This homework is due on gradescope Friday January 14th at 11:59pm on gradescope. Remember to justify your work even if the problem does not explicitly say so. Writing your solutions in \LaTeX is recommended though not required.

**Question 1** (Puzzle Solving, 35 points). Victoria is trying to solve a puzzle. In this puzzle she has an $n \times n$ board. Some of the squares are marked with obstacles, and one of which is a designated target square. The board also has two pawns on it. As a move, Victoria may take either pawn and move it as far as it goes in one of the four cardinal directions until it bumps into an obstacle, the edge of the board, or the other pawn. Her objective is to make some sequence of moves that ends with one of the pawns at the target square. For example, in the figure below, the moves shown allow her to bring pawn A to the target square in 6 moves. Find an algorithm that given $n$, and the locations of the obstacles, pawns, and target runs in time polynomial in $n$ and determines whether or not it is possible for Victoria to solve this puzzle. What is the runtime of this algorithm?

**Solution 1. High Level Description**

We can map the given puzzle into a graph problem. We can build a graph $G$ such that the vertices of $G$ correspond to the pair of pawn locations $(A, B)$, and there is an edge between $(A, B)$ and $(A', B')$ if there is a single move which allows the state of the game to change from one configuration to the other. Before running our graph algorithm on $G$, we can also preprocess our data in a manner which would allow us to efficiently compute the edges in our graph.

**Constructing the graph:** Every possible combination of pawn A and B’s position on the board corresponds to a unique state of the game. Hence we create a vertex for each pair of locations of $(A, B)$. For each such pair $(A, B)$ there would be atmost 8 moves. For each piece that we try to move in each direction, we would compute the resulting puzzle board, and add a directed edge connecting our current vertex to the new vertex (new resulting board). Note that this way of computing an edge takes $O(n)$ time, we shall optimize this operation down to $O(1)$ later.

**Algorithm:** Given the above graph $G(V, E)$, we can call the explore algorithm on the initial positions of the pawn A and B. We can find out it is possible for Victoria to solve the puzzle, if there is a vertex which has A or B on the target square marked as visited (i.e either pawns reaching the target square). To compute $(A', B')$ an edge should connect $(A, B)$ to (with respect to the valid move) we can traverse through each cell in the grid and stop at the square beyond which there is an obstacle or end of the grid.

**Runtime Analysis:** Explore algorithm has a complexity of $O(|V| + |E|)$. There are a total of $n^4$ unique possible positions for A and B on the puzzle board. Hence, $|V| = n^4$, and corresponding to each position there can be 8 possible edges connecting it to a new position. Hence, $|E| = 8n^4$. Thus our explore algorithm has a complexity of $O(n^4 + 8n^4) = O(n^4)$. The cost of computing an edge in our current explore algorithm would take $O(4n)$ time at most for all 4 directions i.e $O(n)$. This results in an overall time complexity of $O(n^5)$.

**Proof:** We are computing edges from $(A, B)$ to $(A', B')$ corresponding to every possible valid move which can be made from one state/configuration of the game to another. The only way pawn A or B can reach the any square on the board, is by transitioning through a sequence of these states, under the given constraints of a valid move. Hence, any possible sequence of moves (using pawn A or B) by Victoria would be captured
by a set of edges connecting those particular states of the game, and these edges would represent the path in our graph G. Thus we can prove that if there is a sequence of moves Victoria can make, there would always exist an equivalent path in our graph.

**Optimizing via preprocessing:** However, with some preprocessing and use of data structures we can optimize the cost of computing an edge down to $O(1)$. We observe that while the position of the other pawn may vary, the position of obstacles will stay the same throughout the game. Thus, while ignoring the position of the other pawn, if we can find the furthest distance a pawn can travel from any square on the board, w.r.t only the obstacles beforehand, it might help us compute an edge effectively. We can use a array or hashmap (with key to value pair being, $(n \times n \times direction)$: new position of the pawn). Since there are $n^2$ possible positions, and $4n$ squares at most from each position to check, this preprocessing step will take us $O(n.n.4n)$ time i.e $O(n^3)$. This helps us compute our edge during the explore algorithm in $O(1)$ time, as we only need to check for a valid move corresponding to position of one B pawn, as the valid move taking obstacles into account has already been calculated beforehand. This reduces our explore algorithm’s complexity from $O(n^5) \rightarrow O(n^3 + n^4) \rightarrow O(n^4)$.

**Question 2** (Connecting a Graph, 20 points). Let $G$ be an undirected graph. Give a linear time algorithm to compute the smallest number of edges that one would need to add to $G$ in order to make it a connected graph.

**Solution 2.** The smallest number of edges that need to be added in order to make G a connected graph is the number of connected components - 1. We can use DFS to calculate the number of connected components in the graph using the algorithm discussed in class.

The time complexity of this algorithm is linear in terms of the number of vertices and edges. It is $O(|V| + |E|)$.

**Proof:** We can prove the above claim by using induction on the number of connected components. Consider the base condition where we have one connected graph. The number of connected components is 1. Therefore 0 edges need to be added. Thus the hypothesis holds true.

Let us assume the hypothesis holds true for $k$ connected components i.e. the smallest number of edges required is $k - 1$. Consider a graph G with $k + 1$ connected components. An edge can be added to this graph in the following two ways:

- **Between two vertices in the same component:** Consider two vertices $u$ and $v$ belonging to the same connected component in G. Therefore $u$ is reachable from $v$. Adding an edge between $u$ and $v$ does not change the number of connected components in G since $u$ was already reachable from $v$ (and vice versa) before adding the edge. Therefore the number of connected components in G remains constant.

- **Between two vertices that belong to two different components:** Let us consider two vertices $u$ and $v$ belonging to two different connected components $C1$ and $C2$ in G. Adding an edge between $u$ and $v$ makes $u$ reachable from $v$ and vice versa. Let there be a vertex $w$ in $C1$. Since $u$ and $w$ are in the same connected component, $u$ is reachable from $w$. Due to the newly added edge $(u, v)$ $v$ is now reachable from $w$. Since $v$ (which was originally in $C2$) is now reachable from $w$ (which was originally in $C1$), we can see that the two components have reduced to a single connected component. The other connected components are unaffected since they are all disconnected from $C1$ and $C2$. Since $C1$ and $C2$ are now merged to form a single connected component, the number of connected components in G has reduced exactly by 1.

Assume for the sake of contradiction that a single edge is added which decreases the number of connected components by 2 or more. This edge can be added in the following two ways:

- **Between two vertices in the same component:** We already showed that the number of connected components remains constant in this case. It is not possible for the number of connected components in G to reduce. This is a contradiction and this edge cannot cause the number of connected components to decrease by 2 or more.
• Between two vertices that belong to two different components: Let us consider two vertices u and v belonging to C1 and C2. Adding an edge between u and v, makes all vertices in C2 reachable from all vertices in C1 (and vice versa) as shown above. The edge added, is between exactly two vertices that belong to C1 and C2. Since a single edge can only affect at most two connected components, all other components in G remain the same. Therefore this edge cannot cause the number of connected components to decrease by $\geq 1$.

Therefore, adding one edge to a graph reduces the number of connected components by at most 1. If we decrease the number of connected components by 1, it takes $k - 1$ edges to connect the graph. As long as the number of connected components $> 1$, there is always an edge that can be added to the graph to reduce this number. This is because an edge can always be added between any two vertices belonging to two different connected components to connect them. Hence proving our claim.

**Question 3** (Explore Children, 20 points). Let G be a connected, undirected graph and v one of its vertices. Suppose that explore(v) is run. Show that the number of children of v in the resulting DFS tree equals the number of connected components of $G - \{v\}$.

**Solution 3.** We show two directions: first, that every connected component of $G - \{v\}$ contains at most a single DFS-tree-child of v, and second, that there cannot be a connected component of $G - \{v\}$ that does not contain a DFS-tree-child of v.

For the first statement, we note that if there are $> 1$ DFS-tree-children (call them $u_i$, where the $u_i$ are ordered by time of discovery) of v in some connected component $S$ of $G - \{v\}$, then $u_1$ and $u_i$ (where $i > 1$) have some path in $S$. This path did not go through v in the original graph $G$, since it is retained in $G - \{v\}$. Since explore(v) would have explored $u_1$ first, $u_i$ would have been explored as a descendant of $u_1$, by an aforementioned path in $G$ that does not pass through v. Thus $u_i$ would not be a DFS-tree-child of v. Therefore each connected component of $G - \{v\}$ contains at most 1 DFS-tree-child of v.

For the second statement, we assume for the sake of contradiction that there is a connected component T of $G - \{v\}$ that does not contain a DFS-tree-child of v. Since T was a connected graph, all vertices in T were connected by some path to v. explore(v) explores every vertex in connected to v. Let w be the first such vertex explored by the algorithm. Since everything in T is connected to w, w is the ancestor of every other vertex in T, and each such vertex must have a path to w that does not pass through v in G. Next, we prove by contradiction that w is a DFS-tree-child of v. If not, there must be some distinct ancestor of w (which we call u) that was a DFS-tree-child of v, but which is not in T. But since w and u are connected by a path that does not pass through v in G, T must contain u, and u must have been visited first by the algorithm, so u and w are not distinct (and we reach a contradiction). Therefore w is a DFS-tree-child of v in T.

Since every DFS-tree-child of v exists in $G - \{v\}$ and each connected component of $G - \{v\}$ contains exactly one DFS-tree-child of v, the number of children of v in the DFS tree is the same as the number of connected components of $G - \{v\}$.

**Question 4** (Pre- and Post-Order Correspondence, 25 points). Suppose that for a graph G you are given the list of the pre-order numbers of its n vertices and the list of the post-order numbers of its n vertices. Show that it is possible to determine which pre-order numbers correspond to which post-order numbers (in terms of being the pre- and post-order numbers for the same vertex).

**Solution 4.** Let us denote the two given lists of pre- and post-order numbers by A and B, respectively. We are given a graph G with n vertices and we know $|A| = |B| = n$. We create another list $P$ of 2n with both elements of A and B, i.e. a list or an array that contains 1, 2, ..., 2n. We define two subroutines: isPreOrder(x) and isPostOrder(x). The former returns True if $x \in A$ and False otherwise. isPostOrder(x) returns True if $x \in B$.

The key data structure to solve this problem will be to use a stack. Let us denote the stack by S. The idea is that the stack S will maintain the list of explore calls currently being executed. So, we traverse the list $P$ and $\forall x \in P$, if isPreOrder(x) is True, we push $x$ to S. Here, the pre-order number $x$ denotes the start of a new recursive call to explore in the program. And when an element is a post-order element, i.e. isPostOrder(x) is True, then it means we have finished a recursive call of the most recently started and
still active subroutine. Thus, we pair the post-order element $x$ with the top of the stack, say $t$ (most recent still active subroutine). We store such $(t, x)$ in a list $Pairs$ and return this list.

Here is the pseudocode for the algorithm:

```plaintext
procedure getPairs(P):
    $S \leftarrow []$ // Stack
    $Pairs \leftarrow []$
    for $x$ in $P$ do
        if isPreOrder($x$) then
            push $x$ to $S$
        end if
        if isPostOrder($x$) then
            $t \leftarrow S.pop()$
            Append $(t, x)$ to $Pairs$
        end if
    end for
    return $Pairs$
end procedure
```

Runtime Complexity: $O(n)$

We just have to scan through each element in $P$ which takes linear time ($O(n)$). All stack operations of pushing and popping elements take constant time. Thus, the time complexity for the algorithm is $O(n)$

**Question 5** (Extra credit, 1 point). *Approximately how much time did you spend working on this homework?*