The CENTRE for EDUCATION in MATHEMATICS and COMPUTING

cemc.uwaterloo.ca

2025 CCC Senior Problem Commentary

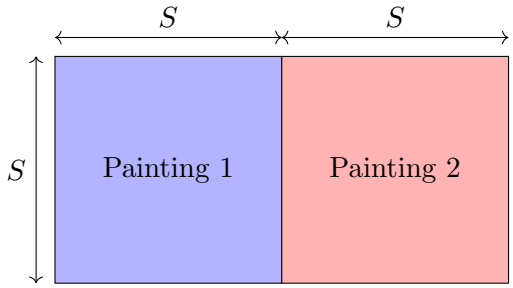
This commentary is meant to give a brief outline of what is required to solve each problem and what challenges may be involved. It can be used to guide anyone interested in trying to solve these problems, but is not intended to describe all the details of a correct solution. We encourage everyone to try and implement these details on their own using their programming language of choice.

S1: Positioning Peter's Paintings

The perimeter of a rectangle is $2(H + L)$ where H and L are the height and length of the rectangle.

Subtask 1

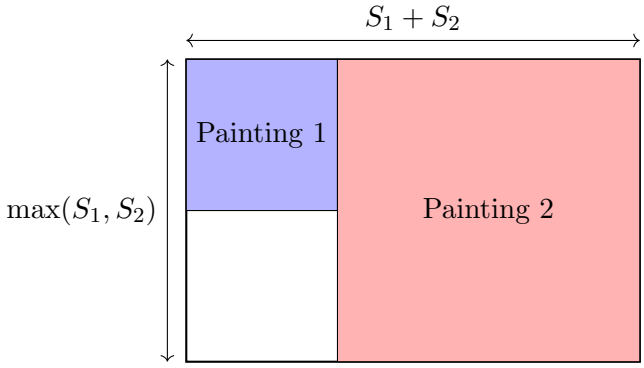
In this subtask, both squares are congruent. Let the side length of the paintings be S. An optimal configuration for all such cases is shown below.



Here, we see that the length of the rectangle with the smallest perimeter that encloses the two paintings has height of S and length of $2S$. This means that the perimeter is $2(2S + S) = 6S$.

Subtask 2

For this subtask, we have two squares. Let the side lengths of the paintings be S_1 and S_2 for Painting 1 and Painting 2, respectively. The best configuration is to put them side by side. An example of the optimal configuration is shown below.



Here, we see that the perimeter of the rectangle with the smallest perimeter that encloses the two paintings has a height of $\max(S_1, S_2)$ and length of $S_1 + S_2$. Thus, we see that the answer is $2(S_1 + S_2 + \max(S_1, S_2))$.

Subtask 3

For the last subtask, assume Painting 1 has a length of A and a height of B and Painting 2 has a length of X and a height of Y. Like in previous subtasks, the optimal configuration may be to put them side by side. However, there may be another case of stacking one painting on top of another painting. As such, we will take the minimum perimeter of each case.

- **Placed side by side:** The best enclosed rectangle has a height of $\max(B, Y)$ and a length of $A + X$. Hence, the perimeter is $2(\max(B, Y) + A + X)$.
- **Stacked on top of each other:** The best enclosed rectangle has a height of $B + Y$ and a length of $\max(A, X)$. Hence, the perimeter is $2(\max(B, Y) + A + X)$.

Thus, the answer is

$$\min(2(\max(B, Y) + A + X), 2(\max(A, X) + B + Y))$$

S2: Cryptogram Cracking Club

Subtask 1

For this subtask, we will reconstruct the repeated pattern using the cipher. To do so, we will alternate between taking a character input and an integer input. To obtain the pattern, we will maintain a string and append each character an according number of times to the end of the string. Finding the c-th character formed by repeating this pattern infinitely can then be done using a loop. We can maintain an integer representing the position of the c-th character in the pattern, and increment this value by one at each step of the loop. During the loop, if we ever reach the end of the pattern string, we will reset this integer to point back to the beginning of the pattern.

Subtask 2

This subtask can be solved similarly to the previous one. However, the count of how many times each character is repeated may no longer be between 1 and 9, so we cannot store this number in a single character. We can use the built-in input functions of programming languages to handle this. (For example, we can use `std::cin` in C++ to alternate between reading a character and a 32-bit integer.)

Moreover, since the length of the repeated pattern is larger, some care may be required to ensure its construction is efficient. One such method is appending the characters one by one to the end of a string, as described in Subtask 1. We can find the c-th character of the long cipher formed by repeating this pattern exactly as in Subtask 1.

Subtask 3

For the third subtask, the value of c can be very large. Thus, using a loop that steps through the pattern c times will no longer be efficient enough. Instead, we will make use of the property that we always loop back to the beginning of the pattern in a predictable way. If we denote the length of the pattern as $|S|$, then we will reset the integer keeping track of our current position in the pattern every $|S|$ steps. Thus, we can take the "remainder" of steps after dividing c by $|S|$. This can be done using the modulo operator. Then, the remaining number of steps needed will be less than $|S|$, the length of the string, which is at most 10^6 characters in this subtask. We can now use a loop to find this character (or access the desired character with direct indexing). Note that c may not fit within a 32-bit integer, and so we should store it in a 64-bit integer.

Subtask 4

For the final subtask, the length of the pattern can be very large. As such, we can no longer explicitly store it as a string. Instead, we will store it in a compressed format as a sequential list of pairs (x, y), where x is the character and y is the number of times that character is repeated. We can also compute $|S|$, the length of the pattern by summing all y that are processed. Note that c and the values of y may not fit within a 32-bit integer, so we should work with 64-bit integers instead.

Once we have constructed the list representing the compressed pattern, we will again set c to its remainder when divided by $|S|$. Now we will loop through the list. During this, c will always represent the number of characters that we still need to step through to arrive at the desired location. At each step, we will check if the character in the current pair (x, y) repeats at least c times. If so, then we step through each of these characters all at once by decrementing c by y and move to the next pair. Otherwise, we will arrive at an occurrence of the character x after the remaining c steps, so we will output that character as the answer. Some care should be taken in the implementation to avoid "off-by-one" indexing errors.

La version française figure à la suite de la version anglaise.

S3: Pretty Pens

Subtask 1

For this subtask, there are no modifications. For every colour c, we can order all pens of that colour by their prettiness. Let m_c be the ID of the pen with the highest (maximum) prettiness value with colour c and let s_c be ID of the second highest. Note that s_c may not exist for some c.

If we do not change any pen's colour, then the answer is

$$\sum_{i=1}^{M} P_{m_i}$$

where P_k is the prettiness of pen with ID k. This can be stored in a variable and used for further calculations. Observe that we can always achieve the optimal answer by only changing the colour of s_c. And the resulting answer if we change s_c to have colour d is

$$(\sum_{i=1}^{M} P_{m_i}) - P_{m_d} + \max(P_{m_d}, P_{s_c})$$

However, we cannot afford to try to change the colour of every pen s_c to every colour d as there are up to M^2 different (c, d) pairs to considers. However, we can notice that it is always optimal to pick the colours with the smallest P_{m_c} or second smallest P_{m_c}. This reduces to $2M$ pairs to check, which will run in time. This solution runs in $\mathcal{O}(N)$ time.

Subtask 2

There is only 1 colour for this subtask. Thus, the answer is the pen with the largest prettiness. Because there are updates, we can maintain this information in a dynamic data structure such as a balanced binary search tree (BBST) such as `std::set<std::pair<int, int>>` (the pair represents a pen by its prettiness and pen ID) or with a `std::multiset<int>` in C++ where the pens are ordered by their prettiness. To update, we also need a mapping from pen ID to their prettiness, which can be stored in an array. If implemented correctly, this runs in $\mathcal{O}((N + Q) \log N)$ time.

Subtask 3

Now the pens could be of 2 colours. Notice that we are always able to pick any two pens. Thus, we can always pick the two pens with the highest prettiness values. We employ a similar solution to the second subtask by maintaining the appropriate data structures and summing the two highest prettiness values instead of just taking the highest prettiness value. This runs in the same time as the intended Subtask 2 solution, $\mathcal{O}((N + Q) \log N)$ time.

Subtask 4

Now, there can be up to 10 colours. Here, we can solve this by using Subtask 1's observations and Subtask 3's data structures by applying Subtask 1 to every query. This results in a time complexity of $\mathcal{O}((N + QM) \log N)$.

Subtask 5

This subtask was meant to award contestants with the correct idea as Subtask 6, even if they have implemented parts incorrectly.

Subtask 6

La version française figure à la suite de la version anglaise.

For the final subtask, we can further reduce the $2M$ pairs to check to two cases:

- We change the colour of no pens.
- We change the colour of the s_c with the largest P_{s_c} to the colour of m_d with the smallest P_{m_d}.

Note that the s_c with the largest prettiness is the same as the pen that is not a m_c with the largest prettiness. We can find s_c with largest P_{s_c} and pen m_d with smallest P_{m_d} by having these data structures and values:

- A BBST of the pens of the form m_c ordered by prettiness
- A BBST of every other pen, which is not a m_c ordered by prettiness
- A BBST for every colour c, storing every pen of the same colour c ordered by prettiness
- An array of pens storing their prettiness and colour, indexed by their ids
- A value representing the sum of prettiness of the first BBST (Let this value be T)

After every query, we must quickly update all data structures. Now the answer is

$$\max(T, T - \text{smallest prettiness of the first BBST} + \text{largest prettiness of the second BBST})$$

If all the data structures are maintained properly, this solution runs in $\mathcal{O}((N+Q)\log N)$ time.

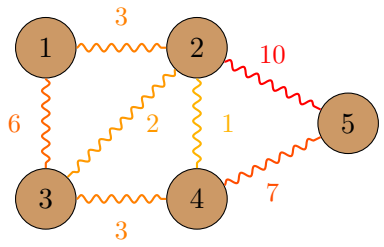
La version française figure à la suite de la version anglaise.

S4: Floor is Lava

Problem Overview and Observations

In the language of graph theory, we are given a graph with N nodes and M edges. Each edge has an integer label, which we will refer to as its *weight*.

For ease of discussion, we will use the graph given in the sample input, shown below, as an example throughout.



Subtask 1

Here, we are given a connected graph on N nodes and $N-1$ edges. Thus the given graph is a tree, so there is only one path from node 1 to node N that doesn't reuse any edges. We should never double back on our path, only changing our chilling level when necessary. Thus we can simply compute the cost incurred by traversing the simple path from node 1 to node N.

Subtask 2

Notice that our state in the dungeon is entirely determined by the pair (u, L) where u is the node we are at and L is our boots' chilling level.

Since all edge weights are between 1 and 10 inclusive, we should never adjust our chilling level above 10, as we would have to adjust it back down before we traverse the next edge. Similarly, we should never the level to be below 0. Thus there are only 11 chilling levels L that we need to consider. This means there are only $11N$ possible states we can be in.

We can carefully construct an undirected graph that represents the costs to transition between these states (u, L). To construct this graph, it helps to think of all of the actions we can take at a state (u, L). We can:

- traverse an edge that has an endpoint at u and weight L at a cost of zero, or
- change the chilling level of our boots by $+1$ or -1 for a cost of 1.

Notice that we don't need to consider changing the chilling level by more than 1 at a time: we can think of the operation "pay d coins to change the chilling level of your boots by d" as "pay 1 coin to change the chilling level of your boots by 1, as many times as we want".

Counting carefully, the undirected graph we construct has $11N$ nodes and $10N + M$ edges, and all weights are non-negative. We can use Dijkstra's algorithm to find the shortest path from the state $(1, 0)$ to all other states, and return the cheapest way to get to one of the states (N, x) for some x between 0 and 10.

Subtask 3

La version française figure à la suite de la version anglaise.

Before traversing an edge, we must adjust our chilling level to match the weight of the edge. Thus, for every two adjacent edges we traverse with weights x and y, we must pay the cost $|x - y|$.

Also, notice that once we have the right chilling level to traverse an edge, we essentially have access to both endpoints at the same time.

Combining these two observations, we notice that our state in the dungeon is determined by which edge e we last traversed. It doesn't matter in which direction we traversed it: conceptually, we can think of traversing e in one direction, and immediately traversing back if we so desire.

Thus we can again construct an undirected graph H that represents the costs to transition between the states e. For every two edges e and f that share an endpoint, we can transition between them at a cost of $|w(e) - w(f)|$, where $w(e)$ and $w(f)$ denote the weights of edges e and f, respectively.

Each state e is connected to at most eight other states, because the two endpoints can each have four other edges adjacent to them. The number of edges in H is then at most $8M$.

To account for the cost of the first edge, we should imagine that we traversed a weight-zero edge before we started our path. We can think of adding an edge e_0 from node 1 to itself with weight 0 in the original graph. We then return the minimum cost to get from state e_0 to any state e where e is incident to node N.

Subtask 4

We try a similar technique to subtask 3, but now since the degree of a node can be much higher than 5, the H we would construct may have too many edges.

The main idea is to reduce the number of edges in the graph we construct. Consider node 3 in the sample graph. The state transitions constructed from the edges incident to node 3 are:

- the transition between edge $(3, 2)$ and $(3, 4)$ with cost $|2 - 3| = 1$,
- the transition between edge $(3, 2)$ and $(3, 1)$ with cost $|2 - 6| = 4$, and
- the transition between edge $(3, 4)$ and $(3, 1)$ with cost $|3 - 6| = 3$.

Notice that the second transition is already "covered" by the other two. If our chilling level is at 2 and we want to move from $(3, 2)$ to $(3, 1)$, it does not cost us anything to make a "pit stop" at $(3, 4)$ with weight 3, because we will be increasing our chilling level beyond 3 anyway. This reasoning is similar to why we only have to consider $d = 1$ in the discussion of subtask 2.

Thus way, a node with degree t only adds $t - 1$ transitions to our graph. The number of nodes and edges are both $O(M)$ in this case.

S5: To-Do List

Problem Overview and Observations

This task is a scheduling problem. It asks us to decide when to work on the assignments so that they're finished as soon as possible. Let's use the term *job* instead of "assignment" and say a *schedule* is an arrangement of when to do the jobs: both how to order them and when to position them in time. These are terms from *scheduling theory*, which studies problems like this one. In this editorial we won't assume knowledge of scheduling theory, but it'll be helpful to borrow a few more terms:

- The *release time* is the moment when we can start working on a job. We'll denote the release time of job i with r_i. It's related to the input s_i by the equation $r_i = s_i - 1$.
- The *completion time* of a job is the point when it's finished, in the context of a particular schedule.
- The *makespan* is the maximum completion time of all the jobs. We're trying to minimize it.

Let's begin by observing the case with just two jobs. If one's released before the other, it's best to do the first job first and second job second.

This strategy—doing jobs in the order they're released—also works for any number of jobs. Let's see why. Re-number the jobs (say there are n of them) from 1 to n so that $r_1 \leq \cdots \leq r_n$. That is, the jobs are ordered in nondecreasing order of r_i.

But how do we know there's no strictly-better schedule with a different ordering? We'll use what's called an *exchange argument*. We'll show that any schedule can only improve when sorted by our ordering. Sorting algorithms such as bubble sort tell us that we'll eventually sort a list if we repeatedly swap pairs of adjacent out-of-order elements. A pair of jobs that are out of order would be appear in the order (j, i) but have $i < j$ according to our numbering, and $r_i \leq r_j$.

Like in bubble sort, we want to swap the jobs to the order (i, j). We've considered the case of two jobs in isolation, and this situation is similar. Let say that while swapping, we keep all other jobs fixed. In the (j, i) schedule, both jobs begin after r_j. But since $r_i \leq r_j$, the jobs begin after both r_j and r_i. So, the jobs can be swapped to (i, j) while maintaining the release-time constraints and makespan. By the end, we'll have a schedule with the same makespan but all the jobs sorted. Therefore, there'll always be an optimal solution to the overall problem among the sorted schedules.

Let's find the minimum makespan of these sorted schedules. Consider a schedule that's optimal, and – makespans being equal – prefers to schedule jobs earlier rather than later. It ends with a clump of jobs with no gaps between them. Then the job i at the start of clump will begin at r_i: otherwise, it could have started earlier. The length of the clump is $\sum_{i \leq j \leq n} t_j$. We'll shorten this to $t([i\,..])$, the idea being that $t(S) = \sum_{j \in S} t_j$ and $[i\,..] = \{i, i+1, \ldots, n\}$. So the schedule's makespan is $r_i + t([i\,..])$, for some i. But this expression is also a lower bound: if the release constraints r_j didn't exist for $j > i$, we still couldn't finish sooner than $r_i + t([i\,..])$. This gives a formula for the optimal makespan: $\max_i (r_i + t([i\,..]))$.

This formula calculates the answer once we've sorted the jobs by release time. But how do we maintain the answer while changing the sequence? As a first step, let's define a "summary" function. It should have these properties:

- Every contiguous segment of elements has a summary.
- The summary of the entire sequence includes the answer we're trying to find.

La version française figure à la suite de la version anglaise.

- The summary of a segment—even a very large segment—is made up of just a few numbers.
- If we concatenate segments A and B to form AB, we can combine summary(A) and summary(B) to calculate summary(AB).

If you're familiar with *segment trees*, you'll know that summary functions are an important idea. If a segment of elements doesn't change, neither will its summary. Because a summary is much more compact than the segment it represents, data structures for combining summaries are efficient at assembling the final answer.

Here's how we might find a summary function for the sequence of jobs when they are sorted by nondecreasing release time. The summary of the segment AB should include the local version of the answer, which is $\max_{i \in AB}(r_i + t(AB \cap [i\,..]))$. The optimal i for this expression is in either A or B. So, we can consider cases:

$$\begin{aligned}
\text{answer}(AB) &= \max_{i \in AB}(r_i + t(AB \cap [i\,..])) \\
&= \max\left(\max_{i \in A}(r_i + t(A \cap [i\,..]) + t(B)),\ \max_{i \in B}(r_i + t(B \cap [i\,..]))\right) \\
&= \max(\text{answer}(A) + t(B), \text{answer}(B))
\end{aligned}$$

This formula combines the answer of A and B, as long as we also know $t(B)$. Luckily it's easy to maintain the sum of t in the summary: $t(AB) = t(A) + t(B)$. This pair of numbers—the answer and the sum of t—is a summary that satisfies all of our properties.

However, we aren't done yet. We still need to decide which data structure to use, and how to implement it. Things will be easy for us if we take advantage of the constraint $s \leq 10^6$. Rather than considering the sequence of jobs, let's shift our perspective to consider the sequence of numbers 0 to $10^6 - 1$, which are all the possible values of r_i. This sequence has the advantage of being static: for a static sequence, we can use a data structure with a static binary tree.

We'll need to tweak our summary function so that it summarizes contiguous segments of release times. It'll still describe a segment of jobs, but it'll be defined by a segment of time, not the jobs themselves. In particular, we'll need to handle segments with no jobs, and handle multiple jobs sharing the same release time.

Now we can use one of the standard segment tree implementations, such as the one that follows. Consider the *implicit binary tree* defined by the following: the root is node 1; the left child of node k is $2k$; the right child of node k is $2k + 1$. The nodes from 2^{20} to $2^{21} - 1$ are all on the same level of the tree. The node at $2^{20} + r$ will correspond to the release time r (this works since $10^6 < 2^{20}$). Any nodes higher in the tree will represent the segment of release-time nodes that are its descendants. Use an array of size 2^{21} such that the data at index k corresponds to the implicit binary tree node k. The type of the data will be our summary—a pair of numbers. Whenever we add or remove a job, we'll update the node for the corresponding release time. Then we'll walk up the implicit binary tree while recalculating the summary of every node we visit. Finally, the summary of the root node will have the answer.

This analysis used the constant upper bound of 10^6. If we instead call this $s_{\max}$, our complexity is $O(s_{\max} + Q \log s_{\max})$.

Subtask 1

Let's go back to the point when we proved that we can sort the tasks in nondecreasing order of release time. What's the easiest way to solve the first subtask? Given the constraints, we should look for a linear-time way to calculate the makespan given the sorted list of jobs.

La version française figure à la suite de la version anglaise.

We'll consider the jobs in order from 1 to n. Let C_i denote the completion time of job i. The earliest we can complete job 1 is $r_1 + t_1$. For every subsequent job i, the earliest it can start is $\max(C_{i-1}, r_i)$. Scheduling it then gives $C_i = \max(C_{i-1}, r_i) + t_i$. Then C_n is the optimal makespan we're looking for, and we can calculate it in linear time.

The overall algorithm will work by maintaining the sorted list of jobs after every update, and running the linear-time algorithm to find the makespan. Depending on how we maintain the sorted list, the complexity is $O(Q^2)$ or $O(Q^2 \log Q)$.

Subtask 2

In the full solution, we used a static segment tree. But for the second subtask, there's an algorithm that uses a different data structure. For this subtask we may assume that jobs are only added and never removed.

Using the framing from subtask 1, let's say that job i is "clumped" with job $i-1$ if, in our schedule, job i starts at time C_i. That is, $r_i \leq C_i$. By grouping together the jobs that are clumped, we get a partition of the jobs into "clumps".

Because jobs are never removed, each C_i will only increase as jobs are added. So once two jobs are in the same clump, they'll never be apart.

Let's maintain an efficient data structure to represent the clumps. We only store two properties of a clump: the start time of its earliest job, and the sum of all its t—which we'll call its length. We'll use a standard-library map data structure (backed by a balanced binary search tree): for every clump we'll store a mapping from start-time to length. When adding a job (r, t), we find the nearest clump starting at or before r. Either the job will merge into the clump, or we'll create a new clump. Then we need to check whether the newly created or newly extended clump overlaps with the next clump. If so, we need to merge and repeat until all clumps are separate. At every step, the completion time of the final clump will be the makespan of the optimal schedule. The overall time complexity is $O(Q \log Q)$.

Alternative Solution

Recall that it's always optimal to do jobs in the order that they're released, and that this greedy algorithm can be directly simulated for a solution to subtask 1.

That begs the question: can we build a better simulation to solve the whole problem?

In fact, we can!

Let's first formalize our task a little bit – we can think of our greedy algorithm as starting with a (sorted) list of jobs along with a current greedy state (the initial completion time of 0). We then iterate over the list of jobs, and for each job i, we update the greedy state based on the r_i and t_i of that job. We will also formalize the update as a function $f_i(x)$ that takes in the current greedy state x and outputs the new greedy state after completing job i (let's call f_i the job-function of job i). Finally, our answer is the last greedy state.

Now, let's write out the formula for $f_i(x)$:

Recall the formula for the completion time of job i:

$$C_i = \max(C_{i-1}, r_i) + t_i$$

La version française figure à la suite de la version anglaise.

where C_i and C_{i-1} are the completion times of job i and $i-1$ respectively. It follows that f_i is defined as follows:

$$f_i(x) = \max(x, r_i) + t_i$$

If we zoom out a bit, we can see that the entire simulation can be captured as the following formula:

$$f_n(f_{n-1}(\cdots f_1(0)\cdots)) = (f_n \circ f_{n-1} \circ \cdots \circ f_1)(0)$$

Thus, it follows that if we can efficiently maintain the composition of $f_1, f_2, \ldots, f_n$ as jobs are added/removed, we can solve the problem completely.

As the updates are incremental, this motivates us to use some kind of tree data structure as opposed to some approach that quickly recomputes the answer from scratch each query. This is because trees are efficient at maintaining data across incremental updates, and can also efficiently answer aggregate queries about their data (which is our goal!).

However, how do trees answer aggregate queries? They do so by storing aggregate information in the nodes themselves – each node stores the aggregate information of its children.

For example, if we wanted to answer aggregate queries about the sum of all elements in a binary search tree, for each node we must store the sum of elements in its subtree. An example diagram is below:

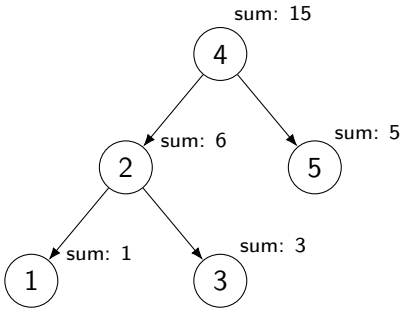


Figure 1: A binary search tree with 5 keys. Each node also stores auxiliary information about the sum of all keys in its subtree.

Thus, if we were to store all of our jobs/job-functions in a tree, we would need to find a general representation for compositions of job-functions. This is because the aggregate information of a node is the composition of all job-functions in its subtree, equivalent to scheduling all of the jobs in its subtree in order of release time.

Now, let's look for a general form for compositions of job-functions.

Recall that job-functions are defined as the form $f_i(x) = \max(x, r_i) + t_i$. We can rearrange this to $f_i(x) = \max(x + t_i, r_i + t_i)$ (this step is optional, but it makes some math later on easier). Then, if we replace our constant values with generic ones (e.g. A and B), we get $f(x) = \max(A+x, B)$ (with $A = t_i$ and $B = r_i + t_i$). From now, we'll call this the *general form* of a job-function.

You may notice that this looks very similar to the general form of linear functions ($f(x) = Ax + B$), except with $+$ changed with max and $\times$ changed with $+$. Moreover, we already know that the compositions of

linear functions is still linear. Could we take a similar approach here and show that the composition of job-functions f_i and f_j is a job-function as well?

Let's try it!

Consider two job-functions $f_i(x) = \max(A + x, B)$ and $f_j(x) = \max(C + x, D)$. Their composition is:

$$\begin{aligned}(f_j \circ f_i)(x) &= f_j(f_i(x)) \\ &= f_j(\max(A + x, B)) \\ &= \max(C + \max(A + x, B), D) \\ &= \max(\max(C + A + x, C + B), D) \\ &= \max((C + A) + x, \max(C + B, D))\end{aligned}$$

Indeed, our hypothesis is correct: the composition of f_i and f_j results in another job-function f', this time with the constants $A' = C + A$ and $B' = \max(C + B, D)$.

Thus, we can proceed with our tree solution!

Note: This technique can be generalized to a wider range of problems. For a more technical explanation on this, see the *Additional Information* section.

Now, the only step that remains is to discuss how we handle updates on our tree data structure. One natural approach for this is to use a binary search tree (BST) – we can insert jobs into the tree based on their release times, and then update aggregate information as we perform our updates. However, this approach is not recommended, as BSTs are difficult to implement in practice due to the need for a balancing algorithm[1].

Moreover, in general, any kind of data structure that maintains a dynamic array/set[2] will have this issue. Thus, we should instead focus on data structures for static arrays[3].

To that end, segment trees come to mind as a good candidate, as they let us efficiently answer aggregate queries over static arrays and mutate elements in the array.

Note: For an introduction on segment trees, see the previous solution.

However, instead of storing jobs directly into the segment tree, we will instead build a segment tree over the release times of the jobs. Then, insertion/deletion of a job becomes a modification to the index corresponding to its release time. At any point, we can also query the total completion time of all jobs by looking at the aggregate information stored in the root node.

Finally, there are two important implementation details to consider: how we handle multiple jobs with the same release time, and how we handle the case when there are no jobs with a particular release time.

The overall time complexity of this approach is $O(Q \log s_{max} + s_{max})$, where s_{max} is the maximum starting

[1] e.g. AVL trees, Red-Black trees, Treaps

[2] In this case, we use dynamic to refer to the fact that the array/set can be resized after it's created, as opposed to just being able to modify elements at all.

[3] Similarly, we use static to refer to the fact that the array cannot change size after it's created. Elements can still be modified however.

time of any job (for this problem, the bound is 10^6), and the memory complexity is $O(s_{max})$.

Additional Information

Our representation of compositions of job-functions as $f(x) = \max(A + x, B)$ can actually be generalized to include a wider range of "linear" functions. Let's look at a generalization that's useful in a variety of data structure problems.

We start with our general job-function definition:

$$f(x) = \max(A + x, B)$$

Let's first introduce a new variable for the right term. This will seem unintuitive at first, but it will make sense later on:

$$\Rightarrow f(x) = \max(A + x, B + 0)$$
$$\Rightarrow f(\langle x, y\rangle) = \max(A + x, B + y), \text{rewriting 0 with a new variable } y$$

Next, for our general class of functions to be composable with itself, we need to ensure that the input and output dimensions are the same. Thus, let's add our new variable y to the output as well, turning it into a vector (since we're trying to *generalize* our definition of job-function, we'll also give our new output variable the same flexibility as we did with the original output variable). This gives us:

$$\Rightarrow f(\langle x, y\rangle) = \begin{bmatrix}\max(A + x, B + y)\\ \max(C + x, D + y)\end{bmatrix}$$

With some renaming of variables, we get the following final general form:

$$\Rightarrow f\left(\begin{bmatrix}v_1\\ v_2\end{bmatrix}\right) = \begin{bmatrix}\max(m_{1,1} + v_1, m_{1,2} + v_2)\\ \max(m_{2,1} + v_1, m_{2,2} + v_2)\end{bmatrix}$$

If you're now thinking to yourself "wow, this really looks like a matrix", you would be correct! This is in fact a variation of matrix-vector multiplication for 2×2 matrices. Let's also look at the standard version to compare:

$$\Rightarrow f\left(\begin{bmatrix}v_1\\ v_2\end{bmatrix}\right) = \begin{bmatrix}m_{1,1}v_1 + m_{1,2}v_2\\ m_{2,1}v_1 + m_{2,2}v_2\end{bmatrix}, \text{ for the matrix } \begin{bmatrix}m_{1,1} & m_{1,2}\\ m_{2,1} & m_{2,2}\end{bmatrix}$$

Normally, when we first learn about matrices, we think of the elements as being real numbers, and our operations over the elements being addition and multiplication. However, this is not a strict requirement – we can actually define matrices over any *Field* (a set of elements with an addition and multiplication operation that satisfy certain properties)[4]. By default, we use the Field of real numbers, with the standard operations of addition and multiplication respectively. However, this is not the only option.

Moreover, if we're only interested in a subset of all matrix operations, we can further loosen this defintion. In our case, we would like to define matrices over a *Semiring*. A Semiring is similar to a Field, but without the requirement that addition and multiplication be invertible.

Our Semiring will use the integers as the set of elements, and have addition be the max operation, and multiplication be standard addition. The matrices that we define over this Semiring have well-defined

[4]In particular, these properties are that both operations are associative, commutative, distributive, distribute over each other, have inverses, and have identities.

matrix-matrix and matrix-vector multiplication operations, and also come with important properties such as associativity of matrix-matrix multiplication.

For a final example, let's also look at what 2×2 matrix multiplication looks like under this Semiring. Consider two matrices A and B:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix}, B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

$$AB = \begin{bmatrix} \max(A_{1,1} + B_{1,1}, A_{1,2} + B_{2,1}) & \max(A_{1,1} + B_{1,2}, A_{1,2} + B_{2,2}) \\ \max(A_{2,1} + B_{1,1}, A_{2,2} + B_{2,1}) & \max(A_{2,1} + B_{1,2}, A_{2,2} + B_{2,2}) \end{bmatrix}$$

This trick is known as "matrix multiplication over the Tropical Semiring", and these matrices can be used to represent a wide range of functions that involve maximizing/minimizing values, including many Dynamic Programming problems.

Good luck using them on your adventures :D

Finally, let's zoom back down and look at how we can apply this technique to our problem.

Consider our job-function definition $f(x) = \max(A + x, B)$. We only have one variable, so you'd think that we would only need a 1×1 matrix (which would be functionally equivalent to a single number). However, our matrices cannot directly represent constants, so this would not work.

Instead, a common workaround is to add an extra *dummy* dimension to our input vector. This dimension will always be 0, and will allow us to encode constants in our matrices via the multiplication operation in our Semiring (which is standard addition).

Thus, our matrix will be of size 2×2, and will look like the following:

$$f(x) = \max(A + x, B) \iff \begin{bmatrix} A & B \\ -\infty & 0 \end{bmatrix}$$

Let's also check this by evaluating the matrix against a vector:

$$\begin{aligned} \begin{bmatrix} A & B \\ -\infty & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} &= \begin{bmatrix} \max(A + x, B + y) \\ \max(-\infty + x, 0 + y) \end{bmatrix} \\ &= \begin{bmatrix} \max(A + x, B + y) \\ y \end{bmatrix} \\ &= \begin{bmatrix} \max(A + x, B) \\ y \end{bmatrix} \text{ if } y = 0 \end{aligned}$$

Notice that y does not change after the multiplication, so as long as we make sure it's 0 initially, we can be confident in our calculations.

Encoding job-functions this way is actually how the alternative solution was implemented. We represent each job-function as a 2×2 matrix, and then multiply them to compose the functions together. Note this also requires that multiplication is associative for our segment tree to be correctly computing aggregate information, but this is already guaranteed by the properties of matrices.

La version française figure à la suite de la version anglaise.