CRACK A HACK

Factorial Array

Using Square Root Decomposition Technique

COURSE: ALGORITHMIC PROBLEM SOLVING

COURSE CODE: 17ECSE309

BY: NAME: ANIKET SATISHCHANDRA PASTE

USN: 01FE15BCS027

1. PROBLEM:

As part of your education on machine learning, you have to learn combinatorics first. After being taught why the factorial!, comes up everywhere in combinatorics, you are given the following challenge problem.

Given an array of length, you need to perform operations. There are three types of operations:

1. Given, l, r increase the values at all indices between l and r (inclusive) by 1.
2. Given, l, r compute the sum of A_i! For all i from l to r, inclusive. Print this value modulo 10^9.
3. Given, l, r set the value at index i to v , i.e., set A_i to v.

Input Format

The first line of input contains two space-separated integers' n and m.

The second line contains n space-separated integers A_1, A_2, A3 … A denoting the initial contents of the array.

The next lines describe the operations in order.

1. For an operation of type 1, the line contains three integers 1, l, andr.
2. For an operation of type 2, the line contains three integers 2, l, and r.
3. For an operation of type 3, the line contains three integers 3, l, and r.

2. CORE LOGIC/ONE WAY OF APPROACH:

The above problem can be solved using Square Root Decomposition Technique. Sqrt (or Square Root) Decomposition Technique is one of the most common query optimization technique used by competitive programmers. This technique helps us to reduce Time Complexity by a factor of **sqrt(n)**.

The key concept of this technique is to decompose given array into small chunks specifically of size sqrt(n).

Let's say we have an array of n elements and we decompose this array into small chunks of size sqrt(n). We will be having exactly sqrt(n) such chunks provided that n is a perfect square. Therefore, now our array on n elements is decomposed into sqrt(n) blocks, where each block contains sqrt(n) elements (assuming size of array is perfect square).

Let's consider these chunks or blocks as an individual array each of which contains sqrt(n) elements and you have computed your desired answer(according to your problem) individually for all the chunks. Now, you need to answer certain queries asking you the answer for the elements in range l to r (l and r are starting and ending indices of the array) in the original n sized array.

The **naive approach** is simply to iterate over each element in range l to r and calculate its corresponding answer. Therefore, the Time Complexity per query will be O(n).

Sqrt Decomposition Trick : As we have already precomputed the answer for all individual chunks and now we need to answer the queries in range l to r. Now we can simply combine the answers of the chunks that lie in between the range l to r in the original array. So, if we see carefully here we are jumping sqrt(n) steps at a time instead of jumping 1 step at a time as done in naive approach. Let's just analyze its Time Complexity and implementation considering the below problem :-

```
Problem:
Given an array of n elements. We need to answer q
Queries telling the sum of elements in range l to
r in the array. Also the array is not static i.e
the values are changed via some point update query.

Range Sum Queries are of form : Q l r ,
where l is the starting index r is the ending
index

Point update Query is of form : U idx val,
where idx is the index to update val is the
updated value
```

Another point that should be kept in mind. In order to make the understanding more simple, let me write out the first few iterations.

```
1 1 1
2 2 2
3 6 6
4 24 24
5 120 120
6 720 720
7 5040 5040
8 40320 40320
9 362880 362880
10 3628800 3628800
11 39916800 39916800
12 479001600 479001600
13 227020800 6227020800
14 178291200 87178291200
15 674368000 1307674368000
16 789888000 20922789888000
17 428096000 355687428096000
18 705728000 6402373705728000
19 408832000 121645100408832000
20 176640000 2432902008176640000
```

```
21 709440000 51090942171709440000
22 607680000 1124000727777607680000
23 976640000 25852016738884976640000
24 439360000 620448401733239439360000
25 984000000 15511210043330985984000000
26 584000000 403291461126605635584000000
27 768000000 10888869450418352160768000000
28 504000000 304888344611713860501504000000
29 616000000 8841761993739701954543616000000
30 480000000 265252859812191058636308480000000
31 880000000 8222838654177922817725562880000000
32 160000000 263130836933693530167218012160000000
33 280000000 8683317618811886495518194401280000000
34 520000000 295232799039604140847618609643520000000
35 200000000 10333147966386144929666651337523200000000
36 200000000 371993326789901217467999448150835200000000
37 400000000 13763753091226345046315979581580902400000000
38 200000000 523022617466601111760007224100074291200000000
39 800000000 20397882081197443358640281739902897356800000000
40 0 815915283247897734345611269596115894272000000000
41 0 33452526613163807108170062053440751665152000000000
42 0 1405006117752879898543142606244511569936384000000000
43 0 60415263063373835637355132068513997507264512000000000
44 0 2658271574788448768043625811014615890319638528000000000
45 0 119622220865480194561963161495657715064383733760000000000
46 0 5502622159812088949850305428800254892961651752960000000000
47 0 258623241511168180642964355153611979969197632389120000000000
48 0 12413915592536072670862289047373375038521486354677760000000000
49 0 608281864034267560872252163321295376887552831379210240000000000
50 0 30414093201713378043612608166064768844377641568960512000000000000
```

Consider the above output as x, y, z. For the above problem, we can ignore all x! Where x>40.
The reason for my statement is third number on each line is the factorial of the first, while the second number is the factorial modulo 10^9.

When you calculate 40!, you multiply 20 even numbers together, so 40! must be a multiple of 2^20. You also multiply 8 multiples of 5 together, so 40! must be a multiple of 5^8. Since one of those multiples was 25, 40! is actually a multiple of of 5^9. Now (2^9) * (5^9) = 10^9, so 40! is a multiple of 10^9.

Since larger factorials are also multiples of 40!, they are also multiples of 10^9.

3. MY SOLUTION in C++:

```
#include <bits/stdc++.h>

using namespace std;
#define M 1000000000
long long buc[325][100];
long long str[100];
long long SQ=320;
long long use[325];
long long ora[100009],n;

void update(int l,int r){
    int bl=l/SQ,br=r/SQ;
    if(bl==br){
        for(int i=l;i<=r;i++){
            if(ora[i]+use[bl]<40){
                buc[bl][ora[i]+use[bl]]--;
                ora[i]+=1;
                buc[bl][ora[i]+use[bl]]++;
            }
        }
    }
    else{

        for(int i = l; i < min((bl + 1) * SQ, n); ++i){
            if(ora[i]+use[bl]<40){
            buc[bl][ora[i]+use[bl]]--;
                ora[i]+=1;
                buc[bl][ora[i]+use[bl]]++;
            }
        }
        for(int i = bl + 1; i < br; ++i){
            use[i] += 1;
            for(int j=39;j>0;j--)buc[i][j]=buc[i][j-1];
    }
        for(int i = br * SQ; i <= r; ++i){
            if(ora[i]+use[br]<40){
                buc[br][ora[i]+use[br]]--;
                ora[i]+=1;
                buc[br][ora[i]+use[br]]++;
            }
        }
    }
}

void ans(int l,int r){

    int bl=l/SQ,br=r/SQ;

    if(bl==br){

        for(int i=l;i<=r;i++){
        if((ora[i]+use[bl])<40)str[ora[i]+use[bl]]++;
        }
    }
    else{

        for(int i = l; i < min((bl + 1) * SQ, n); ++i){
        if((ora[i]+use[bl])<40)str[ora[i]+use[bl]]++;
        }
        for(int i = bl + 1; i < br; ++i){
            for(int j=0;j<40;j++)str[j]+=buc[i][j];
        }
        for(int i = br * SQ; i <= r; ++i){
        if((ora[i]+use[br])<40)str[ora[i]+use[br]]++;}
                }

}

int main() {

    long long fac[100001];
    fac[1]=1;
    for(int i=2;i<=100000;i++)fac[i]=(fac[i-1]*i)%M;

    int q;
    scanf("%lld %d",&n,&q);
    long long int a[n];



    for(int i=0;i<n;i++){
        scanf("%lld",&a[i]);
        ora[i]=a[i];
        if(a[i]>=40)continue;
        else buc[i/SQ][a[i]]++;}


    for(int i=0;i<q;i++){
        int t,l,b;
        scanf("%d %d %d",&t,&l,&b);
        if(t==1){

            update(l-1,b-1);
        }
        if(t==2){

            memset(str,0,sizeof str);
            ans(l-1,b-1);
            long long an=0;
            for(int i=1;i<40;i++){
                long long temp=(str[i]*fac[i]) % M;
                an=(an+temp) % M;

            }
            cout<<an<<endl;
        }
        if(t==3){
            l--;
            if(ora[l]+use[l/SQ]<40)buc[l/SQ][ora[l]+use[l/SQ]]--;
            if(b<40)buc[l/SQ][b]++;
            ora[l]=b-use[l/SQ];
        }
    }


    return 0;
}
```

4. TIME COMPLEXITY:

In the worst case our range can be 0 to n-1(where n is the size of array and assuming n to be a perfect square). In this case all the blocks are completely overlapped by our query range. Therefore,to answer this query we need to iterate over all the decomposed blocks for the array and we know that the number of blocks = sqrt(n). Hence, the complexity for this type of query will be O(sqrt(n)) in worst case.

5. APPLICATIONS:

The above method can be applied to almost any kind of array problem where square root of n is an integer. Using this method we can achieve the efficiency of O(sqrt(n)).

6. REFERENCES :

- https://www.hackerrank.com/contests/world-codesprint-12/challenges/factorial-array/forum
- https://www.youtube.com/watch?v=gWbDocYhwDA
- http://codeforces.com/blog/entry/16883