Name: Fan Wu

Collaborators: None

Problem 1-1.

(a) Simplify these functions using exponentiation and logarithm rules, we have:
$\Theta(f_1) = \Theta(nlogn), \Theta(f_2) = \Theta((logn)^n), \Theta(f_3) = \Theta(logn), \Theta(f_4) = \Theta((logn)^{6006}),$
$\Theta(f_5) = \Theta((log(log6006n)))$.
It is easy to conclude that the result is $\{f_5, f_3, f_4, f_1, f_2\}$

(b) Convert all the exponent bases to 2, we have:
$f_1 = 2^n, f_2 = 2^{nlog6006}, f_3 = 2^{6006^n}, f_4 = 2^{2^n log6006}, f_5 = 2^{n^2 log6006}$
It is obvious to conclude that the result is $\{f_1, f_2, f_5, f_4, f_3\}$

(c) $\Theta(f_1) = \Theta(n^n), \Theta(f_2) = \Theta(n^6) = \Theta(n^c), \Theta(f_5) = \Theta(n^c)$
Using Sterling's approximation, we can simplify f_3 and f_4 to:
$\Theta(f_3) = \Theta(\sqrt{2\pi(6n)}(\frac{6n}{e})^{6n}), \Theta(f_4) = \Theta(\frac{1}{\sqrt{n}}(\frac{6}{5^{5/6}})^n) \approx \Theta(\frac{1}{\sqrt{n}}(1.57)^n)$
So, f_3 is the largest, and f_1 is larger than f_4, while f_2 and f_5 are equal. The result is $\{\{f_2, f_5\}, f_4, f_1, f_3\}$

(d) Take the logarithms of these functions, we have:
$\Theta(logf_1) = \Theta((n+4)logn) = \Theta(nlogn), \Theta(logf_2) = \Theta(\sqrt{n}logn), \Theta(logf_3) = \Theta(nlogn), \Theta(logf_4) = \Theta(n^2), \Theta(logf_5) = \Theta(logn)$
It seems that f_1 and f_3 are asymptotically equal. Transform f_3 into:
$f_3 = (4^{logn})^{3n} = (n^{log4})^{3n} = n^{6n}$
So f_3 is asymptotically larger than f_1 (by about a factor of n^{5n}), the result is $\{f_5, f_2, f_1, f_3, f_4\}$

Problem 1-2.

(a) Method 1: Use a for loop

Use variants x1 and x2 to record the first and last item which are about to be swapped in this loop. We use D.delete_ at function to get the item deleted, then use the D.insert_ at function to swap the position of x1 and x2. After swapping, make the index of x1 move forward 1 step and x2 backward 1 step. The loop ends after k//2 times.This procudure would bu correct by induction.

D.delete_ at and D.insert_ at function cost $O(logn)$ time, so swapping two items needs $O(logn)$ time, the loop takes k//2 steps, so overall the algorithm takes $O(klogn)$ time.

```
for j in range(k//2):
    x1 = D.delete_at(i+j)
    x2 = D.delete_at(i+k-1-j)
    D.insert_at(i+j,x2)
    D.insert_at(i+k-1-j,x1)
```

Method 2: Recursion

In order to reverse all the k items in the sequence, we can swap the item at index i and $i + k - 1$, and then recursively reverse the rest of the items. As a base case, no work needs to be done to reverse a subsequence containing less than 2 items. The procudure would be correct by induction.

The swapping process is the same as Method 1. The swapping process takes $O(logn)$ time, the recursive procedure takes k//2 recursive calls , so the algorithm takes in $O(klogn)$ time.

```
def reverse(D,i,k):
    if k<2:
        return
    x2 = D.delete_at(i+k-1)
    x1 = D.delete_at(i)
    D.insert_at(i+k-1,x1)
    D.insert_at(i, x2)
    reverse(D, i+1, k-2)
```

(b) Use recursion to solve this problem. To move the k-item subsequence starting at i in front of the item at index j, it suffices to move the item A at index i in front of the item B at index j, and recursively move the remainder infront of the item A. As a base case, no work needs to be done if $k = 0$.

If $j < i$, use a variant x to contain the deleted item A at index i, then use insert function to put x in front of index j, then we recursively call the function move(). Because we have moved A in front of index j, and the number of items does not change before index $i + 1$, so we need to move the next item at index $i + 1$, and the total number of

items we want to move become $k-1$, the next move requires us to move the item in front of the item A, whose index have become $j+1$, so the move() function goes like this: $move(D, i+1, k-1, j+1)$.
If $j > i$, use a variant x to contain the deleted item A at index i, then use insert function to put x in front of index j, then we recursively call the function move(). Because we have moved A in front of index j, there is one less item before original index j, so before the insert function, change j into $j-1$. When we use the move() function, we do not need to change index i. But we need to change j into $j+1$ because we minus 1 before, and the total number of items we want to move becomes $k-1$. So the move function goes like this: $move(D, i, k-1, j+1)$.
The problem has assumed that the expression $i \leq j < i+k$ is false, so we have discussed all the possible situations, that makes the algorithm correct.
As for the running time, the delete and insert steps only cost $O(logn)$ time, and the recursive call takes no more than k steps, so the total running time is $O(klogn)$.

```
def move(D,i,k,j):
    if k<1:
        return
    if i>j:
        x = D.delete_at(i)
        D.insert_at(j,x)
        move(D,i+1,k-1,j+1)
    if i<j:
        x = D.delete_at(i)
        j = j-1
        D.insert_at(j,x)
        move(D,i,k-1,j+1)
```

Problem 1-3. Use a dynamic array of size $3n$ to store the pages. To build the array:
1. Place items before bookmark A in the first n places, leaving some empty places,we name these n items P_1
2. Place items between bookmark A and bookmark B starting from index n, we name these n items P_2
3. Place items after bookmark B starting from index 2n, we name these items P_3
Build a dynamic array of n items costs $O(n)$ time, so build a dynamic array of $3n$ items costs $O(n)$ time too.
If we want to place a bookmark between index i and $i + 1$ and these indices are in P_1, we need to move items from index $i+1$ to the last non-empty item in P_1 to the head of P_2, we use $delete\_last$ and $insert\_last$ function cost $O(n)$ time.

Problem 1-4.

(a)

(b)

(c)

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