www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Algoritmi e Strutture Dati

Teorema fondamentale della ricorrenza (Master Theorem)
Permette di analizzare algoritmi basati sulla tecnica del divide et impera:

$$T(n) = \begin{cases} d & \text{per } n = 1 \\ aT(n/b) + cn^{\beta} & \text{per } n > 1 \end{cases}$$

Posto $\alpha = (\log a)/(\log b)$ ha soluzione
$T(n) \in \mathbf{O(n^{\alpha})}$ se $\alpha > \beta$
$T(n) \in \mathbf{O(n^{\alpha}\log n)}$ se $\alpha = \beta$
$T(n) \in \mathbf{O(n^{\beta})}$ se $\alpha < \beta$

- Esempio

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

Risolvo numericamente
$T(n) = 2T(n/2) + n/2$
$2(2T(n/4)+n/4) + n/2$
$2(2(2T(n/8)+n/8)+n/4) + n/2$
$2(2(2(...(2T(n(2^m)+n/2^m)... + n/4) + n/2$
$2^mT(1)+2^{m-1}(n/2^m) +... + 4(n/8)*2(n/4) + n/2$
$2^m + \log n*n/2$
$n + (n/2)\log n \rightarrow$ **O(nlogn)**

Risolvo con Master Theorem
$a = 2$
$b = 2$
$c = ½$
$\beta = 1$
$\alpha = (\log 2) / (\log 2) = 1$

$\alpha = \beta \rightarrow$ **O(nlogn)**

- Esempio

AlgB(**integer** n)
  integer i ← 1;
  while i ≤ n **do**
    i ← 2 * i;
  endwhile

$2^{T(n)} > n \rightarrow T(n) \in O(\log n)$

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

NOTA:
Java ha un **Garbage Collector** che elimina automaticamente gli oggetti che non vengono usati nel programma. (C++ ha bisogno di un codice specifico da parte del programmatore per l'eliminazione degli oggetti inutilizzati).

Analisi Ammortizzata (inserimento in coda in arraylist)
k = numero di resize effettuati
$m + 2m + 3m + \ldots + (k/m)m =$
$m(1 + 2 + \ldots + k/m) \in O(k^2) =$
Costo ammortizzato = $O(k^2)/k$ = **O(k)**

Min-Heap
Code con priorità: implementate da albero binario bilanciato che ci da la garanzia di avere in testa il valore con priorità più alta (min-heap).
Si aggiunge a sinistra perché nell'implementazione con vettore è comodo aggiungere così.

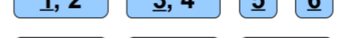
Srutture Merge-Find

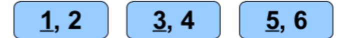
Quando creiamo un'istanza di questa struttura dati, indichiamo un numero n di gruppi che utilizzeremo. Su tali gruppi abbiamo 2 operazioni: merge (unire 2 gruppi tra loro) e find (dato uno elemento di un gruppo, viene restituito il rappresentante univoco).

Operazioni:
Mfset(integer n) - crea n insiemi disgiunti {1}, {2}, ... {n}
integer find(integer x) - restituisce il rappresentante dell'unico insieme contenente x
merge(integer x, integer y) - unisce i due insiemi che contengono x e y (se x e y appartengono già allo stesso insieme, non fa nulla). Il rappresentante può essere scelto in modo arbitrario; ad esempio, come uno dei vecchi rappresentanti degli insiemi contenenti x e y.

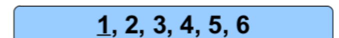
Mfset(6)

merge(1,2)

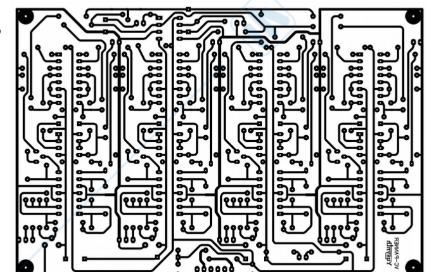
merge(3,4)


merge(5,6)


merge(2,3)

merge(4,6)


Esempio:
- Rappresentiamo il circuito elettronico con un insieme V = {1,..., n} di n nodi (pin) collegati da segmenti conduttivi.
- Indichiamo con E la lista di coppie (v_1, v_2) di pin che sono tra di loro adiacenti (collegati).
- Vogliamo pre-processare il circuito in modo da rispondere in maniera efficiente a interrogazioni del tipo: "i pin x e y sono tra loro collegati?



Creo una struttura merge-find con tutti gli n elementi e inizio a fare merge(a,b) tra tutti gli archi E tra le coppie di nodi collegati tra loro. Eseguendo poi find(z) e find(y) se

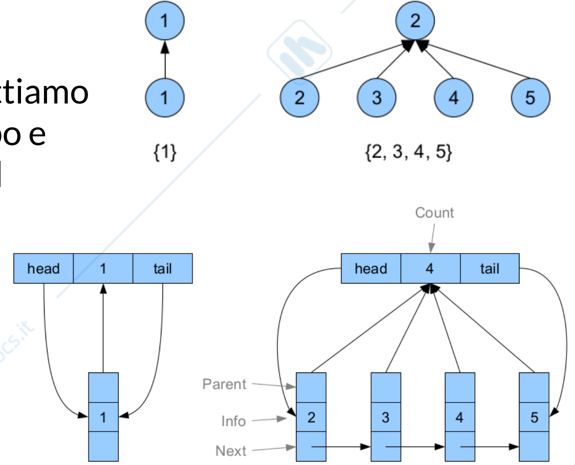
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

mi viene restituito lo stesso rappresentante univoco (dell'insieme) allora appartengono allo stesso inseme e sono quindi in qualche modo collegati. Se i rappresentati restituiti da find(?) sono diversi non sono collegati.

Implementazione Merge-Find

- Quick Find

Si pone attenzione sulla velocità di ricerca

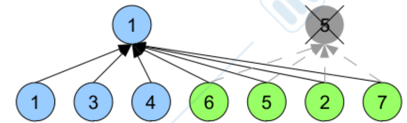


Utilizziamo delle liste (o alberi di profondità 1). Mettiamo assieme elementi che sono parte dello stesso gruppo e ognuno avrà un puntatore al suo rappresentate (del momento). Inizialmente avrò dei singoletti che puntano a se stessi, ma mano che faccio merge, più elementi punteranno al rappresentante.

A destra si ha un'implementazione più dettagliata, non è altro che una lista, il cui padre di ognuno però non è il precedente ma sempre il rappresentante univoco.

In questo modo con un costo costante siamo in grado di <u>trovare</u> il rappresentante di ogni elemento (guardando l'elemento padre dell'oggetto).

In caso di <u>merge</u> invece è più costoso perché devo modificare il padre di ogni elemento del gruppo A puntandolo all'elemento scelto come nuovo rappresentate del gruppo risultante.



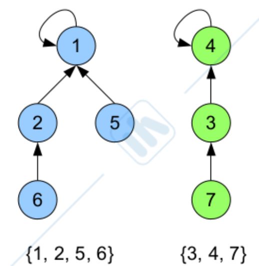
Mfset(n) – crea n liste, ciascuna contenente un intero → **O(n)**

find(x) – restituisce il riferimento al rappresentante di x (il parent dell'elemento x) → **O(1)**

merge(x,y) – Tutti gli elementi della lista contenente y vengono spostati nella lista contenente x → **O(n)**

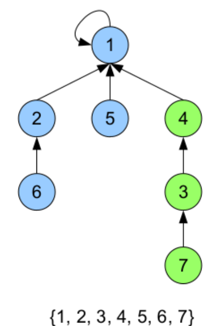
- Quick Union

Si pone attenzione sulla velocità di unione



I gruppi saranno rappresentati da alberi, di profondità arbitraria. Ogni elemento punta a un padre, il rappresentante univoco di un gruppo è la radice (che avrà come padre se stessa).

Quando andiamo a <u>fondere</u> è sufficiente far puntare la radice di un gruppo alla radice di un altro. Quando andiamo a <u>ricercare</u> invece è necessario (nel caso peggiore) risalire tutto l'albero fino al nodo radice (rappresentante univoco).

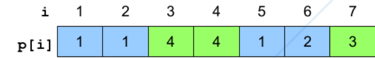


Mfset(n) - crea n alberi, ciascuno contenente un singolo intero → **O(n)**

find(x) - risale la lista degli antenati di x fino a trovare la radice e ne ritorna il contenuto come rappresentante → **O(n)**

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

merge(x,y) - rende la radice dell'albero che contiene y figlia della radice dell'albero che contiene x; costo **O(1)** nel caso ottimo (x e y sono già le radici dei rispettivi alberi), nel caso pessimo (devo anche risalire) → **O(n)**



Nota: Un modo comodo per rappresentare una foresta di alberi QuickUnion è di usare un array di interi (vettore dei padri).

Riassumendo

- Quando usare **QuickFind**?
Quando le **merge()** sono rare e le **find()** frequenti.
- Quando usare **QuickUnion**?
Quando le **find()** sono rare e le **merge()** frequenti.

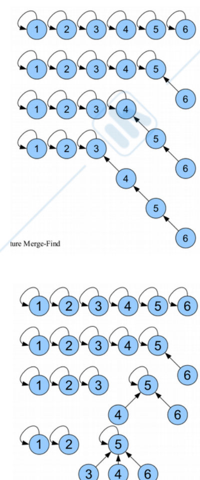
	QuickFind	QuickUnion
`Mfset(n)`	O(*n*)	O(*n*)
`merge(x,y)`	O(*n*)	O(1) caso ottimo O(*n*) caso pessimo
`find(x)`	O(1)	O(1) caso ottimo O(*n*) caso pessimo

• QuickFind - Euristica sul peso
Diminuire il costo dell'operazione di merge():
- Memorizzare nel nodo rappresentante il numero di elementi dell'insieme; la dimensione corretta può essere mantenuta in tempo O(1).
- Appendere l'insieme con meno elementi a quello con più elementi.

Mfset(n) → O(n)
find(x) → O(1)
merge(x,y) - tutti i nodi della lista con meno nodi vengono spostati nella lista con più nodi, dimostriamo che il costo complessivo di una sequenza di (n - 1) merge è O(n log n) quindi ammortizzato per una operazione → **O(log n)**

Dimostrazione (perchè O(n log n)?)
Ogni volta che una foglia di un albero QuickFind cambia padre, entra a far parte di un insieme (che con questa tecnica euristica è per definizione più grande) che ha almeno il doppio di elementi di quello cui apparteneva. Al massimo quindi si cambia padre $\log_2 n$ volte. Tuttavia abbiamo n-1 merge massimi effettuabili e quindi ogni nodo può cambiare logn volte padre → O(nlogn)

• QuickUnion – Euristica sul rango
Nel caso pessimo, il find viene effettuato su una lista molto lunga, avrà complessità **n** (si fa un find sull'ultimo nodo). Si vuole quindi far si che l'altezza di un albero non diventi troppo elevata.

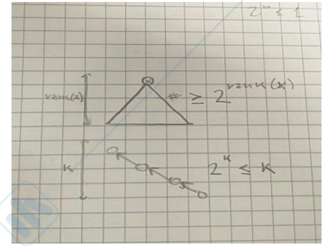


Rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto. Ogni radice quindi dovrà mantenere informazioni sul proprio rango. Grazie a questa euristica potremo avere alberi con profondità al massimo logaritmica.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

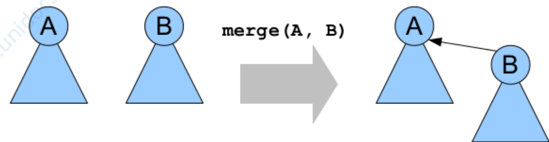
Rango: il rango rank(x) di un nodo x è il numero di archi del cammino più lungo fra x e una foglia sua discendente. Cioè <u>l'altezza dell'albero</u> radicato in x.



Usiamo A e B per rappresentare 2 alberi che vanno uniti.
A ∪ B denota l'insieme ottenuto dopo l'unione.
rank(A ∪ B) è l'altezza dell'albero che denota A ∪ B.
|A ∪ B| è il numero di nodi dell'albero A ∪ B, e risulta |A ∪ B| = |A| + |B| perché stiamo unendo sempre insiemi <u>disgiunti</u>.

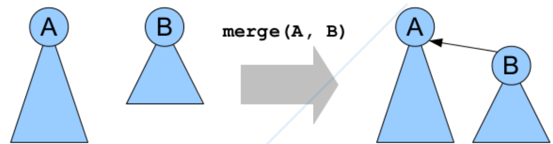
Vogliamo dimostrare che se io ho un'istanza di QuickUnion e qualunque esecuzione di merge, raggiungerò una situazione dove un qualsiasi albero X ha <u>almeno</u> $2^{rank(x)}$ nodi.
Dimostrazione per induzione:

- Caso in cui la (k+1)-esima operazione di merge viene effettuata su due alberi con uguale altezza. **rank(A) = rank(B)**



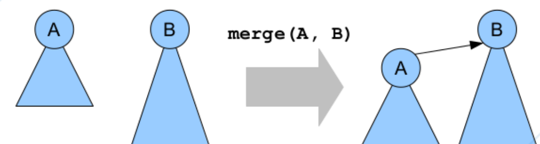
|A ∪ B| = |A| + |B|
rank(A ∪ B) = rank(A) +1
Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
Quindi $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} = 2 * 2^{rank(A)} = 2^{rank(A)+1} = 2^{rank(A \cup B)}$

- Caso in cui la (k+1)-esima operazione di merge viene effettuata su due alberi in cui A ha profondità maggiore dell'albero B.
rank(A) > rank(B)



|A ∪ B| = |A| + |B|
rank(A ∪ B) = rank(A) [perché l'altezza dell'albero A ∪ B è uguale all'altezza dell'albero A]
Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
Quindi $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(A)} = 2^{rank(A \cup B)}$

- Caso in cui la (k+1)-esima operazione di merge viene effettuata su due alberi in cui B ha profondità maggiore dell'albero A.
rank(A) < rank(B)



|A ∪ B| = |A| + |B|
rank(A ∪ B) = rank(B) [perché l'altezza dell'albero A ∪ B è uguale all'altezza dell'albero B]
Per ipotesi induttiva, $|A| \geq 2^{rank(A)}$, $|B| \geq 2^{rank(B)}$
Quindi $|A \cup B| = |A| + |B| \geq 2^{rank(A)} + 2^{rank(B)} > 2^{rank(B)} = 2^{rank(A \cup B)}$

L'altezza di un albero QuickUnion A è rank(A). Da quanto appena visto, $2^{rank(A)} \leq n$.
Quindi **altezza = rank(A) ≤ ($\log_2 n$).**
Mfset(n) → O(n)
find(x) → O(logn)
merge(x,y) → O(logn)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Riepilogo con Euristiche

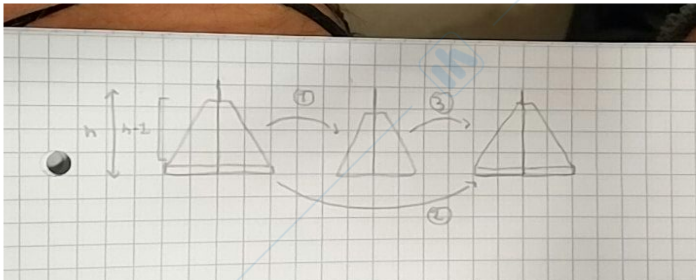
	QuickFind	QuickUnion	QuickFind eur. peso	QuickUnion by eur. rank
`Mfset(n)`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`merge`	$O(n)$	$O(1)$ ottimo $O(n)$ pessimo	$O(\log n)$ ammortizzato	$O(1)$ ottimo $O(\log n)$ pessimo
`find`	$O(1)$	$O(n)$	$O(1)$	$O(\log n)$

Teniche algoritmiche: Divide Et Impera

Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (top-down).
Ambito: problemi di decisione, ricerca.

Tre fasi:
- Divide: Dividi il problema in sotto-problemi indipendenti, di dimensioni "minori".
- Impera: Risolvi i sotto-problemi ricorsivamente.
- Combina: Unisci le soluzioni dei sotto-problemi per costruire la soluzione del problema di partenza.

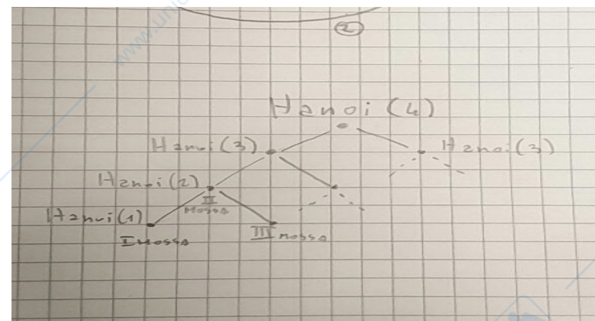
Esempio: Torre di Hanoi



L'esempio a sinistra indica in modo generico la procedura, cambia l'ordine dei parametri durante l'esecuzione, (invece che passo 1 in p1, 2 in p2 e 3 in p3, la torre da spostare potrebbe essere in uno degli altri due pioli, e verrebbero usati come "appoggio" i rimanenti).

Divide:
→ n-1 dischi da p1 a p2
→ 1 disco da p1 a p3
→ n-1 dischi da p2 a p3

Impera
→ Esegui ricorsivamente gli spostamenti



Calcolo costo computazionale

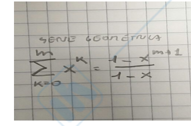
$$T(n) = \begin{cases} 1 & \text{per } n = 1 \\ 2T(n-1) + 1 & \text{per } n > 1 \end{cases}$$

n - 1 = n/b → questa equazione non è soddisfabile per nessun b, quindi il calcolo del costo non è risolvibile con il master theorem. Questo perché il M.T. è utilizzabile quando il problema si divide in parti uguali. Uso allora la relazione di ricorrenza

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

$T(n) = 2T(n-1) + 1$
$= 2(2T(n-2) + 1) + 1$
$= 2(2(2T(n-3) + 1) + 1$ → $2^3T(n-3) + 4 + 2 + 1$
$= \dots$
$= 2(\dots(2(2(2T(1) + \dots + 1) + 1) + 1$ → $2^{n-1}T(1)+2^{n-2} + \dots + 4 + 2 + 1$
$= 2^{n-1} + \sum$(da k=0 a k=n-2)2^k → serie geometrica



$= 2^{n-1} + (1 - 2^{n-1})/(1 - 2)$
$= 2^{n-1} + 2^{n-1} - 1$
$= 2^n - 1$ → **$O(2^n)$** esponenziale

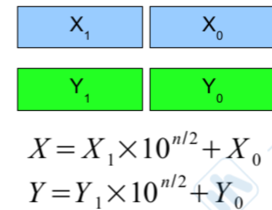
<u>Esempio</u>: moltiplicazione di interi di grandezza arbitraria

$$X = x_{n-1}x_{n-2}\dots x_1 x_0 = \sum_{i=0}^{n-1} x_i \times 10^i$$
$$Y = y_{n-1}y_{n-2}\dots y_1 y_0 = \sum_{i=0}^{n-1} y_i \times 10^i$$

Considerando due interi X e Y. Vogliamo calcolare il prodotto XY.

L'algoritmo base ha complessità n^2 (quello che moltiplica elemento per elemento i due interi). Cerchiamo un algoritmo migliore.

Soluzione 1: Divide et Impera
Divido i due interi due parti, e li moltiplico a due a due.



$$X = X_1 \times 10^{n/2} + X_0$$
$$Y = Y_1 \times 10^{n/2} + Y_0$$

$$X \times Y = (X_1 Y_1) \times 10^n + (X_1 Y_0 + X_0 Y_1) \times 10^{n/2} + X_0 Y_0$$

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

Usando il Master Theorem:
a = 4, b = 2, $\alpha = \log 4/\log 2 = 2$, $\beta = 1$
$O(n^2)$ → questo algoritmo non migliora le prestazioni

Soluzione 2: Divide et Impera

Ponendo:
$$P_1 = (X_1 + X_0) \times (Y_1 + Y_0)$$
$$P_2 = (X_1 Y_1)$$
$$P_3 = (X_0 Y_0)$$

Possiamo scrivere:
$$X \times Y = P_2 10^n + (P_1 - P_2 - P_3) \times 10^{n/2} + P_3$$

Il calcolo di P1, P2 e P3 richiede in tutto solo 3 prodotti tra numeri di n/2 cifre. Quindi:

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

Usando il Master Theorem:
a = 4, b = 2, $\alpha = \log 3/\log 2 = 1.59$, $\beta = 1$
$O(n^{1.59})$ → migliore di n^2

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

! Riuscendo a diminuire il numero di chiamate ricorsive nel divide et impera avremo un abbattimento della complessità.

Esempio: moltiplicazione di matrici
Per la moltiplicazione "riga per colonna", avremo tre cicli for annidati quindi **O(n^3)**. Un altro metodo può essere:

Suddividiamo le matrici $n \times n$ in quattro matrici $n/2 \times n/2$

$$A=\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \qquad B=\begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}$$

La matrice prodotto P risulta definita come

$$P=\begin{pmatrix} A_{1,1}B_{1,1}+A_{1,2}B_{2,1} & A_{1,1}B_{1,2}+A_{1,2}B_{2,2} \\ A_{2,1}B_{1,1}+A_{2,2}B_{2,1} & A_{2,1}B_{1,2}+A_{2,2}B_{2,2} \end{pmatrix}$$

Equazione di ricorrenza:

$$T(n)=\begin{cases} c_1 & \text{se } n \le 1 \\ 8T(n/2)+c_2 n^2 & \text{altrimenti} \end{cases}$$

Usando il Master Theorem:
a = 8, b = 2, α = log8/log2 = 3, β = 2
O(n^3) → questo algoritmo non migliora le prestazioni

Tuttavia c'è un'altra possibilità:
Calcoliamo alcuni termini intermedi

$$M_1=(A_{2,1}+A_{2,2}-A_{1,1})\times(B_{2,2}-B_{1,2}+B_{1,1})$$
$$M_2=A_{1,1}\times B_{1,1}$$
$$M_3=A_{1,2}\times B_{2,1}$$
$$M_4=(A_{1,1}-A_{2,1})\times(B_{2,2}-B_{1,1})$$
$$M_5=(A_{2,1}+A_{2,2})\times(B_{1,2}-B_{1,1})$$
$$M_6=(A_{1,2}-A_{2,1}+A_{1,1}-A_{2,2})\times B_{2,2}$$
$$M_7=A_{2,2}\times(B_{1,1}+B_{2,2}-B_{1,2}-B_{2,1})$$

Matrice finale: $$P=\begin{pmatrix} M_2+M_3 & M_1+M_2+M_5+M_6 \\ M_1+M_2+M_4-M_7 & M_1+M_2+M_4+M_5 \end{pmatrix}$$

Usando il Master Theorem:
a = 7, b = 2, α = log7/log2 = 2.81, β = 2
O($n^{2.81}$) → migliore di n^3

Esempio: sottovettore non vuoto di valore massimo
Assumiamo di partire da un vettore di numeri reali arbitrari (sia positivi che negativi). Cerchiamo una sottoparte (2 indici i e j) tale che la somma degli elementi interni sia massima (non esistono altri sottovettori con somma degli elementi interni maggiore)

3	-5	10	2	-3	1	4	-8	7	-6	-1

Algoritmo base → O(n^3)
Algoritmo migliorato → O(n^2)
Algoritmo Divide et Impera → O(nlogn)
[guardare slide ASD_2 per approfondimenti]

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Algoritmo Divide et Impera:

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

Usando il Master Theorem:
a = 2, b = 2, α = log2/log2 = 1, β = 1
O(nlogn) → migliore di n^2

Pseudocodice Esercizio 1

Consideriamo un arrai A[1…n] composto da n>=0 valori reali, non necessariamente distinti. L'array è ordinato in senso non decrescente (valori maggiori o uguali a dx). Scrivere un algoritmo divide et impera che restituisca tue se e solo se A contiene valori duplicati.

```
Duplicati(real A[1...n], int i, int j)
  if (i<= j)      return false;
  else
    int m = (i+j)/2
    return (Duplicati(A, i, m) or Duplicati(A, m+1, j) or (A[m] == A[m+1])
```

Nota: valutazione ottimizzata di espressioni booleane: in una sequenza di espressioni booleane, se il linguaggio trova un false (ad esempio) esce, senza calcolare il resto.

Calcolo costo computazionale:

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

Usando il Master Theorem:
a = 2, b = 2, α = log2/log2 = 1, β = 0
O(n)

Pseudocodice Esercizio 2

Si consideri un array A[1..n] contenente valori reali ordinati in senso non decrescente; l'array può contenere valori duplicati. Scrivere un algoritmo ricorsivo di tipo divide-et impera che, dato A e due valori reali low < up, calcola quanti valori di A appartengono all'intervallo [low, up].

```
Conta (real A[1..n], real low, real up, integer i, integer j ) → integer
if ( i > j )
  return 0;
elseif ( A[i] > up or A[j] < low )
  return 0;
```

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
elseif ( i == j )
  return 1;
else
  integer m = Floor(( i + j ) / 2);
  return Conta (A, low, up, i, m) + Conta (A, low, up, m+1, j);
```

Calcolo costo computazionale:

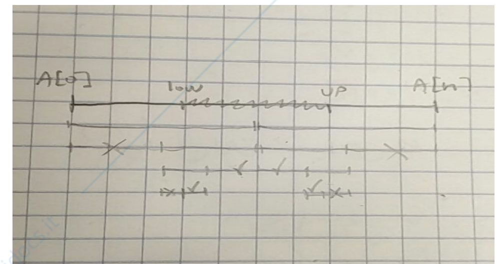
$$T(n) = \begin{cases} c_1 & \text{per } n \leq 1 \\ 2T(n/2) + c_2 & \text{per } n > 1 \end{cases}$$

Usando il Master Theorem:
$a = 2, b = 2, \alpha = \log 2/\log 2 = 1, \beta = 0$
O(n)

Si può fare di meglio?

Si, utilizzando la tecnica per (A[i] > up or A[j] < low) che evita di procedere se non si è nell'intervallo, anche per verificare se si p nell'intervallo, sostituendo (i == j) con (A[i] >= low and A[j] <= low) return (j-i+1).

```
Conta (real A[1..n], real low, real up, integer i, integer j ) → integer
if ( i > j )
  return 0;
elseif ( A[i] > up or A[j] < low )
  return 0;
elseif ( A[i] >= low and A[j] <= low )
  return (j-i+1);
else
  integer m = Floor(( i + j ) / 2);
  return Conta (A, low, up, i, m) + Conta (A, low, up, m+1, j);
```



Costo computazionale → **O(logn)**

Teniche algoritmiche: Greedy

Tecnica algoritmica che vuole risolvere problemi di ottimizzazione. Utilizzando il fatto che, in alcuni casi, c'è una tecnica che ci permette di seguire una strada (ritenuta ottima) e lasciando indietro le alternative. Grosso modo si segue questo schema:
Si ha un insieme di possibili candidati (scelte effettuabili per arrivare alla soluzione). Se ne seleziona uno e si controlla se è una buona

```
Greedy(insieme di candidati C) → soluzione
   S ← Ø
   while ( (not ottimo(S)) and (C ≠ Ø) ) do
      x ← seleziona(C)
      C ← C - {x}
      if (ammissibile(S U {x})) then
         S ← S U {x}
      endif
   endwhile
   if (ottimo(S)) then
      return S
   else
      errore ottimo non trovato
   endif
```

Ritorna *true* sse la soluzione S è ottima

Estrae un candidato dall'insieme C

Ritorna *true* sse la soluzione candidata è una soluzione ammissibile (anche se non necessariamente ottima)

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

soluzione o no, e ripeto ciclicamente per tutti i candidati ed esco se non ci sono soluzioni o ho trovato la soluzione ottima.

<u>Esempio</u>: Problema del resto
Abbiamo un numero intero R che rappresenta l'importo di centesimi da dare come resto. Dobbiamo restituire il numero minimo (intero) di monete necessarie per erogare il resto R.

Scelta greedy: prendo la moneta più grande di valore inferiore ad R, ripeto fino alla fine.
R = 78 → 50+20+5+2+1

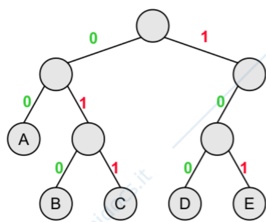
<u>Esempio</u>: codici di Huffman

Si vuole comprimere una serie di caratteri attribuendo a ogni valore binario di ogni carattere un valore binario più piccolo. Si utilizzerà un codice "a prefisso": nessun codice è prefisso di un altro codice. (Es. NON andrebbe bene f(a) = 1, f(b)=10, f(c)=101, se no come decodificheremmo 101? come "c" o "ba"?)

'a'	'b'	'c'	'd'	'e'	'f'
0	101	100	111	1101	1100

addaabca → 0 111 111 0 0 101 100 0

Rappresentazione ad Albero del codice



A: 00
B: 010
C: 011
D: 100
E: 101

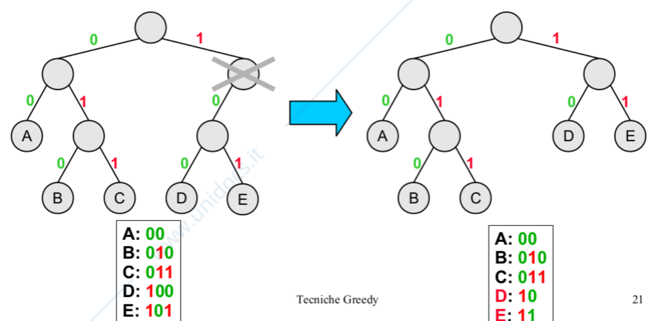
Il figlio sinistro → 0
Il figlio destro → 1

La codifica dei caratteri sono nelle foglie dell'albero

L'algoritmo di *decodifica* è intuitivo:

1. parti dalla radice
2. leggi un bit alla volta percorrendo l'albero:
   0: sinistra
   1: destra
3. stampa il carattere della foglia
4. torna a 1

Ottimizzazione dell'albero: non c'è motivo di avere un nodo intero con un solo figlio



www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

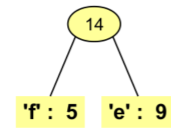
Il principio del codice di Huffman è di minimizzare la lunghezza dei caratteri che compaiono più frequentemente e assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero.

Passo 1: Costruire una lista ordinata di nodi, in cui ogni nodo contiene un carattere e il numero di volte in cui quel carattere compare nel file.

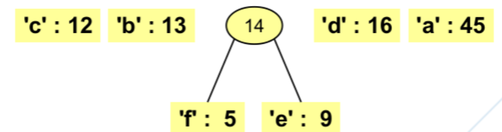
'f' : 5 'e' : 9 'c' : 12 'b' : 13 'd' : 16 'a' : 45

Passo 2: Rimuovere i due nodi con frequenze minori.

'c' : 12 'b' : 13 'd' : 16 'a' : 45

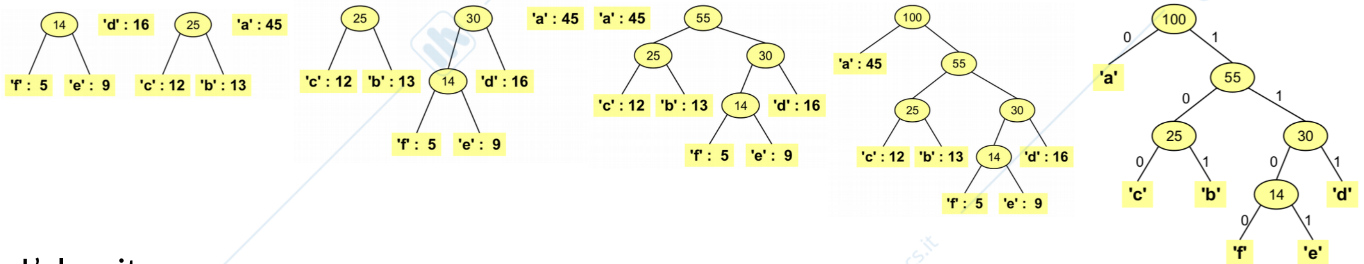


Passo 3: Collegarli ad un nodo padre etichettato con la frequenza combinata (sommata).



Passo 4: Aggiungere il nodo combinato alla lista, mantenendola ordinata in base alle frequenze.

Passo 5: Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista. Al termine si etichettano gli archi dell'albero con 0 o 1.



L'algoritmo:

```
Huffman(real f[1..n], char c[1..n]) → Tree
  Q ← new MinPriorityQueue()
  integer i;
  for i ← 1 to n do
    z ← new TreeNode(f[i], c[i]);
    Q.insert(f[i], z);
  for i ← 1 to n – 1 do
    z1 ← Q.findMin(); Q.deleteMin();
    z2 ← Q.findMin(); Q.deleteMin();
    z ← new TreeNode(z1.f + z2.f, '');
    z.left ← z1;
    z.right ← z2;
    Q.insert(z1.f + z2.f, z);
  return Q.findMin();
```

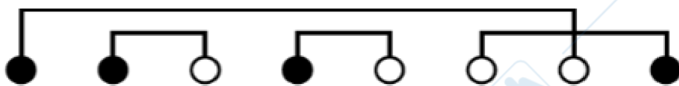
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Esercizio:
Un'auto può percorrere K Km con un litro di carburante, e il serbatoio ha una capacità di C litri. Tale auto deve percorrere un tragitto lungo il quale si trovano n + 1 aree di sosta indicate con 0, 1, ... n, con n ≥ 1. L'area di sosta 0 si trova all'inizio della strada, mentre l'area di sosta n si trova alla fine. Indichiamo con d[i] la distanza in Km tra le aree di sosta i e i + 1. Nelle n - 2 aree di sosta intermedie {1, 2, ... n - 1} si trovano delle stazioni di servizio nelle quali è possibile fare il pieno (vedi figura). Tutte le distanze e i valori di K e C sono numeri reali positivi. La auto parte dall'area 0 con il serbatoio pieno, e si sposta lungo la strada in direzione dell'area n senza mai tornare indietro. Progettare un algoritmo in grado di calcolare il numero minimo di fermate che sono necessarie per fare il pieno e raggiungere l'area di servizio n senza restare a secco per strada, se ciò è possibile. Nel caso in cui la destinazione non sia in alcun modo raggiungibile senza restare senza carburante, l'algoritmo restituisce -1.

```
MinFermate(real d[0,…,n-1], real K, real C) → int
  real res ← K * C
  int i ← 0, f ← 0
  while (i < n) do
    if (res < d[i]) then
      res ← K*C
      f ← f + 1
  res ← res – d[i]
  if (res < 0) then // una distanza è superiore alla capacità massima
    return -1
```

Esercizio:
Lungo una linea, a distanze costanti (che per comodità indichiamo con distanza 1), sono presenti n punti neri ed n punti bianchi. È necessario collegare ogni punto nero ad un corrispondente punto bianco tramite fili; ad ogni punto deve essere collegato uno ed un solo filo. Scrivere un algoritmo efficiente per determinare la quantità minima di filo necessaria. Determinare il costo computazionale dell'algoritmo proposto. A titolo di esempio, nell'immagine sotto viene riportata una istanza del problema con 4 punti neri e 4 punti bianchi, ed un corrispondente collegamento di punti che richiede l'uso di una lunghezza complessiva di filo pari a 10.



```
CollegaPunti(bool p[1...2n])
  int i, filo ← 0
  queue bianchi ← new queue()
  queue neri ← new queue()
  for i ← 1 to 2n
    if (p[i]) then
      if (neri.empty()) then bianchi.enqueue(i)
      else filo ← filo + (i – neri.dequeue())
```

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

```
    else
      if (bianchi.empty()) then neri.enqueue(i)
      else filo ← filo + (i – bianchi.dequeue())
  return filo
```

Teniche algoritmiche: Programmazione Dinamica

Tecnica iterativa con un approccio bottom-up, molto vantaggiosa quando ci sono sottoproblemi ripetuti. Di base si utilizza una struttura dati per registrare le soluzioni parziali (sotto-problemi costruiti partendo dal basso).

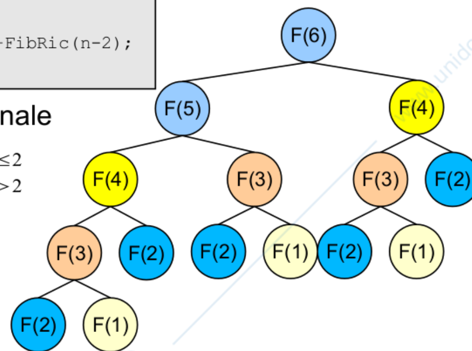
<u>Esempio</u>: successione di Fibonacci ricorsiva

```
integer FibRic(integer n)
  if ((n = 1) or (n = 2)) then
    return 1;
  else
    return FibRic(n-1)+FibRic(n-2);
  endif
```

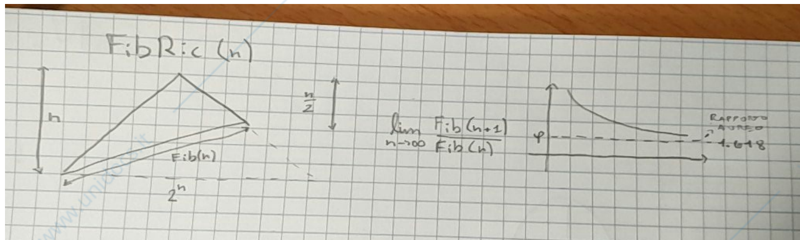
- Costo computazionale

$$T(n)=\begin{cases} c_1 & n\le 2 \\ T(n-1)+T(n-2)+c_2 & n>2 \end{cases}$$

- Soluzione
  - $T(n) = O(2^n)$



φ = 1.618 (sezione aurea)
φ = a / b con a, b tali che a/b = (a+b) / a
Ciò significa che Fib(n+1) / Fib(n) e Fib(n+2) / Fib(n+1) per valori grandi tendono a coincidere.



Quindi il Fib(n) che indica la complessità del nostro albero "tagliato" varrà $O(1.618...^n)$ ovvero il valore della sezione aurea elevato alla n, che è più piccolo di $O(2^n)$

Lo spazio in memoria dell'albero sopra sarebbe O(n) nonché l'altezza dell'albero, questo è ciò che viene occupato nello stack ricorsivo.

NOTA: si nota che in realtà parti dell'albero sono ripetute (con la tecnica divide et impera)

Con la tecnica di programmazione dinamica si parte dal basso: trovo una soluzione per F(1), poi F(2), con queste due creo F(3), con F(3) ed F(2) creo F(4) e così via, fino a raggiungere il problema iniziale F(6). In questo modo la complessità diventa O(n) sia di tempo che memoria (da esponenziale → a lineare).

n	1	2	3	4	5	6	7	8
f []	1	1	2	3	5	8	13	21

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

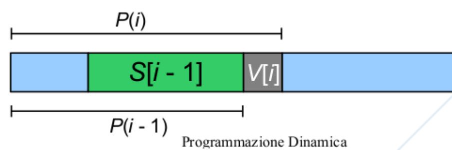
Ancora meglio: non serve lo storico di tutti i Fibonacci passati, sono sufficienti 3 campi che si alternano fino alla soluzione finale. E ciò ci abbasserebbe la complessità spaziale a O(1).

n	3	4	5	6
f[1]	1	1	2	3
f[2]	1	2	3	5
f[3]	2	3	5	8

<u>Esempio</u>: sotto-vettore di valore massimo
Soluzione basata su programmazione dinamica:
Sia P(i) il problema che consiste nel determinare il valore massimo della somma degli elementi dei sottovettori non vuoti del vettore V[1…i] che hanno V[i] come ultimo elemento.

P(1) ammette una unica soluzione
- *S*[1] = *V*[1]

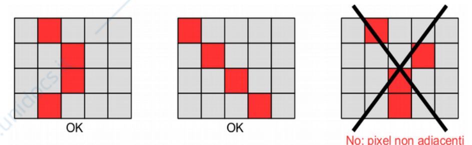
Consideriamo il generico problema *P*(*i*), *i* > 1
- Supponiamo di avere già risolto il problema *P*(*i* - 1), e quindi di conoscere *S*[*i* - 1]
- Se *S*[*i* - 1]+*V*[*i*] ≥ *V*[*i*] allora *S*[*i*] = *S*[*i* - 1]+*V*[*i*]
- Se *S*[*i* - 1]+*V*[*i*] < V[*i*] allora *S*[*i*] = *V*[*i*]



Altro <u>esempio</u> molto interessante: problema dello zaino [slides da 22 → ASD_4.pdf]

<u>Esempio</u>: Seam Carving

Algoritmo per ridimensionare immagini in modo "intelligente". L'immagine viene ridimensionata togliendo cuciture. Una cucitura (seam) è un cammino composto da pixel adiacenti di "minima importanza" (se l'immagine ha M righe per N colonne, una cucitura (verticale) è una sequenza di M pixel adiacenti, uno per ogni riga).



0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

0.1	0.0	0.2	0.9	0.8
0.9	0.2	0.8	0.4	0.7
0.8	0.8	0.1	0.7	0.8
0.1	0.0	0.6	0.5	0.7

0.1	0.2	0.9	0.8
0.9	0.8	0.4	0.7
0.8	0.8	0.7	0.8
0.1	0.6	0.5	0.7

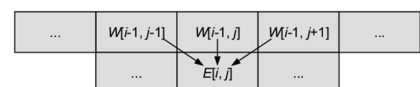
Si assegna ad ogni pixel (i, j) un peso E[i, j] – [da 0 a 1] che denota quanto il pixel è "importante" (ad esempio quanto un pixel è "diverso" da quelli adiacenti → 0 = non importante, 1 = molto importante).

- Determinare una cucitura verticale (o orizzontale se si vuole comprimere i lati alto e basso) di <u>peso minimo</u> (la somma dei valori del percorso è minimale, la più piccola).
- <u>Rimuovere i pixel</u> della cucitura, ottenendo una immagine M x (N – 1).
- <u>Ripetere</u> il procedimento fino ad ottenere la larghezza desiderata.

Per determinare le cuciture di peso minimo: (prog dinam)
Definizione dei sotto-problemi P(i, j):
- Determinare una cucitura di peso minimo che termina nel pixel di coordinate (i, j)

Casi base (*i* = 1):
- *W*[1, *j*] = *E*[1, *j*] per ogni *j* = 1, … *M*

Caso generale (*i* > 1):
- Se *j* = 1
  W[*i*, *j*] = *E*[*i*, *j*] + min { *W*[*i* - 1, *j*], W[*i* - 1, *j* + 1] }
- Se 1 < *j* < *N*
  W[*i*, *j*] = E[*i*, *j*] + min { *W*[*i* - 1, *j* - 1], *W*[*i* - 1, *j*], *W*[*i* - 1, *j* + 1] }
- Se *j* = *N*
  W[i, j] = *E*[*i*, *j*] + min { *W*[*i* - 1, *j* - 1], *W*[*i* - 1, *j*] }

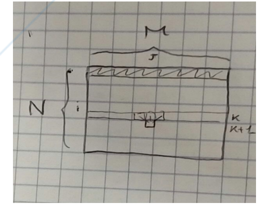


www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Definizione delle soluzioni W[i, j]:
- Minimo peso tra tutte le possibili cuciture che terminano nel pixel di coordinate (i, j)

Calcolo della soluzione del problema originario:
- La cucitura di peso minimo avrà peso pari al minimo tra { W[M, 1], ... W[M, N] }



Esempio: Distanza di Levenshtein
- "edit distance": Numero di operazioni di "editing" che sono necessarie per trasformare una stringa S in una nuova stringa T. Si inizia dal primo carattere di S fino alla fine.

Trasformazioni ammesse:
→ Lasciare immutato il carattere corrente (costo 0)
→ Cancellare un carattere (costo 1)
→ Inserire un carattere (costo 1)
→ Sostituire il carattere corrente con uno diverso (costo 1)

Esempio di trasformazione ALBERO → LIBRO
(posso togliere tutti i caratteri e rimetterli tutti, o tenerne alcuni)

Costo totale: 6 cancellazioni + 5 inserimenti = 11

ALBERO	→	LBERO	cancellazione A
LBERO	→	BERO	cancellazione L
BERO	→	ERO	cancellazione B
ERO	→	RO	cancellazione E
RO	→	O	cancellazione R
O	→	_	cancellazione O
_	→	L_	inserimento L
L_	→	LI_	inserimento I
LI_	→	LIB_	inserimento B
LIB_	→	LIBR_	inserimento R
LIBR_	→	LIBRO_	inserimento O

2 cancellazioni + 1 inserimento = 3

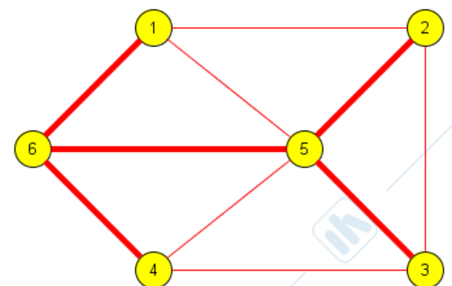
ALBERO	→	LBERO	cancello A
LBERO	→	LBERO	lascio immutato
LBERO	→	LIBERO	inserisco I
LIBERO	→	LIBERO	lascio immutato
LIBERO	→	LIBRO	cancello E

Guardare ultime slide di ASD_4

Algoritmi sui Grafi

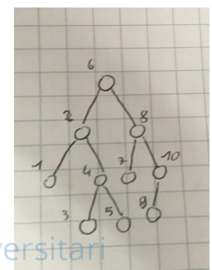
Orientati / Non orientati – Pesati / Non pesati
Implementazione con matrice di adiacenza
Implementazione con liste di adiacenza

Alberi di copertura: un albero che contiene tutti i vertici del grafo, ma degli archi ne contiene soltanto un sottoinsieme, cioè solo quelli necessari per connettere tra loro tutti i vertici con uno e un solo cammino.



RIPASSO: visita di un albero
Pre-Order: visito un nodo, lo stampo ed effettuo ricorsivamente la stessa cosa sui suoi sottoalberi.
[la radice prima dei propri figli]

Post-Order: visito un nodo, effettuo ricorsivamente la stessa cosa sui suoi sottoalberi e stampo il nodo. [i figli prima della radice]

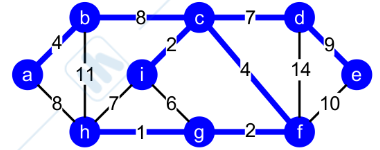


www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

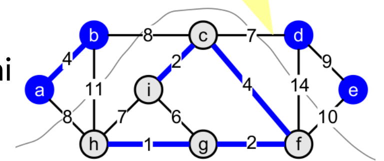
www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

<u>In-Order</u>: (per alberi binari) visito il figlio di sinistra, la radice e poi il figlio di destra, ricorsivamente. (In questo modo se si hanno degli alberi binari di ricerca la stampa sarà in ordine crescente)

Minimum Spanning Tree



Determinare un albero di copertura di un grafo che abbia il peso degli archi il minore possibile.
Utilizzato ad esempio per determinare come interconnettere diversi elementi fra loro minimizzando certi vincoli sulle connessioni (es. progettazione dei circuiti elettronici dove si vuole minimizzare la quantità di filo elettrico per collegare fra loro i diversi componenti).



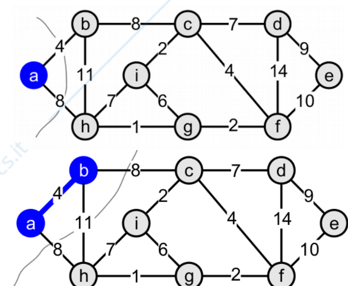
Definizioni:
→ **Taglio** (S, V – S) di un grafo è una partizione di V in due sottoinsiemi disgiunti.
Un arco {u, v} attraversa il taglio se u ∈ S e v ∈ V – S.
Un arco che attraversa un taglio è leggero se il suo peso è minimo fra i pesi degli archi che attraversano un taglio.
! Sarà necessario prendere uno degli archi tagliati (per unire due alberi disgiunti), prenderò quindi quello con peso minore.
! Se trovo un ciclo all'interno di un grafo, posso colorare di **rosso** l'arco (del ciclo) con peso maggiore, per spezzare tale ciclo.

Algoritmo di Kruskal
Si parte più che da un albero, da una foresta. All'inizio tale foresta contiene tutti singoli nodi (alberi con 1 solo elemento).
Ingrandire sottoinsiemi disgiunti di un albero di copertura minimo connettendoli fra di loro fino ad avere l'albero finale. Si considerano gli archi in ordine non decrescente di peso
Inizialmente la foresta di copertura è composta da n alberi, uno per ciascun nodo, e nessun arco
Se l'arco e = {u, v} connette due alberi blu distinti, lo si colora di blu. Altrimenti lo si colora di rosso. L'algoritmo è greedy perché ad ogni passo si aggiunge alla foresta un arco con il peso minimo.

Ogni volta che prendo un arco, valuto se questo unisce già due alberi della stessa foresta, in tal caso creerei un ciclo (lo devo colorare quindi di rosso), in caso contrario di blu, ed entra a far parte dell'albero. Si parte dagli archi con valori minori, crescendo via via.

La struttura dati ottimale per questa operazione è una struttura **merge-find**.
Complessità **O(m log n)**

Algoritmo di Prim
Si parte da un punto di partenza (un certo nodo), qui si fa un taglio tra quello che fa parte dell'albero e quello che non ne fa parte.
Prendo ora l'arco tra quelli tagliati con peso inferiore, ora l'albero inizialmente composto da un solo nodo si amplia di un nodo, e così via.



www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

La struttura dati ottimale per questa operazione è la **coda con priorità**. (Viene utilizzata per tener traccia di come evolve il taglio nel tempo, la priorità non è altro che il peso di ogni arco, più è basso e più la priorità è alta) → NOTA: consigliabile utilizzare un min-heap come coda con priorità.

Cammini di costo minimo

Consideriamo un grafo orientato $G = (V, E)$ in cui ad ogni arco $(x, y) \in E$ sia associato un costo $w(x, y)$
Il costo di un cammino $\pi = (v_0, v_1, \dots v_k)$ che collega il nodo v_0 con v_k è definito come

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

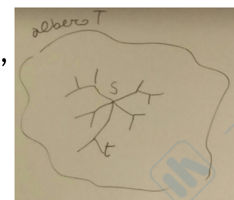
Data una coppia di nodi v_0 e v_k, vogliamo trovare (se esiste) il cammino $\pi_{v0\,vk}*$ di costo minimo tra tutti i cammini che vanno da v_0 a v_k

Problema: dato un grafo orientato pesato, e un nodo **s**, vogliamo trovare per ogni nodo **t** raggiungibile da **s**, un cammino di costo minimo da **s** a **t**.

Come algoritmi di soluzione del problema abbiamo:
- **Bellman-Ford** (presuppone l'assenza di cicli con peso complessivo negativo)
- **Dijkstra** (presuppone che tutti i pesi siano non negativi) [più efficiente]

Proprietà rilevanti:
1) Se ci sono cicli di peso complessivo negativo → non è possibile trovare i cammini minimi (non è proprio presente il concetto di cammino minimo).
[Perchè ad ogni esecuzione del ciclo, riduco il peso complessivo del cammino, quindi ho infiniti cammini diversi in cui ognuno ne ha un successivo ancora più piccolo, è quindi impossibile trovare il minimo]

2) Se non ci sono cicli di peso complessivo negativo:
→ per ogni noto **t** raggiungibile da **s**, esiste un cammino minimo semplice (= non contiene cicli)
[Se avessimo un cammino minimo **s**-**t** di un certo peso con vari cicli, potremmo eliminare tali cicli ed avere un cammino con lo stesso peso]
→ esiste un albero T, radicato in S che contiene tutti i nodi raggiungibili da S, tale che ogni cammino in T risulta essere un cammino di costo minimo.



3) Sotto-struttura ottima → dato un cammino di costo minimo da x a y che attraversa i nodi i e j, avremo che il sotto-cammino da i a j sarà anch'esso di costo minimo.

Algoritmo di Bellman-Ford

Parto da un nodo particolare, imposto la distanza con tutti i nodi pari a infinito.
Imposto la distanza con se stesso a 0. Poi inizio a calcolare il costo di collegamento con gli archi a distanza 1 dal nodo di partenza. Procederò successivamente dai nodi con distanza 1 (a cui ad ognuno assegno il peso) a calcolare i loro nodi adiacenti (a distanza 1) e impostare in essi il peso opportuno (sommando il loro a quello per raggiungerli), nel caso sostituire il peso

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

che c'è già (se un altro arco l'ha già assegnato) se quello che assegniamo noi è inferiore. A ogni esecuzione andiamo però anche a rivalutare i nodi precedenti per controllare se abbiamo un percorso migliore da offrire. Così procedendo creo un albero di cammini minimi.

Dopo n-1 cicli abbiamo la garanzia che se non ci sono cicli negativi abbiamo trovato il nostro albero di cammini minimi. N-1 perché nel caso pessimo l'albero sarà una sorta di lista con ogni nodo un figlio, lungo n (numero nodi nel grafo) – 1 (il nodo di partenza).

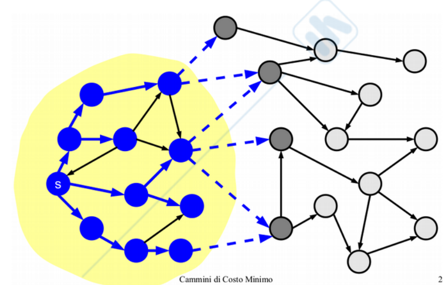
Nel caso in cui all'n-esimo ciclo trovassimo una distanza inferiore, siamo in presenza di un ciclo negativo, in caso contrario l'albero è corretto. Totale n-1 cicli + n-esimo ciclo di conferma.

Costo: **O(nm)**

Algoritmo di Dijkstra

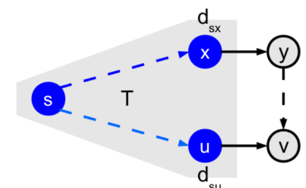
Algoritmo greedy in cui una volta battezzato un arco come valido, tale scelta non verrà più modificata. NOTA: non sono presenti archi con peso negativo. Parto da un nodo e scelgo il nodo con peso inferiore che punta a un nodo di distanza 1. In generale si prende un nodo, si guardano tutti gli archi che puntano a lui e si sceglie il minore.

Lemma di Dijkstra

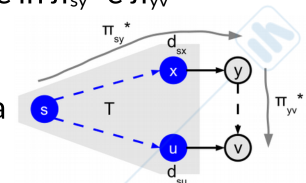


- Sia G = (V, E) un grafo orientato con funzione costo w. (I costi degli archi devono essere ≥ 0)
- Sia T una parte dell'albero dei cammini di costo minimo radicato in s (T rappresenta porzioni di cammini di costo minimo che partono da s)
- Allora l'arco (u, v) con u ∈ V(T) e v ∉ V(T) che minimizza la quantità $d_{su} + w(u, v)$ appartiene ad un cammino minimo da s a v

Dimostrazione



- Supponiamo per assurdo che (u,v) non appartenga ad un cammino di costo minimo tra s e v (quindi $d_{su} + w(u,v) > d_{sv}$)
- Quindi deve esistere π_{sv}^* che porta da s in v senza passare per (u,v) con costo inferiore a $d_{su} + w(u,v)$
- Per il teorema di sotto-struttura ottima, il cammino π_{sv}^* si scompone in π_{sy}^* e π_{yv}^*
- Quindi $d_{sv} = d_{sx} + w(x,y) + d_{yv}$
- Per ipotesi (lemma di Dijkstra), l'arco (u,v) è quello che, tra tutti gli archi che collegano un vertice in T con uno non ancora in T, minimizza la somma d su + w(u,v)
- In particolare: $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$



www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Riassumendo

- Da (1) abbiamo $d_{su} + w(u,v) > d_{sv}$
- Da (2) abbiamo $d_{sv} = d_{sx} + w(x,y) + d_{yv}$
- Da (3) abbiamo $d_{su} + w(u,v) \le d_{sx} + w(x,y)$
- Combinando (1) (2) e (3) otteniamo

$$\begin{aligned} d_{su}+w(u,v) &> d_{sx}+w(x,y)+d_{yv} && \text{da (1) e (2)} \\ &\ge d_{sx}+w(x,y) \\ &\ge d_{su}+w(u,v) && \text{da (3)} \end{aligned}$$

Assurdo!

Analisi algoritmo di Dijkstra
- L'inizializzazione ha costo O(n)
- Le operazioni find() e deleteMin() hanno costo **O(log n)** e sono eseguite al più **n** volte (una volta che un nodo è stato estratto dalla coda di priorità non verrà più reinserito)
- Le operazioni insert() e decreaseKey() hanno costo **O(log n)** e sono eseguite al più **m** volte (una volta per ogni arco)

Totale: **O((n+m) log n) = O(m log n)** se tutti i nodi sono raggiungibili dalla sorgente

! Per trovare l'opposto del minimum spanning tree (ovvero il maximum spanning tree) è banalmente possibile invertire i segni dei pesi, e poi calcolare i minimum spanning tree di Dijkstra come sempre per determinare quello che è in realtà il maximum.

Domanda:
Se ho un grafo con anche pesi negativi, cambio i pesi aggiungendo la quantità necessaria perché il peso minimo sia comunque maggiore uguale a 0, se applico Dijkstra su questo nuovo grafo trovo il MST?
No, perché ad ogni peso ho aggiunto un certo valore C, l+3C (supponendo l il costo originario del primo percorso e 3C le quantità di C aggiunte per essere >= 0) e l'+5C (supponendo l' il costo originario del secondo percorso e 5C le quantità di C aggiunte per essere >= 0) indica che il primo è il migliore la se inizialmente l > l' avremmo un problema.

NOTA: quindi se ci sono degli archi negativi → Belman-Ford (senza stravolgere il grafo per adattarlo a Dijkstra)

Trovare il cammino minimo tra due nodi dati
Banalmente applicare dijkstra su tutti i nodi, ma avrebbe costo O(nmlogn)

Algoritmo di Floyd-Warshall
d^k_{xy} : distanza minima dal nodo x al nodo y, nell'ipotesi in cui gli eventuali nodi intermedi possano appartenere esclusivamente all'insieme {1,...k}. Questa è la soluzione al nostro problema.

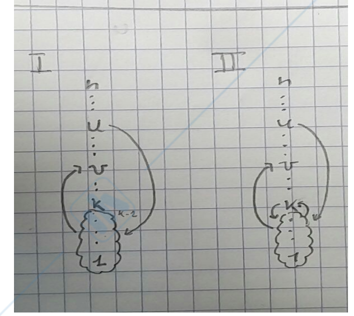
Indicizzo i nodi, da 1 a n, d^k_{uv} è la lunghezza minima di un cammino che parte da u, attraversa nodi da 1 a k e termina in v. **u,v,k** sono nodi compresi tra 1 e n. L'algoritmo calcola d^0_{uv}, facile da calcolare, poi d^1_{uv}, e così via fino a d^n_{uv} che sarà la nostra soluzione finale.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

Assumiamo di conoscere d^{k-1}_{xy} per ogni coppia x, y. Troveremo come calcolare d^{k}_{uv} tramite un'espressione che conterrà solo dei d^{k-1}_{uv}. Osservazione: il cammino <u>ottimale</u> (da u a v rimanendo nel sotto grafo [1,k]), ha due possibilità:

1. tocca anche il nodo k
2. non tocca il nodo k



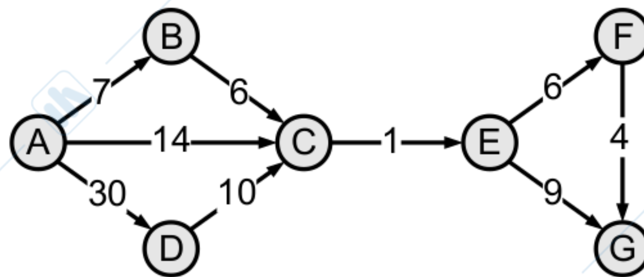
Caso 1.
Il cammino migliore entrerà nella zona 1, k-1 quindi→ $d^{k}_{uv} = d^{k-1}_{uv}$

Caso 2.
Se c'è un cammino minimo, è anche semplice (non ha cicli), quindi si partirà da u, se ha nodi intermedi saranno dentro 1,k, poi toccherà k e se da k a v ci sono altri nodi, saranno sempre nella zona 1,k quindi →$d^{k}_{uv} = d^{k-1}_{uk} + d^{k-1}_{kv}$

In generale quindi sarà semplicemente necessario calcolare il minimo tra il caso 1 e il caso 2
$d^{k}_{uv} = \min\{ d^{k-1}_{uv}, d^{k-1}_{uk} + d^{k-1}_{kv} \}$

Funziona anche se ci sono archi di peso negativo, al termine dell'algoritmo se D[x, x, n] < 0 per un qualche nodo x, allora tale nodo fa parte di un ciclo negativo.

Costo $O(n^3)$ sia per tempo che per spazio.



d^{0}_{xy} (= d^{1}_{xy})

	A	B	C	D	E	F	G
A	0	7	14	30	∞	∞	∞∞∞
B	∞	0	6	∞	∞	∞	∞
C	∞	∞	0	∞	1	∞	∞
D	∞	∞	10	0	∞	∞	∞
E	∞	∞	∞	∞	0	6	9
F	∞	∞	∞	∞	∞	0	4
G	∞	∞	∞	∞	∞	∞	0

d^{n}_{xy}

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	∞	0	6	∞	7	13	16
C	∞	∞	0	∞	1	7	10

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari

D	∞	∞	10	0	11	17	20
E	∞	∞	∞	∞	0	6	9
F	∞	∞	∞	∞	∞	0	4
G	∞	∞	∞	∞	∞	∞	0

Riassunto:
(sotto)Problemi di programmazione dinamica Floyd-Warshall
d^k_{xy}
- Casi base: d^0_{xy} (con k=0)
- Casi induttivi: (conosco già d^{k-1}_{xy})

$$d^k_{xy} = \min\{d^{k-1}_{xy}, d^{k-1}_{xk} + d^{k-1}_{ky}\}$$

Ottimizzazione algoritmo
Quando arriviamo alla k-esima fase dell'algoritmo d^k_{xy} alcuni vecchi elementi non è più necessario mantenerli in memoria. Arrivati alla k-esima fase gli unici valori delle tabelle utilizzati sono D[x,y,k-1], D[x,k,k-1], D[k,y,k-1],
Un'altra intuizione è che nel momento che calcoliamo un nuovo D[x,y,k] possiamo andarlo a sostituire a D[x,y,k-1] in quanto non verrà più utilizzato (questo non sempre però, in alcune situazioni particolari, per calcolare dei nuovi valori per coppie di nodi in cui uno dei due nodi è uguale a k, quindi il d^{k-1}_{xy} può tornare utile solo per calcolare dei nuovi d^{k-1}_{zk} oppure d^{k-1}_{kz} ovvero dove appare k in arrivo o partenza).
Quando uno dei due indici D[k,y,k] o D[x,k,k] allora uno dei due valori è uguale al vecchio.

```
double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n];
    int x, y, k, next[1..n, 1..n];
    for x ← 1 to n do
        for y ← 1 to n do
            if (x == y) then
                D[x,y] ← 0;
                next[x,y] ← -1;
            elseif ((x,y) ∈ ±) then
                D[x,y] ← w(x,y);
                next[x,y] ← y;
            else
                D[x,y] ← +∞;
                next[x,y] ← -1;
            endif
        endfor
    endfor
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                if (D[x,k] + D[k,y] < D[x,y]) then
                    D[x,y] ← D[x,k] + D[k,y];
                    next[x,y] ← next[x,k];
                endif
            endfor
        endfor
    endfor
    return D;
```

Quindi, è implementabile una versione più efficiente utilizzando una tabella bidimensionale, invece che tridimensionale.
Il next è un'altra struttura supplementare per mantenere il percorso del cammino minimo, come sappiamo solo con la programmazione dinamica si ha solo il risultato finale ma non come ci si arriva.

www.unidocs.it - Appunti e dispense per superare i tuoi esami universitari