21S-COMSCI180-1 Midterm

SANCHIT AGARWAL

TOTAL POINTS

100 / 100

QUESTION 1

Q1 20 pts

1.1 a 5 / 5

✓ **- 0 pts Correct**

- 5 pts Incorrect or missing,

1.2 b 5 / 5

✓ **- 0 pts Correct**

- 5 pts Incorrect

1.3 c 5 / 5

✓ **- 0 pts Correct**

- 5 pts Incorrect

1.4 d 5 / 5

✓ **- 0 pts Correct**

- 5 pts Incorrect or missing

QUESTION 2

Q2 20 pts

2.1 a 10 / 10

✓ **- 0 pts Correct**

- 2 pts Accomplish insert within O(logN) time instead of O(logK) time

- 5 pts Do not accomplish insert within O(logK) Time.

- 3 pts Accomplish findKmean within O(logN) timeinstead of O(logK) time

- 5 pts Do not accomplish findKmin within O(logK) Time.

- 2 pts No runtime analysis.

2.2 b 10 / 10

✓ **- 0 pts Correct**

- 3 pts Do not accomplish insert within O(logN) Time.

- 3 pts Do not accomplish findKmin within O(logN) Time.

- 4 pts Do not accomplish pop within O(logN) Time.

Click here to replace this description.

- 0 pts Click here to replace this description.

QUESTION 3

Q3 20 pts

3.1 a 10 / 10

✓ **- 0 pts Correct**

- 10 pts Missing

- 3 pts No proof by contradiction / incomplete proof

- 5 pts Inadequate proof, no contradiction

3.2 b 10 / 10

✓ **- 0 pts Correct**

- 3 pts Incorrect/Insufficient application of Part A proof

- 5 pts Insufficient justification

- 7 pts Click here to replace this description.

- 10 pts Click here to replace this description.

- 2 pts Click here to replace this description.

QUESTION 4

4 Q4 20 / 20

✓ **- 0 pts Correct**

- 10 pts Click here to replace this description.

- 15 pts Click here to replace this description.

- 5 pts Click here to replace this description.

QUESTION 5

20 pts

5.1 a 10 / 10

✓ **- 0 pts Correct**

- 1 pts Misrepresentation of graph

- 1 pts Contradiction step not clear

- 2 pts Missing parts

- 8 pts No proof

5.2 b 10 / 10

✓ **- 0 pts Correct**

- 1 pts Correct intuition

- 3 pts Merging cycles not mentioned.

- 4 pts Reasonable intuition but not intended algorithm

- 8 pts Missing algorithm proof

- 10 pts No answer

CS 180

MIDTERM

Sanchit Agarwal

UID: 105389151

Q1

a) False, $2^n \neq \Theta(3^n)$ as

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \left(\frac{2}{3}\right)^{\infty} = 0 \quad \text{— (1)}$$

for $2^n = \Theta(3^n)$, $\lim_{n\to\infty} \frac{2^n}{3^n} = c$ where $0 < c < \infty$, which is untrue according to (1).

b) True

c) False, DAG can be disconnected

d) True,

If we have a directed connected graph, we can find a tree with some edge deletions and for a tree there exist atleast 2 nodes with degree 1 (as proved in the lecture)

$\Rightarrow$ These 2 nodes can be removed without breaking connectivity.

1.1 a **5 / 5**

✓ **- 0 pts** **Correct**

- 5 pts Incorrect or missing,

Q1

a) False, $2^n \neq \Theta(3^n)$ as

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \left(\frac{2}{3}\right)^{\infty} = 0 \quad \text{— (1)}$$

for $2^n = \Theta(3^n)$, $\lim_{n\to\infty} \frac{2^n}{3^n} = c$ where $0 < c < \infty$, which is untrue according to (1).

b) True

c) False, DAG can be disconnected

d) True,

If we have a directed connected graph, we can find a tree with some edge deletions and for a tree there exist atleast 2 nodes with degree 1 (as proved in the lecture)

$\Rightarrow$ These 2 nodes can be removed without breaking connectivity.

1.2 b **5 / 5**

✓ **- 0 pts** **Correct**

- 5 pts Incorrect

Q1

a) False, $2^n \neq \Theta(3^n)$ as

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \left(\frac{2}{3}\right)^{\infty} = 0 \quad \text{— (1)}$$

for $2^n = \Theta(3^n)$, $\lim_{n\to\infty} \frac{2^n}{3^n} = c$ where $0 < c < \infty$, which is untrue according to (1).

b) True

c) False, DAG can be disconnected

d) True,

If we have a directed connected graph, we can find a tree with some edge deletions and for a tree there exist atleast 2 nodes with degree 1 (as proved in the lecture)

$\Rightarrow$ These 2 nodes can be removed without breaking connectivity.

1.3 C **5 / 5**

✓ **- 0 pts** **Correct**

- 5 pts Incorrect

Q1

a) False, $2^n \neq \Theta(3^n)$ as

$$\lim_{n\to\infty} \frac{2^n}{3^n} = \left(\frac{2}{3}\right)^{\infty} = 0 \quad \text{— (1)}$$

for $2^n = \Theta(3^n)$, $\lim_{n\to\infty} \frac{2^n}{3^n} = c$ where $0 < c < \infty$, which is untrue according to (1).

b) True

c) False, DAG can be disconnected

d) True,

If we have a directed connected graph, we can find a tree with some edge deletions and for a tree there exist atleast 2 nodes with degree 1 (as proved in the lecture)

$\Rightarrow$ These 2 nodes can be removed without breaking connectivity.

1.4 d **5 / 5**

✓ **- 0 pts** **Correct**

- 5 pts Incorrect or missing

Q2

a)

→ Given:

① Supported Operations →

↳ push : insert a new number

↳ find_Kmin : return the k^{th} smallest element without removing

② Complexity constraint → $O(\log K)$

→ Assumption:

→ All the input numbers are unique (freedom given by TA on Piazza)

→ K is fixed at the start

→ Proposed Data Structure: Max heap

Max heap is similar to a minheap, the only difference is that for every element v, at a node i, the element w at i's parent satisfies $key(w) \leq key(v)$.

⇒ Max heap is a heap with key value properties being the reverse of a min heap.

For our purposes, we will mantain a max heap of maximum k elements and a counter which keeps track of how many elements are there in the heap.

Initial: max_heap = array of atmost k elements

counter = no. of elements in max_heap

Push (Algorithm): ← input = element

```
if (counter >= k):
    if (element < first element of max_heap):
        first element of max_heap = element -(1)
        i = 0
        while (i has a child):  -(2)
            j ← i's largest child
            if A[i] ≥ A[j]:
                done
            else:
                exchange A[i] & A[j]
                i = j
    else:
        discard element
else:
    append element to max_heap  -(3)
    i = counter
    while (i ≠ 0 and A[⌊i/2⌋] < A[i])  -(4)
        exchange (A[⌊i/2⌋], A[i])
        i ← ⌊i/2⌋
    counter++
```

Complexity Analysis of Push: $O(\log K)$

The complexity of Push is $O(\log K)$ as

with every iteration of ② or ④ my control variable i is reducing to half of its value.

Since in the worst case scenario, $i = K$

⇒ it will take $\log K$ iterations to end the loop

Complexity of ① = $O(1)$

Complexity of ② = $O(\log K)$

Complexity of ③ = $O(1)$

Complexity of ④ = $O(\log K)$

⇒ Complexity = $O(\log K)$

find_Kmin : Algo

Return the first element of the heap

→ Since we are maintaining a max-heap of k elements, selecting the largest element would be $O(1)$. Now as our array is of length K ⇒ the K^{th} smallest element in the set of all numbers pushed is the largest element among the K elements that stay in the array (max-heap)

Complexity : $O(1)$ → since we are just accesing the first element.

2.1 a **10 / 10**

✓ **- 0 pts** **Correct**

- 2 pts Accomplish insert within O(logN) time instead of O(logK) time

- 5 pts Do not accomplish insert within O(logK) Time.

- 3 pts Accomplish findKmean within O(logN) timeinstead of O(logK) time

- 5 pts Do not accomplish findKmin within O(logK) Time.

- 2 pts No runtime analysis.

b)

Given:

→ Supported Operations:

↳ push: insert a new number

↳ find_Kmin: return the k^{th} smallest element without removing

↳ pop: return & remove the k^{th} smallest element.

→ Complexity Constraint:

↳ O(log N) (where N is the no. of elements in the current set)

→ Proposed Data structure: Min heap & Max heap

The idea here is to maintain 2 data structures a min heap & a max heap. Max heap will add the k smallest elements & Min heap will hold the rest of the N-k elements.

Like before we will maintain a counter to track how many values are there in the heap.

→ Assumption:

→ All the input numbers are unique
(freedom given by TA on Piazza)

→ k is fixed at the start

→ Proposed Algorithms:

Initial: max_heap = array of atmost k elements

counter = no. of elements in max_heap

min_heap = array of atmost n-k elements

Push (Alg):

```
if (counter >= k):
    if (element < first element of max_heap):
        first element of max_heap = element — (1)
        i = 0
        while (i has a child): — (2)
            j ← i's largest child
            if A[i] ≥ A[j]:
                done
            else:
                exchange A[i] & A[j]
                i = j
    else:
        Push element in min_heap: — (3)
```

```
        append element to min_heap
        i = counter
        while (i ≠ 0 and A[⌊i/2⌋] > A[i])
            exchange A[⌊i/2⌋], A[i]
            i ← ⌊i/2⌋
else:
    append element to max_heap          — (4)
    i = counter
    while (i ≠ 0 and A[⌊i/2⌋] < A[i])   — (5)
        exchange (A[⌊i/2⌋], A[i])
        i ← ⌊i/2⌋
    counter++
```

Complexity:

- From part (a),
  - → Total complexity of ①, ②, ④ & ⑤ $= O(\log K)$
  - → Complexity of ③ $= O(\log(N-K))$ (N-K elements, control variable halved at each iteration)

⇒ complexity $= O(\log(K) + \log(N-K)) = O(\log N)$

find_Kmin: Algo

The algorithm for finding the k^{th} smallest element is the same as part @. An O(1) complexity Algorithm where we just return the 1st element of Max_heap

pop (Algo)

```
result = max_heap[0]
max_heap[0] = min_heap[0]
min_heap[0] = last element of min_heap
remove last element of min heap
i = 1
while (i has a child)
    j ← i's smallest child
    if A[i] ≤ A[j]:
        break;
    else
        exchange A[i], A[j]
        i = j
return result
```

↳ (Assuming that we need to return the k^{th} smallest element before popping anything)

Complexity Analysis =

$O(\log(N-K))$ to heapify the min heap after removal of root.

$O(\log(N-k)) = O(\log(N))$

2.2 b 10 / 10

✓ **- 0 pts** **Correct**

- 3 pts Do not accomplish insert within O(logN) Time.

- 3 pts Do not accomplish findKmin within O(logN) Time.

- 4 pts Do not accomplish pop within O(logN) Time.

Click here to replace this description.

- 0 pts Click here to replace this description.

Q3

(a)

Given:

m_1 preference: $w_1 > w_2 >$ all other women

m_2 preference: $w_2 > w_1 >$ all other women

w_1 preference: $m_2 > m_1 >$ all other men

w_2 preference: $m_1 > m_2 >$ all other men.

To proof:

In all stable matchings we either have $\{(m_1, w_2), (m_2, w_1)\}$ or $\{(m_1, w_1), (m_2, w_2)\}$

Proof (By contradiction):

Let there exist a stable matching where one of the men (say m_1) is not paired with w_1 or w_2

$\Rightarrow$ I have the pair (m_1, w_k) in my matching.

let one of the women (say w_1) be paired with a man other than m_2 (say m_p)

Now

m_1 prefers w_1 over w_k &

w_1 prefers m_1 over m_p [According to preference list]

$\Rightarrow$ There is an instability

$\Rightarrow$ our assumption was wrong

$\Rightarrow$ $\nexists$ a stable matching in with either

m_1 or m_2 is paired with any other women than w_1 or w_2.

Hence Proved

b)

Given:

→ n is even

→ To construct an instance of stable matching problem with $2^{n/2}$ stable matching.

① Break n men into n/2 pairs of 2 men each and give each pair a number incrementally starting from 1

② Break n women into n/2 pairs of 2 women each and give each pair a number incrementally starting from 1

③ For each pair i of men & women mesh their preferences as in part (a)

⇒ $m_i = w_i > w_j >$ all other women

$m_j = w_j > w_i >$ all other women

$w_i = m_j > m_i >$ all other men

$w_j = m_j > m_i >$ all other men

3.1 a **10 / 10**

✓ **- 0 pts** **Correct**

- 10 pts Missing

- 3 pts No proof by contradiction / incomplete proof

- 5 pts Inadequate proof, no contradiction

m_1 or m_2 is paired with any other women than w_1 or w_2.

Hence Proved

b)

Given:

→ n is even

→ To construct an instance of stable matching problem with $2^{n/2}$ stable matching.

① Break n men into n/2 pairs of 2 men each and give each pair a number incrementally starting from 1

② Break n women into n/2 pairs of 2 women each and give each pair a number incrementally starting from 1

③ For each pair i of men & women mesh their preferences as in part (a)

⇒ $m_i = w_i > w_j >$ all other women

$m_j = w_j > w_i >$ all other women

$w_i = m_j > m_i >$ all other men

$w_j = m_j > m_i >$ all other men

Thus, this way for each pair of men & women there can be 2 stable matchings as shown in part (a). Since we will have a stable matching for each pair i of 2 men & 2 women, this will apply for all pairs together too as in every pair either the man or the women are paired with their first preference which wouldn't lead them to prefer someone else than their current partner $\Rightarrow$ instability not possible.

$\Rightarrow$ 2 stable matchings for each pair i

Total $n/2$ pairs.

So I can have

$2 \times 2 \times 2 \times \ldots$ $n/2$ times combinations of stable matching

$= 2^{n/2}$

$\therefore$ I will atleast have $2^{n/2}$ matchings

Hence Proved

3.2 b 10 / 10

✓ **- 0 pts** **Correct**

- 3 pts Incorrect/Insufficient application of Part A proof

- 5 pts Insufficient justification

- 7 pts Click here to replace this description.

- 10 pts Click here to replace this description.

- 2 pts Click here to replace this description.

Q4

Given:

No. of switches = n

complexity constraint = $O(n \log n)$

Invert a switch: Turn it off if on & on if off

Assuming indiana jones can find 1st closed door in constant time

Algorithm:

```
start = switch 1
end = switch n
divider = ⌊(start + end)/2⌋
i = first closed door
while (i != n):
    if start == end:
        remove the switch that start equals to
                from set of n switches
        invert the switch that start equals to
        start = 1
        end = n-1 (as 1 switch is removed)
        i = first closed door (door i is now open, move to next closed door)
    invert all switches from start to divider
    if door i open:
        end = divider
        invert all switches from start to end
    else:
        start = divider
    divider = ⌊(start + end)/2⌋
```

Complexity Analysis

For opening every door, My algorithm does a binary search on all the switches to find the switch that coresponds to that door. In the above algorithm with every iteration the range is halfed and so after log N iterations the range will turn to 1 and then the algorithm will move to the next door.

Therefore for every door, there are logN iteration when in each iteration one trial is done where Indiana Jones walks from control room to doors.

$\Rightarrow$ For N doors $\longrightarrow$ complexity = $O(N \log N)$

Coorectness Argument

The above algorithm is correct as I am localizing the coorect switch using binary search. Since the range is halfed with each iteration, it is sure that the loop will end for that door as range will get smaller with each iteration and will result in the coorect switch for that door.

4 Q4 20 / 20

✓ **- 0 pts** **Correct**

- 10 pts Click here to replace this description.

- 15 pts Click here to replace this description.

- 5 pts Click here to replace this description.

Q5

a) To Proof: No matter how many teleporters are placed, you will always reach t.

Given: ① We may only walk eastwards
② None of the endpoints are on the same location

It is hinted in the problem to use segments as nodes

Definition of segment: line between any two adjacent endpoints

Now at the end of every segment you get teleported to another segment and at the start of every segment you get teleported into that segment.
However the first segment just has a teleportation out & the last segment just has a teleportion in.

Now for every segment,
we just walk on the segment once due

to 2 constraints

① we can only walk eastwards

② endpoints can't overlap.

So for a endpoint when I teleport from it I walk on the segment before that & when I teleport to it I walk on the segment after that.

This applies as we can only walk eastwards.

lemma: We can only walk over a segment once, no matter how many teleporters are there

Pf (By contradiction):

Let's assume we walk on a segment twice

⇒ that the first point to which we teleport into is an endpoint of more than one teleporters.

However it is specified in the problem that we cannot have overlaping endpoints

⇒ our assumption is wrong

⇒ we can only walk on a segment once.

Now since, we cannot walk a segment twice & can't have a redundant endpoint

⇒ That if segments are pictured as nodes, each node will have an indegree of 1 & outdegree of 0 except for the start node whose indegree would be 0 & the end node whose outdegree would be 0.

Now

Lemma: I cannot have a cycle in my graph

If I have a cycle I won't have any any way to reach from the start node to any other node in the cycle as each node involved will have an edge from it involved in the cycle and an edge to it involved in the cycle. Since I can only have one incoming edge & one outgoing edge, I won't be able to reach the cycle anyway from the start point unless start node itself is in the cycle which isn't possible.

Now since my graph has no cycle and just has one incoming and outgoing edge eventually as we keep walking eastwards we will keep on being teleported again & again. since there is no cycle possible and we can only walk on a segment once, eventually we will exaust all the segments and reach the endpoint or we might reach the endpoint earlier.

Hence Proved

5.1 a **10 / 10**

✓ **- 0 pts** **Correct**

- 1 pts Misrepresentation of graph

- 1 pts Contradiction step not clear

- 2 pts Missing parts

- 8 pts No proof

b)

Given: To add k teleporters

Background:

Since each segment can be walked upon only once & there cannot be any overlapping edges ⇒ I can maximum use a teleporter twice

↳ one for going from it

↳ one for coming from it.

If I go or come more than once using the same teleporter I will be walking on a segment twice which isn't possible.

Thus for N teleporters, ∃ almost 2N teleportations possible.

Now for any teleporter to maximize its teleportations we must have an end point between the endpoints of the teleporter such that the corresponding endpoint of this point is not in between the endpoints of the teleporter.

eg:-

Having the situation as described above would allow a pair of teleportors to be used twice each and hence would contribute towards maximizing teleportations.

Assumption:- Each teleportors first endpoint is listed and the list of these endpoints is sorted

→ Looking up the corresponding endpoint of a teleportor is constant time

Let endpoint(·) be the method to find the corresponding endpoint of a teleportor given one endpoint.

Algorithm:

```
start_points = sorted list of teleportor start points

i = 0
k = K

while i != n:
    start = start_point[i]
    end = endpoint(start_points[i])
    if i < n-1:
        next_start = start_point[i+1]
```

```
next_end = endpoint(next_start)

if i > 0:
    prev_st  -= start_point[i - 1]
    prev_end = endpoint[prev_start]

if i = 0:
    if start < next_start < end and
       end < next_end:
          i++; continue;
    else:
       add a new teleporter s.t
       start < start_point < end and
       end < endpoint(start_point)
       k--
if i = n-1:
    if  prev_start < start and
       start < prev_end < end:
          i++; continue;
    else:
       add a new teleporter s.t
       start_point < start and
       start < endpoint(start_point) < end:
       k--
```

if $i > 0$ and $i < n-1$

if(start < next_start < end and
end < next_end) or
(prev_start < start and
start < prev_end < end):
i++; continue;
else
add a new teleportor s.t
start < start_point < end and
end < endpoint(start point)
k--

For remaining k!
make meshed pairs of 2 teleportors
one after the other such that
for any pair (s_i, s_j), (k_i, k_j)
either $k_i < s_i < k_j$ and $s_j > k_j$ or
$s_i < k_i < s_j$ and $k_j > s_j$
If odd no. of k remaining add a
single bridge at the end.
k--;

Complexity Analysis

Assuming sorted list of endpoints
if not → complexity = $O(N \log N)$

For each start_point I compare with previous/next teleportors endpoints and then I either add another teleportor or not. As it is given that finding corresponding endpoint is $O(1)$ and we know that comparision is $O(1)$
⇒ Time complexity for N start point = $O(N)$

Ans:- If sorted = $O(N)$
else = $O(N \log N)$.

5.2 b **10 / 10**

✓ **- 0 pts Correct**

- 1 pts Correct intuition

- 3 pts Merging cycles not mentioned.

- 4 pts Reasonable intuition but not intended algorithm

- 8 pts Missing algorithm proof

- 10 pts No answer