Problem Set 1

Name: Kim Heesuk

Problem 1-1.

(a) $f_1 = \log n^n = n \log n \in O(f_2)$, $f_3 = \log n^{6006} = 6006 \log n \in \Theta(\log n)$. So that $f_3 \in O(f_1)$. Meanwhile $f_4 \in O(f_2)$, and $f_5 \in O(f_3)$, the answer is $(f_5, f_3, f_1, f_4, f_2)$.

(b) $f_2 \in O(f_5)$ and $f_5 \in O(f_4)$. Also $f_1 \in O(f_2)$, $f_5 \in O(f_3)$. For f_3 and f_4,

$$\begin{aligned}\log \frac{f_3}{f_4} &= \log \frac{2^{6006^n}}{6006^{2^n}} \\ &= 6006^n \log 2 - 2^n \log 6006 \\ &= 2^n(3003^n \log 2 - \log 6006) \to \infty\end{aligned} \tag{1}$$

$f_4 \in O(f_3)$ and the answer is $(f_1, f_2, f_5, f_4, f_3)$.

(c) Firstly,

$$\begin{aligned}f_2 = \binom{n}{n-6} &= \frac{n!}{6!(n-6)!} \\ &\in \Theta\left(n(n-1)\ldots(n-)\right) \\ &= \Theta(n^6),\end{aligned} \tag{2}$$

so $f_5 = n^6 \in \Theta(f_2)$. For f_4, by the Stirling's approximation,

$$\begin{aligned}\log f_4 = \log \binom{n}{n/6} &= \log \frac{n!}{(n/6)!(5n/6)!} \\ &\sim \log \frac{\sqrt{2\pi n}(n/e)^n}{\sqrt{2\pi n}(n/6e)^{n/6}\sqrt{2\pi n}(5n/6e)^{5n/6}} \\ &= \log \frac{(6/5^{5/6})^n}{\sqrt{2\pi n}} \\ &\in \Theta\left(n \log(6/5^{5/6}) - \frac{1}{2}(\log n + \log 2\pi)\right) = \Theta(n).\end{aligned} \tag{3}$$

since $6/5^{5/6} > 1$. Hence $f_2(\in \Theta(f_5)) \in O(f_4)$, $f_4 \in O(f_1)$ since $f_4 \in \{2^p | p \in \Theta(n)\} \subset \Omega(n^6)$ and $\log f_1 \in \Theta(n \log n)$. Also by the Stirling's approximation,

$$\begin{aligned}f_3 = (6n)! &\sim \sqrt{12\pi n}(6n/e)^{6n} \\ &\in \Theta((6n)^{6n}) \subset \Omega(n^n) = \Omega(f_1).\end{aligned} \tag{4}$$

Thus the answer is $(\{f_2, f_5\}, f_4, f_1, f_3)$.

(d) Using the Stirling's approximation, $f_1 \sim n^{n+4} + \sqrt{2\pi n}(n/e)^n \in \Theta(n^{n+4})$. Also considering $f_5/n^{12} = n^{1/n} \to 1$, $f_5 = n^{12+1/n} \sim n^{12} \in \Theta(n^{12})$. It is obvious that $f_2 = n^{7\sqrt{n}} \in O(n^{n+4}) = O(f_1)$ and $f_3 = 4^{3n\log n} \in O(7^{n^2}) = O(f_4)$. Finally for f_3 and $n^{n+4} \in \Omega(f_1)$,

$$\begin{aligned} \log \frac{f_3}{n^{n+4}} &= \log \frac{4^{3n\log n}}{n^{n+4}} \\ &= 3n \log n \cdot \log 4 - (n+4)\log n \\ &= ((3\log 4 - 1)n - 4)\log n \to \infty, \end{aligned} \tag{5}$$

$f_3 \in O(n^{n+4}) = O(f_1)$. The order should be $(f_5, f_2, f_1, f_3, f_4)$.

Problem 1-2.

(a) $T(k) = O(\log n) + T(k-1)$, $T(k) = O(k \log n)$

```
def reverse(D, i, k): # T(k)
  if k < 2:
    return
  D.insert_at(i + k - 1, D.delete_at(i))      # O(log(n))
  reverse(D, i, k - 1)                        # T(k - 1)
```

If we call `reverse(D, i, 1)`, it returns and it is correct. Assume `reverse(D, i, k)` works right, `reverse(D, i, k + 1)` should work right because it deletes element of i'th index, and insert it to $i + k$ (next to the pre-deletion index $i + k$) and reverses k items starting at index i of `D`, which resulting reverse $k + 1$ items starting at index i.

(b) $T(k) = O(\log n) + O(\log n) + T(k-1)$, $T(k) = O(k \log n)$

```
def move(D, i, k, j):           #T(k)
  if k < 1:
    return
  temp = D.delete_at(i)         # O(log(n))
  if j < i:
    D.insert_at(j + 1, temp)    # O(log(n))
  else:
    D.insert_at(j, temp)        # O(log(n))
  move(D, i, k - 1, j)          # T(k-1)
```

Note that when $j \geq i + k$ (which can be just $\neg(j < i)$ since $i \leq j < i + k$ is false), `D.delete_at(i)` makes the index of element with pre-deletion index j be $j - 1$.

Problem 1-3. build(X) constructs a static array with size |X| and store pages in that array. Also, it initializes some fields to provide operations for bookmarks.

- A index of page placed on in front of bookmark A
- B index of page placed on in front of bookmark B
- array_index initialized by (A + B)/2 when place_mark operates
- array_front Array containing array_index - 1'th page to first bookmark's front page in reverse order
- array_behind Array containing array_index'th page to second bookmark's front page

Pages between the bookmarks will be stored in separate arrays array_front and array_behind on which place_mark(i, m) called.

When there is no bookmark placed, place_mark(i, m) just set field m to i. There are three possible cases left.

1. One bookmark is placed and call place_mark(i, m) for that bookmark
2. One bookmark is placed and call place_mark(i, m) for other bookmark
3. All bookmarks are placed

For each cases, set field m to i.

For second case, lets assume $\texttt{B} > \texttt{A}$. Set array_index to (A + B)/2 and construct dynamic array with size $2 * (\texttt{B} - \texttt{array\_index})$ for array_behind, and move page from index array_index to B here. For array_front, do same thing with size $2*(\texttt{array\_index}-\texttt{A})$. If $\texttt{B} = \texttt{A}$, construct two empty dynamic arrays for $\texttt{A} = \texttt{B} = \texttt{array\_index}$ with certain non-zero fixed size. This is needed to operate shift_mark correctly.

For third case, if such of a operation expands array_front or array_behind, if size of an array does not exceed the capacity of array, it could be just moving the elements. Else, each dynamic array can be expanded with twice of previous capacity. If it reduces but does not cross the array_index on either side, just setting m to i is sufficient. If it crosses, re-allocate array_front and array_behind with same method for second case with new $\texttt{array\_index} = \texttt{(A + B)/2}$.

When reducing, note each deleted elements should be updated to the origianl array. Still, the original array can be static array since there is no page insertions in given operation set. Time complexitiy for this operation is $O(n)$ in worst case and amortized.

Assuming $\texttt{A} < \texttt{B}$, read_page(i) returns the i'th index of original array if it is not between to the two bookmarks. (i.e. $\neg(\texttt{A} \leq i \leq \texttt{B})$) If it is, return sufficient page from array_behind or array_front due to the array_index and i. Note that array_front is inversed. It consumes constant time on either cases since each access are array access, so its $O(1)$ in both worst case and amortized.

On `shift_mark(m, d)`, the method is same as third case for `place_mark(i, m)` so we can just call this function with $\texttt{i} = \texttt{m} + \texttt{d}$. Note `m` denotes the field `A` or `B` which tracks index of page placed on in front of bookmark. Since $\texttt{d} \in -1, 1$, most operation would be done in constant time when it does not exceed the capacity or `array_index` in reduction. When the realocate happend in each case, the time consumption would be $O(n)$ however it can be said that the running time of operation is $O(1)$ in amortized case since it is rarely happend.

Lastly, `move_page(m)`. If bookmarks are placed in same place or one or zero bookmark placed on dataset, this operation does nothing. Else, if it is called with the second bookmark(i.e. index of its front page is larger), delete the last element of `array_behind` and append it to the end of `array_front`. In deletion, if it cross the `array_index` reallocate be done mentioned above. also, in insertion, also if it exceeds the capacity, it should reallocate `array_front` as mentioned above. For the first one, do it oppositly. Same as `shift_mark` this operation consumes $O(n)$ when reallocation happen, however it can be said that the running time is $O(1)$ in amortized case since it is rarely happend.

Problem 1-4.

(a)
- `insert_first(x)` If list is empty (i.e. `L.tail` is `None`), set `L.tail` to `x` first. And then set `x.next` as `L.head`, `x.prev` to `None`, and set `L.head` to `x`.
- `insert_last(x)` If list is empty (i.e. `L.head` is `None`), set `L.head` to `x` first. And then set `x.next` to `None`, `x.prev` to `L.tail`, and set `L.tail` to `x`.
- `delete_first()` If list is empty (i.e. `L.head` is `None`), return. Else, set `L.head` to `L.head.next`. And then, set `L.head.prev` to `None`. If the `L.head.next` was `None` (i.e. there was only one node), set `L.tail` to `None` too.
- `delete_last()` If list is empty (i.e. `L.head` is `None`), return. Else, set `L.tail` to `L.tail.prev`. And then, set `L.tail.next` to `None`. If the `L.tail.prev` was `None` (i.e. there was only one node), set `L.head` to `None` too.

(b) Set `x_1.next` as `x_2.next`, and then set `x_1.next.prev` as `x_1`. If `x_2.next` was `None`, (i.e. `x_2` was the tail of the list) set `L.tail` as `x_1` too.

(c) If `L_2.head` is `None` (i.e. `L_2` is empty), do nothing and return.

Firstly, set `L_2.tail.next` as `x.next`. And then, set `x.next` to `L_2.head`, `L_2.head.prev` to `x`. Now set L_2's head and tail to `None`.

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