Div2-250 Alpha

Given a string with lowercase English characters, find how long prefix matches the English alphabet. For example, given *“abctyf”*, return 3 because *“abc”* is also a prefix of English alphabet.

One naive solution is to copy the whole alphabet from the statement (*“abcdefghijklmnopqrstuvwxyz”*), save it as a string, and use a function to get a substring [0..i] of a string in order to compare two prefixes. It would be O(N^2) which is very fast for small constraints but we can do better. For every index i, we just need to compare the i-th character of the input with the i-th letter of the alphabet in O(N) total.

```
public int maxPref(String s) {
    String good = "abcdefghijklmnopqrstuvwxyz";
    for(int i = 0; i < s.length(); ++i) {
        if(s.charAt(i) != good.charAt(i)) {
            return i;
        }
    }
    return s.length();
}
```

But we can use the fact that we don’t just match two given strings. We compare one string with the English alphabet and it’s very easy to obtain the i-th character of the alphabet — just add i to **‘a’**. For example, **‘a’ + 2** will give you **‘c’**.

```
public int maxPref(String s) {
    for(int i = 0; i < s.length(); ++i) {
        if(s.charAt(i) != 'a' + i) {
            return i;
        }
    }
    return s.length();
}
```

Div2-500 BinaryDistance

In a big binary tree, we need to find a vertex that is farthest from the given vertex V. The tree is very regular: it's a perfect binary tree without some last (rightmost) leaves. Formally, we just say that for every vertex x from 2 to N, there is an edge between x and x/2.

It's optimal to go up from vertex V to the root, then down to the other branch (if we entered the root from the left child, then go right, otherwise left). We need to compute the deepest vertex in that "other branch". It will be either vertex N (which is the rightmost deepest leaf) or the first (leftmost) leaf in that branch. For example, for N=7, the interesting leaves are 4 and 6. Everything else is redundant. More details below.

If the tree wasn't binary (just any tree), we could use DFS or BFS to find distances from V to all other vertices. We can't do that here because the tree is too big and O(N) is too slow.

For any tree, the farthest vertex from a fixed vertex V must be a leaf. If the tree is a binary tree, every path starting from vertex V looks like this: keep going up for some time, then down to the other branch (if you came from a left child to a parent, then go to a right child, otherwise the opposite) and keep going down. When we start going down, the number of steps down we can make is equal to the depth of a subtree. Since the given binary tree is so regular, **it's always optimal to first go up to the root** (instead of stopping earlier in order to go down) because our path keeps getting longer. Formally, we need to say: if vertex x has two children c1 and c2, then the depth of a subtree of c1 is smaller than 2 plus the depth of a subtree of c2. Then instead of going down from c1, it is better to go up to x and then down to c2.

Let's repeat the important observation: starting from V, it's optimal to go up to the root, then down to the other branch (if we entered the root from the right child then go left, otherwise right). We just need to compute the distance from V to the root (which is just the depth of V in a tree) and then the depth of the other branch (the other child of the root).

The depth of some vertex can be computed by dividing it repeatedly by 2 (that is, **x /= 2**) till we get the root. Every step here means going to a parent and we know its index is x/2. Theoretically, we can use logarithm function to figure out the depth of x, but it's a bad idea because the logarithm is computed with floating values. A precision error can produce an answer different by 1.

We can already solve the problem for a perfect binary tree: the answer is the depth of V plus the depth of any leaf (in particular, vertex N is a leaf). The given tree isn't perfect though — some rightmost leaves are missing. If the vertex N is in a left branch (the subtree of vertex 2), then there are no leaves in the right branch (the subtree of vertex 3).

One solution is that we increase N to the next value of form 2^k - 1. That gives us a perfect binary tree, we can compute the answer, and then think about whether the answer should be decreased by 1 because of missing rightmost leaves. If V is in a left branch (which means using the deepest leaf in the right branch), and also original vertex N is in a left branch, leaves in the right branch are one less level deep, so we decrease the answer by 1.

The other solution is to directly compute the depth of the deepest leaf in "the other branch" (not the one in which vertex V is). Both solutions are O(log(N)).

```
int depth(int v) {
    int count = 0;
    while(v >= 2) {
```

```
        count++;
        v /= 2;
    }
    return count;
}
boolean is_left(int v) {
    if(v < 2) {
        throw new IllegalArgumentException("bad v");
    }
    while(v >= 4) {
        v /= 2;
    }
    return v == 2; // else v == 3
}
public int maxDist(int N, int V) {
    if(V == 1) {
        return depth(N);
    }
    if(is_left(V) && is_left(N)) {
        return depth(V) + depth(N) - 1;
    }
    return depth(V) + depth(N);
}
```

Div2-1000 MeanMedianCount

There are N subjects and your default grade in each of them is 0, the maximum possible grade is 10. Count the number of sequence of N grades such the mean of grades at least **needMean** and the median of grades at least **needMedian**, modulo 1e9+7. Input constraints are quite small.

Refer to Div1-250 MeanMedian for some more explanation of the required observations.

With small constraints and being asked to count sequences that satisfy something, you should definitely consider dynamic programming. It's hard to keep track of the median, so we need an equivalent condition.

The median is at least X if and only if there are at least (N+1)/2 numbers that are X or higher, because that would mean that after sorting the sequence the rightmost (N+1)/2 elements are at least X and thus also the middle element is X or higher.

In order to avoid floating values, the mean requirement can be changed to: the sum of all N numbers must be at least N*__needMean__.

We can choose sequence elements one by one and keep track of the number of elements "that are **needMedian** or higher" and the sum of elements. This can be done with dynamic programming like dp[prefix][countBig][sum]. The number of states O(N * N * SUM) = O(N * N * (N * 11)) = O(N^3 * G) where G=11 which can be skipped in complexity if we treat it as a constant. From each state, you should consider each of 11 possible grades and the time complexity is then O(N^3 * G^2).

```
public int getCount(int N, int needMean, int needMedian) {
    int needSum = needMean * N;
    int needBig = (N + 1) / 2; // those with value at least needMedian
    long[][] dp = new long[needSum+1][needBig+1];
    dp[0][0] = 1;
    for(int subject = 0; subject < N; ++subject) {
        long[][] new_dp = new long[needSum+1][needBig+1];
        for(int grade = 0; grade <= 10; ++grade) {
            for(int sum = 0; sum <= needSum; ++sum) {
                for(int big = 0; big <= needBig; ++big) {
                    new_dp[min(needSum, sum + grade)][min(needBig, big + (grade >=
needMedian ? 1 : 0))] += dp[sum][big];
                }
            }
        }
        for(int sum = 0; sum <= needSum; ++sum) {
            for(int big = 0; big <= needBig; ++big) {
                dp[sum][big] = new_dp[sum][big] % MOD;
            }
        }
    }
    return (int) dp[needSum][needBig];
}
```

Div1-250 MeanMedian

There are N subjects and your default grade in each of them is 0, the maximum possible grade is 10. You can study for **d**[i] days in order to increase the i-th grade by 1. What is the minimum possible number of study days in order to get the mean of grades at least **needMean** and the median of grades at least **needMedian**? Input constraints are quite small.

It's better to spend time on subjects that require low effort. Let's sort the given cost sequence d[n]. For example, this would give us {20, 25, 30} for the first example test. Then, the sequence of grades will be decreasing (well, non-increasing).

In order to satisfy the median constraint, the least we can do is to get grade **needMedian** in exactly (N+1)/2 subjects. For median 4 that would imply grades (4, 4, 0) and the cost of 4*20+4*25=180 for the example test. If the mean is small, that's an optimal solution. Indeed, that's the case for the first example test where the mean (4+4+0)/3 is not smaller than the required **needMean** of 2.

If we don't have enough mean of grades yet, it just means we must increase the sum of grades because the equivalent condition is: the sum of grades is at least **needMean***N.

We shouldn't decrease any grades because we anyway need at least (N+1)/2 grades **needMedian** or higher. Instead, we should just increase low-effort subjects if necessary. The intended solution iterates over subjects (from lowest cost **d**[i]) and increases the grade by 1 as long as it's necessary and we can't exceed 10.

We sort in O(N*log(N)) and then can either do 10 increments for each subject or just use a formula like `increase=min(10-currentGrade, needMean*10-currentSumOfGrades)`. The complexity is O(N*log(N)).

One alternative approach is dynamic programming like in knapsack. Since we have requirements about the sum of grades and the number of grades not smaller than **needMedian**, these two values should be dimensions of dp state. See MeanMedianCount for more detail ~~(well, editorial will be posted later)~~.

```
public int minEffort(int needMean, int needMedian, int[] d) {
    int n = d.length;
    int already_sum = (n + 1) / 2 * needMedian;
    int need_sum = needMean * n;
    int answer = 0;
    sort(d);
    for(int i = 0; i < n; ++i) {
        for(int grade = 1; grade <= 10; ++grade) {
            if(i <= n / 2 && grade <= needMedian) { // satisfy the median condition
                answer += d[i];
            }
            else if(already_sum < need_sum) { // increase the sum of grades by 1
                already_sum++;
                answer += d[i];
            }
        }
    }
    return answer;
}
```

Div1-500 PBG

There is a tournament with P polar, B brown and G grizzly bears (P, B, G <= 2000). Two random bears are chosen, they fight and one of them is eliminated, and the process continues till one bear remains. When two bears of the same species are chosen, the winner of this fight is chosen randomly (50-50 chance), otherwise a stronger species survives: grizzly > polar > brown. Find the EV (expected value) of place of Limak who is one of polar bears. Return the answer modulo 10^9+7.

Small constraints and computing EV should give you a hint that dynamic programming can be useful. Unfortunately, we can't have three dimensions because that would be too slow. But it is a working solution: you can define that `dp[P][B][G]` as p-bility of getting to situation with exactly this many bears of each species and use top-down order, or alternatively define `dp[P][B][G]` as the answer (the EV of Limak's place) and then move bottom-up.

The problem would be easier if Limak was one of grizzly (or brown) bears because we can then combine the other two species into one – we don't have to distinguish between them and we just have `dp[grizzly][others]` and this is O(N^2) solution.

Here comes the "Aha!" moment. We can compute the EV of place of a grizzly bear, the EV of a place of a brown bear, and from these two values compute the EV of place of a polar bear. How? Well, the sum of places of all N=P+B+G bears needs to be 1+2+...+N, so the EV of place of a bear is that sum divided by N. Equivalently, you can compute the EV of sum of places of all polar bears and then divide the result by P.

The complexity is O(G*(P+B) + B*(P+G)) = O(N^2) where N=P+B+G.

If you don't understand the details of this solution, first solve the problem Sushi (J) from Atcoder DP contes: https://atcoder.jp/contests/dp/tasks/dp_j.

```
int strong_ev(int STRONG, int WEAK) {
    int[] inv_pairs = new int[STRONG + WEAK + 1];
    for(int i = 0; i <= STRONG + WEAK; ++i) {
        inv_pairs[i] = my_inv(i * (i - 1) / 2); // inverse modulo 1e9+7
    }
    long[][] dp = new long[STRONG+1][WEAK+1];
    dp[STRONG][WEAK] = 1;
    int answer = STRONG + WEAK + 1; // default place
    for(int strong = STRONG; strong >= 0; --strong) {
        int p_strong_lives = mul(strong, my_inv(STRONG)); // division by 0 is ok
        for(int weak = WEAK; weak >= 0; --weak) {
            int here = (int)(dp[strong][weak] % MOD);
            answer = (answer - mul(here, p_strong_lives)) % MOD;
            final int c = mul(here, inv_pairs[strong + weak]);
            if(strong >= 2) {
                dp[strong-1][weak] += mul(c,strong * (strong - 1) / 2);
            }
            if(weak >= 2) {
                dp[strong][weak-1] += mul(c, weak * (weak - 1) / 2);
            }
            if(strong >= 1 && weak >= 1) {
```

```
                dp[strong][weak-1] += mul(c, strong * weak);
            }
        }
    }
    return (answer + MOD) % MOD;
}
public int findEV(int P, int B, int G) {
    // grizzly > polar > brown
    int g = strong_ev(G, P + B);
    int pg = strong_ev(P + G, B);
    // g * G + p * P = pg * (P + G)
    // p = (pg * (P + G) - g * G) / P
    int tmp = mul(pg, P + G) - mul(g, G);
    tmp = (tmp + MOD) % MOD;
    return mul(tmp, my_inv(P));
}
```

Div1-1050 LShape

We are given N points, no two share x-coordinate or y-coordinate. Both N and x[i], y[i] are up to 3000. We are asked to compute the sum over L-scores of each triple of points. The L-score is the minimum total change of coordinates in order to get three points that resemble a letter L, possibly rotated. That is, we should be able to reorder points as (x1, y1), (x2, y2), (x3, y3) such that x1=x2 and y2=y3. Other equalities are allowed but not necessary.

It's sometimes a good idea to get rid of one dimension. For example, in a problem of computing the sum of squared distances of all pairs in given N points – it turns out that the formula for squared distance can be split into the sum for x-coordinate and y-coordinate and we just need to compute something independently for each dimension.

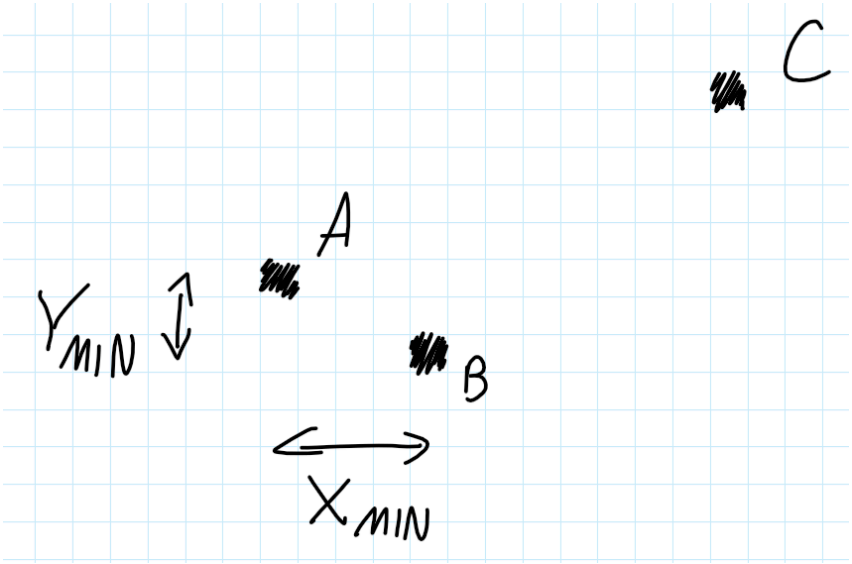
In our problem, it's almost correct to say that the score of a triple is:

```
min(|x1-x2|, |x1-x3|, |x2-x3|) + min(|y1-y2|, |y1-y3|, |y2-y3|)
```

This formula means that for each dimension we are looking for two closest points among the triple. It isn't correct if some two points are close to each other in both dimensions. For example, if points are (0, 0), (2, 2), (3, 3), then one move is enough to get two points with same x-coordinate, also one move is enough to get two points with y-coordinate, but the answer isn't just 1+1.

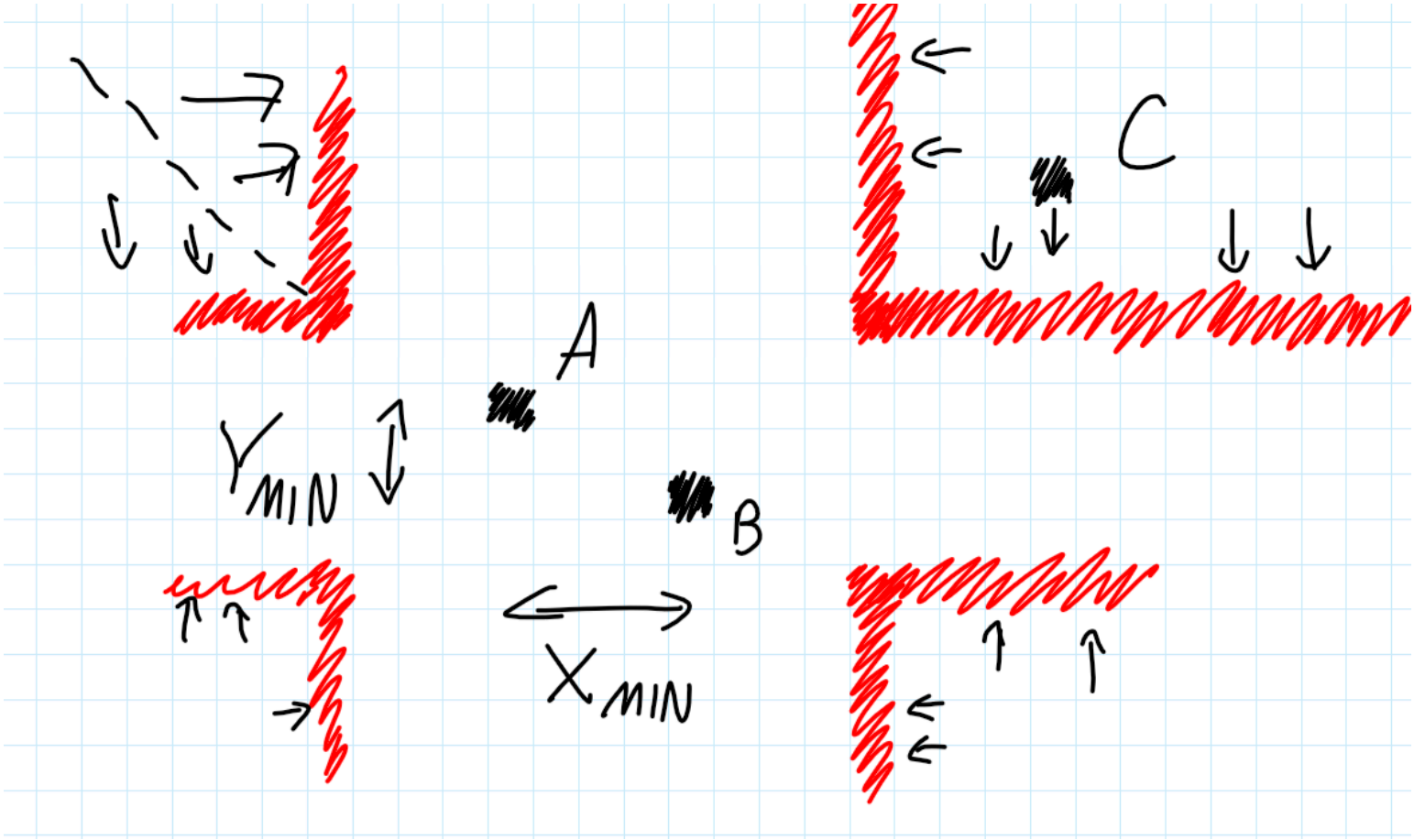
But if the formula was correct (BUT IT ISN'T!) then it isn't hard to solve the problem. For each dimension separately: sort all coordinates (say, x[0] through x[n-1]), then in some smart way find the sum over `min(x[j] - x[i], x[k] - x[j])` for triples of indices `i < j < k`. To do it fast, iterate over pairs `(i, j)` and find the number of indices k for which `x[j] - x[i]` would be the minimum in the triple. This just means asking how many indices k satisfy `x[k] - x[j] > x[j] - x[i]`. You can use binary search to get `O(log(N))` (it was allowed here) or preprocessing to count values bigger than something in `O(1)`.

Now, let's visualize a bad case – a situation when the formula doesn't work:



As you can see, I think about the given points as cells of a grid. Here, two points A and B are very close to each other in both dimensions and then the score of (A, B, C) isn't just X_MIN + Y_MIN. So, by how much is the previous formula wrong? If we can compute that quickly for all triples, we will increase the answer and we're done.

For the situation shown in the drawing, the formula is wrong by `min((C.x - B.x) - (B.x-A.x), (C.y-A.y)-(A.y-B.y))`. This looks scary for now. Think about it as shifting point C in one dimension in such a way that the difference in x-coordinate B-A and C-B is the same, or the difference in y-coordinate is the same. Let's visualize this.



If you think about a bounding box of points A and B, and draw something that is further by another distance A-B, you will get a point that is on the border – anything further will not work with the “formula” we used. And what happens in those bad regions is that you want to move every point C there towards the border. If you draw a diagonal of a bad region, everything above it should be moved towards one wall, everything below towards the other wall. This can be coded for sure with some segment trees and sweeping but let’s do it nicely.

Let’s rotate the grid 4 times and just focus on cases with C that is above-and-on-left from A and B. Let’s iterate over pairs (A, B), and get the coordinates of the red corner in above-left from them. We need the sum of distances of all points C in that region towards a closer red wall. Let define this value as dp[x][y] for red corner at (x, y). This is easy to compute:

```
dp[x][y] = dp[x-1][y-1] + cnt[x-1][y-1]
```

Where `cnt[r][t]` is the number of points up to cell `(r, t)`. It works because every point C will have its distance increased by 1 if we move with red corner by +1 in both coordinates.

The complexity is `O(Z^2)` where `Z=MAX(N, coordinates)`. It’s ok to use binary search for extra logarithm in the first part.

```
public long sumL(int Y[]) {
    final int N = Y.length;
    int[] X = new int[N];
    for(int i = 0; i < N; ++i) {
        X[i] = i;
    }
    int C = 0;
    for(int i = 0; i < N; ++i) {
```

```
        C = max(C, max(X[i], Y[i]));
    }
    C++;
    long answer = 0;
    for(int rep = 0; rep < 2; ++rep) {
        // sum over min(x2-x1, x3-x2) in this dimension
        int[] value = new int[N];
        for(int i = 0; i < N; ++i) {
            value[i] = (rep == 0 ? X[i] : Y[i]);
        }
        sort(value);
        for(int i = 0; i < N; ++i) {
            for(int j = i + 1; j < N; ++j) {
                // you can get rid of BS with prefix/suffix sums
                int first_right = lower_bound(value, value[j] + (value[j] -
value[i]));
                int first_not_left = lower_bound(value, value[i] - (value[j] -
value[i]));
                int times = (N - first_right) + first_not_left;
                answer += ((long) (value[j] - value[i])) * times;
            }
        }
    }
    for(int rep = 0; rep < 4; ++rep) {
        for(int i = 0; i < N; ++i) { // rotate by 90 degrees
            int tmp = X[i];
            X[i] = Y[i];
            Y[i] = C-1-tmp;
        }
        final int INF = 1000 * 1000 * 1000 + 5;
        int[] from_x = new int[C];
        int[] from_y = new int[C];
        for(int i = 0; i < C; ++i) {
            from_x[i] = from_y[i] = INF;
        }
        for(int i = 0; i < N; ++i) {
            from_x[X[i]] = Y[i];
            from_y[Y[i]] = X[i];
        }
        long[][] dp_count = new long[C][C];
        long[][] dp_sum = new long[C][C];
        // dp[x][y] is pair (sum, count) in quarter up to (x, y)
        for(int x = 0; x < C; ++x) {
            for(int y = 0; y < C; ++y) {
                if(x > 0 && y > 0) {
```

```
                    dp_sum[x][y] = dp_sum[x-1][y-1] + dp_count[x-1][y-1];
                    dp_count[x][y] = dp_count[x-1][y-1];
                }
                if(from_x[x] <= y) {
                    dp_count[x][y]++;
                }
                if(from_y[y] < x) {
                    dp_count[x][y]++;
                }
            }
        }
        for(int i = 0; i < N; ++i) {
            for(int j = i + 1; j < N; ++j) {
                int x = min(X[i], X[j]) - abs(X[i] - X[j]);
                int y = min(Y[i], Y[j]) - abs(Y[i] - Y[j]);
                if(x >= 0 && y >= 0) {
                    answer += dp_sum[x][y];
                }
            }
        }
    }
    return answer;
}
```