Aho-corasick

```
#define ll long long
const int ALPHABETS = 27;
int toNum(char c) {
	return c - 'a';
}
struct node {
	node *children[ALPHABETS];
	int end;
	node() : end(false) { for (int i = 0; i < ALPHABETS; i++)
children[i] = NULL; };
	node *fail;
	void insert(const char *key) {
		if (*key == 0)
			end = 1;
		else
		{
			int next = toNum(*key);
			if (children[next] == NULL)
				children[next] = new node();
			children[next]->insert(key + 1);
		}
	}
};
void FFC(node* root) {
	queue<node*> q;
	root->fail = root;
	q.push(root);

	while (!q.empty()) {
		node* here = q.front(); q.pop();
		for (int edge = 0; edge< ALPHABETS; edge++) {
			node *child = here->children[edge];
			if (!child) continue;
			if (here == root) child->fail = root;
			else {
				node *t = here->fail;
				while (t != root && t->children[edge] ==
NULL)
					t = t->fail;
				if (t->children[edge]) t = t->children[edge];
				child->fail = t;
			}
			child->end += child->fail->end;
			q.push(child);
```

```
            }
        }
}
int aho(const string &s, node *root) {
        int ret = 0;
        node *state = root;
        int size = s.size();
        for (int i = 0; i < size; i++) {
                int chr = toNum(s[i]);
                while (state != root && state->children[chr] == NULL)
                        state = state->fail;
                if (state->children[chr]) state = state->children[chr];
                ret += state->end;
        }
        return ret;
}

// example of main
// return the number of string including { root } in dna

node *root = new node();
string dna, marker;
cin >> dna >> marker;
list<string> lis;
for (int i = 0; i< m; i++)
        for (int j = i; j < m; j++) {
                reverse(marker.begin() + i, marker.begin() + j + 1);
                lis.push_back(marker);
                reverse(marker.begin() + i, marker.begin() + j + 1);
        }

unique(lis.begin(), lis.end());
for (auto i : lis) {
        root->insert(i.c_str());
}
FFC(root);
// RETURN //
return aho(dna, root);
*/
```

Convex Hull Trick

```
#define MN 1000001
int i, j, n, x[MN], dn, L[MN];
long long A, B, C, dp[MN], S[MN];
char buffer[5 * MN];
double d[MN];
inline double g(int a) {
        return (double)(dp[i] - dp[a] + A*(S[i] * S[i] - S[a] * S[a])) / (2
* A*(S[i] - S[a]));
}
int main() {
        scanf("%d%lld%lld%lld\n", &n, &A, &B, &C);
        gets(buffer + 1);
        int xn = 1;
        for (i = 1; buffer[i]; i++) {
                if (buffer[i] == ' ') xn++;
                else x[xn] = x[xn] * 10 + (buffer[i] - '0');
        }
        d[dn = 1] = -999999999999;
        for (i = j = 1; i <= n; i++) {
                S[i] = S[i - 1] + x[i];
                while (j + 1 <= dn && d[j + 1] <= S[i]) j++;
                if (j > dn) j = dn;
                dp[i] = dp[L[j]] + A*(S[i] - S[L[j]])*(S[i] - S[L[j]]) + C;
                while (dn >= 2 && g(L[dn]) <= d[dn]) dn--;
                L[++dn] = i;
                d[dn] = g(L[dn - 1]);
        }
        printf("%lld", dp[n] + B*S[n]);
}
```

Euler Path

```
void EulerTour(list<int>::iterator i, int u) {
        for (int j = 0; j<adj[u].size; j++) {
                ii &v = adj[u][j];
                if (v.second) {
                        v.second = 0;
                        for (int k = 0; k<adj[v.first].size(); k++) {
                                ii &uu = adj[v.first][k];
                                if (uu.first == u && uu.second) {
                                        uu.second = 0;
                                        break;
                                }
                        }
                        EulerTour(cycle.insert(i, u), v.first);
                }
```

```
        }
}
/* Usage
cyc.clar();
EulerTour(cyc.begin(), src);
for (auto it : cyc) {
printf("%d\n", (*it));
}

*/
```

Factorization

```
long long modmul(long long a, long long b, long long m)  /* (a*b)%m */
{
        long long y = (long long)((double)a*(double)b / (double)m + 0.5) *
m;
        long long x = a * b, r = x - y;
        return (r < 0) ? r + m : r;
}

long long modexp(long long a, long long e, long long m)  /* (a^e)%m */
{
        if (!e) return 1;
        long long b = modexp(a, e / 2, m);
        return (e & 1) ? modmul(modmul(b, b, m), a, m) : modmul(b, b, m);
}

bool isprime(long long n)  /* for n < 5689719352694202437032697232 1 */
{
        if (n <= 1) return false;
        if (n <= 3) return true;
        static long long a[] = { 2,3,5,7,11,13,17,19,23,29,31 };
        long long s = 0, d = n - 1;
        while (d % 2 == 0) d /= 2, ++s;
        for (int i = 0; i < 11; ++i)
        {
                if (n == a[i]) return true;
                long long x = modexp(a[i], d, n);
                if (x != 1 && x != n - 1)
                {
                        for (int r = 1; r < s; ++r)
                        {
                                x = modmul(x, x, n);
                                if (x == 1) return false;
                                if (x == n - 1) break;
                        }
                        if (x != n - 1) return false;
                }
        }
        return true;
}

long long llrand()
{
        return ((long long)rand() << 32) + rand();
}

long long rho(long long n)
{
        long long d, c = llrand() % n, x = llrand() % n, xx = x;

        if (n % 2 == 0) return 2;
        do {
                x = (modmul(x, x, n) + c) % n;
                xx = (modmul(xx, xx, n) + c) % n;
                xx = (modmul(xx, xx, n) + c) % n;
                d = gcd(abs(x - xx), n);
        } while (d == 1);
        return d;
}

vector<long long> v;
void factor(long long n)
{
        if (n == 1) return;
        if (isprime(n)) { v.push_back(n); return; }
        long long d = rho(n);
        factor(d);
        factor(n / d);
}

//Usage
// factor(N);
```

Function Cycle Detection

```
ii floydCycleFinding(int x) {
        int a = f(x), b = f(f(x));
        while (a != b) { a = f(a); b = f(f(a)); }
```

```
	int mu = 0, b = x;
	while (a != b) { a = f(a); b = f(b); mu++; }
	int lambda = 1; b = f(a);
	while (a != b) { b = f(b); lambda++; }
	return ii(mu, lambda);
}
```

Geometry

```
typedef long long ll;
struct Point {
	ll x, y;
};
struct Line {
	Point p1, p2;
};

// Note that Lines are either vertical or horizontal and variable type is NOT reference
ll get_dist(Line l, Line r) {
	if (l.p1.x > l.p2.x) swap(l.p1, l.p2);
	if (l.p1.y > l.p2.y) swap(l.p1, l.p2);
	if (r.p1.x > r.p2.x) swap(r.p1, r.p2);
	if (r.p1.y > r.p2.y) swap(r.p1, r.p2);
	if (r.p1.x == r.p2.x) swap(l, r);
	const ll INF = 1e15;
	ll res = INF;
	if (l.p1.y == l.p2.y) {
		assert(r.p1.y == r.p2.y);
		if (!(l.p2.x < r.p1.x || r.p2.x < l.p1.x)) res = min(res, abs(l.p1.y - r.p1.y));
	}
	else if (r.p1.x == r.p2.x) {
		assert(l.p1.x == l.p2.x);
		if (!(l.p2.y < r.p1.y || r.p2.y < l.p1.y)) res = min(res, abs(l.p1.x - r.p1.x));
	}
	else {
		assert(l.p1.x == l.p2.x && r.p1.y == r.p2.y);
		if (r.p1.x <= l.p1.x && l.p1.x <= r.p2.x) res = min({ res, abs(r.p1.y - l.p1.y), abs(r.p1.y - l.p2.y) });
		if (l.p1.y <= r.p1.y && r.p1.y <= l.p2.y) res = min({ res, abs(l.p1.x - r.p1.x), abs(l.p2.x - r.p1.x) });
		if (r.p1.x <= l.p1.x && l.p1.x <= r.p2.x &&
			l.p1.y <= r.p1.y && r.p1.y <= l.p2.y) res = 0;
	}
	if (res < INF) res = res * res;
	for (auto &i : { l.p1, l.p2 })
		for (auto &j : { r.p1, r.p2 })
			res = min(res, (i.x - j.x) * (i.x - j.x) + (i.y - j.y) * (i.y - j.y));
	return res;
}


ll cross(const Point &O, const Point &A, const Point &B) {
	return (A.x - O.x) * (B.y - O.y) - (A.y - O.y) * (B.x - O.x);
}

// param:: vector of Point with x,y coordinates in long long int, P.size >= 3
// return:: convex_hull with x, y coordinates in long long int
// the first and the last element is SAME
typedef long long ll;
vector<Point> convex_hull(const vector<Point> &points)
{
	int k = 0;
	vector<Point> result(2 * points.size());
	sort(points.begin(), points.end(), [](Point p, Point q) { return p.second > q.second || ((!(p.second < q.second) && p.first < q.first)); });
	for (int i = 0; i < points.size(); ++i)
	{
		while (k >= 2 && cross(result[k - 2], result[k - 1], points[i]) <= 0)  k--;
		result[k++] = points[i];
	}
	for (int i = points.size() - 2, t = k + 1; i >= 0; i--)
	{
		while (k >= t && cross(result[k - 2], result[k - 1], points[i]) <= 0) k--;
		result[k++] = points[i];
	}
	result.resize(k); //Circular - result[0] == result[k-1]
	return result;
}
```

Graph Theory

```
// O(V+E)
```

```
vector<pii> edges, vector<int> vertexes;
vector<int> dfs_num, dfs_low, dfs_parent; vector<bool> chk;
const int UNVISITED = -1;
void dfs(int u) {
	dfs_low[u] = dfs[num] = dfsCnt++; //dfs_low[u]<=dfs_num[u]
	for (int j = 0; j<(int)adj[u].size(); j++) {
		pii v = adj[u][j];
		if (dfs_num[v.first] == UNVISITED) {
			dfs_parent[v.first] = u;
			if (u == dfsRoot) rootChildren++;

			dfs(v.first);

			if (dfs_low[v.first] >= dfs_num[u])
				chk[u] = true;
			if (dfs_low[v.first] > dfs_num[u])
				edge.push_back({ u, v.first });
			dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
		}
		else if (v.first != dfs_parent[u])
			dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);

	}
}

void findArticulation() {
	dfsCnt = 0;
	dfs_num.assign(V, UNVISITED);
	dfs_low.assign(V, 0);
	dfs_parent.assign(V, 0);
	chk.assign(V, false);
	for (int i = 0; i<V; i++) {
		if (dfs_num[i] == UNVISITED) {
			dfsRoot = i; rootChildren = 0; findArticulation(i);
			chk[i] = (rootChildren > 1);
		}
	}

}


// O(E V^0.5)
size_t q;

namespace HopcroftKarp {
	const size_t &INF = numeric_limits<size_t>::max();
	const size_t &NIL = 0;

	vector<size_t> pairL, pairR, level;
	queue<size_t> que;
	const vector<vector<size_t>> *graph;
	size_t n, totalMatching;

	inline bool bfs() {
		for (size_t left = 1; left <= n; left++) {
			if (pairL[left] == NIL) {
				level[left] = 0;
				que.emplace(left);
			}
			else level[left] = INF;
		}
		level[NIL] = INF;

		while (que.size()) {
			size_t left = que.front();
			que.pop();

			if (level[left] >= level[NIL]) continue;

			for (size_t right : graph->at((left - 1) % q + 1)) {
				size_t prevPair = pairR[right];

				if (level[prevPair] == INF) {
					level[prevPair] = level[left] + 1;
					que.emplace(prevPair);
				}
			}
		}

		return level[NIL] != INF;
	}

	bool dfs(size_t left) {
		if (left == NIL) return true;

		for (size_t right : graph->at((left - 1) % q + 1)) {
			size_t &traceLink = pairR[right];

			if (level[traceLink] == level[left] + 1 &&
dfs(traceLink)) {
				traceLink = left;
				pairL[left] = right;
				return true;
```

```
                }
            }

            level[left] = INF;
            return false;
        }

        size_t maximumMatching(const vector<vector<size_t>> &graph, size_t
n, size_t m) {
            HopcroftKarp::graph = &graph;
            HopcroftKarp::n = n;

            level.resize(n + 1);
            pairL.resize(n + 1);
            fill(pairL.begin(), pairL.end(), NIL);
            pairR.resize(m + 1);
            fill(pairR.begin(), pairR.end(), NIL);
            totalMatching = 0;

            while (bfs()) {
                for (size_t left = 1; left <= n; left++) {
                    if (pairL[left] == NIL && dfs(left)) {
                        totalMatching++;
                    }
                }
            }

            return totalMatching;
        }
}

/* Usage
size_t n, m, p, a;
scanf("%zu%zu", &n, &m);
vector<vector<size_t>> graph(n + 1);

q = n;
for (size_t i = 1; i <= n; i++) {
scanf("%zu", &p);
while (p--) {
scanf("%zu", &a);
graph[i].emplace_back(a);
}
}

printf("%zu", HopcroftKarp::maximumMatching(graph, n + n, m));
*/
```

Kirchhoff - Number of Spanning Trees

```
// # of Spanning Tree
long long count_spantree(vector<int> graph[], int size) {
        int i, j;
        vector<vector<double> > matrix(size - 1);
        for (i = 0; i < size - 1; i++) {
                matrix[i].resize(size - 1);
                for (j = 0; j < size - 1; j++)
                        matrix[i][j] = 0;
                for (j = 0; j < graph[i].size(); j++) {
                        if (graph[i][j] < size - 1) {
                                matrix[i][graph[i][j]]--;
                                matrix[i][i]++;
                        }
                }
        }
        return (long long)(mat_det(matrix, size - 1) + 0.5);
}
```

KMP

```
#define MX 100000
char T[MX], P[MX]; // T - sentece, P - word
int b[MX], n, m;  // b- failure function, len(T) = n , len(P) = m;

void kmpPreprocess() {
        int i = 0, j = -1; b[0] = -1;
        while (i<m) {
                while (j >= 0 && P[i] != P[j]) j = b[j];
                i++; j++;
                b[i] = j;
        }
}

void kmpSearch() {
        int i = 0, j = 0;
        while (i<n) {
                while (j >= 0 && T[i] != P[j]) j = b[j];
                i++; j++;
                if (j == m) {
                        printf("Found at %d\n", i - j);
                        j = b[j];
                }
        }
```

```
}
```

Kruskal

```
// O(ElogV)
// Note that the optimum is NOT UNIQUE
// For minimum SUBGRAPH graph problem, note that it may form cycle.
// For minimum FOREST problem, do it until # of connected components woulud become # of forests
// Minimax path problem (path between i and j) can be solved with MST!
#define MX 10001
int p[MX], rank[MX];
inline int find(short x) {
	return p[x] == x ? x : p[x] = find(p[x]);
}
inline void unite(short x, short y) {
	x = find(x), y = find(y);
	if (x == y) return;

	if (rank[x] > rank[y]) swap(x, y);
	p[x] = y;

	if (rank[x] == rank[y]) ++rank[y];
}
int main() {
	int V, s, e, i, u, v;
	int E, t, ans = 0;

	scanf("%d %d", &V, &E);

	for (i = 1; i <= V; ++i) p[i] = i;
	vector<pair<int, pair<int, int> > > list;

	for (i = 0; i<E; ++i) {
		scanf("%d %d %d", &s, &e, &w); //start, end, w
		list.push_back(make_pair(w, make_pair(s, e)));
	}

	sort(list.begin(), list.end());

	for (i = 0; i<list.size(); ++i) {
		u = list[i].second.first;
		v = list[i].second.second;

		u = find(u);
		v = find(v);
		if (u != v) {
			unite(u, v);
			ans += list[i].first;
		}
	}

	printf("%d\n", ans);
}
```

Lazy Propagation

```
typedef long long ll;

// h : 2^h>N 중 가장 작은 h, tree_size : Segment Tree의 총 노드 수
// tree : Segment Tree, v : 입력 배열 -> tree size : 4 * N
// node : Segment Tree에서 현재 노드 번호( 1 - base )
// start : 현재 노드가 포함하는 범위의 시작, end : 현재 노드가 포함하는 범위의 끝 ( 1 - base )
// left : update하는 구간의 시작, right : update하는 구간의 끝 ( 1 - base )
int get_height(int n) {
	int cnt = 0, t = 1;
	while (t < n) {
		cnt++;
		t *= 2;
	}
	return cnt;
}

long long init(vector<long long> &v, vector<long long> &tree, int node, int start, int end) {
	if (start == end) {
		return tree[node] = v[start];
	}
	else {
		return tree[node] = init(v, tree, node * 2, start, (start + end) / 2) + init(v, tree, node * 2 + 1, (start + end) / 2 + 1, end);
	}
}

void update_lazy(vector<long long> &tree, vector<long long> &lazy, int node, int start, int end) {
	if (lazy[node] != 0) {
		tree[node] += (end - start + 1)*lazy[node];
```

```
			if (start != end) {
				lazy[node * 2] += lazy[node];
				lazy[node * 2 + 1] += lazy[node];
			}
			lazy[node] = 0;
		}
}

void update(vector<long long> &tree, vector<long long> &lazy, int node, int start, int end, int left, int right, long long val) {
		update_lazy(tree, lazy, node, start, end);
		if (left > end || right < start) {
			return;
		}
		if (left <= start && end <= right) {
			tree[node] += (end - start + 1)*val;
			if (start != end) {
				lazy[node * 2] += val;
				lazy[node * 2 + 1] += val;
			}
			return;
		}

		update(tree, lazy, node * 2, start, (start + end) / 2, left, right, val);
		update(tree, lazy, node * 2 + 1, (start + end) / 2 + 1, end, left, right, val);
		tree[node] = tree[node * 2] + tree[node * 2 + 1];
}

long long sum(vector<long long> &tree, vector<long long> &lazy, int node, int start, int end, int left, int right) {
		update_lazy(tree, lazy, node, start, end);
		if (left > end || right < start) {
			return 0;
		}
		if (left <= start && end <= right) {
			return tree[node];
		}

		return sum(tree, lazy, node * 2, start, (start + end) / 2, left, right) + sum(tree, lazy, node * 2 + 1, (start + end) / 2 + 1, end, left, right);
}
```

LCA

```
#define MX 1234567

vector<int> arr[MX];
int depth[MX], parent[MX][18]; // 2^18 should be larger than MX
void dfs(int n)
{
		for (int e = 0; e<arr[n].size(); e++)
		{
			int next = arr[n][e];
			if (depth[next] == -1)
			{
				depth[next] = depth[n] + 1;
				parent[next][0] = n;
				dfs(next);
			}
		}
}
int main(void)
{
		memset(parent, -1, sizeof(parent));
		memset(depth, -1, sizeof(depth));
		int n;
		scanf("%d", &n);
		for (int e = 0; e<n - 1; e++)
		{
			int a, b;
			scanf("%d%d", &a, &b);
			arr[a].push_back(b);
			arr[b].push_back(a);
		}
		depth[1] = 0;
		dfs(1);
		for (int e = 0; e<17; e++)
		{
			for (int p = 2; p <= n; p++)
			{
				if (parent[p][e] != -1)
				{
					parent[p][e + 1] = parent[parent[p][e]][e];
				}
			}
		}
		int m;
		scanf("%d", &m);
```

```
	for (int e = 0; e<m; e++)
	{
		int a, b;
		scanf("%d%d", &a, &b);
		if (depth[a]<depth[b])
		{
			int tmp = a;
			a = b;
			b = tmp;
		}
		int diff = depth[a] - depth[b];
		for (int p = 0; diff; p++)
		{
			if (diff % 2) a = parent[a][p];
			diff /= 2;
		}
		if (a != b)
		{
			for (int p = 17; p >= 0; p--)
			{
				if (parent[a][p] != -1 && parent[a][p] != 
parent[b][p])
				{
					a = parent[a][p];
					b = parent[b][p];
				}
			}
			a = parent[a][0];
		}
		printf("%d\n", a);
	}
}
```

LIS

```
typedef pair<int, int> ii;

struct mycomp {
	bool operator() (const ii &l, const ii &r) const {
		return l.second < r.second;
	}
};
vector<ii> LIS(vector<ii> v) {
	map < ii, int, mycomp> m;
	map < ii, int, mycomp>::iterator k, l;
	vector<ii> res;

	const int N = v.size();
	vector<int> pre(N, -1);

	for (int i = 0; i < N; i++) {
		if (m.insert({ v[i], i }).second) {
			k = m.find(v[i]);
			l = k; k++;
			if (l == m.begin()) {
				pre[i] = -1;
			}
			else {
				l--;
				pre[i] = l->second;
			}
			if (k != m.end()) {
				m.erase(k);
			}
		}
	}
	k = m.end(); k--;
	int j = k->second;
	while (j != -1) {
		res.push_back(v[j]);
		j = pre[j];
	}
	reverse(res.begin(), res.end());
	return res;
}

int main(void) {
	int N; scanf("%d", &N);

	vector<ii> v;
	for (int i = 0; i < N; i++) {
		int a, b; scanf("%d %d", &a, &b);
		v.push_back({ a, b });
	}
	sort(v.begin(), v.end());

	auto r = LIS(v);
	auto it = r.begin();
	printf("%d\n", v.size() - r.size());
	for (auto e : v) {
		if (e != (*it)) {
```

```
				printf("%d\n", e.first);
			}
			else {
				it++;
			}
		}

}
```

Math

```
/* HCN
* number                divisors        factorization
* 12                    6               2^2*3
* 120                   16              2^3*3*5
* 1260                  36              2^2*3^2*5*7
* 10080                 72              2^5*3^2*5*7
* 110880                144             2^5*3^2*5*7*11
* 1081080               256             2^3*3^3*5*7*11*13
* 10810800              480             2^4*3^3*5^2*7*11*13
* 110270160             800             2^4*3^4*5*7*11*13*17
* 1102701600            1440            2^5*3^4*5^2*7*11*13*17
* 10475665200           2400            2^4*3^4*5^2*7*11*13*17*19
* 128501493120          4096            2^7*3^3*5*7*11*13*17*19*23
* 1124388064800         6912            2^5*3^3*5^2*7^2*11*13*17*19*23
* 13492656777600        11520           2^7*3^4*5^2*7^2*11*13*17*19*23
* 130429015516800       18432           2^7*3^3*5^2*7^2*11*13*17*19*23*29
* 1010824870255200      27648  2^5*3^3*5^2*7^2*11*13*17*19*23*29*31
* 10108248702552000     43008  2^6*3^3*5^3*7^2*11*13*17*19*23*29*31
* 121298984430624000    69120  2^8*3^4*5^3*7^2*11*13*17*19*23*29*31
* 800573297242118400    93312  2^8*3^5*5^2*7^2*11^2*13*17*19*23*29*31
* 10^18*/

// Area of Convex Hull = 1/2 * abs( sum ( x1*y2-y1*x2))
//
//  Catalan Number Cat(N) = 2N C N / (N+1) , Cat(N+1) =
(2N+2)(2N+1)/(N+2)(N+1) * Cat(N)
density of prime numbers : x / log x (lim x -> INF)
*/
bool isPrime(int n);
bool isPrime(int n, vector<int> v);
vector<int> getPrimes(int n);
vector<pair<int, int> > factorize(int n);
vector<pair<int, int> > factorize(int n, vector<int> v);
//Complexity : O(N/ logN + N ^ 0.75) for worst case (which means
when n is prime number)
// N= 10^9 -> 5 * 10^7
// N= 10^10 -> 4.6 * 10^8
// N= 10^11 -> 4.1 * 10^9
bool isPrime(int n)
{
	return isPrime(n, getPrimes((int)sqrt(n)));
}
//Complexity : O(N) for worst case (which means when n is prime
number)
bool isPrime(int n, const vector<int> v)
{
	for (auto now : v) {
		if (n % now == 0)
			return false;
	}
	return true;
}
//Verified in range of (0, 10^6) at least by BOJ
//Complexity : O(N ^1.5)
vector<int> getPrimes(int N)
{
	vector<int> ret;
	if (N >= 2)
		ret.push_back(2);
	if (N >= 3)
		ret.push_back(3);
	int i, j, k;
	bool ctn = true;
	int mid_point = (int)sqrt(N - 1) / 6 + 1;
	for (i = 1; ctn && i <= mid_point; i++) {
		for (j = -1; j <= 1; j += 2) {
			int now = i * 6 + j;
			if (now > sqrt(N)) {
				ctn = false;
				break;
			}
			bool flag = true;
			for (auto here : ret) {
				if (now % here == 0) {
					flag = false;
					break;
				}
			}
			if (flag) {
				ret.push_back(now);
			}
		}
```

```
	}
	ctn = true;
	int ret_sqrt_cnt = (int)ret.size();
	for (i = mid_point - 2; ctn && i <= (N - 1) / 6 + 1; i++) {
		for (j = -1; j <= 1; j += 2) {
			int now = i * 6 + j;
			if (now <= ret[ret_sqrt_cnt - 1])
				continue;
			if (now > N) {
				ctn = false;
				break;
			}
			bool flag = true;
			for (k = 0; k < ret_sqrt_cnt; k++) {
				if (now % ret[k] == 0) {
					flag = false;
					break;
				}
			}
			if (flag) {
				ret.push_back(now);
			}
		}
	}
	return ret;
}
//return <prime number, power_cnt>
//ex) N = 12 / return vector<pair<2, 2>, pair<3, 1>>
vector<pair<int, int> > factorize(int N)
{
	auto primes = getPrimes(sqrt(N) + 5);
	return factorize(N, primes);
}
vector<pair<int, int> > factorize(int N, vector<int> primes)
{
	vector<pair<int, int> > ret;
	for (auto p : primes) {
		int c = 0;
		while (N % p == 0) {
			N /= p;
			c++;
		}
		if (c > 0)
			ret.push_back(make_pair(p, c));
	}
	if (N > 1)
		ret.push_back(make_pair(N, 1));
	return ret;
}
//extended gcd function
//returns gcd(a, b) by value,
//and x, y by reference that satisfies ax + by = gcd(a, b)
//Complexity : 12log2/(pi^2) log a + O(1) approximated by "0.85loga + O(1)
in average case ",
// "O(logb) in worst case" when a>=b
template <typename T>
T xGCD(T a, T b, T* x, T* y)
{
	if (a == 0) {
		*x = 0;
		*y = 1;
		return b;
	}
	T x1, y1;
	T gcd = xGCD(b % a, a, &x1, &y1);
	*x = y1 - (b / a) * x1;
	*y = x1;
	return gcd;
}
//m SHOULD BE PRIME NUMER!! It doesn't make any assertion!
//returns multiplicative inverse by modulo
//ex) mul_inverse_modulo(3, 11) = 4 since 3 * 4 is equivalent with 1 by
modulo 11
//Complexity : O( (log m)^2 )
template <typename T>
T mul_inverse_modulo(T a, T m)
{
	T x, y;
	xGCD(a, m, &x, &y);
	return x;
}
//returns ( n C r ) % MOD without caching in
template <typename T>
T combination(T n, T r, T MOD)
{
	if (r > n / 2)
		r = n - r;
	T ret = 1;
	for (T i = n; i >= n - r + 1; i--) {
		ret *= i;
		ret %= MOD;
	}
	for (T i = r; i >= 1; i--) {
		ret *= mul_inverse_modulo(i, MOD);
```

```
            ret %= MOD;
        }
        return ret;
}
//chinese_remainder_Theorem
/* if there is a possibility of k being very big, then prime factorize
m[i],
* find modular inverse of 'temp' of each of the factors
* 'k' equals to the multiplication ( modular mods[i] ) of modular inverses
*/
template <typename type>
type chinese_remainder(const vector<type>& r, const vector<type>&
        mods)
{
        type M = 1;
        for (size_t i = 0; i < size_t(mods.size()); i++)
                M *= mods[i];
        vector<type> m, s;
        for (size_t i = 0; i < size_t(mods.size()); i++) {
                m.push_back(M / mods[i]);
                type temp = m[i] % mods[i];
                type k = 0;
                while (true) {
                        if ((k * temp) % mods[i] == 1)
                                break;
                        k++;
                }
                s.push_back(k);
        }
        long long ret = 0;
        for (int i = 0; i < int(s.size()); i++) {
                ret += ((m[i] * s[i]) % M * r[i]) % M;
                if (ret >= M)
                        ret -= M;
        }
        return ret;
}

// Lucas Theorem
//
// n = sigma n_i p^i, k = sigma k_i p^i
// n C k === pi n_i C k_i  (mod p)
vector<ll> get_digits(ll n, ll b) {
        vector<ll> d;
        while (n) {
                d.push_back(n%b);
                n /= b;
        }
        return d;
}
ll lucas_theorem(ll n, ll k, ll p) {
        ll ret = 1;
        vector<ll> nd = get_digits(n, p), kd = get_digits(k, p);
        for (int i = 0; i < max(nd.size(), kd.size()); i++) {
                ll nn, kk;
                if (i < nd.size())
                        nn = nd[i];
                else
                        nn = 0;
                if (i < kd.size())
                        kk = kd[i];
                else
                        kk = 0;

                if (nn < kk)
                        return 0;
                ret = (ret * binomial(nn, kk, p) % p);
        }
        return ret;
}
```

Matrix

```
#define MAX_N 3              // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {
        // input: N, Augmented Matrix Aug, output: Column vector X, the
answer
        int i, j, k, l; double t;

        for (i = 0; i < N - 1; i++) {          // the forward elimination
phase
                l = i;
                for (j = i + 1; j < N; j++)      // which row has largest
column value
                        if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
                                l = j;                                  //
remember this row l

                                                        // swap this pivot row,
reason: minimize floating point error
```

```
            for (k = i; k <= N; k++)          // t is a temporary
double variable
                t = Aug.mat[i][k], Aug.mat[i][k] = Aug.mat[l][k],
Aug.mat[l][k] = t;
            for (j = i + 1; j < N; j++)     // the actual forward
elimination phase
                for (k = N; k >= i; k--)
                    Aug.mat[j][k] -= Aug.mat[i][k] *
Aug.mat[j][i] / Aug.mat[i][i];
        }

        ColumnVector Ans;                       // the back substitution
phase
        for (j = N - 1; j >= 0; j--) {                      // start from
back
            for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] *
Ans.vec[k];
            Ans.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; // the
answer is here
        }
        return Ans;
}

/* Usage
AugmentedMatrix Aug;
Aug.mat[0][0] = 1; Aug.mat[0][1] = 1; Aug.mat[0][2] = 2; Aug.mat[0][3] = 9;
Aug.mat[1][0] = 2; Aug.mat[1][1] = 4; Aug.mat[1][2] = -3; Aug.mat[1][3] =
1;
Aug.mat[2][0] = 3; Aug.mat[2][1] = 6; Aug.mat[2][2] = -5; Aug.mat[2][3] =
0;

ColumnVector X = GaussianElimination(3, Aug);
printf("X = %.1lf, Y = %.1lf, Z = %.1lf\n", X.vec[0], X.vec[1], X.vec[2]);

return 0;
*/

double det(int n, double mat[10][10])
{
        int c, subi, i, j, subj;
        double submat[10][10];
        if (n == 2)
            return((mat[0][0] * mat[1][1]) - (mat[1][0] * mat[0][1]));
        else {
            for (c = 0; c < n; c++) {
                subi = 0;
                for (i = 1; i < n; i++) {
                    subj = 0;
                    for (j = 0; j < n; j++) {
                        if (j == c) continue;
                        submat[subi][subj] = mat[i][j];
                        subj++;
                    }
                    subi++;
                }
                d = d + (pow(-1, c) * mat[0][c] * det(n - 1,
submat));
            }
        }
        return d;
}
```

MCMF

```
typedef int cap_t;
typedef int cost_t;
typedef pair<cost_t, int> pq_t;
bool isZeroCap(cap_t cap)
{
        return cap == 0;
}
const int INF = 987654321;
const cap_t CAP_MAX = INF;
const cost_t COST_MAX = INF;
struct edge_t {
        int target;
        cap_t cap;
        cost_t cost;
        int rev;
};
int n;
vector<vector<edge_t> > graph;
vector<cost_t> pi;
vector<cost_t> dist;
vector<cap_t> mincap;
vector<int> from, v;
void init(int _n)
{
        n = _n;
        graph.clear();
        graph.resize(n);
        pi.clear();
        pi.resize(n);
```

```
		dist.resize(n);
		mincap.resize(n);
		from.resize(n);
		v.resize(n);
	}
	void addEdge(int a, int b, cap_t cap, cost_t cost)
	{
		edge_t forward = { b, cap, cost, (int)graph[b].size() };
		edge_t backward = { a, 0, -cost, (int)graph[a].size() };
		graph[a].push_back(forward);
		graph[b].push_back(backward);
	}
	bool dijkstra(int s, int t)
	{ // Modified Dijkstra
		priority_queue<pq_t, vector<pq_t>, greater<pq_t> > pq;
		fill(dist.begin(), dist.end(), COST_MAX);
		for (int i = 0; i < n; i++) {
			from[i] = -1;
			v[i] = 0;
		}
		dist[s] = 0;
		mincap[s] = CAP_MAX;
		pq.push(make_pair(dist[s], s));
		while (!pq.empty()) {
			int cur = pq.top().second;
			pq.pop();
			if (v[cur])
				continue;
			v[cur] = 1;
			if (cur == t)
				continue;
			for (int k = 0; k < graph[cur].size(); k++) {
				edge_t edge = graph[cur][k];
				int next = edge.target;
				if (v[next])
					continue;
				if (isZeroCap(edge.cap))
					continue;
				cost_t potCost = dist[cur] + edge.cost - pi[next] +
pi[cur];
				if (dist[next] <= potCost)
					continue;
				dist[next] = potCost;
				mincap[next] = min(mincap[cur], edge.cap);
				from[next] = edge.rev;
				pq.push(make_pair(dist[next], next));
			}
		}
		if (dist[t] == COST_MAX)
			return false;
		for (int i = 0; i < n; i++) {
			if (dist[i] == COST_MAX)
				continue;
			pi[i] += dist[i];
		}
		return true;
	}
	pair<cap_t, cost_t> solve(int source, int sink)
	{
		cap_t total_flow = 0;
		cost_t total_cost = 0;
		while (dijkstra(source, sink)) { // use SPFA in case of negative edges
			cap_t f = mincap[sink];
			total_flow += f;
			for (int p = sink; p != source;) {
				edge_t& backward = graph[p][from[p]];
				edge_t& forward =
graph[backward.target][backward.rev];
				forward.cap -= f;
				backward.cap += f;
				total_cost += forward.cost * f;
				p = backward.target;
			}
		}
		return make_pair(total_flow, total_cost);
	}

	struct SPFA {
		vi dist(n, INF); dist[S] = 0;
		queue<int> q; q.push(S);
		vi in_queue(n, 0); in_queue[S] = 1;

		while (!q.empty()) {
			int u = q.front(); q.pop(); in_queue[u] = 0;
			for (j = 0; j < (int)AdjList[u].size(); j++) { // all
outgoing edges from u
				int v = AdjList[u][j].first, weight_u_v =
AdjList[u][j].second;
				if (dist[u] + weight_u_v < dist[v]) { // if can
relax
					dist[v] = dist[u] + weight_u_v; // relax
					if (!in_queue[v]) { // add to the queue only
if it's not in the queue
```

```
                    q.push(v);
                    in_queue[v] = 1;
                }
            }
        }
    }

    //return dist
}
```

Network Flow

```
/* L-R Flow
* for each edge a->b whose capacity is [l, r]
* 1) a->b with capacity l, cost -1 and with capacity r-l, cost 0
* 2) new source -> b with capacity l, a -> new sink with capacity l, a->b
with capacity r-l, sink->source with capacity INF
* and check that the Maximum Flow is eqaul to the summation of 'l's
*
* actual flow - do maxflow(oldsrc, olddst)
*/
// O(min(fE, V^2E)) / O( min( V^(2/3)E, E^(3/2)) with UNIT capacity!
struct Dinic {
    typedef long long flow_t;
    struct Edge {
        int dest;
        int inv;
        flow_t res;
    };

    vector<vector<Edge>> adj;
    vector<int> level, start;

    Dinic(int n) : adj(n), level(n), start(n) {}

    void addEdge(int here, int there, flow_t cap, flow_t caprev = 0) {
        Edge forward = { there, adj[there].size(), cap };
        Edge backward = { here, adj[here].size(), caprev };
        adj[here].push_back(forward);
        adj[there].push_back(backward);
    }

    bool assignLevel(int source, int sink) {
        fill(level.begin(), level.end(), 0);
        queue<int> q;
        q.push(source);
        level[source] = 1;
        while (!q.empty() && level[sink] == 0) {
            int here = q.front();
            q.pop();
            for (Edge &edge : adj[here]) {
                int next = edge.dest;
                if (level[next] == 0 && edge.res > 0) {
                    level[next] = level[here] + 1;
                    q.push(next);
                }
            }
        }
        return level[sink] != 0;
    }

    flow_t blockFlow(int here, int sink, flow_t flow) {
        if (here == sink) return flow;
        for (int &i = start[here]; i < adj[here].size(); ++i) {
            Edge &edge = adj[here][i];
            if (level[edge.dest] != level[here] + 1 || edge.res
== 0) continue;
            flow_t res = blockFlow(edge.dest, sink, min(flow,
edge.res));
            if (res > 0) {
                edge.res -= res;
                adj[edge.dest][edge.inv].res += res;
                return res;
            }
        }
        return 0;
    }

    flow_t solve(int source, int sink) {
        flow_t ret = 0;
        while (assignLevel(source, sink)) {
            fill(start.begin(), start.end(), 0);
            while (flow_t flow = blockFlow(source, sink,
numeric_limits<flow_t>::max()))
                ret += flow;
        }
        return ret;
    }
};

// O(min(fE, VE^2))
```

```
struct EdmondKarp {
	const int INF = 987654321;

	int min(int a, int b) {
		return a<b ? a : b;
	}
	pair<int, vector<int>> BFS(const vector<vector<int>> &cap, const
vector<vector<int>> &graph,
		vector<vector<int>> &flow, const int src, const int sink) {
		vector<int> prv(graph.size(), -1);
		vector<int> M(graph.size(), -1);
		prv[src] = -2; M[src] = INF;

		queue<int> q; q.push(src);

		while (!q.empty()) {
			int u = q.front(); q.pop();
			for (int v : graph[u]) {
				if (cap[u][v] - flow[u][v] > 0 && prv[v] == -
1) {
					prv[v] = u;
					M[v] = min(M[u], cap[u][v] -
flow[u][v]);
					if (v != sink) {
						q.push(v);
					}
					else {
						return make_pair(M[sink],
prv);
					}
				}
			}
		}
		return make_pair(0, prv);
	}

	//Edmonds Karp Algorithm
	int MaxFlow(const vector<vector<int>> cap, const vector<vector<int>>
graph,
		const int src, const int sink) {
		int sum = 0;
		vector<vector<int>> flow(graph.size(),
vector<int>(graph.size(), 0));

		while (true) {
			//BFS
			pair<int, vector<int>> ret = BFS(cap, graph, flow,
src, sink);
			int m = ret.first; vector<int> &prv = ret.second;

			if (m == 0) break;
			sum += m;

			int v = sink;
			while (v != src) {
				int u = prv[v];
				flow[u][v] += m;
				flow[v][u] -= m;
				v = u;
			}
		}
		return sum;
	}

	/* Usage
	vector<vector<int>> graph(V), cap(V, vector<int>(V, 0));
	graph[src].push_back(dst);
	graph[dst].push_back(src);
	cap[src][dst] = 1;

	printf("%d\n", MaxFlow(cap,graph, src, dst));
	*/
};
// O(fE)
struct FordFulkerson {
#define V 6

	/* Returns true if there is a path from source 's' to sink 't' in
	residual graph. Also fills parent[] to store the path */
	bool bfs(int rGraph[V][V], int s, int t, int parent[])
	{
		// Create a visited array and mark all vertices as not
visited
		bool visited[V];
		memset(visited, 0, sizeof(visited));

		// Create a queue, enqueue source vertex and mark source
vertex
		// as visited
		queue <int> q;
		q.push(s);
		visited[s] = true;
		parent[s] = -1;
```

```
		// Standard BFS Loop
		while (!q.empty())
		{
			int u = q.front();
			q.pop();

			for (int v = 0; v<V; v++)
			{
				if (visited[v] == false && rGraph[u][v] > 0)
				{
					q.push(v);
					parent[v] = u;
					visited[v] = true;
				}
			}
		}

		// If we reached sink in BFS starting from source, then
return
		// true, else false
		return (visited[t] == true);
	}

	// Returns the maximum flow from s to t in the given graph
	int fordFulkerson(int graph[V][V], int s, int t)
	{
		int u, v;

		// Create a residual graph and fill the residual graph with
		// given capacities in the original graph as residual
capacities
		// in residual graph
		int rGraph[V][V]; // Residual graph where rGraph[i][j]
indicates
								// residual capacity of edge
from i to j (if there
								// is an edge. If
rGraph[i][j] is 0, then there is not)
		for (u = 0; u < V; u++)
			for (v = 0; v < V; v++)
				rGraph[u][v] = graph[u][v];

		int parent[V];  // This array is filled by BFS and to store
path

		int max_flow = 0;  // There is no flow initially

								// Augment the flow while
tere is path from source to sink
		while (bfs(rGraph, s, t, parent))
		{
			// Find minimum residual capacity of the edges along
the
			// path filled by BFS. Or we can say find the
maximum flow
			// through the path found.
			int path_flow = INT_MAX;
			for (v = t; v != s; v = parent[v])
			{
				u = parent[v];
				path_flow = min(path_flow, rGraph[u][v]);
			}

			// update residual capacities of the edges and
reverse edges
			// along the path
			for (v = t; v != s; v = parent[v])
			{
				u = parent[v];
				rGraph[u][v] -= path_flow;
				rGraph[v][u] += path_flow;
			}

			// Add path flow to overall flow
			max_flow += path_flow;
		}

		// Return the overall flow
		return max_flow;
	}

};

struct BipartieMatch {
	bool dfs(size_t now, const vector<vector<int>> &graph,
		vector<bool> &visited, vector<size_t> &back_match) {
		if (visited[now]) return false;
		visited[now] = true;
		for (int nxt : graph[now]) {
			if (back_match[nxt] == -1 ||
				dfs(back_match[nxt], graph, visited,
back_match)) {
				back_match[nxt] = now;
				return true;
```

```
                        }
                }

                return false;
        }

        int bipartite_match(const vector<vector<int>> &graph) {
                int matched = 0;
                vector<bool> visited(graph.size(), false);
                vector<size_t> back_match(graph.size(), -1);
                for (size_t i = 0; i<graph.size(); i++) {
                        if (dfs(i, graph, visited, back_match)) {
                                matched++;
                        }
                }
                return matched;
        }
}
```

Palindrome DP

```
#define MX 312345
int a[MX * 2];
char s[MX * 2];
char buf[MX];

int main(void) {
        scanf("%s", buf);

        //builld formatted string
        for (int i = 0; i<strlen(buf) - 1; i++) {
                s[2 * i] = buf[i];
                s[2 * i + 1] = '#';
        }
        s[2 * strlen(buf) - 2] = buf[strlen(buf) - 1];
        s[2 * strlen(buf) - 1] = 0;

        int r = -1, p = -1;
        int len = 2 * strlen(buf) - 1;

        for (int i = 0; i<len; i++) {
                if (i <= r) a[i] = min(a[2 * p - i], r - i);
                else a[i] = 0;
                while (i - a[i] - 1 >= 0 && i + a[i] + 1 < strlen(s) && s[i
- a[i] - 1] == s[i + a[i] + 1]) {
                        a[i]++;
                }
                if (i + a[i] > r) {
                        r = a[i] + i; p = i;
                }
                scanf("%s", buf);

                //builld formatted string
                for (int i = 0; i<strlen(buf) - 1; i++) {
                        s[2 * i] = buf[i];
                        s[2 * i + 1] = '#';
                }
                s[2 * strlen(buf) - 2] = buf[strlen(buf) - 1];
                s[2 * strlen(buf) - 1] = 0;
                int r = -1, p = -1;
                int len = 2 * strlen(buf) - 1;

                for (int i = 0; i<len; i++) {
                        if (i <= r) a[i] = min(a[2 * p - i], r - i);
                        else a[i] = 0;
                        while (i - a[i] - 1 >= 0 && i + a[i] + 1 < strlen(s)
&& s[i - a[i] - 1] == s[i +
                                a[i] + 1]) {
                                a[i]++;
                        }

                        if (i + a[i] > r) {
                                r = a[i] + i; p = i;
                        }
                }
        }
```

Rectangle Area

```
typedef long long int lld;
int tree[MX * 4], lazy[MX * 4];
int len, r;

struct line {
        int x_idx, y1_idx, y2_idx;
        int inc;
};
struct rect {
        int x1, x2, y1, y2;
```

```
};
vector<rect> vec_rects;
vector<int> vec_x_coords, vec_y_coords;
vector<line> vec_lines;

int get_idx(const vector<int> &vec_coord, const int val) {
	return lower_bound(vec_coord.begin(), vec_coord.end(), val) - 
vec_coord.begin();
}
bool line_comp(const line &l, const line &r) {
	return l.x_idx < r.x_idx;
}

void make_unique(vector<int> &vec) {
	sort(vec.begin(), vec.end());
	vec.erase(unique(vec.begin(), vec.end()), vec.end());
}
void update(int node, int start, int end, int left, int right, int inc) {
	if (start > end || right < start || end < left) return;

	if (left <= start && end <= right) {
		lazy[node] += inc;
	}
	else {
		int mid = (start + end) / 2;
		update(node * 2, start, mid, left, right, inc);
		update(node * 2 + 1, mid + 1, end, left, right, inc);
	}

	if (lazy[node] > 0) {
		tree[node] = vec_y_coords[end + 1] - vec_y_coords[start];
	}
	else {
		if (node <= len - r) {
			tree[node] = tree[node * 2] + tree[node * 2 + 1];
		}
		else {
			tree[node] = 0;
		}
	}
}
lld solve() {
	lld res = 0;
	int N; scanf("%d", &N);

	for (int i = 0; i < N; i++) {
		int x1, x2, y1, y2; scanf("%d %d %d %d", &x1, &x2, &y1, 
&y2);
		vec_rects.push_back({ x1, x2, y1, y2 });
		vec_x_coords.push_back(x1); vec_x_coords.push_back(x2);
		vec_y_coords.push_back(y1); vec_y_coords.push_back(y2);
	}
	make_unique(vec_x_coords); make_unique(vec_y_coords);
	for (const rect &current_rect : vec_rects) {
		vec_lines.push_back({
			get_idx(vec_x_coords, current_rect.x1),
			get_idx(vec_y_coords, current_rect.y1),
			get_idx(vec_y_coords, current_rect.y2),
			1
		});
		vec_lines.push_back({
			get_idx(vec_x_coords, current_rect.x2),
			get_idx(vec_y_coords, current_rect.y1),
			get_idx(vec_y_coords, current_rect.y2),
			-1
		});
	}
	sort(vec_lines.begin(), vec_lines.end(), line_comp);

	const int tree_size = vec_y_coords.size() - 1;
	len = 1, r = 1;
	while (r < tree_size) {
		r *= 2;
		len += r;
	}

	for (int i = 0; i < vec_lines.size(); i++) {
		const line &current_line = vec_lines[i];


		if (i > 0) {
			const line &prev_line = vec_lines[i - 1];
			res += lld(tree[1]) *
				lld(vec_x_coords[current_line.x_idx] - 
vec_x_coords[prev_line.x_idx]);
		}
		update(1, 0, r - 1,
			current_line.y1_idx,
			current_line.y2_idx - 1,
			current_line.inc);

	}
	return res;
}
```

Rotating Calipers

```
// H is convex Hull(not circular)
void diameter(const vector<Point> &H) {
        const int M = H.size();
        if (M == 2) {
                printf("%lld %lld %lld %lld\n", H[0].first, H[0].second,
H[1].first, H[1].second);
                return;
        }

        int k = 1;
        while (area(H[M - 1], H[0], H[(k + 1) % M]) > area(H[M - 1], H[0],
H[k]))
                ++k;

        ll maxDist = 0;
        int ti = -1, tj = -1;
        for (int i = 0, j = k; i <= k && j < M; i++) {
                ll now = dist(H[i], H[j]);
                if (maxDist < now) {
                        maxDist = now;
                        ti = i, tj = j;
                }

                while (j<M && area(H[i], H[(i + 1) % M], H[(j + 1) % M]) >
area(H[i], H[(i + 1) % M], H[j])) {
                        ll now = dist(H[i], H[(j + 1) % M]);
                        if (maxDist < now) {
                                maxDist = now;
                                ti = i, tj = (j + 1) % M;
                        }
                        ++j;
                }
        }
        printf("%lld %lld %lld %lld\n", H[ti].first, H[ti].second,
H[tj].first, H[tj].second);
}
```

SCC

```
// O(V+E);
int dfs(int n)
{
        vis[n] = ++curr;
        s.push(n);
        int result = vis[n];
        for (int e = 0; e<arr[n].size(); e++)
        {
                int next = arr[n][e];
                if (vis[next] == 0) result = min(result, dfs(next));
                else if (finished[next] == 0) result = min(result,
vis[next]);
        }
        if (result == vis[n])
        {
                vector<int> kk;
                while (1)
                {
                        int now = s.top(); s.pop();
                        finished[now] = 1;
                        sn[now] = SN;
                        kk.push_back(now);
                        if (now == n) break;
                }
                SN++;
                sort(kk.begin(), kk.end());
                scc.push_back(kk);
        }
        return result;
}
����o���
for (int e = 1; e <= n; e++) if (vis[e] == 0) dfs(e);
```

Shortest Path

```
// Dijkstra - O((V+E)logV) with priority queue - with an important checking
- if (now_dist > dist[now_idx]) continue;
// BelmanFord - do V-1 iteration - O(VE) with adj list. V-th iteration
checks the existence of negative cycle
// Floyd-Warshall - k, i, j O(V^3) - applicable to graph with negative
edges.
//      Cycle Detection - init d[i][i] = INF, check whether d[i][i] >= 0
still
```

Simplex

```
namespace simplex {
        const int MAX_N = 50;
        const int MAX_M = 50;
```

```
	const double eps = 1e-9;
	inline int diff(double a, double b)
	{
		if (a - eps < b && b < a + eps)
			return 0;
		return (a < b) ? -1 : 1;
	}
	int n, m;
	double matrix[MAX_N + 1][MAX_M + MAX_N + 1];
	double c[MAX_N + 1];
	double solution[MAX_M + MAX_N + 1];
	int simplex()
	{
		// 0: found solution, 1: no feasible solution, 2: unbounded
		int i, j;
		while (true) {
			int nonfeasible = -1;
			for (j = 0; j <= n + m; j++) {
				int cnt = 0, pos = -1;
				for (i = 0; i <= n; i++) {
					if (diff(matrix[i][j], 0)) {
						cnt++;
						pos = i;
					}
				}
				if (cnt != 1)
					solution[j] = 0;
				else {
					solution[j] = c[pos] / matrix[pos][j];
					if (solution[j] < 0)
						nonfeasible = i;
				}
			}
			int pivotcol = -1;
			if (nonfeasible != -1) {
				double maxv = 0;
				for (j = 0; j <= n + m; j++) {
					if (maxv < matrix[nonfeasible][j]) {
						maxv = matrix[nonfeasible][j];
						pivotcol = j;
					}
				}
				if (pivotcol == -1)
					return 1;
			}
			else {
				double minv = 0;
				for (j = 0; j <= n + m; j++) {
					if (minv > matrix[0][j]) {
						minv = matrix[0][j];
						pivotcol = j;
					}
				}
				if (pivotcol == -1)
					return 0;
			}
			double minv = -1;
			int pivotrow = -1;
			for (i = 0; i <= n; i++) {
				if (diff(matrix[i][pivotcol], 0) > 0) {
					double test = c[i] / matrix[i][pivotcol];
					if (test < minv || minv < 0) {
						minv = test;
						pivotrow = i;
					}
				}
			}
			if (pivotrow == -1)
				return 2;
			for (i = 0; i <= n; i++) {
				if (i == pivotrow)
					continue;
				if (diff(matrix[i][pivotcol], 0)) {
					double ratio = matrix[i][pivotcol] / matrix[pivotrow][pivotcol];
					for (j = 0; j <= n + m; j++) {
						if (j == pivotcol) {
							matrix[i][j] = 0;
							continue;
						}
						else
							matrix[i][j] -= ratio * matrix[pivotrow][j];
					}
					c[i] -= ratio * c[pivotrow];
				}
			}
		}
	}
} // namespace simplex
```

```
  /* Usage
  To maximize p = -2x + 3y
  Constraints: x+3y <=40, 2x+4y >=10, x>=0, y>=0 // Make sure that RHS >=0
  n=2,m=2, matrix[ [2 -3 1 0 0], [1 3 0 1 0], [2 4 0 0 -1] ] c =
[ [0][4][10]]
  */
```

Splay

```
struct Node {
	Node *l, *r, *p;
	int key;
	int cnt;
	int sum, value, lazy;
	bool inv;
} *root;

void update(Node *x) {
	x->cnt = 1;
	x->sum = x->value;
	if (x->l) {
		x->cnt += x->l->cnt;
		x->sum += x->l->sum;
	}
	if (x->r) {
		x->cnt += x->r->cnt;
		x->sum += x->r->sum;
	}

}
void rotate(Node *x) {
	Node *p = x->p; Node *b;
	if (x == p->l) {
		p->l = b = x->r;
		x->l = p;
	}
	else {
		p->r = b = x->l;
		x->l = p;
	}
	x->p = p->p;
	p->p = x;
	if (b) b->p = p;
	(x->p ? p == x->p->l ? x->p->l : x->p->r : root) = x;
	update(p);
	update(x);
}
void splay(Node *x) {
	while (x->p) {
		Node *p = x->p, *g = p->p;
		if (g) rotate((x == p->l) == (p == g->l) ? p : x);
		rotate(x);
	}
}
void insert(int key) {
	Node *p = root, **pp;
	if (!p) {
		Node *x = new Node;
		root = x;
		x->l = x->r = x->p = NULL;
		x->key = key;
		return;
	}
	while (1) {
		if (key == p->key) return;
		if (key < p->key) {
			if (!(p->l)) {
				pp = &p->l;
				break;
			}
			p = p->l;
		}
		else {
			if (!(p->r)) {
				pp = &p->r;
				break;
			}
			p = p->r;
		}
	}
	Node *x = new Node;
	*pp = x;
	x->l = x->r = NULL;
	x->p = p;
	x->key = key;
	splay(x);
}
bool find(int key) {
	Node *p = root;
	if (!p) return false;
	while (p) {
		if (key == p->key) break;
		if (key < p->key) {
```

```
                if (!p->l) break;
                p = p->l;
            }
            else {
                if (!p->r) break;
                p = p->r;
            }
        }
        splay(p);
        return key == p->key;
}
void remove(int key) {
        if (!find(key))return;
        Node *p = root;
        if (p->l) {
            if (p->r) {
                root = p->l;
                root->p = NULL;
                Node *x = root;
                while (x->r) x = x->r;
                x->r = p->r;
                p->r->p = x;
                splay(x);
                delete p;
                return;
            }
            root = p->l;
            root->p = NULL;
            delete p;
            return;
        }
        if (p->r) {
            root = p->r;
            root->p = NULL;
            delete p;
            return;
        }
        root = NULL;
}

void propagate(Node *x) {
        x->value += x->lazy;
        if (x->inv) {
            Node *t = x->l; x->l = x->r; x->r = t;
            x->inv = false;
            if (x->l) x->l->inv = !x->l->inv;
            if (x->r) x->r->inv = !x->r->inv;
        }
        if (x->l) {
            x->l->lazy += x->lazy;
            x->l->sum += x->l->cnt * x->lazy;
        }
        if (x->r) {
            x->r->lazy += x->lazy;
            x->r->sum += x->r->cnt * x->lazy;
        }
        x->lazy = 0;
}

//Note that k is 0-base !
void findKth(int k) {
        Node *x = root;
        propagate(x);
        while (1) {
            while (x->l && x->l->cnt > k) {
                x = x->l;
                propagate(x);
            }
            if (x->l) k -= x->l->cnt;
            if (!k--) break;
            x = x->r;
            propagate(x);
        }
        splay(x);
}

void init(int n) {
        Node *x;
        int i;
        root = x = new Node;
        x->l = x-> = x->p = NULL;
        x->cnt = n;
        x->sum = x->value = 0;
        for (i = 1; i<n; i++) {
            x->r = new Node;
            x->r->p = x;
            x = x->r;
            x->l = x->r = NULL;
            x->cnt = n - i;
            x->sum = x->value = 0;
        }
}
void add(int i, int z) {
        findKth(i);
```

```
		root->sum += z;
		root->value += z;
}
// [l, r] inclusive
void interval(int l, int r) {
		findKth(l - 1);
		Node *x = root;
		root = x->r;
		root->p = NULL;
		findKth(r - l + 1);
		x->r = root;
		root->p = x;
		root = x;
}
int sum(int l, int r) {
		interval(l, r);
		return root->r->l->sum;
}

void add(int l, int r, int z) {
		interval(l, r);
		Node *x = root->r->l;
		x->sum += x->cnt * z;
		x->lazy += z;
}
void reverse(int l, int r) {
		interval(l, r);
		Node *x = root->r->l;
		x->inv = !x->inv;
}
int a[100001];
int main(void) {
		int N; scanf("%d", &N);
		init(N);
		for (int i = 1; i <= N; i++) {
				scanf("%d", a + i);
		}
		for (int i = 1; i <= N; i++) {
				insert(a[i]);
				root->value = i;
		}
		for (int i = 1; i <= N; i++) {
				find(i);
		}

}
```

Suffix Array & LCP

```
// s : 입력 문자열
// group : 접미사의 첫 글자 (입력 문자열의 각 문자)
// sagroup : gap에 따른 Counting Sort 후의 group
// gap : Counting Sort시, group의 각 원소를 비교하는 길이
// lcp : 최장 공통 접두사 길이

const bool cmp(int i, int j) {
		if (group[i] != group[j]) return group[i] < group[j];
		return group[i + gap] < group[j + gap];
}

void getSuffixArray() {
		for (int i = 0; i < n; i++) {
				sa[i] = i;
				group[i] = s[i];
		}

		group[n] = -1, sagroup[n] = -1, gap = 1;

		while (gap < n) {							// Counting 
Sort
				sort(sa, sa + n, cmp);
				for (int i = 1; i < n; i++)
						sagroup[i] = sagroup[i - 1] + cmp(sa[i - 1], sa[i]);
				for (int i = 0; i < n; i++) group[sa[i]] = sagroup[i];

				if (sagroup[n - 1] == n - 1) break;
				gap *= 2;
		}
}


void getLcpArray() {
		for (int i = 0, k = 0; i < n; i++) {
				if (group[i] == 0) lcp[group[i]] = 0;
				else {
						for (int j = sa[group[i] - 1]; s[i + k] == s[j + k]; 
k++);
						lcp[group[i]] = k;

						if (k != 0) k--;
```

```
			}
		}
}
```

TSP

```
// O(2^N * N^2)

const int MAXN = 16;

int n;
int W[MAXN][MAXN], dp[1 << MAXN][MAXN];

int main() {
	scanf("%d", &n);
	for (int i = 0; i < n; i++)
		for (int j = 0; j < n; j++)
			scanf("%d", &W[i][j]);

	memset(dp, -1, sizeof(dp));

	dp[1][0] = 0; //start from 0.
	for (int bit = 0; bit < (1 << n); bit++) {
		for (int now = 0; now < n; now++) {
			if ((bit & (1 << now)) != (1 << now)) continue;
			if (dp[bit][now] == -1) continue;

			for (int nxt = 0; nxt < n; nxt++) {
				if ((bit & (1 << nxt)) == (1 << nxt))
continue;
				if (W[now][nxt] == 0) continue;
				int status = bit | (1 << nxt);
				if (dp[status][nxt] == -1 || dp[status][nxt]
> dp[bit][now] + W[now][nxt]) {
					dp[status][nxt] = dp[bit][now] +
W[now][nxt];
				}
			}
		}
	}

	int ans = 2e9;
	for (int i = 0; i < n; i++) {
		if (W[i][0] == 0) continue;
		if (dp[(1 << n) - 1][i] == -1) continue;
		ans = std::min(ans, dp[(1 << n) - 1][i] + W[i][0]);
	}
	printf("%d", ans);

	return 0;
}
```

UnionFind

```
typedef vector<int> vi;

class UnionFind {
private:
	vi p, rank, setSize;
	int numSets;
public:
	UnionFind(int N) {
		setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
		p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i;
	}
	int findSet(int i) { return (p[i] == i) ? i : (p[i] =
findSet(p[i])); }
	bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
	void unionSet(int i, int j) {
		if (!isSameSet(i, j)) {
			numSets--;
			int x = findSet(i), y = findSet(j);
			// rank is used to keep the tree short
			if (rank[x] > rank[y]) { p[y] = x; setSize[x] +=
setSize[y]; }
			else {
				p[x] = y; setSize[y] += setSize[x];
				if (rank[x] == rank[y]) rank[y]++;
			}
		}
	}
	int numDisjointSets() { return numSets; }
	int sizeOfSet(int i) { return setSize[findSet(i)]; }
};
```