Problem Set 1

Name: Akshay Raman

Problem 1-1.

(a) $\{f_5, f_3, f_4, f_1, f_2\}$. Using logarithm and exponential rules, we can simplify the functions. $f_1 = \Theta(n \log n)$, $f_2 = \Theta((\log n)^n)$, $f_3 = \Theta(\log n)$, $f_4 = o(n)$, $f_5 = \Theta(\log \log n)$

(b) $\{f_1, f_2, f_5, f_4, f_3\}$. Convert all exponents to the same base (2), $f_1 = \Theta(2^n)$, $f_2 = \Theta(2^{(\log 6006)^n})$, $f_3 = \Theta(2^{6006^n})$, $f_4 = \Theta(2^{(\log 6006)2^n})$, $f_5 = \Theta(2^{(\log 6006)n^2})$.

(c) $\{\{f_2, f_5\}, f_4, f_1, f_3\}$. Using sterling's approximation and formula for n choose k. $f_1 = \Theta(n^n)$, $f_2 = \Theta(n^6)$, $f_3 = \Theta(\sqrt{n}\left(\frac{6}{e}\right)^{6n} n^{6n})$, $f_4 = \Theta((6/5^{5/6})^n/\sqrt{n})$, $f_5 = \Theta(n^6)$

(d) $\{f_5, f_2, f_1, f_3, f_4\}$ Exponent term dominates the base. Can take logarithm of all function to understand better. $f_1 = \Theta(n^{n+4})$, $f_2 = \Theta(n^{7\sqrt{n}})$, $f_3 = \Theta(n^{6n})$, $f_4 = \Theta(7^{n^2})$, $f_5 = \Theta(n^{12} n^{1/n})$

Problem 1-2.

(a) Thinking recursively, to reverse a sub-sequence with k elements, we can swap the ends i.e. elements at indices i and $i + (k - 1)$, a recursively solve the sub-problems. For the base case $k < 2$, no work needs to be done. The correctness of the algorithm can be proved using induction.

The swap can be done by removing the elements in the reverse order (right end then left end). This will preserve the index values while deleting. Then we can insert in the correct order (left end then right end). So make use of index values from before.

Each swap performs four $O(\log n)$-time operations, so it happens in $O(\log n)$ time. At most, $k/2$ recursive calls are made which is $O(k)$. Therefore, the running time of the algorithm is $O(k \log n)$

```
REVERSE(D, i, k):
        if k < 2:
                return None
        xr = D.delete_at(i+k-1)
        xl = D.delete_at(i)
        D.insert_at(i, xr)
        D.insert_at(i+k-1, xl)
        REVERSE(D, i+1, k-2)
```

(b) Thinking recursively, to move a sub-sequence with k elements, we can move the first item at index i in front of index j and then recursively move the sub-sequence of size $(k - 1)$ in front of that. For the base case $k = 0$, we don't have to move anything. If we maintain that: i is the starting element of the subsequence, j is index of the item in front of which we have to place subsequence, and k is the size of the subsequence, we have prove that algorithm works correctly by induction.

After removing an item at index i. If $j > i$, the value of j decreases by 1 i.e. $j = j - 1$. Also, when we insert an item after index j and $j < i$, the entire subsequence shift to the right by one i.e. $i = i + 1$.

The recursive procedure make no more than $O(k)$ recursive call. In each call, it does $O(\log n)$ work. Therefore, the running time of the algorithm is $O(k \log n)$.

```
MOVE(D, i, k, j):
        if k < 1:
                return None
        x = D.delete_at(i)
        if (j>i):
                j -= 1
        D.insert_at(j+1)
        j += 1
        if (j<i):
                i+=1
        MOVE(D, i, k-1, j)
```

Problem 1-3.

Store n pages in a static array S of size $3n$. This can be build in $O(n)$ time whenever $build(X)$ or `place_mark(i, m)` is called. S is divided into three sub-arrays P_1, P_2, P_3 of size n with the bookmarks at the divisions. We maintain some indices: $a_s = 0$, a_e points to the end of P_1, b_s points of the start of P_2, b_e points to the end of P_2, c_s points of the start of P_3, c_e points to the end of P_3.

To support `read_page(i)`, we calculate the actual index of the element using the pointer from above:

- If $i < |P_1|$, return $S[i]$.
- If $|P_1| \leq i < |P_1| + |P_2|$, return $S[b_s + i - a_e]$.
- Otherwise, return $S[c_s + i - a_e - (b_e - b_s)]$.

The running time of this operation is worst-case $O(1)$.

The procedure `shift_mark(m,d)` involves moving the bookmark element one position left or right in the array. If we move bookmark A to the left, the rightmost element in P_1 becomes the first element in P_2. So we only have to make a single swap in $\Theta(1)$ and update pointers to preserve the invariant. Similarly, we can show that all such operations can be done in worst case $O(1)$ running time.

`move_page(m)` will take $O(1)$ time to move elements and update the pointers. However, if any sub-array P_1, P_2, P_3 exceeds its limit (n), we will have to rebuild the array with extra space. Since this rebuilding happens once every n operations, the running time of this procedure is amortized $O(1)$ time.

Problem 1-4.

(a)
- `insert_first(x)`: Create a new node x. If the list is empty, set both $L.head$ and $L.tail$ to x. Otherwise, connect node x to $L.head$ by updating their next and prev pointers. Then, we set $L.head$ to the newly created node x.
- `insert_last(x)`: Same as `insert_first(x)` but this time we update $L.tail$ instead.
- `delete_first()`: Extract and store the head node from the list. Then update $L.head$ to the next node in the list. If there is no next node, then set $L.head$ to None.
- `delete_last()`: Extract and store the tail node from the list. Then update $L.tail$ to the previous node in the list. If there is no previous node, then set $L.tail$ to None.

(b) Construct a new list by setting x_1 and x_2 as the head and tail respectively. When x_1 or x_2 are ends of L, care must be taken to update $L.head$ and $L.tail$. If $L.head = x_1$, then set the new head to be the next node of x_2 (called a). Otherwise, set the next node of x_1's previous node to a. If $L.tail = x_2$, then set the new tail to be the previous node of x_1 (called b). Otherwise, set the previous node of x_2's next node to b. This algorithm remove nodes between x_1 and x_2 directly so it is correct. The running time is $O(1)$ since be make a constant number of pointer updates.

(c) To splice into L_1, first store the next value of x in variable x_{next}. Then, set the next node of x to $L_2.head$ (and previous node of $L_2.head$ to x). Also, we set previous node of x_{next} to $L_2.tail$ (and next node of $L_2.tail$ to x). To make L_2 empty, we can simply set its head and tail to point to None. If x_{next} is None, we set $L_1.tail$ to $L_2.tail$. This algorithm splices L_2 into L_1 directly, so it is correct. Since we make a constant number of pointer updates, the running

(d) This python implementation for all operations is given below:

```python
class Doubly_Linked_List_Node:
    def __init__(self, x):
        self.item = x
        self.prev = None
        self.next = None

    def later_node(self, i):
        if i == 0: return self
        assert self.next
        return self.next.later_node(i - 1)

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):
        new_node = Doubly_Linked_List_Node(x)
        if self.head is None:
            self.head = self.tail = new_node
        else:
            self.head.prev = new_node
            new_node.next = self.head
            self.head = new_node



    def insert_last(self, x):
        new_node = Doubly_Linked_List_Node(x)
        if self.tail is None:
            self.head = self.tail = new_node
        else:
            self.tail.next = new_node
            new_node.prev = self.tail
            self.tail = new_node

    def delete_first(self):
        assert self.head
        x = self.head.item
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        else:
            self.head.prev = None
        return x

```

```python
    def delete_last(self):
        assert self.tail
        x = self.tail.item
        self.tail = self.tail.prev
        if self.tail is None:
            self.head = None
        else:
            self.tail.next = None
        return x

    def remove(self, x1, x2):
        L2 = Doubly_Linked_List_Seq()
        L2.head = x1
        L2.tail = x2
        if x1 == self.head:
            self.head = x2.next
        else:
            x1.prev.next = x2.next
        if x2 == self.tail:
            self.tail = x1.prev
        else:
            x2.next.prev = x1.prev
        x1.prev = None
        x2.next = None
        return L2

    def splice(self, x, L2):
        xn = x.next
        L2.head.prev = x
        x.next = L2.head
        L2.tail.next = xn
        if xn:
            xn.prev = L2.tail
        else:
            self.tail = L2.tail
```