Programmazione dinamica (I parte)

Progettazione di Algoritmi a.a. 2017-18

Matricole congrue a 1

Docente: Annalisa De Bonis

Paradigmi della Progettazione degli Algoritmi

- Greedy. Costruisci una soluzione in modo incrementale, ottimizzando (in modo miope) un certo criterio locale.

- Divide-and-conquer. Suddividi il problema in sottoproblemi, risolvi ciascun sottoproblema indipendentemente e combina le soluzioni dei sottoproblemi per formare la soluzione del problema di partenza.

- Programmazione dinamica. Suddividi il problema in un insieme di sottoproblemi che si sovrappongono, cioè che hanno dei sottoproblemi in comune. Costruisci le soluzioni a sottoproblemi via via sempre più grandi **in modo da computare la soluzione di un dato sottoproblema un'unica volta**.

- Nel divide and conquer, se due sottoproblemi condividono uno stesso sottoproblema quest'ultimo viene risolto più volte.

Storia della programmazione dinamica

- Bellman. Negli anni ‘50 è stato il pionere nello studio sistematico della programmazione dinamica.

- Etimologia.
  - Programmazione dinamica = pianificazione nel tempo.

Applicazioni della programmazione dinamica

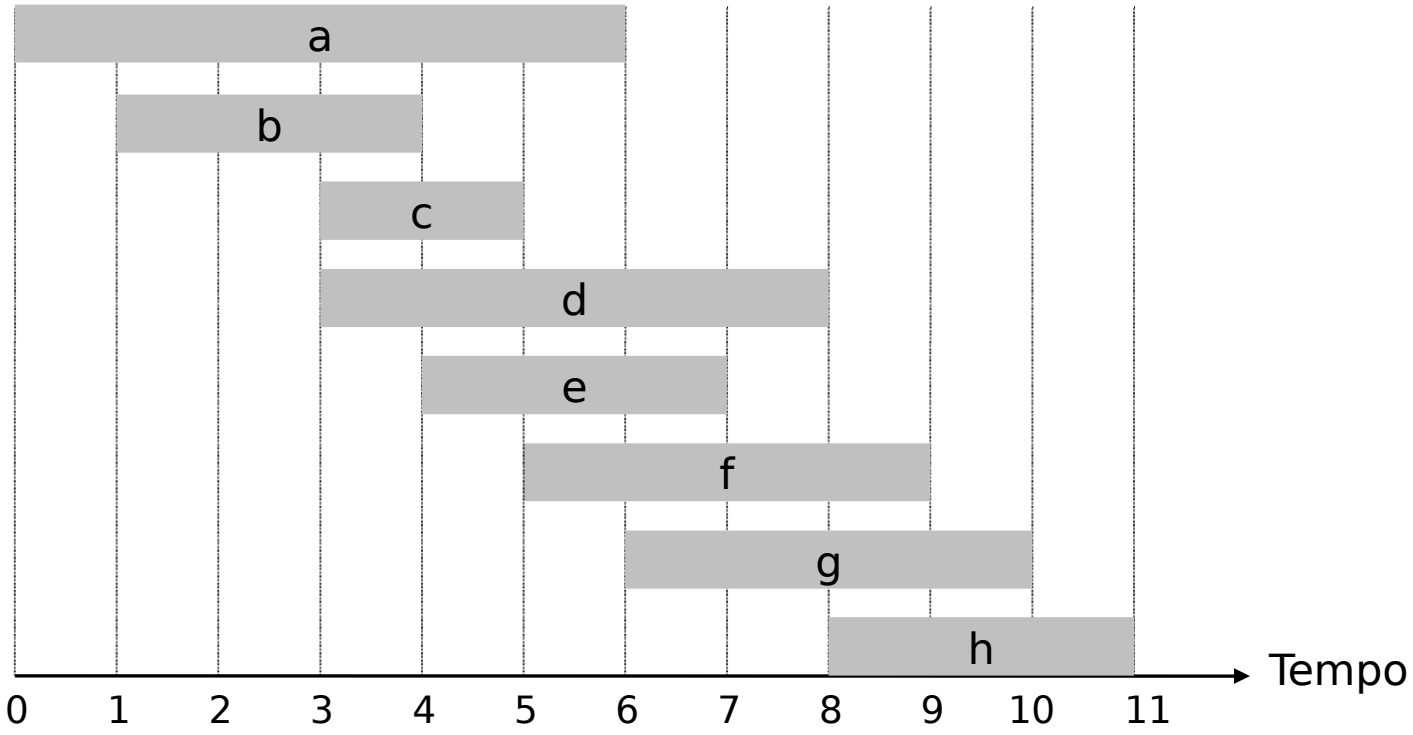
Aree.

- Bioinformatica.
- Teoria dell’informazione
- Ricerca operativa
- Informatica teorica
- Computer graphics
- Sistemi di Intelligenza Artificiale

Interval Scheduling Pesato

Interval scheduling con pesi

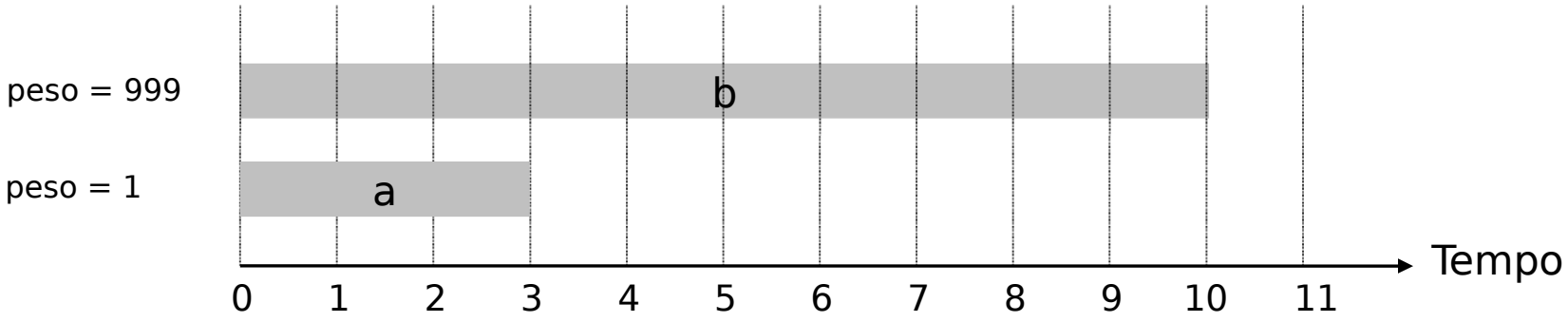
- Job j: comincia al tempo s_j, finisce al tempo f_j, ha associato un valore (peso) v_j .
- Due job sono compatibili se non si sovrappongono
- Obiettivo: trovare il sottoinsieme di job compatibili con il massimo peso totale .



Interval scheduling senza pesi

- L’algoritmo greedy Earliest Finish Time funziona quando tutti i pesi sono uguali ad 1.
- Considera i job in ordine non decrescente dei tempi di fine
- Seleziona un job se è compatibile con quelli già selezionati

Osservazione. L’algoritmo greedy Earliest Finish Time può fallire se i pesi dei job sono valori arbitrari.

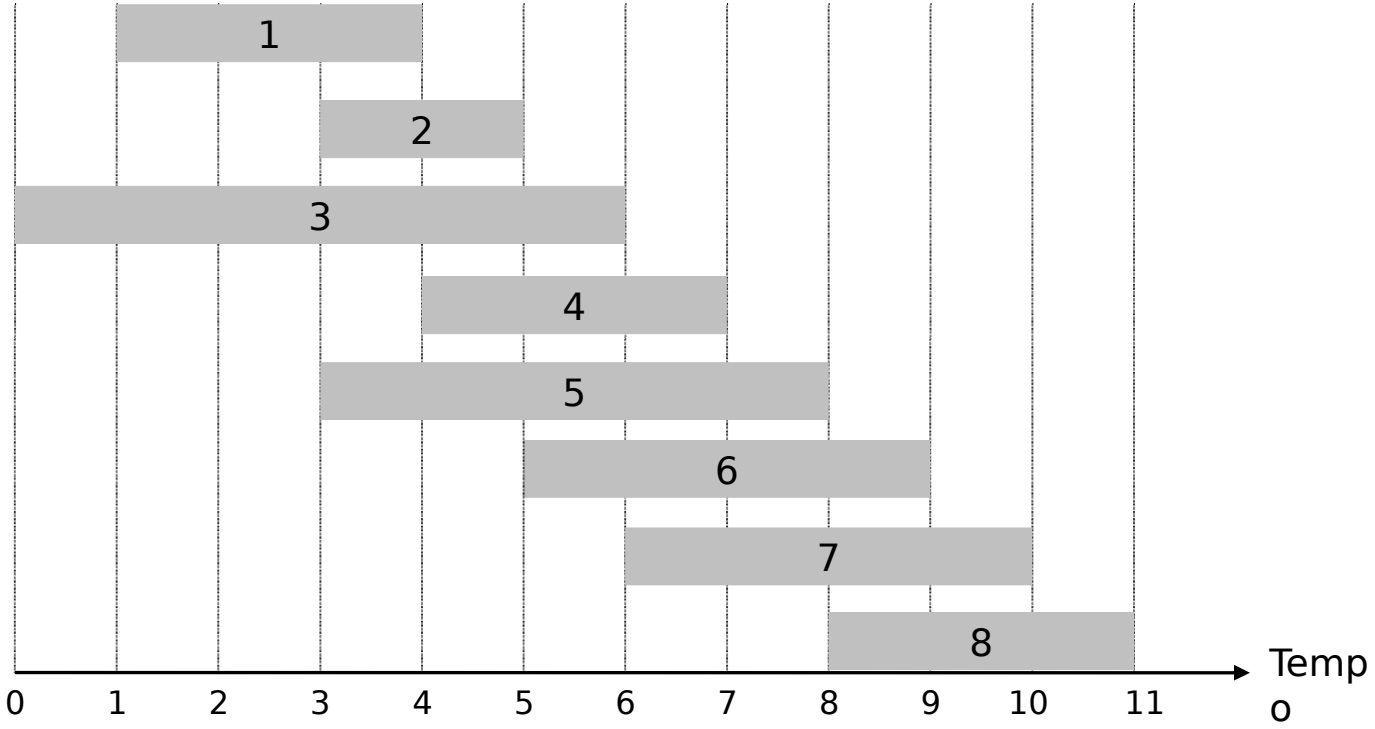


Interval Scheduling Pesato

Notazione. Etichettiamo i job in base al tempo di fine :
$f_1 \leq f_2 \leq \ldots \leq f_n$.

Def. p(j) = il più grande indice i < j tale che i è compatibile con j

Ex: p(8) = 5, p(7) = 3, p(2) = 0.



Interval Scheduling Pesato: soluzione basata sulla PD

- Notazione. OPT(j) = valore della soluzione ottima OPT per il problema che consiste nello schedulare le j richieste con i j tempi di fine più piccoli

  - Caso 1: OPT include il job j.
    - In questo caso la soluzione non può usare i job incompatibili { p(j) + 1, p(j) + 2, ..., j - 1 }
    - Deve includere la soluzione ottima al problema dell'Interval Scheduling Pesato per i job 1, 2, ..., p(j)

  - Caso 2: OPT non seleziona il job j.

In questo caso la soluzione deve includere la soluzione ottima al problema dell'Interval Scheduling Pesato per i job 1, 2, ..., j-1

$$OPT(j) = \begin{cases} 0 & \text{if } j = 0 \\ \max\left\{ v_j + OPT(p(j)),\ OPT(j-1) \right\} & \text{otherwise} \end{cases}$$

Interval Scheduling Pesato: algoritmo ricorsivo **inefficiente**

- Inizialmente Compute-Opt viene invocato con j=n

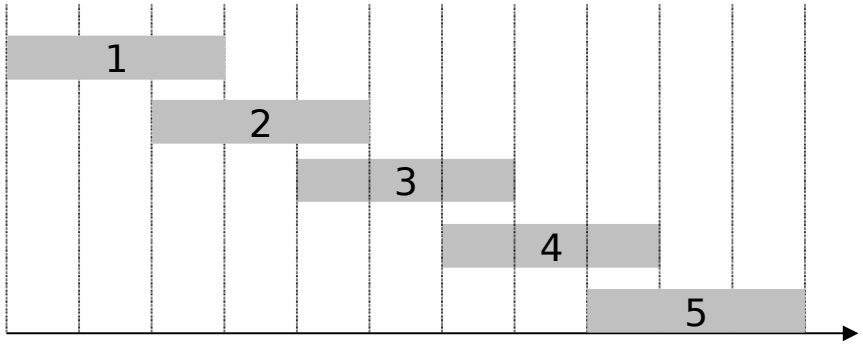
```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Compute-Opt(j) {
   if (j = 0)
      return 0
   else
      return max(vj + Compute-Opt(p(j)), Compute-Opt(j-1))
}
```

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

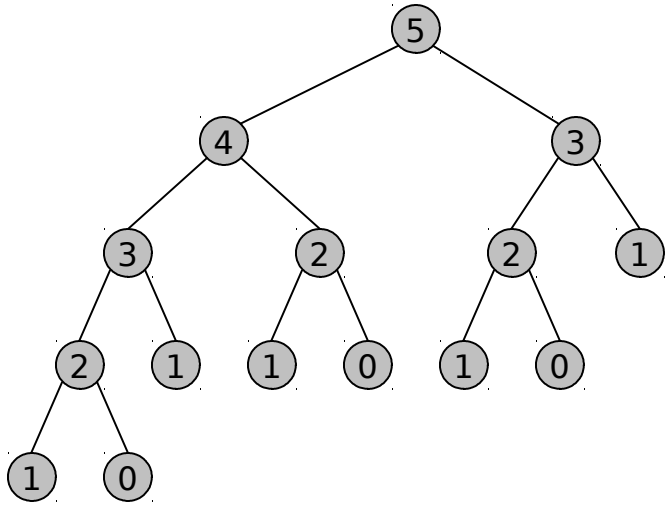
- L'algoritmo computa correttamente OPT(j)
- Dim per induzione.
- Caso base j=0. Il valore restituito è correttamente 0.
- Passo Induttivo. Consideriamo un certo j>0 e supponiamo (ipotesi induttiva) che l'algoritmo produca il valore corretto di OPT(i) per ogni i<j.
- Il valore computato per j dall'algoritmo è
  Compute-Opt(j)= max(v_j + Compute-Opt(p(j),Compute-Opt(j-1))
-
- Siccome per ipotesi induttiva
- **Compute-Opt(p(j))= OPT(p(j)) e**
- **Compute-Opt(j-1)= OPT(j-1)**
-
- allora ne consegue che
- **Compute-Opt(j)= max(v_j + OPT(p(j)), OPT(j-1))= OPT(j)**

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

- Osservazione. L'algoritmo ricorsivo corrisponde ad un algoritmo di forza bruta perchè ha tempo esponenziale
  - Ciò è dovuto al fatto che
    - ✔ Un gran numero di sottoproblemi sono condivisi da più sottoproblemi
    - ✔ L'algoritmo computa più volte la soluzione ad uno stesso sottoproblema.
- **Esempio**. In questo esempio il numero di chiamate ricorsive cresce come i numeri di Fibonacci.
- N(j)= numero chiamate ricorsive per j. N(j)=N(j-1)+N(j-2)



P(1) = 0, p(2)=0 e p(j) = j-2 per ogni j>1



Interval Scheduling Pesato: Memoization

- Osservazione: l'algoritmo ricorsivo precedente computa la soluzione di n+1 sottoproblemi soltanto OPT(0),...,OPT(n). Il motivo dell'inefficienza dell'algoritmo è dovuto al fatto che computa la soluzione ad uno stesso problema più volte.

- Memoization. Consiste nell'immagazzinare le soluzioni di ciascun sottoproblema in un'area di memoria accessibile globalmente.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty        ← array globale

M-Compute-Opt(j) {
   if j = 0 Return 0
   if (M[j] is empty)
      M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
   return M[j]
}
```

inizializzazione

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione. La versione “memoized” dell’algoritmo ha tempo di esecuzione O(n log n).

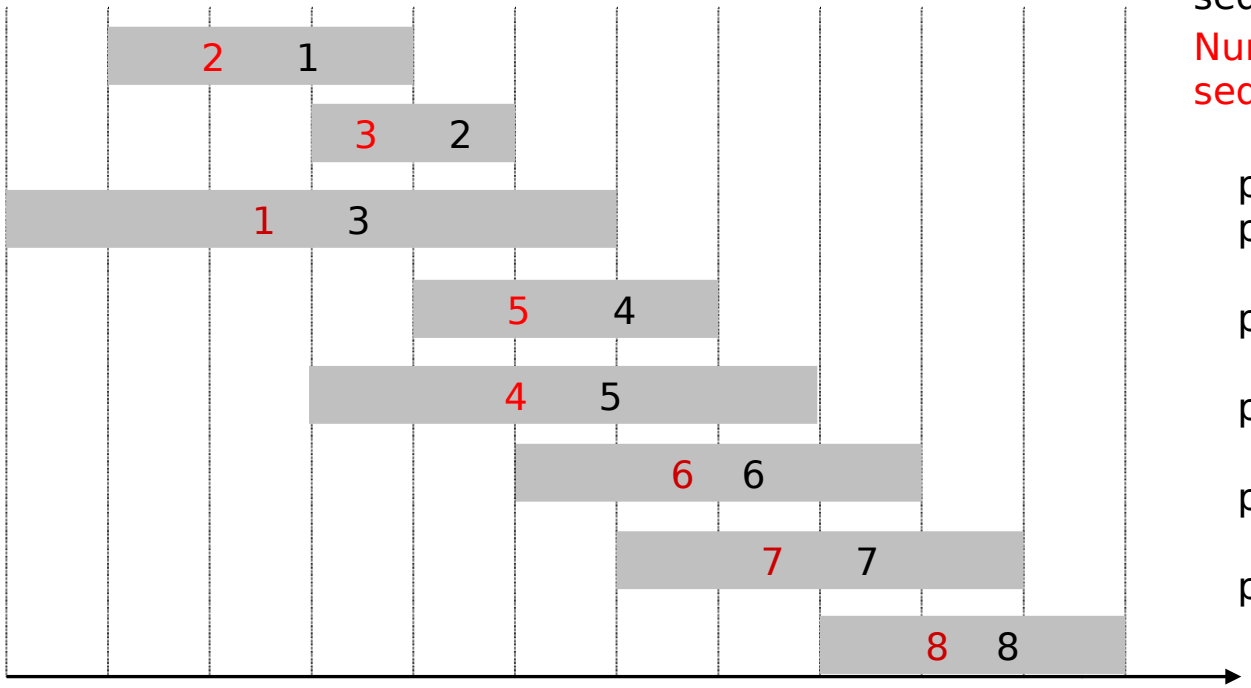
Fase di inizializzazione: O(n log n)

- Ordinamento in base ai tempi di fine: O(n log n).
- Computazione dei valori p(·) : O(n) dopo aver ordinato i job (rispetto ai tempi di inizio e di fine). Siano $a_1,\ldots,a_n$ i job ordinati rispetto ai tempi di inizio e $b_1,\ldots,b_n$ i job ordinati rispetto ai tempi di fine. (si noti che il job con l’i-esimo tempo di inizio non corrisponde necessariamente a quello con l’i-esimo tempo di fine)
  - Si confronta il tempo di fine di b_1 con i tempi di inizio di $a_1,a_2,a_3\ldots$, fino a che non si incontra un job a_j con tempo di inizio $\geq f_1$. Si pone $p'(1)=p'(2)=\ldots=p'(j-1)=0$. Si confronta il tempo di fine di b_2 con i tempi di inizio di $a_j,a_{j+1},a_{j+2}\ldots$, fino a che non si incontra un job a_k con tempo di inizio $\geq f_2$. Si pone $p'(j)=p'(j+1)=p'(j+2)=\ldots=p'(k-1)=1$. Si confronta il tempo di fine di b_3 con i tempi di inizio di $a_k,a_{k+1},a_{k+2}\ldots$, fino a che non si incontra un job a_m con tempo di inizio $\geq f_3$. Si pone $p'(k)=p'(k+1)=p'(k+2)=\ldots=p'(m-1)=2$, e cosi` via.

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

- Si noti che in p'(j), j e` l'indice del job nella sequenza $a_1,\ldots,a_n$.
  - Per ottenere il corrispondente valore p(r) basta sostituire a j l'indice del job nella sequenza $b_1,\ldots,b_n$. L'associazione tra il job e la sua posizione nella sequenza $a_1,\ldots,a_n$ e quella nella sequenza $b_1,\ldots,b_n$ puo` essere creata quando si ordinano i job.



Numeri neri= indici nella sequenza ordinata $b_1,\ldots,b_n$

Numeri rossi= indici nella sequenza ordinata $a_1,\ldots,a_n$

p'(1)=p'(3)=p'(4)=0
p(3)=p(2)=p(5)=0

p'(5)=1 p(4)=1

p'(6)=2 p(6)=2

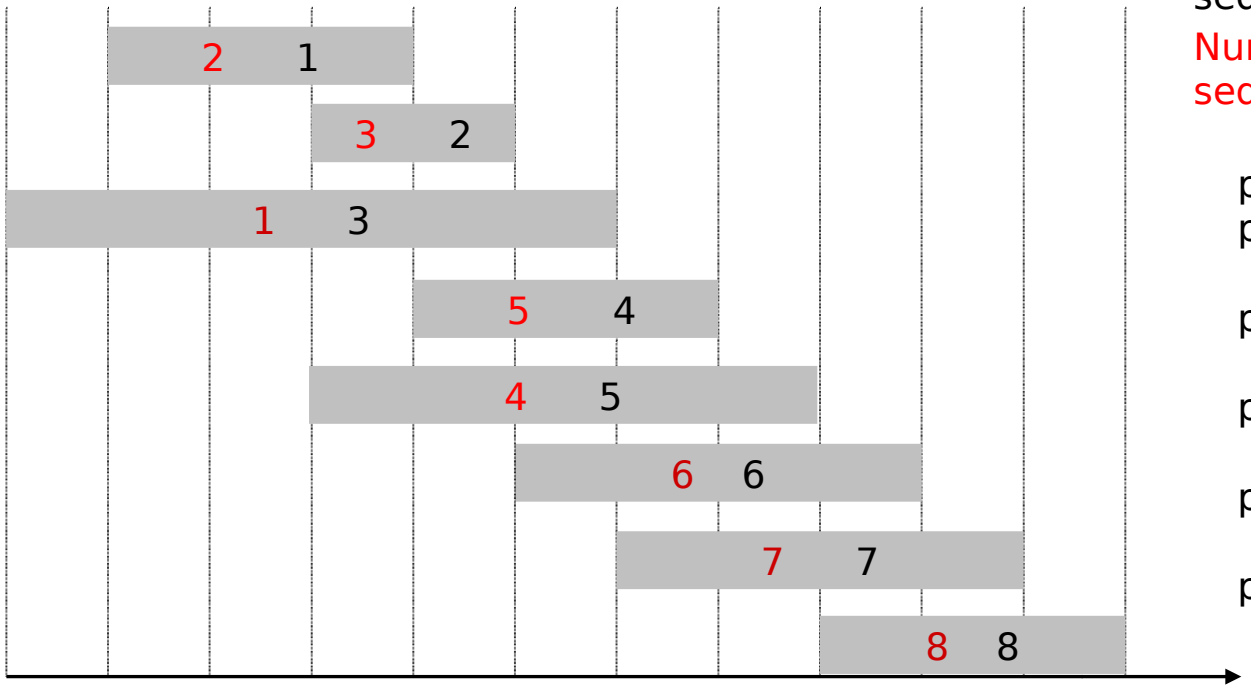
p'(7)=3 p(7)=3

p'(8)=5 p(8)=5

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

- Il tempo per calcolare i valori p(1),…,p(n) (una volta ottenuti i due ordinamenti $a_1,\ldots,a_n$ e $b_1,\ldots,b_n$) e` O(n) perche` dopo ogni confronto l'algoritmo passa a considerare o il prossimo job nell'ordinamento $b_1,\ldots,b_n$ (nel caso di confronto tra due job compatibili) o il prossimo job nell'ordinamento $a_1,\ldots,a_n$ (nel caso di confronto tra due job incompatibili).



Numeri neri= indici nella sequenza ordinata $b_1,\ldots,b_n$

Numeri rossi= indici nella sequenza ordinata $a_1,\ldots,a_n$

p'(1)=p'(2)=p'(3)=p'(4)=0
p(3)=p(1)=p(2)=p(5)=0

p'(5)=1 p(4)=1

p'(6)=2 p(6)=2

p'(7)=3 p(7)=3

p'(8)=5 p(8)=5

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

Affermazione: **`M-Compute-Opt(n)`** richiede O(n)

Dim.

- **`M-Compute-Opt(j)`**: escludendo il tempo per le chiamate ricorsive, ciascuna invocazione prende tempo O(1) e fa una delle seguenti cose
  - (i) restituisce il valore esistente di `M[j]`
  - (ii) riempie l'entrata `M[j]` facendo due chiamate ricorsive

- Per stimare il tempo di esecuzione di **`M-Compute-Opt(j)`** dobbiamo stimare il numero totale di chiamate ricorsive innescate da **`M-Compute-Opt(j)`**
  - Abbiamo bisogno di una misura di come progredisce l'algoritmo
  - Misura di progressione Φ = # numero di entrate non vuote di `M[]`.
  - inizialmente $\Phi = 0$ e durante l'esecuzione si ha sempre $\Phi \leq n$.
  - per far crescere Φ di 1 occorrono al piu` 2 chiamate ricorsive.
  - quindi per far andare Φ da 0 a j, occorrono al piu` 2j chiamate ricorsive per un tempo totale di O(j)

- Il tempo di esecuzione di **`M-Compute-Opt(n)`** e`quindi O(n).
  N.B. O(n), una volta ordinati i job in base ai valori di inizio.

Memoization nei linguaggi di programmazione

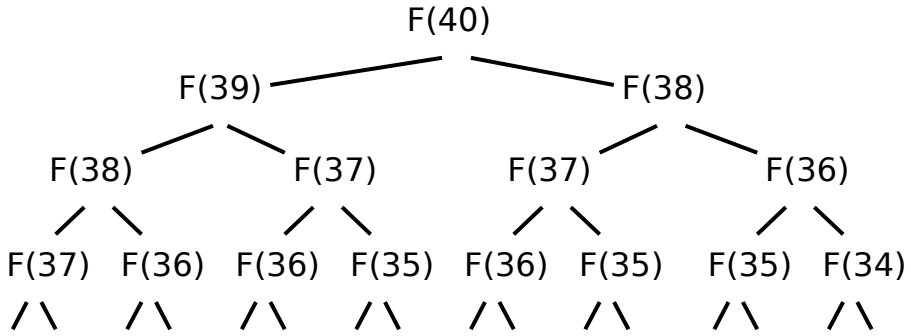
- Automatica. Molti linguaggi di programmazione funzionale, quali il Lisp, prevedono un meccanismo per rendere automatica la memoization

```
(defun F (n)
  (if
    (<= n 1)
    n
    (+ (F (- n 1)) (F (- n 2)))))
```

Lisp (efficiente)

```
static int F(int n) {
   if (n <= 1) return n;
   else return F(n-1) + F(n-2);
}
```

Java (esponenziale



Interval Scheduling Pesato: Trovare una soluzione

- Domanda. Gli algoritmi di programmazione dinamica computano il valore ottimo. E se volessimo trovare la soluzione ottima e non solo il suo valore?
- Risposta. Facciamo del post-processing (computazione a posteriori).

```
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
   if (j = 0)
      output nothing
   else if (v_j + M[p(j)] > M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

Interval Scheduling Pesato: Bottom-Up

Programmazione dinamica bottom-up

Per capire il comportamento dell'algoritmo di programmazione dinamica e` di aiuto formulare una versione iterativa dell'algoritmo.

```
Input: n, s1,…,sn , f1,…,fn , v1,…,vn

Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.

Compute p(1), p(2), …, p(n)

Iterative-Compute-Opt {
   M[0] = 0
   for j = 1 to n
      M[j] = max(vj + M[p(j)], M[j-1])
}
```

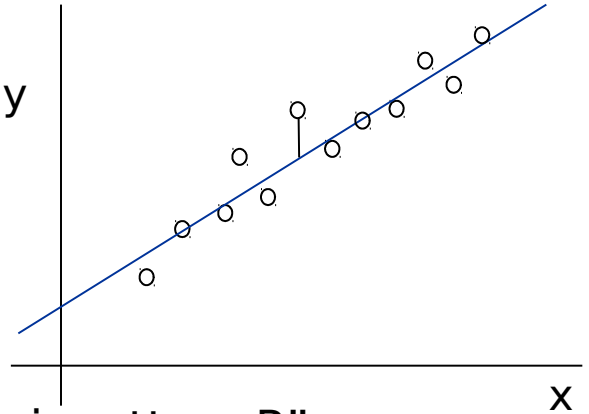
Correttezza: Con l'induzione su j si puo` dimostrare che ogni entrata M[j] contiene il valore OPT(j)

Tempo di esecuzione: n iterazioni del for, ognuna della quali richiede tempo O(1) → tempo totale O(n)

Segmented Least Squares

- Minimi quadrati.
  - Problema fondazionale in statistica e calcolo numerico.
  - Dato un insieme P di n punti del piano $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
  - Trovare una linea L di equazione $y = ax + b$ che minimizza la somma degli errori quadratici.

$$Error(L,P) = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$



  - Chiameremo questa quantita` “Errore di L rispetto a P”

  - Chiameremo “Errore minimo per P”, il minimo valore di Error(L,P) su tutte le possibili linee L

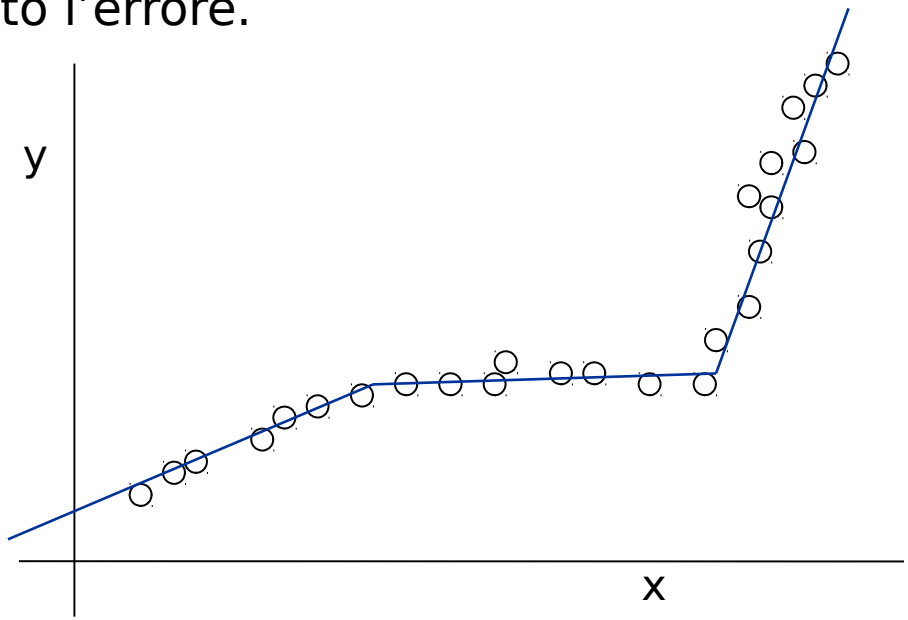
- Soluzione. Analisi $\Rightarrow$ il minimo errore per un dato insieme P di punti si ottiene usando la linea di equazione $y = ax + b$ con a e b dati da

$$a = \frac{n\sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n\sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a\sum_i x_i}{n}$$

Segmented Least Squares

- L’errore minimo per alcuni insiemi input di punti puo` essere molto alto a causa del fatto che i punti potrebbero essere disposti in modo da non poter essere ben approssimati usando un’unica linea.

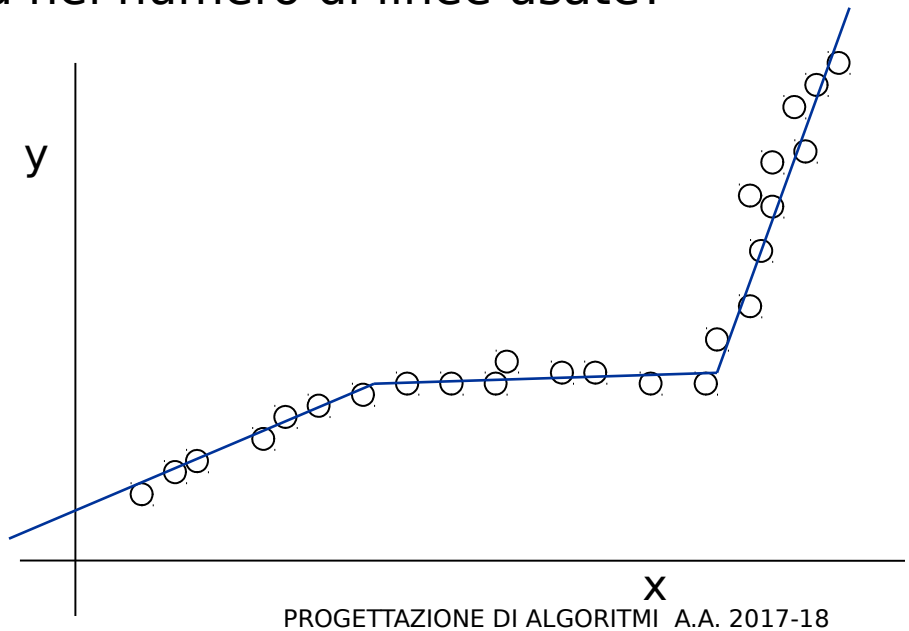
Esempio: i punti in figura non possono essere ben approssimati usando un’unica linea. Se pero` usiamo tre linee riusciamo a ridurre di molto l’errore.



Segmented Least Squares

Segmented least squares.

- In generale per ridurre l'errore avremo bisogno di una sequenza di linee intorno alle quali si distribuiscono sottoinsiemi di punti di P.
- Ovviamente se ci fosse concesso di usare un numero arbitrariamente grande di segmenti potremmo ridurre a zero l'errore:
  - Potremmo usare una linea ogni coppia di punti consecutivi.
- Domanda. Qual e` la misura da ottimizzare se vogliamo trovare un giusto compromesso tra accuratezza della soluzione e parsimonia nel numero di linee usate?



Segmented Least Squares

Formulazione del problema Segmented Least Squares.

- Dato un insieme P di n punti nel piano $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ con $x_1 < x_2 < \ldots < x_n$, vogliamo partizionare P in un certo numero m di sottoinsiemi $P_1, P_2, \ldots, P_m$ in modo tale che
- Ciascun P_i e` costituito da punti contigui lungo l'asse delle ascisse
  - P_i viene chiamato segmento
- La sequenza di linee $L_1, L_2, \ldots, L_m$ ottime rispettivamente per $P_1, P_2, \ldots, P_m$ minimizzi la somma delle 2 seguenti quantita:

1) La somma E degli m errori minimi per $P_1, P_2, \ldots, P_m$ (l'errore minimo per il segmento P_i e` ottenuto dalla linea L_i)

$$E = Error(L_1, L_2, \ldots, L_m; P_1, P_2, \ldots, P_m) = \sum_{j=1}^{m} \sum_{(x_i, y_i) \in P_j} (y_i - a_j x_i - b_j)^2$$

2) Il numero m di linee (pesato per una certa costante C>0)

- Penalita` : E + C m, dove C > 0 e` una costante.

Segmented Least Squares

- Il numero di partizioni in segmenti dei punti in P e` esponenziale
- La programmazione dinamica ci permette di progettare un algoritmo efficiente per trovare una partizione di penalita` minima
- A differenza del problema dell'Interval Scheduling Pesato in cui utilizzavamo una ricorrenza basata su due possibili scelte, per questo problema utilizzeremo una ricorrenza basata su un numero polinomiale di scelte.
-

Approccio basato sulla programmazione dinamica

Notazione

- OPT(j) = costo minimo della penalita` per i punti $p_1, p_2, \ldots, p_j$.
- e(i, j) = minimo errore per l'insieme di punti $\{p_i, p_{i+1}, \ldots, p_j\}$.

Per computare OPT(j), osserviamo che

- se l'ultimo segmento nella partizione di $\{p_1, p_2, \ldots, p_j\}$ e` costituito dai punti $p_i, p_{i+1}, \ldots, p_j$ per un certo i, allora
- penalita` = e(i, j) + C + OPT(i-1).

- Il valore della penalita` cambia in base alla scelta di i
- Il valore OPT(j) e` ottenuto in corrispondenza dell'indice i che minimizza e(i, j) + C + OPT(i-1)

$$OPT(j) = \begin{cases} 0 & \text{se } j = 0 \\ \min_{1 \le i \le j} \left\{ e(i,j) + C + OPT(i-1) \right\} & \text{altrimenti} \end{cases}$$

Segmented Least Squares: Algorithm

```
INPUT: n, p_1,…,p_n, c

Segmented-Least-Squares() {
   M[0] = 0
   for j = 1 to n
      for i = 1 to j
         compute the least square error e_ij for
         the segment p_i,…, p_j

   for j = 1 to n
      M[j] = min_{1 ≤ i ≤ j} (e_ij + C + M[i-1])

   return M[n]
}
```

Tempo di esecuzione. $O(n^3)$.

- Collo di bottiglia = dobbiamo computare il valore e(i, j) per $O(n^2)$ coppie i, j. Usando la formula per computare la minima somma degli errori quadratici, ciascun e(i,j) e` computato in tempo O(n)

Algoritmo che produce la partizione

Tempo di esecuzione $O(n^2)$ se abbiamo memorizzato i valori $e_{i,j}$

```
Find-Segments(j)
  If j = 0 then
    Output nothing
  Else
    Find an i that minimizes e_{i,j} + C + M[i − 1]
    Output the segment {p_i, ..., p_j} and the result of
          Find-Segments(i − 1)
  Endif
```

Subset sums

Input

- n job 1,2,…,n
  - il job i richiede tempo $w_i > 0$.
- Un limite W al tempo per il quale il processore puo` essere utilizzato

- Obiettivo: selezionare un sottoinsieme S degli n job tale che $\sum_{i \in S} w_i$ sia quanto piu` grande e` possibile, con il vincolo $\sum_{i \in S} w_i \leq W$

Greedy 1: ad ogni passo inserisce in S il job con peso piu` alto in modo che la durata complessiva dei job in S non superi W

Esempio: Input una volta ordinato [W/2+1,W/2,W/2]. L'algoritmo greedy seleziona solo il primo mentre la soluzione ottima e` formata dagli ultimi due.

Greedy 2: ad ogni passo inserisce in S il job con peso piu` basso in modo che la duranta complessiva dei job in S non superi W

Esempio: Input [1,W/2,W/2] una volta ordinato . L'algoritmo greedy seleziona i primi due per un peso complessivo di 1+W/2. mentre la soluzione ottima e` formata dagli ultimi due di peso complessivo W.

Programmazione dinamica: falsa partenza

- Def. OPT(i) = valore della soluzione ottima per {1, …, i}.

  - Caso 1: OPT non include i.
    - OPT include la soluzione ottima per { 1, 2, …, i-1 }

  - Caso 2: OPT include i.
    - Prendere i non implica immediatamente l'esclusione di altri elementi.
    - Se non conosciamo i job selezionati prima di i, non sappiamo neanche se c'e` tempo sufficiente per eseguire i

- Conclusione. Approccio sbagliato!

Programmazione dinamica: approccio corretto

- Per esprimere il valore della soluzione ottima per un certo i in termini dei valori delle soluzioni ottime per input piu` piccoli di i, dobbiamo introdurre un limite al tempo totale da dedicare all'esecuzione dei job selezionati prima di i.

- Per ciascun j, consideriamo il valore della soluzione ottima per i job 1, …, j con il vincolo che il tempo necessario per eseguire i job selezionati non superari un certo w.

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

Programmazione dinamica: approccio corretto

- Def. OPT(i, w) = valore della soluzione ottima per i job 1, …, i con limite w sul tempo di utilizzo del processore.

- Caso 1: OPT non include il job i.
  - OPT include la soluzione ottima per { 1, 2, …, i-1 } in modo che il tempo di esecuzione totale dei job non superi w

- Case 2: OPT include il job i.
  - OPT include la soluzione ottima per { 1, 2, …, i–1 } in modo che il tempo di esecuzione totale dei job non superi w– w_i

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{OPT(i-1,w),\ w_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Subset sums: algoritmo

- Versione iterativa in cui si computa la soluzione in modo bottom-up
- Si riempie un array bidimensionale nxW a partire dalle locazioni di indice di riga i piu` piccolo

```
Input: n, w1,…,wn ,W

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], wi + M[i-1, w-wi ]}

return M[n, W]
```

Subset sums: esempio di esecuzione dell'algoritmo

Limite $W = 6$, durate job $w_1 = 2$, $w_2 = 2$, $w_3 = 3$

3							
2							
1							
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Initial values

3							
2							
(1)	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 1$

3							
(2)	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 2$

(3)	0	0	2	3	4	5	5
2	0	0	2	2	4	4	4
1	0	0	2	2	2	2	2
0	0	0	0	0	0	0	0
	0	1	2	3	4	5	6

Filling in values for $i = 3$

Subset sums: correttezza algoritmo

Induzione sui i. Dimostriamo che all'i-esima iterazione ogni riga di M con indice r compreso tra 0 e i contiene i valori OPT(r,0),OPT(r,1), ….,OPT(r, W)

- Base induzione. i=0: La riga 0 ha correttamente tutte le entrate uguali a 0
- Passo induttivo: Supponiamo che alla iterazione i-1 ≥0, le righe di indice r compreso tra 0 e i-1 contengano i valori OPT(r,0),OPT(r,1),….,OPT(r, W).
- Vediamo cosa succede all'i-esima iterazione.
- Per ipotesi induttiva M[i-1, w]=OPT(i-1,w) ed M[i-1, w-w_i]=OPT(i-1,w-w_i)
- All'i-esima iterazione, l'algoritmo pone
- **M[i, w] =** M[i-1, w] **=** OPT(i-1,w) **se w_i > w**

M[i, w] = max {M[i-1, w], w_i + M[i-1, w-w_i]}
= max {OPT[i-1, w], w_i + OPT[i-1, w-w_i]} altrimenti

Subset sums: tempo di esecuzione algoritmo

- Tempo di esecuzione. $\Theta(n\ W)$.

  - Non e` polinomiale nella dimensione dell'input!
  - "Pseudo-polinomiale": L'algoritmo e` efficiente quando W ha un valore ragionevolmente piccolo.

  - Se volessimo produrre la soluzione ottima, potremmo scrivere un algoritmo simile a quelli visti prima in cui la soluzione ottima si ricostruisce andando a ritroso nella matrice M. Tempo O(n).

  - Esercizio: Scrivere lo pseudocodice dell'algoritmo che produce la soluzione ottima per un'istanza di Subset Sum.

Problema dello zaino

- Input
  - n oggetti ed uno zaino
  - L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
  - Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.
- Esempio: { 3, 4 } ha valore 40.

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Greedy: seleziona ad ogni passo l'oggetto con il rapporto v_i/w_i piu` grande in modo che il peso totale dei pesi selezionati non superi w
- Esempio: soluzione greedy { 5, 2, 1 } ha valore = 35 $\Rightarrow$ greedy non e` ottimo

Problema dello zaino

Input

- n oggetti ed uno zaino
- L'oggetto i pesa $w_i > 0$ chili e ha valore $v_i > 0$.
- Lo zaino puo` trasportare fino a W chili.
- Obiettivo: riempire lo zaino in modo da massimizzare il valore totale degli oggetti inseriti.

Corrisponde al problema subset sums quanto $v_i = w_i$ per ogni i.

Problema dello zaino: estensione approccio usato per Subset Sums

Def. OPT(i, w) = valore della soluzione ottima per gli oggetti 1, …, i con limite di peso totale w.

- Caso 1: OPT non include l'elemento i.
  - OPT include la soluzione ottima per { 1, 2, …, i-1 } in modo che il peso totale degli elementi non superi w

- Case 2: OPT include l'elemento i.
  - OPT include la soluzione ottima per { 1, 2, …, i–1 } in modo che il peso totale degli elementi non superi w– w_i

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{ OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\} & \text{otherwise} \end{cases}$$

Problema dello zaino: algoritmo

```
Input: n, W,w1,…,wn, v1,…,vn

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 0 to W
      if (wi > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi ]}

return M[n, W]
```

Algoritmo per il problema della zaino: esempio

W →

n ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	35	40

$OPT(5,11) = OPT(4,11) = v_4 + OPT(3,11-w_4) =$
$v_4 + OPT(3,5) = v_4 + v_3 + OPT(2,0) = v_4 + v_3 + OPT(1,0)$
$= v_4 + v_3 + OPT(0,0) = 22 + 18 + 0 = 40$

Soluzione ottima : { 4, 3 }
Valore soluzione ottima = 22 + 18 = 40

W = 11

Oggetto	Valore	Peso
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7