A4Q1

```
A: Array of size n
PositiveSumPartition(A, n):
  s = sum of all elements of A, assert > 0
  p = sum of mod of all elements of A
  q = mod of sum of all negative elements of A
  # mem[i][j][k] == True means it is possible to find
  # j elements from A[1..i] with sum k - q
  mem = bool[n][n/2][p]

  # Base case
  for i = 1 to n:
    mem[i][1][q + A[i]] = True

  # Build table, O(n^3)
  for i = 1 to n:
    for j = 2 to min(i, n/2):
      for k = q to p:
        # sum is exactly between 1 and s-1 or not
        if (k - A[i] <= q or k - A[i] > q + s):
          mem[i][j][k] = mem[i-1][j][k]
        else:
          mem[i][j][k] = mem[i-1][j-1][k - A[i]]

  # Backtrack, O(n)
  i, j = n, n/2
  partitionOne = []
  for k = q to p:
    if mem[i][j][k] = True:       # Found a soln
      while (i > 0 and j > 0):
        if mem[i][j][k] = mem[i-1][j-1][k - A[i]]:
          partitionOne.push(A[i])
          k -= A[i]
          j--
        i--

      return (partitionOne, A - partitionOne)

  return False
```

Base case:

`mem[i][1][q + A[i]] = True for i = 1 to n`

At least 1 (j) element in `A[1..i]` will produce some `q + A[i]`

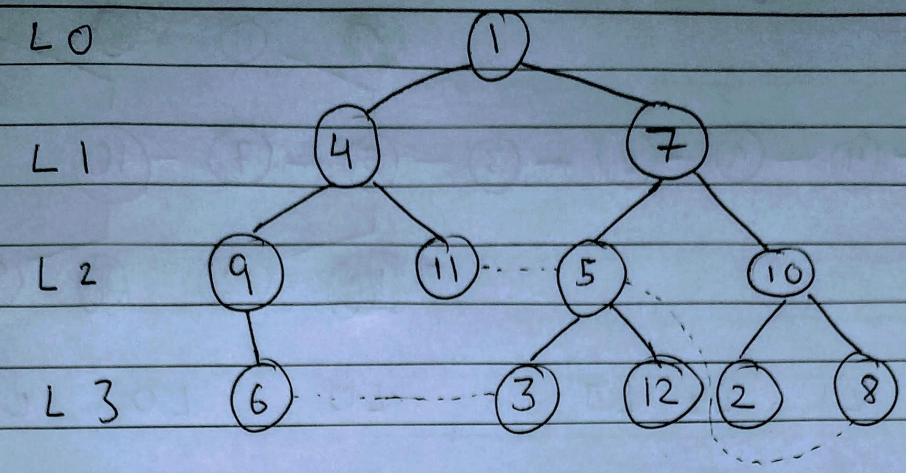
Recursive relation:

`mem[i][j][k] = mem[i-1][j-1][k-A[i]] or mem[i-1][j][k]`

Run-time analysis:

Generating the table is $O(n^3)$, backtracking is $O(n)$ as shown. Therefore total run-time is $O(n^3)$.

2. a



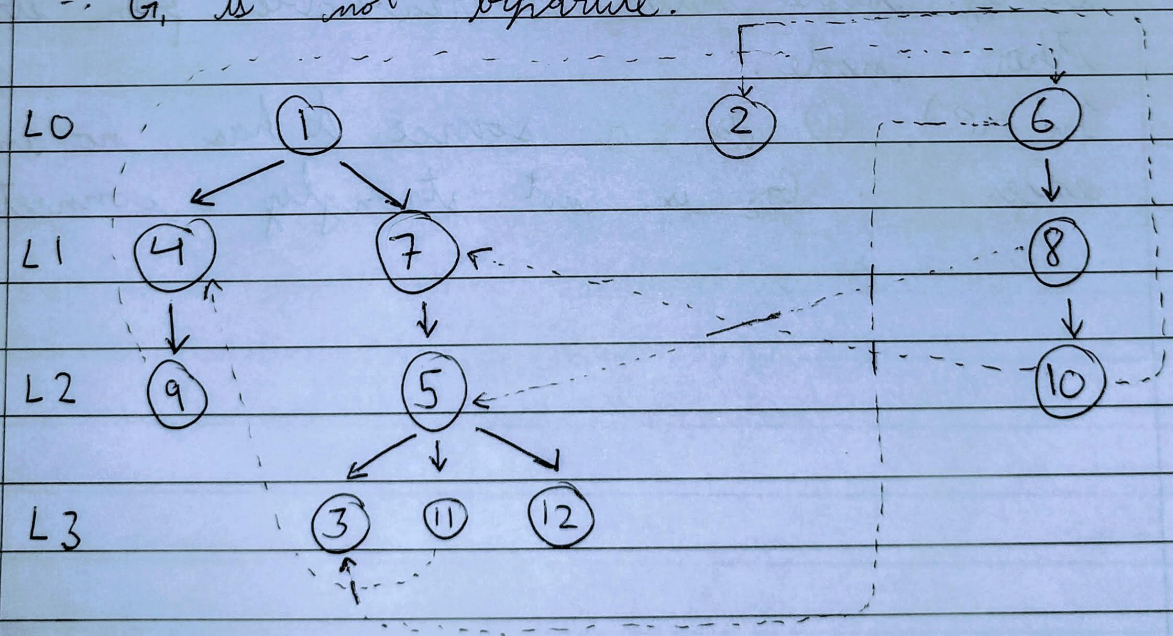
b G_1 contains an odd cycle between vertices

$8 - 6 - 9 - 4 - 1 - 7 - 5$

For a graph to be bipartite, it must not contain an odd cycle.

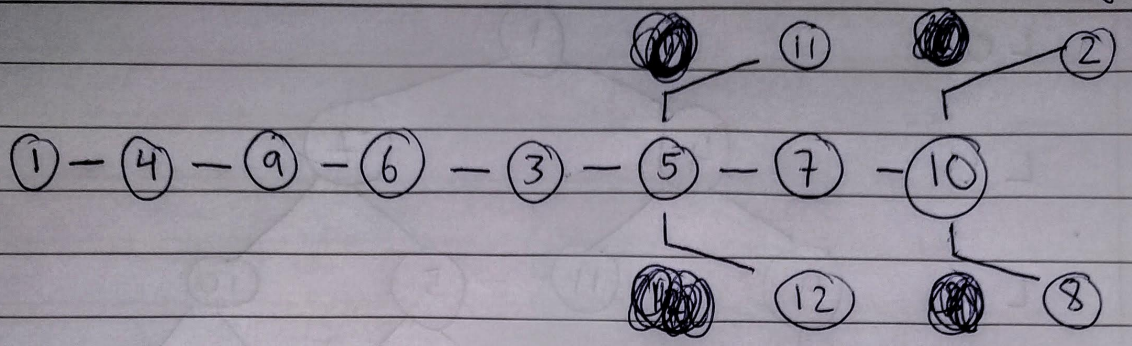
$\therefore G_1$ is not bipartite.

c

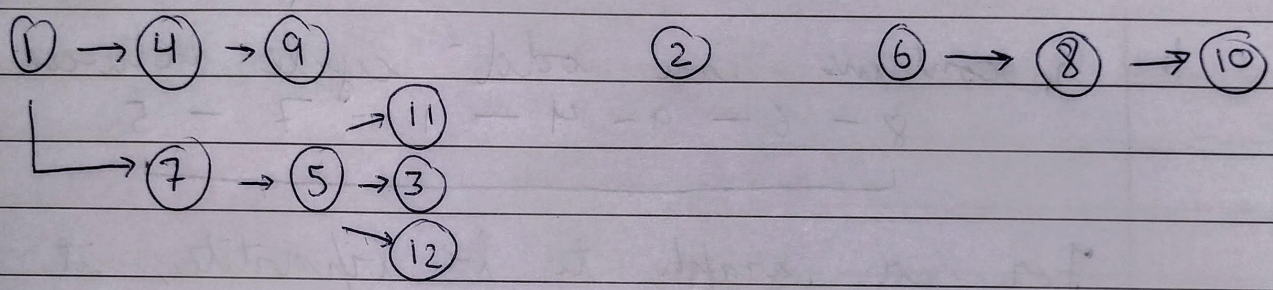


__/__/____

d



e



f For a graph to be strongly connected, every node must be reachable from every other node.
In (e), ① is a source & has no incoming edges. ∴ G_2 is not strongly connected.

A4Q3

```
G: Directed graph
returns array of edges in cycles
findEdgesInCycles(G):
  edgesInCycle = []
  for v in V(G):
    color[v] = white

  for v in V:
    if color[v] = white:
      edgesInCycle += findEdgesInCyclesVisit(v)

  return edgesInCycle


findEdgesInCyclesVisit(v):
  edgesInCycle = []

  color[v] = gray
  # Maintain stack of vertices currently in this traversal
  S.push(v)

  for w in Adj(v):
    if color[w] = white:
      edgesInCycle += findEdgesInCyclesVisit(w)
    if color[w] = gray:                          # Found Cycle
      verticesInCycle = []
      edgesInCycle += all edges between w and v using stack S

  color[v] = black
  S.pop(v)

  return edgesInCycle
```

The algorithm, similar to DFS and DFSVisit from class, is $O(n + m)$.

A4Q4

Create a vertex for every participant. $O(n)$ time.

Parse k requests:

1. "X same team as Y": Create an edge from $x \rightarrow y$ and $y \rightarrow x$
2. "X not same team as Y": Create an edge from $x \rightarrow y$

$O(k)$ time.

Find two strongly connected components in the graph using Kosaraju algorithm from class, remove the separation edge to have two components.

Distribute the vertices with no edges (participants with no requests) equally between the two components. $O(n)$ time.

We now have team 0 and team 1.

A4Q5

For each person, create two vertices x and x'.
x indicates the birth of the person while x' represents the death.
Create a directed edge $x \rightarrow x'$.
This takes $O(n)$ time, resulting in 2n vertices.

For each record,

1. A and B alive at the same time: Create two directed edges $a' \rightarrow b'$ and $b' \rightarrow a'$
2. A died before B was born: Create a directed edge $a' \rightarrow b$

This takes $O(m)$ time, resulting in m edges.

From class, we use topological sort using DFS to see if we have a directed acyclic graph (we make an exception for 2-node cycles $a' \rightarrow b'$ and $b' \rightarrow a'$). If we do, our records are consistent. If not, our records are not consistent.
This takes $O(m+n)$ time.

Runtime
The algorithm takes $\max\big(O(m+n), O(m), O(n)\big) = O(m+n)$ time.

Correctness
For our records to be consistent, we should have no non-two-node cycles in our directed graph. For example if we have two records that A and B lived at the same time and B died before A was born, we would have a cycle $a \rightarrow a' \rightarrow b' \rightarrow a$.