Problem Set 1

Name: Ziyou Ren

Collaborators: None

Problem 1-1.

(a)
1. $f_1 = \log(n^n) = \Theta(n\log(n))$
2. $f_2 = (\log(n))^n = \omega(2^n)$. This asymptotic complexity can be easily confirmed by taking log on both sides.
3. $f_3 = \log(n^{6006}) = 6006\log(n) = \Theta(log(n))$
4. $f_4 = (\log(n))^{6006} = \Theta((\log(n))^{6006})$
5. $f_5 = \log\log(6006n) = \log[\log(6006) + \log(n)] = \Theta(\log\log(n))$

Clearly, the f_5 is the first item. Then comes $\Theta(log(n))$, which is f_3. Then comes $\Theta(log(n)^c)$, where c is a constant. Then f_1 and f_2 which is higher then exponential complexity.
$(f_5, f_3, f_4, f_1, f_2)$

(b)
1. $f_1 = \Theta(2^n)$
2. $f_2 = \Theta(6006^n)$
3. $f_3 = 2^{6006^n} = \Theta((2^{6006})^n)$
4. $f_4 = 6006^{2^n} = \Theta((6006^2)^n)$
5. $f_5 = 6006^{n^2} \iff \log\log(f_5) = \Theta(log(n))$

Clearly, f_1 comes first, then is f_2. For $f_3\ and\ f_4$, we can know that $2^{6006} \geq 6006^2$, so f_4 comes earlier than f_3. Finally, $\log\log(f_3) = \Theta(n)$, so f_5 comes earlier than $f_3\ and\ f_4$.
$(f_1, f_2, f_5, f_4, f_3)$

(c)
1. $f_1 = \Theta(n^n)$
2. $f_2 = \Theta(n^6)$
3. $f_3 = (6n)! = \sqrt{2\pi n}(\frac{6n}{e})^{6n} = \Theta(\sqrt{2\pi n}(\frac{6n}{e})^{6n})$
4. $f_4 = \frac{n!}{(\frac{n}{6})!(n-\frac{n}{6})!} = \Theta(\frac{(6)^{\frac{n}{6}}(\frac{5}{6})^{\frac{5n}{6}}}{\sqrt{\frac{5\pi n}{18}}})$
5. $f_5 = \Theta(n^6)$

$(\{f_2, f_5\}, f_4, f_1, f_3)$

(d) This few functions are easier to compare when taken the logarithmic. While one thing has to be noted is that a constant difference in logarithmic asymptotic complexity matters as a polynomial asymptotic complexity in the original function.
In short, just be careful with the same asymptotic complexity of the logarithm form of functions.

1. $\log(f_1) = (n+4)\log(n) + log(\sqrt{2\pi n}) + n\log(n) - n\log(e) = \Theta(n\log(n))$
2. $\log(f_2) = 7\sqrt{n}\log(n) = \Theta(\sqrt{n}\log(n))$
3. $\log(f_3) = 3n\log(n)\log(4) = 6n\log(n) = \Theta(n\log(n))$
4. $\log(f_4) = n^2\log(7) = \Theta(n^2)$
5. $\log(f_5) = (12 + \frac{1}{n})\log(n) = \Theta(\log(n))$

Clearly, on a rough approximation, we got $(f_5, f_2, \{f_1, f_3\}, f_4)$.
However, we still need to make sure that $f_1\ and\ f_3$ has the same asymptotic complexity. Obviously, we can see that $f_1 < f_3$.
Thereby, we got $(f_5, f_2, f_1, f_3, f_4)$

Problem 1-2.

(a) To be clear about what we are going to do, we assume that the data structure sequence will shift left when we delete an item from the sequence i.e. the sequence length is changed to $n-1$ after one deletion operation.
A simple idea can be that we swap the first item and the last item and run a for loop to expand this operation to all k items. A further idea would be to do this recursively, and stop at the base case where there is only one item left i.e. $k = 1$.
The pseudo code is as following:

```
def reverse(D,i,k):
    '''
    reverse in D the order of the k items starting at indexing $i$.
    try this in python with insert and pop, D is a list.
    '''
    # Base Case
    if k<2:
        return D
    # Inductive step
    first_item = D.delete_at(i)
    last_item = D.delete_at(i+k-2)# after deletion the length is n-1

    D.insert_at(i, last_item)
    D.insert_at(i+k-1, first_item)

    reverse(D,i+1,k-2)
    # note here on the next step indexing from i+1 to k-2 has exactly the
    # remaining part.
```

The above implementation can be either empirically testified by testing the following code in Python or easily proved by induction.

```
def reverse(D,i,k):

    if k<2:
        return D

    first_item = D.pop(i)
    last_item = D.pop(i+k-2)

    D.insert(i,last_item)
    D.insert(i+k-1,first_item)

    reverse(D,i+1,k-2)
```

(b) To do this in a recursive form, we can try to move the first item in front of j and do the same operation recursively to the rest of the k items.

```python
def move(D,i,k,j):
    '''
    :param D: The input data structure i.e. list in python
    :param i: starting index
    :param k: length of items to be moved
    :param j: index to be moved in front of, j should not be in the range of the k ite
    :return: the D after moved
    '''

    # Base case
    if k<1:
        return D

    # Inductive step
    if i<j:
        item = D.pop(i)
        D.insert(j-1,item)

        move(D,i,k-1,j)

    else:
        item = D.pop(i)
        D.insert(j,item)

        move(D,i+1,k-1,j+1)
```

The above implementation can be either empirically testified by testing the above code in Python or easily proved by induction.
Note that the insert_at and delete_at methods are exactly the same as insert and pop in Python.

Problem 1-3.

The data structure should be sequence like and, indeed, should be an array-based implementation considering that the read_ page method need to be run in $O(1)$ time.
One more thing is that we need to keep track of the two bookmarks' index/address, so that we would get the index of the bookmarks given its symbol in $O(1)$ time. This will be like two pointers.

1. build(X): For any array-based data structure, the build of the array is $O(n)$ given n items. In this particular array, we can always rebuild the array using move_ page() method, which will run in $O(1)$ time for moving a given page. Then do this recursively for the rest of the pages. Our methods would be worst case $O(n)$ for rebuilding n items.
2. place_ marks(i,m): For a dynamic array like list in Python, we can insert an item at any index and move the other by shifting right.
   The insert operation takes $O(1)$ time as read_ page() takes $O(1)$ time. However, the shift_ right operation takes $O(n)$ time as rebuilding the rest of items takes $O(n)$ time.
   This is also worst case $O(n)$ if we place a book mark at the beginning of the array.
3. read_ page(i): This is a basic operation of word of RAM model, we can achieve any item given its index/address. **Notice** that we have to compare the index of bookmarks and the pointed index we want first to cancel the index occupied by bookmark.
   Since the comparison takes $O(1)$ time, this method is worst case $O(1)$ time.
4. shift_ mark(m,d): Given our data structure, we can get the index of the bookmark in $O(1)$ time and the rest of the operation is simply swap two items with d decreasing by 1, and do the rest recursively.
5. move_ page(m): Since we have two pointers for the bookmarks, we can easily access the item of the index of the pointers' address. **However**, the hardest part is that shifting other part takes $O(n)$ time and we cannot use a whole link list otherwise the read_ page method takes $O(n)$ time.
   Here we give a solution that the item at the bookmarks' pointers' address should be doubly linked list and these two liked list form won't harm the overall running time.
   Thereby, only if the two bookmarks' addresses are adjacent, the running time is $O(n)$, and the method is amortized $O(1)$.

Problem 1-4.

(a) This part of problem has to consider two aspects, whether the link list is empty or not. If it's empty, the tail and head would point to the same node that's newly added to the list.
Otherwise, the head or tail would point to the newly added node that depends on whether you insert at first or last.

(b) This part of problem has to consider whether x_1 is the first or not and whether x_2 is tail or not.

(c) This part of problem has to consider whether x is tail or not.

(d) Submit your implementation to `alg.mit.edu`.

```
class Doubly_Linked_List_Seq:
    def __init__(self):
        self.head = None
        self.tail = None

    def __iter__(self):
        node = self.head
        while node:
            yield node.item
            node = node.next

    def __str__(self):
        return '-'.join(['(%s)' % x) for x in self])

    def build(self, X):
        for a in X:
            self.insert_last(a)

    def get_at(self, i):
        node = self.head.later_node(i)
        return node.item

    def set_at(self, i, x):
        node = self.head.later_node(i)
        node.item = x

    def insert_first(self, x):
        ############################
        # Part (a): Implement me! #
        ############################
        '''
        :param x: an item (not a link list node)
        :return: a new link list with x to be the first node
        '''
```

```
35
36          new_first_node = Doubly_Linked_List_Node(x)
37          # if the link list is empty
38          if self.head is None:
39              self.head = new_first_node
40              self.tail = new_first_node
41          # the link list is not empty
42          else:
43              new_first_node.next = self.head
44              self.head.prev = new_first_node
45              self.head = new_first_node
46
47      def insert_last(self, x):
48          ############################
49          # Part (a): Implement me! #
50          ############################
51          new_last_node = Doubly_Linked_List_Node(x)
52
53          # if the link list is empty
54          if self.tail is None:
55              self.head = new_last_node
56              self.tail = new_last_node
57
58          # if the link list is not empty
59          else:
60              new_last_node.prev = self.tail
61              self.tail.next = new_last_node
62              self.tail = new_last_node
63
64
65      def delete_first(self):
66          x = None
67          ############################
68          # Part (a): Implement me! #
69          ############################
70          x = self.head.item
71          self.head = self.head.next
72          if self.head is None:
73              self.tail = None
74          else:
75              self.head.prev = None
76
77          return x
78
79      def delete_last(self):
80          x = None
81          ############################
82          # Part (a): Implement me! #
83          ############################
84          x = self.tail.item
85          self.tail = self.tail.prev
```

```
86          if self.tail is None:
87              self.head = None
88          else:
89              self.tail.next = None
90
91          return x
92
93      def remove(self, x1, x2):
94          L2 = Doubly_Linked_List_Seq()
95          ############################
96          # Part (b): Implement me! #
97          ############################
98          L2.head = x1
99          L2.tail = x2
100
101         # connect the rest of the nodes into a new link list
102         if x1 == self.head:
103             self.head = x2.next
104         else:
105             x1.prev.next = x2.next
106
107         if x2 == self.tail:
108             self.tail = x1.prev
109         else:
110             x2.next.prev = x1.prev
111
112         # clear L2's head an tail
113         x1.prev = None
114         x2.next = None
115
116         return L2
117
118     def splice(self, x, L2):
119         ############################
120         # Part (c): Implement me! #
121         ############################
122         x1 = L2.head
123         x2 = L2.tail
124         # empty the L2
125         L2.head = None
126         L2.tail = None
127         xn = x.next
128         # reconstruct the new link list
129         x1.prev = x
130         x.next = x1
131
132         x2.next = xn
133         # if xn is the tail i.e. x is the last node in link list
134         if xn is not None:
135             xn.prev = x2
136         else:
```

```
137                 self.tail = x2
```