Counting Sort

```
count[max size] ← frequencies array
For i = 0 to n − 1 do
    count[v [i]] + + (one more v[i] element)
i = 0
For j = min size to max size do
    While count[j] > 0 do
        v [i] = j (put element on array)
        count[j]−− (one less element of that size)
        i++ (increments first free position on the array)
```

A possible RadixSort (starting on the least significant digit)

```
bucket[10] ← array of lists of numbers (one per digit)
For pos = 1 to max number digits do
    For i = 0 to n − 1 do (for each number)
        Put v [i]in bucket[digit position pos(v [i])]
    For i = 0 to 9 do (for each possible digit)
        While size(bucket[i]) > 0 do
            Take first number of bucket[i] and add it to v []
```

Binary search on a sorted array

```
bsearch(v, low, high, key)
    While (low ≤ high ) do
        middle = low + (high − low )/2
        If (key == v [middle]) return(middle)
        Else If (key < v [middle]) high = middle − 1
        Else low = middle + 1
    return(-1)
```

Binary search for smallest k such that condition(k) is ”yes”

```
bsearch(low, high, condition)
While (low < high ) do
    middle = low + (high − low )/2
    If (condition(middle) == yes)) high = middle
    Else low = middle + 1
If (condition(low ) == no) return(-1)
return(low)
```

Ternary search

```
// a: left edge of interval, b: right edge of interval, f: function, epsilon: tolerance
input a, b, f, epsilon
l ← a
r ← b
while (r-l > epsilon)
    m1 ← (2*l + r)/3
    m2 ← (l + 2*r)/3
    if f(m1) < f(m2)
        l ← m1
    else
        r ← m2
print r
```

Segment tree

cpp

```cpp
const int MAX = 200005;   // Capacity of Segment Tree
const int MAX_ST = MAX*4;

const int NEUTRAL = 0;    // Neutral element

typedef int64_t st_value; // type of segment tree value

int n;                 // Number of elements in the segtree
st_value v[MAX];       // Array of values
st_value st[MAX_ST]; // Segtree (in this case storing interval sums)

// Merge contents of nodes a and b
st_value merge(st_value a, st_value b) {
    return a+b;
}

// Build initial segtree (in position pos, interval [start,end])
void build(int pos, int start, int end) {
    if (start == end) {
        st[pos] = v[start];
    } else {
        int middle = start + (end-start)/2;
        build(pos*2, start, middle);
        build(pos*2+1, middle+1, end);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Update node n to value v
void update(int pos, int start, int end, int n, st_value v) {
    if (start > n || end < n) return;
    if (start == end) {
        st[pos] = v;
    } else {
        int middle = start + (end-start)/2;
        update(pos*2, start, middle, n, v);
        update(pos*2+1, middle+1, end, n, v);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Make a query of interval [a,b]
st_value query(int pos, int start, int end, int a, int b) {
    if (start>b || end<a) return NEUTRAL;
    if (start>=a && end<=b) return st[pos];

    int middle = start + (end-start)/2;
    st_value l = query(pos*2, start, middle, a, b);
    st_value r = query(pos*2+1, middle+1, end, a, b);
    return merge(l, r);
}

// --------------------------------------------------------

int main() {
    int q;
```

```cpp
    cin >> n >> q;
    for (int i=1; i<=n; i++)
        cin >> v[i];

    build(1, 1, n);

    for (int i=1; i<=q; i++) {
        int op, a, b;
        cin >> op >> a >> b;
        if (op == 1) update(1, 1, n, a, b);
        else cout << query(1, 1, n, a, b) << endl;
    }
    return 0;
}
```

Binary Indexed Tree

```cpp
vector<int> tree;
int maxIdx;

int read(int idx) {
    int sum = 0;
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

void update(int idx, int val) {
    while (idx <= MaxIdx) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

int sum(int l, int r) { // Range Query
  return read(r) - read(l - 1);
}

int readSingle(int idx) { // Read the actual number, not cumulative
    int sum = tree[idx]; // this sum will be decreased
    if (idx > 0) { // the special case
        int z = idx - (idx & -idx);
        idx--; // idx is not important anymore, so instead y, you can use idx
        while (idx != z) { // at some iteration idx (y) will become z
        sum -= tree[idx];
        // substruct tree frequency which is between y and "the same path"
        idx -= (idx & -idx);
        }
    }
    return sum;
}

void scale(int c) { // Scale entire tree by c
    for (int i = 1; i <= MaxIdx; i++)
        tree[i] = tree[i] / c;
}
```

Kadane

```py
def max_subarray(numbers):
    """Find the largest sum of any contiguous subarray."""
    best_sum = float('-inf')
    current_sum = 0
    for x in numbers:
        current_sum = max(x, current_sum + x)
        best_sum = max(best_sum, current_sum)
    return best_sum
```

DFS

```
dfs(node v):
    mark v as visited
    For all neighbors w of v do
        If w has not yet been visited then
        dfs(w)
```

Finding Connected Components

```
counter ← 0
set all nodes as not visited
For all nodes v of the graph do
    If v has not yet been visited then
        counter++
        dfs(v)
write(counter)
```

Topological Sorting

```
order ← empty
set all nodes as not visited
For all nodes v of the graph do
    If v has not yet been visited then
        dfs(v)
write(order)

dfs(node v):
    mark v as visited
    For all neighbors w of v do
        If w has not yet been visited then
            dfs(w)
    add v to the beggining of order
```

Cycle Detection

```
color[v ∈ V] ← white
For all nodes v of the graph do
    If color[v] = white then
    dfs(v)

dfs(node v):
    color[v] ← gray
    For all neighbors w of v do
        If color[w] = gray then
            write(”Cycle found!”)
        Else if color[w] = white then
```

```
        dfs(w)
    color[v] ← black
```

Tarjan Algorithm for Strongly Connected Components

Make a DFS and in each node i: Keep pushing the nodes to a stack S. Compute and store the values of num(i) and low(i). If when finishing the visit of a node i we have that num(i) = low(i), then i is the "root" of a SCC. In that case, remove all the elements in the stack until reaching i and report those elements as belonging to a SCC!

```
index ← 0 ; S ← ∅
For all nodes v of the graph do
    If num[v] is still undefined then
        dfs_scc(v)

dfs scc(node v):
    num[v] ← low [v] ← index ; index ← index + 1 ; S.push(v)
    /* Traverse edges of v */
    For all neighbors w of v do
        If num[w] is still undefined then /* Tree Edge */
            dfs scc(w) ; low [v ] ← min(low [v], low [w])
        Else if w is in S then /* Back Edge */
            low [v] ← min(low [v], num[w])
    /* We know that we are at the root of an SCC */
    If num[v] = low [v] then
        Start new SCC C
        Repeat
            w ← S.pop() ; Add w to C
        Until w = v
        Write C
```

Articulation Points

Apply DFS to the graph and obtain the DFS tree. If a node v has a child w without any path to an ancestor of v, then v is an articulation point! (since removing it would disconnect w from the rest of the graph). The only exception is the root of the DFS tree. If it has more than one child in the tree it is also an articulation point!

```
dfs art(node v):
    num[v] ← low[v] ← index ; index ← index + 1 ; S.push(v)
    For all neighbors w of v do
        If num[w] is not yet defined then /* Tree Edge */
            dfs art(w) ; low [v] ← min(low [v], low [w])
            If low [w] ≥ num[v] then
                write(v + "is an articulation point")
        Else if w is in S then /* Back Edge */
        low [v] ← min(low [v], num[w])
    S.pop()
```

BFS - Computing Distances

```
bfs(node v):
    q ← ∅ /* Queue of non visited nodes */
    q.enqueue(v)
    v .distance ← 0 /* distance from v to itself it's zero */
    mark v as visited
    While q != ∅ /* while there are still unprocessed nodes */
        u ← q.dequeue() /* remove first element of q */
        For all neighbors w of u do
            If w has not yet been visited then /* new node */
                q.enqueue(w)
                mark w as visited
                w .distance ← u.distance + 1
```

Edmonds-Karp (Fluxo máximo)

cpp

```cpp
// Classe que representa um grafo
class Graph {
public:
    int n;                      // Numero de nos do grafo
    vector<vector <int>> adj; // Lista de adjacencias
    vector<vector <int>> cap; // Matriz de capacidades

    Graph(int n) {
        this->n = n;
        adj.resize(n+1); // +1 se os nos comecam em 1 ao inves de 0
        cap.resize(n+1);
        for (int i=1; i<=n; i++) cap[i].resize(n+1);
    }

    void addLink(int a, int b, int c) {
        // adjacencias do grafo nao dirigido, porque podemos ter de andar no sentido
        // contrario ao procurarmos caminhos de aumento
        adj[a].push_back(b);
        adj[b].push_back(a);
        cap[a][b] = c;
    }

    // BFS para encontrar caminho de aumento
    // devolve valor do fluxo nesse caminho
    int bfs(int s, int t, vector<int> &parent) {
        for (int i=1; i<=n; i++) parent[i] = -1;

        parent[s] = -2;
        queue<pair<int, int>> q; // fila do BFS com pares (no, capacidade)
        q.push({s, INT_MAX});    // inicializar com no origem e capacidade infinita

        while (!q.empty()) {
            // returar primeiro no da fila
            int cur = q.front().first;
            int flow = q.front().second;
            q.pop();

            // percorrer nos adjacentes ao no atual (cur)
            for (int next : adj[cur]) {
                // se o vizinho ainda nao foi visitado (parent==-1)
                // e a aresta respetiva ainda tem capacidade para passar fluxo
                if (parent[next] == -1 && cap[cur][next]>0) {
                    parent[next] = cur;                       // atualizar pai
                    int new_flow = min(flow, cap[cur][next]);  // atualizar fluxo
                    if (next == t) return new_flow;           // chegamos ao final?
                        q.push({next, new_flow});                 // adicionar a fila
                }
            }
        }

        return 0;
    }
```

```cpp

    // Algoritmo de Edmonds-Karp para fluxo maximo entre s e t
    // devolve valor do fluxo maximo (cap[][] fica com grafo residual)
    int maxFlow(int s, int t) {
        int flow = 0;             // fluxo a calcular
        vector<int> parent(n+1);  // vetor de pais (permite reconstruir caminho)

        while (true) {
            int new_flow = bfs(s, t, parent); // fluxo de um caminho de aumento
            if (new_flow == 0) break;         // se nao existir, terminar

            // imprimir fluxo e caminho de aumento
            cout << "Caminho de aumento: fluxo " << new_flow << " | " << t;

            flow += new_flow;  // aumentar fluxo total com fluxo deste caminho
            int cur = t;
            while (cur != s) { // percorrer caminho de aumento e alterar arestas
                int prev = parent[cur];
                cap[prev][cur] -= new_flow;
                cap[cur][prev] += new_flow;
                cur = prev;
                cout << " <- " << cur; // imprimir proximo no do caminho
            }
            cout << endl;
        }

        return flow;
    }
};

int main() {
    int n, e, a, b, c;

    cin >> n;
    Graph g(n);
    cin >> e;
    for (int i=0; i<e; i++) {
        cin >> a >> b >> c;
        g.addLink(a, b, c);
    }

    // Execucao exemplo usando 1 como no origem a 4 como o destino
    int flow = g.maxFlow(1, 4);
    cout << "Fluxo maximo: " << flow << endl;

    return 0;
}
```

Dijkstra

cpp

```cpp
// Classe que representa um no
class Node {
public:
    list<pair<int, int>> adj;  // Lista de adjacencias
    bool visited;              // No ja foi visitado?
    int distance;              // Distancia ao no origem da pesquisa
};

// Classe que representa um grafo
class Graph {
public:
    int n;          // Numero de nos do grafo
    Node *nodes;    // Array para conter os nos

    Graph(int n) { // Constructor: chamado quando um objeto Graph for criado
        this->n = n;
        nodes = new Node[n+1];  // +1 se os nos comecam em 1 ao inves de 0
    }

    ~Graph() { // Destructor: chamado quando um objeto Graph for destruido
        delete[] nodes;
    }

    void addLink(int a, int b, int c) {
        nodes[a].adj.push_back({b,c});
    }

    // Algoritmo de Dijkstra
    void dijkstra(int s) {
        //Inicializar nos como nao visitados e com distancia infinita
        for (int i=1; i<=n; i++) {
            nodes[i].distance = INT_MAX;
            nodes[i].visited  = false;
        }
        // Inicializar "fila" com no origem
        nodes[s].distance = 0;
        set<pair<int, int>> q; // By "default" um par e comparado pelo primeiro elemento
        q.insert({0, s});      // Criar um par (dist=0, no=s)
        // Ciclo principal do Dijkstra
        while (!q.empty()) {
            // Retirar no com menor distancia (o "primeiro" do set, que e uma BST)
            int u = q.begin()->second;
            q.erase(q.begin());
            nodes[u].visited = true;
            cout << u << " [dist=" << nodes[u].distance << "]" << endl;
            // Relaxar arestas do no retirado
            for (auto edge : nodes[u].adj) {
                int v = edge.first;
                int cost = edge.second;
                if (!nodes[v].visited && nodes[u].distance + cost < nodes[v].distance) {
                    q.erase({nodes[v].distance, v});  // Apagar do set
                    nodes[v].distance = nodes[u].distance + cost;
                    q.insert({nodes[v].distance, v}); // Inserir com nova (e menor) distancia
                }
            }
        }
    }
};

int main() {
    int n, e, a, b, c;
    cin >> n;
    Graph g(n);
    cin >> e;
    for (int i=0; i<e; i++) {
        cin >> a >> b >> c;
```

```cpp
        g.addLink(a, b, c);
    }
    // Execucao exemplo a partir do no 1
    g.dijkstra(1);
    return 0;
}
```

Minimum Spanning Trees

Prim

The minimum spanning tree is built gradually by adding edges one at a time. At first the spanning tree consists only of a single vertex (chosen arbitrarily). Then the minimum weight edge outgoing from this vertex is selected and added to the spanning tree. After that the spanning tree already consists of two vertices. Now select and add the edge with the minimum weight that has one end in an already selected vertex (i.e. a vertex that is already part of the spanning tree), and the other end in an unselected vertex. And so on, i.e. every time we select and add the edge with minimal weight that connects one selected vertex with one unselected vertex. The process is repeated until the spanning tree contains all vertices (or equivalently until we have $n - 1$ edges). In the end the constructed spanning tree will be minimal. If the graph was originally not connected, then there doesn't exist a spanning tree, so the number of selected edges will be less than $n - 1$.

Dense Graphs

```cpp
// We approach this problem from a different angle: for every not yet selected vertex we will store the minimum edge to an already selected vertex.
// Then during a step we only have to look at these minimum weight edges, which will have a complexity of O(n).
// After adding an edge some minimum edge pointers have to be recalculated. Note that the weights only can decrease, i.e. the minimal weight edge of every not yet selected vertex might stay the same, or it will be updated by an edge to the newly selected vertex. Therefore this phase can also be done in O(n).

int n;
vector<vector<int>> adj; // adjacency matrix of graph
const int INF = 1000000000; // weight INF means there is no edge

struct Edge {
    int w = INF, to = -1;
};

void prim() {
    int total_weight = 0;
    vector<bool> selected(n, false);
    vector<Edge> min_e(n);
    min_e[0].w = 0;

    for (int i=0; i<n; ++i) {
        int v = -1;
        for (int j = 0; j < n; ++j) {
            if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
                v = j;
        }

        if (min_e[v].w == INF) {
            cout << "No MST!" << endl;
            exit(0);
        }

        selected[v] = true;
        total_weight += min_e[v].w;
        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (int to = 0; to < n; ++to) {
            if (adj[v][to] < min_e[to].w)
                min_e[to] = {adj[v][to], v};
        }
    }

    cout << total_weight << endl;
}
```

Sparse Graphs

```cpp
// We can find the minimum edge in O(log n)$ time.
// On the other hand recomputing the pointers will now take O(n log n) time, which is worse than in the previous algorithm.
// But when we consider that we only need to update O(m) times in total, and perform O(n) searches for the minimal edge, then the total complexity will be O(m log n). For sparse graphs this is better than the above algorithm, but for dense graphs this will be slower.

const int INF = 1000000000;

struct Edge {
    int w = INF, to = -1;
    bool operator<(Edge const& other) const {
        return make_pair(w, to) < make_pair(other.w, other.to);
    }
};

int n;
vector<vector<Edge>> adj;

void prim() {
    int total_weight = 0;
    vector<Edge> min_e(n);
    min_e[0].w = 0;
    set<Edge> q;
    q.insert({0, 0});
    vector<bool> selected(n, false);
    for (int i = 0; i < n; ++i) {
        if (q.empty()) {
            cout << "No MST!" << endl;
            exit(0);
        }

        int v = q.begin()->to;
        selected[v] = true;
        total_weight += q.begin()->w;
        q.erase(q.begin());

        if (min_e[v].to != -1)
            cout << v << " " << min_e[v].to << endl;

        for (Edge e : adj[v]) {
            if (!selected[e.to] && e.w < min_e[e.to].w) {
```

```
                q.erase({min_e[e.to].w, e.to});
                min_e[e.to] = {e.w, v};
                q.insert({e.w, e.to});
            }
        }
    }

    cout << total_weight << endl;
}
```

Kruskal

Kruskal's algorithm initially places all the nodes of the original graph isolated from each other, to form a forest of single node trees, and then gradually merges these trees, combining at each iteration any two of all the trees with some edge of the original graph. Before the execution of the algorithm, all edges are sorted by weight (in non-decreasing order). Then begins the process of unification: pick all edges from the first to the last (in sorted order), and if the ends of the currently picked edge belong to different subtrees, these subtrees are combined, and the edge is added to the answer. After iterating through all the edges, all the vertices will belong to the same sub-tree, and we will get the answer.

```cpp
struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<int> tree_id(n);
vector<Edge> result;
for (int i = 0; i < n; i++)
    tree_id[i] = i;

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

for (Edge e : edges) {
    if (tree_id[e.u] != tree_id[e.v]) {
        cost += e.weight;
        result.push_back(e);

        int old_id = tree_id[e.u], new_id = tree_id[e.v];
        for (int i = 0; i < n; i++) {
            if (tree_id[i] == old_id)
                tree_id[i] = new_id;
        }
    }
}
```

Kruskal with Disjoint Set Union

Just as in the simple version of the Kruskal algorithm, we sort all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via calls to the make_set function - it will take a total of $O(N)$. We iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two find_set calls in $O(1)$ each). Finally, we need to perform the union of the two trees (sets), for which the DSU union_sets function will be called - also in $O(1)$. So we get the total time complexity of $O(M \log N + N + M) = O(M \log N)$.

```cpp
vector<int> parent, rank;

void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b])
            swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b])
            rank[a]++;
    }
}

struct Edge {
    int u, v, weight;
    bool operator<(Edge const& other) {
        return weight < other.weight;
    }
};

int n;
vector<Edge> edges;

int cost = 0;
vector<Edge> result;
parent.resize(n);
rank.resize(n);
for (int i = 0; i < n; i++)
    make_set(i);

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

for (Edge e : edges) {
    if (find_set(e.u) != find_set(e.v)) {
        cost += e.weight;
        result.push_back(e);
        union_sets(e.u, e.v);
    }
}
```

Bellman-Ford

Let us assume that the graph contains no negative weight cycle. The case of presence of a negative weight cycle will be discussed below in a separate section.

We will create an array of distances $d[0l \ldots n-1]$, which after execution of the algorithm will contain the answer to the problem. In the beginning we fill it as follows: $d[v] = 0$, and all other elements $d[]$ equal to ∞.

The algorithm consists of several phases. Each phase scans through all edges of the graph, and the algorithm tries to produce relaxation along each edge (a, b) having weight c. Relaxation along the edges is an attempt to improve the value $d[b]$ using value $d[a] + c$. In fact, it means that we are trying to improve the answer for this vertex using edge (a, b) and current answer for vertex a. It is claimed that $n - 1$ phases of the algorithm are sufficient to correctly calculate the lengths of all shortest paths in the graph (again, we believe that the cycles of negative weight do not exist). For unreachable vertices the distance $d[]$ will remain equal to ∞.

```cpp
struct Edge {
    int a, b, cost;
};

int n, m, v;
vector<Edge> edges;
const int INF = 1000000000;

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

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

        if (!any)
            break;
    }
    // display d, for example, on the screen
}
```

Retrieving Path

```cpp
void solve() {
    vector<int> d(n, INF);
    d[v] = 0;
    vector<int> p(n, -1);

    for (;;) {
        bool any = false;
        for (Edge e : edges)
            if (d[e.a] < INF)
                if (d[e.b] > d[e.a] + e.cost) {
                    d[e.b] = d[e.a] + e.cost;
                    p[e.b] = e.a;
                    any = true;
                }
        if (!any)
            break;
    }

    if (d[t] == INF)
        cout << "No path from " << v << " to " << t << ".";
    else {
        vector<int> path;
        for (int cur = t; cur != -1; cur = p[cur])
            path.push_back(cur);
        reverse(path.begin(), path.end());

        cout << "Path from " << v << " to " << t << ": ";
        for (int u : path)
            cout << u << ' ';
    }
}
```

Negative Cycle

```cpp
void solve() {
    vector<int> d(n, INF);
    d[v] = 0;
    vector<int> p(n, -1);
    int x;
    for (int i = 0; i < n; ++i) {
        x = -1;
        for (Edge e : edges)
            if (d[e.a] < INF)
                if (d[e.b] > d[e.a] + e.cost) {
                    d[e.b] = max(-INF, d[e.a] + e.cost);
                    p[e.b] = e.a;
                    x = e.b;
                }
    }

    if (x == -1)
        cout << "No negative cycle from " << v;
    else {
        int y = x;
        for (int i = 0; i < n; ++i)
            y = p[y];

        vector<int> path;
        for (int cur = y;; cur = p[cur]) {
            path.push_back(cur);
            if (cur == y && path.size() > 1)
                break;
        }
        reverse(path.begin(), path.end());

        cout << "Negative cycle: ";
        for (int u : path)
            cout << u << ' ';
    }
}
```

Floyd-Warshall

Let $d[][]$ is a 2D array of size $n \times n$, which is filled according to the 0-th phase as explained earlier. Also we will set $d[i][i] = 0$ for any i at the 0-th phase.

For $k = 0$, we can fill matrix with $d[i][j] = w_{\{ij\}}$ if there exists an edge between i and j with weight $w_{\{ij\}}$ and $d[i][j] = \infty$ if there doesn't exist an edge. In practice ∞ will be some high value.

cpp

```cpp
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
    }
}
```

KMP

Pseudocódigo dos slides:

```
KMP-Matcher(T, P)
n = T.length
m = P.length
pi = Compute-Prefix-Function(P)
q = 0
for i = 1 to n
    while q > 0 and P[q+1] != T[i]
        q = pi[q]
    if P[q+1] == T[i]
        q = q+1
    if q == m
        print "Pattern occurs with shift" i-m
        q = pi[q]

Compute-Prefix-Function(P)
m = P.length
let pi[1..m] be a new array
pi[1] = 0
k = 0
for q = 2 to m
    while k > 0 and P[k+1] != P[q]
        k = pi[k]
    if P[k+1] == P[q]
        k = k+1
    pi[q] = k
return pi
```

Prefix function definition

You are given a string s of length n. The prefix function for this string is defined as an array pi of length n, where $pi[i]$ is the length of the longest proper prefix of the substring $s[0...i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $pi[0] = 0$.

For example, prefix function of string "abcabcd" is $[0, 0, 0, 1, 2, 3, 0]$, and prefix function of string "aabaaab" is $[0, 1, 0, 1, 2, 2, 3]$.

cpp

```cpp
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

Trie

cpp

```cpp
const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode* getNode(void) {
    struct TrieNode* pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(struct TrieNode* root, string key) {
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else false
bool search(struct TrieNode* root, string key) {
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++) {
```

```
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// Returns true if root has no children, else false
bool isEmpty(TrieNode* root) {
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root->children[i])
            return false;
    return true;
}

// Recursive function to delete a key from given Trie
TrieNode* remove(TrieNode* root, string key, int depth = 0) {
    // If tree is empty
    if (!root)
        return NULL;
    // If last character of key is being processed
    if (depth == key.size()) {
        // This node is no more end of word after removal of given key
        if (root->isEndOfWord)
            root->isEndOfWord = false;
        // If given is not prefix of any other word
        if (isEmpty(root)) {
            delete (root);
            root = NULL;
        }
        return root;
    }

    // If not last character, recur for the child obtained using ASCII value
    int index = key[depth] - 'a';
    root->children[index] =
          remove(root->children[index], key, depth + 1);

    // If root does not have any child (its only child got
    // deleted), and it is not end of another word.
    if (isEmpty(root) && root->isEndOfWord == false) {
        delete (root);
        root = NULL;
    }

    return root;
}

// Driver
int main() {
    // Input keys (use only 'a' through 'z' and lower case)
    string keys[] = { "the", "a", "there",
                      "answer", "any", "by",
                      "bye", "their", "hero", "heroplane" };
    int n = sizeof(keys) / sizeof(keys[0]);

    struct TrieNode* root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, keys[i]);

    // Search for different keys
    search(root, "the") ? cout << "Yes\n" : cout << "No\n";
    search(root, "these") ? cout << "Yes\n" : cout << "No\n";

    remove(root, "heroplane");
    search(root, "hero") ? cout << "Yes\n" : cout << "No\n";
    return 0;
}
```

Aho-Corasick

Aho-Corasick Algorithm finds all words in O(n + m + z) time where z is total number of occurrences of words in text. The Aho-Corasick string matching algorithm formed the basis of the original Unix command fgrep.

Preprocessing: Build an automaton of all words in arr[] The automaton has mainly three functions:

- Go To: This function simply follows edges of Trie of all words in arr[]. It is represented as 2D array g[][] where we store next state for current state and character.

We build Trie. And for all characters which don't have an edge at root, we add an edge back to root.

- Failure: This function stores all edges that are followed when current character doesn't have edge in Trie. It is represented as 1D array f[] where we store next state for current state.

For a state s, we find the longest proper suffix which is a proper prefix of some pattern. This is done using Breadth First Traversal of Trie.

- Output: Stores indexes of all words that end at current state. It is represented as 1D array o[] where we store indexes of all matching words as a bitmap for current state.

For a state s, indexes of all words ending at s are stored. These indexes are stored as bitwise map (by doing bitwise OR of values). This is also computing using Breadth First Traversal with Failure.

Matching: Traverse the given text over built automaton to find all matching words.

cpp

```cpp
// Max number of states in the matching machine.
// Should be equal to the sum of the length of all keywords.
const int MAXS = 500;

// Maximum number of characters in input alphabet
const int MAXC = 26;

// OUTPUT FUNCTION IS IMPLEMENTED USING out[]
//Bit i in this mask is 1 if the word with idx i appears when the machine enters this state.
int out[MAXS];

// FAILURE FUNCTION IS IMPLEMENTED USING f[]
int f[MAXS];

// GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][]
int g[MAXS][MAXC];

// Builds the string matching machine.
// arr -   array of words. The index of each keyword is important:
```

```
//          "out[state] & (1 << i)" is > 0 if we just found word[i]
//          in the text.
// Returns the number of states that the built machine has.
// States are numbered 0 up to the return value - 1, inclusive.
int buildMatchingMachine(string arr[], int k) {
    // Initialize all values in output function as 0.
    memset(out, 0, sizeof out);

    // Initialize all values in goto function as -1.
    memset(g, -1, sizeof g);

    // Initially, we just have the 0 state
    int states = 1;

    // Construct values for goto function, i.e., fill g[][]
    // This is same as building a Trie for arr[]
    for (int i = 0; i < k; ++i) {
        const string &word = arr[i];
        int currentState = 0;

        // Insert all characters of current word in arr[]
        for (int j = 0; j < word.size(); ++j) {
            int ch = word[j] - 'a';
            // Allocate a new node (create a new state) if a
            // node for ch doesn't exist.
            if (g[currentState][ch] == -1)
                g[currentState][ch] = states++;

            currentState = g[currentState][ch];
        }

        // Add current word in output function
        out[currentState] |= (1 << i);
    }

    // For all characters which don't have an edge from
    // root (or state 0) in Trie, add a goto edge to state
    // 0 itself
    for (int ch = 0; ch < MAXC; ++ch)
        if (g[0][ch] == -1)
            g[0][ch] = 0;

    // Now, let's build the failure function

    // Initialize values in fail function
    memset(f, -1, sizeof f);

    // Failure function is computed in breadth first order
    // using a queue
    queue<int> q;

     // Iterate over every possible input
    for (int ch = 0; ch < MAXC; ++ch) {
        // All nodes of depth 1 have failure function value
        // as 0. For example, in above diagram we move to 0
        // from states 1 and 3.
        if (g[0][ch] != 0) {
            f[g[0][ch]] = 0;
            q.push(g[0][ch]);
        }
    }

    // Now queue has states 1 and 3
    while (q.size()) {
        // Remove the front state from queue
        int state = q.front();
        q.pop();
        // For the removed state, find failure function for
        // all those characters for which goto function is
        // not defined.
        for (int ch = 0; ch <= MAXC; ++ch) {
            // If goto function is defined for character 'ch'
            // and 'state'
            if (g[state][ch] != -1) {
                // Find failure state of removed state
                int failure = f[state];
                // Find the deepest node labeled by proper
                // suffix of string from root to current
                // state.
                while (g[failure][ch] == -1)
                      failure = f[failure];

                failure = g[failure][ch];
                f[g[state][ch]] = failure;
                // Merge output values
                out[g[state][ch]] |= out[failure];
                // Insert the next level node (of Trie) in Queue
                q.push(g[state][ch]);
            }
        }
    }
    return states;
}

// Returns the next state the machine will transition to using goto
// and failure functions.
// currentState - The current state of the machine. Must be between
//                0 and the number of states - 1, inclusive.
// nextInput - The next character that enters into the machine.
int findNextState(int currentState, char nextInput) {
    int answer = currentState;
    int ch = nextInput - 'a';
    // If goto is not defined, use failure function
    while (g[answer][ch] == -1)
        answer = f[answer];

    return g[answer][ch];
}

// This function finds all occurrences of all array words
// in text.
void searchWords(string arr[], int k, string text) {
    // Preprocess patterns.
    // Build machine with goto, failure and output functions
    buildMatchingMachine(arr, k);
    // Initialize current state
```

```
    int currentState = 0;
    // Traverse the text through the built machine to find
    // all occurrences of words in arr[]
    for (int i = 0; i < text.size(); ++i) {
        currentState = findNextState(currentState, text[i]);
        // If match not found, move to next state
        if (out[currentState] == 0)
             continue;
        // Match found, print all matching words of arr[]
        // using output function.
        for (int j = 0; j < k; ++j) {
            if (out[currentState] & (1 << j)) {
                cout << "Word " << arr[j] << " appears from "
                     << i - arr[j].size() + 1 << " to " << i << endl;
            }
        }
    }
}

// Driver program to test above
int main() {
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int k = sizeof(arr)/sizeof(arr[0]);
    searchWords(arr, k, text);
    return 0;
}
```

Midpoint of a line

cpp

```
// function to find the midpoint of a line
void midpoint(int x1, int x2, int y1, int y2) {
    cout << (float)(x1+x2)/2 << " , "<< (float)(y1+y2)/2 ;
}
// Driver Function to test above
int main() {
    int x1 =-1, y1 = 2  ;
    int x2 = 3, y2 = -6 ;
    midpoint(x1, x2, y1, y2);
    return 0;
}
```

Section formula (Point that divides a line in given ratio)

cpp

```
// Function to find the section of the line
void section(double x1, double x2, double y1, double y2, double m, double n) {
    // Applying section formula
    double x = ((n * x1) + (m * x2)) / (m + n);
    double y = ((n * y1) + (m * y2)) / (m + n);
    // Printing result
    cout << "(" << x << ", " << y << ")" << endl;
}
// Driver code
int main() {
    double x1 = 2, x2 = 4, y1 = 4, y2 = 6, m = 2, n = 3;
    section(x1, x2, y1, y2, m, n);
    return 0;
}
```

Slope of a line

cpp

```
// function to find the slope of a straight line
float slope(float x1, float y1, float x2, float y2) {
    if (x2 - x1 != 0)
        return (y2 - y1) / (x2 - x1);
    return INT_MAX;
}
// driver code to check the above function
int main() {
    float x1 = 4, y1 = 2;
    float x2 = 2, y2 = 5;
    cout << "Slope is: " << slope(x1, y1, x2, y2);
    return 0;
}
```

Line Intersection

The idea is to use orientation of lines to determine whether they intersect or not. Two line segments [p1, q1] and [p2, q2] intersect if and only if one of the following two conditions is verified:

1. General Case:
   - [p1, q1, p2] and [p1, q1, q2] have different orientations.
   - [p2, q2, p1] and [p2, q2, q1] have different orientations.
2. Special Case:
   - [p1, q1, p2], [p1, q1, q2], [p2, q2, p1], and [p2, q2, q1] are all collinear.
   - The x-projections of [p1, q1] and [p2, q2] intersect.
   - The y-projections of [p1, q1] and [p2, q2] intersect.

cpp

```
// function to check if point q lies on line segment 'pr'
bool onSegment(vector<int>& p, vector<int>& q, vector<int>& r) {
    return (q[0] <= max(p[0], r[0]) &&
            q[0] >= min(p[0], r[0]) &&
            q[1] <= max(p[1], r[1]) &&
            q[1] >= min(p[1], r[1]));
}
// function to find orientation of ordered triplet (p, q, r)
// 0 --> p, q and r are collinear
// 1 --> Clockwise
// 2 --> Counterclockwise
int orientation(vector<int>& p, vector<int>& q, vector<int>& r) {
    int val = (q[1] - p[1]) * (r[0] - q[0]) -
              (q[0] - p[0]) * (r[1] - q[1]);
    // collinear
    if (val == 0) return 0;
    // clock or counterclock wise
    // 1 for clockwise, 2 for counterclockwise
    return (val > 0) ? 1 : 2;
}
// function to check if two line segments intersect
bool doIntersect(vector<vector<vector<int>>>& points) {
    // find the four orientations needed
    // for general and special cases
    int o1 = orientation(points[0][0], points[0][1], points[1][0]);
    int o2 = orientation(points[0][0], points[0][1], points[1][1]);
    int o3 = orientation(points[1][0], points[1][1], points[0][0]);
    int o4 = orientation(points[1][0], points[1][1], points[0][1]);
    // general case
```

```cpp
    if (o1 != o2 && o3 != o4)
        return true;
    // special cases
    // p1, q1 and p2 are collinear and p2 lies on segment p1q1
    if (o1 == 0 &&
    onSegment(points[0][0], points[1][0], points[0][1])) return true;
    // p1, q1 and q2 are collinear and q2 lies on segment p1q1
    if (o2 == 0 &&
    onSegment(points[0][0], points[1][1], points[0][1])) return true;
    // p2, q2 and p1 are collinear and p1 lies on segment p2q2
    if (o3 == 0 &&
    onSegment(points[1][0], points[0][0], points[1][1])) return true;
    // p2, q2 and q1 are collinear and q1 lies on segment p2q2
    if (o4 == 0 &&
    onSegment(points[1][0], points[0][1], points[1][1])) return true;

    return false;
}

int main() {
    vector<vector<vector<int>>> points =
    {{{1, 1}, {10, 1}}, {{1, 2}, {10, 2}}};

    if(doIntersect(points))
        cout << "Yes";
    else cout << "No";

    return 0;
}
```

Point inside or outside a polygon

The idea to solve this problem is based on how to check if two given line segments intersect:

- Draw a horizontal line to the right of each point and extend it to infinity
- Count the number of times the line intersects with polygon edges.
- A point is inside the polygon if either count of intersections is odd or point lies on an edge of polygon. If none of the conditions is true, then point lies outside.

cpp

```cpp
struct Point {
    double x, y;
};
// Checking if a point is inside a polygon
bool point_in_polygon(Point point, vector<Point> polygon) {
    int num_vertices = polygon.size();
    double x = point.x, y = point.y;
    bool inside = false;
    // Store the first point in the polygon and initialize
    // the second point
    Point p1 = polygon[0], p2;
    // Loop through each edge in the polygon
    for (int i = 1; i <= num_vertices; i++) {
        // Get the next point in the polygon
        p2 = polygon[i % num_vertices];
        // Check if the point is above the minimum y
        // coordinate of the edge
        if (y > min(p1.y, p2.y)) {
            // Check if the point is below the maximum y
            // coordinate of the edge
            if (y <= max(p1.y, p2.y)) {
                // Check if the point is to the left of the
                // maximum x coordinate of the edge
                if (x <= max(p1.x, p2.x)) {
                    // Calculate the x-intersection of the
                    // line connecting the point to the edge
                    double x_intersection
                        = (y - p1.y) * (p2.x - p1.x) / (p2.y - p1.y) + p1.x;
                    // Check if the point is on the same
                    // line as the edge or to the left of
                    // the x-intersection
                    if (p1.x == p2.x || x <= x_intersection) {
                        // Flip the inside flag
                        inside = !inside;
                    }
                }
            }
        }
        // Store the current point as the first point for
        // the next iteration
        p1 = p2;
    }
    // Return the value of the inside flag
    return inside;
}
// Driver code
int main() {
    // Define a point to test
    Point point = { 150, 85 };
    // Define a polygon
    vector<Point> polygon = {
        { 186, 14 }, { 186, 44 }, { 175, 115 }, { 175, 85 }
    };
    // Check if the point is inside the polygon
    if (point_in_polygon(point, polygon)) {
        cout << "Point is inside the polygon" << endl;
    }
    else {
        cout << "Point is outside the polygon" << endl;
    }
    return 0;
}
```

Exemplos de código

Pesquisa binária

cpp

```cpp
// Criar um vector com as máquinas e inserir as máquinas, computando a máquina que demora
// mais tempo.
// Declarar o limite inferior (1 segundo) e superior (máquina que demora mais tempo a produzir
// todos os produtos).
// Fazer pesquisa binária nesse espaço de possibilidades: O(log n).
// A verificação da condição is_possible decide se é possível, num determinado tempo, fazer
// o número de produtos pretendido: O(n)

bool is_possible(vector<unsigned long long> machines, unsigned long long target_products, unsigned long long time) {
```

```cpp
    unsigned long long products_made = 0;
    for (unsigned long long m : machines) {
        products_made += time / m;
    }

    return products_made >= target_products;
}

unsigned long long my_binary_search(vector<unsigned long long> machines, unsigned long long lower, unsigned long long upper, unsigned long long target_products) {
    while (lower < upper) {
        unsigned long long middle = lower + (upper - lower) / 2;
        if (is_possible(machines, target_products, middle)) {
            upper = middle;
        } else {
            lower = middle + 1;
        }
    }

    if (!is_possible(machines, target_products, lower)) {
        return -1;
    }
    return lower;
}

int main() {
    unsigned long long n, t;
    cin >> n >> t;

    unsigned long long highest_value = 0;
    vector<unsigned long long> machines;

    for (unsigned long long i = 0; i < n; i++) {

        unsigned long long machine;
        cin >> machine;

        machines.push_back(machine);

        if (machine > highest_value) {
            highest_value = machine;
        }
    }

    unsigned long long lower_bound = 1;
    unsigned long long upper_bound = t * highest_value;

    unsigned long long minimum_time = my_binary_search(machines, lower_bound, upper_bound, t);

    cout << minimum_time << "\n";
    return 0;
}
```

cpp

```cpp
// Muito semelhante ao PC012, usar pesquisa binária no espaço de possibilidades.
//
// Duas diferenças: valores contínuos e o problema agora é de maximização (em vez de minimização).
// Para os valores contínuos, modificar o critério de paragem para ser um intervalo maior
// ou igual que a precisão pretendida.
// Para a maximização, o espaço de possibilidades é do tipo:
// [yes, yes,..., yes, no, no, ..., no]
// por isso, quando o middle verifica a condição, fazemos low = middle, e quando não verifica
// fazemos high = middle (o oposto da minimização do problema PC012).
//
// Para a condição booleana is_possible, usar novamente um algoritmo greedy que tenta
// dividir o volume de uma tarte pelo volume hipotético para obter o número de pessoas que
// pode alimentar.

bool is_possible(vector<int> pies, int people, double volume) {
    int max_people = 0;
    for (auto p : pies) {
        max_people += int(floor((M_PI * p * p)/volume));
    }

    return max_people >= people;
}


double my_binary_search(vector<int> pies, double low, double high, int people) {
    while (high-low >= 0.0001) {
        double middle = low + (high - low) / 2;

        if (is_possible(pies, people, middle)) {
            low = middle;
        } else {
            high = middle;
        }
    }
    return low;
}

int main() {

    int test_cases;
    cin >> test_cases;

    for (int i = 0; i < test_cases; i++) {
        int nr_pies, friends;
        cin >> nr_pies >> friends;

        int people = friends + 1;

        vector<int> pies;
        int biggest_pie = 0;
        for (int j = 0; j < nr_pies; j++) {
            int p;
            cin >> p;
            pies.push_back(p);
            if (p > biggest_pie) {
                biggest_pie = p;
            }
        }

        double upper_bound = M_PI * biggest_pie * biggest_pie;
        double lower_bound = 0;
```

```
        double volume = my_binary_search(pies, lower_bound, upper_bound, people);

        cout << fixed;
        cout << setprecision(4);
        cout << volume << "\n";
    }
    return 0;
}
```

Pesquisa Ternária

cpp

```
// Pesquisa ternária no espaço de possibilidades k (altura final dos edifícios).
//
// Função cost(k) calcula o custo de ter todos os edifícios à altura k: O(n).
// Fazer uma pesquisa ternária para o k mínimo: O(log3 n).
// A pesquisa ternária tem de ter um critério de paragem ligeiramente diferente porque,
// como o espaço de pesquisa é discreto, a divisão ternária não pode ser aplicada a espaços
// menores que 3.
// Assim, quando reduzimos o espaço de possibilidades a 3 inteiros, fazemos uma pesquisa
// linear nesses 3 inteiros.

typedef struct {
    long height;
    long cost;
} Building;

long cost(vector<Building> buildings, long k) {
    long cost = 0;
    for (auto x : buildings) {
        cost += abs(x.height - k) * x.cost;
    }
    return cost;
}

long my_ternary_search(vector<Building> buildings, long lower, long upper) {
    while (upper - lower > 3) {

        long m1 = lower + (upper - lower) / 3;
        long m2 = upper - (upper - lower) / 3;

        long cost1 = cost(buildings, m1);
        long cost2 = cost(buildings, m2);

        if (cost1 < cost2) {
            upper = m2;
        } else if (cost1 > cost2) {
            lower = m1;
        } else {
            lower = m1;
            upper = m2;
        }
    }

    long min_cost = cost(buildings, lower);
    long k = lower;

    for (long i = lower; i <= upper; i++) {
        long c = cost(buildings, i);
        if (c < min_cost) {
            min_cost = c;
            k = i;
        }
    }

    return k;
}

int main() {
    long nr_tests;
    cin >> nr_tests;

    for (long i = 0; i < nr_tests; i++) {
        long nr_buildings;
        cin >> nr_buildings;

        vector<Building> buildings;
        long highest_building = 0;

        for (long j = 0; j < nr_buildings; j++) {
            long height_in;
            cin >> height_in;

            Building b;
            b.height = height_in;
            b.cost = 0;
            buildings.push_back(b);

            if (height_in > highest_building) {
                highest_building = height_in;
            }
        }

        for (long j = 0; j < nr_buildings; j++) {
            long cost_in;
            cin >> cost_in;
            buildings[j].cost = cost_in;
        }

        long lower = 1;
        long upper = highest_building;

        long k = my_ternary_search(buildings, lower, upper);
        cout << cost(buildings, k) << "\n";
    }

    return 0;
}
```

Segment Tree

cpp

```
// Temos de modificar a segtree para guardar pares: o valor e a frequência desse valor.
// Quando construímos a árvore, as frequências são 1 para todos os números.
// No merge, escolhemos o maior dos números para passar para o nó pai.
// Mas se os números forem iguais, então somamos a frequência.

```

```cpp
const int MAX = 200005; // Capacity of Segment Tree
const int MAX_ST = MAX * 4;

const pair<int, int> NEUTRAL = {0, 0}; // Neutral element

typedef pair<int, int> st_value; // type of segment tree value

int n;                // Number of elements in the segtree
st_value v[MAX];      // Array of values
st_value st[MAX_ST]; // Segtree (in this case storing interval sums)

// Merge contents of nodes a and b
st_value merge(st_value a, st_value b) {
  if (a.first > b.first) {
    return a;
  }
  if (b.first > a.first) {
    return b;
  }
  return make_pair(a.first, a.second + b.second);
}

// Build initial segtree (in position pos, interval [start,end])
void build(int pos, int start, int end) {
  if (start == end) {
    st[pos] = v[start];
  } else {
    int middle = start + (end - start) / 2;
    build(pos * 2, start, middle);
    build(pos * 2 + 1, middle + 1, end);
    st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
  }
}

// Update node n to value v
void update(int pos, int start, int end, int n, st_value v) {
  if (start > n || end < n)
    return;
  if (start == end) {
    st[pos] = v;
  } else {
    int middle = start + (end - start) / 2;
    update(pos * 2, start, middle, n, v);
    update(pos * 2 + 1, middle + 1, end, n, v);
    st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
  }
}

// Make a query of interval [a,b]
st_value query(int pos, int start, int end, int a, int b) {
  if (start > b || end < a)
    return NEUTRAL;
  if (start >= a && end <= b)
    return st[pos];

  int middle = start + (end - start) / 2;
  st_value l = query(pos * 2, start, middle, a, b);
  st_value r = query(pos * 2 + 1, middle + 1, end, a, b);
  return merge(l, r);
}

int main() {
  int q;
  cin >> n >> q;
  for (int i = 1; i <= n; i++) {
    int w;
    cin >> w;
    v[i] = make_pair(w, 1);
  }

  build(1, 1, n);

  for (int i = 1; i <= q; i++) {
    int a, b;
    cin >> a >> b;
    pair<int, int> answer = query(1, 1, n, a, b);
    cout << answer.first << " " << answer.second << "\n";
  }

  return 0;
}
```
cpp

```cpp
// Guardar maps na segtree. Os pares do mapa representam o valor e a frequência desse valor.
// Em cada nó da tree, o map tem no máximo 3 valores: o mais frequente,
// o valor mais à esquerda, e o valor mais à direita (do intervalo correspondente).
//
// Assim, para fazer um merge, é preciso:
// verificar se os dois valores mais "interiores" são iguais. se forem, somar as frequências.
// Calcular o valor mais frequente.
// Colocar no novo map: o mais frequente, e os dois valores dos "extremos".
//
// Isto porque os "extremos" podem somar-se com os extremos de outros ramos da árvore.
//
// Como o elemento neutro é um map vazio, verificar maps vazios antes de aceder aos iteradores.

const int MAX = 200005; // Capacity of Segment Tree
const int MAX_ST = MAX * 4;

const map<int,int> NEUTRAL = {}; // Neutral element

typedef map<int,int> st_value; // type of segment tree value

int n;                // Number of elements in the segtree
st_value v[MAX];      // Array of values
st_value st[MAX_ST]; // Segtree (in this case storing interval sums)

// Merge contents of nodes a and b
st_value merge(st_value a, st_value b) {
    map<int,int> return_map;

    int most_frequent = 0;
    int frequency = 0;

    // calcular se o último elemento do map da direita é
```

```
    // igual ao primeiro elemento do map da esquerda
    if (!a.empty() && !b.empty()) {
        auto last_a = a.end();
        last_a--;
        auto first_b = b.begin();

        // se forem iguais, somar as frequencias e assumir como "mais frequente" para já
        if (last_a->first == first_b->first) {
            most_frequent = last_a->first;
            frequency = last_a->second + first_b->second;
        }
    }

    // verificar o valor mais frequente nos dois maps
    for (auto x : a) {
        if (x.second > frequency) {
            frequency = x.second;
            most_frequent = x.first;
        }
    }

    for (auto x : b) {
        if (x.second > frequency) {
            frequency = x.second;
            most_frequent = x.first;
        }
    }

    // preencher novo return_map com: o mais frequente,
    // o mais à esquerda do mapa esquerdo, e o mais à direita do mapa direito
    if (frequency > 0) {
        return_map.insert({most_frequent, frequency});
    }

    if (!a.empty()) {
        auto first_a = a.begin();
        return_map.insert(*first_a);
    }
    if (!b.empty()) {
        auto last_b = b.end();
        last_b--;
        return_map.insert(*last_b);
    }

    return return_map;
}

// Build initial segtree (in position pos, interval [start,end])
void build(int pos, int start, int end) {
  if (start == end) {
    st[pos] = v[start];
  } else {
    int middle = start + (end - start) / 2;
    build(pos * 2, start, middle);
    build(pos * 2 + 1, middle + 1, end);
    st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
  }
}

// Update node n to value v
void update(int pos, int start, int end, int n, st_value v) {
  if (start > n || end < n)
    return;

  if (start == end) {
    st[pos] = v;

  } else {
    int middle = start + (end - start) / 2;
    update(pos * 2, start, middle, n, v);
    update(pos * 2 + 1, middle + 1, end, n, v);
    st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
  }
}

// Make a query of interval [a,b]
st_value query(int pos, int start, int end, int a, int b) {
  if (start > b || end < a)
    return NEUTRAL;
  if (start >= a && end <= b)
    return st[pos];

  int middle = start + (end - start) / 2;
  st_value l = query(pos * 2, start, middle, a, b);
  st_value r = query(pos * 2 + 1, middle + 1, end, a, b);
  return merge(l, r);
}

int main() {
  int q;
  cin >> n >> q;
  for (int i = 1; i <= n; i++) {
    int w;
    cin >> w;

    map<int,int> m;
    m.insert({w,1});
    v[i] = m;
  }

  build(1, 1, n);

  for (int i = 1; i <= q; i++) {
    int a, b;
    cin >> a >> b;

    map<int,int> q_answer = query(1, 1, n, a, b);

    int frequency = 0;
    for (auto x : q_answer) {
        if (x.second > frequency) {
            frequency = x.second;
        }
    }

    cout << frequency << "\n";
```

```cpp
  }
  return 0;
}
```

```cpp
// Usar uma segtree para guardar o máximo.
//
// Fazer uma pesquisa binária na segtree para encontrar o valor mais à esquerda por defeito
// que for maior ou igual que o número de turistas que procura quarto.
//
// Fazer um update da segtree na posição encontrada na pesquisa anterior, com o valor da
// diferença entre o número de turistas e o número de quartos livres.

const int MAX = 200005;   // Capacity of Segment Tree
const int MAX_ST = MAX*4;

const int NEUTRAL = 0;    // Neutral element

typedef int64_t st_value; // type of segment tree value

int n;               // Number of elements in the segtree
st_value v[MAX];     // Array of values
st_value st[MAX_ST]; // Segtree (in this case storing interval sums)

// Merge contents of nodes a and b
st_value merge(st_value a, st_value b) {
    return max(a,b);
}

// Build initial segtree (in position pos, interval [start,end])
void build(int pos, int start, int end) {
    if (start == end) {
        st[pos] = v[start];
    } else {
        int middle = start + (end-start)/2;
        build(pos*2, start, middle);
        build(pos*2+1, middle+1, end);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Update node n to value v
void update(int pos, int start, int end, int n, st_value v) {
    if (start > n || end < n) return;
    if (start == end) {
        st[pos] = v;
    } else {
        int middle = start + (end-start)/2;
        update(pos*2, start, middle, n, v);
        update(pos*2+1, middle+1, end, n, v);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Make a query of interval [a,b]
st_value query(int pos, int start, int end, int a, int b) {
    if (start>b || end<a) return NEUTRAL;
    if (start>=a && end<=b) return st[pos];

    int middle = start + (end-start)/2;
    st_value l = query(pos*2, start, middle, a, b);
    st_value r = query(pos*2+1, middle+1, end, a, b);
    return merge(l, r);
}

// ---------------------------------------------------------

int main() {
    int q;
    cin >> n >> q;
    for (int i=1; i<=n; i++) {
        int h;
        cin >> h;
        v[i] = h;
    }

    build(1, 1, n);

    for (int i = 0; i < q; i++) {
        int tourists;
        cin >> tourists;

        if (st[1] < tourists) {
            cout << 0;

        } else {
            // binary search na segment tree
            int cur = 1;
            int start = 1;
            int end = n;
            while (start != end) {
                int middle = start + (end-start)/2;

                if (st[2 * cur] >= tourists) {
                    cur = 2 * cur;
                    end = middle;

                } else {
                    cur = 2 * cur + 1;
                    start = middle + 1;
                }
            }
            cout << start;
            update(1, 1, n, start, st[cur] - tourists);
        }

        if (i < q - 1) {
            cout << " ";
        }
    }
    cout << "\n";
    return 0;
}
```

```cpp
// Receber a sequência de números e guardar para usar mais tarde:
// sequence = [0,5,1,2,3,4,5]
//
```

```cpp
// Receber primeiro as queries, para poder criar uma única segtree que responde a todos os queries:
// queries = [(2,4,1),(4,4,4),(1,6,2)]
//
// Pôr todos os K das queries num set, para remover repetidos e ordenar:
// query_set = [1,2,4]
//
// Criar um dicionário auxiliar para ser mais fácil aceder a indexes:
// query_dictionary = [{1:0},{2:1},{4:2}]
//
// Criar o array auxiliar v[] para usar o build() da segtree. Cada elemento da segtree é um
// vector que nos diz, a cada posição, se os elementos desse range são maiores do que um K, para
// K valores do vector.
// Por exemplo, o primeiro elemento da sequência (5) é maior que 1, 2, e 4, logo:
// v[1] = [1,1,1]
// O quarto elemento da sequência (3) é só maior que 1 e 2, logo:
// v[4] = [1,1,0]
//
// O merge() da segtree é simplesmente somar os vectores elemento a elemento, tendo em atenção
// vectores vazios (NEUTRAL são vectores vazios).
//
// A resposta a um query da árvore é, portanto, um vector, que representa o número de vezes que,
// nesse range, os elementos da sequência são maiores que os K todos.
// Depois é só usar o query_dictionary para descobrir o índice desse vector que corresponde ao
// query k que queremos responder.

const int MAX = 200005;   // Capacity of Segment Tree
const int MAX_ST = MAX*4;

const vector<int> NEUTRAL = {};    // Neutral element

typedef vector<int> st_value; // type of segment tree value

int n;               // Number of elements in the segtree
st_value v[MAX];     // Array of values
st_value st[MAX_ST]; // Segtree (in this case storing interval sums)

// Merge contents of nodes a and b
st_value merge(st_value a, st_value b) {
    // verificar vectores vazios, usar tamanho maior
    int size_a = a.size();
    int size_b = b.size();
    int size = max(size_a, size_b);

    vector<int> merge_result;

    for (int i = 0; i < size; i++) {
        if (a.empty()){
            merge_result.push_back(b[i]);
        } else if (b.empty()) {
            merge_result.push_back(a[i]);
        } else {
            merge_result.push_back(a[i]+b[i]);
        }
    }

    return merge_result;
}

// Build initial segtree (in position pos, interval [start,end])
void build(int pos, int start, int end) {
    if (start == end) {
        st[pos] = v[start];
    } else {
        int middle = start + (end-start)/2;
        build(pos*2, start, middle);
        build(pos*2+1, middle+1, end);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Update node n to value v
void update(int pos, int start, int end, int n, st_value v) {
    if (start > n || end < n) return;
    if (start == end) {
        st[pos] = v;
    } else {
        int middle = start + (end-start)/2;
        update(pos*2, start, middle, n, v);
        update(pos*2+1, middle+1, end, n, v);
        st[pos] = merge(st[pos*2], st[pos*2+1]);
    }
}

// Make a query of interval [a,b]
st_value query(int pos, int start, int end, int a, int b) {
    if (start>b || end<a) return NEUTRAL;
    if (start>=a && end<=b) return st[pos];

    int middle = start + (end-start)/2;
    st_value l = query(pos*2, start, middle, a, b);
    st_value r = query(pos*2+1, middle+1, end, a, b);
    return merge(l, r);
}

// ---------------------------------------------------------

int main() {
    int q;
    cin >> n >> q;
    vector<int> sequence;
    sequence.push_back(0); // sentinela para sequence[0]

    // construir sequencia de números
    for (int i = 0; i < n; i++) {
        int number;
        cin >> number;
        sequence.push_back(number);
    }

    vector<tuple<int,int,int>> queries;
    set<int> query_set;
```

```

    // construir vector de queries (para responder depois)
    // construir query_set (para saber que números tenho de responder na segtree)
    for (int i = 0; i < q; i++) {
        tuple<int,int,int> query;
        int a, b, k;
        cin >> a >> b >> k;

        get<0>(query) = a;
        get<1>(query) = b;
        get<2>(query) = k;

        queries.push_back(query);
        query_set.insert(k);
    }

    // construir um dicionário para ser mais fácil saber o índice dos queries
    map<int,int> query_dictionary;
    int counter = 0;
    for (auto x : query_set) {
        query_dictionary.insert({x, counter});
        counter++;
    }

    // construir o array v[] para o build() da segtree
    for (int i = 1; i <= n; i++) {
        vector<int> v_k;
        for (auto x : query_set) {
            if (sequence[i] > x) {
                v_k.push_back(1);
            } else {
                v_k.push_back(0);
            }
        }
        v[i] = v_k;
    }

    build(1, 1, n);

    for (auto x : queries) {
        int a, b, k;
        tie(a, b, k) = x;

        vector<int> query_result = query(1, 1, n, a, b);
        int index = query_dictionary[k];
        int result = query_result[index];

        cout << result << "\n";
    }
    return 0;
}
```

Cumulative Sums

cpp

```cpp
// Construir a matriz de somas cumulativas.
// Fixar duas linhas da matriz, e usar Kadane como se as colunas delimitadas por essas
// linhas fossem elementos únicos (a soma da coluna).

// soma de uma coluna j, desde i_start até i_end
int column(vector<vector<int>> & sums, int i_start, int i_end, int j) {
    return sums[i_end][j] - sums[i_start-1][j] - sums[i_end][j-1] + sums[i_start-1][j-1];
}

// kadane numa dimensão 'n', delimitada por duas linhas 'a' e 'b'
int kadane(vector<vector<int>> & sums, int size, int a, int b) {
    int current_sum = column(sums, a, b, 1);
    int best_sum = current_sum;

    for (int n = 2; n <= size; n++) {
        current_sum = max(column(sums, a, b, n), current_sum + column(sums, a, b, n));
        best_sum = max(best_sum, current_sum);
    }

    return best_sum;
}


int main() {
    int n;
    cin >> n;

    // construir matriz e matriz de somas acumuladas
    vector<vector<int>> cumSums(n + 1, vector<int>(n + 1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            int num;
            cin >> num;
            cumSums[i][j] = num + cumSums[i][j-1];
        }
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            cumSums[j][i] = cumSums[j][i] + cumSums[j-1][i];
        }
    }

    int max_sum = -101;

    // fixar linhas (a e b) e usar Kadane nessa dimensão
    for (int a = 1; a <= n; a++) {
        for (int b = a; b <= n; b++) {
            int sum_at = kadane(cumSums, n, a, b);
            if (sum_at > max_sum) {
                max_sum = sum_at;
            }
        }
    }

    cout << max_sum << "\n";
    return 0;
}
```

Binary Indexed Tree

cpp

```cpp
// Guardar as arestas num dicionário, assim ficam ordenadas por 'n'.
//
// Uma aresta NiMi vai ter intersecções se as arestas anteriores forem do tipo NkMj, com k < i e j > i.
// k < i é garantindo porque estamos a percorrer as arestas ordenadas por 'n'.
// Só temos de verificar quantos valores j > i apareceram até agora.
//
// Ou seja, quantas arestas que começam em nós anteriores de N acabam em nós posteriores de M.
// Só precisamos de guardar as frequências dos 'm' que apareceram até agora (numa BIT) e,
// para cada aresta NM, perguntar quantos 'm' maiores que M apareceram.
//
// Ou seja, fazer um range query a uma BIT tree do tipo sum[l,r] (que é sum[0,r] - sum[0,l-1])
//
// Para os updates, temos de guardar os valores de 'm' num stack temporário e só fazer updates
// se o 'n' seguinte for diferente. Isto porque, por exemplo, as arestas 3,1 e 3,2 não se intersectam.

#define ll long long int

// Implementação da BIT

vector<ll> tree;
ll maxIdx;

ll read(ll idx) {
    ll sum = 0;
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

ll sum(ll l, ll r) {
    return read(r) - read(l - 1);
}

void update(ll idx, ll val) {
    while (idx <= maxIdx) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

int main() {
    ll n, m, k;
    cin >> n >> m >> k;

    maxIdx = m;
    multimap<ll,ll> roads;
    tree.assign(m+1, 0);

    for (ll i = 0; i < k; i++) {
        ll n1, m1;
        cin >> n1 >> m1;
        roads.insert({n1, m1});
    }

    stack<ll> temp_update;
    ll prev = 0;
    ll crossing_sum = 0;

    for (auto x : roads) {
        if (x.first != prev) {
            while (!temp_update.empty()) {
                update(temp_update.top(), 1);
                temp_update.pop();
            }
        }

        ll temp_sum = sum(x.second + 1, maxIdx);
        crossing_sum += temp_sum;

        temp_update.push(x.second);
        prev = x.first;

    }
    cout << crossing_sum << "\n";
    return 0;
}
```

Longest Common Subsequence

cpp

```cpp
#define INF 1e6 + 4

vector<int> alice_seq;
vector<int> bob_seq;
vector<vector<int>> dp;

void receive_seq(vector<int> &seq, int sz) {
    seq.push_back(-INF);
    for (int i = 1; i <= sz; i++) {
        int num;
        cin >> num;
        seq.push_back(num);
    }
}

// longest common subsequence, recursivo com memoization
// i: índice na sequência da alice
// j: índice na sequência do bob
int lcs_rec(int i, int j) {
    // caso base
    // se uma ou ambas as sequências estiverem vazias.
    // não pode ser zero porque podemos ter respostas negativas.
    if (i == 0 || j == 0) {
        return -INF;
    }
    // memoization
    // verificar na tabela dp se a resposta já foi calculada
    if (dp[i][j] != -INF) {
        return dp[i][j];
    }
```

```cpp
    // 3 hipóteses:
    // 1) usar os números dos índices indicados: o nosso valor total vai incrementar da multiplicação
    // (ou vai ser melhor que um valor muito negativo), continuar a procurar em i-1, j-1
    // 2) descartar o número 'j' do bob, continuar a procurar em i, j-1
    // 3) descartar o número 'i' da alice, continuar a procurar em i-1, j
    int use_both = max(alice_seq[i] * bob_seq[j] + lcs_rec(i-1, j-1), alice_seq[i] * bob_seq[j]);
    int use_alice = lcs_rec(i, j-1);
    int use_bob = lcs_rec(i-1, j);
    // retornar e guardar o valor em dp ao mesmo tempo
    return dp[i][j] = max(use_both, max(use_alice, use_bob));
}

int main() {
    int n;
    cin >> n;
    receive_seq(alice_seq, n);

    int m;
    cin >> m;
    receive_seq(bob_seq, m);
    // tabela de memoization, usar 'inf' por causa dos valores negativos
    dp.assign(n+1, vector<int> (m+1, -INF));

    cout << lcs_rec(n, m) << "\n";
    return 0;
}
```

Bellman-Ford

cpp

```cpp
// Estrutura do Grafo do Professor Pedro Ribeiro
//
// Bellman-Ford aplicado quase directamente, trocar só a inicialização dos nós para -inf
// porque é um problema de maximização.
//
// Na verificação de ciclos, só retornar -1 se o relaxamento existir (obviamente) mas também só se:
// 1) o nó relaxado consegue chegar ao nó final E
// 2) o nó inicial consegue chegar ao nó relaxado
// estas condições são verificadas com um DFS reaches_n(v, z) que verifica se um nó v consegue
// chegar ao nó z.

#define INF LONG_LONG_MAX
typedef long long ll;

// Classe que representa um no
class Node {
public:
    list<pair<int, int>> adj;  // Lista de adjacencias
    ll distance;              // Distancia ao no origem da pesquisa
    int parent;
    int visited;
};

// Classe que representa um grafo
class Graph {
public:
    int n;          // Numero de nos do grafo
    Node *nodes;    // Array para conter os nos

    Graph(int n) { // Constructor: chamado quando um objeto Graph for criado
        this->n = n;
        nodes = new Node[n+1];  // +1 se os nos comecam em 1 ao inves de 0
    }

    ~Graph() { // Destructor: chamado quando um objeto Graph for destruido
        delete[] nodes;
    }

    void addLink(int a, int b, int c) {
        nodes[a].adj.push_back({b,c});
    }

    bool dfs(int v, int z) {
        if (v == z) {
            return true;
        }
        nodes[v].visited = true;
        for (auto e : nodes[v].adj) {
            if (!nodes[e.first].visited && dfs(e.first, z)) {
                return true;
            }
        }
        return false;
    }

    bool reaches_n(int v, int z) {
        // cout << "testar se " << v << " chega a " << z << "\n";
        for (int i = 1; i <= n; i++) {
            nodes[i].visited = false;
        }
        bool reaches = dfs(v, z);
        // cout << reaches << "\n";
        return reaches;
    }

    // Bellman-Ford
    ll bellman_ford() {

        // inicializar nós a -inf, estamos à procura da pontuação máxima
        for (int i = 1; i <= n; i++) {
            nodes[i].distance = -INF;
        }
        // pontuação inicial
        nodes[1].distance = 0;
        nodes[1].parent = 0;

        // percorrer todas as arestas V-1 vezes
        for (int i = 1; i < n; i++) {
            for (int v = 1; v <= n; v++) {
                for (auto e : nodes[v].adj) {
                    int neighbour = e.first;
                    int score = e.second;
                    if (nodes[v].distance + score > nodes[neighbour].distance) {
```

```cpp
                        nodes[neighbour].distance = nodes[v].distance + score;
                        nodes[neighbour].parent = v;
                    }
                }
            }
        }

        // detetar ciclos - fazer uma última iteração de todas as arestas
        for (int v = 1; v <= n; v++) {
            for (auto e : nodes[v].adj) {
                int neighbour = e.first;
                int score = e.second;
                // retornar -1 só se o nó estiver envolvido no caminho de maior pontuação
                if (reaches_n(neighbour, n) && reaches_n(1, v) && nodes[v].distance + score > nodes[neighbour].distance) {
                    return -1;
                }
            }
        }
        return nodes[n].distance;
    }
};

int main() {
    int n, e, a, b, c;

    cin >> n;
    Graph g(n);
    cin >> e;
    for (int i=0; i<e; i++) {
        cin >> a >> b >> c;
        g.addLink(a, b, c);
    }

    cout << g.bellman_ford() << "\n";

    return 0;
}
```

Grafo Bipartido

```cpp
// Usei a implementação de Edmonds-Karp do Professor Pedro Ribeiro de DAA.
//
// Só modifiquei a inserção de dados no grafo, para transformar os inputs desta maneira:
// O número de nós do grafo é 2n+2. 2n porque é um grafo bipartido.
// Por exemplo, para o caso de teste:
// 5
// 5
// 0 1
// 1 2
// 2 3
// 3 4
// 4 2
//
// 5 nós e 5 arestas tranformam-se em 10 nós e 5 arestas:
//
//            1     6
//            2     7
//            3     8
//            4     9
//            5     10
//
// com arestas 1-7, 2-8, 3-9, 4-10, 5-8
//
// Para transformar este grafo num problema de fluxo máximo (com capacidades unitárias)
// temos de adicionar dois nós guarda:
//
//            1     6
//            2     7
//      0     3     8     11
//            4     9
//            5     10
//
// e arestas do nó 0 para os nós 1..5 e dos nós 6..10 para o nó 11.
//
// Depois é só resolver um problema de fluxo máximo, onde as capacidades das arestas são 1,
// e se o fluxo final for igual a 'n' então conseguimos distribuir todos os livros por todas
// as pessoas.

// Classe que representa um grafo
class Graph {
public:
    int n;                    // Numero de nos do grafo
    vector<vector <int>> adj; // Lista de adjacencias
    vector<vector <int>> cap; // Matriz de capacidades

    Graph(int n) {
        this->n = n;
        adj.resize(n); // +1 se os nos comecam em 1 ao inves de 0
        cap.resize(n);
        for (int i=0; i<n; i++) cap[i].resize(n);
    }

    void addLink(int a, int b, int c) {
        // adjacencias do grafo nao dirigido, porque podemos ter de andar no sentido
        // contrario ao procurarmos caminhos de aumento
        adj[a].push_back(b);
        adj[b].push_back(a);
        cap[a][b] = c;
    }

    // BFS para encontrar caminho de aumento
    // devolve valor do fluxo nesse caminho
    int bfs(int s, int t, vector<int> &parent) {
        for (int i=0; i<n; i++) parent[i] = -1;

        parent[s] = -2;
        queue<pair<int, int>> q; // fila do BFS com pares (no, capacidade)
        q.push({s, INT_MAX});    // inicializar com no origem e capacidade infinita

        while (!q.empty()) {
            // returar primeiro no da fila
            int cur = q.front().first;
            int flow = q.front().second;
            q.pop();

```

```
        // percorrer nos adjacentes ao no atual (cur)
        for (int next : adj[cur]) {
            // se o vizinho ainda nao foi visitado (parent==-1)
            // e a aresta respetiva ainda tem capacidade para passar fluxo
            if (parent[next] == -1 && cap[cur][next]>0) {
            parent[next] = cur;                    // atualizar pai
            int new_flow = min(flow, cap[cur][next]);  // atualizar fluxo
            if (next == t) return new_flow;            // chegamos ao final?
            q.push({next, new_flow});                  // adicionar a fila
            }
        }
    }
    return 0;
}

// Algoritmo de Edmonds-Karp para fluxo maximo entre s e t
// devolve valor do fluxo maximo (cap[][] fica com grafo residual)
int maxFlow(int s, int t) {
    int flow = 0;            // fluxo a calcular
    vector<int> parent(n+1);  // vetor de pais (permite reconstruir caminho)

    while (true) {
        int new_flow = bfs(s, t, parent); // fluxo de um caminho de aumento
        if (new_flow == 0) break;         // se nao existir, terminar

        // imprimir fluxo e caminho de aumento
        // cout << "Caminho de aumento: fluxo " << new_flow << " | " << t;

        flow += new_flow;  // aumentar fluxo total com fluxo deste caminho
        int cur = t;
        while (cur != s) { // percorrer caminho de aumento e alterar arestas
            int prev = parent[cur];
            cap[prev][cur] -= new_flow;
            cap[cur][prev] += new_flow;
            cur = prev;
            // cout << " <- " << cur; // imprimir proximo no do caminho
        }
        // cout << endl;
    }
    return flow;
}
};

int main() {
    int tests;
    cin >> tests;

    for (int t = 0; t < tests; t++) {
        int n, e, a, b;
        cin >> n;
        // grafo bipartido com nós guarda
        Graph g(2*n+2);
        cin >> e;
        for (int i=0; i<e; i++) {
            cin >> a >> b;
            // grafo bipartido com capacidades unitárias
            g.addLink(a+1, b+1+n, 1);
        }
        // adicionar arestas dos nós guardas (0 e 2n+1)
        for (int i = 1; i <= n; i++) {
            g.addLink(0, i, 1);
            g.addLink(i+n, 2*n+1, 1);
        }
        // resolver fluxo máximo no grafo bipartido
        int flow = g.maxFlow(0, 2*n+1);
        // se o fluxo for igual a 'n', conseguimos distribuir todos os livros
        if (flow != n) {
            cout << "NO\n";
        }
        else {
            cout << "YES\n";
        }
    }
    return 0;
}
```

KMP

cpp

```cpp
// Aplicar KMP
//
// Depois de receber a string original, calcular a sua função Pi.
// Sabendo o valor do prefixo/sufixo maior (é o valor na última posição de Pi),
// criar uma string possível de ser a nossa resposta.
// Já sabemos que obedece às condições de sufixo e prefixo (é isso que a função Pi faz).
// Para descobrir se é infixo, correr KMP da string possível na string original.
//
// KMP foi aplicado quase directamente dos slides, com as modificações:
// 1) Usar indexação 0,
// 2) Em vez de imprimir que "foi encontrado padrão com shift etc.", queremos apenas
// retornar verdadeiro se o padrão acontecer (obviamente) mas também se acontecer para além
// de acontecer no início e no fim da string original.
//
// Se não ocorrer, temos de verificar se a string possível contém ela própria sufixos/prefixos.
// Por exemplo, a string qwertyqwertyqwerty tem como possível resposta qwertyqwerty. Mas qwertyqwerty não
// cumpre as nossas 3 condições (prefixo, sufixo, infixo).
// Mas qwerty cumpre. Por isso temos um ciclo que transforma a nossa string possível em strings cada vez
// mais pequenas que têm prefixos que são sufixos.

void print_pi(string p, vector<int> pi) {
    for (auto c : p) {
        cout << " " << c;
    }
    cout << "\n";
    for (auto i : pi) {
        cout << " " << i;
    }
    cout << "\n";
}

vector<int> compute_prefix(string p) {
    int m = p.length();
    vector<int> pi(m,0);
    int k = 0;
```

```

    for (int q = 1; q < m; q++) {
        while (k > 0 && p[k] != p[q]) {
            k = pi[k-1];
        }
        if (p[k] == p[q]) {
            k++;
        }
        pi[q] = k;
    }

    return pi;
}

// procurar padrão p na string t
bool kmp(string t, string p) {
    int n = t.length();
    int m = p.length();
    // se o sufixo for vazio, não ocorre
    if (m == 0) return false;
    vector<int> pi = compute_prefix(p);
    int q = 0;

    for (int i = 0; i < n; i++) {
        while (q > 0 && p[q] != t[i]) {
            q = pi[q-1];
        }
        if (p[q] == t[i]) {
            q = q + 1;
        }
        if (q == m) {
            int start_pos = i - m + 1;
            // cout << "found " << p << " in " << t << " with shift " << start_pos << "\n";
            //se o padrão for todo igual e não estiver no início nem no fim
            if (start_pos > 0 && start_pos < n - m) return true;
            q = pi[q-1];
        }
    }
    return false;
}

int main() {
    string original;
    cin >> original;

    string possible = original;
    // calcular respostas possíveis, começando pela maior e ir diminuindo
    do {
        vector<int> pi = compute_prefix(possible);
        int l = possible.length();
        possible = possible.substr(l-pi[l-1], pi[l-1]);

        if (kmp(original, possible)) {
            cout << possible << "\n";
            return 0;
        }
    } while (possible.length() > 0);

    cout << "Just a legend\n";
    return 0;
}
```

Trie

cpp

```cpp
// Implementação de Trie retirada de https://www.geeksforgeeks.org/dsa/trie-delete/
// (acabei por não usar a função remove(), mas a estrutura veio de lá)
// Modificada para receber letras maiúsculas e guardar pontuação nos nós que são fins de strings.
//
// Receber o tabuleiro 4 x 4, guardar em board[][].
// Receber as palavras a procurar, e guardar numa Trie.
// Fazer um DFS em todas as casas do tabuleiro.
// No DFS, vamos acompanhar a 'descida' no board com uma 'descida' da Trie.
// Se nalgum ponto não conseguirmos acompanhar a Trie, sair do DFS.
// Por exemplo, no tabuleiro:
// FNEI
// OBCN
// EERI
// VSIR
// A palavra BEER começa na posição (1,1), e a Trie é root->B->E->E->R(isEndOfWord)
// O DFS a partir de (1,1) encontra a letra B, que também é encontrada a partir da raíz da Trie.
// Logo, podemos continuar a procurar, e agora tentamos encontrar um E (a próxima da letra da Trie).
// Ao visitar o vizinho da esquerda, verifica-se que O não pertence aos filhos de B na Trie,
// por isso podemos acabar o DFS aí.
//
const int ALPHABET_SIZE = 26;

int calc_score(int n) {
    switch (n) {
        case 3: return 1;
        case 4: return 1;
        case 5: return 2;
        case 6: return 3;
        case 7: return 5;
        default: return 11;
    }
}

struct TrieNode {
    TrieNode* children[ALPHABET_SIZE];
    bool isEndOfWord;
    int word_score;

    TrieNode() {
        isEndOfWord = false;
        word_score = 0;
        for (int i = 0; i < ALPHABET_SIZE; i++) {
            children[i] = nullptr;
        }
    }
};

TrieNode* getNode() {
    return new TrieNode();
```

```
}

void insert(TrieNode* root, const string& key) {
    TrieNode* pCrawl = root;

    for (int i = 0; i < (int)key.length(); i++) {
        int index = key[i] - 'A';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    pCrawl->isEndOfWord = true;
    pCrawl->word_score = calc_score(key.length());
}

bool isEmpty(TrieNode* root) {
    for (int i = 0; i < ALPHABET_SIZE; i++)
        if (root->children[i])
            return false;
    return true;
}

vector<vector<char>> board(4, vector<char>(4));
vector<vector<bool>> visited(4, vector<bool>(4, false));
vector<pair<int,int>> neighbours = {
    {-1,-1},    //NW
    {-1, 0},    //N
    {-1, 1},    //NE
    { 0, 1},    //E
    { 1, 1},    //SE
    { 1, 0},    //S
    { 1,-1},    //SW
    { 0,-1}     //W
};

// dfs com backtracking
void dfs(int i, int j, TrieNode* curr, string& s, TrieNode* root, int& total_score) {
    // se estivermos fora dos limites ou já tiver sido visitado, fazer nada
    if (i < 0 || i >= 4 || j < 0 || j >= 4 || visited[i][j]) return;

    char c = board[i][j];
    int idx = c - 'A';
    // se a letra não estiver no caminho da trie, a palavra não existe, fazer nada
    if (curr->children[idx] == NULL) return;

    visited[i][j] = true;
    curr = curr->children[idx];
    s.push_back(c);
    // se encontramos uma palavra, pontuar e retirar essa palavra da trie (só pode aparecer uma vez)
    if (curr->isEndOfWord) {
        total_score += curr->word_score;
        curr->isEndOfWord = false;
    }
    // continuar o dfs nos vizinhos
    for (auto &n : neighbours) {
        int x = i + n.first;
        int y = j + n.second;
        if (x >= 0 && x < 4 && y >= 0 && y < 4) {
            dfs(x, y, curr, s, root, total_score);
        }
    }
    // backtrack
    visited[i][j] = false;
    s.pop_back();
}

int main() {
    int n_cases;
    cin >> n_cases;

    for (int game = 1; game <= n_cases; game++) {

        // preencher board
        for (int i = 0; i < 4; i++) {
            string line;
            cin >> line;
            for (int j = 0; j < 4; j++) {
                board[i][j] = line[j];
            }
        }

        int n_words;
        cin >> n_words;

        // criar uma trie com as palavras
        TrieNode* root = new TrieNode();

        for (int i = 0; i < n_words; i++) {
            string word;
            cin >> word;
            insert(root, word);
        }

        int total_score = 0;
        string current_word;
        // dfs no board inteiro
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                // reset de visited
                for (auto &row : visited) {
                    fill(row.begin(), row.end(), false);
                }
                current_word.clear();
                dfs(i, j, root, current_word, root, total_score);
            }
        }
        cout << "Score for Boggle game #" << game << ": " << total_score << "\n";
    }
    return 0;
}
```