

## Introduzione

Un algoritmo si dice efficiente se la sua complessità è di ordine polinomiale nella dimensione  $n$  dell'input  $\rightarrow O(n^c)$

di conseguenza un algoritmo è inefficiente se la sua complessità è di ordine superpolinomiale:

Esponenziale  $\Theta(c^n) = 2^{\Theta(n)}$

Super-esponenziale (cresce più veloce dell'esponenziale)

$$2^{\Theta(n^2)}, 2^{\Theta(n \log n)}$$

Sub-esponenziale (cresce più lentamente dell'esponenziale)

$$n^{\Theta(\log n)} = 2^{\Theta(\log^2 n)}, 2^{\Theta(n^c)} \quad (\text{dove } c \text{ è una costante inferiore a 1})$$

E.S. Test di primalità

l'algoritmo banale "dato un numero  $n$ , cerca un divisore tra tutti i numeri tra 2 e  $n-1$ "

è esponenziale perché ha complessità  $O(n)$  e dimensione dell'input  $\log n$  (perché per rappresentare  $n$  in binario sono necessari  $\log n$  bit es.  $n=10, \lceil \log_2 n \rceil = 4$ )

Un algoritmo efficiente, per questo problema, dovrebbe avere complessità  $O(\log^c n)$  per una

qualche costante  $c$  mentre questo algoritmo ha complessità  $O(2^{\log n})$

$\log n$  sono il numero di bit per rappresentare il  $n$  in binario

Nella ricerca dell'eventuale divisore fermarsi alla radice velocizza l'algoritmo ma non ne cambia la complessità (resta esponenziale)

$$\hookrightarrow O(\sqrt{n}) = O(2^{\log \sqrt{n}}) = O(2^{\frac{1}{2} \log n}) = 2^{\Theta(\log n)}$$

$$, X = a^{\log_a X}$$

E.S. Problema di ordinare una lista di  $n$  interi

Un algoritmo superpolinomiale è il seguente:

genera tutte le possibili permutazioni dei valori da ordinare (complessità  $O(n!)$ ) e verifica se la permutazione soddisfa l'ordinamento (complessità  $O(n)$ )

La complessità complessiva è  $O(n \cdot n!)$

la funzione fattoriale è super-esponenziale ( $n! = 2^{\Theta(n \log n)}$ ) e di conseguenza

l'algoritmo è intrattabile.  $\leftarrow$  (Se non si può avere algoritmi efficienti)

$$\bullet n! = 1 \cdot 2 \cdot 3 \cdots n > \frac{n}{2} \cdot \left(\frac{n}{2} + 1\right) \cdots n > \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{\frac{n}{2} \log \frac{n}{2}} = 2^{\Omega(n \log n)}$$

$$\bullet n! = 1 \cdot 2 \cdot 3 \cdots n < n \cdot n \cdots n = n^n = 2^{n \log n} = 2^{O(n \log n)}$$

$$\hookrightarrow b = a^{\log_a b} \Rightarrow n^n = 2^{\log_a n^n}$$

$$\log_a(b^x) = x \log_a(b) \Rightarrow 2^{\log_a n^n} = 2^{n \log n}$$

Consideriamo questo algoritmo: cerca il minimo tra gli  $n$  valori e mettilo in prima posizione, cerca il minimo tra gli  $n-1$  valori restanti e mettilo in seconda posizione, e così via...

```

def ordina(lista):
    for i in range(len(lista)): ← O(n)
        p_minimo=i
        for p in range(p_minimo+1, len(lista)): ← O(n)
            if lista[p_minimo]> lista[p]: p_minimo=p
        lista[i], lista[p_minimo] = lista[p_minimo], lista[i]
    return lista

```

```

>>> lista=[3,3,1,2,1,1,1,2,7,1]
>>> ordina(lista)
[1, 1, 1, 1, 1, 2, 2, 3, 3, 7]

```

la complessità dell'algoritmo sarà  $O(n^2)$

Quindi: Polinomiale

Se uso una struttura dati, l'heap, la cui costruzione per  $n$  elementi costa  $\Theta(n)$  e dove l'estrazione del minimo costa  $O(\log n)$  otengo un algoritmo (noto come heapsort) che ordina gli  $n$  elementi in tempo  $O(n \log n)$

```

def ordina(lista):
    heap = costruisci_heap(lista)
    lista1 = []
    for _ in range(len(lista)):
        a = estrai_minimo(heap)
        lista1.append(a)
    return lista1

```

← COSTRUZIONE HEAP  $O(n)$   
 ← ESTRAZIONE DEL MINIMO  
 $O(n \log n)$

## Complessità Algoritmo VS Complessità Problema

- Un algoritmo di complessità  $O(g(n))$  per un problema produce una limitazione superiore alla complessità del problema.
- Se si dimostra che qualunque algoritmo per quel problema ha complessità  $\Omega(f(n))$ , si è stabilita una limitazione inferiore alla complessità del problema.
- Se  $f(n)=g(n)$  allora l'algoritmo è detto ottimo, perché la sua complessità in ordine di grandezza risulta la migliore possibile.

## Due modi generali di dimostrazione per limitazioni inferiori

- Dimensione dei DATI: Se un problema ha in ingresso  $n$  DATI e richiede di esaminarli tutti allora una limitazione inferiore della complessità è  $\Omega(n)$ .
- Eventi contabili: Se un problema richiede che un certo evento sia ripetuto almeno  $m$  volte allora una limitazione inferiore della complessità è  $\Omega(m)$ .

Un problema si dice intrattabile se non può avere algoritmi efficienti:

Ese. ovvi "Stampare le stringhe binarie di lunghezza  $n$ " ← limite inf.  $2^{\Omega(n)}$   
 "Stampare tutte le permutazioni di  $n$  elementi" ← limite inf.  $2^{\Omega(n!)}$

## ESERCIZI :

- a) Data una lista di  $n$  interi vogliamo determinare se la lista ha un elemento di maggioranza assoluta (vale a dire un elemento che compare nella lista almeno  $\left\lfloor \frac{n}{2} \right\rfloor + 1$  volta)

Progettare un algoritmo efficiente (possibilmente ottimo) per il problema.

Un possibile limite inferiore potrebbe essere  $\Omega(n)$  in quanto devo GUARDARE almeno tutti gli elementi, quindi per ora



Idea 1: ordino il vettore ( $O(n \log n)$ ) e con una passata conto quante volte si ripetono gli elementi uguali, appena arrivo a contare  $\left\lfloor \frac{n}{2} \right\rfloor + 1$  m' fermo ( $O(n)$ ).

Complessità:  $O(n \log n) + O(n) = O(n \log n)$

Quindi

$$\xrightarrow{\Omega(n)} \xleftarrow{O(n \log n)}$$

Idea 2: manteniamo due contatori

Count: conta il numero di volte che l'elemento corrente compare

Index: memorizza l'indice dell'ultimo elemento che ha fatto

aumentare il contatore count

L'algoritmo scorre la lista e aggiorna i contatori come segue:

+ se l'elemento corrente è uguale all'elemento all'indice index, incrementa count

+ se l'elemento corrente è diverso dall'elemento all'indice index, decrementa count

+ se count è 0 significa che non c'è un elemento di maggioranza assoluta

def majority\_element(list): ← Algoritmo di Boyer-Moore

count = 0  
index = -1

for i in range(len(list)): ← O(n)

    if list[i] == list[index]:  
        count += 1  
    else:  
        count -= 1

    if count == 0:  
        index = i  
        count = 1

# Controlla se l'elemento all'indice 'index' è un elemento di maggioranza

count = 0  
for i in range(len(list)): ← O(n)

    if list[i] == list[index]:  
        count += 1

if count > len(list) // 2:

    return list[index]

else:

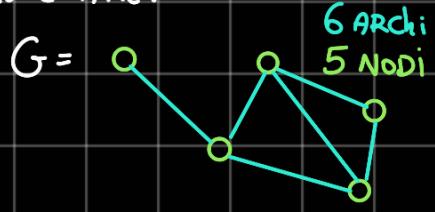
    return None

La complessità è quindi scesa a  $O(n)$ . Risulta quindi che il limite inferiore è  $\Omega(n)$  e quello superiore è  $O(n)$ . Quindi la soluzione trovata è ottima.

# I GRAFI

Sono una STRUTTURA DATI. Sono formati da Nodi e ARchi:

grafici:  $G(V, E)$   $|V| = n$ ,  $|E| = m$

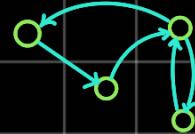


- $0 \leq m \leq \frac{n(n-1)}{2} = O(n^2)$  Se IL GRAFO è NON DIRETTO

- $0 \leq m \leq n(n-1) = O(n^2)$  Se IL GRAFO è DIRETTO

} In caso di GRAFI diretti e non, l'Asintotica NON CAMBIA

Grafo diretto: Quando abbiamo ARchi a senso unico



- Un GRAFO si dice SPARSO se  $m = O(n)$  N.B. UN GRAFO NON SPARSO NON È IN QUESTO CASO IL NUMERO di ARCHI NECESSARIAMENTE DENSO (es.  $O(n \log n)$ ) CRESCE LINEARMENTE CON IL NUMERO di NODI
- Un GRAFO si dice DENSO se  $m = \Omega(n^2)$

IN QUESTO CASO, IL NUMERO di ARCHI CRESCE QUADRATICAMENTE CON IL NUMERO di VERTICI

Ese. di GRAFI Densi:

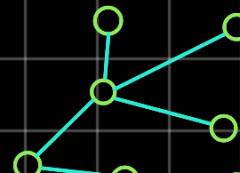
- Un GRAFO si dice COMPLETO se ha tutti gli ARCHI

- Un GRAFO diretto si dice TORNEO Se TRA OGNI COPPIA di nodi c'è ESATTAMENTE UN ARCO

Ese. di GRAFO SPARSO: L'ALBERO

Un ALBERO è UN GRAFO CONNESSO SENZA CICLI

↑  
TUTTI i nodi sono  
RAGGIUNGIBILI



(CONNESSO E SENZA)  
CICLI



(SCONNESSO E)  
con CICLO

Un ALBERO ha SEMPRE  $m = n - 1$  ARCHI

DIMOSTRAZIONE PER INDUZIONE:

CASO base ( $n=1$ ): Abbiamo 0 ARCHI, quindi VERO

PASSO INDUTTIVO ( $n > 1$ ): ASSUMIAMO PER IPOTESI CHE  $n-1$  È VERO

SFRUTTIAMO IL FATTO che gli alberi hanno le foglie. Per un albero con  $n$  nodi, scelgo una foglia e la rimuovo insieme al relativo ARCO.



L'albero ha adesso  $n-1$  nodi e di conseguenza il numero di ARCHI è diventato  $m = (n-1) - 1 = n-2$ . Questo vuol dire che l'albero originale senza l'ARCO RIMOSSO AVEVA  $m = n-2+1 = n-1$  ARCHI.

Quindi la proprietà è dimostrata anche per un albero con  $n$  nodi.

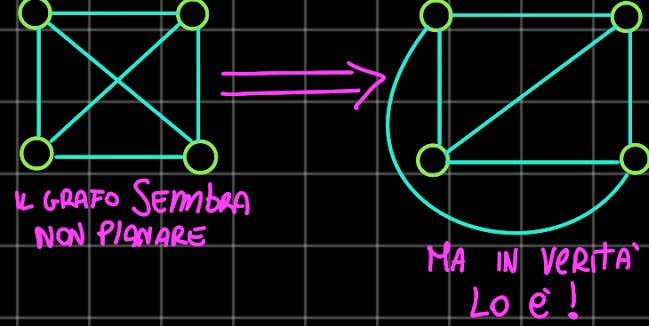
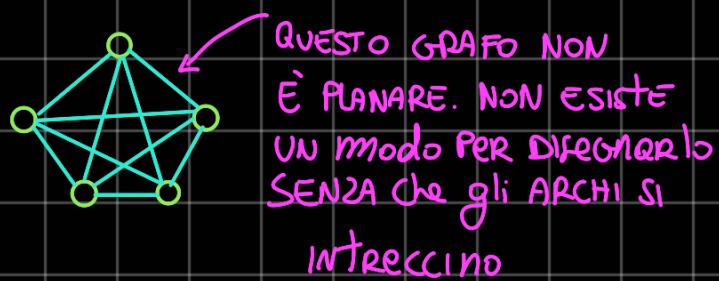
RESTA DA DIMOSTRARE che un ALBERO ha SEMPRE UN FOGGLIA:

PARTIAMO dalla RADICE e INIZIAMO ad ATTRAVERSARE i NODI.

DATO che questi nodi sono finiti, DORRO' PER FORZA RAGGIUNGERE UN nodo che NON ha archi uscenti: se non quello che e' collegato al PADRE.

QUESTO vuol dire che ho raggiunto una foglia. Se questo non accade vuol dire che c'e' un ciclo MA UN ALBERO E' ACICLICO e quindi NON SAREbbe un albero.

**GRAFO PLANARE**: I GRAFI PLANARI SONO quei GRAFI che possono disegnare sul piano senza che gli ARCHI si INTERSECHINO.



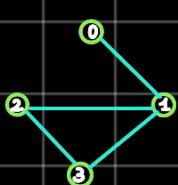
UN GRAFO PLANARE di  $n > 2$  nodi ha al più  $3n - 6$  archi

$$n \quad m = \frac{n(n-1)}{2} \text{ IN COMPLETI} \quad m = 3n - 6 \text{ MAX (IN PLANARE)}$$

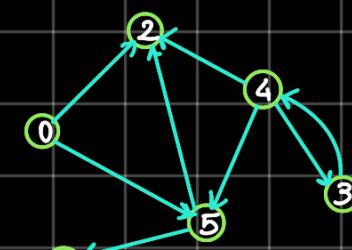
3	3	3	N.B. Dalla TABELLA
4	6	6	deduciamo che da $n \geq 5$
5	10	9	ESISTONO GRAFI NON
6	15	12	PLANARI

## RAPPRESENTARE I GRAFI CON LA MATRICE

IL modo più semplice e' USANDO UNA MATRICE



$i \setminus j$	0	1	2	3
0	0	1	0	0
1	1	0	1	1
2	0	1	0	1
3	0	1	1	0



$i \setminus j$	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
3	0	0	0	0	1	0
4	0	0	1	1	0	1
5	1	0	0	0	0	0

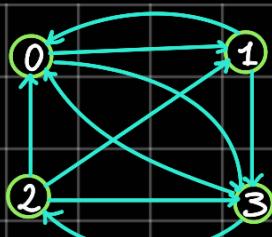
N.B. Quando il GRAFO E' INDIRETTO LA MATRICE E' SIMMETRICA

NON E' detto IL CONTRARIO

## RAPPRESENTARE I GRAFI CON LE LISTE DI ADIACENZA

UTILIZZO UNA LISTA di LISTE G, la LISTA ha TANTI elementi quanti sono i nodi del GRAFO G.  $G[x]$  e' una lista contenente i nodi adiacenti al nodo  $x$  vale a dire quelli raggiunti da archi che partono da  $x$ .

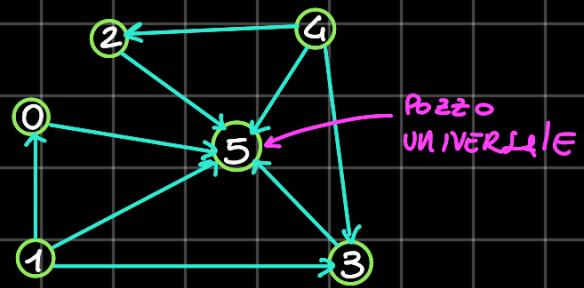
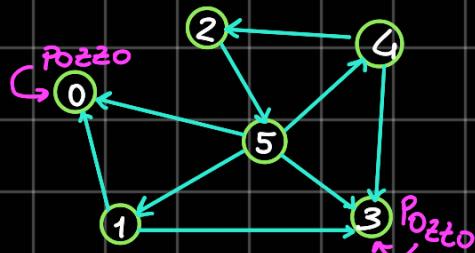
0	→	1	3	
1	→	0		
2	→	0	1	3
3	→	2	0	



N.B.

- NOTEVOLI RISPARMIO DI SPAZIO NEL CASO DI GRAFI SParsi
- VEDERE SE DUE ARCHI SONO CONNESSI PUÒ COSTARE ANCHE  $O(n)$

- IN UN GRAFO DIRETTO UN POZZO è un nodo SENZA ARCHI USCENTI
- IN UN GRAFO DIRETTO UN POZZO UNIVERSALE è UN POZZO VERSO CUI TUTTI gli altri nodi hanno un arco



N.B. IN UN GRAFO diretto POSSONO

ESISTERE FINO A n POZZI,

IL POZZO UNIVERSALE, SE C'E', E' UNICO.

$i \setminus j$	0	1	2	3	4	5
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

Pozzo  
Universale

ALGORITMO PER VERIFICARE CHE UN GRAFO DIRETTO HA UN POZZO UNIVERSALE (Tempo:  $O(n^2)$ )

```
def check_pozzo(G, r):
    for i in range(len(G)):
        if G[r][i]: return False
    for i in range(len(G)):
        if r != i and G[i][r] == 0: return False
    return True

def pozzo_universale(G):
    for e in range(len(G)):
        if check_pozzo(G, e): return True
    return False
```

controllo che la riga abbia solo ZERI  $O(n)$   
 $O(n)$

Complessità

$O(n^2)$

$\Omega(n^2)$

ALGORITMO PIÙ INTELLIGENTE PER VERIFICARE CHE UN GRAFO DIRETTO HA UN POZZO UNIVERSALE (Tempo  $\Theta(n)$ )

```

def pozzo_universale(M):
    L = [x for x in range(len(M))] ← LISTA CON TUTTI i nodi
    while len(L) > 1: ← RESTRIGE IL CAMPO DI RICERCA DI UN
        a = L.pop()
        b = L.pop()
        if M[a][b]: ← POZZO UNIVERSALE AD UN SOLO NODO (MA NON È detto che lo sia)
            L.append(b)
        else:
            L.append(a)
    x = L.pop()
    for j in range(len(M)): ← Controllo se nella riga x
        if M[x][j]: return False ← c'è almeno un 1, se si RETURN FALSE
    for i in range(len(M)): ← Controllo se nella colonna x
        if i != x and M[i][x] == 0: return False ← ci sono tutti 1, tranne nella
    return True ← cella in cui si interseca con la riga

```

Complessità  $\Theta(n)$

Se supera i DUE FOR allora è un POZZO UNIVERSALE

## Visite

Abbiamo un grafo, posso chiedermi: DATO UN NODO, quali nodi posso raggiungere?

N.B. *mentre si esegue una visita, può capitare di incontrare nodi già visitati, si potrebbe presentare un ciclo*

Dobbiamo tener traccia dei nodi già visitati;

**VETTORE CARATTERISTICO:** è un vettore binario di lunghezza pari al numero di nodi nel grafo e contenente

$\begin{cases} 1 & \text{se l'i-esimo nodo è stato visitato} \\ 0 & \text{altrimenti} \end{cases}$

## VISITE IN PROFONDITÀ - DFS (Con MATRICE)

TIPO di visita che visita i nodi in PROFONDITÀ prima di tornare indietro e cercare strade alternative. Fa uso del vettore caratteristico e sarà proprio questo che verrà restituito

```

def DFS(G,n):
    def DFS_ricorsione(G,x,visitati):
        visitati[x] = 1
        for y in range(len(G)):
            if G[x][y] == 1 and visitati[y] == 0:
                DFS_ricorsione(G,y,visitati)
        visitati = [0 for _ in range(len(G))] ← VETTORE CARATTERISTICO
    DFS_ricorsione(G,n,visitati)
    return visitati

```

LA complessità dell'algoritmo è

$$O(n) \cdot \Theta(n) = O(n^2)$$

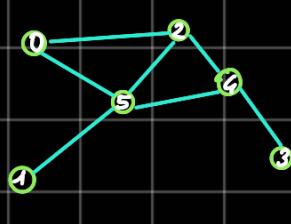
nel peggior caso tutti i nodi sono raggiungibili.  
 tempo della for che scorre tutti i nodi

## LISTE DI ADIACENZA

NEL DEI NODI  $G$ , IL NODO  $G[x]$  HA COMME ATTRIBUTO LA LISTA DEI SUOI NODI ADIACENTI

VALE A DIRE quelli raggiungibili da archi che partono da  $x$

$G = \{ 0: [2, 5]$   
 $1: [5]$   
 $2: [0, 4, 5]$   
 $3: [4]$   
 $4: [2, 3, 5]$   
 $5: [0, 1, 2, 4] \}$



# Visita IN PROFONDITÀ (DFS) (con liste di ADIACENZA)

```
def DFS(G,n):
    def DFS_ricorsione(G,x,visitati):
        visitati[x] = 1
        for y in G[x]:
            if visitati[y] == 0:
                DFS_ricorsione(G,y,visitati)
    visitati = [0 for _ in G]
    DFS_ricorsione(G,n,visitati)
    return visitati
```

Si crea un Albero con Radice il nodo n

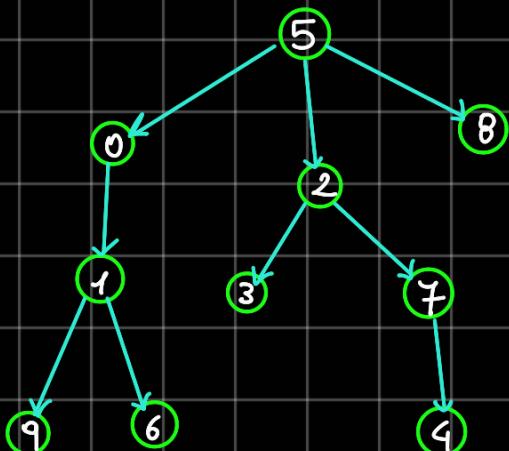
Complessità  $O(n+m)$

Con una VISTA DFS, i nodi visitati e gli archi attraversati formano un albero (Albero DFS).

Un Albero può essere memorizzato tramite i vettori padri.

Il vettore dei padri P di un albero T di n nodi ha almeno n componenti:

Se  $i$  è un nodo dell'albero  $P[i]$  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



$$P = \boxed{5 | 0 | 5 | 2 | 7 | 5 | 1 | 2 | 5 | 1}$$

Modificando lievemente l'algoritmo del DFS è possibile fare in modo che restituisca il vettore dei padri P anziché il vettore dei visitati.

```
def vettoreDeiPadri(u,G):
    def DFSr(x,G,P):
        for y in G[x]:
            if P[y] == -1:
                P[y] = x
                DFSr(y,G,P)
    P = [-1]*len(G)
    P[u] = u
    DFSr(u,G,P)
    return P
```

Possiamo anche voler un cammino che ci permetta di andare dal NODO x al NODO y

Il vettore dei padri permette di ricavare facilmente il cammino. Basta controllare se il nodo y sia nell'albero e poi da y risalire alla radice ed effettuare il reverse.

Procedura iterativa →  
per la ricerca del cammino

Disponendo già del vettore dei padri,  
La Complessità è  $O(n)$

```
def Cammino(u,P):
    if P[u] == -1: return []
    path = []
    while P[u] != u:
        path.append(u)
        u = P[u]
    path.append(u)
    path.reverse()
    return path
```

## PROCEDURA RICORSIVA PER LA RICERCA

Del cammino

Disponendo del vettore dei padri

La complessità è  $O(n)$

```
def cammino_recorsivo(u, P):
    if P[u] == -1: return []
    if P[u] == u: return [u]
    return cammino_recorsivo(P[u], P)
```

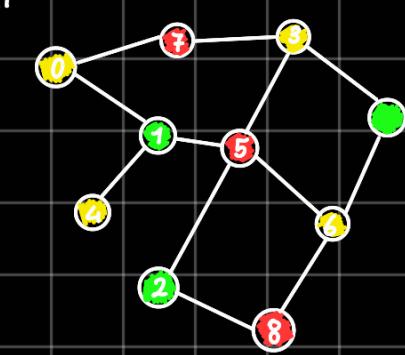
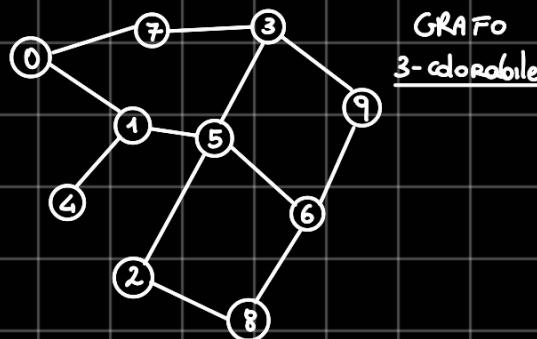
N.B. Se esistono più cammini che dal nodo  $x$

partono dal nodo  $y$  la procedura vista non garantisce di restituire il cammino minimo

## Colorazione Dei GRAFI

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.

Esempio di:

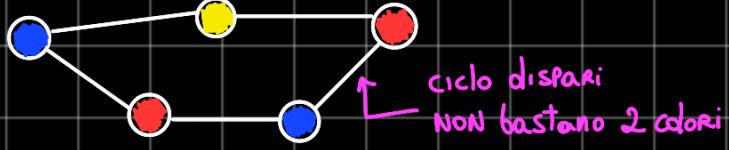


## TEOREMA DEI 4 COLORI

Un GRAFO PLANARE RICHIEDE AL PIÙ 4 COLORI PER ESSERE COLORATO

### Bi-colorazione

- UN GRAFO è 2 se e solo se NON CONTIENE CICLI DI LUNGHEZZA DISPARI



```
def colora(G):
    '''su grafi bicolorabili (i.e. connessi e senza cicli dispari)
    restituisce una bicolorazione dei nodi'''
    def DFSr(x, G, colori, c):
        colori[x] = c
        for y in G[x]:
            if colori[y] == -1:
                DFSr(y, G, colori, 1 - c)
    #####
    colori = [-1 for v in G]
    DFSr(0, G, colori, 0)
    return colori
```



Algoritmo PER LA BICOLORAZIONE  
DI UN GRAFO SENZA  
CICLI DISPARI

```
def colora1(G):
    '''restituisce una bicolorazione di G
    se il grafo è bicolorabile, una lista vuota altrimenti'''
    def DFSr(x, G, colori, c):
        colori[x] = c
        for y in G[x]:
            if colori[y] == -1:
                if not DFSr(y, G, colori, 1 - c):
                    return False
            elif colori[y] == c:
                return False
        return True
    #####
    colori = [-1 for v in G]
    if DFSr(0, G, colori, 0):
        return colori
    return []
```



PRODUCE UNA BICOLORAZIONE SE IL GRAFO È  
BICOLORABILE, PRODUCE UNA LISTA VUOTA IN CASO  
CONTRARIO

LA COMPLESSITÀ PER TESTARE SE UN GRAFO È  
BICOLORABILE È quella di una semplice visita  
del GRAFO CONNESSO DA COLORARE

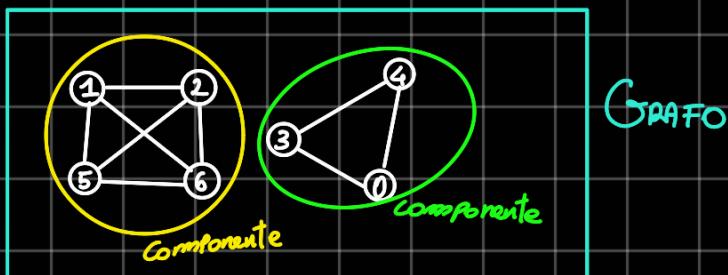
$$O(n+m) = \underbrace{O(m)}_{\text{N.B. QUESTO dipende dal FATTO che}} \quad \text{IN UN GRAFO CONNESSO}$$

$m \geq n-1$

## Componente connessa

Una componente di un GRAFO (INDIRETTO) è un sottografo composto da un insieme massimale di nodi connessi da cammini.

- UN GRAFO SI DICE CONNESSO Se ha UNA SOLO Componente



Vogliamo CALCOLARE IL VETTORE C delle componenti connesse di un GRAFO

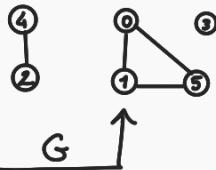
$$C = \begin{bmatrix} 2 & 1 & 1 & 2 & 2 & 1 & 1 \end{bmatrix}$$

Nodi → 0 1 2 3 4 5 6  
len = #Nodi

```
def componenti(G):
    '''restituisce il vettore delle componenti connesse del grafo G'''
def DFSr(x, G, C, c):
    C[x] = c
    for y in G[x]:
        if C[y] == 0:
            DFSr(y, G, C, c)
    #####
C = [0 for _ in G]
c = 0
for x in G:
    if C[x] == 0:
        c += 1
        DFSr(x, G, C, c)
return C

>>> G = {0:[1, 5], 1:[0, 5], 2:[4], 3:[], 4:[2], 5:[0, 1]}
>>> componenti(G)
[1, 1, 2, 3, 2, 1]
```

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



## COMPONENTE FORTEMENTE CONNESSA

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 FORTAMENTE CONNESSO SE HA UNA SOLO Componente

N.B. L'Algoritmo utilizzato per le componenti connesse NON FUNZIONA NEL CASO di componenti Fortemente connesse

- IDEA:
- 1) calcolare l'insieme A dei nodi di G raggiungibili da u Richiede  $O(n+m)$
  - 2) calcolare l'insieme B dei nodi di G che portano a u  $O(?)$
  - 3) RESTITUISCI l'INTERSEZIONE DEI DUE INSIEMI A e B  $O(n)$

PER ESEGUIRE EFFICIENTEMENTE IL PASSO 2 SI PUO' RICORRERE AL GRAFO TRASPOSTO di G ( $G^T$ )

DATO UN GRAFO DIRETTO G IL GRAFO TRASPOSTO di G ha gli stessi nodi di G ma con archi invertiti,



IL PASSO due' puo' ESSERE ESEGUITO IN TEMPO  $O(n+m)$  cercando i raggiungibili da u in  $G^T$  (ovviamente il grafo  $G^T$  andra' prima costruito)

Quindi per costruire l'algoritmo che dato il grafo  $G$  ed un suo nodo  $u$  restituisce i nodi della componente fortemente connessa che contiene  $u$

- 1) calcolare l'insieme  $A$  dei nodi di  $G$  raggiungibili da  $u$  Richiede  $O(n+m)$  (con DFS)
  - 2) calcolare il grafo TRASPOSTO di  $G$   $O(n+m)$
  - 3) calcolare l'insieme  $B$  dei nodi di  $G$  che portano a  $u$   $O(n+m)$  (con DFS)
- Restituisci l'intersezione dei DUE INSIEMI  $A$  e  $B$   $O(n)$

L'algoritmo ha complessità  $O(n+m)$

```
def componenteFC(x,G):
    visitati1=DFS(x,G)
    G1=trasposto(G)
    visitati2=DFS(x,G1)
    componente=[]
    for i in range(len(G)):
        if visitati1[i]==visitati2[i]==1:
            componente.append(i)
    return componente

def trasposto(G):
    GT=[[ ] for _ in G]
    for u in G:
        for v in G[u]:
            GT[v].append(u)
    return GT
```

DA QUESTO POSSO OTTENERE

UN ALGORITMO PER CALCOLARE IL VETTORE  $CF$  DELLE COMPONENTI  
FORTEMENTE CONNESSE

```
def compFC(G):
    FC=[0 for _ in G]
    c=0
    for x in range(len(G)):
        if FC[x]==0:
            E=componentefc(x,G)
            c+=1
            for x in E:FC[x]=c
    return FC

>>> compFC(G)
[1, 1, 2, 1, 1, 3, 1, 1, 4, 5, 4, 4]
```

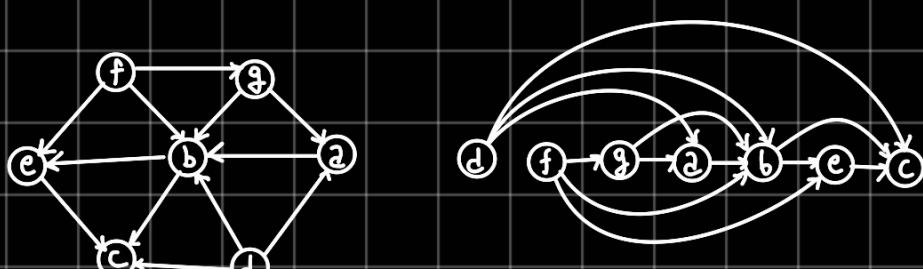
la complessità è  $\Theta(n) \cdot O(n+m) = O(n^2 + nm) = O(n^3)$

N.B. ESISTONO Algoritmi migliori che LAVORANO IN  $O(n+m)$   
ES. Algo di TARJAN  
Algo di KOSARAJU

## ORDINAMENTO TOPOLOGICO

SPESSO UN GRAFO DIRETTO CATTURA RELAZIONI DI PROPEDEVITICITÀ (UN ARCO DA  $a$  A  $b$  INDICA CHE  $a$  È PROPEDEVITICO A  $b$ )

POTRÒ RISPETTARE TUTTE le propriedà se riesco ad ordinare i nodi del grafo in modo che gli archi vadano tutti da SINISTRA verso DESTRA. Questo ordinamento è detto ordinamento topologico



UN GRAFO DIRETTO PUÒ AVERE DA 0 AD  $n!$  ORDINAMENTI TOPOLOGICI

Un algoritmo esaustivo per il problema ha complessità  $\Omega(n!)$  ed è impraticabile.

Perché  $G$  possa avere un ordinamento topologico è necessario che sia un DAG (ovvero un grafo diretto aciclico).

N.B. Non c'è solo NECESSARIO MA ANCHE SUFFICIENTE

N.B. La presenza di un ciclo implicherebbe che nessuno dei nodi del ciclo possa comparire nell'ordine giusto.

UN DAG HA SEMPRE UN NODO SORGENTE (SENZA NODO ENTRANTI)

Ora possiamo costruire l'ordinamento topologico dei nodi del DAG.

1) INIZIO la sequenza dei nodi con una sorgente

2) Cancello dal DAG quel nodo sorgente e le frecce che PARTONO da lui, ottengo un NUOVO DAG

3) ITERO questo ragionamento finché non ho sistemato in ordine lineare tutti i nodi.

```
def SortTop(G):
    '''restituisce un sort topologico ST di G se esiste
    altrimenti restituisce la lista vuota'''
    gradoEnt=[0 for _ in G]
    for u in G:
        for v in G[u]:
            gradoEnt[v]+=1
    sorgenti=[u for u in range(len(G)) if gradoEnt[u]==0]
    ST=[]
    while sorgenti:
        u=sorgenti.pop()
        ST.append(u)
        for v in G[u]:
            gradoEnt[v]-=1
            if gradoEnt[v]==0: sorgenti.append(v)
    if len(ST)==len(G): return ST
    return []
```

Complessità:

- INIZIALIZZARE IL VETTORE dei gradi entranti: COSTA  $O(n+m)$

- INIZIALIZZARE l'insieme delle sorgenti: COSTA  $O(n)$

- IL while viene iterato  $O(n)$  volte e il costo totale del for all'interno del while è  $O(m)$

Quindi il costo è  $O(n+m)$

Algoritmo alternativo basato sul DFS

+ EFFETTUÀ UNA VISITA DEL DAG A PARTIRE DAL NODO 0

+ MAN MANO che termina la visita dei vari nodi, inseriscili in una lista.

+ RESTITUISCI, COME L'ORDINAMENTO DEI NODI, IL REVERSE DELLA LISTA.

$O(n+m)$

## Cicli

Dato un grafo  $G$  (diretto o non diretto) ed un suo nodo  $u$  vogliamo sapere se da  $u$  è possibile raggiungere un ciclo in  $G$ .

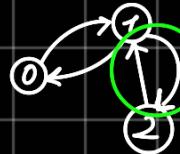
Prima idea (SBAGLIATA) è visita il grafo e se incontro durante la visita un nodo già visitato allora c'è un ciclo.

Il codice è scorretto nel caso di grafi non diretti, perché avremo il grafo in

QUESTO modo



e equivalente a



VERREBBE INTERPRETATO  
COME UN CICLO MA NON LO E'

Quindi bisogna escludere dalla ricerca il nodo precedente

```

def ciclo(u, G):
    visitati = [0] * len(G)
    return DFSr(u, u, G, visitati)

def DFSr(u, padre, G, visitati):
    """ restituisce True se nella visita da u si
    incontra nodo già visitato diverso dal padre,
    False altrimenti """
    visitati[u] = 1
    for v in G[u]:
        if visitati[v]==1:
            if v != padre:
                return True
        else:
            if DFSr(v, u, G, visitati):
                return True
    return False

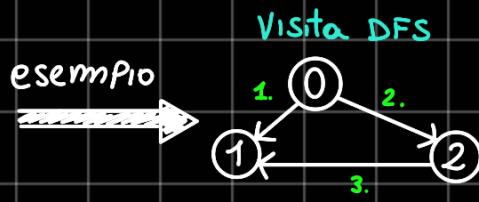
```

N.B. La complessità è  $O(n)$  ed  
è dovuta al fatto che se il grafo non  
contiene cicli allora ha al più  $n-1$  archi  
e quindi  $O(n+m) = O(n)$

MA ANCHE QUESTO CODICE È SCORRETTO NEL CASO DEI GRAFI DIRETTI:

PERCHÉ INCONTRARE UN NODO GIÀ VISITATO NON SIGNIFICA CHE È PRESENTE

UN CICLO



1. DA 0 VADO A 1

2. DA 0 VADO A 2

3. DA 2 VADO A 3, 2 già visitato MA NON  
È UN CICLO

DURANTE LA VISITA DFS POSSO INCONTRARE NODI GIÀ VISITATI IN TRE MODI:

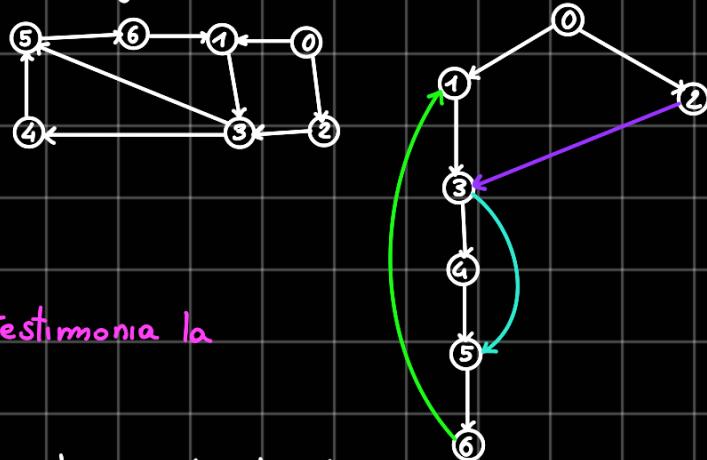
- Archi IN AVANTI
- Archi ALL'INDIETRO
- Archi di ATTRaversamento

N.B.

SOLÒ LA PRESENZA DI UN ARCO ALL'INDIETRO TESTIMONIA LA

PRESenza di un ciclo.

PER RISOLVERE IL PROBLEMA devo POTER DISTINGUERE la scoperta di nodi  
già visitati grazie ad un arco all'indietro dagli altri.



POSSO INDIVIDUARE i visitati da archi all'indietro notando che solo nel caso di archi all'indietro  
la visita del nodo già visitato ha terminato la sua ricorsione.

IDEA: PER IL VETTORE V DEI VISITATI USO 3 STEP:

+ IN V UN NODO VALE 0 SE IL NODO NON È STATO ANCORA VISITATO

+ IN V UN NODO VALE 1 SE IL NODO È STATO VISITATO MA LA RICORSIONE SU quel NODO NON È ANCORA FINITA

+ IN V UN NODO VALE 2 SE IL NODO È STATO VISITATO E LA RICORSIONE SU quel NODO È FINITA

VERSIONE CORRETTA di complessità  $O(n+m)$ :

```

def cicloD(u, G):
    visitati = [0]*len(G)
    return DFSr(u, G, visitati)

```

```

def DFSr(u, G, visitati):
    """ restituisce True se nella visita da u nel grafo
    DIRETTO G incontra un arco all'indietro (vale a dire
    diretto ad un nodo nello stato 1)."""
    visitati[u] = 1
    for v in G[u]:
        if visitati[v] == 1: return True
        if visitati[v] == 0:
            if DFSr(v, G, visitati): return True
    visitati[u] = 2
    return False

```

Se voglio SAPERE se un GRAFO (diretto o non) contiene un CICLO o meno  
devo VISITARLO TUTTO non importa il punto da cui PARTO

Di seguito la procedura modificata di grafi non diretti:

```

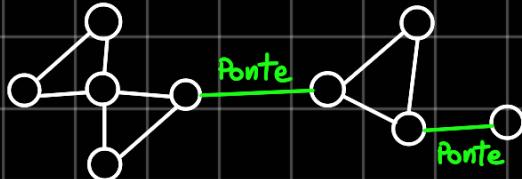
def ciclo1(G):
    ''' esegue una visita dei nodi di G e restituisce
    True se nel corso della visitasi imbatté in un ciclo'''
    V = [0] * len(G)
    for u in range(len(G)):
        if V[u]== 0:
            if DFSr(u, u, G, V): return True
    return False

def DFSr(u, padre, G, V):
    '''restituisce True se nella visita da u si
    incontra nodo già visitato diverso dal padre,
    False altrimenti'''
    V[u] = 1
    for v in G[u]:
        if V[v]==1:
            if v != padre:
                return True
        else:
            if DFSr(v, u, G, V):
                return True
    return False

```

## Ponte

IN UN GRAFO la connessione è una proprietà che può ANDAR PERSA con la perdita di un ARCO  
UN ARCO la cui eliminazione disconnette il GRAFO è detto ponte



N.B. UN GRAFO NON PUÒ AVERE  
PIÙ DI  $n-1$  PONTE

UN GRAFO PUÒ NON AVERE neanche un ponte così come avere tutti i suoi archi  
come ponte

Se voglio determinare l'insieme dei ponti del grafo

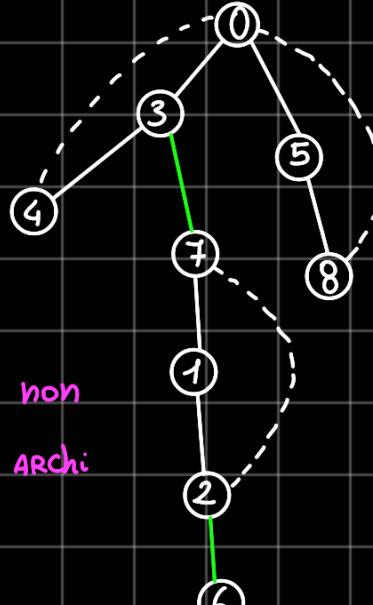
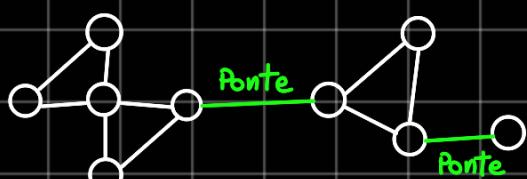
Verificare se un arco  $\{a,b\}$  è un ponte richiede tempo  $O(m)$

Basta eliminare l'arco  $\{a,b\}$  da  $G$  e con una visita DFS controllare se  $b$  è raggiungibile a partire  
da  $a$ . La complessità è  $m \cdot O(m) = O(m^2) = O(n^4)$

Vedremo che il problema è risolvibile in tempo  $O(m)$

IDEA: USARE UN'UNICA VISITA DFS modificata

UN ARCO non PRESENTE nell'ALBERO DFS non può essere ponte (se lo eliminano gli archi  
dell'albero DFS continuano a garantire la connessione)



N.B. gli archi dell'albero che non  
sono ponti risultano coperti dagli archi  
che non sono stati attraversati

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 i nodi del sottoalbero  
Radicato in  $v$  e i nodi  $u$  o nodi Antenati di  $u$ . (nelle slide del prof c'è la)  
(dimostrazione per assurdo)

IDEA: DURANTE la visita ogni nodo  $x$  calcola la sua ALTEZZA nell'albero DFS e a fine  
visita restituisce l'ALTEZZA minima raggiungibile da nodi del sottoalbero Radicato in  $x$   
tramite gli archi non presenti nell'albero DFS. Se non ci sono archi di questo tipo allora  
viene restituito  $+\infty$

Il nodo  $u$  confrontando la sua altezza con il valore che gli viene segnalato dal  
figlio  $v$  è in grado di decidere se l'arco  $\{u,v\}$  è un ponte o meno.

$\{u, v\}$  è un ponte se e solo se l'ALTEZZA di  $u$  è strettamente minore del valore restituito da  $v$

```

def TrovaPonti(G):
    '''restituisce la lista dei ponti di G'''
    Altezza = [-1]*len(G)
    Ponti = []
    DFSModificata(0, 0, G, Altezza, Ponti)
    return Ponti

def DFSModificata(x, t, G, Altezza, Ponti):
    from math import inf
    Altezza[x] = t
    ret = inf
    for y in G[x]:
        if Altezza[y] == -1:
            a = DFSModificata(y, t+1, G, Altezza, Ponti)
            if t < a:
                Ponti.append((x, y))
                ret = min(ret, a)
        elif Altezza[y] != t-1:
            ret = min(ret, Altezza[y])
    return ret

```

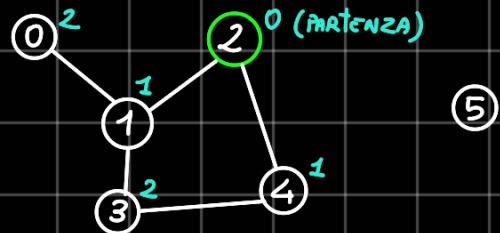
la complessità è  $O(n+m)$

Punto di articolazione: è un vertice la cui rimozione è in grado di disconnettere il grafo

Calcolo delle distanze in  $G$  e la visita in Ampiezza

Dati 2 nodi  $a$  e  $b$  di un grafo  $G$ , definiamo distanza minima di  $a$  da  $b$  il numero minimo di archi che bisogna attraversare per raggiungere  $b$  a partire da  $a$ .

N.B. la distanza è posta a  $+\infty$  se  $b$  non è raggiungibile partendo da  $a$



Vogliamo calcolare il vettore delle distanze  $D$ , dove in  $D[y]$  troviamo la distanza di  $y$  da  $x$

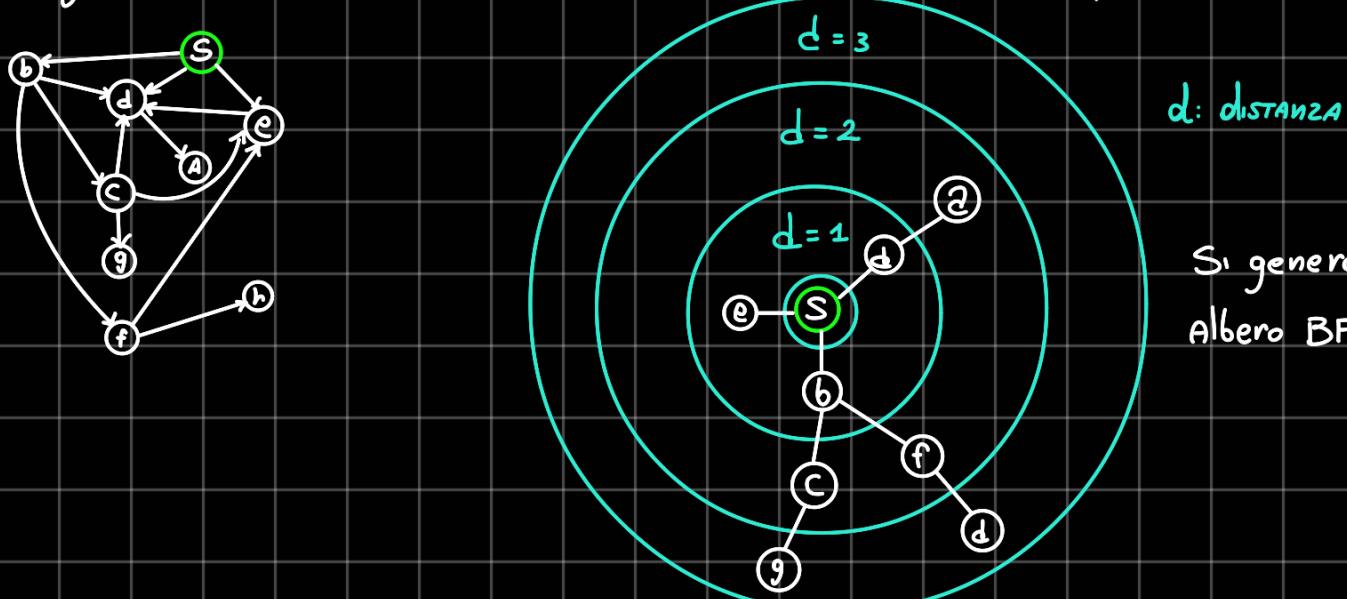
$$D = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 0 & 2 & 1 & +\infty \end{bmatrix}$$

Breadth First Search - BFS

esplora i nodi del grafo partendo da quelli a distanza 1 dalla Sorgente  $S$ .

Poi visita quelli a distanza 2 e così via.

L'algoritmo visita tutti i vertici a livello  $K$  prima di passare a quelli a livello  $K+1$



Per effettuare questo tipo di visita manteniamo in una coda i nodi visitati i cui adiacenti non sono stati ancora esaminati. ad ogni passo, preleviamo il primo nodo dalla coda, esaminiamo i suoi adiacenti e se scopriamo un nuovo nodo lo visitiamo e lo aggiungiamo alla coda.

Algoritmo che restituisce i nodi raggiungibili da  $x$  in  $G$

```
def BFS(x, G):
    '''restituisce i nodi raggiungibili da x in G'''
    visitati=[0] * len(G) ← O(n)
    visitati[x] = 1
    coda = [x]
    while coda: ← finché la coda è piena ← O(n)
        u = coda.pop(0) ← O(n)
        for y in G[u]: ← O(m)
            if visitati[y] == 0:
                visitati[y] = 1
                O(1) ← coda.append(y) #nota che y va in coda solo se non visitato
    return visitati
```

## Complessità

- Un nodo finisce in coda al più una volta quindi il while verrà eseguito  $O(n)$  volte
- le liste di adiacenza verranno scorse al più una volta quindi il costo totale dei for  $y \in G[u]$  sarà  $O(m)$
- l'inserimento in coda (append()) costa  $O(1)$
- estrazione in testa (pop(0)) ha costo  $O(n)$

Quindi il costo totale dell'algoritmo è  $O(n^2)$

## Implementazione alternativa di costo $O(n+m)$

Idea: nella coda effettuo solo cancellazioni logiche e non effettive: uso un puntatore  $i$  che indica l'inizio della coda all'interno della lista. Il puntatore si incrementa ogni volta che si cancella dalla coda

```
def BFS(x, G):
    '''restituisce i nodi raggiungibili da x in G'''
    visitati=[0] * len(G)
    visitati[x] = 1
    coda = [x]
    i = 0
    while len(coda) > i:
        u = coda[i]
        i += 1
        for y in G[u]:
            if visitati[y] == 0:
                visitati[y] = 1
                coda.append(y) #nota che y va in coda solo se non visitato
    return visitati
```

## ALTRA IMPLEMENTAZIONE CON COMPLESSITÀ $O(n+m)$ USANDO LA STRUTTURA DATI deque

Permette di effettuare inserzioni e cancellazioni in tempo  $O(1)$  es. `popleft()` e `appendleft()`

```
def BFS(x, G):
    '''restituisce i nodi raggiungibili da x in G'''
    visitati=[0] * len(G)
    visitati[x] = 1
    from collections import deque
    coda = deque([x])
    while coda:
        u = coda.popleft()
        for y in G[u]:
            if visitati[y] == 0:
                visitati[y] = 1
                coda.appendleft(y) #nota che y va in coda solo se non visitato
    return visitati
```

modifichiamo la procedura in modo che restituisca in  $O(n+m)$  l'albero di visita BFS rappresentato tramite il vettore dei padri

```
def BFSpadri(x, G):
    '''restituisce l'albero BFS radicato in x
    rappresentato col vettore dei padri'''
    P = [-1] * len(G)
    P[x] = x
    coda = [x]
    i = 0
    while len(coda) > i:
        u = coda[i]
        i += 1
        for y in G[u]:
            if P[y] == -1:
                P[y] = u
                coda.append(y) #nota che y va in coda solo se non visitato
    return P
```

modifichiamo la procedura di visita in modo che restituisce in  $O(n+m)$  il vettore delle distanze D

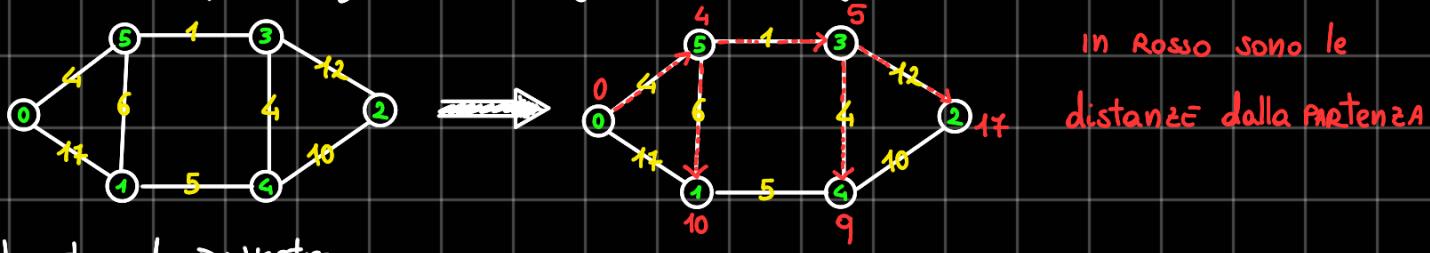
Al termine  $D[u]$  conterrà -1 se il nodo u non è raggiungibile a partire da x, la distanza minima di u da x altrimenti:

```
def BFSdistanze(x, G):
    '''restituisce il vettore delle distanze da x in G'''
    D = [-1] * len(G)
    D[x] = 0
    coda = [x]
    i = 0
    while len(coda) > i:
        u = coda[i]
        i += 1
        for y in G[u]:
            if D[y] == -1:
                D[y] = D[u] + 1
                coda.append(y) #nota che y va in coda solo se non visitato
    return D
```

Vedremo ora un algoritmo che permette di trovare i cammini minimi lavorando direttamente su grafi pesati (con pesi anche non interi)

## ALGORITMO DI DIJKSTRA

Dato un grafo pesato vogliamo trovare i cammini minimi e quindi anche le distanze da un certo nodo s (detto sorgente) a tutti gli altri nodi del grafo



Algoritmo di Dijkstra:

- 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.

N.B. Rientra nella tecnica GREEDY

- le sequenze di decisione sono irrevocabili
- le decisioni vengono prese "localmente"

N.B. L'algoritmo non è corretto in caso di grafi con pesi anche negativi

Per rappresentare questo tipo di grafi (pesati) per l'arco  $(x,y)$  di peso c nella lista di adiacenze di x invece che il solo nodo destinazione y ci sarà la coppia  $(y, c)$

```
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)]
]
```

Idea:

Un vettore binario  $\text{Inserito}[x]$  ci dice se  $x$  è già nell'albero o meno. In lista  $[x]$  c'è presente la coppia ( $distanza\ minima$ ,  $\text{Padre}$ ) il cui significato è il seguente:

- Se  $x$  inserito nell'albero: Padre è il suo nodo Padre e distanza minima la sua vera distanza dalla sorgente
- Se  $x$  non è inserito nell'albero e non è adiacente ad un nodo inserito nell'albero allora ( $distanza\ minima$ ,  $\text{Padre}$ ) =  $(\infty, -1)$
- Se  $x$  non è inserito nell'albero ma è adiacente a nodi dell'albero allora Padre è il nodo più conveniente a cui attaccare  $x$  e distanza è la distanza minima che ne risulterebbe

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

```
def dijkstra(s, G):
    '''restituisce il vettore delle distanze D e l'albero dei cammini minimi rappresentato tramite vettore dei padri P'''
    Inserito = [False] * len(G) ← O(n)
    from math import inf
    Lista = [inf, -1] * len(G) ← O(n)
    Lista[s], Inserito[s] = 0, s, True
    for y, costo in G[s] :
        Lista[y] = (costo, s) ← O(n)
    while True: ← O(n)
        minimo = inf
        for i in range(len(Lista)): ← O(n)
            if not Inserito[i] and Lista[i][0] < minimo:
                minimo, x = Lista[i][0], i
        if minimo == inf: break
        Inserito[x] = True
        for y, costo in G[x] : ← O(n)
            if not Inserito[y] and minimo + costo < Lista[y][0]:
                Lista[y] = (minimo + costo, x)
    D = [costo for costo, _ in Lista] ← O(n)
    P = [padre for _, padre in Lista] ← O(n)
    return D, P
```

Complessità totale  
 $O(n)^2$

N.B. Soluzione ottima  
nel caso di grafici densi  
dove  $m = \Theta(n^2)$

Se si potesse evitare di scorrere ogni volta il vettore  $D$  alla ricerca della posizione in cui c'è presente il minimo, potrei evitare di pagare  $O(n)$  ad ogni iterazione del while

↳ Idea

Un nodo  $x$  di  $G$  per cui c'è un arco  $(p, x)$  con  $p$  nell'albero e  $x$  non nell'albero. Posso mantenere in un heap minimo la tripla  $(D[p] + \text{costo}(p, x), x, p)$  in questo modo ad ogni step possiamo scoprire in tempo logaritmico nella dimensione dell'heap il nodo  $x$  da inserire nell'albero, il costo  $D[p] + \text{costo}(p, x)$  da assegnargli e il padre  $p$  a cui aggiungerlo.

In python abbiamo un modulo `heapq` in cui troviamo le funzioni per creare, inserire un elemento nell'heap, estrarre un elemento dall'heap ecc. ecc., a noi bastano queste due funzioni:

- `heappop(h)`: estrae e restituisce il minimo dall'heap  $h$  in tempo  $O(\log(h))$
- `heappush(h, x)`: inserisce l'elemento  $x$  nell'heap  $h$  in tempo  $O(\log(h))$

```
def dijkstral(s, G):
    '''restituisce il vettore delle distanze ed l'albero dei cammini minimi rappresentato tramite vettore dei padri'''
    P = [-1] * len(G) ← O(n)
    from math import inf
    D = [inf] * len(G) ← O(n)
    D[s], P[s] = 0, s
    from heapq import heappop, heappush
    H = []
    for y, costo in G[s] : ← O(n)
        heappush(H, (costo, s, y)) ← O(log n) } O(n log n)
    while H:
        costo, padre, x = heappop(H) ← O(log n)
        if P[x] == -1:
            P[x] = padre
            D[x] = costo
            for y, costo1 in G[x]:
                heappush(H, (D[x] + costo1, x, y)) ← O(log n)
    return D, P
```

Complessità  
 $O(n \log n) + O(m \log n) + O(m \log n) =$

$O((n+m) \log n)$

N.B. Questa implementazione è  
migliore per i grafici sparsi

UNION-FIND è una struttura dati per gestire insiemi disgiunti.

Le 3 operazioni fondamentali sono:

1. CREA(S): Restituisce una struttura dati UNION-FIND sull'insieme S di elementi: dove ciascun elemento è in un insieme separato
2. FIND(x,C): Restituisce il nome dell'insieme della struttura dati C a cui appartiene l'elemento x
3. UNION(A,B,C): modifica la struttura dati C fondendo i due suoi insiemi, A e B, in un singolo insieme

N.B. Una gestione efficiente di insiemi disgiunti è utile in diversi contesti:

es. un grafo evolve nel tempo con l'aggiunta di archi. In questo caso gli insiemi disgiunti sono le componenti connesse del grafo

Operazione find (μ) permette di scoprire in quale componente si trova il nodo u

L'operazione Union(A,B) può essere usata per fondere due componenti:

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

$\Theta(n)$

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

$\Theta(1)$

```
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
```

$\Theta(n)$

Possiamo bilanciare i costi

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

$\Theta(n)$

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

$\Theta(1)$

$\leftarrow$  Possiamo fare di meglio

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

$\Theta(n)$

$\Theta(\log n)$

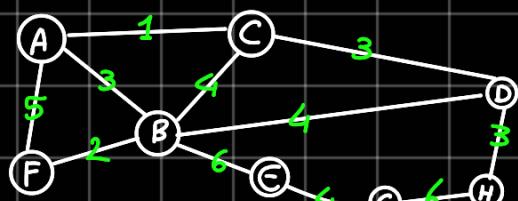
$\Theta(1)$

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

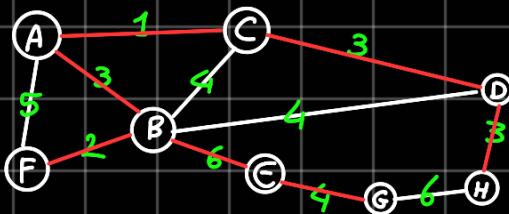
def Find(u,C):
    while u != C[u]:
        u = C[u]
    return u

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[b]=(b, tota + totb)
        C[a]=(a, totb)
```

# SPANNING TREE



Vogliamo minimizzare il costo totale e garantire la connettività tra tutti i nodi



N.B. nel grafo Soluzione (con gli archi rossi) NON sono mai presenti cicli, l'eliminazione di un qualunque arco del ciclo non farebbe perdere la connessione e diminuire

il sottoinsieme degli archi del grafo che formano la soluzione è dunque un

Albero (grafo connesso aciclico). Andiamo alla RICERCA IN G di un albero che copre l'intero grafo e la somma dei costi dei suoi archi sia minima

## MINIMUM SPANNING TREE

Problema: DATO UN grafo G connesso e pesato cerchiamo un suo MINIMUM SPANNING TREE.

### ALGORITMO di KRUSKAL

- Parti con il grafo T che contiene tutti i nodi di G e nessun arco di G
- Considera uno dopo l'altro gli archi del grafo G in ordine di costo crescente
- Se l'arco forma ciclo in T con archi già presi allora non prendendo altrimenti inseriscilo in T
- Al termine restituisce T

```
def kruskal(G):
    E = [(c, u, v) for u in range(len(G)) for v, c in G[u] if u < v] ← O(n+m)
    E.sort() ← O(m log m)
    T = [] for _ in G] ← O(n)
    while E: ← O(m)
        c, x, y = E.pop()
        if not connessi(x, y, T): ← O(n)
            T[x].append(y)
            T[y].append(x)
    return T

def connessi(x, y, T):
    """ esegue una visita nella grafo T a partire da x e restituisce True se nel corso della visita raggiunge y, False altrimenti """
    def connessiR(a, b, T):
        visitati[a] = 1 ← Te un grafo aciclico, al meglio visitato
        for z in T[a]:
            if z == b: return True
            if not visitati[z] and connessiR(z, b, T): return True
        return False
    visitati = [0] * len(T)
    return connessiR(x, y, T)
```

Complessità Totale

$$O(n \cdot m)$$

N.B.

E = [(c, u, v) for u in range(len(G)) for v, c in G[u] if u < v]	ITERA tutti i nodi
DA cosa	IF nodo accedo alla lista
è formata	CREA una sequenza da 0 a n-1
la lista	DEI nodi adiacenti
Costo	Itero ogni arco adiacente
Nodo X	U < V evite di considerare
Nodo Y	DUE volte archi simmetrici;

POSSIAMO MIGLIORARE l'algoritmo SFRUTTANDO la struttura dati Union-Find

```
def kruskal1(G):
    E = [(c, u, v) for u in range(len(G)) for v, c in G[u] if u < v] ← O(n+m)
    E.sort() ← O(m log n)
    T = [ [] for _ in G] ← O(n)
    C = crea(T) ← O(n)
    while E: ← O(m)
        c, x, y = E.pop()
        cx = find(x, C) ← O(log n)
        cy = find(y, C) ← O(log n)
        if cx != cy:
            T[x].append(y)
            T[y].append(x)
            union(cx, cy, C) ← O(1)
    return T
```

$$O(n+m)$$

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

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

```
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[b]=(b, tota + totb)
        C[a]=(a, totb)
```

Complessità Totale  
 $O(m \log n)$

# Algoritmo greedy

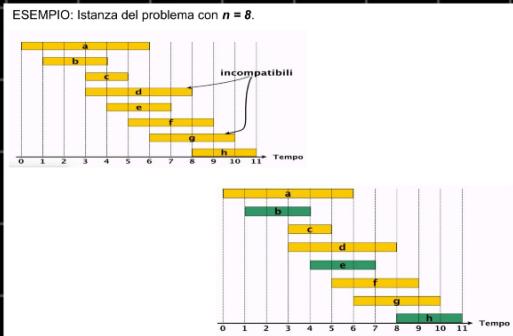
consideriamo un problema chiamato "Selezione di attività"

- abbiamo una lista di  $n$  attività:

- + ciascuna attività è caratterizzata da una coppia con il suo tempo di inizio e di fine

- + due attività sono compatibili se non si sovrappongono.

Vogliamo trovare un sottoinsieme di attività compatibili di massima cardinalità.



l'insieme da selezionare è  $\{b, e, h\}$

Volendo utilizzare il paradigma greedy dobbiamo trovare una regola, semplice da calcolare, che ci permetta di fare la scelta giusta.

diverse potenziali Regole:

- + prendi l'attività compatibile che inizia prima

- + " " " che dura meno

- + " " " che ha meno conflitti con le rimanenti

La regola giusta (per questo problema) sta nel prendere sempre l'attività compatibile che finisce prima

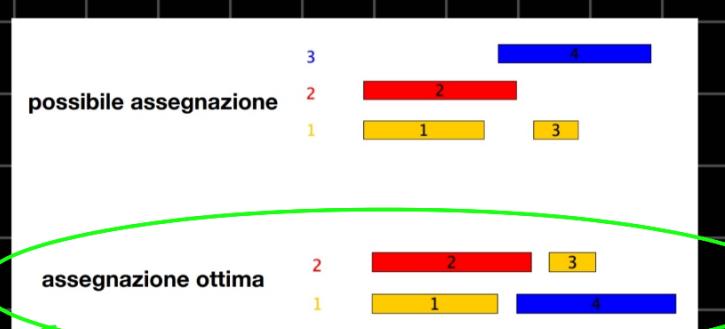
```
def selezione_a(lista):
    lista.sort( key = lambda x: x[1] ) ← ordinamento per tempo di fine crescente } O(n log n)
    libero = 0
    sol = []
    for inizio, fine in lista: ← Θ(n)
        if libero <= inizio:
            sol.append((inizio, fine)) } Θ(1)
            libero = fine
    return sol
```

Complessità Totale

$\Theta(n \log n)$

consideriamo un nuovo problema noto come "assegnazione di attività"

le attività vanno tutte eseguite e vogliamo assegnarle ad un minor numero di aule tenendo conto che in una stessa aula non possono eseguirsi più attività in parallelo



```
def assegnazione_a(list):
    inizializza la soluzione con una aula senza attività
    while lista :
        estrai da lista l'attività (a,b) che inizia prima
        if c'e' nella soluzione un' aula in cui e' possibile eseguirla:
            assegna (a,b) a quell' aula
        else :
            inserisci nella soluzione una nuova aula ed assegna gli (a,b)
    return la soluzione con le aule e le attività assegnate
```

```
def assegnazioneAule(list):
    from heapq import heappop, heappush
    Sol=[[ ]]
    H=[(0,0)] ← Θ(n log n)
    lista.sort() ← n volte
    for inizio,fine in lista:
        libera, aula = H[0]
        if libera <= inizio:
            Sol[aula].append( (inizio, fine) )
            heappop( H )
            heappush( H, (fine, aula) )
        else:
            Sol.append([ (inizio, fine) ])
            heappush( H, (fine, len(Sol)-1) )
    return Sol
```

Complessità totale:  $\Theta(n \log n)$

**ESERCIZIO:** Abbiamo  $n$  file di dimensioni  $d_0, d_1, \dots, d_{n-1}$  che vorremmo 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.

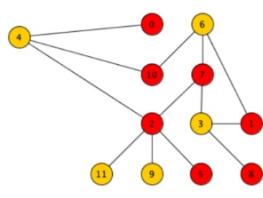
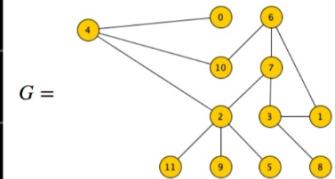
Descrivere un algoritmo *greedy* che risolve il problema in tempo  $\Theta(n \log n)$  e provarne la correttezza.

```
def file(D, k):
    n = len(D)
    lista = [(D[i], i) for i in range(n)]  $\leftarrow \Theta(n)$ 
    lista.sort()  $\leftarrow \Theta(n \log n)$ 
    spazio, sol = 0, []
    for d, i in lista:  $\leftarrow \Theta(n)$ 
        if spazio + d <= k:
            sol.append(i)
            spazio += d
        else:
            break
    return sol
```

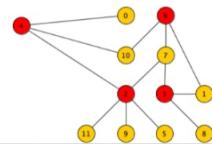
Complessità totale  
 $\Theta(n \log n)$

## Algoritmi di APPROXIMAZIONE

Dato un grafo non diretto  $G$  una sua copertura tramite nodi è un sottoinsieme  $S$  dei suoi nodi tale che tutti gli archi di  $G$  hanno almeno un estremo in  $S$ .



Il problema della copertura tramite nodi:  
dato un grafo non diretto  $G$  trovare una copertura tramite nodi di minima cardinalità.



CORSO DI PROGETTAZIONE DI ALGORITMI - Prof. Angelo Monti

Una STRATEGIA greedy è :

finché ci sono archi non coperti, inserisci in  $S$  il nodo che copre il massimo numero

di archi da coprire.

Questo Algo però non sarebbe corretto.

Contro esempio:

già al primo passo prenderebbe E e sbaglierebbe

In questi casi potrebbe essere già soddisfacente ottenere una soluzione che sia soltanto "vicina" ad una soluzione ottimale.

Fra gli algoritmi che non trovano sempre una soluzione ottima, è importante distinguere 2 categorie piuttosto differenti: **Algoritmi di APPROXIMAZIONE e EURISTICHE**

- **APPROXIMAZIONE:** Sono Algoritmi per cui si dimostra che la soluzione prodotta ha una certa "vicinanza" alla soluzione ottima. In altre parole, è garantito che la soluzione prodotta APPROSSIMA entro un certo grado una Soluzione ottima.

- **EURISTICHE:** Sono Algoritmi per cui non si riesce a dimostrare che la soluzione prodotta ha sempre una certa vicinanza ad una soluzione ottima.

Un Algoritmo di APPROXIMAZIONE per detto PROBLEMA è un Algoritmo per cui si dimostra che la soluzione prodotta APPROXIMA SEMPRE entro un certo grado una soluzione ottima.

INIZIAMO con problemi di minimizzazione, dove ad ogni soluzione ammessa è associato un costo e cerchiamo quindi la soluzione ammessa di costo minimo.

Sia  $P$  un qualsiasi problema di minimizzazione ed  $I$  una sua ISTANZA.

INDICHIAMO con  $OPT(I)$  il costo di una soluzione ottima per l'ISTANZA e con  $A(I)$  la soluzione prodotta dall'algo  $A$  per quell'ISTANZA

Si dice che A approssima P entro un fattore di approssimazione  $\rho$  se t'ISTANZA I di P

vale  $\frac{A(I)}{\text{OTT}(I)} \leq \rho$  ovvero il rapporto al caso pessimo tra il costo della soluzione prodotta dell'algoritmo e il costo della soluzione ottima.

N.B. Risulta sempre  $A(I) \geq \text{OTT}(I)$  di conseguenza il rapporto di approssimazione è sempre  $\geq 1$

- Se A approssima P con fattore 1 allora A è corretto per P perché trova sempre una soluzione ottima.

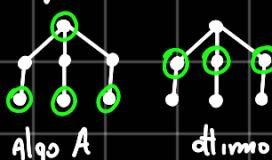
- Se A approssima P entro un fattore 2 allora A trova sempre una soluzione di costo al più doppio di quello della soluzione ottima.

Per problemi di massimizzazione dove ad ogni soluzione ammessa è associato un valore si considera il rapporto inverso, vale a dire  $\frac{\text{OTT}(I)}{A(I)}$

valuteremo l'Algo Greedy per il problema Ricoprimento tramite Nod: [Vedi sopra]

- L'algo greedy non è corretto
- Abbiamo trovato un'ISTANZA per cui l'algo produce una soluzione con 5 nodi mentre la soluzione ottima ha 4 nodi.

Da ciò Possiamo dedurre che il fattore di approssimazione dell'Algo è almeno  $\frac{5}{4} > 1$

Ma potrebbe essere peggiore →   $\frac{5}{3} > 1$

Quindi l'algoritmo di greedy in esame non garantisce nessun fattore di approssimazione Costante

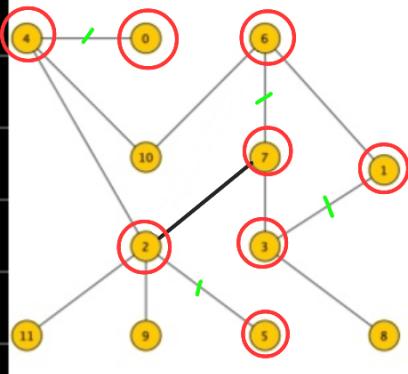
Consideriamo il seguente algoritmo greedy per la copertura di nodi:

considera i vari archi del grafo uno dopo l'altro e ogni volta che ne trovi uno non coperto (vale a dire nessuno dei suoi estremi è in S) aggiungi entrambi gli estremi dell'arco alla soluzione S.

```
def copertura(G):
    inizializza la lista E con gli archi di G
    S = set()
    while E != []:
        estrai da E un arco (x,y)
        if ne' x ne' y sono in S:
            S.add(x)
            S.add(y)
    return S
```

```
def copertura(G):
    n = len(G)
    E = [(x,y) for x in range(n) for y in G[x] if x < y]
    presi = [0] * n
    Sol = []
    for a,b in E:
        if presi[a] == presi[b] == 0:
            Sol.append(a)
            Sol.append(b)
            presi[a] = presi[b] = 1
    return Sol
```

Complessità:  $O(n+m)$



- L'algoritmo produce ovviamente una copertura
- La copertura prodotta non è detta sia minima. L'algoritmo ha rapporto d'approssimazione almeno 2 (come dimostra il grafo con due soli nodi ed un arco)
- Il rapporto d'approssimazione dell'algoritmo greedy è limitato da 2

1) Per come funziona l'algoritmo deduciamo che  $A(I) = 2K$  ( $K$  sono gli archi selezionati e 2 perché ogni arco ha 2 nodi)

2) deduciamo che  $K \leq \text{OTT}(I)$  perché almeno un estremo di ogni arco deve essere presente.

Ricaviamo che  $A(I) = 2K \leq 2 \cdot \text{OTT}(I)$

$$\text{da cui segue } \frac{A(I)}{\text{OTT}(I)} \leq 2$$

# TENICA DIVIDE ET IMPERA

## Il problema della selezione

Data una lista  $A$  di interi distinti, il *rango* di un elemento in  $A$  è il numero di elementi che sono minori o uguali ad  $x$  in  $A$ .

Esempi:

*minimo* in  $A$  è l'elemento di rango 1.  
*mediano* in  $A$  è l'elemento di rango  $\lceil \frac{n}{2} \rceil$   
*massimo* in  $A$  è l'elemento di rango  $n$ .

**Problema:** Abbiamo una lista  $A$  di  $n$  numeri distinti ed un intero  $k$ ,  $1 \leq k \leq n$ . Vogliamo l'elemento di rango  $k$  in  $A$ .

Un semplice algoritmo di complessità  $\Theta(n \log n)$  è il seguente:

```
def selezione1(A, k):
    A.sort()
    return A[k-1]

>>> A = [30, 20, 50, 60, 10, 40, 90, 80, 70]
>>> selezione1(A, 4)
40
```

Se  $K=1$  o  $K=n$  il problema si riduce alla ricerca del minimo o del massimo e questi casi sono risolvibili in tempo  $\Theta(n)$

utilizzando la tecnica del divide et impera vedremo che anche nel caso generale il problema può

Risolversi in  $\Theta(n)$

Approccio basato sul Divide et impera:

- scegli nella lista  $A$  un elemento  $x$  (che chiameremo perno).
- a partire da  $A$  costruisci due liste  $A_1$  ed  $A_2$ , la prima contenente gli elementi di  $A$  minori di  $x$  e la seconda gli elementi di  $A$  maggiori di  $x$ .
- dove si trova l'elemento di rango  $k$ ?
  - se  $|A_1| \geq k$  allora l'elemento di rango  $k$  è nel vettore  $A_1$
  - se  $|A_1| = k - 1$  allora l'elemento di rango  $k$  è proprio il perno  $x$ .
  - se  $|A_1| < k - 1$  allora l'elemento di rango  $k$  è in  $A_2$  è l'elemento di rango  $k - |A_1| - 1$  in  $A_2$

```
from random import randint

def selezione2R(A,k):
    '''restituisce l'elemento di rango k dove 1 <= k <= len(A)'''
    if len(A) == 1: return A[0]
    perno = A[randint(0, len(A)-1)]
    A1, A2 = [], []
    for x in A:
        if x < perno:
            A1.append(x)
        elif x > perno:
            A2.append(x)
    if len(A1) >= k:
        return selezione2R(A1, k)
    elif len(A1) == k-1:
        return perno
    return selezione2R(A2, k - len(A1) - 1)
```

## Complessità totale

In generale la complessità è catturata dalla ricorrenza  $T(n) = T(m) + \Theta(n)$

finché  $m$  è una frazione di  $n$  la ricorrenza dà sempre  $T(n) = \Theta(n)$

Se la scelta del perno avviene in modo equiprobabile a caso tra i vari elementi di  $A$ , il tempo di calcolo risulta con alta probabilità lineare in  $n$

N.B. Se il perno scelto risulta sempre vicino al massimo/minimo della lista, la complessità rimane  $O(n^2)$

Sappiamo che riuscire a selezionare un perno in grado di garantire che nessuna delle due sottoliste  $A_1$  e  $A_2$  abbia più di  $c n$  elementi per una qualche costante  $0 < c < 1$ , avrebbe come conseguenza una complessità di calcolo  $O(n)$

metodo **IL MEDIANO DEI MEDIANI** per selezionare un perno che garantisce di produrre sempre due sottoliste  $A_1$  ed  $A_2$  ciascuna delle quali ha al più  $\frac{3}{4}n$  elementi :

- Dividi gli  $n$  elementi di  $A$  in gruppi di 5 elementi ciascuno tranne al più l'ultimo che potrebbe averne di meno e considera i primi  $\lfloor \frac{n}{5} \rfloor$  gruppi ottenuti (tutti di 5 elementi ciascuno)
- Trova il mediano in ciascuno di questi  $\lfloor \frac{n}{5} \rfloor$  gruppi.
- Trova il mediano  $p$  dei  $\lfloor \frac{n}{5} \rfloor$  mediani.
- Usa  $p$  come perno di  $A$ .

ESEMPIO:

$A = [15, 2, 10, 16, 21, 12, 1, 9, 11, 17, 22, 3, 8, 13, 14, 4, 9, 19, 5, 6, 20, 23, 18, 7]$

$15, 2, 10, 16, 21 | 12, 1, 9, 11, 17 | 22, 3, 8, 13, 14 | 4, 19, 5, 6, 20 | 23, 18, 7$

$2, 10, 15, 16, 21 | 1, 9, 11, 12, 17 | 3, 8, 13, 14, 22 | 4, 5, 6, 19, 20 |$

$15, 11, 13, 6$

$6, 11, 13, 15$

$p=11$

**Proprietà:** se la lista  $A$  contiene almeno 120 elementi e il perno con cui partizionarla viene scelto in base alla regola appena descritta si può esser sicuri che la dimensione di ciascuna delle due sottoliste  $A_1$  e  $A_2$  ottenute sarà limitata da  $\frac{3}{4}n$   
[Dimostrazioni slide prof. Monti]

```

from math import ceil

def selezione(A, k):
    ''' restituisce l'elemento di rango k dove 1<=k<=len(A)
        scegliendo ogni volta il perno con la regola del mediano dei mediani'''
    if len(A) <= 120:
        A.sort()
        return A[k-1]
    # inizializza B con i mediani dei len(A)//5 gruppetti di 5 elementi di A
    B = [sorted(A[5*i : 5*i+5])[2] for i in range(len(A)//5)]
    #individua il perno p con la regola del mediano dei mediani
    perno = selezione(B, ceil(len(A)/10))
    A1, A2 = [], []
    for x in A:
        if x < perno: A1.append(x)
        elif x > perno: A2.append(x)
    if len(A1) >= k:
        return selezione(A1, k)
    elif len(A1) == k-1:
        return perno
    return selezione(A2, k - len(A1) - 1)

```

per  $n \geq 120$  risulta  $|listas| \leq \frac{3}{4}n$

e  $|listas| \leq \frac{3}{4}n$ .

Dunque per la complessità  $T(n)$  si ha

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

Notiamo che la ricorrenza è del tipo

$$T(n) = T(\alpha \cdot n) + T(\beta \cdot n) + \Theta(n)$$

$$\text{con } \alpha + \beta = \frac{1}{5} + \frac{3}{4} = \frac{19}{20} < 1$$

Questa tipo di soluzione ha complessità

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

[dimostr. slide prof. monti]

UN PROGRAMMA SVILUPPATO SECONDO LA TECNICA DEL DIVIDE ET IMPERA E' DIVISO IN 3 PARTI:

**Divide:** si procede alla suddivisione dei problemi in problemi di dimensione minore

**Impera:** i problemi vengono risolti in modo ricorsivo.

Quando i sottoproblemi arrivano ad avere una dimensione sufficientemente

piccola, essi vengono risolti direttamente il caso base

**Combina:** prevede di ricombinare l'output ottenuto dalle precedenti chiamate ricorsive

#### ESERCIZIO:

Dati due interi  $a$  ed  $n$  progettare un algoritmo che calcoli  $a^n$  in tempo  $\Theta(\log n)$  (le uniche operazioni aritmetiche permesse sono  $+$ ,  $*$  e  $//$ ).

Un semplice algoritmo che calcola  $a^n$  utilizzando  $n-1$  prodotti è il seguente:

```

def pow(a, n):
    if n == 0:
        return 1
    x = pow(a, n//2)
    if n % 2:
        return x*x*a
    return x*x

```

la ricorrenza da studiare per la complessità dell'algoritmo (e per il numero di prodotti richiesti) è

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

che risolta dà  $\Theta(\log n)$

#### ESERCIZIO:

Data una stringa binaria  $S$  di  $n$  bit progettare un algoritmo che calcoli il numero di sottostringhe di  $S$  che cominciano con 0 e terminano con 1.

#### IMPLEMENTAZIONE:

```

def es(S, i, j):
    ''' conta le sottostringhe di S che cominciano con 0 e terminano con 1 '''
    if i == j:
        if S[i] == '0': return (1, 0, 0)
        else: return (0, 1, 0)
    m = (i+j)//2
    zs, us, ts = es(S, i, m)
    zd, ud, td = es(S, m+1, j)
    return zs + zd, us + ud, ts + td + zs * ud

>>> S = '0011'
>>> es(S, 0, len(S)-1)
(2, 2, 4)
>>> S = '0101'
>>> es(S, 0, len(S)-1)
(2, 2, 3)
>>> S = '1100'
>>> es(S, 0, len(S)-1)
(2, 2, 0)

```

- La relazione di ricorrenza per il tempo di calcolo di questa implementazione è

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$$

che risulta  $\Theta(n)$ .

# PROGRAMMAZIONE Dinamica

negli esempi finora visti i sottoproblemi che si ottenevano dalla applicazione del passo 1 del divide et impera erano tutti diversi, pertanto ciascuno di essi veniva individualmente risolto della relativa chiamata ricorsiva del passo 2. In molte situazioni i sottoproblemi ottenuti al passo 1 possono risultare uguali. In tal caso, l'algoritmo basato sulla tecnica del divide et impera risolve lo stesso problema più volte svolgendo lavoro inutile.

la sequenza  $f_0, f_1, f_2 \dots$  dei numeri di Fibonacci è definita dall'equazione di ricorrenza:

$$f_i = f_{i-1} + f_{i-2} \text{ con } f_0 = f_1 = 1$$

Dato un intero  $n$  progettare un algoritmo che calcola  $f_n$ .

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

EQUAZIONE DI RICORRENZA

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

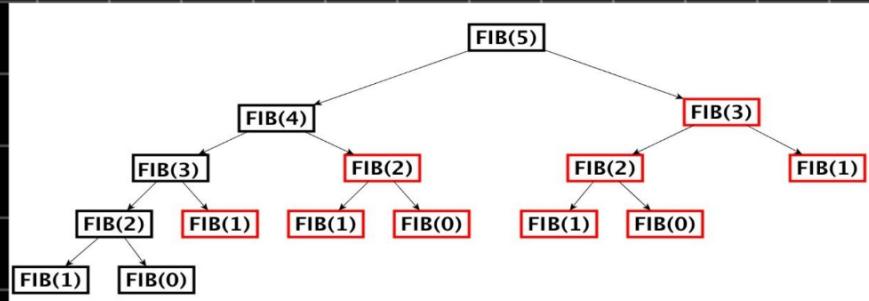
$$\text{OTTENIAMO } T(n) = \Omega(2^n)$$

Il motivo di questa inefficienza sta nel fatto che il programma "FIB" viene chiamato sullo stesso input molte volte e ciò è ridondante.

Basterà memorizzare in una lista i valori  $\text{fib}(i)$  quando li si calcola la prima volta cosicché nelle future chiamate ricorsive non ci sarà bisogno di ricalcularli ma potranno essere ricavati dalla lista (questa tecnica si chiama memorizzazione)

```
def fib1(n):
    F=[-1]*(n+1)
    return memfib(n,F)

def memfib(n,F):
    if n<=1: return 1
    if F[n]==-1:
        a=memfib(n-1,F)
        b=memfib(n-2,F)
        F[n]=a+b
    return F[n]
```



```
def Fib3(n):
    if n <= 1:
        return n
    a = b = 1
    for i in range(2, n+1):
        a, b = b, a+b
    return b
```

- nella versione ricorsiva dell'algoritmo si parte dal problema scomponendolo via via in sottoproblemi di dimensione sempre più piccola fino ad arrivare a problemi facilmente risolvibili (top-down)
- nella versione iterativa dell'algoritmo si comincia col risolvere i sottoproblemi di dimensione sufficientemente piccola per poi passare a quelli di dimensione via via crescente fino ad arrivare al problema originale (bottom-up)

Dato un intero  $n$  vogliamo contare le stringhe binarie lunghe  $n$  in cui non compaiono 2 zeri consecutivi.

Ad esempio  
per  $n=1$  sono 2: (0, 1)  
per  $n=4$  sono 5: (010, 011, 101, 110, 111)

L'algoritmo proposto deve avere complessità  $O(n)$ .

Utilizzeremo una tabella monodimensionale di dimensioni  $n+1$  e definiamo il contenuto delle celle come segue:

$T[i]$  = "numero di stringhe binarie lunghe  $i$  dove non compaiono 2 zeri consecutivi"

Una volta riempita la nostra tabella la soluzione la troveremo nella locazione  $T[n]$

RESTA da definire la regola ricorsiva con cui calcolare i valori  $T[i]$  nella Tabella

$$T[i] = \begin{cases} 1 & \text{Se } i=0 \\ 2 & \text{Se } i=1 \\ T[i-1] + T[i-2] & \text{altrimenti} \end{cases}$$

$n=3 \leftarrow (010, 011, 101, 110, 111)$

0	1	2	3	4	5
1	2	3	5		

la ricorrenza viene fuori da questo ragionamento:

- Il conteggio può essere visto come la somma delle stringhe da contare lunghe  $i$  che terminano con 0 e 1:

1. le stringhe da contare lunghe  $i$  che terminano con 1

Si ottengono accodendo la cifra 1 a quelle da contare

Lunghe  $i-1$ . Queste stringhe sono  $T[i-1]$

2. le stringhe da contare lunghe  $i$  che terminano con 0

devono avere al penultimo posto la cifra 1, si

ottengono dunque accodendo 10 alle stringhe da contare

Lunghe  $i-2$ . Queste stringhe sono dunque  $T[i-2]$

Dato un intero  $n$  vogliamo contare quanti modi diversi ci sono di sistemare  $n$  persone in un albergo che dispone di camere singole e camere doppie.

Ad esempio:

• per  $n=2$  sono 2: {[1], [2]} {[1,2]}

• per  $n=4$  sono 10: {[1], [2], [3], [4]}, {[1,2], [3], [4]}, {[1,3], [2], [4]}, {[1,4], [2], [3]}, {[2,3], [1], [4]}, {[2,4], [1], [3]}, {[3,4], [1], [2]}, {[1,2], [3,4]}, {[1,3], [2,4]}, {[1,4], [2,3]}}

L'algoritmo proposto deve avere complessità  $\Theta(n)$ .

• USIAMO una tabella monodimensionale  $n+1$

•  $T[i]$  = numero di modi in cui è possibile sistemare  $i$  persone nell'albergo

• la soluzione si troverà a  $T[n]$

$$T[i] = \begin{cases} 1 & \text{Se } i=0 \\ 1 & \text{Se } i=1 \\ T[i-1] + (i-1) \cdot T[i-2] & \text{altrimenti} \end{cases}$$

• possiamo vedere le disposizioni delle  $i$  persone come la somma di due diverse tipologie: quelle in cui la persona  $i$  finisce in camera da sola e quelle in cui la persona  $i$  finisce in camera con una delle altre  $i-1$  persone:

1. le differenti disposizioni in cui  $i$  finisce da solo sono  $T[i-1]$ , sono infatti tutti i possibili modi di disporre le altre  $i-1$  persone.
2. le differenti disposizioni in cui  $i$  finisce in camera con una delle altre  $i-1$  persone sono  $(i-1) \cdot T[i-2]$  infatti ci sono  $i-1$  modi di scegliere il compagno di stanza e una volta scelto il compagno ci sono  $T[i-2]$  modi di disporre tutte le altre  $i-2$  persone nelle varie camere.

```
def es3(n):
    T = [0 for _ in range(n + 1)]
    T[0]=T[1]=1
    for i in range(2, n + 1):
        T[i] = T[i-1] + (i-1)*T[i-2]
    return T[n]
```

tempo  $\Theta(n)$

Dato un array  $A$  di  $n \geq 2$  interi positivi vogliamo calcolare la somma massima per una sottosequenza di elementi di  $A$  non consecutivi.

Ad esempio per  $A = [5, 4, 2, 10, 6, 8]$  la risposta è 23 infatti per  $A$  la sottosequenza di interi non consecutivi di valore massimo è quella di seguito evidenziata in rosso 5, 4, 2, 10, 6, 8

L'algoritmo proposto deve avere complessità  $O(n)$

• USIAMO una tabella monodimensionale  $n-1$

•  $T[i]$  = la somma massima per una sottosequenza di elementi non consecutivi in  $A[i+1]$

• la soluzione si troverà in  $T[n-1]$

$$T[i] = \begin{cases} A[0] & \text{Se } i=0 \text{ // contiene un solo elemento} \\ \max(A[0], A[i]) & \text{// 2 elementi} \\ \max(T[i-1], T[i-2] + A[i]) & \text{altrimenti} \end{cases}$$

5	5	7	15	15	23
0	1	2	3	4	5

La ricorrenza viene fuori dal seguente ragionamento:

- Il vettore contiene un solo elemento allora meglio prenderlo perché positivo  $T[0] = A[0]$ .
- Il vettore contiene due soli elementi allora meglio prendere il più grande dei due  $T[1] = \max(A[0], A[1])$ .
- Alla soluzione di  $A[i+1]$  appartiene l'ultimo elemento  $A[i]$ . In questo caso l'elemento  $A[i-1]$  non può appartenere alla soluzione e quindi il massimo deve essere  $T[i-2] + A[i]$ . Se al contrario  $A[i]$  non appartiene alla soluzione allora per la soluzione deve avversi  $T[i] = T[i-1]$ .

In generale possiamo quindi dire che  $T[i] = \max(T[i-2] + A[i], T[i-1])$

```
def es(A):
    n = len(A)
    T[0] = A[0]
    T[1] = max(A[0], A[1])
    for i in range(2, n):
        T[i] = max(T[i-1], T[i-2] + A[i])
    return T[n-1]
```

N.B. Per calcolare il nuovo valore della Tabella bastano i 2 precedenti:

Quindi in Programmazione Si può migliorare passando dalla complessità di SPAZIO  $\Theta(n)$  a  $\Theta(1)$

Dato l'intero  $n$  vogliamo contare il numero di differenti tassellamenti di una superficie di dimensione  $n \times 2$  tramite tessere di domino di dimensione  $1 \times 2$ .

Ad esempio:

- per  $n = 1$  la risposta dell'algoritmo deve ovviamente essere 1
- per  $n = 2$  la risposta deve essere 2 perché sono possibili i soli due seguenti tassellamenti:



L'algoritmo deve avere complessità  $O(n)$

Motivare BENE la correttezza e la complessità dell'algoritmo proposto.

$\Theta(n)$

```
def es(n):
    m = max(3, n)
    T = [0] * (n+1)
    T[1], T[2] = 1, 2
    for i in range(3, n+1):
        T[i] = T[i-1] + T[i-2]
    return T[n]
```

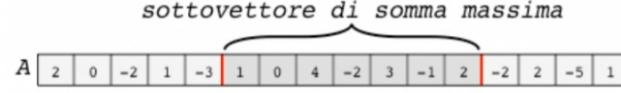
- Utilizzeremo una tabella monodimensionale  $T[i]$  e definiamo il contenuto  $T[i] = \text{il numero di Tassellamenti Possibili per la Superficie di Dimensione } i \times 2$

La soluzione sarà in  $T[n]$

Definiamo la regola RICORSIVA

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

Il problema del massimo sottovettore: Data una lista  $A$  di  $n$  interi, vogliamo trovare una sottolista (una sequenza di elementi consecutivi della lista) la somma dei cui elementi è massima.



$T[i] = \text{massima somma possibile per le sottoliste di } A \text{ che terminano nella posizione } i$

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

```
def es(A):
    n = len(A)
    T = [0] * n
    for i in range(1, n):
        T[i] = max(A[i], A[i] + T[i-1])
    return max(T)
```

tempo  $O(n)$  | SPAZIO  $\Theta(n)$

```
def es(A):
    n = len(A)
    m = t = 0
    for i in range(1, n):
        t = max(A[i], A[i] + t)
        m = max(m, t)
    return m
```

tempo  $O(n)$  | SPAZIO  $O(1)$

