Ethan Wong
CS180
4 May 2021

Midterm Exam

1. a) False
   b) True
   c) False
   d) True

2. a) A data structure that could be used to perform these operations would be one that consists of a max heap and a min heap. The max heap would have a maximum of “k” total nodes. It would hold the “k” many smallest elements. The k^{th} smallest would be at the top of the max heap.

   Pseudocode for algorithms:

```
push(int x):
  int y = value from top of max heap
  if x < y:
    delete top of max heap
    add x to max heap, heapify max heap
```

   The time complexity of push(int x) is O(log k) because you have to heapify the max heap, which contains k elements. The logic behind this algorithm is that if the value to be inserted is less than the k^{th} smallest value, then the k^{th} smallest value would then need to be updated. Therefore we would delete the top of the max heap and replace it with the new value. The new value would then replace the spot where the previous k^{th} smallest value was.

```
find_Kmin():
  return top value of max heap
```

   The time complexity of find_Kmin() would be O(1) because you are just returning the top value of the max heap. This algorithm is pretty simple. As explained above, the k^{th} smallest would be kept at the top of the max heap so that value simply needs to be returned to find the k^{th} smallest value.

b)
A data structure that could be used to perform these operations would be one that is similar to the one from part A. The data structure would consist of a max heap and a min heap. The max heap would have a maximum of “k” total nodes. It would hold the “k” many smallest elements. The k^{th} smallest would be at the top of the max heap. The min heap would hold all the other data that was not included within the max heap. In other words, it would hold any elements that were not a member of the “k” many smallest elements.

Pseudocode for algorithms:

```
push(int x):
  int y = value from top of max heap
  if x < y:
    move top of max heap into min heap, heapify min heap
    add x to max heap, heapify max heap
  if x ≥ y:
     add x to min heap, heapify min heap
```

The time complexity of this algorithm is O(log n) because of the heapify of the min heap. The min heap contains more than k elements. This is similar to the push from part A but with one added component because of the min heap. If the value to be inserted is greater than the k^{th} smallest value, then there is no need to worry about it joining the max heap. We can just add it into the min heap instead and heapify. Otherwise the algorithm is the same as in part A.

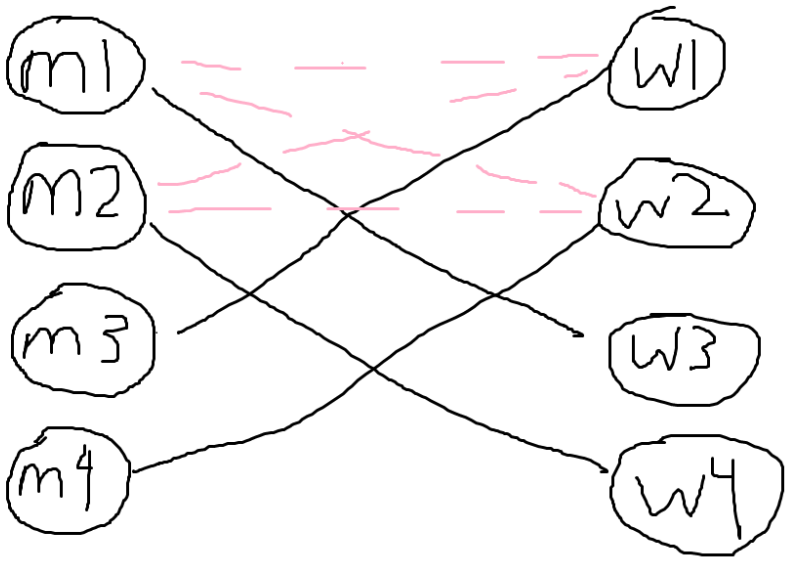
```
pop():
  y = top of the max heap
  delete the top of the max heap
  replace the now-empty top of the max heap with the top of the min heap
  heapify the min heap
  return y
```

The time complexity of this algorithm is O(log n) because of the heapify of the min heap. The min heap contains more than k elements. When we need to pop the k^{th} smallest value, we need to save its value in a variable (y) before we delete it. Once we delete the k^{th} smallest value, then we need to replace it because now there are k-1 nodes in the max heap when it needs to have k nodes. Thus we must replace it with the top of the min heap, because that is the next smallest value that is not already in the max heap. There is no need to heapify the max heap here as the newly inserted element is guaranteed to be the k^{th} smallest value. However, we need to heapify the min heap because we removed one of its elements. Finally, we can return y as pop needs to return the value that got deleted.

```
find_Kmin():
  return top of max heap
```

The time complexity of find_Kmin() would be O(1) because you are just returning the top value. This is the same as the find_Kmin() from part A because the structure of the max heap remains unchanged.

3. a) For a stable matching to occur, the pairings must either be (w1, m1), (w2, m2) or (w1, m2), (w2, m1). This can be proved by contradiction.
    - Assume that of any of these 4 (m1, m2, w1, w2), they are paired with someone that is out of their top two choices.
    - Say that the matchings were as follows: (m1, w3), (m2, w4), (m3, w1), (m4, w2).
    - This would mean that m1 proposed to w3 before he proposed to w1 or w2 and m2 proposed to w4 before he proposed to w1 or w2.
    - This is contradictory because according to the preference lists, m1 would have proposed to either w1 or w2 before proposing to w3, and either w1 or w2 would have accepted because m1 is in both of their top two preferences. The same logic applies for m2.
    - This shows that there would be a contradiction as a stable matching according to the given preference list must have the m1, m2, w1, and w2 all get matched with their top two choices. This is because they all have each other in their top two choices. When this occurs, they must all get matched with one another in order for there to be stability or else there is risk of instability: such as the case of (m1, w3). Because the example below has at least one instance of instability, the overall matching is unstable. This matching is unstable because m1 is not paired with his top two preferences and the women who have him in his top two preferences are not paired with him. Therefore we see that the pairings must either be (w1, m1), (w2, m2) or (w1, m2), (w2, m1).



Example Illustration of the matching (pink represents stable, black represents unstable)
(Sorry about the ugly drawing, I don't have a tablet to draw on so it's just MS Paint)

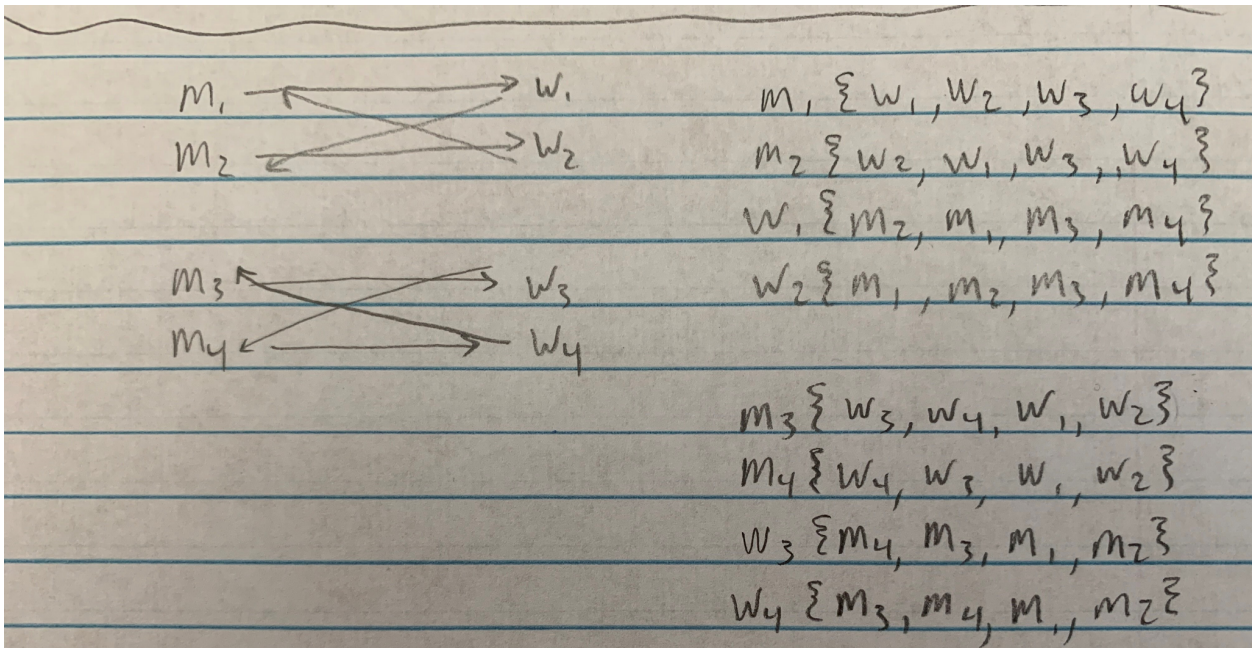
b)

This statement can be proven by constructing an instance of the stable matching problem. The core idea is to split the men and women into subgroups of two men and two women. For instance, m_i and m_{i+1} would be in a subgroup with w_i and w_{i+1}.

This means that for any even value of n, we will have (n / 2) of these subgroups. We want the preference lists to be constructed such that the men in the x^{th} subgroup prefer the women in the x^{th} subgroup over the women in the $(x+1)^{th}$ subgroup. The women in the x^{th} subgroup prefer the men in the x^{th} subgroup over the men in the $(x+1)^{th}$ subgroup. Essentially, the people within a given subgroup prefer each other over any other subgroup. With this information, we know that every engagement will have the men in the x^{th} subgroup get paired with the women of the x^{th} subgroup as long as $1 \leq x \leq (n/2)$.

A man in the x^{th} subgroup will never get engaged with a woman in another subgroup y. This is because if y < x, then the man will prefer his current partner over the women in the y^{th} subgroup. If y > x, then the woman will prefer her current partner over this other man. The only way for an "illegal" engagement to occur would be if both the man and women were already within the same subgroup, which doesn't make sense as either the man or woman will already be engaged to their ideal partner.

This shows that there will be at least $2^{n/2}$ stable matchings because every time you add a subgroup, you increase the number of stable matchings twofold. This can be seen in the example below.



Example of the preference list based on my above explanation

4. This problem can be solved using a variation of binary search and a loop.

Pseudocode for algorithm:

```
openDoors(int n):
  n = number of doors (passed in as parameter to the algorithm)
  for (int i = 1, i <= n; i++):
     L = 1;       //leftmost switch
     R = n;       //rightmost switch
     while (L ≤ R):
       mid = (L + R) / 2
       Flip all switches from L to mid**
       If the status*** of Door[i] changed:
          R = mid - 1
       If the status of Door[i] did not change:
          L = mid + 1
       If L == R:
          mark mid as the switch corresponding to Door[i]
          flip mid so that it is open, never touch this switch again — keep the door open
   Once for loop has concluded, all the doors should be open
```

** (exclude any switches that we know are associated with a door already, want to keep those doors open)
*** (“status” means whether a door is open or closed)

The reasoning behind this algorithm is to use binary search to locate the switch for the first door, second door, etc. Using the modified binary search to single out one door at a time minimizes confusion while still providing a decently quick method of escaping the Emperor’s Tomb. This algorithm runs in O(n logn) time because the overarching for loop will run for “n” iterations — this is the O(n). The binary search that is within the while loop is a slight variation of the one covered during lecture. Binary search has a time complexity of O(logn), which makes the overall complexity O(n logn).

5. a)

As long as there are no cycles that form between the teleporters, there will always be a path from S to T regardless of the amount of teleporters. There will always be one teleporter (X) that contains the point (X2) closest to T. In the case of the example, the B teleporter and point B2 were the closest to T. Since the person will always walk east, they will eventually walk to or get teleported to the starting point of the X teleporter that will guide them to T.

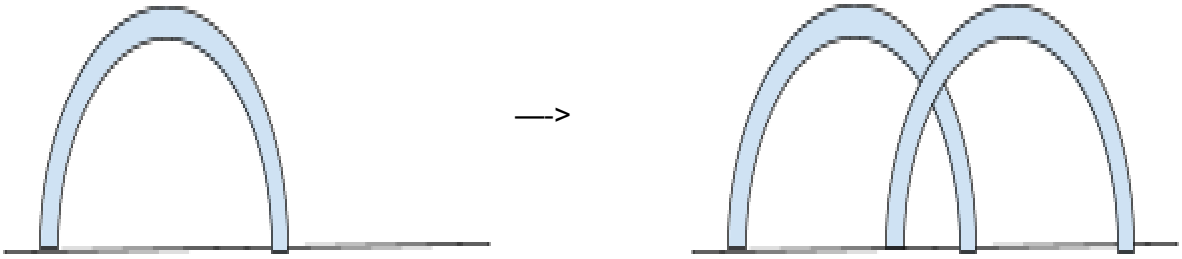
To show that you will always reach T no matter how many teleporters there are, we have to explore the idea of an unreachable point. An unreachable point is any point that can not be reached by either walking to the east or by a teleporter. An instance of this can be seen in the example, because D1 and D2 are not reachable — there is no way to get to them because they are completely encapsulated within C1 and C2. An unreachable set of points will always be encapsulated within another teleporter in this same manner.

Now knowing both of these facts, we can safely conclude that T is not one of these unreachable points. This can be proved by contradiction:

- Say that teleporter X has an endpoint X2 that is right before T. X2 is the point in the 1D space that is closest to T.
- Say that both points of X are encapsulated by a teleporter named Y. This means that both points of X would be unreachable, and it would be an unreachable teleporter.
- When traversing the 1D space, there would be no way to reach X2 and walk east to T, since X2 is unreachable.
- There is a contradiction here — T can not be reached from X2 because X2 itself is unreachable. This means that there is no way that T can be unreachable, because the point right before it can not be an unreachable point. The point Y2 of teleporter Y that encapsulates teleporter X would be the new point that is closest to T. Since Y is reachable, that means T would be reachable as well.

T can be reached regardless of the setup of all the teleporters.

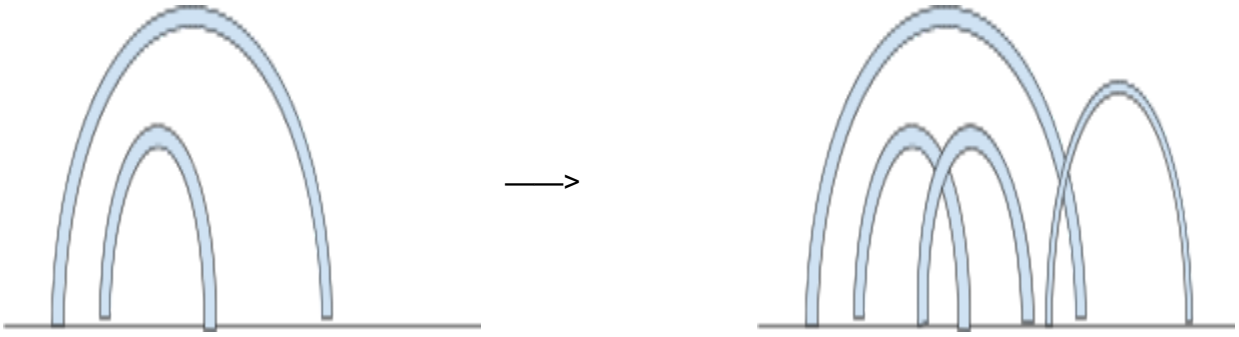
b) The ideal form that would maximize the number of teleports needed to reach T would be like this:



Before (Single Arch) After (Double Arch)

We will call this a “Double Arch” formation because it has a combination of 2 Single Arches. This formation is beneficial because it turns the SIngle Arch (2 maximum teleports going east and west) into 4 maximum teleports.

If there is an instance where a teleporter completely encapsulates a teleporter, this would be the ideal form:



Before (Nested Singles) After (Quad Arch)

We will call this a “Quad Arch” formation. This formation is created from a Single Arch that totally encapsulates another single arch. When a Single Arch totally encapsulates another single arch, I will refer to them as “Nested Singles”. Then both of the single Arches are transformed into Double Arches. The Double M formation is ideal because it would create a maximum of 8 teleports.

Pseudocode for Algorithm:

```
addTeleporters(int K):
  K = number of teleporters that we must add — given as parameter in algorithm
  while traversing the path eastward from S to T and K > 0:
    if you encounter a “Nested Single” and K ≥ 2:
      turn it into a “Quad Arch” by adding two teleporters
      K = K - 2
      continue
    if you encounter a Single Arch and K ≥ 1:
      turn it into a “Double Arch” by adding one teleporter
      K = K - 1
      continue
  if the traversal has ended and K > 0:
    add K many teleporters to the path such that they don’t intersect with any others
```

What this algorithm aims to do is to maximize the number of Quad Arches that get created. These Quad Arches have the potential for causing 8 total teleports when encountered, so they are very useful in increasing the total number of teleports. Next, the Double Arches have potential for causing 4 total teleports when they are encountered, so they are also useful in increasing the total number of teleports. Finally, if everything has already been turned into a Quad Arch or a Double Arch and K > 0, the extra teleporters can be thrown in just to increase the total number of teleports by a maximum of 2 for each one that is added. The time complexity of this algorithm is O(n) because of the traversal with the for loop, assuming adding the teleporters can be done in O(1) time.