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 3 dynamic arrays to store the pages. We name the first array P_1 which contains n_1 elements and empty slots at the end, the second array P_2 which has empty slots at both ends and contains n_2 elements, the third array P_3 which contains n_3 elements and empty slots at the front.
Building 1 dynamic array of n items costs $O(n)$ time. Note that $n_1 + n_2 + n_3 = n$, so building 3 dynamic arrays as described above costs $O(n)$ time when $n = |x|$. We can also re-build in $O(n)$ time whenever $place\_mark(i, m)$ is called.
We will maintain that P_1, P_2, P_3 are stored contiguously. We also maintain four indices with semantic invariants: a_1 pointing to the end of P_1, a_2 pointing to the first non-empty item in P_2, b_1 pointing to the end of P_2 and b_2 pointing to the first non-empty element in P_3.
If we execute the operation $read\_page(i)$, there are 3 cases: either i is the index of a page in $P_1, P_2, or P_3$.

- if $i < n_1$, the page is in P_1, we return $P_1[i]$.
- if $n_1 \leq i < n_1 + n_2$, the page is in P_2, we return $P_2[i - n_1 + a_2]$.
- if $n_1 + n_2 \leq i$,the page is in P_3, we return $P_3[i - n_1 - n_2 + b_2]$

This algorithm returns the correct page as long as the invariants on the stored indices are maintained, and returns in worst-case $O(1)$ time because it only contains some arithmetic operations and 1 array index look up.
The operation $shift\_mark(m, d)$ moves the bookmark forward or backward one page. That means move the page at one of indices (a_1, a_2, b_1, b_2) to the index location $(a_2 - 1, a_1 + 1, b_2 - 1, b_1 + 1)$ respectively. This algorithm maintains the invariants of the data structure so is correct, and runs in $O(1)$ time because the insert/delete_ last operation in P_1, insert/delete_ last/first operation in P_2 and insert/delete_ first operation in P_3 are in $O(1)$ time. If there is no empty slot in either array, rebuild it costs amortized $O(1)$ time. In conclusion, this operation runs in amortized $O(1)$ time.
The operation $move\_page(m)$ moves the page at one of indices(a_1, b_1) to the index location $(b_1 + 1, a_1 + 1)$ respectively. The delete/insert_ last operation in P_1 and P_2 cost $O(1)$ time. If there is no empty slot in either array, rebuild it costs amortized $O(1)$ time. In conclusion, this operation runs in amortized $O(1)$ time.

Problem 1-4.

(a) The following algorithms run in $O(1)$ time because they only contain building 1 node and linking a constant number of pointers.

- $insert\_first(x)$: Create a new node a storing item x. If the doubly linked list is empty, we link the head and the tail pointer to node a. Otherwise, assume the origin head node is b, we link a's next pointer to b, and b's previous pointer to a, and set the head pointer to a.
- $insert\_last(x)$: Create a new node a storing item x. If the doubly linked list is empty, we link the head and the tail pointer to node a. Otherwise, assume the origin last node is b, we link a's previous pointer to b, and b's next pointer to a, then set the tail pointer to a.
- $delete\_first()$: If there is only 1 node in the given doubly linked list, use a variant x to store the value of the node, delete the single node, and set the head and tail pointer to none, then return x. If there is more than 1 node, assume the first node is a and the second node b. Use a variant x to store the value of node a, delete node a, and set the head pointer to node b. Set the previous pointer of node b to none and return x.
- $delete\_last()$: If there is only 1 node in the given doubly linked list, use a variant x to store the value of the node, delete the single node, and set the head and tail pointer to none, then return x. If there is more than 1 node, assume the last node is a and the penultimate node b. Use a varian x to store the value of node a, delete node a, and set the tail pointer to b. Set the next pointer of node b to none and return x.

(b) Construct a new doubly linked list called L_1. Assume the node previous to x_1 is a and the node next to x_2 is b. Set L_1's head pointer to x_1, delete x_1's previous pointer and set a's next pointer to b and set b's previous pointer to a. Set x_2's next pointer to none, and set L_2's tail pointer to x_2. If x_1 is the head of L, set L's head pointer to b. If x_2 is the tail of L, set a's next pointer to none and set L's tail pointer to a.
This algorithm deletes the nodes from x_1 to x_2 directly so is correct. It runs in $O(1)$ time because the number of operations is constant.

(c) Assume x_1 is the head node of L_2 and x_2 the tail node, and a the next node after x in L_1. First we delete the head and tail pointer in L_2, then we reset x's next pointer in L_1 to x_1 and x_1's previous pointer to x in L_1. Then we set x_2's pointer to a and a's previous pointer to x_2. If x is the tail node in L_1, then reset L_1's tail pointer to x_2.
The algorithm deletes the head and tail pointer of L_2 and insert every element in L_2 into L_1. So after the slice operation, L_2 is empty, and L_1 contains all items in L_2, thus this algorithm is correct and it takes a constant number of resetting pointers, so it runs in $O(1)$ time.

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