

Big-O. Suppose $T(m) = 67$ for all $m \leq 381$. Give the closed form expression for $T(n)$ when $n \geq 382$:

1. $T(n) = 3T(n/3) + \log n$
2. $T(n) = T(\sqrt{n}) + 1$
3. $T(n) = 2^n T(n/2) + 2$
4. $T(n) = 32T(n/4) + n$
5. $T(n) = T(n/100) + T(n/40) + T(n/10) + 5T(n/1000) + 5^n$

Solution.

1. $\Theta(n)$
2. $\Theta(\log \log n)$. **Intuition:** the recurrence is counting the number of times I can root n until it becomes a constant. If you think about the bit representation of n , which has $m = \log n$ bits, a square root is basically performing a left shift $\approx m/2$ times. I can only do this $\log m = \log \log n$ times.

More formally, I do a variable replacement to get the recurrence into a form where I can apply master's theorem or tree method, e.g., getting fractions into my recurrence. Let $n = 2^m$. Then, $T(n) = T(2^m) = T(2^{m/2}) + 1$. I do another variable replacement, this time setting $S(m) = T(2^m)$, which gets me $S(m) = S(m/2) + 1$, which I know to be $O(\log m) = O(\log \log n)$. (Take a moment to see how the intuition mirrors the formal proof—wow!)

3. $\Theta(4^n)$
4. $\Theta(n^{5/2})$
5. $\Theta(5^n)$

Summations. Give the tightest asymptotic upperbound for the following summations:

1. $\sum_{i=1}^n \log i$
2. $\sum_{i=1}^n (1.01)^i$
3. $\sum_{i=1}^{n^2} (n^3 + 10/i)$
4. $\sum_{i=0}^n \binom{n}{i}$

Solution.

1. $\Theta(n \log n)$
2. $\Theta((1.01)^n)$
3. $\Theta(n^5 + \log n) = \Theta(n^5)$
4. $\Theta(2^n)$ (hint: binomial theorem or, my personal preference, a counting argument on the number of subsets of $\{1, \dots, n\}$)

I Just Like Driving. You just made a climate-conscious decision to buy an electric car from a San Jose, California car dealership. They delivered it via drone to your place in West Lafayette, Indiana, but they forgot to give you the car deed! Now you have to drive to the car dealership to pick it up. You don't mind the drive but you have to stop to charge up your car after driving at most 450 miles. You're already miffed about this whole thing, so you want to minimize the amount of time you spend charging your car, while making sure you don't run out of electricity between stations.

Let n be the total number of charging stations and suppose they're ordered $1, \dots, n$ along the route by distance from your starting location. You are given two arrays, $D[1 \dots n]$ and $T[1 \dots n]$, and an additional value $Z > 0$. Each $D[i] \in \mathbb{R}_{\geq 0}$ is a non-negative real number representing the distance along the route at which the gas station appears. Each $T[i] \in \mathbb{R}_{\geq 0}$ is the time it takes to charge your car at charging station i . Design and analyze an algorithm the minimum amount of time one has to spend at charging stations in going from location 0 to location Z , ensuring you never drive more than 450 miles without visiting a gas station (output ∞ if it is impossible to do so).

Solution. Let $F[i]$ be the minimum total charging time to arrive to destination i and pay its cost. Then,

$$F[i] = \begin{cases} 0 & i = 0 \\ c_i + \min_{\substack{0 \leq j < i \\ d_i - d_j \leq 450}} F[j] & \text{otherwise} \end{cases}$$

Output $F[m+1]$. The dp takes $O(n^2)$ time naively. (Challenge: there is a $O(n)$ solution using the so-called 'monotonic-dequeue dp')

I Guess We Doing Lowerbounds Now.

1. Take 2 minutes to write down what you remember about the sorting lowerbound.
2. Let $A[1, \dots, n]$ be an array of comparable elements. Consider the problem of *searching* in array A . Show that:
 - (a) searching takes at least $\Omega(n)$ comparisons on arbitrary arrays A .
 - (b) searching takes at least $\Omega(\log n)$ comparisons on *sorted* arrays A .

Solution.

1. zzz
2. Roughly, you want to argue that there exists an array which you would give an incorrect answer if you used less than n comparisons (say $n - 1$ comparisons). If I had such a comparison-based searching algorithm that I assumed to be correct, say it doesn't compare at the k th index for array A . Then, changing $A[k]$, I can create an array which both contains/doesn't contain the element we are searching for.
3. Its a similar argument to sorting, except instead of $n!$ possible permutations, there are n possible elements we can find. Then, if x is the search lower bound, then $n \leq 2^x \implies x \geq \log n$.

Dejavu. ... Suppose I have K sorted arrays each with N/K elements in it. Give an efficient algorithm which outputs the sorted array of all N elements in $O(N \log K)$ time.

Solution. Hint: use a heap which only stores at most K elements at a time

Justin's Rejected Problem.¹ You are a newly minted Meta software engineering intern after passing 67 interview rounds. You match with a team in Instagram Reels, and your intern project is to manually find the majority cat-based trend (e.g., drooling cat, sus cat, kidnapping cat, etc.).

Formally, let $A[1, \dots, n]$ be an array of Instagram Reel screenshots of cats. You can tell when two screenshots are not the same (e.g., drooling cat is not equal to kidnapping cat) but you can't order them (e.g., there is no notion that drooling cat $<$ kidnapping cat or vice versa). Your objective is to find the majority element in this array, an element that appears more than $n/2$ times, via a divide and conquer algorithm. **You are only allowed to use (non-recursive) constant space.**

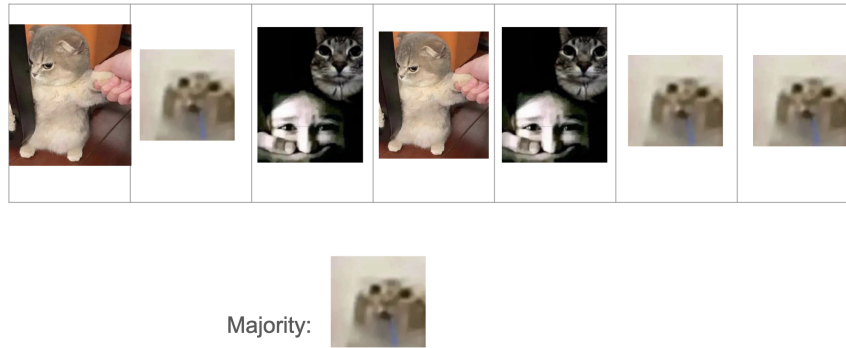


Figure 1: **This picture is wrong!! Pretend there are two more drooling cats at the end, for a total of 9 cats.**

Solution.

Maj(l,r)

//Outputs the majority element of $A[l, r]$.

1. If $l = r$: output $A[l]$.
2. Let $m \leftarrow \lfloor (l + r)/2 \rfloor$.
3. Let $L = \text{Maj}(l, m)$ and $R = \text{Maj}(m + 1, r)$.
4. Within $A[l, r]$, output the majority element between L and R . That is, output L or R if they appear more than $(r - l + 1)/2$ times. Otherwise, output $\perp$. (takes $O(n)$ time).

This takes overall $O(n \log n)$ time.

One can achieve a $O(n)$ algorithm by computing an element and its cancellation count, i.e., for a majority element e , we output $(e, \text{freq}(e) - \text{freq}(\text{everything other than } e)) = (e, 2\text{freq}(e) - |A|)$. If e is indeed the maximum element, then the cancellation count should be greater than 0.

f(l,r)

1. if $r < l$: output $(\perp, 0)$.
2. if $l = r$: output $(A[l], 1)$.
3. Let $m = \lfloor (l + r)/2 \rfloor$.
4. Let $(L, c_L) = f(l, m)$ and $(R, c_R) = f(m + 1, r)$.

¹Lord I am One, Lord I am Two, Lord I am the Majority Element - 500 Meows by Peter, Paw and Meowy

5. If $L == R$: output $(L, c_L + c_R)$
6. If $c_L = 0$: output (R, c_R)
7. If $c_R = 0$: output (L, c_L)
8. If $c_L < c_R$: output $(R, c_R - c_L)$
9. If $c_R < c_L$: output $(L, c_L - c_R)$
10. Otherwise, output $(\perp, 0)$.

Then, run $f(1, n)$ to get (e, c_e) and output e .

Lowkenuinely Non-Linear 2. Given n distinct integers $z_1, \dots, z_n$, we want to find an integer x that minimizes the following quantity: $\max_{z_i} |x - z_i|$. Give an $O(n)$ time algorithm for finding such a x and prove correctness of your algorithm.

Solution. $x = ((\min z_i) + (\max z_i))/2$, whichever way we round is optimal.

Overstimulation as an Algorithm Problem. A sequence of numbers $x_1, \dots, x_k$ is strictly increasing and even if $x_i < x_{i+1}$ for all $i = 1, \dots, (k-1)$ and each x_i is even.

Let $A[1 \dots n]$ be an array of n integers. Compute the maximum sum of any strictly increasing and even subsequence of A and analyze the runtime.

Solution. Let $S[i]$ be the largest sum of a valid subsequence ending at index i (not including odd indices). Then,

$$S[i] = A[i] + \max_{j < i, A[j] \text{ even}} S[j]$$

which takes $O(n^2)$ time naively. Challenge: One can also do this in $O(n \log n)$ time with a segment tree.

Envelope Problem Iteratively. Consider the Envelope problem in PSO 5. Solve it iteratively instead of recursively.

Statistically, 50% of you will Join a Run Club within 3 Years. ² You are a part of Indiana's most prominent run club. In particular, you are a part of the subset of runners who actually are competitive about their personal records (PRs) and not the part that is in highk delusion-eternal-talking-stage-situationships. In conversation about the upcoming Boilermaker Half Marathon, you are arguing with your club members about who is the better runner based on your 5k and 10k times.

There are n runners in this discussion. Runner i has 5k time F_i and 10k time T_i (measured in seconds), where smaller values are better. Note that some runners may have the same PRs as each other. Runner i **gaps** runner j if $F_i \leq F_j$ and $T_i \leq T_j$ with at least one inequality being strict (runners with identical PRs do not dominate each other). A runner is **lowk-goated** if no other runner gaps them. Let h be the number of distinct PR pairs among the lowk-goated runners. Design an algorithm that outputs all lowk-goated runners, along with their frequency in increasing order of 5k time. Your algorithm should run in $O(n \log h)$ time.

For example, suppose the runners' (5k, 10k) times in seconds are

$[(1200, 2700), (1200, 2700), (1250, 2800), (1300, 2600), (1300, 2600), (1400, 2500), (1450, 2600)]$.

The output is $\boxed{[((1200, 2700), 2), ((1300, 2600), 2), ((1400, 2500), 1)]}$.

²Source: I made that shit up