



International Institute Of Information Technology - Hyderabad

CP Notebook

AAD Project, 2021

Sanyam Shah

2020101012

1 Contest	1	7.7 Series	13	9.21 Catalan Numbers	28
2 Number theory	3	7.8 Probability theory	13	9.22 Derangements	30
2.1 Modular arithmetic	3	7.8.1 Discrete distributions	13	9.23 Strings	30
2.2 Primality	3	7.8.2 Continuous distributions	14	9.23.1 Tries:	31
2.3 Divisibility	3	7.9 Markov chains	14	9.23.2 String Hashing:	31
2.4 Pythagorean Triples	4	8 Miscellaneous	14	9.23.3 Z-algorithm	32
2.5 Primes	4	8.1 RNG, Intervals, Ternary Search	14	9.23.4 Knuth-Morris-Pratt Algorithm	33
2.6 Estimates	4	8.2 Debugging tricks	14	9.24 Geometrical Algorithms:	34
3 Graph	4	8.3 Optimization tricks	15	9.24.1 Circle	34
3.1 Heaps	4	8.3.1 Bit hacks	15	9.24.2 Triangle's Medians	34
3.2 Eulerian Path	4	8.3.2 Pragmas	15		
3.3 Minimum Spanning Trees	4	9 Theory	16		
3.4 Shortest Path	5	9.1 Euclidean Algorithm	16		
3.5 Network flow	5	9.2 Extended Euclidean Algorithm	16		
3.6 Vertex Cover	6	9.3 Modulo Inverse	17		
3.7 DFS algorithms	7	9.3.1 Computing Inverses of first N numbers	17		
4 Combinatorial	8	9.4 Discrete Logarithm	17		
4.1 Permutations	8	9.5 Mod'ed Arithmetic Progression	17		
4.1.1 Factorial	8	9.6 Primitive Root	18		
4.1.2 Derangements	8	9.7 Discrete Root	18		
4.2 Partitions and subsets	8	9.7.1 Euler's Criterion (to check whether square root exists)	18		
4.2.1 Partition function	8	9.7.2 Shanks Tonelli's Algorithm	18		
4.2.2 Lucas' Theorem	8	9.8 Primality Test	19		
4.2.3 Binomials	8	9.8.1 Sieve of Eratosthenes	19		
4.3 General purpose numbers	8	9.8.2 Miller-Rabin Primality Test	19		
4.3.1 Stirling numbers of the second kind	8	9.9 Chinese Remainder Theorem	20		
4.3.2 Catalan numbers	8	9.9.1 Case of Multiple Congruences	20		
5 Strings	8	9.10 Bézout's identity	20		
5.1 String Matching	8	9.11 Phi Function	21		
6 Geometry	8	9.12 Heaps	21		
6.1 Geometric primitives	8	9.13 Eulerian Graph	22		
6.2 Circles	9	9.14 Minimum Spanning Trees	22		
6.3 Polygons	10	9.14.1 Prim's Algorithm	22		
6.4 Misc. Point Set Problems	11	9.14.2 Kruskal's Algorithm	23		
6.5 3D	12	9.15 Shortest Path	23		
7 Mathematics	12	9.15.1 Dijkstra's Algorithm	23		
7.1 Equations	12	9.15.2 Bellman-Ford Algorithm	23		
7.2 Recurrences	13	9.15.3 Floyd-Warshall Algorithm	24		
7.3 Trigonometry	13	9.16 Maximum flow - Ford-Fulkerson and Edmonds-Karp	24		
7.4 Geometry	13	9.16.1 Ford-Fulkerson Algorithm	25		
7.4.1 Triangles	13	9.16.2 Edmonds-Karp Algorithm	25		
7.4.2 Quadrilaterals	13	9.16.3 Dinic's Algorithm	26		
7.4.3 Spherical coordinates	13	9.17 Vertex Cover	26		
7.5 Derivatives/Integrals	13	9.17.1 Vertex Cover for a Tree	27		
7.6 Sums	13	9.17.2 Vertex Cover for a Bipartite Graph	27		
		9.18 Strongly Connected Components	27		
		9.19 2-SAT	27		
		9.20 Diameter of a Tree	28		
				Contest (1)	
				template.cpp	63 lines
				#include <bits/stdc++.h>	
				#include <ext/pb_ds/assoc_container.hpp> // Common file	
				#include <ext/pb_ds/tree_policy.hpp> // Including tree_order_statistics_node_update	
				using namespace std;	
				using namespace __gnu_pbds;	
				typedef tree<int,null_type,less<int>,rb_tree_tag,tree_order_statistics_node_update> ordered_set;	
				typedef long long ll;	
				typedef long double ld;	
				typedef unsigned long long ull;	
				typedef pair<int,int> pi;	
				typedef pair<ll,ll> pl;	
				typedef pair<double,double> pd;	
				typedef vector<ll> vl;	
				typedef vector<int> vi;	
				typedef vector<vector<int>> vvi;	
				typedef vector<vector<ll>> vvl;	
				#define MIN(a, b) ((a) < (b) ? (a) : (b))	
				#define MAX(a, b) ((a) > (b) ? (a) : (b))	
				#define ABS(a) ((a) < 0 ? -(a) : (a))	
				#define ABS(a, b) ((a) > (b) ? (a) - (b) : (b) - (a))	
				#define SWAP(type, a, b) { const type tmp = a; a = b; b = tmp; }	
				#define rep(i,l,r) for(ll i=(l);i<(r);i++)	
				#define per(i,l,r) for(ll i=(l);i>=(r);i--)	
				#define dbg(x) cout<<#x<<" = "<<x<<endl	
				#define mp make_pair	
				#define pb push_back	
				#define ff first	
				#define ss second	
				#define tc ll t; cin>>t; while(t--)	
				#define godspeed ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL)	
				#define all(x) (x).begin(), (x).end()	
				#define sz(x) ((ll)(x).size())	
				#define NO cout << "NO" << "\n"	
				#define YES cout << "YES" << "\n"	
				#define clr(x,y) memset(x, y, sizeof(x))	
				#define setbits(x) __builtin_popcountll(x)	
				#define mod 1000000007	
				const ll inf = 1e9;	
				const ll llinf = 2e18;	
				ll NULL(ll a, ll b) {a = a % mod; b = b % mod; return (((a * b) % mod) + mod) % mod;}	

```
ll POWER(ll a, ll b) {a %= mod; ll res = 1; while (b > 0) {if (
    b & 1) res = MULL(res,a); a = MULL(a,a); b >>= 1;} return
    res;}

void solve(){

    return ;

}

int main()
{
    godspeed;
//    #ifndef ONLINE_JUDGE
//        freopen("input.txt", "r", stdin);
//        freopen("output.txt", "w", stdout);
//    #endif
    tc{
        solve();
    }
    return 0;
}
```

Number theory (2)

2.1 Modular arithmetic

ModArith.h

Description: This snippet consists struct MOD which contains +,-,*,/,inverse,^ operations that can be performed for modular arithmetic. Need to set mod to some number first and then you can use the structure.

"euclid.h"35bfea, 18 lines

```
const ll mod = 17; // change to something else
struct Mod {
    ll x;
    Mod(ll xx) : x(xx) {}
    Mod operator+(Mod b) { return Mod((x + b.x) % mod); }
    Mod operator-(Mod b) { return Mod((x - b.x + mod) % mod); }
    Mod operator*(Mod b) { return Mod((x * b.x) % mod); }
    Mod operator/(Mod b) { return *this * invert(b); }
    Mod invert(Mod a) {
        ll x, y, g = euclid(a.x, mod, x, y);
        assert(g == 1); return Mod((x + mod) % mod);
    }
    Mod operator^(ll e) {
        if (!e) return Mod(1);
        Mod r = *this ^ (e / 2); r = r * r;
        return e&1 ? *this * r : r;
    }
};
```

ModInverse.h

Description: This computes modulo inverses for all natural numbers upto LIM assuming $LIM \leq \text{mod}$ and that mod is a prime.

84f78c, 3 lines

```
const ll mod = 1000000007, LIM = 200000;
ll* inv = new ll[LIM] - 1; inv[1] = 1;
REP(i,2,LIM) inv[i] = mod - (mod / i) * inv[mod % i] % mod;
```

ModLog.h

Description: Returns the smallest $x > 0$ s.t. $a^x = b \pmod m$, or -1 if no such x exists. modLog(a,1,m) can be used to calculate the order of a . In this implementation a and m need not be prime.

Time: $\mathcal{O}(\sqrt{m})$

0ff368, 11 lines

```
ll modLog(ll a, ll b, ll m) {
    ll n = (ll) sqrt(m) + 1, e = 1, f = 1, j = 1;
    unordered_map<ll, ll> A;
    while (j <= n && (e = f = e * a % m) != b % m)
        A[e * b % m] = j++;
    if (e == b % m) return j;
    if (__gcd(m, e) == __gcd(m, b))
        REP(i,2,n+2) if (A.count(e = e * f % m))
            return n * i - A[e];
    return -1;
}
```

ModSum.h

Description: Sums of mod'ed arithmetic progressions.

$\text{modsum}(\text{to}, c, k, m) = \sum_{i=0}^{\text{to}-1} (ki + c) \% m$. divsum is similar but for floored division. sumsq = sum of first n natural numbers.

Time: $\log(m)$, with a large constant.

5c5bc5, 16 lines

```
typedef unsigned long long ull;
ull sumsq(ull to) { return to / 2 * ((to-1) | 1); }
```

```
ull divsum(ull to, ull c, ull k, ull m) {
    ull res = k / m * sumsq(to) + c / m * to;
    k %= m; c %= m;
    if (!k) return res;
    ull to2 = (to * k + c) / m;
    return res + (to - 1) * to2 - divsum(to2, m-1 - c, m, k);
}
```

```
}

ll modsum(ull to, ll c, ll k, ll m) {
    c = ((c % m) + m) % m;
    k = ((k % m) + m) % m;
    return to * c + k * sumsq(to) - m * divsum(to, c, k, m);
}
```

ModMulLL.h

Description: Calculate $a \cdot b \pmod c$ (or $a^b \pmod c$) for $0 \leq a, b \leq c \leq 7.2 \cdot 10^{18}$.

Time: $\mathcal{O}(1)$ for modmul, $\mathcal{O}(\log b)$ for modpow

bbbdb8f, 11 lines

```
typedef unsigned long long ull;
ull modmul(ull a, ull b, ull M) {
    ll ret = a * b - M * ull(1.L / M * a * b);
    return ret + M * (ret < 0) - M * (ret >= (ll)M);
}
ull modpow(ull b, ull e, ull mod) {
    ull ans = 1;
    for (; e; b = modmul(b, b, mod), e /= 2)
        if (e & 1) ans = modmul(ans, b, mod);
    return ans;
}
```

ModSqrt.h

Description: Tonelli-Shanks algorithm for modular square roots. Finds x s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution).

Time: $\mathcal{O}(\log^2 p)$ worst case, $\mathcal{O}(\log p)$ for most p

"ModPow.h"19a793, 24 lines

```
ll sqrt(ll a, ll p) {
    a %= p; if (a < 0) a += p;
    if (a == 0) return 0;
    assert(modpow(a, (p-1)/2, p) == 1); // else no solution
    if (p % 4 == 3) return modpow(a, (p+1)/4, p);
    // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
    ll s = p - 1, n = 2;
    int r = 0, m;
    while (s % 2 == 0)
        ++r, s /= 2;
    while (modpow(n, (p - 1) / 2, p) != p - 1) ++n;
    ll x = modpow(a, (s + 1) / 2, p);
    ll b = modpow(a, s, p), g = modpow(n, s, p);
    for (;;) r = m) {
        ll t = b;
        for (m = 0; m < r && t != 1; ++m)
            t = t * t % p;
        if (m == 0) return x;
        ll gs = modpow(g, 1LL << (r - m - 1), p);
        g = gs * gs % p;
        x = x * gs % p;
        b = b * g % p;
    }
}
```

2.2 Primality

FastEratosthenes.h

Description: Prime sieve for generating all primes smaller than given limit.

8286a0, 21 lines

```
vector<bool> prime;
vector<int> primes;
void sieve()
{
    prime.assign(31665, true);
    prime[0] = prime[1] = false;
    for (int i = 2; i*i < 31665; i++)
    {
        if (prime[i])
        {
```

```
        primes.push_back(i);
        for (int j = i * i; j < 31655; j+=i)
        {
            if (prime[j])
            {
                prime[j] = false;
            }
        }
    }
}
```

MillerRabin.h

Description: Deterministic Miller-Rabin primality test. Guaranteed to work for numbers up to $7 \cdot 10^{18}$; for larger numbers, use Python and extend A randomly.

Time: 7 times the complexity of $a^b \pmod c$.

"ModMulLL.h"60dcd1, 12 lines

```
bool isPrime(ull n) {
    if (n < 2 || n % 6 % 4 != 1) return (n | 1) == 3;
    ull A[] = {2, 325, 9375, 28178, 450775, 9780504, 1795265022},
        s = __builtin_ctzll(n-1), d = n >> s;
    for (ull a : A) { // ^ count trailing zeroes
        ull p = modpow(a%n, d, n), i = s;
        while (p != 1 && p != n - 1 && a % n && i--)
            p = modmul(p, p, n);
        if (p != n-1 && i != s) return 0;
    }
    return 1;
}
```

2.3 Divisibility

euclid.h

Description: Finds two integers x and y , such that $ax + by = \gcd(a, b)$. If you just need gcd, use the built in `__gcd` instead. If a and b are coprime, then x is the inverse of $a \pmod b$.

33ba8f, 5 lines

```
ll euclid(ll a, ll b, ll &x, ll &y) {
    if (!b) return x = 1, y = 0, a;
    ll d = euclid(b, a % b, y, x);
    return y -= a/b * x, d;
}
```

CRT.h

Description: Chinese Remainder Theorem.

$\text{crt}(a, m, b, n)$ computes x such that $x \equiv a \pmod m, x \equiv b \pmod n$. If $|a| < m$ and $|b| < n$, x will obey $0 \leq x < \text{lcm}(m, n)$. Assumes $mn < 2^{62}$.

Time: $\log(n)$

"euclid.h"04d93a, 7 lines

```
ll crt(ll a, ll m, ll b, ll n) {
    if (n > m) swap(a, b), swap(m, n);
    ll x, y, g = euclid(m, n, x, y);
    assert((a - b) % g == 0); // else no solution
    x = (b - a) % n * x % n / g * m + a;
    return x < 0 ? x + m*n/g : x;
}
```

phiFunction.h

Description: Euler's ϕ function is defined as $\phi(n) := \#$ of positive integers $\leq n$ that are coprime with n .

e4742a, 7 lines

```
const int LIM = 5000000;
int phi[LIM];
void calculatePhi() {
    REP(i,0,LIM) phi[i] = i&1 ? i : i/2;
    for (int i = 3; i < LIM; i += 2) if(phi[i] == i)
        for (int j = i; j < LIM; j += i) phi[j] -= phi[j] / i;
}
```

2.4 Pythagorean Triples

The Pythagorean triples are uniquely generated by

$$a = k \cdot (m^2 - n^2), \quad b = k \cdot (2mn), \quad c = k \cdot (m^2 + n^2),$$

with $m > n > 0$, $k > 0$, $m \perp n$, and either m or n even.

2.5 Primes

$p = 962592769$ is such that $2^{21} \mid p - 1$, which may be useful. For hashing use 970592641 (31-bit number), 31443539979727 (45-bit), 3006703054056749 (52-bit). There are 78498 primes less than 1 000 000.

2.6 Estimates

$$\sum_{d \mid n} d = O(n \log \log n).$$

The number of divisors of n is at most around 100 for $n < 5e4$, 500 for $n < 1e7$, 2000 for $n < 1e10$, 200 000 for $n < 1e19$.

Graph (3)

3.1 Heaps

Heap.h
Description: A Heap is a special Tree-based data structure in which the tree is a complete binary tree

```
#define LEFT(i) (2*(i+1)-1)
#define RIGHT(i) (2*(i+1))
#define PARENT(i) (((i)+1)/2-1)
```

```
int *min_heap;
long int *keys;
int *heap_place;
int heap_size=0;
```

```
#define update_place(i) heap_place[min_heap[(i)]]=(i)
```

```
void init_heap(int nelems) {
    int i;
    min_heap=(int*)malloc(sizeof(int)*nelems);
    keys=(long int*)malloc(sizeof(long int)*nelems);
    heap_place=(int*)malloc(sizeof(int)*nelems);
    heap_size=nelems;
    for(i=0;i<nelems;i++) {
        min_heap[i]=i;
        heap_place[i]=i;
        keys[i]=INF;
    }
}

void heap_min_heapify(int i) {
    int smallest;
    int temp;
    int l = LEFT(i);
    int r = RIGHT(i);

    if(l<heap_size && keys[min_heap[l]]<keys[min_heap[i]])
        smallest=l;
    else smallest=i;
    if(r<heap_size &&
        keys[min_heap[r]]<keys[min_heap[smallest]])
        smallest=r;

    if(smallest!=i) {
```

```
        temp=min_heap[i];
        min_heap[i]=min_heap[smallest];
        min_heap[smallest]=temp;
        update_place(smallest);
        update_place(i);
        heap_min_heapify(smallest);
    }
}

int heap_extract_min() {
    int res;
    if (heap_size<1) return -1;
    res=min_heap[0];
    heap_size--;
    min_heap[0]=min_heap[heap_size];
    update_place(0);
    heap_min_heapify(0);
    return res;
}

void heap_decrease_key(int elem, long int key) {
    int temp;
    int i=heap_place[elem];

    keys[min_heap[i]]=key;

    while (i>0 && keys[min_heap[PARENT(i)]] >
        keys[min_heap[i]]) {
        temp=min_heap[i];
        min_heap[i]=min_heap[PARENT(i)];
        min_heap[PARENT(i)]=temp;
        update_place(i);
        update_place(PARENT(i));
        i=PARENT(i);
    }
}
```

3.2 Eulerian Path

EulerianPath.h
Description: Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

```
stack<int> s;
vector<list<int> >adj;

void remove_edge(int u, int v) {
    for (list<int>::iterator it=adj[u].begin();
        it != adj[u].end(); it++) {
        if (*it == v) {
            it = adj[u].erase(it);
            return;
        }
    }
}

int path(int v) {
    int w;
    for (;adj[v].size();v = w) {
        s.push(v);
        list<int>::iterator it = adj[v].begin();
        w = *it;;
        remove_edge(v,w);
        remove_edge(w,v);
        edges--;
```

```
//u - source, v-destiny
int eulerian_path(int u, int v) {
    printf("%d\n", v);
    while (path(u) == u && !s.empty()) {
        printf("-%d", u = s.top());
        s.pop();
    }
    return edges == 0;
}
```

3.3 Minimum Spanning Trees

Prim.h
Description: Prim's algorithm for finding minimum spanning trees in graphs.
Time: $O(V^2)$ with adjacency matrix and $O(E \log V)$ with adjacency list.

```
#define V 5

int minKey(int key[], bool mstSet[])
{
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

void printMST(int parent[], int graph[V][V])
{
    cout<<"Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout<<parent[i]<<" - "<<i<<" \t"<<graph[i][parent[i]]<<
            "\n";
}

void primMST(int graph[V][V])
{
    int parent[V];

    int key[V];

    bool mstSet[V];

    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    key[0] = 0;

    for (int count = 0; count < V - 1; count++)
    {
        int u = minKey(key, mstSet);

        mstSet[u] = true;

        for (int v = 0; v < V; v++)

            if (graph[u][v] && mstSet[v] == false && graph[u][v]
                < key[v])
                parent[v] = u, key[v] = graph[u][v];
    }

    printMST(parent, graph);
}
```

Kruskal.h

Description: Kruskal's algorithm for finding minimum spanning trees in graphs.

Time: $\mathcal{O}(E\log V)$ or $\mathcal{O}(E\log E)$

e7bab6, 80 lines

```
typedef long long ll;

struct Edge
{
    ll s;
    ll e;
    ll w;
};

ll find(vector<ll> &parent, ll x)
{
    if(parent[x]==x)
        return x;
    parent[x] = find(parent, parent[x]);
    return parent[x];
}

void Union(vector<ll> &parent, vector<ll> &raank, ll a, ll b)
{
    if (raank[a] < raank[b])
    {
        parent[a] = b;
        raank[b] += raank[a];
    }
    else
    {
        parent[b] = a;
        raank[a] += raank[b];
    }
    return ;
}

int main()
{
    ll V,E;
    cin >> V >> E;
    vector<Edge> listt;
    vector<ll> parent(V+1);
    vector<ll> raank(V+1);

    vector<Edge> MST(V);

    for (ll i = 0; i < E; i++)
    {
        Edge x;
        cin >> x.s >> x.e >> x.w;
        listt.push_back(x);
    }

    sort(listt.begin(), listt.end(), cmpfunc);

    ll mstsum = 0;

    for (ll i = 0; i < listt.size(); i++)
    {
        raank[i] = 1;
        parent[i] = i;
    }

    ll cnt = 0;

    for (ll i = 0; i < listt.size(); i++)
    {
        Edge curr_edge = listt[i];
```

```
        ll a = find(parent, curr_edge.s);
        ll b = find(parent, curr_edge.e);

        if(a!=b)
        {
            MST[cnt] = curr_edge;
            Union(parent, raank, a,b);
            cnt++;
            mstsum += curr_edge.w;
        }
    }

    cout << mstsum << endl;
    return 0;
}
```

3.4 Shortest Path

Dijkstra.h

Description: Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph. It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph. where, E is the number of edges and V is the number of vertices.

Time: $\mathcal{O}(E\log V)$

66994b, 32 lines

```
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);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        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;
            }
        }
    }
}
```

BellmanFord.h

Description: Calculates shortest paths from s in a graph that might have negative edge weights. Unreachable nodes get $\text{dist} = \text{inf}$; nodes reachable through negative-weight cycles get $\text{dist} = -\text{inf}$.

Time: $\mathcal{O}(VE)$

d9cc58, 34 lines

```
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
```

```
    int dist[V];

    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply return
        }
    }

    printArr(dist, V);

    return;
}
```

FloydWarshall.h

Description: Calculates all-pairs shortest path in a directed graph that might have negative edge weights. Input is an distance matrix m , where $m[i][j] = \text{inf}$ if i and j are not adjacent. As output, $m[i][j]$ is set to the shortest distance between i and j , inf if no path, or $-\text{inf}$ if the path goes through a negative-weight cycle.

Time: $\mathcal{O}(N^3)$

Time: $\mathcal{O}(N^3)$

0ff4bf, 12 lines

```
const ll inf = 1LL << 62;
void floydWarshALL(vector<vector<ll>>& m) {
    int n = SZ(m);
    REP(i,0,n) m[i][i] = min(m[i][i], 0LL);
    REP(k,0,n) REP(i,0,n) REP(j,0,n)
        if (m[i][k] != inf && m[k][j] != inf) {
            auto newDist = max(m[i][k] + m[k][j], -inf);
            m[i][j] = min(m[i][j], newDist);
        }
    REP(k,0,n) if (m[k][k] < 0) REP(i,0,n) REP(j,0,n)
        if (m[i][k] != inf && m[k][j] != inf) m[i][j] = -inf;
}
```

3.5 Network flow

EdmondsKarp.h

Description: Flow algorithm with guaranteed complexity $\mathcal{O}(VE^2)$. To get edge flow values, compare capacities before and after, and take the positive values only.

e311a1, 35 lines

```
template<class T> T edmondsKarp(vector<unordered_map<int, T>>&
    graph, int source, int sink) {
    assert(source != sink);
    T flow = 0;
    VI par(SZ(graph)), q = par;

    for (;;) {
        fill(ALL(par), -1);
        par[source] = 0;
```

```

int ptr = 1;
q[0] = source;

REP(i,0,ptr) {
    int x = q[i];
    for (auto e : graph[x]) {
        if (par[e.first] == -1 && e.second > 0) {
            par[e.first] = x;
            q[ptr++] = e.first;
            if (e.first == sink) goto out;
        }
    }
}
return flow;

out:
T inc = numeric_limits<T>::max();
for (int y = sink; y != source; y = par[y])
    inc = min(inc, graph[par[y]][y]);

flow += inc;
for (int y = sink; y != source; y = par[y]) {
    int p = par[y];
    if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
    graph[y][p] += inc;
}
}
}

```

Dinic.h

Description: Dinic's Algorithm with complexity $O(V^2E)$ b8d428, 81 lines

```

struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)

```

```

                    continue;
                    level[edges[id].u] = level[v] + 1;
                    q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};

```

3.6 Vertex Cover

ApproximateVertexCover.h

Description: Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. This approximate algorithm however finds a vertex cover less than the twice of the minimal vertex cover.

Time: $O(V + E)$ b3a2f3, 32 lines

```

int V;
list<int> *adj;

ApproximateVertexCover()
{
    bool visited[V];
    for (int i=0; i<V; i++)
        visited[i] = false;

    list<int>::iterator i;

    for (int u=0; u<V; u++)
    {
        if (visited[u] == false)

```

```

    {
        for (i= adj[u].begin(); i != adj[u].end(); ++i)
        {
            int v = *i;
            if (visited[v] == false)
            {
                visited[v] = true;
                visited[u] = true;
                break;
            }
        }
    }

    for (int i=0; i<V; i++)
        if (visited[i])
            cout << i << " ";
}

```

VertexCoverTree.h

Description: Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. This algorithm however finds a minimal vertex cover for a tree

Time: $O(V)$ 2f8d36, 78 lines

```

struct node
{
    int data;
    int vc;
    struct node *left, *right;
};

int vCover(struct node *root)
{
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    if (root->vc != 0)
        return root->vc;

    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
    if (root->right)
        size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

    root->vc = min(size_incl, size_excl);

    return root->vc;
}

struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->vc = 0;
    return temp;
}

// For k-ary tree, follow the below code snippet

```

```
void dfs(vector<int> adj[], vector<int> dp[], int src, int par)
{
    for (auto child : adj[src]) {
        if (child != par)
            dfs(adj, dp, child, src);
    }

    for (auto child : adj[src]) {
        if (child != par) {
            // not including source in the vertex cover
            dp[src][0] += dp[child][1];

            // including source in the vertex cover
            dp[src][1] += min(dp[child][1], dp[child][0]);
        }
    }
}

// function to find minimum size of vertex cover
void minSizeVertexCover(vector<int> adj[], int N)
{
    vector<int> dp[N + 1];

    for (int i = 1; i <= N; i++) {
        // 0 denotes not included in vertex cover
        dp[i].push_back(0);

        // 1 denotes included in vertex cover
        dp[i].push_back(1);
    }

    dfs(adj, dp, 1, -1);

    // printing minimum size vertex cover
    cout << min(dp[1][0], dp[1][1]) << endl;
}
```

3.7 DFS algorithms

SCC.h
Description: Finds strongly connected components in a directed graph. If vertices u, v belong to the same component, we can reach u from v and vice versa. This is the implementation of Kosaraju's Algorithm.
Time: $\mathcal{O}(E + V)$

```
vector<vector<int>> adj, adj_rev;
vector<bool> used;
vector<int> order, component;

void dfs1(int v) {
    used[v] = true;

    for (auto u : adj[v])
        if (!used[u])
            dfs1(u);

    order.push_back(v);
}

void dfs2(int v) {
    used[v] = true;
    component.push_back(v);

    for (auto u : adj_rev[v])
        if (!used[u])
            dfs2(u);
}

int main() {
```

```
int n;
// ... read n ...

for (;;) {
    int a, b;
    // ... read next directed edge (a,b) ...
    adj[a].push_back(b);
    adj_rev[b].push_back(a);
}

used.assign(n, false);

for (int i = 0; i < n; i++)
    if (!used[i])
        dfs1(i);

used.assign(n, false);
reverse(order.begin(), order.end());

for (auto v : order)
    if (!used[v]) {
        dfs2(v);

        // ... processing next component ...

        component.clear();
    }
}
```

CondensedGraph.h
Description: Condensed graph of SCC's found in previous code. The condensed Graph is always a DAG (Directed Acyclic Graph)
Time: $\mathcal{O}(E + V)$

```
// continuing from previous code

vector<int> roots(n, 0);
vector<int> root_nodes;
vector<vector<int>> adj_scc(n);

for (auto v : order)
    if (!used[v]) {
        dfs2(v);

        int root = component.front();
        for (auto u : component) roots[u] = root;
        root_nodes.push_back(root);

        component.clear();
    }

for (int v = 0; v < n; v++)
    for (auto u : adj[v]) {
        int root_v = roots[v],
            root_u = roots[u];

        if (root_u != root_v)
            adj_scc[root_v].push_back(root_u);
    }
```

2sat.h
Description: Calculates a valid assignment to boolean variables a, b, c, \dots to a 2-SAT problem, so that an expression of the type $(a \vee b) \wedge (\neg a \vee c) \wedge (d \vee \neg b) \wedge \dots$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions ($\sim x$).

Usage: TwoSat ts(number of boolean variables);
ts.either(0, ~3); // Var 0 is true or var 3 is false
ts.setValue(2); // Var 2 is true
ts.atMostOne({0, ~1, 2}); // ≤ 1 of vars 0, ~1 and 2 are true
ts.solve(); // Returns true iff it is solvable
ts.values[0..N-1] holds the assigned values to the vars
Time: $\mathcal{O}(N + E)$, where N is the number of boolean variables, and E is the number of clauses.

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : g[v]) {
        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : gt[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    order.clear();
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }

    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}
```

DiameterTree.h
Description: Find out the diameter of the tree i.e.the longest path in a tree
Time: $\mathcal{O}(E + V)$

```
// continuing from previous code

void dfsUtil(int node, int count, bool visited[], int& maxCount, list<int>* adj)
{
    visited[node] = true;
    count++;
    for (auto i = adj[node].begin(); i != adj[node].end(); ++i)
        {
```



```
if (!visited[*i]) {
    if (count >= maxCount) {
        maxCount = count;
        x = *i;
    }
    dfsUtil(*i, count, visited, maxCount, adj);
}

}

void dfs(int node, int n, list<int>* adj, int& maxCount)
{
    bool visited[n + 1];
    int count = 0;

    for (int i = 1; i <= n; ++i)
        visited[i] = false;

    dfsUtil(node, count + 1, visited, maxCount, adj);
}

int diameter(list<int>* adj, int n)
{
    int maxCount = INT_MIN;

    /* DFS from a random node and then see
    farthest node X from it*/
    dfs(1, n, adj, maxCount);

    /* DFS from X and check the farthest node
    from it */
    dfs(x, n, adj, maxCount);

    return maxCount;
}
```

Combinatorial (4)

4.1 Permutations

4.1.1 Factorial

<i>n</i>	1	2	3	4	5	6	7	8	9	10
<i>n</i> !	1	2	6	24	120	720	5040	40320	362880	3628800
<i>n</i>	11	12	13	14	15	16	17			
<i>n</i> !	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14			
<i>n</i>	20	25	30	40	50	100	150	171		
<i>n</i> !	2e18	2e25	3e32	8e47	3e64	9e157	6e262	>DBL_MAX		

4.1.2 Derangements

Permutations of a set such that none of the elements appear in their original position.

$$D(n) = (n-1)(D(n-1) + D(n-2)) = nD(n-1) + (-1)^n = \left\lfloor \frac{n!}{e} \right\rfloor$$

4.2 Partitions and subsets

4.2.1 Partition function

Number of ways of writing n as a sum of positive integers, disregarding the order of the summands.

$$p(0) = 1, \quad p(n) = \sum_{k \in \mathbb{Z} \setminus \{0\}} (-1)^{k+1} p(n - k(3k-1)/2)$$

multinomial StringHash Zfunc KMP Point

$$p(n) \sim 0.145/n \cdot \exp(2.56\sqrt{n})$$

n	0	1	2	3	4	5	6	7	8	9	20	50	100
$p(n)$	1	1	2	3	5	7	11	15	22	30	627	$\sim 2e5$	$\sim 2e8$

4.2.2 Lucas’ Theorem

Used to calculate $\binom{n}{r} \% p$. Let n, m be non-negative integers and p a prime. Write $n = n_k p^k + \dots + n_1 p + n_0$ and $m = m_k p^k + \dots + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^k \binom{n_i}{m_i} \pmod{p}$.

4.2.3 Binomials

multinomial.h

```
Description: Computes  $\binom{k_1 + \dots + k_n}{k_1, k_2, \dots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \dots k_n!}$ . 037a49, 6 lines

ll multinomial(VI& v) {
    ll c = 1, m = v.empty() ? 1 : v[0];
    REP(i, 1, SZ(v)) REP(j, 0, v[i])
        c = c * ++m / (j+1);
    return c;
}
```

4.3 General purpose numbers

4.3.1 Stirling numbers of the second kind

Partitions of n distinct elements into exactly k non-empty subsets. $S(n, k) = S(n-1, k-1) + kS(n-1, k)$
 $S(n, 1) = S(n, n) = 1$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

4.3.2 Catalan numbers

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \binom{2n}{n} - \binom{2n}{n+1} = \frac{(2n)!}{(n+1)n!}$$

$$C_0 = 1, \quad C_{n+1} = \frac{2(2n+1)}{n+2} C_n, \quad C_{n+1} = \sum C_i C_{n-i}$$

$$C_n = 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, \dots$$

Strings (5)

5.1 String Matching

StringHash.h

Description: The code in this snippet will use $p = 31$ and $m = 10^9 + 9$ to calculate the hash of a string.

```
c2d743, 11 lines

long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
}
```

```
return hash_value;
}
```

Zfunc.h

Description: $z[x]$ computes the length of the longest common prefix of $s[i:]$ and s , except $z[0] = 0$. (abacaba -> 0010301)

```
Time:  $\mathcal{O}(n)$  85d656, 13 lines

vector<int> z_function(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1;
    }
    return z;
}
```

KMP.h

Description: $pi[x]$ computes the length of the longest prefix of s that ends at x , other than $s[0..x]$ itself (abacaba -> 0010123). Can be used to find all occurrences of a string.

```
Time:  $\mathcal{O}(n)$  f6d312, 13 lines

vector<int> prefix_function(string s) {
    int n = (int) s.length();
    vector<int> pi(n);
    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++;
        pi[i] = j;
    }
    return pi;
}
```

Geometry (6)

6.1 Geometric primitives

Point.h

Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

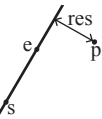
```
47ec0a, 28 lines

template <class T> int sgn(T x) { return (x > 0) - (x < 0); }
template<class T>
struct Point {
    typedef Point P;
    T x, y;
    explicit Point(T x=0, T y=0) : x(x), y(y) {}
    bool operator<(P p) const { return tie(x,y) < tie(p.x,p.y); }
    bool operator==(P p) const { return tie(x,y)==tie(p.x,p.y); }
    P operator+(P p) const { return P(x+p.x, y+p.y); }
    P operator-(P p) const { return P(x-p.x, y-p.y); }
    P operator*(T d) const { return P(x*d, y*d); }
    P operator/(T d) const { return P(x/d, y/d); }
    T dot(P p) const { return x*p.x + y*p.y; }
    T cross(P p) const { return x*p.y - y*p.x; }
    T cross(P a, P b) const { return (a-*this).cross(b-*this); }
    T dist2() const { return x*x + y*y; }
    double dist() const { return sqrt((double)dist2()); }
    // angle to x-axis in interval [-pi, pi]
}
```

```
double angle() const { return atan2(y, x); }
P unit() const { return *this/dist(); } // makes dist()=1
P perp() const { return P(-y, x); } // rotates +90 degrees
P normal() const { return perp().unit(); }
// returns point rotated 'a' radians ccw around the origin
P rotate(double a) const {
    return P(x*cos(a)-y*sin(a),x*sin(a)+y*cos(a)); }
friend ostream& operator<<(ostream& os, P p) {
    return os << "(" << p.x << ", " << p.y << ")"; }
};
```

lineDistance.h

Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long. Using Point3D will always give a non-negative distance. For Point3D, call .dist on the result of the cross product.



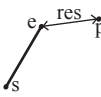
```
template<class P>
double lineDist(const P& a, const P& b, const P& p) {
    return (double)(b-a).cross(p-a)/(b-a).dist();
}
```

f6bf6b, 4 lines

SegmentDistance.h

Description:
Returns the shortest distance between point p and the line segment from point s to e.

Usage: Point<double> a, b(2,2), p(1,1);
bool onSegment = segDist(a,b,p) < 1e-10;



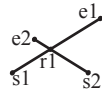
```
typedef Point<double> P;
double segDist(P& s, P& e, P& p) {
    if (s==e) return (p-s).dist();
    auto d = (e-s).dist2(), t = min(d,max(.0,(p-s).dot(e-s)));
    return ((p-s)*d-(e-s)*t).dist()/d;
}
```

5c88f4, 6 lines

SegmentIntersection.h

Description:
If a unique intersection point between the line segments going from s1 to e1 and from s2 to e2 exists then it is returned. If no intersection point exists an empty vector is returned. If infinitely many exist a vector with 2 elements is returned, containing the endpoints of the common line segment. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or long long.

Usage: vector<P> inter = segInter(s1,e1,s2,e2);
if (SZ(inter)==1)
cout << "segments intersect at " << inter[0] << endl;

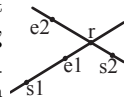


```
"Point.h", "OnSegment.h"
template<class P> vector<P> segInter(P a, P b, P c, P d) {
    auto oa = c.cross(d, a), ob = c.cross(d, b),
          oc = a.cross(b, c), od = a.cross(b, d);
    // Checks if intersection is single non-endpoint point.
    if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
        return {(a * ob - b * oa) / (ob - oa)};
    set<P> s;
    if (onSegment(c, d, a)) s.insert(a);
    if (onSegment(c, d, b)) s.insert(b);
    if (onSegment(a, b, c)) s.insert(c);
    if (onSegment(a, b, d)) s.insert(d);
    return {ALL(s)};
```

36c2d7, 13 lines

```
}

lineIntersection.h
Description:
If a unique intersection point of the lines going through s1,e1 and s2,e2 exists {1, point} is returned. If no intersection point exists {0, (0,0)} is returned and if infinitely many exists {-1, (0,0)} is returned. The wrong position will be returned if P is Point<ll> and the intersection point does not have integer coordinates. Products of three coordinates are used in intermediate steps so watch out for overflow if using int or ll.
Usage: auto res = lineInter(s1,e1,s2,e2);
if (res.first == 1)
cout << "intersection point at " << res.second << endl;
```



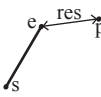
```
"Point.h"
template<class P>
pair<int, P> lineInter(P s1, P e1, P s2, P e2) {
    auto d = (e1 - s1).cross(e2 - s2);
    if (d == 0) // if parallel
        return {-(s1.cross(e1, s2) == 0), P(0, 0)};
    auto p = s2.cross(e1, e2), q = s2.cross(e2, s1);
    return {1, (s1 * p + e1 * q) / d};
}
```

a01f81, 8 lines

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 ⇔ left/on line/right. If the optional argument eps is given 0 is returned if p is within distance eps from the line. P is supposed to be Point<T> where T is e.g. double or long long. It uses products in intermediate steps so watch out for overflow if using int or long long.

Usage: bool left = sideOf(p1,p2,q)==1;



```
"Point.h"
template<class P>
int sideOf(P s, P e, P p) { return sgn(s.cross(e, p)); }
```

3af81c, 9 lines

```
template<class P>
int sideOf(const P& s, const P& e, const P& p, double eps) {
    auto a = (e-s).cross(p-s);
    double l = (e-s).dist()*eps;
    return (a > l) - (a < -l);
}
```

OnSegment.h

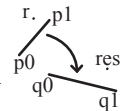
Description: Returns true iff p lies on the line segment from s to e. Use (segDist(s,e,p)<=epsilon) instead when using Point<double>.

```
"Point.h"
template<class P> bool onSegment(P s, P e, P p) {
    return p.cross(s, e) == 0 && (s - p).dot(e - p) <= 0;
}
```

c597e8, 3 lines

linearTransformation.h

Description:
Apply the linear transformation (translation, rotation and scaling) which takes line p0-p1 to line q0-q1 to point r.



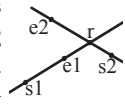
```
"Point.h"
typedef Point<double> P;
P linearTransformation(const P& p0, const P& p1,
    const P& q0, const P& q1, const P& r) {
    P dp = p1-p0, dq = q1-q0, num(dp.cross(dq), dp.dot(dq));
    return q0 + P((r-p0).cross(num), (r-p0).dot(num))/dp.dist2();
}
```

03a306, 6 lines

Angle.h

Description: A class for ordering angles (as represented by int points and a number of rotations around the origin). Useful for rotational sweeping. Sometimes also represents points or vectors.

Usage: vector<Angle> v = {w[0], w[0].t360() ...}; // sorted
int j = 0; REP(i,0,n) { while (v[j] < v[i].t180()) ++j; }
// sweeps j such that (j-i) represents the number of positively oriented triangles with vertices at 0 and i



```
struct Angle {
    int x, y;
    int t;
    Angle(int x, int y, int t=0) : x(x), y(y), t(t) {}
    Angle operator-(Angle b) const { return {x-b.x, y-b.y, t}; }
    int half() const {
        assert(x || y);
        return y < 0 || (y == 0 && x < 0);
    }
    Angle t90() const { return {-y, x, t + (half() && x >= 0)}; }
    Angle t180() const { return {-x, -y, t + half()}; }
    Angle t360() const { return {x, y, t + 1}; }
};
bool operator<(Angle a, Angle b) {
    // add a.dist2() and b.dist2() to also compare distances
    return make_tuple(a.t, a.half(), a.y * (ll)b.x) <
           make_tuple(b.t, b.half(), a.x * (ll)b.y);
}

// Given two points, this calculates the smallest angle between
// them, i.e., the angle that covers the defined line segment.
pair<Angle, Angle> segmentAngles(Angle a, Angle b) {
    if (b < a) swap(a, b);
    return (b < a.t180() ?
        make_pair(a, b) : make_pair(b, a.t360()));
}
Angle operator+(Angle a, Angle b) { // point a + vector b
    Angle r(a.x + b.x, a.y + b.y, a.t);
    if (a.t180() < r) r.t--;
    return r.t180() < a ? r.t360() : r;
}
Angle angleDiff(Angle a, Angle b) { // angle b - angle a
    int tu = b.t - a.t; a.t = b.t;
    return {a.x*b.x + a.y*b.y, a.x*b.y - a.y*b.x, tu - (b < a)};
}
```

0f0602, 35 lines

6.2 Circles

CircleIntersection.h

Description: Computes the pair of points at which two circles intersect. Returns false in case of no intersection.

```
"Point.h"
typedef Point<double> P;
bool circleInter(P a,P b,double r1,double r2,pair<P, P>* out) {
    if (a == b) { assert(r1 != r2); return false; }
    P vec = b - a;
    double d2 = vec.dist2(), sum = r1+r2, dif = r1-r2,
           p = (d2 + r1*r1 - r2*r2)/(d2*2), h2 = r1*r1 - p*p*d2;
    if (sum*sum < d2 || dif*dif > d2) return false;
    P mid = a + vec*p, per = vec.perp() * sqrt(fmax(0, h2) / d2);
    *out = {mid + per, mid - per};
    return true;
}
```

84d6d3, 11 lines

CircleTangents.h

Description: Finds the external tangents of two circles, or internal if r2 is negated. Can return 0, 1, or 2 tangents – 0 if one circle contains the other (or overlaps it, in the internal case, or if the circles are the same); 1 if the circles are tangent to each other (in which case .first = .second and the tangent line is perpendicular to the line between the centers). .first and .second give the tangency points at circle 1 and 2 respectively. To find the tangents of a circle with a point set r2 to 0.

```
"Point.h"
template<class P>
```

b0153d, 13 lines

```
vector<pair<P, P>> tangents(P c1, double r1, P c2, double r2) {
    P d = c2 - c1;
    double dr = r1 - r2, d2 = d.dist2(), h2 = d2 - dr * dr;
    if (d2 == 0 || h2 < 0) return {};
    vector<pair<P, P>> out;
    for (double sign : {-1, 1}) {
        P v = (d * dr + d.perp() * sqrt(h2) * sign) / d2;
        out.push_back({c1 + v * r1, c2 + v * r2});
    }
    if (h2 == 0) out.pop_back();
    return out;
}
```

CircleLine.h

Description: Finds the intersection between a circle and a line. Returns a vector of either 0, 1, or 2 intersection points. P is intended to be Point<double>.

"Point.h"e0cfba, 9 lines

```
template<class P>
vector<P> circleLine(P c, double r, P a, P b) {
    P ab = b - a, p = a + ab * (c-a).dot(ab) / ab.dist2();
    double s = a.cross(b, c), h2 = r*r - s*s / ab.dist2();
    if (h2 < 0) return {};
    if (h2 == 0) return {p};
    P h = ab.unit() * sqrt(h2);
    return {p - h, p + h};
}
```

CirclePolygonIntersection.h

Description: Returns the area of the intersection of a circle with a ccw polygon.

Time: $\mathcal{O}(n)$

"../content/geometry/Point.h"f5c096, 19 lines

```
typedef Point<double> P;
#define arg(p, q) atan2(p.cross(q), p.dot(q))
double circlePoly(P c, double r, vector<P> ps) {
    auto tri = [&](P p, P q) {
        auto r2 = r * r / 2;
        P d = q - p;
        auto a = d.dot(p)/d.dist2(), b = (p.dist2()-r*r)/d.dist2();
        auto det = a * a - b;
        if (det <= 0) return arg(p, q) * r2;
        auto s = max(0., -a-sqrt(det)), t = min(1., -a+sqrt(det));
        if (t < 0 || 1 <= s) return arg(p, q) * r2;
        P u = p + d * s, v = p + d * t;
        return arg(p,u) * r2 + u.cross(v)/2 + arg(v,q) * r2;
    };
    auto sum = 0.0;
    REP(i,0,SZ(ps))
        sum += tri(ps[i] - c, ps[(i + 1) % SZ(ps)] - c);
    return sum;
}
```

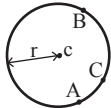
circumcircle.h

Description:

The circumcircle of a triangle is the circle intersecting all three vertices. ccRadius returns the radius of the circle going through points A, B and C and ccCenter returns the center of the same circle.

"Point.h"1caa3a, 9 lines

```
typedef Point<double> P;
double ccRadius(const P& A, const P& B, const P& C) {
    return (B-A).dist()*(C-B).dist()*(A-C).dist()/
        abs((B-A).cross(C-A))/2;
}
P ccCenter(const P& A, const P& B, const P& C) {
    P b = C-A, c = B-A;
```



```
return A + (b*c.dist2()-c*b.dist2()).perp()/b.cross(c)/2;
}
```

MinimumEnclosingCircle.h

Description: Computes the minimum circle that encloses a set of points.

Time: expected $\mathcal{O}(n)$

"circumcircle.h"69dd52, 17 lines

```
pair<P, double> mec(vector<P> ps) {
    shuffle(ALL(ps), mt19937(time(0)));
    P o = ps[0];
    double r = 0, EPS = 1 + 1e-8;
    REP(i,0,SZ(ps)) if ((o - ps[i]).dist() > r * EPS) {
        o = ps[i], r = 0;
        REP(j,0,i) if ((o - ps[j]).dist() > r * EPS) {
            o = (ps[i] + ps[j]) / 2;
            r = (o - ps[i]).dist();
            REP(k,0,j) if ((o - ps[k]).dist() > r * EPS) {
                o = ccCenter(ps[i], ps[j], ps[k]);
                r = (o - ps[i]).dist();
            }
        }
    }
    return {o, r};
}
```

6.3 Polygons

InsidePolygon.h

Description: Returns true if p lies within the polygon. If strict is true, it returns false for points on the boundary. The algorithm uses products in intermediate steps so watch out for overflow.

Usage: vector<P> v = {P{4,4}, P{1,2}, P{2,1}};

bool in = inPolygon(v, P{3, 3}, false);

Time: $\mathcal{O}(n)$

"Point.h", "OnSegment.h", "SegmentDistance.h"2261c4, 11 lines

```
template<class P>
bool inPolygon(vector<P> &p, P a, bool strict = true) {
    int cnt = 0, n = SZ(p);
    REP(i,0,n) {
        P q = p[(i + 1) % n];
        if (onSegment(p[i], q, a)) return !strict;
        //or: if (segDist(p[i], q, a) <= eps) return !strict;
        cnt ^= ((a.y<p[i].y) - (a.y<q.y)) * a.cross(p[i], q) > 0;
    }
    return cnt;
}
```

PolygonArea.h

Description: Returns twice the signed area of a polygon. Clockwise enumeration gives negative area. Watch out for overflow if using int as T!

"Point.h"e287fe, 6 lines

```
template<class T>
T polygonArea2(vector<Point<T>>& v) {
    T a = v.back().cross(v[0]);
    REP(i,0,SZ(v)-1) a += v[i].cross(v[i+1]);
    return a;
}
```

PolygonCenter.h

Description: Returns the center of mass for a polygon.

Time: $\mathcal{O}(n)$

"Point.h"7d84e0, 9 lines

```
typedef Point<double> P;
P polygonCenter(const vector<P>& v) {
    P res(0, 0); double A = 0;
    for (int i = 0, j = SZ(v) - 1; i < SZ(v); j = i++) {
        res = res + (v[i] + v[j]) * v[j].cross(v[i]);
```

```
        A += v[j].cross(v[i]);
    }
    return res / A / 3;
}
```

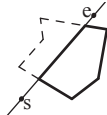
PolygonCut.h

Description:

Returns a vector with the vertices of a polygon with everything to the left of the line going from s to e cut away.

Usage: vector<P> p = ...;

p = polygonCut(p, P(0,0), P(1,0));



"Point.h", "LineIntersection.h"f50354, 18 lines

```
typedef Point<double> P;
vector<P> polygonCut(const vector<P>& poly, P s, P e) {
    if (SZ(poly) <= 2) return {};
    vector<P> res;
    REP(i,0,SZ(poly)) {
        P cur = poly[i], prev = i ? poly[i-1] : poly.back();
        if (zero(s.cross(e, cur))) {
            res.push_back(cur);
            continue;
        }
        bool side = s.cross(e, cur) < 0;
        if (side != (s.cross(e, prev) < 0))
            res.push_back(lineInter(s, e, cur, prev).second);
        if (side)
            res.push_back(cur);
    }
    return res;
}
```

ConvexHull.h

Description:

Returns a vector of the points of the convex hull in counter-clockwise order. Points on the edge of the hull between two other points are not considered part of the hull.

Time: $\mathcal{O}(n \log n)$



"Point.h"c5c490, 13 lines

```
typedef Point<ll> P;
vector<P> convexHull(vector<P> pts) {
    if (SZ(pts) <= 1) return pts;
    sort(ALL(pts));
    vector<P> h(SZ(pts)+1);
    int s = 0, t = 0;
    for (int it = 2; it--; s = --t, reverse(ALL(pts)))
        for (P p : pts) {
            while (t >= s + 2 && h[t-2].cross(h[t-1], p) <= 0) t--;
            h[t++] = p;
        }
    return {h.begin(), h.begin() + t - (t == 2 && h[0] == h[1])};
}
```

HullDiameter.h

Description: Returns the two points with max distance on a convex hull (ccw, no duplicate/collinear points).

"Point.h"261063, 12 lines

```
typedef Point<ll> P;
array<P, 2> hullDiameter(vector<P> S) {
    int n = SZ(S), j = n < 2 ? 0 : 1;
    pair<ll, array<P, 2>> res({0, {S[0], S[0]}});
    REP(i,0,j)
        for (; j = (j + 1) % n) {
            res = max(res, {(S[i] - S[j]).dist2(), {S[i], S[j]}});
            if ((S[(j + 1) % n] - S[j]).cross(S[i + 1] - S[i]) >= 0)
                break;
        }
    return res.second;
}
```

```
}

PointInsideHull.h
Description: Determine whether a point t lies inside a convex hull (CCW
order, with no collinear points). Returns true if point lies within the hull. If
strict is true, points on the boundary aren't included.
Time:  $\mathcal{O}(\log N)$ 
"Point.h", "sideOf.h", "OnSegment.h"
efb6da, 14 lines
```

```
typedef Point<ll> P;

bool inHull(const vector<P>& l, P p, bool strict = true) {
    int a = 1, b = SZ(l) - 1, r = !strict;
    if (SZ(l) < 3) return r && onSegment(l[0], l.back(), p);
    if (sideOf(l[0], l[a], l[b]) > 0) swap(a, b);
    if (sideOf(l[0], l[a], p) >= r || sideOf(l[0], l[b], p)<= -r)
        return false;
    while (abs(a - b) > 1) {
        int c = (a + b) / 2;
        (sideOf(l[0], l[c], p) > 0 ? b : a) = c;
    }
    return sgn(l[a].cross(l[b], p)) < r;
}
```

```
LineHullIntersection.h
Description: Line-convex polygon intersection. The polygon must be ccw
and have no collinear points. lineHull(line, poly) returns a pair describing
the intersection of a line with the polygon:  $\bullet(-1, -1)$  if no collision,  $\bullet(i, -1)$ 
if touching the corner  $i$ ,  $\bullet(i, i)$  if along side  $(i, i+1)$ ,  $\bullet(i, j)$  if crossing sides
 $(i, i+1)$  and  $(j, j+1)$ . In the last case, if a corner  $i$  is crossed, this is treated
as happening on side  $(i, i+1)$ . The points are returned in the same order as
the line hits the polygon. extrVertex returns the point of a hull with the
max projection onto a line.
Time:  $\mathcal{O}(\log n)$ 
"Point.h"
331463, 39 lines
```

```
#define cmp(i,j) sgn(dir.perp().cross(poly[(i)%n]-poly[(j)%n]))
#define extr(i) cmp(i + 1, i) >= 0 && cmp(i, i - 1 + n) < 0
template <class P> int extrVertex(vector<P>& poly, P dir) {
    int n = SZ(poly), lo = 0, hi = n;
    if (extr(0)) return 0;
    while (lo + 1 < hi) {
        int m = (lo + hi) / 2;
        if (extr(m)) return m;
        int ls = cmp(lo + 1, lo), ms = cmp(m + 1, m);
        (ls < ms || (ls == ms && ls == cmp(lo, m)) ? hi : lo) = m;
    }
    return lo;
}

#define cmpL(i) sgn(a.cross(poly[i], b))
template <class P>
array<int, 2> lineHull(P a, P b, vector<P>& poly) {
    int endA = extrVertex(poly, (a - b).perp());
    int endB = extrVertex(poly, (b - a).perp());
    if (cmpL(endA) < 0 || cmpL(endB) > 0)
        return {-1, -1};
    array<int, 2> res;
    REP(i,0,2) {
        int lo = endB, hi = endA, n = SZ(poly);
        while ((lo + 1) % n != hi) {
            int m = ((lo + hi + (lo < hi ? 0 : n)) / 2) % n;
            (cmpL(m) == cmpL(endB) ? lo : hi) = m;
        }
        res[i] = (lo + !cmpL(hi)) % n;
        swap(endA, endB);
    }
    if (res[0] == res[1]) return {res[0], -1};
    if (!cmpL(res[0]) && !cmpL(res[1]))
        switch ((res[0] - res[1] + SZ(poly) + 1) % SZ(poly)) {
```

```
        case 0: return {res[0], res[0]};
        case 2: return {res[1], res[1]};
    }
    return res;
}
```

6.4 Misc. Point Set Problems

```
ClosestPair.h
Description: Finds the closest pair of points.
Time:  $\mathcal{O}(n \log n)$ 
"Point.h"
ac393c, 17 lines

typedef Point<ll> P;
pair<P, P> closest(vector<P> v) {
    assert(SZ(v) > 1);
    set<P> S;
    sort(ALL(v), [](P a, P b) { return a.y < b.y; });
    pair<ll, pair<P, P>> ret{LLONG_MAX, {P(), P()}};
    int j = 0;
    for (P p : v) {
        P d{1 + (ll)sqrt(ret.first), 0};
        while (v[j].y <= p.y - d.x) S.erase(v[j++]);
        auto lo = S.lower_bound(p - d), hi = S.upper_bound(p + d);
        for (; lo != hi; ++lo)
            ret = min(ret, {( *lo - p).dist2(), { *lo, p } });
        S.insert(p);
    }
    return ret.second;
}
```

```
kdTree.h
Description: KD-tree (2d, can be extended to 3d)
"Point.h"
269f22, 63 lines

typedef long long T;
typedef Point<T> P;
const T INF = numeric_limits<T>::max();
```

```
bool on_x(const P& a, const P& b) { return a.x < b.x; }
bool on_y(const P& a, const P& b) { return a.y < b.y; }

struct Node {
    P pt; // if this is a leaf, the single point in it
    T x0 = INF, x1 = -INF, y0 = INF, y1 = -INF; // bounds
    Node *first = 0, *second = 0;

    T distance(const P& p) { // min squared distance to a point
        T x = (p.x < x0 ? x0 : p.x > x1 ? x1 : p.x);
        T y = (p.y < y0 ? y0 : p.y > y1 ? y1 : p.y);
        return (P(x,y) - p).dist2();
    }

    Node(vector<P>&& vp) : pt(vp[0]) {
        for (P p : vp) {
            x0 = min(x0, p.x); x1 = max(x1, p.x);
            y0 = min(y0, p.y); y1 = max(y1, p.y);
        }
        if (vp.size() > 1) {
            // split on x if width >= height (not ideal...)
            sort(ALL(vp), x1 - x0 >= y1 - y0 ? on_x : on_y);
            // divide by taking half the array for each child (not
            // best performance with many duplicates in the middle)
            int half = SZ(vp)/2;
            first = new Node({vp.begin(), vp.begin() + half});
            second = new Node({vp.begin() + half, vp.end()});
        }
    }
};

struct KDTree {
```

```
Node* root;
KDTree(const vector<P>& vp) : root(new Node({ALL(vp)})) {}

pair<T, P> search(Node *node, const P& p) {
    if (!node->first) {
        // uncomment if we should not find the point itself:
        // if (p == node->pt) return {INF, P()};
        return make_pair((p - node->pt).dist2(), node->pt);
    }

    Node *f = node->first, *s = node->second;
    T bfirst = f->distance(p), bsec = s->distance(p);
    if (bfirst > bsec) swap(bsec, bfirst), swap(f, s);

    // search closest side first, other side if needed
    auto best = search(f, p);
    if (bsec < best.first)
        best = min(best, search(s, p));
    return best;
}

// find nearest point to a point, and its squared distance
// (requires an arbitrary operator< for Point)
pair<T, P> nearest(const P& p) {
    return search(root, p);
}
};
```

```
FastDelaunay.h
Description: Fast Delaunay triangulation. Each circumcircle contains none
of the input points. There must be no duplicate points. If all points are on a
line, no triangles will be returned. Should work for doubles as well, though
there may be precision issues in 'circ'. Returns triangles in order {t[0][0],
t[0][1], t[0][2], t[1][0], ...}, all counter-clockwise.
Time:  $\mathcal{O}(n \log n)$ 
"Point.h"
aaca8f, 88 lines

typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128 t ll; // (can be ll if coords are < 2e4)
P arb(LLONG_MAX,LLONG_MAX); // not equal to any other point
```

```
struct Quad {
    Q rot, o; P p = arb; bool mark;
    P& F() { return r()->p; }
    Q& r() { return rot->rot; }
    Q prev() { return rot->o->rot; }
    Q next() { return r()->prev(); }
} *H;

bool circ(P p, P a, P b, P c) { // is p in the circumcircle?
    ll p2 = p.dist2(), A = a.dist2()-p2,
        B = b.dist2()-p2, C = c.dist2()-p2;
    return p.cross(a,b)*C + p.cross(b,c)*A + p.cross(c,a)*B > 0;
}

Q makeEdge(P orig, P dest) {
    Q r = H ? H : new Quad{new Quad{new Quad{new Quad{0}}}};
    H = r->o; r->r()->r() = r;
    REP(i,0,4) r = r->rot, r->p = arb, r->o = i & 1 ? r : r->r();
    r->p = orig; r->F() = dest;
    return r;
}

void splice(Q a, Q b) {
    swap(a->o->rot->o, b->o->rot->o); swap(a->o, b->o);
}

Q connect(Q a, Q b) {
    Q q = makeEdge(a->F(), b->p);
    splice(q, a->next());
    splice(q->r(), b);
    return q;
}
```

```
}

pair<Q,Q> rec(const vector<P>& s) {
    if (SZ(s) <= 3) {
        Q a = makeEdge(s[0], s[1]), b = makeEdge(s[1], s.back());
        if (SZ(s) == 2) return { a, a->r() };
        splice(a->r(), b);
        auto side = s[0].cross(s[1], s[2]);
        Q c = side ? connect(b, a) : 0;
        return {side < 0 ? c->r() : a, side < 0 ? c : b->r() };
    }

#define H(e) e->F(), e->p
#define valid(e) (e->F().cross(H(base)) > 0)
    Q A, B, ra, rb;
    int half = SZ(s) / 2;
    tie(ra, A) = rec({ALL(s) - half});
    tie(B, rb) = rec({SZ(s) - half + ALL(s)});
    while ((B->p.cross(H(A)) < 0 && (A = A->next())) ||
        (A->p.cross(H(B)) > 0 && (B = B->r()->o)));
    Q base = connect(B->r(), A);
    if (A->p == ra->p) ra = base->r();
    if (B->p == rb->p) rb = base;

#define DEL(e, init, dir) Q e = init->dir; if (valid(e)) \
    while (circ(e->dir->F(), H(base), e->F())) { \
        Q t = e->dir; \
        splice(e, e->prev()); \
        splice(e->r(), e->r()->prev()); \
        e->o = H; H = e; e = t; \
    }
    for (;;) {
        DEL(LC, base->r(), o); DEL(RC, base, prev());
        if (!valid(LC) && !valid(RC)) break;
        if (!valid(LC) || (valid(RC) && circ(H(RC), H(LC))))
            base = connect(RC, base->r());
        else
            base = connect(base->r(), LC->r());
    }
    return { ra, rb };
}

vector<P> triangulate(vector<P> pts) {
    sort(ALL(pts)); assert(unique(ALL(pts)) == pts.end());
    if (SZ(pts) < 2) return {};
    Q e = rec(pts).first;
    vector<Q> q = {e};
    int qi = 0;
    while (e->o->F().cross(e->F(), e->p) < 0) e = e->o;
#define ADD { Q c = e; do { c->mark = 1; pts.push_back(c->p); \
    q.push_back(c->r()); c = c->next(); } while (c != e); }
    ADD; pts.clear();
    while (qi < SZ(q)) if (!(e = q[qi++])->mark) ADD;
    return pts;
}
```

6.5 3D

PolyhedronVolume.h

Description: Magic formula for the volume of a polyhedron. Faces should point outwards.

```
template<class V, class L>
double signedPolyVolume(const V& p, const L& trilst) {
    double v = 0;
    for (auto i : trilst) v += p[i.a].cross(p[i.b]).dot(p[i.c]);
    return v / 6;
}
```

3058c3, 6 lines

PolyhedronVolume Point3D 3dHull sphericalDistance

Point3D.h

Description: Class to handle points in 3D space. T can be e.g. double or long long.

```
template<class T> struct Point3D {
    typedef Point3D P;
    typedef const P& R;
    T x, y, z;
    explicit Point3D(T x=0, T y=0, T z=0) : x(x), y(y), z(z) {}
    bool operator<(R p) const {
        return tie(x, y, z) < tie(p.x, p.y, p.z); }
    bool operator==(R p) const {
        return tie(x, y, z) == tie(p.x, p.y, p.z); }
    P operator+(R p) const { return P(x+p.x, y+p.y, z+p.z); }
    P operator-(R p) const { return P(x-p.x, y-p.y, z-p.z); }
    P operator*(T d) const { return P(x*d, y*d, z*d); }
    P operator/(T d) const { return P(x/d, y/d, z/d); }
    T dot(R p) const { return x*p.x + y*p.y + z*p.z; }
    P cross(R p) const {
        return P(y*p.z - z*p.y, z*p.x - x*p.z, x*p.y - y*p.x);
    }
    T dist2() const { return x*x + y*y + z*z; }
    double dist() const { return sqrt((double)dist2()); }
    //Azimuthal angle (longitude) to x-axis in interval [-pi, pi]
    double phi() const { return atan2(y, x); }
    //Zenith angle (latitude) to the z-axis in interval [0, pi]
    double theta() const { return atan2(sqrt(x*x+y*y),z); }
    P unit() const { return *this/(T)dist(); } //makes dist()==1
    //returns unit vector normal to *this and p
    P normal(P p) const { return cross(p).unit(); }
    //returns point rotated 'angle' radians ccw around axis
    P rotate(double angle, P axis) const {
        double s = sin(angle), c = cos(angle); P u = axis.unit();
        return u*dot(u)*(1-c) + (*this)*c - cross(u)*s;
    }
};
```

3dHull.h

Description: Computes all faces of the 3-dimension hull of a point set. *No four points must be coplanar*, or else random results will be returned. All faces will point outwards.

Time: $\mathcal{O}(n^2)$

"Point3D.h" 0754b0, 49 lines

```
typedef Point3D<double> P3;

struct PR {
    void ins(int x) { (a == -1 ? a : b) = x; }
    void rem(int x) { (a == x ? a : b) = -1; }
    int cnt() { return (a != -1) + (b != -1); }
    int a, b;
};

struct F { P3 q; int a, b, c; };

vector<F> hull3d(const vector<P3>& A) {
    assert(SZ(A) >= 4);
    vector<vector<PR>> E(SZ(A), vector<PR>(SZ(A), {-1, -1}));
#define E(x,y) E[f.x][f.y]
    vector<F> FS;
    auto mf = [&](int i, int j, int k, int l) {
        P3 q = (A[j] - A[i]).cross((A[k] - A[i]));
        if (q.dot(A[l]) > q.dot(A[i]))
            q = q * -1;
        F f{q, i, j, k};
        E(a,b).ins(k); E(a,c).ins(j); E(b,c).ins(i);
        FS.push_back(f);
    };
    REP(i,0,4) REP(j,i+1,4) REP(k,j+1,4)
        mf(i, j, k, 6 - i - j - k);
};
```

```
REP(i,4,SZ(A)) {
    REP(j,0,SZ(FS)) {
        F f = FS[j];
        if(f.q.dot(A[i]) > f.q.dot(A[f.a])) {
            E(a,b).rem(f.c);
            E(a,c).rem(f.b);
            E(b,c).rem(f.a);
            swap(FS[j--], FS.back());
            FS.pop_back();
        }
        int nw = SZ(FS);
        REP(j,0,nw) {
            F f = FS[j];
#define C(a, b, c) if (E(a,b).cnt() != 2) mf(f.a, f.b, i, f.c);
            C(a, b, c); C(a, c, b); C(b, c, a);
        }
        for (F& it : FS) if ((A[it.b] - A[it.a]).cross(
            A[it.c] - A[it.a]).dot(it.q) <= 0) swap(it.c, it.b);
        return FS;
    }
};
```

sphericalDistance.h

Description: Returns the shortest distance on the sphere with radius radius between the points with azimuthal angles (longitude) f1 (ϕ_1) and f2 (ϕ_2) from x axis and zenith angles (latitude) t1 (θ_1) and t2 (θ_2) from z axis (0 = north pole). All angles measured in radians. The algorithm starts by converting the spherical coordinates to cartesian coordinates so if that is what you have you can use only the two last rows. dx*radius is then the difference between the two points in the x direction and d*radius is the total distance between the points.

611f07, 8 lines

```
double sphericalDistance(double f1, double t1,
    double f2, double t2, double radius) {
    double dx = sin(t2)*cos(f2) - sin(t1)*cos(f1);
    double dy = sin(t2)*sin(f2) - sin(t1)*sin(f1);
    double dz = cos(t2) - cos(t1);
    double d = sqrt(dx*dx + dy*dy + dz*dz);
    return radius*2*asin(d/2);
}
```

Mathematics (7)

7.1 Equations

$$ax^2 + bx + c = 0 \Rightarrow x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The extremum is given by $x = -b/2a$.

$$\begin{aligned} ax + by &= e & x &= \frac{ed - bf}{ad - bc} \\ cx + dy &= f & y &= \frac{af - ec}{ad - bc} \end{aligned} \Rightarrow$$

In general, given an equation $Ax = b$, the solution to a variable x_i is given by

$$x_i = \frac{\det A'_i}{\det A}$$

where A'_i is A with the i 'th column replaced by b .

7.2 Recurrences

If $a_n = c_1a_{n-1} + \cdots + c_ka_{n-k}$, and r_1, \dots, r_k are distinct roots of $x^k + c_1x^{k-1} + \cdots + c_k$, there are d_1, \dots, d_k s.t.

$$a_n = d_1r_1^n + \cdots + d_kr_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1n + d_2)r^n$.

7.3 Trigonometry

$$\begin{aligned}\sin(v+w) &= \sin v \cos w + \cos v \sin w \\ \cos(v+w) &= \cos v \cos w - \sin v \sin w\end{aligned}$$

$$\begin{aligned}\tan(v+w) &= \frac{\tan v + \tan w}{1 - \tan v \tan w} \\ \sin v + \sin w &= 2 \sin \frac{v+w}{2} \cos \frac{v-w}{2} \\ \cos v + \cos w &= 2 \cos \frac{v+w}{2} \cos \frac{v-w}{2}\end{aligned}$$

$$(V+W)\tan(v-w)/2 = (V-W)\tan(v+w)/2$$

where V, W are lengths of sides opposite angles v, w .

$$\begin{aligned}a \cos x + b \sin x &= r \cos(x - \phi) \\ a \sin x + b \cos x &= r \sin(x + \phi)\end{aligned}$$

where $r = \sqrt{a^2 + b^2}, \phi = \text{atan2}(b, a)$.

7.4 Geometry

7.4.1 Triangles

Side lengths: a, b, c

Semiperimeter: $p = \frac{a+b+c}{2}$

Area: $A = \sqrt{p(p-a)(p-b)(p-c)}$

Circumradius: $R = \frac{abc}{4A}$

Inradius: $r = \frac{A}{p}$

Length of median (divides triangle into two equal-area triangles):

$$m_a = \frac{1}{2}\sqrt{2b^2 + 2c^2 - a^2}$$

Length of bisector (divides angles in two):

$$s_a = \sqrt{bc \left[1 - \left(\frac{a}{b+c} \right)^2 \right]}$$

Law of sines: $\frac{\sin \alpha}{a} = \frac{\sin \beta}{b} = \frac{\sin \gamma}{c} = \frac{1}{2R}$

Law of cosines: $a^2 = b^2 + c^2 - 2bc \cos \alpha$

Law of tangents: $\frac{a+b}{a-b} = \frac{\tan \frac{\alpha+\beta}{2}}{\tan \frac{\alpha-\beta}{2}}$

7.4.2 Quadrilaterals

With side lengths a, b, c, d , diagonals e, f , diagonals angle θ , area A and magic flux $F = b^2 + d^2 - a^2 - c^2$, then $4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2f^2 - F^2}$

For cyclic quadrilaterals the sum of opposite angles is 180° , $ef = ac + bd$, and $A = \sqrt{(p-a)(p-b)(p-c)(p-d)}$.

7.4.3 Spherical coordinates

$$\begin{aligned}x &= r \sin \theta \cos \phi & r &= \sqrt{x^2 + y^2 + z^2} \\ y &= r \sin \theta \sin \phi & \theta &= \text{acos}(z/\sqrt{x^2 + y^2 + z^2}) \\ z &= r \cos \theta & \phi &= \text{atan2}(y, x)\end{aligned}$$

7.5 Derivatives/Integrals

$$\begin{aligned}\frac{d}{dx} \arcsin x &= \frac{1}{\sqrt{1-x^2}} & \frac{d}{dx} \arccos x &= -\frac{1}{\sqrt{1-x^2}} \\ \frac{d}{dx} \tan x &= 1 + \tan^2 x & \frac{d}{dx} \arctan x &= \frac{1}{1+x^2} \\ \int \tan ax &= -\frac{\ln |\cos ax|}{a} & \int x \sin ax &= \frac{\sin ax - ax \cos ax}{a^2} \\ \int e^{-x^2} &= \frac{\sqrt{\pi}}{2} \text{erf}(x) & \int x e^{ax} dx &= \frac{e^{ax}}{a^2} (ax - 1)\end{aligned}$$

Integration by parts:

$$\int_a^b f(x)g(x)dx = [F(x)g(x)]_a^b - \int_a^b F(x)g'(x)dx$$

7.6 Sums

$$c^a + c^{a+1} + \cdots + c^b = \frac{c^{b+1} - c^a}{c - 1}, c \neq 1$$

$$\begin{aligned}1 + 2 + 3 + \cdots + n &= \frac{n(n+1)}{2} \\ 1^2 + 2^2 + 3^2 + \cdots + n^2 &= \frac{n(2n+1)(n+1)}{6} \\ 1^3 + 2^3 + 3^3 + \cdots + n^3 &= \frac{n^2(n+1)^2}{4} \\ 1^4 + 2^4 + 3^4 + \cdots + n^4 &= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}\end{aligned}$$

7.7 Series

$$\begin{aligned}e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots, (-\infty < x < \infty) \\ \ln(1+x) &= x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots, (-1 < x \leq 1)\end{aligned}$$

$$\sqrt{1+x} = 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{2x^3}{32} - \frac{5x^4}{128} + \cdots, (-1 \leq x \leq 1)$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, (-\infty < x < \infty)$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, (-\infty < x < \infty)$$

7.8 Probability theory

Let X be a discrete random variable with probability $p_X(x)$ of assuming the value x . It will then have an expected value (mean) $\mu = \mathbb{E}(X) = \sum_x x p_X(x)$ and variance $\sigma^2 = V(X) = \mathbb{E}(X^2) - (\mathbb{E}(X))^2 = \sum_x (x - \mathbb{E}(X))^2 p_X(x)$ where σ is the standard deviation. If X is instead continuous it will have a probability density function $f_X(x)$ and the sums above will instead be integrals with $p_X(x)$ replaced by $f_X(x)$.

Expectation is linear:

$$\mathbb{E}(aX + bY) = a\mathbb{E}(X) + b\mathbb{E}(Y)$$

For independent X and Y ,

$$V(aX + bY) = a^2V(X) + b^2V(Y).$$

7.8.1 Discrete distributions

Binomial distribution

The number of successes in n independent yes/no experiments, each which yields success with probability p is $\text{Bin}(n, p)$, $n = 1, 2, \dots$, $0 \leq p \leq 1$.

$$p(k) = \binom{n}{k} p^k (1-p)^{n-k}$$

$$\mu = np, \sigma^2 = np(1-p)$$

$\text{Bin}(n, p)$ is approximately $\text{Po}(np)$ for small p .

First success distribution

The number of trials needed to get the first success in independent yes/no experiments, each wich yields success with probability p is $\text{Fs}(p)$, $0 \leq p \leq 1$.

$$p(k) = p(1-p)^{k-1}, k = 1, 2, \dots$$

$$\mu = \frac{1}{p}, \sigma^2 = \frac{1-p}{p^2}$$

Poisson distribution

The number of events occurring in a fixed period of time t if these events occur with a known average rate κ and independently of the time since the last event is $\text{Po}(\lambda)$, $\lambda = t\kappa$.

$$p(k) = e^{-\lambda} \frac{\lambda^k}{k!}, k = 0, 1, 2, \dots$$

$$\mu = \lambda, \sigma^2 = \lambda$$

7.8.2 Continuous distributions

Uniform distribution

If the probability density function is constant between a and b and 0 elsewhere it is $U(a, b)$, $a < b$.

$$f(x) = \begin{cases} \frac{1}{b-a} & a < x < b \\ 0 & \text{otherwise} \end{cases}$$

$$\mu = \frac{a+b}{2}, \sigma^2 = \frac{(b-a)^2}{12}$$

Exponential distribution

The time between events in a Poisson process is $\text{Exp}(\lambda)$, $\lambda > 0$.

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$
$$\mu = \frac{1}{\lambda}, \sigma^2 = \frac{1}{\lambda^2}$$

Normal distribution

Most real random values with mean μ and variance σ^2 are well described by $\mathcal{N}(\mu, \sigma^2)$, $\sigma > 0$.

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

If $X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2)$ and $X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2)$ then

$$aX_1 + bX_2 + c \sim \mathcal{N}(\mu_1 + \mu_2 + c, a^2\sigma_1^2 + b^2\sigma_2^2)$$

7.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let X_1, X_2, \dots be a sequence of random variables generated by the Markov process. Then there is a transition matrix $\mathbf{P} = (p_{ij})$, with $p_{ij} = \Pr(X_n = i | X_{n-1} = j)$, and $\mathbf{p}^{(n)} = \mathbf{P}^n \mathbf{p}^{(0)}$ is the probability distribution for X_n (i.e., $p_i^{(n)} = \Pr(X_n = i)$), where $\mathbf{p}^{(0)}$ is the initial distribution.

π is a stationary distribution if $\pi = \pi \mathbf{P}$. If the Markov chain is *irreducible* (it is possible to get to any state from any state), then $\pi_i = \frac{1}{\mathbb{E}(T_i)}$ where $\mathbb{E}(T_i)$ is the expected time between two visits in state i . π_j / π_i is the expected number of visits in state j between two visits in state i .

For a connected, undirected and non-bipartite graph, where the transition probability is uniform among all neighbors, π_i is proportional to node i 's degree.

A Markov chain is *ergodic* if the asymptotic distribution is independent of the initial distribution. A finite Markov chain is ergodic iff it is irreducible and *aperiodic* (i.e., the gcd of cycle lengths is 1). $\lim_{k \rightarrow \infty} \mathbf{P}^k = \mathbf{1}\pi$.

A Markov chain is an A-chain if the states can be partitioned into two sets \mathbf{A} and \mathbf{G} , such that all states in \mathbf{A} are absorbing ($p_{ii} = 1$), and all states in \mathbf{G} leads to an absorbing state in \mathbf{A} . The probability for absorption in state $i \in \mathbf{A}$, when the initial state is j , is $a_{ij} = p_{ij} + \sum_{k \in \mathbf{G}} a_{ik} p_{kj}$. The expected time until absorption, when the initial state is i , is $t_i = 1 + \sum_{k \in \mathbf{G}} p_{ki} t_k$.

Miscellaneous (8)

8.1 RNG, Intervals, Ternary Search

RNGs.h

6 lines

```
SEED = chrono::steady_clock::now().time_since_epoch().count();
// or use 'high_resolution_clock'
random_device rd; auto SEED = rd();
mt19937 rng(SEED);
uniform_int_distribution<> dis(MIN, MAX); // usage: dis(rng)
// others: uniform_real_distribution,
```

IntervalContainer.h

Description: Add and remove intervals from a set of disjoint intervals. Will merge the added interval with any overlapping intervals in the set when adding. Intervals are [inclusive, exclusive).

Time: $\mathcal{O}(\log N)$

564cdd, 23 lines

```
set<PII>::iterator addInterval(set<PII>& is, int L, int R) {
    if (L == R) return is.end();
    auto it = is.lower_bound({L, R}), before = it;
    while (it != is.end() && it->first <= R) {
        R = max(R, it->second);
        before = it = is.erase(it);
    }
    if (it != is.begin() && (--it)->second >= L) {
        L = min(L, it->first);
        R = max(R, it->second);
        is.erase(it);
    }
    return is.insert(before, {L,R});
}
```

```
void removeInterval(set<PII>& is, int L, int R) {
    if (L == R) return;
    auto it = addInterval(is, L, R);
    auto r2 = it->second;
    if (it->first == L) is.erase(it);
    else (int&)it->second = L;
    if (R != r2) is.emplace(R, r2);
}
```

IntervalCover.h

Description: Compute indices of smallest set of intervals covering another interval. Intervals should be [inclusive, exclusive). To support [inclusive, inclusive], change (A) to add || R.empty(). Returns empty set on failure (or if G is empty).

Time: $\mathcal{O}(N \log N)$

354a6a, 19 lines

```
template<class T>
VI cover(pair<T, T> G, vector<pair<T, T>> I) {
    VI S(SZ(I)), R;
    iota(ALL(S), 0);
    sort(ALL(S), [&](int a, int b) { return I[a] < I[b]; });
    T cur = G.first;
    int at = 0;
    while (cur < G.second) { // (A)
```

```
pair<T, int> mx = make_pair(cur, -1);
while (at < SZ(I) && I[S[at]].first <= cur) {
    mx = max(mx, make_pair(I[S[at]].second, S[at]));
    at++;
}
if (mx.second == -1) return {};
cur = mx.first;
R.push_back(mx.second);
}
return R;
}
```

ConstantIntervals.h

Description: Split a monotone function on [from, to) into a minimal set of half-open intervals on which it has the same value. Runs a callback g for each such interval.

Usage: constantIntervals(0, SZ(v), [&](int x){return v[x];}, [&](int lo, int hi, T val){...});

Time: $\mathcal{O}(k \log \frac{n}{k})$

753a4c, 19 lines

```
template<class F, class G, class T>
void rec(int from, int to, F& f, G& g, int& i, T& p, T q) {
    if (p == q) return;
    if (from == to) {
        g(i, to, p);
        i = to; p = q;
    } else {
        int mid = (from + to) >> 1;
        rec(from, mid, f, g, i, p, f(mid));
        rec(mid+1, to, f, g, i, p, q);
    }
}

template<class F, class G>
void constantIntervals(int from, int to, F f, G g) {
    if (to <= from) return;
    int i = from; auto p = f(i), q = f(to-1);
    rec(from, to-1, f, g, i, p, q);
    g(i, to, q);
}
```

TernarySearch.h

Description: Find the smallest i in [a,b] that maximizes $f(i)$, assuming that $f(a) < \dots < f(i) \geq \dots \geq f(b)$. To reverse which of the sides allows non-strict inequalities, change the < marked with (A) to <=, and reverse the loop at (B). To minimize f, change it to >, also at (B).

Usage: int ind = ternSearch(0,n-1,[&](int i){return a[i];});

Time: $\mathcal{O}(\log(b-a))$

a9cf52, 11 lines

```
template<class F>
int ternSearch(int a, int b, F f) {
    assert(a <= b);
    while (b - a >= 5) {
        int mid = (a + b) / 2;
        if (f(mid) < f(mid+1)) a = mid; // (A)
        else b = mid+1;
    }
    REP(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
    return a;
}
```

8.2 Debugging tricks

- signal(SIGSEGV, [](int) { _Exit(0); }); converts segfaults into Wrong Answers. Similarly one can catch SIGABRT (assertion failures) and SIGFPE (zero divisions). _GLIBCXX_DEBUG failures generate SIGABRT (or SIGSEGV on gcc 5.4.0 apparently).

- feenableexcept(29); kills the program on NaNs (1), 0-divs (4), infinities (8) and denormals (16).

trace.h10 lines

```
#define trace(...) { __f(#_VA_ARGS_, __VA_ARGS_); }
template<typename Arg> void __f(const char* name, Arg&& arg) {
    cerr << name << " = " << arg << endl;
}
template <typename Arg1, typename... Args>
void __f(const char* names, Arg1&& arg1, Args&&... args) {
    const char* comma = strchr(names + 1, ',');
    cerr.write(names, comma - names) << " = " << arg1<<" | ";
    __f(comma+1, args...);
}
```

8.3 Optimization tricks

`__builtin_ia32_ldmxcsr(40896);` disables denormals (which make floats 20x slower near their minimum value).

8.3.1 Bit hacks

- `x & -x` is the least bit in `x`.
- `for (int x = m; x;) { --x &= m; ... }` loops over all subset masks of `m` (except `m` itself).
- `c = x&-x, r = x+c; (((r^x) >> 2)/c) | r` is the next number after `x` with the same number of bits set.
- `REP(b,0,K) REP(i,0,(1 << K)) if (i & 1 << b) D[i] += D[i^(1 << b)];` computes all sums of subsets.

8.3.2 Pragmas

- `#pragma GCC optimize ("0fast")` will make GCC auto-vectorize loops and optimizes floating points better.
- `#pragma GCC target ("avx2")` can double performance of vectorized code, but causes crashes on old machines.
- `#pragma GCC optimize ("trapv")` kills the program on integer overflows (but is really slow).
- `#pragma GCC optimize("unroll-loops")`
- `target("sse,sse2,sse3,ssse3,sse4,popcnt,abm,mmx,avx")`

FastMod.h

Description: Compute $a\%b$ about 5 times faster than usual, where b is constant but not known at compile time. Returns a value congruent to $a \pmod b$ in the range $[0, 2b)$.

751a02, 8 lines

```
typedef unsigned long long ull;
struct FastMod {
    ull b, m;
    FastMod(ull b) : b(b), m(-1ULL / b) {}
    ull reduce(ull a) { // a % b + (0 or b)
        return a - (ull)((__uint128_t(m) * a) >> 64) * b;
    }
};
```

FastInput.h7b3c70, 17 lines

Description: Read an integer from stdin. Usage requires your program to pipe in input from file.

Usage: `./a.out < input.txt`

Time: About 5x as fast as `cin/scanf`.

```
inline char gc() { // like getchar()
    static char buf[1 << 16];
    static size_t bc, be;
    if (bc >= be) {
        buf[0] = 0, bc = 0;
        be = fread(buf, 1, sizeof(buf), stdin);
    }
    return buf[bc++]; // returns 0 on EOF
}

int readInt() {
    int a, c;
    while ((a = gc()) < 40);
    if (a == '-') return -readInt();
    while ((c = gc()) >= 48) a = a * 10 + c - 480;
    return a - 48;
}
```


Theory (9)

9.1 Euclidean Algorithm

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & \text{otherwise} \end{cases}$$

For the proof of correctness, we need to show that $\gcd(a, b) = \gcd(b, a \bmod b)$ for all $a \geq 0, b > 0$. We will show that the value on the left side of the equation divides the value on the right side and vice versa and hence proving Euclid's ALgorithm. Let $d = \gcd(a, b)$. Then by definition $d \mid a$ and $d \mid b$.

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

From this it follows that $d \mid (a \bmod b)$, $d \mid b$ and $d \mid a \bmod b \Rightarrow d \mid \gcd(b, a \bmod b)$. Thus we have shown that the left side of the original equation divides the right. The second half of the proof is similar.

Time Complexity: $O(\log \min(a, b))$ LCM can be calculated using the Euclidean algorithm with the same time complexity using the formula:

$$\text{lcm}(a, b) = \frac{a \cdot b}{\gcd(a, b)}$$

9.2 Extended Euclidean Algorithm

The extended version of Euclid finds a way to represent GCD in terms of a and b , i.e. coefficients x and y for which:

$$a \cdot x + b \cdot y = \gcd(a, b)$$

Let GCD of a and b be g .

From the previous algorithm, we can see that the algorithm ends with $b = 0$ and $a = g$. For these parameters we can easily find coefficients, namely $g \cdot 1 + 0 \cdot 0 = g$.

Starting from these coefficients $(x, y) = (1, 0)$, we can go backwards up the recursive calls. All we need to do is to figure out how the coefficients x and y change during the transition from $(a, b) \rightarrow (b, a \bmod b)$.

Let us assume we found the coefficients (x_1, y_1) for $(b, a \bmod b)$:

$$b \cdot x_1 + (a \bmod b) \cdot y_1 = g$$

and we want to find the pair (x, y) for (a, b) :

$$a \cdot x + b \cdot y = g$$

We can represent $a \bmod b$ as:

$$a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor$$

Substituting this expression in the coefficient equation of (x_1, y_1) gives:

$$g = b \cdot x_1 + (a \bmod b) \cdot y_1 = b \cdot x_1 + (a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b) \cdot y_1$$

and after rearranging the terms:

$$g = a \cdot y_1 + b \cdot (x_1 - y_1 \cdot \left\lfloor \frac{a}{b} \right\rfloor)$$

$$\therefore x = y_1 \text{ and } y = x_1 - y_1 \left\lfloor \frac{a}{b} \right\rfloor$$

9.3 Modulo Inverse

A modular multiplicative inverse of an integer a is an integer x such that $a \cdot x$ is congruent to 1 modular m . We want to find an integer x such that

$$a \cdot x \equiv 1 \pmod{m}.$$

We will also denote x simply with a_1 . Modular inverse exists iff a and m are relatively prime (i.e. $\gcd(a, m) = 1$). With the help of Extended Euclidean algorithm, we can find the modular multiplicative inverse of a in m .

$$a \cdot x + m \cdot y = \gcd(a, m) = 1$$

Here, the modular inverse of a is x .

9.3.1 Computing Inverses of first N numbers

We can easily compute the modular multiplicative inverse of a in m for all $a \in \mathbb{Z}_m$ by using the extended Euclidean algorithm. Suppose we are trying to find the modular inverse for a number a , $a < M$, with respect to M . Now divide M by a . This will be the starting point.

$$M = Q \times a + r, \text{ (where } Q \text{ is the quotient and } r \text{ is the remainder)} \quad (9.1)$$

$$M = \lfloor \frac{M}{a} \rfloor \times a + (M \% a) \quad (9.2)$$

$$\text{Now take modulo } M \text{ on both sides of the equation.} \quad (9.3)$$

$$0 \equiv \lfloor \frac{M}{a} \rfloor \times a + (M \% a) \pmod{M} \quad (9.4)$$

$$(M \% a) \equiv -\lfloor \frac{M}{a} \rfloor \times a \pmod{M} \quad (9.5)$$

Now divide both side by $a \times (M \% a)$.

$$\frac{M \% a}{a \times (M \% a)} \equiv \frac{-\lfloor \frac{M}{a} \rfloor \times a}{a \times (M \% a)} \pmod{M} \quad (9.6)$$

$$a^{-1} \equiv -\lfloor \frac{M}{a} \rfloor \times (M \% a)^{-1} \pmod{M} \quad (9.7)$$

The formula establishes a recurrence relation. The formula says that, in order to find the modular inverse of a in m , we need to find the modular multiplicative inverse of $b = M \% a$ first. Since $b = M \% a$, we can say that its value lies between 0 and $a - 1$. But, a and M are coprime. So a will never fully divide M and hence b will never be zero. So possible values of b are between 1 and $a - 1$. Therefore, if we have all modular inverse from 1 to $a - 1$ already calculated, then we can find the modular inverse of a in $O(1)$. **Time Complexity:** $O(N)$

9.4 Discrete Logarithm

The discrete logarithm of a in b is the smallest integer x such that $a^k \equiv b \pmod{m}$ where a and m are relatively prime. A Naive approach is to run a loop from 0 to m to cover all possible values of k and check for which value of k , the above relation satisfies. If all the values of k exhausted, print -1. Time complexity of this approach is $O(m)$. This approach works because of the cyclic nature of the equation or more so because of the cyclic group property it possess. An efficient approach is to use baby-step, giant-step algorithm by using meet in the middle trick. Given a cyclic group G of order m , a generator a of the group, and a group element b , the problem is to find an integer k such that $a^k \equiv b \pmod{m}$. Here we'll use the meet in the middle trick i.e. split the problem in two parts of each and solve them individually and then find the collision. We can write $k = i \cdot n - j$ with $n = \lceil m \rceil$ and $0 \leq i < n$ and $0 \leq j < n$.

$$a^{i \cdot n - j} \equiv b \pmod{m} \implies a^{i \cdot n} \equiv a^j \cdot b \pmod{m} \quad (9.8)$$

Therefore in order to solve, we precompute a^i for all i and then we can find the value of j by using the equation above. We use unordered maps for this purpose.

9.5 Mod'ed Arithmetic Progression

A mod'ed arithmetic progression is a sequence of numbers $a_0, a_1, a_2, \dots, a_n$ such that $a_i = (a_{i-1} + c) \pmod{m}$ for all $i \in [0, to - 1]$ where $t \in \mathbb{N}$ and $c \in \mathbb{Z}_m$. We need to calculate:

$$\sum_{i=0}^{to-1} a_i = \sum_{i=0}^{to-1} (ki + c) \% m \quad (9.9)$$

We do this by using the formula:

$$(ki + c) = \left\lfloor \frac{ki + c}{m} \right\rfloor m + (ki + c) \% m \quad (9.10)$$

$$(ki + c) \% m = (ki + c) - \left\lfloor \frac{ki + c}{m} \right\rfloor m \quad (9.11)$$

$$\sum_{i=0}^{to-1} (ki + c) \% m = \sum_{i=0}^{to-1} (ki + c) - m \sum_{i=0}^{to-1} \left\lfloor \frac{ki + c}{m} \right\rfloor \quad (9.12)$$

$$(9.13)$$

Here LHS is calculated by **modsum** and $\sum_{i=0}^{to-1} \left\lfloor \frac{ki+c}{m} \right\rfloor$ by **divsum**.

9.6 Primitive Root

In modular arithmetic, a number g is called a primitive root modulo n if every number coprime to n is congruent to a power of g modulo n . Mathematically, g is a primitive root modulo n if and only if for any integer a such that $\gcd(a, n) = 1$, there exists an integer k such that:

9.7 Discrete Root

Problem of finding a discrete root is defined as follows: Given a prime n and two integers a and k , find all x for which $x^k \equiv a \pmod{n}$.

9.7.1 Euler's Criterion (to check whether square root exists)

Given a number n and a prime p , find if square root of n under modulo p exists or not. A number x is square root of n under modulo p if $(x * x) \% p = n \% p$. A Naive Method is to try every number x where x varies from 2 to $p - 1$. For every x , check if $(x * x) \% p$ is equal to $n \% p$.

Euler's criterion states that Square root of n under modulo p exists if and only if $n^{\frac{p-1}{2}} \% p \equiv 1$. As a is coprime to p , Fermat's little theorem says that:

$$a^{p-1} \equiv 1 \pmod{p} \quad (9.14)$$

$$\left(a^{\frac{p-1}{2}} - 1\right) \left(a^{\frac{p-1}{2}} + 1\right) \equiv 1 \pmod{p} \quad (9.15)$$

Since the integers mod p form a field, for each a , one or the other of these factors must be zero. Now if a is a quadratic residue, $a \equiv x^2$.

$$a^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \pmod{p} \quad (9.16)$$

$$(9.17)$$

So every quadratic residue (\pmod{p}) makes the first factor zero. Therefore existence of $n^{\frac{p-1}{2}} \% p \equiv 1$ implies the existence of square root of n under modulo p . Not checked how to prove it the other way around.

9.7.2 Shanks Tonelli's Algorithm

(Thorough Analysis of Algorithm is not done yet.)

The Shanks-Tonelli algorithm is an algorithm for finding a square root of a number under modulo p .

Algorithm steps to find modular square root using shank Tonelli's algorithm :

1. Calculate $n^{\frac{p-1}{2}} \pmod{p}$, it must be 1 or $p - 1$, if it is $p - 1$, then modular square root is not possible.
2. Then after write $p - 1$ as $(s * 2^e)$ for some integer s and e , where s must be an odd number and both s and e should be positive
3. Then find a number q such that $q^{\frac{p-1}{2}} \pmod{p} \equiv -1$.

4. Initialize variable x , b , g and r by following values:

$$x = n^{\frac{s+1}{2}} \text{ (first guess of square root)} \quad (9.18)$$

$$b = n^s \quad (9.19)$$

$$g = q^s \quad (9.20)$$

$$r = e \text{ (exponent } e \text{ will decrease after each updation)} \quad (9.21)$$

5. Now loop until $m > 0$ and update value of x , which will be our final answer.

Find least integer m such that $b^{2^m} = 1 \pmod{p}$ and $0 \leq m \leq r-1$

If $m = 0$, then we found correct answer and return x as result

Else update x , b , g , r as below

$$x = x * g^{2^{r-m-1}} b = b * g^{2^{r-m}} g = g^{2^{r-m}} r = m \quad (9.22)$$

So if m becomes 0 or b becomes 1, we terminate and print the result. This loop guarantees to terminate because value of m is decreased each time after updation.

9.8 Primality Test

9.8.1 Sieve of Eratosthenes

Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment $[1, n]$ using $O(n \log \log n)$ operations. The algorithm is very simple: at the beginning we write down all numbers between 2 and n . We mark all proper multiples of 2 (since 2 is the smallest prime number) as composite. A proper multiple of a number x , is a number greater than x and divisible by x . Then we find the next number that hasn't been marked as composite, in this case it is 3. Which means 3 is prime, and we mark all proper multiples of 3 as composite. The next unmarked number is 5, which is the next prime number, and we mark all proper multiples of it. And we continue this procedure until we processed all numbers in the row. A number is prime, if none of the smaller prime numbers divides it. Since we iterate over the prime numbers in order, we already marked all numbers, who are divisible by at least one of the prime numbers, as divisible. Hence if we reach a cell and it is not marked, then it isn't divisible by any smaller prime number and therefore has to be prime. Obviously, to find all the prime numbers until n , it will be enough just to perform the sifting only by the prime numbers, which do not exceed the root of n .

9.8.2 Miller-Rabin Primality Test

The Miller-Rabin test extends the ideas from the Fermat test. For an odd number n , $n-1$ is even and we can factor out all powers of 2. We can write:

$$n - 1 = 2^s d, \text{ where } d \text{ is odd} \quad (9.23)$$

This allows us to factorize the equation of Fermat's little theorem:

$$a^{n-1} \equiv 1 \pmod{n} \Leftrightarrow a^{2^s d} - 1 \equiv 0 \pmod{n} \quad (9.24)$$

$$\Leftrightarrow (a^{2^{s-1}d} + 1)(a^{2^{s-1}d} - 1) \equiv 0 \pmod{n} \quad (9.25)$$

$$\Leftrightarrow (a^{2^{s-1}d} + 1)(a^{2^{s-2}d} + 1)(a^{2^{s-2}d} - 1) \equiv 0 \pmod{n} \quad (9.26)$$

$$\vdots \quad (9.27)$$

$$\Leftrightarrow (a^{2^{s-1}d} + 1)(a^{2^{s-2}d} + 1) \dots (a^d + 1)(a^d - 1) \equiv 0 \pmod{n} \quad (9.28)$$

If n is prime, then n has to divide one of these factors. And in the Miller-Rabin primality test we check exactly that statement. For a base $2 \leq a \leq n-2$ we check if either $a^d \equiv 1 \pmod{n}$ holds or $a^{2^r d} \equiv -1 \pmod{n}$ holds for some $0 \leq r \leq s-1$.

If we found a base a which doesn't satisfy any of the above equalities, then we found a witness for the compositeness of n . In this case we have proven that n is not a prime number. It is also possible that the set of equations is satisfied for a composite number. In that case the base a is called a *strong liar*. If a base a satisfies the equations (one of them), n is only *strong probable prime*. Except for Carmichael numbers, there are no numbers where only non trivial bases lie. It is also possible to show, that at most $\frac{1}{4}$ of the bases can be strong liars. If n is composite, we have a probability of more than 75% that a random base will tell us that it is composite. By doing multiple iterations, choosing different random bases, we can tell with very high probability if the number is truly prime or if it is composite.

Deterministic version of Miller test

Miller showed that it is possible to make the algorithm deterministic by only checking all bases $\leq O((\ln n)^2)$. Bach later gave a concrete bound, it is only necessary to test all bases $a \leq 2\ln(n)^2$. This is still a pretty large number of bases. So people have invested quite a lot of computation power into finding lower bounds. It turns out, for testing a 32 bit integer it is only necessary to check the first 4 prime bases: 2, 3, 5 and 7. And for testing 64 bit integer it is enough to check the first 12 prime bases: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, and 37.

9.9 Chinese Remainder Theorem

You are given two pairs (main goal is to solve it for t pairs) of integers $(a_1, n_1), (a_2, n_2)$. There is no assumption that n_1 and n_2 are coprime. Find an integer x that satisfies

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases} \quad (9.29)$$

This system of congruences implies that

$$\begin{cases} x = a_1 + n_1 k_1 \\ x = a_2 + n_2 k_2 \end{cases} \quad (9.30)$$

for some integers k_1, k_2 . Let's equate right sides of these equations. We get

$$a_1 + n_1 k_1 = a_2 + n_2 k_2 \quad (9.31)$$

$$n_1(-k_1) + n_2 k_2 = a_1 - a_2 \quad (9.32)$$

Since we know n_1, n_2, a_1, a_2 , this is just linear diophantine equation. Let $d = \text{GCD}(n_1, n_2)$. It divides left-hand side of the equation, so for this equation to have solutions, d must also divide right-hand side which is $a_1 - a_2$. Now, thanks to **Extended Euclidean Algorithm** we can find (x', y') such that

$$n_1 x' + n_2 y' = d \quad (9.33)$$

After multiplying both sides by $\frac{a_1 - a_2}{d}$, we get

$$n_1 x' \frac{a_1 - a_2}{d} + n_2 y' \frac{a_1 - a_2}{d} = a_1 - a_2 \quad (9.34)$$

so $k_1 = -x' \frac{a_1 - a_2}{d}$ and $k_2 = y' \frac{a_1 - a_2}{d}$.

We can substitute k_1 into $x = a_1 + n_1 k_1$ to get our solution: $x = a_1 + x' \frac{a_1 - a_2}{d} n_1$. As expected, there are infinitely many solutions since we are dealing with congruences. Let's say we have two different solutions x_1 and x_2 . Now we have $x_1 \equiv a_1 \pmod{n_1}$ and $x_2 \equiv a_2 \pmod{n_2}$, so from transitivity of congruences we get $x_1 \equiv x_2 \pmod{n_1}$. Doing the same thing for n_2 we get $x_1 \equiv x_2 \pmod{n_2}$. These two congruences are equivalent to

$$x_1 \equiv x_2 \pmod{\text{LCM}(n_1, n_2)} \quad (9.35)$$

It means that any two solutions are congruent modulo $\text{LCM}(n_1, n_2)$.

9.9.1 Case of Multiple Congruences

The last thing to consider is how to handle case of more than 2 congruences. Let's say we have t congruences

$$x \equiv a_i \pmod{n_i} \text{ for } i = 1, 2, \dots, t \quad (9.36)$$

We can just merge equations one by one. After merging first two congruences we get something like $x \equiv s \pmod{\text{LCM}(n_1, n_2)}$ and now we can merge it in the same way with $x \equiv a_3 \pmod{n_3}$ and so on. The complexity of this algorithm is just $\mathcal{O}(t \log \text{LCM}(n_1, n_2, \dots, n_t))$.

9.10 Bézout's identity

For $a \neq 0, b \neq 0$, then $d = \text{gcd}(a, b)$ is the smallest positive integer for which there are integer solutions to

$$ax + by = d$$

If (x, y) is one solution, then all solutions are given by

$$\left(x + \frac{kb}{\text{gcd}(a, b)}, y - \frac{ka}{\text{gcd}(a, b)} \right), \quad k \in \mathbb{Z}$$

Correctness of these solutions can be easily checked by putting them one by one in the Linear diophantine equation.

9.11 Phi Function

Euler's totient function, also known as phi-function $\phi(n)$, counts the number of integers between 1 and n inclusive, which are coprime to n . Two numbers are coprime if their greatest common divisor equals 1 (1 is considered to be coprime to any number).

The following properties of Euler totient function are sufficient to calculate it for any number:

1. If p is a prime number, then $\gcd(p, q) = 1$ for all $1 \leq q < p$. Therefore we have:

$$\phi(p) = p - 1 \quad (9.37)$$

2. If p is a prime number and $k \geq 1$, then there are exactly $\frac{p^k}{p}$ numbers between 1 and p^k that are divisible by p . Which gives us:

$$\phi(p^k) = p^k - p^{k-1} \quad (9.38)$$

3. If a and b are relatively prime, then:

$$\phi(ab) = \phi(a)\phi(b) \quad (9.39)$$

4. This relation is not trivial to see. It follows from the Chinese remainder theorem. The Chinese remainder theorem guarantees, that for each $0 \leq x < a$ and each $0 \leq y < b$, there exists a unique $0 \leq z < ab$ with $z \equiv x \pmod{a}$ and $z \equiv y \pmod{b}$. It's not hard to show that z is coprime to ab if and only if x is coprime to a and y is coprime to b (Not proved myself yet). Therefore the amount of integers coprime to ab is equal to product of the amounts of a and b .

5. In general, for not coprime a and b , the equation

$$\phi(ab) = \phi(a)\phi(b) \frac{d}{\phi(d)} \quad (9.40)$$

with $d = \gcd(a, b)$ holds.

Thus, using the first three properties, we can compute $\phi(n)$ through the factorization of n (decomposition of n into a product of its prime factors). If $n = p_1^{a_1} \cdot p_2^{a_2} \cdots p_k^{a_k}$, where p_i are prime factors of n .

$$\phi(n) = \phi(p_1^{a_1})\phi(p_2^{a_2}) \cdots \phi(p_k^{a_k}) \quad (9.41)$$

$$= (p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1}) \cdots (p_k^{a_k} - p_k^{a_k-1}) \quad (9.42)$$

$$= n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k}) \quad (9.43)$$

9.12 Heaps

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

1. Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

Operations on Min Heap:

1. getMini(): It returns the root element of Min Heap. Time Complexity of this operation is $O(1)$.
2. extractMin(): Removes the minimum element from MinHeap. Time Complexity of this Operation is $O(\log n)$ as this operation needs to maintain the heap property (by calling heapify()) after removing root.

3. `decreaseKey()`: Decreases value of key. The time complexity of this operation is $O(\log n)$. If the decreases key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
4. `insert()`: Inserting a new key takes $O(\log n)$ time. We add a new key at the end of the tree. IF new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
5. `delete()`: Deleting a key also takes $O(\log n)$ time. We replace the key to be deleted with minum infinite by calling `decreaseKey()`. After `decreaseKey()`, the minus infinite value must reach root, so we call `extractMin()` to remove the key.

9.13 Eulerian Graph

A graph is called Eulerian if it has an Eulerian Cycle and called Semi-Eulerian if it has an Eulerian Path. Eulerian Path is a path in graph that visits every edge exactly once. Eulerian Circuit is an Eulerian Path which starts and ends on the same vertex.

How to find whether a given graph is Eulerian or not?

The problem is same as following question. "Is it possible to draw a given graph without lifting pencil from the paper and without tracing any of the edges more than once". We can find it in polynomial time. $O(V + E)$ is the expected time complexity.

Eulerian Cycle : An undirected graph has Eulerian cycle if following two conditions are true.

1. All vertices with non-zero degree are connected. We don't care about vertices with zero degree because they don't belong to Eulerian Cycle or Path (we only consider all edges).
2. All vertices have even degree.

An undirected graph has Eulerian Path if following two conditions are true.

1. Same as condition (a) for Eulerian Cycle
2. If zero or two vertices have odd degree and all other vertices have even degree. Note that only one vertex with odd degree is not possible in an undirected graph

In Eulerian path, each time we visit a vertex v , we walk through two unvisited edges with one end point as v . Therefore, all middle vertices in Eulerian Path must have even degree. For Eulerian Cycle, any vertex can be middle vertex, therefore all vertices must have even degree.

We can use the following strategy: We find all simple cycles and combine them into one - this will be the Eulerian cycle. If the graph is such that the Eulerian path is not a cycle, then add the missing edge, find the Eulerian cycle, then remove the extra edge.

Process to Find the Path:

1. First, take an empty stack and an empty path.
2. If all the vertices have an even number of edges then start from any of them. If two of the vertices have an odd number of edges then start from one of them. Set variable current to this starting vertex.
3. If the current vertex has at least one adjacent node then first discover that node and then discover the current node by backtracking. To do so add the current node to stack, remove the edge between the current node and neighbor node, set current to the neighbor node.
4. If the current node has not any neighbor then add it to the path and pop stack set current to popped vertex.
5. Repeat process 3 and 4 until the stack is empty and the current node has not any neighbor.

9.14 Minimum Spanning Trees

A minimum spanning tree (MST) is a subgraph of a connected graph that is a tree and connects all the vertices together with minimum possible weight.

9.14.1 Prim's Algorithm

The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily) in Prim's Algorithm. Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices.

Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices. (or equivalently until we have $n-1$ edges).

In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n - 1$.

Proof: Let the graph G be connected, i.e. the answer exists. We denote by T the resulting graph found by Prim's algorithm, and by S the minimum spanning tree. Obviously T is indeed a spanning tree and a subgraph of G . We only need to show that the weights of S and T coincide. Consider the first time in the algorithm when we add an edge to T that is not part of S . Let us denote this edge with e , its ends by a and b , and the set of already selected vertices as V ($a \in V$ and $b \notin V$, or visa versa).

In the minimal spanning tree S the vertices a and b are connected by some path P . On this path we can find an edge f such that one end of f lies in V and the other end doesn't. Since the algorithm chose e instead of f , it means that the weight of f is greater or equal to the weight of e . We add the edge e to the minimum spanning tree S and remove the edge f . By adding e we created a cycle, and since f was also part of the only cycle, by removing it the resulting graph is again free of cycles. And because we only removed an edge from a cycle, the resulting graph is still connected.

The resulting spanning tree cannot have a larger total weight, since the weight of e was not larger than the weight of f , and it also cannot have a smaller weight since S was a minimum spanning tree. This means that by replacing the edge f with e we generated a different minimum spanning tree. And e has to have the same weight as f . Thus all the edges we pick in Prim's algorithm have the same weights as the edges of any minimum spanning tree, which means that Prim's algorithm really generates a minimum spanning tree.

9.14.2 Kruskal's Algorithm

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the smallest weight edge that does not cause a cycle in the MST constructed so far.

Proof:

1. Sort the edges in non-decreasing order of their weights.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step 2 until there are (V-1) edges in the spanning tree.

Step 2 uses the Union-Find algorithm to detect cycles.

9.15 Shortest Path

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

9.15.1 Dijkstra's Algorithm

The algorithm is a Greedy Algorithm. The Greedy Choice is to pick the vertex with the smallest distance from the source vertex.

How Dijkstra's Algorithm works?

Dijkstra's Algorithm works on the basis that any subpath $B \rightarrow D$ of the shortest path $A \rightarrow D$ between vertices A and D is also the shortest path between vertices B and D . Dijkstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors. The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Pseudocode: We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

```
function dijkstra(G, S)
for each vertex V in G
    distance[V] <- infinite
    previous[V] <- NULL
    If V != S, add V to Priority Queue Q
distance[S] <- 0

while Q IS NOT EMPTY
    U <- Extract MIN from Q
    for each unvisited neighbour V of U
        tempDistance <- distance[U] + edge.weight(U, V)
        if tempDistance < distance[V]
            distance[V] <- tempDistance
            previous[V] <- U
return distance[], previous[]
```

9.15.2 Bellman-Ford Algorithm

This algorithm is said to be solution to single source shortest path with negative weight edges. The only difference between BellmanFord and Dijkstra's Algorithm is that BellmanFord is used when the graph may contain negative weight edges and BellmanFord does not use a priority queue.

1. This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array $dist[]$ of size $|V|$ with all values as infinite except $dist[src]$ where src is source vertex.
2. This step calculates shortest distances. Do following $|V| - 1$ times where $|V|$ is the number of vertices in given graph.
3. Do following for each edge $u - v$.
 - (a) If $dist[v] > dist[u] + weight\ of\ edge\ u - v$, then update $dist[v]$ by $dist[u] + weight\ of\ edge\ u - v$.
4. This step checks for negative weight cycle. If we get a shorter path to a vertex from the source vertex, then there is a negative weight cycle. Do following for each edge $u - v$:
 If $dist[v] > dist[u] + weight\ of\ edge\ u - v$, then Graph contains negative weight cycle
 The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

Pseudocode: We need to maintain the path distance of every vertex. We can store that in an array of size v , where v is the number of vertices. We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length. Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

```

for each vertex V in G
for each edge (U,V) in G
    tempDistance ← distance[U] + edge-weight(U, V)
    if tempDistance < distance[V]
        distance[V] ← tempDistance
        previous[V] ← U

for each edge (U,V) in G
    If distance[U] + edge-weight(U, V) < distance[V]
        Error: Negative Cycle Exists

return distance [], previous []
    
```

9.15.3 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs. But, it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

1. Create a matrix A^0 of dimension n^2 where n is the number of vertices. The row and the column are indexed as i and j respectively. i and j are the vertices of the graph. Each cell $A[i][j]$ is filled with the distance from the i^{th} vertex to the j^{th} vertex. If there is no path from i^{th} vertex to j^{th} vertex, the cell is left as infinity.
2. Fill the matrix A^0 with the given graph.
3. Now, create a matrix A^1 using matrix A^0 . The elements in the first column and the first row are left as they are. The remaining cells are filled in the following way. Let k be the intermediate vertex in the shortest path from source to destination. In this step, k is the first vertex. $A[i][j]$ is filled with $(A[i][k] + A[k][j])$ if $(A[i][j] > A[i][k] + A[k][j])$. That is, if the direct distance from the source to the destination is greater than the path through the vertex k , then the cell is filled with $A[i][k] + A[k][j]$.
4. Now, create a matrix A^2 using matrix A^1 . The elements in the second column and the second row are left as they are. The remaining cells are filled in similar manner as in step 2.
5. We keep on repeating the above steps until we reach the last step i.e. till we calculate $A^{|V|}$.

Pseudocode:

```

n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            Ak[i, j] = min (Ak-1[i, j], Ak-1[i, k] + Ak-1[k, j])

return A
    
```

9.16 Maximum flow - Ford-Fulkerson and Edmonds-Karp

A network is a directed graph G with vertices V and edges E combined with a function c , which assigns each edge $e \in E$ a non-negative integer value, the capacity of e . Such a network is called a **flow network**, if we additionally label two vertices, one as **source** and one as **sink**.

A **flow** in a flow network is function f , that again assigns each edge e a non-negative integer value, namely the flow. The function has to fulfill the following two conditions:

1. The flow of an edge cannot exceed the capacity.

$$f(e) \leq c(e)$$

(9.44)

$$f(e) = 0 \quad \text{if} \quad e \notin E$$

(9.45)

2. Sum of the incoming flow of a vertex u has to be equal to the sum of the outgoing flow of u except in the source and sink vertices.

$$\sum_{(v,u) \in E} f((v,u)) = \sum_{(u,v) \in E} f((u,v))$$

(9.46)

(9.47)

The source vertex s only has an outgoing flow, and the sink vertex t has only incoming flow. It is easy to see that the following equation holds:

$$\sum_{(s,u) \in E} f((s,u)) = \sum_{(u,t) \in E} f((u,t))$$

(9.48)

(9.49)

The value of a flow of a network is the sum of all flows that gets produced in source s , or equivalently of the flows that are consumed in the sink t . A maximal flow is a flow with the maximal possible value. Finding this maximal flow of a flow network is the problem that we want to solve.

9.16.1 Ford-Fulkerson Algorithm

A **residual capacity** of an directed edge is the capacity minus the flow. It should be noted that if there is a flow along some directed edge (u, v) , then the reversed edge has capacity 0 and we can define the flow of it as $f((v, u)) = -f((u, v))$. This also defines the residual capacity for all reversed edges. From all these edges we can create a residual network, which is just a network with the same vertices and same edges, but we use the residual capacities as capacities.

The Ford-Fulkerson method works as follows. First we set the flow of each edge to zero. Then we look for an augmenting path from s to t . An **augmenting path** is simple path in the residual graph, i.e. along the edges whose residual capacity is positive. If such a path is found, then we can increase the flow along these edges. We keep on searching for augmenting paths and increasing the flow. Once there doesn't exists an augmenting path any more, the flow is maximal. Let us specify in more detail, what increasing the flow along an augmenting path means. Let C be the smallest residual capacity of the edges in the path. Then we increase the flow in the following way: we update $f((u, v)) + = C$ and $f((v, u)) - = C$ for every edge (u, v) in the path. Ford-Fulkerson method doesn't specify a method of finding the augmenting path. Possible approaches are using DFS or BFS which both work in $O(E)$. If all capacities of the network are integers, then for each augmenting path the flow of the network increases by at least 1 (for more details see Integral flow theorem). Therefore the complexity of Ford-Fulkerson is $O(EF)$, where F is the maximal flow of the network. In case of rational capacities, the algorithm will also terminate, but the complexity is not bounded. In case of irrational capacities, the algorithm might never terminate, and might not even converge to the maximal flow.

9.16.2 Edmonds-Karp Algorithm

Edmonds-Karp algorithm is just an implementation of the Ford-Fulkerson method that uses BFS for finding augmenting paths. The complexity can be given independently of the maximal flow. The algorithm runs in $O(VE^2)$ time, even for irrational capacities. The intuition is, that every time we find an augmenting path one of the edges becomes saturated, and the distance from the edge to s will be longer, if it appears later again in an augmenting path. And the length of a simple paths is bounded by V . Edmonds-Karp algorithm can also be thought of as a method of augmentation which repeatedly finds the shortest augmenting path from $s \rightarrow t$ in terms of number of edges used in each iteration. Using a BFS to find augmenting paths ensures that the shortest path from $s \rightarrow t$ is found every iteration.

Pseudocode:

```
inputs
C[n x n] : Capacity Matrix
E[n x n] : Adjacency Matrix
s : source
t : sink
output
f : maximum flow
Edmonds-Karp:
    f = 0 // Flow is initially 0
    F = [n x n] // residual capacity array
    while true:
        m, P = Breadth-First-Search(C, E, s, t, F)
```

```

    if m = 0:
        break
    f = f + m
    v = t
    while v != s:
        u = P[v]
        F[u, v] = F[u, v] - m           //This is reducing the residual capacity of the augmenting path
        F[v, u] = F[v, u] + m           //This is increasing the residual capacity of the reverse edges
        v = u
    return f

```

In this pseudo-code, the flow is initially zero and the initial residual capacity array is all zeroes. Then, the outer loop executes until there are no more paths from the source to the sink in the residual graph. Inside this loop, we are performing breadth-first search to find the shortest path from the source to the sink that has available capacity. Once we have found a path with residual capacity, m , we add that capacity to our current maximum flow. Then, the code backtracks through the network, starting with the sink t . As it backtracks, it finds the parent of the current vertex which is defined as u . Then, the code updates the residual flow matrix F to reflect the newly found augmenting path capacity m . v is then set to be the parent, and the inner loop is repeated until it reaches the source vertex.

Max-flow min-cut theorem

A **s-t-cut** is a partition of the vertices of a flow network into two sets, such that a set includes the source s and the other one includes the sink t . The capacity of a **s-t-cut** is defined as the sum of capacities of the edges from the source side to the sink side. Obviously we cannot send more flow from s to t than the capacity of any **s-t-cut**. Therefore the maximum flow is bounded by the minimum cut capacity. The max-flow min-cut theorem goes even further. It says that the capacity of the maximum flow has to be equal to the capacity of the minimum cut.

Integral flow Theorem

The theorem simply says, that if every capacity in the network is integer, then the flow in each edge will be integer in the maximal flow.

9.16.3 Dinic's Algorithm

This is a faster algorithm than Edmonds Karp algorithm taking $O(V^2E)$ time. Like Edmond Karp's algorithm, Dinic's algorithm uses following concepts :

1. A flow is maximum if there is no s to t path in residual graph.
2. BFS is used in a loop. There is a difference though in the way we use BFS in both algorithms.

In Edmond's Karp algorithm, we use BFS to find an augmenting path and send flow across this path. In Dinic's algorithm, we use BFS to check if more flow is possible and to construct level graph. In level graph, we assign levels to all nodes, level of a node is shortest distance (in terms of number of edges) of the node from source. Once level graph is constructed, we send multiple flows using this level graph. This is the reason it works better than Edmond Karp. In Edmond Karp, we send only flow that is send across the path found by BFS.

Outline:

1. Initialize residual graph G as given graph.
2. Do BFS of G to construct a level graph (or assign levels to vertices) and also check if more flow is possible.
 - (a) If more flow is not possible, then return.
 - (b) Send multiple flows in G using level graph until blocking flow is reached. Here using level graph means, in every flow, levels of path nodes should be 0, 1, 2... (in order) from s to t.

9.17 Vertex Cover

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u, v) of the graph, either u or v is in the vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph. Given an undirected graph, the vertex cover problem is to find minimum size vertex cover. Vertex Cover Problem is a known NP Complete problem, i.e., there is no polynomial-time solution for this unless $P = NP$. There are approximate polynomial-time algorithms to solve the problem though.

Outline:

1. Initialize the result as
2. Consider a set of all edges in given graph. Let the set be E.
3. Do following while E is not empty
 - (a) Pick an arbitrary edge (u, v) from set E and add 'u' and 'v' to result
 - (b) Remove all edges from E which are either incident on u or v.

4. Return result

It can be proved that the above approximate algorithm never finds a vertex cover whose size is more than twice the size of the minimum possible vertex cover. The Time Complexity of the above algorithm is $O(V + E)$. Although the problem is NP complete, it can be solved in polynomial time for the following types of graphs:

1. Bipartite Graph
2. Tree Graph

9.17.1 Vertex Cover for a Tree

The idea is to consider following two possibilities for root and recursively for all nodes down the root.

1. Root is part of vertex cover: In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).
2. Root is not part of vertex cover: In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

For n-ary trees:

1. For every node, if we exclude this node from vertex cover than we have to include its neighbouring nodes, and if we include this node in the vertex cover than we will take the minimum of the two possibilities of taking its neighbouring nodes in the vertex cover to get minimum vertex cover.
2. We will store the above information in the dp array.

9.17.2 Vertex Cover for a Bipartite Graph

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge (u, v) either connects a vertex from U to V or a vertex from V to U. In other words, for every edge (u, v), either u belongs to U and v to V, or u belongs to V and v to U. We can also say that there is no edge that connects vertices of same set.

Konig's Theorem: If G is bipartite, the cardinality of the maximum matching is equal to the cardinality of the minimum vertex cover.

9.18 Strongly Connected Components

A directed graph is strongly connected if there is a path between all pairs of vertices. A strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. We can find all strongly connected components in $O(V+E)$ time using Kosaraju's algorithm. It is based on the depth-first search algorithm implemented twice.

Outline:

1. Create an empty stack 'S' and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack.
2. Reverse directions of all arcs to obtain the transpose graph.
3. One by one pop a vertex from S while S is not empty. Let the popped vertex be 'v'. Take v as source and do DFS. The DFS starting from v prints strongly connected component of v.

How does this work?

DFS of a graph produces a single tree if all vertices are reachable from the DFS starting point. Otherwise DFS produces a forest. So DFS of a graph with only one SCC always produces a tree. The important point to note is DFS may produce a tree or a forest when there are more than one SCCs depending upon the chosen starting point.

However, if we do a DFS of graph and store vertices according to their finish times, we make sure that the finish time of a vertex that connects to other SCCs (other than its own SCC), will always be greater than finish time of vertices in the other SCC (trivial 2 cases).

In the next step, we reverse the graph. Consider the graph of SCCs. In the reversed graph, the edges that connect two components are reversed. So if we do a DFS of the reversed graph using sequence of vertices in stack, we process vertices from sink to source (in reversed graph). That is what we wanted to achieve and that is all needed to print SCCs one by one.

9.19 2-SAT

SAT (Boolean satisfiability problem) is the problem of assigning Boolean values to variables to satisfy a given Boolean formula. The Boolean formula will usually be given in CNF (conjunctive normal form), which is a conjunction of multiple clauses, where each clause is a disjunction of literals (variables or negation of variables).

2-SAT (2-satisfiability) is a restriction of the SAT problem, in 2-SAT every clause has exactly two literals. Here is an example of such a 2-SAT problem. Find an assignment of a,b,c such that the following formula is true:

$$(a \vee b') \wedge (a' \vee b) \wedge (a' \vee b') \wedge (a \vee c') \quad (9.50)$$

SAT is NP-complete, there is no known efficient solution known for it. However 2SAT can be solved efficiently in $O(n + m)$ where n is the number of variables and m is the number of clauses. First we need to convert the problem to a different form, the so-called implicative normal form. Note that the expression $a \vee b$ is equivalent to $(a' \Rightarrow b) \wedge (b' \Rightarrow a)$. We now construct a directed graph of these implications: for each variable x there will be two vertices v_x and $v_{x'}$. The edges will correspond to the implications. Let's look at the example in 2-CNF form:

$$(a \vee b') \wedge (a' \vee b) \wedge (a' \vee b') \wedge (a \vee c') \quad (9.51)$$

The oriented graph will contain the following vertices and edges:

$$\begin{array}{cccc} (a' \Rightarrow b') & (a \Rightarrow b) & (a \Rightarrow b') & (a' \Rightarrow c') \\ (b \Rightarrow a) & (b' \Rightarrow a') & (b \Rightarrow a') & (c \Rightarrow a) \end{array} \quad (9.52)$$

It is worth paying attention to the property of the implication graph: if there is an edge $(a \Rightarrow b)$, then there also is an edge $(b' \Rightarrow a')$. Also note, that if x is reachable from x' , and x' is reachable from x , then the problem has no solution. Whatever value we choose for the variable x , it will always end in a contradiction - if x will be assigned true then the implication tell us that x' should also be true and visa versa. It turns out, that this condition is not only necessary, but also sufficient. If a vertex is reachable from a second one, and the second one is reachable from the first one, then these two vertices are in the same strongly connected component. Therefore we can formulate the criterion for the existence of a solution as follows:

In order for this 2-SAT problem to have a solution, it is necessary and sufficient that for any variable x the vertices x and x' are in different strongly connected components of the strong connection of the implication graph. This criterion can be verified in $O(n + m)$ time by finding all strongly connected components.

Note that, in spite of the fact that the solution exists, it can happen that x' is reachable from x in the implication graph, or that (but not simultaneously) x is reachable from x' . In that case the choice of either true or false for x will lead to a contradiction, while the choice of the other one will not.

Let us sort the strongly connected components in topological order (i.e. $comp[v] \leq comp[u]$ if there is a path from v to u) and let $comp[v]$ denote the index of strongly connected component to which the vertex v belongs. Then, if $comp[x] < comp[x']$ we assign x with false and true otherwise. The proof can be easily understood if you think carefully about it :)

So we have constructed an algorithm that finds the required values of variables under the assumption that for any variable x the vertices x and x' are in different strongly connected components.

Outline:

Now we can implement the entire algorithm. First we construct the graph of implications and find all strongly connected components. This can be accomplished with Kosaraju's algorithm in $O(n + m)$ time. In the second traversal of the graph Kosaraju's algorithm visits the strongly connected components in topological order, therefore it is easy to compute $comp[v]$ for each vertex v .

Afterwards we can choose the assignment of x by comparing $comp[x]$ and $comp[x']$. If $comp[x] = comp[x']$ we return false to indicate that there doesn't exist a valid assignment that satisfies the 2-SAT problem.

Implementation of the solution of the 2-SAT problem for the already constructed graph of implication G and the transpose graph g^T (in which the direction of each edge is reversed). In the graph the vertices with indices $2k$ and $2k + 1$ are the two vertices corresponding to variable k with $2k + 1$ corresponding to the negated variable.

9.20 Diameter of a Tree

The diameter of a tree is the maximum length of a path between two nodes.

Here, I will discuss two $O(n)$ time algorithms for calculating the diameter of a tree. The first algorithm is based on dynamic programming, and the second algorithm uses two depth-first searches.

Algorithm 1:

A general way to approach many tree problems is to first root the tree arbitrarily. After this, we can try to solve the problem separately for each subtree. Our first algorithm for calculating the diameter is based on this idea.

An important observation is that every path in a rooted tree has a highest point: the highest node that belongs to the path. Thus, we can calculate for each node the length of the longest path whose highest point is the node. One of those paths corresponds to the diameter of the tree. We calculate for each node x two values:

1. **toLeaf(x)**: the maximum length of a path from x to any leaf
2. **maxLength(x)**: the maximum length of a path whose highest point is x

Dynamic programming can be used to calculate the above values for all nodes in $O(n)$ time. First, to calculate **toLeaf(x)**, we go through the children of x , choose a child c with maximum **toLeaf(c)** and add one to this value. Then, to calculate **maxLength(x)**, we choose two distinct children a and b such that the sum **toLeaf(a) + toLeaf(b)** is maximum and add two to this sum.

Algorithm 2:

Another efficient way to calculate the diameter of a tree is based on two depth- first searches. First, we choose an arbitrary node a in the tree and find the farthest node b from a . Then, we find the farthest node c from b . The diameter of the tree is the distance between b and c . Why this works? : Whenever we run DFS we always reach one of the end points of diameter of the tree and from that node we are just finding the farthest node which turns out to be the diameter of the tree. This can be easily proved by proof of contradiction.

9.21 Catalan Numbers

Catalan numbers is a number sequence, which is found useful in a number of combinatorial problems, often involving recursively-defined objects. The Catalan number C_n equals the number of valid parenthesis expressions that consist of n left parentheses and n right parentheses.

For example, $C_3 = 5$, because we can construct the following parenthesis expressions using three left and right parentheses:

IIITH:

- $()()()$
- $((()))$
- $()(())$
- $((()))$
- $((()))$

What is exactly a valid parenthesis expression? The following rules precisely define all valid parenthesis expressions:

1. An empty parenthesis expression is valid.
2. If an expression A is valid, then also the expression (A) is valid.
3. If expressions A and B are valid, then also the expression AB is valid.
4. If we choose any prefix of such an expression, it has to contain at least as many left parentheses as right parentheses.
5. The complete expression has to contain an equal number of left and right parentheses.

Catalan numbers can be calculated using the formula:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} \quad (9.53)$$

The sum goes through the ways to divide the expression into two parts such that both parts are valid expressions and the first part is as short as possible but not empty. For any i , the first part contains $i + 1$ pairs of parentheses and the number of expressions is the product of the following values:

1. C_i : the number of ways to construct an expression using the parentheses of the first part, not counting the outermost parentheses
2. C_{n-i-1} : the number of ways to construct an expression using the parentheses of the second part

The base case is $C_0 = 1$, because we can construct an empty parenthesis expression using zero pairs of parentheses.

Catalan numbers can also be calculated using binomial coefficients:

$$C_n = \frac{1}{n+1} \binom{2n}{n} \quad (9.54)$$

The formula can be explained as follows:

There are a total of $2n$ ways to construct a (not necessarily valid) parenthesis expression that contains n left parentheses and n right parentheses. Let us calculate the number of such expressions that are not valid. If a parenthesis expression is not valid, it has to contain a prefix where the number of right parentheses exceeds the number of left parentheses. The idea is to reverse each parenthesis that belongs to such a prefix. For example, the expression $()()()$ contains a prefix $()$, and after reversing the prefix, the expression becomes $)((())$. The resulting expression consists of $n + 1$ left and $n - 1$ right parentheses. The number of such expressions is $\binom{2n}{n+1}$ which equals the number of non-valid parenthesis expressions. Thus, the number of valid parenthesis expressions can be calculated using the formula:

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1} \binom{2n}{n} = \frac{1}{n+1} \binom{2n}{n} \quad (9.55)$$

Catalan numbers are also related to trees:

1. there are C_n binary trees of n nodes
2. there are C_{n-1} rooted trees of n nodes

9.22 Derangements

As an example, let us count the number of derangements of elements $\{1, 2, \dots, n\}$, i.e., permutations where no element remains in its original place. For example, when $n = 3$, there are two derangements: $(2, 3, 1)$ and $(3, 1, 2)$.

One approach for solving the problem is to use inclusion-exclusion. Let X_k be the set of permutations that contain the element k at position k . For example, when $n = 3$, the sets are as follows:

$$X_1 = \{(1, 2, 3), (1, 3, 2)\} X_2 = \{(1, 2, 3), (3, 2, 1)\} X_3 = \{(1, 2, 3), (2, 1, 3)\} \quad (9.56)$$

Using these sets, the number of derangements equals

$$n! - |X_1 \cup X_2 \cup X_3 \cup \dots \cup X_n| \quad (9.57)$$

so it suffices to calculate the size of the union. Using inclusion-exclusion, this reduces to calculating sizes of intersections which can be done efficiently. For example, when $n = 3$, the size of $|X_1 \cup X_2 \cup X_3|$ is

$$|X_1| + |X_2| + |X_3| - |X_1 \cap X_2| - |X_1 \cap X_3| - |X_2 \cap X_3| + |X_1 \cap X_2 \cap X_3| \quad (9.58)$$

so the number of solutions is $3! - 4 = 2$.

It turns out that the problem can also be solved without using inclusion-exclusion. Let $f(n)$ denote the number of derangements for $\{1, 2, \dots, n\}$. We can use the following recursive formula:

$$f(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ (n-1)(f(n-2) + f(n-1)) & n > 2 \end{cases} \quad (9.59)$$

The formula can be derived by considering the possibilities how the element 1 changes in the derangement. There are $n-1$ ways to choose an element x that replaces the element 1. In each such choice, there are two options:

1. We also replace the element x with the element 1. After this, the remaining task is to construct a derangement of $n-2$ elements.
2. We replace the element x with some other element than 1. Now we have to construct a derangement of $n-1$ element, because we cannot replace the element x with the element 1, and all other elements must be changed.

9.23 Strings

Let us first clear important terminology associated with the Strings.

Subsequence: A subsequence is a sequence of (not necessarily consecutive) characters in a string in their original order. A string of length n has $2^n - 1$ subsequences.

Substring: A substring is a sequence of consecutive characters in a string. We use the notation $s[a..b]$ to refer to a substring of s that begins at position a and ends at position b . A string of length n has $\frac{n(n+1)}{2}$ substrings.

Prefix: A prefix is a substring that starts at the beginning of a string.

Suffix: A suffix is a substring that ends at the end of a string.

Period: A period is a prefix of a string such that the string can be constructed by repeating the period. The last repetition may be partial and contain only a prefix of the period.

Border: A border is a string that is both a prefix and a suffix of a string. For example, the borders of ABACABA are A, ABA and ABACABA.

Strings are compared using the lexicographical order (which corresponds to the alphabetical order). It means that $x < y$ if either $x \neq y$ and x is a prefix of y , or there is a position k such that $x[i] = y[i]$ when $i < k$ and $x[k] < y[k]$.

9.23.1 Tries:

A trie is a rooted tree that maintains a set of strings. Each string in the set is stored as a chain of characters that starts at the root. If two strings have a common prefix, they also have a common chain in the tree. The character * in a node means that a string in the set ends at the node. Such a character is needed, because a string may be a prefix of another string.

We can check in $O(n)$ time whether a trie contains a string of length n , because we can follow the chain that starts at the root node. We can also add a string of length n to the trie in $O(n)$ time by first following the chain and then adding new nodes to the trie if necessary.

Using a trie, we can find the longest prefix of a given string such that the prefix belongs to the set. Moreover, by storing additional information in each node, we can calculate the number of strings that belong to the set and have a given string as a prefix.

A trie can be stored in an array: `int trie[N][A]`.

where N is the maximum number of nodes (the maximum total length of the strings in the set) and A is the size of the alphabet. The nodes of a trie are numbered $0, 1, 2, \dots$ so that the number of the root is 0, and $trie[s][c]$ is the next node in the chain when we move from node s using character c .

Trie is an efficient information reTrieval data structure. Using Trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using Trie, we can search the key in $O(M)$ time. However the penalty is on Trie storage requirements.

We can also implement trie structure using pointers instead of array. Every node of Trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as end of word node. A Trie node field `isEndOfWord` is used to distinguish the node as end of word node. A simple structure to represent nodes of the English alphabet can be as following,

```
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    // isEndOfWord is true if the node
    // represents end of a word
    bool isEndOfWord;
};
```

Inserting a string: Every character of the input string is inserted as an individual Trie node. Note that the children is an array of pointers (or references) to next level trie nodes. The string character acts as an index into the array children. If the input string is new or an extension of the existing string, we need to construct non-existing nodes of the string, and mark end of the word for the last node. I If the input string is a prefix of the existing string in Trie, we simply mark the last node of the string as the end of a word. The string length determines Trie depth.

Searching for a string: Searching Recursive function to delete a key from given Trie If tree is empty If last character of key is being processed This node is no more end of word after removal of given key If given is not prefix of any other word If not last character, recur for the child obtained using ASCII value If root does not have any child (its only child got deleted), and it is not end of another word. for a string is similar to insert operation, however, we only compare the characters and move down. The search can terminate due to the end of a string or lack of string in the trie. In the former case, if the `isEndofWord` field of the last node is true, then the string exists in the trie. In the second case, the search terminates without examining all the characters of the string, since the string is not present in the trie.

Insert and search costs $O(string_length)$, however the memory requirements of Trie is $O(ALPHABET_SIZE * string_length * N)$ where N is number of strings in Trie.

Deleting a string: During delete operation we delete the string in bottom up manner using recursion. The following are possible conditions when deleting string from trie,

- 1. String may not be there in trie. Delete operation should not modify trie.
- 2. String present as unique string (no part of string contains another string (prefix), nor the string itself is prefix of another string in trie). Delete all the nodes.
- 3. String is prefix string of another long string in trie. Unmark the leaf node.
- 4. String present in trie, having atleast one other string as prefix string. Delete nodes from end of string until first leaf node of longest prefix string.

The time complexity of the deletion operation is $O(string_length)$.

9.23.2 String Hashing:

String hashing is a technique that allows us to efficiently check whether two strings are equal or not. The idea in string hashing is to compare hash values of strings instead of their individual characters. A hash value of a string is a number that is calculated from the characters of the string. If two strings are the same, their hash values are also the same, which makes it possible to compare strings based on their hash values.

A usual way to implement string hashing is polynomial hashing, which means that the hash value of a string s of length n is

$$(s[0]A^{n-1} + s[1]A^{n-2} + \dots + s[n-1]A^0) \bmod B \tag{9.60}$$

where $s[0], s[1], \dots, s[n - 1]$ are interpreted as the ASCII codes of the characters of s , and A and B are carefully chosen constants. It is reasonable to make A a prime number roughly equal to the number of characters in the input alphabet. Obviously m should be a large number since the probability of two random strings colliding is about $\approx \frac{1}{m}$. A good choice for m is some large prime number. For example, the codes of the characters of ALLEY are:

A	L	L	E	Y
65	76	76	69	89

(9.61)

Thus, if $A = 3$ and $B = 97$, the hash value of ALLEY is $(65 \cdot 3^4 + 76 \cdot 3^3 + 76 \cdot 3^2 + 69 \cdot 3^1 + 89 \cdot 3^0) \bmod 97 = 52$.

Preprocessing:
Using polynomial hashing, we can calculate the hash value of any substring of a string s in $O(1)$ time after an $O(n)$ time preprocessing. The idea is to construct an array h such that $h[k]$ contains the hash value of the prefix $s[0\dots k]$. The array values can be recursively calculated as follows:

$$h[0] = s[0]$$

(9.62)

$$h[k] = (h[k - 1]A + s[k]) \bmod B$$

(9.63)

In addition, we construct an array p where $p[k] = A^k \bmod B$:

$$p[0] = 1$$

(9.64)

$$p[k] = (p[k - 1]A) \bmod B$$

(9.65)

Constructing these arrays takes $O(n)$ time. After this, the hash value of any substring $s[a\dots b]$ can be calculated in $O(1)$ time using the formula:

$$(h[b] - h[a - 1]p[b - a + 1]) \bmod B$$

(9.66)

assuming that $a > 0$. If $a = 0$, the hash value is simply $h[b]$. We can efficiently compare strings using hash values. Instead of comparing the individual characters of the strings, the idea is to compare their hash values. If the hash values are equal, the strings are probably equal, and if the hash values are different, the strings are certainly different. Using hashing, we can often make a brute force algorithm efficient. As an example, consider the pattern matching problem: given a string s and a pattern p , find the positions where p occurs in s . A brute force algorithm gives time complexity of $O(n^2)$. We can make the brute force algorithm more efficient by using hashing, because the algorithm compares substrings of strings. Using hashing, each comparison only takes $O(1)$ time, because only hash values of substrings are compared. This results in an algorithm with time complexity $O(n)$, which is the best possible time complexity for this problem. By combining hashing and binary search, it is also possible to find out the lexicographic order of two strings in logarithmic time. This can be done by calculating the length of the common prefix of the strings using binary search. Once we know the length of the common prefix, we can just check the next character after the prefix, because this determines the order of the strings.

9.23.3 Z-algorithm

The Z-array z of a string s of length n contains for each $k = 0, 1, \dots, n - 1$ the length of the longest substring of s that begins at position k and is a prefix of s . Thus, $z[k] = p$ tells us that $s[0\dots p - 1]$ equals $s[k\dots k + p - 1]$. Many string processing problems can be efficiently solved using the Z-array. In other words, $z[i]$ is the length of the longest string that is, at the same time, a prefix of s and a prefix of the suffix of s starting at i . Next we describe an algorithm, called the **Z-algorithm**, that efficiently constructs the Z-array in $O(n)$ time. The algorithm calculates the Z-array values from left to right by both using information already stored in the Z-array and comparing substrings character by character. To efficiently calculate the Z-array values, the algorithm maintains a range $[x, y]$ such that $s[x\dots y]$ is a prefix of s and y is as large as possible. Since we know that $s[0\dots y - x]$ and $s[x\dots y]$ are equal, we can use this information when calculating Z-values for positions $x + 1, x + 2, \dots, y$. The first element of Z-function, $z[0]$, is generally not well defined. In this article we will assume it is zero (although it doesn't change anything in the algorithm implementation). Following is the naive implementation with $O(n^2)$ implementation:

```
vector<int> z_function_trivial(string s) {
    int n = (int) s.length();
    vector<int> z(n);
    for (int i = 1; i < n; ++i)
        while (i + z[i] < n && s[z[i]] == s[i + z[i]])
            ++z[i];
    return z;
}
```

To obtain an efficient algorithm we will compute the values of $z[i]$ in turn from $i = 1$ to $n - 1$ but at the same time, when computing a new value, we'll try to make the best use possible of the previously computed values. For the sake of brevity, let's call segment matches those substrings that coincide with a prefix of s . For example, the value of the desired Z-function $z[i]$ is the length of the segment match starting at position i (and that ends at position $i + z[i] - 1$). To do this, we will keep the $[l, r]$ indices of the rightmost segment match. That is, among all detected segments we will keep the one that ends rightmost. In a way, the index r can be seen as the "boundary" to which our string s has been scanned by the algorithm; everything beyond that point is not yet known. Then, if the current index (for which we have to compute the next value of the Z-function) is i , we have one of two options:

1. $i > r$ – the current position is outside of what we have already processed.
We will then compute $z[i]$ with the trivial algorithm (that is, just comparing values one by one). Note that in the end, if $z[i] > 0$, we'll have to update the indices of the rightmost segment, because it's guaranteed that the new $r = i + z[i] - 1$ is better than the previous r .
2. $i \leq r$ – the current position is inside the current segment match $[l, r]$. Then we can use the already calculated Z-values to "initialize" the value of $z[i]$ to something (it sure is better than "starting from zero"), maybe even some big number. For this, we observe that the substrings $s[l \dots r]$ and $s[0 \dots r - l]$ match. This means that as an initial approximation for $z[i]$ we can take the value already computed for the corresponding segment $s[0 \dots r - l]$, and that is $z[i - l]$.
However, the value $z[i - l]$ could be too large: when applied to position i it could exceed the index r . This is not allowed because we know nothing about the characters to the right of r : they may differ from those required.

$$z_0[i] = \min(r - i + 1, z[i - l]) \quad (9.67)$$

After having $z[i]$ initialized to $z_0[i]$, we try to increment $z[i]$ by running the trivial algorithm – because in general, after the border r , we cannot know if the segment will continue to match or not.

Thus, the whole algorithm is split in two cases, which differ only in the initial value of $z[i]$: in the first case it's assumed to be zero, in the second case it is determined by the previously computed values (using the above formula). After that, both branches of this algorithm can be reduced to the implementation of the trivial algorithm, which starts immediately after we specify the initial value. The algorithm turns out to be very simple. Despite the fact that on each iteration the trivial algorithm is run, we have made significant progress, having an algorithm that runs in linear time. The whole solution is given as a function which returns an array of length n – the Z-function of s . Array z is initially filled with zeros. The current rightmost match segment is assumed to be $[0; 0]$ (that is, a deliberately small segment which doesn't contain any i). Inside the loop for $i = 1 \dots n - 1$ we first determine the initial value $z[i]$ – it will either remain zero or be computed using the above formula. Thereafter, the trivial algorithm attempts to increase the value of $z[i]$ as much as possible. In the end, if it's required (that is, if $i + z[i] - 1 > r$), we update the rightmost match segment $[l, r]$.

9.23.4 Knuth-Morris-Pratt Algorithm

Given a string s of length n , the prefix function for this string is defined as an array A of length n , where $A[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $A[0] = 0$. Mathematically the definition of the prefix function can be written as follows:

$$A[i] = \max_{k=0 \dots i} k : s[0 \dots k - 1] = s[i - (k - 1) \dots i] \quad (9.68)$$

Trivial Algorithm:

```
vector<int> prefix_function(string s)
{
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 0; i < n; i++)
        for (int k = 0; k <= i; k++)
            if (s.substr(0, k) == s.substr(i-k+1, i))
                pi[i] = k;
    return pi;
}
```

IIITH:

It has a time complexity of $O(n^3)$.

Efficient Algorithm: Prefix function was used as the main function of a substring search algorithm.

The first important observation is, that the values of the prefix function can only increase by at most one.

Thus when moving to the next position, the value of the prefix function can either increase by one, stay the same, or decrease by some amount. This fact already allows us to reduce the complexity of the algorithm to $O(n^2)$, because in one step the prefix function can grow at most by one. In total the function can grow at most n steps, and therefore also only can decrease a total of n steps. This means we only have to perform $O(n)$ string comparisons, and reach the complexity $O(n^2)$.

To get rid of the string comparisons, we have to use all the information computed in the previous steps. So let us compute the value of the prefix function A for $i + 1$. If $s[i + 1] = s[A[i]]$, then we can say with certainty that $A[i + 1] = A[i] + 1$, since we already know that the suffix at position i of length $A[i]$ is equal to the prefix of length $A[i]$.

If this is not the case, $s[i + 1] \neq s[A[i]]$, then we need to try a shorter string. In order to speed things up, we would like to immediately move to the longest length $j < A[i]$, such that the prefix property in the position i holds, i.e. $s[0 \dots j - 1] = s[i - j + 1 \dots i]$.

Indeed, if we find such a length j , then we again only need to compare the characters $s[i + 1]$ and $s[j]$. If they are equal, then we can assign $A[i + 1] = j + 1$. Otherwise we will need to find the largest value smaller than j , for which the prefix property holds, and so on. It can happen that this goes until $j = 0$. If then $s[i + 1] = s[0]$, we assign $A[i + 1] = 1$, and $A[i + 1] = 0$ otherwise.

for the current length j at the position i for which the prefix property holds, i.e. $s[0 \dots j - 1] = s[i - j + 1 \dots i]$, we want to find the greatest $k < j$, for which the prefix property holds.

This has to be the value of $A[j - 1]$.

Outline:

1. We compute the prefix values $A[i]$ in a loop by iterating from $i = 1$ to $i = n - 1$ ($A[0]$ just gets assigned with 0).
2. To calculate the current value $A[i]$ we set the variable j denoting the length of the best suffix for $i - 1$. Initially $j = A[i - 1]$.
3. Test if the suffix of length $j + 1$ is also a prefix by comparing $s[j]$ and $s[i]$. If they are equal then we assign $A[i] = j + 1$, otherwise we reduce j to $A[j - 1]$ and repeat this step.
4. If we have reached the length $j = 0$ and still don't have a match, then we assign $A[i] = 0$ and go to the next index $i + 1$.

9.24 Geometrical Algorithms:

9.24.1 Circle

Formula is given By

$$x^2 + y^2 = r^2 \tag{9.69}$$

9.24.2 Triangle's Medians

Any triangle's area T can be expressed in terms of its medians m_a, m_b, m_c as follows. Denoting their semi-sum $(m_a + m_b + m_c)/2$ as s , we have