Lezione 18

Sommario della Lezione

Introduzione alla Tecnica Greedy

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;
- Ad ogni passo aggiunge una nuova parte alla soluzione precedentemente computata,

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;
- Ad ogni passo aggiunge una nuova parte alla soluzione precedentemente computata, fino ad ottenere una soluzione al problema intero;

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;
- Ad ogni passo aggiunge una nuova parte alla soluzione precedentemente computata, fino ad ottenere una soluzione al problema intero;
- La nuova parte di soluzione che viene scelta ad ogni dato passo è quella che, tra *tutte* le possibili parti che si potrebbero aggiungere, risulta essere la migliore

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;
- Ad ogni passo aggiunge una nuova parte alla soluzione precedentemente computata, fino ad ottenere una soluzione al problema intero;
- La nuova parte di soluzione che viene scelta ad ogni dato passo è quella che, tra *tutte* le possibili parti che si potrebbero aggiungere, risulta essere la migliore (ad es., in un problema di ottimizzazione di massimo, potrebbe essere quella che ci dà il maggior incremento di valore alla soluzione parziale fin'ora calcolata,

Informalmente, un algoritmo Greedy costruisce una soluzione ad un dato problema algoritmico *attraverso una sequenza iterativa di passi*:

- L'algoritmo inizia con il trovare una soluzione ad un sottoproblema di piccola taglia;
- Ad ogni passo aggiunge una nuova parte alla soluzione precedentemente computata, fino ad ottenere una soluzione al problema intero;
- La nuova parte di soluzione che viene scelta ad ogni dato passo è quella che, tra *tutte* le possibili parti che si potrebbero aggiungere, risulta essere la migliore (ad es., in un problema di ottimizzazione di massimo, potrebbe essere quella che ci dà il maggior incremento di valore alla soluzione parziale fin'ora calcolata, in un problema di minimo potrebbe essere quella che causa il minor incremento di costo).

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \ldots v[n]$, con $v[1] > v[2] > \ldots > v[n] = 1$.

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
```

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
```

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
```

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
```

Input: Un valore monetario V, ed un vettore di valori di monete $v[1] \dots v[n]$, con $v[1] > v[2] > \dots > v[n] = 1$.
Output: Un insieme S di *cardinalità minima* di monete, la cui somma dei valori sia esattamente pari a V.

In accordo all'idea Greedy, un possibile algoritmo potrebbe essere quello in cui si cerca di esprimere il valore monetario V utilizzando, se possibile, sempre la moneta di valore massimo il cui valore sia $\leq$ del valore corrente che vogliamo cambiare.

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo Cambio_greedy con il seguente input:
v[1]=10,v[2]=5,v[3]=1,V=37,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$, $S = \{1, 1\}$ e $V = 17$,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$, $S = \{1, 1\}$ e $V = 17$, $S = \{1, 1, 1\}$ e $V = 7$,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$, $S = \{1, 1\}$ e $V = 17$, $S = \{1, 1, 1\}$ e $V = 7$, $S = \{1, 1, 1, 2\}$ e $V = 2$,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$, $S = \{1, 1\}$ e $V = 17$, $S = \{1, 1, 1\}$ e $V = 7$, $S = \{1, 1, 1, 2\}$ e $V = 2$, $S = \{1, 1, 1, 2, 3\}$ e $V = 1$,

```
Cambio_greedy (V,v[1]...v[n])
1. S=∅
2. WHILE (V>0) {
3.     sia i la moneta di massimo valore tale che v[i]≤V
4.     S=S∪{i},
5.     V=V-v[i]
      }
6. RETURN S
```

Eseguiamo l'algoritmo `Cambio_greedy` con il seguente input: `v[1]=10,v[2]=5,v[3]=1,V=37`, otterremmo che l'insieme `S` ha la seguente composizione, ad ogni iterazione del ciclo `WHILE`.
$S = \{1\}$ e $V = 27$, $S = \{1, 1\}$ e $V = 17$, $S = \{1, 1, 1\}$ e $V = 7$, $S = \{1, 1, 1, 2\}$ e $V = 2$, $S = \{1, 1, 1, 2, 3\}$ e $V = 1$, $S = \{1, 1, 1, 2, 3, 3\}$ e $V = 0$.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1)

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1) o 2 occorrenze della moneta 2 (di valore 5),

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1) o 2 occorrenze della moneta 2 (di valore 5), oppure 5 occorrenze della moneta 3 ed una occorrenza della moneta 2.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1) o 2 occorrenze della moneta 2 (di valore 5), oppure 5 occorrenze della moneta 3 ed una occorrenza della moneta 2.
Sostituendo a tali occorrenze **una sola** occorrenza della moneta 1 di valore 10 diminueremmo il numero di monete in $Opt(V)$, contro l'ipotesi che $Opt(V)$ fosse un insieme di cardinalitá minima.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1) o 2 occorrenze della moneta 2 (di valore 5), oppure 5 occorrenze della moneta 3 ed una occorrenza della moneta 2.
Sostituendo a tali occorrenze **una sola** occorrenza della moneta 1 di valore 10 diminueremmo il numero di monete in $Opt(V)$, contro l'ipotesi che $Opt(V)$ fosse un insieme di cardinalitá minima.
Per ipotesi induttiva, l'insieme S' prodotto dall'algoritmo `Cambio_greedy`$(V - 10, v[1]v[2]v[3])$ è una soluzione ottima per il sottoproblema di esprimere il valore monetario $V - 10$ usando le monete $v[1], v[2], v[3]$, ovvero $S' = Opt(V - 10)$.

`Cambio_greedy` restituisce l'insieme S di cardinalità minima, *sull'esempio in questione*: `v[1]=10,v[2]=5,v[3]=1,V=37`.
La prova è per induzione su V.
Sia $V > 10$. Proviamo che la soluzione ottima, ovvero quella con il minor numero di monete, (sia essa $Opt(V)$) **deve** contenere la moneta 1 di valore 10.
Se non la contenesse, allora al suo posto $Opt(V)$ contiene o 10 occorrenze della moneta 3 (di valore 1) o 2 occorrenze della moneta 2 (di valore 5), oppure 5 occorrenze della moneta 3 ed una occorrenza della moneta 2.
Sostituendo a tali occorrenze **una sola** occorrenza della moneta 1 di valore 10 diminueremmo il numero di monete in $Opt(V)$, contro l'ipotesi che $Opt(V)$ fosse un insieme di cardinalitá minima.
Per ipotesi induttiva, l'insieme S' prodotto dall'algoritmo `Cambio_greedy`$(V - 10, v[1]v[2]v[3])$ è una soluzione ottima per il sottoproblema di esprimere il valore monetario $V - 10$ usando le monete $v[1], v[2], v[3]$, ovvero $S' = Opt(V - 10)$. Di conseguenza, vale che $S = S' \cup \{1\} = Opt(V - 10) \cup \{1\} = Opt(V)$.

Ma allora perchè abbiamo precedentemente usato Programmazione Dinamica?

Ma allora perchè abbiamo precedentemente usato Programmazione Dinamica?

Perchè l'algoritmo `Cambio_greedy` non fornisce la soluzione ottima su **tutti** gli input.

Ma allora perchè abbiamo precedentemente usato Programmazione Dinamica?

Perchè l'algoritmo `Cambio_greedy` non fornisce la soluzione ottima su **tutti** gli input.

Sia $V = 12$, e $v[1] = 10, v[2] = 6, v[3] = 1$,

Ma allora perchè abbiamo precedentemente usato Programmazione Dinamica?

Perchè l'algoritmo `Cambio_greedy` non fornisce la soluzione ottima su **tutti** gli input.

Sia $V = 12$, e $v[1] = 10, v[2] = 6, v[3] = 1$, `Cambio_greedy` restituirebbe la soluzione $S = \{1, 3, 3\}$ di cardinalità 3, mentre la soluzione con il minor numero di monete è $\{2, 2\}$.

Ma allora perchè abbiamo precedentemente usato Programmazione Dinamica?

Perchè l'algoritmo `Cambio_greedy` non fornisce la soluzione ottima su **tutti** gli input.

Sia $V = 12$, e $v[1] = 10, v[2] = 6, v[3] = 1$, `Cambio_greedy` restituirebbe la soluzione $S = \{1, 3, 3\}$ di cardinalità 3, mentre la soluzione con il minor numero di monete è $\{2, 2\}$.

Il problema di stabilire se, dati V e $v[1], \ldots v[n]$, l'algoritmo Greedy restituisce una soluzione ottima, non è agevole da risolvere, ma di questo non ci occuperemo.

Esempio di Applicazione di Greedy

Input del problema: Supponiamo di avere un insieme $A = \{A_1, A_2, \ldots, A_n\}$ di attività , dove ciascuna attività A_i ha un tempo di *inizio* s_i, ed un tempo di *fine* f_i, con $s_i < f_i$ (in altre parole, l'attività A_i deve essere svolta nell'intervallo temporale $[s_i, f_i]$)

Esempio di Applicazione di Greedy

Input del problema: Supponiamo di avere un insieme $A = \{A_1, A_2, \ldots, A_n\}$ di attività , dove ciascuna attività A_i ha un tempo di *inizio* s_i, ed un tempo di *fine* f_i, con $s_i < f_i$ (in altre parole, l'attività A_i deve essere svolta nell'intervallo temporale $[s_i, f_i]$)

Le attività in A devono essere assegnate ad una *risorsa*, sotto la condizione che se A_i ed A_j vengono entrambe assegnate alla risorsa, allora $[s_i, f_i] \cap [s_j, f_j] = \emptyset$ (in tal caso, diremo che l'attività A_i ed A_j sono *compatibili*).

Esempio di Applicazione di Greedy

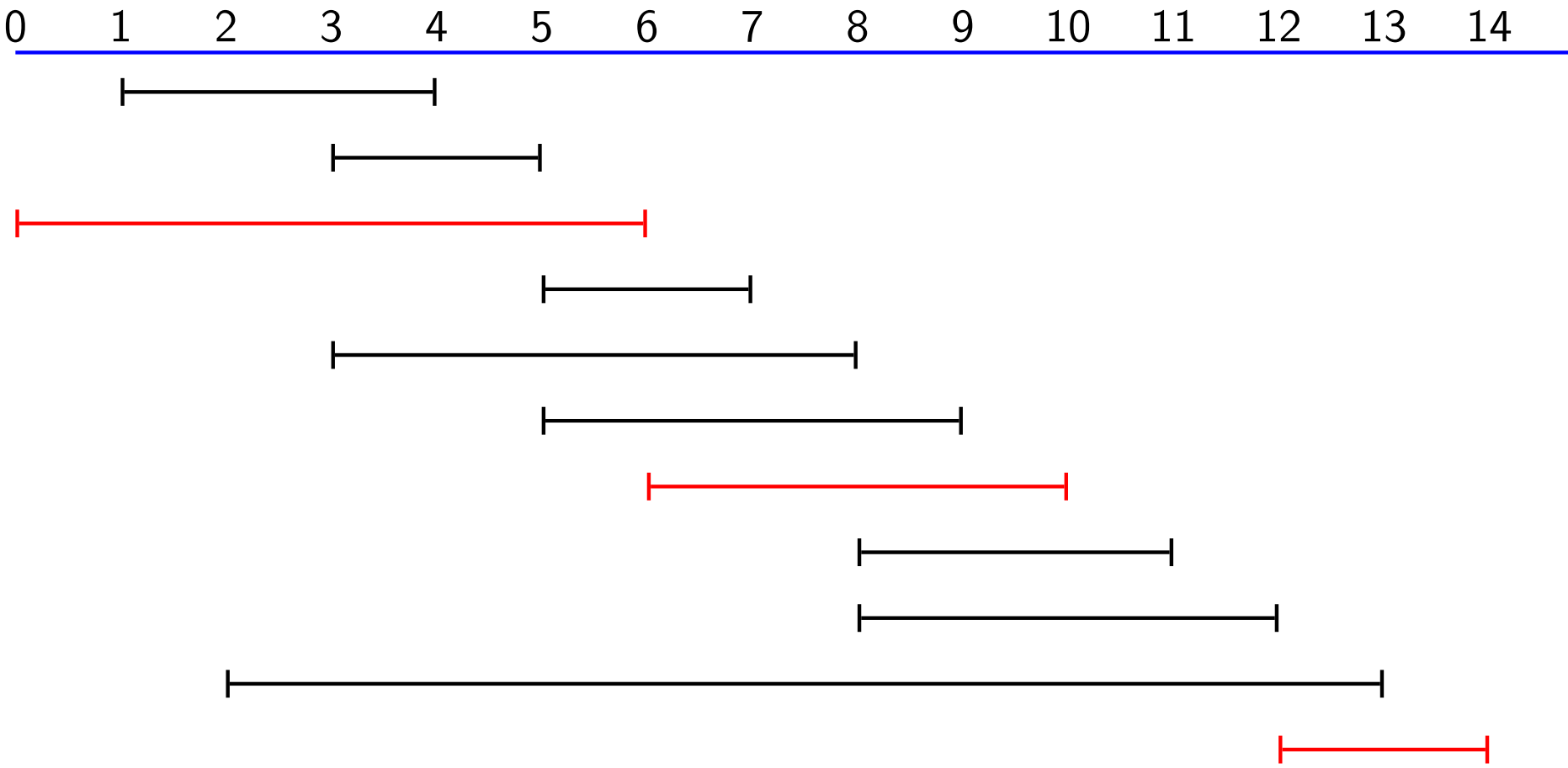
Input del problema: Supponiamo di avere un insieme $A = \{A_1, A_2, \ldots, A_n\}$ di attività , dove ciascuna attività A_i ha un tempo di *inizio* s_i, ed un tempo di *fine* f_i, con $s_i < f_i$ (in altre parole, l'attività A_i deve essere svolta nell'intervallo temporale $[s_i, f_i]$)

Le attività in A devono essere assegnate ad una *risorsa*, sotto la condizione che se A_i ed A_j vengono entrambe assegnate alla risorsa, allora $[s_i, f_i] \cap [s_j, f_j] = \emptyset$ (in tal caso, diremo che l'attività A_i ed A_j sono *compatibili*).

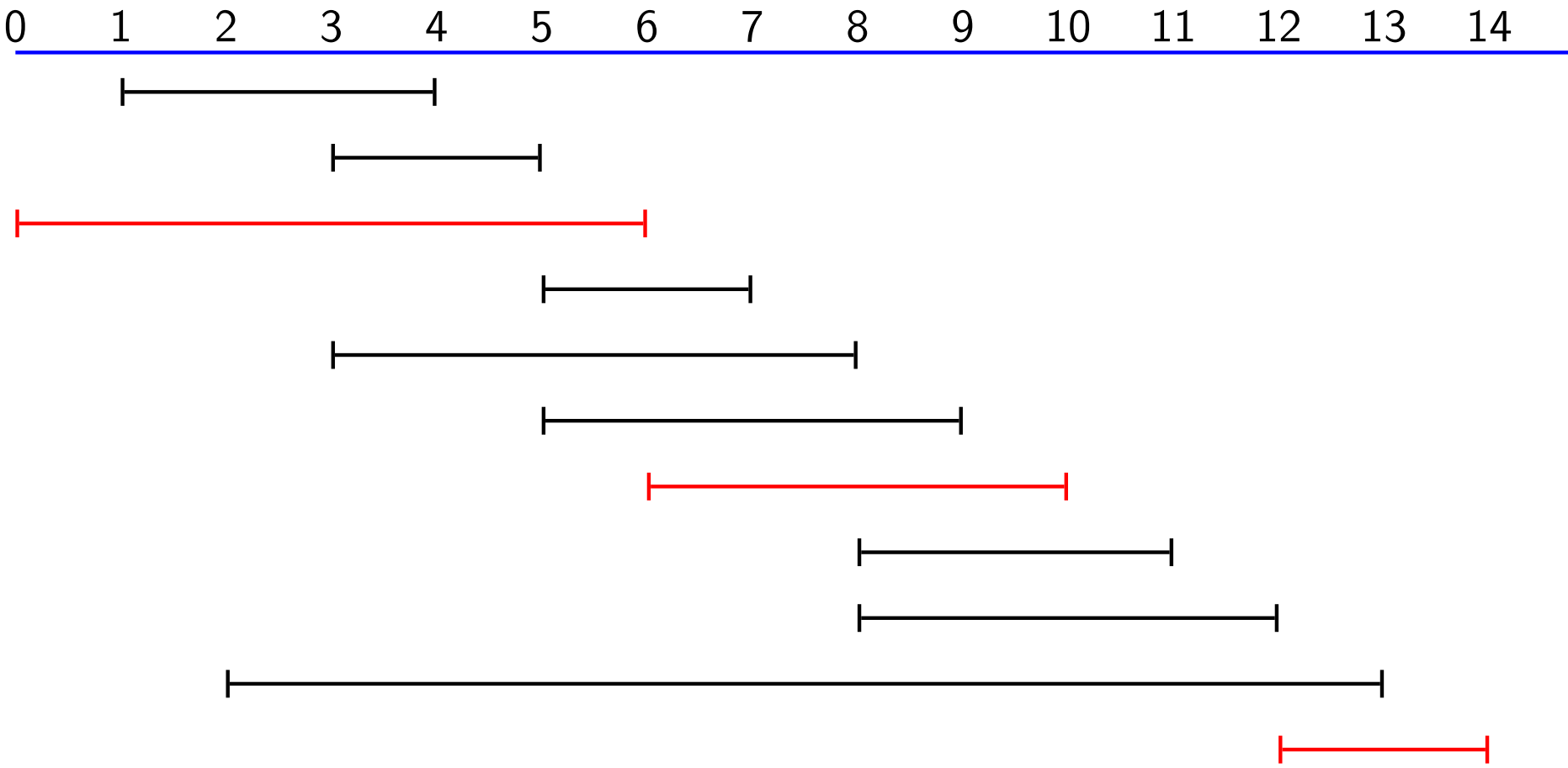
Output del problema: Calcolare un sottoinsieme di $S \subseteq A$ di attività a due a due compatibili, di cardinalità **massima**.

Vediamo un esempio, con $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$.

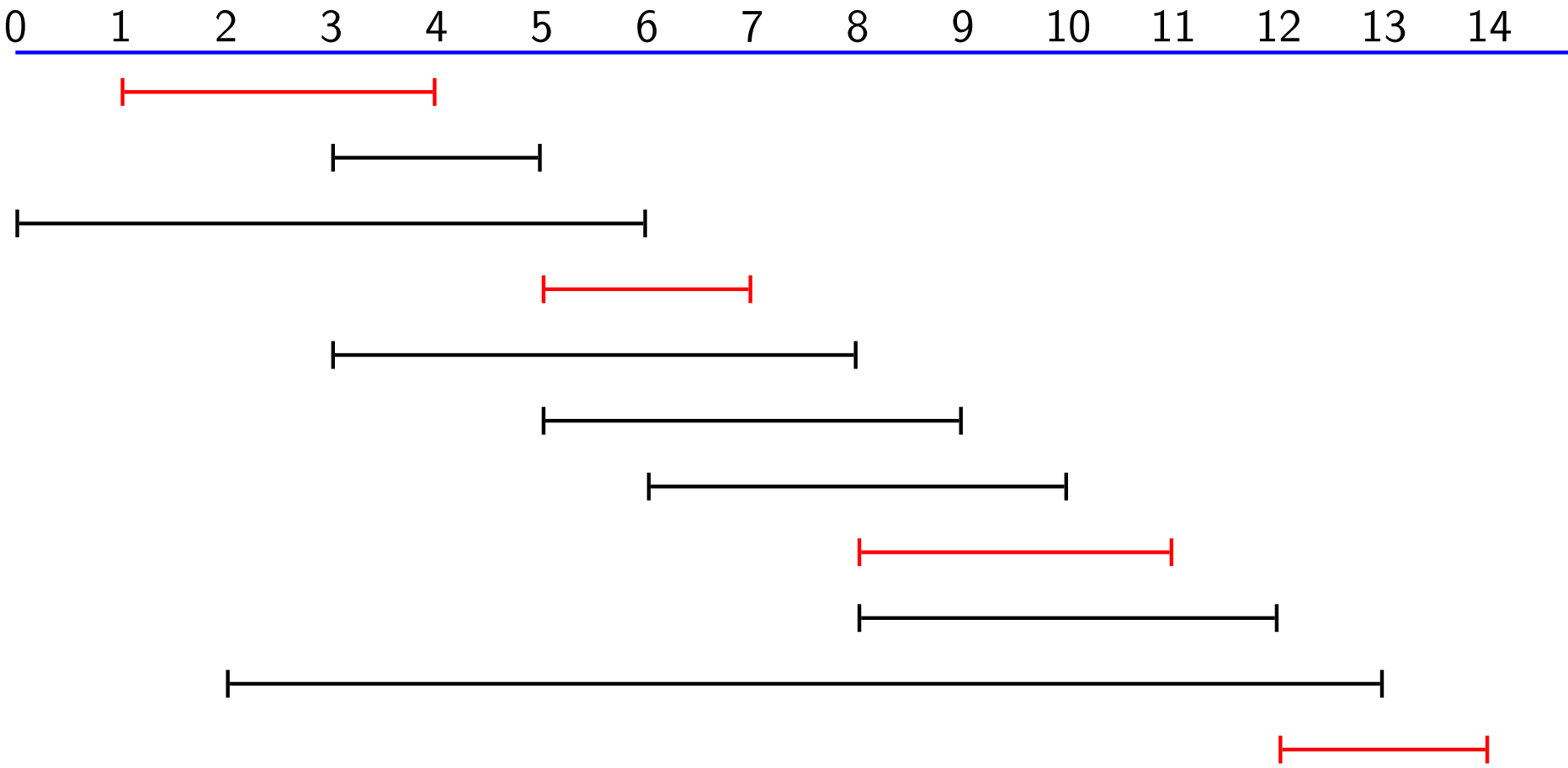
Vediamo un esempio, con $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$.



Vediamo un esempio, con $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. In tal caso, una prima possibile soluzione sarebbe: A_3, A_7, A_{11}.



Date $A = \{A_1, A_2, \ldots, A_{11}\}$, $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$ una seconda possibile soluzione sarebbe A_1, A_4, A_8, A_{11}.

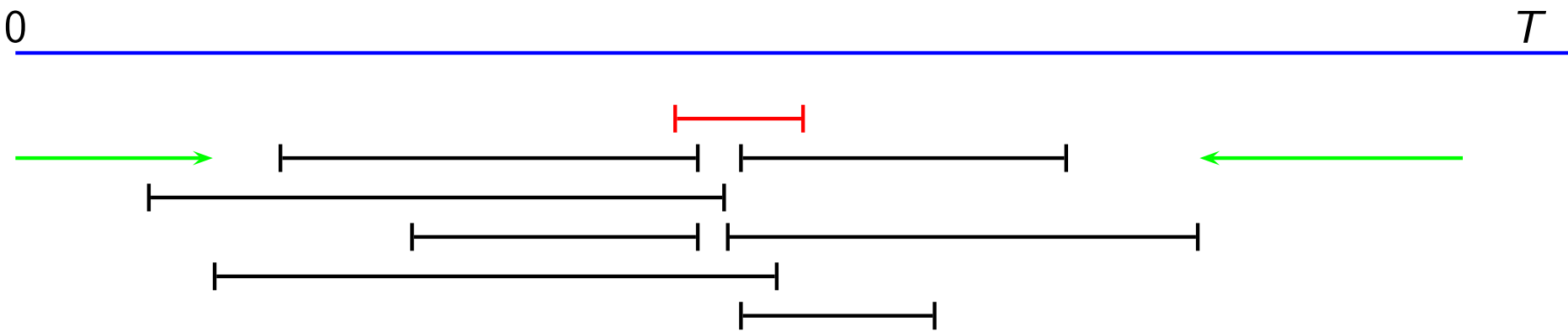


L'algoritmo Greedy costruisce la soluzione iterativamente, effettuando ad ogni passo la scelta che in quel momento sembra essere la migliore.

L'algoritmo Greedy costruisce la soluzione iterativamente, effettuando ad ogni passo la scelta che in quel momento sembra essere la migliore.
Quale potrebbe essere la prima *migliore* scelta (ovvero la prima attività A_i) da effettuare?

L'algoritmo Greedy costruisce la soluzione iterativamente, effettuando ad ogni passo la scelta che in quel momento sembra essere la migliore.
Quale potrebbe essere la prima *migliore* scelta (ovvero la prima attività A_i) da effettuare?
Sembra ragionevole scegliere come prima attività da inserire nella soluzione una che richieda il *minor tempo* di essere svolta, così da *massimizzare* il tempo utile rimanente in cui svolgere eventuali altre attività .

L'algoritmo Greedy costruisce la soluzione iterativamente, effettuando ad ogni passo la scelta che in quel momento sembra essere la migliore.
Quale potrebbe essere la prima *migliore* scelta (ovvero la prima attività A_i) da effettuare?
Sembra ragionevole scegliere come prima attività da inserire nella soluzione una che richieda il *minor tempo* di essere svolta, così da *massimizzare* il tempo utile rimanente in cui svolgere eventuali altre attività . Immaginiamo però la situazione seguente:



Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Se A_1 è presente, allora occorre trovare la *migliore* soluzione possibile ai sottoproblemi in A' ed A'' relativi ad attività compatibili con A_1:

$$A' = \{A_j : f_j < s_1\} \quad A'' = \{A_k : s_k > f_1\}.$$

Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Se A_1 è presente, allora occorre trovare la *migliore* soluzione possibile ai sottoproblemi in A' ed A'' relativi ad attività compatibili con A_1:

$$A' = \{A_j : f_j < s_1\} \quad A'' = \{A_k : s_k > f_1\}.$$

Se invece A_1 non è presente, allora occorre risolvere in maniera ottima il sottoproblema relativamente alle attività $\{A_2, \ldots, A_n\}$.

Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Se A_1 è presente, allora occorre trovare la *migliore* soluzione possibile ai sottoproblemi in A' ed A'' relativi ad attività compatibili con A_1:

$$A' = \{A_j : f_j < s_1\} \quad A'' = \{A_k : s_k > f_1\}.$$

Se invece A_1 non è presente, allora occorre risolvere in maniera ottima il sottoproblema relativamente alle attività $\{A_2, \ldots, A_n\}$.

La soluzione ottima al problema globale è la migliore di quella che potremmo ottenere dalle due alternative disopra.

Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Se A_1 è presente, allora occorre trovare la *migliore* soluzione possibile ai sottoproblemi in A' ed A'' relativi ad attività compatibili con A_1:

$$A' = \{A_j : f_j < s_1\} \quad A'' = \{A_k : s_k > f_1\}.$$

Se invece A_1 non è presente, allora occorre risolvere in maniera ottima il sottoproblema relativamente alle attività $\{A_2, \ldots, A_n\}$.

La soluzione ottima al problema globale è la migliore di quella che potremmo ottenere dalle due alternative disopra.

Queste considerazioni danno origine ad una equazione di ricorrenza e relativo algoritmo di Programmazione Dinamica di complessità $O(n^2)$

Osservazione: nella soluzione ottima o ci sarà la prima attività A_1 oppure essa non sarà presente.

Se A_1 è presente, allora occorre trovare la *migliore* soluzione possibile ai sottoproblemi in A' ed A'' relativi ad attività compatibili con A_1:

$$A' = \{A_j : f_j < s_1\} \quad A'' = \{A_k : s_k > f_1\}.$$

Se invece A_1 non è presente, allora occorre risolvere in maniera ottima il sottoproblema relativamente alle attività $\{A_2, \ldots, A_n\}$.

La soluzione ottima al problema globale è la migliore di quella che potremmo ottenere dalle due alternative disopra.

Queste considerazioni danno origine ad una equazione di ricorrenza e relativo algoritmo di Programmazione Dinamica di complessità $O(n^2)$ (sviluppare un tale algoritmo sarebbe un ulteriore utile esercizio.)

Osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora l'attività A_1 sarebbe quella che termina prima di tutte.

Osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora l'attività A_1 sarebbe quella che termina prima di tutte.

Pertanto, il sottoproblema $A' = \{A_j : f_j < s_1\} = \emptyset$.

Osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora l'attività A_1 sarebbe quella che termina prima di tutte.

Pertanto, il sottoproblema $A' = \{A_j : f_j < s_1\} = \emptyset$.

Altra osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora scegliendo l'attività A_1, avremmo *massimizzato* il tempo libero che ci rimane,

Osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora l'attività A_1 sarebbe quella che termina prima di tutte.

Pertanto, il sottoproblema $A' = \{A_j : f_j < s_1\} = \emptyset$.

Altra osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora scegliendo l'attività A_1, avremmo *massimizzato* il tempo libero che ci rimane, in cui poter eventualmente eseguire le restanti attività .

Osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora l'attività A_1 sarebbe quella che termina prima di tutte.

Pertanto, il sottoproblema $A' = \{A_j : f_j < s_1\} = \emptyset$.

Altra osservazione: Se le attività fossero ordinate in accordo ai loro tempi di fine, allora scegliendo l'attività A_1, avremmo *massimizzato* il tempo libero che ci rimane, in cui poter eventualmente eseguire le restanti attività .

Ciò ci lascia pensare che forse *conviene* prenderla l'attività A_1, e quindi ignorare il sottoproblema derivante dalla non presa in considerazione di A_1.

Un possibile algoritmo:

Un possibile algoritmo:

- Ordina le attività in base al loro *tempo di fine* (sia $A_1, \ldots, A_n$ tale sequenza ordinata).

Un possibile algoritmo:

- Ordina le attività in base al loro *tempo di fine* (sia $A_1, \ldots, A_n$ tale sequenza ordinata).
- Scegli la *prima* attività A_1

Un possibile algoritmo:

- Ordina le attività in base al loro *tempo di fine* (sia $A_1, \ldots, A_n$ tale sequenza ordinata).
- Scegli la *prima* attività A_1
- Elimina tutte le attività che si intersecano con A_1 (ovvero che iniziano prima che A_1 finisca)

Un possibile algoritmo:

- Ordina le attività in base al loro *tempo di fine* (sia $A_1, \ldots, A_n$ tale sequenza ordinata).
- Scegli la *prima* attività A_1
- Elimina tutte le attività che si intersecano con A_1 (ovvero che iniziano prima che A_1 finisca)
- Ricorsivamente risolvi il problema sulle attività che restano.

`Greedy_Activity_Selector`$(s_1, \ldots, s_n, f_1, \ldots, f_n)$

Greedy_Activity_Selector($s_1, \ldots, s_n, f_1, \ldots, f_n$)

1. Ordina le attività in accordo alle f_i

Greedy_Activity_Selector($s_1, \ldots, s_n, f_1, \ldots, f_n$)

1. Ordina le attività in accordo alle f_i
2. S=$\{A_1\}$, j=1

Greedy_Activity_Selector($s_1, \ldots, s_n, f_1, \ldots, f_n$)

1. Ordina le attività in accordo alle f_i
2. S=$\{A_1\}$, j=1
3. FOR (i=2, i<n+1, i=i+1) {

Greedy_Activity_Selector($s_1, \ldots, s_n, f_1, \ldots, f_n$)

1. Ordina le attività in accordo alle f_i
2. S=$\{A_1\}$, j=1
3. FOR (i=2, i<n+1, i=i+1) {
4. IF ($s_i \geq f_j$) {

Greedy_Activity_Selector($s_1, \ldots, s_n, f_1, \ldots, f_n$)

1. Ordina le attività in accordo alle f_i
2. S=$\{A_1\}$, j=1
3. FOR (i=2, i<n+1, i=i+1) {
4. IF ($s_i \geq f_j$) {
5. S=S$\cup\{A_i\}$

```
Greedy_Activity_Selector(s_1, ..., s_n, f_1, ..., f_n)
1. Ordina le attività in accordo alle f_i
2. S={A_1},  j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.    IF (s_i ≥ f_j) {
5.       S=S∪{A_i}
6.       j=i
        }
    }
```

```
Greedy_Activity_Selector(s_1, ..., s_n, f_1, ..., f_n)
1. Ordina le attività in accordo alle f_i
2. S={A_1},  j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.    IF (s_i ≥ f_j) {
5.       S=S∪{A_i}
6.       j=i
        }
    }
7.    RETURN S
```

```
Greedy_Activity_Selector(s_1, ..., s_n, f_1, ..., f_n)
1. Ordina le attività in accordo alle f_i
2. S={A_1},  j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.    IF (s_i ≥ f_j) {
5.       S=S∪{A_i}
6.       j=i
        }
    }
7.    RETURN S
```

La complessità dell'algoritmo è dominata dal tempo richiesto nell'esecuzione dell'ordinamento nel passo 1.,

```
Greedy_Activity_Selector(s_1, ..., s_n, f_1, ..., f_n)
1. Ordina le attività in accordo alle f_i
2. S={A_1},  j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.    IF (s_i ≥ f_j) {
5.       S=S∪{A_i}
6.       j=i
        }
    }
7.    RETURN S
```

La complessità dell'algoritmo è dominata dal tempo richiesto nell'esecuzione dell'ordinamento nel passo 1., in quanto il resto dell'algoritmo può essere realizzato in tempo $O(n)$.

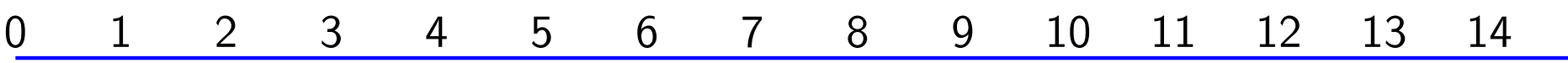
```
Greedy_Activity_Selector(s_1, ..., s_n, f_1, ..., f_n)
1. Ordina le attività in accordo alle f_i
2. S={A_1},  j=1
3. FOR (i=2, i<n+1, i=i+1) {
4.    IF (s_i ≥ f_j) {
5.       S=S∪{A_i}
6.       j=i
        }
    }
7.    RETURN S
```

La complessità dell'algoritmo è dominata dal tempo richiesto nell'esecuzione dell'ordinamento nel passo 1., in quanto il resto dell'algoritmo può essere realizzato in tempo $O(n)$.

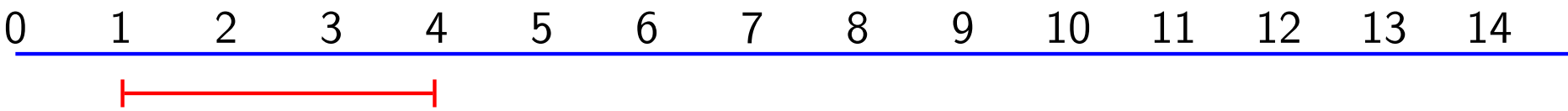
L'intero algoritmo richiede tempo $\Theta(n \log n)$ nel caso peggiore

Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$.

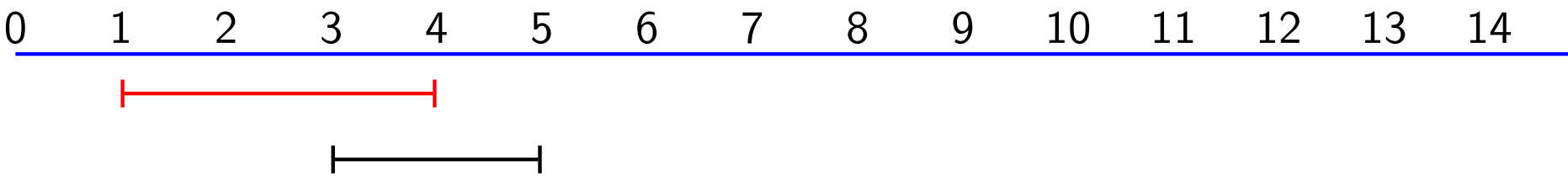
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



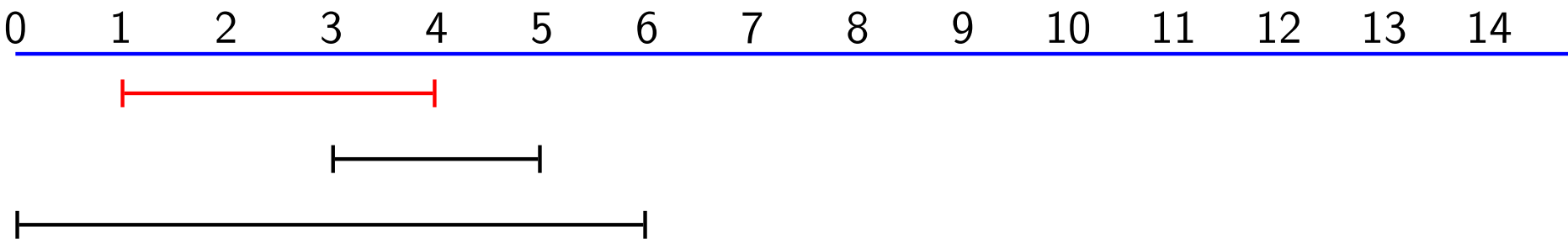
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



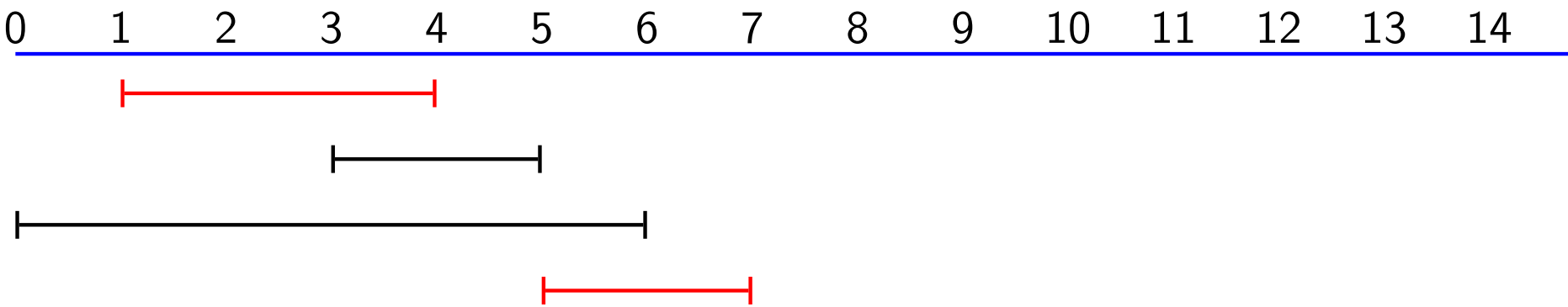
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



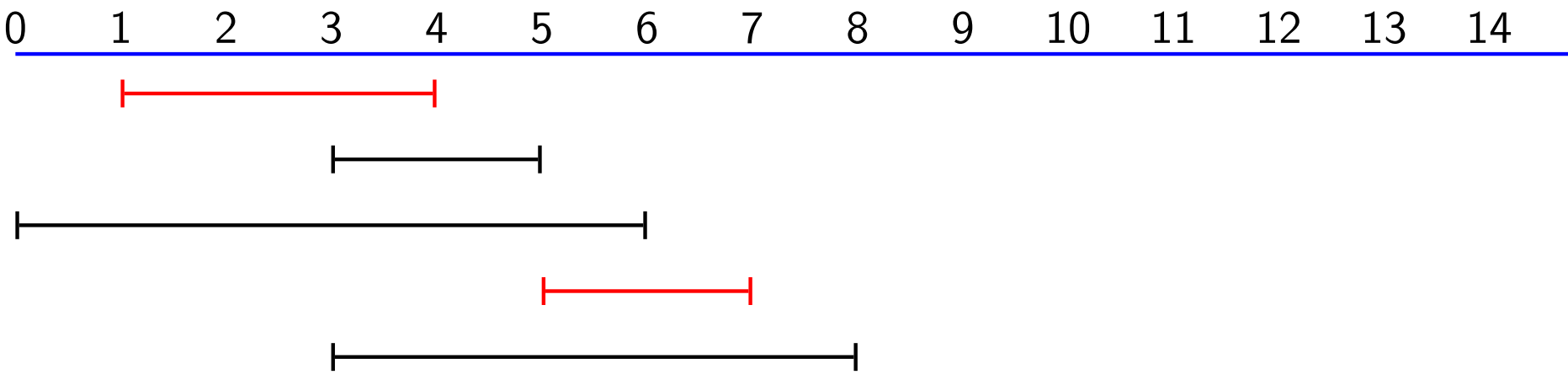
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



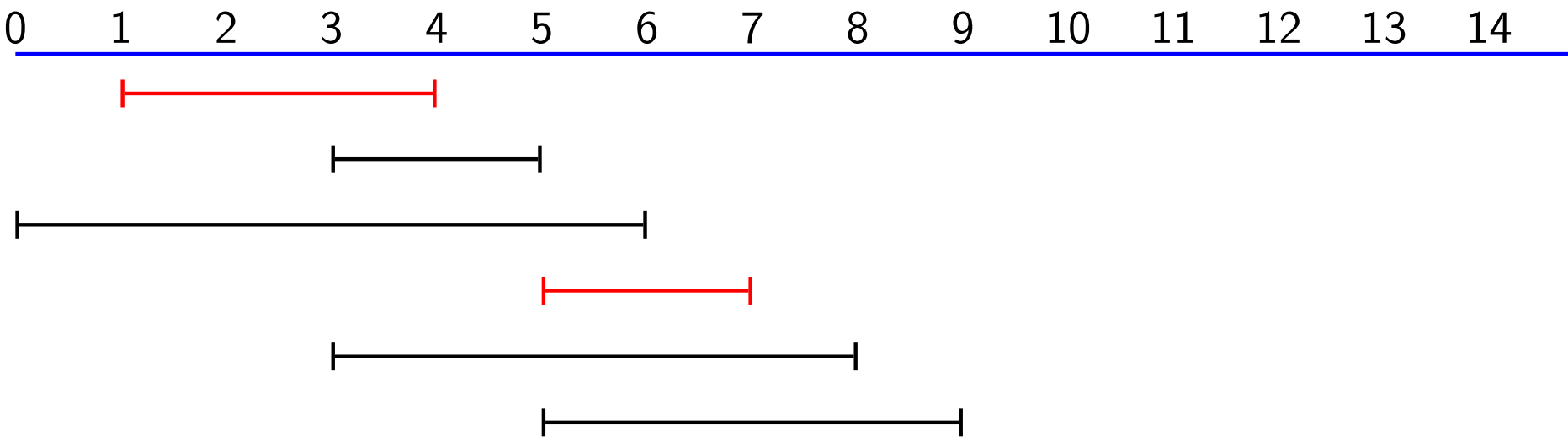
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



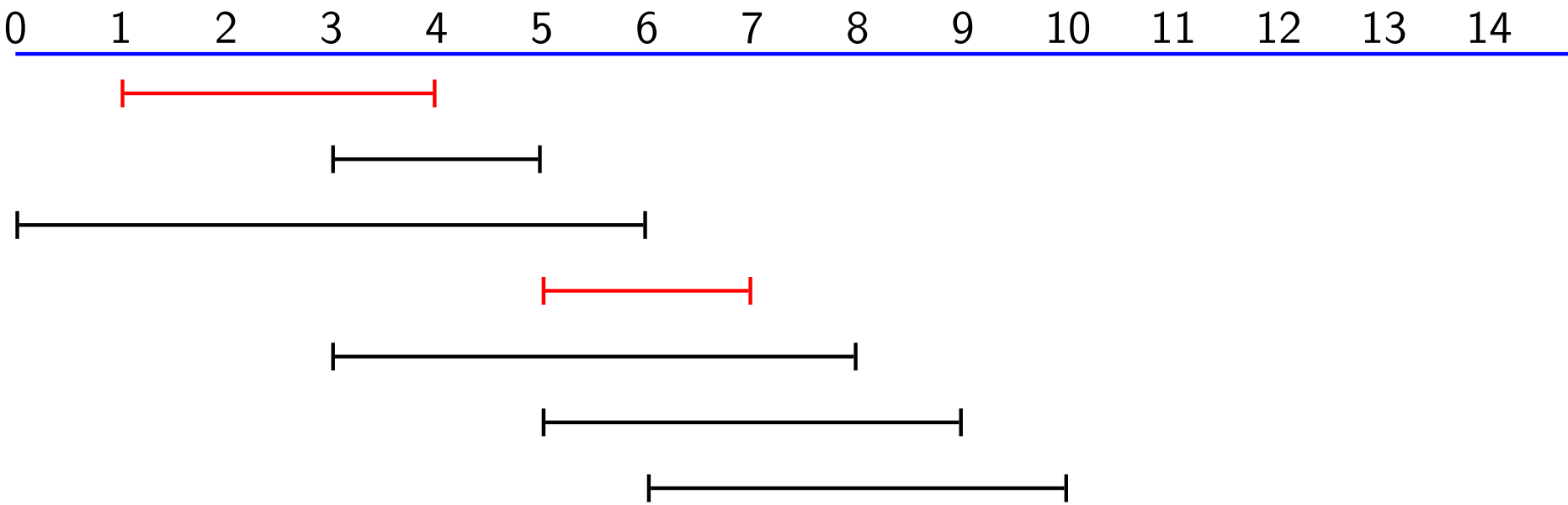
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



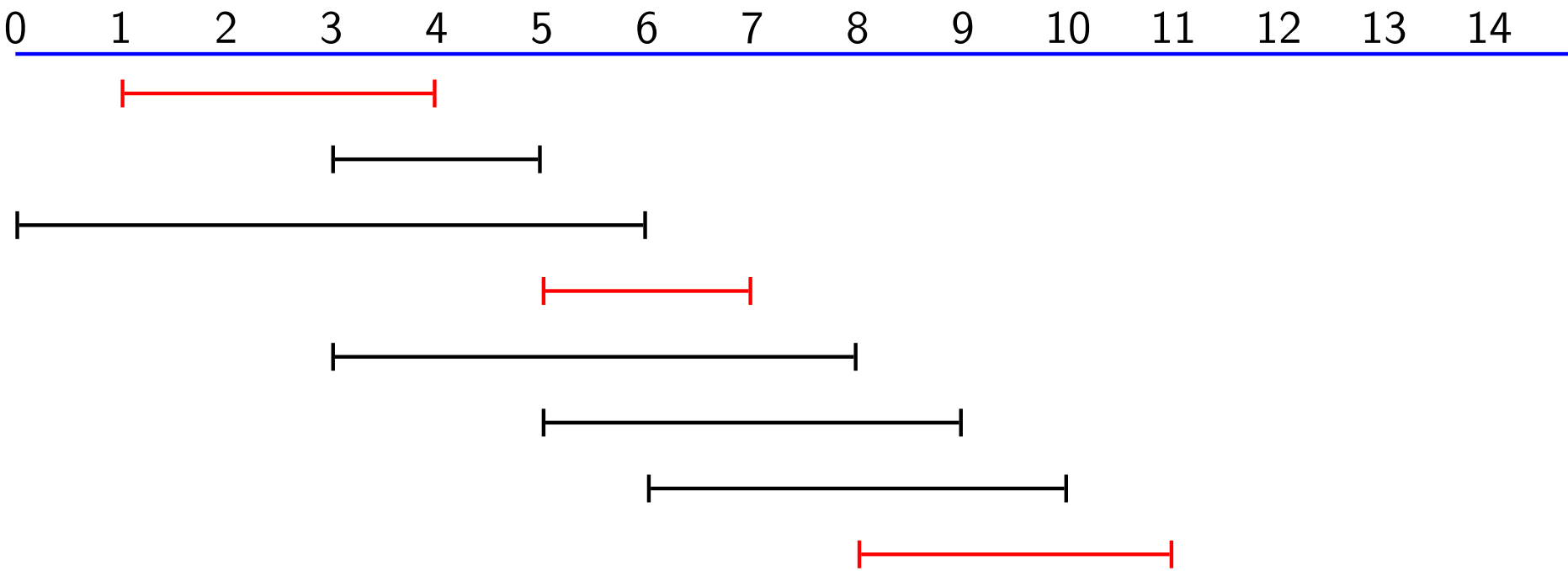
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



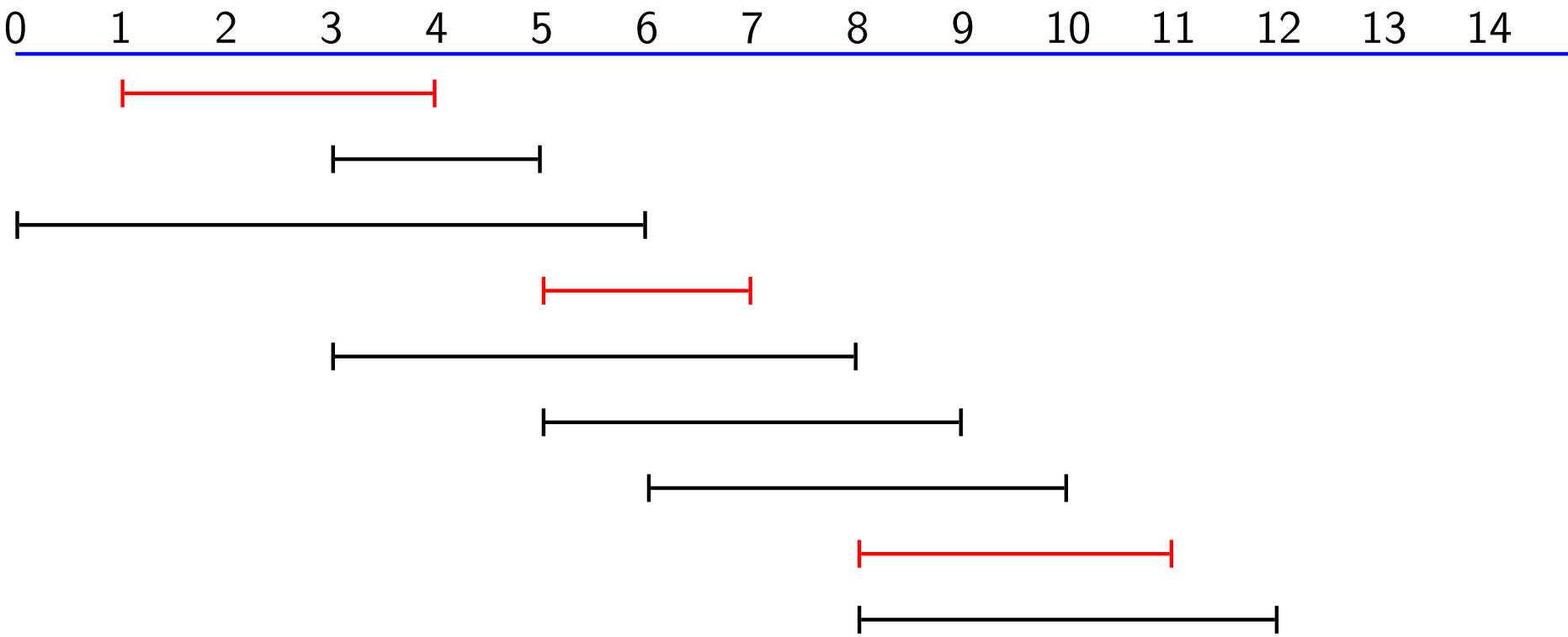
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



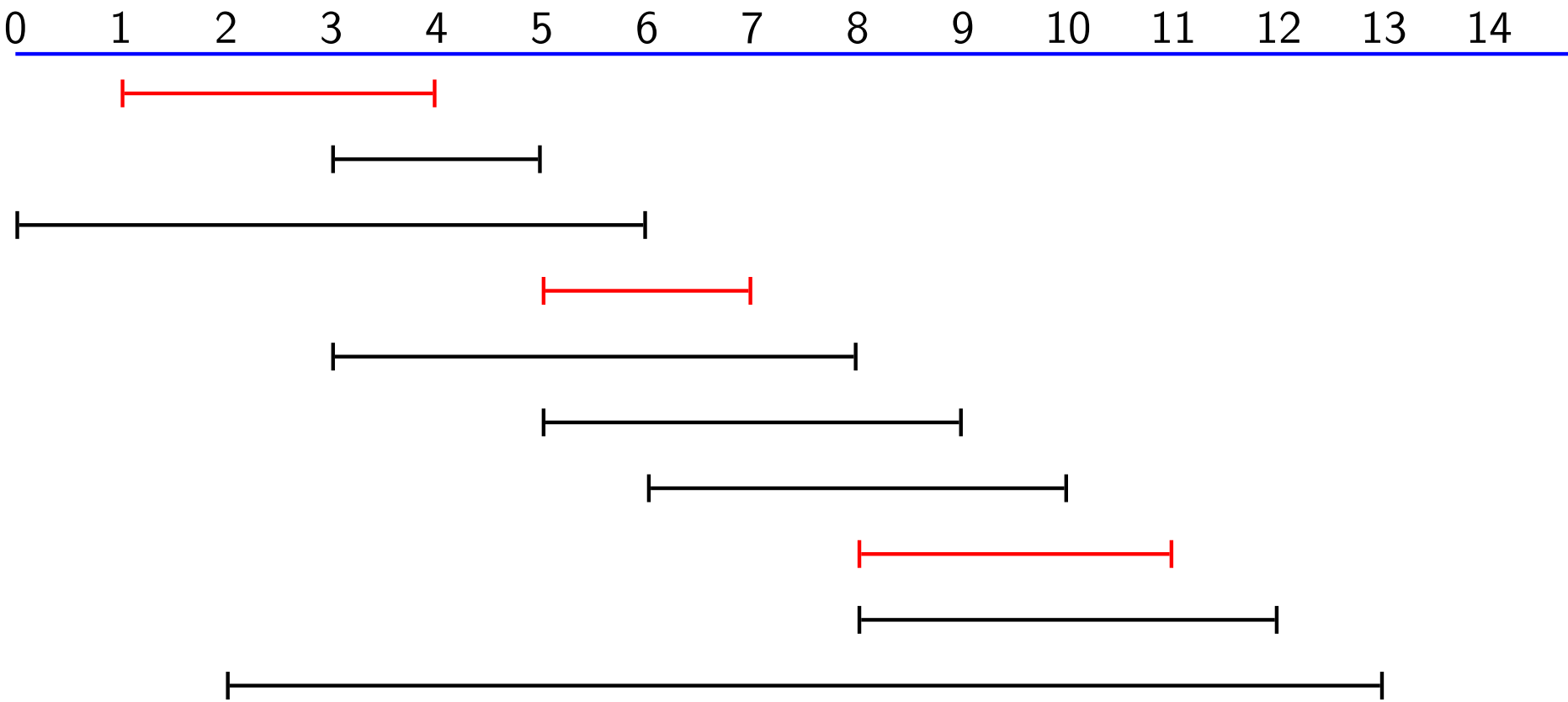
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



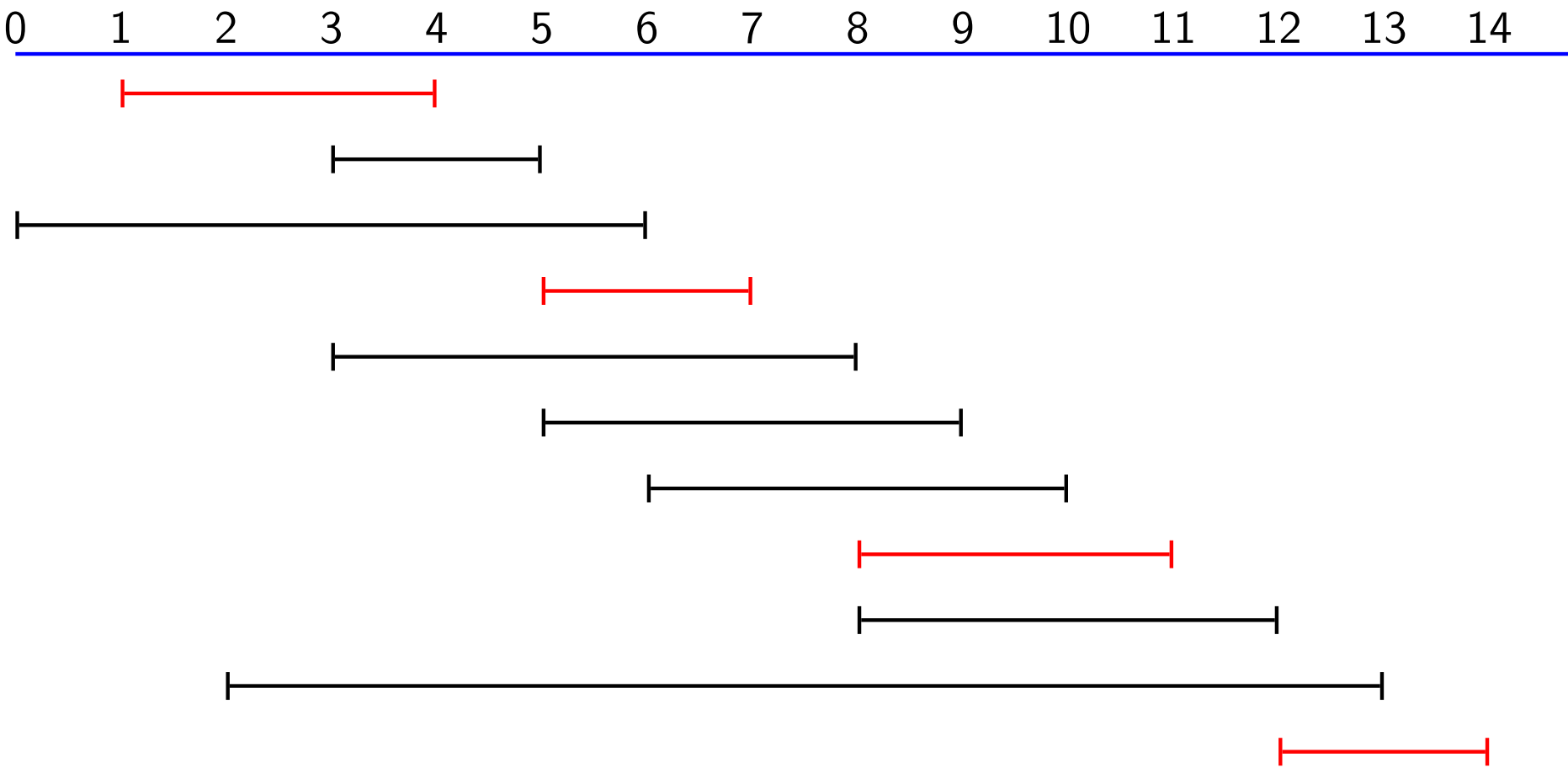
Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



Vediamo un esempio di esecuzione dell'algoritmo, in cui $n = 11$, $A = \{A_1, A_2, \ldots, A_{11}\}$, con $A_1 = [1, 4]$, $A_2 = [3, 5]$, $A_3 = [0, 6]$, $A_4 = [5, 7]$, $A_5 = [3, 8]$, $A_6 = [5, 9]$, $A_7 = [6, 10]$, $A_8 = [8, 11]$, $A_9 = [8, 12]$, $A_{10} = [2, 13]$, $A_{11} = [12, 14]$. Le attività in rosso sono quelle selezionate dall'algoritmo.



`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.

`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.

Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio.

`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione.

`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

- Siano $A_{i_1}, \ldots, A_{i_k}$ le attività selezionate da `G_A_S`.

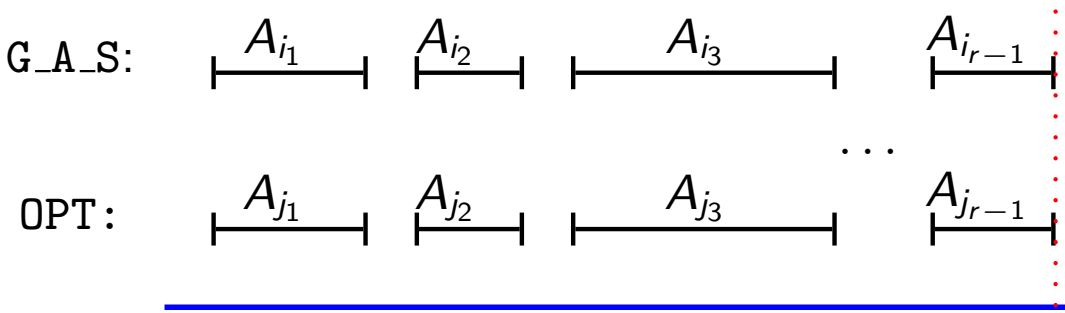
`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

- Siano $A_{i_1}, \ldots, A_{i_k}$ le attività selezionate da `G_A_S`.
- Siano $A_{j_1}, \ldots, A_{j_m}$ le attività in una *soluzione ottima* (`OPT`), con $r-1$ il **più grande** intero per cui $A_{i_1} = A_{j_1}, \ldots, A_{i_{r-1}} = A_{j_{r-1}}$ ($r-1$ può anche essere $= 0$).

`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

- Siano $A_{i_1}, \ldots, A_{i_k}$ le attività selezionate da `G_A_S`.
- Siano $A_{j_1}, \ldots, A_{j_m}$ le attività in una *soluzione ottima* (`OPT`), con $r-1$ il **più grande** intero per cui $A_{i_1} = A_{j_1}, \ldots, A_{i_{r-1}} = A_{j_{r-1}}$ ($r-1$ può anche essere $= 0$).

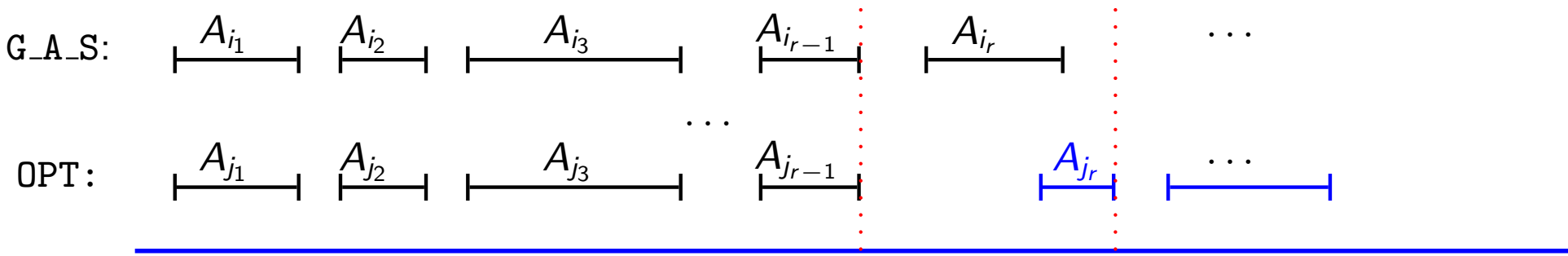
Ovvero, siamo in una situazione siffatta:



`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

- Siano $A_{i_1}, \ldots, A_{i_k}$ le attività selezionate da `G_A_S`.
- Siano $A_{j_1}, \ldots, A_{j_m}$ le attività in una *soluzione ottima* (`OPT`), con $r-1$ il **più grande** intero per cui $A_{i_1} = A_{j_1}, \ldots, A_{i_{r-1}} = A_{j_{r-1}}$ ($r-1$ può anche essere $= 0$).

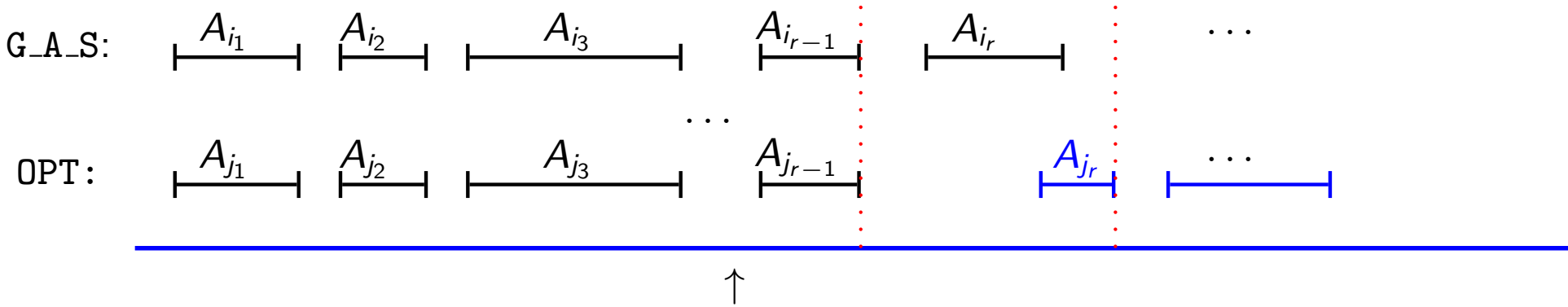
mentre



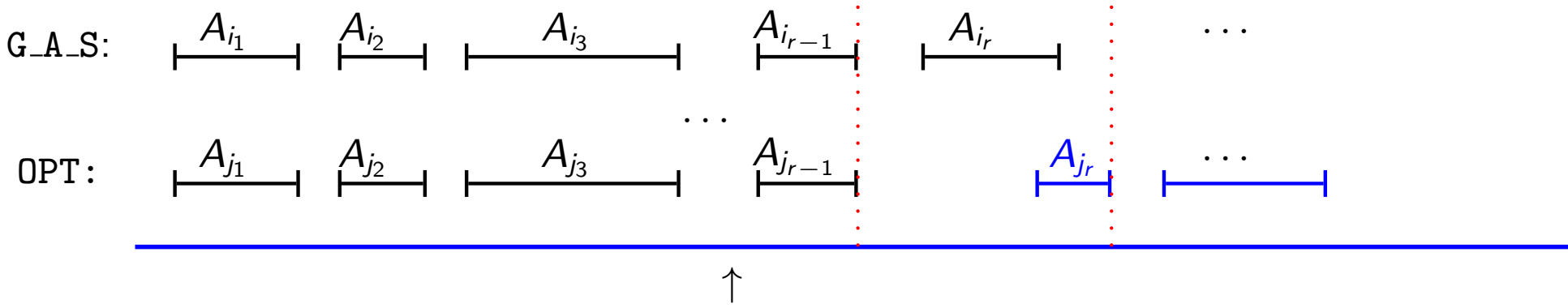
`Greedy_Activity_Selector` (`G_A_S`) produce un sottoinsieme S di attività a due a due compatibili, di cardinalità massima.
Prova. Che le attività scelte da `G_A_S` siano mutualmente compatibili è ovvio. Per la prova che l'insieme prodotto dall'algoritmo è di cardinalità massima, procediamo per contraddizione. Supponiamo che la soluzione prodotta da `G_A_S` **non** sia quella di di cardinalità massima, che denotiamo con `OPT`.

- Siano $A_{i_1}, \ldots, A_{i_k}$ le attività selezionate da `G_A_S`.
- Siano $A_{j_1}, \ldots, A_{j_m}$ le attività in una *soluzione ottima* (`OPT`), con $r-1$ il **più grande** intero per cui $A_{i_1} = A_{j_1}, \ldots, A_{i_{r-1}} = A_{j_{r-1}}$ ($r-1$ può anche essere $= 0$).

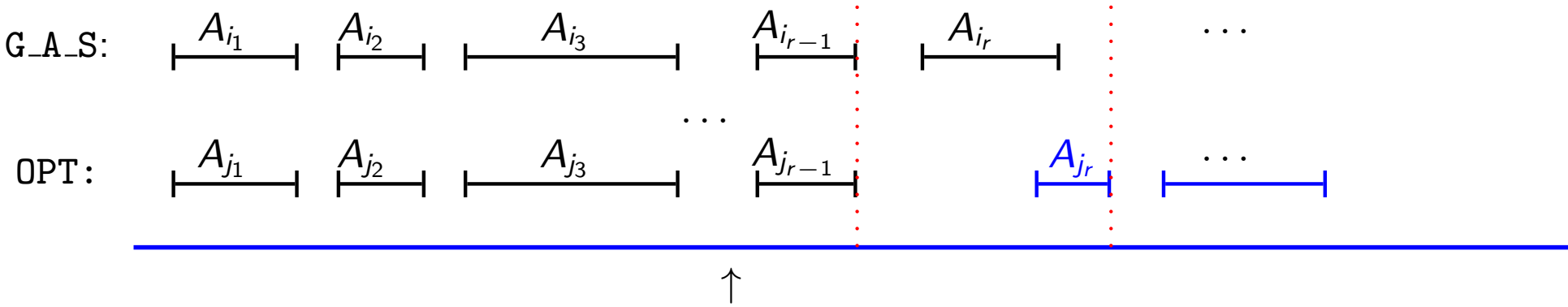
mentre



Per come funziona l'algoritmo `G_A_S`, A_{i_r} termina *prima* di A_{j_r}

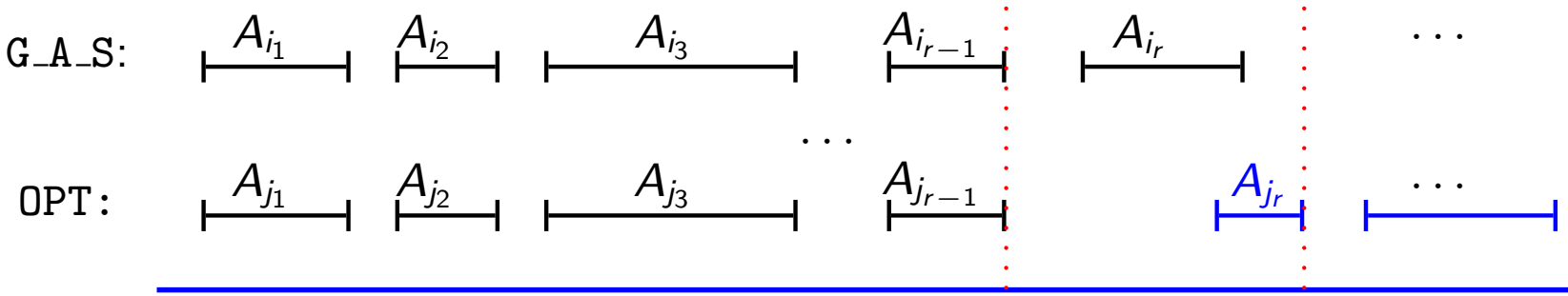


Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}



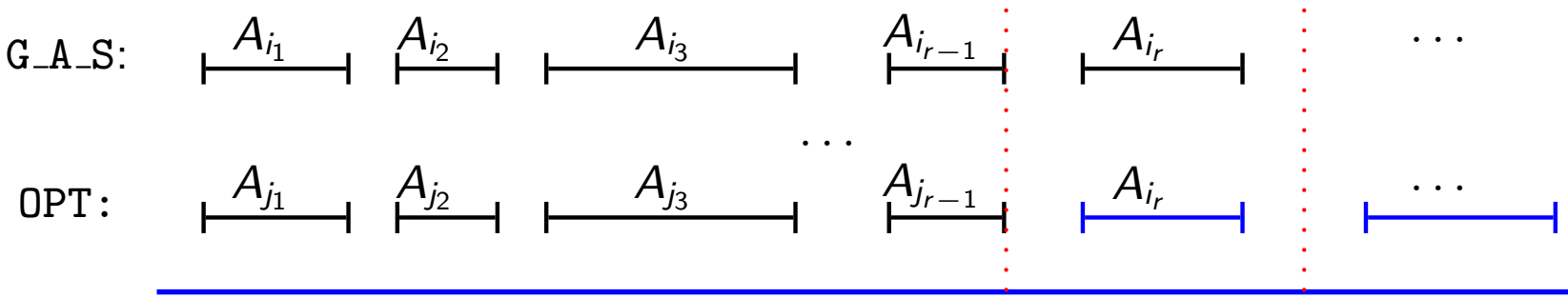
Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}

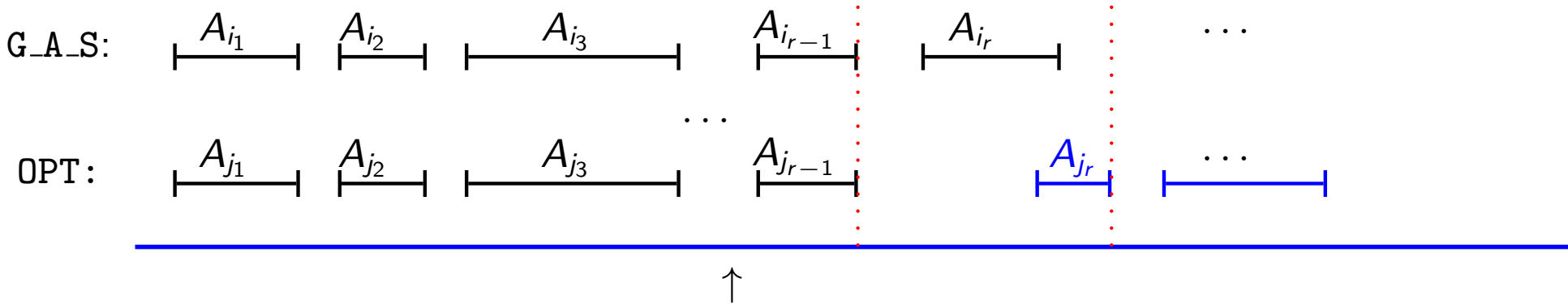
Perchè allora non sostituiamo l'attività A_{j_r} in OPT con A_{i_r}?



Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}

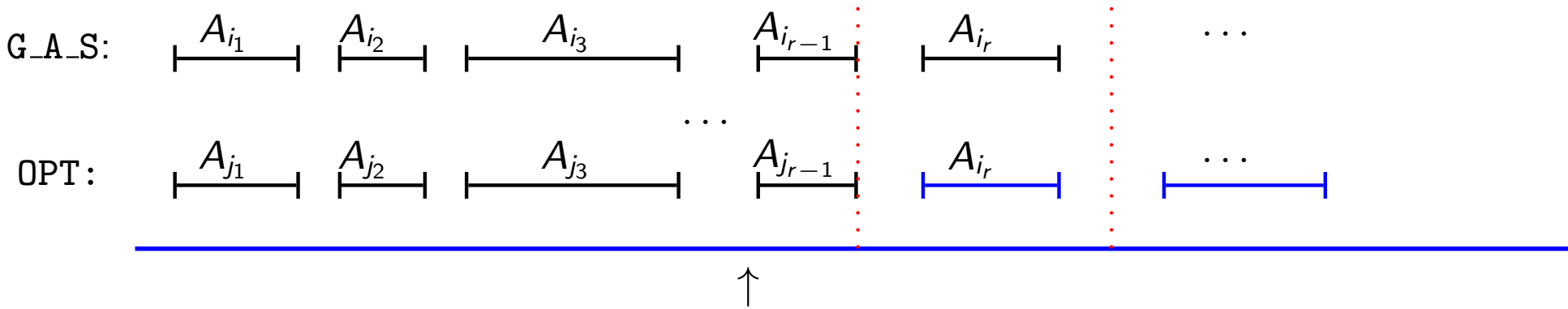
Perchè allora non sostituiamo l'attività A_{j_r} in OPT con A_{i_r}? Facciamolo, ed otteniamo



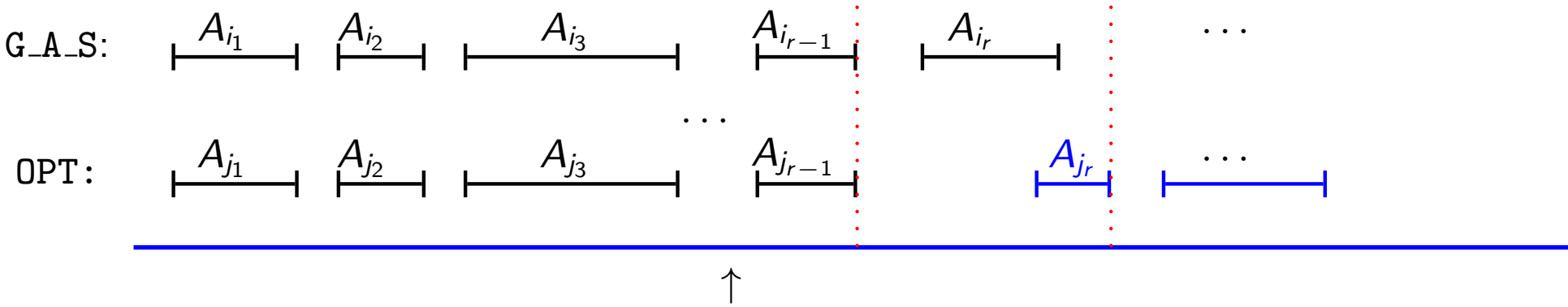


Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}

Perchè allora non sostituiamo l'attività A_{j_r} in OPT con A_{i_r}? Facciamolo, ed otteniamo

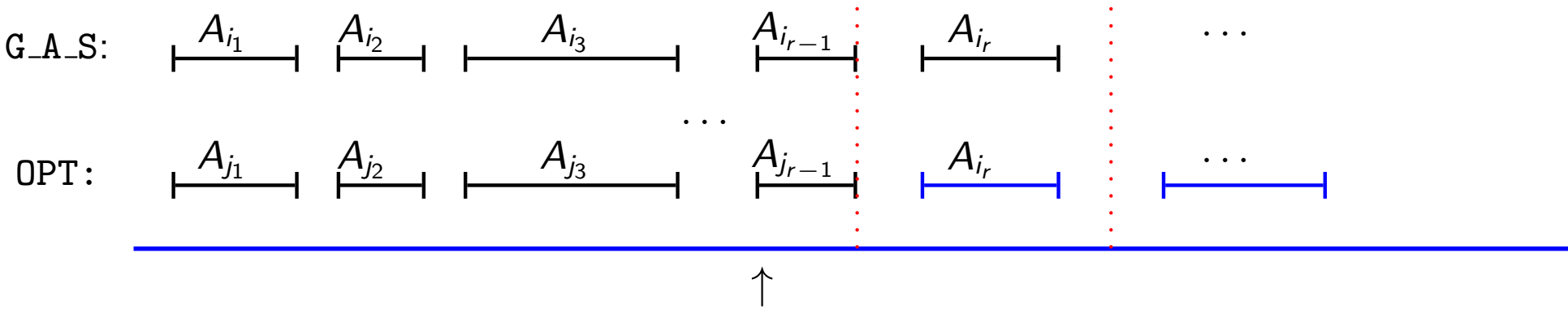


La soluzione OPT rimane *compatibile*,

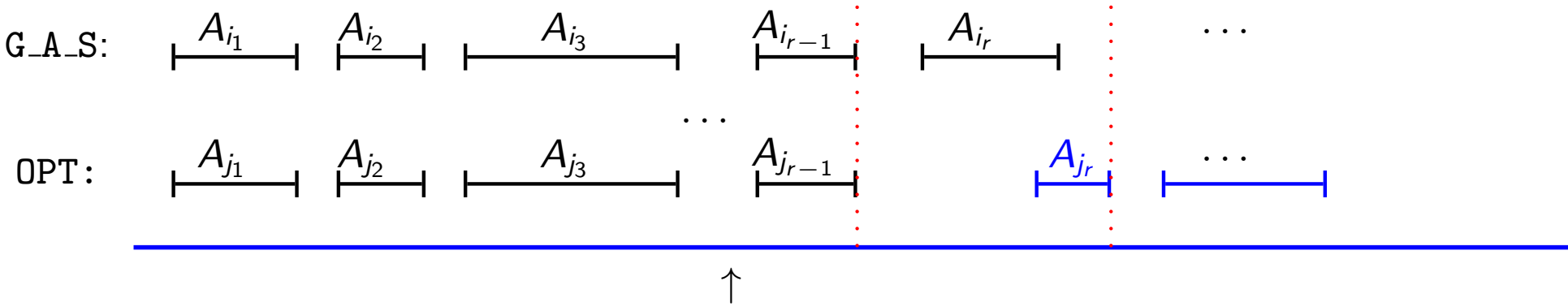


Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}

Perchè allora non sostituiamo l'attività A_{j_r} in OPT con A_{i_r}? Facciamolo, ed otteniamo

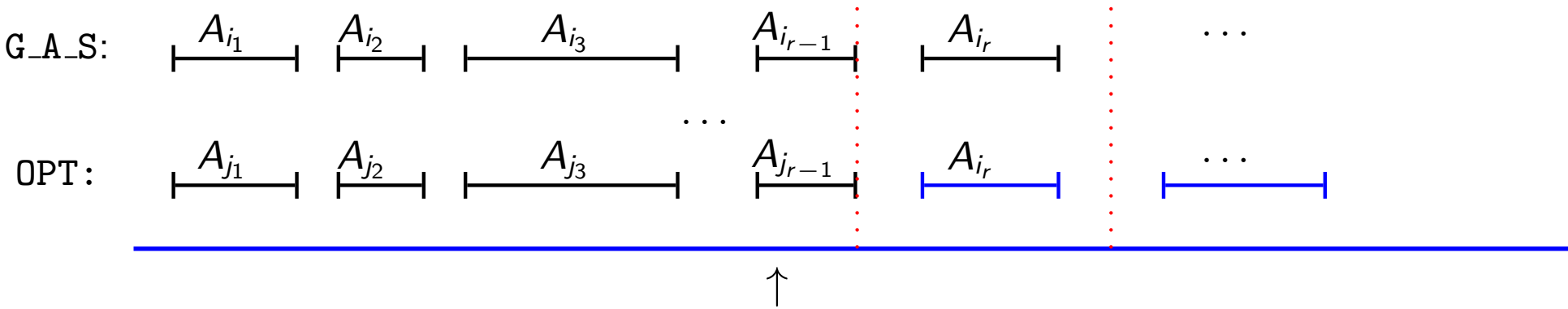


La soluzione OPT rimane *compatibile*, *ottima*,



Per come funziona l'algoritmo G_A_S, A_{i_r} termina *prima* di A_{j_r}

Perchè allora non sostituiamo l'attività A_{j_r} in OPT con A_{i_r}? Facciamolo, ed otteniamo



La soluzione OPT rimane *compatibile*, *ottima*, ma adesso contraddice la massimalità di $r-1$, **assurdo**! Con questo, il risultato è provato.

L'algoritmo `G_A_S` può **non** produrre soluzioni ottime nel caso in cui le attività abbiano associato un valore ed occorre selezionare un sottoinsieme di attività compatibili di valore totale *massimo*.

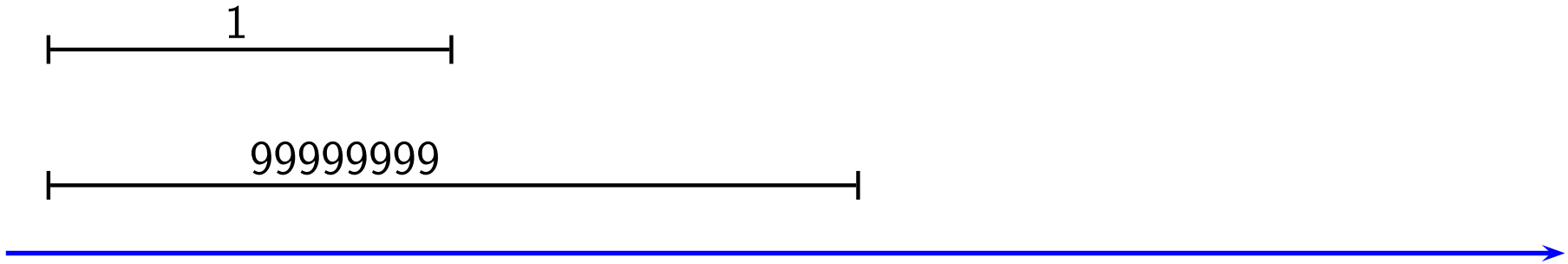
L'algoritmo `G_A_S` può **non** produrre soluzioni ottime nel caso in cui le attività abbiano associato un valore ed occorre selezionare un sottoinsieme di attività compatibili di valore totale *massimo*.

In altre parole, abbiamo provato che `G_A_S` produce soluzioni ottime solo quando tutte le attività abbiano valore $=1$.

L'algoritmo G_A_S può **non** produrre soluzioni ottime nel caso in cui le attività abbiano associato un valore ed occorre selezionare un sottoinsieme di attività compatibili di valore totale *massimo*.

In altre parole, abbiamo provato che G_A_S produce soluzioni ottime solo quando tutte le attività abbiano valore $=1$.

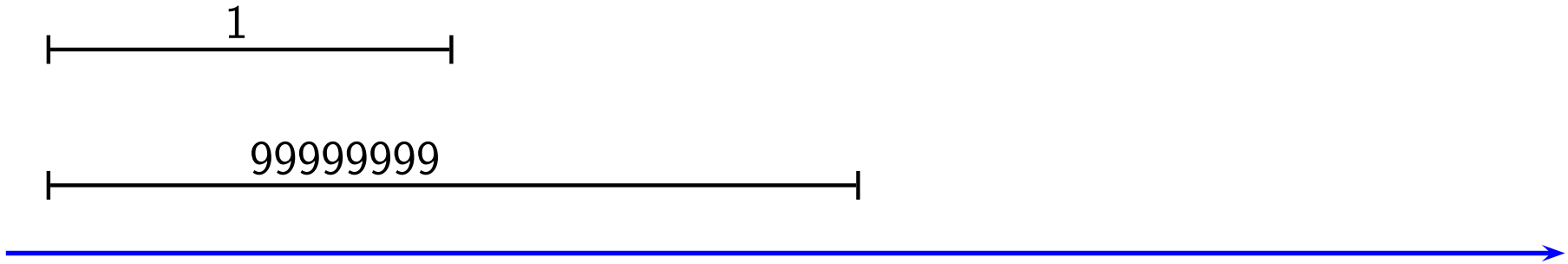
Esempio:



L'algoritmo G_A_S può **non** produrre soluzioni ottime nel caso in cui le attività abbiano associato un valore ed occorre selezionare un sottoinsieme di attività compatibili di valore totale *massimo*.

In altre parole, abbiamo provato che G_A_S produce soluzioni ottime solo quando tutte le attività abbiano valore $=1$.

Esempio:



La soluzione prodotta da G_A_S ha valore totale 1, quella di valore ottimo ha valore 99999999. Questo è il motivo per cui abbiamo usato PD per risolvere il problema delle attività quando ciascuna di esse ha un arbitrario valore.