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

February 15, 2020
Problem Set 2

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`.

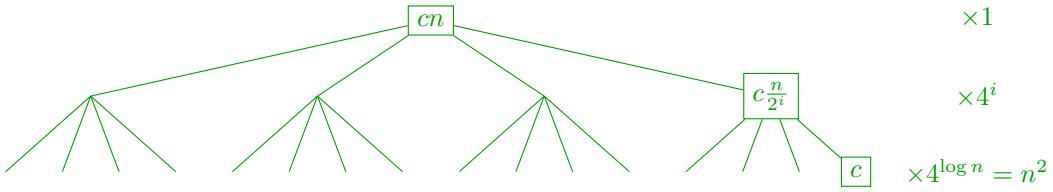
Problem 2-1. [15 points] **Solving recurrences**

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

Solution: $T(n) = \Theta(n^2)$ by case 1 of the Master Theorem, since $\log_b a = 2$ and $f(n) = O(n^{2-\varepsilon})$ for any positive $\varepsilon \leq 1$. Note that this is true no matter the choice of $f(n) \in O(n)$.



Drawing a tree, there are 4^i vertices at depth i each doing at most $c\frac{n}{2^i}$ work, so the total work at depth i is at most $4^i c\frac{n}{2^i} = 2^i cn$. Summing over the entire tree, the total work is at most

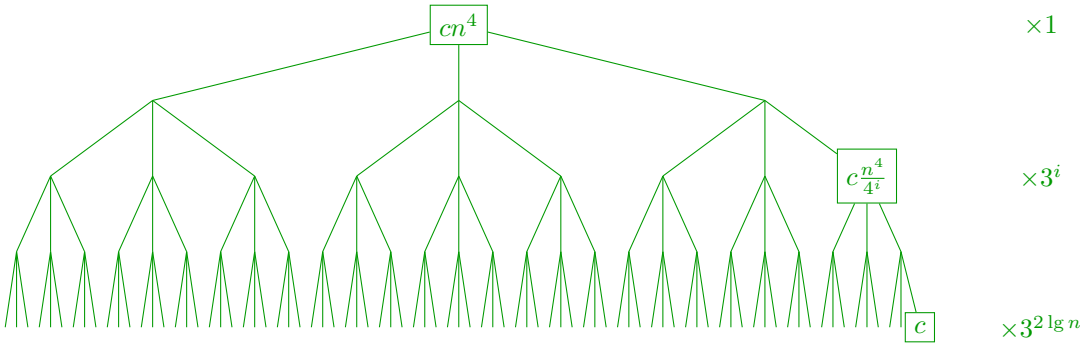
$$\sum_{i=0}^{\log n} 2^i cn = cn \sum_{i=0}^{\log n} 2^i = cn(2^{\log n+1} - 1) < 2cn^2 = O(n^2)$$

Since $\Theta(1)$ work is done at each leaf, and there are n^2 leaves, the total work is at least $\Omega(n^2)$ leading to $\Theta(n^2)$ running time.

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

Solution: We can upper bound $T(n)$ by choosing $f(n) = \Theta(n^4)$. Then $T(n) = O(n^4)$ by case 3 of the Master Theorem, since $\log_b a = \log_{\sqrt{2}} 3 = 2\lg 3$ and $\Theta(n^4) \subseteq \Omega(n^{2\lg 3+\varepsilon})$ for any positive $\varepsilon \leq (4 - 2\lg 3)$, **and** $3(\frac{n}{\sqrt{2}})^4 = \frac{3}{4}n^4 < cn^4$ for any $\frac{3}{4} < c < 1$.

Alternatively, we can lower bound $T(n)$ by choosing $f(n) = 0$. Then $T(n) = \Omega(n^{2\lg 3})$ by case 1 of the Master Theorem, since $\log_b a = \log_{\sqrt{2}} 3 = 2\lg 3$ and $0 \in O(n^{2\lg 3-\varepsilon})$ for any positive $\varepsilon \leq 2\lg 3$.



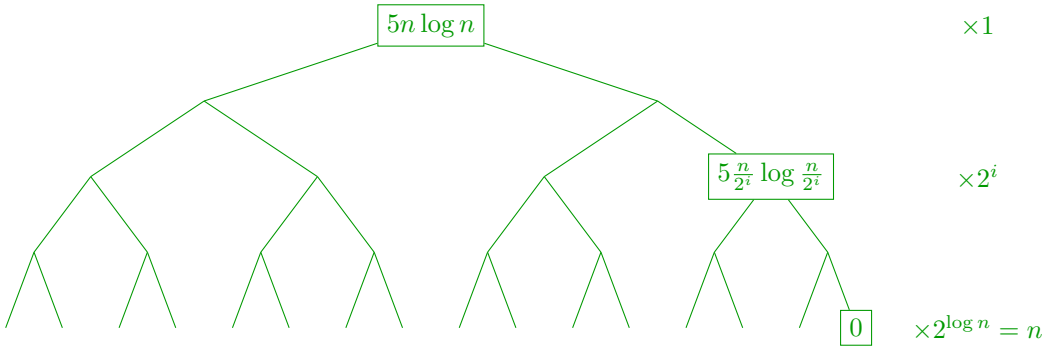
Drawing a tree, there are 3^i vertices at depth i, each doing at most $c\left(\frac{n}{\sqrt{2^i}}\right)^4 = c\frac{n^4}{4^i}$ work, so the total work at depth i is at most $c\frac{3^i}{4^i}n^4$. Summing over the entire tree, the total work is at most

$$\sum_{i=0}^{2\log n} c\frac{3^i}{4^i}n^4 < \sum_{i=0}^{\infty}\left(\frac{3}{4}\right)^i cn^4 = 4cn^4,$$

so $T(n) = O(n^4)$. Alternatively, there are $3^{2\lg n} = n^{2\lg 3}$ leaves in the tree, each doing $\Theta(1)$ work, so $T(n)$ is at least $\Omega(n^{2\lg 3})$.

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

Solution: $T(n) = \Theta(n\log^2 n)$ by case 2 of the Master Theorem, since $\log_b a = 1$ and $f(n) = 5n\log n = \Theta(n^1\log^1 n)$.



Drawing a tree, there are 2^i vertices at depth i each doing $5\frac{n}{2^i}\log\frac{n}{2^i}$ work, so the total work at depth i is $5n\log\frac{n}{2^i} = 5n(\log n - i)$. In other words, the total work on j^{th} level from the bottom is $5nj$. Summing over the entire tree, the total work is

$$\sum_{i=0}^{\log n} 5n(\log n - i) = 5n\sum_{j=0}^{\log n} j = 5n\frac{\log n(\log n - 1)}{2} = \Theta(n\log^2 n).$$

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

Solution: Guess $T(n) = cn^2$

$$\begin{aligned} cn^2 &\stackrel{?}{=} c(n-2)^2 + \Theta(n) \\ cn^2 &\stackrel{?}{=} cn^2 - 4cn + 4c + \Theta(n) \\ 4cn - 4c &= \Theta(n) \end{aligned}$$

So $T(n) = \Theta(n^2)$. □

Rubric: For parts (a)-(c)

- Parts (a)-(c)
  - 1 points for drawing of tree
  - 1 points for analysis based on tree
  - 2 points for analysis via Master Theorem
  - Partial credit may be awarded
- Part (d)
  - 3 points for correct analysis based on substitution
  - Partial credit may be awarded

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**.

Solution: This part requires an in-place sorting algorithm, so we cannot choose merge sort, as merge sort is not in-place. Insertion sort performs $O(n^2)$ `get_at` and $O(n^2)$ `set_at` operations, so would take $O(n^3 \log n)$ time with this data structure. Alternatively, selection sort performs $O(n^2)$ `get_at` operations but only $O(n)$ set_at operations, so would take at most $O(n^2 \log n)$ time with this data structure, so we choose **selection sort**.

(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.

Solution: For this problem, reads and writes take constant time, but comparisons are expensive, $O(\log n)$. Selection and insertion sorts both perform $O(n^2)$ comparisons in the worst case, while merge sort only performs $O(n \log n)$ comparisons, so we choose **merge sort**.

(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.

Solution: The performance of selection sort and merge sort do not depend on the input; they will run in $\Theta(n^2)$ and $\Theta(n \log n)$ time, regardless of the input. Insertion sort, on the other hand, can break early on the inner loop, so can run in $O(n)$ time on some inputs. To prove that insertion sort runs in $O(n)$ time for the provided inputs, observe that performing a single swap between adjacent items can change the number of inversions[1] in the array by at most one. Alternatively, every time insertion sort swaps two items in the inner loop, it fixes an inversion. Thus, if an array is k adjacent swaps from sorted, insertion sort will run in $O(n + k)$ time. For this problem, since $k = \log \log n = O(n)$, insertion sort runs in $O(n)$ time, so we choose **insertion sort**.

Rubric:

- 2 points for choice of sorting algorithm
- 3 points for justification
- Partial credit may be awarded

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.

Solution: We can generalize the idea of binary search to search quickly from both ends. The idea will be to alternately search inward from either end of the island exponentially until Πcard just

[1] Two comparable items in an array are inverted if they appear in the wrong order. The number of inversions in an array are the number of pairs of items that are inverted.

passes Datum, and then use normal binary search to pinpoint Datum's precise location. Specifically, tell Πcard to alternately teleport to 2^i and $n - 2^i$ for increasing i starting at 0 until either Datum is found, or until Datum has been passed, i.e., Datum is observed north of 2^{j-1} but south of 2^j for some j, or south of $n - 2^{j-1}$ but north of $n - 2^j$ for some j. Reaching this state will take $O(j)$ time, since at most $2j$ locations will be visited, and then binary searching within the remaining 2^{j-1} kilometer stretch of the island will also take at most $O(j)$ time. But since either $2^{j-1} < k < 2^j$ or $n - 2^j < n - k < n - 2^{j-1}$, then $j - 1 < \lg k < j$ and $j = O(\log k)$, as desired. This algorithm is correct because it identifies a bounded range where Datum is known to exists, and reduces to binary search which will correctly return Datum's location.

Rubric:

- 6 points for description of a correct algorithm
- 2 points for a correct argument of correctness
- 2 points for a correct analysis of running time
- Partial credit may be awarded

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[2]. 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.

Solution: We will implement the database by maintaining two data structures:

- a doubly-linked list L containing the sequence of all undeleted messages in the chat in chronological order, and
- a sorted array S of pairs (v, p_v) keyed on v, where v is a viewer ID and p_v is a pointer to a viewer-specific singly-linked list L_v storing pointers to all the nodes in L containing a

[2]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.

message posted by viewer v. We will use p_v pointing to None to signify that viewer v has been banned.

To support `build(V)`, initialize L to be an empty linked list in $O(1)$ time, initialize S of size $n = |\mathtt{V}|$ containing (v, p_v) for each viewer $v \in \mathtt{V}$ in $O(n)$ time, initialize the empty linked list L_v for each $v \in \mathtt{V}$ in $O(1)$ time each, and then sort S in $O(n \log n)$ time, e.g., using merge sort. This operation takes worst-case $O(n \log n)$ time in total, and maintains the invariants of the database.

To support `send(v, m)`, find L_v by searching for v in S in worst-case $O(\log n)$ time, and then, if L_v is not None (i.e., the user is not banned), insert m into a node x at the front of L in worst-case constant time and insert a pointer to x into L_v in worst-case constant time. This operation takes worst-case $O(\log n)$ time in total, and maintains the invariants of the database.

To support `recent(k)`, simply traverse the first k nodes of L and return their messages. As long as the invariants on the database are correct, this operation directly returns the requested messages in worst-case $O(k)$ time.

To support `ban(v)`, find L_v by searching for v in S in worst-case $O(\log n)$ time. Then for each of the n_v pointers in L_v pointing to a node x in L, remove x from L by relinking pointers in L in worst-case constant time. Lastly, set p_v to point to None. This operation is correct because it maintains the invariants of the database, and runs in worst-case $O(n_v + \log n)$ time.

Rubric:

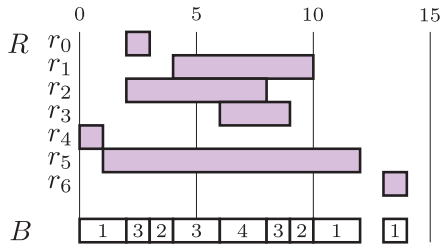
- 3 points for general description of a correct database
- 2 point for description of a correct `build(V)`
- 2 points for description of a correct `send(v, m)`
- 2 points for description of a correct `recent(k`
- 2 points for description of a correct `ban(v)`
- 1 point for analysis for each operation (4 points total)
- Partial credit may be awarded

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 $R = R_1 \cup R_2$.

Solution: To merge two booking schedules, our approach will be to maintain a time x (initially 0) and an (initially empty) booking schedule B, so that B will be the satisfying booking for $R_1 \cup R_2$ up to time x. This invariant is trivially true at initialization since all times are positive, and so long as we maintain this invariant while increasing x, B will be a covering booking schedule for all of $R_1 \cup R_2$ once x increases past the last ending time of any request (will be satisfying except for possible adjacent bookings with the same number of rooms reserved). During this process, we also maintain indices i_1 and i_2 (initially zero) into booking schedules B_1 and B_2 respectively, which will each correspond to the index of the earliest room booking from its respective schedule whose ending time is strictly after x.

Now we perform a loop to repeatedly increase x and append a new booking request to B so as to satisfy the invariant, until all requests from B_1 and B_2 have been processed, i.e., $i_1 = |B_1|$ and $i_2 = |B_2|$. There are five cases, either:

- one schedule has been depleted (either $i_1 = |B_1|$ or $i_2 = |B_2|$), so take the next booking request (k,s,t) from the non-depleted schedule, and append to B $(k, \max(s,x), t)$, increase x to t, and increase the relevent schedule index; or
- neither schedule has been depleted, so either:
  - neither next booking from either schedule overlaps x, so increase x to be the minimum start time in either booking; or
  - the next booking (k,s,t) from either schedule does not overlap a booking in the other schedule after time x, so append (k,x,t) to B, increase x to t, and increase the relevant schedule index; or
  - the next booking (k,s,t) from either schedule overlaps a booking (k',s',t') in the other schedule at a time $s' > x$, so append (k,x,s') to B and increase x to s'; or
  - bookings (k,s,t) and (k',s',t') from both schedules overlap after and at time x until $t^* = \min(t,t')$, so append $(k+k',x,t^*)$ to B, increase x to t^*, and increase whichever schedule indices correspond to reservations that end at time t^*.

This procedure maintains the invariants asserted above, so upon termination, B is a covering booking request for R. Constant work is done in each execution of the above loop, and since x increases in every loop to a strictly larger time from the set of $O(n)$ times found in B_1 or B_2, this procedure takes $O(n)$ time.

Lastly, to make the booking satisfying, we loop through the bookings and combine any adjacent bookings that have the same number of rooms in $O(n)$ time.

Rubric:

- 9 points for description of a correct algorithm
- 3 points for a correct argument of correctness
- 3 points for a correct analysis of running time
- Partial credit may be awarded

(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.

Solution: Evenly divide R into two $\Theta(|R|)$ sized sets R_1 and R_2 and recursively compute booking schedules B_1 and B_2 that satisfy R_1 and R_2 respectively. Then compute a satisfying booking B for R using part (a) in $O(n)$ time. As a base case, when $|R| = 1$ and $R = \{(s, t)\}$, the satisfying booking is $((1, s, t),)$, so return it in $Theta(1)$ time. This algorithm follows the recurrence $T(n) = 2T(n/2) + O(n)$, so by Master Theorem case 2, this algorithm runs in $O(n \log n)$ time.

Rubric:

- 3 points for description of a correct algorithm
- 1 point for a correct argument of correctness
- 1 point for a correct analysis of running time
- Partial credit may be awarded

(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`.

Solution:

```
def merge_bookings(B1, B2):
    n1, n2, i1, i2 = len(B1), len(B2), 0, 0
    x = 0       # invariant: t < min(t1, t2)
    B = []      # invariant: B is satisfying booking up to time x
    while i1 + i2 < n1 + n2:
        if i1 < n1: k1, s1, t1 = B1[i1]
        if i2 < n2: k2, s2, t2 = B2[i2]
        if i2 == n2:                # only bookings in B1 remain
            k, s, x = k1, max(x, s1), t1
            i1 += 1
        elif i1 == n1:              # only bookings in B2 remain
            k, s, x = k2, max(x, s2), t2
            i2 += 1
        else:                       # bookings remain in B1 and B2
            if x < min(s1, s2):     # shift x to start of first booking
                x = min(s1, s2)
            if t1 <= s2:            # overlaps only B1 up to t1
                k, s, x = k1, x, t1
                i1 += 1
            elif t2 <= s1:          # overlaps only B2 up to t2
                k, s, x = k2, x, t2
                i2 += 1
            elif x < s2:            # overlaps only B1 up to s2
                k, s, x = k1, x, s2
            elif x < s1:            # overlaps only B2 up to s1
                k, s, x = k2, x, s1
            else:                   # overlaps B1 and B2 up to t1 or t2
                k, s, x = k1 + k2, x, min(t1, t2)
                if t1 == x: i1 += 1
                if t2 == x: i2 += 1
        B.append((k, s, x))
    B_ = [B[0]]                     # remove adjacent with same rooms
    for k, s, t in B[1:]:
        k_, s_, t_ = B_[-1]
        if (k == k_) and (t_ == s):
            B_.pop()
            s = s_
        B_.append((k, s, t))
    return B_

def satisfying_booking(R):
    if len(R) == 1:                 # base case
        s, t = R[0]
        return ((1, s, t),)
    m = len(R) // 2
    R1, R2 = R[:m], R[m:]           # divide
    B1 = satisfying_booking(R1)     # conquer
    B2 = satisfying_booking(R2)     # conquer
    B = merge_bookings(B1, B2)      # combine
    return tuple(B)
```