# Team notebook

November 9, 2018

## Contents



# 1 Algorithms

## 1.1 2SAT

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

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

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

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

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

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

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

## 1.2 Convex Hull Trick

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

typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype cross(point a, point b) {
    return (conj(a) * b).y();
}

vector<point> hull, vecs;

// lines as k*x+b
void add_line(ftype k, ftype b) {
    point nw = {k, b};
    while(!vecs.empty() && dot(vecs.back(), nw - hull.back()) < 0) {
        hull.pop_back();
        vecs.pop_back();
    }
    if(!hull.empty()) {
        vecs.push_back(1i * (nw - hull.back()));
    }
    hull.push_back(nw);
}

// minimum in some point x
int get(ftype x) {
    point query = {x, 1};
    auto it = lower_bound(vecs.begin(), vecs.end(), query, [](point a,
        point b) {
        return cross(a, b) > 0;
    });
    return dot(query, hull[it - vecs.begin()]);
}
```

## 1.3 Divide Conquer Optimization

```
// solution for problem guardians of the lunatic IOI 14
```

```
#include <stdio.h>
#define ll long long
#define infinity 11111111111111111111ll

ll C[8111];
ll sums[8111];
ll F[811][8111];
int P[811][8111];

ll cost(int i, int j) {
    return i > j ? 0 : (sums[j] - sums[i-1]) * (j - i + 1);
}

void fill(int g, int l1, int l2, int p1, int p2) {
    // fill(g,l1,l2,p1,p2) calculates all P[g][l] and F[g][l] for l1 <= l
        <= l2,
    // with the knowledge that p1 <= P[g][l] <= p2

    // if l1 > l2, then there's nothing to calculate
    if (l1 > l2) return;

    int lm = l1 + l2 >> 1;
    // calculate P[g][lm] and F[g][lm]
    P[g][lm] = -1;
    F[g][lm] = infinity;
    for (int k = p1; k <= p2; k++) {
        ll new_cost = F[g-1][k] + cost(k+1,lm);
        if (F[g][lm] > new_cost) {
            F[g][lm] = new_cost;
            P[g][lm] = k;
        }
    }

    // calculate both sides of lm
    fill(g, l1, lm-1, p1, P[g][lm]);
    fill(g, lm+1, l2, P[g][lm], p2);
}

int main() {
    int G, L;
    scanf("%d%d", &L, &G);
    sums[0] = 0;
    for (int i = 1; i <= L; i++) {
        scanf("%lld", C + i);
        sums[i] = sums[i-1] + C[i];
    }
    #define cost(i,j) (sums[j]-sums[(i)-1])*((j)-(i)+1)

    for (int l = 0; l <= L; l++) {
        F[1][l] = cost(1,l);
        P[1][l] = 0;
    }

    for (int g = 2; g <= G; g++) {
        fill(g, 0, L, 0, L);
    }
    printf("%lld\n", F[G][L]);
}
```

## 1.4 HLD

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

int chainNo = 0; // index of current chain (last chain started)
int chainHead[N]; // head of the chain (entry point)
int chainPos[N]; // the position of the node in the chain
int chainInd[N]; // the chain index of each node
int chainSize[N]; // the size of each chain

void hld(int cur) {
    if(chainHead[chainNo] == -1) {
        chainHead[chainNo]=cur;
    }
    chainInd[cur] = chainNo;
    chainPos[cur] = chainSize[chainNo];
    chainSize[chainNo]++;

    int ind = -1, mai = -1;
    for (int i = 0; i < adj[cur].sz; i++) {
        if (subsize[ adj[cur][i] ] > mai) {
            mai = subsize[ adj[cur][i] ];
            ind = i;
        }
    }

    if (ind >= 0) {
```

```
        hld(adj[cur][ind]);
    }

    for(int i = 0; i < adj[cur].sz; i++) {
        if (i != ind) {
            chainNo++;
            hld(adj[cur][i]);
        }
    }
}
```

## 1.5 Li Chao Tree

```
typedef int ftype;
typedef complex<ftype> point;
#define x real
#define y imag

ftype dot(point a, point b) {
    return (conj(a) * b).x();
}

ftype f(point a, ftype x) {
    return dot(a, {x, 1});
}

const int maxn = 2e5;

point line[4 * maxn];

void add_line(point nw, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    bool lef = f(nw, l) < f(line[v], l);
    bool mid = f(nw, m) < f(line[v], m);
    if(mid) {
        swap(line[v], nw);
    }
    if(r - l == 1) {
        return;
    } else if(lef != mid) {
        add_line(nw, 2 * v, l, m);
    } else {
        add_line(nw, 2 * v + 1, m, r);
    }
}

// minimum in some point x
int get(int x, int v = 1, int l = 0, int r = maxn) {
    int m = (l + r) / 2;
    if(r - l == 1) {
        return f(line[v], x);
    } else if(x < m) {
        return min(f(line[v], x), get(x, 2 * v, l, m));
    } else {
        return min(f(line[v], x), get(x, 2 * v + 1, m, r));
    }
}
```

## 1.6 MO

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define MAX 200005
#define SQ 450

// This is the solution for codeforces 86D
// queries in range l...r of sum of freq(x)^2 * x

struct query {
    int l, r, pos;
};

int n, q;
query queries[MAX];
int arr[MAX], freq[5*MAX];
ll tot, ans[MAX];

bool operator<(const query &q1, const query &q2) {
    int v1 = q1.l/SQ;
    int v2 = q2.l/SQ;
    if (v1 == v2) {
        return v1%2 ? q1.r > q2.r : q1.r < q2.r;
    }
    return v1 < v2;
}
```

```
#define add(pos) (tot += arr[pos]*(2LL*(freq[arr[pos]]++)+1))
#define rem(pos) (tot += arr[pos]*(-2LL*(freq[arr[pos]]--)+1))

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    cin >> n >> q;
    for (int i = 1; i <= n; ++i) {
        cin >> arr[i];
    }
    for (int i = 0; i < q; ++i) {
        cin >> queries[i].l >> queries[i].r;
        queries[i].pos = i;
    }
    int l = 1, r = 0;
    sort(queries, queries+q);
    for (int i = 0; i < q; ++i) {
        while (r < queries[i].r) ++r, add(r);
        while (l > queries[i].l) --l, add(l);
        while (r > queries[i].r) rem(r), r--;
        while (l < queries[i].l) rem(l), l++;
        ans[queries[i].pos] = tot;
    }
    for (int i = 0; i < q; ++i) {
        cout << ans[i] << "\n";
    }
}
```

## 1.7 Matrix Exponentiation

Find matrix M that satisfies

$$M \times \begin{bmatrix} f(n) \\ f(n-1) \\ f(n-2) \\ \dots \\ f(n-k) \end{bmatrix} = \begin{bmatrix} f(n+1) \\ f(n) \\ f(n-1) \\ \dots \\ f(n-k+1) \end{bmatrix}$$

Then calculate $M^x$ using fast exponentiation and matrix multiplication

## 1.8 Matrix Multiplication

```
#include <bits/stdc++.h>
using namespace std;
typedef vector<int> vi;
typedef vector<vi> vii;

// multiply two matrices (n1xm) x (mxn2) resulting in (nxn) matrix
vii multiply(vii &A, vii &B) {
    int N1 = A.size();
    int N2 = B[0].size();
    int M = A[0].size();

    vii C(N1, vi(N2));
    for (int i = 0; i < N1; ++i) {
        for (int j = 0; j < N2; ++j) {
            for (int k = 0; k < M; ++k) {
                C[i][j] += A[i][k]*B[k][j];
            }
        }
    }
    return C;
}

// get identity matrix of size NxN
vii get_identity(int N) {
    vii C(N, vi(N));
    for (int i = 0; i < N; ++i) {
        C[i][i] = 1;
    }
    return C;
}

// fast matrix exponentiation (matrices must be square) A^x
vii fast_pow(vii A, int x) {
    int N = A.size();
    vii C = get_identity(N);
    while (x > 0) {
        if (x&1) {
            C = multiply(C, A);
        }
        A = multiply(A, A);
        x /= 2;
    }
    return C;
}
```

```
int main() {
    // init fib recurrence matrix
    // | 1 1 |
    // | 1 0 |
    vii A(2, vi(2));
    A[0][0] = A[0][1] = A[1][0] = 1;

    // init fib vector
    // | 1 | f(2)
    // | 1 | f(1)
    vii B(2, vi(1));
    B[0][0] = B[1][0] = 1;

    // do fast exponentiation -- f(10) = 34+21
    // | 34 21 |
    // | 21 13 |
    vii C = fast_pow(A, 10-2); // fib(10) = 10-initial_n (2)
    print_mat(C);

    // get result multiplying by vector --
    // | 55 |
    // | 34 |
    vii res = multiply(C, B);
    cout << res[0][0] << "\n";
}
```

# 2 Data Structures

## 2.1 Comparator

```
struct CustomCompare {
    bool operator()(const int &v1, const int &v2) {
        return v1 > v2; // smaller to bigger (reverse)
    }
};

// pq with custom comparator
priority_queue<int, vector<int>, CustomCompare> pq;

// pq from smaller to bigger
priority_queue<int, vector<int>, greater<int> > q;
```

## 2.2 Dynamic Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
#define SIZE 1000000000

// segtree for sum in a very large range (SIZE=10^9)
struct node {
    int l, r, v;
    node() {
        l = r = -1;
        v = 0;
    }
};
vector<node> segtree;

// initial build with 0 for all items
void build() {
    segtree.push_back(node());
}

// query the sum in a range [a...b]
int query(int i, int l, int r, int a, int b) {
    if (i == -1 || b < l || a > r) {
        return 0;
    } else if (a <= l && r <= b) {
        return segtree[i].v;
    } else {
        int mid = (l+r)/2;
        int v1 = query(segtree[i].l, l, mid, a, b);
        int v2 = query(segtree[i].r, mid+1, r, a, b);
        return v1+v2;
    }
}

int query(int l, int r) {
    return query(0, 1, SIZE, l, r);
}

// add a node to complement the traversing path
int add_node() {
    segtree.push_back(node());
    return segtree.size()-1;
}
```

```
// Add v to item in position p
void update(int i, int l, int r, int p, int v) {
    if (l == r) {
        segtree[i].v += v;
    } else {
        int mid = (l+r)/2;
        if (p <= mid) {
            if (segtree[i].l == -1) {
                int pos = add_node();
                segtree[i].l = pos;
            }
            update(segtree[i].l, l, mid, p, v);
        } else {
            if (segtree[i].r == -1) {
                int pos = add_node();
                segtree[i].r = pos;
            }
            update(segtree[i].r, mid+1, r, p, v);
        }
        int v1 = segtree[i].l == -1 ? 0 : segtree[segtree[i].l].v;
        int v2 = segtree[i].r == -1 ? 0 : segtree[segtree[i].r].v;
        segtree[i].v = v1+v2;
    }
}

void update(int p, int v) {
    update(0, 1, SIZE, p, v);
}

int main() {
    // initialize the segtree
    build();

    // add some values to the tree
    update(10, 150);
    update(1000, 150);
    update(100000000, 150);

    // query some values from the tree -- 300, 150, 450
    cout << query(1, 1000000) << "\n";
    cout << query(1001, 100000000) << "\n";
    cout << query(9, 1000000000) << "\n";
}
```

## 2.3 Lazy Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
#define SIZE 100000
#define SEGTREE (((int)log2(SIZE))+2)

// segtree for swapping & counting items color in range
struct node {
        int lazy, black, white;
        node() {
        lazy = black = white = 0;
    }
};

vector<node> segtree(1<<SEGTREE);

// initial build, all values are white
void build(int i, int l, int r) {
    if (l == r) {
        segtree[i].white = 1;
    } else {
        int mid = (l+r)/2;
        build(2*i, l, mid);
        build(2*i+1, mid+1, r);
        segtree[i].white = segtree[2*i].white+segtree[2*i+1].white;
    }
}

void build() {
    build(1, 1, SIZE);
}

// lazy propagation function for visited nodes
void lazyprop(int i, int l, int r) {
    if (l < r) {
        segtree[2*i].lazy += segtree[i].lazy;
        segtree[2*i+1].lazy += segtree[i].lazy;
    }
    if (segtree[i].lazy%2 != 0) {
        swap(segtree[i].white, segtree[i].black);
    }
    segtree[i].lazy = 0;
}
```

```
// query the number of black items in range [a...b]
int query(int i, int l, int r, int a, int b) {
    lazyprop(i, l, r);
    if (b < l || a > r) {
        return 0;
    } else if (a <= l && r <= b) {
        return segtree[i].black;
    } else {
        int mid = (l+r)/2;
        int v1 = query(2*i, l, mid, a, b);
        int v2 = query(2*i+1, mid+1, r, a, b);
        return v1+v2;
    }
}

int query(int l, int r) {
    return query(1, 1, SIZE, l, r);
}

// swap all items color in range [a...b]
void update(int i, int l, int r, int a, int b) {
    lazyprop(i, l, r);
    if (b < l || a > r) {
        return;
    } else if (a <= l && r <= b) {
        ++segtree[i].lazy;
        lazyprop(i, l, r);
    } else {
        int mid = (l+r)/2;
        update(2*i, l, mid, a, b);
        update(2*i+1, mid+1, r, a, b);
        segtree[i].white = segtree[2*i].white+segtree[2*i+1].white;
        segtree[i].black = segtree[2*i].black+segtree[2*i+1].black;
    }
}

void update(int l, int r) {
    update(1, 1, SIZE, l, r);
}

int main() {
    // initialize the segtree
    build();

    // update some items in range
    update(1, 7);
    update(100000, 100000);
    update(1000, 10000);
    update(9500, 10001);

    // query some items in range -- 5, 8509, 4501
    cout << query(3, 999) << "\n";
    cout << query(1, 100000) << "\n";
    cout << query(5000, 10500) << "\n";
}
```

## 2.4 Persistent Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
#define MAXNUM 9

// segment tree for kth order statistic in a range l...r
// this one assumes each number in the array is unique
// in case numbers are too large or duplicate, compress coords
struct node {
    int l, r, v;
    node() { }
    node(int _l, int _r, int _v) {
        l = _l, r = _r, v = _v;
    }
};
vector<node> segtree;
vector<int> roots;

// initial build with 0 for all items
int build(int l, int r) {
    if (l == r) {
        segtree.push_back(node(-1, -1, 0));
        return segtree.size()-1;
    } else {
        int mid = (l+r)/2;
        int li = build(l, mid);
        int ri = build(mid+1, r);
        segtree.push_back(node(li, ri, 0));
        return segtree.size()-1;
    }
}
```

```
// query the kth val if sorted in range rep by segtrees [p1...p2]
int query(int l, int r, int p1, int p2, int K) {
    int mid = (l+r)/2;
    if (l == r) {
        return l;
    }
    int v1 = segtree[segtree[p1].l].v;
    int v2 = segtree[segtree[p2].l].v;
    if (v2-v1 >= K) {
        return query(l, mid, segtree[p1].l, segtree[p2].l, K);
    } else {
        return query(mid+1, r, segtree[p1].r, segtree[p2].r, K-v2+v1);
    }
}

int query(int l, int r, int K) {
    return query(1, MAXNUM, roots[l-1], roots[r], K);
}

// insert next item & add new nodes to segtree
int insert(int l, int r, int i, int p) {
    if (l == r) {
        segtree.push_back(node(-1, -1, 1));
        return segtree.size()-1;
    } else {
        int mid = (l+r)/2;
        int lpos = segtree[i].l;
        int rpos = segtree[i].r;
        if (p <= mid) {
            lpos = insert(l, mid, lpos, p);
        } else {
            rpos = insert(mid+1, r, rpos, p);
        }
        segtree.push_back(node(lpos, rpos,
            segtree[lpos].v+segtree[rpos].v));
        return segtree.size()-1;
    }
}

int insert(int i, int p) {
    return insert(1, MAXNUM, roots[i], p);
}

int main() {
    int arr[] = {6, 5, 4, 3, 8, 7, 1, 3, 2};

    // initialize segtrees
    roots.push_back(build(1, MAXNUM));
    for (int i = 0; i < MAXNUM; ++i) {
        roots.push_back(insert(i, arr[i]));
    }

    // query kth order for ranges -- 4, 7, 8
    cout << query(1, 9, 4) << "\n";
    cout << query(5, 9, 3) << "\n";
    cout << query(3, 5, 3) << "\n";
}
```

## 2.5 Segment Tree

```
#include <bits/stdc++.h>
using namespace std;
#define inf ((int)1e9)
#define SIZE 9
#define SEGTREE (((int)log2(SIZE))+2)

// segment tree for RMQ
vector<int> segtree(1<<SEGTREE);

// query the min in range [a...b]
int query(int i, int l, int r, int a, int b) {
    if (b < l || a > r) {
        return inf;
    } else if (a <= l && r <= b) {
        return segtree[i];
    } else {
        int mid = (l+r)/2;
        int v1 = query(2*i, l, mid, a, b);
        int v2 = query(2*i+1, mid+1, r, a, b);
        return min(v1, v2);
    }
}

int query(int l, int r) {
    return query(1, 1, SIZE, l, r);
}
```

```
// set item in position p to have value v
void update(int i, int l, int r, int p, int v) {
    if (l == r) {
        segtree[i] = v;
    } else {
        int mid = (l+r)/2;
        if (p <= mid) {
            update(2*i, l, mid, p, v);
        } else {
            update(2*i+1, mid+1, r, p, v);
        }
        segtree[i] = min(segtree[2*i], segtree[2*i+1]);
    }
}

void update(int p, int v) {
    update(1, 1, SIZE, p, v);
}

int main() {
    int arr[] = {6, 1, 6, 200, 195, 1, 6, 1, 90};

    // build initial segtree
    for (int i = 1; i <= SIZE; ++i) {
        update(i, arr[i-1]);
    }

    // query some ranges in segtree -- 1, 195, 6
    cout << query(1, 9) << "\n";
    cout << query(4, 5) << "\n";
    cout << query(7, 7) << "\n";
}
```

### 2.6 Union Find

```
#include <bits/stdc++.h>
using namespace std;
#define SIZE 4

// union Find container (negative is size)
vector<int> uf(SIZE, -1);

// find the component to which v belongs
int find(int v) {
    return uf[v] < 0 ? v : uf[v] = find(uf[v]);
}

// merge components of u and v
void merge(int u, int v) {
    int pu = find(u);
    int pv = find(v);
    if (pu == pv) {
        return;
    }
    if (-uf[pu] < -uf[pv]) {
        swap(pu, pv);
    }
    uf[pu] += uf[pv];
    uf[pv] = pu;
}

int main() {
    // merge 1 & 2 (sz = 2)
    merge(1, 2);
    assert(find(1) == find(2));
    cout << -uf[find(1)] << "\n";

    // merge 2 & 4 (sz = 3)
    merge(2, 4);
    assert(find(1) == find(4));
    cout << -uf[find(4)] << "\n";
}
```

## 3 Geometry

### 3.1 Convex Hull

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

struct pt {
    double x, y;
};

bool cmp(pt a, pt b) {
```

```
    return a.x < b.x || (a.x == b.x && a.y < b.y);
}

bool cw(pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) < 0;
}

bool ccw(pt a, pt b, pt c) {
    return a.x*(b.y-c.y)+b.x*(c.y-a.y)+c.x*(a.y-b.y) > 0;
}

void convex_hull(vector<pt>& a) {
    if (a.size() == 1)
        return;

    sort(a.begin(), a.end(), &cmp);
    pt p1 = a[0], p2 = a.back();
    vector<pt> up, down;
    up.push_back(p1);
    down.push_back(p1);
    for (int i = 1; i < (int)a.size(); i++) {
        if (i == a.size() - 1 || cw(p1, a[i], p2)) {
            while (up.size() >= 2 && !cw(up[up.size()-2], up[up.size()-1],
                a[i]))
                up.pop_back();
            up.push_back(a[i]);
        }
        if (i == a.size() - 1 || ccw(p1, a[i], p2)) {
            while(down.size() >= 2 && !ccw(down[down.size()-2],
                down[down.size()-1], a[i]))
                down.pop_back();
            down.push_back(a[i]);
        }
    }

    a.clear();
    for (int i = 0; i < (int)up.size(); i++)
        a.push_back(up[i]);
    for (int i = down.size() - 2; i > 0; i--)
        a.push_back(down[i]);
}
```

## 3.2 Heron Formula

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

$p$ = Triangle perimeter

$a$ = Triangle side

$b$ = Triangle side

$c$ = Triangle side

## 3.3 Pick Theorem

$$Area = \frac{B}{2} + I - 1$$

$B$ = Lattice points on the polygon

$I$ = Lattice points in polygon interior

## 3.4 Segment Intersection

```
const double EPS = 1E-9;

struct pt {
    double x, y;
};

struct seg {
    pt p, q;
    int id;

    double get_y(double x) const {
        if (abs(p.x - q.x) < EPS)
            return p.y;
        return p.y + (q.y - p.y) * (x - p.x) / (q.x - p.x);
    }
};

bool intersect1d(double l1, double r1, double l2, double r2) {
    if (l1 > r1)
        swap(l1, r1);
    if (l2 > r2)
        swap(l2, r2);
    return max(l1, l2) <= min(r1, r2) + EPS;
```

```
}

int vec(const pt& a, const pt& b, const pt& c) {
    double s = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    return abs(s) < EPS ? 0 : s > 0 ? +1 : -1;
}

bool intersect(const seg& a, const seg& b)
{
    return intersect1d(a.p.x, a.q.x, b.p.x, b.q.x) &&
           intersect1d(a.p.y, a.q.y, b.p.y, b.q.y) &&
           vec(a.p, a.q, b.p) * vec(a.p, a.q, b.q) <= 0 &&
           vec(b.p, b.q, a.p) * vec(b.p, b.q, a.q) <= 0;
}

bool operator<(const seg& a, const seg& b)
{
    double x = max(min(a.p.x, a.q.x), min(b.p.x, b.q.x));
    return a.get_y(x) < b.get_y(x) - EPS;
}

struct event {
    double x;
    int tp, id;

    event() {}
    event(double x, int tp, int id) : x(x), tp(tp), id(id) {}

    bool operator<(const event& e) const {
        if (abs(x - e.x) > EPS)
            return x < e.x;
        return tp > e.tp;
    }
};

set<seg> s;
vector<set<seg>::iterator> where;

set<seg>::iterator prev(set<seg>::iterator it) {
    return it == s.begin() ? s.end() : --it;
}

set<seg>::iterator next(set<seg>::iterator it) {
    return ++it;
}

pair<int, int> solve(const vector<seg>& a) {
    int n = (int)a.size();
    vector<event> e;
    for (int i = 0; i < n; ++i) {
        e.push_back(event(min(a[i].p.x, a[i].q.x), +1, i));
        e.push_back(event(max(a[i].p.x, a[i].q.x), -1, i));
    }
    sort(e.begin(), e.end());

    s.clear();
    where.resize(a.size());
    for (size_t i = 0; i < e.size(); ++i) {
        int id = e[i].id;
        if (e[i].tp == +1) {
            set<seg>::iterator nxt = s.lower_bound(a[id]), prv = prev(nxt);
            if (nxt != s.end() && intersect(*nxt, a[id]))
                return make_pair(nxt->id, id);
            if (prv != s.end() && intersect(*prv, a[id]))
                return make_pair(prv->id, id);
            where[id] = s.insert(nxt, a[id]);
        } else {
            set<seg>::iterator nxt = next(where[id]), prv =
                prev(where[id]);
            if (nxt != s.end() && prv != s.end() && intersect(*nxt, *prv))
                return make_pair(prv->id, nxt->id);
            s.erase(where[id]);
        }
    }

    return make_pair(-1, -1);
}
```

# 4 Graphs

## 4.1 Articulation Points

```
Let disc_time[v] = -1 and explored[v] = false forall v
dfscounter = 0

DFS(v):
  explored[v] = true
```

```
disc_time[v] = low[v] = ++dfscounter
foreach edge (v,x):
  if !explored[x]:
    DFS(x)
    low[v] = min(low[v], low[x])
    if low[x] >= disc_time[v]:
      print  v  is articulation point separating x
  elif x is not  vs  parent:
    low[v] = min(low[v], disc_time[x])
```

## 4.2 Bellman Ford

```
function BellmanFord(list vertices, list edges, vertex source)
  ::distance[],predecessor[]

  // This implementation takes in a graph, represented as
  // lists of vertices and edges, and fills two arrays
  // (distance and predecessor) about the shortest path
  // from the source to each vertex

  // Step 1: initialize graph
  for each vertex v in vertices:
      distance[v] := inf          // At the beginning , all vertices have
          a weight of infinity
      predecessor[v] := null      // And a null predecessor

  distance[source] := 0           // The weight is zero at the source

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
      for each edge (u, v) with weight w in edges:
          if distance[u] + w < distance[v]:
              distance[v] := distance[u] + w
              predecessor[v] := u

  // Step 3: check for negative-weight cycles
  for each edge (u, v) with weight w in edges:
      if distance[u] + w < distance[v]:
          error "Graph contains a negative-weight cycle"

  return distance[], predecessor[]
```

## 4.3 Bipartite Matching

```
// fast approach taken from geeksforgeeks
#include <bits/stdc++.h>
using namespace std;

// M is number of applicants
// and N is number of jobs
#define M 6
#define N 6

// A DFS based recursive function
// that returns true if a matching
// for vertex u is possible
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[]) {
    // Try every job one by one
    for (int v = 0; v < N; v++) {
        // If applicant u is interested in
        // job v and v is not visited
        if (bpGraph[u][v] && !seen[v]) {
            // Mark v as visited
            seen[v] = true;

            // If job 'v' is not assigned to an
            // applicant OR previously assigned
            // applicant for job v (which is matchR[v])
            // has an alternate job available.
            // Since v is marked as visited in
            // the above line, matchR[v] in the following
            // recursive call will not get job 'v' again
            if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR)) {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}

// Returns maximum number
// of matching from M to N
int maxBPM(bool bpGraph[M][N]) {
    // An array to keep track of the
    // applicants assigned to jobs.
    // The value of matchR[i] is the
```

```
    // applicant number assigned to job i,
    // the value -1 indicates nobody is
    // assigned.
    int matchR[N];

    // Initially all jobs are available
    memset(matchR, -1, sizeof(matchR));

    // Count of jobs assigned to applicants
    int result = 0;
    for (int u = 0; u < M; u++) {
        // Mark all jobs as not seen
        // for next applicant.
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Find if the applicant 'u' can get a job
        if (bpm(bpGraph, u, seen, matchR)) {
            result++;
        }
    }
    return result;
}

int main() {
    // create basic bipartite graph
    bool bpGraph[M][N] = {
        {0, 1, 1, 0, 0, 0},
        {1, 0, 0, 1, 0, 0},
        {0, 0, 1, 0, 0, 0},
        {0, 0, 1, 1, 0, 0},
        {0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1}
    };
    cout << "Maximum matching is " << maxBPM(bpGraph) << "\n";
}
```

## 4.4 Bridges

```
private int bridges;    // number of bridges
    private int cnt;        // counter
    private int[] pre;      // pre[v] = order in which dfs examines v
    private int[] low;      // low[v] = lowest preorder of any vertex
        connected to v

    public Bridge(Graph G) {
        low = new int[G.V()];
        pre = new int[G.V()];
        for (int v = 0; v < G.V(); v++) low[v] = -1;
        for (int v = 0; v < G.V(); v++) pre[v] = -1;

        for (int v = 0; v < G.V(); v++)
            if (pre[v] == -1)
                dfs(G, v, v);
    }

    public int components() { return bridges + 1; }

    private void dfs(Graph G, int u, int v) {
        pre[v] = cnt++;
        low[v] = pre[v];
        for (int w : G.adj(v)) {
            if (pre[w] == -1) {
                dfs(G, v, w);
                low[v] = Math.min(low[v], low[w]);
                if (low[w] == pre[w]) {
                    StdOut.println(v + "-" + w + " is a bridge");
                    bridges++;
                }
            }

            // update low number - ignore reverse of edge leading to v
            else if (w != u)
                low[v] = Math.min(low[v], pre[w]);
        }
    }
```

## 4.5 Dilworth Theorem

An antichain in a partially ordered set is a set of elements no two of which are comparable to each other, and a chain is a set of elements every two of which are comparable. Dilworth's theorem states that there exists an antichain A, and a partition of the order into a family P of chains, such that the number of chains in the partition equals the cardinality of A. When this occurs, A must be the largest antichain in the order, for any antichain can have at most one element

from each member of P. Similarly, P must be the smallest family of chains into which the order can be partitioned, for any partition into chains must have at least one chain per element of A. The width of the partial order is defined as the common size of A and P.

An equivalent way of stating Dilworth's theorem is that, in any finite partially ordered set, the maximum number of elements in any antichain equals the minimum number of chains in any partition of the set into chains. A version of the theorem for infinite partially ordered sets states that, in this case, a partially ordered set has finite width w if and only if it may be partitioned into w chains, but not less.

## 4.6 Edmonds Karp

```
// solution to problem kill the werewolf latin america icpc 2015
// maybe ignore main, important part in cannot_win procedure

#include <bits/stdc++.h>
using namespace std;
#define MAX 110
#define INF 100000000
#define max(a, b) (((a)>(b))?(a):(b))
#define min(a, b) (((a)<(b))?(a):(b))

bool e[MAX][MAX]; // adj matrix e[u][v] = true iff edge (u,v)
int n, votes[MAX][2], freq[MAX];
int g[MAX][MAX]; // adj matrix of capacity of each edge
int f[MAX][MAX]; // adj matrix of flow going through each edge
int p[MAX]; // parents of each node (used by bfs)
int m[MAX]; // flow thourgh each node (used by bfs)

int bfs() {
    // initialize flow entering each node & parents
        for (int u = 0; u < MAX; u++) {
        m[u] = INF;
        p[u] = -1;
    }
    // do the bfs procedure from source s (0)
        queue<int> q;
    q.push(0);
        while (q.size() > 0) {
               int u = q.front(); q.pop();
               for (int v = 1; v <= 2*n+1; v++) {
          // check if neighbors, capacity is > 0 & not visited yet
                    if (e[u][v] && g[u][v]-f[u][v] > 0 && p[v] == -1) {
              // update parent and flow going through node v
                                p[v] = u;
              m[v] = min(m[u], g[u][v]-f[u][v]);
              q.push(v);
              // finish if sink was reached
                                if (v == 2*n+1) {
                  return m[v];
              }
                     }
              }
        }
    // no augmenting path found, return 0
        return 0;
}

bool cannot_win() {
        int req = 0, tot = 0, curr, v;
        for (int u = 1; u <= n; u++) {
        req += g[0][u];
    }
    // clean flow as initially its 0
        for (int u = 0; u < MAX; u++) {
        memset(f[u], 0, sizeof(f[u]));
    }
    // while there are augmenting paths
        while ((curr = bfs()) > 0) {
               tot += curr; // add flow found on curr path
        v = 2*n+1; // this is the sink
               while (v != 0) {
            f[p[v]][v] += curr; // increase flow f(u, v) by curr
            f[v][p[v]] -= curr; // decrease flow f(v, u) by curr
            v = p[v]; // go back until source
        }
        }
        return (req == tot);
}

int main() {
        iostream::sync_with_stdio(false); cin.tie(NULL);
        cin >> n;
        memset(freq, 0, sizeof(freq));
        for (int u = 1; u <= n; u++) {
               cin >> votes[u][0] >> votes[u][1];
```

```
        freq[votes[u][0]]++, freq[votes[u][1]]++;
    }
    int ans = 0;
    for (int i = 1; i <= n; i++) {
    // this is the graph construction part
        for (int u = 0; u < MAX; u++) {
            memset(g[u], 0, sizeof(g[u]));
            memset(e[u], false, sizeof(e[u]));
        }
        for (int u = 1; u <= n; u++) {
            g[0][u] = (u != i && votes[u][0] != i &&
                votes[u][1] != i);
            e[0][u] = e[u][0] = g[0][u];
        }
        for (int u = 1; u <= n; u++) {
            for (int v = 1; v <= n; v++) {
                g[u][v+n] = (votes[u][0] == v || votes[u][1]
                    == v);
                e[u][v+n] = e[v+n][u] = g[u][v+n];
            }
        }
        for (int u = 1; u <= n; u++) {
            e[u+n][2*n+1] = e[2*n+1][u+n] = true;
            g[u+n][2*n+1] = freq[i]-1-(votes[i][0] == u ||
                votes[i][1] == u);
        }

    // this is the maxflow calculation part
        ans += cannot_win();
    }
    cout << n-ans << '\n';
}
```

### 4.7 Konig Theorem

Given some bipartite graph, the cardinality of the vertex cover is the same as the cardinality of the maximum matching.

## 5 Math

### 5.1 Diophantine

```
// solve equations of the form ax + by = c
// if c%gcd(a, b) != 0 there are no solutions.
// otherwise find some solution a(xg)+b(yg)=g (g = gcd(a,b))
// using extended euclidean algorithm.

// first solution is x0 = (xg) * x/c and y0 = (yg) * c/g
// all other solutions are of the form x = x0 + k * b/g, y = y0 - k * a/g
// this is true because equality ax + by = g holds

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

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

bool find_any_solution(int a, int b, int c, int &x0, int &y0, int &g) {
    g = gcd(abs(a), abs(b), x0, y0);
    if (c % g) {
        return false;
    }

    x0 *= c / g;
    y0 *= c / g;
    if (a < 0) x0 = -x0;
    if (b < 0) y0 = -y0;
    return true;
}

void shift_solution (int & x, int & y, int a, int b, int cnt) {
    x += cnt * b;
    y -= cnt * a;
}

// ranges are inclusive
int find_all_solutions (int a, int b, int c, int minx, int maxx, int
    miny, int maxy) {
```

```
int x, y, g;
if (! find_any_solution (a, b, c, x, y, g)) {
    return 0;
}
a /= g; b /= g;

int sign_a = a>0 ? +1 : -1;
int sign_b = b>0 ? +1 : -1;

shift_solution (x, y, a, b, (minx - x) / b);
if (x < minx) {
    shift_solution (x, y, a, b, sign_b);
}
if (x > maxx) {
    return 0;
}
int lx1 = x;

shift_solution (x, y, a, b, (maxx - x) / b);
if (x > maxx) {
    shift_solution (x, y, a, b, -sign_b);
}
int rx1 = x;

shift_solution (x, y, a, b, - (miny - y) / a);
if (y < miny) {
    shift_solution (x, y, a, b, -sign_a);
}
if (y > maxy) {
    return 0;
}
int lx2 = x;

shift_solution (x, y, a, b, - (maxy - y) / a);
if (y > maxy) {
    shift_solution (x, y, a, b, sign_a);
}
int rx2 = x;

if (lx2 > rx2) {
    swap (lx2, rx2);
}
int lx = max (lx1, lx2);
int rx = min (rx1, rx2);
    if (lx > rx) return 0;
    return (rx - lx) / abs(b) + 1;
}

int main() {
    // find any solution for 5x+3y = 20 -- x = -20, y = 40
    int x, y, g;
    find_any_solution(5, 3, 20, x, y, g);
    cout << "x = " << x << ", y = " << y << "\n";

    // find solutions for 5x+3y = 20 in range x[-100..100], y[-200..200]
        -- 67
    cout << find_all_solutions(5, 3, 20, -100, 100, -200, 200) << "\n";
}
```

## 5.2 Inclusion Exclusion Principle

$$\begin{aligned}\left|\bigcup_{i=1}^{n} A_i\right| = &\sum_{i=1}^{n} |A_i| \\ &- \sum_{1\le i<j\le n} |A_i \cap A_j| \\ &+ \sum_{1\le i<j<k\le n} |A_i \cap A_j \cap A_k| \\ &- \ldots + (-1)^{n-1} \left|\bigcap_{i=1}^{n} A_i\right|\end{aligned}$$

How many ways to chose $1, 2, ..., n$ sets.

## 5.3 Linear Sieve

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

// O(n) sieve for all primes
vector<int> linear_sieve(int n) {
    vector<int> primes;
    vector<bool> is_composite(n, false);
    for (int i = 2; i < n; ++i) {
        // no composite, so must be prime
```

```
        if (!is_composite[i]) {
            primes.push_back(i);
        }
        // all smallest primes will be multiplied by every i
        for (int j = 0; j < primes.size() && primes[j]*i < n; ++j) {
            is_composite[i*primes[j]] = true; // can be writter as i*p
            // future primes wont be smallest, break
            if (i%primes[j] == 0) {
                break;
            }
        }
    }
    return primes;
}

int main() {
    // get all primes between up to 100 (exclusive)
    vector<int> primes = linear_sieve(100);

    // 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83
        89 97
    for (int prime : primes) {
        cout << prime << " ";
    }
    cout << "\n";
}
```

## 5.4 Multiplicative Function

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

// Function f is multiplicative if:
// For every pair of co-prime integers p, q (gcd(p, q) = 1), then f(p*q)
    = f(p)*f(q)
// Unless for all n, f(n) = 0. We know that f(1) must be 1 as f(n) =
    f(n)*f(1)
// f(n) & g(n) are identical iff for every prime p and number k >= 0 then
    f(p^k) = g(p^k)
// So we only need to know about f(p^k)

// Example of linear sieve for calculating phi(n) multiplicative function
    (euler totient)
vector<int> euler_totient(int n) {
    vector<int> primes, phi(n);
    vector<bool> is_composite(n, false);
    for (int i = 2; i < n; ++i) {
        if (!is_composite[i]) {
            // i is prime, so get f(i) manually, for totient is i-1
            primes.push_back(i);
            phi[i] = i-1;
        }
        for (int j = 0; j < primes.size() && i*primes[j] < n; ++j) {
            if (i%primes[j] == 0) {
                // i and primes[j] are NOT co-prime, we need to find a
                    relation
                // in the case of euler totient, phi(i*primes[j]) =
                    primes[h]*phi(i)
                phi[i*primes[j]] = phi[i]*primes[j];
                break; // break as further primes wont be smallest
            } else {
                // i and primes[j] are co-prime, so f(i*primes[j]) =
                    f(i)*f(primes[j])
                phi[i*primes[j]] = phi[i]*phi[primes[j]];
            }
        }
    }
    phi[1] = 1;
    return phi;
}

int main() {
    // get number of co-primes for all numbers up to 20 (exclusive)
    vector<int> phi = euler_totient(20);

    // 0 1 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
    for (int num_coprimes : phi) {
        cout << num_coprimes << " ";
    }
    cout << "\n";
}
```

## 5.5 Sprague Grundy Theorem

```
// impartial games are equivalent to Nim.
// a game is impartial iff the only difference between
```

```
// the players is the order in which they take turns.

// each composite game can be divided into independent
// subgames (like each Nim-pile is an independent game).

// the XOR of a composite game determines the winner of
// the game when both parts play optimally.
//  * if XOR = 0, player 1 will lose
//  * otherwise, player 1 will win

// grundy number of some state is the mex of the grundy
// numbers of reachable states from that state.

// mex is the smallest non-negative number not present in a set.
// example: mex(0, 1, 2) = 3, mex(1, 2, 3) = 0, mex(0, 2) = 1

// this is the solution for the problem split game (cf gym)

#include <bits/stdc++.h>
using namespace std;
#define MAX 2001

vector<int> dp;

int grundy(int n) {
    assert(n > 0);
    if (n == 1) {
        return dp[n] = 0;
    } else if (dp[n] != -1) {
        return dp[n];
    } else {
        set<int> mex;
        for (int x = 1; x < n; ++x) {
            int q = n/x, r = n%x;
            if (q%2 == 0) {
                if (r > 0) {
                    mex.insert(grundy(r));
                } else {
                    mex.insert(0);
                }
            } else {
                if (r > 0) {
                    mex.insert(grundy(r)^grundy(x));
                } else {
                    mex.insert(grundy(x));
                }
            }
        }
        int ans = 0;
        while (mex.find(ans) != mex.end()) {
            ++ans;
        }
        return dp[n] = ans;
    }
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);

    int n, ans = 0;
    cin >> n;
    dp.assign(MAX, -1);
    for (int i = 0, v; i < n; ++i) {
        cin >> v;
        ans ^= grundy(v);
    }
    cout << (ans == 0 ? "Second\n" : "First\n");
}
```

# 6 Strings

## 6.1 Aho Corasick

```
#include <bits/stdc++.h>
using namespace std;
typedef pair<int, int> pii;
#define MAXCHAR 26 // english lowercase letters

// node for english lowercase letters */
struct node {
    int f;
    vector<int> edges;
    unordered_set<int> out;
    node() {
        f = 0;
        edges.assign(MAXCHAR, -1);
```

```
    }
};
vector<node> trie;

// add root to the trie
void build() {
    trie.push_back(node());
}

// given a word, add it to the trie
void add(int index, const string &s) {
    int curr = 0;
    for (int i = 0; i < s.size(); ++i) {
        int pos = s[i]-'a';
        if (trie[curr].edges[pos] == -1) {
            trie.push_back(node());
            trie[curr].edges[pos] = trie.size()-1;
        }
        curr = trie[curr].edges[pos];
    }
    // Mark last node as string index
    trie[curr].out.insert(index);
}

// build the aho-corasick automaton
void ahocorasick() {
    queue<int> q;
    for (int i = 0; i < MAXCHAR; ++i) {
        if (trie[0].edges[i] == -1) {
            trie[0].edges[i] = 0; // root always fail to 0
        } else {
            q.push(trie[0].edges[i]);
        }
    }
    while (q.size() > 0) {
        int u = q.front(); q.pop();
        for (int i = 0; i < MAXCHAR; ++i) {
            if (trie[u].edges[i] == -1) continue;
            // perform KMP like step
            int v = trie[u].edges[i], w = trie[u].f;
            while (trie[w].edges[i] == -1) {
                w = trie[w].f;
            }
            trie[v].f = trie[w].edges[i];
            // combine occurrences
            auto out = trie[trie[v].f].out;
            for (int pos : out) {
                trie[v].out.insert(pos);
            }
            q.push(v);
        }
    }
}

// given some word, print occurrences of added patterns on it
void search(const string &s) {
    vector<pii> ans;
    for (int i = 0, curr = 0, j; i < s.size(); ++i) {
        int pos = s[i]-'a';
        // perform KMP like step
        while (trie[curr].edges[pos] == -1) {
            curr = trie[curr].f;
        }
        curr = trie[curr].edges[pos];
        auto out = trie[curr].out;
        for (int pos : out) {
            ans.push_back({i, pos});
        }
    }

    // print answrs
    for (int i = 0; i < ans.size(); ++i) {
        cout << "{" << ans[i].first << ", " << ans[i].second << "}\n";
    }
}

int main() {
    // build the trie
    build();

    // add some patterns to the trie
    string patterns[] = {"alabala", "bala", "ilibili"};
    for (int i = 0; i < 3; ++i) {
        add(i, patterns[i]);
    }

    // build the aho-corasick automaton
    ahocorasick();
```

```
    // search for occs of pattern in given word -- {25, 1}, {25, 0}, {60,
        2}
    search("thisisatestwordwithalabalabutnotonlythatalsoalabaandilibili");
}
```

## 6.2 Hashing

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define X 137
#define MOD ((ll)1e9+7)
#define mod(n) (((n)%(MOD)+(MOD))%(MOD))

// given a string A and a target B, print the occurrences of B in A
void hashing(const string &a, const string &b) {
    // build the hash function
    string s = b + "$" + a;
    int n = s.size();
    vector<ll> h(n+1), r(n+1);
    r[0] = 1;
    for (int i = 1; i <= n; ++i) {
        h[i] = mod(h[i-1]*X+s[i-1]);
        r[i] = mod(r[i-1]*X);
    }

    // print occurrences
    int m = b.size();
    for (int i = m+1; i <= n; ++i) {
        if (mod(h[i]-mod(h[i-m]*r[m])) == h[m]) {
            cout << i-2*m-1 << '\n';
        }
    }
}

int main() {
    string a = "alabalalabala";
    string b = "bala";

    // 3, 9 (0-based index)
    hashing(a, b);
}
```

## 6.3 KMP

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

// given a string A and a target B, print the occurrences of B in A
void kmp(const string &a, const string &b) {
    // build the PI function
    string s = b + "$" + a;
    int n = s.size();
    vector<int> pi(n+1);
    for (int i = 2, j; i <= n; ++i) {
        j = pi[i-1];
        while (j > 0 && s[j] != s[i-1]) {
            j = pi[j];
        }
        j += s[j] == s[i-1];
        pi[i] = j;
    }

    // print occurrences
    int m = b.size();
    for (int i = 0; i <= n; ++i) {
        if (pi[i] == m) cout << i-2*m-1 << '\n';
    }
}

int main() {
    string a = "alabalalabala";
    string b = "bala";

    // 3, 9 (0-based index)
    kmp(a, b);
}
```

## 6.4 Suffix Array

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

// calculate the suffix array for the given word O(nlog^2n)
vector<int> suffix_array(const string &s) {
```

```
    int n = s.size();
    vector<int> sa(n), rank(n);
    vector<ll> rank2(n);

    for (int i = 0; i < n; i++) {
        sa[i] = i;
        rank[i] = s[i];
    }
    for (int len = 1; len < n; len *= 2) {
        for (int i = 0; i < n; i++) {
            rank2[i] = ((ll) rank[i] << 32) + (i+len < n ? rank[i+len] :
                -1);
        }
        sort(sa.begin(), sa.end(), [&](int i, int j) {
            return rank2[i] < rank2[j];
        });
        for (int i = 0; i < n; i++) {
            if (i > 0 && rank2[sa[i]] == rank2[sa[i-1]]) {
                rank[sa[i]] = rank[sa[i-1]];
            } else {
                rank[sa[i]] = i;
            }
        }
    }
    return sa;
}

// calculate the lcp array for the given word + suffix array
vector<int> lcp_array(const vector<int> &sa, const string &s) {
    int n = s.size();
    vector<int> rank(n);
    for (int i = 0; i < n; i++) {
        rank[sa[i]] = i;
    }

    vector<int> ans(n);
    for (int i = 0, l = 0; i < n; i++) {
        if (rank[i] > 0) {
            int j = sa[rank[i]-1];
            while (s[i+l] == s[j+l]) {
                l++;
            }
            ans[rank[i]] = l > 0 ? l-- : l;
        }
    }
    return ans;
}

int main() {
    // create word and append min char
    string s = "banana";
    s += char(0);

    // $
    // a$
    // ana$
    // anana$
    // banana$
    // na$
    // nana$

    // calculate suffix array -- 6, 5, 3, 1, 0, 4, 2
    vector<int> sa = suffix_array(s);
    for (int i = 0; i < sa.size(); ++i) {
        cerr << sa[i] << " \n"[i == sa.size()-1];
    }

    // calculate lcp array -- 0, 0, 1, 3, 0, 0, 2
    vector<int> lcp = lcp_array(sa, s);
    for (int i = 0; i < lcp.size(); ++i) {
        cerr << lcp[i] << " \n"[i == lcp.size()-1];
    }
}
```

## 6.5 Trie

```
#include <bits/stdc++.h>
using namespace std;
#define MAXCHAR 26 // english lowercase letters

struct node {
    bool is_end;
    vector<int> edges;
    node() {
        is_end = false;
        edges.assign(MAXCHAR, -1);
    }
};
```

```
vector<node> trie;

// add root to the trie
void build() {
    trie.push_back(node());
}

// given some word s, add it to the trie
void add(const string &s) {
    int curr = 0;
    for (int i = 0; i < s.size(); ++i) {
        int pos = s[i]-'a';
        if (trie[curr].edges[pos] == -1) {
            trie.push_back(node());
            trie[curr].edges[pos] = trie.size()-1;
        }
        curr = trie[curr].edges[pos];
    }
    // mark last node as an end
    trie[curr].is_end = true;
}

// given some word, check if it exist in the trie
bool exist(const string &s) {
    int curr = 0;
    for (int i = 0; i < s.size(); ++i) {
        int pos = s[i]-'a';
        if (trie[curr].edges[pos] == -1) {
            return false;
        }
        curr = trie[curr].edges[pos];
    }
    // string exist if it's an end (not only a preffix)
    return trie[curr].is_end;
}

int main() {
    // build the trie
    build();

    // add some words to the trie
    add("alabalafirstword");
    add("alabalasecondword");
    add("thirdword");

    // query for some words existence -- true, false, false
    cout << exist("alabalasecondword") << "\n";
    cout << exist("thirdwor") << "\n";
    cout << exist("notexistingword") << "\n";
}
```

## 6.6 Z Algorithm

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

// given a string A and a target B, print the occurrences of B in A
void zfunc(const string &a, const string &b) {
    // build the Z function
    string s = b + "$" + a;
    int n = s.size();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0, k; i < n; ++i) {
        if (i > r) {
            l = r = i;
            while (r < n && s[r-l] == s[r]) {
                ++r;
            }
            z[i] = r-l;
            --r;
        } else {
            k = i-l;
            if (z[k] < r-i+1) {
                z[i] = z[k];
            } else {
                l = i;
                while (r < n && s[r-l] == s[r]) {
                    ++r;
                }
                z[i] = r-l;
                --r;
            }
        }
    }

    // print occurrences
    int m = b.size();
    for (int i = 0; i < n; ++i) {
```

```
        if (z[i] == m) {
            cout << i-m-1 << '\n';
        }
    }
}

int main() {
    string a = "alabalalabala";
    string b = "bala";

    // 3, 9 (0-based index)
    zfunc(a, b);
}
```