COMPETITIVE PROGRAMMING IN C++

SIDDHANT CHAUDHARY

Abstract. This is a collection of code snippets and various other things that I use in competitive programming.

Contents

Date: December 2019.

1. An Amazing Source

The link https://cp-algorithms.com/ is one of the best that I've seen for learning CP algorithms. Check out https://cses.fi/problemset/ for some problems.

2. Basic IO and Code Environment

First and foremost, include these statements at the top of your code:

```
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll
```

and here is the reason why: this header has everything you will ever need in a contest format. The input limit never goes beyond `long long`, and `ll` is a nice way to define it.

Now, I will discuss about how I go about IO in contests. Instead of using the generic `cin` method, I use the `printf` and `scanf` methods from C, which are much more flexible and quicker (it is important that you master input/output for contests).
The use of `printf` is easy. A code snippet for `scanf` is given below.

```
char name[50]; int age;
scanf("%s %d" , name , &age);
printf("The name is %s and the age is %d" , name , age);
```

An importnat thing to be noted is the `&` in the argument for `scanf`; `scanf` requires pointer arguments. Because `name` is already a pointer (we have defined it as an array), we don't put an `&` before it.

To redirect input/output to files, the following two lines of code do the trick:

```
    freopen("input.in" , "r" , stdin);
    freopen("output.out" , "w" , stdout);
```

Finally, to make I/O faster, you can add the following two lines to the beginning of the code.

```
ios::sync_with_stdio(0);
cin.tie(0);
```

Sometimes the input is given in a way in which we have to manually detect the *end of file*. Suppose are input file is `input.in`, and we want to give the output a file named `output.out`. Then, we can do something as given below.

```
//program to copy contents of input.in to a new file
//named output.out

char temp_string[100];
FILE * inpFile; FILE * outFile;
inpFile = fopen("input.in" , "r");
outFile = fopen("output.out" , "w");

while(!feof(inpFile)){ //while the end of file is not reached
    fscanf(inpFile , "%s" , &temp_string);
    fprintf(outFile , "%s " , temp_string);
}
```

You can check out the documentation for more functions for file handling.

3. C++ Tricks

3.1. **Auto Keyword.** This is one of the most important keywords for *range based loops* in C++. The following example should be clear.

```
vector <int> a;
a.push_back(0);
a.push_back(1);
```

```
a.push_back(2);
a.push_back(3);
for (auto i : a)
    cout << i << " "; //prints the contents of a
```

3.2. **The complexity of vectors in C++.** To be completed. Need to write about the average and worst case complexity of the pushback function in vectors in C++.

3.3. **STL size function horror.** This is a very minor detail that I learned the hard way: in any STL data structure, the `size()` function returns an *unsigned integer*. This is important: suppose you have a vector named `stack`, whose size is zero. Then, suppose you do the following.

```
long long int y = stack.size() - 1
```

Then, `y` will *not* be 0. Instead, it will be a different integer, because `stack.size()` returns an unsigned integer. To fix this, do the following.

```
long long int y = (int)stack.size() - 1
```

3.4. **Arrays over vectors!** In contests, always try to use arrays over vectors as much as possible. This is because arrays are much faster in terms of time as compared to vectors.

4. Bit Manipulation

4.1. **AND Operation.** Do this by writing: `x&y`. Gives the number whose bits are the AND of the bits of x, y.

4.2. **OR Operation.** Do this by writing: `x|y`. Gives the number whose bits are the OR of the bits of x, y.

4.3. **XOR Operation.** Do this by writing: `x^y`. Gives the number whose bits are the XOR of the bits of x, y.

4.4. **NOT Operation.** Do this by writing: $\sim$`x`. Gives the number whose bits are the NOT of the bits of x. The formula is $\sim$`x = -x - 1`, as signed bits are used in C++.

4.5. **Bit Mask.** The number `1<<k` has a 1 as its k^{th} bit, and all other bits are zero. This can be used to extract the k^{th} bit of any number. Some techniques.

- `x| (1 << k)`: set the k^{th} bit to 1.
- `x & ~ (1 << k)` : set the k^{th} bit to 0.
- `x ^ (1 << k)`: invert the k^{th} bit.

To create a `long long` bit mask, just do `1LL << k`.

4.6. **In Built Bit Functions.** Here are some useful inbuilt functions (an `ll` suffix is added after every function to make the functions work for `long long int` type).

- `__builtin_clzll(x)` : the number of zeros at the beginning of the bit representation.
- `__builtin_ctzll(x)` : the number of zeros at the end of the bit representation.
- `__builtin_popcountll(x)` : the number of ones in the bit representation.
- `__builtin_parityll(x)` : the parity (even or odd) of the number of ones in thebit representation.

5. Sorting

5.1. **The Sort Function.** As an example, we can sort a `vector` in C++ as follows.

```
vector <int> v = {4 , 2 , 5 , 3 , 5 , 8 , 3};
sort(v.begin() , v.end());
```

`v.begin()` returns a pointer to the first element of the vector `v`, and `v.end()` returns a pointer to the theoretical element that follows the last element of the vector `v`. To sort in the reverse order, we can do the following.

```
vector <int> v = {4 , 2 , 5 , 3 , 5 , 8 , 3};
sort(v.rbegin() , v.rend());
```

`v.rbegin()` and `v.rend()` refer to *reverse pointers*.

5.2. **Comparison Operators in C++.** In C++, whenever there is a need for a *ordering* on a particular data structure, we can define a *comparison operator* that can be used for the ordering. For instance, consider the following.

```
struct name_of_structure{
    int x , y;
    bool operator < (const name_of_structure &p){
        if(x == p.x) return y < p.y;
        else return x < p.x;
    }
};
```

Above, we have made a new structure `name_of_structure` which has a comparison operator < that sorts first by the x-coordinate, then by the y-coordinate.

5.3. **Comparison Functions.** We can also pass a comparison function as an argument to `sort`. For example we can do the following to compare strings in length-lexicographic order.

```
bool comp(string a , string b){
    if (a.size() == b.size()) return a < b;
    else return a.size() < b.size();
}
//a vector v of string is sorted as follows
sort(v.begin() , v.end() , comp);
```

5.4. **Binary Search.** Here is a simple way of implementing binary search.

```
int ans = -1;
while(lo <= hi){
    int mid = lo + (hi - lo)/2;

    if(arr[mid] == element){
        ans = mid;
    }
    if(arr[mid] < element){
        lo = mid + 1;
    }
    else{
        hi = mid - 1;
    }
}
```

5.5. **Binary Searching the Answer.** Sometimes we are given a problem in which we can binary search the answer in a given interval $[L, R]$. For any $x \in [L, R]$, suppose the predicate $\phi(x)$ indicates whether x is a valid solution to our answer or not. Also, suppose the problem has the following structure:

$$\phi(x) \text{ is false for some } x \in [L, R] \implies \phi(y) \text{ is false for all } y \geq x$$

Then, we can use binary search to find the *largest* value of x for which $\phi(x)$ is true. The idea is this: for the given interval $[L, R]$, we take $x = \frac{L+R}{2}$ and test whether $\phi(x)$ is true or not. If it is true, we put $L = x$. If not, we put $R = x - 1$. Then we recursively do the procedure on the next interval. Clearly, this takes logarithmic time.

```
/*
    In this pseudocode, mid will hold the final candidate value. The
        boolean variable flag represents whether or not phi(mid) is true
        for the final value of mid or not.
*/
//we are given an interval [L , R] and a predicate phi
lo = L , hi = R;
flag = 0;
while (lo <= hi){
    mid = lo + (hi - lo + 1)/2;
    if (phi(mid)){
        if (lo == mid){
            flag = 1;
            break;
        }
        else
            lo = mid;
    }
    else {
        hi = mid - 1;
    }
```

```
}
if (flag == 1){
    //mid is the highest value in [L , R] for which phi(mid) is true
}
else{
    //there is no value x in [L , R] for which phi(x) is true
}
```

We can obviously apply this technique to the case where we have the following property:

$$\phi(x) \text{ is false for some } x \in [L, R] \implies \phi(y) \text{ is false for all } y \leq x$$

and we can find the *smallest* value of x for which $\phi(x)$ is true. The pseudocode for that case is as follows.

```
/*
    In this pseudocode, mid will hold the final candidate value. The
        boolean variable flag represents whether or not phi(mid) is true
        for the final value of mid or not.
*/
//we are given an interval [L , R] and a predicate phi
lo = L , hi = R;
flag = 0;
while (lo <= hi){
    mid = lo + (hi - lo)/2;
    if (phi(mid)){
        if (lo == mid){
            flag = 1;
            break;
        }
        else
            hi = mid;
    }
    else {
        lo = mid + 1;
    }
}
if (flag == 1){
    //mid is the highest value in [L , R] for which phi(mid) is true
}
else{
    //there is no value x in [L , R] for which phi(x) is true
}
```

5.6. **Binary Search vs Two Pointers.** Sometimes we can apply the two pointer method to various situations where binary search is also applicable, but using two pointers can make the solution linear instead of linearithmic. A good example of this problem is Codeforces 1494C; I have written two solutions to this, one of which uses binary search, while the other uses two pointers. The binary search solution does not work, while two pointers works seemlessly. The solutions can be found in my GitHub repository.

6. Data Structures

6.1. **DSUs.** DSU stands for *Disjoint Set Union*. The idea is to maintain a collection of disjoint sets and have the ability to take the union of two sets efficiently. The most effective way of implementing these is using binary trees, and using two heuristics known as *union by rank* and *path compression*. The code snippet for doing this is given below.

```
//par[k] is the parent of the element k
//root uses path compression
int root(int x){
    par[x] = (x == par[x]) ? x : root(par[x]);
    return par[x];
}
//union uses union by rank
void union(int x , int y){
    a = root(x); b = root(y);
    if(a != b){
        if(rank[a] >= rank[b]){
            par[a] = b; rank[b]++;
        }
        else
            par[b] = a; rank[a]++;
    }
}
```

Finding the root of the element `k` takes time $\alpha(n)$, where α is a very slowly growing function called the *Inverse Ackermann Function* and n is the number of elements.

6.2. **Sparse Tables: Precomputation.** Suppose we have an array `A[1,...,N]`. Then, precomputation is done as follows.

```
//K must satisfy K >= floor(log_2(N))
int sparse_table[N][K]

for (int i = 1; i <= N; i++)
    sparse_table[i][0] = f(array[i]);

for (int j = 1; j <= K; j++)
    for (int i = 1; i + (1 << j) - 1 <= N; i++)
        sparse_table[i][j] = f(sparse_table[i][j-1], sparse_table[i + (1
            << (j - 1))][j - 1]);
```

Here f depends upon the type of query. For taking the minimum, f will be the minimum function for example.

6.3. **Sparse Tables: Range Sum.** For range sum, the function f above will be the sum of the arguments. To get the sum of a range, we do the following.

```
long long sum = 0;
for (int j = K; j >= 0; j--){
    if ((1 << j) <= R - L + 1){
```

```
        sum += sparse_table[L][j];
        L += 1 << j;
    }
}
```

6.4. **Sparse Tables: RMQ.** RMQ is where the true power of sparse tables is apparent. To compute the range minimum in the range $[L, Q]$, let j be the largest index such that $2^j <= R - L + 1$. Then, the minimum in the range $[L, R]$ is simply

```
min(sparse_table[L][j] , sparse_table[R - (1 << j) + 1][j])
```

7. Segment Trees

7.1. **Recursive Segment Trees.** Here is the recursive implementation of segment trees.

```
void build(ll array[] , ll tree[] , ll node , ll start , ll end){
    //leaf node
    if (start == end){
        tree[node] = array[start];
    }
    else{
        ll mid = start + (end - start)/2;
        //recurse on the left child and build the left subtree
        build(array , tree , 2*node , start , mid);
        //recurse on the right child and build the right subtree
        build(array , tree , 2*node + 1 , mid + 1 , end);
        //combine the results
        tree[node] = tree[node * 2] + tree[node * 2 + 1];
    }
}

//to get sum of the range [l , r]
ll sum(ll tree[] , ll node , ll start , ll end , ll l , ll r){
    //if given range is invalid return 0
    if (l > r)
        return 0;
    //if the input range is the range of the node
    if (l == start && r == end)
        return tree[node];
    ll mid = start + (end - start)/2;
    return sum(tree , 2*node , start , mid , l , min(r , mid)) +
        sum(tree , 2*node + 1 , mid + 1 , end , max(l , mid + 1) , r);
}

void update(ll tree[] , ll node , ll start , ll end , ll pos , ll
    new_val){
    //if leaf node, then simply update
    if (start == end){
        tree[node] = new_val;
    }
    else{
```

```
        ll mid = start + (end - start)/2;
        //if position is in the left child, recurse on the left child
        if(pos <= mid)
            update(tree , 2*node , start , mid , pos , new_val);
        //if position is in the right child, recurse on the right child
        else
            update(tree , 2*node + 1 , mid + 1 , end , pos , new_val);
        //update the value of the current node
        tree[node] = tree[2*node] + tree[2*node + 1];
    }
}

//size of the array is N
ll N;
// size of the tree will be atmost 4N
//array indexing starts at 1, and the tree indexing also starts at 1
//tree is tree[] , array is array[]
ll tree[4*N]; ll array[N + 1];

/*
    to build a tree, use the syntax
    build(array,tree,1,left_index,right_index)
    left_index -> leftmost index of array
    right_index -> rightmost index of array
*/

build(array , tree , 1 , 1 , N);

/*
    to update a value, use the syntax
    update(tree,1,left_index,right_index,pos,new_val)
    left_index -> leftmost index of array
    right_index -> rightmost index of array
*/

update(tree , 1 , 1 , N , 10 , 11);

/*
    to find the sum in a range [l , r] do
    sum(tree,1,left_index,right_index,l,r)
    left_index -> leftmost index of array
    right_index -> rightmost index of array
*/
cout << sum(t , 1 , 1 , N , 1 , 10);
```

7.2. **Efficient Segment Trees.** In this section I will be discussing non-recursive implementations of *segment trees* and various other modifications.

Example 7.1 (**Range Sum Queries**). (More information at this link)Suppose we are given an array `A[1,...,n]` and we want to quickly find the sum in the range `A[i,...,j]` efficiently, while also providing a way of updating values in the array (this is an example of a dynamic data structure). First, suppose n is a power

of 2. Then we can make a binary tree out of this array: the leaves will be the array elements; the parent of two adjacent leaves will contain the sum of the two leaves, and this trend will continue upwards. To find the sum in a range, we just need to sum the values in the parents. This is done as in the following code snippet.

```
int array[n]; //this is our array, index starts at 0
int segTree[2n]; //length of segment tree will be 2n
void build(){
    for (int i = 0; i < n; i++){
        segTree[i + n] = arr[i];
    }
    for (int i = n - 1; i > 0; --i){
        segTree[i] = segTree[i << 1] + segTree[i << 1|1]; // left
            shift operator to multiply by 2
    }
}
void modify(int p , int value){ //to modify at position p
    for (segTree[p += n] = value; p > 1; p >>= 1)
        segTree[p>>1] = segTree[p] + segTree[p^1];
}
int query(int n , int l , int r){ //return sum in the range [l , r)
    int res = 0;
    l += n; r += n;
    while (l < r){
        if (l & 1){
            res += segTree[l]; l+= 1; l >>= 1;
        }
        else
            l >>= 1;
        if (r & 1){
            res += segTree[r - 1]; r >>= 1;
        }
        else
            r >>= 1;
    }
    return res;
}
```

The basic idea of the range query is easy; we compute bottom up in the tree.

8. Graph Algorithms

8.1. **BFS.** Here is the implementation.

```
vector<vector<int>> adj; // adjacency list representation
int n; // number of nodes
int s; // source vertex

queue<int> q;
vector<bool> used(n);
vector<int> d(n), p(n);
```

```
    q.push(s);
    used[s] = true;
    p[s] = -1;
    while (!q.empty()) {
        int v = q.front();
        q.pop();
        for (int u : adj[v]) {
            if (!used[u]) {
                used[u] = true;
                q.push(u);
                d[u] = d[v] + 1;
                p[u] = v;
            }
        }
    }
```

8.2. **DFS.** Here is the implementation.

```
vector<vector<int>> adj; // graph represented as an adjacency list
int n; // number of vertices

vector<int> color;

vector<int> time_in, time_out;
int dfs_timer = 0;

void dfs(int v) {
    time_in[v] = dfs_timer++;
    color[v] = 1;
    for (int u : adj[v])
        if (color[u] == 0)
            dfs(u);
    color[v] = 2;
    time_out[v] = dfs_timer++;
}
```

8.3. **Bridges in a Graph.** Fix a root in the graph, and do a DFS from this root. Let $d(u)$ be the discovery time for a vertex u. Compute $\text{low}(v)$ bottom-up in the DFS tree as follows.

$$\text{low}(v) = \min \begin{cases} d(v) & \\ d(p) & , \quad (v,p) \text{ is a back edge} \\ \text{low}(x) & , \quad x \text{ is a child of } v \end{cases}$$

Then, a tree edge (u, v) is a bridge if and only if $\text{low}(v) > d(u)$, and this is easy to see. See problem UVa 796-Critical Links. My solution can be found in the uva796.cpp file.

8.4. **Articulation Points.** We can find articulation points efficiently using the above method. A vertex u is an articulation point if and only if $\text{low}(v) \geq d(u)$ for all children v of u in the DFS tree.

8.5. **Strongly Connected Components.** Here is the algorithm.

(1) Run a DFS on G, recording the finishing times of each vertex.
(2) Compute G^T.
(3) Do a DFS in G^T in the reverse-order of finishing times (highest to lowest). Mark the SCCs.

8.6. **Euler Tour.** The following pseudo-code is a good implementation to find an Euler Tour.

```
stack St;
put start vertex in St;
until St is empty
let V be the value at the top of St;
if degree(V) = 0, then
    add V to the answer;
    remove V from the top of St;
otherwise
    find any edge coming out of V;
    remove it from the graph;
    put the second end of this edge in St;
```

9. Shortest Paths

9.1. **Dijkstra Using set.** The advantage here is that `set` in C++ allows the *delete* operation, while `priority_queue` does not.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    set<pair<int, int>> q;
    q.insert({0, s});
    while (!q.empty()) {
        int v = q.begin()->second;
        q.erase(q.begin());

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
```

```
                q.erase({d[to], to});
                d[to] = d[v] + len;
                p[to] = v;
                q.insert({d[to], to});
            }
        }
    }
}
```

9.2. **Dijkstra using priority queue.** Here, a drawback is that unlike sets, the STL `priority_queue` does *not* support deleting (or updating) elements, but heaps support these operations in theory.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);

    d[s] = 0;
    using pii = pair<int, int>;
    priority_queue<pii, vector<pii>, greater<pii>> q;
    q.push({0, s});
    while (!q.empty()) {
        int v = q.top().second;
        int d_v = q.top().first;
        q.pop();
        if (d_v != d[v])
            continue;

        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
                q.push({d[to], to});
            }
        }
    }
}
```

9.3. **Bellman-Ford Usual.** Here is the usual implementation of Bellman-Ford.

```
struct edge
{
    int a, b, cost;
```

```
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (int i=0; i<n-1; ++i)
        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                d[e[j].b] = min (d[e[j].b], d[e[j].a] + e[j].cost);
    // display d, for example, on the screen
}
```

9.4. **Bellman-Ford Optimised.** There is an optimised version of Bellman-Ford, which is given below. Instead of relaxing edges for $n-1$ rounds, we relax the edges until they can be. In some graphs, this is a great improvement in speed.

```
struct edge
{
    int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve()
{
    vector<int> d (n, INF);
    d[v] = 0;
    for (;;)
    {
        bool any = false;

        for (int j=0; j<m; ++j)
            if (d[e[j].a] < INF)
                if (d[e[j].b] > d[e[j].a] + e[j].cost)
                {
                    d[e[j].b] = d[e[j].a] + e[j].cost;
                    any = true;
                }

        if (!any) break;
    }
    // display d, for example, on the screen
}
```

The above method makes sure that if no relaxation is being done, then just stop the for loop.

9.5. **Negative Cycles using Bellman-Ford.** The criterion for determining whether a negative cycle exists is simple: if there is a negative cycle, then there will be atleast one edge which can be relaxed after Bellman-Ford finishes executing. So, we just need to check whether any edge can be relaxed. If you need more information, check this link: https://cp-algorithms.com/graph/bellman_ford.html.

9.6. **Floyd-Warshall.** In the following implementation, we initialise `d[i][i] = 0` for all i.

```
    for (int k = 1; k <= n; ++k) {
        for (int i = 1; i <= n; ++i) {
            for (int j = 1; j <= n; ++j) {
                if (d[i][k] < INF && d[k][j] < INF)
                    d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

9.7. **Negative Cycles using Floyd-Warshall.** The idea is simple: there is a negative cycle if and only if after Floyd-Warshall, there is some vertex `v` such that `d[v][v] < 0`.

10. MSTs

10.1. **Prim's for dense graphs $O(n^2)$.** In this implementation of Prim, we store, for each *non-selected vertex* (i.e a vertex which hasn't been yet included in the MST), the minimum weight edge to an already selected vertex. After adding an edge, we update this minimum weight edge for each vertex. So, the overall complexity is $O(n^2)$.

```
int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
```

```
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}
```

Example 10.1. This implementation is useful for complete graphs, since we can compute the MST in $O(n^2)$ time and $O(n)$ space.

10.2. **Prim's for sparse graphs $O(m \log n)$.** This is the same implementation as above, the only difference being we use either a `set` or a `priority_queue` to find the minimum weight edge, and to update the minimum weights. In this implementation, the minimum weight can be found in $O(\log n)$ steps, but updating the weights takes $O(n \log n)$ time. The total time taken will be $O(m \log n)$, which is better than the previous implementation for sparse graphs.

```
const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
```

```
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

10.3. **Kruskal Using DSUs.** The algorithm is quite simple to describe.

(1) Sort the edges by weight.
(2) For each edge, check if it forms a cycle if we add it to the current tree. This can be done by checking whether the two ends of the edge belong to the same set, using the `find_root` operation of the DSU.
(3) If the ends does not create a cycle, add it to the tree, and take the union of the two sets the ends belong to.

This clearly takes time $O(m \log n)$.

```
//assuming that the DSU has been implemented as usual
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
```

```
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

sort(edges.begin(), edges.end());

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

11. Tree Algorithms

11.1. **LCA using Euler Tour.** Suppose we have a rooted tree G with N vertices. Then do the following.

(1) Perform a DFS on the tree, and make a list called `euler` which stores the order of the vertices that we visit. Clearly, the size of the list will be $O(N)$.
(2) Make an array `first[1,...,N]` which stores for each vertex i its first occurrence in `euler`. Also, compute the depth `depth[i]` for each node i.

Then, given any two vertices `v1` and `v2`, `LCA(v1,v2)` is the vertex in `euler` having the minimum height in the range between `first[v1]` and `first[v2]`. So, to find the LCA, you can either use a segment tree or a sparse table to solve the RMQ.

11.2. **LCA using Binary Lifting.** The idea here is to use sparse tables. So suppose we are given a tree G with N vertices with root `root`. Let $l = \log_2(N)$. For each `i` between 1 and N and for each `j` between 1 and l, let `up[i][j]` be the 2^jth ancestor of i (if it doesn't exist, we simply let it be `root`). So, `up[i][0]` is the parent node of `i`. Clearly, the array `up` can be easily calculated in $O(N \log N)$ preprocessing. Using DFS, we also compute the discovery and finishing times of each vertex. Then, the LCA can be found as follows (see this link for an explanation).

```
int N, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout; //discovery and finishing times
vector<vector<int>> up;

void dfs(int v, int p)
{
    tin[v] = ++timer;
    up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];

    for (int u : adj[v]) {
```

```
            if (u != p)
                dfs(u, v);
        }

        tout[v] = ++timer;
    }

    bool is_ancestor(int u, int v)
    {
        return tin[u] <= tin[v] && tout[u] >= tout[v];
    }

    int lca(int u, int v)
    {
        if (is_ancestor(u, v))
            return u;
        if (is_ancestor(v, u))
            return v;
        for (int i = l; i >= 0; --i) {
            if (!is_ancestor(up[u][i], v))
            u = up[u][i];
        }
        return up[u][0];
    }

    void preprocess(int root) {
        tin.resize(n);
        tout.resize(n);
        timer = 0;
        l = ceil(log2(n));
        up.assign(n, vector<int>(l + 1));
        dfs(root, root);
    }
```

12. Numerical Methods

12.1. **Ternary Search.** Ternary search is a useful way to find the minimum value of a function that first decreases and then increases, say on an interval $[x_L, x_R]$. The idea is this: divide the range $[x_L, x_R]$ into three equal parts $[x_L, a]$, $[a, b]$ and $[b, x_R]$ by choosing

$$x = \frac{2x_L + x_R}{3} \quad b = \frac{x_L + 2x_R}{3}$$

Then if $f(a) \leq f(b)$, the minimum is in the range $[x_L, b]$; otherwise it is in the range $[a, x_R]$. We can apply this technique to minimise *convex functions*.

13. Number Theoretic Algorithms

13.1. **Sieve of Eratosthenes.** This is a very simple algorithm to find *all* primes upto n. We make an array `sieve[x]` such that for any $1 \leq x \leq n$, `sieve[x] = 1` if and only if x is a prime. Here is the code snippet.

```
for (x = 2; x <= n; x++){
    sieve[x] = 1;
}
for (int x = 2; x <= n; x++){
    if(sieve[x] == 0) continue;
    for (int u = 2*x; u <= n; u += x)
        sieve[u] = 0;
}
```

This algorithm takes time $O(n\text{loglog } n)$

13.2. **Smallest Prime Factor using the Sieve.** Using the Sieve, we can also compute the *smallest prime factor* of all numbers upto a given number n. Using this, we can easily obtain the prime factorisation of *any* number in the range efficiently. The code is as follows.

```
for (x = 2; x <= n; x++){
    sieve[x] = 1;
    smallest_prime_factor[x] = 0;
}
for (int x = 2; x <= n; x++){
    if(sieve[x] == 0) continue;
    for (int u = 2*x; u <= n; u += x){
        sieve[u] = 0;
        if(smallest_prime_factor[u] == 0){
            smallest_prime_factor[u] = x;
        }
    }
}
```

13.3. **Precomputing Factors Using Sieve.** Using the Sieve, we can also precompute factors of all numbers upto a given limit. Consider a positive integer M. To precompute factors of all integers upto M, we can do the following.

```
//let factors be a vector of vectors
for (ll i = 1; i <= M; i++){
    for (ll j = i; j <= M; j += i){
        factors[j].push_back(i);
    }
}
```

This takes time

$$M(1 + 1/2 + 1/3 + ... + 1/M) = O(M \log M)$$

13.4. **GCD.** C++ has an inbuilt function for this, based on the Euclidean Algorithm.

```
int d = __gcd(a , b); //returns the gcd of a , b
```

13.5. **Modular Exponentiation.** Here is the code for modular exponentiation.

```
template <class T> T expo(T base , T exp , T mod){
    T res = 1;
    base = base % mod;
    while (exp > 0){
        if (exp & 1)
            res = (resbase) % mod;
        exp = exp>>1;
        base = (basebase) % mod;
    }
    return res;
}
```

14. Arithmetic Multiplicative Functions

14.1. **Euler's Totient Function from 1 to n.** ϕ is a multiplicative function. For a prime p and $j \geq 1$, we have

$$\phi(p^j) = p^j - p^{j-1}$$

Since ϕ is multiplicative, we can compute $\phi(x)$ for all x upto a number N using the Sieve. This can be done using the following code.

```
void phi_1_to_n(int n) {
    vector<int> phi(n + 1);
    phi[0] = 0;
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
        phi[i] = i;

    for (int i = 2; i <= n; i++) {
        if (phi[i] == i) {
            for (int j = i; j <= n; j += i)
                phi[j] -= phi[j] / i;
        }
    }
}
```

14.2. **Computing Multiplicative Functions Using the Sieve.** Suppose f is a multiplicative function. It follows that f is completely determined by its values on prime powers. Using this idea, we can efficiently compute the values of f for all numbers upto an integer N. The idea is to use the Sieve using smallest prime factors.

```
//we want to compute f[i] for each i upto n
for (x = 2; x <= n; x++){
    sieve[x] = 1;
    smallest_prime_factor[x] = 0;
}
//compute f[0], f[1] separately
```

```
for (int x = 2; x <= n; x++){
    //if x is not a prime
    if(sieve[x] == 0){
        temp = x; k = 1;
        while(temp % smallest_prime_factor[x] == 0){
            temp /= smallest_prime_factor[x];
            k *= smallest_prime_factor[x];
        }
        //x = temp * k and k,temp are coprime
        f[x] = f[temp] * f[k];
    }
    //otherwise x is a prime
    else{
        //compute f for all powers of x
        //temp = x^j
        temp = x; j = 1;
        while (temp <= n){
            //compute f[temp] = f[x^j]

            //increasing the exponent
            temp *= x;
            j++;
        }
        for (int u = 2*x; u <= n; u += x){
            sieve[u] = 0;
            if(smallest_prime_factor[u] == 0){
                smallest_prime_factor[u] = x;
            }
        }
    }
}
```

This will take about $O(n \log n \log \log n)$ time.

14.3. **Dirichlet Convolution.** Suppose f and g are two multiplicative functions. It turns out that the *Dirichlet Convolution* $f * g$ of these functions defined by

$$(f * g)(n) = \sum_{d|n} f(d) g\left(\frac{n}{d}\right)$$

is also a multiplicative function. This is actually pretty useful; in some problems, one has to identify that the function to be computed is actually a Dirichlet convolution of two multiplicative functions. Once we know this, we can use the Sieve to efficiently compute the function. An example of this is the problem on this link: https://www.codechef.com/MAY21C/problems/ISS.

14.4. **Some important multiplicative functions.** Here are some important multiplicative functions that I've encountered.

(1) $\phi(x)$: Euler's Totient Function.
(2) $\tau(x)$: Number of Divisors.
(3) $\sigma_k(x)$: Sum of k^{th} powers of divisors of x.

(4) $g(x) = \sum_{i=1}^{x} \gcd(i, x)$: With a little thought, it can be shown that

$$g(x) = \sum_{d|x} d\phi\left(\frac{x}{d}\right)$$

which is clearly the convolution of the identity function and ϕ, and hence it is multiplicative. This was used in this problem: https://www.codechef.com/MAY21C/problems/ISS.

15. Advanced Graph Algorithms

15.1. **Euler's Formula.** Let G be a planar graph with v, e, k being the number of vertices, edges and the number of connected components of G. Let f be the number of faces that G divides a plane into. Then,

$$v - e + f = 1 + k$$

The problem RCTEXSCC is a great example of this.