Insertion sort

L'idea generale: data un vettore con la parte sinistra, A[0..i−1], ordinata, è facile inserirci l'elemento A[i] in modo tale che il sottovettore A[0..i] risulti ordinata (aumentando così la parte ordinata). Punto di partenza: i = 1 (così la parte a sinistra di A[i] contiene un singolo elemento e quindi è ordinata).

```
Insertion-Sort(A[0…n])
for i ← 1 to n do
    j ← i
    while j > 1 and A[j-1] > A[j] do
        Scambia A[j-1] con A[j]
        j ← j-1
    end while
end for
```

Invariante ciclo for: `A[0…i-1]` è ordinato.
Valido per caso base `i = 1` perché ci sarà 1 solo elemento nel vettore di sinistra.
Valido per il resto del ciclo perché gli elementi più grandi dell'elemento corrente verranno spostati verso destra.

Invariante ciclo while: per quanto riguarda il segmento `A[1 .. i]`, togliendo `A[j]` si ottiene un vettore ordinato e gli elementi dopo `A[j]` sono maggiori di `A[j]`.
Valido per caso base grazie all'invariante del ciclo esterno.
Valido per il resto del ciclo perché:

- Se `A[j-1] ⩽ A[j]` oppure `j = 1` si esce dal ciclo, quindi il sottovettore `A[0…i]` è ordinato
- Se `A[j-1] > A[j]` i 2 elementi si scambiano e l'invariante rimane valido

L'algoritmo è **quadratico nel caso peggiore**: il tempo di esecuzione nel caso peggiore è proporzionale al quadrato del numero di elementi del vettore. $T_{cp}(n) \approx an^2$
L'algoritmo è **lineare nel caso migliore**: il tempo di esecuzione nel caso migliore è proporzionale al numero di elementi del vettore. $T_{cm}(n) \approx an$

Selection Sort

L'idea generale: data un vettore in cui la parte sinistra, A[0..i − 1], è ordinata e contiene gli `i - 1` numeri più piccoli del vettore, si cerca il minimo della parte `A[i .. n]` e si mette nella posizione i (aumentando così la parte ordinata). Punto di partenza: `i = 0` (così la parte a sinistra di `A[i]` è un vettore "vuoto").

```
Selection-Sort(A[0…n])
for i ← 0 to n-1 do
    k ← i
```

```
    for j ← i+1 to n do
        if A[k] > A[j] then
            k ← j
        end if
    end for
    Scambia A[i] con A[k]
end for
```

Invariante del ciclo for esterno: `A[0…i-1]` è ordinato e per ogni k, l (con $0 \leq k \leq i-1$ e $i \leq l \leq n$) $A[k] \leq A[l]$
Valido ad inizializzazione, perché partendo da `i = 0` la proposizione è "vuota" quindi valida.
L'invariante è mantenuto all'interno del ciclo se il ciclo interno riesce a prendere correttamente il minimo in `A[i…n]`
Valido all'uscita perché l'invariante implica "A[0…n-1] è ordinato e $A[n-1] \leq A[n]$", quindi il vettore è ordinato
Invariante del ciclo for interno: `A[k]` è il minimo in `A[i…j-1]`.
Valido ad inizializzazione perché `k=i`, `j=i+1` quindi `A[k] = A[i] = A[j-1]`.
L'invariante viene mantenuto all'interno del ciclo perché il ciclo aggiorna la posizione del minimo se `A[k] > A[j]` e poi incrementa j.
All'uscita l'invariante implica che `A[k]` è il minimo in `A[i…n]` e quindi il ciclo funziona.

L'algoritmo è quadratico sia nel caso migliore sia nel caso peggiore, in quanto si eseguono `n*n controlli` (n istanze del ciclo interno per ogni istanza del ciclo esterno).

Partizione costante

Esempio:

```
Fattoriale(n)
if n < 2 then return 1
else return n *  Fattoriale(n-1)
```

N° di operazioni: 1 se n < 2, 1 + T(n-1) se n >= 2
La relazione di ricorrenza sviluppata è quindi:

$$T(n) = \begin{cases} 1, & n < 2 \\ 1 + T(n-1), & n \geq 2 \end{cases}$$

Ci sono 2 approcci per trovare la soluzione di una relazione di ricorrenza: **iterazione** e **sostituzione.**

Iterazione: consiste nello sviluppare la relazione fino ad intuire la soluzione.

$$T(n) = \begin{cases} c, & n = 0 \\ d + T(n-1), & n \geq 1 \end{cases}$$

Usiamo questa relazione di ricorrenza diversa perché più generica. Vengono sostituite le costanti di ogni riga con simboli diversi, e le condizioni vengono cambiate per includere un caso base per cui fermarsi.

$$
\begin{aligned}
T(n) &= T(n-1) + d \\
&= T(n-2) + d + d \\
&= T(n-3) + d + d + d \\
&= T(n-k) + kd && \text{dopo } k \leq n \text{ iterazioni} \\
&= T(0) + nd && \text{dopo } n \text{ iterazioni} \\
&= c + nd
\end{aligned}
$$

La soluzione è quindi $T(n) = c + nd \in \Theta(n)$
Sostituzione: si ipotizza una soluzione e la si verifica tramite induzione.
Ipotizziamo $T(n) = c + nd$ come soluzione.
Verifichiamola per il caso base $n = 0$: $T(0) = c + 0 * d = c = T(0)$ quindi è valido.
Per induzione, $T(n) = c + nd$ è valido quindi verifichiamo che $T(n+1) = c + (n+1)d$
Usando la relazione di ricorrenza:

$$
\begin{aligned}
T(n+1) &= T((n+1) - 1) + d \\
&= T(n) + d \\
&= c + nd + d \\
&= c + (n+1)d
\end{aligned}
$$

Quindi la soluzione è corretta.

Quick Sort

Si seleziona un elemento del vettore (il "perno") attorno al quale organizzare gli elementi; gli elementi <= al perno vengono posizionati a sinistra, quelli > del perno vengono posizionati a destra, ripetendo l'operazione per le partizioni a sinistra e destra del perno.

```
Quick-Sort(A[0…n])
if(n > 0) then
    p ← Partition(A[0…n])
    if(p > 1) then (se prima del perno ci sono almeno 2 elementi)
        Quick-Sort(A[0…p-1])
    end if
    if(p < n-1) then (se dopo il perno ci sono almeno 2 elementi)
        Quick-Sort(A[p+1…n])
    end if
end if

Partition(A[0…n])
perno ← A[0]
i ← 1
j ← n
```

```
while i ⩽ j do
    if(A[i] ⩽ perno) then
        i ← i+1
    else
        If(A[j] > perno) then
            j ← j-1
        else
            Scambia A[i] con A[j]
            i ← i+1
            j ← j-1
        end if
    end if
end while
Scambia A[0] con A[j]
return j
```

$T_p(n) = an$

Caso peggiore:

$$T(n) = \begin{cases} c, & n = 0 \\ T(n-1) + T_p(n) + b, & n > 0 \end{cases} = \begin{cases} c, & n = 0 \\ T(n-1) + an + b, & n > 0 \end{cases}$$

Metodo dell'iterazione:

$$\begin{aligned} T(n) &= T(n-1) + an + b \\ &= T(n-2) + a(n-1) + b + an + b \\ &= T(n-3) + a(n-2) + b + a(n-1) + b + an + b \\ &= T(n-k) + a\sum_{i=0}^{k-1}(n-i) + kb && \text{dopo } k \leq n \text{ iterazioni} \\ &= T(0) + a\sum_{i=0}^{n-2}(n-i) + (n-1)b && \text{dopo } n \text{ iterazioni} \\ &= c + a\sum_{i=0}^{n-2}(n-i) + (n-1)b \end{aligned}$$

$T(n) \in \Theta(n^2)$ quindi Quick Sort è quadratico nel caso peggiore

Il seguente teorema fornisce una "ricetta" per trovare un limite superiore in senso O data una relazione di ricorrenza di ordine costante.
Siano $a_1, \ldots, a_h$ costanti intere non negative, c, d e β costanti reali positive tale che

$$T(n) = \begin{cases} d, & se\ n \leq m \leq h \\ \sum_{i=1}^{h} a_i T(n-i) + cn^{\beta}, & x \geq 0 \end{cases}$$

Posto $a = \sum_{i=1}^{h} a_i$

$T(n) \in O(n^{\beta+1})$ se $a = 1$

$T(n) \in O(n^{\beta} a^{n})$ se ≥ 2

Counting sort

L'idea è di contare quante volte un intero occorra in `A[1 .. n]` in tempo `O(n)` usando un array `C[0 .. k]` di interi, e poi di copiare in `A[1 .. n]`, `m = C[j]` copie di `j` per ogni `j` $\in$ `0 .. k`.

```
Counting-Sort(A[1 .. n], k)   # pre: ∀p ∈ A[1..n]0 ≤ p ≤ k
C[0 .. k] array di interi
for i ← 0 to k do
      C[i] ← 0
end for
for i ← 0 to n do
      C[A[I]] ← C[A[i]] + 1
end for     # ∀j ∈ 0..k, C[j] = il numero di volte che j occorre in
A[1 .. n]
j ← 0
for i ← 1 to n do
      if C[j] > 0 then
            A[i] ← j, C[j] ← C[j] - 1
      else j ← j + 1
      end if
end for
```

$T_{CountSort}(n, k) = O(n) + O(k) = O(max(n, k))$

N.B: *k* è calcolabile in tempo *O(n)*, ma tra *n* e *k* non vi è nessuna relazione!

Merge Sort

L'idea:

- un vettore che contiene un elemento è ordinato;
- se il vettore contiene più di un elemento allora viene diviso in due parti bilanciate;
- ognuna di queste due parti viene ordinata applicando ricorsivamente l'algoritmo;
- le due parti vengono fuse

f = first

m = middle

l = last

n = numero di elementi in A

Precondizione: $1 \leq f \leq m \leq l \leq n \wedge$ A[f…m] e A[m+1…l] sono ordinati

```
Merge-Sort(A[1…n])
if n > 1 then
      m ← n/2
      Merge-Sort(A[1…m])
      Merge-Sort(A[m+1…n])
      Merge(A, 1, m, n)
end if
```

```
Merge(A, f, m, l)
i, j, k ← f, m+1, 1
while i ⩽ m ∧ j ⩽ l do
      if A[i] ⩽ A[j] then
            B[k] ← A[i]
            i ← i+1
      else
            B[k] ← A[j]
            j ← j+1
      end if
      k ← k+1
end while
if i ⩽ m then
      B[k…k+m-i] ← A[i…m]
else
      B[k…k+l-j] ← A[j…l]
end if
A[f,l] ← B[1…l-f+1]
```

La funzione Merge che unisce i due array può essere effettuata anche in maniera ricorsiva:

```
Merge-Rec(B[1…n_B], C[1…n_C])
if n_B = 0 then
      return C
else
      if n_C = 0 then
            return B
      else
            if B[1] ⩽ C[1] then
                  return [B[1], Merge-Rec(B[2…n_B], C)]
            else
                  return [C[1], Merge-Rec(B, C[2…n_C])]
            end if
```

```
    end if
end if
```

Tabelle Hash

Tavole a indirizzamento diretto

Le tavole a indirizzamento diretto sono un'idea preliminare a quella delle hash table.

Sia U l'universo delle chiavi: `U = {0, 1, ..., m - 1}`. L'insieme dinamico viene rappresentato con un array `T` di dimensione `m` in cui ogni posizione corrisponde ad una chiave. `T` è la tavola a indirizzamento diretto perché ogni sua cella corrisponde direttamente ad una chiave.

Le operazioni sono tutte O(1):

```
TableInsert(T, x)
T[x.key] ← x

TableDelete(T, x)
T[x.key] ← nil

TableSearch(k)
return T[k]
```

Anche se sembra efficiente, questa struttura dati spreca moltissima memoria.

Tavole hash

Utilizziamo una tabella T di dimensione `m` con `m` molto più piccolo di `|U|`. La posizione della chiave
k è determinata utilizzando una funzione
$h: U \to \{0, 1, \dots, m - 1\}$
chiamata la funzione hash.

L'indirizzamento in questa maniera non è più diretto, e l'elemento con chiave k si trova nella posizione **h(k)**.

Conseguenze:

- riduciamo lo spazio utilizzato
- perdiamo la diretta corrispondenza fra chiavi e posizioni
- $m < |U|$ e quindi inevitabilmente **possono esserci delle collisioni**

Una buona funzione hash posizione le chiavi in modo **apparentemente** uniforme e casuale, riducendo il numero di collisioni.

Un hash perfetto è una funzione che non crea mai collisioni.

Tavole hash con concatenamento

Per evitare collisioni, si potrebbero concatenare gli elementi in collisione in una lista.

```
HashInsert(T, x)
L ← T[h(x.key)]

ListInsert(L, x)

HashSearch(T, k)
L ← T[h(k)]
return ListSearch(L, k)

HashDelete(T, x)
L ← T[h(x.key)]

ListDelete(L, x)
```

Tutte le operazioni richiedono O(1) tempo, se la lista usata è doppiamente concatenata.

Indirizzamento aperto

Tutti gli elementi vengono memorizzati nella tavola *T*. L'elemento con chiave *k* viene inserito nella posizione *h(k)* se essa è libera. Se non lo è, si cerca la prima posizione libera seconda uno **schema di ispezione**. Il più semplice è l'**ispezione lineare**, per cui l'elemento viene inserito nella prima cella libera a disposizione.

Inserzione

```
HashInsert(T, x)
i ← 0
while i < m do
      j ← h(x.key, i)
      if T[j] = nil then
            T[j] ← x
            return j
      end if
      i ← i + 1
end while
return nil
```

Ricerca

```
HashSearch(T, k)
i ← 0
while i < m do
      j ← h(k, i)
      if T[j] = nil then
            return nil
      end if
      if T[j].key = k then
            return T[j]
      end if
      i ← i + 1
end while
return nil
```

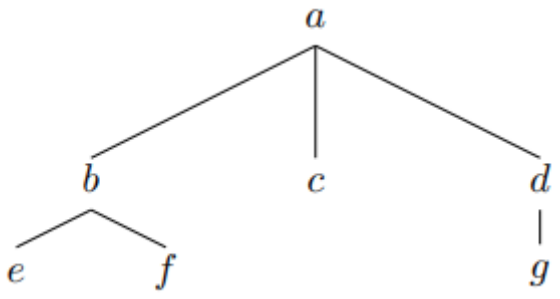
La cancellazione diventa più difficile: o si marca la posizione con **deleted**, o si modifica la procedura d'inserimento. Per questo questa struttura dati si usa quando non c'è necessità di cancellare.

Alberi

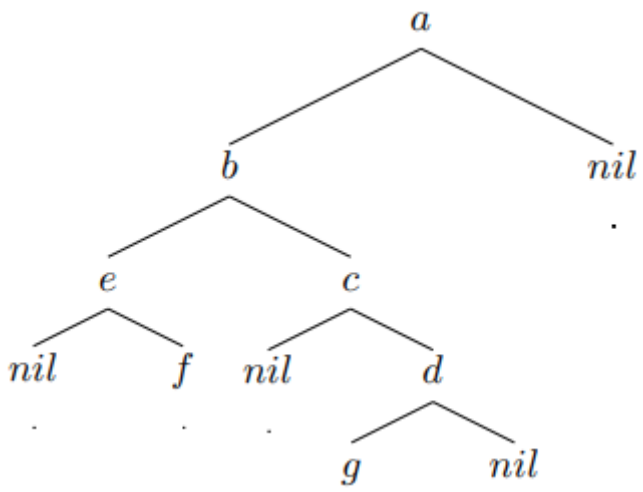
Un albero è un **grafo connesso** (esiste un cammino fra qualunque coppia di nodi) **aciclico** (partendo da un nodo non si può tornare a se stesso con nessun cammino). Una **foresta** è un insieme di alberi.

La ***radice*** di un albero è il nodo iniziale. Un nodo può avere dei ***figli***, e ogni figlio ha un ***padre***. Più nodi su una stessa riga sono considerati ***fratelli***. Un nodo è una ***foglia*** se non ha alcun figlio. Un cammino dalla radice ad una foglia è un ramo. Il ***livello*** di un nodo è il numero degli archi del cammino che porta dalla radice (livello 0) al nodo (livello n). L'***altezza*** di un albero è il livello del nodo con livello massimo. Il ***grado*** è il numero di figli del nodo che ha più figli. L'albero viene detto ***ordinato*** se l'ordine in cui appaiono i figli di un nodo conta.

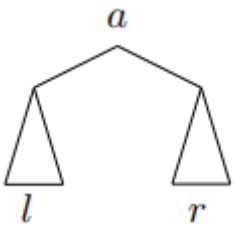
Rappresentazione naturale di un albero di grado k: ogni nodo contiene l'etichetta e k puntatori che fanno riferimenti ai figli (se un nodo ha meno di k gli allora una parte dei puntatori è `nil`).



Rappresentazione binaria di un albero di grado k: ogni nodo contiene l'etichetta, un primo puntatore al primo figlio (detto **child**) e un secondo puntatore al fratello successivo (detto **sibling**). (Naturalmente, anche in questo caso, child e/o sibling possono essere nil.)



Gli **alberi binari posizionali** sono alberi binari (cioè di grado 2) in cui conta l'ordine dei nodi. Ogni nodo contiene l'etichetta e due puntatori, *left* e *right*, che fanno riferimento al sottoalbero sinistro e destro.



Algoritmi di base

La **cardinalità** di un albero è il numero dei suoi nodi.

```
2Tree-Card(T)   (albero binario posizionale)
if T == nil then
     return 0
else
     l ← 2Tree-Card(T.left)
     r ← 2Tree-Card(T.right)
     return l+r+1
end if
```

```
kTree-Card(T)   (albero rappresentato con child e sibling)
if T == nil then
     return 0
else
```

```
      card ← 1
      C ← T.child
      while C ≠ nil do
            card ← card + kTree-Card(C)
            C ← C.sibling
      end while
      return card
end if

2Tree-Height(T)   (albero binario posizionale)
      > Pre: T non è vuoto
if T.left = nil and T.right = nil then
      return 0      > T ha un solo nodo
else
      hl, hr ← 0
      if T.left ≠ nil then
            hl ← 2Tree-Height(T.left)
      end if
      if T.right ≠ nil then
            hr ← 2Tree-Height(T.right)
      end if
      return 1+max{hl, hr}
end if

kTree-Height(T)   (albero rappresentato con child e sibling)
      > Pre: T non è vuoto
if T.child = nil then
      return 0     > T ha un solo nodo
else
      h ← 0
      C ← T.child
      while C ≠ nil do
            h ← max{h, kTree-Height(C)}
            C ← C.sibling
      end while
      return h+1
end if
```

Visite

La visita (completa) di un albero consiste in un'ispezione dei nodi dell'albero in cui ciascun nodo sia visitato (ispezionato) esattamente una volta. Due tipi di visite:

- **Depth First Search, DFS**
- **Breadth First Search, BFS**

La visita in **profondità** scende lungo un ramo fino ad una foglia, poi, tornando su, ricomincia a scendere appena ci sono dei nodi ancora non visitati.
Per la visita in profondità si può utilizzare una **pila** per memorizzare i nodi visitati.

```
Tree-DFS-Stack(T)
      > Pre: T non è vuoto
S ← pila vuota
Push(S, T)
while !Empty(S) do
      T' ← Pop(S)
      visita T'
      for all C figlio di T' do
            Push(S, C)
      end for
end while
```

La visita in profondità può essere effettuata anche in maniera ricorsiva, senza l'utilizzo di una struttura aggiuntiva:

```
Tree-DFS(T)   (albero rappresentato con child e sibling)
      > Pre: T non è vuoto
visita T.key
C ← T.child
while C ≠ nil do
      Tree-DFS(C)
      C ← C.sibling
end while
```

La visita in **ampiezza** deve visitare l'albero livello per livello. Questo si può ottenere con una **coda**.

```
Tree-BFS-Queue(T)
      > Pre: T non è vuoto
Q ← coda vuota
Enqueue(Q, T)
while !Empty(Q) do
      T' ← Dequeue(Q)
      visita T'
      for all C figlio di T' do
            Enqueue(Q, C)
      end for
end while
```

DFS e BFS hanno complessità $O(n)$ perché:

- per trovare un limite superiore possiamo contare quante operazioni Push/Pop (oppure Enqueue/Dequeue) avvengono in una DFS (o BFS);
- per ogni nodo si fa un inserimento ed una estrazione nella struttura dati d'appoggio (pila o coda);
- quindi DFS e BFS hanno costo $O(2n) = O(n)$ dove n è la cardinalità dell'albero.

Alberi binari di ricerca

Data un'etichetta, a, e due alberi, l e r in BRT(A), agganciando *l* come sottoalbero sinistro e *r* come sottoalbero destro al nodo con etichetta *a*, si ottiene un nuovo albero binario di ricerca se tutte le etichette in l sono minori di a e tutte le etichette in r sono maggiori di *a*.

Stampa di tutte le etichette in ordine. Idea: dato l'albero {a, l, r} prima bisogna stampare tutte le etichette in l in ordine, poi stampare a e, infine, stampare tutte le etichette in r in ordine:

```
Print-Inorder(T)
      ▷ pre: T binario di ricerca
      ▷ post: stampate le chiavi in T in ordine
if T = nil then
return
end if
Print-Inorder(T.left)
print T.key
Print-Inorder(T.right)
```

Copia di tutte le etichette in una lista semplice (non circolare e senza riferimento all'ultimo elemento) in ordine. Assumiamo di avere a disposizione due algoritmi che fanno operazioni con liste:

- ListInsert(key c, list L) restituisce una lista in cui si ha un nodo in testa con etichetta c e L agganciata a questo nodo (complessità O(1));
- Append(list L1, list L2) restituisce una lista in cui L2 è agganciata a L1 in coda (complessità O(|L1|) dove |L1| denota il numero di elementi in L1).

```
ToList-Inorder(T)
      ▷ pre: T binario di ricerca
      ▷ post: ritorna la lista ordinata delle chiavi in T
if T = nil then
     return nil
else
     L ← ToList-Inorder(T.left)
     R ← ToList-Inorder(T.right)
     R ← ListInsert(T.key, R)
     return Append(L, R)
end if
```

Complessità nel caso peggiore: $O(n^2)$
Possiamo semplificare questo algoritmo senza usare Append

```
ToList-Inorder(T, L)
      ▷ pre: T binario di ricerca
      ▷ post: ritorna la lista ordinata delle chiavi in T concatenata con L
if T = nil then
      return L
else
      L ← ToList-Inorder(T.right, L)
      L ← ListInsert(T.key, L)
      return ToList-Inorder(T.left, L)
end if
```

In questo modo la complessità diventa O(n)

Per la ricerca di un elemento si scende nell'albero in modo ricorsivo o iterativo

```
Ric-Search(x, T)
      > Pre: x chiave, T albero binario di ricerca
      > Post: restituito un nodo S ∈ T con S.key = x se esiste, nil altrimenti
if T == nil then
      return nil
else
      if x == T.key then
            return T
      else
            if x < T.key then
                  return Ric-Search(x, T.left)
            else
                  return Ric-Search(x, T.right)
            end if
      end if
end if
```

```
It-Search(x, T)
      > Pre: x chiave, T albero binario di ricerca
      > Post: restituito un nodo S ∈ T con S.key = x se esiste, nil altrimenti
S ← T
while S ≠ nil and x ≠ S.key do
      if x < S.key then
            S ← S.left
      else
```

```
            S ← S.right
      end if
end while
return S
```

La complessità è $O(h)$ dove h è l'altezza dell'albero.

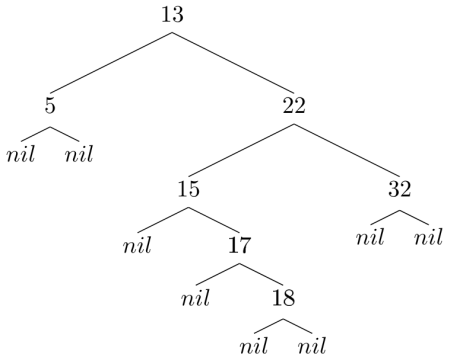
Per cercare il **minimo** si scende verso sinistra.

```
Tree-Min(T)
      > Pre: T binario di ricerca non vuoto
      > Post: il nodo S ∈ T con S.key minimo
S ← T
while S.left ≠ nil do
      S ← S.left
end while
return S
```

La complessità è $O(h)$ dove h è l'altezza dell'albero.

Ricerca del successore: il successore di un nodo N in un albero binario di ricerca T è il nodo con etichetta minima tra quelle maggiore di N.key.

- Se il nodo N contiene la chiave massima allora non ha successore nell'albero.
- Se il nodo N ha un discendente destro allora il successore di N è il massimo del sottoalbero che ha radice in N.right.
- Se N non contiene l'etichetta massima dell'albero e non ha un discendente destro allora il successore di N si trova risalendo a partire dal nodo N e fermandosi al primo nodo che ha un'etichetta maggiore di quella di N



Il nodo con etichetta 32 non ha successore nell'albero. Il successore del nodo 13 è il nodo 15. Il successore del nodo 18 è il nodo 22.

```
Tree-Succ(N)
      ▷ pre: N nodo di un albero bin. di ricerca
      ▷ post: il successore di N se esiste, nil altrimenti
if N.right ≠ nil then
      return Tree-Min(N.right)
else  ▷ il successore è l'avo più vicino con etichetta maggiore
```

```
      P ← N.parent
      while P ≠ nil and N = P.right do
            N ← P
            P ← N.parent
      end while
      return P
end if
```

Inserimento: L'inserimento sostituisce un sottoalbero vuoto (un `nil`) con un sottoalbero che contiene un singolo nodo con l'etichetta da inserire. La posizione si cerca secondo le caratteristiche dell'albero binario di ricerca.
Il ciclo `while` trova il posto del nuovo nodo e il resto fa l'inserimento

```
Tree-Insert(N, T)
      > Pre: N nuovo nodo con N.left == N.right == nil, T è
            un albero binario di ricerca
      > Post: N è un nodo di T, T è un albero binario di ricerca
P ← nil
S ← T
while S ≠ nil do    > inv: se P ≠ nil allora P è il padre di S
      P ← S
      if N.key == S.key then
            return
      else
            if N.key < S.key
                  S ← S.left
            else
                  S ← S.right
            end if
      end if
end while
N.parent ← P
if P == nil then
      T ← N
else
      if N.key < P.key then
            P.left ← N
      else
            P.right ← N
      end if
end if
```

Cancellazione: si devono distinguere 3 casi quando si deve cancellare un nodo `Z`:

1. Z è una foglia → settare a nil il riferimento a Z nel suo genitore;
2. Z ha solo 1 figlio → agganciare l'unico figlio di Z al parent di Z;
3. Z ha 2 figli → l'etichetta in Z si sostituisce con l'etichetta minima in `Z.right`, poi eliminare il nodo dove si trovava originariamente l'etichetta.

```
1-Delete(Z, T)
      ▷ pre: Z nodo di T con esattamente un discendente
      ▷ post: Z non è più un nodo di T
if Z = T then
      if Z.left ≠ nil then
            T ← Z.left
      else
            T ← Z.right
      end if
      Z.parent ← nil
else
      if Z.left ≠ nil then
            Z.left.parent ← Z.parent
            S ← Z.left
      else
            Z.right.parent ← Z.parent
            S ← Z.right
      end if
      if Z.parent.right = Z then
            Z.parent.right ← S
      else
            Z.parent.left ← S
      end if
end if
```

```
Tree-Delete(Z, T)
      ▷ pre: Z nodo di T
      ▷ post: Z non è più un nodo di T
if Z.left = nil and Z.right = nil then      ▷ Z è una foglia
      if Z = T then
            T ← nil
      else
            if Z.parent.left = Z then   ▷ Z è figlio sinistro
                  Z.parent.left ← nil
            else  ▷ Z è figlio destro
                  Z.parent.right ← nil
            end if
```

```
      end if
else
      if Z.left = nil ∨ Z.right = nil then
            1-Delete(Z, T)
      else        ▷ Z ha due figli e dunque si può cercare il minimo in Z right
      Y ← Tree-Min(Z.right)
      Z.key ← Y.key
      Tree-Delete(Y, T)
      end if
end if
```

Heap

Heap **massimo**: un albero binario etichettato con numeri interi con le seguenti proprietà:

- l'albero è completo salvo al più l'ultimo livello (riempito da sinistra);
- per ogni nodo (tranne la radice), l'etichetta del suo genitore è maggiore dell'etichetta del nodo (proprietà invertita per un heap **minimo**).

Posizionamento. Sia `H` un vettore che rappresenta uno heap e l'indice della radice sia 0. Quindi `H[0]` è l'etichetta della radice. Inoltre, i figli di `H[i]` sono `H[2i]` (figlio sinistro) e `H[2i + 1]` (figlio destro). Il genitore di `H[i]` è `H[i/2]`.

```
Parent(H, i)
      > Pre: 1 ⩽ i ⩽ H.N
      > Post: restituisce la posizione del genitore se esiste, 0 altrimenti
return i/2;

Left(H, i)
      >Pre: 1 ⩽ i ⩽ H.N
      >Post: restituisce la posizione del figlio sinistro se esiste, i altrimenti
if 2i ⩽ H.N then
      return 2i
else
      return i

Right(H, i)
      > Pre: 1 ⩽ i ⩽ H.N
      > Post: restituisce la posizione del figlio destro se esiste, i altrimenti
if 2i+1 ⩽ H.N then
      return 2i+1
else
      return i
```

Inserimento. L'etichetta da inserire viene inserita inizialmente in fondo dell'array e poi viene fatta risalire tramite scambi verso la radice fino a trovare una posizione corretta per quanto riguarda le caratteristiche dello heap.

```
HeapInsert(H, x)
      > Pre: H è un heap
      > Post: H è un heap con x inserito
H.N ← H.N + 1
p ← H.N
H[p] ← x
while p > 1 and H[p] > H[Parent(H, p)] do
      scambia H[p] e H[Parent(H, p)]
      p ← Parent(H, p)
end while
```

L'algoritmo effettua al massimo $l - 1$ scambi dove l è il numero di livelli dell'albero. La complessità quindi è $O(\log n)$ dove n è il numero di nodi dell'albero.

Estrazione del massimo. La cancellazione del massimo (che si trova per forza nella radice) si effettua in due fasi:

1. l'elemento più a destra dell'ultimo livello rimpiazza la radice;
2. l'elemento ora in radice viene fatto discendere lungo l'albero finché non sia maggiore di entrambi i figli; nel discendere si sceglie sempre il figlio con l'etichetta maggiore.

```
HeapExtract(H)
      > Pre: H è un heap
      > Post: H è un heap con etichetta massimo eliminata
H[0] ← H[H.N]
H.N ← H.N-1
Heapify(H, 0)

Heapify(H, i)
      > Pre: 1 ⩽ i ⩽ H.N, i sottoalberi con radice Left(H, i) e Right(H.i) sono heap
      > Post: l'albero con radice in i è heap
m ← index of Max{H[i], H[Left(H, i)], H[Right(H, i)]}
if m ≠ i then
      scambia H[m] e H[i]
      Heapify(H, m)
end if
```

L'algoritmo effettua al massimo $l - 1$ scambi dove l è il numero di livelli dell'albero. La complessità quindi è $O(\log n)$ dove n è il numero di nodi dell'albero.

Heapsort

Consideriamo il seguente vettore `V = V[0…N], V[N+1…M]`.
Se `V[0…N]` rappresentasse uno heap allora sfruttando l'estrazione del massimo si può allargare la parte ordinata del vettore:
Gli elementi che sono in posizione foglia sono già heap (un singolo nodo è heap).
Una posizione `i` corrisponde ad una foglia se `2i > M` (il nodo non deve avere un figlio sinistro per essere una foglia). Di conseguenza, le foglie sono nelle posizioni `[(M/2)+1…M]`.
Bisogna iterare dunque Heapify a partire dalla posizione `M/2` fino alla posizione `0`.
Denoteremo con `V.M` il numero di elementi in `V` e con `V.N` il numero di elementi su cui devono operare gli algoritmi introdotti nella sezione precedente.

```
BuildHeap(V)
      > Pre: V è un vettore di M elementi
      > Post: V rappresenta uno heap
V.N ← V.M
for i ← V.M/2 downto 0 do
      Heapify(V, i)
end for
```

```
HeapSort(V)
      > Pre: V è un vettore di M elementi
      > Post: V è ordinato
BuildHeap(V)
for i ← V.N downto 1 do
      scambia V[i] e V[0]
      V.N ← V.N-1
      Heapify(V, 0)
end for
```

Coda di priorità con Heap

Una coda di priorità si può realizzare con uno heap massimo. Le operazioni `EnqueuePr` e `DequeuePr` si possono implementare utilizzando direttamente i meccanismi dell'inserimento e dell'estrazione dello heap massimo. L'operazione `FrontPr` richiede soltanto restituire il primo elemento del vettore dello heap che rappresenta la coda di priorità.

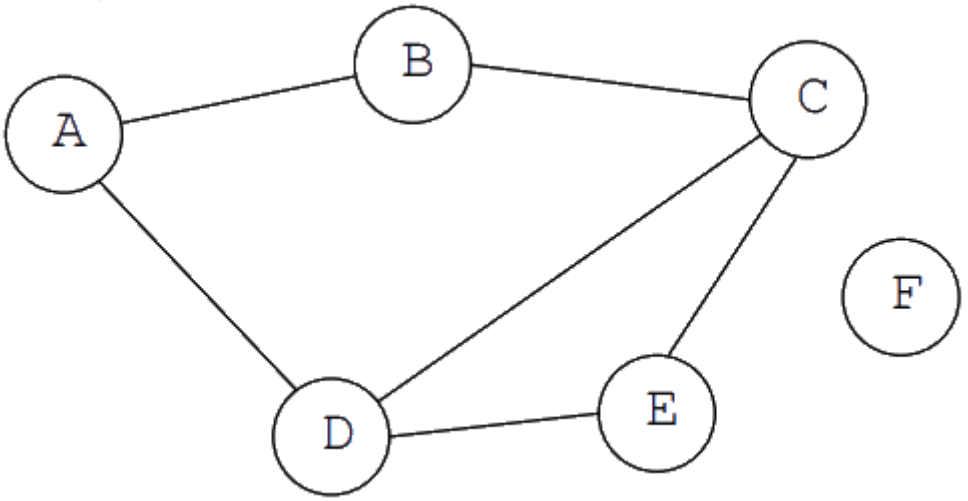
Operazioni di una coda non prioritaria:

```
Size(Q)
if Q.tail ⩾ Q.head then
      return Q.tail - Q.head
```

```
return Q.M - (Q.head - Q.tail)

Empty(Q)
if Q.tail == Q.head then
     return true
return false

NextCell(Q, c)
if c ≠ Q.M then
     return c+1
return 1

Enqueue(Q, t)
if Size(Q) ≠ Q.M-1 then
     Q[Q.tail] ← t
     Q.tail ← NextCell(Q, Q.tail)
else
     error overflow

Front(Q)
if Size(Q) == 0 then
     error underflow
else
     return Q[Q.head]

Dequeue(Q)
if Size(Q) == 0 then
     error underflow
else
     t ← Q[Q.head]
     Q.head ← NextCell(Q, Q.head)
     return t
```

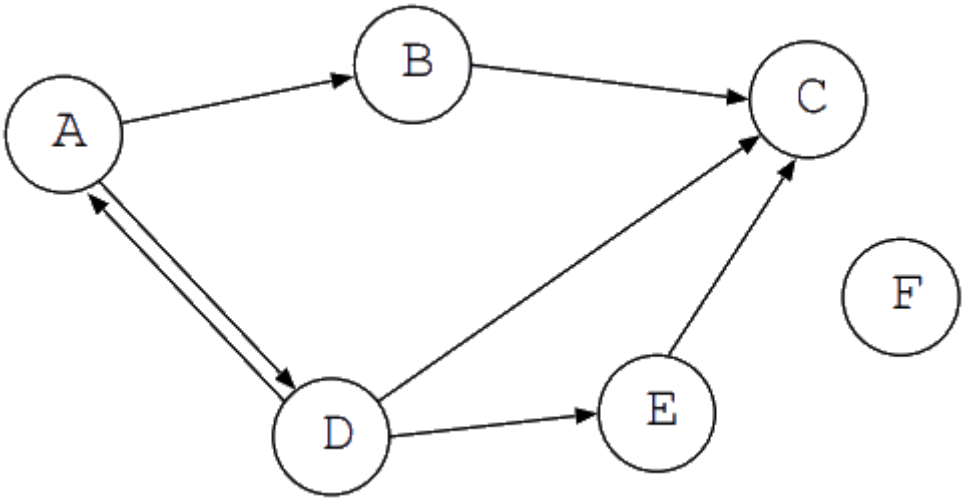
Grafi

Un **grafo** `G = (V, E)` consiste in un insieme V di *vertici* (o **nodi**) e un insieme E di *coppie di vertici* (o **archi**), dove ogni arco connette due vertici.
Relazione **simmetrica =** coppie non orientate → grafo **non orientato**



`V = {A, B, C, D, E, F}`
`E = {(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)}`
Ogni arco ha un corrispettivo (es: se c'è (A, D), c'è anche (D, A)).

Relazione **non simmetrica** = coppie ordinate → grafo **orientato**



`V = {A, B, C, D, E, F}`
`E = {(A, B), (A, D), (B, C), (D, C), (E, C), (D, E), (D, A)}`
Archi simmetrici sono diversi tra loro (es: (A, D) e (D, A) sono diversi).

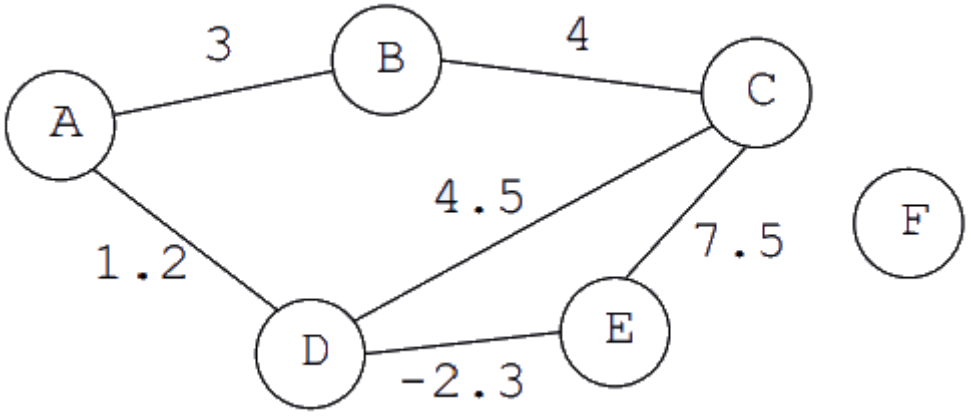
Un vertice x si dice *adiacente* a y solo se esiste (y, x) in E (es: B, D ed E sono adiacenti a C). In un grafo non orientato, l'adiacenza è simmetrica.

Il **grado** di un grafo è diverso in base al tipo del grafo:

- non orientato: il grado di un vertice è il numero di archi che da esso si dipartono;
- orientato: il grado entrante/uscente di un vertice è il numero di archi incidenti in/da esso; il grado di un vertice è la somma del suo grado entrante e del suo grado uscente.

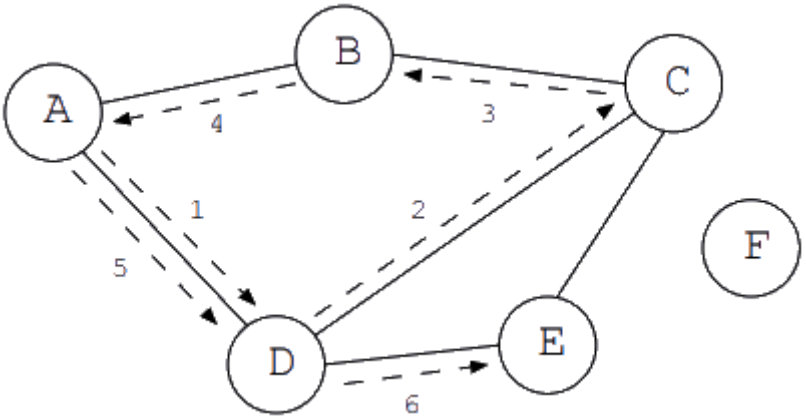
Ad ogni arco è associato un **peso**. Un **grafo pesato** (G, W) contiene una *funzione peso*

$W : E \rightarrow \mathbb{R}$

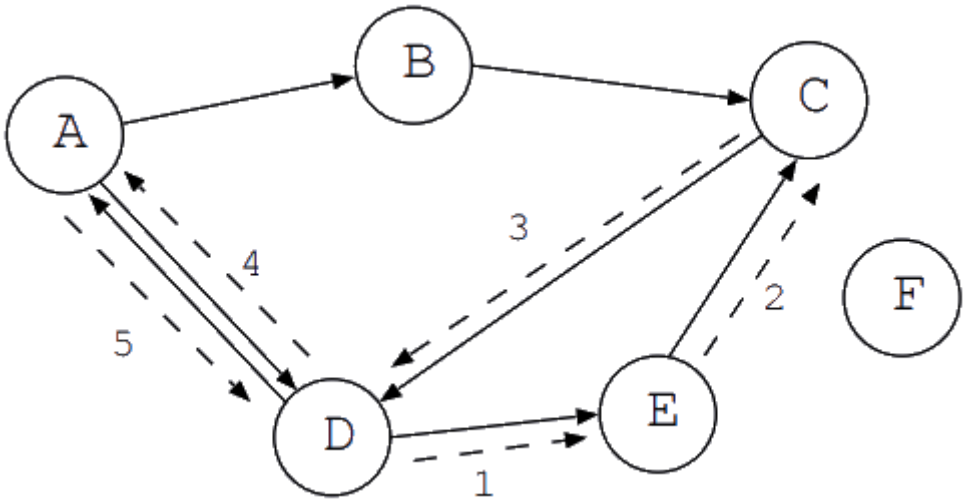


`W((A, B)) = 3`
`W((D, E)) = -2.3`
`W((C, F)) = ∞`

Il **cammino** in un grafo è la sequenza di passaggi che vengono fatti per passare da un vertice ad un altro in un grafo. La *lunghezza* del cammino è il numero totale di passaggi effettuati.



Grafo non orientato:
A, D, C, B, A, D, E è un cammino nel G di lunghezza 6.



Grafo orientato:
D, E, C, D, A, D è un cammino nel G;
D, E, C, B, A, D non è un cammino nel G.
Un cammino è *semplice* se tutti i suoi vertici compaiono 1 sola volta nella sequenza (sono distinti, fanno eccezione il primo e l'ultimo)

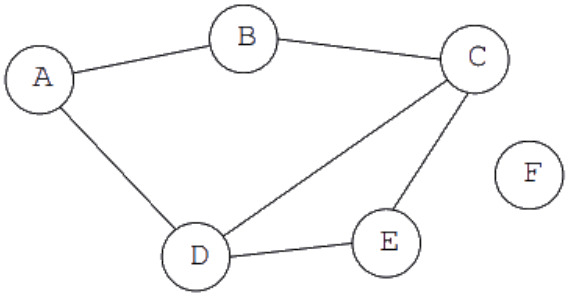
Un grafo non orientato è *connesso* se esiste un cammino da ogni vertice a ogni altro vertice.

Un grafo orientato è *fortemente connesso* se esiste un cammino da ogni vertice a ogni altro vertice.
Un grafo orientato è debolmente connesso se il grafo ottenuto da G dimenticando la direzione degli archi è connesso.

Un *ciclo* in un grafo orientato è un cammino $x_1 \ldots x_n$ con $n > 2$ e $x_1 = x_n$.
Un ciclo in un grafo non orientato è un cammino $x_1 \ldots x_n$ con $n > 2$ e $x_1 = x_n$ che non attraversa lo stesso arco due volte di seguito.
Un grafo senza cicli è detto *aciclico*.

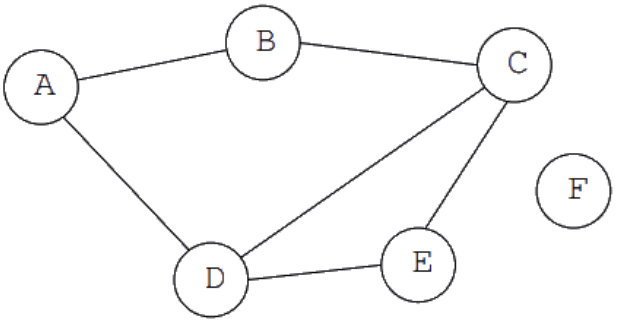
Per rappresentare le adiacenze, si possono usare le **matrici di adiacenza** o le **liste di adiacenza**.

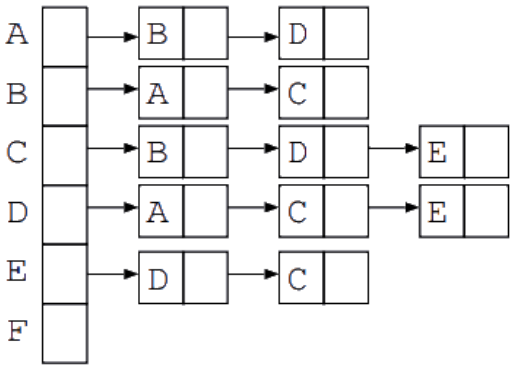
Per un grafo non orientato:



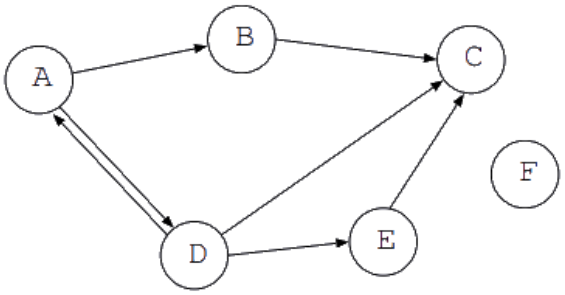
$$M(x,y) = \begin{cases} 1 & \text{se } (x,y) \in E \\ 0 & \text{altrimenti} \end{cases}$$

A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0



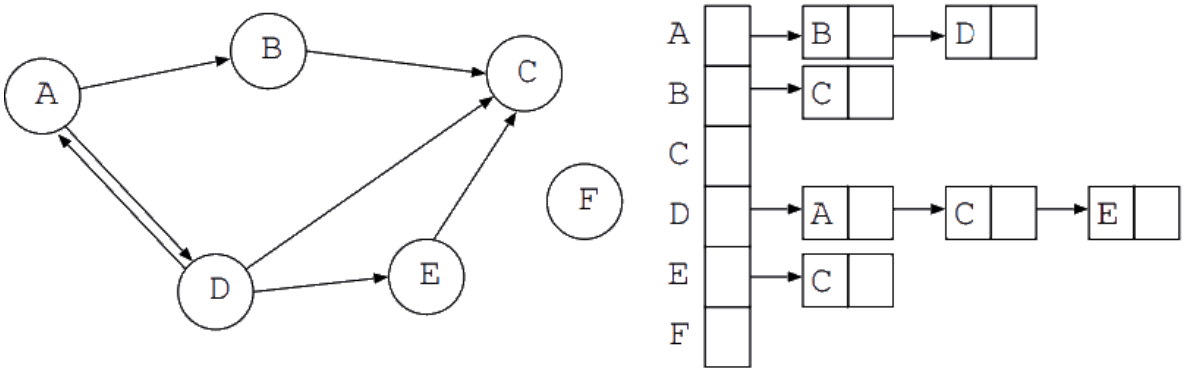


Per un grafo orientato:



$$M(x,y) = \begin{cases} 1 & \text{se } (x,y) \in E \\ 0 & \text{altrimenti} \end{cases}$$

A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	0	0	0
D	1	0	1	0	1	0
E	0	0	1	0	0	0
F	0	0	0	0	0	0



Per un grafo pesato:

- Nella matrice si sostituiscono gli 1 con i pesi degli archi;
- Nella lista si aggiunge un campo per il peso dell'arco.

Visita

Dividiamo gli insieme di vertici in tre sotto insiemi:

- bianco: nodi non ancora scoperti (cioè non visitati)
- grigio: vertici scoperti di cui adiacenti non sono ancora tutti scoperti
- nero: nodi scoperti di cui adiacenti sono già stati scoperti

Proprietà:

- colore di un nodo può solo passare da bianco a grigio a nero

Invarianti:

- se (u, v) ∈ E e u è nero, allora v è grigio o nero
- tutti i vertici grigi o neri sono raggiungibili da s

- qualunque cammino da s ad un nodo bianco deve contenere almeno un vertice grigio

Complessità della visita:

- inizializzazione richiede tempo $O(|V|)$
- ogni vertice viene inserito (eliminato) una volta in (da) D
- assumiamo che le operazioni di inserimento e eliminazione richiedano un tempo costante
- tempo totale dedicato alle operazioni su D è $O(|V|)$
- le liste di adiacenza vengono percorse una volta
- tempo totale dedicato alla ricerca di nodi bianchi è $O(|E|)$
- tempo totale: $O(|V| + |E|)$

Visita in ampiezza

```
BFS(G, s)
Q Empty-Queue
s.color ← grigio
Enqueue(Q, s)
while Non-Empty(Q) do
      u ← First(Q)
      if ∃v bianco ∈ adj[u] then
            v.color ← grigio
            v.π ← u
            Enqueue(Q, v)
      else
            u.color ← black
            DeQueue(Q)
      end if
end while
```

```
DFS-Rec(G, s)
s.color ← grigio
for all v bianco ∈ adj[s] do
      v.π ← s
      DFS-Rec(G, v)
end for
s.color ← nero
```

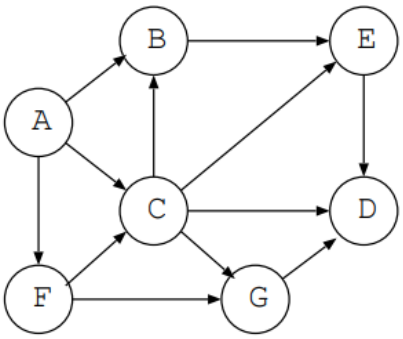
Visita in larghezza

```
DFS(G, s)
S ← Empty-Stack
s.color ← grigio
```

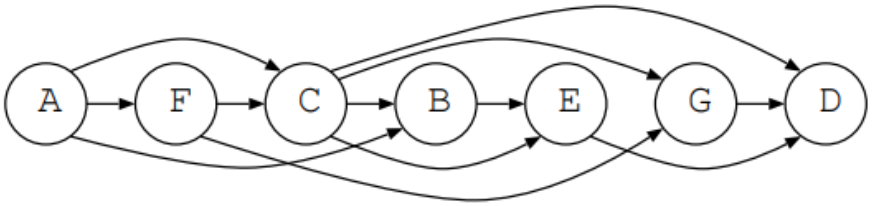
```
Push(S, s)
while Non-Empty(S) do
      u ← Top(S)
      if ∃v : v bianco ∈ adj[u] then
            v.color ← grigio
            v.π ← u
            Push(S, v)
      else
            u.color ← black
            Pop(S)
      end if
end while
```

Ordinamento topologico

L'ordinamento topologico è una funzione $\sigma : V \rightarrow \{1,\ldots,|V|\}$ tale che $\sigma(u) < \sigma(v)$ se esiste un arco (u, v) in G.
Esempio:



$$\sigma(A) = 1, \sigma(F) = 2,$$
$$\sigma(C) = 3, \sigma(B) = 4,$$
$$\sigma(E) = 5, \sigma(G) = 6,$$
$$\sigma(D) = 7$$



Il seguente algoritmo determina un ordine topologico sulla base di una visita in profondità.
In pratica l'algoritmo crea una lista L che contiene i nodi in ordine decrescente di tempi di ne visita (l'attributo f).

```
Topological-Sort(G)
L ← lista vuota di vertici
Inizializza(G)
for ∀u ∈ V do
      if u.color = bianco then
            DFS-Topological(G, u, L)
```

```
     end if
end if
return L

Inizializza(G)
for ∀u ∈ V do
     u.color ← bianco
     u.π ← nil
     u.i ← ∞
     u.f ← ∞
end for
time ← 1

DFS-Topological(G, s, L)
s.color ← grigio
s.d ← time
time ← time+1
while ∃v ∈ V tale che v.color = bianco ∧ (s, v) ∈ E do
     v ← un nodo tale che v.color = bianco ∧ (s, v) ∈ E
     v.π = s
     DFS-Topological(G, v, L)
end while
s.color ← nero
s.f ← time
time ← time+1
in testa di L inserisci s
```

In un grafo orientato e aciclico ogni arco (u, v) ricade in una delle tre seguenti categorie:

- arco della foresta generata dalla visita;
- arco in avanti;
- arco di attraversamento.

In tutti i tre casi, il tempo di fine visita di u è maggiore di quello di v ($u.f > u.v$). Quindi per ogni arco (u, v) in L il nodo u precede il nodo v. Quindi in L si ha un ordine topologico.

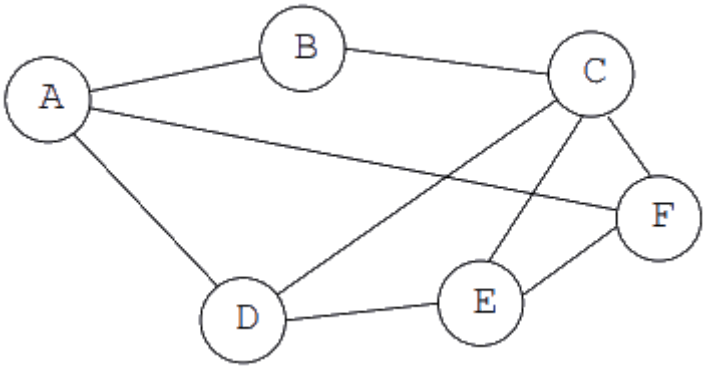
Alberi di copertura minima

Un albero di copertura minima è quell'albero in cui dato un grafo con archi pesati, la somma dei pesi degli archi restituisce un valore minimo.

```
Generic-MST(G, w)
     > pre: G = (V, E) non orientato e connesso
     > post: A ⊆ E è un MST di G
A ← ∅
```

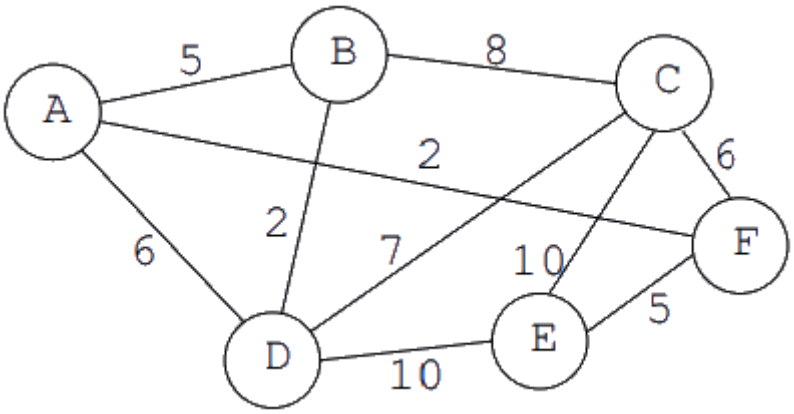
```
while V [A] ≠ V do   > inv. ∃T MST di G. A ⊆ T
      (u, v) ← un arco sicuro per A in E \ A
      A ← A ∪ {(u, v)}
end while
return A
```

Il taglio di un grafo G(V, E) è una partizione di V in due insiemi X e V - X.



`X = {A, C, E}, V-X = {B, D, F}`

Un arco `(u, v)` *attraversa* il taglio `(X, V-X)` se `u ∈ X, v ∈ V-X`
Esempio: l'arco `(B, C)` è un arco che attraversa il taglio `(X = {A, C, E}, V-X)`.



Un taglio *rispetta* un insieme di archi A se nessun arco di A attraversa il taglio.
Esempio: il taglio `(X = {A, C, E}, V-X)` rispetta l'insieme di archi `{(C, E), (B, D)}` ma non rispetta `{(A, F), (B, C)}`.

Un arco che attraversa un taglio è un *arco leggero* se è un arco di peso minimo tra quelli che attraversano il taglio.
Esempio: l'arco `(A, F)` è leggero per il taglio `(X = {A, C, E}, V-X)`.

Esiste un teorema per riconoscere **archi "sicuri"**:

1. Sia `G = (V, E)` un grafo non orientato, connesso e pesato;
2. Sia `A` un sottoinsieme di `E` contenuto in qualche albero di copertura minimo;
3. Sia `(X, V-X)` un qualunque taglio che rispetta `A`;

4. Sia `(u, v)` un arco leggero che attraversa `(X, V-X)`;
5. Allora l'arco `(u, v)` è sicuro per `A`;

Corollario del teorema:

1. Sia `G = (V, E)` un grafo non orientato, connesso e pesato;
2. Sia `A` un sottoinsieme di `E` contenuto in qualche albero di copertura minimo;
3. Sia `C` una componente connessa (un albero) nella foresta `G(A) = (V, A)`;
4. Sia `(u, v)` è un arco leggero che connette `C` a una qualche altra componente connessa di `G(A)`;
5. Allora l'arco `(u, v)` è sicuro per `A`.

Algoritmo di Kruskal

Il seguente algoritmo ottimo è utilizzato per calcolare i MST in un grafo non orientato e con archi senza costi negativi.

- mantiene un sottografo non necessariamente connesso
- (V, A) di un MST (minimal spanning tree, minimo albero
- ricoprente)
- all'inizio: tutti i vertici del grafo e nessun arco
- per ogni arco, in ordine non decrescente di costo:
    - se l'arco ha entrambi gli estremi nella stessa componente connessa di (V, A) escluderlo dalla soluzione
    - altrimenti, basandosi sul corollario I, includilo nella soluzione (fondendo due componenti connesse)

```
MST-Kruskal(G, w)
      > pre: G = (V, E) non orientato e connesso
      > post: A ⊆ E è un MST di G
A ← 0
P ← {{v1}, ... , {vn}} una partizione di V
L ← la lista degli archi in E in ordine non decrescente rispetto a
w
while L ≠ nil do
      (u, w) ← Head(L), L Tail(L)
      U, W ← gli unici insiemi in P t.c. u ∈ U e w ∈ W
      if U ≠ W then
            A ← A ∪ {(u, w)}
            P ← (P \ {U, W}) ∪ {U ∪ W}
      end if
end while
return A
```

La complessità non può essere valutata se non conosciamo la struttura dati usata per rispondere alla domanda di esistenza del ciclo.

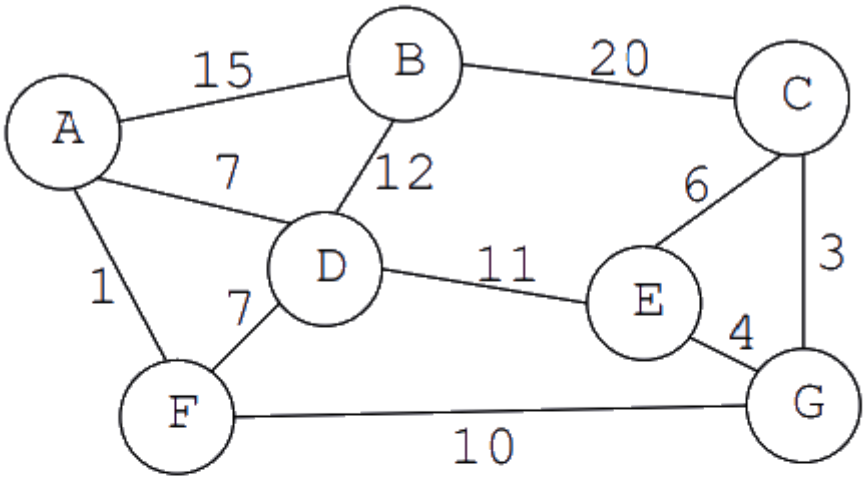
Algoritmo di Prim

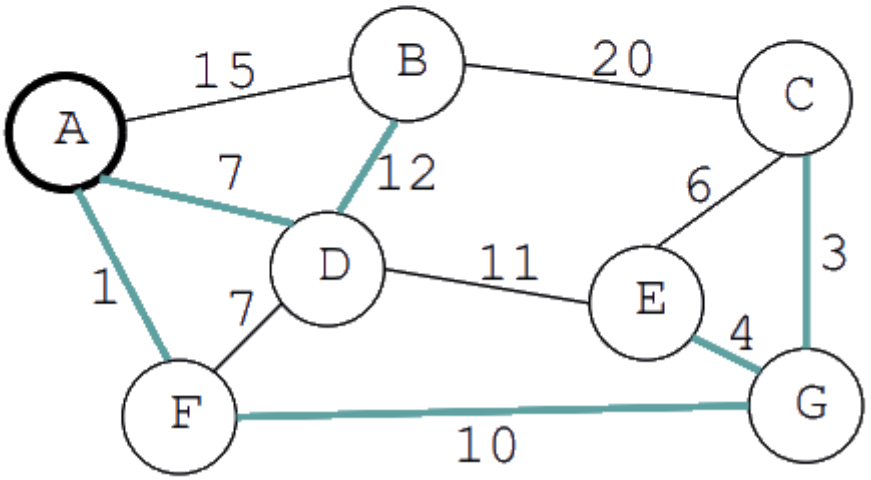
Mantiene un sottografo connesso `(V-Q, A)` di un MST. All'inizio consiste di un nodo arbitrario: `Q` contiene tutti i nodi tranne quello iniziale e `A` è vuoto. Applica `n-1` volte il seguente passo: scegli un arco `(u, v)` di peso minimo incidente su `(V-Q, A)`, aggiungilo ad `A` e togli da `Q` il vertice a cui porta l'arco.

```
MST-Prim(G, s)
A ← ∅
Q ← V - {s}
while Q ≠ ∅ do
      scegli un arco (u, v) con peso minimo e u ∈ V-Q, v ∈ Q
      A ← A ∪ (u, v)
      Q ← Q - v
end while
```

Algoritmo con coda di priorità:

```
MST-Prim(G, w, r)
      > Pre: G = (V, E), n = |V |, m = |E|
for all v ∈ V do
      v.key ← ∞, v.π ← nil
end for
A ← ∅, r.key ← 0
Q ← MakePriorityQueue(V)
while Q ≠ ∅ do
      u ← Extract-Min(Q)
      A ← A ∪ {(u.π, u)}
      for all v  Adj[u] and v ∈ Q do
            if v.key > w(u, v) then
                  v.key ← w(u, v)
                  v.π ← u
                  Decrease-Key(v, w(u, v), Q)
            end if
      end for
end while
return A
```





Complessità dell'algoritmo:
Q = coda di priorità con heap binario;
Costo dell'algoritmo limitato da:

- costo dell'inizializzazione $O(|V|)$;
- costo dell'estrazione del minimo: $O(|V|log|V|)$;
- costo di risistemazione dello heap dopo il decremento eventuale delle chiavi: $O(|E|log|V|)$;

La complessità finale è quindi di $O((|V| + |E|)log|V|) = O(|E|log|V|)$.

Dijkstra

L'algoritmo di dijkstra serve per trovare i cammini minimi in un grafo ben definito. Richiede che w (l'insieme dei pesi nel grafo) non sia mai negativo.

```
Dijkstra(G, w, s)      > G = (V, E) orientato con pesi non negativi w
for all v ∈ V do
      v.d ← ∞          > stima in eccesso di  (s, v)
      v.π ← nil
```

```
end for
s.d ← 0
Q ← MakePriorityQueue(V)  > Q è uno heap minimo con v.key = v.d
S ← ∅
while Q ≠ ∅ do            > inv. Q = V \ S e ∀v ∈ S.v.d =  (s, v)
     u ← Extract-Min(Q)
     S ← S ∪ {u}
     for all v ∈ Adj[u] do
          if v.d > u.d + w(u, v) then
               v.d ← u.d + w(u, v)
               v.π ← u
               Decrease-Key(v, w(u, v), Q)
          end if
     end for
end while
return S
```

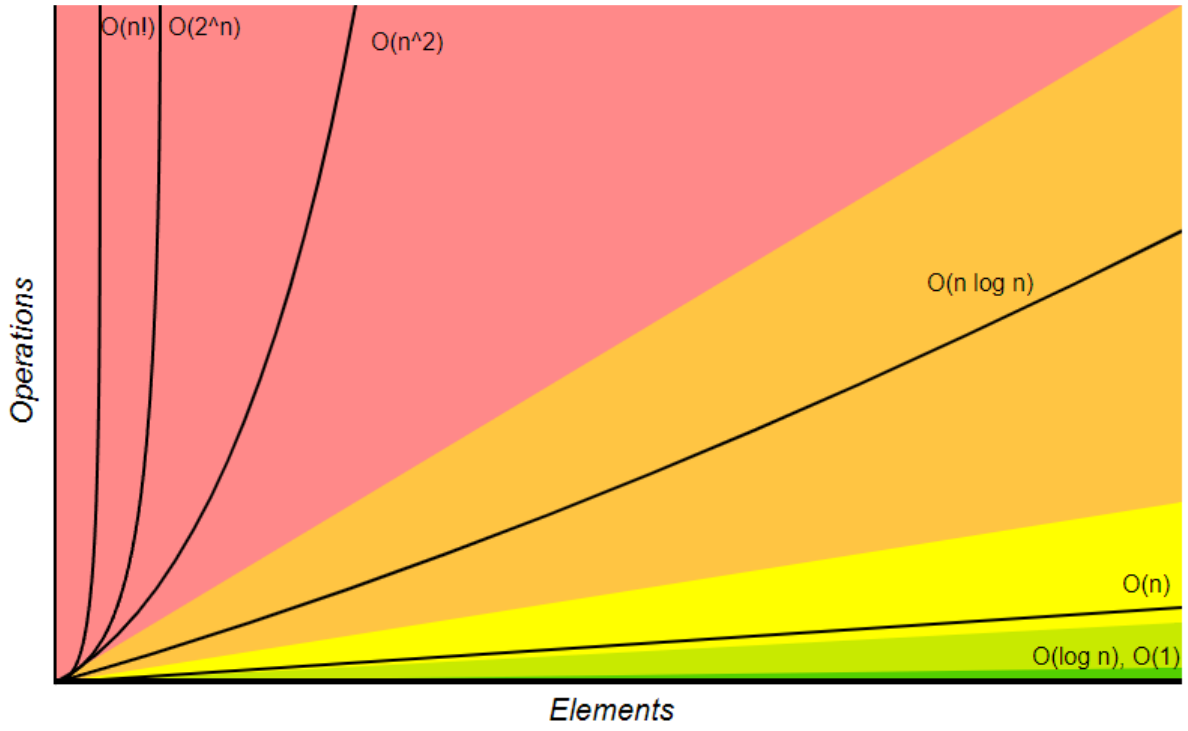
Tabella degli O-grande

Relazione di ricorrenza	O-grande	Esempi
$T(n) = T(n-1) + 1$ $= T((n-1)-1) + 1 + 1$ $= T(n-2) + 2$ $= T(n-k) + k\ (dopo\ k \leq n\ iterazioni)$ $= T(0) + n\ (dopo\ n\ iterazioni)$	$O(n)$	Ricerca del massimo/minimo
$T(n) = T(n-1) + n$ $= T((n-1)-1) + n - 1 + n$ $= T(n-2) + n - 1 + n$ $= T((n-2)-1) + n - 2 + n - 1 + n$ $= T(n-3) + n - 2 + n - 1 + n$ $= T(n-k) + \sum_{i=0}^{k-1} (n-i)\ (dopo\ k \leq n\ iterazioni)$ $= T(0) + \sum_{i=0}^{n-1} (n-i)\ (dopo\ n\ iterazioni)$ $= 1 + \sum_{i=1}^{n} i$ $= 1 + \frac{n(n+1)}{2}$	$O(n^2)$	Insertion Sort
$T(n) = 2T(n-1) + 1$ $= 2[2T((n-1)-1) + 1] + 1$ $= 4T(n-2) + 2 + 1$	$O(2^n)$	Hanoi

$= 4[2T((n-2)-1)+1]+2+1$ $= 8T(n-3)+4+2+1$ $= 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i \ (dopo\ k \leq n\ iterazioni)$ $= 2^n T(0) + \sum_{i=0}^{n-1} 2^i \ (dopo\ n\ iterazioni)$ $= 2^n + 2^n - 1$		
$T(n) = T(\frac{1}{2}) + 1$ $= [T(\frac{\frac{n}{2}}{2}) + 1] + 1$ $= T(\frac{n}{4}) + 2$ $= T(\frac{n}{8}) + 3$ $= T(\frac{n}{2^k}) + k \ (k \leq log_2 n)$ $= T(1) + log_2 n$	$O(log_2 n)$	Ricerca dicotomica
$T(n) = T(\frac{n}{2}) + n$ $= T(\frac{n}{4}) + \frac{n}{2} + n$ $= T(\frac{n}{8}) + \frac{n}{4} + \frac{n}{2} + n$ $= T(\frac{n}{2^3}) + \frac{n}{2^2} + \frac{n}{2^1} + \frac{n}{2^0}$ $= T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} \frac{n}{2^i} \ (k \leq log_2 n)$ $= T(\frac{n}{2^{log_2 n}}) + \sum_{i=1}^{log_2 n} (\frac{1}{2})^i$ $= 1 + n(2 - 2/n)$ $= 1 + 2n - 2$	$O(n)$	
$T(n) = 2T(\frac{n}{2}) + 1$ $= 2[2T(\frac{n}{4}) + 1] + 1$ $= 4T(\frac{n}{4}) + 2 + 1$ $= 4[2T(\frac{n}{8}) + 1] + 2 + 1$ $= 8T(\frac{n}{8}) + 4 + 2 + 1$ $= 2^k T(\frac{n}{2^k}) + \sum_{i=0}^{k-1} 2^i \ (k \leq log_2 n)$ $= n + \sum_{i=0}^{log_2 n - 1} 2^i$ $= n + 2^{log_2 n} - 1$ $= 2n - 1$	$O(n)$	
$T(n) = 2T(\frac{n}{2}) + n$	$O(nlogn)$	Merge Sort

$= 2[2T(\frac{n}{4}) + \frac{n}{2}] + n$ $= 4T(\frac{n}{4}) + 2n$ $= 4[2T(\frac{n}{8}) + \frac{n}{4}] + 2n$ $= 8T(\frac{n}{8}) + 3n$ $= 2^k T(\frac{n}{2^k}) + kn \ (k \leq log_2 n)$ $= 2^{log_2 n} T(\frac{n}{2^{log_2 n}}) + (log_2 n)n$ $= n + nlog_2 n$		

Tabella della complessità degli algoritmi



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)

Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)