Task 1: `Topical`

Authored by: Benson Lin Zhan Li

Prepared by: Tan Yi Kai

Editorial written by: Tan Yi Kai

Subtask 1

Additional constraints: $n = 1$

Since $n = 1$, the answer can only be 0 or 1. Benson starts with zero knowledge, so $p[j] = 0$ for all topics j. Hence, he can only complete the first module if $r[1][j] = 0$ for all topics j, and the answer would be 1. Otherwise, the answer is 0.

Time complexity: $O(k)$

Subtask 2

Additional constraints: $1 \leq n, k \leq 100$

Notice that at any point in time, if Benson has not completed module i and has sufficient knowledge to do so (i.e. $p[j] \geq r[i][j]$ for all j), then he should complete module i as that will not stop him for completing any other module.

Initialise knowledge $p[j] = 0$ for all topics. Then, check for all modules whether Benson has sufficient knowledge to complete any module that hasn't been previously completed. This can be done with a simple $O(nk)$ search, by checking for all n modules whether he has previously completed it and if $p[j] \geq r[i][j]$ for all k topics.

When a module that can be completed is found, mark the module as completed, increment the answer, and increase $p[j]$ by $u[i][j]$ for all topics.

Searching for a next module to complete can take up to $O(nk)$. Benson can complete up to n modules. Hence, the worst case time complexity is $O(n^2k)$.

Time complexity: $O(n^2k)$

Subtask 3

Additional constraints: $k = 1$

With only $k = 1$ topic, notice that it is always optimal to complete the modules in increasing order of knowledge requirement. We can never complete more modules by completing a module with larger $r[i][1]$ before a module with smaller value.

Hence, we first sort the modules by $r[i][1]$. Maintain the knowledge of topic 1 $p[1]$, initialising it with value 0. Starting from the module with lowest $r[i][1]$, we check if $p[1] \geq r[i][1]$. If yes, we increment the answer by 1, increase $p[1]$ by $u[i][1]$, and continue processing the module with the next lowest $r[i][1]$ value. Once we reach a value where $p[1] < r[i][1]$, we stop the processing as the knowledge $p[1]$ will not fulfil the remaining modules' knowledge requirements anyway.

Note that the scientific committee does not have a solution in Python that attains this subtask, as sorting 1 million numbers in Python exceeds the time limit.

Time complexity: $O(n \log n)$

Subtask 4

Additional constraints: None

We use the same idea as subtask 2. However, we will need a faster way to find a module that can be completed next.

To do this, for each module i, maintain a count $c[i]$ of the number of topics j where $p[j] \geq r[i][j]$. Initially, $c[i]$ is the number of topics where $r[i][j] = 0$.

To maintain $c[i]$ for all modules: for each topic j, we store a sorted array of knowledge requirement and their modules, as well as an index $ind[j]$ of the largest knowledge requirement that $p[j]$ fulfils. Whenever $p[j]$ is increased by completing a module, we increment $ind[j]$ until the next knowledge requirement is greater than $p[j]$. Before $ind[j]$ is incremented, $c[i]$ for the module of $ind[j]$ should be incremented.

To quickly find modules that can be solved next simply store the modules that have $c[i] = k$ which can be tracked whenever $c[i]$ is incremented.

Time complexity: $O(nk \log n)$

Task 2: `Inspections`

Authored by: Benson Lin Zhan Li

Prepared by: Benson Lin Zhan Li

Editorial written by: Stuart Lim Yi Xiong

Subtask 1

Additional constraints: $1 \leq n, m, q \leq 200$

For each task, in the worst case, all n machines are run. Hence, at most nm machines are run during the entire process. For each of q safety values, we can simulate the entire process of running a machine each day.

For each machine k, let $last[k]$ be the last day that machine k was run. Initially, let $last[k] = \infty$ for all k, where the value ∞ denotes a machine that has not been run once. We make updates to $last$ throughout the simulation.

Suppose we run machine k on day d. Assuming $last[k] \neq \infty$, let $\Delta d = d - last[k]$ be the number of days since machine k was last run. Benson needs to make an inspection if and only if $s[q] < \Delta d$. We can store the number of inspections needed in a counter and increment it whenever the above conditions are met.

Time complexity: $O(nmq)$

Subtask 2

Additional constraints: $1 \leq n, m \leq 2000$

Notice that each time a simulation is performed in Subtask 1, all values of Δd are the same. We can run the simulation once to find all Δd in $O(nm)$ time.

For each query, we want to find the number of Δd that are strictly larger than $s[q]$. To do so, we can store all Δd in a sorted array and perform binary search.

Time complexity: $O((nm + q) \log(nm))$

Subtask 4

Additional constraints: $1 \leq m \leq 2000$

Considering Ranges of Machines

During a task, the machines are run on consecutive days from $l[i]$ to $r[i]$, so their $last$ values after updates also take consecutive values. In these cases, we can store $last$ values for ranges of machines using what we will call *range tuples*. These are tuples in the form (b, e, t) that define the values of $last[k]$ for the range $b \leq k \leq e$ as follows.

$$last[k] = \begin{cases} \infty & \text{if } t = \infty \\ t + k & \text{otherwise} \end{cases}$$

(Note: The function $t + k$ was chosen for simplicity as well as taking later subtasks into consideration, but any similar function that is linear with respect to k will work for this subtask.)

We want to partition all of the machines into disjoint ranges, such that each machine is in exactly one range and each task covers all or none of the machines that are in the same range. It is possible to find a partition with at most $2m + 1$ disjoint ranges. Let B be the set of indices where ranges begin. We have

$$B = \{1\} \cup \{l[i] \mid 1 \leq i \leq m\} \cup \{r[i] + 1 \mid 1 \leq i \leq m, r[i] \neq n\}.$$

1 and $l[i]$ are natural choices because machine labels begin with 1 and tasks start from $l[i]$. Since tasks end at $r[i]$, it is necessary to split $r[i]$ and $r[i] + 1$, so $r[i] + 1$ is included in B as well.

After removing duplicates, let $|B|$ and B_h be the number of elements in B and the h-th smallest value in B respectively. Then we obtain the ranges $[B_1, B_2 - 1], [B_2, B_3 - 1], \ldots, [B_{|B|}, n]$. It can be easily shown that for any given range and any given task, the range is either completely covered by the task or not covered at all.

Simulating Ranges of Machines

We simulate the performance of tasks, but this time we run entire ranges of machines together. Let R be the list of range tuples. $R = \{(B_1, B_2 - 1, \infty), (B_2, B_3 - 1, \infty), \ldots, (B_{|B|}, n, \infty)\}$ initially. We make updates to R throughout the simulation.

Each of the m tasks cover $O(m)$ ranges of machines, giving $O(m^2)$ steps in the simulation. For any range covered by the current task, let the old and new range tuples be (b, e, t_0) and (b, e, t_1) respectively. t_1 can be calculated by observing that d and t_1 replace $last[k]$ and t_0 respectively and are therefore related by $d = t_1 + k$. If $t_0 \neq \infty$, then we have

$$\begin{aligned}\Delta d &= d - last[k] \\ &= (t_1 + k) - (t_0 + k) \\ &= t_1 - t_0\end{aligned}$$

for all k where $b \leq k \leq e$. Hence, all $e - b + 1$ machines in this range give the same Δd. We can store all Δd in (value, frequency) pairs. There are $O(m^2)$ pairs, but pairs with the same value can optionally be combined to give $O(m)$ pairs with distinct Δd — this will be proven under Subtask 3.

Answering Queries

Recall that we want to count the number of Δd that are larger than $s[q]$. Sort the Δd pairs by decreasing Δd value. Observe that as the safety value decreases from infinity to zero, we add the frequencies of new Δd pairs to the answer. Using this fact, we answer queries in decreasing $s[q]$, instead of doing so in the order that they are given. This requires us to sort the queries as well.

Time complexity: $O(m^2 \log m + q \log q)$ or $O((m^2 + q) \log(m^2 + q))$

Subtasks 3 and 5

Additional constraints: $l[i] = 1$ (Subtask 3), None (Subtask 5)

In the solution to Subtask 4, we only edited the t values in the range tuples. From here on, we edit the ranges during the simulation as well.

Initially, let $R = \{(1, n, \infty)\}$ because $last[k] = \infty$ for all k. We simulate tasks in order and consider all range tuples (b, e, t) in R such that $[b, e]$ intersects with the range $[l[i], r[i]]$ (representing machines run during task i). For each of these ranges, we calculate the size of their intersection with $[l[i], r[i]]$, as well as the value of Δd similar to Subtask 4. Since the machines in the range $[l[i], r[i]]$ are run on consecutive days, their $last$ values will also take consecutive values represented by a new range tuple $(l[i], r[i], t_{new})$ where t_{new} needs to be calculated. Existing tuples in R that intersect with the range $[l[i], r[i]]$ will make way for the new tuple, as described below.

- If $[b, e]$ is entirely contained within $[l[i], r[i]]$, the tuple is deleted. Let the number of such tuples be $c[i]$.
- Otherwise, if $[b, e]$ intersects $[l[i], r[i]]$, the tuple is shortened or split. There are at most 2 such tuples.

At most 2 tuples can be created during a single task. This happens when the task range $[l[i], r[i]]$ splits an existing range $[b, e]$, producing the ranges $[b, l[i] - 1], [l[i], r[i]], [r[i] + 1, e]$.

A range tuple that is counted towards $c[i]$ gets deleted, so the sum of $c[i]$ is bounded above by the total number of range tuples that exist, $2m + 1$. Hence, throughout the entire simulation process, we process $O(m)$ range tuples. (Since machines in the same range always give the same Δd, we have also shown that there are $O(m)$ distinct values of Δd.)

For Subtask 3, since $l[i] = 1$, therefore $[l[i], r[i]]$ always intersects with the first few ranges represented in R. Thus, tuples are always deleted from the start of R, with at most one tuple intersecting but not entirely contained in $[l[i], r[i]]$. There is no need to consider the case where a range is split. This simplifies implementation and also allows R to be stored as a stack data structure.

For the full solution, it is possible that $[l[i], r[i]]$ intersects only with the ranges in the middle or at the end of R. We use binary search to quickly find the first intersecting range. Since we also need to delete and insert range tuples in the middle of R, we store R as a set data structure.

We answer queries in the same way as Subtask 4.

Time complexity: $O(m \log m + q \log q)$ or $O((m + q) \log(m + q))$

Task 3: `Airplane`

Authored by: Lim Rui Yuan, Benson Lin Zhan Li

Prepared by: Lim Rui Yuan

Editorial written by: Lim Rui Yuan

Subtask 1

Additional constraints: $m = n - 1, u[j] = j, v[j] = j + 1$

The graph is a line in this subtask. There are a number of solutions to this, but we will present one which is helpful for future subtasks.

Observation 1

Observe that the height of the plane can be broken down into three phases:

1. Ascend: the height of the plane increases strictly
2. Maintain: the height of the plane stays the same
3. Descend: the height of the plane decreases strictly

Observation 2

Notice that the descend phase from some region t to the region n is a mirror of an ascend phase from region n to region t.

Returning to the line subtask, let region t is the t such that $a[t]$ is maximum. Consider the path from 1 to t. We should increase our height at every minute until it reaches $a[t]$, which is the ascend phase, then we maintain at height $a[t]$ and move to region t which is the first half of the maintain phase. Then, we repeat this computation but from n to t instead, which will give us the second half of the maintain phase and the descend phase.

Time complexity: $O(n)$

Subtask 2

Additional constraints: $n \leq 2000, m \leq 4000, a[i] \leq 2000$

Since the height limit per node is small, we can consider building a graph with $a_{max} \times n$ nodes, one for each region and for each height. Let the node (u, h) refer to the state the plane at region u and height h, we only have this node if $h \geq a[u]$.

For all pairs (u, v) that can be travelled, we draw the following edges in the expanded graph: (u, h) to (v, h), (u, h) to $(v, h + 1)$ and (u, h) to $(v, h - 1)$, which corresponds to moving to another node and changing height by 0, 1 and -1 respectively. Additionally, draw the edges (u, h) to $(u, h + 1)$ and (u, h) to $(u, h - 1)$ for all u, which corresponds to remaining the same node and changing your height.

On this new graph, we can run breath first search from node $(1, 0)$ and check the shortest distance to $(n, 0)$.

Time complexity: $O((n + m)a_{max})$

Subtask 3

Additional constraints: $n \leq 2000, m \leq 4000$

Suppose we fix some node t where $a[t]$ is the maximum height we will maintain at. We will try all t.

Let our path be $1 \to x \to t \to y \to n$, where $1 \to x$ is ascend, $x \to t \to y$ is maintain, and $y \to n$ is descend. For $x \to t$ and $t \to y$, we can run a bfs from t, ignoring all nodes with height larger than $a[t]$.

$1 \to x$ and $y \to n$ are symmetrical, so now we will only focus on 1 to x. We can compute the minimum time needed to reach all nodes x by ascending only by using a modified Dijkstra's Algorithm. In the modified Dijkstra's Algorithm, when relaxing node v from node u, we set $d[v] = max(d[u] + 1, a[v])$, where $d[u]$ is the shortest distance to node u.

Putting the two parts together, we can iterate each x to find the best x going from $1 \to x \to t$. Similarly, we iterate each y to find the best y going from $t \to y \to n$.

Time complexity: $O(n(n + m) \log n)$

Subtask 4

No additional constraints

We will need one more observation to fully solve this problem. Notice that if the maintain phase is 2 or more minutes long, we can always extend the ascend and descend phase each by 1 minute and shorten the maintain phase by 2 minutes. This means that the maintain phase should only be 0 or 1 minutes long.

From this, we just need to consider all nodes (for 0 minutes maintain) and all edges (for 1 minute maintain). After which, we can reuse the modified Dijkstra's Algorithm from subtask 3.

Time complexity: $O((n+m)\log n)$

Task 4: `Curtains`

Authored by: Benson Lin Zhan Li

Prepared by: Benson Lin Zhan Li

Editorial written by: Benson Lin Zhan Li, Lim Rui Yuan

Subtask 1

Additional constraints: $1 \leq n, m, q \leq 200$

For each query i, iterate through all m curtains. A curtain j can be used if $s[i] \leq l[j]$ and $r[j] \leq e[i]$; i.e. the curtain lies completely in the query range.

Among the at most m curtains that can be used for answering query i, individually mark each section from $l[j]$ to $r[j]$ as covered, and then later check if every section from $l[j]$ to $r[j]$ is covered to answer the query.

Time complexity: $O(nmq)$

Subtask 2

Additional constraints: $1 \leq n, m, q \leq 2000$

We perform the same iteration to find curtains that can be used for query i. However, we can no longer perform the same slow $O(nm)$. There are many ways to speed this up.

The easiest way is to use bitsets to do it in $O(\frac{nm}{64})$, resulting in overall time complexity $O(\frac{nmq}{64})$.

It is also possible to do it in $O(m \log m)$ by storing the covered sections as an ordered set of ranges, similar to the idea used to solve task 2 inspections. This results in overall time complexity $O(qm \log m)$.

Subtask 3

Additional constraints: $1 \leq n \leq 2000$

Notice that having at most 2000 sections means having only 2000^2 possible query ranges. As

such, we can do some sort of precomputation for all possible queries.

One way to do this in $O(n^2)$ is to use dynamic programming. Let $dp[l][r]$ be 1 if a subset of curtains can be chosen to cover only sections $[l, r)$ (inclusive of l, exclusive of r) and 0 otherwise.

Initialization: mark $dp[l][r] = 1$ if there exists a single curtain covering exactly $[l, r)$, and 0 otherwise. We now update the values of $dp[\cdot][\cdot]$ in lexicographical order
(i.e. $(1, 1), (1, 2), \ldots, (1, n), (2, 1), \ldots (n, n)$).

As we proceed, we store another variable j representing 'the largest value $i < r$ such that $dp[l][i] = 1$'. Keeping track of this variable is easy, if $dp[l][r] = 1$, we update i to r, else, $dp[l][r]$ remains unchanged.

To determine if $dp[l][r] = 1$,

- If there is a single curtain covering $[l, r)$, obviously $dp[l][r] = 1$.
- Else, there must exists some $j < r$ such that $dp[l][j] = 1$ and there exists some i between l and j such that there is a single curtain covering exactly $[i, r)$. We do not need to check all j such that $dp[l][j] = 1$, but instead the largest such $j < r$ with $dp[l][j] = 1$ is sufficient, which is nice because we already keep track of j. We simply need to check the existence i between l and j such that $dp[i][l] = 1$. There are several ways to do this, for example keeping a prefix sum array $p[i][l] = p[1][l] + \ldots + p[i][l]$.

Time complexity: $O(n^2 + q)$.

Subtask 4

Additional constraints: $s[j] = 1$

For this subtask, we can perform a sweep similar to subtask 3. Since $s[j] = 1$, we can let $ans[i]$ be 1 if the curtains can cover exactly $[1, i]$ and 0 otherwise.

Initially, for curtains where $l[i] = 1$, we can immediately set $ans[r[i]]$ to 1. We also want to maintain for each right endpoint R, what is the smallest L such that a curtain $[L, R]$ exists, or if no curtain ends at R.

Then, sweeping i from 1 to n, we maintain the current maximum j where $j < i$ and $ans[j] = 1$. Using the previously computed results, get the smallest L such that a curtain $[L, i]$ exists. If $L - 1 \leq j$, then $ans[i] = 1$. Else, 0.

To answer queries, simply check if $ans[s[j]] = 1$.

Time complexity: $O(n + m + q)$

Subtask 5

Additional constraints: $1 \leq n, m, q \leq 100000$

This subtask is meant for solutions with time complexity similar to $O(n\sqrt{n})$ or $O(n \log^2 n)$. Potential ways to achieve this include solutions using buckets of size $\sqrt{n}$ or divide and conquer. Here we present an $O(n \log^2 n)$ solution.

Consider some interval $[a, b]$ and all curtains whose **left** endpoint lies inside $[a, b]$ which we will denote as $S_{[a,b]}$ and sort by non-decreasing right endpoint. Suppose the query $[s, e]$ entirely covers $[a, b]$. Then the curtains in $S_{[a,b]}$ that we can use forms a prefix of $S_{[a,b]}$ since the left endpoint constraint is always in $[s, e]$. Call this $S_{[a,b]}(e)$.

This set of curtains that we can use may not cover all of $[a, b]$, and may also cover some sections outside of $[a, b]$. We notice that no such curtain can cover a section $x < a$, and if it covers a section $y > b$, then all sections in $[b, y]$ are also covered. Thus, we may associate each prefix of curtains in $S_{[a,b]}$ with two values:

- The largest index x in $[a, b]$ such that no curtain in the prefix covers x. (Hole)
- The largest index $y > b$ such that some curtain in the prefix covers y. (Reach)

Now consider two intervals $[a, b]$ and $[b + 1, c]$ alongside a query $[s, e]$ that entirely covers $[a, c]$. If $S_{[a,b]}(e)$ contains a hole, $S_{[b+1,c]}(e)$ will never be able to cover it. On the other hand, if $S_{[b+1,c]}(e)$ has a hole, $S_{[a,b]}(e)$ may be able to cover it depending on whether the reach of $S_{[a,b]}(e)$ is sufficiently large.

Additionally, the reach of $S_{[a,b]}(e) \cup S_{[b+1,c]}(e)$ is simply the larger of the reach of each of them. This means that, knowing the holes and reach of two adjacent intervals, we can determine if their union contains a hole and their reach. If we can divide the query range into a small number of intervals that cover the range, then we can efficiently determine if the final union of curtains contains a hole.

We build a segment tree where each node in the segment tree stores the curtains whose left endpoints lie in the range represented by the node, sorted by non-decreasing right endpoint. We then compute the hole and the reach of each prefix. This can be done efficiently by maintaining a disjoint-union data structure over each interval to check which sections have yet to be covered, or by employing a merge-sort idea to merge the results from child nodes to the parent.

When performing a query, we divide the query range into at most $2 \log n$ ranges using the segment tree. We then use a binary search on each node to determine the correct hole and reach

values for each segment. We can then process the segments from left to right. If no holes exist within the query range at the end, the answer is yes. Otherwise, the answer is no.

Building the segment tree is between $O((n+m)\log n)$ and $O((n+m)\log^2 n)$ depending on implementation. Each query involves $O(\log n)$ binary searches which are $O(\log n)$ each, so the queries are $O(q\log^2 n)$ in total.

Time complexity: $O((n+m+q)\log^2 n)$.

Subtask 6

Additional constraints: None

We present two different approaches that solve this task fully.

Solution 1: Offline Lazy Propagation Segment Tree

Since all curtains and queries are given upfront, we may perform some offline preprocessing. Sort the curtains and queries by non-decreasing right endpoint. We will process the curtains as updates and queries and queries. Do note that with the same right endpoint, updates take priority. We can do this because any query with a right endpoint of e will only ever use curtains with a right endpoint $\leq e$.

We use another modified segment tree. In each leaf node of the segment tree (i.e. a node only covering a singleton interval $[x, x]$), we store the largest index y such that at the current moment, there exists a set of curtains that covers $[x, y]$ exactly. Let this index be far(x). far(x) is initially set to $x-1$ as there are no curtains to cover $[x, x]$ yet.

Each curtain $[l, r]$ is effectively an update of the form: For all indices $1 \leq x \leq l$, if far$(x) \geq l-1$, set far$(x) = r$, else do nothing. We represent this update as $\langle l, r\rangle$. For each node of the segment tree, we lazily store this update and we will attempt to combine updates lazily as well.

Now suppose the current lazy update of a node of the segment tree with $\langle l_1, r_1\rangle$ and now will be updating it again with $\langle l_2, r_2\rangle$. Since we sorted the curtains and queries by non-decreasing right endpoint, we have $r_2 \geq r_1$. We have 2 cases:

- If $r_1 \geq l_2 - 1$, then any x such that far$(x) \geq \min(l_1, l_2) - 1$ will end up with far$(x) = r_2$. We can combine the updates into $\langle \min(l_1, l_2), r_2\rangle$
- Otherwise $r_1 < l_2 - 1$. We claim that we need not update $\langle l_1, r_1\rangle$ at all. Since all updates are prefix updates on the segment tree, $\langle l_2, r_2\rangle$ is the first update with $l_2 \geq l_1$ since $\langle l_1, r_1\rangle$

was set. This means that none of the sections in this range represented by the segment tree are able to cover the sections $[r_1 + 1, l_2 - 1]$ at all. Any future update that covers any of these sections will strictly contain $[l_2, r_2]$. In other words, the curtain $[l_2, r_2]$ is useless for these sections.

This means that the updates are armortized $O(\log n)$ per curtain. When querying, we start from the leaf node corresponding to the left endpoint of the query range and work up the segment tree, updating the value of far(x) as necessary. Each check is $O(1)$, so each query is $O(\log n)$.

Time complexity: $O((n + m + q) \log n)$

Solution 2: Divide and Conquer

We can use divide and conquer to solve this problem. Let's have a recursive function $dnc(X, Y)$. Let $m = \lfloor \frac{X+Y}{2} \rfloor$. During $dnc(X, Y)$, we will handle all queries j that satisfies $s[j] \leq m$ and $e[j] \geq m + 1$. After which, we will call $dnc(X, m)$ and $dnc(m + 1, Y)$ to handle all other queries that do not cross m.

We will now describe how to handle all queries that satisfy $s[j] \leq m$ and $e[j] \geq m + 1$. We will define $closeL(i)$ for $X \leq i \leq m$ to be the smallest j where $j \geq m$, such that it is possible to cover $[i, j]$ exactly. Similarly, $closeR(j)$ for $m + 1 \leq i \leq Y$ to be the largest i where $i \leq m + 1$ such that it is possible to cover $[i, j]$ exactly. A query range $[s, e]$ is good if and only if $closeL(s) \leq e$ and $closeR(e) \geq s$.

We will now describe how to compute $closeL$. When computing $closeL(i)$, consider all curtains where $l = i$. Among them, only two are important, the one with the smallest $r \geq m$, the one with the largest $r \leq m - 1$. For the former, that already is potential value for $closeL(i)$.

For the latter, we will need to find the minimum value of $closeL(k)$ for $i + 1 \leq k \leq r + 1$. We can do so by maintaining a monotonic stack storing pairs of $(k, closeL(k))$. We pop out all values in the stack where $k \leq r + 1$ and set the minimum value of $closeL(i)$ to that. After which, we will know the minimum value of $closeL(i)$. We can then push it into the monotonic stack.

We can compute $closeR$ similarly. Computing a single layer runs in $O(Y - X)$ time, and hence the algorithm runs in $O((n + m + q) \log n)$ in total.

Task 5: `Toxic`

Authored by: Benson Lin Zhan Li

Prepared by: Benson Lin Zhan Li

Editorial written by: Benson Lin Zhan Li

Definitions

For a multiset $S = \{b_1, b_2, \ldots, b_k\}$ of size k with $1 \leq b_i \leq n$ for all i, we let $rem(S)$ be the number of surviving bacteria if S was queried.

Identifying Toxic with Binary Search

We can easily test if a species is Toxic by testing each one individually and checking if the query returns 0. To improve upon this, we make the following observation: Suppose we query a sample S of size k. Then $rem(S) < k$ if and only if at least one of the species in S is toxic.

Thus, we can perform binary search to identify Toxic species. Given a **set** R of species where we **know** at least one is Toxic, we can query half of the species with one copy each. If there is at least one Toxic in this half, then we continue the binary search on this half. If not, we know the Toxic species is in the other half, so we continue the binary search on the other half. Hence, we can identify one of the Toxic in $\lceil (\log_2 |R|) \rceil$ queries.

This allows us to find all Toxic in $t\,(\lceil \log_2 n \rceil + 1) - 1$ queries by repeatedly binary searching for each Toxic species and removing it from the set of species to test. Note that the $+1$ is necessary as we need to query the entire set once before each binary search in case there are no Toxic left to identify.

Identifying Strong with Powers of Two

At this point, we realise that it is impossible to distinguish Strong from Regular without Toxic. Thus, we assume that we have found all the Toxic species, one of which we will call tox.

Testing one species that is either Regular or Strong with one species that is Toxic will yield an answer of 0 or 1 respectively. Simply testing each of the remaining species with tox in this way will identify the Strong and Regular in $n - t$ queries.

We can do better. We can use up to 300 specimens in a single sample, which means that we can use powers of two to identify which species die. Let us select 8 species $s_0, s_1, \cdots, s_7$ where we do not know whether they are Regular or Toxic. We create a multiset S such that for each $0 \leq i \leq 7$, we add 2^i copies of s_i into S. We then add one copy of tox into S.

When queried, tox is guaranteed to die. Then, for each i, the 2^i copies of s_i survive if and only if s_i is Strong. Thus, we can uniquely identify which s_i are Strong based on the binary representation of $rem(S)$.

Batching

Currently, we require $\lceil \log_2 n \rceil + 1$ queries to identify each Toxic species using binary search. Instead of binary searching on the entire set of species, we can split the n species into smaller groups and binary search on each group instead.

If we use a group size of c, then we need one query per group to check if there is a toxic in that group. This takes $\frac{n}{c}$ queries. Each toxic that we find then requires $\lceil \log_2 c \rceil + 1$ queries. It turns out that choosing $c = 8$ is optimal.

Solving $m = 188$

We combine the above 3 ideas into a solution that can solve the task in at most 188 queries.

The solution has 3 phases:

- **Phase 1: Toxic Group Identification**

  Split the n species into $\frac{n}{8}$ groups of 8. For each group, we perform a query using one copy of each species in that group. This will tell us if there is a Toxic in this group or not. If there is, we mark it for processing in Phase 2. Otherwise, we classify all of the species in the group as Regular/Strong as we cannot distinguish between those two yet.

  This requires $\frac{n}{8}$ queries.

- **Phase 2: Binary Search for Toxic**

  For each group that contains toxic, we perform a binary search to identify the Toxic in the group. This requires $\log_2(8) = 3$ queries, plus 1 more query to ensure that there are no more Toxic in the group.

  This takes $4t$ queries.

- **Phase 3: Strong Identification**

Now that we have identified all of the Toxic, we can use powers of two to identify the remaining Strong in $\frac{n-t}{8}$ queries

The following combinations of optimisations obtain lower partials (exact numbers will vary depending on implementation):

- Binary Search: 540 queries
- Powers of Two: 334 queries
- Binary Search + Powers of Two: 304 queries
- Binary Search + Batching: 428 queries

These optimisations together obtain around 50 points. The following sections focus on optimisations that improve the performance of the 3 phase solution mentioned above to obtain full score.

Randomisation

Since the grader is non-adaptive, we can randomise the order in which we query the species. This allows us to, with high probability, avoid edge cases where queries perform badly.

Powers of Two in Phase 1

Since the batch size in Phase 1 is 8, we can query these species in powers of two instead of just having one copy of each. If any of them die, we know there is at least one Toxic in the batch. Otherwise, there are no Toxic in the batch.

If there is a Toxic in the batch, then we can identify all of the Strong bacteria in the batch. The rest can be classified as Toxic/Neutral. If there is no Toxic in the batch, we can classify the remaining as Regular/Strong.

This optimisation on its own saves a few queries in Phase 3 since there are fewer species to check. Completely separating Strong and Toxic also helps with later optimisations.

Phase 2 and 3 Concurrency

Notice that in Phase 2, each query only uses at most 8 specimens and never contains any Strong bacteria. In Phase 3, each query uses at most 256 specimens and contains exactly 1 Toxic

bacteria, with the rest being either Regular or Strong. Thus, we can perform both queries concurrently.

Suppose we have a set $R = \{r_1, r_2, \cdots, r_k\}$ with $k \leq 8$ which contains at least one Toxic and no Strong and another set $S = \{s_0, s_1, \cdots, s_{l-1}\}$ with $l \leq 8$ which contains no Toxic. Then, we can perform a query containing half of R (which we will call R') with one copy each and 2^i copies of each s_i.

If any of the species in R' are Toxic, then every species in R' will die, and the only living specimens in the sample will be the Strong species in S, which we can uniquely identify. Otherwise all specimens in the sample will survive. We can then infer that all species in R' are Regular and obtain no new information about S. We can continue the binary search on the other half of R.

In other words, we can perform Phase 2 binary search queries and Phase 3 powers of two queries concurrently. We notice that in one binary search of a Toxic we either query the Toxic at least once and identify 8 Regular/Strong species using Phase 2 and 3 concurrency, or we have queried every other species in the group and identified them as Regular. Thus, in one binary search of Toxic, we identify at least 8 species no matter the results. Thus, we can expect that Phase 3 has at most $n - 8t$ species left to identify, bringing the absolute worst case number of queries down to $\frac{n}{8} + 4t + \frac{n-8t}{8}$ which is around 165 for $t = 30$.

Using randomisation, we can expect each query during Phase 2 has a half chance of querying a Toxic. Thus, we can expect that most of Phase 3 will be completed within Phase 2 when t is greater than 25, since we need $4t$ queries for Phase 2 and $\frac{n-t}{8}$ queries for Phase 3. The probability that all of Phase 3 is completed in Phase 2 is around 99% for $t = 25$ and gets exponentially closer to 100% as t increases.

Adaptive Batch Sizes

When we have a set of species S which may or may not have a Toxic, identifying a Toxic in this set needs $\lceil (\log_2 |S|) \rceil + 1$ queries. If $|S|$ is **not** a power of two and we still have other species which are either Regular or Toxic that need to be identified, we can add these into the set until we hit a power of two. The exact power of two varies depending on how many Toxic we still need to identify and the number of Regular/Toxic left.

One possible heuristic is, given t toxic and n remaining Regular/Toxic to search for, determine the power of two for c that minimises $\frac{n}{c} + t\,(\lceil \log_2 c \rceil + 1)$. This estimates the best power of two that minimises the number of queries necessary to identify all the Toxic, saving some of the binary search queries.