Assignment 5

Answer to question 1

- Since have a fixed number of records (3000), radix-exchange sort would be the right option as it would be the faster one in this case. Even though the heap sort will most likely take less memory to execute, it will most likely take more time. One major problem that will arise when using the heap sort is that the 64-bit integers will be considered large. This will eventually, under certain circumstances, cause complications when using heap sort. So, the best way to avoid this problem would be to use radix sort.

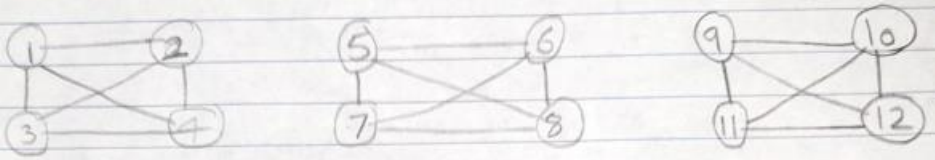
Answer to question 2

2)

3 connected components, 12 vertices, → minimum number of edges possible = 9 edges

maximum possible number of edges is $10C2 = 45$ edges

a)



b) Can't be done, as stated above, 3 connected components, 12 vertices → minimum number of edges = 9. $8 < 9$, so we can not achieve the required minimum number of edges.

c) can not be done, as stated above, maximum possible number of edges is $10C2 = 45$ edges. $56 > 45$, we can not get the required number of edges as it exceeds the maximum

D) 3 vertices with degree 3
4 " " " 2

Total = $3 \times 3 + 4 \times 2$
= $9 + 8$ = (17) degrees

edges $\times 2$ = Total degrees
= 17
= $\frac{17}{2}$ = 8.5

can't be done. can't achieve the minimum required number of edges.

Answer to question 3

a. List needs 10,000 + 20,000 = 30,000 entries. The matrix needs 10,000*10,0000 = 100,000,000 entries. We'll use the adjacency list
b. List needs:10,000+20,000,000=20,010,000 entries. The matrix needs $10,000^2$ or 10,000 * 10,000 = 100,000,000 entries. We'll use the adjacency list.
   So in the previous cases (a & b), to use as little space as possible we will need to use the adjacency list.
c. If you use the matrix structure you can answer the query are adjacent in constant time (practically instantaneously) but you will need to transverse the entire list of a node to do so in the list structure, so you must use the matrix structure.
d. To answer the areAdjacent query we would need to use the matrix representation. It will be done almost instantaneously (in constant time.) Moreover, we will need to transverse the entire list of nodes to do the same thing in adjacency list representation. One more reason to do it in matrix structure

Answer to Question 4

```
Algorithm DFS(graph,start)
input: Graph graph, vertex start

DFS (graph, start):
      let stackvar be stack
      stackvar.push( start )            //stackvar is a stack variable. Pushing
start in stack
      mark start as visited.
      while (stackvar is not empty):
          vertex  =  stackvar.top( )
          stackvar.pop( )
          for all neighbours nb of vertex in Graph graph:
             if nb is not visited then
                    stackvar.push( nb )
                    mark nb as visited
```

What a DFS would do is start at a certain vertex and go will keep going through the vertices after until it comes to a cross paths point (a point where a decision should be made regarding which vertex to go through first). When that happens, it will pick one and keep going through it until the end. If it comes to another cross paths point it will do the same thing. When it reaches the end of a “path”, it will go back to the previous “cross path” and choose the other “path” and go through it until it reaches the end and the process will repeat until our target is reached or what we are looking for is not present in the stack (DFS always involves a stack).

Answer to Question 5

```
Verices[G.size()] array;

Algorithm cycleDFS(G, v)
        setLabel(v, VISITED)
        S.push(v)
        for all  e ∈ G.incidentEdges(v)
                if  getLabel(e) = UNEXPLORED
                        w ← opposite(v,e)
                        S.push(e)
                        if getLabel(w) = UNEXPLORED
                         setLabel(e, DISCOVERY)
                        cycleDFS(G, w)
                        S.pop(e)
                else
                        T ← new empty stack
                        repeat
                        o ← S.pop()

                        exists ← false

                        for (i ← 0; i  < array.length; i++)

                                if (array [i] = o)

                                        exists ← true

                        if( exists = false)

                                add o to the array
                        T.push(o)

                until o = w
                 return T.elements()

        S.pop(v)
```

Answer to Question 6

To create the algorithm, first consider graph called “W.” Each edge in the graph represents a rivalry while each vertex represents a wrestler. That graph is then to be created containing r edges and n vertices. After this step, we will preform breadth first search (BFS) as many times as we need to visit (check) all the vertices. After that, on one hand, we will assign even-weighed wrestlers to a category (list) and let’s call it evens. On the other hand, we will assign odd-weighed wrestlers to a list, and we will call it odds. After that we will check that every edge goes between an element (wrestler) in the odds list and another element (wrestler) in the evens list.

This algorithm should take O (n + r) time.