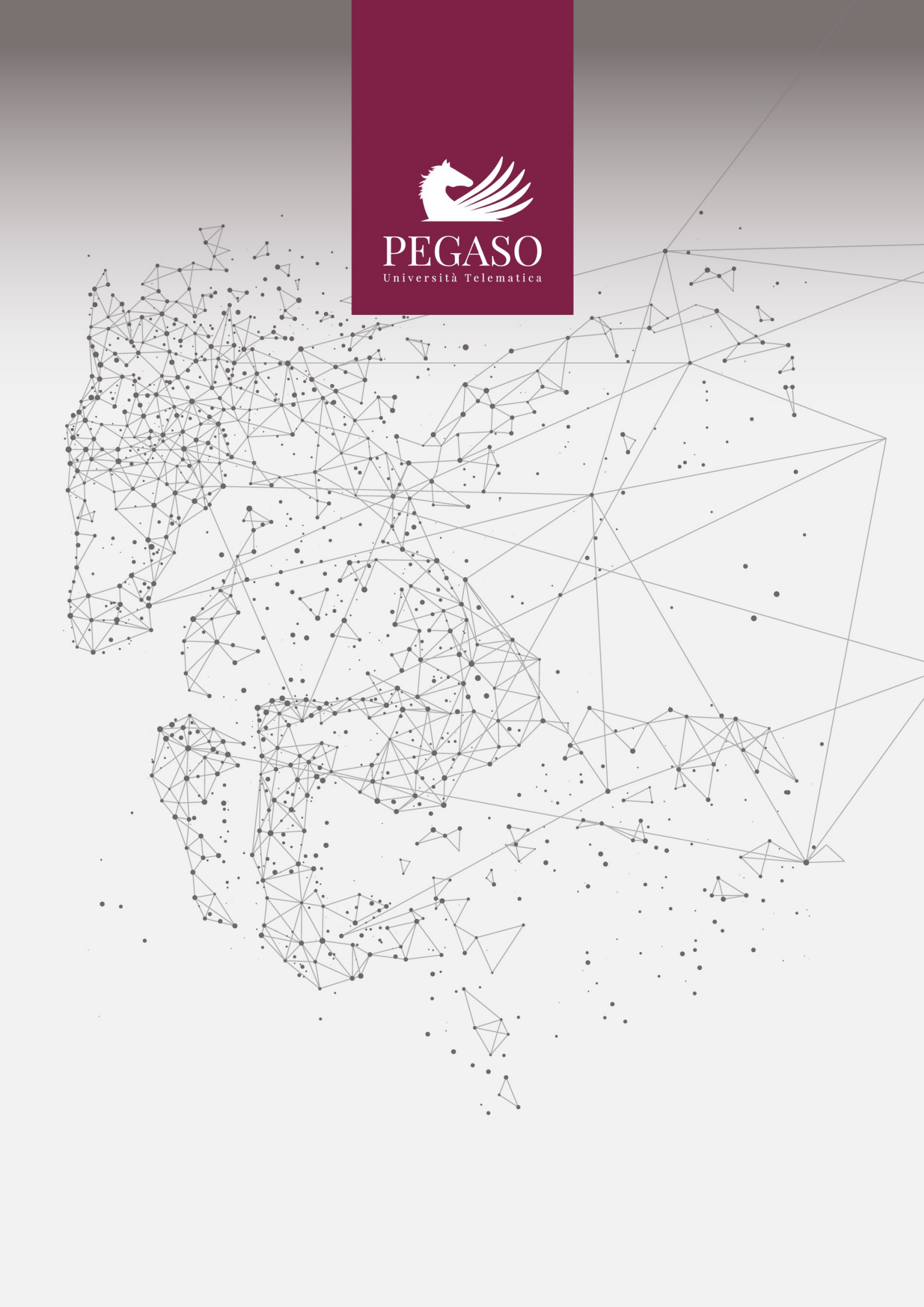

PEGASO
Università Telematica

Indice

1. Tecniche di risoluzione di un problema

Quando dobbiamo affrontare un problema possiamo approcciare diverse tecniche di risoluzione:

- Divide-et-impera
- Programmazione dinamica / memoization
- Tecnica greedy
- Ricerca locale
- Backtrack
- Algoritmi probabilistici
- Tecniche di soluzione per problemi intrattabili

Focalizziamo la nostra attenzione sulla **tecnica greedy**.

Gli algoritmi golosi (greedy algorithms) sono una tecnica di risoluzione di problemi di ottimizzazione. In generale, un problema di ottimizzazione si riferisce alla ricerca della soluzione migliore o ottimale tra un insieme di soluzioni possibili: questo può essere definito come la massimizzazione o la minimizzazione di una certa funzione obiettivo, che rappresenta il criterio di valutazione per le diverse soluzioni possibili.

L'idea alla base della tecnica golosa è quella di prendere decisioni localmente ottimali in modo da raggiungere un risultato globalmente ottimale.

Gli algoritmi golosi sono spesso utilizzati quando la soluzione del problema può essere costruita passo dopo passo, scegliendo ogni volta l'opzione che sembra migliore in quel momento, senza considerare le conseguenze a lungo termine di quella scelta.

Esempi comuni di problemi che possono essere risolti con gli algoritmi golosi includono la selezione degli oggetti da mettere in uno zaino (problema dello zaino), la scelta del percorso più breve tra due punti (problema del cammino minimo), la scelta delle attività da svolgere in un determinato periodo di tempo (problema delle attività), la scelta degli intervalli di tempo in cui programmare determinate attività (problema degli intervalli).

Un esempio concreto di utilizzo degli algoritmi golosi è il problema del cambio, in cui si cerca di restituire il numero minimo di monete necessarie per restituire un certo importo. In questo caso, l'algoritmo goloso sceglie in ogni momento la moneta di valore più alto disponibile che sia compatibile con l'importo da restituire, fino a quando non si raggiunge l'importo desiderato.

Questo approccio porta alla soluzione ottimale del problema, ovvero il numero minimo di monete necessarie per restituire l'importo, purché le monete disponibili siano tutte multipli l'una dell'altra. Se ci sono monete con valori non multipli l'uno dell'altro, l'algoritmo goloso non è più ottimale.

Consideriamo la conversione da decimale a binario: tale conversione utilizza in un certo senso una tecnica golosa; si parte infatti dalla potenza di 2 più grande che non supera il numero decimale e si procede verso destra, utilizzando potenze di 2 decrescenti.

Per esempio, per convertire il numero decimale 13 in binario, si comincia cercando la potenza di 2 più grande che non supera 13, che è $2^3 = 8$. Poiché 8 è minore di 13, si scrive un 1 sotto la colonna di 2^3 e si sottrae 8 da 13, ottenendo 5. Si procede poi cercando la potenza di 2 più grande che non supera 5, che è 2^2 = 4. Poiché 4 è minore di 5, si scrive un 1 sotto la colonna di 2^2 e si sottrae 4 da 5, ottenendo 1. Si prosegue poi cercando la potenza di 2 più grande che non supera 1, che è 2^0 = 1. Poiché 1 è uguale a 1, si scrive un 1 sotto la colonna di 2^0. Quindi il numero decimale 13 in binario è 1101.

In questo caso, la tecnica "golosa" utilizzata è quella di scegliere la potenza di 2 più grande possibile in ogni passaggio, in modo da ottenere il bit più significativo del numero binario.

Tuttavia, è importante notare che gli algoritmi golosi non sempre portano alla soluzione ottimale del problema. A volte, possono condurre ad una soluzione sub-ottimale o addirittura errata. Per questo motivo, è importante valutare attentamente se la tecnica golosa è appropriata per il problema specifico che si vuole risolvere.

Se confrontiamo la programmazione dinamica con la tecnica golosa, possiamo dire:

- **Obiettivo**: entrambe le tecniche sono utilizzate per risolvere problemi di ottimizzazione. L'obiettivo è quello di trovare la soluzione ottimale che massimizza o minimizza una certa funzione obiettivo.
- **Approccio**: la tecnica golosa e la programmazione dinamica utilizzano approcci diversi per risolvere i problemi di ottimizzazione. L'approccio goloso consiste nel prendere decisioni localmente ottimali in ogni passaggio, senza considerare le conseguenze a lungo termine di quelle decisioni. L'approccio della programmazione dinamica, invece, consiste nel suddividere il problema in sotto-problemi più piccoli, risolvendo ogni sotto-problema una sola volta e memorizzando la soluzione per poterla riutilizzare nei passaggi successivi.
- **Scelte**: nella tecnica golosa, si sceglie la soluzione che sembra la migliore in quel momento, senza considerare le conseguenze a lungo termine di quella scelta. Nella programmazione dinamica, invece, si considerano tutte le scelte possibili per ogni sotto-problema, memorizzando le soluzioni ottenute per poterle utilizzare nei passaggi successivi.

- **Complessità**: la complessità dell'algoritmo goloso dipende dalle decisioni prese in ogni passaggio e può essere calcolata in modo relativamente semplice. La complessità della programmazione dinamica dipende dal numero di sotto-problemi e dalla loro dimensione, e può essere calcolata utilizzando tecniche di analisi di complessità come la tabella delle chiamate.
- **Precisione**: la tecnica golosa non garantisce sempre la soluzione ottimale del problema, ma spesso produce soluzioni sub-ottimali accettabili in tempi molto rapidi. La programmazione dinamica, invece, garantisce sempre la soluzione ottimale del problema, ma richiede più tempo e memoria per eseguire l'algoritmo.

In sintesi, la tecnica golosa è una tecnica semplice e veloce per risolvere problemi di ottimizzazione, che tuttavia non garantisce sempre la soluzione ottimale del problema. La programmazione dinamica, invece, è una tecnica più complessa e costosa in termini di tempo e memoria, ma che garantisce sempre la soluzione ottimale del problema. La scelta tra le due tecniche dipende dalle caratteristiche del problema specifico e dalle esigenze dell'applicazione.

La domanda che ci dobbiamo porre è: quando applicare la tecnica greedy? Se è possibile dimostrare che esiste una scelta ingorda, fra le molte scelte possibili, ne può essere facilmente individuata una che porta sicuramente alla soluzione ottima; se il problema ha sottostruttura ottima, fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale.

Da notare che comunque, in alcuni casi, soluzioni ingorde non ottime possono essere comunque interessanti.

2. Insieme indipendente massimale

Sia $S = \{1, 2, \ldots, n\}$ un insieme di intervalli della retta reale. Ogni intervallo $[a_i, b_i[$, con $i \in S$, è chiuso a sinistra e aperto a destra.

- a_i: tempo di inizio
- b_i: tempo di fine

Consideriamo il seguente set di intervalli:

i	ai	bi
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

Rappresentiamoli da un punto di vista grafico:

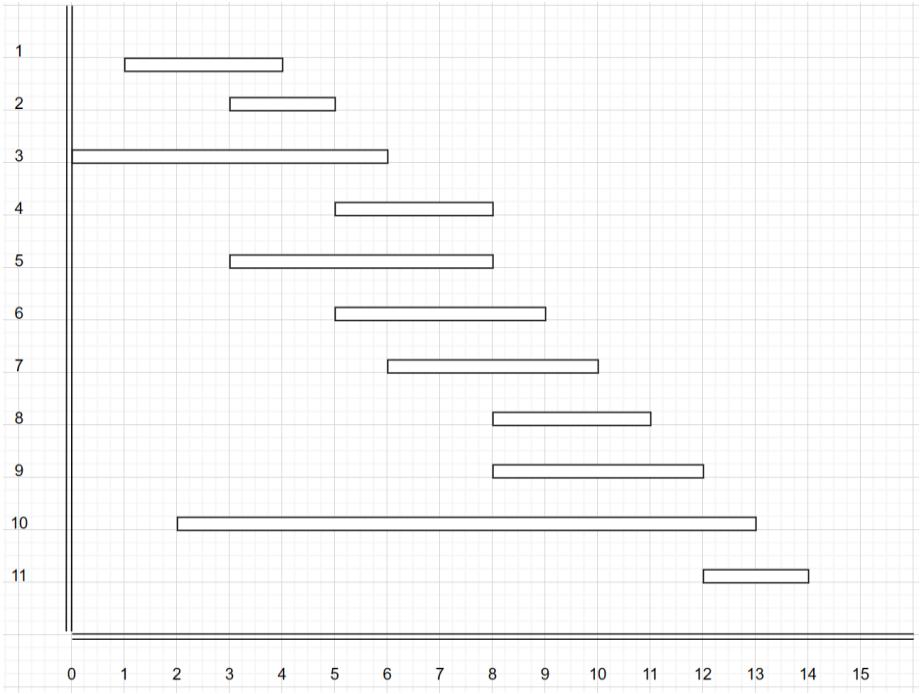


Figura 1: Intervalli

Di seguito l'algoritmo per implementare una tecnica golosa:

```
Set independentSet(int[] a, int[] b)
{ ordina a e b in modo che b[1] ≤ b[2] ≤ · · · ≤ b[n] }
Set S = Set()
S.insert(1)
int last = 1                    % Ultimo intervallo inserito
for i = 2 to n
     if a[i] ≥ b[last] then     % Controllo indipendenza
          S.insert(i)
          last = i
return S
```

La complessità dell'algoritmo è:

- $O(n \log n)$ se input non è ordinato
- $O(n)$ se l'input è già ordinato

Applichiamo l'algoritmo allo scenario precedente; partiamo dunque con la scelta dell'attività 1 e settiamo last pari ad 1:

- scelta: {1}
- last: 1

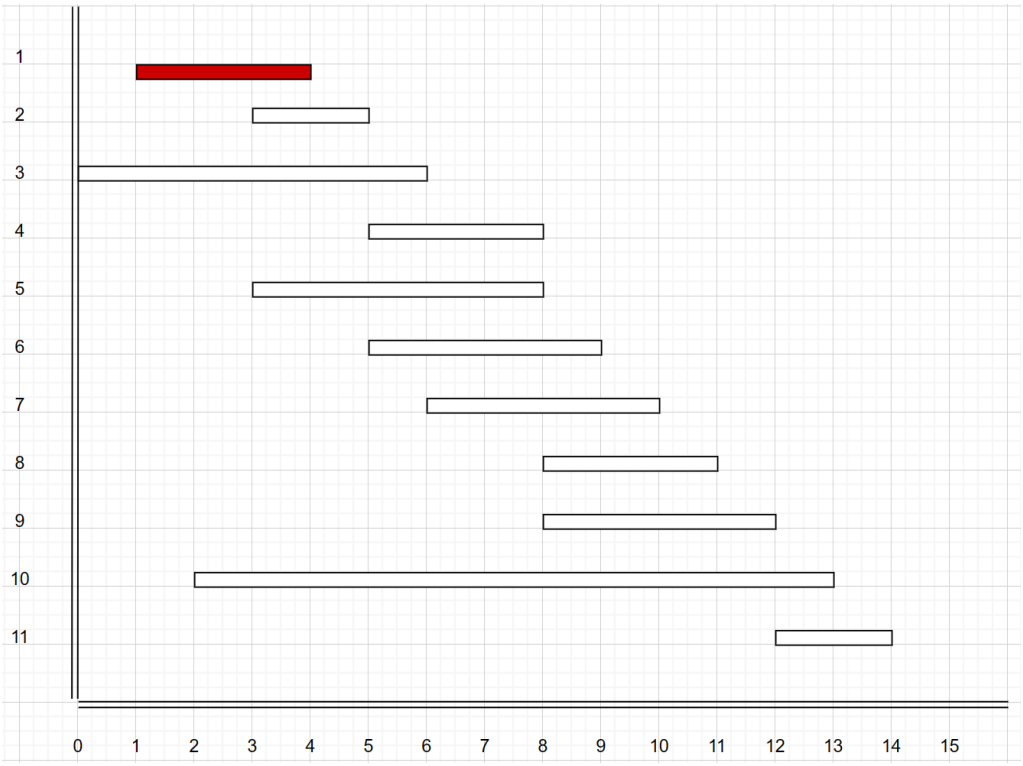
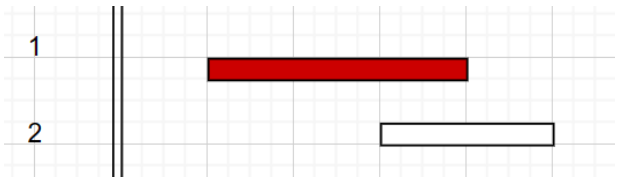
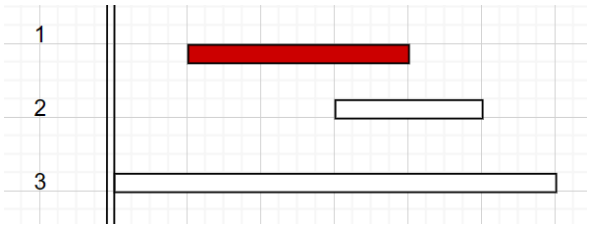


Figura 2: Scelta: attività 1

Proseguendo nell'algoritmo, abbiamo che per $i = 2$ dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 2 è maggiore o uguale della fine dell'attività 1; come si evince dalla figura, non lo è:

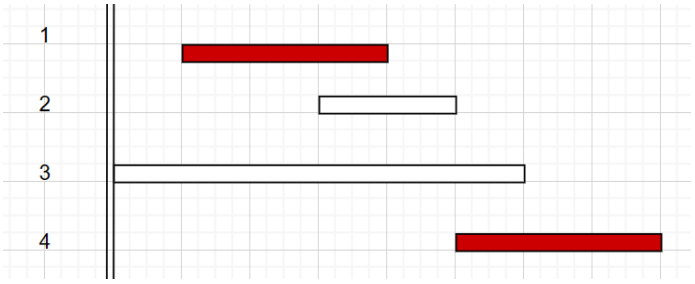


Pertanto, proseguiamo con $i = 3$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 3 è maggiore o uguale della fine dell'attività 1; come si evince dalla figura, non lo è;

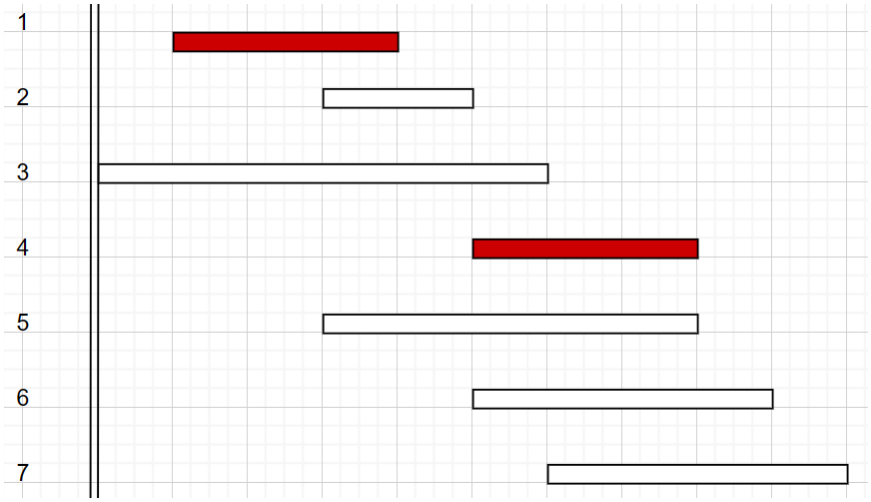


Proseguiamo con $i = 4$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 4 è maggiore o uguale della fine dell'attività 1; come si evince dalla figura lo è; pertanto, scegliamo l'attività 4 ed aggiorniamo last a 4:

- scelta: {1, 4}
- last: 4

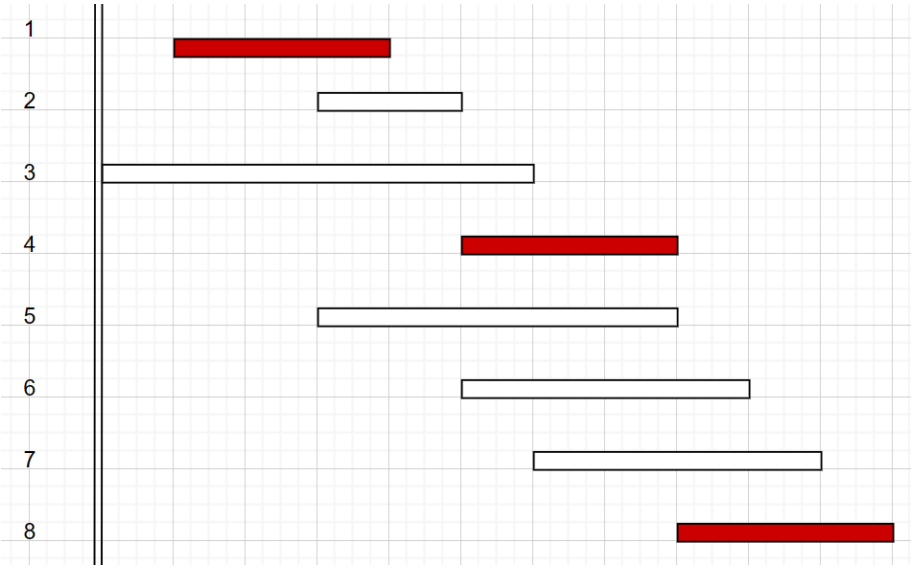


Proseguiamo con $i = 5$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 5 è maggiore o uguale della fine dell'attività 4; come si evince dalla figura non lo è, così come non lo sono le attività 6 e 7:

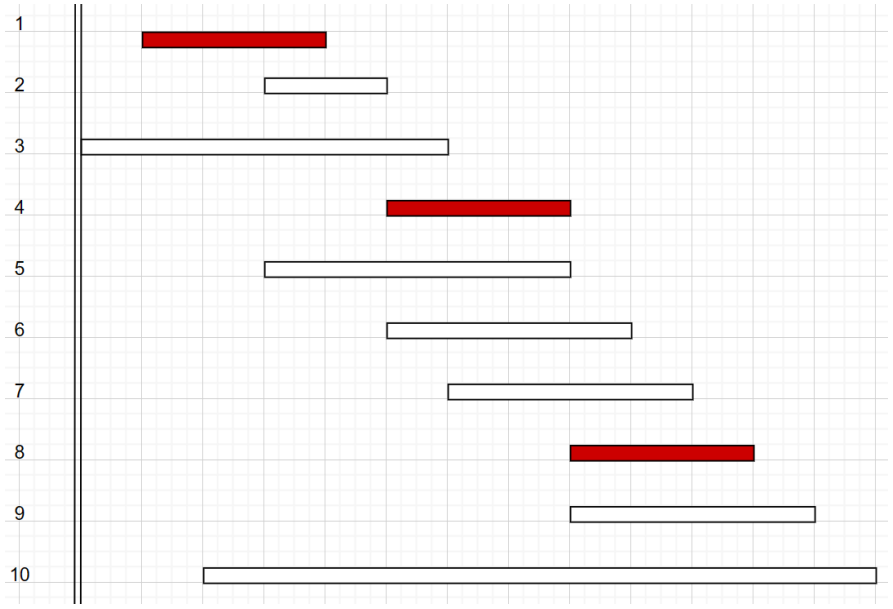


Proseguiamo con $i = 8$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 8 è maggiore o uguale della fine dell'attività 4; come si evince dalla figura lo è; pertanto, scegliamo l'attività 8 ed aggiorniamo last a 8:

- scelta: {1, 4, 8}
- last: 8



Proseguiamo con $i = 9$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 9 è maggiore o uguale della fine dell'attività 8; come si evince dalla figura non lo è, così come non lo è l'attività 10.



Proseguiamo con $i = 11$; dobbiamo verificare se $a[i] \geq b[last]$, cioè se il tempo di inizio dell'attività 11 è maggiore o uguale della fine dell'attività 8; come si evince dalla figura lo è; pertanto, scegliamo l'attività 11 ed aggiorniamo last a 11; in realtà abbiamo terminato.

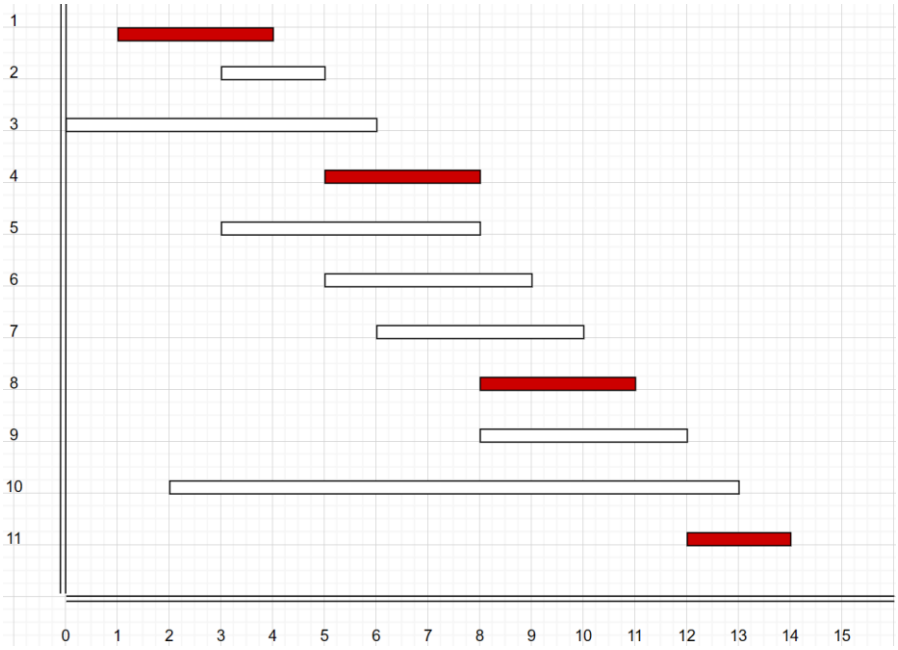


Figura 3: Scelta finale

L'idea alla base dell'algoritmo è dunque quella di selezionare gli intervalli uno alla volta in ordine crescente di tempo di fine, e di eliminare tutti gli intervalli che si sovrappongono con l'intervallo selezionato. In altre parole, in ogni iterazione viene selezionato l'intervallo con il tempo di fine più basso, e viene eliminato insieme a tutti gli intervalli che si sovrappongono ad esso.

In questo modo, l'algoritmo garantisce che l'insieme selezionato sia sempre il più grande possibile di intervalli disgiunti. Tuttavia, anche in questo caso, ci sono situazioni in cui l'algoritmo può fallire, ad esempio quando ci sono intervalli che si sovrappongono perfettamente, ovvero con lo stesso tempo di inizio e fine.

In generale, l'algoritmo dell'insieme indipendente massimale è un approccio ingordo efficiente per risolvere questo problema degli intervalli sulla retta reale. Tuttavia, come per ogni algoritmo ingordo, la soluzione trovata potrebbe non essere la migliore in assoluto, ma solo la migliore tra le scelte localmente ottimali.

3. Resto

Definiamo il seguente problema del Resto (Money Change):

Dato un insieme di "tagli" di monete, memorizzati in un vettore di interi positivi $t[1 \dots n]$ ed un intero R rappresentante il resto che dobbiamo restituire, il problema del Money Change consiste nel trovare il più piccolo numero intero di pezzi necessari per dare un resto di R centesimi utilizzando i tagli disponibili (assumiamo di avere un numero illimitato di monete per ogni taglio).

Da un punto di vista formale questo significa trovare un vettore x di interi non negativi tale che:

$$R = \sum_{i=1}^{n} x[i] \cdot t[i]$$

$$m = \sum_{i=1}^{n} x[i] \ \ ha\ valore\ minimo$$

Applicando la tecnica della programmazione dinamica:

```
int[] moneyChange(int[] t, int n, int R)
int[] DP = new int[0 ...R]
int[] coin = new int[0 ...R]
DP[0] = 0
for i = 1 to R
     DP[icatt] = +∞
     for j = 1 to n
          if i > t[j] and DP[i – t[j]] + 1 < DP[i]
               DP[i] = DP[i – t[j]] + 1
               coin[i] = j
int[] x = new int[1 ...n] = {0}
while R > 0
     x[coin[R]] = x[coin[R]] + 1
     R = R – t[coin[R]]
return x
```

La complessità di questo algoritmo è $O(nR)$

L'idea è quella di inserire in $DP[i]$ il numero minimo di monete per risolvere il problema $S[i]$ (cioè il problema di dare un resto pari ad i).

Se volessimo adottare una tecnica greedy, la soluzione potrebbe essere quella di selezionare la moneta j più grande tale per cui $t[j] \leq R$, e poi risolvere il problema $S(R - t[j])$

La strategia può essere dunque la seguente:

```
int[] moneyChange(int[] t, int n, int R)
int[] x = new int[1 ...n]
{ Ordina le monete in modo decrescente }
for i = 1 to n
     x[i] = floor(R/t[i])
     R = R – x[i] · t[i]
return x
```

La complessità è:

- $O(n \log n)$ se input non è ordinato
- $O(n)$ se l'input è già ordinato

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.