Problem Session 2

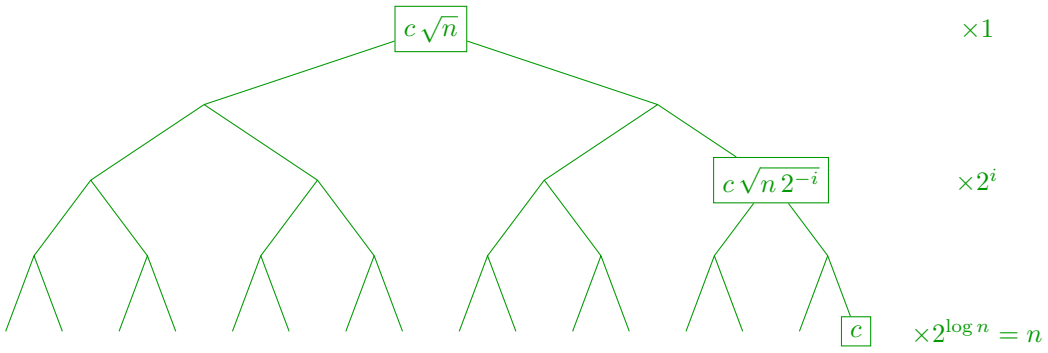
Problem 1-1. Solving recurrences

Derive solutions to the following recurrences in two ways: via a recursion tree **and** via Master Theorem. A solution should include the tightest upper and lower bounds that the recurrence will allow. Assume $T(1) \in \Theta(1)$.

(a) $T(n) = 2\,T(\frac{n}{2}) + O(\sqrt{n})$

Solution: $T(n) \in \Theta(n)$ by case 1 of the Master Theorem, since:

$$O(\sqrt{n}) \subseteq O(n) = O(n^{\log_2 2}).$$

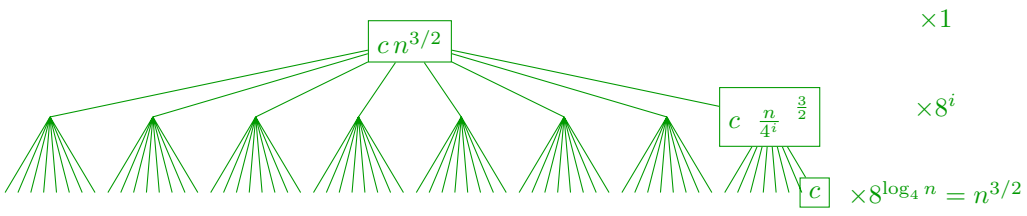


Drawing a tree, there are 2^i vertices at depth i each doing at most $c\sqrt{n\,2^{-i}}$ work, so the total work at depth i is at most $c\,2^{i/2}\sqrt{n}$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\log n} c\,2^{i/2}\sqrt{n} = c\sqrt{n}\sum_{i=0}^{\log n} 2^{i/2} = cn\,\frac{\sqrt{2}}{\sqrt{2}-1} \in O(n)$. Since $\Theta(1)$ work is done at each leaf, and there are n leaves, the total work is also $\Omega(n)$ leading to $\Theta(n)$ running time.

(b) $T(n) = 8\,T(\frac{n}{4}) + O(n\sqrt{n})$

Solution: $T(n) \in O(n^{3/2}\log n)$ by case 2 of the Master Theorem, since:

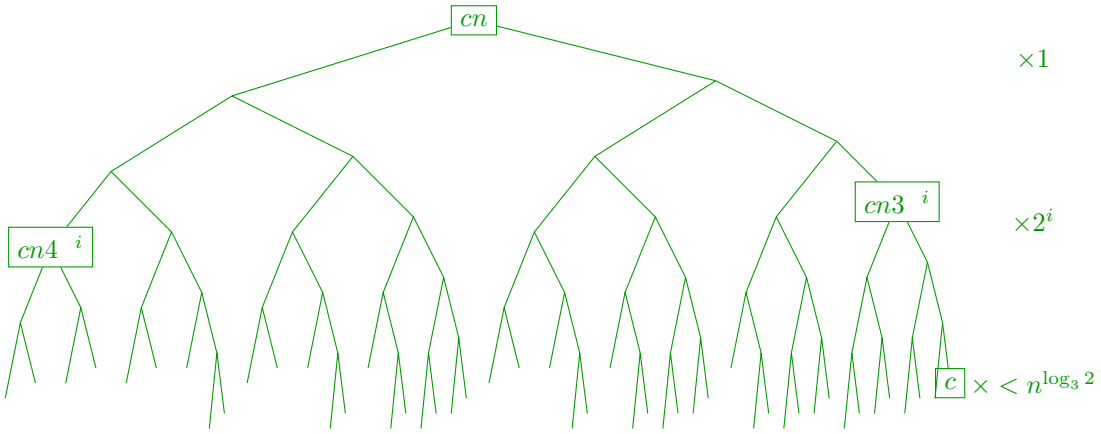
$$O(n\sqrt{n}) = O(n^{3/2}) = O(n^{\log_4 8}).$$



Drawing a tree, there are 8^i vertices at depth i each doing at most $c\,(n\,4^{\,i})^{3/2} = c\,n^{3/2}\,8^{\,i}$ work, so the total work at depth i is at most $c\,n^{3/2}$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\log_4 n} c\,n^{3/2} = \frac{c}{2}\,n^{3/2}\log n \in O(n^{3/2}\log n)$.

(c) $T(n) = T(\frac{n}{3}) + T(\frac{n}{4}) + \Theta(n)$ assuming $T(a) < T(b)$ for all $a < b$

Solution: By monotonicity, $T(n) \le 2\,T(\frac{n}{3}) + \Theta(n)$. Then $T(n) \in O(n)$ by case 3 of the Master Theorem, since $\Theta(n) \subset \Omega(n^{\log_3 2+\epsilon})$ for some $\epsilon > 0$ (e.g. $\epsilon = 1/3$). On the other hand, if we ignore the recursive component, $T(n) \in \Omega(n)$. Combining the two gives $T(n) \in \Theta(n)$.



Drawing a tree, there are 2^i vertices at depth i, $\binom{i}{j}$ of which do $cn3^{\,j}4^{j\ i}$ work (as long as $3^j4^{j\ i} \le n$). As a crude upper bound, this is at most $cn3^{\ i}$, so the total work at depth i is at most $cn\left(\frac{2}{3}\right)^i$. Summing over the entire tree, the total work is at most $\sum_{i=0}^{\infty} cn\left(\frac{2}{3}\right)^i = 3cn \in O(n)$. We have cn work at the root, giving a lower bound of $\Omega(n)$. Combining the two gives a total of $\Theta(n)$.

Problem 1-2. Stone Searching

Sanos is a supervillain on an intergalactic quest in search of an ancient and powerful artifact called the Thoul Stone. Unfortunately she has no idea what planet the stone is on. The universe is composed of an infinite number of planets, each identified by a unique positive integer. On each planet is an oracle who, after some persuasion, will tell Sanos whether or not the Thoul Stone is on a planet having a strictly higher planet identifier than their own. Interviewing every oracle in the universe would take forever, and Sanos wants to find the Thoul Stone quickly. Supposing the Thoul Stone resides on planet k, describe an algorithm to help Sanos find the Thoul Stone by interviewing at most $O(\log k)$ oracles.

Solution: First observe that if we could find a planet with identifier $x > k$ that is not too much larger than k (specifically, $x = \Theta(k)$), then we would be done, as binary searching the planets from 1 to $x \quad 1$ would find the value of k by visiting at most $O(\log x) = O(\log k)$ oracles. It remains to find such a planent x.

To find x, instruct Sanos to visit planets 2^i starting at $i = 0$ until an orcale on planet $x = 2^{i^*}$ first tells Sanos that $x > k$. Since x is the first planet for which $2^i > k$, then $x/2 < k$ and $x < 2k = \Theta(k)$ as desired. To reach planet $x = 2^{i^*}$, Sanos interviews $i^* = \lceil \log_2 k \rceil = \Theta(\log k)$ oracles, so to find k, this algorithm interviews at most $O(\log k)$ oracles as desired.

Problem 1-3. Collage Collating

Fodoby is a company that makes customized software tools for creative people. Their newest software, Ottoshop, helps users make collages by allowing them to overlay images on top of each other in a single document. Describe a database to keep track of the images in a given document which supports the following operations:

1. `make_document()`: construct an empty document containing no images
2. `import_image(x)`: add an image with unique integer ID `x` to the top of the document
3. `display()`: return an array of the document's image IDs in order from bottom to top
4. `move_below(x, y)`: move the image with ID `x` directly below the image with ID `y`

Operation (1) should run in worst-case $O(1)$ time, operations (2) and (3) should each run in worst-case $O(n)$ time, while operation (4) should run in worst-case $O(\log n)$ time, where n is the number of images contained in a document at the time of the operation.

Solution: This database requires us to maintain a sequence of images ordered extrinsically, but also support searching intrinsically for images based on their ID. So, we will implement the database with a combination of both a sequence data structure, specifically a **doubly** linked list (as implemented in PS1-4) storing image IDs, and a set data structure, specifically a sorted array storing pairs (x, v_x) sorted by x values, where x is the ID of an image, and v_x is a pointer to the linked list node containing x.

To implement `make_document()`, simply initialize an empty linked list L and an empty sorted array S, each in $O(1)$ time. There is no output to this operation, so it is trivially correct.

To implement `import_image(x)`, add x to the front of L in node v_x in $O(1)$ time and add (x, v_x) to S in $O(n)$ time. Delegating to these data structures ensures that x is added to front of the sequence stored in L, and that S now contains (x, v_x) and remains sorted after insertion.

To implement `display()`, construct and return an array by iterating the items of L in sequence order which can be done in $O(n)$.

To implement `move_below(x, y)`, use binary search to find pairs (x, v_x) and (y, v_y) in S, each in $O(\log n)$ time. Then we can remove node v_x from L in $O(1)$ time and insert it after node v_y, also in $O(1)$ time, by relinking pointers. For completeness, here is one way to relink the pointers in a doubly linked list:

```
def relink(S, vx, vy):
    if vx.prev: vx.prev.next = vx.next
    else:       S.head = vx.next
    if vx.next: vx.next.prev = vx.prev
    else:       S.tail = vx.prev
    vx.prev = vy
    vx.next = vy.next
    if vy.next: vy.next.prev = vx
    else:       S.tail = vx
    vy.next = vx
```

Problem 1-4. Brick Blowing

Porkland is a community of pigs who live in n houses lined up along one side of a long, straight street running east to west. Every house in Porkland was built from straw and bricks, but some houses were built with more bricks than others. One day, a wolf arrives in Porkland and all the pigs run inside their homes to hide. Unfortunately for the pigs, this wolf is extremely skilled at blowing down pig houses, aided by a strong wind already blowing from west to east. If the wolf blows in an easterly direction on a house containing b bricks, that house will fall down, along with every house east of it containing strictly fewer than b bricks. For every house in Porkland, the wolf wants to know its **damage**, i.e., the number of houses that would fall were he to blow on it in an easterly direction.

(a) Suppose $n = 10$ and the number of bricks in each house in Porkland from west to east is `[34, 57, 70, 19, 48, 2, 94, 7, 63, 75]`. Compute for this instance the damage for every house in Porkland.

Solution: `[4, 5, 6, 3, 3, 1, 4, 1, 1, 1]`

(b) A house in Porkland is **special** if it either (1) has no easterly neighbor or (2) its adjacent neighbor to the east contains at least as many bricks as it does. Given an array containing the number of bricks in each house of Porkland, describe an $O(n)$-time algorithm to return the damage for every house in Porkland **when all but one house** in Porkland is special.

Solution: Maintain an array D of the same size as the input array H to store updated damages, where the i^{th} item of D is an integer representing the number of damages counted so far. To add damage to the i^{th} house, add to the value at $D[i]$ in $O(1)$ time. As each house will itself fall down when blown on, initialize every element of D to 1 in $O(n)$ time, and count other damages using the following algorithm.

If exactly one house (say the h^{th} house) in Porkland is not special, that means the subarray A from the east-most house to h non-strictly monotonically increases, as does the subarray B from the $(h+1)^{\text{th}}$ house to the west-most house. We can find h in $O(n)$ time via a linear scan. The damage for any house in subarray B is 1, as no house to the west contains strictly fewer bricks, so these values are set correctly at initialization.

It remains to compute the damage for the houses in A. Use a two-finger algorithm starting with one index i at the beginning of A ($i = 0$) and another index j at the beginning of B ($j = 0$). Then repeat the following process until $i = |A|$: if $j < |B|$ and house $A[i]$ has strictly more bricks than $B[j]$, then increase j by 1; otherwise, add j to the damage at $D[i]$ and increase i by 1. This loop halts when $i+j = |A|+|B| = n$, and $i + j$ increases by one in each iteration. Since the work done in each iteration is $O(1)$, this algorithm runs in $O(n)$ time.

To prove that this algorithm correctly computes the damage for each house in A we first prove that the loop above maintains the invariant that at the start of each iteration, $A[i] > B[k]$ for all $k \in \{0, \ldots, j \quad 1\}$. This property implies the algorithm computes damage correctly for each house in A: the algorithm updates the damage for $A[i]$ when $A[i] \leq B[j]$, so the claim implies the houses west of $A[i]$ with strictly fewer bricks are exactly the houses $H = \{B[k] \mid k \in \{0, \ldots, j \quad 1\}\}$ where $|H| = j$; so the damage blowing on house $A[i]$ is $D[i] = j + 1$ as recorded.

To prove the claim, we induct on $i + j$. When $i + j = 0$, the claim is vacuously true as the set of possible k is empty. Now assume for induction that the claim holds for some $i + j$. If $A[i] > B[j]$, then increasing j by one directly maintains the invariant since $A[i] > B[k]$ for $k \in \{0, \ldots, j \quad 1\}$ by induction. Alternatively, if $A[i] \leq B[j]$, then increasing i by 1 also maintains the induction hypothesis since $A[i + 1] > A[i]$, proving the claim.

(c) Given an array containing the number of bricks in each house of Porkland, describe an $O(n \log n)$-time algorithm to return the damage for every house in Porkland.

Solution: We modify merge sort to record all damages that occur between houses within each subarray before every merge. Since we will be moving brick values in H from their original locations, we will replace each brick value $b_i = H[i]$ with tuples $H[i] = (b_i, i)$, to keep track which house is associated with b_i.

As in (b), initialize a damages array D to 1s. Then, recursively sort and record damages that would occur between houses in the first half of H, and then do the same for the second half. Next use the $O(n)$-time algorithm from part (b) to count the damages that would occur between one house in the first half and one house in the second half, and then use the merge step of merge sort to combine the two sorted halves into one sorted array in $O(n)$ time. Since both (b) and merge take $O(n)$ time, the recurrence for this algorithm is the same as merge sort, yielding an $O(n \log n)$ running time.

Now we prove this algorithm correctly records the damage of every house in a given subarray with any other house in the subarray (in addition to sorting the subarray), by inducting on the size of the subarray. When the subarray has size 1, there is exactly one damage between houses within that subarray, and the initialization step records it. Alternatively, assume for induction that the claim is true for all $k < n$. By induction, the algorithm correctly records all damages between houses within the first half of the subarray, and also all damages between houses within the second half. It remains to record damages between houses in the left half with houses on the right half.

Fortunately, since the first and last halves of the subarray are sorted, the algorithm in (b) counts exactly those damages. Then sorting using the merge step of merge sort maintains the invariant as desired.

Note that the merge step and the algorithm in part (b) are both two finger algorithms that traverse from the starts of the same two subarrays. Our implementation for (d) utilizes this observation to record the damages with the merge step of merge sort, rather than separately.

(d) Write a Python function `get_damages` that implements your algorithm.

Solution:

```
def get_damages(H):
    D = [1 for _ in H]
    H2 = [(H[i], i) for i in range(len(H))]
    def merge_sort(A, a = 0, b = None):
        if b is None:   b = len(A)
        if 1 < b - a:
            c = (a + b + 1) // 2
            merge_sort(A, a, c)
            merge_sort(A, c, b)
            i, j, L, R = 0, 0, A[a:c], A[c:b]
            while a < b:
                if (j >= len(R)) or (i < len(L) and L[i][0] <= R[j][0]):
                    D[L[i][1]] += j
                    A[a] = L[i]
                    i += 1
                else:
                    A[a] = R[j]
                    j += 1
                a += 1
    merge_sort(H2)
    return D
```

6.006 Introduction to Algorithms

Spring 2020