CS 180 Homework 1

Make sure you start each problem on a new page

1. (a) We can start off by saying that there are two cases (m1 proposes first or m2 proposes first).

- Step 1: If m1 proposes first, then m1 will propose to w1 since w1 is at the top of m1's preference list. Here, w1 will accept and m1/w1 will be matched because w1 has not been matched to anyone yet.
- Step 2: After m1 proposes, m2 will propose to w2 since w2 is at the top of m2's preference list. Since w2 has not been matched with anyone yet, w2 will say yes.

- Step 1: If m2 proposes first, m2 will propose to w2 since w2 is at the top of m2's preference list. Since w2 has not been matched with anyone yet, w2 will say yes.
- Step 1: If m1 proposes after, then m1 will propose to w1 since w1 is at the top of m1's preference list. Here, w1 will accept and m1/w1 will be matched because w1 has not been matched to anyone yet.

(b) Proof by contradiction

- The claim is that if a match is a stable, then m1, m2 must be matched to w1, w2. Assume by contradiction that m1, m2 are NOT matched to w1, w2, but the match is somehow stable. This can be done in two different ways:

    Case 1: We can assume that m1 is matched to w3 (that is not w1 or w2). In this case, we know that m1 prefers w2 to w3 and that w2 prefers m1 to any m. Therefore, because m1 and w2 prefer each other to their current matchings, then this is unstable.

    Case 2: We can assume that m2 is matched to w3 (that is not w1 or w2). In this case, we know that m2 prefers w1 to w3 and that w1 prefers m2 to any m. Therefore, because m2 and w1 prefer each other to their current matchings, then this is unstable.

Therefore, in either case, we have a contradiction and the claim is proven.

2. (b)

Preference List:

m': w>w'	w: m''>m>m'
m: w> w'	w': m>m''>m'
m'': w'>w	

Truthful Case: w: m> m'

- m' first proposes to w and this is a match since w is unmatched.
- Then, m tries to propose to w and since w has m>m', then w unmatched with m' and matches with m.
- Since m' is now unmatched, we can have m' propose to w' and this is a match since w' is not currently matched with anyone.
- Lastly, we have m'' propose to w' since that is the highest on his preference list. Since w' has m'' higher than m', w' will unmatch with m' and match with m''.

Final Matches for the truthful case: (m, w) and (m", w')

Untruthful Case: w: m'>m

- M' first proposes to w and this is a match since w is unmatched.
- Then, M tries to propose to w and since w now has m'>m, then w will stay with m' and not change.
- Now, M will propose to the second person on his list which is w' and they will match since w' is currently unmatched.
- Now, m'' will propose to w' since it is on the top of his preference list, however since w' is already proposed to m and m>m'' on w's preference list, then m'' will not match with w'.
- M'' will now propose to w since it is second on its preference list and then m'' and w will match because w has m'' at the top of its preference list.

Final Matches for the Untruthful Case: (m'', w) and (m, w')

Conclusion: We can see that when w lies, she ends up matching with a better partner.

3. (a) Counterexample: Let f(n) = 2n and g(n) = n.

From the right side, we want it to be true such that 2^(2n) = O(2^n). If this were in fact true, then we would have 2^(2n) <= c(2^n) for some constant c. From here, we can simplify this equation by dividing each side by 2^n and our result is 2^n <= c. If we solve for n, we get n <= log2(c). Since n is supposed to be unbounded greater than 0, however, this ends up being a contradiction.

(b) Suppose n^n = Big-Theta(N). Then, there exists constants c1 and c2 such that c1(n!) <= n^n <= c2(n!). Here, we have two different inequalities and both need to hold in order for this to be true, therefore if one is not true, then the whole inequality is also not true. We can start by proving if n^n <= c2(n!). If we divide each side by n, we get n^(n-1) <= c2(n-1)! And if we continue doing this, the (n-1) term on both sides will decrease by 1 each time. Formally, this will become 1 <= c2(n!/n^n) and for each term, we have i/n for some i <= n. Because we are multiplying each side by the same number of terms, but we are multiplying the left side by n each time and the right side by n, then n-1, then n-2, etc, we can see that n^n will grow faster than n!, so it does not make sense for n^n <= c2(n!). Therefore, this is disproven.

(c) Suppose f(n) = O(g(n)), then we can say that f(n) <= c(g(n)) for any n >= n0 and for a constant c. Now, we can multiply both sides by g(n), the result is f(n)g(n) <= c(g(n)^2). By definition, this proves that If $f(n) = O(g(n))$, then $f(n) \cdot g(n) = O(g(n)^2)$.

4. Data Structure of “Medium Heap”:

There will be 2 heaps that we will use:

1) Max-Heap: will store all elements >= medium element
2) Min-Heap: will store all elements <= medium element

- If we add the size of min heap and max heap, we get total size of the “Medium Heap”

1st Operation: “push”

push (int): **Time Complexity is O(logn). Since we have a heap, the only operation we are doing besides pushing the value into the heap is checking the integer in comparison to the medium value and if the heap needs rebalancing, we do the same thing there. Because we already know the push and pop time complexities from before (both are O(logn)), the overall complexity is also O(logn)**

```
If (total size <= 0) //edge case where size is less than or equal to
        push int into min heap;
Else if (int < medium value)
        Push int into max heap;
Else if (int >= medium value)
        Push int into min heap;

//rebalancing: O(logn)
If (min-heap size > 1+max-heap size)
        P = pop min heap; //pop is O(logn)
        Push p into max heap; //push is O(logn)
If (max-heap size > 1+min-heap size)
        P = pop max heap;
        Push p into min heap;
```

2nd operation: “find-medium”

Find-medium: **Time Complexity is O(1) or constant time since all we are doing is an operation by checking the size of the max heap in relation to the min heap and then setting the medium value to that. This does not have any loops or anything else.**

```
If (size of medium heap = 0)
        Medium value = null;
Else if (max heap size = min heap size)
        Medium value = (root of maxheap + root of minheap)/2;
Else if (max heap size > min heap size)
        Medium value = root of maxheap;
Else if (max heap size < min heap size)
        Medium value = root of min heap;
```

5. **Time Complexity Justification**: Since the algorithm should run in O(logn) time, we can use binary search. Each time, we are reducing the searching array by half and only going through (log(base 2)(n)) times. Since the base of "2" can be ignored, we end up getting O(logn).

Algorithm:

Set left pointer (L) = 0, and right pointer (r) = n-1
While (L <= r)
    Middle = floor ((r+L)/2);
    If (A[middle] = middle) //we found where i = A[i] and are done!
        Return true;
    Else if (A[middle] > middle) //ignore the second (larger) half of the array and go lower
        r = middle-1;
    Else //A[middle] < middle //ignore the first (smaller) half of the array and go higher
        L =1+mid;

Return false; //unsuccessful

Correctness Justification:
Case 1: There exists an index i such that A[i] = i

-1	1	9	10	11
Index 0	Index 1	Index 2	Index 3	Index 4

- Set L = 0 and r = 4
- Iteration 1: middle = floor((4+0)/2) = 2. A[2] = 9 (and 9 > 2), therefore we do r = middle-1 and get the new r = 2-1 = 1. (NOW, L = 0, r = 1)
- Iteration 2: middle = floor((0+1)/2) = 0. A[0] = -1 and -1 < 2, therefore we do L = 1+middle, so we get the new L = 0 + 1 = 1. (NOW< L=1, r=1)
- Iteration 3: middle = floor((1+1)/2) = 1. A[1] = 1. Now, A[middle] = middle and we found our index!

Case 2: There does NOT exist an index i such that A[i] = i

1	3	9	10	11
Index 0	Index 1	Index 2	Index 3	Index 4

- Set L = 0 and r = 4

- Iteration 1: middle = floor((4+0)/2) = 2. A[2] = 9 (and 9 > 2), therefore we do r = middle-1 and get the new r = 2-1 = 1. (NOW, L = 0, r = 1)
- Iteration 2: middle = floor((0+1)/2) = 0. A[0] = 1 and 1 < 2, therefore we do L = 1+middle, so we get the new L = 0 + 1 = 1. (NOW< L=1, r=1)
- Iteration 3: middle = floor((1+1)/2) = 1. A[1] = 3 and 3 > 1 so we increment L again. Now, L = 2 and r = 1.
- Since L > r, we must exit the while loop and return false

5.