Algoritmi per Bioinformatica

Tecniche di Progettazione

Programmazione Dinamica
Pt. 1

Modelli di Progettazione di Algoritmi

- Greedy
  - Costruisce la soluzione in maniera incrementale
  - Usa scelte localmente ottime
- Divide et Impera (Divide and Conquer)
  - Suddivide in sottoproblemi (indipendenti)
  - Combina le soluzioni ai sottoproblemi nella soluzione al problema originario
- Programmazione Dinamica
  - Analizza lo spazio delle soluzioni risolvendo i sottoproblemi in maniera "incrementale"
  - Sfrutta sottoproblemi condivisi / ripetuti

Programmazione Dinamica

- Troviamo una soluzione ricorsiva al problema
- Identifichiamo l'esistenza di un numero "piccolo" di sottoproblemi
- Risolviamo "bottom-up"

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n) = \begin{cases} 1 & n = 0,1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
  if n = 1 OR n = 0
      return 1
  return Fib(n-1) + Fib(n-2)
}
```


Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
  if n = 1 OR n = 0
      return 1
  return Fib(n-1) + Fib(n-2)
}
```

Analizziamo questa procedura

T(n) : *running time di Fib(n)*

$$T(n)=\begin{cases}1 & n=0,1\\ T(n-1)+T(n-2)+1 & n>1\end{cases}$$

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
  if n = 1 OR n = 0
      return 1
  return Fib(n-1) + Fib(n-2)
}
```

Analizziamo questa procedura

T(n) : *running time di Fib(n)*

$$T(n)=\begin{cases}1 & n=0,1\\ T(n-1)+T(n-2)+1 & n>1\end{cases} \Rightarrow T(n)=2F(n)-1=2^{1+0.694n}$$

Esercizio

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
 if n = 1 OR n = 0
     return 1
 return Fib(n-1) + Fib(n-2)
}
```

Analizziamo questa procedura

T(n) : *running time di Fib(n)*

$$T(n)=\begin{cases}1 & n=0,1\\ T(n-1)+T(n-2)+1 & n>1\end{cases} \Rightarrow T(n)=2F(n)-1=2^{1+0.694n}$$

Esercizio

Troppo!!!

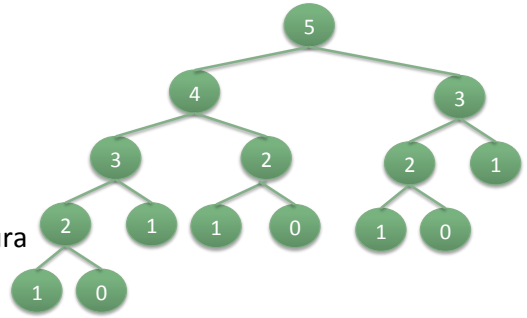
Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
 if n = 1 OR n = 0
     return 1
 return Fib(n-1) + Fib(n-2)
}
```



Analizziamo questa procedura

T(n) : *running time di Fib(n)*

$$T(n)=\begin{cases}1 & n=0,1\\ T(n-1)+T(n-2)+1 & n>1\end{cases} \Rightarrow T(n)=2F(n)-1=2^{1+0.694n}$$

Esercizio

Troppo!!!

Fibonacci e Programmazione Dinamica

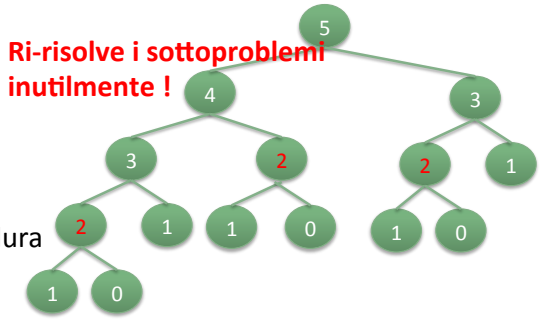
- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Fib(n)
{
 if n = 1 OR n = 0
     return 1
 return Fib(n-1) + Fib(n-2)
}
```

Ri-risolve i sottoproblemi inutilmente !



Analizziamo questa procedura

T(n) : *running time di Fib(n)*

$$T(n)=\begin{cases}1 & n=0,1\\ T(n-1)+T(n-2)+1 & n>1\end{cases} \Rightarrow T(n)=2F(n)-1=2^{1+0.694n}$$

Esercizio

Troppo!!!

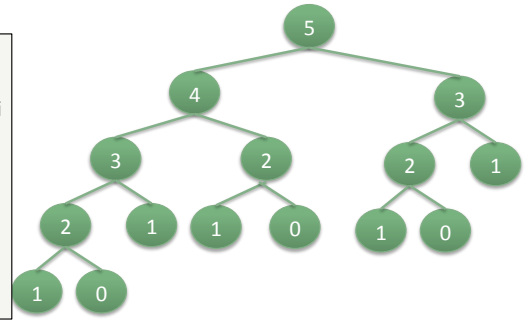
Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Set M[0] = M[1] = 1
Set M[2] = M[3] = ... = M[n] =  -1
 // M[] memorizza i valori già calcolati

Procedure MFib(n)
{
 if M[n] = -1   // non ancora calcolato
     M[n] = MFib(n-1)+ MFib(n-2)
 return M[n]
}
```



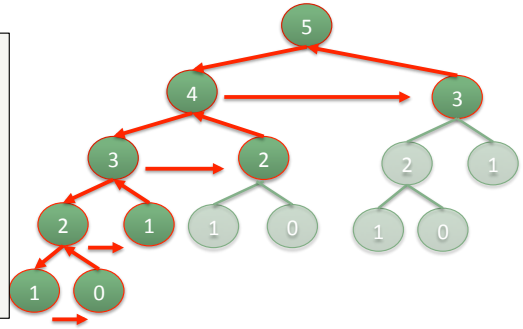
Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Set M[0] = M[1] = 1
Set M[2] = M[3] = … = M[n] = -1
 // M[] memorizza i valori già calcolati

Procedure MFib(n)
{
 if M[n] = -1  // non ancora calcolato
      M[n] = MFib(n-1)+ MFib(n-2)
 return M[n]
}
```



i	0	1	2	3	4	5
M[i]	1	1	2	3	-1	-1

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Set M[0] = M[1] = 1
Set M[2] = M[3] = … = M[n] = -1
 // M[] memorizza i valori già calcolati

Procedure MFib(n)
{
 if M[n] = -1  // non ancora calcolato
      M[n] = MFib(n-1)+ MFib(n-2)
 return M[n]
}
```

La complessità di MFib è O(n)

Consideriamo le chiamate ad MFib()
Ogni nuova (coppia di) chiamate corrisponde a riempire un nuovo valore di M[]
ce ne sono n, quindi ≤ 2n chiamate!

i	0	1	2	3	4	5
M[i]	1	1	2	3	5	8

Fibonacci e Programmazione Dinamica

- Esempio: Calcolo dei numeri di Fibonacci

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

n	0	1	2	3	4	5	6	7	...
F(n)	1	1	2	3	5	8	13	21	...

```
Set M[0] = M[1] = 1
Set M[2] = M[3] = … = M[n] = -1
 // M[] memorizza i valori già calcolati

Procedure MFib(n)
{
 if M[n] = -1  // non ancora calcolato
      M[n] = MFib(n-1)+ MFib(n-2)
 return M[n]
}
```

Memoization

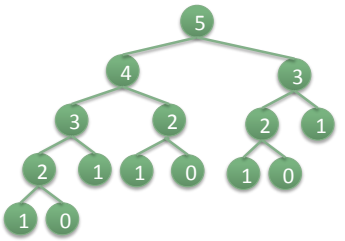
Quando risolvi un nuovo sottoproblema memorizzane la soluzione
Per ogni sottoproblema controlla prima, O(1), se è stato già risolto, altrimenti risolvilo *ex novo*

Fibonacci: passi della Prog. Dinamica

- Soluzione ricorsiva

$$F(n)=\begin{cases}1 & n=0,1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

- L'algoritmo ricorsivo "naturale" risolve lo stesso sottoproblema più volte
- Usa una tabella per ricordare i sottoproblemi già calcolati



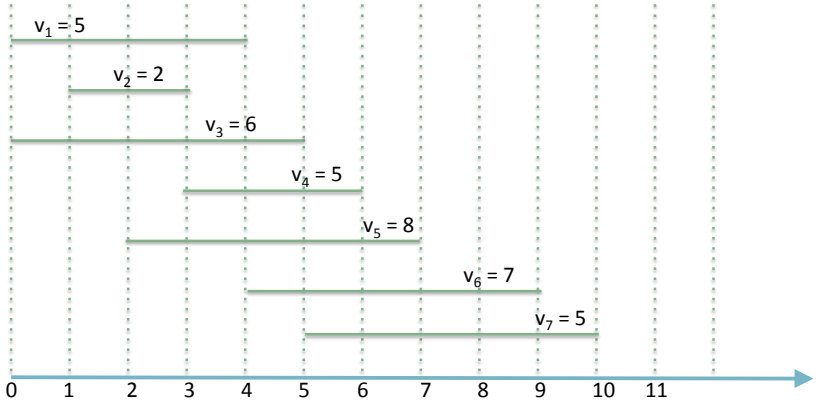

```
Set M[0] = M[1] = 1
Set M[2] = M[3] = … = M[n] = -1
 // M[] memorizza i valori già calcolati

Procedure MFib(n){
 if M[n] = -1  // non ancora calcolato
      M[n] = MFib(n-1)+ MFib(n-2)
 return M[n]
}
```

Programmazione Dinamica e Ottimizzazione

- Problema di Ottimizzazione
  - Ogni istanza ammette più soluzioni
  - Ogni soluzione ha associato un costo/valore
  - Scopo: identificare una soluzione di
    - Costo minimo/Valore massimo

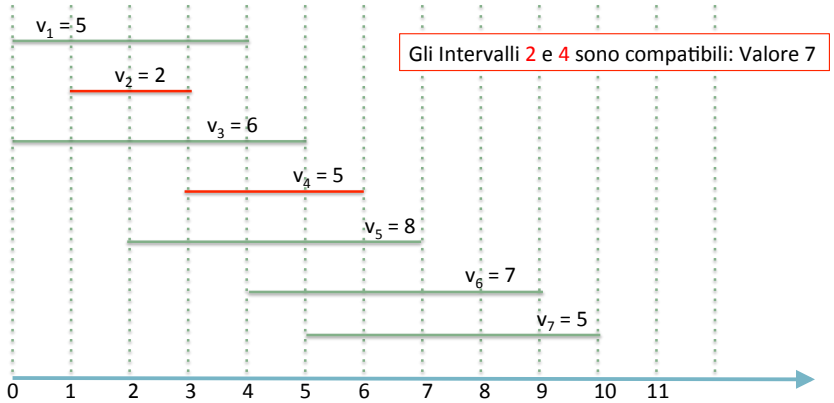
Scheduling di attività / intervalli

- Input: n intervalli
  - Intervallo j: inizia al tempo s_j; termina al tempo f_j ;ha valore v_j
  - Intervalli j ed i sono *compatibili*
    - se j ed i non si sovrappongono
- Output: Un insieme di intervalli compatibili di massimo valore



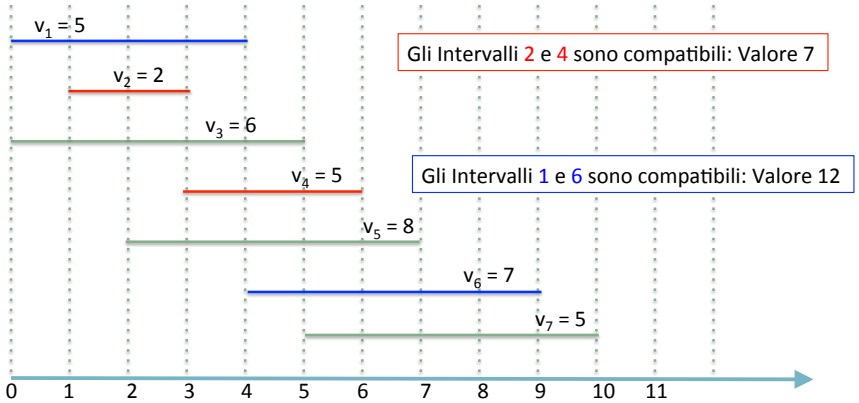
Scheduling di attività / intervalli

- Input: n intervalli
  - Intervallo j: inizia al tempo s_j; termina al tempo f_j ;ha valore v_j
  - Intervalli j ed i sono *compatibili*
    - se j ed i non si sovrappongono
- Output: Un insieme di intervalli compatibili di massimo valore



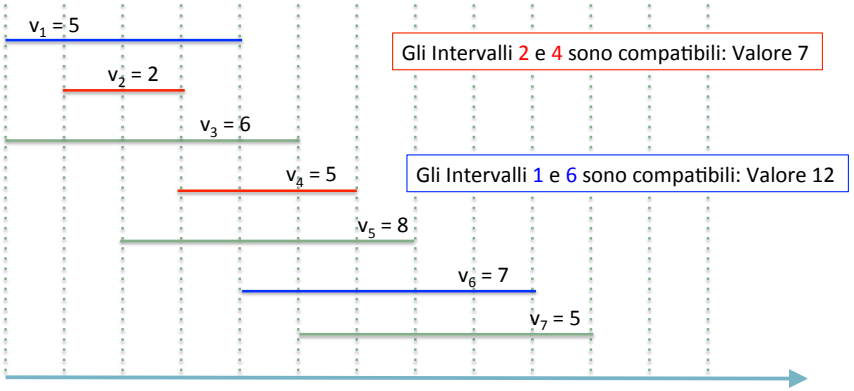
Scheduling di attività / intervalli

- Input: n intervalli
  - Intervallo j: inizia al tempo s_j; termina al tempo f_j ;ha valore v_j
  - Intervalli j ed i sono *compatibili*
    - se j ed i non si sovrappongono
- Output: Un insieme di intervalli compatibili di massimo valore



Scheduling di attività / intervalli

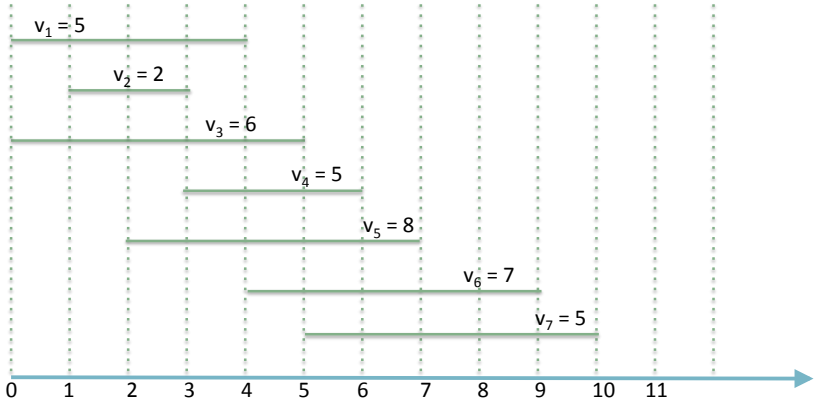
- Input: n intervalli
  - Intervallo j: inizia al tempo s_j; termina al tempo f_j ;ha valore v_j
  - Intervalli j ed i sono *compatibili*
    - se j ed i non si sovrappongono
- Output: Un insieme di intervalli compatibili di massimo valore



Greedy funzionava per valori uniformi. Qui fallisce !

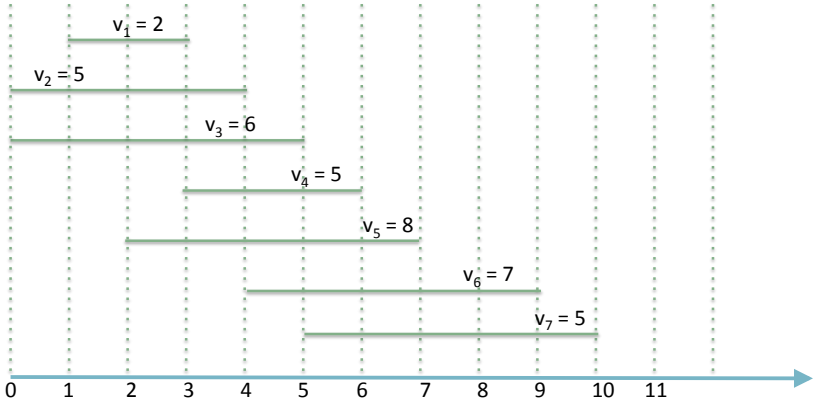
Scheduling di attività / intervalli – Progr. Din.

- Input: n intervalli
  - Intervallo j: inizia al tempo s_j; termina al tempo f_j ;ha valore v_j
  - Intervalli j ed i sono *compatibili*
    - se j ed i non si sovrappongono
- Output: Un insieme di intervalli compatibili di massimo valore



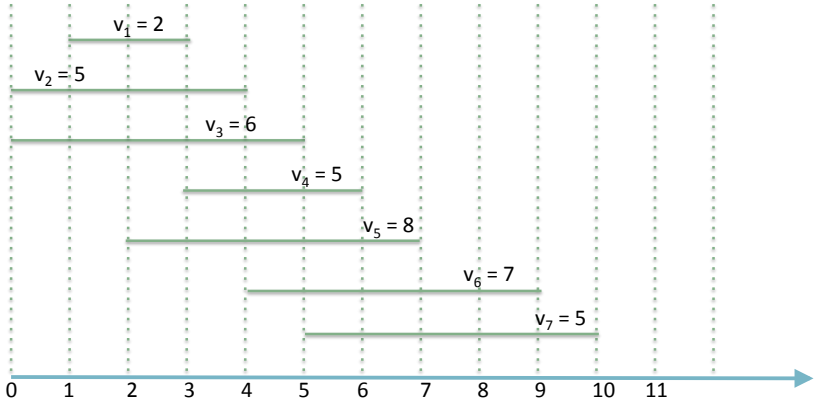
Scheduling di attività / intervalli – Progr. Din.

- $f_1 \le f_2 \le f_3 \le \ldots \le f_n$ (*Riordiniamo gli intervalli per tempo di fine*)



Scheduling di attività / intervalli – Progr. Din.

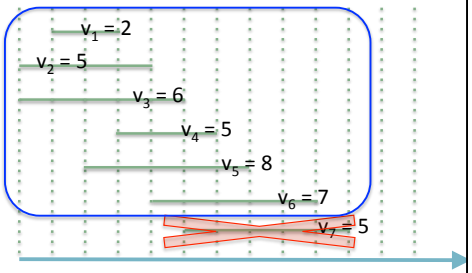
- $f_1 \le f_2 \le f_3 \le \ldots \le f_n$ (*Riordiniamo gli intervalli per tempo di fine*)
- Definizione: p(j): ultimo intervallo prima di j e compatibile con j
  - p(j+1), p(j+2), …, j-1 sono incompatibili con j
  - Es. p(4) = 1, p(7) = 3, p(3) = 0 (*nessun intervallo compatibile che precede*)



Caratterizziamo una soluzione ottima

Sia A una soluzione ottima

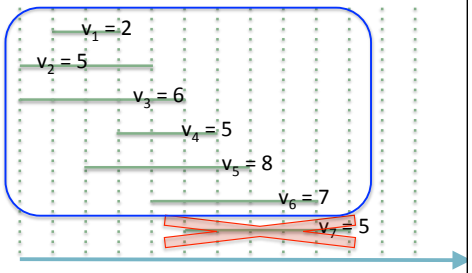
- n è in A oppure n non è in A (ovvio)
- Se n non è in A allora
  - A è una soluzione ottima per {1, …, n-1}



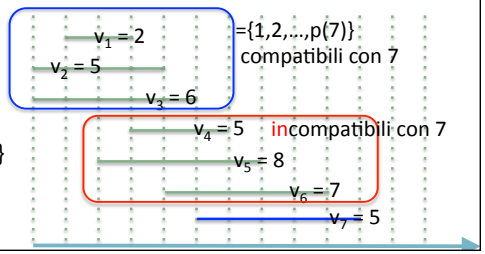
Caratterizziamo una soluzione ottima

Sia A una soluzione ottima

- n è in A oppure n non è in A (ovvio)
- Se n non è in A allora
  - A è una soluzione ottima per {1, …, n-1}



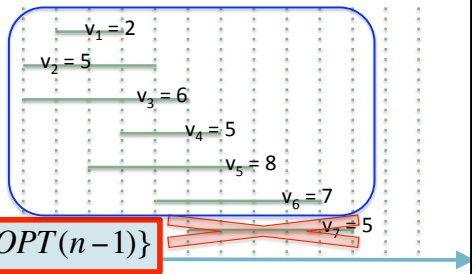
- Se n è in A allora
  - A\{n} è una soluzione ottima per {1, 2, …, p(n)}
    - p(n)+1, …, n-1, non sono in A
    - una soluzione B per {1,… p(n)} migliore di A\{n} implica che B' = B ∪{n} è una soluzione per {1, …, n} migliore di A



Caratterizziamo una soluzione ottima

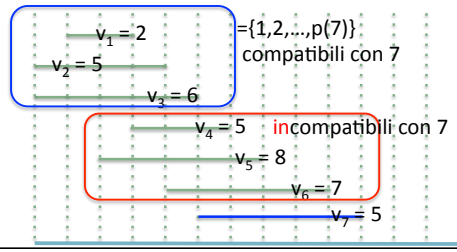
Sia A una soluzione ottima

- n è in A oppure n non è in A (ovvio)
- Se n non è in A allora
  - A è una soluzione ottima per {1, …, n-1}



$$OPT(n) = \max\{v_n + OPT(p(n)),\ OPT(n-1)\}$$

- Se n è in A allora
  - A\{n} è una soluzione ottima per {1, 2, …, p(n)}



Soluzione Ricorsiva: sottoproblemi utili

- Sia *OPT(j)* il valore della soluzione ottima per il sottoproblema definito sugli intervalli *{1, …, j}*

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

Il valore ottimo per il problema originario è *OPT(n)*

Soluzione Ricorsiva: sottoproblemi utili

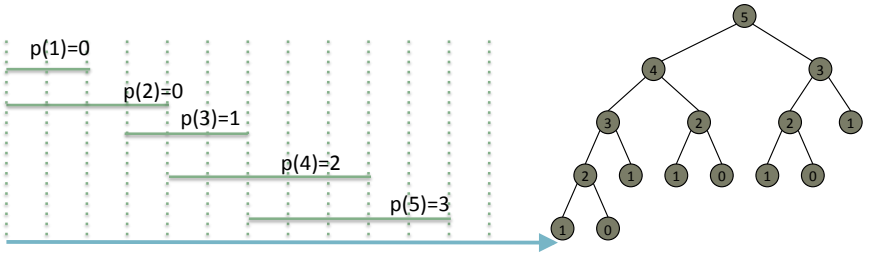
- Sia *OPT(j)* il valore della soluzione ottima per il sottoproblema definito sugli intervalli *{1, …, j}*

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

Il valore ottimo per il problema originario è *OPT(n)*

```
Input: n, s1, …, sn, f1, …, fn, v1, …, vn
Sort intervals by finishing time, f1 ≤ f2 ≤ … ≤ fn
Compute p(1), p(2), … p(n)
ComputeOpt(n)

Procedure ComputeOpt(j)
{
  if j = 0 return 0
  return max{ vj + ComputeOpt(p(j)), ComputeOpt(j-1) }
}
```

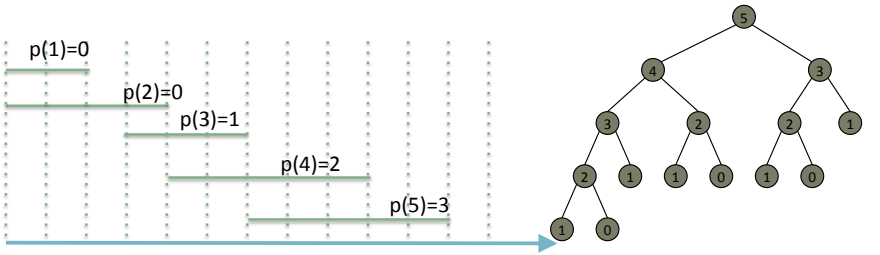

Soluzione Ricorsiva: sottoproblemi inutili



p(j)=j-2 $\Longrightarrow$ #chiamate ricorsive come fibonacci

```
Input: n, s1, …, sn, f1, …, fn, v1, …, vn
Sort intervals by finishing time, f1 ≤ f2 ≤ … ≤ fn
Compute p(1), p(2), … p(n)
ComputeOpt(n)

Procedure ComputeOpt(j)
{
  if j = 0 return 0
  return max{ vj + ComputeOpt(p(j)), ComputeOpt(j-1) }
}
```


Soluzione Ricorsiva con memoization

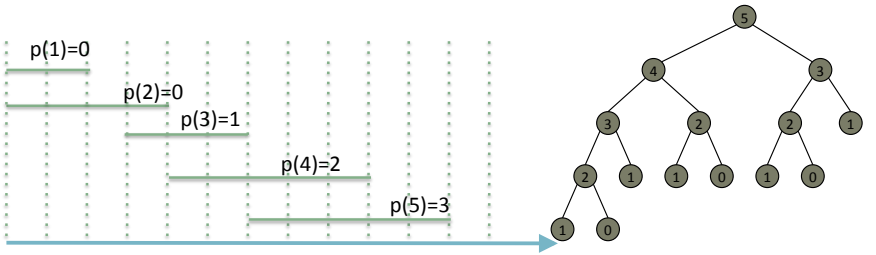


p(j)=j-2 $\Longrightarrow$ #chiamate ricorsive come fibonacci

```
Input: n, s1, …, sn, f1, …, fn, v1, …, vn
Sort intervals by finishing time, f1 ≤ f2 ≤ … ≤ fn
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Procedure MComputeOpt(j){
  if M[j] = -1
      M[j] = max{ vj + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

Memoization

Soluzione Ricorsiva con memoization



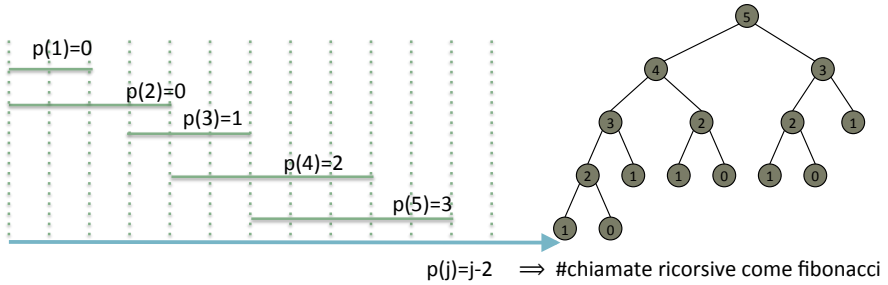
p(j)=j-2 $\Longrightarrow$ #chiamate ricorsive come fibonacci

```
Input: n, s1, …, sn, f1, …, fn, v1, …, vn
Sort intervals by finishing time, f1 ≤ f2 ≤ … ≤ fn
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Procedure MComputeOpt(j){
  if M[j] = -1
      M[j] = max{ vj + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

Memoization

MComputeOpt() ha complessità O(n). Perché???

Soluzione Ricorsiva con memoization



p(j)=j-2 $\Longrightarrow$ #chiamate ricorsive come fibonacci

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
     M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

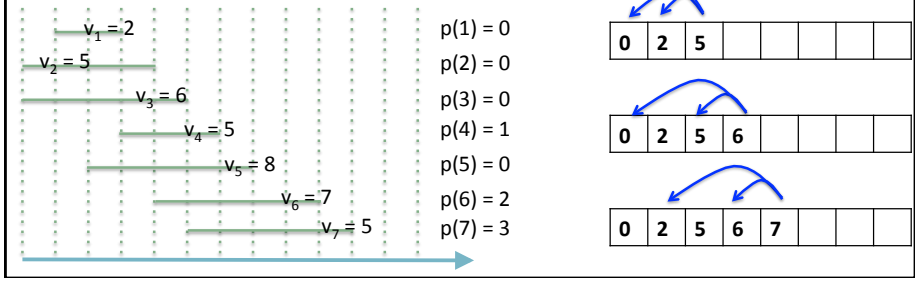
MComputeOpt() ha complessità O(n). Perché??? (come per Fibonacci)

Memoization vs Soluzione Iterativa

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
     M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

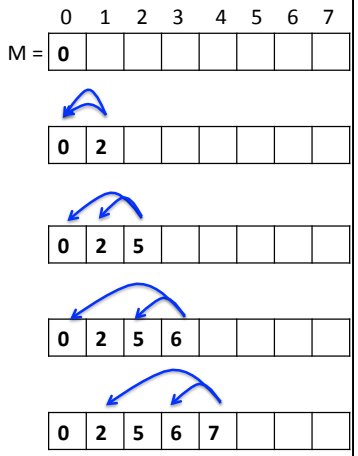


Memoization vs Soluzione Iterativa

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
     M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```



Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
     M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```


Come troviamo l'insieme di intervalli ???

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
     M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

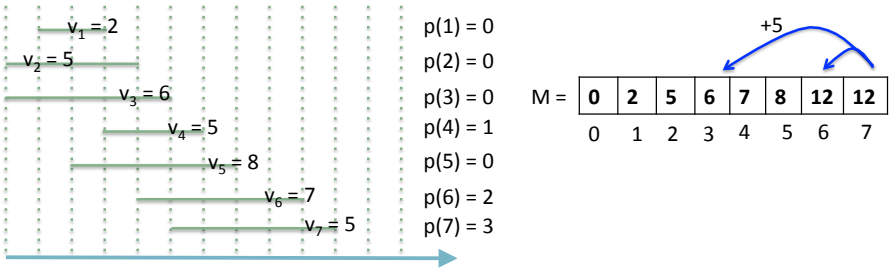
Qui calcoliamo solo il valore di una soluzione ottima

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
     M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Come troviamo l'insieme di intervalli ???



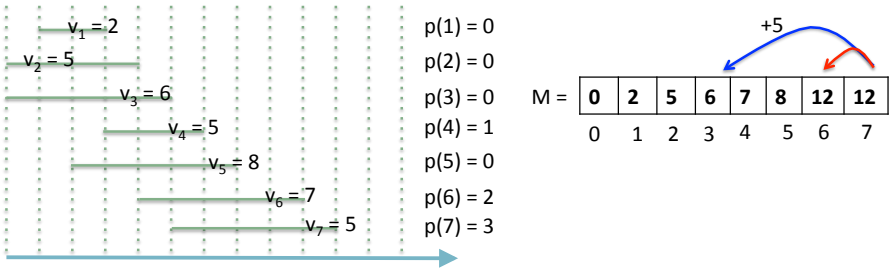
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???



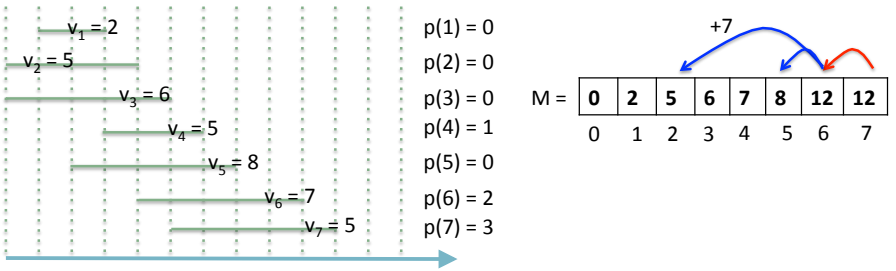
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???



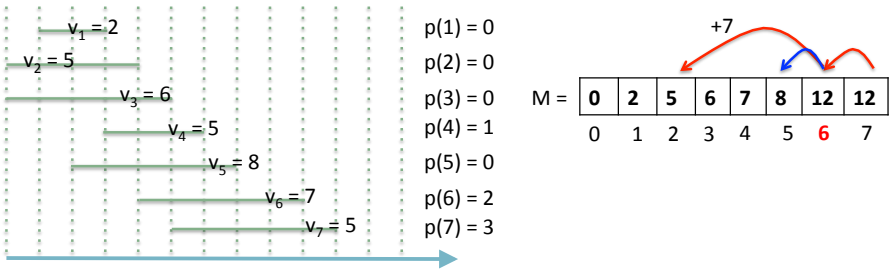
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???



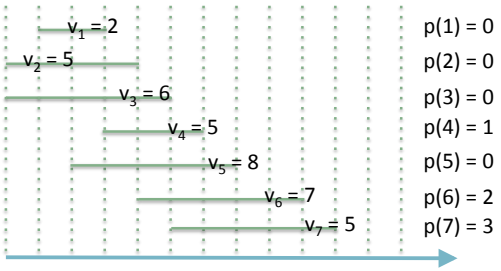
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

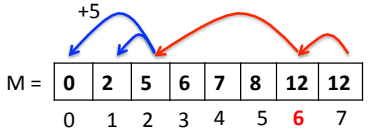
Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???





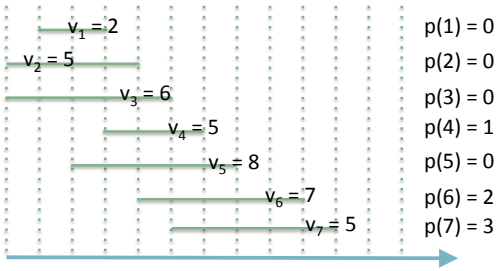
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

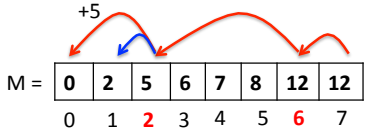
Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???





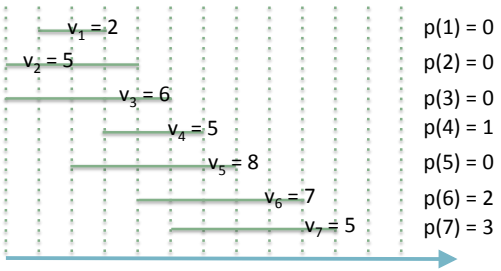
Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

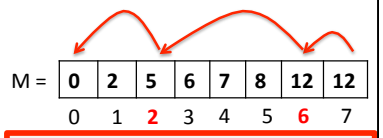
Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???






```
ComputeSolution(j)
{
  if j=0 return nothing

  if M[j-1] < v_j + M[p(j)]
          Output j
          ComputeSolution(p(j))
  else
          ComputeSolution(j-1)
}
```

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Seguiamo la ricorsione all'indietro per costruire l'insieme di intervalli di valore ottimo

Come troviamo l'insieme di intervalli ???

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
      M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

```
ComputeSolution(j)
{
  if j=0 return nothing

  if M[j-1] < v_j + M[p(j)]
          Output j
          ComputeSolution(p(j))
  else
          ComputeSolution(j-1)
}
```

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
      M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

Come troviamo l'insieme di intervalli ???

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Memoization

```
Procedure MComputeOpt(j){
  if M[j] = -1
     M[j] = max{ v_j + ComputeOpt(p(j)), ComputeOpt(j-1) }
  return M[j] }
```

Input: n, s_1, …, s_n, f_1, …, f_n, v_1, …, v_n
Sort intervals by finishing time, $f_1 \le f_2 \le \ldots \le f_n$
Compute p(1), p(2), … p(n)
Set M[0]=0, and M[1] = M[2] = … = M[n] = -1
MComputeOpt(n)

Iterativa

```
Procedure MComputeOpt(j){
  for i=1, …, j
     M[i] = max{ v_i + M[p(i)], M[i-1] }
  return M[j]}
```

```
ComputeSolution(j)
{
  if j=0 return nothing

  if M[j-1] < v_j + M[p(j)]
          Output j
          ComputeSolution(p(j))
  else
          ComputeSolution(j-1)
}
```

O(n) time :
chiamate ricorsive ≤ n

Programmazione Dinamica: La ricetta

- Caratterizza la struttura del problema
- Definisci ricorsivamente il valore ottimo
- Calcola il valore ottimo *una volta* per ogni sottoproblema utile
  - Memoization
  - Iterativamente da piccoli a grandi
- Costruisci la soluzione ottima dalle informazioni memorizzate sul valore

Esercizi

- Dimostrare che per Fib(n) abbiamo running time T(n) = 2F(n) -1
- Quante volte viene eseguito ogni sottoproblema da Fib(n)
  - Calcolare per j < n, quante volte Fib(j) viene invocato nell'albero di ricorsione di Fib(n)
- Scrivere una procedura che calcola *in tempo lineare* i valori p(j) necessari all'algoritmo che risolve il problema dello scheduling di intervalli
  [*Sugg.*: usare anche un array con gli intervalli ordinati per tempo di inizio]
- Far vedere che l'algoritmo greedy che sceglie un nuovo intervallo compatibile con quelli già scelti cercando di massimizzare il rapporto tra il valore e la durata (o il tempo di terminazione) non fornisce la soluzione ottima.