Indice

martedì 1 luglio 2025 12:58

1) **Riassunto**
   - 1.1 Grafi
   - 1.2 Correttezze
   - 1.3 Dimostrazioni
   - 1.4 Riassunto Ristretto
   - 1.5 Progr. Dinamica
   - 1.6 Logaritmi
   - 1.7 Sommatorie Importanti
2) **Algoritmi**
   - 2.1 Efficienza
   - 2.2 Limitazione inferiore
3) **Grafi**
   - 3.1 Grafi
   - 3.2 Grafi ciclici
   - 3.3 DFS
   - 3.4 Grafi Colorati
   - 3.5 Componenti
   - 3.6 Ordinamento Topologico
   - 3.7 Ricerca Cicli
   - 3.8 Ponti
   - 3.9 BFS e Distanze
   - 3.10 Grafi Pesati
   - 3.11 UNION-FIND
   - 3.12 Minimo Albero di Copertura
   - 3.13 Cicli Negativi
4) **Progr. Dinamica**
   - 4.1 Ottimizzazione
   - 4.2 Algoritmi Greedy
   - 4.3 Divide et Impera
   - 4.4 Programmazione Dinamica
     - 4.4.1 Programmazione Dinamica
     - 4.4.2 Tab. Bidimensionale
   - 4.5 Backtracking Monodimensionale
   - 4.6 Backtracking Bidimensionale
5) **Esercizi**
   - 5.1 Slide Introduzione
   - 5.2 Slide Grafi 1 (grafi)
   - 5.3 Slide Grafi 2 (DFS e padri)
   - 5.4 Slide Grafi 3 (colori e componenti)
   - 5.5 Slide Grafi 4 (sort topologico)
   - 5.6 Slide Grafi 5 (cicli)
   - 5.7 Slide Grafi 5 finale (cicli)
   - 5.8 Slide Grafi 6 (Ponti)
   - 5.9 Slide Grafi 8 (BFS)
   - 5.10 Slide Grafi Pesati
   - 5.11 Min albero copertura
   - 5.12 Tracce esercizi 2025
   - 5.13 Esonero 2024
   - 5.14 Esonero 2025 saltato
   - 5.15 Slide 12 (Approssimazione)
   - 5.16 Slide 13 (Greedy)

5.17 Slide 13b (Greedy)
5.18 Slide 17 (Progr Dinamica Unidimensionale)
5.19 Slide 18 (Progr Dinamica Bidimensionale)
5.20 Slide 19 (Backtracking 1)
5.21 Slide 20 (Backtracking 2)
5.22 Esercizi Preparatori all'Esame
5.23 Esercizi sul Diario delle Lezioni
5.24 Esercizi Programmazione Dinamica

6) Esame

6.1 Programma
6.2 Appunti 23-24 e 15-16
6.3 Esami
6.4 Esonero Aprile 2024
6.5 Febbraio 1
6.6 Gennaio 1
6.7 Giugno 1
6.8 Giugno 2
6.9 Giugno A
6.10 Giugno B
6.11 Luglio 1
6.12 Luglio 2
6.13 Luglio 3
6.14 Luglio 4
6.15 Settembre 1
6.16 Ottobre 1
6.17 Straordinario 1
6.18 Febbraio 2025

Grafi

lunedì 24 febbraio 2025 11:14

Grafo

Un grafo G è una **coppia (V, E)**, dove |V| = **n nodi**, |E| = **m archi** che **collegano i nodi**.
Un grafo può essere:

- **Diretto (o Orientato)**: archi con **una direzione** → collegamento tra nodi in **un solo verso** (max nodi per arco: n-1 → num. max archi: n*(n-1) = O(n²))
- **Indiretto**: archi **senza direzione** → collegamento tra nodi **bidirezionale** (max nodi per arco: $\frac{n-1}{2}$, arco condiviso tra due nodi → num. max archi: $\frac{n(n-1)}{2}$ = O(n²))
- **Sparso**: **pochi archi tra nodi rispetto al max**. Num. archi cresce al massimo come O(n) (non troppo velocemente rispetto al num. nodi) → m ≤ n
- **Denso**: **num. archi vicino al max**. Num. archi cresce almeno come Ω(n²) (vicino al max possibile) → m > n
  - **Grafo Completo:** ha il max num. di archi → |E| = Θ(n²)
- **Connesso**: un grafo è **connesso** se per ogni coppia di nodi esiste un **cammino che li collega** (non ci sono parti del grafo separate tra di loro)
  - **Grafo strettamente connesso**: se per ogni coppia di nodi esiste un **cammino che li collega** partenda da **entrambi i nodi** (nei grafi diretti)
- **Planare**: grafo che si può disegnare senza che gli archi si **intersechino**.
  - **Teorema (Eulero)**: un grafo planare di n > 2 ha al **massimo 3n - 6 archi**
- **Albero**: è un **grafo sparso connesso senza cicli**, che ha sempre:
  - **m archi = n-1 nodi**
  - Almeno un nodo con **grado 1** (**foglie**)
- **Ciclico**: un grafo è **ciclico** se esiste un **sottogruppo connesso** in cui ogni vertice ha grado ≥ 2.
  Se nel grafo **tutti i vertici** hanno **grado ≥ 2,** allora il grafo è **sicuramente ciclico**.
  In un grafo **diretto**, se ogni nodo ha **almeno un arco uscente** allora il grafo è **ciclico**
- **Grafo Bipartito**: si può **partizionare** i suoi vertici in **due insiemi V1 e V2** t.c. tutti gli archi abbiano **un estremo in V1 e l'alltro in V2**
- **Grafo Complementare**: è un grafo G^C il quale ha gli **stessi nodi** di G ma ogni arco è **presente solo se manca a G**
- **Euleriano:** un grafo indiretto si dice **euleriano se esiste un ciclo** che attraversa ogni arco del grafo **esattamente una volta e ritorna al nodo di partenza**

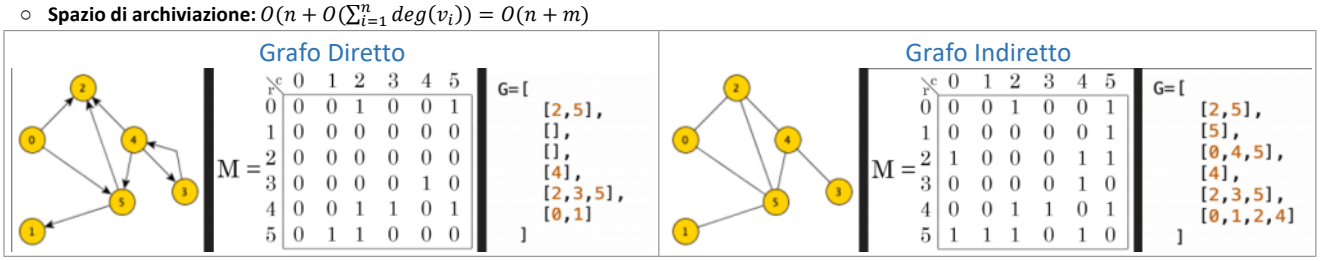
Definizioni

- **Nodi adiacenti**: due nodi collegati da un arco. Il **grado** di un nodo indica il num. di nodi **adiacenti ad esso** (sia in grafi diretti e indiretti).
- **Passeggiata/Cammino:**
  - Una **passeggiata** è una sequenza alternata di archi e nodi: v0, e1, v1, e2, v2, …, ek-1, vk-1, ek, vk
  - Un **cammino** è una passeggiata in cui **non si ripetono i nodi**
- **Componente connessa:** è un **sottografo** composto da un insieme **massimale** di nodi **connessi da cammini**. I diversi componenti sono una **partizione** del grafo
  - Un **grafo indiretto** si dice **connesso** se ha **una sola** componente
- **Componente fortemente connessa**: in un grafo **diretto** è un **sottografo** composto da un insieme **massimale** di nodi **connessi da cammini**
  - Un **grafo diretto** si dice **fortemente connesso** se ha **una sola** componente

Rappresentazione Grafo

La rappresentazione si può effettuare tramite:

- Matrici binarie: è una matrice di dimensioni n x n dove M[r][c] = 1 solo se c'è un arco diretto da r a c
  - Costo per controllare se r è vicino di c: O(1)
  - Spazio di archiviazione: O(n²)
- Liste di adiacenza: una lista di liste G
  - G ha tanti elem. quanti sono i nodi del grafo
  - G[x] è una lista contenente i nodi adiacenti al nodo x, cioè quelli raggiunti da archi che partono da x
  - **Costo** per controllare se **x è vicino di y**: O(n)
  - **Spazio di archiviazione:** $O(n + O(\sum_{i=1}^{n} deg(v_i)) = O(n+m)$



DFS (Visita in Profondità) (Slide Grafi 2)

La **DFS** è un metodo per **visitare un grafo in profondità** partendo da un nodo e spostandosi su un suo vicino casuale non ancora visitato

- **DFS Ricorsiva:** usa una **lista** per indicare i **nodi visitati**.
- **DFS Iterativa**: usa una **pila** per indicare i **nodi visitati**. Per ogni nodo che incontriamo lo salviamo nella pila e ci spostiamo sui vicini non visitati.
  Se tutti i vicini sono stati già visitati allora ritorniamo a quelli precedenti (back tracking o rollback), ritornando indietro sui nodi nella pila per visitare i vicini non visitati.

L'output contiene **tutti i nodi raggiungibili** dal **nodo di partenza**. Per ogni nodo x si ha **visitati[x] = 1** solo se il nodo è **raggiungibile**, 0 altrimenti

Ricorsiva su Matrice	Ricorsiva su Liste di Adiacenza

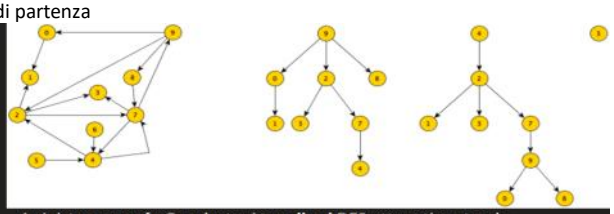
```python
def DFS_matrix(M, x):
    # visitati[a] = 1 -> il nodo è stato già visitato
    # visitati[a] = 0 -> il nodo deve ancora essere visitato
    ####
    def DFS_rec(M, x, visitati):
        visitati[x] = 1
        # impostiamo il nodo corrente a 1 per dire che è stato visitato
        for i in range(len(M)): # per ogni nodo del grafo
            if M[x][i] == 1 and not visitati[i]:
                # se c'è un arco tra x ed i, e i non è tra i nodi visitati
                DFS_rec(M, i, visitati) # controlliamo anche esso
    ####
    n = len(M)
    visitati = [0]*n # inizializziamo tutti i nodi a 0 (non visitati ancora)
    DFS_rec(M, x, visitati) # iniziamo la ricerca in profondità su x
    return [x for x in range(n) if visitati[x]]
    # ritorniamo i nodi visitati (uguali a 1)
```

- Costo: **O(n) x Θ(n) = O(n²)**
- Costo spazio: O(n) (lista visitati)

```python
def DFS_list(G, x):

    # visitati[a] = 1 -> il nodo è stato già visitato
    # visitati[a] = 0 -> il nodo deve ancora essere visitato
    ####
    def DFS_rec(G, x, visitati):
        visitati[x] = 1
        # impostiamo il nodo corrente a 1 per dire che è stato visitato
        for v in G[x]: # per ogni nodo adiacente a x
            if not visitati[v]: # se non è stato già visitato
                DFS_rec(G, v, visitati) # controlliamo anche esso
    ####
    n = len(G)
    visitati = [0]*n # inizializziamo tutti i nodi a 0 (non visitati ancora)
    DFS_rec(G, x, visitati) # iniziamo la ricerca in profondità su x
    return [x for x in range(n) if visitati[x]]
    # ritorniamo i nodi visitati (uguali a 1)
```

- Costo: **O(n + m)**
- Costo spazio: O(n) (lista visitati)

I nodi e gli archi **attraversati** formano un **albero DFS** con radice uguale al nodo di partenza



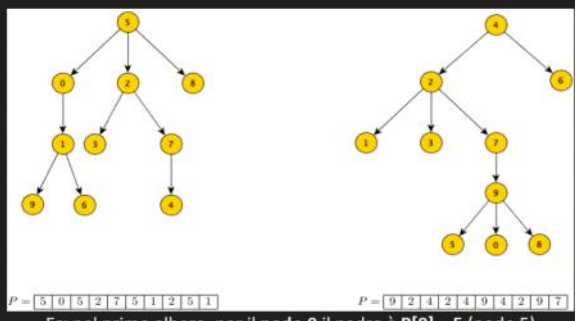
A sinistra un grafo G, a destra i tre **alberi DFS** ottenuti partendo rispettivamente dai nodi 9, 4 e 3

Vettore dei Padri

Un **albero DFS** può essere memorizzato con un **vettore dei padri**, il quale è un vettore in cui:

- Se **x è un nodo** dell'albero DFS → **P[x]** contiene il nodo **padre** di x (il padre della radice è la radice stessa)
- Se **x non è un nodo** dell'albero DFS → **P[x] = -1**

```python
def Padri(G, x):
    # P[a] = b -> b è il padre di a
    # P[a] = -1 -> a non è stato visitato
    ####
    def DFS_rec(G, x, P):
        for v in G[x]: # per ogni nodo adiacente a x
            if P[v] == -1: # se non è stato già visitato
                P[v] = x # il nodo da cui arriviamo sarà il padre
                DFS_rec(G, v, P) # controlliamo anche esso
    ####
    n = len(G)
    P = [-1]*n # inizializziamo tutti i nodi a -1 (non visitati ancora)
    P[x] = x # il padre della radice è la radice stessa
    DFS_rec(G, x, P) # iniziamo la ricerca in profondità su x
    return P # ritorniamo il vettore dei padri
```



Es: nel primo albero, per il **nodo 0** il padre è **P[0] = 5** (nodo 5)

Grazie al vettore dei padri radicato in x, possiamo determinare se esiste un **cammino** che va **da x a y.**
Basta vedere se **y è nell'albero** e restituire la lista in "reverse" dei nodi incontrati da x a y.

Metodo Iterativo

```python
def Cammino_iter(P, x):
    if P[x] == -1: return []
    # se il nodo non è nel vettore dei padri ritorniamo nulla
    path = [] # lista contenente i nodi che incontreremo
    while P[x] != x: # finchè il padre non arriviamo alla radice
        path.append(x) # aggiungiamo il nodo incontrato alla lista
        x = P[x] # risaliamo al nodo padre per controllarlo
    path. append(x) # aggiungiamo la radice
    path.reverse() # invertiamo la lista
    return path # e la ritorniamo
```

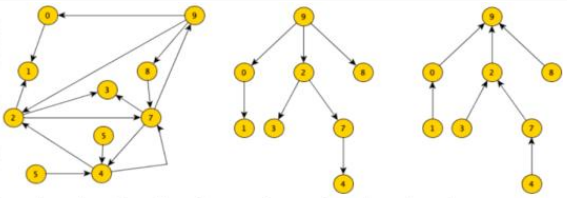
Costo: O(n)

Metodo Ricorsivo

```python
def Cammino_rec(P, x):
    if P[x] == -1: return []
    # se il nodo non è stato visitato, ritorniamo nulla
    if P[x] == x: return [x] # se siamo alla radice ritorniamo essa
    return Cammino_rec(P, P[x]) + [x]
    # ritorniamo i padri visitati ricorsivamente e il nodo corrente
```

Costo: O(n)

```
>>>P=padri(9,G)
>>>cammino(7,P)
[9,2,7]
>>>cammino(1,P)
[9,0,1]
>>>cammino(5,P)
[]
```



$P = \boxed{9 \mid 0 \mid 9 \mid 2 \mid 7 \mid -1 \mid -1 \mid 2 \mid 9 \mid 9}$

ATTENZIONE: questo algoritmo **non garantisce** di restituire **il cammino minimo**

Grafi Colorati (Slide Grafi 3)

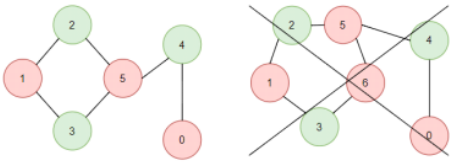
Un grafo connesso è **colorabile** se è possibile **colorare** i nodi in modo tale che quelli **adiacenti abbiano sempre colori distinti**

Teorema dei 4 Colori

- Dice che un **grafo planare** richiede **max 4 colori**.
- Non esiste algoritmo polinomiale che determini se un **grafo planare è 3-colorabile**
- Un grafo **è 2-colorabile se NON contiene cicli** di lunghezza **dispari**, poiché lungo il ciclo i colori devono **alternarsi** (cosa impossibile su un ciclo dispari)

Prova di Correttezza: siano x e y due nodi adiacenti in G, consideriamo i due possibili casi e vediamo che in entrambi i casi i due nodi al termine avranno colori opposti.

1) **L'arco (x, y) viene attraversato durante la visita**. In questo caso i due nodi avranno colori distinti.
2) **L'arco (x, y) NON viene attraversato durante la visita**:
   - Sia x il nodo visitato prima, **esiste un cammino** in G che da **x porta a y** (quello seguito dalla visita)
   - Questo cammino si chiude a formare un ciclo con l'arco (y, x).
   - Il **ciclo** è di lunghezza **pari** per ipotesi, quindi il **cammino** è di lunghezza **dispari.**
   - Poiché sul cammino i colori si **alternano**, il primo nodo (x) e l'ultimo (y) avranno **colori diversi**



Bicolorabile NON Bicolorabile

Algoritmo per bi-colorare grafi connessi G senza cicli dispari

```python
def Colora(G):
    # funziona solo su grafi connessi bicolorabili (senza cicli dispari)
    ####
    def DFS_rec(G, x, Colore, c):
        Colore[x] = c # assegna il colore al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a x
            if Colore[y] == -1: # se non è stato
                DFS_rec(G, y, Colore, 1-c)
                # lo colora con il colore inverso a quello di x
                # se x = 0 -> 1-0 = 1 -> y = 1
                # se x = 1 -> 1-1 = 0 -> y = 0
    ####
    Colore = [-1]*len(G) # inizializza i nodi a -1 (non colorati)
    DFS_rec(G, 0, Colore, 0) # partiamo la colorazione dal nodo 0
    return Colore # ritorniamo i nodi colorati
```

Algoritmo per bi-colorare grafi G se è bi-colorabile

```python
def Colora1(G):
    # Se il grafo è bicolorabile (senza cicli dispari) esegue la bicolorazione
    # altrimenti una lista vuota
    ####
    def DFS_rec(G, x, Colore, c):
        Colore[x] = c # assegna il colore al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a x
            if Colore[y] == -1: # se non è stato colorato
                if not DFS_rec(G, y, Colore, 1-c):
                    # se dal nodo successivo ci sono cicli dispari
                    return False # ritorna False (impossibile colorare)
            elif Colore[y] == Colore[x]:
                # se è stato colorato ed ha lo stesso colore del padre
                return False # ritorna False (impossibile colorare)
        return True # se non ci sono stati problemi ritorna True (colorato)
    ####
    Colore = [-1]*len(G) # inizializza i nodi a -1 (non colorati)
    if DFS_rec(G, 0, Colore, 0): # se non ci sono cicli dispari
        return Colore # ritorniamo la lista dei nodi colorati
    return [] # altrimenti la lista vuota
```

Nell'esempio a destra dell'immagine sopra avremmo che sul controllo al nodo rosso 6, vediamo che il nodo adiacente 5 ha Colore[5] = "rosso", quindi entra nell'elif e siccome i colori di 5 e 6 sono uguali, ritorna False.

Il costo è quello di una visita del grafo connesso da colorare.

O(n + m) = O(m)

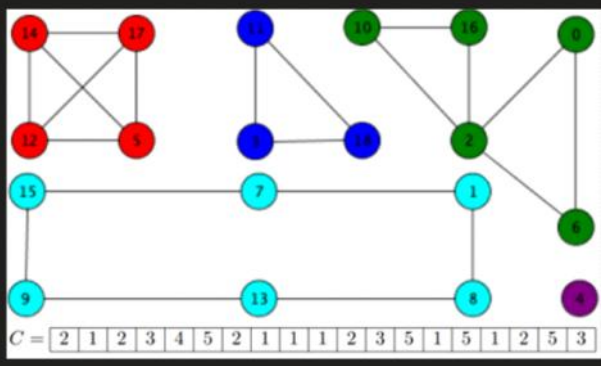
Poiché in un grafo connesso abbiamo che **m ≥ n - 1**

Componenti

Algoritmo che associa, in un vettore **Componenti**, ogni nodo ad un **indice** che ne indica la componente di cui fa parte, quindi:

Componenti[x] = Componenti[y] → se e solo se x e y sono **nella stessa componente**

```python
def Componenti(G):
    ####
    def DSF_rec(G, x, Comp, c):
        Comp[x] = c # assegna il val. del componente al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a quello corrente
            if C[y] == 0: # se non gli è stato gia assegnato il componente
                DFS_rec(G, y, Comp, c)
                # assegna il componente del nodo corrente a quello adiacente
    ####
    Comp = [0]*len(G)
    c = 1
    for x in range(len(G)): # per ogni nodo
        if C[x] == 0: # se non gli è stato assegnato il componente
            DFS_rec(G, x, Comp, c) # eseguiamo la ricerca in profondità
            # per cercare i nodi del componente
            c += 1 # finito il componente corrente,
            # aumentiamo c per passare al componente successivo
    return C
```



Componenti Fortemente Connessi

```python
def ComponenteFC(G, x):
    AVisit = DFS(G, x) # Visita DFS sul grafo normale
    GT = Trasposto(G) # Creiamo il Trasposto del grafo
    BVisit = DFS(GT, x) # Visita DFS sul grafo trasposto
    CompFC = []
    # prendiamo l'intersezione tra i due array
    for n in range(len(G)):
        if AVisit[n] == BVisit[n] == 1:
            # se entrambi sono stati visitati
            # quindi il nodo è:
            # - raggiungibile da x (grafo normale)
            # - porta ad x (grafo trasposto)
            CompFC.append(n) # lo aggiungiamo al componente
    return CompFC

def Trasposto(G):
    GT = [[] for n in G] # lista di adiacenze vuota
    for n in range(len(G)):
        for v in G[n]: # per ogni nodo adiacente
            GT[v].append(n)
            # aggiunge alla lista di adiacenze il nodo padre
    return GT

def CompFC(G):
    FC = [0]*len(G)
    c = 1
    for n in range(len(G)): # per ogni nodo di G
        if FC[n] == 0: # se non è stato visitato
            E = ComponenteFC(G, n)
            # prendiamo tutti i nodi della componente connessa di n
            for x in E: # per ognuno di quei nodi
                FC[x] = c # assegnamo il val. del componente
            c += 1 # e lo aumentiamo alla fine dell'assegnazione
    return FC
```

Idea per l'algoritmo ComponenteFC per la ricerca della componente connessa di un nodo x:

1) Calcola l'insieme **A** dei **nodi raggiungibili da x** (Semplice DFS. Costo O(n + m))
2) Calcola il **G^T trasposto di G** (Costo O(n + m))
3) Calcola l'insieme **B** dei **nodi che portano ad x** (DFS su **grafo trasposto.** Costo O(n + m))
4) Restituisce **l'intersezione** tra **A** e **B** (Costo O(n))

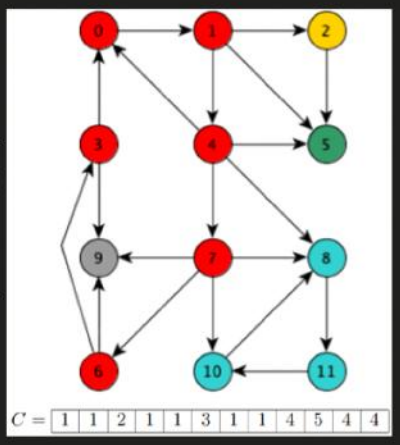
Costo totale **ComponenteFC**: O(n + m)

5) Grazie a **ComponenteFC** posso calcolare il vettore CF delle comp. fortemente connesse

Costo **CompFC**: ciclo nodi * ComponenteFC =

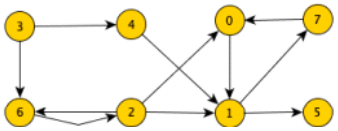
$\Theta(n) * O(n + m) = O(n^2 + n*m) =$ (in un grafo completo $m = O(n^2)$) $= O(n^2 + n^3) = O(n^3)$

```
>>>G=[
      [1], [2, 4, 5], [5], [1, 6],[0, 5, 7, 8],
      [], [3], [6, 8, 9, 10], [11], [],[8], [10]
      ]

>>> componenteFC(0, G)
[0,1,3,4,6,7]

>>> componenteFC(8, G)
[8,10,11]
```

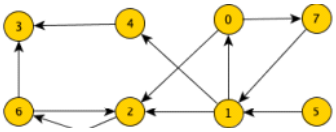


Il **grafo trasposto di G (G^T)** ha gli stessi nodi di G ma gli **archi** sono in **direzione opposta.** Cosi possiamo eseguire il passo 2 in tempo O(n + m) cercando i **raggiungibili da x** in G^T

G=



G^T=



Quindi i nodi che in G **portano ad x**, sono i nodi che in **G^T** sono **raggiungibili a partire da x**, e viceversa (i nodi in G che sono raggiungibili da x, in G^T portano ad x)

Es: nell'immagine i nodi in G che portano al nodo 5 sono **[1, 4, 0, 7, 2, 3, 6].** Allo stesso modo in G^T i nodi raggiungibili a partire dal nodo 5 sono **[1, 4, 0, 7, 2, 3, 6].**

Ordinamento Topologico (Slide Grafi 4)

L'**ordinamento topologico** è una disposizione dei nodi di un **grafo diretto aciclico** (DAG) in cui ogni arco vada **da sinistra verso destra**, rispettando le dipendenze.

Un grafo diretto può avere da **0 a n!** ordinamenti topologici. Se si ha un solo arco per n nodi, possiamo avere $\frac{n!}{2}$ ordinamenti diversi (formula generica: $\frac{n!}{m!}$)

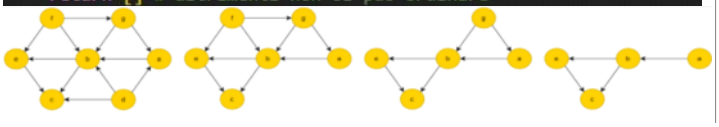
L'ordinamento si può effettuare solo su un DAG poiché se c'è un **ciclo** c'è una **dipendenza circolare**, che rende impossibile trovare l'ordine di esecuzione.

Algoritmi per l'Ordinamento Topologico

Tramite Sorgente del DAG

- Trova un **nodo sorgente** (senza archi entranti)
- Lo inserisco nella lista ordinata e lo elimino dal DAG, ottenendo una nuova DAG con un nuovo nodo sorgente
- Ripeto fino ad esaurire i nodi del grafo

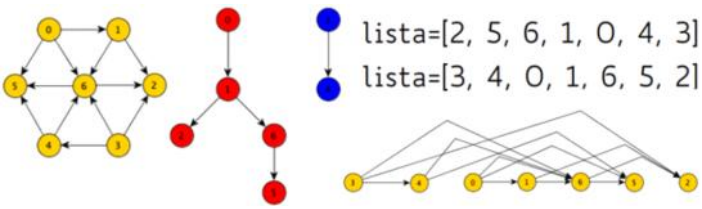
```python
def SortTopDAG(G):
    n = len(G)
    gradoEntrata = [0]*n
    # inizializziamo il vettore dei gradi entranti
    for x in range(n):
        for y in G[x]: # per ogni nodo adiacente
        # incrementa il num. di archi in entrata
            gradoEntrata[y] += 1
    # prende i nodi che hanno 0 archi in entrata
    sorgenti = [ i for i in range(n) if gradoEntrata[i] == 0 ]
    Sorted = []
    while sorgenti: # finche non finisce i nodi sorgente
        x = sorgenti.pop() # prende l'ultimo nodo sorgente
        Sorted.append(x) # aggiunge quel nodo alla lista ordinata
        for v in G[x]: # per ogni nodo adiacente al sorgente
            gradoEntrata[v] -= 1 # rimuove il suo arco a quel nodo
            if gradoEntrata[v] == 0:
                # se il nodo ora non ha più archi in entrata
                sorgenti.append(v) # allora è un nuovo sorgente
    if len(Sorted) == n: return Sorted # se ha tutti i nodi
    return [] # altrimenti non si può ordinare
```

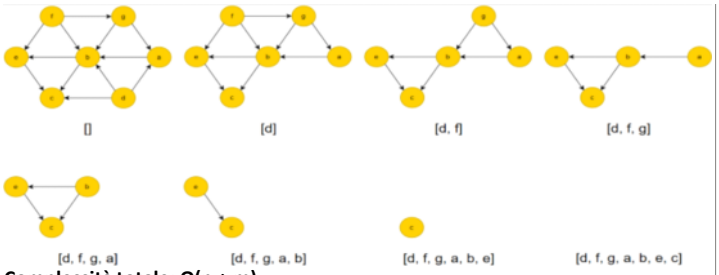


Tramite DFS su DAG

- Eseguo una visita **DFS** a partire dal nodo 0
- Inserisco ogni nodo nella lista **dopo** aver visitato i suoi adiacenti
- L'ordinamento finale è la lista invertita

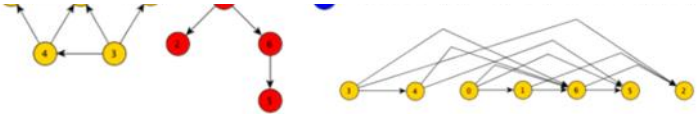
```python
def SortTopDFS(G):
    def DFS_Sort(G, x, visitati, Sorted):
        visitati[x] = 1 # segniamo il nodo come visitato
        for v in G[x]:
            if not visitati[v]: # se non è stato visitato
                DFS_Sort(G, v, visitati, Sorted) # continua il DFS
        Sorted.append(x) # aggiungiamo il nodo alla lista
    ####
    n = len(G)
    visitati = [0]*n
    Sorted = []
    for x in range(n): # per ogni nodo
        if not visitati[x]: # non visitato
            DFS_Sort(G, x, visitati, Sorted) # esegue il DFS
    Sorted.reverse() # ritorniamo il reverse della lista
    return Sorted
```





Complessità totale: O(n + m)

- Inizializza vettore gradi entranti: O(n + m)
- Inizializza insieme delle sorgenti: Θ(n)
- Costo ciclo while: O(n + m) → iter while = O(n), iter for = O(m)



Complessità totale: O(n + m) (costo DFS)

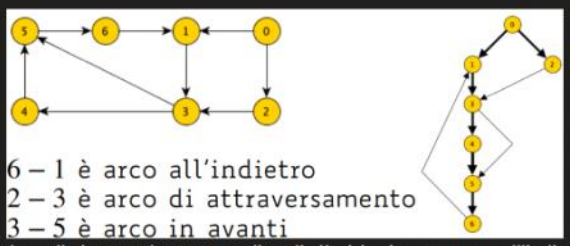
Ricerca Cicli (Slide Grafi 5)

Nella visita DFS abbiamo **quattro tipi di archi**:

- **Archi del DFS** (archi usati nel DFS per la visita)
- **Archi in avanti** (archi **da antenato a discendente**)
- **Archi di attraversamento** (archi che vanno a nodi già visitati)
- **Archi all'indietro** (archi **da discendente a antenato**)

Solo la presenza di archi all'indietro testimonia la presenza di un ciclo.

```python
def CicloInd(G):
    def DFS_ciclo2(G, x, visitati):
        visitati[x] = 1
        for y in G[x]:
            if visitati[y] == 1:
                # nodo in "elaborazione" -> ciclo
                return True
            elif visitati[y] == 0:
                # nodo non visitato -> continua DFS
                if DFS_ciclo2(G, y, visitati):
                    return True
    visitati[x] = 2 # nodo completamente esplorato
    return False
    ####
    # 0: non visitato, 1: in elaborazione, 2: completato
    visitati = [0]*len(G)
    for x in range(len(G)):
        if visitati[x] == 0:
            # avvia DFS solo sui nodi non ancora visitati
            if DFS_ciclo2(G, x, visitati):
                return True
    return False
```



Quindi devo distinguere la scoperta di nodi già visitati con un arco all'indietro per poter determinare la presenza di un ciclo e terminare la ricorsione.

Possiamo impostare un valore per ogni nodo visitato:

- Vale **0** se **non è ancora** stato **visitato**
- Vale **1** se è stato **visitato ma** la **visita ricorsiva non è terminata** su quel ramo
- Vale **2** se è stato **visitato e la visita ricorsiva è finita**.

Scopro un ciclo solo se trovo un arco diretto verso un nodo in **"elaborazione"** (val. 1) (visitato ma la visita non è ancora finita)

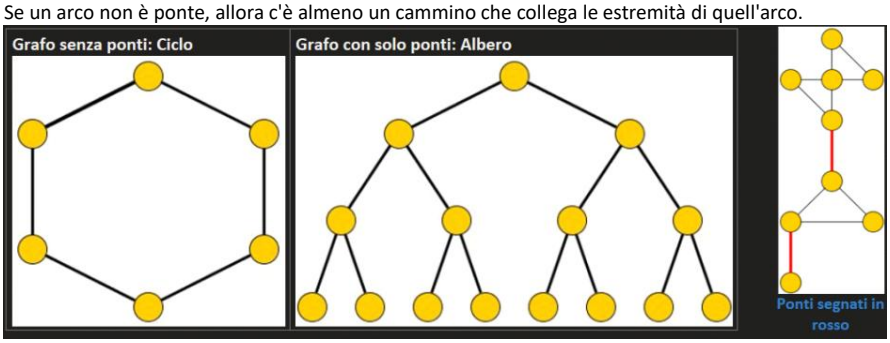
Per sapere se un grafo contiene un ciclo, devo visitarlo tutto, quindi dovrò eseguire la DFS su tutti i nodi non ancora visitati.

Costo tot: O(n + m)

Ponti (Slide Grafi 6)

Un **ponte** è un **arco critico** la cui **eliminazione disconnette il grafo**, creando due componenti separate, facendo **perdere** al grafo la **proprietà di connessione**.

Se un arco non è ponte, allora c'è almeno un cammino che collega le estremità di quell'arco.



Algoritmo Ricerca Ponti

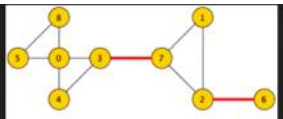
Viene usata una visita DFS modificata:

- I ponti vengono cercati tra gli **n - 1 archi dell'albero DFS**. Infatti un arco non presente nell'albero non può essere ponte, quindi non è critico
- Gli archi dell'albero che non sono ponti sono **"coperti"** dagli altri archi del grafo che non sono stati attraversati dalla DFS (gli archi all'indietro)

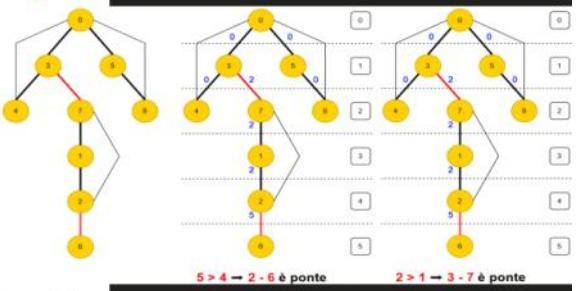
Controlliamo gli archi padre-figlio (x, y) dell'albero per vedere se sono ponti:

- Ogni nodo y (figlio):
  1. Calcola la sua altezza nell'albero
  2. Calcola e restituisce al padre x, l'altezza minima che riesce a raggiungere con archi che partono nel suo sottoalbero (anche archi all'indietro)
- Il nodo x (padre), confronta la sua altezza con l'altezza minima restituita dal figlio (min_alt)
  - Se min_alt > altezza x (il nodo y arriva solo piu in basso di x), l'arco x - y è l'unico collegamento → **x - y è ponte**
  - Se min_alt ≤ altezza x (il nodo y raggiunge o arriva più in alto di x), c'è un collegamento alternativo che collega y ad un antenato di x → **x - y non è ponte**

```python
def TrovaPonti(G):
    def DFS_ponti(G, x, padre, altezza, ponti):
        # impostiamo l'altezza del nodo corrente
        if padre != -1:
            altezza[x] = altezza[padre] + 1
        else:
            altezza[x] = 0
        # inizializzo l'altezza minima raggiungibile dal sottoalbero di x
        min_raggiungibile = altezza[x]
        for v in G[x]: # per ogni nodo adiacente di x
            if altezza[v] == -1: # se non è stato ancora visitato
                min_alt = DFS_ponti(G, v, x, altezza, ponti)
                # cerco l'altezza minima che riesce a raggiungere il sottoalbero di v
                if min_alt > altezza[x]:
                    # se l'altezza di x è minore di quella restituita da v
                    ponti.append((x, v)) # allora (x, v) è un ponte
                min_raggiungibile = min(min_raggiungibile, min_alt)
                # prendo il minimo tra l'altezza di x e quella restituita da v
            elif v != padre:
                # se è stato visitato e (x, v) è un arco all'indietro
                # prendo il minimo tra l'altezza di x e quella di v
                min_raggiungibile = min(min_raggiungibile, altezza[v])
        return min_raggiungibile
    ####
    n = len(G)
    altezza = [-1]*n
    ponti = []
    # inizio la DFS dal nodo 0
    DFS_ponti(G, 0, -1, altezza, ponti)
    return ponti
```



```
G=[
[3,4,5,8],
[2,7],
[1,6,7],
[0,4,7],
[0,3],
[0,8],
[2],
[1,2,3],
[0,5]
]
```



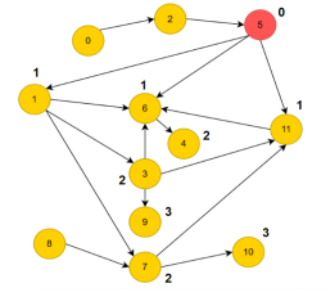

```
>>> TrovaPonti(G)
[(2, 6), (3, 7)]
```

La complessità della procedura è $O(n+m)$

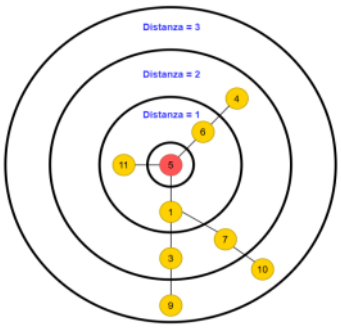
BFS (visita in ampiezza) e Distanze (Slide Grafi 8 (?))

La **BFS** visita prima i nodi adiacenti e poi passa ai livelli successivi (gli adiacenti degli adiacenti). È molto utile per calcolare la **distanza minima** tra due nodi.
È necessario mantenere in una **coda** i nodi visitati i cui adiacenti non sono ancora stati esaminati completamente.
I passaggi base del BFS a partire dal nodo x sono:

- Inizializziamo la coda col nodo di partenza x
- Finchè la coda non viene svuotata (non ci sono più nodi da visitare)
  - Estraggo il primo nodo della coda
  - Tutti i nodi adiacenti non visitati del nodo estratto vengono visitati e messi in coda



$D =$ | $+\infty$ | 1 | $+\infty$ | 2 | 2 | 0 | 1 | 2 | $+\infty$ | 2 | 2 | 1 |



L'implementazione con un array semplice porterebbe il costo a $O(n^2)$ poiché l'estrazione dell'elemento ha costo O(n) (deve spostare ogni elemento indietro di 1)
Abbiamo due implementazioni alternative che riducono il costo a O(n + m)

Implementazione con Cancellazioni Logiche

Invece di estrarre effettivamente l'elemento, usiamo un puntatore per indicare l'inizio della coda nella lista

```python
def BFSLogica(G, x):
    visitati = [0]*len(G)
    visitati[x] = 1
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non superiamo la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if visitati[y] == 0: # se non è stato visitato
                visitati[y] = 1 # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return visitati
```

Implementazione con Deque

Usiamo una **deque, una coda doppiamente puntata**, con costo O(1) per l'inserimento/cancellazione da entrambi i lati.

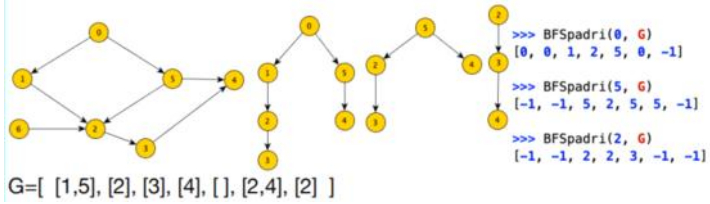
```python
def BFSDeque(G, x):
    visitati = [0]*len(G)
    visitati[x] = 1
    from collections import deque
    coda = deque([x]) # deque inizializzata col nodo di partenza
    while coda: # finche non si svuota la coda
        u = coda.popleft() # copiamo il primo nodo della coda
        for y in G[u]: # per ogni adiacente al nodo
            if visitati[y] == 0: # se non è stato visitato
                visitati[y] = 1 # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return visitati
```

Possiamo modificare il codice per ottenere sia il **vettore dei padri** che il **vettore delle distanze** (per la distanza di un nodo si assegna la distanza del padre + 1).

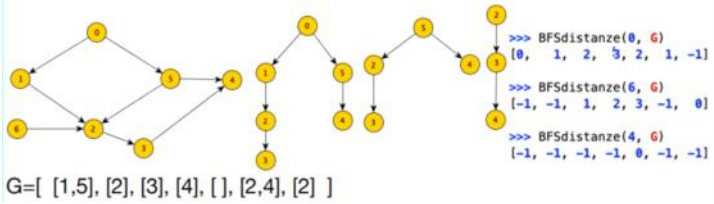
Vettore dei Padri

Vettore delle Distanze

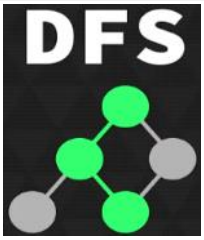
```python
def BFSPadri(G, x):
    P = [-1]*len(G)
    P[x] = x # il padre della radice è la radice stessa
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non si svuota la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if P[y] == -1: # se non è stato visitato
                P[y] = u # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return P
```



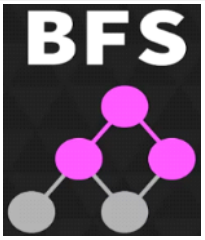

```python
def BFSDistanze(G, x):
    D = [-1]*len(G)
    D[x] = 0 # la distanza dalla radice è 0
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non si svuota la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if D[y] == -1: # se non è stato visitato
                D[y] = D[u] + 1 # viene visitato incrementando la sua distanza
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return D
```



Differenza DFS e BFS



- Esplora un ramo in **profondità** prima esplorare altri percorsi
- Utile per **trovare cicli, algoritmi su alberi, esplorazione di grafi e backtracking**
- Richiede uno **Stack** (esplicito o tramite ricorsione)



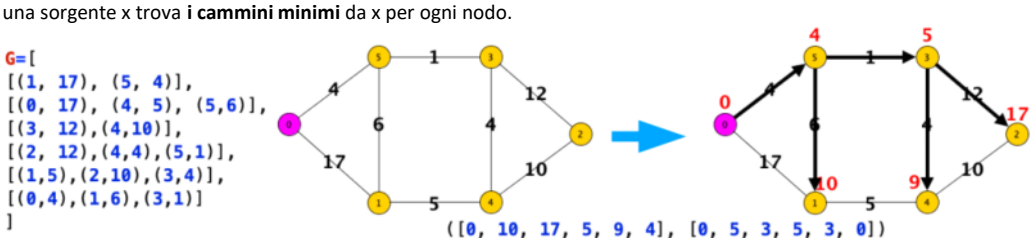
- Esplora i nodi in ordine di distanza dalla radice.
- Utile per **le distanze minime in grafi non pesati o per visitare nodi in livelli**
- Richieda una **coda**

Grafi Pesati

I **grafi pesati** sono grafi in cui gli archi hanno un **peso numerico** (intero o reale, anche negativo).
Per rappresentare il peso c di (x, y), nella lista di adiacenza di x ci sarà la coppia (y, c)

Algoritmo di Dijkstra

È un algoritmo utilizzato per trovare i **cammini minimi** su **grafi pesati**
Funziona sia con pesi **interi** che **reali ma non** con pesi **negativi**, e se è un grafo senza pesi allora si comporta come una visita BFS
Dato un grafo pesato e una sorgente x trova **i cammini minimi** da x per ogni nodo.



I cammini minimi (in grassetto) e le distanze (in rosso)

Passaggi Principali

- Inizializzare il costo dei nodi a ∞ (tranne s che è 0)
- Estrarre il nodo x con la **distanza minima** e lo rendiamo **definitivo**
- Per ogni nodo y adiacente di x ancora **non definitivo** (non si è trovato il percorso migliore), calcolare la distanza attuale da s a y e se è inferiore a quella attualmente memorizzata per y, allora aggiornare la distanza per y.
- Continuiamo finche rendiamo tutti i nodi definitivi

Possiamo implementare questo algoritmo con due strutture dati diverse

Tramite Lista

Per ogni nodo x memorizziamo una **tupla(definitivo, costo, origine):**

- **Definitivo:** flag che vale **1** se il costo per x è **definitivo**, 0 altrimenti
- **Costo**: costo **minimo corrente** noto per raggiungere x da s
- **Origine**: **predecessore di x** lungo il cammino minimo

Inizialmente la Lista è inizializzata a:

$$lista[x] = \begin{cases} (1, 0, s) & se\ x = s \\ (0, costo, s) & se\ (x, costo) \in G[s] \\ (0, +\infty, -1) & altrimenti \end{cases}$$

Iterazioni principali:

1. **Selezione**: scorre la lista per cercare il nodo x **non definitivo** col costo min.
2. **Terminazione**: se il costo min. non si è trovato, allora tutti i nodi sono definitivi e si ferma l'algoritmo
3. **Marcatura**: il nodo x trovato viene impostato come **definitivo**
4. **Aggiornamento**: per ogni vicino y **non definitivo** di x, se **$costo_x$ + peso(x, y) è inferiore** al costo corrente per y, aggiorniamo il nuovo costo di y.

Tramite Heap Minimo

Manteniamo un **heap minimo** contenente una **tupla(costo, x, y)**:

- **x:** nodo già inserito nell'albero dei cammini minimi
- **y:** nodo candidato ad essere aggiunto
- **Costo:** costo per raggiungere y passando per x

Iterazioni principali:

1. **Estrazione:** estrae la tupla col **costo minimo** dall'heap (O(log(n))
2. **Aggiornamento Heap:** quando x viene aggiunto all'albero, per ogni adiacente y, inseriamo nell'Heap la tupla **(costo aggiornato, x, y)** (O(log(n))
3. **Gestione duplicazioni:** se esistono più entry per lo stesso y, viene solo estratta la tupla col costo minimo e le successive le ignoriamo

Complessità:

Nell'Heap ci possono essere O(m) elem. → costo inserimento/estrazione sarà O(log(m)) = O(log(n^2) = O(2*log(n)) = O(log(n))

- L'inizializzazione dei vettori D e P ha costo Θ(n)

Complessità:

- Il costo delle istr. prima del while è Θ(n)
- Il while è eseguito **n-1 volte**
  - Il **primo for** viene iterato **n volte → Θ(n)**
  - Il **secondo for** viene eseguito al massimo **n volte** → O(n)

Costo complessivo: Θ(n) + (n-1)*O(n) = **O(n²)**

Implementazione ottima per grafi densi dove m = O(n²)

```python
def DijkstraLista(G, s):
    n = len(G)
    Lista = [ (0, float("inf"), -1) ]*n # inizializziamo la lista
    # 0 = nessun nodo definitivo, costo = infinito, origine = -1 (sconosciuta)
    Lista[s] = (1, 0, s) # il sorgente è il primo nodo definitivo
    for y, costo_arco in G[s]: # la lista di adiacenza ha (nodo_adiacente, costo_arco)
        # aggiorno gli adiacenti vicini a s, pero non sono ancora definitivi
        Lista[y] = (0, costo_arco, s)
    while True:
        # cerco il nodo non definitivo con costo minimo
        minimo, x = float("inf"), -1
        for i in range(n):
            if Lista[i][0] == 0 and Lista[i][1] < minimo:
                minimo, x = Lista[i][1], i
        if minimo == float("inf"):
            # se tutti i nodi sono definitivi
            break # esce
        # altrimenti rendi definitivo il nodo x
        definitivo, costo_x, origine = Lista[x]
        Lista[x] = (1, costo_x, origine)
        # aggiorna eventualmente i vicini di x non definitivi
        for y, costo_arco in G[x]:
            if Lista[y][0] == 0 and minimo + costo_arco < Lista[y][1]:
                # se non è definitivo e il cammino è migliore passando per x
                Lista[y] = (0, minimo+costo_arco, x) # aggiorna il nodo
    # Estrae da Lista i vettori delle distanze e dei padri
    D = [ costo for _, costo, _ in Lista]
    P = [ origine for _, _, origine in Lista]
    return D, P
```

- L'inserimento dei vicini di s nell'Heap ha costo O(n*log(n))
- Nel while vengono estratti O(m) elem. e ogni estrazione ha costo O(log(n))
- Il for viene iterato O(m) volte e ogni inserimento ha costo O(log(n))

Costo complessivo: O(n*log(n)) + O(m*log(n)) + O(m*log(n)) = **O((n+m)*log(n))**

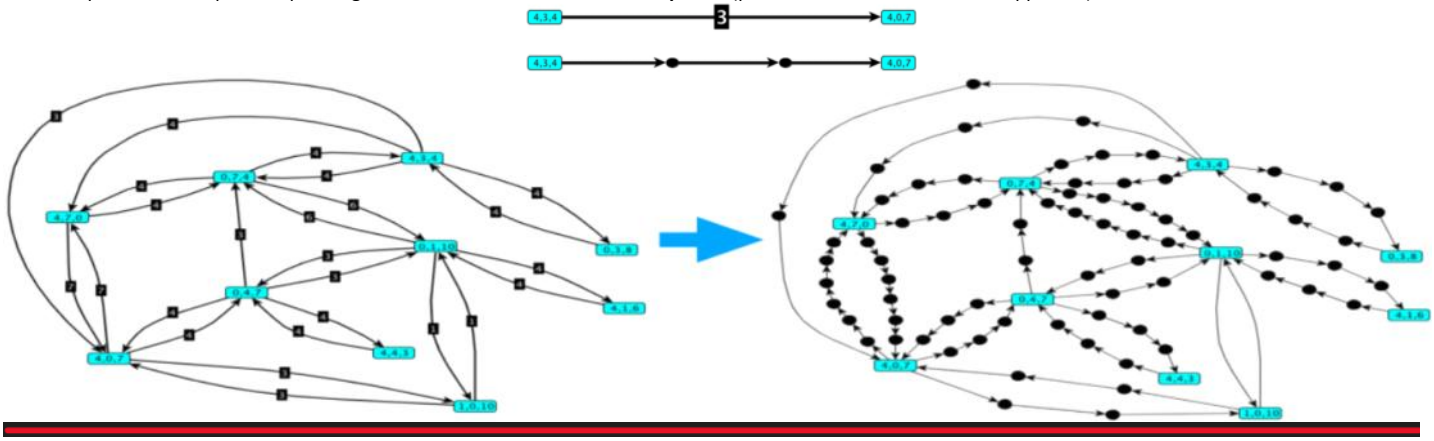
Implementazione ottima per grafi sparsi → costo O(n*log(n))

È da evitare per grafi densi → costo O(n²*log(n))

```python
def DijkstraHeap(G, s):
    n = len(G)
    D = [float("inf")]*n # distanze inizialmente infinite
    P = [-1] * n # predecessori inizialmente sconosciuti
    D[s], P[s] = 0, s # inizializziamo la sorgente
    from heapq import heappush, heappop
    Heap = [] # Heap minimo
    for y, costo_arco in G[s]: # inizializziamo i nodi adiacenti alla sorgente
        heappush(Heap, (costo_arco, s, y))
    while Heap:
        costo, x, y = heappop(Heap) # estraiamo il nodo con distanza minore
        if P[y] == -1: # se non ha predecessore
            P[y] = x # imposta y come figlio di x
            D[y] = costo # imposta la distanza min. di y
            for v, costo_arco in G[y]: # esplora gli adiacenti di y
                if P[v] == -1: # se non è stato ancora aggiunto all'albero lo aggiungiamo
                    heappush(Heap, (D[y] + costo_arco, y, v))
    return D, P # Restituiamo le distanze e i predecessori
```

Ridurre un Grafo Pesato in uno Non Pesato

È possibile ricondurre il problema su un **grafo pesato** in uno con un **grafo non pesato** sostituendo il **peso c** di un arco tra x e y con **c-1 nuovi nodi** tra x e y.
Ora basta usare una visita BFS per trovare i cammini minimi tra archi con lo stesso valore 1.
Questo è possibile solo quando i pesi degli archi sono **interi e relativamente piccoli** (poiché senno si creerebbero troppi nodi)



UNION-FIND

Union-Find è una struttura dati per **gestire insiemi disgiunti** (nei grafi sono i componenti di G). Le tre op. fondamentali sono:

1. **Crea(S)**: crea la struttura dati Union-Find sull'insieme S, dove ciascun elem. è un **insieme separato**
2. **Find(x, C)**: restituisce il nome del componente a cui appartiene l'elem. x
3. **Union(x, y, C)**: fonde la componente x con quella y e restituisce il nome della nuova componente

Si può usare **Find(x, C) == Find(y, C)** per vedere se due nodi sono nello stesso componente.
Come primo approccio all'assegnazione di un **nome dell'insieme**, si può dare il nome dell'**elemento massimo dell'insieme**.

Ci sono tre tipi di implementazioni ma quella più bilanciata in termini di costi è l'**implementazione con alberi bilanciati.**
Essa migliora l'implementazione col vettore dei padri mantenendo gli **alberi bilanciati** per diminuire il costo del Find.
Viene associato ai nodi radice (quello che indica l'insieme) anche il **num. di elem. che contiene il componente**.
Ogni elem. è una coppia **(x, num)** dove **x** è il **nome** del componente e **num.** è il num. di nodi nel componente (l'albero radicato in x).
Nell'Union, scegliamo il componente col **num. maggiore di elem.** come **nuova radice**, cosi che almeno per la **metà dei nodi** delle due componenti la lunghezza del cammino **non aumenta.** In questo modo garantiamo la seguente proprietà:

Se un **insieme** ha **altezza h** allora l'insieme **contiene** almeno **2^h elementi**

Quindi l'altezza delle componenti **non supererà mai O(log(n))**. Cosi possiamo ridurre il costo di Find **da O(n) a O(log(n))**

```python
def Crea(G):
    C=[ (i,1) for i in range(len(G))]
    return C
```

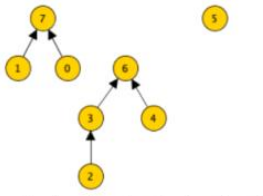
Crea( ) costo $\Theta(n)$

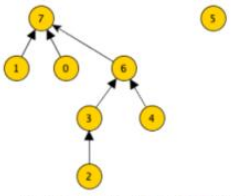
```python
def Find(u, C):
    while u != C[u]:
        u = C[u]
    return u
```

Find( ) costo $O(\log n)$

```python
def Union (a, b, C):
    tota, totb = C[a][1], C[b][1]
    if tota >= totb:
        C[a]=(a, tota + totb)
        C[b]=(a, totb)
    else:
        C[a]=(b, tota)
        C[b]=(b, tota + totb)
```

Union( ) costo $O(1)$





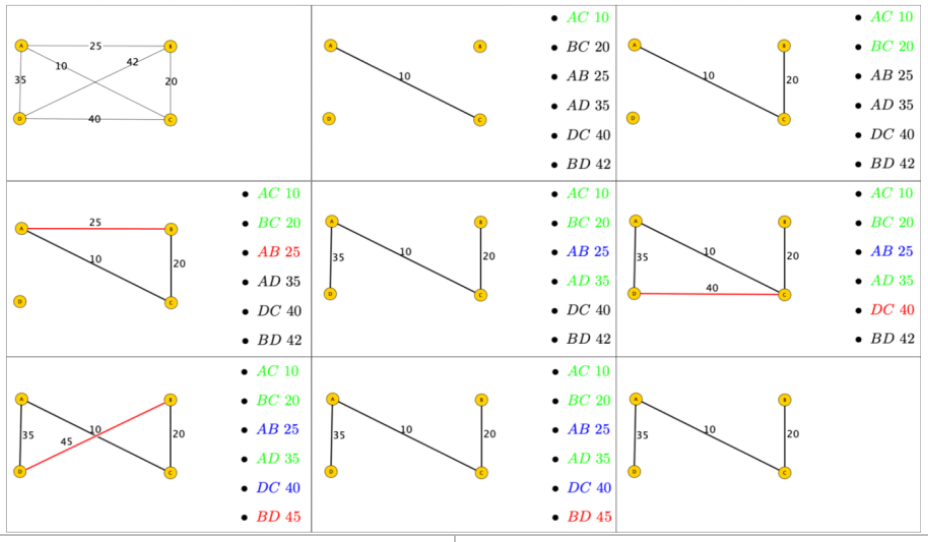
Minimi Alberi di Copertura

Un **albero minimo di copertura** è un albero che **"copre"** l'intero grafo pesato e in cui la somma dei suoi costi degli archi sia **minima.**
In questo albero non ci possono **mai essere cicli** ed esiste **un solo albero minimo di copertura** per ogni grafo pesato.

Algoritmo di Kruskal

L'**algoritmo di Kruskal** è un algoritmo **greedy** ottimo per la creazione di un albero minimo di copertura, dato un grafo connesso e pesato:

- Crea un **grafo T** che contiene **tutti i nodi** di G **ma nessun arco**
- Considera uno dopo l'altro gli archi di G in **ordine** di **costo crescente**
- Se l'arco forma un **ciclo** in T allora **non prenderlo**, altrimenti inseriscilo in T
- Al termine restituisci T



Implementazione Semplice

Idee:

- Con un **pre-processing ordino** gli archi in E cosicchè scorrendo la lista ottengo di volta in volta l'arco di costo min. in tempo O(1)
- Verifico che l'arco (x, y) **non formi ciclo** in T controllando se y è raggiungibile da x in T

```
def connessi(x, y, T):
    # esegue una visita in T da x e ritorna True se c'è un cammino fino a y
    def DFS_conn(x, y, T, visitati):
        visitati[x] = 1 # visita il nodo corrente
        for v in T[x]: # per ogni nodo adiacente
            if v == y: # se abbiamo trovato il nodo da ricercare
                return True # ritorniamo True
            if not visitati[v]: # se non è stato visitato
                if DFS_conn(v, y, T, visitati): # continuiamo la visita
                    return True
        return False # se non l'abbiamo trovato ritorniamo False
    ###
    visitati = [0]*len(T) # inizializzo la lista visitati
    return DFS_conn(x, y, T, visitati)

def Kruskal(G):
    # creo la lista con gli archi pesati di G
    E = [(c, x, y) for x in range(len(G)) for y, c in G[x] if x<y]
    E.sort() # ordino gli archi in ordine crescente di peso
    T = [[] for _ in G] # inizializzo la lista vuota dell'albero di copertura
    for c, x, y in E: # per ogni nodo in E
        if not connessi(x, y, T): # se non c'è un cammino tra i due nodi x-y
            T[x].append(y) # al nodo x connetto y
            T[y].append(x) # e al nodo y connetto x
    return T # ritorno l'albero minimo di copertura
```

Costo:

- L'ordinamento esterno al for costa **O(m*log(m)) = O(m*log(n^2)) = O(m*log(n))**
- Il for viene iterato **m volte**:
  - Il controllo che l'arco (x, y) non crei ciclo in T con la procedura **connessi(x, y, T)** richiede il costo di una visita di un grafo aciclico quindi **O(n)**
- Il for richiede tempo **O(m*n)**

Costo totale: O(m*n)

Implementazione con Union-Find

Invece di pagare O(n) ad ogni iter. del for per controllare se esiste un ciclo, possiamo usare la struttura dati **Union-Find** che permette di testare se due nodi appartengono o meno alla **stessa componente connessa**. Usiamo le due op:

- **UNION(x, y, C)** fonde due componenti connesse **x e y in C** in tempo **O(1)**
- **FIND(x, C)** trova in C la componente connessa in cui si trova **x** in tempo **O(log(n))**

```
def Kruskal_Union_Find(G):
    # creo la lista con gli archi pesati di G
    E = [(c, x, y) for x in range(len(G)) for y, c in G[x] if x<y]
    E.sort() # ordino gli archi in ordine crescente di peso
    T = [[] for _ in G] # inizializzo la lista vuota dell'albero di copertura
    C = Crea(T)
    for c, x, y in E:
        cx = Find(x, C) # prendo i rappresentati dei componenti di x e y
        cy = Find(y, C)
        if cx != cy: # se stanno in componenti distinti, x-y non crea ciclo
            T[x].append(y) # al nodo x connetto y
            T[y].append(x) # e al nodo y connetto x
            Union(cx, cy, C) # unisco i componenti di x e y
    return T # ritorno l'albero minimo di copertura
def Crea(G): # crea l'insieme dei componenti
    C = [ (i, 1) for i in range(len(G))]
    # i è l'indice nell'insieme dei componenti
    # 1 è la "dimensione" del set
    return C
def Find(x, C): # restituisce il set a cui appartiene
    while x != C[x]: # finchè non troviamo il nodo
        x = C[x] # aggiorna x al suo padre
    return x
def Union(x, y, C): # unisce i set
    totx, toty = C[x][1], C[y][1] # prendo le grandezze dei componenti
    if totx >= toty: # se il componente di x è più grande o uguale a quello di y
        C[x] = (x, totx+toty) # incremento la dimensione di x con quella di y
        C[y] = (x, toty) # aggiorno il rappresentante di y a x
    else: # altrimenti se il componente di y è più grande
        C[y] = (y, totx+toty) # incremento la dimensione di y con quella di x
        C[x] = (y, toty) # aggiorno il rappresentante di x a y
```

Costo:

- L'ordinamento costa **O(m*log(n))**
- Il for viene iterato **m volte:**
  - L'estrazione dell'arco (x, y) di costo minimo da E richiede **Θ(1)**
  - Eseguire **FIND** costa **O(log(n))**
  - Eseguire **UNION** costa **Θ(1)** (nel for viene eseguito n-1 volte)

- Il for richiede tempo **O(m*log(n))**

Costo totale: O(m*log(n))

Cicli Negativi

Un **ciclo negativo** è un ciclo diretto in cui la **somma dei pesi** degli archi **è negativa**.
Il problema di esso è che **ripassando** più volte nel ciclo, possiamo abbassare continuamente il costo del cammino.
Proprietà: se G **non contiene cicli negativi**, allora per ogni nodo esiste un **cammino di costo minimo** che attraversa al più **n-1 archi.**
Se un cammino avesse più di n-1 archi, allora almeno un nodo verrebbe ripetuto, formando un ciclo. Poiché G non ha cicli negativi, rimuovere eventuali cicli dal cammino **non aumenta** il suo costo complessivo. Quindi esiste sempre un cammino ottimale di lunghezza n-1.

Quindi il costo minimo può essere calcolato considerando solo **cammini di lunghezza n-1 e non oltre**.

Per il costo min. dei cammini tra i nodi del grafo non possiamo usare **Dijkstra** poiché non funziona con i pesi negativi. Bisogna usare **l'algoritmo di Bellman-Ford**

Algoritmo di Bellman-Ford

Esso calcola i cammini minimi su un grafo diretto pesato con anche pesi negativi.
In base alla proprietà vista in precedenza, possiamo ridurre i cammini minimi di lunghezza **max n-1**
Usiamo una tabella di dimensione **nxn**, dove:

T[i][j] = costo cammino min. da s a j di lunghezza al più i (archi attraversati)

Quindi la sol. del problema saranno i val che troviamo **nell'ultima riga:**

T[n-1][0], T[n-1][1], …, T[n-1][n-1]

Quindi il costo min per andare **da s a t sarà T[n-1][t]**
I valori delle celle sono riempite in base a questa regola:

$$T[i][j] = \begin{cases} 0 & se\ j = s\ (colonna\ sorgente) \\ +\infty & se\ i = 0\ (prima\ riga) \\ \min\left(T[i-1][j], \min_{(x,j)\in E}\left(T[i-1][x] + costo(x,j)\right)\right) & altrimenti \end{cases}$$

Dove il terzo valore è definito dalla considerazione dell'aggiornamento del costo nella tabella, poiché si può effettuare in base a due casi:

1. Il val. T[i][j] **rimane uguale** a T[i-1][j] (il precedente) se il cammino migliore era già stato trovato con max i-1 archi
2. Il val. T[i][j] viene aggiornato come il **minimo tra** T[i-1][j] e il costo per raggiungere j passando per un nodo x con un arco (x, j). In questo caso prendiamo il minimo su tutti nodi x che hanno un arco verso j

Cosi per ogni nodo, si sceglie tra mantenere il costo precedente o migliorarlo aggiungendo un nuovo arco.
NOTA: per rendere l'implementazione più efficiente, si usa il **grafo trasposto GT**, per accedere rapidamente agli archi entranti in un nodo, evitando di scansionare l'intero grafo a ogni aggiornamento, poiché in GT[j] avremmo i nodi **x raggiungibili da j** t.c. in G **esiste un arco da x a j**.

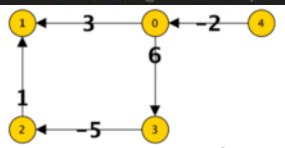
```python
def TraspostoPesato(G):
    n = len(G)
    GT = [ [] for _ in G ] # inizializza la lista vuota
    for i in range(n): # per ogni nodo
        for j, costo in G[i]: # per ogni adiacente e il costo dell'arco (i->j)
            GT[j].append( (i, costo) ) # aggiungiamo l'arco invertito (j->i) in GT
            # ad ogni adiacente inseriamo il nodo padre e il costo dell'arco
    return GT

def CostoCammini(G, s):
    n = len(G)
    T = [[float("inf")]*n for _ in range(n)] # inizializza tabella nxn a infinito
    T[0][s] = 0 # il costo per raggiungere s con 0 archi è 0
    GT = TraspostoPesato(G) # crea il trasposto pesato di G
    for i in range(1, n): # per ogni lunghezza da 1 a n-1
        for j in range(n): # per ogni nodo j
            T[i][j] = T[i-1][j] # copiamo il costo della riga precedente (1° caso)
            if j != s: # per ogni nodo diverso dalla sorgente
                for x, costo in GT[j]: # per ogni arco entrante in j (x->j in GT)
                    T[i][j] = min(T[i][j], T[i-1][x]+costo) # aggiorna il costo minimo
                    # min tra quello attuale e cammino che arriva da x più costo arco
    return T[n-1] # ritorna i costi minimi per cammini di lunghezza al più n-1
```

Costo:

- L'inizializzazione di T richiede tempo $\Theta(n^2)$
- La costruzione del **grafo trasposto** GT richiede tempo O(n + m)
- Per i tre for annidati è ovvio il limite superiore $O(n^3)$, però un analisi più attenta può dare un limite più stretto:
  - Il primo for ovviamente richiede tempo O(n)
  - Gli ultimi due for hanno costo tot O(n+m)
    Quindi il tempo richiesto per scorrere le liste di adiacenza del grafo GT con lunghezza tot. m

Costo complessivo: $n*O(n+m) = O(n^2 + n*m)$

- Se il grafo è **sparso** (m = O(n)), il costo diventa $O(n^2)$
- Se il grafo è **denso** (m = $O(n^2)$), il costo diventa $O(n^3)$




```
G=[
    [(1,3),(3,6)],
    [],
    [(1,1)],
    [(2,-5)],
    [(0,-2)]
]
```

i \ j	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	3	∞	6	∞
2	0	3	1	6	∞
3	0	2	1	6	∞
4	0	2	1	6	∞

```
>>> costo_cammini(G,0)
[0, 2, 1, 6, inf]
```

i \ j	0	1	2	3	4
0	∞	∞	∞	∞	0
1	-2	∞	∞	∞	0
2	-2	1	∞	4	0
3	-2	1	-1	4	0
4	-2	0	-1	4	0

```
>>> costo_cammini(G,4)
[-2, 0, -1, 4, 0]
```

Trovare Cammini con Vettore dei Padri

Oltre a T possiamo calcolare l'albero P dei **cammini minimi.** Per ogni nodo j manteniamo il suo **predecessore**, cioè il nodo x che precede j.
Il predecessore in P[j] va aggiornato ogni volta che si trova un cammino migliore per T[i][j].

```
def CostoCamminiPadre(G, s):
    n = len(G)
    T = [[float("inf")]*n for _ in range(n)] # inizializza tabella nxn a infinito
    T[0][s] = 0 # il costo per raggiungere s con 0 archi è 0
    GT = TraspostoPesato(G) # crea il trasposto pesato di G
    P = [-1]*n # inizializziamo la lista dei padri
    P[s] = s # il padre della radice è la radice stessa
    for i in range(1, n): # per ogni lunghezza da 1 a n-1
        for j in range(n): # per ogni nodo j
            T[i][j] = T[i-1][j] # copiamo il costo della riga precedente (1° caso)
            if j != s: # per ogni nodo diverso dalla sorgente
                for x, costo in GT[j]: # per ogni arco entrante in j (x->j in GT)
                    T[i][j] = min(T[i][j], T[i-1][x]+costo) # aggiorna il costo minimo
                    # min tra quello attuale e cammino che arriva da x più costo arco
                    P[j] = x # aggiorniamo il padre del nodo
    return T[n-1], P # ritorna i costi minimi per cammini di lunghezza al più n-1
```

In questa implementazione avremmo:

- **T[n-1][j] ≠ +∞** indica che **j è raggiungibile a partire da s**
  Allora P[j] avrà il nodo che **precede** j nel cammino minimo
- **T[n-1][j] = +∞** indica che **j non è raggiungibile a partire da s**
  Allora P[j] conterrà **-1**

Correttezze

mercoledì 26 marzo 2025 12:28

Grafi Colorati

Algoritmo di bi-colorazione

Questo algoritmo prova che un **grafo senza cicli dispari** può essere sempre colorato:

- Colora il nodo 0 col colore 0
- Effettua una **DFS** a partire da 0 e nella visita, a ciascun nodo x incontrato si assegna **0 o 1**.
  Si sceglie il colore da assegnare in modo che sia diverso dal colore dato al nodo padre che ti ha portato a x

Prova di Correttezza bi-colorazione

Siano x e y due nodi adiacenti in G, consideriamo i due possibili casi e vediamo che
in entrambi i casi i due nodi al termine avranno colori opposti.

1) **L'arco (x, y) viene attraversato durante la visita**. In questo caso i due nodi avranno colori distinti.
2) **L'arco (x, y) NON viene attraversato durante la visita**:
   - Sia x il nodo visitato prima, **esiste un cammino** in G che da **x porta a y** (quello seguito dalla visita)
   - Questo cammino si chiude a formare un ciclo con l'arco (y, x).
   - Il **ciclo** è di lunghezza **pari** per ipotesi, quindi il **cammino** è di lunghezza **dispari.**
   - Poiché sul cammino i colori si **alternano**, il primo nodo (x) e l'ultimo (y) avranno **colori diversi**

Ponti

Algoritmo TrovaPonti con DFS

- I ponti vanno ricercati tra gli **n - 1 archi dell'albero DFS.** Infatti un arco non presente nell'albero DFS non può essere ponte (se lo elimino, gli archi dell'albero continuano a garantire la connessione)
- Gli archi che non sono ponti sono **"coperti"** dagli altri archi del grafo che non sono stati attraversati nel DFS (archi all'indietro)

Proprietà: Sia (u, v) un arco dell'albero DFS con **u padre di v**. L'arco **u - v** è un **ponte se e solo se** non ci sono archi tra **u o antenati di u** e i nodi del sottoalbero radicato in **v.**

Prova:

- **Per assurdo**: sia **x - y** un arco tra un **antenato di u** e un **discendente di v**. Dopo l'eliminazione di **u - v** i nodi dell'albero restano comunque **connessi grazie all'arco x - y**
- L'eliminazione di u - v disconnette i nodi dell'albero radicato in v dal resto del grafo. In questo caso tutti gli archi che non appartengono all'albero e che partono dal sottoalbero di v punterebbero solo verso v o suoi discendenti, perché non esistono altri archi che li connettono a nodi "più in alto" di v nell'albero.

Durante la visita da un nodo u al nodo v, il nodo u non sa ancora se nel sottoalbero di v ci sia un arco che torna sugli antenati di v.
Quindi dopo la visita del sottoalbero di v, v comunica a u il nodo più in alto raggiungibile (con archi di ritorno o altri percorsi).
Tramite questo valore possiamo vedere che, se il **sottoalbero v non può arrivare ad un antenato di u**, allora u - v **è un ponte.**

BFS

Correttezza:

Alla fine dell'esecuzione, **visitati[y]** contiene **1** solo se il **nodo y è raggiungibile da x**

- **Se y è raggiungibile a partire da x allora visitati[y] = 1**. In questo caso c'è un **cammino P** da x a y.
  Supponiamo **per assurdo** che al termine **visitati[y] = 0**. Sia:
  - **b** il primo nodo che incontro nel cammino P con V[b] = 0
  - **a** il suo predecessore

  (questi due nodi esistono perche il cammino comincia con x e V[x] = 1 e termina con y e V[y] = 0)
  Un nodo che diventa 1 finisce in coda, quindi a è finito in coda, ma l'algoritmo si ferma quando la coda si svuota quindi c'è stato un momento in cui a è stato estratto dalla coda, e i suoi vicini sono stati controllati e, se non erano ancora stati visitati, visitati (e poi inseriti in coda) quindi anche b è stato controllato e se non ancora visitato sarebbe stato visitato e posto a 1
- **Se y non è raggiungibile a partire da x allora visitati[y] = 0**. Supponiamo **per assurdo** che al termine **visitati[y] = 1**
  Visitati[y] è 1 solo se y viene inserito in coda ma questo accade perché risulta adiacente ad un nodo che viene estratto, andando all'indietro questo processo deve terminare perche ciascun nodo viene inserito al più una volta nella coda, quindi si crea un cammino da x a y, il che è assurdo avendo supposto che y non è raggiungibile da x.

Non è difficile dimostrare che ogni vertice raggiungibile da x finisce nell'albero BFS. Vale in più:

Proprietà: la distanza minima di un vertice y da x nel grafo G equivale alla **profondità di y** nell'albero BFS

Prova: per **induzione** sulla distanza d di y da x.

- **Caso base:** È ovviamente vero per d = 0 in quanto l'unico vertice a distanza 0 x è x stesso che è a profondita 0 nell'albero.
- **Ipotesi induttiva:** Supponiamo sia vero per tutti i vertici con distanza al più **d - 1** e consideriamo quindi un vertice **u a distanza d**.
  Sia P un **cammino minimo** da x a u e sia **v il predecessore di u** in questo cammino.
  Per **ipotesi induttiva** v è a profondità **d - 1** nell'albero BFS. Se u è stato inserito nell'albero grazie a v allora si troverà in

profondità d, assumiamo quindi che v sia stato inserito nell'albero grazie ad un nodo u ≠ v.
La profondità di u **non** può essere **inferiore a d - 1** altrimenti avremmo trovato un cammino che va da x a v (tramite u) di lunghezza inferiore a d. D'altra parte non può essere la profondità di u maggiore di d - 1 perche il nodo v sarebbe stato visitato prima di u e y sarebbe stato inserito grazie al nodo v. Deve quindi aversi che la profondità di u è d - 1 e quella di v è d.

Riassunto prova:

- **Caso base:** Se d = 0, il solo nodo a distanza 0 da x è x stesso
- **Passo induttivo**: Supponiam oche i nodi a distanza d - 1 da x abbiano profondità d - 1 nell'albero BFS. Consideriamo un nodo **u** a distanza **d**.
  - Esiste un cammino minimo da x a u, e chiamiamo **v** il predecessore di **u** in questo cammino
  - Per ipotesi induttiva, **v** ha profondità d - 1 nell'albero
    - Se **u** viene scoperto grazie a **v**, allora viene inserito a **profondità d** nell'albero
    - Se **u** viene scoperto grazie ad un altro nodo **w ≠ v**, la sua profondità non potrebbe essere inferiore a **d**, perché altrimenti esisterebbe un **cammino più corto da x a v**, contraddicendo la **minimalità di d**
  - Inoltre **u** non può avere **profondità maggiore di d,** perché **v** sarebbe stato scoperto prima e avrebbe inserito **u** al liv. corretto

Grazie a questa proprietà, i cammini prodotti usando l'albero BFS sono **di lunghezza minima**, ecco perche l'albero BFS è detto **albero dei cammini minimi**

Grafi Pesati

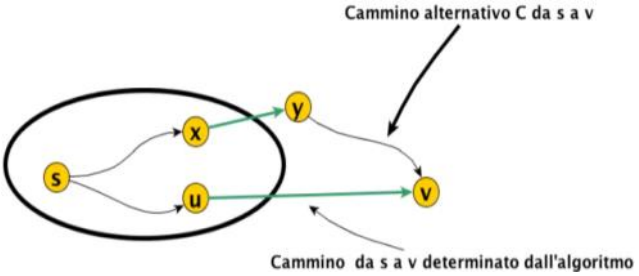
Correttezza Algoritmo di Dijkstra

Correttezza per grafi con pesi positivi: ad ogni iter. del while viene assegnata una nuova distanza ad un nodo.
Per induzione sul num. di iter. mostreremo che la distanza assegnata è quella minima.

- **Caso base**: al passo 0 si assegna distanza zero alla sorgente e con pesi pos. non ci può essere una distanza inferiore
- Sia T_i l'albero dei cammini minimi costruiti fino al passo i > 0 e (u, v) l'arco aggiunto all'albero al passo i + 1.
  Vedremo che D[v] è la distanza min. di v da s. Basterà mostrare che il costo di un eventuale cammino alternativo è ≥ D[v].
  - Sia C un qualsiasi cammino da s a v alternativo a quello nell'albero, e (x, y) il primo arco che incontriamo percorrendo C all'indietro t.c x è nell'albero T_i e y no (tale arco esiste perché s è in T_i mentre v no).
  - **Per ipotesi induttiva** costo(C) ≥ Dist(x) + peso(x, y) (**questa affermazione è vera perche i pesi del grafo sono positivi)**
  - L'algoritmo ha preferito estendere l'albero T_i con l'arco (u, v) anziché l'arco (x, y) e in base alla regola con cui l'algoritmo sceglie l'arco con cui estendere l'albero deve quindi aversi D[x] + p(x, y) ≥ D[u] + p(u, v)
  - da cui segue: costo(C) ≥ D[x] + peso(x, y) ≥ D[u] + p(u, v) = D[v].
  - **Quindi il cammino alternativo ha costo superiore a D[v]**

Riassunto prova:
Caso base: la sorgente ha distanza 0, che è corretto perché nessun cammino può avere distanza negativa (pesi positivi)
Passo induttivo:

- Al passo i + 1, si aggiunge l'arco (u, v) nell'albero dei cammini minimi
- Si dimostra che D[v] assegnata è la min. possibile
- Se esiste un **cammino alternativo** C più conveniente per arrivare a v, ci sarebbe un arco (x, y) che attraversa per la prima volta un nodo fuori dall'albero.
- Dalla propr. dell'algoritmo, l'arco scelto (u, v) è sempre quello col costo minore tra i disponibili, quindi:
  D[x] + p(x, y) ≥ D[u] + p(u, v) = D[v]
- Da ciò segue che qualunque cammino alternativo ha costo D[v], quindi l'algoritmo assegna sempre il val. minimo



Dimostrazioni

martedì 18 marzo 2025 13:16

Metodi generali per la dimostrazione delle proprietà nei grafi

Contraddizione

Assumi che la **proprietà** da dimostrare sia **falsa** e mostra come questa assunzione porta ad un **risultato assurdo** o una **violazione** di una **proprietà nota**.
Se è cosi allora la proprietà deve essere per forza vera.
Esempio: Dimostrare che se tutti i vertici di G hanno grado almeno 2, allora c'è almeno un ciclo

- Supponiamo per assurdo che il grafo sia **privo di cicli**, quindi è un **albero**
- Però in un albero ci deve essere almeno un **nodo con grado 1** (foglia), contraddicendo l'ipotesi

Induzione

Usata soprattutto per dimostrare una proprietà per **tutti i grafi con un certo num. di nodi**

- **Caso Base**: Mostra che la proprietà **vale** per il caso **più piccolo** (es n = 1 o n = 2)
- **Passo induttivo**: Assumi che valga per un grafo con n nodi e dimostra che deve valere per un grafo con n + 1 nodi

Esempio: Dimostrare che la somma dei gradi dei nodi di un grafo è sempre **pari**

- **Caso Base:** In un grafo con **un nodo e zero archi**, la somma dei gradi è **0,** che è **pari**
- **Passo induttivo**: Aggiungere un **nuovo nodo** e un **nuovo arco** incrementa la somma dei gradi di **due** (1 per ogni nodo collegato) che è **pari**

Principio dei Cassetti

Se distribuisci **n elem. in m categorie** e n > m, almeno una categoria conterrà più di un elem.
Esempio: Dimostrare che se G ha almeno due nodi, allora ci sono due nodi con lo stesso grado

- Se ci sono n > 2 nodi, i gradi possibili vanno da 1 a n-1, quindi n-1 valori.
- Per il principio dei cassetti, almeno due nodi avranno lo stesso grado

Esempio e Controesempio

Per dimostrare che una proprietà è **vera**, costruisci un **esempio** che la **soddisfa**
Per dimostrare che una proprietà è **falsa**, costruisci un **controesempio** che la **viola**
Esempio: Se il grado di ogni vertice è esattamente due, si può affermare che G è un ciclo

- **Si:** Un grafo con ogni nodo di grado 2 deve formare uno o più cicli
- **No**: se è connesso, è un ciclo, se non è connesso è una collezione di cicli disgiunti

Applicazione dei metodi

- Contraddizione: esistenza cicli
- Induzione: proprietà numeriche
- Teoremi noti: grafi planari o proprietà di grafi completi
- Principio dei cassetti: distribuzioni di proprietà tra i nodi

Riassunto Ristretto

martedì 1 aprile 2025 13:24

Grafi

Num. max di archi: $O(n^2)$

- **Diretto:** archi hanno direzione (u→v ≠ v→u).
  - Proprietà di **forte connettività**: è **fortemente connesso** se per ogni coppia di vertici (x, y) è collegata da un cammino (quindi x→y e y→x)
    Un compo
- **Indiretto**: archi non hanno direzione (u→v = v→u)
  - Proprietà di **connettività:** è **connesso** se ogni coppia di vertici è collegata da un cammino (non ci sono sottografi separati da altri)
- **Sparso:** poche connessioni tra vertici. Num. archi **m = O(n) → m ≤ n**
- **Denso:** quasi ogni coppia di vertici è connessa da un arco. Num. archi **m = O(n²) → m > n**
- **Colorato:** ad ogni vertice viene assegnato un colore. Un grafo è **colorabile** se si possono colorare i nodi t.c i nodi **adiacenti** abbiano **colori distinti**
  Un grafo è **bi-colorabile** se **NON contiene cicli** di lunghezza **dispari** (i colori devono alternarsi)
- **Pesato:** Ad ogni arco si associa un costo (peso) che può essere anche negativo
- **Trasposto:** ha gli stessi nodi di G ma gli **archi** sono in **direzione opposta**. I nodi in G che **portano ad x**, sono i nodi in **GT** che **sono raggiungibili da x** e viceversa
  Molto utile per conoscere i grafi entranti di un nodo, poiché in G saranno quelli entranti (pero non si conoscono) e in GT sono quelli uscenti (che si conoscono)
  Es: se in G i nodi che **portano** al nodo 5 sono [1, 4, 0, 7, 2, 3, 6], allora in GT i nodi **raggiungibili** dal nodo 5 sono [1, 4, 0, 7, 2, 3, 6] (lista di adiacenza)
- **Ciclico:** Un grafo è **ciclico** se esiste un **sottogruppo connesso** con ogni vertice di grado ≥ 2
  - Se nel grafo **indiretto** ogni vertice ha **grado ≥ 2**, allora il grafo è **ciclico**
  - Se nel grafo **diretto** ogni vertice ha **almeno un arco uscente**, allora il grafo è **ciclico**
- **Componente: sottoinsieme massimale** (non si possono aggiungere altri vertici) di vertici t.c. per ogni coppia di vertici del sottoinsieme, esiste un cammino
  - **Connesse**: Ogni coppia di vertici del sottoinsieme è collegata da un cammino
  - **Fortemente connesse:** Ogni coppia di vertici del sottoinsieme ha cammini in entrambe le direzioni

Un **nodo** può essere:

- **Pozzo:** in un grafo diretto è un nodo **senza archi uscenti**
  - **Pozzo universale:** pozzo verso cui tutti gli altri nodi hanno un arco

Vettore dei Padri

Permette di ricostruire il **cammino** di visita da un nodo alla radice dell'albero

DFS	BFS
• Visita in **profondità**, segue il percorso più lungo prima di ritornare indietro • Utile per **trovare cicli, cammini, esplorazione grafi, ord. topologico**	• Esplora per **livelli**, partendo dal nodo sorgente • Trova i **cammini minimi** in grafi non pesati o visitare per livelli • Richiede una **coda** per i nodi adiacenti ancora non visitati

Alberi

Grafo **connesso** e **senza cicli** (DAG). Num. archi **m = n-1**

- Albero DFS: Generato dall'esecuzione di una DFS. La radice è il nodo di partenza
- Albero BFS: Generato dall'esecuzione di una BFS. La radice è il nodo di partenza. Il liv. di un nodo rappresenta la distanza (num. archi attraversati) dalla radice

Ordinamento Topologico

L'**ordinamento topologico** è una disposizione dei nodi di un **grafo diretto aciclico** (DAG) in cui ogni arco vada **da sinistra verso destra**, rispettando le dipendenze.
Un grafo diretto può avere da **0 a n!** ordinamenti topologici. Se si ha un solo arco per n nodi, possiamo avere $\frac{n!}{2}$ ordinamenti diversi (formula generica: $\frac{n!}{m!}$)
L'ordinamento si può effettuare solo su un DAG poiché se c'è un **ciclo** c'è una **dipendenza circolare**, che rende impossibile trovare l'ordine di esecuzione.

Ponti

È un **arco critico** poiché la rimozione **disconnette il grafo**, creando due componenti separate.

- **Grafo senza ponte**: Ciclo. Se ogni vertice ha grado ≥ 2 non ci sono ponti (forse)
- **Grafo con solo ponti**: Albero

Union-Find

Struttura dati per gestire componenti di G. Nell'implementazione con **alberi bilanciati** manteniamo un vettore dei padri e il vettore dei componenti e abbiamo che il **nome dei componenti** (la radice) è l'**elem. massimo del componente**. Oltre al nome del componente si associa anche il **num. di elem. che contiene**: (x, num)
Nell'op. Union il componente con meno elem. viene collegata a quella con piu elem., cosi a metà dei nodi il cammino non aumenta, cosi garantiamo che l'altezza delle componenti **non superi O(log(n))**
Le tre op sono:

- **Crea(S):** crea la struttura dati C in base all'insieme S, dove ciascun elem. è un **insieme separato** (inizialmente C[i] = i). Costo **Θ(n)**
- **Find(x, C)**: ritorna il nome del componente a cui appartiene x. Costo **O(log(n))**
- **Union(x, y, C)**: fonde la componente x con quella y. Sceglie come componente principale quella col maggior num. di elem. Costo **O(1)**

Minimi Alberi di Copertura

Un **albero min. di copertura** è un albero che **"copre"** l'intero grafo pesato e in cui la somma dei costi degli archi è **minima**.
Non ci sono **mai cicli** e si ha un solo albero min. di copertura per grafo
Algoritmo di Kruskal: Ordina gli archi per peso crescente e usa union-find per evitare i cicli (se due componenti stanno nello stesso componente allora c'è ciclo)

Cicli Negativi

Ciclo in cui la somma dei pesi è **negativa**. Se esiste un ciclo negativo da s a j allora **non esiste** un **cammino minimo** (si può ridurre indefinitamente il costo)
Algoritmo di Bellman-Ford: Calcola i cammini minimi su grafo diretto pesato senza cicli negativi
Poiché un cammino ottimale ha al più n-1 archi, si itera n-1 volte e se dopo n-1 iter. si può migliorare un cammino, allora c'è un ciclo negativo.

Progr. Dinamica

lunedì 19 maggio 2025 17:28

Raga onestamente per la programmazione dinamica il metodo migliore è esercitarsi e capire come funzionano le tabelle. Purtroppo non c'è un metodo generale per farli

Logaritmi

martedì 11 marzo 2025 22:27

$\log_b(x) = y \Leftrightarrow b^y = x$

$\mathrm{a}^{\mathrm{b}} = 2^{b*\log_2(a)}$

$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$ (utile per convertire tra basi diverse, es: $\log_2(x) = \frac{ln(x)}{ln(2)}$)

$\log_b(xy) = \log_b(x) + \log_b(y)$

$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$

$\log_b\left(\sqrt[n]{x}\right) = \frac{1}{n} * \log_b(x)$

$\log_b(x^y) = y * \log_b(x)$

$2^{\log_b(n)} = n^{\log_b(2)}$

$$\log_a(b) = \begin{cases} > 1 & se\ a < b \\ 1 & se\ a = b \\ < 1 & se\ a > b \mid base < \end{cases}$$

Sommatorie Importanti

giovedì 24 ottobre 2024 18:38

$$\sum_{i=0}^{n} i = \theta(n^2) = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} i^c = \theta(n^{c+1})$$

$$\sum_{i=0}^{n} 2^i = \theta(2^n)$$

$$\sum_{i=0}^{n} c^i = \begin{cases} \frac{c^{n+1}-1}{c-1} & c > 1 \\ 1 & c \leq 1 \end{cases}$$

$$\sum_{i=0}^{n} i2^i = \theta(n2^n)$$

$$\sum_{i=0}^{n} ic^i = \theta(nc^n)$$

$$\sum_{i=0}^{n} log^c i = \theta(nlog^c n)$$

$$\sum_{i=0}^{n} \frac{1}{i} = \theta(logn)$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 = \Theta(2^n)$$

$$\sum_{i=0}^{n} i * 2^i = (n-1) * 2^{n+1} + 2 = \Theta(n * 2^n)$$

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

$$\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$$

quando n cresce $\frac{1}{2^n} \to 0$, dunque

S(n) = 2 = Θ(1)

$$\sum_{i=0}^{n} \log(i) = \Theta(n * \log(n))$$

$$\sum_{i=0}^{n} n - 1 = \sum_{j=0}^{n} j = \frac{n(n+1)}{2}$$

$$\sum_{i=a}^{n} c = c * n - a + 1$$

Efficienza

martedì 11 marzo 2025 13:35

Un algoritmo si dice **efficiente** se la sua complessità è **O(n^c)** per una costance c. Cioè è di ordine **polinomiale** nella dim. **n dell'input.**
Un algoritmo è **inefficiente** se la sua complessità è di ordine **superpolinomiale:**

- **Esponenziale:** funz. di ordine $\Theta(c^n) = 2^{\Theta(n)}$
  ($c^n = 2^{log(c^n)} = 2^{n*\log(c)}$ e poiché log(c) è una costante, l'esponente $n * log(c)$ è una funz. lineare in n, quindi appartiente a Θ(n) e possiamo scrivere c^n =2^(Θ(n)) siccome log(c) viene ignorato da Θ)
- **Super-esponenziale:** funz. cresce **più velocemente di un esponenziale** (Es: $2^{\Theta(n^2)}$ oppure $2^{\Theta(n*log(n))}$)
- **Sub-esponenziale:** funz. cresce **più lentamente di un esponenziale**, cioè **$2^{o(n)}$** (Es: $n^{\Theta(log(n))} = 2^{\Theta(log^2(n))}$ $oppure$ 2^(Θ(n^c)) dove c è una costante < 1)

Come caso di studio si può pensare al **test di primalità** (vede se un num. n è primo)

- L'algoritmo **banale** (dato n, cerca un divisore tra i num. da 2 e n-1) è **esponenziale** poiché ha complessita O(n) (nota che la dim. di input è log(n))
  - La dim. di input è misurata in **bit**, per rappresentare un num. n servono circa **L = $\log_2$(n)** bit ⇒ **n = 2^L**.
  - Se l'algoritmo esegue O(n) operazioni, il tempo di esec. diventa **O(2^L) = O($2^{\log(n)}$)** quindi è esponenziale rispetto alla dim. dell'input
- Un algoritmo **efficiente** dovrebbe avere complessità **O($\log^c$(n))** per una qualche costante **c** mentre questo algoritmo ha complessità **O($2^{\log(n)}$)**
- Nella ricerca dell'eventuale divisore **fermarsi alla radice** velocizza l'algoritmo ma non cambia la complessità asintotica che resta esponenziale
  Per un num. composto, se non esiste divisore ≤ √(n) allora non esiste nessun divisore intermedio
  $O(\sqrt{n}) = O\left(\sqrt{2^{log(n)}}\right) = O\left(2^{log(\sqrt{n})}\right) = O\left(2^{\frac{1}{2}log(n)}\right) = 2^{\Theta(\log(n))}$ (poichè a*log(n) ∈ Θ(log(n) dove a > 0)
- Gia dagli anni 70 si conoscevano algoritmi **sub-exponenziali** ad es. di complessità $log(n)^{\Theta(log(log(n)))}$ e al**goritmi probabi**listici (che possono sbagliare con una probabilità che si può rendere piccola a piacere) con complessità poli$nomiale$ $O(\log^3(n))$
- Solo nel 2004 si è trovato un algoritmo **polinomiale deterministico** di tempo $O(\log^{12}(n))$ velocizzato poi a $O(\log^3(n))$ anche se con costanti moltiplicative molto alte che non lo rendono competitivo con i ben noti algoritmi probabilistici

Vediamo anche il problema dell'ordinazione di una lista di n interi

- Un banale algoritmo **superpolinomiale** fa:
  - Genera **tutte** le possibili **permutazioni** dei val. da ordinare (complessità O(n!)) e controlla se la permutazione soddisfa l'ordinamento (O(n))
  - Complessità complessiva è O(n*n!)
  - Estendiamo il fattoriale e mettiamo in evidenza due categorie:
    - $\underbrace{1 \cdot 2 \cdot 3 \cdots \frac{n}{2}}_{\text{numeri} \le n/2} \quad \underbrace{\left(\frac{n}{2}+1\right) \cdots (n-2) \cdot (n-1) \cdot n}_{\text{numeri} > n/2}$
    - Quindi metà dei fattori (quelli compresi tra $\frac{n}{2} + 1$ e n) sono tutti ≥ a $\frac{n}{2}$. Quindi:
    - $n! = 1 * 2 * \cdots * n > \frac{n}{2} * \frac{n}{2} * \cdots * \frac{n}{2} > \left(\frac{n}{2}\right)^{\frac{n}{2}}$
    - Applicando la proprietà $a^b = 2^{b*\log(a)}$ otteniamo:
    - $n! > \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{\frac{n}{2}*\log(\frac{n}{2})} = 2^{\frac{n}{2}*(\log(n)-1)} = 2^{\Theta(n*\log(n))}$. Poichè $n! > 2^{\Theta(n*\log(n)}$ allora $n! = 2^{\Omega(n*\log(n))}$
    - Ogni fattore i ≤ n è sempre ≤ n quindi:
    - $n! < n * n * \cdots * n = n^n$
    - Possiamo riscrivere n^n in forma esponenziale usando la proprietà $a^b = 2^{b*\log(a))}$, quindi:
    - $n! < 2^{n*\log(n)}$ $ovvero$ $n! = 2^{O(n*\log(n))}$
  - Poiché abbiamo sia $n! = 2^{\Omega(n*\log(n))}$ che $n! = 2^{O(n*\log(n))}$ allora $n! = 2^{\Theta(n*\log(n))}$
  - Vediamo che la funzione fattoriale è **super-esponenziale** ($n! = 2^{\Theta(n*\log(n))}$) e di conseguenza l'algoritmo è **intrattabile**

- Possiamo considerare un altro algoritmo: cerca il min. tra gli n val. e lo mette in prima posizione; poi ricerca il min. tra gli n-1 val e lo mette in seconda e cosi via
  -
    ```
    def Selection_Sort(A)
    for i in range(len(A)-1):        (n - 1) Θ(1) + Θ(1)
      m = i                          Θ(1)
      for j in range(i+1,len(A)):      (n - i) Θ(1) + Θ(1)
          if (A[j] < A[m])               Θ(1)
            m = j                        Θ(1)
        A[m],A[i]=A[i],A[m]          Θ(1)
    ```
  - La complessità è **Θ(n^2)** quindi è un algoritmo **polinomiale**
- Se uso la struttura dati opportuna, **l'heap**, la cui costruzione costa Θ(n) e l'estrazione del min. costa O(log(n)) ottengo un algoritmo (**heapsort**) che ordina gli elem. in tempo O(n*log(n))
  -
    ```
    def Heapsort (A):
      Build_heap(A)                      O(n)
      for x in reversed(range(1,len(A))): (n-1) iteraz.
        A[0],A[x]=A[x],A[0]                Θ(1)
        Heapify(A, 0, x)                   O(log n)
    ```

Limitazione inferiore

mercoledì 12 marzo 2025 13:24

Non bisogna **confondere** la **complessità dell'algoritmo** con la **complessità del problema.**

- Un alg. di complessità **O(g(n))** produce una **limitazione superiore** alla complessità del problema
- Se si dimostra che **qualunque algoritmo** per quel problema ha complessità **Ω(f(n))** si è stabilità una **limitazione inferiore** alla complessità del problema
- Se **f(n) = g(n)** allora l'algoritmo ha complessità **Θ(f(n)** (o Θ(g(n)) ed è detto **ottimo** poiché la sua complessità in ordine di grandezza risultà la **migliore possibile**

Calcolare limitazioni **inferiori significative** ai problemi **non** è un **compito semplice**
Vi sono pochi modi generali per la dimostrazione di una limitazione inferiore, vediamone due semplici:

- **Dimensione dei dati:**
  Se un problema ha **n dati** in input e richiede di **esaminarli tutti**, allora una limitazione inferiore della complessità è **Ω(n)** poiché li scorrera tutti quanti.
  - **Es:** per la ricerca del max in una lista, l'algoritmo banale che scorre la lista è **ottimo**.

  Per **l'ordinamento** non è sufficiente questo ragionamento
- **Eventi contabili**:
  Se un problema richiede che un **certo evento** sia **ripetuto m volte**, allora una limitazione inferiore della complessità è **Ω(m)**.
  - **Es:** per gli alg. di ordinamento basati sul **confronto**, si può dimostrare che se **n** sono gli interi da ordinare allora bisogna **effettuare** almeno **n*log(n) confronti**, quindi **Ω(n*log(n))** è il limite inferiore alla complessità e di conseguenza l'algoritmo **heapsort è ottimo**

Se aggiungo **vincoli** al problema dell'ordinamento, il lower bound **O(n*log(n)) non vale più**, poiché posso usare algoritmi che non si basano sul confronto

- **Es:** se gli elem. nella lista sono di due tipi, ho un vincolo sull'intervallo dei numeri da ordinare (2 in questo caso) quindi uso il **countingSort** che ha costo **O(n)**
- Più in generale l'alg. si applica quando i val. da ordinare sono **interi positivi** e ha costo **O(k+n)** dove k è il **max** tra gli elem. da ordinare, quindi non è **polinomiale**

Intrattabile

Un problema si dice **intrattabile** se **non può avere algoritmi efficienti.**
Esempi ovvi sono:

- Stampare tutte le stringhe bin. di lunghezza **n**. Ha un limite inferiore $2^{\Omega(n)}$
- Stampare tutte le permutazioni di n elem. Ha un limite inferiore $2^{\Omega(n*\log(n))}$

Tesi di Church-Turing

Perché per algoritmo **efficiente** si intende uno con costo **O(n^c)** per un intero c e non più semplicemente **O(n^2) o O(n^3)?**

- **Tesi di Church-Turing**: I modelli di calcolo realistici sono **equivalenti** dal punto di vista **computazionale**.
  Se qualcosa è **non calcolabile** (es. sulla macchina di Turing) allora **resterà non calcolabile su qualunque macchina** di calcolo automatico
- **Tesi di Church-Turing estesa**: i modelli di calcolo realistici sono **tra loro polinomialmente correlati.**
  Il concetto di **trattabilità** è dunque indipendente dalla macchina

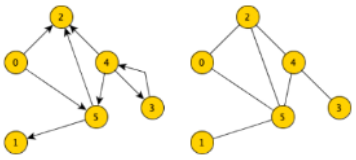
Grafi

mercoledì 12 marzo 2025 14:30

Un **grafo G** è una coppia **(V, E)** dove **V** è un insieme di **nodi o vertici** ed **E** è un insieme di **archi** che collegano i nodi.
Grafo: **G(V, E), |V| = n (nodi), |E| = m (archi)**

Un grafo può essere:

- **Diretto:** gli archi hanno una **direzione**, quindi il **collegamento** tra nodi è in **un solo verso**
  - Ogni nodo può avere un arco verso ognuno degli altri n-1 nodi → num. max archi: n(n-1) = **O(n²)**
- **Indiretto:** gli archi **non hanno direzione**, quindi un **collegamento** tra nodi è **bidirezionale**
  - Ogni arco è condiviso, quindi conta una sola volta invece che due → num. max archi: $\frac{n(n-1)}{2}$ = **O(n²)**
- **Sparso:** un grafo con pochi archi rispetto al num. massimo possibile, ovvero con **pochi collegamenti tra nodi**
  - Il num. di archi cresce al massimo come **O(n)**, ovvero non cresce troppo velocemente rispetto al num. di nodi
- **Denso**: un grafo con un num. di archi vicino al massimo possibile, quindi con **molti collegamenti tra i nodi**
  - Il num. di archi cresce almeno come **Ω(n²)**, ovvero ha un num. di archi vicino al massimo possibile.
  - Esempi di grafi densi:
    - **Completo:** ha il num. max di archi, cioè m = Θ(n²)
    - **Torneo (grafo diretto):** tra ogni coppia di nodi c'è esattamente un arco, quindi m = Θ(n²)



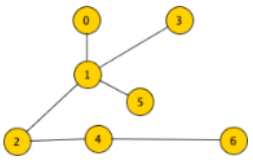
Definizioni Base

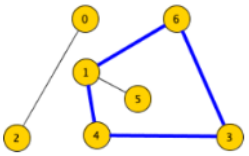
- Nodi Adiacenti
  Due nodi collegati da un arco si dicono **adiacenti** (o vicini) e il **grado** di un nodo **deg(x)** è il num. di nodi adiacenti ad esso
- Grafo Connesso
  - Un grafo si dice **connesso** se per qualsiasi coppia di nodi esiste un **cammino** che li **collega** ($\forall v_i, v_j \in V(G)\ \exists$cammino $\mid v_i \rightarrow v_j \vee v_j \rightarrow v_i$)
  - Un grafo si dice **strettamente connesso** se per qualsiasi coppia di nodi esiste un cammino che li collega partendo da entrambi i nodi ($\forall v_i, v_j \in V(G)\ \exists$cammino $\mid v_i \rightarrow v_j \wedge v_j \rightarrow v_i$)
- Passeggiata
  - Una **passeggiata** è una sequenza alternata di archi e nodi: $v_0, e_1, v_1, e_2, v_2, \ldots, v_{k-1}, e_k, v_k$
  - Un **cammino** è una passeggiata in cui non si ripetono i nodi
  - Un grafo ha una **passeggiata euleriana solo se è connesso**, ed esistono al massimo 2 vertici di grado dispari

Albero

L'albero è un **grafo sparso connesso senza cicli**

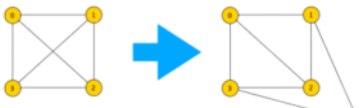
- Ha sempre **m = n - 1 archi**

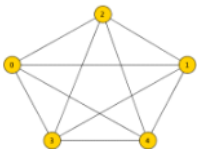

Grafo connesso senza cicli


Grafo sconnesso con ciclo

Grafo Planare

Il grafo **planare** è un grafo che posso disegnae senza che gli archi si intersechino.
Gli alberi sono un sottoinsieme dei grafi planari


Grafo Planare


Grafo NON Planare

L'albero è un grafo Planare

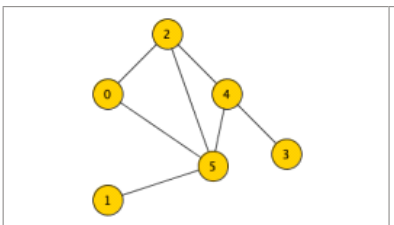
- **Teorema (Eulero)**: **Un grafo planare di n > 2 nodi ha al più 3n - 6 archi**

nodi	$m = \frac{n(n-1)}{2}$ **in g. completi**	$m = 3n - 6$ **max in g. planari**
3	3	3
4	6	6
5	10	9
6	15	12
…	…	…
10	45	24

Rappresentazione Grafo

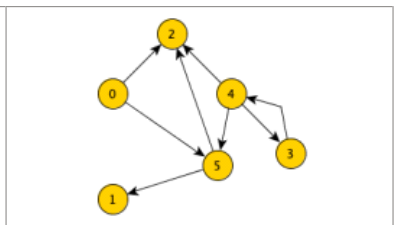
La rappresentazione si può effettuare tramite:

- **Matrici binarie**: è una matrice di dimensioni **n x n** dove M[r][c] = 1 solo se c'è un arco diretto da **r** a **c**
  - **Costo** per controllare se **r è vicino di c**: O(1)
  - **Spazio di archiviazione**: O(n²)
- **Liste di adiacenza**: una lista di liste G
  - G ha tanti elem. quanti sono i nodi del grafo
  - G[x] è una lista contenente i nodi adiacenti al nodo x, cioè quelli raggiunti da archi che partono da x
  - **Costo** per controllare se **x è vicino di y**: O(n)
  - **Spazio di archiviazione:** $O(n + O(\sum_{i=1}^{n} \deg(v_i)) = O(n + m)$



Matrice Binarie

r \ c	0	1	2	3	4	5
0	0	0	1	0	0	1
1	0	0	0	0	0	1
2	1	0	0	0	1	1



Matrice Binarie

r \ c	0	1	2	3	4	5
0	0	0	1	0	0	1
1	0	0	0	0	0	0
2	0	0	0	0	0	0

$$M = \begin{array}{c|cccccc} r \backslash c & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 1 & 0 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Liste di Adiacenza

```
G=[
    [2,5],
    [5],
    [0,4,5],
    [4],
    [2,3,5],
    [0,1,2,4]
]
```

$$M = \begin{array}{c|cccccc} r \backslash c & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 1 & 1 & 0 & 0 & 0 \end{array}$$

Liste di Adiacenza

```
G=[
    [2,5],
    [],
    [],
    [4],
    [2,3,5],
    [0,1]
]
```

Pozzo

In un grafo diretto:

- Un **pozzo** è un nodo senza archi uscenti
- Un **pozzo universale** è un pozzo verso cui tutti gli altri nodi hanno un arco

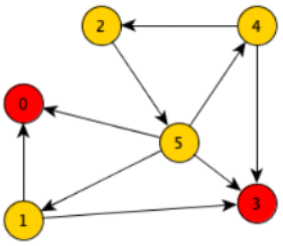
In un grafo diretto possono esserci fino a n pozzi, mentre il pozzo universale, se c'è, è unico

Matrice con pozzo universale nel nodo 2

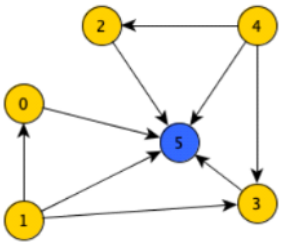
$$\begin{array}{|cccccccc|} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Possiamo vedere che:

$$M[r][c] = \begin{cases} se\ 1\ (arco\ tra\ r\ e\ c), & allora\ r\ non\ è\ pozzo\ (universale) \\ se\ 0\ (no\ arco\ tra\ r\ e\ c), & allora\ c\ non\ è\ pozzo\ universale\ se\ r\ \neq c \end{cases}$$



2 Pozzi (0 e 3)



Unico Pozzo Universale

$$M = \begin{array}{c|cccccc} r \backslash c & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 1 & 0 & 0 \\ 5 & 1 & 1 & 0 & 1 & 1 & 0 \end{array}$$

Rappresentazione con Matrice

$$M = \begin{array}{c|cccccc} r \backslash c & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 0 & 0 & 1 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Rappresentazione con Matrice

Scrivere un algoritmo di tempo O(n²) che verifica se un grafo diretto M ha un pozzo universale

```python
# Algoritmo che verifica se un Grafo diretto M ha un pozzo universale
# test_pozzoU(r, M) controlla se il nodo x è un pozzo universale
def test_pozzoU(x, M):
    # controlla se il nodo non abbia archi uscenti
    # quindi la riga deve essere tutta di 0
    for j in range(len(M)):
        if M[x][j] == 1: # se ha trovato un 1, allora il nodo non è pozzo universale
            return False
    # controlla che tutti gli altri nodi abbiano un arco che indica a x
    # quindi la colonna deve essere tutta di 1 (tranne nella riga del nodo)
    for i in range(len(M)):
        if i!=x and M[i][x] == 0:
            return False # se un nodo i non punta a x allora x non è pozzo universale
    return True

# restituisce True se il grafo M ha un pozzo universale, False altrimenti
def pozzoU2(M):
    # per ogni nodo x nel grafo, verifica se è un pozzo universal
    for x in range(len(M)):
        if test_pozzoU(x, M): # se x è un pozzo universale, restituisce True
            return True
    return False # se nessun nodo è pozzo universale, restituisce False
```

Esegue due for separati che iterano al massimo su tutti i nodi di M, quindi ha costo O(n)
pozzoU2 ha un solo for ma anche esso itera al massimo su tutti i nodi di M, quindi ha costo O(n)
Complessita = O(n) x O(n) = **O(n²)**

Scrivere un algoritmo di tempo Θ(n) che verifica se un grafo diretto M ha un pozzo universale.
Con un semplice test posso eliminare uno dei nodi dai possibili pozzi universali. Dopo n -1 test mi resta un unico nodo da controllare:

```python
# se M[r][c] = 1 -> r non è pozzo (universale)
# se M[r][c] = 0 -> c non è pozzo universale
def pozzoU1(M):
    l = len(M)
    N = [x for x in range(l)]  # prendiamo la lista di nodi
    while len(N) > 1:  # finche non rimane un solo elem
        r = N.pop()  # prendiamo due nodi diversi tra lodo
        c = N.pop()
        if M[r][c] == 1:  # r non è pozzo, quindi lo rimuoviamo
            N.append(c)   # salvando c per il prossimo controllo
        else:             # c non è pozzo universale, quindi lo rimuoviamo
            N.append(r)   # salvando r per il prossimo controllo
    x = N.pop() # prendiamo l'ultimo elemento (il probabile pozzo univers.)
    for j in range(l):
        if M[x][j] == 1:  # se ogni nodo (x compreso) punta ad x
                          # allora esso non è un pozzo universale
            return False
    for i in range(l):
        if i != x and M[i][x] == 0: # se ogni altro nodo non punta ad X
                                    # allora non è un pozzo universale
            return False
    return True # se non ci sono stati problemi allora x è un nodo universale
```

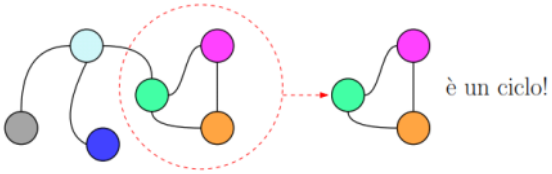
Grafi ciclici

venerdì 14 marzo 2025 18:05

Un grafo è **ciclico** se esiste un **sottogruppo connesso** in cui ogni **vertice** ha grado ≥ 2.
Se nel grafo **tutti i vertici** hanno grado ≥ 2, allora il grafo è **sicuramente ciclico**

$\forall v \in V(G)\ \deg(v) \geq 2 \Rightarrow G$ è ciclico

In un grafo **diretto**, se ogni nodo ha **almeno un arco uscente** allora il grafo è ciclico.



Ricerca di un Ciclo

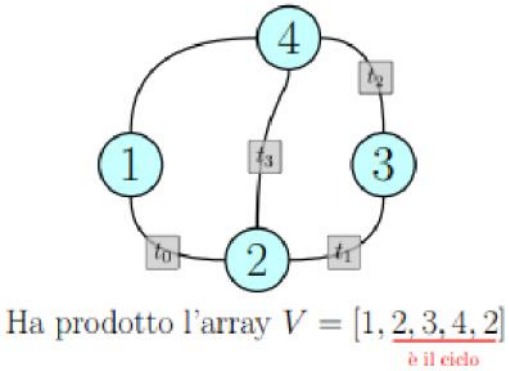
Definiamo un algoritmo con:

- **Input:** un grafo G = (V, E), dove ogni vertice ha grado ≥ 2
- **Output:** un qualsiasi ciclo presente nel grafo, mantendendo costo O(n + m)

Idea di soluzione:

Ogni vertice ha almeno 2 nodi adiacenti, quindi è sempre possibile entrare in un vertice e uscire da un arco diverso da quello di ingresso.
Si parte da un qualsiasi vertice del grafo e si procede selezionando uno qualsiasi dei due nodi adiacenti successivi, almeno uno sarà diverso da quello di ingresso.
Utilizziamo un **vettore** per salvare i **nodi visitati**, e infine, appena troviamo un nodo già salvato in precedenza, eliminiamo uno ad uno gli elem. finche non troviamo l'elemento gia trovato in precedenza. Quindi infine avremmo nella prima e ultima posizione (separati da altri nodi) due nodi uguali, indicando che c'è un ciclo.

```python
def CercaCiclo(G):
    x = V[random] # prendiamo un nodo a caso
    C = [x] # inizializzo il vett. di output
    current = x # inizializzo l'elem. corrente per il ciclo
    next = x.adiacente[random] # prendiamo un nodo adiacente a caso
    while(next not in C):  # finche non trovo un nodo gia visitato
        C.append(next)  # aggiungiamo il prossimo nodo a quelli visitati
        current = next # cambiamo il nodo corrente a quello successivo
        if current.adiacente[0] != C[-2]:
            # se il primo nodo adiacente è diverso dal penultimo della lista
            next = current.adiacente[0] # continuo la ricerca su quel nodo
        else:
            next = current.adiacente[1] # eseguo la ricerca sull'altro nodo successivo
    while C[0] != next:
        # se il primo elem. di C è diverso dall'ultimo nodo trovato che fosse gia in C
        C.pop(0) # rimuovo il primo elemento
    return C
```



Entrambi i cicli eseguo O(n) iter. però, nel primo while, il controllo "next not in C" scorre tutto il vettore, rendendo cosi il costo totale $O(n^2)$, e non O(n+m)

DFS

venerdì 14 marzo 2025 17:10

Visita in Profondità (DFS)

Per controllare se esiste un cammino tra due vertici x e y in un grafo, bisogna prima controllare tutti i nodi che si possono raggiungere partendo da x.
La **DFS (Depth First Search)** è un metodo per **visitare** un grafo in **profondità** partendo da un nodo e spostandosi in un suo vicino casuale non ancora visitato.

- **Versione ricorsiva**: usiamo una **lista** per indicare i **nodi visitati**.
- **Versione iterativa**: usiamo una **pila** per indicare i **nodi visitati**. Per ogni nodo che incontriamo lo salviamo nella pila e ci spostiamo sui vicini non visitati. Se tutti i vicini sono stati già visitati allora **ritorniamo a quelli precedenti** (**back tracking o rollback**), ritornando indietro sui nodi nella pila.

L'output contiene **tutti i nodi raggiungibili** dal nodo di partenza e si ha una **lista visitati** in cui se il nodo x ha val **visitati[x] = 1** → allora il nodo è raggiungibile, 0 altrimenti.

Ricorsiva su Matrice

```python
def DFS_matrix(M, x):
    # visitati[a] = 1 -> il nodo è stato già visitato
    # visitati[a] = 0 -> il nodo deve ancora essere visitato
    ####
    def DFS_rec(M, x, visitati):
        visitati[x] = 1
        # impostiamo il nodo corrente a 1 per dire che è stato visitato
        for i in range(len(M)): # per ogni nodo del grafo
            if M[x][i] == 1 and not visitati[i]:
                # se c'è un arco tra x ed i, e i non è tra i nodi visitati
                DFS_rec(M, i, visitati) # controlliamo anche esso
    ####
    n = len(M)
    visitati = [0]*n # inizializziamo tutti i nodi a 0 (non visitati ancora)
    DFS_rec(M, x, visitati) # iniziamo la ricerca in profondità su x
    return [x for x in range(n) if visitati[x]]
    # ritorniamo i nodi visitati (uguali a 1)
```

- Costo: **O(n) x Θ(n) = O(n²)**
- Costo spazio: O(n) (lista visitati)

Ricorsiva su Liste di Adiacenza

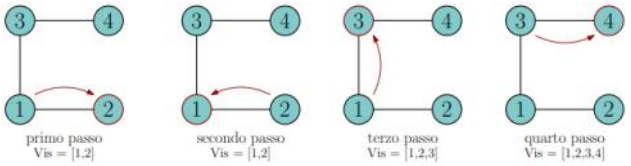
```python
def DFS_list(G, x):

    # visitati[a] = 1 -> il nodo è stato già visitato
    # visitati[a] = 0 -> il nodo deve ancora essere visitato
    ####
    def DFS_rec(G, x, visitati):
        visitati[x] = 1
        # impostiamo il nodo corrente a 1 per dire che è stato visitato
        for v in G[x]: # per ogni nodo adiacente a x
            if not visitati[v]: # se non è stato già visitato
                DFS_rec(G, v, visitati) # controlliamo anche esso
    ####
    n = len(G)
    visitati = [0]*n # inizializziamo tutti i nodi a 0 (non visitati ancora)
    DFS_rec(G, x, visitati) # iniziamo la ricerca in profondità su x
    return [x for x in range(n) if visitati[x]]
    # ritorniamo i nodi visitati (uguali a 1)
```

- Costo: **O(n + m)**
- Costo spazio: O(n) (lista visitati)

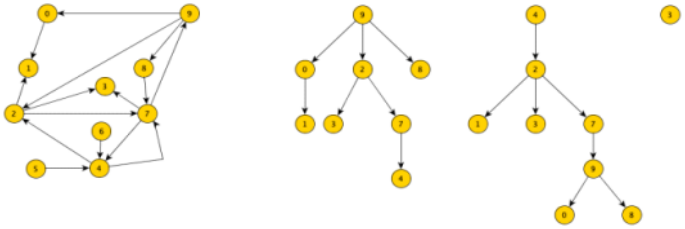
Iterativa su Liste di Adiacenza

```python
def DFS_iter_list(G, x):
    visitati = [0]*len(G) # inizializziamo tutti i nodi a 0
    pila = [x] # inizializza la pila col nodo di partenza
    while len(pila) != 0: # finche non si svuota la pila
        x = pila.pop() # prendiamo l'ultimo elem.
        if not visitati[x]: # se non è stato già visitato (0)
            visitati[x] = 1 # lo segnamo come visitato (1)
            # aggiungiamo i vicini non visitati
            for v in G[x]: # per ogni nodo adiacente a x
                if not visitati[v]: # se non è stato visitato
                    pila.append(v) # lo aggiungiamo alla fine
    return [x for x in range(n) if visitati[x]]
    # ritorniamo i nodi visitati (uguali a 1)
```

- Costo: **O(n + m)**
- Costo spazio: O(n)



Il nodo di partenza è 1



A sinistra un grafo G, a destra i tre **alberi DFS** ottenuti partendo rispettivamente dai nodi 9, 4 e 3

Con una visita DFS gli archi si dividono in:

- **Archi attraversati:** quelli che consentono di raggiungere nuovi nodi
- **Altri archi:** quelli che non vengono usati perche separati o impossibili da usare (grafo diretto)

I nodi e gli archi **attraversati** formano un **albero DFS**. Un **albero DFS** può essere memorizzato tramite **vettore dei padri**.

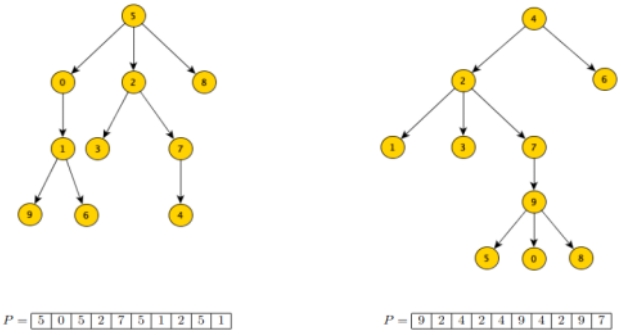
Vettore dei Padri

Il **vettore dei padri P** di un albero DFS di un grafo di **n nodi** ha **n componenti**:

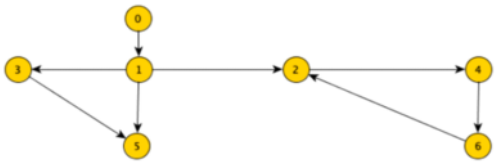
- Se **i** è un **nodo** dell'albero DFS: **P[i]** contiene il **padre** del **nodo i** (il padre della radice è la radice stessa)
- Se **i non è un nodo** dell'albero**: P[i] contiene -1**

Modificando leggermente il codice del DFS possiamo farci restituire il **vettore dei padri** invece che il **vettore dei visitati**

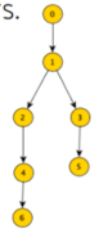
```python
def Padri(G, x):
    # P[a] = b -> b è il padre di a
    # P[a] = -1 -> a non è stato visitato
    ####
    def DFS_rec(G, x, P):
        for v in G[x]: # per ogni nodo adiacente a x
            if P[v] == -1: # se non è stato già visitato
                P[v] = x # il nodo da cui arriviamo sarà il padre
                DFS_rec(G, v, P) # controlliamo anche esso
    ####
    n = len(G)
    P = [-1]*n # inizializziamo tutti i nodi a -1 (non visitati ancora)
    P[x] = x # il padre della radice è la radice stessa
    DFS_rec(G, x, P) # iniziamo la ricerca in profondità su x
    return P # ritorniamo il vettore dei padri
```



Es: nel primo albero, per il **nodo 0** il padre è **P[0] = 5** (nodo 5)



Al termine $P[v]$ contiene -1 se v non è stato visitato altrimenti contiene il padre di v nell'albero DFS.



```
>>> G=[[1], [2, 3, 5], [4], [5], [6], [], [2]]

>>> Padri(0,G)
[0, 0, 1, 1, 2, 3, 4]

>>> Padri(5,G)
[-1, -1, -1, -1, -1, 5, -1]
```

Spesso però, invece di sapere solo se un **nodo y è raggiungibile da x,** vogliamo anche determinare un **cammino** che ci consenta di **andare da x a y**.
Tramite il vettore dei padri dell'albero DFS, radicato in **x**, basta vedere se il **nodo y sia nell'albero** e restituire la lista in "reverse" dei nodi incontrati fino ad y.

Metodo Iterativo

```python
def Cammino_iter(P, x):
    if P[x] == -1: return []
    # se il nodo non è nel vettore dei padri ritorniamo nulla
    path = [] # lista contenente i nodi che incontreremo
    while P[x] != x: # finchè il padre non arriviamo alla radice
        path.append(x) # aggiungiamo il nodo incontrato alla lista
        x = P[x] # risaliamo al nodo padre per controllarlo
    path. append(x) # aggiungiamo la radice
    path.reverse() # invertiamo la lista
    return path # e la ritorniamo
```

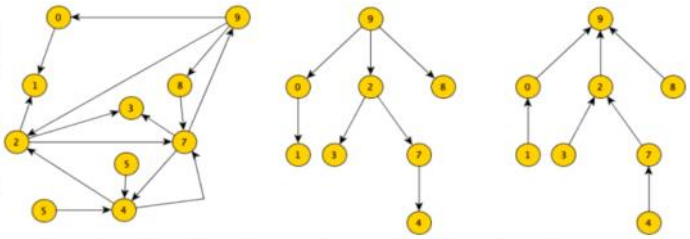
Costo: O(n)

Metodo Ricorsivo

```python
def Cammino_rec(P, x):
    if P[x] == -1: return []
    # se il nodo non è stato visitato, ritorniamo nulla
    if P[x] == x: return [x] # se siamo alla radice ritorniamo essa
    return Cammino_rec(P, P[x]) + [x]
    # ritorniamo i padri visitati ricorsivamente e il nodo corrente
```

Costo: O(n)

```
>>>P=padri(9,G)
>>>cammino(7,P)
[9,2,7]
>>>cammino(1,P)
[9,0,1]
>>>cammino(5,P)
[]
```



$$P = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 9 & 0 & 9 & 2 & 7 & -1 & -1 & 2 & 9 & 9 \\ \hline \end{array}$$

ATTENZIONE: se esistono più cammini dal nodo x a y, questo algoritmo **non garantisce** di **restituituire il cammino minimo** (quello che attraversa meno archi)

Qui ad esempio il **cammino minimo da 4 a 3** è **[4, 3]**
Però **cammino(3, P)** restituisce **[4, 0, 1, 2, 3]**

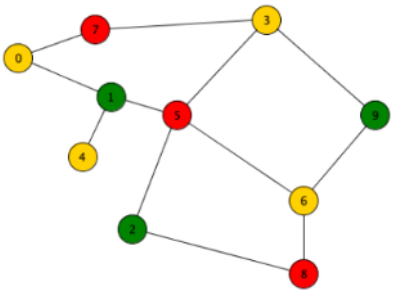
Grafi Colorati

lunedì 17 marzo 2025 12:45

Dato un grafo **connesso G** ed un intero **k** vogliamo sapere se è possibile **colorare** i nodi del grafo in modo che nodi **adiacenti** abbiano sempre **colori distinti**.

Teorema dei 4 colori

Questo teorema dice che un **grafo planare** richiede **al più 4 colori** per essere **colorato**.
In generale un grafo può richiedere anche Θ(n) colori, ma non esiste algoritmo polinomiale che, dato un grafo planare G, **determini** se **G è 3-colorabile.**
Invece semplicemente un grafo è 2-colorabile **se e solo se non contiene cicli** di lunghezza **dispari**, poiché tale ciclo rende **impossibile** la colorazione del grafo con due colori, poiché lungo il ciclo i colori devono **alternarsi**.



Es. grafo 3-colorabile
(che può essere anche 2-colorabile)

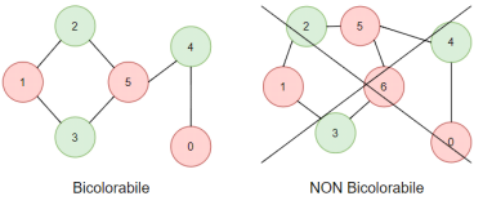
Algoritmo di bi-colorazione

Questo algoritmo prova che un **grafo senza cicli dispari** può essere sempre colorato:

- Colora il nodo 0 col colore 0
- Effettua una **DFS** a partire da 0 e nella visita, a ciascun nodo x incontrato si assegna **0 o 1**.
  Si sceglie il colore da assegnare in modo che sia diverso dal colore dato al nodo padre che ti ha portato a x

Prova di Correttezza: siano x e y due nodi adiacenti in G, consideriamo i due possibili casi e vediamo che in entrambi i casi i due nodi al termine avranno colori opposti.

1) **L'arco (x, y) viene attraversato durante la visita**. In questo caso i due nodi avranno colori distinti.
2) **L'arco (x, y) NON viene attraversato durante la visita**:
   - Sia x il nodo visitato prima, **esiste un cammino** in G che da **x porta a y** (quello seguito dalla visita)
   - Questo cammino si chiude a formare un ciclo con l'arco (y, x).
   - Il **ciclo** è di lunghezza **pari** per ipotesi, quindi il **cammino** è di lunghezza **dispari.**
   - Poiché sul cammino i colori si **alternano**, il primo nodo (x) e l'ultimo (y) avranno **colori diversi**



Algoritmo per bi-colorare grafi connessi G senza cicli dispari

```python
def Colora(G):
    # funziona solo su grafi connessi bicolorabili (senza cicli dispari)
    ####
    def DFS_rec(G, x, Colore, c):
        Colore[x] = c # assegna il colore al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a x
            if Colore[y] == -1: # se non è stato
                DFS_rec(G, y, Colore, 1-c)
                # lo colora con il colore inverso a quello di x
                # se x = 0 -> 1-0 = 1 -> y = 1
                # se x = 1 -> 1-1 = 0 -> y = 0
    ####
    Colore = [-1]*len(G) # inizializza i nodi a -1 (non colorati)
    DFS_rec(G, 0, Colore, 0) # partiamo la colorazione dal nodo 0
    return Colore # ritorniamo i nodi colorati
```

Algoritmo per bi-colorare grafi G se è bi-colorabile

```python
def Colora1(G):
    # Se il grafo è bicolorabile (senza cicli dispari) esegue la bicolorazione
    # altrimenti una lista vuota
    ####
    def DFS_rec(G, x, Colore, c):
        Colore[x] = c # assegna il colore al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a x
            if Colore[y] == -1: # se non è stato colorato
                if not DFS_rec(G, y, Colore, 1-c):
                    # se dal nodo successivo ci sono cicli dispari
                    return False # ritorna False (impossibile colorare)
            elif Colore[y] == Colore[x]:
                # se è stato colorato ed ha lo stesso colore del padre
                return False # ritorna False (impossibile colorare)
        return True # se non ci sono stati problemi ritorna True (colorato)
    ####
    Colore = [-1]*len(G) # inizializza i nodi a -1 (non colorati)
    if DFS_rec(G, 0, Colore, 0): # se non ci sono cicli dispari
        return Colore # ritorniamo la lista dei nodi colorati
    return [] # altrimenti la lista vuota
```

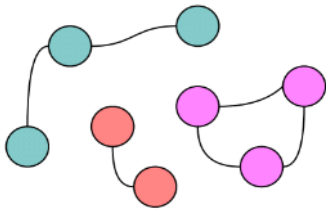
Nell'esempio a destra dell'immagine sopra avremmo che sul controllo al nodo rosso 6, vediamo che il nodo adiacente 5 ha Colore[5] = "rosso", quindi entra nell'elif e siccome i colori di 5 e 6 sono uguali, ritorna False

La complessità di entrambi è O(n + m). Però siccome in un grafo connesso m ≥ n-1 abbiamo che il costo è **O(m)**

Componenti

lunedì 17 marzo 2025 17:17

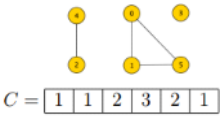
Una **componente connessa** (o semplicemente componente) di un grafo (**indiretto**) è un **sottografo** composto da un insieme **massimale** di nodi connessi da cammini
Quindi i diversi sottografi componenti costituiscono una **partizione** del grafo originale.
Un **grafo indiretto** si dice **connesso** se ha **una sola** componente connessa.



Vogliamo calcolare il vettore Componenti delle **componenti connesse** di un grafo G, associando ad ogni vertice, un indice che ne indica la componente. Quindi dato un grafo G, si vuole costruire un array Componenti t.c.:

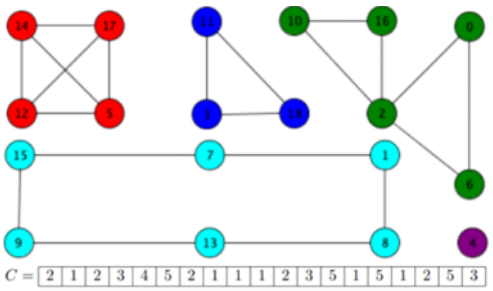
Componenti[x] = Componenti[y] solo se x e y sono nella **stessa componente**

```python
def Componenti(G):
    ####
    def DSF_rec(G, x, Comp, c):
        Comp[x] = c # assegna il val. del componente al nodo corrente
        for y in G[x]: # per ogni nodo adiacente a quello corrente
            if C[y] == 0: # se non gli è stato gia assegnato il componente
                DFS_rec(G, y, Comp, c)
                # assegna il componente del nodo corrente a quello adiacente
    ####
    Comp = [0]*len(G)
    c = 1
    for x in range(len(G)): # per ogni nodo
        if C[x] == 0: # se non gli è stato assegnato il componente
            DFS_rec(G, x, Comp, c) # eseguiamo la ricerca in profondità
            # per cercare i nodi del componente
            c += 1 # finito il componente corrente,
            # aumentiamo c per passare al componente successivo
    return C
```



$C = \boxed{1}\boxed{1}\boxed{2}\boxed{3}\boxed{2}\boxed{1}$

3 componenti del grafo



$C = \boxed{2}\boxed{1}\boxed{2}\boxed{3}\boxed{4}\boxed{5}\boxed{2}\boxed{1}\boxed{1}\boxed{1}\boxed{2}\boxed{3}\boxed{5}\boxed{1}\boxed{5}\boxed{1}\boxed{2}\boxed{5}\boxed{3}$

Una componente **fortemente connessa** di un grafo **diretto** è un **sottografo** composto da un insieme **massimale** di nodi **connessi da cammini**.
Un **grafo diretto** si dice **fortemente connesso** se ha una sola componente.

Vogliamo calcolare il vettore Componenti delle **componenti fortemente connesse** di un grafo G, quindi:

Componenti[x] = Componenti[y] solo se x ed y sono nella stessa **componente fortemente connessa**

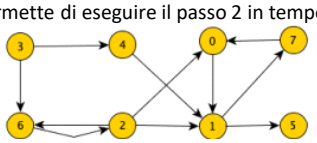
L'algoritmo delle componenti connesse non basta per quelle fortemente connesse, poiché se c'è un **cammino che da x mi porta ad y**, è necessario che ci sia anche un **cammino da y mi porta ad x. Un idea per l'algoritmo è:**

1) Calcola l'insieme **A** dei **nodi raggiungibili da x** (Semplice visita DFS. Costo O(n + m))
2) Calcola il **G^T trasposto di G** (Costo O(n + m))
3) Calcola l'insieme **B** dei **nodi che portano ad x** (Visita DFS su **grafo trasposto.** Costo O(n + m))
4) Restituisce l'**intersezione** tra **A e B** (Costo O(n))

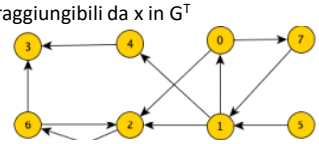
Costo totale: O(n + m)

Per il passo 3 possiamo usare il **grafo trasposto di G (G^T)** che ha gli stessi nodi di G ma gli **archi** sono in **direzione opposta.**
Questo ci permette di eseguire il passo 2 in tempo O(n + m) cercando i raggiungibili da x in G^T

G=



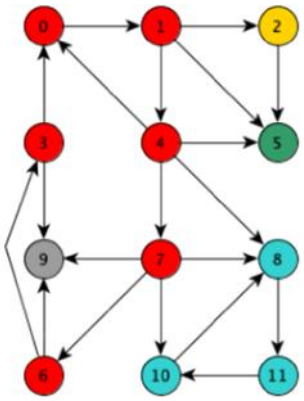
G^T=



Quindi i nodi che in G **portano ad x**, sono i nodi che in **G^T** sono **raggiungibili a partire da x**, e viceversa (nodi in G che sono raggiungibili da x, in G^T portano ad x)
Es: nell'immagine i nodi in G che portano al nodo 5 sono **[1, 4, 0, 7, 2, 3, 6].** Allo stesso modo in G^T i nodi raggiungibili a partire dal nodo 5 sono **[1, 4, 0, 7, 2, 3, 6].**

```python
def ComponenteFC(G, x):
    AVisit = DFS(G, x) # Visita DFS sul grafo normale
    GT = Trasposto(G) # Creiamo il Trasposto del grafo
    BVisit = DFS(GT, x) # Visita DFS sul grafo trasposto
    CompFC = []
    # prendiamo l'intersezione tra i due array
    for n in range(len(G)):
        if AVisit[n] == BVisit[n] == 1: # se entrambi sono stati visitati
            # quindi il nodo è:
            # - raggiungibile da x (grafo normale)
            # - porta ad x (grafo trasposto)
            CompFC.append(n) # lo aggiungiamo al componente
    return CompFC

def Trasposto(G):
    GT = [[] for n in G] # lista di adiacenze vuota
    for n in range(len(G)):
        for v in G[n]: # per ogni nodo adiacente
            GT[v].append(n)
            # aggiunge alla lista di adiacenze il nodo padre
    return GT
```

```python
>>>G=[
    [1], [2, 4, 5], [5], [1, 6],[0, 5, 7, 8],
    [], [3], [6, 8, 9, 10], [11], [],[8], [10]
    ]

>>> componenteFC(0, G)
[0,1,3,4,6,7]

>>> componenteFC(8, G)
[8,10,11]
```



$C = \boxed{1}\boxed{1}\boxed{2}\boxed{1}\boxed{1}\boxed{3}\boxed{1}\boxed{1}\boxed{4}\boxed{5}\boxed{4}\boxed{4}$

Grazie all'algoritmo ComponenteFC(G, x) posso ottenere un algoritmo per calcolare il vettore CF delle componenti fortemente connesse

```python
def CompFC(G):
    FC = [0]*len(G)
    c = 1
    for n in range(len(G)): # per ogni nodo di G
        if FC[n] == 0: # se non è stato visitato
            E = ComponenteFC(G, n)
            # prendiamo tutti i nodi della componente connessa di n
            for x in E: # per ognuno di quei nodi
                FC[x] = c # assegnamo il val. del componente
            c += 1 # e lo aumentiamo alla fine dell'assegnazione
    return FC
```

Il costo è Θ(n) (ciclo dei nodi) * O(n + m) (ricerca componente connessa di ogni nodo) = $O(n^2 + n*m) = O(n^3)$ (siccome in un grafo completo $m = O(n^2)$)

Ordinamento Topologico

giovedì 20 marzo 2025 15:46

Problema di Introduzione

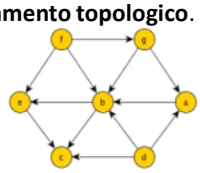
Abbiano un insieme di compiti da eseguire e alcuni compiti possono iniziare solo dopo che altri siano completati.
La coppia (a, b) indica che il **compito b** può essere **avviato** solo **dopo** che il **compito a è terminato**.
Quindi possiamo rappresentare questo problema con un **grafo orientato**:

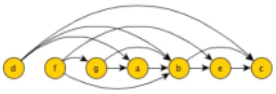
- I nodi sono i singoli compiti
- Gli archi rappresentano le dipendenze tra i compiti. Es: l'arco (a, b) indica che b dipende da a

È possibile completare i compiti rispettando le dipendenze? Ed è possibile un ordinamento con cui eseguire i compiti rispetta ndo le dipendenze?

Ordinamento Topologico

Per rispettare le dipendenze bisogna **ordinare** i nodi in modo che gli archi vadano **da sinistra verso destra**.
Questo ordinamento è chiamato **ordinamento topologico**.





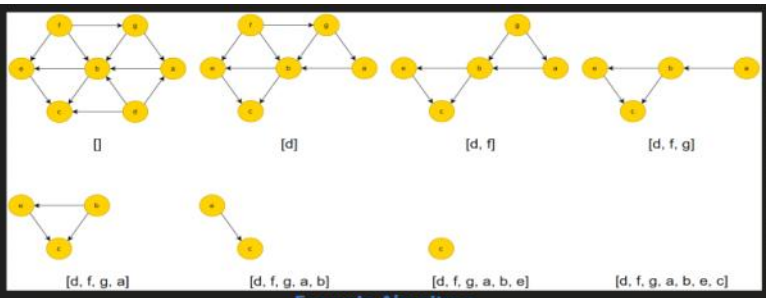
Un grafo diretto può avere da 0 a n! ordinamenti topologici.
Per poter avere un ordinamento topologico è **necessario che il grafo sia un DAG** (grafo diretto aciclico), poiché la presenza di un **ciclo** indica una **dipendenza circolare**, rendendo impossibile trovare un ordine di esecuzione.
Quindi il fatto che il grafo sia un DAG è **sufficiente** per dire che l'ordinamento topologico esista.

Algoritmo tramite Sorgente del DAG

Una DAG ha sempre un nodo sorgente (un nodo da cui non entrano archi).
Grazie a questa proprietà posso costruire un ordinamento topologico dei nodi del DAG cosi:

1) Inizio la sequenza dei nodi con una sorgente
2) Cancello dal DAG quel nodo sorgente e le frecce che partono da lui → ottengo cosi una nuova DAG che avrà un altro nodo sorgente
3) Itero questo ragionamento per ogni nodo dell'albero

```python
def SortTopDAG(G):
    n = len(G)
    gradoEntrata = [0]*n
    # inizializziamo il vettore dei gradi entranti
    for x in range(n):
        for y in G[x]: # per ogni nodo adiacente
        # incrementa il num. di archi in entrata
            gradoEntrata[y] += 1
    sorgenti = [ i for i in range(n) if gradoEntrata[i] == 0 ]
    # prende i nodi che hanno 0 archi in entrata
    Sorted = []
    while sorgenti: # finche non finisce i nodi sorgente
        x = sorgenti.pop() # prende l'ultimo nodo sorgente
        print(x)
        Sorted.append(x) # aggiunge quel nodo alla lista ordinata
        for v in G[x]: # per ogni nodo adiacente al sorgente
            print(f"\t{v}")
            gradoEntrata[v] -= 1 # rimuove il suo arco a quel nodo
            if gradoEntrata[v] == 0:
                # se il nodo ora non ha più archi in entrata
                sorgenti.append(v) # allora è un nuovo sorgente
    if len(Sorted) == n: return Sorted # se ha tutti i nodi
    return [] # altrimenti non si può ordinare
```



Esempio Algoritmo

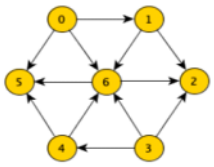
Complessità:

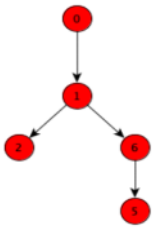
- Inizializza vettore gradi entranti: O(n + m)
- Inizializza insieme delle sorgenti: Θ(n)
  - Il while viene iterato O(n) volte
  - Il for interno al while viene iterato O(m) volte
- Costo ciclo while: O(n + m)

Costo complessivo: O(n + m)

Algoritmo tramite DFS su DAG

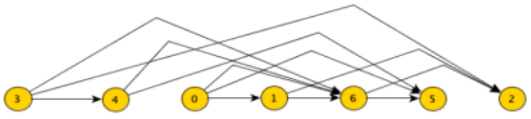
1) Effettua una visita DFS del DAG a partire dal nodo 0
2) Man mano che termina la visita dei vari nodi, inseriscili in una lista (quindi dopo che esegue il DFS sui nodi adiacenti)
3) L'ordinamento finale è il reverse della lista





lista=[2, 5, 6, 1, O, 4, 3]

lista=[3, 4, O, 1, 6, 5, 2]



```python
def SortTopDFS(G):
    def DFS_Sort(G, x, visitati, Sorted):
        visitati[x] = 1 # segniamo il nodo come visitato
        for v in G[x]:
            if not visitati[v]: # se non è stato visitato
                DFS_Sort(G, v, visitati, Sorted) # continua il DFS
        Sorted.append(x) # aggiungiamo il nodo alla lista
    ####
    n = len(G)
    visitati = [0]*n
    Sorted = []
    for x in range(n): # per ogni nodo
        if not visitati[x]: # non visitato
            DFS_Sort(G, x, visitati, Sorted) # esegue il DFS
    Sorted.reverse() # ritorniamo il reverse della lista
    return Sorted
```

Costo complessivo: O(n + m) + O(n) = O(n + m)

Prova di correttezza: Siano **x** e **y** due nodi in G, con arco da **x** a **y**. Consideriamo i due possibili casi e vediamo che, prima di effettuare il reverse, y precede x.

1. **L'arco (x, y) viene attraversato durante la visita**. Banalmente la visita di **y** finisce prima della visita di **x** e **y finisce nella lista prima di x**.
2. **L'arco (x, y) NON viene attraversato durante la visita**. Durante la visita di **x** il nodo **y** è **già stato visitato** e la sua **visita** è anche gia **terminata** (da y non c'è cammino che porta a x, altrimenti ci sarebbe un ciclo), anche in questo caso **y finisce nella lista prima di x**

Ricerca Cicli

venerdì 21 marzo 2025 12:27

Dato un grafo G (diretto o non) e un suo nodo x, vogliamo sapere se da x possiamo raggiungere un ciclo in G.

Un idea semplice (**SBAGLIATA**) sarebbe che se nella visita si incontra un nodo già visitato allora è un ciclo e ritorniamo True.
Il problema è che funziona solo con grafi diretti, poiché, nei grafi non diretti, se x ha un arco su y allo stesso modo y ne ha uno su x.
Quindi questo viene interpretato come un ciclo di lunghezza 2.

Per risolvere il problema dobbiamo distinguere se il nodo già visitato che incontro non sia il nodo padre che mi ci ha portato.

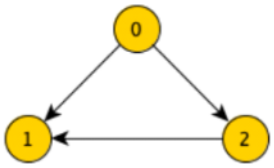
```
def Ciclo_Sbag(G, x):
    def DFS_ciclo1(G, x, padre, visitati):
        visitati[x] = 1
        for v in G[x]:
            if visitati[v] == 1: # se è un nodo visitato
                if v != padre: # e non è il padre
                    return True # c'è un ciclo
            else: # se non è stato visitato
                if DFS_ciclo1(G, v, x, visitati):
                    # controlliamo nei nodi adiacenti
                    return True
        return False # non abbiamo trovato cicli
    ####
    visitati = [0]*len(G)
    return DFS_ciclo1(G, x, x, visitati)
```

Complessità: O(n)

Il costo O(n) è dovuto dal fatto che, se il grafo non contiene cicli allora ha **m = n-1 archi** e quindi O(n + m) = O(n).
Se invece ci sono cicli, e se ne scopre uno dopo aver considerato al più n archi, allora termina la visita

Però il codice è ancora **sbagliato** poiché in un grafo diretto, se incontriamo un nodo gia visitato, non significa necessariamente che si è in presenza di un ciclo (quindi termina con True anche in assenza di ciclo)

```
G=[
    [1, 2],
    [  ],
    [1]
]
```

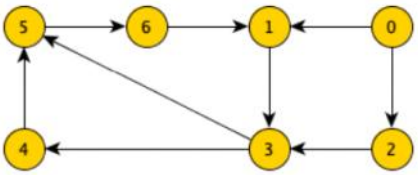



```
>>>ciclo(0,G)
True
```

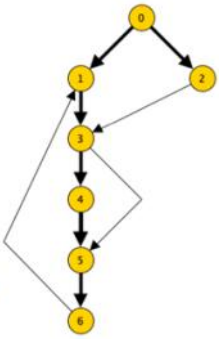
Ad esempio in questa immagine partendo da 0, si arriva prima al nodo 1 (impostandolo come visitato) e poi al nodo 2. Dal nodo 2 però arriviamo al nodo 1 e siccome è già stato visitato e quel nodo non è il padre di 2 (il padre è il nodo 0) allora pensa che ci sia un ciclo.

Nella visita DFS possiamo incontrare i **nodi già visitati in tre modi diversi**:

- **Archi in avanti** (frecce dirette **da antenato a discendente**)
- **Archi di attraversamento**
- **Archi all'indietro** (frecce dirette **da discendente a antenato**)



6 – 1 è arco all'indietro
2 – 3 è arco di attraversamento
3 – 5 è arco in avanti



Solo la presenza di archi all'indietro testimonia la presenza di un ciclo.
Quindi per poter risolvere il problema sulla ricerca del ciclo, devo distinguere la scoperta di nodi già visitati con un arco all'indietro.
Durante una visita, solo nel caso di un arco all'indietro, la visita del nodo già visitato termina la ricorsione.
Quindi possiamo impostare un nuovo valore per un nodo visitato:

- Un nodo vale **0** se **non è ancora** stato **visitato**
- Un nodo vale **1** se è stato **visitato ma** la **visita ricorsiva** su quel nodo **non è ancora finita**
- Un nodo vale **2** se è stato **visitato e** la **visita ricorsiva è finita**

Cosi scopro un **ciclo** solo se trovo un arco diretto verso un nodo in "**elaborazione**" (visitato ma la visita non è ancora finita)
Per sapere se un grafo contiene un ciclo, devo visitarlo tutto, quindi dovrò eseguire il DFS su tutti i nodi non ancora visitati

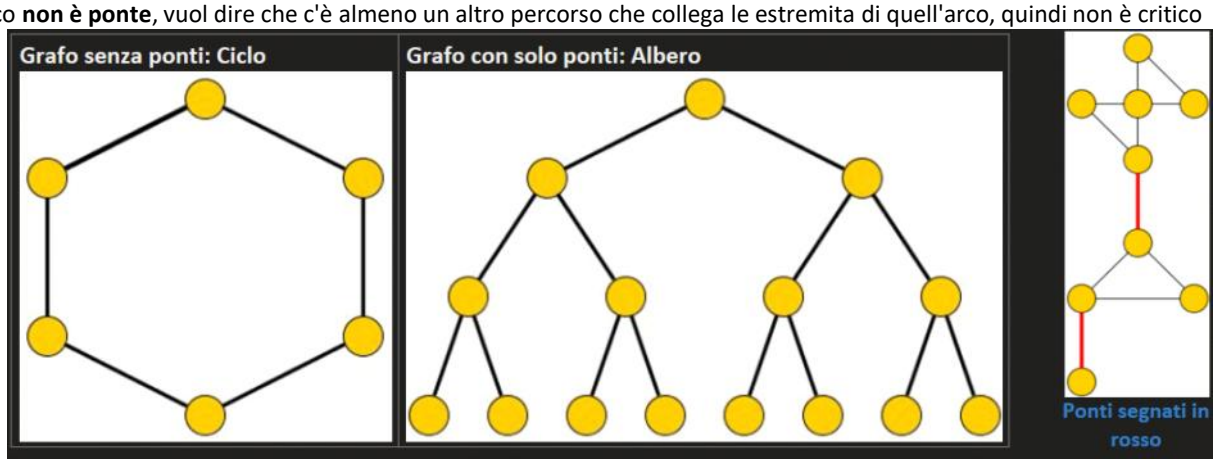
```python
def CicloInd(G):
    def DFS_ciclo2(G, x, visitati):
        visitati[x] = 1
        for y in G[x]:
            if visitati[y] == 1:
                # nodo in "elaborazione" -> ciclo
                return True
            elif visitati[y] == 0:
                # nodo non visitato -> continua DFS
                if DFS_ciclo2(G, y, visitati):
                    return True
    visitati[x] = 2 # nodo completamente esplorato
    return False
    ####
    # 0: non visitato, 1: in elaborazione, 2: completato
    visitati = [0]*len(G)
    for x in range(len(G)):
        if visitati[x] == 0:
            # avvia DFS solo sui nodi non ancora visitati
            if DFS_ciclo2(G, x, visitati):
                return True
    return False
```

Complessità: O(n + m)

Ponti

venerdì 21 marzo 2025 14:41

In un grafo, se viene rimosso un arco, si può perdere la proprietà di connessione, creando due componenti separate.
Un **arco** la cui **eliminazione disconnette il grafo** è detto **ponte**.
Essi rappresentano **criticità** del grafo ed è quindi utile identificarli.
Se un arco **non è ponte**, vuol dire che c'è almeno un altro percorso che collega le estremita di quell'arco, quindi non è critico



Dato un grafo connesso G, vogliamo determinare l'insieme di ponti del grafo

Prima soluzione basata sulla **ricerca esaustiva:** prova per ogni arco se questo è un ponte o meno

- Verificare se un arco (a, b) è un ponte per G richiede tempo O(m)
  Basta eliminare l'arco (a, b) e, con una visita DFS, controllare **se b è raggiungibile da a**
- Complessità: **m (num archi) * O(m) (controllo ponte) = O(m^2) = O(n^4)**

Seconda soluzione: usare una visita DFS modificata

- I ponti vanno ricercati tra gli **n - 1 archi dell'albero DFS.** Infatti un arco non presente nell'albero DFS non può essere ponte (se lo elimino, gli archi dell'albero continuano a garantire la connessione)
- Gli archi dell'albero che non sono ponti sono **"coperti"** dagli altri archi del grafo che non sono stati attraversati nel DFS (archi all'indietro)

Proprietà: Sia (u, v) un arco dell'albero DFS con **u padre di v**. L'arco **u - v** è un **ponte se e solo se** non ci sono archi tra **u o antenati di u** e i nodi del sottoalbero radicato in **v.**
Prova:

- **Per assurdo**: sia **x - y** un arco tra un **antenato di u** e un **discendente di v**. Dopo l'eliminazione di **u - v** i nodi dell'albero restano comunque **connessi grazie all'arco x - y**
- L'eliminazione di u - v disconnette i nodi dell'albero radicato in v dal resto del grafo. In questo caso tutti gli archi che non appartengono all'albero e che partono dal sottoalbero di v punterebbero solo verso v o suoi discendenti, perché non esistono altri archi che li connettono a nodi "più in alto" di v nell'albero.

Durante la visita da un nodo u al nodo v, il nodo u non sa ancora se nel sottoalbero di v ci sia un arco che torna sugli antenati di v.
Quindi dopo la visita del sottoalbero di v, v comunica a u il nodo più in alto raggiungibile (con archi di ritorno o altri percorsi).
Tramite questo valore possiamo vedere che, se il **sottoalbero v non può arrivare ad un antenato di u**, allora u - v **è un ponte.**

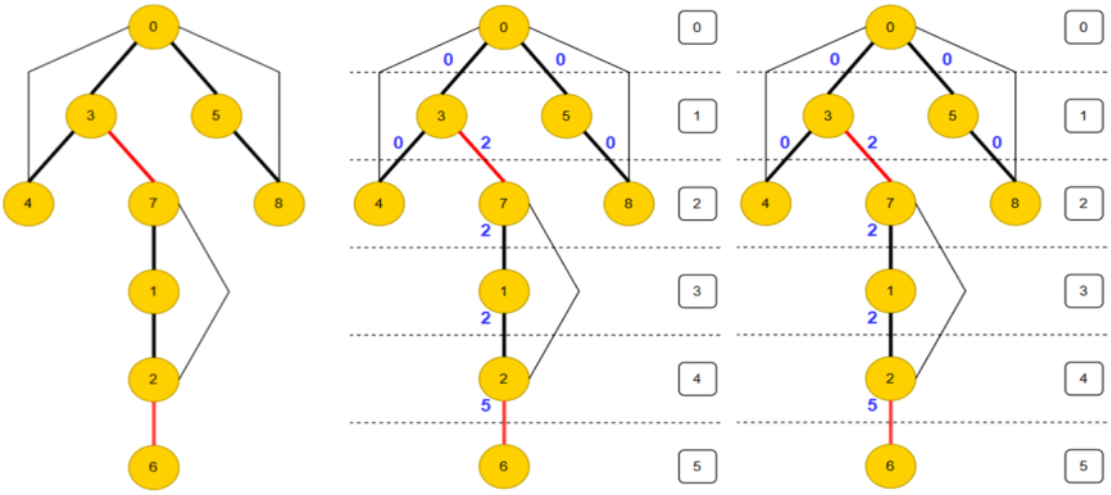
Algoritmo Tramite DFS

Per ogni arco padre-figlio (u, v) dell'albero DFS, possiamo scoprire se quell'arco è un ponte tramite la seguente strategia:

- Ogni nodo v:
  1. Calcola la sua altezza nell'albero
  2. Calcola e restituisce al padre u, l'altezza minima che riesce a raggiungere con archi che partono dai suoi discendenti
- Il nodo u, ricevuta l'altezza minima raggiungibile dal figlio, la confronta con la sua altezza e nel caso in cui l'altezza di u sia minore di quella restituita allora l'arco **u - v è ponte**.

Riassunto algoritmo:

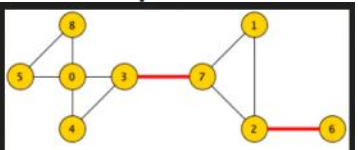
- **Nodo v**: Esplora il suo sottoalbero e restituisce al padre u il val**. min_alt** del livello minimo raggiungibile da v e i suoi discendenti, usando anche eventuali archi all'indietro
- **Nodo u**: Confronta min_alt restituito con la propria altezza
  - Se min_alt > altezza di u, l'arco u - v è l'unico collegamento, quindi è **un ponte**
  - Se min_alt ≤ altezza di u, vi è un collegamento alternativo che collega il sottoalbero di v a u o ad un suo antenato, quindi l'arco **non è un ponte**



5 > 4 → 2 - 6 è ponte

2 > 1 → 3 - 7 è ponte

```python
def TrovaPonti(G):
    def DFS_ponti(G, x, padre, altezza, ponti):
        # impostiamo l'altezza del nodo corrente
        if padre != -1:
            altezza[x] = altezza[padre] + 1
        else:
            altezza[x] = 0
        # inizializzo l'altezza minima raggiungibile dal sottoalbero di x
        min_raggiungibile = altezza[x]
        for v in G[x]: # per ogni nodo adiacente di x
            if altezza[v] == -1: # se non è stato ancora visitato
                min_alt = DFS_ponti(G, v, x, altezza, ponti)
                # cerco l'altezza minima che riesce a raggiungere il sottoalbero di v
                if min_alt > altezza[x]:
                    # se l'altezza di x è minore di quella restituita da v
                    ponti.append((x, v)) # allora (x, v) è un ponte
                min_raggiungibile = min(min_raggiungibile, min_alt)
                # prendo il minimo tra l'altezza di x e quella restituita da v
            elif v != padre:
                # se è stato visitato e (x, v) è un arco all'indietro
                # prendo il minimo tra l'altezza di x e quella di v
                min_raggiungibile = min(min_raggiungibile, altezza[v])
        return min_raggiungibile
    ####
    n = len(G)
    altezza = [-1]*n
    ponti = []
    # inizio la DFS dal nodo 0
    DFS_ponti(G, 0, -1, altezza, ponti)
    return ponti
```



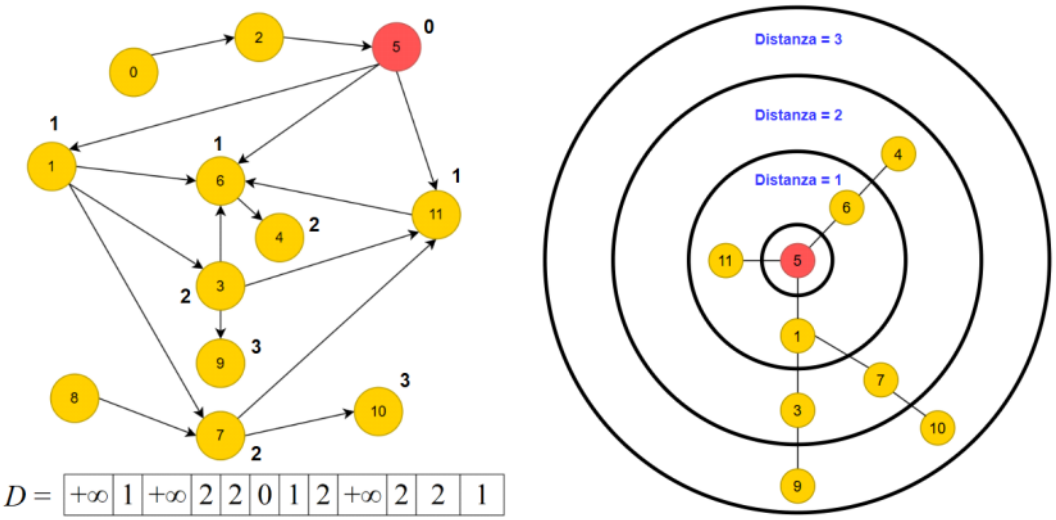
```
G=[
[3,4,5,8],
[2,7],
[1,6,7],
[0,4,7],
[0,3],
[0,8],
[2],
[1,2,3],
[0,5]
]

>>> TrovaPonti(G)
[(2, 6), (3, 7)]
```

La complessità della procedura è $O(n+m)$

BFS e Distanze

sabato 22 marzo 2025 13:00

Vogliamo scoprire la **distanza (minima)** tra due nodi a e b, quindi il num. minimo di archi da attreversare per raggiungere b da a.
Dobbiamo prima calcolare le **distanze** di tutti i nodi di G partendo da x.
Le distanze calcolate verrano inserite in un vettore D, dove in D[y] troviamo la distanza di y da x. Se y non è raggiungibile D[y] = +∞



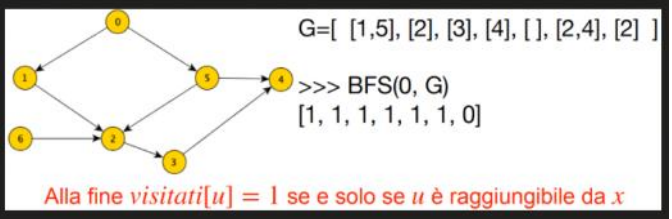
$D = \boxed{+\infty \mid 1 \mid +\infty \mid 2 \mid 2 \mid 0 \mid 1 \mid 2 \mid +\infty \mid 2 \mid 2 \mid 1}$

La **visita in ampiezza (Breadth First Search o BFS)** esplora i nodi partendo dai nodi adiacenti prima di passare ai livelli successivi.
Per ottenere un **albero BFS** manteniamo in una **coda** i nodi visitati i cui adiacenti non sono ancora stati esaminati completamente.
Ad ogni passo, preleviamo il primo nodo dalla coda, e se tra gli adiacenti vi è un nuovo nodo, lo visitiamo e lo aggiungiamo alla coda.
Quindi nella coda finiscono solo i nodi da visitare successivamente per controllare i suoi adiacenti.
I passaggi base dell'algoritmo della visita BFS a partire dal nodo x sono:

- Cominciamo con la coda contenente solo x (nodo di partenza)
- Finchè la coda non viene svuotata (quindi ancora ci sono nodi da visitare), ad ogni passo:
  - Un nodo y viene estratto dalla coda
  - Tutti i nodi adiacenti di y, che non sono stati gia visitati, vengono visitati e messi in coda

Implementazione con Coda Semplice

```python
def BFS(G, x):
    visitati = [0]*len(G)
    visitati[x] = 1
    coda = [x] # coda inizializzata col nodo di partenza
    while coda: # finche non si svuota la coda
        u = coda.pop() # prendiamo il primo nodo della coda
        for y in G[u]: # per ogni adiacente al nodo
            if visitati[y] == 0: # se non è stato visitato
                visitati[y] = 1 # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return visitati
```



Complessità:

- Un nodo finisce nella coda **una volta sola** (perche risulta visitato e non ci finisce più) quindi il **while** itera **O(n) volte**
- Le liste di adiacenza vengono scorse al più una volta, quindi il costo tot. dei **for** è **O(m)**

Se le op. di inserimento/cancellazione della coda costassero Θ(1), avremmo complessità totale **O(n + m)**

Però abbiamo implementato una **coda sempolice** tramite lista, inserendo i nodi in coda e rimuovendoli in testa.

- **Inserimento in coda**: (append()) costa **O(1)**
- **Estrazione in testa**: (pop(0)) ha costo proporzionale al num. di elem. presenti e possono essere anche O(n)

Quindi il costo di questa implementazione è **O(n²)**

Abbiamo due implementazioni alternative che riducono il costo a O(n + m)

- Effettuare solo cancellazioni logiche (quindi invece di rimuovere veramente l'elemento, lo copiamo con una var di appoggio)
- Uso di **deque** (coda doppiamente puntata)

Implementazione con Cancellazioni Logiche

Invece di eliminare effettivamente il primo elemento, uso un puntatore i che indica l'inizio della coda nella lista.

Ogni volta che "eliminiamo" il nodo dalla coda, incrementiamo il puntatore (il controllo della coda vuota ora vedrà se il num. di elem è maggiore del puntatore).

```python
def BFSLogica(G, x):
    visitati = [0]*len(G)
    visitati[x] = 1
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non superiamo la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if visitati[y] == 0: # se non è stato visitato
                visitati[y] = 1 # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return visitati
```

Implementazione con Deque

Il modulo python **collections** mette a disposizione molte collezioni di dati rispetto a quelle basiche.
Una di queste è **deque** che è ottima per l'inserimento/cancellazione da entrambi i lati poiché il costo di entrambe le op. è O(1).
In particolare permette anche l'inserimento/estrazione dalla sinistra (appendleft() / popleft())

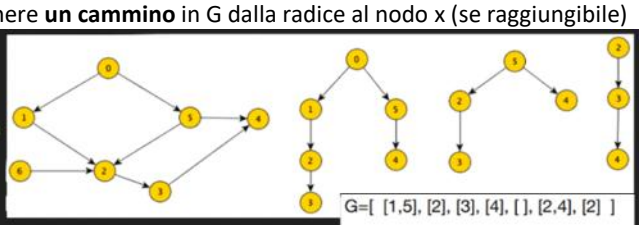
```python
def BFS_deque(G, x):
    visitati = [0]*len(G)
    visitati[x] = 1
    from collections import deque
    coda = deque([x]) # deque inizializzata col nodo di partenza
    while coda: # finche non si svuota la coda
        u = coda.popleft() # copiamo il primo nodo della coda
        for y in G[u]: # per ogni adiacente al nodo
            if visitati[y] == 0: # se non è stato visitato
                visitati[y] = 1 # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return visitati
```

Operazione	list (array dinamico)	deque (doubly linked list)
append(x)	$O(1)$ ammortizzato	$O(1)$
pop()	$O(1)$	$O(1)$
appendleft(x)	$O(n)$	$O(1)$
popleft()	$O(n)$	$O(1)$
Accesso casuale	$O(1)$	$O(n)$

Visita BFS con Vettore dei Padri

Modifichiamo il codice per restituire l'albero BFS rappresentato tramite vettore dei padri
Grazie al vettore dei padri P, con **cammini(P, x)** in O(n) possiamo ottenere **un cammino** in G dalla radice al nodo x (se raggiungibile)

```python
def BFSPadri(G, x):
    P = [-1]*len(G)
    P[x] = x # il padre della radice è la radice stessa
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non si svuota la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if P[y] == -1: # se non è stato visitato
                P[y] = u # viene visitato
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return P
```



G=[[1,5], [2], [3], [4], [ ], [2,4], [2]]

```python
>>> BFSpadri(0, G)
[0, 0, 1, 2, 5, 0, -1]

>>> BFSpadri(5, G)
[-1, -1, 5, 2, 5, 5, -1]

>>> BFSpadri(2, G)
[-1, -1, 2, 2, 3, -1, -1]
```

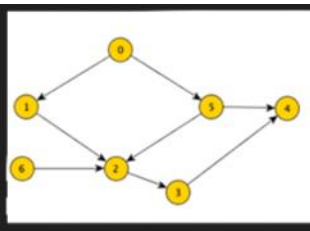
Visita BFS per le Distanze

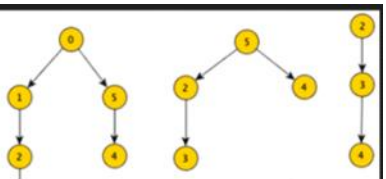
Modifichiamo il codice per restituire il **vettore delle distanze D**

- Inizialmente al nodo x viene assegnata distanza zero e a tutti gli altri nodi distanza -1
- A ciascun nodo via via visitato viene assegnata la distanza del padre incrementata di 1.

Al termine D[y] = -1 se il nodo y non è raggiungibile da x, altrimenti si avrà la **distanza minima** da x.

```python
def BFSDistanze(G, x):
    D = [-1]*len(G)
    D[x] = x # il padre della radice è la radice stessa
    coda = [x] # coda inizializzata col nodo di partenza
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non si svuota la coda
        u = coda[i] # copiamo il primo nodo della coda
        i += 1 # incrementiamo il puntatore
        for y in G[u]: # per ogni adiacente al nodo
            if D[y] == -1: # se non è stato visitato
                D[y] = D[u] + 1 # viene visitato incrementando la sua distanza
                # e finisce in coda per la visita degli adiacenti
                coda.append(y)
    return D
```





```
>>> BFSdistanze(0, G)
[0,  1,  2,  3, 2,  1, -1]

>>> BFSdistanze(6, G)
[-1, -1,  1,  2, 3, -1,  0]

>>> BFSdistanze(4, G)
[-1, -1, -1, -1, 0, -1, -1]
```

Correttezze BFS

Correttezza:
Alla fine dell'esecuzione, **visitati[y]** contiene **1** solo se il **nodo y è raggiungibile da x**

- **Se y è raggiungibile a partire da x allora visitati[y] = 1**. In questo caso c'è un **cammino P** da x a y.
  Supponiamo **per assurdo** che al termine **visitati[y] = 0**. Sia:
  - **b** il primo nodo che incontro nel cammino P con V[b] = 0
  - **a** il suo predecessore

  (questi due nodi esistono perche il cammino comincia con x e V[x] = 1 e termina con y e V[y] = 0)
  Un nodo che diventa 1 finisce in coda, quindi a è finito in coda, ma l'algoritmo si ferma quando la coda si svuota quindi c'è stato un momento in cui a è stato estratto dalla coda, e i suoi vicini sono stati controllati e, se non erano ancora stati visitati, v isitati (e poi inseriti in coda) quindi anche b è stato controllato e se non ancora visitato sarebbe stato visitato e posto a 1
- **Se y non è raggiungibile a partire da x allora visitati[y] = 0**. Supponiamo **per assurdo** che al termine **visitati[y] = 1**
  Visitati[y] è 1 solo se y viene inserito in coda ma questo accade perché risulta adiacente ad un nodo che viene estratto, and ando all'indietro questo processo deve terminare perche ciascun nodo viene inserito al più una volta nella coda, quindi si crea un cammino da x a y, il che è assurdo avendo supposto che y non è raggiungibile da x.

Non è difficile dimostrare che ogni vertice raggiungibile da x finisce nell'albero BFS. Vale in più:
Proprietà: la distanza minima di un vertice y da x nel grafo G equivale alla **profondità di y** nell'albero BFS
Prova: per **induzione** sulla distanza d di y da x.

- **Caso base:** È ovviamente vero per d = 0 in quanto l'unico vertice a distanza 0 x è x stesso che è a profondita 0 nell'albero.
- **Ipotesi induttiva:** Supponiamo sia vero per tutti i vertici con distanza al più **d - 1** e consideriamo quindi un vertice **u a distanza d**.
  Sia P un **cammino minimo** da x a u e sia **v il predecessore di u** in questo cammino.
  Per **ipotesi induttiva** v è a profondità **d - 1** nell'albero BFS. Se u è stato inserito nell'albero grazie a v allora si troverà in profondità d, assumiamo quindi che v sia stato inserito nell'albero grazie ad un nodo u $\neq$ v.
  La profondità di u **non** può essere **inferiore a d - 1** altrimenti avremmo trovato un cammino che va da x a v (tramite u) di lunghezza inferiore a d. D'altra parte non può essere la profondità di u maggiore di d - 1 perche il nodo v sarebbe stato visitato prima di u e y sarebbe stato inserito grazie al nodo v. Deve quindi aversi che la profondità di u è d - 1 e quella di v è d.

Riassunto prova:

- **Caso base:** Se d = 0, il solo nodo a distanza 0 da x è x stesso
- **Passo induttivo**: Supponiam oche i nodi a distanza d - 1 da x abbiano profondità d - 1 nell'albero BFS. Consideriamo un nodo **u** a distanza **d**.
  - Esiste un cammino minimo da x a u, e chiamiamo **v** il predecessore di **u** in questo cammino
  - Per ipotesi induttiva, **v** ha profondità d - 1 nell'albero
    - Se **u** viene scoperto grazie a **v**, allora viene inserito a **profondità d** nell'albero
    - Se **u** viene scoperto grazie ad un altro nodo **w ≠ v**, la sua profondità non potrebbe essere inferiore a **d**, perché altrimenti esisterebbe un **cammino più corto da x a v**, contraddicendo la **minimalità di d**
  - Inoltre **u** non può avere **profondità maggiore di d,** perché **v** sarebbe stato scoperto prima e avrebbe inserito **u** al liv. corretto

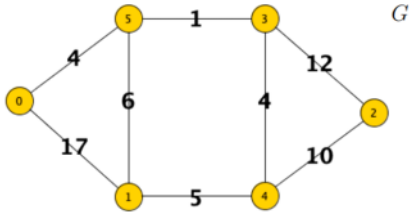
Grazie a questa proprietà, i cammini prodotti usando l'albero BFS sono **di lunghezza minima**, ecco perche l'albero BFS è detto **albero dei cammini minimi.**

Grafi Pesati

lunedì 24 marzo 2025 15:13

I **grafi pesati** sono grafi in cui gli archi hanno un val. (peso) numerico (intero o reale, anche negativo).
Per rappresentare questo tipo di grafi per l'arco **(x, y)** di peso **c**, nella lista di adiacenza di **x** ci sarà la coppia **(y, c)**.

$G =$



$$G = [\ [(1,17),(5,4)],\ [(0,17),(4,5),(5,6)],\ [(3,12),(4,10)],\ [(2,12),(4,4),(5,1)],\ [(1,5),(2,10),(3,4)],\ [(0,4),(1,6),(3,1)]\ ]$$

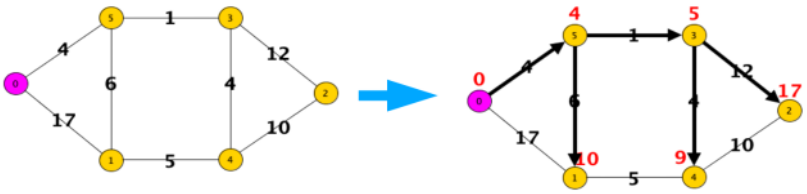
Se un grafo pesato ha val. interi e relativamente piccoli è possibile anche trasformarli in grafi non pesati.
Esistono diversi problemi che ne fanno uso, ad esempio:

- Abbiamo una mappa e vogliamo determinare il percorso più breve tra due punti. I nodi del grafo sono le località e un arco da una località all'altra ha costo pari alla lunghezza delle strade che collega i due punti.

In questo esempio però non possiamo trasformare il grafo in uno non pesato, poiché la lunghezza delle strade non è un num. intero e il val degli archi può essere anche molto alto.
Serve quindi un algoritmo che permette di trovare i **cammini minimi** lavorando direttamente **su grafi pesati** (con pesi anche non interi).

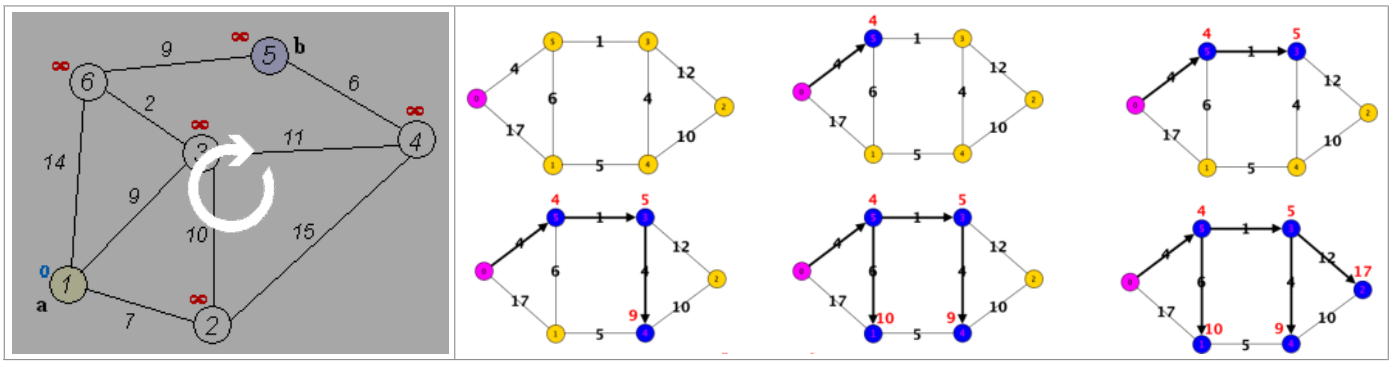
Algoritmo di Dijkstra

Problema: Dato un grafo pesato vogliamo trovare i cammini minimi e le distanze da un nodo x a tutti gli altri nodi.



I cammini minimi (in grassetto) e le distanze (in rosso)

- Costruisci l'albero dei cammini minimi un arco per volta, partendo dal nodo sorgente
  - Ad ogni passo aggiungi all'albero l'arco che produce il nuovo cammino **più economico**
  - Alla nuova destinazione assegna come distanza il costo del cammino



L'algoritmo rientra nel **paradigma della tecnica greedy**:

- La **seq. di decisioni irrevocabili**: decidi ad ogni passo il cammino (quindi la distanza) dal nodo sorgente ad un nuovo nodo. Una volta deciso non ritornare più su questa decisione.
- Le **decisioni vengono prese in base ad un criterio "locale"**: tra tutti i nuovi cammini che puoi trovare estendendo i vecchi di un arco prendi quello che costa di meno

Pseudo-codice Dijkstra(G, s):

- P[0, …, n-1]: vettore dei padri inizializzato a -1
- D[0, …, n-1]: vettore delle distanze inizializzato a +∞
- D[s], P[s] = 0, s
- while esistono archi {x, y} con P[x] != -1 e P[y] == -1 (nodo sorgente ha un padre e il nodo destinatario no):
  - Sia {x, y} quello per cui è minimo D[x] + peso(x, y)
  - D[y], P[y] = D[x] + peso(x, y), x
- return P, D

L'algoritmo non funziona però per grafi con pesi anche negativi e se è un grafo senza pesi si comporta come una visita BFS.

Correttezza

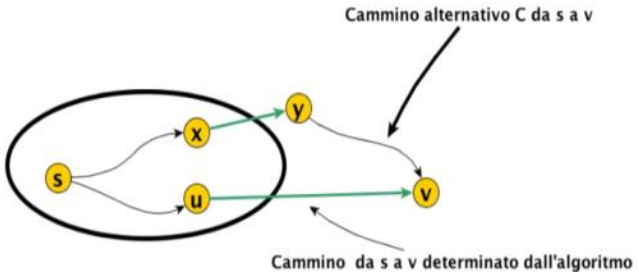
Correttezza per grafi con pesi positivi: ad ogni iter. del while viene assegnata una nuova distanza ad un nodo.

Per induzione sul num. di iter. mostreremo che la distanza assegnata è quella minima.

- **Caso base**: al passo 0 si assegna distanza zero alla sorgente e con pesi pos. non ci può essere una distanza inferiore
- Sia T_i l'albero dei cammini minimi costruiti fino al passo i > 0 e (u, v) l'arco aggiunto all'albero al passo i + 1.
  Vedremo che D[v] è la distanza min. di v da s. Basterà mostrare che il costo di un eventuale cammino alternativo è ≥ D[v].
  - Sia C un qualsiasi cammino da s a v alternativo a quello nell'albero, e (x, y) il primo arco che incontriamo percorrendo C all'indietro t.c x è nell'albero T_i e y no (tale arco esiste perché s è in T_i mentre v no).
  - **Per ipotesi induttiva** costo(C) ≥ Dist(x) + peso(x, y) (**questa affermazione è vera perche i pesi del grafo sono positivi)**
  - L'algoritmo ha preferito estendere l'albero T_i con l'arco (u, v) anziché l'arco (x, y) e in base alla regola con cui l'algoritmo sceglie l'arco con cui estendere l'albero deve quindi aversi D[x] + p(x, y) ≥ D[u] + p(u, v)
  - da cui segue: costo(C) ≥ D[x] + peso(x, y) ≥ D[u] + p(u, v) = D[v].
  - **Quindi il cammino alternativo ha costo superiore a D[v]**

Riassunto prova:
Caso base: la sorgente ha distanza 0, che è corretto perché nessun cammino può avere distanza negativa (pesi positivi)
Passo induttivo:

- Al passo i + 1, si aggiunge l'arco (u, v) nell'albero dei cammini minimi
- Si dimostra che D[v] assegnata è la min. possibile
- Se esiste un **cammino alternativo** C più conveniente per arrivare a v, ci sarebbe un arco (x, y) che attraversa per la prima volta un nodo fuori dall'albero.
- Dalla propr. dell'algoritmo, l'arco scelto (u, v) è sempre quello col costo minore tra i disponibili, quindi:
  D[x] + p(x, y) ≥ D[u] + p(u, v) = D[v]
- Da ciò segue che qualunque cammino alternativo ha costo D[v], quindi l'algoritmo assegna sempre il val. minimo



Implementazione Algoritmo tramite Lista

Nel vettore Lista, per ogni nodo x memorizziamo una **terna (definitivo, costo, origine)**

- **Definitivo**
  - È una **flag** che assume **1** se il costo per raggiungere x è stato **"definitivamente" stabilito**, quindi l'algoritmo ha confermato che non è possibile ottenere un percorso migliore a partire dalla sorgente
  - Se vale **0**, il costo per x è ancora in fase di **aggiornamento** (non definitivo)
- **Costo**
  - Rappresenta il **costo corrente minimo noto** per raggiungere x dalla sorgente s
  - Inizialmente, ha val. +∞ e per s a 0. Durante l'esecuzione, il val. viene aggiornato quando si trova un percorso migliore
- **Origine**
  - Indica il nodo "padre" o "predecessore" lungo il cammino minimo dalla sorgente s a x
  - Se non è stato ancora trovato un percorso per x oppure x non ha predecessori, questo val è impostato a -1

Quindi la terna associata al nodo x contiene tutte le info necessarie per sapere se il cammino minimo verso x è stato **determinato**, qual è il **costo** di tale **cammino** e **da quale nodo si giunge** a x lungo il percorso minimo.
Inizialmente la lista è inizializzata cosi:

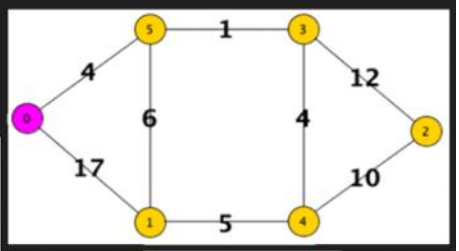
$$lista[x] = \begin{cases} (1, 0, s) & se\ x = s \\ (0, costo, s) & se\ (x, costo) \in G[s] \\ (0, +\infty, -1) & altrimenti \end{cases}$$

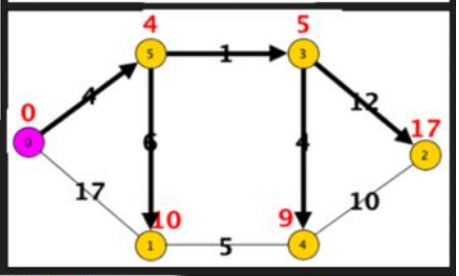
Vi sono poi una serie di iterazioni dove si eseguono i seguenti passaggi:

1. **Selezione del nodo con costo minimo non definitivo:** Si scorre l'intera **Lista** per individuare il nodo **x** che non è ancora definitivo (il cui flag è 0) e che ha il costo corrente minimo.
   Questo nodo è il candidato per il quale il cammino minimo dalla sorgente è attualmente noto
2. **Verifica di terminazione**: Se il costo minimo trovato è +∞, significa che non esistono altri nodi raggiungibili non ancora definitivi. In tal caso, il ciclo si interrompe
3. **Marcare il nodo x come definitivo**: Il nodo x selezionato viene aggiornato in **Lista** impostando il **flag a 1**, indicando che il suo costo definitivo è stato fissato e non verrà più modificato
4. **Aggiornamento dei vicini di x**: Per ogni nodo y adiacente a x, se y non è ancora definitivo e il nuovo costo ottenuto passando per x (cioè $costo_x + costo_{(x, y)}$) è **inferiore** al costo **attuale memorizzato per y**, allora si aggiorna la terna in **Lista[y]**
   Lista[y] = (0, $costo_x + costo_{(x, y)}$, x)

```python
def DijkstraLista(G, s):
    n = len(G)
    Lista = [ (0, float("inf"), -1) ]*n # inizializziamo la lista
    # 0 = nessun nodo definitivo, costo = infinito, origine = -1 (sconosciuta)
    Lista[s] = (1, 0, s) # il sorgente è il primo nodo definitivo
    for y, costo_arco in G[s]: # la lista di adiacenza ha (nodo_adiacente, costo_arco)
        # aggiorno gli adiacenti vicini a s, pero non sono ancora definitivi
        Lista[y] = (0, costo_arco, s)
    while True:
        # cerco il nodo non definitivo con costo minimo
        minimo, x = float("inf"), -1
        for i in range(n):
            if Lista[i][0] == 0 and Lista[i][1] < minimo:
                minimo, x = Lista[i][1], i
        if minimo == float("inf"):
            # se tutti i nodi sono definitivi
            break # esce
        # altrimenti rendi definitivo il nodo x
        definitivo, costo_x, origine = Lista[x]
        Lista[x] = (1, costo_x, origine)
        # aggiorna eventualmente i vicini di x non definitivi
        for y, costo_arco in G[x]:
            if Lista[y][0] == 0 and minimo + costo_arco < Lista[y][1]:
                # se non è definitivo e il cammino è migliore passando per x
                Lista[y] = (0, minimo+costo_arco, x) # aggiorna il nodo
    # Estrae da Lista i vettori delle distanze e dei padri
    D = [ costo for _, costo, _ in Lista]
    P = [ origine for _, _, origine in Lista]
    return D, P
```

```python
G=[
[(1, 17), (5, 4)],
[(0, 17), (4, 5), (5,6)],
[(3, 12),(4,10)],
[(2, 12),(4,4),(5,1)],
[(1,5),(2,10),(3,4)],
[(0,4),(1,6),(3,1)]
]
```






```python
>>> dijkstra(0,G)
([0, 10, 17, 5, 9, 4], [0, 5, 3, 5, 3, 0])
```

Complessità:

- Il costo delle istr. prima del while è **Θ(n)**
- Il **while** viene eseguito al più n-1 volte (ad ogni iter. un nuovo nodo viene selezionato e reso definitivo), nel while c'è:
  - Un **primo for** che viene iterato n volte → Θ(n)
  - Un **secondo for** che viene eseguito al più n volte (dipende dal num. di nodi adiacenti del nodo inserito nell'albero) → O(n)

Il costo del while e quello totale è **O(n²)**

Questa implementazione è ottima nei **grafi densi** dove m = Θ(n²)

Implementazione Algoritmo con Heap Minimo

Invece di scorrere ogni volta Lista per la ricerca del nodo con costo minimo, possiamo usare un **heap minimo** da cui estraiamo l'elem. minimo in **tempo logaritmico** nel num. di elem. presenti nell'**heap**

- **Mantenimento dell'Heap:** contenente la tripla (costo, x, y) dove:
  - **x** è un nodo già inserito nell'albero
  - **y** è un nodo candidato ad essere aggiunto
  - **costo** è la distanza minima per raggiungere y da x
- **Estrazione del nodo migliore:** ad ogni iter. si estrae dall'heap la tripla col minor costo in tempo logaritmico e gli modifichiamo il costo da assegnarli come distanza da s e il padre x a cui collegarlo
- **Aggiornamento dell'Heap:** ad ogni aggiunta di un nodo x all'albero, per ogni y adiacente di x, inseriamo nell'Heap una nuova tupla (**DistanzaAggiornata, x, y**). Ogni inserimento ha costo logaritmico, quindi il tempo complessivo rimane gestibile
- **Gestione duplicazioni:** nell'Heap possono esserci più entry dello stesso nodo con distanze diverse, però ad ogni nodo y estratto, esso è per forza il nodo con la distanza minima calcolata fino a quel punto, quindi ad ogni estrazione ignoriamo i nodi già aggiunti all'albero

```python
def DijkstraHeap(G, s):
    n = len(G)
    D = [float("inf")]*n # distanze inizialmente infinite
    P = [-1] * n # predecessori inizialmente sconosciuti
    D[s], P[s] = 0, s # inizializziamo la sorgente
    from heapq import heappush, heappop
    Heap = [] # Heap minimo
    for y, costo_arco in G[s]: # inizializziamo i nodi adiacenti alla sorgente
        heappush(Heap, (costo_arco, s, y))
    while Heap:
        costo, x, y = heappop(Heap) # estraiamo il nodo con distanza minore
        if P[y] == -1: # se non ha predecessore
            P[y] = x # imposta y come figlio di x
            D[y] = costo # imposta la distanza min. di y
            for v, costo_arco in G[y]: # esplora gli adiacenti di y
                if P[v] == -1: # se non è stato ancora aggiunto all'albero lo aggiungiamo
                    heappush(Heap, (D[y] + costo_arco, y, v))
    return D, P # Restituiamo le distanze e i predecessori
```

Complessità:

Nell'Heap possono esserci O(m) elem. → costo inserimento/estrazione sarà O(log(m)) = O(log(n²)) = O(2*log(n)) = O(log(n))

L'inizializzazione di D e P ha costo Θ(n) e l'inserimento dei vicini di s nell'Heap ha costo O(n*log(n)).

Abbiamo poi un while con all'interno un for:

- Ad ogni iter. del while si elimina un elem. dall'Heap e **eventualmente**, per mezzo del for annidato, si scorrono i nodi adiacenti di un nodo e, se non sono stati gia aggiunti, vengono aggiunti all'Heap. Ogni lista di adiacenza viene scorsa al più una volta, quindi in

Heap possono essere aggiunti al più O(m) elem. Quindi il num. di iterazioni del while è O(m)
- Il costo dell'estrazione dall'Heap, e quindi di ogni iter. del while (senza contare il for), ha costo O(log(n)) → costo tot. O(m*log(m))
- Ad ogni iter. del while scorro una lista di adiacenza diversa. Si iterano O(m) archi e ad ogni arco si esegue l'inserimento con costo O(log(n)) → costo tot. O(m*log(n))

Costo totale: Θ(n) + O(n*log(n)) + O(m) + O(m*log(n)) + O(m*log(n)) = O(n*log(n)) + O(m*log(n)) = O((n+m)*log(n))
Questa implementazione è preferita nel caso di **grafi sparsi**, con un costo di O(n*log(n))
Mentre andrebbe evitata nel caso di grafi densi, poiché il costo totale sarebbe $O(n^2*log(n))$

Esistono implementazioni più efficienti che utilizzano strutture dati più sofisticate, ad esempio usando gli **heap di Fibonacci** il costo scende a O(m + n*log(n))

Ricondurre Grafo Pesato in uno Non Pesato

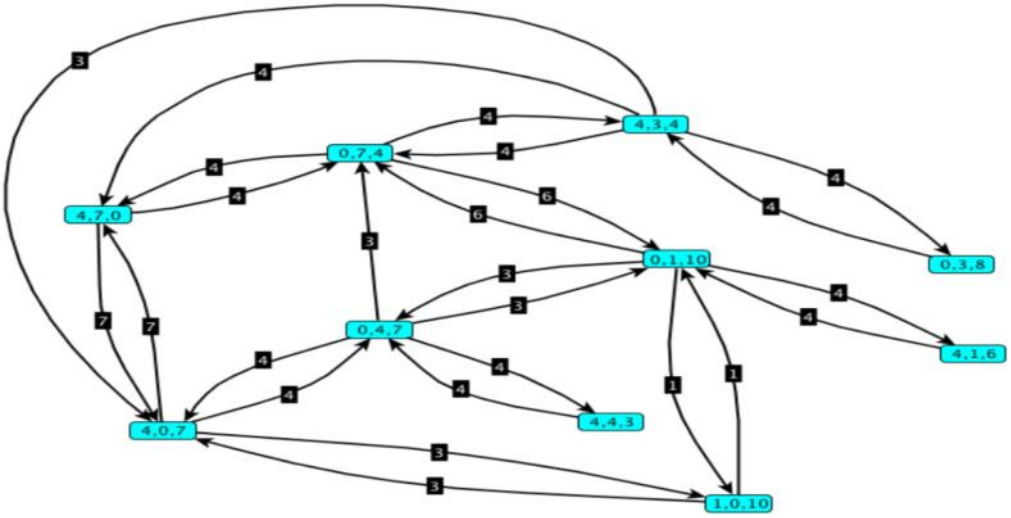
È possibile ricondurre il problema su un **grafo pesato in** uno con un **grafo non pesato** sostituendo il val. C di un arco tra x e y con C - 1 nuovi nodi nel cammino tra x e y.
L'approccio di ricondursi al problema dei cammini minimi in cui tutti gli archi hanno lo stesso valore 1 è possibile solo quando le etichette degli archi sono **interi** relativamente **piccoli**.

Abbiamo tre contenitori con capienza **4, 7 e 10 litri.** Inizialmente quelli da 4 e 7 sono pieni, mentre quello da 10 è vuoto.
Possiamo versare l'acqua tra i contenitori fermandoci solo quando il contenitore sorgente è vuoto o quello destinazione è pieno.
Problema: esiste una seq. di op. di versamento che terminano lasciando esattamente 2 litri nel contenitore da 4 o quello da 7?.
Si può modellare il problema con un **grafo diretto**:
- I nodi di G sono i possibili stati di riempimento dei contenitori. Quindi ogni nodo ha una configurazione **(a, b, c)** dove a, b e c sono rispettivamente il num. di litri dei contenitori 4, 7 e 10. (es possiamo avere nodi (4, 2, 5) o (1, 5, 5) ecc)
- Inserisco un arco dal nodo (a, b, c) al nodo (a', b', c') se è possibile passare dal primo al secondo stato (nodo) con un versamento lecito.

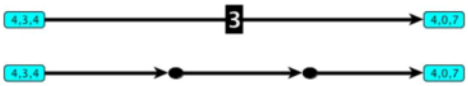
Per risolvere il problema basta chiedersi se esiste almeno un nodo (2, ?, ?) o (?, 2, ?) raggiungibile dal nodo (4, 7, 0)
Possiamo aggiungere un ulteriore **nodo pozzo (-1, -1, -1)** con archi entranti dai nodi (2, ?, ?) o (?, 2, ?) e chiederci se il nodo pozzo è raggiungibile da (4, 7, 0)

Un contenitore di capienza x, può contenere 0, 1, 2, …, x litri, quindi x + 1 stati diversi.
Quindi il num. dei nodi è **5 x 8 x 11 + nodo pozzo = 441 nodi tot del grafo** (anche nodi che non possono essere raggiunti da (4, 7, 0))
Per trovare una delle due configurazioni target basta usare una visita BFS per la ricerca dei cammini minimi con costo O(n + m)

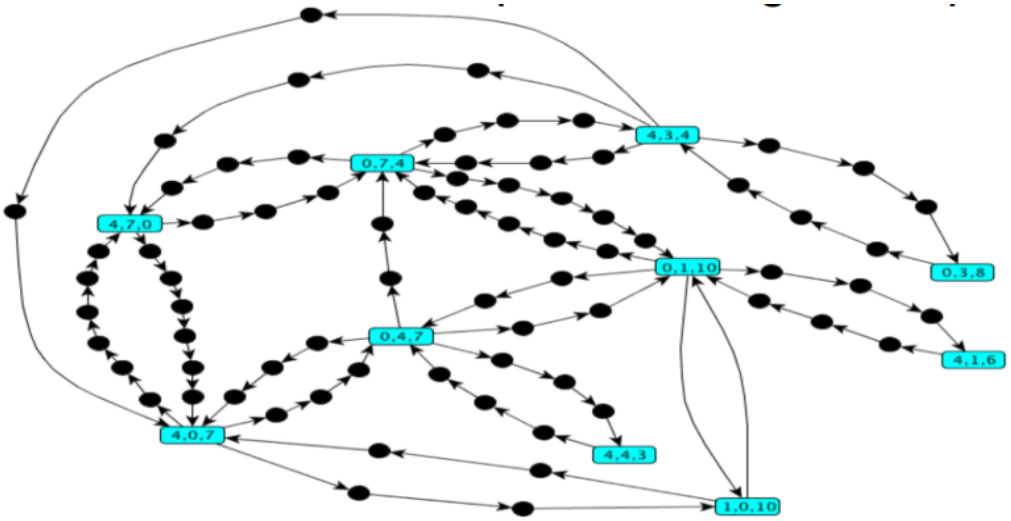
Consideriamo una **variante del problema:**
Una seq. di op. di versamento è **buona** se termina lasciando esattamente 2 litri nel contenitore da 4 o da 7. Inoltre una seq. **buona è parsimoniosa** se il tot. dei litri versati nella seq. è **minimo** rispetto a tutte le altre seq. buone.
Vogliamo trovare una **seq. sia buona sia parsimoniosa**.
Dovendo misurare il num. di litri d'acqua versati nei vari versamenti, **assegniamo un costo ad ogni arco**, indicante il num. di litri versati da un nodo all'altro. Ecco una piccola porzione del grafo che si ottiene:



Il problema ora diventa quello di trovare un cammino dal nodo (4, 7, 0) al nodo (-1, -1, -1) che **minimizza** la somma dei costi degli archi attraversati. Quindi ora gli archi invece di avere lo stesso valore, hanno valori differenti.
Idea: possiamo sostituire l'arco da x a y di costo C con un cammino con C - 1 nuovi nodi



La porzione di grafo precedente diventerebbe quindi:



Adesso abbiamo riportato il costo degli archi a 1 e possiamo effettuare una semplice visita **BFS** nel nuovo grafo. Cosi contando semplicemente gli archi nel cammino tra due nodi contiamo il num. tot di litri versati nelle mosse relative al proprio cammino.
L'algoritmo avrà complessità **O(n' + m')** dove n' e m' sono i nuovi e archi del nuovo grafo.
Siccome il peso degli archi non poteva superare 7, si ha quindi n' < 7n e m' < 7m.

Abbiamo ricondotto un problema di **cammini minimi su grafi pesati a quello di cammini minimi su grafi non pesati**.

UNION-FIND

domenica 30 marzo 2025 18:57

Union-Find noto anche come Disjoint Set Union (DSU) è una struttura dati per **gestire insiemi disgiunti.**
È usato per op. di **unione e ricerca** efficienti su **insiemi disgiunti.** Le tre op. fondamentali sono:

1. **Crea(S)**: restituisce una struttura dati Union-Find sull'insieme S dove ciascun elem. è un **insieme separato**
2. **Find(x, C)**: restituisce il nome dell'insieme della struttura dati C a cui appartiene l'elem. x
3. **Union(x, y, C)**: fonde la componente x con quella y e restituisce il nome della nuova componente

La gestione di insiemi disgiunti è utile se nell'evoluzione del grafo aggiungiamo archi nuovi. In questo caso gli insiemi disgiunti sono le componenti connesse di G.
L'op. **find(x)** consente di determinare la componente a cui appartiene x. Si può usare per vedere se due nodi **x e y** stanno nella **stessa componente** (find(x) == find(y)
Si **aggiunge un arco** (x, y) se x e y sono nella stessa componente connessa. Se cx = find(x) e cy = find(y) sono **distinti**, allora **Union(cx, cy, C)** può unire le componenti.

Come primo approccio per l'assegnazione di un **nome all'insieme** possiamo dare all'insieme il nome dell'**elem. massimo in esso contenuto**.

Implementazione con Vettore

Il modo più semplice per implementare questa struttura per n elem. è di mantenere il **vettore C delle n componenti**
Inizialmente ogni elem. è un **insieme distinto** quindi C[i] = i. Quando la **componente i** viene **fusa** con la **componente j**, se **i > j** allora le occorrenze di **j** in C vengono sostituite da i (se **i < j** accadrà il contrario)

```
Crea(G):
C=[ i for i in range(len(G))]
return C
```

Crea( ) costo $\Theta(n)$

```
def Find(u,C):
    return C[u]
```

Find( ) costo $\Theta(1)$

```
def Union (a,b,C):
    if a > b:
        for i in range(len(C)):
            if C[i]==b: C[i]=a
    else:
        for i in range(len(C)):
            if C[i]==a: C[i]=b
```

Union( ) costo $\Theta(n)$

Implementazione con Vettore dei Padri

Posso bilanciare i costi rendendo **meno costosa la UNION** ma **più costoso il FIND**
FIND: per sapere in che componente si trova un nodo risalgo semplicemente alla **sua radice → O(n)**
UNION: quando fondo due componenti **una diventa figlia dell'altra → O(1)**

```
def Crea(G):
    C=[ i for i in range(len(G))]
    return C
```

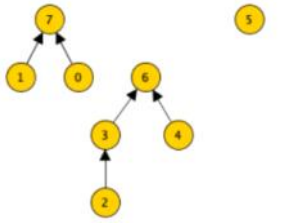
Crea( ) costo $\Theta(n)$

```
def Find(u,C):
    while u != C[u]:
        u = C[u]
    return u
```

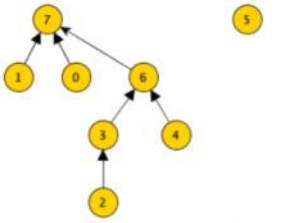
Find( ) costo $O(n)$

```
def Union (a,b,C):
    if a > b:
        C[b]=a
    else:
        C[a]=b
```

Union( ) costo $\Theta(1)$



$Comp = \boxed{7}\boxed{7}\boxed{3}\boxed{6}\boxed{6}\boxed{5}\boxed{6}\boxed{7}$



$Comp = \boxed{7}\boxed{7}\boxed{3}\boxed{6}\boxed{6}\boxed{5}\boxed{7}\boxed{7}$

Implementazione con Alberi Bilanciati

Insieme al vettore dei padri, mantengo gli **alberi bilanciati** in modo tale da diminuire il costo del FIND.
Durante l'**UNION**, scelgo il componente che contiene il maggior num. di elem. come **nuova radice**, cosi che almeno per la metà dei nodi delle due componenti la lunghezza del cammino non aumenta. In questo modo garantiamo la seguente proprietà:

Se un insieme ha **altezza h** allora l'insieme **contiene** almeno **2^h elem.**

Da ciò deduciamo che l'altezza delle componenti **non superera mai log(n)** (altrimenti avrei più di n nodi nella componente, il che è assurdo)
In questo modo possiamo ridurre il costo di FIND **da O(n) a O(log(n))**

In questa implementazione devo associare ai nodi radice anche il **num. di elem. che contiene la componente.**
Ogni elem. è una coppia **(x, num.)** dove **x** è il nome del componente e **num** è il num. di nodi nell'albero radicato in x

```
def Crea(G):
    C=[ (i,1) for i in range(len(G))]
    return C
```

Crea( ) costo $\Theta(n)$

```
def Find(u, C):
    while u != C[u]:
        u = C[u]
    return u
```

Find( ) costo $O(\log n)$

```
def Union (a, b, C):
    tota, totb = C[a][1], C[b][1]
    if tota >= totb:
        C[a]=(a, tota + totb)
        C[b]=(a, totb)
    else:
        C[a]=(b, tota)
        C[b]=(b, tota + totb)
```

Union( ) costo $O(1)$

Minimo Albero di Copertura

sabato 29 marzo 2025 17:37

L'**albero minimo di copertura** (spanning tree) è un albero che "**copre**" l'intero grafo pesato e in cui la somma dei costi dei suoi archi sia **minima**.
In questo albero non ci possono **mai essere cicli** (l'eliminazione di un arco del ciclo non farebbe perdere la connessione e diminuirebbe il costo della soluzione)
Esiste **un solo albero minimo di copertura** per ogni grafo pesato.

Algoritmo di Kruskal

L'**algoritmo di Kruskal** è uno ottimo algoritmo **greedy** che, dato un grafo connesso e pesato G, cerca **un** suo albero minimo di copertura.

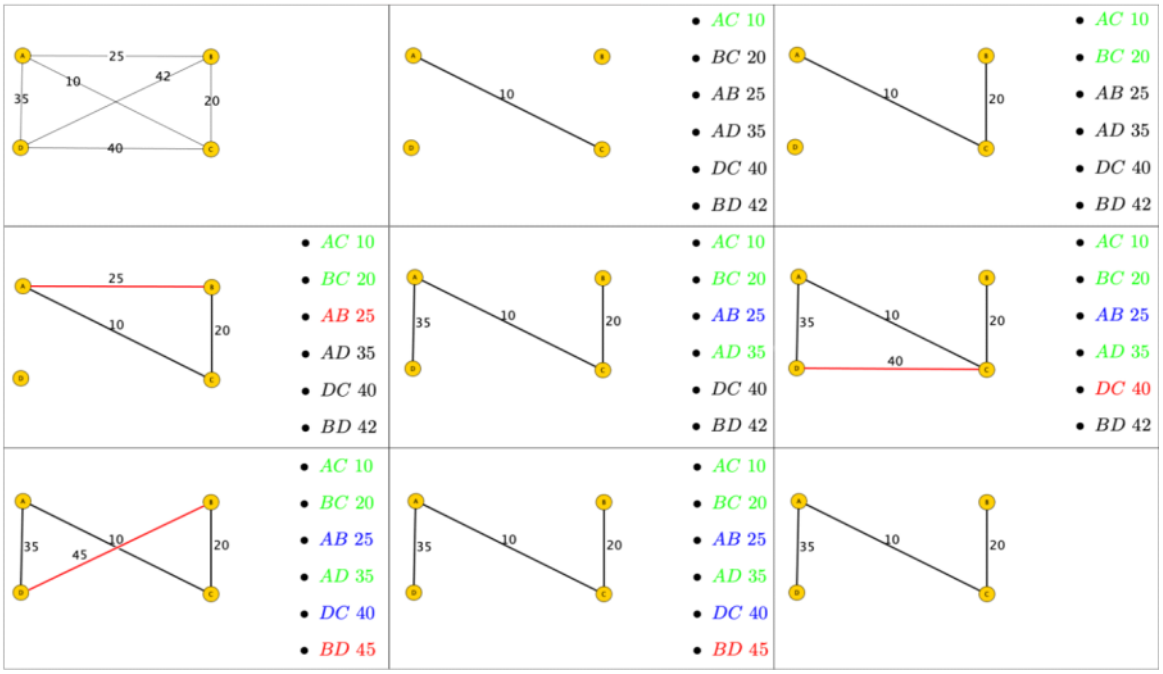
- Creiamo un **grafo T** che contiene **tutti i nodi** di G ma **nessun arco** di G
- Considera uno dopo l'altro gli archi di G in ordine di **costo crescente**
- Se l'arco forma un **ciclo** in T con archi già presi allora **non prenderlo**, altrimenti inseriscilo in T
- Al termine, restituisci T

Pseudo-codice:

```
kruskal(G):
    T = set()
    inizializza E con gli archi di G
    while E! = []:
        estrai da E un arco (x,y) di peso minimo
        if l'inserimento di (x,y) in T non crea ciclo con gli archi in T:
            inserisci arco (x,y) in T
    return T
```

L'algoritmo è **greedy** poiché:

- **La sequenza di decisioni irrevocabili:** decide per ogni arco di G se inserirlo o meno in T. Una volta deciso non ritorna più su questa decisione
- **Le decisioni vengono prese in base ad un criterio "locale":** se l'arco crea ciclo allora non si inserisce, altrimenti si inserisce siccome è il **meno costoso** a non creare cicli tra gli archi che restano da considerare.



Implementazione semplice

Idee:

- Con un **pre-processing ordino** gli archi in E cosicchè scorrendo la lista ottengo di volta in volta l'arco di costo min. in tempo O(1)
- Verifico che l'arco (x, y) **non formi ciclo** in T controllando se y è raggiungibile da x in T

```python
def connessi(x, y, T):
    # esegue una visita in T da x e ritorna True se c'è un cammino fino a y
    def DFS_conn(x, y, T, visitati):
        visitati[x] = 1 # visita il nodo corrente
        for v in T[x]: # per ogni nodo adiacente
            if v == y: # se abbiamo trovato il nodo da ricercare
                return True # ritorniamo True
            if not visitati[v]: # se non è stato visitato
                if DFS_conn(v, y, T, visitati): # continuiamo la visita
                    return True
        return False # se non l'abbiamo trovato ritorniamo False
    ###
    visitati = [0]*len(T) # inizializzo la lista visitati
    return DFS_conn(x, y, T, visitati)

def Kruskal(G):
    # creo la lista con gli archi pesati di G
    E = [(c, x, y) for x in range(len(G)) for y, c in G[x] if x<y]
    E.sort() # ordino gli archi in ordine crescente di peso
    T = [[] for _ in G] # inizializzo la lista vuota dell'albero di copertura
    for c, x, y in E: # per ogni nodo in E
        if not connessi(x, y, T): # se non c'è un cammino tra i due nodi x-y
            T[x].append(y) # al nodo x connetto y
            T[y].append(x) # e al nodo y connetto x
    return T # ritorno l'albero minimo di copertura
```

Costo:

- L'ordinamento esterno al for costa **O(m*log(m)) = O(m*log(n^2)) = O(m*log(n))**
- Il for viene iterato **m volte**:
  - Il controllo che l'arco (x, y) non crei ciclo in T con la procedura **connessi(x, y, T)** richiede il costo di una visita di un grafo aciclico quindi **O(n)**
- Il for richiede tempo **O(m*n)**

Costo totale: O(m*n)

Implementazione con UNION e FIND

Invece di pagare O(n) ad ogni iter. del for per controllare se esiste un ciclo, possiamo usare la struttura dati **UNION e FIND** che permette di testare se due nodi appartengono o meno alla stessa componente connessa.

La **UNION-FIND** è una struttura dati per la collezione C delle componenti connesse di un grafo di n nodi t.c sia possibile "efficientemente" effettuare le due op:

- **UNION(x, y, C)** fonde due componenti connesse **x e y in C** in tempo **O(1)**
- **FIND(x, C)** trova in C la componente connessa in cui si trova **x** in tempo **O(log(n))**

Kruskal

```python
def Kruskal_Union_Find(G):
    # creo la lista con gli archi pesati di G
    E = [(c, x, y) for x in range(len(G)) for y, c in G[x] if x<y]
    E.sort() # ordino gli archi in ordine crescente di peso
    T = [[] for _ in G] # inizializzo la lista vuota dell'albero di copertura
    C = Crea(T)
    for c, x, y in E:
        cx = Find(x, C) # prendo i rappresentati dei componenti di x e y
        cy = Find(y, C)
        if cx != cy: # se stanno in componenti distinti, x-y non crea ciclo
            T[x].append(y) # al nodo x connetto y
            T[y].append(x) # e al nodo y connetto x
            Union(cx, cy, C) # unisco i componenti di x e y
    return T # ritorno l'albero minimo di copertura
```

UNION-FIND

```python
def Crea(G): # crea l'insieme dei componenti
    C = [ (i, 1) for i in range(len(G))]
    # i è l'indice nell'insieme dei componenti
    # 1 è la "dimensione" del set
    return C
def Find(x, C): # restituisce il set a cui appartiene
    while x != C[x]: # finchè non troviamo il nodo
        x = C[x] # aggiorna x al suo padre
    return x
def Union(x, y, C): # unisce i set
    totx, toty = C[x][1], C[y][1] # prendo le grandezze dei componenti
    if totx >= toty: # se il componente di x è più grande o uguale a quello di y
        C[x] = (x, totx+toty) # incremento la dimensione di x con quella di y
        C[y] = (x, toty) # aggiorno il rappresentante di y a x
    else: # altrimenti se il componente di y è più grande
        C[y] = (y, totx+toty) # incremento la dimensione di y con quella di x
        C[x] = (y, toty) # aggiorno il rappresentante di x a y
```

Costo:

- L'ordinamento costa **O(m*log(n))**
- Il for viene iterato **m volte:**
  - L'estrazione dell'arco (x, y) di costo minimo da E richiede **Θ(1)**
  - Eseguire **FIND** costa **O(log(n))**
  - Eseguire **UNION** costa **Θ(1)** (nel for viene eseguito n-1 volte)
- Il for richiede tempo **O(m*log(n))**

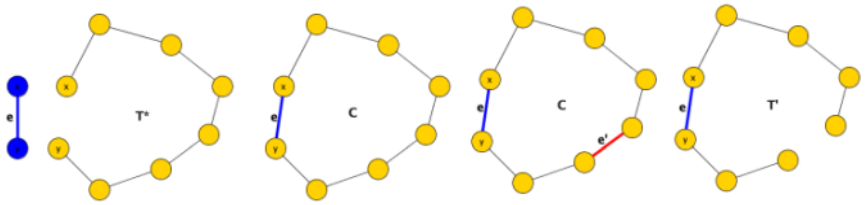
Costo totale: O(m*log(n))

Correttezza

Dobbiamo verificare che alla fine T è un albero di copertura e che non ci sono altri alberi con costo minore

- **Produce un albero di copertura. Proviamo per assurdo**
  Supponiamo che al termine in T ci sia più di una componente. Consideriamo una **A**, poiché G è connesso nel grafo c'è un arco (x, y) da un nodo x di **A** ad un nodo y di un'altra componente **B** ma l'arco (x, y) ad un certo punto è stato estratto da E e se non crea ciclo in T ora che l'algoritmo è terminato non lo creava neanche al momento in cui è stato estratto e quindi sarebbe stato aggiunto in T e non scartato
- **Non c'è un albero di copertura per G che costa meno dell'albero T ottenuto da Kruskal. Proviamo per assurdo**
  Tra tutti gli alberi di copertura di costo minimo per G prendiamo quello che differisce nel minor num. di archi da T e lo chiamiamo T*.
  Supponiamo per assurdo che T differsca da T*. Vedremo che questa assunzione porterebbe all'assurdo perché avrebbe come conseguenza l'esistenza di un altro albero di copertura di costo minimo per G che differisce da T in meno archi di T*
  Considera l'ordine e_1, e_2, … con cui gli archi sono stati presi in considerazione e sia **e** il primo arco preso che non compare in T*. Se inserisco e in T* si forma un un ciclo C che contiene almeno un arco **e'** che non appartiene a T (infatti tutti gli archi del ciclo C sono in T altrimenti **e** non sarebbe stato preso dall'algoritmo).
  Considera ora l'albero T' che ottendo da T* inserendo l'arco **e** ed eliminando **e'**. Il costo del nuovo albero T' (che è **costo(T') - costo(e') + costo(e)**) non può aumentare rispetto a quello di T* (perche **costo(e) ≤ costo(e')**, nota infatti che tra i due archi **e** ed **e'** che non creavano cicli, l'alg. ha considerato prima l'arco **e**)
  ma allora T' è un altro albero di copertura ottimo che differisce da T in meno archi di quanto faccia T* il che contraddice l'ipotesi che T* differisce da T nel min.

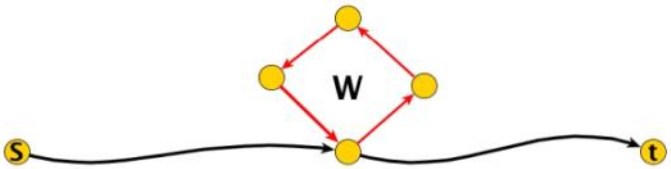
num. di archi



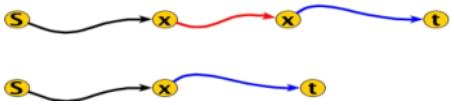
Cicli Negativi

lunedì 31 marzo 2025 13:40

Un **ciclo negativo** è un ciclo diretto in cui la **somma dei pesi** degli archi **è negativa.**
Il problema di un ciclo negativo è che **ripassando** più volte attraverso il ciclo, possiamo abbassare continuamente il costo del cammino tra due nodi (anche all'infinito)
Quindi **non esiste un cammino di costo minimo** tra i due nodi.



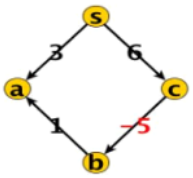
Proprietà: se G **non contiene cicli negativi**, allora per ogni nodo **t** raggiungibile dalla sorgente **s** esiste un **cammino di costo minimo** che attraversa al più **n-1 archi**
Se un cammino avesse più di n-1 archi, allora almeno un nodo verrebbe ripetuto, formando un ciclo. Poiché G non ha cicli negativi, rimuovere eventuali cicli dal cammino **non aumenta** il suo costo complessivo. Quindi esiste sempre un cammino ottimale di lunghezza n-1. Quindi il costo minimo può essere calcolato considerando solo **cammini di lunghezza n-1 e non oltre**.



L'arco rosso indica un ciclo che ci riporta a x

Problema: dato un grafo diretto e pesato in cui i pesi degli archi possono essere anche negativi **ma che non contiene cicli negativi**, e fissato un nodo s, vogliamo determinare il costo min. dei cammini che conducono da s a tutti gli altri nodi del grafo
In presenza di archi con **peso negativo Dijkstra** non funzionerebbe, poiché assume che, una volta visitato un nodo con costo min., tale costo non possa essere più migliorato. Questa assunzione però non è valida se esistono archi di peso negativo.

Ad esempio in questo grafo:



L'algoritmo di Dijkstra sceglie prima il cammino dell'arco **(s, a)** di **costo 3**, però il cammino da **s** ad **a** che passa per **c** e **b** ha **costo 6 - 5 + 1 = 2** e non viene visto da Dijkstra.

Algoritmo di Bellman-Ford

L'algoritmo di **Bellman-Ford** calcola i cammini minimi su un grafo diretto pesato con anche pesi negativi.
In base alla proprietà precedente sui cammini minimi, possiamo restringere i cammini minimi di lunghezza max n-1.
Definiamo la tabella di dimensione **nxn:**

T[i][j] = costo cammino min. da s al nodo j di lunghezza al più i (archi attraversati)

La soluzione del problema sarà data dagli **n val.** che troviamo nell'**ultima riga:**

T[n-1][0], T[n-1][1], ..., T[n-1][n-1]

Infatti il costo min. per andare **da s al nodo t sarà T[n-1][t]**

I val nella prima riga sono tutti **+∞** tranne per T[0][s] = 0 (e resterà 0 per tutta la colonna di s, quindi T[i][s] = 0 per ogni i > 0)
Bisogna ora definire la **regola** che permette di calcolare i val. delle celle T[i][j] con j ≠ s della riga i > 0 in funzione delle celle già calcolate nella **riga i-1**
Distinguiamo due casi a seconda che il cammino al più i **da s a j** abbia lungh. "**< i" o "= i"**:

1. **T[i][j] = T[i-1][j]:** Il costo non è cambiato → si è gia trovato il miglior cammino
2. **T[i][j] =** $\min_{(x,j)\in E}\big(T[i-1][x] + costo(x,j)\big)$**:** esiste un cammino minimo di lunghezza al più i - 1 ad un nodo x e poi un arco che da x mi porta a j.
   - Quindi da s arriviamo ad x con al più i-1 archi e costo T[i-1][x]
   - Poi attraversiamo l'arco (x, j) che ha un certo costo
   - Visto che x può essere qualsiasi nodo che arriva a j, dobbiamo prendere il min. su tutti i nodi x possibili

Non sapendo in quale dei due casi siamo la formula giusta è:

$$T[i][j] = \min\Big(T[i-1][j], \min_{(x,j)\in E}\big(T[i-1][x] + costo(x,j)\big)\Big)$$

Cosi per ogni cella si considera il miglior costo possibile, o mantenendo la soluzione già trovata (caso 1) o aggiornandola con un arco in più a costo minore (caso 2)
Quindi le celle della tabella possono essere riempite per righe in base a questa regola

$$T[i][j] = \begin{cases} 0 & se\ j = s \\ +\infty & se\ i = 0 \\ \min\Big(T[i-1][j], \min_{(x,j)\in E}\big(T[i-1][x] + costo(x,j)\big)\Big) & altrimenti \end{cases}$$

NOTA: per un implementazione efficiente, poiché bisogna più volte conoscere gli archi entranti in j, conviene precalcolare il **grafo trasposto GT** di G.
Cosi che in **GT[j]** avremo l'elenco di tutti i nodi **x** t.c in G **esiste un arco da x a j**. Cosi possiamo accedere più velocemente agli archi entranti in un nodo

```python
def TraspostoPesato(G):
    n = len(G)
    GT = [ [] for _ in G ] # inizializza la lista vuota
    for i in range(n): # per ogni nodo
        for j, costo in G[i]: # per ogni adiacente e il costo dell'arco (i->j)
            GT[j].append( (i, costo) ) # aggiungiamo l'arco invertito (j->i) in GT
            # ad ogni adiacente inseriamo il nodo padre e il costo dell'arco
    return GT
```

Costo:

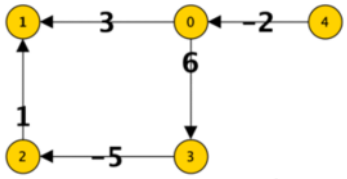
```python
        for j, costo in G[i]: # per ogni adiacente e il costo dell'arco (i->j)
            GT[j].append( (i, costo) ) # aggiungiamo l'arco invertito (j->i) in GT
            # ad ogni adiacente inseriamo il nodo padre e il costo dell'arco
    return GT

def CostoCammini(G, s):
    n = len(G)
    T = [[float("inf")]*n for _ in range(n)] # inizializza tabella nxn a infinito
    T[0][s] = 0 # il costo per raggiungere s con 0 archi è 0
    GT = TraspostoPesato(G) # crea il trasposto pesato di G
    for i in range(1, n): # per ogni lunghezza da 1 a n-1
        for j in range(n): # per ogni nodo j
            T[i][j] = T[i-1][j] # copiamo il costo della riga precedente (1° caso)
            if j != s: # per ogni nodo diverso dalla sorgente
                for x, costo in GT[j]: # per ogni arco entrante in j (x->j in GT)
                    T[i][j] = min(T[i][j], T[i-1][x]+costo) # aggiorna il costo minimo
                    # min tra quello attuale e cammino che arriva da x più costo arco
    return T[n-1] # ritorna i costi minimi per cammini di lunghezza al più n-1
```

Costo:

- L'inizializzazione di T richiede tempo $\Theta(n^2)$
- La costruzione del **grafo trasposto** GT richiede tempo O(n + m)
- Per i tre for annidati è ovvio il limite superiore $O(n^3)$, però un analisi più attenta può dare un limite più stretto:
  - Il primo for ovviamente richiede tempo O(n)
  - Gli ultimi due for hanno costo tot O(n+m)
    Quindi il tempo richiesto per scorrere le liste di adiacenza del grafo GT con lunghezza tot. m

Costo complessivo: $n*O(n+m) = O(n^2 + n*m)$

- Se il grafo è **sparso** (m = O(n)), il costo diventa $O(n^2)$
- Se il grafo è **denso** (m = $O(n^2)$), il costo diventa $O(n^3)$




```
G=[
    [(1,3),(3,6)],
    [],
    [(1,1)],
    [(2,-5)],
    [(0,-2)]
]
```

i \ j	0	1	2	3	4
0	0	∞	∞	∞	∞
1	0	3	∞	6	∞
2	0	3	1	6	∞
3	0	2	1	6	∞
4	0	2	1	6	∞

```
>>> costo_cammini(G,0)
[0, 2, 1, 6, inf]
```

i \ j	0	1	2	3	4
0	∞	∞	∞	∞	0
1	-2	∞	∞	∞	0
2	-2	1	∞	4	0
3	-2	1	-1	4	0
4	-2	0	-1	4	0

```
>>> costo_cammini(G,4)
[-2, 0, -1, 4, 0]
```

Ritrovare i Cammini

Oltre al costo, con la tabella T possiamo calcolare l'albero P dei **cammini minimi**. Per ogni nodo j manteniamo il suo **predecessore** cioè il nodo x che precede j.
Il val. di P[j] andrà aggiornato ogni volta che il val. di T[i][j] cambia (diminuisce) in quanto abbiamo trovato un cammino migliore

```python
def CostoCamminiPadre(G, s):
    n = len(G)
    T = [[float("inf")]*n for _ in range(n)] # inizializza tabella nxn a infinito
    T[0][s] = 0 # il costo per raggiungere s con 0 archi è 0
    GT = TraspostoPesato(G) # crea il trasposto pesato di G
    P = [-1]*n # inizializziamo la lista dei padri
    P[s] = s # il padre della radice è la radice stessa
    for i in range(1, n): # per ogni lunghezza da 1 a n-1
        for j in range(n): # per ogni nodo j
            T[i][j] = T[i-1][j] # copiamo il costo della riga precedente (1° caso)
            if j != s: # per ogni nodo diverso dalla sorgente
                for x, costo in GT[j]: # per ogni arco entrante in j (x->j in GT)
                    T[i][j] = min(T[i][j], T[i-1][x]+costo) # aggiorna il costo minimo
                    # min tra quello attuale e cammino che arriva da x più costo arco
                    P[j] = x # aggiorniamo il padre del nodo
    return T[n-1], P # ritorna i costi minimi per cammini di lunghezza al più n-1
```

In questa implementazione avremmo:

- **T[n-1][j] ≠ +∞** indica che **j è raggiungibile a partire da s**
  Allora P[j] avrà il nodo che **precede** j nel cammino minimo
- **T[n-1][j] = +∞** indica che **j non è raggiungibile a partire da s**
  Allora P[j] conterrà **-1**

Ottimizzazioni:

- **Il contenuto della cella in riga k dipende da quello in riga k - 1**, quindi:
  - Se la riga k è **identica** alla riga **k - 1** anche le righe successive non varieranno, allora si può **terminare l'algoritmo** direttamente.
    Questo accorgimento non modifica la complessità asintotica ma in pratica può contare molto
  - Non serve memorizzare l'intera tabella T, bastano le **ultime due righe**. Perciò l'algoritmo si può modificare per usare memoria O(n) invece di $O(n^2)$

Si può modificare l'algoritmo per scoprire per sapere se il grafo **contiene cicli negativi:**

- Calcola **una riga in più** (la riga n) con il costo dei cammini minimi di lunghezza al più n
- Se le righe **n ed n-1** risultano **uguali** allora nel grafo **non ci sono cicli negativi** (altrimenti si abbasserebbe il costo nella riga n poiché si passa nel ciclo negativo)

Ottimizzazione

giovedì 8 maggio 2025 13:18

Un **problema di ottimizzazione** è un problema il cui obiettivo è trovare la soluzione migliore possibile tra un insieme di sol. ammissibili e ogni sol. ammissibile ha un val. associato che può essere un "costo" o un "beneficio"
A seconda del problema, l'obiettivo può essere minimizzare quel val. (es: ridurre i costi) o massimizzarlo (es: ottenere massimo guadagno)
Abbiamo quindi **problemi di minimizzazione e massimizzazione**

Esempi:

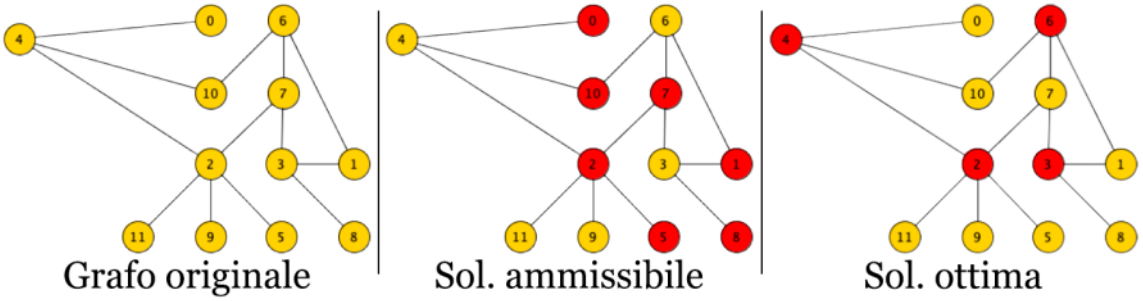
- Nel **problema dell'albero minimo di copertura** su grafo pesato, una sol. ammissibile è un albero di copertura (un sottoinsieme di archi che connette tutti i nodi senza formare cicli).
  La soluzione ottima è l'albero di copertura che ha il **costo minimo,** cioè la somma dei pesi degli archi è la più bassa possibile tra tutte le sol. ammissibili
  In questo caso si può trovare una sol. ottima in **tempo polinomiale (algoritmo di Kruskal)**

- Nella **ricerca del cammino minimo** tra due nodi a e b in un grafo pesato, una sol. ammissibile è un cammino tra a e b attraverso una seq. di archi del grafo.
  La soluzione ottima è il cammino che ha il **costo minimo**, cioè la somma dei pesi degli archi del cammino è la più bassa possibile.
  Anche in questo caso, si può trovare una sol. ottima in **tempo polinomiale (algoritmo di Dijkstra)**

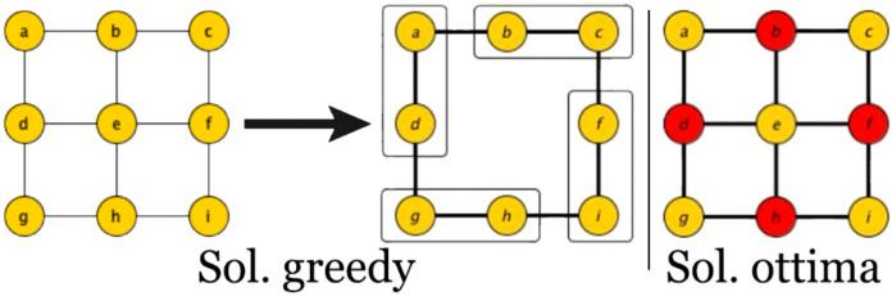
Però in molti casi è difficile trovare una soluzione ottima nei problemi di ottimizzazione, è può essere un problema **NP-difficile** (tanto difficile quanto i problemi in NP (non garantito che la sol. sia in tempo polinomiale)), dove la **complessità cresce esponenzialmente con la dim. del problema**.
Sebbene trovare una sol. ammissibile può essere fatto in tempo polinomiale, trovare la sol. ottima richiede spesso algoritmi più complessi e costosi.

Algoritmi di Approssimazione

Dato un grafo non diretto G, una sua **copertura tramite nodi** è un sottoinsieme S dei suoi nodi t.c. tutti gli archi di G hanno almeno un estremo in S.
Problema di ottimizzazione della copertura tramite nodi: trovare una copertura tramite nodi **con cardinalità minima**



Una semplice **strategia greedy** è: finche ci sono archi non coperti, inserisci in S il nodo che copre il **massimo** num. di archi ancora scoperti
L'algoritmo però non è ottimo:
Sul grafo G al primo passo verrebbe inserito in S il **nodo e** (l'unico che copre 4 archi) dopodichè tutti i nodi restanti possono coprire lo stesso num. di archi.
Nei passi successivi almeno un nodo per ogni coppia evidenziata nella figura in basso viene scelto. **La soluzione prodotta consiste di 5 nodi**
La soluzione ottima però consiste in 4 nodi



Come questo problema di ottimizzazione qui, ne esistono moltissimi che sono computazionalmente difficili e non si conoscono algoritmi efficienti (solo esponenziali)
In questi casi è soddisfacente anche solo ottenere una **sol. ammissibile** che sia "vicina" ad una sol. ottima (più vicina meglio è)

Fra gli algoritmi che non trovano sempre una sol. ammissibile ottima bisogna distinguere due categorie differenti: **Algoritmi di approssimazione ed Euristiche**

Algoritmi di Approssimazione	Euristiche
Algoritmi per cui si **dimostra** che la sol. ammissibile ha una certa **"vicinanza" alla sol. ottima**. Quindi è garantito che la sol. prodotta **approssima** entro un certo grado una sol. ottima	Algoritmi per cui **non si riesce** a **dimostrare** che la sol. ammissibile prodotta ha una certa **vicinanza alla sol. ottima**. Però, sperimentalmente sembrano comportarsi bene. È l'ultima possibilità quando non si trovano algoritmi corretti efficienti né algoritmi di approssimazione efficienti che garantiscano un buon grado di approssimazione

Per molti algoritmi computazionalmente difficili, non si conoscono neanche algoritmi di approssimazione, quindi gli algoritmi euristici sono la classe più ampia.

In un algoritmo di approssimazione bisogna specificare cosa si intende per "approssimazione entro un certo grado".

Problemi di Minimizzazione

Nei **problemi di minimizzazione** (es problema della copertura tramite vertici) ad ogni sol. ammissibile è associato un costo e cerchiamo una sol. col costo minimo
Il modo usuale per misurare il grado di approssimazione è il **rapporto** al **caso pessimo tra costo della sol. prodotta** dall'algoritmo **e costo della sol. ottima:**
Più formalmente, si dice che l'algoritmo A approssima il problema di minimizzazione P entro un **fattore di approssimazione ρ se per ogni istanza I vale**:

$$\frac{A(I)}{OTT(I)} \leq \rho$$

dove, **A(I)** è il costo della sol. prodotta da A per quell'istanza e **OTT(I)** è il costo di una sol. ottima per quell'istanza

(per i problemi di massimizzazione vale il rapporto inverso $\frac{OTT(I)}{A(I)}$)

Nota che trattandosi di un problema di minimizzazione, risulta sempre A(I) ≥ OTT(I) (non puoi superare la sol. ottima), quindi ρ è sempre ≥ 1:

- Se A approssima P con **fattore 1** allora A è corretto per P perché trova **sempre una sol. ottima**
- Se A approssima P con **fattore 2** allora A trova sempre una sol. di **costo al più doppio di quello della sol. ottima**
- Ovviamente più il rapporto di approssimazione è vicino ad 1 tanto più l'algoritmo di approssimazione è buono

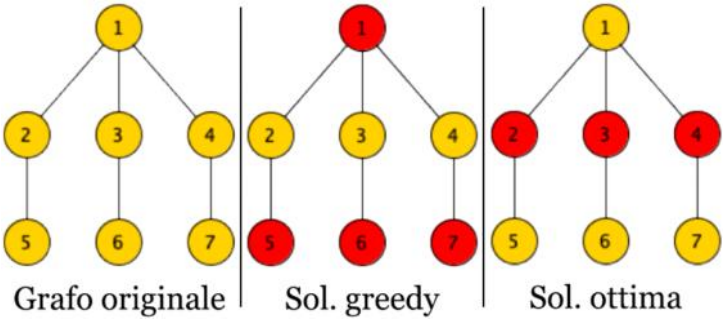
Quindi più ρ è vicino ad 1, più l'algoritmo di approssimazione è buono

- Se esiste un **ρ costante** (es: 2, 10, 100, ecc) che **vale per tutte le possibili istanze**, allora l'algoritmo garantisce "sempre" di non superare **ρ volte** la sol. ottima
- Se **non esiste un ρ costante** per ogni istanza, allora l'algoritmo **non è in grado di garantire un fattore di appr. costante** ed è quindi "**cattivo**"
  - Si dimostra che per ogni costante R possiamo costruire un grafo t.c. $\frac{A(I)}{OTT(I)} > R$.
    Quindi non esiste un limite costante sopra al quale la prestazione di A non salga

Esempio:

Proviamo a valutare il rapporto d'approssimazione dell'algoritmo greedy visto nella copertura tramite nodi:

- L'algoritmo greedy che abbiamo visto non è corretto
- Abbiamo trovato un istanza per cui l'algoritmo (A) produce una sol. con 5 nodi, mentre l'algoritmo ottimo (OTT) ha 4 nodi
- Deduciamo che il fattore di approssimazione dell'algoritmo è **almeno 5/4 > 1.**
  Ma potrebbe essere peggiore



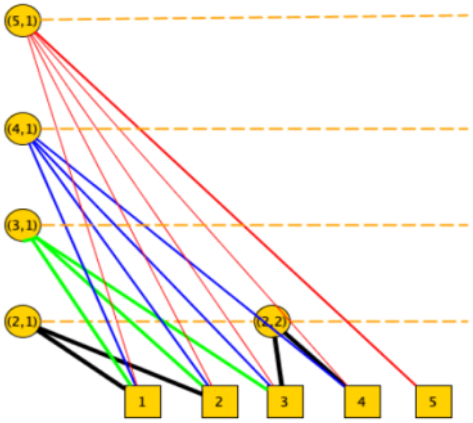
Per ogni costante R si possono esibire grafi su cui A sbaglia di un fattore superiore ad R. Quindi l'algoritmo greedy non garantisce **nessun fattore di approssimazione costante**

PROVA: Dimostrazione che per **ogni intero l** si può costruire un grafo G_l sui cui l'algoritmo greedy avrà **rapporto di approssimazione Ω(log(l))**

Il grafo G_l è strutturato in **l livelli** che vanno da 1 ad l e gli archi del grafo collegano i nodi a liv. i > 1 con nodi al liv. 1.

Più precisamente, a liv. i sono presenti $\left\lfloor \frac{l}{i} \right\rfloor$ nodi, ciascuno ha grado i e gli $i * \left\lfloor \frac{l}{i} \right\rfloor$ archi vanno a nodi distinti del liv. 1

Esempio grafo G_5:



Sul grafo G_l l'euristica funziona cosi:

- Il primo nodo selezionato è il nodo al liv. l di grado l (i nodi rimasti hanno grado ≤ l-1)
- Verranno poi selezionati tutti i nodi al liv. l-1 di grado l-1 (i nodi rimasti hanno grado ≤ l-2)
- …
- Verranno infine selezionati tutti i nodi a liv 2 di grado 2 (i nodi rimasti al liv. hanno grado 1)

Per il costo della sol. prodotta dall'euristica si ha:

$$c(SOL) = \sum_{i=2}^{l} \left\lfloor \frac{l}{i} \right\rfloor \ (siccome\ \lfloor x \rfloor \geq x - 1)$$

serie armonica $\sum_{i=2}^{l} \frac{1}{i} \geq \int_2^{l+1} \frac{1}{x} dx$

$$\geq \sum_{i=2}^{l} \left(\frac{l}{i} - 1 \right) = l * \sum_{i=2}^{l} \frac{1}{i} - \sum_{i=2}^{l} 1 = l * \sum_{i=2}^{l} \frac{1}{i} - (l-1) \ \ (stiamo\ sommando\ 1\ esattamente\ l-1\ volte\ (da\ 2\ a\ l)$$

$$\geq l * \int_2^{l+1} \frac{1}{x} dx - l + 1 = l * \big(ln(l+1) - ln(2) \big) - l + 1$$

$$\geq \ \Omega(l * log(l))$$

Una possibile sol. al problema è di **selezionare** gli **l nodi a liv. 1** (per costruzione tutti gli archi incidono su quegli l nodi) (questa è anche la sol. ottima).

Possiamo quindi dire che **c(SOL*) = O(l)**

Abbiamo quindi dimostrato che il rapp. di appr. dell'euristica è **almeno** $\frac{c(SOL)}{c(SOL*)} = \frac{\Omega(l*log(l))}{O(l)} = \Omega(log(\ l))$

Poiché log(l) **cresce indefinitamente** al crescere di l, il rapporto **non** è **limitato** da una costante fissa, quindi non garantisce un fattore di appr. costante

Ovviamente il fatto d'aver dimostrato che per un problema un certo algoritmo d'appr. ha un **cattivo rapp. di appr.** non impedisce che possano esistere altri algoritmi di appr. con un fattore d'appr. costante

Consideriamo il seguente algoritmo greedy per la copertura dei nodi

```python
def copertura1(G):
    Inizializza la lista E con gli archi di G
    S = []
    for nodo (x, y) in E: # per ogni arco del grafo
        if x not in S and y not in S: # se l'arco non è coperto da x e y
            # aggiungiamo i due nodi per coprire l'arco
            S.append(x)
            S.append(y)
    return S
```

```python
def copertura1(G):
    # Inizializza la lista E con gli archi di G
    n = len(G)
    E = [(x, y) for x in range(n) for y in G[x] if x < y]
    presi = [0]*n # per efficienza: presi[i] = 1 se nodo i è in S, 0 altrimenti
    S = []
    for x, y in E:
        # per ogni arco (x, y) in E
         if presi[x] == presi[y] == 0: # se l'arco non è coperto da x e y (x e y non in S)
            # aggiungiamo i due nodi per coprire l'arco
            S.append(x)
            S.append(y)
            presi[x] = presi[y] = 1
    return S
```

- L'algoritmo produce **ovviamente una copertura** (ogni arco verrà esaminato e se risulta non coperto verrà coperto da entrambi i lati
- La copertura prodotta non è detto sia minima, l'algoritmo ha un **rapporto d'approssimazione almeno 2** come dimostra il grafo con due soli nodi ed un arco

Dimostriamo che il **rapp. d'appr è limitato da 2** (non sono noti algoritmi con rapp inferiore a 2)

PROVA:

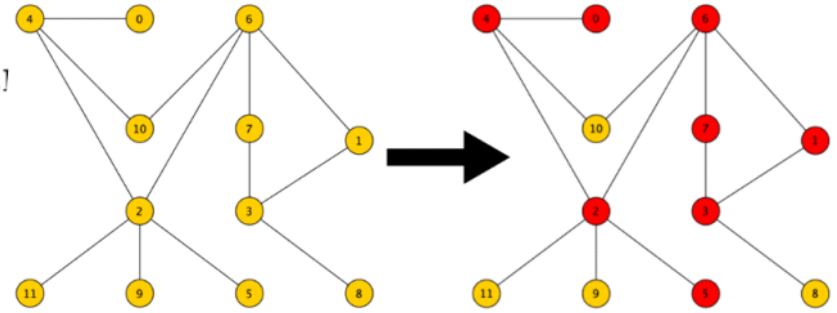
Siano $e_1, e_2, \ldots, e_k$ gli archi di G che vengono trovati **non coperti** durante l'esec. dell'algoritmo greedy

1) Per come funziona l'alg. deduciamo che A(I) = 2k (perche ogni volta che un arco non è coperto, aggiunge 2 nodi a S)
   I k archi non coperti sono tra loro disgiunti (i due estremi di ciascuno di questi archi vengono incontrati per la prima volta quando viene esaminato l'arco)
   quindi in una qualunque delle sol. ottime almeno un estremo di ciascuno di questi k archi deve essere presente
2) Ne deduciamo che k $\leq$ OTT(I)

Quindi ricaviamo che A(I) = 2k $\leq$ 2*OTT(I) da cui segue $\frac{A(I)}{OTT(I)} \leq 2$

La complessità dell'algoritmo è **O(n+m)** (per l'inizializzazione di E)

```
>>>G=[
     [4],[3,6],[4,5,7,9,11]
     [1,7,8],[0,2,10],[2],
     [1,7,10],[3,6],[3],
     [2],[4,6],[2]
     ]
>>> copertura1(G)
[0, 4, 1, 3, 2, 5, 6, 7]
```



Algoritmi Greedy

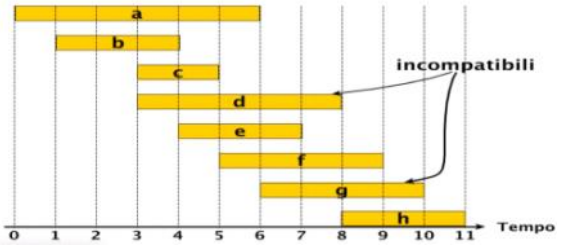
giovedì 8 maggio 2025 14:39

Selezione di Attività

Per illustrare il progetto e l'analisi di un algoritmo greedy consideriamo il problema della **selezione di attività**
Abbiamo una lista di n attività da eseguire:

- Ciascun attività ha una coppia con **tempo inizio e tempo fine**
- Due attività sono **compatibili** se non si sovrappongono

Vogliamo trovare un sottoinsieme di attività compatibili di massima cardinalità
Es: n = 8

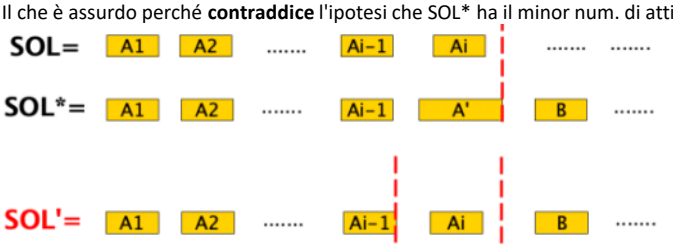


La soluzione ottima è {b, e, h}

Usando il paradigma **greedy** dobbiamo trovare una regola semplice che permette di effettuare sempre la scelta giusta.
Delle potenziali regole sono prendere l'attività compatibile che: inizia prima, dura meno o che ha meno conflitti con le rimanenti:
Però la regola giusta è prendere l'attività compatibile che **finisce prima**
PROVA PER ASSURDO:
Supponiamo **per assurdo** che la sol greedy SOL non sia ottima e prendiamo la sol ottima SOL* che ha il minor num. di attività diverse da SOL (più attività in comune)
Sia Ai la prima attività scelta da SOL e non da SOL* (esiste perche se SOL* prendeva tutte le attività di SOL poi non poteva prenderne di più).
Allora deve esistere in SOL* un'attività A' in conflitto con Ai, altrimenti si potrebbe aggiungere a SOL* migliorandola.
Poiché Ai è stata scelta dal greedy, deve terminare prima di A'. Quindi si può **sostituire A' con Ai in SOL*** ottenendo una nuova **SOL'** che:

- Ha lo stesso num di attività di SOL*
- Le attività restano compatibili
- Ha un num. minore di SOL* di attività diverse da SOL

Il che è assurdo perché **contraddice** l'ipotesi che SOL* ha il minor num. di attività diverse da SOL



Idee implementazione:

- Siccome cercare l'attività che finisce prima richiede tempo O(n), ordiniamo le attività per tempo di fine crescente, pagando Θ(n*log(n)) per l'ordinamento ma Θ(1) per l'estrazione
- Per controllare se l'attività non va in conflitto con altre già prese, manteniamo una var. col tempo di fine dell'ultima attività presa.

```
def selezione_att(lista):
    lista.sort(key=lambda x: x[1]) # sort per tempo di fine crescente
    libero = 0
    sol = []
    for inizio, fine in lista: # per ogni attività
        if inizio > libero: # se inizia dopo la fine dell'ultima presa
            sol.append((inizio, fine)) # aggiungiamo l'attività
            libero = fine # aggiorniamo con la fine dell'attività presa
    return sol
```

Complessità:

- Ordinare la lista delle attività costa Θ(n*log(n))
- Il for viene eseguito n volte e il costo di ogni iter è Θ(1)

Costo totale: Θ(n*log(n))

Assegnazione di Attività

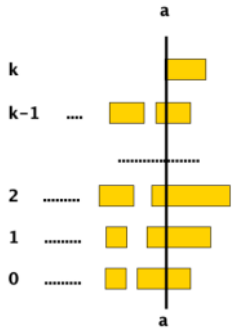
Consideriamo il nuovo problema di **assegnazione di attività**, dove le attività vanno tutte eseguite in seriale e vogliamo **assegnarle nel minor num. di aule**

$$lista = [(1,4),\ (1,6),\ (7,8),\ (5,10)]$$

Un possibile algoritmo greedy si basa sull'idea di occupare aule finchè ci sono attività da assegnare e ad ogni aula, una volta inaugurata, assegnare il maggior num. di attività non ancora assegnate che è in grado di contenere. Però cosi prende prima le attività compatibili e non quelle migliori che occuperebbero più spazio nell'aula.
Un algoritmo alternativo estrae invece l'attività che **inizia prima** e se nella soluzione c'è un aula in cui quell'attività è compatibile, allora la assegnamo a quell'aula, altrimenti la aggiungiamo ad una nuova aula.

$$lista = [(1,4),\ (1,6),\ (7,8),\ (5,10)]$$

CORRETTEZZA:
Nella soluzione vengono usate k aule, poiché nella lista k attività sono incompatibili a coppie.
Se (a, b) è l'attività che ha creato l'aula k, significa che nelle altre k-1 aule era impossibile eseguire (a, b) però per il criterio di scelta greedy posso dire che le altre aule erano **occupate nell'istante a** (poiché le attività nelle altre aule iniziavano prima del tempo a e non erano ancora finite) quindi a due a due tutte queste k attività sono incompatibili.



Idee implementazione:

- Per prendere l'attività che inizia prima eseguo prima un sort per tempo d'inizio crescente
- Per trovare se un'aula nella soluzione può eseguire l'attività, uso un **heap minimo** in cui metto le coppie (libera, i) dove libera indica il tempo in cui si libera l'aula i. Cosi verranno ordinate per quando si liberano prima e basta controllare in H[0][0] quale si libera prima
    - Se nell'aula in H[0][0] si può eseguire l'attività allora dovrò assegnargliela e aggiornare il val. libera della coppia nell'heap.
    - Se invece non si può assegnare l'attività, non si potrà fare in nessun altra aula, quindi devo assegnarla ad una nuova aula che verrà aggiunta all'heap

```python
def AssegnazioneAule(lista):
    from heapq import heappop, heappush
    Sol = [[]] # matrice per le aule
    H = [(0, 0)] # inizializzo l'heap
    lista.sort() # sort per tempo d'inizio crescente
    for inizio, fine in lista: # per ogni attività
        libera, aula = H[0] # prendo l'aula che termina prima
        if inizio > libera: # se l'attività è compatibile con l'aula
            Sol[aula].append((inizio, fine)) # la aggiungiamo all'aula
            heappop(H)
            Heappush(H, (fine, aula)) # e aggiorniamo l'aula nell'heap
        else: # altrimenti se non è compatibile
            Sol.append([(inizio, fine)]) # la aggiungiamo ad una nuova aula
            heappush(H, (fine, len(Sol)-1)) # che mettiamo nell'heap
    return Sol
```

Costo:

- Il sort ha costo Θ(n*log(n))
- Il ciclo viene iterato n volte e nel caso pessimo, in cui ogni attività viene inserita nella stessa aula, verranno eseguite heappop e heappush entrambe di costo O(log(n))
  Quindi il ciclo ha costo O(n*log(n))

Costo totale: Θ(n*log(n))

Altro esempio

Abbiamo n file di dim $d_0, d_1, \ldots, d_{n-1}$ che vogliamo memorizzare su un disco di capacità k. Tuttavia la somma delle dimensioni di questi file eccede la capacità del disco. Vogliamo dunque selezionare un sottoinsieme degli n file che abbia cardinalità massima e che possa essere memorizzato sul disco.

Es: per D = [5, 6, 3, 5, 4, 7, 3] e k = 11
La risposta sarà una tripla di file 2, 4 e 6 che occupa spazio 100

Un algoritmo greedy è: consideriamo i file in **ordine crescente** e inseriamo i file finchè non si possono più inserire (superano la capacità del disco)

```python
def file(D, k):
    n = len(D)
    lista = [(D[i], i) for i in range(n)]
    lista.sort() # sort in ordine crescente
    spazio, sol = 0, []
    for d, i in lista: # per ogni file
        if spazio + d <= k: # se la sua aggiunta non supera la capacità del disco
            sol.append(i) # aggiungiamo il file
            spazio += d # incrementiamo lo spazio sul disco
        else: # altrimenti abbiamo finito la capacità su disco
            return sol # e possiamo ritornare la soluzione
```

Costo: Θ(n*log(n))

PROVA PER ASSURDO:
Assumiamo per assurdo che la sol del greedy SOL non sia ottima, quindi tra le soluzioni migliori che hanno inserito più file nel disco prendiamo SOL* che ha più elementi in comune con SOL. Poiché SOL* non coincide con SOL, esiste:

- Un file x che ∉ SOL e ∈ SOL* e che occupa più spazio di qualunque file in SOL (poiché gli elementi in SOL occupano meno spazio di quelli non presenti)
- Un file y che ∈ SOL e ∉ SOL* (poiché SOL ⊄ SOL* infatti l'aggiunta di qualsiasi elem. a SOL porterebbe a superare la capacità del disco)

Quindi posso eliminare da SOL* il file x e inserire il file y (poiché y è minore di x per via della regola del greedy) ottenendo un nuovo insieme di file che rispetta le capacità del disco ed ha un elem. in più in comune con SOL contraddicendo l'ipotesi che SOL* è quello con più elementi in comune con SOL

Divide et Impera

mercoledì 14 maggio 2025 19:35

Problema della Selezione

Data una lista A di n interi distinti, ed un intero k, con $1 \leq k \leq n$, vogliamo sapere quale elem. occuperebbe la posizione k se il vettore venisse ordinato
Casi particolari:

- k = 1 → minimo di A
- k = n → massimo di A
- k = $\left\lfloor \frac{n}{2} \right\rfloor$ → mediano di A

Un semplice algoritmo sta nel eseguire il sort della lista e di ritornare l'elemento in posizione k. Però cosi il costo sarà Θ(n*log(n))
Invece usando la **tecnica Divide et Impera**, possiamo abbassare il costo a Θ(n) anche nel caso generale.
Approccio basato sul Divide et Impera:

- Scegli il **perno** in pos A[0]
- Costruisci due liste A1 con elem. < perno e A2 con elem. > perno
- Dove si trova l'elem. di rango k?
  - Se |A1| ≥ k allora l'elem di rango k è nel vettore A1
  - Se |A1| = k-1 allora l'elem di rango k è il perno
  - Se |A1| < k-1 allora l'elem di rango k è l'elem. di rango k - |A1| - 1 in A2 (togliamo gli elem in A1 da k, poiché il rango scalera in A2)

  Quindi eseguiamo una chiamata ricorsiva nel vettore dove si trova k

```python
def selezione2(A, k):
    # ritorna l'elem di rango k dove 1<=k<=len(A)
    if len(A) == 1: # se la lista ha un solo elem.
        return A[0] # lo ritorniamo
    perno = A[0] # prendiamo il primo elemento come perno
    A1, A2 = [], []
    for i in range(1, len(A)): # per ogni elem.
        if A[i] < perno: # se è minore del perno
            A1.append(A[i]) # va in A1
        else: # se è maggiore uguale al perno
            A2.append(A[i]) # va in A2
    if len(A1) >= k: # se |A1| supera k
        return selezione2(A1, k) # k sta in A1
    elif len(A1) == k-1: # se |A1| uguale a k-1
        return perno # k è il perno
    return selezione2(A2, k-len(A1)-1) # altrimenti sta in A2
```

Però può capitare che venga restituita una partizione massimamente sbilanciata con ad esempio |A1| = 0 e |A2| = n-1 (poiché il perno va sempre in A2) e accade quando il perno risulta l'elem. minimo della lista. E nel caso peggiore in cui cerchiamo il massimo in una lista già ordinata allora questo evento si ripete diminuendo A2 di un elemento alla volta. Quindi il costo sarebbe

$$T(n) = T(n-1) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n^2)$$

In generale la complessità superiore della procedura è:

$$T(n) = T(m) + \Theta(n)$$

dove $m = \max(|A1|, |A2|)$
E se abbiamo una regola di scelta del perno che garantisce una partizione bilanciata, ossia:

$$m = \max(|A1|, |A2|) \approx \frac{\mathrm{n}}{2}$$

Allora avremmo:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n) \quad \Rightarrow \quad T(n) = \Theta(n)$$

Però creare una partizione perfettamente bilanciata è complicato, ma possiamo accontentarci anche che le partizioni **non siano troppo sbilanciate**.
Anche se abbiamo m $\approx \frac{3}{4}n$ (o anche piuttosto vicina ad n come $\frac{99}{100}n$) comunque la ricorrenza da sempre T(n) = Θ(n)

IDEA: scegliamo quindi un **perno a caso in modo equiprobabile** tra gli elem. della lista

```python
def selezione2R(A, k):
    # ritorna l'elem di rango k dove 1<=k<=len(A)
    if len(A) == 1: # se la lista ha un solo elem.
        return A[0] # lo ritorniamo
    perno = A[randint(0, len(A)-1)] # prendiamo il perno a caso
    A1, A2 = [], []
    for x in A: # per ogni elem.
        if x < perno: # se è minore del perno
            A1.append(x) # va in A1
        else: # se è maggiore uguale al perno
            A2.append(x) # va in A2
    if len(A1) >= k: # se |A1| supera k
        return selezione2R(A1, k) # k sta in A1
    elif len(A1) == k-1: # se |A1| uguale a k-1
        return perno # k è il perno
    return selezione2R(A2, k-len(A1)-1) # altrimenti sta in A2
```

Analisi formale del caso medio:
Con la randomizzazione, la scelta del perno sugli elem. ha **uguale probabilità** $\frac{1}{n}$ e poiché la scelta dell'elem. di rango k produce |A1| = k-1 e |A2| = n-k per il tempo atteso dell'algoritmo va studiata la ricorrenza:

$$T(n) \leq \frac{1}{n}\sum_{k=1}^{n} T\big(\max\big(T(k-1), T(n-k)\big)\big) + \Theta(\mathrm{n}) \leq \frac{1}{\mathrm{n}}\sum_{\mathrm{k}=\left\lfloor \frac{n}{2} \right\rfloor}^{n-1} 2T(k) + \Theta(\mathrm{n})$$

(La prima sommatoria considera tutti i valori di k da 1 a n. Tuttavia i val. di k molto piccoli (vicini a 1) o molto grandi (vicini a n) porteranno a uno dei sottoarray ad essere più piccolo, quindi il tempo di esecuzione sarà dominato dall'altra parte. Mentre la seconda sommatoria si concentra sui val **più centrali**, che dividono l'array in due parti **più equilibrate**. Quando k è vicino a n/2, entrambi i sottoarray sono di dim simili, quindi il tempo di esecuzione è **più bilanciato**.
Perciò la seconda sommatoria è più rappresentativa del tempo atteso dell'algoritmo
Quindi la prima sommatoria è ≤ della seconda perche non considera i casi estremi (che non contribuiscono in modo significativo al tempo atteso) ma considera casi più equilibrati, dando una stima più realistica del tempo)

Proviamo a vedere se T(n) = O(n)
Prova per sostituzione:

$$T(n) = \begin{cases} \frac{1}{n}\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} 2T(k) + a * n & se\ n \geq 3 \\ b & altrimenti \end{cases}$$

(Si sceglie n ≥ 3 poiché per n < 3 (0, 1, 2) la procedura risolve in tempo costante (poiché n/2 nell'indice inferiore))
Dimostriamo che T(n) ≤ cn per una qualunque c > 0 costante
Per n ≤ 3 abbiamo T(n) ≤ b ≤ 3c che è vera per c ≥ b

Sfruttando l'ipotesi induttiva T(k) ≤ c*k e per k < n abbiamo

$$T(n) \leq \frac{1}{n}\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} 2c * k + a * n = \frac{2c}{n}\sum_{k=\lfloor \frac{n}{2} \rfloor}^{n-1} k + a * n$$

Ora possiamo dividere la sommatoria in due parti (e rendere $\lfloor \frac{n}{2} \rfloor - 1$ in $\frac{n}{2} - 2$ poiché $\lfloor \frac{n}{2} \rfloor \geq \frac{n}{2} - 1$ e siccome la stiamo sottraendo dalla sommatoria fino a n-1 la disuguaglianza rimane valida)

$$T(n) \leq \frac{2c}{n}\left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor - 1} k\right) + an \leq \frac{2c}{n}\left(\frac{n(n-1)}{2} - \frac{\left(\frac{n}{2}-1\right)\left(\frac{n}{2}-2\right)}{2}\right) + an \leq \frac{2c}{n}\left(\frac{n^2-n}{2} - \frac{\frac{n^2}{4}-n-\frac{n}{2}+2}{2}\right) + an$$

$$\leq \frac{2c}{n}\left(\left(\frac{n^2-n}{2} - \frac{n^2-6n+8}{8}\right)\right) + an \leq \frac{2c}{n}\left(\frac{3n^2+2n-8}{8}\right) + an \leq \frac{3cn}{4} + \frac{c}{2} - \frac{2c}{n} + an \leq \frac{3cn}{4} + \frac{c}{2} + an = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right) \leq cn$$

(Rimuove -2c/n poiché è negativo e toglierlo non invalida l'ineguaglianza)
(Inoltre trasforma $\left(\frac{3cn}{4} + \frac{c}{2} + an\right)$ in $cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$ per evidenziare una forma più utile per la disuguaglianza.
Possiamo riscrivere cn come $cn \geq cn - x \Rightarrow cn - (cn - x) \geq 0$ se x ≥ 0, quindi:

$$\left(\frac{3cn}{4} + \frac{c}{2} + an\right) = cn - \left(cn - \left(\frac{3cn}{4} + \frac{c}{2} + an\right)\right) = cn - \left(cn - \frac{3cn}{4} - \frac{c}{2} - an\right) = cn - \left(\frac{cn}{4} - \frac{c}{2} - an\right)$$

e infatti vediamo dopo che dobbiamo trovare una n per dire che x ≥ 0)

Per arrivare a T(n) ≤ cn vogliamo che:

$$-\left(\frac{cn}{4} - \frac{c}{2} - an\right) \leq 0 \Rightarrow \left(\frac{cn}{4} - \frac{c}{2} - an\right) \geq 0$$

Quindi abbiamo:

$$n\left(\frac{c}{4} - a\right) - \frac{c}{2} \geq 0 \Rightarrow n\left(\frac{c}{4} - a\right) \geq \frac{c}{2} \Rightarrow n \geq \frac{\frac{c}{2}}{\frac{c}{4} - a} = \frac{c}{2} * \frac{4}{c - 4a} = \frac{2c}{c - 4a}$$

Per garantire che valga per ogni n ≥ 3, basta prendere c tale che il denominatore sia piccolo, ad esempio

$$c - 4a = 4a \Rightarrow c = 8a$$

Diventa

$$\frac{cn}{4} - \frac{c}{2} - an = \frac{8an}{4} - \frac{8a}{2} - an = 2an - 4a - an = an - 4a \geq 0 \Rightarrow an \geq 4a \Rightarrow n \geq 4$$

e quindi per n ≥ 3 il termine è gia non negativo
E quindi il costo sarà T(n) = O(n)

Quindi, se si sceglie il perno in modo equiprobabile, il tempo di selezione2R è O(n) con **alta probabilità** (ovviamente nel caso peggiore, rimane O(n²) però questo accade con probabilità molto piccola)

Mediano dei Mediani

Vediamo ora un algoritmo deterministico che garantisce tempo O(n) anche nel caso pessimo
Abbiamo visto che selezionando un perno che garantisce che nessuna delle due sottoliste abbia più di c*n elem. (per qualunque costante 0<c<1) abbiamo una complessità di O(n)

Vediamo ora un metodo (noto come **il mediano dei mediani**) per selezionare un perno che garantisce di produrre due sottoliste A1 e A2 ciascuna delle quali ha non più di $\frac{3}{4}n$ elem

Algoritmo di selezione:

- Dividi l'insieme A, contenente n elem, in gruppi da 5 elem ciascuno. Considera solo i primi $\lfloor \frac{n}{5} \rfloor$ gruppi, ciascuno composto esattamente da 5 elem. (cioè scartiamo l'ultimo gruppo nel caso abbia meno di 5 elem)
- Trova il **mediano** all'interno di ciascuno di questi $\lfloor \frac{n}{5} \rfloor$ gruppi
- Calcola il **mediano p dei mediani** ottenuti al passo precedente
- Usa **p come pivot per A**

Esempio:
A = [15, 2, 10, 16, 21, 12, 1, 9, 11, 17, 22, 3, 8, 13, 14, 4, 19, 5, 6, 20, 23, 18, 7]
15, 2, 10, 16, 21 |12, 1, 9, 11, 17|22, 3, 8, 13, 14|4, 19, 5, 6, 20| 23, 18, 7 (rimuove l'ultimo gruppo poiché non ha esattamente 5 elem)
2, 10, 15, 16, 21 |1, 9, 11, 12, 17|3, 8, 13, 14, 22|4, 5, 6, 19, 20 (ordina i 5 gruppi separatamente e prende il mediano)
15, 11, 13, 6
6, 11, 13, 15 (ordina i mediani e prende il mediano dei mediani, che sarà il pivot per A)
p = 11

Se invece ordinavamo la lista normalmente e prendevamo il mediano:
A = [15, 2, 10, 16, 21, 12, 1, 9, 11, 17, 22, 3, 8, 13, 14, 4, 19, 5, 6, 20, 23, 18, 7]
A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23] (prendiamo l'elem in pos $\left\lceil \frac{n}{2} \right\rceil$, quindi $\left\lceil \frac{23}{2} \right\rceil = 12$)
Però l'ordinamento richiede tempo O(n*log(n)) rispetto all'O(n) del mediano dei mediani

Proprietà:
Se A contiene 120 elem e il perno p viene scelto in base alla regola descritta si può essere sicuri che la dim di ciascuna delle due sottoliste sarà limitata da $\frac{3}{4}n$
PROVA:

Nella lista degli m = $\left\lfloor \frac{n}{5} \right\rfloor$ (n elem in gruppi da 5) mediani selezionati in A, il perno p scelto si trova in posizione $\left\lfloor \frac{m}{2} \right\rfloor = \left\lceil \frac{\left\lfloor \frac{n}{5} \right\rfloor}{2} \right\rceil \approx \left\lfloor \frac{n}{10} \right\rfloor$ (mediano dei mediani) .
Ci sono dunque $\left\lfloor \frac{n}{10} \right\rfloor - 1$ mediani di val. < p e $\left\lfloor \frac{n}{5} \right\rfloor - \left\lfloor \frac{n}{10} \right\rfloor$ mediani di val > p
Prova che $|A2| < \frac{3}{4}n$:
Ognuno dei $\left\lfloor \frac{n}{10} \right\rfloor - 1$ mediani di val < p appartiene ad un gruppo di 5 elem. in n. Ci sono dunque in A altri 2 elem. inferiori a p per ogni mediano.
In totale abbiamo $3\left(\left\lfloor \frac{n}{10} \right\rfloor - 1\right) \geq 3\frac{n}{10} - 3$ elem. di A che finiranno in A1.
Dunque abbiamo $|A2| \leq n - \left(3\frac{n}{10} - 3\right) = \frac{7}{10}n + 3 \leq \frac{3}{4}n$ (dove l'ultima disuguaglianza segue dal fatto che n ≥ 120)
(in realtà matematicamente la disuguaglianza è vera già da $\frac{7}{10}n - \frac{3}{4}n \leq -3 \Rightarrow \frac{14-15}{20}n \leq -3 \Rightarrow -\frac{n}{20} \leq -3 \Rightarrow n \geq 60$)

Prova che $|A1| < \frac{3}{4}n$:
Ognuno dei $\left\lfloor \frac{n}{5} \right\rfloor - \left\lfloor \frac{n}{10} \right\rfloor \geq \left(\frac{n}{5} - 1\right) - \left(\frac{n}{10} + 1\right) = \frac{n}{10} - 2$ mediani di val > p appartiene ad un gruppo di 5 elem. in n. Ci sono dunque in A altri 2 elem. superiori a p per ogni mediano
In totale abbiamo $3\left(\frac{n}{10} - 2\right) = \frac{3n}{10} - 6$ elem. di A che finiranno in A2.
Dunque abbiamo $|A1| \leq n - \left(\frac{3n}{10} - 6\right) = \frac{7}{10}n + 6 \leq \frac{3}{4}n$ (dove l'ultima disuguaglianza segue dal fatto che n ≥ 10)
(infatti matematicamente la disuguaglianza è vera già da $\frac{7}{10}n - \frac{3}{4}n \leq -6 \Rightarrow \frac{14-15}{20}n \leq -6 \Rightarrow -\frac{n}{20} \leq -6 \Rightarrow n \geq 120$)

```python
def selezioneMediani(A, k):
    # ritorna l'elem di rango k dove 1<=k<=len(A)
    # scegliendo il perno con la regola del mediano dei mediani
    if len(A) <= 120: # se abbiamo 120 elem
        A.sort() # il sort sarà limitato è avrà costo O(1)
        return A[k-1] # ritorniamo l'elem in pos k-1
    # inizializza B con i mediani dei len(A)//5 di 5 elem. di A
    B = [ sorted( A[5*i: 5*i+5] )[2] for i in range(len(A)//5) ]
    # i = 0 => A[0:5]; i = 1 => A[5:10]; i = 2 =>  A[10:15] ecc
    # individua il perno con la regola del mediano dei mediani
    perno = selezioneMediani(B, ceil(len(A)/10))
    A1, A2 = [], []
    for x in A:
        if x < perno: A1.append(x) # se x < p => x va in A1
        elif x > perno: A2.append(x) # se x > p => x va in A2
    if len(A1) >= k: # se |A1| supera k
        return selezioneMediani(A1, k) # k sta in A1
    elif len(A1) == k-1: # se |A1| uguale a k-1
        return perno # k è il perno
    return selezioneMediani(A2, k-len(A1)-1) # altrimenti sta in A2
```

Complessità:
- Nota che:
  - Ordinare 120 elem. richiede tempo O(1)
  - Ordinare una lista di n elem. in gruppetti da 5 richiede tempo Θ(n) ((n/5)*sort(5 elem) = Θ(n)*Θ(1) = Θ(n))
  - Selezionare i mediani dei mediani di gruppi da 5 da una lista in cui gli elem. sono stati ordinati in gruppi da 5 richiede te mpo Θ(n)
- Sappiamo che per n ≥ 120 risulta $|A1| \leq \frac{3}{4}n$ e $|A2| \leq \frac{3}{4}n$

Dunque la complessità dell'algoritmo è:

$$T(n) \leq \begin{cases} O(1) & se\ n \leq 120 \\ T\left(\frac{n}{5}\right) + T\left(\frac{3n}{4}\right) + \Theta(n) & altrimenti \end{cases}$$

Che è del tipo

$$T(n) = T(\alpha * n) + T(\beta * n) + \Theta(n) \quad con\ \alpha + \beta = \frac{1}{5} + \frac{3}{4} = \frac{19}{20} < 1$$

Ora mostreremo che ricorrenze di questo tipo hanno costo T(n) = Θ(n)

Se T(n) = T(α * n) + T(β * n) + cn e α + β < 1, allora T(n) = Θ(n)
PROVA:
- Il fatto che α + β sia < 1 gioca un ruolo fondamentale. Consideriamo l'albeo delle chiamate ricorsive generato dalla ricorrenza e analizziamo il costo per livelli
  - Liv 1: αn + βn = (α + β)*n
  - Liv 2: (α^2n + αβn) + (β^2n + αβn) = (α^2 + 2αβ + β^2)*n = (α + β)^2*n
  - Live 3: [(α^3n + $\alpha^2\beta$n) + ($\alpha^2\beta$n + $\alpha\beta^2$n)] + [(β^3n + $\alpha\beta^2$n) + ($\alpha^2\beta$n + $\alpha\beta^2$n)] = (α^3 + 3$\alpha^2\beta$n + 3$\alpha\beta^2$n + β^3)*n = (α + β)^3*n

  Abbiamo al primo liv un costo (α + β)*n, al secondo (α + β)^2*n, al terzo (α + β)^3*n, e cosi via
- Il tempo di esecuzione totale è la somma dei vari livelli:

$$T(n) < c * n + c(\alpha + \beta) * n + c(\alpha + \beta)^2 * n \ldots = c * n * \sum_{i=0}^{\infty} (\alpha + \beta)^i = c * n * \frac{1}{1 - (\alpha + \beta)} = \Theta(n)$$

  Siccome α + β < 1 allora la serie geometrica $\sum_{i=0}^{\infty} x^i$ con x < 1, converge a $\frac{1}{1-x}$

Ora questo algoritmo risolve il problema (anche nel caso peggiore) in tempo O(n). Tuttavia, a causa delle grandi costanti moltiplicative nascoste dall'O(n), nella pratica si comporta meglio l'algoritmo randomizzato che ha tempo O(n) con alta probabilità

Programmazione Dinamica

giovedì 15 maggio 2025 17:54

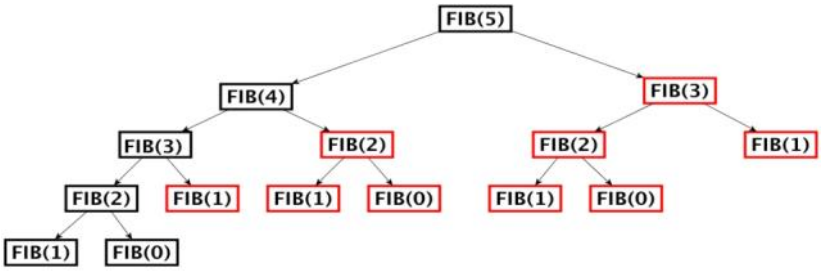
La tecnica del **Divide et Impera** segue 3 passi principali:

1. Dividi il problema in sottoproblemi di taglia inferiore
2. Risolvi (ricorsivamente) i sottoproblemi di taglia inferiore
3. Combina le soluzioni dei sottoproblemi in una soluzione del problema originale

Negli esempi visti finora i sottoproblemi ottenuti del passo 1 erano tutti **diversi**, perciò venivano individualmente risolti nella chiamata ricorsiva al passo 2.
In molte situazioni i sottoproblemi possono risultare **uguali**, quindi la tecnica svolgerebbe lo **stesso problema più volte inutilmente**

Es: Fibonacci

```
def Fib(n):
    if n <= 1: return 1
    a = Fib(n-1)
    b = Fib(n-2)
    return a + b
```



In rosso le chiamate "ridondanti"

La relazione di ricorrenza è

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Ovviamente T(n) ≥ T(n - 2) + O(1)
E risolvendola col metodo iterativo otteniamo

$$T(n) \geq \Theta\left(2^{\frac{n}{2}}\right) \Rightarrow T(n) = \Omega\left(2^{\frac{n}{2}}\right)$$

Tale inefficienza sta nel fatto che **il programma viene chiamato sullo stesso input molte volte**
I sottoproblemi in cui viene scomposto il problema non sono disgiunti (questo fenomeno prende il nome di **overlapping di sottoproblemi**) mentre l'algoritmo si comporta come se lo fossero e li risolve nuovamente.
Un modo per ottimizzare sta nel **memorizzare** in una lista i val **fib(i)** quando li si calcola cosicchè nelle chiamate future a fib(i) basta recuperare il val. salvato in precedenza senza ricalcolarlo (questa tecnica prende il nome di **memoizzazione**)

```
def Fib(n):
    F = [-1]*(n+1) # creiamo una lista vuota
    return memFib(n, F) # eseguiamo la prima chiamata

def memFib(n, F):
    if n <= 1: return 1 # caso base
    if F[n] == -1: # se non abbiamo gia salvato questo val
        a = memFib(n-1, F) # chiamata ricorsiva a n-1
        b = memFib(n-2, F) # chiamata ricorsiva a n-2
        F[n] = a + b # salviamo il valore ottenuto
    return F[n] # ritorniamo quel valore
```

Ora prima di iniziare la ricorsione per il calcolo di qualche f_i con i < n, controlla se quel val. è stato già calcolato e posto in F[i], cosi può saltare la ricorsione
Cosi verranno effettuate **esattamente n chiamate ricorsive**, portando il calcolo a Θ(n) (ogni chiamata ricorsiva ha costo O(1))
Possiamo anche **eliminare la ricorsione** e rimuovere direttamente anche la necessità della lista F (basta conservare in memoria solo gli ultimi due val calcolati) portando la complessità di spazio da Θ(n) a O(1)

```
def FibIter(n):
    if n <= 1:
        return n
    a = b = 1
    for i in range(2, n+1):
        a, b = b, a+b
    return b
```

Riassumendo:

- Siamo partiti con un algoritmo ricorsivo non efficiente che usa la tecnica del Divide et Impera
- Il motivo dell'inefficienza era la presenza di **overlapping di sottoproblemi** che abbiamo risolto tramite la tecnica della **memoizzazione** ricorrendo a "tabelle" per conservare i risultati di sottoproblemi già calcolati, cosi che non dovessero essere ricalcolati in altre chiamate ricorsive
- Abbiamo sviluppato una versione iterativa che si sbarazzava della ricorsione
- Abbiamo ottimizzato lo spazio di memoria mantenendo memorizzato solo la parte della tabella che serviva nel seguito

Nota che:

- **Nella versione ricorsiva** si parte scomponendo il problema in sottoproblemi di dimensione sempre più piccola fino ad arrivare a problemi facilmente risolvibili
  Questo approccio si chiama **approccio top-down al problema**
- **Nella versione iterativa** si comincia col risolvere i sottoproblemi di dimensioni piccola per poi via via crescendo fino ad arrivare alla soluzione originale
  Questo approccio si chiama **approccio bottom-up al problema**

Esempio:
Abbiamo n file di vari dim. ciascuna inferiore a C ed un disco di capacità C, bisogna trovare il sottoinsieme di file che può essere memorizzato e massimizza lo spazio
Progettare un algoritmo che, dati C e la lista A, dove A[i] è la dim del file i, risolva il problema
Es: per C = 100 e A = [82, 15, 40, 95, 31, 50, 40, 28] la risposta deve essere {2, 4, 7} che occupa spazio 40 + 31 + 28 = 99

Non è difficile vedere che algoritmi **greedy** tipo "memorizza i file più grandi finche c'è spazio" non trovano sempre la soluzione ottima
Per questo problema, ad ora **non si conoscono algoritmi efficienti**. Migliori algoritmi si basano su un qualche tipo di ricerca esaustiva della soluzione
Però è facile trovare algoritmi d'approssimazione efficienti con rapporti d'approssimazione costante
Per semplicità di esposizione, ci limitiamo a calcolare il val della sol. ottima (massimo spazio che può essere occupato dagli n file)

Un algoritmo basato sul **Divide et Impera** può partire cosi:

- Sia (A, C) l'istanza da risolvere:
  - Se la **lista dei file A è vuota o la capacità C è 0** la sol. ottima è 0
  - Altrimenti l'ultimo file della lista può appartenere o meno alla sol. ottima
    - **Se l'ultimo file non appartiene alla sol.** i rimanenti n-1 file devono essere una sol. ottima per **(A[:-1], C)**
    - **Se l'ultimo file appartiene alla sol** i rimanenti n-1 file devono essere una sol. ottima per **(A[:-1], C - A[n-1])**
  - Possiamo ricondurci al calcolo della sol. ottima di due sottoproblemi di dimensione inferiore in quanto manca l'ultimo file, e una volta risolti, otteniamo

i val **v1 e v2** delle loro soluzioni e la soluzione del problema di partenza sarà data da

$$\max(v1, v2 + A[-1])$$

Implementazione 1:

```
def file1(A, C):
    if len(A) == 0 or C == 0:
        return 0
    # Non prendo l'ultimo file
    lascio = file1(A[:-1], C) # calcoliamo la sol. senza l'ultimo file
    if A[-1] > C: # se l'ultimo file è maggiore della capacità del disco
        return lascio # allora ritorno la sol. senza l'ultimo file
    # Prendo l'ultimo file
    prendo = A[-1] + file1(A[:-1], C-A[-1]) # aggiungo l'ultimo file alla soluzione sul problema con capacità ridotta C-A[-1]
    return max(lascio, prendo) # ritorno il massimo tra le due scelte
```

Alla fine esploreremo **tutte le combinazioni possibili** (ricerca esaustiva) restituendo la combinazione valida che occupa più spazio possibile
Con questa implementazione il costo sarà:

$$T(n, C) = T(n-1, C) + T(n-1, C - A[n-1]) + \Theta(n)$$

Dove Θ(n) è dovuto al passaggio della copia della lista

Implementazione 2:

Posso ridurre il costo togliendo il bisogno di passare la copia della lista, ma usando un indice per indicare ogni volta la posizione dell'ultimo elem. nella lista

```
def file2(A, i, C):
    if i == 0 or C == 0:
        return 0
    # Non prendo l'ultimo file
    lascio = file1(A, i-1, C) # calcoliamo la sol. senza l'ultimo file
    if A[i-1] > C: # se l'ultimo file è maggiore della capacità del disco
        return lascio # allora ritorno la sol. senza l'ultimo file
    # Prendo l'ultimo file
    prendo = A[i-1] + file1(A, i-1, C-A[i-1]) # aggiungo l'ultimo file alla soluzione sul problema con capacità ridotta (senza l'ultimo file)
    return max(lascio, prendo) # ritorno il massimo tra le due scelte
```

Cosi il costo scende a:

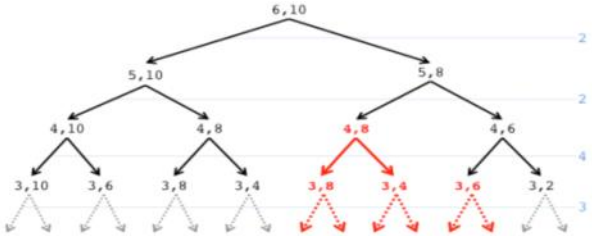
$$T(n, C) = T(n-1, C) + T(n-1, C - A[n-1]) + \Theta(1)$$

Considerando l'istanza in cui C = n-1 e A[i] = 1, 0 ≤ i < n (ogni elem. = 1), il costo dell'equazione di ricorrenza sarà:

$$\begin{aligned} T(n, n-1) &= T(n-1, C) + T(n-1, C - A[n-1]) + \Theta(1) \\ &= T(n-1, n-1) + T(n-1, n-2) + \Theta(1) \\ &\geq 2T(n-1, n-2) + \Theta(1) \\ &= 2T(n-1) + \Theta(1) \ (approssimato\ per\ stimare\ la\ crescita\ complessiva) \end{aligned}$$

che risolta dà T(n, n-1) = $\Omega\left(2^{\frac{n}{2}}\right)$ (in realtà non capisco perche siccome la sol. iterativa mi viene 2^n)

Il motivo del costo cosi alto è dovuto all'**overlappin** dei sottoproblemi
Es: parte dell'albero di ricorsione generato dall'esecuzione dell'istanza C = 10, A = [1, 5, 3, 4, 2, 2]



- ciascun nodo è una coppia con al primo posto il numero di file dell'istanza ed al secondo la capacità del disco.
- in rosso sono marcate le chiamate ricorsive duplicate e che l'algoritmo ricalcola.
- più l'istanza è grande e più chiamate ricorsive duplicate saranno presenti.

Per evitare la duplicazione ricorriamo alla **memoizzazione**, usando una tabella in cui salviamo i risultati via via calcolati per non doverli ricalcolare
La tabella avrà dimensione n*(C+1) e nella cella T[i][j] memorizzeremo il val. della soluzione del sottoproblema in cui si hanno i primi i file e la dim. del disco è j

```
def file3(A, C):
    T = [ [-1]*(C+1) for i in range(len(A)+1) ] # inizializzo la tabella di memorizzazione
    return mem_file3(A, len(A), C, T) # eseguo la prima chiamata ricorsiva

def mem_file3(A, i, C, T):
    if T[i][C] == -1: # se non abbiamo un val salvato nella tabella
        if i == 0 or c == 0: # caso base
            T[i][C] = 0
        else:
            lascio = mem_file3(A, i-1, C, T) # calcoliamo la sol. senza l'ultimo file
            T[i][C] = lascio # salviamo nella tabella la soluzione calcolata
            if A[i-1] <= C: # se l'ultimo file è minore della capacità del disco
                prendo = A[i-1] + mem_file3(A, i-1, C-A[i-1], T) # aggiungo l'ultimo file alla soluzione sul problema con capacità ridotta (senza l'ultimo file)
                T[i][C] = max(lascio, prendo) # salvo il massimo tra le due scelte nella tabella
    return T[i][C] # ritorno il val salvato nella tabella
```

Il tempo di calcolo della versione **memoizzata** è limitato dalla dimensione della tabella. Infatti **mem_file3()** esegue O(1) op. in ogni chiamata e il num. totale di chiamate non può superare la dimensione della tabella.
Quindi la complessità dell'implementazione memoizzata è O(n*C) (in realtà Θ(n*C) perche la tab. va inizializzata con i val. -1)

Quindi la complessità della versione **memoizzata è O(n*C)** e quella dell'algoritmo **esaustivo è O(2^n).**
C'è stato un risparmio? Dipende

- Se C è molto grande, ad esempio supera 2^n, la vers. memoizzata fa peggio.
- Se C non è molto grande (< 2^n), la vers. memoizzata è più efficiente e a volte enormemente più efficiente

Es:
Consideriamo un caso abbastanza realistico: **n = 100.000 file** e **capacità C = 1.000.000** (1TB (assumiamo che le dim. sono espresse in multipli di MB)) con **dimensione max dei file 1000** (1 GB) di modo che qualsiasi sottoinsieme di 1000 file può essere memorizzato su disco

- **L'alg esaustivo** eseguirà tutte le chiamate ricorsive (sarà sempre possibile scegliere il k-esimo file) almeno relativamente ai primi 1000 file (da n=100.000 a 99.000). Quindi la complessità dell'algoritmo esaustivo è $\mathbf{2^{1000}}$ (≈ 1.071509e+301, quindi 302 cifre in totale di lunghezza)
- **L'alg memoizzato** ha una complessità d'ordine di **n*C = 100.000 x 1.000.000 = 100.000.000.000**

Anche usando tutti i pc del mondo, non riusciremo mai a vedere il risultato della versione esaustiva

Nella versione memoizzata ci si può concentrare direttamente sul calcolo della tabella T eliminando cosi la ricorsione:

- Definiamo una tab. T di dim (n+1) x (C+1) dove:
  T[i, c] = spazio max con i primi i file su disco di capacità c con $0 \leq i \leq n$ e $0 \leq c \leq C$
- La sol. del problema originario la troviamo in T[n][C]
- Possiamo calcolare la tab. per righe grazie alla seguente formula ricorsiva:

$$T[i,c] = \begin{cases} 0 & se\ i = 0 \\ T[i-1,c] & se\ c < A[i-1] \\ \max\big(T[i-1,c], A[i-1] + T\big[i-1, c - A[i-1]\big]\big) & altrimenti \end{cases}$$

```python
def fileIter(A, C):
    n = len(A)
    # T[i][c] = max spazio occupabile usando i primi i file (A[:i-1]) con capacità c
    T = [ [0]*(C+1) for i in range(n+1) ] # inizializziamo a 0 la tabella
    for i in range(1, n+1): # per ogni i file aggiunti
        for c in range(C+1): # per ogni capacità crescente
            if c < A[i-1]: # se il file i-esimo è troppo grande per la capacità -> non lo includiamo
                T[i][c] = T[i-1][c] # ereditiamo il val. da T[i-1][c]
            else: # altrimenti se possiamo prendere il file i-esimo
                T[i][c] = max( # prendiamo il massimo tra lo spazio
                    T[i-1][c], # senza il file i-esimo
                    A[i-1]+T[i-1][c-A[i-1]]) # con il file i-esimo
    return T[n][C] # torniamo la sol. per tutti gli n file con capacità C
```

Vengono calcolati i risultati a partire dalle foglie (gli elem. T[0, c]), poi i risultati del liv. superiore e cosi via risalendo l'albero fino alla radice T[n, C]
Quindi è un **approccio bottom-up** al problema

RIASSUMENDO:

- In termini generali la **progr. dinamica**, al pari del divide et impera può essere vista come un metodo per risolve problemi scomponendoli in sottoproblemi e grazie all'uso di tabelle permette di non ricalcolare più volte lo stesso problema (**memoizzazione**)
- I sottoproblemi possono essere risolti a partire:
  - **Dal più grande al più piccolo (approccio top-down)** tipico delle versioni ricorsive
  - **Dal più piccolo al più grande (approccio bottom-up)** tipico delle versioni iterative

Algoritmi Pseudopolinomiali

Viene detto **pseudopolinomiale** un algoritmo che risolve un problema in **tempo polinomiale** quando i num. presenti in input sono **codificati in unario** (una cifra per ogni unità del numero, es 6 = IIIIII)
L'algoritmo ottenuto con costo O(nC) è un esempio di algoritmo pseudopolinomiale.
Il tempo è polinomiale rispetto a n e a C, ma non rispetto alla **dimensione reale dell'input**, perché la capacità C del disco viene codificata in **log(C) bit.**
Quindi per ottenere un algoritmo polinomiale alla lunghezza dell'input (in codifica binaria), nella sua complessità può comparire $\log^c(C)$ per una qualche costante c.
la presenza di C nella complessità indica che può essere anche esponenziale rispetto alla reale dimensione dell'input

Dal Valore della Soluzione alla Soluzione

La tabella che riporta le soluzioni di tutti i sottoproblemi è stata fin'ora usata solamente per calcolare il val. della soluzione ottima.
Però può essere usata anche ricavare la soluzione ottima (gli elementi che compongono la sol. ottima)
Questa è una proprietà generale degli alg. di progr. dinamica: prima ci si può concentrare sul calcolo del valore della soluzione e poi si usa la tabella per ritrovare gli elementi della soluzione.

Si parte dall'elem. che rappresenta il val. ottimo del problema e di ripercorrere all'indietro la tabella, di elemento in elemento seguendo a ritroso le "decisioni" prese per arrivare a quel valore.

Es: consideriamo la tabella T del problema dei file relativo all'istanza C = 10 e A = [1, 5, 3, 4, 2, 2]

- Tenendo conto che la tab. è calcolata per mezzo della seguente formula ricorsiva:

$$T[i,c] = \begin{cases} 0 & se\ i = 0 \\ T[i-1,c] & se\ c < A[i-1] \\ \max\big(T[i-1,c], A[i-1] + T\big[i-1, c - A[i-1]\big]\big) & altrimenti \end{cases}$$

- A partire da T[n, C] (quindi T[6][10]) da ogni elem toccato della tab T[k, c] è tracciata una freccia verso l'elem. in base al quale è stato calcolato T[k, c] (questo elemento sarà T[k-1, c] o T[k-1, c-A[k]]). Quindi se:
  - La freccia è verso T[k-1, c] (**freccia verticale**) allora il k-esimo file **non è** scelto (nella sol. del sottoproblema T[k, c]) ⇒ T[k, c] = T[k-1, c]
  - La freccia è verso T[k-1, c-A[k]] (**freccia obliqua**) allora il k-esimo file **è stato scelto** ⇒ T[k, c] > T[k-1, c]
- In base alle frecce possiamo dire quali file sono stati scelti nella sol. ottima (i file 4, 5, 1)

i \ c	0	1	2	3	4	5	6	7	8	9	10	
0	0	0	0	0	0	0	0	0	0	0	0	
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	1	1	1	5	6	6	6	6	6	5
3	0	1	1	3	4	5	6	6	8	9	9	3
4	0	1	1	3	4	5	6	7	8	9	10	4
5	0	1	2	3	4	5	6	7	8	9	10	2
6	0	1	2	3	4	5	6	7	8	9	10	2

Implementazione:

```python
def solFile(A, C):
    n = len(A)
    # T[i][c] = max spazio occupabile usando i primi i file (A[:i-1]) con capacità c
    T = [ [0]*(C+1) for i in range(n+1) ] # inizializziamo a 0 la tabella
    for i in range(1, n+1): # per ogni i file aggiunti
        for c in range(C+1): # per ogni capacità crescente
            if c < A[i-1]: # se il file i-esimo è troppo grande per la capacità -> non lo includiamo
                T[i][c] = T[i-1][c] # ereditiamo il val. da T[i-1][c]
            else: # altrimenti se possiamo prendere il file i-esimo
                T[i][c] = max( # prendiamo il massimo tra lo spazio
                    T[i-1][c], # senza il file i-esimo
                    A[i-1]+T[i-1][c-A[i-1]]) # con il file i-esimo
    #######
    valore = T[n][C] # prendiamo la sol. per tutti gli n file con capacità C
    sol = []
    i = n # partiamo dall'ultima posizione della lista
    while i > 0: # finche non finiamo le righe della lista
        if T[i][valore] != T[i-1][valore]: # se il valore nella riga corrente è diverso da quello nella riga precedente
            sol.append(i-1) # aggiungiamo il suo indice
            valore -= A[i-1] # decrementiamo il valore preso dal valore da controllare nella lista
        i -= 1 # decrementiamo l'indice
    return T[n][C], sol # ritorniamo il valore della soluzione e gli elementi della soluzione
```

- Il costo della **prima parte** è come prima O(nC) per il calcolo della tabella
- Nella **seconda parte**, la tabella viene visitata riga per riga dall'ultima cella T[n][C] per trovare i val della sol. ottima. Il costo è O(n)

Costo totale: O(nC)

Tab. Unidimensionale

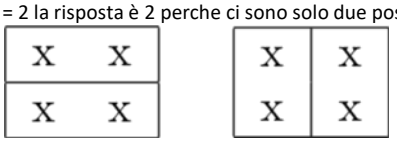
martedì 20 maggio 2025 13:31

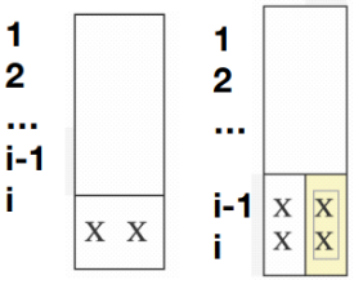
Esempi di Programmazione Dinamica con Tabella Unidimensionale

Esempio 1 - Domino

Dato l'intero n vogliamo contare il num. di differenti tassellamenti di una superficie di dimensione **n x 2** tramite tessere di dominio di dimensione **1 x 2**
L'algoritmo deve avere complessità O(n)
Es:

- Per n = 1 la risposta è ovviamente 1
- Per n = 2 la risposta è 2 perche ci sono solo due possibili tassellamenti





Implementazione:

- Useremo una tabella **monodimensionale** di dimensioni n+1 e definiamo il contenuto delle celle come segue:
  T[i] = num. di tassellamenti possibili per la superficie di dimensioni i x 2
- Una volta riempita la tabella, troveremo la soluzione in **T[n]**
- Resta da definire la **regola ricorsiva** con cui calcolare i val T[i]
  Le uniche disposizioni che possiamo avere è **mettere 2 tasselle verticali o 1 orizzontale**
  Possiamo vedere i tassellament della superficie di dim i x 2 come la somma di due diverse tipologie:
  1. Tassellamenti in cui il **tassello che copre l'ultima cella in basso a sinistra è posizionato in orizzontale sono T[i-1]**, infatti il tassetto occupa l'i-esima riga della superficie e resta da tassellare una superficie di dim (i-1) x 2 (le prime i-1 righe della superficie) in tutti i modi possibili che sono T[i-1]
  2. Tassellamenti in cui il **tassello che copre l'ultima cella in basso a sinistra è posizionato in verticale sono T[i-2]**, infatti in questo caso il vertice in basso a destra deve essere anch'esso coperto da un tassello in verticale, questi due tasselli lasciano da coprire una superficie di dim (i-2) x 2 (le prime i-2 righe della superficie iniziale) in tutti i modi possibili che sono T[i-2]

  La formula sarà:

$$T[i] = \begin{cases} 1 & se\ i = 1 \\ 2 & se\ i = 2 \\ T[i-1] + T[i-2] & altrimenti \end{cases}$$

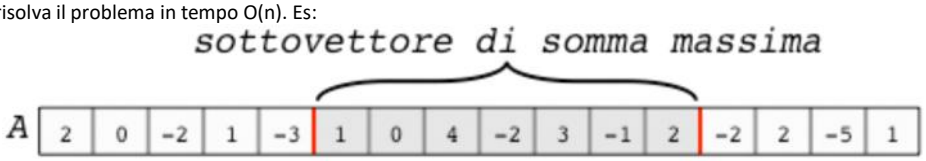
Implementazione:

```python
def domino(n):
    if n <= 2: return n
    T = [0]*(n+1) # inizializziamo la tabella monodimensionale
    T[1], T[2] = 1, 2 # inizializziamo i valori dei casi base
    for i in range(3, n+1): # per ogni altra posizione
        T[i] = T[i-1] + T[i-2] # utilizziamo la formula descritta
    return T[n] # ritorniamo la soluzione che sta in T[n]
```

Complessità: O(n)

Esempio 2 - Massimo Sottovettore

Data una lista A di n interi, vogliamo trovare una sottolista (seq. di elem. consecutivi della lista) la somma dei cui elem. è **massima**
Progettare un algoritmo che risolva il problema in tempo O(n). Es:



- Una scelta che ci porta a definire n sottoproblemi è:
  T[i] = max somma possibile per le sottoliste di A che terminano in i
- Poiché la sottolista a val. max deve terminare in una posizione qualsiasi, il val. della soluzione sarà dato da $\max_{0 \le i \le n} T[i]$
- Definiamo la **regola ricorsiva**
  - **T[0] = A[0]** in quanto esiste una sola sottosequenza nell'array di un solo elem.
  - Per T[i], con i > 0, la sottolista di val. max che termina con A[i] può essere di due tipi:
    - **Consiste del solo elem. A[i]** (in questo caso ha solo val A[i])
    - **Ha lunghezza superiore a 1**. In questo caso vale A[i] + S, dove S è la somma massima di un sottovettore che termina in i -1 (S = T[i-1])
  - Pertanto abbiamo:

$$T[i] = \begin{cases} A[0] & se\ i = 0 \\ \max\{A[i], A[i] + T[i-1] & altrimenti \end{cases}$$

Implementazione:

```python
def max_sottovettore(A):
    n = len(A)
    if n == 0: return 0 # se la lista è vuota ritorniamo 0
    T = [0]*(n+1) # inizializziamo la tabella
    T[0] = A[0] # inizializziamo il caso base
    for i in range(1, n+1): # per ogni posizione dell'array
        T[i] = max(A[i], A[i]+T[i-1]) # applichiamo la formula
    return max(T) # ritorniamo il valore massimo di T
```

Complessità: O(n)
Ottimizzazioni:

- Invece di mantenere tutta la tabella, basta mantenere solo la cella T[i-1] (e del val A[i]), portando la complessità di spazio da O(n) a O(1)
- Invece di calcolare il massimo con il metodo **max()** basta mantenere una variabile che contiene il massimo valore ottenuto per le celle di T

```python
def max_sottovettore2(A):
    n = len(A)
    if n == 0: return 0 # se la lista è vuota ritorniamo 0
    m = t = A[0] # inizializziamo la var. max e il caso base
    for i in range(1, n+1): # per ogni posizione dell'array
        t = max(A[i], A[i]+t) # applichiamo la formula
        m = max(m, t) # prendiamo il massimo tra t e m
    return m # ritorniamo il valore massimo di T
```

Esempio 3 - Sottosequenza Crescente Più Lunga

Data una seq. S di elem., una **sottosequenza di S** si ottiene eliminando zero o più elem. da S (non necessariamente in testa/coda di S)
NOTA: una sequenza di n elem. ha $\Theta(2^n)$ sottosequenze possibili
Es: S = 9, 3, 2, 4, 1, 5, 8, 6, 7, 2 una sottosequenza è 3, 1, 5, 6 (ottenuta eliminando gli elem. in rosso)
Una sottosequenza è detta **crescente** se i suoi elem. risultano **ordinati in modo crescente**

Data una sequenza S di n interi, vogliamo trovare la **lunghezza massima** per le sottosequenze crescenti presenti in S.
Es: per S = 9, 3, 2, 4, 1, 5, 8, 6, 7, 2 la risposta è **5** infatti la sottosequenza crescente più lunga in S è 3, 4, 5, 6, 7 (l'altra possibile è 2, 4, 5, 6, 7)
Progettare un algoritmo che data una sequenza S di n elem., in tempo O(n²) risolve il problema

- Usiamo una tabella monodimensionale di dim. n e definiamo le celle per:
  T[i] = lungh. max per una sottosequenza crescente che termina in S[i]
- La soluzione del problema è il valore massimo nella tabella, quindi $\max_{0\le i<n} T[i]$
- Resta da definire la regola ricorsiva
  Possiamo dividere il valore della cella in due modi:
  - Se il val. in S[i] è più **piccolo** di tutti i suoi valori che lo precedono allora la sottosequenza crescente fino a quell'elemento sarà di lunghezza 1 perché conterrà solo l'elemento S[i]
  - Altrimenti prendiamo la lunghezza massima salvata negli elementi precedenti di T[i] (solo per elementi più piccoli di S[i]) e aggiungiamo 1 per indicare l'inserimento dell'elemento corrente alla sottosequenza

  Quindi la regola sarà:

$$T[i] = \begin{cases} 1 & se\ S_j > S_i\ per\ ogni\ 0 \le j < i \\ \max\limits_{0\le j<i\ con\ S_j<S_i} (T[i]) + 1 & altrimenti \end{cases}$$

Implementazione

```python
def sottosequenza_lunga(S):
    n = len(S)
    if n == 0: return 0
    T = [0]*n # inizializziamo la tabella di T
    for i in range(1, n):
        for j in range(i): # per ogni elem. prima di i
            if A[i] > A[j]: # se l'elemento corrente è maggiore (crescente)
                T[i] = max(T[i], T[j]) # prendiamo il massimo tra le due lunghezze massime
        T[i] += 1 # aggiungiamo 1, cosi copriamo entrambe le regole, poichè:
                  # - Se tutti gli elem. precedenti erano più grandi allora passa da 0 a 1
                  # - Se abbiamo trovato una sottosequenza crescente aggiungiamo 1
    return max(T) # ritorniamo il valore massimo della tabella
```

Esempio 4 - Somma di Quadrati

Un num. intero può sempre essere rappresentato come somma di quadrati di altri num. interi. Infatti possiamo scomporre un num x come somma di addendi tuttu uguali a 1² (x = 1² + 1² + … + 1²)
Dato un intero n vogliamo sapere qual è il **num. minimo** di quadrati necessari per rappresentarlo
Es:

- Per n = 0, la risposta è 0
- Per n = 41, la risposta è 2 infatti 41 = 5² + 4². Nota che vale anche 41 = 6² + 2² + 1² ma usa 3 quadrati e non è minimale
- Per n = 6 la risposta è 3 infatti 6 = 2² + 1² + 1² e non è difficile vedere che 6 non può esprimersi come somma di due soli quadrati

Progettare un algoritmo che dato n in calcoli il num. minimo di quadrati che lo rappresentano in $\Theta\left(n^{\frac{3}{2}}\right)\Theta$

- Usiamo una tabella monodimensionale di dimensione n+1 con la regola per le celle:
- La soluzione sarà in T[n]
- La regola ricorsiva è:
  - Se i = 0 ovviamente vale T[0] = 0
  - Altrimenti servirà usare almeno un quadrato. Notiamo che prendendo un quadrato valido del tipo j con $1 \le j < \lfloor\sqrt{i}\rfloor$ e sottraendolo da i dovremmo comunque rappresentare il numero rimasto i - j². Quindi usando questa formula $T[i] = \min_{1\le j<\lfloor\sqrt{i}\rfloor}(T[i-j^2]) + 1$ indichiamo che utilizziamo sia il quadrato preso (+1) che il num. minimo di quadrati minimi per il numero restante (T[i-j²])

  Quindi la regola sarà:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ \min\limits_{1\le j<\lfloor\sqrt{i}\rfloor} (T[i-j^2]) + 1 & altrimenti \end{cases}$$

Implementazione:

```python
def somma_quadrati(n):
    if n == 0: return 0
    T = [0]*(n+1) # inizializziamo la tabella
    for i in range(1, n+1):
        T[i] = n # inizializziamo T[i] a n
        j = 1 # partiamo con j da 1
        while j**2 <= i: # finche j <= sqrt(i)
            # diminuiamo il num. di quadrati minimi necessari:
            if T[i-j**2] + 1 < T[i]: # se è minore del T[i] salvato in precedenza
                T[i] = T[i-j**2]+1 # aggiorniamo T[i]
            j += 1 # incrementiamo j
    return T[n] # ritorniamo la soluzione
```

Costo:

- Il ciclo while viene eseguito √(i) volte ($j^2 \leq i \Rightarrow j \leq \sqrt{i}$)
- Il for viene eseguito n volte
- Costo totale:

$$\sum_{i=1}^{n} \sqrt{i} = \Theta\left(n^{\frac{1}{2}+1}\right) = \Theta\left(n^{3/2}\right) \quad \left(Usando\ la\ formula\ \sum_{i=0}^{n} i^c = \Theta\left(n^{c+1}\right)\right)$$

Tab. Bidimensionale

martedì 20 maggio 2025 13:31

Esempi di Programmazione Dinamica con Tabella Bidimensionale

Esempio 1 - Cifre Decimali Non Decrescenti

Dato un intero n voglia sapere quante sono le sequenze di cifre decimali non decrescenti lunghe n
Es:

- Per n = 1 la risposta è 10
- Per n = 2 la risposta è 55
  Infatti alla cifra x al primo posto possono seguire 10 - x cifre diverse. Quindi si ha
  $$\sum_{x=0}^{9}(10-x)=\sum_{i=1}^{10} i=\frac{10*11}{2}=55$$

Progettare un algoritmo che risolva in tempo O(n)

- Usiamo una tabella bidimensionale di dim. n x 10 e definiamo il contenuto delle celle come segue:
  T[i][j] = num. di seq. decimali non decrescenti lunghe i che terminano con la cifra j
- Una volta riempita la tab. la soluzione sarà data dalla somma degli elem. nell'ultima riga: $\sum_{j=0}^{9} T[n][j]$
- Definiamo la regola ricorsiva:
  - Esiste un'unica seq. lunga 1 che termina con la cifra j, la cifra j ed è ovviamente non decrescente
  - La seq. lunga i non decrescente che termina con j è composta da una seq. non decrescente lunga i-1 cui si accoda la cifra j.
    Poiché la seq risultante sia non decrescente la seq. non decrescente lunga i-1 deve terminare con una qualunque cifra k non superiore a j (quindi deve essere una della contata con T[i-1][k] e k ≤ j)

  Quindi la regola sarà:
  $$T[i][j]=\begin{cases}1 & se\ i=1\\ \sum_{k=0}^{j} T[i-1][k] & altrimenti\end{cases}$$

Implementazione:

```
def seq_non_decr(n):
    T = [[0]*10 for _ in range(n+1)]
    for j in range(10): # inizializziamo la prima riga a 1
        T[1][j] = 1
    for i in range(n): # per ogni val
        for j in range(10): # per ogni cifra decimale
            for k in range(j+1): # per qualunque cifra k <= j
                T[i][j] += T[i-1][k] # sommiamo gli elem. <= j della riga prec.
    return sum(T[n]) # ritorniamo la somma dell'ultima riga
```

Costo:

- L'inizializzazione della tab. ha costo Θ(n) (abbiamo n+1 per un num. costante di colonne)
- Il primo for viene eseguito n volte
- Il secondo viene eseguito 10 volte
- Il terzo viene eseguito k ≤ 10 volte → costo costante Θ(1)
  Quindi il costo del secondo e terzo for sono costanti → Θ(1)

Costo tot: Θ(n)

Esempio 2 - Cammino Matrice

Data una matrice M binaria n x n vogliamo verificare se nella matrice è possibile raggiungere la cella **in basso a destra** partendo da quella in **alto a sinistra**, senza mai toccare celle col num. 1 e muovendosi solo di un passo verso destra o in basso.
Si assume che M[0][0] = 0
Es: per la matrice A la risposta è SI, per la matrice B la risposta è NO

$$A=\begin{array}{|c|c|c|c|c|c|}\hline 0&0&0&0&0&1\\\hline 0&1&0&1&1&1\\\hline 0&0&0&1&0&1\\\hline 0&1&0&0&0&0\\\hline 0&0&0&0&1&0\\\hline 1&1&0&1&0&0\\\hline\end{array}\qquad B=\begin{array}{|c|c|c|c|c|c|}\hline 0&0&0&0&0&0\\\hline 0&1&1&1&0&0\\\hline 0&1&0&0&0&0\\\hline 0&1&0&1&1&1\\\hline 0&1&0&0&0&0\\\hline 0&0&0&0&1&0\\\hline\end{array}$$

Progettare un algoritmo che in tempo Θ(n^2) risolva il problema rispondendo True o False

- Utilizziamo una tabella bidimensionale di grandezza n x n e definiamo questa regola per le celle:
  T[i][j] = True se esiste un cammino da M[0][0] fino aM[i][j] senza toccare celle di val 1 (False altrimenti)
- Troveremo il risultato nella cella in basso a destra T[n-1][n-1]
- Definiamo la regola ricorsiva:
  - M[0][0] è la posizione di partenza ed è sempre raggiungibile
  - Se il val in M[i][j] è già 1 allora è impossibile che esista un cammino che passi per quella cella, quindi T[i][j] sarà False
  - Per un cammino da M[0][0] fino alla cella M[i][j] allora significa che:
    - La cella j nella riga i è stata raggiunta dalla cella precedente della stessa riga, quindi da T[i][j-1]. Però se stiamo nella prima colonna (j = 0), non possiamo eseguire questa regola
    - Oppure la cella j nella riga i è stata raggiunta dalla cella della riga precedente, quindi da T[i-1][j]. Però se stiamo nella prima riga (i = 0), non possiamo eseguire questa regola
  - Quindi:

$$T[i][j] = \begin{cases} False & se\ M[i][j] = 1 \\ True & se\ i = j = 0 \\ T[i][j-1] & se\ i = 0 \\ T[i-1][j] & se\ j = 0 \\ T[i-1][j]\ or\ T[i][j-1] & altrimenti \end{cases}$$

Implementazione:

```python
def cammino_matrice(M):
    n = len(M)
    T = [[True]*n for _ in range(n)] # inizializziamo la tabella
    for i in range(n):
        for j in range(n):
            if M[i][j] == 1: # se nella matrice ci sta 1
                T[i][j] = False # allora non possiamo camminare su quella cella
            elif i == 0: # se siamo nella prima riga
                T[i][j] = T[i][j-1] # controlliamo solo le colonne a sx
            elif j == 0: # se siamo nella prima colonna
                T[i][j] = T[i-1][j] # controlliamo solo le righe in alto
            else: # se siamo nelle altre celle
                T[i][j] = T[i-1][j] or T[i][j-1] # controlliamo le righe in alto e le colonne a sx
    return T[n-1][n-1] # ritorniamo la cella in basso a destra
```

Entrambi i cicli vengono iterati esattamente n volte → $\Theta(n^2)$

Esempio 3 - Dimensione Sottomatrici Quadrate

Abbiamo una matrice quadrata binaria M di dim. n x n e vogliamo sapere qual è la dim. max per le sottomatrici quadrate di soli uni contenute in M
Es: per M in figura la risposta è 3 (in blu la sottomatrice di dim. 3 x 3)
Progettare un algoritmo che data la matrice M restituisce il max intero l per cui la matrice l x l è una sottomatrice di soli uni di M, in tempo $O(n^2)$

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & 0 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 \\ \hline 1 & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

- Usiamo una tabella bidimensionale n x n dove:
  T[i][j] = lato max della sottomatrice quadrata di soli 1 il cui angolo in basso a destra è M[i][j] (se M[i][j] = 0 allora T[i][j] = 0)
- La sol. sarà il max dei val. della tabella
- Definiamo la regola ricorsiva:
  - Se M[i][j] = 0 allora non si può creare un quadrato con quella cella in basso a destra, quindi il lato è 0
  - Altrimenti se M[i][j] = 1:
    - Se stiamo nella prima riga (i = 0) o nella prima colonna (j = 0), allora il quadrato ha lato 1
    - Altrimenti per formare un quadrato di lato k che termina in M[i][j], i lati a sx, in alto e in diagonale devono avere lato k-1. Il minimo di questi tre val. ci dice quanto possiamo far crescere al massimo il quadrato mantenendo al suo interno solo val 1
  - Quindi:

$$T[i][j] = \begin{cases} 0 & se\ M[i][j] = 0 \\ 1 & se\ i = 0\ or\ j = 0 \\ \min\big(T[i-1][j], T[i-1][j-1], T[i][j-1]\big) + 1 & altrimenti \end{cases}$$

Implementazione:

```python
def quadrato_sottomatrice(M):
    n = len(M)
    T = [[1]*n for _ in range(n)] # inizializziamo le celle della tabella a 1
    m = 0 # inizializziamo il val. massimo della grandezza della sottomatrice quadrata
    for i in range(n):
        for j in range(n):
            if M[i][j] == 0: # se nella matrice abbiamo 0
                T[i][j] = 0 # allora impostiamo 0
            elif i != 0 and j != 0: # se non siamo nella prima riga/colonna
                # prendiamo il minimo tra i valori delle celle in alto, a sx e in diagonale
                T[i][j] = min(T[i-1][j], T[i-1][j-1], T[i][j-1]) + 1
                m = max(m, T[i][j]) # aggiorniamo il val. massimo
    return m # ritorniamo il val. massimo
```

Costo: Il primo e il secondo for iterano esattamente n volte → $\Theta(n^2)$

Esempio 4 - Problema dello Zaino

Abbiamo uno zaino di capacità c ed n oggetti, ognuno con peso pi_i e un val v_i
Vogliamo sapere il val. max che si può inserire nello zaino
Es: consideriamo l'istanza con c = 11 e n = 5 oggetti con peso e val:

Oggetto	Valore	Peso
1	1	1
2	6	4
3	18	5
4	22	5

5	28	7

- Il sottoinsieme {3, 5} non è una sol. ammissibile (perche peso 12 > 11)
- Il sottoinsieme {1, 2, 4} è una sol. ammissibile (peso 10 e val 29) ma non ottima (esistono di val. maggiore)
- Il sottoinsieme {1, 3, 4} è una sol ammissibile (peso 11 e val 41) e si può dimostrare che è anche una sol. ottima

Progettare un algoritmo che, data la capacità c e i vettori P (pesi) e V (valori) degli oggetti, risolve il problema in tempo Θ(n*c)

- Usiamo una tab bidimensionale T di dim. (n+1)x(c+1) dove:
  T[i][j] = max val ottenibile dai primi i oggetti per uno zaino di capacità j
- La soluzione al nostro problema sarà in T[n][c]
- Definiamo la regola ricorsiva:
  - Se non si hanno oggetti (i = 0) o la capacita dello zaino è nulla (j = 0) allora il val della soluzione è 0
  - Se l'i-esimo oggetto ha un peso superiore alla capacità j dello zaino (vale a dire j < P[i-1]), allora non può esservi inserito e quindi il val della soluzione dipende da quello che si può ottenere dagli altri i-1 oggetti (Cioè T[i-1][c])
  - Altrimenti, se nello zaino si può inserire l'oggetto i (P[i-1] ≤ j) allora abbiamo due possibilità:
    - **L'oggetto i non viene inserito nello zaino**. Allora il max valore sarà dato dall'elem precedente T[i-1][j]
    - **L'oggetto i viene inserito nello zaino**. Allora abbiamo un guadagno V[i] e per i rimanenti i-1 oggetti resta disponibile una capacità residua di j - P[i] Quindi il val massimo della soluzione è V[i-1]+T[i-1, j-P[i-1]]

  Quindi si va a scegliere tra i due con **max(T[i-1][j], V[i-1] + T[i-1][j - P[i-1]])**

Quindi la regola è:

$$T[i][j] = \begin{cases} 0 & se\ i = 0\ o\ j = 0 \\ T[i-1][j] & se\ P[i-1] > j \\ \max\big(T[i-1][c], V[i-1] + T[i-1][j - P[i-1]]\big) & altrimenti \end{cases}$$

Implementazione:

```python
def zaino(P, V, c):
    n = len(P)
    T = [[0]*(c+1) for _ in range(n+1)] # inizializziamo la lista
    for i in range(1, n+1): # per ogni oggetto i
        for j in range(1, c+1): # per ogni capacita j
            if P[i-1] > j: # se il peso dell'oggetto i supera la capacita
                T[i][j] = T[i-1][j] # allora prendiamo il valore della cella precedente (il max degli altri i-1 oggetti)
            else: # altrimenti se possiamo mettere l'oggetto, scegliamo tra:
                # * Il max valore dell'elemento precedente
                # * Il guadagno dell'oggetto i insieme a quello dei rimanenti i-1 oggetti di capacita residua j-P[i]
                T[i][j] = max(T[i-1][j], V[i-1]+T[i-1][j-P[i-1]])
    return T[n][c] # ritorniamo il risultato
```

Costo: Θ(n*c) (il primo for itera per n volte, il secondo per c volte e l'inizializzazione ha lo stesso costo)

Es: con V = [1, 6, 18, 22, 28]

P = [1, 4, 5, 5, 7] e c = 10

```
T= [
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
      [0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
      [0, 1, 1, 1, 6, 7, 7, 7, 7, 7, 7],
      [0, 1, 1, 1, 6, 18, 19, 19, 19, 24, 25],
      [0, 1, 1, 1, 6, 22, 23, 23, 23, 28, 40],
      [0, 1, 1, 1, 6, 22, 23, 28, 29, 29, 40]
 ]
```

Esempio 5 - Problema del Resto

Ho n diversi tagli di banconote ed un resto c da dare. Voglio sapere in quanti modi ci sono di produrre il resto c se ho una quantità illimitata di banconote dei vari tagli

Es: con tre tagli 1, 2 e 3 e c = 5 la risposta è 5, poiché si hanno i possibili resti: 11111, 1112, 113, 23, 122

Progettare un algoritmo che dato l'intero c e la lista A con gli n tagli delle banconote (A[i] è l'i-esimo taglio) in tempo Θ(n*c) risolva il problema

- Usiamo una tabella bidimensionale di dimensioni (c+1)*(n+1) e definiamo il contenuto delle celle cosi:
  T[i][j] = num. di modi di dare il resto i quando si dispone dei soli primi j tagli 0 ≤ i < c, 0 ≤ j ≤ n
- Il nostro risultato si troverà in posizione T[c][n]
- La regola ricorsiva è data da:
  - C'è un solo modo di dare resto zero (non dare nulla) → T[0][j] = 1
  - Con zero tagli non si può dare resto maggiore di 0 → T[i][0] = 0
  - Se il j-esimo taglio è maggiore del resto da dare allora quel taglio non si può usare in quel resto e bisogna usare i tagli precedenti → T[i][j] = T[i][j-1]
  - In generale se la banconota può essere usata nel resto allora si può avere un resto in cui viene usata o un resto in cui non viene usata:
    - **La banconota i non viene usata per produrre il resto j**. Allora il resto si ottiene solo con gli altri tagli → T[i][j] = T[i][j-1]
    - **La banconota i viene usata almeno una volta per il resto j**. Allora dovrò produrre ancora il resto i - A[j-1] → T[i][j] = T[i - A[j-1]][j]

  Quindi i due modi per dare il resto sono entrambi i casi se si prende o meno la banconota → T[i][j] = T[i][j-1] + T[i - A[j-1]][j]

Quindi la regola è:

$$T[i][j] = \begin{cases} 1 & se\ i = 0 \\ 0 & se\ j = 0 \\ T[i][j-1] & se\ A[j-1] > i \\ T[i][j-1] + T[i - A[j-1]][j] & altrimenti \end{cases}$$

Implementazione:

```python
def resto(A, c):
    n = len(A)
    T = [[0]*(n+1) for _ in range(c+1)] # inizializziamo T
    for j in range(n+1):
        T[0][j] = 1 # inizializziamo T se i = 0
    for i in range(1,c+1): # per ogni resto i
        for j in range(1, n+1): # per ogni taglio j
            if A[j-1] > i: # se il taglio è maggiore del resto
                T[i][j] = T[i][j-1] # prendiamo il resto con gli altri tagli
            else: # altrimenti se possiamo usare il taglio, allora il resto si ha:
                # * se non prendiamo quel taglio -> T[i][j-1]
                # * se scegliamo il taglio dobbiamo prendere il resto rimanente -> T[i-A[j-1]][j]
                T[i][j] = T[i][j-1] + T[i - A[j-1]][j]
    return T[n][c] # ritorniamo la soluzione
```

Costo: Θ(n*c) (il primo for itera per n volte, il secondo per c volte e l'inizializzazione ha lo stesso costo)

Es: con A = [1, 2, 5] e c = 5

```
T=[
[[1, 1, 1, 1],
 [0, 1, 1, 1],
 [0, 1, 2, 2],
 [0, 1, 2, 2],
 [0, 1, 3, 3],
 [0, 1, 3, 4]
]
```

Esempio 6 - Transazione

Una **transazione** è l'acquisto di un oggetto seguito dalla sua vendita (che non può ovviamente avvenire prima del giorno dell'acquisto)
Disponiamo di un vettore A di interi dove A[i] è la quotazione dell'oggetto nel giorno i
Dato il vettore A con le quotazioni dei prossimi n giorni e dovendo eseguire una singola transazione vogliamo sapere qual è il guadagno massimo a cui possiamo aspirare
Nota che se il vettore è decrescente il guadagno massimo è 0
Es:

- Per A = [7, 9, 5, 6, 3, 2] il guadagno massimo è 3 (basta acquistare 7 e rivendere a 9)
- Per B = [3, 2, 6, 10, 4, 8, 1] il guadagno massimo è 8 (basta acquistare a 2 e rivender a 10)

Progettare un algoritmo che risolva il problema in tempo Θ(n)

- Utilizziamo una tabella monodimensionale di grandezza n+1 e definiamo il contenuto delle celle con:
  T[i] = guadagno massimo se vendo il giorno i
- Il risultato sarà il valore massimo della tabella
- La regola ricorsiva è data da:
  - Il primo giorno (i = 0) non avremo nessun guadagno (possiamo al massimo comprare e vendere allo stesso valore)
  - Vendendo il giorno i ottengo A[i] e per avere il guadagno massimo devo aver acquistato tra il giorno 0 e il giorno i in cui l'oggetto costava meno

  Quindi la regola è:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ A[i] - \min(A[:i+1]) & altrimenti \end{cases}$$

- Per ottenere tempo Θ(n) devo calcolare ciascuna delle n celle in tempo O(1), quindi conviene pre-calcolare per ciascun giorno i il costo minimo fino a quel giorno

Implementazione:

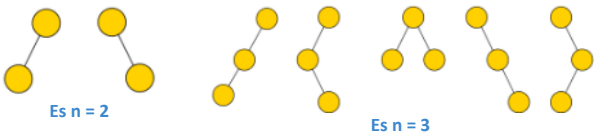
```python
def transazione(A):
    n = len(A)
    A_min = [0]*n # inizializziamo la lista dei valori minimi
    A_min[0] = A[0]
    for i in range(1, n):
        B[i] = min(A[i], B[i-1]) # prendiamo il minimo tra quel giorno e il giorno precedente
    T = [0]*n # inizializziamo la tabella
    for i in range(1, n):
        T[i] = A[i] - A_min[i] # prendiamo la vendita del giorno i - il minimo costo d'acquisto fino a i
    return max(T) # ritorniamo il massimo della lista
```

Costo: Θ(n) (i due for e la ricerca del massimo hanno costo Θ(n))

Esempio 7 - Diversi Alberi Binari

Voglio sapere quanti diversi alberi binari posso avere con n nodi (si distingue tra figlio destro e figlio sinistro)
Es:

- Per n = 0 la risposta è 1
- Per n = 1 la risposta è 1
- Per n = 2 la risposta è 2
- Per n = 3 la risposta è 5
- Per n = 4 la risposta è 14
- Per n = 5 la risposta è 42
- Per n = 6 la risposta è 132
- Per n = 7 la risposta è 429
- Per n = 8 la risposta è 1430
- Per n = 9 la risposta è 4862



Progettare un algoritmo che risolva in tempo O(n²)

- Usiamo una tabella monodimensionale di dimensione n+1 dove:
  T[i] = num. alberi con i nodi
- La sol sarà al valore T[n]
- La ricorrenza è data da:
  - Per i = 0 c'è l'albero vuoto quindi T[0] = 1
  - Se i > 0 a parte la radice restano i-1 nodi da sistemare. Nel sottoalbero di sinistra posso sistemane j con 0 ≤ j < i e nel sottoalbero di destra posso sistemarne i restanti i - 1 - j
    Per ogni j, ho dunque T[j] alberi a sinistra e T[i-1-j] alberi a destra. Il num. totale di alberi possibili sara la sommatoria tra la moltiplicazione dei due alberi

  Quindi la regola è:

$$T[i] = \begin{cases} 1 & se\ i = 0 \\ \sum_{j=0}^{i-1} T[j] * T[i-1-j] & altrimenti \end{cases}$$

Implementazione:

```python
def conta_alberi(n):
    T = [0]*(n+1) # inizializziamo la tabella
    T[0] = 1
    for i in range(1, n+1):
        for j in range(n):
            T[i] += T[j] * T[i-1-j] # aggiungiamo il prodotto tra i due sottoalberi (sx e dx)
    return T[n] # ritorniamo la soluzione
```

Costo: Θ(n²)

Backtracking Monodimensionale

giovedì 22 maggio 2025 19:45

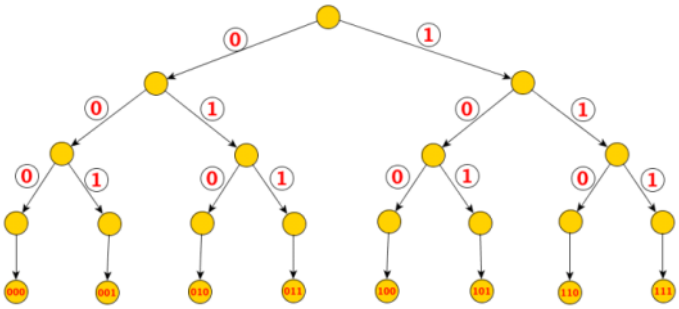
Esercizio 1

Algoritmo che prende un intero n e stampa tutte le stringhe binarie lunghe n
Es: n = 3, l'algoritmo deve stampare $2^3 = 8$ stringhe: 000, 001, 010, 011, 100, 101, 110, 111
Osservazione:

- Le stringhe da stampare sono 2^n
- Stampare una stringa lunga n costa $\Theta(n)$

Quindi il meglio che ci si può augurare per un algoritmo di questo tipo è una complessità $\Omega(2^n*n)$

```
def bk(n, sol = []):
    if len(sol) == n: # se abbiamo raggiunto n
        print("".join(sol)) # stampiamo i bit di questa soluzione
        return
    # per ogni sol < n dobbiamo stampare sia 0 che 1 in due chiamate diverse
    sol.append("0") # aggiungiamo 0 alla sol
    bk(n, sol) # chiamiamo la funzione con 0 aggiunto
    sol.pop() # rimuoviamo 0
    sol.append("1") # aggiungiamo 1 alla sol
    bk(n, sol) # chiamiamo la funzione con 1 aggiunto
    sol.pop() # rimuoviamo 1
```



Albero di ricerca con n = 3

L'albero di ricorsione ha 2^n-1 nodi interni e 2^n foglie:

- Ciascun nodo interno richiede O(1)
- Ciascuna foglia richiede O(n)

Complessità algoritmo $O(2^n*n)$

Esercizio 2

Algoritmo che prende due interi n e k, con $0 \leq k \leq n$ e stampa tutte le stringe binarie lunghe n che contengono al più k ∪
Es: n = 4, k = 2, delle $2^4 = 16$ stringhe lunghe n stampa le seguenti 11: 0000, 0001, 0010, 0011, 0100, 0101, 0110, 1000, 1001, 1010, 1100

```
def bk1(n, k, sol = []):
    if len(sol) == n: # se abbiamo raggiunto n
        if sol.count("1") <= k: # se il num. di 1 è al più k:
            print("".join(sol)) # stampiamo i bit di questa soluzione
        return
    # per ogni sol < n dobbiamo stampare sia 0 che 1 in due chiamate diverse
    sol.append("0") # aggiungiamo 0 alla sol
    bk1(n, k, sol) # chiamiamo la funzione con 0 aggiunto
    sol.pop() # rimuoviamo 0
    sol.append("1") # aggiungiamo 1 alla sol
    bk1(n, k, sol) # chiamiamo la funzione con 1 aggiunto
    sol.pop() # rimuoviamo 1
```

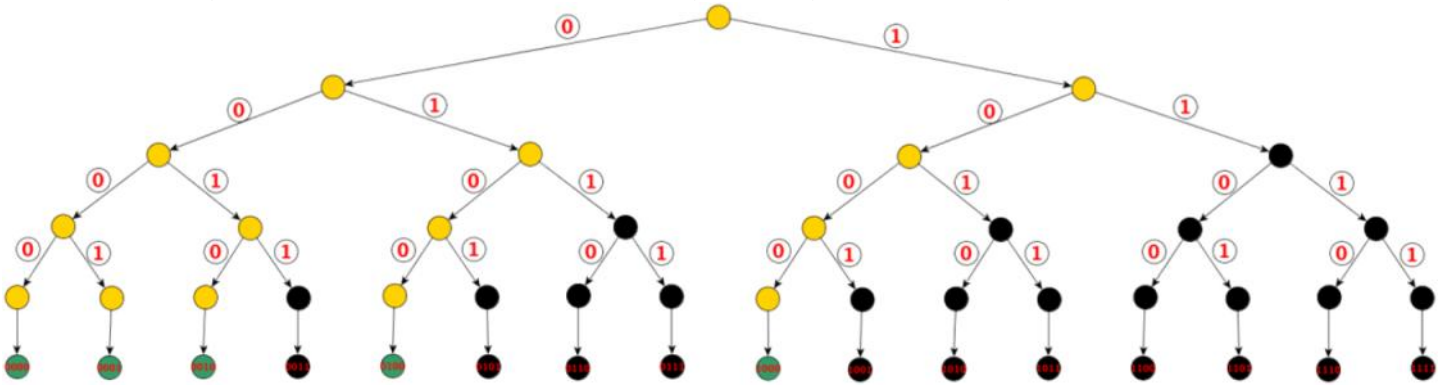
Costo: $\Theta(2^n*n)$

Indichiamo con **S(n, k)** il **num. di stringhe da stampare**. Un buon algoritmo per questo problema dovrebbe avere complessità proporzionale alle stringhe da stampare, cioè O(S(n, k)*n) (poche stringhe = poco tempo)
Es: k = 2 si ha

$$S(n,k) = 1 + n + \binom{n}{2} = 1 + n + \frac{n!}{2(n-2)!} = 1 + n + \frac{n*(n-1)*(n-2)!}{2(n-2)!} = 1 + n + \frac{n^2 - n}{2} = \Theta(n^2)$$

e quindi un buon algoritmo per k = 2 dovrebbe avere **complessità polinomiale $O(n^3)$** mentre l'algoritmo proposto ha **complessità esponenziale $\Theta(2^n*n)$** (indipendente da k)

Osservazione: **è inutile generare nell'albero di ricorsione nodi che non hanno possibilità di portare a sol. da stampare**



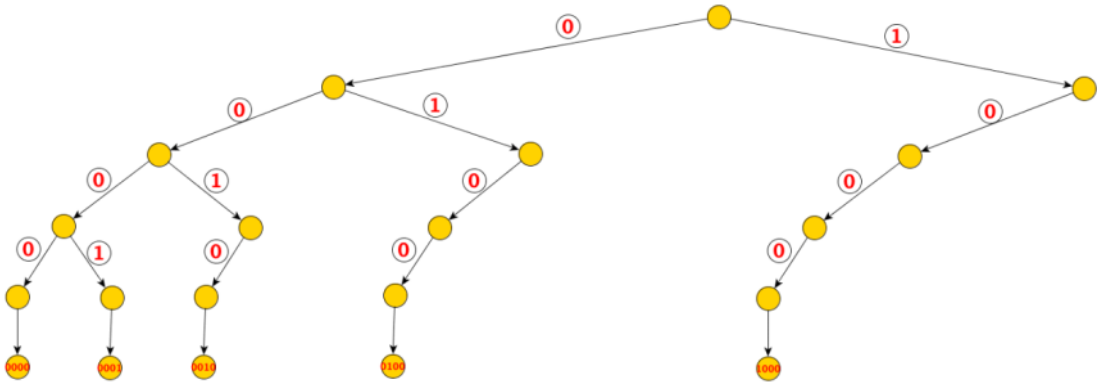
Es: n = 4, k = 1

- **In verde le foglie da stampare**
- **In nero i nodi che potrei evitare**

Algoritmo Alternativo Migliore

```python
def bk2(n, k, tot1 = 0, sol = []):
    if len(sol) == n: # se abbiamo raggiunto n
        print("".join(sol)) # stampiamo i bit di questa soluzione
        return
    # per ogni sol < n dobbiamo stampare sia 0 che 1 in due chiamate diverse
    sol.append("0") # aggiungiamo 0 alla sol
    bk2(n, k, tot1, sol) # chiamiamo la funzione con 0 aggiunto
    sol.pop() # rimuoviamo 0
    if tot1 < k: # se il num. di 1 nella stringa è minore a k
        sol.append("1") # allora possiamo aggiungere 1 alla sol
        bk2(n, k, tot1+1, sol) # chiamiamo la funzione con 1 aggiunto
        sol.pop() # rimuoviamo 1
```

Ora l'albero con n = 4 e k = 1 sarà



Calcolando il tempo di esecuzione di bk1(24, 2) e bk2(24, 2) (togliendo il print che è molto costoso) il tempo sarà:

- tempo di calcolo per bk1(24, 2): 18.11323380
- tempo di calcolo per bk2(24, 2): 0.00232100

Si consideri un algoritmo di **enumerazione basato sul backtracking** dove l'albero di ricorsione ha altezza h, il costo di una foglia è g(n) e il costo di un nodo interno è O(f(n))
Se l'algoritmo gode della proprietà: **un nodo viene generato solo se ha la possibilità di portare ad una foglia da stampare**
Allora la complessità dell'algoritmo è proporzionale al num. di cose da stampare S(n), più precisamente la complessità è:

$$O\big(S(n) * h * f(n) + S(n) * g(n)\big)$$

Questo perche:

- Il costo tot dei nodi foglia sarà O(S(n)*g(n)) (in quanto solo le foglie da enumerare verranno generate)
- I nodi interi dell'albero che verranno effettivamente generati saranno O(S(n)*h) (in quanto ogni nodo interno generato apparterrà ad un cammino che parte dalla radice e arriva ad una delle S(n) foglie da enumerare

Infatti per bk2(n, k), la proprietà di generare un nodo solo se questo può portare ad una delle S(n, k) foglie da stampare è rispettata.
Inoltre h = n, g(n) = Θ(n), f(n) = O(1)
Quindi la complessità dell'algoritmo è:

$$S(n,k) * n * O(1) + S(n,k) * \Theta(n) = \Theta(S(n,k) * n)$$

e l'algoritmo risulta ottimale e la complessità è $O(n^{k+1})$
infatti: $S(n,k) = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{k} < 2 * n^k$

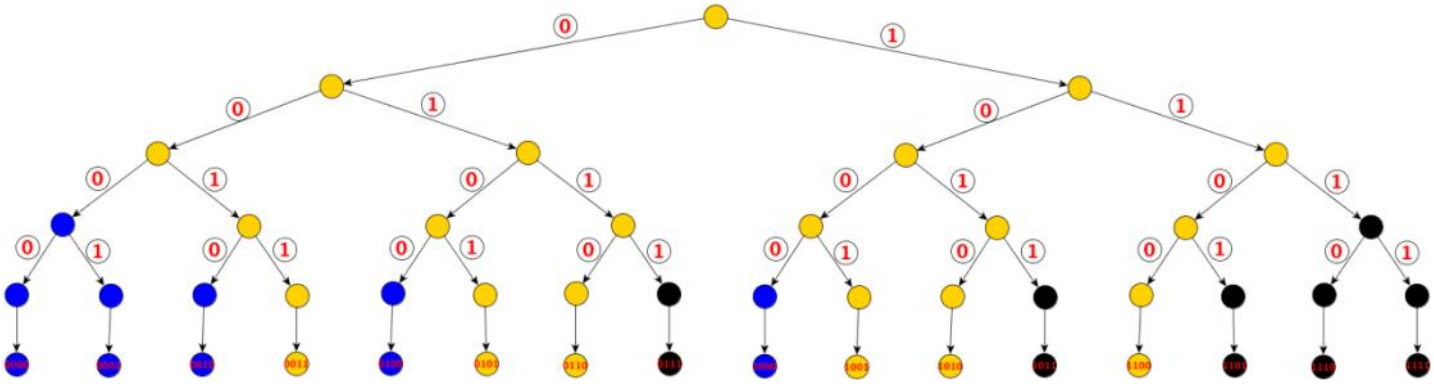
Esercizio 3

Algoritmo che, presi due interi n e k, con $0 \leq k \leq n$, stampi tutte le stringhe che contengono **ESATTAMENTE** k uni
Es: n = 6, k = 3 delle $2^6 = 64$ stringhe lunghe n bisogna stampare le seguenti 20:

000111	001011	001101	001110
010011	010101	010110	011001
011010	011100	100011	100101
100110	101001	101010	101100
110001	110010	110100	111000

Stavolta dobbiamo anche controllare se al prefisso viene aggiunto uno zero. Bisogna infatti assicurarsi che si possa sempre completare quel prefisso per ottenere una stringa valida da stampare

```python
def bk3(n, k, tot1 = 0, sol = []):
    if len(sol) == n: # se abbiamo raggiunto n
        print("".join(sol)) # stampiamo i bit di questa soluzione
        return
    # per ogni sol < n dobbiamo stampare sia 0 che 1 in due chiamate diverse
    if tot1 + n - (len(sol)+1) >= k: # se si possono aggiungere 0
        sol.append("0") # allora aggiungiamo 0 alla sol
        bk3(n, k, tot1, sol) # chiamiamo la funzione con 0 aggiunto
        sol.pop() # rimuoviamo 0
    if tot1 < k: # se il num. di 1 nella stringa è minore a k
        sol.append("1") # allora possiamo aggiungere 1 alla sol
        bk3(n, k, tot1+1, sol) # chiamiamo la funzione con 1 aggiunto
        sol.pop() # rimuoviamo 1
```



Es: albero di ricorsione bk(n, k) con n = 4 e k = 2

- In nero i nodi che evito di generare con la funzione di taglio sugli 1
- In blu i nodi che evito di generare con la funzione di taglio sugli 0

Per bk3(n, k) la proprietà di generare un nodo solo se può portare ad una sol delle S(n, k) foglie è rispettata

- Altezza albero h = n
- Tempo per una foglia g(n) = Θ(n)
- Tempo per un nodo interno f(n) = O(1)

Complessità:

$$S(n,k) * n * O(1) + S(n,k) * \Theta(n) = \Theta(S(n,k) * n)$$

e l'algoritmo risulta ottimale

La complessità è $O(n^{k+1})$, infatti $S(n,k) = \binom{n}{k} < n^k$

Esercizio Esame Settembre 2020

Algoritmo che, dato un intero n, stampi tutte le stringhe sull'alfabeto dei 3 simboli a, b, c, in cui il num. delle b supera quello di ciascuno degli altri due simboli

Es: n = 2, l'algoritmo deve stampare la sola stringa bb mentre per n = 3 deve stampare: bbb, abb, bab, bba, cbb, bcb, bbc

L'algoritmo proposto deve avere complessità O(nD(n)) dove D(n) è il num. di stringhe da stampare

- Al passo ricorsivo i-esimo possiamo potenzialmente inserire uno qualunque dei tre simboli a, b, c, però il backtracking controlla che la scelta di inserire ciascuno dei tre simboli venga fatta solo se la sol. parziale che poi si ottiene può portare ad uno o più elementi da stampare, portando la complessità proporzionale alle D(n) stringhe da stampare effettivamente.
- A questo punto usiamo una semplice funzione di taglio che si avvale di due contatori (**na** col num. di a presenti e **nc** col num. di c presenti)
  - **Inseriamo b**, che può essere sempre inserito
  - **Inseriamo a** solo se l'inserimento lascia spazio sufficiente per inserire altre b t.c. alla fine il num. di b sarà > di quello delle a e c presenti.
    Il num. di a (na) dopo l'inserimento incrementa di 1 (na+1), e il num. di b che si possono ancora mettere è **n - (na+nc+1)** e il num. di c resta nc.
    Deve quindi avversi **n - (na+nc+1) > na+1** e anche **n - (na+nc+1) > nc**
  - **Inseriamo c** (come con a) solo se l'inserimento lascia spazio sufficiente per inserire altre b t.c. alla fine il num. di b sarà > di quello delle a e c presenti.
    Il num. di c (nc) dopo l'inserimento incrementa di 1 (nc+1), e il num. di b che si possono ancora mettere è **n - (na+nc+1)** e il num. di a resta na.
    Deve quindi avversi **n - (na+nc+1) > nc+1** e anche **n - (na+nc+1) > nc**

```python
def es_9_2020(n, na=0, nc=0, sol=[]):
    if len(sol) == n: # se abbiamo raggiunto n
        print("".join(sol)) # stampiamo la soluzione
        return
    sol.append("b") # inserisco "b"
    es_9_2020(n, na, nc, sol) # chiamiamo la funzione con "b" aggiunto
    sol.pop() # rimuoviamo "b"
    # inserisco "a" solo se posso, con i simboli rimanenti, ottenere una sol valida
    if n-(na+nc+1) > na+1 and n-(na+nc+1) > nc: # se nb > na+1 ("a" aggiunto) e nb > nc
        sol.append("a") # possiamo inserire "a"
        es_9_2020(n, na+1, nc, sol) # chiamiamo la funzione con "a" aggiunto
        sol.pop() # rimuoviamo "a"
    # inserisco "c" solo se posso, con i simboli rimanenti, ottenere una sol valida
    if n-(na+nc+1) > na and n-(na+nc+1) > nc+1: # se nb > na e nb > nc+1 ("c" aggiunto)
        sol.append("c") # possiamo inserire "c"
        es_9_2020(n, na, nc+1, sol) # chiamiamo la funzione con "c" aggiunto
        sol.pop() # rimuoviamo "c"
```

Nota che nell'albero di ricorsione un nodo **viene generato solo se porta ad una foglia da stampare**

Possiamo dire che la complessità sia:

$$O(D(n) * h * f(n) + D(n) * g(n))$$

- D(n) → num. stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) → costo nodo interno
- g(n) = Θ(n) → costo foglia

Quindi il costo è: $O\big(D(n) * n + O(1) + D(n) * \Theta(n)\big) = \Theta(D(n) * n)$

Esercizio Esame Marzo 2023

Dati tre interi n, k e T positivi con T ≤ n*k. Definiamo **valida** una seq. di lunghezza n con interi tra 0 e k e la somma dei cui elementi è T

Es: n = 6, k = 4, T = 12 allora la seq 132042 (somma 12 = T) è **valida** mentre 121213 (somma 10 < T) **non è valida**

Progettare algoritmo con complessità O(n*k*S(n, k)) che, dati n, k e T, stampi tutte le sequenze valide, dove S(n, k) è il num. di seq. valide esistenti.

- Per risolvere il problema usiamo un algoritmo di backtracking per la stampa di tutte le stringhe lunghe n con i num {0, …, k} a cui aggiungiamo una funz di taglio che garantisca di generare solo stringhe di somma T
- Per la funz di taglio ad ogni inserimento nella sequenza teniamo traccia della somma degli interi usati. Dopo l'inserimento dei primi i-1 val, l'inserimento del

val j, con 0 ≤ j ≤ k avverrà se e solo se la seq. potrà poi completarsi in modo che la somma dei suoi elem. sia almeno T.
Quindi l'inserimento di j avviene solo se entrambi i vincoli sono soddisfatti:

$somma + j + (n - i - 1) * k \geq T$ ((n-i-1) sono le restanti posizioni da riempire dopo j, e ciascuna vale al massimo k, possiamo ancora raggiungere T)

$somma + j \leq T$ (se aggiungendo j non superiamo il totale T)

- Grazie alla funzione di taglio la sol parziale è sempre un prefisso di una sol da stampare

```
def es_3_2023(n, k, T, somma=0, sol=[]):
    if len(sol) == n: # se abbiamo raggiunto n
        print(sol) # stampiamo la soluzione
        return
    i = len(sol)
    for j in range(k+1): # per ogni val tra 0 e k
        if somma+i+(n-i-1)*k >= T and somma+i <= T: # se i vincoli sono soddisfatti
            sol.append(j) # possiamo aggiungere il valore
            es_3_2023(n, k, T, somma+j, sol) # incrementiamo la somma
            sol.pop() # rimuoviamo il valore
```

- Nell'albero un nodo viene generato solo se porta ad una soluzione valida
- Il costo sarà:

$$O\big(S(n,k) * h * f(n) + S(n,k) * g(n)\big)$$

- S(n, k) → num. di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = Θ(k) → costo nodo interno
- g(n) = Θ(n) → costo foglia

Quindi il costo è $O\big(S(n,k) * n * \Theta(k) + S(n,k) * \Theta(n)\big) = \Theta(S(n,k) * n * k)$

Esercizio 1

Algoritmo che, data una stringa X lunga n sull'alfabeto ternario {0, 1, 2}, stampi tutte le stringhe che è possibile ottenere da X sostituendo con "*" ad alcuni dei caratteri in modo che i caratteri rimanenti risultino in ordine strettamente crescente. Es:

- X = 021, deve stampare (non nello stesso ordine): ***, 0**, *2*, **1, 0*1, 02*
- X = 2110, deve stampare (non nello stesso ordine): ****, 2***, *1**, **1*, ***0

L'algoritmo deve avere costo O(n*S(X)) dove S(X) è il num. di stringhe da stampare

- In ogni posizione della sol o inseriamo * o lasciamo il simbolo di X. Poiché * può essere sempre aggiunto non ha bisogno di una funzione di taglio
- Il simbolo di X può essere aggiunto solo se è il primo della stringa oppure se è maggiore dell'ultimo simbolo diverso da *. Allora serve una funzione di taglio

Per tenere il tempo della funzione di taglio a O(1) usiamo una var. p che tiene traccia dell'ultimo simbolo diverso da * (p = * se non ci sono altri simboli diversi da *)

```
def str_crescente(X, p="*", sol=[]):
    if len(sol) == len(X): # se abbiamo raggiunto n
        print("".join(sol)) # stampiamo la soluzione
        return
    sol.append("*") # aggiungiamo il simbolo *
    str_crescente(X, p, sol) # chiamiamo la funzione dopo aver aggiunto *
    sol.pop() # rimuoviamo *
    i = len(sol)
    if p == "*" or X[i] > p: # se non abbiamo aggiunto simboli di X, o è maggiore del precedente
        sol.append(X[i]) # possiamo aggiungere il simbolo in posizione i
        str_crescente(X, p=X[i], sol) # chiamiamo la funzione dopo aver aggiunto X[i]
        sol.pop() # rimuoviamo X[i]
```

- Nell'albero un nodo viene generato solo se porta ad una soluzione valida
- Il costo sarà:

$$O\big(S(X) * h * f(n) + S(X) * g(n)\big)$$

- S(X) → num. di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) → costo nodo interno
- g(n) = Θ(n) → costo foglia

Quindi il costo è $O\big(S(X) * n * O(1) + S(X) * \Theta(n)\big) = \Theta(S(X) * n)$

Backtracking Bidimensionale

sabato 24 maggio 2025 19:13

Esercizio 1

Algoritmo che prende come parametro un intero n e stampa tutte le matrici binarie n x n
Es: n = 2, bisogna stampare 2^4 = 16 matrici:

0	0
0	0

0	0
0	1

0	0
1	0

0	0
1	1

0	1
0	0

0	1
0	1

0	1
1	0

0	1
1	1

1	0
0	0

1	0
0	1

1	0
1	0

1	0
1	1

1	1
0	0

1	1
0	1

1	1
1	0

1	1
1	1

```python
def stampa_matrici(n):
    sol = [ [0]*n for _ in range(n) ] # matrice base
    riempi_matrici(n, sol) # riempiamo la matrice

def riempi_matrici(n, sol, i=0, j=0):
    if i == n: # se siamo arrivati alla fine della matrice
        for row in range(n): # per ogni riga
            print(sol[riga]) # stampa la riga
        print() # stampa a capo
        return
    i1, j1 = i, j+1 # passiamo alla prossima colonna
    if j1 == n: # se abbiamo superato l'ultima colonna
        i1, j1 = i+1, 0 # passiamo alla prossima riga (resettiamo la colonna)
    sol[i][j] = 0 # impostiamo 0 la cella corrente
    riempi_matrici(n, sol, i1, j1) # chiamiamo la funzione dopo aver aggiunto 0
    sol[i][j] = 1 # impostiamo 1 la cella corrente
    riempi_matrici(n, sol, i1, j1) # chiamiamo la funzione dopo aver aggiunto 1
```

L'albero di ricorsione è binario e di altezza n^2 ha dunque $2^{n^2} - 1$ nodi interni e 2^{n^2} foglie.
Ciascun nodo interno ha costo O(1) e ciascuna foglia $O(n^2)$.
Poiché le matrici da stampare sono 2^{n^2} e la stampa di una matriche richiede tempo $\Theta(n^2)$, l'algoritmo ha complessità $O(2^{n^2} * n^2)$

Qualunque algoritmo per questo problema richiede tempo $\Omega(2^{n^2} * n^2)$, quindi l'algoritmo è ottimo

Esercizio 2

Algoritmo che dato un intero n stampa tutte le matrici binarie n x n in cui non uni adiacenti (orizzontale, verticale, diagonale)
Es: n = 3, delle 2^9 = 512 matrici quadrate 3 x 3 bisogna stampare le seguenti 34:

0	0	0
0	0	0
0	0	0

0	0	0
0	0	0
0	0	1

0	0	0
0	0	0
0	1	0

0	0	0
0	0	0
1	0	0

0	0	0
0	0	0
1	0	1

0	0	0
0	0	1
0	0	0

0	0	0
0	0	1
1	0	0

0	0	0
0	1	0
0	0	0

0	0	0
1	0	0
0	0	0

0	0	0
1	0	0
0	0	1

0	0	0
1	0	1
0	0	0

0	0	1
0	0	0
0	0	0

0	0	1
0	0	0
0	0	1

0	0	1
0	0	0
0	1	0

0	0	1
0	0	0
1	0	0

0	0	1
0	0	0
1	0	1

0	0	1
1	0	0
0	0	0

0	0	1
1	0	0
0	0	1

0	0	1
1	0	0
0	1	0

0	1	0
0	0	0
0	0	0

0	1	0
0	0	0
0	0	1

0	1	0
0	0	0
0	1	0

0	1	0
0	0	0
1	0	0

0	1	0
1	0	0
1	0	1

1	0	0
0	0	0
0	0	0

1	0	0
0	0	0
0	0	1

1	0	0
0	0	0
0	1	0

1	0	0
0	0	0
1	0	0

1	0	0
0	0	0
1	0	1

1	0	0
0	0	1
0	0	0

1	0	0
0	0	1
1	0	0

1	0	1
0	0	0
0	0	0

1	0	1
0	0	0
0	0	1

1	0	1
0	0	0
0	1	0

1	0	1
0	0	0
1	0	0

1	0	1
0	0	0
1	0	1

La complessità deve essere $O(S(n)*n^2)$ dove S(n) è il num. di matrici da stampare

- Usiamo una funzione di backtracking
- Nella cella sol[i][j] è potenzialmente consentito inserire 0 o 1. Basta controllare che il bit inserito porti ad una soluzione valida
  0 può sempre essere inserito. Il bit 1 invece richiede una funzione di taglio che controlli che le caselle di sol già riempite e adiacenti a sol[j][i] siano diverse da 1. Poiché il riempimento viene eseguito dall'alto a sinistra della matrice fino in basso a destra, riga per riga, le potenziali celle già riempite da controllare sono sol[i-1][j-1], sol[i-1][j], sol[i-1][j+1], sol[i][j-1]. Quindi inseriamo 1 in sol[j][i] solo se:
  - (i == 0 and j == 0) or (siamo nella prima cella)
  - (i == 0 and j > 0 and sol[i][j-1] == 0) or (siamo nella prima riga e dopo la prima colonna)
  - (i > 0 and j == 0 and sol[i-1] == sol[i-1][j+1] == 0) or (siamo dopo la prima riga e nella prima colonna)
  - (i > 0 and j == n-1 and sol[i-1][j-1] == sol[i-1][j] == sol[i][j-1] == 0) or (siamo dopo la prima riga e nell'ultima colonna)
  - (i > 0 and 0 < j < n-1 and sol[i-1][j-1] == sol[i-1][j] == sol[i-1][j+1] == sol[i][j-1] == 0) (siamo dopo la prima riga e tra la prima e l'ultima colonna)

```python
def matrici_uni_adiacenti(n):
    sol = [ [0]*n for _ in range(n) ] # matrice base
    riempi_matrici_uni(n, sol) # riempiamo la matrice

def riempi_matrici_uni(n, sol, i=0, j=0):
    if i == n: # se siamo arrivati alla fine della matrice
        for row in range(n): # per ogni riga
            print(sol[riga]) # stampa la riga
        print() # stampa a capo
        return
    i1, j1 = i, j+1 # passiamo alla prossima colonna
    if j1 == n: # se abbiamo superato l'ultima colonna
        i1, j1 = i+1, 0 # passiamo alla prossima riga (resettiamo la colonna)
    sol[i][j] = 0 # impostiamo 0 la cella corrente
    riempi_matrici_uni(n, sol, i1, j1) # chiamiamo la funzione dopo aver aggiunto 0
    # se possiamo inserire l'1 in una posizione già inserita in cui non ci sia un uno adiacente
    if (i == 0 and j == 0) or
    (i == 0 and j > 0 and sol[i][j - 1] == 0) or
    (i > 0 and j == 0 and sol[i - 1][j] == sol[i - 1][j + 1] == 0) or
    (i > 0 and j == n - 1 and sol[i - 1][j - 1] == sol[i - 1][j] == sol[i][j - 1] == 0) or
    (i > 0 and 0 < j < n - 1 and sol[i - 1][j - 1] == sol[i - 1][j] == sol[i - 1][j + 1] == sol[i][j - 1] == 0):
        sol[i][j] = 1 # impostiamo 1 la cella corrente
        riempi_matrici_uni(n, sol, i1, j1) # chiamiamo la funzione dopo aver aggiunto 1
```

Siano S(n) le matrici da stampare:

- L'algoritmo ha complessità O(S(n)*n²) perche:
  - L'albero di ricorsione è di altezza n²
  - Solo i nodi che portano ad una sol di S(n) vengono generati
  - I nodi interni sono O(S(n)*n²) e le foglie generate saranno S(n)
  - Ciascun nodo richiede tempo O(1) e ciascuna foglia O(n²)
  - Il tempo totale sarà $O(S(n) * n^2) + O(S(n) * n^2) = O(S(n) * n^2)$
- **Qualunque algoritmo per questo problema richiede tempo Ω(S(n)*n²)**
  - Le sol da stampare sono S(n) e il tempo per stampare ciascuna di queste è Θ(n²)

L'algoritmo proposto è ottimo

Esercizio 4 (il 3 l'ho saltato)

Algoritmo che preso intero n stampa tutte le permutazioni da 0 a n-1

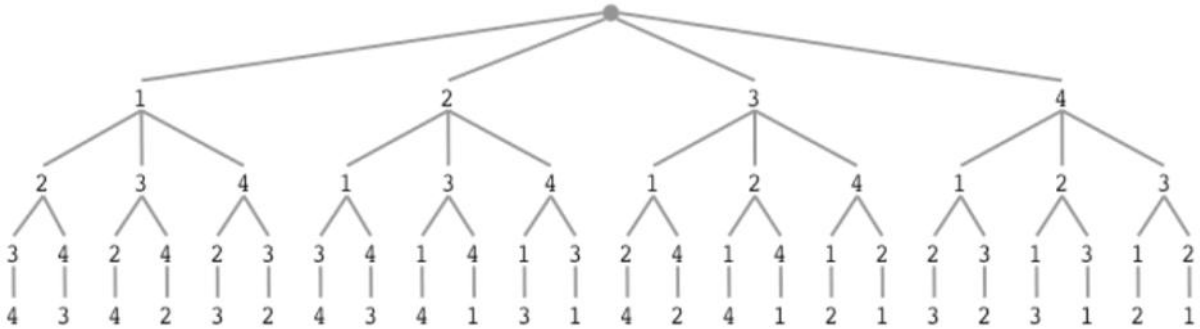
Es: per n = 4 deve stampare 4! = 24 permutazioni:

$[0,1,2,3]$ $[0,1,3,2]$ $[0,2,1,3]$ $[0,2,3,1]$ $[0,3,1,2]$ $[0,3,2,1]$ $[1,0,2,3]$

$[1,0,3,2]$ $[1,2,0,3]$ $[1,2,3,0]$ $[1,3,0,2]$ $[1,3,2,0]$ $[2,0,1,3]$ $[2,0,3,1]$

$[2,1,0,3]$ $[2,1,3,0]$ $[2,3,0,1]$ $[2,3,1,0]$ $[3,0,1,2]$ $[3,0,2,1]$ $[3,1,0,2]$

$[3,1,2,0]$ $[3,2,0,1]$ $[3,2,1,0]$



Albero delle Permutazioni

L'albero delle permutazioni ha Θ(n!) foglie e Θ(n!) nodi interni

- **Nodi interni:**

$$\sum_{i=0}^{n} \frac{n!}{i!} < n! * \sum_{i=0}^{\infty} \frac{1}{i!} \leq n! * \sum_{i=0}^{\infty} \frac{2}{2^i} = n! * \frac{2}{1-\frac{1}{2}} = 4 * n!$$

- Ho usato:
  1) $i! \geq \frac{2^i}{2}$
  2) $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x} \quad per\ x < 1$

Implementazioni:

- per tenere conto dei numeri gia inseriti, usiamo una tabella "preso" dove indica con 0 e 1 se l'elem. in posizione i è stato preso o meno.
  - preso[i] = 0 → i non è stato preso → possiamo inserire l'elem. i nella soluzione

- preso[i] = 1 → i è stato già preso → non possiamo inserire l'elem. i nella soluzione

```
def permutazioni(n):
    preso = [0]*n # tabella base
    crea_permutazioni(n, preso) # chiamata ricorsiva per creare le permutazioni

def crea_permutazioni(n, sol=[], preso):
    if len(sol) == n: # se abbiamo raggiunto n
        print(sol) # stampiamo la soluzione
        return
    for j in range(n): # per ogni num. da 0 a n-1
        if preso[j] == 0: # se non è stato gia inserito
            sol.append(j) # lo aggiungiamo alla soluzione
            preso[j] = 1 # lo impostiamo come preso
            crea_permutazioni(n, sol, preso) # continuiamo con la creazione della permutazione
            sol.pop() # togliamo l'elem. j
            preso[j] = 0 # reimpostiamo come non preso
```

- L'algoritmo ha complessità Θ(n!*n)
  - L'albero ha Θ(n!) nodi interni e n! foglie = $2^{\Theta(n*\log(n))}$
  - Ciascun nodo interno richiede tempo Θ(n) e ciascuna foglia richiede tempo Θ(n)
- Qualsiasi algoritmo per questo problema richiede tempo Ω(n!*n)
  - Le sol da stampare sono n! e il tempo per stampare ciascuna di queste è Θ(n)

L'algoritmo proposto è ottimo

Esercizio 5

Algoritmo che preso intero n stampa tutte le permutazioni dei num. da 0 a n-1 dove nelle pos pari compaiono numeri pari
La complessità deve essere O(S(n)*n²) dove S(n) è il num. di permutazioni da stampare
Es: n = 5 delle 5! = 120 permutazioni bisogna stampare solo queste 12:

$$[0,1,2,3,4] \quad [0,1,4,3,2] \quad [0,3,2,1,4] \quad [0,3,4,1,2]$$

$$[2,1,0,3,4] \quad [2,1,4,3,0] \quad [2,3,0,1,4] \quad [2,3,4,1,0]$$

$$[4,1,0,3,2] \quad [4,1,2,3,0] \quad [4,3,0,1,2] \quad [4,3,2,1,0]$$

- Per la stampa usiamo una funzione di backtracking
- Dobbiamo controllare se al passo i-esimo (quindi pos i della tabella) l'intero da inserire j sia inserito solo se j è pari e la posizione in cui viene inserito (i) è pari oppure se j è dispari e la posizione in cui viene inserito (i) è dispari (qunidi j modulo 2 == i modulo 2)

```
def permutazioni_pari(n):
    preso = [0]*n # tabella base
    crea_permutazioni_pari(n, preso) # chiamata ricorsiva per creare le permutazioni

def crea_permutazioni_pari(n, sol=[], preso):
    if len(sol) == n: # se abbiamo raggiunto n
        print(sol) # stampiamo la soluzione
        return
    for j in range(n): # per ogni num. da 0 a n-1
        if preso[j] == 0 and j%2 == len(sol)%2: # se non è stato gia inserito e j e i sono entrambi pari/dispari
            sol.append(j) # lo aggiungiamo alla soluzione
            preso[j] = 1 # lo impostiamo come preso
            crea_permutazioni_pari(n, sol, preso) # continuiamo con la creazione della permutazione
            sol.pop() # togliamo l'elem. j
            preso[j] = 0 # reimpostiamo come non preso
```

Sia S(n) il num. di permutazioni da stampare:

- L'algoritmo ha complessità O(S(n)*n²)
  - L'albero di ricorsione ha altezza n
  - Solo i nodi che portano ad una sol di S(n) vengono generati
  - I nodi interni sono O(S(n)*n) e le foglie sono S(n)
  - Ciascun nodo interno richiede tempo Θ(n) e ciascuna foglia Θ(n)
  - Tempo totale: $O(S(n)*n)*\Theta(n)+S(n)*\Theta(n)=O(S(n)*n^2)$

Slide Introduzione

mercoledì 12 marzo 2025 16:19

1

Data una lista di n interi vogliamo determinare se la lista ha un elemento di maggioranza assoluta (vale a dire un elemento c he compare nella lista almeno $\left\lfloor \frac{n}{2} \right\rfloor + 1$ volte).
Progettare un algoritmo efficiente (possibilmente ottimo) per questo problema.
[0, 0, 0, 0, 0, 1, 5, 7, 9]. len()//2 +1 = 5. 0 c'è 5 volte → return True
[0, 0, 0, 0, 1, 5, 6, 7, 9]. 0 c'è 4 volte < 5 → return False

```
def es_1_intro(L):
    n = len(L)
    L.sort() # sort ha costo O(n*log(n))
    check = n//2 + 1 # val di controllo maggioranza assoluta
    c = 1 # contatore delle occorrenze dei val
    for i in range(1,n): # per ogni elem. sortato
        if L[i] == L[i-1]: # se quello precedente è uguale al corrente
            c += 1 # aumentiamo il contatore
            if c >= check: # se supera il controllo
                return L[i] # questo val è un elem. di magg. assoluta
        else: # se stiamo controllando un altro numero
            c = 1 # resettiamo il contatore
    return None # altrimenti non lo ha
```

Solo che ha costo O(n*log(n)) per via del sort.
Quello dello del prof che ha costo O(n) si basa sul fatto che se c'è un elem. x di maggioranza assoluta allora se per ogni el em. diverso da x rimuoviamo un x dalla lista B alla fine comunque avremmo un elem. di x rimasto. Quindi per ogni elem. che incontriamo, se è diverso dal'ultimo salvato nell a lista B allora rimuoviamo l'ultimo elem. e non insieriamo quello corrente, se invece l'ultimo è uguale a quello corrente allora lo aggiungiamo aumentando gli elem. nell a lista B.
Quindi alla fine, se c'è l'elem di maggioranza in A, alla fine si troverà in B, e B avrà solo valori uguali.
es A = [0,0,0,0,0,1,2,3,4]. Verranno inseriti inizialmente i cinque 0 in B e poi siccome incontriamo ancora 4 numeri diversi da 0, quattro zero verranno rimossi dalla lista B, lasciandola B = [0] e quello sarà l'elem. di maggioranza assoluta

```
def es_1_intro_prof(A):
    B = []
    for x in A:
        if B and B[-1]!=x:
        # se non è vuoto e l'ultimo è uguale a x
            B.pop() # rimuove l'ultimo
        else:
            B.append(x) #altrimenti lo aggiungiamo
    if B and A.count(B[0]) > len(A)//2:
        return B[0]
    return None
```

Ha costo O(n) perche c'è un solo for

2

Dati $2^{n-1} + 1$ distinti sottoinsiemi ottenuti a partire da un insieme S di n elementi. Progettare un algoritmo ottimo che testi se tra i sot toinsiemi ne esistano due, A e B, con A ∪ B = S.
Da un insieme di n elem. abbiamo 2^n - 1 sottoinsiemi distinti (senza contare l'insieme vuoto)

3

Data una lista di n interi distinti ed un intero k vogliamo sapere se nella lista sono presenti due elementi la cui somma dia k e nel caso ci siano quali sono.

1) Progettare un algoritmo che risolve il problema in O(n^2)

```
def es_3_intro(L, k):
    for x in L:
        for y in L:
            if x != y and x + y == k:
                # se non sono lo stesso elem.
                # e la loro somma è k
                return x, y # li ritorniamo
    return None, None
```

Il doppio ciclo for ha costo O(n^2)

2) Progettare un algoritmo che risolve il problema in tempo o(n^2) (più efficiente di O(n^2), es. O(n*log(n)) o O(n))

```
def es_3_intro(L, k):
    s, d = 0, len(L)-1
    L.sort() # sortiamo la lista costo O(n*log(n))
    while s < d: # finche l'indice sx non supera quello dx
        som = L[s] + L[d]
        if som == k: # se la somma dei due elem. è k
            return L[s], L[d] # ritorniamo i due elem.
        elif som > k: # se supera k
            d -= 1 # vado ad un num. piu basso a dx
        else: # se ancora non siamo arrivati a k
            s += 1 # vado ad un num. più alto a sx
    return None, None
```

Il sort ha costo O(n*log(n)) e il while ha costo O(n). costo tot = O(n*log(n))

3) Progettare un algoritmo che risolve il problema in tempo ammortizzato O(n)

4

Data una lista di n interi distinti ed un intero k vogliamo contare le triple di elementi della lista che hanno come somma k.

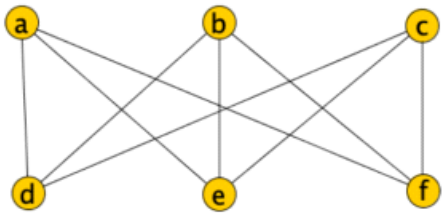
1) Progettare un algoritmo che risolve il problema in tempo $O(n^3)$
2) Progettare un algoritmo che risolve il problema in tempo $o(n^3)$
3) Progettare un algoritmo che risolve i problema in tempo ammortizzato $O(n^2)$

Slide Grafi 1 (grafi)

mercoledì 12 marzo 2025 18:32

Sia G un grafo **non-diretto**:

- Dimostrare che se tutti i vertici di G hanno grado almeno due allora nel grafo c'è almeno un ciclo
  Se il grado di ogni vertice è esattamente due, si può affermare che G è un ciclo?
- Dimostrare che la somma dei gradi dei nodi di G è sempre un num. pari
- Dimostrare che in G i nodi di grado dispari sono sempre in num. pari
- Dimostrare che se G ha almeno due nodi allora in G ci sono due nodi con lo stesso grado
- Assumete che G sia un grafo completo di n nodi con $n \geq 3$ e che i suoi archi siano stati colorati con due colori.
  Quale è il min. val. di n per cui si può essere sicuri che G contiene un ciclo di 3 nodi monocromatico?
- Dimostrare che il grafo di 6 nodi in figura non è planare



Slide Grafi 2 (DFS e padri)

lunedì 17 marzo 2025 12:38

Esercizio 1

Sia G un grafo non orientato e connesso con n nodi. Rispondere ai seguenti quesiti:

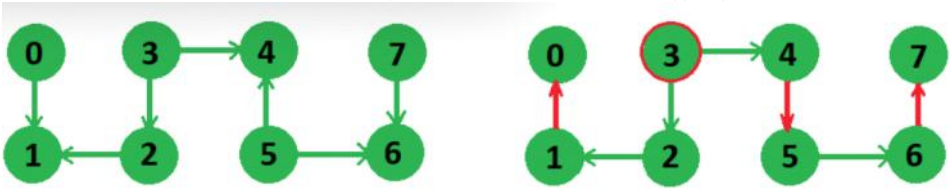
1. Determinare condizioni necessarie e sufficienti affinché il grafo G possa avere tutti i suoi nodi di grado esattamente k.

2. Dimostrare che G contiene almeno un vertice la cui rimozione non rende il grafo sconnesso.
3. Sia m il numero di archi di G. Durante una visita in profondità del grafo, quante volte si incontrano nodi già visitati? In altre parole, quante volte, nel corso della visita, scorrendo le liste di adiacenza dei nodi si incontrano nodi che erano già stati esplorati?
4. Per ogni nodo x in G, sia D_x la distanza massima da x nel grafo G (cioè il numero massimo di archi che bisogna attraversare per raggiungere, a partire da x, un qualunque altro nodo di G). Rispondere ai seguenti quesiti:
   a. Possono esistere due nodi u e v in G tali che D_u =4 e D_v = 8?
   b. Possono esistere due nodi u e v in G tali che D_u =4e D_v = 9?

   In caso affermativo, fornire un esempio di un grafo G con questa proprietà
   In caso negativo, dimostrare l'impossibilità di tale configurazione

Esercizio 2

In un grafo aciclico T con nodi (0, 1, 2, …, n-1) gli archi sono stati orientati a caso. Vogliamo sapere in quali nodi radicare il grafo in modo tale che risulti minimo il numero di archi la cui direzione va invertita per far si che tutti i nodi siano raggiungibili dalla radice.



Ad esempio in figura è riportato a sinistra il grafo aciclico T con gli archi orientati ed a destra si mostra che radicandolo nel nodo 3 bisogna poi invertire 3 archi.
Progettare un algoritmo che prende come parametro l'albero orientato e in tempo O(n) risolve il problema restituendo l'insiem e di nodi da scegliere come radici.
Esempi:

- Per l'albero diretto T=[[1], [ ], [1], [2, 4], [ ], [4, 6], [ ], [6]] l'algoritmo deve restituire l'insieme {3, 5, 7} (siccome bisogna invertire solo 3 archi per tutti e tre).
- Per l'albero diretto T=[[1, 2], [ ], [6], [0, 7, 8], [1], [1], [ ], [ ], [ ]] l'algoritmo deve restituire l'insieme {3}.

Siccome è un grafo diretto aciclico, il grafo è un **albero** → num. di archi è sempre m = n - 1 → complessita di un DFS è O(n + (n-1)) = O(2n) = O(n).
Creiamo un altro grafo non orientato per eseguire la visita DFS.
Per ogni nodo eseguiamo la visità DFS e contiamo e salviamo quanti archi bisogna invertire nella ricerca. Durante la ricerca salviamo il valore minimo degli archi invertiti.
Dopo aver creato una lista dove ogni elemento ha come chiave il nodo e come valore il num. di archi da invertire se fosse rad ice iteriamo questa lista e ritorniamo tutti i nodi che hanno il valore uguale al val. minimo salvato in precedenza.

Slide Grafi 3 (colori e componenti)

lunedì 17 marzo 2025 18:59

Esercizio 1

1. Di quanto può variare il numero di componenti connesse di un grafo non orientato con l'aggiunta o la rimozione di un arco?
2. Di quanto può variare il numero di componenti fortemente connesse di un grafo orientato con l'aggiunta o la rimozione di un a rco?
3. Un grafo si dice bipartito se è possibile partizionare i suoi vertici in due insiemi V1 e V2 in modo che tutti gli archi abbi ano un estremo in V1 e l'altro in V2. Progettare un algoritmo che, dato un grafo G, determini in tempo O(n + m) se G è bipartito o meno.
4. Un grafo non orientato si dice euleriano se esiste un ciclo che attraversa ogni arco del grafo esattamente una volta e ritorn a al nodo di partenza. Progettare un algoritmo che dato un grafo non orientato determina se questo è euleriano o meno in tempo O(n + m). Nota che in base alla definizione un grafo euleriano non deve essere necessariamente connesso

Soluzioni 1

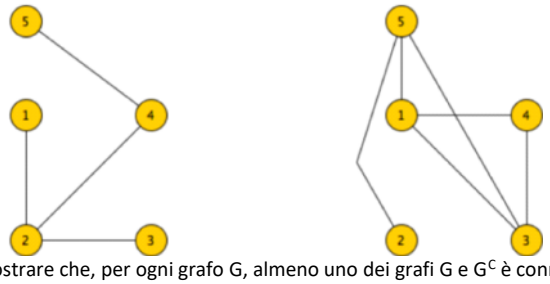
1. Il numero varia in base al fatto se l'arco aggiunto/rimosso è ponte o meno.
   a. Se si rimuove un ponte (arco critico) di un componente, allora il componente si divide in due componenti. Quindi si aggiunge un componente in piu. Altrimenti se rimuoviamo un arco non critico, quindi vi è un ciclo nel componente, allora il numero di componenti non cambia
   b. Se si aggiunge un ponte tra due componenti, allora diventano un componente unico. Quindi si rimuove un componente dal totale Altrimenti se aggiungiamo un arco tra due nodi dello stesso componente, allora il num. di componenti non cambia

   Quindi se il num. di componenti è x, allora varia da x-1 a x+1 se rispettivamente si aggiunge o rimuove un ponte, altrimenti il num. non cambia
2. Il num. varia in base al fatto se l'arco aggiunto/rimosso crea o meno nuovi componenti fortemente connessi
   a. Se la rimozione dell'arco rimuove la proprietà di connessione di una componente allora si creeranno O(k) nuovi componenti, dove k è il num. di nodi della componente a cui si è rimosso l'arco. Poiché se ad esempio la componente ha k nodi con 1 nodo uscente ognuno allora, rimuovendo un arco, si creano k nuovi componenti. Se invece tra il nodo 0 e il nodo k-1 del componente vi è un arco, rimuovendo l'arco tra k-1 e k, allora avremmo un componente con i nodi da 0 a k-1 e un componente con il solo nodo k.
   Se invece la proprietà di connessione rimane, il numero di componenti rimane uguale
   b. Se l'aggiunta di un arco crea un nuovo componente fortemente connesso allora si rimuoveranno O(k) componenti, dove k è il num. di nodi della nuova componente creata. Se ad esempio abbiamo una catena di k nodi e connettiamo con un arco l'ultimo nodo al primo, creeremo un ciclo e quindi un componente unico, passando da k componenti distinti a 1 componente singolo.
   Se invece non vengono creati nuovi componenti fortemente connessi allora il num. di componenti rimane uguale.

   Quindi il num. di componenti può aumentare/diminuire di O(k) componenti (con $k \leq n$) in base al fatto che l'aggiunta/rimozione del nuovo arco crei o meno delle nuove componenti fortemente connesse
3.

Esercizio 2

Dato un grafo G il suo grafo complementare G^C è un grafo che ha gli stessi nodi di G ma in cui un arco è presente se e solo se manca a G
Ad esempio, di seguitp a sinistra un grafo G e a destra il suo complemento G^C.



Dimostrare che, per ogni grafo G, almeno uno dei grafi G e G^C è connesso

Soluzione 2

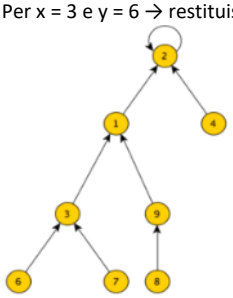
Partiamo dall'ipotesi che G non è connesso, quindi ci sono più componenti in G. Dobbiamo provare che GC è connesso
Per assurdo diciamo che anche GC non è connesso, però siccome è il complemento di G, avremmo che i nodi dei vari componenti, poiché in G sono separati tra i differenti componenti del grafo, in GC quei nodi saranno connessi tra di loro.
Quindi ogni nodo x di un componente sara connesso con i vari nodi y degli altri componenti, e viceversa, creando un unico componente in GC, il che è assurdo siccome abbiamo detto che GC non è connesso.
Ad esempio, avendo n nodi e 0 archi in G, in GC tutti i nodi saranno connessi tra di loro. Allo stesso modo se abbiamo un grafo completo con n nodi e $\Theta(n^2)$ archi in G (quindi un grafo connesso), in GC avremmo n nodi e 0 archi

Esercizio 3

Dato un albero di n nodi rappresentato tramite il vettore dei padri P (il padre della radice è la radice stessa) e due nodi dell'albero x e y, dare lo pseudocodice di un algoritmo che in tempo O(n) calcola la distanza tra x e y nell'albero.
Es:

P = [2, 2, 1, 2, 4, 3, 3, 9, 1]

- Per x = 9 e y = 4 → restituisce 3
- Per x = 3 e y = 6 → restituisce 1

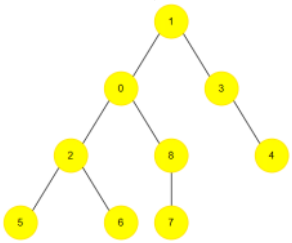


(non so perche ma partono da 1 invece che da 0 gli indici e non ce neanche il nodo 5 che dovrebbe essere collegato al nodo 4)

in realta se partiamo da 0 avremmo

P = [1, 1, 0, 1, 3, 2, 2, 8, 0]

- Per x = 8 e y = 3 → restituisce 3
- Per x = 2 e y = 5 → restituisce 1



Soluzione 3

Prendo il cammino fino alla radice di entrambi i nodi dati scorrendo il vettore dei padri fino ad arrivare alla radice e aggiungendo di padre in padre il cammino eseguito. Poi conto il numero di antenati in comune tra i due nodi e ritorno la somma dei due cammini a cui sottraggo a entrambe il numero di antenati incontrati.
Ad esempio:

- Per x = 8 → cammino_x = [1, 0, 8]
- Per y = 5 → cammino_y = [1, 0, 2, 3]
- Num. antenati in comune: 2

(Nodi cammino x - antenati in comune) + (nodi cammino y - antenati in comune) = (3 - 2) + (4 - 2) = 1 + 2 = 3. dobbiamo superare 3 nodi per andare da x a y.
Teoricamente nell'esempio il cammino sarebbe 0, 2, 5 siccome non conta il nodo di partenza. Invece il calcolo sugli antenati rimuove lo 0 (poiché antenato comune) ma conta il cammino partendo dal nodo di partenza quindi il risultato resta giusto poiché al posto del conto del nodo antenato comune rimosso viene contato il nodo di partenza.
Infatti in realta il codice conta quanti nodi distano i cammini dall'antenato comune ai nodi x e y.

```python
def es_3_grafi_3(P, x, y):
    def cammino_radice(P, x):
        cammino = []
        while P[x] != x: # finche non incontriamo la radicce
            cammino.append(x) # aggiungiamo il nodo al cammino
            x = P[x] # continuiamo sul padre del nodo
        cammino.append(x) # aggiungiamo la radice
        cammino.reverse() # invertiamo la lista
        return cammino
    camm_x = cammino_radice(P, x) # prendiamo il cammino dalla radice a x
    camm_y = cammino_radice(P, y) # prendiamo il cammino dalla radice a y
    antenati = 0
    for i in range(min(len(camm_x), len(camm_y))):
        if camm_x[i] != camm_y[i]: # appena abbiamo superato tutti gli antenati in comune
            # abbiamo antenati che indica il num. di antenati superati
            break
        antenati += 1 # incrementiamo gli antenati incontrati
    # quindi torniamo la lunghezza di entrambe le liste rimuovendo a entrambe i+1 (gli antenati superati)
    return len(camm_x)+len(camm_y) - ((antenati)*2) # per 2 poichè rimuoviamo da entrambe le liste
```

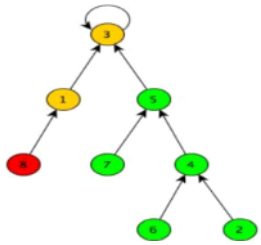
Il costo della ricerca del cammino è di O(n) e il for itera in base alla lunghezza del cammino minimo, quindi nel caso peggiore ha costo O(n).
Il costo totale è O(n)

Esercizio 4

Dato un albero di n nodi rappresentato tramite il vettore dei padri P e un suo nodo x, dare lo pseudocodice di un algoritmo che in tempo O(n) produce la lista dei nodi di T presenti nel sottoalbero radicato in x
Es:

P = [3, 4, 3, 5, 3, 4, 5, 1]

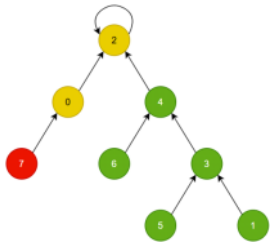
- x = 5 → restituisce {2, 4, 5, 6, 7}
- x = 8 → restituisce {8}



(pure questo parte da 1 invece che 0)
Se partiamo da 0 abbiamo

P = [2, 3, 2, 4, 2, 3, 4, 0]

- x = 4 → restituisce {1, 3, 4, 5, 6}
- x = 7 → restituisce {7}



Soluzione 4

Creo il trasposto del grafo (quindi una lista di adiacenza), e eseguo una visita DFS ricorsiva partendo dal nodo x cosi che visito tutti i nodi nella sua sottoradice.

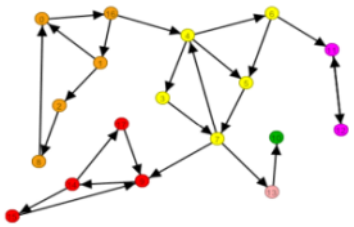
Ritorno poi il set dei nodi visitati

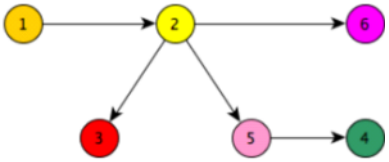
```python
def es_4_grafi_3(P, x):
    def DFS(G, x, visitati):  # DFS semplice
        visitati[x] = 1
        for y in G[x]:
            if not visitati[y]:
                DFS(G, y, visitati)
    n = len(P)
    G = [[] for _ in range(n)] # inizializza la lista di adiacenza
    for i in range(n):
        if P[i] != i: #se non è la radice
            G[P[i]].append(i) # aggiungiamo al padre il nodo figlio
    visitati = [0]*n # inizializza a 0 i visitati
    DFS(G, x, visitati) # esegue la visita dal nodo x
    return set(i for i in range(n) if visitati[i] == 1) # ritorna il set dei nodi visitati
```

L'inizializzazione della lista di adiacenza, dei visitati e il costo del trasposto è Θ(n) mentre il costo della visita è O(n+m) però poiché è un albero abbiamo che m = n-1 quindi il costo è O(n)
Costo totale: O(n)

Esercizio 5

Dato un grafo diretto G, si definisce grafo delle parti il grafo G' che contiene un vertice per ogni componente fortemente connessa di G e tra due suoi nodi a e b c'è un arco che va da a fino a b se in G è possibile andare da un nodo della componente fortemente connessa corrispondente ad a ad un nodo della componente fortemente connessa corrispondente a b





- Descrivere un algoritmo che, a partire dal grafo diretto G, costruisce il suo grafo delle parti G' in O(n + m) tempo
- Dimostrare che il grafo delle parti è sempre un DAG (grafo diretto aciclico)

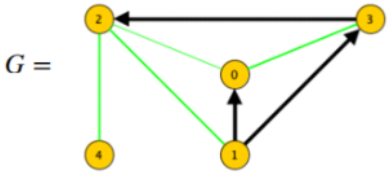
Probabilmente per ricavarlo in tempo O(n+m) richiede di usare l'algoritmo di Kosaraju o Tarjan che non abbiamo visto a lezione, siccome quello che abbiamo visto per le componenti fortemente connesse è quello con tempo $O(n^2 + n*m)$

Slide Grafi 4 (sort topologico)

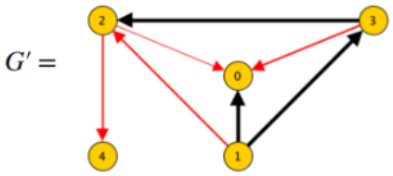
mercoledì 2 aprile 2025 12:33

Esercizio: da Grafo Parzialmente Orientato Aciclico a DAG

Un grafo è parzialmente orientato se contiene sia archi orientati che non orientati

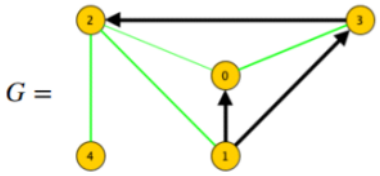


Problema: Abbiamo un grafo G parzialmente orientato e sappiamo che gli archi orientati di G non formano cicli orientati.
Vogliamo assegnare una direzione a tutti gli archi non orientati di G in modo che il grafo G' risultante risulti un DAG (vale a dire un grafo orientato aciclico).



Esercizio

1. Ideare un algoritmo che dato un grafo G parzialmente orientato senza cicli orientati ne orienta gli archi producendo un DAG. L'algoritmo deve avere complessità O(n + m).
2. Progettare la funzione python che codifica l'algoritmo per orientare il grafo G parzialmente orientato. Il grafo G è rappresentato tramite liste di adiacenza dove ad ogni nodo x, $0 \leq x < n$ è associata una coppia di liste (A, B)
   - La lista A contiene i vicini di x raggiungibili tramite gli archi diretti
   - La lista B contiene i vicini di x raggiungibili tramite gli archi non diretti




```
G=[
    ( [ ],     [2,3] ),
    ( [0,3],   [2] ),
    ( [ ],     [0,1,4] ),
    ( [2],     [0] ),
    ( [ ],     [2] )
  ]
```

Soluzione

Prendiamo il sort topologico effettuato tramite gli archi orientati e in base alla posizione dei nodi nel sort impostiamo la direzione degli archi non orientati.

- Creiamo una lista di adiacenza G_dir con solo gli archi orientati (quindi G[x][0])
- Eseguiamo il sort topologico su questa lista
- Per ogni nodo del sort, dobbiamo conoscere la sua posizione nel sort per poter capire dopo quale nodo viene prima di un altro.
  Creiamo quindi una lista SortPos per salvare le posizioni, e per ogni nodo x in pos. i nel sort, salviamo nella pos. x della lista la sua pos. i.
  Quindi infine avremmo Sort[i] = x e SortPos[x] = i
- Per ogni nodo x e per ogni suo nodo adiacente y passando per archi non orientati (quindi G[x][1]), se nel sort topologico il nodo x è prima del nodo y (SortPos[x] < SortPos[y]) allora possiamo impostare l'arco ordinato x - y nella lista di adiacenza G_dir. Altrimenti se x sta dopo y allora l'arco ordinato è in direzione y - x ma non lo aggiungiamo poiché verra aggiunto quando x sara tra gli adiacenti di y.

```python
def es_grafi_4(G):
    from lezioni import SortTopDFS, SortTopDAG
    n = len(G)
    G_dir = [[] for _ in range(n)] # inizializzo la lista di archi diretti
    for i in range(n): # per ogni nodo
        G_dir[i] = G[i][0] # prendiamo solo gli archi diretti
    GSort = SortTopDFS(G_dir) # prendiamo l'ordinamento topologico solo sugli archi ordinati
    SortPos = [-1]*n # inizializzo la lista delle posizioni
    for i in range(n): # prendiamo la posizione del nodo x nella lista dell'ordinamento topologico
        SortPos[GSort[i]] = i # in SortPos[x] avremmo la posizione di x nell'ordinamento
    for x in range(n): # per ogni nodo
        for y in range(G[x][1]): # per ogni nodo adiacente passando per archi non diretti
            if SortPos[x] < SortPos[y]: # se nell'ordinamento il nodo x è prima del nodo y
                G_dir[x].append(y) # aggiungo l'arco diretto da x a y
    return G_dir
```

- Il costo dell'inizializzazione e riempimento di G_dir e SortPos è Θ(n)
- Il costo del sort topologico è O(n + m)
- Il costo dei due for annidati è O(n + m)

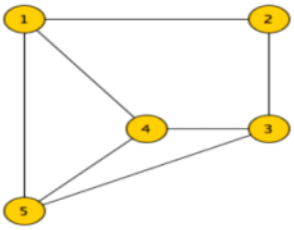
Costo totale: O(n + m)

Slide Grafi 5 (cicli)

venerdì 28 marzo 2025 16:03

Esercizio 2

Progettare un algoritmo che, dato un grafo connesso G, restituisce in tempo O(m) un cammino che attraversa tutti gli archi di G una e una sola volta in entrambe le direzioni, (i nodi possono essere toccati anche più volte).



Ad esempio per il grafo in figura che ha 7 archi una possibile soluzione è il seguente cammino di lunghezza 14:

$1-4-5-4-1-5-1-2-3-4-3-5-3-2-1$

Soluzione

IDEA: Eseguo una visita DFS ma invece di tenere conto dei nodi visitati, tengo conto degli archi attraversati

In modo simile all'algoritmo del **sort topologico tramite DFS**

- Creo un dizionario contenente come chiave gli archi distinti (x, y) del grafo (ad es. (3, 5) è diverso da (5, 3)), e come valore iniziale 0 (arco non attraversato)
- Eseguo una visita DFS a partire dal nodo 0 (essendo un grafo connesso, l'albero DFS radicato a 0 avrà tutti i nodi del grafo)
- Nella visita invece di tenere conto dei nodi visitati, tengo conto degli archi visitati (e la loro direzione). Nella visita di un nodo x:
  - Se l'arco (x, y) col suo nodo adiacente y non è stato attraversato (0) allora lo attraverso (imposto a 1) e eseguo la visita sul nodo adiacente
- Inserisco il nodo padre nel cammino solo **dopo** che eseguo la visita sui suoi nodi adiacenti

Rispetto alla lista ritornata del sort topologico, questo cammino non è necessario invertirlo poiché è possibile effettuarlo sia da sinistra che da destra e alla fine avremmo comunque attraversato tutti gli archi due volte (una per direzione).

Correttezza:

- La marcatura degli archi evita che un arco venga attraversato in una direzione più di una volta. Quindi ogni direzione dell'arco è attraversata una sola
- L'inserimento del nodo dopo la visita garantisce che gli archi uscenti dal nodo vengono attraversati per proseguire il cammino e quindi evitiamo di saltare parti del cammino e di "lasciare indietro" archi inesplorati. Infatti viene prima eseguito il cammino su tutti i nodi e poi nel "ritorno" della ricorsione vengono inseriti nella lista del cammino solo i nodi attraversati nell'arco esplorato.

```python
def es_2_grafi_6(G):
    def DFS_2(G, x, archi,cammino):
        for y in G[x]: # per ogni nodo adiacente
            if archi[(x, y)] == 0: # se l'arco in direzione x-y non è stato passato
                archi[(x, y)] = 1 # impostiamo l'arco in direzione x-y come visitato
                DFS_2(G, y, archi, cammino) # continuiamo la visita sul nodo adiacente
        cammino.append(x) # alla fine della visita aggiungiamo il nodo nel cammino
    ####
    archi = {}
    # usiamo un dizionario per controllare se un arco è stato visitato in una direzione
    for x in range(len(G)):
        for y in G[x]:
            archi[(x, y)] = 0 # inizializziamo la coppia (nodo, adiacente) con 0 (non visitato)
    # avremmo alla fine 2*m coppie distinte x-y. (es avremmo sia l'arco 3-4 che l'arco 4-3 distinti)
    cammino = []
    DFS_2(G, 0, archi, cammino) # partiamo dal nodo 0 (o anche un altro nodo del grafo va bene)
    return cammino
```

Costo

Poiché è un grafo connesso dove m ≥ n-1, i costi O(n + m) diventano O(m)

Costo:

- L'inizializzazione del dizionario degli archi ha costo O(m)
- La DFS visita ogni arco una sola volta quindi ha costo O(m)
  - La ricerca di un arco nel dizionario ha costo O(1)
  - L'append nella lista ha costo O(1)

Costo totale: O(m)

Però il prof mi ha fatto notare che nella media il costo è O(m), però nel caso in cui ci siano numerose collisioni di hash ne lla ricerca dell'elemento nel dizionario allora il costo si alzerebbe a $O(m^2)$.

Soluzione 2

Inizialmente avevo pensato a creare una seconda lista di adiacenza cosi da poterla modificare dur ante la visita. Quindi dur ante la visita di un nodo adiacente y posso muovere dalla lista di adiacenza quel nodo cosi che il nodo padre non possa piu visitarlo in seguito nella visita.

```python
def es_2_grafi_6_2(G):
    def DFS_2(G, x,cammino):
        for y in G[x]: # per ogni nodo adiacente
            G[x].remove(y) # rimuoviamo il nodo adiacente dalla lista
            DFS_2(G, y, cammino) # continuiamo la visita sul nodo adiacente
        cammino.append(x) # alla fine della visita aggiungiamo il nodo nel cammino
    ####
    G2 = []
    # creo una copia della lista di adiacenza
    for x in range(len(G)):
        G2.append([])
        for y in G[x]:
            G2[x].append(y)
    cammino = []
    DFS_2(G2, 0, cammino) # partiamo dal nodo 0 (o anche un altro nodo del grafo va bene)
    return cammino
```

Il problema è che nel caso peggiore il nodo da rimuovere può trovarsi nella prima posizione quindi la rimozione chiederebbe d i spostare indietro massimo n elementi portando il costo della rimozione a O(n).

Però invece di usare una lista di adiacenza uguale posso utilizzare una lista di adiacenza che ha delle flag per indicare se un nodo adiacente è stato visitato o meno. Quindi ogni nodo x avrà due liste di adiacenza diverse: quella classica che indica i nodi adiacenti y e quella delle flag che indica se il nodo adiacente y è stato visitato (1) partendo da x oppure no (0).

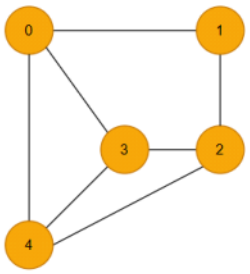
- Creo una lista con lo stesso numero di elementi della lista di adiacenza però invece di contenere i nodi adiacenti, contiene una flag indicante la visita su quel nodo partendo dal padre
- Faccio partire la visita DFS dal nodo 0 del grafo
- Durante la visita del nodo x, per ogni nodo adiacente y e il suo indice nella lista di adiacenza, controllo se non è stato già visitato (quindi l'arco x-y non è stato attraversato).
- Se posso attraversare quell'arco, imposto la visita sul nodo adiacente a 1 (cosi che non possa piu attraversare quell'arco in quella direzione) e continuo la visita sul nodo adiacente

```python
def es_2_grafi_6_2(G):
    def DFS_2(G, x, visitati, cammino):
        for i, y in enumerate(G[x]): # per ogni nodo adiacente e il suo indice nella lista
            if visitati[x][i] == 0: # se non è stato visitato partendo da x
                visitati[x][i] = 1 # passiamo sull'arco in direzione x-y
                DFS_2(G, y, visitati, cammino) # continuiamo la visita sul nodo adiacente
        cammino.append(x) # alla fine della visita aggiungiamo il nodo nel cammino
    ####
    visitati = [0]*len(G) # creo le n liste dei nodi
    for x in range(len(G)):
        visitati[x] = [0]*len(G[x]) # inizializzo a 0 i nodi adiacenti
    cammino = []
    DFS_2(G, 0, visitati, cammino) # partiamo dal nodo 0 (o anche un altro nodo del grafo va bene)
    return cammino
```

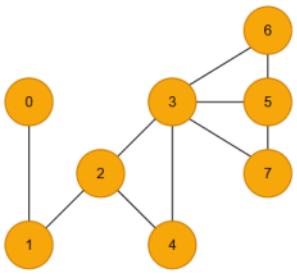
Costo

Rispetto al codice precedente, stavola invece di un dizionario utilizzo una lista semplice ad accesso diretto quindi non si h anno problemi di collisioni di hash e ogni accesso alla flag del nodo è in tempo costante O(1).
L'inizializzazione della lista "visitati" e il costo della visita DFS hanno entrambi costo O(m) (sempre dovuto al fatto che n el grafo connesso si ha m $\geq$ n-1)
Costo totale: O(m)

Risultato.



0 - 4 - 3 - 4 - 2 - 3 - 2 - 1 - 2 - 4 - 0 - 3 - 0 - 1 - 0



0 - 1 - 2 - 4 - 3 - 7 - 5 - 6 - 5 - 7 - 3 - 6 - 3 - 5 - 3 - 4 - 2 - 3 - 2 - 1 - 0

Altre Versioni

Si può anche utilizzare la formula arco = (min(x, y), max(x, y)) cosi da utilizzare una entry sola nel dizionario per indicar e le due direzioni dell'arco.
Questo è utile solo per abbassare la grandezza del dizionario da 2*m a m e comunque accedere in tempo O(1) all'arco nel dizio nario.
Bisogna però cambiare il controllo dell'attraversamento dell'arco in modo tale che sia visitato almeno 2 volte. Quindi inizia lizziamo gli archi nel dizionario a 0 e durante la visita se l'arco viene attraversato incrementiamo il suo valore. Se l'arco è gia stato attraversato 2 volte allora non possiamo piu attraversarlo.

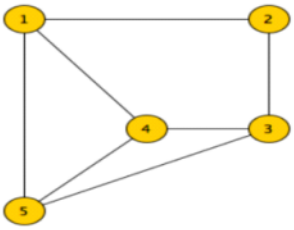
```python
def es_2_grafi_6_1(G):
    def DFS_2(G, x, archi,cammino):
        for y in G[x]: # per ogni nodo adiacente
            arco = (min(x, y), max(x, y))
            if archi[arco] < 2: # se l'arco in direzione x-y è stato attraversato 0 o 1 volta
                archi[arco] += 1 # incrementiamo il num. di visite sul nodo
                DFS_2(G, y, archi, cammino) # continuiamo la visita sul nodo adiacente
        cammino.append(x) # alla fine della visita aggiungiamo il nodo nel cammino
    ####
    archi = {}
    # usiamo un dizionario per controllare se un arco è stato visitato in una direzione
    for x in range(len(G)):
        for y in G[x]:
            # inseriamo gli archi x-y e y-x come unico arco
            archi[(min(x, y), max(x, y))] = 0 # inizializziamo la coppia x-y o y-x con 0 (non visitato)
    # avremmo alla fine m coppie distinte x-y. (in questo caso l'arco 3-4 e l'arco 4-3 indicano lo stesso arco)
    cammino = []
    DFS_2(G, 0, archi, cammino) # partiamo dal nodo 0 (o anche un altro nodo del grafo va bene)
    return cammino
```

Slide Grafi 5 finale (cicli)

venerdì 28 marzo 2025 16:03

Esercizio 2

Progettare un algoritmo che, dato un grafo connesso G, restituisce in tempo O(m) un cammino che attraversa tutti gli archi di G una e una sola volta in entrambe le direzioni, (i nodi possono essere toccati anche più volte).



Ad esempio per il grafo in figura che ha 7 archi una possibile soluzione è il seguente cammino di lunghezza 14:

$1-4-5-4-1-5-1-2-3-4-3-5-3-2-1$

Soluzione

IDEA: Eseguo una visita DFS ma invece di tenere conto dei nodi visitati, tengo conto degli archi attraversati

Utilizzo una lista simile a quella di adiacenza che, invece di avere i nodi adiacenti, contiene delle flag per indicare se un nodo adiacente è stato visitato o meno. Quindi ogni nodo x avrà due liste di adiacenza diverse: quella classica che indica i nodi adiacenti y e quella delle flag che indica se il nodo adiacente y è stato visitato (1) partendo da x oppure no (0).

- Creo una lista con lo stesso numero di elementi della lista di adiacenza però invece di contenere i nodi adiacenti, contiene una flag indicante la visita su quel nodo partendo dal padre
- Faccio partire la visita DFS dal nodo 0 del grafo
- Durante la visita del nodo x, per ogni nodo adiacente y e il suo indice nella lista di adiacenza, controllo se non è stato già visitato (quindi l'arco x-y non è stato attraversato).
- Se posso attraversare quell'arco, imposto la visita sul nodo adiacente a 1 (cosi che non possa piu attraversare quell'arco in quella direzione) e continuo la visita sul nodo adiacente

```
def es_2_grafi_6_2(G):
    def DFS_2(G, x, visitati, cammino):
        for i, y in enumerate(G[x]): # per ogni nodo adiacente e il suo indice nella lista
            if visitati[x][i] == 0: # se non è stato visitato partendo da x
                visitati[x][i] = 1 # passiamo sull'arco in direzione x-y
                DFS_2(G, y, visitati, cammino) # continuiamo la visita sul nodo adiacente
        cammino.append(x) # alla fine della visita aggiungiamo il nodo nel cammino
    ####
    visitati = [0]*len(G) # creo le n liste dei nodi
    for x in range(len(G)):
        visitati[x] = [0]*len(G[x]) # inizializzo a 0 i nodi adiacenti
    cammino = []
    DFS_2(G, 0, visitati, cammino) # partiamo dal nodo 0 (o anche un altro nodo del grafo va bene)
    return cammino
```

Correttezza:

- La marcatura degli archi evita che un arco venga attraversato in una direzione più di una volta. Quindi ogni direzione dell'arco è attraversata una sola
- L'inserimento del nodo dopo la visita garantisce che gli archi uscenti dal nodo vengono attraversati per proseguire il cammino e quindi evitiamo di saltare parti del cammino e di "lasciare indietro" archi inesplorati. Infatti viene prima eseguito il cammino su tutti i nodi e poi nel "ritorno" della ricorsione vengono inseriti nella lista del cammino solo i nodi attraversati nell'arco esplorato.

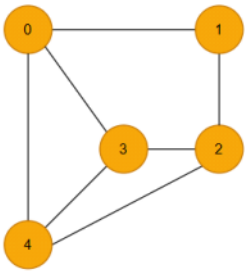
Costo

Rispetto al codice precedente, stavola invece di un dizionario utilizzo una lista semplice ad accesso diretto quindi non si h anno problemi di collisioni di hash e ogni accesso alla flag del nodo è in tempo costante O(1).

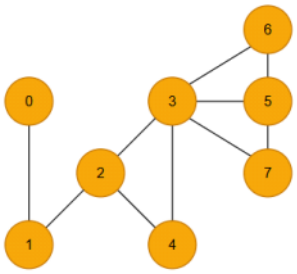
L'inizializzazione della lista "visitati" e il costo della visita DFS hanno entrambi costo O(m) (sempre dovuto al fatto che n el grafo connesso si ha m $\geq$ n-1)

Costo totale: O(m)

Risultato.



0 - 4 - 3 - 4 - 2 - 3 - 2 - 1 - 2 - 4 - 0 - 3 - 0 - 1 - 0



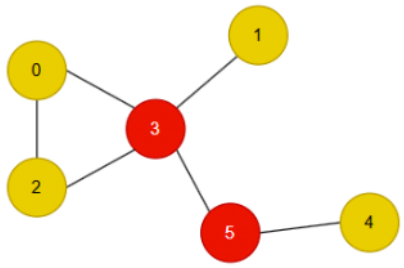
0 - 1 - 2 - 4 - 3 - 7 - 5 - 6 - 5 - 7 - 3 - 6 - 3 - 5 - 3 - 4 - 2 - 3 - 2 - 1 - 0

Slide Grafi 6 (Ponti)

mercoledì 2 aprile 2025 14:42

Esercizio 1

un vertice la cui rimozione è in grado di sconnettere il grafo è detto **punto di articolazione.**
quindi se c'è un cammino da x a y che passa per quel nodo, se esso viene rimosso non esiste un altro cammino da x a y.
Ad esempio, nella figura i nodi 3 e 5 sono punti di articolazione perche se viene rimosso uno dei due, il grafo perde la proprietà di connessione



1. Dimostrare o confutare che esiste un grafo G con la proprietà di avere un arco la cui rimozione fa aumentare di 2 il numero dei suoi punti di articolazione e di 3 il numero dei suoi ponti.
2. Dimostrare o confutare che non esistono grafi i cui nodi sono tutti punti di articolazione.
3. Dimostrare o confutare che in un grafo connesso G esiste sempre un ordinamento dei suoi nodi tale che la rimozione dei nodi in quell'ordine mantiene sempre il grafo connesso. In caso questa sequenza esista progettare un algoritmo che la fornisce in tempo O(m).

Sia G un grafo connesso con m archi.

4. dimostrare che il nodo radice dell'albero DFS di G è un punto di articolazione di G se e solo se ha almeno due figli.
5. dimostrare che un nodo x dell'albero DFS di G diverso dalla radice è un punto di articolazione di G se e solo se non esiste in G un arco (a, b) tale che nell'albero DFS di G a è un discendente di x e b un antenato di x.
6. Progettare un algoritmo che dato il grafo G restituisce la lista dei suoi nodi di articolazione in tempo O(m).

Soluzioni 1

1. Siccome l'eliminazione di un ponte toglierebbe la proprietà di connessione al grafo, dobbiamo vedere la richiesta si può raggiungere se rimuoviamo un arco non critico. Se andiamo a rimuovere un arco non critico x-y, significa che, oltre a quell'arco, esiste un cammino C da x a y alternativo.
   Dopo la rimozione dell'arco x-y, abbiamo due casi:
   - Se x e y diventano due punti di articolazione, significa che non c'è un cammino alternativo da x o da y ai nodi del cammino C, però cosi anche i nodi del cammino diventano nuovi punti di articolazione, quindi il numero di punti di articolazione aumenta di > 2
   - Se x e y non diventano due punti di articolazione, allora c'è un cammino alternativo da x o da y ai nodi del cammino C, però:
     - Se il cammino C ha due nodi (diversi da x e y) si creeranno due nuovi punti di articolazione, ma un solo nuovo ponte (quello tra i due nodi di C)
     - Se il cammino C ha 4 nodi (diversi da x e y) si creeranno 3 nuovi ponti ma 4 nuovi punti di articolazione

   Quindi non è possibile aumentare di 2 i punti di articolazione e di 3 i ponti, rimuovendo un solo arco
2. Un punto di articolazione è un nodo la cui rimozione aumenta il num. di componenti connesse del grafo. Quindi se esiste un punto di articolazione, allora esistono almeno due componenti, quindi il num. minimo di nodi del grafo per avere un punto di articolazione è 3 (catena di 3 nodi dove quello centrale è il punto di articolazione)
   Per avere il numero massimo di punti di articolazione bisogna che non ci siano cicli nel grafo poiché la rimozione di un nodo dal ciclo può aumentare il num. di componenti, però i nodi nel ciclo resteranno un componente unico, quindi non si può raggiungere il massimo di punti di articolazione.
   Se invece abbiamo una catena connessa di $n \geq 3$ nodi, i due nodi finali (primo e ultimo della catena) non sono punti di articolazione poiché la loro rimozione non aumenta il numero di componenti. Invece gli n-2 nodi restanti della catena saranno tutti punti di articolazione. Quindi il numero massimo di punti di articolazione di un grafo è n-2, quindi non esistono grafi in cui tutti i nodi sono punti di articolazione
3. Se rimuoviamo i nodi in ordine crescente del loro grado, rimuoveremo prima i nodi agli estremi con un solo arco, poi i nodi con 2 archi (senza contare quelli rimossi) e cosi via. Ad esempio in una catena con $n \geq 3$ nodi, in cui abbiamo n-2 punti di articolazione, rimuovendo prima i due nodi estremi di grado 1, rimarremmo con una nuova catena connessa formata dagli n-2 nodi restanti, dove ci saranno altri due nuovi nodi estremi di grado 1 da poter rimuovere senza togliere la proprietà di connessione della catena, e cosi via fino a rimuovere tutti i nodi della catena.
   Idea algoritmo:
   - Eseguiamo una visita DFS semplice partendo dal nodo con grado maggiore pero con l'inserimento dei nodi in post-order, quindi prima scendiamo in profondita agli nodi estremi del grafo (quelli con grado 1) e poi nel ritorno inseriamo i nodi nella lista da ritornare.

```python
def es_1_3_grafi_6_1(G):
    def SortNodi(G, x, visitati, ordinamento):
        visitati[x] = 1 # visitiamo il nodo
        for y in G[x]:
            if not visitati[y]: # per ogni adiacente non visitato
                SortNodi(G, y, visitati, ordinamento) # continuiamo la visita
        ordinamento.append(x) # dopo la fine della visita in profondità aggiungiamo il nodo corrente
    n = len(G)
    max_nodo = -1
    max_grado = 0
    for x in range(n):
        if len(G[x]) > max_grado: # se ha un num. di adiacenti maggiore di quello gia salvato
            max_grado = len(G[x]) # aggiorniamo il nuovo max
            max_nodo = x # prendiamo il nodo col maggior numero di adiacenti
    visitati = [0]*n # inizializza la lista dei nodi visitati
    ordinamento = []
    SortNodi(G, max_nodo, visitati, ordinamento) # eseguiamo la DFS partendo dal nodo con grado maggiore
    return ordinamento
```

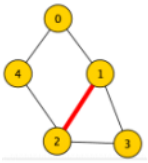
   Il costo è quello di una visita DFS quindi O(n + m) che si riduce a O(m) poiché nei grafi connessi $m \geq n-1$
4. Come detto dal punto 2, il punto di articolazione, se rimosso, aumenta il num. di componenti, e se esiste nel grafo, allora ci devono essere almeno due componenti (quelle da poter dividere). Quindi un nodo per essere punto di articolazione deve avere almeno grado ≥ 2. Quindi se la radice dell'albero ha almeno 2 figli allora è punto di articolazione, poiché quando viene rimosso separa i rami dell'albero tra di loro. Se avesse avuto un solo figlio allora alla sua rimozione non avrebbe aumentato il numero di componenti del grafo ma avrebbe fatto diventare il figlio la radice stessa.
5. Dobbiamo dimostrare l'equivalenza:
   $\Rightarrow$ Se x è punto di articolazione, allora non esiste in G arco (a, b) t.c. nell'albero DFS a è discendente di x e b è antenato di x.

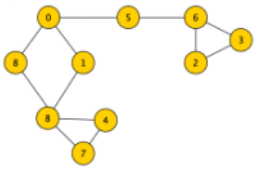
Supponiamo per assurdo che x sia punto di articolazione e che esiste in G un arco (a, b) t.c nell'albero DFS a è discendente di x e b è antenato di x. Questo significa tra i componenti collegati ad x esiste un cammino alternativo che passa per (a, b) ma non per x. Quindi rimuovendo x, i componenti sarebbero comunque collegati dal cammino che passa per (a, b) il che è assurdo poiché contraddice l'ipotesi che x è punto di articolazione.
⇐ Se non esiste in G l'arco (a, b) t.c. nell'albero DFS a è discendente di x e b è antenato di x, allora x è punto di articolazione
Se cosi fosse, significa che nessun nodo del sottoalbero di x può raggiungere (attraverso un arco non nell'albero DFS) un antenato di x senza passare per x. Quindi se x venisse rimosso, il grafo si disconnette tra i rami di x e gli antenati di x, aumentando il numero di componenti connesse. Pertanto x è un punto di articolazione.

6.

Esercizio 2

Un grafo cactus è un grafo connesso non orientato in cui ogni **arco** appartiene al massimo a un **ciclo semplice**. In altre parole, in un grafo cactus, due cicli distinti possono avere al massimo un vertice in comune, ma non possono condividere archi.
Di seguito due grafi: quello di sinistra non è un cactus, quello di destra si.





1. Dimostrare che un grafo cactus è un grafo sparso provando che se il grafo ha n nodi allora il num. dei suoi archi non può superare $\left\lfloor \frac{3(n-1)}{2} \right\rfloor$
2. Progettare un algoritmo che, dato un grafo non orientato e connesso G di n nodi, rappresentato tramite liste di adiacenza, determina se G è un grafo cactus o meno. L'algoritmo proposto deve avere complessità O(n)

Soluzioni 2

1.
2. Siccome in un grafo cactus il massimo numero di nodi è $\left\lfloor \frac{3(n-1)}{2} \right\rfloor$ prima controlliamo se il numero di archi non supera il numero di nodi.
   Poi possiamo passare a controllare se non abbiamo nodi che condividono archi: Eseguo un controllo simile a quello della ricerca dei cicli
   - Creo una lista dei nodi visitati
   - Creo una lista Archi simile alla lista di adiacenza dove per ogni nodo x abbiamo una lista di lunghezza y (dove y è il num. di nodi adiacenti di x) di interi, che indicano se l'arco da x a y è stato attraversato (1) o meno (0). Per ogni nodo x e ogni arco y adiacente imposto inizialmente Archi[x][y] = 0
   - Eseguo inizialmente una DFS dal nodo 0 (siccome il grafo è connesso arrivero ad ogni nodo del grafo)
   - Per ogni nodo adiacente y del nodo x che sto visitando
     - Se il nodo y non è stato visitato (0) allora imposto come visitato il nodo e l'arco x-y e y-x (1) e continuo la visita DFS
     - Se il nodo y è gia stato visitato (1) ma l'arco x-y e y-x no (0) allora significa che sono tornato ad un nodo condiviso tra cicli ma l'arco non è condiviso tra cicli. Quindi imposto l'arco come attraversato (1) e continuo la visita DFS sugli altri adiacenti.
     - Se il nodo y è stato visitato completamente (2) e l'arco x-y e y-x è gia stato visitato (1) allora significa che sto passando per un nodo che ha gia finito i nodi adiacenti e l'arco x-y è gia stato attraversato passando da y a x, quindi l'arco è condiviso tra piu cicli e quindi il grafo non è un grafo cactus e posso ritornare False
   - Alla fine del ciclo degli adiacenti imposto il nodo come completamente visitato (2)

Slide Grafi 8 (BFS)

mercoledì 2 aprile 2025 18:24

Esercizio 1

Descrivere un algoritmo che, dato un grafo G non diretto e connesso e due suoi nodi u e v, in tempo O(n + m) trova i nodi che hanno la stessa distanza da u e v.

Soluzione 1

Eseguiamo separatamente la visita BFS dai nodi u e v e ricaviamo due array delle distanze dei nodi da u e v. Iteriamo entrambe le liste allo stesso tempo e se gli elementi in posizione i nelle liste hanno la stessa distanza allora lo aggiungiamo alla lista di nodi da ritornare.

```
def es_1_grafi_8(G, u, v):
    from lezioni import BFSDistanze
    distanze_u = BFSDistanze(G, u) # prendiamo le distanze da u
    distanze_v = BFSDistanze(G, v) # prendiamo le distanze da v
    equi = []
    for i in range(len(G)): # entrambe le liste hanno lo stesso num. di elementi siccome è un grafo connesso
        if distanze_u[i] == distanze_v[i]: # se entrambi i nodi hanno la stessa distanza da u e da v
            equi.append(i) # lo ritorniamo
    return equi
```

Il costo delle due visite separate è O(n + m)

Esercizio 2

Che caratteristiche deve avere un grafo connesso perch´e le visite DFS e BFS del grafo producano alberi di visita uguali? Motivare la risposta.

Soluzione 2

La DFS esegue le visite in profondità andando fino alla fine di un cammino prima di tornare indietro su altri cammini. La BFS invece esegue la visita a livelli partendo dalla radice, quindi prima visita i nodi adiacenti a distanza 1, poi 2, e cosi via fino a visitare tutti i nodi in base alle distanze dalla radice.
Per produrre lo stesso albero di visita dalla DFS e BFS bisogna:

- O partire da un nodo connesso agli altri n-1 nodi e quindi l'albero sara radicato su quel nodo e avra n-1 foglie
- O se il grafo ha un solo nodo e quindi si avra un albero con solo la radice
- O effetture la visita partendo da un estremo di una catena di n nodi e quindi si scendera in entrambe le visite un nodo alla volta fino alla fine della catena

Quindi non si possono avere cicli, poiché se c'è un collegamento diverso per arrivare da un nodo x ad un altro nodo y, la BFS scendera a livelli partendo da x e visitando prima gli adiacenti fino ad arrivare ad y, mentre la DFS scegliera uno dei due cammini e dopo essere passato per y tornera a x tramite l'altro cammino però nel senso opposto prima di tornare indietro poiché i nodi nell'altro cammino non sono stati visitati.
Non si possono avere nodi con grado maggiore di due e dove gli adiacenti hanno grado maggiore di 1 poiché se partiamo dal nodo con grado maggiore di 2 e visitiamo i nodi adiacenti, la DFS visitera in profondita uno dei cammini che partono dal nodo adiacente prima di visitare gli altri nodi adiacenti, mentre la BFS visitera a livelli tutti i cammini parallelamente.
Se non abbiamo cicli, e quindi è una catena, non possiamo partire da un nodo centrale con grado 2 poiché in quel caso la DFS prenderebbe uno dei due cammini e arriverebbe all'estremo del cammino prima di tornare indietro per visitare l'altro cammino non visitato, mentre la BFS visiterebbe entrambi i cammini un nodo alla volta.

Esercizio 3

Progetta un algoritmo che, dato un grafo G, in tempo O(n) verifica se `e un albero o meno.

Soluzione 3

Se è un albero allora il num. di archi è m = n-1 quindi se il num. di archi supera n-1 allora non è un albero.
Abbiamo due modi per effettuare il controllo
Metodo 1:

- Un albero deve essere connesso quindi dobbiamo controllare che il num. di nodi raggiungibili da un nodo qualunque sia n. Per fare ciò possiamo eseguire una BFS e controllare il num. di nodi visitati e se è inferiore a n allora non è un albero.
- Poiché abbiamo gia controllato che m = n-1 abbiamo che il costo della BFS invece di essere O(n + m) diventa O(n) quindi il costo è valido per la richiesta

Metodo 2:

- Effettuiamo una visita BFS con l'inserimento dei padre e controlliamo che, se durante la visita arriviamo ad un nodo gia visitato che non è il padre, allora c'è un arco all'indietro e quindi c'è un ciclo e quindi non è un albero.
- Se invece arriviamo ad un nodo non ancora visitato allora gli assegnamo un padre e continuiamo la visita
- Nel caso peggiore se dopo la visita dell'intero albero attraversiamo un arco in piu esso ci riportera ad un nodo gia visitato quindi abbiamo trovato un ciclo e possiamo uscire dal while direttamente poiché sappiamo che il grafo non è un albero. Siccome abbiamo che in un albero m = n-1, nel caso peggiore dopo aver attraversato n-1 archi per visitare l'intero albero, se attraversiamo un altro arco allora incontriamo ciclo e usciamo dal ciclo, quindi il costo della visita nel caso peggiore è O(n)

```
def es_3_grafi_8(G):
    n = len(G)
    P = [-1]*n
    P[0] = 0 # visitiamo il nodo di partenza
    coda = [0] # inseriamo il nodo di partenza nella coda per far partire da lui la visita
    i = 0 # inizializziamo il puntatore della coda
    while len(coda) > i: # finche non superiamo il puntatore
        x = coda[i] # prendiamo il primo elem. della coda
        i += 1 # incrementiamo il puntatore
        for y in G[x]: # per ogni adiacente
            if P[y] == -1: # se non è stato visitato
                P[y] = x # gli assegnamo il padre
                coda.append(y) # e lo visitiamo
            elif P[x]!= y: # se è stato gia visitato e non è il nodo padre allora c'è ciclo
                return False # quindi il grafo non è albero
    return True # se non ci sono stati intoppi allora è un albero
```

Esercizio 4

Dato un grafo connesso e aciclico G vogliamo trovare in tempo O(n) il suo diametro (vale a dire la lunghezza del cammino pi`u lungo tra due nodi di G).

- Progetta un algoritmo basato sulla visita DFS del grafo.
- Prova la correttezza del seguente algoritmo basato su due visite BFS (o produci un controesempio)
    a. effettua una visita BFS a partire dal nodo 0 alla ricerca del nodo x pi`u lontano

b. effettua una visita BFS a partire dal nodo x alla ricerca del nodo y pi`u lontano
c. restituisci come diametro la distanza del nodo x dal nodo y.

Soluzione 4

Essendo un grafo connesso e aciclico esso è un albero DAG quindi il cammino più lungo tra due nodi è la somma tra le distanze massime a cui arrivano i sottoalberi della radice.
Siccome la radice può avere numerosi sottoalberi, il modo piu facile per eseguire questo controllo e tramite due visite DFS e due vettori di distanza.

- Inizializziamo il primo vettore di distanza D per ogni nodo a -1
- Effettuiamo la prima visita DFS partendo dal nodo 0 per trovare il nodo più distante dalla radice.
  - Per ogni nodo y adiacente di x che non è stato visitato (D[y] == -1), aggiorniamo la sua distanza come quella del padre D[x] + 1 (l'arco attraversato x-y).
  - Continuiamo la visita sul nodo y
- Cerchiamo nel vettore delle distanze da 0, il nodo x con la distanza maggiore e ritorniamo il nodo e la sua distanza da 0
- Effettuiamo poi la seconda visita sul nodo x, quindi quello con la distanza maggiore da 0.
- Dobbiamo ricreare un secondo vettore di distanze per questo nodo. Solo che stavolta invece di inizializzare la distanza di x a 0, la inizializziamo a 1 poiché nella distanza del cammino finale dobbiamo contare anche x nel cammino.
- Eseguiamo la visita DFS in modo simile a prima, cosi che stavolta troviamo il nodo più distante dal nodo in max.
- Come prima cerchiamo nel vettore delle distanze da x, il nodo y con la distanza maggiore e ritorniamo il nodo e la sua distanza da x
- A questo punto la distanza da x a y è il diametro del grafo

```python
def es_4_grafi_8(G):
    def DFS(G, x, D):
        for y in G[x]: # per ogni nodo adiacente
            if D[y] == -1: # se non è stato visitato
                D[y] = D[x] + 1 # incrementiamo la sua distanza dalla radice
                DFS(G, y, D) # continuiamo la visita
    def NodoPiuDistante(D):
        max_d = (-1, -1) # max_d[0] è il nodo, max_d[1] è la distanza
        for i in range(len(D)):
            if D[i] > max_d[1]: # se la distanza per quel nodo è maggiore di quella salvata
                max_d = (i, D[i]) # aggiorniamo la tupla con la distanza maggiore e il nodo piu distante
        return max_d
    n = len(G)
    D_rad = [-1]*n
    D_rad[0] = 0 # inizializziamo la distanza per la radice 0
    DFS(G, 0, D_rad) # effettuiamo la prima visita da 0
    max_d = NodoPiuDistante(D_rad) # cerchiamo il nodo x più distante da 0
    D_max = [-1]*n
    D_max[max_d[0]] = 1 # inizializziamo la distanza per la radice x perche nel cammino dobbiamo contare anche il nodo x
    DFS(G, max_d[0], D_max) # effettuiamo la seconda visita sul nodo x (foglia più distante da 0)
    max_d2 = NodoPiuDistante(D_max) # cerchiamo il nuovo nodo più distante da x
    return max_d2[1] # il diametro è la distanza tra x e il nuovo nodo più distante da x
```

Siccome non sappiamo se 0 è la radice vera dell'albero (quella che è antenato comune di tutti i nodi) quindi la prima visita DFS su 0 ci porterà al nodo x più distante da 0, quindi alla foglia x del primo o secondo ramo col cammino maggiore dalla radice, siccome 0 potrebbe essere nel ramo col cammino maggiore.
Siccome appunto non sappiamo se 0 è il nodo più distante da x (poiché se 0 ha sottoalberi allora c'è un nodo con un cammino più lungo) effettuiamo da x una seconda visita per trovare il nodo y più distante da esso. Infine la distanza da x a y sarà il diametro del grafo.
Quindi dopo il primo albero radicato in 0 potrebbe avere piu rami, e in quel caso prendiamo il cammino da 0 alla foglia x del ramo col cammino massimo.
Il secondo albero radicato in x avrà un solo ramo (poiché x era foglia dell'albero radicato in 0 non aveva sottoalberi ma solo un padre) e tramite la seconda visita DFS su x cerchiamo il nuovo nodo foglia y più distante da x. Questa distanza sarà infine il diametro del grafo

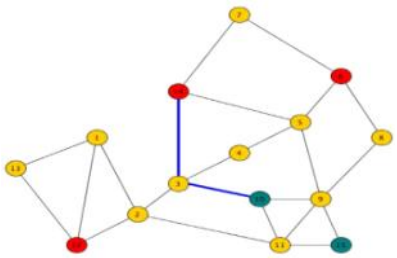
Costo:

- L'inizializzazione dei due vettori di distanza è Θ(n)
- Le due visite DFS sono effettuate separatamente e entrambe hanno costo O(n + m) però poiché ci troviamo in un albero abbiamo che m = n-1 quindi il costo è O(n)

Costo totale: O(n)

Esercizio 5

Dato un grafo G e due sottoinsiemi V1 e V2 dei suoi vertici si definisce la distanza tra V1 e V2 la distanza minima per andare da un nodo V1 ad un nodo V2.
Nel caso in cui V1 e V2 non sono disgiunti allora il val è 0.
Descrivere un algoritmo che, dato un grafo G e i due sottoinsiemi dei vertici V1 e V2 calcola la loro distanza. L'algoritmo deve avere complessita O(n+m)
Ad esempio per il grafo in figura, dove i nodi dell'insieme A sono in verde mentre i nodi dell'insieme B sono in rosso, la distanza tra i due insiemi è 2 come evidenziato dal cammino in blu



Slide Grafi Pesati

mercoledì 2 aprile 2025 18:32

Esercizio 2

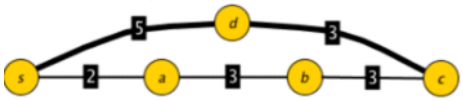
Abbiamo un grafo connesso G di n nodi, i nodi sono di due tipi: ”pericolosi” o meno. Un vettore binario P di n componenti per mette di individuare i nodi pericolosi (P[i] = 1 se e solo se i `e pericoloso).

- Progettare un algoritmo che, dato G, un suo nodo s ed il vettore P, restituisce in tempo O(n + m) una lista con tutti i nodi irragiungibili da i senza dover passare per nodi pericolosi.
- Progettare un algoritmo che, dato G, un suo nodo s ed il vettore P, genera in tempo O(n + m) un albero dei cammini radicato in s di costo minimo di G dove il costo di un cammino `e dato dal numero di nodi pericolosi toccati lungo il cammino.

Soluzione 2

Esercizio 3

Un cammino da un nodo u ad un nodo v si dice **super-minimo** se ha peso minimo tra tutti i cammini da u a v e inoltre tra tutti i cammini di peso minimo da u a v ha il minimo num. di archi. Dato un grafo pesato G t.c. i pesi sono interi positivi, si vogliono trovare i cammini super -minimi da un nodo s.
Ad esempio in questo grafo vogliamo il cammino (s, d, c) e non quello di pari peso ma più lungo (s, a, b, c)



Mostrare come modificare i pesi del grafo G in modo tale che applicando Dijkstra al grafo coi nuovi pesi si ottengono i cammi ni super minimi di G.

Min albero copertura

mercoledì 2 aprile 2025 18:37

Esercizio 1

Sia G un grafo connesso e pesato con pesi positivi. Sia T un suo minimo albero di copertura e Ts il suo albero dei cammini mi nimi a partire da un suo nodo s. Provare o confutare che la somma dei pesi degli archi di T `e uguale alla somma dei pesi degli archi di Ts.

Soluzione 1

Esercizio 2

Sia G un grafo non diretto, connesso e pesato dove i pesi sono tutti diversi tra loro. Sia T un minimo albero di copertura di G, sia s un nodo e Ts l'albero dei cammini minimi da s verso tutti gli altri nodi di G. Dimostrare oppure fornire un controesempio che Ts e T condividono almeno un arco .

Soluzione 2

Esercizio 3

Sia G un grafo non diretto, connesso e pesato, e G0 il grafo non diretto,connesso e pesato che si ottiene da G moltiplicando il peso di ciascun arco per un a fissata costante c > 0. Sia T un albero di copertura di costo minimo per G e Ts l'albero dei cammini costruito a partire dal nodo s p er G. Dimostrare o confutare che i due alberi sono di copertura minima e dei cammini minimi anche per il grafo G0 .

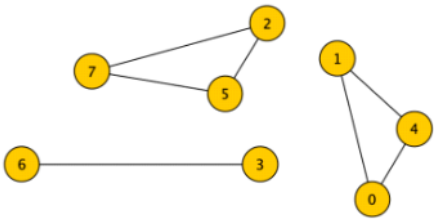
Soluzione 3

Tracce esercizi 2025

giovedì 27 marzo 2025 16:43

Esercizio 1

Progettare un algoritmo che, dato un grafo non orientato G rappresentato tramite liste di adiacenza, restituisca una lista di archi in numero minimo necessari per rendere il grafo connesso. Se il grafo è già connesso la lista sarà vuota. Se invece il grafo ha c > 1 componenti connesse, l a lista conterrà c-1 archi.
Ad esempio, per il grafo in figura, che presenta 3 componenti connesse, una possibile soluzione è la lista [(1,5), (7, 6) .
L'algoritmo proposto deve avere tempo di calcolo Θ(n + m) dove n ed m sono i nodi e gli archi del grafo, rispettivamente.
Motivare BENE la correttezza e la complessità dell'algoritmo proposto.



Soluzione 1

Per ogni nodo eseguo una visita DFS semplice e utilizzo una var di appoggio "comp_prec" per mantenere la radice sul quale abb iamo effettuato la DFS in precedenza. Inizialmente "comp_prec" è inizializzato a -1 e verra aggiornato al nodo 0 poiché è il primo su cui effettuiamo la DFS.
Siccome non conto i nodi già raggiunti dalla DFS, quando incontro un nodo non ancora visitato, significa che sto visitando un nuovo componente del grafo. In questo caso posso creare un arco per collegare la radice della componente precedente (salvata in "comp_prec") e il nodo corre nte che ancora non ho visitato (della nuova componente). Aggiungo questo nuovo arco alla lista da ritornare, aggiorno la radice della componente precedente col nodo corrente e poi eseguo la visita in questo nuovo componente.
Dopo aver visitato tutti i nodi posso ritornare la lista degli archi aggiunti

```python
def traccia_1(G): # archi minimi per collegare i componenti
    def DFS_comp(G, x, visitati): # DFS semplice
        visitati[x] = 1
        for y in G[x]:
            if visitati[y] == 0:
                DFS_comp(G, y, visitati)

    visitati = [0]*len(G)
    comp_prec = -1 # inizializzo il nodo componente precedente
    archi_min = []
    for x in range(len(G)): # per ogni nodo
        if visitati[x] == 0: # non visitato
            if comp_prec != -1: # se abbiamo già un componente precedente
                archi_min.append((comp_prec, x))
                # allora colleghiamo il componente precedente al nodo corrente
            comp_prec = x # aggiorniamo il componente precedente al nodo corrente
            DFS_comp(G, x, visitati) # eseguo la visita DFS semplice
    return archi_min
```

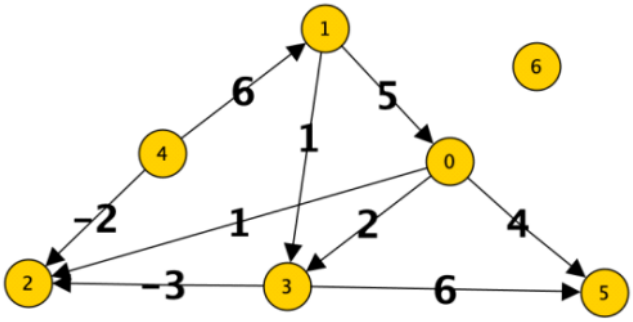
Ad ogni visita della DFS impostero come visitati i nodi delle diverse componenti. Quando incontro un nodo non visitato allora sto per visitare un nuovo componente. Allora collego il componente precedente (dalla radice) al nuovo nodo del nuovo componente. L'aggiunta del nodo cr eerebbe una nuova componente piu grande, quindi visitando tutti i nodi e tutte le componenti diverse, collegerei con un solo nodo le diverse radici delle componenti.
Il costo è quello di una visita DFS quindi O(n+m)
La soluzione dell'algoritmo in base al grafo nell'immagine è [(0, 2), (2, 3)]

Esercizio 2

Progettare un algoritmo che, dato un DAG pesato G rappresentato mediante liste di adiacenza ed un suo vertice sorgente s, res tituisca il vettore delle distanze dei nodi da s.
Ad esempio per il DAG in figura:

- per s = 0, l'algoritmo deve restituire il vettore delle distanze D = [0, +∞, 1, 2, +∞, 4, +∞]
- per s = 4, l'algoritmo deve restituire il vettore delle distanze D = [11, 6, -2, 7, 0, 13, +∞]
- per s = 2, l'algoritmo deve restituire il vettore delle distanze D = [+∞, +∞, 0, +∞, +∞, +∞, +∞]

L'algoritmo proposto deve avere tempo di calcolo Θ(n + m) dove n ed m sono i nodi e gli archi del grafo, rispettivamente.
Motivare BENE la correttezza e la complessità dell'algoritmo proposto.



Soluzione 2

Inizialmente avevo pensato di eseguire una DFS senza controllare i nodi gia visitati cosi che un nodo possa essere ricontroll ato piu volte nel caso in cui durante una visita in profondità troviamo un cammino che non sia ancora il migliore. Il problema è che una visita di questo tipo alzerebb e molto il costo nel caso di un grafo denso (sempre aciclico pero) poiché se molti nodi hanno molti nodi entranti allora le visite su quel nodo sarebbero numerose e il costo supererebbe perfino

O(n+m)

```python
def traccia_2(G, s):
    def DFS_dist(G, x, distanze):
        for y, costo in G[x]: # per ogni nodo y adiacente di x
            if distanze[x] + costo < distanze[y]:
                # se il cammino da x a y è minore del costo già salvato in y
                distanze[y] = distanze[x] + costo # aggiorniamo il cammino minore di y
            DFS_dist(G, y, distanze) # continuo la visita
    distanze = [float("inf")]*len(G) # inizializzo le distanze a infinito
    distanze[s] = 0 # la radice ha distanza 0
    DFS_dist(G, s, distanze) # eseguo la DFS per trovare le distanze dalla radice
    return distanze
```

La soluzione corretta è quindi quella di effettuare prima un ordinamento topologico sul grafo e poi per ogni ordinato aggiornare il costo dei cammini sui nodi adiacenti.
Creiamo prima la lista dei nodi ordinati con una DFS in post-order, inizializziamo le distanze dei nodi a infinito e per ogni nodo della lista ordinata, controlliamo se il cammino per arrivare al nodo adiacente è minore di quello già salvato nel nodo adiacente e in quel caso aggiornare col cammino minimo migliore.

```python
def traccia_2_1(G, s):
    def DFS_ord_pesato(G, x, visitati, sorted):
        visitati[x] = 1 # visito il nodo
        for y, costo in G[x]: # per ogni nodo adiacente
            if not visitati[y]: # se non è stato visitato
                DFS_ord_pesato(G, y, visitati, sorted) # continuo la visita
        sorted.append(x) # aggiungo in post-order il nodo corrente alla lista
    n = len(G)
    visitati = [0]*n # inizializzo i nodi come non visitati
    sorted = []
    for x in range(n): # per ogni nodo del grafo
        if not visitati[x]: # se non è stato visitato
            DFS_ord_pesato(G, x, visitati, sorted) # eseguo la DFS
    sorted.reverse() # inverto la lista per avere l'ordinamento topologico corretto
    distanze = [float("inf")]*n # inizializzo le distanze a infinito
    distanze[s] = 0 # la radice ha distanza 0
    for x in sorted: # per ogni nodo ordinato
        for y, costo in G[x]: # per ogni suo adiacente
            cammino = distanze[x] + costo
            if cammino < distanze[y]: # se il cammino dal padre ha costo minore di quello gia salvato
                distanze[y] = cammino # aggiorno la distanza in base al cammino dal padre
    return distanze
```

Il costo della DFS è O(n+m) e quello dei due for annidati è Θ(n+m). Quindi il costo finale è O(n+m)

Esercizio 3

Soluzione scritta in aula col prof.
Una volta trovata una porta eseguo una visita e scambio da 0 a 1 le celle visitate, eseguo per tutte le porte trovate
bisogna trasformare la matrice in un grafo e non lavorare sul labirinto della matrice stessa (tipo controllare le celle adiacenti per ogni punto della matrice)
Per trasformare una matrice in un grafo diamo ad ogni cella un numero identificativo.
Faremo diventare la cella in pos i, j in un nodo num. i*n + j (es. n = 5 | cella 0, 0 → 0*5+0 = 0, cella 2, 3 → 2*5+3 = 13)
eseguo due for per scorrere la matrice e ad ogni cella controllo se i 4 nodi adiacenti sono 0 (controllo se non esco dalla matrice) e nel caso lo connetto con un arco
Alla fine avrò una lista di adiacenza e mi basta eseguire una visita DFS per contare il numero di 0.
Per convertire un nodo x nelle posizioni della cella facciamo la divsione per avere la riga e il resto della divisione per la colonna (es, nodo 15 → 15/5 = 3 (i), 15%5 = 0 (j))
Dobbiamo trovare le porte del nuovo grafo, per ogni nodo, se la sua riga è 0 o n-1 oppure se la colonna è 0 o n-1 allora è una porta, e da ogni porta eseguiamo una visita DFS. il numero di nodi visitati partendo dalle porte è il numero da ritornare.

Esercizio 4

Devo eseguire n lavori, ognuno dei quali ha un tempo d'esecuzione specifico. Mi è fornito un vettore T di n componenti, dove T[i] rappresenta il tempo richiesto per eseguire il lavoro i. Inoltre ho una lista di liste P di n componenti, dove P[i] contiene i lavori che devono essere completati prima che io possa iniziare il lavoro i. Progettare un algoritmo che, dati T e P, in tempo O(n²), calcoli il tempo minimo necessario per completare tutti i lavori tenendo conto che è possibile eseguire più lavori in parallelo. L'algoritmo deve restituire +∞ nel caso in cui non sia possibile comletare tutti i lavori (ad esempio a causa di cicli di dipendenza tra i lavori).
Ad esempio: per n = 6, T = [3, 2, 5, 4, 3, 1] e

$$P = \begin{bmatrix} [1,2], \\ [3], \\ [3], \\ [4], \\ [5], \\ [] \end{bmatrix} \quad significa \quad \begin{matrix} "Lavoro\ 0\ dipende\ da\ 1\ e\ 2" \\ "Lavoro\ 1\ dipende\ da\ 3" \\ "Lavoro\ 2\ dipende\ da\ 3" \\ "Lavoro\ 3\ dipende\ da\ 4" \\ "Lavoro\ 4\ dipende\ da\ 5" \\ "Lavoro\ 5\ dipende\ da\ nessuno" \end{matrix}$$

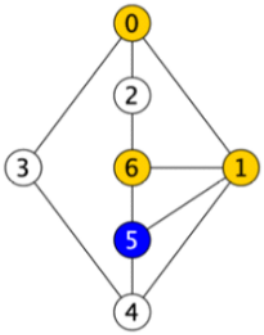
La risposta è 16 (il prof ha detto 14 ma è sbagliato)

Soluzione

Si può creare un grafo diretto e pesato dove gli archi x-y dipendono dalle dipendenze in P e il peso dell'arco dipende dal tempo in T del nodo di destinazione y.

Esercizio 5

Sia dato un grafo non orientato e connesso G, i cui nodi sono colorati. Un cammino lecito nel grafo è un cammino che non attraversa nodi adiacenti dello stesso colore.
Dati due nodi a e b, l'obiettivo è trovare il num. min. di archi da percorrere per andare da a a b seguendo un cammino lecito. Ad esempio il grafo in figura:



- Il cammino minimo lecito da 0 a 1 ha lunghezza 4 (e tocca i nodi 0-2-6-5-1) (poiché 0, 1 e 6 hanno lo stesso colore e 3 e 4 anche)
- Il cammino minimo lecito da 3 a 4 ha lunghezza 5 (e tocca i nodi 3-0-2-6-5-4) (poiché 3 e 4 hanno lo stesso colore)

Il grafo G è rappresentato tramite liste di adiacenza e i colori dei nodi sono memorizzati in un vettore C, dove C[i] rappres enta il colore del nodo i
Progettare un algoritmo che, dati G, C, a e b risolva il problema restituendo il min. num. di archi da percorrere da a fino a b lungo un cammino lecito.
Se non esiste alcun cammino lecito tra a e b, l'algoritmo deve restituire None.
L'algoritmo deve avere una complessità O(m) dove m è il num. di archi nel grafo G.

1. Spiegare a parole il funzionamento dell'algoritmo
2. Scrivere lo pseudocodice dell'algoritmo
3. Giustificare la complessità O(m) dell'algoritmo

Soluzione 5

L'algoritmo è una semplice visita BFS utilizzando una coda e partendo da a come nodo sorgente, dove per ogni nodo visitato co ntrolliamo se i suoi adiacenti non sono stati gia visitati (quindi se non hanno ancora una distanza precisa dal nodo sorgente) e se il passaggio al nodo è lecit o (quindi ha colore diverso dal padre). In quel caso possiamo continuare il cammino sul nodo e incrementare la distanza.
Nel caso in cui il nodo stesso sia il nodo b che dobbiamo raggiungere e siccome so che la BFS trova la distanza minima, allor a posso direttamente ritornare la distanza di quel nodo.
Se invece abbiamo finito la visita BFS e non abbiamo trovato un cammino lecito allora ritorniamo None

```python
def traccia_5(G, C, a, b): # grafi colorati
    D = [-1]*len(G)
    D[a] = 0
    coda = [a]
    i = 0
    while i < len(coda):
        x = coda[i]
        i += 1
        for y in G[x]:
            if D[y] == -1 and C[x] != C[y]:
                # se non è stato visitato e ha colore diverso dal padre
                D[y] = D[x]+1 # possiamo visitarlo e continuare il cammino
                if y == b: # se abbiamo trovato il nodo di arrivo
                    return D[y] # ritorniamo la sua distanza
                coda.append(y) # altrimenti continuiamo con la visita
    return None # se non abbiamo trovato nulla ritorniamo None
```

Con una semplice visita BFS si può pensare che il costo sia O(n + m) pero siccome in un grafo connesso **m ≥ n-1** allora possiamo dire che il costo della visita sia O(m)

Esonero 2024

martedì 25 marzo 2025 09:12

Esercizio 1

Dato un grafo diretto G con n nodi e m archi e due suoi nodi a e b, chiamiamo equidistante un qualunque nodo x di G che ha uguale distanza da a e da b, in caso contrario diciamo che x è nella sfera di influenza di quello dei due nodi che è a lui più vicino. Chiamiamo infine vettore delle influenze rispetto ai nodi a e b il vettore D di n componenti dove D[i] indica il nodo alla cui sfera d'influenza i appartiene e nel caso x sia equidistante da a e b allora D[i] = −1. In figura ad esempio è riportato a sinistra un grafo con evidenziati in bianco i due nodi a = 2 e b = 6 e sulla destra i nodi colorati differentemente ad indicare le influenze e le equidistanze. In questo caso il vettore delle influenze è D = [2, −1, 2, −1, 6, −1, 6, −1, 2, 6, 6]. Progettare un algoritmo che, dato G tramite liste di adiacenza e i nodi a e b, in tempo O(n + m) restituisce il vettore delle influenze.

Prendiamo i vettori delle distanze da a e da b con due visite BFS in serie per a e b. Inizializziamo il vettore delle distanze a -1: eseguiamo un for unico per entrambi i vettori delle distanze e controlliamo insieme gli elem. delle due liste delle distanze:

- Se la distanza in indice i della lista A e minore di quello della lista B, allora il nodo i sarà nella sfera di influenza di A → D[i] = a
- Se invece la distanza i in A è maggiore di quella in B, il nodo i sarà nella sfera di influenza di B → D[i] = b
- Altrimenti se il nodo è equidistante da a e b, il suo valore nel vettore resterà -1.

Pseudo-codice:

```
A <- BFS_dist(G, a)
B <- BFS_dist(G, b)
D <- [-1, …, -1]
for i in range(len(G)):
    if A[i] < B[i]:
        D[i] = a
    elif A[i] > B[i]:
        D[i] = b
return D
```

Esercizio 2

1. Sia T un minimo albero di copertura di un grafo pesato G. Sia G' il grafo che si ottiene da G incrementando di una stessa costante positiva c il peso di ciascun arco. L'albero T è un albero di copertura minimo anche per G'?
2. Sia G un grafo fortemente connesso e si consideri la visita in profondità di G a partire dal nodo 0. Tra archi all'indietro, archi in avanti e archi di attraversamento quali verranno di certo incontrati e quali possono invece anche non comparire?

3. Si consideri l'albero T ottenuto a seguito di una visita in ampiezza a partire dal nodo 0 di un grafo non diretto e connesso G. Sia a, b un arco di G non presente nell'albero T. Sia h(x) l'altezza del generico nodo x in T. Può aversi h[a] − h[b] > 2? La risposta cambia se il grafo G è diretto e l'arco va da a a b?

4. Quanti sort topologici sono possibili per un grafo diretto con n nodi ed un solo arco?

5. Considera un grafo connesso G in cui tutti i nodi hanno grado al più 3 qual'è il numero massimo di nodi che possono trovarsi a distanza esattamente d ≥ 1 dal nodo 0 di G? E quanti a distanza al più d?

SOL

1. Supponiamo per assurdo che T non sia più albero minimo. Allora esiste un altro albero minimo T' tale che la somma degli archi di T' è minore della somma degli archi di T. Incrementando di k il peso di ogni arco. avremmo comunque che

$$\sum_{e\in T'} t(e) < \sum_{e\in T} t(e)$$
$$\sum_{e\in T'} k + t(e) < \sum_{e\in T} k + t(e)$$
$$k * (n-1) \sum_{e\in T'} t(e) < k * (n-1) \sum_{e\in T} t(e)\, ung$$
$$\sum_{e\in T'} t(e) < \sum_{e\in T} t(e)$$

2. in un grafo fortemente connesso c'è sempre un ciclo, e se c'è un ciclo c'è un arco all'indietro.
   Essendo un grafo fortemente connesso la DFS effettua un solo cammino, e di conseguenza non ci possono essere archi di

attraversamento.
Quelli in avanti potrebbero essere incontrati

3. E impossibile poiché essendoci un arco tra a - b la BFS creerà un albero in cui b e a sono due nodi figli/padre e quindi la loro distanza è sempre 1.
   Dimostr per assurdo di Tutor: Supponiamo sia vero, Sia x l'antenato di a,b più lontano dalla radice. Sia P il cammino x → b e sia Q quello x → a, Siccome h(b) < h(a) - 2 devono essitere due nodi c e d sul cammino Q, situati tra x e a.
   Quindi Q = (x, c) (c, d), (d, a) Ma esiste (b, a) quindi P ∪ (b, a) è più corto di Q, quindi la BFS ha sbagliato a trovare un cammino minimo, il che è assurdo.
   3.1 Se G è orientato, dipende dalla direzione di {a, b}. Se (a, b) si
   Se (b, a) no perche discorso analogo al caso indiretto

4. Se non ci fosse un vincolo di un arco, sarebbero n! sort diversi. Se c'è un arco, abbiamo due casi in cui un arco tra due nod i (u, v) avremmo metà dei sort in cui u è prima di v, e metà in cui v è prima di u. A noi interessano solo la metà dei sort in cui u è prima di v. ⇒ n!/2

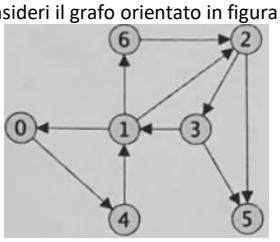
5. n/d - 1 per difetto. È un problema di combinatoria. Non esiste un grafo con num. dispari di nodi e vertici di grado uguale
   5.1

Esonero 2025 saltato

mercoledì 2 aprile 2025 23:16

Esercizio 1

Si consideri il grafo orientato in figura



Viene effettuata una DFS di G a partire dal nodo 4. Assumendo che G sia memorizzato tramite liste di adiacenza e che nelle liste di adiacenza i nodi compaiano in ordine crescente:

- Disegnare l'albero DFS che si ottiene dalla visita
- Dire se nel corso della visita si incontrano archi di attraversamento (e quali), archi all'indietro (e quali) e archi in avanti (e quali)

Esercizio 2

Si consideri una DAG G

1. Quale è il num. max di archi che può avere G se ha n nodi?
2. Quante componenti fortemente connesse ha G?
3. Si supponga che G abbia 5 nodi ed un singolo arco. Quanti ordinamenti topologici distinti ammette G in questo caso?
4. Si supponga che G abbia n nodi, un pozzo universale x e un tot. di s ordinamenti topologici distinti. Quanti ordinamenti topologici distinti avrà il DAG G1, ottenuto da G eliminando tutti gli archi incidenti su x?

Esercizio 3

Hai due parole s e t di uguale lunghezza l, e il tuo obiettivo è determinare il num. min. di trasformazioni necessarie per passare dalla parola di partenza s a quella di destinazione t, cambiando una lettera alla volta. Ogni parola intermedia deve essere una parola valida presente in una lista A contenente n parole di lunghezza l e comprendente s e t.
Progettare un algoritmo che, date le due parole s e t e la lista A, in tempo $O(l*n^2)$, restituisca la sequenza di trasformazioni effettuate per trasformare s in t o una lista vuota se la trasformazione non è possibile.
Ad esempio per:

- s = "cane"
- t = "data"
- A = ["dote", "rata", "cave", "data", "cate", "rapa", "cane", "core", "rate", "cose"]

L'algoritmo deve restituire ["cane", "cate", "rate", "rata", "data"]

1. Spiegare a parole il funzionamento dell'algoritmo
2. Scrivere lo pseudocodice dell'algoritmo
3. Giustificare la complessità $O(l*n^2)$ dell'algoritmo

Creiamo prima un grafo G non diretto dove ogni nodo è una tupla (i, x), dove i è un intero che identifica il nodo e x è una parola tra quelle di A.
Nel grafo esiste un arco tra il nodo x e y solo se hanno 3 lettere in comune. Quindi il passaggio da un nodo all'altro è effettuato cambiando di una lettera.

- Prima di controllare i collegamenti validi tra le parole di A, creo una lista di adiacenza G vuota di n elementi per indicare il grafo.
- Creiamo poi una nuova lista A_tupla dove per ogni parola x di A abbiamo una tupla (i, x), dove i è l'indice di x in A e x è la parola.
  Mentre creo la lista, salvo gli indici di s e t in A, per poterli usare dopo
  Quindi in base all'esempio di prima avremmo A_tupla = [(0, "dote"), (1, "rata"), (2, "cave"), …, (n-1, "cose")]
- Per ogni parola x di A e per ogni parola y di A diversa da x, inizializziamo a 0 il contatore di lettere differenti e controlliamo insieme le lettere di x e y
- Per ogni lettera in pos. k < l diversa tra le due parole incrementiamo il contatore di lettere differenti.
- Se il num. di lettere differenti è 1 lettera allora tra c'è un passaggio valido tra le due parole e possiamo collegare x e y con un arco
  Quindi aggiungiamo in G[i] la tupla (j, y) dove i è la posizione di x in A e j è la posizione di y in A
- Altrimenti se il num. di lettere differenti tra x e y è maggiore di 1, non possiamo creare un collegamento valido tra di loro
- Infine avremmo una lista di adiacenza G dove

Dopo aver creato questo grafo valido, posso utilizzare una BFS per cercareil cammino minimo partendo da s per arrivare a t.
Per cercare il cammino, con la BFS devo creare il vettore dei padri dell'albero radicato in s.
Nella BFS lavoreremo in base agli indici dei nodi salvati in G. Poiché la ricerca in base alle parole stesse richiederebbe un dizionario in cui le chiavi sono le parole e i valori sono le parole adiacenti valide, pero la ricerca in un dizionario nel caso medio è O(1) ma nel caso peggiore è O(n) dove n è la lunghezza del dizionario (e quindi di A)

- Inizializzo nell vettore dei padri ogni nodo a -1 (non visitato). Invece per il nodo s inizializzo il suo padre come s stesso
- Inizio la visita BFS partendo dal nodo s.
- Per ogni nodo x e ogni suo adiacente (j, y), se il suo adiacente non è stato visitato ancora (P[j] == -1) allora posso assegnargli il padre (P[j] = x) e continuare la visita sul nodo j.
- Infine avro il vettore dei padri radicato in s

Tramite il vettore dei padri posso controllare se esiste un cammino da s a t. Mi basterà ritornare di padre in padre, partendo dal nodo t e risalendo fino a s

- Creo una lista vuota Cammino che usero per inserire i nodi attraversati nel cammino da t a s
- Prima controllo se esiste un cammino da s a t, quindi vedo se P[i] != -1, altrimenti non esiste un cammino da s a t e posso tornare la lista vuota
- Nel caso in cui esiste un cammino, continuo con la creazione del cammino
- Utilizzo una var. x per indicare il nodo di scorrimento, il quale inizializzo alla pos di t (i)
- Eseguo un ciclo while finche non arrivo alla radice (P[x] == x)
- Nel ciclo, aggiorno x al padre di x (x = P[x]) e aggiungo la parola in posizione x in A_tuple al cammino
- Dopo il ciclo, aggiungo la parola radice al cammino (non è stata aggiunta perche è finito il ciclo quando siamo arrivati alla radice)
- E restituisco il cammino

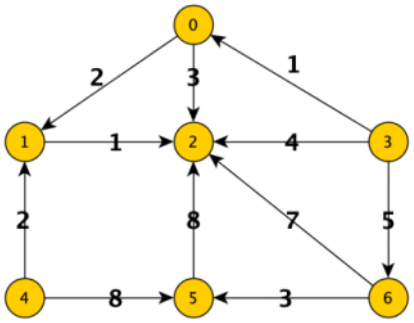
Infine il cammino finale richiesto dall'algoritmo è il cammino creato tramite il vettore dei padri

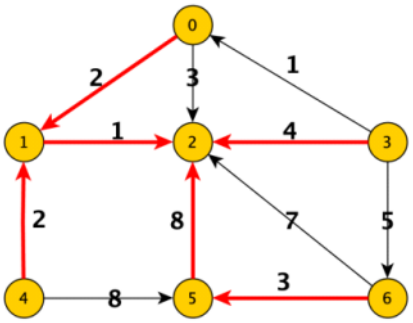
Slide 12 (Approssimazione)

lunedì 12 maggio 2025 12:55

Esercizio 1

In un grafo diretto e pesato G un sottoinsieme A degli archi è detto **univoco** se non contiene 2 o più archi uscenti dallo stesso nodo (solo un arco uscente per nodo)
Il peso dell'insieme univoco è dato dalla somma dei pesi dei suoi archi





In rosso un insieme univoco di G di peso 20

Progettare un algoritmo greedy che, dato un grafo G diretto e pesato, in tempo O(n + m) restituisce il peso e gli archi di un insieme univoco di peso massimo di G
Motivare correttezza e complessità dell'algoritmo

Soluzione:
Per ogni nodo controllo quale tra i suoi archi uscenti è quello col peso maggiore e lo aggiungo ad A.
Siccome è un grafo diretto, prenderò solo un arco uscente per ogni nodo e quindi avremmo un sottoinsieme A **univoco**, in cui ogni arco è quello col peso massimo per ogni nodo.

Poiché per ogni nodo itero i suoi archi ai nodi adiacenti una sola volta, quindi il costo sarà O(n + m)

```
def es_1_grafi_12(G):
    A = []
    for x in range(len(G)): # per ogni nodo x
        max_y = -1 # adiacente inizializzato a -1
        max_p = float("-inf") # peso inizializzato a -inf
        for y, p in G[x]: # per ogni adiacente y e peso p
            if p > max_p: # se abbiamo trovato un peso maggiore
                max_p = p # salviamo il peso e il nodo adiacente
                max_y = y
        if max_y != -1:
            A.append((max_y, max_p)) # aggiungiamo l'arco massimo
    return A
```

Esercizio 2

All'acquisto di n biglietti per un'importante prima teatrale sono interessati m gruppi di persone. Ciascun gruppo ha intenzione di acquistare i biglietti solo se questi sono a sufficienza per tutti i membri del gruppo.
Vogliamo selezionare i gruppi a cui vendere i biglietti in modo da massimizzare il num. di biglietti venduti

Per risolvere il problema ci viene proposto il seguente algoritmo greedy che:

- Prende in input il num. n di posti disponibili e la lista di m elem. dove in lista[i] c'è un num $\leq$ n che rappresenta i biglietti richiesti dal gruppo i
- Restituisce il num max di biglietti che è possibile vendere e la lista dei gruppi a cui venderli

```
def selezione(lista, n):
    gruppi=[(lista[i],i) for i in range(len(lista))]
    gruppi.sort(reverse=True)
    tot, Sol=0, []
    for l,i in gruppi:
        if tot+l<=n:
            Sol.append(i)
            tot+=l
    return tot, Sol
```

1. Provare che l'algoritmo non è corretto
2. Provare che l'algoritmo ha un rapporto d'approssimazione limitato a 2

Soluzione:
L'algoritmo crea inizialmente una lista gruppi dove ogni elemento è una coppia (x, i) dove x è il num. di biglietti per il gruppo i
Esegue il sort al contrario della lista gruppi cosi che abbiamo una lista in ordine decrescente in base al num. di biglietti per gruppo
Inizializza una variabile **tot** per indicare il num. di biglietti venduti
Per ogni gruppo, se la somma del tot e del num. di biglietti per il gruppo è minore del totale, allora possiamo vendere i biglietti al gruppo i e aggiungiamo il num. di biglietti venduti al totale.

1)
Vengono scelti quindi prima i gruppi col maggior num. di biglietti però cosi vengono esauriti prima i posti, lasciando fuori gruppi più piccoli che nel totale potevano

vendere più biglietti. (Es: n = 10, [9, 6, 5, 5] → prende prima 9 e basta (poiché resta 1 posto solo). Scelta ottima: i gruppi da 5 avrebbe venduto 10 biglietti)
L'algoritmo non era corretto anche se il sort era in ordine crescente (Es: n=10, [1, 5, 6, 9] → prende prima 1 e 5 e poi non prende altro. Scelta ottima: 1 e 9)
In realtà l'algoritmo ottimo dovrebbe controllare tutte le permutazioni dei gruppi per trovare la combinazione che ha il num. maggiore di biglietti venduti

2) Questo è un problema di massimizzazione quindi useremo il confronto $\frac{OTT(I)}{A(I)}$
   Supponiamo che l'algoritmo greedy prenda solo un gruppo g1 escludendo gli altri gruppi più piccoli perche superano n, pero la soluzione ottima prende invece la combinazione di gruppi piccoli per raggiungere il num. massimo di biglietti venduti.
   Nella soluzione peggiore abbiamo un solo gruppo g ≥ n/2 e tutti gli altri gruppi rimasti sono troppo grandi per arrivare a n
   Però la soluzione ottima non può superare n, quindi:
   - A(I) $\geq \frac{n}{2}$
   - OTT(I) < n
   - $A(I) \geq \frac{OTT(I)}{2} \Rightarrow \frac{A(I)}{OTT(I)} \geq \frac{1}{2} \Rightarrow \frac{OTT(I)}{A(i)} \geq 2$

Esercizio 3

Dato un grafo connesso e pesato G, ed un intero k ≤ n vogliamo trovare un sottografo aciclico di G che comprende il nodo 0, contiene k nodi e sia di costo minimo (nota che quando n = k il problema diviene quello di trovare il minimo albero di copertura di G)

Viene proposto questo algoritmo greedy che preso G e l'intero k, restituisce i nodi del sottografo aciclico ed il suo costo.

```
def copertura(G,k):
    A,costo= [0],0
    E=[(c,i,y) for i in range(len(G)) for y in G[i] ]
    for _ in range(k):
        E1=[(c,x,y) for c,x,y in E if x in A and y not in A]
        c,x,y= min(E1)
        costo+=c
        A.append(y)
    return A, costo
```

1) Provare che l'algoritmo sbaglia
2) Provare che l'algoritmo non ha un rapporto d'approssimazione costante

Soluzioni:
Viene inizializzata una lista E contenente le coppie (c, x, y) dove c è il costo dell'arco da x a y del grafo.
Poi per k volte:
- Crea una nuova lista E1 con le coppie (c, x, y) per ogni arco (x, y) con x già aggiunto al sottografo e y no.
- Prende l'arco (c, x, y) col costo minore
- Aggiunge il nodo y al sottografo e incrementa il costo totale del sottografo

Siccome il ciclo è eseguito k volte, verranno inseriti solo k nodi

1) Supponiamo di avere un albero radicato in 0, e che 0 abbia due sottoalberi a e b con il costo rispettivamente x+1 e x:
   - Ogni nodo del sottoalbero di a ha un arco con costo 0. Quindi il costo totale del sottoalbero di a è x+1
   - Ogni nodo del sottoalbero di b ha un arco con costo x. Quindi il costo totale del sottoalbero di b è x*(num di nodi nel sottoalbero)

   Perciò se a e b hanno entrambi almeno un figlio (quindi entrambi i sottoalberi avranno almeno un nodo diverso da a o b), il sottoalbero con costo minore sarà sempre quello che parte da a.
   Però l'algoritmo partendo da 0, tra gli archi validi sono (0, a) con peso x e (0, b) con peso x-1, sceglie l'arco (0, b) poiché ha costo minore.
   Nei passi successivi sceglierà sempre gli archi del sottoalbero di b rispetto all'arco ancora valido (0, a).
   Vediamo quindi che siccome l'algoritmo non può vedere oltre agli archi dei nodi del sottografo, non potra mai prendere il cammino nel sottoalbero col costo minore.
2) Per il fatto descritto nel punto 1, all'aumentare dei nodi nel sottoalbero di b, il rapporto $\frac{A(I)}{OTT(I)}$ tende all'infinito, quindi non ha valore costante

Esercizio 4

Sia S una sequenza binaria, una sottosequenza di S si ottiene eliminando da S un num. arbitrario di caratteri
Es: S = 1011001
- 01101 e 10001 sono sottosequenze di S (ottenute da _011_01 e 10__001)
- 00110 e 01010 non sono sottosequenze di S

Problema: date due sequenze binarie A e B vogliamo trovare una sottosequenza di lungh. max comune ad entrambe
Es: per A = 01101 e B = 110100 una sottosequenza comune di lungh. max è 1101 (sol. ottima)
Viene proposto il seguente algoritmo:

```
def comune( A, B):
    z = min( A.count(0), B.count(0))
    u = min( A.count(1), B.count(1))
    if z >= u:
        return [0] * z
    return [1] * u
```

1) trovare un controesempio che fa sbagliare l'algoritmo con un rapporto d'approssimazione di 2
2) dimostrare che il rapporto d'approssimazione dell'algoritmo è 2.

Soluzione:
L'algoritmo avrà in z il num minore di 0 tra A e B e in u il num. minore di 1 tra A e B. Poi se z è ≥ u allora ritorna 0 per z volte altrimenti 1 per u volte
(es: A = 01101 e B = 110100 → z = 2; u = 3. u>z → ritorna 111)

1) Se abbiamo A = 001100 e B = 0110, allora:

- L'algoritmo greedy conterà z = 2 e u = 2. Siccome z ≥ u allora ritornerà 00 → A(I) = 2
- L'algoritmo ottimo ritornerebbe invece 0110 poiché è presente sia in A che in B ed ha lung. max. → OTT(I) = 4

Quindi avremmo:

$$\frac{OTT(I)}{A(I)} = \frac{4}{2} = 2$$

2) A(I) è uguale a OTT(I) solo quando le sequenze hanno un valore solo e sono uguali.
Altrimenti negli altri casi in cui A(I) è valida, siccome prende sempre il val. con meno occorrenze, non potrà mai superare la metà degli elementi della sequenza più corta.
Ad es, se abbiamo A composto da $\left\lfloor\frac{n}{2}\right\rfloor$ 0 e $\left\lceil\frac{n}{2}\right\rceil$ 1 e al contrario B è composto da $\left\lfloor\frac{n}{2}\right\rfloor$ 1 e $\left\lceil\frac{n}{2}\right\rceil$ 0 allora avrà z e u = $\left\lceil\frac{n}{2}\right\rceil$ e stamperà $\left\lfloor\frac{n}{2}\right\rfloor$ 0 (stampa prima 0 di 1 se hanno lo stesso val)
Mentre OTT(I) potrà prendere almeno n elementi (nel caso in cui A e B sono uguali o A è interamente in B o viceversa)
Quindi avremmo:
   - $OTT(I) \leq n$
   - $A(I) \leq \frac{n}{2}$
   - $\frac{OTT(I)}{A(I)} \leq \frac{n}{\frac{n}{2}} = 2$

Esercizio 5

Dato un grafo G = (V, E) vogliamo assegnare ad ogni nodo un colore nell'insieme {1, 2, …, n} in modo tale che nodi adiacenti ricevano colori distinti e che il num di colori distinti usati sia minimo.
Si propone il seguente algoritmo greedy che restituisce il vettore Sol di n componenti dove Sol[i] riporta il colore assegnato al nodo i

```
def es(G):
    for i in range(n):
        C=[1]*n
        for j in range (1,i-1):
            if j in G[i]: C[sol[j]]=0
        c=1
        while C[c]==0: c+=1
        sol[i]=c
    return sol
```

Provare o confutare che l'algoritmo risolve il problema

Soluzione:
L'algoritmo crea un for che itera per n volte che indica l'insieme di colori. Per ogni colore crea inizialmente una tabella grande n, inizializzando ogni elem. a 1
Itera poi per i colori da 1 a i-1 (quindi quelli precedenti di i) e se abbiamo inserito il colore

Esercizi Simili fatti da ChatGPT

Esercizio 6 - Bilanciamento di Carico su Due Macchine

Hai un insieme di n lavori indipendenti, ciascuno con tempo di esecuzione pi > 0. Vuoi assegnarli a due macchine identiche per minimizzare il tempo di completamento massimo (makespan).
Algoritmo Greedy
Processa i lavori in un ordine qualunque, e per ciascun lavoro assegnalo alla macchina che, al momento, ha carico minore.

```
def bilanciamento(p):
    # p: lista di tempi [p0, p1, …, p_{n-1}]
    carico = [0, 0]
    assegn = [[], []]
    for i, pi in enumerate(p):
        # scegli la macchina con carico minore
        j = 0 if carico[0] <= carico[1] else 1
        assegn[j].append(i)
        carico[j] += pi
    return carico, assegn
```

1. Costruisci un controesempio (inizializzazione di p) per cui l'algoritmo **non produce** l'assegnamento ottimo.
2. Dimostra che, qualunque sia l'istanza,

$$makespan_{greedy} \leq 2 * makespan_{opt}$$

Soluzione

1. Se diamo una lista crescente di n lavori dove il tempo di esecuzione per pi = i, allora l'algoritmo inserira i lavori alternandoli uno ad uno tra le due macchine, separando i lavori pari e dispari nei due macchinari. Pero in questo modo il tempo di completamento massimo è dato dal massimo tra le somme dei valori dispari e quelli pari.
Il numero dei primi n numeri pari è da e la somma dei primi n numeri dispari è data da n^2, quindi il tempo di completamento massimo per l'algoritmo greedy sarà max(n*(n+1), n^2)
Un algoritmo ottimo in questo caso dovrebbe ritornare la metà della somma dei lavori per bilanciare i lavori tra le macchine, cioe $\left\lceil\frac{\frac{n*(n+1)}{2}}{2}\right\rceil = \left\lceil\frac{n*(n+1)}{4}\right\rceil$.
Vediamo che:

$$\left\lceil\frac{n*(n+1)}{4}\right\rceil < \frac{n*(n+1)+4}{4}$$

Quindi se n = 8, abbiamo i lavori con tempo [1, 2, 3, …, 8]:
   - Nell'algoritmo greedy avremmo la lista pari [2, 4, 6, 8] con somma 20 e i dispari [1, 3, 5, 7] con somma 16. Quindi il tempo di completamento massimo è

20

- Nell'algoritmo ottimo avremmo $\left\lceil \frac{8*9}{4} \right\rceil = \left\lceil \frac{72}{4} \right\rceil = 18$, che è meglio dell'algoritmo greedy

Esercizio 7 - Matching Massimale in Grafo Bipartito

Dato un grafo bipartito G=(U ∪ V,E), vogliamo trovare un matching (coppie (u,v)) di **max** cardinalità.

Algoritmo Greedy

Scorri tutti gli archi in un ordine qualunque; se l'arco (u,v) unisce due vertici ancora non "coperti" dal matching, aggiungi lo.

```python
def matching_greedy(edges):
    # edges: lista di tuple (u,v)
    matched = set()
    M = []
    for u, v in edges:
        if u not in matched and v not in matched:
            M.append((u, v))
            matched.add(u)
            matched.add(v)
    return M
```

1. **Controesempio**
   Mostra un grafo e un ordinamento degli archi per cui questo matching non è massimo.
2. **Rapporto d'approssimazione**
   Dimostra che il greedy restituisce sempre un matching di **almeno metà** della dimensione ottima:
   $$|M_{greedy}| \geq \frac{1}{2}|M_{opt}|$$

Soluzione

Esercizio 8 - Set Cover

Hai un universo finito U di n elementi e una famiglia di sottoinsiemi S={S1, S2, …, Sm} tali che $\bigcup_i S_i = U$. Vuoi scegliere il minor numero di insiemi che coprano U.

Algoritmo Greedy

Ad ogni passo, prendi l'insieme che **copre più** elementi ancora scoperti.

```python
def set_cover_greedy(U, sets):
# U: set di elementi
# sets: lista di set S_i
covered = set()
C = []
while covered != U:
    # scegli S che massimizza |S \ covered|
    S = max(sets, key=lambda S: len(S - covered))
    C.append(S)
    covered |= S
return C
```

1. **Controesempio**
   Costruisci un'istanza in cui il greedy non è ottimo (es. quando coprire singoli elementi separati è peggiore di scelte più "bilanciate").
2. **Rapporto d'approssimazione**
   Dimostra la famosa disuguaglianza
   $$|C_{greedy}| \leq H_n * |C_{opt}|, dove\ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{n}{1}$$

Soluzione

Slide 13 (Greedy)

mercoledì 14 maggio 2025 17:44

Esercizio 1 - Cassonetti per Strada

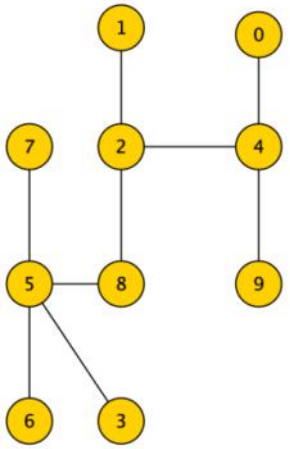
Abbiamo una lista con le pos $p_0 < p_1 < p_2 < \dots < p_{n-1}$ in cui si trovano n case lungo una strada rettilinea. Lungo la strada bisogna posizionare dei cassonetti e si vuole che ciascuna casa abbia almeno un cassonetto ad una distanza che non superi k.
Es: lista = [2, 5, 7, 11, 14, 16, 18] e k = 3, la sol ottima richiede 3 cassonetti, una possibile dislocazione è [4, 11, 15]
Progettare un algoritmo greedt che, data la lista e k, restituisca il minimo num. di cassonetti necessari a soddisfare il requisito e una lista con le pos in cui localizzarli.
La complessità deve essere O(n)

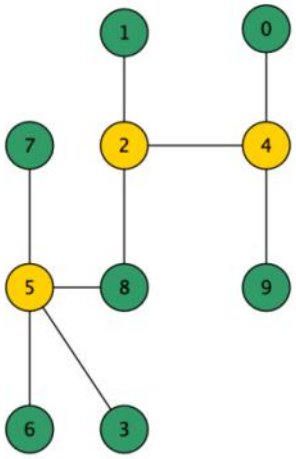
Soluzione

- Una possibile soluzione è contare quante case possiamo raggiungere se mettiamo il cassonetto alla casa i, con $0 \leq i < n$.
  Dobbiamo quindi contare quante case possiamo raggiungere ,

Esercizio 2 - Grafo Indipendente

Dato un grafo non diretto G un sottoinsieme dei suoi nodi è detto **indipendente** se non contiene vertici adiacenti.
Progettare un algoritmo greedy che, dato un grafo connesso e aciclico T, restituisca in tempo O(n), un insieme dominante per T di dim. max
Motivare correttezza e complessità dell'algoritmo
Es: per il grafo aciclico connesso rappresentato a sx, a dx in verde sono evidenziati i nodi del suo insieme indipendente di dim. max (di 7 nodi)





Soluzione

Idea: rimuoviamo i nodi di grado maggiore finche non rimaniamo con tutti i nodi di grado 0

- Per ogni nodo teniamo una lista G con G[i] = grado del nodo i.
- Eseguiamo una visita DFS da 0 e per ogni nodo che incontriamo, gestisco questi 3 casi:
  - Se il grado del nodo i che stiamo visitando è 1, e il suo unico nodo adiacente non è stato ancora visitato, allora rimuovendo quel nodo adiacente rimarremmo con i che fa parte dell'insieme indipendente, percio aggiungiamo i alla soluzione e visitiamo l'unico nodo adiacen te
  - Se il grado del nodo i che stiamo visitando è 1, e il suo unico nodo adiacente è gia stato visitato, allora significa che que sto è il nodo che verra rimosso, lasciando il nodo adiacente a far parte della soluzione. Quindi passiamo la visita sui nodi adiacenti.
  - Se il grado del nodo i che stiamo visitando è > 1, allora è un nodo che fa da vertici a più nodi, quindi dobbiamo eliminarlo

Esercizio 3 - Somma dopo k Operazioni

Hai due vettori di interi A e B entrambi di lunghezza n. Con un'operazione puoi scegliere una pos i, $0 \leq i < n$, e scambiare il contenuto di A[i] con quello di B[i]
Progettare un algoritmo greedy che dati i vettori A, B ed un intero $k \leq n$ in tempo O(n*log(n)) determina il max val che si può ottenere dalla somma degli elem. di A dopo al più k operazioni

Soluzione Sbagliata (modifica le liste)

- Ordino il vettore A in ordine crescente e il vettore B in ordine decrescente.
  Ora avrò che i primi k val. di A saranno i più piccoli di A e i primi k val. di B saranno i più grandi di B.
- Ora devo controllare tra i primi k val delle due liste, se conviene o meno scambiare i valori tra A e B. Per $0 \leq i < k$:
  - Se l'elem. A[i] < B[i], allora conviene scambiare i due valori
  - Se l'elem. A[i] > B[i], allora siamo arrivati ad un punto in cui non conviene fare lo scambio, quindi fermiamo il controllo
- Alla fine avrò effettuato al massimo k operazioni, e gli elem. più grandi di B che che ci facevano comodo in A (perche in quella posizione gli elem. di A erano più piccoli di quelli di B) sono stati scambiati in A.
- Ora possiamo ritornare la somma degli elem. di A

Soluzione

- Per ogni elem. in pos. i di A e B, salvo in una lista D la differenza tra i due elem e la posizione i. Quindi D[i] avrà (B[i] - A[i], i)
- Ordino la lista delle differenze in ordine decrescente e per i primi k val della lista, vedo se conviene o meno scambiare i valori tra A e B. Per $0 \leq i < k$:
  - Se D[i][0] > 0, conviene effettuare lo scambio e scambio gli elem. tra A e B in pos D[i][1]
  - Altrimenti, non conviene effettuare lo scambio e esco dal controllo
- Ora avrò scambiato al massimo O(k) elem. tra A e B, e posso ritornare la somma del vettore A

```python
def es_3_greedy_13(A, B, k):
    n = len(A)
    D = [(0, 0)]*n # inizializzo la lista delle differenze
    for i in range(n):
        D[i] = (B[i] - A[i], i) # salvo la differenza e la posizione
    D.sort(reverse=True) # ordino la lista in ordine decrescente
    for i in range(k): # per le k operazioni
        if D[i][0] > 0: # se la differenza è maggiore di 0
            pos = D[i][1]
            A[pos], B[pos] = B[pos], A[pos] # conviene scambiare gli elem.
        else: break # altrimenti non conviene scambiare più
    return sum(A) # ritorniamo la somma degli elem. di A
```

Costo:

- Il costo della creazione della lista è Θ(n)
- Il primo for itera n volte → Θ(n)
- Il sort ha costo O(n*log(n))
- Il secondo for itera al massimo per k ≤ n volte → O(k)
- La somma del vettore di A ha costo Θ(n)

Costo totale: Θ(n) + O(k) + O(n*log(n)) = O(n*log(n)) (siccome k ≤ n)

Esercizio 4 - Benzinai

Devi guidare dal punto a al punto b di un autostrada e vuoi minimizzare il numero di fermate alle n pompe di benzina che incontrerai sapendo che un pieno ti permette di percorrere L chilometri.

Progetta un algoritmo greedy che in tempo Θ(n) restituisce il numero minimo di fermate che devi effettuare. L'algoritmo prende in input l'intero L e la lista di n interi x1, x2,...xn contenente le distanze da a delle n pompe di benzina che incontrerai una dopo l'altra sull'autostrada lungo l'autostrada.

Soluzione

- Inizializziamo un contatore benz a L che usiamo per tenere traccia dei chilometri che abbiamo percorso per arrivare dall'ultima pompa al chilometro generico x
- Creiamo una lista D contenente le distanze tra le varie pompe di benzina. Ad esempio D[i] conterrà la distanza pompa[i] - pompa[i-1]. Se ci troviamo nell'ultima posizione allora mettiamo 0.
- Eseguiamo un for da 0 a n per indicare le pompe di benzina , quindi con 0 ≤ i < n:
    - Se arrivo all'ultima pompa mi fermo, e non serve fare di nuovo il pieno
    - Quando arrivo alla pompa i, decremento il contatore di benz per la distanza effettuata per arrivare alla pompa.
    - Se il contatore è 0, dovrò fare benzina alla pompa i.
    - Se la distanza fino alla pompa successiva i+1 è maggiore di benz allora significa che non potrò arrivare a quella pompa senza fare un pieno alla pompa i. Quindi aggiungo L al contatore di benz, incremento il num. di fermate effettuate e passo avanti
    - Se invece la distanza fino alla pompa successiva i+1 è minore, allora non serve che faccio nulla
- Alla fine ritorno il contatore che indica le fermate effetutate

Implementazione:

```python
def es_4_greedy_13(pompe, L):
    n = len(pompe)
    fermate = 0 # inizializziamo a 0 il num. di fermate effettuate
    benz = L # inizializziamo a L il contatore dei chilometri
    D = [0]*n # inizializzo la lista delle distanze
    D[0] = pompe[0] # inserisco la prima distanza
    for i in range(1, n): # per tutte le pompe tranne la prima
        D[i] = pompe[i] - pompe[i-1] # prendo la distanza dalla pompa i a quella successiva
    for i in range(n-1): # per ogni pompa
        benz -= D[i] # decremento benz per la distanza per arrivare a questa pompa
        if benz == 0 or benz < D[i+1]:
            # se ho finito la benzina oppure la distanza alla pompa successiva è maggiore della benzina rimasta
            benz += L # rifaccio il pieno
            fermate += 1 # incremento il num. di fermate
    return fermate
```

Costo:

- L'inizializzazione di D ha costo Θ(n)
- Il primo e il secondo for vengono entrambi iterati n volte → Θ(n)

Costo tot: Θ(n)

Slide 13b (Greedy)

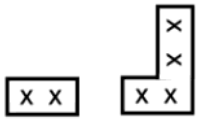
lunedì 26 maggio 2025 15:37

Slide 17 (Progr Dinamica Unidimensionale)

sabato 17 maggio 2025 20:16

Esercizio 1

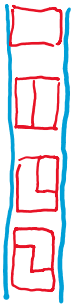
Dato l'intero n vogliamo contare il num di differenti tassellamenti di una superficie di dimensione n x 2 tramite tessere di domino dei due formati indicati di seguito



es n=3

Es:

- Per n = 1 la risposta è 1, in quanto l'unico possibile è il tassello rettangolare
- Per n = 3 la risposta è 7 perche ci sono 3 tassellamenti col solo tassello rettangolare e 4 tassellamenti che usano due tassellii di tipi differente



Soluzione

- Implementiamo una tabella unidimensionale di dimensione n+1 con la seguente regola per riempire le celle:
  T[i] = num. di tassellamenti possibili per la superficie di dimensione i x 2
- La soluzione sarà nella cella T[n]
- Le due tessere possono formare due tipi di tassellamenti:
  - La tessera rettangolare può essere posizionata con:
    - **1 tessera orizzontale** occupando uno spazio solo della superficie, quindi T[n-1]
    - **2 tessere verticali** occupando due spazi della superficie, quindi T[n-2]
  - La tessera ad L invece può essere posizionata con:
    - **1 tessera ad L e una rettangolare** che occupa tre spazi e può essere inserita in quattro modi diversi, quindi 4*T[n-3]
    - **2 tessere ad L** che occupa quattro spazi e può essere inserita in 2 modi diversi, quindi 2*T[n-4]
- Poiché esegue n-4 il caso generale deve iniziare per n ≥ 4, quindi dobbiamo inserire i 4 casi base per n = 0, 1, 2, 3. La formula sarà:

$$T[n] = \begin{cases} 1 & se\ n = 0, 1 \\ 2 & se\ n = 2 \\ 7 & se\ n = 3 \\ T[n-1] + T[n-2] + 4T[n-3] + 2T[n-4] & altrimenti \end{cases}$$

Implementazione:

```python
def es_1_PD_17(n):
    T = [0]*(n+1) # inizializziamo la tabella
    T[0] = T[1] = 1 # impostiamo i casi base
    T[2], T[3] = 2, 7
    if n < 4: return T[n] # ritorniamo i casi base
    for i in range(4, n+1): # per ogni i maggiore uguale di 4
        T[i] = T[i-1]+T[i-2]+4*T[i-3]+2*T[i-4] # applichiamo la formula
```

Costo: Θ(n)

Esercizio 2

Progettare una procedura che, presa in input la lista A dell'esericizio 2 e la tabella T prodotta dall'algoritmo proposto per la soluzione dell'esercizio 2, in tempo O(n) restituisca il sottovettore a somma massima

Es: A = [2, 0, -2, 1, -3, 1, 0, 4, -2, 3, -1, 2, -2, 1, -5, 1] la procedura restituisce [1, 0, 4, -2, 3, -1, 2]

Questo è l'algoritmo dell'esercizio 2

```python
def max_sottovettore(A):
    n = len(A)
    if n == 0: return 0 # se la lista è vuota ritorniamo 0
    T = [0]*(n+1) # inizializziamo la tabella
    T[0] = A[0] # inizializziamo il caso base
    for i in range(1, n+1): # per ogni posizione dell'array
        T[i] = max(A[i], A[i]+T[i-1]) # applichiamo la formula
    return max(T) # ritorniamo il valore massimo di T
```

che utilizza questo ragionamento:

- Una scelta che ci porta a definire n sottoproblemi è:
  T[i] = max somma possibile per le sottoliste di A che terminano in i
- Poiché la sottolista a val. max deve terminare in una posizione qualsiasi, il val. della soluzione sarà dato da $\max_{0 \le i \le n} T[i]$
- Definiamo la **regola ricorsiva**
  - **T[0] = A[0]** in quanto esiste una sola sottosequenza nell'array di un solo elem.
  - Per T[i], con i > 0, la sottolista di val. max che termina con A[i] può essere di due tipi:
    - **Consiste del solo elem. A[i]** (in questo caso ha solo val A[i])
    - **Ha lunghezza superiore a 1**. In questo caso vale A[i] + S, dove S è la somma massima di un sottovettore che termina in i-1 (S = T[i-1])
  - Pertanto abbiamo:

$$T[i] = \begin{cases} A[0] & se\ i = 0 \\ \max\{A[i], A[i] + T[i-1] & altrimenti \end{cases}$$

Soluzione

- L'esercizio 2 ritorna il valore massimo m di T che indica l'indice su cui finisce il sottovettore con somma massima.
  Ci sono due casi di inizio del sottovettore:
  - Il sottovettore parte dal primo elemento di A (quindi A[0]) e arriva all'indice di m
  - Il sottovettore parte dall'elemento A[i] poiché A[i]+T[i-1] < A[i] e arriva all'indice di m

  Poiché in entrambi i casi il sottovettore con somma massima inizia in qualche A[i] e continua a inserire elementi finchè non arriva a A[m] per trovare gli elementi del sottovettore basta partire da m e sottrarre/sommare a ritroso tutti gli elementi finche la m non arriva a 0.

Se troviamo in A[i] un elemento positivo allora lo sottraiamo, se è negativo lo sommiamo e ogni elemento incontrato lo aggiungiamo alla soluzione.
Posso usare in entrambi i casi l'operatore "-=" poiché nel caso x sia negativo eseguo "m -= -x" quindi esegue la somma di x

Implementazione:

```
def es_2_PD_17(A):
    # Es 2
    n = len(A)
    if n == 0: return 0 # se la lista è vuota ritorniamo 0
    T = [0]*(n) # inizializziamo la tabella
    T[0] = A[0] # inizializziamo il caso base
    for i in range(1, n): # per ogni posizione dell'array
        T[i] = max(A[i], A[i]+T[i-1]) # applichiamo la formula
    m = max(T) # ritorniamo il valore massimo di T
    ######
    sol = []
    i = T.index(m) # partiamo dall'indice del valore massimo di T
    while i >= 0 and m > 0: # finche non finiamo la lista o m arriva a 0
        sol.append(A[i]) # aggiungiamo il valore alla soluzione
        m -= A[i] # sommo/sottraggo il valore
        i -= 1 # decremento l'indice
    sol.reverse() # inverte la lista
    return sol
```

Costo:

- Ricerca indice del val. massimo → O(n)
- Il while nel caso peggiore si ferma quando finisce gli elementi della lista → O(n)

Costo totale: O(n)

Esercizio 3 (forse sbagliata)

Data una lista A con una serie di n tagli di monete ed un importo r da raggiungere, trovare il num. minimo di monete necessarie per ottenere tale importo. Se non è possibile ottenere quell'importo restituire -1

Es

- Per A = [5, 7, 10, 25] e r = 8 la risposta deve essere -1
- Per A = [5, 7, 10, 25] e r = 62 la risposta è 4 (utilizzando i 4 tagli 7, 5, 25, 25)
- Per A = [5, 7, 10, 25] e r = 49 la risposta è 4 (usa 4 tagli 7, 7, 10, 25)
- Per A = [5, 7, 10, 25] e r = 92 la risposta è 5 (usa 5 tagli 7, 10, 25, 25, 25)

L'algoritmo deve avere complessità O(n*r)

Soluzione

- Implementiamo una tabella unidimensionale di grandezza r+1 che contiene gli elem:
  T[i] = num. minimo di monete per ottenere i
- La soluzione sarà in T[r]
- T[i] sarà uguale a:
  - -1 come valore iniziale di ogni elem.
  - 1 per ogni elemento di A minore di r, quindi i casi base che non superano la grandezza della tabella
  - il minimo tra tutti i valori di T[i-A[j]], cioè il num. minimo di monete per il valore i a cui viene rimosso un elem. di A. Perciò prendiamo il valore minimo tra quelli trovati per i-A[j] e aggiungiamo 1 per indicare che serve una moneta in più (quella rimossa) per arrivare a i.
    Per poter controllare il minimo tra questi valori dobbiamo vedere se il valore in T[i-A[j]] esiste nella tabella (i-A[j] > 0) ed è diverso da -1

La formula sarà quindi:

$$T[i] = \begin{cases} 1 & se\ i = A[j]\ \ e\ A[j] < r\ per\ 0 \le j < n \\ \min\limits_{0 \le j < n} \left(T\left[i - A[j]\right]\right) + 1 & se\ T\left[i - A[j]\right] \ne -1 \wedge (i - A[j]) > 0 \\ 0 & altrimenti \end{cases}$$

Implementazione:

```
def es_3_PD_17(A, r):
    T = [-1]*(r+1) # inizializziamo la tabella a -1
    for i in range(1, r+1):
        m = float("inf") # inizializziamo il minimo a infinito
        if i in A: T[i] = 1# se è un elemento di A lo inizializziamo a 1
        else: # altrimenti controlliamo se possiamo raggiungere quell'importo
            for j in range(len(A)): # per ogni taglio di moneta
                if i-A[j] > 0 and T[i-A[j]] != -1: # se i-A[j] ha un num. minimo di monete (diverso da -1)
                    m = min(m, T[i-A[j]]) # prendiamo il minimo tra il minimo precedente e questo valore
            if m != float("inf"): # se abbiamo trovato un valore minimo che utilizza le monete
                T[i] = m+1 # impostiamo nella tabella quel valore
        # altrimenti rimane -1
    return T[r] # ritorniamo la soluzione
```

- Il for viene iterato r volte
  - Il costo dell'operatore "in" è O(n) poiché cerca se in A esiste quel valore
  - Il costo del for interno è O(n) poiché itera su tutti gli elem. di A

Costo totale: r*max(O(n), O(n)) = O(n*r)

Esercizio 4

Una **progressione aritmetica** è una sequenza di numeri in cui la differenza tra le coppie di numeri adiacenti è costante. Questa differenza costante è chiamata "**ragione**". Es, la sequenza 2, 5, 8, 11, 14, è una progressione aritmetica di ragione 3. Una sequenza di un solo elemento è una progressione aritmetica di qualunque ragione.

Dato un array di n interi positivi ed un intero c vogliamo contare le progressioni aritmetiche di ragione c che compaiono come sottosequenza dell'array

Es: per la sequenza 1, 3, 4, 6 e c = 2 la risposta è 6 infatti sono presenti le seguenti 6 sequenze di ragione 2: 1, 3, 4, 6, (1,3) e (4, 6)

L'algoritmo deve avere complessità $O(n^2)$

Soluzione:

- Utilizzeremo una tabella unidimensionale di grandezza n dove n è il num. di elem. nella sequenza, e dove il valore della cella i è:
  T[i] = num. di sottosequenze in cui si trova A[i] con elem. precedenti ad esso
- Il risultato sarà la somma dei valori in T + n (il num. di elem. che indica le sottosequenze di un solo elem)
- Se abbiamo una progressione aritmetica di tre elem. x, y, z, il risultato sarà 6 con le sequenze x, y, z, (x, y), (y, z) (x, y, z). Nelle sequenze composte con almeno due elem. vediamo che per transitività siccome (x,y) e (y,z) sono sottosequenze valide, allora (x, y, z) sarà una sequenza valida. Se per ogni elem. i salviamo in T[i] un valore che indica in quante sottosequenze si trova con gli elem. precedenti a i, se poi l'elem. successivo della sequenza j fa parte della progressione aritmetica con i, allora possiamo prendere il valore di T[i] e aggiungere 1 per indicare la nuova sottosequenza (i, j) alle altre sottosequenze di i.
  Prendendo il nostro esempio avremmo T[x] = 0, T[y] = 1 (cioè (x, y)) e T[z] = 2 (T[y] + 1, quindi (y, z) e la nuova (x, y, z))
- Quindi la regola ricorsiva per T[i] è:
  - Se la sottrazione tra l'elem. in i e gli altri elem. precedenti è c (cioè A[i] - A[j] = c per ogni j < i) allora è una sottosequenza valida e possiamo prendere T[j] (le sottosequenze con gli elem. precedenti a i-1) e aggiungere 1 (il nuovo elemento)
  - 0 se |A[i] - A[i-1]| ≠ c

  La regola sarà:

$$T[i] = \begin{cases} \sum(T[j]+1) & se\ A[i]-A[j]=c\ per\ 0 \le j < i \\ 0 & altrimenti \end{cases}$$

Implementazione

```python
def es_4_PD_17(A, c):
    n = len(A)
    T = [0]*n # inizializziamo la lista
    somma = 0 # inizializziamo la somma
    for i in range(n): # per ogni elem i
        for j in range(i): # per ogni elem. precedente a i
            if (A[i] - A[j]) == c: # se è una progressione aritmetica
                T[i] += T[j] + 1 # incrementiamo il valore di i con quello di j + 1 (la sottosequenza (i, j))
        somma += T[i] # incrementiamo la somma del num. di sottosequenze fino a i
    return somma + n # ritorniamo le sottosequenze composte + quelle da un solo elemento
```

- Il primo for itera i per n volte
- Il secondo for itera da 0 a i

Costo totale:

$$\sum_{i=0}^{n-1}\sum_{j=0}^{i-1}\Theta(1) = \Theta(1) * \sum_{i=0}^{n-1} i = \Theta(1) * \left(\frac{(n-1)*(n-2)}{2}\right) = \Theta(n^2)$$

Esercizio 5

Progettare un algoritmo che, dato un intero n, calcoli il num. di stringhe binarie lunghe 2n per le quali la somma dei primi n bit è uguale alla somma degli ultimi n bit
Es: per n = 2 per la risposta dell'algoritmo deve essere 6 infatti le stringhe di lunghezza 2*2 = 4 che soddisfano il vincolo sono: 0101, 0110, 1010, 1001, 0000, 1111
Es:

- n = 1 ⇒ sol = 2
- n = 2 ⇒ sol = 6
- n = 3 ⇒ sol = 20
- n = 4 ⇒ sol = 70
- n = 5 ⇒ sol = 252
- n = 6 ⇒ sol = 924
- n = 7 ⇒ sol = 3432
- n = 8 ⇒ sol = 12870
- n = 9 ⇒ sol = 48620

L'algoritmo deve avere complessità Θ(n)

Soluzione

bisogna usare il coeff. bin $\binom{2n}{n}$
Ora non mi va di farlo

Esercizio 6

Progettare una procedura che, presa in input la sequenza S dell'esercizio 3 e la tabella T prodotta dall'algoritmo proposto per la soluzione dell'esercizio 3, in tempo O(n), restituisca la sequenza crescente più lunga
Es: per A = [10, 9, 2, 5, 3, 101, 7, 18] la procedura deve restituire [2, 5, 7, 18]

L'esercizio 3 ha queste regole per la **lunghezza massima** per le sottosequenze crescenti presenti in S:

- Usiamo una tabella monodimensionale di dim. n e definiamo le celle per:
  T[i] = lungh. max per una sottosequenza crescente che termina in S[i]
- La soluzione del problema è il valore massimo nella tabella, quindi $\max_{0\le i<n} T[i]$
- Resta da definire la regola ricorsiva
  Possiamo dividere il valore della cella in due modi:
  - Se il val. in S[i] è più **piccolo** di tutti i suoi valori che lo precedono allora la sottosequenza crescente fino a quell'elemento sarà di lunghezza 1 perché conterrà solo l'elemento S[i]
  - Altrimenti prendiamo la lunghezza massima salvata negli elementi precedenti di T[i] (solo per elementi più piccoli di S[i]) e aggiungiamo 1 per indicare l'inserimento dell'elemento corrente alla sottosequenza

  Quindi la regola sarà:

$$T[i] = \begin{cases} 1 & se\ S_j > S_i\ per\ ogni\ 0 \le j < i \\ \max\limits_{0\le j<i\ con\ S_j<S_i}(T[i]) + 1 & altrimenti \end{cases}$$

Es: per A = [10, 9, 2, 5, 3, 101, 7, 18] abbiamo
T = [1, 1, 1, 2, 2, 3, 3, 4] e bisogna restituire [2, 5, 7, 18]

Soluzione

- Partiamo dall'indice i del valore massimo m in T, che indica la lunghezza massima per le sottosequenze crescenti in S.
- Nella regola dell'es 3, se per T[i] viene trovato un elem. S[j] minore di S[i] allora viene preso T[j]+1, incrementando la lunghezza della sottosequenza. Quindi per ogni elem. che troviamo che faccia parte della soluzione, dobbiamo decrementare di 1 il valore m di partenza.
- Iteriamo a ritroso partendo da i, sugli elem. di indice j e, per ogni val. minore di quello di partenza (quindi S[j] < S[i]), quando troviamo l'ultimo elem. in T[j] che sia uguale al val. m decrementato (nella tabella normale in realtà è il primo che incontriamo con quel valore), allora aggiungiamo quel valore alla soluzione e decrementiamo nuovamente m di 1.
- Per capire se l'elem. in T[j] è l'ultimo col val. m, per ogni elem. S[j] < S[i] e T[j] == m salviamo l'indice di j temporaneamente e controlliamo se l'indice precedente a j (T[j-1]) esiste e sia diverso da T[j], cioè che è un val. < m:
- Finita la tabella, dobbiamo poi aggiungere l'ultimo elem. della soluzione che non è stato aggiunto durante il ciclo

Implementazione:

```
def es_6_PD_17(S):
    n = len(S)
    if n == 0: return 0
    T = [0]*n # inizializziamo la tabella di T
    for i in range(1, n):
        for j in range(i): # per ogni elem. prima di i
            if S[i] > S[j]: # se l'elemento corrente è maggiore (crescente)
                T[i] = max(T[i], T[j]) # prendiamo il massimo tra le due lunghezze massime
        T[i] += 1 # aggiungiamo 1, cosi copriamo entrambe le regole, poichè:
                  # - Se tutti gli elem. precedenti erano più grandi allora passa da 0 a 1
                  # - Se abbiamo trovato una sottosequenza crescente aggiungiamo 1
    m = max(T) # ritorniamo il valore massimo della tabella
    #### Es 6
    i = T.index(m) # troviamo l'indice del valore massimo nella tabella
    succ = S[i] # salviamo l'elem. successivo nella sequenza per il controllo dei valori
    sol = [succ] # aggiungiamo quel valore alla soluzione
    m -= 1 # decrementiamo di 1 il valore massimo
    s = 0 # inizializziamo un candidato della sequenza
    while i >= 0: # finche non finiamo la tabella
        if S[i] < succ: # se la è un valore di una sequenza crescente
            if T[i] < m: # se abbiamo incontrato un elem.
                succ = s # aggiorniamo l'elem. della sequenza
                sol.append(succ) # lo inseriamo nella soluzione
                m -= 1 # e decrementiamo m
            s = S[i] # salviamo temporaneamente un candidato della sequenza
        i -= 1 # decrementiamo di 1 l'indice
    sol.append(s) # aggiungiamo l'ultimo elem. della soluzione
    sol.reverse() # invertiamo la lista
    return sol
```

Costo (solo dell'esercizio 6):

- Il ciclo itera dalla posizione del val max della tabella fino a 0 → costo O(n)

Esercizio 7

Il problema della sottosequenza crescente di valore massimo

Il valore di una sottosequenza è dato dalla somma dei suoi elementi. Data una sequenza S di n interi, vogliamo trovare il val. max tra quello delle sue sottosequenze crescenti

Es: data la sequenza S = 22, 3, 9, 4, 1, 5, 20, 6, 7, 2 una sottosequenza crescente di val 25 per S è 3, 4, 5, 6, 7.

Per S la soluzione al problema è 32 grazie alla sottosequenza 3, 9, 20 (in realtà c'è anche 3, 4, 5, 20)

A) Progettare un algoritmo che data una sequenza S di n elem. in tempo O(n^2) risolve il problema
B) Progettare un algoritmo che data una sequenza S di n elem. e la tabella T usata al punto A) per trovare il val. max, in tempo O(n), restituisca la sottosequenza di S di val max

Soluzione A

- Utilizzo una tabella unidimensionale di n celle, dove per ogni cella vale questa regola:
  T[i] = somma della sottosequenza crescente di valore massimo che finisce in i
- La soluzione sarà quindi il valore massimo della tabella, quindi $\max_{0 \le i < n} T[i]$
- La regola ricorsiva per T[i] è data da: $T[i] = S[i] + \max(T[j])\ per\ 0 \le j < i\ se\ S[i] > S[j]$
  Dove per ogni elemento della sequenza S[i] sommiamo il valore massimo tra le somme delle sottosequenze T[j] precedenti a S[i] per ogni elem. di S[j] che sia minore di S[i] (quindi possa creare una sequenza crescente)

Implementazione

```
def es_7_PD_17(S):
    max_sott = 0 # massimo della tabella
    n = len(S)
    T = [0]*n # inizializziamo a 0 la tabella
    for i in range(n): # per ogni elem. della sequenza
        m = 0 # inizializzo il massimo delle sottosequenze precedenti
        for j in range(i): # per ogni elem. precedente a i
            if S[i] > S[j]: # se è una sottosequenza crescente
                m = max(m, T[j]) # prendo il massimo tra m e la somma della sottosequenza massima che finisce in j
        T[i] = S[i] + m # sommo il valore della sottosequenza al massimo trovato
        max_sott = max(max_sott, T[i]) # aggiorno il massimo della tabella
    return max_sott # ritorniamo il massimo della tabella
```

Costo:

- Il primo ciclo itera i da 0 a n-1
- Il ciclo interno itera da 0 a i-1
- Il costo delle altre operazioni è costante

Costo tot:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i-1} \Theta(1) = \Theta(1) * \sum_{i=0}^{n-1} i = \Theta(n^2)$$

Soluzione B

Es: S = [22, 3, 9, 4, 1, 5, 20, 6, 7, 2] e

T = [22, 3, 12, 7, 1, 12, 32, 18, 25, 3] la soluzione è [3, 4, 5, 20] (oppure [3, 9, 20] dipende se l'algoritmo prende i massimi da sinistra a destra o viceversa)
Poiché la soluzione di A) è il valore massimo T[m] di una sottosequenza di S, basta partire da quel valore e andare a ritroso.
Però se cerchiamo direttamente il massimo per ogni val. precedente a m, aggiungerlo e continuare la ricerca da quel valore, il costo finale diventerebbe $O(n^2)$ poiché per ogni elem. m vado a cercare tra tutti gli elem. precedenti quale è il val. massimo in T (ad es. in una sequenza crescente tipo 1,2,3,4,5,6 partirei da 6 e cercherei il massimo tra tutti i val precedenti, poi da 5 e cosi via, con un totale di 15 iterazioni ($(\frac{6*(6-1)}{2} = 15)$)

Siccome il risultato di A è la somma dei vari elem. della sottosequenza col val. max, cioè la soluzione richiesta
Quindi un altro modo per trovare i valori massimi precedenti a m sta nel decrementare m di S[m] ogni volta e cercare il nuovo valore in T partendo da m a ritroso.
Quindi partendo dall'indice i_m del val. massimo della tabella m, decrementiamo da m il valore di S[i_m]. Cerchiamo poi, tra i valori j precedenti a i_m, il valore S[j] che coincide col nuovo m. Ora possiamo decrementare nuovamente da m il valore di S[j] e continuare fino ad arrivare all'ultimo elem. della tabella.
Ogni elem. che decrementiamo da m lo aggiungiamo alla soluzione poiché fa parte della soluzione richiesta.

Implementazione:

```
def es_7_PD_17(S):
    ####### Soluzione A
    max_sott = 0 # massimo della tabella
    n = len(S)
    T = [0]*n # inizializziamo a 0 la tabella
    for i in range(n): # per ogni elem. della sequenza
        m = 0 # inizializzo il massimo delle sottosequenze precedenti
        for j in range(i): # per ogni elem. precedente a i
            if S[i] > S[j]: # se è una sottosequenza crescente
                m = max(m, T[j]) # prendo il massimo tra m e la somma della sottosequenza massima che finisce in j
        T[i] = S[i] + m # sommo il valore della sottosequenza al massimo trovato
        max_sott = max(max_sott, T[i]) # aggiorno il massimo della tabella
    m = max_sott # prendiamo il massimo della tabella
    ######## Soluzione B
    i = T.index(m) # cerchiamo l'indice del valore massimo della tabella
    m -= S[i] # decrementiamo da m l'ultimo valore della sequenza da ritornare
    sol = [S[i]] # aggiungiamo quel valore alla soluzione
    while i >= 0: # finche non finiamo la tabella
        if T[i] == m: # se troviamo il val nella tabella che corrisponde al nuovo m
            m -= S[i] # decrementiamo il val della sequenza da m
            sol.append(S[i]) # aggiungiamo quel valore alla soluzione
            if m == 0: break # esci dal ciclo se abbiamo finito i valori della sequenza
        i -= 1 # decrementiamo di 1
    sol.reverse() # invertiamo la tabella
    return sol
```

Costo (solo della soluzione B):

- L'operatore "index()" controlla la lista finche non trova l'elem. → costo O(n)
- Il ciclo itera a partire dall'indice del val. max. della tabella fino a 0 (oppure si ferma prima se finisce la tabella)
  Nel caso peggiore (una sequenza crescente di partenza) itera dall'ultimo elem fino al primo, inserendo tutti gli elem. → costo O(n)

Quindi il costo nel caso peggiore è O(n)

Esercizio 8

Abbiamo un'asta lunga n ed una lista V ad n+1 componenti dove V[i] è il guadagno che si ottiene vendendo un'asta di lunghezza i. Vogliamo sapere qual è il guadagno massimo che è possibile ottenere vendendo l'asta di partenza dopo averla eventualmente partizionata
Es: per n = 8

- Se V = [0, 1, 5, 8, 9 ,10, 17, 17, 20] la risposta è 22 (guadagno che si ottiene vendendo 2 pezzi di lunghezza 2 e 6 rispettivamente)
- Se V = [0, 3, 5, 8, 9, 10, 17, 17, 20] la risposta è 24 (guadagno che si ottiene vendendo 8 pezzi di lunghezza 1)

Progettare algoritmo che data V in tempo $\Theta(n^2)$ risolva il problema

Soluzione

- Usiamo una tabella monodimensionale di grandezza n+1 dove:
  T[i] = guadagno massimo per un'asta di lunghezza i
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo un'asta di lunghezza 0, il guadagno massimo è 0
  - Se abbiamo un'asta di lunghezza 1, il guadagno è V[1]
  - Se abbiamo un'asta di lunghezza i > 1, il guadagno massimo per i è dato dal massimo tra la somma del guadagno di j e del guadagno massimo per i-j, con $1 \leq j \leq i$.
    Cioè tra i guadagni massimi delle varie combinazioni di coppie di interi minori di i che compongono i

  Quindi la regola ricorsiva è:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ V[i] & se\ i = 1 \\ \max\limits_{1 \leq j \leq i}(V[j] + T[i-j]) & altrimenti \end{cases}$$

Implementazione

```
def es_8_PD_17(V):
    l = len(V)
    T = [0]*(l) # inizializziamo la tabella
    T[1] = V[1]
    for i in range(1, l): # per l'asta lunga i
        for j in range(1, i+1): # per ogni j <= i
            T[i] = max(T[i], V[j]+T[i-j]) # prendiamo il max tra T[i] e la coppia che compone i (j, i-j)
    return T[l-1] # ritorniamo il massimo
```

Costo:

- La tabella viene creata con n valori → Θ(n)
- Il primo for itera i per n volte
- Il secondo for itera da 1 a i
- Le altre operazioni hanno costo costante → O(1)

Costo:

$$\sum_{i=1}^{n} i = \Theta(n^2)$$

Slide 18 (Progr Dinamica Bidimensionale)

sabato 17 maggio 2025 22:22

Esercizio 1 - Somma Dadi

Abbiamo n dadi distinti ciascuno con f facce e da un lancio di questi dadi vogliamo ottenere la somma s. Proggettare un algoritmo che in tempo Θ(n*s*f) restituisce il num. di modi diversi in cui questo è possibile

Es:

- Per n = 2, f = 6, s = 3 la risposta è 2 (1,2 2,1)
- Per n = 3, f = 2, s = 6 la risposta è 1 (2,2,2)
- Per n = 5, f = 5, s = 30 la risposta è 0 (nessuna combinazione possibile)
- Per n = 4, f = 5, s = 3 la risposta è 0 (nessuna combinazione possibile)
- Per n = 3, f = 5, s = 5 la risposta è 6 (1,1,3 1,2,2 1,3,1 2,1,2 2,2,1 3,1,1)

Soluzione

- Usiamo una tabella bidimensionale di grandezza (n+1) x (s+1), dove:
  T[k][t] = modi di ottenere la somma t con i primi k dadi
- Il risultato sarà in T[n][s]
- Formuliamo la regola ricorsiva:
  - Usando 0 dadi, otteniamo 0 in un solo modo (la sequenza vuota)
  - Usando 0 dadi, otteniamo la somma t > 0, in nessun modo
  - Poiché i dadi partono da 1, non possiamo mai ottenere la somma 0 con k > 0 dadi
  - In generale, se abbiamo k dadi, i modi per arrivare alla somma t sono la somma dei modi per ottenere la somma t meno le facce dei dadi minori uguali a t, usando k-1 dadi

  Quindi la formula ricorsiva è:

$$T[k][t] = \begin{cases} 0 & se\ k = 0\ e\ t > 0 \\ 0 & se\ k > 0\ e\ t = 0 \\ 1 & se\ k = 0\ e\ t = 0 \\ \sum_{v=1}^{f} T[k-1][t-v] & se\ k > 0, t > 0\ e\ t \geq v \end{cases}$$

Implementazione

Esercizio 2 - Stringhe Ternarie

Progettare un algoritmo che dato l'int positivo n in tempo O(n) conta il num. di stringhe ternarie lunghe n che non contengono né la sottostringa 02 ne la sottostringa 20 (quindi le cifre adiacenti differiscono al più di 1). Es:

- Per n = 1 la risposta è 3 (0, 1, 2)
- Per n = 2 la risposta è 7 (00, 01, 10, 11, 12, 21, 22)
- Per n = 3 la risposta è 17, perche delle 3^3 = 27 stringhe ternarie lunghe 3 le sole stringhe "vietate" sono: 020, 021, 022, 002, 102, 202, 200, 201, 120, 220
- Per n = 4 la risposta è 41
- Per n = 5 la risposta è 99
- Per n = 6 la risposta è 239
- Per n = 7 la risposta è 577
- Per n = 8 la risposta è 1393
- Per n = 9 la risposta è 3363

Soluzione

- Usiamo una tabella monodimensionale di grandezza n+1 x 3 (0, 1, 2) che contiene gli elem:
  T[i][c] = num di stringhe ternarie lunghe i che finiscono con c
- La soluzione sarà T[n][0] + T[n][1] + T[n][2]
- Formuliamo la regola ricorsiva
  - Nel caso in cui abbiamo una stringa di lunghezza 1, allora possiamo stampare solo 3 stringhe, una per carattere
  - L'inserimento di c alla fine di una stringa di lunghezza > 1 varia in base a c:
    - Se c = 0, allora può essere preceduto solo da 0 o 1
    - Se c = 1, allora può essere preceduto da 0, 1 o 2
    - Se c = 2, allora può essere preceduto solo da 1 o 2

    Quindi una stringa lunga i che finisce con c è formata dalla somma del num. di stringhe da cui può essere preceduto nella stringa lunga i-1

  Quindi la formula è:

$$T[i][c] = \begin{cases} 1 & se\ i = 1 \\ T[i-1][c] + T[i-1][c+1] & se\ c = 0 \\ T[i-1][c] + T[i-1][c] + T[i-1][c+1] & se\ c = 1 \\ T[i-1][c-1] + T[i-1][c] & se\ c = 2 \end{cases}$$

Implementazione:

```python
def es_2_PD_18(n):
    T = [ [0]*3 for _ in range(n+1) ] # inizializziamo la tabella
    T[1] = [1]*3 # impostiamo il caso base
    for i in range(2, n+1): # andiamo da 2 a n
        T[i][0] = T[i-1][0] + T[i-1][1] # 0 è preceduto da 0 o 1
        T[i][1] = T[i-1][0] + T[i-1][1] + T[i-1][2] # 1 è preceduto da 0, 1 o 2
        T[i][2] = T[i-1][1] + T[i-1][2] # 2 è preceduto da 1 o 2
    return T[n][0] + T[n][1] + T[n][2] # ritorniamo la somma al liv n
```

costo:

- La matrice itera per n volte la creazione di una lista di lunghezza 3 (quindi ha tempo costante) → Θ(n)
- Il for viene iterato n volte → Θ(n)

Costo totale: Θ(n)

Esercizio 3 - Stringhe con 4 Simboli

Progettare un algoritmo che dato un int n, restituisce il num. di stringhe lunghe n che è possibile ottenere con i 4 simboli 0, 1, 2, 3 e facendo in modo che nelle stringhe non compaiano mai due cifre dispari adiacenti. Es:

- Per n = 1, la sol è 4 (0, 1, 2, 3)
- Per n = 2 la risposta è 12, le stringhe lecite sono: 00, 01, 02, 03, 10, 12, 20, 21, 22, 23, 30, 32
- Per n = 3, la sol è 40
- Per n = 4, la sol è 128
- Per n = 5, la sol è 416
- Per n = 6, la sol è 1344
- Per n = 7, la sol è 4352
- Per n = 8, la sol è 14080
- Per n = 9, la sol è 45568

L'algoritmo deve avere complessità O(n)

Soluzione

- Usiamo una tabella bidimensionale di grandezza n+1 x 4 (0, 1, 2, 3) che contiene gli elem:
  T[i][c] = num. di stringhe lunghe i che finiscono con c
- La soluzione è la somma dell'ultimo livello T[n][0] + T[n][1] + T[n][2] + T[n][3]
- Formuliamo la regola ricorsiva:
  - Nel caso in cui abbiamo una stringa di lunghezza 1, le uniche stringhe che possiamo formare sono i 4 simboli di lunghezza 1: 0, 1, 2, 3.
  - Nella stringa di lunghezza i > 1 possiamo aggiungere:
    - 0 e 2 sempre, poiché possono essere preceduti da tutti e 4 i simboli
    - Invece 1 e 3 essendo dispari, possono essere preceduti solo da un num. pari, quindi da 0 e 2.

    Quindi il valore nella tabella T[i][0] e T[i][2] prendono la somma di tutti i valori al livello precedente i-1, mentre T[i][1] e T[i][3] prendono la somma dal livello precedente i-1 solo dalle colonne di 0 e 2

  Quindi la formula è:

$$T[i][c] = \begin{cases} 1 & se\ i = 0 \\ T[i-1][j] & se\ c\ è\ pari, con\ 0 \le j < 4 \\ T[i-1][j] & se\ c\ è\ dispari, con\ j = 0\ o\ j = 2 \end{cases}$$

Implementazione

```python
def es_3_PD_18(n):
    T = [ [0]*4 for _ in range(n+1) ] # inizializziamo la tabella
    T[1] = [1]*4 # impostiamo il caso base
    for i in range(2, n+1): # per i da 2 a n
        for c in range(4): # per c da 0 a 3
            if c%2 == 0: # se è pari (0 e 2) può essere preceduto da qualsiasi numero
                T[i][c] = sum(T[i-1]) # prendiamo la somma del livello precedente
            else: # se è dispari (1 e 3) può essere preceduto solo da 0 e 2
                T[i][c] = T[i-1][0] + T[i-1][2] # prendiamo la somma di 0 e 2 del liv. precedente
    return T[n][0] + T[n][1] + T[n][2] + T[n][3] # ritorniamo la somma al liv n
```

Costo:

- La tabella viene inizializzata usando un for che itera per n volte un altro for che itera 4 volte (quindi ha tempo costante) → Θ(n)
- Il primo for itera da 2 a n+1
- Il secondo for interno itera per 4 volte, quindi ha tempo costante → O(1)
- L'operazione sum itera i 4 elem. al livello i-1, quindi ha tempo costante → O(1)
- Il resto delle operazioni ha costo costante → O(1)

Costo totale: Θ(n)

Esercizio 4 - Sottostringhe Numeriche Divisibili per 3

Progettare un algoritmo che, data una stringa numerica decimale s lunga n, conta le sottostringhe numeriche di s divisibili per 3. Es:

- Per s = 1111 la risposta è 2 (le sottostringhe divisibili per 3 sono 111, 111)
- Per s = 33588 la risposta è 6 (le sottostringhe divisibili per 3 sono 3, 3, 33, 588, 3588, 33588)
- Per s = 3456 la risposta è 6 (le sottostringhe divisibili per 3 sono 3, 6, 45, 345, 456, 3456)

L'algoritmo deve avere complessità O(n)

Soluzione

Esercizio 5 - Cammini Matrice Somma

Dato un intero positivo k ed una matrice M con interi positivi di dim. n x n contare i cammini di costo k che in M partono da in alto a sx e arrivano in basso a dx
Ci si può spostare solo di un passo verso destra o un passo verso il basso e che il costo di un cammino è dato dalla somma dei val. delle celle toccate
Es: per A di seguito a sinistra con k = 12 la risposta è 2

$$A = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 6 & 5 \\ \hline 3 & 2 & 1 \\ \hline \end{array}$$

I due cammini sono:

$$1 \to 2 \to 6 \to 2 \to 1$$

$$1 \to 2 \to 3 \to 5 \to 1$$

Progettare un algoritmo che in tempo $\Theta(n^2k)$ risolva il problema

Soluzione

- Usiamo una tabella tridimensionale di grandezza n x n x (k+1), che contiene gli elem:
  T[i][j][s] = cammini da M[0][0] a M[i][j] con somma esattamente s
- Il risultato sarà in T[n-1][n-1][k]
- Formuliamo la regola ricorsiva:
  - La prima cella è quella di partenza, quindi il costo in T[0][0][M[0][0]] sarà 1
  - Gli altri cammini nella cella 0 con somma diversa da M[0][0] saranno 0
  - Un cammino che arriva in M[i][j] con somma s o viene da sopra (M[i-1][j]) con somma s - M[i][j] o da sinistra (M[i][j-1]) con somma s - M[i][j]

Quindi per ogni cella e per ogni cammino con costo s a quella cella, prende il costo del cammino s - M[i][j] da sopra (i-1) e da sinistra (j-1)

Ovviamente nel caso in cui siamo nella prima riga prendiamo solo dalle celle di sinistra, e se siamo nella prima colonna prendiamo solo dalle celle in alto

Quindi la regola è:

$$T[i][j][s] = \begin{cases} 1 & se\ i = 0, j = 0\ e\ s = M[0][0] \\ 0 & se\ i = 0, j = 0\ e\ s \neq M[0][0] \\ T[i-1][j]\big[s - M[i][j]\big] & se\ j = 0\ e\ s \geq M[i][j] \\ T[i][j-1]\big[s - M[i][j]\big] & se\ i = 0\ e\ s \geq M[i][j] \\ T[i-1][j]\big[s - M[i][j]\big] + T[i][j-1]\big[s - M[i][j]\big] & se\ j > 0, i > 0\ e\ s \geq M[i][j] \end{cases}$$

Implementazione

```python
def es_5_PD_18(M, k):
    n = len(M)
    T = [ [ [0]*(k+1) for _ in range(n) ] for _ in range(n)] # inizializziamo la tabella
    T[0][0][ M[0][0] ] = 1 # caso base
    for i in range(n):
        for j in range(n):
            if i != 0 or j != 0: # se non stiamo nella prima cella
                for s in range(k+1): # da 0 a k
                    up, left = 0, 0
                    if i > 0: # se sono dopo la prima riga
                        up = T[i-1][j][ s - M[i][j] ]# posso prendere da sopra
                    if j > 0: # se sono dopo la prima colonna
                        left = T[i][j-1][ s - M[i][j] ] # posso prendere da sinistra
                    T[i][j][s] = up + left # prendo da sopra e da sinistra
    return T[n-1][n-1][k] # ritorniamo il risultato
```

- L'inizializzazione della tabella ha costo $\Theta(n^2*k)$
- I 2 for iniziali iterano entrambi per n volte
- Il terzo for itera k volte
- Il resto delle operazioni ha costo costante

Costo totale: $\Theta(n^2*k)$

Slide 19 (Backtracking 1)

sabato 24 maggio 2025 18:59

Esercizio 1 - Esame Ottobre 2020

Progettare un algoritmo che, data una stringa X di lunghezza n, sull'alfabeto di {0, 1, 2} stampa tutte le stringhe di lunghezza n sull'alfabeto che differiscono da X in ciascuna posizione. L'algoritmo deve avere costo O(n*D(n)) dove D(n) è il num. di stringhe da stampare
Es: X = "020", deve stampare (non necessariamente nello stesso ordine): 101, 102, 111, 112, 201, 202, 211, 212

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Effettuiamo una funzione di taglio per controllare se durante l'inserimento del carattere x nella posizione i quel carattere non sia già in X[i]. In questo modo la scelta dell'inserimento di x nella soluzione è effettuata solo se il suo inserimento porta ad una sol da stampare di D(n)

```
def es_1_BK_19(X, sol=[]):
    if len(sol) == len(X): # se abbiamo una soluzione lunga n
        print("".join(sol)) # stampa la soluzione
        return # finisce la ricorsione
    i = len(sol) # prendiamo la posizione in cui inseriremo il carattere
    for j in range(3): # per ogni carattere in {0, 1, 2}
        if str(j) != X[i]: # se il carattere è diverso da quello in X nella posizione i
            sol.append(str(j)) # possiamo aggiungere il carattere j
            es_1_BK_19(X, sol) # continuiamo la ricorsione
            sol.pop() # rimuoviamo j
```

Nell'albero di ricorsione un nodo viene generato solo se porta ad una foglia da stampare
La complessità è:

$$O\big(D(n) * h * f(n) + D(n) * g(n)\big)$$

- D(n) → num di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) (poiché il for itera 3 volte, ha costo costante) → costo dei nodi interni
- g(n) = Θ(n) → costo dei nodi foglia

Il costo totale sarà: O(D(n)*n*O(1) + D(n)*n) = Θ(D(n)*n))

Esercizio 2 - Esame Gennaio 2021

Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe binarie lunghe n in cui non appaiono mai 3 simboli uguali consecutivi
L'algoritmo deve avere costo O(n*D(n)) dove D(n) è il num. di stringhe da stampare
Es: n = 3, deve stampare (non necessariamente nello stesso ordine): 001, 010, 011, 100, 101, 110

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Al passo ricorsivo i-esimo dobbiamo controllare con una funzione di taglio se possiamo inserire o meno l'elem 0 o 1. In entrambi i casi dobbiamo controllare se i due elem. precedenti (se esistono) siano uguali all'elemento che vogliamo inserire (0 o 1). Nel passo 1 e 2 possiamo inserire entrambi senza problemi, nei prossimi passi dobbiamo controllare le posizioni in i-1 e i-2.
  Quindi bisogna effettuare questo controllo (x è uno dei due caratteri 0 o 1):
  if i < 2 or (sol[i-1] != x or sol[i-2] != x)

```
def es_2_BK_19(n, sol=[]):
    if len(sol) == n: # se abbiamo una soluzione lunga n
        print("".join(sol)) # stampa la soluzione
        return # finisce la ricorsione
    i = len(sol) # prendiamo la pos in cui inseriamo il carattere
    if i < 2 or (sol[i-1] != "0" or sol[i-2] != "0"):
    # se siamo nelle prime 2 posizioni oppure non ci sono 2 simboli uguali a 0
        sol.append("0") # possiamo aggiungere 0
        es_2_BK_19(n, sol) # continuiamo la ricorsione
        sol.pop() # rimuoviamo 0
    if i < 2 or (sol[i-1] != "1" or sol[i-2] != "1"):
    # se siamo nelle prime 2 posizioni oppure non ci sono 2 simboli uguali a 1
        sol.append("1") # possiamo aggiungere 1
        es_2_BK_19(n, sol) # continuiamo la ricorsione
        sol.pop() # rimuoviamo 1
```

Nell'albero di ricorsione, un nodo viene generato solo se porta ad una soluzione valida
Il costo è:

$$O\big(D(n) * h * f(n) + D(n) * g(n)\big)$$

- D(n) → il num. di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) → costo dei nodi interni
- g(n) = Θ(n) → costo dei nodi foglia

Il costo totale è: Θ(D(n)*n)

Esercizio 3 - Stringhe Lunghe 2n

Progettare un algoritmo che dato un intero n, stampa tutte le stringhe binarie lunghe 2n per le quali la somma dei primi n bit è uguale alla somma degli ultimi n bit
Es: n = 2, bisogna stampare: 0101, 0110, 1001, 1010, 0000, 1111
L'algoritmo deve avere costo Θ(n*S(n)) dove S(n) è il num. di stringhe da stampare

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Per ottenere soltanto le soluzioni di S(n) dobbiamo effettuare una funzione di taglio per controllare se l'aggiunta di un elemento nella soluzione parziale porti ad una soluzione valida, generando solo i nodi necessari.
- Per i primi n bit non abbiamo bisogno di un controllo, pero conviene mantenere una variabile "totL" per indicare la somma dei primi n bit
- Bisogna invece controllare se durante l'aggiunta di un bit negli ultimi n bit non andiamo a superare la somma dei primi n bit. Manteniamo quindi anche una variabile "totR" per indicare la somma degli ultimi n bit

All'aggiunta di un elem. in pos i con n ≤ i < 2n (quindi negli ultimi n bit), dobbiamo vedere se all'aggiunta di quell'elemento alla somma di destra totR:
  - La nuova somma e gli elem. che si possono inserire nelle posizioni rimanenti (2n-i-1) siano maggiori o uguali alla somma di sinistra totL
  - La nuova somma sia minore o uguale alla somma di sinistra totL

  Quindi, prima di poter inserire un elem. generico x, dobbiamo effettuare prima il controllo:
  if i < n or ((totR + x + (2n - i - 1) ≥ totL) and (totR + x ≤ totL))
- Nella chiamata ricorsiva, nel caso in cui abbiamo aggiunto il valore 1, dobbiamo incrementare di 1 il valore di totL se ci tr oviamo nei primi n bit, oppure di totR se ci troviamo negli ultimi n bit

```python
def es_3_BK_19(n, sol=[], totL=0, totR=0):
    if len(sol) == 2*n: # se abbiamo una soluzione lunga 2n
        print("".join(sol)) # stampa la soluzione
        return # finisce la ricorsione
    i = len(sol)
    if i < n or ((totR + 0 + (2*n - i - 1) >= totL) and (totR + 0 <= totL)):
        # se siamo nei primi n bit oppure possiamo aggiungere 0 senza problemi
        sol.append("0") # aggiungiamo 0
        es_3_BK_19(n, sol, totL, totR) # non serve incrementare i contatori siccome aggiungiamo 0
        sol.pop() # rimuoviamo 0
    if i < n or ((totR + 1 + (2*n - i - 1) >= totL) and (totR + 1 <= totL)):
        # se siamo nei primi n bit oppure possiamo aggiungere 0 senza problemi
        sol.append("1") # aggiungiamo 1
        if i < n: es_3_BK_19(n, sol, totL+1, totR) # se siamo nei primi n bit incrementiamo il cont di sx
        else: es_3_BK_19(n, sol, totL, totR+1) # altrimenti siamo negli ultimi n bit e incrementiamo il cont di dx
        sol.pop() # rimuoviamo 1
```

Nota che nell'albero ricorsivo un nodo viene generato solo se può portare ad una soluzione valida di S(n)
Il costo è:

$$O\big(S(n) * h * f(n) + S(n) * g(n)\big)$$

- S(n) → num. di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) → costo dei nodi interni
- g(n) = Θ(n) → costo dei nodi foglia

Il costo sarà: Θ(S(n)*n)

Esercizio 4 - Esame Gennaio 2023

Data una stringa binaria s, la differenza tra il num. di 1 e il num. di 0 nelle prime i pos. di s è detto vantaggio alla pos i in s
Es: s = 010010111000 il vantaggio nelle dodici posizioni è rispettivamente: -1, 0, -1, -2, -1, -2, -1, 0, 1, 0, -1, -2
Dati due int positivi n e a, con a ≥ 1, con S(n, a) definiamo l'insieme di stringhe binarie di lunghezza n il cui vantaggio in ogni pos cade nell'intervallo che va da -a ad a
Es: S(5, 2) = {01010, 01011, 01100, 01101, 10010, 10011, 10100, 10101}
Trova un algoritmo che dati n e a stampi tutte le stringhe in S(n, a). La complessità deve essere O(n*|S(n,a)|)

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Il vantaggio alla pos i incrementa di 1 se viene aggiunto 1 e decrementa di 1 se viene aggiunto 0. Quindi per poter generare solo i nodi che portano a sol valide dobbiamo effettuare una funzione di taglio per controllare se l'inserimento di un valore non porti il vantaggio fuori dall'in tervallo |a|
  - Se aggiungiamo 0 in pos i dobbiamo vedere se decrementando di 1 il vantaggio fino ad i-1 esso non superi -a
  - Se aggiungiamo 1 in pos i dobbiamo vedere se incrementando di 1 il vantaggio fino ad i-1 esso non superi a
- Per non dover contare ogni volta il vantaggio fino alla pos i, teniamo una variabile "vant" che indica il vantaggio, che sara inizializzata a 0

```python
def es_4_BK_19(n, a, vant=0, sol=[]):
    if len(sol) == n: # se abbiamo una soluzione lunga n
        print("".join(sol)) # stampa la soluzione
        return # finisce la ricorsione
    if vant - 1 > -a: # se decrementando il vantaggio non usciamo dal range
        sol.append("0") # possiamo aggiungere 0
        es_4_BK_19(n, a, vant-1, sol) # decrementiamo il vantaggio
        sol.pop() # rimuoviamo 0
    if vant + 1 < a: # se incrementando il vantaggio non usciamo dal range
        sol.append("1") # possiamo aggiungere 1
        es_4_BK_19(n, a, vant+1, sol) # incrementiamo il vantaggio
        sol.pop() # rimuoviamo 1
```

Nota che nell'albero di ricorsione un nodo viene generato solo se può portare ad una soluzione valida di S(n, a)
Sia |S(n,a)| la cardinalità di S(n, a), il costo sarà:

$$O\big(|S(n,a)| * h * f(n) + |S(n,a)| * g(n)\big)$$

- |S(n,a)| → num. di stringhe da stampare
- h = n → altezza dell'albero
- f(n) = O(1) → costo di un nodo interno
- g(n) = Θ(n) → costo di un nodo foglia

Costo totale: O(|S(n,a)|*n)

Slide 20 (Backtracking 2)

domenica 25 maggio 2025 12:07

Esercizio 1 - Matrici con Righe Uguali

Algoritmo che preso intero n, stampa tutte le matrici n x n con val in {a, b, c} con la proprietà che i simboli in ogni riga sono tutti uguali
Es: n = 2 le matrici da stampare sono:

$$\begin{matrix} a & a \\ a & a \end{matrix} \quad \begin{matrix} a & a \\ b & b \end{matrix} \quad \begin{matrix} a & a \\ c & c \end{matrix} \quad \begin{matrix} b & b \\ a & a \end{matrix} \quad \begin{matrix} b & b \\ b & b \end{matrix}$$

$$\begin{matrix} b & b \\ c & c \end{matrix} \quad \begin{matrix} c & c \\ a & a \end{matrix} \quad \begin{matrix} c & c \\ b & b \end{matrix} \quad \begin{matrix} c & c \\ c & c \end{matrix}$$

L'algoritmo deve avere complessità $O(n^2S(n))$ dove S(n) è il num di matrici da stampare

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Alla riga i-esima per fare in modo che i simboli nella riga siano tutti uguali, basta riempire tutta la riga i con lo stesso simbolo. In questo modo siamo certi che non creiamo nodi che non portino a soluzioni non valide

```python
def es_1_BK_20(n):
    sol = [ [None]*n for _ in range(n) ] # inizializziamo la matrice vuota
    es_1_BK_20_r(n, sol) # riempiamo la matrice

def es_1_BK_20_r(n, sol, i=0):
    if i == n: # se abbiamo finito le righe
        for row in range(n): # allora per ogni riga
            print(sol[row]) # stampiamo la riga
        print() # stampa a capo
        return # finisce la ricorsione
    for j in range(n): # per ogni colonna
        sol[i][j] = "a" # inseriamo a
    es_1_BK_20_r(n, sol, i+1) # andiamo alla riga successiva
    for j in range(n): # per ogni colonna
        sol[i][j] = "b" # inseriamo b
    es_1_BK_20_r(n, sol, i+1) # andiamo alla riga successiva
    for j in range(n): # per ogni colonna
        sol[i][j] = "c" # inseriamo c
    es_1_BK_20_r(n, sol, i+1) # andiamo alla riga successiva
```

Nota che nell'albero ricorsivo ogni nodo generato porta ad una soluzione
Il costo sarà:

$$O\big(S(n) * h * f(n) + S(n) * g(n)\big)$$

- S(n) → num. di matrici da stampare
- h = n → altezza dell'albero
- f(n) = Θ(n) → costo di un nodo interno
- g(n) = $\Theta(n^2)$ → costo di un nodo foglia

Quindi il costo è: $O(S(n)*n^2)$

Esercizio 2 - Permutazioni Pari/Dispari Consecutivi

Algoritmo che preso intero pari n, stampa tutte le permutazioni dei primi n interi in cui nell'ordinamento non appaiono mai due pari/dispari consecutivi
Es: n = 4 deve stampare queste 8 permutazioni

1234 1432 2143 2341 3214 3412 4123 4321

L'algoritmo deve avere complessità $O(S(n)n^2)$ dove S(n) è il num di permutazioni da stampare

Soluzione

- Usiamo una funzione di backtracking per trovare le soluzioni
- Per non dover generare nodi che non portano a soluzioni valide, usiamo una funzione di taglio per controllare che non vengano inseriti due elem. pari o dispari consecutivamente. Se siamo nella prima posizione possiamo aggiungere qualunque elem. Se dobbiamo aggiungere un elem che non è stato ancora preso e siamo in una pos i > 0, allora possiamo aggiungerlo solo se:
  - l'elem. in pos i è pari e l'elem. in pos i-1 è dispari. Quindi bisogna vedere se il modulo dell'elem. in pos i è diverso dal modulo di quello in pos i-1
  - l'elem. in pos i è dispari e l'elem. in pos i-1 è pari. Quindi bisogna vedere se il modulo dell'elem. in pos i è diverso dal modulo di quello in pos i-1
- Per controllare se un elem. è stato preso o meno, usiamo una tabella "preso" di grandezza n che indica se l'elem. x è stato preso o meno.
  - preso[x] = 0 → l'elem. è stato preso
  - preso[x] = 1 → l'elem. non è stato preso
- Quindi per poter inserire un valore generico j in pos i bisogna controllare che:
  if i == 0 or (preso[j] == 0 and (j%2 != sol[i-1]%2))

```python
def es_2_BK_20(n):
    preso = [0]*(n+1) # inizializziamo la tabella
    es_2_BK_20_r(n, preso) # creiamo le permutazioni

def es_2_BK_20_r(n, preso, sol=[]):
    if len(sol) == n: # se abbiamo una soluzione lunga n
        print(sol) # stampa la soluzione
        return # finisce la ricorsione
    i = len(sol)
    for j in range(1, n+1): # per ogni elem. da 1 a n
        # se siamo nella prima pos oppure
        # i moduli tra l'elem. da aggiungere e quello precedente sono diversi e
        # l'elem. da aggiungere non è stato preso
        if i == 0 or (preso[j] == 0 and (j%2 != sol[i-1]%2)):
            sol.append(j) # lo aggiungiamo alla soluzione
            preso[j] = 1 # lo impostiamo come preso
            es_2_BK_20_r(n, preso, sol) # continuiamo la ricorsione
            sol.pop() # rimuoviamo l'elem. j
            preso[j] = 0 # lo reimpostiamo come non preso
```

Nota che nell'albero di ricorsione vengono generati solo nodi che portano ad una soluzione valida, quindi il num. di nodi foglie è S(n)
Il costo è dato da:

$$O\left(S(n) * \left(h * f(n) + g(n)\right)\right)$$

- S(n) → num. di permutazioni da stampare
- h = n → altezza dell'albero
- f(n) = Θ(n) → costo dei nodi interni
- g(n) = Θ(n) → costo dei nodi foglia

Costo totale: $O(S(n)*n^2)$

Esercizio 3 - Permutazioni Punti Fissi

Algoritmo che, dati n e k con k ≤ n, stampa tutte le permutazioni dei primi n interi in cui compaiono almeno k punti fissi.
Per una permutazione un punto fisso è un elem. i che compare nella pos i della permutazione. Es per n = 5 la permutazione 2,1,4,3,0 contiene due punti fissi (1 e 3)
Es: n = 4 e k = 2 deve stampare queste 7 permutazioni:
0123, 0132, 0213, 0321, 1023, 2103, 3120
La complessità deve essere $O(n^2S(n))$ dove S(n) sono le permutazioni da stampare

Soluzione

- Usiamo una funzione di backtracking, a cui aggiungiamo una funzione di taglio per poter ottenere solo le permutazioni da stampare
- Per ottenere una complessità proporzionale alle S(n) permutazioni da stampare basta controllare che il num. di punti fissi nella sol. parziale sia almeno k
  Per fare cio usiamo una var "totPF" che tiene conto dei punti fissi che abbiamo inserito fino a i e una var "totNPF" che tiene conto dei val che non sono punti fissi che abbiamo inserito fino a i.
  Se l'elem. x inserito è uguale alla sua posizione allora è un punto fisso e incrementiamo di 1 il contatore totPF. Viceversa se non è un punto fisso incrementiamo di 1 il contatore totNPF.
- Durante l'inserimento se l'elem x che possiamo inserire:
  - Il totale dei punti fissi è gia ≥ k, allora possiamo aggiungere comunque x
  - È uguale alla posizione i in cui lo stiamo inserendo, allora è un punto fisso e possiamo sempre inserirlo
  - Altrimenti, se non è un punto fisso, possiamo inserirlo solo nel caso in cui il suo inserimento permette poi l'aggiunta di almeno k punti fissi. Dobbiamo quindi controllare:
    - Se l'elem. punta ad una posizione già presa allora puntera ad un altro valore che non è un punto fisso, quindi il totale dei punti non fissi incrementerebbe solo 1 in quel caso. Per controllare se l'aggiunta dell'elem. porta poi ad una sol valida bisogna controllare se la somma tra gli elem. rimanenti e i punti fissi trovati è maggiore o uguale alla somma tra i punti non fissi trovati e 1 (l'elem. che aggiungeremo)
    - Invece se l'elem. punta ad una posizione non presa allora puntera ad un val che non è un punto fisso che non abbiamo ancora preso, quindi il totale dei punti non fissi incrementerebbe di 2 in quel caso. Per controllare se l'aggiunta dell'elem. porta poi ad una sol valida bisogna controllare se la somma tra gli elem. rimanenti e i punti fissi trovati è maggiore o uguale alla somma tra i punti non fissi trovati e 2 (l'elem. che aggiungeremo e quello che verrà poi aggiunto)

  Bisogna quindi effettuare questo controllo:
  if totPF ≥ k or (se abbiamo gia inserito k punti fissi)
  i == x or (se è un punto fisso)
  (preso[i] == 1 and ((n-i-1) + totPF ≥ totNPF + 1) or (se punta ad una pos gia presa e gli elem. rimanenti + i punti fissi è almeno i punti non fissi + l'elem. x
  (preso[i] == 0 and ((n-i-1) + totPF ≥ totNPF + 2) (se punta ad una pos non presa e gli elem. rimanenti + i punti fissi è almeno i punti non fissi + l'elem. x + l'elem puntato da x)

Esercizi Preparatori all'Esame

lunedì 26 maggio 2025 18:37

PARTE ESONERO 1 (ES GRAFI)

Esercizio 1 - Grado

Dimostrare che in un grafo non diretto e connesso *G* ci sono sempre almeno due nodi che hanno lo stesso grado.

Soluzione

Esercizio 2 - Alberi

Dato un grafo diretto *G*, si spieghi come un suo nodo *u* con almeno un arco entrante ed almeno un arco uscente, possa finire nella foresta DFS prodotta da una visita in profondità di *G*, in un albero DFS che contiene solo *u*.

Soluzione

Esercizio 3 - Hamiltoniano

Un cammino Hamiltoniano è un cammino (orientato) che passa per ogni nodo esattamente una volta. Descrivere un algoritmo che dato un grafo diretto e aciclico *G*, determina se *G* contiene un cammino Hamiltoniano. La complessità dell'algoritmo deve essere O($n + m$).

Soluzione

Esercizio 4 - Lontani

Descrivere un algoritmo che, dato un grafo diretto *G* e un suo nodo *s*, restituisce il numero dei nodi raggiungibili da *s* che si trovano alla massima distanza da *s*. L'algoritmo deve avere complessità O($n+m$).

Soluzione

Esercizio 5 - Copertura

Dato un grafo non diretto *G* si vuole trovare un suo albero di copertura. Si propone il seguente algoritmo:

```
INPUT un grafo non diretto G = (V, E)
SOL <- Ø
R <- {s} dove s è un qualsiasi nodo di V
WHILE E ≠ Ø DO
    Estrai un qualsiasi arco {u, v} da E
    IF |{u, v} ∩ R| = 1  THEN
        SOL <- SOL ∪ {{u, v}}
        R <- R ∪ {u, v}
OUTPUT SOL
```

a. Dare un'implementazione dell'algoritmo di complessità O($n + m$).
b. Dire se l'algoritmo proposto risolve il problema e in caso affermativo spiegare perché l'algoritmo è corretto (meglio ancora dimostrarne la correttezza) mentre in caso negativo fornire un controesempio.

Soluzione

Esercizio 6 - MaxST

Si consideri l'algoritmo di Prim modificato in modo che venga scelto ogni volta l'arco di peso massimo anziché quello di peso minimo. Provare che l'algoritmo trova l'albero di copertura di peso massimo o fornire un controesempio.

Soluzione

Esercizio 7 - Accoppiamento

Un *accoppiamento* in un grafo non diretto *G* è un sottoinsieme di archi disgiunti di *G*. Dato *G* si vuole trovare un accoppiamento di massima cardinalità. Viene proposto il seguente algoritmo greedy

```
INPUT un grafo non diretto G = (V, E)
SOL ← Ø
WHILE E ≠ Ø DO
    Tra gli archi in E prendi un arco {x, y} per cui il nodo x ha grado minimo
    SOL <- SOL ∪ {{x, y}}
    Cancella da E tutti gli archi che hanno per estremo x o y
OUTPUT SOL
```

a. Dare un'implementazione dell'algoritmo di complessità O(n^2).
b. Mostrare che l'algoritmo non produce necessariamente un accoppiamento di massima cardinalità.
c. Mostrare che nel caso di grafi aciclici l'algoritmo è corretto.

Soluzione

Esercizio 8 - Conta

Dato un grafo diretto e pesato *G* ed un suo nodo *s*, l'algoritmo di Dijkstra calcola l'albero dei cammini minimi da *s* ad un qualsiasi altro nodo di *G* raggiungibile da *s*. In generale, tra il nodo *s* e un altro nodo *u* di *G* può esserci più di un cammino minimo. Dare lo pseudo-codice di un algoritmo che dato un grafo diretto *G* ed un suo nodo *s*, calcoli per ogni nodo *u* il numero di cammini distinti di peso minimo da *s* a *u*. *Suggerimento*: modificare l'algoritmo di Dijkstra. Discutere la complessità dell'algoritmo.

Soluzione

PARTE ESONERO 2 (ES GREEDY/PROGRAMMAZIONE DINAMICA/BACKTRACKING)

Esercizio 9 - CD

Si supponga di voler creare un CD di brani musicali contente il maggior numero di brani dalla nostra collezione di musica. Si assuma che la somma delle dimensioni dei brani nella collezione ecceda la capacità del CD. Il problema sta nel selezionare un sottoinsieme degli *n* brani musicali che abbia cardinalità massima e che possa essere memorizzato sul CD.

a. Descrivere un algoritmo greedy che risolve il problema, provarne la correttezza e valutare la complessità di una sua efficiente implementazione.
b. Si supponga di volere anche che i brani da memorizzare appartengano a cantanti differenti. Descrivere un algoritmo per questo problema.

Soluzione

a. Un idea semplice sta nell'ordinare i brani per ordine di grandezza e inserire i brani più piccoli finche non si finisce lo spazio o non si può piu inserire altro poiche i rimanenti brani superano la capacità residua.
   Prova per assurdo:
   Assumiamo per assurdo che la sol del greedy SOL non sia ottima, quindi tra le soluzioni migliori che hanno inserito più file nel disco prendiamo SOL* che ha più elementi in comune con SOL. Poiché SOL* non coincide con SOL, esiste:
   - Un file x che ∉ SOL e ∈ SOL* e che occupa più spazio di qualunque file in SOL (poiché gli elementi in SOL occupano meno spazio di quelli non presenti)
   - Un file y che ∈ SOL e ∉ SOL* (poiché SOL ⊄ SOL* infatti l'aggiunta di qualsiasi elem. a SOL porterebbe a superare la capacità del disco)

   Quindi posso eliminare da SOL* il file x e inserire il file y (poiché y è minore di x per via della regola del greedy) ottenendo un nuovo insieme di file che rispetta le capacità del disco ed ha un elem. in più in comune con SOL contraddicendo l'ipotesi che SOL* è quello con più elementi in comune con SOL

   Il costo di questa implementazione sarebbe Θ(n*log(n)) poiché il sort ha costo Θ(n*log(n)) e l'inserimento nel CD ha costo al massimo O(n)

Esercizio 10 - Continuo

Sia *V* un vettore di *n* interi. Si dice che *V* è *continuo* se per ogni indice *i*, $1 \leq i \leq n - 1$, vale $|V[i + 1] - V[i]| \leq 1$. Si dice *zero* del vettore un indice *k* tale che *V*[*k*] = 0.

a. Dato un vettore *V* di $n \geq 2$ interi continuo tale che *V*[1] < 0 e *V*[n] > 0, provare che *V* ha almeno uno zero.
b. Descrivere un algoritmo che, dato un vettore *V* di $n \geq 2$ interi continuo e tale che *V*[1] < 0 e *V*[*n*] > 0, trovi uno zero in tempo O(log*n*)

Es: V = [-2, -1, 0, 1, 2], output indice 3 (V[3] = 0)

Soluzione

Esercizio 11 - Somma

Sia *A* un vettore di *n* interi e *x* un intero. Descrivere un algoritmo che trovi, se esiste, una coppia di indici *i*, *j* tali che *A*[*i*] + *A*[*j*] = *x*. L'algoritmo deve avere complessità O(*n*log*n*).

Es: A = [1, 4, 5, 2, 7] e x = 9, output (1, 2) (o anche (3, 4))

Soluzione

- Ordiniamo la lista in ordine crescente
- Per ogni elem. A[i] cerchiamo con la ricerca binaria l'elem. A[j] = x - A[i], per decidere dove cercare abbiamo due scelte:
  - Se x-A[i] > A[i] allora l'elem. si trovera nella parte destra del vettore partendo da i
  - Se x-A[i] < A[i] allora l'elem. si trovera nella parte sinistra
- Se non troviamo nulla, allora non esiste per l'elem. A[i] un altro elem. A[j] t.c. A[i] + A[j] = x, allora continuiamo all'indice i successivo

Esercizio 12 - Sottosequenza

Dare lo pseudo-codice di un algoritmo che data una sequenza di interi *S* trova una sottosequenza crescente di *S* di somma massima. Discutere la correttezza dell'algoritmo. L'algoritmo deve avere complessità O(n^2).

Es: S = [1, 101, 2, 3, 100, 4, 5], output = [1, 2, 3, 100]

Soluzione

- Usiamo una tabella unidimensionale di grandezza n, dove:
  T[i] = somma. max di una sottosequenza crescente che finisce in i
- Il risultato sarà il valore massimo della tabella, cioè $\max_{0 \leq i < n} T[i]$
- Formuliamo la regola ricorsiva:
  - Nel caso in cui abbiamo un solo valore, allora l'unica sottosequenza che possiamo avere è di grandezza 1 e contiene solo quel valore, quindi la somma sarà il valore stesso
  - Altrimenti una sottosequenza massima che finisce in i è il valore di i aggiunto alla sottosequenza massima dei valori precedenti ad i che sono minori di i. Quindi a S[i] aggiungiamo la sottosequenza massima trovata tra i val j con 0 ≤ j < i di T[j] se S[i] > S[j]

  Quindi la formula sarà:

$$T[i] = \begin{cases} S[i] & se\ i = 0 \\ S[i] + \max(T[j]) & se\ S[i] > S[j]\ con\ 0 \leq j < i \end{cases}$$

f

Esercizio 13 - Prodotto

Dato un vettore *V* di *n* numeri razionali si vuole trovare un sottovettore il cui prodotto degli elementi sia massimo. Descrivere un algoritmo che risolve il problema in tempo O(*n*) e discuterne la correttezza

Soluzione

Esercizio 14 - Sistemazioni

Si supponga di avere *n* elementi identificati dagli interi 1,…,*n*, da sistemare in *p* posizioni {1,2,…,*p*} con $n \leq p$. Una sistemazione *S* è *lecita* se vale *S*[*i*] < *S*[*i* + 1] dove *S*[*i*] è la posizione assegnata all'elemento *i*. Si dispone di una matrice *n*×*p* di interi non necessariamente positivi dove *M*[*i*, *j*] rappresenta il valore che si ottiene assegnando la posizione *j* all'elemento *i*. Il valore di una sistemazione è dato dalla somma dei valori ottenuti dagli *n* posizionamenti scelti.

a. Descrivere un algoritmo che in tempo O(*np*) calcola il valore massimo che si può ottenere con sistemazioni lecite. Discutere la correttezza dell'algoritmo.
b. Modificare l'algoritmo in modo che, con un tempo aggiuntivo O(*n*), venga trovata una sistemazione lecita di valore massimo.

Soluzione

Esercizio 15 - Esami

Sia dato un insieme di n esami identificati dagli interi 1,2,…,n. Ogni esame i vale c_i crediti ed ha un grado di difficoltà d_i. Ogni studente può redigere il proprio piano di studio individuale scegliendo nella lista degli esami attivati un sottoinsieme di esami tali che la somma dei crediti corrispondenti sia almeno C. Descrivere un algoritmo che redige un piano di studi regolare di difficoltà (la somma dei gradi di difficoltà degli esami del piano) minima in tempo O(nC). Discutere la correttezza dell'algoritmo.

Es, dato l'insieme E con gli esami i, E[i] = (c_i, d_i) E = [(3, 4), (2, 2), (4, 5), (1, 1), (3, 3)], C = 6. L'output dovra essere (1, 5) dove i crediti sono 6 e la difficolta è 7 (anche (1, 2, 4) ha crediti 6 e difficolta 7)

Soluzione

Esercizio 16 - Pari

Dare lo pseudo-codice di un algoritmo che, preso in input un intero n, stampi tutti i sottoinsiemi di {1,2,…,n} che contengono almeno un elemento che è un numero pari. La complessità dell'algoritmo deve essere O(nS(n)), dove S(n) è il numero di sottoinsiemi da stampare

Soluzione

Esercizio 17 - ABC

Dare lo pseudo-codice di un algoritmo che, preso in input un intero n, stampi tutte le stringhe di lunghezza n sull'alfabeto {a, b, c} tali che il numero di occorrenze di a è minore o uguale al numero di occorrenze di b che a sua volta è minore o uguale al numero di occorrenze di c. Ad esempio se n = 3 allora l'algoritmo deve produrre (non necessariamente in quest'ordine): *ccc*, *bcc*, *cbc*, *ccb*, *abc*, *bac*, *bca*, *acb*, *cab* e *cba*. La complessità dell'algoritmo deve essere O(nS(n)), dove S(n) è il numero di stringhe da stampare.

Soluzione

Esercizio 18 - Matrici

Dare lo pseudo-codice di un algoritmo che, preso in input un intero n, stampi tutte le matrici n×n con valori in {a, b, c} tali che due elementi adiacenti (per riga o colonna) abbiano valori differenti. Ad esempio se n = 2, le matrici da stampare sono:

```
ab  ab  ab  ac  ac  ac  ba  ba  ba  bc  bc  bc  ca  ca  ca  cb  cb  cb
ba  bc  ca  ca  cb  ba  ab  ac  cb  cb  ca  ab  ac  ab  bc  bc  ba  ac
```

La complessità è O(n^2S(n)) dove S(n) è il num. di matrici da stampare

Soluzione

Esercizi sul Diario delle Lezioni

martedì 27 maggio 2025 12:46

Esercizio 1 - 16 Maggio

Progettare un algoritmo di programmazione dinamica che, dato un intero n, in tempo $O(n)$, calcoli il numero di stringhe sull 'alfabeto {0,1,2,3 } in cui non compaiono mai due cifre pari adiacenti.

Soluzione

Esercizio 2 - 16 Maggio

Progettare un algoritmo di programmazione dinamica che, data una sequenza A di n interi positivi ed un intero k, in tempo O(n k), calcoli il numero di sottosequenze di A la somma dei cui elementi sia k.

Soluzione

- Utilizzo una tabella bidimensionale di grandezza (n+1) x (k+1) dove:
  T[i][s] = num. di sottosequenze con i primi i elem. di A di cui la somma sia s
- La soluzione sarà T[n][k]
- Formuliamo la regola ricorsiva:
  - Esiste un solo modo per prendere i elem. e avere somma 0, prendere nessun elemento (insieme vuoto)
  - Esiste una sola somma che si può ottenere con 1 elem., la somma A[0]
  - Altrimenti se abbiamo s > 0, il num. di sottosequenze con i primi i elem. di A è dato dal fatto se prendiamo o meno l'elem A[i] per aggiungerlo alla sottosequenza che da somma s:
    - Se A[i] > s, allora è inutile che lo prendiamo siccome il suo valore supera la somma, quindi il num. di sottosequenze sarà quello dato dagli elem. precedenti (quindi T[i-1][s])
    - Altrimenti possiamo prendere A[i], e il num. di sottosequenze T[i][s] è dato dal num. di sottosequenze per gli elem. precedenti di i per quella somma (quindi T[i-1][s]) + il num. di sottosequenze per gli elem. precedenti di i per la somma meno il valore preso (quindi T[i-1][s-A[i]])

La regola è quindi:

$$T[i][s] = \begin{cases} 1 & se\ s = 0\ o\ se\ i = 0, s = A[0] \\ T[i-1][s] & se\ A[i] > s \\ T[i-1][s] + T[i-1][s - A[i]] & se\ A[i] \leq s \end{cases}$$

Implementazione

```python
def es_2_DL(A, k):
    n = len(A)
    T = [ [0]*(k+1) for _ in range(n) ] # inizializziamo la tabella
    for i in range(n):
        T[i][0] = 1 # impostiamo il num. di sequenze per somma 0
    T[0][A[0]] = 1 # impostiamo il caso base per il primo elem.
    for i in range(1, n): # per ogni elem. di A
        for s in range(k+1): # per ogni somma s <= k
            T[i][s] = T[i-1][s] # prendiamo il num. per gli elem. precedenti
            if s >= A[i]: # se è un valore che possiamo prendere
                # aggiungiamo il num. di sottosequenze con la somma senza quel valore
                T[i][s] += T[i-1][s-A[i]]
    return T[n-1][k] # ritorniamo il risultato
```

Costo:

- L'inizializzazione della tabella ha costo Θ(n*k)
- Il for per inizializzare la tabella ha costo Θ(n)
- i due for per il controllo hanno costo Θ(n*k)
- Le altre operazioni hanno costo costante O(1)

Costo tot: Θ(n*k)

Esercizio 3 - 22 Maggio

Progettare un algoritmo che, dato n, in tempo O(n) calcoli il numero di modi che ci sono per risalire una scala con n pioli t enendo conto che per ciascuno step si puo' risalire di 1, 2 o 3 pioli.

Ad esempio per n=4 la risposta deve essere 7. Ecco di seguito i diversi modi di procedere: (1111), (112), (121), (211), (22), (13),(31)

Soluzione

- Usiamo una tabella unidimensionale di grandezza n+1, dove:
  T[i] = modi per risalire una scala con i pioli, risalendo solo di 1, 2 o 3 pioli
- Il risultato sarà in posizione n ⇒ T[n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo 0 pioli, abbiamo un solo modo per risalire la scala, non facendo passi (cioè l'insieme vuoto)
  - Se abbiamo almeno 1 piolo, possiamo fare un solo passo
  - Se abbiamo i > 1 pioli, allora il num. di passi che possiamo effettuare dipende dal num. di passi effettuati per arrivare al piolo j precedente al passo effettuato per arrivare a i. Quindi se arriviamo a i con un solo passo, i modi per arrivare a i sono i modi per arrivare a i -1.
    Percio siccome per ogni piolo i possiamo risalire di 1, 2 o 3 pioli, il modo per arrivare a i sarà dato dal modo per arrivare ai pioli i-1, i-2 e i-3

La regola sarà:

$$T[i] = \begin{cases} 1 & se\ i = 0\ o\ i = 1 \\ \sum_{j=1}^{\min(i,3)} T[i-j] & altrimenti \end{cases}$$

oppure

$$T[i] = \begin{cases} 1 & se\ i = 0\ o\ i = 1 \\ 2 & se\ i = 2 \\ T[i-1] + T[i-2] + T[i-3] & altrimenti \end{cases}$$

Implementazione

Esercizio 4 - 22 Maggio

Progettare un algoritmo che, date due stringhe X e Y ciascuna di n caratteri, in tempo O(n^2), restituisca la lunghezza massi ma tra quelle delle sottosequenze comuni ad X e Y.
Ad esempio per X='abzcdcd' e Y='baccbdz' la risposta deve essere 4 (la sottosequenza comune piu' lunga è 'accd' o 'bccd')

Soluzione

- Utilizzo una tabella bidimensionale di grandezza (n+1) x (n+1), dove:
  T[i][j] = lunghezza massima tra i primi i caratteri di X e i primi j caratteri di Y
- Il risultato sarà in posizione T[n][n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo 0 elem. in X o in Y, allora la lunghezza massima sarà 0
  - Altrimenti, la lunghezza massima dipende se X[i] e Y[j] siano uguali o meno:
    - Se X[i] = Y[j], allora abbiamo un elem. in comune, percio prendiamo la lunghezza massima degli elem. precedenti di X e Y, cioè per le sottosequenze con gli elem. di X[i-1] e Y[j-1] e incrementiamo di 1 la lunghezza della sottosequenza
    - Se X[i] ≠ Y[j], allora non siamo su un elem. in comune e dobbiamo scegliere quale dei due mantenere, per fare ciò prendiamo il massimo tra le lunghezze massime tra le sottosequenze che finiscono negli elem. precedenti tra X o Y, cioè il massimo delle lunghezza delle sottosequenze che finiscono in X[i-1] oppure in Y[j-1]

La regola quindi è:

$$T[i][j] = \begin{cases} 0 & se\ i = 0\ o\ j = 0 \\ \max(T[i-1][j], T[i][j-1]) & se\ X[i] \neq Y[j] \\ T[i-1][j-1] + 1 & se\ X[i] = Y[j] \end{cases}$$

Implementazione

Esercizio 5 - 22 Maggio

data una matrice quadrata di lato n contenente interi non negativi a partire dalla cella (0,0) dobbiamo raggiungere la cella (n-1,n-1) e ad ogni step possiamo muoverci nella cella adiacente a destra o nella cella adiacente in basso. Il costo del cammino è dato dalla somma dei contenu ti delle celle toccate.
Progettare un algoritmo che, data la matrice M ed un intero k, in tempo O(n^2) restituisce True se esiste un cammino di costo esattamente k, False altrimenti

Soluzione

Mi sembra impossibile farlo in tempo O(n^2), poiché servirebbe una matrice tridimensionale per le posizioni della matrice e la somma k, quindi costo O(n^2*k)
Questo è un esercizio gia visto tra gli esercizi simili

Esercizio 6 - 22 Maggio

Abbiamo una scacchiera di lato n>4, con un cavallo posizionato nella cella (o,o) bisogna trovare un cammino del cavallo che l o porti a toccare tutte le n^2 celle una ed una sola volta. Gli spostamenti possibili tra celle consecutive del cammino sono quelli possibili in base alle regole di s postamento del cavallo. Progettare un algoritmo di backtracking che dato n restituisca un possibile cammino.

Soluzione

- Usiamo una funzione backtracking con una funzione di taglio per poter controllare che nell'albero ricorsivo gli unici nodi che verranno generati sono quelli che portano ad una soluzione valida
- Per controllare velocemente se una casella della scacchiera è stata visitata o meno, creo una matrice M n x n in cui M[i][j] = 0 se la casella non è stata visitata, 1 altrimenti.
- Teniamo conto anche di una tabella "sol" che contiene le coppie (i, j) che indicano le caselle attraversate. Quando sol contiene n^2 elem. allora abbiamo visitato tutte le caselle della scacchiera e possiamo stampare il cammino.
- La funzione di taglio effettua un controllo sulle caselle su cui il cavallo si può muovere o meno, cosi che non vada su caselle già visitate
  Se il cavallo si trova in posizione i, j si può movere di 3 passi in una forma ad L. Quindi le posizioni valide sono le seguenti le seguenti 8:
  (i-2, j-1), (i-2, j+1), (i-1, j-2), (i-1, j+2), (i+1, j-2), (i+1, j+2), (i+2, j-1), (i+2, j+1)
  Per ogni posizione in cui il cavallo può muoversi devo controllare che non esca dal bordo della scacchiera e che la posizione in cui si può muovere non sia stata già visitata. Se non è stata visitata allora possiamo continuare la ricorsione su quella casella, altrimenti controlliamo le altre posizioni valide.
  Se tutte le caselle valide in cui si può muovere sono già state visitate allora non possiamo muoverci e non facciamo nulla

Implementazione:

```python
def es_6_DL(n):
    M = [ [0]*n for _ in range(n) ] # inizializziamo la matrice
    M[0][0] = 1 # impostiamo la prima cella come visitata
    es_6_DLr(M, [(-2, -1), (-2, +1), (-1, -2), (-1, +2), (+1, -2), (+1, +2), (+2, -1), (+2, +1)]) # cerchiamo le soluzioni

def es_6_DLr(M, pos, i=0, j=0, sol=[(0, 0)]):
    if len(sol) == len(M)**2: # se le soluzioni sono n^2
        print(sol) # allora abbiamo trovato un nodo foglia
        return # ci fermiamo
    for pi, pj in pos: # per ogni mossa possibile
        inext, jnext = i+pi, j+pj
        if (inext >= 0 and inext < len(M)) and (jnext >= 0 and jnext < len(M)) and M[inext][jnext] == 0:
            # se la prossima posizione non esce dalla tabella e non è stata visitata
            M[inext][jnext] = 1 # visitiamo quella cella
            sol.append((inext, jnext)) # aggiungiamo la casella attraversata
            es_6_DLr(M, pos, inext, jnext, sol) # continuiamo la ricerca
            sol.pop() # rimuoviamo la mossa
            M[inext][jnext] = 0 # impostiamo la cella come non visitata
```

Costo:

- L'inizializzazione della tabella ha costo Θ(n^2)
- Il for che controlla le posizioni viene iterato 8 volte, e le op. al suo interno hanno costo costante → O(1)

Costo nel caso pessimo (nel caso in cui l'unica soluzione è l'ultima dopo aver provato tutte le altre possibili combinazioni:

$$O\big(S(n) * h * f(n) + S(n) * g(n)\big)$$

- S(n) → Il num. di soluzioni da stampare
- h = n^2 → altezza dell'albero
- f(n) = O(1) → costo di un nodo interno

- $g(n) = \Theta(n^2)$ → costo di un nodo foglia (deve stampare le sol. che sono n^2)

Il costo è quindi: $O(S(n)*n^2)$

Esercizi Programmazione Dinamica

martedì 3 giugno 2025 19:33

Esercizio 1

Viene dato in input un intero positivo n. Scrivere lo pseudocodice di un algoritmo che in tempo $O(n)$ restituisce il numero di stringhe binarie lunghe n in cui non compaiono mai uni consecutivi.

Ad esempio per $n = 4$ deve essere restituito 8, le stringhe possibili sono infatti:

0000 1000 0100 0010 0001 1010 1001 0101

Soluzione

- Usiamo una tabella bidimensionale di grandezza n x 2, dove:
  T[i][x] = num. di stringhe binarie lunghe i che finiscono con x e in cui non compaiono mai uni consecutivi, con x ∈ {0, 1}
- Il risultato sarà la somma dell'ultimo livello, quindi T[n-1][0] + T[n-1][1]
- Formuliamo la regola recorsiva:
  - Non esistono stringhe lunghe 0
  - Esiste una sola stringa lunga 1 che finisce col carattere 0 o 1
  - In una stringa lunga i > 1, l'elemento 0 si può trovare successivo alla stringa con i-1 caratteri che finisce con 0 o con 1. Mentre l'elemento 1 si può trovare successivo solo alla stringa con i-1 caratteri che finisce con 0, poiche altrimenti si avrebbero due 1 consecutivi

  Quindi la regola è:

$$T[i][x] = \begin{cases} 0 & se\ i = 0\ e\ x = 0\ o\ x = 1 \\ 1 & se\ i = 1\ e\ x = 0\ o\ x = 1 \\ T[i-1][0] + T[i-1][1] & se\ i > 1\ e\ x = 0 \\ T[i-1][0] & se\ i > 1\ e\ x = 1 \end{cases}$$

Soluzione.

Esercizio 2

Viene dato in input un intero positivo n. Scrivere lo pseudocodice di un algoritmo che in tempo $O(n)$ restituisce il numero di stringhe binarie lunghe n in cui non compaiono mai tre uni consecutivi.

Ad esempio per $n = 4$ deve essere restituito 8, le stringhe possibili sono infatti:

0000 1000 0100 0010 0001 1010 1001 0101 0011 1011 0110 1100 1101

per n = 4, deve restituire 13 non 8

Soluzione

Esercizio 3

Si consideri una sequenza di interi X. La cancellazione da X di un certo numero di elementi determina una *sottosequenza*. La sottosequenza è crescente se il valore dei suoi elementi è crescente. La lunghezza della sottosequenza è il numero di elementi che la compongono. Il valore della sottosequenza è dato dalla somma dei valori degli elementi che la compongono.
Data la sequenza $X = (x_1, x_2, \ldots x_n)$ a valori distinti si considerino i seguenti tre problemi:

a. Trovare la sottosequenza crescente di X di lunghezza massima.
Ad esempio per $X = (50, 4, 1002, 48, 3, 34, 30)$ la sottosequenza crescente di lunghezza massima è $(2, 3, 34)$.

b. Trovare la sottosequenza crescente di X di valore massimo.
Ad esempio per $X = (50, 2, 100, 1, 20, 30)$ la sottosequenza crescente di valore massimo è $(50, 100)$.

c. Trovare il numero delle sottosequenze crescenti di X.
Ad esempio per $X = (5, 3, 7, 8, 6)$ il numero di sottosequenze crescenti è 14 (le sottosequenze sono: (5), $(5, 7)$, $(5, 7, 8)$, $(5, 8)$, $(5, 6)$, (3), $(3, 7)$, $(3, 7, 8)$, $(3, 8)$, $(3, 6)$, (7), $(7, 8)$, (8), (6).

c. Dato l'intero k, con $k \leq n$, trovare il numero delle sottosequenze crescenti di X.
Ad esempio per $k = 2$ e $X = (5, 3, 7, 8, 6)$ il numero di sottosequenze è 7 (le sottosequenze sono: $(5, 7)$, $(5, 8)$, $(5, 6)$, $(3, 7)$, $(3, 8)$, $(3, 6)$, $(7, 8)$).

Progettare quattro algoritmi che risolvono i quattro problemi. La complessità dei primi tre algoritmi deve essere $O(n^2)$, la complessità del quarto deve essere $O(n^2k)$

Soluzione a

- Usiamo una tabella unidimensionale di grandezza n, dove:
  T[i] = lunghezza della sottosequenza crescente con i primi i elementi
- Il risultato sarà il valore massimo della tabella
- Formuliamo la regola ricorsiva:
  - Con 0 elementi la lunghezza massima è 0
  - Se abbiamo 1 solo elemento allora la sottosequenza sarà di lunghezza 1
  - Altrimenti per trovare la lunghezza massima degli i elementi dobbiamo prendere la sottosequenza crescente massima per gli elementi precedenti che siano minori dell'elemento corrente, quindi il $\max_{0 \leq j < i} T[j]$ se X[j] < X[i] e aggiungere 1 per indicare l'elemento corrente.

Quindi la regola è:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ 1 & se\ i = 1 \\ \max\limits_{\substack{0 \leq j < i \\ se\ X[j] < X[i]}} (T[j]) + 1 & altrimenti \end{cases}$$

Imp

Esercizio 4

Data una matrice quadrata binaria M di dimensione $n \times n$ si vuole sapere qual'è il massimo m per cui la matrice quadrata $m \times m$ di soli uni risulta sottomatrice di M.

- provare che il numero di sottomatrici quadrate presenti in una matrice quadrata$n \times n$ è $\Theta(n^3)$ calcolando esattamente il loro numero.
- Descrivere un algoritmo che, data la matrice M, risolve il problema in tempo $O(n^3)$.

Ad esempio: per la matrice $M =$

1	0	1	1	1
1	1	1	1	1
1	1	1	0	1
1	1	1	1	1
1	1	0	1	1

di dimensione 5×5 la risposta è 3 (e gli elementi della sottomatrice 3×3 sono in blu).

Soluzione

Esercizio 5

Data una matrice $n \times m$ di interi $M = [m_{i,j}]$, una M-lista è una sequenza $(m_{1,j_1}, m_{2,j_2} \dots m_{n,j_m})$ tale che $1 \leq j_1 \leq \dots j_m \leq m$. Il valore di una M-lista è la somma degli elementi che la compongono.
Progettare un algoritmo che trova un M-lista di valore minimo in $O(n \cdot m)$.

Soluzione

Esercizio 6

Dato un insieme di m stringhe binarie dette *primitive* ed una stringa di n bit X, vogliamo sapere se la stringa X si può ottenere dalla concatenazione di stringhe primitive.
Ad esempio: dato l'insieme di primitive $\{01, 10, 011, 101\}$, per la stringa $X = 0111010101$ la risposta è sì (due possibili soluzioni sono $011 - 10 - 10 - 101$ e $011 - 101 - 01 - 01$) mentre per la stringa $X = 0110001$ la risposta è no.

a) Descrivere un algoritmo che, data la stringa X e le m primitive, risolve il problema in $O((m + n) \cdot l)$ dove l è la lunghezza massima per le stringhe primitive.

b) Modificare l'algoritmo proposto in modo che, nel caso X sia ottenibile dalla concatenazione di primitive, vengano prodotti in output gli indici delle primitive la cui concatenazione genera X.
Ad esempio: data la stringa $X = 0111010101$ e le primitive $Y_1 = 01, Y_2 = 10, Y_3 = 011$ e $Y_4 = 101$, l'algoritmo deve produrre in output la sequenza $3, 2, 2, 4$ o la sequenza $3, 4, 1, 1$.

Soluzione

Esercizio 7

Data una matrice binaria di dimensioni $n \times n$ vogliamo verificare se nella matrice è possibile raggiungere la cella in basso a destra partendo da quella in alto a sinistra senza mai toccare celle che contengono il numero 1.

Si tenga conto che dalla generica cella (i, j) ci si può spostare solo nella cella in basso (vale a dire la cella $(i + i, j)$) o nella cella a destra (vale a dire la cella $(i, j + 1)$).

Progettare un algoritmo che risolve il problema in tempo $O(n^2)$

Ad esempio: per la matrice A la risposta deve essere SI mentre per la matrice B la risposta deve essere NO.

$A =$

0	0	0	0	0	1
0	1	0	1	1	1
0	0	0	1	0	1
0	1	0	0	0	0
0	0	0	0	1	0
1	1	0	1	0	0

$B =$

0	0	0	0	0	0
0	1	1	1	0	0
0	1	0	0	0	0
0	1	0	1	1	1
0	1	0	0	0	0
0	0	0	0	1	0

Soluzione

Esercizio 8

Data una matrice di interi positivi e dimensione $n \times n$ vogliamo contare il numero di cammini di costo k che partono dalla cella in alto a sinistra e raggiungono la cella in basso a destra.

Il costo di un cammino è dato dalla somma dei valori delle celle toccate dal cammino inoltre, nel corso del cammino, dalla generica cella (i, j) ci si può spostare nella cella in basso (vale a dire la cella $(i + i, j)$) o nella cella a destra (vale a dire la cella $(i, j + 1)$).

Progettare un algoritmo che risolve il problema in tempo $O(n^2k)$

Ad esempio: per la matrice

1	2	3
4	6	5
3	2	1

e $k = 12$ la risposta è 2. I cammini sono:

$$1 \to 2 \to 6 \to 2 \to 1$$

$$1 \to 2 \to 3 \to 5 \to 1$$

Soluzione

Esercizio 9

Data una matrice di dimensione $n \times n$ le cui celle sono numerate con numeri distinti che vanno da 1 a n^2, vogliamo trovare la massima lunghezza possibile per cammini che toccano celle con numerazione crescente e incremento di 1.

I cammini possono partire da una qualunque cella e, nel corso del cammino, dalla generica cella (i, j) ci si può spostare in una qualunque cella adiacente in orizzontale o verticale (vale a dire in una delle celle $(i, j+1), (i+1, j), (i, j-1), (i-1, j)$). La lunghezza di un cammino è data dal numero di nodi toccati dal cammino.

Progettare un algoritmo che risolve il problema in tempo $O(n^2)$

Ad esempio: per la matrice A la risposta è 1 mentre per la matrice B, grazie al cammino $2 \to 3 \to 4 \to 5 \to 6 \to 7$, la risposta è 6

$A =$

3	6	2
7	1	9
4	8	5

$B =$

9	7	6
8	2	5
1	3	4

Soluzione

Esercizio 10

Data una sequenza di n interi positivi X e un intero positivo s vogliamo trovare la più lunga sottosequenza di X di somma s.
Ad esempio se $X = (5, 2, 2, 6, 1, 7, 3, 5, 11, 3, 6)$ e $s = 25$, la più lunga sottosequenza è lunga 7 $(5, 2, 2, , 1, 7, 3, 5, , ,)$; se $X = (3, 3, 5, 13, 3, 5)$ e $s = 28$, non ci sono sottosequenze di somma 28.

- Dare lo pseudo-codice di un algoritmo che dati X e s ritorna la lunghezza massima di una sottosequenza di somma s di X in tempo $O(ns)$ (se non ci sono sottosequenze di somma s, ritorna 0).
- Dare poi lo pseudo-codice di un algoritmo che ritorna una sottosequenza di lunghezza massima per X e s.

Soluzione

Esercizio 11

Abbiamo una sequenza $S = (s_1, s_2, \ldots, s_n)$ di interi positivi.
Una sottosequenza S' di S si definisce *valida* se per ogni coppia di elementi consecutivi di S almeno un elemento della coppia compare in S'.
Il valore di una sottosequenza valida è la somma dei suoi elementi.
Ad esempio: per $S = (1, 2, 3, 5, 4, 6, 7)$, la sottosequenza $S' = (1, 3, 6)$ non è valida, mentre la sottosequenza $S' = (2, 5, 4, 7)$ è valida ed ha valore valore 18 e la sottosequenza $S'' = (2, 3, 4, 6)$ è valida ed ha valore 15.

- Descrivere un algoritmo che, data la sequenza S, calcola il valore minimo di una sottosequenza valida in tempo $O(n)$.
- Descrivere poi un algoritmo che trova una sottosequenza valida di valore minimo.

Soluzione

- Utilizziamo una tabella bidimensionale di grandezza 2 x n, dove:
  T[0][i] = val. minimo per la sottosequenza che finisce in i, dove i non è selezionato
  T[1][i] = val. minimo per la sottosequenza che finisce in i, dove i è selezionato
- Il risultato sarà il valore minimo dell'ultimo valore, cioe min(T[0][n], T[1][n])
- Formuliamo la regola ricorsiva:
  - Se abbiamo un solo elemento (i=0), allora se non viene selezionato il valore sarà 0, altrimenti il valore minimo è quello dell'elemento stesso
  - Altrimenti, il valore minimo sarà dato:
    - Se S[i] viene selezionato, il valore minimo è dato dal minimo del valore della sottosequenza in cui l'elemento precedente non è stato selezionato (T[0][i-1]) e della sottosequenza in cui è stato selezionato, a cui aggiungiamo il valore di S[i], poiché della coppia (S[i-1], S[i]) stiamo includendo l'elemento S[i]
    - Se S[i] non viene selezionato, il valore minimo è dato dal valore della sottosequenza in cui l'elemento precedente è stato preso (T[1][i-1]), poiche della coppia (S[i-1], S[i]) dobbiamo includere l'elemento S[i-1]

Quindi la regola è:

$$T[i][0] = \begin{cases} 0 & se\ i = 0 \\ T[1][i-1] & altrimenti \end{cases}$$

$$T[i][1] = \begin{cases} S[0] & se\ i = 0 \\ \min\big(T[i-1][0], T[i-1][1]\big) + S[i] & altrimenti \end{cases}$$

Implementazione

Esercizio 12

Data una sequenza S di interi positivi si vuole trovare una sottosequenza di S senza elementi in posizioni consecutive e la somma dei cui elementi sia massima. Ad esempio, per $S = (1, 5, 4, 6, 10, 3, 2, 9)$ una soluzione è $(-, 5, -, -, 10, -, -, 9)$ e vale 24 (anche $(1, -, 4, -, 10, -, -, 9)$ è una soluzione), mentre $(1, -, -, 6, 10, -, -, 9)$ che vale 26 non è una sottosequenza ammissibile perché contiene gli elementi 6 e 10 in posizioni consecutive.

- Dare lo pseudo-codice di un algoritmo che risolve il problema in $O(n)$.
- Dare poi lo pseudo-codice di un algoritmo che trova una sottosequenza ottimale.

Soluzione

- Usiamo una tabella unidimensionale di grandezza n, dove:
  T[i] = somma massima degli elementi che finiscono in i
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Per 0 elementi la somma è 0
  - Per 1 elementi la somma è il valore di quell'elemento
  - Altrimenti, la somma per gli elementi che finiscono in i (cioè in S[i-1], dipende se viene preso l'elemento i o meno:
    - Se prendiamo S[i-1], allora non possiamo prendere la somma massima che finisce in S[i-2] ma solo quella che finisce all'elemento S[i-3], poiche l'elemento i-1 è consecutivo all'elemento S[i-1]
    - Se non prendiamo S[i-1], allora la somma massima sarà quella che finisce in S[i-2]

    Quindi dobbiamo prendere il valore massimo tra la somma massima che finisce in S[i-2] e la somma massima che finisce in S[i-3] a cui aggiungiamo l'elemento S[i-1]

  Quindi la formula sarà

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ 1 & se\ i = 1 \\ \max(T[i-2] + S[i-1], T[i-1]) & altrimenti \end{cases}$$

Implementazione

Esercizio 13

Si considerino 3 stringhe A, B e C, di lunghezza n, m ed $n+m$, rispettivamente. Diciamo che C è l'intreccio di A e B se contiene tutti i caratteri di A e tutti i caratteri di B e l'ordine di tutti i caratteri delle stringhe individuali è preservato.

Ad esempio se $A = aabxxz, B = abxy$ e $C = abaaxbxyxz$ allora C è l'intreccio di A e B infatti: $C = abaaxbxyxz$

- Assumendo che A e B non hanno caratteri in comune, dare lo pseudocodice di un algoritmo che, in tempo $O(n + m)$, determina se C è l'intreccio di A e B.
- Dare lo pseudocodice di un algoritmo che, in tempo $O(nm)$, determina se C è l'intreccio di A e B.

Soluzione 1 (sbagliata?)

- Usiamo una tabella bidimensionale di grandezza 2 x (n+m), dove:
  T[0][i] = caratteri di A trovati in ordine nei primi i caratteri di C
  T[1][i] = caratteri di B trovati in ordine nei primi i caratteri di C
- Se i caratteri trovati di A sono n e quelli di B sono m, allora C è un l'intreccio di A e B, quindi dobbiamo controllare se T[0][n-1] = n e T[0][n-1] = m. Altrimenti C non è un l'intreccio di A e B
- Formuliamo la regola ricorsiva:
  - Per il primo elemento di C (C[0]), se C[i] = A[i], allora incrementeremo il contatore di A, mentre quello di B sarà 0, viceversa se C[i] = B[i]
  - Per gli elementi successivi, l'ordine dei caratteri da controllare per A e B sarà dato dai rispettivi contatori dei caratteri trovati, poiché essi indicheranno alla posizione del prossimo carattere da controllare:
    - Per A, la posizione del carattere da controllare è dato dal contatore dei caratteri di A trovati in precedenza T[0][i-1]:
      - Se il contatore è uguale a n allora abbiamo finito i caratteri di A da controllare e passiamo avanti il contatore, T[0][i] =T[0][i-1]
      - Altrimenti, dobbiamo controllare se il carattere corrente di C (C[i]) è uguale all'elemento in ordine di A da controllare.
        Se C[i] = A[T[0][i-1]], allora l'elemento C[i] sarà un carattere di A, quindi possiamo incrementare il contatore dei caratteri di A trovati (T[0][i] = T[0][i-1]+1), altrimenti può essere un carattere di B, allora portiamo avanti il contatore dei caratteri di A precedente (T[0][i] =T[0][i-1])
    - Per B, facciamo allo stesso modo, controllando però se il contatore è uguale a m
  - Nel caso in cui incontriamo un carattere di A o di B fuori ordine, il contatore rimarrà all'ultimo elemento valido trovato, quindi alla fine se il contatore sarà diverso dalla lunghezza della stringa, indica che C non è un intreccio di A e B

  La formula è:

$$T[0][i] = \begin{cases} 0 & se\ C[0] \neq A[0] \\ 1 & se\ C[0] = A[0] \\ T[0][i-1] & se\ i > 0\ e\ T[0][i-1] = n\ o\ C[i] \neq A[T[0][i-1]] \\ T[0][i-1]+1 & se\ i > 0\ e\ C[i] = A[T[0][i-1]] \end{cases}$$

$$T[1][i] = \begin{cases} 0 & se\ C[0] \neq B[0] \\ 1 & se\ C[0] = B[0] \\ T[1][i-1] & se\ i > 0\ e\ T[1][i-1] = m\ o\ C[i] \neq B[T[1][i-1]] \\ T[1][i-1]+1 & se\ i > 0\ e\ C[i] = B[T[1][i-1]] \end{cases}$$

Soluzione 1 (giusta?)

- Usiamo una tabella bidimensionale di grandezza (n+m), dove:
  T[i] = VERO se i primi i caratteri di C formano l'intreccio dei primi x caratteri di A e i-x caratteri di B
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Per tenere conto dei caratteri incontrati in C che fanno parte di A o B, usiamo due puntatori pa e pb che inizializziamo a 0
    Siccome assumiamo che A e B non hanno caratteri in comune, se in C l'ordine di A e B è mantenuto, C[i] sarà uguale ad almeno un carattere di A e B, ma non entrambi, quindi almeno uno dei due contatori incrementa mentre l'altro rimane uguale, poiché avremmo C[i] = A[pa] e C[i] ≠ B[pb] o viceversa
  - Per il primo elemento di C (C[0]), se C[0] = A[0], allora impostiamo a VERO il controllo di T[0] e incrementeremo il contatore di A, mentre quello di B sarà 0, viceversa se C[0] = B[0]. Altrimenti se C[0] ≠ A[0] e B[0], abbiamo gia che l'intreccio di C è fuori ordine, quindi impostiamo a FALSO T[0]
  - Per gli elementi successivi abbiamo:
    - Se il controllo per i valori precedenti di C (T[i-1]) è FALSO, allora in qualche punto il controllo dei caratteri di A o B non è andato a buon fine, quindi in C almeno una delle due stringhe è fuori ordine, percio potremmo ritornare direttamente FALSO oppure portare avanti il valore alla posizione successiva della tabella, poiché non è necessario effettuare i0] controlli sui caratteri successivi
    - Se non abbiamo finito i caratteri per entrambe le stringhe (pa < n e pb < m), allora dobbiamo controllare se il carattere corrente di C sia uguale ad almeno uno dei due caratteri da controllare di A o B, se C[i] è uguale a A[pa] (quindi diverso da B[pb]) oppure uguale a B[pb] (quindi diverso da A[pa]), allora il controllo è andato bene (T[i] = VERO) e incrementiamo di 1 il contatore della stringa con cui C[i] è uguale.
      Altrimenti se C[i] è diverso sia da A[pa] che da B[pb], allora almeno una delle due stringhe è fuori ordine, quindi impostiamo a FALSO il controllo
    - Se abbiamo finito i caratteri da controllare di A (pa = n), allora dobbiamo continuare a controllare i caratteri di B, quindi impostiamo a VERO il controllo se C[i] = B[pb] o FALSO se C[i] ≠ B[pb]
    - Viceversa se abbiamo finito i caratteri da controllare di B (pb = m), allora dobbiamo continuare a controllare i caratteri di A, quindi impostiamo a VERO il controllo se C[i] = A[pa] o FALSO se C[i] ≠ A[pa]

  Quindi la regola sarà

$$T[i] = \begin{cases} VERO & se\ C[0] = A[0]\ o\ C[0] = B[0] \\ FALSO & se\ C[0] \neq A[0]\ e\ C[0] \neq B[0] \\ FALSO & se\ T[i-1] = FALSO \\ VERO & se\ pa < n\ e\ pb < m\ e\ C[i] = A[pa]\ o\ C[i] = B[pb] \\ FALSO & se\ pa < n\ e\ pb < m\ e\ C[i] \neq A[pa]\ e\ C[i] \neq B[pb] \\ VERO & se\ pa = n\ e\ pb < m\ e\ C[i] = B[pb] \\ FALSO & se\ pa = n\ e\ pb < m\ e\ C[i] \neq B[pb] \\ VERO & se\ pa < n\ e\ pb = m\ e\ C[i] = A[pa] \\ FALSO & se\ pa < n\ e\ pb = m\ e\ C[i] \neq A[pa] \end{cases}$$

Implementazione

Esercizio 14

Si hanno n attività e per ogni attività, $1 \leq i \leq n$, l'intervallo temporale $[s_i, f_i)$ in cui l'attività dovrebbe svolgersi e il guadagno v_i che si ottiene dallo svolgimento dell'attività. Due attività i e j sono compatibili se gli intervalli temporali $[s_i, f_i)$ e$[s_j, f_j)$ sono disgiunti ed il valore di un sottoinsieme di attività è dato dalla somma dei valori delle attività del sottoinsieme. Vogliamo selezionare un sottoinsieme S di valore massimo di attività tra loro compatibili. Descrivere un algoritmo che risolve il problema in $O(n^2)$.

Esempio: A=[(1,3,5),(2,5,6),(4,6,3),(5,7,2),(6,8,4),(7,9,1)] dove (s_i, f_i, v_i)
la soluzione ottima è 12 (attività 1 (5), attività 3 (3) e attività 5 (4))

Soluzione

- Usiamo una tabella bidimensionale di grandezza n+1, dove:
  T[i]= valore massimo usando le prime i attività
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo 0 attività, il valore massimo è 0
  - Se abbiamo una sola attività, il valore massimo è quello dell'attività singola
  - Altrimenti il costo di un attività generica i (S[i-1]) è dato dal massimo tra il valore massimo usando le attività precedenti (T[i-1]) e il costo del valore massimo per le attività precedenti compatibili (cioe che l'inizio dell'attività i è maggiore uguale alla fine delle attività precedenti) più l'attività i stessa.

  Quindi la formula è:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ A[0][2] & se\ i = 1 \\ \max\limits_{0 \leq j < i}(T[i-1], T[j] + A[i-1][2]) & se\ i > 1\ e\ A[i-1][0] \geq A[j][1] \end{cases}$$

Implementazione

Esercizio 15

Una banda di tre ladri deve spartirsi il frutto di una rapina di n oggetti ciascuno caratterizzato da un intero positivo che ne rappresenta il valore. Sapendo che l'ammontare totale del bottino M `e divisibile per tre, descrivere un algoritmo che verifichi se `e possibile spartire gli n oggetti in parti di ugual valore e, in caso a↵ermativo, produca la spartizione. L'algoritmo deve avere complessit`a O(n · M²).

Esempio:

oggetti = [2, 3, 4, 6, 7, 8, 9]

M = 39 → target = 13

Una possibile partizione: [4, 9], [6, 7], [2, 3, 8]

oggetti = [1, 6, 4, 3, 4, 2, 1]

M = 21 → target = 7

Una possibile partizione è [1, 6], [4, 3], [4, 2, 1]

oggetti = [6, 4, 5, 3, 8, 1, 3]

M = 30 → target = 10

Possiamo partizionare [6, 4], pero non possiamo formare 10 con gli oggetti rimasti [5, 3, 8, 1, 3]

Soluzione

- Usiamo una tabella tridimensionale di grandezza n

Esercizio 16

Vogliamo visitare in sequenza n citt`a. Per la visita di ciascuna citt`a `e previsto il pagamento di una tassa a seguito del pagamento della tassa si riceve un bonus per l'esenzione dal pagamento di tasse successive. Ogni esenzione pu`o essere usata una sola volta.

a) Descrivere un algoritmo basato sulla tecnica della programmazione dinamica che, dati i costi t1, t2,...,tn delle n tasse determina la tassa totale minima che un viaggiatore deve pagare per visitare tutte le n citt`a. La complessit`a dell'algoritmo deve essere O(n²).
b) Per minimizzare la tassa totale viene proposta la seguente strategia greedy: all'arrivo nella generica citt`a i la tassa va pagate se e solo se non si dispone di esenzioni o si dispone di j ≥ 1 esenzioni e bisogna ancora visitare almeno j citt`a che richiedono una tassa superiore a ti. Verificare se la strategia greedy minimizza effettivamente i costi.

Soluzione

- Usiamo una tabella bidimensionale di grandezza (n+1) x n dove:
  - T[i][k] = tassa totale minima per le prime i citta con ancora j esenzioni

  In realtà visitando i città possiamo avere al massimo i buoni, quindi ci sono delle celle che non vengono usate
- Il risultato sarà il valore minimo dell'ultima riga, min(T[n])
- Formuliamo la regola ricorsiva:
  - Avendo visitato 0 città, non abbiamo buoni e il costo è 0. Con 0 città gli altri buoni sono impossibili, impostiamo il loro costo a infinito
  - Altrimenti il costo minimo per arrivare alla città i (C[i-1]) con j esenzioni è dato dal costo minimo nel caso in cui arriviamo alla città usando o meno un esenzione (se è possibile usarla).

    Possiamo arrivare alla città i dalla città precedente i-1 solo in due modi:
    - Senza usare l'esenzione, paghiamo la città i. di esenzioni usate di 1. Quindi possiamo arrivare da T[i-1][k-1] a cui aggiungiamo C[i-1]. Se arriviamo a i avendo 0 esenzioni, possiamo arrivarci solo usandone una (quindi da T[i-1][k+1]).
    - Usando l'esenzione, non paghiamo la città i. Quindi possiamo arrivare da T[i-1][k+1]. Se arriviamo a i avendo n esenzioni, possiamo arrivarci solo senza usarne una (quindi da T[i-1][k-1] + C[i-1])

    Il costo per arrivare alla città i sarà il minimo tra il costo in cui non usiamo l'esenzione e il costo in cui usiamo l'esenzione
  - Se siamo alla città i, non possiamo mai avere i buoni k > i, quindi li impostiamo a infinito

  Quindi la formula è:

$$T[i][k] = \begin{cases} 0 & se\ i = 0\ e\ k = 0 \\ \infty & se\ i = 0\ e\ k > 0 \\ \infty & se\ k > i \\ T[i-1][k+1] & se\ i = 0 \\ T[i-1][k-1] + C[i-1] & se\ i = n \\ \min(T[i-1][k+1], T[i-1][k-1] + C[i-1]) & altrimenti \end{cases}$$

Implementazione

Esercizio 17

Vengono dati in input tre interi positivi x1, x2 e x3, con x1 < x2 < x3, ed un intero positivo n.

a) Scrivere lo pseudocodice di un algoritmo che in tempo O(n) restituisce il numero di sequenze sull'alfabeto {x1, x2, x3} la somma dei cui elementi `e n.
   Ad esempio per x1 = 2, x2 = 4, x3 =8 e n = 10 la risposta deve essere 10, le uniche sequenze possibili sono infatti:
   2, 8 8, 2 2, 4, 4 4, 2, 4 4, 4, 2 2, 2, 2, 4 2, 2, 4, 2 2, 4, 2, 2 4, 2, 2, 2 2, 2, 2, 2, 2
b) Scrivere lo pseudocodice di un algoritmo che in tempo O(n) restituisce il numero di multiinsiemi sull'alfabeto {x1, x2, x3} la somma dei cui elementi `e n.
   Ad esempio per x1 = 2, x2 = 4, x3 =8 e n = 10 la risposta deve essere 4, gli unici multiinsiemi possibili sono infatti:
   2, 8 2, 4, 4 2, 2, 2, 4 2, 2, 2, 2, 2

Soluzione

-

Esercizio 18

Date due sequenze X = (x1,...,xn) e Y = (y1,...,ym) una supersequenza di X e Y `e una qualsiasi sequenza Z tale che sia X che Y sono sottosequenze di Z. Ad esempio, per le sequenze di lettere alberi e libri le seguenti sono supersequenze: alberilibri, albelibri, lialberi, a liberi.

- Dare lo pseudo-codice di un algoritmo che, date due sequenze X e Y , di lunghezze n ed m, calcola la lunghezza minima di una supersequenza di X e Y in O(nm).
- Dare poi lo pseudo-codice di un algoritmo che ritorna una supersequenza di lunghezza minima di X e Y .

Soluzione

Esercizio 19

In una sequenza S = (a1, a2 ...an) di interi positivi, l'intero ai rappresenta il prezzo di una certa merce fra i giorni, si vuole sapere qual'`e il giorno i in cui conviene comprare la merce ed il giorno j, con j ≥ i, in cui conviene rivenderla in modo da massimizzare il profitto. In altre parole siamo interessati a conoscere la coppia (i , j) con i ≤ j per cui risulta massimo il valore aj ai. Descrivere un algoritmo che risolve il problema in O(n) tempo.

Soluzione

Esercizio 20

Data una sequenza S = (a1, a2,...an) di interi, sia positivi che negativi ma non zero, una sottosequenza di indici 1 ≤ i1 < i2 ...< im ≤ n di S `e detta alternante se per ogni k = 1,...,m 1 il segno di aik `e diverso dal segno di aik+1 . Chiaramente ogni sottosequenza di lunghezza 1 `e alternant e. Data una sequenza S di interi (negativi e positivi ma non zero) vogliamo trovare una sottosequenza alternante di somma massima. Ad esempio, se S = (2, 3, 4, 5, 4, 3, 3 , 5, 2, 1) una sottosequenza alternante di somma massima `e quella di indici 4, 7, 8 con somma 7.

- Dare lo pseudo-codice di un algoritmo che calcola la somma massima di una sottosequenza alternante in O(n2)
- Dare poi lo pseudo-codice di un algoritmo che trova una sottosequenza alternante di somma massima (come lista degli indici).

Soluzione

Esercizio 21

Sia A = {a1, a2,...an} un insieme di n esami, dove per ogni i = 1, 2,...n, l'esame ai vale ci crediti e si supponga di avere per ogni esame ai un coeciente di che rappresenta il grado di dicolt`a dell'esame. Ogni studente pu`o redigere il proprio piano di studio individuale scegliendo ne lla lista degli esami attivati un insieme di esami tali che la somma dei crediti corrispondenti sia almeno P. Progettare un algoritmo che redige un piano di studi regolar e di dicolt`a minima in O(n · P)

Soluzione

Esercizio 22

Nella versione classica abbiamo uno Zaino di capacit`a C ed n oggetti ciascuno con un suo valore vi ed un suo peso pi, 1 ≤ i ≤ n. Possiamo mettere nello zaino un qualunque sottoinsieme degli oggetti il cui peso complessivo non superi la capacit`a C e vogliamo trovare il sottoinsieme di valore massimo. Nella versione rivisitata disponiamo di un numero arbitrario di copie di ciascuno degli n oggetti e quindi pi`u copie di uno stesso oggetto possono ess ere inserite nello zaino allo scopo di ottenere una soluzione di valore massimo. Ad esempio per uno zaino di capacit`a C = 18 e tre oggetti con valore e peso specif icati dalle seguenti tabelle

	p_1	p_2	p_3
peso	9	5	4

	v_1	v_2	v_3
valore	6	4	3

la combinazione di oggetti che massimizza il valore dello zaino rispettandone la capacit`a `e quella che prende due copie del secondo oggetto e due copie del terzo (questa combinazione ha peso 18 e valore 14).

- Dare lo pseudo-codice di un algoritmo che calcola il valore della soluzione ottima in O(nC).
- Dare poi lo pseudo-codice di un algoritmo che produce una soluzione ottimale restituendo in output il numero di copie di ciascun oggetto da inserire nello zaino (per l'esempio l'algoritmo restituir`a il vettore (0, 2, 2)).

Soluzione

Esercizio 23

Una stringa `e palindroma se non cambia leggendola da sinistra a destra o viceversa. Data una stringa S = a1a2 ...an di n car atteri si considerino i seguenti problemi:

a) Calcolare il minimo numero di caratteri che occorre inserire per rendere S palindroma.
b) Determinare la lunghezza della pi`u lunga sottostringa palindroma di S.

Progettare due algoritmi che risolvono i due problemi. La complessit`a degli algoritmi deve essere O(n2)

Soluzione

Esercizio 24

Si ha una sequenza di n carte, ciascuna carta ha come valore un numero intero. Due giocatori a turno prendono una della carte da uno degli estremi della sequenza. Al termine del gioco il punteggio di ciascun giocatore `e dato dalla somma dei valori delle carte da lui prese. Lo scopo del gioco `e ottenere il punteggio massimo. Dati gli n valori v1, v2,...vn della sequenza all'inizio del gioco, progettare un algoritmo che in tempo O(n2) determini se i l primo giocatore dispone di una strategia vincente. Un giocatore dispone di una strategia vincente se ha la possibilit`a di vincere qualunque sia la sequenza di mosse e↵ettuata dal suo avversario. (per

semplicit`a si pu`o assumere che n sia un numero pari).

Soluzione

Esercizio 25

Abbiamo una scacchiera composta di tre righe ed n colonne. Ogni casella della scacchiera `e contrassegnata da un intero posit ivo . In ciascuna casella della scacchiera `e possibile piazzare una pedina. Un piazzamento di pedine `e regolare se non prevede due caselle con pedina adiac enti in orizzontale o in verticale (l'adiacenza di caselle con pedine adiacenti in diagonale non crea invece alcun problema). Il valore del piazzamento `e dato dalla somma dei contrassegni delle caselle in cui `e stata posizionata una pedina.

- Progettare un algoritmo che, data la matrice M con il valore dei contrassegni delle 3 ×n caselle, determina in tempo O(n) il valore del piazzamento regolare di valore massimo.
- Assumendo che i contrasegni delle caselle possano avere anche valore negativo. Progettare un algoritmo che, data la matrice Mcon il valore dei contrassegni delle 3 ×n caselle, determina in tempo O(n) il piazzamento regolare di valore massimo.

Soluzione

Esercizio 26

Abbiamo una matrice M di interi (non necessariamente positivi) di dimensione n x n. Il valore di una sua sottomatrice (non necessariamente quadrata) `e dato dalla somma dei suoi elementi. Si vuole trovare una sottomatrice quadrata di M la somma dei cui elementi sia massima. Ad esempio pe r la seguente matrice di dimensione 4 x 4

-20	10	-10	30
15	-6	20	-10
20	10	5	0
1	-5	-10	20

la sottomatrice cercata ha valore 64 e dimensione 2 ×3, i suoi elementi sono sottolineati

a) Provare che il numero di sottomatrici (non necessariamente quadrate) presenti in una matrice n ×n `e $O(n^4)$ calcolando esattamente il loro numero.
b) Descrivere un algoritmo che, data la matrice M, risolve il problema in $O(n^3)$ tempo (la sottomatrice da restituire pu`o essere rappresentata dando le coordinate della sua cella in basso a destra, la sua altezza e la sua larghezza).

Soluzione

Esercizio 27

Abbiamo una scacchiera M composta di 4 righe ed n colonne. Ogni casella della scacchiera `e contrassegnata da un intero posit ivo. In ciascuna casella della scacchiera `e possibile piazzare una pedina. Un piazzamento di pedine `e regolare se non prevede due caselle con pedine adiac enti in orizzontale o in verticale (l'adiacenza di pedine in diagonale non crea invece alcun problema). Il valore del piazzamento `e dato dalla somma dei contra ssegni delle caselle in cui `e stata posizionata una pedina. Descrivere un algoritmo che, dati i valori delle 4 x n caselle della scacchiera, in O(n) tempo determina un piazzamento regolare di valore massimo.

Ad esempio: per la scacchiera

30	6	5	2	10
1	7	1	3	28
15	30	4	20	1
20	2	8	10	6

con $n = 5$ due possibili piazzamenti di pedine regolari sono

30	6	5	2	10
1	7	1	3	28
15	30	4	20	1
20	2	8	10	6

e

30	6	5	2	10
1	7	1	3	28
15	30	4	20	1
20	2	8	10	6

.

Il primo piazzamento vale 87, il secondo 90 ed esistono altri piazzamenti regolari di valore superiore.

Soluzione

Esercizio 28

Alice e Bob decidono di comunicare attraverso messaggi codificati. Viene concordato un codice binario comune. La lingua madre di Alice e Bob usa un alfabeto di sette lettere: A, B, C, D, E, F e G. Viene quindi deciso di assegnare un codice binario a ciascuna delle lettere suddette:

lettera	A	B	C	D	E	F	G
codifica	0	00	001	010	0010	0100	0110

Presto Alice e Bob realizzano che messaggi diversi sono codificati dalla stessa sequenza binaria. Ad esempio, quando arriva l a sequenza 00100, questa risulta ambigua in quanto pu`o essere stata prodotta da ADA, AF, CAA, CB oppure EA. Con sequenze binarie pi`u lunghe, il grado di amb iguit`a aumenta ulteriormente. Inoltre, non tutte le sequenze binarie corrispondono a messaggi: per esempio, 00101 non codifica alcun messaggio. Descrivere un algoritmo che, data una sequenza binaria S = (a1, a2 ...an), in tempo O(n), calcola il numero di messaggi diversi che hanno codifica S. Ad esempio, l a sequenza S = (0, 0, 0, 0, 0, 0) corrisponde a 13 messaggi distinti.

Soluzione

Programma

venerdì 21 febbraio 2025 15:07

Per ogni algoritmo nel programma è data una dimostrazione di correttezza e valutata la complessità di possibili implementazioni. Alla fine di ogni argomento sono elencati i testi consigliati in ordine di rilevanza. Gli appunti saranno pubblicati durante lo svolgimento del corso.

Grafi

Rappresentazione tramite matrici di adiacenza e liste di adiacenza. Visite in ampiezza (BFS) e in profondità (DFS). Albero di visita e classificazione degli archi di una visita. Componenti connesse e fortemente connesse, algoritmo di Tarjan. Ordinamento topologico. Distanze in un grafo.

[Appunti del corso. Testo 1: Cap. 3, 4. Testo 3: Cap. 22, 23. Testo 5: Cap. 11. Testo 6: Cap.7]

Greedy

Descrizione della tecnica e schema generale per dimostrare la correttezza di un algoritmo Greedy. Algoritmi per Selezione Attività. Cammini minimi in grafi pesati: algoritmo di Dijkstra. Minimo albero di copertura: algoritmi di Prim e di Kruskal, strutture dati per insiemi disgiunti. Algoritmi di approssimazione e quantificazione dell'errore. Il problema del ricoprimento tramite nodi (Vertex Cover). Codici di Huffman.

[Appunti del corso. Testo 1: Cap. 4, 5. Testo 3: Cap. 16, 23, 24, 25. Testo 4: Cap. 4, 11]

Divide et Impera

Descrizione della tecnica. Il problema del massimo sottovettore. Algoritmo per la ricerca della coppia di punti più vicini (Closest Pair). Il problema della mediana.

[Appunti del corso. Testo 1: Cap. 2. Testo 4: Cap. 5. Testo 5: Cap. 5, 10. Testo 6: Cap. 3]

Programmazione Dinamica

Descrizione della tecnica e confronto con Divide et Impera. Ricerca esaustiva e memoizzazione. Dal calcolo del valore ottimo al trovare una soluzione ottima. Il problema dello Zaino (Knapsack). Cammino più lungo in grafi aciclici (Longest Path in DAG). Pianificazione Attività (CPM). Sistemi con vincoli di differenza e cammini minimi in grafi con pesi anche negativi. Algoritmi di Bellman-Ford e di Floyd-Warshall. Il problema della massima sottosequenza comune (LCS). Distanza tra stringhe (Edit Distance). Prodotto di una sequenza di matrici (Matrix Chaining).

[Appunti del corso. Testo 1: Cap. 6. Testo 2: Cap. 5. Testo 3: Cap. 15, 24, 25. Testo 4: Cap. 6. Testo 5: Cap. 10. Testo 6: Cap. 6]

Backtracking

Descrizione della tecnica. Enumerazione di strutture semplici: sottoinsiemi, sequenze, permutazioni. Esplorazione dello spazio di ricerca, ovvero generazione delle possibili soluzioni di un problema: 3-colorazione, ciclo Hamiltoniano. Euristiche di taglio: Zaino.

[Appunti del corso. Test 1: Cap. 9. Testo 2: Cap. 7]

Appunti 23-24 e 15-16

lunedì 24 febbraio 2025 13:04

Gli appunti degli studenti CasuFrost e SimoneLid del 2023/24 differiscono dagli appunti del prof del 2015/16 in:

Argomenti Aggiunti nel 2023

1. **Ricerca di un ciclo**: viene trattata in entrambi gli indici del 2023, ma non è esplicitamente citata nel 2015.
2. **Pozzo Universale**: non è presente nell'indice del 2015.
3. **Ponti sui grafi non diretti**: era presente nel 2015, ma senza il focus specifico sui grafi non diretti.
4. **Contrazione di Vertici e C-radice di un Componente Fortemente Connesso**: non vengono menzionati nel 2015.
5. **Distanza fra Insiemi e Vettore dei Padri**: non compare nel 2015.
6. **Flussi di Grafi e Grafo Residuo**: nuovi concetti assenti nel 2015.
7. **Scheduling di Processi**: applicazione della programmazione dinamica non presente nel 2015.
8. **Cammini Colorati su una Scacchiera**: esercizio specifico di programmazione dinamica assente nel 2015.
9. **Cammino di Peso Massimo**: non menzionato nel 2015.

Argomenti Mancanti nel 2023

1. **Snodi Critici e Punti di Articolazione**: presenti nel 2015 come parte dell'analisi dei grafi.
2. **Biconnessione**: trattata nel 2015, ma assente negli indici del 2023.
3. **Algoritmo di Bellman-Ford**: non appare negli appunti del 2023.
4. **Algoritmo di Floyd-Warshall**: era presente nel 2015 per i cammini minimi tra tutte le coppie di nodi, ma non è menzionato nel 2023.
5. **Ricoprimento tramite nodi (VC) e algoritmi di approssimazione**: trattati nel 2015, ma non nel 2023.
6. **Backtracking**: nel 2015 era presente con esempi specifici (colorazioni, cicli Hamiltoniani, zaino, regine, cricca), ma non appare negli indici del 2023.
7. **Prodotto di Matrici in Programmazione Dinamica**: presente nel 2015, assente nel 2023.
8. **Sistemi con Vincoli di Differenza**: trattati nel 2015, non citati nel 2023.

Esami

lunedì 24 febbraio 2025 13:07

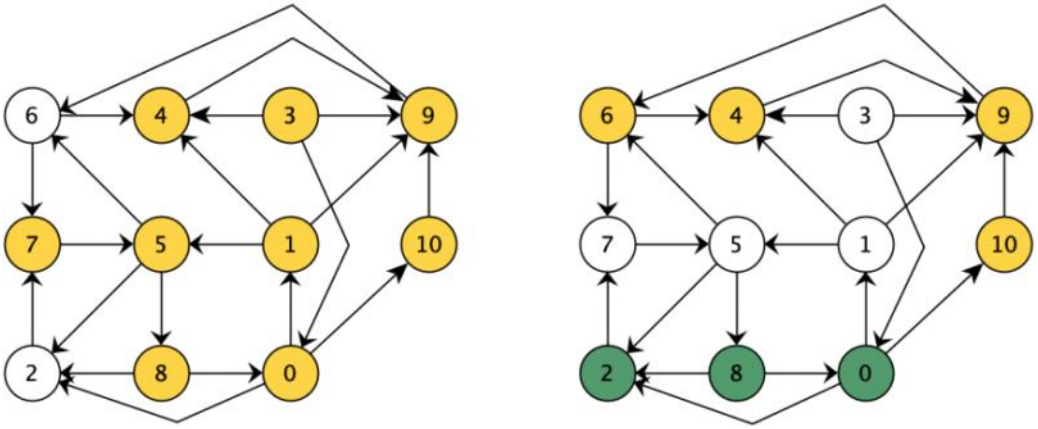
Molti esami passati e un esonero del 2022 stanno nelle risorse di EnzaSapi
Alcuni esercizi del 2015/16 sono nelle risorse fra tra gli appunti del prof e nella cartella esercizi

Esonero Aprile 2024

venerdì 14 marzo 2025 17:20

Esercizio 1

Dato un grafo diretto G con **n** nodi e **m** archi e due suoi nodi **a** e **b**, chiamiamo **equidistante** un qualunque nodo x di G che ha **uguale distanza** da **a** e da **b**, in caso contrario diciamo che x è nella **sfera di influenza** di quello dei due nodi che è a lui più vicino. Chiamiamo infine **vettore delle influenze** rispetto ai nodi a e b il vettore D di n componenti dove D[i] indica il nodo alla cui sfera d'influenza i appartiene e nel caso x sia equidistante da a e b allora D[i] = −1.
In figura ad esempio è riportato a sinistra un grafo con evidenziati in bianco i due nodi a = 2 e b = 6 e sulla destra i nodi colorati differentemente ad indicare le influenze e le equidistanze. In questo caso il vettore delle influenze è D = [2, −1, 2, −1, 6, −1, 6, −1, 2, 6, 6]. Progettare un algoritmo che, dato G tramite liste di adiacenza e i nodi a e b, in tempo O(n + m) restituisce il vettore delle 1 influenze



Esercizio 2

Le risposte alle domande vanno opportunamente giustificate. Non sono ammesse risposte troppo lunghe.

1. Sia T un minimo albero di copertura di un grafo pesato G. Sia G' il grafo che si ottiene da G incrementando di una stessa costante positiva c il peso di ciascun arco. L'albero T è un albero di copertura minimo anche per G'?
2. Sia G un grafo fortemente connesso e si consideri la visita in profondità di G a partire dal nodo 0. Tra archi all'indietro, archi in avanti e archi di attraversamento quali verranno di certo incontrati e quali possono invece anche non comparire?
3. Si consideri l'albero T ottenuto a seguito di una visita in ampiezza a partire dal nodo 0 di un grafo non diretto e connesso G. Sia a, b un arco di G non presente nell'albero T. Sia h(x) l'altezza del generico nodo x in T. Può aversi h[a] − h[b] > 2? La risposta cambia se il grafo G è diretto e l'arco va da a a b?
4. Quanti sort topologici sono possibili per un grafo diretto con n nodi ed un solo arco?
5. Considera un grafo connesso G in cui tutti i nodi hanno grado al più 3 qual'è il numero massimo di nodi che possono trovarsi a distanza esattamente d ≥ 1 dal nodo 0 di G? E quanti a distanza al più d?

Febbraio 1

mercoledì 28 maggio 2025 10:56

Esercizio 1

Dato un intero n, con $n \geq 2$, vogliamo contare il numero di modi in cui è possibile ottenere n partendo dal numero 2 e potendo effettuare le sole 3 operazioni di incremento di 1, prodotto per 2 e prodotto per 3.

Ad esempio per $n = 10$ la risposta è 9. Infatti le sequenze possibili sono:
(2,3,4,5,6,7,8,9,10), (2,3,4,5,10), (2,3,4,8,9,10), (2,3,6,7,8,9,10), (2,3,9,10), (2,4,5,10), (2,4,5,6,7,8,9,10), (2,4,8,9,10), (2,6,7,8,9,10).

Progettare un algoritmo che risolve il problema in tempo $O(n)$.

Soluzione

- Usiamo una tabella unidimensionale di grandezza (n+1), dove:
  T[i] = num. di modi per ottenere i partendo da 2 e effettuando solo incremento di 1 o prodotto per 2/3
- La risposta sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo n = 2, c'è un solo modo per partire da 2 e arrivare a 2, effettuando 0 operazioni
  - Se abbiamo i ≥ 2, allora il num. di modi per otterenere i è tramite i modi dei vari numeri che ottengono i tramite una delle tre operazioni. Dobbiamo quindi fare il contrario delle tre operazioni (quindi decremento di 1, divisione per 2 e divisione per 3) e se otteniamo un valore intero valido j con 2 ≤ j < i (quindi la divisione non deve dare un numero razionale) incrementiamo i modi per ottenere i per il num. di modi per ottenere j. Per controllare se la divisione non porta ad un num. razionale, effettuiamo la divisione per x solo se il modulo per x è uguale a 0.

  La regola sarà:

$$T[i] = \begin{cases} 1 & se\ i = 2 \\ T[i-1] & se\ i\%2 \neq 0\ e\ i\%3 \neq 0 \\ T[i-1] + T\left[\frac{i}{2}\right] & se\ i\%2 = 0\ e\ i\%3 \neq 0 \\ T[i-1] + T\left[\frac{i}{3}\right] & se\ i\%2 \neq 0\ e\ i\%3 = 0 \\ T[i-1] + T\left[\frac{i}{2}\right] + T\left[\frac{i}{3}\right] & se\ i\%2 = 0\ e\ i\%3 = 0 \end{cases}$$

Implementazione

Esercizio 2

Progettare un algoritmo che dato un intero n stampa tutte le matrici binarie quadrate $n \times n$ dove il numero di zeri presenti in ciascuna colonna è minore o uguale al numero di uni della colonna.

Ad esempio per $n = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 9 matrici:

$$\begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

L'algoritmo proposto deve avere complessità $O(n^2 S(n))$ dove $S(n)$ è il numero di stringhe da stampare.

Soluzione

- Usiamo un algoritmo di Backtracking con una funzione di taglio per generare solo nodi che portano ad una soluzione valida di S(n).
- Per controllare il num. di uni in una colonna possiamo mantenere un vettore "tot" di grandezza n in cui per ogni posizione indica il num. di i per quella colonna. Quindi tot[j] indica il num. di uni nella colonna j, cosi possiamo controllare il num. di 0 che possiamo inserire nella colonna j
- Nell'inserimento nella posizione i, j:
  - l'elem. 1 possiamo aggiungerlo sempre, pero aggiungendolo dobbiamo incrementare di 1 il num. di uni in tot[j]
  - L'elem. 0 invece possiamo aggiungerlo solo se il num. 0 in quella colonna è minore o uguale al num. di 1 in tot[j]. Per conoscere il num. di 0 già inseriti basta sottrarre dalla riga i il num. di 1 per quella colonna. Quindi il num. di 0 è dato da i - tot[j].

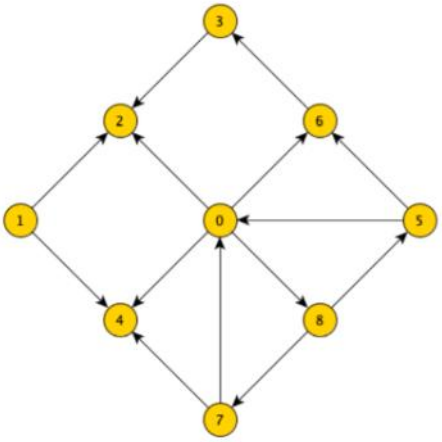
Implementazione

Esercizio 3

Considerate il grafo G in figura.

1. Quale è il numero minimo di archi da eliminare da G perché il grafo abbia sort topologici? **Motivare BENE la vostra risposta.**
2. Eliminate da grafo G il numero minimo di archi perché questi risulti avere sort topologici e applicate al grafo così ottenuto l'algoritmo per la ricerca del sort topologico basato sulla visita in profondità. Nell'eseguire l'algoritmo qualora risultino più nodi tra cui scegliere per proseguire la visita prendete sempre quella di indice minimo. Qual' è il sort topologico che si ottiene in questo modo?

G=



Progettazione d'algoritmi
Prof. Monti

Soluzione

Gennaio 1

mercoledì 28 maggio 2025 11:00

Esercizio 1

Data una sequenza S crescente di n interi, vogliamo trovare la lunghezza massima per le sottosequenze di S dove ciascun elemento è divisore del successivo.

Ad esempio:
per $S = [3,5,10,20]$ la risposta è 3 (la sottosequenza più lunga è $[5,10,20]$).
Per $S = [1,3,6,13,17,18]$ la risposta è 4 (la sottosequenza più lunga è $[1,3,6,18]$).

Progettare un algoritmo che risolve il problema in tempo $O(n^2)$.

Soluzione

- Usiamo una tabella unidimensionale di grandezza n dove:
  T[i] = lunghezza massima per i primi i elem. di S dove ciascun elem. è divisore del successivo
- La soluzione sarà il valore massimo della tabella, $\max_{0 \le i < n} T[i]$
- Formuliamo la regola ricorsiva
  - Se abbiamo un solo elem. allora la sottosequenza sarà formata da quell'elemento soltanto
  - Se abbiamo più elem. dobbiamo cercare la lunghezza massima per gli elem. precedenti a S[i] dove il modulo con S[j] con 0≤j<i è uguale a 0, quindi dove S[i] è divisibile per S[j]. Prendiamo l'elem. massimo poiche nella sottosequenza se il valore x è divisibile per il suo precedente x-1 allora è divisibile per tutti i valori precedenti a x-1.

  Quindi la regola ricorsiva è:

$$T[i] = \begin{cases} 1 & se\ i = 0 \\ \max\limits_{0 \le j < i} T[j] & con\ T[i]\%T[j] = 0 \end{cases}$$

Implementazione:

Esercizio 2

Progettare un algoritmo che data una stringa X lunga n sull'alfabeto $\{0,1,2\}$ stampa tutte le stringhe lunghe n sull'alfabeto $\{0,1,2\}$ che differiscono da X in ciascuna posizione e non hanno simboli adiacenti uguali.

Ad esempio per $X = 2001$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 5 stringhe:

$$0210 \quad 0212 \quad 0120 \quad 1210 \quad 1212$$

L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

Soluzione

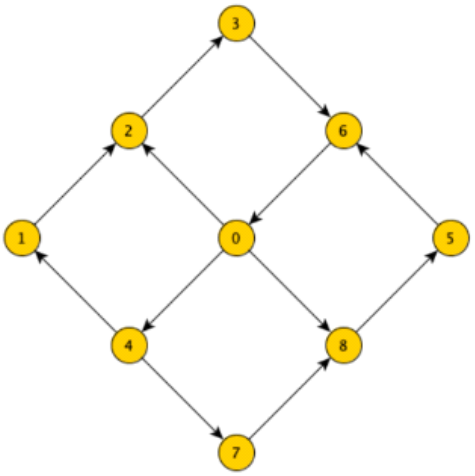
Esercizio già fatto tra gli esercizi delle slide di backtracking

Esercizio 3

Considerate il grafo G in figura.

1. Riportate l'albero dfs che si ottiene eseguendo una visita di G a partire dal nodo 0. Nel corso della visita ogni qualvolta un nodo ha più vicini non visitati scegliete sempre quello di indice minimo (in altre parole assumete il grafo rappresentato tramite liste di adiacenza dove i nodi di ciascuna lista sono ordinati per indice crescente).
2. dire quali sono gli archi di attraversamento, quali gli archi in avanti e quali gli archi all'indietro che si incontrano nel corso della visita.
3. Quale è il numero minimo di archi da eliminare da G perché il grafo abbia sort topologici? **Motivare BENE la vostra risposta.**
4. Eliminate da grafo G il numero minimo di archi perché questi risulti avere sort topologici e applicate al grafo così ottenuto l'algoritmo per la ricerca del sort topologico basato sulla visita in profondità. Nell'eseguire l'algoritmo qualora risultino più nodi tra cui scegliere per proseguire la visita prendete sempre quella di indice minimo. Qual' è il sort topologico che si ottiene in questo modo?

G=



Progettazione d'algoritmi
Prof. Monti

Soluzione

Giugno 1

mercoledì 28 maggio 2025 11:04

Esercizio 1

Data una sequenza S di n cifre decimali, vogliamo calcolare la lunghezza massima per una sottosequenza non decrescente che contiene i tre simboli 0, 1 e 2.

Ad esempio:

- per $S = 5,0,6,7,2,2,1,0,1$ la risposte deve essere 0 (non esistono infatti in S sottosequenze nondecrescenti che contengono i tre simboli 0, 1, e 2).
- per $S = 0,1,2,5,6,1,4,1,2,2,2,1,5,2$ la risposta deve essere 8 (infatti la sottosequenza non decrescente più lunga con tutti e tre i simboli è $0,1,_,_,_,1,_,1,2,2,2,_,_,2$.

Progettare un algoritmo che risolve il problema in tempo $O(n)$.

Soluzione

- Usiamo una tabella di grandezza 3 x n, dove:
  T[j][i] = lunghezza massima per una sottosequenza non decrescente formata dai primi i elem. che contiene i primi j simboli
- Il risultato sarà in T[2][n-1]
- Formuliamo la regola ricorsiva:
  - Una sottosequenza i che utilizza solo il primo simbolo 0 è data dal num. di 0 che abbiamo incontrato fino a i
  - Altrimenti una sottosequenza che contiene i primi j simboli può partire al minimo dall'elemento j della sequenza, quindi i precedenti sono 0. Dall'elemento j della sequenza in poi, se incontriamo:
    - Un elemento diverso dal simbolo in pos j, allora portiamo avanti la sequenza più lunga incontrata finora
    - Un elemento uguale al simbolo in pos. j, allora prendiamo la sequenza più lunga incontrata finora, scegliendo il massimo tra la sequenza che usa j simboli e quella che usa j-1 simboli per i primi i-1 elementi e aggiungiamo 1, per indicare l'elemento aggiunto. Possiamo incrementare la sottosequenza solo se abbiamo già costruito una sottosequenza non decrescente con i simboli precedenti, quindi se T[j-1][i] è maggiore di 0

Quindi la formula è:

$$\mathrm{T[0][i]} = \begin{cases} 0 & \text{se i} = 0 \text{ e S[0]} \neq 0 \\ 1 & \text{se i} = 0 \text{ e S[0]} = 0 \\ \mathrm{T[0][i-1]} & \text{se i} > 0 \text{ e S[i]} \neq 0 \\ \mathrm{T[0][i-1]} + 1 & \text{se i} > 0 \text{ e S[i]} = 0 \end{cases}$$

$$T[j][i] = \begin{cases} 0 & se\ i < j \\ \mathrm{T[j][i-1]} & se\ i > j\ e\ S[i] \neq j \\ \max(T[j][i-1], T[j-1][i-1]) + 1 & se\ i > j\ e\ S[i] = j\ e\ T[j-1][i] > 0 \end{cases}$$

Implementazione

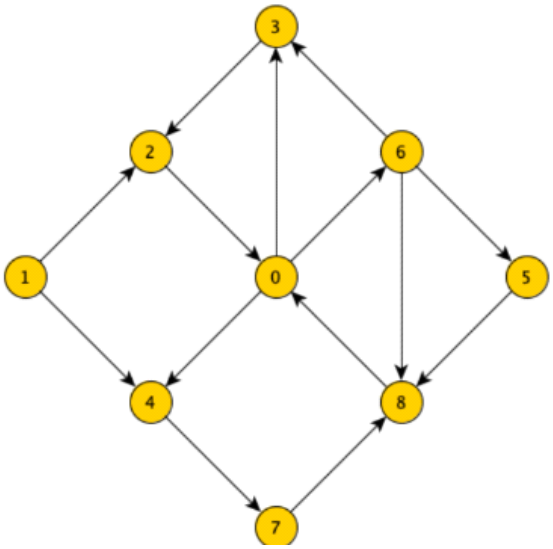
```python
def es_1_2_25(S):
    n = len(S)
    T = [ [0]*n for _ in range(3) ]
    if S[0] == 0: # se il primo elemento è 0
        T[0][0] = 1 # allora abbiamo una sottosequenza con l'elemento 0
    for j in range(3):
        for i in range(1, n):
            if j == 0: # se stiamo controllando il primo simbolo
                T[0][i] = T[0][i-1] # prendiamo il val. precedente
                if S[i] == 0: # se l'elemento i è 0
                    T[0][i] += 1 # incrementiamo di 1 il contatore degli 0
            else: # se stiamo controllando gli altri simboli
                if i >= j : # se possiamo avere una sequenza con gli j elementi
                    T[j][i] = T[j][i-1] # passiamo il valore della seq. con gli elem. precedenti
                    if S[i] == j and T[j-1][i] > 0: # se l'elem. è uguale al simbolo e
                    # abbiamo già costruito una sottosequenza con i simboli precedenti
                    # prendiamo il max tra:
                    # * La seq. più lunga che usa j simboli negli elem. precedenti
                    # * La seq. più lunga che usa j-1 simboli negli elem. precedenti
                        T[j][i] = max(T[j][i-1], T[j-1][i-1]) + 1 # +1 per indicare l'elem. i
    return T[2][n-1] # ritorniamo la lunghezza della sequenza massima
```

Esercizio 2

Considerate il grafo G in figura.

1. Quale è il numero minimo di archi da eliminare da G perché il grafo abbia sort topologici? **Motivare BENE la vostra risposta.**
2. Eliminate da grafo G il numero minimo di archi perché questi risulti avere sort topologici e applicate al grafo così ottenuto l'algoritmo per la ricerca del sort topologico basato sulla visita in profondità. Nell'eseguire l'algoritmo qualora risultino più nodi tra cui scegliere per proseguire la visita prendete sempre quella di indice minimo. Qual' è il sort topologico che si ottiene in questo modo?

G=



Progettazione d'algoritmi

Soluzione

Esercizio 3

Progettare un algoritmo che, dato un intero n, stampa tutte le matrici binarie $n \times n$con la proprietà che nella riga i, $0 \leq i < n$, della matrice sono presenti esattamente i uni. L'algoritmo proposto deve avere complessità $O(n^2 S(n))$ dove $S(n)$ è il numero di matrici da stampare.

Ad esempio per $n = 3$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 9 matrici:

$$\begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Soluzione

Giugno 2

mercoledì 28 maggio 2025 11:04

Esercizio 1

Abbiamo due interi k ed n con $1 \leq k \leq n$. Vogliamo sapere quanti diversi modi ci sono di partizionare l'insieme dei primi n interi in k sottoinsiemi.

Ad esempio per $k = 2$ ed $n = 3$ l'algoritmo deve restituire 3, possiamo infatti partizionare i 3 interi in due sottoinsiemi nei modi seguenti:

- $\{\{1,2\},\{3\}\}$
- $\{\{1\},\{2,3\}\}$
- $\{\{1,3\},\{2\}\}$

Progettare un algoritmo che risolve il problema in tempo $O(n \cdot k)$.

Soluzione

Esercizio 2

Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe binarie di lunghezza $2 \cdot n$ tali che il numero di uni presenti nella prima metà della stringa è lo stesso del numero di uni presenti nella seconda metà.

Ad esempio per $n = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 6 stringhe:

$1111 \quad 1010 \quad 1001 \quad 0110 \quad 0101 \quad 0000$

L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

Soluzione

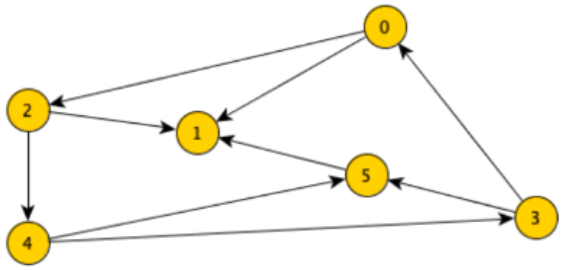
Esercizio 3

Considerate il grafo G in figura.

1. Riportate l'albero dfs che si ottiene eseguendo una visita di G a partire dal nodo 0.
2. dire quali sono gli archi di attraversamento, quali gli archi in avanti e quali gli archi all'indietro che si incontrano nel corso della visita.

Nel corso della visita ogni qualvolta un nodo ha più vicini non visitati scegliete sempre quello di indice minimo (in altre parole assumete il grafo rappresentato tramite liste di adiacenza dove i nodi di ciascuna lista sono ordinati per indice crescente).

G=



Soluzione

Giugno A

mercoledì 28 maggio 2025 11:04

Esercizio 1

ESERCIZIO 1

Il display di un telefonino si presenta come di seguito indicato:

1	2	3
4	5	6
7	8	9
*	0	#

Cerchiamo un particolare numero telefonico e sappiamo che:

- il numero è composto da n cifre.
- non contiene cifre uguali adiacenti
- nel comporre il numero sul tastierino basta spostarsi solo tra tasti adiacenti in orizzontale o verticale

Ad esempio, per $n = 7$, la combinazione 12108586996 non è di certo il numero telefonico che cerchiamo a causa della presenza delle seguenti tre coppie di cifre adiacenti 10 e 86 e 99.

Progettare un algoritmo che, dato n, restituisce il numero di combinazioni possibili per il numero telefonico da ricercare.

Ad esempio:

- per $n = 1$ la risposta dell'algoritmo deve essere 10
- per $n = 2$ la risposta dell'algoritmo deve essere 26.
  (i numeri possibili sono infatti: $08, 12, 14, 21, 23, 25, 32, 36, 41, 45, 47, 52, 54, 56, 58, 63, 65, 67, 74, 78, 80, 85, 87, 89, 96, 98$).

L'algoritmo deve avere complessità $O(n)$. **Motivare bene la correttezza e la complessità dell'algoritmo proposto**.

Soluzione

- Usiamo una tabella bidimensionale di grandezza (n+1) x 10, dove:
  T[i][k] = num. di combinazioni per il num. di telefono che finisce con k
- Il risultato sarà la somma degli elementi dell'ultima riga, cioè $\sum_{k=0}^{10} T[n-1][k]$
- Formuliamo la regola ricorsiva:
  - Se abbiamo un numero di 1 cifra, allora può essere formato solo da una delle 10 cifre della tastiera.
  - Se abbiamo n > 1, il num. di combinazioni per un num. con n cifre che finisce con la cifra k è dato dalla somma delle combina zioi per il numero con n-1 cifre che finisce con una cifra adiacente in orizzontale e verticale di k. Quindi se abbiamo una cifra che finisce con k = 2, può essere preceduta solo dalle combinazioni dei numeri con n-1 cifre che finiscono con 1, con 3 o con 5.

Esercizio 2

Abbiamo una matrice M di interi di dimensione $n \times n$ con $n > 1$. Una *discesa* su questa matrice è una sequenza di n celle della matrice con i seguenti vincoli

- le celle appartengono a righe diverse della matrice
- la prima cella appartiene alla prima riga della matrice
- ogni altra cella è adiacente (in verticale o in diagonale) alla cella che la precede.

Ad esempio, per $M = \begin{pmatrix} 12 & 10 & 3 & 14 & 9 \\ 0 & 1 & 13 & 15 & 13 \\ 8 & 10 & 1 & 2 & 7 \\ 7 & 11 & 10 & 5 & 7 \\ 18 & 4 & 6 & 10 & 0 \end{pmatrix}$

sono evidenziate due possibili discese $(12, 1, 1, 11, 4$ e $3, 15, 2, 5, 6)$.

Progettare un algoritmo che, data la matrice M, stampa tutte le possibili discese di M.

Ad esempio per $M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ l'algoritmo stamperà le 4 discese: $1, 3 \quad 1, 4 \quad 2, 3 \quad 2, 4$.

L'algoritmo deve avere complessità $O(n^2 S(n))$ dove $S(n)$ è il numero di discese da stampare.

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Soluzione

Giugno B

mercoledì 28 maggio 2025 11:04

Esercizio 1

Abbiamo una matrice M di interi di dimensione $n \times n$ con $n > 1$. Una *discesa* su questa matrice è una sequenza di n celle della matrice con i seguenti vincoli

- le celle appartengono a righe diverse della matrice
- la prima cella appartiene alla prima riga della matrice
- ogni altra cella è adiacente (in verticale o in diagonale) alla cella che la precede.

Il *valore* di una discesa è dato dalla somma dei valori delle sue n celle.

Ad esempio, per $M = \begin{pmatrix} 12 & 10 & 3 & 14 & 9 \\ 0 & 1 & 13 & 15 & 13 \\ 8 & 10 & 1 & 2 & 7 \\ 7 & 11 & 10 & 5 & 7 \\ 18 & 4 & 6 & 10 & 0 \end{pmatrix}$

sono evidenziate due possibili discese (la prima vale 29 e la seconda vale 31).

Progettare un algoritmo che, data la matrice M, trova il valore massimo tra i valori delle possibili discese di M.

Per la matrice dell'esempio precedente l'algoritmo deve restituire 66 (la discesa di valore massimo in M è infatti $14, 13, 10, 11, 18$)

L'algoritmo deve avere complessità $O(n^2)$. **Motivare bene la correttezza e la complessità dell'algoritmo proposto.**

Soluzione

- Usiamo una tabella bidimensionale di grandezza n x n dove:
  T[i][j] = valore max della discesa da una cella della prima riga alla cella (i, j)
- Il risultato sarà il valore massimo nell'ultima riga, quindi max(T[n-1][0])
- Formuliamo la regola ricorsiva:
  - Se siamo nella prima riga, poiché è quella di partenza, il valore di T[0][j] sarà quello che abbiamo in M[0][j]
  - Altrimenti dobbiamo prendere il valore massimo tra le celle da cui possiamo scendere per arrivare a T[i][j] e aggiungere il valore della cella nella matrice. Quindi prendiamo il massimo tra le celle adiacenti nella riga precedente cioè T[i-1][j-1], T[i-1][j], T[i-1][j+1] (se non escono dalla matrice) e aggiungiamo il valore M[i][j].
    Ovviamente dobbiamo controllare i casi in cui le posizioni adiacenti siano all'interno della matrice:
    - Se siamo nella prima colonna possiamo controllare solo nelle posizioni in alto o in alto a destra (quindi T[i-1][j] o T[i-1][j+1])
    - Se siamo nell'ultima colonna possiamo controllare solo nelle posizioni in alto o in alto a sinistra (quindi T[i-1][j] o T[i-1][j-1])
    - Altrimenti se siamo in una colonna diversa dalla prima o dall'ultima possiamo controllare in alto, in alto a sinistra o in alto a destra (quindi T[i-1][j], T[i-1][j-1], T[i-1][j+1])

  Quindi la regola è:

$$T[i][j] = \begin{cases} M[0][j] & se\ i = 0 \\ \mathrm{M[i][j]} + \max(T[i-1][j], T[i-1][j+1]) & se\ i > 0\ e\ j = 0 \\ \mathrm{M[i][j]} + \max(T[i-1][j], T[i-1][j-1]) & se\ i > 0\ e\ j = n-1 \\ \mathrm{M[i][j]} + \max\big(T[i-1][j], T[i-1][j-1], T[i-1][j+1]\big) & se\ i > 0\ e\ j > 0\ e\ j < n \end{cases}$$

Implementazione

Esercizio 2

ESERCIZIO 2 Il display di un telefonino si presenta come di seguito indicato:

1	2	3
4	5	6
7	8	9
*	0	#

Cerchiamo un particolare numero telefonico e sappiamo che:

- il numero è composto da n cifre.
- non contiene cifre uguali adiacenti
- nel comporre il numero sul tastierino basta spostarsi solo tra tasti adiacenti in orizzontale o verticale

Ad esempio, per $n = 7$, la combinazione 12108586996 non è di certo il numero telefonico che cerchiamo a causa della presenza delle seguenti tre coppie di cifre adiacenti 10 e 86 e 99.

Progettare un algoritmo che, dato n, enumera tutte le combinazioni possibili per il numero telefonico da ricercare.
Ad esempio:

- per $n = 1$ l'algoritmo deve stampare le 10 seguenti combinazioni $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ (non necessariamente in quest'ordine)
- per $n = 2$ l'algoritmo deve stampare le 26 seguenti combinazioni : $08, 12, 14, 21, 23, 25, 32, 36, 41, 45, 47, 52, 54, 56, 58, 63, 65, 67, 74, 78, 80, 85, 87, 89, 96, 98$ (non necessariamente in quest'ordine).

La complessità dell'algoritmo proposto deve essere $O(nS(n))$ dove $S(n)$ è il numero di combinazioni da stampare.
Motivare bene la complessità del vostro algoritmo.

Soluzione

Luglio 1

mercoledì 28 maggio 2025 11:04

Esercizio 1

ESERCIZIO 1. Progettare un algoritmo che, data una sequenza decimale S lunga n, conta il numero di sottosequenze di S strettamente crescenti. L'algoritmo proposto deve avere complessità $O(n)$.

Ad esempio per $S = [3, 2, 4, 5, 4]$ l'algoritmo deve rispondere 14, infatti S presenta le seguenti sottosequenze crescenti:

$$[3],\ [2],\ [4],\ [3,4],\ [2,4],\ [5],\ [3,5],\ [2,5],\ [4,5],\ [3,2,5],\ [3,4,5],\ [4],\ [3,4],\ [2,4]$$

Notate che possono esserci diverse sottosequenze contenenti gli stessi elementi (ad esempio la sottosequenza [4] viene contata due volte perché il 4 compare in S nella seconda e nella quarta posizione).

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

In realtà le sottosequenze sono:
[3], [2], [4], [3, 4], [2, 4], [5], [3, 5], [2, 5], [4, 5], [3, 4, 5], [2, 4, 5], [4], [3, 4], [2, 4]

Soluzione

- Utilizziamo una tabella di grandezza n dove:
  T[i] = num.
- Il risultato sarà in posizione T[n]

Esercizio 2

ESERCIZIO 2. Progettare un algoritmo che, data una sequenza decimale S lunga n, stampa le sottosequenze di S strettamente crescenti.
L'algoritmo proposto deve avere complessità $O(nD(n))$ dove $D(n)$ è il numero di sequenze da stampare.

Ad esempio per $S = [3, 2, 4, 5, 4]$ l'algoritmo deve stampare, non necessariamente in quest'ordine, le seguenti sottosequenze:

$$[3],\ [2],\ [4],\ [3,4],\ [2,4],\ [5],\ [3,5],\ [2,5],\ [4,5],\ [3,2,5],\ [3,4,5],\ [4],\ [3,4],\ [2,4]$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Soluzione

- Usiamo un algoritmo di backtracking
- Inseriamo nella soluzione:
  - L'elem. S[i] sempre se la soluzione è vuota
  - Altrimenti inseriamo S[i] solo se l'elem. precedente della soluzione in pos i-1 è minore dell'elem. S[i]

Luglio 2

mercoledì 28 maggio 2025 11:04

Esercizio 1

ESERCIZIO 1. Abbiamo diversi tipi di stringhe binarie lunghe al più 2, concatenando stringhe siffatte possiamo ottenere stringhe binarie di lunghezza arbitraria.
Progettare un algoritmo che prende in input l'insieme I coi tipi di stringhe disponibili ed una stringa binaria S lunga n e, in tempo $O(n)$, conta i diversi modi con cui e' possibile ottenere S concatentando le stringhe dei tipi disponibili.

Ad esempio per $I = \{'0', '01', '10'\}$

- se $S =' 001010'$ la risposta dell'algoritmo è 3, infatti è possibile ottenere S nei seguenti modi:
  $'0' +' 0' +' 10' +' 10'$,
  $'0' +' 01' +' 01' +' 0'$
  $'0' +' 01' +' 0' +' 10'$
- se $S =' 0011010'$ la risposte dell'algoritmo deve essere 0 infatti non c'è modo di ottenere S concatenando stringhe del tipo specificato dall' insieme I.

Soluzione

- Usiamo una tabella unidimensionale di grandezza n+1 dove:
  T[i] = modi per ottenere i primi i elem. di S concatenando le stringhe di I
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Siccome le stringhe di I sono lunghe al massimo n, allora la grandezza massima di I sarà 6 ("0", "1", "00", "01", "10", "11") quindi è possibile iterare tra gli elem. di I rimanendo con un costo costante poiché nel caso peggiore il costo sarà O(6).
  - C'è un solo modo per ottenere 0 elem. di S, cioè concatenando 0 elem. di I
  - Altrimenti il num. di modi per ottenere i primi i elem. di S (cioè S[:i+1]) è dato dalla somma dei modi per ottenere S[:i+1] senza i vari elem. di I se si possono rimuovere da S[i]. Quindi se l'elem. x di I è di lunghezza 1 (quindi è "0" o "1") allora se l'ultimo elem. di S (cioè S[i]) è uguale a x, si incrementa il num. di modi per ottenere S col num. di modi per ottenere gli elem. precedenti i-1. Quindi T[i] += T[i-1] se S[i] = x e x è lunga 1
  - Se l'elem. x di I è di lunghezza 2 (quindi è "00", "01", "10", "11") allora la somma dei modi per ottenere S[:i+1] è dato dalla somma dei modi per ottenere gli ultimi due elem. di S (quindi S[i-1:i+1]) senza i vari elem. di I se si possono rimuovere da S[i]. Quindi T[i] += T[i-2] se S[i-1:i+1] = x e x è lunga 2
  - Poiché la stringa è binaria, si possono effettuare al massimo 2 somme di T[i-1] e T[i-2] poiché se x è di lunghezza 1 allora possiamo scegliere solo 1 tra "0" o "1" e se è lunga 2 possiamo scegliere solo 1 tra "00", "01", "10", "11"

  La regola è:

$$T[i] = \begin{cases} 1 & se\ i = 0 \\ T[i] + T[i-1] & se\ len(x) = 1\ e\ S[i] = x\ con\ x \in I \\ T[i] + T[i-2] & se\ len(x) = 2\ e\ S[i-1] = x[0] e\ S[i] = x[1] con\ x \in I \end{cases}$$

Implementazione

Esercizio 2

ESERCIZIO 2: Progettare un algoritmo che, dato l'intero n, stampa tutte le sequenze lunghe n sull'alfabeto ternario $\{a, b, c\}$ dove il simbolo a è sempre seguito da almeno due simboli b.
L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di sequenze da stampare.

Ad esempio per $n = 4$ l'algoritmo deve stampare, non necessariamente in quest'ordine, le seguenti 20 sottosequenze:

$$abbb\ abbc\ babb\ bbbb\ bbbc\ bbcb\ bbcc\ bcbb\ bcbc\ bccb$$

$$bccc\ cabb\ cbbb\ cbbc\ cbcb\ cbcc\ ccbb\ ccbc\ cccb\ cccc.$$

Motivare bene la correttezza e la complessità dell'algoritmo proposto.

Soluzione

Luglio 3

mercoledì 28 maggio 2025 11:04

Esercizio 1

Data una stringa binaria S vogliamo contare il numero di diverse coppie $(0,1)$ presenti nella sequenza.

Ad esempio:

- per $S = 1110100$ la risposte deve essere 1 (abbiamo infatti $___01__$).
- per $S = 010010$ la risposta deve essere 8 (abbiamo infatti $01____$, $0___1_$, $__0_1_$, e $___01_$).

Progettare un algoritmo che risolve il problema in tempo $O(n)$.

Soluzione

- Utilizziamo una tabela unidimensionale di grandezza n, dove:
  T[i] = num. massimo di coppie (0, 1) per i primi i elem. di S
- Il risultato sarà in T[n]
- Formuliamo la regola ricorsiva:
  - Se abbiamo 1 solo elem. di S allora non possiamo avere coppie (0, 1)
  - Le diverse coppie (0,1) per l'elem. i varia dal fatto se S[i] è un 0 o un 1:
    - Se S[i] = 1 allora il num. di coppie (0, 1) per l'elem. i è dato dal num. di 0 presenti negli elem. precedenti a i. Quindi si aggiunge il num. di 0 al num. di coppie precedenti (0,1) a i-1.
      Per semplificare i calcoli, invece di contare ogni volta il num. di 0 precedenti a i, usiamo un contatore "tot" che indica i num. di 0 incontrati nella stringa prima di arrivare a i.
    - Se S[i] = 0 allora per i non si possono formare coppie (0,1), quindi portiamo avanti il num. di coppie che si possono avere con gli i-1 elem. precedenti di S e incrementiamo il contatore "tot".

Quindi la regola è:

$$T[i] = \begin{cases} 1 & se\ i = 0 \\ T[i-1] + num.di\ 0\ in\ S[:i] & altrimenti \end{cases}$$

Implementazione:

Esercizio 2

Progettare un algoritmo che, dati due interi n e k, stampa tutte le stringhe binarie di lunghezza n in cui siano presenti almeno k zeri consecutivi.

Ad esempio per $n = 4$ e $k = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 8 stringhe:
0000 0001 0010 0011 0100 1000 1001 1100

L'algoritmo proposto deve avere complessità $O(nS(n,k))$ dove $S(n,k)$ è il numero di stringhe da stampare.

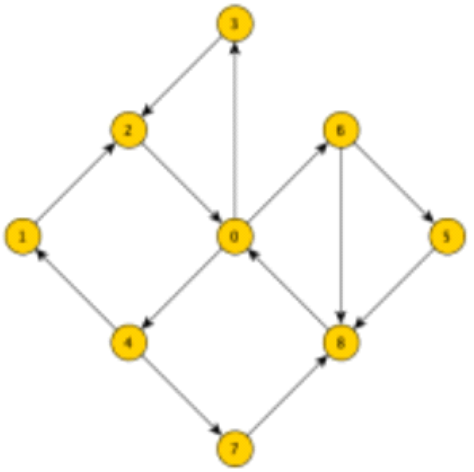
Soluzione

Esercizio 3

Considerate il grafo G in figura.

1. Riportate l'albero dfs che si ottiene eseguendo una visita di G a partire dal nodo . Nel corso della visita ogni qualvolta un nodo ha più vicini non visitati scegliete sempre quello di indice minimo (in altre parole assumete il grafo rappresentato tramite liste di adiacenza dove i nodi di ciascuna lista sono ordinati per indice crescente).
2. dire quali sono gli archi di attraversamento, quali gli archi in avanti e quali gli archi all'indietro che si incontrano nel corso della visita.

G=



Soluzione

Luglio 4

mercoledì 28 maggio 2025 11:04

Esercizio 1

Un numero intero positivo può sempre essere rappresentato come somma di quadrati di altri numeri interi. Ad esempio, il generico numero n può scomporsi come somma di n addendi tutti uguali a 1^2 (vale a dire $n = \sum_{i=1}^{n} 1^2$).
Dato un intero positivo n vogliamo scoprire qual'è il numero minimo di quadrati necessari ad ottenere n.
Ad esempio:

a. per $n = 100$ la risposta è 1 infatti $100 = 10^2$. Nota che vale anche $100 = 5^2 + 5^2 + 5^2 + 5^2$ ma questa scomposizione usa 4 quadrati e non è minimale. Analogamente non è minimale la scomposizione $100 = 8^2 + 6^2$ che usa 2 quadrati.

b. per $n = 6$ la risposta è 3 infatti $6 = 2^2 + 1^2 + 1^2$ e non è difficile vedere che 6 non può esprimersi come somma di due soli quadrati.

Soluzione

Esercizio 2

Progettare un algoritmo che dati tre interi positivi n, m e k, con $1 \leq ,m,k \leq n$, stampa tutte le stringhe ternarie lunghe n sull'alfabeto $\{0,1,2\}$ dove il numero di occorrenze di 1 non supera m ed il numero di occorrenze di 2 non supera k.
L'algoritmo proposto deve avere complessità $O(n \cdot S(n,m,k))$ dove $S(n,n,k)$ è il numero di stringhe da stampare.

Ad esempio per $n = 3$, $m = 1$ e $k = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 19 stringhe:

000, 001, 002, 010, 012, 020, 021, 022, 100, 102, 120, 122, 200, 201, 202, 210, 212, 220, 221

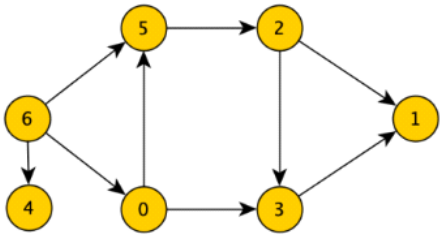
Soluzione

Esercizio 3

Considerate il grafo aciclico G in figura. Alla ricerca di un ordinamento topologico per G vogliamo applicare l'algoritmo visto a lezione basato sulla visita in profondità di G.

1. Assumiamo che, durante l'esecuzione dell'algoritmo, quando ci sono più nodi tra cui scegliere per proseguire la visita, viene sempre scelto quello di indice minimo
   Riportate l'ordinamento topologico che l'algoritmo produce.
2. Assumiamo ora che, durante l'esecuzione dell'algoritmo, quando ci sono più nodi tra cui scegliere per proseguire la visita, viene sempre scelto quello di indice massimo.
   Riportate l'ordinamento topologico che l'algoritmo produce.

G=



Soluzione

Settembre 1

mercoledì 28 maggio 2025 11:04

Esercizio 1

Dati tre interi positivi n, m e k, con $1 \leq , m, k \leq n$, vogliamo calcolare il numero di stringhe lunghe n sull'alfabeto $\{0,1,2\}$ dove il numero di occorrenze di 1 non supera m ed il numero di occorrenze di 2 non supera k.

Ad esempio per $n = 3$, $m = 1$ e $k = 2$ il numero è 19. Infatti le sole stringhe ternarie lunghe 3 con al più un 1 ed al più due 2 sono:

$000,\ 001,\ 002,\ 010,\ 012,\ 020,\ 021,\ 022,\ 100,\ 102,\ 120,\ 122,\ 200,\ 201,\ 202,\ 210,\ 212,\ 220,\ 221$

Progettare un algoritmo che risolve il problema in tempo $O(n \cdot m \cdot k)$.

Soluzione

- Usiamo una tabella tridimensionale di grandezza n x m x k, dove:
  T[i][
- Il risultato sarà in

Esercizio 2

Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe di lunghezza $2 \cdot n$ sull'alfabeto $\{1,2,3\}$ tali che il numero di cifre dispari presenti nella prima metà della stringa è lo stesso del numero di cifre dispari presenti nella seconda metà.

Ad esempio per $n = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 33 stringhe:

$1111\ \ 1113\ \ 1131\ \ 1133\ \ 1221\ \ 1223\ \ 1212\ \ 1232\ \ 1311\ \ 1313\ \ 1331$
$1333\ \ 2121\ \ 2123\ \ 2112\ \ 2132\ \ 2222\ \ 2321\ \ 2323\ \ 2312\ \ 2332\ \ 3111$
$3113\ \ 3131\ \ 3133\ \ 3221\ \ 3223\ \ 3212\ \ 3232\ \ 3311\ \ 3313\ \ 3331\ \ 3333$

L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

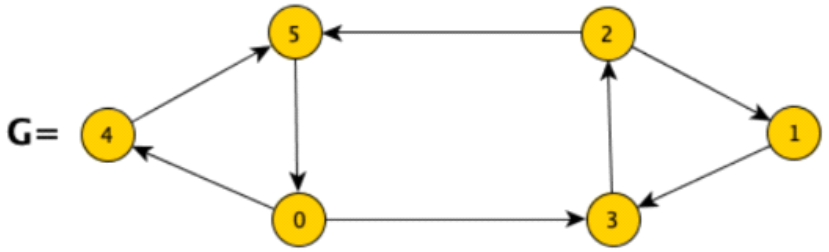
Soluzione

Esercizio 3

Considerate il grafo G in figura.

1. Quale è il numero minimo di archi da eliminare da G perché il grafo abbia sort topologici? **Motivare BENE la vostra risposta.**

3. Eliminate da grafo G il numero minimo di archi perché questi risulti avere sort topologici e applicate al grafo così ottenuto l'algoritmo delle sorgenti visto a lezione . Nell'eseguire l'algoritmo qualora risultino più sorgenti tra cui scegliere prendete sempre quella di indice minimo. Qual' è il sort topologico che si ottiene in questo modo?



Soluzione

Ottobre 1

mercoledì 28 maggio 2025 11:04

Esercizio 1

Un numero intero positivo può sempre essere rappresentato come somma di quadrati di altri numeri interi. Ad esempio, il generico numero n può scomporsi come somma di n addendi tutti uguali a 1^2 (vale a dire $n = \sum_{i=1}^{n} 1^2$).
Dato un intero positivo n vogliamo scoprire qual'è il numero minimo di quadrati necessari ad ottenere n.
Ad esempio:

a. per $n = 100$ la risposta è 1 infatti $100 = 10^2$. Nota che vale anche $100 = 5^2 + 5^2 + 5^2 + 5^2$ ma questa scomposizione usa 4 quadrati e non è minimale. Analogamente non è minimale la scomposizione $100 = 8^2 + 6^2$ che usa 2 quadrati.

b. per $n = 6$ la risposta è 3 infatti $6 = 2^2 + 1^2 + 1^2$ e non è difficile vedere che 6 non può esprimersi come somma di due soli quadrati.

Scrivere lo pseudocodice di un algoritmo che risolve il problema in tempo $O\left(n^{\frac{3}{2}}\right)$

Soluzione

- Usiamo una tabella unidimensionale di grandezza n dove:
  T[i] = num. minimo di quadrati per ottenere i
- Il risultato sarà in T[n]
- Formuliamo la formula ricorsiva:
  - Non ci sono num. minimi positivi per ottenere 0
  - L'unico num. minimo di quadrati per ottenere 1 è 1
  - Altrimenti un num i si può ottenere con quadrati da 1 a $\lfloor\sqrt{i}\rfloor$. Un num. i è dato quindi dalla somma di un quadrato j e la differenza i - j^2. Per ottenere il num. minimo di quadrati per ottenere i basta quindi prendere il num. minimo tra tutti i num. minimi di quadrati per ottenere i - j^2 con $1 \leq j \leq \lfloor\sqrt{i}\rfloor$. Infine aggiungiamo 1 per indicare che abbiamo preso il valore j^2.

Quindi la formula è:

$$T[i] = \begin{cases} 0 & se\ i = 0 \\ 1 & se\ i = 1 \\ \min\limits_{1 \leq j \leq \lfloor\sqrt{i}\rfloor} (i - j^2) + 1 & altrimenti \end{cases}$$

Implementazione

Il costo sarà: $O(n * \sqrt{n}) = O\left(n * n^{\frac{1}{2}}\right) = O\left(n^{\frac{1}{2}+1}\right) = O\left(n^{\frac{3}{2}}\right)$

Esercizio 2

Progettare un algoritmo che, dato un intero n, stampa tutte le stringhe binarie di lunghezza $2n$ tali che il numero di zeri presenti nella prima metà della stringa è inferiore o uguale al numero di uni presenti nella seconda metà.
L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

Ad esempio per $n = 2$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 11 stringhe:

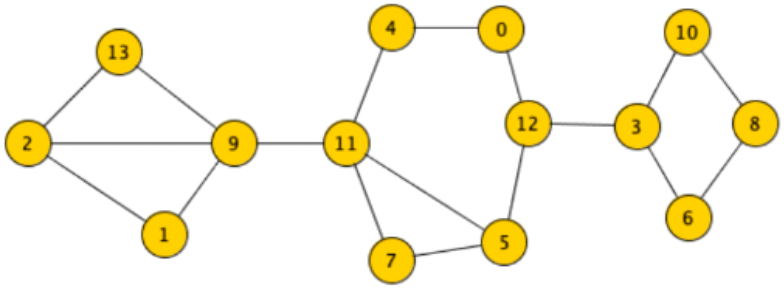
$$0011 \quad 0101 \quad 0110 \quad 0111 \quad 1001 \quad 1010 \quad 1011 \quad 1100 \quad 1101 \quad 1110 \quad 1111$$

Motivare BENE la correttezza e la complessità del vostro algoritmo.

Soluzione

Esercizio 3

G=



Considerate il grafo G in figura. Determinate il numero minimo di archi che bisogna eliminare da G perché risulti due-colorabile.
Motivare BENE la vostra risposta.

Soluzione

Straordinario 1

mercoledì 28 maggio 2025 11:04

Esercizio 1

Una pedina è posizionata sulla casella $(0,0)$ in alto a sinistra di una scacchiera $n \times n$ e mediante una sequenza di mosse tra caselle adiacenti deve raggiungere la casella $(n-1,n-1)$ in basso a destra. Una pedina posizionata sulla casella (i,j) ha al più due mosse possibili: spostarsi verso La sequenza di caselle toccate dalla pedina nello spostarsi da $(0,0)$ a $(n-1,n-1)$ determina un cammino. Ogni casella della della scacchiera è labellata con 0 o 1. Un cammino è definito lecito se non contiene caselle adiacenti con la stessa label.

Descrivere un algoritmo che data una matrice binaria M di dimensioni $n \times n$ calcola in tempo $O(n^2)$il numero di cammini leciti di$M$.

Ad esempio per la matrice $M = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}$ la risposta è 3 (i cammini leciti sono infatti: $\begin{bmatrix} 1 & 0 & 1 \\ - & - & 0 \\ - & - & 1 \end{bmatrix}$, $\begin{bmatrix} 1 & 0 & - \\ - & 1 & 0 \\ - & - & 1 \end{bmatrix}$, $\begin{bmatrix} 1 & - & - \\ 0 & 1 & 0 \\ - & - & 1 \end{bmatrix}$).

Progettare un algoritmo che risolve il problema in tempo $O(n^2)$.

Soluzione

- Usiamo una tabella bidimensionale di grandezza n x n, dove:
  T[i][j] = num. max di cammini leciti per arrivare da (0, 0) a (i, j)
- Il risultato sarà in T[n-1][n-1]
- Formuliamo la regola ricorsiva:
  - Se siamo nella prima cella (0, 0), poiché partiamo da quella stessa cella, avremmo un solo cammino
  - Gli unici modi per arrivare ad una cella (i, j) sono due, da sopra e da sinistra. Poiché un cammino lecito non può contenere caselle adiacenti uguali, la cella (i, j) se ha val x, può arrivare solo da una cella con valore (1-x) (quindi se x = 0, può arrivare dalla cella 1-0=1, se x =1, può arrivare dalla cella 1-1=0). Il num. max di cammini leciti per arrivare a (i, j) è dato dalla somma dei cammini per le caselle che possono portare a M[i][j].
    I modi per arrivare a (i, j) variano in base a dove è posizionato (i, j):
    - Se siamo nella prima riga (i=0) possiamo muoverci solo da sinistra, quindi dalla casella M[0][j-1] e porteremo avanti i cammini che portano alla casella di sinistra
    - Se siamo nella prima colonna (j=0) possiamo muoverci solo dall'alto, quindi dalla casella M[i-1][0] e porteremo avanti i cammini che portano alla casella in alto
    - Se siamo in una casella generica i > 0 e j > 0, possiamo muoverci sia da sinistra che dall'alto (quindi da M[0][j-1] e da M[i-1][0]), quindi possiamo sommare il num. dei cammini che portano alla casella a sinistra e alla casella in alto

La formula quindi è:

$$T[i][j] = \begin{cases} 1 & se\ i = 0\ e\ j = 0 \\ T[0][j-1] & se\ i = 0\ e\ j > 0\ e\ M[0][j-1] = (1 - M[0][j]) \\ T[i-1][0] & se\ i > 0\ e\ j = 0\ e\ M[i-1][0] = (1 - M[i][0]) \\ T[i][j-1] & se\ i > 0\ e\ j > 0\ e\ M[i][j-1] = (1 - M[i][j])\ e\ M[i-1][j] \neq (1 - M[i][j]) \\ T[i-1][j] & se\ i > 0\ e\ j > 0\ e\ M[i][j-1] \neq (1 - M[i][j])\ e\ M[i-1][j] = (1 - M[i][j]) \\ T[i][j-1] + T[i-1][j] & se\ i > 0\ e\ j > 0\ e\ M[i][j-1] = (1 - M[i][j])\ e\ M[i-1][j] = (1 - M[i][j]) \end{cases}$$

Implementazione

Esercizio 2

Progettare un algoritmo che data una stringa X lunga n sull'alfabeto $\{0,1,2\}$ stampa tutte le stringhe lunghe n sull'alfabeto $\{0,1,2\}$ che concordano con la stringa X in esattamente una posizione.

Ad esempio per $X = '200'$ l'algoritmo deve stampare, non necessariamente nello stesso ordine, le seguenti 12 stringhe:

110 101 102 120 010 001 002 020 211 212 221 222

L'algoritmo proposto deve avere complessità $O(nS(n))$ dove $S(n)$ è il numero di stringhe da stampare.

Soluzione

Esercizio 3

In generale un grafo pesato può avere diversi alberi di copertura di peso minimo. Dimostrare o confutare che se i pesi degli archi di sono tutti distinti allora ha un unico albero di copertura.

Soluzione

Febbraio 2025

martedì 3 giugno 2025 15:37

Esercizio 2

Data una lista A di lunghezza n contenente solo valori 0, 1 e 2 bisogna trovare il numero di sue sottosequenze in cui la somma degli elementi è pari. Ad esempio:

- per $A = [0, 1, 2, 1]$ la risposta è 7, le sottosequenze a somma pari sono : $[0, -, -, -]$ $[0, -, 2, -]$ $[-, -, 2, -]$ $[-, 1, -, 1]$ $[0, 1, -, 1]$ $[-, 1, 2, 1]$ $[0, 1, 2, 1]$
- per $A = [5]$ la risposta è 0 perché non ci sono in A sottosequenze a somma pari.

Progettare un algoritmo che, dato in input il vettore A risolva il problema in tempo $O(n)$.
Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

Soluzione

- Utilizziamo una tabella di grandezza 2 x n, dove:
  - T[0][i] = sottosequenze di somma pari con i primi i caratteri
  - T[1][i] = sottosequenze di somma dispari con i primi i caratteri
- Il risultato sarà in T[0][n-1]
- Formuliamo la regola ricorsiva
  - Se il primo carattere è pari, la somma pari sarà 1 e quella dispari 0, viceversa se il carattere è dispari
  - Se il carattere i è pari:

  Quindi la formula sarà:

$$T[0][i] = \begin{cases} 1 & se\ i = 0\ e\ A[0] \text{ è } pari \\ 0 & se\ i = 0\ e\ A[0] \text{ è } dispari \\ T[0][i-1] * 2 + 1 & se\ A[i] \text{ è } pari \\ T[0][i-1] + T[1][i-1] & altrimenti \end{cases}$$

$$T[1][i] = \begin{cases} 1 & se\ i = 0\ e\ A[0] \text{ è } dispari \\ 0 & se\ i = 0\ e\ A[0] \text{ è } pari \\ T[0][i-1] + T[1][i-1] + 1 & se\ A[i] \text{ è } dispari \\ T[0][i-1] * 2 & altrimenti \end{cases}$$

Implementazione

Esercizio 3

Dato un intero n vogliamo stampare tutte le stringhe binarie di lunghezza n in cui non compare la sottostringa '101' né la sottostringa '010'.
Ad esempio per $n = 4'$ delle 16 stringhe binarie lunghe 4 vanno stampate le seguenti 10 (non necessariamente nello stesso ordine):

0000 0001 0011 0110 0111 1000 1001 1100 1110 1111

Progettare un algoritmo, basato sulla tecnica del backtracking, che risolva il problema in tempo $O(n \cdot S(n))$ dove $S(n)$ è il numero di stringhe da stampare.
Motivare BENE la correttezza e la complessità dell'algoritmo proposto