Solution for 'dungeons'

It is stated in the constraints section that the player will eventually reach dungeon n. Why is this so?

Observe that the player's size increases after each step. Once the player's size is greater than 10^7, he wins in every chamber. From there, the chamber number can only increase.

1 Subtask 1

Simulate the movement of the player stepwise in a naive way.

2 Subtask 2

We make the following observation:

> Once the player loses, the size of the player must double. As such, the number of losses is bounded by $\log_2 S$, where $S = \max s[i]$.

For each dungeon i, we draw an 'outgoing edge' $(s[i], w[i])$ indicating the gain in size and the next location if a player wins.

We may accumulate edges as follows:

$$(s[i], w[i]) \star (s[w[i]], w[w[i]]) = (s[i] + s[w[i]], w[w[i]])$$

to indicate the movement of the player when the player wins twice in a row.

Next we make this observation:

> A player can cross that if the player's size (before the increment) is larger than every $s[i]$ on the path of this player.

As such, we can record down the largest $s[i]$ for every edge and build power-of-2 tables.

Using power-of-2 tables, we can binary search till the point which the player loses next.

It takes $O(n \log n)$ time to build the power-of-2 tables, after which finding the next loss will take $O(\log n)$ time.

The total time complexity is $O(n \log n + q \log n \log S)$.

3 Subtask 3

Similar to the previous subtask, we accumulate edges by the following rule:

$$(p[i], l[i]) \star (p[l[i]], l[l[i]]) = (p[i] + p[l[i]], l[l[i]])$$

Again, we build power-of-2 tables.

The movement of the player can be simulated until $z >= s$ (unless the player reaches dungeon n first). After that, we can simulate all the way to the ending chamber using a simple prefix sum.

The movement of the player can be broken down into two phases. The first phase consists of the movement before the player's first win. The second consists of the remaining movements.

The first phase can be solved by using power-of-2 tables.

The second phase can be solved by using a static prefix sum to the winning dungeon. Time complexity $O((n+q)\log S)$.

4 Subtask 4

Let the distinct values of s be $t_0, t_1, \ldots, t_4$.

Consider a player with size z, where $t_0 \leq z < t_1$. We wish to simulate the movements of this player until $z \geq t_1$ (or $x = n$).

We say that dungeon i is winning if $s[i] \leq t_0$, and losing if $s[i] \geq t_1$.

Following the above subtasks, we draw an edge $(p[i], l[i])$ for losing dungeons and $(s[i], w[i])$ for winning dungeons.

The edges can be accumulated as before. Construct power-of-2 tables until the player's size exceeds t_1.

We can repeat the above process until $x = n$. Time complexity: $O(k(n+q)\log S)$ where k is the number of different sizes.

5 Subtask 5

If we use the approach in the previous subtasks, there are too many tables to build. The idea is to group dungeons of similar size together.

We will group the dungeons by the $\log_2 s[i]$, and construct phases $1, 2, \ldots, \log_2 10^7$.

For phase k, dungeon i is winning if $s[i] < 2^k$ and losing otherwise.

Now consider a player with size z, where $2^k \leq z < 2^{k+1}$. We cannot simply jump in the naive way until $z \geq 2^{k+1}$. This is because we have declared all dungeons with size $\geq 2^k$ as winning. Thus the player might win before his size exceeds 2^{k+1}. This contrasts with the previous situation where there cannot be any dungeon with size between t_0 and t_1.

As such, we have to store an extra parameter in the edge: a 'winning threshold'.

Let the winning threshold be $s[i]$ for a winning edge and ∞ for a losing edge. Then we can now accumulate edges via the following rule:

```
(inc[i], out[i], threshold[i]) * (inc[out[i]], out[out[i]], threshold[out[i]])
= (inc[i] + inc[out[i]], out[out[i]], min(threshold[i], threshold[out[i]] - inc[i]))
```

A player can now travel past an edge if

1. The player's size (before the increment) is less than the threshold.
2. The player's size (after the increment) is less than 2^{k+1}.

In actual fact, condition (2) can be dropped, as the threshold prevents a player from accidentally losing a dungeon that should be won. Observe that once we run the simulation until (1) is met and win that chamber, the player's size must have increased by 2^k, so that the player's size is at least 2^{k+1}.

Therefore, condition (1) is only triggered at most $\log S$ times.

By building the power-of-2 tables this way, it takes $O(\log S)$ time to advance 1 phase. Therefore the total time taken is $O((n+q)\log^2 S)$.

6 Subtask 6

The solution for subtask 5 will run out of memory as it requires $O(n \log^2 S)$ memory.

It is possible to reduce the time and memory consumption taken to build the power-of-2 tables to $O(n)$ for each table. There are a few ways to do it.

One way is to use the technique described in `https://codeforces.com/blog/entry/74847`, which basically speeds up the building of the tables. Another solution is to adjust the phases such that each phase consists of strengths $[b^k, b^{k+1}-1]$ for some larger b. In practice, $b = 8$ should work.

We will present an alternative solution here.

6.1 Alternative solution

For every dungeon i and every phase p, we compute the path from dungeon i to the next dungeon of phase p (we do not allow the empty edge, even if dungeon i is of phase p). Take note that when computing the path, we use the behavior of phase p (i.e. win if $s[i] < 2^p$, lose otherwise). The size increase, threshold, and outgoing location are tracked as described in the earlier subtasks. We call these edges 'superedges'. Among these superedges, call a superedge special if dungeon i is of phase p. This gives rise to $n \log S$, which are split into $\log S$ phases.

We build a power-of-2 table on the special superedges. This uses $O(n \log S)$ pre-computation time as there are only $O(n)$ superedges.

To answer the queries. We have a variable p, denoting the player's phase. Initially, $p = 0$. Now suppose that the player is at dungeon x and has phase p:

- Case 1: $s[x] < 2^p$ or $s[x] \geq 2^{p+1}$. Look for the superedge of dungeon x and phase p. If we can follow that edge, follow it and proceed to case 2. If not, increment p.

- Case 2: $2^p \leq s[x] < 2^{p+1}$. Follow the superedges of phase p (note that no matter how many steps we take, we only remain in dungeons of phase p). Proceed until we cannot cross any more superedges in phase p. Simulate 1 step using the naive method, and then increase p.

The reason why this algorithm works depends on the following invariant:

- When the player has phase p, the player will not lose in any dungeon x with $x < 2^p$.

To see why this is true, we only need to focus on what happens when p is incremented. (In the beginning, $p = 0$ so the condition is trivially true.)

Suppose we reach case 1 and we cannot pass through the superedge leading to phase p. This means that we must have won somewhere along this path of that superedge. What is the size of this dungeon? It must be at least 2^p by assumption. However, along this path there cannot be any dungeons with size between 2^p and 2^{p+1}, because of the way the superedges are constructed. Therefore we must have won a dungeon of size at least 2^{p+1}.

Suppose we reach case 2. This time round, the argument from the previous paragraph mostly holds, except for the fact that we may win on the dungeon the player is currently on. Therefore, we are required to take 1 baby step to simulate.

Each query can be answered in $O(\log^2 S)$ time.