Array

`int binarySearch(Item[] A, Item v, int i, int j)`

```
if i > j then
    return 0
else
    int m ← ⌊(i + j)/2⌋
    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)
```

Ricerca in `A` l'oggetto `v`. I parametri `i` e `j` indicano la parte del sottoinsieme in cui cercare. Assume che il vettore `A` sia ordinato.

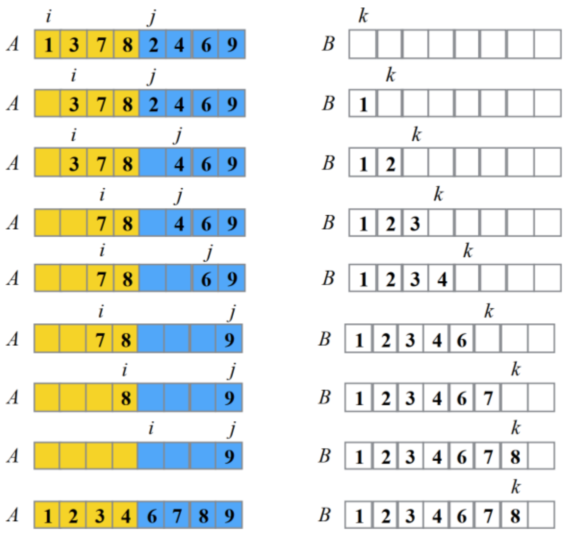
Complessità: ***O(log(n))***

`merge(Item A[], int primo, int ultimo, int mezzo)`

```
int i, j, k, h
i ← primo
j ← mezzo + 1
k ← primo
while i ≤ mezzo and j ≤ ultimo do
    if A[i] ≤ A[j] then
        B[k] ← A[i]
        i ← i + 1
    else
        B[k] ← A[j]
        j ← j + 1
    k ← k + 1
j ← ultimo
for h ← mezzo downto i do
    A[j] ← A[h]
    j ← j − 1
for j ← primo to k − 1 do
    A[j] ← B[j]
```

Unisce, in maniera ordinata, due sottovettori nell'insieme `A`. Assume che i sottovettori `A[primo…mezzo]` e `A[mezzo+1…ultimo]` siano già ordinati.

Complessità: **O(n)**



`sort(Item[] A, int n)`

Una funzione generica, utilizzabile liberamente durante l'esame, che ordina il vettore `A`. Il parametro n è il numero di elementi nell'array.

Complessità: ***O(nlog(n))***

Alberi

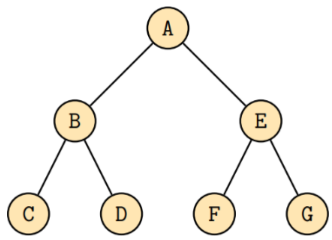
`dfs(Tree t)`

if $t \neq$ **nil then**
  % pre-order visit of t
  print t
  dfs(t.left())
  % in-order visit of t
  print t
  dfs(t.right())
  % post-order visit of t
  print t

Visita in profondità l'albero radicato `t`. Può essere implementato in pre/in/postvisita.

Complessità: ***O(n)***

Previsita: A B C D E F G
Invisita: C B D A F E G
Postvisita: C D B F G E A



`bfs(Tree t)`

QUEUE $Q \leftarrow$ Queue()
Q.enqueue(t)
while not Q.isEmpty() **do**
  TREE $u \leftarrow Q$.dequeue()
  % visita per livelli dal nodo u
  print u
  $u \leftarrow u$.leftmostChild()
  while $u \neq$ **nil do**
    Q.enqueue(u)
    $u \leftarrow u$.rightSibling()

Visita l'albero radicato `t` in ampiezza (per livelli).

Complessità: ***O(n)***

`int count(Tree t)`

if $T ==$ **nil then**
  return 0
else
  $C_\ell =$ count(T.left())
  $C_r =$ count(T.right())
  return $C_\ell + C_r + 1$

Conta il numero di nodi che possiede un albero binario.

Complessità: ***O(n)***

Hashtable

Costi delle operazioni delle tabelle hash in base all'implementazione:

	Array non ordinato	**Array ordinato**	**Lista**	**Alberi RB**	**Implemen. ideale**
insert()	$O(1), O(n)$	$O(n)$	$O(1), O(n)$	$O(\log n)$	$O(1)$
lookup()	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(1)$
remove()	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
foreach	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Grafi

```
bfs(Graph G, Node r)
```

QUEUE S = Queue( )
S.enqueue(r)
boolean[ ] $visited$ = **new boolean**[$1 \ldots G.n$]
foreach $u \in G$.V() $- \{r\}$ **do**
  $visited[u]$ = **false**
$visited[r]$ = **true**
while not S.isEmpty() **do**
  NODE u = S.dequeue()
  { visita il nodo u }
  foreach $v \in G$.adj(u) **do**
    { visita l'arco (u, v) }
    if not $visited[v]$ **then**
      $visited[v]$ = **true**
      S.enqueue(v)

Visita il grafo (non) orientato per livelli (in ampiezza), a partire dal nodo `r`.

Complessità: ***O(n+m)***

```
erdos(Graph G, Node r, int[] erdős, Node[] p)
```

QUEUE S = Queue()
S.enqueue(r)
foreach $u \in G$.V() $- \{r\}$ **do**
  $erdős[u] = \infty$
$erdős[r] = 0$
$p[r]$ = **nil**
while not S.isEmpty() **do**
  NODE u = S.dequeue()
  foreach $v \in G$.adj(u) **do**
    if $erdős[v] == \infty$ **then**
      $erdős[v] = erdős[u] + 1$
      $p[v] = u$
      S.enqueue(v)

Un'applicazione della visita bfs. Visita il grafo a partire dal nodo `r`. Salva per ciascun nodo del vettore `erdős` la sua distanza del nodo r. Per recuperare la distanza di u dal nodo r:

```
integer distanza <- erdős[u]
```

Ciascun elemento del vettore `p` contiene il nodo "padre" passando per il quale l'algoritmo lo ha trovato. Il vettore `p` è spesso utilizzato per recuperare il percorso più breve per arrivare da un nodo qualsiasi al nodo `r`; un esempio ne è la funzione `printPath()`. Il nodo `r` ha padre **`nil`**.

Complessità: ***O(n+m)***

```
printPath(Node r, Node s, Node[] p)
```

if $r == s$ **then**
  print s
else if $p[s]$ == **nil then**
  print "error"
else
  printPath($r, p[s], p$)
  print s

Stampa il percorso che porta dal nodo `s` all radice `r`, utilizzando il vettore dei padri `p` generato dalla la funzione `erdos()`.

Complessità: ***O(n)***

```
dfs(Graph G, Node r)
```

```
STACK S ← Stack()
S.push(r)
boolean[] visited = new boolean[1 ... G.n]
foreach u ∈ G.V() − {r} do
    visited[u] = false
visited[r] = true
while not S.isEmpty() do
    NODE u = S.pop()
    { visita il nodo u (pre-order) }
    foreach v ∈ G.adj(u) do
        if not visited[v] then
            { visita l'arco (u, v) }
            visited[v] = true
            S.push(v)
```

Vista in profondità del grafo `G`, a partire dal nodo `r`; versione iterativa con lo stack esplicito.

Complessità: ***O(n+m)***

```
dfs(Graph G, Node u, boolean[] visited)
```

```
visited[u] = true
{ visita il nodo u (pre-order) }
foreach v ∈ G.adj(u) do
    if not visited[v] then
        { visita l'arco (u, v) }
        dfs(G, v, visited)
{ visita il nodo u (post-order) }
```

Vista in profondità del grafo `G`, a partire dal nodo `r`; versione ricorsiva con lo stack implicito. Richiede anche un vettore in cui si memorizzerà quali sono nodi saranno visitati.

Complessità: ***O(n+m)***

```
int[] cc(Graph G)
```

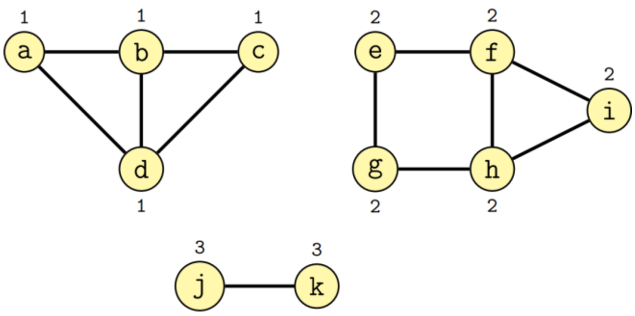
```
int[] id = new int[1 ... G.n]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
foreach u ∈ G.V() do
    if id[u] == 0 then
        counter = counter + 1
        ccdfs(G, counter, u, id)
return id
```

```
ccdfs(GRAPH G, int counter, NODE u, int[] id)
```

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

Ricerca le componenti connesse di un grafo (non) orientato. Ritorna un vettore che assegna a ciascun nodo un numero che identifica la componente connessa.

Complessità: ***O(n+m)***

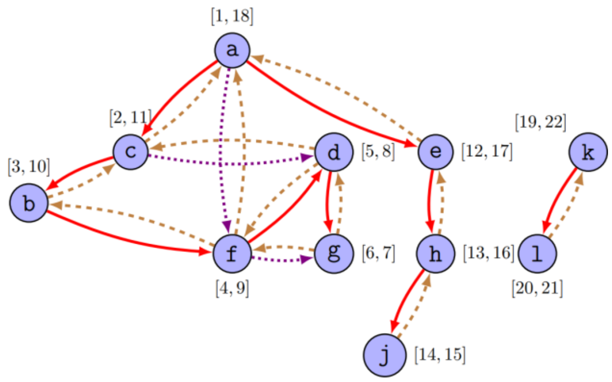


```
dfs-schema(Graph G, Node u, int &time, int[] dt, int[] ft)
```

```
{ visita il nodo u (pre-order) }
time = time + 1;   dt[u] = time
foreach v ∈ G.adj(u) do
    { visita l'arco (u, v) (any kind) }
    if dt[v] = 0 then
        { visita l'arco (u, v) (tree edge) }
        dfs-schema(G, v, time, dt, ft)
    else if dt[u] > dt[v] and ft[v] = 0 then
        { visita l'arco (u, v) (back edge) }
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        { visita l'arco (u, v) (forward edge) }
    else
        { visita l'arco (u, v) (cross edge) }
{ visita il nodo u (post-order) }
time = time + 1;   ft[u] = time
```

Esegue una visita del grafo (non) orientrato, badando a classificare ciascun arco come arco dell'albero di copertura/all'indietro/in avanti/di attraversamento. Memorizza nei vettori `dt` e `ft` i tempi di scoperta e di fine di ciascun nodo.

Complessità: ***O(n+m)***




```
boolean hasCycle(Graph G, Node u, int &time, int[] dt, int[] ft)
```

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

Ritorna **`true`** se il grafo orientato passato ha dei cicli (o archi all'indietro), altrimenti **`false`**.

Complessità: ***O(n+m)***

```
Stack topSort(Graph G)
```

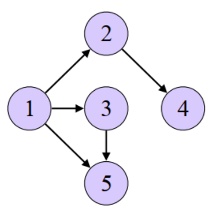
```
STACK S ← Stack()
boolean[] visited = boolean[1 ... G.size()]
foreach u ∈ G.V() do visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        ts-dfs(G, u, visited, S)
return S
```

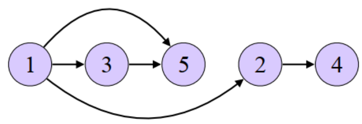
```
ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
```

```
visited[u] = true
foreach v ∈ G.adj(u) do
    if not visited[v] then
        ts-dfs(G, v, visited, S)
S.push(u)
```

Ordina topologicamente un grafo orientato aciclico. Ritorna uno stack il qui elemento in cima è il primo elemento dell'ordinamento.

Complessità: ***O(n+m)***



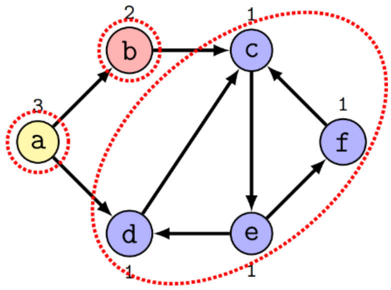


```
int[] scc(Graph G)
```

$\text{Stack } S = \mathsf{topSort}(G)$
$\text{Graph } G^T = \mathsf{Graph}()$
foreach $u \in G.\mathsf{V}()$ **do** $G^T.\mathsf{insertNode}(u)$
foreach $u \in G.\mathsf{V}()$ **do**
  foreach $v \in G.\mathsf{adj}(u)$ **do**
    $G^T.\mathsf{insertEdge}(v, u)$

return $\mathsf{cc}(G^T, S)$

Dato un grafo orientato ritorna un vettore delle componenti fortemente connesse. Una componente è fortemente connessa se e solo se esiste un sottografo fortemente connesso (un grafo in cui ogni nodo è raggiungibile da ogni altro suo nodo).



Complessità: ***O(n+m)***