Task: EXP
Exponents (author: Mateusz Radecki)

BOI TORUŃ 2025

BOI 2025, Day 1: Analysis.

There are multiple possible solutions working in complexities ranging from $\mathcal{O}((n+q)\log^2 n)$ to $\mathcal{O}((n+q)\log n)$, below we describe only one of them.

According to the statement, $2^a + 2^b$ evaluates to $2^{\max(a,b)+1}$. Thus, evaluating an expression $2^{a_1} + 2^{a_2} + \ldots + 2^{a_k}$ can be seen as building a binary tree on leaves labelled by $a_1, a_2, \ldots, a_k$ when read from left to right that minimises the largest sum of a_i plus the depth of its corresponding leaf, over $i = 1, 2, \ldots, k$. We call this quantity the cost of $a_1, a_2, \ldots, a_k$.

Denoting the largest number occurring in the input by M, we observe that the answer to any query never exceeds $M + \log n$, as we can always build a complete binary tree of depth $\log n$.

Now let us consider how to determine the answers to all the queries. Let the largest number occurring in the input be M, and denote the positions where it occurs by $x_1 < x_2 < \ldots < x_k$. We will first (recursively) determine the answer for each query range that does not contain any occurrence of M inside, and then the others. Thus, each recursive call concerns a contiguous fragment $A[i..j]$ of the input. On top of determining the answer for each query range contained in $A[i..j]$, we will also determine some additional information concerning the prefixes and suffixes of $A[i..j]$. Namely, let M' be the largest number occurring in $A[i..j]$. Then, we partition $A[i..j]$ into blocks as follows. Starting from the whole $A[i..j]$, we keep cutting off the longest prefix of the current suffix $A[k..j]$ with cost at most M', until the current suffix is empty. This gives us a greedy partition of $A[i..j]$ into blocks of cost at most M'. It is immediate that the cost of every but the very last block is exactly M', while it might be smaller for the last one. We observe that given such a partition we can obtain the partition corresponding to cutting off longest prefixes with cost at most M by simply repeatedly merging $2^{M-M'}$ consecutive blocks. Because $M' < M$, the amortized cost of such a conversion is constant, as the number of blocks decreases by a factor of two (or more).

Now consider a query range that contains at least one occurrence of M, and let x_i be the rightmost such occurrence. The information recursively computed for $A[(x_i+1)..(x_{i+1}-1)]$ allows us to determine the longest prefix of $A[(x_i+1)..(x_{i+1}-1)]$ equal to a concatenation to x blocks of cost at most M each. Similarly (after appropriately merging the information about suffixes of fragments) we can determine the longest suffix of $A[1..x_i]$ equal to a concatenation of y blocks of cost at most M each. We assume that in both cases this information is stored in an array, so given x or y we can determine in constant time the length of the corresponding prefix or suffix. We observe that for the cost of the currently considered query range to be $M+t$ we must have such x and y with $x + y = 2^t$. For a fixed t, we can determine maximal ranges with such a property by iterating over every x, retrieving its corresponding longest prefix, and then retrieving the longest suffix for $y = 2^t - x$. This takes time proportional to the number of blocks obtained for $A[(x_i+1)..(x_{i+1}-1)]$ plus one per each t.

Updating the information about suffixes of fragments can be done proportional to the number of blocks obtained for $A[(x_i+1)..(x_{i+1}-1)]$ plus , as this is essentially prepending that many entries to the array. After having processed the last fragment, this also gives us the information about suffixes that should be returned by the recursive call (the information about prefixes can be computed similarly).

The amortized cost of processing all the fragments can be bounded by $\mathcal{O}(k \log n)$, so $\mathcal{O}(n \log n)$ over all the recursive calls. This gives us maximal ranges with a given cost, which allows to answer each query in $\mathcal{O}(\log n)$ time (or even $\mathcal{O}(\log \log n)$ by binary searching for the cost).