Ravi Dayabhai

Introduction to Algorithms: 6.006
Massachusetts Institute of Technology
Instructors: Erik Demaine, Jason Ku, and Justin Solomon — Problem Set 2

Problem Set 2

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.

Problem 2-1. [15 points] **Solving recurrences**

Derive solutions to the following recurrences. A solution should include the tightest upper and lower bounds that the recurrence will allow. Assume $T(1) = \Theta(1)$.

Solve parts (a), (b), and (c) in **two ways**: drawing a recursion tree **and** applying Master Theorem. Solve part (d) **only by substitution**.

(a) [4 points] $T(n) = 4\,T(\frac{n}{2}) + O(n)$

(b) [4 points] $T(n) = 3\,T(\frac{n}{\sqrt{2}}) + O(n^4)$

(c) [4 points] $T(n) = 2\,T(\frac{n}{2}) + 5n \log n$

(d) [3 points] $T(n) = T(n-2) + \Theta(n)$

(SEE NEXT FEW PAGES)

Problem 2-2. [15 points] **Sorting Sorts**

For each of the following scenarios, choose a sorting algorithm (from either selection sort, insertion sort, or merge sort) that best applies, and justify your choice. **Don't forget this! Your justification will be worth more points than your choice.** Each sort may be used more than once. If you find that multiple sorts could be appropriate for a scenario, identify their pros and cons, and choose the one that best suits the application. State and justify any assumptions you make. "Best" should be evaluated by asymptotic running time.

(a) [5 points] Suppose you are given a data structure `D` maintaining an extrinsic order on n items, supporting two standard sequence operations: `D.get_at(i)` in worst-case $\Theta(1)$ time and `D.set_at(i, x)` in worst-case $\Theta(n \log n)$ time. Choose an algorithm to best sort the items in `D` **in-place**.

selection

(b) [5 points] Suppose you have a static array A containing pointers to n comparable objects, pairs of which take $\Theta(\log n)$ time to compare. Choose an algorithm to best sort the pointers in A so that the pointed-to objects appear in non-decreasing order.

merge

(c) [5 points] Suppose you have a **sorted array** A containing n integers, each of which fits into a single machine word. Now suppose someone performs some $\log \log n$ swaps between pairs of adjacent items in A so that A is no longer sorted. Choose an algorithm to best re-sort the integers in A.

insertion

2 A Because the sort must occur in-place, we must choose between selection & insertion sorts. The former is better for sorting D because in the worst case only $O(n)$ swaps will need to be made (where a swap is $O(n \log n)$), whereas for insertion sort, these expensive swaps occur $\Omega(n^2)$ times in the worst case $\Rightarrow O(n^2 \log n)$ from the swaps dominating the costs of comparisons.

2 B Merge sort is the best choice because the comparison operation is expensive ($O(\log n)$) and merge sort does $\Theta(n \log n)$ comparisons in the worst case $\Rightarrow$ $T(n) = 2T(\frac{n}{2}) + O(n \log n) = \Theta(n \log^2 n)$

2 C Insertion sort is best suited for this task because most of the array remains sorted and only $\log \log n$ $O(1)$ swaps will be needed to resort the array and the cost to make comparisons is $O(1)$ $\Rightarrow$ $O(n)$. from the cost of comparisons

e.g.,

1	2	3	5	4	6	7	8	9	10	12	13	11	14	15	16

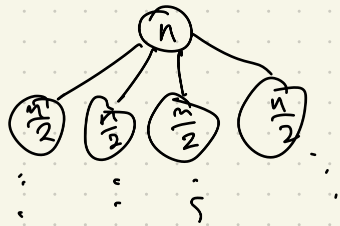
① $\boxed{A}$ $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$

By master method: $a = 4,\ b = 2,\ d = 1 \Rightarrow a > b^d$, so the work done at leaves of recursion tree dominate:

$O(n^{\log_b a}) = O(n^{\log_2 4}) = O(n^2)$. For lower bound,

let $T(n) \geq 4T\left(\frac{n}{2}\right) + O(n^0) \Rightarrow a > b^d \Rightarrow$

$\Omega(n^{\log_b a}) = \Omega(n^2) \Rightarrow \Theta(n^2)$



By recursion tree:

$$T(n) \leq \sum_{i=0}^{\log_2 n} 4^i \left(\frac{cn}{2^i}\right)$$

$$T(n) \leq cn \cdot \underbrace{\sum_{i=0}^{\log_2 n} 2^i}_{} = O(n \cdot 2^{\log_2 n}) = O(n^2)$$

//

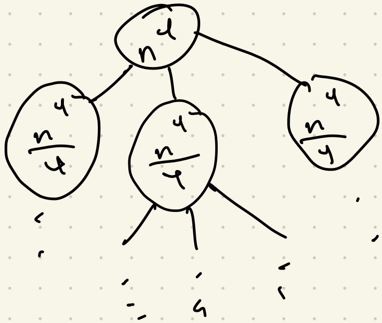
$$T(n) = \frac{1 - 2^{\log_2 n + 1}}{1 - 2} = cn(2n) = 2cn^2 \approx \Theta(n^2)$$

The lower bound is also $\Omega(n^2)$ because $\Theta(1)$ work is done across the cn^2 leaves. $\Rightarrow \Theta(n^2)$

① $\boxed{3}$ $T(n) \leq 3T\left(\frac{n}{\sqrt{2}}\right) + O(n^4)$

By Master Method: $a = 3,\ b = \sqrt{2},\ d = 4 \Rightarrow a < b^d$

The work done in recursion tree is dominated by the root node: $O(n^4)$.



By recursion tree:

$$T(n) = cn^4 \sum_{i=0}^{L} 3^i \cdot \left(\frac{1}{\sqrt{2}^4}\right)^i$$

where L is the number of levels in the recursion tree: $\log_{\sqrt{2}} n$.

$$T(n) = cn^4 \sum_{i=0}^{2\log_2 n} \left(\frac{3}{4}\right)^i \Rightarrow O(n^4)$$ as the geometric series is dominated by the first term. The lower bound is given by work done in the $3^{2\log_2 n}$ leaves (with $\Theta(1)$ work done per leaf): $\Omega(n^{2\log_2 3})$

Yes, correct bounds, <u>but</u> you should revisit formal application of Master Theorem for lower asymptotic bounds!

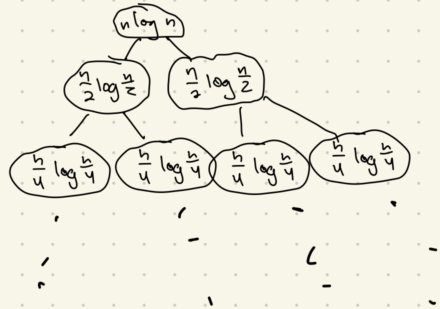
1 [c] $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n \log n)$

By Master Method: CASE 2, because $g(n) =$

$\Theta(n^{\log_2 2} \log^k n)$ for $k \geq 0$, in our case $k = 1 \Rightarrow$

$T(n) = \Theta(n^{\log_2 2} \log^{k+1} n) = \Theta(n \log^2 n)$

By recurrence tree:



$$T(n) = \sum_{i=0}^{\log_2 n} \cancel{2^i} \left(\frac{cn}{\cancel{2^i}} \log \frac{n}{2^i}\right)$$

$$= cn \sum_{i=0}^{\log_2 n} \log\left(\frac{n}{2^i}\right)$$

$$= cn \sum_{i=0}^{\log_2 n} \log n - i = cn \sum_{i=0}^{\log_2 n} \log_2 n - cn \sum_{i=0}^{\log_2 n} i$$

$$= cn \log^2 n - cn \frac{(\log_2 n)(\log_2 n + 1)}{2}$$

$$= \frac{1}{2} cn \log^2 n - \frac{1}{2} cn \log_2 n = \Theta(n \log^2 n)$$

4 $T(n) = T(n-2) + \Theta(n)$; guessing $T(n) = \Theta(n^2) \Rightarrow$

$$\cancel{cn^2} = c(n-2)^2 + \Theta(n) = \cancel{cn^2} - 4cn + 4c + \Theta(n)$$

$$\Updownarrow$$

$$4c(n-1) = \Theta(n)$$

$$\Theta(n) = \Theta(n) \Rightarrow$$

guess of $\Theta(n^2) = T(n)$ is correct.

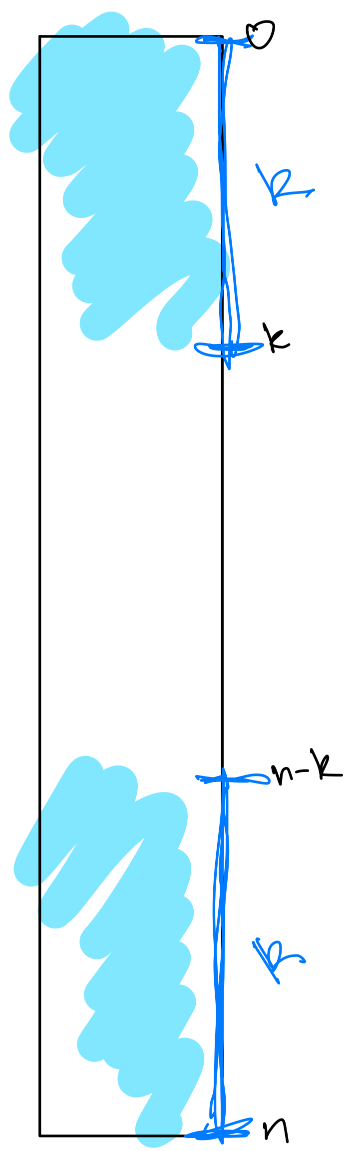
"within arbitrary k of end"

⇕

"at unknown location k from some end"

This would find Datum immediately / presupposes we know where he is already for specific k!

~~① WLOG, start at position k. If device says (↑), binary search [0, k] starting at position $\frac{k}{2}$. Otherwise jump to position $\frac{n-(n-k)}{2} + (n-k)$ and continue binary search of [n-k, n].~~



if Datum in [shaded region], he can be found in O(log k) time

Problem 2-3. [10 points] **Friend Finder** misunderstood question — read carefully!

Jean-Locutus ΠΙcard is searching for his incapacitated friend, Datum, on Gravity Island. The island is a narrow strip running north–south for n kilometers, and ΠΙcard needs to pinpoint Datum's location to the nearest integer kilometer so that he is within visual range. Fortunately, ΠΙcard has a tracking device, which will always tell him whether Datum is north or south of his current position (but sadly, not how far away he is), as well as a teleportation device, which allows him to jump to specified coordinates on the island in constant time.

Unfortunately, Gravity Island is rapidly sinking. The topography of the island is such that the north and south ends will submerge into the water first, with the center of the island submerging last. Therefore, it is more important that ΠΙcard find Datum quickly if he is close to either end of the island, lest he short-circuit. Describe an algorithm so that, if Datum is k kilometers from the nearest end of the island (i.e., he is either at the kth or the $(n-k)$th kilometer, measured from north to south), then ΠΙcard can find him after visiting $O(\log k)$ locations with his teleportation and tracking devices.

Problem 2-4. [15 points] **MixBookTube.tv Chat**

MixBookTube.tv is a service that lets viewers chat while watching someone play video games. Each viewer is identified by a known unique integer ID[1]. The chat consists of a linear stream of messages, each written by a viewer. Viewers can see the most recent k chat messages, where k depends on the size of their screen. Sometimes a viewer misbehaves in chat and gets ***banned*** by the streamer. When a viewer gets banned, not only can they not post new messages in chat, but all of their previously sent messages are removed from the chat.

Describe a database to efficiently implement MixBookTube.tv's chat, supporting the following operations, where n is the number of all viewers (banned or not) in the database at the time of the operation (all operations should be **worst-case**):

`build(V)`	Initialize a new chat room with the $n = \lvert V \rvert$ viewers in `V` in $O(n \log n)$ time.
`send(v, m)`	Send message `m` to the chat from viewer `v` (unless banned) in $O(\log n)$ time.
`recent(k)`	Return the `k` most recent not-deleted messages (or all if < `k`) in $O(k)$ time.
`ban(v)`	Ban viewer `v` and delete all their messages in $O(n_v + \log n)$ time, where n_v is the number of messages that viewer `v` sent before being banned.

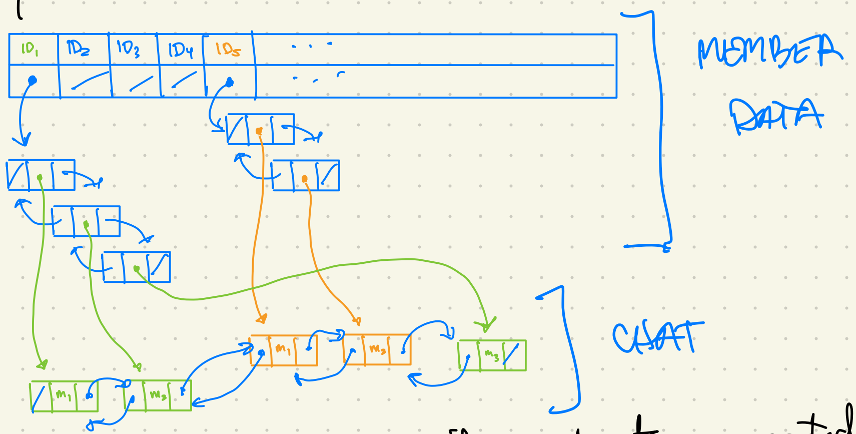
(SEE NEXT PAGE)

(Officially solution uses singly linked list for given member's messages, whereas I used doubly LL. No real impact on time complexity, though.)

[1] As mentioned in lecture, unless we parameterize the size of numbers in our input, you should assume that input integers each fit within a machine word, so pairs of them may be compared in constant time.

4b Our data structure will consist of two primary components:

- a sorted array of member IDs, where each element points to a doubly-linked list of pointers to chat messages
- a doubly-linked list of chat messages (in chronological order); those messages are pointed to by the nodes in the doubly-linked lists associated w/ each member



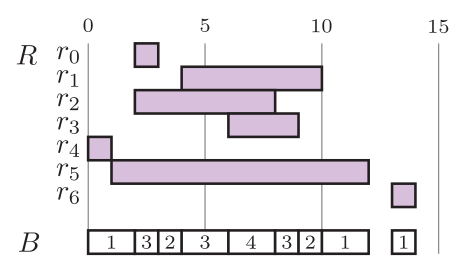
- 'build' takes O(nlog n) because it constructs a sorted array
- 'send' takes O(log n) from the binary search; the append takes constant time because we keep tail pointer
- 'recent' takes O(k) by traversing the chat doubly-LL backwards k nodes from tail
- 'ban' takes $O(n_v + \log n)$ time from the member lookup in $O(\log n)$ time and $O(n_v)$ constant time operations (deleting node in chat + relinking is O(1) operation). Unsetting pointer in sorted array is also O(1) operation.

Problem 2-5. [45 points] **Beaver Bookings**

Tim the Beaver is arranging **Tim Talks**, a lecture series that allows anyone in the MIT community to schedule a time to talk publicly. A ***talk request*** is a tuple (s, t), where s and t are the starting and ending times of the talk respectively with $s < t$ (times are positive integers representing the number of time units since some fixed time).

Tim must make room reservations to hold the talks. A ***room booking*** is a triple (k, s, t), corresponding to reserving $k > 0$ rooms between the times s and t where $s < t$. Two room bookings (k_1, s_1, t_1) and (k_2, s_2, t_2) are ***disjoint*** if either $t_1 \leq s_2$ or $t_2 \leq s_1$, and ***adjacent*** if either $t_1 = s_2$ or $t_2 = s_1$. A ***booking schedule*** is an ordered tuple of room bookings where: every pair of room bookings from the schedule are disjoint, room bookings appear with increasing starting time in the sequence, and every adjacent pair of room bookings reserves a different number of rooms.

Given a set R of talk requests, there is a unique booking schedule B that ***satisfies*** the requests, i.e., the schedule books exactly enough rooms to host all the talks. For example, given a set of talk requests $R = \{(2,3), (4,10), (2,8), (6,9), (0,1), (1,12), (13,14)\}$ pictured to the right, the satisfying room booking is:



$$B = ((1,0,2), (3,2,3), (2,3,4), (3,4,6), (4,6,8), (3,8,9), (2,9,10), (1,10,12), (1,13,14)).$$

NICE.

(a) [15 points] Given two booking schedules B_1 and B_2, where $n = |B_1| + |B_2|$ and B_1 and B_2 are the respective booking schedules of two sets of talk requests R_1 and R_2, describe an $O(n)$-time algorithm to compute a booking schedule B for $R = R_1 \cup R_2$.

(b) [5 points] Given a set R of n talk requests, describe an $O(n \log n)$-time algorithm to return the booking schedule that satisfies R.

Needed hint to use merge sort!

(c) [25 points] Write a Python function `satisfying_booking(R)` that implements your algorithm. You can download a code template containing some test cases from the website.

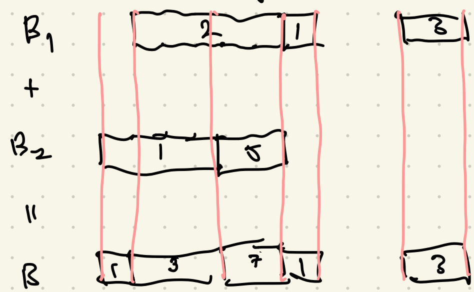
```
def satisfying_booking(R):
    '''
    Input:  R | Tuple of |R| talk request tuples (s, t)
    Output: B | Tuple of room booking triples (k, s, t)
              | that is the booking schedule that satisfies R
    '''
    B = []
    ##################
    # YOUR CODE HERE #
    ##################
    return tuple(B)
```

(SEE CODE)

(5) [A] We can leverage the fact that all bookings in a booking schedule are disjoint. Visually we want to partition the time spanned by B_1 & B_2 by using a greedy approach:

$B_1 :=$ (2, 1, 6)
(1, 6, 7)
(3, 10, 12)

$B_2 :=$ (1, 0, 4)
(5, 4, 6)



EXPECTED

start	end	k
0	1	1
1	4	3
4	6	7
6	7	1
10	12	3

We need to form a booking in B whenever a "boundary" (see black vertical lines) is reached in either B_1 or B_2. We proceed by initializing pointers $i, j = 0$. The particular booking being indexed is given by resp. pointer // 2, and the "boundary" is resp. pointer % 2 + 1. By assigning the minimum as either 'start' or end for a booking in B (alternating between 'start' and 'stop' and incrementing the pointer pointing to the minimum and recomputing the indices as described above. To get the # of rooms, k, whenever we have a 'start' & 'end', we check to see (1) if 'end' > the start of the booking we're indexed into B_1, then k += # of rooms in this booking (2) if 'end' > the start of the booking we've indexed into in B_2, then k += # of rooms in this booking. We can append this tuple (k, start, end) to B and set start = end, end = None, k = 0; note if k = 0 after checking (1), (2), we need not add this tuple to B. If a pointer results in index beyond its enclosing booking schedule, it can be set to ∞ and (1) (or (2) depending on whether B_1 or B_2 is exhausted) resolves to false hence.

This algorithm runs in O(n) time since it operates until all "boundaries" are processed, which is at most 2n.