```
BinarySearch(L,i,j,key)
 if j-i < 0
  then
   return -1
  else
   k := ⌊(i+j)/2⌋
   if key = L[k]
    then
     return k
   if key < L[k]
    then
     return BinarySearch(L,i,k-1,key)
    else
     return BinarySearch(L,k+1,j,key)
```

```
BinarySearch(L,n,key)
i := 0
j := n -1
while i ≤ j
 k := ⌊(i+j)/2⌋
 if key = L[k]
  then
   return k
 if key < L[k]
  then
   j := k-1
  else
   i := k+1
return -1
```

```
SelectionSort(A,i,n)
 if i < n-1                    O(n^2)
  then
   k := indice dell'elemento minimo in A[i..n-1]
   inverti A[i] e A[k]
   SelectionSort(A,i+1,n)
```

```
IndiceDelMinimo(A,i,n)
k := i
for j = i+1 to n-1
 if A[j] < A[k]
  then
   k := j          teta(n-1)
return k
```

```
MERGESORT(A,i,j)
 if i < j                O(n log n)
  then
   k := ⌊(i+j)/2⌋
   MERGESORT(A,i,k)
   MERGESORT(A,k+1,j)
   MERGE(A,i,k,j)
```

```
InsertionSort(A,n)
for j = 1 to n-1 do
 i := j-1                      teta(n^2)
 tmp := A[j]
 while (i≥0 AND A[i]>tmp) do
  A[i+1] := A[i]
  i := i-1
 A[i+1] := tmp
```

```
PARTITION(A,p,r)
x := A[r]
i := p-1
for j = p to r-1
 if A[j] ≤ x
  then
   i := i+1
   scambia A[i] e A[j]
scambia A[r] e A[i+1]
return i+1
```

```
QUICKSORT(A,p,r)
 if p < r                  bil O(n log n) sbil O(n^2)
  then
   q := PARTITION(A,p,r)
   QUICKSORT(A,p,q-1)
   QUICKSORT(A,q+1,r)
```

```
TopologicalSort(G)
S := new_stack()
for all v ∈ V
 visited[v] := FALSE
for all v ∈ V
 if visited[v] = FALSE
  DFS-TS(G,v)
return S
```

```
DFS-TS(G,v)
 visited[v] := TRUE
 for all (v,u) ∈ E
 // archi uscenti da v
  if visited[u] = FALSE
   then DFS-TS(G,u)
 push(S,v)
```

```
DFS_pre_order(t)
if NOT (t = NIL)
 then
  // analisi di t
  print t.val
  DFS_pre_order(t.left)
  DFS_pre_order(t.right)
```

```
BFS(T)                          O(n * costoxnodo)
if (NOT T = NIL)
 Q := new_queue()
 enqueue(Q,T)
 while NOT is_empty_queue(Q)
  u := dequeue(Q)
  // analisi di u
  print u.val
  if NOT (u.left = NIL)
   then enqueue(Q,u.left)
  if NOT (u.right = NIL)
   then enqueue(Q,u.right)
```

```
BFS(T)
if (NOT T = NIL)
 Q := new_queue()
 enqueue(Q,T)
 while NOT is_empty_queue(Q) do
  u := dequeue(Q)
  // analisi di u
  print u.val
  for all v ∈ children(t) do
   enqueue(Q,v)
```

```
DFS_pre_order(t)
if NOT (t = NIL)
 then
  // analisi di t
  print t.val
  for all u ∈ children(t) do
   DFS_pre_order(u)
```

```
DFS_pre_order(t)
if NOT (t = NIL)
 then
  // analisi di t
  print t.val
  w := t.fistchild
  while (NOT w = NIL) do
   DFS_pre_order(w)
   w := w.nextsibling
```

```
BFS(T)
if (NOT T = NIL)
 then
  Q := new_queue()
  enqueue(Q,T)
  while NOT is_empty_queue(Q) do
   u := dequeue(Q)
   // analisi di u
   print u.val
   w := u.fistchild
   while (NOT w = NIL) do
    enqueue(Q,w)
    w := w.nextsibling
```

```
DFS(G)                          O(|V|+|E|)
 for all v ∈ V
  visited[v] := FALSE
 prev[v] := 0
 for all v ∈ V
  if visited[v] = FALSE
   then DFS-Visit(G,v)
 return prev[]
```

```
DFS-Visit(G,v)
 visited[v] := TRUE
 for all (v,u) ∈ E
 // archi incidenti in v
  if visited[u] = FALSE
   then
    prev[u] := v
    DFS-Visit(G,u)
```

```
DFS(G)                          O(|V|+|E|)
 for all v ∈ V
  visited[v] := FALSE
 for all v ∈ V
  if visited[v] = FALSE AND DFS-Visit(G,v)
   then return TRUE
 return FALSE
```

```
DFS-Visit(G,v)
 visited[v] := TRUE
 for all (v,u) ∈ E
 // archi incidenti in v
  if visited[u] = TRUE OR DFS-Visit(G,u)
   then return TRUE
 return FALSE
```

```
COUNTINGSORT(A,k)
 n := length(A) // cardinalità di A
 crea C[0..k]
 for i = 0 to k // inizializzazione C
  C[i] := 0
 for j = 0 to n-1
  C[A[j]]:= C[A[j]] + 1
 //calcolo occorrenze: C[i] = numero di
 // occorrenze di i
 j = 0
 for i = 0 to k // "riempimento" di A
   while C[i] > 0
    A[j] := i
    j := j + 1
    C[i] := C[i] - 1
```

non stabile O(n+k)

```
Dijkstra(G,c,s)
 for all u ∈ V                  O((|V|+|E|)*log|V|)
  dist[u] := +∞
  prev[u] := 0
dist[s] := 0

Q := Make_priority_queue({(v,dist[v])|v = 1,…,n})
                          // coda con priorità:
                          //  nodi di V con pri

while NOT is_empty_queue(Q) do
 u:= DeQueue(Q) // restituisce l'elemento con pric
                // massima (valore minimo) e lo
                // dalla coda
 for all (u,v)  E do
  if dist[v] > dist[u] + c(u,v)
   then
    dist[v] := dist[u] + c(u,v)
    prev[v] := u
    Decrease_Priority(Q,v,dist[v])
return prev[]
```

```
Bellman-Ford(G,c,s)
 for all u ∈ V                  O(|V||E|)
  dist[u] := +∞
  prev[u] := 0                  INIZIALIZZAZIONE
dist[s] := 0

for i=1 to |V|-1 do
 for all (u,v) ∈ E do
  if dist[v] > dist[u] + c(u,v)
   then
    dist[v] := dist[u] + c(u,v)
    prev[v] := u

for all (u,v) ∈ E do
 if dist[v] > dist[v] + c(u,v)
  then return "esiste un ciclo negativo"

return prev[]
```

```
BFS(G,s)
 for all u ∈ V
  dist[u] := +∞
  prev[u] := -1
 dist[s] := 0
 Q := new_queue() // coda FIFO
 enqueue(Q,s)
 while NOT is_empty_queue(Q)do
  u:= dequeue(Q)
  eventuale esame di u
  for all (u,v) ∈ E do
   if dist[v] = +∞
    then
     enqueue(Q,v)
     dist[v] := dist[u] + 1
     prev[v] := u
```

```
Kruscal(G=(V,E),c)
S := make_set(V) //disjoint set
T := new_list()
ordina gli archi di E per ordine non decrescente di costo c
count := 0
while count < n - 1 do
 scegli il prossimo arco (u,v)∈ E nell'ordinamento
 if find-set(S,u) ≠ find-set(S,v)
  then p := new_list_node()
       p.val := (u,v)
       insert-head(T,p)
       union(S,u,v)
       count := count + 1
return T
```

O(m log n)

```
Prim(G=(V,E),c)
for all v ∈ V do
  cost[v] := +∞
  prev[v] := NIL
  S[v] := 0
scegli un nodo s ∈ V
cost[s] := 0
S[s] := 1
Q := make_priority_queue(V') // coppie nodi di V e cost[]
while NOT is_empty_queue(Q) do
  u := DeQueue(Q)
  S[u] := 1
  for all (u,v)∈ E do
    if S[v] = 0 AND cost[v] > c(u,v)
      then
        cost[v] := c(u,v)
        prev[v] := u
        Decrease_Priority(Q,v,cost[v])
return prev[]
```

Prim O(m log n)