

ANALISI

integer min(ITEM[] A, integer n)

```

ITEM min  $\leftarrow A[1]$ 
for integer i  $\leftarrow 2$  to n do
    if A[i] < min then
        min  $\leftarrow A[i]$ 
return min

```

	Costo	# Volte
<i>c</i> ₁	1	
<i>c</i> ₂	<i>n</i>	
<i>c</i> ₃	<i>n</i> - 1	
<i>c</i> ₄	<i>n</i> - 1	
<i>c</i> ₅	1	

ITEM binarySearch(ITEM[] A, ITEM *v*, integer *i*, integer *j*)

```

if i > j then
    return 0
else
    integer m  $\leftarrow \lfloor (i + j)/2 \rfloor$ 
    if A[m] = v then
        return m
    else if A[m] < v then
        return binarySearch(A, v, m + 1, j)
    else
        return binarySearch(A, v, i, m - 1)

```

	Costo	# (<i>i</i> > <i>j</i>)	# (<i>i</i> \leq <i>j</i>)
<i>c</i> ₁	1	1	
<i>c</i> ₂	1	0	
<i>c</i> ₃	0	1	
<i>c</i> ₄	0	1	
<i>c</i> ₅	0	0	
<i>c</i> ₆	0	1	
<i>c</i> ₇	$T(\lfloor (n-1)/2 \rfloor)$	0	0/1
<i>c</i> ₇	$T(\lfloor n/2 \rfloor)$	0	1/0

ITEM iterativeBinarySearch(ITEM[] A, ITEM *v*)

```

integer i  $\leftarrow 1$ 
integer j  $\leftarrow n$ 
integer m  $\leftarrow \lfloor (i + j)/2 \rfloor$ 
while i < j and A[m]  $\neq v$  do
    if A[m] < v then
        i  $\leftarrow m + 1$ 
    else
        j  $\leftarrow m - 1$ 
        m  $\leftarrow \lfloor (i + j)/2 \rfloor$ 
if i > j or A[m]  $\neq v$  then return 0 else return m

```

add(integer[] A, integer *k*)

```

integer i  $\leftarrow 0$ 
while i < k and A[i] = 1 do
    A[i]  $\leftarrow 0$ 
    i  $\leftarrow i + 1$ 
if i < k then A[i]  $\leftarrow 1$ 

```

Elemento minimo in un vettore:
O(n-1)

$$T(n) = c_1 + c_2n + c_3(n-1) + c_4(n-1) + c_5 = (c_2 + c_3 + c_4)n + (c_1 + c_5 - c_3 - c_4) = an + b$$

Ricerca elemento in un vettore:
O(logn)

	Parte SX	Parte DX	<i>n</i> pari	<i>n</i> dispari
	$\lfloor (n-1)/2 \rfloor$	$\lfloor n/2 \rfloor$	$n/2 - 1$	$(n-1)/2$
			$n/2$	$(n-1)/2$

$$i > j \quad (n=0) \quad T(n) = c_1 + c_2 = c$$

$$i \leq j \quad (n>0) \quad T(n) = T(n/2) + c_1 + c_3 + c_4 + c_6 + c_7 = T(n/2) + d$$

$$T(n) = \begin{cases} c & n=0 \\ T(n/2) + d & n>0 \end{cases}$$

Prodotto numeri binari D1:
O(n²)

- $X = a 2^{n/2} + b$
- $Y = c 2^{n/2} + d$
- $XY = ac 2^n + (ad+bc) 2^{n/2} + bd$

$$T(n) = \begin{cases} c_1 & n=1 \\ 4T(n/2) + c_2 \cdot n & n>1 \end{cases}$$

Prodotto numeri binari Gaussified (Karatsuba):
O(n^{log 3}) = O(n^{1.58})

- $A1 = ac$
- $A3 = bd$
- $m = (a+b)(c+d) = ac + ad + bc + bd$
- $A2 = m - A1 - A3 = ad + bc$

$$T(n) = \begin{cases} c_1 & n=1 \\ 3T(n/2) + c_2 \cdot n & n>1 \end{cases}$$

boolean [] KARATSUBA(boolean[] X, boolean[] Y, integer *n*)

```

if n = 1 then
    return X[1] · Y[1]
else
    spezza X in a; b e Y in c; d
    boolean[] A1  $\leftarrow$  KARATSUBA(a, c, n/2)
    boolean[] A3  $\leftarrow$  KARATSUBA(b, d, n/2)
    boolean[] m  $\leftarrow$  KARATSUBA(a + b, c + d, n/2)
    boolean[] A2  $\leftarrow$  m - A1 - A3
    return A1 ·  $2^n$  + A2 ·  $2^{n/2}$  + A3

```

boolean [] PDI(boolean[] X, boolean[] Y, integer *n*)

```

if n = 1 then
    return X[1] · Y[1]
else
    spezza X in a; b e Y in c; d
    return PDI(a, c, n/2) ·  $2^n$  + (PDI(a, d, n/2) + PDI(b, c, n/2)) ·  $2^{n/2}$  + PDI(b, d, n/2)

```

integer maxsum(integer[] A, integer *n*)

```

integer max  $\leftarrow 0$  % Massimo valore trovato
integer here  $\leftarrow 0$  % Massimo valore che termina nella posizione attuale
for i  $\leftarrow 1$  to n do
    here  $\leftarrow \max(\text{here} + A[i], 0)$ 
    max  $\leftarrow \max(\text{here}, m)$ 
return max

```

ordinaBandiera(integer[] *B*, integer *n*)

```

integer k  $\leftarrow 1$ 
integer j  $\leftarrow n$ 
while k  $\leq n$  and B[k] = VERDE do k  $\leftarrow k + 1$ 
while j  $\geq 1$  and B[j] = ROSSO do j  $\leftarrow j - 1$ 
integer i  $\leftarrow k$ 
while i  $\leq j$  do
    if B[i] = ROSSO then
        B[i]  $\leftrightarrow$  B[j]
        j  $\leftarrow j - 1$ 
    else if B[i] = VERDE then
        B[i]  $\leftrightarrow$  B[k]
        k  $\leftarrow k + 1$ 
    if B[i] = BIANCO then
        i  $\leftarrow i + 1$ 

```

2.10

SORT

selectionSort(ITEM[] A, integer n)

```

for integer  $i \leftarrow 1$  to n do
    integer  $j \leftarrow \min(A, i, n)$ 
     $A[i] \leftrightarrow A[j]$ 

    integer min(ITEM[] A, integer k, integer n)
        integer min  $\leftarrow k$                                 % Posizione del minimo parziale
        for integer  $h \leftarrow k + 1$  to n do
            if  $A[h] < A[min]$  then min  $\leftarrow h$           % Nuovo minimo parziale
        return min

```

$$\sum_{i=1}^n (n-i) = n(n-1)/2 = n^2/2 - n/2 = \Theta(n^2)$$

countingSort(ITEM[] A, integer n, integer k)

```

integer i, j, k
integer[] B  $\leftarrow$  new integer[1 ... k]
for i  $\leftarrow 1$  to k do B[i] = 0
for j  $\leftarrow 1$  to n do B[A[j]]  $\leftarrow$  B[A[j]] + 1
j  $\leftarrow 1$ 
for i  $\leftarrow 1$  to k do
    while B[i]  $>$  0 do
        A[j]  $\leftarrow$  i
        j  $\leftarrow$  j + 1
        B[i]  $\leftarrow$  B[i] - 1

```

Complessità: $O(n+k)$

Merge(ITEM A[], integer primo, integer ultimo, integer mezzo)

```

integer i, j, k, h
i  $\leftarrow$  primo; j  $\leftarrow$  mezzo + 1; k  $\leftarrow$  primo
while i  $\leq$  mezzo and j  $\leq$  ultimo do
    if A[i]  $\leq$  A[j] then
        B[k]  $\leftarrow$  A[i]
        i  $\leftarrow$  i + 1
    else
        B[k]  $\leftarrow$  A[j]
        j  $\leftarrow$  j + 1
    k  $\leftarrow$  k + 1
j  $\leftarrow$  ultimo
for h  $\leftarrow$  mezzo downto i do
    A[j]  $\leftarrow$  A[h]
    j  $\leftarrow$  j - 1
for j  $\leftarrow$  primo to k - 1 do A[j]  $\leftarrow$  B[j]

```

Complessità: $\Theta(n\log n)$

$$T(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + \Theta(n) & n > 1 \\ \Theta(1) & n = 1 \end{cases}$$

MergeSort(ITEM A[], integer primo, integer ultimo)

```

if primo  $<$  ultimo then
    integer mezzo  $\leftarrow \lfloor (primo + ultimo)/2 \rfloor$ 
    MergeSort(A, primo, mezzo)
    MergeSort(A, mezzo + 1, ultimo)
    Merge(A, primo, ultimo, mezzo)

```

STRUTTURE DI DATI

SEQUENCE

```

% Restituisce true se la sequenza è vuota
boolean empty()

% Restituisce true se p è uguale a pos_0 oppure a pos_{n+1}
boolean finished(POS p)

% Restituisce la posizione del primo elemento
POS head()

% Restituisce la posizione dell'ultimo elemento
POS tail()

% Restituisce la posizione dell'elemento che segue p
POS next(POS p)

% Restituisce la posizione dell'elemento che precede p
POS prev(POS p)

% Inserisce l'elemento v di tipo ITEM nella posizione p.
% Ritorna la nuova posizione, che diviene il predecessore di p
POS insert(POS p, ITEM v)

% Rimuove l'elemento contenuto nella posizione p.
% Ritorna il successore di p, che diviene successore del predecessore di p
POS remove(POS p)

% Legge l'elemento di tipo ITEM contenuto nella posizione p
ITEM read(POS p)

% Scrive l'elemento v di tipo ITEM nella posizione p
write(POS p, ITEM v)

```

SET

```

% Restituisce la cardinalità dell'insieme
integer size()

% Restituisce true se x è contenuto nell'insieme
boolean contains(ITEM x)

% Inserisce x nell'insieme, se non già presente
insert(ITEM x)

% Rimuove x dall'insieme, se presente
remove(ITEM x)

% Restituisce un nuovo insieme che è l'unione di A e B
SET union(SET A, SET B)

% Restituisce un nuovo insieme che è l'intersezione di A e B
SET intersection(SET A, SET B)

% Restituisce un nuovo insieme che è la differenza di A e B
SET difference(SET A, SET B)

```

DICTIONARY

```

% Restituisce il valore associato alla chiave k se presente, nil altrimenti
ITEM lookup(ITEM k)

% Associa il valore v alla chiave k
insert(ITEM k, ITEM v)

% Rimuove l'associazione della chiave k
remove(ITEM k)

```

stampadiDifferenze(SEQUENCE L)

```

POS  $p_1 \leftarrow L.\text{head}()$ 
POS  $p_2 \leftarrow L.\text{next}(p_1)$ 
POS  $p_3 \leftarrow L.\text{next}(p_2)$ 
while not  $L.\text{finished}(p_3)$  do
  if  $L.\text{read}(p_2) = L.\text{read}(p_1) - L.\text{read}(p_3)$  then print  $L.\text{read}(p_2)$ 
   $p_1 \leftarrow p_2$ 
   $p_2 \leftarrow p_3$ 
   $p_3 \leftarrow L.\text{next}(p_2)$ 

```

3.6

rango(SEQUENCE L, POS p)

```

POS  $q \leftarrow L.\text{next}(p)$ 
if not  $L.\text{finished}(q)$  then
   $rango(L, q)$ 
  integer  $a \leftarrow L.\text{read}(p) + L.\text{read}(q)$ 
   $L.\text{write}(p, a)$ 

```

3.8

STRUTTURE DI DATI ELEMENTARI

donGiovanni(SEQUENCE catalogo)

```

SEQUENCE  $L \leftarrow \text{Sequence}()$ 
POS  $q \leftarrow catalogo.\text{head}()$ 
while not  $catalogo.\text{finished}(q)$  do
  ITEM  $v \leftarrow catalogo.\text{read}(q)$ 
  if  $v.\text{nazione} = \text{spagnola}$  then  $L.\text{insert}(L.\text{tail}().\text{next}(), v)$ 
  if  $v.\text{rango} = \text{duchessa}$  then
     $| q \leftarrow catalogo.\text{remove}(q)$ 
  else
     $| q \leftarrow catalogo.\text{next}(q)$ 

```

STACK

ITEM[] A	% Elementi	boolean isEmpty()
integer n	% Cursore	<u>return</u> $n = 0$
integer m	% Dim. max	
STACK Stack(integer dim)		ITEM pop()
$ STACK t \leftarrow \text{new STACK}$		precondition: $n > 0$
$ t.A \leftarrow \text{new integer}[1 \dots dim]$		ITEM $t \leftarrow A[n]$
$ t.m \leftarrow dim$		$n \leftarrow n - 1$
$ t.n \leftarrow 0$		return t
return t		
ITEM top()	precondition: $n > 0$	push(ITEM v)
		precondition: $n < m$
		$n \leftarrow n + 1$
		$A[n] \leftarrow v$

% Restituisce true se la pila è vuota
integer isEmpty()

% Inserisce v in cima alla pila
push(ITEM v)

% Estrae l'elemento in cima alla pila e lo restituisce al chiamante
ITEM pop()

% Legge l'elemento in cima alla pila
ITEM top()

QUEUE

ITEM[] A	% Elementi	boolean isEmpty()
integer n	% Dim. attuale	<u>return</u> $n = 0$
integer $testa$	% Testa	
integer m	% Dim. max	ITEM dequeue()
QUEUE Queue(integer dim)		precondition: $n > 0$
$ QUEUE t \leftarrow \text{new QUEUE}$		ITEM $t \leftarrow A[testa]$
$ t.A \leftarrow \text{new integer}[0 \dots dim - 1]$		$testa \leftarrow (testa + 1) \bmod m$
$ t.m \leftarrow dim$		$n \leftarrow n - 1$
$ t.testa \leftarrow 0$		return t
$ t.n \leftarrow 0$		
return t		
ITEM top()	precondition: $n > 0$	enqueue(ITEM v)
		precondition: $n < m$
		$A[(testa + n) \bmod m] \leftarrow v$
		$n \leftarrow n + 1$

% Restituisce true se la coda è vuota
integer isEmpty()

% Inserisce v in fondo alla coda
enqueue(ITEM v)

% Estrae l'elemento in testa alla coda e lo restituisce al chiamante
ITEM dequeue()

% Legge l'elemento in testa alla coda
ITEM top()

Lista bidirezionale circolare con sentinella:

Costo operazioni O(1)

LIST

LIST $pred$	% Predecessore	boolean finished(Pos p)
LIST $succ$	% Successore	<u>return</u> ($p = \text{this}$)
ITEM $value$	% Elemento	
LIST List()		ITEM read(Pos p)
$ LIST t \leftarrow \text{new LIST}$		<u>return</u> $p.value$
$ t.pred \leftarrow t$		write(Pos p , ITEM v)
$ t.succ \leftarrow t$		<u>return</u> $p.value \leftarrow v$
return t		
boolean isEmpty()		Pos insert(Pos p , ITEM v)
<u>return</u> $pred = succ = \text{this}$		$ LIST t \leftarrow \text{List}()$
Pos head()		$ t.value \leftarrow v$
<u>return</u> $succ$		$ t.pred \leftarrow p.pred$
Pos tail()		$ p.pred.succ \leftarrow t$
<u>return</u> $pred$		$ t.succ \leftarrow p$
Pos next(Pos p)		$ p.pred \leftarrow t$
<u>return</u> $p.succ$		return t
Pos prev(Pos p)		
<u>return</u> $p.pred$		Pos remove(Pos p)
		$ p.pred.succ \leftarrow p.succ$
		$ p.succ.pred \leftarrow p.pred$
		$ LIST t \leftarrow p.succ$
		$ \text{delete } p$
		$ \text{return } t$

primiN(integer n , LIST L , LIST M)

```

if  $n \geq 1$  then
   $L.\text{insert}(L.\text{head}(), n)$ 
   $M.\text{insert}(M.\text{tail}().\text{next}(), n)$ 
  primiN( $n - 1$ ,  $L$ ,  $M$ )

```

4.1

epurazione(LIST L , LIST M)

```

POS  $p \leftarrow L.\text{head}()$ 
while not  $L.\text{finished}(p)$  do
   $conta \leftarrow 0$ 
  POS  $q \leftarrow L.\text{head}()$ 
  while not  $L.\text{finished}(q)$  do
    if  $L.\text{read}(q) = L.\text{read}(p)$  then
       $| conta \leftarrow conta + 1$ 
       $| q \leftarrow L.\text{next}(q)$ 
    if  $conta \neq 2$  then
       $| M.\text{insert}(M.\text{tail}().\text{next}(), L.\text{read}(p))$ 
     $p \leftarrow L.\text{next}(p)$ 

```

4.2

tangentopoli(LIST L)

```

POS  $p \leftarrow L.\text{head}()$ 
while not  $L.\text{empty}()$  do
  for  $i \leftarrow 1$  to  $k - 1$  do
     $| p \leftarrow L.\text{next}(p)$ 
    if  $L.\text{finished}(p)$  then  $p \leftarrow L.\text{head}()$ 
  print  $L.\text{read}(p)$ 
   $p \leftarrow L.\text{remove}(p)$ 

```

4.4

ALBERI

TREE

```
% Costruisce un nuovo albero, costituito da un solo nodo e contenente v
Tree(ITEM v)

% Legge il valore
ITEM read()

% Scrive v nel nodo
write(ITEM v)

% Restituisce il padre; nil se questo nodo è radice
TREE parent()

% Restituisce il primo figlio; nil se questo nodo è foglia
TREE leftmostChild()

% Restituisce il prossimo fratello del nodo a cui è applicato; nil se assente
TREE rightSibling()

% Inserisce il sottoalbero t come primo figlio di questo nodo
insertChild(TREE t)
  precondition: t.parent() = nil

% Inserisce il sottoalbero t come successivo fratello di questo nodo
insertSibling(TREE t)
  precondition: t.parent() = nil

% Distrugge il sottoalbero radicato nel primo figlio di questo nodo
deleteChild()

% Distrugge il sottoalbero radicato nel prossimo fratello di questo nodo
deleteSibling()
```

Realizzazione con vettore dei padri:

TREE

```
integer[] p
```

```
% Costruisce una "foresta" con n nodi isolati
Tree(integer n)
  p ← new integer[1...n]
  for i ← 1 to n do p[i] ← 0

% Restituisce il padre del nodo i; restituisce 0 se i è radice
integer parent(integer i)
  return p[i]

% Rende il nodo i un figlio del nodo j
setParent(integer i, integer j)
  p[i] ← j
```

depth(t)

```
integer max = 0
TREE u ← t.leftmostChild()
while u ≠ nil do
  integer t = depth(u) + 1
  if t > max then max ← t
  u ← u.rightSibling()
return max
```

5.1

visitaAmpiezza(TREE t)

```
integer larghezza ← 1
integer level ← 1
integer count ← 1
QUEUE Q ← Queue()
Q.enqueue(t)
t.level ← 0

while not Q.isEmpty() do
  TREE u ← Q.dequeue()
  if u.level ≠ level then
    level ← u.level
    count ← 0
  count ← count + 1
  if count > larghezza then larghezza ← count
  TREE v ← u.leftmostChild()
  while v ≠ nil do
    v.level ← u.level + 1
    Q.enqueue(v)
    v ← v.rightSibling()

return larghezza
```

5.3

TREE

```
NODE parent
NODE child
NODE sibling
ITEM value

Tree(ITEM v)
  TREE t ← new TREE
  t.value ← v
  t.parent ← t.child ← t.sibling ← nil
  return t

insertChild(TREE t)
  t.parent ← this
  t.sibling ← child
  child ← t

insertSibling(TREE t)
  t.parent ← this
  t.sibling ← sibling
  sibling ← t

deleteChild()
  NODE newChild ← child.rightSibling()
  delete(child)
  child ← newChild

deleteSibling()
  NODE newBrother ← sibling.rightSibling()
  delete(sibling)
  sibling ← newBrother

delete(TREE t)
  NODE u ← t.leftmostChild()
  while u ≠ nil do
    TREE next ← u.rightSibling()
    delete(u)
    u ← next
  delete t
```

visitaProfondità(TREE t)

precondition: t ≠ nil

(1) esame "anticipato" del nodo radice di t

```
TREE u ← t.leftmostChild()
while u ≠ nil do
  visitaProfondità(u)
  u ← u.rightSibling()
```

(2) esame "posticipato" del nodo radice di t

F B A D C E G I H

invisita(TREE t)

precondition: t ≠ nil

```
TREE u ← t.leftmostChild()
integer k ← 0
while u ≠ nil and k < i do
  k ← k + 1
  invisita(u)
  u ← u.rightSibling()
esame "simmetrico" del nodo t
while u ≠ nil do
  invisita(u)
  u ← u.rightSibling()
```

A B C D E F G H I

visitaProfondità(TREE t)

precondition: t ≠ nil

(1) esame "anticipato" del nodo radice di t

```
TREE u ← t.leftmostChild()
while u ≠ nil do
  visitaProfondità(u)
  u ← u.rightSibling()
```

(2) esame "posticipato" del nodo radice di t

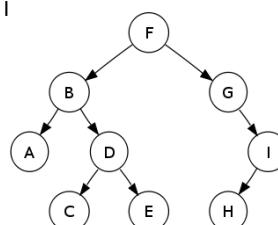
A C E D B H I G F

visitaAmpiezza(TREE t)

precondition: t ≠ nil

```
QUEUE Q ← Queue()
Q.enqueue(t)
while not Q.isEmpty() do
  TREE u ← Q.dequeue()
  esame "per livelli" del nodo u
  u ← u.leftmostChild()
  while u ≠ nil do
    Q.enqueue(u)
    u ← u.rightSibling()
```

F B G A D I C E H



Complessità: O(n)

```

boolean leafK(TREE t, integer k, integer somma)
    somma ← somma + t.read()
    if somma = k and t.leftmostChild() = nil then
        return true
    else
        while t.leftmostChild() ≠ nil and leafK(t.leftmostChild(), k, somma) do
            t.deleteChild()
        TREE u ← t.leftmostChild()                                % Nodo da cui cancellare
        TREE v ← if(u ≠ nil, u.rightSibling(), nil);           % Potenziale nodo da cancellare
        while v ≠ nil do
            if leafK(v, k, somma) then
                v.deleteSibling()                               % u rimane fisso e un fratello viene eliminato
            else
                u ← v                                         % u avanza
            v ← u.rightSibling()                            % Prossimo fratello
        return false

```

5.2

ALBERI BINARI

TREE

```

Tree(ITEM v)
    TREE t = new TREE
    t.parent ← nil
    t.left ← t.right ← nil
    t.value ← v
    return t

insertLeft(TREE T)
    T.parent ← this
    left ← T

insertRight(TREE T)
    T.parent ← this
    right ← T

% Restituisce il figlio sinistro (destro) di questo nodo; restituisce nil se assente
TREE left()
TREE right()

% Inserisce il sottoalbero t come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
    precondition: t.parent = nil
insertRight(TREE t)
    precondition: t.parent = nil

% Distrugge il sottoalbero sinistro (destro) di questo nodo
deleteLeft()
deleteRight()

```

visitaProfondità(TREE t)

```

if t ≠ nil then
    (1) esame “anticipato” del nodo radice di t
        visitaProfondità(t.left())
    (2) esame “simmetrico” del nodo radice di t
        visitaProfondità(t.right())
    (3) esame “posticipato” del nodo radice di t

```

previsita-iterativa(TREE t)

```

precondition: t ≠ nil
STACK S ← Stack()
S.push(t)
while not S.isEmpty() do
    TREE u ← S.pop()
    visita del nodo u
    if u.right ≠ nil then S.push(u.right)
    if u.left ≠ nil then S.push(u.left)

```

integer altezza-minimale(TREE T)

```

if T = nil then
    return +∞
if T.left() = nil and T.right() = nil then
    return 0
hl ← altezza-minimale(T.left())
hr ← altezza-minimale(T.right())
return min(hl, hr) + 1

```

```

deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        delete left
        left ← nil

deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        delete right
        right ← nil

```

% Restituisce il figlio sinistro (destro) di questo nodo; restituisce nil se assente
TREE left()
TREE right()
% Inserisce il sottoalbero t come figlio sinistro (destro) di questo nodo
insertLeft(TREE t)
 precondition: t.parent = nil
insertRight(TREE t)
 precondition: t.parent = nil
% Distrugge il sottoalbero sinistro (destro) di questo nodo
deleteLeft()
deleteRight()

integer count(TREE t)

```

if t = nil then return 0
else
    integer somma ← 1 + count(t.left()) + count(t.right())
    t.write(somma)
    return somma

```

5.5

deleteLeaf(TREE t)

```

if t ≠ nil then
    TREE u ← t.left()
    if u ≠ nil and u.left() = u.right() = nil and t.read() = u.read() then
        t.deleteLeft()
    deleteLeaf(t.left())
    deleteLeaf(t.right())

```

5.6

inserisciFoglia(TREE t)

```

if t ≠ nil then
    inserisciFoglia(t.left())
    inserisciFoglia(t.right())
    if t.left() = t.right() = nil then
        TREE new ← Tree(0)
        t.insertLeft(new)

```

5.7

addChild(TREE t, integer v)

```

if t = nil then
    return
v ← v + t.value
if t.left = t.right = nil then
    t.insertLeft(v)
else
    addChild(t.left, v)
    addChild(t.right, v)

```

5.10

inverti(TREE t)

```

if t = nil then
    return
t.left ↔ t.right
inverti(t.left())
inverti(t.right())

```

5.9

invisita-iterativa(TREE t)

```

precondition: t ≠ nil
STACK S ← Stack()
S.push((t, false))
while not S.isEmpty() do
    (u, f) ← S.pop()
    if f then
        | visita del nodo u
    else
        S.push((u, true))
        if u.right ≠ nil then S.push((u.right, false))
        if u.left ≠ nil then S.push((u.left, false))

```

5.13

postvisita-iterativa(TREE t)

```

precondition: t ≠ nil
STACK S ← Stack()
S.push((t, false))
while not S.isEmpty() do
    (u, f) ← S.pop()
    if f then
        | visita del nodo u
    else
        S.push((u, true))
        if u.right ≠ nil then S.push((u.right, false))
        if u.left ≠ nil then S.push((u.left, false))

```

integer alberipieni(integer n)

```

if n è pari then
    | return 0
else
    integer[] D ← new integer[1 ... n]
    D[1] ← 1
    for integer i = 3 to n step 2 do
        D[i] ← 0
        for integer j = 1 to i - 2 step 2 do
            | D[i] ← D[i] + D[j] · D[(n - 1) - j]
    return D[n]

```

5.9

5.7

```

integer countK(TREE v)
  s  $\leftarrow$  0
  visita(v)
    integer visita(TREE v)
      integer L  $\leftarrow$  iif(v.left = nil, -1, visita(v.left))
      integer R  $\leftarrow$  iif(v.right = nil, -1, visita(v.right))
      integer h  $\leftarrow$  max(L, R) + 1
      if h = k then
        _ s  $\leftarrow$  s + 1
      return h
  return s

```

5.10

```

integer lunghezzaCammino(TREE t, integer v)
  if t = nil then
    _ return 0
  return v + lunghezzaCammino(t.left, v + 1) + lunghezzaCammino(t.right, v + 1)

```

5.11

```

integer depth(TREE v, integer k)
  if t = nil then return 0
  if k = 0 then return 1
  return depth(T.left(), k - 1) + depth(T.right(), k - 1)

```

5.12

ALBERI BILANCIATI DI RICERCA

```

TREE lookupNode(TREE T, ITEM x)
  while T  $\neq$  nil and T.key  $\neq$  x do
    _ T  $\leftarrow$  iif(x < T.key, T.left, T.right)
  return T

link(TREE p, TREE u, ITEM x)
  if u  $\neq$  nil then u.parent  $\leftarrow$  p
  if p  $\neq$  nil then
    _ if x < p.key then p.left  $\leftarrow$  u
    _ else p.right  $\leftarrow$  u

TREE min(TREE T)
  while T.left  $\neq$  nil do
    _ T  $\leftarrow$  T.left
  return T

TREE max(TREE T)
  while T.right  $\neq$  nil do
    _ T  $\leftarrow$  T.right
  return T

```

```

TREE successorNode(TREE t)
  if t = nil then
    _ return t
  if t.right  $\neq$  nil then
    _ return min(t.right())
  TREE p  $\leftarrow$  t.parent
  while p  $\neq$  nil and t = p.right do
    _ t  $\leftarrow$  p
    _ p  $\leftarrow$  p.parent
  return p

```

```

TREE predecessorNode(TREE t)
  if t = nil then
    _ return t
  if t.left  $\neq$  nil then
    _ return max(t.left())
  TREE p  $\leftarrow$  t.parent
  while p  $\neq$  nil and t = p.left do
    _ t  $\leftarrow$  p
    _ p  $\leftarrow$  p.parent
  return p

```

Complessità: O(n)

```

stampaNodi (TREE t, integer A, integer B)
  integer v  $\leftarrow$  t.read()
  if t.left()  $\neq$  nil and A < v then stampaNodi(t.left(), A, B)
  if A  $\leq$  v and v  $\leq$  B then print v
  if t.right()  $\neq$  nil and B > v then stampaNodi(t.right(), A, B)

```

6.4

```

boolean verifyABR(TREE t)
  if t = nil then
    _ return true
  if (t.left  $\neq$  nil and t.value < t.left.value) or (t.right  $\neq$  nil and t.value > t.right.value) then
    _ return false
  return verifyABR(t.left) and verifyABR(t.right)

```

6.3

```

TREE insertNode(TREE T, ITEM x, ITEM v)
  TREE p  $\leftarrow$  nil
  TREE u  $\leftarrow$  T
  while u  $\neq$  nil and u.key  $\neq$  x do
    _ p  $\leftarrow$  u
    _ u  $\leftarrow$  iif(x < u.key, u.left, u.right)
  if u  $\neq$  nil and u.key = x then
    _ u.value  $\leftarrow$  v
  else
    _ TREE n  $\leftarrow$  Tree(x, v)
    _ link(p, n, x)
    _ if p = nil then return n
  return T

```

% Primo nodo ad essere inserito
% Ritorna albero non modificato

```

TREE removeNode(TREE T, ITEM x)
  TREE u  $\leftarrow$  lookupNode(T, x)
  if u  $\neq$  nil then
    if u.left  $\neq$  nil and u.right  $\neq$  nil then
      TREE s  $\leftarrow$  u.right
      while s.left  $\neq$  nil do s  $\leftarrow$  s.left
      u.key  $\leftarrow$  s.key
      u.value  $\leftarrow$  s.value
      u  $\leftarrow$  s
    TREE t
    if u.left  $\neq$  nil and u.right = nil then
      _ t  $\leftarrow$  u.left
    else
      _ t  $\leftarrow$  u.right
      link(u.parent, t, x)
    if u.parent = nil then T = t
    delete u
  return T

```

% Caso (2) - Solo figlio sx
% Caso (2) - Solo figlio dx / Caso (1)
% Caso (3)

Complessità: O(h) \rightarrow O(n) se sbilanciato
O(logn) per alberi Red-Black

```

TREE insertNodeRecursive(TREE T, TREE p, ITEM x, ITEM v)
  if T = nil then
    _ TREE n  $\leftarrow$  Tree(x, v)
    _ link(p, n, x)
    _ if p = nil then return n
  else
    if x < T.key then
      _ insertNodeRecursive(T.left, T, x, v)
    else if x > T.key then
      _ insertNodeRecursive(T.right, T, x, v)
    else
      _ T.value  $\leftarrow$  v
  return T

```

6.2

```

invisita(TREE t)
  TREE u  $\leftarrow$  t.min()
  while u  $\neq$  nil do
    {esamina il nodo u}
    u  $\leftarrow$  u.successorNode()

```

6.5

```
concatenate(TREE  $T_1$ , TREE  $T_2$ )
TREE  $v \leftarrow \max(T_1)$ 
link( $v, T_2, T_2.value$ )
```

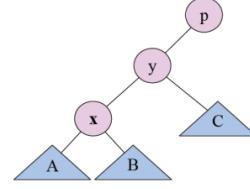
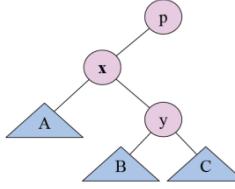
6.6

```
mindist(TREE  $t$ )
TREE  $u \leftarrow t.\min()$ 
integer  $\min \leftarrow +\infty$ 
integer  $\text{prev} \leftarrow -\infty$ 
while  $u \neq \text{nil}$  do
  if  $u.value - \text{prev} < \min$  then
     $\min \leftarrow u.value - \text{prev}$ 
     $u \leftarrow u.\text{successorNode}()$ 
return  $\min$ 
```

6.9

TREE rotateLeft(TREE x)

```
TREE  $y \leftarrow x.\text{right}$ 
TREE  $p \leftarrow x.\text{parent}$ 
(1)  $x.\text{right} \leftarrow y.\text{left}$  % Il sottoalbero  $B$  diventa figlio destro di  $x$ 
(1) if  $y.\text{left} \neq \text{nil}$  then  $y.\text{left}.\text{parent} \leftarrow x$ 
(2)  $y.\text{left} \leftarrow x$  %  $x$  diventa figlio sinistro di  $y$ 
(2)  $x.\text{parent} \leftarrow y$ 
(3)  $y.\text{parent} \leftarrow p$  %  $y$  diventa figlio di  $p$ 
(3) if  $p \neq \text{nil}$  then
  if  $p.\text{left} = x$  then  $p.\text{left} \leftarrow y$  else  $p.\text{right} \leftarrow y$ 
return  $y$ 
```



balanceInsert(TREE t)

```
 $t.\text{color} \leftarrow \text{RED}$ 
while  $t \neq \text{nil}$  do
  TREE  $p \leftarrow t.\text{parent}$ 
  TREE  $n \leftarrow \text{if}(p \neq \text{nil}, p.\text{parent}, \text{nil})$ 
  TREE  $z \leftarrow \text{if}(n = \text{nil}, \text{nil}, \text{if}(n.\text{left} = p, n.\text{right}, n.\text{left}))$ 
  if  $p = \text{nil}$  then
     $t.\text{color} \leftarrow \text{BLACK}$ 
     $t \leftarrow \text{nil}$ 
  else if  $p.\text{color} = \text{BLACK}$  then
     $t \leftarrow \text{nil}$ 
  else if  $z.\text{color} = \text{RED}$  then
     $p.\text{color} \leftarrow z.\text{color} \leftarrow \text{BLACK}$ 
     $n.\text{color} \leftarrow \text{RED}$ 
     $t \leftarrow n$ 
  else
    if  $(t = p.\text{right}) \text{ and } (p = n.\text{left})$  then
      rotateLeft( $p$ )
       $t \leftarrow p$ 
    else if  $(t = p.\text{left}) \text{ and } (p = n.\text{right})$  then
      rotateRight( $p$ )
       $t \leftarrow p$ 
    else
      if  $(t = p.\text{left}) \text{ and } (p = n.\text{left})$  then
        rotateRight( $n$ )
      else if  $(t = p.\text{right}) \text{ and } (p = n.\text{right})$  then
        rotateLeft( $n$ )
       $p.\text{color} \leftarrow \text{BLACK}$ 
       $n.\text{color} \leftarrow \text{RED}$ 
       $t \leftarrow \text{nil}$ 
```

% Padre

% Nonno

% Zio

% Caso (1)

% Caso (2)

% Caso (3)

% Caso (4.a)

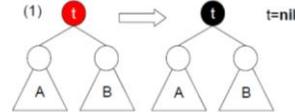
% Caso (4.b)

% Caso (5.a)

% Caso (5.b)

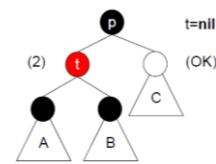
+ Caso 1:

- Nuovo nodo t non ha padre
- Primo nodo ad essere inserito o siamo risaliti fino alla radice
- Si colora t di nero



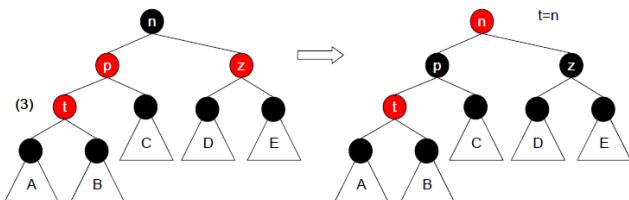
+ Caso 2

- Padre p di t è nero
- Nessun vincolo violato

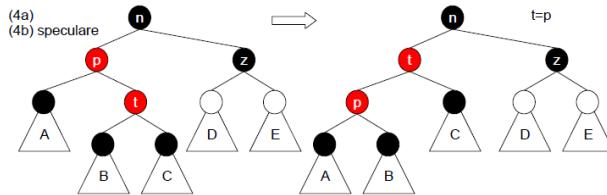


+ Caso 3

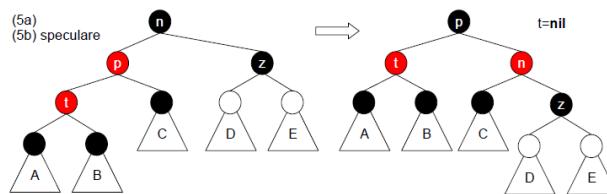
- Se z è rosso, è possibile colorare di nero p, z , e di rosso n .
- Poiché tutti i cammini che passano per z e p passano per n , la lunghezza dei cammini neri non è cambiata.
- Il problema può essere ora sul nonno:
 - violato vincolo (1), ovvero n può essere una radice rossa
 - violato vincolo (3), ovvero n rosso può avere un padre rosso.
 - Poniamo $t = n$, e il ciclo continua.



- Caso 4a,4b
 - Si assume che t sia figlio destro di p e che p sia figlio sinistro di n
 - t rosso
 - p rosso
 - z nero
 - Una rotazione a sinistra a partire dal nodo p scambia i ruoli di t e p ottenendo il caso (5a), dove i nodi rossi in conflitto sul vincolo (3) sono entrambi figli sinistri dei loro padri
 - I nodi coinvolti nel cambiamento sono p e t , entrambi rossi, quindi la lunghezza dei cammini neri non cambia



- Caso 5a,5b
 - Si assume che t sia figlio sinistro di p e p sia figlio sinistro di n
 - t rosso
 - p rosso
 - z nero
 - Una rotazione a destra a partire da n ci porta ad una situazione in cui t e n sono figli di p
 - Colorando n di rosso e p di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati
 - in particolare, la lunghezza dei cammini neri che passano per la radice è uguale alla situazione iniziale



balanceDelete(TREE T , TREE t)

```

while  $t \neq T$  and  $t.color = \text{BLACK}$  do
  TREE  $p \leftarrow t.parent$ 
  if  $t = p.left()$  then
    TREE  $f \leftarrow p.right$ 
    TREE  $ns \leftarrow f.left$ 
    TREE  $nd \leftarrow f.right$ 
    if  $f.color = \text{RED}$  then
       $p.color \leftarrow \text{RED}$ 
       $f.color \leftarrow \text{BLACK}$ 
      rotateLeft( $p$ )
      %  $t$  viene lasciato inalterato, quindi si ricade nei casi 2,3,4
    else
      if  $ns.color = nd.color = \text{BLACK}$  then
         $f.color \leftarrow \text{RED}$ 
         $t \leftarrow p$ 
      else if  $ns.color = \text{RED}$  and  $nd.color = \text{BLACK}$  then
         $ns.color \leftarrow \text{BLACK}$ 
         $f.color \leftarrow \text{RED}$ 
        rotateRight( $f$ )
        %  $t$  viene lasciato inalterato, quindi si ricade nel caso 4
      else if  $nd.color = \text{RED}$  then
         $f.color \leftarrow p.color$ 
         $p.color \leftarrow \text{BLACK}$ 
         $nd.color \leftarrow \text{BLACK}$ 
        rotateLeft( $p$ )
         $t \leftarrow \text{nil}$ 
      else
        % Casi (5)-(8) speculari a (1)-(4)
    if  $t \neq \text{nil}$  then  $t.color \leftarrow \text{BLACK}$ 
  end
end

```

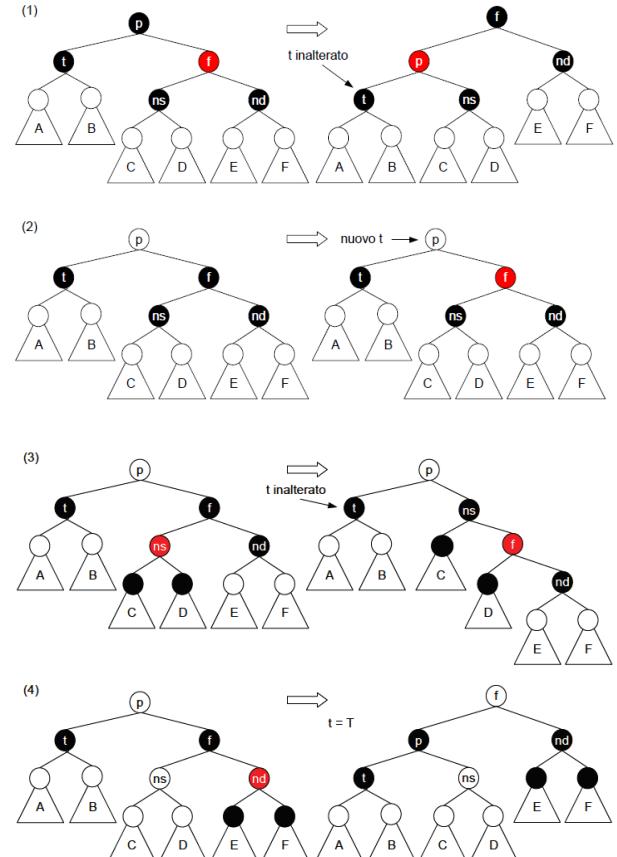


TABELLE HASH

HASH	
ITEM[] A	% Tabella delle chiavi
ITEM[] V	% Tabella dei valori
integer m	% Dimensione della tabella
HASH Hash(integer capacità)	
HASH t = new HASH	
t.m ← capacità	
t.A ← new Item[0...t.m - 1]	
t.V ← new Item[0...t.m - 1]	
for i ← 0 to t.m - 1 do t.A[i] ← nil	
return t	
integer scan(ITEM k, boolean insert)	
integer c ← m	% Prima posizione deleted
integer i ← 0	% Numero di ispezione
integer j ← H(k)	% Posizione attuale
while A[j] ≠ k and A[j] ≠ nil and i < m do	
if A[j] = deleted and c = m then c ← j	
j ← (j + H'(k)) mod m	
i ← i + 1	
if insert and A[j] ≠ k and c < m then j = c	
return j	
ITEM lookup(ITEM k)	
integer i ← scan(k, false)	
if A[i] = k then	
return V[i]	
else	
return nil	
insert(ITEM k, ITEM v)	
integer i ← scan(k, true)	
if A[i] = nil or A[i] = deleted or A[i] = k then	
A[i] ← k	
V[i] ← v	
else	
% Errore: tabella hash piena	
remove(ITEM k)	
integer i ← scan(k, false)	
if A[i] = k then	
A[i] ← deleted	

INSIEMI E DIZIONARI

SET (vettore booleano)	remove(integer x)	SET difference(SET A, SET B)
boolean[] V	if V[x] then	C ← Set()
integer dim	dim ← dim - 1	p ← A.head()
integer capacità	V[x] ← false	r ← C.head()
SET Set(integer N)	SET union(SET A, SET B)	while not A.finished(p) do
SET t ← new SET	C ← Set(max(A.capacità, B.capacità))	if not B.contains(A.read(p)) then
t.dim ← 0	for integer i ← 1 to A.capacità do	C.insert(A.read(p), r)
t.capacità ← N	if A.contains(i) then C.insert(i)	p ← A.next(p)
t.V ← new boolean[1...N]	for i ← 1 to B.capacità do	return C
for integer i ← 1 to N do V[i] ← false	if B.contains(i) then C.insert(i)	
return t		
integer size()	SET intersection(SET A, SET B)	
return dim	C ← Set(min(A.capacità, B.capacità))	
boolean contains(integer x)	for integer i ← 1 to min(A.capacità, B.capacità) do	
return V[x]	if A.contains(i) and B.contains(i) then C.insert(i)	
insert(integer x)	SET difference(SET A, SET B)	
if not V[x] then	C ← Set(A.capacità)	
dim ← dim + 1	for integer i ← 1 to A.capacità do	
V[x] ← true	if A.contains(i) and (i > B.capacità or not B.contains(i)) then	
	C.insert(i)	

Realizzazione con liste ordinate:

SET intersection(SET A, SET B)	SET union(SET A, SET B)	SET difference(SET A, SET B)
C ← Set() p ← A.head() q ← B.head() r ← C.head() while not A.finished(p) and not B.finished(q) do if A.read(p) = B.read(q) then C.insert(A.read(p), r) p ← A.next(p) q ← B.next(q) else if A.read(p) < B.read(q) then p ← A.next(p) else q ← B.next(q) return C	C ← Set() p ← A.head() r ← B.head() r ← C.head() while not A.finished(p) and not B.finished(q) do if A.read(p) = B.read(q) then C.insert(A.read(p), r) p ← A.next(p) q ← B.next(q) else if A.read(p) < B.read(q) then C.insert(A.read(p), r) p ← A.next(p) else C.insert(B.read(q), r) q ← B.next(q) while not A.finished(p) do C.insert(A.read(p), r) p ← A.next(p) while not B.finished(q) do C.insert(A.read(p), r) q ← B.next(q) return C	C ← Set() p ← A.head() r ← B.head() r ← C.head() while not A.finished() and not B.finished(q) do if A.read(p) < B.read(q) then C.insert(A.read(p), r) p ← A.next(p) else if A.read(p) = B.read(q) then p ← A.next(p) q ← B.next(q) while not A.finished(p) do C.insert(A.read(p), r) p ← A.next(p) return C

GRAFI

GRAPH

```

Graph()                                % Crea un grafo vuoto
InsertNode(NODE u)                     % Aggiunge il nodo u al grafo
insertEdge(NODE u, NODE v)              % Aggiunge l'arco (u, v) al grafo
deleteNode(NODE u)                     % Rimuove il nodo u dal grafo
deleteEdge(NODE u, NODE v)              % Rimuove l'arco (u, v) nel grafo
SET adj(NODE u)                        % Restituisce l'insieme dei nodi adiacenti ad u
SET V()                               % Restituisce l'insieme di tutti i nodi

```

visita(GRAPH G, NODE r)

```

SET S ← Set()
S.insert(r)
{ marca il nodo r come "scoperto" }
while S.size() > 0 do
  NODE u ← S.remove()
  { esamina il nodo u }
  foreach v ∈ G.adj(u) do
    { esamina l'arco (u, v) }
    if v non è già stato scoperto then
      { marca il nodo v come "scoperto" }
      S.insert(v)

```

bfs(GRAPH G, NODE r)

```

QUEUE S ← Queue()
S.enqueue(r)
boolean[] visitato ← new boolean[1...G.n]
foreach u ∈ G.V() - {r} do visitato[u] ← false
visitato[r] ← true
while not S.isEmpty() do
  NODE u ← S.dequeue()
  { esamina il nodo u }
  foreach v ∈ G.adj(u) do
    { esamina l'arco (u, v) }
    if not visitato[v] then
      visitato[v] ← true
      S.enqueue(v)

```

erdos(GRAPH G, NODE r, integer[] erdos, NODE[] p)

```

QUEUE S ← Queue()
S.enqueue(r)
foreach u ∈ G.V() - {r} do erdos[u] = ∞
erdos[r] ← 0
p[r] ← nil
while not S.isEmpty() do
  NODE u ← S.dequeue()
  foreach v ∈ G.adj(u) do
    if erdos[v] = ∞ then
      erdos[v] ← erdos[u] + 1
      p[v] ← u
      S.enqueue(v)

```

Cammino più breve tra due vertici:
Memorizzato tramite vettore dei padri p

```

stampaCammino(GRAPH G, NODE r, NODE s, NODE[] p)
if r = s then print s
else if p[s] = nil then
  print "nessun cammino da r a s"
else
  stampaCammino(G, r, p[s], p)
  print s

```

dfs(GRAPH G, NODE u, boolean[] visitato)

```

visitato[u] ← true
① { esamina il nodo u (caso previsita) }
foreach v ∈ G.adj(u) do
  { esamina l'arco (u, v) }
  if not visitato[v] then
    dfs(G, v, visitato)
② { esamina il nodo u (caso postvisita) }

```

Complessità:

O(n+m) Liste di adiacenza
O(n²) Matrice di adiacenza
O(m) Operazioni

dfs-schema(GRAPH G, NODE u)

```

esamina il nodo u prima (caso pre-visita)
time ← time + 1; dt[u] ← time
foreach v ∈ G.adj(u) do
  esamina l'arco (u, v) di qualsiasi tipo
  if dt[v] = 0 then
    esamina l'arco (u, v) in T
    dfs-schema(g, v)
  else if dt[u] > dt[v] and ft[v] = 0 then
    esamina l'arco (u, v) all'indietro
  else if dt[u] < dt[v] and ft[v] ≠ 0 then
    esamina l'arco (u, v) in avanti
  else
    esamina l'arco (u, v) di attraversamento
esamina il nodo u dopo (caso post-visita)
time ← time + 1; ft[u] ← time

```

Albero dei cammini DFS:

Gli archi non inclusi in T possono essere divisi in tre categorie durante la visita: se l'arco è esaminato passando da un nodo di T ad un altro nodo che è suo antenato in T, è detto arco all'indietro; se l'arco è esaminato passando da un nodo di T ad un suo discendente (che non sia figlio) in T è detto arco in avanti; altrimenti, è detto arco di attraversamento

Componenti connesse (G non orientato):

```

integer[] cc(GRAPH G, STACK S)
integer[] id ← new integer[1...G.n]
foreach u ∈ G.V() do id[u] ← 0
integer counter ← 0
while not S.isEmpty() do
  u ← S.pop()
  if id[u] = 0 then
    counter ← counter + 1
    ccdfs(G, counter, u, id)
return id

ccdfs(GRAPH G, integer counter, NODE u, integer[] id)
id[u] ← counter
foreach v ∈ G.adj(u) do
  if id[v] = 0 then
    ccdfs(G, counter, v, id)

```

DAG (Grafo orientato aciclico):

Non ha archi all'indietro

boolean ciclico(GRAPH G, NODE u)

```

time ← time + 1; dt[u] ← time
foreach v ∈ G.adj(u) do
  if dt[v] = 0 then
    if ciclico(G, v) then return true
  else if dt[u] > dt[v] and ft[v] = 0 then
    return true
time ← time + 1; ft[u] ← time
return false;

```

Ordinamento topologico (DAG):

STACK topSort(GRAPH G)

```

boolean[] visitato ← boolean[1...G.n]
foreach u ∈ G.V() do visitato[u] ← false
STACK S ← Stack()
foreach u ∈ G.V() do
  if not visitato[u] then
    ts-dfs(G, u, visitato, S)
return STACK

```

```

integer ts-dfs(GRAPH G, NODE u, boolean[] visitato, NODE[] STACK S)
visitato[u] ← true
foreach v ∈ G.adj(u) do
  if not visitato[v] then
    i ← ts-dfs(G, v, visitato, S)
  S.push(u)

```

Componenti fortemente connesse (G orientato) (Kosaraju):

```

integer[] scc(GRAPH G)
  STACK  $\leftarrow$  Stack()
  boolean[] visitato  $\leftarrow$  new boolean[1...G.n]
  foreach  $u \in G.V()$  do visitato[ $u$ ]  $\leftarrow$  false
  foreach  $u \in G.V()$  do dfsStack( $G$ , visitato, S,  $u$ )      In alternativa  STACKS  $\leftarrow$  topSort( $G$ )
  GRAPH  $G^T \leftarrow$  Graph()
  foreach  $u \in G.V()$  do  $G^T$ .insertNode( $u$ )          % Calcolo grafo trasposto
  foreach  $u \in G.V()$  do
    foreach  $v \in G.adj(u)$  do
       $G^T$ .insertEdge( $v, u$ )
  integer[] ordine  $\leftarrow$  new integer[1...G.n]           % Seconda visita
  foreach  $u \in G.V()$  do ordine[i]  $\leftarrow$  S.pop()
  return cc( $G^T$ , ordine)

```

dfsStack(GRAPH G, boolean [] visitato, STACK S, NODE u)

```

visitato[u]  $\leftarrow$  true
foreach  $v \in G.adj(u)$  do
  if not visitato[v] then
    dfsStack( $G$ , visitato, S,  $v$ )
S.push( $u$ )

```

boolean bipartito(GRAPH G, NODE r)

```

QUEUE S  $\leftarrow$  Queue()
S.enqueue(r)
corrente  $\leftarrow$  rosso
integer[] color  $\leftarrow$  new integer[1...G.n]
foreach  $u \in G.V()$  do color[ $u$ ]  $\leftarrow$  senzacolore
while not S.isEmpty() do
  NODE u  $\leftarrow$  S.dequeue()
  foreach  $v \in G.adj(u)$  do
    if color[v] = senzacolore then
      color[v]  $\leftarrow$  corrente
      S.enqueue(v)
    else
      if color[v]  $\neq$  corrente then return false
  corrente  $\leftarrow$  1 - corrente
return true

```

9.4

scheduling(GRAPH G, integer[] durata)

```

ordine  $\leftarrow$  new integer[1...G.n]
topSort( $G$ , ordine)
partenza  $\leftarrow$  new integer[1...G.n]
partenza[ordine[1]]  $\leftarrow$  0
for  $i \leftarrow 2$  to  $G.n$  do
  integer j  $\leftarrow$  ordine[i]
  partenza[j]  $\leftarrow$  partenza[j] + durata[j]

```

9.7

integer diameter(GRAPH G)

```

integer max  $\leftarrow$  0
foreach  $r \in G.V()$  do
  QUEUE S  $\leftarrow$  Queue()
  S.enqueue(r)
  integer[] dist  $\leftarrow$  new integer[1...G.n]
  foreach  $u \in G.V() - \{r\}$  do dist[ $u$ ]  $\leftarrow$  -1
  dist[r]  $\leftarrow$  0
  while not S.isEmpty() do
    NODE u  $\leftarrow$  S.dequeue()
    foreach  $v \in G.adj(u)$  do
      if dist[v] < 0 then
        dist[v]  $\leftarrow$  dist[u] + 1
        if dist[v] > max then
          max  $\leftarrow$  dist[v]
        S.enqueue(v)
  return max

```

9.11

dfs(GRAPH G, NODE r)

```

STACK S  $\leftarrow$  Stack()
S.push( $r$ )
boolean[] visitato  $\leftarrow$  new boolean[1...G.n]
foreach  $u \in G.V() - \{r\}$  do visitato[ $u$ ]  $\leftarrow$  false
visitato[r]  $\leftarrow$  true
while not S.isEmpty() do
  NODE u  $\leftarrow$  S.pop()
  { esamina il nodo  $u$  }
  foreach  $v \in G.adj(u)$  do
    { esamina l'arco ( $u, v$ ) }
    if not visitato[v] then
      visitato[v]  $\leftarrow$  true
      S.push( $v$ )

```

9.2

integer larghezza(GRAPH G, NODE r)

```

integer max  $\leftarrow$  0
QUEUE S  $\leftarrow$  Queue()
S.enqueue(r)
integer[] dist  $\leftarrow$  new integer[1...G.n]
integer[] count  $\leftarrow$  new integer[1...G.n]
foreach  $u \in G.V() - \{r\}$  do
  dist[ $u$ ]  $\leftarrow$  -1
  count[ $u$ ]  $\leftarrow$  0
dist[r]  $\leftarrow$  0
while not S.isEmpty() do
  NODE u  $\leftarrow$  S.dequeue()
  foreach  $v \in G.adj(u)$  do
    if dist[v] < 0 then
      dist[v]  $\leftarrow$  dist[u] + 1
      count[dist[v]]  $\leftarrow$  count[dist[v]] + 1
      S.enqueue(v)
return max(count, n)

```

9.5

integer maxDistance(GRAPH G, NODE r)

```

integer[] d  $\leftarrow$  new integer[1...G.n]          % Vettore distanze
integer max  $\leftarrow$  0                          % Distanza massima trovata finora
integer count  $\leftarrow$  0                        % Numero di nodi alla distanza massima
QUEUE S  $\leftarrow$  Queue()
S.enqueue(r)
foreach  $u \in G.V() - \{r\}$  do  $d[u] = \infty$ 
d[r]  $\leftarrow$  0
while not S.isEmpty() do
  NODE u  $\leftarrow$  S.dequeue()
  foreach  $v \in G.adj(u)$  do
    if  $d[v] = \infty$  then
      d[v]  $\leftarrow$  d[u] + 1
    if  $d[v] > max$  then
      max  $\leftarrow$  d[v]
      count  $\leftarrow$  1
    else if  $d[v] = max$  then
      count  $\leftarrow$  count + 1
    S.enqueue(v)
return count

```

9.9

quadrato(integer[][] A)

```

for i = 1 to n do
  for j = 1 to n do
    k  $\leftarrow$  1
    found  $\leftarrow$  false
    while k  $\leq n$   $\wedge$  not found do
      found  $\leftarrow$  A[i, k] = 1  $\wedge$  A[k, j] = 1  $\wedge$  i  $\neq$  k  $\wedge$  j  $\neq$  k
      k  $\leftarrow$  k + 1
    if found then
      A2[i, j]  $\leftarrow$  1
    else
      A2[i, j]  $\leftarrow$  0

```

9.10

```

universalSink(integer[][] A)
  i ← 1
  candidate ← false
  while i < n ∧ candidate = false do
    j ← i + 1
    while j ≤ n ∧ A[i, j] = 0 do
      j ← j + 1
    if j > n then
      candidate ← true
    else
      j ← i
  rowtot =  $\sum_{j \in \{1\dots n\} - \{i\}} A[i, j]$ 
  coltot =  $\sum_{j \in \{1\dots n\} - \{i\}} A[j, i]$ 
  return rowtot = 0 ∧ coltot = n - 1

```

9.9

```

maxdist(GRAPH G, NODE s)
  foreach v ∈ G.V() do
    mark[v] ← false;
  mark[s] ← true d[s] ← 0
  QUEUE Q ← Queue()
  Q.enqueue(s)
  current ← 0
  count ← 0
  while not Q.isEmpty() do
    v ← Q.dequeue()
    foreach u ∈ G.adj(v) do
      if mark[u] = false then
        mark[u] ← true
        d[u] ← d[v] + 1
        if u.d > current then
          current ← d[u]
        count ← 0
        count ← count + 1

```

9.10

```

integer[] reachability(GRAPH G)
  integer[] c ← newinteger[1\dots G.n]
  foreach u ∈ G.V() do count[u] ← 0
  foreach u ∈ G.V() do
    if c[u] = 0 then
      c[u] ← reach(u)
  return c

```

```

integer[] reach(GRAPH G, integer[] c)
  if c[u] = 0 then
    c[u] ← 1
    foreach v ∈ G.adj(u) do
      c[u] ← c[u] + reach(G, v)
  return c[u]

```

9.16

```

integer treeCount(GRAPH G)
  integer count ← 0
  time ← 0
  foreach u ∈ G.V() do
    dt[u] ← 0
    ft[u] ← 0
  foreach u ∈ G.V() do
    if dt[u] = 0 then
      if not ciclico(G, u) then
        count ← count + 1
  return count

boolean ciclico(GRAPH G, NODE u)
  time ← time + 1; dt[u] ← time
  foreach v ∈ G.adj(u) do
    if dt[v] = 0 then
      if ciclico(G, v) then return true
    else if dt[u] > dt[v] and ft[v] = 0 then
      return true
  time ← time + 1; ft[u] ← time
  return false;

```

9.20

```

integer averageDistance(GRAPH G, NODE r)
  QUEUE S ← Queue()
  S.enqueue(r)
  integer[] dist ← new integer[1\dots G.n]
  integer tot ← 0
  foreach u ∈ G.V() - {r} do dist[u] ← -1
  dist[r] ← 0
  while not S.isEmpty() do
    NODE u ← S.dequeue()
    foreach v ∈ G.adj(u) do
      if dist[v] < 0 then
        dist[v] ← dist[u] + 1
        tot ← tot + dist[v]
        S.enqueue(v)
  return tot/(G.n - 1)

```

9.21

```

integer facebook3(GRAPH G)
  integer max ← 0
  foreach v ∈ G.V() do
    integer t ← conta(GRAPH G, NODE v)
    if t > max then
      max ← t
  return max

integer conta(GRAPH G, NODE r)
  boolean[] visitato ← new boolean[1\dots G.n]
  foreach v ∈ G.V() - {r} do visitato[v] ← false
  visitato[r] ← true
  integer f ← 0
  foreach u ∈ G.adj(r) do
    visitato[u] ← true
    f ← f + 1
    foreach v ∈ G.adj(u) do
      if not visitato[v] then
        visitato[v] ← true
        f ← f + 1
  return f

```

9.23

```

boolean hasGreenCycle(GRAPH G, integer[] color, NODE u)
  time ← time + 1; dt[u] ← time
  foreach v ∈ G.adj(u) do
    if color[v] = GREEN then
      if dt[v] = 0 then
        if hasGreenCycle(G, v) then return true
      else if dt[u] > dt[v] and ft[v] = 0 then
        return true
  time ← time + 1; ft[u] ← time
  return false;

```

9.24

```

integer sameDistance(GRAPH G, NODE s1, NODE s2)
  QUEUE S ← Queue()
  S.enqueue(s1)
  integer[] dist1 ← new integer[1\dots G.n]
  foreach u ∈ G.V() - {r} do dist1[u] ← -1
  dist1[r] ← 0
  while not S.isEmpty() do
    NODE u ← S.dequeue()
    foreach v ∈ G.adj(u) do
      if dist1[v] < 0 then
        dist1[v] ← dist1[u] + 1
        S.enqueue(v)

  S ← Queue()
  S.enqueue(s2)
  integer[] dist2 ← new integer[1\dots G.n]
  foreach u ∈ G.V() - {r} do dist2[u] ← -1
  dist2[r] ← 0
  while not S.isEmpty() do
    NODE u ← S.dequeue()
    foreach v ∈ G.adj(u) do
      if dist2[v] < 0 then
        dist2[v] ← dist2[u] + 1
        S.enqueue(v)

```

```

integer c ← 0
foreach u ∈ G.V() do
  if dist1[s1] = dist2[s2] then
    c ← c + 1
return c

```

9.27

```

stampaPozziSorgenti(GRAPH G)
  boolean[] in ← new boolean[1\dots n]
  foreach u ∈ G.V() do
    in[u] ← false
  foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
      in[v] ← true
  print "Sorgenti:"
  foreach u ∈ G.V() do
    if not in[u] then
      print u
  print "Pozzi:"
  foreach u ∈ G.V() do
    if G.adj(u) = ∅ then
      print u

```

9.25

CODE CON PRIORITA' - HEAPSORT

PRIORITYQUEUE

```
integer capacità % Numero massimo di elementi nella coda
integer dim % Numero attuale di elementi nella coda
PRIORITYITEM[] H % Vettore heap
```

```
PRIORITYQUEUE PriorityQueue(integer n)
  PRIORITYQUEUE t ← new PRIORITYQUEUE
  t.capacità ← n
  t.dim ← 0
  t.H ← new PRIORITYITEM[1 . . . n]
  return t
```

```
ITEM min()
  precondition: dim > 0
  return H[1].valore
```

% Crea una coda con priorità vuota
PriorityQueue()

% Restituisce true se la coda con priorità è vuota
boolean isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota
ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota
ITEM deleteMin()

% Inserisce l'elemento x con priorità p in una coda con priorità non piena
% Restituisce un oggetto PRIORITYITEM che identifica x all'interno della coda
PRIORITYITEM insert(ITEM x , integer p)

% Diminuisce la priorità dell'elemento identificato da x portandola al valore p
decrease(PRIORITYITEM x , integer p)

PRIORITYITEM insert(ITEM x , integer p)

precondition: dim < capacità

```
dim ← dim + 1
H[dim] ← new PRIORITYITEM()
H[dim].valore ← x
H[dim].priorità ← p
H[dim].pos ← dim
integer i ← dim
while i > 1 and H[i].priorità < H[p(i)].priorità do
  swap(H, i, p(i))
  i ← p(i)
return H[i]
```

```
swap(PRIORITYITEM[] H, integer i, integer j)
  H[i] ← H[j]
  H[i].pos ← i
  H[j].pos ← j
```

maxHeapRestore(ITEM[] A, integer i, integer dim)

```
integer max ← i
if l(i) ≤ dim and A[l(i)] > A[max] then max ← l(i)
if r(i) ≤ dim and A[r(i)] > A[max] then max ← r(i)
if i ≠ max then
  A[i] ↔ A[max]
  maxHeapRestore(A, max, dim)
```

Complessità: O(logn)

maxHeapRestore(ITEM[] A, integer i, integer dim)

```
boolean stop ← false
while not stop do
  integer max ← i
  if l(i) ≤ dim and A[l(i)] > A[max] then max ← l(i)
  if r(i) ≤ dim and A[r(i)] > A[max] then max ← r(i)
  if i ≠ max then
    A[i] ↔ A[max]
    i ← max
  else
    stop ← true
```

10.5

heapBuild(ITEM[] A, integer n)

```
PRIORITYQUEUE H ← new PriorityQueue(n)
for i ← 1 to n do
  H.insert(A[i])
```

10.6

ITEM deleteMin(ITEM x)

```
precondition: dim > 0
swap(H, 1, dim)
dim ← dim - 1
minHeapRestore(H, 1, dim)
return H[dim + 1]
```

minHeapRestore(PRIORITYITEM[] A, integer i, integer dim)

```
integer min ← i
if l(i) ≤ dim and A[l(i)].priorità < A[min].priorità then min ← l(i)
if r(i) ≤ dim and A[r(i)].priorità < A[min].priorità then min ← r(i)
if i ≠ min then
  swap(A, i, min)
minHeapRestore(A, min, dim)
```

decrease(PRIORITYITEM x , integer p)

```
precondition: p < x.priorità
x.priorità ← p
integer i ← x.pos
while i > 1 and H[i].priorità < H[p(i)].priorità do
  swap(H, i, p(i))
  i ← p(i)
```

sort(ITEM[] A, integer n)

```
PRIORITYQUEUE Q ← PriorityQueue(n)
for integer i ← 1 to n do
  Q.insert(A[i], A[i])
for integer i ← 1 to n do
  A[i] = Q.min()
  Q.deleteMin()
```

heapsort(ITEM[] A, integer n)

```
heapBuild(A, n)
for integer i ← n downto 2 do
  A[1] ↔ A[i]
  maxHeapRestore(A, 1, i - 1)
```

heapBuild(ITEM[] A, integer n)

```
for integer i ← ⌊n/2⌋ downto 1 do
  maxHeapRestore(A, i, n)
```

Complessità: O(nlogn)

integer [][] merge(integer[][] A, integer k, integer m)

```
PRIORITYQUEUE Q ← new PriorityQueue(k) % Coda con priorità per decidere il minimo
integer[] V ← new integer[km] % Vettore unito
integer[] p ← new integer[k] % Posizione attuale di ognuno dei vettori da unire
for i ← 1 to k do
  p[i] ← 2
  Q.insert((i, A[i][1]), A[i][1])
integer c ← 1 % Posizione nel vettore unito
while not Q.isEmpty() do
  (i, v) ← Q.deleteMin()
  V[c] ← v
  c ← c + 1
  if p[i] ≤ m then
    Q.insert((i, A[i][p[i]]), A[i][p[i]])
    p[i] ← p[i] + 1
return V
```

10.7

CHECKMAXHEAP(integer[] A, integer n)

```
for i ← 2 to n do
  if A[i/2] < A[i] then
    return false;
```

return true

10.10

UNIONE DI INSIEMI DISGIUNTI (MERGE-FIND)

MFSET

```
% Crea n componenti {1}, ..., {n}
MFSET Mset(integer n)
% Restituisce il rappresentante della componente contenente x
integer find(integer x)
% Unisce le componenti che contengono x e y
merge(integer x, integer y)
```

```
MFSET cc(GRAPH G)
MFSET M ← Mset(|G.V|)
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        M.merge(u, v)
return M
```

```
integer find(integer x)
integer r ← x;
while p[r] ≠ r do r ← p[r]
while p[x] ≠ r do
    integer temp ← p[x]
    p[x] ← r
    x ← temp
return r
```

10.9

Senza compressione:

```
integer find(integer x)
if p[x] = x then
    return x
else
    return find(x,p)
```

Con compressione:

```
integer find(integer x)
if p[x] ≠ x then
    p[x] ← find(p[x])
return p[x]
```

MFSET

```
integer[] p
integer[] rango
Mset(integer n)
MFSET t ← new MFSET
t.p ← integer[1...n]
t.rango ← integer[1...n]
for integer i ← 1 to n do
    t.p[i] ← i
    t.rango[i] ← 0
return t
```

Senza euristica sul rango:

```
merge(integer x, integer y)
rx ← find(x)
ry ← find(y)
if rx ≠ ry then
    p[ry] ← rx
```

Con euristica sul rango:

```
merge(integer x, integer y)
rx ← find(x)
ry ← find(y)
if rx ≠ ry then
    if rango[rx] > rango[ry] then
        p[ry] ← rx
    else if rango[ry] > rango[rx] then
        p[rx] ← ry
    else
        p[rx] ← ry
        rango[ry] ← rango[ry] + 1
```

CAMMINI MINIMI

Dijkstra:

Coda con priorità, realizzata tramite vettore/lista non ordinati

Costo totale: $O(n^2)$

	Costo	Ripetizioni
• Riga (1):	$O(n)$	1
• Riga (2):	$O(n)$	$O(n)$
• Riga (3):	$O(1)$	$O(n)$
• Riga (4):	$O(1)$	$O(m)$

Johnson:

Coda con priorità, tramite Heap binario

Costo totale: $O(m \log n)$

	Costo	Ripetizioni
• Riga (1):	$O(n)$	1
• Riga (2):	$O(\log n)$	$O(n)$
• Riga (3):	$O(\log n)$	$O(n)$
• Riga (4):	$O(\log n)$	$O(m)$

Fredman - Tarjan:

Coda con priorità, tramite Heap di Fibonacci

Costo totale: $O(m + n \log n)$

	Costo	Ripetizioni
• Riga (1):	$O(n)$	1
• Riga (2):	$O(\log n)$	$O(n)$
• Riga (3):	$O(\log n)$	$O(n)$
• Riga (4):	$O(1)$	$O(m)$

- (1) PRIORITYQUEUE $S \leftarrow$ PriorityQueue(); $S.insert(r, 0)$
- (2) $u \leftarrow S.deleteMin()$
- (3) $S.insert(v, d[u] + w(u, v))$
- (4) $S.decrease(v, d[u] + w(u, v))$

camminiMinimi(GRAPH G, NODE r, integer[] T)

```
integer[] d ← new integer[1...G.n] % d[u] è la distanza da r a u
boolean[] b ← new boolean[1...G.n] % b[u] è true se u è contenuto in S
foreach u ∈ G.V() – {r} do
    T[u] ← nil
    d[u] ← +∞
    b[u] ← false
    T[r] ← nil
    d[r] ← 0
    b[r] ← true
(1) STRUTTURADATI  $S \leftarrow$  StrutturaDati();  $S.aggungi(r)$ 
    while not  $S.isEmpty()$  do
        (2)  $u \leftarrow S.estrai()$ 
            b[u] ← false
            foreach v ∈ G.adj(u) do
                if  $d[u] + w(u, v) < d[v]$  then
                    (3) if not b[v] then
                        S.aggungi(v)
                        b[v] ← true
                    else
                        (4) % Azione da intraprendere nel caso v sia già presente in S
                        T[v] ← u
                        d[v] ← d[u] + w(u, v)
```

Bellman - Ford - Moore:

Coda

Costo totale: $O(mn)$

	Costo	Ripetizioni
• Riga (1):	$O(1)$	1
• Riga (2):	$O(1)$	$O(n^2)$
• Riga (3):	$O(1)$	$O(nm)$

Pape - D'Esopo:

	DeQueue	Tempo di calcolo	In generale, superpolinomiale	In pratica, veloce per grafi che rappresentano reti di circolazione stradale
• Riga (1):	$O(1)$	$O(1)$		
• Riga (2):	$O(1)$	$O(n^2)$		
• Riga (3):	$O(1)$	$O(nm)$		
(1) QUEUE $S \leftarrow$ Queue(); $S.enqueue(r)$			(1) DeQUEUE $S \leftarrow$ DeQueue(); $S.insertHead(r)$	
(2) $u \leftarrow S.dequeue()$			(2) $u \leftarrow S.removeHead()$	
(3) $S.enqueue(v)$			(3) if $d[v] = +\infty$ then $S.insertTail(v)$ else $S.insertHead(v)$	
(4) Sezione non necessaria			(4) Sezione non necessaria	

```
negativeCycle(GRAPH G, NODE r)
```

```
integer[] d ← new integer[1...G.n]
boolean[] b ← new boolean[1...G.n]
boolean[] L ← new boolean[1...G.n]
foreach  $u \in G.V() - \{r\}$  do  $d[u] \leftarrow +\infty$ 
foreach  $u \in G.V() - \{r\}$  do  $b[u] \leftarrow \text{false}$ 
 $L[r] \leftarrow 0$ ;  $d[r] \leftarrow 0$ ;  $b[r] \leftarrow \text{true}$ 
QUEUE  $S \leftarrow \text{Queue}(); S.\text{enqueue}(r)$ 
while not  $S.\text{isEmpty}()$  do
     $u \leftarrow S.\text{dequeue}()$ 
     $b[u] \leftarrow \text{false}$ 
    foreach  $v \in G.\text{adj}(u)$  do
        if  $d[u] + w(u, v) < d[v]$  then
            if not  $b[v]$  then
                 $S.\text{enqueue}(v)$ 
                 $b[v] \leftarrow \text{true}$ 
                 $d[v] \leftarrow d[u] + w(u, v)$ 
                 $L[v] \leftarrow L(u) + 1$ 
            if  $L[v] \geq n$  then return true
return false
```

% $d[u]$ è la distanza da r a u

% $b[u]$ è **true** se u è contenuto in S

% $L[u]$ è la distanza da r a u (in numero di archi)

```
integer pathcount(GRAPH G, NODE s, NODE t)
foreach  $v \in V$  do
     $a[v] \leftarrow -1$ 
 $a[t] \leftarrow 1$ 
return r-pathcount( $G, s$ )
```

```
integer r-pathcount(GRAPH G, NODE u)
if  $a[u] < 0$  then
     $a[u] \leftarrow 0$ 
    foreach  $v \in G.\text{adj}(u)$  do
         $a[u] \leftarrow a[u] + \text{r-pathcount}(G, v)$ 
return  $a[s]$ 
```

11.7

11.2

DIVIDE ET IMPERA

```
hanoi-ricorsiva(integer n, integer origine, integer destinazione, integer intermedio)
```

```
if  $n = 1$  then
    % Trasferisci un disco da origine a destinazione
else
    (1) hanoi-ricorsiva( $n - 1, \text{origine}, \text{intermedio}, \text{destinazione}$ )
    % Trasferisci un disco da origine a destinazione
    (2) hanoi-ricorsiva( $n - 1, \text{intermedio}, \text{destinazione}, \text{origine}$ )
```

Equazione di ricorrenza: $T(n) = 2T(n-1)+1$

```
moltiplicaPolinomi(real[] p, real[] q, real[] r, integer n)
```

```
for integer  $h \leftarrow 0$  to  $2 \cdot (n - 1)$  do  $r[h] \leftarrow 0$ 
for  $h \leftarrow 0$  to  $n - 1$  do
    for integer  $k \leftarrow 0$  to  $n - 1$  do
         $r[h + k] \leftarrow r[h + k] + p[h] \cdot q[k]$ 
```

Complessità: $\Theta(n^2)$

```
moltiplicaPolinomiD&I(real[] p, real[] q, real[] r, integer n)
```

```
if  $n = 1$  then
     $r[0] \leftarrow p[0] \cdot q[0]$ 
else
    ripartisci  $p$  in  $p^S$  e  $p^D$ 
    ripartisci  $q$  in  $q^S$  e  $q^D$ 
    calcola  $r^S$ ,  $r^M$  ed  $r^D$  richiamando moltiplicaPolinomiD&I()
    ricava  $r$  in funzione di  $r^S$ ,  $r^M$  ed  $r^D$ 
```

Complessità: $O(n^{1.59})$

```
moltiplicaMatrici(real[][] A, real[][] B, real[][] C, integer n)
```

```
for integer  $i \leftarrow 1$  to  $n$  do
    for integer  $j \leftarrow 1$  to  $n$  do
         $C[i, j] \leftarrow 0$ 
        for integer  $k \leftarrow 1$  to  $n$  do
             $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
```

Complessità: $\Theta(n^3)$

```
MergeSort(integer[] A, integer n)
```

```
integer  $s \leftarrow 1$ 
while  $s < n$  do
    integer  $p \leftarrow 1$ 
    while  $p + s \leq n$  do
        Merge( $A, p, p + s, \min(p + 2 \cdot s - 1, n)$ )
         $p \leftarrow p + 2 \cdot s$ 
     $s \leftarrow s \cdot 2$ 
```

12.5

```
QuickSort(ITEM[] A, integer primo, integer ultimo)
```

```
if  $primo < ultimo$  then
    integer  $j \leftarrow \text{perno}(A, primo, ultimo)$ 
    QuickSort( $A, primo, j - 1$ )
    QuickSort( $A, j + 1, ultimo$ )
```

```
integer perno(ITEM[] A, integer primo, integer ultimo)
```

```
ITEM  $x \leftarrow A[primo]$ 
integer  $j \leftarrow primo$ 
for integer  $i \leftarrow primo$  to  $ultimo$  do
    if  $A[i] < x$  then
         $j \leftarrow j + 1$ 
         $A[i] \leftrightarrow A[j]$ 
 $A[primo] \leftarrow A[j]$ 
 $A[j] \leftarrow x$ 
return  $j$ 
```

Caso medio:

$O(n \log n)$

Caso pessimo:

$O(n^2)$ vettore già ordinato

```
strassen(real[][] A, real[][] B, real[][] C, integer n)
```

```
if  $n = 1$  then
     $C[1, 1] \leftarrow A[1, 1] \cdot B[1, 1]$ 
else
    ripartisci  $A$  in  $A_{11}, A_{12}, A_{21}$  e  $A_{22}$ 
    ripartisci  $B$  in  $B_{11}, B_{12}, B_{21}$  e  $B_{22}$ 
    calcola  $X_1, \dots, X_7$  richiamando strassen()
    ricava  $C_{11}, C_{12}, C_{21}, C_{22}$  in funzione di  $X_1, \dots, X_7$ 
```

Complessità: $\Theta(n^{2.81})$

$$T(n) = \begin{cases} 1 & n = 1 \\ 7T(n/2) + n^2 & n > 1 \end{cases}$$

```
integer inversioni(ITEM A[], integer primo, integer ultimo)
```

```
if  $primo \geq ultimo$  then
    return 0
else
    mezzo  $\leftarrow \lfloor (primo + ultimo)/2 \rfloor$ 
    return inversioni( $A, primo, mezzo$ ) + inversioni( $A, mezzo + 1, ultimo$ ) +
    Contamerge( $A, primo, ultimo, mezzo$ )
```

```
if  $A[i] \leq A[j]$  then
```

```
     $B[k] \leftarrow A[i]$ 
```

```
     $i \leftarrow i + 1$ 
```

else

```
     $B[k] \leftarrow A[j]$ 
```

```
     $j \leftarrow j + 1$ 
```

```
    conta  $\leftarrow conta + mezzo - i + 1$ 
```

12.3

boolean ricerca(ITEM[][] M , **integer** i , **integer** j , **integer** k , **integer** n)

```

if  $M[i, j] = k$  then
| return true
else if  $M[i, j] < k$  then
| if  $i < n$  then return ricerca( $M, i + 1, j, k, n$ )
| else return false
else
| if  $j < 1$  then return ricerca( $M, i, j - 1, k, n$ )
| else return false

```

12.6

integer conta0(**integer**[] V , **integer** i, j)

```

if  $i = j$  or  $i = j - 1$  then
| while  $V[i] \neq 1$  and  $i \leq j$  do  $i \leftarrow i + 1$ 
| return  $i - 1$ 
else
|  $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
| if  $V[m] = 1$  then
| | return conta0( $V, i, m - 1$ )
| else
| | return conta0( $V, m + 1, j$ )

```

integer trovaH(**integer**[] V , **integer** n)

```

integer  $i \leftarrow 1$ 
while  $V[i] = 0$  and  $2 \cdot i \leq n$  do
|  $i \leftarrow 2 \cdot i$ 
if  $V[i] = 0$  then
| return conta0( $V, i, n$ )
else if  $i/2 > 1$  then
| return conta0( $V, i/2, i$ )
else
| return conta0( $V, 1, i$ )

```

12.9

integer missing(**integer**[] A , **integer** i, j)

```

if  $i = j$  then
| return  $i + 1$ 
 $m \leftarrow \lceil (i + j)/2 \rceil$ 
if  $A[m] = m$  then
| return missing( $A, m, j$ )
else
| return missing( $A, i, m - 1$ )

```

12.11

boolean puntofisso(**integer**[] A , **integer** i, j)

```

if  $j > i$  then
| return false
integer  $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
if  $A[m] = m$  then
| return true
else if  $A[m] < m$  then
| return puntofisso( $A, m + 1, j$ )
else
| return puntofisso( $A, i, m - 1$ )

```

12.5

integer maxsumRic(**integer**[] A , **integer** i, j)

```

integer  $max_s, max_d, max'_s, max'_d, s, m, k$ 
if  $i > j$  then return 0
if  $i = j$  then return max(0,  $A[i]$ )
 $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
 $max_s \leftarrow \text{maxsumRic}(A, i, m)$ 
 $max_d \leftarrow \text{maxsumRic}(A, m + 1, j)$ 
 $sum \leftarrow 0$ 
 $max'_d \leftarrow max'_s 0$ 
for  $k \leftarrow m$  downto  $i$  do
|  $sum \leftarrow sum + A[k]$ 
| if  $sum > max'_s$  then  $max'_s \leftarrow sum$ 
 $sum \leftarrow 0$ 
for  $k \leftarrow m + 1$  to  $j$  do
|  $sum \leftarrow sum + A[k]$ 
| if  $sum > max'_d$  then  $max'_d \leftarrow sum$ 
return max( $max_s, max_d, max'_s + max'_d$ )

```

12.8

boolean torneo(ITEM[][] M , **integer** $imin, imax, jmin, jmax, gmin, gmax$)

```

if  $gmin = gmax - 1$  then
|  $M[imin, jmin] \leftarrow gmin$ 
|  $M[imin, jmax] \leftarrow gmax$ 
|  $M[imax, jmin] \leftarrow gmax$ 
|  $M[imax, jmax] \leftarrow gmin$ 
else
| integer  $imed \leftarrow \lfloor (imin + imax)/2 \rfloor$ 
| integer  $jmed \leftarrow \lfloor (jmin + jmax)/2 \rfloor$ 
| integer  $gmed \leftarrow \lfloor (gmin + gmax)/2 \rfloor$ 
| torneo( $M, imin, imed, jmin, jmed, gmin, gmed$ )
| torneo( $M, imed + 1, imax, jmed + 1, jmax, gmin, gmed$ )
| torneo( $M, imed + 1, imax, jmin, jmed, gmed + 1, gmax$ )
| torneo( $M, imin, imed, jmed + 1, jmax, gmed + 1, gmax$ )

```

12.7

integer min(ITEM[][] M , **integer** $imin, imax, integer jmin, integer jmax$)

```

if  $imin > imax$  or  $jmin > jmax$  then
| return  $+\infty$ 
else if  $imin = imax$  and  $jmin = jmax$  then
| return  $M[imin, imax]$ 
else
| integer  $imed \leftarrow \lfloor (imin + imax)/2 \rfloor$ 
| integer  $jmed \leftarrow \lfloor (jmin + jmax)/2 \rfloor$ 
| integer  $a \leftarrow \min(imin, imed, jmin, jmed)$ 
| integer  $b \leftarrow \min(imed + 1, imax, jmin, jmed)$ 
| integer  $c \leftarrow \min(imin, imed, jmed + 1, jmax)$ 
| integer  $d \leftarrow \min(imin + 1, imax, jmed + 1, jmax)$ 
| return min4( $a, b, c, d$ )

```

12.10

integer maxUnimodale(**ITEM**[] A , **integer** i, j)

```

integer  $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
if  $i = j$  then
| return  $A[m]$ 
else
| if  $A[m] < A[m + 1]$  then
| | return maxUnimodale( $A, m + 1, j$ )
| else
| | return maxUnimodale( $A, i, m$ )

```

12.12

integer vum(**integer**[] A , **integer** i, j)

```

if  $i = j$  then
| return  $A[i]$ 
if  $j = i + 1$  then
| return min( $A[i], A[j]$ )
integer  $m = (i + j)/2$ 
if  $A[m - 1] > A[m]$  and  $A[m + 1] > A[m]$  then
| return  $A[m]$ 
if  $A[m - 1] > A[m]$  then
| return vum( $A, m + 1, j$ )
else
| return vum( $A, i, m - 1$ )

```

12.6

boolean majority(**integer**[] A , **integer** n)

```

 $m \leftarrow \text{median}(A, n)$ 
 $count \leftarrow 0$ 
for  $i \leftarrow 1$  to  $n$  do
| if  $A[i] = m$  then
| |  $count \leftarrow count + 1$ 
return ( $count > n/2$ )

```

12.9

integer trova(ITEM [] A , **integer** i, j)

```

if  $i + 1 = j$  then
| return  $i$ 
else
| integer  $m \leftarrow \lfloor (i + j)/2 \rfloor$ 
| if  $A[i] < A[m]$  then
| | return trova( $A, i, m$ )
| else
| | return trova( $A, m, j$ )

```

12.11

PROGRAMMAZIONE DINAMICA

```
integer C(integer n, k)
  if n = k or k = 0 then return 1
    else return C(n - 1, k - 1) + C(n - 1, k)
```

```
tartaglia(integer n, integer[][] C)
  for integer i ← 0 to n do
    C[i, 0] ← 1
    C[i, i] ← 1
  for integer i ← 2 to n do
    for integer j ← 1 to i - 1 do
      C[i, j] ← C[i - 1, j - 1] + C[i - 1, j]
```

```
SET maxinterval(integer[] a, integer[] b, integer[] w)
  { ordina gli intervalli per estremi di fine crescenti }
  { calcola  $p_j$  }
  integer[] D ← new integer[0 … n]
  D[0] ← 0
  for integer i ← 1 to n do
    D[i] ← max(D[i - 1], w[i] + D[p_i])
  i ← n
  SET S ← Set()
  while i > 0 do
    if D[i - 1] > w[i] + D[p_i] then
      i ← i - 1
    else
      S.insert(i)
      i ← p_i
  return S
```

```
integer stringMatching(ITEM[] P, ITEM[] T, integer m, integer n)
  integer[][] D ← new integer[0 … m][0 … n]
  for integer i ← 1 to m do D[i, 0] ← i
  for integer j ← 0 to n do D[0, j] ← 0
  for i ← 1 to m do
    for j ← 1 to n do
      integer t ← D[i - 1, j - 1] + if(P[i] = T[j], 0, 1)
      t ← min(t, D[i - 1, j] + 1)
      t ← min(t, D[i, j - 1] + 1)
      D[i, j] ← t
  integer min ← D[m, 0]
  integer pos ← 0
  for j ← 1 to n do
    if D[m, j] < min then
      min ← D[m, j]
      pos ← j
  return pos
```

$D[i, j] = \begin{cases} 0 & \text{se } i = 0 \\ i & \text{se } j = 0 \\ \min\{D[i - 1, j - 1] + \delta, D[i - 1, j] + 1, D[i, j - 1] + 1\} & \text{altrimenti} \end{cases}$

```
integer stringMatchingSingleRow(ITEM[] P, ITEM[] T, integer m, integer n)
  integer[] D ← new [0 … n]
  for integer j ← 0 to n do D[j] ← 0
  for i ← 1 to m do
    integer dx ← i
    for j ← 1 to n do
      integer t ← min(D[j - 1] + if(P[i] = T[j], 0, 1), D[j] + 1, dx + 1)
      dx ← D[j]
      D[j] ← t
    D[0] ← i
    integer min ← D[0]
    for j ← 1 to n do min ← min(min, D[j])
  return min
```

```
parantesizzazione(integer[] c, integer[][] M, integer[][] S, integern)
  integer i, j, h, k, t
  for i ← 1 to n do
    M[i, i] ← 0
  for h ← 2 to n do
    for i ← 1 to n - h + 1 do
      j ← i + h - 1
      M[i, j] ← +∞
      for k ← i to j - 1 do
        t ← M[i, k] + M[k + 1, j] + c[i - 1] · c[k] · c[j]
        if t < M[i, j] then
          M[i, j] ← t
          S[i, j] ← k
```

$$M[i, j] = \begin{cases} \min_{1 \leq k \leq j-1} \{M[i, k] + M[k+1, j] + c_{i-1} c_k c_j\} & \text{se } 1 \leq i < j \leq n, \\ 0 & \text{se } 1 \leq i = j \leq n \end{cases}$$

$$F(n) = \begin{cases} 1 & \text{se } n = 1, \\ \sum_{k=1}^{n-1} F(k)F(n-k) & \text{altrimenti} \end{cases}$$

Complessità: $O(n^3)$

```
integer recPar(integer[] c, integer i, integer j)
  if i = j then
    return 0
  else
    min ← +∞
    for integer k ← i to j - 1 do
      integer q ← recPar(c, i, k) + recPar(c, k + 1, j) + c[i - 1] · c[k] · c[j]
      if q < min then min ← q
    return min
```

Complessità: $\Omega(2^n)$

```
integer[][] multiply(integer[][] S, integer i, integer j)
  if i = j then
    return A[i]
  else
    integer[][] X ← multiply(S, i, S[i, j])
    integer[][] Y ← multiply(S, S[i, j] + 1, j)
    return matrix-multiplication(X, Y)
```

```
stampaPar(integer[][] S, integer i, integer j)
  if i = j then
    print "A["; print i; print "]"
  else
    print "("
    stampaPar(S, i, S[i, j])
    print ","
    stampaPar(S, S[i, j] + 1, j)
    print ")"
```

Fibonacci ricorsiva:

```
integer fibonacci(integer n)
  if n ≤ 1 then
    return 1
  else
    return fibonacci(n - 1) + fibonacci(n - 2)
```

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

Complessità: $O(2^n)$

Fibonacci iterativa:

```
integer fibonacci(integer n)
  integer[] F ← new integer[0 … max(n, 1)]
  F[0] ← F[1] ← 1
  for integer i ← 2 to n do
    F[i] ← F[i - 1] + F[i - 2]
  return F[n]
```

Complessità tempo: $O(n)$
Complessità spazio: $O(n)$

Fibonacci iterativa con risparmio:

```
integer fibonacci(integer n)
  integer F0, F1, F2
  F0 ← F1 ← F2 ← 1
  for integer i ← 2 to n do
    F0 ← F1
    F1 ← F2
    F2 ← F0 + F1
  return F2
```

Complessità tempo: $O(n)$
Complessità spazio: $O(1)$

```

integer zaino(integer[] p, integer[] v, integer i, integer c, integer[][] D)
if i = 0 or c = 0 then
    ↘ return 0
if c < 0 then
    ↘ return -∞
if D[i, c] = ∞ then
    ↘ D[i, c] ← max(zaino(p, v, i - 1, c, D), zaino(p, v, i - 1, c - v[i], D) + p[i])
return D[i, c]

```

$$D[i, c] = \begin{cases} 0 & \text{se } i = 0 \vee c = 0 \\ -\infty & \text{se } c < 0 \\ \max\{D[i - 1, c], D[i - 1, c - v_i] + p_i\} & \text{altrimenti} \end{cases}$$

Complessità:

$O(nC)$ caso pessimo

```

floydWarshall(GRAPH G, integer[][] d, integer[][] p)
foreach u, v ∈ G.V() do
    d[u, v] ← +∞
    p[u, v] ← 0
foreach u ∈ G.V() do
    foreach v ∈ G.adj(u) do
        d[u, v] ← w(u, v)
        p[u, v] ← u
for k ← 1 to n do
    foreach u ∈ G.V() do
        foreach v ∈ G.V() do
            if d[u, k] + d[k, v] < d[u, v] then
                d[u, v] ← d[u, k] + d[k, v]
                p[u, v] ← d[k, v]

```

Complessità: $O(n^3)$

```

printApprox(ITEM[] P, ITEM[] T, integer[][] D, integer m, integer n)
integer i ← m
integer j ← 2
for k ← 2 to n do
    if D[m, k] < D[m, j] then j ← k
while i ≥ 0 and j ≥ 0 do
    if P[i] = T[j] then
        print P[i]
        i ← i - 1; j ← j - 1
    else if D[i, j] = D[i, j - 1] + 1 then
        print Cancellazione T[j]
        j ← j - 1
    else if D[i, j] = D[i, j - 1] + 1 then
        print Inserimento P[i]
        i ← i - 1
    else
        print Sostituzione T[j], P[i]
        i ← i - 1; j ← j - 1

```

13.1

```

longestIncreasingSequence(integer[] V, integer n)
integer[] D ← new integer[1...n]
integer[] P ← new integer[1...n]
integer max ← 1
for i ← 1 to n do
    D[i] ← 1;
    P[i] ← 0;
    for j ← 1 to i - 1 do
        if V[j] < V[i] and D[j] + 1 > D[i] then
            D[i] ← D[j] + 1
            P[i] ← j
            if D[i] > D[max] then max ← i
    printLongest(v, P, n, max)

```

```

printLongest(integer[] V, integer[] D, integer n, integer i))
if i > 0 then
    printLongest(v, P, n, P[i])
    print V[i]

```

13.6

```

integer lcs(integer[][], ITEM[] P, ITEM[] T, integer i, integer j)

```

```

if i = 0 or j = 0 then
    ↘ return 0
if D[i, j] = nil then
    if P[i] = T[j] then
        ↘ D[i, j] ← lcs(D, P, T, i - 1, j - 1) + 1
    else
        ↘ D[i, j] ← max(lcs(D, P, T, i - 1, j), lcs(D, P, T, i, j - 1))
return D[i, j]

```

$$D[i, j] = \begin{cases} 0 & \text{se } i = 0 \vee j = 0 \\ D[i - 1, j - 1] + 1 & \text{se } i > 0 \wedge j > 0 \wedge p_i = t_j \\ \max\{D[i - 1, j], D[i, j - 1]\} & \text{se } i > 0 \wedge j > 0 \wedge p_i \neq t_j \end{cases}$$

```

integer printLCS(ITEM[] T, ITEM[] P, integer[][] D, integer i, integer j)

```

```

if i ≠ 0 and j ≠ 0 then
    if T[i] = P[j] then
        printLCS(T, P, D, i - 1, j - 1)
        print P[i]
    else if D[i - 1, j] > D[i, j - 1] then
        printLCS(T, P, D, i - 1, j)
    else
        printLCS(T, P, D, i, j - 1)

```

```

camminiMinimi(GRAPH G, NODE r, integer[] T)

```

```
integer[] d ← new integer[1...G.n]
```

```
foreach u ∈ G.V() - {r} do
```

```
    T[u] ← nil
```

```
    d[u] ← +∞
```

```
T[r] ← nil
```

```
d[r] ← 0
```

```
for i ← 1 to G.n - 1 do
```

```
    foreach u ∈ G.V() do
```

```
        foreach v ∈ G.adj(u) do
```

```
            if d[u] + w(u, v) < d[v] then
```

```
                T[v] ← u
```

```
                d[v] ← d[u] + w(u, v)
```

13.3

```

camminiMinimi-DAG(GRAPH G, integer[] T)

```

```
integer[] d ← new integer[1...G.n]
```

```
for u ← 2 to G.n do
```

```
    T[u] ← nil
```

```
    d[u] ← +∞
```

```
T[1] ← nil
```

```
d[1] ← 0
```

```
for u ← 1 to G.n - 1 do
```

```
    foreach v ∈ G.adj(u) do
```

```
        if d[u] + w(u, v) < d[v] then
```

```
            T[v] ← u
```

```
            d[v] ← d[u] + w(u, v)
```

13.4

% Indice del valore più alto in D

% Se non troviamo valori minori, consideriamo
% la sottosequenza composta dal solo valore $v[i]$

```

printdiff(integer[][] D, ITEM[] P, ITEM[] T, integer i, integer j)
  if  $i > 0$  and  $j > 0$  and  $P[i] = T[j]$  then
    | printdiff( $D, P, T, i - 1, j - 1$ )
    | print " "  $P[i]$ 
  else if  $j > 0$  and ( $i = 0$  or  $D[i, j - 1] \geq D[i - 1, j]$ ) then
    | printdiff( $D, P, T, i, j - 1$ )
    | print " "  $T[j]$ 
  else if  $i > 0$  and ( $j = 0$  or  $D[i, j - 1] < D[i - 1, j]$ ) then
    | printdiff( $D, P, T, i - 1, j$ )
    | print "+"  $P[i]$ 

```

13.10

```

mincosto(integer[] D, integer[] C, integer n)
  integer[] M  $\leftarrow$  new integer[ $0 \dots n + 1$ ]
   $M[n + 1] \leftarrow 0$ 
  for  $i \leftarrow n$  downto 0 do
    | integer  $j \leftarrow i + 1$ 
    | integer  $M[i] = +\infty$ 
    | while  $j \leq n + 1$  and  $D[j] \leq D[i] + r$  do
      | |  $M[i] \leftarrow \min(M[i], M[j])$ 
      | |  $j \leftarrow j + 1$ 
    | |  $M[i] \leftarrow M[i] + C[i]$ 
  return  $M[0]$ 

```

13.8

```

integer studente-pigro(integer[] v, integer[] t, integer n, integer V)
  integer[][] T  $\leftarrow$  newinteger[ $0 \dots n$ ][ $1 \dots V$ ]
  for  $v \leftarrow 1$  to  $V$  do
    |  $T[0, v] \leftarrow +\infty$ 
  for  $i \leftarrow 1$  to  $n$  do
    |  $T[i, v] \leftarrow \min(T[i - 1, v], t[i] + \text{iif}(v - v[i] > 0, M[i - 1, v - v[i]], 0))$ 
  return  $T[n, V]$ 

```

13.10

```

isitalian(ITEM[] s, integer i, integer j, integer[][] M)
  if  $i > j$  then
    | return -1;
  if  $M[i, j] = 0$  then
    | if h.lookup(s[i ... j]) then
      | |  $M[i, j] \leftarrow n + 1$ 
    | else
      | | integer k  $\leftarrow 1$ 
      | |  $M[i, j] = -1$ 
      | | while  $k \leq j - 1$  and  $M[i, j] < 0$  do
        | | | if isitalian(s, i, k, M) and isitalian(s, k + 1, j, M) then
          | | | |  $M[i, j] \leftarrow k$ 
  return  $M[i, j]$ 

```

```
printItalian(ITEM[] s, integer i, integer j, integer[][] M)
```

```

  if  $M[i, j] = n + 1$  then
    | print s[i ... j]
  else if  $M[i, j] < 0$  then
    | print "Non è un testo italiano"
  else
    | | print printItalian(s, i, M[i, j], M) {spazio} printItalian(s, M[i, j] + 1, j, M)

```

```

integer[][] M  $\leftarrow$  new integer[ $1 \dots n$ ][ $1 \dots n$ ]
for  $i \leftarrow 1$  to  $n$  do for  $i \leftarrow 1$  to  $n$  do  $M[i, j] \leftarrow 0$ 
isitalian(s, 1, n, M)
printItalian(s, 1, n, M)

```

13.12

```

integer datacenter(integer[] x, integer[] s, integer n)
  integer[][] P  $\leftarrow$  new integer[ $1 \dots n, 1 \dots n - 1$ ]
  for  $j \leftarrow 1$  to  $n$  do
    |  $P[n, j] \leftarrow \min(x[i], s[j])$ 
  for  $i \leftarrow n - 1$  downto 1 do
    | | for  $j \leftarrow 1$  to  $n - 1$  do
      | | |  $P[i, j] = \max(P[i + 1, 1], \min(x[i], s[j]) + P[i + 1, j + 1])$ 
  return  $P[1, 1]$ 

```

13.17

search-path(integer[][] p, integer n)

```

  { Calcola la tabella g }
  for  $x \leftarrow 1$  to  $n$  do  $g[x, n] \leftarrow p[x, n]$ 
  for  $y \leftarrow n - 1$  downto 1 do
    for  $x \leftarrow 1$  to  $n$  do
      |  $g[x, y] \leftarrow -\infty$ 
      foreach  $d \in \{-1, 0, +1\}$  do
        | | integer  $x' \leftarrow x + d$ 
        | | if  $x' \geq 1$  and  $x' \leq n$  then
          | | | integer  $t \leftarrow g[x', y + 1] + p[x, y]$ 
          | | | if  $t > g[x, y]$  then
            | | | |  $g[x, y] \leftarrow t$ 
            | | | |  $m[x, y] \leftarrow d$ 

```

{ Cerca la casella iniziale con massimo guadagno }

```

  integer x
  for  $i \leftarrow 1$  to  $n$  do
    | if  $g[i, 1] > g[x, 1]$  then
      | |  $x \leftarrow i$ 
  { Stampa il percorso }
  for  $y \leftarrow 1$  to  $n - 1$  do
    | print "(x, y)  $\leftarrow$  (x + m[x, y], y + 1)"
    | |  $x \leftarrow x + m[x, y]$ 

```

13.7

computePredecessor(integer[] a, integer[] b, integer[] p, integer n)

```

   $p[1] \leftarrow 0$ 
  for  $i \leftarrow 2$  to  $n$  do
    | |  $p[i] \leftarrow \text{binarySearch}(b, a_i, 1, i - 1)$ 

```

integer binarySearch(ITEM[] A, ITEM v, integer i, integer j)

```

  if  $i > j$  then
    | | return  $j$ 
  else
    | | integer m  $\leftarrow \lfloor (i + j) / 2 \rfloor$ 
    | | if  $A[m] = v$  then
      | | | return  $m$ 
    | | else if  $A[m] < v$  then
      | | | return binarySearch(A, v, m + 1, j)
    | | else
      | | | return binarySearch(A, v, i, m - 1)

```

13.11

```

calcolaF(ITEM[] s, integer[][] F, integer i, integer j)
  if  $j \leq i$  then
    | | return 0
  else if  $F[i, j] = \perp$  then
    | | | if  $s[i] = s[j]$  then
      | | | |  $F[i, j] \leftarrow \text{calcolaF}(s, i + 1, j - 1)$ 
    | | | else
      | | | |  $F[i, j] \leftarrow \min(\text{calcolaF}(s, i, j - 1), \text{calcolaF}(s, i + 1, j)) + 1$ 
    | | | return  $F[i, j]$ 

```

```

stampa(ITEM[] s, integer[][] F, integer i, integer j)
  if  $s[i] = S[j]$  then
    | | print  $s[i]$  + stampa( $s, f, i + 1, j - 1$ ) +  $s[j]$ 
  else if  $F[i, j] = F[i + 1, j] + 1$  then
    | | print  $s[i]$  + stampa( $s, F, i + 1, j$ ) +  $s[i]$ 
  else
    | | print  $s[j]$  + stampa( $s, F, i, j - 1$ ) +  $s[j]$ 

```

13.15

ferrovia(integer[] d, integer[] f, integer n)

```

  integer[] C  $\leftarrow$  new integer[ $1 \dots n + 1$ ]
  boolean[] B  $\leftarrow$  new boolean[ $1 \dots n$ ]
   $C[n + 1] \leftarrow 0$ 
  for  $i \leftarrow n$  downto 1 do
    | |  $j \leftarrow i$ 
    | | while  $j \leq n$  and  $d[j] < d[i] + 30$  do
      | | |  $j \leftarrow j + 1$ 
      | | |  $B[i] \leftarrow (f[i] + C[i + 1] > X + C[j])$ 
      | | |  $C[i] \leftarrow \text{iif}(B[i], X + C[j], f[i] + C[i + 1])$ 
    | | { Stampa abbonamenti } last  $\leftarrow +\infty$ 
  for  $i \leftarrow 1$  to  $n$  do
    | | if  $B[i]$  and  $d[i] \leq last + 29$  then
      | | | last  $\leftarrow d[i]$ 
      | | | print Abbonamento  $d[i]$ 
  print Costo totale:  $C[1]$ 

```

13.16

componibile(ITEM[] x , ITEM[][] p , integer n , integer m)

```

boolean[]  $T \leftarrow \text{new integer}[0 \dots n]$  % Per ogni lunghezza  $i$ , riporta se è componibile o meno
integer[]  $V \leftarrow \text{new integer}[0 \dots n]$  % Per ogni lunghezza  $i$ , riporta la primitiva che fa da prefisso
 $T[0] = \text{true}$ 
for  $i \leftarrow 1$  to  $n$  do
  integer  $k \leftarrow 1$ 
   $T[i] \leftarrow \text{false}$ 
  while  $k < m$  and  $T[i] = \text{false}$  do
    integer  $j \leftarrow i - |p[k]|$ 
    if  $x_{j+1}x_{j+2} \dots x_i = p[k]$  then
       $T[i] \leftarrow \text{true}$ 
       $V[i] \leftarrow k$ 
  if  $T[n]$  then
    stampa( $p, V, n$ )

```

13.18

GREEDY

SET greedy(SET A)

```

SET  $S \leftarrow \emptyset$ 
{ ordina  $A$  per "appetibilità" decrescente }
foreach  $a \in A$  do
  if  $a$  può essere aggiunto ad  $S$  then
     $S \leftarrow S \cup \{a\}$ 
return  $S$ 

```

TREE huffman(integer[] f , integer[] c , integer n)

```

PRIORITYQUEUE  $Q \leftarrow \text{MinPriorityQueue}()$ 
for  $i \leftarrow 1$  to  $n$  do
   $Q.\text{insert}(f[i], \text{Tree}(f[i], c[i]))$ 
for  $i \leftarrow 1$  to  $n - 1$  do
   $z_1 \leftarrow Q.\text{deleteMin}()$ 
   $z_2 \leftarrow Q.\text{deleteMin}()$ 
   $z \leftarrow \text{Tree}(z_1.f + z_2.f, \text{nil})$ 
   $z.left \leftarrow z_1$ 
   $z.right \leftarrow z_2$ 
   $Q.\text{insert}(z.f, z)$ 
return  $Q.\text{deleteMin}()$ 

```

Complessità: $\Theta(n \log n)$

SET mst-generico(GRAPH G , integer[] w)

```

SET  $A \leftarrow \emptyset$ 
while  $A$  non forma un albero di copertura
  trova un arco sicuro  $[u, v]$ 
   $A \leftarrow A \cup \{[u, v]\}$ 
return  $A$ 

```

SET kruskal(ARCO[] A , integer n , integer m)

```

SET  $T \leftarrow \text{Set}()$ 
MFSET  $M \leftarrow \text{Mfset}(n)$ 
{ ordina  $A[1, \dots, m]$  in modo che  $A[1].peso \leq \dots \leq A[m].peso$  }
integer  $c \leftarrow 0$ 
integer  $i \leftarrow 1$ 
while  $c < n - 1$  and  $i \leq m$  do
  if  $M.\text{find}(A[i].u) \neq M.\text{find}(A[i].v)$  then
     $M.\text{merge}(A[i].u, A[i].v)$ 
     $T.\text{insert}(A[i])$ 
     $c \leftarrow c + 1$ 
   $i \leftarrow i + 1$ 
return  $T$ 

```

Il tempo di esecuzione per l'algoritmo di Kruskal dipende dalla realizzazione della struttura dati per insiemi disgiunti

- Utilizziamo la versione con euristica sul rango + compressione
- L'inizializzazione richiede $O(n)$
- L'ordinamento richiede $O(m \log m) = O(m \log n^2) = O(m \log n)$
- Vengono eseguite $O(m)$ operazioni sulla foresta di insiemi disgiunti, il che richiede un tempo $O(m)$

Totale:

- $O(n+m \log n + m) = O(m \log n)$

boolean stampa(ITEM[][] p , integer[] V , integer i)

```

if  $i = 0$  then return false
if stampa( $p, V, i - |p[V[i]]|$ ) then
  print -
  print  $P[V[i]]$ 
return true

```

13.18

SET independentSet(integer[] a , integer[] b)

```

{ ordina  $a$  e  $b$  in modo che  $b[1] \leq b[2] \leq \dots \leq b[n]$  }
SET  $S \leftarrow \text{Set}()$ 
 $S.\text{insert}(1)$ 
integer  $ultimo \leftarrow 1$  % Ultimo intervallo inserito
for integer  $i \leftarrow 2$  to  $n$  do
  if  $a[i] \geq b[ultimo]$  then % Controllo indipendenza
     $S.\text{insert}(i)$ 
     $ultimo \leftarrow i$ 
return  $S$ 

```

Complessità: $\Theta(n \log n)$

zaino(integer[] p , integer[] v , integer C , integer n , integer[] x)

```

{ ordina  $p$  e  $v$  in modo che  $p[1]/v[1] \geq p[2]/v[2] \geq \dots \geq p[n]/v[n]$  }
integer  $i \leftarrow 1$ 
while  $i \leq n$  and  $C > 0$  do

```

```

  if  $v[i] \geq C$  then
     $x[i] \leftarrow C/v[i]$ 
     $C \leftarrow 0;$ 
  else
     $x[i] \leftarrow 1$ 
     $C \leftarrow C - v[i]$ 
   $i \leftarrow i + 1$ 

```

```

for integer  $j \leftarrow i$  to  $n$  do  $x[j] \leftarrow 0$ 

```

Complessità:
 $O(n \log n)$ per l'ordinamento
 $O(n)$ per la scelta dei valori

prim(GRAPH G , integer r , integer[] p)

```

PRIORITYQUEUE  $Q \leftarrow \text{MinPriorityQueue}()$ 
PRIORITYITEM[]  $pos \leftarrow \text{new PRIORITYITEM}[1 \dots G.n]$ 

```

```

foreach  $u \in G.V() - \{r\}$  do
   $pos[u] \leftarrow Q.\text{insert}(u, +\infty)$ 
 $pos[r] \leftarrow Q.\text{insert}(r, 0)$ 
 $p[r] \leftarrow 0$ 
while not  $Q.\text{isEmpty}()$  do
   $u \leftarrow Q.\text{deleteMin}()$ 
   $pos[u] \leftarrow \text{nil}$ 
  foreach  $v \in G.\text{adj}(u)$  do
    if  $pos[v] \neq \text{nil}$  and  $w(u, v) < pos[v].priorità$  then
       $Q.\text{decrease}(pos[v], w(u, v))$ 
       $p[v] \leftarrow u$ 

```

L'efficienza dell'algoritmo di Prim dipende dalla coda Q

Se Q viene realizzata tramite uno heap binario:

- Inizializzazione: $O(n \log n)$
- Il ciclo principale viene eseguito n volte ed ogni operazione $\text{extractMin}()$ è $O(\log n)$
- Il ciclo interno viene eseguito $O(m)$ volte
- L'operazione $\text{decreaseKey}()$ sullo heap che costa $O(\log n)$

Tempo totale:

- $O(n+n \log n + m \log n) = O(m \log n)$
- asintoticamente uguale a quello di Kruskal

```

moore(integer[] d, integer[] t, integer n, integer[] r)
  PRIORITYQUEUE Q ← MaxPriorityQueue()
  integer k ← 0
  integer T ← 0
  for integer i ← 1 to n do r[i] ← false
  {ordina d[1...n] e t[1...n] per "scadenze" d[i] crescenti}
  for i ← 1 to n do
    Q.insert(i, t[i])
    T ← T + t[i]
    if T ≥ d[i] then
      j ← Q.deleteMax()
      T ← T - t[j]
      r[j] ← true
      k ← k + 1

```

Complessità: O(nlogn)

```

SET fermate(integer[] D, integer n)
  deadline ← 0
  SET stops ← Set()
  for i ← 1 to n do
    if D[i] ≤ deadline and D[i + 1] > deadline then
      stops.insert(i)
      deadline ← D[i] + r
  return stops

```

14.3

RICERCA LOCALE

```

ricercaLocale()
  Sol ← una soluzione ammissibile del problema
  while esiste S ∈ I(Sol) che è migliore di Sol do Sol ← S
  return Sol

```

```

shellSort(ITEM[] A, integer n)
  integer h ← 1
  while h ≤ n do h ← 3 · h + 1
  h ← ⌊h/3⌋
  while h ≥ 1 do
    for integer i ← h + 1 to n do
      ITEM temp ← A[i]
      integer j ← i
      while j > h and A[j - h] > temp do
        A[j] ← A[j - h]
        j ← j - h
      A[j] ← temp
      h ← ⌊h/3⌋

```

Complessità:

O($n^{1.5}$) con $h = (3^i - 1)/2$
O($n^{5/3}$) con $h = 1.72 n^{1/3}$ o 1
O($n \log^2 n$) con $h = 3^j 2^j$
O($n^{1.25}$) nel caso pessimo

```

resto(integer[] t, integer k, integer n, integer[] C, integer[] S)
  C[0] ← 0
  for m ← 1 to n do
    C[m] ← +∞
    for j ← 1 to k do
      if m > t[j] and C[m - t[j]] + 1 < C[m] then
        C[m] ← C[m - t[j]] + 1
        S[m] ← j

```

```

resto(integer[] t, integer k, integer n, integer[] A)
  for i ← 1 to k do
    A[i] ← ⌊n/t[i]⌋
    n ← n - A[i] · t[i]

```

```

stableMarriage(integer n, integer[][] RM, integer[][] RW⁻¹)
  integer[] wife ← new integer[1...n]
  integer[] husband ← new integer[1...n]
  for integer i ← 1 to n do
    wife[i] ← 0
    husband[i] ← 0
    last[i] ← 0
  while ∃m : wife[m] = 0 and last[m] < n do
    last[m] ← last[m] + 1
    w ← RM[m, last[m]]
    if husband[w] = 0 then
      | wife[m] ← w; husband[w] ← m;
    else
      m' ← husband[w]
      k ← RW⁻¹[w, m]
      k' ← RW⁻¹[w, m']
      if k < k' then
        | wife[m] ← w; husband[w] ← m; wife[m'] ← 0;

```

14.5

Ford - Fulkerson: O(|f*|·(m+n))
Edmonds - Karp: O(nm²)
Algoritmo dei tre indiani (Kumar - Malhotra - Maheswari): O(n³)

```

integer[][] maxFlow(GRAPH G, NODE s, NODE p, integer[][] c)
  NODE u, v
  integer[][] f ← new integer[][] % Indici nodi
  integer[][] g ← new integer[][] % Flusso parziale
  foreach u, v ∈ G.V() do f[u, v] = 0 % Inizializza un flusso nullo
  boolean stop ← false
  while not stop do
    R ← Rete di flusso residua del flusso f in G
    g ← flusso associato ad uno o più cammini aumentanti in R
    foreach u, v ∈ G.V() do f[u, v] = f[u, v] + g[u, v] % f = f + g
    if ∀u, v ∈ G.V() : g[u, v] = 0 then stop ← true
  return f

```

```

integer[][] kmn(GRAPH G, NODE s, NODE p, integer[][] c)
  NODE u, v
  GRAPH R ← Graph()
  integer[][] r ← new integer[][] % Indici nodi
  integer[][] f ← new integer[][] % Grafo residuo
  integer[][] g ← new integer[][] % Capacità residua
  foreach u, v ∈ V do f[u, v] ← 0 % Inizializza un flusso nullo
  boolean stop ← false
  while not stop do
    residuo(G, s, p, c, f, R, r)
    saturazione(R, s, p, r, g)
    foreach u, v ∈ V do f[u, v] ← f[u, v] + g[u, v] % f = f + g
    if ∀u, v ∈ V : g[u, v] = 0 then stop ← true
  return f

```

```

integer[][] riempì(integer[] r, integer[] c, integer n)
  integer[][] M ← new integer[1...n, 1...n]
  for i ← 1 to n do
    for j ← 1 to n do
      M[i, j] ← 0
    i ← j ← n
  while i > 0 and j > 0 do
    if r[i] < c[j] then
      M[i, j] ← r[i]
      c[j] ← c[j] - r[i]
      i ← i - 1
    else
      M[i, j] ← c[j]
      r[i] ← r[i] - c[j]
      j ← j - 1
  return M

```

15.4

```

integer satura(GRAPH R, NODE s, NODE p, integer[][] r, integer[][] g)
  NODE u, v                                     % Indici nodi
  integer[] portata ← new integer[1...R.n]          % Vettore portata
  foreach u, v ∈ V do g[u, v] ← 0             % Inizializza g al flusso nullo
  foreach u ∈ R.V() do
    portata[u] ← min{ $\sum_v r[v, u]$ ,  $\sum_v r[u, v]$ }
  while portata[s] > 0 do
    seleziona il nodo u con portata[u] minima
    instrada(R, u, portata, g)
    aggiorna(R, portata, g)
  return g

```

BACKTRACK

```

boolean enumerazione(ITEM[] S, integer n, integer i, ...)
  SET C ← choices(S, n, i, ...)                  % Determina C in funzione di S[1...i - 1]
  foreach c ∈ C do
    S[i] ← c
    if S[1...i] è una soluzione ammissibile then
      if processSolution(S, n, i, ...) then return true
    if enumerazione(S, n, i + 1, ...) then return true
  return false

```

```

subsets(integer[] S, integer n, integer i)
  SET C ← iff(i ≤ n, {0, 1}, ∅)
  foreach c ∈ C do
    S[i] ← c
    if i = n then
      processSolution(S, n)
    subsets(S, n, i + 1)

```

```

subsets(integer[] S, integer n, integer k, integer i, integer count)
  SET C = iff(count < k and count + (n - i + 1) ≥ k, {0, 1}, ∅)
  foreach c ∈ C do
    S[i] ← c
    count ← count + S[i]
    if count = k then
      processSolution(S, i)
    else
      subsets(S, n, k, i + 1, count)
    count ← count - S[i]

```

Complessità: $O(2^n)$

```

SET graham(POINT p1, ..., pn)
  { trova il punto "più basso a destra" e scambialo con p1 }
  { riordina p2, ..., pn in base all'angolo formato rispetto all'asse orizzontale
    quando sono connessi con p1 }
  { elimina gli eventuali punti "allineati" tranne i più lontani da p1, aggiornando n }
  { inserisci p1 nell'inviluppo "corrente" e, se n ≥ 2, inserisci anche p2 }
  for integer i ← 3 to n do
    { siano ph e pj, con h < j = i - 1, gli ultimi due vertici dell'inviluppo "corrente" }
    { scandisci "a ritroso" i punti nell'inviluppo "corrente" ed elimina pj se
      stessaparte(pj, ph, p1, pi) = false;
      termina tale scansione se pj non deve essere eliminato }
    { aggiungi pi all'inviluppo "corrente" }
  return inviluppo "corrente"

```

```

boolean stessaparte(POINT p1, POINT p2, POINT p, POINT q)
  real dx ← p2.x - p1.x
  real dy ← p2.y - p1.y
  real dx1 ← p.x - p1.x
  real dy1 ← p.y - p1.y
  real dx2 ← q.x - p2.x
  real dy2 ← q.y - p2.y
  return ((dx · dy1 - dy · dx1) · (dx · dy2 - dy · dx2) ≥ 0)

```

```

instrada(GRAPH R, NODE u, integer[] portata, integer[][] g)
  % Instrada portata[u] unità di flusso da u a p, modificando g
  % L'instradamento da s a u non è mostrato in quanto simmetrico
  QUEUE Q ← Queue()
  Q.enqueue(u)
  integer[] esci ← new integer[1...R.n]
  foreach v ∈ R.V() do esci[v] ← 0
  esci[u] ← portata[u]
  portata[u] ← 0
  while not Q.isEmpty() do
    v ← Q.dequeue()
    while esci[v] > 0 do
      considera un arco (v, w) uscente da R
      M ← min{r[v, w], esci[v]}
      r[v, w] ← r[v, w] - M
      if r[v, w] = 0 then cancella (v, w) da R
      esci[v] ← esci[v] - M
      esci[w] ← esci[w] + M
      g[v, w] = g[v, w] + M
      g[w, v] = -g[w, v]
      portata[w] = portata[w] - M
      Q.enqueue(w)

```

```

integer ricercaBruta(ITEM[] P, ITEM[] T, integer n, integer m)
  integer i, j, k
  i ← j ← k ← 1
  while i ≤ n and j ≤ m do
    if T[i] = P[j] then i ← i + 1; j ← j + 1
    else k ← k + 1; i ← k; j ← 1
  return iff(j > m, k, i)

```

String matching (Knuth - Morris - Pratt): $O(n+m)$

```

integer kmp(ITEM[] T, ITEM[] P, integer n, integer m)
  integer[] back ← new integer[1...m]
  computeBack(P, back, m)
  integer i ← 1
  integer j ← 1
  while i ≤ n and j ≤ m do
    if j = 0 or T[i] = P[j] then
      i ← i + 1
      j ← j + 1
    else j ← back[j]
  return iff(j > m, i - m, i)

```

```

computeBack(ITEM[] P, integer[] back, integer m)
  back[1] ← 0
  integer j ← 1
  integer h ← 0
  while j ≤ m do
    if h = 0 or P[j] = P[h] then
      j ← j + 1
      h ← h + 1
      back[j] ← h
    else h ← back[h]

```

```

integer graham(POINT[] p)
  integer min ← 1
  for integer i ← 2 to n do
    if p[i].y < p[min].y then min ← i
    p[1] ↔ p[min]
  { riordina p[2, ..., n] in base all'angolo formato rispetto all'asse orizzontale
    quando sono connessi con p[1] }
  { elimina gli eventuali punti "allineati" tranne i più lontani da p1, aggiornando n }
  integer j ← iff(n ≥ 2, 2, 1)
  for integer i ← 3 to n do
    while not stessaparte(p[j], p[j - 1], p[1], p[i]) do
      j ← j - 1
      j ← j + 1
      p[j] ↔ p[i]
  return j

```

Inviluppo convesso (Graham): $O(n \log n)$

```
permutations(SET A, integer n, ITEM[] S, integer i)
```

```
foreach c ∈ A do
  S[i] ← c
  A.remove(c)
  if A.isEmpty() then processSolution(S, n)
  permutations(A, n, S, i + 1)
  A.insert(c)
```

16.4

```
boolean sudoku(integer[][] S, integer i)
```

```
SET C ← Set()
integer x ← i mod 9
integer y ← [i/9]
if i ≤ 80 then
  if S[x, y] ≠ 0 then
    C.insert(S[x, y])
  else
    for integer c ← 1 to 9 do
      if check(S, x, y, c) then C.insert(c)
integer old ← S[x, y]
foreach c ∈ C do
  S[x, y] ← c
  if i = 80 then
    processSolution(S, n)
    return true
  if sudoku(S, i + 1) then return true
S[x, y] ← old
return false
```

```
boolean cavallo(integer[][] S, integer i, integer x, integer y)
```

```
SET C ← mosse(S, x, y)
foreach c ∈ C do
  S[x, y] ← i
  if i = 64 then
    processSolution(S)
    return true
  if cavallo(S, i + 1, x + mx[c], y + my[c]) then
    return true;
  S[x, y] ← 0
return false
```

```
SET mosse(integer[][] S, integer x, integer y)
```

```
SET C ← Set()
for integer i ← 1 to 8 do
  nx ← x + mx[i]
  ny ← y + my[i]
  if 1 ≤ nx ≤ 8 and 1 ≤ ny ≤ 8 and S[nx, ny] = 0 then
    C.insert(i)
return C
```

16.8

```
boolean check(integer[][] S, integer x, integer y, integer c)
```

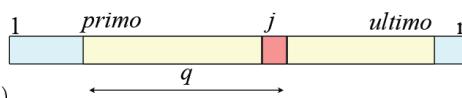
```
for integer j ← 0 to 8 do
  if S[x, j] = c then return false
  if S[j, y] = c then return false
  % Controllo sulla colonna
  % Controllo sulla riga
  integer bx ← [x/3]
  integer by ← [y/3]
  for integer ix ← 0 to 2 do
    for integer iy ← 0 to 2 do
      if S[bx · 3 + ix, by · 3 + iy] = c then return false
      % Controllo sulla sottotabella
return true
```

16.9

ALGORITMI PROBABILISTICI

```
ITEM selezione(ITEM[] A, integer primo, integer ultimo, integer k)
```

```
if primo = ultimo then
  return A[primo]
else
  integer j ← perno(A, primo, ultimo)
  integer q ← j - primo + 1
  if k ≤ q then return selezione(A, primo, j, k)
  else return selezione(A, j + 1, ultimo, k - q)
```



```
boolean primo(integer n)
```

```
integer j = 2
while j ≤ [√n] do
  if n mod j = 0 then return false
  j ← j + 1
return true
```

```
boolean primo(integer n)
```

```
for integer j ← 1 to K do
  b ← random(1, n - 1)
  if b testimonia che n è composto then
    return false
return true
```

```
printOneThird(integer[] A, integer n)
```

```
integer m ← n/3
integer min ← selezione(A, 1, n, m)
integer max ← selezione(A, 1, n, 2m)
for i ← 1 to n do
  if A[i] > min and A[i] < max then
    print A[i]
```

17.4

Selezione in tempo lineare:

$$T(n) = \begin{cases} 11/5n + T(\lfloor n/5 \rfloor) + T(\lceil 7n/10 \rceil) & n > 1 \\ 1 & n = 1 \end{cases}$$

NP-COMPLETEZZA

Certificato:

Verifica la correttezza di una soluzione in tempo polinomiale

```
boolean certificatoCricca(GRAPH G, SET S, integer k)
```

```
if S.size() < k then return false
foreach u ∈ S do
  foreach v ∈ S - {u} do
    if v ∉ G.adj(u) then return false
return true
```

```
boolean certificatoTsp(integer[][] D, integer n, integer k, integer[] P)
```

```
integer t ← D[P[n], P[1]]
for i ← 2 to n do t ← t + D[P[i - 1], P[i]]
return t ≤ k
```

Algoritmi non deterministici:

Seguono parallelamente percorsi differenti

```
ndCricca(GRAPH G, integer k)
```

```
SET S ← Set()
for integer i ← 1 to G.n do
  if choice({true, false}) then
    S.insert(i)
if certificatoCricca(G, S, k) then success else failure
```

```
ndTsp(integer[][] D, integer n, integer k)
```

```
integer[] S ← new integer[1...n]
SET C ← {1, 2, ..., n}
for integer i ← 1 to n do
  S[i] ← choice(C)
  C.remove(S[i])
if certificatoTsp(D, n, k, S) then success else failure
```

```
nonDeterministica({opportuni parametri})
```

```
integer[] S ← new integer[1...n]
for integer i ← 1 to n do
    SET C ← choices(S, n, i,...)           % Determina C in funzione di S[1...i - 1]
    if C = ∅ then
        | failure
    else
        | S[i] ← choice(C)
        | if S[1...i] è una soluzione then
            |   success
```

```
ndZaino(integer[] V, integer[] P, integer n, integer C, integer k)
```

```
SET S ← Set()
for integer i ← 1 to n do
    | if choice ({true, false }) then S.insert(i)
integer Tp ← 0
integer Tv ← 0
foreach i ∈ S do
    | Tp ← Tp + P[i]
    | Tv ← Tv + V[i]
if Tp ≥ k and Tv ≤ C then success else failure
```

18.2

Enumerazione basata su backtrack:

Simula il comportamento di una procedura non deterministica con una deterministica

```
boolean enCricca(GRAPH G, integer k, SET S, integer i)
```

```
SET C ← iff(S.size() < k and S.size() + (G.n - i + 1) ≥ k, {true, false}, ∅)
foreach c ∈ C do
    | if c then S.insert(i)
    | if certificatoCricca(G, S, k) then return true
    | if enCricca(G, k, S, i + 1) then return true
    |   if c then S.remove(i)
return false
```

```
boolean enTsp(integer[] S, SET C, integer i)
```

```
foreach c ∈ C do
    | C.remove(c)
    | S[i] ← c
    | if i = n and certificatoTsp(D, S, n, k) then return true
    | if enTsp(S, C, i + 1) then return true
    |   C.insert(c)
return false
```

PROBLEMI INTRATTABILI

Generalità: per particolari valori di input, il problema potrebbe essere trattabile

- * *Algoritmi pseudo-polinomiali*

Ottimalità: si cercano soluzioni non troppo distanti da quella ottima

- * *Algoritmi di approssimazione*

Efficienza: si cercano soluzioni esponenziali che “potano” lo spazio di ricerca

- * *Algoritmi branch-&-bound*

Formalità: soluzioni che “sembrano” comportarsi bene, anche se non vi è prova matematica del loro comportamento

- * *Algoritmi euristici*

```
apSubsetSum(integer[] A, integer n, integer k)
```

```
{ ordina A in modo che A[1] ≥ A[2] ≥ ⋯ ≥ A[n] }
SET maxSol ← Set()
integer max ← 0
(i) foreach S : S ⊆ {1, ..., n} and |S| ≤ h do
    integer z ← ∑i ∈ S A[i]
    if z ≤ k then
        for integer j ← 1 to n do
            | if j ∉ S and z + A[j] ≤ k then
                |   S.insert(j)
                |   z ← z + A[j]
            | if z ≥ max then
                |   max ← z
                |   maxSol ← S
```

```
bbTsp(ITEM[] S, integer[] C, SET R, integer n, integer i)
```

```
foreach c ∈ R do
    | S[i] ← c
    | R.remove(c)
    | C[i] ← C[i - 1] + d[S[i - 1], S[i]]
    | {calcola A, B, c D[h] per ogni h ∈ R}
    | integer lb ← C[i] + iff(i < n, ⌈(A + B + ∑h ∈ S D[h])/2⌉, d[S[i], S[1]])
    | if lb < minCost then
        |   if i < n then
            |     | bbTsp(S, C, R, n, i + 1,...)
        |   else
            |     | C[n] ← lb
            |     | minSol ← S
            |     | minCost ← C[n]
    | R.insert(c)
```

```
boolean subsetSum(integer[] A, integer n, integer k)
```

```
boolean[] M ← new boolean[0...k]
for integer j ← 1 to k do M[j] ← false
M[0] ← true
for integer i ← 1 to n do
    for j ← k downto A[i] do
        | M[j] ← M[j - A[i]] or M[j]
return M[k]
```

```
branch&bound(ITEM[] S, integer n, integer i, ...)
```

```
SET C ← choices(S, n, i,...)
foreach c ∈ C do
    | S[i] ← c
    | integer lb ← lb(S, i)
    | if lb < minCost then
        |   if i < n then
            |     | branch&bound(S, n, i + 1,...)
        |   else
            |     | if c(S, i) < minCost then
            |       | minSol ← S
            |       | minCost ← c(S, i)
```

```
SET greedyTsp(GRAPH G)
```

```
SET S ← Set()
MFSET M ← Mfset(G.n)
for integer i ← 1 to G.n do in[i] ← 0
{ ordina gli archi per peso non decrescente }
foreach [u, v] ∈ G.E do
    | if in[u] < 2 and in[v] < 2 and M.find(u) ≠ M.find(v) then
        |   S.insert([u, v])
        |   in[u] ← in[u] + 1
        |   in[v] ← in[v] + 1
        |   M.merge(u, v)
integer u ← 1; while in[u] ≠ 1 do u ← u + 1
integer v ← u + 1; while in[v] ≠ 1 do v ← v + 1
S.insert([u, v])
return S
```
