A Printing Papers

Author: Prayaag

B Watch Your Sugar!

Author: Adithya

Solution Essentially, this problem requires a greedy solution where we are aiming to maximize the number of chocolates Thomas can have without crossing his limit. Observe that in order to maximize the quantity of chocolates he can eat, he must select the chocolates with the lowest amounts of sugar. As we are given a list of all the chocolates' sugar amounts, we must find the ones with the lowest amounts. One appearant approach involves sorting the list of chocolate sugar amounts in ascending order. That way, we have all the chocolates sorted by their sugar amount and we can proceed by counting how many chocolates Thomas can have without crossing the limit. We simply loop element by element in this sorted list and maintain a count as long as we don't cross a limit or reach the end of our list. Finally, we output that count. Since at each iteration, we are selecting the chocolate with the lowest amount of sugar that hasn't already been counted, we can be sure our solution is succinctly correct.

C Flush-tastic Throwing Challenge

Author: Bennett

Implementation This problem is pretty much all about implementation. Given a bunch of points, we need to find whether they lie inside or outside of a circle. Given the input sizes, it suffices to do this for each individual point. For each point, we can find its distance from the circle center using the Pythagorean theorem. If the distance $\sqrt{(x-x_c)^2+(y-y_c)^2} \leq r$, then the point must lie within the circle. Since we don't want to deal with square roots and floating point numbers, we can instead check the square of this, $(x-x_c)^2+(y-y_c)^2 \leq r^2$. To get our answer, we can increment a counter for each such point.

D Speedy Stamping

Author: Jimmy

Recursive Solution Let's start with a simple way to calculate the answer – simply generate every possible way to create the string. We can do this with a recursive brute-force solution: take turns appending each stamp T, C, TC, or CC to the current string, then recurse and try appending each stamp to *those* 4 strings, then recurse and try appending each stamp to *those* 16 strings, and so on. When the string exceeds or equals the target string in length (our base case), we check if it matches the target string, and increment the answer if so. This runs in $O(4^{|S|} \cdot |S|)$ time (corresponding to an exponential recursive function with a branch factor of 4, and then the final step of comparing the generated and target strings).
We can improve this solution by noting that we can throw out strings immediately when they don't match the target string, rather than waiting until the string is fully generated. Unfortunately, this new recursive backtracking solution still runs in exponential time, and is too slow to pass.[1]

[1]Specifically, we can observe that only one or two stamps are possible at any time – either the next character is a T, or the next character is a C. So this solution runs in approximately $O(2^{|S|})$ time.

Dynamic Programming Solution We can observe that if we currently have a string s, it doesn't really matter how s was generated, only how many ways s can be completed into the target string. This leads us to a dynamic programming approach. We use an array to store the number of ways to generate $S.substring(0, i)$ for each index i. Then, we can iterate over $i = 0...|S|$ and use these recurrence relations:

- If the next character is C, dp$[i+1]$ += dp$[i]$
- If the next character is T, dp$[i+1]$ += dp$[i]$
- If the next two characters are TC, dp$[i+2]$ += dp$[i]$
- If the next two characters are CC, dp$[i+2]$ += dp$[i]$

which gives us an $O(|S|)$ solution.

E Brainless Brainstorming

Author: Arthur

Brute Force To start, we can think of trying to invite each of either Jim, Dwight, or Kevin for each meeting slot. This can be done recursively while also making sure not to have two consecutive meetings. Unfortunately, this will not run in time because there are 3 choices for each time slot for about $3^{N/2}$ different possibilities. For $N = 1000$, this is much too large.

Dynamic Programming An observation that simplifies this problem is we can replace create an array max[] where $max_i = max(a_i, b_i, c_i)$. This is because if we are going to invite someone at a given meeting point, it is optimal to invite the one with the most ideas.

Then the problem reduces to computing the max over an array where you cannot select two consecutive elements. This is a classic DP problem, where we can maintain $dp[i] = max(dp[i-2], dp[i-3]) + arr[i]$.

The answer will just be the maximum $dp[i]$ over all i.

F Toilet Orders

Author: Amit Joshi

Solution One solution to this problem is to find the greatest common divisor of the two numbers b and l using the Euclidean gcd algorithm to find $g = gcd(b, l)$, which is $O(log(min(b, l)))$. Then, use the Sieve of Erastothenes to calculate prime numbers up until 10^6 in $O(n(log(log(n))))$ time. Note that small prime factors of g (less than or equal to 10^6) can appear multiple times in g. Now, evenly divide g by each small prime factor p. The number of times p evenly divides g is its multiplicity. By contrast, large prime factors of g can have multiplicity at most one in g, since if its multiplicity was greater, then $b, l \geq g \geq q^2 > (10^6)^2 = 10^{12}$, which goes over the bounds of the problem, as both $b, l \leq 10^{12}$. Not only that, there can be at most one large prime factor q in the prime factorization of g via a similar proof (if primes $q_1, q_2 > 10^6$ were both factors of g, then $b, l \geq g \geq q_1 * q_2 > 10^12$). Now, since g has been divided by all its small prime factors, the number that remains is either 1, meaning that there are no more prime factors in g, or it is a large prime number, since no small prime number divided it.

Alternate Solution Another solution to this problem is to find the prime factorizations of b and l separately, and then calculate the multiplicity of each prime factor p of g by taking the minimum of p's multiplicity in b, and p's multiplicity in l (which can be calculated in the same way as the prime factorization of g was calculated above). This solution is more based on the number theory that inspired the problem.

> *Proposition: The multiplicity of p in g equals the minimum of its multiplicity in b and its multiplicity in l.*
>
> Proof: Let e_l be the multiplicity of p in b and let e_b be the multiplicity of p in l. Without loss of generality, let $e_l \leq e_b$. Let e_g be the multiplicity of p in g.
>
> Let's say for the sake of contradiction that $e_g < e_l$. Then, $p^{e_g}|g|l, b$. But, consider $e_g+1 \leq e_l \leq e_b$. Then, $p^{e_g+1}|l, b$. Denote $g' = g/p^{e^g}$. Since $g|l \implies g'|l$. Then, $p^{e_g+1} * g'|l, b$ since p^{e_g+1} and g' share no common prime factors. But this contradicts the fact that $g = gcd(b, l)$ since $p^{e_g+1} * g' > g$.
>
> Let's say for the sake of contradiction that $e_g > e_l$. But then, $p^{e_g} * g' = g$ so $p^{e^g}|g|l, b \implies p^{e_g}|l$. But this contradicts the fact that e_l is the multiplicity of prime factor p in l.
>
> So, we know that $e_g = e_l = min(e_l, e_b)$. QED.

G Crappy Typing

Author: Yu

Brute Force The problem guarantees that an M exists which will ensure the contest finishes in time. M can be at most N, since any $M > N$ will contain redundant computers that do not accomodate any employee. M can be at least 1 if the sum of the times of all employees is $\leq D$. Thus, we can simulate the contest for all M from 1 to N until we find the first M that allows the contest to finish on time. However, since there are at most 10^5 employees, simulating this process will be too slow if M is large (e.g. $M = N$).

Binary Search Notice, that as M grows larger, the total running time of the contest either decreases or stays the same. Thus if we know for some M', where $1 \leq M' \leq N$, that M' computers will ensure the contest finishes on time, then all $M > M'$ will also ensure that the contest finishes on time. On the flip side, if M' computers will make the contest run overtime, then all $M < M'$ will also cause the contest to go overtime. Thus, we can binary search for M' between 1 and N. At most we will have to do $O(logN)$ operations to search for M', and for each operation we will have to run through all N employees to finish the simulation. This makes binary search solution $O(Nlog(N))$, which is fast enough given $1 \leq N \leq 10^5$.

Implementation As for simulating the contest, we can keep a priority queue of size M as well as the total time taken. Inside the priority queue, we store the time that each employee at a computer will finish the test. When an employee finishes, we insert the next employee in line into the queue, represented by the (current time) + (time the employee will take to finish the test).

H Crapper's Collapse Catastrophe

Author: Neo

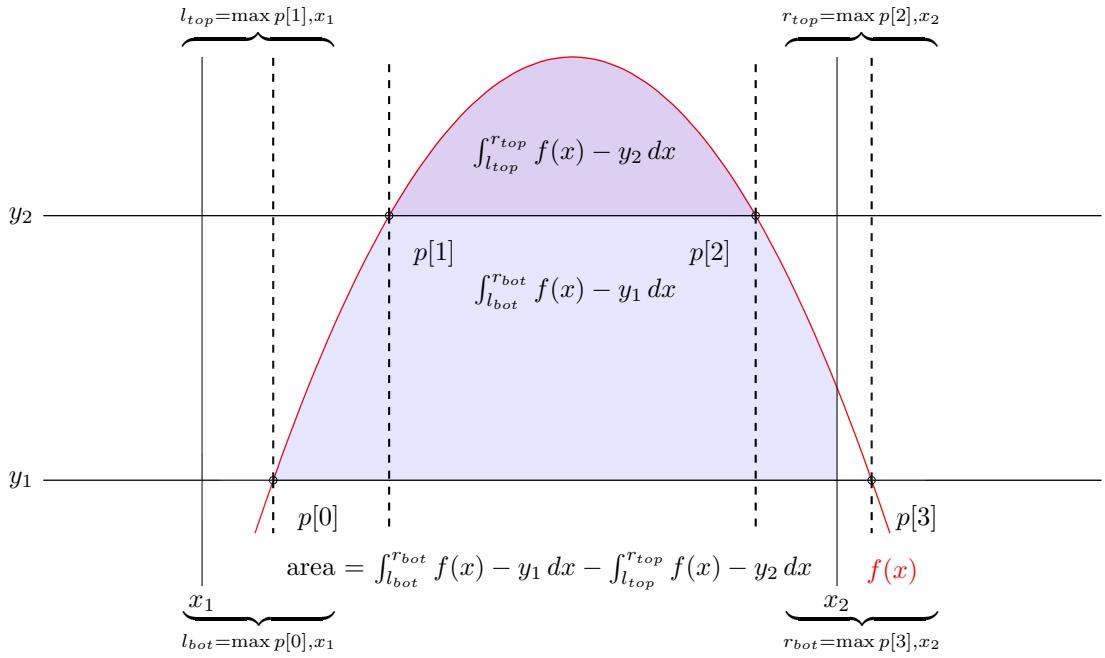
Solution Notice that the tower forms a tree that alternates between nodes per level. Essentially, this problem is asking us for the lowest common ancestor of this tree. Since each node has at least two children, you can notice that the tree is bounded by $\log_2 10^9$ which is roughly equivalent to 32. So how do we get from a node to its parent? Notice that we can construct 'levels' for each floor. If we know the index of the left-most node of the tree then we can compute if each node is on an even level or an odd level and then we can then deduce the parent of each node. Then, we maintain pointers for the value of x and y as they

walk down the building, moving whichever one is higher to its parent. Eventually we can see that the nodes will meet (convince yourself that this is true). Then, the only hard part remains is to get a node's parent. We can alternate and multiply to get the leftmost node for each level, and then we can create a mapping from each node to its parent by calculating the distance to its left node and then dividing by the 'branching factor' of the layer beneath it. For example, in the example case, we can see that 11 is 3 away from the left most node 9. Then, it's parent must be $3 + (11 - 9)/2 = 4$.

I Drunk Coworker

Author: Michelle

Implementation In order to find the safe region, we need to find the total area that does not lie between the two parabolas shifted k units above and below the original function. For now, we can focus on finding the area under one parabola. Let's multiply all coefficients by -1 if the parabola is facing up so that it faces down. Then we can consider two cases: one where we have two intersection points, so we integrate from the max of the left side of the room and leftmost intersection point to the min of the right side of the room and rightmost intersection point, and one where we have four intersection points, so in addition we will have to subtract the top part of the parabola from our overall area. The diagram below shows the casework for this.



J High Jump

Author: Zeki

Observations Let us first make some basic observations about the problem. Firstly, if some employee has a jump height J and the heights of the toilet stacks in each of the tiles are $h_1, \ldots, h_k$, then when can the employee reach the end? Well, this is if $J \geq \max(0, h_2 - h_1)$ and $J \geq \max(0, h_3 - h_2)$ and so on all the way up to $J \geq \max(0, h_k - h_{k-1})$. In English, this just means that the employee can jump from tile 1 to tile 2, from tile 2 to tile 3, and keep jumping all the way to tile k.

Let $H := \{\max(0, h_{i+1} - h_i) \mid i \in \{1, 2, \ldots, k-1\}\}$. By our analysis above, the employee will reach the end if and only if $J \geq \max(H)$. We have now simplified the problem a bit. If we can somehow maintain this "maximum difference" round by round, it will make checking how many employees can finish the round a lot easier.

One other observation – while we are tracking this "maximum difference", how does it change with respect to the updates? Well, if we have $l \leq i, i+1 \leq r$, then $\max(0, h_{i+1} - h_i) = \max(0, (h_{i+1} + x) - (h_i + x))$, so clearly the jump height necessary to go from i to $i+1$ does not change if they are both in the update interval. We realize that the only relative heights that *do* change are those which are on the boundaries of the interval. For example, if $i + 1 = l$, then in general $\max(0, h_{i+1} - h_i) \neq \max(0, (h_{i+1} + x) - h_i)$ (here h_{i+1} changes due to the update but h_i does not, there is a similar and symmetric case for the other boundary of the interval).

What this means is that we only need to observe the boundaries of the interval when we handle each update to check if we need to reconsider the current answer.

Implementation (Handling Updates) How do we handle range updates for this problem? The answer comes in the form of the Binary Indexed Tree (BIT). This allows us to perform range add updates and point queries (meaning we can ask the BIT what the height of a specific tile is as well as add a constant height to each tile in a range). This is exactly what we want, because we can check the heights on the boundaries of the interval before and after we add the query height, which allows us to interact with the final data structure we will need for this solution, a map.

The map we will use will track the frequency of each adjacent height difference (ignoring those which are zero). This way, we can quickly query $\max(H)$ by looking at the largest key in the map, and we can also quickly add and remove keys based on their frequency when we interact with the BIT. The final detail is that we need to use $\max(H)$ in order to calculate what our output should be. This is easy enough, for example if we sort the jump heights in the input, we can use C++'s `lower_bound` function (or whatever your favorite language's standard library offers) to calculate exactly how many people have jump heights of at least $\max(H)$. My C++ implementation can be found here.

K Office Odyssey

Author: Steph

Rooting Undirected Trees The vast majority of tree problems start by picking an arbitrary vertex to root the tree at. This provides numerous simplifications, such as the ability to compute Least Common Ancestors (LCAs) in $\log n$ time, but for this problem we'll specifically look at what it does to paths.

Firstly, every path from a to b can be viewed as two shorter paths: an upwards path from a to the LCA, and a downwards path from the LCA to b. We'll treat the fast LCA computation as a black box, but you can read more about it here.

Now suppose we have two paths, a to b and c to d that intersect at some shared node s. WLOG, s is on the upwards part of each path (if not, we can just reverse the path). The key observation here is to look at what happens as we continue along the path — since the parent of each node is unique, then taking one step upwards from s is the same for both paths. And as long as both paths haven't reached the LCA, both paths will keep taking the same steps. Importantly, this means that if two paths intersect, *the LCA of one path must lie on the other*. It's not hard to see that the converse is true, and thus we can conclude this LCA condition is equivalent to paths intersecting.

Counting LCAs At this point, we can easily attain an $O(nm)$ solution by first marking the LCA of each path (possibly multiple times if a node is the LCA for multiple paths), and then naively summing the number

of marks along each path. However, this isn't fast enough, and we'll need to find a way to count the number of marks faster than just following the path.

However, there's another trick we can employ here, again using rooted trees. Let $m(a, b)$ be the number of marked nodes along the path from a to b. Then, we have

$$m(a, b) - marks[lca] = m(a, root) + m(b, root) - 2m(lca, root)$$

Thus, by just computing $m(v, root)$ for all vertices v (which can be done via a simple DFS), we can (almost) compute the sum along any path we want. This doesn't count the number of marks on the LCA itself, but that's fine – we need to handle that separately anyways, since we'd otherwise double count the number of pairs of paths which share the same LCA.