Base Template

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;

#define sep() cout<<"-----------------------------------"<<endl;
#define all(x) (x).begin(), (x).end()

int main(){
    ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);

    // mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
    // auto genRand = [&]()->ll{return (ll)rng();};
    // auto randrange = [&](ll l, ll r){return uniform_int_distribution<ll>(l, r)(rng);};

}
```

Handy Info

```
STL -
1) pq/set/mset<T, v<T>, decltype(cmp)> name(cmp)
    - cmp should not capture by ref
    - use [=] or [var..] instead of [&] for capture by value
    - pq.top returns largest while (m)set.begin() returns smallest acc to cmp
2) cmp(element, key) for ub/lb
    - ub(all(v), key, cmp); where element and key need not be of same type
3) Returns (map also behaves very similar to set (likewise, mmap to mset))
    - set.insert returns pair<iter, bool> (iter to inserted/existing el, true if insertion success)
    - mset.insert returns iter
    - erase by key returns #elements erased in (m)set
    - erase by iter returns "iter after the erased one in (m)set"
4) cout << fixed << setprecision(p) << val;
    - W/out fixed, precision is the #{Significant digits}
    - With fixed, precision is #{Digits after decimal point}
5) 31 - clz(x) = floor(log2(x))


Handy Asymptotics / Constants

1) #Primes <= n ~ n / log n
2) Avg #prime factors of n ~ log log n
3) Product of first 10 primes (2..29) ~ 6e9
4) Product of first 15 primes (2..47) ~ 6e17
5) Max #divisors of n:
    - upto 1e9  ~ 1.3e3
    - upto 1e18 ~ 1e5
6) Max distinct prime factors:
    - <= 1e9  → <= 9
    - <= 1e18 → <= 15
7) Max prime factors with multiplicity:
    - <= 1e9  → <= 30
    - <= 1e18 → <= 60
```

```
8) Fibonacci:
    - digits(Fn) ~ log10(Fn) ~ nlogn10(psi) - log10(rt(5))
    - Every 5 steps <==> 1 new digit
    - psi = (1+rt(5))/2 = 1.62
    - F(45) ~ 1e9
    - F(92) ~ 7e18
9) Divisibility tests:
    - 10a+b divides 7 iff 50a+5b divides 7 iff a-2b divides 7 (5 = 10invmod7)
    - 10a+b divides 11 iff 100a+10b divides 11 iff a-b divides 11 (10 = 10invmod11)
    - 10a+b divides 13 iff 40a+4b divides 13 iff a+4b divides 13 (4 = 10invmod7)
    - 10a+b divides 17 iff 120a+12b divides 17 iff a-5b divides 17 (12 = 10invmod17)
    - 10a+b divides 19 iff 20a+2b divides 19 iff a+2b divieds 19 (2 = 10invmod19)
```

Strings

```
vector<int> prefn(const string &s) {
    // pi[i] = len of longest proper prefix of s[0..i] that is also a suffix of s[0..i]
    // kmp
    int n = s.size();
    vector<int> pi(n,0);
    for (int i=1; i<n; i++) {
        int j = pi[i-1];
        while(j>0 && s[i]!=s[j]) {j = pi[j-1];}
        if(s[i]==s[j]) {j++;} // otherwise, j==0 and s[i]!=s[j]
        pi[i] = j;
    }
    return pi;
}

vector<int> zfn(const string &s){
    // z[i] = len of longest prefix of s that is also a prefix of the suffix of s starting at i
    int n = s.size();
    vector<int> z(n,0);
    int l=0, r=0;
    for(int i=1; i<n; i++){
        if(i<r) z[i] = min(r-i, z[i-l]);
        while(i+z[i]<n && s[i+z[i]]==s[z[i]]) z[i]++;
        if(i+z[i]>r){l=i; r=i+z[i];}
    }
    return z;
}

vector<int> sa(const string &s){
    // sa[i] = starting index of the i-th smallest suffix
    // rank[i] = position of suffix starting at i in the sorted order
    int n = s.size();
    vector<int> rank(n), sufarr(n); iota(all(sufarr),0);
    for(int i=0; i<n; i++) {rank[i] = s[i];}
```

```
    for(int k=1; k<n; k<<=1){
        auto cmp = [&](int i, int j)->bool{
            if(rank[i]!=rank[j]) return rank[i]<rank[j];
            return ((i+k<n ? rank[i+k] : -1) < (j+k<n ? rank[j+k] : -1));
        };
        sort(all(sufarr), cmp);
        vector<int> temp(n); temp[sufarr[0]] = 0;
        for(int i=1; i<n; i++){
            temp[sufarr[i]] = temp[sufarr[i-1]] + (cmp(sufarr[i-1],sufarr[i]) ? 1:0);
        }
        rank = temp;
        if(rank[sufarr[n-1]]==n-1) break;
    }
    return sufarr;
}

vector<int> lcp(const string &s, vector<int> &sa) {
    // lcp[i] = length of longest common prefix of sa[i] and sa[i-1], lcp[0]=0
    int n = s.size();
    vector<int> rank(n), lcp(n);
    for(int i=0; i<n; i++) {rank[sa[i]] = i;}

    int k = 0;
    for(int i=0; i<n; i++){
        int r = rank[i];
        if(r==0){lcp[r] = 0;continue;}
        int j = sa[r-1];  // previous suffix in sorted order
        while (i+k<n && j+k<n && s[i+k]==s[j+k]) k++;
        lcp[r] = k;
        if(k>0) {k--;}
    }
    return lcp;
}
```

Fenwick Tree

```
class FenwickTree{ // Binary Indexed Tree for pointUpdates
public:
    vector<int> tree; int sz; int neutral;
    function<int(int, int)> merge;
    function<int(int ,int)> inverseMerge;

    FenwickTree(int _sz, int _neutral, function<int(int, int)> _merge, function<int(int, int)> _inverseMerge, const
vector<int>& arr={}){
        sz = _sz; neutral = _neutral; merge = _merge; inverseMerge = _inverseMerge;
        tree.resize(sz+1, neutral);
        if(!arr.empty()) this->build(arr);
    }

    void build(const vector<int>& arr={}){
        for(int i=0; i<sz; i++){
            update(i, arr[i]);
        }
    }

    void update(int ind, int delta){
        ind++;
        for(; ind<=sz; ind+=(ind&(-ind))){
            tree[ind] = merge(tree[ind], delta);
        }
    }

    int prefixQuery(int ind){
        ind++;
        int result = neutral;
        for(; ind>0; ind&=(ind-1)){
            result = merge(result, tree[ind]);
        }
        return result;
```

```
    }

    int rangeQuery(int l, int r){
        return inverseMerge(prefixQuery(r), prefixQuery(l-1));
    }
};
```

SegTree-1 (point-updates)

```
class SGTree{ // point updates
public:
    int sz, neutral;
    vector<int> seg;
    function<int(int, int)> merge;

    SGTree(int _sz, int _neutral, function<int(int, int)> _merge, const vector<int>& arr={}) {
        sz=_sz; neutral=_neutral; merge=_merge; seg.resize(4*_sz+1);
        if(!arr.empty()) this->build(arr);
    }

    //query, build and point updates
    void build(const vector<int>& arr, int ind=0, int low=0, int high=-1){
        if(high==-1) high=sz-1;
        if(low==high){seg[ind]=arr[low];return;}

        int mid=(low+high)>>1;
        build(arr, 2*ind+1, low, mid); // build left subtree
        build(arr, 2*ind+2, mid+1, high); // build right subtree
        seg[ind]=merge(seg[2*ind+1],seg[2*ind+2]);
    }

    int query(int l, int r, int ind=0, int low=0, int high=-1){
        if(high==-1) high=sz-1;
        if(r<low || high<l) return neutral; // no overlap {l r low high} or {low high l r}
        if(low>=l && high<=r) return seg[ind]; // complete overlap {l low high r}

        int mid=(low+high)>>1; // partial overlap
        int lTree=query(l, r, 2*ind+1, low, mid);
        int rTree=query(l, r, 2*ind+2, mid+1, high);
        return merge(lTree,rTree);
    }
```

```
    void update(int i, int val, int ind=0, int low=0, int high=-1){
        if(high==-1) high=sz-1;
        if(low==high) {seg[ind]=val; return;}

        int mid=(low+high)>>1;
        if(i<=mid){update(i, val, 2*ind+1, low, mid);}
        else{update(i, val, 2*ind+2, mid+1, high);}

        seg[ind]=merge(seg[2*ind+1],seg[2*ind+2]);
    }
};
```

SegTree-2 (Range updates)

```
class SGTree2{ // range updates
public:
    int sz;
    vector<int> seg,lazy;
    SGTree2(int n){sz=n; seg.resize(4*n+1); lazy.resize(4*n+1);}

    //query, build and range updates
    void build(vector<int>& arr, int ind=0, int low=0, int high=-1){
        lazy[ind]=0;
        if(high==-1) high=sz-1;
        if(low==high) {seg[ind]=arr[low]; return;}

        int mid=(low+high)>>1;
        build(arr, 2*ind+1, low, mid); // build left subtree
        build(arr, 2*ind+2, mid+1, high); // build right subtree
        seg[ind]=seg[2*ind+1]+seg[2*ind+2];
    }

    void update(int l, int r, int val, int ind=0, int low=0, int high=-1){
        if(high==-1) high=sz-1;

        if(lazy[ind]!=0){ // do remaining updates if any
            seg[ind]+=(high-low+1)*lazy[ind];
            if(low!=high){lazy[2*ind+1]+=lazy[ind]; lazy[2*ind+2]+=lazy[ind]; } // propagate updates if not a leaf node
            lazy[ind]=0;
        }

        if(r<low || high<l) return; // no overlap {l r low high} or {low high l r}
        if(low>=l && high<=r) { // complete overlap {l low high r}
            seg[ind]+=(high-low+1)*val;
            if(low!=high){lazy[2*ind+1]+=val; lazy[2*ind+2]+=val;}
            return;
        }
```

```
        int mid = (low+high)>>1; // partial overlap
        update(l, r, val, 2*ind+1, low, mid);
        update(l, r, val, 2*ind+2, mid+1, high);
        seg[ind]=seg[2*ind+1]+seg[2*ind+2];
    }

    int query(int l, int r, int ind=0, int low=0, int high=-1){
        if(high==-1) high=sz-1;

        if(lazy[ind]!=0){ // do remaining updates if any
            seg[ind]+=(high-low+1)*lazy[ind];
            if(low!=high){lazy[2*ind+1]+=lazy[ind]; lazy[2*ind+2]+=lazy[ind]; } // propagate updates if not a leaf node
            lazy[ind]=0;
        }

        if(r<low || high<l) return 0; // no overlap {l r low high} or {low high l r}
        if(low>=l && high<=r) return seg[ind]; // complete overlap {l low high r}

        int mid=(low+high)>>1; // partial overlap
        int lTree=query(l, r, 2*ind+1, low, mid);
        int rTree=query(l, r, 2*ind+2, mid+1, high);
        return lTree+rTree;
    }
};
```

Math

```
// implement factorial, modular exp, nCr, lucas optimization with a custom mod m
const int __szf = 1e7;
ll __factorial[__szf];
bool __factorialPreprocessingDone = false;
void __factorialPreprocessing(ll m){
    __factorial[0]=1;
    for(int i=1; i<__szf; i++){
        __factorial[i] = ((i%m)*__factorial[i-1])%m;
    }
}

ll modExp(ll base, ll pow, ll m){
    if(pow==0) return 1LL;
    if(base==1) return 1LL;

    if(pow%2==0){
        ll temp = modExp(base, pow/2, m);
        return (temp*temp)%m;
    }
    else{
        ll temp = modExp(base, (pow-1)/2, m);
        temp = (temp*temp)%m;
        return ((base%m)*temp)%m;
    }
    return -1LL;
}

// verify that m is prime
ll modInv(ll a, ll m){
    return modExp(a, m-2, m); // fermat
}

ll nCr(ll n, ll r, ll m){
```

```
    if(n<r) return 0LL;
    if(r>n-r) r = n-r;
    if(r==0) return 1LL;
    if(r>=__szf) return -1LL;

    if(!__factorialPreprocessingDone){
        // costly
        __factorialPreprocessing(m);
        __factorialPreprocessingDone = true;
    }

    if(n<__szf){
        // ensure m is prime
        ll df = modInv(__factorial[r], m);
        ll df2 = modInv(__factorial[n-r], m);
        return (__factorial[n]*((df*df2)%m))%m;
    }
    else if(r<__szf){
        // costly, do falling factorial
        ll df = modInv(__factorial[r], m);
        ll nf = 1, cnt = 0, ptr = n;
        while(cnt<r){
            nf = (nf*ptr)%m;
            ptr--; cnt++;
        }
        return (nf*df)%m;
    }
    return -1LL;
}

ll lucasOptimization(ll n, ll r, ll m){
    if(n<r) return 0LL;
    if(r>n-r) r=n-r;
    if(r==0) return 1LL;
    ll val = lucasOptimization(n/m, r/m, m)%m;
    return (val*nCr(n%m, r%m, m))%m;
}
```

DSU

```
class dsu { // dsu
    public:
    vector<int> rank, size, parent;

    dsu(int sz){
        rank.resize(sz+1); size.resize(sz+1); parent.resize(sz+1);
        for(int i=0;i<=sz;i++) { parent[i]=i; size[i]=1; rank[i]=0; }
    }
    int find(int node){
        if (node==parent[node]) return node;
        return parent[node] = find(parent[node]);
    }
    void unionByRank(int u, int v){
        // Node with greater rank becomes parent
        int ulp_u = find(u); int ulp_v = find(v); if(ulp_u==ulp_v) return;
        if (rank[ulp_u]==rank[ulp_v]) {parent[ulp_v] = ulp_u; rank[ulp_u]++;}
        else if(rank[ulp_u]<rank[ulp_v]) parent[ulp_u] = ulp_v;
        else parent[ulp_v] = ulp_u; // rank[ulp_u]>rank[ulp_v]
    }
    void unionBySize(int u, int v){
        // Bigger size node becomes parent
        int ulp_u = find(u); int ulp_v = find(v); if(ulp_u==ulp_v) return;
        if (size[ulp_u]<size[ulp_v]) { parent[ulp_u] = ulp_v; size[ulp_v]+=size[ulp_u]; }
        else { parent[ulp_v] = ulp_u; size[ulp_u]+=size[ulp_v]; }
    }
    void unite(int u, int v){
        unionBySize(u,v);
    }
};
```