General Ideas

md.mottakin.chowdhury

November 2018

1 GCD on subsegments

Assume you have set of numbers in which you add elements one by one and on each step calculate *gcd* of all numbers from set. Then we will have no more than $log(a[i])$ different values of *gcd*. Thus you can keep compressed info about all *gcd* on Subsegments of $a[i]$:

```
    int a[n];
    map<int, int> sub_gcd[n];
    /*
    Key is gcd,
    Value is the largest length such that gcd(a[i - len], ..., a[i]) equals to key.
    */
    sub_gcd[0][a[0]] = 0;
    for(int i = 1; i < n; i++)
    {
        sub_gcd[i][a[i]] = 0;
        for(auto it: sub_gcd[i - 1])
        {
            int new_gcd = __gcd(it.first, a[i]);
            sub_gcd[i][new_gcd] = max(sub_gcd[i][new_gcd], it.second + 1);
        }
}
```

2 From Static Set to Expandable via $O(log(n))$

Assume you have some static set and you can calculate some function f of the whole set such that $f(x_1, ..., x_n) = g(f(x_1, ..., x_{k-1}), f(x_k, ..., x_n))$, where g is some function which can be calculated fast. For example, $f()$ as the number of elements less than k and $g(a + b) = a + b$. Or $f(S)$ as the number of occurences of strings from S into T and g is a sum again.

With additional $log(n)$ factor you can also insert elements into your set. For this let's keep $log(n)$ disjoint sets such that their union is the whole set. Let the size of k^{th} be either 0 or 2^k depending on binary presentation of the whole set size. Now when inserting element you should add it to 0^{th} set and rebuild every set keeping said constraint. Thus k^{th} set will tell $F(2^k)$ operations each 2^k steps where $F(n)$ is the cost of building set over n elements from scratch which is usually something about n.

3 XOR Subsets

Assume you have set of numbers and you have to calculate something considering xors of its subsets. Then you can assume numbers to be vectors in k-dimensional space over field Z_2 of residues modulo 2. This interpretation useful because ordinary methods of linear algebra work here. For example, here you can see how using gaussian elimination to keep basis in such space and answer queries of k^{th} largest subset xor:

```
    // Problem: in each query, either add a number to the set, or find the k-th
    // largest subset xor from the set. So if the set is currently {1,2} then subset // xors are {0,1,2,3
```

```
int b[32];
int sz = 0;

void add(int x)
{
    for(int i = 0; i < sz; i++)
        if((x ^ b[i]) < x)
            x ^= b[i];
    for(int i = 0; i < sz; i++)
        if((x ^ b[i]) < b[i])
            b[i] ^= x;
    if(x)
{
    b[sz++] = x;
        for(int i = sz - 1; i; i--)
            if(b[i] < b[i - 1])
                swap(b[i], b[i - 1]);
    }
}
int get(int k)
{
    k--;
    int ans = 0;
    for(int i = 0; i < sz; i++)
        if((k >> i) & 1)
            ans ^= b[i];
    return ans;
}
int main()
{
    int n;
    cin >> n;
    while(n--)
    {
        int t, x;
        cin >> t >> x;
        if(t == 1)
            add(x);
        else
            cout << get(x) << "\n";
    }
}
```

4 Cycles in Graph as Linear Space

Assume every set of cycles in graph to be vector in E-dimensional space over Z_2 having one if corresponding edge is taken into set or zero otherwise. One can consider combination of such sets of cycles as sum of vectors in such space. Then you can see that basis of such space will be included in the set of cycles which you can get by adding to the tree of depth first search exactly one edge. You can consider combination of cycles as the one whole cycle which goes through 1-edges odd number of times and even number of times through 0-edges. Thus you can represent any cycle as combination of simple cycles and any path as combination as one simple path and set of simple cycles. It could be useful if we consider pathes in such a way that going through some edge twice annihilates its contribution into some final value. Example: find path from vertex u to v with minimum xor-sum.

5 Matrix Exponentiation Optimization

Assume we have $n \times n$ matrix A and we have to compute $b = A^m x$ several times for different m. Naive solution would consume $O(qn^3 log(n))$ time. But we can precalculate binary powers of A and use $O(log(n))$ multiplications of matrix and vector instead of matrix and matrix. Then the solution will be $O((n^3+qn^2)log(n)$.

6 Euler Tour Technique

You have a tree and there are lots of queries of kind add number on subtree of some vertex or calculate sum on the path between some vertices.

Let's consider two euler tours: in first we write the vertex when we enter it, in second we write it when we exit from it. We can see that difference between prefixes including subtree of v from first and second tours will exactly form vertices from v to the root. Thus problem is reduced to adding number on segment and calculating sum on prefixes.

7 From Expandable Set to Dynamic via $O(log(n))$

Assume for some set we can make non-amortized insert and calculate some queries. Then with additional $O(log(n))$ factor we can handle erase queries. Let's for each element x find the moment when it's erased from set. Thus for each element we will wind segment of time $[a, b]$ such that element is present in the set during this whole segment. Now we can come up with recursive procedure which handles $[l, r]$ time segment considering that all elements such that $[l, r] \subset [a, b]$ are already included into the set. Now, keeping this invariant we recursively go into $[l, m]$ and $[m, r]$ subsegments. Finally when we come into segment of length 1 we can handle the query having static set.

8 Counting the Number of Ways for Reaching a Vertex

You are given an unweighted directed graph (may contain multiple edges) containing N vertices ($1 \leq N \leq 200$) and an integer $b(1 \leq b \leq 10^9)$. You are also given Q queries ($1 \leq Q \leq 10^5$). For each query you are given two vertices u and v and you have to find the number of ways for reaching vertex v starting from u after exactly b steps. (A step is passing through an edge. Each edge may be passed multiple number of times).

Let $M1$ be a matrix where $M1[i][j]$ equals the number of edges connecting vertex i to vertex j. Let $M2$ be $M1$ raised to the power of $b(M1^b)$. Now for any pair u and v, the number of ways for reaching vertex v starting from u after b steps is $M2[u][v]$.

9 Shortest path with a specified number of steps

You are given a weighted graph containing N vertices ($1 \leq N \leq 200$) and an integer $b(1 \leq b \leq 10^9)$. You are also given Q queries ($1 \leq Q \leq 10^5$). For each query you are given two vertices u and v and you have to find the minimum cost for reaching vertex v starting from u after exactly b steps. (A step is passing through an edge. Each edge may be passed multiple number of times).

Let $M1$ be a matrix where $M1[i][j]$ equals the cost of passing the edge connecting i to j (infinity if there is no edge). Let $M2$ be $M1$ raised to the power of b (but this time using the distance product for multiplication). Now for any pair u and v, the minimum cost for reaching vertex v starting from u after b steps is $M2[u][v]$.

What is distance product?

Min-plus matrix multiplication, also known as the distance product, is an operation on matrices.

Given two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$, their distance product $C = (c_{ij})$ is defined as an $n \times n$ mtarix such that:

$$c_{ij} = \min_{k=1...n} (a_{ik} + b_{kj})$$

10 Kirchoff's Theorem

Kirchhoff's theorem provides a way to calculate the number of spanning trees of a graph as a determinant of a special matrix.

To calculate the number of spanning trees, we construct a *Laplacean Matrix* L, where $L[i,i]$ is the degree of node i and $L[i,j] = -1$ if there is an edge between nodes i and j, and otherwise $L[i,j] = 0$.

It can be shown that the number of spanning trees equals the determinant of a matrix that is obtained when we remove any row and any column from L. The determinant is always the same, regardless of which row and column we remove from L.

11 Cayley's Formula

Cayley's formula states that there are n^{n-2} labeled trees that contain n nodes. The nodes are labeled $1, 2, \ldots, n$ and two trees are different if either their structure or labeling is different.

12 Verifying Matrix Multiplication

We are given three matrices A, B and C. The task is to verify if $AB = C$ holds. Of course we can do $O(n^3)$ multiplication but it's possible to do better than that.

We can solve the problem using a Monte Carlo algorithm whose time complexity is only $O(n^2)$. The idea is simple: we choose a random vector X of n elements, and calculate the matrices ABX and CX. If $ABX = C$, we report that $AB = C$.

It can be possible that the algorithm might give wrong answer. To ensure more correctness, we can consider multiple random vectors X and do the above operation.

13 Graph Coloring

Given a graph that contains n nodes and m edges, our task is to find a way to color the nodes of the graph using two colors so that for at least $m/2$ edges, the endpoints have different colors.

The problem can be solved using a Las Vegas algorithm that generates random colorings until a valid coloring has been found. In a random coloring, the color of each node is independently chosen so that the probability of both colors is $1/2$. In a random coloring, the probability that the endpoints of a single edge have different colors is $1/2$. Hence, the expected number of edges whose endpoints have different colors is $m/2$. Since it is expected that a random coloring is valid, we will quickly find a valid coloring in practice.

14 Batch Processing

We have a grid with n cells where initially all of the cells are white except one. We perform $n-1$ operations, each of which first calculates the minimum distance from a given white cell to a black cell, and then paints the white cell black.

The solution is to dividing the operations into $\sqrt{n}$ batches. each of which consists of $\sqrt{n}$ operations.

Initially, we calculate the distance of each white cell from the black cell using BFS and initialize a list of size $\sqrt{n}$ to store latest black colored cells. When we are given the operations, we first find the distance of the white cell from the black cells stored in the list. The answer for the white cell will be the distance found from this checking and also the distance found in BFS. We color this white cell black and insert it into the list.

When the size of the list is more than $\sqrt{n}$, we run BFS again from the black cells and calculate minimum distance for each white cell by multi-source BFS and empty the list.

We run BFS at most $O(\sqrt(n))$ times. And when we check linearly the distance between the black cells stored in the list and the given white cell, we can do this at most $O(\sqrt{n})$ times because its size is not more than $\sqrt{n}$. So the total complexity is $O(n\sqrt{n})$.

15 Knapsack with Sqrt Decomposition

Suppose that we are given a list of integer weights whose sum is n. Our task is to find out all sums that can be formed using a subset of the weights.

By using the fact that there are at most $\sqrt{n}$ distinct weights, we can process the weights in groups that consists of similar weights. We can process each group in $O(\sqrt{n})$ time which yields an $O(n\sqrt{n})$ time algorithm.

The idea is to use an array that records the sums of weights that can be formed using the groups processed so far. The array contains n elements: element k is 1 if the sum k can be formed and 0 otherwise. To process a group of weights, we scan the array from left to right and record the new sums of weights that can be formed using this group and the previous groups.

```
// dp[i] = can we make sum i
for (int i = 1; i <= n; i++)
    dp[i] = -1;
// v contains sqrt(n) different values
for (int z = 0; z < v.size(); z++)
{
    int len = v[z].first; // value of a weight
    int cnt = v[z].second; // occurrence count of a weight
    for (int x = 0; x + v[z].first <= n; x++)
    {
        int y = x + len; // let's build value y
        if (dp[x] != -1 && dp[y] == -1)
        {
            if (pr[x] != len)
                dp[y] = 1, pr[y] = len;
            else
                if (dp[x] < cnt)
                    dp[y] = dp[x] + 1, pr[y] = len;
        }
    }
}
```

16 String Construction

Given a string s of length n and a set of strings D whose total length is m, consider the problem of counting the number of ways s can be formed as a concatenation of strings in D. For example, if $s = ABAB$ and $D = A, B, AB$, there are 4 ways.

Let's assume $count(k)$ denotes the number of ways to construct prefix $s[0 \ldots k]$.

We can solve the problem by using string hashing and the fact that there are at most $O(\sqrt{m})$ distinct string lengths in D. First, we construct a set H that contains all hash values of the strings in D. Then, when calculating a value of $count(k)$, we go through all values of p such that there is a string of length p in D, calculate the hash value of $s[k-p+1 \ldots k]$ and check if it belongs to H. Since there are at most $O(\sqrt{m})$ distinct string lengths, this results in an algorithm whose running time is $O(n\sqrt{m})$.