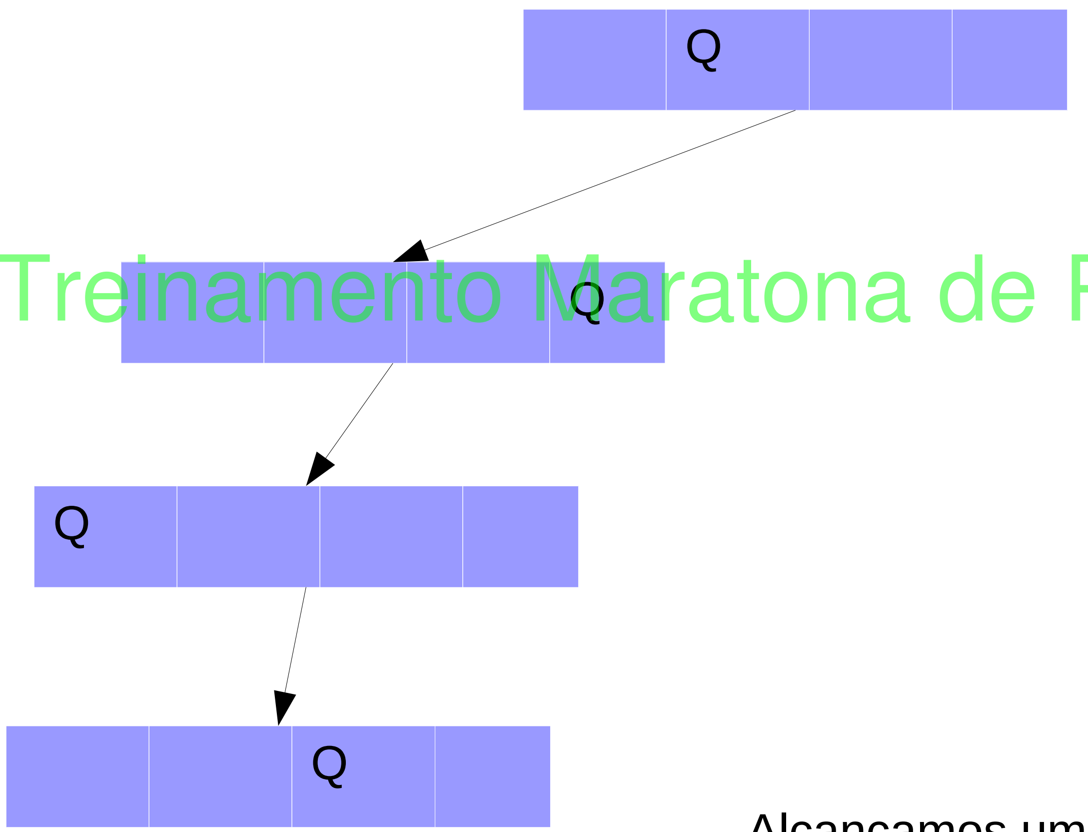
Voltando ao problema das rainhas

- Usando backtracking vamos estipular o posicionamento das rainhas, linha a linha.



Voltando ao problema das rainhas

- Usando backtracking vamos estipular o posicionamento das rainhas, linha a linha.



Pos= (0,1)

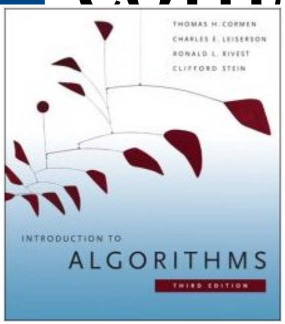
Pos= (1,3)

Pos= (2,0)

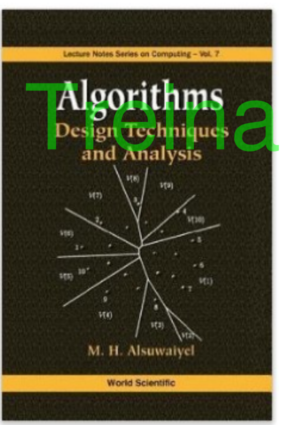
Pos= (3,2)

Alcançamos uma solução!!

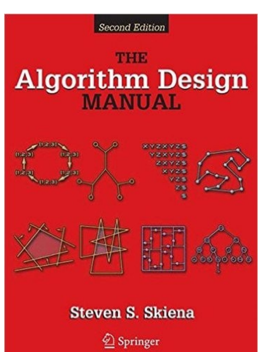
Bibliografia – Direitos das imagens e conteúdo



- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. Introduction to Algorithms, 3rd edition.



- Alsuwaiyel ALGORITHMS DESIGN TECHNIQUES AND ANALYSIS, 1999 World Sci.



- The Algorithm Design Manual 2010 by Steven S. Skiena