Solution sketches

Competitive Programming Tryouts 2016-2017

Robin Yu

July 7, 2016

Abstract

The competitive programming tryouts 2016-2017 was held from June 30 to July 7. All candidates for programming varsity had to achieve a minimum of either 550 points, or 300 points and a score not lower than the score of the candidate at fifth place.

The candidates were presented with eight problems of increasing difficulty, each worth a maximum of 100 points. The maximum score for this contest was therefore 800 points.

Only two candidates reached the minimum required score of 300. The first to achieve this minimum required score was Alex Go, who achieved it on July 5, 6:57 PM. Kirby Chua followed the next day, clinching the minimum required score on July 6, 11:51 PM—just nine minutes before the end of the contest! This is proof that one should never give up, even when the chance of success seems very slim. Congratulations to these two candidates for qualifying!

Out of all eight problems, only two problems were fully solved. Hopscotch and 中式点心 both had several solutions scoring 100 points. Four other problems had submissions scoring partial points; Reproduction had three solutions with 74 points, The Ampersand Literature and Art Folio had two solutions with 26 points, Hopscotch 2 had two solutions with 12 points, and Trouble at Bunnyburrow had a single solution which managed to score 12 points. Papers, Please only had a single submission which did not score any points, and #DayaangMatuwad had no attempts at all.[1]

In total, 555 submissions were made during the course of the contest. The effort many participants put to solve these problems was commendable, but unfortunately, many fell just short of reaching the minimum required score of 300. We sincerely hope that you will continue these efforts to qualify next time!

This document contains short explanations of working solutions for each of the problems. If you have any questions about the explanations, do not hestitate to send me an e-mail.

Working code for each of these problems can be found at this GitHub repository.

[1] This is understandable, as the problem does not even have a trivial brute force solution, unlike the others; the easiest subtask already requires an understanding of graph theory and related algorithms.

Contents

1 Hopscotch

1.1 Problem statement

The problem can be summarized as follows: Given an array $A_1, A_2, A_3, \dots, A_N$, answer M queries. In each query, you are given two integers, L and R, and you need to find the value of $A_L + A_{L+2} + A_{L+4} + \cdots + A_R$. Note that $R - L$ is divisible by 2.

1.2 Brute force

For the first two subtasks, we have N and M up to 8000. For each query, let's perform a simple loop from L to R, skipping every other element. In the worst case, we have to iterate through $\frac{N}{2}$ elements, so each query is answered in $O(N)$. There are M queries, so this runs in $O(NM)$, enough for the first two subtasks.

1.3 Prefix sums

The next two subtasks have N and M up to 200000, so this will no longer work, as NM can be up to $4 \cdot 10^{10}$. If we can perform 10^9 operations per second, then this solution can take up to 40 seconds, which is too slow.[2]

To perform this more efficiently, let's consider a simpler problem. Instead of having to find the value of $A_L + A_{L+2} + A_{L+4} + \cdots + A_R$, suppose we simply want to look for the value of $A_L + A_{L+1} + A_{L+2} + \cdots + A_R$. To solve this, let's create an array C where $C_i = A_1 + A_2 + A_3 + \cdots + A_i$. This can be computed in $O(N)$ using just one linear pass, by noting that $C_i = C_{i-1} + A_i$.

Now, we note that $A_L + A_{L+1} + A_{L+2} + \cdots + A_R = C_R - C_{L-1}$, because $C_R = A_1 + A_2 + A_3 + \cdots + A_R$, but we don't want the values $A_1 + A_2 + A_3 + \cdots + A_{L-1}$, so we subtract these off using C_{L-1}. Hence, each query can be answered in $O(1)$ time, so the problem can now be solved in just $O(N + M)$ time, which is fast enough to solve this modified problem.

Let's use the same idea to solve this problem.

Let $C_i = A_1 + A_3 + A_5 + \cdots + A_i$ if i is odd, and $C_i = A_2 + A_4 + A_6 + \cdots + A_i$ if i is even. We can use a similar reasoning to see that the answer to each query is simply $C_R - C_{L-2}$.

This also runs in $O(N + M)$, which scores full points for this problem.

1.4 Gotchas

There are a couple of potential pitfalls in this problem.

1. When A_i is up to 10^9, the sum can exceed the range of a 32-bit integer. We need to use a 64-bit integer in this case, or we risk facing overflow.

[2] No amount of code optimization will allow an inefficient algorithm to work; focus on improving the algorithm first, before optimizing constants!

2. There are a number of off-by-one errors to watch out for. For example, $L - 2$ is -1 when $L = 1$. Hence, C_{L-2} will be out-of-bounds, and we face a possible runtime error.

2 The Ampersand Literature and Art Folio

2.1 Problem statement

Given an array $P_1, P_2, P_3, \ldots, P_N$, compute the value of:

$$\sum_{\substack{i=1\\P_i \neq P_j}}^{N} \sum_{j=1}^{N} P_i P_j$$

Since this can be quite large, output the value modulo $10^9 + 7$.

2.2 Brute force

The first subtask only has N up to 5000. We can simply compute the formula as it is stated[3], using two nested loops. This runs in $O(N^2)$ and is too slow for anything else.

2.3 Playing around with the formulas

We want to find:

$$\sum_{\substack{i=1\\P_i \neq P_j}}^{N} \sum_{j=1}^{N} P_i P_j$$

It's easy to see that this is equal to:

$$\sum_{i=1}^{N} \sum_{j=1}^{N} P_i P_j - \sum_{\substack{i=1\\P_i = P_j}}^{N} \sum_{j=1}^{N} P_i P_j$$

Note, however, that the first nested summation is simply $(P_1 + P_2 + P_3 + \cdots + P_N)^2$. This can easily be computed in $O(N)$. The problem is therefore to compute the second nested summation efficiently.

When all P_i are distinct, as in the second subtask, then the second nested summation is easy to compute; it is simply $P_1^2 + P_2^2 + P_3^2 + \cdots + P_N^2$, because the only time $P_i = P_j$ is when $i = j$. Hence we can solve the second subtask in $O(N)$.

Otherwise, we have a bit more work to do.

Suppose the number x appears c times among $P_1, P_2, P_3, \ldots, P_N$. We want to find the value of:

$$\sum_{\substack{i=1\\P_i = P_j = x}}^{N} \sum_{j=1}^{N} P_i P_j$$

[3]Don't forget to take the value modulo $10^9 + 7$.

Since there are c ways to choose P_i and c ways to choose P_j, and $P_iP_j = x^2$, this is simply c^2x^2. The problem therefore reduces to computing c^2x^2 for all x. Of course, the only values of x that matter are those that actually appear in P, because otherwise $c = 0$ and therefore $c^2x^2 = 0$.
In the third subtask, we have P_i up to 10^6 only, as opposed to 10^9 in the other subtasks. Hence we can solve this by creating an array C where C_i is the number of occurrences of i in P. It holds:

$$\sum_{i=1}^{N}\sum_{\substack{j=1\\P_i=P_j}}^{N} P_iP_j = \sum_{i=1}^{\max P_i} C_i^2 i^2$$

This gives us an $O(N + \max P_i)$ algorithm, which passes the third (and only the third) subtask. In order to get points for the previous subtasks, it is enough to revert to brute force when $N \leq 5000$, and compute $P_1^2 + P_2^2 + P_3^2 + \cdots + P_N^2$ when all P_i are distinct (or if $N > 5000$ and some $P_i > 10^6$).

2.4 Full points

2.4.1 Map

To get points for the last subtask, we need to observe that, although P_i is up to 10^9, there are only up to 200000 distinct values present in P. Hence most elements in C are actually equal to 0. Hence, let's simply use a map instead of an array to store the values of C. This way, we can keep only the C_i which are greater than 0. The map gives us an extra logarithmic factor, but gets rid of the $\max P_i$ factor entirely; this is an $O(N \log N)$ solution. An efficient implementation of this should suffice for all the subtasks.

2.4.2 Sorting

Even if one does not know about map, it is still possible to solve this problem. Let's simply sort P using an efficient $O(N \log N)$ sorting algorithm, or using one of the built-in sorting methods in your language. Afterwards, we can note that equal P_i will all appear in consecutive positions in this array. We can walk through each of the elements of P and maintain a running count c of equal elements, adding $c^2P_{i-1}^2$ and then resetting c when $P_i \neq P_{i-1}$. Hence we no longer need to explicitly store the array C.
This also runs in $O(N \log N)$, and in fact has a smaller constant factor than the previous solution.[4]

[4]Although these solutions theoretically have the same runtime, in practice, the difference in running time is huge—the setter's C++ solution, using a map, takes 1.63 seconds, while the C++ solution using sorting only takes 0.35 seconds in the worst case. Had the constraints been larger, the map solution would already time out, while the sorting solution would still pass the time limit. Watch out for that constant factor!

3 Reproduction

3.1 Problem statement

Given two integers N and K, find K integers $A_1, A_2, A_3, \ldots, A_K$ such that $A_i > 1$ and $A_1 A_2 A_3 \ldots A_K = N$.

3.2 Solution for small K

3.2.1 When $K = 1$

When $K = 1$, there is no answer when $N = 1$. Otherwise, the answer is N.

3.2.2 When $K = 2$

When $K = 2$, there is no answer when $N = 1$ or when N is a prime number. Otherwise, we can simply find one nontrivial factor p of N, and then output p and $\frac{N}{p}$. The most straightforward way to do this runs in $O(N)$ per test case.

3.2.3 When $K = 3$

When $K = 3$, there is no answer when $N = 1$, when N is a prime number, or when N is a semiprime.
To solve this case, we can first perform some preprocessing. Make an array P where P_i is a nontrivial factor of i, or -1 if it doesn't exist. It's enough to do the preprocessing in $O(N^2)$ using two nested loops. Afterwards, for each test case, we can iterate over all p to find a nontrivial factor p of N, such that $\frac{N}{p}$ also has a nontrivial factor (i.e. $\frac{N}{p}$ is not 1 or a prime number).
This runs in $O(N)$ per test case, with an $O(N^2)$ preprocessing time.

3.3 General case

The number of cases we will have to consider grows very quickly as K grows, and the next subtask already has K up to 1000. We hence describe a solution which works for any K.
Let's get the prime factorization $p_1 p_2 p_3 \ldots p_k$ of N. If the number of prime factors k of N is smaller than K, then it is clear that no solution exists.[5] If it is equal to K, then we can simply print the prime factorization. Otherwise, K is less than k. In this case, let $A_1 = p_1 p_2 p_3 \ldots p_{k-K+1}$, and $A_i = p_{k-K+i}$ for all other i, then print A.
The problem hence boils down to finding the prime factorization of N.

[5]As all of the integers in the prime factorization are prime, by definition, they cannot be split further into two integers that are greater than 1.

3.3.1 When N is up to 5000

It's enough to iterate through all integers p from 2 up to N. When we find some p that divides N, we should divide N by p and then add p to the prime factorization. Note that we should try p again, because it may happen that p appears several times in the prime factorization. If p no longer divides N, then we should try the next p.
It can be easily proven using contradiction that all p found this way are prime.
This algorithm clearly runs in $O(N)$ in the worst case, which is when N is prime.

3.3.2 When N is up to 10^9

When N is up to 10^9, this is too slow. Instead, we should perform the previous algorithm, but instead of iterating through all p from 2 to $\sqrt{N}$, we can simply iterate through all p up to $\sqrt{N}$. This will find all prime factors up to $\sqrt{N}$. Note, however, that because N has at most one prime factor greater than $\sqrt{N}$, we can easily deduce this prime factor after we have found the rest of the prime factors of N.
This algorithm runs in $O(\sqrt{N})$ in the worst case.

3.3.3 When N is up to 10^{12}

When N is up to 10^{12}, you might have some trouble getting the previous algorithm accepted, because $\sqrt{N}$ is 10^6, and there are up to 2000 test cases. This hence takes $2 \cdot 10^9$ operations, which theoretically should pass in two seconds, but in practice is too slow because of the high constant factor involved with doing many multiplications and divisions.
We need one final observation—there is no point testing for divisibility by p which is not prime. Let's therefore perform some preprocessing to find all the primes p up to 10^6. This is most easily done using the Sieve of Eratosthenes, which runs in linearithmic time. Afterwards, we can simply perform the previous algorithm, but only testing for divisibility by primes.
This runs in $O(\pi(\sqrt{N}))$ time per test case, where $\pi(n)$ is the number of primes at most n. Since $\pi(10^6) = 78948$, this is already an over an order of magnitude faster than the previous solution. An efficient implementation of this should pass the time limit comfortably.

4 中式点心

4.1 Problem statement

Answer Q queries. In each query, compute:

$$\sum_{i=A}^{B}\sum_{j=C}^{D} ij$$

Where $1 \leq A \leq B \leq N$ and $1 \leq C \leq D \leq M$.
Output this sum modulo $10^9 + 7$.

4.2 Slow solutions

4.2.1 Brute force

The most obvious way is to compute the formula as is, using two nested loops. The first two subtasks can be solved simply by iteration, which runs in $O(QNM)$.

4.2.2 Preprocessing

The third subtask can also be solved with an efficient implementation of the previous idea.
Another solution is the following:
Define the following function f:

$$f(x, y) = \sum_{i=1}^{x}\sum_{j=1}^{y} ij$$

Now, each query can be answered using $f(B, D) - f(A-1, D) - f(B, C-1) + f(A-1, C-1)$.
For the third subtask, we can simply preprocess all values of $f(i, j)$ for all $1 \leq i \leq N$ and $1 \leq j \leq M$ in a similar way to the summed area table.
This takes $O(NM)$ time for preprocessing, and then each query can be answered in $O(1)$, giving us an $O(NM + Q)$ algorithm.

4.3 Arithmetic progressions

For the next subtask, this no longer works because M can be up to 10^9.
Let's go back to the formula we want to compute:

$$\sum_{i=A}^{B}\sum_{j=C}^{D} ij$$

Factor out i from the inner sum:

$$\sum_{i=A}^{B} i \sum_{j=C}^{D} j$$

Now, the inner sum is simply $C + (C+1) + (C+2) + \cdots + D$.
But this is just an arithmetic progression! Recall (or derive) the formula for the sum of the first n terms of an arithmetic progression:

$$S_n = \frac{n(a_1 + a_n)}{2}$$

Hence:

$$\sum_{j=C}^{D} j = \frac{(D-C+1)(C+D)}{2}$$

We can substitute this to arrive at:

$$\sum_{i=A}^{B} i \sum_{j=C}^{D} j = \sum_{i=A}^{B} i \frac{(D-C+1)(C+D)}{2}$$

Computing the formula like this gives us an $O(QN)$ algorithm, which passes the fourth subtask.
To solve the fifth and final subtask, both N and M are now up to 10^9. We simply continue:

$$\sum_{i=A}^{B} i \sum_{j=C}^{D} j = \sum_{i=A}^{B} i \frac{(D-C+1)(C+D)}{2}$$

Factor out $\frac{(D-C+1)(C+D)}{2}$:

$$\frac{(D-C+1)(C+D)}{2} \sum_{i=A}^{B} i$$

The remaining sum is simply $A + (A+1) + (A+2) + \cdots + B$. Using the same reasoning, we arrive at the following closed form:

$$\sum_{i=A}^{B} \sum_{i=C}^{D} ij = \frac{(B-A+1)(A+B)(D-C+1)(C+D)}{4}$$

Implementing this formula correctly gives us an $O(Q)$ solution, which should get full points for this problem.

4.4 Remarks

This problem turned out to be much easier for the candidates than expected, with many candidates easily coming up with the aforementioned formula. Many, however, could not implement it correctly due to various pitfalls.
Some of the common mistakes included:

1. Forgetting to take the sum modulo $10^9 + 7$. This only gives points for the first subtask, where the sum is always less than $10^9 + 7$.

2. Using 32-bit integers instead of 64-bit integers.

3. Taking $(B - A + 1)(A + B)(D - C + 1)(C + D)$ modulo $10^9 + 7$ and then dividing by 4 afterwards. This does not work because $\frac{a}{b} \mod m$ is not always equal to $\frac{a \mod m}{b}$. Instead, what one should have done instead is take the product modulo $4 \cdot 10^9 + 28$ and then divide by 4, because $\frac{a}{b} \mod m \equiv \frac{a \mod bm}{b}$. Alternatively, one could have taken the product modulo $10^9 + 7$ and then multiplied the product by 250000002, the modular multiplicative inverse of 4, before printing the resulting product modulo $10^9 + 7$.

4. Computing $\frac{(B-A+1)(A+B)(D-C+1)(C+D)}{4}$ and then taking it modulo $10^9 + 7$. This does not work because the numerator can potentially exceed the range of even a 64-bit integer. The exception to this is Python, where integers are automatically treated with arbitrary precision. One can also avert this issue in Java by using BigIntegers.

5. Using floating point division (/) instead of integer division (//) in Python. While Python integers are naturally arbitrary precision[6], using the floating point division in place of the integer division treats them as floating points, which quickly lose precision and does not give the exact answer.

[6] Indeed, using Python to solve this problem almost seems like cheating ;)

5 Hopscotch 2

5.1 Problem statement

You have an array $A_1, A_2, A_3 \ldots, A_N$. When you are at a given position i, you may jump to another position j if $|i - j| \leq A_i$, $i \neq j$ and $1 \leq j \leq N$.
Answer M queries. In each query, given two integers S and K, how many ways are there to perform K jumps, starting at position S?
Output the number of ways modulo $10^9 + 7$.

5.2 When $K = 1$

The first subtask has $K = 1$ for all the queries. This case is trivial; given the starting position S, we can jump up to A_S positions to the left, and up to A_S positions to the right.
Since we are not allowed to go out of bounds, we should take special consideration for this case. Specifically, there are $S - 1$ positions to the left of position S, and there are $N - S$ positions to the right of position S. So, we have $\min(A_S, S - 1)$ choices for jumping to the left, and $\min(A_S, N - S)$ positions to jump to the right.
Hence the answer to each query is simply $\min(A_S, S - 1) + \min(A_S, N - S)$.

5.3 Dynamic programming

Define $f(p, m)$ to be the number of ways to perform m jumps if we are at position s. The answer to a query is simply $f(S, K)$.
We can use our previous analysis for $m = 1$ as our base case. We have:

$$f(p, 1) = \min(A_p, p - 1) + \min(A_p, N - p)$$

Otherwise, if we are at the position p, we can jump to any position between $\max(1, p - A_p)$ and $\min(N, p + A_p)$, except for p itself. If we jump to a position, we will have $m - 1$ jumps left.
Note that the number of ways to perform m jumps at a given position is simply the number of ways to perform $m - 1$ jumps at the positions we can jump to from here:

$$f(p, m) = \sum_{\substack{i = \max(1, p - A_p) \\ i \neq p}}^{\min(N, p + A_p)} f(i, m - 1)$$

Computing this formula directly gives is too slow, as there are an exponential number of calls, and will likely only pass the second subtask.
In order to improve this algorithm, note that there are only N different possible arguments p, and K different possible arguments m; hence there are only NK states in total. Computing $f(p, m)$ twice for the same p and m is a waste of resources, so we can simply store the results

of $f(p, m)$ in an array when we are done computing them, and then consult this array instead of recomputing if we ever need this value again.
There are NK states and each state is computed in at worst $O(N)$ time, so this runs in $O(N^2K + M)$, which suffices for the third subtask. Note that each query is answered in amortized $O(1)$ time.

5.4 Range sum query

The final subtask has N up to 100000, so this no longer passes.
The trick here is to compute the function bottom-up, instead of top-down. We will compute $f(p, 1)$ for all p, then $f(p, 2)$, then $f(p, 3)$, and so on, until $f(p, 140)$, the maximum possible value of K. Like before, we store their values in an array. We will compute all of these values before we answer any queries.
Recall the formula:

$$f(p, m) = \sum_{\substack{i=\max(1,p-A_p)\\ i\neq p}}^{\min(N,p+A_p)} f(i, m-1)$$

Note that this is really just $f(l, m-1)+f(l+1, m-1)+f(l+2, m-1)+\cdots+f(r, m-1)-f(p, m-1)$ for some l and r. But since we're computing this bottom-up, we already know all of these values, and they are already stored in consecutive elements in an array!
Hence the problem really boils down to being able to quickly compute the sum of a consecutive range. We also need to be able to update single elements in the array quickly, once we have finished computing a value. But this is a classical problem known as the range sum query, which can be solved in $O(\log N)$ time per query using a Fenwick tree. Specifically, we build K Fenwick trees, one for each m, and then we can update one Fenwick tree by querying the other. This runs in $O(NK \log N + M)$ time.
Note that we can in fact solve this problem in $O(NK + M)$ time, without using Fenwick trees, by simply noting that the range query and update prodecures happen separately, and so we can actually solve it using a simple modified cumulative sum approach, building a cumulative sum array on each layer after we have computed all p for that particular m.

6 #DayaangMatuwad

6.1 Problem statement

Given a weighted, undirected graph with N nodes and M edges, answer Q queries of the following form:

- Find the minimum possible maximum length of an edge on a path from vertex a to vertex b, or state that no path exists.

Note that the queries are encrypted, and have to be answered online. That is, you must answer the previous query before you can get the next one.

6.2 Slow solutions

6.2.1 Modified Dijkstra's per query

For the first two subtasks, we can use a sort of brute force. In each query, we can simply perform a modified Dijkstra's algorithm where the cost function is the maximum weight of an edge instead of the sum of all weights. An efficient implementation of this runs in $O(M + N \log N)$ per query, or $O(Q(M+N \log N))$ for all Q queries. This is enough to pass the first two subtasks.

6.2.2 Binary search the answer

An alternative solution for the first two subtasks is to binary search the minimum path length w such that a and b are connected using edges of weight at most w. Checking can be done using a simple depth-first search or breadth-first search. This runs in $O((N + M) \log \max A_i)$ per query. We can also make it run faster, in $O((N + M) \log M)$, by noting that the only weights we have to check are those which correspond to the weight of some edge. This solution, although possibly easier to think of and implement, is unfortunately not as readily extendible as Dijkstra's algorithm for the third subtask.

6.2.3 Preprocessing

For the third subtask, this does not work because Q is up to 100000, and the constant factors associated with both of these solutions are very high. What we can do instead is to precompute, for each source i, the minimum cost paths to all other vertices. This is easily done with the modified Dijkstra's algorithm—it actually gives us, in $O(M + N \log N)$, the minimum cost paths from a single vertex to all other vertices, unlike our binary search algorithm which only gives us the minimum path weight between two specific nodes. We can hence preprocess all pairs using Dijkstra's algorithm just N times, once for each source vertex.
Now, each query can be answered in $O(1)$, just by consulting our preprocessed answers.

The algorithm runs in $O(NM + N^2 \log N + Q)$ and passes the third subtask. Clearly, this no longer works for the last subtask—we still need a lot of observations and the ability to combine many different algorithms.

6.3 Reduction to a tree

Notice that the only edges we ever have to traverse in an optimal path are those which are in the minimum spanning tree of the graph. This can be readily proven using contradiction.
Therefore, let's compute the minimum spanning tree of the graph. We can use Kruskal's algorithm to do this in $O(M \log M)$ time, or Prim's algorithm which runs in $O(M \log N)$.
Note that, in both of these cases, we may actually end up with a forest instead of just a single tree. This is not a big problem; we will assume in the rest of this solution sketch, for simplicity, that we are only dealing with a single tree, as if two nodes are in separate trees then obviously no path exists.
Of course, just because we have reduced it to a tree does not mean we can just use the aforementioned Dijkstra's algorithm; it will still take $O(N \log N)$ time per query! We need to exploit more properties of the tree.

6.4 Lowest common ancestor

The key is to notice that, because we are dealing with a tree, there is exactly one unique path between any two nodes. The problem is hence reduced to finding the maximum weight on this one path, instead of having to find the best path among multiple different paths as in our previous solution!
When dealing with problems related to trees, it is often a good idea to root the tree at some vertex, like vertex 1. Now, every path $u \to v$ is uniquely described as $u \to \text{lca}(u,v) \to v$, where $\text{lca}(u,v)$ is the lowest common ancestor of vertices u and v with respect to vertex 1.
We can, in fact, think of this as two separate paths, one from $u \to \text{lca}(u,v)$ and another from $\text{lca}(u,v) \to v$. As this graph is undirected, we can reverse the latter and get $v \to \text{lca}(u,v)$. Now, both of our paths are going upwards. We now see the two underlying subproblems; efficiently finding the lowest common ancestor of u and v, and getting the maximum edge weight on the path from these nodes to their lowest common ancestor.
Surprisingly, we can actually compute both of these using the same method, and, in fact, simultaneously! The first one will help us compute the second one.

6.4.1 Binary lifting

Let $p(i,j)$ denote the $2^{j\text{th}}$ ancestor of vertex i when the tree is rooted at vertex 1, or -1 if the ancestor doesn't exist. We observe the following base cases:

- $p(-1,j) = -1$

- $p(1, j) = -1$
- $p(i, 0)$ is the parent of i

The recurrence is simple—$p(i, j) = p(p(i, j-1), j-1)$. This is because the $2^{j\text{th}}$ ancestor of a vertex is simply the $2^{(j-1)\text{th}}$ ancestor of the $2^{(j-1)\text{th}}$ ancestor of i.
Note that the only relevant j are those where $2^j < N$. Hence $j < \log N$. Therefore this part can be precomputed in just $O(N \log N)$ time.
Now, we define $w(i, j)$ the same way—it is the maximum weight of any edge in the path from the vertex i to its $2^{j\text{th}}$ ancestor. We can compute it similarly:

- $w(-1, j) = 0$
- $w(1, j) = 0$
- $w(i, 0)$ is the weight of the edge from i to its parent.

We now have $w(i, j) = \max(w(p(i, j-1), j-1), w(i, j-1))$. That is, the maximum weight of an edge in the path from i to its $2^{j\text{th}}$ ancestor is the maximum between the maximum weight of an edge in the path from i to its $2^{(j-1)\text{th}}$ ancestor, and the maximum weight of an edge in the path from the $2^{(j-1)\text{th}}$ ancestor of i to its $2^{j\text{th}}$ ancestor.
Once we have preprocessed these two functions for all relevant values, the computation of the lowest common ancestor and the maximum path weights becomes much simpler.
Let l_1 be the distance of node a from the root, and l_2 the distance of node b from the root. Without loss of generality assume $l_1 \leq l_2$. First, we need to get a and b on the same level (distance from the root). For this, we will "lift" b using $l_2 - l_1$ levels. We can do this in $O(\log N)$ time using our preprocessed p function; start from $j = \log N$ and decrease j until $l_2 + 2^j \leq l_1$. Then, we set b to $p(b, j)$. At the same time, we have to keep track of the current maximum edge weight have already passed; this is the maximum between the current maximum edge weight and $w(b, j)$.
After this, a and b should now be on the same level.
Observe that if $p(a, j) = p(b, j)$, then $p(a, j\prime) = p(b, j\prime)$ for all $j\prime > j$, and if $p(a, j) \neq p(b, j)$, then $p(a, j\prime) \neq p(b, j\prime)$ for all $j\prime < j$. Hence we can perform a pseudo-binary search on the lowest common ancestor of a and b.
First, we start with $j = \log N$, and work our way downwards. If $p(a, j) = p(b, j)$, we should decrease j to find a possibly lower common ancestor; otherwise, if $p(a, j) \neq p(b, j)$, then we should update our current maximum edge weight by comparing our current maximum with $w(a, j)$ and $w(b, j)$, and then lift both a and b by setting $a \leftarrow p(a, j)$ and $b \leftarrow p(b, j)$.
After at most $O(\log N)$ iterations, a and b will correspond to their lowest common ancestor. The correct answer is then in the maximum edge weight among all the edge weights we have encountered.

In total, after computing the minimum spanning tree, this algorithm runs in $O((N+Q)\log N)$; $O(N\log N)$ for preprocessing and then $O(\log N)$ for each of the Q queries. Finally, this is fast enough to solve this problem!

6.5 Other solutions

There are other solutions to this problem, such as heavy-light decomposition and other reductions, with similar or better runtimes. Here, we simply settled for the simplest working solution.[7]

[7]In competitive programming, we usually settle for the simplest solution which passes the given constraints, as there is no benefit to coding more complicated solutions that will just score the same number of points.

7 Trouble at Bunnyburrow

7.1 Problem statement

You have a grid with N rows and M columns. You have to support R queries. Each query takes on either one of the following forms:

- Given A_i and B_i, increase the values of all the cells between rows A_i and B_i by 1.
- Given A_i and B_i, increase the values of all the cells between columns A_i and B_i by 1.

Initially, all cells have a value of 0.
After all R queries, output $R + 1$ integers; the i^{th} integer should contain the number of cells with a value of at least $i - 1$.

7.2 Remark

It is easier to calculate the number of cells with a value exactly $i - 1$, instead of the number of cells with a value at least $i - 1$. We can hence simply compute the former, and then simply cumulate the values in $O(R)$ afterwards. For the succeeding explanations, we will thus just focus on solving the former problem instead of the latter.

7.3 Brute force

The first subtask has N, M and R up to 300. This points to an obvious brute force solution; simply create the a 2D array with N rows and M columns, and then, for each query, increase the values of the corresponding cells.
In the worst case, we will have to increase the values of $O(NM)$ cells in a single update; hence this runs in $O(NMR)$, which is enough only for the first subtask.

7.4 Do away with the grid

For the second subtask, N, M and R can assume values up to 5000, so this is too slow.
For this case, let's just keep two arrays, C and D. C_i corresponds to the number of times the i^{th} column was updated, and D_i corresponds to the number of times the i^{th} row was updated. Now, increasing the values of a number of rows can be done in $O(N)$, while increasing the values of a number of columns can be done in $O(M)$. Hence, the updating phase takes $O(\max(N, M)R)$ time at worst.
When we are done with all the updates, we can simply go through the entire "grid" using two nested loops as before. We can recover the value at cell (i, j) simply using $D_i + C_j$. This part runs in $O(NM)$ time.
Hence our algorithm runs in $O(\max(N, M)R + NM)$, which is sufficient for the second subtask.

7.5 Sweep line

The third subtask now has N and M up to 10^9, so even this will not work any more! Even if we use data structures like implicit segment trees to efficiently perform the updating phase, there is still no way for us to perform the latter half of our algorithm, as we still have to go through each and every cell in $O(NM)$ time. Hence we need a new idea.

Instead of computing the value of each column and each row, let's try another perspective—let's try to compute, for each value, how many columns and rows have this value. Hence we redefine C_i to be the number of columns with a value i, and D_i to be the number of rows with a value i.

Let's focus on computing this for the columns, and then we can perform the same idea to compute it for the rows.

Let's imagine an x-axis. Consider all queries which state "increase the values of the cells between columns A_i and B_i by 1". We will place the point A_i and $B_i + 1$ on this axis, with the A_i point given a value of 1, and the $B_i + 1$ point given a value of -1. After we have placed all of the points on this axis, we will then iterate from the first point on this axis, moving right until we reach the last point, maintaining a running sum as we go right. Whenever we encounter a point, we will simply add its value to our running sum. The value of at any particular point is simply the value of the running sum when we reached that point. Hence we increase the value of C_s by 1, where s is the running sum.

Note that this runs in $O(\max B_i)$, which is too slow. The key is to notice that there are only a few points of interest; namely, the points with values, because the segments between these points will all have the exact same value!

So, we can skip from point to point, only checking the points which have values (and therefore change the sum s), instead of iterating through many redundant points on the line. If we are at the labelled point x, and the previous labelled point was y, then we can add $y - x$ to C_s. Clearly, there are at most $2R$ points on our line, so this only takes $O(R)$ time. Sorting the points in increasing x-coordinate takes $O(R \log R)$ time, so this entire portion runs in $O(R \log R)$, dominated by the sorting.

Suppose we have already found the number of columns which have a value i for all i, and the number of rows which have a value i for all i. How would we be able to find the number of cells for each value?

If we have C_a columns with a value a, and D_b rows with a value b, their intersection forms $C_a D_b$ cells with a value $a + b$. Hence let's sum this value for each $0 \le a \le R$ and $0 \le b \le R$.

If we denote X_i to be the number of cells with value i, we have:

$$X_k = \sum_{i=0}^{R} C_i D_{k-i}$$

where all out of range elements are defined to be 0.

We can use this formula to compute the values of all X_k in $O(R^2)$ time. This suffices to pass the third subtask.

7.6 Efficient polynomial multiplication

The final subtask has R up to 64000, so this is too slow.
In order to speed this up, we will need to note that what we are actually doing is essentially multiplying two polynomials. In essence, we are multiplying polynomials whose coefficients are stored in C and D and then storing the result in R.
There are a number of classical algorithms for efficiently multiplying two polynomials:

7.6.1 Karatsuba algorithm

The Karatsuba algorithm is a divide-and-conquer algorithm, which multiplies two polynomials efficiently. Let's write the polynomials a and b like so:

$$a = a_0 + a_1 x^{\frac{n}{2}}$$
$$b = b_0 + b_1 x^{\frac{n}{2}}$$

Where a_0, b_0, a_1 and b_1 are themselves polynomials of degree $\frac{n}{2}$.
Now, ab can be written:

$$ab = (a_0 + a_1 x^{\frac{n}{2}})(b_0 + b_1 x^{\frac{n}{2}})$$
$$ab = a_0 b_0 + (a_0 b_1 + a_1 b_0) x^{\frac{n}{2}} + a_1 b_1 x^n$$

We can recurse to multiply the resulting polynomials.
Note that there are four multiplications, so this actually still takes $O(n^2)$, which is just as bad as our previous solution.
A better way would be to use only three multiplications.
Let:

$$p = a_0 b_0$$
$$q = a_1 b_1$$
$$r = (a_0 + a_1)(b_0 + b_1)$$

Note how there are only three multiplications here. Expanding $(a_0 + a_1)(b_0 + b_1)$ gives:

$$(a_0 + a_1)(b_0 + b_1) = a_0 b_0 + a_0 b_1 + a_1 b_0 + a_1 b_1$$

But we want $(a_0 b_1 + a_1 b_0)$, so we subtract off $a_0 b_0$ and $a_1 b_1$:

$$(a_0 + a_1)(b_0 + b_1) - a_0b_0 - a_1b_1 = a_0b_1 + a_1b_0$$

If we compute the values p, q and r once, recursively, and then store them, we can thus reconstruct ab using only three recursive multiplications:

$$ab = p + (r - p - q)x^{\frac{n}{2}} + qx^2$$

This runs in $O(n^{\log_2 3})$. In our case, using this polynomial multiplication algorithm gives us a runtime of $O(R^{\log_2 3})$, which should be enough to pass the final subtask. You can refer to some implementations of this algorithm online to get an idea of how it works.

7.6.2 Fast Fourier transform

We remark that a more efficient algorithm exists, namely the fast Fourier transform, which runs in $O(R \log R)$. It requires knowledge of a few advanced algebraic concepts. Although the algorithm is theoretically faster, it comes with a higher constant factor; one might need an efficient implementation to get it to pass the cases. Also, the usual fast Fourier transform implementation uses floating points, which becomes a big problem as we cannot afford to lose any precision here. There are a number of variations in the implementations of this algorithm which will allow one to avert these issues, but will require a lot of work on your part to get it accepted.

8 Papers, Please

8.1 Problem statement

You have a string S of length N. Answer Q queries. In each query, you are given a string T of length L, and you must output whether:

- T is a substring of S;
- T is a subsequence of S, but not a substring;
- T is neither a substring nor a subsequence of S.

The sum of L over all queries does not exceed 10^6.

8.2 Efficient subsequence detection

It is easy to see that there are actually two different problems here; quickly determining whether a string is a substring of another, and quickly determining whether a string is a subsequence of another. The latter is easier to do; we will cover this first.

8.2.1 Greedy

The most straightforward algorithm is as follows. Keep two pointers, i and j. Originally, $i = 1$ and $j = 1$. Check whether $S_i = T_j$. If they match, then increment both i and j. Otherwise, increment i only.
T is a subsequence of S if and only if we are able to match all characters of T.
This runs in $O(N + L)$ time per query, for a total runtime of $O((N + L)Q)$.

8.2.2 Binary search

A better algorithm is as follows. Create 26 arrays, one for each character. Each array will contain the sorted positions i where S_i corresponds the given character. Now, instead of having to iterate through, at most, all the characters of S, we will only need to iterate through T. Keep two pointers i and j, and set $i = 1$ and $j = 1$. At each point, instead of checking through iterating through S, simply binary search the first character k such that $i \leq k$ in the array corresponding to T_j. Afterwards, we can increment both i and j.
This runs in $O(L \log N)$ per query. Although Q is up to 100000, we know that the sum of L in all queries is only up to 10^6, so this should be fast enough with respect to testing for subsequences.

8.2.3 Offline algorithm

There is also a faster algorithm that runs in $O(L)$ per query, which involves taking Q queries in advance and creating 26 queues. We will leave it to the reader to discover this solution.

8.3 Efficient substring detection

It is arguably harder to determine whether a string is a substring of another string quickly. The solutions for this problem are more complicated, but are considered classical; in this sense, some may argue that this subproblem is actually easier.

8.3.1 Brute force

The brute force algorithm is also the most straightforward one; check if T matches $S_1 S_2 S_3 \dots S_L$. If not, check if T matches $S_2 S_3 S_4 \dots S_{L+1}$, and so on, until we find a match or we reach the end of S and we have not found any matches.
If we break as soon as we find a mismatch, this runs in $O(N)$ on average. Consider, however, a case like the following

$$S = \text{AAAAAAAAAAAAAAAA} \dots \text{A}$$
$$T = \text{AAAAAAAAAB}$$

In this case, we will repeatedly match all L characters of T, and then repeatedly find a mismatch at the end, moving only one position to the right in S, then having to compare many characters again. Hence we will end up performing $O(NL)$ operations per query.
In the worst case, this runs in $O(NLQ)$ time, which is very slow. For sure, you can expect that this is in the given test cases.
Note that many programming languages have built-in substring checking methods; most of them use this method, so the built-in method is only sufficient for the smallest cases.

8.3.2 Rolling hash

Let's define the hash $h(s)$ of a string s as follows:

$$h(s) = s_1 b^{n-1} + s_2 b^{n-2} + s_3 b^{n-3} + \cdots + s_n b^0$$

for some prime base b.
This number grows very quickly, so we should take it modulo a large prime, such as $10^9 + 7$. We can compute the hash of a string with n characters in $O(n)$.
Now, let's exploit the fact that if two string hashes are equal, then with a very high probability the strings are also equal.
Let's compute $h(T)$. This takes $O(L)$ time.
Now, we will compute $h(S_1 S_2 S_3 \dots S_L)$. Compare this with the computed value of $h(T)$, if they are equal, then we are done (alternatively, we can check if they are really equal to be sure).
We need to find a way to compute the next hash, $h(S_2 S_3 S_4 \dots S_{L+1})$, without recomputing everything. To do this, we should first subtract off $S_1 b^{n-1}$. Now, multiply the hash by b, and then add S_{L+1}. Take it modulo $10^9 + 7$. This is now the hash of $S_2 S_3 S_4 \dots S_{L+1}$—note that we only had to do $O(1)$ operations to compute the next hash!

Now we can simply compare this to $h(T)$, if they are equal then we are done. Keep doing this for the entire string S. Implemented properly, this runs in $O(N+L)$ per query, or $O((N+L)Q)$ in total.

8.3.3 Suffix array

This is not yet fast enough to get full points, because $O(NQ)$ can be up to 10^{10}.
Let's create the suffix array for S. There are several easy, well-known algorithms to do this in $O(N \log N)$ or $O(N \log^2 N)$ (and a number of more complex ones which even work in $O(N)$). The discussion on how to efficiently create a suffix array is classical and we suggest the reader to look it up on their own.
Recall that a suffix array is essentially a sorted array of suffixes, stored in a compact manner, and that each substring of a string appears as a prefix of one of its suffixes. Hence, given a string T, if we want to check whether it is a substring of S, we can simply perform a binary search over the suffix array!
This runs in $O(L \log N)$ time per query. Combined with an efficient algorithm for subsequence detection, we can solve this problem for full points.[8]

[8] Glory to Arstotzka.