# La tecnica **Greedy**

Algoritmi 2 – Lezione 10

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

# Indice


1. Algoritmi greedy
   - Definizione e proprietà
   - Esempi e correttezza
2. Il problema della selezione di attività
   - Descrizione del problema
   - Strategie Greedy sbagliate
   - La strategia greedy corretta
   - Prova di correttezza
   - Implementazione
3. Il Problema dell' Assegnazione di Attività
   - Strategia Greedy sbagliata
   - La strategia greedy corretta
   - Prova di correttezza
   - Implementazioni
     - Implementazione semplice con array
     - Implementazione efficiente con min-heap

# Introduzione agli Algoritmi Greedy

- Un algoritmo **greedy** (o avido) risolve un problema tramite una sequenza di decisioni.
- A ogni passo, l'algoritmo sceglie l'opzione che sembra la migliore in quel momento, senza considerare le conseguenze future.
- Si basa sull'assunto che una serie di scelte localmente ottimali possa portare a una soluzione globalmente ottima, un'assunzione che richiede una dimostrazione.
- **Caratteristiche Principali:**
  - **Scelta Ottimale Locale:** Si seleziona l'opzione che offre il massimo beneficio immediato.
  - **Scelta Irrevocabile:** Una volta presa una decisione, non si può tornare indietro.

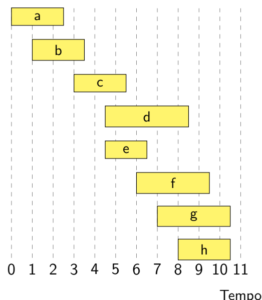
# Esempi e Prova di Correttezza

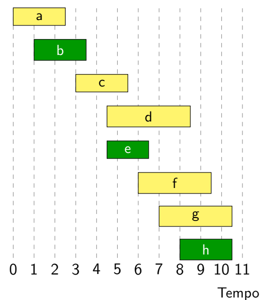
- Esempi già studiati di algoritmi greedy:
  - **Algoritmo di Dijkstra:** A ogni passo si sceglie il vertice non ancora visitato con la distanza minima.
  - **Algoritmo di Kruskal:** A ogni passo si aggiunge l'arco con il peso minimo che non forma un ciclo.
- È fondamentale dimostrarne la correttezza perché l'ottimalità locale non garantisce quella globale.
- **Argomento di Scambio :**
  - Tecnica di prova comune per dimostrare l'ottimalità.
  - Si confronta la soluzione greedy con una soluzione ottima ipotetica.
  - Si dimostra che si può "scambiare" parti della soluzione ottima per renderla identica a quella greedy senza peggiorarne la qualità.
  - Questo porta a una contraddizione dimostrando che la soluzione greedy deve essere ottima fin dall'inizio.

# Il Problema della Selezione di Attività

- Dato un insieme di attività con orario di inizio e fine, selezionare il sottoinsieme massimo di attività compatibili (che non si sovrappongono).

A sinistra è mostrata un'istanza del problema con $n = 8$ attività ( intervalli nel tempo) e sulla destra una soluzione, evidenziata in verde.

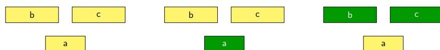


# Strategie Greedy sbagliate

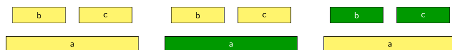
Vediamo alcune **strategie greedy plausibili ma non corrette**, ovvero che **non garantiscono una soluzione ottima**.

1. **Scegliere l'attività che dura di meno**:



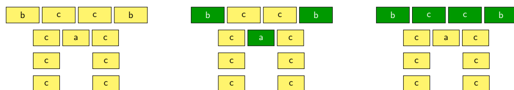

A sinistra l'istanza del problema, al centro la soluzione prodotta e a destra la soluzione ottima

2. **Scegliere l'attività che inizia prima**:




A sinistra l'istanza del problema, al centro la soluzione prodotta e a destra la soluzione ottima

3. **Scegliere l'attività che crea meno conflitti con le attività rimanenti**:




A sinistra l'istanza del problema, al centro la soluzione prodotta e a destra la soluzione ottima

# Il Criterio Corretto: Finisce per Prima

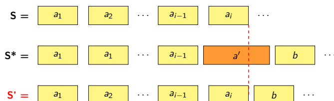
Il criterio greedy corretto per risolvere il problema è: **scegliere sempre l'attività che termina per prima** tra quelle compatibili con le attività già scelte.

1. Ordina tutte le attività in base all'orario di fine in modo crescente.
2. Inizializza un insieme di soluzioni $S$ vuoto.
3. Scansiona la lista ordinata e, per ogni attività, se è compatibile con quelle già in $S$, aggiungila.

# Prova di Correttezza (per assurdo)

- Si assume che la soluzione greedy $S$ non sia ottimale.
- Sia $S^*$ la soluzione ottima che differisce il meno possibile da $S$.
- Sia $a_i$ la prima attività in $S$ che non è in $S^*$.
- Dato che l'algoritmo greedy ha scelto $a_i$, questa deve terminare per prima tra le attività compatibili, quindi $fine(a_i) \leq fine(a')$. Dove $a'$ è la prima attività di $S^*$ che confligge con $a_i$.
- Si può costruire una nuova soluzione $S'$ scambiando $a'$ con $a_i$ in $S^*$.
- La nuova soluzione $S'$ è ancora ottimale, ma differisce meno da $S$, in contraddizione con l'ipotesi iniziale.
- L'assurdo dimostra che la soluzione greedy deve essere ottima.

La figura seguente illustra la relazione tra le soluzioni $S$ e $S^*$ e $S'$, evidenziando il conflitto tra l'attività $a_i$ e $a'$:



# Implementazione dell'algoritmo greedy

```
def selezioneFiniscePrima(A):
    '''
    Restituisce una lista S con il massimo numero
    di attività compatibili  nella  lista A
    '''
    A.sort( key = lambda x: x[1] )
    S, tempo = [], 0
    for inizio, fine in A:
        if tempo <= inizio:
            S.append((inizio ,fine))
            tempo = fine
    return S
```

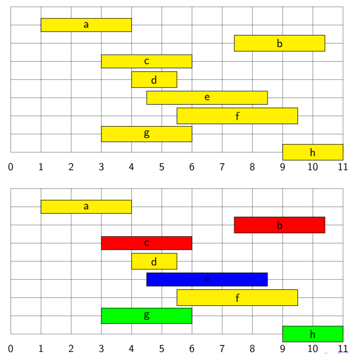
**Complessità**

- L'algoritmo ha una complessità temporale di $O(n \log n)$.
- Il costo principale è l'ordinamento delle attività.

# Il Problema dell'Assegnazione di Attività. Obiettivo: minimizzare il numero di aule

Assegnare un insieme di attività con orario di inizio e fine al **minor numero possibile di aule**. La figura che segue mostra un'istanza del problema con

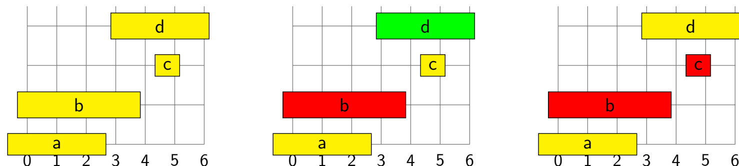
8 attività e la sua soluzione che richiede 4 aule. Le attività assegnate a un'unica aula sono indicate con lo stesso colore.



# Strategie Greedy Sbagliate: Ordinamento per Fine

- **Differenza dal problema della selezione di attività**: Non si massimizzano le attività, ma si minimizzano le risorse (aule).
- In questo caso il criterio Greedy di Ordinare le attività per orario di fine è sbagliato.

Ecco un controesempio:
In figura a sinistra è mostrata l'istanza del problema, al centro la soluzione prodotta applicando il criterio errato (che richiede 3 aule) e a destra la soluzione ottima (che utilizza solo 2 aule).



# Strategia Corretta: Ordinamento per Inizio

La regola greedy corretta per questo problema è **ordinare le attività per orario di inizio** e assegnarle.

1. Ordina tutte le attività in base all'orario di inizio in modo crescente.
2. Scansiona le attività e assegna ciascuna alla prima aula disponibile.
3. Se non ci sono aule disponibili, se ne crea una nuova.

# Prova di Correttezza

- Sia $k$ il numero di aule utilizzate dall'algoritmo greedy.
- Dobbiamo dimostrare che sono necessarie almeno $k$ aule.
- L'algoritmo crea la $k$-esima aula quando una nuova attività $a_{new}$ non può essere assegnata a nessuna delle $k-1$ aule esistenti.
- Per il nostro criterio (ordinamento per inizio), tutte le $k-1$ attività nelle aule esistenti devono essere iniziate prima o simultaneamente ad $a_{new}$.
- Dato che le aule sono occupate, le loro attività devono terminare dopo l'inizio di $a_{new}$.
- Questo dimostra che l'attività $a_{new}$ e le $k-1$ attività già in corso sono tutte attive contemporaneamente.
- Poiché esistono $k$ attività che si sovrappongono, sono necessarie almeno $k$ aule, provando l'ottimalità dell'algoritmo.

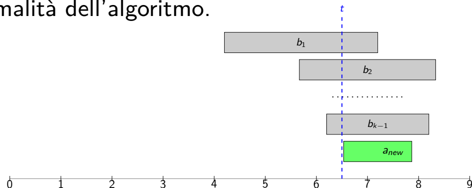



Figure: Al momento $t$, tutte le $k-1$ aule sono occupate, dimostrando il conflitto

# Implementazione semplice: scansione lineare

L'implementazione più semplice con liste ha una complessità $O(n^2)$.

```
def AssegnazioneAuleIniziaPrima(lista):
    """
    Assegna le attività nella lista al minor numero
     di aule possibile.
    Restituisce una lista di liste che rappresenta
    l'assegnazione delle aule.
    """
    # Ordina le attività in base all'orario di inizio.
    lista.sort()
    aule = [[]]
    for inizio, fine in lista:
        assegnata = False
        # Scansiona tutte le aule esistenti per trovare
        # un'aula disponibile.
        for i in range(len(aule)):
            if not aule[i] or aule[i][-1][1] <= inizio:
                aule[i].append((inizio, fine))
                assegnata = True
                break
        if not assegnata:
            # Se nessuna aula esistente è disponibile,
            # ne crea una nuova.
            aule.append([(inizio, fine)])
    return aule
```

# Strategia efficiente: heap per gestione aule

Questa seconda implementazione per migliorare l'efficienza sfrutta la struttura **min-heap**, che permette di trovare rapidamente l'elemento minimo.

Ogni elemento dell'heap è una coppia (*orario_liberazione*, *numero_aula*).

1. **Ordinamento Iniziale:** L'algoritmo inizia ordinando tutte le attività in base al loro orario di inizio in modo crescente.
2. **Esame dell'Aula Disponibile:** Per ogni attività, si esamina l'elemento minimo nella min-heap. Questo elemento rappresenta l'aula che si libera prima di tutte le altre.
3. **Assegnazione a un'Aula Esistente:** Se l'orario di inizio della nuova attività è maggiore o uguale all'orario di liberazione dell'aula trovata nella heap, quell'aula è considerata disponibile. Questa è la nostra scelta greedy: riutilizzare l'aula che si libera per prima per massimizzare l'efficienza delle risorse esistenti. L'aula viene quindi estratta dall' heap, le viene assegnata la nuova attività, e l'elemento aggiornato (con il nuovo orario di fine dell'attività) viene reinserito nella heap.
4. **Creazione di una Nuova Aula:** Se l'aula esaminata non è disponibile (cioè, il suo orario di liberazione è successivo all'orario di inizio della nuova attività), significa che tutte le aule esistenti sono occupate. L'utilizzo di una min-heap ci assicura di aver già considerato l'aula che si libera prima di tutte le altre; se anche questa non è adatta, nessuna lo sarà. In questo caso, l'algoritmo deve creare una nuova aula e aggiungerla alla heap.

# Implementazione efficiente con min-heap

```
def assegnazioneAuleIniziaPrima2(lista):
     '''
    Assegna le attività nella lista al minor numero
    di aule.
    Restituisce una lista di liste che rappresenta
    l'assegnazione delle aule.
     '''
    from heapq import heappop, heappush
    lista.sort()
    Aule=[[ ]]
    H=[(0,0)]
    for inizio,fine in lista:
        orario_liberazione, numero_aula = H[0]
        if orario_liberazione <= inizio:
            Aule[numero_aula].append( (inizio, fine) )
            heappop( H )
            heappush( H,(fine, numero_aula) )
        else:
            Aule.append([( inizio, fine )])
            heappush( H, (fine, len(Aule)-1) )
    return Aule
```

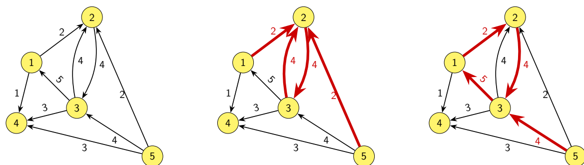
# ESERCIZI

**Suggerimento:** Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

# Esercizi

1. In un grafo orientato e pesato $G$, un sottoinsieme di archi è definito **univoco** se non contiene due o più archi uscenti dallo stesso nodo. Il peso dell'insieme univoco è la somma dei pesi di tutti i suoi archi.

   Ad esempio, nella figura seguente, a sinistra è mostrato un grafo $G$ orientato e pesato. Al centro e a destra sono rappresentati due possibili insiemi univoci di $G$ (con gli archi evidenziati in rosso). L'insieme univoco centrale ha peso 12, quello a destra ha peso 15.



   - Sviluppare un algoritmo greedy che, dato un grafo $G$ orientato e pesato, in tempo $O(n + m)$ restituisca la lista degli archi di un insieme univoco di peso massimo.
   - Fornire infine una dimostrazione formale che l'algoritmo greedy proposto sia corretto e produca una soluzione ottimale.

# Esercizi

2 Si consideri una lista di $n$ interi non negativi. Si vuole determinare il numero minimo di elementi la cui somma sia strettamente maggiore della somma di tutti gli altri elementi rimanenti nella lista.
Sviluppare un algoritmo greedy che, data la lista di interi, restituisca in tempo $O(n \log n)$ una sottolista di elementi che risolve il problema.

Ad esempio:

- per $[3, 1, 7, 1]$ la risposta è $[7]$
- per $[1, 2, 1]$ la risposta è $[1, 2]$

Fornire infine una dimostrazione formale che l'algoritmo greedy proposto sia corretto e produca una soluzione ottimale.

3 Si consideri una lista di $n$ interi $D$, dove $D[i]$ è la lunghezza del' $i$-esimo file.
L'obiettivo è memorizzare il maggior numero possibile di file su un disco rigido di capacità totale $c$.

1. Sviluppare un algoritmo greedy che, data la lista delle lunghezze dei file $D$ e la capacità del disco $c$, in tempo $O(n \log n)$, selezioni un sottoinsieme massimo di file che possono essere memorizzati sul disco.
   Ad esempio, data la lista delle lunghezze dei file $D = [5, 6, 3, 5, 4, 7, 3]$ e una capacità del disco $c = 11$, l'algoritmo deve restituire una lista con gli indici 2, 4 e 6.
2. Fornire infine una dimostrazione formale che l'algoritmo greedy proposto sia corretto e produca una soluzione ottimale, ovvero che selezioni effettivamente il numero massimo di file.

# Esercizi

4 Siano dati due vettori di interi, $A$ e $B$, entrambi di dimensione $2n$. L'obiettivo è selezionare esattamente $n$ elementi in $A$ ed $n$ elementi di $B$ in modo che, per ogni posizione $i$, (con $0 \leq i < 2n$), un elemento da $A$ o un elemento da $B$ risulti selezionato. La somma dei $2 \cdot n$ elementi selezionati sia massimizzata.

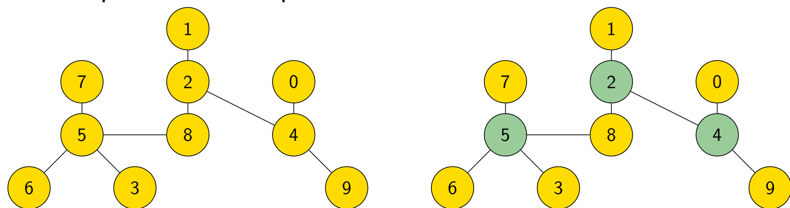
Ad esempio: per $A = [10, 2, 4, 6, 1, 7, 3, 4]$ e $B = [6, 6, 1, 0, 3, 8, 5, 7]$ la selezione ottima vale 48 esi ottiene selezionando
Da $A = [10, _, 4, 6, _, 7, _, _]$
Da $B = [_, 6, _, _, 3, _, 5, 7]$

1. Sviluppare un algoritmo che, dati i vettori $A$ e $B$, in tempo $O(n \log n)$, restituisca la somma massima.
2. Fornire infine una dimostrazione formale che l'algoritmo greedy proposto sia corretto.

# Esercizi

5 Si consideri un albero $G$, un sottoinsieme dei suoi nodi è una **copertura** se per ogni arco di $G$ almeno uno dei suoi estremi appartiene alla copertura.
L'obiettivo è trovare una copertura di $G$ di minima cardinalità.

Ad esempio, per l'albero rappresentato in figura a sinistra, i nodi di una minima copertura sono riportati in verde a destra




1. Sviluppare un algoritmo greedy che, dato $G$ tramite lista di liste, in tempo $O(n)$, restituisce una lista dei nodi appartenenti ad una copertura minima di $G$.
2. Fornire infine una dimostrazione formale che l'algoritmo greedy proposto sia corretto.

# Esercizio

Un'automobile con il serbatoio pieno può percorrere al massimo una distanza di $L$ chilometri. Si desidera viaggiare dal punto di partenza $a$ al punto di arrivo $b$, separati da una distanza complessiva di $d$ chilometri, **minimizzando il numero di soste per il rifornimento**.
Lungo il percorso sono presenti $n$ pompe di benzina, le cui distanze dal punto di partenza $a$ sono fornite in una lista ordinata di interi:

$$lista[i] = \text{distanza (in km) della } i\text{-esima pompa da } a$$

1. Progetta un algoritmo che, dati in input: $L$, $d$, e la lista $lista[1 \dots n]$, restituisca, in tempo $O(n)$, gli indici delle pompe di benzina in cui fermarsi per fare rifornimento.
   L'auto parte da $a$ con il serbatoio pieno e si assume inoltre che la distanza tra due pompe consecutive, e quella tra l'ultima pompa e la destinazione, non superi mai l'autonomia $L$ del veicolo.

   Ad esempio, per m $L = 200$, $d = 500$ e $lista = [100, 150, 250, 300, 400]$.
   La soluzione ottimale è: $[1, 3]$

   *Spiegazione:* L'auto parte con il pieno (autonomia 200 km) e può raggiungere al massimo la stazione 1 (a 150 km). Dopo il rifornimento, può arrivare alla stazione 3 (a 300 km). Da lì, con un pieno, può percorrere i restanti 200 km e raggiungere la destinazione ($300 + 200 = 500$ km). È facile verificare che con una sola sosta non sarebbe possibile arrivare a destinazione.

2. Fornisci una dimostrazione formale della correttezza dell'algoritmo greedy proposto.