



# International Collegiate Programming Contest

Latin American Regional Contests

*October 21, 2023*

Solution Sketches

## Problem A { Analyzing Contracts

*Author*: Agust n Santiago Gutierrez, Argentina

This is a more challenging version of problem D: \Money for Nothing" from the ICPC World Finals 2017. The problem can be found, for example, at this link: `https://vjudge.net/problem/Kattis-money`. The expected solution for this problem has a time complexity of $O(N \log N)$ and uses a divide and conquer technique known as \divide and conquer optimization". This technique is commonly applied within a larger dynamic programming algorithm to speed up the computation of a formula. In the 2017 problem, there is no DP algorithm, and this \divide and conquer optimization" is essentially the whole solution, as detailed in the following link: `https://www.csc.kth.se/~austrin/icpc/finals2017solutions.pdf`. Another solution mentioned involves a more complex geometric sweep line approach that deals with convex polygons \Voronoi style".

The author found no way to solve the proposed problem using either of these two techniques because they are essentially \global" and don't handle queries well. The proposed problem is clearly more challenging because solving an input where all clients are given rst and only after all queries are made would solve the corresponding 2017 problem.

The $N$ suppliers are naturally sorted as given in the input le, which is something that the 2017 solution also uses. Let's de ne, for each client $j$, a function $f_j$ that maps supplier indexes to the corresponding matching value, where $f_j(i) = (C_j - V_i) \times (E_j - S_i + 1)$, representing the pro t from matching client $j$ with supplier $i$ (or 0 if such a pair is not pro table).

The critical property is that when considering any two particular clients $j_1$ and $j_2$, one of two things will happen:

1. One client will \dominate" the other, having greater or equal values for both $C_j$ and $E_j$. In this case, one function $f_j$ is greater or equal to the other function over the entire domain.

2. We have $E_{j_1} < E_{j_2}$ and $C_{j_1} > C_{j_2}$. In this case, \the sign" of the di erence function $f_{j_2} - f_{j_1}$ can never go from positive to negative. In other words, \once $f_{j_2}$ is strictly larger than $f_{j_1}$, it will be larger or equal for the rest of the domain".

A geometric proof, similar to the one used for the 2017 problem, can be provided. Here, $f_j(i)$ is the area of a rectangle with vertices $(S_i, V_i)$ and $(E_j + 1, C_j)$. It can be observed that the di erence function $f_{j_2} - f_{j_1}$ actually increases when moving from $i$ to $i + 1$ if both $f_{j_2}(i)$ and $f_{j_2}(i + 1)$ are greater than 0. If $f_{j_2}$ becomes 0 at some $i$ (no supplier point available below and to the left), after already being positive, then $f_{j_1}$ will also be 0 from that point on.

In both cases, we arrive at the critical property that any two $f_j$ functions \cross" at most once. This is a condition needed for functions to be stored in a Li-Chao Tree, as detailed here: `https://robert1003.github.io/2020/02/06/li-chao-segment-tree.html`.

Thus, the solution is as follows: view each client $j$ as a function to be stored in a Li-Chao Tree over the domain $[1, N]$. Then, the problem queries ask for the maximum value of $f_j(i)$ for a given $i$, considering the functions $f_j$ already stored in the Li-Chao Tree. As long as we can reasonably and quickly compare two functions at a node of the tree to decide \which one wins over which half", we can use the Li-Chao Tree structure in the standard way. Each \add client" query adds a function to the tree, and this can be computed with care, taking into account the 0-valued region of each function $f_j$ in both the lowest and highest values of the domain.

The entire solution has a runtime complexity of $O(N + Q \log N)$ if e ciently implemented, and possibly $O(N + Q \log^2 N)$ if some unnecessary binary search to handle the 0s is performed at each node during the Li-Chao insertion. E cient $O(N + Q \log^2 N)$ implementations were allowed to pass. Implementations with a complexity of $O(N + Q \log N)$ were expected to pass even if not coded as e ciently.

There is also the possiblity of implementing an $O(N+Q\sqrt{N})$ approach by sqrt-decomposition on the queries: If we split the queries into $\frac{Q}{B}$ blocks of size $B$, then after reading each block we can run the global divide and conquer solution for the 2017 version of the problem to compute for each supplier the best client in that block. The complexity of that divide and conquer is

$$O(N + B \log N)$$

if done e ciently, and we have to run it $\frac{Q}{B}$ times. Then when answering each query we only need to explicitly check clients in the current block one by one, as the previous ones were already considered by the runs of the divide and conquer at the end of previous blocks, so that has a total cost of $Q \cdot B$. Adding both costs we get a total of $\frac{Q}{B}(N + B \log N) + Q \cdot B = Q(\frac{N}{B} + \log N + B)$, which choosing $B = \sqrt{N}$ gives an $O(Q\sqrt{N})$ total query complexity. The runtime of this solution varies quite a lot depending on implementation e ciency, and e cient implementations were accepted and even much faster than the tested $O(N + Q \log^2 N)$ solutions.

Please note that the fact that suppliers (while not clients) are already \ ltered" so that there are no \dominated pairs" is a requirement for this solution to work. The well-de ned total order over the abstract domain of indexes is what ultimately allows using a Li-Chao Tree. The situation is unlike in the 2017 problem, which asks for only the global maximum. In this problem, lifting the \sorted" restriction would not allow for the simple removal of dominated pairs as in the 2017 problem because queries might ask about any particular supplier, even those that are dominated by a better supplier.

# Problem B { Blackboard Game

*Author*: Arthur Nascimento, Brasil

Notice that if, at any moment in the middle of the game, it is Carlinhos' turn and his current sum is greater than Equalizer's, he can win by always picking the greatest number from now on. The same applies if his current sum is smaller. So Equalizer is always forced to circle a number that is equal to Carlinhos' last move.

Now, after working out some small cases, we can guess that the winning positions for Equalizer are the ones where the frequency of each value is a multiple of 3:

- If all the frequencies are indeed divisible by 3, no matter Carlinhos' chosen number, Equalizer will be able to circle and erase two occurrences of that value, making it so all frequencies are still divisible by 3. Eventually, the game will end, and the sums will stay equal.

- If not, there will either exist at least 3 values $x$ with $freq[x]\%3 = 1$, or 3 values $y$ with $freq[y]\%3 = 2$, or two values $x$ and $y$ with $freq[x]\%3 = 1$ and $freq[y]\%3 = 2$. In all of these cases, Carlinhos can guarantee that after Equalizer's move, it will still hold that not all frequencies are divisible by 3. In the rst two cases, he can do anything, and in the third case, he should pick $x$. Eventually, there will appear some number $x$ with $freq[x] = 1$, and Carlinhos can win by picking it.

So the solution is just to count the frequency of each value.

# Problem C { Candy Rush

*Author*: Rafael Grandsire & Pedro Paniago, Brasil

### Special case: two candies only

When $k = 2$, the problem can be seen as nding the largest subsegment with the same number of 1s and 2s. The observation for a fast solution is that if we create another sequence $d_i = 2c_i - 3$ (1s are turned into -1 and 2s into 1), one valid subsegment of $\{c_i\}$ is now a sum-0 subsegment of $\{d_i\}$. The largest one can easily be found in $O(n)$ time and memory.

When we compute the pre x sum sequence $p_i = \sum_{1 \le j \le i} d_j$, the sum of a subsegment de ned by $l$ and $r$ is $\sum_{l \le i \le r} d_i = p_r - p_{l-1}$. The above equation only equals 0 when $p_r = p_{l-1}$. At the end, we can compute for each index $r$ the least index $l - 1$ that has the same pre x sum to update our answer.

### $\mathbf{O}(nk \log(n))$ solution

Now, instead of having 1s and -1s, we can have a frequency vector for each element and match two indices when their di erences are multiples of the all-1s vector.

Take $n = 7$, $k = 3$, and $c = \{2, 1, 2, 3, 1, 2, 3\}$. The pre x vectors are:

$$p_0 = \begin{bmatrix}0\\1\\0\end{bmatrix} \; p_1 = \begin{bmatrix}1\\1\\0\end{bmatrix} \; p_2 = \begin{bmatrix}1\\2\\0\end{bmatrix} \; p_3 = \begin{bmatrix}1\\2\\1\end{bmatrix} \; p_4 = \begin{bmatrix}2\\2\\1\end{bmatrix} \; p_5 = \begin{bmatrix}2\\3\\1\end{bmatrix} \; p_6 = \begin{bmatrix}2\\3\\2\end{bmatrix}$$

Notice that the subsegments $[1, 3]$, $[4, 6]$, and $[1, 6]$ all have the same number of elements, and thus, the di erence is a multiple of the all-1s vector. This leads to a solution that uses the pre x vector of frequency as a key.

To check whether the di erence of two vectors is a multiple of the all-1s vector, we need a normalization trick: whenever all the entries are at least one, we decrement 1 from them, making at least one of the entries equal to 0. For the same instance we analyzed before, the pre x vectors we compute are, in fact:

$$p_0 = \begin{bmatrix}0\\1\\0\end{bmatrix} \; p_1 = \begin{bmatrix}1\\1\\0\end{bmatrix} \; p_2 = \begin{bmatrix}1\\2\\0\end{bmatrix} \; p_3 = \begin{bmatrix}0\\1\\0\end{bmatrix} \; p_4 = \begin{bmatrix}1\\1\\0\end{bmatrix} \; p_5 = \begin{bmatrix}1\\2\\0\end{bmatrix} \; p_6 = \begin{bmatrix}0\\1\\0\end{bmatrix}$$

With that, we should check for equality instead of the multiple of all-1s vector property. We can easily detect that the segment $[1, 6]$ is valid because $p_0 = p_6$, the same property we exploited when there were only two candies.

### $\mathbf{O}(n^2 \log(n)/k)$ solution

Even though the last solution can be enough for large $n$ and small $k$, it behaves like a quadratic solution when $k =$ $(n)$. We will now think of a solution that works fast for large $k$.

Intuitively, when $k$ is large, there will be only a small number of multiples of $k$ that can be the answer. In fact, the number of multiples of $k$ less than $n$ is $\lfloor n/k \rfloor$. We can then use a sliding window for each multiple of $k$ and use a data structure to check if all frequencies are the same.

### $\mathbf{O(n\sqrt{n}\log(n))}$ solution

If we combine the solutions with complexity $O(nk\log(n))$ and $O(n^2\log(n)/k)$ and decide to use the faster one depending on the relationship between $n$ and $k$, we have a solution of complexity $O(n\sqrt{n}\log(n))$. The threshold to use the first solution should be $k \leq \sqrt{n}$, and we should use the second one in the other case.

### $\mathbf{O(n\log(n))}$ solution

All previous solutions have super simple and direct implementation and analysis. But an $O(n\log(n))$ solution with hashes also exists. One can use amortized analysis to notice that the normalization step described earlier is O(1) and maintain dynamically the hash of the pref vector. Instead of indexing using the vector as the key, we will index by the hash of the vector and reduce the complexity to look up a similar vector to $O(\log(n))$, instead of $O(k\log(n))$.

The hash we choose is $\sum_{1\leq i\leq k} b^i \cdot \text{freq}[i]$. It is pretty similar to the well-known hash of strings and supports updates on the freq vector easily.

Time limit was set such that well implemented $O(n\sqrt{n}\log(n))$ could also get an AC.

## Problem D { Deciphering WordWhiz

*Author*: Paulo Cezar Pereira Costa, Brasil

One of the ways to approach this problem is simply interating on all words of the dictonary for each guess and counting those that would have the same feedback.

Note that it's guaranteed that valid words do not have repeated letters, this is what allows the game rules to be so succintly described and make it quite easy to calculate the feedback for each word. To calculate the feedback we simply iterate on each letter of the guess word and compare it with the characters of the secret word. Below is a sample python implementation of this approach.

```python
#!/usr/bin/env python3

N = int(input())
dictionary = [input() for _ in range(N)]
secret_word = dictionary[0]
G = int(input())
feedbacks = [input() for _ in range(G)]

def generate_feedback(word, secret_word):
    feedback = ""
    for i in range(5):
        if word[i] == secret_word[i]:
            feedback += "*"
        elif word[i] in secret_word:
            feedback += "!"
        else:
            feedback += "X"
    return feedback

for feedback in feedbacks:
    possible_word_count = 0

    for word in dictionary:
        if generate_feedback(word, secret_word) == feedback:
            possible_word_count += 1

    print(possible_word_count)
```

# Problem E { Elevated Pro ts

$Author$: Arthur Nascimento, Brasil

We will think about the answer backwards, so let's imagine we are removing leaves from the tree and want to maximize $\sum_{i=1}^{N}(N-i+1)\cdot L[i]$, where $L[i]$ is the $i$-th removed leaf.

Let's rst solve the particular case where the tree's root has 2 kids, and both are paths. We will focus on how to build the optimal permutation $L$. If we group vertices that are next to each order in $L$ and belong to the same subtree, we will have a series of alternating color buckets.

If we swap two adjacent buckets $A$ and $B$, that would still produce a valid permutation, and its score in relation to the old one would depend on how the indices of the vertices in the buckets changed: Each element $x$ in $B$ would contribute $|A|\cdot x$ to the di erence, and each element $x$ in $A$ would contribute $-|B|\cdot x$. In the optimal answer, this change should always be zero or negative, so it holds that $|A|\cdot\sum x_b \le |B|\cdot\sum x_a$. So, the buckets must be sorted in non-increasing order by the average value of their indices.

Now we claim that $L$ will begin with the greatest average pre x amongst both arrays. If this was not the case, that would lead to existing some bucket with greater average than the rst bucket, which is a contradiction. So the following algorithm will solve the problem for two arrays:

- For each of the arrays, build its buckets partition: nd its greatest average pre x, delete it, nd the greatest average pre x of the remaining array, delete it, and so on. (we will discuss later how to do this fast)
- Merge the two partitions greedily with two pointers, always picking the one with greater average

If there are ties in the algorithm above (for example, multiple pre xes with maximum average) we can prove that any choice works.

Now, to solve the general case of a tree, we can think that the algorithm above is basically merging two subtrees that are paths into one larger path. But we can turn any tree into a path by repeating this. Just do a dfs, recursively turn your kids into paths, and merge them. So this gives a slow solution for the general case.

(To actually prove the algorithm above works, we need to prove the the optimal way to merge two paths does not depend on the rest of the tree. To this, we can consider the optimal answer up to the end $x$ of those paths, and also group them into buckets by color, but now there will be a third color to indicate vertices outside our two paths. And now we can replicate the exchange argument and average monotonicity from above.)

The key point to optimize the solution is that, when we merge two paths, the buckets partition of the merged set of nodes will be the union of the buckets partition of the paths, so there is no need to recalculate it. So we can use set::set to merge those sets and always maintain them sorted by decreasing average, and use small-to-large technique to do so in $O(N\log^2 N)$.

However, after we have merged the sets from all the kids of some vertex $x$, we have to append $x$ to the end of this path, and this last addition might change a lot our path's buckets partition (imagine, for instance, if $x$ is much bigger than all its children). We can x this fast with a stack-ish approach: let $B$ be our last bucket (that is, the one with the smallest average). If $x <$ average($B$), then our partition will not change, and $x$ will be added in the end, as a single-element bucket.b And if $x >$ average($B$), then $B$ and $x$ will be in the same bucket in the new partition (possibly with some more elements), so we merge them into a single bucket. Now, we compare our new bucket with the second-to-last bucket and keep repeating the process. This runs in amortized complexity.

So we keep merging paths until the whole tree becomes a path, which will be the optimal permutation.

One last observation is that we don't need to find the permutation itself, only its value . But since a bucket $S$ that we create will never be torn apart, all we need to keep in order to run the algorithm and find the final answer is its size, its sum, and the sum $\sum_{i=1}^{N}(|S|-i+1)\cdot S[i]$.

# Problem F { Forward and Backward

*Author*: Alejandro Strejilevich de Loma, Argentina

## Naive algorithm

Let $N_b$ be the representation of $N$ in base $b$.

The naive algorithm is to check whether $N_b$ is a palindrome for each possible $b$. Since there are $O(N)$ allowed values for $b$, while computing and checking $N_b$ can be done in $O(\log N)$, the whole process is $O(N \log N)$.

The running time can be improved by adding some math. The more math we add, the better algorithm we get.

## Improved algorithm

Notice that the statement bans the values $b > N$ because $N_b$ would be a single-digit palindrome for any of those infinitely many cases. Regarding $b = N$, it is easy to see that $N_N = 10_N$ is not a palindrome, while for $b = N-1 \geq 2$ we have $N_{N-1} = 11_{N-1}$ which is a palindrome.

Once those high values for $b$ have been worked out, it can be proved that $N_b$ is not a palindrome for any $b \in [N/2, N-2]$ (because we would have $N_b = 1d_b$ with the digit $d_b \neq 1$). A simple observation that can be used to speed up the algorithm a little further is that $N$ cannot be a multiple of $b$ (because the last digit of $N_b$ would be 0).

With the above ideas we can halve the running time of the naive algorithm. To summarize, it is enough to check the values $b \in [2, N/2)$ that do not divide $N$, including $b = N-1 \geq 2$ in the final output.

## Intended algorithm

It is possible to extend our analysis of the interval $[N/2, N)$ to any interval of the form $[N/(d+1), N/d)$. A simple way to achieve this is to split the possible palindromes in two groups: one group having two-digits palindromes, and the other group having palindromes of at least three digits.

$\geq 3$: The "smallest" palindrome in this group is $101_b$. This means that we must have $N \geq b^2 + 1$, which is equivalent to $b \leq \sqrt{N-1}$

Thus, to obtain the bases in this group of palindromes it is enough to apply the naive algorithm for $b$ in the interval $[2, \sqrt{N-1}]$.

Since $O(\sqrt{N})$ values of $b$ are analyzed, and each of them requires $O(\log N)$ operations, the total complexity for this group of palindromes is $O(\sqrt{N} \log N)$.

Note that we can still speed up the process by discarding the values of $b$ such that $N$ is a multiple of $b$.

2: In this case we have $N_b = dd_b$, where $d_b$ is a digit in base $b$, that is, $1 \leq d \leq b-1$ ($d = 0$ is not allowed in this context). This means that $N = db + d = d(b+1)$, which is equivalent to $b = N/d - 1$. Since $d \leq b-1$ we conclude $d \leq N/d - 2$, which is equivalent to $d^2 + 2d - N \leq 0$. The roots of the smiling parabola $f(d) = d^2 + 2d - N$ are $-1 \pm \sqrt{1+N}$, and then we get the restriction $d \leq -1 + \sqrt{1+N}$.

Thus, to obtain the bases of this group of palindromes in ascending order, it is enough to iterate $d$ backwards in the interval $[1, -1 + \sqrt{1 + N}]$. If for any of these values, $N$ is a multiple of $d$, then we know that $N_b$ is the two-digits palindrome $dd_b$ with $b = N/d - 1$.

Since $O(\sqrt{N})$ values of $d$ are analyzed, and each of them requires $O(1)$ operations, the total complexity for this group of palindromes is $O(\sqrt{N})$.

As it can be seen, the two groups of palindromes can be handled in $O(\sqrt{N} \log N)$ time.

# Problem G { GPS on a Flat Earth

*Author*: Mario da Silva, Brasil

Just keep track of the current set of possible locations. This may be one or two points, or a couple line segments.

Implementing this in a bug-free manner seems to be the main challenge here. In the context of Manhattan distance, the coverage area resembles a diamond, essentially a square rotated by 45 degrees. If we rotate the coordinate system by 45 degrees, the diamond becomes a square, which is easier to work with.

Then, instead of worrying about the entire square at once, we can look only at the segments that it's comprised of, for which calculating the intersection should be a lot easier. The only case work needed in this scenario is whether the segments share the same orientation or not as that de nes how their intersection (if any) will look like. Once we are done handling all towers we can simply iterate on the points in each segment, converting them back into the original coordinate system, sort all of the visited points and print the result. Below is a possible way to implement all of this.

```
struct segment
{
        const bool asc;
        const int b, e, h;
        segment(bool a = 0, int b = 0, int e = 0, int h = 0)
                : asc(a), b(b), e(e), h(h) {}

        bool isempty() const { return b == e; }

        bool inbetween(int p) const { return b <= p && p < e; }

        segment intersect(const segment& s) const {
                if (isempty() || s.isempty()) return segment();

                if (asc == s.asc) {
                        if (h != s.h) return segment();

                        int nb = max(b, s.b);
                        int ne = min(e, s.e);
                        ne = max(ne, nb);
                        return segment(asc, nb, ne, h);
                } else {
                        if (inbetween(s.h) && s.inbetween(h)) {
                                return segment(asc, s.h, s.h+1, h);
                        } else return segment();
                }
        }

        vector<pii> traverse() const {
                if (isempty()) return {};

                vector<pii> ret;
```

```
33
34                  for (int p = b; p < e; p++) {
35                          int x2 = p - h;
36                          int y2 = p + h;
37                          if (!asc) x2 = -x2;
38
39                          if (!(x2&1) || !(y2&1))
40                                  ret.pb(pii(x2/2, y2/2));
41                  }
42
43                  return ret;
44          }
45  };
46
47  vector<segment> interpret(int x, int y, int d)
48  {
49          int cx = x+y;
50          int cy = y-x;
51
52          if (d == 0) {
53                  return {
54                          segment(1, cx, cx+1, cy)
55                  };
56          }
57
58          return {
59                  segment(1, cx-d, cx+d+1, cy-d),
60                  segment(1, cx-d, cx+d+1, cy+d),
61                  segment(0, cy-d+1, cy+d, cx-d),
62                  segment(0, cy-d+1, cy+d, cx+d)
63          };
64  }
65
66  vector<segment> intersect(const vector<segment>& a, const vector<segment>& b)
67  {
68          vector<segment> result;
69
70          for (auto sa: a) {
71                  for (auto sb: b) {
72                          segment c = sa.intersect(sb);
73                          if (!c.isempty()) result.push_back(c);
74                  }
75          }
76
77          return result;
78  }
```

# Problem H { Health in Hazard

*Author*: Victor de Sousa Lamarca, Brasil

This problem can be solved using binary search along with the half-plane intersection algorithm. The half-plane intersection problem is well-known in ICPC, although it may not be very common (it's rarely seen in Latin America stages, but it has been encountered in UNICAMP summer camps). There are tutorials available on the cp-algorithms website that explain the half-plane intersection algorithm in detail.

More speci cally, each given line partitions the space into two parts, and we select the half-plane that contains the origin. We use binary search on the prediction. For prediction $i$, we consider the rst $i$ half-planes and check if the area forms a nite polygon. If it does we look at the distances from the polygon vertices to the origin, if all of them are lower than $D$, then the answer is to the left of $i$ (including $i$); if not, the answer is to the right.

Tutorials for the half-plane intersection algorithm typically list the segments that belong to the intersection or calculate the area without handling the case for in nite area. However, it is certainly possible to check for the case of an in nite area. The complexity of calculating the half-plane for $n$ lines is O($n \log n$). When performing the binary search, the nal complexity is O($n \log^2 n$).

Alternatively, one can completely solve the problem without half-plane intersection by reducing the problem to checking if a list of circular arcs span the whole circle. Like before, binary search and consider the rst $i$ half-planes. Intersect each of those lines with the circle of radius $D$ to get an open circular arc of unreachable positions (if there is an intersection with the circle at all for that halfplane). Then we just need to check if those arcs span the whole circle. For that, the intersection points can be sorted by angle and the circle \unrolled" so that points are now in $[0, 2\pi]$ number line and arcs are normal intervals (arcs crossing the 0 = $2\pi$ \cutpoint" can be split into two intervals). Then for each interval there are two events, a +1 event at its left end and a $-1$ event at its right end. We can sort and process events and check that current count is never zero to check if full circle is covered.

# Problem I { Inversions

*Author*: Gabriel Poesia, Brasil

The classical algorithm based on merge-sort to count inversions in $O(nlogn)$ doesnt work since we can't even build the string explicitly ($n$ would go up to $10^{17}$). However, here the number of distinct elements in the string is small (up to 26), and this can be exploited.

Let $f(s)$ be the number of inversions in a string $s$. Given two strings $s_1$ and $s_2$, we have $f(s_1 + s_2) = f(s_1) + f(s_2) + g(s_1, s_2)$ where $g$ is the number of inversions that will occur due to characters in $s_2$ being smaller (in alphabetical order) than characters from $s_1$.

We can easily calculate $f(s)$ for our input string in $O(26 \times |s|)$, initialize an array $cnts$ with zeroes then for each character $c$ in $s$ add the current counts of characters greater than $c$ to the number of inversions and increment the count for $c$.

We can then multiply that number of inversions by $N$ and all that is left to compute are the intersections of type $g$. In order to do that we can reuse the $cnts$ array we just calculated. We know that we need to add $g(S, S)$ to the total number of inversions $N * (N - 1)/2$ times, and to calculate $g$ we can just iterate on all $(c_1, c_2)$ pairs of letters where $c_1 < c_2$ and multiply their $cnts$. Below is a sample implementation of this approach in Python:

```python
#!/usr/bin/env python3

MOD = 10**9 + 7

S = input()
N = int(input())

cnts = [0] * 26

inversions = 0
for c in S:
    idx = ord(c) - ord('a')
    inversions += sum(cnts[idx+1:])
    cnts[idx] += 1

inversions *= N

g = 0
for c_1 in range(26):
    for c_2 in range(c_1+1, 26):
        g += cnts[c_1] * cnts[c_2]
```

```
22
23 inversions += g * N * (N - 1) // 2
24
25 print(inversions % MOD)
```

Note that in C++ one would have to be more careful to avoid over ows. The modulo operation could not be simply excuted at the end but also on the intermediate calculations.

# Problem J { Journey of the Robber

*Author*: Giovanna Kobus Conrado, Brasil

We can use centroid decomposition to solve the problem in O($n \log^2 n$), by building the centroid tree and storing the distance for every node in the subtree. After this list has been sorted by node index, pre x minimum distance can be calculated and the closest node smaller than a certain index can be calculated via binary search.

There's also a O($n\sqrt{n}$) solutions based on sqrt decompostion. From higher to lower identi ers you activate blocks of size sqrt, then you wanna get the answer for the next block. For that do a multi source bfs from each of the already activated vertices. This will give some candidate for the answer to each of the vertices of the current block. Besides that, for a certain block, you also need to check the distance for each pair of the vertices for another candidate answer which can be done using LCA.

# Problem K { Keen on Order

*Author*: Arthur Nascimento, Brasil

If $K \geq 25$, we can always nd one permutation by following this strategy:

Pick the number $x$ such that its rst occurrence in V is the greatest possible. Then pick the number $y \neq x$ such that its rst occurrence, only considering occurrences after $x$, is the greatest possible. And so on. If at any time there is some unpicked number with no such rst occurrence, we have found a valid answer. But there are at least $K$ numbers up to $x$. There are at least $K - 1$ numbers after $x$ and up to $y$, and so on. Since 1 + 2 + ... + 25 $>$ 300, we will always nd one answer.

Now, if $K \leq 24$, we can use bitmask dynamic programming:

Given a subset mask of the values, dp[$mask$] will return the maximum value of the minimal ending of an occurrence of a permutation as a subsequence of V, over all permutations of the elements in mask. If there is one such permutation that is not a subsequence of V, dp[$mask$] = $\infty$. So,

$$\text{dp}[mask] = \max_{x \in \text{mask}} \text{next}(x, \text{dp}[mask - x])$$

Where next($A, i$) is the rst index $j > i$ such that $V[j] = A$. So we can use the DP to construct a valid answer in $O(K \cdot 2^K)$.

# Problem L { Latam++

*Author*: Agust n Santiago Gutierrez, Argentina

A contestant might get the impression that this is a parsers or language-theoretic problem and avoid it, but the problem is better solved in a relatively intuitive way without involving formal grammars. The rst observation is that since variable names and even which variables are equal to others is completely irrelevant, we can imagine that \a" is the only letter for ease of explanation, as replacing all letters with \a" does not change the answer.

Similarly, we can imagine that \+" is the only operator, as we can replace all others with \+" without changing the answer, because the validity of an expression does not care about its meaning or the nature of its operators (as crucially, they are all binary operators).

Now, our string consists of only four characters: \a", \+", \(", and \)". By thinking a bit about what is a valid expression, one can discover that a non-empty string is a valid expression if and only if:

1. The \(" and \)" are \correctly-parenthesized"
2. The string does not contain any of these particular pairs of consecutive characters:
   - `()`
   - `(+`
   - `+)`
   - `++`
   - `)(`
   - `)a`
   - `a(`
3. The string does not start or end with \+"

For checking the rst condition, we can de ne the balance of a string as the number of \(" characters minus the number of \)" characters. A string is \correctly-parenthesized" if and only if it has zero balance, and all of its pre xes have non-negative balance.

Now we have to count the number of substrings that have all of these properties. We can easily make one $O(N)$ pass through the input string and identify all forbidden pairs that occur. Then we know that valid substrings will never span any of those pairs.

We can then divide our problem into independent substrings that no longer contain forbidden pairs. For example, for the string \aaaaaa)(bbb++bbb", there are two forbidden pairs, and we can split it into \aaaaaa)", \(bbb+", and \+bbb". The nal answer is the sum of the answers for each part.

Then for each part, we just need to focus on counting the number of substrings that are correctly parenthesized and start and end at valid characters (di erent to \+"), so we can think that each character is either -1 (closing parenthesis), 0 (\+" or \a"), or 1 (opening parenthesis) as explained before.

The author sees two main ways to count this:

1. Divide and conquer: if we split at the middle, we can independently count substrings of the left and right parts, and then we are only missing substrings that span the cut, so that they contain both the last character of the rst part and the rst character of the second part.

   We can count those in $O(N)$: they are formed by a \good" pre x of the second part and a \good" su x of the rst part having opposite balance so that the sum is zero. A good su x of the rst part is one having greater or equal balance than all its proper su xes.

Analogously, a good pre x of the second part is one having lower or equal balance than all its proper pre xes. So if we iterate and keep count for each possible balance k how many good pre xes of the second part and good su xes of the rst part have balance $k$, then the total is $\sum_k$ left($k$) $\cdot$ right($-k$).

This has complexity $O(NlgN)$ as it has the same recursion formula as the well-known Merge Sort.

2. Brute force the last character of the substring, iterating left to right. If we have already precomputed the balance of all pre xes of the main string, then possible starting positions must be pre xes having the same balance as the pre x ending at this last character. From those, only those that do not cross a pre x having strictly less balance can be used. With a data structure for RMQ and binary search, as well as keeping the indices of pre xes having balance k for each k, we can actually count how many starting positions will work in logarithmic time, for a total $O(NlgN)$ solution.

# Problem M { Meeting Point

*Author*: Daniel Bossle, Brasil

This problem can be easily solved in O($M \log N$) with Dijkstra's shortest path algorithm.

1. Run Dijkstra to nd the distance from $P$ to every node.
2. Remove $G$ from the graph and run Dijkstra again. A simple way to do achieve this is implementing Dijkstra in a separate function that takes a \skip" parameter indicating a vertex that should not be added to the queue.
3. Vertices $V$ with dist($V$) = 2 $\times$ dist($G$) on the rst pass, and higher dist on the second pass are all viable options for the misleading meeting point.