Programmazione dinamica: Selezione di intervalli pesati

5 aprile 2023

Programmazione dinamica: caratteristiche

`Fibonacci3-memo` e `Fibonacci3-iter` sono algoritmi di programmazione dinamica: perché?

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 per un sottoproblema
3. Le soluzioni ai sottoproblemi sono calcolate una sola volta e via via memorizzate in una tabella

Due implementazioni possibili:

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

Programmazione dinamica vs Divide et Impera

Entrambe le tecniche dividono il problema in sottoproblemi: dalle soluzioni dei sottoproblemi è possibile risalire alla soluzione del problema di partenza

Dobbiamo allora considerare la tecnica Divide et Impera superata?

NO: La programmazione dinamica risulta più efficiente quando:

- Ci sono dei sottoproblemi ripetuti
- Ci sono solo un numero polinomiale di sottoproblemi (da potere memorizzare in una tabella)

Per esempio: nel MergeSort non ci sono sottoproblemi ripetuti.

Appello 29 gennaio 2015

Quesito 2 (24 punti)

Dopo la Laurea in Informatica avete aperto **un campo** di calcetto che ha tantissime richieste e siete diventati ricchissimi. Ciò nonostante volete guadagnare sempre di più, per cui avete organizzato una sorta di asta: chiunque volesse affittare il vostro campo (purtroppo è uno solo), oltre ad indicare da che ora a che ora lo vorrebbe utilizzare, deve dire anche quanto sia disposto a pagare. Il vostro problema è quindi scegliere le richieste compatibili per orario, che vi diano il guadagno totale maggiore. Formalizzate il problema reale in un problema computazionale.

6.1 Weighted Interval Scheduling

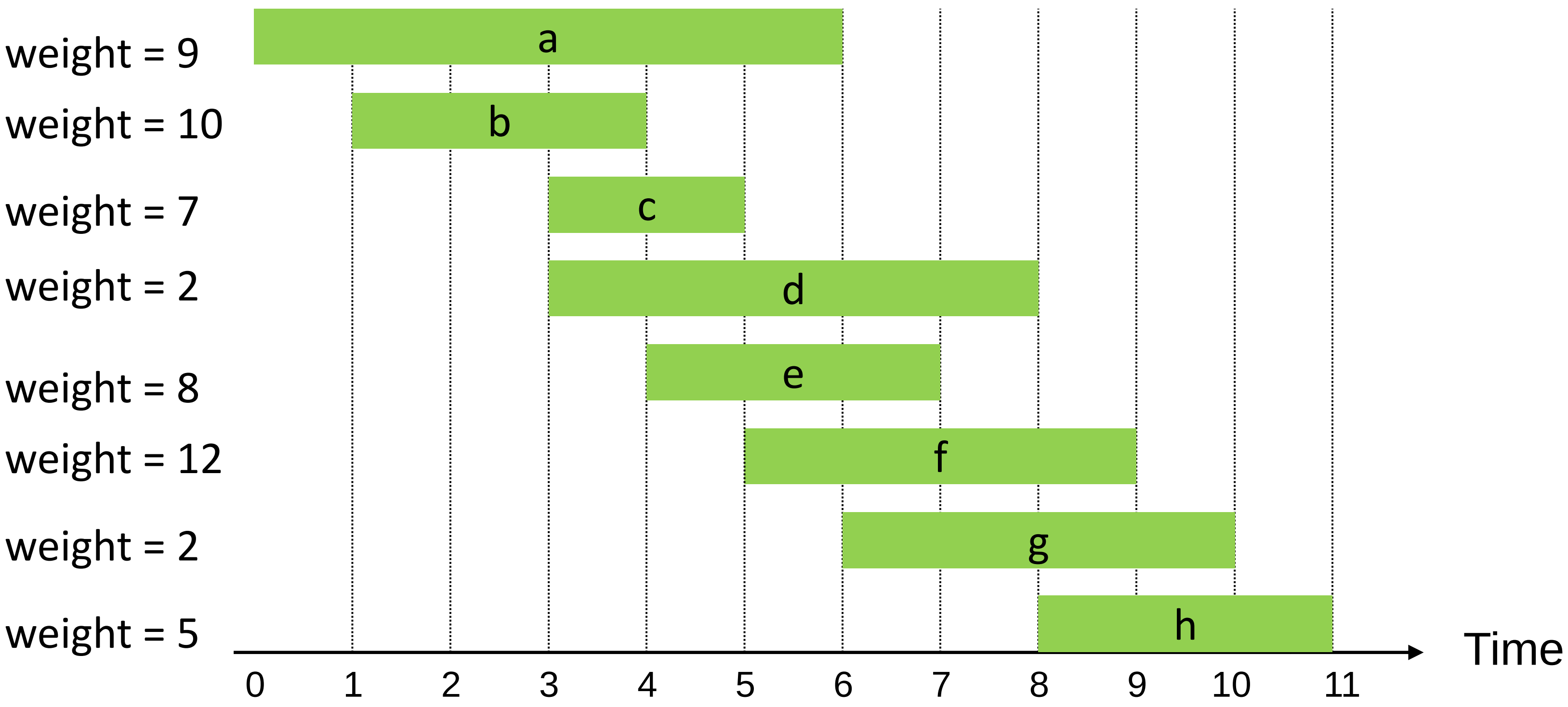
Weighted Interval Scheduling (WIS)

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.



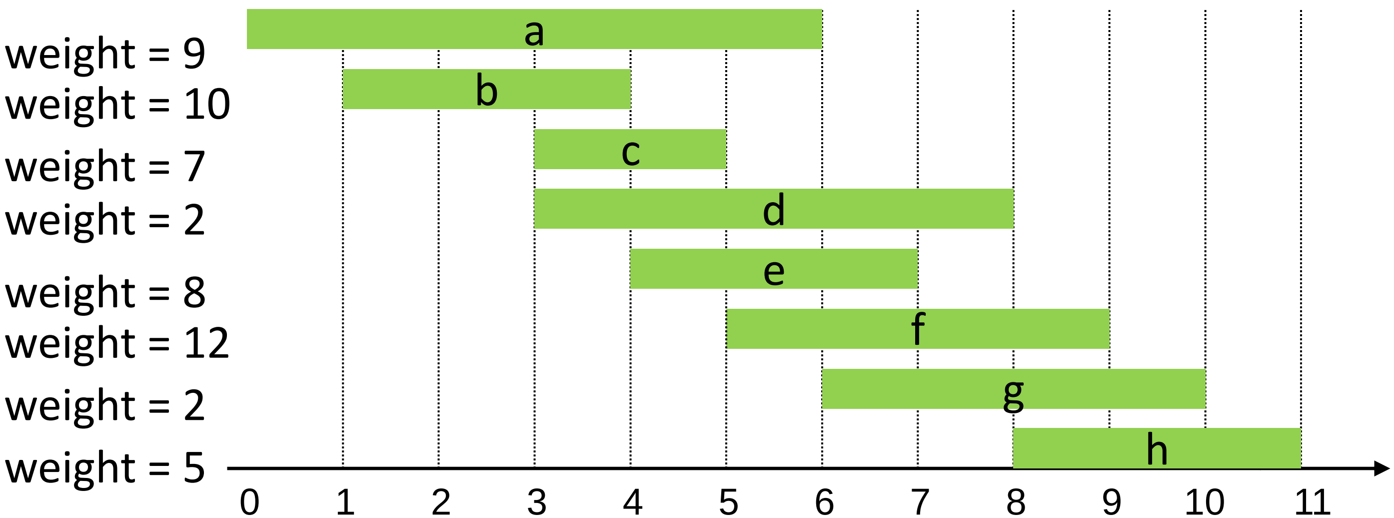
Approccio «intuitivo»

Costruire la soluzione passo passo seguendo un criterio di scelta, di presunta convenienza.

Questo è l’approccio del goloso: la tecnica *greedy*!

Purtroppo, per questo problema, nessun criterio di scelta ci darebbe la soluzione migliore per ogni input!

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)

Come risolverlo?

Si può sempre provare con la **ricerca esaustiva** (brute force, naïf):

- Considero tutti i sottinsiemi di S
- Per ognuno verifico la compatibilità e calcolo il peso
- Restituisco un sottinsieme compatibile di peso massimo

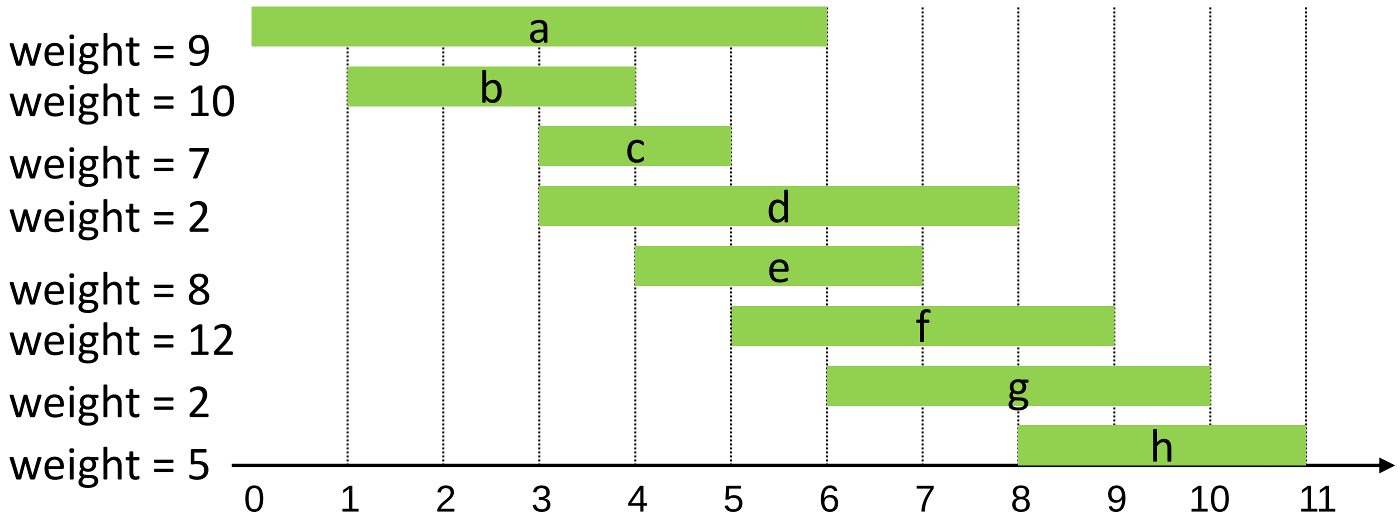
Per piccoli input può andare, ma per input grandi?
Qual è la complessità del tempo di esecuzione al crescere della taglia dell'input?

Purtroppo… il numero di tutti i sottinsiemi di un insieme di n elementi è 2^n

Come risolverlo?

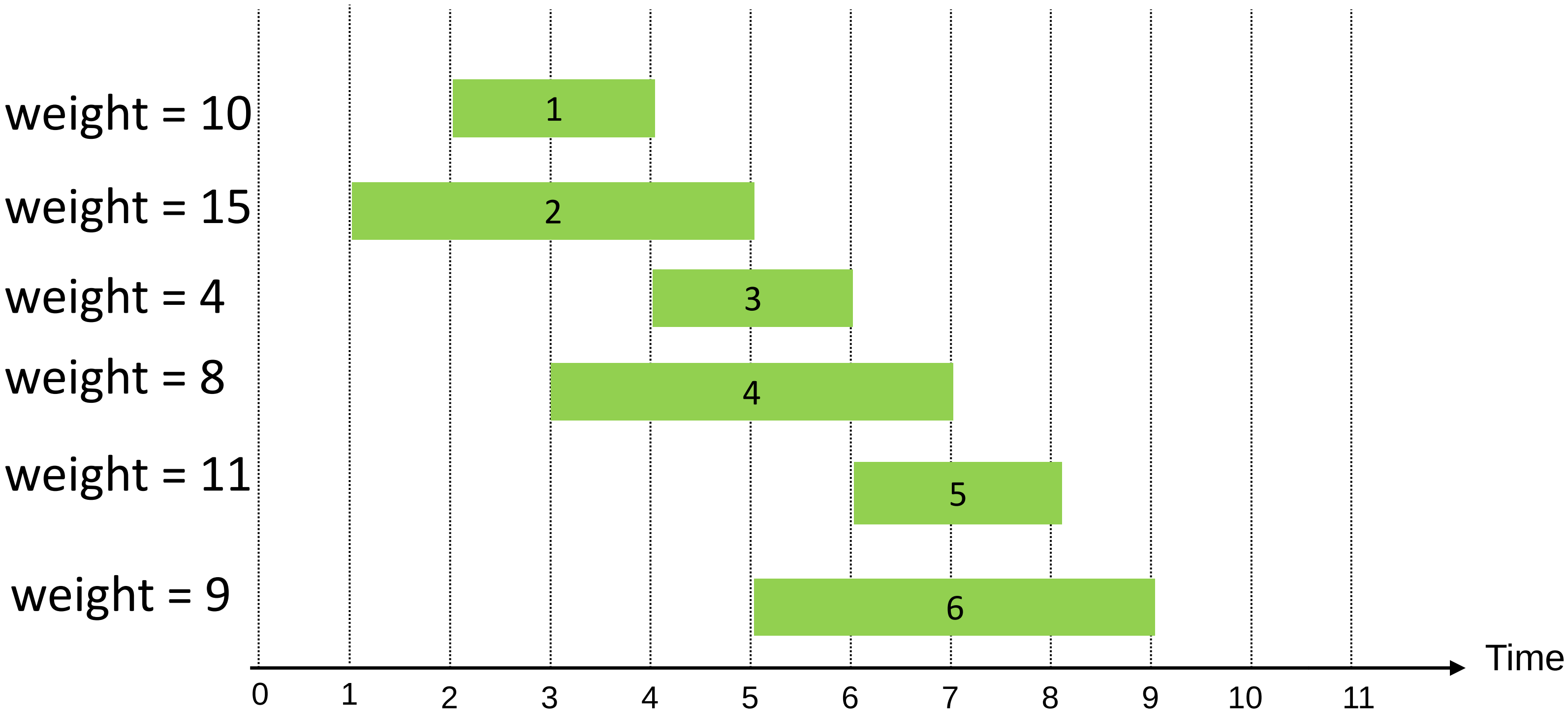
Bisogna cambiare approccio!

Provo con la tecnica Divide et impera:



Divido in due metà; trovo l'ottimo per {a,b,c,d} e l'ottimo per {e,f,g,h}.
Ottengo {b, e, h} che non è ottimale.
E non è detto che le due soluzioni siano compatibili.

Weighted Interval Scheduling

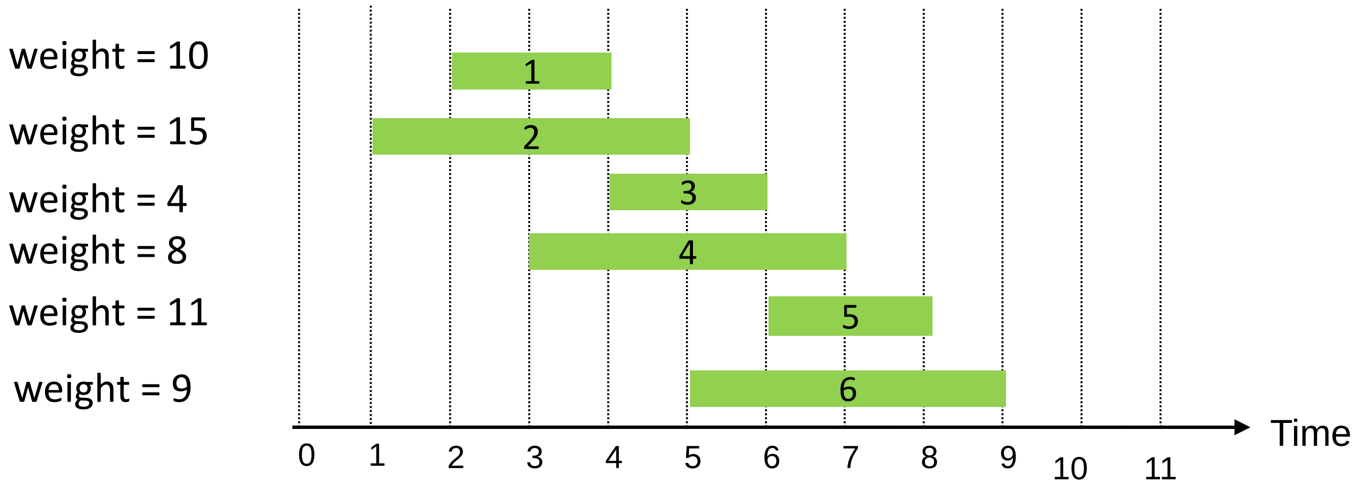


Comincio col considerare il problema per un caso «piccolo».

Se avessi solo l'intervallo 1? soluzione ottimale {1} di peso 10.

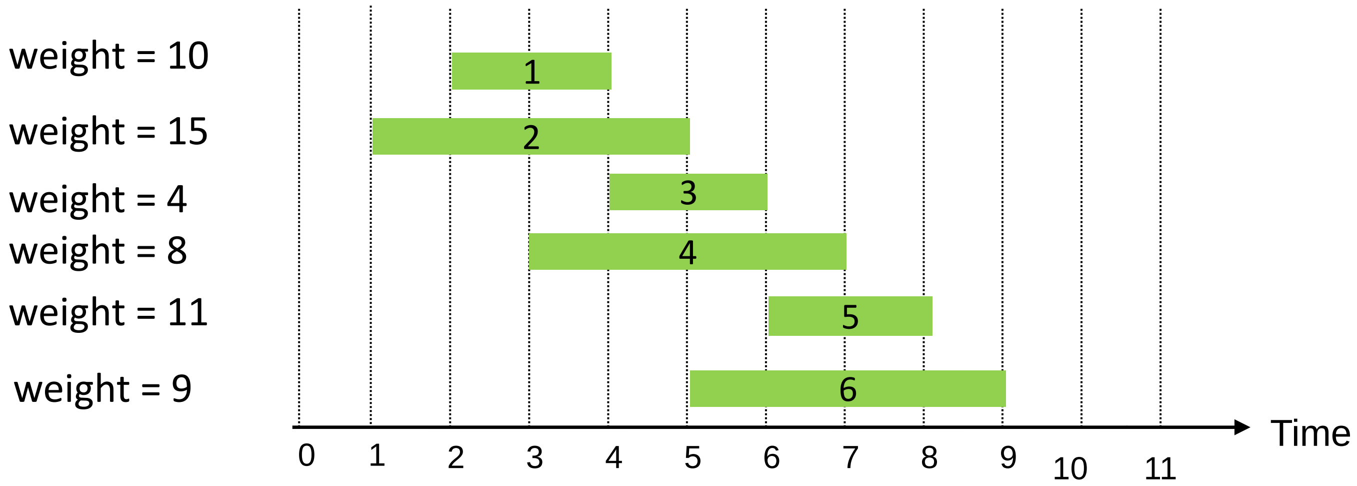
Se avessi il problema per gli intervalli 1 e 2? 2 non lo posso aggiungere a {1}; soluzione ottimale {2} con peso 15.

Se aggiungessi 3? Come posso riutilizzare i valori già calcolati?



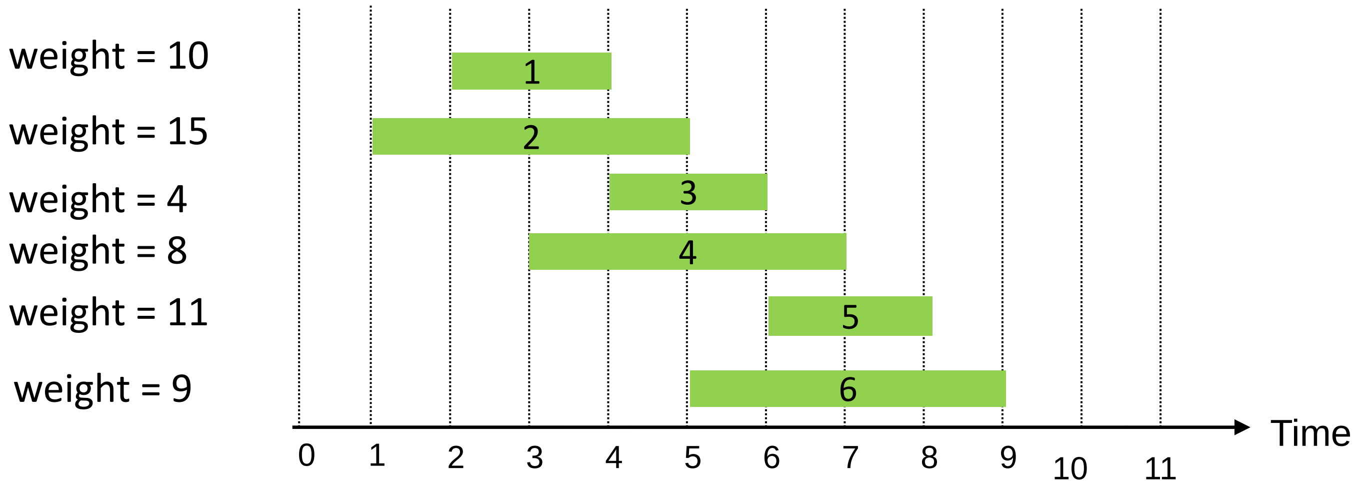
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}		

{1, 3} o {2}? max{10+4, 15} =15



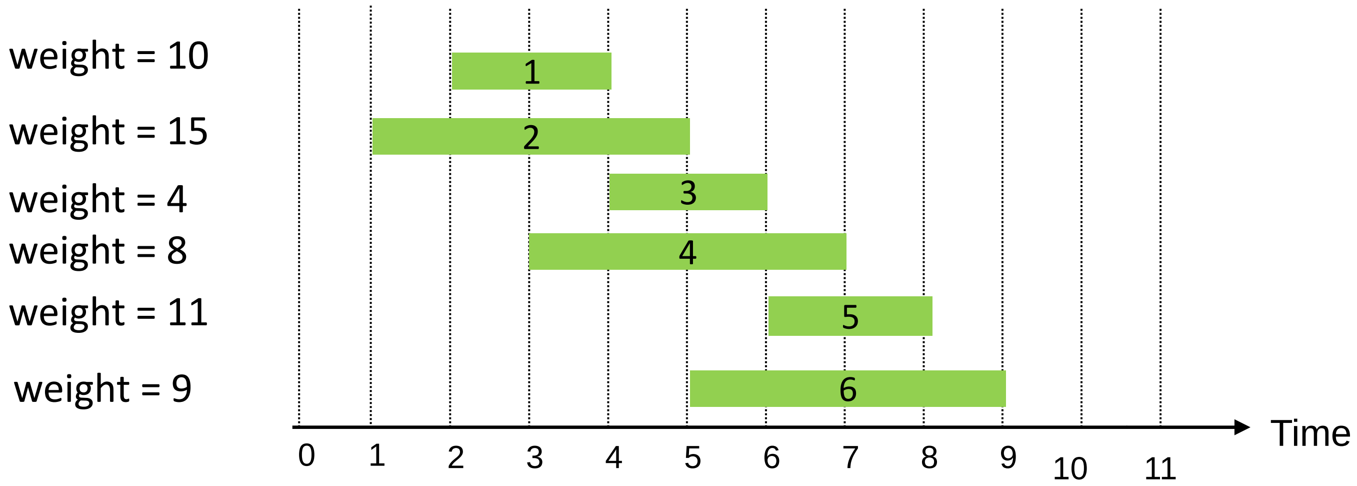
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}		

{1, 3} o {2}? max{10+4, 15} =15



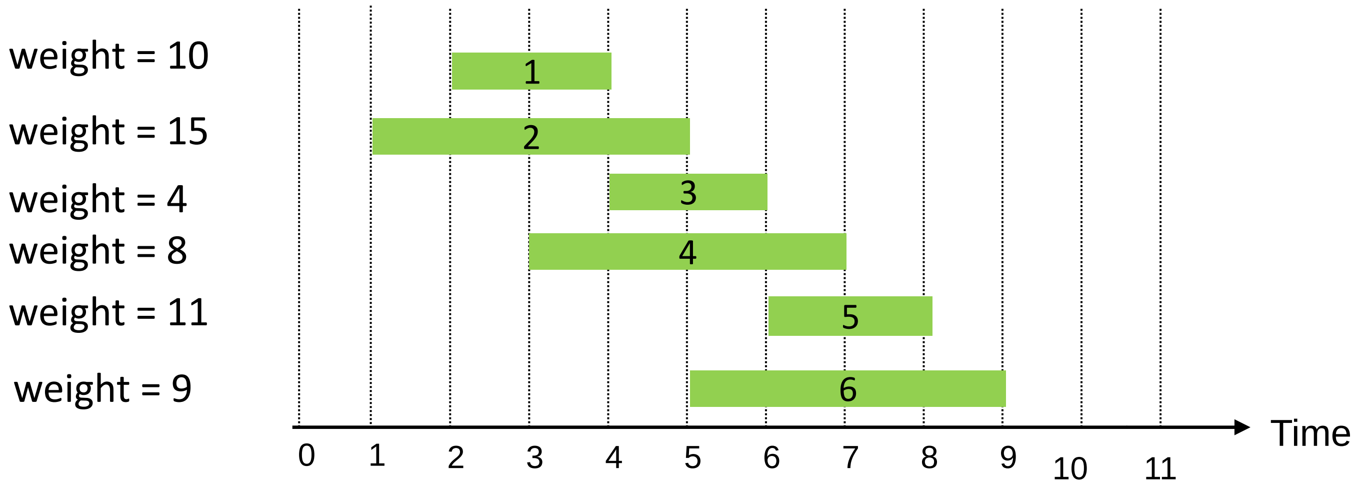
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}		

{4} o {2}? max{8, 15} =15



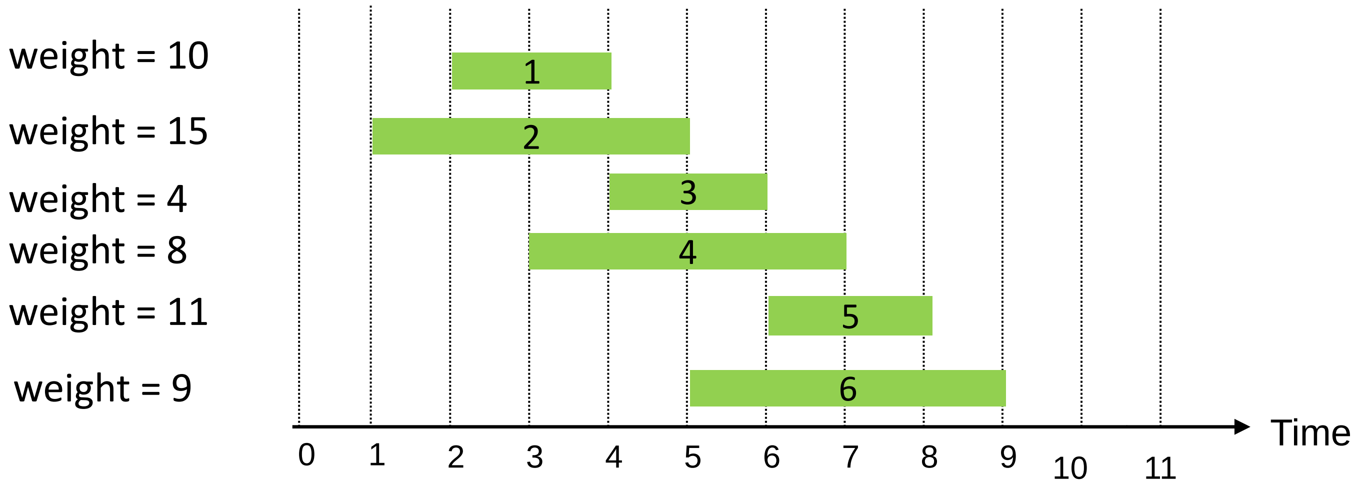
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}	{2}	15
{1,2,3,4,5}		

{4} o {2}? max{8, 15} =15



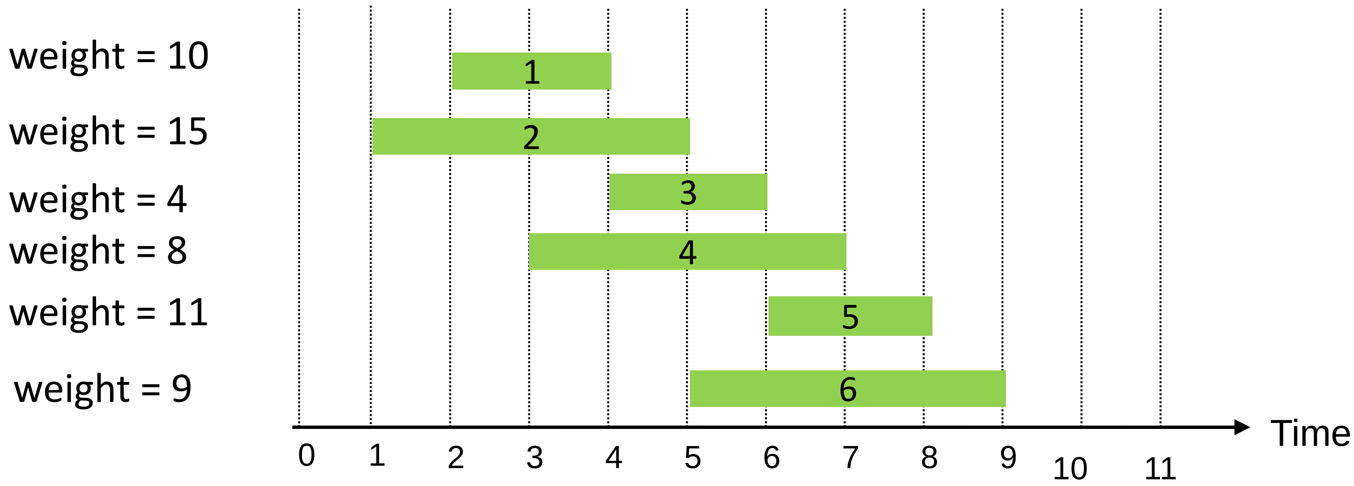
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}	{2}	15
{1,2,3,4,5}		

{2,5} o {2}? max{15+11, 15} =26



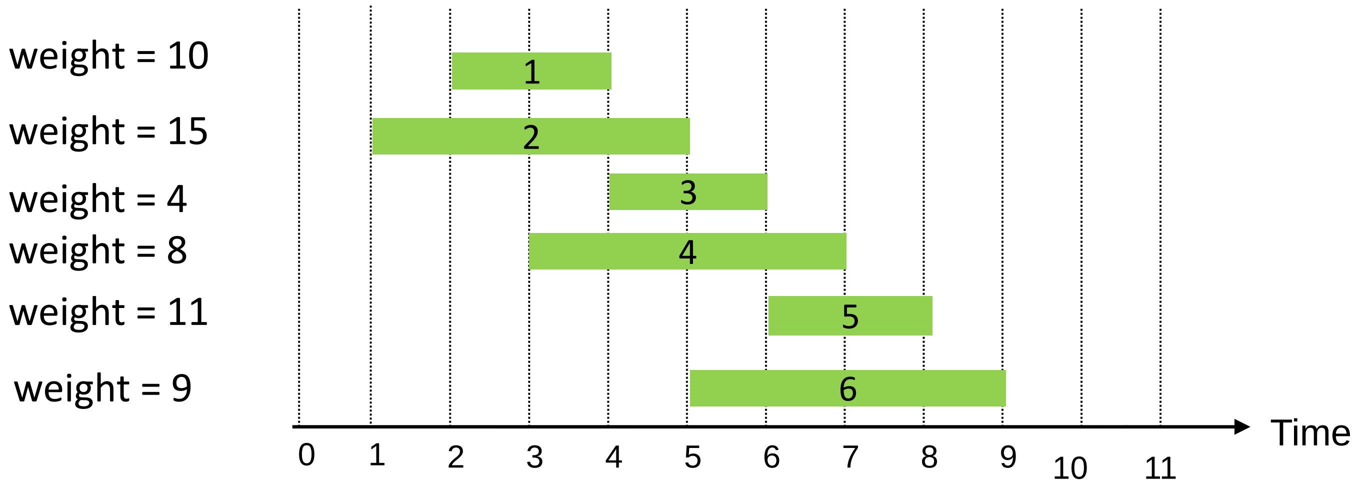
Problema	Soluzione ottimale	Valore
{1}	{1}	10
{1,2}	{2}	15
{1,2,3}	{2}	15
{1,2,3,4}	{2}	15
{1,2,3,4,5}	{2,5}	26
{1,2,3,4,5,6}		

{2,5} o {2}? max{15+11, 15} =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}		

{2,6} 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:

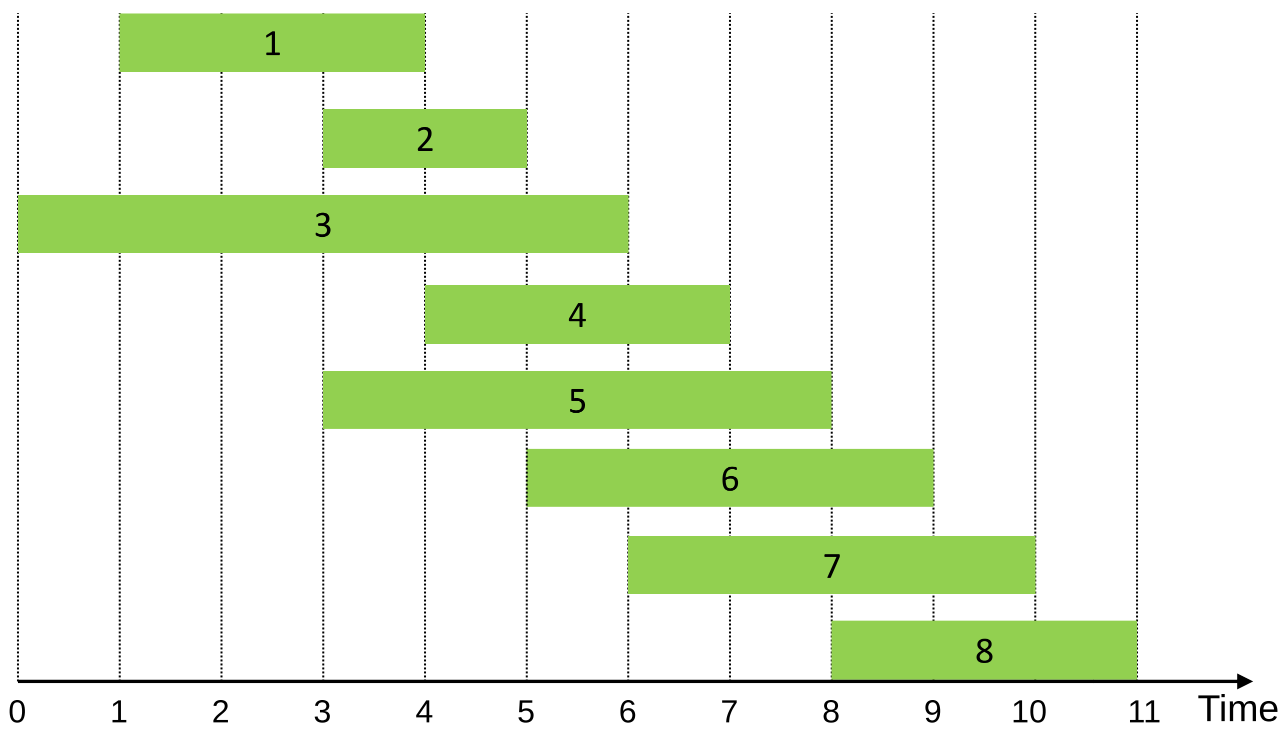
- aggiungere i+1 a una soluzione ottimale per {1, …,k} compatibile
- tralasciare i+1 e prendere una soluzione ottimale per {1, …, i}

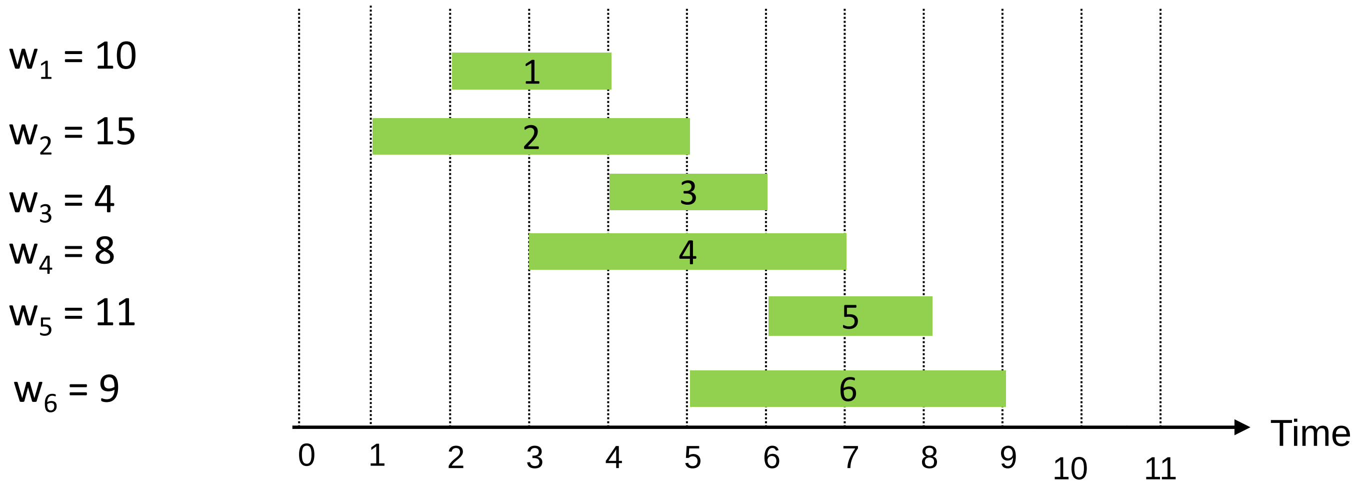
Weighted Interval Scheduling

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, 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(n)


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

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

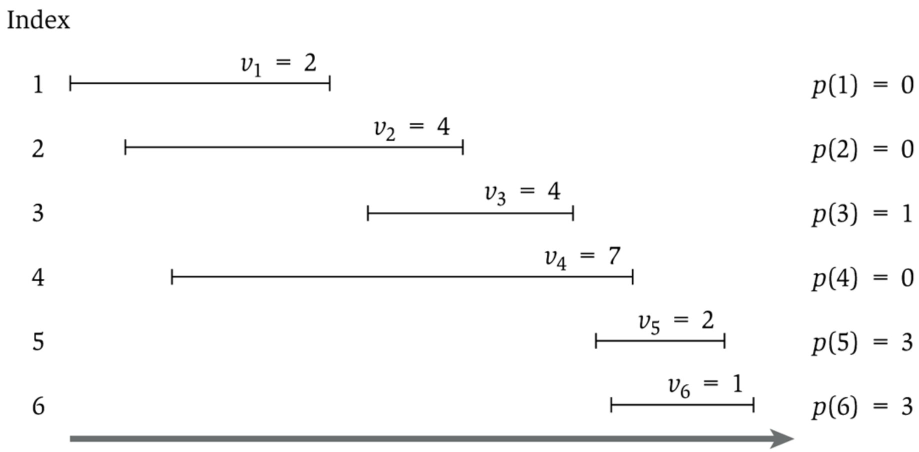


Figure 6.2 An instance of weighted interval scheduling with the functions $p(j)$ defined for each interval j.

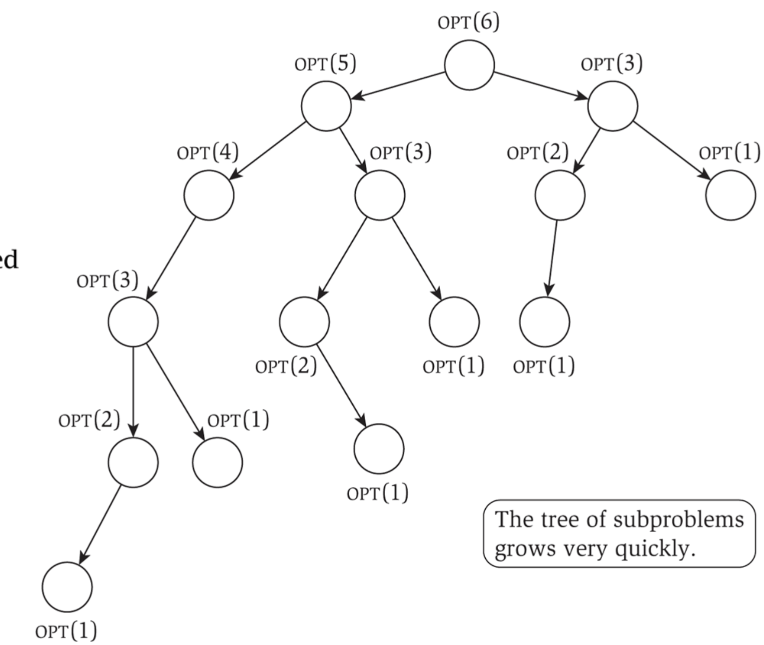
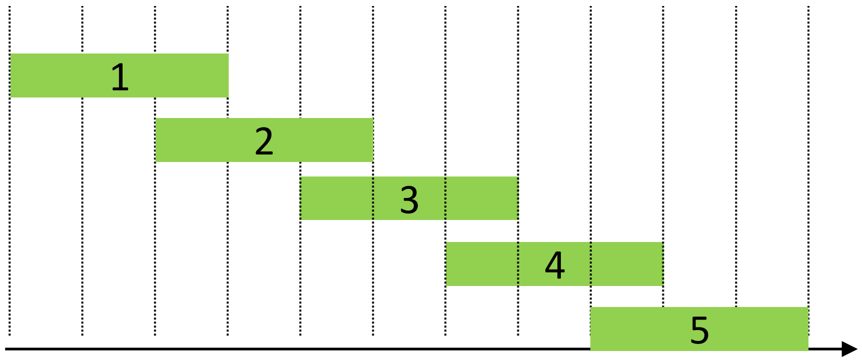


Figure 6.3 The tree of subproblems called by `Compute-Opt` on the problem instance of Figure 6.2.

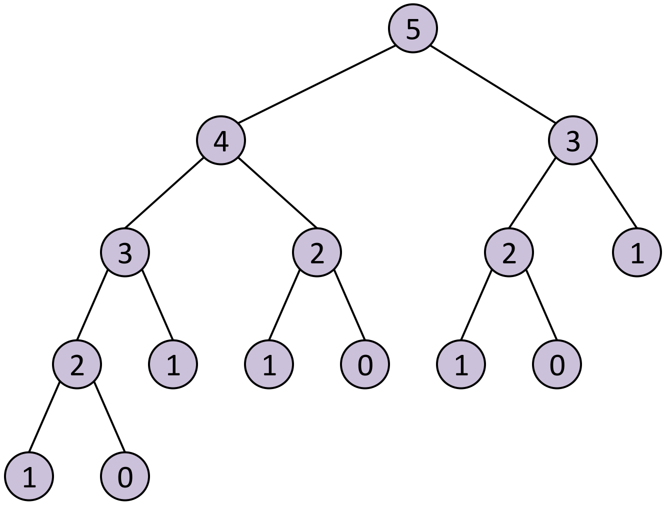
Weighted Interval Scheduling: Recursive algorithm

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: Running Time

Claim. Memoized 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).
- `M-Compute-Opt(j)`: each invocation takes O(1) time and either
  - (i) returns an existing value `M[j]`
  - (ii) fills in one new entry `M[j]` and makes two recursive calls
- Progress measure Φ = # nonempty entries of `M[]`.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases Φ by 1 $\Rightarrow$ at most 2n recursive calls.
- Overall running time of `M-Compute-Opt(n)` is O(n).

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

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).

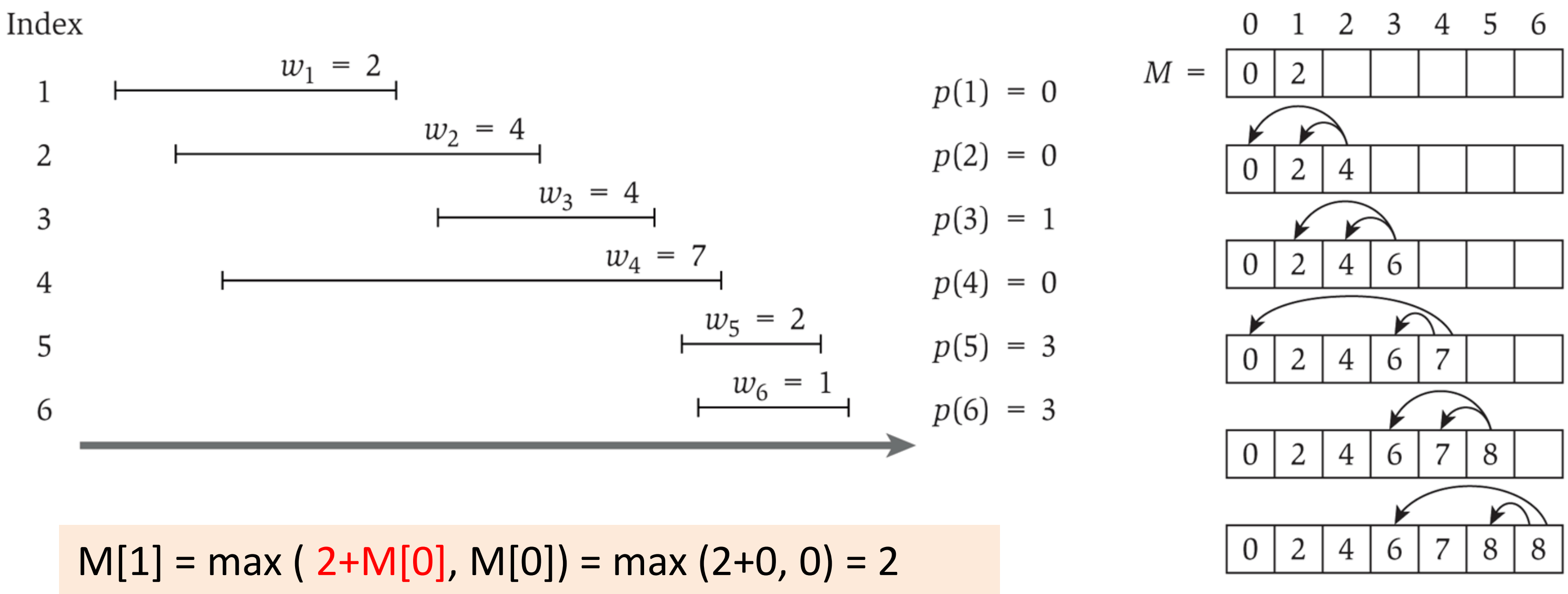
`Iterative-Compute-Opt(j)`: O(n) since the `for` loop repeats n times an operation of constant time

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 (v_j + 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

Buona Pasqua!