Task 1: Mountains (`mountains`)

Authored and prepared by: Lim An Jun

Subtask 1

Limits: H_i is non-decreasing.

Since H_i is non-decreasing, there does not exist mountains y and z such that $y < z$ and $H_y > H_z$. Thus, there are no valid triplets and we output 0.

Subtask 2

Limits: $0 \leq H_i \leq 1$

All heights are 0 or 1, so $H_x = H_z = 0$ and $H_y = 1$. Let $smallLeft_i$ be the number of '0's in $H_1, H_2, ..., H_{i-1}$ and $smallRight_i$ be the number of '0's in $H_{i+1}, H_{i+2}, ..., H_n$. Calculate and store their values in seperate arrays in O(n).

Consider a candidate for center mountain y where $H_y = 1$. There are $smallLeft_y$ candidates for x and $smallRight_y$ candidates for z. The number of choices is $smallLeft_y \times smallRight_y$. Calculate the sum of possible choices over all candidates y and output it.

Time complexity: O(n)

Subtask 3

Limits: $0 \leq H_i \leq 99$

Consider each possible height h for the center mountain y where $0 \leq h \leq 99$. Note that the magnitude of heights do not matter as we only care if each mountain is shorter than h or not. Treat all $H_i < h$ as 0 and all $H_i \geq h$ as 1. Then, the same approach to subtask 1 can be used.

Time complexity: O($\max H_i \times n$)

Subtask 4

Limits: $0 \leq n \leq 500$

Iterate through all possible triplets of x, y, z such that $1 \leq x < y < z \leq n$ and count how many are valid, which takes O(1) for each triplet.

Time complexity: O(n^3)

Subtask 5

Limits: $0 \leq n \leq 10^4$

If we copy the solution from subtask 3 directly, it will fail as $\max H_i = 10^{18}$. However, we only need to check $h = H_i$ for some $1 \leq i \leq n$. There are at most n such different values, and thus a modified solution to subtask 3 will pass.

Time complexity: O(n^2)

Subtask 6

Limits: $0 \leq H_i \leq 10^5$

So far, we have been re-counting $smallLeft_i$ and $smallRight_i$ naively for different heights. We can use a data structure that supports range queries and point updates with logarithmic complexity (e.g. fenwick tree, segment tree) to calculate them for all heights. Let query(h) denote the number of heights that are at most h so far, and update(h) adds a height h.

Iterate through H from left to right. At each index i, $smallLeft_i$ = query($H_i - 1$). Then, we add in the current height with update(H_i). $smallRight$ can be calculated similarly. The answer is the sum of $smallLeft_i \times smallRight_i$.

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

Subtask 7

Limits: No additional constraints

The solution for subtask 6 fails as $\max H_i = 10^{18}$. However, note that only the relative orders of heights matter. Thus, we can discretise the heights into values within 1 to n (set smallest H_i to 1, second smallest to 2 etc.) and use the same approach.

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

Task 2: Visiting Singapore (`visitingsingapore`)

Authored by: Sung Wing Kin, Ken
Prepared by: Ng Yu Peng

Note: To avoid confusion, we will be treating A, B as positive integers throughout this writeup, so when we say a loss of A happiness we really mean $-A$ happiness algebraically.

Subtask 1

Limits: $K = 1, m \leq n \leq 10^3$

Since $K = 1$, the only event is event 1. As $m \leq n$, we can just attend the first m days all of which are event 1, and since we don't skip any events during the stay and we attend all events $T[1], T[2], \ldots, T[m]$, no loss of happiness is incurred and our answer is simply $mV[1]$.

Time Complexity: $O(1)$

Subtask 2

Limits: $K = 1, n < m \leq 10^3$

Again the only event is 1. This time $n < m$ so we can only attend n events. So we will attend the first n days of events and miss $T[n+1], T[n+2], \ldots, T[m]$. Hence the maximum happiness is $nV[1] - (n - m)B - A$.

Time Complexity: $O(1)$

Subtask 3

Limits: $A = B = 0$

For this subtask we will need to do some dynamic programming. We first treat each attended event as a pair of numbers: if we attend event $T[j]$ on day i, we denote it by the pair (i, j). Let $dp(i, j)$ be the maximum happiness if the last event (x, y) we attend satisfies $x \leq i$ and $y \leq j$.

Since $A = B = 0$ we do not need to worry about incurring losses to happiness. If $S[i] = T[j]$ then we may choose to attend the event (i, j). Otherwise we cannot attend (i, j). Hence

$$dp(i,j) = \begin{cases} \max(dp(i-1,j-1) + V[T[j]], dp(i,j-1), dp(i-1,j)) & \text{if } S[i] = T[j] \\ \max(dp(i,j-1), dp(i-1,j)) & \text{if } S[i] \neq T[j] \end{cases}$$

Then our final answer is just $dp(n, m)$.

Time Complexity: $O(nm)$

Subtask 4

Limits: $A = 0$

We will still use dynamic programmming, but now we need to change up the state. Redefine $dp(i, j)$ to represent the maximum happiness achievable if the last event (x, y) attended has $x \leq i$ and $y \leq j$, including the loss of happiness incurred from skipping events up to $S[i]$ and $T[j]$.

We first set $dp(i, j) = \max(dp(i, j-1) - B, dp(i-1, j) - B)$, to take into account the cases where we do not attend event (i, j). In each case we additionally skip $T[j]$ and $S[i]$ respectively so we incur an additional loss of B.

Then if $S[i] = T[j]$, we have the option of attending the event (i, j) now. We can also have (i, j) be the first event we attend. Hence, if $f(i, j)$ is the maximum happiness achievable if the last event attended is (i, j), ignoring costs from skipping events after $S[i]$ and $T[j]$, then

$$f(i, j) = max(dp[i-1][j-1] + V[T[j]], V[T[j]] - (j-1)B)$$

Now we set if $f(i, j) > dp(i, j)$ then we set $dp(i, j) = f(i, j)$ as attending event (i, j) gives more happiness.

Now the answer is just the maximum of $f(i, j) - (m-j)B$ across all i, j, with the $-(m-j)B$ to account for the loss of happiness for skipping events after $T[j]$, as well as $-A - mB$ in the case we do not attend any events

Time Complexity: $O(nm)$

Subtask 5

Limits: $B = 0$

Now, if we attend event (i, j), we need to know if we attended $S[i-1]$ and $T[j-1]$ as well to decide if we will incur an additional loss of A happiness. Let $dp(i, j, k)$ be the maximum happiness (not including losses for skipping events after $S[i], T[j]$) if the last event attended (x, y) has $x \leq i$ and $y \leq j$, where $k = 0$ has no restrictions, $k = 1$ means we attended $T[j]$, $k = 2$ means we attended $S[i]$, and $k = 3$ means we attended both $S[i]$ and $T[j]$.

We first set

$$dp(i, j, 0) = \max(dp(i, j-1, 0), dp(i-1, j, 0)$$

$$dp(i, j, 1) = dp(i-1, j, 1)$$

$$dp(i, j, 2) = dp(i, j-1, 2)$$

to note the cases where we do not attend event (i, j).

Now if $S[i] = T[j]$, we need to consider the event (i, j). We first consider the case where (i, j) is our first event. So firstly we set

$$dp(i,j,3) = \begin{cases} V[T[j]] & \text{if } j = 1 \\ V[T[j]] - A & \text{if } j > 1 \end{cases}$$

Next we consider the case where there is a previous event. We incur a loss of A happiness for each of the events $S[i-1], T[j-1]$ we do not attend, so we consider the values $dp(i-1, j-1, 0) - 2*a + V[T[j]], dp(i-1, j-1, 1) - a + V[T[j]], dp(i-1, j-1, 2) - a + V[T[j]], dp(i-1, j-1, 3) + V[T[j]]$. If the maximum among these numbers is larger than the current value of $dp(i, j, 3)$, set $dp(i, j, 3)$ to this maximum. Hence now $dp(i, j, 3)$ is the maximum happiness attainable if we attend event (i, j) last, not considering penalties to happiness from events after $T[j]$.

With this we can now update our other dp values. If $dp(i, j, 3)$ is larger than any of the other $dp(i, j, k)$ for $k = 0, 1, 2$, set $dp(i, j, k) = dp(i, j, 3)$.

However, notice that as i, j can each go up to 5000, our memory usage will be around 400MB if we use int arrays. Hence we will need to optimise memory usage.

Note that to compute $dp(i, j, k)$ we only refer to indices $i - 1, i, j - 1, j$. Hence we can do DP on the fly and just use a $2 \times m \times 4$ array, with the first index representing $i - 1$ and i in some order.

Now the answer is simply the maximum of $-A$ (don't attend any events), or the maximum among $dp(i, j, 3) - A$ for $j < m$ or the maximum among $dp(i, j, 3)$ for $j = m$.

Time Complexity: $O(nm)$

Subtask 6

Limits: $n, m < 100$

Let $f(i, j)$ be the maximum happiness achievable if the last event attended is (i, j), ignoring costs from skipping events after $S[i]$ and $T[j]$, then we can just consider all the possible cases to compute $f(i, j)$. In particular, $f(i, j)$ is the maximum among all of

$$\begin{cases} f(i-1, j-1) + V[T[j]] & \\ f(i-1, j_1) + V[T[j]] - A - (j - j_1 - 1)B & \text{for } j_1 < j - 1 \\ f(i_1, j-1) + V[T[j]] - A - (i - i_1 - 1)B & \text{for } i_1 < i - 1 \\ f(i_1, j_1) + V[T[j]] - A - (i + j - i_1 - j_1 - 2)B & \text{for } j_1 < j - 1 \text{ and } i_1 < i - 1 \\ V[T[j]] & \text{if } j = 1 \\ V[T[j]] - A - (j-1)B & \text{if } j > 1 \end{cases}$$

Now the answer is just the maximum of $f(i, m)$ across all i, $f(i, j) - (m - j)B - A$ for all i, j with $j < m$, as well as $-A - mB$ in the case we do not attend any events.

Time Complexity: $O(n^2m^2)$

Subtask 7

Limits: No additional constraints

To solve the final subtask, we just need to merge our solutions for subtasks 4 and 5.

Everything is the same as in Subtask 5 except for the cases where we do not attend event (i, j) when considering $dp(i, j, k)$. So we first set

$$dp(i, j, 0) = \max(dp(i, j-1, 0) - b, dp(i-1, j, 0) - b$$

$$dp(i, j, 1) = dp(i-1, j, 1) - b$$
$$dp(i, j, 2) = dp(i, j-1, 2) - b$$

to account for the additional loss of B happiness when we skip event $S[i]$ or $T[j]$. Then the rest of the solution follows as in Subtask 5.

Now the answer is simply the maximum of $-A-mB$ (don't attend any events), or the maximum among $dp(i, j, 3) - A - (m-j)B$ for $j < m$ or the maximum among $dp(i, j, 3)$ for $j = m$.

Time Complexity: $O(nm)$

Task 3: Solar Storm (`solarstorm`)

Authored and prepared by: Bernard Teo Zhi Yi

Subtask 1

Limits: $S = 1$, $N \leq 10^4$, $K \leq 10^9$, $d_i \leq 10^5$ for all $1 \leq i \leq N-1$, $v_i \leq 10^5$ for all $1 \leq i \leq N$

There is exactly one shield. For each module, calculate the total value of all shielded modules if we deploy the shield there. This takes $O(N)$ per module, and we have N modules to check.

Time complexity: $O(N^2)$

Note: The constraints of this subtask are such that solutions which store cumulative distances and values using 32-bit signed integers will not overflow.

Subtask 2

Limits: $S = 1$, $d_i = 1$ for all $1 \leq i \leq N-1$

There is exactly one shield, and furthermore the distance between any two adjacent modules is fixed at 1. This means that the optimal solution must protect a contiguous range of $2K + 1$ modules (unless $N \leq 2K$, in which case we can deploy the shield to protect all modules). We can solve this in $O(N)$ using the sliding window technique.

Time complexity: $O(N)$

Subtask 3

Limits: $S = 1$

This is similar to subtask 2, but now the distances between adjacent modules are no longer fixed.

For each module i, we can determine r_i, the rightmost module that will be protected if the shield is deployed at module i. This can be done in $O(N)$ with the sliding window technique. The modules can also be scanned in the reverse order to determine l_i, the leftmost module that will be protected if the shield is deployed at module i.

The optimal location to deploy the shield is then $\arg\max_{i \in \{1,\ldots,N\}} \sum_{j=l_i}^{r_i} v_j$. The value $\sum_{j=l_i}^{r_i} v_j$ may be computed in $O(1)$ after computing the prefix sum of the v_i's.

Time complexity: $O(N)$

Subtask 4

Limits: $K = 1$, $d_i = 2$ for all $1 \leq i \leq N - 1$

This is similar in essence to subtask 2. Each shield can only protect the module that it is deployed in. Any valid solution must hence deploy shields at S consecutive modules. We can check all possibilities in $O(N)$ using the sliding window technique.

Time complexity: $O(N)$

Subtask 5

Limits: $N \leq 10^4$

Given module i, let r_i be the rightmost module that is at most distance K away from module i (this is equivalent to the definition of r_i in subtask 3).

Suppose module i is the leftmost protected module. Then let $e_1(i) - 1$ be the rightmost module that is protected by the same shield that protects module i. $e_1(i) - 1$ can be determined by traversing the spaceship rightwards from module i. Then $e_1(i)$ is the leftmost module that is protected by the second shield (from the left), and we can similarly traverse the spaceship to find $e_2(i) - 1$, the rightmost module that is protected by the second shield. In this way, we can find $e_S(i) - 1$, the rightmost module that is protected by the S^{th} shield. At the same time, we keep track of the cumulative value of protected modules, so that after arriving at module $e_S(i) - 1$ we know the total value of protected modules. Since we traverse from left to right after starting from module i, it takes $O(N)$ to determine $e_S(i) - 1$ given a fixed i.

We try every i as a possible leftmost protected module – there are N modules that we could possibly start with.

Time complexity: $O(N^2)$

Note: Care should be taken to ensure that the implementation does not read past the right bound of the spaceship.

Subtask 6

Limits: $S \leq 10^2$

We can speed up the solution for subtask 5 by observing that we can compute the values of r_i once (using the sliding window technique described in subtask 3) and then cache these values, so that we can compute $e_S(i)$ in $O(S)$ time.

For each i, we can use the prefix sum method introduced in subtask 3 to compute the cumulative sum of values of modules in the range $[i, e_S(i))$.

Time complexity: $O(NS)$

Subtask 7

Limits: No additional constraints

We can build a directed graph of $N + 1$ vertices. Each vertex represents a module, and vertex $N + 1$ represents a dummy module past the right bound of the spaceship. For every vertex i ($1 \leq i \leq N$), add an edge from vertex i to vertex $e_1(i)$. Observe that the graph is a tree that is rooted at vertex $N + 1$ (i.e. all edges are directed towards vertex $N + 1$). Finding $e_S(i)$ is then equivalent to finding the S^{th} ancestor of the vertex i (if vertex i has depth less than S, then we define$e_S(i)$ to be vertex $N + 1$).

We then do a depth-first search on the tree starting from the root, while maintaining an auxiliary stack. When visiting vertex i, push i onto the stack, then visit its children, then pop i from the stack. Hence, when processing vertex i, the stack contains the list of vertices from i to the root (from top to bottom of the stack). The S^{th} element from the top of the stack is thus $e_S(i)$, which may be obtained in $O(1)$ if the stack is stored in a contiguous block of memory. Since the stack operations are $O(1)$, the whole depth-first search takes $O(N)$ time.

Lastly, we use the same prefix sum method as per subtask 3 and 6, which takes $O(1)$ time per range.

Time complexity: $O(N)$

Note: A solution that runs in $O(N \log N)$ is likely to get full points as well. There are a number of alternatives to the depth-first search which run in $O(N \log N)$, such as:

- Path compression on the tree. For each vertex i, we traverse S steps towards the root, and compress the path (i.e. replace the outgoing edge of each vertex j we traverse with an edge that goes from j directly to $e_S(i)$ (tagged with the number of steps the edge represents)). Since the function e_S is monotonic, we may process the vertices in increasing order of i (without fear of "skipping" the vertex that we want).

- Binary lifting on the tree. This is a well-known technique that can find the h^{th} ancestor of any given vertex in $O(\log h)$, after a preprocessing step that takes $O(N \log N)$ time.