Task 1: `Train Or Bus`

Authored by: Ling Yan Hao

Prepared by: Ling Yan Hao

Editorial written by: Ling Yan Hao

For each i, we simply take the faster option to travel from city i to city $i + 1$ and sum them up. This can be done using an if-else statement or the min function.

Time complexity: $O(n)$

Task 2: `Ducks and Buttons`

Authored by: Kiew Tian Le, Ernest

Prepared by: Kiew Tian Le, Ernest

Editorial written by: Kiew Tian Le, Ernest

Subtask 1

Additional constraints: $n = 2$

There are only two buttons, and the first button is always immediately pressed as $a[i] \leq d$. Hence, we just need to find the minimum number of duck moves to press the second button. By using $a[2]$ moves, we can move $a[2]$ ducks onto the 2nd square. Since it is not possible to use fewer moves, the solution is just to output $a[2]$.

Time complexity: $O(1)$

Subtask 2

Additional constraints: $a[i] = 0$ for all i

As all buttons require 0 ducks, all buttons are pressed without having to make any ducks move. As such, the solution is to output 0.

Time complexity: $O(1)$

Subtask 3

Additional constraints: $a[i] \leq 1$ for all i

Consider what happens if $a = [0, 0, 1, 0, 1, 0, 0]$. This means we must move at least one duck to buttons 3 and 5. We can move just a single duck all the way to button 5, and by travelling one step at a time, all buttons before button 5 are also pressed. There is also no need to travel past the 5th button, as all further buttons are 0. Hence, the solution is to output the index of the last 1 inside a. (0-indexed)

Time complexity: $O(n)$

Subtask 4

Additional constraints: All values of $a[i]$ are equal

Let k be the value of $a[1]$ (which is equal to all other elements in a). This means we need to have k ducks at every position at some point in time. It is optimal to press buttons in left-to-right order. As such, we should move k ducks to position 2, then move the same k ducks to position 3, and so on until all buttons are pressed.

The number of moves used by this strategy is $(n-1) \times k$.

Time complexity: $O(1)$

Subtask 6

Note that the explanation for subtask 5 is left for after subtask 6 and 7.

Additional constraints: $a[i]$ is non-decreasing

We can use a very similar strategy to subtask 4, pressing buttons from left to right.
To press the ith button, we need $a[i]$ ducks.
However, as $a[i]$ is non-decreasing, the last element in a is the largest.
This means that in the end, we need $a[n]$ ducks on the nth button.
If we move $a[n]$ ducks to any position i, we will also press button i as $a[i] \leq a[n]$.
Hence, we can move ducks similarly to subtask 4, but using $a[n]$ instead of k.

The number of moves used by this strategy is $(n-1) \times a[n]$.

Time complexity: $O(n)$ for reading input

Subtask 7

Additional constraints: $a[i]$ is non-increasing

Consider $a = [7, 5, 4, 4, 2]$.
If we press buttons in order from left to right, the number of ducks we need will not increase.
Hence, it is optimal to move 5 ducks to position 2, but continue moving 4 ducks to position 3 and leave one duck behind in position 2 as it is no longer necessary since all future buttons only need 4 or less ducks.

In general, the number of ducks that need to reach position i is just $a[i]$, as for all $j > i$,

$a[j] \leq a[i]$.
Calculating the number of moves this takes can be done by considering position by position.
The number of ducks that have moved to position i is simply $a[i]$, using $a[i]$ moves.
Therefore, summing all $a[i]$ (except for $a[1]$) will give us the total number of moves used.

Time complexity: $O(n)$

Subtask 5

Additional constraints: $n, d \leq 1000$

Consider $a = [7, 5, 4, 2, 4]$.
Although the 4th position only requires 2 ducks, the 5th position requires 4 ducks, so at some point, 4 ducks must have reached or passed through the 4th position.

More generally, the number of ducks that have to reach position i is $max(a[i], a[i+1], ...a[n])$.

Hence, if we create a new array $b[i]$ such that $b[i] = max(a[i], a[i+1], ...a[n])$, the answer is the sum of all $b[i]$ (except for $b[1]$).

Time complexity: $O(n^2)$

Subtask 8

Additional constraints: None

Constructing the array $b[i]$ is the bottleneck in terms of our code's runtime, as it is the only component that runs in $O(n^2)$ time.

To speed it up, we can just loop backwards, maintaining the maximum value seen so far.
Example Pseudocode:

```
maximumValue = 0
For each index i from n to 1:
    maximumValue = max(maximumValue, a[i])
    b[i] = maximumValue
```

Time complexity: $O(n)$

Task 3: `Snacks`

Authored by: Kiew Tian Le, Ernest and Yaw Chur Zhe

Prepared by: Kiew Tian Le, Ernest

Editorial written by: Kiew Tian Le, Ernest

Subtask 1

Additional constraints: $q = 0$

There are no queries, but you are still expected to print the sum of the array.

Time complexity: $O(n)$

Subtask 2

Additional constraints: $n, q \leq 1000$

As $n, q \leq 1000$, an $O(nq)$ solution will pass under the time limit.
For every query, iterate through the entire array, changing values accordingly and obtaining the sum of the array.

Time complexity: $O(nq)$

Subtask 3

Additional constraints: $l[j] = r[j] \leq 200\ 000$ and $a[i], x[j] \leq 200\ 000$ Shor will only eat snacks with deliciousness exactly equal to $l[j]$.
As deliciousness will not exceed $200\ 000$, we can just maintain the number of times each value appears in our array (the frequency array) and the current sum.

To improve implementation, we handle replacement in two steps: deleting old and inserting new values.
When we erase X elements of value Y, our sum decreases by $X \times Y$.
When we add X elements of value Y, our sum increases by $X \times Y$.

Since $l[j] = r[j]$, only two indices in the frequency array are modified in each query.

Hence, we can maintain the frequency array and the sum in $O(n+q)$ time.

Time complexity: $O(n+q)$

Subtask 4

Additional constraints: $l[j] = r[j]$

This is the same as subtask 4, but deliciousness can be up to 10^9.
Instead of using an array with 10^9 elements which would exceed both time and memory limits, we can use a `map` or `unordered_map` in C++, or a dictionary in Python.

Time complexity: $O(n+q)$ with `unordered_map`, or $O((n+q)\ log(n))$ with `map`.

Subtask 5

Additional constraints: $x[j] = 0$

Every element that is affected by a query will turn to 0.
This leads us to a solution with amortisation.
Since all elements affected by queries are deleted, if we are able to only iterate through elements affected by queries, we only iterate through each element at most once.
To iterate through only elements affected by queries, we can use `map.lower_bound(l[j])` to get an iterator to the first key greater than or equal to $l[j]$, which is the first element to be deleted.
We can then increment the iterator until the key exceeds $r[j]$, at which point we break the loop.
This ensures we only loop through elements that will be deleted.

Since we iterate through each element at most once, the number of accesses, insertions and deletions on the map is $O(n)$, which takes $O(n\ log\ n)$ time.

In each query, we also call the lower bound function on the map, which takes $O(log\ n)$ time.

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

Subtask 6

Additional constraints: None

Every element that is affected by a query no longer has to turn to 0.

However, the amortisation proof still works, as we insert only one new element per query. This causes the number of accesses, deletions and insertions on the map to be $O(n+q)$, taking $O((n+q)\ log\ (n+q))$ time, which passes.

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

Task 4: `Itinerary`

Authored by: Stuart Lim Yi Xiong

Prepared by: Stuart Lim Yi Xiong

Editorial written by: Stuart Lim Yi Xiong

Subtask 1

Additional constraints: $n \leq 1000, m = 2n - 1$

As noted in the task description, a tour visits all cities a total of $2n - 1$ times. For this subtask, $2n - 1$ is equal to m which is the length of s. Hence, a tour is an itinerary if and only if the sequence of cities visited by the tour is equal to s.

It remains to check whether the trip that visits exactly the sequence of cities s is a tour at all. The trip would use the roads $(s[1], s[2]), (s[2], s[3]), \ldots, (s[2n - 2], s[2n - 1])$. We need to check that all these roads exist in the input and that each road in the input is used exactly twice. (You may recall that tours have several other properties, but those do not need to be checked as they follow the main property that each road is used twice.) Any solution that can do this in $O(n^2)$ time or better should pass.

After verifying that the trip is a tour, we can output 1 for the starting city $s[1]$ and 0 for all other cities.

Time complexity: $O(n^2)$ or better

Subtask 2

Additional constraints: $u[i] = 1, v[i] = i + 1$

The input describes a star graph, where there is a road connecting city 1 directly to all other cities. All cities other than city 1 are *leaves*.

Consider what a tour would look like. A tour starting from city 1 would alternate between city 1 and all leaves, so each leaf is visited exactly once. A tour starting from a leaf would move to city 1, alternate between city 1 and all leaves except for the start and finally return to the start, so the start is visited exactly twice and all other leaves are visited exactly once. We can count how many times each leaf appears in s. This allows us to identify two situations where no itinerary can be constructed.

1. Any leaf appears more than twice.
2. Any leaf appears exactly twice, but $s[1]$ is not this leaf, or $s[m]$ is not this leaf.

After eliminating the above cases, all leaves appear at most once, except for at most one leaf that can appear twice but only as $s[1]$ and $s[m]$. If there is such a leaf, then an itinerary must start from and end at $s[1] = s[m]$. One way to construct such an itinerary is as follows: find all leaves that don't appear in s and insert them just before $s[m]$ in an arbitrary order, then insert 1's between all adjacent pairs of leaves (convince yourself that this works!). On the other hand, a tour starting from any other city already uses the road $(1, s[1])$ twice to move from the start to $s[1]$, then to $s[2]$, and therefore cannot use the same road to visit $s[m] = s[1]$ again.

Otherwise, all leaves appear at most once in s. Suppose $s[i]$ is a leaf and $i \neq 1, i \neq m$, then $s[i]$ cannot be a starting position as the road $(1, s[i])$ must be used more than twice to visit all required cities. However, it is possible to construct itineraries starting from all other cities, including cities $1, s[1], s[m]$, using a similar strategy as the case where a leaf appears twice.

It is **not** sufficient to check if $s[1] = s[m]$ due to the edge case where $m = 1$, for which all cities are valid starting positions.

Time complexity: $O(n)$

Subtask 3

Additional constraints: $n \leq 1000, u[i] = i, v[i] = i + 1$

The input describes a path graph, where the n cities are arranged in increasing order on a straight line. There are at most two tours starting from each city i.

1. Decreasing-first tour: From city i, go to city 1, then to city n and back to city i.
2. Increasing-first tour: From city i, go to city n, then to city 1 and back to city i.

Build the sequences of visited cities of the two tours. Then an itinerary starts from city i if s is a subsequence of at least one of the built sequences. Subsequence checking is a classic dynamic programming problem which can be done in $O(n + m) = O(n)$ time. Repeat for all cities.

Time complexity: $O(n^2)$

Subtask 4

Additional constraints: $u[i] = i, v[i] = i + 1$

Observe the two types of tours starting from each city in the Subtask 3 section. A tour switches direction (decreasing or increasing city indices) at most twice. Hence, for an itinerary to exist, s can only switch direction at most twice. Assume s is increasing-first and consider three cases depending on the number of directional switches. Proofs are omitted.

1. s is increasing. The decreasing-first tour from any starting position is an itinerary.
2. s increases, then decreases.
    - If $s[1] < s[m]$, the decreasing-first tours from cities $\leq s[m]$ are itineraries. There are no itineraries from other cities.
    - If $s[m] \leq s[1]$, the increasing-first tours from cities $\leq s[1]$ are itineraries. There are no itineraries from other cities.
3. s increases, decreases, then increases.
    - If $s[1] < s[m]$, there are no itineraries.
    - If $s[m] \leq s[1]$, the increasing-first tours between cities $s[m]$ and $s[1]$ inclusive are itineraries. There are no itineraries from other cities.

The case where s is decreasing-first is symmetric.

Time complexity: $O(n)$

Subtask 5

Additional constraints: $n \leq 1000, m \leq 10$

Trees and Simple Paths

We are given that there are n cities, $n - 1$ roads and it is possible to move between all pairs of cities using the roads (in other words, the graph is connected). Such a graph is known as a tree. Another property of trees is that trees are acyclic. Due to a tree's connected and acyclic nature, between any pair of cities, there is exactly one walk that uses each road at most once and visits each city at most once. This walk is called a **simple path**, or just a **path**. The following two lemmas are well known.

Lemma 4.1. *Any road used in a path is also used in any walk between the same two cities.*

Lemma 4.2. *Any city visited in a path is also visited in any walk between the same two cities.*

Note that by Lemma 4.1, among all walks between two given cities, the path uses each road the minimum possible number of times.

Event Trips

Instead of building tours and determining if they are itineraries, we will prioritise visiting the sequence of cities first and see if we can modify the walk to make it a tour. If so, the modified walk is an itinerary. We define event walks and shortest event walks as follows.

Definition (Event Walk)**.** *A walk that satisfies the following properties:*

D1. The walk starts and ends at the same city.

D2. The walk visits the cities $s[1], s[2], \ldots, s[m]$ *in order, not necessarily consecutively.*

Definition (Shortest Event Walk)**.** *An event walk that uses each road the minimum possible number of times over all event walks starting from the same city.*

Lemma 4.1 tells us that the path between two cities is the unique walk that uses each road the minimum possible number of times. Hence, the shortest event walk starting from some city src is the unique walk obtained by appending the paths between cities src and $s[1]$, cities $s[1]$ and $s[2]$ and so on, ending with the path between $s[m]$ and src.

By observing definitions, we note that a walk is an itinerary if and only if it is both an event walk and a tour. Given some starting city, if our shortest event walk uses any road more than twice, then any event walk uses that road more than twice, so no event walks are tours, so there are no itineraries from that starting city.

Constructing an Itinerary

Now suppose the shortest event walk uses all roads at most twice. We will show that we can always insert the unused roads into the shortest event walk to construct a tour, therefore creating an itinerary. *Note: Contestants are* ***not*** *required to implement the construction of an itinerary.*

Here are more properties of walks (not necessarily event walks).

Theorem 4.3. *For all walks, the visited cities and used roads form a connected subtree of the original graph.*

Proof. We can use induction by adding one road and one city in each step. We start on some city without using any roads. Clearly the subtree containing only the start city itself is connected. Each time we use a road to move from a city u to another city v, the road (u, v) and city v are connected to city u which is in the connected subtree. □

Theorem 4.4. *For all walks starting and ending in the same city, each road is used the same number of times in each direction.*

Proof. Consider each road separately. Since the original graph is acyclic, the road splits the graph into two connected components. The only way to cross from one component into the other is via the road linking them. Since we start and end in the same city, we also start and end in the same component with respect to the road, so we must have crossed the road the same number of times in each direction. □

Corollary 4.5. *For all walks starting and ending in the same city, each road is used an even number of times.*

Proof. Follows from Theorem 4.4. □

By Theorem 4.4/Corollary 4.5 and the assumption that the shortest event walk uses all roads at most twice, we know that each road is either unused or used twice (once in each direction) by the shortest event walk. Furthermore, by Theorem 4.3, the roads that are used twice and the cities at the ends of used roads form a connected subtree. Call this the **event subtree**.

Now consider unused roads. Partition all unused roads into sets of connected roads. These sets should be maximal, i.e., two unused roads that are directly or indirectly connected by other unused roads should be in the same set. Then each set of roads and the cities at the ends of these roads form a connected **unused subtree**.

Theorem 4.6. *Exactly one city in each unused subtree is also in the event subtree.*

Proof. We separately prove "at least one" and "at most one."

(At least one) Arbitrarily choose cities u and v in the unused and event subtrees respectively. If either u or v are in both subtrees then we are done. Otherwise, consider the path from u to v. The first few roads from u lie in the unused subtree, while the last few roads into v lie in the event subtree. Adapting the intermediate value theorem into this situation, there exists a city w where the road leading into w is in the unused subtree while the road leading out of w is in the event subtree. City w is in both subtrees.

(At most one) Let u and v be two distinct cities in both subtrees and consider the path between them. Since the event subtree is connected, the path between them contains only used roads, whereas since the unused subtree is connected, the path between them contains only unused roads. The only way to resolve this contradiction is to let there be no roads between the two cities. Then $u = v$ which is also a contradiction. □

Since we pass through all cities in the event subtree during the shortest event walk, by Theorem 4.6 we can make a detour into each unused subtree. By traversing the unused subtree in a manner similar to DFS (depth-first search), we can use all roads twice. Performing this for all unused subtrees gives us an itinerary. In conclusion, we have the following theorem:

Theorem 4.7. *There exists an itinerary starting from city v if and only if the shortest event walk from city v uses each road at most twice.*

Implementation

To find the path from city u to city v, perform DFS (depth-first search) or BFS (breadth-first search) from city u. For each city i, record its 'parent' which is the city we enter city i from. After DFS or BFS has ended, we can follow the trail of parents starting from v to trace the path back to u. It takes $O(n)$ time to find a path between any two cities in this way.

A shortest event walk is composed of $m + 1$ paths, so it takes $O(nm)$ time to build the shortest event walk starting from a given city. To determine if there is an itinerary, it remains to check that no road is used more than twice. We can use a C++ map or Python dictionary which maps pairs of integers (representing roads) to integers (representing frequencies). Since traversing a road in either direction ($u \to v$ or $v \to u$) should contribute to the same counter, we increment the counter of the key pair $(\min(u, v), \max(u, v))$. At the end, we check that no counter is more than 2. This takes $O(nm \log nm)$ time.

All of the above is done starting from all n cities.

Time complexity: $O(n^2 m \log nm)$

Subtask 6

Additional constraints: $n \leq 1000$

Notice that any shortest event walk with length strictly more than $2(n - 1) = 2n - 2$ definitely uses a road more than twice, because there are only $n - 1$ distinct roads.

Hence, before incrementing the counter of a road, we can simply check if the counter has already reached 2. If so, we can conclude early that no itineraries exist by only checking $O(n)$ roads. On the other hand, if we are able to process the entire shortest event walk without running into the above case, then we must have only checked $O(n)$ roads as well.

Alternatively, since using a road twice is equivalent to using the road once in each direction, we can store each used road with its direction in a set. Thus, traversing a road in the directions $u \to v$ and $v \to u$ are respectively represented by inserting (u, v) and (v, u) into the set. We can check if the directed road already exists in the set before trying to insert it.

Both solutions use $O(n \log n)$ time to check each starting city.

Time complexity: $O(n^2 \log n)$

Subtask 7

Additional constraints: $m \leq 10$

Subtask 6 still counts the same roads many times. Specifically, all roads between $s[1]$ and $s[2]$, $s[2]$ and $s[3]$, ..., $s[m-1]$ and $s[m]$ are recounted. Also suppose that src is changed from u to v, where (u, v) is a road. Only the road (u, v) is added or removed from the paths between src and $s[1]$ and between $s[m]$ and src. Whether the road is added or removed can be determined by observing the parent data. For example, if $parent[v] = u$ when DFS or BFS is performed from $s[1]$, then src moves away from $s[1]$ and the road (u, v) is added.

Perform DFS or BFS from $s[1], s[2], \ldots, s[m]$ and store the results. First place src at an arbitrary city and build the frequency map of the shortest event walk as in Subtask 5 or 6. To test all possible starting cities, we move src across roads one road at a time until all cities have been tested. This can be done with DFS from the original src.

Finally, it would be too slow to check the entire frequency map for roads used more than twice for every single starting city. We can maintain a separate set data structure that stores all roads used more than twice. This set should be updated together with the frequency map.

Time complexity: $O(nm \log nm)$ or $O(nm + n \log n)$

Subtasks 8 and 9

Additional constraints: $s[1] = s[m]$ (Subtask 8), None (Subtask 9)

Using DFS and BFS to find the shortest event walk takes $O(nm)$ time, which is clearly too

slow. Below we discuss a faster way.

Root the whole tree at some arbitrary city and run DFS or BFS from the root. In addition to parents, find each city's depth, defined as its distance from the root. The algorithm to find the roads along the path between two cities u and v is given as follows.

1. If $depth[u] < depth[v]$, swap u and v.
2. While $depth[u] > depth[v]$, add the road $(u, parent[u])$, then set $u := parent[u]$.
3. While $u \neq v$, add the roads $(u, parent[u])$ and $(v, parent[v])$, then set $u := parent[u]$ and $v := parent[v]$.

The algorithm is modified from the one used to find lowest common ancestors and can be proved correct in a similar way. Its runtime is proportional to the length of the path. Hence, we can construct the shortest event walk from an arbitrary city in time proportional to its length, with $O(n)$ time preprocessing for one run of DFS/BFS. A naive implementation would still be too slow in the worst case, where the shortest event walk has $O(nm)$ length. But by checking a road's frequency counter before updating it, as described in Subtask 6, we only need to check $O(n)$ roads.

Finally, we need to test all possible starting cities by moving the starting city src across roads, like in Subtask 7. The required parent data can be obtained via DFS or BFS from $s[1]$ and $s[m]$.

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

The $\log n$ factor is due to the frequency checking of roads and can be removed. Consider the endpoint city of each road that has larger depth. It can be shown that these cities are pairwise distinct. Hence, these cities can be used as indices for a frequency array.

Time complexity: $O(n)$

Alternative Solution

Tours have the interesting property of supporting rotations. Suppose a tour starts from city u and first uses the road $u \to v$, so that the tour takes the form $u \to v \to \cdots \to u$. By moving the road $u \to v$ to the end, we obtain $v \to \cdots \to u \to v$ which is a tour starting from city v. Similarly, we can move roads from the end of a tour to its beginning to change the starting city. This hints that itineraries can also be constructed from other itineraries via rotation, subject to the condition that $s[1], \ldots, s[m]$ are visited in order.

Let $s[1]$ be the root. We find the shortest event walk starting from $s[1]$, so that the event subtree consists exactly of the cities on the paths between $s[i]$ and $s[i+1]$.

Theorem 4.8. *Let u be a city in the event subtree. Let v be a city in the unused subtree of u. There is an itinerary starting from u if and only if there is an itinerary starting from v.*

Proof. We prove the $\Rightarrow$ direction. Suppose there is an itinerary starting from u, where we make a detour into its unused subtree at some arbitrary time.

$$u \underbrace{\to \cdots \to}_{\text{rest of tree}} u \underbrace{\to \cdots \to v \to \cdots \to}_{\text{unused subtree}} u \underbrace{\to \cdots \to}_{\text{rest of tree}} u$$

Other than city u itself, the unused subtree does not contain the cities $s[1], s[2], \ldots, s[m]$, so the important cities are visited in the "rest of tree" parts. Thus, if we pass through city u several times in the shortest event walk, we can pick any time to make a detour into the unused subtree. We will choose to make the detour in the very start of the itinerary. This is not a rotation.

$$u \underbrace{\to \cdots \to v \to \cdots \to}_{\text{unused subtree}} u \underbrace{\to \cdots \to}_{\text{rest of tree}} u$$

Finally, we can rotate roads from the start to the end to obtain an itinerary that starts from city v.

$$\underbrace{v \to \cdots \to}_{\text{unused subtree}} u \underbrace{\to \cdots \to}_{\text{rest of tree}} u \underbrace{\to \cdots \to v}_{\text{unused subtree}}$$

The $\Leftarrow$ direction is similar. An itinerary starting from v takes the structure of the third itinerary above. Rotations can be used to convert it into the second itinerary, which completes the proof. □

The above theorem suggests that we only need to solve the problem of whether itineraries exist starting from each city in the event subtree.

Theorem 4.9. *Let u be a city on the path between $s[1]$ and $s[m]$. There is an itinerary starting from $s[1]$ if and only if there is an itinerary starting from u.*

Proof. This is clearly true if $u = s[1]$. Suppose $u \neq s[1]$.

($\Rightarrow$) By Lemma 4.2, any walk from $s[m]$ to $s[1]$ must pass through u. Hence, an itinerary starting from $s[1]$ looks like:

$$s[1] \to \cdots \to s[m] \to \cdots \to u \to \cdots \to s[1]$$

By rotating edges from the end to the start we get an itinerary starting from u:

$$u \to \cdots \to s[1] \to \cdots \to s[m] \to \cdots \to u$$

($\Leftarrow$) An itinerary starting from $u \neq s[1]$ takes the form of the second itinerary above. By rotating edges from the start to the end we get the first itinerary. □

Theorem 4.10. *Let v be a city in the event subtree but **not** on the path between $s[1]$ and $s[m]$. There is no itinerary starting from v.*

Proof. Let e be any road that is on the path between v and $s[1]$, but not on the path between $s[1]$ and $s[m]$. At least one such road must exist, because v is not on the path between $s[1]$ and $s[m]$.

Recall that the shortest event walk from $s[1]$ consists of the paths from $s[1]$ to $s[2]$, $s[2]$ to $s[3]$, $\ldots$, $s[m-1]$ to $s[m]$, $s[m]$ to $s[1]$. Since v is in the event subtree, the shortest event walk from $s[1]$ visits v, and thus contains a walk from $s[1]$ to v. Hence, e is used at least once in the shortest event walk. But each road is used an even number of times (Corollary 4.5), so e is used at least twice.

As defined above, e does not lie on the path between $s[1]$ and $s[m]$, so the paths from $s[1]$ to $s[2]$, $s[2]$ to $s[3]$, $\ldots$, $s[m-1]$ to $s[m]$ use the road e at least twice.

The shortest event walk from v consists of these paths, as well as the path between v and $s[1]$, and between $s[m]$ and v. Since the path between v and $s[1]$ contains e, the shortest event walk from v uses e more than twice. By Theorem 4.7, there is no itinerary starting from v. □

Using the above theorems, the solution is simple. Similar to the first solution, construct the shortest event walk from $s[1]$. If any road is used more than twice, then there are no itineraries at all. Otherwise, there are itineraries starting from all cities on the path between $s[1]$ and $s[m]$, as well as their unused subtrees.

Time complexity: $O(n)$

Task 5: `Lasers 2`

Authored by: Yaw Chur Zhe

Prepared by: Ryan Goh Choon Aik

Editorial written by: Yaw Chur Zhe and Ryan Goh Choon Aik

Subtask 1

Additional constraints: $k = 0, c[i] = 10^9$

Since Pavement is broke, all the sliding walls will remain in their original positions. It suffices to loop through all the sliding walls and keep track of the set of columns that are blocked, and report the answer accordingly.

Time complexity: $O(hw)$ or better

Subtask 2

Additional constraints: $l[i] = r[i]$ for all i

To unblock a column, we will need to unlock all the walls that originally cross that column and move them out of the way. Let $g(x)$ be the associated total cost for the x-th column.

Since all walls are of length 1, we just need to choose one column to slide all the unlocked walls to, in order to achieve the maximum number of unblocked columns.

To select the set of columns to be unblocked, a natural greedy approach would be to order the columns from lowest $g(x)$ to highest $g(x)$, then unblock the columns in that order until we run out of money or have 1 column left.

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

Subtask 3

Additional constraints: $h, w \leq 18$

For the rest of the editorial, denote i^* to be the row with the longest wall. If there are multiple

choices for i^*, choose any one.

Since h and w are small, we can use subset brute force. Specifically, we can brute force all the possible set of walls to be unlocked (Method 1) or all the possible sets of columns to be unblocked (Method 2).

Method 1: Firstly, we make sure that we have enough budget to unlock all the walls for a certain set before we process the set. There are two scenarios:

Scenario 1: The chosen set of walls do not contain the longest wall. In this case, we can slide all the unlocked walls behind the longest wall. This obviously will give the maximum number of unblocked columns for this set of walls.

Scenario 2: The chosen set of walls do contain the longest wall. In this case, we can brute force all the possible positions of the longest wall, reducing it to scenario 1.

Time complexity: $O(2^h hw)$

Method 2: For a certain set of columns that we want to unlock, we compute the total cost to unlock the walls that cross the selected columns. If the total cost exceeds k, we reject this set of columns. Then, we compute X, the length of the longest wall unlocked. We also compute Y, the longest section of columns that we did not intend to unblock. If $X > Y$, then it is impossible to unblock this specific set of columns (since the longest block has nowhere to go without blocking at least one of the columns that we intend to clear), otherwise it is possible as we can slide all the unlocked walls to this longest section.

Time complexity: $O(2^w hw)$

Subtask 4

Additional constraints: $h, w \leq 100, k \leq 2000$

We employ wishful thinking and assume that wall i^* does not move. Now, to clear any column that does not contain the longest wall, we can move the blocking walls behind the wall on row i^*. Hence, the columns from l_{i^*} to r_{i^*} inclusive can never be unblocked.

We can do DP on the other columns. Let $dp(x, c)$ be the maximum number of columns less than x that we can clear with a budget of c dollars, assuming that column x is empty.

We will have

$$dp(x,c) = \begin{cases} -\infty, & \text{if } c < 0 \\ \max(0, \max_{1 \leq z < x}(dp(z, c - f(z,x)) + 1)), & \text{if } x \geq 2 \end{cases}$$

where $f(z, x)$ is the total cost to unlock the walls that cross column z but not column x, which can be precomputed in $O(hw^2)$ time. This is to ensure that we do not double count when we clear the x-th column later on. The unlocked walls essentially "disappear" as they are moved under the umbrella of wall i^*, rendering them redundant. To ensure that wall i^* is never moved, we can temporarily set its cost to infinity.

Since there are $O(wk)$ states and the transition takes $O(w)$ time (assuming that f is precomputed), the DP will take $O(w^2k)$ time.

To calculate the final answer for the case where the longest block does not move, since $dp(x, c)$ assumes that column x is empty, we will also have to calculate the cost $g(x)$ to clear the x-th column, which is simply the total cost of walls that cross column x. Hence, our final answer will be the maximum of $(dp(x, k - g(x)) + 1)$ across all columns x that do not contain the longest wall.

To handle the cases where the longest wall moves, we first unlock the longest wall for a cost of c_{i^*}, and then brute force all the possible locations of the longest wall. For each possible location, we use the method from above, assuming the wall is immovable. This will increase the time complexity by a factor of $O(w)$, as the number of possible positions the longest wall can be placed is bounded by w.

Time complexity: $O(hw^2 + w^3k)$

Subtask 5

Additional constraints: $h, w \leq 100$

We will need to kick c out of our state, since c can be up to 10^9.

We can redefine our DP as follows (state shuffling):

Let $dp(x, y)$ be the minimum amount of money (in dollars) required to ensure that there are at least y empty columns among columns 1 to x, with the additional assumption that column x is empty.

Our transitions can be modified as such:

$$dp(x, y) = \begin{cases} \min_{1 \leq z < x}(dp(z, y-1) + f(z, x)), & \text{if } y \geq 2 \\ 0, & \text{if } y = 1 \\ \infty, & \text{if } y > x \end{cases}$$

Now there are $O(w^2)$ states and the transition takes $O(w)$ time, so the DP will take $O(w^3)$ time.

Our final answer is the maximum y such that there exists an x where $dp(x, y) + g(x) \leq k$.

We can then brute force the position of the longest wall in a similar way as Subtask 4 to get the actual final answer. As before, this will take $O(w)$ time.

Time complexity: $O(hw^2 + w^4)$

Subtask 6

Additional constraints: $h, w \leq 500$

We will need to speed up the case where the longest wall moves. We can do this using another DP.

Let $dp2(x, y, b)$ be the minimum amount of money (in dollars) required to ensure that there are at least y empty columns among columns 1 to x, with the additional assumption that column x is empty, where b is a boolean variable indicating whether the final position of the block in row i^* has been decided.

Let l be the length of i^*. Our transitions are as follows:

$$dp2(x, y, 0) = \begin{cases} \min_{1 \leq z < x}(dp2(z, y-1, [x-z-1 \geq l]) + f(z, x)), & \text{if } y \geq 2 \\ 0, & \text{if } y = 1 \text{ and } x - 1 \geq l \\ \infty, & \text{if } y = 1 \\ \infty, & \text{if } y > x \end{cases}$$

$$dp2(x, y, 1) = \begin{cases} \min_{1 \leq z < x}(dp2(z, y-1, 1) + f(z, x)), & \text{if } y \geq 2 \\ 0, & \text{if } y = 1 \\ \infty, & \text{if } y > x \end{cases}$$

In essence, if we want to transition from $b = 0$ to $b = 1$, we check whether the number of blocked columns between two consecutive empty columns is at least equal to the length of the longest block. If so, we can place the longest block in that space. Note that we will have to decide where to place the longest block, so $dp2(x, 1, 0) = \infty$ for all $x - 1 < l$ because this means that there is no gap between two consecutive empty columns that can fit the longest block.

Once we decide on the position of the longest block (so $b = 1$), the rest of the DP is exactly the same as the DP in Subtask 5.

Since there are $O(w^2)$ states (the b parameter only increases the number of states by a constant factor) and $O(w)$ transition for both cases, this DP also runs in $O(w^3)$ time.

For this case, the final answer can be retrieved by finding the maximum y such that there exists an x where $dp(x, y, 0) + g(x) \leq k$ or there exists an x where $dp(x, y, 1) + g(x) \leq k$ and $w - x \geq l$.

The actual final answer can be computed by the maximum of the final answers arising from both DPs in Subtasks 5 and 6. However, since we do not have to loop through $O(w)$ positions for the longest block, the time complexity has been reduced to $O(hw^2 + w^3)$.

Time complexity: $O(hw^2 + w^3)$

Subtasks 7 and 8

Additional constraints: All locked sliding walls are the same length (Subtask 7), None (Subtask 8)

We will need to optimise the $O(w)$ transition of the 2 DPs. First, we cease precomputation of the function f outside of our DP.

We will attempt to optimise the transition of our first DP.

For all values of y, we will sweep from column 1 to column w, and store the value of $dp(z, y - 1) + f(z, x)$ in index z of a segment tree. Clearly, $dp(x, y)$ = minimum of all the values in the segment tree. Also, recall that $f(z, x)$ is the total cost to unlock the walls that cross column z but not column x. So, when we sweep from column x to column $x + 1$, we add the costs of all the walls that end at column x to our segment tree. This can be done in $O(\log w)$ using a range add operation for each wall. Then we add $dp(x, y - 1)$ to our segment tree. This also takes $O(\log w)$ time.

Since there are only $O(h)$ walls, the total time complexity of the transition is $O(h \log w + w \log w)$ for each y which is bounded by w.

For the second DP, the optimisation for the transition is rather similar. We can create 2 segment trees for $b = 0$ and $b = 1$ respectively, and modify the updates and queries accordingly.

Subtask 7 is for solutions that did not make the longest block observation but chose a random block, or for buggy solutions.

Time complexity: $O(w(w + h) \log(w))$