Task 1: `L-Board`

Task author: Aloysius Lim

Subtask 1

Compute the maximum value over all possible subsets of 1,2 or 3 cells.

Subtask 2

We want to pick indices i, j such that $A[i] + \ldots + A[j]$ is maximized. A straightforward implementation of this will take $\mathcal{O}(m^3)$ time, which is too slow.

We can speed up the computation using *prefix sums* given by

$$P[i] = \begin{cases} 0 & i = 0 \\ P[i-1] + A[i] & i \geq 1 \end{cases} \tag{1}$$

Observe that $P[i] = A[1] + \ldots + A[i]$, and therefore

$$A[i] + \ldots + A[j] = A[j] - A[i-1] \tag{2}$$

We can now take the maximum of this expression over all possible $1 \leq i \leq j$.

Time complexity: $\mathcal{O}(m^2)$.

Subtask 3

We now compute prefix sums for every row and every column. For each (i, j) define

$$P_{i,j} = \begin{cases} 0 & j = 0 \\ P_{i,j-1} + A_{i,j} & j \geq 1 \end{cases} \tag{3}$$

such that

$$A_{x,y_1} + A_{x,y_1+1} + \ldots + A_{x,y_2} = P_{x,y_2} - P_{x,y_1-1} \tag{4}$$

Similarly, we can define vertical prefix sums by

$$Q_{i,j} = \begin{cases} 0 & i = 0 \\ P_{i-1,j} + A_{i,j} & i \geq 1 \end{cases} \tag{5}$$

such that

$$A_{x_1,y} + A_{x_1+1,y} + \ldots + A_{x_2,y} = P_{x_2,y} - P_{x_1-1,y} \tag{6}$$

For each (x_1, x_2, y_1, y_2), we can compute V in $\mathcal{O}(1)$ time.

Time complexity $\mathcal{O}(m^2n^2)$.

Subtask 4

Suppose we fix a corner (x, y) for the L. Notice that the horizontal arm and the vertical arm can be maximized separately and independently. Thus, we only need $\mathcal{O}(m + n)$ time to find the best L for a fixed corner. We can now maximize this over all possible corners.

Time complexity: $\mathcal{O}(nm(n + m))$.

Subtask 5

Suppose we fix the corner of the L at (x, y). Then the horizontal arm of the L can either go left or right, while the vertical arm of the L can go top or down.

In either case, we want the arm to stretch all the way to the end of the board because a longer arm is always better when $A_{i,j}$ is non-negative.

By using the prefix sums, we can do this calculation in $\mathcal{O}(1)$ time for each (i, j). Time complexity: $\mathcal{O}(mn)$.

Subtask 6

Similar to subtask 4, except that we want to speed up the computation to $\mathcal{O}(1)$ for each corner (x, y).

Define the prefix minimum by

$$M_{x,y} = \max_{1 \leq z \leq y} P_{x,z} \tag{7}$$

and observe that

$$\max_{1\leq z\leq y} A_{x,z} + \ldots A_{x,y} = P_{x,y} - M_{x,y} \tag{8}$$

Task 2: Tree Cutting

Task author: Marc Phua

Subtask 1

Consider all possible combinations to:

- Demolish a highway
- Build a high way
- Pick a starting point x
- Pick an ending point y

We compute the maximum distance across all such possible configurations, to get a solution.

Time complexity: $\mathcal{O}(n^6)$

Subtask 2

Let $d(x, y)$ denote the distance between cities x and y. Suppose we have decided to remove a highway (u, v), breaking the city into components A and B. If $x \in A$ and $y \in B$ and we choose to build a new highway, (u', v') where $u' \in A$ and $v' \in B$, then the new distance between x and y is given by

$$d(x, y) = d(x, u') + 1 + d(v', y) \tag{9}$$

Note that the maximization across u' and v' can be done independently. This suggests the algorithm:

- Choose a highway (u, v) to remove. Let the corresponding partition be A and B
- For each $x \in A$, do a bfs rooted at x to find $\max_{u' \in A} d(x, u')$. Similarly, find $\max_{v' \in B} d(v', y)$

Time complexity: $\mathcal{O}(n^3)$

Subtask 3

The idea is the same as subtask 2, except that we need to speedup the computation of $\max_{x,u'} d(x, u')$.

To do this, we do the following:

- Start at an arbitrary city x
- Using BFS to find the furthest city x_1 from x
- Use BFS again to find the furthest city x_2 from x_1

For more details, refer to here.

This will speedup the computation of $\max_{x,u' \in A} d(x, u')$ to $\mathcal{O}(n)$.

Time complexity: $\mathcal{O}(n^2)$.

Subtask 4

If the cities form a straight line, then the answer is clearly $n - 1$. If not, we can represent the network by a star.

For the city with at least 3 highways connected, call it the center. Each city which is connected to only 1 highway is called a terminal.

Let $d_1, d_2, d_3 \ldots$ be the distances from each of the terminals to the center, sorted in descending order.

Then the answer is given by $d_1 + d_2 + d_3$.

Subtask 5

We say that the **diameter** is the longest path between any two cities.

There are two possible cases.

Case 1: The highway to be demolished is not on the diameter. Let the demolition of this highway break the cities into A and B. Clearly, the diameter of A is the same as the diameter of the original network. The maximum possible diameter of B can be found the following way:

- Delete all cities on the diameter (if a highway contains a deleted city, that highway is also deleted)
- For each connected component, find the diameter, and return the largest value

This gives us the maximum possible value for case 1.

Case 2: The highway to be demolished is on the diameter. Suppose the endpoints of the diameter are (a, b) and the highway demolished is (u, v), where a and u are on the same side, v and b are on the other side after the demolition.

Claim: the diameter of the component containing a has a as one of its endpoints.

Proof: Suppose we accept the algorithm for finding the diameter in subtask 2. If we try to find the longest path from u, we will reach a.

For each w such that w is on the diameter, let $f(w)$ be the longest path starting at w that does not use any edges from the diameter. So if we remove the edge (u, v), the diameter of the component containing a is simply given by

$$\max_{w} f(w) + d(a, w) \tag{10}$$

where w is maximized over all cities on the path from a to w. As we move the highway (u, v) along the diameter, we can dynamically maintain this value. We can do this similarly for b.

Time complexity: $\mathcal{O}(n)$

Task 3: Dragonfly (`dragonfly`)

Task author: Benson Lin

Subtask 1

Subtask constraints: $n, d \leq 1000$.

We can perform a direct simulation of the problem. As every dragonfly moves from pond $h[i]$ to 1, we can simply keep track of all the bugs eaten along the way.

Time complexity: $\mathcal{O}(nd)$.

Subtask 2

Subtask constraints: $d \leq 2 \cdot 10^5$, $b[i] = d$. Since $b[i] = d$, every pond will always have at least 1 bug. We keep an array $count[j]$, which stores the number of ponds with bugs of species j along the path. Perform a depth-first-search. Whenever we visit a new pond i, we increment $count[s[i]]$ by 1, and undo the increment when we leave this pond for the last time. As we move traverse the tree, we can record down the number of nonzero values of $count$.

Time complexity: $\mathcal{O}(n + d)$.

Subtask 3

Observe that the total number of bugs eaten by all the dragonflies is at most $\sum b[i]$. However, it is possible for a dragonfly to enter a pond with no bugs, and there can be very long paths with no bugs.

However, we can use **path compression**: suppose we have a pond i_1 such that i_1 has parent i_2 and grandparent i_3. If i_2 has no more bugs left, we can reset i_1's parent to i_3.

Time complexity: $\mathcal{O}(n + d \log n)$

Subtask 4

We make use of the following definition:

- A pond i is **active** if and only if a hypothetical dragonfly going from pond 1 to pond i eats a bug of species $s[i]$ for the first time.

We wish to dynamically maintain an array $active[i]$ indicating whether pond i is active.

It is easy to compute the initial value of $active[i]$ for all i. If dragonfly eats the last bug of pond i (i.e. $b[i]$ changes from 1 to 0), then we change $active[i]$ by the following rules:

- If $active[i]$ is false, do nothing.
- If $active[i]$ is true, set $active[i]$ to false. Among all ponds $i' > i$ such that have the same species as i and has nonzero bugs, let i_0 be the first such pond. Set i_0 to be active (if such i_0 does not exist, do nothing).

Then we simply need to report the prefix sum of $active$ to answer the queries. The updates can be done using a segment tree.

Time complexity: $\mathcal{O}((n + d) \log n)$

Subtask 5

For each pond i, let $t[i]$ be the time which $b[i]$ becomes zero (for simplicity, $t[i] = d + 1$ if it never becomes zero, or $t[i] = 0$ if $b[i] = 0$ in the beginning). We wish to compute $t[i]$ for each i.

There are a few possible ways to do it. One of the solutions involves heavy-light decomposition:

- Attach a segment tree on every heavy path recording the number of bugs on that path
- For each dragonfly, we can decompose the path taken into heavy paths, and perform $\mathcal{O}(\log n)$ range updates
- By doing so, we can detect when the number of bugs turn to 0
- This runs in $\mathcal{O}((n + d) \log^2 n)$ as there are $\mathcal{O}(\log n)$ segment tree updates for every dragonfly

Another possible idea uses set merging:

- For every pond i, write record down the set $S_i = \{j | h[j] = i\}$ (in other words, the set of all dragonflies which visited pond i).

- Observe that, $t[i]$ is the $b[i]$-th smallest number among the union of $S_{i'}$, where i' is taken over all descendants of i.
- By small-to-large merging of order statistics trees, we can solve the problem in $\mathcal{O}((n + d)\log^2 n)$.

Once this is done, we simply store the active/inactive state of every pond. Since species are distinct, a pond is active if and only if it has no bugs.

Time complexity: $\mathcal{O}((n + d)\log^2 n)$

Subtask 6

We first calculate $t[i]$ as per subtask 5. However, the rules for maintaining the active/inactive state of the ponds is now more complicated.

A pond is active iff

- Every ancestor with the same bug species is inactive
- The current time is smaller than $t[i]$ (i.e. there is at least 1 bug left).

We can set up a forest such that every pond's parent is the first pond on its path to root that shares the same bug species. Call this the species forest.

To 'activate' a pond i,

- If current time is smaller than $t[i]$, we simply turn the pond active.
- Else, we 'activate' all its children in the species forest.

To 'deactivate' a pond i, we turn the pond to inactive, as well as 'activate' its children.

By storing the ponds on a fenwick tree with preorder traversal, we can toggle the active/inactive state, as well as answer path sums, both in logarithmic time.

Time complexity: $\mathcal{O}((n + d)\log^2 n)$

Subtask 7

We can use a divide and conquer strategy.

- At time $d/2$, decide which ponds are active.
- For every active pond, add $d/2$ to $t[i]$.
- For every pond, define its active (inactive) parent as the first active (inactive) ancestor on its path to root, if it exists.
- If a dragonfly ends up at an inactive pond after time $d/2$, we change its ending pond to be its active parent.
- Similarly, if a dragonfly ends up at an active pond before time $d/2$, we change its ending pond to be its inactive parent.
- We can build an active tree and an inactive tree, solving the two subproblems independently.

Let $T(n, d)$ be the time taken to solve this problem. Then we have the recurrence relation

$$T(n, d) = T(n_1, d/2) + T(n_2, d/2) + \mathcal{O}(n) \tag{11}$$

for some $n_1 + n_2 = n$. This gives us $T(n, d) = \mathcal{O}((n + d) \log d)$.

There is another solution based on this data structure known as wavelet tree. Given an array $a[]$, the wavelet tree data structure can be built in $\mathcal{O}(n \log n)$ time, and supports the following operation:

- Given (l, r, k), return the k-th smallest item in $a[l \dots r]$

We use the same idea as the set merging solution.

Time complexity: $\mathcal{O}((n + d) \log(d))$