InfO(1)Cup 2026 Editorial

Problem Digit Sum

Problem author: Luca Mureșan

In this problem, you are given integers $A_1, \ldots, A_n, X_1, \ldots, X_q$, all at most D digits long, and must compute the sum of the digits of all the numbers $A_1 + X_i, \cdots, A_n + X_i$ for each $i = 1, \ldots, q$.

First, suppose that no "carries" happen when adding $A_j + X_i$ for any $j = 1, \ldots, n$. (E.g. if we add $3 + 9$ we get 12, but we needed to carry a 1; whereas if we add $3 + 2$ we get 5 without carrying.) Then the sum of the digits is simply the sum of the digits of $A_1, \ldots, A_n$, plus the sum of the digits of X_i multiplied by n. Now, consider any carry in any particular addition; for example $3 + 9 = 12$. In this case, our original carry-less estimate would say that the sum of digits of $3 + 9$ is 12; however we must subtract a correcting factor of -9.

By this logic, it is not difficult to show that the sum of the digits of $A_1 + X_i, \ldots, A_n + X_i$ is just the sum of the digits of $A_1, \ldots, A_n$, plus n times the sum of digits of X_i, minus 9 times the number of carries that happen in any addition. Let us now compute this.

In general, for any integer number T, let $T^{(i)}$ denote its last i digits. For example, $54321^{(3)} = 321$. When is it the case that a carry occurs at digit i (from the right) when adding $X + Y$? A simple calculation shows that this only happens if

$$X^{(i)} + Y^{(i)} > \underbrace{9 \ldots 9}_{i \text{ times}}.$$

If we define $\neg T$ to be the result of turning each digit T from i into $9 - i$, we see equivalently that a carry happens at digit i when adding $X + Y$ if and only if $X > \neg Y$ (of course padding $\neg Y$ appropriately with 9s).

Thus, we see that the answer for a number X_i is determined by the sum, for each t, of the number of integers among $A_1^{(t)}, \ldots, A_n^{(t)}$ which are greater than than $\neg X^{(t)}$. But this implies that, if we can compute the sorted order of $A_1^{(t)}, \ldots, A_n^{(t)}, \neg X_1^{(t)}, \ldots, \neg X_q^{(t)}$ for each $t = 1, \ldots, D$, with at most $O(D(n + q))$ extra time we can solve our problem.

Luckily, it is easy to do this. Indeed, the order of $A_1^{(0)}, \ldots, A_n^{(0)}, X_1^{(0)}, \ldots, X_q^{(0)}$ is trivial to compute (all the numbers are 0). For every $t > 0$, we see that the order is given by (i) first ordering by the digit at position t, (ii) then by ordering the order for the suffix of length $t - 1$. This shows us that we can recompute the order for larger and larger values of t by stably partitioning with respect to the current digit. (This is essentially the exact same procedure as we would do if applying radix sort to the sequence $A_1, \dots, A_n, \neg X_1, \dots, \neg X_q$. All of this takes $O(D(n+q))$ time, which is sufficient for a full score.

Problem Training

Problem author: Luca Mureșan

Let's analyze a more general version of the game played in this task:

Definition 1 (Token game). A and B are playing a game on an undirected graph G. First, A places a token on one of the vertices. Then, starting with B, players alternate turns. At each step, the player whose turn it is places a token on a neighbor of the last placed token. The players can't place two or more tokens on the same vertex. Whoever can't move loses.

Observation 1. *B wins the game if and only if G admits a perfect matching.*

Proof. Proving an "if and only if" statement means proving two implications; we do each separately.

$\Leftarrow$. We show that, if G admits a perfect matching, then B can win easily. Let

$$(u_1, v_1), \dots, (u_k, v_k)$$

be a perfect matching in G. B's strategy is then as follows: Whenever A places a token on u_i, B can place a token on v_i and "complete" the pair (u_i, v_i). Similarly, if A places a token on v_i, B can place a token on u_i. Basically, B just adopts a symmetric strategy relative to A with respect to this perfect matching. Thus, any time A moves, B can make one more move. Since A will eventually run out of moves (by parity), B will win.

$\Rightarrow$ Now let's prove the harder part: if G has no perfect matching, then A can win.

Let $(u_1, v_1), (u_2, v_2), \dots, (u_k, v_k)$ be a maximum matching in G. Since G has no perfect matching, there must be some vertex w which isn't in any pair (u_i, v_i). A can start the game by placing a token on this vertex. Since w is unmatched and $(u_1, v_1), (u_2, v_2), \dots$ is a maximum matching, every neighbor of w must be matched. Otherwise, we could match w with an unmatched neighbor, increasing the size of the maximum matching. This is a contradiction.

Therefore, every neighbor of w is matched. B will choose one of these neighbors and place a token on it. Now, A can use the same symmetric strategy that B used in the case of a perfect matching. There is only one issue: B could move to an unmatched vertex. This would completely ruin A's strategy, but we can actually show that this is impossible.

Suppose $x_1, x_2, \dots, x_t$ are the vertices on which we placed tokens (x_i is the vertex where we placed the i-th token). Currently, the matched pairs are $(x_2, x_3), (x_4, x_5), \dots, (x_{t-2}, x_{t-1})$. Since x_1 and x_t are unmatched, and $x_1, x_2, x_3, \dots, x_t$ are neighbors consecutively, we could rematch them as follows: $(x_1, x_2), (x_3, x_4), \dots, (x_{t-1}, x_t)$, increasing the size of the maximum matching by 1. This is a contradiction. $\square$

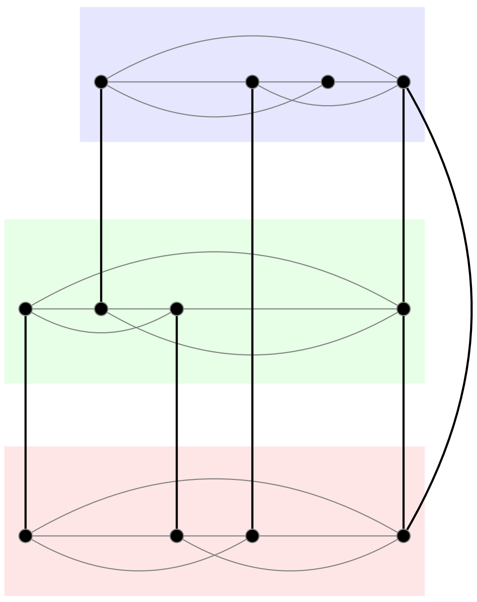

Figure 1: Example graph.

Now let's solve the actual task. Let's play the game separately for each color. The vertices of the graph are the cells with this color, and the edges are between any two vertices sharing a row or a column. Let us investigate the structure of this graph: the vertices of the graph are a set of cells of form (i, j) where $i = 1, 2, 3$, and we have an edge between (i, j) and (i', j') if and only if $i = i'$ or $j = j'$. In particular, we see that we can split our graph into (at most) 3 parts that are all connected within themselves: the vertices of form $(1, i)$, $(2, j)$ and $(3, k)$ respectively. (These correspond to the three rows.) See Figure 1 for illustration.

We observe that the total number of vertices must be even — or else no perfect matching is possible. Consider the number of vertices on each row, call these numbers c_1, c_2, c_3. If all of these numbers are even, then it is easy to find a perfect matching: in each row all vertices are connected, and if the row has an even number of vertices, we can match them arbitrarily and still get a perfect matching.

Now let us assume that $c_i \not\equiv 0 \bmod 2$ for at least one $i = 1, 2, 3$. Since $c_1 + c_2 + c_3 \equiv 0 \bmod 2$, we have that exactly 2 of c_1, c_2, c_3 are even, and the last is odd — so assume without loss of generality that c_1, c_2 are even, and c_3 is odd.

Suppose now that there exists a perfect matching: such a perfect matching cannot match all of rows 1 and 2 within themselves (due to parity). In particular, they must match at least one vertex of these rows with some vertex in another row. Either we have an edge between rows 1 and 2 in this matching, or edges between rows 1 and 3 and rows 2 and 3. By contrapositive, we deduce that if no such edges exist, then no matching exists.

But conversely, it is easy to check that if edges between rows 1 and 2, or edges from row 1 to row 3 to row 2 exist, then a perfect matching must exist. Indeed, in the first case we can add any such edge to our perfect matching, and we are back to the case where all rows have an even number of vertices (which we showed earlier admits a perfect matching). If no such edge exists, but edges from row 1 to row 3 to row 2 exist — then we note that if these edges cannot share a common vertex in row 3 (or else all 3 vertices are on the same column, and an edge from row 1 to row 2 should exist as well, which we assumed doesn't exist). Hence, we can add these 2 *disjoint* edges to the matching, and yet again all our rows now have an even number of vertices.

Hence, we see that a matching exists if and only if (i) every row has an even number of vertices, or (ii) two rows have an odd number of vertices, but have an edge between them, or (iii) two rows have an odd number of vertices, but both have an edge to the 3rd row.

To solve the problem, we maintain at every step, and for every color (i) the parity of the number of appearances of that color on every row, and (ii) for every pair of rows i, j, the number of columns k where the elements (i, k) and (j, k) both contain that color. Using this information, for each particular color we can compute whether a perfect matching exists or not (simply implement the condition from the previous paragraph). The last observation is that every update only changes 2 colours.

Bonus. Solve the problem when N can take any value.

Problem Triangulation

Problem author: Matei Neacșu

In this problem, you are given a triangulated polygon with N vertices labeled from 1 to N. You must assign each triangle a distinct integer from 2 to $N - 1$ from among the labels of its vertices, and count the number of ways of doing this.

First, we say that a vertex x is *isolated* if it is part of a single triangle xyz, and $y - x - z$ are edge of the polygon. Our solution will revolve around these isolated vertices.

First, note that if $x = 2, \dots, N - 1$ is an isolated vertex, then the triangle it is a corner of must always be assigned its label. Furthermore, an isolated vertex can be removed without disconnecting the polygon — so we may immediately eliminate any isolated vertices except if 1 or $N - 1$ are isolated.

Let us next observe that (i) if the polygon is a triangle, then there are 3 isolated vertices, and (ii) if the polygon is not a triangle, there must exist at least two isolated vertices, which are not neighbours in the polygon. To see that this is the case, consider any pair of vertices xy, which belong to a triangle, but are not on the edge of the polygon. Cut up the polygon along xy, into two smaller polygons A, B (see Figure 2 for an illustration). Consider A. If A is a triangle, then one of its 3 vertices is isolated. Otherwise, we know that A has at least 2 non-neighbour isolated vertices. Hence, at least one of these vertices is not found among x or y — and thus is also an isolated vertex in the original polygon. Applying the same reasoning to B implies the desired result.

Furthermore, let us consider the case where there are only 2 isolated vertices. We claim that every triangle edge xy which is internal to the polygon must separate the two isolated vertices. Indeed, consider again Figure 2 and assume for contradiction that the two isolated vertices are in A. But, by the discussion previously there must also be an isolated vertex coming from B, which contradicts there only being 2 isolated vertices. Hence, we see that every such edge separates the two isolated vertices.

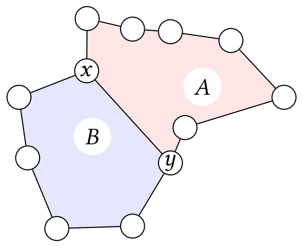

Figure 2: Example polygon

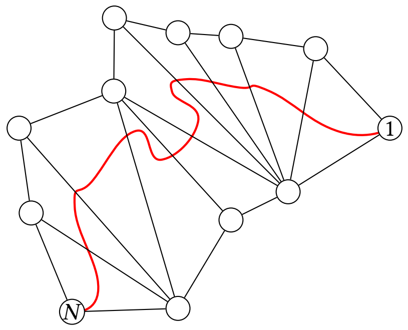

Figure 3: Polygon with only 2 isolated vertices

But this implies that in the 2-isolated vertex case, the triangles form a *path* between the two isolated vertices. By this, we mean that to reach one isolated vertex from another (without passing through any other vertices), one must pass once through every triangle once. Indeed, if we could get away without using some triangle, then that triangle could be separated from both isolated vertices by one of its edges — which contradicts what we proved above. Hence, in this case we observe that the triangles are linearly ordered, based on the distance from one isolated vertex or the other. (See Figure 3 for illustration.)

Our observations thus far let us reach an algorithm. We have eliminated all isolated vertices except perhaps 1 and N. If only these two vertices are isolated, the triangles form a path as in Figure 3. Note that now, we may choose to set the triangle adjacent to 1 to one of the other two vertices in the triangle, say x — however even after doing this, we are left with a suffix of the path of triangles from 1 to N, where two vertex labels don't need to be used (i.e. x and N — since 1 was just eliminated, we needn't even remember it). It is easy to see that any choice leads to a valid solution (so long as, in future, we continue to immediately eliminate isolated vertices), so this suffices for finding a solution.

For counting: notice that, so long as we save the triangle containing N for last, we are left with a suffix of the path from 1 to N. Furthermore, the vertices whose labels no longer need to be mentioned are N and one of the vertices on the edge passed by the path to leave the current suffix of the path we are left with. There are hence at most $2N$ possible states, and we can use dynamic programming to compute our answer.

Problem Trees

Problem author: Tamio-Vesa Nakajima

In this task, you are given a rooted binary tree T, and must create sets of vertices with disjoint subtrees, whose subtree sizes sum up to as many distinct integers as possible, and such that the number of selected vertices is small. If you have $|T|$ vertices in T, it turns out that you can represent every number from 1 to $|T|$ as the sum of sizes of a set of vertices, while using only $O(|T| \log |T|)$ vertices overall. We explain how to do this shortly — this is sufficient to get 100 points on this task.

For some arbitrary tree T, let $f(T)$ be a function that takes T, and returns a set of sets of vertices of T. We define $f(T)$ in the following way:

***T* is a leaf.** In this case, suppose the root of T is r. We return $\{\{r\}\}$.

***T* has one child.** In this case suppose r is the root of T, and C is the subtree of the child of r. We return $f(C) \cup \{\{r\}\}$.

***T* has two children.** In this case, suppose r is the root, suppose a, b are the two children, and suppose A, B are the subtrees rooted at a, b. Suppose $|A| \leq |B|$. We return

$$\{\{r\}\} \cup \{S \cup b \mid S \in f(A)\} \cup f(B).$$

It is not difficult to check, inductively, that

- $f(T)$ returns a set $\{S_1, \ldots, S_{|T|}\}$ containing $|T|$ sets, such that the sum of the sizes of the subtrees of the vertices in S_i is exactly i.
- The sum of the sizes of the sets in $f(T)$ is at most $O(|T| \log |T|)$.

Furthermore, it is easy to implement f efficiently, e.g. return the result of f as a linked list of linked lists. This is sufficient to solve the task.

An even more efficient implementation is possible though — one which uses only $O(n)$ extra space. Indeed, rather than imaging that each vertex recursively calls its children then finally merges the lists returned, perhaps appending an element — we can imagine that each vertex receives from its parent a list of vertices that should be appended to whatever lists it adds to the result. (The interested reader should read up on *continuation passing style* — this transformation is merely an application of this generic idea.) Then, we note that every vertex passes to its larger child the list it received from its parent unaltered; whereas it gives the smaller child the list but appending the index of the larger child. In any case, by manually maintaining the list structure and not copying the list unnecesarily, but rather reusing the same list suffixes, we get the following implementation.

```
#include <iostream>
using namespace std;

constexpr int maxn = (1 << 18) + 100;
```

```
/* Linked list type. nil represents an empty list */
struct node {
    int head, len; node *tail;
} *nil = new node { 0, 0, nullptr };

/* cons(x, t) creates a new list containing x then t */
node *cons(int head, node *tail){
    return new node { head, 1 + tail->len, tail };
}

int n, par[maxn], sz[maxn], sum[maxn];
node *ret[maxn] = { nullptr, nil };
/* par[i] = parent of i
 * sz[i]  = size of subtree of i
 * sum[i] = sum of *indices* of children of i
 * ret[i] = list of values to prepend to output for node i
 */

int main() {
    /* Read n; account for input size */
    cin >> n;
    n = (1 << n);

    /* Read parent vector; compute sum[i] */
    for (int i = 2; i <= n; ++i) cin >> par[i], sum[par[i]] += i;

    /* Compute subtree sizes */
    for (int i = n; i > 1; --i) sz[par[i]] += ++sz[i];

    /* Compute ret[i] */
    for (int i = 2; i <= n; ++i) {
        /* This is my brother */
        int bro = sum[par[i]] - i;

        /* If i is the heavy child of par[i] */
        if(sz[i] > sz[bro] || (sz[i] == sz[bro] && i > bro))
            /* Then our list is our parents list */
            ret[i] = ret[par[i]];
        else
            /* Otherwise, we add our brother */
            ret[i] = cons(bro, ret[par[i]]);
    }

    /* Print output */
    cout << n << '\n';
    for (int i = 1; i <= n; ++i) {
        /* For a node, print i then its list */
        cout << ret[i]->len + 1 << ' ' << i << ' ';
        for(node *x = ret[i] ; x != nil; x = x->tail) cout << x->head << ' ';
        cout << '\n';
    }

    return 0;
}
```

An even more parsimonious implementation is possible, which we now showcase:

```
#include <iostream>

long long n, i, j, b, p[300000], s[300000], x[300000];
struct node { long long h, t, l; } r[300000];

int main() {
    std::cin >> n, std::cout << (1 << n);
    for (i = 2; i <= (1 << n); ++i) std::cin >> p[i], x[p[i]] ^= i;
    for (i = 1 << n; i > 1; --i) s[p[i]] += ++s[i], s[i] <<= 32, s[i] += i;
    for (i = 1, b = 0; i <= (1 << n); ++i, b = i ^ x[p[i]]) {
        r[i] = (s[i] < s[b] ? node{b, p[i], r[p[i]].l + 1} : r[p[i]]);
        std::cout << '\n' << r[i].l + 1 << ' ' << i << ' ';
        for (j = i; r[j].h; j = r[j].t) std::cout << r[j].h << ' ';
    }
}
```

This solution should get 100 points.

Problem Grow Gravity

Problem author: Lucian Badea

In this problem, we are given a shape that initially has only one 1×1 square that grows based on two given types of operations. Additionally, after each grow operation, a gravity operation is applied, meaning that each column of squares will fall to the level of the lowest square.

First subtask. In this subtask we have only $*$ operations, meaning that the shape will always be a square, and, if there are x such operations, the side of the square will have length $2x + 1$, so the formula is $(2x + 1)^2$.

Second and third subtasks. In these subtasks we have to analyze the behaviour of the grow operations.

Observation 2. *The gravity operation only affects the first and the last column of the shape.*

Therefore, we can maintain the number of squares on each column and update them accordingly. The total time complexity will be $O(Q \times N^2)$.

Fourth subtask. Here, we can observe that we have a very particular shape, composed of a square, a pyramid on top of it and two additional squares on both sides. Thus, if we have x + operations, the formula is $3x^2 + x + 1$.

Full solution. For the full solution, we have to rely on the following:

Observation 3. *The order of the operations does not matter.*

This can be checked taking an arbitrary shape and performing the operations $+*$ and then $*+$. Thus, we can firstly perform the $*$ operations, that will give us a square and then all the $+$ operations will form 'staircases' on the left and the right of the shape. Therefore, if we have x_+ $+$ operations and x_* $*$ operations, the formula will be $(2x_+ + 2x_* + 1)(2x_* + 1) + 2(2x_* + 1) + (x_+ - 1)(3x_+ + 4x_* + 2)$. Note that if we plug $x_+ = 0$ or $x_* = 0$ the formulas are the same as the ones in subtasks 1 and 4. In order to answer each query, we will maintain a binary indexed tree to find x_+ and x_* in logarithmic time. Therefore, the final complexity will be $O(Q \times \log N)$.

Problem Hearts

Problem author: Iulian Arsenoiu

For simplicity, we will consider that $A_i < B_i$ for all $1 \le i \le n$. In order to find a solution for this problem, it is necessary that we find a better way to interpret $\text{val}(C)$.

Definition 2. We define the *contribution* of an element A_i in query (l, r), denoted as $\text{contrib}(A_i)$ as the number of arrays C such that $C_{i-l} = A_i$ and there is no index j with $l \le j < i$ such that $C_{j-l} > A_i$. We also define the contribution of an element B_i analogously.

Observation 4. $\text{val}(C) = \sum_{i=l}^{r} \text{contrib}(A_i) + \text{contrib}(B_i)$

We are now left with the problem of calculating the contribution of all the elements included in each query. Since calculating the contribution for the values of A_i and B_i is identical, further in this proof we will only focus on calculating $\text{contrib}(A_i)$.

Lemma 1. *Let k be the number of indices j with $l \le j < i$ such that $B_j < A_i$. Then,*

$$\text{contrib}(A_i) = \begin{cases} 0, & \text{there exists } j \text{ with } l \le j < i \text{ and } A_j > A_i, \\ 2^k \times 2^{r-i}, & \text{otherwise.} \end{cases}$$

Proof. Firstly, if there is a j with $l \le j < i$ and $A_j > A_i$, then it doesn't matter whether $C_{j-l} = A_j$ or $C_{j-l} = B_j$, as in both cases $C_{j-l} > A_i$ so A_i has contribution 0. Otherwise, for all k values of j such that $B_j < A_i$, C_{j-l} can be either A_j or B_j, so there are two possible choices. For all the other values of $j < i$, we can only choose $C_{j-l} = A_j$ in order for A_i to be counted towards $\text{val}(C)$ so there is one possible choice. For all values $l \ge j > i$, our choice for C_{j-l} does not matter, so we also have two choices. Thus the total number of arrays is $2^k \times 2^{r-i}$. □

Slow solution We can derive the following solution: we iterate with i from l to r and we calculate the contribution of each A_i according to the formula in Lemma 1. We can either choose to calculate k naively (by iterating through all the possible values of j) or using Binary Indexed Trees. Depending on our approach, we obtain a time complexity of $O(Q \times N^2)$ or $O(Q \times N \times \log N)$.

Fast solution Suppose we iterate the left side of the current interval ℓ, and want to maintain a data structure that is able to answer queries of the form "what is the sum of the contributions of the elements at position at most r?" If we are able to maintain such a data structure then we are done.

First, a small observation: We see that the contribution of an element A_i is either 0 or $2^k \times 2^{r-i}$ for some values of k and i. We can rewrite this as (something) $\times\ 2^r$, where the something is either 0 or 2^{k-i}. We call this value, either 0 or 2^{k-i}, the *effective contribution* of element A_i. The contribution of A_i depended on r — but the effective contribution *does not.* Hence we will want to only compute the sum of the *effective contributions* for all elements in our query interval, and at the end multiply by 2^r.

How do the effective contributions change as we change ℓ? It is easy to see that every time we decrease ℓ, we need to multiply the effective contributions of those elements $< A_\ell$ with 0, and of those elements $> B_\ell$ with 2. Furthermore we need to add in the effective contribution of elements A_ℓ, B_ℓ, which are easy to compute. Hence we need a data structure that has the following characteristics:

1. The data structure maintains position–key–value tuples $\{(p_1, k_1, v_1), \ldots, (p_c, k_c, v_c)\}$.
2. We need the following updates:

   Point update. Add in a new position–key–value tuple (p, k, v).

   Range update. Given (k_0, k_1, m), multiply values of all keys between k_0 and k_1 by m.
3. We have one query: given a position p_0, give the sum of all values with position $p \le p_0$.

Crucially, we know from beforehand the set of position–key values (these are just the positions in A_i, B_i and the values of A_i, B_i respectively). We propose the following data structure: split the *keys* into strips of size $B = \sqrt{N}$. For each strip and each position, precompute how many keys are in that strip up to that position (total $O(N\sqrt{N})$ time). Now, for every strip we maintain the sum of all the values for every prefix of positions (this is total $O(N)$ memory, since each position is only in one strip). How do we deal with the operations?

- For a query, we simply need to walk through all the strips and output the relevant partial sum — we can know which one that is because of the precomputation step we did. This takes $O(\sqrt{N})$ time.
- For a point update, we simply recompute all the partial sums within that strip. This takes $O(\sqrt{N})$ time.
- For a range update, we have two cases. For any strip completely contained within that range, we may simply keep an extra "multiplier" value that we remember to multiply all the partial sums with. Multiplying that special value takes $O(1)$ time, so this part of the update takes $O(\sqrt{N})$ time overall. There is also perhaps one strip that is not completely covered by the range — here we yet again recompute the partial sums by brute force.

The time complexity of this approach is $O((N + Q)\sqrt{N})$; and we note that it is quite fast in practice.

Online solution Although it was not intended for this contest, the queries can be answered online while maintaining the same time complexity of $O((N + Q) \times \sqrt{N})$.

Let us considering splitting the arrays into buckets of size $\sqrt{N}$. Then, there will be a total of $\sqrt{N}$ buckets.

Firstly, we consider the case when l and r belong to the same bucket. To solve such queries, we need to precompute for each bucket the result of the query for any two possible indices that belong to that bucket. We can do this brute-force by fixing the l index and observing that $P(l, i) = P(l, i - 1) \times 2 + \text{contrib}(A_i) + \text{contrib}(B_i)$.

In order to be able to solve each query in $O(\sqrt{N})$ time, we require to split each query into three parts, and for each part we need additional precomputation.

The sum of the contributions of all values in the bucket of l Let l' be the last element in the bucket of l. Then, this value is equal to $P(l, l') \times 2^{r-l'}$. Since l and l' belong to the same bucket, $P(l, l')$ is already precomputed.

The contributions of all values in the bucket of r We need to precompute for each bucket the elements in A_i and B_i that belong to that bucket in increasing order.

We also need to compute for each bucket b and for all values i from 1 to $2 \times N$ the number of occurences of values in B from 1 to i in the buckets from 1 to b. This allows for queries of the form: "How many values B_i lower than x are there in buckets $b_l, b_{l+1} \dots b_r$?" in $O(1)$ per query. This can be precomputed using partial sums.

We also need for any bucket (l, r) to compute the contributions of A_i and B_i with $l \le i \le r$ considering a query from l to i. This can be done brute-force.

Finally, with these precomputations, we are ready to calculate the contributions. We already know for each value its contribution considering the query starts at r' = the first index in the bucket of r, so we can start from there. The contribution of each value also changes according to the number of elements from $l' + 1$ to $r' - 1$ that are lower than it. Since $l' + 1$ and $r' - 1$ are bucket endpoints, this can be calculated in $O(1)$ using our precomputation.

We then need to find a way to modify the contributions according to the values in (l, l'). We will consider the elements in the bucket of l and in the bucket of r in increasing order and use two pointers to adjust the contributions. All indices smaller than l or bigger than r are simply discarded.

The contributions of all values in the interval $(l' + 1, r' - 1)$ This is the tricky part of the problem, and requires a very unusual precomputation: For each pair of buckets, say b_1 and b_2 with $1 < b_1 < b_2$, and for each value $B_i = x$ in bucket $b_1 - 1$, we calculate the sum of the contributions of all values bigger than x that are in the buckets from b_1 to b_2, considering a query from the left bound of b_1 to the right bound of b_2.

The reason for this computation is simple: For a query (l, r) where the bucket of l is $b_1 - 1$ and the bucket of r is $b_2 + 1$, all the values greater than a given value x will change their contribution by the same factor when we consider that the query is, in fact, left bounded by l and not by $l' + 1$ (as the precomputation suggests). Then, in order to actually resolve the query, we can simply iterate through the values in bucket $b_1 - 1$ in increasing order (discarding those before l) and sum the values in the precomputation.

Finally, we need to find a way to actually build this precomputation. In order to do so, we fix b_1 and then iterate with b_2 increasingly. Notably, when transitioning to a new b_2, all the old contributions multiply by a power of 2 (given by the size of the bucket). The new contributions can again be computed by considering the elements of $b_1 - 1$ and b_2 in increasing order and using two pointers.