

2023–2024

Palestra di algoritmi



Obiettivi del corso



Cosa faremo durante queste lezioni...

Potenziamento di abilità di problem solving	Ricerca di soluzioni informatiche ai problemi	
Potenziamento di programmazione (Python)	Ripasso e potenziamento di alcuni concetti di programmazione nel linguaggio Python	
Algoritmi	Implementazione degli algoritmi e discussione sulla complessità computazionale.	
Lavoro di gruppo	Tutte le competenze e le abilità che si impareranno in “ squadra ”, compresi gli insegnanti. In questo corso, non siamo studenti e docenti, ma TUTTI possiamo imparare da TUTTI.	
Partecipazione alle Olimpiadi dell'Informatica	Iscrizione alle competizioni a squadre ed individuali (queste solo per le classi 3° e 4°)	



01



Dal problem solving alla Programmazione

1-Problem solving



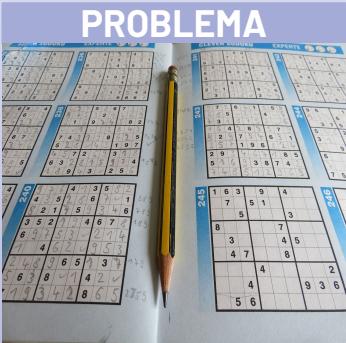
Dato un **problema** → trovare una **strategia** per arrivare ad una **soluzione**



1-Problem solving



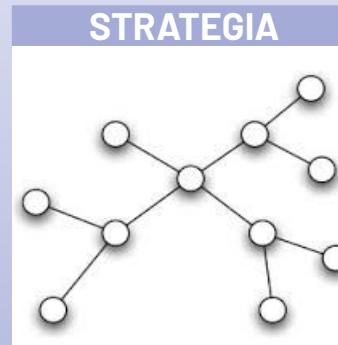
Dato un **problema** → trovare una **strategia** per arrivare ad una **soluzione** risolve



1-Problem solving: informatica



Dato un **problema** → trovare una **strategia** per arrivare ad una **soluzione** risolve



2-Algoritmo

Trovare una **sequenza di istruzioni** che soddisfino la soluzione. Molto spesso si utilizza lo pseudo-codice.

```
inizio programma
    leggi voto
    se voto ≥ 6 allora
        scrivi "Sufficiente"
    altrimenti
        scrivi "Insufficiente"
    fine se
    scrivi "Fine"
fine programma
```

```
inizio programma
    fintanto che non sei uscito
        vai avanti di un passo
        se di fronte non c'è strada allora
            gira a destra (cioè ruota su te stesso in senso orario di 90 gradi)
        fine se
    fine fintanto che
fine programma
```

3-Programmazione

Scrittura del codice con un
linguaggio di programmazione

```
def binary_search(v, value):
    """Cerca un valore nel vettore v, utilizzando la ricerca binaria, o dicotomica.
    Se il valore è presente, ritorna l'indice di una occorrenza di value nel vettore,
    altrimenti ritorna -1.
    """
    start = 0 # Indice inferiore della ricerca
    end = len(v) - 1 # Indice superiore della ricerca

    while start <= end:
        i = (start + end) // 2 # Indice del valore centrale del (sotto)vettore considerato
        print(f"v[{i}] = {v[i]}, v = {v[start:end + 1]}")

        if v[i] == value:
            return i # Valore trovato!
        elif value < v[i]:
            end = i - 1 # Sposta a sinistra l'indice superiore (considera la prima metà)
        else:
            start = i + 1 # Sposta a destra l'indice inferiore (considera la seconda metà)
    return -1

v = [17, 20, 21, 26, 31, 44, 54, 55, 77, 83, 97] # Vettore ordinato
value = int(input('Numero da cercare: '))
index = binary_search(v, value)

if index >= 0:
    print(f'Trovato {value} alla posizione di indice {index}.')
else:
    print(f'Il numero {value} non è presente nel vettore.')
```



02

Syllabus del corso



Syllabus

Syllabus

Olimpiadi di Informatica a Squadre

Versione 2 --- 19 maggio 2019

Livello 1.

- Tipi di dato primitivo (e.g. int, char, double, bool)
- Array mono-dimensionali (e.g. int[], char[], double[], bool[])
- Branching (costrutti if/else)
- Cicli limitati (costrutti for semplici)

Livello 2.

- Array multi-dimensionali (e.g. int[][][])
- Strutture dati coda e pila
- Cicli illimitati (costrutto while)
- Funzioni e ricorsione esaustiva (e.g. elencare le permutazioni)
- Ricerca binaria
- Algoritmi di ordinamento quadratici (e.g. bubble sort)
- Stringhe e ricerca quadratica di una sottocorrispondenza in una stringa
- Algoritmo di Euclide per il massimo comun divisore
- Concetti base di geometria e aritmetica



Syllabus

Livello 3.

- Contenitori standard (e.g. vector, set, map)
- Divide et impera
- Programmazione dinamica su array mono- o multi-dimensional
- Visite di grafi (BFS, DFS)
- Algoritmi di ordinamento efficienti (sort o qsort)
- Ottimizzazione approssimata tramite tecniche euristiche

Livello 4.

- Struttura dati union-find per insiemi disgiunti
- Strutture dati per query su range
- Minimo antenato comune su alberi
- Algoritmi su grafi: cammini minimi, albero ricoprente
- Programmazione dinamica su grafi aciclici
- Backtracking (algoritmi branch and bound)

Livello 5.

- Tutti gli argomenti non menzionati nei livelli precedenti



Esempio di problema



Lightweight Ladder (ladder)

In the recent festivities, Luca observed some kids playing in a gym. They seemed to enjoy very much moving around with all these colorful blocks, but sometimes they were unable to climb on them as they were too high when stacked!

Consider this simplified representation of a typical situation: each block is a cube with all sides of one meter. Some cubes can optionally be stacked to form a heap and the heaps are aligned one next to each other, as in picture 2.

When two heaps have the same number of cubes stacked, a kid can just walk to proceed to the next one. Similarly, in case of a descent when the next heap has fewer cubes than the current one, the kid exploits the gravity and falls gently on the soft cube.

As gravity does not work in the reverse direction, kids cannot climb to higher heaps without the help of a ladder. In the situation presented in figure 2, two ladders can be strategically placed between the start and the first heap and before the last heap.



Figure 1: Kids at work.

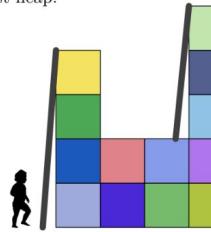


Figure 2: A sequence of four heaps with two ladders of length 4 and 3 meters.

Placing these ladders all over the path can be quite expensive, especially when it is really long. Luca had an idea: kids could carry themselves a single lightweight ladder to reach the end of the path; thus, they need a ladder which covers all their needs, optionally being longer in some cases, but never shorter (otherwise they cannot make it!).

What is the shortest ladder that one can carry to complete the path from the ground at the beginning until the last of N heaps?

Among the attachments of this task you may find a template file `ladder.*` with a sample incomplete implementation.



Esempio di problema



Input

The first line contains the number N of heaps. The second line contains N integers C_i , the number of cubes stacked in the i -th heap.

Output

You need to write a single line with an integer: the minimum length of a ladder that can be carried to complete the path.

ladder

Page 1 of 3

Constraints

- $1 \leq N \leq 1\,000\,000$.
- $0 \leq C_i \leq 10^9$ for each $i = 0 \dots N - 1$.
- In case no ladder is needed at all, you must output the value 0.

Scoring

Your program will be tested against several test cases grouped in subtasks. In order to obtain the score of a subtask, your program needs to correctly solve all of its test cases.

- Subtask 1 (0 points) Examples.



- Subtask 2 (10 points) $N \leq 2$.



- Subtask 3 (45 points) $N, C_i \leq 1000$ for each $i = 0 \dots N - 1$.



- Subtask 4 (45 points) No additional limitations.



Examples

input	output
4 4 2 2 5	4
10 2 6 3 7 0 0 7 3 6 4	7

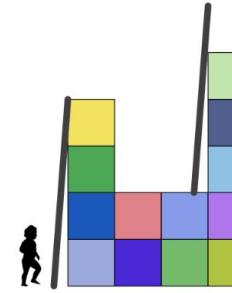


Esempio di problema

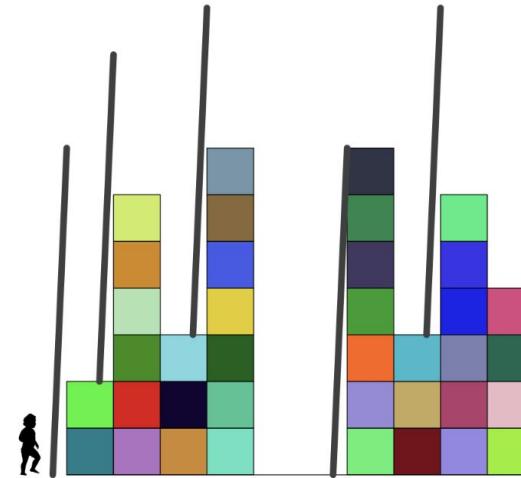


Explanation

In the **first sample case**, already described in this statement, the shortest possible length for the ladder is 4 meters (note that in the picture they are shown as two different ladders, but in reality it is just one carried by the kid as he proceeds):



In the **second sample case** we need at least a ladder of length 7 meters (any higher length would work as well, but we are required to find the minimum solution):





Esempio di problema

```
# input data
N = int(input().strip())
C = list(map(int, input().strip().split()))

# insert your code here

ladder_list=[C[0]]
for i in range(N-1):
    if(C[i+1]-C[i]>0):
        ladder_list.append(C[i+1]-C[i])

print(str(max(ladder_list))) # print the result
```



03



Iscrizione alle OII



OII: cosa sono

<https://www.olimpiadi-informatica.it/index.php/ioi/cosa-sono.html>





Sito web ufficiale

<https://sites.google.com/aldini.istruzioneer.it/olimpiadi-informatica-squadre/homepage>

The image shows a screenshot of the official website for the Team Informatics Olympiad (OI(S)). The main title "Olimpiadi di Informatica a Squadre" is displayed prominently in white text against a dark blue background. Below the title, it says "Campionato 2023/24 - 15° Edizione". At the bottom left, there are logos for "OLIMPIADI DI INFORMATICA A SQUADRE" and "ISTITUTO DI ISTRUZIONE SUPERIORE ALDINI VALERIANI". The background of the page features a blurred view of a computer screen displaying code.

Olimpiadi di Informatica a Squadre

Campionato 2023/24 - 15° Edizione

IN EVIDENZA:

OLIMPIADI DI INFORMATICA A SQUADRE

ISTITUTO DI ISTRUZIONE SUPERIORE ALDINI VALERIANI



01

I/O da e su file



Standard input e Standard output

I programmi **operano su variabili**, che rappresentano riferimenti a locazioni di memoria, che contengono i dati durante l'esecuzione del programma.

La memoria **RAM è volatile**, ovvero i dati contenuti vengono persi quando il programma in esecuzione termina, oppure il computer viene spento.



```
stringa = input("Inserisci una stringa: ")
```

```
print("Il risultato è ", valore)
```





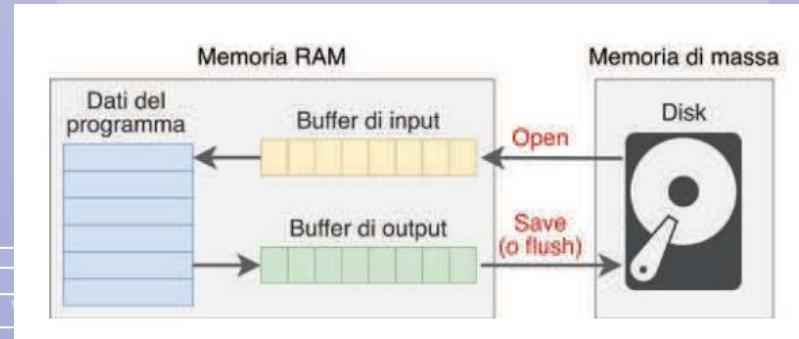
Input e Output da file

Per memorizzare i dati in modo permanente, affinché siano disponibili in una futura sessione di lavoro, è necessario **salvarli in file** sugli storage device, cioè nelle memorie di massa.

I programmi possono accedere alle memorie di massa, sia in input sia in output, **attraverso il sistema operativo**.

Per leggere o scrivere dati in un file con Python distinguiamo due possibili tipologie di operazione sui file:

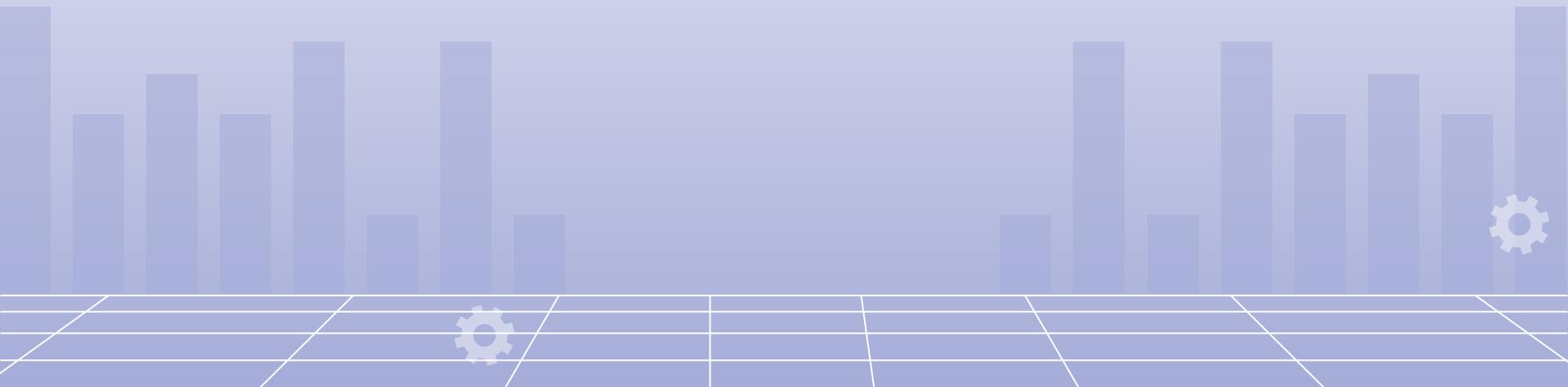
1. **file input**, che apre un file in lettura, caricando il suo contenuto, o parte di esso, nel buffer di lettura, cioè un'area di memoria temporanea della memoria RAM;
2. **file output**, che salva i dati nel buffer di scrittura, affinché, in seguito, possano essere salvati su file.





3 passi fondamentali:

1. Apertura del file
2. Operazioni sul file (lettura o scrittura)
3. Chiusura del file





Apertura e chiusura FILE

```
file = open(filename, mode="", encoding='UTF-8')
```

filename = nome del file e il suo percorso dove lo avete salvato → CONSIGLIO: metterlo sempre nella stessa cartella del programma principale

mode= modalità di apertura. Di default un file viene aperto in lettura, ma si può specificare una modalità diversa con il parametro

codifica dei caratteri = da utilizzare per interpretare il contenuto del file

Una volta finite le operazioni sul file:

```
file.close()
```

Per aprire il file *Miofile.txt* in **lettura** si usa il seguente comando, dove si suppone che la **cartella corrente** sia la stessa nella quale è collocato il file:

```
>>> fin = open('MioFile.txt', 'r')
```

Modalità	Esempio	Osservazioni
'w'	open(<i>NomeFile</i> , 'w')	Creazione del file e apertura in sola scrittura (se il file esiste, viene cancellato e ricreato)
'r'	open(<i>NomeFile</i> , 'r')	Apertura del file in sola lettura (il file deve esistere: errore se non c'è)
'a'	open(<i>NomeFile</i> , 'a')	Apertura in sola scrittura in modalità <i>append</i> (se il file non esiste, lo crea)
'r+'	open(<i>NomeFile</i> , 'r+')	Apertura in lettura e in scrittura in modalità <i>append</i> (il file deve esistere)



Scrittura su FILE

```
file = open("./output.txt", mode="w")
file.write("Ciao mondo!")
```

ESEMPIO

Per scrivere alcune informazioni nel file *MioFile.txt*, che si suppone collocato nella directory corrente, si eseguono i seguenti comandi:

```
>>> fout = open('MioFile.txt', 'w')
>>> fout.write('Prima riga di dati\n')
```

19

write scrive una stringa
e restituisce il numero di
caratteri scritti



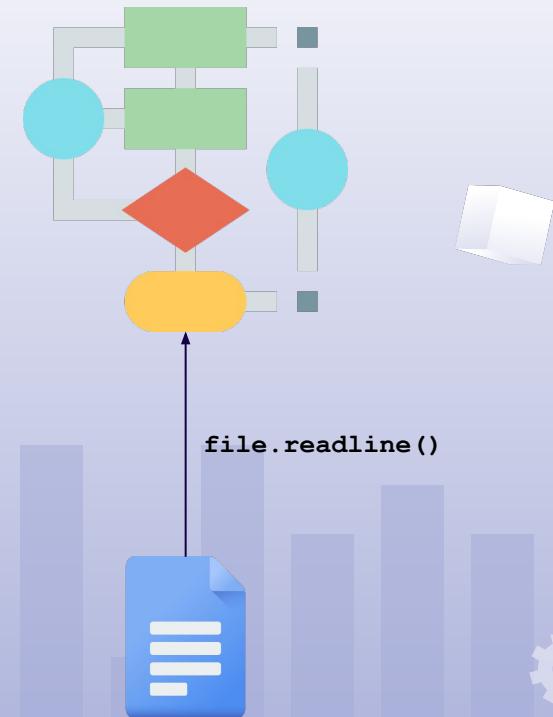


Lettura da FILE

Per leggere i dati di un file di testo aperto in lettura si usa il metodo **readline()** che restituisce una riga di caratteri, incluso il carattere '\n' di fine riga

```
file = open("./input.txt", mode="r")
riga = file.readline()
```

```
file = open("./input.txt", mode="r")
riga = file.readline()
while riga != "":
    print(riga)
    riga = file.readline()
```

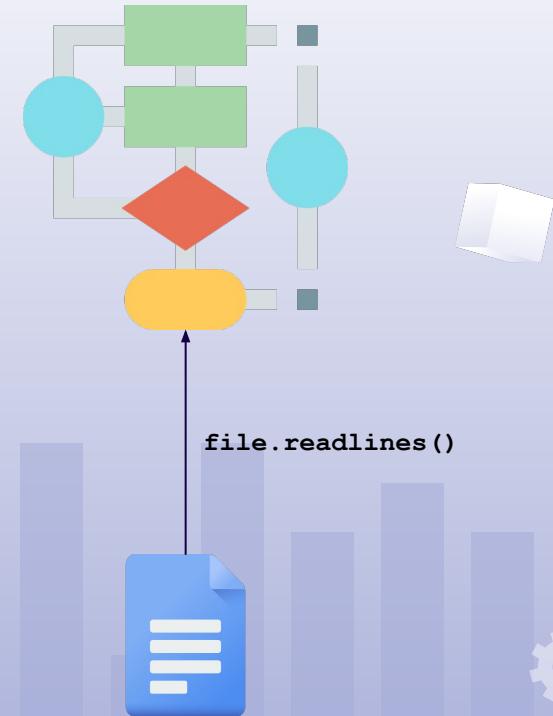




Lettura da FILE

Con il metodo **readlines()** si legge un intero file: esso restituisce una **lista** con il contenuto del file (ogni elemento della lista contiene una riga del file)

```
file = open("./input.txt", mode="r")  
lista = file.readlines()
```





Esercizi

1. Scrivere un programma che legga da file un intero N ed N numeri interi, li sommi e stampi a video la somma.
2. Scrivere un programma che legga da file un intero N ed N numeri interi, poi calcoli quanti numeri pari sono stati inseriti.
3. Scrivere un programma che legga da file un intero N ed N numeri interi, poi stampi a video il maggiore ed il minore.





Soluzioni

```
file = open("./input.txt", mode = "r")

n = int(file.readline())
somma = 0

for i in range(n):
    numero = int(file.readline())
    somma += numero

print(somma)
```

1. Scrivere un programma che legga da file un intero N ed N numeri interi, li sommi e stampi a video la somma.





Soluzioni

```
file = open("./input.txt", mode = "r")

n = int(file.readline())
contatore = 0

for i in range(n):
    numero = int(file.readline())
    if numero % 2 == 0:
        contatore += 1

print(contatore)
```

2. Scrivere un programma che legga da file un intero N ed N numeri interi, poi calcoli quanti numeri pari sono stati inseriti.





Soluzioni

```
file = open("./input.txt", mode = "r")

n = int(file.readline())
maggiori = minore = int(file.readline())

for i in range(n - 1):
    numero = int(file.readline())
    if numero > maggiori:
        maggiori = numero
    if numero < minore:
        minore = numero

print("Minore:" + str(minore))
print("Maggiori:" + str(maggiori))
```

3. Scrivere un programma che legga da file un intero N ed N numeri interi, poi stampi a video il maggiore ed il minore.





02



Matrice



Matrice

Una lista può avere come elementi altre liste.

La **matrice** (o array a due dimensioni) è un insieme di elementi dello stesso tipo e in corrispondenza biunivoca con un insieme di coppie ordinate di numeri interi, che rappresentano rispettivamente il numero della riga e il numero della colonna della matrice.

La lista *mat*:

```
>>> mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
```

Index	Value
0	[1, 2, 3]
1	[4, 5, 6]
2	[7, 8, 9]
3	[10, 11, 12]

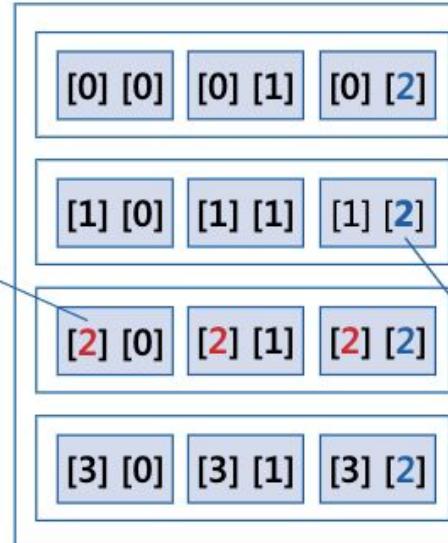
Per accedere a un elemento della matrice mat si usano scritture con **due indici**, come in **mat[r][c]**, dove il primo indice è detto indice di **riga** e il secondo indice di **colonna**



Matrice

L'accesso agli elementi della matrice

mat[r,c]: r indice di riga



mat[r,c]: c indice di colonna



Matrice

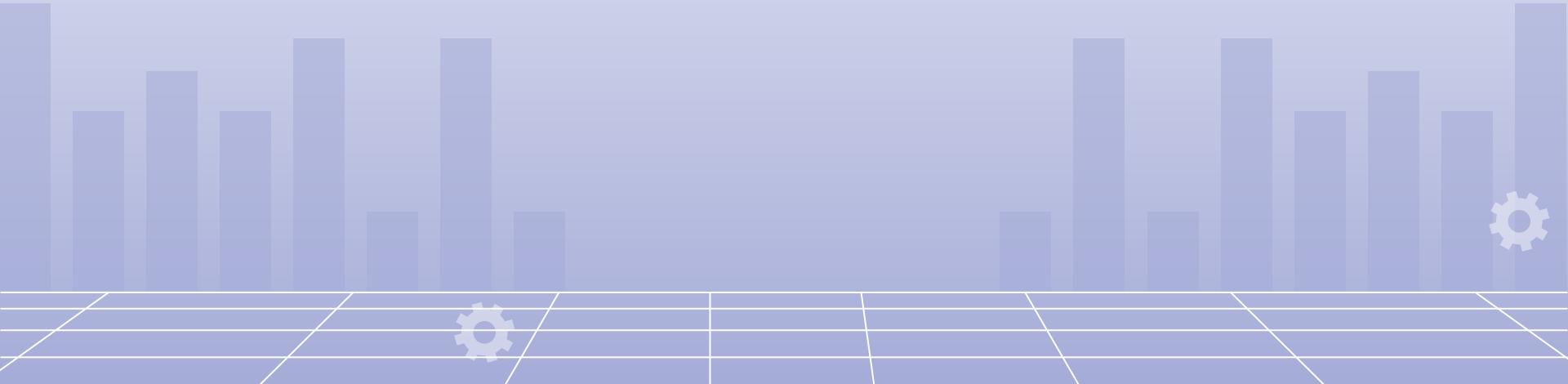
```
matrice = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]  
  
print(matrice[0][0])  
print(matrice[0][1])  
print(matrice[0][2])  
  
print(matrice[1][0])  
print(matrice[1][1])  
print(matrice[1][2])
```

```
for r in range(len(matrice)):  
    for c in range(len(matrice[r])):  
        print(matrice[r][c])
```



Esercizi (3)

- Prendere da un file di input (input.txt) due interi: il primo (**N**) sarà il numero di righe e il secondo (**M**) il numero di colonne
- Creare una matrice **NxM** contenente tutti zeri





Esercizi (3)

- Prendere da un file di input (input.txt) due interi: il primo (**N**) sarà il numero di righe e il secondo (**M**) il numero di colonne
- Creare una matrice **NxM** contenente tutti zeri

Soluzione 1) (i 2 numeri sono in righe diverse)

```
file = open("./input.txt", mode = "r")

n = int(file.readline())
m = int(file.readline())
matrice = []

for i in range(n):
    riga = [0] * m
    matrice.append(riga)

print(matrice)
```

Soluzione 2) (i 2 numeri sono nella stessa riga)

```
file = open("./input.txt", mode = "r")

contenuto = file.readline()
numeri = contenuto.split()
n = int(numeri[0])
m = int(numeri[1])
matrice = []

for i in range(n):
    riga = [0] * m
    matrice.append(riga)

print(matrice)
```

03

Dizionario



Dizionari

- ★ Hanno qualche somiglianza con le liste visto che anche i dizionari si usano per contenere degli elementi, ma non hanno un proprio ordine
- ★ Ogni elemento dentro il dizionario è formato da una chiave e da un valore
- ★ Vediamo un paio di esempi:

```
dizionario = {'Gianfranco': 15, 'Anna': 10, 'Mario': 23}
```

```
dizionario = {'a': 1, 'c': 5, 'f': 13}
```



Dizionari

- ★ Sono utilissimi perchè possiamo accedere direttamente ad un determinato senza metterci a scorrere tutti gli elementi per trovarlo (come succede nelle liste)
- ★ Se vogliamo accedere ad un elemento lo facciamo tramite la sua chiave
- ★ Vediamo un paio di esempi:

```
dizionario = {'Gianfranco': 15, 'Anna': 10, 'Mario': 23}

print(dizionario['Gianfranco']) # Verrà stampato 15
print(dizionario['Casa']) # Se la chiave non esiste restituisce un KeyError
print('Casa' in dizionario) # Verrà stampato False
print('Anna' in dizionario) # Verrà stampato True
```



Dizionari

- ★ È possibile aggiungere o modificare elementi come nelle liste
- ★ Vediamo un paio di esempi:

```
dizionario = {'Gianfranco': 15, 'Anna': 10, 'Mario': 23}

dizionario['Mario'] = 99 # Modifica il valore associato a una chiave esistente
dizionario['Luca'] = 50 # Crea un nuovo elemento, con chiave 'Luca' e valore 50
del dizionario['Luca'] # Rimuove l'elemento (chiave e valore) con chiave 'Luca'
```

- ★ Per approfondire vedi [qui](#)



04



Esercizi



Esercizi di riscaldamento

- Trova il massimo
- Trova la somma pari massima
- Trova la somma pari massima v2.0





Trova il massimo - Soluzione

```
input_file = open("./input.txt", mode = "r")
output_file = open("./output.txt", mode = "w")

n = int(input_file.readline())
vettore = input_file.readline().split()

maggiori = int(vettore[0]);

for i in range(1, n):
    numero = int(vettore[i])
    if numero > maggiori:
        maggiori = numero

output_file.write(str(maggiori))
```





Trova la somma massima – Soluzione

```
1  input_file = open("input.txt", "r")
2
3  n = int(input_file.readline())
4
5  somma_massima = -1
6
7  # Leggi le coppie e calcola la somma massima pari
8  for _ in range(n):
9      riga = input_file.readline().split()
10     a = int(riga[0])
11     b = int(riga[1])
12     somma = a + b
13     if somma % 2 == 0 and somma > somma_massima:
14         somma_massima = somma
15
16 input_file.close()
17
18 output_file = open("output.txt", "w")
19 output_file.write(str(somma_massima))
20
21 output_file.close()
```



Trova la somma massima v2



Soluzione 1



```
1  input_file = open("./input.txt", mode = "r")
2  output_file = open("./output.txt", mode = "w")
3
4  n = int(input_file.readline())
5  vettore = input_file.readline().split()
6
7  somma_massima = -1
8
9  for i in range(n - 1):
10     for j in range(i + 1, n):
11         somma = int(vettore[i]) + int(vettore[j])
12         print(somma)
13         if somma % 2 == 0 and somma > somma_massima:
14             somma_massima = somma
15
16     input_file.close()
17
18     output_file = open("output.txt", "w")
19     output_file.write(str(somma_massima))
20
21     output_file.close()
```



Trova la somma massima v2



Soluzione 2

```
1 input_file = open("./input.txt", mode = "r")
2 output_file = open("./output.txt", mode = "w")
3
4 n = int(input_file.readline())
5 vettore = input_file.readline().split()
6
7 massimo_pari = -1;
8 massimo_pari2 = -1;
9
10 massimo_dispari = -1;
11 massimo_dispari2 = -1;
12
13 for i in range(n):
14     numero = int(vettore[i])
15     if numero % 2 == 0:
16         if numero > massimo_pari:
17             massimo_pari2 = massimo_pari
18             massimo_pari = numero
19         elif numero > massimo_pari2:
20             massimo_pari2 = numero
21     else:
22         if numero > massimo_dispari:
23             massimo_dispari2 = massimo_dispari
24             massimo_dispari = numero
25         elif numero > massimo_dispari2:
26             massimo_dispari2 = numero
27
28 somma_massima = max(massimo_pari + massimo_pari2, massimo_dispari + massimo_dispari2)
29
30 input_file.close()
31
32 output_file = open("output.txt", "w")
33 output_file.write(str(somma_massima))
34
35 output_file.close()
```



Nimbus

Proviamo a pensare come risolvere questo algoritmo...

Passi:

- Capire il problema
- Formalizzarlo (togliere la storiella)
- Capire i dati di input
- Trovare una strategia per risolverlo
- Scrivere un algoritmo
- Implementare la soluzione

Giri sulla Scopa Nimbus3000 (nimbus)

Difficoltà D = 2 (tempo limite 1 sec).

Descrizione del problema

Al celebre maghetto Harry Potter è stata regalata una scopa volante modello Nimbus3000 e tutti i suoi compagni del Grifondoro gli chiedono di poterla provare. Il buon Harry ha promesso che nei giorni a venire soddisferà le richieste di tutti, ma ogni ragazzo è impaziente e vuole provare la scopa il giorno stesso. Ognuno propone ad Harry un intervallo di tempo della giornata durante il quale, essendo libero da lezioni di magia, può fare un giro sulla scopa, e per convincerlo gli offre una fantastica caramella Tuttigusti+1. Tenendo presente che una sola persona alla volta può salire sulla Nimbus3000 in ogni istante di tempo, Harry decide di soddisfare, tra tutte le richieste dei ragazzi, quelle che gli procureranno la massima quantità di caramelle (che poi spartirà coi suoi amici Ron e Hermione). Aiutalo a trovare la migliore soluzione possibile.

Dati di input

Il file `input.txt` contiene nella prima riga un intero positivo N, che indica il numero di richieste, che sono numerate da 1 a N. Ognuna delle successive N righe contiene una coppia di interi. Ciascuna di tali righe contiene una coppia di interi positivi A e B, separati da uno spazio, a rappresentare la richiesta di poter utilizzare la scopa dall'istante iniziale A fino all'istante finale B, in cambio di una caramella (dove A < B). A tal fine, il tempo è diviso in istanti discreti numerati a partire da 1 in poi.

Dati di output

Il file `output.txt` è composto da una riga contenente un solo intero, che rappresenta il massimo numero di caramelle che Harry può ottenere. Assunzioni

$1 < N < 1000$ Gli interi nelle N coppie sono distinti l'uno dall'altro (non esistono due interi uguali, anche in coppie diverse).

Esempi di input/output

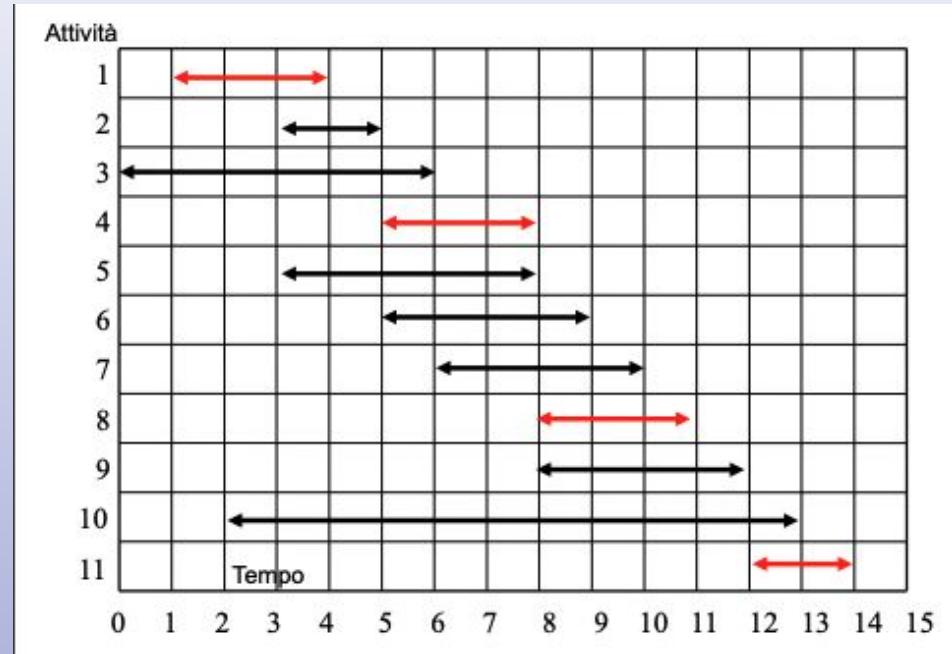
File input.txt	File output.txt
5	
1 5	
3 7	
9 11	
10 12	
6 13	2



Una possibile visualizzazione

Di seguito una possibile visualizzazione dell'input:

11
14
35
06
58
38
59
610
811
812
213
1214



Le immagini inserite nelle seguenti slide sono prese dal sito del Prof. Alberto Montresor, docente di informatica presso l'Università di Trento (<https://cricca.disi.unitn.it/montresor/>)."



Riordiniamo gli intervalli!

- Sembra che selezionando gli intervalli per istante di fine ci consenta di scegliere sempre il numero massimo di intervalli 😊.
- L'idea chiave infatti è che selezionando l'intervallo con la fine minore, si libera lo spazio “più velocemente” per gli altri intervalli, consentendo di inserire il massimo numero possibile di intervalli disgiunti.
- Basta questo per stabilire che questo approccio è corretto?
NO, dovremmo formulare una vera e propria dimostrazione, che assicura che utilizzando questo metodo avremo sempre una soluzione ottima. Se volete un assaggio...





E perché non per istante di inizio?

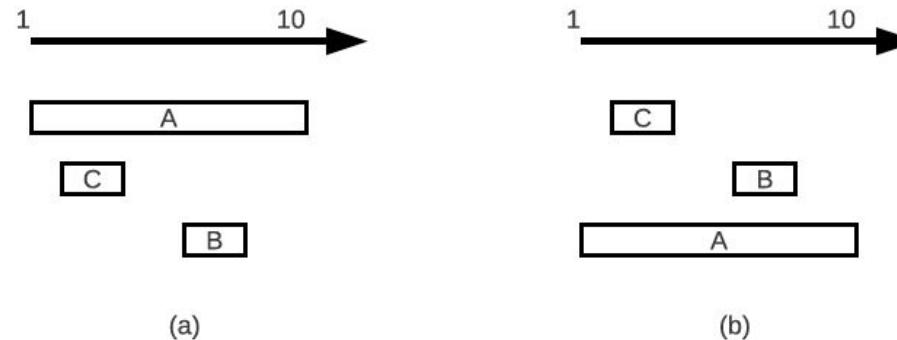
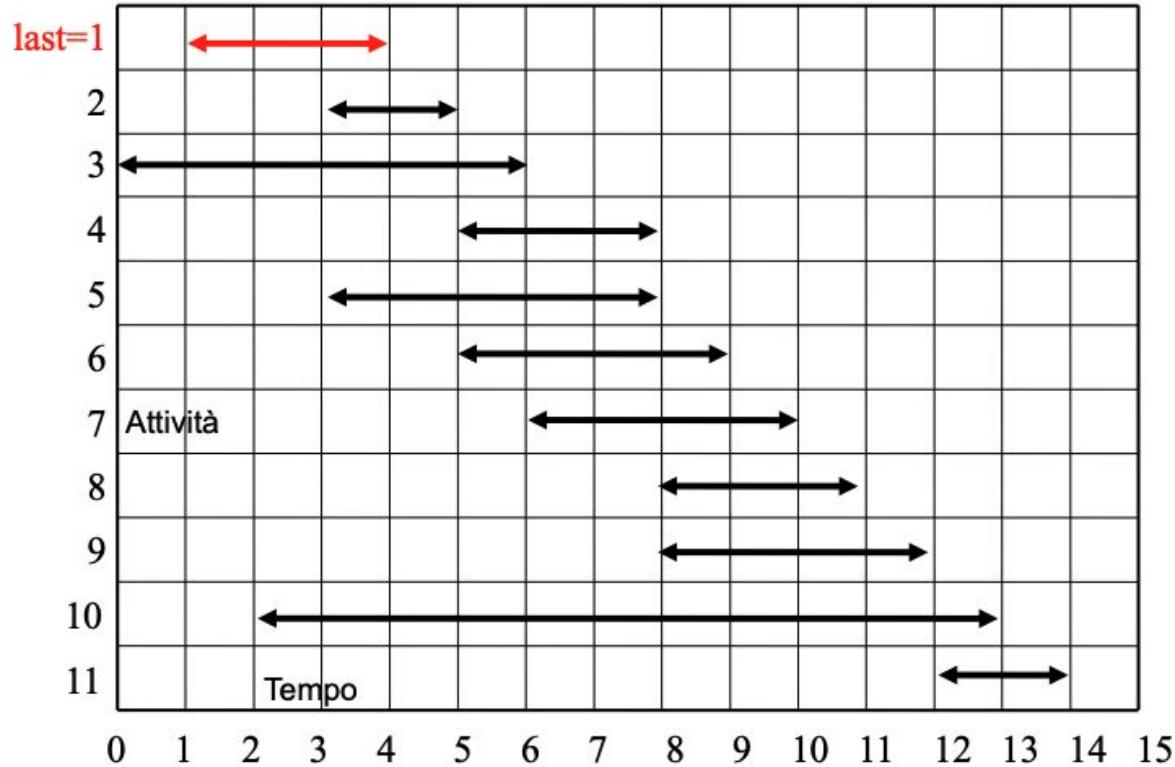


Figura 5.1: Ordinamento per istante di inizio (a) e per istante di fine (b)

Se arriva una persona che prenota subito la scopa Nimbus3000 e se la tiene per tutto il giorno siamo fregati...

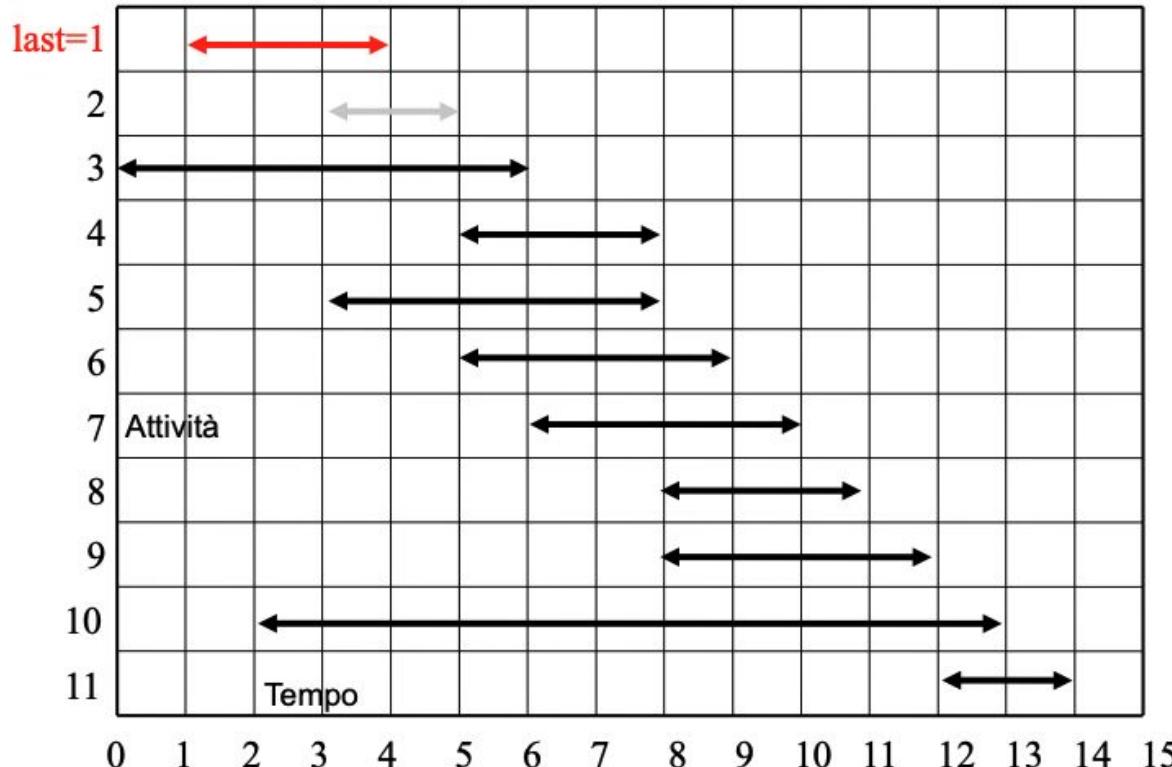


Soluzione passo dopo passo



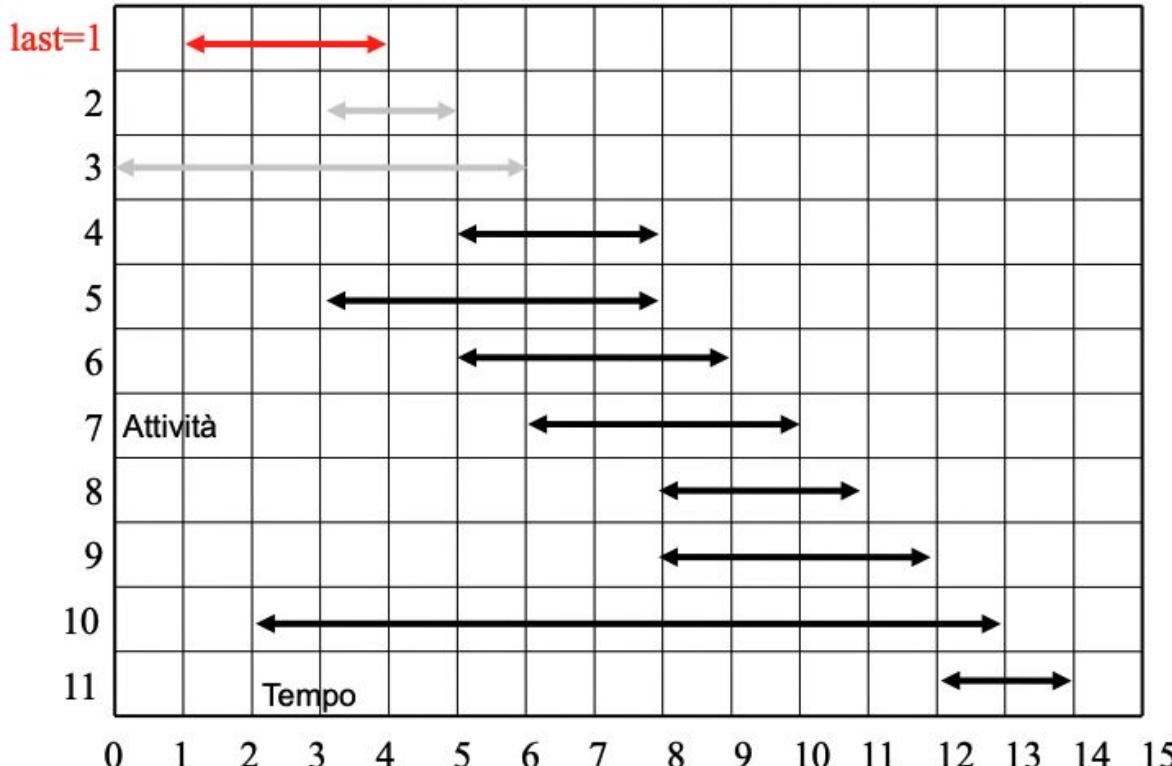


Soluzione passo dopo passo



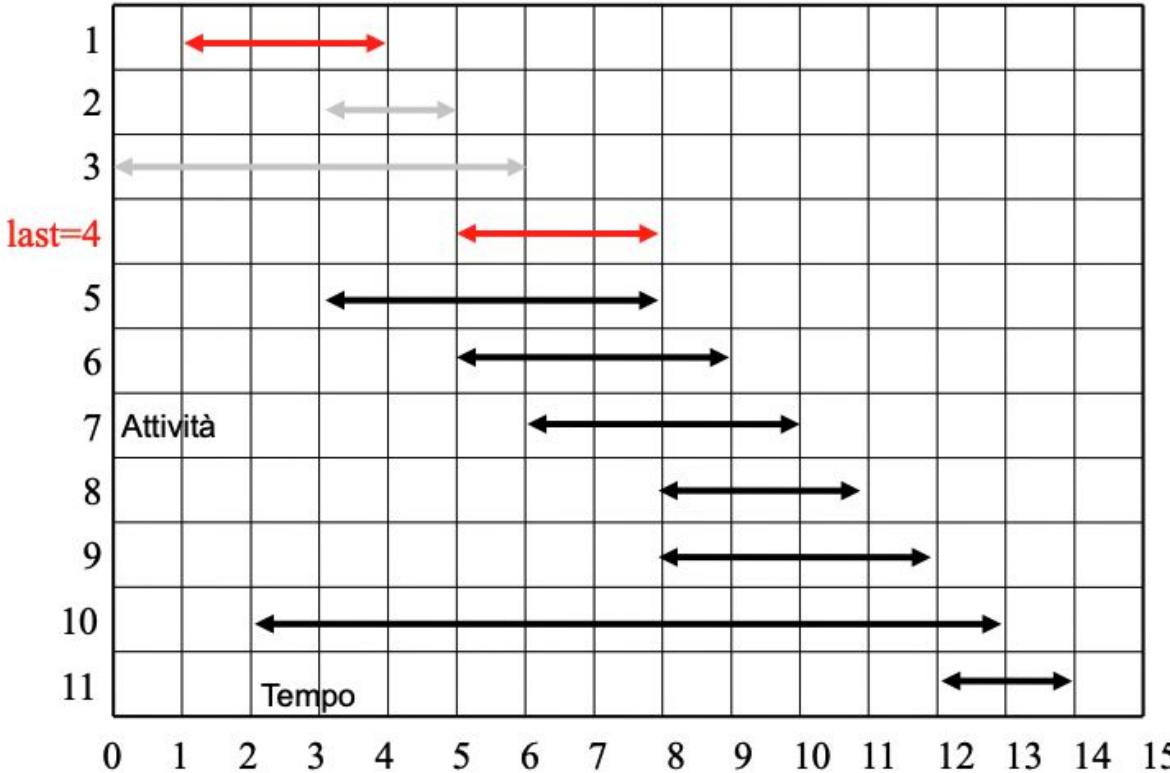


Soluzione passo dopo passo



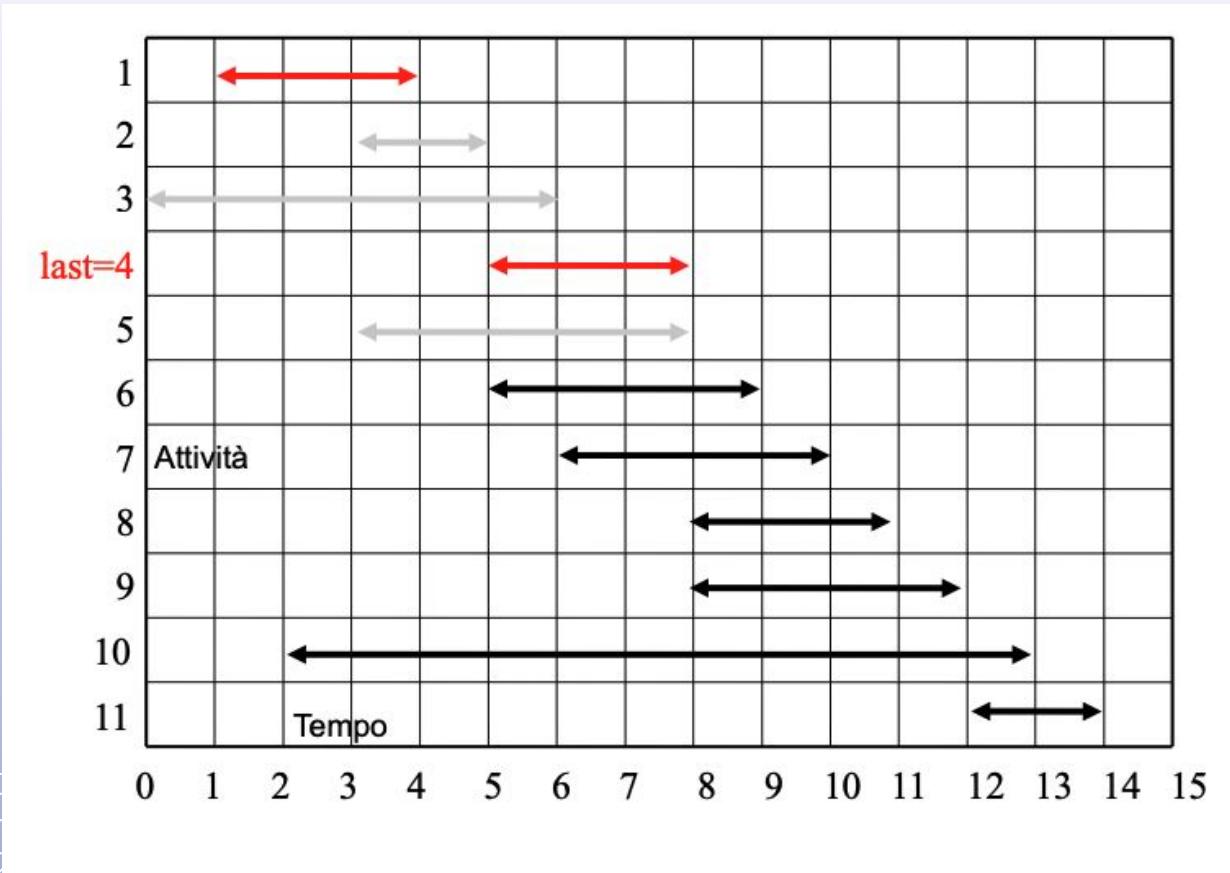


Soluzione passo dopo passo



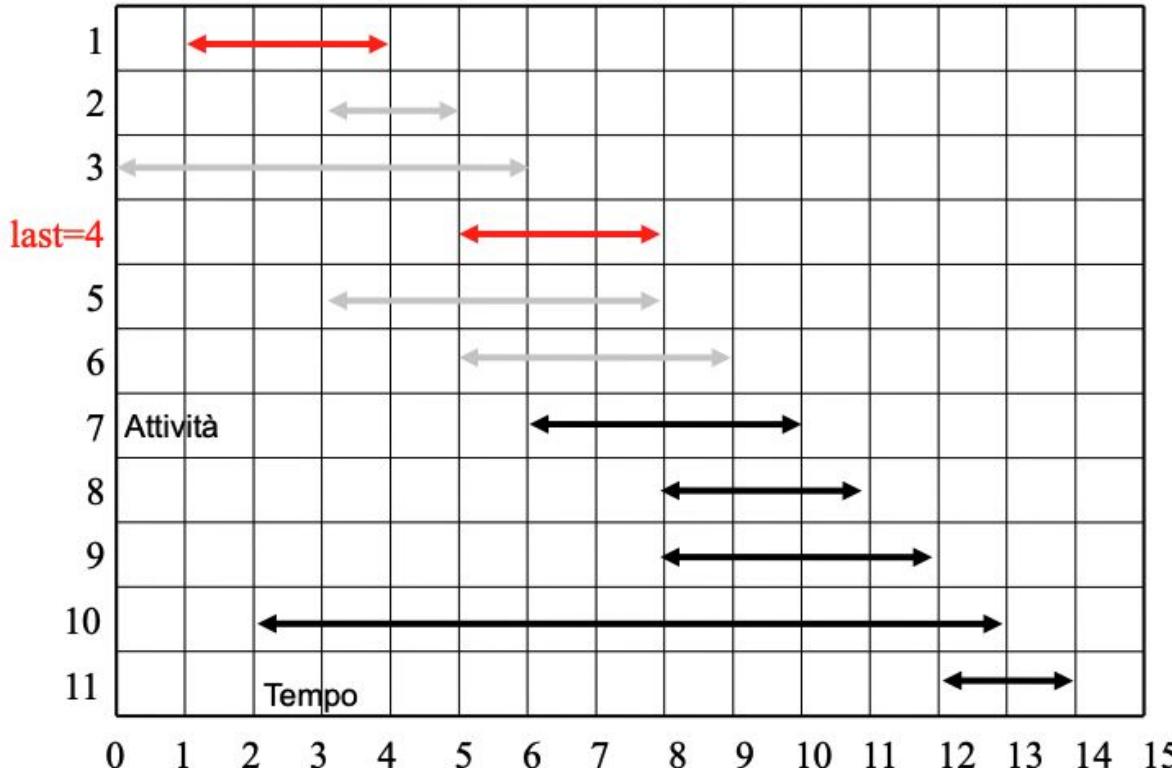


Soluzione passo dopo passo



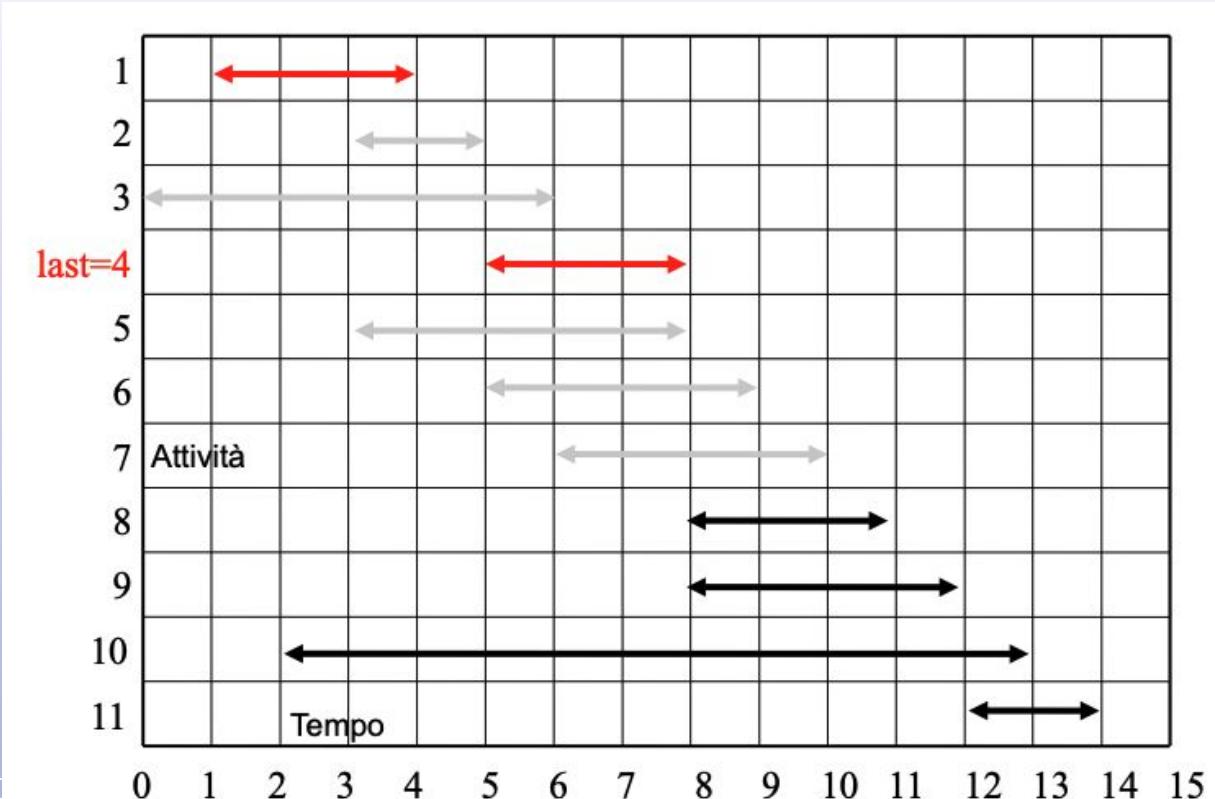


Soluzione passo dopo passo



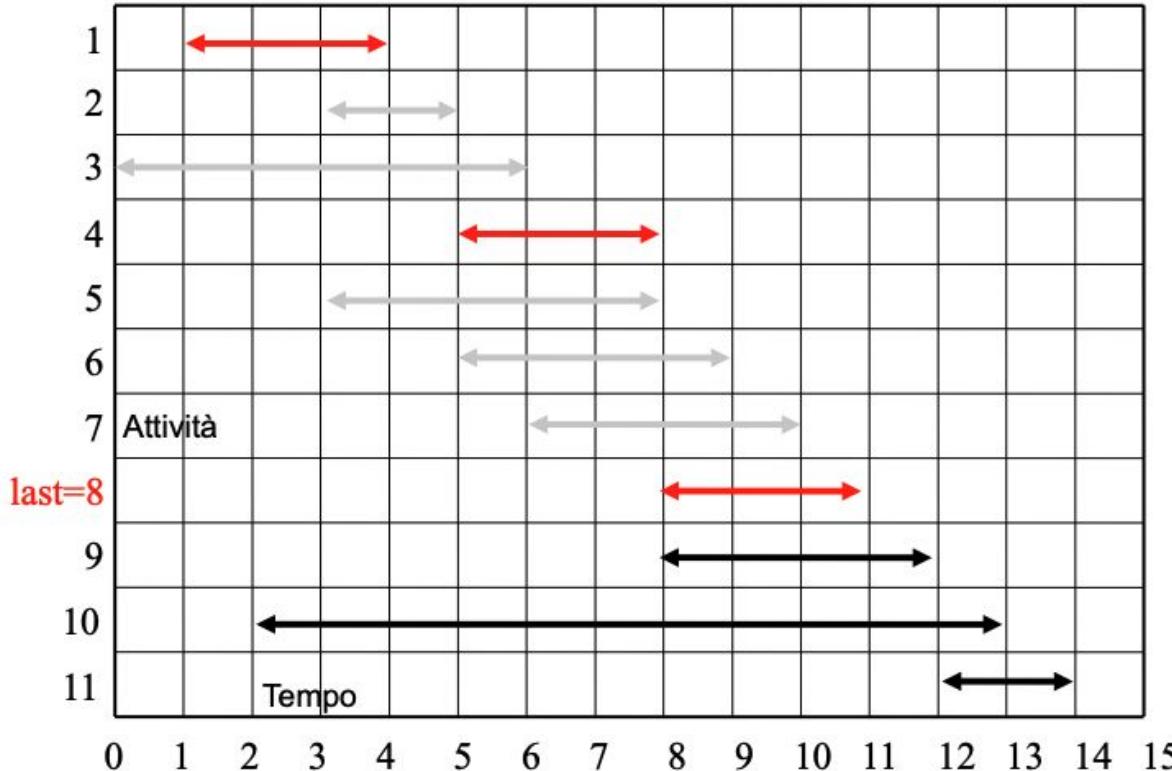


Soluzione passo dopo passo



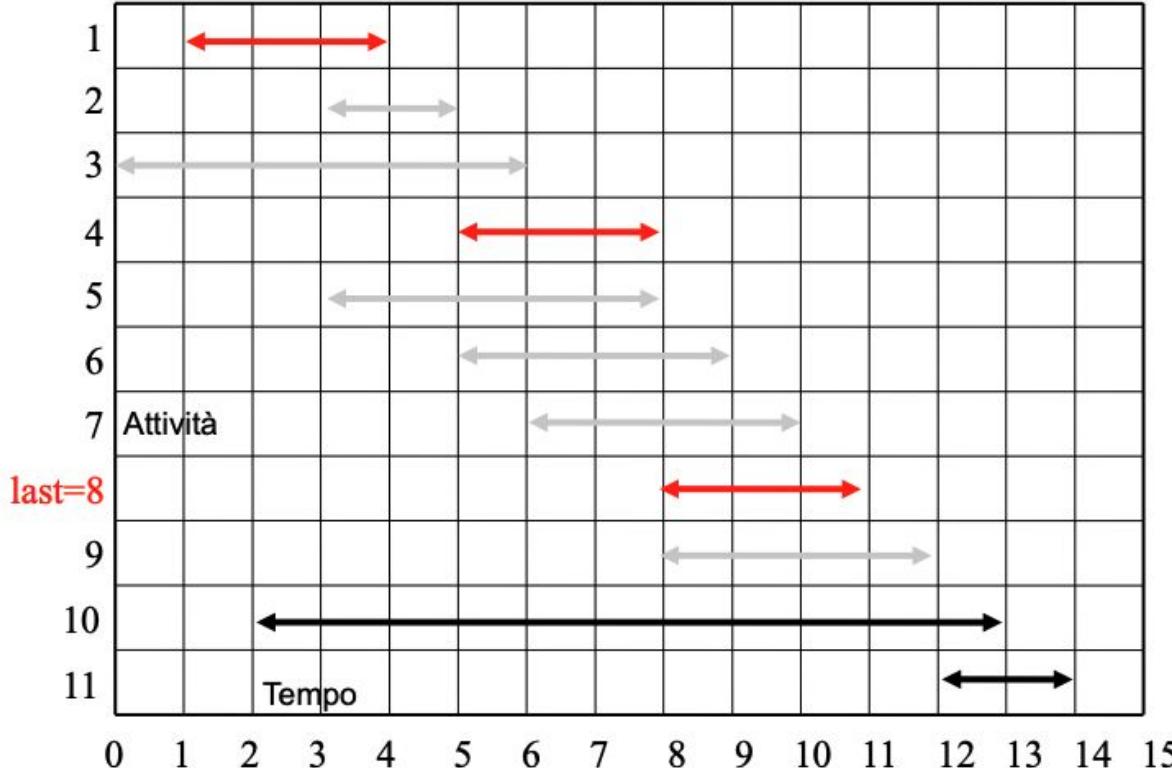


Soluzione passo dopo passo



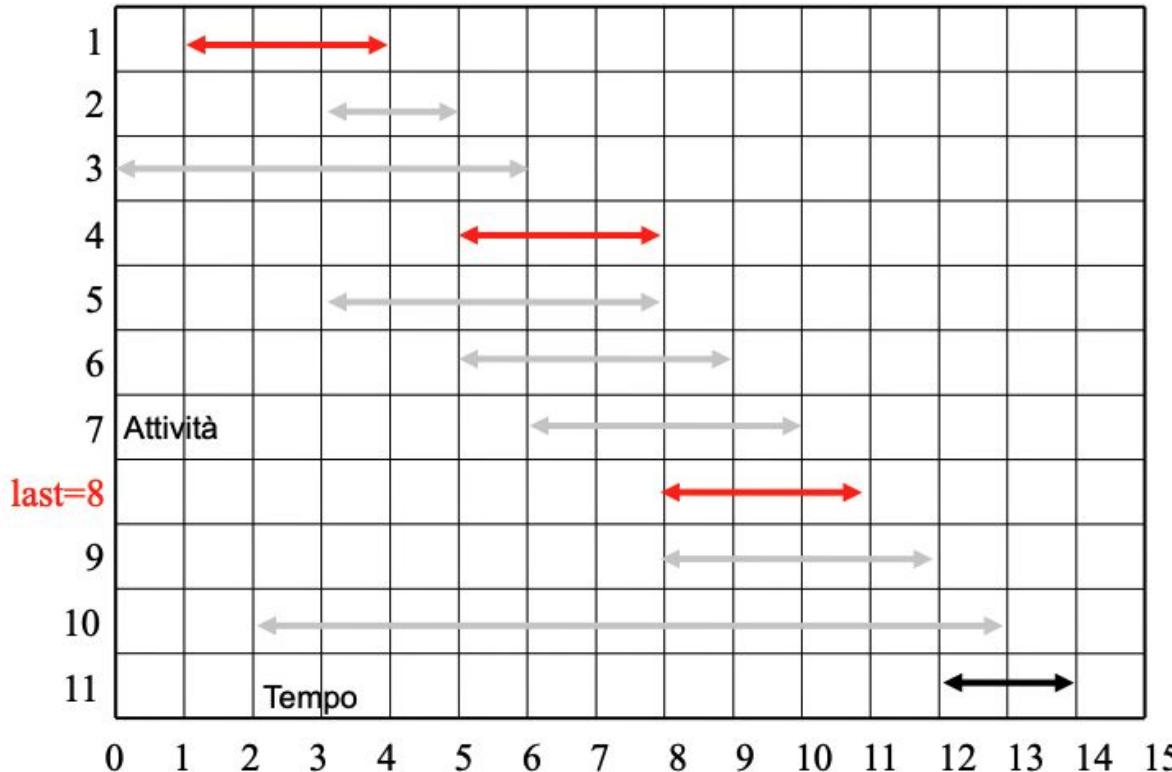


Soluzione passo dopo passo



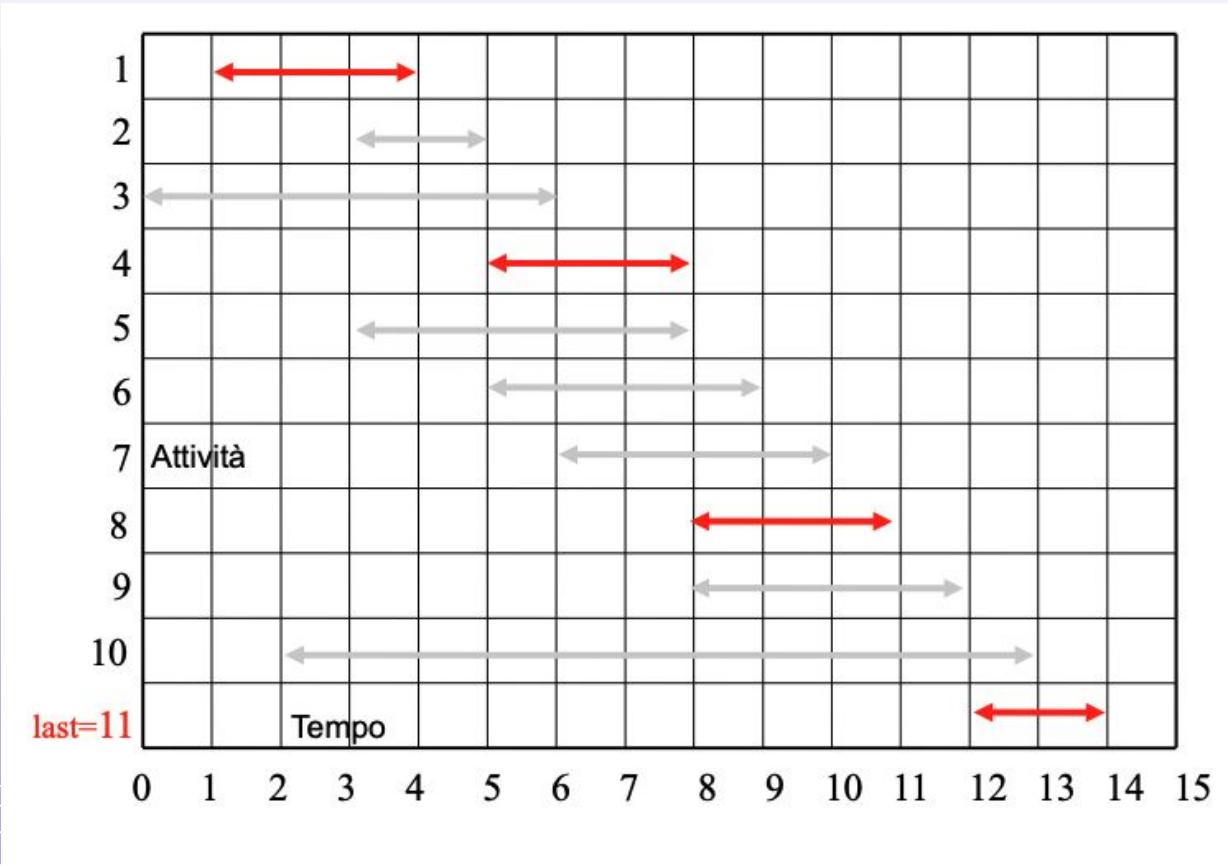


Soluzione passo dopo passo





Soluzione passo dopo passo





Nimbus: soluzione 1

Algoritmo Greedy

- Riordino gli intervalli di tempo di utilizzo per estremo di fine crescente
- Decido di avere una lista in cui aggiungo solo gli intervalli che, dopo aver scelto il primo (scelta greedy), hanno il tempo di inizio più alto rispetto al precedente preso.

```
import operator

file_in = open("nimbus/input.txt", mode="r")
# file_in = open("nimbus/input2.txt", mode="r")
file_out = open("nimbus/output.txt", mode="w")

lista_finale = []
n_richieste = int(file_in.readline())
intervalli = []

for i in range(n_richieste):
    lista = file_in.readline().split()
    lista[0] = int(lista[0])
    lista[1] = int(lista[1])
    intervalli.append(lista)

intervalli_sorted = sorted(intervalli, key=operator.itemgetter(1))

lista_finale.append(intervalli_sorted[0])

for i in range(len(intervalli_sorted)):
    if intervalli_sorted[i][0] > lista_finale[len(lista_finale)-1][1]:
        lista_finale.append(intervalli_sorted[i])

file_out.write(str(len(lista_finale)))

file_in.close()
file_out.close()
```



Nimbus: soluzione 2

Algoritmo Greedy

- Riordino gli intervalli di tempo di utilizzo per estremo di fine crescente
- Uso un semplice contatore che aumenta quando trovo intervalli che, dopo aver scelto il primo (scelta greedy), hanno il tempo di inizio più alto rispetto al precedente preso.



```
1 import operator
2
3 input_file = open("./input.txt", mode = "r")
4 output_file = open("./output.txt", mode = "w")
5
6 n = int(input_file.readline())
7 lista = []
8 contatore = 0
9
10 for i in range(n):
11     riga = input_file.readline()
12     intervallo = riga.split()
13     inizio = int(intervallo[0])
14     fine = int(intervallo[1])
15     lista.append((inizio, fine))
16
17 # la funzione sorted() restituisce la lista ordinata
18 # operator.itemgetter(1) specifica vogliamo ordinare in
19 # base al secondo elemento
20 lista = sorted(lista, key = operator.itemgetter(1))
21
22 # il primo intervallo di sicuro è da prendere, salviamocelo
23 fine_ultimo = lista[0][1]
24 contatore += 1
25
26 for i in range(1, n):
27     inizio = lista[i][0]
28     fine = lista[i][1]
29
30     # l'intervallo che stiamo considerando inizia dopo
31     # l'ultimo intervallo che ci eravamo salvati?
32     if(inizio >= fine_ultimo):
33         contatore += 1
34         fine_ultimo = fine
35
36 output_file.write(str(contatore))
```



Tecniche greedy

- L'approccio che abbiamo appena visto è detto tecnica *greedy* (avidò in italiano), perchè stiamo risolvendo un problema facendo di volta in volta la scelta migliore localmente, cioè dipendenti solo da quello che si sa al momento della scelta. Nonostante questo, l'unione di queste scelte, in questo caso, porta alla fine alla soluzione migliore possibile.
- Bisogna ovviamente essere sicuri che la tecnica greedy possa essere applicata, cosa che è vera solo per alcuni problemi...





La pizza degli Hamtaro

- Si assume che sia sempre possibile portare tutti gli Hamtaro in pizzeria.

Esempio

File input.txt	File output.txt
2 4	
2 1	
2 1	
1 5	
1 4	

La pizza degli Hamtaro (pizza)

Livello di difficoltà D = 2.(tempo limite 2 sec.)

La numerosa famiglia degli Hamtaro, composta da N criceti, ha prenotato un tavolo in una nota pizzeria. I membri si danno appuntamento presso un autonoleggio con M automobili a disposizione per raggiungere successivamente la pizzeria. Purtroppo gli Hamtaro non arrivano al volante e quindi devono pagare generosamente l'unico autista a disposizione dell'autonoleggio in quel momento. Nell'ambiente dei cartoni gli Hamtaro sono notoriamente dei taccagni e vogliono perciò spendere il meno possibile per la serata, pena il passare la cena a pianger miseria. Ogni automobile ha solo il carburante necessario per un viaggio autonoleggio-pizzeria-autonoleggio e non esistono distributori di carburante in zona: per cui dopo un viaggio con un'automobile i , l'automobile i rimane a secco e non può più essere usata. L'automobile i permette il trasporto di P_i membri della famiglia degli Hamtaro, al costo di E_i euro per criceto. Gli Hamtaro scelgono, ad ogni partenza, un'automobile tra quelle disponibili (tra quelle, cioè, non usate precedentemente) e l'autista la usa per accompagnare una parte di loro in pizzeria. Aiuta la famiglia a risparmiare indicandole qual è la minima cifra che dovrà spendere per far arrivare tutti gli N Hamtaro in pizzeria!

Dati di input

Il file input.txt contiene sulla prima riga i due interi positivi N e M, separati da uno spazio. Le successive M righe (per $i = 1, 2, \dots, M$) contengono ciascuna due numeri interi positivi separati da uno spazio, a rappresentare il costo per criceto e la capacità dell'automobile: il primo intero indica E_i mentre il secondo intero indica P_i .

Dati di output

Il programma, dopo aver letto il file di input, deve scrivere una sola riga nel file output.txt contenente un intero positivo che rappresenta la minima quantità di denaro necessaria per far arrivare tutti gli Hamtaro in pizzeria.

Assunzioni

1. $1 \leq N \leq 4000$
2. $1 \leq M \leq 4000$
3. $1 \leq E_i \leq 1000$
4. $1 \leq P_i \leq 1000$



Soluzione

- Semplice, basta selezionare sempre la macchina col costo più basso e portare più hamtarō possibili 😊.
Finiti gli “utilizzi” a disposizione, usiamo la seconda macchina col costo più basso e così via...
- La scelta greedy qui è l’utilizzo delle macchine con i costi più bassi, che porta ad una soluzione ottima in cui spendiamo il meno possibile

```
1 import operator
2
3 input_file = open("./input.txt", mode = "r")
4 output_file = open("./output.txt", mode = "w")
5
6 riga = input_file.readline().split()
7 hamtarō = int(riga[0])
8 macchine = int(riga[1])
9 lista = []
10 portati = 0
11 spesa_totale = 0
12
13 for i in range(macchine):
14     riga = input_file.readline()
15     intervallo = riga.split()
16     inizio = int(intervallo[0])
17     fine = int(intervallo[1])
18     lista.append((inizio, fine))
19
20 # la funzione sorted() restituisce la lista ordinata
21 # operator.itemgetter(0) specifica vogliamo ordinare in
22 # base al primo elemento
23 lista = sorted(lista, key = operator.itemgetter(0))
24
25 i = 0
26 while(portati < hamtarō):
27     portati += lista[i][1]
28     spesa_singola = lista[i][0]
29     spesa_totale += spesa_singola * lista[i][1]
30     # se ne abbiamo presi troppo dobbiamo lasciarne giù portati - hamtarō (richiesti)
31     if(portati > hamtarō):
32         spesa_totale -= (portati - hamtarō) * spesa_singola
33     i += 1
34
35 output_file.write(str(spesa_totale))
```



Ricorsione



Ricordiamo cos'è una funzione...

- Riducono la duplicazione di codice
- Raggruppano più operazioni per creare una più complessa (es. stampa_array(v))
- Semplificano la scrittura e la lettura del codice da parte di altri (dove "altri" comprende anche voi il giorno dopo che lo avete scritto 😊)
- Molti problemi possono essere risolti con le funzioni ricorsive

```
1 def somma(a, b):  
2     return a + b;  
3  
4 print(somma(2, 2)) # 4  
5 print(somma(2, 3)) # 5  
6 print(somma(4, 4)) # 8
```

Ogni funzione è composta di 4 elementi:

- Nome
- Lista di parametri
- Valore di ritorno
- Corpo/codice





Ricorsione

Ma che succede se somma() chiama se stessa 😱?



```
1 def somma(a, b):  
2     somma(a, b)  
3     return a + b;  
  
4  
5 print(somma(2, 2)) # Death  
6 print(somma(2, 3))  
7 print(somma(4, 4))
```

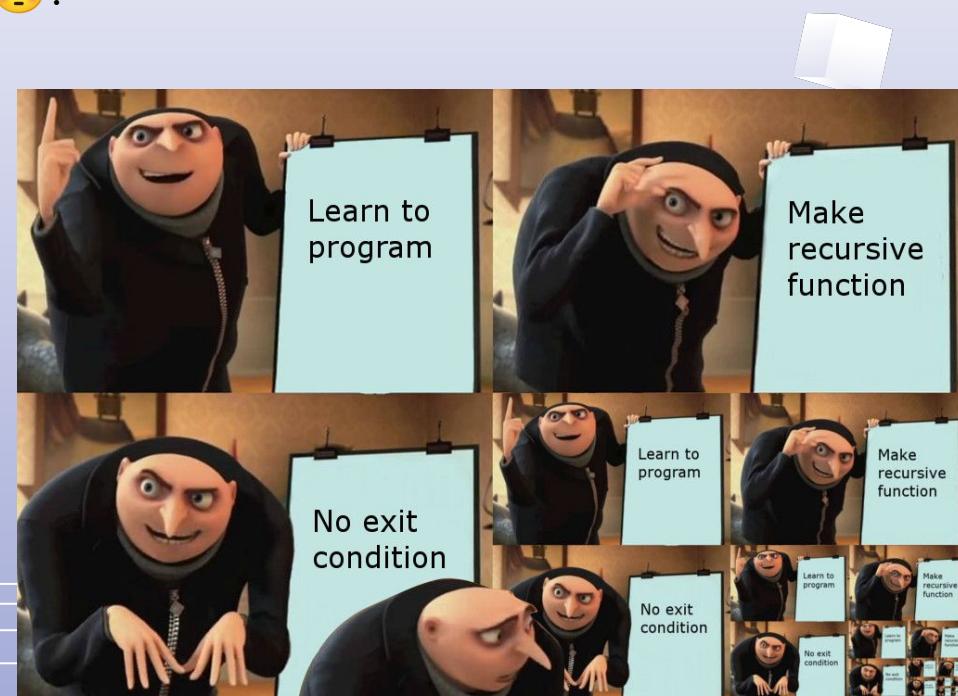




Ricorsione

Ma che succede se somma() chiama se stessa 😳?

In breve...





Una possibile metafora

Supponiamo di avere una matrioska chiusa





Una possibile metafora

Supponiamo di voler mettere una di fianco all'altra tutte le 5 bambole che la compongono, cioè il nostro obiettivo è questo:





Una possibile metafora

- ★ Chiamiamo bambola **numero 1** quella più grande e bambola **numero 5** la più piccola.
- ★ Ogni bambola è ovviamente composta da due pezzi: **la testa e la base**.
- ★ Aggiungiamo infine una piccola regola: non possiamo mai prendere una matrioska dalla base, ma solo dalla testa



Una possibile metafora

Come si procede? Ragioniamo
ricorsivamente.

Raccogliamo la testa della bambola 1, che poi è l'unico pezzo accessibile.

Ora ci serve la base della 1, che però non è accessibile perché è sotto la 2, che dobbiamo prima togliere.

Sospendiamo allora la composizione della bambola 1 e concentriamoci sulla bambola 2.





Una possibile metafora

- ★ Raccogliamo **la testa della bambola 2**, che è accessibile perché abbiamo tolto la 1. Ora ci serve **la base della 2**, che però **non è accessibile perché è sotto la 3**, che dobbiamo prima togliere. Sospendiamo allora la composizione della bambola 2 e **concentriamoci sulla bambola 3**.
- ★ Raccogliamo **la testa della bambola 3**, che è accessibile perché abbiamo tolto la 2. Ora ci serve **la base della 3**, che però **non è accessibile perché è sotto la 4**, che dobbiamo prima togliere. Sospendiamo allora la composizione della bambola 3 e **concentriamoci sulla bambola 4**.
- ★ Raccogliamo **la testa della bambola 4**, che è accessibile perché abbiamo tolto la 3. Ora ci serve **la base della 4**, che però **non è accessibile perché è sotto la 5**, che dobbiamo prima togliere. Sospendiamo allora la composizione della bambola 4 e **concentriamoci sulla bambola 5**.



Una possibile metafora

Ma qui accade qualcosa di speciale...

La bambola 5 è un pezzo solo!
Quindi finalmente siamo riusciti a completare
una delle 5 bambole!
La tiriamo fuori e la mettiamo da parte.





Una possibile metafora

- ★ A questo punto, non solo abbiamo risolto la bambola 5, ma **abbiamo accesso alla base della 4**. La raccogliamo e, unendola alla testa n. 4 precedentemente estratta, **componiamo la bambola 4 e la mettiamo da parte**.
- ★ A questo punto, non solo abbiamo risolto la bambola 4, ma **abbiamo accesso alla base della 3**. La raccogliamo e, unendola alla testa n. 3 precedentemente estratta, **componiamo la bambola 3 e la mettiamo da parte**.
- ★ A questo punto, non solo abbiamo risolto la bambola 3, ma **abbiamo accesso alla base della 2**. La raccogliamo e, unendola alla testa n. 2 precedentemente estratta, **componiamo la bambola 2 e la mettiamo da parte**.
- ★ A questo punto, non solo abbiamo risolto la bambola 2, ma **abbiamo accesso alla base della 1**. La raccogliamo e, unendola alla testa n. 1 precedentemente estratta, **componiamo la bambola 1 e la mettiamo da parte**.



Una possibile metafora

Abbiamo finito!





Una possibile metafora

- ★ Nella ricorsione si risolve un problema di dimensione N “assumendo” di essere in grado di risolverne uno di dimensioni $N-1$.
- ★ Guardando la procedura soltanto dal punto di vista del primo livello, per comporre la bambola numero 1 facciamo questo:
 - Prendi la testa numero 1
 - Componi la bambola numero 2
 - Prendi la base numero 1
- ★ Ma il punto 2 non è altro che lo stesso problema iniziale spostato su una bambola più piccola. In questo modo si passa a problemi di dimensioni sempre minori finché non si arriva ad uno che si risolve banalmente: la bambola 5 è già risolta da sola!
- ★ Il problema più elementare (la bambola 5) a cui riconduciamo quelli più complessi si chiama **base della ricorsione**.



Una possibile metafora

- ★ È importante notare che prima di arrivare alla bambola 5, abbiamo dovuto aprire le altre 4 e posare sul tavolo le rispettive teste, ricordandoci di quale base restassero in attesa.
- ★ Quando i livelli di ricorsione (uno per bambola) sono molti, il tavolo si riempie di teste e per ognuna dovremo ricordarci a quale base corrisponde.
È per questo che nel contesto della programmazione la ricorsione può essere estremamente dispendiosa in termini di consumo di memoria, fino ad esaurire tutta quella a disposizione.
- ★ In quel caso la ricorsione termina rovinosamente nel famigerato **stack overflow**.





Ricorsione

Fare questo quindi non serve a molto, se non per divertirsi a far crashare il programma...

Possiamo però utilizzarle in maniera intelligente! 😊

Vediamo come si potrebbe scrivere una funzione per calcolare il fattoriale di un numero

Possiamo notare per esempio che $4! = 4 * 3! = 4 * 3 * 2! = 4 * 3 * 2 * 1! = 4 * 3 * 2 * 1 * 0! = 4 * 3 * 2 * 1 * 1 = 24$

```
1 def fattoriale(numero):
2     if numero == 0:
3         return 1
4     else:
5         return numero * fattoriale(numero - 1)
6
7 print(fattoriale(4)) # 24
```

```
9 def fattoriale(numero):
10     f = 1
11     for i in range (1, numero + 1):
12         f *= i
13     return f
14
15 print(fattoriale(4)) # 24
```

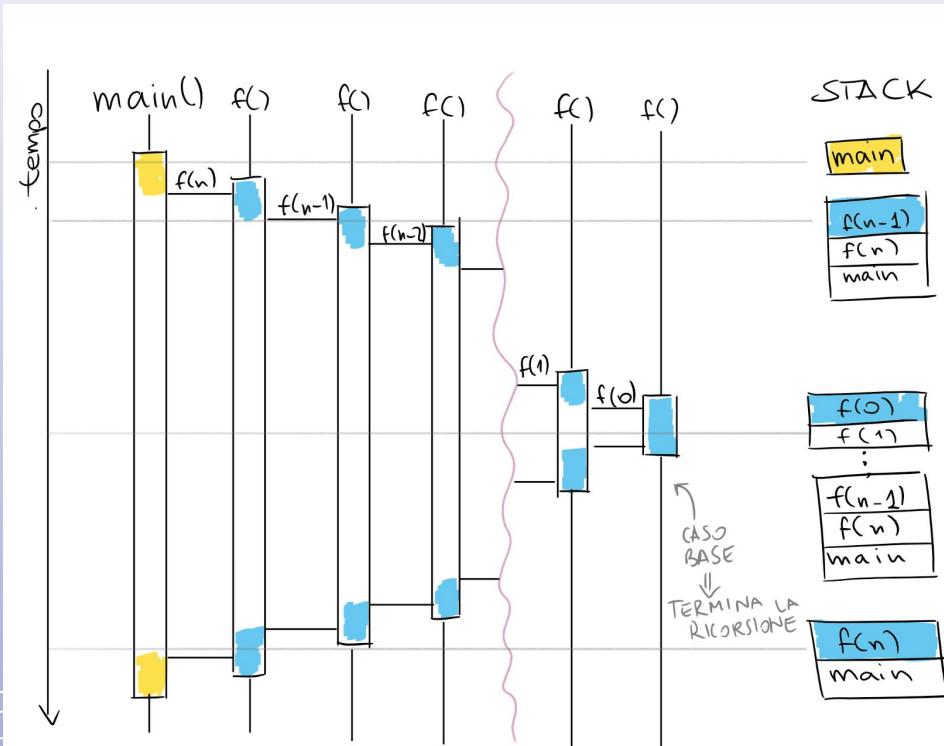


Non è tutto oro quel che luccica...

Ma cosa succede internamente?

Durante le chiamate ricorsive il programma ha necessità di tenere memoria dei valori "sospesi" e per far questo utilizza lo stack, un'area di memoria pensata per memorizzare cose come le variabili locali, i parametri, l'indirizzo e il valore di ritorno.

Se ad esempio l'input è $n=100000$ le chiamate ricorsive che devono essere memorizzate sono 100000, prima di poter arrivare alla condizione base e procedere al calcolo a ritroso.

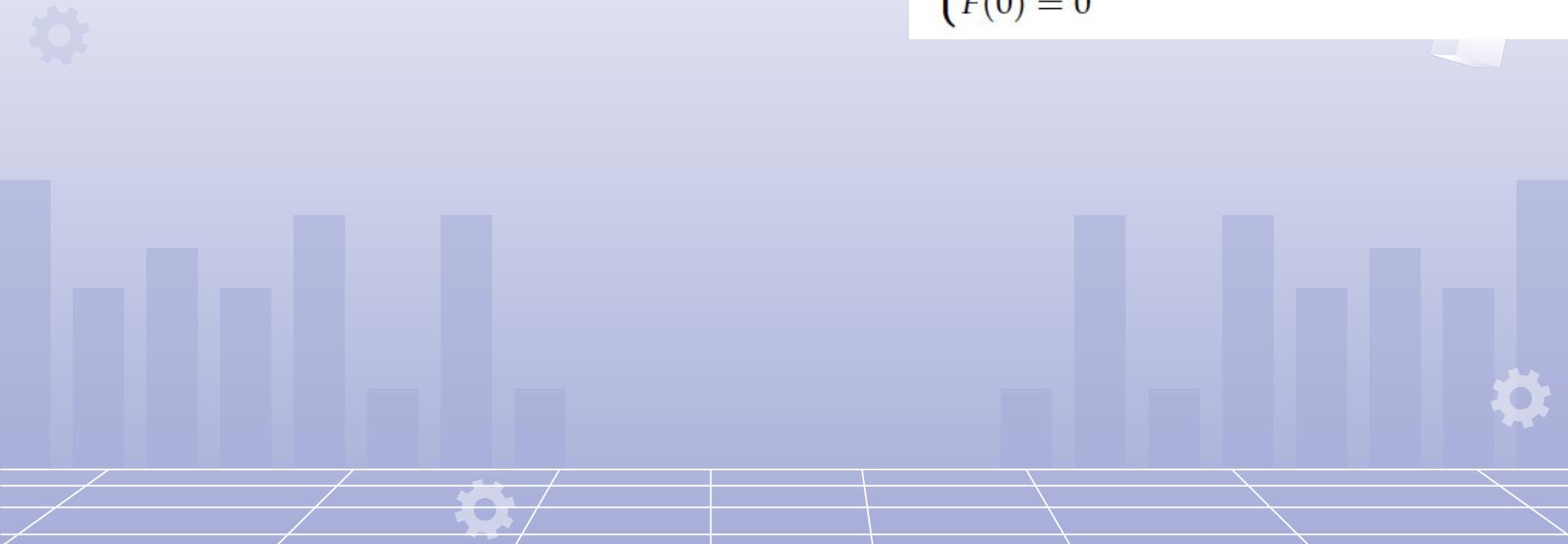




Numero di Fibonacci

Provate ora a calcolare il numero di Fibonacci...

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





Numero di Fibonacci

Provate ora a calcolare il numero di Fibonacci...

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

Soluzione:

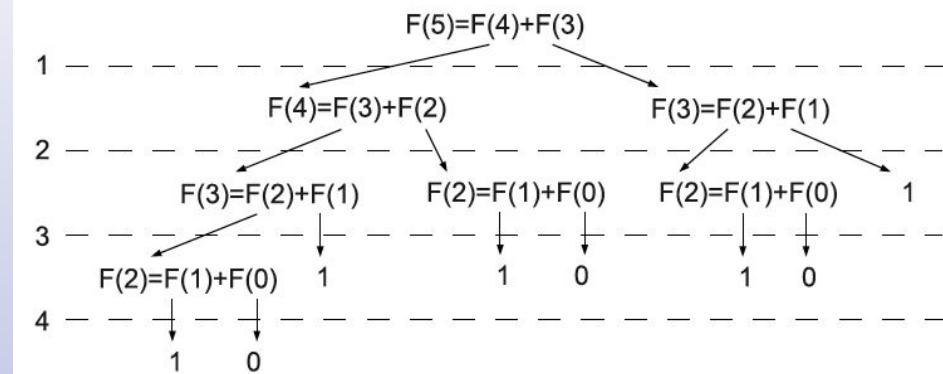
```
def fibonacci(numero):
    if numero == 0:
        return 0
    if numero == 1:
        return 1
    return fibonacci(numero - 1) + fibonacci(numero - 2)

print(fibonacci(5)) # 5
```



Numero di Fibonacci

Riassunto del fattoriale di 5:



Notevole differenza rispetto al fattoriale

La funzione richiama 2 volte sé stessa, su due versioni più piccole del problema di partenza. Anche se può risultare non immediatamente evidente, questo fa sì che il numero di chiamate ricorsive non cresca linearmente all'aumentare di n , ma esponenzialmente!

Provate a calcolare il fibonacci di 50 😊





Quindi sì o no all'uso ricorsione?

In alcuni casi, pur essendo la soluzione ricorsiva comoda da implementare, di fatto non può essere utilizzata per i limiti di performance che ha

Tuttavia...



Alcuni problemi e strutture dati hanno una definizione intrinsecamente ricorsiva, ad esempio tutti i problemi definiti per induzione. Sapere come tradurre la soluzione questi problemi in una funzione ricorsiva può quindi permetterci di avere in tempi brevi una soluzione che risolve almeno le istanze più semplici del problema.

Inoltre, molte volte non tutti i rami di esecuzione vengono svolti, riducendo così la mole di operazioni da eseguire 😺





Mappa antica

Mappa antica (mappa)

Difficoltà D = 2.

Descrizione del problema

Topolino è in missione per accompagnare una spedizione archeologica che segue un'antica mappa acquisita di recente dal museo di Topolinia. Raggiunta la località dove dovrebbe trovarsi un prezioso e raro reperto archeologico, Topolino si imbatte in un labirinto che ha la forma di una gigantesca scacchiera quadrata di $N \times N$ lastroni di marmo.

Nella mappa, sia le righe che le colonne del labirinto sono numerate da 1 a N . Il lastrone che si trova nella posizione corrispondente alla riga r e alla colonna c viene identificato mediante la coppia di interi (r, c) . I lastroni segnalati da una crocetta '+' sulla mappa contengono un trabocchetto mortale e sono quindi da evitare, mentre i rimanenti sono innocui e segnalati da un asterisco '*'.

Topolino deve partire dal lastrone in posizione $(1, 1)$ e raggiungere il lastrone in posizione (N, N) , entrambi innocui. Può passare da un lastrone a un altro soltanto se questi condividono un lato o uno spigolo (quindi può procedere in direzione orizzontale, verticale o diagonale ma non saltare) e, ovviamente, questi lastroni devono essere innocui.

Tuttavia, le insidie non sono finite qui: per poter attraversare incolume il labirinto, Topolino deve calpestare il minor numero possibile di lastroni innocui (e ovviamente nessun lastrone con trabocchetto). Aiutate Topolino a calcolare tale numero minimo.



Mappa antica



Dati di input

Il file `input.txt` è composto da $N+1$ righe.

La prima riga contiene un intero positivo che rappresenta la dimensione N di un lato del labirinto a scacchiera.

Le successive N righe rappresentano il labirinto a scacchiera: la r -esima di tali righe contiene una sequenza di N caratteri '+' oppure '*', dove '+' indica un lastrone con trabocchetto mentre '*' indica un lastrone sicuro. Tale riga rappresenta quindi i lastroni che si trovano sulla r -esima riga della scacchiera: di conseguenza, il c -esimo carattere corrisponde al lastrone in posizione (r, c) .

Dati di output

Il file `output.txt` è composto da una sola riga contenente un intero che rappresenta il minimo numero di lastroni innocui (ossia indicati con '*') che Topolino deve attraversare a partire dal lastrone in posizione $(1, 1)$ per arrivare incolume al lastrone in posizione (N, N) . Notare che i lastroni $(1, 1)$ e (N, N) vanno inclusi nel conteggio dei lastroni attraversati.

Assunzioni

$1 \leq N \leq 100$. $1 \leq r, c \leq N$. È sempre possibile attraversare il labirinto dal lastrone in posizione $(1, 1)$ al lastrone in posizione (N, N) ; inoltre tali due lastroni sono innocui.

Esempi di input/output

File input.txt	File output.txt
4 *+++ +*** +*** +***	5





Soluzione



```
1 N = 0
2 mappa = []
3
4 def fuori_matrice(i, j, y, x):
5     if i + y < 0 or i + y > N - 1 or j + x < 0 or j + x > N - 1:
6         return True
7     else:
8         return False
9
10 def attraversa(i, j):
11     for y in range(-1, 2):
12         for x in range(-1, 2):
13             if fuori_matrice(i, j, y, x):
14                 # Non fare niente ...
15                 # Il for sottostante non sarà eseguito
16                 pass
17             # Se la cella in cui vogliamo andare è sicura
18             # oppure abbiamo trovato un percorso migliore
19             elif mappa[i + y][j + x] == 0 or mappa[i + y][j + x] > mappa[i][j] + 1:
20                 # Aggiungiamo 1 a quella cella
21                 mappa[i + y][j + x] = mappa[i][j] + 1
22                 # Andiamoci!
23                 attraversa(i + y, j + x)
24
25 input_file = open("./input.txt", mode = "r")
26 output_file = open("./output.txt", mode = "w")
27 N = int(input_file.readline())
28
29 for i in range(N):
30     line = input_file.readline()
31     riga = []
32     for j in range(N):
33         if line[j] == '*':
34             riga.append(0)
35         else:
36             riga.append(-1)
37     mappa.append(riga)
38
39 mappa[0][0] = 1
40 attraversa(0, 0)
41 output_file.write(str(mappa[N - 1][N - 1]))
```





Accenno alla complessità computazionale

- ★ Come avete potuto vedere in queste lezioni, ci sono moltissimi modi per risolvere un problema...
- ★ Non tutti però vi fanno fare il massimo dei punti e abbiamo accennato che alcune soluzioni sono troppo “inefficienti”. Un esempio sono i problemi che tentiamo di risolvere con la ricorsione, provando tutte le combinazioni possibili.
- ★ Ma c'è un modo rigoroso per valutare l'efficienza di una vostra soluzione? 🤔





Accenno alla complessità computazionale

- ★ La risposta ovviamente è sì ma non è così semplice...
Ci vorrebbero tutte le lezioni da qui fino a febbraio per capire a fondo questo argomento...
- ★ Meglio che semplifichiamo un po' il tutto.
Proviamo a vedere diversi modi per risolvere il problema qui sotto, dal più inefficiente al più efficiente.

Problema: Data una stringa di caratteri alfabetici minuscoli (da 'a' a 'z'), determinare se la stringa ha tutti caratteri unici.

Esempio:

Input:	Output:	Input:	Output:
ciao	SI	casa	NO



Accenno alla complessità computazionale

★ Soluzione 1

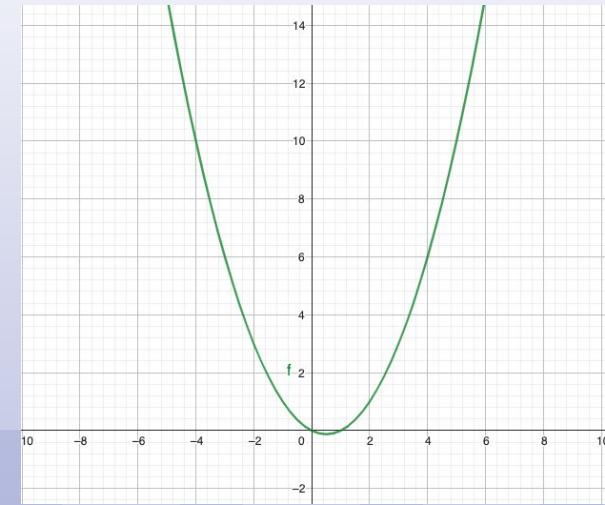
- ★ Per ogni carattere controlliamo se nella parola ci sono doppioni.
- ★ Quanti controlli facciamo?
Nella stringa ci sono N caratteri.
Per ogni carattere controlliamo nei rimanenti se ci sono doppioni.
Quindi partiamo controllando $N - 1$ caratteri, poi $N - 2$, poi $N - 3$, ecc..

```
def has_unique_chars(string):  
    length = len(string)  
  
    for i in range(length):  
        for j in range(i + 1, length):  
            if string[i] == string[j]:  
                return False  
  
    return True  
  
string = "caleidoscopio"  
result = has_unique_chars(string)  
  
if result:  
    print("SI")  
else:  
    print("NO")
```



Accenno alla complessità computazionale

- ★ I più bravi di voi sapranno che questa sequenza equivale alla formula di Gauss, ossia $N(N-1)/2 = \frac{1}{2}N^2 - \frac{1}{2}N$. Una parabola!
- ★ Possiamo quindi dire che il nostro algoritmo ha una complessità simile a quella di una parabola (non ci interessa di quanto è traslata), ossia N^2 .
- ★ Quindi all'aumentare della lunghezza della parola che diano in input, i confronti aumentano abbastanza velocemente.





Accenno alla complessità computazionale

- ★ Se approfondite online, troverete che in matematica la complessità di un algoritmo si esprime con la lettera O.
- ★ Quindi in questo caso, la complessità si dice **O(N²)**
- ★ Per chi invece non se ne intende di matematica, possiamo dire che ci sono due **for annidati** che confrontano circa N elementi ciascuno, quindi N²!





Accenno alla complessità computazionale

★ Soluzione 2

★ Ordino la stringa, e poi controllo se ogni carattere di fianco a sé ha un carattere uguale, se sì la stringa non è unica.

★ Quanti controlli facciamo?
Beh facciamo un solo **for** con **N controlli** no? 😎

```
def has_unique_chars(string):  
    sorted_string = sorted(string)  
  
    for i in range(len(sorted_string) - 1):  
        if sorted_string[i] == sorted_string[i + 1]:  
            return False  
  
    return True  
  
string = "caleidoscopio"  
result = has_unique_chars(string)  
  
if result:  
    print("SI")  
else:  
    print("NO")
```



Accenno alla complessità computazionale

- ★ E quel **sorted()**? Non sappiamo internamente come funziona, e si potrebbe dedicare un corso intero sullo studio del riordinamento di una stringa/vettore.
Sappiate che in media gli algoritmi di ordinamento hanno una complessità di **N * log(N)**, quindi già meglio rispetto all'esempio di prima ma si può fare ancora meglio.
- ★ Sappiate quindi che se non viene accettata una vostra soluzione che usa **sorted()** per "**Time limit exceeded**" vuol dire che non potete ordinare l'input, ma dovete lasciare gli elementi così come sono e usare un'altra strategia. Vediamo ora come!



Accenno alla complessità computazionale

★ Soluzione 3

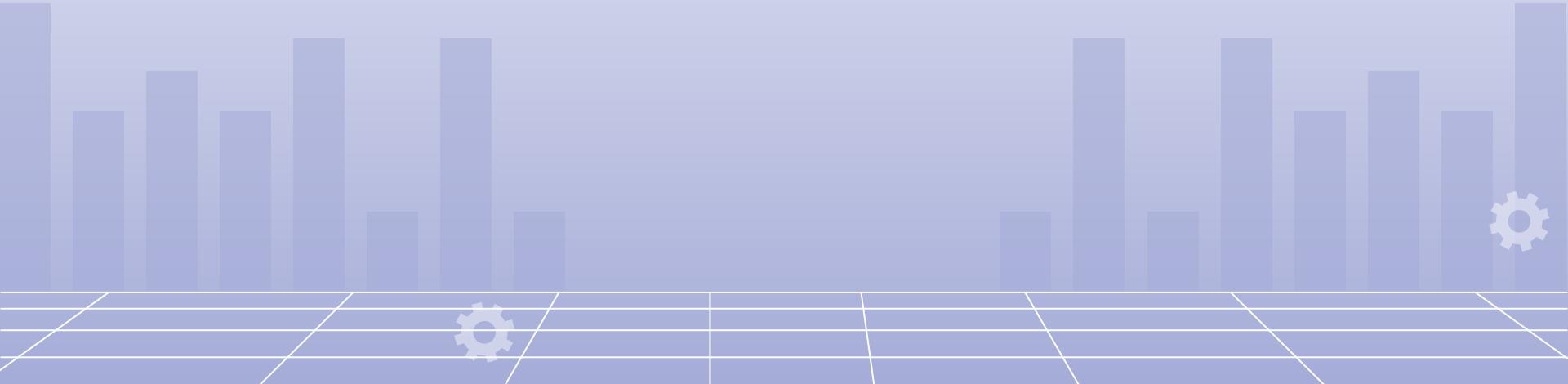
- ★ Creo un array di 26 elementi dove la posizione 0 corrisponde ad 'a', la posizione 1 a 'b', fino a posizione 25 per 'z'. Per ogni carattere imposta la sua rispettiva di cella a true. Se capito in una cella già a true stampo NO.
- ★ Quanti controlli facciamo? Controlloamo **N** caratteri, quindi la complessità è **O(n)**

```
def has_unique_chars(string):  
    seen_chars = [False] * 26  
  
    for char in string:  
        index = ord(char) - ord('a')  
        if seen_chars[index]:  
            return False  
        seen_chars[index] = True  
  
    return True  
  
string = "caleidoscopio"  
result = has_unique_chars(string)  
  
if result:  
    print("SI")  
else:  
    print("NO")
```



Vetrare colorate (OII 2023)

- Link al problema





Soluzione greedy

- ★ Funziona l'approccio greedy.

Scorriamo da sinistra a destra e ribaltiamo tutti gli intervalli ordinati in senso decrescente più lunghi possibile (così che ribaltati siano ordinati in senso crescente).

Alla fine controlliamo se l'array ottenuto è ordinato.

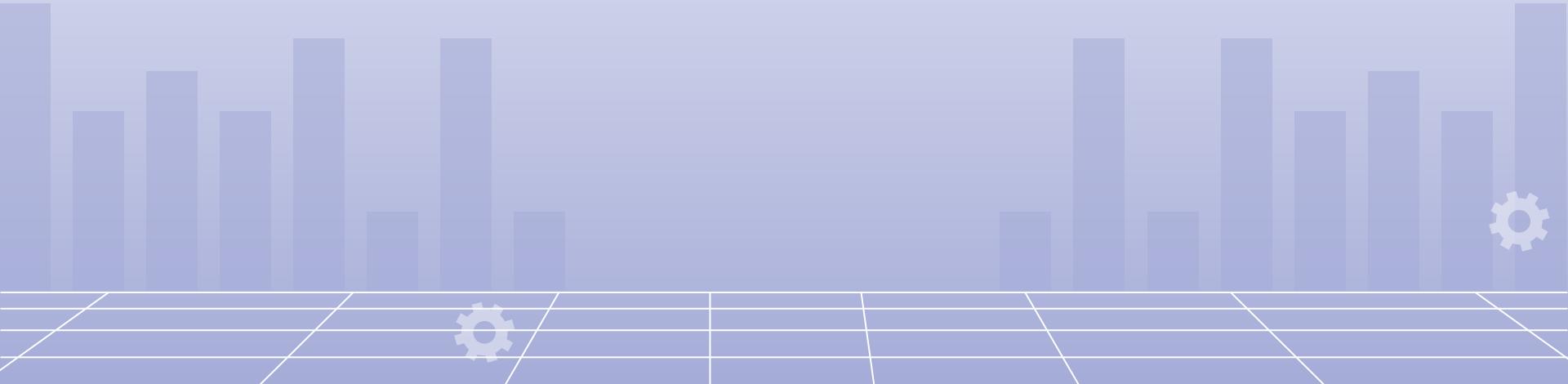
- ★ Notiamo che è obbligatorio spaccare se $V[i] < V[i+1]$. Se non spaccassimo, terremmo uniti una serie di vetri ordinati in ordine crescente, che verrebbero poi inevitabilmente ruotati. Ce li ritroveremmo poi in ordine decrescente.
- ★ Notiamo anche che, in caso di numeri uguali consecutivi, conviene spaccare più a destra possibile.

```
def array_swap(V, i, j):  
    while i < j:  
        V[i], V[j] = V[j], V[i]  
        i += 1  
        j -= 1  
  
def ordina(N, V, L):  
    i = 0  
    start = 0  
    counter = 0  
  
    while i < N:  
        if i == N - 1 or V[i] < V[i + 1]:  
            counter += 1  
            array_swap(V, start, i)  
            L.append(i - start + 1)  
            start = i + 1  
        i += 1  
  
    for i in range(N - 1):  
        if V[i] > V[i + 1]:  
            return False  
  
    return True
```



Police investigation 2

- Link al problema





Soluzione



```
N = int(input().strip())
V = list(map(int, input().strip().split()))
visited = [False] * N
maxRes = 0

for i in range(N):
    j = i
    x = i
    resSoFar = 0
    while visited[j] == False:
        visited[j] = True
        j = V[j]
        resSoFar += 1
    while x != j:
        x = V[x]
        resSoFar -= 1

    maxRes = max(maxRes, resSoFar)

print(maxRes)
```



Numero di Fibonacci

Riprendiamo ora il numero di Fibonacci

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

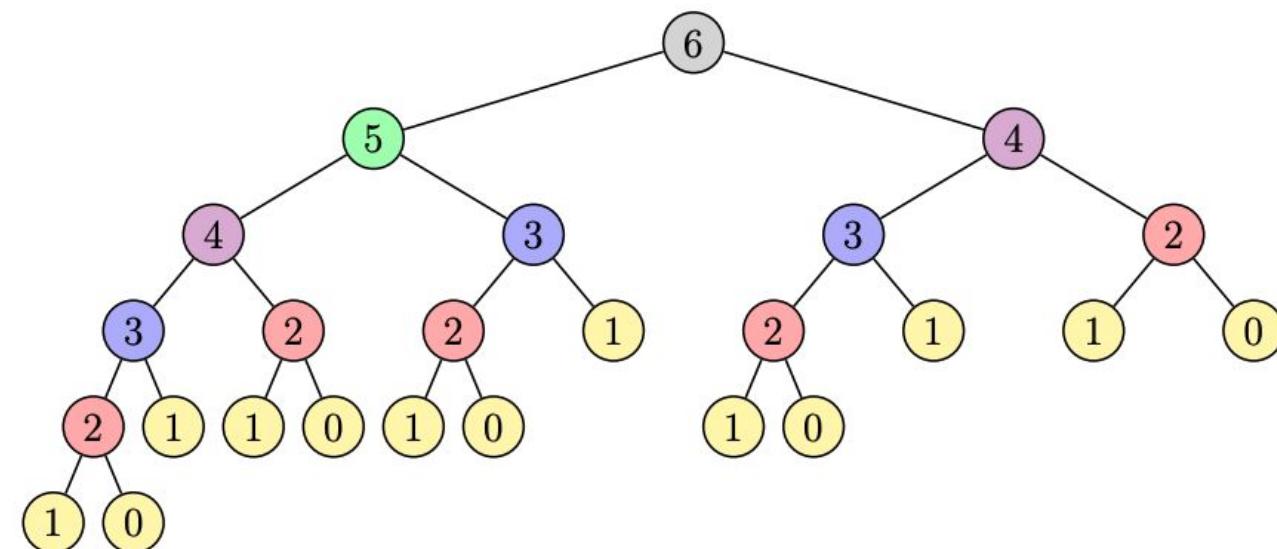
```
def fibonacci(numero):
    if numero == 0:
        return 0
    if numero == 1:
        return 1
    return fibonacci(numero - 1) + fibonacci(numero - 2)

print(fibonacci(5)) # 5
```



Numero di Fibonacci

Riflettiamo ora sul Fibonacci di 6:



Molti sotto-problemi ripetuti!



Numero di Fibonacci

- ★ Come potete notare, dobbiamo ricalcolarci il Fibonacci di 4 due volte, quello di 3 tre volte, quello di 2 quattro volte...
- ★ Non possiamo invece calcolarceli tutti una volta sola?
- ★ Possiamo salvarceli in una lista! 

n	1	2	3	4	5	6
$DP[]$	1	1	2	3	5	8

A blue double-headed arrow points from the value 3 in the $DP[]$ row to the cell containing 3 in the original table, illustrating how the current value is being calculated as the sum of the previous two values.



Numero di Fibonacci

- ★ Quindi, se vogliamo calcolare il Fibonacci di 6, partiamo calcolando il Fibonacci di 2, che poi ci servirà per il Fibonacci di 3, che poi ci servirà per il Fibonacci di 4 e così via fino al Fibonacci di 6. In questo modo li stiamo calcolando tutti una sola volta dal basso verso l'alto... molto efficiente! 😊

n	1	2	3	4	5	6
$DP[]$	1	1	2	3	5	8

A blue double-headed arrow points from the value 1 in the $DP[2]$ cell to the $DP[3]$ cell, indicating the recursive step where the value for $n=3$ is being calculated by summing the previous two values.



Numero di Fibonacci

- Vediamo come potremmo scriverlo in python:

```
def fibonacci(numero):
    DP = [0] * (numero + 1)
    DP[0] = 0
    DP[1] = 1

    for i in range (2, numero + 1):
        DP[i] = DP[i - 1] + DP[i - 2]

    return DP[numero]

print(fibonacci(6))
```



Programmazione dinamica

- ★ Cosa abbiamo appena fatto? Abbiamo usato una tecnica chiamata **programmazione dinamica!**

Dalle slide del mio professore di algoritmi:

- ★ Un metodo per spezzare un problema ricorsivamente in sottoproblemi
- ★ Ogni sottoproblema viene risolto una volta sola
- ★ La sua soluzione viene memorizzata in una tabella
- ★ Nel caso un sottoproblema debba essere risolto nuovamente, si ottiene la sua soluzione dalla tabella



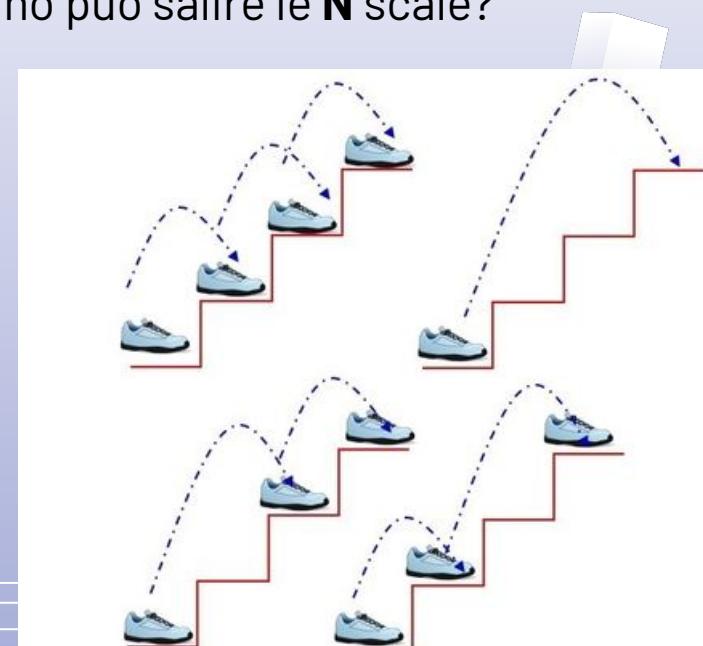


Esercizio su programmazione dinamica

Un bambino sta salendo una scala con **N** gradini e può salire di **1** gradino, **2** gradini o **3** gradini alla volta. In quanti modi il bambino può salire le **N** scale?

Input:	Output:
3	4

Perchè 4 ->





Esercizio su programmazione dinamica

Un bambino sta salendo una scala con **N** gradini e può salire di **1** gradino, **2** gradini o **3** gradini alla volta. In quanti modi il bambino può salire le **N** scale?

Input:	Output:
4	7

Spiegazione: Il bambino può salire le scale nei seguenti modi:

1111

121

211

112

22

31

13



Esercizio su programmazione dinamica

- ★ **Soluzione:** Ci sono N scale e una persona può saltare al gradino successivo, saltare due gradini o saltarne tre. Quindi, se una persona si trova al i-esimo gradino, ci è arrivata o dal gradino $i - 1$ saltando 1, o dal gradino $i - 2$ saltando 2 o dal gradino $i - 3$ saltando 3.
- ★ Dunque un gradino è la somma di "come sono arrivato" ai 3 precedenti, ossia **gradino($i - 1$) + gradino($i - 2$) + gradino ($i - 3$)**

Molto simile al numero
di Fibonacci!

```
def scala(numero):
    DP = [0] * (numero + 1)
    DP[0] = 1
    DP[1] = 1
    DP[2] = 2

    for i in range (3, numero + 1):
        DP[i] = DP[i - 1] + DP[i - 2] + DP[i - 3]

    return DP[numero]
```



Esercizio con i dizionari

★ [Link al problema](#)

