Editorial
Online Selection Contest
BACS Regional Programming Camp, 2018

Editorial

Online Selection Contest
BACS Regional Programming Contest, 2018

Organized by

Bangladesh Advanced Compting Society

Sponsored by

CodeMarshal
www.codemarshal.com

SSL WIRELESS
www.sslwireless.com

Problem A - Contiguous Sequence

Problem setter: Raihat Zaman Neloy, JU
Alternate solution writer 1: Tarango Khan, NSU
Alternate solution writer 2: Aminul Haq, NSU

Subtask 1:
As this problems constraints **N** was only **100**, it can be solved using any bruteforce idea. But the easiest solution is just to check if we can find any **i** in the array such that **A[i] - A[i+1] = 1**, then we just simple output **"Yes"** otherwise **"No"**. Why this works, you can just figure out that using some pen and paper!

Subtask 2:
Naive brute force would pass this subtask. Fix the arrival and departure time of Mr Kashem by running two nested loops. Then check whether the number of overlaps with that time frame is at least **P** or not. You can do this check with a linear search over all persons.
Complexity: Let K be the maximum among all ending times. Then the complexity of this solution is $O(K^2 N)$

Subtask 2:
Key observation is that we'll only have to start or end the range in event points (points where given segments either start or end). There will be **2N** such event points. We'll iterate over them to set the left pointer. And then binary search on right pointer. Finally to check the number of overlap counts we'll run the same linear search we ran in subtask 1.
Complexity: $O(N^2 \log N)$

Subtask 3:
Similar to subtask 2. we'll fix the left pointer by iterating over the event points. Then we'll binary search on event points to fix the right pointer. Now, we'll try to replace the linear search to check overlap with a better approach. We can do two binary searches for this. Let the time frame we are currently checking is $[A, B]$. Then,
$overlap\ count = N - (number\ of\ segments\ that\ start\ after\ B) - (number\ of\ segments\ that\ end\ before\ B)$
The complexity of checking is $O(\log N)$ here.
We can do this in $O(1)$, with linear pre-computation over all event points.
This problem can also be solved with sliding window/ two pointer technique. But to do so, we'll have to sort the event points initially. In this solution, complexity will be dominated by the complexity of the sorting algorithm you are using.
We've tried to allow solutions having a maximum complexity of $O(N \log^2 N)$.
Complexity: $O(N \log N)$ or $O(N \log^2 N)$

Problem B - School Reunion

Problem setter: Sourav Sen Tonmoy, DU
Alternate solution writer 1: Tarango Khan, NSU
Alternate solution writer 2: Md. Kazi Nayeem, SUST

Subtask 1:
Naive brute force would pass this subtask. Fix the arrival and departure time of Mr Kashem by running two nested loops. Then check whether the number of overlaps with that time frame is at least **P** or not. You can do this check with a linear search over all persons.
Complexity: Let K be the maximum among all ending times. Then the complexity of this solution is $O(K^2 N)$

Subtask 2:
Key observation is that we'll only have to start or end the range in event points (points where given segments either start or end). There will be **2N** such event points. We'll iterate over them to set the left pointer. And then binary search on right pointer. Finally to check the number of overlap counts we'll run the same linear search we ran in subtask 1.
Complexity: $O(N^2 \log N)$

Subtask 3:
Similar to subtask 2. we'll fix the left pointer by iterating over the event points. Then we'll binary search on event points to fix the right pointer. Now, we'll try to replace the linear search to check overlap with a better approach. We can do two binary searches for this. Let the time frame we are currently checking is $[A, B]$. Then,
$overlap\ count = N - (number\ of\ segments\ that\ start\ after\ B) - (number\ of\ segments\ that\ end\ before\ B)$
The complexity of checking is $O(\log N)$ here.
We can do this in $O(1)$, with linear pre-computation over all event points.
This problem can also be solved with sliding window/ two pointer technique. But to do so, we'll have to sort the event points initially. In this solution, complexity will be dominated by the complexity of the sorting algorithm you are using.
We've tried to allow solutions having a maximum complexity of $O(N \log^2 N)$.
Complexity: $O(N \log N)$ or $O(N \log^2 N)$

Problem C - A Criminal

Problem setter: Meshab Uddin Tanvir, DU
Alternate solution writer 1: Sourav Sen Tonmoy, DU
Alternate solution writer 2: Shahadat Hossain Shahin, DU

Subtask 1:

Simulate the whole process stated on problem statement for every possible Y (using array, queue or stack) within time limit Complexity: $O(n^3)$.

Subtask 2:

This problem actually reflect the josephus problem.
The main part of this problem is to find Number of possible Y such that $josephus(Y,2) = X$.

After finding this rest part is obvious calculation.

The general solution for $josephus(n,k)$ [wikipedia] with compexity $O(n)$. When $k = 2$ there is special solution which is with complexity $O(\log n)$ [wikipedia].
We can precompute all $josephus(i,2)$ where $1 \le i \le 10^5$. And then can run a loop through the precomputed value to find all Y such that $josephus(Y,2) = X$.

Subtask 3:

Let's see few values of $josephus(i,2)$

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$josephus(i,2)$	1	1	3	1	3	5	7	1	3	5	7	9	11	13	15

We can see that there is a nice pattern here. $josephus(n,2)$ is an increasing odd sequence that restarts with $josephus(n,2) = 1$ whenever the index n is a power of 2.

First we will find minimum value of p such that $josephus(p,2) = X$, we can do this in $O(\log n)$, because for every $q>0$ $josephus(2^q - 1,2) = 2^q - 1$ and $josephus(2^q,2) = 1$. So we can run loop through $q \ge 0$ and find highest value of q such that $josephus(2^q - 1,2) < X$. From the pattern it is obvious that the Pies between 2^q and $2^{q+1} - 1$. As this pattern is an increasing odd sequence a few calculation enough to find p from q.

Now we know the minimum value of P such that $josephus(p,2) = X$ if the first minimum value of P is a and the second minimum value is b, then it is true that $b - a$ is power of 2and this is true for next possible values of P.
From all these observation we can find all possible Y in range $[1,N]$ such that $josephus(Y,2) = X$, and there are at most $log(N)$values.

Complexity: $O(\log n)$

Problem D - Non Super Boring Substring

Problem setter: Tarango Khan, NSU
Alternate solution writer 1: Aminul Haq, NSU
Alternate solution writer 2: Sourav Sen Tonmoy, DU

Subtask 1:
Tag: Bruteforce

We can select each substring(u, v) of the string S in O(N^2) and check how many of those are non super-boring. To check if a substring(u, v) is non super-boring, we can again select each substring(p, q) of length >=k inside the substring(u, v) in O(N^2) and check if it's a palindrome or not in O(N). Overall Complexity: O(N^5)

Subtask 2:
Tag: Dynamic Programming, Two Pointer

Let's define an array DP[][] where DP[u][k] = 1, if the substring(u, u+k) is a palindrome, otherwise DP[u][k] = 0. We can easily pre calculate this DP array using dynamic programming.

Now let's define another array MX[][] where MX[u][k] = length of the largest palindromic substring inside the substring(u, u+k). We can calculate this MX array using similar approach to calculate DP array. To calculate the MX array we can use the following recurrence.
if(DP[u][k] = 1) MX[u][k] = k+1;
else MX[u][k] = max(MX[u][k-1], MX[u+1][k-1]);
Overall Complexity: O(N^2)

Subtask 3:
Tag: Manacher, Hashing, Two Pointer

Let's consider a super-boring substring(u, v) which contains a palindromic substring of length z. Now if z > k+1, then the super boring substring(u, v) must also contain another palindromic substring of length k or k+1. So we only need to consider those palindromic substring whose length is either k or k+1.

Now let's apply manacher. For each position u in the string s, manacher finds the length of the largest palindrome whose center is u.

Using this information we can build an array MXP where:
MXP[u] = k, if substring(u, u+k) is a palindrome,
or MXP[u] = k+1, if substring(u, u+k+1) is palindrome,
otherwise MXP[u] = -1.

Now we iterate from right to the left and use the 2 pointer technique. The 1st pointer is v, which means that we consider the index v as the ending position of a non super-boring substring. The 2nd pointer is u, which means u is the smallest position such that substring(u, v) doesn't contain any palindromic substring of length k or k+1. We can move this two pointers to the left(when necessary) using the information from the MXP array and keep track of the result of how many non super-boring substring are there which ends at position v. Overall Complexity: O(N).

Problem E - GCD and LCM of 3 numbers

Problem setter: Kazi Nayeem, SUST
Alternate solution writer 1: Shahed Shahriar, DU
Alternate solution writer 2: Meshab Uddin Tanvir, DU

Subtask 1:
Select x, y, z using nested loop where x<=y<=z. For each, triple check the formula F(a,b,c) = x*y*z/GCD(x,y,z) and GCD(a,b,c) = GCD(x,y,z).

Subtask 2:
Let, x = a'GCD(a,b,c), y = b/GCD(a,b,c), z = c/GCD(a,b,c). {a, b, c are multiple of GCD(a,b,c)}
So, a = x*GCD(a,b,c), b = y*GCD(a,b,c), c = z*GCD(a,b,c)
Then F(a,b,c) = a*b*c/GCD(a,b,c)
= x*GCD(a,b,c)*y*GCD(a,b,c)*z*GCD(a,b,c)/GCD(a,b,c)
So x*y*z = F(a,b,c)/GCD(a,b,c)^2 = N [Let] ..... (equation 1)
if F(a,b,c) is not a multiple of GCD(a,b,c)^2 then answer is zero.
If x<=y<=z then a<=b<=c.
x,y,z will be divisors of N. If we select x, y from divisors of N where x<=y, we can tell z = N/(x*y). If x<=y<=z and GCD(x,y,z) = 1 then we will add answer.

Subtask 3:
From equation 1,
x*y*z = F(a,b,c)/GCD(a,b,c)^2 = N
x,y,z will be divisors of N and will have GCD(x,y,z) = 1.
Let,
N = $p1^{e1} * p2^{e2} * ..... pn^{en}$ where p1, p2,....pn all are unique prime numbers.
Factors of x,y,z will be subset of p1,p2,..pn. At most 2 of x,y,z can have same prime factors because we want GCD(x,y,z) = 1. Now total combination depends on e1,e2,...,en only. From this we can calculate total number of triples possible. But it does not ensure x<=y<=z. We can divide the result by 3!(3 factorial) or 6.Because 3 numbers can permute in 6 ways if the numbers is different. But previous result calculated less for (x,x,z) types triples. So we have to add extra for those combination.

Problem F - Beauty and The Tree

Problem setter: Md Shahadat Hossain Shahin, DU
Alternate solution writer 1: Meshab Uddin Tanvir, DU
Alternate solution writer 2: Aminul Haq, NSU

Subtask 1:

Tag : Brute Force

For subtask 1 , **1 <= N <= 1000** and every node has at most one child and **1 <= P_i <= i** . So the tree basically becomes a line which can be represented by a linear array. Removing subtree of a node **v** means cutting the range **[v,N]**. The beauty of the tree after removing the node v will be maximum value of the range **[1,v-1]**. The maximum of some range can be determined in **O(N)** by simple looping or in **O(1)** by doing some preprocessing.

Subtask 2:

Tag : DFS/BFS

The tree is not a line anymore. But **1 <= N <= 1000**. If no subtree is removed from the tree, we can get the levelwise beauty values by a tree traversal (bfs/dfs). This will take **O(N)** time.
When a subtree is removed it just ignore the subtree in our traversal, the rest is same as previous. This also takes **O(N)**.
Overall complexity per test case : **O(N^N)**

Subtask 3:

Tag : DSU on Tree/ Mo's Algorithm

Let **BL[i]** denote the beauty value of the level **i**.
Beauty of the total tree is maximum value from the **BL** array.

DSU on Tree Technique

Now, the subtree removal operation can be done using DSU on Tree algorithm which does some operation for a subtree in **O(N * logN)**. While removing any node from the total solution, we also need to keep track of the maximum value of the **BL** array. As there is continuous addition and removal operation in DSU on Tree, we need extra **O(logN)**.
Overall complexity per test case : **O(N * logN * logN)**

Mo's Algorithm

The subtree removal can also be done using Mo's algorithm. We run a dfs first. If we build an array **A** of size N and **A[i]** holds a the node having starting time **i**, subtree of every node in the tree can be represented as a continuous interval of the array. Mo's algorithm can process continuous intervals. There will be **N+1** intervals to process. Complexity will be **O(N * sqrtN)**.
For keeping track of the maximum value of the **BL** array, we will need extra **O(logN)**.
Overall complexity per test case : **O(N * sqrtN * logN)**

Problem G - Little T2 and Derangements

Problem setter: Shahed Shahriar, DU
Alternate solution writer 1: Kazi Nayeem, SUST
Alternate solution writer 2: Tarango Khan, NSU

Subtask 1: N can be at most 4. So, we can take a pen and paper, write down all the permutations and for each permutation check whether it is a derangement or not. Also we can use backtrack or use the built-in next_permutation() function to generate all the permutations.
For N = 2, we can get 2 permutaions and 1 of them is derangement.
For N = 3, we can get 6 permutaions and 2 of them are derangement.
For N = 4, we can get 24 permutaions and 9 of them are derangement.

Subtask 2: N can be at most 18 and K can be at most 2e15. For N = 18, we can get more derangements than 2e15. We can write a bitmask DP to generate the number of derangements of N or we can simply use the formula !n = (n-1)![!(n-1)+!(n-2)] where !n denotes the number of derangements for n and !0 = 1 and !1 = 0.
Now we can solve the problem with bitmask DP. The state of the DP function is "mask". This is an N bit number that represents the availability of the numbers. If the i'th bit is 1 then the number i is already put somewhere. If the i'th bit is 0 then we can still put the number i somewhere. By seeing the mask we can know the number of items have been positioned and the current position where we are wanting to put something. The current position is (1 + number of 1 bit of mask). We iterate over all the available numbers(remember, we can not put x in the x'th position) and put the number in current position and call the next state by changing the mask. If we can put every number successfully, this is a derangement of N. This is how we can get the number of derangements of N.
But our problem wants the lexicographically K'th derangement of N. This can be found from this DP. In the first position we put the lexicographically smallest available number and check the number of derangements possible further. If this value is less than K then we can reduce our K by this value as these derangements will never be considered and they all are lexicographically smaller than out expected derangement. Now, we try with the next possible smallest available number and check the number of derangements possible further. If this value is less than K we do the same as above. But if this value is equal or greater than K we know we can put this number in this position and all the further derangements are possible with our generated prefix. Now we try the second position same as above. This is how we will get the lexicographically K'th derangement of N. As K will always be valid and you get a derangement at last, the remaining K will be 1.

Subtask 3: N can be at most 100 and K can be at most 2e15. Here is an easy observation. If we are told to generate the K'th permutation, we can easily observe that the change will occur only with a few large numbers. We can use that observation here too. Only last 18/19 numbers are important.
Lets call this number of important numbers, M. M can be either 18 or 19 here. We are discussing the solution for N > M.
Lets put the first N-M numbers at their actual position. We know the number of derangements for M is always greater than maximum K. Now what to do with this remaining N-M numbers. Lets say N-M is even. We can take first two numbers and swap them. Then we can take the 3rd and 4th number and swap them. Then we can take the 5th and 6th number and so on. This is how we can get the lexicographically K'th derangement for any N. If N is odd we set M = 19 or if N is even we set M = 18. So, for each test case the complexity is O(M^2 * M). But for 1000 test cases this will take so much time. But we only need to calculate for each M once, no matter what is N. So, regardless of T and N the complexity of this solution is **O(M*M^2*M)**.

Problem H - Ray Ray Array

Problem setter: Aminul Haq, NSU
Alternate solution writer : Shahed Shahriar, DU

Subtask 1: A O(N^Q) bruteforce will get AC.

Subtask 2: The trick here is, the order of values won't change after each update. So first sort the array. Now if you subtract **Y** from smaller values or add Y to larger values, the array will still be sorted, their order won't change. So this can be solved easily with Segment Tree lazy propagation in O(Q*log(n)).

Subtask 3: We need a treap(with lazy propagation) to solve this task. For non-negative Y simply do lazy update (just like subtask-2, but in treap). And for negative Y, remove those numbers (less/equal or greater/equal to **X**) from treap, and re-insert them after adding/subtracting with Y. At most N numbers will be removed and re-inserted in the treap for negative Y. This can also be done in O(Q*log(n)).