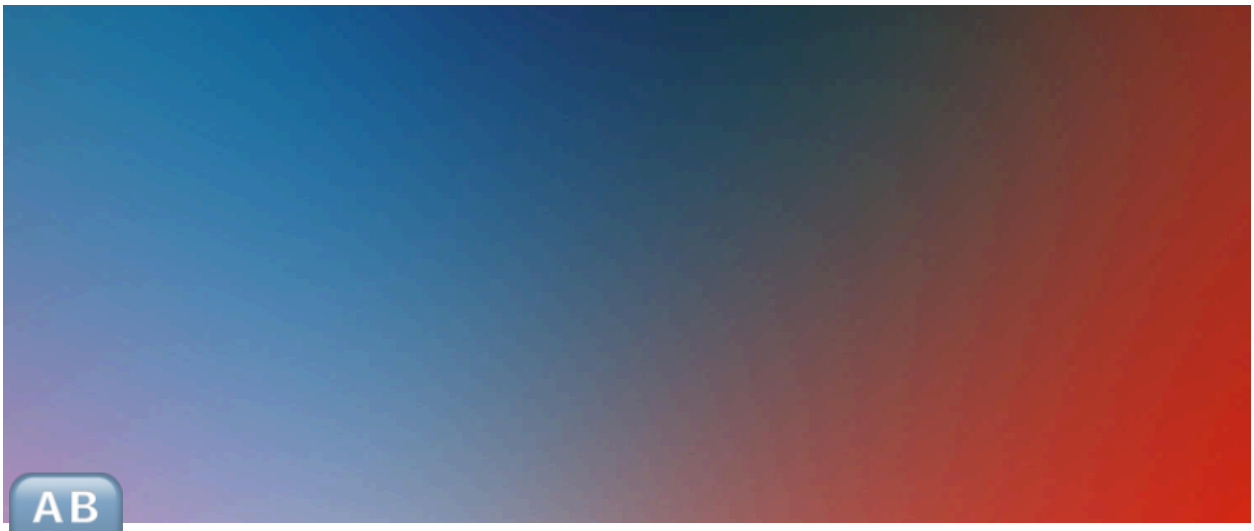

Struttura Pseudocodici

BFS con coda FIFO

```
poni discovered[s] = true e discovered[v] = false ∀ v
Q = ⌀
T = ⌀
Q.enqueue(s)
while (Q ≠ ⌀)
   Q.dequeue(u)
    foreach nodo v adiacente ad u
      if(discovered[v] = false)
         discovered[v] = true
        T = T U {(u, v)}
         Q.enqueue(v)
return T
```

Struttura Pseudocodice

- **Inizializzazione**: utilizziamo un array `discovered` per tenere traccia dei nodi già visitati, ponendo il nodo sorgente `s` a `true` e tutti gli altri nodi `v` a `false`. Inizializziamo una coda `Q` e un albero `T` come insiemi vuoti. La coda viene

usata per esplorare i nodi per livelli, mentre l'albero sarà l'albero dei percorsi. Inseriamo il nodo `s` nella coda.

- **Ciclo principale**: finché la coda non è vuota, rimuoviamo il nodo frontale `u` da `Q` e, per ogni arco `(u, v)` incidente su `u`, verifichiamo se `v` non è stato scoperto. Se non è stato scoperto, lo scopriamo, aggiungiamo l'arco a `T` e inseriamo `v` nella coda.
- **Terminazione**: restituiamo l'albero BFS `T` che è l'insieme degli archi visitati.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n + m)$, dove n è il numero dei nodi e m è il numero degli archi. L'inizializzazione richiede $O(n)$, poiché si itera su n nodi del grafo. Ogni nodo viene visitato al massimo una volta, quindi sempre $O(n)$. Per ogni nodo, l'algoritmo esamina ciascuno degli archi adiacenti, quindi ogni arco viene visitato al massimo una volta, allora $O(m)$.

BFS con liste di adiacenza

```
poni discovered[s] = true e discovered[v] = false ∀ v
poni il contatore dei livelli i=0
T = ∅
L_0 = {s}
while (i ≤ n-1)
   L_i+1 = ∅
   foreach nodo u in L_i
      foreach nodo v adiacente ad u
         if(discovered[v] = false)
            discovered[v] = true
            T = T U {(u, v)}
            L_i+1 = L_i+1 U {v}
   i=i+1
return T
```

Struttura Pseudocodice

- **Inizializzazione**: utilizziamo un array `discovered` per tenere traccia dei nodi già visitati, ponendo il nodo sorgente `s` a `true` e tutti gli altri nodi `v` a `false`. Inizializziamo il contatore dei livelli a zero, mentre inseriamo il nodo `s` nel primo livello `L_0`. Inizializziamo anche un albero vuoto.
- **Ciclo principale**: finché non si raggiungono tutti i livelli possibili, creiamo una lista vuota per il livello successivo. Per ogni nodo `u` nel livello corrente `L_i`, scopriamo tutti i suoi vicini `v`. Se `v` non è stato ancora scoperto, lo scopriamo, aggiungiamo l'arco `(u, v)` all'albero e inseriamo `v` nella lista del livello successivo. Alla fine, incrementiamo il contatore dei livello per passare al livello successivo.
- **Terminazione**: ritorniamo l'albero BFS `T` che è l'insieme degli archi visitati.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n + m)$, dove n è il numero dei nodi e m è il numero degli archi. Ogni nodo viene aggiunto ad un solo livello, quindi $O(n)$, la stessa cosa, ogni arco viene analizzato solo una volta, quindi $O(m)$.

BFS con liste di adiacenza per individuare grafo bipartito

```
poni discovered[s] = true e discovered[v] = false ∀ v
poni il contatore dei livelli i=0
L_0 = {s}
color[s] = red
while (i ≤ n-1)
   L_i+1 = ⌀
   foreach nodo u in L_i
      foreach nodo v adiacente ad u
         if(discovered[v] = false)
            discovered[v] = true
            L_i+1 = L_i+1 U {v}
            if(i%2==0)
               color[v] = blue
```

```
            else
                color[v] = red
    i=i+1
foreach arco (u, v) in G
    if(color[u] = color[v])
        return false
return true
```

Struttura Pseudocodice

- **Inizializzazione**: utilizziamo un array `discovered` per tenere traccia dei nodi già visitati, ponendo il nodo sorgente `s` a `true` e tutti gli altri nodi `v` a `false`. Inizializziamo il contatore dei livelli a zero, mentre inseriamo il nodo `s` nel primo livello `L_0`. Inizializziamo anche un albero vuoto. Per avere riferimenti per i nodi del primo livello, coloriamo `s` di rosso.
- **Ciclo principale**: finché non si raggiungono tutti i livelli possibili, creiamo una lista vuota per il livello successivo. Per ogni nodo `u` nel livello corrente `L_i`, scopriamo tutti i suoi vicini `v`. Se `v` non è stato ancora scoperto, lo scopriamo, aggiungiamo l'arco `(u, v)` all'albero e inseriamo `v` nella lista del livello successivo. Coloriamo di blu i nodi nei livelli pari e di rosso i nodi nei livelli dispari. Alla fine, incrementiamo il contatore dei livello per passare al livello successivo.
- **Verifica Bipartizione**: scorriamo ogni arco `(u, v)` in `G`, se troviamo nodi dello stesso colore, allora ritorniamo `false`, altrimenti ritorniamo `true`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n + m)$, dove n è il numero dei nodi e m è il numero degli archi. Ogni nodo viene aggiunto ad un solo livello, quindi $O(n)$, la stessa cosa, ogni arco viene analizzato solo una volta, quindi $O(m)$. L'aggiunta della colorazione non modifica la complessità.

DFS ricorsivo con albero

```
poni explored[v] = false ∀v
T = ø
DFS(G, s, explored[])
   poni explored[s] = true
   foreach nodo v adiacente ad s
      if(explored[v]=false)
         T = T U {(s, v)}
         DFS(G, v, explored[])
return T
```

Struttura Pseudocodice

- **Inizializzazione**: utilizziamo un array `explored` dove impostiamo tutti i nodi `v` del grafo a `false`. Inizializziamo un albero `T` come insieme vuoto.
- **Funzione DFS ricorsiva**: la funzione prende in input il grafo, il nodo sorgente `s` e l'array `explored`. Innanzitutto, esplora `s`. Poi, per ogni nodo `v` vicino di `s`, verifica se `v` non è stato ancora esplorato e aggiunge l'arco `(s, v)` a `T`, per poi chiamare la funzione ricorsiva DFS su `v`.
- **Terminazione**: ritorniamo l'albero DFS `T` che è l'insieme degli archi visitati.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n + m)$, dove n è il numero dei nodi e m è il numero degli archi. Ogni nodo viene esplorato esattamente una volta, quindi $O(n)$, la stessa cosa per gli archi, che vengono analizzati una sola volta, quindi $O(m)$.

DFS con stack

```
poni explored[s] = true e explored[v] = false ∀v
S = ø
T = ø
S.push(s)
while (S ≠ ø)
   S.top(u)
```

```
    if(c'è un nodo v adiacente ad u non ancora esaminato)
        if(explored[v] = false)
            explored[v] = true
            T = T U {(u, v)}
            S.push(v)
    else S.pop()
return T
```

Struttura Pseudocodice

- **Inizializzazione**: utilizziamo un array `explored` dove impostiamo il nodo sorgente `s` a `true` e tutti gli altri nodi `v` a `false`. Inizializziamo uno stack vuoto `S` e un albero vuoto dei percorsi `T` ed effettuiamo un'operazione di `push()`, cioè inseriamo `s` in cima allo stack.
- **Ciclo principale**: finché lo stack non è vuoto, esaminiamo un nodo `u` in cima allo stack tramite un'operazione di `top()` e controlliamo se c'è un arco `(u, v)` incidente non ancora esaminato. Se `v` non è stato ancora esplorato, lo esploriamo e aggiungiamo l'arco a T. Poi, effettuiamo un'operazione di `push()`, cioè inseriamo `v` in cima allo stack. Altrimenti, se non ci sono archi incidenti da esplorare, tramite un'operazione di `pop()`, rimuoviamo il nodo `u` dallo stack.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n + m)$, dove n è il numero dei nodi e m è il numero degli archi. Ogni nodo viene esplorato esattamente una volta, quindi $O(n)$, la stessa cosa per gli archi, che vengono analizzati una sola volta, quindi $O(m)$.

AllComponents

```
poni discovered[v] = false ∀v
    foreach nodo v in G
        if(discovered[v] = false)
            BFS(u)
```

Struttura Pseudocodice

- **Inizializzazione**: per ogni nodo `v` del grafo, inizializziamo un array `discovered` a false.
- **Ciclo principale**: scorriamo tutti i nodi del grafo. Se un nodo `v` non è stato ancora scoperto, allora chiamiamo la funzione `BFS()` sul nodo, esplorando l'intera componente connessa a cui `v` appartiene.

Tempo di Esecuzione

Il tempo di esecuzione di questo è algoritmo è $O(n+m)$. L'inizializzazione scorre tutti i nodi del grafo, quindi $O(n)$. Nella `BFS()` ogni nodo viene visita i suoi vicini e noi sappiamo che `BFS()` ha tempo di esecuzione $O(n+m)$.

Ordinamento Topologico

```
if (c'è un nodo v senza archi entranti) {
    cancella v da G
    L = OrdinamentoTopologico(G-{v})
    aggiungi v all'inizio di L
    return L
else return lista vuota
```

Struttura Pseudocodice

- **Condizione principale**: se c'è un nodo senza archi entranti, cancelliamo quel nodo da `G` e richiamiamo ricorsivamente l'algoritmo su `G-{v}`, salvando il grafo risultante in una lista `L`. Il nodo `v`, inizialmente rimosso, viene aggiunto a `L`.
- **Fallimento**: ovviamente, se c'è un nodo senza archi entranti, l'ordinamento topologico non è possibile, quindi viene semplicemente ritornata una lista vuota.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n^2)$, dove n rappresenta il numero dei nodi. Per scorrere il grafo e verificare la presenza di nodi senza archi entranti,

ci vuole tempo $O(n)$. Questa operazione viene fatta per ciascun nodo, quindi per n volte.

Ordinamento Topologico con Aggiunta

```
foreach nodo u in G
   foreach nodo v adiacente ad u
      count[v]++
S = ø
foreach nodo u in G
   if(count[u]=0)
      S.push(u)
L = ø
OrdinamentoTopologico(G)
   if(S ≠ 0)
    S.pop(u)
    cancella u da G con tutti i suoi archi uscenti
    foreach nodo v adiacente a u
       count[v]--;
       if(count[v]=0)
          S.push(v)
      L = OrdinamentoTopologico(G-{v})
      aggiungi v all'inizio di L
      return L
   else return lista vuota
```

Struttura Pseudocodice

- **Conteggio archi entranti**: per ogni nodo `u` e per ogni nodo `v` adiacente ad `u`, incrementiamo una variabile `count` che conterrà il conteggio degli archi entranti.
- **Pila con i nodi senza archi entranti**: inizializziamo uno stack `S` vuoto, per ogni nodo `u` che non ha archi entranti, quindi con `count[u]=0`, inseriamo `u` in `S`.

- **Algoritmo principale**: inizializziamo una lista vuota `L` e, se `S` non è vuoto, prendiamo un nodo `v` da `S` e rimuoviamolo, aggiungendolo alla lista `L` e cancellandolo da `G` con tutti i suoi archi uscenti. Per ogni nodo `u` adiacente a `v`, riduciamo il conteggio degli archi entranti. Se `count[u]` si azzera, allora mettiamo `u` in `S`. Dopodiché, richiamiamo ricorsivamente l'algoritmo su `L` e aggiungiamo `v` a `L`, restituendo quest'ultima. Altrimenti, se lo stack è vuoto, ritorniamo una lista vuota.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n+m)$. Il primo ciclo annidato scorre tutti gli archi una sola volta, quindi $O(m)$. Il secondo ciclo itera su ogni nodo, quindi $O(n)$. Ogni nodo viene inserito e rimosso una sola volta da S e ogni arco viene considerato una sola volta per la diminuzione del conteggio, quindi $O(n+m)$.

Interval Scheduling

```
ordina i job in base al tempo di fine in modo che f_1 ≤ f_2 ≤ ... ≤ f_n
f = 0
A = ø
for j=1 ad n{
   if(s_j ≥ f)
      A = A U {j}
      f = f_j
      }
return A
```

Struttura Pseudocodice

- **Inizializzazione**: ordiniamo i job in base ai loro tempi di fine, questo ci garantisce la strategia greedy, cioè selezionato il job che finisce prima. Inizializziamo una variabile `f=0`, che indica il tempo di fine dell'ultimo job selezionato e un insieme `A` vuoto, che rappresenta il sottoinsieme di cardinalità massima di job a due a due compatibili, cioè quindi l'obiettivo finale del nostro algoritmo.

- **Ciclo principale**: scorriamo tutti i job da `1` a `n` e verifichiamo quali job possiamo inserire in `A`. Se il job `j` inizia dopo il tempo di fine dell'ultimo job, allora possiamo aggiungerlo ad `A` e possiamo aggiornare `f` col valore di `f_j`.
- **Terminazione**: ritorniamo l'insieme `A`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$. Per ordinare i job, possiamo utilizzare un algoritmo noto efficiente, ad esempio `MergeSort`, che ha come tempo $O(n \log n)$, dove n è il numero di job. Dato che ciascun job viene esaminato una sola volta, allora il `for` avrà tempo $O(n)$, ma è trascurabile e andiamo a prendere il termine dominante.

Partizionamento di Intervalli

```
ordina gli intervalli in base al tempo di inizio in modo che s_1 ≤ s_2 ≤ ... ≤ s_n
d = 0
S = ø
for j=1 ad n{
   if(l'intervallo j può essere assegnato ad una risorsa v già allocata)
      assegna la risorsa v all'intervallo j
      S = S U {(j, v)}
   else
      alloca una nuova risorsa d+1
      assegna la nuova risorsa d+1 all'intervallo j
      S = S U {(j, d+1)}
      d = d+1
   }
   return S
```

Struttura Pseudocodice

- **Inizializzazione**: ordiniamo gli intervalli in base ai loro tempi di inizio, questo ci garantisce la strategia greedy, cioè ci permette di assegnare ogni intervallo alla prima risorsa disponibile in modo ordinato. Inizializziamo il contatore di risorse `d` a `0`, che terrà traccia di tutte le risorse utilizzate e un insieme `S`

come insieme vuoto, che conterrà le coppie intervallo-risorsa per le assegnazioni finali.

- **Ciclo principale**: scorriamo tutti gli intervalli da `1` a `n` e verifichiamo se esiste una risorsa `v` già utilizzata che non abbia conflitti di sovrapposizione con `j`. Se esiste, allora assegniamo `v` a `j` e aggiungiamo la coppia `(j, v)` ad `S`. Altrimenti, se tutte le risorse sono occupate, allochiamo una nuova risorsa `d+1` e assegniamola a `j`, inserendo la coppia `(j, d+1)` ad `S` e aggiornando `d` con `d+1`.
- **Terminazione**: ritorniamo l'insieme `S`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$. Per ordinare gli intervalli, possiamo utilizzare un algoritmo noto efficiente, ad esempio `MergeSort`, che ha come tempo $O(n \log n)$, dove n è il numero di intervalli. Dato che ciascun intervallo viene esaminato una sola volta, allora il `for` avrà tempo $O(d)$, ma è trascurabile e andiamo a prendere il termine dominante.

Minimizzazione dei ritardi

```
ordina i job in base alle scadenze in modo che d_1 ≤ d_2 ≤ ... ≤ d_n
t = 0
for j=1 ad n{
    assegna il job j all'intervallo [t, t+t_j]
    s_j = t
    f_j = t+t_j
    t = t+t_j
    }
return [s_1, f_1], ..., [s_n, f_n]
```

Struttura Pseudocodice

- **Inizializzazione**: ordiniamo i job in base alle scadenze, in modo da concentrarci sul job con la scadenza più vicina, riducendo l'accumulo di ritardi

significativi. Inizializziamo una variabile `t` che rappresenta l'inizio dell'elaborazione.

- **Ciclo principale**: scorriamo tutti i job da `1` a `n` e assegniamo il job `j` all'intervallo di tempo corrente che inizia a t e finisce a `t+t_j`, dove `t_j` è il tempo necessario per completare `j`. Memorizziamo `t` in `s_j`, quindi come tempo di inizio del job e `t+t_j` come tempo di fine `f_j`. Dopodiché, aggiorniamo nuovamente `t` a `t+t_j`.
- **Terminazione**: ritorniamo gli intervalli di tempo `[s_1, f_1], ..., [s_n, f_n]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$. Per ordinare gli intervalli, possiamo utilizzare un algoritmo noto efficiente, ad esempio `MergeSort`, che ha come tempo $O(n \log n)$, dove n è il numero di intervalli. Dato che il job viene assegnato ad un intervallo temporale, allora il `for` avrà tempo $O(n)$, ma è trascurabile e andiamo a prendere il termine dominante.

Dijkstra con coda a priorità e albero dei cammini minimi

```
S = ø
T = ø
Q = ø
foreach u in G
   if(u=s)
      d(u) = 0
      Q.enqueue(0, u)
   else
      d(u) = ∞
      Q.enqueue(∞, u)
   pred[u] = null
while (Q ≠ ø)
 Q.extractMin(d(u), u)
 S = S U {u}
 foreach arco e = (u, v)
```

```
    if( v ∉ S && d(u)+l_e < d(v))
        Q.changeKey(d(u)+l_e, d(v))
        pred[v] = u
        T = T U {(u, v)}
return T
```

Struttura Pseudocodice

- **Inizializzazione**: inizializziamo l'insieme `S` dei nodi esplorati come insieme vuoto. Facciamo la stessa cosa con `T`, l'albero dei percorsi minimi e con `Q`, ossia la coda a priorità che contiene i nodi con le loro distanze. Scorriamo tutti i nodi del grafo se il nodo `u=s`, quindi corrisponde alla sorgente, allora impostiamo la distanza `d(u)=0` e inseriamo la coppia `(0, u)` nella coda. Altrimenti, impostiamo la distanza a `∞` e inseriamo la coppia `(∞, u)` nella coda. Inizializziamo il predecessore di ogni nodo a `null`.
- **Ciclo principale**: fino a quando la coda non è vuota, estraiamo il nodo con distanza minima dalla coda e lo andiamo a inserire nell'insieme dei nodi esplorati. Esaminiamo tutti gli archi che partono da `u` e controlliamo se un ipotetico nodo `v` adiacente ad `u` non sia stato ancora esplorato e che la distanza di `u` più il costo dell'arco è minore della distanza di `v`. In caso affermativo, , aggiorniamo la distanza di `v` nella coda a priorità. Non ci dimentichiamo di impostare `u` come predecessore di `v` e di aggiungere l'arco all'albero `T`
- **Terminazione**: ritorniamo l'albero `T` dei cammini minimi.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O((n+m)\log n)$. L'inizializzazione della coda richiede $O(n \log n)$, dove n è il numero di nodi del grafo, mentre le restanti inizializzazioni richiedono $O(n)$. La coda è implementata con un heap binario, che ha una profondità di $\log n$, quindi, poiché estraiamo un nodo per ogni nodo del grafo, allora $O(n \log n)$. L'operazione di aggiornamento della chiave richiede $O(\log n)$, mentre per tutti gli archi del grafo, la fase di aggiornamento ha tempo totale $O(m \log n)$.

Prim con albero

```
S = ø
T = ø
Q = ø
foreach u in G
   if(u==s)
      a[u] = 0
      Q.enqueue(0, u)
   else
      a[u] = ∞
      Q.enqueue(∞, u)
   pred[u] = null
while (Q ≠ ø)
   Q.extractMin(a[u], u)
   S = S U {u}
   if(pred[u] ≠ null)
      T = T U {(pred[u], u)}
   foreach arco e = (u, v)
      if(v ∉ S && (c_e < a[v]))
         Q.changeKey(v, c_e)
         pred[v] = u
return T
```

Struttura Pseudocodice

- **Inizializzazione**: inizializziamo l'insieme `S` dei nodi esplorati come insieme vuoto. Facciamo la stessa cosa con `T`, l'albero MST e con `Q`, ossia la coda a priorità che contiene i nodi con i loro costi. Scorriamo tutti i nodi del grafo se il nodo `u=s`, quindi corrisponde alla sorgente, allora impostiamo il costo `a[u]=0` e inseriamo la coppia `(0, u)` nella coda. Altrimenti, impostiamo il costo a `∞` e inseriamo la coppia `(∞, u)` nella coda. Inizializziamo il predecessore di ogni nodo a `null`.
- **Ciclo principale**: fino a quando la coda non è vuota, estraiamo il nodo `u` con distanza minima e inseriamo il nodo `u` nell'insieme `S` dei nodi esplorati. Se il

nodo ha un predecessore, aggiungi l'arco `(pred[u], u)` a `T`. Esaminiamo tutti gli archi incidenti ad `u`. Se il nodo `v` non è stato esplorato e il costo dell'arco è minore del costo di `a[v]` allora aggiorniamo il costo di `v` nella coda e aggiorniamo `u` come predecessore di `v`.

- **Terminazione**: ritorniamo l'MST `T`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O((n+m)\log n)$, dove n è il numero di nodi e m il numero degli archi. L'inizializzazione della coda richiede $O(n)$, poiché dobbiamo scorrere tutti i nodi, mentre l'inserimento nella coda richiede $O(n \log n)$, poiché andremo ad inserire tutti gli n nodi. La fase di estrazione viene eseguita una volta per ogni nodo, quindi $O(\log n)$, mentre la fase di aggiornamento delle chiavi richiede $O(\log n)$ e viene eseguita m volte, quindi $O(m \log n)$.

Kruskal

```
ordina gli archi in base ai costi in modo che c_1 ≤ c_2 ≤ ... ≤ c_n
T = ø
foreach u in G
   makeSet(u)
for i=1 ad m{
   if(find(u) ≠ find(v))
      T = T U {(u, v)}
      union(u, v)
      }
return T
```

Struttura Pseudocodice

- **Inizializzazione**: ordiniamo gli archi in base ai costi, così da considerare prima quelli di costo minore e inizializziamo l'albero MST inizialmente vuoto. Usando l'operazione `makeSet()` della struttura data `UnionFind()`, creiamo un insieme distinto per ogni nodo del grafo.

- **Ciclo principale**: scorriamo tutti gli archi da `1` a `m` e, per ciascun arco i-esimo, usiamo l'operazione `find()` per controllare se `u` e `v` sono in insieme differenti. Se sì, aggiungiamo l'arco a `T` e, tramite un'operazione di `union()`, uniamo i due insiemi.
- **Terminazione**: ritorniamo l'MST `T`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(m \log m + m \log n)$, dove n è il numero di nodi e m il numero degli archi. Per ordinare gli archi in base ai costi, possiamo usare uno degli algoritmi noti di ordinamento, che sappiamo essere efficienti, come `MergeSort()`, che ha tempo $O(m \log m)$. Creare un insieme distinto per ogni nodo richiede $O(n)$, poiché dobbiamo scorrere ogni nodo. Ogni `find()` e `union()` vengono eseguiti con una struttura dati `UnionFind()` e richiede $O(\log n)$, quindi la complessità è $O(m \log n)$.

Interval Scheduling Pesato

```
ordina i job in base al tempo di fine f_1 ≤ f_2 ≤ ... ≤ f_n
calcola p(1), p(2), ..., p(j)
for j=1 ad n{
   M[j] = ø
   }
IntervalScheduling(j){
   if (j=0)
      return 0
   if (M[j] = ø)
      M[j] = max{v_j + IntervalScheduling(p(j)), IntervalScheduling(j-1)}
return M[j]
}
```

Struttura Pseudocodice

- **Inizializzazione**: ordiniamo i job in base ai loro tempi di fine, così da scegliere i job che terminano prima e calcoliamo, per ogni job, il `p(j)`, cioè il job

compatibile immediatamente precedente. Scorriamo tutti i job da `1` a `n` e inizializziamo l'array `M[j]` della *memoization*, inizialmente vuoto.

- **Funzione Ricorsiva**: se `j=0`, allora non ci sono job, quindi ritorniamo `0`. Se l'array della *memoization* non è stato ancora calcolato, allora calcoliamo come il massimo tra l'inclusione e l'esclusione del job.
- **Terminazione**: restituiamo l'array `M[j]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$, dove n è il numero di job. Per ordinare i job, possiamo scegliere un algoritmo di ordinamento noto, che sappiamo essere efficiente, come `MergeSort()`, che ha quindi, tempo $O(n \log n)$. Il calcolo dei `p(j)` viene effettuato usando una ricerca binaria sugli intervalli già ordinati, quindi $O(n \log n)$. L'inizializzazione della tabella richiede $O(n)$, poiché si itera su tutti i job. Ogni chiamata ricorsiva richiede tempo costante e, grazie alla *memoization*, ogni job viene calcolato una sola volta, quindi $O(n)$.

Stampa della Soluzione Ottima

```
FindSolution(j){
   if(j=0)
      return 0
   else if(v_j + IntervalScheduling(p(j))>M[j-1])
      FindSolution(p(j))
      print j
   else
      FindSolution(j-1)
}
```

Pseudocodice Iterativo

```
ordina i job in base al tempo di fine f_1 ≤ f_2 ≤ ... ≤ f_n
calcola p(1), p(2), ..., p(j)
IterativeIntervalScheduling(j){
   M[0] = 0
```

```
    for j=1 a n{
        M[j] = max{v_j + M[p(j)], M[j-1]}
    return M[j]
}
```

Subset Sum

```
for i=0 ad n{
    for w=0 a W{
        M[i, w] = ∅
        }
    }
SubsetSum(i, w):
    if i=0
        return 0
    if (M[i, w] = ∅)
        if w_i > w
            M[i, w] = SubsetSum(i-1, w)
        else
            M[i, w] = max{SubsetSum(i-1, w), w_i+SubsetSum(i-1, w-w_i)}
return M[i, w]
```

Struttura Pseudocodice

- **Inizializzazione**: scorriamo tutti i job da `0` a `n` e tutti i pesi da `0` a `W` e inizializziamo la matrice della *memoization* come vuota.
- **Funzione Ricorsiva**: innanzitutto, controlliamo se ci sono job, altrimenti la matrice sarà pari a `0`. Se la matrice è già stata calcolata, allora la ritorniamo semplicemente. Altrimenti, se il job ha un peso maggiore del limite di peso, allora il job è escluso dalla soluzione, altrimenti, è il massimo tra l'escluderlo e includerlo.
- **Terminazione**: restituiamo l'array `M[i, w]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \cdot W)$, dove n è il numero di job e W è il limite di peso per il processore. Questo perché ogni cella della tabella viene calcolata una sola volta grazie alla *memoization*.

Stampa della Soluzione Ottima

```
FindSubset(i, w){
   if(i=0)
      return 0
   else if(M[i, w]=M[i-1, w])
      FindSubset(i-1, w)
   else
      FindSubset(i-1, w-w_i)
      print i
}
```

Pseudocodice Iterativo

```
for i=0 ad n{
   for w=0 a W{
      M[i, w] = ⌀
      }
   }
for i=0 ad n{
   for w=0 a W{
      if i=0
       return 0
      if (M[i, w] = ⌀)
         if w_i > w
            M[i, w] = M[i-1, w]
         else
            M[i, w] = max{M[i-1, w], w_i+M[i-1, w-w_i]}
return M[i, w]
```

Problema dello zaino

```
for i=0 ad n{
   for w=0 a W{
      M[i, w] = ø
      }
   }
Zaino(i, w):
   if i=0
      return 0
   if (M[i, w] = ø)
      if w_i > w
         M[i, w] = Zaino(i-1, w)
      else
         M[i, w] = max{Zaino(i-1, w), v_i+Zaino(i-1, w-w_i)}
return M[i, w]
```

Struttura Pseudocodice

- **Inizializzazione**: scorriamo tutti gli oggetti da `0` a `n` e tutti i pesi da `0` a `W` e inizializziamo la matrice della *memoization* come vuota.
- **Funzione Ricorsiva**: innanzitutto, controlliamo se ci sono oggetti, altrimenti la matrice sarà pari a `0`. Se la matrice è già stata calcolata, allora la ritorniamo semplicemente. Altrimenti, se l'oggetto ha un peso maggiore del limite di peso, allora l'oggetto è escluso dalla soluzione, altrimenti, è il massimo tra l'escluderlo e includerlo.
- **Terminazione**: restituiamo l'array `M[i, w]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \cdot W)$, dove n è il numero di oggetti e W è il limite di peso dello zaino. Questo perché ogni cella della tabella viene calcolata una sola volta grazie alla *memoization*.

Stampa della Soluzione Ottima

```
FindZaino(i, w){
   if(i=0)
      return 0
   else if(M[i, w]=M[i-1, w])
      FindZaino(i-1, w)
   else
      FindZaino(i-1, w-w_i)
      print i
}
```

Pseudocodice Iterativo

```
for i=0 ad n{
   for w=0 a W{
      M[i, w] = ⌀
      }
   }
for i=0 ad n{
   for w=0 a W{
      if i=0
       return 0
      if (M[i, w] = ⌀)
         if w_i > w
            M[i, w] = M[i-1, w]
         else
            M[i, w] = max{M[i-1, w], v_i+M[i-1, w-w_i]}
return M[i, w]
```

Minimum Coin Change

```
for i=1 ad n{
      M[i, 0] = 0
   }
```

```
for v=1 a V{
   M[1, v] = v
   }
MinimumCoinChange(i, v){
   if(M[i, v] = ⌀)
      if(v_i>v)
         M[i, v] = MinimumCoinChange(i-1, v)
      else
         M[i, v] = min{MinimumCoinChange(i-1, v), 1 + MinimumCoinChange(i, v-v
   return M[i, v]
}
```

Struttura Pseudocodice

- **Inizializzazione**: scorriamo tutti i valori delle monete e inizializziamo la tabella della *memoization* come uguale a `0`. Se abbiamo solo la monete di valore `1`, allora per ottenere `v` servono esattamente `v` monete, quindi inizializziamo la tabella come `M[1, v] = v`.
- **Ciclo principale**: con un doppio for annidato, scorriamo tutti i valori delle monete e tutta la banconota, se `v_i > v`, significa che il valore della moneta eccede col valore della banconota, quindi usiamo il calcolo precedente. Altrimenti, scegliamo il minimo tra l'esclusione e l'inclusione della moneta.
- **Terminazione**: restituiamo l'array `M[i, v]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \cdot V)$, dove n sono i tipi di monete e V è il valore della banconota. Questo perché ogni cella della tabella viene calcolata una sola volta grazie alla *memoization*.

Stampa della Soluzione Ottima

```
FindCoinChange(i, v){
   if(v=0)
      return 0
   else if(i=0)
```

```
        return 0
    else if(M[i, v]=M[i-1, v])
        FindCoinChange(i-1, v)
    else
        FindCoinChange(i, v-v_i)
        print v_i
}
```

Pseudocodice Iterativo

```
for i=1 ad n{
        M[i, 0] = 0
    }
for v=1 a V{
    M[1, v] = v
    }
for i=1 a n{
    for v=1 a V{
        if(M[i, v] = ø)
            if(v_i>v)
                M[i, v] = M[i-1, v]
            else
                M[i, v] = min{M[i-1, v], 1 + M[i, v-v_i]}
    return M[i, v]
}
```

Sottosequenza Comune Più Lunga

```
m = x.length().
n = y.length();
    for i=0 ad m{
        M[i, 0]=0
    }
    for j=0 ad n{
```

```
        M[0, j]=0
    }
b[i, j] = " "
LCS(i, j)
    if(M[i, j] = ⌀)
        if x[i] = y[j]
            M[i, j] = LCS(i-1, j-1)+1
            b[i, j] = "↖";
        if M[i-1, j] ≥ M[i, j-1]
            M[i, j] = LCS(i-1, j)
            b[i, j] = "↑";
        else
            M[i, j] = LCS(i, j-1)
            b[i, j] = "←";
return M[i, j]
```

Struttura Pseudocodice

- **Inizializzazione**: scorriamo tutti i caratteri delle due stringhe, inizializzando la prima riga e la prima colonna a `0`, ossia la stringa vuota ϵ.
- **Ciclo principale**: scorriamo nuovamente tutti i caratteri delle due stringhe. Se c'è una corrispondenza tra i caratteri, si aggiunge 1 al valore precedente in diagonale e si appone una freccia, appunto, diagonale. Altrimenti, in caso di mancata corrispondenza, andiamo a scegliere il massimo tra il valore della cella subito sopra o di quella a sinistra, apponendo la freccia corrispondente.
- **Terminazione**: restituiamo l'array `M[x, y]`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(m \cdot n)$, dove m è la lunghezza di x e n è la lunghezza di y e, grazie alla *memoization*, ogni cella viene calcolata soltanto una volta.

Stampa della Soluzione Ottima

```
FindLCS(x, y){
   if(i=0) || (j=0)
      return 0
   else if(b[i, j] = "↖")
      FindLCS(i-1, j-1)
      print x[i]
   else if(b[i, j] = "↑")
      FindLCS(i-1, j)
   else
      FindLCS(i, j-1)
}
```

Pseudocodice Iterativo

```
m = x.length().
n = y.length();
   for i=0 ad m{
      M[i, 0]=0
   }
   for j=0 ad n{
      M[0, j]=0
   }
b[i, j] = " "
for i=1 a m{
   for j=1 a n{
      if(M[i, j] = ø)
         if x[i] = y[j]
            M[i, j] = M[i-1, j-1]+1
            b[i, j] = "↖";
         else if M[i-1, j] ≥ M[i, j-1]
            M[i, j] = M[i-1, j]
            b[i, j] = "↑";
         else
            M[i, j] = M[i, j-1]
```

```
            b[i, j] = "←";
return M[i, j]
```

Bellman-Ford

```
BellmanFord(G, t)
foreach nodo v ∈ V {
   M[v] = ∞
   S[v] = null
   }
M[t] = 0
S[t] = t
for i=1 ad V-1{
   aggiornamento = false
   foreach (v, w) ∈ E{
      if M[w] > M[v] + c_vw
         M[w] = M[v] + c_vw
         S[w] = v
         aggiornamento = true
      }
   if aggiornamento = false
               break;
   }
return M, S
```

Struttura Pseudocodice

- **Inizializzazione**: scorriamo tutti i nodi del nostro grafo, inizializzando l'array delle distanze `M` e impostando tutte le distanze a `∞`. Inizializziamo l'array dei percorsi `S` come `null`, dato che il predecessore è inizialmente nullo. Inizializziamo la distanza di `t` a `0`, poiché già ci troviamo nella destinazione e ovviamente il percorso di `t` è uguale a `t`.
- **Ciclo principale**: scorriamo nuovamente tutti i nodi da `1` a `V-1` e impostiamo un flag `aggiornamento` a `false` per tenere traccia di eventuali cambiamenti.

Iteriamo su tutti gli archi del grafo e aggiorniamo la distanza minima se troviamo un percorso più breve. Usiamo `aggiornamento` per terminare anticipatamente se nessun valore è cambiato, poiché così riduciamo iterazioni inutili

- **Terminazione**: restituiamo le tabelle `M` e `S`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(V \cdot E)$, dove V è l'insieme dei nodi ed E l'insieme degli archi. L'inizializzazione richiede $O(V)$, poiché scorre ogni nodo e imposta le distanze e i percorsi. Nel caso peggiore, l'algoritmo itera $O(V-1)$, mentre scorre tutti gli archi $O(E)$ volte.

Stampa della Soluzione Ottima

```
FindShortestPath(S, t, v){
   if(v=t)
      print v
      return 0
   else if(S[v] == null)
      print "Nessun percorso da t a v";
      return 0
   else
      FindShortestPath(S, t, S[v]
      print v
}
```

Pseudocodice Ricorsivo

```
foreach nodo v ∈ V {
   M[v] = ∞
   S[v] = null
   }
M[t] = 0
S[t] = t
```

```
BellmanFord(G, s, i){
    if i=0
        return
    aggiornamento = false
    foreach(v, w) ∈ E{
        if M[w] > M[v]+c_vw
            M[w] = M[v]+c_vw
            S[w] = v
            aggiornamento = true
    if aggiornamento = false
        return
    BellmanFord(G, s, i-1)
```

MergeSort

```
MergeSort(A, sin, des)
    if(sin < des){
        cen = (sin+des)/2
        MergeSort(A, sin, cen-1)
        MergeSort(A, cen, des)
        Merge(A, sin, cen, des)
        }

Merge(A, sin, cen, des)
    B[0]
    i=sin
    j=cen+1
    k=0
    while((i≤cen)&&(j≤des)){
        if(A[i]≤A[j]){
            B[k] = A[i]
            i++
        else{
            B[k] = A[j]
            j++
```

```
        }
        k++
      }
    for(i≤cen;i++;k++){B[k]=A[i]}
    for(j≤des;j++;k++){B[k]=A[j]}
    for(i=sin;i≤des;i++){A[i]=B[i-sin]}
```

Struttura Pseudocodice

- **Algoritmo Ricorsivo Principale**: calcoliamo l'indice centrale per dividere l'array e ordiniamo ricorsivamente la metà di sinistra e la metà di destra. Dopodiché, fondiamo le due metà per avere l'array completamente ordinato.
- **Funzione Merge**: inizializziamo un array temporaneo `B` a `0` e individuiamo gli indici `i` e `j` rispettivamente per la parte sinistra e destra. Inizializziamo un contatore `k` a `0`, come indice del vettore ausiliario. Fino a quando `i` e `j` non eccedono dalle loro dimensioni, confrontiamo gli elementi delle due metà, copiandoli nell'array `B` e incrementando di volta in volta `i` e `j`. Alla fine, incrementiamo anche `k`. Infine, attraverso due `for`, copiamo gli elementi rimanenti della prima e della seconda metà, per poi copiare il contenuto di `B` nell'array originale `A`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$, dove n è la dimensione dell'input. Ogni livello di ricorsione divide l'array in due parti, quindi $\log_2(n)$ livelli di ricorsione. Ogni chiamata a `Merge()` processa $O(k)$, cioè la dimensione dell'array ausiliario e, sommando tutti i livelli, il costo totale della fusione è $O(n)$.

Paradigma Divide et Impera

- **Decomposizione**: in questo caso, l'array A viene diviso in due metà fino a quando le sottosequenze non sono di dimensione 1.
- **Ricorsione**: chiamiamo `MergeSort()` ricorsivamente sulle due metà, continuando a dividere l'array fino a quando non giungiamo a intervalli con un solo elemento.

- **Ricombinazione**: si prendono le due metà ordinate e si fondono in un'unica sequenza ordinata. Usiamo due indici i e j. Confrontiamo gli elementi delle due metà e li inseriamo in un array ausiliario B, dove poi i suoi elementi passeranno ad A.
- **Caso Ottimo e Caso Pessimo**: nel caso migliore, l'array è già ordinato, ma il comportamento dell'algoritmo non cambia, poiché i passaggi di divisione e fusione vengono comunque sempre eseguiti. Nel caso peggiore, invece, è quando l'array è ordinato in ordine inverso, ma ciò comunque non cambia il numero di confronti e operazioni rispetto al caso ottimo, quindi il tempo di esecuzione rimane comunque $O(n \log n)$.

Ricerca Binaria Ricorsiva

```
RicercaBinariaRicorsiva(A, k, sin, des)
   if(sin > des)
      return -1
   cen = (sin+des)/2
   if(k==A[cen])
      return cen
   else if(k<A[cen])
      return RicercaBinariaRicorsiva(A, k, sin, cen-1)
   else
      return RicercaBinariaRicorsiva(A, k, cen, des)
```

Struttura Pseudocodice

- **Algoritmo Ricorsivo Principale**: se l'indice di sinistra è maggiore di quello di destra, l'intervallo è vuoto, quindi l'elemento non è presente nell'array. Altrimenti, ci calcoliamo il centro. Innanzitutto, verifichiamo se l'elemento da cercare è uguale al centro e così ne restituiamo la posizione. Invece, se l'elemento centrale si trova nella parte di sinistra, allora richiamiamo ricorsivamente l'algoritmo su quella parte. Altrimenti, la richiamiamo sulla parte destra.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(\log n)$, dove n è la dimensione dell'input. Ad ogni livello di ricorsione, l'input si riduce della metà, quindi servono $\log_2(n)$ divisioni.

Paradigma Divide et Impera

- **Decomposizione**: in questo caso, l'array viene diviso a metà ad ogni passaggio, calcolando un indice centrale cen.
- **Ricorsione**: una volta diviso l'array in due metà, l'algoritmo continua a cercare ricorsivamente solo nella metà in cui può trovarsi l'elemento, quindi o a sinistra o a destra.
- **Ricombinazione**: non occorre fare nessun lavoro di ricombinazione, poiché la ricerca si conclude restituendo l'elemento trovato.
- **Caso Ottimo e Caso Pessimo**: nel caso migliore, l'elemento cercato è uguale all'elemento centrale dell'array, quindi il tempo è banalmente $O(1)$. Nel caso peggiore, invece, l'elemento si trova in una posizione tale da richiedere il massimo numero di divisioni. Il tempo rimane comunque $O(\log n)$.

QuickSort

```
QuickSort(A, sin, des)
   if(sin<des)
      pivot = scegli dall'intervallo [sin, ..., des]
      indiceFinalePivot = Distribuzione(A, sin, pivot, des)
      QuickSort(A, sin, indiceFinalePivot-1)
      QuickSort(A, indiceFinalePivot+1, des)

Distribuzione(A, sin, piv, des)
   if(piv != des)
      Scambia(piv, des)
   i=sin
   j=des-1
   while(i≤j){
      while((i≤j) && (A[i]≤A[des]))
         i++
```

```
        while((i≤j) && (A[j]≥A[des]))
            j--
        if(i<j)
            Scambia(i, j)
            i++
            j--
        }
        if(i!=des)
            Scambia(i, des)
    return i

Scambia(i, j)
    temp = A[j]
    A[j] = A[i]
    A[i] = temp
```

Struttura Pseudocodice

- **Algoritmo Ricorsivo Principale**: se l'intervallo contiene più di un elemento, selezioniamo un `pivot` randomicamente e lo posizioniamo correttamente con la funzione `Distribuzione()`, richiamando `QuickSort()` ricorsivamente sulle due metà.
- **Funzione Distribuzione**: la funzione posiziona il `pivot` nella sua posizione finale, in modo che gli elementi a sinistra siano $\leq$ e gli elementi a destra siano $>$. Innanzitutto, porta il `pivot` in fondo, usando la funzione `Scambia()`. Usiamo gli indici `i` e `j` per iterare l'array e scambiare gli elementi fuori posto, incrementando la `i` o decrementando la `j`. Alla fine, il `pivot` viene portato nella posizione corretta, usando sempre la funzione `Scambia()`.
- **Funzione Scambia**: scambia semplicemente gli indici `i` e `j` usando una variabile temporanea `temp`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n \log n)$, dove n è la dimensione dell'input. La funzione `Distribuzione()` scorre l'intervallo una sola volta, con un costo

$O(n)$. Ad ogni livello di ricorsione, l'array viene diviso in due sotto-array, ciascuno la metà della dimensione originale, quindi $O(\log n)$.

Paradigma Divide et Impera

- **Decomposizione**: in questo caso, si sceglie un `pivot` nell'array `A`, nell'intervallo compreso tra gli indici `sin` e `des` e si usa la funzione `Distribuzione()` per riordinare gli elementi attorno al pivot, facendo in modo che gli elementi $\leq$ al pivot si trovino a sinistra e quelli $>$ a destra.
- **Ricorsione**: l'algoritmo ordina ricorsivamente le due metà separate dal pivot.
- **Ricombinazione**: non occorre fare nessun lavoro di ricombinazione, poiché l'array è già completamente ordinato, dato che il pivot si trova nella posizione corretta dopo la chiamata a distribuzione.
- **Caso Ottimo e Caso Pessimo**: nel caso migliore, il pivot scelto divide sempre l'intervallo in due parti uguali, e il costo rimane comunque $O(n \log n)$. Nel caso peggiore, il pivot o è tutto a sinistra o tutto a destra. In questo caso, la ricorsione non divide l'array in due metà uguali, ma una metà ha 1 elemento e l'altra $n-1$ elementi. In questo caso, il tempo sarà $O(n^2)$.

QuickSelect

```
QuickSelect(A, sin, r, des)
      if(sin<des)
       pivot = scegli dall'intervallo [sin, ..., des]
       indiceFinalePivot = Distribuzione(A, sin, pivot, des)
       if(r==indiceFinalePivot)
          return A[indiceFinalePivot]
       else if(r<indiceFinalePivot)
          return QuickSelect(A, sin, r, indiceFinalePivot-1)
       else
          return QuickSelect(A, indiceFinalePivot+1, r, des)

Distribuzione(A, sin, piv, des)
   if(piv != des)
      Scambia(piv, des)
```

```
    i=sin
    j=des-1
    while(i≤j){
        while((i≤j) && (A[i]≤A[des]))
            i++
        while((i≤j) && (A[j]≥A[des]))
            j--
        if(i<j)
            Scambia(i, j)
            i++
            j--
        }
        if(i!=des)
            Scambia(i, des)
    return i

Scambia(i, j)
    temp = a[j]
    a[j] = a[i]
    a[i] = temp
```

Struttura Pseudocodice

- **Algoritmo Ricorsivo Principale**: nel caso base, verifichiamo se l'indice del `pivot` trovato corrisponde all'indice `r` dell'r-esimo elemento dell'array ordinato. Altrimenti, chiamiamo ricorsivamente `QuickSelect()` sulle due metà.
- **Funzione Distribuzione**: la funzione posiziona il `pivot` nella sua posizione finale, in modo che gli elementi a sinistra siano $\leq$ e gli elementi a destra siano $>$. Innanzitutto, porta il `pivot` in fondo, usando la funzione `Scambia()`. Usiamo gli indici `i` e `j` per iterare l'array e scambiare gli elementi fuori posto, incrementando la `i` o decrementando la `j`. Alla fine, il `pivot` viene portato nella posizione corretta, usando sempre la funzione `Scambia()`.
- **Funzione Scambia**: scambia semplicemente gli indici `i` e `j` usando una variabile temporanea `temp`.

Tempo di Esecuzione

Il tempo di esecuzione di questo algoritmo è $O(n)$, dove n è la dimensione dell'input. La funzione `Distribuzione()` scorre l'intervallo una sola volta, con un costo $O(n)$. L'algoritmo si occupa solo di uno dei due sotto-array in ogni iterazione, riducendo il problema a metà dell'intervallo ad ogni passo, quindi $O(n)$.

Paradigma Divide et Impera

- **Decomposizione**: si seleziona un pivot dall'array e la funzione `Distribuzione()` riorganizza gli elementi dell'array in modo che gli elementi $\leq$ del pivot vengono spostati a sinistra e quelli $>$ a destra. Alla fine, il pivot si troverà nella sua posizione finale.
- **Ricorsione**: durante la ricorsione, l'algoritmo verifica se il pivot si trova in posizione corretta. In caso affermativo, allora viene restituito l'r-esimo elemento, se si trova a destra o a sinistra, allora iteriamo solo sulla parte sinistra o destra.
- **Ricombinazione**: non occorre fare nessun lavoro di ricombinazione, poiché trovato il pivot nella posizione corretta, esso viene restituito.
- **Caso Ottimo e Caso Pessimo**: nel caso migliore, il pivot scelto divide l'intervallo in due parti quasi uguali, quindi $O(n)$. Nel caso peggiore, invece, il pivot scelto è sempre o tutto a destra o tutto a sinistra, quindi $O(n^2)$.