1) Il problema della selezione delle attività prevede l'organizzazione di una serie di impegni con tempi di inizio e fine dedicati tali che siano compatibili tra di loro (riusciamo a combinare attività in maniera tale che non si intersechino tra di loro; quando finisce una, ne inizia almeno subito un'altra).

Matematicamente:

Vediamo un esempio classico di applicazione di un algoritmo greedy: la selezione di attività compatibili.

Abbiamo:

- risorsa condivisa (e.g. aula);
- insieme di attività $S = \{a_i : 1 \leq i \leq n\}$

  $a_i = [s_i, f_i), \quad 0 \leq s_i \leq f_i \quad (s_i = \text{tempo di inizio}, f_i = \text{tempo di fine})$

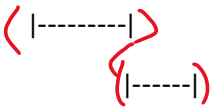
Def Diciamo che a_i e a_j sono **compatibili** sse

$$[s_i, f_i) \cap [s_j, f_j) = \emptyset$$

Equivalentemente

$$f_i \leq s_j \text{ oppure } f_j \leq s_i$$

2)



L'algoritmo si basa su una scelta greedy: date le attività, consideriamo la prima come ottima (perché inizia almeno da un punto >=0) e inseriamo tutte le altre attività con tempo di inizio >= all'attività subito precedente (quella greedy), costruendo così un insieme di attività ottimo.

```
GREEDY-SEL(S, f)
1  n = S.length
2  A = {a_1}
3  last = 1 // indice dell'ultima attività selezionata
4  for m = 2 to n
5      if s_m ≥ f_last
6          A = A ∪ {a_m}
7          last = m
8  return A
```

3)

Esempi non ottimi misto ad altro esempio ottimo (= attività scelta per ultima)

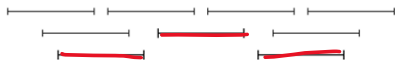
Oltre alla scelta greedy vista precedentemente, ne esistono altre:

- Scegli l'attività di durata inferiore → non è ottima.

  Controesempio:

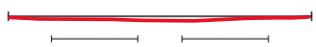
- Scegli l'attività col minor numero di sovrapposizioni → non è ottima.

  Controesempio:



- Scegli l'attività che inizia per prima → non è ottima.

  Controesempio:



- Scegli l'attività che inizia per ultima → è ottima.

Greedy-Sel-Reverse(S, f)

1 n = S.length

2 A = {a[n]}

3 first = n // indice della prima attività selezionata (partendo dalla fine)

4 for m = n-1 down to 1

5 if f[m] <= s[first]

```
6       A = A ∪ {a[m]}
7       first = m
8  return A
```

Esercizio (11 punti) Si consideri il problema della selezione delle attività compatibili. Si ha un insieme S di n attività, dove ogni attività ai ha un tempo di inizio s_i e un tempo di fine f_i. Due attività ai e aj sono compatibili se i loro intervalli di tempo non si sovrappongono, ovvero se f_i ≤ s_j o f_j ≤ s_i. L'obiettivo è selezionare il sottoinsieme di attività compatibili di cardinalità massima.

(a) Qual è la complessità dell'algoritmo esaustivo che esamina tutti i possibili sottoinsiemi di attività?

(b) Assumendo di conoscere un algoritmo che determina se due attività sono compatibili in tempo O(1), come si può modificare l'algoritmo del punto precedente per renderlo più efficiente?

(c) Progettare un algoritmo greedy efficiente per risolvere il problema. Sono richiesti:

- La strategia greedy utilizzata
- Lo pseudocodice dell'algoritmo
- La dimostrazione della correttezza (proprietà di scelta greedy e sottostruttura ottima)
- L'analisi della complessità

- Progetta un algoritmo su LCS e scrivi la ricorrenza

LCS → Due stringhe X, Y con lunghezza n

Dato l'algoritmo, dopo scrivi la relazione di ricorrenza

$$\begin{cases} 0, & i = 0, j = 0 \\ LCS(i-1, j-1) + 1, & X_i = X_j \\ \max\big(LCS(i-1, j), LCS(i, j-1)\big), & X_i \neq X_j \end{cases}$$

- Algoritmo con due array come max heap in un unico array che era max heap (con pseudocodice)

Max heap:

- Parent >= Figli

$A\left[\frac{i}{2}\right] \geq A[i]$ → Parent

```
funzione MERGE_MAX_HEAPS(heap1, heap2)
    // Assumiamo che heap1 e heap2 siano array che rappresentano max heap
    n1 = lunghezza(heap1)
    n2 = lunghezza(heap2)

    // Creiamo un nuovo array per contenere entrambi gli heap
    risultato = nuovo array di dimensione (n1 + n2)

    // Copiamo gli elementi di entrambi gli heap nel nuovo array
```

```
    per i da 0 a n1-1
        risultato[i] = heap1[i]
    per i da 0 a n2-1
        risultato[n1 + i] = heap2[i]

    // Ora abbiamo un array che contiene tutti gli elementi, ma non è un
max heap
    // Dobbiamo heapify l'intero array dal basso verso l'alto

    per i da ((n1 + n2) / 2) - 1 fino a 0 con passo -1
        MAX_HEAPIFY(risultato, n1 + n2, i)

    return risultato

funzione MAX_HEAPIFY(A, n, i)
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    se left < n e A[left] > A[largest] allora
        largest = left

    se right < n e A[right] > A[largest] allora
        largest = right

    se largest != i allora
        scambia A[i] con A[largest]
        MAX_HEAPIFY(A, n, largest)
```

```
doMerge(A,B,i,j)
      l = 2*i
      r = 2*i+1
      if(A[l] >= B[l]
            C[i] = A[l] // salvo max come posizione attuale
            C[l] = B[l] // posiziono l'altro nel livello che sto toccando
      else
            C[i] = A[r]
            C[r] = B[r]
      return C
```

- Greedy delle attività ma con l’attività scelta per ultima