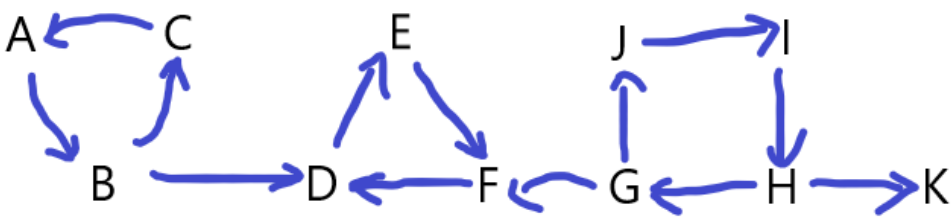
Зад. 1
Да се провери дали свързан неориентиран граф е двуделен.

```
isBipartite(G=(V, E)):
        n←|V|
        colors[1..n]←[none, …, none]
        q←Queue.Init()
        q.push(V[1])
        while q.isEmpty()=FALSE do
                u←q.pop()
                for each (u, v)∈E do
                        if colors[v]=none then
                                colors[v]←other(colors[u])
                                q.push(v)
                        else if colors[v]=colors[u] then
                                return false
        return true
```

Зад. 2
Да се намерят всички силно свързани компоненти на ориентиран граф G=(V, E)



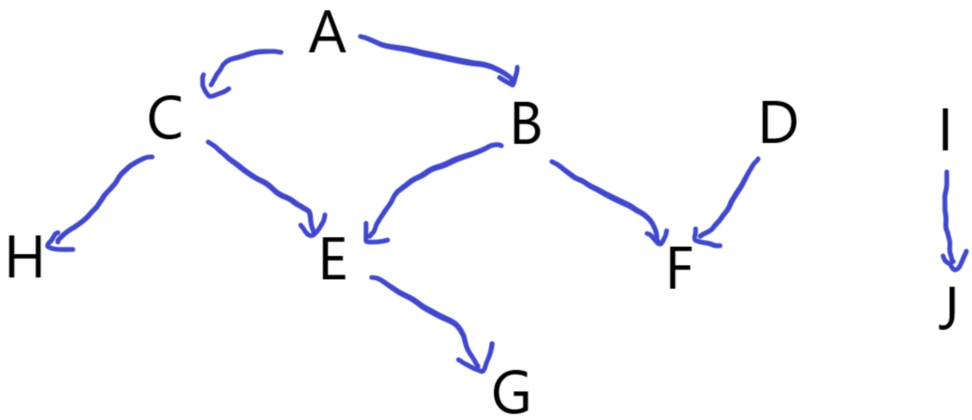

```
dfs(G=(V, E), n, v, s, visited[1..n]):
        if visited[v]=TRUE then
                return
        visited[v]←TRUE
        for each (v, u)∈E do
                if visited[u]=FALSE then
                        dfs(G, n, u, s, visited)
        s.push(v)

dfs2(G=(V, E), n, v, comp[1..n], compCnt, visited[1..n]):
        if visited[v]=TRUE then
                return
        visited[v]←TRUE
        comp[v]←compCnt
        for each (v, u)∈E do
                if visited[u]=FALSE then
                        dfs2(G, n, u, comp, compCnt, visited)
```

```
SCC(G=(V, E)):
        n←|V|
        visited[1..n]←[FALSE, …, FALSE]
        comp[1..n]←[-1, …, -1]
        compCnt←0
        s←Stack.Init() //stack of vertices
        for each v∈V do
                dfs(G, n, v, s, visited)
        G'←reverseGraph(G) //makes graph G'=(V', E'): V'=V & ∀(u, v)∈E ∃(v, u)∈E' & |E|=|E'|
        visited[1..n]←[FALSE, …, FALSE]
        while s.isEmpty()=FALSE do
                v←s.pop()
                if visited[v] then
                        continue
                dfs2(G, n, v, comp, compCnt, visited)
                compCnt←compCnt+1
        return comp
```

Зад. 3

Да се сортира топологически ацикличен ориентиран граф G=(V, E)




```
dfs(G=(V, E), n, v, s, visited[1..n]):
        if visited[v]=TRUE then
                return
        visited[v]←TRUE
        for each (v, u)∈E do
                if visited[u]=FALSE then
                        dfs(G, n, u, s, visited)
        s.push(v)

topSort(G=(V, E)):
        n←|V|
        s←Stack.Init() //stack of vertices
        ans←List.Init() //list of vertices
        visited[1..n]←[FALSE, …, FALSE]
```

```
for each v∈V do
    dfs(G, n, v, s, visited)
while s.isEmpty()=FALSE do
    ans.push_back(s.pop())
return ans
```

Зад. 4

Да се построи МПД на тегловен граф G=(V, E), n=|V|, m=|E|

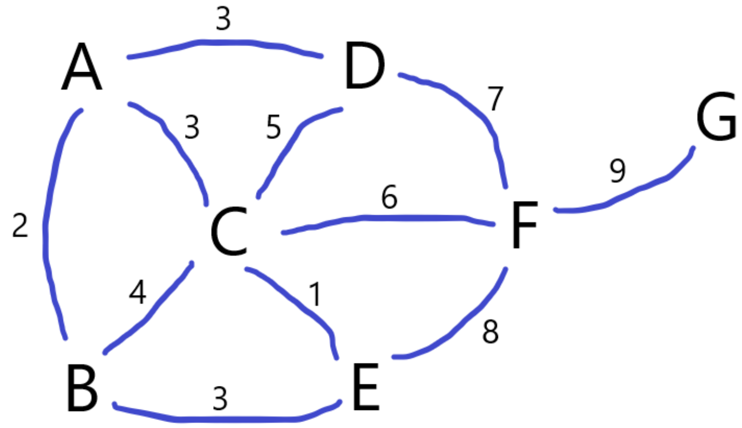
а) За време O(m*log(n)) - крускал

б) За време $O(n^2)$ - прим с масив (бърз спрямо а) и в), ако $m = O(n^2)$)

в) За време O(m*log(n)) - прим с binary heap

г) За време O(m+n*log(n)) - прим с Fibonacci heap

Б.О.О. G е свързан



а)

```
makeDSU(n):
    return [1, …, n]

findDSU(set, a):
    if set[a]≠a then
        set[a]←findDSU(set[a])
    return set[a]

unionDSU(set, a, b):
    a←findDSU(set, a)
    b←findDSU(set, b)
    if a≠b then
        set[b]←a
        return true
    return false

kruskal(G=(V, E)):
    sortIncByCoordinate(E, 3) //(u, v, w) - от къде, на къде, колко тежи
    E'←List.Init()
```

```
	set←makeDSU(|V|)
	for each (u, v, w)∈E do //in order
		if unionDSU(set, u, v)=TRUE then
			E’.push_back((u, v, w))
	return G’=(V, E’)
```

O(m*log(m) + 1 + 1 + m + 1) = O(m*log(n))

Ако имаме m на брой операции unionDSU/findDSU при set с размер n, то сложността е O(m*α(n)), където α е обратната ф-я на Ackermann.
т.е практически имаме, че O(m*α(n))=O(4m)=O(m)

б)

```
getMinIdx(dist[1..n], visited[1..n], n):
	idx←0
	for i←1 to n
		if visited[i]=FALSE then
			idx←i
			break
	for i←idx+1 to n
		if visited[i]=FALSE and dist[i]<dist[idx] then
			idx←i
	return idx

prim(G=(V, E)):
	n←|V|
	E’←List.Init()
	dist[1..n]←[∞, …, ∞]
	from[1..n]←[-1, …, -1]
	visited[1..n]←[TRUE, FALSE, …, FALSE]
	for each (V[1], u, w)∈E do
		from[u]←V[1]
		dist[u]←w
	for i←2 to n
		idx←getMinIdx(dist, visited, n)
		E’.push_back((from[idx], V[idx], dist[idx]))
		visited[idx]←TRUE
		for each (V[idx], u, v)∈E do
			if w<dist[u] then
				from[u]←V[idx]
				dist[u]←w
	return G’=(V, E’)
```

в)

```
prim2(G=(V, E)):
```

```
n←|V|
E'←List.Init()
visited[1..n]←[TRUE, FALSE, …, FALSE]
pq←PriorityQueue.Init() //min pq of 3-tuples, compared by 3rd coordinate
for each (V[1], u, w)∊E do
        pq.push((V[1], u, w))
for i←2 to n
        (v, u, w)←pq.pop()
        while visited[u]=TRUE do
                (v, u, w)←pq.pop()
        visited[u]←TRUE
        E'.push_back((v, u, w))
        for each (u, u', w')∊E do
                if visited[u']=FALSE then
                        pq.push((u, u', w'))
return G'=(V, E')
```

г) Няма да правим реализация, но за пълнота го добавяме към задачата. За разлика от binary heap-a, Fibonacci heap-a поддържа операцията decreaseKey(H, key, value) за амортизирано O(1) време. Това ни предоставя възможност да поддържаме heap с най-много |V| елемента, вместо |E| елемента както при binary heap-a. От това сложността ще падне на O(m+n*log(n))