**Sieve:**

const int MAXN = 100000;

std::vector<int> prime;

bool is\_composite[MAXN];

void sieve(int n)

{

    std::fill(is\_composite, is\_composite + n, false);

    for (int i = 2; i < n; ++i)

    {

        if (!is\_composite[i])

            prime.push\_back(i);

        for (int j = 2; i \* j < n; ++j)

            is\_composite[i \* j] = true;

    }

}

**NGE:**

void printNGE(int arr[], int n)

{

    int next, i, j;

    for (i = 0; i < n; i++) {

        next = -1;

        for (j = i + 1; j < n; j++) {

            if (arr[i] < arr[j]) {

                next = arr[j];

                break;

            }

        }

        cout << arr[i] << " --> " << next << endl;

    }

}

**LCA:**

const int LOG = 15;

const int N = 10001;

vector<int> graph[N];

int up[N][LOG], depth[N];

void dfs(int u)

{

    for (auto v : graph[u])

    {

        depth[v] = depth[u] + 1;

        up[v][0] = u;

        for (int j = 1; j < LOG; j++)

        {

            up[v][j] = up[up[v][j - 1]][j - 1];

        }

        dfs(v);

    }

}

int get\_lca(int a, int b)

{

    if (depth[a] < depth[b])

        swap(a, b);

    int k = depth[a] - depth[b];

    for (int j = LOG - 1; j >= 0; j--)

    {

        if (k & (1 << j))

            a = up[a][j];

    }

    if (a == b)

        return a;

    for (int j = LOG - 1; j >= 0; j--)

    {

        if (up[a][j] != up[b][j])

        {

            a = up[a][j];

            b = up[b][j];

        }

    }

    return up[a][0];

}

**Kadane:**

int kadane(vector<int> &arr)

{

    int best = 0, sum = 0, i;

    for (auto it : arr)

    {

        sum = max(it, sum + it);

        best = max(best, sum);

    }

    return best;

}

**ETF:**

vector<int> phi(1000000);

void CalPhi(int n)

{

    phi[0] = 0;

    phi[1] = 1;

    for (int i = 2; i <= n; i++)

        phi[i] = i - 1;

    for (int i = 2; i <= n; i++)

        for (int j = 2 \* i; j <= n; j += i)

            phi[j] -= phi[i];

}

**Bridge articulation point:**

const int N = 1e5 + 5;

vector<int> child(N), graph[N], tin(N), low(N);

bool vis[N];

int n, m, ans, timer;

void Bridge(int u, int v)

{

    ;

}

void dfs(int u, int par = -1)

{

    vis[u] = 1;

    tin[u] = low[u] = timer++;

    child[u]++;

    for (auto v : graph[u])

    {

        if (v == par)

            continue;

        if (vis[v])

        {

            low[u] = min(low[u], tin[v]);

        }

        else

        {

            dfs(v, u);

            low[u] = min(low[v], low[u]);

            child[u] += child[v];

            if (low[v] > tin[u])

                Bridge(u, v);

        }

    }

}

**Bigmod:**

int BigMod(int a, int p, int mod)

{

    int ans = 1;

    while (p)

    {

        if (p & 1)

            ans = (ans \* a) % mod;

        a = (a \* a) % mod;

        p >>= 1;

    }

    return ans;

}

**Trie:**

struct Node

{

    Node \*links[26];

    bool flag = false;

    bool containsKey(int ind)

    {

        return (links[ind] != NULL);

    }

    void put(int ind, Node \*node)

    {

        links[ind] = node;

    }

    Node \*get(int ind)

    {

        return links[ind];

    }

    void setEnd()

    {

        flag = true;

    }

    bool isEnd()

    {

        return flag;

    }

};

class Trie

{

private:

    Node \*root;

public:

    Trie()

    {

        root = new Node();

    }

    void insert(string word)

    {

        Node \*node = root;

        for (int i = 0; i < word.length(); i++)

        {

            if (!node->containsKey(word[i] - 'a'))

                node->put(word[i] - 'a', new Node());

            node = node->get(word[i] - 'a');

        }

        node->setEnd();

    }

    bool search(string word)

    {

        Node \*node = root;

        for (int i = 0; i < word.length(); i++)

        {

            if (!node->containsKey(word[i] - 'a'))

                return false;

            node = node->get(word[i] - 'a');

        }

        return node->isEnd();

    }

    bool startsWith(string prefix)

    {

        Node \*node = root;

        for (int i = 0; i < prefix.length(); i++)

        {

            if (!node->containsKey(prefix[i] - 'a'))

                return false;

            node = node->get(prefix[i] - 'a');

        }

        return true;

    }

};

**Fenwick Tree:**

/// @brief uses 1 based indexing.

vector<ll> tree(200005, 0);

ll lim = 200005;

ll p(ll k)

{

    return (k & (-k));

}

// adds x to ind.

void updateSum(ll ind, ll x)

{

    while (ind <= lim)

    {

        tree[ind] += x;

        ind += p(ind);

    }

}

ll getSum(ll k)

{ /// Returns Sum(1, k)

    ll s = 0;

    while (k > 0)

    {

        s += tree[k];

        k -= p(k);

    }

    return s;

}

**SQRT Decomposition:**

int blockSize;

struct query

{

    int id, l, r;

    bool operator<(query &b)

    {

        int blockNumber1 = l / blockSize;

        int blockNumber2 = l / blockSize;

        if (blockNumber1 != blockNumber2)

            return blockNumber1 < blockNumber2;

        return (blockNumber1 % 2 == 0) ? r < b.r : r > b.r;

    }

};

int32\_t main()

{

    ios::sync\_with\_stdio(0);

    cin.tie(0);

    cout.tie(0);

    int n;

    cin >> n;

    int arr[n];

    for (int i = 0; i < n; i++)

        cin >> arr[i];

    int q;

    cin >> q;

    vector<query> ranges(q);

    for (int i = 0; i < q; i++)

    {

        int l, r;

        cin >> l >> r;

        ranges[i].id = i;

        ranges[i].l = l - 1; // Make it 0-based

        ranges[i].r = r - 1; // Make it 0-based

    }

    blockSize = sqrt(n);

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

    auto add = [&](int ind)

    {

        ;

    };

    auto remove = [&](int ind)

    {

        ;

    };

    for (int i = 0; i < n; i++)

        add(i);

    int l = 0, r = n - 1;

    vector<int> ans(q);

    // Processing each query

    for (int i = 0; i < q; i++)

    {

        int curL = ranges[i].l, curR = ranges[i].r, index = ranges[i].id;

        while (l > curL)

            add(--l);

        while (l < curL)

            remove(l++);

        while (r < curR)

            add(++r);

        while (r > curR)

            remove(r--);

        // ans = ?

    }

    for (int i = 0; i < q; i++)

        cout << ans[i] << endl;

}

**Segment Tree:**

class SegmentTree

{

    int size = 1;

    vector<int> Sums, Lazy;                                // Stores the Sums

    void buildSum(vector<int> &a, int index, int l, int r) // Builds the tree

    {

        if (r - l == 1)

        {

            if (l < a.size())

                Sums[index] = a[l];

            return;

        }

        int mid = (l + r) / 2;

        buildSum(a, 2 \* index + 1, l, mid);

        buildSum(a, 2 \* index + 2, mid, r);

        Sums[index] = Sums[2 \* index + 1] + Sums[2 \* index + 2]; // Adds the sum of its chilren to the node

    }

    void updateRange(int lx, int rx, int l, int r, int index)

    {

        Sums[index] += (Lazy[index] \* (rx - lx));

        if (rx - lx > 1)

        {

            Lazy[2 \* index + 1] += Lazy[index];

            Lazy[2 \* index + 2] += Lazy[index];

        }

        Lazy[index] = 0;

        if (lx >= r or rx <= l)

            return;

        if (lx >= l and rx <= r)

        {

            Sums[index] += (rx - lx);

            if (rx - lx > 1)

            {

                Lazy[2 \* index + 1] += 1;

                Lazy[2 \* index + 2] += 1;

            }

            return;

        }

        int mid = (lx + rx) / 2;

        updateRange(lx, mid, l, r, 2 \* index + 1);

        updateRange(mid, rx, l, r, 2 \* index + 2);

        Sums[index] = Sums[2 \* index + 1] + Sums[2 \* index + 2];

    }

    int get(int l, int r, int index, int i)

    {

        Sums[index] += (Lazy[index] \* (r - l));

        if (r - l > 1)

        {

            Lazy[2 \* index + 1] += Lazy[index];

            Lazy[2 \* index + 2] += Lazy[index];

        }

        Lazy[index] = 0;

        if (r - l == 1)

        {

            return Sums[index];

        }

        int mid = (l + r) / 2;

        if (i < mid)

        {

            return get(l, mid, 2 \* index + 1, i);

        }

        else

        {

            return get(mid, r, 2 \* index + 2, i);

        }

    }

    int RangeSum(int lx, int rx, int l, int r, int index)

    {

        Sums[index] += (Lazy[index] \* (rx - lx));

        if (rx - lx > 1)

        {

            Lazy[2 \* index + 1] += Lazy[index];

            Lazy[2 \* index + 2] += Lazy[index];

        }

        Lazy[index] = 0;

        if (lx >= r or rx <= l)

            return 0;

        if (lx >= l and rx <= r)

        {

            return Sums[index];

        }

        int mid = (lx + rx) / 2;

        int a = RangeSum(lx, mid, l, r, 2 \* index + 1);

        int b = RangeSum(mid, rx, l, r, 2 \* index + 2);

        return a + b;

    }

public:

    SegmentTree(int n)

    {

        while (size < n)

            size \*= 2;

        Sums.assign(2 \* size, 0LL);

        Lazy.assign(2 \* size, 0LL);

    }

    void build(vector<int> &a)

    {

        buildSum(a, 0, 0, size);

    }

    void updateRange(int l, int r)

    {

        updateRange(0, size, l, r, 0);

    }

    int get(int i)

    {

        return get(0, size, 0, i);

    }

    int RangeSum(int l, int r)

    {

        return RangeSum(0, size, l, r, 0);

    }

    void show()

    {

        for (auto i : Sums)

        {

            cout << i << ' ';

        }

        cout << endl;

    }

};

**Sparse Table:**

#define bitlen 31

#define LOG 18

#define MAXLEN 200005

// 2^16 < 10^5

int arr[MAXLEN], pre[MAXLEN][LOG], n, q;

// 0-indexed

int query(int l, int r)

{

    int k = bitlen - \_\_builtin\_clz(r - l + 1);

    return min(pre[l][k], pre[r - (1 << k) + 1][k]);

}

void SparseTable()

{

    // Preprocessing

    for (int k = 1; k < LOG; k++)

    {

        for (i = 0; i + (1 << k) - 1 < n; i++)

        {

            pre[i][k] = min(pre[i][k - 1], pre[i + (1 << (k - 1))][k - 1]);

        }

    }

}

**LCP(hashing):**

const int p1 = 137;

const int mod1 = 127657753;

const int p2 = 277;

const int mod2 = 987654319;

const int N = 1e5 + 9;

pair<int, int> pw[N], ipw[N], pref[N];

int invp1, invp2;

string s;

int power(long long n, long long k, int mod)

{

    int ans = 1 % mod;

    n %= mod;

    if (n < 0)

        n += mod;

    while (k)

    {

        if (k & 1)

            ans = (long long)ans \* n % mod;

        n = (long long)n \* n % mod;

        k >>= 1;

    }

    return ans;

}

void prec()

{

    pw[0] = {1, 1};

    for (int i = 1; i < N; i++)

    {

        pw[i].first = 1LL \* pw[i - 1].first \* p1 % mod1;

        pw[i].second = 1LL \* pw[i - 1].second \* p2 % mod2;

    }

    invp1 = power(p1, mod1 - 2, mod1);

    invp2 = power(p2, mod2 - 2, mod2);

    ipw[0] = {1, 1};

    for (int i = 1; i < N; i++)

    {

        ipw[i].first = 1LL \* ipw[i - 1].first \* invp1 % mod1;

        ipw[i].second = 1LL \* ipw[i - 1].second \* invp2 % mod2;

    }

}

void build(string s)

{

    int n = s.size();

    for (int i = 0; i < n; i++)

    {

        pref[i].first = 1LL \* s[i] \* pw[i].first % mod1;

        if (i)

            (pref[i].first += pref[i - 1].first) %= mod1;

        pref[i].second = 1LL \* s[i] \* pw[i].second % mod2;

        if (i)

            (pref[i].second += pref[i - 1].second) %= mod2;

    }

}

pair<int, int> get\_hash(int i, int j)

{

    assert(i <= j);

    pair<int, int> hs({0, 0});

    hs.first = pref[j].first;

    if (i)

        hs.first = (hs.first - pref[i - 1].first + mod1) % mod1;

    hs.first = 1LL \* hs.first \* ipw[i].first % mod1;

    hs.second = pref[j].second;

    if (i)

        hs.second = (hs.second - pref[i - 1].second + mod2) % mod2;

    hs.second = 1LL \* hs.second \* ipw[i].second % mod2;

    return hs;

}

int lcp(int i, int j, int x, int y)

{

    int l = 1, r = min(j - i + 1, y - x + 1), ans = 0;

    while (l <= r)

    {

        int mid = l + r >> 1;

        if (get\_hash(i, i + mid - 1) == get\_hash(x, x + mid - 1))

            ans = mid, l = mid + 1;

        else

            r = mid - 1;

    }

    return ans;

}

int compare(int i, int j, int x, int y)

{

    /\* @brief

        0 -> equal

       -1 -> lesser

        1 -> greater

    \*/

    int l = lcp(i, j, x, y);

    if (j - i == y - x and l == j - i + 1)

        return 0; // equal

    else if (l == j - i + 1)

        return -1;

    else if (l == y - x + 1)

        return 1;

    // i + l or x + l may not exist so corner cases are handled separately.

    return (s[i + l] < s[x + l] ? -1 : 1);

}

**Convex Hull:**

struct Point

{

    int x, y;

};

int SQdist(Point p, Point q)

{

    return (p.x - q.x) \* (p.x - q.x) + (p.y - q.y) \* (p.y - q.y);

}

int orientation(Point p, Point q, Point r)

{ // orientation of p and r w.r.t. q

    int o = (p.x - q.x) \* (r.y - q.y) - (p.y - q.y) \* (r.x - q.x);

    if (o == 0)

        return 0;

    return (o < 0 ? -1 : 1); // -1 for clockwise

}

Point nextToTop(stack<Point> &s)

{

    Point tmp = s.top();

    s.pop();

    Point ret = s.top();

    s.push(tmp);

    return ret;

}

vector<Point> ConvexHull(vector<Point> &points)

{

    int n = points.size();

    // Find the minimum point

    int mnY = INT\_MAX, mnX = INT\_MAX, mnInd = -1;

    for (int i = 0; i < n; i++)

    {

        if (points[i].y < mnY)

        {

            mnY = points[i].y, mnX = points[i].x, mnInd = i;

        }

        else if (points[i].y == mnY and points[i].x < mnX)

        {

            mnX = points[i].x;

            mnInd = i;

        }

    }

    if (mnInd != 0)

        swap(points[0], points[mnInd]);

    Point P0 = points[0];

    // sort w.r.t polar angle with p0

    auto cmp = [&](Point &a, Point &b) -> bool

    {

        int o = orientation(P0, a, b);

        if (o == 0)

            return SQdist(P0, a) <= SQdist(P0, b);

        return o == -1;

    };

    sort(points.begin() + 1, points.end(), cmp);

    // Now remove the co-linear points by keeping just the farthest one

    vector<Point> pointsModified;

    pointsModified.push\_back(P0);

    for (int i = 1; i < n; i++)

    {

        while (i < n - 1 and orientation(P0, points[i], points[i + 1]) == 0)

            i++;

        pointsModified.push\_back(points[i]);

    }

    stack<Point> s;

    if (pointsModified.size() <= 3)

        return pointsModified;

    s.push(P0);

    s.push(pointsModified[1]);

    s.push(pointsModified[2]);

    for (int i = 3; i < pointsModified.size(); i++)

    {

        while (s.size() > 1 and orientation(nextToTop(s), s.top(), pointsModified[i]) == 1)

            s.pop();

        s.push(pointsModified[i]);

    }

    vector<Point> ret;

    while (!s.empty())

    {

        ret.push\_back(s.top());

        s.pop();

    }

    return ret;

}

int32\_t main()

{

    ios::sync\_with\_stdio(0);

    cin.tie(0);

    cout.tie(0);

    vector<Point> points = {{0, 3}, {1, 1}, {2, 2}, {4, 4}, {0, 0}, {1, 2}, {3, 1}, {3, 3}};

    vector<Point> Hull = ConvexHull(points);

    for (Point p : Hull)

        cout << p.x << ' ' << p.y << endl;

}

**Dinic’s algorithm:**

#define ll long long

const ll maxnodes = 10005;

ll nodes = maxnodes, src, dest;

ll dist[maxnodes], q[maxnodes], work[maxnodes];

struct Edge

{

ll to, rev;

ll f, cap;

};

vector<Edge> g[maxnodes];

void addEdge(ll s, ll t, ll cap)

{

Edge a = {t, g[t].size(), 0, cap};

Edge b = {s, g[s].size(), 0, 0};

g[s].push\_back(a);

g[t].push\_back(b);

}

bool dinic\_bfs()

{

fill(dist, dist + nodes, -1);

dist[src] = 0;

ll index = 0;

q[index++] = src;

for (ll i = 0; i < index; i++)

{

ll u = q[i];

for (ll j = 0; j < (ll) g[u].size(); j++)

{

Edge &e = g[u][j];

if (dist[e.to] < 0 && e.f < e.cap)

{

dist[e.to] = dist[u] + 1;

q[index++] = e.to;

}

}

}

return dist[dest] >= 0;

}

ll dinic\_dfs(ll u, ll f)

{

if (u == dest)

return f;

for (ll &i = work[u]; i < (ll) g[u].size(); i++)

{

Edge &e = g[u][i];

if (e.cap <= e.f) continue;

if (dist[e.to] == dist[u] + 1)

{

ll flow = dinic\_dfs(e.to, min(f, e.cap - e.f));

if (flow > 0)

{

e.f += flow;

g[e.to][e.rev].f -= flow;

return flow;

}

}

}

return 0;

}

ll maxFlow(ll \_src, ll \_dest)

{

src = \_src;

dest = \_dest;

ll result = 0;

while (dinic\_bfs())

{

fill(work, work + nodes, 0);

while (ll delta = dinic\_dfs(src, inf))

result += delta;

}

return result;

}

**Kuhn’s algorithm:**

// For Maximum Bipartite Matching

// Complexity: O(min(n\*m, n^3))

ll l\_siz, r\_siz; // l\_siz = left part size, r\_siz = right part size;

vector <ll> g[1500], lft, rgt;

vector <bool> used;

bool try\_kuhn(ll v)

{

for (ll &to : g[v]) {

if(used[to]) continue;

used[to] = 1;

if(rgt[to]==-1 || try\_kuhn(rgt[to])) {

lft[v] = to, rgt[to] = v;

return true;

}

}

return false;

}

ll kuhn()

{

ll max\_match = 0;

lft.assign(l\_siz+1, -1), rgt.assign(r\_siz+1, -1);

for(ll v=1; v<=l\_siz; ++v) {

used.assign(r\_siz+1, false);

max\_match += try\_kuhn(v);

}

return max\_match;

}

/\* Optimized Kuhn's Algorithm. Blog: https://codeforces.com/blog/entry/17023 \*/

ll kuhn2()

{

ll max\_match = 0;

lft.assign(l\_siz+1, -1), rgt.assign(r\_siz+1, -1);

// Shuffle the left part randomly to traverse them randomly

mt19937\_64 rng(chrono::steady\_clock::now().time\_since\_epoch().count()); // Random Seed

vector <ll> lft\_part;

for(ll v=1; v<=l\_siz; ++v) lft\_part.push\_back(v);

shuffle(lft\_part.begin(), lft\_part.end(), rng);

// Greedy matching with adjacent nodes at first

for(auto &v : lft\_part) {

// Shuffle the adjacent nodes to match them randomly

shuffle(g[v].begin(), g[v].end(), rng);

for(auto &to : g[v]) {

if(rgt[to] == -1) {

lft[v] = to, rgt[to] = v;

max\_match++;

break;

}

}

}

// Main Kuhn's Algorithm Part

bool new\_mat = 1;

while(new\_mat) {

// used is cleared one time in each iteration so that we can find several

// matchings in O(E). This makes the whole algorithm significantly faster.

used.assign(r\_siz+1, false);

// If no new match is found, the loop will break

new\_mat = 0;

for(auto &v : lft\_part) {

if(lft[v] != -1)

continue;

bool got = try\_kuhn(v);

max\_match += got, new\_mat |= got;

}

}

return max\_match;

}