Problem Set 2

All parts are due on February 21, 2020 at 6PM. 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. Solutions should be submitted on the course website, and any code should be submitted for automated checking on `alg.mit.edu`.

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] $T(n) = T(n-2) + \Theta(n)$

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**.

(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.

(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.

Problem 2-3. [10 points] **Friend Finder**

Jean-Locutus Πcard is searching for his incapacitated friend, Datum, on Gravity Island. The island is a narrow strip running north–south for n kilometers, and Πcard needs to pinpoint Datum's location to the nearest integer kilometer so that he is within visual range. Fortunately, Πcard 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 Πcard 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 Πcard 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 $n = \|\mathtt{V}\|$ 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 $O(\mathtt{k})$ 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.

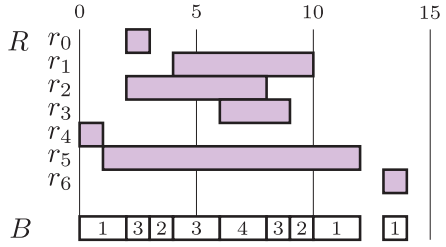
[1] As mentioned in lecture, unless we parameterize the size of numbers in our input, you should assume that input integers each fit within a machine word, so pairs of them may be compared in constant time.

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:



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

(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.

(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. Submit your code online at `alg.mit.edu`.

```
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)
```