

# ALGORITMI UTILI

## PROGRAMMAZIONE DINAMICA:

Codice Python del problema del Knapsack:

```
def matrix(r, c, v):  
    m = []  
    for i in range(r):  
        m.append([])  
        for j in range(c):  
            m[i].append(v)  
    return m
```

```
def Knapsack(n, C, w, p):
```

# n= numero di oggetti a disposizione

# C= capacità dello zaino

# w= lista contenente il peso di ciascun oggetto

# p= lista contenente il profitto di ciascun oggetto

```
    DP = matrix(n+1, C+1, 0)
```

```
    for i in range(n):
```

```
        for c in range(C+1):
```

```
            if (w[i] <= c):
```

$$DP[i+1][c] = \max(DP[i][c], DP[i][c-w[i]] + p[i])$$

else:

$$DP[i+1][c] = DP[i][c]$$

```
return DP[n][C]
```

# ANALISI DELLE RICORRENZE

## ANALISI DEI LIVELLI (o "SROTOLAMENTO"):

$$T(n) = \begin{cases} 1 & \text{se } n=0 \\ 4T(\frac{n}{2}) + n & \text{se } n>0 \end{cases}$$

Espando la relazione:  $T(n) = 4T(\frac{n}{2}) + n = 4(4T(\frac{n}{4}) + \frac{n}{2}) + n = 16T(\frac{n}{4}) + 2n + n = 16(4T(\frac{n}{8}) + \frac{n}{4}) + 2n + n = 64T(\frac{n}{8}) + 4n + 2n + n = \dots =$

$$= \underbrace{n^2}_{\approx T(1)} + n \left( \frac{n}{2} + \dots + 4 + 2 + 1 \right) = n^2 + n \left( 2^{\log n - 1} + \dots + 2^2 + 2^1 + 2^0 \right) =$$

$$= n^2 + n \sum_{i=0}^{\log n - 1} 2^i = n^2 + n \frac{1 - 2^{\log n}}{1 - 2} = n^2 + n (2^{\log n} - 1) = 2n^2 - n = \Theta(n^2)$$

## METODO DI SOSTITUZIONE (CON DIMOSTRAZIONE PER INDUZIONE):

$$T(n) = \begin{cases} T(\lfloor \frac{n}{2} \rfloor) + n & \text{se } n > 1 \\ 1 & \text{se } n \leq 1 \end{cases}$$

$$\begin{aligned} T(n) &= T(\lfloor \frac{n}{2} \rfloor) + n = T(\lfloor \frac{n}{4} \rfloor) + \lfloor \frac{n}{2} \rfloor + n = T(\lfloor \frac{n}{8} \rfloor) + \lfloor \frac{n}{4} \rfloor + \lfloor \frac{n}{2} \rfloor + n \leq \\ &\leq n \left( 2^0 + 2^{-1} + 2^{-2} + \dots + 2^{-\log n} \right) = n \left( (\frac{1}{2})^0 + (\frac{1}{2})^1 + (\frac{1}{2})^2 + \dots + (\frac{1}{2})^{\log n} \right) = \\ &= n \sum_{i=0}^{\log n} \left( \frac{1}{2} \right)^i \leq n \sum_{i=0}^{+\infty} \left( \frac{1}{2} \right)^i = n \frac{1}{1 - \frac{1}{2}} = 2n \end{aligned}$$

Provo a dimostrare per induzione che  $\overline{T}(n) = O(n) \wedge \underline{T}(n) = \Omega(n) \Rightarrow T(n) = \Theta(n)$   
 $\rightarrow T(n) = O(n) \Leftrightarrow \exists c \in \mathbb{R} \ \exists n_0 \in \mathbb{N} : T(n) \leq cn \quad \forall n \geq n_0$ .

PASSO BASE:  $T(1) \leq c \cdot 1 \Rightarrow c \geq 1$

PASSO INDUTTIVO:  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n \leq c \lfloor \frac{n}{2} \rfloor + n \leq \frac{cn}{2} + n \leq cn$

$$\text{per } \frac{c}{2} + 1 \leq c \Rightarrow 1 \leq \frac{c}{2} \Rightarrow c \geq 2$$

$\Rightarrow T(n) \leq 2n \quad \forall n \geq 1 \Rightarrow T(n) = O(n)$

$\rightarrow T(n) = \Omega(n) \Leftrightarrow \exists c \in \mathbb{R} \ \exists n_0 \in \mathbb{N} : T(n) \geq cn \quad \forall n \geq n_0$

PASSO BASE:  $T(1) \geq c \cdot 1 \Rightarrow c \leq 1$

PASSO INDUTTIVO:  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + n \geq c \lfloor \frac{n}{2} \rfloor + n \geq \frac{cn}{2} - 1 + n = \left( \frac{c}{2} - \frac{1}{n} + 1 \right) n \geq cn$   
 $\text{per } \frac{c}{2} - \frac{1}{n} + 1 \geq c \Rightarrow c \leq 2 - \frac{2}{n}$

$\Rightarrow T(n) \geq n \quad \forall n \geq 1 \Rightarrow T(n) = \Omega(n)$

$$T(n) = O(n) \wedge T(n) = \Omega(n) \Rightarrow T(n) = \Theta(n)$$

# 1° CASO:

$$a_n = a_{n-1} + 2a_{n-2}$$

$$\text{con } a_0 = 2, \quad a_1 = 7$$

Equazione caratteristica:  $r^2 - r - 2 = 0 \Rightarrow r_1 = 2, \quad r_2 = -1$   
 $\Rightarrow a_n = \alpha_1 \cdot 2^n + \alpha_2 \cdot (-1)^n$

$$\alpha_0 = \alpha_1 + \alpha_2 = 2$$

$$\alpha_1 = 2\alpha_1 - \alpha_2 = 7 \quad \left. \right\} \Rightarrow \alpha_1 = 3, \quad \alpha_2 = -1$$

$$\Rightarrow a_n = 3 \cdot 2^n + (-1)^{n+1}$$

# 2° CASO:

$$a_n = 6a_{n-1} - 9a_{n-2}$$

Equazione caratteristica:  $\text{con } a_0 = 1, \quad a_1 = 6$

$$\Rightarrow a_n = \alpha_1 \cdot 3^n + \alpha_2 n \cdot 3^n$$

$$\alpha_0 = \alpha_1 + 0 = 1$$

$$\alpha_1 = 3\alpha_1 + 3\alpha_2 = 6 \quad \left. \right\} \Rightarrow \alpha_1 = 1, \quad \alpha_2 = 1$$

$$\Rightarrow a_n = 3^n(1+n)$$

# 3° CASO:

$$a_n = 6a_{n-1} - 11a_{n-2} + 6a_{n-3}$$

Equazione caratteristica:  $\text{con } a_0 = 2, \quad a_1 = 5, \quad a_2 = 15$   
 $r^3 - 6r^2 + 11r - 6 = 0 \Rightarrow r_1 = 1, \quad r_2 = 2, \quad r_3 = 3$

$$\Rightarrow a_n = \alpha_1 + \alpha_2 \cdot 2^n + \alpha_3 \cdot 3^n$$

$$\alpha_0 = \alpha_1 + \alpha_2 + \alpha_3 = 2$$

$$\alpha_1 = \alpha_1 + 2\alpha_2 + 3\alpha_3 = 5 \quad \left. \right\}$$

$$\alpha_2 = \alpha_1 + 4\alpha_2 + 9\alpha_3 = 15 \quad \Rightarrow \alpha_1 = 1, \quad \alpha_2 = -1, \quad \alpha_3 = 2$$

$$\Rightarrow a_n = 1 - 2^n + 2 \cdot 3^n$$

# 4° CASO:

$$a_n = a_{n-1} + n$$

$$\text{con } a_1 = 1$$

Equazione omogenea associata:  $a_n = a_{n-1}$

Equazione caratteristica:  $r - 1 = 0 \Rightarrow r = 1$

$$\Rightarrow a_n^{(h)} = \alpha \cdot 1^n = \alpha$$

Soluzione particolare:  $a_n^{(p)} = cn + d \Rightarrow cn + d = c(n-1) + d + n \Rightarrow c = n$

Ma c deve essere una costante  $\Rightarrow a_n^{(p)} = cn^2 + dn + e$

$$\Rightarrow cn^2 + dn + e = c(n^2 - 2n + 1) + d(n-1) + e + n \Rightarrow c = \frac{1}{2}, \quad d = \frac{1}{2}$$

$$\Rightarrow a_n^{(p)} = \frac{1}{2}n^2 + \frac{1}{2}n + e$$

$$a_n = a_n^{(h)} + a_n^{(p)} = \alpha + \frac{1}{2}n^2 + \frac{1}{2}n + e$$

$$\alpha_1 = \alpha + \frac{1}{2} + \frac{1}{2} + e \Rightarrow \alpha + e = 0 \Rightarrow a_n = \frac{1}{2}n^2 + \frac{1}{2}n \Rightarrow a_n = \frac{n(n+1)}{2}$$

## BACKTRACKING:

Codice Python del problema delle otto regine:

```
def solve(n, i, a, b, c):
    # n = dimensione della scacchiera nonche' numero delle regine che devo posizionare
    # i = numero di colonne occupate nonche' numero di regine posizioante
    # a = lista con gli indici delle righe occupate
    # b = lista con gli indici delle diagonali occupate (/)
    # c = lista con gli indici delle diagonali occupate (\)

    if i < n:
        for j in range(n):
            if j not in a and i+j not in b and i-j not in c:
                for solution in solve(n, i+1, a+[j], b+[i+j], c+[i-j]):
                    yield solution
    else:
        yield a

for solution in solve(8, 0, [], [], []):
    print solution
```

ALGORITMO	$T(n)$ CASO OTTIMO	$T(n)$ CASO MEDIO	$T(n)$ CASO PESSIMO	$S(n)$ (INPUT ESCLUSO)	STABILE	IN-PLACE	ADATTATIVO
STUPID SORT	$O(n)$	$O(n \cdot n!)$	$+\infty$	$O(n)$	NO	NO	NO
SELECTION SORT	$O(n^2)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	SÍ	SÍ	NO
BUBBLE SORT	$O(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	SÍ	SÍ	SÍ
INSERTION SORT	$\Omega(n)$	$O(n^2)$	$O(n^2)$	$\Theta(1)$	SÍ	SÍ	SÍ
MERGE SORT	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$	SÍ	NO	NO
BUCKET SORT	$O(n)$	$O(n)$	$O(n^2)$	$\Theta(n)$	SÍ	NO	NO
QUICK SORT	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$\Theta(1)$	NO	SÍ	NO
RADIX SORT	$O(nK)$	$O(nK)$	$O(nK)$	$\Theta(n)$	NO	NO	NO

- STABILITÀ: un algoritmo di ordinamento è stabile se preserva l'ordine iniziale tra due elementi con la stessa chiave.
- ORDINAMENTO IN PLACE: si tratta di un algoritmo che non crea copie dell'input per generare la sequenza ordinata.
- ADATTATIVITÀ: un algoritmo di ordinamento è adattativo se trae vantaggio dagli elementi già ordinati.

## TEOREMA PRINCIPALE:

$$\text{Sia } T(n) = \begin{cases} aT(\frac{n}{b}) + cn^{\beta} & n > 1 \\ d & n \leq 1 \end{cases}$$

$a \in \mathbb{N}, a \geq 1$        $b \in \mathbb{N}, b \geq 2$   
 $c \in \mathbb{R}, c > 0$        $\beta \in \mathbb{R}, \beta > 0$

Posto  $\alpha := \log_b a$ , si ha che:

$$T(n) = \begin{cases} \Theta(n^\alpha) & \text{se } \alpha > \beta \\ \Theta(n^\alpha \log n) & \text{se } \alpha = \beta \\ \Theta(n^\beta) & \text{se } \alpha < \beta \end{cases}$$

Esempio:

$$T(n) = \begin{cases} 9T(\frac{n}{3}) + n & n > 1 \\ 1 & n \leq 1 \end{cases}$$

$$a=9, b=3, c=1, \beta=1, d=1, \alpha = \log_3 9 = 2 \Rightarrow \alpha > \beta \Rightarrow T(n) = \Theta(n^\alpha) = \boxed{\Theta(n^2)}$$

## RICORRENZE LINEARI DI ORDINE COSTANTE:

$$\text{Sia } T(n) = \begin{cases} \sum_{1 \leq i \leq h} a_i T(n-i) + cn^{\beta} & n > m \\ \Theta(1) & n \leq m \leq h \end{cases}$$

con  $a_1, a_2, \dots, a_h$  costanti intere non negative,  $h \in \mathbb{N}, h \geq 1$   
 $c \in \mathbb{R}, c > 0$        $\beta \in \mathbb{R}, \beta \geq 0$

Posto  $a := \sum_{1 \leq i \leq h} a_i$ , si ha che:

$$T(n) = \begin{cases} \Theta(n^{\beta+1}) & \text{se } a=1 \\ \Theta(a^n n^\beta) & \text{se } a \geq 2 \end{cases}$$

Esempio:

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

$$a=1+1=2, c=1, \beta=0, h=2 \Rightarrow a \geq 2 \Rightarrow T(n) = \Theta(a^n n^\beta) = \boxed{\Theta(2^n)}$$

## STUPID SORT

Data una sequenza A, verifica se essa è ordinata. Se non lo è, viene generata una sua permutazione casuale (oppure due dei suoi elementi vengono scambiati) e poi viene effettuato di nuovo il controllo.

## RADIX SORT

Utilizza un ordinamento per ciascuna cifra dei numeri, a partire da quella meno significativa.

## SELECTION SORT

Data una sequenza A, cerca il suo valore minimo ed eventualmente lo scambia con quello in prima posizione.

Così rimangono da ordinare gli altri  $n-1$  elementi, e si può applicare la stessa strategia sulla sottosequenza di A formata da tali  $n-1$  elementi.

CODICE PYTHON:

```
def selectionSort(A):
    for i in range(len(A)):
        min = i
        for j in range(i+1, len(A)):
            if A[j] < A[min]:
                min = j
        if min != i:
            tmp = A[i]
            A[i] = A[min]
            A[min] = tmp
```

## BUBBLE SORT

Confronta due elementi adiacenti alla volta e li inverte di posizione se sono nel l'ordine sbagliato. In questo modo i valori maggiori salgono velocemente verso le posizioni più alte della sequenza.

CODICE PYTHON:

```
def bubbleSort(A):  
    sorted = False  
    while sorted == False:  
        sorted = True  
        for i in range(len(A)-1):  
            if A[i] > A[i+1]:  
                tmp = A[i]  
                A[i] = A[i+1]  
                A[i+1] = tmp  
                sorted = False
```

## INSERTION SORT

A ogni iterazione, inserisce l'elemento  $A[i]$  nella posizione corretta nella sotto-sequenza già ordinata  $A[0, \dots, i-1]$ .

CODICE PYTHON:

```
def insertionSort(A):  
    for i in range(len(A)):  
        Key = A[i]  
        J = i - 1  
        while J >= 0 and A[J] > Key:  
            A[J + 1] = A[J]  
            J = J - 1  
        A[J + 1] = Key
```

## BUCKET SORT

un algoritmo suddiviso in più fasi:

- Genera un certo numero di bucket.
- SCATTER: scandisce il vettore e inserisce gli elementi in un bucket.
- ORDINA ciascun bucket non vuoto (tipicamente con l'insertion sort).
- + GATHER: estrae da ciascun bucket gli elementi e li inserisce nel vettore originale.

CODICE PYTHON:

```
def newList2(n):
```

```
    lis = []
```

```
    for i in range(n):
```

```
        lis.append([])
```

```
    return lis
```

```
def bucketSort(A, K):
```

```
    buckets = newList2(K)
```

```
    M = A[0]
```

```
    for h in range(1, len(A)):
```

```
        if A[h] > M:
```

```
            M = A[h]
```

```
M = M + 1
```

```
for i in range(len(A)):
```

```
buckets[(K * A[i]) / M].append(A[i])
```

```
for i in range(K):
```

```
    insertionSort(buckets[i])
```

```
B = []
```

```
for j in range(K):
```

```
    for h in range(len(buckets[j])):
```

```
B.append(buckets[j][h])
```

```
return B
```

## MERGE SORT

Utilizza la tecnica del Divide et Impera e opera in maniera ricorsiva.  
Idea di base: se la sequenza ha lunghezza 1  $\Rightarrow$  è già ordinata. Altrimenti:  
1- Si divide la sequenza in due metà.  
2- Ciascuna delle due sottosequenze viene ordinata ricorsivamente.  
3- Le due sottosequenze vengono fuse estraendo ripetutamente il minimo delle due sottosequenze.

CODICE PYTHON:

```
def newList(n, val):
```

```
    lis = []
```

```
    for i in range(n):
```

```
        lis.append(val)
```

```
    return lis
```

```
def merge(A, left, center, right):
```

```
    i = left
```

```
    j = center + 1
```

```
    k = left
```

```
    B = newList(right + 1, 0)
```

```
    while i <= center and j <= right:
```

```
        if A[i] <= A[j]:
```

```
            B[k] = A[i]
```

```
            i = i + 1
```

```
        else:
```

```
            B[k] = A[j]
```

```
            j = j + 1
```

```
        k = k + 1
```

```
    J = right
```

```
    for h in range(center, i - 1, -1):
```

```
        A[j] = A[h]
```

```
        j = j - 1
```

```
    for j in range(left, k):
```

```
        A[j] = B[j]
```

```
def mergeSort(A, left, right):
```

```
    if left < right:
```

```
        center = (left + right) / 2
```

```
        mergeSort(A, left, center)
```

```
        mergeSort(A, center + 1, right)
```

```
        merge(A, left, center, right)
```

# GORITMI SUGLI ALBERI

## SITE:

Visita in profondità in preordine:

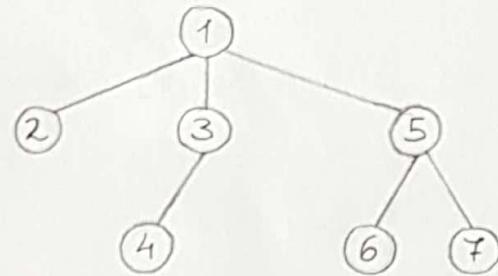
REORDER DFS(v):

if  $v \neq \perp$  then:

< do something with v >

PREORDER DFS(v.left)

PREORDER DFS(v.right)



→ Visita in profondità in ordine simmetrico:

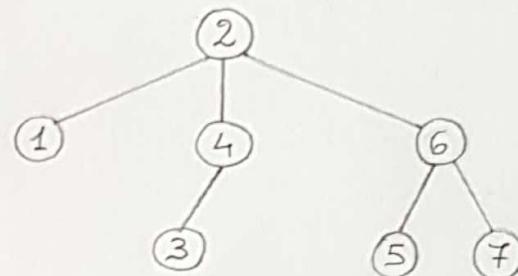
INORDER DFS(v):

if  $v \neq \perp$  then:

INORDER DFS(v.left)

< do something with v >

INORDER DFS(v.right)



→ Visita in profondità in postordine:

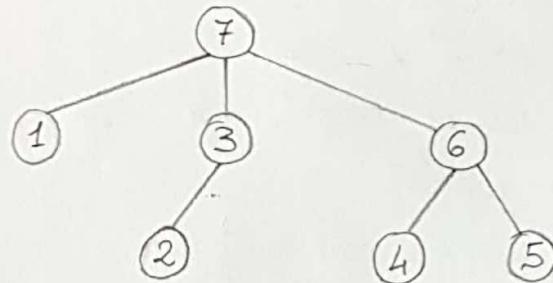
POSTORDER DFS(v):

if  $v \neq \perp$  then:

POSTORDER DFS(v.left)

POSTORDER DFS(v.right)

< do something with v >



→ Visita in ampiezza:

BFS(root):

$q \leftarrow \text{Queue}()$

$\text{node} \leftarrow \text{root}$

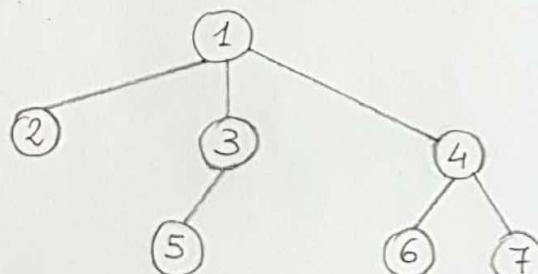
while  $\text{node} \neq \perp$ :

< do something with node >

for each child of node:

$q.\text{enqueue}(\text{child})$

$\text{node} \leftarrow q.\text{dequeue}()$



## QUICK SORT

Utilizza la tecnica del *Divide et Impera* ed è organizzato in tre fasi:

1- Seleziona un elemento (*pivot*) dal vettore.

2- Partiziona il vettore: tutti gli elementi più grandi del pivot vengono spostati a destra, tutti quelli più piccoli a sinistra.

3- Applica questa strategia ricorsivamente alla parte destra e sinistra.

CODICE PYTHON:

```
def partition(A, low, high):
    pivot = A[low + (high - low) / 2]
    i = low
    j = high
    while i <= j:
        while A[i] < pivot:
            i = i + 1
        while A[j] > pivot:
            j = j - 1
        if i >= j:
            return j
        tmp = A[i]
        A[i] = A[j]
        A[j] = tmp
```

```
def quickSort(A, low, high):
```

```
if low < high:
    p = partition(A, low, high)
    quickSort(A, low, p)
    quickSort(A, p + 1, high)
```

## ALBERI BINARI DI RICERCA:

→ Ricerca di un nodo:

TREE SEARCH ( $v$ , Key):

if  $v = \perp$  or  $\text{Key} = v.\text{Key}$  then:  
    return  $v$

if  $\text{Key} < v.\text{Key}$  then:

    return TREE SEARCH ( $v.\text{left}$ , Key)

return TREE SEARCH ( $v.\text{right}$ , Key)

→ Massimo e minimo:

TREE MAXIMUM ( $v$ ):

while  $v.\text{right} \neq \perp$ :  
     $v \leftarrow v.\text{right}$

return  $v$

TREE MINIMUM ( $v$ ):

while  $v.\text{left} \neq \perp$ :  
     $v \leftarrow v.\text{left}$

return  $v$

• Inserimento:

SEE INSERT ( $T, z$ ):

$y \leftarrow \perp$                        $\rightarrow$  GENITORE DI  $x$   
 $x \leftarrow T.\text{root}$                        $\rightarrow$  PUNTATORE DEL PERCORSO  
while  $x \neq \perp$ :  
   $y \leftarrow x$   
  if  $z.\text{Key} < x.\text{Key}$  then:  
     $x \leftarrow x.\text{left}$   
  else:  
     $x \leftarrow x.\text{right}$   
 $z.\text{parent} \leftarrow y$   
if  $y = \perp$  then:                       $\rightarrow$  ALBERO VUOTO  
   $T.\text{root} \leftarrow z$   
else if  $z.\text{Key} < y.\text{Key}$  then:  
   $y.\text{left} \leftarrow z$   
else:  
   $y.\text{right} \leftarrow z$

→ Eliminazione:

TREE DELETE ( $T, z$ ):

if  $z.\text{left} = \perp$  or  $z.\text{right} = \perp$  then:  
   $y \leftarrow z$   
else:  
   $y \leftarrow \text{TREE SUCCESSOR}(z)$   
if  $y.\text{left} \neq \perp$  then:  
   $x \leftarrow y.\text{left}$   
else:  
   $x \leftarrow y.\text{right}$   
 $x.\text{parent} \leftarrow y.\text{parent}$   
if  $y.\text{parent} = \perp$  then:  
   $T.\text{root} \leftarrow x$   
else if  $y = y.\text{parent}.\text{left}$  then:  
   $y.\text{parent}.\text{left} \leftarrow x$   
else:  
   $y.\text{parent}.\text{right} \leftarrow x$   
if  $y \neq z$  then:  
   $z.\text{Key} \leftarrow y.\text{Key}$   
return  $y$

## RED-BLACK TREE:

→ Rotazione a sinistra:

LEFT ROTATE ( $T, x$ ):

$y \leftarrow x.\text{right}$

$x.\text{right} \leftarrow y.\text{left}$

$y.\text{left.parent} \leftarrow x$

$y.\text{parent} \leftarrow x.\text{parent}$

if  $x.\text{parent} = \perp$  then:

$T.\text{root} \leftarrow y$

else if  $x = x.\text{parent.left}$  then:

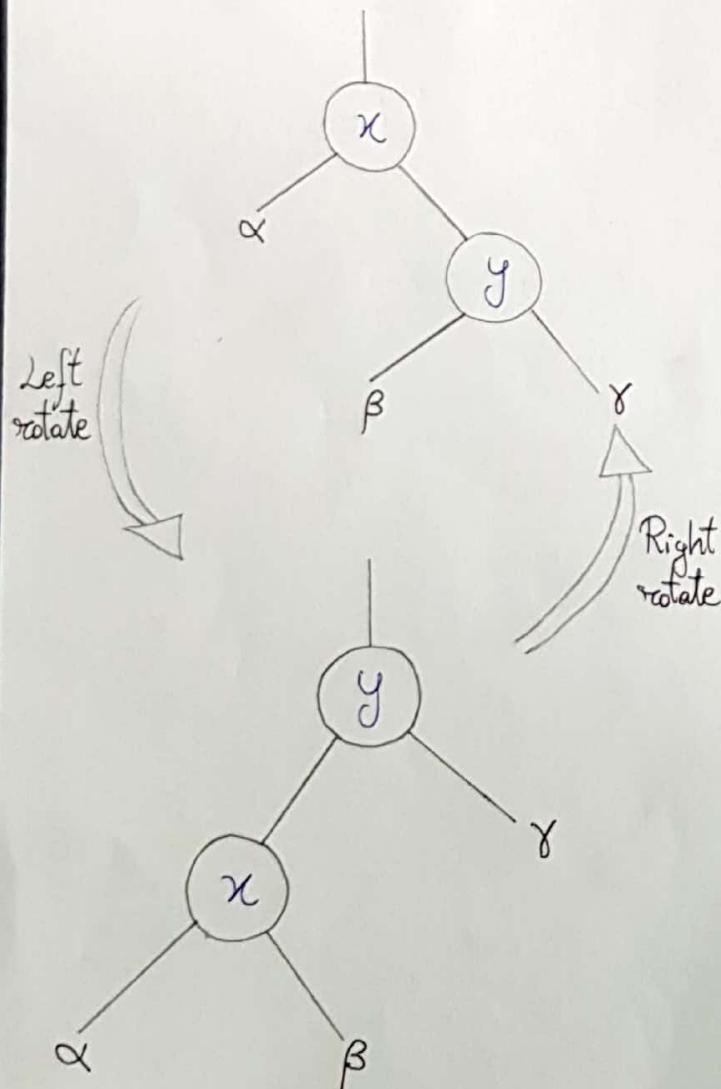
$x.\text{parent.left} \leftarrow y$

else:

$x.\text{parent.right} \leftarrow y$

$y.\text{left} \leftarrow x$

$x.\text{parent} \leftarrow y$



→ Inserimento:

RB-INSERT ( $T, z$ ):

$y \leftarrow \perp$

$x \leftarrow T.\text{root}$

while  $x \neq \perp$ :

$y \leftarrow x$

if  $z.\text{Key} < x.\text{Key}$  then:

$x \leftarrow x.\text{left}$

else:

$x \leftarrow x.\text{right}$

$z.\text{parent} \leftarrow y$

if  $y = \perp$  then:

$T.\text{root} \leftarrow z$

else if  $z.\text{Key} < y.\text{Key}$  then:

$y.\text{left} \leftarrow z$

else:

$y.\text{right} \leftarrow z$

$z.\text{left} \leftarrow \perp$

$z.\text{right} \leftarrow \perp$

$z.\text{color} \leftarrow \text{RED}$

RB-INSERTFixup ( $T, z$ )

LE LINEE DI CODICE SCRITTE IN ROSSO RAPPRESENTANO LA PARTE IN PIÙ RISPETTO ALLA FUNZIONE TREEINSERT() DEGLI ALBERI BINARI DI RICERCA.

→ Ribilanciamento:

RB-INSERT FIXUP ( $T, z$ ):

while  $z.parent.color = \text{RED}$ :

if  $z.parent = z.parent.parent.left$  then:

$y \leftarrow z.parent.parent.right$

if  $y.color = \text{RED}$  then:

$z.parent.color \leftarrow \text{BLACK}$

$y.color \leftarrow \text{BLACK}$

$z.parent.parent.color \leftarrow \text{RED}$

$z \leftarrow z.parent.parent$

I TRE CASI IN CUI OPERIAMO SU UN  
SOTTOALBERO SINISTRO

CASO 1

else if  $z = z.parent.right$  then:

$z \leftarrow z.parent$

LEFT ROTATE ( $T, z$ )

CASO 2

else:

$z.parent.color \leftarrow \text{BLACK}$

$z.parent.parent.color \leftarrow \text{RED}$

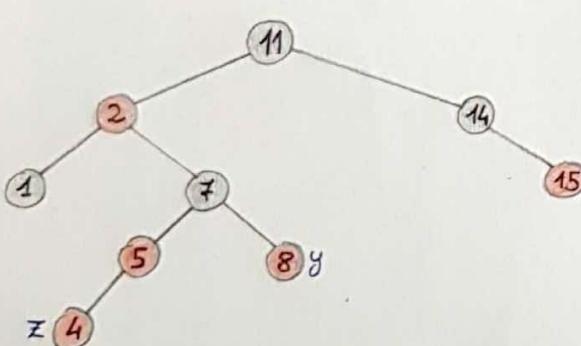
RIGHT ROTATE ( $T, z.parent.parent$ )

CASO 3

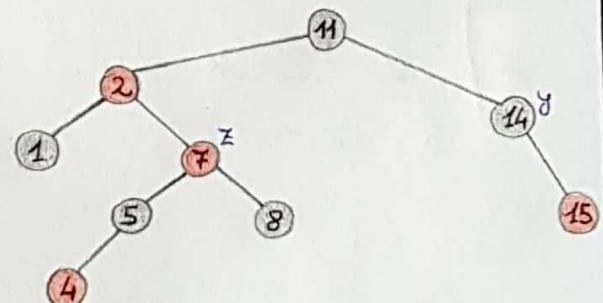
else:

< come prima, ma con "right" e "left" scambiati >

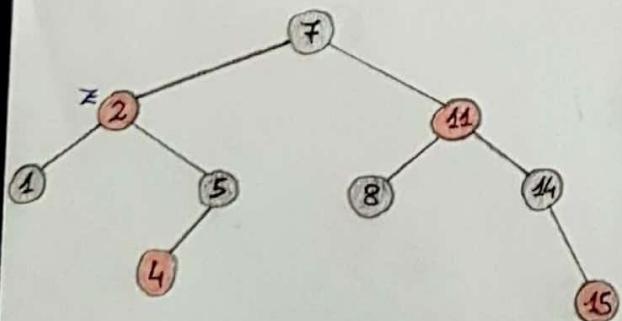
$T.root.color \leftarrow \text{BLACK}$



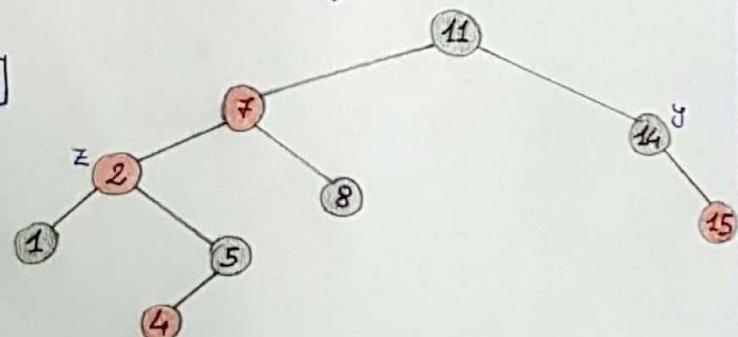
CASO 1



CASO 2



CASO 3



→ Eliminazione:

RB-DELETE ( $T, z$ ):

if  $z.\text{left} = \perp$  or  $z.\text{right} = \perp$  then:

$y \leftarrow z$

else:

$y \leftarrow \text{TREE SUCCESSOR}(z)$

if  $y.\text{left} \neq \perp$  then:

$x \leftarrow y.\text{left}$

else:

$x \leftarrow y.\text{right}$

$x.\text{parent} \leftarrow y.\text{parent}$

if  $y.\text{parent} = \perp$  then:

$T.\text{root} \leftarrow x$

else if  $y = y.\text{parent}.\text{left}$  then:

$y.\text{parent}.\text{left} \leftarrow x$

else:

$y.\text{parent}.\text{right} \leftarrow x$

if  $y \neq z$  then:

$z.\text{Key} \leftarrow y.\text{Key}$

if  $y.\text{color} = \text{BLACK}$  then:

RB-DELETEFIXUP( $T, x$ )

return  $y$

LE LINEE DI CODICE SCRITTE IN ROSSO  
RAPPRESENTANO LA PARTE IN PIÙ RISPETTO  
ALLA FUNZIONE TREE DELETE() DEGLI  
ALBERI BINARI DI RICERCA.

## HEAP:

→ Scambio:

EXCHANGE ( $A, i, j$ ):

$$tmp \leftarrow A[i]$$

$$A[i] \leftarrow A[j]$$

$$A[j] \leftarrow tmp$$

→ SCAMBIO TRA ELEMENTI DI UN VETTORE

EXCHANGEH<sup>H</sup>EAP ( $node_1, node_2$ ):

$$tmp \leftarrow node_1.Key$$

$$node_1.Key \leftarrow node_2.Key$$

$$node_2.Key \leftarrow tmp$$

→ SCAMBIO TRA CHIAVI DI NODI DI UN ALBERO

→ Mantenimento della proprietà di heap ( $\forall v \neq H.root \quad v.parent.Key > v.Key$ ) :

HEAPIFY ( $v$ ):

$$l \leftarrow v.left$$

$$r \leftarrow v.right$$

if  $l \neq \perp$  and  $l.Key > v.Key$  then:

$$largest \leftarrow l$$

else:

$$largest \leftarrow v.$$

if  $r \neq \perp$  and  $r.Key > largest.Key$  then:

$$largest \leftarrow r$$

if  $largest \neq v$  then:

EXCHANGEH<sup>H</sup>EAP ( $v, largest$ )

HEAPIFY ( $v$ )

CASO MaxHeap

→ Inserimento:

HEAP INSERT ( $H, Key$ ):

$$node \leftarrow Node(Key)$$

→ node = ELEMENTO DELLA CLASSE Node

LEAF INSERT ( $H, node$ )

while  $node.parent \neq \perp$  and  $node.parent.Key < node.Key$ :

EXCHANGEH<sup>H</sup>EAP ( $node, node.parent$ )

$$node \leftarrow node.parent$$

→ Rimozione dell'elemento radice (che è a priorità massima):

DELETE FIRST (H):

lastLeaf  $\leftarrow$  <get the last leaf in the heap>

ret  $\leftarrow$  H.root

H.root.Key  $\leftarrow$  lastLeaf.Key

HEAPIFY (H.root)

return ret

→ Algoritmo di Floyd:

→ L'ALGORITMO FUNZIONA GRAZIE ALLA RAPPRESENTAZIONE TRAMITE  
ARRAY DELL'ALBERO

ARRAYToHEAP (A):

for i  $\leftarrow$  len(A) - 1 down to 0:

HEAPIFY IN PLACE (A, i)

HEAPIFY IN PLACE (A, i):

if ISLEAF (A, i) then:

return

j  $\leftarrow$  GETMAXCHILDINDEX (A, i)

if A[i] < A[j] then:

EXCHANGE (A, i, j)

HEAPIFY IN PLACE (A, j)

→ Heap Sort:

HEAPSORT (A):

heap  $\leftarrow$  ARRAYTOHEAP (A) → COSTO: O(n)

for i  $\leftarrow$  len(A) - 1 down to 0: → n ITERAZIONI

max  $\leftarrow$  heap.root

DELETE FIRST (heap) → COSTO: O(log n) PER OGNI ITERAZIONE

A[i]  $\leftarrow$  max

STO: O(n) + O(n log n) = O(n log n)

## INSIEMI DISGIUNTI (RAPPRESENTATI COME ALBERI):

→ Creazione di un albero formato da un solo nodo (che sarà l'elemento rappresentativo).

MAKE SET ( $x$ ):

$$\begin{aligned}x.\text{parent} &\leftarrow x \\x.\text{rank} &\leftarrow 0\end{aligned}$$

→ Ricerca dell'elemento rappresentativo (radice) dell'albero che contiene il nodo  $x$ :

FIND SET ( $x$ ):

if  $x \neq x.\text{parent}$  then: → SI ASSUME CHE IL GENITORE DEL NODO RADICE SIA SE STESSO  
 $x.\text{parent} \leftarrow \text{FINDSET}(x.\text{parent})$   
return  $x.\text{parent}$

→ Unione dei due alberi contenenti rispettivamente i nodi  $x, y$ :

LINK ( $x, y$ ):

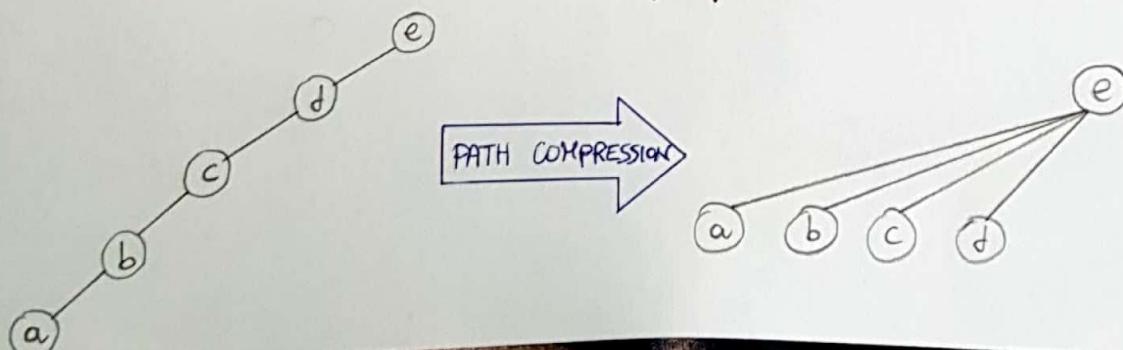
if  $x.\text{rank} > y.\text{rank}$  then:  
 $y.\text{parent} \leftarrow x$   
else:  
 $x.\text{parent} \leftarrow y$   
if  $x.\text{rank} = y.\text{rank}$  then:  
 $y.\text{rank} \leftarrow y.\text{rank} + 1$

UNION ( $x, y$ ):

LINK (FINDSET( $x$ ), FINDSET( $y$ ))

## Euristiche per risolvere il problema dello sbilanciamento:

- UNION BY RANK: tramite la funzione LINK(), far sì che la radice dell'albero con meno livelli punti alla radice dell'albero con più livelli.
- PATH COMPRESSION: si utilizza FINDSET() per far risalire i nodi verso la radice.



## ALGORITMI SUI GRAFI

→ Visita in ampiezza:

BFS ( $G, r$ ):

Queue  $Q \leftarrow \emptyset$

$Q.\text{enqueue}(r)$

foreach  $u$  in  $G.\text{vertices}() \setminus \{r\}$ :

$u.\text{discovered} \leftarrow \text{false}$

while not  $Q.\text{isEmpty}()$ :

Note  $u \leftarrow Q.\text{dequeue}()$

< visita il nodo  $u$  >

foreach  $v$  in  $G.\text{adj}(u)$ :

< visita l'arco  $(u, v)$  >

if not  $v.\text{discovered}$  then:

$v.\text{discovered} \leftarrow \text{true}$

$Q.\text{enqueue}(v)$

→ Visita in profondità:

DFS ( $G, r$ ):

Stack  $S \leftarrow \emptyset$

$S.\text{push}(r)$

foreach  $u$  in  $G.\text{vertices}()$ :

$u.\text{discovered} \leftarrow \text{false}$

while not  $S.\text{isEmpty}()$ :

Note  $u \leftarrow S.\text{pop}()$

if not  $u.\text{discovered}$  then:

< visita il nodo  $u$  >

$u.\text{discovered} \leftarrow \text{true}$

foreach  $v$  in  $G.\text{adj}(u)$ :

< visita l'arco  $(u, v)$  >

$S.\text{push}(v)$

→ Cammini minimi tra un nodo sorgente e tutti gli altri nodi (SSSP)  
Algoritmo di Dijkstra:

SSSP( $G, r$ ):

$r.\text{distance} \leftarrow 0$

MinHeap PQ  $\leftarrow \emptyset$

foreach  $v$  in  $G$ :

if  $v \neq r$  then:

$v.\text{distance} \leftarrow \infty$

$v.\text{parent} \leftarrow \perp$

PQ.enqueue( $v$ )

while PQ is not empty:

$u \leftarrow \text{PQ.getMin}()$

foreach  $v$  in  $G.\text{adj}(u)$ :

if  $v$  is in PQ then:

$\text{newDist} \leftarrow u.\text{distance} + w(u, v)$

if  $\text{newDist} < v.\text{distance}$  then:

$v.\text{distance} \leftarrow \text{newDist}$

$v.\text{parent} \leftarrow u$

PQ.decreasePriority( $v$ , newDist)

→ Verifica se un grafo non orientato ha cicli:

HAS CYCLES ( $G$ ):

foreach  $u$  in  $G.V()$ :  $\rightarrow G.V() = \text{insieme dei nodi di } G$   
 $u.\text{visited} \leftarrow \text{false}$

foreach  $u$  in  $G.V()$ :

if not  $u.\text{visited}$  then:

if HAS CYCLE ( $G, u, \perp$ ) then:  
return true

return false

HAS CYCLE ( $G, u, p$ ):

$u.\text{visited} \leftarrow \text{true}$

foreach  $v$  in  $G.\text{adj}(u) \setminus \{p\}$ :

if  $v.\text{visited}$  then:

return true

else if HAS CYCLE ( $G, v, u$ ) then:

return true

return false

→ Classificazione degli archi orientati:

global timestamp  $\leftarrow 0$

CLASSIFYEDGES ( $G, u$ ):

timestamp  $\leftarrow$  timestamp + 1

$u.\text{enterTime} \leftarrow$  timestamp

foreach  $v$  in  $G.\text{adj}(u)$ :

if  $v.\text{enterTime} = 0$  then:

< arco dell'albero >

CLASSIFYEDGES ( $G, v$ )

else if  $u.\text{enterTime} > v.\text{enterTime}$  and  $v.\text{exitTime} = 0$  then:

< arco all'interno >

else if  $u.\text{enterTime} < v.\text{enterTime}$  and  $v.\text{exitTime} \neq 0$  then:

< arco in avanti >

else:

< arco di attraversamento >

timestamp  $\leftarrow$  timestamp + 1

$u.\text{exitTime} \leftarrow$  timestamp

TEOREMA: un grafo orientato è aciclico  $\Leftrightarrow$  non esistono archi all'interno del grafo.



SARÀ QUESTA LA CONDIZIONE CHE SFRUTTEREHO PER IMPLEMENTARE UNO PSEUDOCODICE CHE VERIFICA SE UN GRAFO ORIENTATO HA CICLI.

→ Verifica se un grafo orientato ha cicli:  
global timestamp  $\leftarrow 0$

HAS CYCLES ( $G, u$ ):

timestamp  $\leftarrow \text{timestamp} + 1$

$u.\text{enterTime} \leftarrow \text{timestamp}$

foreach  $v$  in  $G.\text{adj}(u)$ :

if  $v.\text{enterTime} = 0$  then:

if HAS CYCLES ( $G, v$ ) then:

return true

else if  $u.\text{enterTime} > v.\text{enterTime}$  and  $v.\text{exitTime} = 0$  then:

return true

timestamp  $\leftarrow \text{timestamp} + 1$

$u.\text{exitTime} \leftarrow \text{timestamp}$

return false

→ Ordinamento topologico di un grafo orientato aciclico con DFS e stack:

TOPLOGICAL SORT ( $G$ ):

Stack  $S \leftarrow \emptyset$

foreach  $u$  in  $G.V()$ :

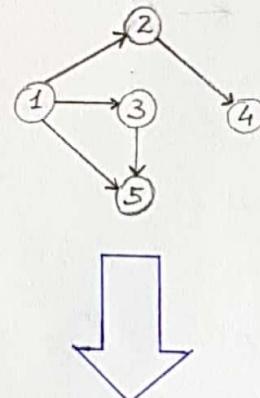
$u.\text{visited} \leftarrow \text{false}$

foreach  $u$  in  $G.V()$ :

if not  $u.\text{visited}$  then:

TOPLOGICAL SORT DFS ( $G, u, S$ )

return  $S$



TOPLOGICAL SORT DFS ( $G, u, S$ ):

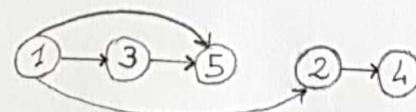
$u.\text{visited} \leftarrow \text{true}$

foreach  $v$  in  $G.\text{adj}(u)$ :

if not  $v.\text{visited}$  then:

TOPLOGICAL SORT DFS ( $G, v, S$ )

$S.\text{push}(u)$



→ Individuazione delle componenti connesse di una foresta:

ccDFS( $G$ ,  $id$ ,  $v$ ):

$v.id \leftarrow id$

foreach  $v$  in  $G.adj(u)$ :

if  $v.id = 0$  then:

ccDFS( $G$ ,  $id$ ,  $v$ )

cc( $G$ ):

foreach  $u$  in  $G.V()$ :

$u.id \leftarrow 0$  → SIGNIFICA CHE NON È STATA ASSEGNATA ANCORA ALCUNA COMPONENTE CONNESSA AL NODO  $u$

$id \leftarrow 0$  → VARIABILE CHE CONTA IL NUMERO DELLE DIVERSE COMPONENTI CONNESSE ESPLORATE

foreach  $u$  in  $G.V()$ :

if  $u.id = 0$  then:

$id \leftarrow id + 1$

ccDFS( $G$ ,  $id$ ,  $u$ )

→ Individuazione delle componenti fortemente connesse - Algoritmo di Kosaraju:

TRANSPOSE( $G$ ): → FUNZIONE CHE RESTITUISCE IL GRAFO TRASPOSTO DI  $G$

Graph  $G^T \leftarrow \emptyset$

foreach  $u$  in  $G$ :

$G^T.insertNode(u)$

foreach  $u$  in  $G$ :

foreach  $v$  in  $G.adj(u)$ :

$G^T.insertEdge(v, u)$

return  $G^T$

KOSARAJU( $G$ ):

Stack  $S \leftarrow$  TOPOLOGICAL SORT( $G$ )

$G^T \leftarrow$  TRANSPOSE( $G$ )

return cc( $G^T, S$ )

→ FUNZIONE SIMILE A cc( $G$ ), CON L'UNICA DIFFERENZA CHE L'ORDINE DEI NODI SU CUI ITERA È PRESTABILITO DALLO STACK

NEL CASO DELL'INDIVIDUAZIONE DELLE COMPONENTI CONNESSE, L'ORDINE CON CUI SI ITERA SUI NODI È FONDAMENTALE (MOTIVO PER CUI SI IMPLEMENTA L'ALGORITMO DI KOSARAJU)

## MINIMUM SPANNING TREE (MINIMO ALBERO RICOPRENTE):

→ Algoritmo di Boruvka:

- SIA A UN INSIEME DI COMPONENTI CONNESSE; INIZIALMENTE SI HA UNA COMPONENTE CONNESSA PER CIASCU  
NO.
- VIENE SELEZIONATA UNA COMPONENTE CONNESSA DI A.
- SI SELEZIONA L'ARCO A PESO MINIMO CHE CONNETTE LA COMPONENTE CON UN'ALTRA COMPONENTE DI A.
- SI RIMUOVONO GLI ARCHI CHE CONNETTONO DUE VERTICI APPARTENENTI ALLA STESSA COMPONENTE E NON IN
- SI RIPETONO QUESTI ULTIMI TRE STEP FINCHÉ NON SI OTTIENE UN'UNICA COMPONENTE CONNESSA.

→ Algoritmo di Kruskal:

KRUSKAL(G):

$$A \leftarrow \emptyset$$

foreach  $v$  in  $G.V()$ :

MAKESET( $v$ )

foreach  $(u,v)$  in  $G.E()$  ordinati per  $w(u,v)$  crescente:

if  $\text{FINDSET}(u) \neq \text{FINDSET}(v)$  then:

$$A \leftarrow A \cup \{(u,v)\}$$

UNION( $\text{FINDSET}(u), \text{FINDSET}(v)$ )

return  $A$

→ Algoritmo di Prim:

PRIM( $G, r$ ):

$$A = \{r\}$$

$$\text{MinHeap } PQ \leftarrow \emptyset$$

foreach  $v$  in  $G.V()$ :

$v.\text{distance} \leftarrow w(r, v)$  → VALE  $\infty$  SE  $r, v$  NON SONO ADIACENTI

$$v.\text{parent} \leftarrow \perp$$

PQ.enqueue( $v$ )

while PQ is not empty:

$$u \leftarrow PQ.\text{getMin}()$$

$(u, v) \leftarrow$  arco a minima distanza, con  $v \in A$

$$A \leftarrow A \cup u$$

$$u.\text{parent} \leftarrow v$$

foreach  $u$  in PQ tale che  $(u, v) \in E$ :

if  $u.\text{distance} > w(v, u)$  then:

PQ.updatePriority( $u, w(v, u)$ )

