Contents

1 Strings

1.1 Knuth - Morris - Pratt algorithm

```
#include "../template.cpp"

// Pattern matching in O(n) query, O(m) preprocess, O(m↩
    ) space
// n=|txt|, m=|patt|
// ===================
#define MAX 10007 // max length of patt
ll B[MAX];
// B[i]: length of longest common (own) prefix (↩
    strictly shorter)
//          -1 = sentinel
//     0  1  2  3  4  5  6  7  8  9 10 11
//str: a  b  c  a  b  b  a  b  c  a  b  b
//brd:-1  0  0  0  1  2  0  1  2  3  4  5  6

void buildTable(string & patt) {
    CL(B, 0); B[0] = -1;
    REP(i,patt.size()) {
```

```
        int j=B[i];
        while(j!=-1&&patt[j]!=patt[i]) j = B[j]; // jump back while mismatch on i-th
        B[i+1]=j+1;
    }
}

// returns vector of positions of match in txt
vll kmp(string & txt, string & patt) {
    vll foundpos;
    buildTable(patt);
    ll pos = 0; // position in patt
    REP(i,txt.size()) {
        while (pos != -1 && patt[pos] != txt[i]) pos = B[pos]; // -||-
        pos++;
        ll ns = patt.size();
        if (pos==ns){
            foundpos.pb(i-ns+1);
            pos = B[ns];
        }
    }
    return foundpos;
}

int main() {
    string txt="aaabcabbabcabbabcabb";
    string patt="abcabbabcabb";
    //ret-> [2,8]    ^     ^
    vll v = kmp(txt,patt);
    for(auto i:v)cout<<i<<' ';
        cout<<endl;
    return 0;
}
```

1.2 Aho-Corasick

```
#include "../template.cpp"
// -Aho-Corasick Algorithm-
// Multiple patterns matching

// |txt|=n, combined pattern length=m
//in O(n+m) query, O(m) space
// ---
// can be enhanced by replacing unordered_map with static array (smaller alphabeths)

#define MAXN 1000007 // max num of states = maxNumOfpatterns * maxLenOfPattern

unordered_map<char,ll> go[MAXN];
int fail[MAXN],
    que[MAXN],
    patt[MAXN],
    cnt=1;

void add_patt(string & str, int k) { // creates trie
    int act=0;
    for(auto ch : str) {
        if (!go[act][ch]) go[act][ch]=cnt++; // new state
        act=go[act][ch];
    }
    patt[act]=k;
}

void push_fails() {
    CL(fail,0);
    fail[0]=-1;
    int be=0, en=1; que[0]=0; // init queue
    while(be < en) {
        int act=que[be++];
        for(auto it : go[act]) {
            ll ch=it.first,
                u=it.second,
                j=fail[act];
            while(j != -1 && !go[j][ch])j=fail[j]; // if there is not a way from fail (prefix), jump back
            if(j != -1) fail[u]=go[j][ch];
            que[en++]=u;
        }
    }
}

// returns pair<index of found needle, pos of found needle in txt>
vpll aho(vector<string> patterns, string txt) {
    CL(patt,-1);
    REP(i,patterns.size()) add_patt(patterns[i],i);
    push_fails();
    int act=0;
    vpll found;
    int i=0;

    for(auto ch : txt) {
        while (act != -1 && !go[act].count(ch)) act=fail[act]; // go back until can go with 'ch' or is -1
        if (act != -1) { // if can follow
            act=go[act][ch];
            int pos=act; // add all matched keywords in own suffix
            while (pos != -1) {
                if (patt[pos] != -1) found.push_back({patt[pos], i - (patterns[patt[pos]].size()-1)});
                pos=fail[pos];
            }
        } else
            act=0;
        i++;
    }
    return found;
}

int main()
{
    vector<string> patterns{"he", "she", "hers", "his"};
    string txt="ahishersblablablashe";
    vpll found=aho(patterns, txt);
    for (auto i : found)
        cout << patterns[i.first] << " at " << i.second << endl;
    return 0;
}
```

1.3 Rabin-Karp (Rolling hash)

```
#include "../template.cpp"
// -Rabin-Karp Algorithm-
// Pattern matching algo
// - usefull for random distributed or when O(1) space needed

// averige O(n+m), worst case O(n*m)
// n=|txt|, m=|patt|
// O(1) space

#define ABC 256 // size of alphabeth

ll m(ll a, ll q){ return ((a%q)+q)%q; }

// q must be PRIME !!!
vll search(string& patt, string& txt, int q) {
    vll found;
    ll M=patt.size(),
       N=txt.size(),
       p=0,  // hash value for pattern
       t=0, // hash value for act window of txt
       h=1;
    // h="pow(ABC, M-1)%q"
    F(M-1) h=(h*ABC)%q;
    F(M){ // calc hash
        p=(ABC*p + patt[i])%q;
        t=(ABC*t + txt[i])%q;
    }
    F(N-M+1) { // if hash of curr wind is ok, check one by one
        if (p==t) {
            bool ok=1;
            FF(M) if(txt[i+j]!=patt[j]) { ok=0;break; }
            if(ok)found.pb(i);
        }
        // recalc hash for next wind
        if (i < N-M) t=m(ABC*(t - txt[i]*h) + txt[i+M], q);
    }
    return found;
}

/* Driver program to test above function */
int main()
{
    string txt="aaabcabbabcabbabcabb";
    string patt="abcabbabcabb";
    int q=101;//PRIME
    vll found=search(patt,txt,q);
```

```
    for(auto i:found) D(i);
    return 0;
}
```

1.4 Boyer-moore

```
int boyer_moore(char* sstr, char* pattern)
{
    char* inits = sstr;
    char* initp = pattern;

    int spatt = strlen(pattern);
    while(*pattern != '\0') pattern++;
    // this algorithm tested for printable ASCII characters
    // from ASCII, 65-90 and 97-122
    int* jump_table=(int*) calloc(128, sizeof(int));
    int count=0;

    while(pattern != initp) {
        pattern--;
        jump_table[*pattern]=count;
        count++;
    }

    char* stmp=0;
    char* iter=0;
    int shift=0;
    int bad_count=0;
    int bcount=0;
    while(*sstr != '\0')
    {
        bcount=0;
        bad_count=spatt;
        stmp = sstr+ (spatt-1);
        iter = pattern + (spatt-1);
        while(*stmp == *iter) {
            bad_count--;
            bcount++;
            stmp--;
            iter--;
            if(bcount==spatt)
                return sstr-inits;
        }

        //jump table
        if(jump_table[*stmp] == 0) {
            // the character not found in pattern
            shift=bad_count;
        } else {
            shift=jump_table[*stmp];
            (shift - bcount < 1)?shift = 1: shift = shift-bcount;
        }
        sstr += shift;
    }
    //not found
    return -1;
}


void main()
{
    char* source = "aabaaabbbbbbaaaaabbabaaaaaaaaaa";
    char* pattern = "baaaa";

    std::cout<<boyer_moore(source, pattern);
}
```

1.5 Manacher's longest pallindromic substring

```
#include "../template.cpp"

#define MAX 100007 // max length of string

// -Manacher algorithm-
// Longest palindromic substring finder

// O(n) query, O(n) preprocess, O(n) space
// n = |string|

// (preprocessed string -> ^#s#t#r#i#n#g#$ for odd lengths)
void preprocess(string & str) {
    string ret = "^";
    for (auto ch : str) { ret+="#"; ret += ch; }
    ret += "#$"; str = ret;
}

// Array P
//  on i-th position of P is placed length of palindrome
ll P[MAX*2];

// returns {position, length} of longest palindromic substring
pll manacher(string & str) {
    memset(P, 0, sizeof P);
    preprocess(str);
    ll ret=0,retpos;
    ll center = 0, right = 0;
    for(ll i = 1; i < str.size(); i++) {
        ll mirr = center * 2 - i;
        if (i < right) { // !crutial enhancement: if for mirrored calculated, use it
            P[i] = min(P[mirr], right-i);
            ret=max(ret,P[mirr]);
        }
        // expanding palindrome:
        while (str[i - (1 + P[i])] ==  str[i + P[i] + 1]) P[i]++;
        if (i + P[i] > right) { // out of palindrome used for mirroring
            center = i;
            right = i + P[i];
        }
        if(ret<P[i]){ret=P[i]; retpos=i; }
    }
    return {retpos,ret};
}

int main() {
    string str="aba";
    pll r = manacher(str);
    cout<<str<<endl;
    cout<< r.x << ' ' << r.y <<endl;
    return 0;
}
```

1.6 Suffix array + Longest common prefix

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M(L) {
    for (int i = 0; i < L; i++) P[0][i] = int(s[i]);
    for (int skip = 1, level = 1; skip < L; skip *= 2, level++) {
      P.push_back(vector<int>(L, 0));
      for (int i = 0; i < L; i++)
        M[i] = make_pair(make_pair(P[level-1][i], i + skip < L ? P[level-1][i + skip] : -1000), i);
      sort(M.begin(), M.end());
      for (int i = 0; i < L; i++)
        P[level][M[i].second] = (i > 0 && M[i].first == M[i-1].first) ? P[level][M[i-1].second] : i;
    }
  }

  vector<int> GetSuffixArray() { return P.back(); }
```

```
  // returns the length of the longest common prefix of
      s[i...L-1] and s[j...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i < L && j < L
        ; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};

int main() {

  // bobocel is the 0'th suffix
  //  obocel is the 5'th suffix
  //   bocel is the 1'st suffix
  SuffixArray suffix("bobocel");
  vector<int> v = suffix.GetSuffixArray();

  // Expected output: 0 5 1 6 2 3 4
  //                  2
  for (int i = 0; i < v.size(); i++) cout << v[i] << " 
      ";
  cout << endl;
  cout << suffix.LongestCommonPrefix(0, 2) << endl;
}
```

2 Graphs

2.1 DFS/BFS

```
#include "../template.cpp"

// vraci vektor vzdalenosti
vll bfs(vector<vll>& G, ll u)
{
  vll d(G.size(), -1);
  queue<ll> q; q.push(u);
  while (q.size())
  {
    auto u = q.front();q.pop();
    F(G[u].size())
    {
      ll v = G[u][i];
      if (~d[v])
      {
        d[v] = d[u] + 1;
        q.push(d[v]);
      }
    }
  }
  return d;
}

void dfs(vector<vll>& G, ll u, vll& res)
{
  res[u] = 1;
  F(G[u].size())
  {
    ll v = G[u][i];
    if (!res[v])
      dfs(G, v, res);
  }
}
```

2.2 Shortest path

2.2.1 Dijkstra's algorithm

```
// Dijkstra's algorithm
// shortest paths from src to all vertices in the given
    graph
// directed, weighted, negative NOT ALLOWED!!

// O(VLogV)

#include "../template.cpp"

#define MX 207
// {neighbour, cost}
vpll dijkstra(vector<vpll>& G, ll s)
{
  ll S = G.size();
  vpll res(S, {0, INF});
  res[s] = {s, 0};
  bitset<MX> closed;
  priority_queue<pll, vpll, greater<pll>> Q;
  Q.push({0, s});
  while (Q.size())
  {
    pll top = Q.top(); Q.pop();
    // if (end == top.y) break;
    ll cost = top.x, next = top.y;
    if (!closed[next])
    {
      closed[next] = 1;
      F(G[next].size())
      {
        ll v = G[next][i].x;
        ll ncost = cost + G[next][i].y;
        if (ncost < res[v].y)
        {
          res[v] = {next, ncost};
          Q.push({ncost, v});
        }
      }
    }
  }
  return res;
}
```

2.2.2 Bellman Ford's algorithm

```
// Bellman Ford Algorithm
// shortest paths from src to all vertices in the given
    graph
// directed, weighted, allowed negative

// O(VE)

#include <limits>
#include <vector>
#define REP(i,a,b) for(int i=a;i<b;i++)
using namespace std;

struct TEdge {
    int a, b, w;
};
int d[107];
int p[107];
bool cyc[107];

int n; // num of verticles
vector<TEdge> edges;

// calculates d     = dist from 0.th verticle
//        and p     = prev - path to start
//        and cyc == true if one can get there from 
    some negative cycle
void bellford() {
    REP(i,n) d[i] = numeric_limits<int>::max();
    d[0] = 0; p[0] = -1;
    CL(cyc, 0);
    REP(i,n-1) {
        for (auto e : edges) {
            if (d[e.b] > d[e.a] + e.w) {
                // d[e.b] = min(d[e.b], d[e.a] + e.w); 
                    // relax
                p[e.b] = e.a;
                d[e.b] = d[e.a] + e.w;
            }
        }
    }
    for (auto e : edges) {
        if (d[e.a] + e.w < d[e.b]) {
            cyc[e.b] = true;
            // detected negative cycle
        }
    }

    // Handy addition: propagates cycle into whole 
        graph - answers: can i get to finish using neg
        cycle?
    REP(i,0,n-1) {
```

```
        for (auto e : edges) {
            if (cyc[e.a]) cyc[e.b] = 1;
        }
    }
}
```

2.2.3 Floyd–Warshall's algorithm

```
// Floyd Warshall Algorithm
// All Pairs Shortest Path problem
// O(V^3)

#include "../template.cpp"

#define FFF(n) REP(k, n)
ll M[MX][MX];
ll P[MX][MX];
// fw method ACed on https://www.codechef.com/IOIPRAC/←
    problems/INOI1402
void fw(vector<vpll>& G)
{
  CL(P, -1);
  F(MX)FF(MX) M[i][j] = INF;
  F(G.size()) FF(G[i].size()) M[i][G[i][j].x] = G[i][j←
      ].y, P[i][G[i][j].x] = G[i][j].x;
  FFF(MX) F(MX) FF(MX)
  if (M[i][k] + M[k][j] < M[i][j])
  {
    M[i][j] = M[i][k] + M[k][j];
    P[i][j] = P[i][k];
  }
}
vll fwp(ll a, ll b)
{
  vll res;
  if (~P[a][b]) return res;
  res.pb(a);
  while (a != b)
    a = P[a][b], res.pb(a);
  return res;
}
```

2.2.4 Johnson's algorithm

```
// Bellman Ford Algorithm
// shortest paths from src to all vertices in the given←
    graph
// directed, weighted, allowed negative

// O(VE)

#include <limits>
#include <vector>
#define REP(i,a,b) for(int i=a;i<b;i++)
using namespace std;

struct TEdge {
    int a, b, w;
};
int d[107];
int p[107];
bool cyc[107];

int n; // num of verticles
vector<TEdge> edges;

// calculates d     = dist from 0.th verticle
//         and p    = prev - path to start
//         and cyc == true if one can get there from ←
    some negative cycle
void bellford() {
    REP(i,n) d[i] = numeric_limits<int>::max();
    d[0] = 0; p[0] = -1;
    CL(cyc, 0);
    REP(i,n-1) {
        for (auto e : edges) {
            if (d[e.b] > d[e.a] + e.w) {
                // d[e.b] = min(d[e.b], d[e.a] + e.w); ←
                    // relax
                p[e.b] = e.a;
                d[e.b] = d[e.a] + e.w;
            }
        }
    }
    for (auto e : edges) {
        if (d[e.a] + e.w < d[e.b]) {
            cyc[e.b] = true;
            // detected negative cycle
        }
    }

    // Handy addition: propagates cycle into whole ←
        graph - answers: can i get to finish using neg←
         cycle?
    REP(i,0,n-1) {
        for (auto e : edges) {
            if (cyc[e.a]) cyc[e.b] = 1;
        }
    }
}
```

2.3 Bipartity check

```
#include "../template.cpp"

bool isbip(vector<vll>& G, ll s = 0)
{
  vll c(G.size(), 2); c[s] = 0;
  queue<ll> q;q.push(s);
  while (q.size())
  {
    ll n = q.front(); q.pop();
    F(G[n].size())
    {
      ll v = G[n][i];
      if (c[v] == 2)
        c[v] = 1-c[n], q.push(v);
      else
        if (c[v] == c[n])
          return false;
    }
  }
  return true;
}
```

2.4 Articulations & bridges

```
#include "../template.cpp"

// Finds all articulations and bridges in graph
// O(V+H)

#define MAX 10007 // max edges
#define UNVISITED 0

ll n;
vll neigh[MAX];

ll num[MAX];
ll low[MAX];
int cnt=1;

int dfsRoot, rootChildred;

set<ll> articPts;
vpll bridges;

void dfs(ll u, ll par) {
    low[u]=num[u] = cnt++;
    ll childs=0;
    for (auto v : neigh[u]) {
        if (num[v]==0) { // unvisited
            childs++; // root
            dfs(v, u);
            if (low[v] >= num[u] && par!=-1) articPts.←
                insert(u); // back edges goes only ←
                lower or into 'u'; root resolve later
            if (low[v] > num[u]) bridges.pb({u, v}); //←
                 goes only lower
            low[u] = min(low[u], low[v]); // remin ←
                lowest point
        } else if (v != par) { // zpetna hrana: remin ←
             kam vede
            low[u] = min(low[u], num[v]);
        }
    }
    // For root
    if(par==-1 && childs>1) articPts.insert(u);
}
```

```
void addEdge(int u,int v) {
    neigh[u].pb(v);
    neigh[v].pb(u);
}

int main ()
{
    n=4;
    addEdge(0,1);
    addEdge(2,1);
    addEdge(2,3);
    addEdge(3,1);
    F(n) if (!num[i]) dfs(i, -1);
    cout<<"Articulation points: "; for(auto i:articPts) cout<<i<<' '; cout<<endl;
    cout<<"Bridges: "; for(auto i:bridges) cout<<i.x<<"-"<<i.y<<",";  cout<<endl;
}
```

2.5 Ford Fulkerson's maximal flow & minimal cut

2.5.1 Edmonds–Karp's algorithm

```
#include "../template.cpp"

// Edmonds-Karp's impl of Ford Fulkerson
// maxFlow & mincut
// O(|E|^2 * |V|) (V<=500,E<=5000)

#define MAX 1000 // of nodes
struct Edge { ll from, to, cap; }; // capacity in one way, residue in the other
vector<Edge> edges;
vll ng[MAX]; // indexes of edges
ll back[MAX]; // indexes of edges for reconstructing augment path
bool fromS[MAX]; // for minCut: can get from s when augment not found?

void init(){
    edges.clear();
    F(MAX)ng[i].clear();
    CL(back,0);
    CL(fromS,0);
}

void addEdge(ll from, ll to, ll capacity) { // 2 edges, back is residual, accessible by ^1 (even/odd)
    ng[from].pb(edges.size());
    edges.pb(Edge{from, to, capacity});

    ng[to].pb(edges.size());
    edges.pb(Edge{to, from, 0});
}

bool bfs(ll s, ll t) { // source,sink
    CL(back,-1); back[s] = -2;
    CL(fromS,0);
    queue<ll> q; q.push(s);
    while (!q.empty() && back[t] == -1) { // exists augment path to sink
        ll u = q.front(); q.pop();
        fromS[u]=1;
        F(ng[u].size()) {
            Edge & edge = edges[ng[u][i]];
            if (edge.cap && back[edge.to] == -1) { // has capacity
                back[edge.to] = ng[u][i];
                q.push(edge.to);
            }
        }
    }
    return back[t] != -1;
}

ll maxFlow(ll s, ll t) {
    ll maxFlow = 0;
    while (bfs(s, t)) {
        ll flow = 1<<30, node = t; // from sink to source(=-2)
        // find size of the flow = min capacity on the way:
        while (back[node] != -2) {
            Edge & edge = edges[back[node]];
            flow = min(flow, edge.cap);
            node = edge.from;
        }
        // push the flow:
        node=t;
        while (back[node] != -2) {
            Edge & edge = edges[back[node]],
                 & edge2 = edges[back[node]^1];
            edge.cap -= flow;
            edge2.cap += flow;
            node = edge.from; // going back
        }

        maxFlow += flow;
    }
    cout<<"Max flow: "<< maxFlow <<endl;
    cout<<"Min cut: "; F(edges.size()) {
        if(i&1)continue;
        auto& e=edges[i];
        if(fromS[e.from] != fromS[e.to]) cout<<e.from<<","<<e.to<<" ";
    }cout<<endl;
    return maxFlow;
}

int main() {
    init();
    addEdge(0,1,1);
    addEdge(0,2,2);
    addEdge(0,3,1);
    addEdge(0,4,1);
    addEdge(1,5,2);
    addEdge(2,5,1);
    addEdge(3,5,2);
    addEdge(4,5,2);
    addEdge(5,6,100);
    maxFlow(0,6);
    return 0;
}
```

2.5.2 Dinic's algorithm

```
#include "../template.cpp"

// Dinic's algorithm
// maxFlow & mincut
// O(|E| * |V|^2)

const int NODES = 5000, EDGES = 30000;
int N, M = NODES, source, sink;

struct Edge {
    int from, to;
    long long residue;
} edges[NODES+2*EDGES];
int adj[NODES+2*EDGES], work_adj[NODES+2*EDGES], work_edges[NODES+2*EDGES];

bool bfs() {
    static int dist[NODES], q[NODES];
    int qsz = 0;

    for (int i = 0; i < N; ++i) {
        work_adj[i] = -1;
        dist[i] = N+1;
    }
    int work_M  = NODES;

    q[qsz++] = source;
    dist[source] = 0;
    for (int qi = 0; qi < qsz && dist[q[qi]] + 1 <= dist[sink]; ++qi)
        for (int i = adj[q[qi]]; i >= 0; i = adj[i]) {
            Edge & edge = edges[i];
            if (edge.residue && dist[edge.from] + 1 <= dist[edge.to]) {
                work_adj[work_M] = work_adj[edge.from];
                work_edges[work_M] = i;
                work_adj[edge.from] = work_M++;

                if (dist[edge.to] == N+1) {
                    dist[edge.to] = dist[edge.from] + 1;
                    q[qsz++] = edge.to;
                }
            }
        }
    return dist[sink] != N+1;
}

long long dfs(int node, long long flow) {
```

```
    if (!flow || node == sink)
        return flow;
    for (int & i = work_adj[node]; i >= 0; i = work_adj↩
        [i]) {
        int eindex = work_edges[i];
        long long fl;
        if (fl = dfs(edges[eindex].to, min(flow, edges[↩
            eindex].residue))) {
            edges[eindex].residue -= fl;
            edges[eindex^1].residue += fl;
            return fl;
        }
    }
    return 0;
}

long long maxFlow(ll s, ll t) {
    source=s;sink=t;
    long long total_flow = 0, flow;
    while (bfs())
        while (flow = dfs(s, 1<<30))
            total_flow += flow;
    return total_flow;
}

void addEdge(int from, int to, long long cap1) {
    edges[M] = Edge{from, to, cap1};
    adj[M] = adj[from];
    adj[from] = M++;

    edges[M] = Edge{to, from, 0};
    adj[M] = adj[to];
    adj[to] = M++;

    N=max(from,to)+1;
}

void init() {
    CL(adj,-1);
    M = NODES;
}

int main()
{
    init();
    addEdge(0,1,1);
    addEdge(0,2,2);
    addEdge(0,3,1);
    addEdge(0,4,1);
    addEdge(1,5,2);
    addEdge(2,5,1);
    addEdge(3,5,2);
    addEdge(4,5,2);
    addEdge(5,6,100);
    cout<<"Max flow: "<< maxFlow(0,6) << endl;
    return 0;
}
```

2.6 Bipartite check

```
#include "../template.cpp"

bool isbip(vector<vll>& G, ll s = 0)
{
  vll c(G.size(), 2); c[s] = 0;
  queue<ll> q;q.push(s);
  while (q.size())
  {
    ll n = q.front(); q.pop();
    F(G[n].size())
    {
      ll v = G[n][i];
      if (c[v] == 2)
        c[v] = 1-c[n], q.push(v);
      else
        if (c[v] == c[n])
          return false;
    }
  }
  return true;
}
```

2.7 Tarjan's strongly connected components algorithm

```
#include "../template.cpp"
// Tarjan's strongly connected components algorithm
// O(|V|+|E|)

#define MX 201
ll D[MX], LO[MX], SC[MX]; // SC - ID komponent
stack<ll> ST;
bitset<MX> STA;
ll TI, SCC;
void tarjanR(vector<vll>& G, ll u)
{
  D[u] = LO[u] = TI++;
  ST.push(u);
  STA[u] = 1;
  F(G[u].size())
  {
    ll v = G[u][i];
    if (~D[v])
      tarjanR(G, v);
    if (~D[v] || STA[v])
      LO[u] = min(LO[u], LO[v]);
  }
  if (D[u] == LO[u])
  {
    ll v;
    do {
      v = ST.top();
      ST.pop();
      STA[v] = 0;
      SC[v] = SCC;
    } while (u != v);
    SCC++;
  }
}
void tarjan(vector<vll>& G, ll u)
{
  CL(D, -1);
  tarjanR(G, u);
}
```

2.8 Toposort

```
#include "../template.cpp"

#define MX 207
bitset<MX> TM;
vll path; // reverse path
void tdfs(vector<vll>& G, ll n)
{
  TM[n] = 1;
  F(G[n].size())
    if (!TM[G[n][i]])
      tdfs(G, G[n][i]);
  path.pb(n);
}
void topo(vector<vll>& G)
{
  F(MX) if (!TM[i]) tdfs(G, i);
}
```

2.9 Minimum spanning tree

2.9.1 Kruskal-Boruvka

```
ll n; // number of vertices

// union find
ll uf[MX];
int find(int e) { return uf[e] == e ? e : uf[e] = find(↩
    uf[e]); }
void mrg(int i, int j) { uf[find(i)] = find(j); }
bool same(int i, int j) { return find(i) == find(j); }

vector<pair<ll,ll>> g[MX]; // graph with edges from i ↩
    to j with price as a second part

using edg = tuple<ll,ll,ll>;
vector<edg> ali2edgs() { // adjacency list ↩
    representation (in g) to vector of edges
    vector<edg> edgs;
    F(n) for(auto& e : g[i]) edgs.pb({e.y,i,e.x});
    return edgs;
```

```
}

//ll mst() { // return cost of minimum-spanning tree (swap commenting in return)
vector<edg> mst() { // returns all edges used in minimum-spanning tree
    auto edgs = ali2edgs();
    ll m = edgs.size();
    vector<edg> res;

    F(n) uf[i] = i;
    sort(edgs);
    int cv = 1;
    int k = 0;
    ll rp = 0;

    REP(k,m and cv < n) {
        ll prc, i, j;
        tie(prc,i,j) = edgs[k];
        if (find(i) != find(j)) {
                mrg(i,j);
                cv++;
                res.pb(edgs[k]);
                rp += prc;
        }
    }
    assert(cv==n);
    return res;
    //return rp;
}
```

2.9.2 Jarnik-Prim

```
const ll MAX_NODES = 100007;
struct TEdge {
    ll start, end, cost;
};

struct TPrio {
    ll node;
    ll pred;
    ll cost;
    bool operator < (const TPrio & p) const {
        return cost > p.cost; // PRIO QUEUE IS MAXIMAL BY DEFAULT
    }
};

vector<TEdge> neighbours[MAX_NODES];
ll dists[MAX_NODES];
bool closed[MAX_NODES];

vector<TEdge> jarnik() {
    vector<TEdge> span_tree;
    priority_queue<TPrio> q;
    q.push({0, -1, 0});
    CL(dists, -1); CL(closed,0);
    while(q.size()) {
        TPrio prio = q.top();
        q.pop();
        if(closed[prio.node]) continue;
        if(prio.pred != -1) {
            span_tree.pb({prio.node, prio.pred, prio.cost});
        }
        closed[prio.node] = true;
        dists[prio.node] = prio.cost;
        for(TEdge edge : neighbours[prio.node]) {
            if(closed[edge.end]) continue;
            ll newcost = edge.cost;
            if(dists[edge.end] == -1 || dists[edge.end] > newcost) {
                dists[edge.end] = newcost;
                q.push({edge.end, prio.node, newcost});
            }
        }
    }
    return span_tree;
}
int main() {
    ll n, m;
    cin >> n >> m;
    REP(i, m) {
        ll x, y, c;
        cin >> x >> y >> c;
        neighbours[x].pb({x,y,c});
        neighbours[y].pb({y,x,c});
    }
    for(TEdge edge : jarnik()) cout << edge.start << ' ' << edge.end << ' ' << edge.cost << endl;
    return 0;
}
```

2.10 Bipartite matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
// For larger input, consider Dinic, which runs in O(E sqrt(V))
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<int> VI;
typedef vector<VI> VVI;

bool FindMatch(int i, const VVI &w, VI &mr, VI &mc, VI &seen) {
  for (int j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
  }
  return false;
}

int BipartiteMatching(const VVI &w, VI &mr, VI &mc) {
  mr = VI(w.size(), -1);
  mc = VI(w[0].size(), -1);

  int ct = 0;
  for (int i = 0; i < w.size(); i++) {
    VI seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct++;
  }
  return ct;
}
```

2.11 NP-complete

2.11.1 Travelling salesman problem – TSP

```
// Find length of shortest path that visits all cities
// Complexity: O(n^2*2^n)
// Usage tsp(0,(1<<number_of_vertices) - 2)
// graph is represented as matrix dist with distances from i to j
using T = ll;
#define MX
T dist[MX][MX];
T dp[MX][1<<MX];

void init() {
    fill(*dp,*dp+sizeof(dp)/sizeof(T),-1);
}
T tsp(int pos, int vis) {
    if (not vis) return dist[pos][0];
    T& res = dp[pos][vis];
    T mix = INF;
    if (res == -1) {
        F(MX) {
            if ((1<<i) & vis) {
                mix = min(mix,tsp(i,vis ^ (1<<i)) + dist[pos][i]);
            }
        }
    }
    return mix;
}
```

2.11.2 Hamiltonian path

```
// Finds Hamilnonian path or cycle in O(n^2*2^n)
// Usage: ham(starting_vertex,((1<<number_of_vertices)-1) ^ (1 << starting_vertex))
// graph is in g as adjacency list
// If you want to find hamiltonian path you need to check all vertices
// Also modify checking for right path (1)
// For hamiltonian cycle use 0 as starting vertex (no need to loop through all)
// In dp is next vertex to choose if you are at pos with vis vertices to visit
#define MX
vll g[MX];

int dp[MX][1<<MX];
void init() {
    CL(dp,-1);
}

int ham(int pos,int vis) {
    int& res = dp[pos][vis];
    if (res == -1) {
        res = -2;
        if (not vis) {
            for (auto& e : g[pos]) if (e == 0) res = 0; // hamiltonian cycle
            //res = 0; // (1) hamiltonian path
        }
        else {
            for(auto& e : g[pos]) {
                if (vis & (1 << e) and ham(e,vis ^ (1 << e)) != -2) res = e;
            }
        }
    }
    return res;
}

int print() {
    int nx = 0;
    int vis = (1 << 20/* number of vertices */) - 2;
    while(vis) {
        cout << nx << endl;
        nx = ham(nx, vis);
        vis ^= (1 << nx);
    }
    cout << nx << endl;
}
```

2.11.3 Eulerian path

```
// Find euler path stored in forward_list in O(n) - rather slow
// Initialize by add_edge (uncomment if bidirectional)
// Check whether euler path exist:
// 1) not oriented: number of vertices with odd rank
//     should be 0 or 2
// 2) oriented: all except 0 or 2 vertices should have
//     same number of in and out edges, 1 can have 1 more out
//     and the other one 1 more in
#define MX
vector<pll> g[MX]; // number of vertices
bitset<MX> used; // number of edges
deque<ll> pth;
int ec;
void init() {
    ec = 0;
    F(MX) g[i].clear();
    used.reset();
    pth.clear();
}

void add_edge(int f, int t) {
    //g[t].pb({f,ec}); // if bidirectional
    g[f].pb({t,ec++});
}

void path(int i) {
    for(auto& e : g[i]) {
        if (used[e.y]) continue;
        used[e.y] = 1;
        path(e.x);
    }
    pth.push_front(i);
}
deque<ll> euler_path(int st) {
    path(st);
    return pth;
}
```

2.11.4 Vertex cover

```
// Vertex cover, complement is independent set
/* Dynamic programming based program for Vertex Cover problem for
   a Binary Tree */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    int vc;
    struct node *left, *right;
};

// A memoization based function that returns size of the minimum vertex cover.
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated, then return it
    // to save recomputation of same subproblem again.
    if (root->vc != 0)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    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);

    // Minimum of two values is vertex cover, store it before returning
    root->vc =  min(size_incl, size_excl);

    return root->vc;
}

// A utility function to create a node
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; // Set the vertex cover as 0
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root         = newNode(20);
    root->left                = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left   = newNode(10);
    root->left->right->right  = newNode(14);
```

```
    root->right              = newNode(22);
    root->right->right       = newNode(25);

    printf ("Size of the smallest vertex cover is %d ",↩
         vCover(root));

    return 0;
}
```

3 Data structures

3.1 Union find

```
// uf needs to be initialized with sequence from 0 to n↩
     !!!
ll uf[MX];
int find(int e) { return uf[e] == e ? e : uf[e] = find(↩
     uf[e]); }
void mrg(int i, int j) { if(rand()%2)swap(i,j); uf[find↩
     (i)] = find(j); }
```

3.2 Fenwick tree

```
// Fenwick tree data structure (cummulative sums)
// ====
// log(n) get and inc

// Doesn't work with index 0!!!

const int MX = >>number of elements + 1<<;
typedef int T;

T ft[MX];

T get(int en) {
    T sum = T();
    for(;en; en -= en & -en) sum += ft[en];
    return sum;
}
T get(int st, int en) {
    return get(en) - get(st-1);
}

void inc(int i,T val) {
    for(;i < MX; i += i & -i) ft[i] += val;
}
```

3.3 Segment tree

```
#define MX 112345
typedef int T;
function<T(T,T)> mrg = [](T a,T b){return max(a,b);}; ↩
     // merge two subtrees
function<T(int)> ini; // init element
T seg[MX*4];
void build(int sz=(MX)-1, int st=0, int en=(MX)-1, int ↩
     p=1) {
    if (st == en)
        seg[p] = st < sz ? ini(st) : T();
    else {
        build(sz,st,(st+en)/2,p*2);
        build(sz,(st+en)/2+1,en,p*2+1);
        seg[p] = mrg(seg[p*2],seg[p*2+1]);
    }
}
T get(int st, int en, int sst=0,int sen=(MX)-1,int p=1)↩
     {
    if (st > sen or en < sst) return T();
    if (st <= sst and en >= sen) return seg[p];
    int mid = (sst+sen)/2;
    T lt = get(st,en,sst,mid,2*p);
    T rt = get(st,en,mid+1,sen,2*p+1);
    if (st <= mid and en > mid) return mrg(lt,rt);
    return st <= mid ? lt : rt;
}
T upd(int pos,T val, int sst=0,int sen=(MX)-1,int p=1) ↩
     {
    if (sst == sen) return seg[p] = val;
    int mid = (sst+sen)/2;
    if (pos <= mid) return seg[p] = mrg(upd(pos,val,sst↩
         ,mid,2*p),seg[2*p+1]);
    else return seg[p] = mrg(seg[2*p],upd(pos,val,mid↩
         +1,sen,2*p+1));
}


int a[MX];

int main(void) {
    ios_base::sync_with_stdio(false);
    int a[] = {1,2,3,4,5};
    ini = [&](int pt){return a[pt];};
    build(5);
    cout << st.get(0,4) << endl;
    return 0;
}
```

3.4 Segment tree Lazy

```
// This is set up for range minimum queries, but can be↩
     easily adapted for computing other quantities.
// To enable lazy propagation and range updates, ↩
     uncomment the following line.
// #define LAZY
struct Segtree {
    int n;
    vector<int> data;
#ifdef LAZY
#define NOLAZY 2e9
#define GET(node) (lazy[node] == NOLAZY ? data[node] : ↩
     lazy[node])
    vector<int> lazy;
#else
#define GET(node) data[node]
#endif
    void build_rec(int node, int* begin, int* end) {
        if (end == begin+1) {
            if (data.size() <= node) data.resize(node↩
                 +1);
            data[node] = *begin;
        } else {
            int* mid = begin + (end-begin+1)/2;
            build_rec(2*node+1, begin, mid);
            build_rec(2*node+2, mid, end);
            data[node] = min(data[2*node+1], data[2*↩
                 node+2]);
        }
    }
#ifndef LAZY
    void update_rec(int node, int begin, int end, int ↩
         pos, int val) {
        if (end == begin+1) {
            data[node] = val;
        } else {
            int mid = begin + (end-begin+1)/2;
            if (pos < mid) {
                update_rec(2*node+1, begin, mid, pos, ↩
                     val);
            } else {
                update_rec(2*node+2, mid, end, pos, val↩
                     );
            }
            data[node] = min(data[2*node+1], data[2*↩
                 node+2]);
        }
    }
#else
    void update_range_rec(int node, int tbegin, int ↩
         tend, int abegin, int aend, int val) {
        if (tbegin >= abegin && tend <= aend) {
            lazy[node] = val;
        } else {
            int mid = tbegin + (tend - tbegin + 1)/2;
            if (lazy[node] != NOLAZY) {
                lazy[2*node+1] = lazy[2*node+2] = lazy[↩
                     node]; lazy[node] = NOLAZY;
            }
            if (mid > abegin && tbegin < aend)
                update_range_rec(2*node+1, tbegin, mid,↩
                     abegin, aend, val);
            if (tend > abegin && mid < aend)
                update_range_rec(2*node+2, mid, tend, ↩
                     abegin, aend, val);
            data[node] = min(GET(2*node+1), GET(2*node↩
                 +2));
        }
```

```
    }
#endif
    int query_rec(int node, int tbegin, int tend, int ↩
        abegin, int aend) {
        if (tbegin >= abegin && tend <= aend) {
            return GET(node);
        } else {
#ifdef LAZY
            if (lazy[node] != NOLAZY) {
                data[node] = lazy[2*node+1] = lazy[2*↩
                    node+2] = lazy[node]; lazy[node] =↩
                     NOLAZY;
            }
#endif
            int mid = tbegin + (tend - tbegin + 1)/2;
            int res = INT_MAX;
            if (mid > abegin && tbegin < aend)
                res = min(res, query_rec(2*node+1, ↩
                    tbegin, mid, abegin, aend));
            if (tend > abegin && mid < aend)
                res = min(res, query_rec(2*node+2, mid,↩
                     tend, abegin, aend));
            return res;
        }
    }

    // Create a segtree which stores the range [begin, ↩
        end) in its bottommost level.
    Segtree(int* begin, int* end): n(end - begin) {
        build_rec(0, begin, end);
#ifdef LAZY
        lazy.assign(data.size(), NOLAZY);
#endif
    }

#ifndef LAZY
    // Call this to update a value (indices are 0-based↩
        ). If lazy propagation is enabled, use ↩
        update_range(pos, pos+1, val) instead.
    void update(int pos, int val) {
        update_rec(0, 0, n, pos, val);
    }
#else
    // Call this to update range [begin, end), if lazy ↩
        propagation is enabled. Indices are 0-based.
    void update_range(int begin, int end, int val) {
        update_range_rec(0, 0, n, begin, end, val);
    }
#endif
    // Returns minimum in range [begin, end). Indices ↩
        are 0-based.
    int query(int begin, int end) {
        return query_rec(0, 0, n, begin, end);
    }
};
```

3.5 Range Minimum Query (RMQ)

```
#include "../template.cpp"

// Range minimum query
// <O(NlogN),O(1)>
// preprocess=space, query

#define MAX 10000
#define lMAX 20 // log2(MAX)

ll A[MAX]; // input array
ll M[MAX][lMAX]; // rmq struct
ll N;

void buildRMQ(){
    F(N) M[i][0]=i;
    ll lN=(ll)log2(N);
    FOR(j,1,lN+1)
        for(ll i=0;i+(1<<j)-1<N; i++)
            if(A[M[i][j-1]] < A[M[i+(1<<(j-1))][j-1]])
                M[i][j]=M[i][j-1];
            else M[i][j]=M[i+(1<<(j-1))][j-1];
}

ll queryRMQ(ll i,ll j) {
    ll k = (ll)log2(j-i+1);
    if(A[M[i][k]] < A[M[j-(1<<k)+1][k]])
         return M[i][k];
    else return M[j-(1<<k)+1][k];
}

ll cnt=0;
vll ng[MAX];
ll id[MAX]; // id of vertex in rmq array
ll ind[MAX]; // id of rmq pos of vertex

void addItem(ll u, ll d) {
    id[cnt]=u;
    ind[u]=cnt;
    A[cnt++]=d;
}
void dfs(ll u,ll d, ll par) {
    addItem(u,d);
    for(auto i:ng[u]) {
        if(i==par)continue;
        dfs(i,d+1,u);
        addItem(u,d);
    }
}
void buildLCA(){
    dfs(0,0,-1);
    N=cnt;
    buildRMQ();
}

ll queryLCA(ll i, ll j){
    ll a=ind[i], b=ind[j]; if(a>b)swap(a,b);
    return id[queryRMQ(a,b)];
}

void addEdge(ll i, ll j) { ng[i].pb(j); ng[j].pb(i); }

#ifndef NOMAIN
int main(void) {
    ios_base::sync_with_stdio(false);
    vll a={5,2,1,6,3,3,2,6,8};
    copy(a.begin(),a.end(),A);
    N=a.size();
    buildRMQ();
    ll lN=(ll)log2(N);
    F(N){FF(lN+1) cout<<A[M[i][j]]<<' '; cout<<endl;}
    F(N)FOR(j,i,N)cout<<"RMQ "<<i<<","<<j<<": "<<A[↩
        queryRMQ(i,j)]<<endl;
    return 0;
}
#endif
```

3.6 Lowest Common Ancestor (LCA)

```
#define NOMAIN 1
#include "rmq.cpp"

// Lowest common ancestor
// <O(NlogN),O(1)> where N=|E|
//
// using from RMQ:
// MAX, ll N, ll A[MAX];
// MAX=2*maximum number of edges

ll cnt=0;
vll ng[MAX];
ll id[MAX]; // id of vertex in rmq array
ll ind[MAX]; // id of rmq pos of vertex

void addItem(ll u, ll d) {
    id[cnt]=u;
    ind[u]=cnt;
    A[cnt++]=d;
}
void dfs(ll u,ll d, ll par) {
    addItem(u,d);
    for(auto i:ng[u]) {
        if(i==par)continue;
        dfs(i,d+1,u);
        addItem(u,d);
    }
}

void buildLCA(){
    dfs(0,0,-1);
    N=cnt;
    buildRMQ();
}

ll queryLCA(ll i, ll j){
    ll a=ind[i], b=ind[j]; if(a>b)swap(a,b);
    return id[queryRMQ(a,b)];
}
```

```
void addEdge(ll i, ll j) { ng[i].pb(j); ng[j].pb(i); }

int main() {
    addEdge(0,1);
    addEdge(1,2);
    addEdge(1,3);
    addEdge(0,4);
    addEdge(4,5);
    addEdge(5,6);
    addEdge(4,7);
    buildLCA();
    cout<<"LCA(1,4): "<<queryLCA(1,4)<<endl;
    cout<<"LCA(6,4): "<<queryLCA(6,4)<<endl;
    cout<<"LCA(7,6): "<<queryLCA(7,6)<<endl;
    cout<<"LCA(2,3): "<<queryLCA(2,3)<<endl;
    return 0;
}
```

3.7 LCA iterative

```
#include "../template.cpp"
#define MAX 100107
#define lMAX 20

// Lowest common ancestor - iterative version
// Doesn't recurse - use when stack would overflow.

// <O(NlogN),O(1)> where N=|E|
// usage: clear, n=X, addEdge, build, query, query, query, ...

ll n;
vll ng[MAX];
ll par[MAX];
ll dp[MAX][lMAX];
ll lev[MAX];

void build(ll root) {
    CL(lev,-1);
    queue<pair<ll,pll>> q;
    q.push({root,{-1,1}});
    while(q.size()) {
        auto act=q.front();q.pop();
        lev[act.x]=act.y.x;
        par[act.x]=act.y.y;
        for(auto i: ng[act.x]) {
            if(lev[i]!=-1) continue;
            q.push({i, {act.y.x+1, act.x}});
        }
    }
    CL(dp, -1);
    F(n) dp[i][0] = par[i];
    for (int j = 1; (1 << j) < n; j++) F(n) {
        if (dp[i][j-1] != -1) dp[i][j] = dp[dp[i][j-1]][j-1];
    }
}

int query(int p, int q)  {
    if (lev[p] < lev[q]) swap(p,q);
    ll log;
    for (log = 1; 1 << log <= lev[p]; log++);
    log--;
    for(ll i=log; i>=0; i--) if (lev[p] - (1 << i) >= lev[q]) {
        p = dp[p][i];
    }
    if (p==q)return p;
    for(ll i=log; i>=0; i--) if (dp[p][i] != -1 && dp[p][i] != dp[q][i]) {
        p = dp[p][i];
        q = dp[q][i];
    }
    return par[p];
}

void clear() {
    F(MAX) ng[i].clear();
    CL(par,-1);
}

void addEdge(ll i, ll j) {
    if(i>j) swap(i,j);
    ng[i].pb(j); ng[j].pb(i);
}

int main(void) {
    ios_base::sync_with_stdio(false);
    n=7;
    addEdge(0,1); addEdge(1,2); addEdge(1,3); addEdge(3,4); addEdge(4,5);  addEdge(3,6);
    build(0);
    cout<<query(5,6)<<endl;
    cout<<query(2,6)<<endl;
    cout<<query(0,0)<<endl;
    return 0;
}
```

3.8 Tarjan's offline LCA

```
#include "../template.cpp"

// Tarjan's offline lowest common ancestor (LCA) algorithm
// =====
// Answers offline 'm' LCA queries on tree of 'n' vertices
// O(n+m * ackr(n))

// Usage: 1) init with queries, 2) add edges, 3) run TarjanOLCA(root, -1)
// answers in order of 'queries' array appears in 'ret' array

#define MX >>#nodes<<
vll ng[MX];
vpll q[MX];
bool col[MX];
ll anc[MX];
vll ret;

// Union Find:
ll uf[MX];
int find(int e) { return uf[e] == e ? e : uf[e] = find(uf[e]); }
void mrg(int i, int j) { uf[find(i)] = find(j); }

 vll TarjanOLCA(ll u, ll par) {
    uf[u]=u;
    anc[u]=u;
    for (auto v: ng[u]) {
        if(v==par)continue;
         TarjanOLCA(v,u);
         mrg(u,v);
         anc[find(u)]=u;
     }
     col[u]=1;
     for(auto v: q[u]) {
         if (col[v.x])
             ret[v.y] = anc[find(v.x)];
         }
}

void init(vpll & qrs) {
    F(MX) ng[i].clear();
    CL(col,0);
    ret=vll(qrs.size());
    F(qrs.size()) q[qrs[i].x].pb({qrs[i].y,i}),q[qrs[i].y].pb({qrs[i].x,i});
}

void addEdge(ll a, ll b) {
    ng[a].pb(b); ng[b].pb(a);
}

int main() {
    vpll qrs = {{0, 1},{2,3},{5,6},{2,6},{0,0},{6,3}};
    init(qrs);
    addEdge(0,1); addEdge(1,2); addEdge(1,3); addEdge(3,4); addEdge(4,5);  addEdge(3,6);
    TarjanOLCA(0,-1);
    F(qrs.size()){
        cout<<"lca("<<qrs[i].x<<","<<qrs[i].y<<") = "<<ret[i]<<endl;
    }
    return 0;
}
```

3.9 Treap

```
#include "../template.cpp"
using namespace std;
```

```
// Treap data structure
// ========
// Set structure with logarithmic inserts and deletes.
// - Heap(min) order on priority, BST ordered on key
// Usage:
// - Fast and short SET and MAP (just add val propery ↩
    to node)
// - Fast implementation of array if merges are ↩
    required: key=index of elem
// - When priority is randomized hash of value, treap ↩
    provides unique binary tree representation of ↩
    content

// All complexities are O(log n) avg, O(n) worst case
// Unique keys only!
// Remember to srand!

#define MOD >>LONG_PRIME<<
using T=ll;

typedef struct _Node {
    _Node(T k) : key(k), prt(rand() % MOD), l(0), r(0) ↩
        {}
    T key;
    ll prt;
    _Node * l, * r;
} *Node;

bool find(T x, Node n) {
    if (!n)
        return 0;
    if (n->key == x)
        return 1;
    if (n->key > x)
        return find(x, n->l);
    return find(x, n->r);
}

// values of one have to be strictly smaller than the ↩
    other!!!
Node merge(Node l, Node r) {
    if (!l || !r)
        return l ? l : r;
    if (l->prt > r->prt) {
        l->r = merge(l->r, r);
        return l;
    }
    r->l = merge(l, r->l);
    return r;
}

void split(T x, Node n, Node& l, Node& r) {
    if (!n)
        l = r = 0;
    else if (x < n->key)
        split(x, n->l, l, n->l), r = n;
    else
        split(x, n->r, n->r, r), l = n;
}

void ins(Node x, Node& n) {
    if (!n)
        n = x;
    else if (x->prt > n->prt)
        split(x->key, n, x->l, x->r), n = x;
    else
        ins(x, x->key < n->key ? n->l : n->r);
}

void del(T x, Node& n) {
    if (n->key == x) {
        delete n;
        n = merge(n->l, n->r);
    } else {
        del(x, x < n->key ? n->l : n->r);
    }
}

void prt(Node n, vll& ret) {
    if (!n) return ;
    ret.pb(n->key);
    prt(n->l,ret);prt(n->r,ret);
}

// just for test:
void test(vll& nums) {
    Node tree = 0;
    for (auto i : nums) {
        ins(new _Node(i), tree);
    }
    for (auto i : nums) {
        assert(find(i, tree));
        del(i, tree);
        assert(!find(i, tree));
    }
}

int main() {
    srand(time(NULL));
    vll t1 = {10, 5, 12, 13, 14, 3, 7}, t2 = {1, 2, 3, ↩
        4, 5},
        t3 = {5, 4, 3, 2, 1}, t4;
    set<ll> s;
    REP(i, 1000) {
        ll x = rand() % MOD;
        while (s.count(x))
            x = rand() % MOD;
        s.insert(x);
        t4.pb(x);
    }
    test(t1);test(t2);test(t3);test(t4);
}
```

3.10 HLD

```
// SegmentTree with operations init,set,range modify,↩
    query
// Graph with vector<vector<int>>


template <class T, int V>
class SegmentTree {
  T seg[V];
public:
  T mrg(T a,T b) {
    return max(a,b);
  }
  T init(ll a) {
    return A[a];
  }
  void build(ll sz=V-1, ll st=0, ll en=V-1, ll p=1) {
    if (st == en)
      seg[p] = st < sz ? init(st) : T();
    else {
      build(sz,st,(st+en)/2,p*2);
      build(sz,(st+en)/2+1,en,p*2+1);
      seg[p] = mrg(seg[p*2],seg[p*2+1]);
    }
  }
  T get(ll st, ll en, ll sst=0,ll sen=V-1,ll p=1) {
    if (st > sen or en < sst) return T();
    if (st <= sst and en >= sen) return seg[p];
    ll mid = (sst+sen)/2;
    T lt = get(st,en,sst,mid,2*p);
    T rt = get(st,en,mid+1,sen,2*p+1);
    if (st <= mid and en > mid) return mrg(lt,rt);
    return st <= mid ? lt : rt;
  }
  T upd(ll pos,T val, ll sst=0,ll sen=V-1,ll p=1) {
    if (sst == sen) return seg[p] = val;
    ll mid = (sst+sen)/2;
    if (pos <= mid) return seg[p] = mrg(upd(pos,val,sst↩
        ,mid,2*p),seg[2*p+1]);
    else return seg[p] = mrg(seg[2*p],upd(pos,val,mid↩
        +1,sen,2*p+1));
  }
}

template <class T, int V>
class HeavyLight {
  int parent[V], heavy[V], depth[V];
  // heavy: root of biggest subtree
  int root[V], treePos[V];
  // root of current chain, position in current chain
  SegmentTree<T> tree;

  template <class G>
  // calculates parent[], depth[] and heavy[]
  int dfs(const G& graph, int v) {
    int size = 1, maxSubtree = 0;
    for (int u : graph[v]) if (u != parent[v]) {
      parent[u] = v;
      depth[u] = depth[v] + 1;
      int subtree = dfs(graph, u);
      if (subtree > maxSubtree) heavy[v] = u, ↩
          maxSubtree = subtree;
      size += subtree;
    }
    return size;
  }

  template <class BinaryOperation>
  void processPath(int u, int v, BinaryOperation op) {
```

```
    for (; root[u] != root[v]; v = parent[root[v]]) { ↩
        // iterate the heavier
      if (depth[root[u]] > depth[root[v]]) swap(u, v);
      op(treePos[root[v]], treePos[v] + 1);
    }
    if (depth[u] > depth[v]) swap(u, v);
    op(treePos[u], treePos[v] + 1);
  }

public:
  template <class G>
  void init(const G& graph) {
    int n = graph.size();
    fill_n(heavy, n, -1);
    parent[0] = -1;
    depth[0] = 0;
    dfs(graph, 0);
    for (int i = 0, currentPos = 0; i < n; ++i)
      if (parent[i] == -1 || heavy[parent[i]] != i) // ↩
          iterate chains
        for (int j = i; j != -1; j = heavy[j]) {
          root[j] = i;
          treePos[j] = currentPos++;
        }
    tree.init(n);
  }

  void set(int v, const T& value) {
    tree.set(treePos[v], value);
  }

  // void modifyPath(int u, int v, const T& value) {
  //   processPath(u, v, [this, &value](int l, int r) {↩
       tree.modify(l, r, value); });
  // }

  T queryPath(int u, int v) {
    T res = T();
    processPath(u, v, [this, &res](int l, int r) { res.↩
        add(tree.query(l, r)); });
    return res;
  }
};
```

3.11 HLD2

```
// working hld with segtree
vector<int> adj[N];
vector<int> edges[N];
vector<int> idx[N];

int subSize[N];
int depth[N];

int lca[LN][N];

int segTree[N<<2];
int pos;
int chainNo;
int chainHead[N];
int chainIndex[N];
int arr[N];
int basePos[N];
int endNode[N];

void Dfs(int node, int parent, int level) {
    depth[node] = level;
    lca[0][node] = parent;
    subSize[node] = 1;
    int x = adj[node].size();
    while (x--) {
        int next = adj[node][x];
        if (next != parent){
            endNode[idx[node][x]] = next;
            Dfs(next, node, level+1);
            subSize[node] += subSize[next];
        }
    }
}

void HLD(int node, int cost, int parent) {
    if (chainHead[chainNo] == -1) {
        chainHead[chainNo] = node;
    }
    pos++;
    chainIndex[node] = chainNo;
    basePos[node] = pos;
    arr[pos] = cost;
    int specialChild = -1, edgeCost = 0;
    int x = adj[node].size();
    while (x--) {
        int next = adj[node][x];
        if (next != parent) {
            if (specialChild == -1 || subSize[next] > ↩
                 subSize[specialChild]) {
                specialChild = next;
                edgeCost = edges[node][x];
            }
        }
    }
    if (specialChild != -1) {
        HLD(specialChild, edgeCost, node);
    }
    x = adj[node].size();
    while (x--) {
        int next = adj[node][x];
        if (next != parent && next != specialChild) {
            chainNo++;
            HLD(next, edges[node][x], node);
        }
    }
}

void initializeLCA(int n) {
    for (int j = 1; j < LN; j++) {
        for (int i = 1; i <= n; i++) {
            lca[j][i] = lca[j - 1][lca[j - 1][i]];
        }
    }
}

int LCA(int x, int y) {
    if (depth[x] < depth[y]) {
        std::swap(x, y);
    }
    for (int i = LN - 1; i >= 0; i--) {
        if (depth[x] - (1 << i) >= depth[y]) {
            x = lca[i][x];
        }
    }
    if (x == y) {
        return x;
    }
    for (int i = LN - 1; i >= 0; i--) {
        if (lca[i][x] != lca[i][y]) {
            x = lca[i][x];
            y = lca[i][y];
        }
    }
    return lca[0][x];
}

void buildSegTree(int node, int u, int v) {
    if(u == v) {
        segTree[node] = arr[u];
        return ;
    }
    int mid = (u + v) >> 1;
    int lc = node << 1;
    int rc = lc | 1;
    buildSegTree (lc, u, mid);
    buildSegTree (rc, mid + 1, v);
    segTree[node] = std::max(segTree[lc], segTree[rc]);
}

void updateSegTree(int node, int u, int v, int i, int ↩
     val) {
    if(u == v) {
        segTree[node] = val;
        return;
    }
    int mid = (u + v) >> 1;
    int lc = node << 1;
    int rc= lc | 1;
    if(i <= mid) {
        updateSegTree(lc, u, mid, i, val);
    } else {
        updateSegTree(rc, mid+  1, v, i, val);
    }
    segTree[node]=std::max(segTree[lc], segTree[rc]);
}

int querySegTree(int node, int u, int v, int ql, int qr↩
    ) {
    if(ql > v || u > qr) {
        return 0;
    }
    if(u >= ql && v <= qr) {
        return segTree[node];
    }
    int mid = (u + v) >> 1;
    int lc = node << 1;
    int rc = lc | 1;
```

```
    int lv = querySegTree(lc, u, mid, ql, qr);
    int rv = querySegTree(rc, mid + 1, v, ql, qr);
    return std::max(rv,lv);
}

int queryUp(int u,int v) {
    if(u == v) {
        return 0;
    }
    int lchain, rchain = chainIndex[v], ans = -1;
    while (1) {
        lchain = chainIndex[u];
        if(lchain == rchain) {
            if(u == v) {
                break;
            }
            int currAns = querySegTree(1, 1, pos, ↩
             basePos[v] + 1, basePos[u]);
            ans = std::max(ans, currAns);
            break;
        }
        int currAns = querySegTree(1, 1, pos, basePos[↩
         chainHead[lchain]], basePos[u]);
        ans = std::max(ans, currAns);
        u = chainHead[lchain];
        u = lca[0][u];
    }
    return ans;
}

void Initialize(int n) {
    for (int i = 0; i <= n; i++) {
        adj[i].clear();
        edges[i].clear();
        idx[i].clear();
        chainHead[i] = -1;
        for (int j = 0; j < LN; j++) {
            lca[j][i]=-1;
        }
    }
}

int queryPath(int u, int v) {
    int lca = LCA(u, v);
    int a = queryUp(u, lca);
    int b = queryUp(v, lca);
    return std::max(a, b);
}

void Update(int i, int val) {
    int node = endNode[i];
    updateSegTree(1, 1, pos, basePos[node], val);
}

int main() {
    int t;
    SI(t);
    while (t--) {
        int n;
        SI(n);
        Initialize(n);
        for (int i=1;i<n;i++) {
            int u, v, w;
            SI(u), SI(v), SI(w);
            adj[u].pb(v);
            edges[u].pb(w);
            idx[u].pb(i);
            adj[v].pb(u);
            edges[v].pb(w);
            idx[v].pb(i);
        }
        Dfs(1, 0, 0);
        initializeLCA(n);
        pos = -1;
        chainNo = 1;
        HLD(1, 0, 0);
        buildSegTree(1, 1,pos);
        char str[10];
        scanf("%s", str);
        while (str[0] != 'D') {
            int type, u ,v;
            SI(u);
            SI(v);
            if (str[0] == 'Q') {
                PI(queryPath(u, v));
                printf("\n");
            } else {
                Update(u, v);
            }
            scanf("%s", str);
        }
    }
}
```

4 Math

4.1 List divisors

```
#include "../template.cpp"

// Finds all divisors of n
// in O(sqrt(n))

vll divisors(ll n){
    vll a,b;
    for (int i = 1; i*i <= n; ++i) {
        if(n % i == 0){
            a.pb(i);
            b.pb(n/i);
        }
    }
    if(b.back()==a.back())b.pop_back();
    reverse(b.begin(),b.end());
    a.insert(a.end(),b.begin(),b.end());
    return a;// list of sorted divisors
}

int main() {
    vll d=divisors(28);
    for(auto i:d)cout<<i<<",";
    cout<<endl;
    return 0;
}
```

4.2 Modulo

```
// Modulo as defined mathematically
ll mod(ll a, ll b) {
  return ((a%b)+b)%b;
}
```

4.3 Modular inverse

```
// Computes b such that ab = 1 (mod n), returns -1 on ↩
    failure
ll mod_inverse(ll a, ll n) {
  ll x, y;
  ll d = extended_euclid(a, n, x, y);
  if (d > 1) return -1;
  return mod(x,n);
}

// using power
ll pw(ll n,ll k){
    ll r(1);
    while(k){
        if(k&1)r*=n,r%=MOD;
        n*=n,n%=MOD,k>>=1;
    }
    return r;
}
```

4.4 Sieve of Eratosthenes

```
#define MX 10000
bitset<MX> era;
ll sp[MX]; // for prime factorization (can be ommited)
           // also in init function
vll pri;
void init() {
    era.set(); // all ones
    era[0] = era[1] = 0;
    for(int i = 2; i < MX; i += i > 2 : 2 : 1) {
        if (!era[i]) continue;
```

```
        pri.pb(i);
        sp[i] = i;
        for (int m = 2; i*m < MX; m++) {
            if (era[i*m]) sp[i*m]=i;
            era[i*m] = 0;
        }
    }
}
```

4.5 Euclidean algorithm

4.5.1 Euclidean algorithm (GCD)

```
// Finds gcd of numbers
// When not needed extended,
// use __gcd(a,b); but carefull -> exception when 0
ll gcd(ll a, ll b) {
  ll tmp;
  while(b){a%=b; tmp=a; a=b; b=tmp;}
  return a;
}

ll lcm(ll a, ll b) {
    return a/gcd(a,b)*b;
}
```

4.5.2 Extended Euclidean algorithm (EGCD)

```
// Returns
// [ d = gcd(a,b);
//   + finds x,y such that d = ax + by ]

ll extended_euclid(ll a,ll b,ll &x,ll &y) {
    if (a == 0) { x = 0; y = 1; return b;}
    ll x1, y1; ll d = extended_euclid(b%a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}
```

4.6 Pascal's triangle

```
#include "../template.cpp"

#define MX 1001
ll PT[MX][MX];
void pt()
{
  F(MX)
  {
    PT[i][0] = PT[i][i] = 1;
    FOR(k, 1, i) PT[i][k] = PT[i-1][k] + PT[i-1][k-1];
  }
}
```

4.7 Powers

4.7.1 Power of number

```
// Power of number
// in log time

ll power(ll a, ll p) {
    ll res = 1, x = a;
    while(p) {
        if (p & 1) res = (res * x);
        x = (x * x); p >>= 1;
    }
    return res;
}
```

4.7.2 Modular power

```
// Returns a^p mod m
ll modular_power( ll a, ll p, ll m ) {
    ll res = 1 % m, x = a % m;
    while ( p ) {
        if ( p & 1 ) res = ( res * x ) % m;
        x = ( x * x ) % m; p >>= 1;
    }
    return res;
}
```

```
#include "../template.cpp"

ll pw(ll n, ll k)
{
  ll r = 1;
  while (k)
  {
    if (k&1) r*=n;
    n*=n;k>>=1;
  }
  return r;
}

#define MD (1000000007)
#define MX (1<<19)
ll pwmod(ll n, ll k)
{
  ll r = 1;
  while (k)
  {
    if (k&1) r*=n, r%=MD;
    n*=n, n%=MD;k>>=1;
  }
  return r;
}

ll inv(ll a) { return pwmod(a, MD-2); }
ll IN[MX] = {1}, FA[MX] = {1};
void comp() {FOR(i, 1, MX) IN[i] = inv(FA[i]=FA[i-1]*i%↩
    MD); }
ll comb(ll n, ll k)
{
  return n < k ? 0 : (FA[n]*IN[k]%MD)*IN[n-k]%MD;
}
```

4.7.3 Matrix power

```
struct mt : vector<vector<ll>> {
    using vector::vector;
    int w() const { return at(0).size(); }
    int h() const { return size(); }
    mt(int h,int w) : vector<vector<ll>>(h,vector<ll>(w↩
       )) {};
    mt operator*(const mt ot) const {
        mt res(h(),ot.w());
        REP(i,h()) REP(j,ot.w()) REP(k,w()) {
            res[i][j] += (*this)[i][k]*ot[k][j];
            res[i][j] %= mod;
        }
        return res;
    }
};

mt pw(mt a, int n) {
    mt x(a.h(),a.w());
    F(min(a.h(),a.w())) x[i][i] = 1;
    while (n) {
        if (n & 1) {
            x = x * a;
        }
        a = a * a;
        n >>= 1;
    }
    return x;
}
```

4.8 Equations

4.8.1 Gauss elimination method – GEM

```
using mt = vector<vector<double>>;
// gauss elimination O(n^3) with partial pivoting
// returns a matrix in upper triangular form with
// Usage:
//    mt A = { {1,2,3,4},{1,0,1,0},{5,3,2,4},{6,1,4,6} };
//    mt b = { {1,2},{4,3},{5,6},{8,7} };
//    mt I(4,vector<double>(4));
//    F(4) I[i][i] = 1;
//    gem(merge(A,I),4);  // inverse is in last four columns of result
//    gem(merge(A,b),4); // solutions are as last two columns of result

// Helper function to concatenate two matrixes side by side
mt merge(mt a, mt b) {
    mt res(a.size());
    F(a.size()) {
        res[i].insert(res[i].end(),all(a[i]));
        res[i].insert(res[i].end(),all(b[i]));
    }
    return res;
}

// output fields
double det = 1;
int rnk = -1;
vll ord;
mt gem(mt m, int cols) {
    int h = m.size();
    int w = m[0].size();
    int nncs = cols;
    ord.resize(cols);
    iota(all(ord),0);
    F(min(h,nncs)) {
        pair<double,int> mx = {abs(m[i][i]),i};
        // find pivot
        FOR(j,i,h) mx = max(mx,{abs(m[j][i]),j});
        // null column
        if (mx.x < EPS) {
            FF(w) swap(m[nncs-1][j],m[i][j]);
            swap(ord[i],ord[mx.y]);
            i--;
            nncs--;
            continue;
        }
        det *= ((mx.y - i) % 2 ? -1 : 1); // swapping rows affects determinant
        // swap rows
        swap(m[i],m[mx.y]);

        FOR(j,i+1,h) {
            double val = -m[j][i]/m[i][i];
            FOR(k,i,w) {
                m[j][k] = m[j][k] + val*m[i][k];
            }
        }
    }
    rnk = min(h,nncs);

    F(h) {
         det *= m[i][i];
    }

    // Stop here for just Gauss Jordan elimination
    // Upper triangular form
    // return m;

    // Backpropagation
    // Full gauss elimination
    // ones on diagonal
    for(int i=rnk-1; i>= 0; i--) {
        double val = m[i][i];
        FOR(j,i,w) {
            m[i][j] /= val;
        }
    }
    for(int i=rnk-1; i>0; i--) {
        for(int j=i-1; j>=0;j--) {
            double val = -m[j][i];
            FOR(k,i,w) {
                m[j][k] = m[j][k] + val*m[i][k];
            }
        }
    }
    // upper left diagonal form (diagonal if matrix has full rank)
    return m;
}
```

4.8.2 Linear diophantine (ax+by=c)

```
#include "gcd.cpp";
#include "mod_inverse.cpp";

// Computes x and y such that
// [ ax + by = c; ]
// ...on failure, x = y = -1

void linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
  ll d = gcd(a,b);
  if (c%d) {
    x = y = -1;
  } else {
    x = c/d * mod_inverse(a/d, b/d);
    y = (c-a*x)/b;
  }
}
```

4.8.3 Modular linear equation (ax=b mod m)

```
// Finds all solutions to
// [ ax = b (mod n) ]
#include "mod.cpp"
#include "extended_euclid.cpp"

// Finds all solutions to
// [ ax = b (mod n) ]]
vll modular_linear_equation_solver(ll a, ll b, ll n) {
  ll x, y;
  vll solutions;
  ll d = extended_euclid(a, n, x, y);
  if (!(b%d)) {
    x = mod(x*(b/d), n);
    for (ll i = 0; i < d; i++)
      solutions.pb(mod(x + i*(n/d), n));
  }
  return solutions;
}
```

4.8.4 Recurrent equations

```
#include "../template.cpp"

#define MM 2
void mul(ll A[MM][MM], ll B[MM][MM], ll R[MM][MM], ll W, ll MD)
{
  F(W) FF(W) R[i][j] = 0;
  F(W) FF(W) FFF(W) R[i][j] += A[i][k]*B[k][j],R[i][j]%=MD;
}

void pw(ll M[MM][MM], ll R[MM][MM], ll W, ll k, ll MD)
{
  static ll E[MM][MM], H[MM][MM];
  F(W) FF(W) R[i][j] = E[i][j] = i == j;
  while (k)
  {
    if (k & 1)mul(E,M,R,W,MD), memcpy(E,R,sizeof(E));
    mul(M,M,H,W,MD);
    memcpy(M, H, sizeof(H));
    k >>= 1;
  }
}

// T(n) = a * T(n - 1)+ b * T(n - 2) + T(0) = Z, T(1) = C
ll MA[2][2], MR[2][2];
ll bar(ll a, ll b, ll Z, ll C, ll n, ll MD = 1)
{
  if (n < 2) return n ? C : Z;
  MA[0][0] = a;
  MA[0][1] = b;
  MA[1][0] = 1;
```

```
  MA[1][1] = 0;
  pw(MA, MR, 2, n-1, MD);
  return MR[0][0] * C + MR[0][1] * Z;
}
```

4.8.5 Recurrent equations 2

```
/*
    Calculates nth member of linear recurrence
    f(i) = c1 * f(i - 1) +  ... + ck * f(i - k)
    ----
    creates matrix:
       | 0 1 0 0 ... 0 |   | f(n-K)   |
       | 0 0 1 0 ... 0 |   | f(n-K+1) |
f(n) = | 0 0 0 1 ... 0 | * | f(n-K+3) |
       | ...           |   | ...      |
       | ck  ...    c1 |   | f(n-1)   |
    and caltulates n-k th power by square & multiply
    and multiplies it with init vector
*/

#define REP(i,b) for(int i=0;i<=b;i++)
#define REP2(i,a,b) for(int i=a;i<=b;i++)

typedef vector<vector<int> > TMatrix;
int K; // level of recurrence
int n0 = 1; // first init val is n0-th member of ↩
    recurrence

// computes A * B
TMatrix mul(TMatrix A, TMatrix B)
{
    TMatrix C(K, vector<int>(K));
    REP(i, K-1) REP(j, K-1) REP(k, K-1)
        C[i][j] = (C[i][j] + A[i][k] * B[k][j]) % MOD;
    return C;
}

// computes A ^ p
TMatrix pow(TMatrix A, int p)
{
    if (p == 1)
        return A;
    if (p % 2)
        return mul(A, pow(A, p-1));
    TMatrix X = pow(A, p/2);
    return mul(X, X);
}

int nth(vector<int> c, vector<int> init,int n) {
    K = c.size();
    TMatrix T(K, vector<int>(K));
    REP(i, K-2) {
        REP(j,K-1) {
            if (i == j-1)
                T[i][j] = 1;
            else
                T[i][j] = 0;
        }
    }
    REP(j,K-1) T[K-1][j] = c[j]; // coeficients
    T = pow(T, n-K+n0);
    int res = 0;
    REP(i, K-1)
        res = (res + T[1][i] * init[i]) % MOD;
    return res;
}

int main(int argc, char const *argv[])
{
    // 6th fib:
    cout << nth({1,1}, {1,1}, 6) << endl;
    // 6th f(i) = 2*f(i-1) + f(i-3):
    cout << nth({1,0,2}, {0,1,2}, 6) << endl;
    return 0;
}
```

4.9 Factorization

```
// Prime factorization
// 3 simmilar functions, they differ in output type and↩
    speed
// fact (no dependency) list of prime factors O(sqrt(N)↩
   )
// sfact (erasthotenes computed to MX) count of each ↩
   prime factor
//          O(sqrt(N)/log(N)) + log(MX))
// ffact (erasthotenes computed to MX)
//          list of prime factors O(log(N))
vll fact(ll n) {
    assert(n>1);
    vll res;
    ll f = 2;
    while(f*f <= n) {
        if ((n % f) == 0) {
            res.pb(f);
            n /= f;
        }
        else f++;
    }
    if (n > 1) res.pb(n);
    return res;
}

vll sfact(ll n) {
    assert(n>1);
    vll res(1);
    int i = 0;
    while(pri[i]*pri[i] <= n) {
        if ((n % pri[i]) == 0) {
            res.back()++;
            n /= pri[i];
        }
        else {
            res.pb(0);
            i++;
        }
    }
    if (n > 1) {
        int ind = lower_bound(all(pri),n) - begin(pri);
        int zb = ind - res.size() + 1;
        F(zb) res.pb(0);
        res.back()++;
    }
    return res;
}

vll ffact(ll n) {
    assert(n>1);
    vll res;
    while(n > 1) {
        res.pb(sp[n]);
        n /= sp[n];
    }
    return res;
}
```

4.10 Chinese reminder theorem

```
#include "extended_euclid.cpp"

// Chinese remainder theorem (special case): find z ↩
    such that
// z % x = a, z % y = b.  Here, z is unique modulo M = ↩
    lcm(x,y).
// Return (z,M).  On failure, M = -1.
pll chinese_remainder_theorem(ll x, ll a, ll y, ll b) {
  ll s, t;
  ll d = extended_euclid(x, y, s, t);
  if (a%d != b%d) return {0, -1};
  return {mod(s*b*x+t*a*y,x*y)/d, x*y/d};
}

// Chinese remainder theorem: find z such that
//  z % x[i] = a[i] for all i.
// Note that the solution is
//  unique modulo M = lcm_i (x[i]).
// Return (z,M).  On failure, M = -1.
//  Note that we do not require the a[i]'s to be ↩
    relatively prime.
pll chinese_remainder_theorem(const vll &x, const vll &↩
    a) {
  pll ret = make_pair(a[0], x[0]);
  for (ll i = 1; i < x.size(); i++) {
    ret = chinese_remainder_theorem(ret.first, ret.↩
        second, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}
```

4.11 Big numbers in Java

```
// example of use of BigInteger in Java
import java.io.*;
import java.util.*;
import java.math.*;
class Main{
    public static void main (String args[]){
        int upper = 10000000;
        Boolean[] sieve = new Boolean[upper];
        sieve[0] = sieve[1] = false;
        for(int i = 2; i < upper; ++i) {
            sieve[i] = true;
        }
        ArrayList<BigInteger> primes = new ArrayList<BigInteger>();
        for(int i = 2; i < upper; ++i) {
            if(sieve[i]) {
                BigInteger prime = BigInteger.valueOf(i);
                primes.add(prime);
                BigInteger tmp = prime.multiply(BigInteger.valueOf(2));
                while(tmp.compareTo(BigInteger.valueOf(upper)) < 0) {
                    sieve[tmp.intValue()] = false;
                    tmp = tmp.add(prime);
                }
            }
        }
        Scanner sc = new Scanner(System.in);
        while(sc.hasNextInt()) {
            int n = sc.nextInt();
            System.out.println(primes.get(n-1));
        }
    }
}
```

4.12 Big numbers

```
// Depending on your application it's pretty unlikely that you'll have to type out the entirety of this struct. For example, in most cases you won't need division, and you can leave out most of the operators too.
// NB: Thse are fairly terrible implementations. Multiplication is about twice as slow as Python, and division is about 20 times slower. Use only as a last resort when for some reason you can't use Java's native bignums.
struct bignum {
    typedef unsigned int uint;
    vector<uint> digits;
    static const uint RADIX = 1000000000;
    bignum(): digits(1, 0) {}
    bignum(const bignum& x): digits(x.digits) {}

    bignum(unsigned long long x) {
        *this = x;
    }

    bignum(const char* x) {
        *this = x;
    }

    bignum(const string& s) {
        *this = s;
    }

    bignum& operator=(const bignum& y) {
        digits = y.digits; return *this;
    }

    bignum& operator=(unsigned long long x) {
        digits.assign(1, x%RADIX);
        if (x >= RADIX) {
            digits.push_back(x/RADIX);
        }
        return *this;
    }
    bignum& operator=(const char* s) {
        int slen=strlen(s),i,l;
        digits.resize((slen+8)/9);
        for (l=0; slen>0; l++,slen-=9) {
            digits[l]=0;
            for (i=slen>9?slen-9:0; i<slen; i++) {
                digits[l]=10*digits[l]+s[i]-'0';
            }
        }
        while (digits.size() > 1 && !digits.back()) digits.pop_back();
        return *this;
    }

    bignum& operator=(const string& s) {
        return *this = s.c_str();
    }
    void add(const bignum& x) {
        int l = max(digits.size(), x.digits.size());
        digits.resize(l+1);
        for (int d=0, carry=0; d<=l; d++) {
            uint sum=carry;
            if (d<digits.size()) sum+=digits[d];
            if (d<x.digits.size()) sum+=x.digits[d];
            digits[d]=sum;
            if (digits[d]>=RADIX) {
                digits[d]-=RADIX; carry=1;
            } else {
                carry=0;
            }
        }
        if (!digits.back()) digits.pop_back();
    }
    void sub(const bignum& x) {
        // if ((*this)<x) throw; //negative numbers not yet supported
        for (int d=0, borrow=0; d<digits.size(); d++) {
            digits[d]-=borrow;
            if (d<x.digits.size()) digits[d]-=x.digits[d];
            if (digits[d]>>31) { digits[d]+=RADIX; borrow=1; } else borrow=0;
        }
        while (digits.size() > 1 && !digits.back()) digits.pop_back();
    }
    void mult(const bignum& x) {
        vector<uint> res(digits.size() + x.digits.size());
        unsigned long long y,z;
        for (int i=0; i<digits.size(); i++) {
            for (int j=0; j<x.digits.size(); j++) {
                unsigned long long y=digits[i]; y*=x.digits[j];
                unsigned long long z=y/RADIX;
                res[i+j+1]+=z; res[i+j]+=y-RADIX*z; //mod is slow
                if (res[i+j] >= RADIX) { res[i+j] -= RADIX; res[i+j+1]++; }
                for (int k = i+j+1; res[k] >= RADIX; res[k] -= RADIX, res[++k]++);
            }
        }
        digits = res;
        while (digits.size() > 1 && !digits.back()) digits.pop_back();
    }

    // returns the remainder
    bignum div(const bignum& x) {
        bignum dividend(*this);
        bignum divisor(x);
        fill(digits.begin(), digits.end(), 0);
        // shift divisor up
        int pwr = dividend.digits.size() - divisor.digits.size();
        if (pwr > 0) {
            divisor.digits.insert(divisor.digits.begin(), pwr, 0);
        }
        while (pwr >= 0) {
            if (dividend.digits.size() > divisor.digits.size()) {
                unsigned long long q = dividend.digits.back();
                q *= RADIX; q += dividend.digits[dividend.digits.size()-2];
                q /= 1+divisor.digits.back();
                dividend -= divisor*q; digits[pwr] = q;
                if (dividend >= divisor) { digits[pwr]++; dividend -= divisor; }
                assert(dividend.digits.size() <= divisor.digits.size()); continue;
            }
            while (dividend.digits.size() == divisor.digits.size()) {
                uint q = dividend.digits.back() / (1+divisor.digits.back());
                if (q == 0) break;
                digits[pwr] += q; dividend -= divisor*q;
```

```
        }
        if (dividend >= divisor) { dividend -= ↩
            divisor; digits[pwr]++; }
        pwr--; divisor.digits.erase(divisor.digits.↩
            begin());
    }
    while (digits.size() > 1 && !digits.back()) ↩
        digits.pop_back();
    return dividend;
}

string to_string() const {
    ostringstream oss;
    oss << digits.back();
    for (int i = digits.size() - 2; i >= 0; i--) {
        oss << setfill('0') << setw(9) << digits[i↩
            ];
    }
    return oss.str();
}
bignum operator+(const bignum& y) const {
    bignum res(*this); res.add(y); return res;
}

bignum operator-(const bignum& y) const {
    bignum res(*this); res.sub(y); return res;
}

bignum operator*(const bignum& y) const {
    bignum res(*this); res.mult(y); return res;
}

bignum operator/(const bignum& y) const {
    bignum res(*this); res.div(y); return res;
}

bignum operator%(const bignum& y) const {
    bignum res(*this); return res.div(y);
}
bignum& operator+=(const bignum& y) {
    add(y); return *this;
}
bignum& operator-=(const bignum& y) {
    sub(y); return *this;
}
bignum& operator*=(const bignum& y) {
    mult(y); return *this;
}

bignum& operator/=(const bignum& y) {
    div(y); return *this;
}

bignum& operator%=(const bignum& y) {
    *this = div(y);
}
bool operator==(const bignum& y) const {
    return digits == y.digits;
}
bool operator<(const bignum& y) const {
    if (digits.size() < y.digits.size()) return ↩
        true;
    if (digits.size() > y.digits.size()) return ↩
        false;
    for (int i = digits.size()-1; i >= 0; i--) {
        if (digits[i] < y.digits[i]) {
            return true;
        } else if (digits[i] > y.digits[i]) {
            return false;
        }
    }
    return false;
}
bool operator>(const bignum& y) const {
    return y<*this;
}
bool operator<=(const bignum& y) const {
    return !(y<*this);
}
bool operator>=(const bignum& y) const {
    return !(*this<y);
}
};
```

4.13 Catalan numbers

```
// Catalan numbers:

// 1) Count the number of expressions containing n ↩
    pairs of parentheses which are correctly matched. ↩
    For n = 3, possible expressions are ((())), ()(())↩
    , ()()(), (())(), (()()).
// 2) Count the number of possible Binary Search Trees ↩
    with n keys (See this)
// 3) Count the number of full binary trees (A rooted ↩
    binary tree is full if every vertex has either two↩
     children or no children) with n+1 leaves.

// The first few Catalan numbers for n = 0, 1, 2, 3, ↩
        are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, ↩


// A recursive function to find nth catalan number
ll catalan(ll n){
    if (n <= 1) return 1;
    // catalan(n) is sum of catalan(i)*catalan(n-i-1)
    ll res = 0;
    for (int i=0; i<n; i++)
        res += catalan(i)*catalan(n-i-1);
    return res;
}
```

4.14 Combinatinal numbers

```
ull choose(ull n, ull k) {
    if (k > n) return 0;
    ull r = 1;
    F(k){
        r *= n--;
        r /= i+1;
    }
    return r;
}
```

4.15 Combinatinal numbers modulo

```
ll mod = 100000000711;

const int X = 400007;

ll P[X], I[X];
ll comb(ll x, ll y, ll mod) {
    return P[x] * I[y] % mod * I[x - y] % mod;
}
void init(ll n, ll mod){
    P[0] = 1;
    F(n) P[i+1] = P[i]*(i+1) % mod;
    I[n-1] = modinv(P[n-1], mod);
    FF(n-1) {
        ll i=n-2-j;
        I[i] = I[i+1] * (i+1) % mod;
    }
}

int main () {
    init(30, mod);
    cout << comb(20, 10, mod) << endl;

    return 0;
}
```

4.16 Euler's totient function

```
// The totient function , also called Euler's totient ↩
    function, is defined as the number of positive ↩
    integers that are relatively prime to (i.e., do ↩
    not contain any factor in common with) , where 1 ↩
    is counted as being relatively prime to all ↩
    numbers.
// This code took less than 0.5s to calculate with MAX ↩
    = 10^7
#define MAX 10000000

int phi[MAX];
bool pr[MAX];
```

```
void totient(){
  for(int i = 0; i < MAX; i++){
    phi[i] = i;
    pr[i] = true;
  }
  for(int i = 2; i < MAX; i++)
    if(pr[i]){
      for(int j = i; j < MAX; j+=i){
        pr[j] = false;
        phi[j] = phi[j] - (phi[j] / i);
      }
      pr[i] = true;
    }
}
```

4.17 Fast polynom multiplication using DFFT

```
#include "../template.cpp"

// Algorithm for fast polynom multiplication using DFFT↩
    .
// For polynoms P,Q returs P*Q.
//
// N(logN) where N = max(|P|,|Q|})
//
// IN: polynom P, size of P pn, Q, qn, Return polynom R
// OUT: length of polynom R = P*Q


// use on:
// - multiplication of long numbers
// - convolution: constant difference -> constant sum ↩
    trick
//      reverse second polynom, multiplication will ↩
    give you
//      at index i overlapping multiplication after ↩
    shift by i
//      e.g.   0 1 2
//           3 2 1 0
//      convolution[2]=a[0]*b[2] + a[1]*b[1] + a[2]*b↩
    [0]
//      because 0+2 = 1+1 = 0+2 = 2

#define MX >>4*length of P,Q<<

typedef complex<double> cpx;
void dft(const cpx *s,cpx *r,ll n,const cpx &wn,cpx *h)↩
    {
    if(n==1){*r=*s;return;}
    ll N=n>>1,j(-2),k(-1);;
    cpx *os=h,*es=h+N,*oR=h+2*N,*eR=h+3*N;
    F(N)es[i]=s[j+=2],os[i]=s[k+=2];
    cpx w(1,0),t(wn*wn);
    dft(es,eR,N,t,h+4*N),dft(os,oR,N,t,h+4*N);
    F(N)t=w*oR[i],r[i]=eR[i]+t,r[i+N]=eR[i]-t,w=wn*w;
}
ll fft(ll*P,ll pn,ll*Q,ll qn,ll*R){
    static cpx ra[MX],b[MX],a[MX],rb[MX],tr[MX],wn,o[(↩
        MX)<<2];ll N(1),h(max(pn,qn)*2-1),H(pn+qn-1);
    while(N<=h)N<<=1;
    F(N)a[i]=b[i]=ra[i]=rb[i]=tr[i]={0,0};F(pn)a[i].↩
        real(P[i]);
    F(qn)b[i].real(Q[i]);
    wn={cos(2*M_PI/N),sin(2*M_PI/N)};
    dft(a,ra,N,wn,o),dft(b,rb,N,wn,o);F(N)tr[i]=ra[i]*↩
        rb[i];
    dft(tr,rb,N,pow(wn,-1),o);
    F(N)rb[i].real(rb[i].real()/N);F(H)R[i]=(ll)(rb[i].↩
        real()+0.5);
    return H;
}

ll pa[]={1,2,3}, pb[]={1,2};
ll r[6];
int main(void) {
    ios_base::sync_with_stdio(false);
    ll len=fft(pa,3,pb,2,r);
    F(len)cout<<r[i]<<' ';cout<<endl;
    // ret: 1 4 7 6
    // => 1+2x+3x^2 * 1+2x = 1+4x+7x^2+6x^3
    // also 321*21=6741
    return 0;
}
```

4.18 Horner's rule for evaluation of polynoms

```
#include "../template.cpp"

double horner(vector<double> v, double x) {
    double s = 0;
    for(ll i=v.size()-1; i>=0; i--) s=s*x+v[i];
    return s;
}

int main() {
    cout << horner({1,2,3}, 3.0) << endl;
    return 0;
}
```

5 Geometry

5.1 Global features

```
using ptt = double;
using pt = complex<ptt>;
#define x real()
#define y imag()
#define eq(x,y) (abs(x-(y)) <= EPS)

// no basic function use this handy defines but you can↩
      use them
pt I(0,1);
#define dot(a,b) (conj(a)*(b)).x // dot product
#define crs(a,b) (conj(a)*(b)).y // cross product
pt projp(pt p, pt a, pt b) { return a+dot(p-a,b-a)/conj↩
    (b-a); } // project point onto line (a,b)
pt reflep(pt p, pt a,pt b) { return a+conj((p-a)/(b-a))↩
    *(b-a); } // reflect point across line (a,b)
pt rotp(pt a, pt p, ptt ang) { return (a-p) * polar↩
    (1.0, ang) + p; } // rotate point around p ang ↩
    radians
#define sgn(x) ((x > -EPS) - (x < EPS)) // signum ↩
    function

/*
(defined) Dot product: (conj(a) * b).x

(defined) Cross product: (conj(a) * b).y

Squared distance: norm(a - b)

Relative stretch: (b / a).x

Euclidean distance: abs(a - b)

Angle of elevation: arg(b - a)

Slope of line (a, b): tan(arg(b - a))

Polar to cartesian: polar(r, theta)

Cartesian to polar: point(abs(p), arg(p))

Rotation about pivot p: (a-p) * polar(1.0, theta) + p

Angle ABC: abs(remainder(arg(a-b) - arg(c-b), 2.0 * ↩
    M_PI))

remainder normalizes the angle to be between [-PI, PI].↩
      Thus, we can get the positive non-reflex angle by↩
      taking its abs value.
*/

pt ins(pt a, pt b, pt p, pt q) { // intersection; lines↩
      are represented by start and endpoint
    auto c1 = (conj(p - a)*(b - a)).y, c2 = (conj(q - a↩
        )*(b - a)).y;
    return (c1 * q - c2 * p) / (c1 - c2); // undefined ↩
        if parallel
}

// Polygons

// dependency: cross product (crs)
pt area(vector<pt> pts) {
    ptt ar = 0;
    F(pts.size()-1) ar += crs(pts[i],pts[i+1]);
```

```
    return abs(ar)/2;
}

// Compute centroid
// dependency: area, crs
pt centroid(const vector<polt> &pol) {
  pt c(0,0);
  ptt sc = 6.0 * area(pol);
  F(pol.size()-1) {
    c = c + (pol[i]+pol[i+1])*(pol[i].x*pol[i+1].y - pol[i+1].x*pol[i].y);
  }
  return c / sc;
}

// cuts polygon pol along the line
// keeps that polygon that is on the left side of line
// dependency: cross product (crs) and line intersection (ins)
vector<pt> cut(vector<pt> pol, pt a, pt b) {
    vector<pt> lp;
    for (int i = 0; i < pol.size(); i++) {
        ptt l1 = crs(b-a, pol[i]-a), l2 = 0;
        if (i != pol.size()-1) l2 = crs(b-a, pol[i+1]-a);
        if (l1 > -EPS) lp.pb(pol[i]);
        if (l1*l2 < -EPS) lp.pb(ins(pol[i], pol[i+1], a, b));
    }
    if (!lp.empty() && !(lp.back() == lp[0]))
        lp.pb(lp[0]);
    return lp;
}

// check whether point is inside a polygon
// dependency: cross product (crs)
bool isin(pt p,vector<pt>& pol) {
    int wn=0;
    int n=pol.size()-1;
    for(int i=0;i<n;i++) {
        pt& p1=pol[i];
        pt& p2=pol[i+1];
        pt tmp = (p-p1)/(p2-p1);
        if(eq(tmp.y,0) and tmp.x >= -EPS and tmp.x <= 1 + EPS) return true; // on the edge
        auto k = crs(p2-p1,p-p1);
        auto d1= p1.y-p.y;
        auto d2= p2.y-p.y;
        if(k > EPS && d1 < EPS && d2 > EPS) wn++;
        if(k < -EPS && d2 < EPS && d1 > EPS) wn--;
    }
    return wn!=0;
}

// is polygon convex
// dependency: crs
bool is_conv(vector<pt> pol) {
    int sz = pol.size();
    if (sz <= 3) return 1; // degenerative (choose arbitrary)
    ptt pr = crs(pol[sz-1]-pol[0],pol[1]-pol[0]);
    F(sz-2) {
        ptt ac = crs(pol[i+1]-pol[i],pol[i+2]-pol[i]);
        if (ac*pr < -EPS) return 0;
        if (abs(ac) > pr) pr = ac;
    }
    return 1;
}

// projections onto line

// project point onto line segment
// dependency: projp
pt projpls(pt p,pt a,pt b) {
    ptt i = ((p-a)/(b-a)).x;
    return i < 0 + EPS ? a : i > 1 - EPS ? b : projp(p,a,b);
}

// add implicit ordering for complex numbers (beware that arg is slow as hell)

struct pt : complex<ptt> {
    using complex<ptt>::complex;
    bool operator<(const pt& o) const {
        return crs(*this,o) > EPS;
    }
};


//triangles
//2. A triangle with base b and height h has area A=0.5 b h .
//3. A triangle with three sides: a, b, c has perimeter p = a + b + c and semi-perimeter
//s = 0.5    p.
//4. A triangle with 3 sides: a, b, c and semi-perimeter s has area
//   A = sqrt(s    (s    a)    (s    b)    (s    c)). This formula is called the Heron s Formula.
//5. A triangle with area A and semi-perimeter s has an inscribed circle (incircle) with
//   radius r = A/s.

// quadrilaterals
//1. Quadrilateral or Quadrangle is a polygon with four edges (and four vertices). The term  polygon  itself is described in more details below (Section 7.3).
//Figure 7.7 shows a few examples of Quadrilateral objects.
//2. Rectangle is a polygon with four edges, four vertices, and four right angles.
//3. A rectangle with width w and height h has area A = w    h and perimeter p =
//2  (w+h).
//4. Square is a special case of a rectangle where w = h.
//5. Trapezium is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides have the same length, we have an Isosceles Trapezium.
//6. A trapezium with a pair of parallel edges of lengths w1 and w2; and a height h between both parallel edges has area A = 0.5    (w1 + w2)    h.
//7. Parallelogram is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.
//8. Kite is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is diagonal1    diagonal2/2.
//9. Rhombus is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.
```

5.2 Line equation

```
// line equation
struct ln {
    ptt a, b, c;
    ptt gx(ptt y) {
        return (b*y + c)/a;
    }
    ptt gy(ptt x) {
        return (a*x + c)/b;
    }
    bool dege() {
        return eq(a,0) or eq(b,0);
    }
};
#define dot(a,b) (conj(a)*(b)).x // dot product
#define crs(a,b) (conj(a)*(b)).y // cross product

//ln tolni(pt a, pt b) {
//    ln l;
//    if (eq(a.x,b.x)) {
//        l.a = 1; l.b = 0; l.c = -a.x;
//    }
//    else {
//        l.a = (b.y - a.y)/(a.x - b.x);
//        l.b = 1;
//        l.c = -l.a*a.x - a.y;
//    }
//    return l;
//}

// for precise calculation (integers)
ln toln(pt a, pt b) {
    ln l;
    l.a = (b.y - a.y);
    l.b = (a.x - b.x);
    l.c = -l.a*a.x - l.b*a.y;
    return l;
}

#define sq(a) ((a)*(a))
// squared distance from point to line
ptt ln2ptdist(ln l, pt p) {
    return sq(l.a*p.x+l.b*p.y+l.c)/(sq(l.a)+sq(l.b));
}

bool para(ln l1, ln l2) {
```

```
    if (l1.dege() or l2.dege()) return eq(l1.a,l2.a) or eq(l1.b,l2.b);
    return eq(l1.a*l2.b,l1.b*l2.a);
}

bool same(ln l1, ln l2) {
    if (para(l1,l2)) {
        if (eq(l1.a,0)) return eq(l1.b*l2.c,l1.c*l2.b);
        return eq(l1.a*l2.c,l1.c*l2.a);
    }
    return 0;
}

pt lins(ln l1, ln l2) {
    return {(l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b),
            (l2.a * l1.c - l1.a * l2.c) / (l2.b * l1.a - l1.b * l2.a)};
}
```

5.3 Circles

5.3.1 Circle-line intersection

```
vector<pt> cir_ln_int(pt c, ptt r, pt a, pt b) {
    auto prc = a + ((conj(b-a)*c).x)/conj(b-a);
    auto cpr = prc - c;
    auto ul = (b-a)/abs(b-a);
    if (eq(abs(cpr),r)) {
        return {prc};
    }
    if (abs(cpr) > r) {
        return {};
    }
    auto len = sqrt(r*r-norm(cpr));
    return {prc + len*ul,prc - len*ul};
}
```

5.3.2 Two circles intersection

```
// circles

// 2 circles intersection
vector<pt> cirs_int(pt c1, ptt r1, pt c2, ptt r2) {
    auto cl = c1 - c2;
    auto cln = cl/abs(cl);
    if (eq(norm(cl),0.0)) {
        if(eq(r1,r2)) return {c1+r1,c1-r1,c2-r1}; // return some three points
        return {};
    }
    if (eq(abs(cl),r1 + r2)) {
        return {c2+cln*r2};
    }
    if (abs(cl) > r1 + r2) {
        return {};
    }
    auto ang = polar(r2,acos((norm(cl)+r2*r2-r1*r1)/(2*r1*abs(cl))));
    return {c2+cln*conj(ang),c2+cln*ang};
}
```

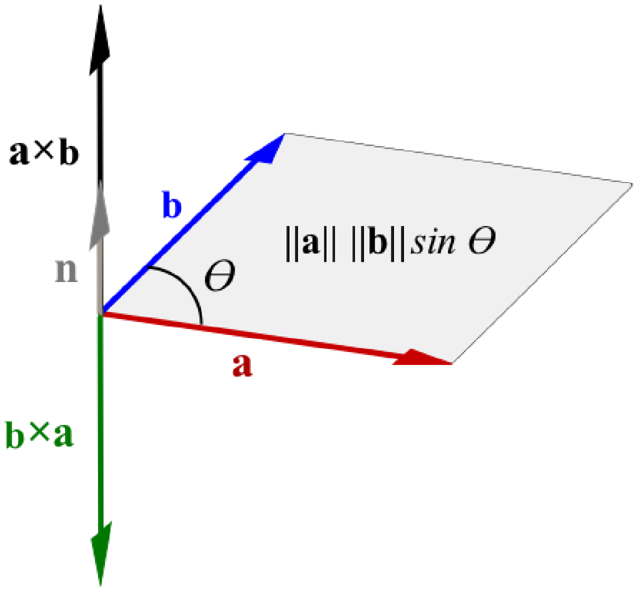
5.3.3 Triangles: inscribed + circumscribed circle

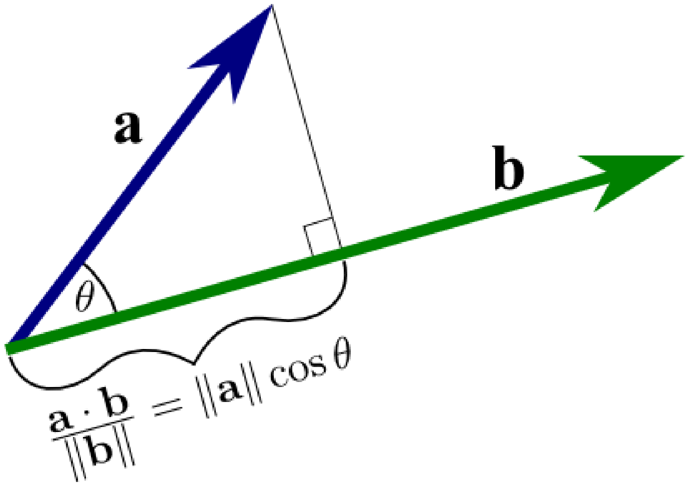
```
// triangles
// inscribed circle center (uses ins)
pt incirc(pt a, pt b, pt c) {
    pt ct1 = (a + b)/2.0;
    pt ct2 = (b + c)/2.0;
    pt an(1,1);
    return ins(ct1,ct1*an,ct2,ct2*an);
}
// circumscribed circle center (for radius output abs(ct1 - ins(...))  (uses ins)
// radius using only lengths R = a    b    c/(4    A).
pt circirc(pt a, pt b, pt c) {
    pt ct1 = (a+b)/2.0;
    pt ct2 = (b+c)/2.0;
    return ins(c,ct1,a,ct2);
}
```

5.3.4 Triangles & quadrilaterals facts

```
//  triangles
//  2. A triangle with base b and height h has area A=0.5  b  h .
//  3. A triangle with three sides: a, b, c has perimeter p = a + b + c and semi-perimeter
//  s = 0.5    p.
//  4. A triangle with 3 sides: a, b, c and semi-perimeter s has area
//     A = sqrt(s    (s     a)    (s     b)    (s     c)). This formula is called the Heron s Formula.
//  5. A triangle with area A and semi-perimeter s has an inscribed circle (incircle) with
//     radius r = A/s.

//   quadrilaterals
//  1. Quadrilateral or Quadrangle is a polygon with four edges (and four vertices). The term
//  polygon  itself is described in more details below (Section 7.3).
//  Figure 7.7 shows a few examples of Quadrilateral objects.
//  2. Rectangle is a polygon with four edges, four vertices, and four right angles.
//  3. A rectangle with width w and height h has area A = w    h and perimeter p =
//  2  (w+h).
//  4. Square is a special case of a rectangle where w = h.
//  5. Trapezium is a polygon with four edges, four vertices, and one pair of parallel edges. If the two non-parallel sides have the same length, we have an Isosceles Trapezium.
//  6. A trapezium with a pair of parallel edges of lengths w1 and w2; and a height h between both parallel edges has area A = 0.5    (w1 + w2)    h.
//  7. Parallelogram is a polygon with four edges and four vertices. Moreover, the opposite sides must be parallel.
//  8. Kite is a quadrilateral which has two pairs of sides of the same length which are adjacent to each other. The area of a kite is diagonal1      diagonal2/2.
//  9. Rhombus is a special parallelogram where every side has equal length. It is also a special case of kite where every side has equal length.
```





5.4 Convex hull

```
using ptt = double;
using pt = complex<ptt>;
#define x real()
#define y imag()
#define eq(a,b) (abs(a-(b)) <= EPS)
#define crs(a,b) (conj(a)*(b)).y
pt pts[MX];
ll n;
// returns set of points oriented counter clockwise with 1st point having
// minimal X and then Y coordinate
// first point and the last point is the same! (polygon representation)
// comment last push_back if you want just the vertices
// edit (1) to add also collinear points
vector<pt> conv_hull() {
    int lp = 0;
    FOR(i,1,n)
    if (pts[i].x < pts[lp].x || (eq(pts[i].x,pts[lp].x) && pts[i].y < pts[lp].y)) lp = i;
    swap(pts[0],pts[lp]);
    vector<pt> res;
    if (n < 3) {
        F(n) res.pb(pts[i]);
        res.pb(pts[0]);
        return res;
    }
    pt piv = pts[0];
    sort(pts+1,pts+n,[&](pt a, pt b) {
        pt d1 = a - piv;
        pt d2 = b - piv;
        if (eq(crs(d1,d2),0)) return norm(d1) < norm(d2);
        return crs(d1,d2) > 0; });
    res.pb(pts[0]); res.pb(pts[1]);
    ll i = 2;
    while (i < n) {
        ll j = res.size()-1;
        if (j == 0 or crs(res[j] - res[j-1],pts[i] - res[j]) > 0 + EPS) res.pb(pts[i++]); // (1) change + EPS to -EPS to add collinear segments as well
        else res.pop_back();
    }
    res.pb(pts[0]);
    return res;
}
```

5.5 3D geometry

5.5.1 Gloabal features

```
struct pt{
    double x, y, z;
    pt(){};
    pt(double _x, double _y, double _z){ x=_x; y=_y; z=_z; }
    pt operator+ (pt p) { return pt(x+p.x, y+p.y, z+p.z); }
    pt operator- (pt p) { return pt(x-p.x, y-p.y, z-p.z); }
    pt operator* (double c) { return pt(x*c, y*c, z*c); }
};

double dot(pt a, pt b){
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

pt crs(pt a, pt b) {
    return pt(a.y*b.z-a.z*b.y,
              a.z*b.x-a.x*b.z,
              a.x*b.y-a.y*b.x);
}

double norm(pt a) {
    return dot(a,a);
}

double abs(pt a) {
    return sqrt(norm(a));
}

// compute a, b, c, d such that all pts lie on ax + by + cz = d. TODO: test this
void planeFromPts(pt p1, pt p2, pt p3, double& a, double& b, double& c, double& d) {
    pt normal = crs(p2-p1, p3-p1);
    a = normal.x; b = normal.y; c = normal.z;
    d = -a*p1.x-b*p1.y-c*p1.z;
}

// project pt onto plane. TODO: test this
pt ptPlaneProj(pt p, double a, double b, double c, double d) {
    double l = (a*p.x+b*p.y+c*p.z+d)/(a*a+b*b+c*c);
    return pt(p.x-a*l, p.y-b*l, p.z-c*l);
}

// distance from pt p to plane aX + bY + cZ + d = 0
double ptPlaneDist(pt p, double a, double b, double c, double d){
    return abs(a*p.x + b*p.y + c*p.z + d) / sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
double planePlaneDist(double a, double b, double c, double d1, double d2){
    return fabs(d1 - d2) / sqrt(a*a + b*b + c*c);
}

// square distance between pt and line
double ptLineDistSq(pt x1, pt x2, pt x0){
    return norm(crs(x2-x1,x1-x0))/norm(x2-x1);
}

// Distance between lines ab and cd.
double lineLineDistanceSq(pt x1, pt x2, pt x3, pt x4) {
    pt a = x2 - x1;
    pt b = x4 - x3;
    pt c = x3 - x1;
    return norm(dot(c,crs(a,b)))/norm(crs(a,b));
}

double signedTetrahedronVol(pt A, pt B, pt C, pt D) {
    double A11 = A.x - B.x;
    double A12 = A.x - C.x;
    double A13 = A.x - D.x;
    double A21 = A.y - B.y;
    double A22 = A.y - C.y;
    double A23 = A.y - D.y;
    double A31 = A.z - B.z;
    double A32 = A.z - C.z;
    double A33 = A.z - D.z;
    double det =
        A11*A22*A33 + A12*A23*A31 +
        A13*A21*A32 - A11*A23*A32 -
        A12*A21*A33 - A13*A22*A31;
    return det / 6;
}

// Parameter is a vector of vectors of pts - each interior vector
// represents the 3 pts that make up 1 face, in any order.
// Note: The polyhedron must be convex, with all faces given as triangles.
double polyhedronVol(vector<vector<pt> > poly) {
    int i,j;
    pt cent(0,0,0);
```

```
    for (i=0; i<poly.size(); i++)
        for (j=0; j<3; j++)
            cent=cent+poly[i][j];
    cent=cent*(1.0/(poly.size()*3));
    double v=0;
    for (i=0; i<poly.size(); i++)
        v+=fabs(signedTetrahedronVol(cent,poly[i][0],↩
             poly[i][1],poly[i][2]));
    return v;
}
```

5.5.2 Spherical distance

```
#include "../template.h"

// Computes distance between 2 points on sphere
// Each point: pair{lat,lon} in RADIANS

double dist(pair<double,double> a, pair<double,double> ↩
    b, double r) {
    long double d_lat=fabs(a.x-b.x),
        d_lon=fabs(a.y-b.y),
        d_ang=2*asin(sqrt(sin(d_lat/2)*sin(d_lat/2)+cos↩
             (a.x)*cos(b.x)*sin(d_lon/2)*sin(d_lon/   ↩
             2)));
    return r*d_ang;
}
```

6 Dynamic programming

6.1 Longest increasing subsequence

```
ll a[MX]; // input
ll n; // input size
ll par[MX]; // parent
ll tl[MX];  // current max lis (not always lis)
vll lis(){ // largest strictly increasing subsequence
    ll sl = 0;
    tl[0] = -1;
    F(n) {
        auto le = lower_bound(tl+1,tl+sl+1,a[i],[&](ll ↩
            i, ll b){return a[i] < b;});
        *le = i;
        par[i] = *(le - 1);
        sl = max(sl,(ll)(le-tl));
    }
    ll i = tl[sl];
    vll res;
    while(i != -1) {
        res.pb(a[i]);
            i = par[i];
    }
    reverse(all(res));
    return res;
}
```

6.2 Minimal cost path

```
#include "../template.cpp"


ll cost[MX][MX];
ll tc[MX][MX];
ll minCost(ll cost[R][C], ll m, ll n){
     ll i, j;
     tc[0][0] = cost[0][0];
     /* Initialize first column of total cost(tc) array↩
          */
     for (i = 1; i <= m; i++)
        tc[i][0] = tc[i-1][0] + cost[i][0];
     /* Initialize first row of tc array */
     for (j = 1; j <= n; j++)
        tc[0][j] = tc[0][j-1] + cost[0][j];
     /* Construct rest of the tc array */
     for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            tc[i][j] = min(tc[i-1][j-1], min(tc[i-1][j↩
                ], tc[i][j-1])) + cost[i][j];

     return tc[m][n];
}
```

7 Others

7.1 Binary search

```
// Binary search. This is included because binary ↩
    search can be tricky.
// n is size of array A, c is value we're searching for↩
    . Semantics follow those of std::lower_bound and ↩
    std::upper_bound
int lower_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l; //prevents integer overflow
        if (A[m] < c) l = m+1; else r = m;
    }
    return l;
}

int upper_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l;
        if (A[m] <= c) l = m+1; else r = m;
    }
    return l;
}
```

7.2 Ternary search

```
function ternarySearch(f, left, right, ↩
     absolutePrecision)
//left and right are the current bounds; the maximum is↩
      between them
        if (right-left < absolutePrecision)
        return (left+right)/2     leftThird := (left*2+↩
             right)/3     rightThird := (left+right*2)↩
             /3
        if (f(leftThird) < f(rightThird))
                return ternarySearch(f, leftThird, ↩
                    right, absolutePrecision)
        else
                return ternarySearch(f, left, ↩
                    rightThird, absolutePrecision)
end
```

7.3 Bit tricks

```
## Operator precedence http://en.cppreference.com/w/cpp↩
    /language/operator_precedence
## Bit operations
i & -i  // Zeroes all bits except the least significant↩
     non-zero bit
(x^y) < 0 // Numbers have different signs
i^(1<<j) // toggle jth bit
i&~(1<<j) // turn off jth bit

## Zabudovan  C++ operace
//V echny funkce  p  i j  m a j   unsigned long/long longy↩
     se suffixem l/ll;
int __builtin_popcount (unsigned int x); //Returns the ↩
    number of 1-bits in x.
uint8_t reverse_bits(uint8_t b) { return (b * 0↩
    x0202020202ULL & 0x010884422010ULL) % 1023; }
int __builtin_clz (unsigned int x); //Returns the ↩
    number of leading 0-bits in x, starting at the ↩
    most significant bit (MSB) position. If x is 0, ↩
    the result is undefined.
```

```
// Usage: nearest higher power of 2: 1ULL << (sizeof(x)*8-__builtin_clz(x))
int __builtin_ffs (int x)
//Returns one plus the index of the least significant 1-bit of x, or if x is zero, returns zero.
int __builtin_ctz (unsigned int x)
//Returns the number of trailing 0-bits in x, starting at the least significant bit position. If x is 0, the result is undefined.
int __builtin_clrsb (int x)
//Returns the number of leading redundant sign bits in x, i.e. the number of bits following the most significant bit that are identical to it. There are no special cases for 0 or other values.
```

7.4 K-th minimum

```
#include "../template.cpp"

// todo non-destructive - this method swaps elements and destroyes indices

// put in unordered array and what smallest elem you want to find
// idx from 0 to arr.size()-1
int findKthMin(vi &arr, int kth){
    int low = 0, high = arr.size()-1, pivot, left, right;
    bool flag;
    while(low < high){
        pivot = (low + high) / 2;
        swap(arr[pivot], arr[low]);
        pivot = low;
        flag = true; // pivot on left?
        left = low;
        right = high;
        while(true){
            if(flag){
                while(left < right && arr[left] < arr[right]) right--;
            }else{
                while(left < right && arr[left] < arr[right]) left++;
            }
            if(left >= right)break;
            swap(arr[left], arr[right]);
            if(flag){
                left++;
            }else{
                right--;
            }
            flag = !flag;
        }
        if(left == kth)break;

        if(left < kth){
            low = left + 1;
        }else{
            high = left - 1;
        }
    }
    return kth;
}

void test(vi &arr, ll k){
    ll r=findKthMin(arr, k);
    cout << k << "-th lowest elem is " << arr[r] << " at index " << r << endl;
}

int main () {
    vi arr={5,9,1,5,2};
    test(arr, 0);
    test(arr, 1);
    test(arr, 2);
    test(arr, 3);
    test(arr, 4);

    return 0;
}
```

7.5 All nearest smaller values

```
// Linear time all nearest smaller values, standard stack-based algorithm.
// res[i] = index of nearest smallest value in interval <0,i-1>; -1 if not found
void ansv_left(vector<int>& v, vector<int>& res) {
    stack<pair<int, int> > stk; stk.push(make_pair(INT_MIN, v.size()));
    for (int i = v.size()-1; i >= 0; i--) {
        while (stk.top().first > v[i]) {
            res[stk.top().second] = i; stk.pop();
        }
        stk.push(make_pair(v[i], i));
    }
    while (stk.top().second < v.size()) {
        res[stk.top().second] = -1; stk.pop();
    }
}
```

7.6 STL

```
#include "../template.cpp"

struct Foo{
    Foo(){}
    Foo(int i):i(i){}
    int i;
};

struct Cmp {
    bool operator()(const Foo & a, const Foo & b) const { return a.i<b.i; }
};

struct cmpByStringLength {
    bool operator()(const string& a, const string& b) const {
        return a.length() < b.length();
    }
};


int main() {
    Foo one(1);

    priority_queue<Foo, vector<Foo>, Cmp> pq;
    pq.push(one); pq.top(); pq.pop();

    set<Foo, Cmp> s; s.insert(one); s.erase(one);

    map<Foo, Foo, Cmp> m;
    m[one]=one;
    dbg(m[one].i);


    vector<Foo> v;
    lower_bound(all(v), one, [](const Foo& a, const Foo& b)->bool{return a.i<b.i;});
    sort(all(v), [](const Foo& a, const Foo& b)->bool{return a.i<b.i;});



    return 0;
}
```

7.7 Template

```
#include <bits/stdc++.h>

using namespace std;

using ll = long long;
using ld = long double;

using pll = pair<ll, ll>;
using vll = vector<ll>;
using vpll = vector<pll>;

#define pb push_back
#define FOR(i, m, n) for (ll i(m); i < n; i++)
#define REP(i, n) FOR(i, 0, n)
#define F(n) REP(i, n)
#define FF(n) REP(j, n)
struct d_ {
```

```
    template<typename T> d_ & operator ,(const T & x) {↩
        cerr << ' ' <<  x; return *this;}
    template<typename T> d_ & operator ,(const vector<T↩
        > & x) { for(auto x: x) cerr << ' ' <<  x; ↩
        return *this;}
} d_t;
#define D(args ...) { d_t,"|",__LINE__,"|",#args,":",↩
    args,"\n"; }
#define dbg(args ...) D(args)
#define EPS (1e-10)
#define INF (1LL<<61)
#define CL(A,I) (memset(A,I,sizeof(A)))
#define all(x) begin(x),end(x)
#define IN(n) ll n;cin >> n;
#define x first
#define y second
```

7.8 .bash-profile

```
mkcd() { mkdir $1 && cd $1; }
alias c="g++ -Wall -pedantic -g -std=c++11 main.cpp"
alias rt='c && find *.in | xargs -I @ sh -c "echo Test ↩
    @; ./a.out < @;"'
alias r="c && ./a.out"
alias main="cp ../t.cpp ./main.cpp"
```

7.9 .vimrc

```
set ts=4
set sw=4
set sr
set et
set sta
set nu
set bs=2
set ai
set tw=79
set fo=c,q,r,t
set bg=dark " only dark terminal bg
set mouse=a
set cb=unnamedplus
set hid
colo delek
filetype plugin indent on
syntax on
map <S-q> :s,^,//,<cr>
map <S-e> :s,^//,<cr>
inoremap jk <ESC>
```

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com