Not Stanford University Team Notebook (2018-19)

Contents

1 Everything

1.1 Utilities

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

using namespace std;

using ll = long long;
using ull = unsigned long long;


// find the first x in [start, end] inclusive on both, such that [condition(x) == true]
// [start, end] should be sorted, such that there is only
// a single point at which [condition(i) == false] and [condition(i + 1) == true]
// condition can be `auto lower_bound = [arr, v](ull i) { return arr[i] >= v; };`
template <typename F>
ull bin_search(ull start, ull end, F condition) {
    ull l = start, r = end;
    while (l < r) {
        ull mid = (l + r) / 2;
        condition(mid) ? (r = mid) : (l = mid + 1);
    }
    return l; // first index that fulfils the condition
}

// print container elements with space in between and end line after them
// container must not be empty!
template <typename T>
void print_elements(const T& v) {
    copy(v.begin(), prev(v.end()), ostream_iterator<typename T::value_type>(cout, " "));
    cout << *prev(v.end()) << endl; // can change from space above and remove new line here
}


int main() {
    ios_base::sync_with_stdio(false); // stop sync between cin/cout and scanf/printf
    cin.tie(nullptr);                 // stop flushing stdout whenever stdin is used

    vector<ull> arr = {1, 2, 3, 3, 3, 3, 5, 6};
    // v = 0   both   [v  v  v  v  v  v  v  v] -> arr[0] = 1; for both
    // v = 1   lb     [v  v  v  v  v  v  v  v] -> arr[0] = 1; for lb
    //         ub     [x  v  v  v  v  v  v  v] -> arr[1] = 2; for ub
    // v = 3   lb     [x  x  v  v  v  v  v  v] -> arr[2] = 3; for lb
    //         ub     [x  x  x  x  x  x  v  v] -> arr[6] = 5; for ub
    // v = 4   both   [x  x  x  x  x  x  v  v] -> arr[6] = 5; for both
    // v = 6   lb     [x  x  x  x  x  x  x  v] -> arr[7] = 6; for lb
    //         ub     [x  x  x  x  x  x  x  x] -> arr[7] = 6; for ub - special case
    // v = 9   both   [x  x  x  x  x  x  x  x] -> arr[7] = 6; for both

    for (ull v : {0, 1, 3, 4, 6, 9}) {
        auto lower_bound = [arr, v](ull i) { return arr[i] >= v; };
        auto upper_bound = [arr, v](ull i) { return arr[i] > v; };

        ull lb = bin_search(0, arr.size() - 1, lower_bound);
        ull ub = bin_search(0, arr.size() - 1, upper_bound);

        cout << v << " -> ";
        cout << "arr[" << lb << "] = " << arr[lb] << "; ";
        cout << "arr[" << ub << "] = " << arr[ub] << endl;
    }

    // iterate on all permutations
    vector<ll> p = {1, 2, 3};
    do { // 1 2 3 -> 1 3 2 -> 2 1 3 -> 2 3 1 -> 3 1 2 -> 3 2 1
        print_elements(p);
    } while (next_permutation(p.begin(), p.end()));
    print_elements(p); // ends back with 1 2 3
```

```
    cout << endl;
    vector<ll> q = {1, 3, 3};
    do { // 1 3 3 -> 3 1 3 -> 3 3 1
        print_elements(q);
    } while (next_permutation(q.begin(), q.end()));
    print_elements(q); // ends back with 1 3 3

    // min/max between two numbers only, use more than once for more
    ll n1 = 123, n2 = 987, n3 = -555;
    cout << min(min(n1, n2), n3) << endl; // -555
    cout << max(max(n1, n2), n3) << endl; // 987

    // min/max between all element in a range, returns iterator, de-ref for the value
    vector<ll> nums = {34, -125, 452, -23, 12, -54, 234, -627};
    cout << *min_element(nums.begin(), nums.end()) << endl; // -627
    cout << *max_element(nums.begin(), nums.end()) << endl; // 452

    // string to lower and string to upper usage
    string str = "Some String";
    transform(str.begin(), str.end(), str.begin(), ::tolower);
    cout << str << endl; // some string
    transform(str.begin(), str.end(), str.begin(), ::toupper);
    cout << str << endl; // SOME STRING

    // example use of distance(it1, it2), next(it, by_n), prev(it, by_n)
    cout << distance(arr.begin(), next(arr.begin(), 2)) << endl; // 2
    cout << distance(prev(arr.end(), 3), arr.end()) << endl;     // 3

    // erase only one element from multi set vs all elements
    multiset<ll> ms = {1, 2, 3, 4, 5, 5, 5, 5, 5, 6, 7, 7};
    print_elements(ms);           // 1 2 3 4 5 5 5 5 5 6 7 7
    ms.erase(ms.lower_bound(5)); // element must exist
    print_elements(ms);           // 1 2 3 4 5 5 5 5 6 7 7
    ms.erase(5);                  // this will remove all occurrences of 5
    print_elements(ms);           // 1 2 3 4 6 7 7
    ms.erase(ms.lower_bound(5)); // if it does not exist
    print_elements(ms);           // 1 2 3 4 7 7
}
```

1.2 Union Find

```
#include <vector>

using namespace std;

using ull = unsigned long long;

struct UnionFind {
    vector<ull> rank;
    vector<ull> parent;

    explicit UnionFind(ull size) {
        rank = vector<ull>(size, 0);
        parent = vector<ull>(size);
        for (ull i = 0; i < size; i++) {
            parent[i] = i;
        }
    }

    ull find(ull x) {
        ull tmp = x; // variable only used in log* code
        while (x != parent[x]) {
            x = parent[x];
        }
        while (tmp != x) { // for log*, not needed most of the time
            ull next = parent[tmp];
            parent[tmp] = x;
            tmp = next;
        } // end of log* code
        return x;
    }

    void unite(ull p, ull q) {
        p = find(p);
        q = find(q);
        if (q == p) { // already in the same group
            return;
        }
        if (rank[p] < rank[q]) { // add shorter to longer
            parent[p] = q;
        }
        else {
            parent[q] = p;
        }
        if (rank[p] == rank[q]) { // update rank if needed
            rank[p]++;
```

```
        }
    }
};
```

1.3 Fenwick Tree

```
#include <array>

using namespace std;

using ll = long long;
using ull = unsigned long long;


/// zero-based Fenwick tree with N elements
template <ull N>
struct FenwickTree {
    array<ll, N + 1> FT = {0};

    void reset() { // reset tree
        FT = {0};
    }

    void init(ull i, ll v) { // pre-build set value at position i
        FT[i + 1] = v;
    }

    void build() { // build the tree in O(n)
        for (ull i = 1; i <= N; ++i) {
            ull j = i + (i & -i);
            if (j <= N)
                FT[j] += FT[i];
        }
    }

    /// must have function. the rest are optional.
    void update(ull i, ll d) { // post-build update value at position i
        for (++i; i <= N; i += i & -i)
            FT[i] += d;
    }

    /// must have function. the rest are optional.
    ll query_0i(ull i) { // query the tree from 0 to i inclusive on both ends
        ll sum = 0;
        for (++i; i > 0; i -= i & -i)
            sum += FT[i];
        return sum;
    }

    ll query_r(ull l, ull r) { // query the tree from l to r inclusive on both ends
        return query_0i(r) - query_0i(l - 1);
    }

    ll read(ull i) { // read value at position i
        return query_r(i, i);
    }

    void set(ull i, ll v) { // post-build set value at position i
        update(i, v - read(i));
    }

    void update_r(ull l, ull r, ll d) { // post-build update values from l to r inclusive on both ends
        for (ull i = l; i <= r; ++i)
            update(i, d);
    }
};


/// zero-based Fenwick tree with N elements, with range update. not good for big numbers.
template <ull N>
struct FenwickTree2 {
    FenwickTree<N + 1> FT1;
    FenwickTree<N + 1> FT2;

    void reset() { // reset tree
        FT1.reset();
        FT2.reset();
    }

    void init(ull i, ll v) { // pre-build set value at position i
        FT1.FT[i] = v;
    }

    void build() { // build the tree in O(n)
        for (ull v = 0, i = N; i > 0; --i) {
            v = FT1.FT[i];
```

```
            FT1.FT[i + 1] -= v;
            FT2.FT[i] += v * (i - 1);
            FT2.FT[i + 1] -= v * i;
        }
        FT1.build();
        FT2.build();
    }

    /// must have function. the rest are optional.
    void update_r(ull l, ull r, ll d) { // post-build update values from l to r inclusive on both ends
        FT1.update(l, d);
        FT1.update(r + 1, -d);
        FT2.update(l, d * (l - 1));
        FT2.update(r + 1, -d * r);
    }

    /// must have function. the rest are optional.
    ll query_0i(ull i) { // query the tree from 0 to i inclusive on both ends
        return FT1.query_0i(i) * i - FT2.query_0i(i);
    }

    ll query_r(ull l, ull r) { // query the tree from l to r inclusive on both ends
        return query_0i(r) - query_0i(l - 1);
    }

    ll read(ull i) { // read value at position i
        return query_r(i, i);
    }

    void update(ull i, ll d) { // post-build update value at position i
        update_r(i, i, d);
    }

    void set(ull i, ll v) { // post-build set value at position i
        update_r(i, i, v - read(i));
    }
};
```

1.4 Segment Tree

```
#include <array>


using namespace std;

using ll = long long;
using ull = unsigned long long;


/// zero-based segment tree with T_N elements
using t_type = ll;
constexpr t_type T_START_V = 0;
constexpr t_type T_QUERY_V = 0;
#define T_OP +

template <ull N>
struct SegmentTree {
    array<t_type, 2 * N> ST = {T_START_V};

    void reset() { // reset tree
        ST = {T_START_V};
    }

    void init(ull i, t_type v) { // pre-build set value at position i
        ST[i + N] = v;
    }

    void build() { // build the tree in O(n)
        for (ull i = N - 1; i > 0; --i)
            ST[i] = ST[i << 1] T_OP ST[i << 1 | 1];
    }

    /// must have function. the rest are optional.
    void set(ull i, t_type v) { // post-build set value at position i
        for (ST[i += N] = v; i > 1; i >>= 1)
            ST[i >> 1] = ST[i] T_OP ST[i ^ 1];
    }

    /// must have function. the rest are optional.
    t_type query_r(ull l, ull r) { // from l to r inclusive on both ends
        t_type res = T_QUERY_V;
        for (l += N, r += N + 1; l < r; l >>= 1, r >>= 1) {
            if (l & 1)
                res = res T_OP ST[l++];
            if (r & 1)
                res = res T_OP ST[--r];
```

```
        }
        return res;
    }

    ll read(ull i) { // read value at position i
        return query_r(i, i);
    }

    void update(ull i, t_type d) { // post-build update value at position i
        set(i, read(i) + d);
    }

    void update_r(ull l, ull r, ll d) { // post-build update values from l to r inclusive on both ends
        for (ull i = l; i <= r; ++i)
            update(i, d);
    }
};
```

1.5 Math

```
#include <cmath>
#include <vector>

using namespace std;

using ll = long long;
using ull = unsigned long long;

// compute (a^q) mod n. should be used with q >= 0 and n >= 1
ll powmodn(ll a, ll q, ll n) {
    ll res = 1;
    while (q) {
        if (q % 2)
            res = (res * a) % n;
        a = (a * a) % n;
        q >>= 1;
    }
    return res;
}

// compute gcd(a, b) aka greatest common divisor
ll gcd(ll a, ll b) {
    while (b) {
        a %= b;
        swap(a, b);
    }
    return a;
}

// compute lcm(a, b) aka least common multiple
ll lcm(ll a, ll b) {
    return a * (b / gcd(a, b)); // order: divide before multiplying!
}

// compute the value of (n choose k)
ull n_choose_k(ull n, ull k) {
    if (k > n)
        return 0;
    if (k * 2 > n)
        k = n - k;
    if (k == 0)
        return 1;

    ull result = n;
    for (ull i = 2; i <= k; ++i) {
        result *= (n - i + 1);
        result /= i;
    }
    return result;
}

// return true if n is a prime number, better for (n > 2^20) / (n > 10^6) values
bool MillerRabin(ull n, ll k = 5) {
    if (n == 2 or n == 3) {
        return true;
    }
    if (n < 5) {
        return false;
    }
    ull m = n - 1;
    ull r = 0;
    while (m % 2 == 0) {
        m >>= 1;
        r += 1;
    }
    // !!! Deterministic version for n<2^64 !!!
```

```
    ull dtrm_set[12] = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
    for (ull a : dtrm_set) {
        if (a >= n)
            break;
        // !!! End of deterministic version !!! */

        /* // !!! Probabilistic version !!!
        while (k--) {
            ull a = rand() % (n - 4) + 2;
            // !!! end of probabilistic version !!! */
        a = powmodn(a, m, n);
        if (a == 1)
            continue;
        ll i = r;
        while (i-- and a != n - 1) {
            a = (a * a) % n;
            if (a == 1)
                return false;
        }
        if (i == -1)
            return false;
    }
    return true;
}

// return true if n is a prime number, better for (n < 2^20) / (n < 10^6) values
bool is_prime(ull n) {
    if (n < 2)
        return false;
    ull sqrtn = (ull)sqrt(n);
    for (ull i = 2; i <= sqrtn; ++i) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}

// return bool array where (ps[n] == true) iff n is a prime number, until n inclusive
vector<bool> sieve(ull n) {
    vector<bool> ps(n + 1, true);
    ps[0] = false;
    ps[1] = false;
    ull sqrtn = (ull)sqrt(n);
    for (ull i = 0; i <= sqrtn; ++i) {
        if (ps[i]) {
            for (ull j = i * i; j <= n; j += i) {
                ps[j] = false;
            }
        }
    }
    return ps;
}

// return an array of primes that are less or equal to n
vector<ull> sieve_primes(ull n) {
    vector<ull> primes;
    auto pbools = sieve(n);
    for (ull i = 0; i < pbools.size(); ++i) {
        if (pbools[i])
            primes.push_back(i);
    }
    return primes;
}

// return array of fibonacci numbers smaller or equal to n
// fib[0..15] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610}
vector<ull> fibonacci_numbers(ull n) {
    ull f0 = 0, f1 = 1;

    vector<ull> fibs;
    fibs.push_back(0);

    if (n < 1)
        return fibs;

    while (f1 <= n and fibs.size() < 94) { // prevent inf loop from overflow
        fibs.push_back(f1);
        ull next = f0 + f1;
        f0 = f1;
        f1 = next;
    }
    return fibs;
}

// compute the factors of a number, good if used a few amount of times (1..1000)
vector<ull> factorizationA(ull n) {
    vector<ull> facs;
    for (ull d = 2; d * d <= n; d++) {
```

```
        while (n % d == 0) {
            facs.push_back(d);
            n /= d;
        }
    }
    if (n > 1)
        facs.push_back(n);
    return facs;
}

// compute the factors of a number, good if used a lot of times (>100)
template <ull N> // the highest number that is going to be factorized
vector<ull> factorizationB(ull n) {
    static const auto primes = sieve_primes((ull)sqrt(N));
    vector<ull> facs;
    for (auto p : primes) {
        if (p * p > n)
            break;
        while (n % p == 0) {
            facs.push_back(p);
            n /= p;
        }
    }
    if (n > 1)
        facs.push_back(n);
    return facs;
}
```

1.6 Geometry

```
#define _USE_MATH_DEFINES
#include <algorithm>
#include <cmath>
#include <stack>
#include <vector>

using ll = long long;
using ull = unsigned long long;

using namespace std;

static constexpr double EPS = 1e-9;
static constexpr double INF = 1e9;

double DEG_to_RAD(double d) {
    return d * M_PI / 180.0;
}
double RAD_to_DEG(double r) {
    return r * 180.0 / M_PI;
}

/// type of points can be chosen here, but some modification are also needed in some functions.
using pt_type = double;
// using pt_type = ll;
struct pt {
    pt_type x, y;

    pt() = default;

    pt(pt_type _x, pt_type _y) : x(_x), y(_y) {
    }
    pt operator+(const pt& p) const {
        return {x + p.x, y + p.y};
    }

    pt operator-(const pt& p) const {
        return {x - p.x, y - p.y};
    }

    pt operator-() const {
        return {-x, -y};
    }

    pt_type cross(const pt& p) const {
        return x * p.y - y * p.x;
    }

    pt_type dot(const pt& p) const {
        return x * p.x + y * p.y;
    }

    pt_type cross(const pt& a, const pt& b) const { // cross of a and b
        return (a - *this).cross(b - *this);         // with this point as the origin
    }

    pt_type dot(const pt& a, const pt& b) const { // dot of a and b
```

```
        return (a - *this).dot(b - *this);        // with this point as the origin
    }

    pt_type len_sqr() const { // X^2 + Y^2 without taking the root
        return this->dot(*this);
    }

    // *** start of code for near precision compares ***
    bool operator<(const pt& p) const { // lexicographical smaller than
        if (abs(x - p.x) > EPS)         // useful for sorting
            return x < p.x;             // first criteria , by x-coordinate
        return y < p.y;                 // second criteria, by y-coordinate
    }

    // use EPS (1e-9) when testing equality of two floating points
    bool operator==(const pt& p) const {
        return (abs(x - p.x) < EPS) and (abs(y - p.y) < EPS);
    }
    // *** end of code for near precision compares *** */

    /* // *** start of code for exact compares ***
    bool operator<(const pt& p) const {           // lexicographical smaller than
        return x < p.x or (x == p.x and y < p.y); // useful for sorting
    }

    bool operator==(const pt& p) const {
        return x == p.x and y == p.y;
    }
    // *** end of code for exact compares *** */
};

double dist(pt p1, pt p2) { // Euclidean distance
    // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
    return hypot(p1.x - p2.x, p1.y - p2.y); // return double
}

// returns true if point o is on the left side of line ab
bool ccw(pt a, pt b, pt o) { // algorithms are using me
    return o.cross(a, b) > EPS;
}

// returns true if point o is on the right side of line ab
bool cw(pt a, pt b, pt o) {
    return o.cross(a, b) < EPS;
}

// returns true if point o is on the same line as the line ab
bool collinear(pt a, pt b, pt o) {
    return abs(o.cross(a, b)) < EPS; // case for near values (doubles)
    // return o.cross(a, b) == 0; // case for exact values (ll)
}

pt_type sgn(pt_type val) { // return 1, 0 or -1 for positive, zero or negative values respectively
    return val > 0 ? 1 : (val == 0 ? 0 : -1);
}

pt scale(pt v, double s) {      // non-negative s = [<1  ..  1 .. >1]
    return {v.x * s, v.y * s}; //                    shorter same longer
}

double angle(pt a, pt o, pt c) {                     // returns angle aob in rad (0 <= x <= PI)
    return abs(atan2(o.cross(a, c), o.dot(a, c))); // remove abs for (-PI <= x <= PI)
}

pt circumcircle(pt a, pt b, pt c) { // find the center of a circle with points (a, b, c)
    pt BA = b - a, CA = c - a;
    double BAS = BA.len_sqr(), CAS = CA.len_sqr(), D = 0.5 / BA.cross(CA);
    return a + scale({CA.y * BAS - BA.y * CAS, BA.x * CAS - CA.x * BAS}, D);
}

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double dist_to_line(pt p, pt a, pt b, pt& c) {
    // formula: c = a + u * ab
    pt ap = p - a, ab = b - a;
    double u = 1.0 * ap.dot(ab) / ab.len_sqr();
    c = a + scale(ab, u);
    return dist(p, c); // Euclidean distance between p and c
}

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double dist_to_line_segment(pt p, pt a, pt b, pt& c) {
    pt ap = p - a, ab = b - a;
    double u = 1.0 * ap.dot(ab) / ab.len_sqr();
    if (u < 0.0) {
        c = pt(a.x, a.y);  // closer to a
```

```
        return dist(p, a); // Euclidean distance between p and a
    }
    if (u > 1.0) {
        c = pt(b.x, b.y);  // closer to b
        return dist(p, b); // Euclidean distance between p and b
    }
    return dist_to_line(p, a, b, c); // run distToLine as above
}


struct line {
    double a, b, c;
}; // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line& l) {
    if (abs(p1.x - p2.x) < EPS) { // vertical line is fine
        l.a = 1.0;
        l.b = 0.0;
        l.c = -p1.x; // default values
    }
    else {
        l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
        l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0
        l.c = -(double)(l.a * p1.x) - p1.y;
    }
}

bool areParallel(line l1, line l2) { // check coefficients a & b
    return (abs(l1.a - l2.a) < EPS) && (abs(l1.b - l2.b) < EPS);
}

bool areSame(line l1, line l2) { // also check coefficient c
    return areParallel(l1, l2) && (abs(l1.c - l2.c) < EPS);
}

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point& p) {
    if (areParallel(l1, l2))
        return false; // no intersection
    // solve system of 2 linear algebraic equations with 2 unknowns
    p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
    // special case: test for vertical line to avoid division by zero
    if (abs(l1.b) > EPS)
        p.y = -(l1.a * p.x + l1.c);
    else
        p.y = -(l2.a * p.x + l2.c);
    return true;
}


vector<pt> convex_hull(vector<pt> pts) {
    auto POP = [](stack<pt>& s) {
        pt p = s.top(); // this is just
        s.pop();        // such a really
        return p;       // painful API
    };

    // upper hull
    sort(pts.begin(), pts.end());
    stack<pt> stk_up;
    stk_up.push(pts[0]);
    stk_up.push(pts[1]);
    for (ull i = 2; i < pts.size(); i++) {
        while (stk_up.size() >= 2) {
            pt p = POP(stk_up);
            if (ccw(pts[i], p, stk_up.top())) {
                stk_up.push(p);
                break;
            }
        }
        stk_up.push(pts[i]);
    }

    // lower hull pre
    for (auto& p : pts) {
        p = -p;
    }
    // lower hull - c&p from upper hull, f&r (stk_up)->(stk_low)
    sort(pts.begin(), pts.end());
    stack<pt> stk_low;
    stk_low.push(pts[0]);
    stk_low.push(pts[1]);
    for (ull i = 2; i < pts.size(); i++) {
        while (stk_low.size() >= 2) {
            pt p = POP(stk_low);
            if (ccw(pts[i], p, stk_low.top())) {
                stk_low.push(p);
                break;
```

```
            }
        }
        stk_low.push(pts[i]);
    }

    // build convex hull
    vector<pt> ch;
    stk_low.pop();
    while (!stk_low.empty()) {
        ch.push_back(-POP(stk_low)); // note the minus
    }
    stk_up.pop();
    while (!stk_up.empty()) {
        ch.push_back(POP(stk_up));
    }
    // add one (not both) of those if needed
    rotate(ch.begin(), prev(ch.end()), ch.end()); // ccw from most left point
    // reverse(ch.begin(), ch.end()); // ccw -> cw from most left point
    return ch;
}

bool point_in_triangle(pt a, pt b, pt c, pt p) {
    pt_type s1 = abs(a.cross(b, c)); // == triangle area, times 2
    pt_type s2 = abs(p.cross(a, b)) + abs(p.cross(b, c)) + abs(p.cross(c, a));
    return abs(s1 - s2) <= EPS;
}

// requires the result from convex hull in ccw order from the most left point
vector<pt> pre_point_in_convex_polygon(vector<pt>& pts) {
    /* // *** those lines can fix the convex hull if needed ***
    if (!ccw(pts[0], pts[1], pts[2])) {
        reverse(next(pts.begin()), pts.end());
    }
    ull pos = distance(pts.begin(), min_element(pts.begin(), pts.end()));
    rotate(pts.begin(), pts.begin() + pos, pts.end());
    // *** end of convex hull fix *** */

    vector<pt> seq;
    seq.reserve(pts.size());
    for (auto it = next(pts.begin()); it != pts.end(); ++it)
        seq.emplace_back((*it) - pts.front());
    seq.emplace_back(pts.front());
    return seq;
}

// requires the result from the pre function above
// the given point to check is normalized internally
bool point_in_convex_polygon(const vector<pt>& seq, pt p_) {
    pt base = seq.back();   // saved base is the last place in the seq
    ull n = seq.size() - 1; // effective size is (n-1), valid numbers range from 0 to (n-2)
    pt p = p_ - base;       // normalize the point using the base

    // normalized point has to be on the right side of the y-axis
    if (p.x < 0)
        return false;

    // case for when the point is on the same line as the start of the first segment
    pt_type s_p = seq[0].cross(p);
    if (s_p == 0)
        return seq[0].len_sqr() >= p.len_sqr() and seq[0].dot(p) >= 0;

    // case for when the point is on the same line as the end of the last segment
    pt_type l_p = seq[n - 1].cross(p);
    if (l_p == 0)
        return seq[n - 1].len_sqr() >= p.len_sqr() and seq[n - 1].dot(p) >= 0;

    // make sure the point is above the start of the first segment
    pt_type s_l = seq[0].cross(seq[n - 1]);
    if (sgn(s_p) != sgn(s_l))
        return false;

    // make sure the point is below the end of the last segment
    pt_type l_s = seq[n - 1].cross(seq[0]);
    if (sgn(l_p) != sgn(l_s))
        return false;

    // binary search for the angle the point is placed in
    ll l = 0, r = n - 1;
    while (r - l > 1) {
        ll mid = (l + r) / 2;
        if (seq[mid].cross(p) >= 0)
            l = mid;
        else
            r = mid;
    }
    ll pos = l;

    // check if the point is inside the triangle created there or outside of it
    return point_in_triangle(seq[pos], seq[pos + 1], pt(0, 0), p);
```

```
}

// requires the result from convex hull in ccw order (safer to make it from the most left point too)
#define INCN(x) ((x + 1) % n)
double minimum_width(vector<pt>& pts) {
    ull n = pts.size();

    if (n <= 2) // simple case
        return 0.0;

    double ret = INF;

    for (ull i = 0, j = 0; i < n; ++i) {
        while (pts[i].cross(pts[INCN(i)], pts[INCN(j)]) >= pts[i].cross(pts[INCN(i)], pts[j]))
            j = INCN(j);
        pt tmp{};
        double dist = dist_to_line(pts[j], pts[i], pts[INCN(i)], tmp);
        ret = min(ret, dist);
    }
    return ret;
}
```

1.7 Graphs

```
// This file contains implementations of some well-known graph algorithms.
// Written by Nofar Carmeli. Some code is based on the book Competitive Programming 3 by Steven and Felix Halim.

#include <algorithm>
#include <queue>
#include <set>
#include <vector>

using namespace std;

using ll = long long;
using ull = unsigned long long;

typedef pair<ll, ll> ii;
typedef pair<ll, ii> iii;

typedef vector<ii> vii;
typedef vector<vii> vvii;

typedef vector<ll> vi;
typedef vector<vi> vvi;

typedef set<ll> si;
typedef vector<si> vsi;

const ll INF = 1e9;


/*** Useful BFS ***/


// BFS on digraph g from node s:
// input: directed graph (g[u] contains the neighbors of u, nodes are named 0,1,...,|V|-1).
// input: starting node (s)
// output: distance from s to each node (d).
void bfs(const vvi& g, ll s, vector<ll>& d) {
    queue<ll> q;
    q.push(s);
    vector<bool> visible(g.size(), false);
    visible[s] = true;
    d.assign(g.size(), INF);
    d[s] = 0;
    while (!q.empty()) {
        ll u = q.front();
        q.pop();
        for (ll v : g[u]) {
            if (!visible[v]) {
                visible[v] = true;
                d[v] = d[u] + 1;
                q.push(v);
            }
        }
    }
}

/*** DFS with prints in it. is it useful? ***/
void dfs(const vvi& g, ll s) {
    stack<ll> q;
    q.push(s);
    vector<bool> visible(g.size(), false);
    visible[s] = true;
    while (!q.empty()) {
```

```
        ll u = q.top();
        q.pop();
        printf("%d\n", u);
        for (ll v : g[u])
            if (!visible[v]) {
                visible[v] = true;
                q.push(v);
            }
    }
}


/*** DFS with prints in it. is it useful? - Recursive ***/
vvi g;
vector<bool> visible;
void dfs_recursive(ll s) {
    printf("%d\n", s);
    visible[s] = true;
    for (ll u : g[s])
        if (!visible[u])
            dfs_recursive(u);
}
// In main:
//   load graph to g
//   visible.assign(g.size(), false);
//   dfs(0);

/********** Topological Sort **********/


// input: directed graph (g[u] contains the neighbors of u, nodes are named 0,1,...,|V|-1).
// output: is g a DAG (return value), a topological ordering of g (order).
// comment: order is valid only if g is a DAG.
// time: O(V+E).
bool topological_sort(const vvi& g, vi& order) {
    // compute indegree of all nodes
    vi indegree(g.size(), 0);
    for (const vi& v : g)
        for (ll u : v)
            indegree[u]++;
    // order sources first
    order = vector<ll>();
    for (ull v = 0; v < g.size(); v++)
        if (indegree[v] == 0)
            order.push_back(v);
    // go over the ordered nodes and remove outgoing edges,
    // add new sources to the ordering
    for (ull i = 0; i < order.size(); i++)
        for (ll u : g[order[i]]) {
            indegree[u]--;
            if (indegree[u] == 0)
                order.push_back(u);
        }
    return order.size() == g.size();
}


/********** Strongly Connected Components **********/


const ll UNSEEN = -1;
const ll SEEN = 1;

void KosarajuDFS(const vvi& g, ll u, vi& S, vi& colorMap, ll color) {
    // DFS on digraph g from node u:
    // visit a node only if it is mapped to the color UNSEEN,
    // Mark all visited nodes in the color map using the given color.
    // input: digraph (g), node (v), mapping:node->color (colorMap), color (color).
    // output: DFS post-order (S), node coloring (colorMap).
    colorMap[u] = color;
    for (ll v : g[u])
        if (colorMap[v] == UNSEEN)
            KosarajuDFS(g, v, S, colorMap, color);
    S.push_back(u);
}

// Compute the number of SCCs and maps nodes to their corresponding SCCs.
// input: directed graph (g[u] contains the neighbors of u, nodes are named 0,1,...,|V|-1).
// output: the number of SCCs (return value), a mapping from node to SCC color (components).
// time: O(V+E).
ll findSCC(const vvi& g, vi& components) {
    // first pass: record the 'post-order' of original graph
    vi postOrder, seen;
    seen.assign(g.size(), UNSEEN);
    for (ll i = 0; i < g.size(); ++i)
        if (seen[i] == UNSEEN)
            KosarajuDFS(g, i, postOrder, seen, SEEN);
    // second pass: explore the SCCs based on first pass result
```

```
    vvi reverse_g(g.size(), vi());
    for (ll u = 0; u < g.size(); u++)
        for (ll v : g[u])
            reverse_g[v].push_back(u);
    vi dummy;
    components.assign(g.size(), UNSEEN);
    ll numSCC = 0;
    for (ll i = g.size() - 1; i >= 0; --i)
        if (components[postOrder[i]] == UNSEEN)
            KosarajuDFS(reverse_g, postOrder[i], dummy, components, numSCC++);
    return numSCC;
}

// Computes the SCC graph of a given digraph.
// input: directed graph (g[u] contains the neighbors of u, nodes are named 0,1,...,|V|-1).
// output: strongly connected components graph of g (sccg).
// time: O(V+E).
void findSCCgraph(const vvi& g, vsi& sccg) {
    vi component;
    ll n = findSCC(g, component);
    sccg.assign(n, si());
    for (ll u = 0; u < g.size(); u++)
        for (ll v : g[u]) // for every edge u->v
            if (component[u] != component[v])
                sccg[component[u]].insert(component[v]);
}


/********** Shortest Paths **********/


// input: non-negatively weighted directed graph (g[u] contains pairs (v,w) such that u->v has weight w, nodes
    are named
// 0,1,...,|V|-1), source (s). output: distances from s (dist). time: O(ElogV).
void Dijkstra(const vvii& g, ll s, vi& dist) {
    dist = vi(g.size(), INF);
    dist[s] = 0;
    priority_queue<ii, vii, greater<ii>> q;
    q.push({0, s});
    while (!q.empty()) {
        ii front = q.top();
        q.pop();
        ll d = front.first, u = front.second;
        if (d > dist[u])
            continue; // We may have found a shorter way to get to u after inserting it to q.
        // In that case, we want to ignore the previous insertion to q.
        for (ii next : g[u]) {
            ll v = next.first, w = next.second;
            if (dist[u] + w < dist[v]) {
                dist[v] = dist[u] + w;
                q.push({dist[v], v});
            }
        }
    }
}

// input: weighted directed graph (g[u] contains pairs (v,w) such that u->v has weight w, nodes are named
// 0,1,...,|V|-1), source node (s). output: is there a negative cycle in g? (return value), the distances from s
    (d)
// comment: the values in d are valid only if there is no negative cycle. time: O(VE).
bool BellmanFord(const vvii& g, ll s, vi& d) {
    d.assign(g.size(), INF);
    d[s] = 0;
    bool changed = false;
    // V times
    for (ull i = 0; i < g.size(); ++i) {
        changed = false;
        // go over all edges u->v with weight w
        for (ull u = 0; u < g.size(); ++u) {
            for (ii e : g[u]) {
                ll v = e.first;
                ll w = e.second;
                // relax the edge
                if (d[u] < INF && d[u] + w < d[v]) {
                    d[v] = d[u] + w;
                    changed = true;
                }
            }
        }
    }
    // there is a negative cycle if there were changes in the last iteration
    return changed;
}


// input: weighted directed graph (g[u] contains pairs (v,w) such that u->v has weight w, nodes are named
// 0,1,...,|V|-1). output: the pairwise distances (d). time: O(V^3).
void FloydWarshall(const vvii& g, vvi& d) {
```

```
    // initialize distances according to the graph edges
    d.assign(g.size(), vi(g.size(), INF));
    for (ll u = 0; u < g.size(); ++u)
        d[u][u] = 0;
    for (ll u = 0; u < g.size(); ++u)
        for (ii e : g[u]) {
            ll v = e.first;
            ll w = e.second;
            d[u][v] = min(d[u][v], w);
        }
    // relax distances using the Floyd-Warshall algorithm
    for (ll k = 0; k < g.size(); ++k)
        for (ll u = 0; u < g.size(); ++u)
            for (ll v = 0; v < g.size(); ++v)
                d[u][v] = min(d[u][v], d[u][k] + d[k][v]);
}


/********** Min Spanning Tree **********/


struct UnionFind {
    vector<ull> rank;
    vector<ull> parent;

    explicit UnionFind(ull size) {
        rank = vector<ull>(size, 0);
        parent = vector<ull>(size);
        for (ull i = 0; i < size; i++) {
            parent[i] = i;
        }
    }

    ull find(ull x) {
        ull tmp = x; // variable only used in log* code
        while (x != parent[x]) {
            x = parent[x];
        }
        while (tmp != x) { // for log*, not needed most of the time
            ull next = parent[tmp];
            parent[tmp] = x;
            tmp = next;
        } // end of log* code
        return x;
    }

    void unite(ull p, ull q) {
        p = find(p);
        q = find(q);
        if (q == p) { // already in the same group
            return;
        }
        if (rank[p] < rank[q]) { // add shorter to longer
            parent[p] = q;
        }
        else {
            parent[q] = p;
        }
        if (rank[p] == rank[q]) { // update rank if needed
            rank[p]++;
        }
    }
};


// input: edges v1->v2 of the form (weight,(v1,v2)),
//        number of nodes (n), all nodes are between 0 and n-1.
// output: weight of a minimum spanning tree.
// time: O(ElogV).
ll Kruskal(vector<iii>& edges, ll n) {
    sort(edges.begin(), edges.end());
    ll mst_cost = 0;
    UnionFind components(n);
    for (iii e : edges) {
        if (components.find(e.second.first) != components.find(e.second.second)) {
            mst_cost += e.first;
            components.unite(e.second.first, e.second.second);
        }
    }
    return mst_cost;
}


/********** Max Flow **********/


ll augment(vvi& res, ll s, ll t, const vi& p, ll minEdge) {
    // traverse the path from s to t according to p.
    // change the residuals on this path according to the min edge weight along this path.
```

```
    // return the amount of flow that was added.
    if (t == s) {
        return minEdge;
    }
    else if (p[t] != -1) {
        ll f = augment(res, s, p[t], p, min(minEdge, res[p[t]][t]));
        res[p[t]][t] -= f;
        res[t][p[t]] += f;
        return f;
    }
    return 0;
}

// input: number of nodes (n), all nodes are between 0 and n-1,
//        edges v1->v2 of the form (weight,(v1,v2)), source (s) and target (t).
// output: max flow from s to t over the edges.
// time: O(VE^2) and O(EF).
ll EdmondsKarp(ll n, vector<iii>& edges, ll s, ll t) {
    // initialise adjacency list and residuals adjacency matrix
    vvi res(n, vi(n, 0));
    vvi adj(n);
    for (iii e : edges) {
        res[e.second.first][e.second.second] += e.first;
        adj[e.second.first].push_back(e.second.second);
        adj[e.second.second].push_back(e.second.first);
    }
    // while we can add flow
    ll addedFlow, maxFlow = 0;
    do {
        // save to p the BFS tree from s to t using only edges with residuals
        vi dist(res.size(), INF);
        dist[s] = 0;
        queue<ll> q;
        q.push(s);
        vi p(res.size(), -1);
        while (!q.empty()) {
            ll u = q.front();
            q.pop();
            if (u == t)
                break;
            for (ll v : adj[u])
                if (res[u][v] > 0 && dist[v] == INF) {
                    dist[v] = dist[u] + 1;
                    q.push(v);
                    p[v] = u;
                }
        }
        // add flow on the path between s to t according to p
        addedFlow = augment(res, s, t, p, INF);
        maxFlow += addedFlow;
    } while (addedFlow > 0);
    return maxFlow;
}
```

1.8 Strings - find substring in O(n) ?

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> vi;

string KMP_str; // The string to search in
string KMP_pat; // The pattern to search
vi lps;

// KMP Init
void KMP_init(){
    int m = KMP_pat.length();
    lps.resize(m,0);
    lps[0]=-1;
    int i = 0, j = -1;
    while (i < m) {
        while (j >= 0 && KMP_pat[i] != KMP_pat[j]) j = lps[j];
        i++; j++;
        lps[i] = j;
    }
}

// Search a pattern in a string
// Assuming lps is allready initialized with KMP_init
void KMP_search() {
    int n = KMP_str.length();
    int m = KMP_pat.length();
    int i = 0, j = 0;
```

```
    while (i < n) {
        while (j >= 0 && KMP_str[i] != KMP_pat[j]) j = lps[j];
        i++; j++;
        if (j == m) { // Pattern found
            cout << "The pattern is found at index " << i-j << endl;
            j = lps[j];
        }
    }
}

int main() {
    KMP_pat = "ababac";
    KMP_str = "abababacababac";
    KMP_init();
    KMP_search();
    return 0;
}
```

1.9 Strings - prefix table

```
#include <algorithm>
#include <string>
#include <iostream>

using namespace std;
typedef vector<int> vi;

string SA_str; // the input string, up to 100K characters
vi RA, tempRA; // rank array and temporary rank array
vi SA, tempSA; // suffix array and temporary suffix array
vi c; // for counting/radix sort
vi Phi,PLCP,LCP;

void countingSort(ll k) { // O(n)
    ll n = SA_str.size();
    ll i, sum, maxi = max(256, n); // up to 255 ASCII chars or length of n
    c.assign(maxi,0);
    for (i = 0; i < n; i++) // count the frequency of each integer rank
        c[i + k < n ? RA[i + k] : 0]++;
    for (i = sum = 0; i < maxi; i++) {
        ll t = c[i];
        c[i] = sum;
        sum += t;
    }
    for (i = 0; i < n; i++) // shuffle the suffix array if necessary
        tempSA[c[SA[i] + k < n ? RA[SA[i] + k] : 0]++] = SA[i];
    for (i = 0; i < n; i++) // update the suffix array SA
        SA[i] = tempSA[i];
}

void constructSA() { // this version can go up to 100000 characters
    ll n = SA_str.size();
    RA.assign(n, 0);
    tempRA.assign(n, 0);
    SA.assign(n, 0);
    tempSA.assign(n, 0);
    ll i, k, r;
    for (i = 0; i < n; i++)
        RA[i] = SA_str[i]; // initial rankings
    for (i = 0; i < n; i++)
        SA[i] = i;                  // initial SA: {0, 1, 2, ..., n-1}
    for (k = 1; k < n; k <<= 1) { // repeat sorting process log n times
        countingSort(k);            // actually radix sort: sort based on the second item
        countingSort(0);            // then (stable) sort based on the first item
        tempRA[SA[0]] = r = 0;      // re-ranking; start from rank r = 0
        for (i = 1; i < n; i++)     // compare adjacent suffixes
            tempRA[SA[i]] =         // if same pair => same rank r; otherwise, increase r
                (RA[SA[i]] == RA[SA[i - 1]] && RA[SA[i] + k] == RA[SA[i - 1] + k]) ? r : ++r;
        for (i = 0; i < n; i++) // update the rank array RA
            RA[i] = tempRA[i];
        if (RA[SA[n - 1]] == n - 1)
            break; // nice optimization trick
    }
}

void computeLCP() { // original
    ll i, L;
    ll n = SA_str.size();
    Phi.assign(n, 0);
    PLCP.assign(n, 0);
    LCP.assign(n, 0);
    Phi[SA[0]] = -1;                // default value
    for (i = 1; i < n; i++)         // compute Phi in O(n)
        Phi[SA[i]] = SA[i - 1];     // remember which suffix is behind this suffix
    for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
        if (Phi[i] == -1) {
```

```
            PLCP[i] = 0;
            continue;
        } // special case
        while (SA_str[i + L] == SA_str[Phi[i] + L])
            L++; // L increased max n times
        PLCP[i] = L;
        L = max(L - 1, 0LL); // L decreased max n times
    }
    for (i = 0; i < n; i++) // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct position
}

pair<ll, vl> computeLCP() { // used, not sure what it does though :3
    ll i, L;
    ll n = SA_str.size();
    Phi.assign(n, 0);
    PLCP.assign(n, 0);
    LCP.assign(n, 0);
    Phi[SA[0]] = -1;              // default value
    for (i = 1; i < n; i++)       // compute Phi in O(n)
        Phi[SA[i]] = SA[i - 1];   // remember which suffix is behind this suffix
    for (i = L = 0; i < n; i++) { // compute Permuted LCP in O(n)
        if (Phi[i] == -1) {
            PLCP[i] = 0;
            continue;
        } // special case
        while (SA_str[i + L] == SA_str[Phi[i] + L])
            L++; // L increased max n times
        PLCP[i] = L;
        L = max(L - 1, 0LL); // L decreased max n times
    }
    ll max_n = 1;
    vl idxs;
    LCP[0] = PLCP[SA[0]];      // put the permuted LCP to the correct position
    for (i = 1; i < n; i++) { // compute LCP in O(n)
        LCP[i] = PLCP[SA[i]]; // put the permuted LCP to the correct position
        bool a = SA_str.substr(SA[i]).find('$') != string::npos;
        bool b = SA_str.substr(SA[i - 1]).find('$') != string::npos;
        if (a != b) {
            idxs.push_back(i);
            max_n = max(max_n, LCP[i]);
        }
    }
    return make_pair(max_n, idxs);
}

int main() {
    cin >> SA_str;
    n = SA_str.length();
    SA_str += '$'; // add terminating character
    constructSA();
//    for (int i = 0; i < n; i++)
//        cout << SA[i] << "\t" << SA_str.substr(SA[i]) << endl;
    computeLCP();
    cout << "SA[i]\tLCP[i]\tSubstring" << endl;
    for (int i = 0; i < n; i++) {
        cout << SA[i] << "\t" << LCP[i] << "\t" << SA_str.substr(SA[i]) << endl;
    }
    return 0;
}
```