Hannah Kwon
705182275
Professor Hsieh

Homework #1

Computer Science 180 - Introduction to Algorithms and Complexity

Question 1

(a) When we run the Gale-Shapley algorithm, if the man is not matched with any women, the men propose to the women first by their preference. In this scenario, at the start m1 will propose to m1 and m2 will propose to w2.

W1 will accept the proposal of m1, because m1 is the second best choice of w1 and w1 didn't get a proposal from her first preference m2, so w1 will accept m1's proposal and rejects all other proposals that she would receive later on. In this way (m1, w1) matches. Similarly, w2 will accept the proposal of m2, because m2 is the second best choice of w2 and w2 will not get a proposal from m1(w2's first preference), so w2 will accept m2's proposal and reject all other proposals that she gets. In this way (m2, w2) matches

1)If w1, w2 are both unmatched
Whenever the men m1 or m2 in this case propose, if the women w1 and w2 are unmatched at the time, they can easily match pairs which add (m1, w1) and (m2, w2) to S.

2)If either w1, w2 are unmatched or w1,w2 are both matched
If woman w1 is matched with any other man before m1 proposes, and m1 proposes to w1, the pairs will be switched from (m', w1) to (m1,w1) since w1 prefers m1 to any other man besides m2 (m2 prefers w2 over w1 so this would not happen). Same thing happens for w2 as well. So (m1, m2) and (m2, w2) will be both added to S.

In conclusion, no matter what order the men propose to, (m1, w1) and (m2, w2) will be paired where the men are matched with their first choice and the women are matched with their second choice.

(b) There can be two possible ways to find the stable matching using Gale-Shapley's algorithm:

1. Men proposing
2. Women proposing

Before we prove this let us define the meaning of “stable meaning”.
The definition of stable matching is :
(1) the pairing is “perfect matching” which means each man is paired with exactly one woman.
(2) there shouldn’t be an unstable pair. An unstable pair occurs when there is a man and a woman who prefer each other to their current partners and occurs in two cases:
(2-1)m preferring another woman to his matched woman or
(2-2)woman preferring man to her matched man.

Case 1 : Men proposing
M1 proposes to w1(m1’s first choice)
M2 proposes to w2(m2’s first choice)
So if we consider the (m1, w1), (m2, w2) pairing from the men proposing case, this will be part of stable matching because
(1) men only pairs with no more than one women
(2-1) m1 prefers w1 over w2 and m2 prefers w2 over w1

Thus in this case (m1, w1) and (m2, w2) will surely be the part of the stable matching as explained in part (a).

Case 2 : Women proposing
W1 proposes to m2(w1’s first choice)
W2 proposes to m1(w2’s first choice)
M2 will accept the proposal of w1 since m2’s first choice which is w1 will not propose to m2. Thus, m2 will accept w1’s proposal and reject all other proposals that he receives. Similarly, m1 will accept the proposal of w2 since m1’s first choice which is w2 will not propose to m1. So m1 will accept w2’s proposal and reject all other proposals that he receives.

So if we consider the (m1, w2), (m2, w2) pairing from the women proposing case, this is a stable matching because
(1) men only pairs with no more than one women
(2-2) w1 prefer m2 over m1 and w2 prefer m1 over m2
thus (m1, w2) and (m2, w1) will be in the part of the stable matching.

Hence, we have proved that in every stable matching m1, m2 are matched to w1, w2
(m1, w1), (m2, w2) in case 1 when men are proposing or
(m1, w2), (m2, w2) in case 2 when women are proposing.

Question 2

(b) An example of a set of preference lists that a switch that would improve the partner of a woman would be

Men preference list :
m : w > w’ > w’’
m’ : w > w’ > w’’
m’’ : w’ > w > w’’

Women preference list :
w : m’’ > m > m’
w’ : m > m’’ > m’
w’’ : m > m’’ > m’

Before the woman w lies about her preference list to m’’> m’ > m,
(1) m proposes to w first (1st preference) (m, w)
(2) m’ proposes to w as well (1st preference). Since w prefers m over m’, m’’s proposal is rejected and (m,w) is still paired.
(3) m’ proposes to w’ (2nd preference) (m’, w’)
(4) m’’ proposes to w’ (1st preference). Since w’ prefers m’’ over m’, now (m’’, w’) is paired and m’ has no matching.
(5) m’ now proposes to w’’ (3rd preference) and since w’’ is not paired yet, (m’, w’’) pairs.

Pairs before lying :
(m, w), (m’, w’’), (m’’, w’)

However, if the woman w lies her preference list, to
w : m’’ > m’ > m

Men preference list :
m : w > w’ > w’’
m’ : w > w’ > w’’
m’’ : w’ > w > w’’

Women preference list :
w : m’’ > m’ > m
w’ : m > m’’ > m’
w’’ : m > m’’ > m’

(1) m proposes to w first (1st preference) (m, w)
(2) m’ proposes to w (1st preference). Since w prefers m’ over m, w is now paired with m’. (m’, w)
(3) m proposes to w’ (second preference) (m, w’)
(4) m’’ proposes to w’ (first preference). Since w’ prefers m over m’’, it reject’s proposal from m’’. (m, w’)
(5) m’’ proposes to w (second preference). Since w prefers m’’ over m’. (m’’, w)
(6) m’ proposes to w’ (2nd preference), but w’ prefers m over m’ so reject’s proposal from m’.
(7) m’ proposes to w’’ (3rd preference) (m’, w’’)

Pairs after lying :
(m, w’), (m’, w’’), **(m’’, w)**

And from this preference list example, it disapproves that a man cannot switch a preference list. However, a woman can switch preferences. Before lying, w was paired with m which is her 2nd preference, but after lying she is paired with w’’ which is her first preference.

Question3

(a)

Prove By contradiction.

Let's assume $f(n)=2\log_2 n$ and $g(n)=\log_2 n$

We know $2\log n <= c\log n$ thus $f(n)=O(g(n))$

$2^{f(n)}=2^{2\log_2 n}=2^{\log_2 n^2}=n^2$

$2^{g(n)}=2^{\log_2 n}=n$

in this example we found $2^{f(n)}=n^2$ which is not $O(n)$ as $2^{g(n)}=n$ which is $O(n)$.

Therefore, $2^{f(n)} \neq O(2^{g(n)})$ and this statement is <u>false</u>.

(b) $n^n = \Theta(n!)$

To prove $n^n=\Theta(n!)$, we could prove by using induction.

Base Case: $n=1$

$1^1 \leq 1!$

$1 \leq 1$ True

Induction hypothesis: If for arbitrarily value $n=k$, $k! \leq k^k$

Prove the hypothesis by proving for if $n=k+1$, $(k+1)! \leq (k+1)^{(k+1)}$

$(k+1)k! \leq (k+1)k^k$

$(k+1)! \leq (k+1)k^k$

and we know that $(k+1)k^k < (k+1)(k+1)^k$ since that $k>0$

$$k^k < (k+1)^k$$

$$\underbrace{k \cdot k \cdot k \cdots k}_{k \text{ times}} < \underbrace{(k+1)\cdot(k+1)\cdots(k+1)}_{k \text{ times}}$$

andsince $k>0$, $(k+1)^k$ is always bigger than k^k.

So we can put

$(k+1)! \leq (k+1)k^k < (k+1)(k+1)^k$

$(k+1)! \leq (k+1)(k+1)^k$

$\therefore$ This proves $k! \leq k^k$ for all values $k \geq 1$

which also proves $n! \leq n^n$ for all values $n \geq 1$

Thus $n! \in O(n^n)$ which the statement $n^n=\Theta(n!)$ is <u>false</u>.

(C) If $f(n) = O(g(n))$, then $f(n) \cdot g(n) = O(g(n)^2)$

Given $f(n) = O(g(n))$

then $f(n) \leq c\, g(n)$ since $f(n).g(n) > 0$, for every large n and some constant c

then if we multiply both sides by $g(n)$

Since $g(n) > 0$, $f(n)\, g(n) \leq c(g(n))^2$

Thus proves the statement. True

Question 4

So the big idea for the new data structure medium heap is that it builds both a max heap for the lower half of the numbers and a min heap for the upper half of the numbers that are being pushed into the algorithm (lowers : stores the lower half of the numbers, greaters : stores the upper half of the numbers). With the variable medium it will continuously update the median.
(1) push : Used pushToMaxOrMinHeap function to push in elements
(2) find-medium : Used findMedium for class name and updateMedium for function to specify and used find-medium value to find the medium

```
#build both max heap and min heap
class Heap :
  MAX_HEAP_FUNC(a, b) :
    return a > b
  MIN_HEAP_FUNC(a, b) :
    return a < b

#will be storing the
#lower half of the value as max heap and
#upper half of the value as min heap
#will be updating the medium value in variable find-medium
class findMedium:
  lowers = Heap(MAX_HEAP_FUNC, [])
  greaters = Heap(MIN_HEAP_FUNC, [])
  find-medium = None

  #if lowers is empty or
  #the number being pusheis smaller than the maximum number
  #from the max heap of the lower half of the heap
  #push the number into the lower half of the heap
  #O(logn) time since it's either pushing it into a
  #min-heap or a max-heap (discussed in class)
  #both for pushToMaxOrMinHeap and rebalanceHeaps which makes it 2*O(logn)=O(logn)
  #O(n) space
  def pushToMaxOrMinHeap(self, number) :
    if lowers is empty or number < maximum number in lowers :
      lowers.pushToMaxOrMinHeap(number)
    else:
      greaters.pushToMaxOrMinHeap(number)
    rebalanceHeaps()
    updateMedium()
```

```
#rebalance heap by making the length between two heaps no bigger than 2
#pop() would be removing the biggest value in lowers and smallest value in greaters
#and then pushing it to the opposite heap
def rebalanceHeaps(self) :
  if lowers.length - greaters.length == 2 :
    greaters.pushToMaxOrMinHeap(lowers.pop())
  elif greaters.length - lowers.length == 2 :
    lowers.pushToMaxOrMinHeap(greaters.pop())

#updating find-medium is just comparing the length
#O(1) time
def updateMedium(self) :
  if lowers.length == greaters.length: #even number
    find-medium = (maximum number in lowers + maximum number in greaters) / 2
  elif lowers.length > greaters.length: #odd number
    find-medium = maximum number in lowers
  else :
    find-medium = maximum number in greaters
```

Question 5

In this algorithm, in order to find a[i] = i we will use binary search by eliminating half of the arrays by checking if the element is greater than the mid element, smaller than or equal to it.

In the function binarySearchAfterSorted(sortedArray, left, right), sortedArray is the array sorted and left, right are two pointers each pointing to the first element of the sortedArray and the last element of the sortedArray.

It would be done in a recursive algorithm.
def binarySearchAfterSorted(sortedArray, left, right) :

1. Base case : if left > right return
2. Need to calculate med = (left + right) / 2 to compare with the value i.
    1. if med == sortedArray[i] return True. This means an element from sortedArray[i] that equals i is found and should return True.
    2. elif med < sortedArray[i] return binarySearchAfterSorted(sortedArray, left, med). This means that the med value is in the left half of the array which means that it only needs to search for the left half of the array.
    3. elif med > sortedArray[i] return binarySearchAfterSorted(sortedArray, med, right). This means that the med value is in the right half of the array which means that it only needs to search for the right half of the array.
3. return False The algorithm should return False if element i is not found in the array.

Since this algorithm is using a modified version of the binary tree, the time complexity is still O(logn).