A Find Values

Denote with:

- $p_1 = s_2 + s_3 + s_4$.
- $p_2 = s_1 + s_3 + s_4$.
- $p_3 = s_1 + s_2 + s_4$.
- $p_4 = s_1 + s_2 + s_3$.

1 Test cases where $s_1 = s_2 = s_3 = s_4$

Since all the numbers are the same, $p_1 = s_2 + s_3 + s_4 = 3 \cdot s_1$, and $s_1 = \frac{p_1}{3}$.

2 Test cases where $0 \leq s_1, s_2, s_3, s_4 \leq 1$

We can iterate over all possible combinations for the values of s_1, s_2, s_3 and s_4 and check which one matches the input

3 Full solution

Notice that $p_1 + p_2 + p_3 + p_4 = (s_2 + s_3 + s_4) + (s_1 + s_3 + s_4) + (s_1 + s_2 + s_4) + (s_1 + s_2 + s_3) = 3 \cdot (s_1 + s_2 + s_3 + s_4)$. Let $S = s_1 + s_2 + s_3 + s_4$. Then, $S = \frac{p_1+p_2+p_3+p_4}{3}$. Finally:

- $s_1 = S - p_1$.
- $s_2 = S - p_2$.
- $s_3 = S - p_3$.
- $s_4 = S - p_4$.

B Split Array

1 Subtask 1

Iterate over all possible divisions of the array in intervals, and check if the length of each interval is twice the length of the previous.

Complexity: $O(n \cdot 2^n)$.

2 Subtask 2

Iterate over all possible divisions of the array in intervals, and check if the sum of each interval is twice the sum of the previous.

Complexity: $O(n \cdot 2^n)$.

3 Subtask 3

Let's do the following dynamic programming:

- $dp[i][j]$ - the total number of valid ways to split the first i elements of the array such that the last interval starts at position j.

Then, the answer is:

$$\sum_{j \leq n} dp[n][j]$$

To calculate this dp, we will iterate over the previous interval as follows:

- $dp[i][j] = \sum dp[k][j-1]$, for all $k \leq j-1$ such that $2 \cdot (a_k + a_{k+1} + ... + a_{j-1}) = (a_j + a_{j+1} + ... + a_i)$.

Complexity: $O(n^3)$.

4 Subtask 4

We will do the same dynamic programming as in the previous subtask. We are only going to observe that for each pair (j, i), there is at most one k such that $2 \cdot (a_k + a_{k+1} + ... + a_{j-1}) = (a_j + a_{j+1} + ... + a_i)$. We are going to do the binary search over such k.

Complexity: $O(n^2 \log n)$.

5 Full solution

Let the sum of the array be S and let's assume that in one of the divisions of the array on intervals we have k intervals with the sum of the first interval being x. Then the sum of the second interval is $2x$, the sum of the third interval is $4x$, etc. The sum of the k-th one is $2^{k-1} \cdot x$. Notice that the sum of all of them is:

$$S = x + 2 \cdot x + 4 \cdot x + ... + \cdot 2^{k-1} \cdot x = (2^k - 1) \cdot x$$

This means that we have at most $\log_2 S$ choices for the number of intervals, but also that their sums for the fixed length are predetermined. We will iterate over all possible choices for k and then for each k check if $(2^k - 1)|S$. If it does, we will calculate $x = \frac{S}{2^k-1}$. Then we will check if we can split the array on intervals with fixed sums. We can do this efficiently by considering the prefix sums for example. In subtask 5 we don't have to perform this additional check.

Complexity: $O(n \log n)$.

C Missing Lengths

1 Subtask 1

We are given a path. As the missing length, we can simply print the difference of distances of any two adjacent vertices with an edge of unknown length between them.

Complexity: $O(n)$.

2 Subtask 2

We can iterate over all possibilities for the missing length and for each of them execute a Dijkstra's algorithm to check distances.

Complexity: $O(w \cdot m \log m)$.

3 Subtask 4

Let us denote the shortest distances from the input as $true[u]$. Let's fix the length of the unknown edges. Notice that by increasing this length, the distances to all the vertices can only increase. This inspires us to do a binary search. We fix the length to be x and we execute Dijkstra's algorithm. Denote these distances as dist. Then we check if there exists a vertex u such that $dist[u] \neq true[u]$. If such a vertex doesn't exist we have found an answer. Otherwise:

- If $dist[u] > true[u]$ we need to decrease x, i.e. we go to lower values in the binary search.
- If $dist[u] < true[u]$ we need to increase x, i.e. we go to higher values in the binary search.

To solve subtask three, we need the same observation for the binary search approach, but we can do so without implementing Dijsktra's algorithm.

Complexity:$O(m \log m \log w)$.

4 Full solution

Let:

$$l = \max(|true[u] - true[v]|) \text{ over all edges with an unknown length}$$

Notice that l is a lower bound to the solution. If the edge length was any smaller, the shortest distance to one of the vertices would decrease. It can be shown that this edge is a valid choice for the solution (by analyzing the shortest distances and showing that no vertex u such that $dist[u] < true[u]$ exists).

Complexity:$O(m)$.

D Count Intervals

1 Subtask 1

We iterate over every interval. Then we check if OR of values in the interval is equal to the OR of values outside of it.

Complexity: $O(n^3)$.

2 Subtask 2

We iterate over every interval. We first iterate over the left end, and then over the possible right ends in increasing order. We keep track of OR of values inside the interval while we iterate. Then we check if it is equal to the OR of values outside the interval, we first precompute the prefix and suffix OR.

Complexity: $O(n^2)$.

3 Subtask 3

Since values are either 0 or 1, we have two cases:

- All elements are equal to 0. Then any interval is valid.
- There is at least one value 1. The interval we select must contain at least one value 1 and it mustn't contain all of them. We iterate over the left end and find the next value 1. If there is at least one value 1 before the current left end, we can pick any right end after the next value 1, otherwise, we can pick any right end between the next value 1 and the last value 1 in the array.

Complexity: $O(n)$.

4 Subtask 4

Let OR of all elements be equal to x. Then both the OR of values inside $[l, r]$ and OR of values outside it must be equal to x. Like in the previous solution, we iterate over the left end, l. Then we iterate over bits and find the index in the array where the next bit appears. The leftmost position on which this interval can end is the maximum of these indices. Finally, we iterate over all bits that haven't appeared before l. For all of them, we find their last appearance in the array. The rightmost position on which the interval starting on l can end is the minimum of these indices.

Complexity: $O(n \log A)$.

5 Full solution

We need to optimize the previous solution. We will again iterate over the left end l and keep track of the right end r using the two-pointer technique. To find the OR of elements in $[l, r]$, we will keep track of the elements in the interval using the queue. Then we just need to find the OR of elements in the queue. For this, we can simulate the queue using two stacks in the following way:

- When an element is added, we add it to the first stack.

- When an element is removed, we first check if the second stack is empty. If it is, we will add all elements from the first stack to it (and pop them from the first stack). Using this process, we reverse the order of elements from the first stack, thereby ensuring the "FIFO" (First-In-First-Out) policy. Finally, we remove the top of the second stack.

Keeping track of OR of elements in the stack is easier because we only need to add prefix OR to the stack. Using this procedure we can find the leftmost right end. In the similar way, we can cyclically repeat the array, reverse it and then use the same procedure to find the rightmost right end.

Complexity: $O(n)$.