Homework 1

Raymond Lin, 304937942

April 17, 2020

Problem 1

Proof: (by contradiction)
Let the opposite be true - ie. let there be a stable matching, S, where $(m, w) \notin S$. Then, $\exists m'$ and w' such that $(m, w') \in S$ and $(m', w) \in S$. We know that m is ranked first on w's preference list and vice versa. Therefore, m prefers w over w' and w prefers m over m'. By definition, this is an unstable matching. So S is an unstable matching.
$\implies$ Contradiction, since we assumed S was stable. Thus, if $(m, w) \in S$, S must be stable. $\square$

Problem 2

a)

Since w_1, w_2 are at the top of m_1, m_2's preference lists, we can disregard all other males in the algorithm. We consider two cases, when either m_1 or m_2 propose before the other, for the first time:

1. If m_1 proposes before m_2, then m_1 will be matched with w_1 because m_1 is ranked higher than any other man, except m_2 on w_1's preference list. Since, m_2 has not yet proposed, m_1 is matched with w_1. Then, m_2 proposes to w_2, because she is at the top of his preference list. w_2 will be matched with m_2, because m_2 is ranked higher than any other man except m_1 on w_2'w preference list. Since both m_1 and m_2 are matched, the only way they become unmatched is if another man proposes to their matches, and their matches agree. However, this will be impossible, because all other men are ranked lower than m_1 and m_2 in w_1 and w_2's preference lists.

2. If m_2 proposes before m_1, then the same thing would happen as above, except m_1 and m_2's positions are switched. Regardless, m_1 still becomes matched with w_1 and m_2 still becomes matched with w_2.

In both cases, m_1 is matched with w_1 and m_2 is matched with w_2.

b)

Proof: (by contradiction)
Let there be a stable matching S, such that $(m_1, w') \in S$ where $w' \neq w_1, w_2$ and $(m_2, w_2) \in S$. Because $(m_1, w') \in S$ and m_1's first preference is w_1, m_1 must have been rejected by w_1. We know that the only person who w_1 prefers over m_1 is m_2. Because $(m_2, w_2) \in S$, and m_2's first preference is w_2, m_2 must have only proposed to w_2.
$\implies$ Contradiction, since w_1 can only reject m_1 if m_2 proposed to her. Thus, S is not stable, but we assumed S was stable. Thus, every stable matching must contain either contain the pairs (m_1, w_1) and (m_2, w_2) or the pairs (m_1, w_2) and (m_2, w_1). $\square$

Problem 3

$$f_2(n), f_3(n), f_6(n), f_1(n), f_4(n), f_5(n)$$
$$\sqrt{2n}, n + 10, n^2 log(n), n^{2.5}, 10^n, 100^n$$

Problem 4 Given: $f(n) = O(g(n))$

a) $g(n) = \Omega(f(n))$

Proof:
It must be true that $\exists n_0, c$ such that

$$f(n) \leq cg(n) \forall n > n_0$$

Rearranging, we have

$$\frac{1}{c} f(n) \leq g(n)$$
$$g(n) \geq \frac{1}{c} f(n)$$

Let $\frac{1}{c} = C$. We have Rearranging, we have

$$g(n) \geq C f(n)$$

By definition, $g(n) = \Omega(f(n))$. $\square$

b) $f(n) \bullet g(n) = O(g(n)^2)$

From above,

$$f(n) \leq cg(n)$$

Multiply both sides by $g(n)$

$$f(n) \bullet g(n) \leq cg(n)^2$$

By definition, $f(n) \bullet g(n) = O(g(n)^2)$. $\square$

c) $2^{f(n)} = O(2^{g(n)})$

Counterexample: let $f(n) = 2n, g(n) = n$.

$$2^{f(n)} = O(2^{g(n)})$$
$$2^{2n} = O(2^n)$$
$$4^n = O(2^n)$$
$$4^n \leq c \bullet 2^n \forall n > n_0$$

There is no c that satisfies the above. Thus, $2^{f(n)} \neq O(2^{g(n)})$. $\square$

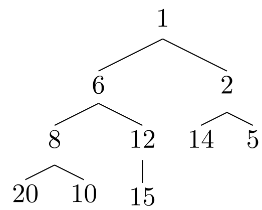
Problem 5

Idea: First find the length of the linked list. Then we traverse it again until we reach halfway. At this point, we reverse the second half of the linked list, such that the pointers point inwards towards the middle. At the end, we save a pointer to the tail. Finally, we scan the linked from out to in. We use two pointers, one starting at the head and one starting at the tail, and we compare numbers that the two pointers are pointing to. We stop if any two values are different, meaning the list is not a palindrome, or when we reach the middle, meaning the list is a palindrome. The time complexity of this algorithm is $O(n)$, because in total, we traverse the linked list 3 times.

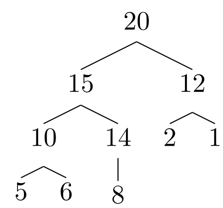
```
isPalindrome(head):
   //get the length of the linked list
   length = 0
   curr = head
   while curr is not null:
      length += 1
      curr = curr.next

   //reverse half of the linked list
   mid = ⌊length/2⌋
   index = 0
   curr = head
   tail =null
   prev = null
   while curr is not null:
      if index > mid:
         temp = curr.next
         curr.next = prev
         prev = curr
         curr = temp
      else:
         curr = curr.next
   index += 1
   tail =prev

   //two pointers from both ends
   left = head
   right = tail
   index = 0
   while index < mid:
      if left.val not equal right.val:
         return false
      left = left.next
      right = right.next
      index += 0
   return true
```

Problem 6

a)



b)



c)

Idea: We maintain a minheap and a maxheap which each contain around half of the number of elements. If the number of elements in each heap is the same, then the median is the average between the two roots of the heaps. If they are not the same, then the median is the root of the heap with a larger number of elements.

Each insertion is of order $O(logn)$, because we either add to the minheap or the maxheap. If we need to rebalance, we pop an element off one of the heaps and insert into the other heap, which is still $O(logn)$. Finding the median is also order $O(logn)$, because we simply look at the roots of each heap. In fact, finding the median is constant time - $O(1)$.

```
class MediumHeap:
public:
  def insert(x):
    //insert in either minheap or maxheap
    curr_median = this.find_medium()
    if x > curr_median
      minheap.push(x)
      minheapsize += 1
    else if x < curr_median
      maxheap.push(x)
      maxheapsize += 1
    else
      if minheapsize < maxheapsize
        minheap.push(x)
      else
        maxheap.push(x)

    //rebalance if needed
    if absolute value of (maxheapsize - minheapsize) > 1
      if minheapsize < maxheapsize
        minheap.push(maxheap.pop())
        minheapsize += 1
        maxheapsize -= 1
```

```
      else
        maxheap.push(minheap.pop())
        maxheapsize += 1
        minheapsize -= 1

  def find_medium():
    if maxheapsize > minheapsize
      return max_heap.find_max()
    else if maxheapsize < minheapsize
      return min_heap.find_min()
    else
      return (max_heap.find_max()+min_heap.find_min())/2
private:
  maxheap
  minheap
  maxheapsize = 0
  minheapsize = 0
```