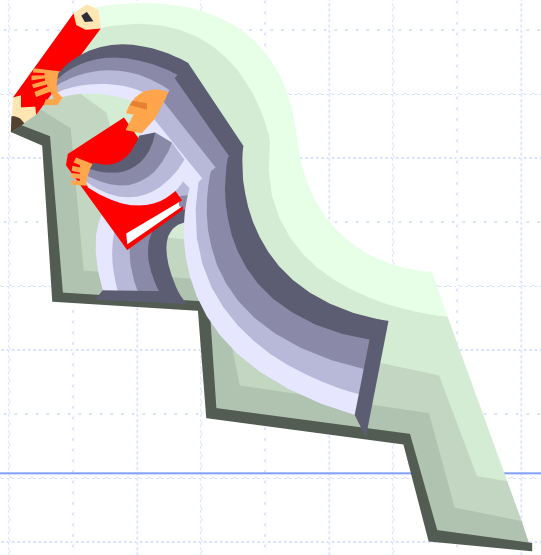
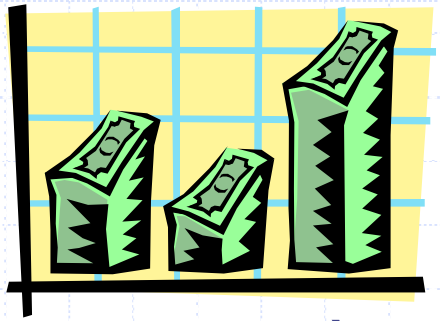
Il Metodo Greedy e Compressione di Testi



Il Metodo Greedy



- **Il metodo Greedy (greedy = ghiotto, ingordo)** è un metodo generale di progetto di algoritmi basato sui seguenti elementi:
  - **configurazioni**: diverse scelte, elementi o valori da cercare
  - **Funzione obiettivo**: un valore associato a ciascuna configurazione; vogliamo trovare a seconda dei casi la configurazione che massimizza o minimizza il valore
- La tecnica di progettazione di algoritmi Greedy, si basa sulla semplice strategia dell'ingordo, compiere, ad ogni passo, la scelta migliore nell'immediato piuttosto che adottare una strategia a lungo termine.
- Il metodo permette di avere algoritmi semplici da programmare; non sempre fornisce la soluzione ottima

Compressione di Testi

- Data una stringa X codifica X in modo efficiente (cioè veloce) in una stringa Y più corta
  - Si risparmia memoria e/o banda trasmissione
- Un buon approccio (greedy): **Codifica di Huffman**
  - Calcola la frequenza f(c) per ogni carattere c.
  - Codifica caratteri molto frequenti con parole di codice brevi
  - La correttezza richiede che nessuna parola di codice sia un prefisso di altra parola di codice
  - Usa un albero pe rdeterminare le parole codice per ciascun carattere

Compressione di Testi

Esempio alfabeto di 6 caratteri:

codice a lunghezza fissa: 3 bit per rappresentare i caratteri. Quindi file di 100K caratteri richiede 300K bit.

Codice a lunghezza variabile: si codificano i simboli che occorrono più di frequente con meno bit quelli meno frequenti con più bit

Supponiamo che la frequenza sia la seguente :

Caratteri	a	b	c	d	e	f
frequenza	45.000	13.000	12.000	16.000	9.000	5.000
codice l. fissa	000	001	010	011	100	101
codice l. var.	0	101	100	111	1101	1100

Facendo i conti il codice a lunghezza variabile permette un risparmio del 25%.

Codici prefissi

- Un **codice** è un mapping di ogni carattere di un alfabeto ad una sequenza di parole-codici binarie
- Un **codice prefisso** è un codice binario tale che nessuna parola-codice è prefisso di un'altra parola codice
- Non usare codici prefissi crea problemi di ambiguità.
- Consideriamo la codifica di tabella

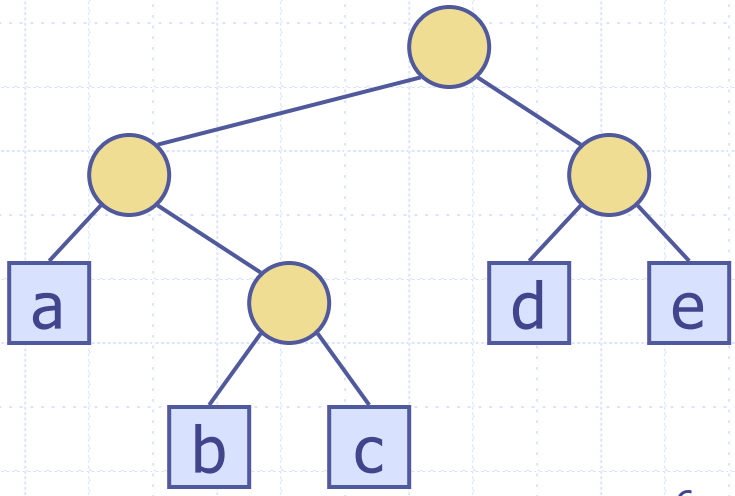
00	01	10	11	000
a	b	c	d	e

- Utilizzando la tabella per codificare la stringa aaa otteniamo 000000
- Ma quando decodifichiamo la sequenza 000000 non sappiamo decidere se essa corrisponda a 'aa' oppure a 'ee'?
- Usare codici prefissi elimina questo problema

Codici prefissi, Albero di codifica

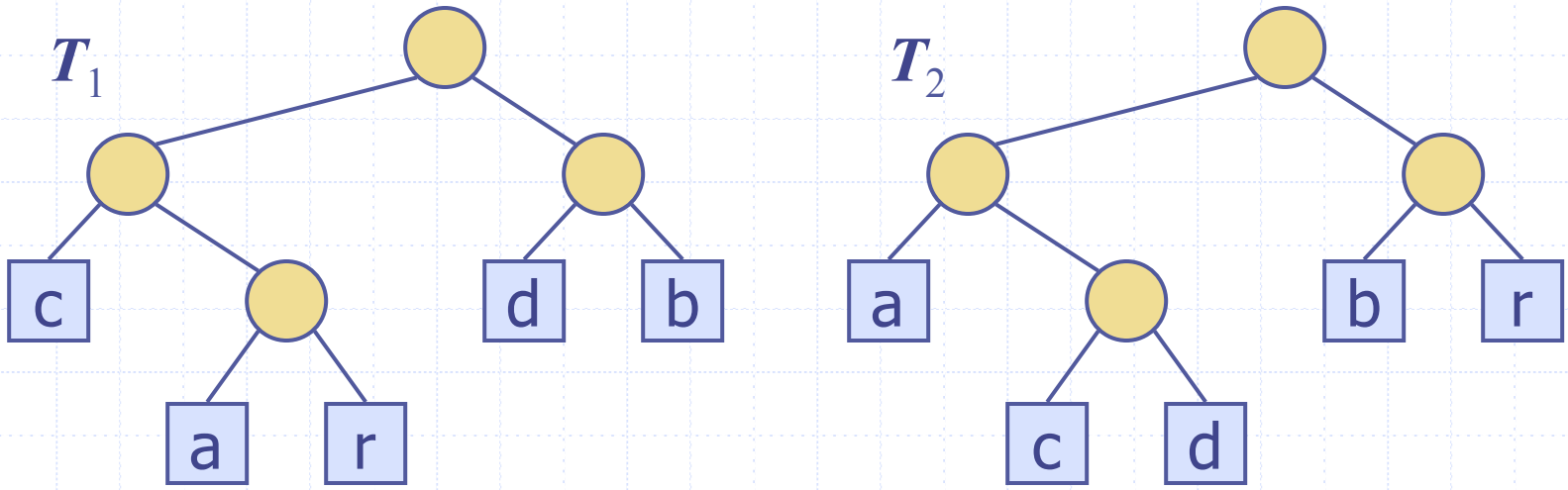
- Un **codice** è un mapping di ogni carattere di un alfabeto ad una sequenza di parole-codici binarie
- Un **codice prefisso** è un codice binario tale che nessuna parola-codice è prefisso di un'altra parola codice
- Un **albero di codifica** rappresenta un codice prefisso
  - Ogni fogla memorizza un carattere
  - La parola codice di un carattere è data dal cammino dalla radice al nodo che memorizza il carattere (0 per il figlio sinistro 1 per il figlio destro)

00	010	011	10	11
a	b	c	d	e



Albero di codifica: Ottimizzazione

- Data una stringa di testo X, vogliamo trovare un codice prefisso per I caratteri di X che fornisce una codifica efficiente di X
  - Caratteri frequenti dovrebbero avere parole codici corti
  - Caratteri rari dovrebbero avere parole codici lunghe
- Esempio
  - X = abracadabra
  - T_1 codifica X in 29 bits, T_2 codifica X in 24 bits
  - Nota: a - carattere più frequente – in T_1 è codificato con 010, in T_2 con 00



Algoritmo di Huffman

- Data una stringa X, l'algoritmo di Huffman costruisce un codice prefisso che minimizza la dimensione della codifica di X
- Ha tempo esecuzione $O(n + d \log d)$, dove n è la dimensione di X e d è il numero di caratteri distinti di X
- Utilizza una coda di priorità come struttura ausiliaria
- Idea: per ogni carattere assegno prima gli ultimi bit della parola-codice associata

Algoritmo di Huffman

Esempio X = aaaabbbaaacc

a compare 8 volte, b 3 volte, c 2 volte

- b,c: caratteri meno frequenti definisco nuovo carattere W che rappresenta b e c;
- codifico b come W0 e c come W1; W rappresenta il prefisso di b e c
- caratteri che iniziano con W compaiono 5 volte; le frequenze sono a compare 8 volte, W compaiono 5 (3+2) volte
- Codifico a con 0 e W come 1. Tenuto conto del passo precedente ottengo

  a=0 b=10 c=11

Algoritmo di Huffman

Idea: per ogni carattere assegno prima gli ultimi bit della parola-codice associata

Esempio

X = abracadabra (a compare 5 volte, b e r 2 volte, d e c una volta)

- c,d: caratteri meno frequenti definisco codifico c come W0 e d come W1; W rappresenta il prefisso della codifica di c e d
- caratteri che iniziano con W compaiono due volte; le frequenze sono quindi a=5 volte , b,r e W compaiono 2 volte
- scelgo b e r e li codifico come b=S0 e r=S1; frequenze diventano a=5, S=4, W=2
- Scelgo i meno frequenti S e W e codifico S come R0 e W come R1; quindi abbiamo b come R00, r come R01, c come R10, d come R11
- Ottengo a e R codifico a con 0 e R con 1
- Codifica finale a=0, b=100, c=110, d=111, r=101

Algoritmo di Huffman

Algorithm Huffman(X):

Input: String X of length n with d distinct characters

Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
  Create a single-node binary tree T storing c.
  Insert T into Q with key $f(c)$.
while len(Q) > 1 **do**
  $(f_1, T_1) = Q$.remove_min()
  $(f_2, T_2) = Q$.remove_min()
  Create a new binary tree T with left subtree T_1 and right subtree T_2.
  Insert T into Q with key $f_1 + f_2$.
$(f, T) = Q$.remove_min()
return tree T

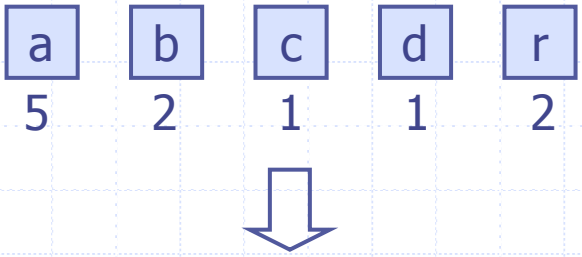
Esempio

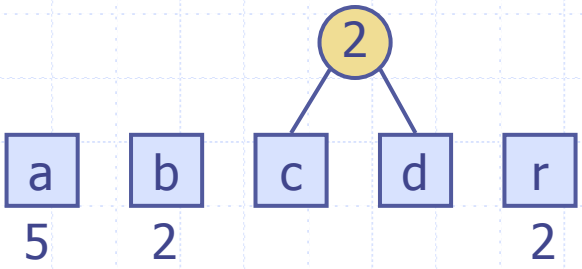
X = abracadabra

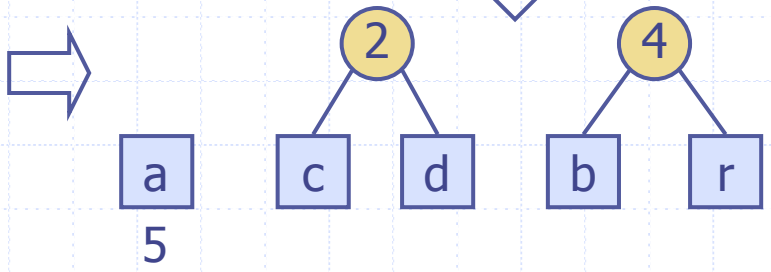
Frequenze

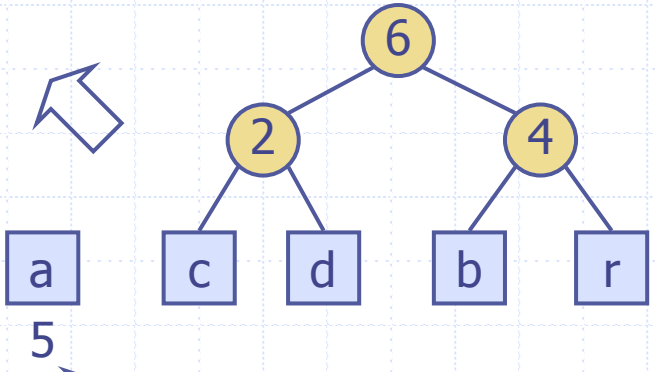
a	b	c	d	r
5	2	1	1	2

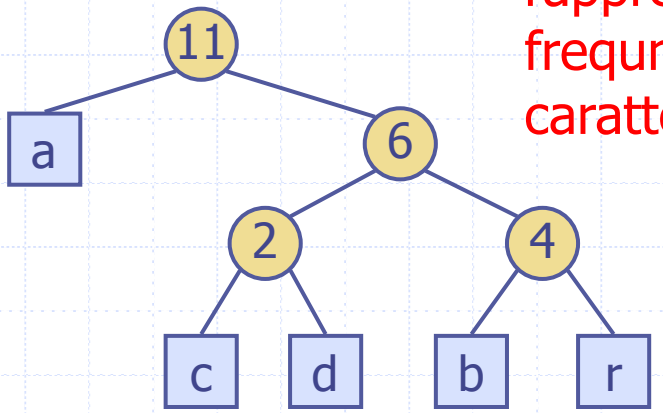
L'intero nel nodo rappresenta la frequnza totale dei caratteri del nodo







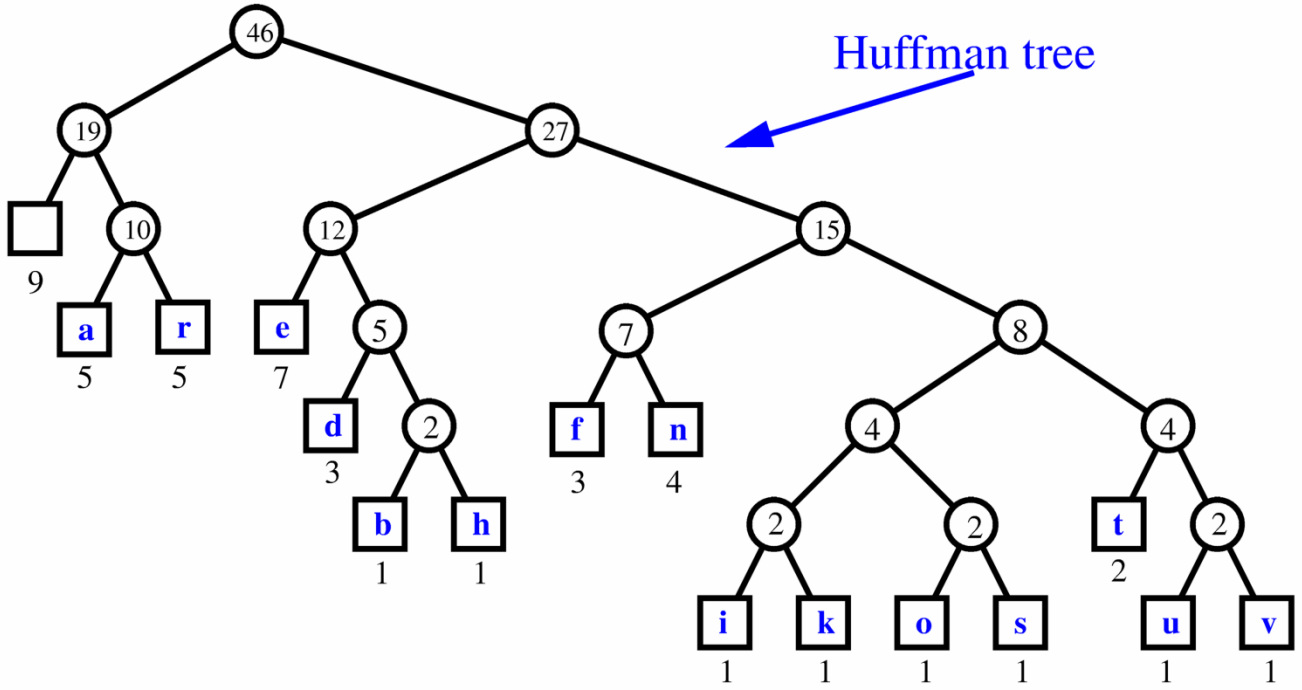




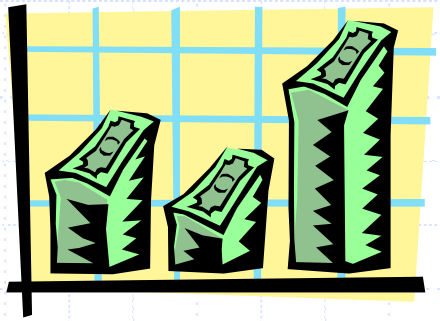
Extended Huffman Tree Example

String: **a fast runner need never be afraid of the dark**

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1



Il Metodo Greedy

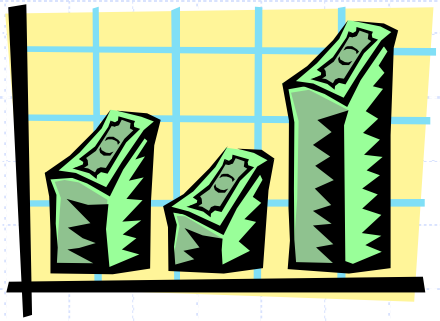


Gli algoritmi Greedy sono particolarmente indicati per la soluzione di quei problemi nei quali è prevista la selezione da un insieme dato di elementi S=(a1, a2,…,an) di un sottoinsieme S' di S, che verificano certe proprietà.

- Il metodo funziona meglio se applicato a problemi che verificano la seguente proprietà **greedy-choice**:
  - Una soluzione globale ottima può essere sempre trovata con una serie di miglioramenti locali a partire da una configurazione inizilale
- Si può applicare anche a problemi che non verificano la proprietà greedy-choice:

  vertex cover, independent set Soddisfacibilità, colorazione,…

Il Metodo Greedy



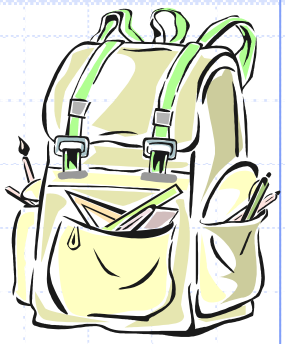
Un algoritmo Greedy ordina dapprima gli oggetti in base al criterio di ottimalità.

La soluzione del problema è costruita in modo incrementale considerando gli elementi uno alla volta ed aggiungendo, se possibile, l'oggetto migliore secondo il criterio di ottimalità.

L'algoritmo effettua quindi una sequenza di scelte, preferendo ogni volta la scelta che fornisce immediatamente il miglior risultato.

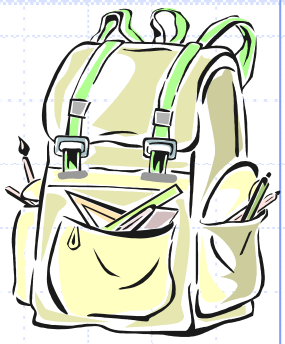
```
procedure Greedy( insieme A ) {
  /* A = {a1,…,an} */
  S = ∅;
  {ordina ai∈A per criterio ottimo};
  for( i=1; i<=n; i++ )
     if ( S∪{ai} è soluzione )
           S = S ∪ {ai};
  /* restituisci S come soluzione */
};
```

Il problema Knapsack (Zaino) frazionario

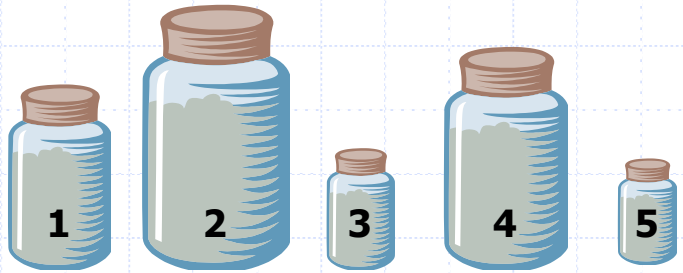


- Input: un insieme S di n elementi, ciascuno con
  - b_i – un beneficio positivo
  - w_i – un peso positivo, una costante W
- Goal: scegli un sottoinsieme di S in modo tale da massimizzare il beneficio totale verificando che il peso totale sia inferiore a W.
- Se possiamo prendere una frazione di ciascun oggetto questo è il problema **Knapsack frazionario**
  - In questo caso sia x_i la frazione scelta dell'elemento i
  - Obiettivo: massimizza $\sum_{i \in S} b_i (x_i / w_i)$
  - Verificando il Vincolo: $\sum_{i \in S} x_i \leq W$

Esempio



- Input: un insieme S di n elementi, ciascuno con
  - b_i – un beneficio positivo
  - w_i – un peso positivo, una costante W
- Valore: ci interessa il rapporto b_i / w_i che rappresenta il valore dell'elemento (beneficio per unità di peso)



elementi:	1	2	3	4	5
Peso:	4 ml	8 ml	2 ml	6 ml	1 ml
Beneficio:	$12	$32	$40	$30	$50
Valore: ($ per ml)	3	4	20	5	50

W=10 ml

"knapsack"

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

Il problema Knapsack frazionario: Algoritmo ottimo

- Scelta Greedy: prima elementi con massimo valore (rapporto beneficio / valore)
  - Nota $\sum_{i \in S} b_i (x_i / w_i) = \sum_{i \in S} (b_i / w_i) x_i$
  - Tempo esecuzione: O(n log n)
- Algoritmo trova ottimo

Prova Supponi no e che SOL sia soluzione migliore di quella Greedy

- In SOL c'è un elemento con elemento i non in SOL con valore maggiore di un elemento j, ma $x_i<w_i$, $x_j>0$ e $v_i<v_j$
- Se sostituisci in SOL i con j, otteniamo soluzione meglio di SOL
- Quanto di i: $\min\{w_i-x_i, x_j\}$
- Quindi non c'è soluzione migliore di quella Greedy

Algorithm ***fractionalKnapsack(S, W)***

Input: set ***S*** of items w/ benefit b_i and weight w_i; max. weight W

Output: amount x_i of each item i to maximize benefit w/ weight at most W

for ***each item i in S***

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {total weight}

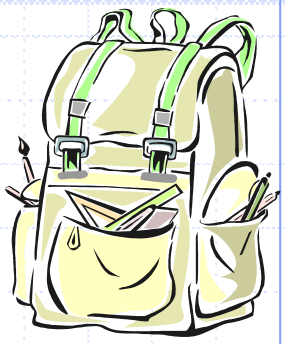
while $w < W$

remove item i w/ highest v_i

$x_i \leftarrow \min\{w_i , W - w\}$

$w \leftarrow w + \min\{w_i , W - w\}$

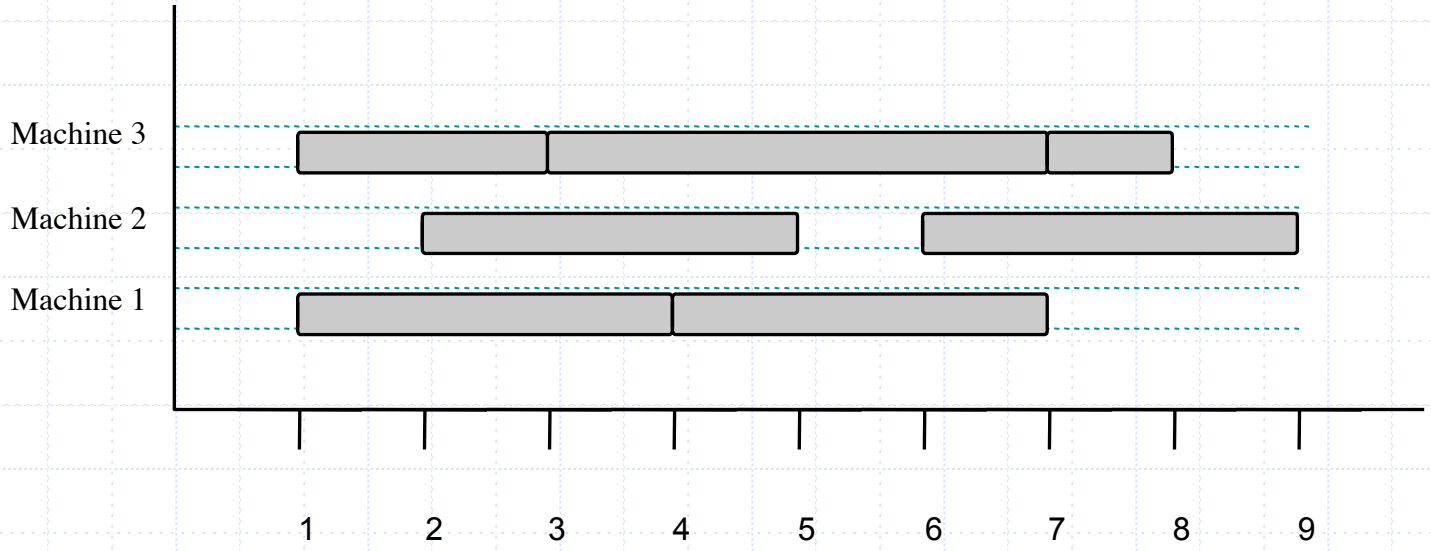
Il problema 0-1 Knapsack (Zaino)



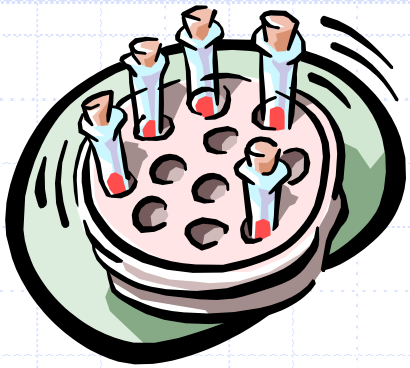
- Input: un insieme S di n elementi, ciascuno con
  - b_i – un beneficio positivo
  - w_i – un peso positivo, una costante W
- **0-1 Knapsack: NON** possiamo prendere una frazione di ciascun oggetto o oggetto intero o nulla
  - In questo caso x_i vale 0 (non prendo ogg. i) o 1 (prendo)
  - Obiettivo: massimizza $\sum_{i \in S} b_i (x_i / w_i)$
  - Verificando il Vincolo: $\sum_{i \in S} x_i \leq W$
- 0-1 Knapsack è NP-completo
- Possiamo applicare algoritmo greedy ma non troveremo soluzione ottima

Sequenziamento di lavori

- Input: un insieme T di n lavori da eseguire, ognuno con
  - Un tempo di inizio s_i
  - Un tempo di terminazione f_i (nota $s_i < f_i$)
- Obiettivo: Esegui tutti i compiti con il minor numero di macchine



Sequenziamento di lavori: Algoritmo



Scelta Greedy: considera lavori dal tempo di inzio e usa il minor numero di macchine con questo ordine

- Tempo esecuzione: O(n log n)

- Correttezza: Supponi ci sia un ordine sequenziamento migliore
  - Possiamo usare k-1 macchine
  - L'algoritmo ne usa k
  - Sia i il primo lavoro eseguito sulla macchina k
  - Macchina i deve essere in conflitto con altri k-1 lavori
  - Ma questo implica che non ci può essere un sequenzia-mento che usa k-1 macchine

Algorithm *taskSchedule*(T)

Input: set T of tasks w/ start time s_i and finish time f_i

Output: non-conflicting schedule with minimum number of machines

$m \leftarrow 0$ {no. of machines}

while *T is not empty*

remove task i w/ smallest s_i

if *there's a machine j for i* **then**

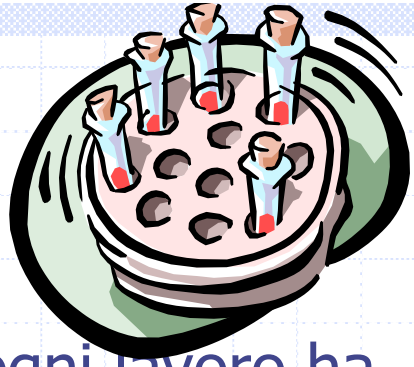
schedule i on machine j

else

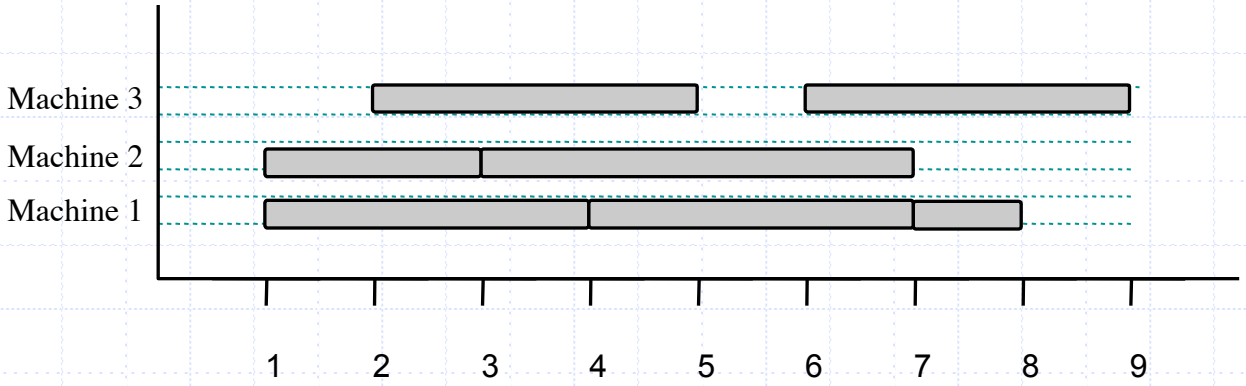
$m \leftarrow m + 1$

schedule i on machine m

Esempio



- Dati un insieme T di n lavori da eseguire; ogni lavoro ha
  - Un tempo di inizio, s_i
  - Un tempo di fine finish time, f_i (nota $s_i < f_i$)
- Esempio: [1,4], [1,3], [2,5], [3,7], [4,7], [6,9], [7,8] (ordinati a partire dall'inizio)
- Goal: Eseguire tutti i lavori con il minimo numero di macchine



Variante del Metodo Greedy

In molti problemi non vale la Greedy-choice property e la soluzione trovata non è ottimale

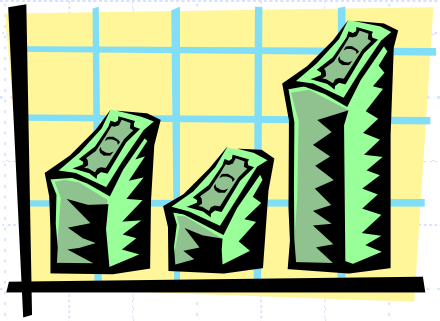
In questi casi è ragionevole applicare una variante del metodo

La soluzione del problema è sempre costruita in modo incrementale considerando gli elementi uno alla volta ed aggiungendo, se possibile, l'oggetto migliore secondo il criterio di ottimalità.

La differenza è che, dopo ogni inserimento fatto nella soluzione, si aggiorna il criterio che misura il beneficio di ciascun elemento

```
Algoritmo Greedy (insieme A)
SOL= Insieme vuoto;
while(A non vuoto) {
        sia i elemento in A che massimizza il beneficio
        A= A –{i}
        if (SOL U {i}) è soluzione ammissibile)
                {SOL= SOL U {i}} / aggiorno SOL/
        }
```

Il Metodo Greedy



Esempio: Vertex Cover

Criterio di scelta: ad ogni passo scegli il vertice v che massimizza il numero di archi che sono coperti ma che non erano coperti dai nodi scelti nelle iterazioni precedenti)

Algoritmo Greedy-Vertex cover (Grafo G(V,A))

SOL= Insieme vuoto;

while(il grafo G ha almeno un arco) {

sia i nodo in G che ha massimo grado

G= G –{i} / elimina nodo i da G con tutti i suoi archi /

{SOL= SOL U {i}

}

Esercizi

- Mostra una istanza del problema 0-1 Knapsack per cui l'algortimo Greedy fornisce la soluzione ottima e una in cui non fornisce la soluzione ottima
- Applica metodo Greedy ai seguenti problemi: SAT, Graph coloring, Independent set
- Per ciascuno dei problemi precedenti mostra una istanza in cui risolve il problema ed una in cui non riesce