Task 1: Voting City (`voting city`)

Authored and prepared by: Aloysius Lim

Subtask 1

Limits $Q = 1, K = 1$, No tickets sold

This is just a standard shortest path problem.

Time Complexity: $O(E \log V)$

Subtask 2

Limits: $K = 1$, No tickets sold.

There are 2 ways to do it. You can run the shortest path problem Q times which will work for this subtask. However, in preparation for the later tasks, the intended way of reverse the problem. Instead of finding the shortest path to the voting city, find the shortest path from the only voting city to each city. You have to reverse the edges however, and do shortest path from the only voting city. Then, you can answer the query in O(1) time.
Time Complexity: $O(E \log V + Q)$ or $O(QE \log V)$

Subtask 3

Limits: $K = 1$, $Q = 1$. No tickets sold.

Similar to the above, running the shortest path Q times would work, and you check each voting city. However, you can continue with the latter approach as well. In this case, there are multiple voting cities, so a common technique is to use a super source city connecting the voting city with road edge of 0.

Time Complexity: $O(E \log V + Q)$ or $O(QE \log V)$

Subtask 4

Limits: $K = 1, Q = 1$

From this sub task onwards, we have to consider the effects of the tickets. We know we can use a ticket at most once, and once per road. Furthermore, notice that the number of possible tickets is small at 5. This motivates keeping a state of the tickets that have been used. Thus, we can make 32 copies of the cities and compute the shortest path to (city, tickets used). We reconnect the edges and modify the weights between the new cities whether or not we use a ticket between them. Then, at the destination, we compare the 32 different possible ticket state and choose the best cost of them. Note that, while it is possible to incorporate the ticket cost when constructing the new edges for this subtask, to facilitate further subtasks, it is better to just leave the cost at the end and then add the costs while computing the best state.
To analysise time complexity, let the number of tickets be T. We thus have $T2^T E$ edges and $2^T V$ vertices.
Time Complexity: $O(T^2 2^T E \log V)$

Subtask 5

Limits: $K = 1$

Notice that in this case, we can no longer just blindly run Q queries. Thus, we have to apply the trick we did in subtask 2.
Time Complexity: $O(T^2 2^T E \log V + Q2^T)$

Subtask 6

Limits: There is only at most 1 ticket.

This is a simplified version of the 5 tickets. However, if the contestant is unable to do the above, he can just code out a simpler state (city, taken) where taken is just a T/F boolean. Note that here, since the number of edges and vertices is small, you can indeed run the algorithm Q times.

Time Complexity: $O(E \log V + Q)$ with a higher constant.

Subtask 7

Limits: $N \leq 100, E \leq 1000$

I added this subtask to allow bad implementation to pass or any not so optimal solutions. I personally do not know of a weaker solution that can pass this solution without passing the final one. However, because the constraints are small, if you pass subtask 4 while running Q times, you should be able to pass this

Time Complexity: $O(T^2 2^T E \log V + Q 2^T)$

Subtask 8

Limits: None

Finally in this subtask, you add the final trick of the supersource node and finally this concludes the question.
Time Complexity: $O(T^2 2^T E \log V + Q 2^T)$

Task 2: Gym Badges (`gymbadges`)

Authored and prepared by: Teow Hua Jun

Subtask 1

Limits $N \leq 10$

Iterate through all permutations of gyms and simulate going to the gyms in order and challenge them if possible. The maximum gym badges you can obtain among all permutations will be the answer.

Time Complexity: $O(N!)$

Subtask 2

Limits: L is Constant

Since L is constant the only difference between gyms is the level gained. Sort the gym by L_i and greedily choose the smallest X_i until sum of X chosen is greater than L.

Time Complexity: $O(N \log N)$

Subtask 3

$N \leq 5000$

Limits: $N \leq 5000$

Note that there exist an optimal solution, $a_1, a_2, a_3, ..., a_k$ where a_i is the i^{th} gym challenged, such that $X_{a_i} + L_{a_i} \leq X_{a_{i+1}} + L_{a_{i+1}}, \forall i < k$.

Proof: Assume exist optimal solution, $a_1, a_2, a_3, ..., a_k$ where exist adjacent gyms such that $X_{a_i} + L_{a_i} > X_{a_{i+1}} + L_{a_{i+1}}$.

$$L_{cur} = \sum_{n=1}^{i-1} X_{a_i}$$

$$L_{cur} + X_{a_i} \leq L_{a_{i+1}}$$
$$L_{cur} + X_{a_i} + X_{a_{i+1}} \leq L_{a_{i+1}} + X_{a_{i+1}} < L_{a_i} + X_{a_i}$$
$$\therefore L_{cur} + X_{a_{i+1}} < L_{a_i}$$

Thus, we can challenge gym a_{i+1} before a_i, swapping the order of these two adjacent gyms, note that gyms after the a_{i+1} gym are not affected as the gyms a_i and a_{i+1} still make Wabbit gain $X_{a_i} + X_{a_{i+1}}$ levels.

Therefore, from any optimal solution we can keep swapping adjacent gyms to make the solution sorted with respect to $X_i + L_i$

Using this observation, we can sort the gyms and process them in order.

Let $dp(a, b) =$ Minimum level Wabbit will be at after challenging a gyms after processing b gyms.

We can form a simple transition of

$$dp(a,b) = \begin{cases} \min(dp(a-1,b-1) + X_a, dp(a-1,b)), & \text{if } dp(a-1,b-1) \leq L_a \\ dp(a-1,b), & \text{otherwise} \end{cases}$$

State: $O(N^2)$ Transition: $O(1)$

Time Complexity: $O(N^2)$

Subtask 4

Limits: No further constraints

Using the observation made in subtask 3, we keep the gyms sorted by $X_i + L_i$

Let $S_x = \{a_1, a_2, ..., a_k\}$ be an optimal set of gyms challenged to obtain the maximum number of badges with the minimum level gain from the first x gyms and $G_x = \sum_{i=1}^{|S_x|} X_{a_i}$, the total level gain after challenging the gyms in S_x.

Lemma 1: If $L_{n+1} < G_n$ and exist $y \in S_n$ such that $X_y > X_{n+1}$, then we can replace gym y with gym $n + 1$.

Let gym k be the last gym in S_n. Note that k can be the same as y

$$\begin{aligned} G_n - X_k \leq L_k \implies G_n &\leq X_k + L_k \\ &\leq X_{n+1} + L_{n+1} \\ &< X_y + L_{n+1} \end{aligned}$$

$$\therefore G_n - X_y < L_{n+1}$$

Thus, we can swap gym y with gym $(n+1)$

Solution:

Assume we have the optimal solution for the first n^{th} gyms with the maximum number of badges with minimum level gain and currently processing our $(n+1)^{th}$ gym.

If $L_{n+1} \geq G_n$, then $S_{n+1} = S_n \cup \{n+1\}$:

Namely, we add gym $n+1$ to the current solution set of gyms.

This can be proven by contradiction as if exist another set of gyms with greater number of badges or less level gain. Then there will exist a set of gyms in the first n gyms that have greater number of badges or less level gain than S_n

Let the gym will the maximum level gain in S_n be gym y

Else if $L_{n+1} \geq G_n$ and $X_y > X_{n+1}$, then $S_{n+1} = S_n \setminus \{y\} \cup \{n+1\}$:

Namely, we swap gym y for gym $(n+1)$ to have a lower level gain.

We can prove this is optimal as we cannot have $|S_n| + 1$ gym badges as $L_{n+1} \geq G_n$ and the level gain is minimised as otherwise S_n will have a smaller level gain. From lemma 1 we can also see that this swap is possible.

Else $S_{n+1} = S_n$:

We can prove this is optimal using similar logic as the above case.

Hence, as we have shown how to construct the optimal solution of $n+1$ gyms from n, we can inductively generate the answer. The actual implementation can be done using a priority queue or set to maintain the set of gyms.

Time Complexity: $O(N \log N)$

Alternative solution

The alternative solution is based on the following observations:

Observation 1: suppose gym i has the smallest value of X_i among all gyms. Then there exists an optimal solution which challenges gym i at some point in time.

Proof of observation 1: Consider the first gym challenged in any optimal solution which does not use gym i. Change the first gym to gym i.

Observation 2: If we know that there exists an optimal solution which challenges gym i, then we can calculate the answer by doing the following:

- Remove gym i
- For all gyms $j \neq i$, if $X_j + L_j > X_i + L_i$, replace L_j by $L_j - X_i$ (if L_j becomes negative due to this operation, we can delete gym j)
- Calculate the answer to this new problem, and increase the answer by 1

Proof of observation 2: suppose we have a sequence of gym challenges that uses gym i. Now consider what happens when we delete gym i from the sequence. For every gym j challenged after i, the entering level when gym j is challenged is decreased by X_i.

Similarly, if we have a solution to the new problem, we can attempt to insert gym j at the latest possible opportunity, and observe that gyms which satisfy $X_j + L_j > X_i + L_i$ are affected.

These two observations immediately give us an $O(N^2)$ solution. We can speedup this to $O(N \log N)$ by using lazy propagation segment tree.

Task 3: Towers (`towers`)

Authored and prepared by: Ng Yu Peng

Terminology

Throughout this writeup, points only refer to points where cities are located, and we will refer to building towers in cities as picking points. A **column** refers to points with the same x-coordinate and a **row** refers to points with the same y-coordinate.

Subtask 1

Limits $N \leq 3$

Check if all points are in the same row or column. If they aren't, pick all the points. If they are all in the same row, pick the leftmost and rightmost points. If they are all in the same column, pick the topmost and bottommost points.

Time Complexity: $O(1)$

Subtask 2

Limitsl: $N \leq 16$

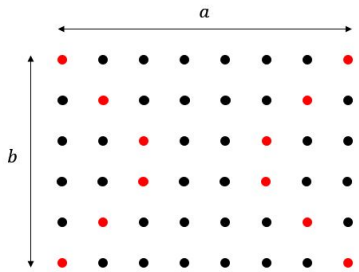
Use a bitmask to simulate all 2^N ways of building towers, and for each of them, iterate through the points to store the max/min x-coordinates in each row, and the max/min y-coordinates in each column, as well as the number of points in each row/column. Now iterate through all the points again and check the conditions are satisfied.

Time Complexity: $O(N2^N)$

Subtask 3

Limits: $N = ab$ for some positive integers a, b and $(X_{ai+j}, Y_{ai+j}) = (i+1, j)$ for all integers i, j with $0 \leq i \leq b-1, 1 \leq j \leq a$

Basically the points form a rectangular lattice, just consider the cases $a > b$ and $a \leq b$. In the case $a > b$, choose the points (red points are chosen) like so:



The case $a \leq b$ is similar.

Time Complexity: $O(N)$

Subtask 4

Limits: For every integer a, there are at most two cities whose x-coordinate is a

Pick the leftmost and rightmost cities in each row/ Since each column has at most 2 points the conditions are satisfied,

Either sort the sets of points in each row to find the leftmost and rightmost, or keep track of the leftmost and rightmost x-coordinates in each row while iterating through the points.

Time Complexity: $O(N \log N)$ or $O(N)$

Subtask 5

Limits: $N \leq 5000$

We now describe a general algorithm to choose where to build the towers. Let's extend the idea in the previous subtask: first pick the leftmost and rightmost points in each row.

Since there may be more than 2 points in a column, some columns may have more than 2 points picked. Thus we will keep making changes to the set of points picked, while making sure each row still has at most 2 points picked and each point is either picked or lies between two picked points, until no column has more than 2 points picked.

Consider any column with more than 2 points picked, then look at a picked point in that column which is not the topmost or bottommost picked point. If it is the only point picked in its row, delete it from the set of picked points. If it is the rightmost picked point in its row, delete it from the set and add the point immediately to its left in the same row. If it is the leftmost picked point in its row, delete it from the set and add the point immediately to its right in the same row.

Clearly, any point in the column of the removed point will still be between two points in the same row or column, and since we effectively just replaced a point with the point to its immediate left/right, all points in its row will still be between two points in the same row/column, and obviously each row will still have at most 2 points picked. The process terminates when there is no column with more than 2 picked points, and when this happens all conditions are simultaneously satisfied.

Let the distance between the 0/1/2 picked points in each row be the number of points between them including themselves. Each time we update the set of picked points, the distance decreases by 1 in the row we change, so the sum of distances in all rows decreases by 1. Since the sum of distances at the start is N, the process terminates in at most N changes to the picked points.

This subtask is catered to let solutions using $O(N)$ per change pass.

Time Complexity: $O(N^2)$

Subtask 6

Limits: $N \leq 100000$

You can store points as pairs (row, index in row) to find the point to its immediate left/right in the same row quickly. Use a set to store all picked points in each column in this way, then keep changing the picked points as long as there is some set with size more than 2.

Time Complexity: $O(N \log N)$ with large constant.

Subtask 7

Limits: $N \leq 1000000$

Since the topmost/bottommost picked point in each column can only get higher/lower every time we update the set, we can just store these two points for each column, and every other point goes into a waiting list for removal.

Time Complexity: $O(N \log N)$

Task 4: Grapevine (`Grapevine`)

Authored by: Teow Hua Jun, Jeffrey Lee

Prepared by: Jeffrey Lee, Leong Eu-Shaun

Introduction

Taking joints as vertices and branches as edges, the Grapevine takes the form of a weighted undirected tree graph. We will denote the distance between two vertices i and j as $d_{i,j}$.

Subtask 1

Limits: $N, Q \leq 2000$

Store the tree in adjacency-list format. We can maintain the tree by simply marking/unmarking vertices and updating edges for **soak** and **anneal** actions respectively. **Seek** queries can then be answered by running a depth-first search over the entire tree, for a time of $O(N)$ per query.

Time complexity: $O(NQ)$

Subtask 2

Limits: For all **seek** actions, $q_i = 1$

For this subtask, we root the tree at vertex 1. Starting from vertex 1, run a depth-first search to construct an arbitrary Euler Tour representation sequence of the tree, taking only the first occurence of each vertex such that every vertex appears exactly once. Note in particular that when any one vertex is picked, the subtree consisting of itself and all its descendants forms a contiguous subsequence in this Euler sequence. We can hence maintain an auxillary array S of the same length, such that wherever the i^{th} element of the Euler sequence is v_i, the i^{th} value of the array S is:

$$S_i = \begin{cases} d_{1,v_i}, & \text{if vertex } v_i \text{ has a grape} \\ d_{1,v_i} + 10^{15}, & \text{if vertex } v_i \text{ has no grapes} \end{cases}$$

With this array, **soak** actions become a point update to S at the target vertex, while **anneal**

actions become a range add/subtract to the subtree of the target edge's lower vertex. **Seek** queries are then answered by finding the smallest element of the array S, i.e. the range minimum over all of S. We can perform all three types of query in $O(\log N)$ each using a lazy-propagation segment tree on S.

Time complexity: $O((N+Q)\log N)$

Subtask 3

Limits: The vine forms a complete binary tree, $A_i = \lfloor \frac{i+1}{2} \rfloor$, $B_i = i+1$

For this subtask, we root the tree at vertex 1. The tree has a depth of $O(\log N)$, while each vertex has up to 2 children.

At each vertex, we initially store the shortest distance from that vertex to any of its marked (grape) descendants. We find that these stored values can be correctly maintained across any **soak** and **anneal** queries by starting at the target vertex, updating its stored value according to those of its immediate children, and repeating for its parent until all ancestors have also been updated.

We can then evaluate **seek** queries by starting from the query vertex q_i and trying the stored values of all of its ancestors, taking the minimum out of these trials. It is guaranteed that the shortest distance to a marked vertex will be produced this way: The shortest path between any two vertices in this graph consists of an ascending path from one vertex to their lowest common ancestor, followed by a descending path to the other vertex. Thus, each ancestor p_i covers the shortest paths from q_i to it entire subtree except in the direction of q_i itself, which is instead covered by p_i's child in that direction.

Each query traverses $O(\log N)$ ancestors in $O(1)$ time for a complexity of $O(\log N)$ each.

Time complexity: $O((N+Q)\log N)$

Subtask 4

Limits: There is at most 1 grape on the vine at any point in time.

Root the tree arbitrarily and construct an Euler Tour sequence as in Subtask 2. By creating an auxillary array with $S_i = d_{\text{root},v_i}$, we can handle **anneal** queries and also retrieve $d_{\text{root},v}$ for any one vertex v in $O(\log N)$ each.

The answer to a **seek** query is the length of the direct path between the query vertex q_i and the

single marked vertex m. As described in Subtask 3, this path travels from q_i towards the root until it reaches the lowest common ancestor of q_i and m, where it then proceeds away from the root and to m. The distance between q_i and m can thus be expressed as:

$$d_{q_i,m} = d_{q_i,\text{lca}(q_i,m)} + d_{\text{lca}(q_i,m),m} = d_{\text{root},q_i} + d_{\text{root},m} - 2d_{\text{root},\text{lca}(q_i,m)}$$

We can find the lowest common ancestor of q_i and m in $O(\log N)$ via binary lifting, allowing us to evaluate **seek** queries using the above formula to yield a total $O(\log N)$ per query.

Time complexity: $O((N + Q) \log N)$

Subtask 5

Limits: All **soak** actions will occur before any **seek** or **anneal** actions. For all **anneal** actions, $w_i = 0$.

Prepare and maintain an Euler Tour sequence + binary lifting structure similarly to the previous subtask, in order to find the distance between arbitrary pairs of vertices quickly.

Construct a centroid decomposition on the tree, initially storing at each vertex the shortest distance from the vertex itself to any marked vertex in its covered subtree. We seek to keep these stored values updated across **anneal** and **soak** operations.

Suppose an **anneal** query is performed on an edge connecting vertices $a_i \longleftrightarrow b_i$, reducing the distance between them to 0. Without loss of generality, let vertex b_i be deeper in the centroid-hierachy tree than a_i. It follows that b_i must be a descendant of a_i in the hierachy tree; vertex a_i's covered subtree is bounded only at leaves or by its ancestor centroids, and thus contains b_i. Further, vertex a_i is the lowest-order centroid whose covered tree contains the edge $a_i \longleftrightarrow b_i$, and whose stored value may be affected by the **anneal** operation.

There are then two possibilities for the stored value in a_i after the **anneal**: either the closest marked vertex in a_i's covered subtree is now on a_i's side of the edge $a_i \longleftrightarrow b_i$, or is instead on b_i's side. In the former case, the closest marked vertex to a_i is the same as before the **anneal**, and no update is necessary to a_i's stored value.

It is the latter which needs to be evaluated to cover both cases. This is equivalent to finding the closest marked vertex to b_i within a_i's subtree, which can in turn be retrieved by using the stored values of every centroid on the hierachy-tree path from b_i to a_i. Remember in particular that the covered subtree of b_i extends outwards from the edge, terminating only at leaves and its centroid ancestors - which in turn cover more of a_i's subtree radiating away from the edge til their own ancestors. There are $O(\log N)$ vertices in the centroid tree path from b_i to a_i,

each evaluated in $O(\log N)$ time from using the Euler Tour sequence's distances, for a total complexity of $O(\log^2 N)$ to update the lowest affected centroid a_i.

The remaining centroids on the path from a_i to the hierachal root can be updated using the same process - higher centroids on a_i's side of the edge are to retrieve stored values from their descendants on b_i's side, while higher centroids on b_i's side will retrieve from descendants on a_i's side starting from a_i itself. However, centroids after a_i can be updated in $O(\log N)$ each by keeping a running prefix minimum for each side, such that when iterating upwards from a_i each centroid need only apply its own stored value to the prefix in order to be accounted for by all its ancestors.

Soak and **seek** operations are classic on a centroid decomposition with these stored values, and can also be done in $O(\log^2 N)$ each.

Time complexity: $O(N \log N + Q \log^2 N)$

Subtask 6

Construct a centroid decomposition on the tree. At every centroid, we will store an Euler Tour sequence over the centroid's covering subtree, using the auxillary array value in Subtask 2. **Soak** queries can then be applied to the target centroid and its ancestors by performing the point update to each of their Euler Tour arrays; while **anneal** queries are applied by starting from the higher of the edge's incident centroids, and performing the range add/subtract on its and its ancestors' Euler Tour arrays. These take $O(\log N)$ per centroid, and $O(\log^2 N)$ in total.

The closest marked vertex in any one centroid's subtree can then be obtained in $O(\log N)$ via the range minimum, over which **seeks** can be evaluated in classic pattern in $O(\log^2 N)$.

Time complexity: $O(N \log N + Q \log^2 N)$

Task 5: Fruits (`towers`)

Authored by: Benson Lin Zhan Li

Prepared by: Benson Lin Zhan Li and Marc Phua

Terminology

We say that a fruit is fixed if it is already placed, and free otherwise. Similarly, a section is fixed if a fruit is assigned to it, and free otherwise.

Subtask 1

Limits: $N \leq 8$

Since N is very small, we can afford to test every single possible permutation of fruits.

For each of the $N!$ possible permutations, we first check that the fixed fruits are in the correct sections. Then we can compute, for each prefix of sections, what the cost incurred is.

Time Complexity: $O(N \cdot N!)$

Subtask 2

Limits: $A_j = -1$ for all $1 \leq j \leq n$

In this subtask, we are free to place the fruits in any permutation that we want. If Benson only takes fruits from the first k sections, we should ensure that he takes the k most expensive fruits.

This is possible by placing fruit $n - k + 1$ at section 1, fruit $n - k + 2$ at section 2 and so on, with fruit n at section k. This gives us a cost of $C_{n-k+1} + C_{n-k+2} + \cdots + C_n$. This can be computed quickly using a suffix sum on C_i.

Time Complexity: $O(N)$

Subtask 3

Limits: $N \leq 200$

Let's take some optimal solution for a certain prefix k and consider what happens when we place the some fruits. The fruits that we have yet to place can be divided into 2 categories, those worse than the current best and those better than the current best (best meaning highest tastiness).

We realise that the exact fruits that are worse than the current best don't matter anymore, since at this point all of them are simply filler fruits. Thus, we can represent the current state using the number of sections we have filled so far and the value of the best fruit.

Thus, we let $dp[x][v]$ be the maximum cost using the first x sections such that the best fruit used is fruit v. Invalid states are set to $-\infty$ and $dp[0][0] = 0$. There are 2 transitions:

- $A_x \neq -1$:

  In this case, the fruit at this section is fixed. Thus, the best fruit up to this point is no worse than A_x and we should only consider $v \geq A_x$.

  If $v = A_x$, then the previous best fruit must be some fruit $w < v$, so $dp[x][v] = C_v + \max(dp[x-1][w])$ across all $w < v$

  If $v > A_x$, then the previous best fruit must be fruit v, so $dp[x][v] = dp[x-1][v]$

- $A_x = -1$:

  In this case, the fruit at this section is not fixed. If the current fruit is the best fruit, then the total cost is $C_v + dp[x-1][w]$ where w is the previous best fruit. Otherwise the best fruit had already been placed, so the number of fruits taken is $dp[x-1][v]$.

  Thus $dp[x][v]$ is the best of $dp[x-1][v]$ and $C_v + dp[x-1][w]$ across all $w < v$.

There are $O(n^2)$ states and $O(N)$ transition, which gives us an $O(N^3)$ solution.

Time Complexity: $O(N^3)$

Subtask 4

Limits: $N \leq 2000$

We can speedup the transition from $O(N)$ to $O(1)$ by precomputing the prefix maximums $\max(dp[x-1][w])$ for all w in $O(N)$ time, and thus the solution runs in $O(N^2)$.

Time Complexity: $O(N^2)$

Subtask 5

Limits: $C_i = 1$ for all $1 \leq i \leq n$

Take an example test case of $N = 13, A = \{-1, -1, 5, 6, -1, -1, 7, 11, -1, -1, 10, -1, -1\}$. The dp table is as follows ($dp[x][v]$ is on the xth column from the left and the vth row from the bottom)

1	**2**	2	2	**5**	**6**	6	6	**8**	**9**	9	**9**	**9**
1	**2**	2	2	**5**	**6**	6	6	**8**	**8**	8	**8**	**-**
-	**-**	-	-	**-**	**-**	-	7	**7**	**7**	7	**-**	**-**
-	**-**	-	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**
1	**2**	2	2	**5**	**6**	6	-	**-**	**-**	-	**-**	**-**
1	**2**	2	2	**5**	**5**	5	-	**-**	**-**	-	**-**	**-**
-	**-**	-	-	**-**	**-**	5	-	**-**	**-**	-	**-**	**-**
-	**-**	-	4	**4**	**4**	-	-	**-**	**-**	-	**-**	**-**
-	**-**	3	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**
1	**2**	-	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**
1	**2**	-	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**
1	**2**	-	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**
1	**-**	-	-	**-**	**-**	-	-	**-**	**-**	-	**-**	**-**

If we look at the sections which are free (marked in bold), we see that if $dp[x][v]$ and $dp[x][v-1]$ are valid, then

1. $dp[x][v] - dp[x][v-1] \geq 0$
2. $dp[x][v] - dp[x][v-1] \leq 1$

In fact, we can rigorously prove these observations. The non-decreasing condition can be shown with some manipulation of the transition.

$$\begin{aligned} dp[x][v] &= \max(dp[x-1][v], 1 + \max_{w<v}(dp[x-1][w])) \\ &\geq \max(1 + dp[x-1][v-1], 1 + \max_{w<v-1}(dp[x-1][w])) \\ &\geq \max(dp[x-1][v-1], 1 + \max_{w<v-1}(dp[x-1][w])) \\ &= dp[x][v-1] \end{aligned}$$

To prove the second condition, we first extend it to all columns (not just the sections that are free) and use induction. In the 0th column (i.e. $dp[0]$), $dp[0][0] = 0$ and $dp[0][v] = -\infty$ for all $v \geq 1$ so the condition is true.

For a fixed section x, if A_x is useless (i.e. never used even if we place the smallest possible fruits in front), then $dp[x][v] = dp[x-1][v]$ for all v. Otherwise, $dp[x][v] = dp[x-1][v]$ for all $v > A_x$ and $dp[x][v] = -\infty$ for all $v < A_x$. The condition is true for all $v > A_x$ based on the previous section, and is true for $v = A_x$ trivially because $dp[x][A_x - 1]$ is not a valid state.

For a free section x, notice that the transitions for $dp[x][v]$ and $dp[x][v-1]$ differ in only 2 areas. The transition for $dp[x][v]$ includes $1 + dp[x-1][v-1]$ instead of $dp[x-1][v-1]$ and includes $dp[x-1][v]$. Since $dp[x-1][v] - dp[x-1][v-1] \leq 1$, we have $\max(dp[x-1][v], 1 + dp[x-1][v-1]) - dp[x-1][v-1] \leq 1$. All other parameters in the $\max$ are the same, so $dp[x][v] - dp[x][v-1] \leq 1$.

An immediate consequence of this observation is that the dp states look like ranges of identical values that increment by 1 as we move up the column. Thus, instead of tracking the exact value of each state, we can track the ranges of states that all have the same value.

Now we need to solve the problem of accurately modelling the transitions between columns using these ranges. Consider the partial dp table below.

c+1	c+2	c+3
c+1	c+2	c+2
c+1	c+1	c+2
c	c+1	c+1
c	c	c
-	-	-

We can make 2 observations.

- The first is that a range of values will increment together by moving up and right by 1 (e.g the range of c in the first column moves to the range of $c+1$ in the second column).
- The second is that a new range may need to be added at the lowest valid tastiness.

Thus our solution is as follows:

- Maintain a double-ended queue containing tuples of (section,value,lower bound), each representing one of the ranges.
- Bottom or lower tuples refer to those representing dp states with lower tastiness values, top or higher tuples refer to those representing dp states with higher tastiness values.

- The answer we want at each index is at the top of the deque.
- To process a fixed section, we pop ranges from the bottom of the deque and take a range max, then add a new range (which may combine with another existing range)
- To process a free section, we check if any ranges at the top need to be popped out and check if any new ranges need to be added at the bottom.

Since each range is added and removed once and there are at most O(N) ranges, adding and removal of ranges is amortized O(N). Checking the answer is an O(1) computation since we access the top of the deque, so this is also O(N) overall.

Time Complexity: $O(N)$

Subtask 6

To simplify the remainder of the editorial, we will do a number of preprocessing steps.

We can do a linear pass to eliminate all fixed fruits that will never be picked even if we use the smallest possible free fruits at each section.

We can now rephrase the problem. Instead of having fixed fruits in the sections, we have n' sections that are all free, and the n' free fruits have tastiness from 1 to n' based on their original order. C_i is now equal to the cost of the free fruit with tastiness i.

The fixed fruits are offered to Benson after he passes a specific section, possibly multiple times in a row. Each fixed fruit has a new tastiness, equal to the number of free fruits which originally were less tasty than this fixed fruit.

Let n' be the number of free fruits, and let us renumerate the free fruits to take on indices from 1 to n'. Also, he now picks a fixed fruit if its tastiness is **at least** the maximum in the basket (free fruits continue to follow the strict inequality).

We let $dp[x][v]$ be the maximum cost using the first x sections such that the tastiest fruit has tastiness v **without taking the fixed fruits offered at** x. Bear in mind that this tastiest fruit might not be a free fruit. For simplicity, we can take $dp[0][0] = 0$ and $dp[0][v] = -\infty$ for all $1 \leq v \leq n'$.

If $x > v$, then the state $dp[x][v]$ does not make sense, as using any combination of x free fruits will contain one with tastiness $> v$, so such states are invalid and will not be considered.

We can generalise the observation in Subtask 5 that adjacent $dp[x][v]$ values can only differ by at most 1 with the following claim: **For all** x, v **where** $dp[x][v]$ **and** $dp[x][v-1]$ **are valid states, we have** $0 \leq dp[x][v] - dp[x][v-1] \leq C_v$**.** This can be shown by modifying the proof in Subtask 5 with the additional fact that $C_v \geq C_{v-1}$.

Now consider the optimal configuration for $dp[x][v]$. The immediate consequence of this claim is that if this configuration does not use any fixed fruit between section $x-1$ and section x, then the optimal transition is simply $dp[x][v] = C_v + dp[x-1][v-1]$. These transitions can be chained together, similar to how the ranges in Subtask 5 move together.

Suppose the last fixed fruit that the configuration used was just before section x' with tastiness t, and after that last fixed fruit the total cost was c. We can compute $dp[x][v]$ as

$$C_v + C_{v-1} + \cdots + C_{\max(t+1, v+x'-x+1)} + c$$

by chaining the optimal transitions together.

Letting $p_v = C_v + C_{v-1} + \cdots + C_1$, we get

$$dp[x][v] = p_v - p_{\max(t, v+x'-x)} + c$$

which not only allows $O(1)$ computation on the fly, it also means that we no longer need to explcitly store the dp states; we simply need to maintain the tuples of (c, x', t) to compute the cost. Let's call these tuples **bases**.

We maintain a double-ended queue consisting of bases that cover at least one dp state at the current section. Initially, this deque only contains the base $(0, 0, 0)$. Free sections can be handled similarly to that in Subtask 5; we truncate the top base if necessary, and add another base at the bottom if necessary.

There are 4 steps to processing a fixed section x.

1. **Compute new base**:

   This requires us to process each dp state with tastiness $\leq A_x$. This can be done by processing each base in the deque from bottom to top. Note that we only need to check the highest possible tastiness dp state for each base since the dp values are non-decreasing.

2. **Delete unusable / non-optimal bases**:

   Unusable bases are bases that cover a range of dp states that all have tastiness $\leq A_x$. Non-optimal bases are bases whose dp states have lower values than those provided by the new base.

   Since dp states are non-decreasing from bottom to top, these states are at the bottom of the deque, and can be removed one by one.

3. **Truncate bottom base**:

   When we add the new base, there may be a base that it does not **entirely** remove, i.e. only a part of the dp states of that base are less than optimal.

   Here, we can perform a binary search to determine where the cutoff point is. This takes $O(\log N)$ time.

4. **Add new base**:

   We push the new base to the bottom of the deque.

The computation of the new bases can be done concurrently with the removal of unusable bases. Since the dp values within a base is non-decreasing, removing non-optimal bases can be done in O(1) each by testing the highest dp state within that base.

All deque operations are $O(N)$ overall, and we perform at most N binary searches. Hence, the overall solution is $O(N \log N)$.

Time Complexity: $O(N \log N)$ with small constant