

Programmazione dinamica

- Progettazione di Algoritmi a.a. 2016-17
- 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 pioniere nello studio sistematico della programmazione dinamica.
- **Etimologia.**
 - Programmazione dinamica = pianificazione nel tempo.

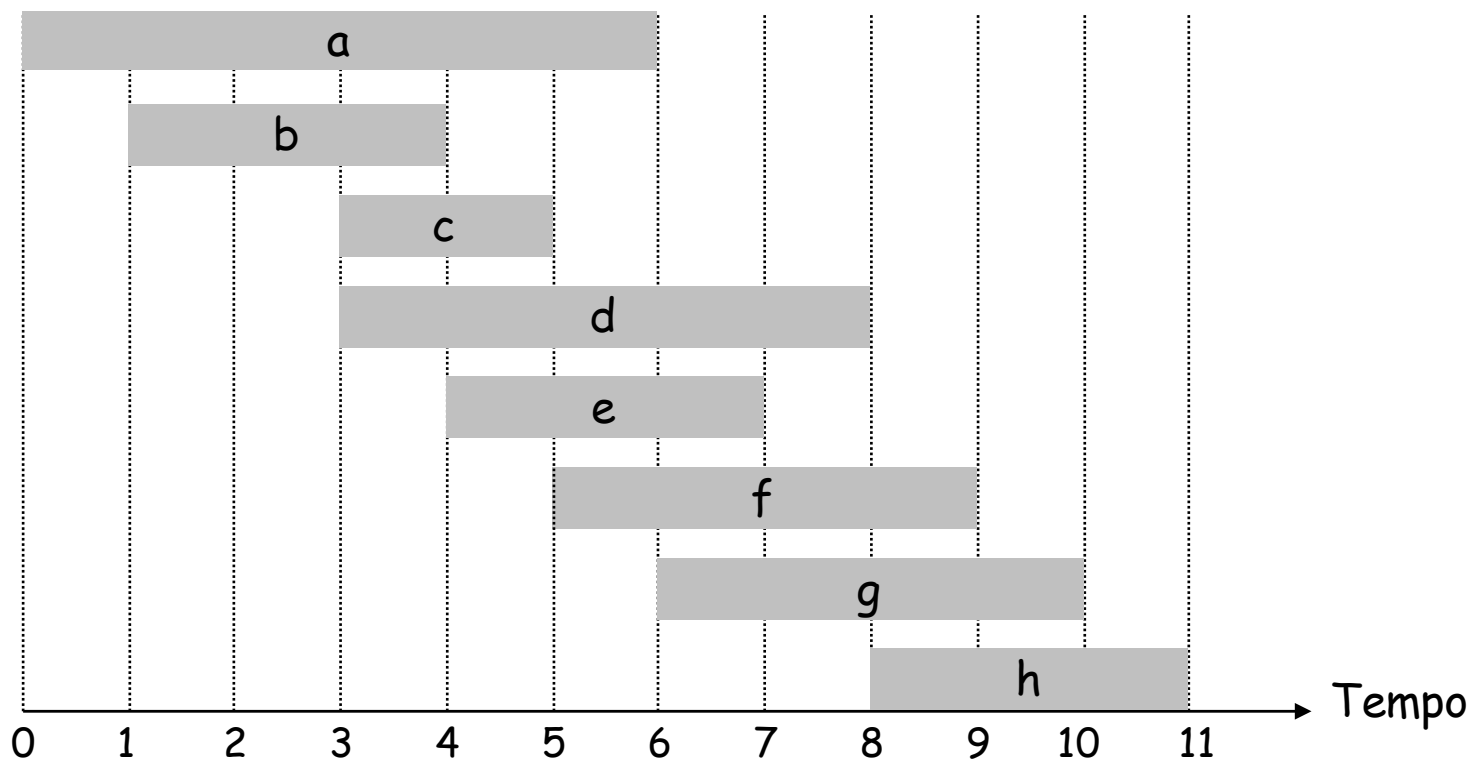
Applicazioni della programmazione dinamica

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

Interval Scheduling Pesato

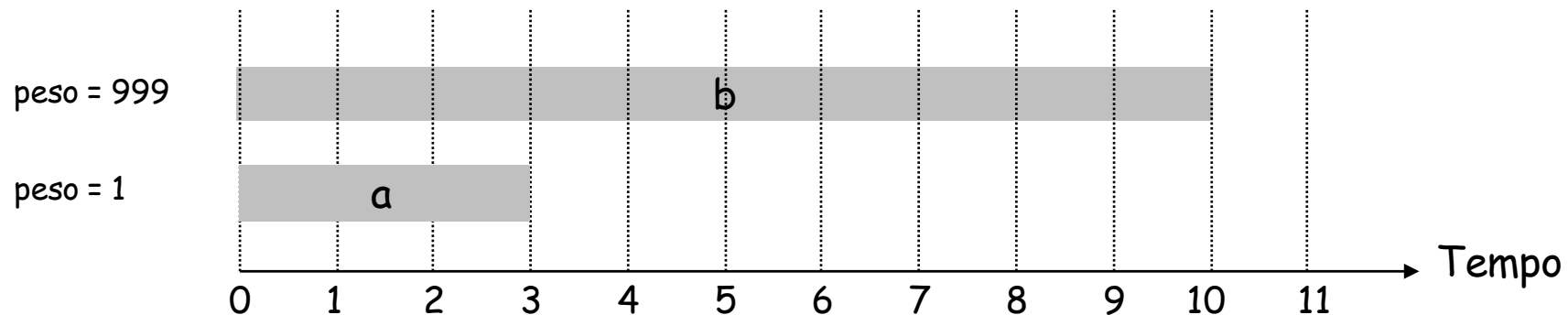
Interval scheduling con pesi

- 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.



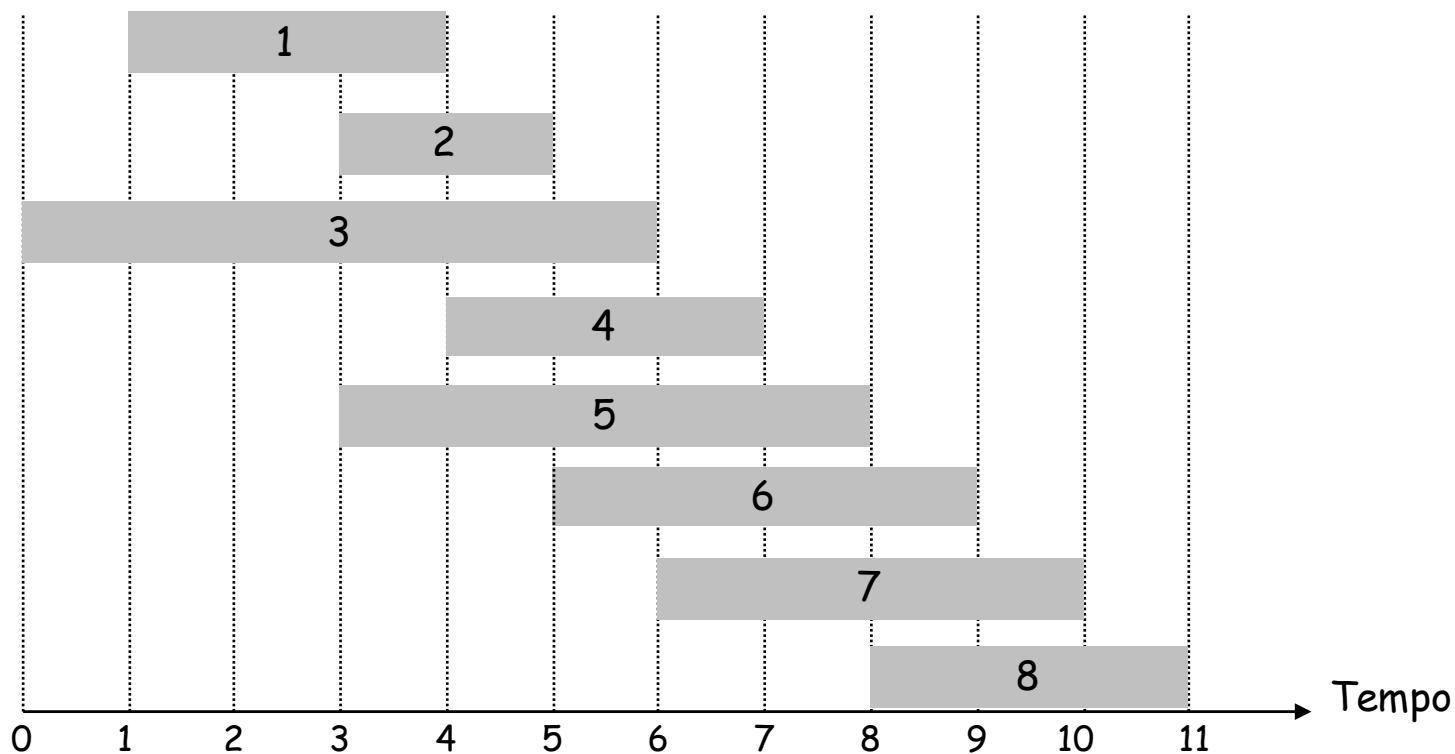
Interval Scheduling Pesato

Notazione. Etichettiamo i job in base al tempo di fine :

$$f_1 \leq f_2 \leq \dots \leq f_n.$$

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** seleziona il job j .
 - In questo caso la soluzione non può usare i job incompatibili $\{ p(j) + 1, p(j) + 2, \dots, j - 1 \}$
 - Deve includere la soluzione ottima al problema che consiste nello schedulare i job $1, 2, \dots, p(j)$
 - **Caso 2:** **OPT** non seleziona il job j .
 - In questo caso la soluzione deve includere la soluzione ottima al problema che consiste nello schedulare i job $1, 2, \dots, j-1$

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

Interval Scheduling Pesato: algoritmo ricorsivo inefficiente

- Inizialmente Compute-Opt viene invocato con $j=n$

```
Input:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
Compute-Opt( $j$ ) {  
    if ( $j = 0$ )  
        return 0  
    else  
        return  $\max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$   
}
```

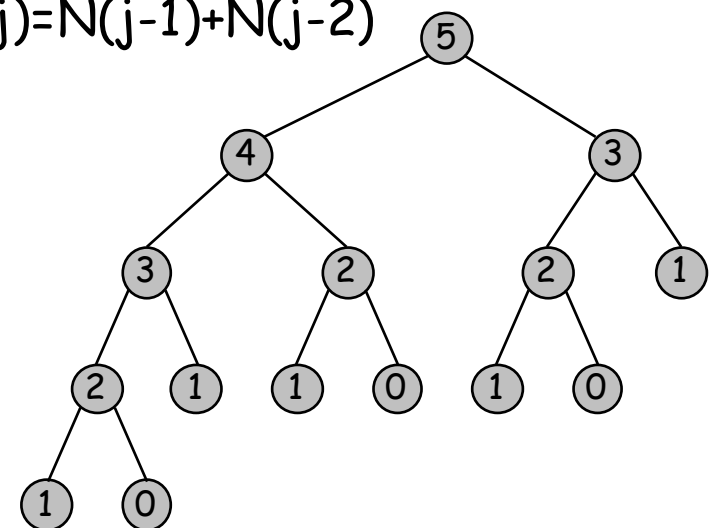
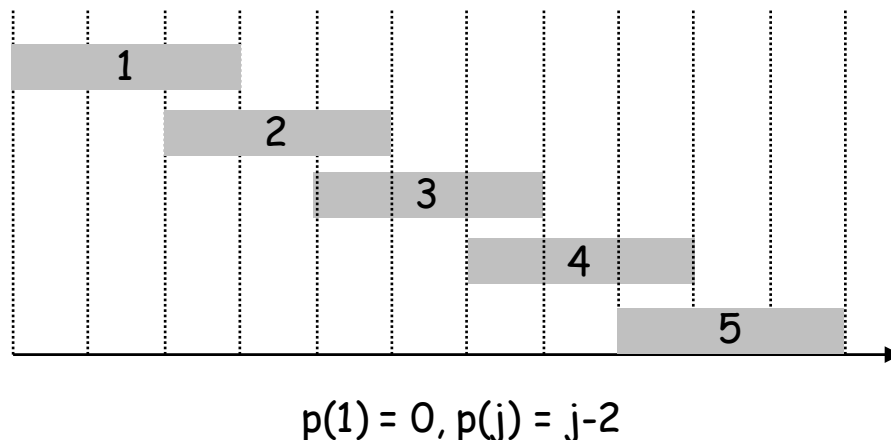
•

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
$$\text{Compute-Opt}(j) = \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))$$
- Siccome per ipotesi induttiva $\text{Compute-Opt}(p(j)) = OPT(p(j))$ e
- $\text{Compute-Opt}(j-1) = OPT(j-1)$ allora si ha
-
- $\text{Compute-Opt}(j) = \max(v_j + OPT(p(j)), OPT(j-1)) = OPT(j)$

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 . $N(j) = N(j-1) + N(j-2)$



Interval Scheduling Pesato: Memoization

- **Osservazione:** l'algoritmo ricorsivo precedente computa la soluzione di $n+1$ sottoproblemi soltanto $OPT(0), \dots, OPT(n)$. 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:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

```
for  $j = 1$  to  $n$ 
```

```
     $M[j] = \text{empty}$  ← array globale
```

```
M-Compute-Opt( $j$ ) {
```

```
    if  $j = 0$  Return 0
```

```
    if ( $M[j]$  is empty)
```

```
         $M[j] = \max(v_j + \text{M-Compute-Opt}(p(j)), \text{M-Compute-Opt}(j-1))$ 
```

```
    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: $O(n \log n)$

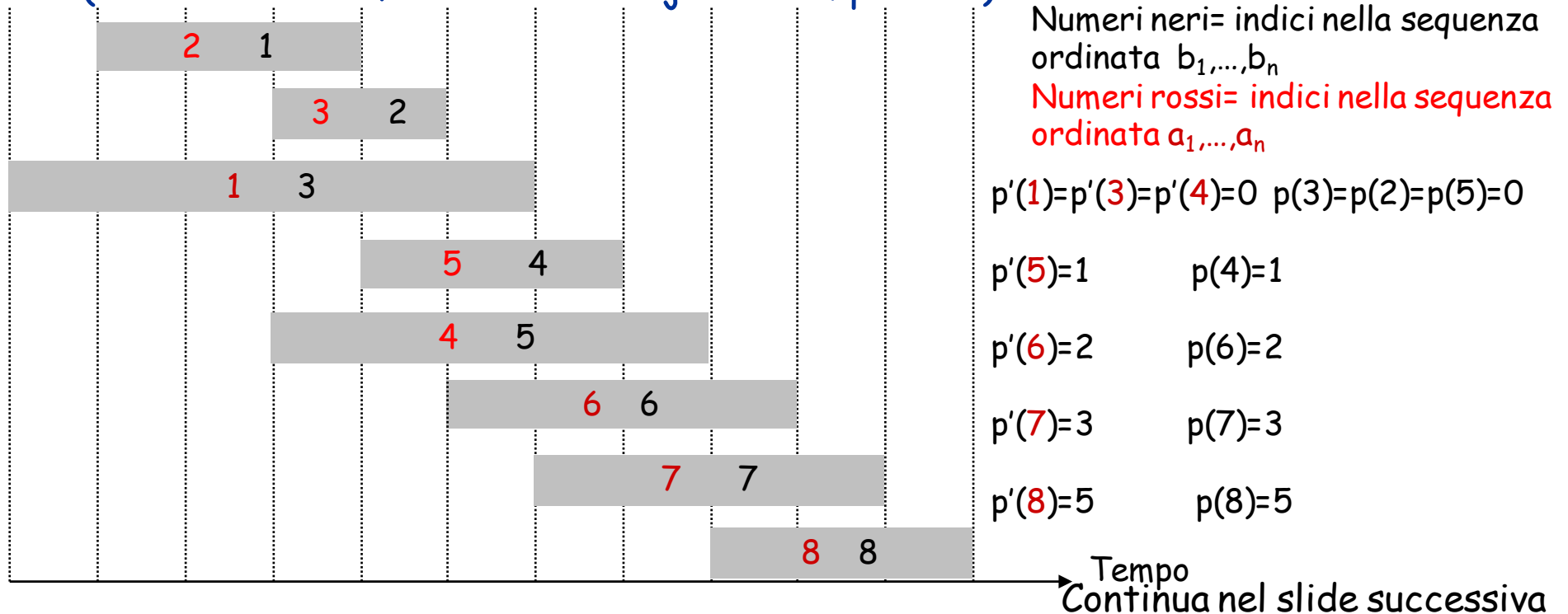
- Ordinamento in base ai tempi di fine: $O(n \log n)$.
- Computazione dei valori $p(\cdot)$: $O(n)$ dopo aver ordinato i job (rispetto ai tempi di inizio e di fine). Siano a_1, \dots, a_n i job ordinati rispetto ai tempi di inizio e b_1, \dots, b_n i job ordinati rispetto ai tempi di fine. (si noti che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

Continua nel slide successiva

Interval Scheduling pesato: Tempo di Esecuzione

Si noti che in $p'(j)$, j è l'indice del job nella sequenza a_1, \dots, a_n . Per ottenere il corrispondente valore $p(r)$ basta sostituire a j l'indice r del job nella sequenza b_1, \dots, b_n .

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



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** $\Phi = \#$ 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 più 2 chiamate ricorsive.
 - quindi per far andare Φ da 0 a j , occorrono al più $2j$ chiamate ricorsive per un tempo totale di $O(j)$
- Il tempo di esecuzione di **M-Compute-Opt** (n) è quindi $O(n)$.
- **N.B.** $O(n)$, una volta ordinati i job in base ai valori di inizio.

Memoization nei linguaggi di programmazione

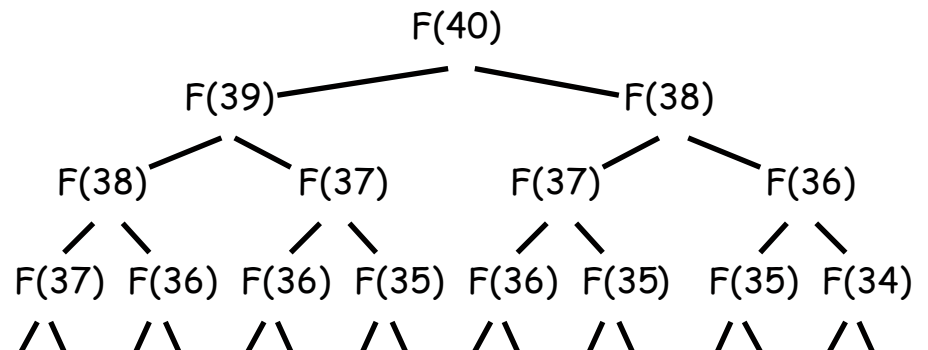
- **Automatica.** Molti linguaggi di programmazione funzionale, quali il Lisp, prevedono un meccanismo per rendere automatica la memoization

```
(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 (vj + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

- **# chiamate ricorsive $\leq n \Rightarrow O(n)$.**

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:  $n, s_1, \dots, s_n, f_1, \dots, f_n, v_1, \dots, v_n$ 
```

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
```

```
Compute  $p(1), p(2), \dots, p(n)$ 
```

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

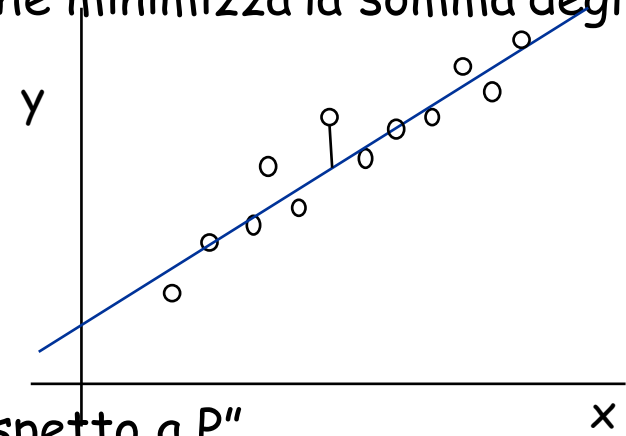
- **Correttezza:** Con l'induzione su j si puo` dimostrare che ogni entrata $M[j]$ contiene il valore $OPT(j)$
- **Tempo di esecuzione:** n iterazioni del for, ognuna della quali richiede tempo $O(1) \rightarrow$ tempo totale $O(n)$

Segmented Least Squares

- **Minimi quadrati.**

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

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



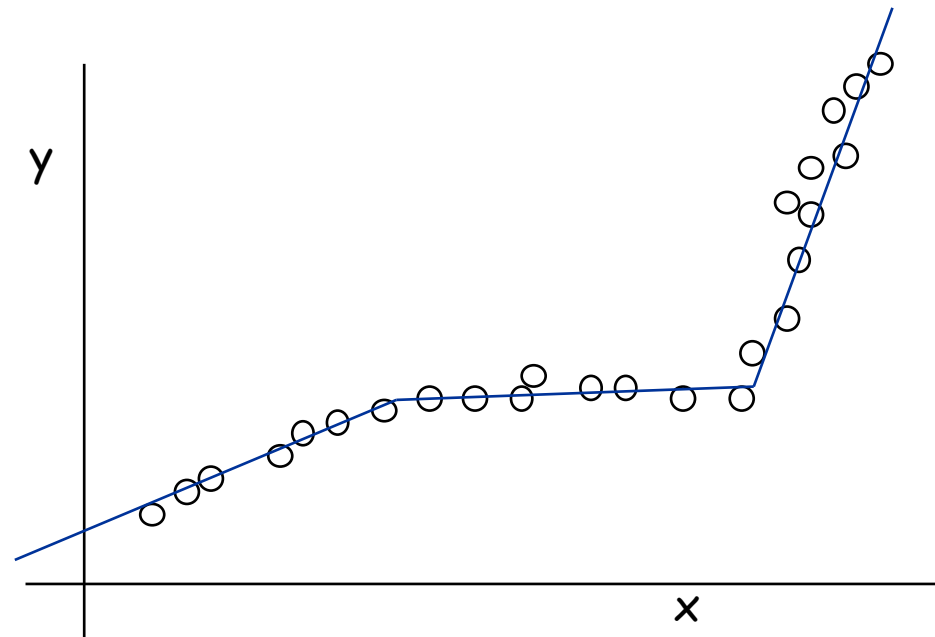
- Chiameremo questa quantità "Errore di L rispetto a P "
- Chiameremo "Errore minimo per P ", il minimo valore di $Error(L,P)$ su tutte le possibili linee L

- **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

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

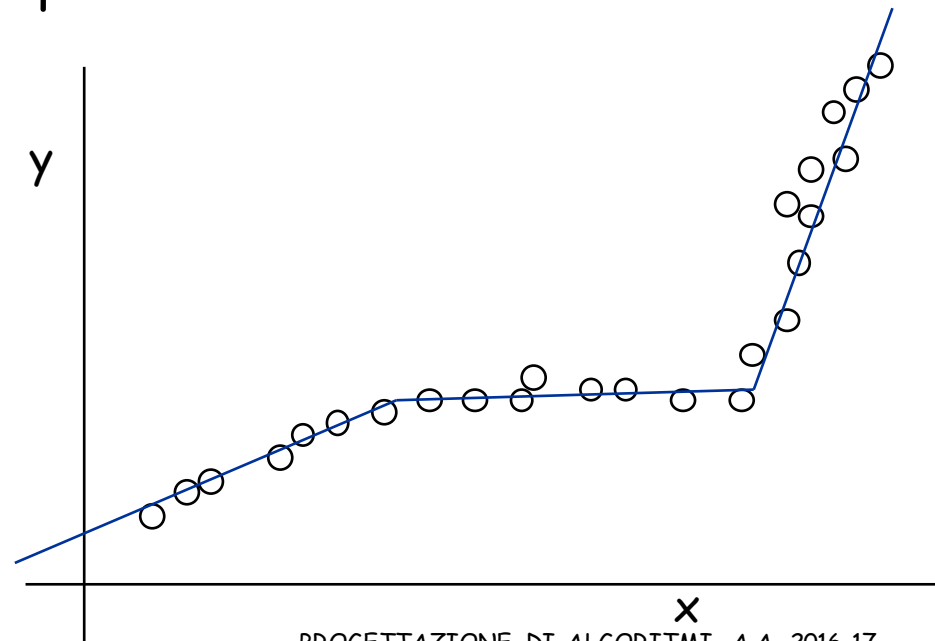
Segmented Least Squares

- **Esempio:** i punti in figura non possono essere ben approssimati usando un'unica linea. Se però 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 $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$ con $x_1 < x_2 < \dots < x_n$, vogliamo partizionare P in un certo numero m di sottoinsiemi P_1, P_2, \dots, 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, \dots, L_m **ottime** rispettivamente per P_1, P_2, \dots, P_m minimizzi la **somma delle 2 seguenti quantita**:
 - La somma degli m errori (minimi) di L_1, L_2, \dots, L_m per P_1, P_2, \dots, P_m , rispettivamente

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

- Il numero m di linee (pesato per una certa costante)

- **Penalita'**: $E + C m$, per una certa costante $C > 0$.

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, \dots, p_j .
- $e(i, j)$ = minimo errore per l'insieme di punti $\{p_i, p_{i+1}, \dots, p_j\}$.

• Per computare $OPT(j)$, osserviamo che

- se l'ultimo segmento nella partizione di $\{p_1, p_2, \dots, p_j\}$ e` costituito dai punti p_i, p_{i+1}, \dots, 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 $e(i, j) + C + OPT(i-1)$

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

Segmented Least Squares: Algorithm

```
INPUT:  $n, p_1, \dots, p_N, c$ 

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  $p_i, \dots, p_j$ 

  for  $j = 1$  to  $n$ 
     $M[j] = \min_{1 \leq i \leq j} (e_{ij} + C + M[i-1])$ 

  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 $\{p_i, \dots, p_j\}$ and the result of
 Find-Segments($i - 1$)

 Endif
