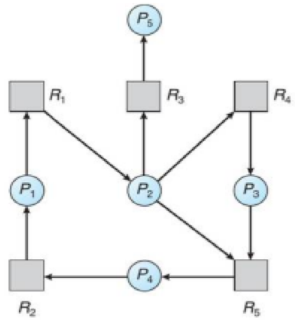
1. Write a program to create a resource allocation graph using adjacency list or adjacency matrix data structure. The final graph must be displayed as (i) the process names, (ii) the resource names, (iii) the list of request edges, and (iv) the list of allocation edges.



<u>Input:</u>

`5 5 11` →*(no. of processes, no. of resources, no. of edges): process names start from 0 (i.e. 0, 1, 2, …), resource names start at 100 (i.e. 100, 101, 102, …)*

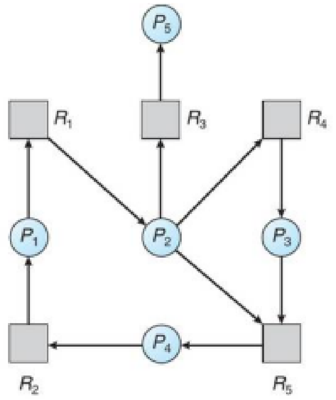
```
102 4
0 100
100 1
1 102
1 103
103 2
101 0
1 104
2 104
3 101
104 3
```

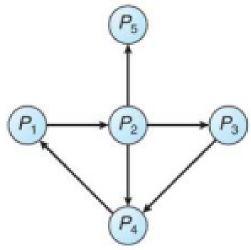
<u>Output:</u>

```
Processes: 0 1 2 3 4
Resources: 100 101 102 103 104
Request edges:
0 100
1 102
1 103
1 104
2 104
3 101
Allocation edges:
102 4
100 1
103 2
101 0
104 3
```

2. Write a program to check if cycles are present in a given resource allocation graph. Also check if a claim edge can create a cycle If no cycle is present print “Not deadlocked”

3. Write a program to convert a resource allocation graph to a wait-for graph. An edge P_i -> P_j exists in a wait-for graph if and only if the corresponding resource-allocation graph contains two edges P_i -> R_q and R_q -> P_j for some resource R_q.





<u>Input:</u>

`5 5 11` →*(no. of processes, no. of resources, no. of edges): process names start from 0 (i.e. 0, 1, 2, …), resource names start at 100 (i.e. 100, 101, 102, …)*

```
102 4
0 100
100 1
1 102
1 103
103 2
101 0
1 104
2 104
3 101
104 3
```

<u>Output:</u>

```
Processes: 0 1 2 3 4
Resources: 100 101 102 103 104
Request edges:
0 100
1 102
1 103
1 104
2 104
3 101
Allocation edges:
102 4
100 1
103 2
101 0
104 3
Wait-for edges:
0 1
1 4
1 2
1 3
2 3
3 0
```

4. Write a program to implement Banker's algorithm to find whether the system is in a safe state or not.

Process	Max	Allocation	Available
	A, B, C, D	A, B, C, D	A, B, C, D
P0	6 0 1 2	4 0 0 1	3 2 1 1
P1	2 7 5 0	1 1 0 0	
P2	2 3 5 6	1 2 5 4	
P3	1 6 5 3	0 6 3 3	
P4	1 6 5 6	0 2 1 2	

Explanation

- *Available:* A vector of length m indicates the number of available resources of each type. If *Available[j]* equals k, then k instances of resource type R_j are available.
- *Max:* An $n \times m$ matrix defines the maximum demand of each process. If *Max[i][j]* equals k, then process P_i may request at most k instances of resource type R_j.
- *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals k, then process *Pi* is currently allocated k instances of resource type R_j.
- *Need:* An $n \times m$ matrix indicates the remaining resource need of each process. If *Need[i][j]* equals k, then process P_i may need k more instances of resource type R_j to complete its task. Note that *Need[i][j]* equals *Max[i][j]* − *Allocation[i][j]*.

Algorithm

Step 1:

(a) Let `Work` and `Finish` be vectors of length m and n, respectively.

(b) Initialize `Work = Available` and `Finish[i] = false` for $i = 0, 1, ..., n - 1$.

Step 2: Find an index i such that both

(a) `Finish[i] == false`

(b) $\texttt{Need}_i \le \texttt{Work}$

If no such i exists, go to step 4.

Step 3:

$\texttt{Work = Work + Allocation}_i$

`Finish[i] = true`

Go to step 2.

Step 4: If `Finish[i] == true` for all i, then the system is in a safe state.

5. The table given below presents the current system state. The system is currently in safe state. Write a program to check if the following independent requests for additional resources makes the system safe or unsafe -

(a) REQ1: P0 requests 0 units of X, 0 units of Y and 2 units of Z

(b) REQ2: P1 requests 2 units of X, 0 units of Y and 0 units of Z

	Allocation			Max		
	X	Y	Z	X	Y	Z
P0	0	0	1	8	4	3
P1	3	2	0	6	2	0
P2	2	1	1	3	3	3

Algorithm

Let $Request_i$ be the request vector for process P_i. If `Requesti[j] == k`, then process P_i wants k instances of resource type R_j.

Step 1: If $\texttt{Request}_i \le \texttt{Need}_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

Step 2: If $\texttt{Request}_i \le \texttt{Available}$, go to step 3. Otherwise, print that P_i must wait, since the resources are not available.

Step 3: Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$$\texttt{Available} = \texttt{Available} - \texttt{Request}_i;$$
$$\texttt{Allocation}_i = \texttt{Allocation}_i + \texttt{Request}_i;$$
$$\texttt{Need}_i = \texttt{Need}_i - \texttt{Request}_i;$$

Check if the resulting resource-allocation state is safe using the safety algorithm.