InfO(1)Cup 2023 Editorial

Problem "Valentine's Day"

Prop: Iulian Arsenoiu

Subtask 1. For this subtask, we need to find the minimum lexicographic perfect permutation. We can easily prove that in order to make our permutations as small (lexicographically) as possible, we can simply start from left to right and place on each position the minimum value that is still available (it was not used already). We can do this using the `set` data structure.

Time complexity: $O(n \log n)$

Subtask 2. For this subtask, we can generate every single permutation in lexicographical order and check if it is perfect. We can do this using backtracking or the `std::next_permutation` function.

Time complexity: $O(n! \times n)$

Subtask 3. For this subtask, the previous solution will not work. However, we can still generate every single *perfect* permutation in lexicographical order. To do this, we can make a function which, given a perfect permutation p, can generate the next perfect permutation in lexicographical order. How do we do this? First, we need to find the smallest suffix of p which needs to be permuted in order to get the next permutation. Let the first position of that suffix be pos. Let S be the set of values p_i so that $i \geq pos$ and Q the set of values a_i so that $i > pos$. To get the next permutation, for the position pos, we will use the smallest value in S which is strictly bigger than p_{pos}. Then, we remove that value from S. These operations can be easily implemented using a `set` data structure. Now, for all positions greater than pos, it will always be optimal to build the suffix using the strategy explained in Subtask 1. However, we won't be able to use this strategy for all suffixes of p, since some suffixes can't be permuted in order to obtain a perfect permutation. A very important observation, which will help us further, is that a suffix can be permuted to make a perfect permutation if and only if for each value x from 1 to n, the frequency of all values greater than or equal to x in S, let us call it frs_{val}, is greater than or equal to the frequency of all values greater than or equal to x in Q, let us call it frq_{val}. In order to test if this property is true for all suffixes, we can make a segment tree for which $SegmentTree_{val} = frs_{val} - frq_{val}$. We can start from the right side of the array and update the segment tree, then check if the suffix can be used to get the next permutation. As long as the minimal value in the segment tree is greater that or equal to 0, the suffix is good, and we can use the strategy described above. Then, we can simply apply this function $k - 1$ times.

Time complexity: $O(nk\log(n))$

Subtask 4. From now on, we need to start thinking about a solution which is not affected by k. We will start from the left side of the array and for each position pos, we will iterate through all values val which were not already used and are greater than or equal to a_{pos}. Then, we will once again build the sets S and Q, but this time S will contain the values which are not yet used in $1, \ldots, pos$. Note that we can simply build them every time from scratch. Now, we want to find out how many permutations exist for which we've already chosen the prefix $1, \ldots, pos - 1$ and $p_{pos} = val$. We will consider the sets as (0-indexed) arrays, sorted in decreasing order. Let ind_i be the biggest index so that $S_{ind_i} \geq Q_i$, then we can use the following formula to find the number of such permutations:

$$cntPerms = \prod_{i=0}^{n-pos-1} ind_i - i + 1$$

Now, if $cntPerms$ is greater than k, then $p_{pos} = val$, otherwise, we subtract $cntPerms$ from k and move on to the next value of val. We will prove for the next subtasks that we will iterate through no more than $n + \log^2 k$ values of val overall, but for now, it is not necessary.

Time complexity: $O(n^2 \log n)$

Subtask 5. For this subtask, we need to make an important observation. Let $frGreater_{val}$ be the frequency of values greater than or equal to val in the array a. We call a fixed value a value val for which $frGreater_{val} = n - val + 1$. The observation is that all fixed values will only contribute with 1 to the product in the formula for *cntPerms*, and all non-fixed values will contribute with at least 2. That means we only need at most $\log k$ non fixed values to exceed the current value of k. Therefore, we can simply build an array *next* so that $next_i$ is the next non-fixed value, starting from i. Then we iterate, exactly how we did in Subtask 4. This time, when we find a fixed value, we can simply assign it using the strategy in Subtask 1. If the value is not fixed, there are two possibilities: if the amount of non fixed values to the right is bigger than $\log k$, we can once again assign the value using the method in Subtask 1, otherwise we can iterate through all non fixed values until we exceed the value of k or reach the end of the array. We can easily see that we only need to iterate at most $\log k$ times through values.

Time complexity: $O(n \log k)$

Subtask 6. Note that the solution below also provides the proof for its complexity. If you are looking for a simpler solution, check the last paragraph. For this subtask, we need to make an almost complete solution of the problem. First of all, we define a bucket as an interval $[x, y]$ for which x is fixed, $y + 1$ is fixed or $y = n$ and there is no value a so that $x \leq a \leq y$, and a is fixed.

Lemma 1. *If we use the strategy described in Subtask 1 to assign the values for the prefix* $1, \ldots, pos$, *as long as there are at least* l *different values* $i_1, \ldots, i_l$ *greater than pos so that* $a_{i_1}, \ldots, a_{i_l}$ *are all in the same bucket, that bucket will contribute with at least* 2^{l-1} *in the formula for finding cntPerms (since all values will contribute with at least* 2 *except for the smallest one, which is fixed).*

We denote the size of a bucket (after choosing a prefix of length *pos* and assigning its values using the strategy in Subtask 1) as the number of indices i for which $i > pos$ and a_i is in that bucket. We can easily observe that for all values which are contained by buckets of size 1, there is only one way of assigning them, so it is redundant to find *cntPerms*.

Let *pos* be the biggest value for which there are at least $\log k$ values which contribute with at least 2 (we say that a value contributes with at least 2 if the size of its bucket is at least 2 and it is not the smallest value left in the bucket). We can observe that, for this value of *pos*, there will be at most $2 \log k$ values contained by buckets of size greater than 1. Now, we will use the strategy in Subtask 1 to assign the values on indices from 1 to *pos*. Then for the indices from $pos + 1$ to n, if a value a_i is contained by a bucket of size 1, then we can once again assign it using the strategy in Subtask 1. Otherwise, we can just use the algorithm in Subtask 4 and iterate through all values *val*, find S and Q, then compute *cntPerms*. Now, what is the complexity of this? For at most $\log k$ indices, we will need to iterate all possible values *val*, which are at most $\log k$. Then, for each such value, we also find S and Q. Please note that we need to use `set` data structures for finding S and Q in order not to add an additional $\log n$ to the complexity.

Time complexity: $O(n \log^2 k)$

Full solution. We need to optimise the solution above. The observation is that we don't need to find S and Q every time. In fact, we observe that buckets don't influence each other, meaning that if a_i is in a bucket, then p_i must be in the same bucket. The proof for this is quite simple: if for some i, p_i is not in the same bucket as a_i, we will be left with too many values in that bucket and won't be able to complete the permutation (note that p_i can't be in a bucket lower than a_i). This also gives us a proof of the lemma used in Subtask 6: assigning all values using the strategy in Subtask 1 always guarantees that p_i will be in the same bucket as a_i. Assigning a value from a certain bucket is like removing that value from the bucket completely, while the new formed bucket still has the initial properties. And since the lemma holds for the initial state, it will therefore hold for any state of the array.

Now, we can normalise the values in the buckets whose sizes are greater than 1 (those through which we will need to iterate, as we explained in Subtask 6) and solve the problem separately for them. This takes around $\log^3 k$ time. All the other values can be assigned using the strategy in Subtask 1, this process taking $n \log n$ time.

Time complexity: $O(n \log n + \log^3 k)$

Finding a simpler solution. Of course, there are other solutions, but most of them don't also provide a very good explanation of their complexity. In fact, we expected the competitors to find solutions with very good complexities, without being able to completely prove them. For example, an other optimisation of the solution in Subtask 4 is the following:

First of all, we will reverse the formula for *cntPerms*, by redefining ind_i to be the smallest index so that $Q_{ind_i} \leq S_i$. Now, the formula becomes:

$$cntPerms = \prod_{i=0}^{n-pos-1} (n - pos - ind_i) - i + 1$$

We observe that the only values which are changed in the new formula for *cntPerms* when selecting a value *val* for position *pos* and moving on to the next position are those between a_{pos} and p_{pos}. Note that all those values are not yet selected (meaning all of them are in S). But those are exactly the values we iterate when finding *val*! Each one of them is decreased by one (this happens because all of them are in the same bucket, as explained above, but understanding this is not necessary for solving the problem this way), meaning that if we have a data structure which can find the product of all its values, decrease one of the values by one and remove a value (for example a *Segment Tree* which makes sure the product does not exceed k or a *Treap*), we can update the value of *cntPerms* in just $\log n$ time. Therefore, the problem is solved, without needing any extra observations. However, the full proof of the complexity is given by the initial solution.

Time complexity: $O(n \log n)$

Problem "Treasure Hunting"

Prop: Alexandru Luchianov

For all subtasks, we will use an additional matrix t, where $t_{i,j}$ is equal to the number of treasures in the cell (i, j).

Subtasks 1 & 2. For these subtasks, all cells are treasure cells. In the initial state, the number of treasure cells reachable from (i, j) is $(n - i + 1)(m - j + 1)$ We will make an additional matrix *sum*, $sum_{i,j}$ is equal to the sum $t_{1,j} + \ldots + t_{i,j}$. For update operations, we can change the value of $t_{x,y}$, then update all values $sum_{x,y}, sum_{x+1,y}, \ldots, sum_{n,y}$ in $O(n)$. For query operations, the result will be equal to the sum $(sum_{n,y} - sum_{x-1,y}) + \ldots + (sum_{n,m} - sum_{x-1,m})$, which we can find in $O(m)$ time.

Time complexity: $O(nm + Q(n + m))$

Subtasks 1,3 & 4. We observe that for these subtasks, the amount of cells reachable from any treasure cell is very small. Therefore, we can use a *fill* algorithm in order to find all cells reachable from a starting cell. For updates, we simply change the value of $t_{i,j}$. For queries (and for finding S), we will find the sum of all cells (i, j) which are reachable from the starting cell (x, y) (using the fill algorithm).

Time complexity: $O(n^2m^2 + Qnm)$

Subtask 5. For this subtask, finding S needs an optimal solution. However, we can solve the operations exactly how we did for Subtask 4. There are many ways of finding S without knowing the full solution for the operations. One of the easiest one of them to understand is the following:

We will calculate a matrix d where $d_{i,j}$ is equal to the number of treasure cells reachable from (i, j). $d_{i,j} = 0$ if (i, j) is a wall. If only one of the cells $(i, j + 1)$ and $(i + 1, j)$ is a treasure cell, then it is easy to see that $d_{i,j}$ is equal to the value of d in that cell. However, if both $(i, j + 1)$ and $(i + 1, j)$ are treasure cells, it gets a bit harder. If we simply add $d_{i,j+1}$ and $d_{i+1,j}$, then we will count some cells twice. In fact, we only need to subtract the number of cells which are reachable from both $(i, j + 1)$ and $(i + 1, j)$, since those are the ones which we've counted twice. But how do we calculate the number of such cells? Let (i', j') be the cell with the smallest coordinates which is reachable from both $(i, j + 1)$ and $(i + 1, j)$ (note that it will have the smallest line and the smallest column among all such cells). Because of the property of the matrix, all cells reachable from both $(i, j+1)$ and $(i+1, j)$ are also reachable from (i', j').

The only thing left to do is to find (i', j'). This can be done by calculating an additional matrix c, which can be calculated as follows:

$$c_{i,j} = \begin{cases} d_{i,j}, & \text{if } (i,j) \text{ is a treasure cell} \\ c_{i,j+1}, & \text{if } (i,j+1) \text{ is a wall} \\ c_{i+1,j}, & \text{if } (i+1,j) \text{ is a wall} \\ c_{i+1,j+1}, & \text{if both } (i,j+1) \text{ and } (i+1,j) \text{ are treasure cells} \end{cases}$$

Now, we can calculate $d_{i,j} = d_{i,j+1} + d_{i+1,j} - c_{i+1,j+1}$. Why does this work? Let's assume we have an area of walls which are connected to each other. Let (x_{max}, y_{max}) be the wall with the biggest coordinates among them. Then, $c_{i,j}$ for all these walls will be equal to $d_{x_{max}+1, y_{max}+1}$. But for any treasure cell (i, j) for which $(i, j+1)$ and $(i+1, j)$ are also treasure cells and $(i+1, j+1)$ is a wall, the cell (i', j') we talked about earlier is exactly $(x_{max}+1, y_{max}+1)$! Therefore, all values of d and c can be calculated using the formulas above. The value of S will be the sum of all values of d.

Time complexity: $O(Qnm)$

Subtask 6. For this subtask, let's say we can build a matrix b of bitmasks, for which the k-th bit in the bitmask $b_{i,j}$ is equal to 1 if the cell $(k/m + 1, k\%m + 1)$ is reachable from (i, j). In other words, we assign each cell (x, y) the $((x-1)m + y - 1)$-th position in the bitmasks. Then, if the cell (i, j) is a wall, $b_{i,j} = 0$, otherwise, $b_{i,j} = b_{i+1,j} \mid b_{i,j+1}$. For finding S, the number of treasure cells reachable from the cell (i, j) is equal to the number of set bits in $b_{i,j}$. For solving the operations, we will first make an other bitmask B. The k-th bit in B is 1 if the cell assigned to that bit is a treasure cell which contains a treasure. For updates, we can simply update B. For queries, the result will be the number of set bits in $(b_{i,j} \,\&\, B)$. But now, how can we maintain all those bitmasks? One way of doing that is using `bitset`. Another way is simply building a vector of integers, in which the first position represents the first 32 bits, the second position represents the next 32 bits and so on.

Note that the complexity of this solution is the same as the one presented in Subtask 4. However, when using bitmask operations, we can process 32 bits at a time, meaning that the constant of the solution is around $1/32$, making it fast enough for this subtask.

Time complexity: $O(n^2m^2 + Qnm)$

Full solution for answering the queries. For the last subtask, we will need a full solution. We can find S exactly like we did for Subtask 5. For the queries, however, we will need an algorithm which takes at most $n + m$ steps per operation.

First of all, for a query on the cell (x, y), let us consider two paths from (x, y) to (n, m):

1. A path which is obtained by a fill algorithm which prioritizes going to the right.
2. A path which is obtained by a fill algorithm which prioritizes going down

We observe that, because of the special property of the maze, every single treasure cell which is inside of the area in the matrix which is delimited by these two paths is reachable from (x, y). Why? If a cell (i, j) is not reachable from (x, y), but it is inside of that area, that means there is at least one path from $(1, 1)$ to (i, j) which does not intersect with any of the two paths. But since the two paths surround that area completely, this is impossible. Therefore, we can just find these two paths, then calculate the number of treasure cells which contain a treasure which are inside of the area delimited by these paths.

To calculate the number, we will make a solution similar to the one in Subtasks 1 and 2. Once again, we will calculate t and sum. For the updates, we can modify t and sum exactly like we did for Subtasks 1 and 2. For the queries, we can find the paths in $O(n + m)$ time. Now, let up_i be the smallest line of a cell on column i which is included in the first path, and $down_i$ be the biggest line of a cell on column i which is included in the second path. Then, the answer to the query is equal to $(sum_{down_y,y} - sum_{up_y-1,y}) + (sum_{down_{y+1},y+1} - sum_{up_{y+1}-1,y+1}) + \ldots + (sum_{down_m,m} - sum_{up_m-1,m})$. This can be calculated in $O(m)$ time.

Time complexity: $O(nm + Q(n + m))$

Problem "Statues"

Prop: Alexandru Luchianov

In the following we will denote $A_n^k = \frac{n!}{(n-k)!}$. It represents the number of ways of selecting k objects from n possible objects without replacements. Note that in order to calculate this modulo M, we will have to use the *modular multiplicative inverse*. Since $A_n^k = (n-k+1)(n-k+2)\dots n$, if there exists a value Mx for which $n-k < Mx \leq n$ then $A_n^k \equiv 0 \pmod{M}$, so we only care about the values of n and k for which there is no such x. In this case, $A_n^k \equiv A_{n \bmod M}^k \equiv (n \bmod M)!(((n-k) \bmod M)!)^{-1} \pmod{M}$. Thus $A_n^k \bmod M$ can be computed immediately from the values of $0!, 1! \dots, (M-1)!$ and $(0!)^{-1}, (1!)^{-1}, \dots, ((M-1)!)^{-1}$, which can be precomputed.

Subtask 1. For this subtask we will generate all permutations of length n and check for each one of them if it respects all restrictions. Note that q is also low for this subtask.

Time complexity: $O(n! \times nq)$

Subtask 2. For this subtask we will define, for each statue i, the a mask $smask_i$, where $smask_{i,j}$ is 1 if statue j must be built before i, or 0 otherwise. We can now observe that statue i can be built if and only if all the statues in $smask_i$ have already been built. Furthermore, these are consecutive bits, so the value $smask_i$ will always be of value $2^x - 1$.

We will solve the subtask using dynamic programming. Let's define dp_{mask} = number of ways to build the statues in the positions of *mask*. Let's fix a bit j which is set in the *mask*. Then j can be built if and only if $smask_j$ is a submask of $mask \oplus j$. Therefore, we will have the following dynamic programming:

$$dp_{mask} = \sum_{j=0}^{n} \begin{cases} dp_{mask \oplus 2^j}, & \text{if the } j\text{-th bit is set and } smask_j \subseteq mask \oplus 2^j \\ 0, & \text{otherwise} \end{cases}$$

Time complexity: $O(2^n n)$

Subtask 3. For this subtask we only have one restriction. The statues in the towns from $y+1$ to n have no restriction, so they can be placed in A_n^{n-y} ways, leaving us with having to place the other y. Now, we can imagine that the statues in the towns from x to y can only be placed in the days from x to y, since the ones after that have already been placed. Therefore, they can be placed in $(y-x+1)!$ ways, while the statues in the towns from 1 to $x-1$ can be placed in $(x-1)!$ ways. In conclusion, the formula for this subtask is

$$\frac{n!}{(n-y)!}(y-x+1)!(x-1)!$$

Time complexity: $O(\log M)$

Subtasks 4 & 5. For the fifth subtask, the restrictions are *disjoint* and *sorted*. Therefore, we can apply the same intuition as we did for the third subtask, while traversing the restrictions in descending order. The answer is given by the formula

$$A_n^{n-y_q}(y_q-x_q+1)!A_{x_q-1}^{x_q-1-y_{q-1}}(y_{q-1}-x_{q-1}+1)!\ldots(y_1-x_1+1)!(x_1-1)!$$

For the fourth subtask, we can observe that for any two restrictions (x_i, n) and (x_j, n) where $x_i < x_j$, they can be broken up into $(x_i, x_j - 1)$ and (x_j, n), therefore obtaining disjoint restrictions and applying the formula above.

Time complexity: $O(q \log q + q \log M)$

Subtask 6. Let $r(i)$ be the rightmost x of any restriction (x_k, y_k) such that $x_k \le i \le y_k$.

Theorem 1. *The solution is equal to* $\prod_{i=1}^{n}(i-r(i)+1)$.

Proof. We will generate the permutation from left to right. At step i we will have a permutation containing all elements $1 \ldots i-1$. When we try to introduce i into our permutation, we can observe that we have to position it to the right of $r(i)-1$ (that being the only constraint). Therefore, we can position the i-th statue in $i-r(i)+1$ towns. □

The only step remaining is computing the values for $r(i)$. In order to do this, we can use a segment tree with range maximum updates. When a restriction appears, we will update the range $[x, y]$ with the value of x (if it is greater than the current value).

Time complexity: $O((n+q)\log n)$

Full solution. We can observe that r has at most q different values. In conclusion, many consecutive positions have the same r value. Let $i, i+1, \ldots i+k$ be such consecutive positions. By the formula above, they will contribute to the answer with

$$(i - r(i) + 1)(i + 1 - r(i) + 1) \ldots (i + k - r(i) + 1) = \frac{(i + k - r(i) + 1)!}{(i - r(i))!}$$

We can now sort the restrictions and calculate the answer for an entire interval with the formula above.

Time complexity: $O(q \log q + q \log M)$

Problem “Game”

Prop: Alexandru Luchianov

We refer the reader to [1] for a good survey paper on this problem.

We will first present some partial solutions.

Let’s ignore the condition that intervals have to be disjoint for now.

It is obvious that we have to select all intervals of length 1. Now, we are able to answer all queries with length at most K. The obvious next step is to also select all the intervals of length $K+1$. Using these intervals we can now also easily answer all queries of length up to $K(K+1)$. By continuing this process we will select all intervals with lengths 1, $K+1$, $K(K+1)+1$, $K(K(K+1)+1)+1$ and so on.

Some more experienced readers may have already realised that the above solution is just a generalisation of a standard problem known as RMQ. Unfortunately, we will have to change our approach entirely to force the intervals to be disjoint. One technique related to RMQ are segment trees so let’s try adapting them to the problem at hand.

For simplicity, we will assume that $K = 2$. We will try to split the original problem $[1, N]$ into two independent halves, $[1, \lfloor N/2 \rfloor]$ and $[\lfloor N/2 \rfloor + 1, N]$. We will solve the queries that fit completely inside one of these halves in a recursive manner, we are left only with the queries that cross the middle point. To solve them we will select all suffixes of $[1, \lfloor N/2 \rfloor]$ and all prefixes of $[\lfloor N/2 \rfloor + 1, N]$. This solution is optimal for $K = 2$, however how can we generalize it for other K’s?

One simple way to generalize is to split the original problem $[1, N]$ into K parts/buckets. We will select all intervals that are a suffix or a prefix of a bucket. How can we now solve a query? If the current query is contained inside a bucket, we solve it recursively. Otherwise, the query can be split into the suffix of a bucket, a few whole buckets, and the suffix of another bucket. Unfortunately, this solution still does not achieve full score since we can still do better.

To realize why this solution is not yet optimal, let’s try to improve our solution for $K = 3$. We can try to select only even-length suffixes and even-length prefixes of the second bucket. If a query requires a prefix from the second bucket, we can just choose the closest even-length prefix and add one additional unit interval to expand it. For example, let’s say that N is 27, the first bucket is $[1, 9]$, the second bucket is $[10, 18]$ and we have to answer the query $[8, 14]$. We will represent the query as a suffix from the first bucket $[8, 9]$, an even-length prefix from the second bucket $[10, 13]$ and a unit interval from the second bucket $[14, 14]$.

We can try to extent this idea to a higher K. Let's split the original problem into K buckets. For the i-th bucket we will select all suffixes of length divisible by i, and all prefixes of length divisible by $k - i + 1$. Let's think about what happens when we receive a query that starts in the i-th bucket and ends in the j-th bucket. The suffix from the i-th bucket will require at most one selected suffix and at most $i - 1$ unit intervals. The prefix from the j-th bucket will require at most one selected prefix at at most $k - j$ unit intervals. We will also require $j - i - 1$ intervals that span the buckets $i + 1, i + 2, \ldots, j - 1$. Adding all of this together we have at most K intervals.

Problem "Sequences"

Prop: Tamio Vesa-Nakajima

Let *smax* be the maximum sum of the values in a.

Subtasks 1, 2, 3 & 4. The length of the array is quite small, giving us the possibility of using a slower solution. For the first 3 subtasks, we could simply select two indices i and j with $i \leq j$, then iterate the interval $[i, j]$ to find the sum of $a_i + a_{i+1} + \ldots + a_j$. Then, all we need to do is check for all values 2^k, with $2^k \leq 10^6$, if they divide the sum or not. To make this solution run faster, it is easy to see why iterating the hole interval $[i, j]$ once more is redundant, since we could simply modify the sum of the interval as we increase j.

Time complexity: $O(n^2 \log vmax)$

Full solution. For the last two subtasks, we need a faster solution. In fact, we need a solution which doesn't check every single possible contiguous subsequence. First of all, let nr_k be the number of all contiguous subsequences whose sum is divisible by 2^k. We observe that if 2^{k+1} divides a number, then 2^k divides that number as well. We are interested in the number of subsequences for which k is the maximum value so that 2^k divides their sum, but we could also rephrase this as: the number of sequences for which 2^k divides their sum but 2^{k+1} doesn't. So, out of all the subsequences whose sum is divisible by 2^k, we remove those whose sum is divisible by 2^{k+1}. Therefore, the result is equal to:

$$S(a_1, \ldots, a_n) = \sum_{k=1}^{\log smax} (nr_k - nr_{k+1}) 2^k$$

Why does the sum end at $\log smax$? Because by definition, $2^{\log smax + 1}$ exceeds the sum of any subsequence in the array. This value is actually quite small, meaning that if we can find nr_k for each value of k in linear time, the problem is solved.

How do we do that? Let $sum_i = (a_1 + a_2 + \ldots a_i) \ \% \ 2^k$. Let s be the sum of values of an interval $[i, j]$. But $s \equiv sum_j - sum_{i-1}$ (modulo 2^k), so if we want s to be divisible by 2^k, meaning that $s \equiv 0$ (modulo 2^k), then sum_j should be equal to sum_{i-1}. Therefore, the problem becomes equivalent to finding the number of pairs of indices $0 \leq i < j \leq n$ for which $sum_i = sum_j$. To do this, we can start from the left side of the array and keep track of the frequency of all values in *sum*. At each step i, we increase the number of sequences with the number of occurences of sum_i which we've found so far. Now, all we have to do is to solve this for all values of k and then use the formula above to get the answer.

Time complexity: $O(n \log smax)$

Problem “Caesar is back”

Prop: Lucian Badea, Tudor Mușat

Subtask 1. For each query, any subsequence included in the interval in the query will be a 1-step transformation. So for each query $[l, r]$ the answer will be the number of subsequences that it completely includes equal to:

$$\frac{(r - l + 1) \times (r - l)}{2}$$

Time complexity: $O(n + q)$

Subtask 3. For each query $[l, r]$ we can iterate through all subsequences completely included in the interval $[l, r]$ and check whether they are a k-step transformation or not.

Time complexity: $O(n + qn^2)$

Subtask 4. This approach no longer works, but we can precalculate for each subsequence whether it's a k-step transformation or not, so we can answer the queries more efficiently.

Time complexity: $O(n + n^2 + q)$

Subtask 5. We can divide the initial strings into disjoint intervals which are k-steps transformation. We will also keep for each of these intervals the number of fully included subsequences, which can be easily calculated like in subtask 1. We know that for any subsequence that is a k-step transformation it must be completely included in one of these intervals.

Therefore, for each query we will have to add the answer of all completely included intervals, and for the partially included ones, which are a maximum number of two, we will calculate their contribution separately. To find the partially included intervals we can maintain the intervals in increasing order and do a binary search.

Time complexity: $O(n + q \log n)$

Subtasks 2 & 6. For the second subtask, n is a 13-step transformation of a and vice versa, which means that we have only two possible values of k, 0 and 13. The idea is exactly the same, but instead of doing a binary search every time, we can calculate at the start, for each index, what interval includes it.

Therefore, we can find the partially included intervals with just one operation.

Now, we can extend this idea for all values of k in order to answer in constant time for each query.

Time complexity: $O(n+q)$

References

[1] S. Raskhodnikova, *Transitive-Closure Spanners: A Survey*, pp. 167–196. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.