Homework 2

● Graded

Student

John Ezra See

Total Points

98 / 100 pts

Question 1

Search in a Rotated Sorted Array

15 / 15 pts

✔ **– 0 pts** Correct

– 2.5 pts Incorrect logic implementation - minute errors / Insufficient algo explanation

– 5 pts Incorrect logic and missing conditions - Partial credit

– 7.5 pts Partial credit

– 10 pts Partial credit

– 12.5 pts Partial credit

– 15 pts Partial credit

Question 2

Heapify-Up Example

10 / 10 pts

✔ **– 0 pts** Correct

– 2.5 pts Partial Grade

– 5 pts Partial Grade

– 7.5 pts Partial Grade

– 10 pts Incorrect

Question 3

Ex 3.1

10 / 10 pts

✔ **– 0 pts** Correct

– 2.5 pts Partial Credit

– 5 pts Partial Credit

– 7.5 pts Partial Credit

– 10 pts Incorrect

Question 4

Ex 3.2

15 / 15 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** Partial Grade / reconstruction missing / TC
- **– 5 pts** Partial Grade
- **– 7.5 pts** Partial Grade
- **– 10 pts** Partial Grade
- **– 12.5 pts** Incorrect - Partial grade
- **– 15 pts** Incorrect

Question 5

DFS with edges

3 / 5 pts

- ✔ **– 0 pts** Correct
- **– 1 pt** Partially Correct (or) Missing condition checks (or) missing parent tracking (or) missing Data structures mentioning
- ✔ **– 2 pts** Patial Grading
- **– 3 pts** Patial Grading
- **– 4 pts** Patial Grading
- **– 5 pts** Incorrect / unattempted

Question 6

Ex 3.5

15 / 15 pts

- ✔ **– 0 pts** Correct
- **– 2.5 pts** proof logic / Parital Grade
- **– 5 pts** Induction 2 missing / Parital Grade
- **– 7.5 pts** Induction 1 missing / Parital Grade
- **– 10 pts** Hypothesis / Parital Grade
- **– 12.5 pts** Base Case / Parital Grade
- **– 15 pts** Incorrect

Question 7

Ex 3.7 **15** / 15 pts

✔ **– 0 pts** Correct

– 2.5 pts Partial Grade - Claim true but insufficient explanation

– 5 pts Partial Grade

– 7.5 pts Partial Grade

– 10 pts Partial Grade

– 12.5 pts Partial Grade

– 15 pts Incorrect / unattempted

Question 8

Ex 3.10 **15** / 15 pts

✔ **– 0 pts** Correct

– 2.5 pts Minute errors but correct solution approach and logic

– 5 pts Partial Grade - counts all nodes instead of just w

– 7.5 pts Partial Grade - Other than BFS

– 10 pts Partial Grade

– 12.5 pts Partial Grade

– 15 pts Partial Grade

Question assigned to the following page: 1

John Ezra Sec CS 401 HW2

① We can achieve a O(log n) solution by utilizing a binary search algorithm.

```
left, right = 0, len(A) - 1

while left ≤ right:
    mid = (left + right) // 2
    if A[mid] == x:
        return mid
    if A[left] ≤ A[mid]:
        if A[left] ≤ x < A[mid]:
            right = mid - 1
        else:
            left = mid + 1
    else:
        if A[mid] < x ≤ A[right]:
            left = mid + 1
        else:
            right = mid - 1
return -1
```

Explanation: By picking any index, we can narrow down the search space by taking advantage of the sorted nature.
So, if left is sorted, we can check if x is within the left side. If it is, then narrow down the search space to the left side. If not, then x must be on the right side.
* In all cases, we are shrinking the bounds and we get closer to x. We will eventually find it if the mid pointer points to x. Otherwise, while loop terminates and returns -1.

Question assigned to the following page: 1

Time Complexity Analysis:

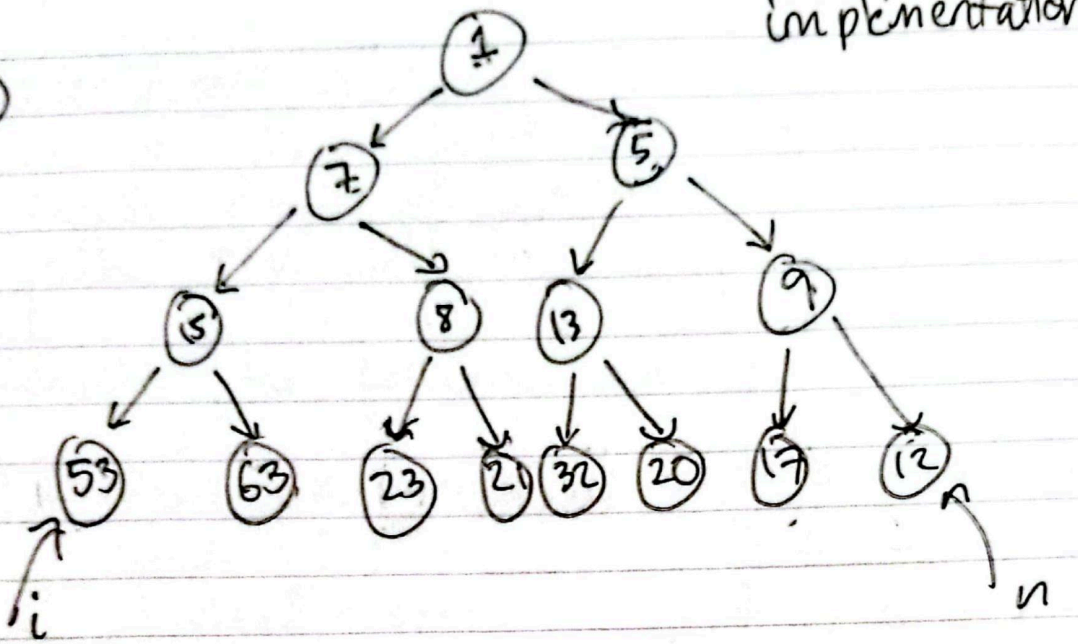
* The time complexity of this algorithm is O(log n).

For every iteration, we are cutting the possible search space by 2. Given, the line mid = (left + right)//2, we either do left = mid + 1 or right = mid - 1, in both cases, the bounds are divided by two. We will terminate when right > low, thus, it's O(log n).

Question assigned to the following page: 2

min-heap implementation

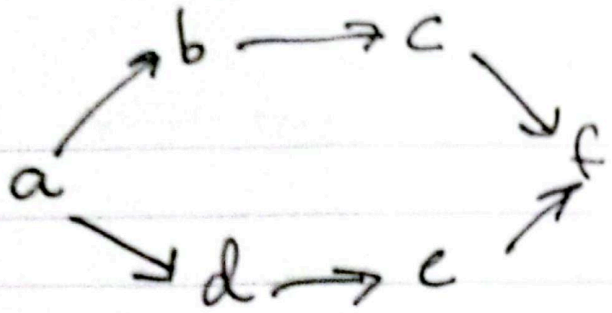
2.



At i, the element is 53.
Deleting that element will result in the execution of calls to Heapify-up. The element at position n will be moved to position i. The parent of i will then be bigger than the element i which breaks the heap. To fix that, 12 will be heapify'd up because 15 > 12 to restore the heap structure.

Question assigned to the following page: 3

(3)



a b c d e f
a b d c e f
a b d e c f
a d b c e f
a d b e c f
a d e b c f

= There are 6 topological orderings

Question assigned to the following page: 4

(4.) To detect a cycle in an undirected graph, we can do a depth-first search approach while keeping track of our visited node and current path. This will be O(m+n) in TC.

Consider the following code/algorithm:

```
def find_cycle(graph):
    path = []
    visited = set()
    cycle_found = [False]

    def dfs(node, prev):
        visited.add(node)
        path.append(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                if dfs(neighbor, node): return True
            elif neighbor != prev:
                cycle_found[0] = true
                return True
        path.pop()
        return False
    for node in graph:
        if node in visited: continue
        if dfs(node, None):
            return path if cycle_found[0] else []
    return []
```

Question assigned to the following page: 4

Explanation: We are essentially doing a dfs search while keeping track of our current path. Since the graph is undirected, we detect cycles if a visited node AND that node isn't the previous node is encountered.
In the case of a dis-joint graph, we will just have a for loop that goes through unvisited nodes. However, once a cycle is encountered, we output the path that represents the path that led to a cycle.

Time Complexity Analysis:
Since we are utilizing a set of visited nodes, we prevent repetitive traversals. We are exploring nodes / edges once only. Thus, the time complexity will be linear:
$O(m+n)$ where m = edges, n = nodes

Question assigned to the following page: 5

(5.)
```
function dfs(graph):
    visited = an empty set # data structure set
                           # this is used to prevent cycles.

    function go(node): # the helper function
        visited.add(node)
        for each neighbor u in graph[node]:
            if u is not in visited:
                print "Edge from {node} => {u}"
                go(u)
    for node in graph: # For disjoint nodes.
        if node is already visited:
            continue
        otherwise,
            go(node)
```

Question assigned to the following page: 6

(6) Proof by Induction:

We will prove that $N_2 = L-1$ where N_2 is the number of nodes with two children and L is the number of leaves.

- Basis step: Consider a tree with only the root. Then, $N_2 = 0$ and $L = 1$ (the root is a leaf). So, $N_2 = L-1 \Rightarrow 0 = 0$ ✓

- Inductive Hypothesis: Let's assume that $P(k)$ is true for $k \geq 1$ which is the # of nodes.

- Inductive Step: Show that $P(k+1)$ is true such that: Adding an extra node should hold the statement $N_2 = L-1$ true.

- Complete proof:

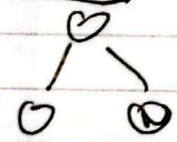
By adding a node, there are two possibilities:

Case 1: the new node is an only child;

thus, N_2 remains the same and L also remains the same (the parent stops becoming a leaf, and the child becomes the leaf).

$\therefore N_2 + 0 = L - 1 + 0$ ✓

Case 2: the new node has a sibling,



thus, N_2 increases by one and the new node is also a leaf. So, the statement becomes $\Rightarrow N_2 + 1 = L - 1 + 1$

$= N_2 = L - 1$ ✓

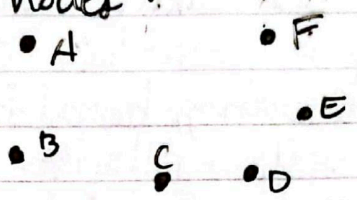
Conclusion: In either case, the statement holds true. By induction, we proved $P(n)$ to be true for all $n > 0$. QED

Question assigned to the following page: 7

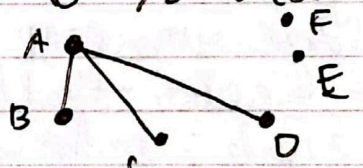
(7.) True.

Proof: In a graph G of n even nodes, we want to prove that they are connected since each node has a degree of $n/2$ to other nodes.

Consider a graph of n nodes:
Let's say it's $n=6$,



We need all nodes to have a degree $\geq n/2$,
Let's connect node A to $n/2$ nodes $\{A, B, C, D\}$:

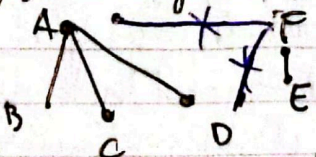


We get a connected graph of $(n/2 + 1)$ nodes after connecting one node.

Thus, we have

$$n - (n/2 + 1) = \frac{n}{2} - 1$$ nodes that have no edges yet. In this case, $\{E, F\}$ are unconnected.

Now, let's form $n/2$ edges from node F.



$\Rightarrow$ which is connected

There are simply not enough nodes to make the graph unconnected. Mathematically, $(\frac{n}{2}+1)$ nodes are initially connected. Forming $\frac{n}{2}$ edges from an unconnected node results to $(\frac{n}{2}+1)$ connected nodes. We only had $(\frac{n}{2}-1)$ nodes to choose from.

Thus, $\frac{n}{2}+1 \neq \frac{n}{2}-1 \Rightarrow 0 \neq 2$, which is false.

Hence, G must be connected.

Question assigned to the following page: 8

(8.) Given an undirected graph and two nodes v and w in G, give an algorithm that computes the number of shortest paths in G.

Algorithm: We can utilize the BFS algorithm which has a runtime of O(M+n) because edges and nodes are visited only once.

As we're traversing, keep track of distances and count maps. We will use a queue to implement a standard BFS implementation.

```
def solve(graph, v, w):
    # Let's initialize our map for counting and distances
    distances = {node: -float('inf') for node in graph}
    path_count = {node: 0 for node in graph}
    distances[v] = 0
    path_counts[v] = 1

    queue = deque([v]) # our queue
    while queue:
        node = q.popleft()
        if node == w: break # we found the shortest already
        for neighbor in graph[node]:
            if distances[neighbor] > (distances[node] + 1):
                distances[neighbor] = distances[node] + 1
                path_count[neighbor] = path_count[node]
                queue.append(neighbor)
            elif distances[neighbor] == distances[node]+1
                path_count[neighbor] += path_count[node]
    return path_count[w]
```

Question assigned to the following page: 8

Algorithm high level explanation:

It's a modified BFS in the sense that we are also storing a count map (path_count). It's similar to dynamic programming because the number of paths to a node is calculated by the precomputed values of the current's path count.

* On the other hand, the distance map is similar to Dijkstra's algorithm because we want the shortest path. Each edge weight is a value of 1 which is why we didn't need a priority queue which is standard for Dijkstra's algorithm.

By utilizing only a queue and using dp techniques, we're able to count the number of shortest v-w paths in G in $O(n+m)$ time.