International Institute Of Information Technology - Hyderabad

CP Notebook

AAD Project, 2021

Sanyam Shah

2020101012

Contest (1)

template.cpp (63 lines)

```cpp
#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 ABSS(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<<ln
#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)
```

```cpp
#define mod 1000000007

const ll inf = 1e9;
const ll llinf = 2e18;
ll MULL(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 $\leq$ 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: $\mathcal{O}\left(\sqrt{m}\right)$

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.

modsum(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 $a \cdot b \bmod c$ (or $a^b \bmod c$) 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

bbbd8f, 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: $\mathcal{O}\left(\log^2 p\right)$ 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.

8285a0, 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 $a^b \mod c$.

"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.

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() {
  RP(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

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

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

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

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

5581ec, 74 lines

```
#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.

737711, 35 lines

```
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--;
   }
   return v;
}
```

```
//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: $\mathcal{O}(V^2)$ with adjacency matrix and $\mathcal{O}(ElogV)$ with adjacency list.

8fb539, 47 lines

```
#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: $\mathcal{O}(ElogV)$ or $\mathcal{O}(ElogE)$

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: $\mathcal{O}(ElogV)$

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 dist = inf; nodes reachable through negative-weight cycles get dist = -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 $m[i][j] =$ `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 `-inf` if the path goes through a negative-weight cycle.

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 $O(VE^2)$. 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: $\mathcal{O}(V+E)$ 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: $\mathcal{O}(V)$ 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: $\mathcal{O}(E+V)$

f31f40, 52 lines

```
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: $\mathcal{O}(E+V)$

b5c48b, 26 lines

```
// 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,... to a 2-SAT problem, so that an expression of the type $(a||!b)\&\&(!a||c)\&\&(d||!b)\&\&\ldots$ becomes true, or reports that it is unsatisfiable. Negated variables are represented by bit-inversions (~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.

a5d58f, 46 lines

```
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: $\mathcal{O}(E+V)$

dbcf44, 43 lines

```
// 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

n	1	2	3	4	5	6	7	8	9	10
$n!$	1	2	6	24	120	720	5040	40320	362880	3628800

n	11	12	13	14	15	16	17
$n!$	4.0e7	4.8e8	6.2e9	8.7e10	1.3e12	2.1e13	3.6e14

n	20	25	30	40	50	100	150	171
$n!$	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.

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

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.

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

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

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	∼2e5	∼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 $n = n_k p^k + ... + n_1 p + n_0$ and $m = m_k p^k + ... + m_1 p + m_0$. Then $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \pmod p$.

4.2.3 Binomials

multinomial.h

Description: Computes $\binom{k_1 + \cdots + k_n}{k_1, k_2, \ldots, k_n} = \frac{(\sum k_i)!}{k_1! k_2! \ldots k_n!}$. 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 first kind

Number of permutations on n items with k cycles.

$$c(n,k) = c(n-1,k-1) + (n-1)c(n-1,k),\ c(0,0) = 1$$

$$\textstyle\sum_{k=0}^{n} c(n,k)x^k = x(x+1)\ldots(x+n-1)$$

$c(8,k) = 8, 0, 5040, 13068, 13132, 6769, 1960, 322, 28, 1$
$c(n,2) = 0, 0, 1, 3, 11, 50, 274, 1764, 13068, 109584, \ldots$

4.3.2 Stirling numbers of the second kind

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

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

4.3.3 Catalan numbers

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

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

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

- sub-diagonal monotone paths in an $n \times n$ grid.
- strings with n pairs of parenthesis, correctly nested.
- binary trees with with $n+1$ leaves (0 or 2 children).
- ordered trees with $n+1$ vertices.
- ways a convex polygon with $n+2$ sides can be cut into triangles by connecting vertices with straight lines.
- permutations of $[n]$ with no 3-term increasing subseq.

Numerical (5)

5.1 Polynomials and recurrences

Polynomial.h

```
struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i = SZ(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    REP(i,1,SZ(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i=SZ(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b, b=c;
    a.pop_back();
  }
};
```

PolyRoots.h

Description: Finds the real roots to a polynomial.
Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0
Time: $\mathcal{O}\left(n^2 \log(1/\epsilon)\right)$
"Polynomial.h" 7118fe, 23 lines

```
vector<double> polyRoots(Poly p, double xmin, double xmax) {
  if (SZ(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = polyRoots(der, xmin, xmax);
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(ALL(dr));
  REP(i,0,SZ(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      REP(it,0,60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
```

PolyInterpolate.h

Description: Given n points (x[i], y[i]), computes an n-1-degree polynomial p that passes through them: $p(x) = a[0] * x^0 + ... + a[n-1] * x^{n-1}$. For numerical precision, pick $x[k] = c * \cos(k/(n-1) * \pi), k = 0 \ldots n-1$.
Time: $\mathcal{O}\left(n^2\right)$ a4f803, 13 lines

```
typedef vector<double> vd;
vd interpolate(vd x, vd y, int n) {
  vd res(n), temp(n);
  REP(k,0,n-1) REP(i,k+1,n)
    y[i] = (y[i] - y[k]) / (x[i] - x[k]);
  double last = 0; temp[0] = 1;
  REP(k,0,n) REP(i,0,n) {
```

```
    res[i] += y[k] * temp[i];
    swap(last, temp[i]);
    temp[i] -= last * x[k];
  }
  return res;
}
```

BerlekampMassey.h

Description: Recovers any n-order linear recurrence relation from the first $2n$ terms of the recurrence. Useful for guessing linear recurrences after brute-forcing the first terms. Should work on any field, but numerical stability for floats is not guaranteed. Output will have size $\leq n$.
Usage: berlekampMassey({0, 1, 1, 3, 5, 11}) // {1, 2}
Time: $\mathcal{O}(N^2)$

"../number-theory/ModPow.h" 6120d6, 20 lines

```
vector<ll> berlekampMassey(vector<ll> s) {
  int n = SZ(s), L = 0, m = 0;
  vector<ll> C(n), B(n), T;
  C[0] = B[0] = 1;

  ll b = 1;
  REP(i,0,n) { ++m;
    ll d = s[i] % mod;
    REP(j,1,L+1) d = (d + C[j] * s[i - j]) % mod;
    if (!d) continue;
    T = C; ll coef = d * modpow(b, mod-2) % mod;
    REP(j,m,n) C[j] = (C[j] - coef * B[j - m]) % mod;
    if (2 * L > i) continue;
    L = i + 1 - L; B = T; b = d; m = 0;
  }

  C.resize(L + 1); C.erase(C.begin());
  for (ll& x : C) x = (mod - x) % mod;
  return C;
}
```

LinearRecurrence.h

Description: Generates the k'th term of an n-order linear recurrence $S[i] = \sum_j S[i-j-1]tr[j]$, given $S[0 \dots \geq n-1]$ and $tr[0 \dots n-1]$. Faster than matrix multiplication. Useful together with Berlekamp–Massey.
Usage: linearRec({0, 1}, {1, 1}, k) // k'th Fibonacci number
Time: $\mathcal{O}(n^2 \log k)$

f72958, 26 lines

```
typedef vector<ll> Poly;
ll linearRec(Poly S, Poly tr, ll k) {
  int n = SZ(tr);

  auto combine = [&](Poly a, Poly b) {
    Poly res(n * 2 + 1);
    REP(i,0,n+1) REP(j,0,n+1)
      res[i + j] = (res[i + j] + a[i] * b[j]) % mod;
    for (int i = 2 * n; i > n; --i) REP(j,0,n)
      res[i - 1 - j] = (res[i - 1 - j] + res[i] * tr[j]) % mod;
    res.resize(n + 1);
    return res;
  };

  Poly pol(n + 1), e(pol);
  pol[0] = e[1] = 1;

  for (++k; k; k /= 2) {
    if (k % 2) pol = combine(pol, e);
    e = combine(e, e);
  }

  ll res = 0;
  REP(i,0,n) res = (res + pol[i + 1] * S[i]) % mod;
  return res;
}
```

5.2 Optimization

GoldenSectionSearch.h

Description: Finds the argument minimizing the function f in the interval $[a, b]$ assuming f is unimodal on the interval, i.e. has only one local minimum. The maximum error in the result is eps. Works equally well for maximization with a small change in the code. See TernarySearch.h in the Various chapter for a discrete version.
Usage: double func(double x) { return 4+x+.3*x*x; }
double xmin = gss(-1000,1000,func);
Time: $\mathcal{O}(\log((b-a)/\epsilon))$

31d45b, 14 lines

```
double gss(double a, double b, double (*f)(double)) {
  double r = (sqrt(5)-1)/2, eps = 1e-7;
  double x1 = b - r*(b-a), x2 = a + r*(b-a);
  double f1 = f(x1), f2 = f(x2);
  while (b-a > eps)
    if (f1 < f2) { //change to > to find maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r*(b-a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r*(b-a); f2 = f(x2);
    }
  return a;
}
```

HillClimbing.h

Description: Poor man's optimization for unimodal functions.

fff601, 14 lines

```
typedef array<double, 2> P;

template<class F> pair<double, P> hillClimb(P start, F f) {
  pair<double, P> cur(f(start), start);
  for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    REP(j,0,100) REP(dx,-1,2) REP(dy,-1,2) {
      P p = cur.second;
      p[0] += dx*jmp;
      p[1] += dy*jmp;
      cur = min(cur, make_pair(f(p), p));
    }
  }
  return cur;
}
```

Integrate.h

Description: Simple integration of a function over an interval using Simpson's rule. The error should be proportional to h^4, although in practice you will want to verify that the result is stable to desired precision when epsilon changes.

f6be40, 7 lines

```
template<class F>
double quad(double a, double b, F f, const int n = 1000) {
  double h = (b - a) / 2 / n, v = f(a) + f(b);
  REP(i,1,n*2)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
}
```

IntegrateAdaptive.h

Description: Fast integration using an adaptive Simpson's rule.
Usage: double sphereVolume = quad(-1, 1, [](double x) {
return quad(-1, 1, [&](double y) {
return quad(-1, 1, [&](double z) {
return x*x + y*y + z*z < 1; });});});

92dd79, 15 lines

```
typedef double d;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (b-a) / 6

template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

Simplex.h

Description: Solves a general linear maximization problem: maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$. Returns -inf if there is no solution, inf if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
Usage: vvd A = {{1,-1}, {-1,1}, {-1,-2}};
vd b = {1,1,-4}, c = {-1,-1}, x;
T val = LPSolver(A, b, c).solve(x);
Time: $\mathcal{O}(NM * \#pivots)$, where a pivot may be e.g. an edge relaxation. $\mathcal{O}(2^n)$ in the general case.

dffe7c, 68 lines

```
typedef double T; // long double, Rational, double + mod<P>...
typedef vector<T> vd;
typedef vector<vd> vvd;

const T eps = 1e-8, inf = 1/.0;
#define MP make_pair
#define ltj(X) if(s == -1 || MP(X[j],N[j]) < MP(X[s],N[s])) s=j

struct LPSolver {
  int m, n;
  VI N, B;
  vvd D;

  LPSolver(const vvd& A, const vd& b, const vd& c) :
    m(SZ(b)), n(SZ(c)), N(n+1), B(m), D(m+2, vd(n+2)) {
      REP(i,0,m) REP(j,0,n) D[i][j] = A[i][j];
      REP(i,0,m) { B[i] = n+i; D[i][n] = -1; D[i][n+1] = b[i];}
      REP(j,0,n) { N[j] = j; D[m][j] = -c[j]; }
      N[n] = -1; D[m+1][n] = 1;
    }

  void pivot(int r, int s) {
    T *a = D[r].data(), inv = 1 / a[s];
    REP(i,0,m+2) if (i != r && abs(D[i][s]) > eps) {
      T *b = D[i].data(), inv2 = b[s] * inv;
      REP(j,0,n+2) b[j] -= a[j] * inv2;
      b[s] = a[s] * inv2;
    }
    REP(j,0,n+2) if (j != s) D[r][j] *= inv;
    REP(i,0,m+2) if (i != r) D[i][s] *= -inv;
    D[r][s] = inv;
    swap(B[r], N[s]);
  }

  bool simplex(int phase) {
    int x = m + phase - 1;
    for (;;) {
      int s = -1;
      REP(j,0,n+1) if (N[j] != -phase) ltj(D[x]);
      if (D[x][s] >= -eps) return true;
      int r = -1;
      REP(i,0,m) {
```

```
        if (D[i][s] <= eps) continue;
        if (r == -1 || MP(D[i][n+1] / D[i][s], B[i])
                     < MP(D[r][n+1] / D[r][s], B[r])) r = i;
      }
      if (r == -1) return false;
      pivot(r, s);
    }
  }

  T solve(vd &x) {
    int r = 0;
    REP(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] < -eps) {
      pivot(r, n);
      if (!simplex(2) || D[m+1][n+1] < -eps) return -inf;
      REP(i,0,m) if (B[i] == -1) {
        int s = 0;
        REP(j,1,n+1) ltj(D[i]);
        pivot(i, s);
      }
    }
    bool ok = simplex(1); x = vd(n);
    REP(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
    return ok ? D[m][n+1] : inf;
  }
};
```

5.3 Matrices

Determinant.h

Description: Calculates determinant of a matrix. Destroys the matrix.
Time: $\mathcal{O}\left(N^3\right)$

d14002, 15 lines

```
double det(vector<vector<double>>& a) {
  int n = SZ(a); double res = 1;
  REP(i,0,n) {
    int b = i;
    REP(j,i+1,n) if (fabs(a[j][i]) > fabs(a[b][i])) b = j;
    if (i != b) swap(a[i], a[b]), res *= -1;
    res *= a[i][i];
    if (res == 0) return 0;
    REP(j,i+1,n) {
      double v = a[j][i] / a[i][i];
      if (v != 0) REP(k,i+1,n) a[j][k] -= v * a[i][k];
    }
  }
  return res;
}
```

IntDeterminant.h

Description: Calculates determinant using modular arithmetics. Modulos can also be removed to get a pure-integer version.
Time: $\mathcal{O}\left(N^3\right)$

669167, 18 lines

```
const ll mod = 12345;
ll det(vector<vector<ll>>& a) {
  int n = SZ(a); ll ans = 1;
  REP(i,0,n) {
    REP(j,i+1,n) {
      while (a[j][i] != 0) { // gcd step
        ll t = a[i][i] / a[j][i];
        if (t) REP(k,i,n)
          a[i][k] = (a[i][k] - a[j][k] * t) % mod;
        swap(a[i], a[j]);
        ans *= -1;
      }
    }
    ans = ans * a[i][i] % mod;
    if (!ans) return 0;
  }
  return (ans + mod) % mod;
}
```

SolveLinear.h

Description: Solves $A * x = b$. If there are multiple solutions, an arbitrary one is returned. Returns rank, or -1 if no solutions. Data in A and b is lost.
Time: $\mathcal{O}\left(n^2m\right)$

d6dca7, 38 lines

```
typedef vector<double> vd;
const double eps = 1e-12;

int solveLinear(vector<vd>& A, vd& b, vd& x) {
  int n = SZ(A), m = SZ(x), rank = 0, br, bc;
  if (n) assert(SZ(A[0]) == m);
  VI col(m); iota(ALL(col), 0);

  REP(i,0,n) {
    double v, bv = 0;
    REP(r,i,n) REP(c,i,m)
      if ((v = fabs(A[r][c])) > bv)
        br = r, bc = c, bv = v;
    if (bv <= eps) {
      REP(j,i,n) if (fabs(b[j]) > eps) return -1;
      break;
    }
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    REP(j,0,n) swap(A[j][i], A[j][bc]);
    bv = 1/A[i][i];
    REP(j,i+1,n) {
      double fac = A[j][i] * bv;
      b[j] -= fac * b[i];
      REP(k,i+1,m) A[j][k] -= fac*A[i][k];
    }
    rank++;
  }

  x.assign(m, 0);
  for (int i = rank; i--;) {
    b[i] /= A[i][i];
    x[col[i]] = b[i];
    REP(j,0,i) b[j] -= A[j][i] * b[i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

SolveLinear2.h

Description: To get all uniquely determined values of x back from SolveLinear, make the following changes:

"SolveLinear.h" 5cad07, 7 lines

```
REP(j,0,n) if (j != i) // instead of REP(j,i+1,n)
// ... then at the end:
x.assign(m, undefined);
REP(i,0,rank) {
  REP(j,rank,m) if (fabs(A[i][j]) > eps) goto fail;
  x[col[i]] = b[i] / A[i][i];
fail:; }
```

SolveLinearBinary.h

Description: Solves $Ax = b$ over $\mathbb{F}_2$. If there are multiple solutions, one is returned arbitrarily. Returns rank, or -1 if no solutions. Destroys A and b.
Time: $\mathcal{O}\left(n^2m\right)$

c9c00b, 34 lines

```
typedef bitset<1000> bs;

int solveLinear(vector<bs>& A, VI& b, bs& x, int m) {
  int n = SZ(A), rank = 0, br;
  assert(m <= SZ(x));
  VI col(m); iota(ALL(col), 0);
  REP(i,0,n) {
    for (br=i; br<n; ++br) if (A[br].any()) break;
    if (br == n) {
      REP(j,i,n) if(b[j]) return -1;
      break;
    }
    int bc = (int)A[br]._Find_next(i-1);
    swap(A[i], A[br]);
    swap(b[i], b[br]);
    swap(col[i], col[bc]);
    REP(j,0,n) if (A[j][i] != A[j][bc]) {
      A[j].flip(i); A[j].flip(bc);
    }
    REP(j,i+1,n) if (A[j][i]) {
      b[j] ^= b[i];
      A[j] ^= A[i];
    }
    rank++;
  }

  x = bs();
  for (int i = rank; i--;) {
    if (!b[i]) continue;
    x[col[i]] = 1;
    REP(j,0,i) b[j] ^= A[j][i];
  }
  return rank; // (multiple solutions if rank < m)
}
```

MatrixInverse.h

Description: Invert matrix A. Returns rank; result is stored in A unless singular (rank < n). Can easily be extended to prime moduli; for prime powers, repeatedly set $A^{-1} = A^{-1}(2I - AA^{-1}) \pmod{p^k}$ where A^{-1} starts as the inverse of A mod p, and k is doubled in each step.
Time: $\mathcal{O}\left(n^3\right)$

43464f, 35 lines

```
int matInv(vector<vector<double>>& A) {
  int n = SZ(A); VI col(n);
  vector<vector<double>> tmp(n, vector<double>(n));
  REP(i,0,n) tmp[i][i] = 1, col[i] = i;

  REP(i,0,n) {
    int r = i, c = i;
    REP(j,i,n) REP(k,i,n)
      if (fabs(A[j][k]) > fabs(A[r][c]))
        r = j, c = k;
    if (fabs(A[r][c]) < 1e-12) return i;
    A[i].swap(A[r]); tmp[i].swap(tmp[r]);
    REP(j,0,n)
      swap(A[j][i], A[j][c]), swap(tmp[j][i], tmp[j][c]);
    swap(col[i], col[c]);
    double v = A[i][i];
    REP(j,i+1,n) {
      double f = A[j][i] / v;
      A[j][i] = 0;
      REP(k,i+1,n) A[j][k] -= f*A[i][k];
      REP(k,0,n) tmp[j][k] -= f*tmp[i][k];
    }
    REP(j,i+1,n) A[i][j] /= v;
    REP(j,0,n) tmp[i][j] /= v;
    A[i][i] = 1;
  }

  for (int i = n-1; i > 0; --i) REP(j,0,i) {
    double v = A[j][i];
    REP(k,0,n) tmp[j][k] -= v*tmp[i][k];
  }
```

```
  REP(i,0,n) REP(j,0,n) A[col[i]][col[j]] = tmp[i][j];
  return n;
}
```

Tridiagonal.h

Description: $x = \text{tridiagonal}(d, p, q, b)$ solves the equation system

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_{n-1} \end{pmatrix} = \begin{pmatrix} d_0 & p_0 & 0 & 0 & \cdots & 0 \\ q_0 & d_1 & p_1 & 0 & \cdots & 0 \\ 0 & q_1 & d_2 & p_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & q_{n-3} & d_{n-2} & p_{n-2} \\ 0 & 0 & \cdots & 0 & q_{n-2} & d_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$

This is useful for solving problems on the type

$$a_i = b_i a_{i-1} + c_i a_{i+1} + d_i,\ 1 \le i \le n,$$

where a_0, a_{n+1}, b_i, c_i and d_i are known. a can then be obtained from

$$\{a_i\} = \text{tridiagonal}(\{1, -1, -1, ..., -1, 1\}, \{0, c_1, c_2, \ldots, c_n\}, \{b_1, b_2, \ldots, b_n, 0\}, \{a_0, d_1, d_2, \ldots, d_n, a_{n+1}\}).$$

Fails if the solution is not unique.
If $|d_i| > |p_i| + |q_{i-1}|$ for all i, or $|d_i| > |p_{i-1}| + |q_i|$, or the matrix is positive definite, the algorithm is numerically stable and neither `tr` nor the check for `diag[i] == 0` is needed.
Time: $\mathcal{O}(N)$

a74eda, 26 lines

```
typedef double T;
vector<T> tridiagonal(vector<T> diag, const vector<T>& super,
    const vector<T>& sub, vector<T> b) {
  int n = SZ(b); VI tr(n);
  REP(i,0,n-1) {
    if (abs(diag[i]) < 1e-9 * abs(super[i])) { // diag[i] == 0
      b[i+1] -= b[i] * diag[i+1] / super[i];
      if (i+2 < n) b[i+2] -= b[i] * sub[i+1] / super[i];
      diag[i+1] = sub[i]; tr[++i] = 1;
    } else {
      diag[i+1] -= super[i]*sub[i]/diag[i];
      b[i+1] -= b[i]*sub[i]/diag[i];
    }
  }
  for (int i = n; i--;) {
    if (tr[i]) {
      swap(b[i], b[i-1]);
      diag[i-1] = diag[i];
      b[i] /= super[i-1];
    } else {
      b[i] /= diag[i];
      if (i) b[i-1] -= b[i]*super[i-1];
    }
  }
  return b;
}
```

5.4 Fourier transforms

FastFourierTransform.h

Description: fft(a) computes $\hat{f}(k) = \sum_x a[x] \exp(2\pi i \cdot kx/N)$ for all k. N must be a power of 2. Useful for convolution: `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For convolution of complex numbers or more than two vectors: FFT, multiply pointwise, divide by n, reverse(start+1, end), FFT back. Rounding is safe if $(\sum a_i^2 + \sum b_i^2) \log_2 N < 9 \cdot 10^{14}$ (in practice 10^{16}; higher for random inputs). Otherwise, use NTT/FFTMod.
Time: $\mathcal{O}(N \log N)$ with $N = |A| + |B|$ ($\sim 1s$ for $N = 2^{22}$)

a333b0, 35 lines

```
typedef complex<double> C;
typedef vector<double> vd;
void fft(vector<C>& a) {
  int n = SZ(a), L = 31 - __builtin_clz(n);
  static vector<complex<long double>> R(2, 1);
  static vector<C> rt(2, 1);  // (^ 10% faster if double)
  for (static int k = 2; k < n; k *= 2) {
    R.resize(n); rt.resize(n);
    auto x = polar(1.0L, acos(-1.0L) / k);
    REP(i,k,2*k) rt[i] = R[i] = i&1 ? R[i/2] * x : R[i/2];
  }
  VI rev(n);
  REP(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  REP(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) REP(j,0,k) {
      C z = rt[j+k] * a[i+j+k]; // (25% faster if hand-rolled)
      a[i + j + k] = a[i + j] - z;
      a[i + j] += z;
    }
}
vd conv(const vd& a, const vd& b) {
  if (a.empty() || b.empty()) return {};
  vd res(SZ(a) + SZ(b) - 1);
  int L = 32 - __builtin_clz(SZ(res)), n = 1 << L;
  vector<C> in(n), out(n);
  copy(ALL(a), begin(in));
  REP(i,0,SZ(b)) in[i].imag(b[i]);
  fft(in);
  for (C& x : in) x *= x;
  REP(i,0,n) out[i] = in[-i & (n - 1)] - conj(in[i]);
  fft(out);
  REP(i,0,SZ(res)) res[i] = imag(out[i]) / (4 * n);
  return res;
}
```

FastFourierTransformMod.h

Description: Higher precision FFT, can be used for convolutions modulo arbitrary integers as long as $N \log_2 N \cdot \text{mod} < 8.6 \cdot 10^{14}$ (in practice 10^{16} or higher). Inputs must be in $[0, \text{mod})$.
Time: $\mathcal{O}(N \log N)$, where $N = |A| + |B|$ (twice as slow as NTT or FFT)

"FastFourierTransform.h" 860295, 22 lines

```
typedef vector<ll> vl;
template<int M> vl convMod(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  vl res(SZ(a) + SZ(b) - 1);
  int B=32-__builtin_clz(SZ(res)), n=1<<B, cut=int(sqrt(M));
  vector<C> L(n), R(n), outs(n), outl(n);
  REP(i,0,SZ(a)) L[i] = C((int)a[i] / cut, (int)a[i] % cut);
  REP(i,0,SZ(b)) R[i] = C((int)b[i] / cut, (int)b[i] % cut);
  fft(L), fft(R);
  REP(i,0,n) {
    int j = -i & (n - 1);
    outl[j] = (L[i] + conj(L[j])) * R[i] / (2.0 * n);
    outs[j] = (L[i] - conj(L[j])) * R[i] / (2.0 * n) / 1i;
  }
  fft(outl), fft(outs);
  REP(i,0,SZ(res)) {
    ll av = ll(real(outl[i])+.5), cv = ll(imag(outs[i])+.5);
    ll bv = ll(imag(outl[i])+.5) + ll(real(outs[i])+.5);
    res[i] = ((av % M * cut + bv) % M * cut + cv) % M;
  }
  return res;
}
```

NumberTheoreticTransform.h

Description: ntt(a) computes $\hat{f}(k) = \sum_x a[x] g^{xk}$ for all k, where $g = \text{root}^{(mod-1)/N}$. N must be a power of 2. Useful for convolution modulo specific nice primes of the form $2^a b + 1$, where the convolution result has size at most 2^a. For arbitrary modulo, see FFTMod. `conv(a, b) = c`, where $c[x] = \sum a[i]b[x-i]$. For manual convolution: NTT the inputs, multiply pointwise, divide by n, reverse(start+1, end), NTT back. Inputs must be in [0, mod).
Time: $\mathcal{O}(N \log N)$

"../number-theory/ModPow.h" 14d0bb, 33 lines

```
const ll mod = (119 << 23) + 1, root = 62; // = 998244353
// For p < 2^30 there is also e.g. 5 << 25, 7 << 26, 479 << 21
// and 483 << 21 (same root). The last two are > 10^9.
typedef vector<ll> vl;
void ntt(vl &a) {
  int n = SZ(a), L = 31 - __builtin_clz(n);
  static vl rt(2, 1);
  for (static int k = 2, s = 2; k < n; k *= 2, s++) {
    rt.resize(n);
    ll z[] = {1, modpow(root, mod >> s)};
    REP(i,k,2*k) rt[i] = rt[i / 2] * z[i & 1] % mod;
  }
  VI rev(n);
  REP(i,0,n) rev[i] = (rev[i / 2] | (i & 1) << L) / 2;
  REP(i,0,n) if (i < rev[i]) swap(a[i], a[rev[i]]);
  for (int k = 1; k < n; k *= 2)
    for (int i = 0; i < n; i += 2 * k) REP(j,0,k) {
      ll z = rt[j + k] * a[i + j + k] % mod, &ai = a[i + j];
      a[i + j + k] = ai - z + (z > ai ? mod : 0);
      ai += (ai + z >= mod ? z - mod : z);
    }
}
vl conv(const vl &a, const vl &b) {
  if (a.empty() || b.empty()) return {};
  int s = SZ(a) + SZ(b) - 1, B = 32 - __builtin_clz(s), n = 1
      << B;
  int inv = modpow(n, mod - 2);
  vl L(a), R(b), out(n);
  L.resize(n), R.resize(n);
  ntt(L), ntt(R);
  REP(i,0,n) out[-i & (n - 1)] = (ll)L[i] * R[i] % mod * inv %
      mod;
  ntt(out);
  return {out.begin(), out.begin() + s};
}
```

FastSubsetTransform.h

Description: Transform to a basis with fast convolutions of the form $c[z] = \sum_{z=x \oplus y} a[x] \cdot b[y]$, where $\oplus$ is one of AND, OR, XOR. The size of a must be a power of two.
Time: $\mathcal{O}(N \log N)$

b1dc68, 16 lines

```
void FST(VI& a, bool inv) {
  for (int n = SZ(a), step = 1; step < n; step *= 2) {
    for (int i = 0; i < n; i += 2 * step) REP(j,i,i+step) {
      int &u = a[j], &v = a[j + step]; tie(u, v) =
        inv ? PII(v - u, u) : PII(v, u + v); // AND
        inv ? PII(v, u - v) : PII(u + v, u); // OR
        PII(u + v, u - v);                   // XOR
    }
  }
  if (inv) for (int& x : a) x /= SZ(a); // XOR only
}
VI conv(VI a, VI b) {
  FST(a, 0); FST(b, 0);
  REP(i,0,SZ(a)) a[i] *= b[i];
  FST(a, 1); return a;
}
```

WalshHadamard.h

Description: $C_k = \sum_{i \otimes j = k} A_i B_j$
Usage: Apply the transform, point multiply and invert
Time: $\mathcal{O}(N \log N)$

905e71, 11 lines

```
void WalshHadamard(poly &P, bool invert) {
  for (int len = 1; 2 * len <= SZ(P); len <<= 1) {
```

```
    for (int i = 0; i < SZ(P); i += 2 * len) {
      REP(j, 0, len) {
        auto u = P[i + j], v = P[i + len + j];
        P[i + j] = u + v, P[i + len + j] = u - v; // XOR
      }
    }
  }
  if (invert) for (auto &x : P) x /= SZ(P);
}
```

OnlineFFT.h

Description: Given $B_1, \dots B_m$, compute $A_i = \sum_{j=1}^{i-1} A_j * B_{i-j}$
Usage: 1-indexed, pad B[i] = 0 for i > m
Time: $\mathcal{O}\left(N \log^2 N\right)$

211338, 18 lines

```
void online(const Poly &B, CD a1, int n, Poly &A) {
  const int m = SZ(B) - 1;
  A.assign(n + 1, 0); A[1] = a1;
  auto bst = B.begin(), ast = A.begin();
  REP(i, 1, n) {
    A[i + 1] += A[i] * B[1];
    if (i + 2 <= n) A[i + 2] += A[i] * B[2];
    for (int pw = 2; i % pw == 0 && pw + 1 <= m; pw <<= 1) {
      Poly blockA(ast + i - pw, ast + i);
      Poly blockB(bst + pw + 1, bst + min(pw * 2, m) + 1);
      Poly prod = conv(blockA, blockB);
      REP(j, 0, SZ(prod)) {
        if (i + 1 + j <= n)
          A[i + 1 + j] += prod[j];
      }
    }
  }
}
```

Data structures (6)

6.1 Set and Map like

OrderStatisticTree.h

Description: A set (not multiset!) with support for finding the n'th element, and finding the index of an element. To get a map, change `null_type`.
Time: $\mathcal{O}\left(\log N\right)$

782797, 16 lines

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

template<class T>
using Tree = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;

void example() {
  Tree<int> t, t2; t.insert(8);
  auto it = t.insert(10).first;
  assert(it == t.lower_bound(9));
  assert(t.order_of_key(10) == 1);
  assert(t.order_of_key(11) == 2);
  assert(*t.find_by_order(0) == 8);
  t.join(t2); // assuming T < T2 or T > T2, merge t2 into t
}
```

HashMap.h

Description: Hash map with mostly the same API as unordered_map, but ~3x faster. Uses 1.5x memory. Initial capacity must be a power of 2 (if provided).

d77092, 7 lines

```
#include <bits/extc++.h>
// To use most bits rather than just the lowest ones:
struct chash { // large odd number for C
  const uint64_t C = ll(4e18 * acos(0)) | 71;
  ll operator()(ll x) const { return __builtin_bswap64(x*C); }
};
__gnu_pbds::gp_hash_table<ll,int,chash> h({},{},{},{},{1<<16});
```

Treap.h

Description: A short self-balancing tree. It acts as a sequential container with log-time splits/joins, and is easy to augment with additional data.
Time: $\mathcal{O}\left(\log N\right)$

9556fc, 55 lines

```
struct Node {
  Node *l = 0, *r = 0;
  int val, y, c = 1;
  Node(int val) : val(val), y(rand()) {}
  void recalc();
};

int cnt(Node* n) { return n ? n->c : 0; }
void Node::recalc() { c = cnt(l) + cnt(r) + 1; }

template<class F> void each(Node* n, F f) {
  if (n) { each(n->l, f); f(n->val); each(n->r, f); }
}

pair<Node*, Node*> split(Node* n, int k) {
  if (!n) return {};
  if (cnt(n->l) >= k) { // "n->val >= k" for lower_bound(k)
    auto pa = split(n->l, k);
    n->l = pa.second;
    n->recalc();
    return {pa.first, n};
  } else {
    auto pa = split(n->r, k - cnt(n->l) - 1); // and just "k"
    n->r = pa.first;
    n->recalc();
    return {n, pa.second};
  }
}

Node* merge(Node* l, Node* r) {
  if (!l) return r;
  if (!r) return l;
  if (l->y > r->y) {
    l->r = merge(l->r, r);
    l->recalc();
    return l;
  } else {
    r->l = merge(l, r->l);
    r->recalc();
    return r;
  }
}

Node* ins(Node* t, Node* n, int pos) {
  auto pa = split(t, pos);
  return merge(merge(pa.first, n), pa.second);
}

// Example application: move the range [l, r) to index k
void move(Node*& t, int l, int r, int k) {
  Node *a, *b, *c;
  tie(a,b) = split(t, l); tie(b,c) = split(b, r - l);
  if (k <= l) t = merge(ins(a, b, k), c);
  else t = merge(a, ins(c, b, k - r));
}
```

LineContainer.h

Description: Container where you can add lines of the form kx+m, and query maximum values at points x. Useful for dynamic programming ("convex hull trick").
Time: $\mathcal{O}\left(\log N\right)$

8ec1c7, 30 lines

```
struct Line {
  mutable ll k, m, p;
  bool operator<(const Line& o) const { return k < o.k; }
  bool operator<(ll x) const { return p < x; }
};

struct LineContainer : multiset<Line, less<>> {
  // (for doubles, use inf = 1/.0, div(a,b) = a/b)
  static const ll inf = LLONG_MAX;
  ll div(ll a, ll b) { // floored division
    return a / b - ((a ^ b) < 0 && a % b); }
  bool isect(iterator x, iterator y) {
    if (y == end()) return x->p = inf, 0;
    if (x->k == y->k) x->p = x->m > y->m ? inf : -inf;
    else x->p = div(y->m - x->m, x->k - y->k);
    return x->p >= y->p;
  }
  void add(ll k, ll m) {
    auto z = insert({k, m, 0}), y = z++, x = y;
    while (isect(y, z)) z = erase(z);
    if (x != begin() && isect(--x, y)) isect(x, y = erase(y));
    while ((y = x) != begin() && (--x)->p >= y->p)
      isect(x, erase(y));
  }
  ll query(ll x) {
    assert(!empty());
    auto l = *lower_bound(x);
    return l.k * x + l.m;
  }
};
```

UnionFindRollback.h

Description: Disjoint-set data structure with undo. If undo is not needed, skip st, time() and rollback().
Usage: `int t = uf.time(); ...; uf.rollback(t);`
Time: $\mathcal{O}\left(\log(N)\right)$

f79afc, 21 lines

```
struct RollbackUF {
  VI e; vector<PII> st;
  RollbackUF(int n) : e(n, -1) {}
  int size(int x) { return -e[find(x)]; }
  int find(int x) { return e[x] < 0 ? x : find(e[x]); }
  int time() { return SZ(st); }
  void rollback(int t) {
    for (int i = time(); i --> t;)
      e[st[i].first] = st[i].second;
    st.resize(t);
  }
  bool join(int a, int b) {
    a = find(a), b = find(b);
    if (a == b) return false;
    if (e[a] > e[b]) swap(a, b);
    st.push_back({a, e[a]});
    st.push_back({b, e[b]});
    e[a] += e[b]; e[b] = a;
    return true;
  }
};
```

6.2 Matrix

SubMatrix.h

Description: Calculate submatrix sums quickly, given upper-left and lower-right corners (half-open).

Usage: SubMatrix<int> m(matrix);
m.sum(0, 0, 2, 2); // top left 4 elements
Time: $\mathcal{O}\left(N^2 + Q\right)$

fe231e, 13 lines

```
template<class T>
struct SubMatrix {
  vector<vector<T>> p;
  SubMatrix(vector<vector<T>>& v) {
    int R = SZ(v), C = SZ(v[0]);
    p.assign(R+1, vector<T>(C+1));
    REP(r,0,R) REP(c,0,C)
      p[r+1][c+1] = v[r][c] + p[r][c+1] + p[r+1][c] - p[r][c];
  }
  T sum(int u, int l, int d, int r) {
    return p[d][r] - p[d][l] - p[u][r] + p[u][l];
  }
};
```

Matrix.h

Description: Basic operations on square matrices.
Usage: Matrix<int, 3> A;
A.d = {{{{1,2,3}}, {{4,5,6}}, {{7,8,9}}}};
vector<int> vec = {1,2,3};
vec = (A^N) * vec;

d8e33f, 26 lines

```
template<class T, int N> struct Matrix {
  typedef Matrix M;
  array<array<T, N>, N> d{};
  M operator*(const M& m) const {
    M a;
    REP(i,0,N) REP(j,0,N)
      REP(k,0,N) a.d[i][j] += d[i][k]*m.d[k][j];
    return a;
  }
  vector<T> operator*(const vector<T>& vec) const {
    vector<T> ret(N);
    REP(i,0,N) REP(j,0,N) ret[i] += d[i][j] * vec[j];
    return ret;
  }
  M operator^(ll p) const {
    assert(p >= 0);
    M a, b(*this);
    REP(i,0,N) a.d[i][i] = 1;
    while (p) {
      if (p&1) a = a*b;
      b = b*b;
      p >>= 1;
    }
    return a;
  }
};
```

6.3 Range DS

FenwickTree.h

Description: Computes partial sums a[0] + a[1] + ... + a[pos - 1], and updates single elements a[i], taking the difference between the old and new value.
Time: Both operations are $\mathcal{O}\left(\log N\right)$.

b06af0, 22 lines

```
struct FT {
  vector<ll> s;
  FT(int n) : s(n) {}
  void update(int pos, ll dif) { // a[pos] += dif
    for (; pos < SZ(s); pos |= pos + 1) s[pos] += dif;
  }
  ll query(int pos) { // sum of values in [0, pos)
    ll res = 0;
    for (; pos > 0; pos &= pos - 1) res += s[pos-1];
    return res;
  }
  int lower_bound(ll sum) {// min pos st sum of [0, pos] >= sum
    // Returns n if no sum is >= sum, or -1 if empty sum is.
    if (sum <= 0) return -1;
    int pos = 0;
    for (int pw = 1 << 25; pw; pw >>= 1) {
      if (pos + pw <= SZ(s) && s[pos + pw-1] < sum)
        pos += pw, sum -= s[pos-1];
    }
    return pos;
  }
};
```

FenwickTree2d.h

Description: Computes sums a[i,j] for all i<I, j<J, and increases single elements a[i,j]. Requires that the elements to be updated are known in advance (call fakeUpdate() before init()).
Time: $\mathcal{O}\left(\log^2 N\right)$. (Use persistent segment trees for $\mathcal{O}\left(\log N\right)$.)

"FenwickTree.h" d5c1b7, 22 lines

```
struct FT2 {
  vector<VI> ys; vector<FT> ft;
  FT2(int limx) : ys(limx) {}
  void fakeUpdate(int x, int y) {
    for (; x < SZ(ys); x |= x + 1) ys[x].push_back(y);
  }
  void init() {
    for (VI& v : ys) sort(ALL(v)), ft.emplace_back(SZ(v));
  }
  int ind(int x, int y) {
    return (int)(lower_bound(ALL(ys[x]), y) - ys[x].begin()); }
  void update(int x, int y, ll dif) {
    for (; x < SZ(ys); x |= x + 1)
      ft[x].update(ind(x, y), dif);
  }
  ll query(int x, int y) {
    ll sum = 0;
    for (; x; x &= x - 1)
      sum += ft[x-1].query(ind(x-1, y));
    return sum;
  }
};
```

RMQ.h

Description: Range Minimum Queries on an array. Returns min(V[a], V[a + 1], ... V[b - 1]) in constant time.
Usage: RMQ rmq(values);
rmq.query(inclusive, exclusive);
Time: $\mathcal{O}\left(|V| \log |V| + Q\right)$

9a1bbf, 16 lines

```
template<class T>
struct RMQ {
  vector<vector<T>> jmp;
  RMQ(const vector<T>& V) : jmp(1, V) {
    for (int pw = 1, k = 1; pw * 2 <= SZ(V); pw *= 2, ++k) {
      jmp.emplace_back(SZ(V) - pw * 2 + 1);
      REP(j,0,SZ(jmp[k]))
        jmp[k][j] = min(jmp[k - 1][j], jmp[k - 1][j + pw]);
    }
  }
  T query(int a, int b) {
    assert(a < b); // or return inf if a == b
    int dep = 31 - __builtin_clz(b - a);
    return min(jmp[dep][a], jmp[dep][b - (1 << dep)]);
  }
};
```

MoQueries.h

Description: Answer interval or tree path queries by finding an approximate TSP through the queries, and moving from one query to the next by adding/removing points at the ends. If values are on tree edges, change `step` to add/remove the edge (a, c) and remove the initial `add` call (but keep `in`).
Time: $\mathcal{O}\left(N\sqrt{Q}\right)$

3015c5, 49 lines

```
void add(int ind, int end) { ... } // add a[ind] (end = 0 or 1)
void del(int ind, int end) { ... } // remove a[ind]
int calc() { ... } // compute current answer

VI mo(vector<PII> Q) {
  int L = 0, R = 0, blk = 350; // ~N/sqrt(Q)
  VI s(SZ(Q)), res = s;
#define K(x) PII(x.first/blk, x.second ^ -(x.first/blk & 1))
  iota(ALL(s), 0);
  sort(ALL(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
  for (int qi : s) {
    PII q = Q[qi];
    while (L > q.first) add(--L, 0);
    while (R < q.second) add(R++, 1);
    while (L < q.first) del(L++, 0);
    while (R > q.second) del(--R, 1);
    res[qi] = calc();
  }
  return res;
}

VI moTree(vector<array<int, 2>> Q, vector<VI>& ed, int root=0){
  int N = SZ(ed), pos[2] = {}, blk = 350; // ~N/sqrt(Q)
  VI s(SZ(Q)), res = s, I(N), L(N), R(N), in(N), par(N);
  add(0, 0), in[0] = 1;
  auto dfs = [&](int x, int p, int dep, auto& f) -> void {
    par[x] = p;
    L[x] = N;
    if (dep) I[x] = N++;
    for (int y : ed[x]) if (y != p) f(y, x, !dep, f);
    if (!dep) I[x] = N++;
    R[x] = N;
  };
  dfs(root, -1, 0, dfs);
#define K(x) PII(I[x[0]] / blk, I[x[1]] ^ -(I[x[0]] / blk & 1))
  iota(ALL(s), 0);
  sort(ALL(s), [&](int s, int t){ return K(Q[s]) < K(Q[t]); });
  for (int qi : s) REP(end,0,2) {
    int &a = pos[end], b = Q[qi][end], i = 0;
#define step(c) { if (in[c]) { del(a, end); in[a] = 0; } \
                  else { add(c, end); in[c] = 1; } a = c; }
    while (!(L[b] <= L[a] && R[a] <= R[b]))
      I[i++] = b, b = par[b];
    while (a != b) step(par[a]);
    while (i--) step(I[i]);
    if (end) res[qi] = calc();
  }
  return res;
}
```

MoWithUpdates.h

Description: Supports point updates at position
Time: $\mathcal{O}\left(n^{5/3}\right)$ when block $= n^{2/3}$

303d07, 17 lines

```
struct Query { int l, r, id, t; }
struct Update { int pos, pre, now; };
void MoWithUpdates(vector<Query> qs, vector<Update> upd) {
  int BLK; // set block size
  sort(qs.begin(), qs.end(), [&](Query a, Query b) {
    return {a.l/BLK, a.r/BLK, a.t} < {b.l/BLK, b.r/BLK, b.t};
  });
  for (auto q : qs) {
```

```
    while (t < q.t) ++t, apply(upd[t].pos, upd[t].now);
    while (t > q.t) apply(upd[t].pos, upd[t].pre), --t;
    while (l > q.l) add(--l);
    while (l < q.l) remove(l++);
    while (r < q.r) add(++r);
    while (r > q.r) remove(r--);
    ans[q.id] = get();
  }
}
```

SegmentTreeBeats.h

Description: example below - range sum query and two range updates: $a_i \leftarrow a_i \mod x$ and $a_i \leftarrow x$

d3ae07, 13 lines

```
bool break_condition() // when can we break
// eg. (l > rr || r < ll || max_val[node] < x)
bool tag_condition(); // when can we put tag for lazy update
// eg. (l >= ll && r <= rr && max_val[node] == min_val[node])
void modify(int node, int l, int r, int ll, int rr) {
  if (break_condition()) return;
  if (tag_condition()) { puttag(node); return; }
  pushdown(node);
  int mid = (l + r) >> 1;
  modify(node * 2, l, mid, ll, rr);
  modify(node * 2 + 1, mid + 1, r, ll ,rr);
  update();
}
```

Strings (7)

7.1 String Matching

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: $\mathcal{O}(n)$

578c4b, 16 lines

```
VI pi(const string& s) {
  VI p(SZ(s));
  REP(i,1,SZ(s)) {
    int g = p[i-1];
    while (g && s[i] != s[g]) g = p[g-1];
    p[i] = g + (s[i] == s[g]);
  }
  return p;
}

VI match(const string& s, const string& pat) {
  VI p = pi(pat + '\0' + s), res;
  REP(i,SZ(p)-SZ(s),SZ(p))
    if (p[i] == SZ(pat)) res.push_back(i - 2 * SZ(pat));
  return res;
}
```

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: $\mathcal{O}(n)$

a706e2, 12 lines

```
VI Z(string S) {
  VI z(SZ(S));
  int l = -1, r = -1;
  REP(i,1,SZ(S)) {
    z[i] = i >= r ? 0 : min(r - i, z[i - l]);
    while (i + z[i] < SZ(S) && S[i + z[i]] == S[z[i]])
      z[i]++;
    if (i + z[i] > r)
      l = i, r = i + z[i];
  }
  return z;
}
```

AhoCorasick.h

Description: Aho-Corasick automaton, used for multiple pattern matching. Initialize with AhoCorasick ac(patterns); the automaton start node will be at index 0. find(word) returns for each position the index of the longest word that ends there, or -1 if none. findAll(−, word) finds all words (up to $N\sqrt{N}$ many if no duplicate patterns) that start at each position (shortest first). Duplicate patterns are allowed; empty patterns are not. To find the longest words that start at each position, reverse all input. For large alphabets, split each symbol into chunks, with sentinel bits for symbol boundaries.

Time: construction takes $\mathcal{O}(26N)$, where N = sum of length of patterns. find(x) is $\mathcal{O}(N)$, where N = length of x. findAll is $\mathcal{O}(NM)$.

f7553c, 66 lines

```
struct AhoCorasick {
  enum {alpha = 26, first = 'A'}; // change this!
  struct Node {
    // (nmatches is optional)
    int back, next[alpha], start = -1, end = -1, nmatches = 0;
    Node(int v) { memset(next, v, sizeof(next)); }
  };
  vector<Node> N;
  VI backp;
  void insert(string& s, int j) {
    assert(!s.empty());
    int n = 0;
    for (char c : s) {
      int& m = N[n].next[c - first];
      if (m == -1) { n = m = SZ(N); N.emplace_back(-1); }
      else n = m;
    }
    if (N[n].end == -1) N[n].start = j;
    backp.push_back(N[n].end);
    N[n].end = j;
    N[n].nmatches++;
  }
  AhoCorasick(vector<string>& pat) : N(1, -1) {
    REP(i,0,SZ(pat)) insert(pat[i], i);
    N[0].back = SZ(N);
    N.emplace_back(0);

    queue<int> q;
    for (q.push(0); !q.empty(); q.pop()) {
      int n = q.front(), prev = N[n].back;
      REP(i,0,alpha) {
        int &ed = N[n].next[i], y = N[prev].next[i];
        if (ed == -1) ed = y;
        else {
          N[ed].back = y;
          (N[ed].end == -1 ? N[ed].end : backp[N[ed].start])
            = N[y].end;
          N[ed].nmatches += N[y].nmatches;
          q.push(ed);
        }
      }
    }
  }
  VI find(string word) {
    int n = 0;
    VI res; // ll count = 0;
    for (char c : word) {
      n = N[n].next[c - first];
      res.push_back(N[n].end);
      // count += N[n].nmatches;
    }
    return res;
  }
  vector<VI> findAll(vector<string>& pat, string word) {
    VI r = find(word);
    vector<VI> res(SZ(word));
    REP(i,0,SZ(word)) {
      int ind = r[i];
      while (ind != -1) {
        res[i - SZ(pat[ind]) + 1].push_back(ind);
        ind = backp[ind];
      }
    }
    return res;
  }
};
```

Hashing.h

Description: Self-explanatory methods for string hashing.

6bb778, 44 lines

```
// Arithmetic mod 2^64-1. 2x slower than mod 2^64 and more
// code, but works on evil test data (e.g. Thue-Morse, where
// ABBA... and BAAB... of length 2^10 hash the same mod 2^64).
// "typedef ull H;" instead if you think test data is random,
// or work mod 10^9+7 if the Birthday paradox is not a problem.
struct H {
  typedef uint64_t ull;
  ull x; H(ull x=0) : x(x) {}
#define OP(O,A,B) H operator O(H o) { ull r = x; asm \
  (A "addq %%rdx, %0\n adcq $0,%0" : "+a"(r) : B); return r; }
  OP(+,,"d"(o.x)) OP(*,"mul %1\n", "r"(o.x) : "rdx")
  H operator-(H o) { return *this + ~o.x; }
  ull get() const { return x + !~x; }
  bool operator==(H o) const { return get() == o.get(); }
  bool operator<(H o) const { return get() < o.get(); }
};
static const H C = (ll)1e11+3; // (order ~ 3e9; random also ok)

struct HashInterval {
  vector<H> ha, pw;
  HashInterval(string& str) : ha(SZ(str)+1), pw(ha) {
    pw[0] = 1;
    REP(i,0,SZ(str))
      ha[i+1] = ha[i] * C + str[i],
      pw[i+1] = pw[i] * C;
  }
  H hashInterval(int a, int b) { // hash [a, b)
    return ha[b] - ha[a] * pw[b - a];
  }
};

vector<H> getHashes(string& str, int length) {
  if (SZ(str) < length) return {};
  H h = 0, pw = 1;
  REP(i,0,length)
    h = h * C + str[i], pw = pw * C;
  vector<H> ret = {h};
  REP(i,length,SZ(str)) {
    ret.push_back(h = h * C + str[i] - pw * str[i-length]);
  }
  return ret;
}

H hashString(string& s){H h{}; for(char c:s) h=h*C+c;return h;}
```

MinRotation.h

Description: Finds the lexicographically smallest rotation of a string.

Usage: rotate(v.begin(), v.begin()+minRotation(v), v.end());

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

43e521, 8 lines

```
int minRotation(string s) {
```

```
  int a=0, N=SZ(s); s += s;
  REP(b,0,N) REP(k,0,N) {
    if (a+k == b || s[a+k] < s[b+k]) {b += max(0, k-1); break;}
    if (s[a+k] > s[b+k]) { a = b; break; }
  }
  return a;
}
```

7.2 Palindromes

Manacher.h

Description: For each position in a string, computes p[0][i] = half length of longest even palindrome around pos i, p[1][i] = longest odd (half rounded down).
Time: $\mathcal{O}(N)$

1deebd, 13 lines

```
array<VI, 2> manacher(const string& s) {
  int n = SZ(s);
  array<VI,2> p = {VI(n+1), VI(n)};
  REP(z,0,2) for (int i=0,l=0,r=0; i < n; i++) {
    int t = r-i+!z;
    if (i<r) p[z][i] = min(t, p[z][l+t]);
    int L = i-p[z][i], R = i+p[z][i]-!z;
    while (L>=1 && R+1<n && s[L-1] == s[R+1])
      p[z][i]++, L--, R++;
    if (R>r) l=L, r=R;
  }
  return p;
}
```

PalindromicTree.h

Description: unknown
Time: unknown

c514c5, 47 lines

```
const int SIGMA = 256;
struct node{ // use suff_link, smart_link after copying
  int suf_lnk, len, next[SIGMA], smt_lnk[SIGMA];
  node() : suf_lnk(0), len(0) { SET(next, -1), SET(smt_lnk, 0);
       }
};
struct eertree {
  int rto, rte, n, last;
  VI s; vector<node> tree;
  eertree() : rto(0), rte(1), n(0), last(1), s(1, -1), tree(2)
       {
    tree[rto].suf_lnk = tree[rte].suf_lnk = rto;
    tree[rto].len = -1;
    tree[rte].len = 0;
  }
  int add(int c) {
    s.PB(c);
    n++;
    if (s[n - tree[last].len - 1] != c) {
      last = tree[last].smt_lnk[c];
    }
    int flag = (tree[last].next[c] == -1);
    if (flag) {
      int idx = SZ(tree);
      tree.PB(node());
      tree[idx].len = tree[last].len + 2;
      if (tree[idx].len == 1) {
        tree[idx].suf_lnk = rte;
      } else {
        tree[idx].suf_lnk = tree[tree[last].smt_lnk[c]].next[c
            ];
      }
      if (tree[idx].len == 1) {
        REP(cc, 0, SIGMA) { tree[idx].smt_lnk[cc] = 0; }
        tree[idx].smt_lnk[c] = 1;
      } else {
        REP(cc, 0, SIGMA) {
          int x = tree[idx].suf_lnk;
          if (s[n - tree[x].len] == cc)
            tree[idx].smt_lnk[cc] = x;
          else
            tree[idx].smt_lnk[cc] = tree[x].smt_lnk[cc];
        }
      }
      tree[last].next[c] = idx;
    }
    last = tree[last].next[c];
    return tree[last].len;
  }
};
```

7.3 Suffix DS

SuffixArray.h

Description: Builds suffix array for a string. `sa[i]` is the starting index of the suffix which is i'th in the sorted suffix array. The returned vector is of size $n+1$, and `sa[0] = n`. The `lcp` array contains longest common prefixes for neighbouring strings in the suffix array: `lcp[i] = lcp(sa[i], sa[i-1])`, `lcp[0] = 0`. The input string must not contain any zero bytes.
Time: $\mathcal{O}(n \log n)$

3e14f8, 23 lines

```
struct SuffixArray {
  VI sa, lcp;
  SuffixArray(string& s, int lim=256) { // or basic_string<int>
    int n = SZ(s) + 1, k = 0, a, b;
    VI x(ALL(s)+1), y(n), ws(max(n, lim)), rank(n);
    sa = lcp = y, iota(ALL(sa), 0);
    for (int j = 0, p = 0; p < n; j = max(1, j * 2), lim = p) {
      p = j, iota(ALL(y), n - j);
      REP(i,0,n) if (sa[i] >= j) y[p++] = sa[i] - j;
      fill(ALL(ws), 0);
      REP(i,0,n) ws[x[i]]++;
      REP(i,1,lim) ws[i] += ws[i - 1];
      for (int i = n; i--;) sa[--ws[x[y[i]]]] = y[i];
      swap(x, y), p = 1, x[sa[0]] = 0;
      REP(i,1,n) a = sa[i - 1], b = sa[i], x[b] =
        (y[a] == y[b] && y[a + j] == y[b + j]) ? p - 1 : p++;
    }
    REP(i,1,n) rank[sa[i]] = i;
    for (int i = 0, j; i < n - 1; lcp[rank[i++]] = k)
      for (k && k--, j = sa[rank[i] - 1];
          s[i + k] == s[j + k]; k++);
  }
};
```

SuffixAutomaton.h

Description: Each path in the automaton is a substring (if it ends in a terminal node, it is a suffix) And no. of occurences = no. of ways to reach a terminal node. Or keep reverse edges of suffix links(all prefixes for that substring), then no. of ways to reach a root.
Time: $\mathcal{O}(len)$ map accesses, map can be at most of size alphabet, can also use unordered_map

37fe84, 34 lines

```
struct SuffixAutomaton {
  vector<map<char, int>> edges;
  VI link, length; // length[i]: longest string in i-th class
  int last;        // index of equivalence class of whole
       string
  SuffixAutomaton(string s) : edges{}, link{-1}, length{0},
       last(0) {
    edges.emplace_back();
    REP(i, 0, SZ(s)) {
      edges.emplace_back();
      length.push_back(i + 1);
      link.push_back(0);
      int r = SZ(edges) - 1, p = last;
      while (p >= 0 && edges[p].find(s[i]) == edges[p].end()) {
        edges[p][s[i]] = r, p = link[p];
      }
      if (p != -1) {
        const int q = edges[p][s[i]];
        if (length[p] + 1 == length[q]) link[r] = q;
        else {
          edges.push_back(edges[q]);
          length.push_back(length[p] + 1);
          link.push_back(link[q]);
          const int qq = SZ(edges) - 1;
          link[q] = link[r] = qq;
          for (; p >= 0 && edges[p][s[i]] == q; p = link[p])
            edges[p][s[i]] = qq;
        }
      }
      last = r;
    }
    VI terminals;
    for (int p = last; p > 0; p = link[p])
      terminals.push_back(p);
  }
};
```

SuffixTree.h

Description: Ukkonen's algorithm for online suffix tree construction. Each node contains indices [l, r) into the string, and a list of child nodes. Suffixes are given by traversals of this tree, joining [l, r) substrings. The root is 0 (has l = -1, r = 0), non-existent children are -1. To get a complete tree, append a dummy symbol – otherwise it may contain an incomplete path (still useful for substring matching, though).
Time: $\mathcal{O}(26N)$

5e69b7, 50 lines

```
struct SuffixTree {
  enum { N = 200010, ALPHA = 26 }; // N ~ 2*maxlen+10
  int toi(char c) { return c - 'a'; }
  string a; // v = cur node, q = cur position
  int t[N][ALPHA],l[N],r[N],p[N],s[N],v=0,q=0,m=2;

  void ukkadd(int i, int c) { suff:
    if (r[v]<=q) {
      if (t[v][c]==-1) { t[v][c]=m;  l[m]=i;
        p[m++]=v; v=s[v]; q=r[v];  goto suff; }
      v=t[v][c]; q=l[v];
    }
    if (q==-1 || c==toi(a[q])) q++; else {
      l[m+1]=i;  p[m+1]=m;  l[m]=l[v];  r[m]=q;
      p[m]=p[v];  t[m][c]=m+1;  t[m][toi(a[q])]=v;
      l[v]=q;  p[v]=m;  t[p[m]][toi(a[l[m]])]=m;
      v=s[p[m]];  q=l[m];
      while (q<r[m]) { v=t[v][toi(a[q])];  q+=r[v]-l[v]; }
      if (q==r[m])  s[m]=v;  else s[m]=m+2;
      q=r[v]-(q-r[m]);  m+=2;  goto suff;
    }
  }

  SuffixTree(string a) : a(a) {
    fill(r,r+N,SZ(a));
    memset(s, 0, sizeof s);
    memset(t, -1, sizeof t);
    fill(t[1],t[1]+ALPHA,0);
    s[0] = 1; l[0] = l[1] = -1; r[0] = r[1] = p[0] = p[1] = 0;
    REP(i,0,SZ(a)) ukkadd(i, toi(a[i]));
  }

  // example: find longest common substring (uses ALPHA = 28)
  PII best;
  int lcs(int node, int i1, int i2, int olen) {
    if (l[node] <= i1 && i1 < r[node]) return 1;
```

```
    if (l[node] <= i2 && i2 < r[node]) return 2;
    int mask = 0, len = node ? olen + (r[node] - l[node]) : 0;
    REP(c,0,ALPHA) if (t[node][c] != -1)
      mask |= lcs(t[node][c], i1, i2, len);
    if (mask == 3)
      best = max(best, {len, r[node] - len});
    return mask;
  }
  static PII LCS(string s, string t) {
    SuffixTree st(s + (char)('z' + 1) + t + (char)('z' + 2));
    st.lcs(0, SZ(s), SZ(s) + 1 + SZ(t), 0);
    return st.best;
  }
};
```

Geometry (8)

8.1 Geometric primitives

Point.h

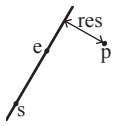
Description: Class to handle points in the plane. T can be e.g. double or long long. (Avoid int.)

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

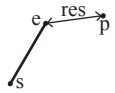
Description:
Returns the signed distance between point p and the line containing points a and b. Positive value on left side and negative on right as seen from a towards b. a==b gives nan. P is supposed to be Point<T> or Point3D<T> where T is e.g. double or long long. It uses products in intermediate 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.



"Point.h" f6bf6b, 4 lines

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

SegmentDistance.h



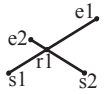
Description:
Returns the shortest distance between point p and the line segment from point s to e.
Usage: `Point<double> a, b(2,2), p(1,1);`
`bool onSegment = segDist(a,b,p) < 1e-10;`

"Point.h" 5c88f4, 6 lines

```
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;
}
```

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.



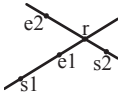
Usage: `vector<P> inter = segInter(s1,e1,s2,e2);`
`if (SZ(inter)==1)`
`cout << "segments intersect at " << inter[0] << endl;`

"Point.h", "OnSegment.h" 36c2d7, 13 lines

```
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)};
}
```

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" a01f81, 8 lines

```
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};
}
```

sideOf.h

Description: Returns where p is as seen from s towards e. 1/0/-1 $\Leftrightarrow$ 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" 3af81c, 9 lines

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

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

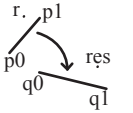
Description: Returns true iff p lies on the line segment from s to e. Use `(segDist(s,e,p)<=epsilon)` instead when using Point<double>.

"Point.h" c597e8, 3 lines

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

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" 03a306, 6 lines

```
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();
}
```

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`

0f0602, 35 lines

```
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)};
}
```

8.2 Circles

CircleIntersection.h

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

"Point.h" 84d6d3, 11 lines

```
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;
}
```

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" b0153d, 13 lines

```
template<class P>
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: $\mathcal{O}(n)$

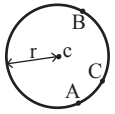
"../../content/geometry/Point.h" f5c096, 19 lines

```
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 circumcirle 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};
}
```

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

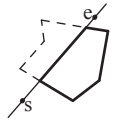
"Point.h" 7d84e0, 9 lines

```
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;
}
```

8.4 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" aaea8f, 88 lines

```
typedef Point<ll> P;
typedef struct Quad* Q;
typedef __int128_t lll; // (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?
  lll 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;
}
```

8.5 3D

PolyhedronVolume.h

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

3058c3, 6 lines

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

Point3D.h

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

8058ae, 32 lines

```
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 (9)

9.1 Equations

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

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

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

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.

9.2 Recurrences

If $a_n = c_1 a_{n-1} + \cdots + c_k a_{n-k}$, and $r_1, \ldots, r_k$ are distinct roots of $x^k + c_1 x^{k-1} + \cdots + c_k$, there are $d_1, \ldots, d_k$ s.t.

$$a_n = d_1 r_1^n + \cdots + d_k r_k^n.$$

Non-distinct roots r become polynomial factors, e.g. $a_n = (d_1 n + d_2) r^n$.

9.3 Trigonometry

$$\sin(v + w) = \sin v \cos w + \cos v \sin w$$
$$\cos(v + w) = \cos v \cos w - \sin v \sin w$$

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

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

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

$$a \cos x + b \sin x = r \cos(x - \phi)$$
$$a \sin x + b \cos x = r \sin(x + \phi)$$

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

9.4 Geometry

9.4.1 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: $\frac{a + b}{a - b} = \frac{\tan \frac{\alpha + \beta}{2}}{\tan \frac{\alpha - \beta}{2}}$

9.4.2 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 $4A = 2ef \cdot \sin \theta = F \tan \theta = \sqrt{4e^2 f^2 - F^2}$

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

9.4.3 Spherical coordinates

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

9.5 Derivatives/Integrals

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

Integration by parts:

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

9.6 Sums

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

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

9.7 Series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots, (-\infty < x < \infty)$$

$$\ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots, (-1 < x \leq 1)$$

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

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

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

9.8 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,

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

9.8.1 Discrete distributions

Binomial distribution

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

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

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

$\text{Bin}(n, p)$ 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 $\text{Fs}(p)$, $0 \le p \le 1$.

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

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

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 $\text{Po}(\lambda)$, $\lambda = t\kappa$.

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

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

9.8.2 Continuous distributions

Uniform distribution

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

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

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

Exponential distribution

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

$$f(x) = \begin{cases} \lambda e^{-\lambda x} & x \ge 0 \\ 0 & x < 0 \end{cases}$$

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

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

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

9.9 Markov chains

A *Markov chain* is a discrete random process with the property that the next state depends only on the current state. Let $X_1, X_2, \ldots$ 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). $\lim_{k\to\infty} \mathbf{P}^k = \mathbf{1}\pi$.

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 (10)

10.1 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: $\mathcal{O}\left(k \log \frac{n}{k}\right)$

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 $f(a) < \ldots < f(i) \geq \cdots \geq f(b)$. 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;
}
```

10.2 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.h

10 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...);
}
```

10.3 Optimization tricks

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

10.3.1 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.

10.3.2 Pragmas

- `#pragma GCC optimize ("Ofast")` 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 (mod 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.h

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.

7b3c70, 17 lines

```
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 (11)

11.1 Euclidean Algorithm

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

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:

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

11.2 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 $(a,b) \to (b, a \bmod b)$.

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:

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

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

11.3 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

$$a \cdot x \equiv 1 \ mod \ m.$$

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.

11.3.1 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.

$$M = Q \times a + r, (where \ Q \ is \ the \ quotient \ and \ r \ is \ the \ remainder) \tag{11.1}$$

$$M = \lfloor \frac{M}{a} \rfloor \times a + (M\%a) \tag{11.2}$$

$$Now \ take \ modulo \ M \ on \ both \ sides \ of \ the \ equation. \tag{11.3}$$

$$0 \equiv \lfloor \frac{M}{a} \rfloor \times a + (M\%a)(mod \ M) \tag{11.4}$$

$$(M\%a) \equiv -\lfloor \frac{M}{a} \rfloor \times a(mod \ M) \tag{11.5}$$

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

$$\frac{M\%a}{a \times (M\%a)} \equiv \frac{-\lfloor \frac{M}{a} \rfloor \times a}{a \times (M\%a)}(mod \ M) \tag{11.6}$$

$$a^{-1} \equiv -\lfloor \frac{M}{a} \rfloor \times (M\%a)^{-1}(mod \ M) \tag{11.7}$$

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 cooprime. 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)

11.4 Discrete Logarithm

The discrete logarithm of a in b is the smallest integer x such that $a^k \equiv b(mod \ m)$ 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 thuis approach is O(m). This approach works pecause 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 $a^k \equiv b(mod \ m)$. 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 $0 \leq j < n$.

$$a^{i \cdot n - j} \equiv b(mod \ m)a^{i \cdot n} \equiv a^j \cdot b(mod \ m) \tag{11.8}$$

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.

11.5 Mod'ed Arithmetic Progression

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

$$\sum_{i=0}^{to-1} a_i = \sum_{i=0}^{to-1} (ki + c)\% \ m \tag{11.9}$$

We do this by using the formula:

$$(ki+c) = \left\lfloor \frac{ki+c}{m} \right\rfloor m + (ki+c)\%m \tag{11.10}$$

$$(ki+c)\%m = (ki+c) - \left\lfloor \frac{ki+c}{m} \right\rfloor m \tag{11.11}$$

$$\sum_{i=0}^{to-1} (ki+c)\%m = \sum_{i=0}^{to-1} (ki+c) - m \sum_{i=0}^{to-1} \left\lfloor \frac{ki+c}{m} \right\rfloor \tag{11.12}$$

$$\tag{11.13}$$

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

11.6 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:

11.7 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 $x^k \equiv a \ (mod\ n)$.

11.7.1 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:

$$a^{p-1} \equiv 1 \mod p \tag{11.14}$$

$$\left(a^{\frac{p-1}{2}} - 1\right)\left(a^{\frac{p-1}{2}} + 1\right) \equiv 1 \mod p \tag{11.15}$$

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, $a \equiv x^2$.

$$a^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \ mod\ p \tag{11.16}$$

$$\tag{11.17}$$

So every quadratic residue (mod 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.

11.7.2 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 $n^{\frac{p-1}{2}} (mod\ p)$, 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 $q^{\frac{p-1}{2}} (mod\ p) \equiv -1$.

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

$$x = n^(\frac{s+1}{2} \ (first\ guess\ of\ square\ root) \tag{11.18}$$
$$b = n^s \tag{11.19}$$
$$g = q^s \tag{11.20}$$
$$r = e\ (exponent\ e\ will\ decrease\ after\ each\ updation) \tag{11.21}$$

5. Now loop until $m > 0$ and update value of x, which will be our final answer.
   Find least integer m such that $b^{2^m} = 1(mod\ p)$ and 0 <= m <= r–1
   If $m = 0$, then we found correct answer and return x as result
   Else update x, b, g, r as below

$$x = x * g^{2^{r-m-1}} b = b * g^{2^{r-m}} g = g^{2^{r-m}} r = m \tag{11.22}$$

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.

11.8 Primality Test

11.8.1 Sieve of Eratosthenes

Sieve of Eratosthenes is an algorithm for finding all the prime numbers in a segment $[1, n]$ using $O(nloglogn)$ 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.

11.8.2 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:

$$n - 1 = 2^s d,\ where\ d\ is\ odd \tag{11.23}$$

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

$$a^{n-1} \equiv 1\ mod\ n \Leftrightarrow a^{2^s d} - 1 \equiv 0\ mod\ n \tag{11.24}$$
$$\Leftrightarrow (a^{2^{s-1} d} + 1)(a^{2^{s-1} d} - 1) \equiv 0\ mod\ n \tag{11.25}$$
$$\Leftrightarrow (a^{2^{s-1} d} + 1)(a^{2^{s-2} d} + 1)(a^{2^{s-2} d} - 1) \equiv 0\ mod\ n \tag{11.26}$$
$$\vdots \tag{11.27}$$
$$\Leftrightarrow (a^{2^{s-1} d} + 1)(a^{2^{s-2} d} + 1) \ldots (a^d + 1)(a^d - 1) \equiv 0\ mod\ n \tag{11.28}$$

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 $a^d \equiv 1\ mod\ n$ holds or $a^{2^r d} \equiv -1\ mod\ n$ holds for some $0 \leq r \leq s - 1$.

If we found a base a which doesn't satisfy any of the above equalities, than 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 $a \leq 2ln(n)^2$. 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: $2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31,\ and\ 37$.

11.9 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

$$\begin{cases} x \equiv a_1 (mod\ n_1) \\ x \equiv a_2 (mod\ n_2) \end{cases} \tag{11.29}$$

This system of congruences implies that

$$\begin{cases} x = a_1 + n_1 k_1 \\ x = a_2 + n_2 k_2 \end{cases} \tag{11.30}$$

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

$$a_1 + n_1 k_1 = a_2 + n_2 k_2 \tag{11.31}$$

$$n_1(-k_1) + n_2 k_2 = a_1 - a_2 \tag{11.32}$$

Since we know n_1, n_2, a_1, a_2, this is just linear diophantine equation. Let $d = GCD(n_1, n_2)$. 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

$$n_1 x' + n_2 y' = d \tag{11.33}$$

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

$$n_1 x' \frac{a_1 - a_2}{d} + n_2 y' \frac{a_1 - a_2}{d} = a_1 - a_2 \tag{11.34}$$

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 $x = a_1 + nk_1$ 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 $x_1 \equiv a_1\ mod\ n_1$ and $x_2 \equiv a_2\ mod\ n_2$, so from transitivity of congruences we get $x_1 \equiv x_2\ mod\ n_1$. Doing the same thing for n_2 we get $x_1 \equiv x_2\ mod\ n_2$. These two congruences are equivalent to

$$x_1 \equiv x_2\ mod\ LCM(n_1, n_2) \tag{11.35}$$

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

11.9.1 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

$$x \equiv a_i\ mod\ n_i\ for\ i = 1, 2, ..., t \tag{11.36}$$

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

11.10 Bézout's identity

For $a \neq, b \neq 0$, then $d = gcd(a, b)$ 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

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

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

11.11 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:
$$\phi(p) = p - 1 \tag{11.37}$$
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:
$$\phi(p^k) = p^k - p^{k-1} \tag{11.38}$$
3. If a and b are relatively prime, then:
$$\phi(ab) = \phi(a)\phi(b) \tag{11.39}$$
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 $z \equiv x(mod\ a)$ and $z \equiv y(modb)$. 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
$$\phi(ab) = \phi(a)\phi(b)\frac{d}{\phi(d)} \tag{11.40}$$
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.

$$\begin{aligned}\phi(n) &= \phi(p_1^{a_1})\phi(p_2^{a_2}) \cdots \phi(p_k^{a_k}) &(11.41)\\ &= (p_1^{a_1} - p_1^{a_1-1})(p_2^{a_2} - p_2^{a_2-1}) \cdots (p_k^{a_k} - p_k^{a_k-1}) &(11.42)\\ &= n(1 - \frac{1}{p_1})(1 - \frac{1}{p_2}) \cdots (1 - \frac{1}{p_k}) &(11.43)\end{aligned}$$

11.12 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.

11.13 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.

11.14 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.

11.14.1 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.

11.14.2 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.

11.15 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.

11.15.1 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 $B \to D$ of the shortest path $A \to D$ between vertices A and D is also the shortest path between vertices B and D. Djikstra 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[]
```

11.15.2 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 $dist[u] + weightofedgeu - v$.
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[]
```

11.15.3 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
```

11.16 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.

$$f(e) \leq c(e) \tag{11.44}$$
$$f(e) = 0 \quad \text{if} \quad e \notin E \tag{11.45}$$

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.

$$\sum_{(v,u)\in E} f((v,u)) = \sum_{(u,v)\in E} f((u,v)) \tag{11.46}$$

(11.47)

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:

$$\sum_{(s,u)\in E} f((s,u)) = \sum_{(u,t)\in E} f((u,t)) \tag{11.48}$$

(11.49)

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.

11.16.1 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.

11.16.2 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 $s \to t$ in terms of number of edges used in each iteration. Using a BFS to find augmenting paths ensures that the shortest path from $s \to t$ 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.

11.16.3 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.

11.17 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

11.17.1 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.

11.17.2 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.

11.18 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 that 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.

11.19 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:

$$(a \vee b') \wedge (a' \vee b) \wedge (a' \vee b') \wedge (a \vee c') \tag{11.50}$$

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:

$$(a \vee b') \wedge (a' \vee b) \wedge (a' \vee b') \wedge (a \vee c') \tag{11.51}$$

The oriented graph will contain the following vertices and edges:

$$\begin{array}{cccc} (a' \Rightarrow b') & (a \Rightarrow b) & (a \Rightarrow b') & (a' \Rightarrow c') \\ (b \Rightarrow a) & (b' \Rightarrow a') & (b \Rightarrow a') & (c \Rightarrow a) \end{array} \tag{11.52}$$

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 thtink 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.

11.20 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.

11.21 Catalan Numbers

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:

- ()()()

- (())()
- ()(())
- ((()))
- (()())

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:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1} \tag{11.53}$$

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 parenthe- ses 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:

$$C_n = \frac{1}{n+1}\binom{2n}{n} \tag{11.54}$$

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:

$$\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n} - \frac{n}{n+1}\binom{2n}{n} = \frac{1}{n+1}\binom{2n}{n} \tag{11.55}$$

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