Egg Drop Challenge

We will calculate $dp[i]$ - minimum time required for the i-th person to catch an egg. It is clear that in the optimal $dp[i]$ if the i-th person catches an egg that was thrown by j-th person, than either j-th person should be throwing the egg with maximum possible speed (v_j), or i-th person should be catching the egg with maximum possible speed (u_i), because otherwise j-th person could've thrown it faster and the time spent would be less.

We can distinguish this cases with inequality $u_j^2 + 2 \cdot h_j \le v_i^2 + 2 \cdot h_i$

- If the condition is satisfied, we should relax $dp[i]$ with $dp[j] - u_j + \sqrt{u_j^2 + 2 \cdot (h_j - h_i)}$
- Otherwise, we should relax $dp[i]$ with $dp[j] - \sqrt{v_i^2 + 2 \cdot (h_i - h_j)} + v_i$ (if the value under the root is non-negative)

To support the relaxations of the second type, we will make a structure similar to Li Chao Tree. We can see that we should find minimum value among the functions which can be represented as $f_j(x) = a_j - \sqrt{b_j + x}$ in the point $x = v_i^2 + 2 \cdot h_i$, where $a_j = dp[j]$, $b_j = -2 \cdot h_j$, the domain of $f_j(x)$ is $[2 \cdot h_j, \infty)$. Also, there is a special condition - we should relax $dp[i]$ throw $f_j(x)$ only if $u_j^2 + 2 \cdot h_j > v_i^2 + 2 \cdot h_i$, this inequality actually meaning that $x < u_j^2 + 2 \cdot h_j$. That's why we will define $f_j(x)$ on $[2 \cdot h_j, u_j^2 + 2 \cdot h_j)$. Now we just have to find minimum value among $f_j(x)$ to relax $dp[i]$ with it $+v_i$.

The structure will support two operations:

- Add a function $a - \sqrt{b + x}$ on $[l, r]$
- For x find minimum $f_j(x)$ among functions in the structure

All the x from the queries are known in advance: $v_i^2 + 2 \cdot h_i$ for all i. We will build a segment tree on these points. Just like in ordinary Li Chao Tree we will store a function in each segment tree node, and for each point x an optimal function will be somewhere among the nodes on the path from root to leaf x. Since two functions can intersect only in one point, this structure works just like ordinary Li Chao Tree with operation of adding a function on subsegment in $O(\log^2 n)$.

The relaxations of first type are more complicated. Similarly, we have to find minimum value among the functions which can be represented as $f_j(x) = a_j + \sqrt{b_j + x}$ in the point $x = -2 \cdot h_i$, where $a_j = dp[j] - u_j$, $b_j = u_j^2 + 2 \cdot h_j$ and the domain of $f_j(x)$ is $[-u_j^2 - 2 \cdot h_j, \infty)$, but here we also have to support an extra condition $u_j^2 + 2 \cdot h_j \le v_i^2 + 2 \cdot h_i$.

That's why we will sort the people by value $u_j^2 + 2 \cdot h_j$, so the query is finding minimum among the functions with their indices j on the prefix of array. For it we will maintain a Fenwick Tree storing Li Chao Trees. We will add function in the position of j (which actually causes $O(\log n)$ additions to Li Chao Tree), and find minimum among up to $O(\log n)$ Li Chao Trees to find real minimum. So, the adding operations work in $O(\log^3 n)$, operations to find minimum work in $O(\log^2 n)$.

There is a way to speed up the adding operations. We can see that all the queries we make are done in the points $-2 \cdot h_i$, while calculating $dp[i]$ for i from n to 1, meaning that these x_i form an increasing sequence. Also, each next add operation is adding a function on subsegment $[-u_j^2 - 2 \cdot h_j, \infty)$, while all the actual queries will be only on $[-2 \cdot h_j, \infty)$. That's why the Li Chao Trees we are using have an extra parameter - T, so that at any moment of time all the added functions are defined on $[T, \infty)$ and all the queries are in $[T, \infty)$ and T is increasing over time. That's why in the operations inside the Li Chao Tree we will just consider that instead of the original points x_i all the points are $max(T, x_i)$. Now, instead of adding a function on a subsegment we will add a function on the whole structure. The add operation works in $O(\log n)$, with all the additions from Fenwick Tree it works in $O(\log^2 n)$.

So, the total time is $O(n \log^2 n)$.