

# LU ICPC kladite ;)

## Contents

Algebra .....	1
Sum formulas .....	1
FFT .....	1
Geometry .....	1
Basics .....	1
Closest pair .....	2
Data structures .....	3
Treap .....	3
Sparse table .....	3
Trie .....	3
Graph algorithms .....	3
Bellman-Ford .....	3
Dijkstra .....	3
Floyd-warshall .....	3
Bridges & articulations .....	4
Dinic's max flow / matching .....	4
Kosaraju's algorithm .....	4
Lowest common ancestor (LCA) .....	5
String Processing .....	5
Knuth-Morris-Pratt (KMP) .....	5
Suffix Array .....	5
Rabin-Karp .....	6
Z-function .....	6
Manacher's .....	6
Dynamic programming .....	6
Convex hull trick .....	6
Longest Increasing Subsequence .....	6

## Algebra

### Sum formulas

$$(a+b)^2 = a^2 + 2ab + b^2$$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2}$$

$$\sum_{k=1}^n k^2 = \frac{n(n+1)(2n+1)}{6}$$

$$\sum_{k=1}^n k^3 = \frac{n(n+1)(2n+1)}{6}$$

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

### FFT

// Fast Fourier Transform - O(nlogn)

```
/*
// Use struct instead. Performance will be way better!
typedef complex<ld> T;
T a[N], b[N];
*/

struct T {
    ld x, y;
    T() : x(0), y(0) {}
    T(ld a, ld b=0) : x(a), y(b) {}

    T operator+=(ld k) { x/=k; y/=k; return (*this); }
    T operator*(T a) const { return T(x*a.x - y*a.y, x*a.y + y*a.x); }
    T operator+(T a) const { return T(x+a.x, y+a.y); }
    T operator-(T a) const { return T(x-a.x, y-a.y); }
} a[N], b[N];

// a: vector containing polynomial
// n: power of two greater or equal product size
/*
// Use iterative version!
void fft_recursive(T* a, int n, int s) {
    if (n == 1) return;
    T tmp[n];
    for (int i = 0; i < n/2; ++i)
        tmp[i] = a[2*i], tmp[i+n/2] = a[2*i+1];

    fft_recursive(&tmp[0], n/2, s);
    fft_recursive(&tmp[n/2], n/2, s);

    T wn = T(cos(s*2*PI/n), sin(s*2*PI/n)), w(1,0);
    for (int i = 0; i < n/2; i++, w=w*wn)
        a[i] = tmp[i] + w*tmp[i+n/2],
        a[i+n/2] = tmp[i] - w*tmp[i+n/2];
}
*/

void fft(T* a, int n, int s) {
    for (int i=0, j=0; i<n; i++) {
        if (i>j) swap(a[i], a[j]);
        for (int l=n/2; (j^=l) < l; l>=>=1);
    }

    for(int i = 1; (1<=i) <= n; i++){
        int M = 1 << i;
        int K = M >> 1;
        T wn = T(cos(s*2*PI/M), sin(s*2*PI/M));
        for(int j = 0; j < n; j += M) {
            T w = T(1, 0);
            for(int l = j; l < K + j; ++l){
                T t = w*a[l + K];
                a[l + K] = a[l]-t;
                a[l] = a[l] + t;
                w = wn*w;
            }
        }
    }
}

// assert n is a power of two greater or equal product size
```

```
// n = na + nb; while (n&(n-1)) n++;
void multiply(T* a, T* b, int n) {
    fft(a,n,1);
    fft(b,n,1);
    for (int i = 0; i < n; i++) a[i] = a[i]*b[i];
    fft(a,n,-1);
    for (int i = 0; i < n; i++) a[i] /= n;
}

// Convert to integers after multiplying:
// (int)(a[i].x + 0.5);
```

## Geometry

### Basics

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

```
using namespace std;
```

```
#define st first
#define nd second
#define pb push_back
#define cl(x,v) memset((x), (v), sizeof(x))
#define db(x) cerr << #x << " == " << x << endl
#define dbs(x) cerr << x << endl
#define _ << ", " <<
```

```
typedef long long ll;
typedef long double ld;
typedef pair<int,int> pii;
typedef pair<int, pii> piii;
typedef pair<ll,ll> pll;
typedef pair<ll, pll> pll;
typedef vector<int> vi;
typedef vector<vi> vii;
```

```
const ld EPS = 1e-9, PI = acos(-1.);
const ll LINF = 0x3f3f3f3f3f3f3f3f;
const int INF = 0x3f3f3f3f, MOD = 1e9+7;
const int N = 1e5+5;
```

```
typedef long double type;
//for big coordinates change to long long
```

```
bool ge(type x, type y) { return x + EPS > y; }
bool le(type x, type y) { return x - EPS < y; }
bool eq(type x, type y) { return ge(x, y) and le(x, y); }
int sign(type x) { return ge(x, 0) - le(x, 0); }
```

```
struct point {
    type x, y;

    point() : x(0), y(0) {}
    point(type _x, type _y) : x(_x), y(_y) {}
```

```
    point operator -( ) { return point(-x, -y); }
    point operator +(point p) { return point(x + p.x, y + p.y); }
    point operator -(point p) { return point(x - p.x, y - p.y); }
```

```

point operator *(type k) { return point(x*k, y*k); }
point operator /(type k) { return point(x/k, y/k); }

//inner product
type operator *(point p) { return x*p.x + y*p.y; }
//cross product
type operator %(point p) { return x*p.y - y*p.x; }

bool operator ==(const point &p) const{ return x == p.x and y == p.y; }
bool operator !=(const point &p) const{ return x != p.x or y != p.y; }
bool operator <(const point &p) const { return (x < p.x) or (x == p.x and y < p.y); }

// 0 => same direction
// 1 => p is on the left
// -1 => p is on the right
int dir(point o, point p) {
    type x = (*this - o) % (p - o);
    return ge(x,0) - le(x,0);
}

bool on_seg(point p, point q) {
    if (this->dir(p, q) return 0;
    return ge(x, min(p.x, q.x)) and le(x, max(p.x, q.x)) and
ge(y, min(p.y, q.y)) and le(y, max(p.y, q.y));
}

ld abs() { return sqrt(x*x + y*y); }
type abs2() { return x*x + y*y; }
ld dist(point q) { return (*this - q).abs(); }
type dist2(point q) { return (*this - q).abs2(); }

ld arg() { return atan2l(y, x); }

// Project point on vector y
point project(point y) { return y * ((*this * y) / (y * y)); }

// Project point on line generated by points x and y
point project(point x, point y) { return x + (*this - x).project(y-x); }

ld dist_line(point x, point y) { return dist(project(x, y)); }

ld dist_seg(point x, point y) {
    return project(x, y).on_seg(x, y) ? dist_line(x, y) :
min(dist(x), dist(y));
}

point rotate(ld sin, ld cos) { return point(cos*x - sin*y, sin*x + cos*y); }
point rotate(ld a) { return rotate(sin(a), cos(a)); }

// rotate around the argument of vector p
point rotate(point p) { return rotate(p.y / p.abs(), p.x / p.abs()); }
};

```

```

int direction(point o, point p, point q) { return p.dir(o, q); }

point rotate_ccw90(point p) { return point(-p.y,p.x); }
point rotate_cw90(point p) { return point(p.y,-p.x); }

//for reading purposes avoid using * and % operators, use the
functions below:
type dot(point p, point q) { return p.x*q.x + p.y*q.y; }
type cross(point p, point q) { return p.x*q.y - p.y*q.x; }

//double area
type area_2(point a, point b, point c) { return cross(a,b) +
cross(b,c) + cross(c,a); }

//angle between (a1 and b1) vs angle between (a2 and b2)
//1 : bigger
//-1 : smaller
//0 : equal
int angle_less(const point& a1, const point& b1, const point& a2,
const point& b2) {
    point p1(dot( a1, b1), abs(cross( a1, b1)));
    point p2(dot( a2, b2), abs(cross( a2, b2)));
    if(cross(p1, p2) < 0) return 1;
    if(cross(p1, p2) > 0) return -1;
    return 0;
}

ostream &operator<<(ostream &os, const point &p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

Closest pair
#include "basics.cpp"
//DIVIDE AND CONQUER METHOD
//Warning: include variable id into the struct point

```

```

struct cmp_y {
    bool operator()(const point &a, const point &b) const {
        return a.y < b.y;
    }
};

ld min_dist = LINF;
pair<int, int> best_pair;
vector<point> pts, stripe;
int n;

void upd_ans(const point &a, const point &b) {
    ld dist = sqrt((a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y));
    if (dist < min_dist) {
        min_dist = dist;
        // best_pair = {a.id, b.id};
    }
}

void closest_pair(int l, int r) {
    if (r - l <= 3) {

```

```

        for (int i = l; i < r; ++i) {
            for (int j = i + 1; j < r; ++j) {
                upd_ans(pts[i], pts[j]);
            }
        }
        sort(pts.begin() + l, pts.begin() + r, cmp_y());
        return;
    }

    int m = (l + r) >> 1;
    type midx = pts[m].x;
    closest_pair(l, m);
    closest_pair(m, r);

    merge(pts.begin() + l, pts.begin() + m, pts.begin() + m,
pts.begin() + r, stripe.begin(), cmp_y());
    copy(stripe.begin(), stripe.begin() + r - l, pts.begin() + l);

    int stripe_sz = 0;
    for (int i = l; i < r; ++i) {
        if (abs(pts[i].x - midx) < min_dist) {
            for (int j = stripe_sz - 1; j >= 0 && pts[i].y -
stripe[j].y < min_dist; --j)
                upd_ans(pts[i], stripe[j]);
            stripe[stripe_sz++] = pts[i];
        }
    }
}

int main(){
    //read and save in vector pts
    min_dist = LINF;
    stripe.resize(n);
    sort(pts.begin(), pts.end());
    closest_pair(0, n);
}

//LINE SWEEP
int n; //amount of points
point pnt[N];

struct cmp_y {
    bool operator()(const point &a, const point &b) const {
        if(a.y == b.y) return a.x < b.x;
        return a.y < b.y;
    }
};

ld closest_pair() {
    sort(pnt, pnt+n);
    ld best = numeric_limits<double>::infinity();
    set<point, cmp_y> box;

    box.insert(pnt[0]);
    int l = 0;

    for (int i = 1; i < n; i++){
        while(l < i and pnt[i].x - pnt[l].x > best)
            box.erase(pnt[l++]);

```

```

    for(auto it = box.lower_bound({0, pnt[i].y - best}); it !=
box.end() and pnt[i].y + best >= it->y; it++)
    best = min(best, hypot(pnt[i].x - it->x, pnt[i].y - it->y));
    box.insert(pnt[i]);
}
return best;
}

```

## Data structures

### Treap

// Implicit segment tree implementation

```

struct Node{
    int value;
    int cnt;
    int priority;
    Node *left, *right;
    Node(int p) : value(p), cnt(1), priority(gen()), left(NULL),
right(NULL) {};
};

```

typedef Node\* pnode;

```

int get(pnode q){
    if(!q) return 0;
    return q->cnt;
}

```

```

void update_cnt(pnode &q){
    if(!q) return;
    q->cnt = get(q->left) + get(q->right) + 1;
}

```

```

void merge(pnode &T, pnode lef, pnode rig){
    if(!lef) {
        T=rig;
        return;
    }
    if(!rig){
        T=lef;
        return;
    }
    if(lef->priority > rig->priority){
        merge(lef->right, lef->right, rig);
        T = lef;
    }
    else{
        merge(rig->left, lef, rig->left);
        T = rig;
    }
    update_cnt(T);
}

```

```

void split(pnode cur, pnode &lef, pnode &rig, int key){
    if(!cur){
        lef = rig = NULL;
        return;
    }

```

```

    int id = get(cur->left) + 1;
    if(id <= key){
        split(cur->right, cur->right, rig, key - id);
        lef = cur;
    }
    else{
        split(cur->left, lef, cur->left, key);
        rig = cur;
    }
    update_cnt(cur);
}

```

### Sparse table

```

const int N;
const int M; //log2(N)
int sparse[N][M];

void build() {
    for(int i = 0; i < n; i++)
        sparse[i][0] = v[i];

    for(int j = 1; j < M; j++)
        for(int i = 0; i < n; i++)
            sparse[i][j] =
                i + (1 << j - 1) < n
                ? min(sparse[i][j - 1], sparse[i + (1 << j - 1)][j - 1])
                : sparse[i][j - 1];
}

```

```

int query(int a, int b){
    int pot = 32 - __builtin_clz(b - a) - 1;
    return min(sparse[a][pot], sparse[b - (1 << pot) + 1][pot]);
}

```

### Trie

```

// Trie <0(|S|), 0(|S|)>
int trie[N][26], trien = 1;

int add(int u, char c){
    c-='a';
    if (trie[u][c]) return trie[u][c];
    return trie[u][c] = ++trien;
}

```

```

//to add a string s in the trie
int u = 1;
for(char c : s) u = add(u, c);

```

## Graph algorithms

### Bellman-Ford

```

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

        for (Edge e : edges)

```

```

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

    if (!any)
        break;
}

// display d, for example, on the screen
}

```

### Dijkstra

```

/*****
*   DIJKSTRA'S   ALGORITHM   (SHORTEST   PATH   TO   A   VERTEX)
*
* Time complexity: O((V+E)logE)
* Usage: dist[node]
* Notation: m:      number of edges
*           (a, b, w): edge between a and b with weight w
*
*           s:      starting node
*           par[v]:  parent node of u, used to rebuild the
shortest path
*****/

```

```

vector<int> adj[N], adjw[N];
int dist[N];

```

```

memset(dist, 63, sizeof(dist));
priority_queue<pii> pq;
pq.push(mp(0,0));

```

```

while (!pq.empty()) {
    int u = pq.top().nd;
    int d = -pq.top().st;
    pq.pop();

    if (d > dist[u]) continue;
    for (int i = 0; i < adj[u].size(); ++i) {
        int v = adj[u][i];
        int w = adjw[u][i];
        if (dist[u] + w < dist[v])
            dist[v] = dist[u]+w, pq.push(mp(-dist[v], v));
    }
}

```

### Floyd-warshall

```

/*****
*   FLOYD-WARSHALL   ALGORITHM   (SHORTEST   PATH   TO   ANY   VERTEX)
*
* Time complexity: O(V^3)
* Usage: dist[from][to]
* Notation: m:      number of edges
*           n:      number of vertices
*           (a, b, w): edge between a and b with weight w
*
*****/

```

```
int adj[N][N]; // no-edge = INF

for (int k = 0; k < n; ++k)
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            adj[i][j] = min(adj[i][j], adj[i][k]+adj[k][j]);
```

## Bridges & articulations

```
// Articulation points and Bridges O(V+E)
int par[N], art[N], low[N], num[N], ch[N], cnt;

void articulation(int u) {
    low[u] = num[u] = ++cnt;
    for (int v : adj[u]) {
        if (!num[v]) {
            par[v] = u; ch[u]++;
            articulation(v);
            if (low[v] >= num[u]) art[u] = 1;
            if (low[v] > num[u]) { /* u-v bridge */
                low[u] = min(low[u], low[v]);
            }
        } else if (v != par[u]) low[u] = min(low[u], num[v]);
    }
}

for (int i = 0; i < n; ++i) if (!num[i])
    articulation(i), art[i] = ch[i]>1;
```

## Dinic's max flow / matching

```
// Dinic -  $O(V^2 * E)$ 
// Bipartite graph or unit flow -  $O(\sqrt{V} * E)$ 
// Small flow -  $O(F * (V + E))$ 
// USE INF = 1e9!
```

```

/*****
*   DINIC   (FIND   MAX   FLOW   /   BIPARTITE   MATCHING)
*
* Time complexity:  $O(EV^2)$ 
* Usage: dinic()
*   add_edge(from, to, capacity)
* Testcase:
* add_edge(src, 1, 1);   add_edge(1, snk, 1);   add_edge(2, 3, INF);
*
* add_edge(src, 2, 1);   add_edge(2, snk, 1);   add_edge(3, 4, INF);
*
*   add_edge(src,    2,    1);           add_edge(3,    snk,    1);
*
* add_edge(src, 2, 1);   add_edge(4, snk, 1);   => dinic() = 4
*
*****/
```

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

const int N = 1e5+1, INF = 1e9;
struct edge {int v, c, f;};

int n, src, snk, h[N], ptr[N];
vector<edge> eds;
```

```
vector<int> g[N];

void add_edge (int u, int v, int c) {
    int k = eds.size();
    eds.push_back({v, c, 0});
    eds.push_back({u, 0, 0});
    g[u].push_back(k);
    g[v].push_back(k+1);
}

void clear() {
    memset(h, 0, sizeof h);
    memset(ptr, 0, sizeof ptr);
    eds.clear();
    for (int i = 0; i < N; i++) g[i].clear();
    src = 0;
    snk = N-1;
}

bool bfs() {
    memset(h, 0, sizeof h);
    queue<int> q;
    h[src] = 1;
    q.push(src);
    while(!q.empty()) {
        int u = q.front(); q.pop();
        for(int i : g[u]) {
            int v = eds[i].v;
            if (!h[v] and eds[i].f < eds[i].c)
                q.push(v), h[v] = h[u] + 1;
        }
    }
    return h[snk];
}

int dfs (int u, int flow) {
    if (!flow or u == snk) return flow;
    for (int &i = ptr[u]; i < g[u].size(); ++i) {
        edge &dir = eds[g[u][i]], &rev = eds[g[u][i]^1];
        int v = dir.v;
        if (h[v] != h[u] + 1) continue;
        int inc = min(flow, dir.c - dir.f);
        inc = dfs(v, inc);
        if (inc) {
            dir.f += inc, rev.f -= inc;
            return inc;
        }
    }
    return 0;
}

int dinic() {
    int flow = 0;
    while (bfs()) {
        memset(ptr, 0, sizeof ptr);
        while (int inc = dfs(src, INF)) flow += inc;
    }
    return flow;
}
```

```
//Recover Dinic
void recover(){
    for(int i = 0; i < eds.size(); i += 2){
        //edge (u -> v) is being used with flow f
        if(eds[i].f > 0) {
            int v = eds[i].v;
            int u = eds[i^1].v;
        }
    }
}

/*****
*                               FLOW                               WITH                               DEMANDS
*
*                               *
*   1       -       Finding       an       arbitrary       flow
*
* Assume a network with [L, R] on edges (some may have L = 0), let's
call it old network.
* Create a New Source and New Sink (this will be the src and snk for
Dinic).
*                               Modelling                               Network:
*
* 1) Every edge from the old network will have cost R - L
*
* 2) Add an edge from New Source to every vertex v with cost:
*
*       Sum(L) for every (u, v). (sum all L that LEAVES v)
*
* 3) Add an edge from every vertex v to New Sink with cost:
*
*       Sum(L) for every (v, w). (sum all L that ARRIVES v)
*
* 4) Add an edge from Old Source to Old Sink with cost INF (circulation
problem)
* The Network will be valid if and only if the flow saturates the
network (max flow == sum(L))
*
*   2       -       Finding       Min       Flow
*
* To find min flow that satisfies just do a binary search in the
(Old Sink -> Old Source) edge
* The cost of this edge represents all the flow from old network
*
* Min flow = Sum(L) that arrives in Old Sink + flow that leaves (Old
Sink -> Old Source)
*****/
```

```
int main () {
    clear();
    return 0;
}
```

## Kosaraju's algorithm

```
//*****
* KOSARAJU'S ALGORITHM (GET EVERY STRONGLY CONNECTED COMPONENTS
(SCC))
* Description: Given a directed graph, the algorithm generates a
```

```

list of every *
* strongly connected components. A SCC is a set of points in which
you can reach *
* every point regardless of where you start from. For instance,
cycles can be *
* a SCC themselves or part of a greater SCC.
*
* This algorithm starts with a DFS and generates an array called
"ord" which *
* stores vertices according to the finish times (i.e. when it reaches
"return"). *
* Then, it makes a reversed DFS according to "ord" list. The set of
points *
* visited by the reversed DFS defines a new SCC.
*
* One of the uses of getting all SCC is that you can generate a new
DAG (Directed *
* Acyclic Graph), easier to work with, in which each SCC being a
"supernode" of *
* the DAG.
* Time complexity: O(V+E)
* Notation: adj[i]: adjacency list for node i
*
* adjt[i]: reversed adjacency list for node i
*
* ord: array of vertices according to their finish
time *
* ordn: ord counter
* scc[i]: supernode assigned to i
* scc_cnt: amount of supernodes in the graph
*
*****
const int N = 2e5 + 5;

```

```

vector<int> adj[N], adjt[N];
int n, ordn, scc_cnt, vis[N], ord[N], scc[N];

```

```

//Directed Version
void dfs(int u) {
    vis[u] = 1;
    for (auto v : adj[u]) if (!vis[v]) dfs(v);
    ord[ordn++] = u;
}

```

```

void dfst(int u) {
    scc[u] = scc_cnt, vis[u] = 0;
    for (auto v : adjt[u]) if (vis[v]) dfst(v);
}

```

```

// add edge: u -> v
void add_edge(int u, int v){
    adj[u].push_back(v);
    adjt[v].push_back(u);
}

```

```

//Undirected version:
/*
int par[N];

void dfs(int u) {

```

```

    vis[u] = 1;
    for (auto v : adj[u]) if(!vis[v]) par[v] = u, dfs(v);
    ord[ordn++] = u;
}

void dfst(int u) {
    scc[u] = scc_cnt, vis[u] = 0;
    for (auto v : adj[u]) if(vis[v] and u != par[v]) dfst(v);
}

// add edge: u -> v
void add_edge(int u, int v){
    adj[u].push_back(v);
    adj[v].push_back(u);
}

*/

// run kosaraju
void kosaraju(){
    for (int i = 1; i <= n; ++i) if (!vis[i]) dfs(i);
    for (int i = ordn - 1; i >= 0; --i) if (vis[ord[i]]) scc_cnt++,
dfst(ord[i]);
}

```

## Lowest common ancestor (LCA)

```

// Lowest Common Ancestor <O(nlogn), O(logn)>
const int N = 1e6, M = 25;
int anc[M][N], h[N], rt;

// TODO: Calculate h[u] and set anc[0][u] = parent of node u for
each u

// build (sparse table)
anc[0][rt] = rt; // set parent of the root to itself
for (int i = 1; i < M; ++i)
    for (int j = 1; j <= n; ++j)
        anc[i][j] = anc[i-1][anc[i-1][j]];

// query
int lca(int u, int v) {
    if (h[u] < h[v]) swap(u, v);
    for (int i = M-1; i >= 0; --i) if (h[u]-(1<<i) >= h[v])
        u = anc[i][u];

    if (u == v) return u;

    for (int i = M-1; i >= 0; --i) if (anc[i][u] != anc[i][v])
        u = anc[i][u], v = anc[i][v];
    return anc[0][u];
}

```

## String Processing

### Knuth-Morris-Pratt (KMP)

```

// Knuth-Morris-Pratt - String Matching O(n+m)
char s[N], p[N];
int b[N], n, m; // n = strlen(s), m = strlen(p);

```

```

void kmppre() {
    b[0] = -1;
    for (int i = 0, j = -1; i < m; b[++i] = ++j)
        while (j >= 0 and p[i] != p[j])
            j = b[j];
}

```

```

void kmp() {
    for (int i = 0, j = 0; i < n; ) {
        while (j >= 0 and s[i] != p[j]) j=b[j];
        i++, j++;
        if (j == m) {
            // match position i-j
            j = b[j];
        }
    }
}

```

### Suffix Array

```

// Suffix Array O(nlogn)
// s.push('$');
vector<int> suffix_array(string &s){
    int n = s.size(), alph = 256;
    vector<int> cnt(max(n, alph)), p(n), c(n);

    for(auto c : s) cnt[c]++;
    for(int i = 1; i < alph; i++) cnt[i] += cnt[i - 1];
    for(int i = 0; i < n; i++) p[--cnt[s[i]]] = i;
    for(int i = 1; i < n; i++)
        c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);

    vector<int> c2(n), p2(n);

    for(int k = 0; (1 << k) < n; k++){
        int classes = c[p[n - 1]] + 1;
        fill(cnt.begin(), cnt.begin() + classes, 0);

        for(int i = 0; i < n; i++) p2[i] = (p[i] - (1 << k) + n)%n;
        for(int i = 0; i < n; i++) cnt[c[i]]++;
        for(int i = 1; i < classes; i++) cnt[i] += cnt[i - 1];
        for(int i = n - 1; i >= 0; i--) p[--cnt[c[p2[i]]]] = p2[i];

        c2[p[0]] = 0;
        for(int i = 1; i < n; i++){
            pair<int, int> b1 = {c[p[i]], c[(p[i] + (1 << k))%n]};
            pair<int, int> b2 = {c[p[i - 1]], c[(p[i - 1] + (1 << k))%n]};
            c2[p[i]] = c2[p[i - 1]] + (b1 != b2);
        }

        c.swap(c2);
    }
    return p;
}

```

```

// Longest Common Prefix with SA O(n)
vector<int> lcp(string &s, vector<int> &p){
    int n = s.size();
    vector<int> ans(n - 1), pi(n);
    for(int i = 0; i < n; i++) pi[p[i]] = i;
}

```

```

int lst = 0;
for(int i = 0; i < n - 1; i++){
    if(pi[i] == n - 1) continue;
    while(s[i + lst] == s[p[pi[i] + 1] + lst]) lst++;

    ans[pi[i]] = lst;
    lst = max(0, lst - 1);
}

return ans;
}

// Longest Repeated Substring O(n)
int lrs = 0;
for (int i = 0; i < n; ++i) lrs = max(lrs, lcp[i]);

// Longest Common Substring O(n)
// m = strlen(s);
// strcat(s, "$"); strcat(s, p); strcat(s, "#");
// n = strlen(s);
int lcs = 0;
for (int i = 1; i < n; ++i) if ((sa[i] < m) != (sa[i-1] < m))
    lcs = max(lcs, lcp[i]);

// To calc LCS for multiple texts use a slide window with minqueue
// The number of different substrings of a string is n*(n + 1)/2
- sum(lcs[i])

```

## Rabin-Karp

```

// Rabin-Karp - String Matching + Hashing O(n+m)
const int B = 31;
char s[N], p[N];
int n, m; // n = strlen(s), m = strlen(p)

```

```

void rabin() {
    if (n < m) return;

    ull hp = 0, hs = 0, E = 1;
    for (int i = 0; i < m; ++i)
        hp = ((hp*B)%MOD + p[i])%MOD,
        hs = ((hs*B)%MOD + s[i])%MOD,
        E = (E*B)%MOD;

    if (hs == hp) { /* matching position 0 */ }
    for (int i = m; i < n; ++i) {
        hs = ((hs*B)%MOD + s[i])%MOD;
        hhs = (hs - s[i-m]*E%MOD + MOD)%MOD;
        if (hs == hp) { /* matching position i-m+1 */ }
    }
}

```

## Z-function

```

// Z-Function - O(n)

```

```

vector<int> zfunction(const string& s){
    vector<int> z (s.size());
    for (int i = 1, l = 0, r = 0, n = s.size(); i < n; i++){
        if (i <= r) z[i] = min(z[i-l], r - i + 1);
    }
}

```

```

        while (i + z[i] < n and s[z[i]] == s[z[i] + i]) z[i]++;
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}

```

## Manacher's

```

// Manacher (Longest Palindromic String) - O(n)
int lps[2*N+5];
char s[N];

```

```

int manacher() {
    int n = strlen(s);

    string p (2*n+3, '#');
    p[0] = '^';
    for (int i = 0; i < n; i++) p[2*(i+1)] = s[i];
    p[2*n+2] = '$';

    int k = 0, r = 0, m = 0;
    int l = p.length();
    for (int i = 1; i < l; i++) {
        int o = 2*k - i;
        lps[i] = (r > i) ? min(r-i, lps[o]) : 0;
        while (p[i + 1 + lps[i]] == p[i - 1 - lps[i]]) lps[i]++;
        if (i + lps[i] > r) k = i, r = i + lps[i];
        m = max(m, lps[i]);
    }
    return m;
}

```

## Dynamic programming

### Convex hull trick

```

// Convex Hull Trick

```

```

// ATTENTION: This is the maximum convex hull. If you need the
minimum
// CHT use {-b, -m} and modify the query function.

```

```

// In case of floating point parameters swap long long with long
double
typedef long long type;
struct line { type b, m; };

```

```

line v[N]; // lines from input
int n; // number of lines
// Sort slopes in ascending order (in main):
sort(v, v+n, [](line s, line t){
    return (s.m == t.m) ? (s.b < t.b) : (s.m < t.m); });

```

```

// nh: number of lines on convex hull
// pos: position for linear time search
// hull: lines in the convex hull
int nh, pos;
line hull[N];

```

```

bool check(line s, line t, line u) {
    // verify if it can overflow. If it can just divide using long

```

```

double
    return (s.b - t.b)*(u.m - s.m) < (s.b - u.b)*(t.m - s.m);
}

```

```

// Add new line to convex hull, if possible
// Must receive lines in the correct order, otherwise it won't work
void update(line s) {
    // 1. if first lines have the same b, get the one with bigger m
    // 2. if line is parallel to the one at the top, ignore
    // 3. pop lines that are worse
    // 3.1 if you can do a linear time search, use
    // 4. add new line

```

```

    if (nh == 1 and hull[nh-1].b == s.b) nh--;
    if (nh > 0 and hull[nh-1].m >= s.m) return;
    while (nh >= 2 and !check(hull[nh-2], hull[nh-1], s)) nh--;
    pos = min(pos, nh);
    hull[nh++] = s;
}

```

```

type eval(int id, type x) { return hull[id].b + hull[id].m * x; }

```

```

// Linear search query - O(n) for all queries
// Only possible if the queries always move to the right
type query(type x) {
    while (pos+1 < nh and eval(pos, x) < eval(pos+1, x)) pos++;
    return eval(pos, x);
    // return -eval(pos, x);    ATTENTION: Uncomment for minimum CHT
}

```

```

// Ternary search query - O(logn) for each query
/*
type query(type x) {
    int lo = 0, hi = nh-1;
    while (lo < hi) {
        int mid = (lo+hi)/2;
        if (eval(mid, x) > eval(mid+1, x)) hi = mid;
        else lo = mid+1;
    }
    return eval(lo, x);
    // return -eval(lo, x);    ATTENTION: Uncomment for minimum CHT
}

```

```

// better use geometry line_intersect (this assumes s and t are not
parallel)
ld intersect_x(line s, line t) { return (t.b - s.b)/(ld)(s.m - t.m); }
ld intersect_y(line s, line t) { return s.b + s.m * intersect_x(s,
t); }
*/

```

## Longest Increasing Subsequence

```

// Longest Increasing Subsequence - O(nlogn)
//
// dp(i) = max j<i { dp(j) | a[j] < a[i] } + 1
//

```

```

// int dp[N], v[N], n, lis;

```

```

memset(dp, 63, sizeof dp);

```

```
for (int i = 0; i < n; ++i) {  
    // increasing: lower_bound  
    // non-decreasing: upper_bound  
    int j = lower_bound(dp, dp + lis, v[i]) - dp;  
    dp[j] = min(dp[j], v[i]);  
    lis = max(lis, j + 1);  
}
```