Ravi Dayabhai

Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Erik Demaine, Jason Ku, and Justin Solomon — Problem Set 1

Problem Set 1

Please write your solutions in the LaTeX and Python templates provided. Aim for concise solutions; convoluted and obtuse descriptions might receive low marks, even when they are correct.

18/20

Problem 1-1. [20 points] **Asymptotic behavior of functions**

For each of the following sets of five functions, order them so that if f_a appears before f_b in your sequence, then $f_a = O(f_b)$. If $f_a = O(f_b)$ and $f_b = O(f_a)$ (meaning f_a and f_b could appear in either order), indicate this by enclosing f_a and f_b in a set with curly braces. For example, if the functions are:

$$f_1 = n, \qquad f_2 = \sqrt{n}, \qquad f_3 = n + \sqrt{n},$$

the correct answers are $(f_2, \{f_1, f_3\})$ or $(f_2, \{f_3, f_1\})$.

Note: Recall that a^{b^c} means $a^{(b^c)}$, not $(a^b)^c$, and that $\log$ means $\log_2$ unless a different base is specified explicitly. Stirling's approximation may help for comparing factorials.

a)	b)	c)	d)
$f_1 = \log(n^n)$	$f_1 = 2^n$	$f_1 = n^n$	$f_1 = n^{n+4} + n!$
$f_2 = (\log n)^n$	$f_2 = 6006^n$	$f_2 = \binom{n}{n-6}$	$f_2 = n^{7\sqrt{n}}$
$f_3 = \log(n^{6006})$	$f_3 = 2^{6006^n}$	$f_3 = (6n)!$	$f_3 = 4^{3n \log n}$
$f_4 = (\log n)^{6006}$	$f_4 = 6006^{2^n}$	$f_4 = \binom{n}{n/6}$	$f_4 = 7^{n^2}$
$f_5 = \log\log(6006n)$	$f_5 = 6006^{n^2}$	$f_5 = n^6$	$f_5 = n^{12+1/n}$

1 $\boxed{A}$ $\{f_5, f_3, f_4, f_1, f_2\}$ ✓

$\boxed{B}$ $\{f_1, f_2, f_5, f_4, f_3\}$ ✓

$\boxed{C}$ $\{\{f_2, f_5\}, f_4, f_1, f_3\}$ ✓

1 $\boxed{D}$ $\{f_5, f_2, \cancel{\{f_3, f_1\}}, f_4\}$ ✓ ✓ ✓

$, f_1, f_3,$

$f_1 = O(f_3)$

but $f_3 \neq O(f_1)$

* Asymptotically similar logarithms $\not\Rightarrow$ functions are asymptotically the same!

Problem 1-2. [16 points] Given a data structure `D` that supports Sequence operations:

- `D.build(X)` in $O(n)$ time, and
- `D.insert_at(i, x)` and `D.delete_at(i)`, each in $O(\log n)$ time,

(SEE NEXT PAGE)

where n is the number of items stored in `D` at the time of the operation, describe algorithms to implement the following higher-level operations in terms of the provided lower-level operations. Each operation below should run in $O(k \log n)$ time. Recall, `delete_at` returns the deleted item.

(a) `reverse(D, i, k)`: Reverse in `D` the order of the k items starting at index i (up to index $i + k - 1$).

(b) `move(D, i, k, j)`: Move the k items in `D` starting at index i, in order, to be in front of the item at index j. Assume that expression $i \leq j < i + k$ is false.

Problem 1-3. [20 points] **Binder Bookmarks**

(SEE PAGE AFTER NEXT)

Sisa Limpson is a very organized second grade student who keeps all of her course notes on individual pages stored in a three-ring binder. If she has n pages of notes in her binder, the first page is at index 0 and the last page is at index $n - 1$. While studying, Sisa often reorders pages of her notes. To help her reorganize, she has two bookmarks, A and B, which help her keep track of locations in the binder.

Describe a database to keep track of pages in Sisa's binder, supporting the following operations, where n is the number of pages in the binder at the time of the operation. Assume that both bookmarks will be placed in the binder before any shift or move operation can occur, and that bookmark A will always be at a lower index than B. For each operation, state whether your running time is worst-case or amortized.

`build(X)`	Initialize database with pages from iterator `X` in $O(\lvert X \rvert)$ time.
`place_mark(i, m)`	Place bookmark `m` $\in \{A, B\}$ between the page at index `i` and the page at index `i` $+ 1$ in $O(n)$ time.
`read_page(i)`	Return the page at index `i` in $O(1)$ time.
`shift_mark(m, d)`	Take the bookmark `m` $\in \{A, B\}$, currently in front of the page at index i, and move it in front of the page at index $i +$ `d` for `d` $\in \{-1, 1\}$ in $O(1)$ time.
`move_page(m)`	Take the page currently in front of bookmark `m` $\in \{A, B\}$, and move it in front of the other bookmark in $O(1)$ time.

needs to be array-based

only need O(1) for adjacent pages!

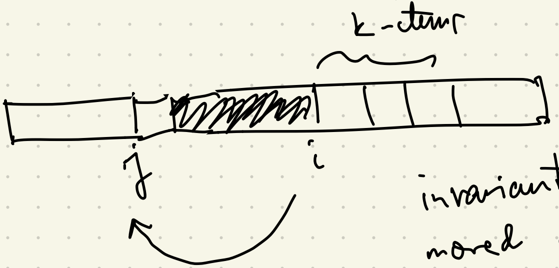
must be multiple arrays otherwise inserts/deletes will be O(n) ≠ O(1)

② $\boxed{A}$ Delete the $(i+k-1)^{th}$ item, saving to `temp` variable. Insert `temp` at i^{th} location. Repeat a total of $k-1$ times (i.e., until the original item at i is at position $(i+k-1)$).

This algorithm takes $O((k-1)[2\log n + 1]) = O(k \log n)$ time: $(k-1)$ iterations, each iteration calling 2 $O(\log n)$ functions + 1 $O(1)$ assignment.

$\boxed{B}$ Delete the i^{th} item, saving to `temp` variable. Insert `temp` at j^{th} location. Repeat a total of k times (i.e., until the original item at location $(i+k-1)$ is at position j).

Similar to `reverse` algorithm above, this algorithm iterates k times, doing 2 $O(\log n)$ operations (and 1 $O(1)$ assignment) each iteration. $\Rightarrow O(k \log n)$

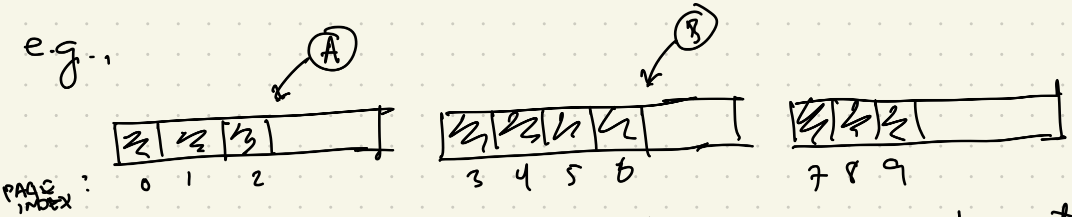


invariant: ordering of moved elements

(3) Consider a data structure composed of 3 dynamic arrays where:

1. the first list holds all pages [0, A)
2. " " second " " (A, B)
3. " " third " " (B, n-1]

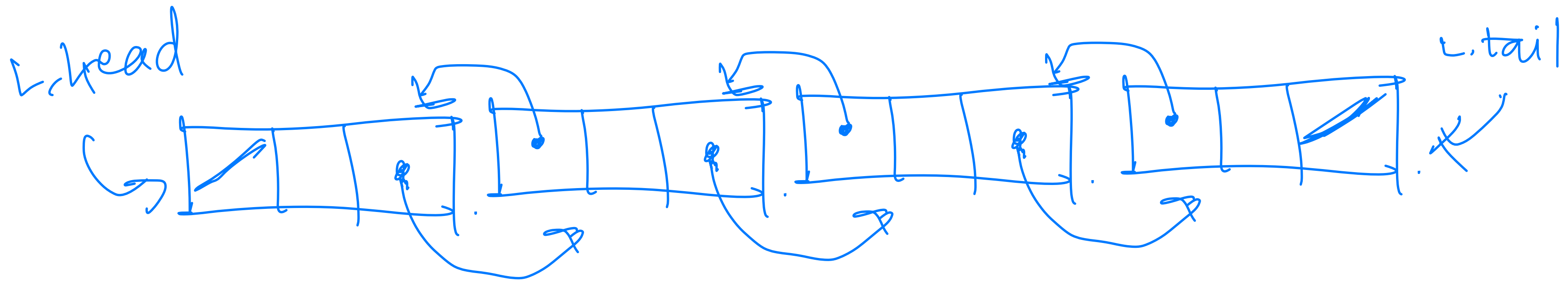
e.g.,



- build would take O(n) worst case time to put all pages into the first list
- place_mark would take O(n) worst case time popping from the first array and appending into the appropriate subsequent array
- read_page is O(1) worst case time by property indexing into appropriate list (read: pointer arithmetic)
- shift_mark would take $O(1)_a$ since an insertion/deletion of a page in a list could cause resizing after O(n) such operations had already occurred.
- move_page would take $O(1)_a$ time (for same reason as shift_mark)

15+/20

Right idea!



Problem 1-4. [44 points] **Doubly Linked List**

In Lecture 2, we described a singly linked list. In this problem, you will implement a **doubly linked list**, supporting some additional constant-time operations. Each node `x` of a doubly linked list maintains an `x.prev` pointer to the node preceeding it in the sequence, in addition to an `x.next` pointer to the node following it in the sequence. A doubly linked list `L` maintains a pointer to `L.tail`, the last node in the sequence, in addition to `L.head`, the first node in the sequence. For this problem, doubly linked lists **should not maintain their length**.

(a) [8 points] Given a doubly linked list as described above, describe algorithms to implement the following sequence operations, each in $O(1)$ time.

`insert_first(x)` `insert_last(x)` `delete_first()` `delete_last()`

(b) [5 points] Given two nodes x_1 and x_2 from a doubly linked list L, where x_1 occurs before x_2, describe a constant-time algorithm to **remove** all nodes from x_1 to x_2 inclusive from L, and return them as a new doubly linked list.

(c) [6 points] Given node x from a doubly linked list L_1 and second doubly linked list L_2, describe a constant-time algorithm to **splice** list L_2 into list L_1 after node x. After the splice operation, L_1 should contain all items previously in either list, and L_2 should be empty.

(d) [25 points] Implement the operations above in the `Doubly_Linked_List_Seq` class in the provided code template; **do not modify** the `Doubly_Linked_List_Node` class. You can download the code template including some test cases from the website.

(SEE CODE)

④ [A]

```
def insert_first(x):
    L.head = Node(x, None, L.head)      } O(1)
    L.head.next.prev = L.head

def insert_last(x):
    L.tail = Node(x, L.tail, None)      } O(1)
    L.tail.prev.next = L.tail

def delete_first():
    L.head = L.head.next                } O(1)
    L.head.prev = None

def delete_last():
    L.tail = L.tail.prev                } O(1)
    L.tail.next = None
```

* correct, but doesn't address edge case if doubly-LL is empty or deletions empty the DLL.

5+/8

[B]

```
def remove(x1, x2):
    L2 = L(x1, x)                                          O(1)

    left, right = x1.prev, x2.next                         O(1)
    if left:  left.next = right   else  L.head = right     O(1)
    if right: right.prev = left   else  L.tail = left      O(1)

    return L2         overall: O(1)
```

4/5

need to reassign L.head, L.tail in case where left or right are ends

✓ ④ [c] def splice(x, L2):

```
    L2.head.prev = x                    # O(1)
    if x.next:
        L2.tail.next = x.next           # O(1)
        x.next.prev = L2.tail           # O(1)
    else:
        L.tail = L2.tail                # O(1)
    x.next = L2.head                    # O(1)
    L2.head = L2.tail = None            # O(1)
```