Please follow the instructions provided below to submit your assignment.

Worth: 10% **Due:** By 11:59 pm on Friday Dec 8

- You must submit your assignment as a single PDF file through the MarkUs system.

  https://markus.teach.cs.toronto.edu/csc263-2017-09/

  The filename must be the assignment's name followed by "sol", e.g. your submission for the assignment "A4" must be "A4sol.pdf". Any group of two students would submit a single solution through Markus. Your PDF file must contain both team member's full names.
- Make sure you read and understand "POLICY REGARDING PLAGIARISM AND ACADEMIC OFFENSE" provided in the course information sheet.
- The PDF file that you submit must be clearly legible. To this end, we encourage you to learn and use the LaTex typesetting system, which is designed to produce high-quality documents that contain mathematical notation. You can find a latex template in Piazza under resources. You can use other typesetting systems if you prefer. Handwritten documents are acceptable but not recommended. The submitted documents with low quality that are not clearly legible, will not be marked.
- You may not include extra descriptions in your provided solution. The maximum space limit for each question is 2 pages. (using a reasonable font size and page margin)
- For any question, you may use data structures and algorithms previously described in class, or in prerequisites of this course, without describing them. You may also use any result that we covered in class, or is in the assigned sections of the official course textbook, by referring to it.
- Unless we explicitly state otherwise, you should justify your answers. Your paper will be marked based on the correctness and completeness of your answers, and the clarity, precision, and conciseness of your presentation.
- For describing algorithms, you may not use a specific programming language. Describe your algorithms clearly and precisely in plain English or write pseudo-code.

1. (25 Mark) This question shows the importance of data structures in the study of philosophy. A **paradox** is a group of statements that lead to a contradiction. For example, consider the following group of two statements which form a paradox:

   1) Statement 2 is FALSE.
   2) Statement 1 is TRUE.

   If we assume Statement 1 is TRUE, which says Statement 2 is FALSE, which in turn means Statement 1 is FALSE, therefore we have got a contradiction. If we assume Statement 1 is FALSE, which means Statement 2 is TRUE, which in turn means Statement 1 is TRUE, again we have got a contraction. That is, assuming one statement to be TRUE or FALSE will lead to deriving a conclusion that is opposite to the assumption.

   Now you are given a group of N statements, numbered from 1 to N. Each statement has the format "`Statement X is TRUE/FALSE`" where `X` is a number between 1 and N. Your task is to figure out whether this group of statements form a paradox. In particular, answer the following questions.

   (a) Apparently you are supposed to use a graph. How do you construct the graph for solving this problem? We construct of a directed graph G with N vertices, each of which represents one of the given statements, numbered from 1 to N. A directed edges (x, y) is added to the graph if statement x comments on statement y. If x says y is TRUE, then (x, y) is a "true edge" to which we assign a weight 0; if x says y is FALSE, then (x, y) is "false edge" to which we assign a weight 1.

   (b) What property must the graph have if the N statements do form a paradox? What property must the graph have if the N statements do NOT form a paradox? Justify your answer. The group of statements will form a paradox if and only if there exists an odd-weighted cycle in the graph G, *i.e.*, a cycle with an odd number "false edges". The reason is that, starting from any vertex in this cycle, classifying the starting vertex as either TRUE or FALSE, going through this cycle will negate the initial classification of the starting vertex, which is exactly how a contradiction is formed.

   (c) How do you efficiently detect whether the graph has the properties you described? Describe your algorithm in concise English. Knowing this properties, we can simply perform DFS on the graph G. During the procedure, whenever we encounter a back edge, *i.e.*, we got a cycle, we backtrack the $\pi[u]$ pointers and compute the total weight of the edges in the cycle (we stop backtracking when we reach the vertex which the back edge is pointing at). If the weight is odd, then we can terminate and output "PARADOX"; if the weight is even, keeping going with DFS. If we finish DFS on G (*i.e.*, all vertices in G are finished) and there is not an odd-weighted cycle, then we can output "NO PARADOX".

   (d) What is the worst-case runtime of your algorithm? constructing the graph takes $\Theta(N)$, the DFS (including the backtracking) takes $\Theta(|V| + |E|)$ which is also $\Theta(N)$. Therefore the overall worst-case runtime of this algorithm is $\Theta(N)$.

   Another approach for this problem is using vertex colouring, *i.e.*, the group of statements do not form a paradox if and only if you can colour all vertices using two colours (representing the classification of true or false for each statement), where if statement x says y is true then x and y should be of the same colour and if statement x says y is false then x and y should be of different colours. You can then use either BFS or DFS to traverse and colour the graph, and whenever you find that a valid colouring is impossible you return "PARADOX".

2. (25 Marks) Assume that you are a kindergarten teacher, who is given a task to arrange n ill-behaved children in a straight line, facing front. You are given a list of m statements of the form "kid i hates kid j". If kid i hates kid j, then you do NOT want to put i somewhere behind j, because then i is capable of throwing something at j.

   Design an algorithm that efficiently computes a way of arranging these kids so that no throwing can happen, if such an arrangement is possible at all. In particular, answer the following questions.

   (a) How do you model this problem using a data structure that we have studied?

   Use a directed graph where each vertex represents a kid. If kid i hates kid j, we add directed edge (i, j) to the graph.

   (b) Is it always possible to find a valid arrangement? If yes, briefly explain why. If no, briefly explain what is the case in which it is not possible.

   An arrangement is possible if and only if the graph is acyclic.

   (c) How does your algorithm work? And what is its worst-case runtime (in asymptotic notation)? Explain in concise English.

   Perform a topological sort, *i.e.*, do a DFS and sort all vertices according to their finishing time $f[u]$. The worst-case runtime is $\Theta(|V| + |E|) = \Theta(m + n)$.

   (d) Suppose instead you want to arrange the children in *rows* such that if kid i hates kid j, then kid i must be in a lower numbered row than kid j (they cannot be in the same row). **Design** an efficient algorithm to find the minimum number of rows needed, if the arrangement is possible. **Explain** in clear English the design of your algorithm and its worst-case runtime (in asymptotic notation). Write pseudo-code if it helps explaining.
   Hint: Start from the straight line you got from the previous arrangement.

   The number of rows needed is computed based on the topological sort we obtained in the previous part. Let $S[1:n]$ be the sequence of topological sorted vertices. The algorithm is described in the following pseudo-code.

```
MINIMUM-ROWS(S):
1   rows-needed ← 1
2   row[1:n] ← a sequence of n 1's    # keeps the row number kid i should be in
3   for i from 1 to n - 1:
4       for each v in Adj[S[i]]:
5           if row[v] < row[S[i]] + 1:
6               row[v] ← row[S[i]] + 1
7               if row[v] > rows-needed:
8                   rows-needed ← row[v]
9   return rows-needed
```

   Intuitively, you put the kids arranged in the line into rows, starting from first kid in the line, putting them into the lowest numbered row possible. Whenever you have a kid who is hated by someone already in the rows, you have to put him into the next row of hater's row, creating a new row when there is no next row.
   The runtime this algorithm is still $\Theta(m + n)$, since essentially it just traverses the graph again.

3. (25 Marks) Consider the abstract data type DEPR that consists of a set S of positive integers. Initially, S is a set of n consecutive integers. $S = \{a, ..., b\}$ such that $b = a + n - 1$. DEPR supports just the following two operations:

   - DELETE(S, i): Delete integer i from the set S. If $i \notin S$, there is no effect.
   - PREDECESSOR(S, i): Return the predecessor in S of integer i, i.e. $\max\{j \in S \mid j < i\}$.
     If i has no predecessor in S, i.e. if $i \leq \min S$, then return 0. Note that it is not necessary for i to be in S.

   For example, if $a = 1, n = 7$ and $S = 1, 3, 6, 7$, then PREDECESSOR$(S, 1)$ returns 0, PREDECESSOR$(S, 2)$ and PREDECESSOR$(S, 3)$ return 1.

   Describe a data structure with $O(log^* m)$ amortized cost per operation, where m is the number of operations that are performed. Justify the correctness and time complexity of your data structure.

   How do you initialize your data structure and how much time does it take?

   Use a disjoint-set forest data structure for the elements in $\{0\} \cup \{a, \ldots, b\}$ with union by rank and path compression. The root r of each tree also contains the value MT$[r]$ of the minimum element in its tree. Initialize the disjoint-set forest data structure by performing $n + 1$ MAKESET operations and setting the MT field of every element to be itself.
   The two operations can be performed as follows:

```
PREDECESSOR(S,i):
1  if i <= a then return 0
2  if i > b then return MT[FIND(b)]
3  return MT[FIND(i-1)]
```

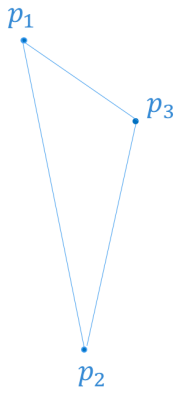
```
DELETE(S,i):
1  if (i < a) or (i > b) then return
2  if i = a then UNION(0,a)
3  else UNION(i-1,i)
4  set the MT field of the resulting tree to be the minimum of
   the MT fields of the two trees that were linked
5  return
```

   Every tree in the forest consists of a single element of $S \cup \{0\}$ plus all the elements of $\{a, \ldots, b\} - S$ that have that element as their predecessor in $S \cup \{0\}$. That element is the smallest element in the tree and thus the value of the MT field at the root of the tree. When i is deleted, all elements that used to have i as their predecessor now have the next largest element in $S \cup \{0\}$ as their predecessor. This is the smallest element in the set containing $i - 1$.
   If $i \leq a$, then i has no predecessor in S, so PREDECESSOR(S, i) must return 0. If $a < i \leq b$, then the predecessor of i is either $i - 1$, if $i - 1 \in S$, or the predecessor of $i - 1$ in $S \cup \{0\}$, if $i - 1 \notin S$. In either case, this is the smallest element in the set containing $i - 1$. Finally, if $i > b$, then the predecessor of i is the largest element in $S \cup \{0\}$, which is in the set containing b.
   Because PREDECESSOR performs at most one FIND operations plus a constant amount of other work and DELETE performs at most one UNION operation plus a constant amount of other work, the result state in the lectures immediately gives sequence complexity $O(m \log^* n)$ where $n < m$ and $O(mlog^* m)$ where $m < n$ and, hence, amortized time complexity $O(\log^* m)$.

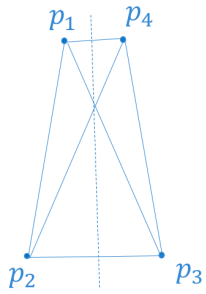
(Note that if $m < n$, then the union operations are executed maximum m times, so the size of the disjoint trees cannot exceed $O(m)$.)

4. (10 Marks) Consider the problem of finding the minimum spanning tree connecting n distinct points in the plane, where the distance between two points is the ordinary Euclidean distance. In this problem we assume the distances between all pairs of points are distinct. For each of the following procedures, either argue that it constructs the minimum spanning tree of the points or give a counterexample.

   (a) Sort the points in order of their x-coordinate. (You may assume without loss of generality that all points have distinct x-coordinates; this can be achieved if necessary by rotating the axes slightly.) Let $(p_1, p_2, ...p_n)$ be the sorted sequence of points. For each point p_i, $(2 \leq i \leq n)$ connect p_i with its closest neighbor among $p_1, ..., p_{i-1}$.
   This procedure does not work. Notice that in the figure below (a) we must use the line segment (p_1, p_2). However, using the segment (p_1, p_3) would yield a smaller spanning tree. Therefore, this procedure does not produce a MST.



(a)



(b)

   (b) Draw an arbitrary straight line that separates the set of points into two parts of equal or nearly equal size (i.e. within one). (Assume that this line is chosen so it doesn't intersect any of the points.) Recursively find the minimum spanning tree of each part, then connect them with the minimum-length line segment connecting some point in one part with some point in the other (i.e. connect the two parts in the cheapest possible manner).
   This procedure does not work. Notice that in the figure above (b) if we follow the method suggested with the choice of line as in figure, then $(p_1, p_2), (p_3, p_4), (p_1, p_4)$ or $(p_1, p_2), (p_3, p_4), (p_2, p_3)$ would be the tree output by the algorithm. Clearly the tree $(p_1, p_4), (p_1, p_2), (p_2, p_3)$ is a MST.

5. (15 Marks) Let G be a connected undirected weighted graph. Assume that the weights are distinct meaning that no two edges has the same weight. Prove that the graph has a unique MST.

   Suppose that for every cut of the graph, there is a unique light edge crossing the cut, but that the graph has two spanning trees T and T'. Since T and T' are distinct, there must exist edges (u, v) and (x, y) such that (u, v) is in T but not T' and (x, y) is in T' but not T. Let $S = \{u, x\}$. There is a unique light edge which spans this cut. Without loss of generality, suppose that it is not (u, v). Then we can replace (u, v) by this edge in T to obtain a spanning tree of strictly smaller weight, a contradiction. Thus the spanning tree is unique.