Ancora su scheduling di intervalli pesati

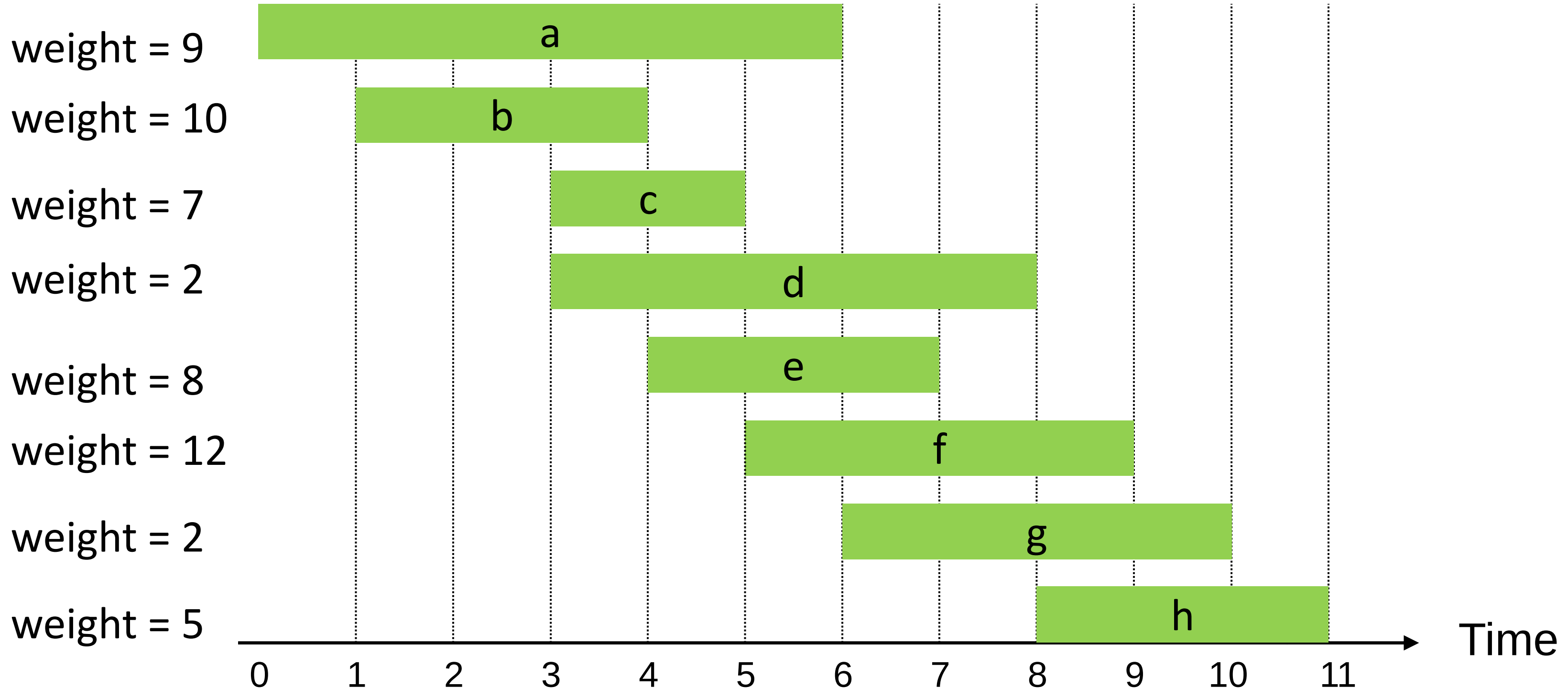
Esercitazione

12 aprile 2023

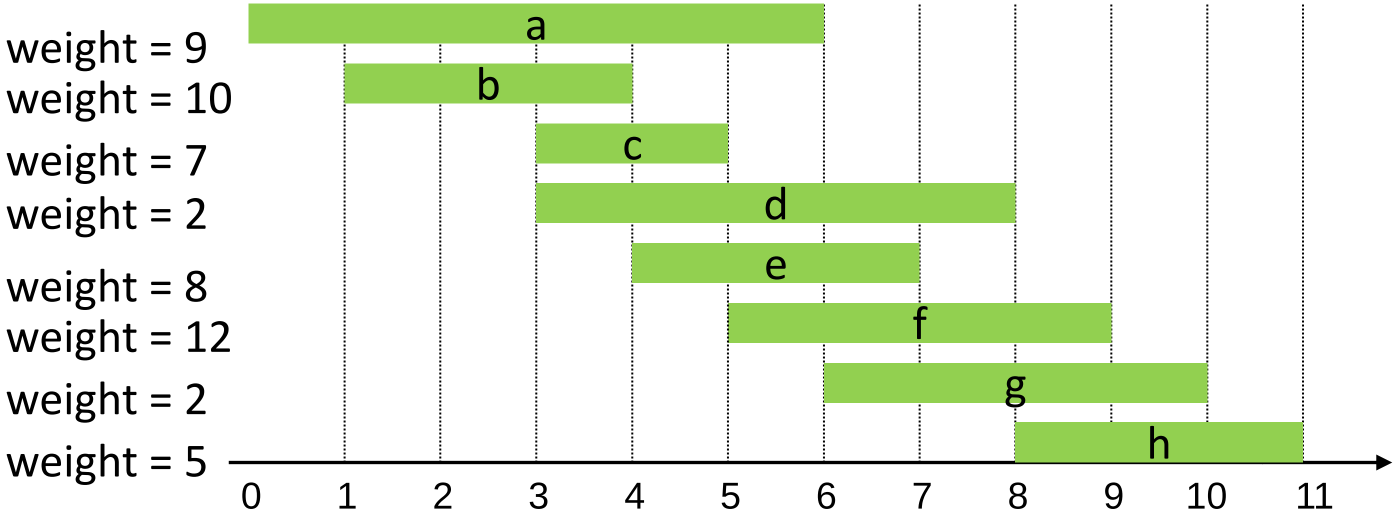
Weighted Interval Scheduling (WIS)

Weighted interval scheduling problem.

- Job j starts at s_j, finishes at f_j, and has weight or value v_j .
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.



Qualche soluzione

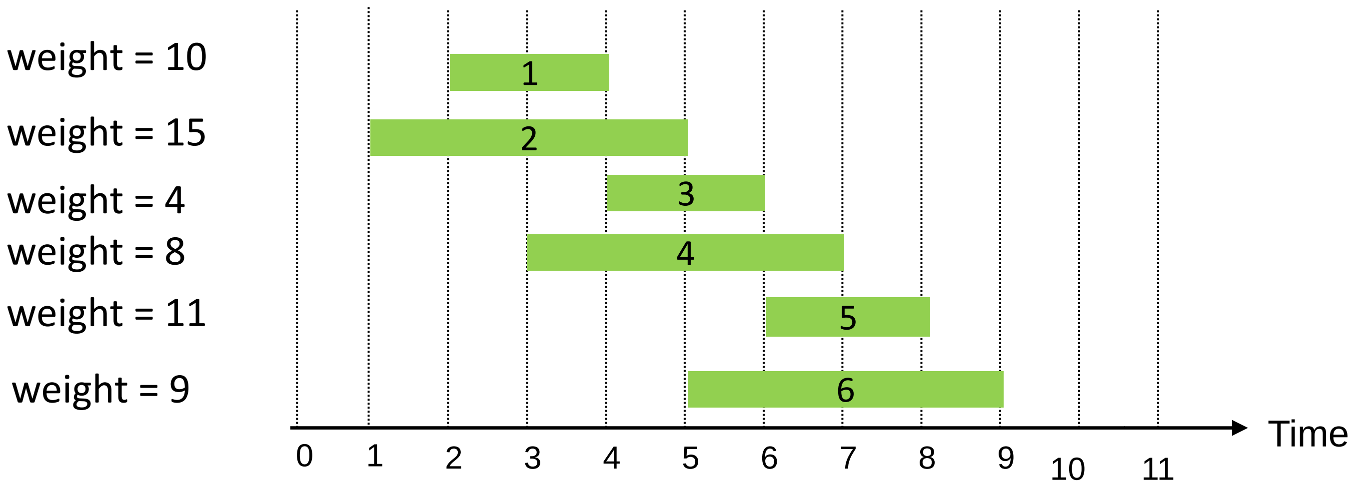


S_1={a, g} di peso 9+2=11 (comincio dal prim in ordine di start)

S_2={c, h} di peso 7+5=12 (comincio dal più piccolo)

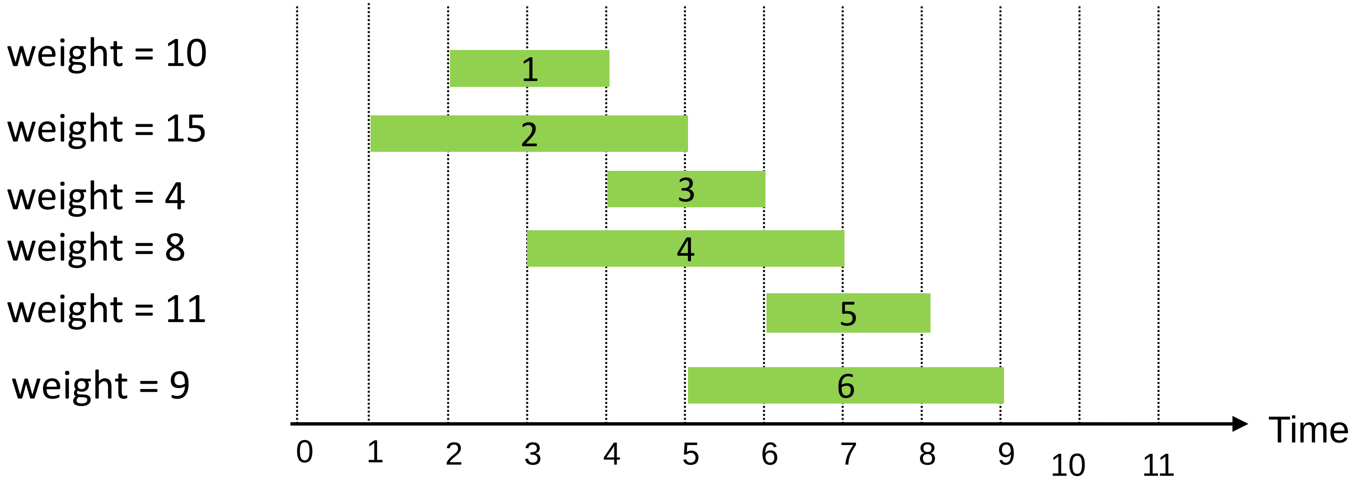
S_3={f, b} di peso 12+10=22 (comincio dal peso massimo)

S_4={b, e, h} di peso 10+8+5=23 (comincio da quello che finisce per primo)



Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}	{2}	15
{1,2,3,4,5}	{2,5}	26
{1,2,3,4,5,6}		

{2,6} o {2,5}? max{15+9, 26} =26



Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}	{2}	15
{1,2,3,4,5}	{2,5}	26
{1,2,3,4,5,6}	{2,5}	26

In generale:
come possiamo ottenere il valore ottimo per {1, 2, …, i, i+1}, supponendo di conoscere i valori ottimi per i problemi {1, …, j} più piccoli?
Considero i+1 e vedo cosa conviene:

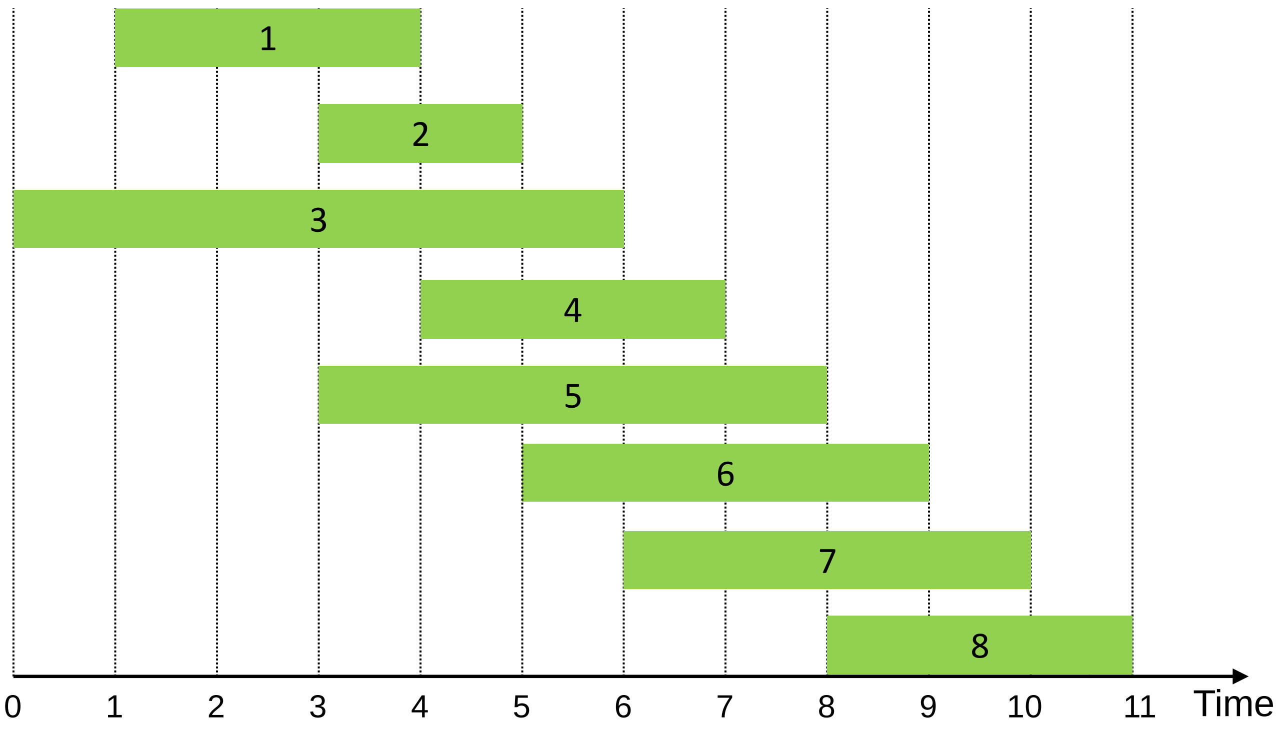
- aggiungere i+1 a una soluzione ottimale per {1, …,k} compatibile
- tralasciare i+1 e prendere una soluzione ottimale per {1, …, i}

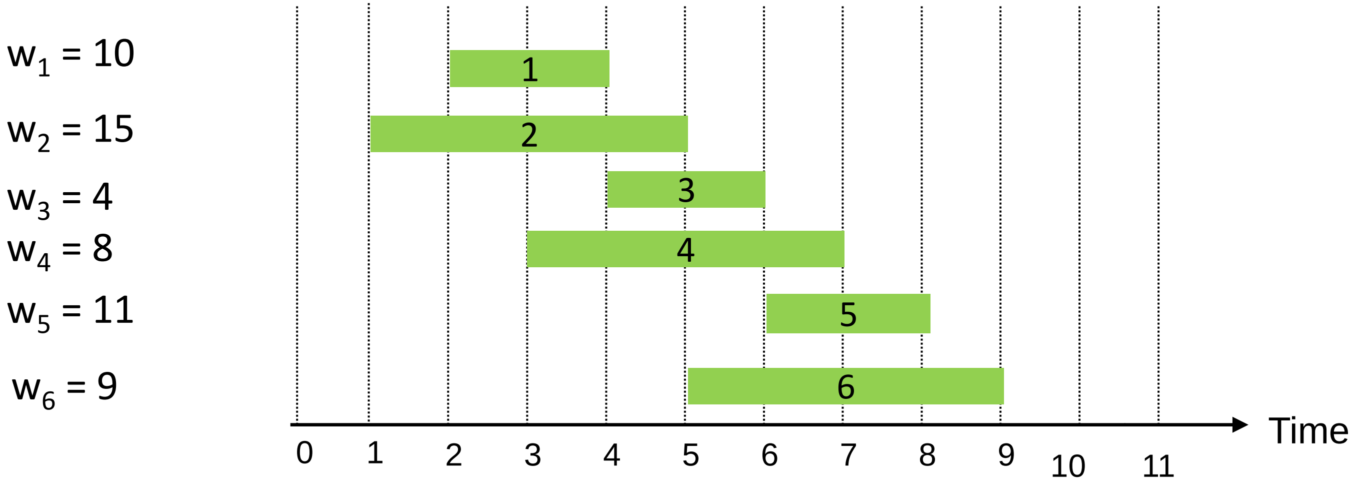
Weighted Interval Scheduling

Notation. Label jobs by finishing time: $f_1 \leq f_2 \leq . . . \leq f_n$.

Def. p(j) = largest index i < j such that job i is compatible with j.

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





Problema	Soluzione ottimale	Valore
{}	{}	0 = OPT(0)
{1}	{1}	10 = OPT(1)
{1,2}	{2}	15 = OPT(2)
{1,2,3}	{2}	15 = OPT(3)
{1,2,3,4}	{2}	15 = OPT(4)
{1,2,3,4,5}	{2,5}	26 = OPT(5)
{1,2,3,4,5,6}	{2,5}	26 = OPT(6)

$$OPT(2) = \max\{15+0, 10\} = \max\{w_2 + OPT(0), OPT(1)\}$$

$$OPT(3) = \max\{4+10, 15\} = \max\{w_3 + OPT(1), OPT(2)\}$$

$$OPT(4) = \max\{8+0, 15\} = \max\{w_4 + OPT(0), OPT(3)\}$$

$$OPT(5) = \max\{11+15, 15\} = \max\{w_5 + OPT(3), OPT(4)\}$$

$$OPT(6) = \max\{9+15, 26\} = \max\{w_6 + OPT(2), OPT(5)\}$$

Dynamic Programming: Binary Choice

Notation. **OPT(j)** = value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

Case 1: OPT selects job j.

can't use incompatible jobs { p(j) + 1, p(j) + 2, ..., j - 1 }

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

optimal substructure

Case 2: OPT does not select job j.

must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

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

Programmazione dinamica: caratteristiche

1. La soluzione al problema originale si può ottenere da soluzioni a sottoproblemi
2. Esiste una relazione di ricorrenza per la funzione che dà il valore ottimo ad un sottoproblema
3. I valori ottimi ai sottoproblemi sono calcolati una sola volta e via via memorizzati in una tabella

   Due implementazioni possibili:

   - Con annotazione (*memoized*) o *top-down*
   - Iterativa o *bottom-up*

Weighted Interval Scheduling: Recursive algorithm

Recursive algorithm.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.

Compute p(1), p(2), …, p(n)

Compute-Opt(n)


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

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

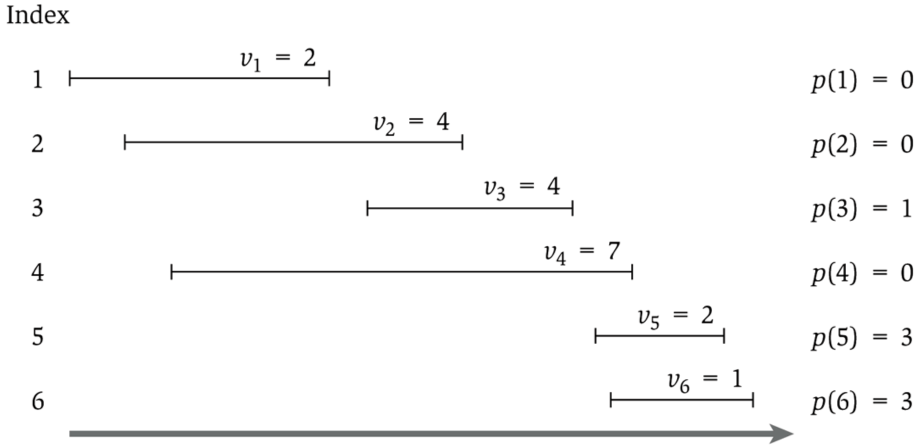


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

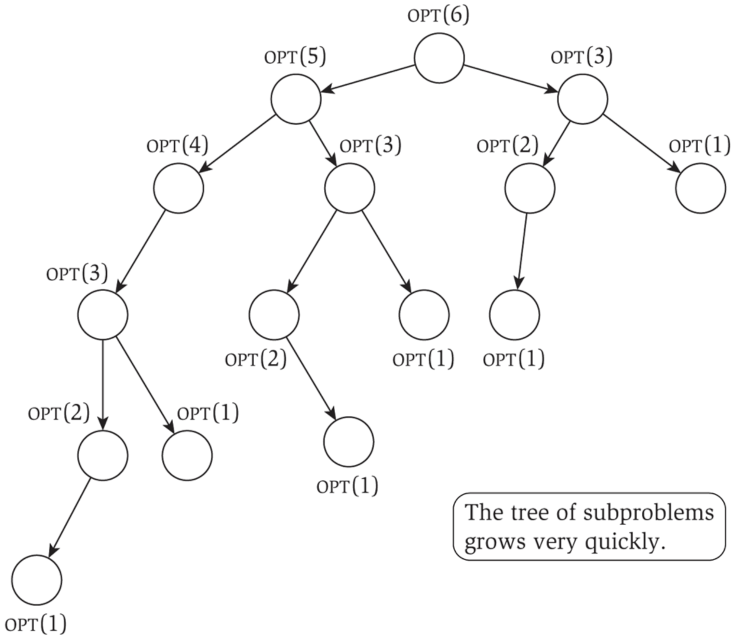
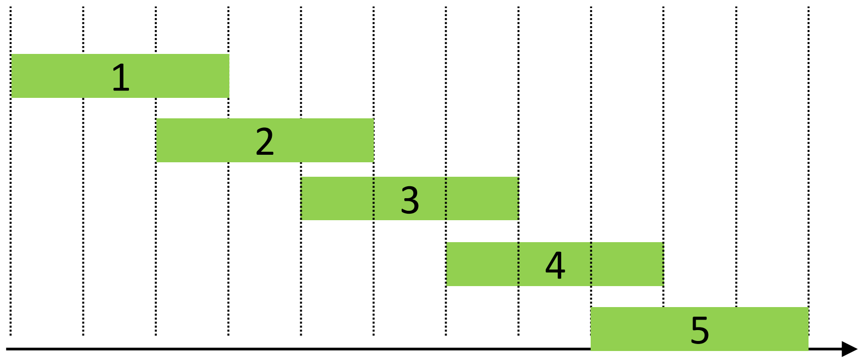


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

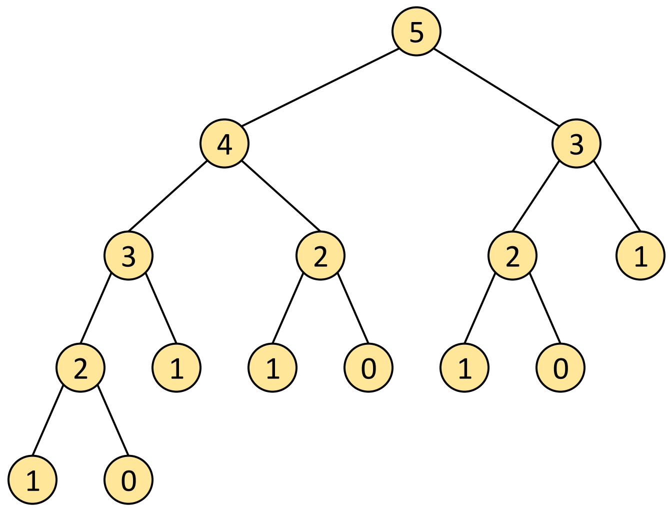
Weighted Interval Scheduling: Recursive algorithm

Observation. Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

Ex. Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.



p(1) = 0, p(j) = j-2



$T(n) = \Omega(\phi^n)$: too much!

Sottoproblemi

- Ci sono molti sottoproblemi ripetuti
- I sottoproblemi distinti sono pochi, sono gli n+1 sottoproblemi, i cui valori ottimi sono:

  OPT(0), OPT(1), … , OPT(n)

La programmazione dinamica può migliorare l'efficienza!

Uso una tabella M[0..n].

In M[i] inserisco OPT(i) appena calcolato.

Weighted Interval Scheduling: Memoization

Memoization. Store results of each sub-problem in a cache; lookup as needed.

```
Input: n, s_1,…,s_n , f_1,…,f_n , v_1,…,v_n

Sort jobs by finish times so that f_1 ≤ f_2 ≤ ... ≤ f_n.
Compute p(1), p(2), …, p(n)

for j = 1 to n
   M[j] = empty
M[0] = 0
M-Compute-Opt(n)

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

Weighted Interval Scheduling: Bottom-Up

Bottom-up dynamic programming. Unwind recursion.

```
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])
}
```

Weighted Interval Scheduling: Running Time

Claim. Iterative version of algorithm takes O(n log n) time.

Sort by finish time: O(n log n).

Computing p(·): O(n) after sorting by start time (exercise).

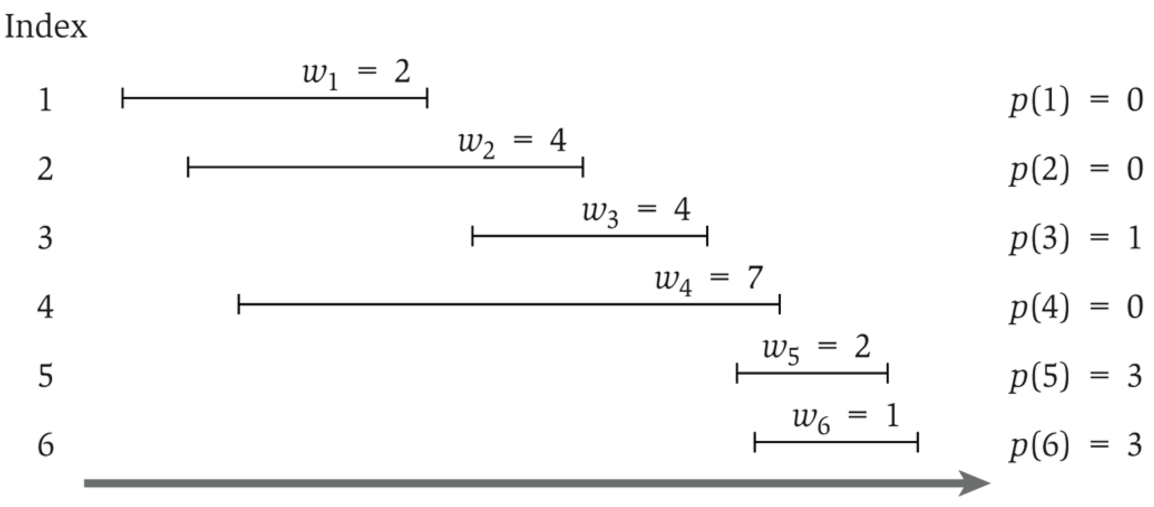
`Iterative-Compute-Opt(j)`: O(n) since the `for` loop repeats n times an operation of constant time

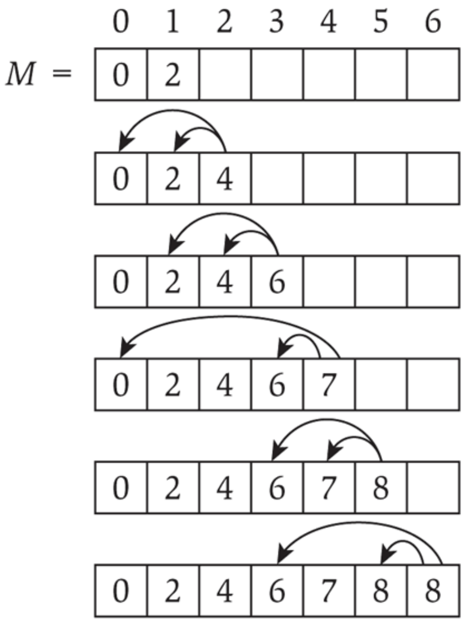
Claim. Also Memoized version of algorithm takes O(n log n) time.

Remark. O(n) if jobs are pre-sorted by start and finish times.

Spazio di memoria utilizzato dato dalle dimensioni di M: $S(n)=\Theta(n)$

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





(b)

M[1] = max (2+M[0], M[0]) = max (2+0, 0) = 2
M[2] = max (4+M[0], M[1]) = max (4+0, 2) = 4
M[3] = max (4+M[1], M[2]) = max (4+2, 4) = 6
M[4] = max (7+M[0], M[3]) = max (7+0, 6) = 7
M[5] = max (2+M[3], M[4]) = max (2+6, 7) = 8
M[6] = max (1+M[3], M[5]) = max (1+6, 8) = 8

Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value.

What if we want the solution itself (the set of intervals)?

A. Do some post-processing.

```
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)
}
```

# of recursive calls $\leq$ n $\Rightarrow$ O(n).

Esempio del calcolo di una soluzione

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

M[1] = max (2+M[0], M[0]) = max (2+0, 0) = 2
M[2] = max (4+M[0], M[1]) = max (4+0, 2) = 4
M[3] = max (4+M[1], M[2]) = max (4+2, 4) = 6
M[4] = max (7+M[0], M[3]) = max (7+0, 6) = 7
M[5] = max (2+M[3], M[4]) = max (2+6, 7) = 8
M[6] = max (1+M[3], M[5]) = max (1+6, 8) = 8

	0	1	2	3	4	5	6
$M =$	0	2	4	6	7	8	8

M[6]=M[5]: 6 non appartiene a OPT
M[5]=v_5+M[3]: OPT contiene 5 e una soluzione ottimale al problema per {1,2,3}
M[3]=v_3+M[1]: OPT contiene 5, 3 e una soluzione ottimale al problema per {1}
M[1]=v_1+M[0]: OPT contiene 5, 3 e 1 (e una soluzione ottimale al problema vuoto)

Soluzione = {5, 3, 1}

Valore = 2+4+2 = 8

Esercitazione

Tipologie di esercizi

1. Formalizzazione problema computazionale
2. Notazioni asintotiche:
   a) Via definizione (c, n_0)
   b) Applicando le proprietà
   c) Sequenze di funzioni da ordinare
   d) Confronto tempo di esecuzione di algoritmi
3. Calcolo del tempo di esecuzione di algoritmi (senza chiamate ricorsive)
4. Scrivere la relazione di ricorrenza per il tempo di esecuzione di algoritmi ricorsivi
5. Risolvere relazioni di ricorrenza
6. Tecnica del Divide et Impera
7. Tecnica della Programmazione Dinamica

Esercizi svolti in classe

Conta 1 (D&I)

Quesito 2 (18 punti)

a) Descrivere un algoritmo efficiente basato sul paradigma **divide et impera** che dato un vettore **ordinato** A[1..n] di interi strettamente positivi (cioè per ogni $1 \leq i \leq n$, $A[i] \geq 1$), restituisca il numero di occorrenze di 1 nel vettore A. Commentare il funzionamento dell'algoritmo.

b) Sia $T(n)$ il tempo di esecuzione dell'algoritmo proposto al punto precedente. Scrivere la relazione di ricorrenza soddisfatta da $T(n)$. Non è necessario mostrarne la soluzione.

Nota: Può essere utile sapere che esiste un algoritmo che risolve il problema in tempo $O(\log n)$.

Esercizi da svolgere

(possibilmente sulla piattaforma)

(Soluzione relazione di ricorrenza 3)

La soluzione della relazione di ricorrenza $\mathbf{T(n) = 2T(n/2) + c}$ è:

(Soluzione relazione di ricorrenza 4)

La soluzione della relazione di ricorrenza $\mathbf{T(n) = 4T(n/2) + n}$ è:

T(n)=T($\sqrt{n}$)+1 con T(2)=1

Relazione di ricorrenza 1 (soluzione)

- Risolvere la seguente relazione di ricorrenza con 2 diversi metodi di risoluzione, nell'ipotesi che n sia una potenza di 2.

$$T(1)=a$$
$$T(n)=T(n/2)+c$$

- Cosa potete dire della soluzione nel caso generale che n non sia necessariamente una potenza di 2?

Relazione di ricorrenza 2 (soluzione)

- Si consideri la seguente relazione di ricorrenza.

$$T(0) = 1$$
$$T(1) = 3$$
$$T(n) = T(n - 2) + n$$

Quanto valgono $T(6)$ e $T(9)$?

- Risolvere la relazione di ricorrenza con tutti i metodi possibili.

Ricerca ternaria (D&I)

- Progettare un algoritmo per la ricerca di un elemento **key** in un array ordinato **A[1..n]**, basato sulla tecnica **Divide-et-impera** che nella prima fase divide l'array in **3** parti «uguali» (le 3 parti differiranno di al più 1 elemento).
- **Scrivere** la relazione di ricorrenza per il tempo di esecuzione dell'algoritmo proposto. Potete supporre che n sia una potenza di 3.
- **Risolvere** la relazione di ricorrenza.
- **Confrontare** il tempo di esecuzione ottenuto con quello della ricerca binaria.

Occorrenze consecutive di 2 (D&I) (dalla piattaforma)

Si scriva lo pseudo-codice di un algoritmo ricorsivo basato sulla tecnica Divide et Impera che prende in input un array di interi positivi e restituisce il massimo numero di occorrenze **consecutive** del numero ‘2’.

Ad esempio, se l'array contiene la sequenza <2 2 3 6 2 2 2 2 3 3> allora l'algoritmo restituisce 4. Occorre specificare l'input e l'output dell’algoritmo.

Programmazione dinamica (pseudocodice)

Si supponga che la soluzione ad un certo problema (a noi ignoto) sia data, per un certo intero n positivo, dal massimo fra i valori OPT(n,R) e OPT(n,B) definiti ricorsivamente come segue (R sta per Rosso e B sta per Blu):

$OPT(1, R) = 2$
$OPT(1, B) = 1$
$OPT(i, R) = OPT(i - 1, B) + 1$, se $i > 1$
$OPT(i, B) = \max \{OPT(i, R) - 1, OPT(i - 1, R)\}$, se $i > 1$

a) Calcolare i valori di OPT(i, R) e OPT(i, B) per ogni i=1, 2, …, 5, organizzandoli in una **tabella**.

b) Scrivere lo pseudocodice di un algoritmo **ricorsivo** per il calcolo della soluzione al problema.

c) Scrivere lo pseudocodice di un algoritmo di **programmazione dinamica** per il calcolo della soluzione al problema. Analizzarne la complessità di **tempo** e di **spazio**, giustificando la risposta.

Esercizio (analisi Fibonacci2)

Nelle slides precedenti è dimostrato che il tempo di esecuzione $T(n)$ dell'algoritmo ricorsivo Fibonacci2(n) è $T(n)=O(2^n)$.

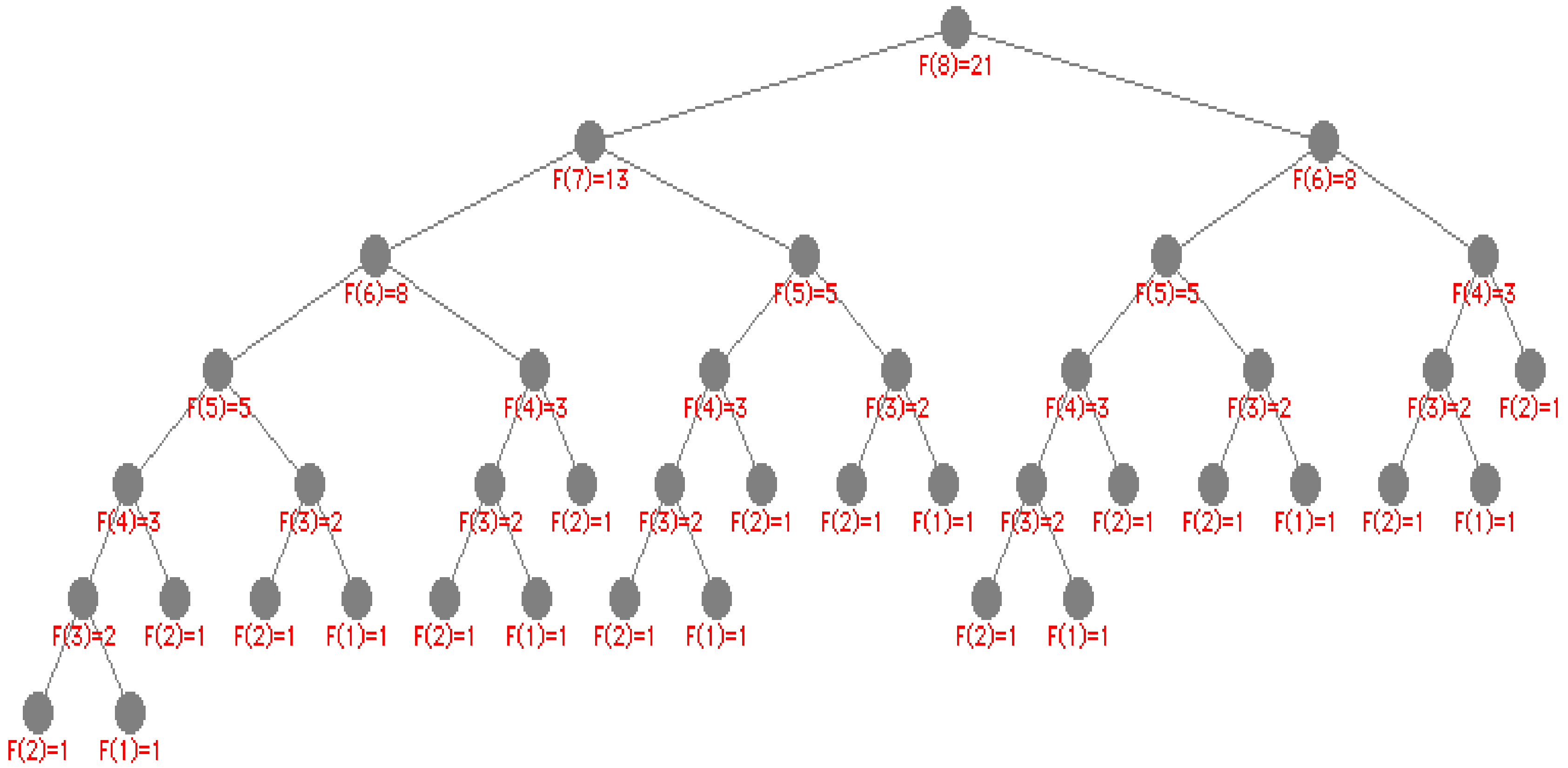
Questa in realtà è soltanto una limitazione superiore. Dimostrare che:

1. $T(n)= \Omega(l(n))$, dove $l(n)$ è il numero di foglie dell'albero della ricorsione per Fibonacci2(n)
2. Per ogni $n \geq 1$, $l(n)=F(n)$ (il numero di foglie è esattamente uguale all'n-esimo numero di Fibonacci), usando l'induzione strutturale.

Un esempio

```
Fibonacci2(n)

If n≤2 then Return 1
Else Return Fibonacci2(n-1)+ Fibonacci2(n-2)
```



Esercizio (Fibonacci con 2 celle)

Fornire una variante dell'algoritmo **`Fibonacci3-iter(n)`**, mostrato nelle slide precedenti, che utilizzi soltanto 2 celle di memoria (anziché n).

Appello 9 luglio 2015

Quesito 2 (24 punti)
Da quando ti sei registrato su Facebook ad oggi, i tuoi amici sono aumentati in maniera vertiginosa. Il primo anno avevi solo 10 amici; il secondo 35; il terzo 100 e nessuno ti elimina mai dagli amici. Hai poi notato che ogni tuo amico, **a partire da** 3 anni dopo averti dato la sua amicizia, ti porta un nuovo amico (spesso è un collega di università/lavoro, fidanzato/a, fratello/a, cugino/a). E tutti i tuoi nuovi amici si aggiungono sempre e solo in questo modo.
Sapresti calcolare quanti diventeranno i tuoi amici nei prossimi anni?

a) Descrivere un algoritmo efficiente per il calcolo del numero dei tuoi amici dopo *n* anni dalla tua registrazione su Facebook, supponendo che aumentino sempre rispettando la regola sopra descritta. E'necessario analizzare la complessità di tempo e di spazio dell'algoritmo proposto.
b) Valutare la crescita del numero di amici rispetto ad *n* (in notazione asintotica).

Appello 9 febbraio 2011

I numeri di Tribonacci sono cosi' definiti:

$R(0) = 0$

$R(1) = 0$

$R(2) = 1$

$R(n) = R(n-1) + R(n-2) + R(n-3)$ se $n \geq 3$.

a) Scrivere lo pseudocodice di un algoritmo di programmazione dinamica per il calcolo dell'n-esimo numero di Tribonacci $R(n)$.

b) Analizzare la complessita' di tempo e di spazio dell'algoritmo proposto.

c) E' possibile realizzare l'algoritmo con spazio $O(1)$? Giustificare la risposta.

Appello 12 settembre 2016

Quesito 1 (24 punti) (*Cappanacci*)

La sequenza dei numeri di Fibonacci k-generalizzati, per un intero k, è definita come segue

$F_{n,k} = 0$ per $n = 0, 1, \ldots, k-2$

$F_{k-1,k} = 1$

$F_{n,k} = F_{n-1,k} + F_{n-2,k} + \ldots + F_{n-k,k}$ per ogni $n \geq k$.

Descrivere ed analizzare un algoritmo di programmazione dinamica che dati due interi, n e k, calcola il numero $F_{n,k}$.

Esempio. Per k=3, i primi numeri di Fibonacci 3-generalizzati sono:

$F_{0,3} = F_{1,3} = 0$, $F_{2,3} = 1$, $F_{3,3} = 1$, $F_{4,3} = 2$, $F_{5,3} = 4$, …