Esercizio 1.1 (Nodi gemelli). *Dato un grafo G tale che $|V(G)| \geq 2$, dimostrare che esistono almeno due nodi distinti con lo stesso grado*

Supponiamo che ogni nodo ha grado diverso, ordiniamo tali nodi nel modo seguente: $\{v_1, v_2 \dots, v_n\}$ con $i > j \Rightarrow deg(v_i) > deg(v_j)$. Il nodo v_1 ha $k \geq 1$ adiacenti, il nodo v_2 ha $k' > k \geq 1$ adiacenti, e' chiaro che il nodo v_n avra' ALMENO n adiacenti, ma ci sono al piu' $n-1$ nodi che possono esserygli adiacenti $\Rightarrow$ CONTRADDIZIONE.

Esercizio 1.2 (Grafo complementare). *Dato un grafo G, definiamo $\overline{G}$ come grafo complementare di G se per ogni arco (u, v) si verifica che $(u, v) \in E(G)$ se e solo se $(u, v) \notin E(\overline{G})$. Dimostrare che almeno uno tra G e $\overline{G}$ è connesso.*

Supponiamo che entrambi i grafi non siano connessi

Esercizio 1.3 (Grado minimo $\frac{n}{2}$). *Dimostrare che un grafo non diretto G in cui ogni nodo ha grado maggiore o uguale a $\left\lceil \frac{|V(G)|}{2} \right\rceil$ è sempre connesso. L'affermazione vale anche se G è diretto?*

Esercizio 1.4 (Arcipelago). *Un arcipelago è rappresentato da una matrice $n \times m$, dove ogni cella è marcata da uno 0, rappresentante il mare, o da un 1, rappresentante il terreno. In particolare, due celle appartengono alla stessa isola se e solo se sono marcate entrambe con 1 e sono adiacenti. Data in input la matrice M, progettare un algoritmo che in tempo $O(nm)$ restituisca il numero di isole nell'arcipelago. Modificare l'algoritmo affinché restituisca la dimensione dell'isola di grandezza maggiore.*

```
Arcipelago( M[n,m]: matrice) {
    c=1
    For(i=0...,n){ Θ(n)
        For(j=0...,m){ Θ(m)
            if(M[i,j]==1){
                K= adj_island(M,i,j)//Θ(9)
                if(K==1){
                    c++
                    M[i,j]=c
                }else{ M[i,j]=K }
            }
        }
    }
    ni=0
    For(i=0...,n){
        For(j=0...,m){
            ni=max(ni,M[i,j]
        }
    }
    return ni-1
}

Biggest_island(M[n,m]: matrice) {
    ni=Arcipelago(M)// la funzione modifica la matrice
    A[ni+1]={0,0...,0}
    For(i=0...,n-1){
        For(j=0...,m-1){
            if(M[i,j]≠0){ A[M[i,j]]++ }
        }
    }
    bi=0
    in:intero
    For(i=0...,ni+1){
        if(A[i]>bi){
            bi=A[i]
            in=i
        }
    }
    return in
}
```

```
adj_island( M[n,m]:matrice, x:intero, y:intero){
    K=1
    For(i=x-1...,x+1){ Θ(3)
        For(j=y-1...,y+1){ Θ(3)
            i=clamp(0,n-1)
            j=clamp(0,m-1)
            K=max(K,M[i,j])
        }
    }
    return K
}
```

Esercizio 1.5 (Corsi propedeutici). *L'Università della Croce di Santa Minerva sta introducendo un nuovo corso di studi in Icosidodecaedrologia. Per gli n esami di tale corso sono previste m propedeuticità, ossia per sostenere alcuni corsi è necessario averne sostenuti prima altri. Ogni propedeuticità è rappresentata da una coppia (e_i, e_j) se per sostenere l'esame e_j è necessario sostenere prima l'esame e_i.*

Progettare un algoritmo che dati in input il valore n e un insieme P contenente le m coppie restituisca una possibile programmazione degli esami in modo che tutti possano essere sostenuti. Discutere il costo dell'algoritmo dato.

Si può modellare un grafo con nodi e_k ed archi (e_i, e_j). Basta fare un ordinamento topologico.

```
Prop( P={(e_i,e_j)} ){
    G:grafo
    For each (e_i,e_j)∈P {
        V(G).add(e_i)
        V(G).add(e_j)
        E(G).add((e_i,e_j))
    }
    L:list
    Vis[n]={0,...,0}
    return Ord(G,L,Vis)
}
```

```
Ord(G:grafo, L:list, Vis:array) {
    For each x∈V(G) {
        iF(Vis[x]≠1){ DFS_rec(G,x,L,Vis) }
    }
    return L
}

DFS_rec(G:grafo, x:nodo, L:lista, Vis:array) {
    Vis[x]=1
    For each y~x {
        iF(Vis[y]≠1){ DFS_rec(G,y,L,Vis) }
    }
    L.add(x)
}
```

Esercizio 1.8 (Cammino crescente più lungo). *Data una matrice M di dimensioni $n \times m$ di numeri interi positivi, definiamo come cammino della matrice una serie di indici $(i_1, j_1), \ldots, (i_k, j_k)$ ottenuti spostandoci ad ogni passo solo verso le entrate verticalmente o orizzontalmente adiacenti. Inoltre, definiamo un cammino della matrice come crescente se $m_{i_1,j_1} < \ldots < m_{i_k,j_k}$.*

Progettare un algoritmo di complessità $O(nm)$ che data in input la matrice M restituisca la lunghezza il cammino crescente più lungo al suo interno.

Creo un grafo diretto in cui ogni entrata della matrice e' un nodo ed ha un arco verso i nodi a destra/giu' di valore maggiore

```
Es8(M:matrice n×m){
    G:grafo
    For(i=1...,n){
        For(j=1...,m){
            V(G).add(i,j))
        }
    }
    For(i=1...,n){
        For(j=1...,m){
            iF(M[i,j]<M[i+1,j]){
                E(G).add((i,j),(i+1,j))
            }
            iF(M[i,j]<M[i,j+1]){
                E(G).add((i,j),(i,j+1))
            }
        }
    }
    return max_dist(G)//distanza massima fra 2 nodi
}
```

Esercizio 2.2 (Voli di costo minimo). *Un'azienda sta organizzando un evento in due sedi: una situata a Milano ed una a Roma. Per entrambe le sedi, l'azienda deve inviare n impiegati situati in città sparse. Per ognuno dei 2n impiegati viene calcolato il costo necessario per il viaggio verso Milano e il viaggio verso Roma. Tali costi sono descritti da una coppia (m_i, r_i), dove m_i e r_i sono rispettivamente il costo del viaggio verso Milano e verso Roma dell'i-esimo impiegato.*
Progettare un algoritmo di costo $O(n \log n)$ che date in input le 2n coppie $(m_1, r_1), \ldots, (m_{2n}, r_{2n})$ restituisca i due insiemi di n impiegati che minimizzino la spesa dell'azienda.

```
Voli( R = {(m1, r1)....,(m2n, r2n)}){
    M = R.copy()
    R = ordina secondo ri
    M = ordina secondo mi
    c=0
    while(R≠∅ ∧ M≠∅){
        r= R[0]
        R.remove(r)
        M.remove(r)
        m = M[0]
        c+= m[0]+r[1]
        M.remove(m)
        R.remove(m)
    }
    return c
}
```

NOTAZIONE

$k = (m_i, r_i) \Rightarrow \begin{cases} K[0] = m_i \\ K[1] = r_i \end{cases}$

Esercizio 2.3 (Shopping di lusso). *Siria ha recentemente fatto shopping sfrenato per articoli di lusso in una boutique esclusiva, all'interno della quale vi erano n articoli da cui scegliere. Siria, essendo nota per la sua ampia ricchezza, ha utilizzato una "parsimoniosa" strategia di acquisto: ispezionando ogni articolo dal primo all'ultimo, ha acquistato l'articolo attualmente ispezionato solo se il suo prezzo era inferiore ai fondi ancora disponibili, ignorandolo e passando al successivo se tali fondi fossero insufficienti per acquistarlo.*

Esercizio 2.4 (Ciclo di peso minimo). *Dato un grafo con pesi positivi G, il peso di un ciclo all'interno di G è la somma dei pesi degli archi del ciclo. Progettare un algoritmo che dato in input il grafo G restituisca l'insieme di archi che compone il ciclo di peso minimo presente in G (restituire $\varnothing$ se G è aciclico). La complessità prevista dell'algoritmo è $O(m(n+m)\log n)$.*

Per trovare un ciclo, faccio una DFS, quando considero un nodo che si trova ancora nello Stack, allora vuol dire che vi è un ciclo.

```
Compute_cycle(G:grafo, S:stack, u:nodo){ // questa funzione restituisce il ciclo considerato
    SS=S.copy() // mi prendo una copia dello stack
    A={ } // conterrà gli archi
    v=u
    do{
        w=SS.pop()
        A.add((w,v))
        v=w
    } while(w≠u)
    return A
}

MinCycle(G:grafo){
    Vis[n]={0,0...,0} // array lungo n
    inS[n]={0,0...,0} // array lungo n, se inS[u]=1⟺u e' nello stack
    S:Stack
    Out={ }
    outW=∞
    For each u∈V(G){ //nel caso il grafo sia diretto/non connesso
        iF(Vis[u]==0){
            Vis[u]=1
            S.push(u)
            inS[u]=1
            while(S≠∅){
                x=S.top()
                ex=0
                For each y∈x.adj(){
                    iF(Vis[y]≠1){
                        ex=1
                        S.push(y)
                        Dist[y]=1
                        inS[y]=1
                    }
                }
                iF(ex==0){ // x va tolto dallo stack
                    For each y∈x.adj(){
                        iF(inS[y]==1){ // controllo se forma cicli
                            C=Compute_Cycle(G,S,y)
                            iF(ω(C)<outW){
                                outW=ω(C)
                                Out=C
                            }
                        }
                    }
                    inS[S.pop()]=0
                }
            }
        }
    }
    return C
}
```

Esercizio 3.1 (Potenza modulare). *Progettare un algoritmo che dati tre interi a, n ed m calcoli il valore $a^n \pmod m$ in tempo $O(\log n)$.*

Esercizio 3.2 (Radice quadrata). *Progettare un algoritmo che dato un intero n calcoli il valore $\lfloor \sqrt{n} \rfloor$ in tempo $O(\log n)$*

Prop: $\begin{cases} \text{se } a^2 < n \Rightarrow \forall b < a,\ b^2 < n \\ \text{se } a^2 > n \Rightarrow \forall b > a,\ b^2 > n \end{cases}$

```
sqrt(n:intero, i:intero, prec:intero){
    if(n==1 ∨ n==2 ∨ n==3){return 1}
    if(i·i==n){return i}
    if((i-1)·(i-1)<n ∧ (i+1)·(i+1)>n){return i}
    if(i·i<n){
        k=⌊(prec+i)·1/2⌋
        return sqrt(n, k, i)
    }
    return sqrt(n,⌊i/2⌋, i)
}
```

Esercizio 3.3 (Somma complementare). *Siano X e Y due array di n interi ordinati in senso crescente. Dato un intero z e i due array X, Y, progettare un algoritmo di complessità $O(n \log n)$ che trovi, se esiste, una coppia di indici (i, j) tali che $X[i] + Y[j] = z$.*

```
ComSum(X:array, Y:array, z:int){
    n=X.length()
    LX:list
    LY:list
    For(i=0,1...,n-1){
        if(X[i]<z){LX.add(X[i])}
        if(Y[i]<z){LY.add(Y[i])}
    }
    i:intero
    For(i=0,1...,LX.length()){
        K=BS(LY, LX[i], z)
        if(k==NULL){continue}
        else{
            J=intero tale che LY[j]=K
            return (i,j)
        }
    }
}
```

```
BS(LY:array, x:int, z:int){
    K=LY.length()
    if(K==1){
        if(LY[0]+x==z){return LY[0]}
        else{return NULL}
    }
    i=⌊K/2⌋
    if(LY[i]+x==z){return LY[i]}
    if(LY[i]+x<z){return BS(LY[i+1,k], x, z)}
    return BS(LY[0,i-1], x, z)}
}
```

Esercizio 3.4 (Array con pozzo). *Dato un array A di n elementi, un indice i al suo interno è detto "pozzo" se si verifica che $A[1] > \ldots > A[i] < \ldots < A[n]$. Progettare un algoritmo di complessità $O(\log n)$ che, assumendo la sua esistenza, trovi il pozzo di un array.*

```
Pozzo(A:array){
    n=A.length()
    i=⌊n/2⌋
    if(A[i-1]>A[i]<A[i+1]){return A[i]}
    if(A[i-1]>A[i]){return Pozzo(A[i ,n-1])}
    return Pozzo(A[0, i])
}
```

Esercizio 4.1 (Sotto-array di prodotto massimo). *Dato un array A di n interi (positivi, nulli o negativi), progettare un algoritmo di complessità $O(n)$ che restituisca il prodotto massimo ottenibile con gli elementi di un sotto-array (un sotto-array contiene elementi contigui di A).*

Considero T[i] = Prodotto massimo di un solto array di A[0:i] contenente A[i]

```
PrMax(A: array){
    n = A.length()
    T[n] = {0,0...,0}
    T[0] = A[0]
    m = T[0]            {
    For(i = 1,2...n-1)
        T[i] = max(A[i], T[i-1]·A[i])
        m = max(m, T[i])
    }
    iF(m<0){return 0}
    return m
}
```

Esercizio 4.2 (Numero di passeggiate). *Progettare un algoritmo che dato in input un grafo diretto G, due vertici $x, y \in V(G)$ e un valore $k \in \mathbb{N}$ restituisca il numero di passeggiate da x a y distinte di lunghezza al massimo k. La complessità dell'algoritmo deve essere $O(nmk)$*

```
NumPass(G:grafo, x:nodo, y:nodo, K:intero){
    T[n x K]: matrix
    T[x,0] = 1
    For(i = 0,1...n-1){
        iF(i ∈ x.adj){T[i,1] = 1}
        else {T[i.1] = 0}
    }
    For(i = 0,1...,n-1){
        For(j = 1,2...K){
            s = 0
            For each(u ∈ i.adj){
                s += T[u,j-1]
            }
            T[i,j] = s
        }
    }
    sol = 0
    For(i = 0...,K){
        sol += T[y,i]
    }
    return sol
}
```

Esercizio 4.4 (Il Fantastico Mr. Fox). *Il Fantastico Mr. Fox è un ladro professionista e sta progettando un grande furto in una nota strada in cui si trovano n case adiacenti tra loro. Ogni casa del quartiere ha una certa quantità m_i di soldi al suo interno. Inoltre, ciascuna di esse è dotata di un sistema di sicurezza che contatterà automaticamente la polizia se sia essa che una delle case ad essere adiacenti vengono derubate. Ad esempio, se la casa 4 e la casa 5 venissero derubate verrebbe allertato il sistema, mentre ciò non accadrebbe se venissero derubate le case 3 e 5.*

1. *Progettare un algoritmo che dato in input l'insieme $m_1, \dots, m_n$ di soldi all'interno delle case restituisca la quantità massima di soldi che Mr. Fox può rubare in una sola notte*
2. *Se le case sono poste a cerchio, dunque la prima casa è adiacente sia alla seconda che all'ultima, come cambia l'algoritmo?*

```
Fox( {m1, m2 .... mn} ) {
    A[n] : array
    A[0] = m1
    A[1] = max(m1, m2)
    For (i = 2 ..., n-1) {
        A[i] = A[i-2] + m_{i+1}
    }
    return max(A)
}
```

Esercizio 4.7 (Dungeon). *Dei demoni hanno catturato ed imprigionato la principessa Leila alla fine di un dungeon, rappresentato da una griglia di $n \times m$ stanze. Partendo dall'inizio del dungeon, ossia la stanza $(0,0)$, un cavaliere vuole raggiungere la principessa, situata alla fine del dungeon, ossia la stanza (n,m). Durante la sua avanzata, il cavaliere può solo procedere nella stanza a sud o ad est della precedente (dunque può solo spostarsi dalla casella (i,j) alla casella $(i+1,j)$ o $(i,j+1)$).*

Il cavaliere possiede dei punti salute iniziali, rappresentati da un numero positivo p. Ogni stanza del dungeon, inclusa quella iniziale e quella finale, può contenere dei demoni all'interno (la cui quantità è rappresentata da un numero negativo) o contenere delle cure (la cui quantità è rappresentata da un intero positivo). Per ogni demone affrontato in una stanza, il cavaliere perderà un punto salute, mentre ne guadagnerà uno per ogni cura consumata. Se in un qualsiasi momento la salute del cavaliere raggiunge 0 o un numero negativo, il cavaliere morirà immediatamente.

Progettare un algoritmo che data in input la matrice M, i cui valori indicano gli interi rappresentanti gli elementi all'interno delle stanze del dungeon, trovi la minima quantità di salute che il cavaliere deve avere per salvare la principessa.

```
Dungeon(M: matrice n×m) {
    T: matrice n×m
    if(M[1,0] ≥ 0){ T[1,0] = (1, 1+M[1,0]) }
    else { T[1,0] = { (M[1,0]·(-1)) + 1, 1 ) }
    if(M[0,1] ≥ 0){ T[0,1] = (1, 1+M[0,1]) }
    else { T[0,1] = { (M[0,1]·(-1)) + 1, 1 ) }
    if(M[i,j] ≥ 0){
        P = {T[i-1,j], T[i,j-1]}
        (x,y) = argmin_{(x,y)∈P} (y)
        T[i,j] = (x, M[i,j] + y)
    }
    else{
        K = T[i-1,j].y + M[i,j]
        if(K > 1){ (x1,y1) = (T[i-1,j].x, K) }
        else { hp = (K·(-1)) + 1
               (x1,y1) = (hp, 1)
        }
        K = T[i,j-1].y + M[i,j]
        if(K > 1){ (x2,y2) = (T[i,j-1].x, K) }
        else { hp = (K·(-1)) + 1
               (x2,y2) = (hp, 1)
        }
        if(x1 < x2){ T[i,j] = (x1, y1) }
        else if(x2 < x1){ T[i,j] = (x2, y2) }
        else {
            if(y1 > y2){ T[i,j] = (x1, y1) }
            else { T[i,j] = (x2, y2) }
        }
    }
    return T[n-1, m-1]
}
```

Esercizio 4.8 (Bi-partizione di somma uguale). *Dato un insieme S di n interi positivi, progettare un algoritmo che restituisca True se esiste una partizione di S in due sotto-insiemi la cui somma degli elementi di entrambi sia uguale e False altrimenti.*

```
BiP(A:array, K:int, Sol){
    sum=0
    For(i=0,1...,n){ sum+=A[i] }
    For(i=0,1...,n){
        iF(K+A[i]=sum-A[i]){
            Sol=True
            return
        }else { BiP(A\{A[i]}, K+A[i], Sol)}
    }
}
```

```
Glob(A:array){
    Sol=False
    BiP(A,0)
    return Sol
}
```

Esercizio 4.9 (Din, Don, Jump!). *La parrocchia della Croce di Santa Minerva sta organizzando un torneo di un nuovo gioco dato dalla combinazione di Campana e Salto triplo: Din, Don, Jump!*

Il gioco prevede n caselle poste di fila, ciascuna dotata di un punteggio (positivo o negativo). Partendo dalla prima casella, ogni giocatore può effettuare un salto di k caselle, il quale può essere seguito da un numero arbitrario di salti i quali tuttavia devono essere di al massimo k caselle (dunque il numero di caselle saltate non può crescere). Per ogni casella toccata, il giocatore riceve il punteggio da essa indicato. In particolare, ogni giocatore è costretto a toccare la prima e l'ultima casella.

1. *Progettare un algoritmo di complessità $O(n^3)$ che dati in input i punteggi $p_1, \ldots, p_n$ restituisca il punteggio massimo ottenibile da un giocatore*
2. *È possibile ridurre il costo di tale algoritmo a $O(n^2)$?*

T[K,J]: punteggio ottenuto facendo salti di esattamente K fino alla J-esima casella

$$T[1,J] = \left(\sum_{i=2}^{J} P_i\right) + P_1$$

caselle

$$T[K,J] = \left(\sum_{i=2}^{J} P_{iK}\right) + P_1$$

se $iK \leq J$

	7	-2	5	1	10	12	-5	-5	-9	4	3
1	5	10	11	21	33	28	23	14	18	21	
2	~~x~~	12	~~x~~	22	~~x~~	17	~~x~~	8	~~x~~	11	
3	~~x~~	~~x~~	8	~~x~~	~~x~~	3	~~x~~	~~x~~	7	~~x~~	
4	~~x~~	~~x~~	~~x~~	19	~~x~~	~~x~~	~~x~~	10	~~x~~	~~x~~	
5	~~x~~	~~x~~		~~x~~							
6	~~x~~				~~x~~						
7						~~x~~					
8							~~x~~				
9								~~x~~			
10									~~x~~		