TOKI Open 2014

Problem Analysis

Hosted by:

Indonesian Computing

Olympiad Team (TOKI)

May 8 - 10, 2014

Generous Butcher

This problem seems difficult and seems to involve complicated math. Not actually. Go get your pencil and paper, then start simulating what the problem asks. You may notice something familiar!

For subtasks 1 through 4, K_1 is equal to K_2. For simplicity, let's call both of them K.

Subtask 1

We only need to find the K^{th} digit, where K is not greater than 10. Notice that the built-in 64-bit floating-point data type (for example `double` in C++) can handle up to approximately 16 digits after the decimal mark. So, just store A/B in that data type. Retrieving the K^{th} digit can be done by multiplying it by 10^K, rounding it down, and taking it modulo 10.

Subtask 2

Now, K is up to 10^5 and this is when the paper-and-pencil strategy starts working.

First, if A is not smaller than B, take A modulo B. To get K^{th} digit in the fractional part, we can simply repeat this process K times:

1. Let A' to be A multiplied by 10.
2. Let R to be A' divided by B, rounded down.
3. Set A to A' modulo B.

In the end, the last R will be the answer.

For a better understanding, take a look at the following picture. We need to find the 4^{th} digit with $A = 15$ and $B = 123$. Blue zeros indicate the results of multiplication of A and 10 at each iteration. The digit we are looking for is colored with red.

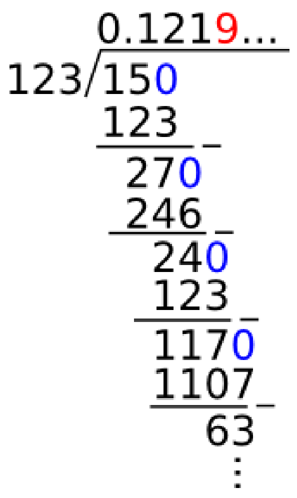


Figure 1. Simulation of 15 divided by 123

This solution runs in $O(K)$, and is sufficiently fast for this subtask.

Subtask 3

The solution described in the previous subtask is too slow for $K = 10^9$. We need faster solution.

You may notice that after certain iterations, the fractional digits will keep repeating and form a cycle. For example:

$15/17 = 0.\mathbf{8823529411764705}8823529411764705882352941176470588 23\ldots$
$1/360 = 0.002\mathbf{7}7777777\ldots$

In the first example, the digits "8823529411764705" will keep repeating forever. In the second example, after digits "002", the digit "7" will keep repeating forever.

Based on this observation, we can use cycle-finding algorithm to retrieve the K^{th} digit. A brute-force cycle-finding approach works in $O(L + U)$, where L is the number of steps before a cycle occurs and U is the length of the cycle. In fact, both of L and U are bounded above by B. This is because there are at most B possibilities for A' in the algorithm described in the previous subtask 2. So, this cycle-finding solution runs in $O(B)$.

Subtask 4

Forget about cycle finding and go back to the "paper-and-pencil" technique. In the Figure 1, the 4^{th} digit is the result of $\lfloor(117 \times 10)/123\rfloor$ (where $\lfloor a\rfloor$ denotes the floor function of a). Where does 117 come from? Actually, 117 comes from $A \times 10^{K-1}\ mod\ B$. So, in order to find K^{th} digit, we just need to compute $\lfloor(A \times 10^{K-1}\ mod\ B) \times 10/B\rfloor$. The hardest part is to compute $10^{K-1}\ mod\ B$, as K can be really large. Fortunately, there is a well-known fast algorithm for computing $X^Y\ mod\ Z$, which runs in $O(\log Y)$ called exponentiation by squaring. The complexity of this solution is $O(\log K)$.

Subtask 5

Previously, we can find K^{th} digit in $O(\log K)$. Now K_1 is not always equal to K_2, but we can compute each of the ${K_1}^{th}$ through ${K_2}^{th}$ digits with the previously described algorithm. The complexity is $O((K_2 - K_1) \times \log K)$, where K is $max(K_1, K_2)$. This algorithm is fast enough to obtain full score.

Social Inequality

Let's convert a citizen living at location x and having income y to a point (x, y) on a plane. The problem is then reduced to: given N points, compute the total area of all rectangles formed by each pair of points as diagonally opposite corners.

Subtask 1

N is small enough, so brute-force approach will work. Simply consider all possible pair of points, compute the area, and sum them. This solution works in $O(N^2)$.

Subtask 2

Notice that the maximum size of the plane is only 100×100. There are $C(100,2) \times C(100,2)$ different rectangles that can be formed. This number is actually not too large: $C(100,2)$ is only 4,950. So, we can enumerate all possible rectangles. Since two or more pairs of points can form the same rectangle, we have to find the number of ways we can form each rectangle.

Take a look at Figure 2.

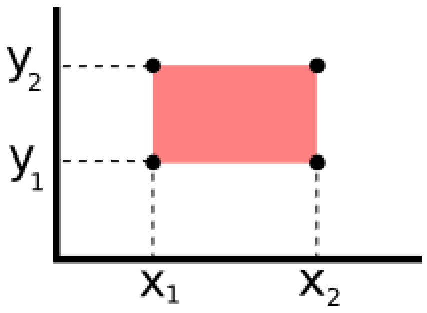


Figure 2. Rectangle with lower-left corner (x_1, y_1) and upper-right corner (x_2, y_2)

Let $n(x, y)$ denote the number of points whose location is (x, y). The number of ways we can form a rectangle with lower-left corner (x_1, y_1) and upper-right corner (x_2, y_2) is $n(x_1, y_1) \times n(x_2, y_2) + n(x_1, y_2) \times n(x_2, y_1)$. Storing $n(x, y)$ for any x and y can be done easily by creating 2D-array acting as frequency table.

The answer is simply the sum of each rectangle's area multiplied by how many ways to form it. This solution works in $O(MAX\_X^2 \times MAX\_Y^2)$, which is fast enough to run under time limit.

Subtask 3

When forming a rectangle, notice that each point has four possibilities. It may become:

1. Upper-left corner
2. Upper-right corner
3. Lower-left corner
4. Lower-right corner

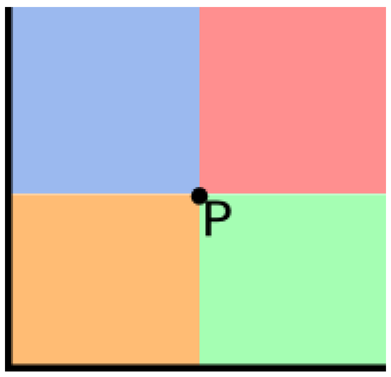


Figure 3. Point location possibilities visualization (for every different colored rectangle)

If some point is located in blue region (in Figure 3), then together with point P they can form a rectangle so that point P becomes its lower-right corner. For each point P, we can efficiently compute the total area of all rectangles which use P as their lower-left/right point. Take a look at Figure 4.

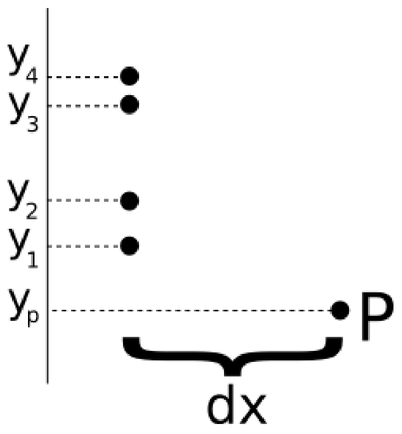


Figure 4. Computing the sum of area of rectangles having point P as lower-right corner and difference of abscissa dx

Consider a point P. Let dx be a fixed integer. Let's say that there are four points, whose abscissa is dx less than the abscissa of P. The total area of rectangles that can be formed so that P becomes the lower-right corner is:

$$area = dx \times ((y_1 - y_P) + (y_2 - y_P) + (y_3 - y_P) + (y_4 - y_P))$$

This can be rewritten as:

$$area = dx \times (y_1 + y_2 + y_3 + y_4 - 4 \times y_P)$$

In general, if there are M points whose abscissa is dx less than abscissa of P, and they are located at ordinates $y_1, y_2, \ldots, y_M$, then the total area of rectangles that can be formed having P as their lower-right corner is:

$$area = dx \times \left(\sum_{i=1}^{M} y_i - M \times y_P \right)$$

Finding M and $y_1 + y_2 + \cdots + y_M$ can be done in constant time using partial sum arrays. As there are at most only 100 possible values for dx, we can just iterate it to compute the total area of rectangles area having P as their lower-right corner. Similar method can be used to compute the total area of rectangles area having P as their lower-left corner. The running time is $O(MAX\_X)$.

There are N points to be considered, so the repeat the same process for each point, and accumulate the result. The total complexity for this solution is $O(N \times MAX\_X)$.

Subtask 4

There are at least two approaches to solve this subtask: 1) line sweep, and 2) divide-and-conquer.

Line Sweep

Generalizing the basic idea explained in solution of Subtask 3:
Given a point P, can we compute the total area of rectangles having P as their upper-right corner efficiently?

For two points (x_1, y_1) and (x_2, y_2), another way to compute the rectangle area is shown in Figure 5.

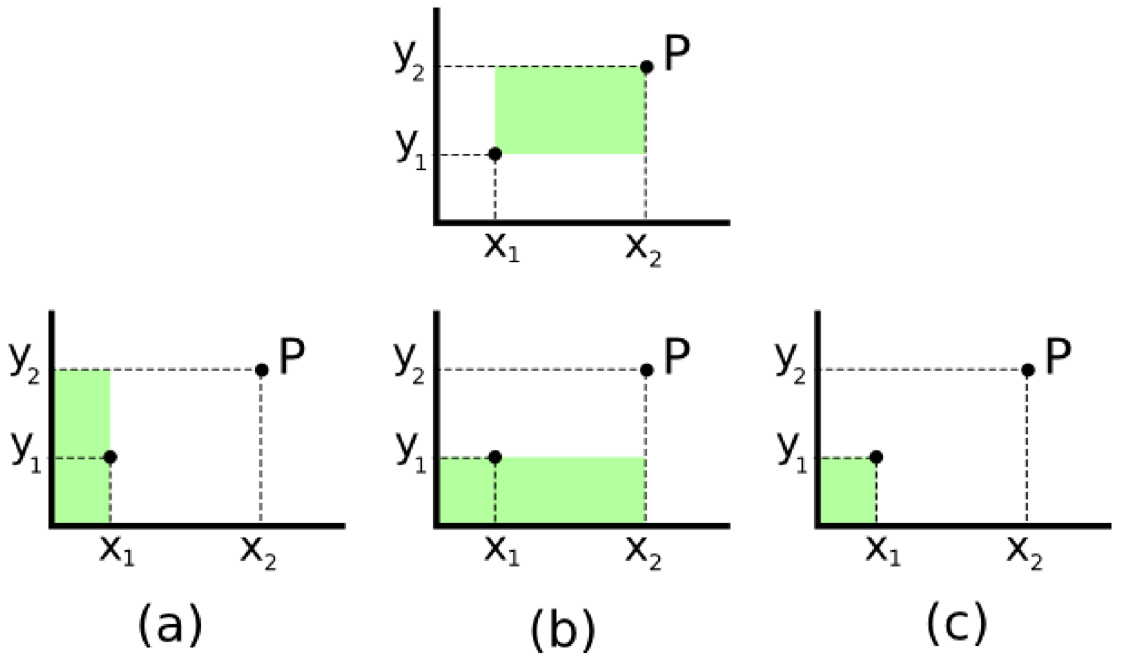
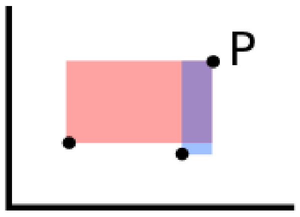


Figure 5. Shaded area shown in the upper picture can be computed as $x_2 \times y_2$ subtracted by (a) and (b), then added with (c)

What if there are more points (x, y) satisfying $0 \leq x \leq x_2$ and $0 \leq y \leq y_2$? Similar method to compute the areas can be generalized and shown in Figure 6.



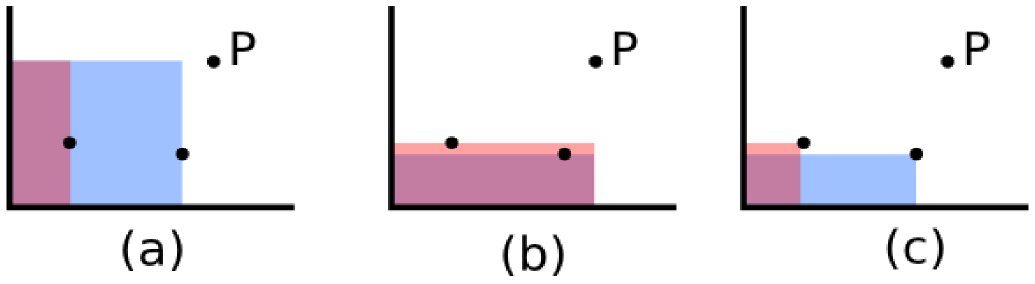


Figure 6. Generalizing the method when there are more than one point within P

In general, if there are M points (x_1, y_1), (x_2, y_2), (x_3, y_3), …, (x_M, y_M), whose abscissas are not greater than x_P and ordinates are not greater than y_P, then the total area of rectangles having $P = (x_P, y_P)$ as their upper-right corner can be computed as:

$$area = (M \times x_P \times y_P) - x_P \sum_{i=1}^{M} y_i - y_P \sum_{i=1}^{M} x_i + \sum_{i=1}^{M} x_i \times y_i$$

Now, the basic idea for this line sweep approach is to sweep an imaginary line, starting from left to right. While sweeping, every time the line encounters a point, compute the total area of rectangles having that point the upper-right corner using the above formula. We can use four separated data structures like segment trees or Fenwick trees for keeping track of M, $(y_1 + y_2 + \cdots + y_M)$, $(x_1 + x_2 + \cdots + x_M)$, $(x_1 \times y_1 + x_2 \times y_2 + \cdots + x_M \times y_M)$, and query them efficiently. When we reach the rightmost point, sweep again to the left to cover cases when the point become upper-left corner. The complexity of this solution is $O(N \log N)$, as updating and querying the data structures are at most in $O(\log N)$.

Divide-and-Conquer

The basic idea is really generic. Given N points P_1, P_2, …, P_N, computing the total area of all rectangles can be done as follows:

1. **Divide**: partition $[P_1, P_2, \dots, P_N]$ into $[P_1, P_2, \dots, P_m]$ and $[P_{m+1}, P_{m+2}, \dots, P_N]$. Compute the total area of all rectangles formed by $[P_1, P_2, \dots, P_m]$ and $[P_{m+1}, P_{m+2}, \dots, P_N]$ independently.
2. **Conquer**: when the given array of points has less than 2 points, the total area of all rectangles is 0.
3. **Combine**: given $[P_1, P_2, \dots, P_m]$ and $[P_{m+1}, P_{m+2}, \dots, P_N]$, compute the total area of all rectangles formed by all possible P_i and P_j where $1 \le i \le m$ and $m + 1 \le j \le N$.

The first and second steps can be computed easily. The real challenge lies in the third step. We can use similar idea from solution explained in the previous section.

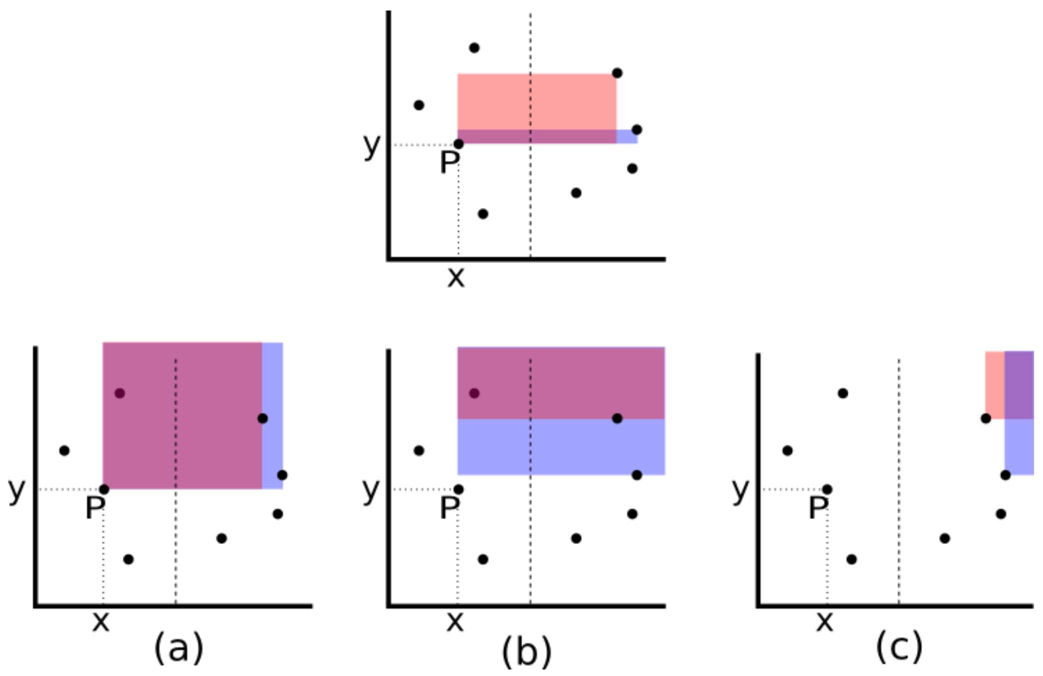


**Figure 7. Computing the total area of rectangles having P as lower left-corner.
Relation of upper picture with (a), (b), and (c) is similar to that of Figure 6**

We can process the points in sorted order of ordinates. While processing, we update the value of M, $(y_1 + y_2 + \cdots + y_M)$, $(x_1 + x_2 + \cdots + x_M)$, $(x_1 \times y_1 + x_2 \times y_2 + \cdots + x_M \times y_M)$ in the same way as the solution explained in previous section. Note that we don't really need tree data structures. You will find out that this algorithm has the same fashion as classic merge-sort.

With good implementation, the total complexity for this solution is $O(N \log N)$.

Multiple Choice

Subtask 1

We can use dynamic programming to solve this subtask. Let $f(i,a,b)$ denote the number of answer keys for problem i through problem N, if Anto answered a problems and Budi answered b problems correctly. The recurrence itself is quite straightforward:

$$f(i,a,b) = \begin{cases} 0 & ,i = 0 \wedge (a \neq P \vee b \neq Q) \\ 1 & ,i = 0 \wedge (a = P \vee b = Q) \\ \sum_u f\big(i-1, a + answered(Anto,u), b + answered(Budi,u)\big) & ,i > 0 \end{cases}$$

where $answered(S,x)$ is 1 if person S answered x for his i^{th} problem.

The complexity of this solution is $O(N^3)$.

Subtask 2

We can solve this subtask with dynamic programming, but with a little modification.

For each problem, there are two possible answers and Anto's answer will be different to Budi's answer. This suggests two possible cases:

1. Anto is the one who answered correctly, or
2. Budi is the one who answered correctly.

Now we can formulate $f(i,a,b)$ with the following recurrence:

$$f(i,a,b) = \begin{cases} 0 & ,i = 0 \wedge (a \neq P \vee b \neq Q) \\ 1 & ,i = 0 \wedge (a = P \vee b = Q) \\ f(i-1,a+1,b) + f(i-1,a,b+1) & ,i > 0 \end{cases}$$

Again, the complexity is $O(N^3)$, which is too slow for N up to 2,000. We need optimization.

Notice that for any case, i is decreased by one and whether a or b is increased by one. We can drop parameter i in our formulation, and retrieve i as $N - a - b$. This modification yields $O(N^2)$ solution which works fast for N up to 2,000.

Subtask 3

We need several observations.

1. There are problems where Anto and Budi have the same answer. We will group these problems and call it the $SAME$ group.
2. There are problems where Anto and Budi have different answers. We will group these problems and call it the $DIFF$ group.

For $SAME$ group:

1. There are problems where both of them answered correctly. We will group them and call it group A.
2. There are problems where both of them answered incorrectly. We will group them and call it group B.

For $DIFF$ group:

1. There are problems where Anto answered correctly but Budi answered incorrectly. We will group them and call it group C.
2. There are problems where Anto answered incorrectly but Budi answered correctly. We will group them and call it group D.
3. There are problems where both Anto and Budi answered incorrectly. We will group them and call it group E.

Let $n(S)$ denote the number of elements in S. Define:

- $a = n(A)$
- $b = n(B)$
- $c = n(C)$
- $d = n(D)$
- $e = n(E)$
- $WA =$ the number of problems Anto answered incorrectly
- $WB =$ the number of problems Budi answered incorrectly

We can connect the variables by these equations:

- $a + b = n(SAME)$
- $c + d + e = n(DIFF)$
- $b + d + e = WA$
- $b + c + e = WB$

There are 4 equations and 5 variables, so the degree of freedom is 1. Therefore, there are at most N different solutions for this equation. We can enumerate all possible (a, b, c, d, e) tuples in $O(N)$.

Now we know the number of problems which were answered incorrectly by both of them and how many problems which were answered incorrectly by one of them. We need to know the number of answer keys satisfying these constraints. This can be computed by this formula:

$$f(a,b,c,d,e) = fSame(n(SAME), b) \times fDiff(n(DIFF), e) \times C(c + d, c)$$

Where:

- $fSame(a, b) =$ the number of ways so that Anto and Budi answered b problems incorrectly out of the first a problems from $SAME$.
- $fDiff(a, b) =$ the number of ways so that Anto and Budi answered b problems incorrectly out of the first a problems from $DIFF$.
- $C(a, b) =$ the number of ways to choose b items out of a items.

Before that, let's precompute $fSame$ and $fDiff$. This can be done via dynamic programming using following recurrences:

$$fSame(a,b) = \begin{cases} 1 & , a = 0 \wedge b = 0 \\ 0 & , a = 0 \wedge b \neq 0 \\ fSame(a-1,b) + (choices(a) - 1) \times fSame(a-1,b-1) & \text{, otherwise} \end{cases}$$

$$fDiff(a,b) = \begin{cases} 1 & , a = 0 \wedge b = 0 \\ 0 & , a = 0 \wedge b \neq 0 \\ fDiff(a-1,b) + (choices(a) - 2) \times fDiff(a-1,b-1) & \text{, otherwise} \end{cases}$$

where $choices(a)$ denotes the number of possible answers in the a^{th} question of the corresponding group.

The complexity to compute $fSame$, $fDiff$, and combination table is $O(N^2)$, and looping through all possible (a, b, c, d, e) tuples is only $O(N)$. The overall complexity is $O(N^2)$.

Company Planning

Subtask 1

We try all combinations: for each employee, decide whether to fire him or not. Out of all combinations, we select valid combinations that have at least M employees remaining in the company. There are at most 2^N valid combinations.

For each valid combination C, we define $count(C) =$ the maximum number of subordinates (that are still in the company) that an employee (that is still in the company) has. We choose the combination X that has the minimum value of $count(X)$.

Calculating $count$ can be done in $O(N)$, so this algorithm runs in $O(N \times 2^N)$.

Subtask 2 & 3

We try all possible values of K. Now the problem is reduced to finding the maximum number of employees that can still in the company, given that each employee should have at most K subordinates.

Let $optimal(T, K) =$ the maximum number of employee in the subtree T that can still remains in the company, given that every employee have at most K subordinates.

The function $optimal()$ has an optimal substructure. To maximize $optimal(T, K)$, we need to maximize $optimal(T', K)$, for all T' subordinate of T. Therefore, $optimal(T, K) =$ the total of K largest $optimal(T', K) + 1$, The "$+1$" addition comes from root of T itself.

The answer for a particular K is $optimal(root, K)$, which can be computed in can be done in $O(N \log N)$. For subtask 2, there are at most N different values of K, so the complexity is $O(N^2 \log N)$, which runs sufficiently fast for N up to 1,000. For subtask 3, there are at most 3 different values of K, so the complexity is $O(N \log N)$, which runs sufficiently fast for N up to 100,000.

There might be solutions that distinguish subtasks 2 and 3, though.

Subtask 4

Since we cannot fire anyone, the answer is just the maximum number of subordinates that an employee has in the company.

Subtask 5

Observation: suppose if each employee has at most K subordinates, the maximum number of employees that are still remaining in the company is M. Then, if each employee has at most $K + 1$ subordinates, the maximum number of employee that are remaining in the company is at least M.

This observation enables us to do binary search the answer. This solutions runs in $O(N \times (\log N)^2)$.

Safest Route

This is probably the hardest problem in TOKI Open 2014. The solution heavily involves shortest path algorithm and ideas beyond it.

Subtask 1

The number of possible routes is $(2^N \times N!)$. When N is small enough, we can enumerate all possible answers and pick the best one.

Subtask 2 - 4

The formula for computing the “not-so-ideal” length can be used as path cost in any shortest path algorithm. For example, we will use the classic Dijkstra algorithm:

```
dijkstra(source){
   initialize dist, an array of size N with initial values INFs
   dist[source] = 0
   while(){
      u = extract_min()

      for v = 1 .. N
         relax(u, v, dist)
   }
}

relax(u, v, dist){
   path_cost = dist[u] + cost[u][v]
   if (dist[v] > path_cost)
      dist[v] = path_cost
}
```

In the algorithm above, the path cost is simply the total costs of traveled edges cost. We change the path cost into “not-so-ideal” length:

```
relax(u, v, dist){
   path_cost = max(optimal_blocked[u][v], dist[v] + cost[u][v])
   if (dist[v] > path_cost)
      dist[v] = path_cost
}
```

where `optimal_blocked[u][v]` is minimum cost needed to travel from `source` to `v` without using edge (`u`, `v`). Based on this observation, we need to compute `optimal_blocked` for all possible `u` and `v`. There are numerous approaches with various complexities, which may yield to various number of solved subtasks. To keep this analysis short and compact, we will explain the basic idea and the model solution we used.

Basic Idea

Let’s run a shortest path algorithm. This algorithm will determine a shortest path from source to all nodes. Let’s fix any shortest path:

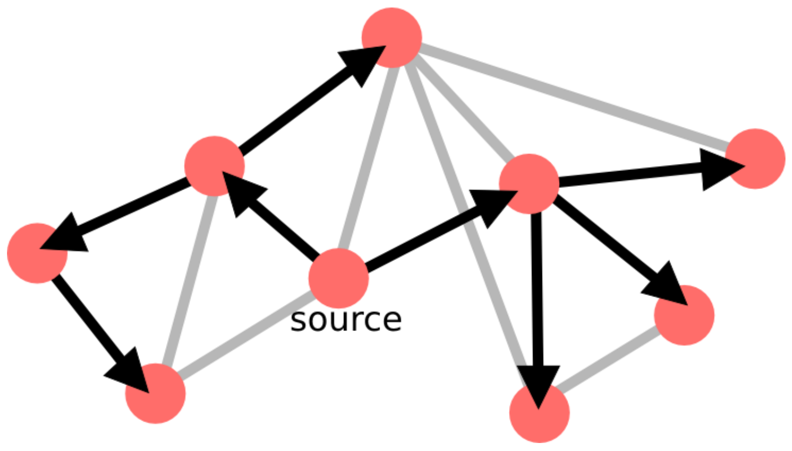


Figure 8. Result after executing shortest path algorithm in an undirected graph

In the Figure 8 above, black edges indicate shortest paths, and grey edges indicates edges not used in shortest paths.

Then,

1. Consider edge (`u`, `v`). If the shortest path from `source` to it does not involve that edge, then `optimal_blocked[u][v]` is equal to `shortest_path[v]`.
2. If edge (`u`, `v`) is not a part of the shortest path from `source` to any node, then we call it unimportant edge. Otherwise, it is important edge.

We break the computation of `optimal_blocked[u][v]` down into two cases:

1. For any unimportant edge (`u`, `v`), the cost of `optimal_blocked[u][v]` is equal to `shortest_path[v]`. If we do preprocessing to compute the shortest path from `source` to any node, we can obtain `optimal_blocked[u][v]` for any unimportant edge in $O(1)$.
2. For any important edge (`u`, `v`), we need to re-run the shortest path algorithm from `source` with edge (`u`, `v`) removed.

The cost for running shortest path algorithm is $O(N^2)$. So the overall complexity will be $O(k \times N^2)$ where k is the number of important edge. Now, how big is k?

From Figure 8, obviously, the set of important edges form a spanning tree in the graph. So, the number of important edges is equal to $N - 1$.

Now we can find all `optimal_blocked[u][v]` for all possible `u` and `v` in $O(N^3)$. The rest is just running shortest path algorithm to find the safest route using "not-so-ideal" cost definition in $O(N^2)$.

Model Solution

In Figure 8, it is shown that a spanning tree will be formed after running shortest path algorithm. If we take source as the root node, we will get a rooted tree.

Suppose we have a rooted tree, and we are trying to find `optimal_blocked[u'][u]` where `u'` stands for the parent of node `u`. Then the tree will be disconnected into two trees as shown in Figure 9.

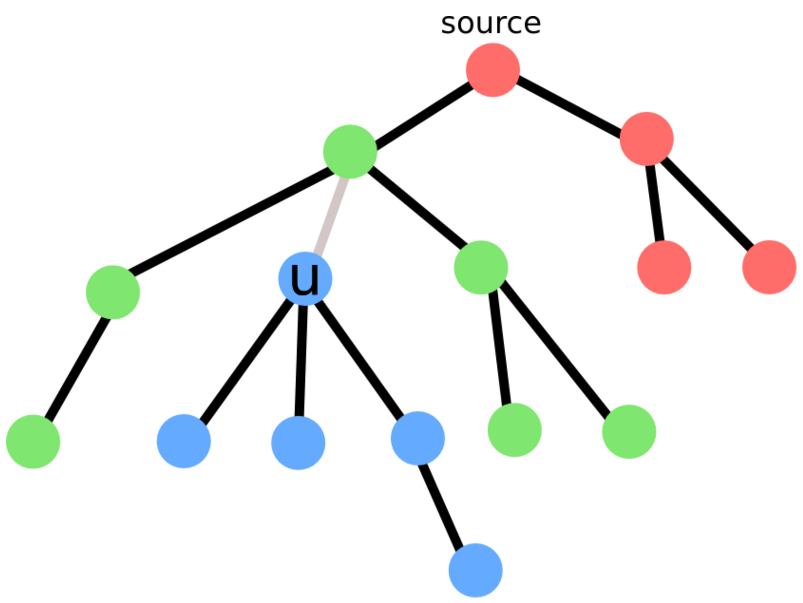


Figure 9. Disconnected tree after removing edge between `u` and its parent, shown in grey line

The value of `optimal_blocked[u'][u]` can be achieved by selecting node `v` and `w`, such that `v` is a node in subtree rooted in `u` (in Figure 9, subtree of `u` is colored blue) and `w` is not contained in subtree rooted in `u` (in Figure 9, possible nodes for `w` are colored green/red). The cost of the choice of node `v` and `w` will be `dist[u][v] + dist[v][w] + shortest_path[w]`, or simply a distance traveled from `u` to `v`, `v` to `w`, and `w` to `source` (here, `source` should be equal to destination/finish node).

Checking for all possible `v` and `w` will be too slow. In fact, we can prune the search space. The candidates for `w` actually are composed of all nodes in subtree rooted in parent of `u`, with nodes in subtree rooted in `u` excluded. In Figure 9, these nodes are shaded in green. With this observation, each node will be compared to at most N times. So the complexity to compute all `optimal_blocked` will be in $O(N^2)$.

Subtask 5

This is a very special case of this problem. From the constraints, we can conclude that the graph is ring-shaped. We just need to compute shortest path from `source` and from finish node, then find the maximum cost if a certain edge is unavailable. The complexity of this solution is $O(N)$.

Jump!

For convenience, let's **renumber** the stones into $0, 1, 2, \ldots, N-1$.

Subtask 1

We can do simulation. For each `UPDATE` and `RESIZE` query we just change whatever necessary. We can do this in $O(1)$.

For `JUMP` query, we just try all possible values of P. For any value of P, we compute the total points assigned in the marked stones and we choose the maximum one. Since there are $N-2$ possible values of P and Si Katak will visit all the stones in the worst case, the complexity is in $O(N^2)$. The total running time for this algorithm is $O(Q \times N^2)$.

Subtask 2

To obtain more points, we must have several observations.

Observation 1: For each P, Mr. Dengklek will mark all stones with number X for all X multiple of $gcd(N, P)$.

Observation 2: For each N, the value of $gcd(N, P)$ for all $P \leq N$ is a factor of N.
Proof: By definition, $gcd(N, P)$ is factor of both N and P.

Observation 3: For each N, for all M where M is a factor of N, then $M = gcd(N, P)$ for some $P \leq N$.
Proof: Set $P = M$. $gcd(N, P) = gcd(N, M) = gcd(M, N \bmod M) = gcd(M, 0) = M$.

Observation 4: For each N, suppose we have sets S_1 and S_2 where for all element X in S_1, $gcd(N, P) = X$ for some $P \leq N$ and for all element Y in S_2, Y is a factor of N. Then $S_1 = S_2$.
Proof: Combination of **Observation 2** and **Observation 3**.

Axiom 1: There are at most $O(\sqrt{N})$ factors of N.

Observation 1 and **Observation 4** make `JUMP` queries faster to compute. For each `JUMP` query, we just try all possible values of P where P is a factor of N. Since we do another simulation, Si Katak will visit all the stones in the worst case.

The running time for this algorithm is $O(Q \times N \times \sqrt{N})$.

Subtask 3

If there are no `RESIZE` queries, then we can create an array `sum[]`, where `sum[X]` = the sum of all points assigned in all stones with number Y, for all Y multiple of `X`.

For each `UPDATE X Y` query, we must update all `sum[X']`, where `X'` is a factor of `X`. We can do this in $O(\sqrt{N})$.

For each `JUMP` query, we just try all possible values of `X` where `X` is a factor of N. We choose `X` such that `sum[X]` is maximized. We can do this in $O(\sqrt{N})$.

The total running time of this algorithm is $O(Q \times \sqrt{N})$.

Subtask 4

Define `sum[X][K]` as the sum of all points assigned to all stones between 0 and `K` with number Y, where Y is a multiple of `X`.

This data structure is necessary, so we can retrieve the sum of the first N stones when there are `RESIZE` queries. The data structure can be implemented with Fenwick tree.

The total running time of this algorithm is $O(Q \times \sqrt{N} \times \log N)$. This algorithm will solve all subtasks.

Fowl Sculptures

The main idea of this problem is: given N points on a plane, find a line L which crosses the origin and maximizes number of points which has reflection with respect to line L. A point P is said to have reflection with respect to line L if there exists point P', such that P' on the reflection of P with respect to line L.

Computational geometry problems like this are likely to be prone to precision errors. However, in this problem, almost all computations can be done in integers. The only computation which involves floating-point is converting the answer into degrees.

Subtask 1

It is guaranteed that θ is 0. That means we just need to count the number of points that have the reflection. Reflecting a point with respect to Y-axis can be done by simply negating the ordinate of the point. To check the existence of the reflection, just iterate through all points in $O(N)$. As we repeat this process N times, the total complexity is $O(N^2)$.

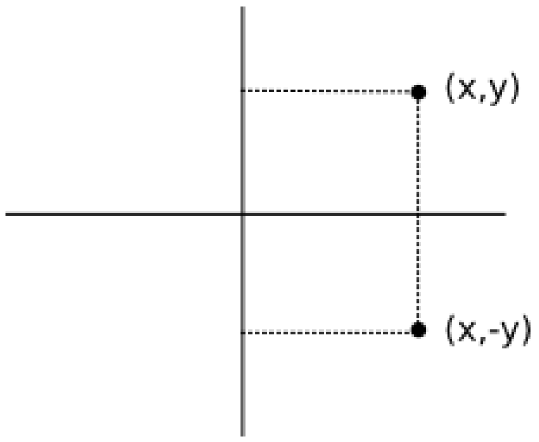


Figure 10. Reflecting a point with respect to Y-axis

Subtask 2

Let's begin with brute-force approach: for each line L, compute the number of points that have the reflection. Of all possible L, pick one which maximizes the number of points that have the reflection. We will call this line "optimal line".

There are infinitely many choices for line L, but notice that the optimal line will be always the bisector of two existing points. There are at most C_2^N bisectors which crosses the origin, and counting the number of points that have the reflection can be done in $O(N)$. So, the complexity of this solution is $O(N^3)$.

The difficulty lies in finding the bisector of two given points. This can be done by utilizing some algebra and basic geometry properties, depending on how you represent lines. There are a lot of ways to represent a line. The judge's solution uses $Ax + By = C$ representation. As we are interested in lines which cross the origin, C will be always 0.

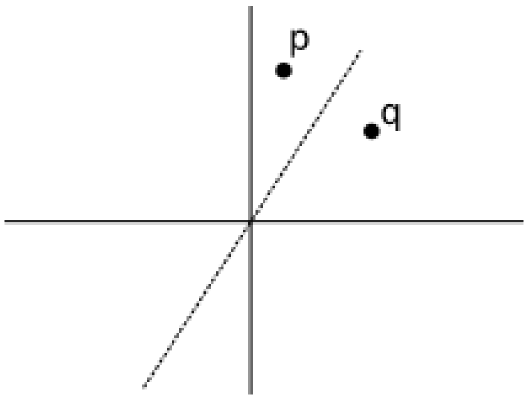


Figure 11. The bisector between points p and q

Subtask 3

Based on the solution explained above, the optimal line maximizes the number of points which have the reflection. The implication is the optimal line L will have the most number of pair of points that have L as their bisector. So, we can enumerate all bisectors of each pair of points, and pick one that appears the most number of times.

There are at most C_2^N bisectors. To pick one that appear the most, we can sort those bisectors according to their θ and do a linear sweep. The complexity is $O(N^2 \log N^2)$, or can simply be written as $O(N^2 \log N)$.