

## Counting Sort

```
1 count[max size] ← frequencies array
2 For i = 0 to n - 1 do
3     count[v[i]] ++ (one more v[i] element)
4 i = 0
5 For j = min size to max size do
6     While count[j] > 0 do
7         v[i] = j (put element on array)
8         count[j]-- (one less element of that size)
9         i++ (increments first free position on the array)
```

## A possible RadixSort (starting on the least significant digit)

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

## Binary search on a sorted array

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

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

```
1 bsearch(low, high, condition)
2 While (low < high) do
3     middle = low + (high - low)/2
4     If (condition(middle) == yes) high = middle
5     Else low = middle + 1
6 If (condition(low) == no) return(-1)
7 return(low)
```

## Ternary search

```
1 // a: left edge of interval
2 // b: right edge of interval
3 // f: function
4 // epsilon: tolerance
5 input a, b, f, epsilon
6 l ← a
7 r ← b
8 while (r-l > epsilon)
9     m1 ← (2*l + r)/3
10    m2 ← (l + 2*r)/3
11    if f(m1) < f(m2)
```

```
12     l ← m1
13 else
14     r ← m2
15 print r
```

## Segment tree

```
1 const int MAX = 200005; // Capacity of Segment Tree
2 const int MAX_ST = MAX*4;
3
4 const int NEUTRAL = 0; // Neutral element
5
6 typedef int64_t st_value; // type of segment tree value
7
8 int n; // Number of elements in the segtree
9 st_value v[MAX]; // Array of values
10 st_value st[MAX_ST]; // Segtree (in this case storing interval sums)
11
12 // Merge contents of nodes a and b
13 st_value merge(st_value a, st_value b) {
14     return a+b;
15 }
16
17 // Build initial segtree (in position pos, interval [start,end])
18 void build(int pos, int start, int end) {
19     if (start == end) {
20         st[pos] = v[start];
21     } else {
22         int middle = start + (end-start)/2;
23         build(pos*2, start, middle);
24         build(pos*2+1, middle+1, end);
25         st[pos] = merge(st[pos*2], st[pos*2+1]);
26     }
27 }
28
29 // Update node n to value v
30 void update(int pos, int start, int end, int n, st_value v) {
31     if (start > n || end < n) return;
32     if (start == end) {
33         st[pos] = v;
34     } else {
35         int middle = start + (end-start)/2;
36         update(pos*2, start, middle, n, v);
37         update(pos*2+1, middle+1, end, n, v);
38         st[pos] = merge(st[pos*2], st[pos*2+1]);
39     }
40 }
41
42 // Make a query of interval [a,b]
43 st_value query(int pos, int start, int end, int a, int b) {
44     if (start>b || end<a) return NEUTRAL;
45     if (start≥a & end≤b) return st[pos];
46
47     int middle = start + (end-start)/2;
48     st_value l = query(pos*2, start, middle, a, b);
```

```

49     st_value r = query(pos*2+1, middle+1, end, a, b);
50     return merge(l, r);
51 }
52
53 // -----
54
55 int main() {
56     int q;
57     cin >> n >> q;
58     for (int i=1; i<=n; i++)
59         cin >> v[i];
60
61     build(1, 1, n);
62
63     for (int i=1; i<=q; i++) {
64         int op, a, b;
65         cin >> op >> a >> b;
66         if (op == 1) update(1, 1, n, a, b);
67         else cout << query(1, 1, n, a, b) << endl;
68     }
69
70     return 0;
71 }
```

## Binary Indexed Tree

cpp

```

1 vector<int> tree;
2 int maxIdx;
3
4 int read(int idx) {
5     int sum = 0;
6     while (idx > 0) {
7         sum += tree[idx];
8         idx -= (idx & -idx);
9     }
10    return sum;
11 }
12
13 void update(int idx, int val) {
14     while (idx <= MaxIdx) {
15         tree[idx] += val;
16         idx += (idx & -idx);
17     }
18 }
19
20 int sum(int l, int r) { // Range Query
21     return read(r) - read(l - 1);
22 }
23
24 int readSingle(int idx) { // Read the actual number, not cumulative
25     int sum = tree[idx]; // this sum will be decreased
26     if (idx > 0) { // the special case
27         int z = idx - (idx & -idx);
28         idx--; // idx is not important anymore, so instead y, you can use idx
29         while (idx != z) { // at some iteration idx (y) will become z
30             sum -= tree[idx];
31             // subtract tree frequency which is between y and "the same path"
32             idx -= (idx & -idx);
33         }
34     }
35     return sum;
36 }
37
38 void scale(int c) { // Scale entire tree by c
39     for (int i = 1; i <= MaxIdx; i++)
40         tree[i] = tree[i] / c;
41 }
```

```

30     sum -= tree[idx];
31     // subtract tree frequency which is between y and "the same path"
32     idx -= (idx & -idx);
33 }
34 }
35 return sum;
36 }
37
38 void scale(int c) { // Scale entire tree by c
39     for (int i = 1; i <= MaxIdx; i++)
40         tree[i] = tree[i] / c;
41 }
```

## Kadane

py

```

1 def max_subarray(numbers):
2     """Find the largest sum of any contiguous subarray."""
3     best_sum = float('-inf')
4     current_sum = 0
5     for x in numbers:
6         current_sum = max(x, current_sum + x)
7         best_sum = max(best_sum, current_sum)
8     return best_sum
```

## DFS

```

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

## Finding Connected Components

```

1 counter ← 0
2 set all nodes as not visited
3 For all nodes v of the graph do
4     If v has not yet been visited then
5         counter++
6         dfs(v)
7 write(counter)
```

## Topological Sorting

```

1 order ← empty
2 set all nodes as not visited
3 For all nodes v of the graph do
4     If v has not yet been visited then
5         dfs(v)
6 write(order)
7
8 dfs(node v):
9     mark v as visited
10    For all neighbors w of v do
11        If w has not yet been visited then
```

```

12     dfs(w)
13     add v to the beginning of order

```

## Cycle Detection

```

1 color[v ∈ V] ← white
2 For all nodes v of the graph do
3     If color[v] = white then
4         dfs(v)
5
6 dfs(node v):
7     color[v] ← gray
8     For all neighbors w of v do
9         If color[w] = gray then
10            write("Cycle found!")
11        Else if color[w] = white then
12            dfs(w)
13        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!

```

1 index ← 0 ; S ← ∅
2 For all nodes v of the graph do
3     If num[v] is still undefined then
4         dfs_scc(v)
5
6 dfs scc(node v):
7     num[v] ← low [v] ← index ; index ← index + 1 ; S.push(v)
8     /* Traverse edges of v */
9     For all neighbors w of v do
10        If num[w] is still undefined then /* Tree Edge */
11            dfs scc(w) ; low [v] ← min(low [v], low [w])
12        Else if w is in S then /* Back Edge */
13            low [v] ← min(low [v], num[w])
14        /* We know that we are at the root of an SCC */
15        If num[v] = low [v] then
16            Start new SCC C
17            Repeat
18                w ← S.pop() ; Add w to C
19            Until w = v
20            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!

```

1 dfs art(node v):
2     num[v] ← low[v] ← index ; index ← index + 1 ; S.push(v)

```

```

3     For all neighbors w of v do
4         If num[w] is not yet defined then /* Tree Edge */
5             dfs art(w) ; low [v] ← min(low [v], low [w])
6             If low [w] ≥ num[v] then
7                 write(v + " is an articulation point")
8             Else if w is in S then /* Back Edge */
9                 low [v] ← min(low [v], num[w])
10            S.pop()

```

## BFS - Computing Distances

```

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

```

## Edmonds-Karp (Fluxo máximo)

cpp

```

1 // Classe que representa um grafo
2 class Graph {
3 public:
4     int n; // Numero de nos do grafo
5     vector<vector <int>> adj; // Lista de adjacencias
6     vector<vector <int>> cap; // Matriz de capacidades
7
8     Graph(int n) {
9         this->n = n;
10        adj.resize(n+1); // +1 se os nos comecam em 1 ao inves de 0
11        cap.resize(n+1);
12        for (int i=1; i<=n; i++) cap[i].resize(n+1);
13    }
14
15    void addLink(int a, int b, int c) {
16        // adjacencias do grafo nao dirigido, porque podemos ter de andar no sentido
17        // contrario ao procurarmos caminhos de aumento
18        adj[a].push_back(b);
19        adj[b].push_back(a);
20        cap[a][b] = c;
21    }
22
23    // BFS para encontrar caminho de aumento
24    // devolve valor do fluxo nesse caminho
25    int bfs(int s, int t, vector<int> &parent) {
26        for (int i=1; i<=n; i++) parent[i] = -1;
27
28        parent[s] = -2;
29        queue<pair<int, int>> q; // fila do BFS com pares (no, capacidade)

```

```

30     q.push({s, INT_MAX});    // inicializar com no origem e capacidade infinita
31
32     while (!q.empty()) {
33         // retornar primeiro no da fila
34         int cur = q.front().first;
35         int flow = q.front().second;
36         q.pop();
37
38         // percorrer nos adjacentes ao no atual (cur)
39         for (int next : adj[cur]) {
40             // se o vizinho ainda nao foi visitado (parent== -1)
41             // e a aresta respetiva ainda tem capacidade para passar fluxo
42             if (parent[next] == -1 && cap[cur][next]>0) {
43                 parent[next] = cur;                      // atualizar pai
44                 int new_flow = min(flow, cap[cur][next]); // atualizar fluxo
45                 if (next == t) return new_flow;           // chegamos ao final?
46                 q.push({next, new_flow});               // adicionar a fila
47             }
48         }
49     }
50
51     return 0;
52 }
53
54 // Algoritmo de Edmonds-Karp para fluxo maximo entre s e t
55 // devolve valor do fluxo maximo (cap[][] fica com grafo residual)
56 int maxFlow(int s, int t) {
57     int flow = 0;          // fluxo a calcular
58     vector<int> parent(n+1); // vetor de pais (permite reconstruir caminho)
59
60     while (true) {
61         int new_flow = bfs(s, t, parent); // fluxo de um caminho de aumento
62         if (new_flow == 0) break;        // se nao existir, terminar
63
64         // imprimir fluxo e caminho de aumento
65         cout << "Caminho de aumento: fluxo " << new_flow << " | " << t;
66
67         flow += new_flow; // aumentar fluxo total com fluxo deste caminho
68         int cur = t;
69         while (cur != s) { // percorrer caminho de aumento e alterar arestas
70             int prev = parent[cur];
71             cap[prev][cur] -= new_flow;
72             cap[cur][prev] += new_flow;
73             cur = prev;
74             cout << " <- " << cur; // imprimir proximo no do caminho
75         }
76         cout << endl;
77     }
78
79     return flow;
80 }
81 }
82
83 int main() {
84     int n, e, a, b, c;

```

```

85
86     cin >> n;
87     Graph g(n);
88     cin >> e;
89     for (int i=0; i<e; i++) {
90         cin >> a >> b >> c;
91         g.addLink(a, b, c);
92     }
93
94     // Execucao exemplo usando 1 como no origem a 4 como o destino
95     int flow = g.maxFlow(1, 4);
96     cout << "Fluxo maximo: " << flow << endl;
97
98     return 0;
99 }

```

## Dijkstra

cpp

```

1 // Classe que representa um no
2 class Node {
3 public:
4     list<pair<int, int>> adj; // Lista de adjacencias
5     bool visited;              // No ja foi visitado?
6     int distance;             // Distancia ao no origem da pesquisa
7 };
8
9 // Classe que representa um grafo
10 class Graph {
11 public:
12     int n;                   // Numero de nos do grafo
13     Node *nodes;             // Array para conter os nos
14
15     Graph(int n) { // Constructor: chamado quando um objeto Graph for criado
16         this->n = n;
17         nodes = new Node[n+1]; // +1 se os nos comecam em 1 ao inves de 0
18     }
19
20     ~Graph() { // Destructor: chamado quando um objeto Graph for destruido
21         delete[] nodes;
22     }
23
24     void addLink(int a, int b, int c) {
25         nodes[a].adj.push_back({b,c});
26     }
27
28     // Algoritmo de Dijkstra
29     void dijkstra(int s) {
30
31         //Inicializar nos como nao visitados e com distancia infinita
32         for (int i=1; i<=n; i++) {
33             nodes[i].distance = INT_MAX;
34             nodes[i].visited = false;
35         }
36
37         // Inicializar "fila" com no origem

```

```

38     nodes[s].distance = 0;
39     set<pair<int, int>> q; // By "default" um par e comparado pelo primeiro elemento
40     q.insert({0, s});      // Criar um par (dist=0, no=s)
41
42     // Ciclo principal do Dijkstra
43     while (!q.empty()) {
44
45         // Retirar no com menor distancia (o "primeiro" do set, que e uma BST)
46         int u = q.begin()->second;
47         q.erase(q.begin());
48         nodes[u].visited = true;
49         cout << u << " [dist=" << nodes[u].distance << "] " << endl;
50
51         // Relaxar arestas do no retirado
52         for (auto edge : nodes[u].adj) {
53             int v = edge.first;
54             int cost = edge.second;
55             if (!nodes[v].visited && nodes[u].distance + cost < nodes[v].distance) {
56                 q.erase({nodes[v].distance, v}); // Apagar do set
57                 nodes[v].distance = nodes[u].distance + cost;
58                 q.insert({nodes[v].distance, v}); // Inserir com nova (e menor) distancia
59             }
60         }
61     }
62 }
63 };
64
65 int main() {
66     int n, e, a, b, c;
67
68     cin >> n;
69     Graph g(n);
70     cin >> e;
71     for (int i=0; i<e; i++) {
72         cin >> a >> b >> c;
73         g.addLink(a, b, c);
74     }
75
76     // Execucao exemplo a partir do no 1
77     g.dijkstra(1);
78
79     return 0;
80 }
```

## 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

```

1 // We approach this problem from a different angle: for every not yet selected vertex we
2 // will store the minimum edge to an already selected vertex.
3 // Then during a step we only have to look at these minimum weight edges, which will have
4 // a complexity of O(n).
5 // After adding an edge some minimum edge pointers have to be recalculated. Note that the
6 // weights only can decrease, i.e. the minimal weight edge of every not yet selected vertex
7 // might stay the same, or it will be updated by an edge to the newly selected vertex.
8 // Therefore this phase can also be done in O(n).
9
10 int n;
11 vector<vector<int>> adj; // adjacency matrix of graph
12 const int INF = 1000000000; // weight INF means there is no edge
13
14 struct Edge {
15     int w = INF, to = -1;
16 };
17
18 void prim() {
19     int total_weight = 0;
20     vector<bool> selected(n, false);
21     vector<Edge> min_e(n);
22     min_e[0].w = 0;
23
24     for (int i=0; i<n; ++i) {
25         int v = -1;
26         for (int j = 0; j < n; ++j) {
27             if (!selected[j] && (v == -1 || min_e[j].w < min_e[v].w))
28                 v = j;
29         }
30
31         if (min_e[v].w == INF) {
32             cout << "No MST!" << endl;
33             exit(0);
34         }
35
36         selected[v] = true;
37         total_weight += min_e[v].w;
38         if (min_e[v].to != -1)
39             cout << v << " " << min_e[v].to << endl;
40
41         for (int to = 0; to < n; ++to) {
42             if (adj[v][to] < min_e[to].w)
43                 min_e[to] = {adj[v][to], v};
44         }
45     }
46     cout << total_weight << endl;
47 }
```

### Sparse Graphs

cpp

```

1 // We can find the minimum edge in  $O(\log n)$  time.
2 // On the other hand recomputing the pointers will now take  $O(n \log n)$  time, which is
3 // worse than in the previous algorithm.
4 // But when we consider that we only need to update  $O(m)$  times in total, and perform  $O(n)$ 
5 // searches for the minimal edge, then the total complexity will be  $O(m \log n)$ . For sparse
6 // graphs this is better than the above algorithm, but for dense graphs this will be slower.
7
8 const int INF = 1000000000;
9
10 struct Edge {
11     int w = INF, to = -1;
12     bool operator<(Edge const& other) const {
13         return make_pair(w, to) < make_pair(other.w, other.to);
14     }
15 };
16
17 int n;
18 vector<vector<Edge>> adj;
19
20 void prim() {
21     int total_weight = 0;
22     vector<Edge> min_e(n);
23     min_e[0].w = 0;
24     set<Edge> q;
25     q.insert({0, 0});
26     vector<bool> selected(n, false);
27     for (int i = 0; i < n; ++i) {
28         if (q.empty()) {
29             cout << "No MST!" << endl;
30             exit(0);
31         }
32         int v = q.begin()->to;
33         selected[v] = true;
34         total_weight += q.begin()->w;
35         q.erase(q.begin());
36         if (min_e[v].to != -1)
37             cout << v << " " << min_e[v].to << endl;
38         for (Edge e : adj[v]) {
39             if (!selected[e.to] && e.w < min_e[e.to].w) {
40                 q.erase({min_e[e.to].w, e.to});
41                 min_e[e.to] = {e.w, v};
42                 q.insert({e.w, e.to});
43             }
44         }
45     }
46     cout << total_weight << endl;
47 }

```

## 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

```

1 struct Edge {
2     int u, v, weight;
3     bool operator<(Edge const& other) {
4         return weight < other.weight;
5     }
6 };
7
8 int n;
9 vector<Edge> edges;
10
11 int cost = 0;
12 vector<int> tree_id(n);
13 vector<Edge> result;
14 for (int i = 0; i < n; i++)
15     tree_id[i] = i;
16
17 sort(edges.begin(), edges.end());
18
19 for (Edge e : edges) {
20     if (tree_id[e.u] != tree_id[e.v]) {
21         cost += e.weight;
22         result.push_back(e);
23
24         int old_id = tree_id[e.u], new_id = tree_id[e.v];
25         for (int i = 0; i < n; i++) {
26             if (tree_id[i] == old_id)
27                 tree_id[i] = new_id;
28         }
29     }
30 }

```

## 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

```

1 vector<int> parent, rank;
2
3 void make_set(int v) {
4     parent[v] = v;
5     rank[v] = 0;

```

```

6 }
7
8 int find_set(int v) {
9     if (v == parent[v])
10        return v;
11    return parent[v] = find_set(parent[v]);
12 }
13
14 void union_sets(int a, int b) {
15    a = find_set(a);
16    b = find_set(b);
17    if (a != b) {
18        if (rank[a] < rank[b])
19            swap(a, b);
20        parent[b] = a;
21        if (rank[a] == rank[b])
22            rank[a]++;
23    }
24 }
25
26 struct Edge {
27     int u, v, weight;
28     bool operator<(Edge const& other) {
29         return weight < other.weight;
30     }
31 };
32
33 int n;
34 vector<Edge> edges;
35
36 int cost = 0;
37 vector<Edge> result;
38 parent.resize(n);
39 rank.resize(n);
40 for (int i = 0; i < n; i++)
41    make_set(i);
42
43 sort(edges.begin(), edges.end());
44
45 for (Edge e : edges) {
46    if (find_set(e.u) != find_set(e.v)) {
47        cost += e.weight;
48        result.push_back(e);
49        union_sets(e.u, e.v);
50    }
51 }

```

## 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  $d[0 \dots n - 1]$ , 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  $\infty$ .

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  $\infty$ .

cpp

```

1 struct Edge {
2     int a, b, cost;
3 };
4
5 int n, m, v;
6 vector<Edge> edges;
7 const int INF = 1000000000;
8
9 void solve()
10 {
11    vector<int> d(n, INF);
12    d[v] = 0;
13    for (;;) {
14        bool any = false;
15
16        for (Edge e : edges)
17            if (d[e.a] < INF)
18                if (d[e.b] > d[e.a] + e.cost) {
19                    d[e.b] = d[e.a] + e.cost;
20                    any = true;
21                }
22
23        if (!any)
24            break;
25    }
26    // display d, for example, on the screen
27 }

```

## Retrieving Path

cpp

```

1 void solve()
2 {
3    vector<int> d(n, INF);
4    d[v] = 0;
5    vector<int> p(n, -1);
6
7    for (;;) {
8        bool any = false;
9        for (Edge e : edges)
10            if (d[e.a] < INF)
11                if (d[e.b] > d[e.a] + e.cost) {
12                    d[e.b] = d[e.a] + e.cost;
13                    p[e.b] = e.a;
14                    any = true;
15                }
16        if (!any)
17            break;

```

```

18 }
19
20 if (d[t] == INF)
21     cout << "No path from " << v << " to " << t << ".";
22 else {
23     vector<int> path;
24     for (int cur = t; cur != -1; cur = p[cur])
25         path.push_back(cur);
26     reverse(path.begin(), path.end());
27
28     cout << "Path from " << v << " to " << t << ":" ;
29     for (int u : path)
30         cout << u << ' ';
31 }
32 }

```

## Negative Cycle

cpp

```

1 void solve()
2 {
3     vector<int> d(n, INF);
4     d[v] = 0;
5     vector<int> p(n, -1);
6     int x;
7     for (int i = 0; i < n; ++i) {
8         x = -1;
9         for (Edge e : edges)
10            if (d[e.a] < INF)
11                if (d[e.b] > d[e.a] + e.cost) {
12                    d[e.b] = max(-INF, d[e.a] + e.cost);
13                    p[e.b] = e.a;
14                    x = e.b;
15                }
16    }
17
18    if (x == -1)
19        cout << "No negative cycle from " << v;
20    else {
21        int y = x;
22        for (int i = 0; i < n; ++i)
23            y = p[y];
24
25        vector<int> path;
26        for (int cur = y;; cur = p[cur]) {
27            path.push_back(cur);
28            if (cur == y && path.size() > 1)
29                break;
30        }
31        reverse(path.begin(), path.end());
32
33        cout << "Negative cycle: ";
34        for (int u : path)
35            cout << u << ' ';
36    }
37 }

```

## 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  $\infty$  will be some high value.

cpp

```

1 for (int k = 0; k < n; ++k) {
2     for (int i = 0; i < n; ++i) {
3         for (int j = 0; j < n; ++j) {
4             if (d[i][k] < INF && d[k][j] < INF)
5                 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
6         }
7     }
8 }

```

## KMP

### Pseudocódigo dos slides:

```

1 KMP-Matcher(T, P)
2 n = T.length
3 m = P.length
4 pi = Compute-Prefix-Function(P)
5 q = 0
6 for i = 1 to n
7     while q > 0 and P[q+1] != T[i]
8         q = pi[q]
9     if P[q+1] == T[i]
10        q = q+1
11    if q == m
12        print "Pattern occurs with shift" i-m
13        q = pi[q]
14
15 Compute-Prefix-Function(P)
16 m = P.length
17 let pi[1..m] be a new array
18 pi[1] = 0
19 k = 0
20 for q = 2 to m
21     while k > 0 and P[k+1] != P[q]
22         k = pi[k]
23     if P[k+1] == P[q]
24         k = k+1
25     pi[q] = k
26 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  $[0, 1, 0, 1, 2, 2, 3]$ .

```

1 vector<int> prefix_function(string s) {
2     int n = (int)s.length();
3     vector<int> pi(n);
4     for (int i = 1; i < n; i++) {
5         int j = pi[i-1];
6         while (j > 0 && s[i] != s[j])
7             j = pi[j-1];
8         if (s[i] == s[j])
9             j++;
10        pi[i] = j;
11    }
12    return pi;
13 }

```

cpp

**Trie**

```

1 const int ALPHABET_SIZE = 26;
2
3 // trie node
4 struct TrieNode {
5     struct TrieNode* children[ALPHABET_SIZE];
6
7     // isEndOfWord is true if the node represents end of a word
8     bool isEndOfWord;
9 };
10
11 // Returns new trie node (initialized to NULLs)
12 struct TrieNode* getNode(void)
13 {
14     struct TrieNode* pNode = new TrieNode;
15
16     pNode->isEndOfWord = false;
17
18     for (int i = 0; i < ALPHABET_SIZE; i++)
19         pNode->children[i] = NULL;
20
21     return pNode;
22 }
23
24 // If not present, inserts key into trie
25 // If the key is prefix of trie node, just marks leaf node
26 void insert(struct TrieNode* root, string key)
27 {
28     struct TrieNode* pCrawl = root;
29
30     for (int i = 0; i < key.length(); i++) {
31         int index = key[i] - 'a';
32         if (!pCrawl->children[index])
33             pCrawl->children[index] = getNode();
34
35         pCrawl = pCrawl->children[index];
36     }
37 }

```

cpp

```

38     // mark last node as leaf
39     pCrawl->isEndOfWord = true;
40 }
41
42 // Returns true if key presents in trie, else false
43 bool search(struct TrieNode* root, string key)
44 {
45     struct TrieNode* pCrawl = root;
46
47     for (int i = 0; i < key.length(); i++) {
48         int index = key[i] - 'a';
49         if (!pCrawl->children[index])
50             return false;
51
52         pCrawl = pCrawl->children[index];
53     }
54
55     return (pCrawl != NULL && pCrawl->isEndOfWord);
56 }
57
58 // Returns true if root has no children, else false
59 bool isEmpty(TrieNode* root)
60 {
61     for (int i = 0; i < ALPHABET_SIZE; i++)
62         if (root->children[i])
63             return false;
64     return true;
65 }
66
67 // Recursive function to delete a key from given Trie
68 TrieNode* remove(TrieNode* root, string key, int depth = 0)
69 {
70     // If tree is empty
71     if (!root)
72         return NULL;
73
74     // If last character of key is being processed
75     if (depth == key.size()) {
76
77         // This node is no more end of word after removal of given key
78         if (root->isEndOfWord)
79             root->isEndOfWord = false;
80
81         // If given is not prefix of any other word
82         if (isEmpty(root)) {
83             delete (root);
84             root = NULL;
85         }
86
87         return root;
88     }
89
90     // If not last character, recur for the child obtained using ASCII value
91     int index = key[depth] - 'a';
92     root->children[index] =

```

```

93     remove(root->children[index], key, depth + 1);
94
95 // If root does not have any child (its only child got
96 // deleted), and it is not end of another word.
97 if (isEmpty(root) && root->isEndOfWord == false) {
98     delete (root);
99     root = NULL;
100 }
101
102 return root;
103 }
104
105 // Driver
106 int main()
107 {
108 // Input keys (use only 'a' through 'z' and lower case)
109 string keys[] = { "the", "a", "there",
110                     "answer", "any", "by",
111                     "bye", "their", "hero", "heroplane" };
112 int n = sizeof(keys) / sizeof(keys[0]);
113
114 struct TrieNode* root = getNode();
115
116 // Construct trie
117 for (int i = 0; i < n; i++)
118     insert(root, keys[i]);
119
120 // Search for different keys
121 search(root, "the") ? cout << "Yes\n" : cout << "No\n";
122 search(root, "these") ? cout << "Yes\n" : cout << "No\n";
123
124 remove(root, "heroplane");
125 search(root, "hero") ? cout << "Yes\n" : cout << "No\n";
126 return 0;
127 }

```

## 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

```

1 // Max number of states in the matching machine.
2 // Should be equal to the sum of the length of all keywords.
3 const int MAXS = 500;
4
5 // Maximum number of characters in input alphabet
6 const int MAXC = 26;
7
8 // OUTPUT FUNCTION IS IMPLEMENTED USING out[]
9 // Bit i in this mask is one if the word with index i
10 // appears when the machine enters this state.
11 int out[MAXS];
12
13 // FAILURE FUNCTION IS IMPLEMENTED USING f[]
14 int f[MAXS];
15
16 // GOTO FUNCTION (OR TRIE) IS IMPLEMENTED USING g[][][]
17 int g[MAXS][MAXC];
18
19 // Builds the string matching machine.
20 // arr - array of words. The index of each keyword is important:
21 //         "out[state] & (1 << i)" is > 0 if we just found word[i]
22 //         in the text.
23 // Returns the number of states that the built machine has.
24 // States are numbered 0 up to the return value - 1, inclusive.
25 int buildMatchingMachine(string arr[], int k)
26 {
27     // Initialize all values in output function as 0.
28     memset(out, 0, sizeof out);
29
30     // Initialize all values in goto function as -1.
31     memset(g, -1, sizeof g);
32
33     // Initially, we just have the 0 state
34     int states = 1;
35
36     // Construct values for goto function, i.e., fill g[][][]
37     // This is same as building a Trie for arr[]
38     for (int i = 0; i < k; ++i)
39     {
40         const string &word = arr[i];
41         int currentState = 0;
42
43         // Insert all characters of current word in arr[]
44         for (int j = 0; j < word.size(); ++j)
45         {
46             int ch = word[j] - 'a';
47
48             // Allocate a new node (create a new state) if a
49             // node for ch doesn't exist.
50             if (g[currentState][ch] == -1)

```

```

51         g[currentState][ch] = states++;
52
53     currentState = g[currentState][ch];
54 }
55
56 // Add current word in output function
57 out[currentState] |= (1 << i);
58 }
59
60 // For all characters which don't have an edge from
61 // root (or state 0) in Trie, add a goto edge to state
62 // 0 itself
63 for (int ch = 0; ch < MAXC; ++ch)
64     if (g[0][ch] == -1)
65         g[0][ch] = 0;
66
67 // Now, let's build the failure function
68
69 // Initialize values in fail function
70 memset(f, -1, sizeof f);
71
72 // Failure function is computed in breadth first order
73 // using a queue
74 queue<int> q;
75
76 // Iterate over every possible input
77 for (int ch = 0; ch < MAXC; ++ch)
78 {
79     // All nodes of depth 1 have failure function value
80     // as 0. For example, in above diagram we move to 0
81     // from states 1 and 3.
82     if (g[0][ch] != 0)
83     {
84         f[g[0][ch]] = 0;
85         q.push(g[0][ch]);
86     }
87 }
88
89 // Now queue has states 1 and 3
90 while (q.size())
91 {
92     // Remove the front state from queue
93     int state = q.front();
94     q.pop();
95
96     // For the removed state, find failure function for
97     // all those characters for which goto function is
98     // not defined.
99     for (int ch = 0; ch <= MAXC; ++ch)
100    {
101        // If goto function is defined for character 'ch'
102        // and 'state'
103        if (g[state][ch] != -1)
104        {
105            // Find failure state of removed state

```

```

106         int failure = f[state];
107
108         // Find the deepest node labeled by proper
109         // suffix of string from root to current
110         // state.
111         while (g[failure][ch] == -1)
112             failure = f[failure];
113
114         failure = g[failure][ch];
115         f[g[state][ch]] = failure;
116
117         // Merge output values
118         out[g[state][ch]] |= out[failure];
119
120         // Insert the next level node (of Trie) in Queue
121         q.push(g[state][ch]);
122     }
123 }
124 }
125
126 return states;
127 }
128
129 // Returns the next state the machine will transition to using goto
130 // and failure functions.
131 // currentState - The current state of the machine. Must be between
132 // 0 and the number of states - 1, inclusive.
133 // nextInput - The next character that enters into the machine.
134 int findNextState(int currentState, char nextInput)
135 {
136     int answer = currentState;
137     int ch = nextInput - 'a';
138
139     // If goto is not defined, use failure function
140     while (g[answer][ch] == -1)
141         answer = f[answer];
142
143     return g[answer][ch];
144 }
145
146 // This function finds all occurrences of all array words
147 // in text.
148 void searchWords(string arr[], int k, string text)
149 {
150     // Preprocess patterns.
151     // Build machine with goto, failure and output functions
152     buildMatchingMachine(arr, k);
153
154     // Initialize current state
155     int currentState = 0;
156
157     // Traverse the text through the built machine to find
158     // all occurrences of words in arr[]
159     for (int i = 0; i < text.size(); ++i)
160     {

```

```

161     currentState = findNextState(currentState, text[i]);
162
163     // If match not found, move to next state
164     if (out[currentState] == 0)
165         continue;
166
167     // Match found, print all matching words of arr[]
168     // using output function.
169     for (int j = 0; j < k; ++j)
170     {
171         if (out[currentState] & (1 << j))
172         {
173             cout << "Word " << arr[j] << " appears from "
174                 << i - arr[j].size() + 1 << " to " << i << endl;
175         }
176     }
177 }
178 }
179
180 // Driver program to test above
181 int main()
182 {
183     string arr[] = {"he", "she", "hers", "his"};
184     string text = "ahishers";
185     int k = sizeof(arr)/sizeof(arr[0]);
186
187     searchWords(arr, k, text);
188
189     return 0;
190 }

```

## Midpoint of a line

```

1 // function to find the midpoint of a line
2 void midpoint(int x1, int x2,
3                 int y1, int y2)
4 {
5     cout << (float)(x1+x2)/2 <<
6         " , "<< (float)(y1+y2)/2 ;
7 }
8
9 // Driver Function to test above
10 int main()
11 {
12     int x1 = -1, y1 = 2 ;
13     int x2 = 3, y2 = -6 ;
14     midpoint(x1, x2, y1, y2);
15     return 0;
16 }

```

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

```

1 // Function to find the section of the line
2 void section(double x1, double x2, double y1,
3              double y2, double m, double n)

```

```

4 {
5     // Applying section formula
6     double x = ((n * x1) + (m * x2)) /
7                     (m + n);
8     double y = ((n * y1) + (m * y2)) /
9                     (m + n);
10
11    // Printing result
12    cout << "(" << x << ", ";
13    cout << y << ")" << endl;
14 }
15
16 // Driver code
17 int main()
18 {
19     double x1 = 2, x2 = 4, y1 = 4,
20         y2 = 6, m = 2, n = 3;
21     section(x1, x2, y1, y2, m, n);
22     return 0;
23 }

```

## Slope of a line

```

1 // function to find the slope of a straight line
2 float slope(float x1, float y1, float x2, float y2)
3 {
4     if (x2 - x1 != 0)
5         return (y2 - y1) / (x2 - x1);
6     return INT_MAX;
7 }
8
9 // driver code to check the above function
10 int main()
11 {
12     float x1 = 4, y1 = 2;
13     float x2 = 2, y2 = 5;
14     cout << "Slope is: " << slope(x1, y1, x2, y2);
15     return 0;
16 }

```

## Line Intersection

The idea is to use orientation of lines to determine whether they intersect or not. Two line segments  $[p_1, q_1]$  and  $[p_2, q_2]$  intersect if and only if one of the following two conditions is verified:

1. General Case:
  - $[p_1, q_1, p_2]$  and  $[p_1, q_1, q_2]$  have different orientations.
  - $[p_2, q_2, p_1]$  and  $[p_2, q_2, q_1]$  have different orientations.
2. Special Case:
  - $[p_1, q_1, p_2], [p_1, q_1, q_2], [p_2, q_2, p_1]$ , and  $[p_2, q_2, q_1]$  are all collinear.
  - The x-projections of  $[p_1, q_1]$  and  $[p_2, q_2]$  intersect.
  - The y-projections of  $[p_1, q_1]$  and  $[p_2, q_2]$  intersect.

```

1 // function to check if point q lies on line segment 'pr'

```

```

2 bool onSegment(vector<int>& p, vector<int>& q, vector<int>& r) {
3     return (q[0] <= max(p[0], r[0]) &&
4            q[0] >= min(p[0], r[0]) &&
5            q[1] <= max(p[1], r[1]) &&
6            q[1] >= min(p[1], r[1]));
7 }
8
9 // function to find orientation of ordered triplet (p, q, r)
10 // 0 --> p, q and r are collinear
11 // 1 --> Clockwise
12 // 2 --> Counterclockwise
13 int orientation(vector<int>& p, vector<int>& q, vector<int>& r) {
14     int val = (q[1] - p[1]) * (r[0] - q[0]) -
15               (q[0] - p[0]) * (r[1] - q[1]);
16
17     // collinear
18     if (val == 0) return 0;
19
20     // clock or counterclock wise
21     // 1 for clockwise, 2 for counterclockwise
22     return (val > 0) ? 1 : 2;
23 }
24
25
26 // function to check if two line segments intersect
27 bool doIntersect(vector<vector<vector<int>>>& points) {
28
29     // find the four orientations needed
30     // for general and special cases
31     int o1 = orientation(points[0][0], points[0][1], points[1][0]);
32     int o2 = orientation(points[0][0], points[0][1], points[1][1]);
33     int o3 = orientation(points[1][0], points[1][1], points[0][0]);
34     int o4 = orientation(points[1][0], points[1][1], points[0][1]);
35
36     // general case
37     if (o1 != o2 && o3 != o4)
38         return true;
39
40     // special cases
41     // p1, q1 and p2 are collinear and p2 lies on segment p1q1
42     if (o1 == 0 &&
43         onSegment(points[0][0], points[1][0], points[0][1])) return true;
44
45     // p1, q1 and q2 are collinear and q2 lies on segment p1q1
46     if (o2 == 0 &&
47         onSegment(points[0][0], points[1][1], points[0][1])) return true;
48
49     // p2, q2 and p1 are collinear and p1 lies on segment p2q2
50     if (o3 == 0 &&
51         onSegment(points[1][0], points[0][0], points[1][1])) return true;
52
53     // p2, q2 and q1 are collinear and q1 lies on segment p2q2
54     if (o4 == 0 &&
55         onSegment(points[1][0], points[0][1], points[1][1])) return true;
56

```

```

57     return false;
58 }
59
60 int main() {
61     vector<vector<vector<int>>> points =
62     {{ {1, 1}, {10, 1} }, { {1, 2}, {10, 2} } };
63
64     if (doIntersect(points))
65         cout << "Yes";
66     else cout << "No";
67
68     return 0;
69 }

```

## 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

```

1
2 struct Point {
3     double x, y;
4 };
5
6 // Checking if a point is inside a polygon
7 bool point_in_polygon(Point point, vector<Point> polygon)
8 {
9     int num_vertices = polygon.size();
10    double x = point.x, y = point.y;
11    bool inside = false;
12
13    // Store the first point in the polygon and initialize
14    // the second point
15    Point p1 = polygon[0], p2;
16
17    // Loop through each edge in the polygon
18    for (int i = 1; i <= num_vertices; i++) {
19        // Get the next point in the polygon
20        p2 = polygon[i % num_vertices];
21
22        // Check if the point is above the minimum y
23        // coordinate of the edge
24        if (y > min(p1.y, p2.y)) {
25            // Check if the point is below the maximum y
26            // coordinate of the edge
27            if (y <= max(p1.y, p2.y)) {
28                // Check if the point is to the left of the
29                // maximum x coordinate of the edge
30                if (x <= max(p1.x, p2.x)) {
31                    // Calculate the x-intersection of the
32                    // line connecting the point to the edge
33                    double x_intersection

```

```

34             = (y - p1.y) * (p2.x - p1.x)
35             / (p2.y - p1.y)
36             + p1.x;
37
38         // Check if the point is on the same
39         // line as the edge or to the left of
40         // the x-intersection
41         if (p1.x == p2.x
42             || x <= x_intersection) {
43             // Flip the inside flag
44             inside = !inside;
45         }
46     }
47 }
48
49     // Store the current point as the first point for
50     // the next iteration
51     p1 = p2;
52 }
53
54     // Return the value of the inside flag
55     return inside;
56 }
57 }
58
59 // Driver code
60 int main()
61 {
62     // Define a point to test
63     Point point = { 150, 85 };
64
65     // Define a polygon
66     vector<Point> polygon = {
67         { 186, 14 }, { 186, 44 }, { 175, 115 }, { 175, 85 }
68     };
69
70     // Check if the point is inside the polygon
71     if (point_in_polygon(point, polygon)) {
72         cout << "Point is inside the polygon" << endl;
73     }
74     else {
75         cout << "Point is outside the polygon" << endl;
76     }
77     return 0;
78 }
```

## Exemplos de código

### Pesquisa binária

cpp

```

1 // Criar um vector com as máquinas e inserir as máquinas, computando a máquina que demora
2 // mais tempo.
3 // Declarar o limite inferior (1 segundo) e superior (máquina que demora mais tempo a
4 // produzir
4 // todos os produtos).
```

```

5 // Fazer pesquisa binária nesse espaço de possibilidades: O(log n).
6 // A verificação da condição is_possible decide se é possível, num determinado tempo,
7 // o número de produtos pretendido: O(n)
8
9 bool is_possible(vector<unsigned long long> machines, unsigned long long target_products,
10     unsigned long long time) {
11     unsigned long long products_made = 0;
12     for (unsigned long long m : machines) {
13         products_made += time / m;
14     }
15     return products_made >= target_products;
16 }
17
18 unsigned long long my_binary_search(vector<unsigned long long> machines, unsigned long
long lower, unsigned long long upper, unsigned long long target_products) {
19     while (lower < upper) {
20         unsigned long long middle = lower + (upper - lower) / 2;
21         if (is_possible(machines, target_products, middle)) {
22             upper = middle;
23         } else {
24             lower = middle + 1;
25         }
26     }
27     if (!is_possible(machines, target_products, lower)) {
28         return -1;
29     }
30     return lower;
31 }
32 }
33
34 int main() {
35     unsigned long long n, t;
36     cin >> n >> t;
37
38     unsigned long long highest_value = 0;
39     vector<unsigned long long> machines;
40
41     for (unsigned long long i = 0; i < n; i++) {
42
43         unsigned long long machine;
44         cin >> machine;
45
46         machines.push_back(machine);
47
48         if (machine > highest_value) {
49             highest_value = machine;
50         }
51     }
52
53     unsigned long long lower_bound = 1;
54     unsigned long long upper_bound = t * highest_value;
55 }
```

```

56     unsigned long long minimum_time = my_binary_search(machines, lower_bound,
57     upper_bound, t);
58
59     cout << minimum_time << "\n";
60 }

```

cpp

```

1 // Muito semelhante ao PC012, usar pesquisa binária no espaço de possibilidades.
2 //
3 // Duas diferenças: valores contínuos e o problema agora é de maximização (em vez de
4 // minimização).
5 // Para os valores contínuos, modificar o critério de paragem para ser um intervalo maior
6 // ou igual que a precisão pretendida.
7 // Para a maximização, o espaço de possibilidades é do tipo:
8 // [yes, yes,..., yes, no, no, ..., no]
9 // por isso, quando o middle verifica a condição, fazemos low = middle, e quando não
10 // verifica
11 // fazemos high = middle (o oposto da minimização do problema PC012).
12 // Para a condição booleana is_possible, usar novamente um algoritmo greedy que tenta
13 // dividir o volume de uma tarte pelo volume hipotético para obter o número de pessoas
14 // que
15 // pode alimentar.
16
17 bool is_possible(vector<int> pies, int people, double volume) {
18     int max_people = 0;
19     for (auto p : pies) {
20         max_people += int(floor((M_PI * p * p)/volume));
21     }
22
23     return max_people >= people;
24 }
25
26 double my_binary_search(vector<int> pies, double low, double high, int people) {
27     while (high-low >= 0.0001) {
28         double middle = low + (high - low) / 2;
29
30         if (is_possible(pies, people, middle)) {
31             low = middle;
32         } else {
33             high = middle;
34         }
35     }
36     return low;
37 }
38 int main() {
39
40     int test_cases;
41     cin >> test_cases;
42
43     for (int i = 0; i < test_cases; i++) {
44         int nr_pies, friends;
45         cin >> nr_pies >> friends;

```

```

46
47     int people = friends + 1;
48
49     vector<int> pies;
50     int biggest_pie = 0;
51     for (int j = 0; j < nr_pies; j++) {
52         int p;
53         cin >> p;
54         pies.push_back(p);
55         if (p > biggest_pie) {
56             biggest_pie = p;
57         }
58     }
59
60     double upper_bound = M_PI * biggest_pie * biggest_pie;
61     double lower_bound = 0;
62
63     double volume = my_binary_search(pies, lower_bound, upper_bound, people);
64
65     cout << fixed;
66     cout << setprecision(4);
67     cout << volume << "\n";
68 }
69
70 }

```

## Pesquisa Ternária

cpp

```

1 // Pesquisa ternária no espaço de possibilidades k (altura final dos edifícios).
2 //
3 // Função cost(k) calcula o custo de ter todos os edifícios à altura k: O(n).
4 // Fazer uma pesquisa ternária para o k mínimo: O(log3 n).
5 // A pesquisa ternária tem de ter um critério de paragem ligeiramente diferente porque,
6 // como o espaço de pesquisa é discreto, a divisão ternária não pode ser aplicada a
7 // espaços
8 // menores que 3.
9 // Assim, quando reduzimos o espaço de possibilidades a 3 inteiros, fazemos uma pesquisa
10 // linear nesses 3 inteiros.
11
12 typedef struct {
13     long height;
14     long cost;
15 } Building;
16
17 long cost(vector<Building> buildings, long k) {
18     long cost = 0;
19     for (auto x : buildings) {
20         cost += abs(x.height - k) * x.cost;
21     }
22     return cost;
23 }
24
25 long my_ternary_search(vector<Building> buildings, long lower, long upper) {
26     while (upper - lower > 3) {
27

```

```

27     long m1 = lower + (upper - lower) / 3;
28     long m2 = upper - (upper - lower) / 3;
29
30     long cost1 = cost(buildings, m1);
31     long cost2 = cost(buildings, m2);
32
33     if (cost1 < cost2) {
34         upper = m2;
35     } else if (cost1 > cost2) {
36         lower = m1;
37     } else {
38         lower = m1;
39         upper = m2;
40     }
41 }
42
43 long min_cost = cost(buildings, lower);
44 long k = lower;
45
46 for (long i = lower; i <= upper; i++) {
47     long c = cost(buildings, i);
48     if (c < min_cost) {
49         min_cost = c;
50         k = i;
51     }
52 }
53
54 return k;
55 }
56
57 int main() {
58     long nr_tests;
59     cin >> nr_tests;
60
61     for (long i = 0; i < nr_tests; i++) {
62         long nr_buildings;
63         cin >> nr_buildings;
64
65         vector<Building> buildings;
66         long highest_building = 0;
67
68         for (long j = 0; j < nr_buildings; j++) {
69             long height_in;
70             cin >> height_in;
71
72             Building b;
73             b.height = height_in;
74             b.cost = 0;
75             buildings.push_back(b);
76
77             if (height_in > highest_building) {
78                 highest_building = height_in;
79             }
80         }
81     }

```

```

82     for (long j = 0; j < nr_buildings; j++) {
83         long cost_in;
84         cin >> cost_in;
85         buildings[j].cost = cost_in;
86     }
87
88     long lower = 1;
89     long upper = highest_building;
90
91     long k = my_ternary_search(buildings, lower, upper);
92     cout << cost(buildings, k) << "\n";
93 }
94
95 return 0;
96 }

```

## Segment Tree

cpp

```

1 // Temos de modificar a segtree para guardar pares: o valor e a frequênciadesse valor.
2 // Quando construímos a árvore, as frequências são 1 para todos os números.
3 // No merge, escolhemos o maior dos números para passar para o nó pai.
4 // Mas se os números forem iguais, então somamos a frequência.
5
6 const int MAX = 200005; // Capacity of Segment Tree
7 const int MAX_ST = MAX * 4;
8
9 const pair<int, int> NEUTRAL = {0, 0}; // Neutral element
10
11 typedef pair<int, int> st_value; // type of segment tree value
12
13 int n; // Number of elements in the segtree
14 st_value v[MAX]; // Array of values
15 st_value st[MAX_ST]; // Segtree (in this case storing interval sums)
16
17 // Merge contents of nodes a and b
18 st_value merge(st_value a, st_value b) {
19     if (a.first > b.first) {
20         return a;
21     }
22     if (b.first > a.first) {
23         return b;
24     }
25     return make_pair(a.first, a.second + b.second);
26 }
27
28 // Build initial segtree (in position pos, interval [start,end])
29 void build(int pos, int start, int end) {
30     if (start == end) {
31         st[pos] = v[start];
32     } else {
33         int middle = start + (end - start) / 2;
34         build(pos * 2, start, middle);
35         build(pos * 2 + 1, middle + 1, end);
36         st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
37     }

```

```

38 }
39
40 // Update node n to value v
41 void update(int pos, int start, int end, int n, st_value v) {
42     if (start > n || end < n)
43         return;
44     if (start == end) {
45         st[pos] = v;
46     } else {
47         int middle = start + (end - start) / 2;
48         update(pos * 2, start, middle, n, v);
49         update(pos * 2 + 1, middle + 1, end, n, v);
50         st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
51     }
52 }
53
54 // Make a query of interval [a,b]
55 st_value query(int pos, int start, int end, int a, int b) {
56     if (start > b || end < a)
57         return NEUTRAL;
58     if (start >= a && end <= b)
59         return st[pos];
60
61     int middle = start + (end - start) / 2;
62     st_value l = query(pos * 2, start, middle, a, b);
63     st_value r = query(pos * 2 + 1, middle + 1, end, a, b);
64     return merge(l, r);
65 }
66
67 int main() {
68     int q;
69     cin >> n >> q;
70     for (int i = 1; i <= n; i++) {
71         int w;
72         cin >> w;
73         v[i] = make_pair(w, 1);
74     }
75
76     build(1, 1, n);
77
78     for (int i = 1; i <= q; i++) {
79         int a, b;
80         cin >> a >> b;
81         pair<int, int> answer = query(1, 1, n, a, b);
82         cout << answer.first << " " << answer.second << "\n";
83     }
84
85     return 0;
86 }

```

cpp

```

1 // Guardar maps na segtree. Os pares do mapa representam o valor e a frequência desse
   // valor.
2 // Em cada nó da tree, o map tem no máximo 3 valores: o mais frequente,
3 // o valor mais à esquerda, e o valor mais à direita (do intervalo correspondente).
4 //

```

```

5 // Assim, para fazer um merge, é preciso:
6 // verificar se os dois valores mais "interiores" são iguais. se forem, somar as
   // frequências.
7 // Calcular o valor mais frequente.
8 // Colocar no novo map: o mais frequente, e os dois valores dos "extremos".
9 //
10 // Isto porque os "extremos" podem somar-se com os extremos de outros ramos da árvore.
11 //
12 // Como o elemento neutro é um map vazio, verificar maps vazios antes de aceder aos
   // iteradores.
13
14 const int MAX = 200005; // Capacity of Segment Tree
15 const int MAX_ST = MAX * 4;
16
17 const map<int,int> NEUTRAL = {};// Neutral element
18
19 typedef map<int,int> st_value; // type of segment tree value
20
21 int n; // Number of elements in the segtree
22 st_value v[MAX]; // Array of values
23 st_value st[MAX_ST]; // Segtree (in this case storing interval sums)
24
25 // Merge contents of nodes a and b
26 st_value merge(st_value a, st_value b) {
27     map<int,int> return_map;
28
29     int most_frequent = 0;
30     int frequency = 0;
31
32     // calcular se o último elemento do map da direita é
33     // igual ao primeiro elemento do map da esquerda
34     if (!a.empty() && !b.empty()) {
35         auto last_a = a.end();
36         last_a--;
37         auto first_b = b.begin();
38
39         // se forem iguais, somar as frequencias e assumir como "mais frequente" para já
40         if (last_a->first == first_b->first) {
41             most_frequent = last_a->first;
42             frequency = last_a->second + first_b->second;
43         }
44     }
45
46     // verificar o valor mais frequente nos dois maps
47     for (auto x : a) {
48         if (x.second > frequency) {
49             frequency = x.second;
50             most_frequent = x.first;
51         }
52     }
53
54     for (auto x : b) {
55         if (x.second > frequency) {
56             frequency = x.second;
57             most_frequent = x.first;
58         }
59     }
60
61     return_map[most_frequent] = frequency;
62     return_map[st_value{most_frequent, frequency}];
63     return_map[st_value{most_frequent, frequency}];
64
65     return return_map;
66 }

```

```

58     }
59 }
60
61 // preencher novo return_map com: o mais frequente,
62 // o mais à esquerda do mapa esquerdo, e o mais à direita do mapa direito
63 if (frequency > 0) {
64     return_map.insert({most_frequent, frequency});
65 }
66
67 if (!a.empty()) {
68     auto first_a = a.begin();
69     return_map.insert(*first_a);
70 }
71 if (!b.empty()) {
72     auto last_b = b.end();
73     last_b--;
74     return_map.insert(*last_b);
75 }
76
77 return return_map;
78 }
79
80 // Build initial segtree (in position pos, interval [start,end])
81 void build(int pos, int start, int end) {
82     if (start == end) {
83         st[pos] = v[start];
84     } else {
85         int middle = start + (end - start) / 2;
86         build(pos * 2, start, middle);
87         build(pos * 2 + 1, middle + 1, end);
88         st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
89     }
90 }
91
92 // Update node n to value v
93 void update(int pos, int start, int end, int n, st_value v) {
94     if (start > n || end < n)
95         return;
96
97     if (start == end) {
98         st[pos] = v;
99     } else {
100        int middle = start + (end - start) / 2;
101        update(pos * 2, start, middle, n, v);
102        update(pos * 2 + 1, middle + 1, end, n, v);
103        st[pos] = merge(st[pos * 2], st[pos * 2 + 1]);
104    }
105 }
106
107
108 // Make a query of interval [a,b]
109 st_value query(int pos, int start, int end, int a, int b) {
110     if (start > b || end < a)
111         return NEUTRAL;
112     if (start >= a && end <= b)

```

```

113     return st[pos];
114
115     int middle = start + (end - start) / 2;
116     st_value l = query(pos * 2, start, middle, a, b);
117     st_value r = query(pos * 2 + 1, middle + 1, end, a, b);
118     return merge(l, r);
119 }
120
121 int main() {
122     int q;
123     cin >> n >> q;
124     for (int i = 1; i <= n; i++) {
125         int w;
126         cin >> w;
127
128         map<int,int> m;
129         m.insert({w,1});
130         v[i] = m;
131     }
132
133     build(1, 1, n);
134
135     for (int i = 1; i <= q; i++) {
136         int a, b;
137         cin >> a >> b;
138
139         map<int,int> q_answer = query(1, 1, n, a, b);
140
141         int frequency = 0;
142         for (auto x : q_answer) {
143             if (x.second > frequency) {
144                 frequency = x.second;
145             }
146         }
147
148         cout << frequency << "\n";
149     }
150     return 0;
151 }

// 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

```

```

15
16 int n; // Number of elements in the segtree
17 st_value v[MAX]; // Array of values
18 st_value st[MAX_ST]; // Segtree (in this case storing interval sums)
19
20 // Merge contents of nodes a and b
21 st_value merge(st_value a, st_value b) {
22     return max(a,b);
23 }
24
25 // Build initial segtree (in position pos, interval [start,end])
26 void build(int pos, int start, int end) {
27     if (start == end) {
28         st[pos] = v[start];
29     } else {
30         int middle = start + (end-start)/2;
31         build(pos*2, start, middle);
32         build(pos*2+1, middle+1, end);
33         st[pos] = merge(st[pos*2], st[pos*2+1]);
34     }
35 }
36
37 // Update node n to value v
38 void update(int pos, int start, int end, int n, st_value v) {
39     if (start > n || end < n) return;
40     if (start == end) {
41         st[pos] = v;
42     } else {
43         int middle = start + (end-start)/2;
44         update(pos*2, start, middle, n, v);
45         update(pos*2+1, middle+1, end, n, v);
46         st[pos] = merge(st[pos*2], st[pos*2+1]);
47     }
48 }
49
50 // Make a query of interval [a,b]
51 st_value query(int pos, int start, int end, int a, int b) {
52     if (start>b || end<a) return NEUTRAL;
53     if (start>=a && end<=b) return st[pos];
54
55     int middle = start + (end-start)/2;
56     st_value l = query(pos*2, start, middle, a, b);
57     st_value r = query(pos*2+1, middle+1, end, a, b);
58     return merge(l, r);
59 }
60
61 // -----
62
63 int main() {
64     int q;
65     cin >> n >> q;
66     for (int i=1; i<=n; i++) {
67         int h;
68         cin >> h;
69         v[i] = h;

```

```

70     }
71
72     build(1, 1, n);
73
74     for (int i = 0; i < q; i++) {
75         int tourists;
76         cin >> tourists;
77
78         if (st[1] < tourists) {
79             cout << 0;
80
81         } else {
82             // binary search na segment tree
83             int cur = 1;
84             int start = 1;
85             int end = n;
86             while (start != end) {
87                 int middle = start + (end-start)/2;
88
89                 if (st[2 * cur] >= tourists) {
90                     cur = 2 * cur;
91                     end = middle;
92
93                 } else {
94                     cur = 2 * cur + 1;
95                     start = middle + 1;
96                 }
97
98                 cout << start;
99                 update(1, 1, n, start, st[cur] - tourists);
100            }
101
102            if (i < q - 1) {
103                cout << " ";
104            }
105        }
106        cout << "\n";
107    }
108 }

```

cpp

```

1 // Receber a sequência de números e guardar para usar mais tarde:
2 // sequence = [0,5,1,2,3,4,5]
3 //
4 // Receber primeiro as queries, para poder criar uma única segtree que responde a todos
os queries:
5 // queries = [(2,4,1),(4,4,4),(1,6,2)]
6 //
7 // Pôr todos os K das queries num set, para remover repetidos e ordenar:
8 // query_set = [1,2,4]
9 //
10 // Criar um dicionário auxiliar para ser mais fácil aceder a indexes:
11 // query_dictionary = [{1:0},{2:1},{4:2}]
12 //
13 // Criar o array auxiliar v[] para usar o build() da segtree. Cada elemento da segtree é
um

```

```

14 // vector que nos diz, a cada posição, se os elementos desse range são maiores do que um
15 // K, para
16 // Por exemplo, o primeiro elemento da sequência (5) é maior que 1, 2, e 4, logo:
17 // v[1] = [1,1,1]
18 // O quarto elemento da sequência (3) é só maior que 1 e 2, logo:
19 // v[4] = [1,1,0]
20 //
21 // O merge() da segtree é simplesmente somar os vectores elemento a elemento, tendo em
22 // atenção
23 // vectores vazios (NEUTRAL são vectores vazios).
24 //
25 // A resposta a um query da árvore é, portanto, um vector, que representa o número de
26 // vezes que,
27 // nesse range, os elementos da sequência são maiores que os K todos.
28 // Depois é só usar o query_dictionary para descobrir o índice desse vector que
29 // corresponde ao
30 // query k que queremos responder.
31
32 const int MAX = 200005; // Capacity of Segment Tree
33 const int MAX_ST = MAX*4;
34
35
36 int n; // Number of elements in the segtree
37 st_value v[MAX]; // Array of values
38 st_value st[MAX_ST]; // Segtree (in this case storing interval sums)
39
40 // Merge contents of nodes a and b
41 st_value merge(st_value a, st_value b) {
42     // verificar vectores vazios, usar tamanho maior
43     int size_a = a.size();
44     int size_b = b.size();
45     int size = max(size_a, size_b);
46
47     vector<int> merge_result;
48
49     for (int i = 0; i < size; i++) {
50         if (a.empty()){
51             merge_result.push_back(b[i]);
52         } else if (b.empty()) {
53             merge_result.push_back(a[i]);
54         } else {
55             merge_result.push_back(a[i]+b[i]);
56         }
57     }
58
59     return merge_result;
60 }
61
62 // Build initial segtree (in position pos, interval [start,end])
63 void build(int pos, int start, int end) {
64     if (start == end) {

```

```

65         st[pos] = v[start];
66     } else {
67         int middle = start + (end-start)/2;
68         build(pos*2, start, middle);
69         build(pos*2+1, middle+1, end);
70         st[pos] = merge(st[pos*2], st[pos*2+1]);
71     }
72 }
73
74 // Update node n to value v
75 void update(int pos, int start, int end, int n, st_value v) {
76     if (start > n || end < n) return;
77     if (start == end) {
78         st[pos] = v;
79     } else {
80         int middle = start + (end-start)/2;
81         update(pos*2, start, middle, n, v);
82         update(pos*2+1, middle+1, end, n, v);
83         st[pos] = merge(st[pos*2], st[pos*2+1]);
84     }
85 }
86
87 // Make a query of interval [a,b]
88 st_value query(int pos, int start, int end, int a, int b) {
89     if (start>b || end<a) return NEUTRAL;
90     if (start>=a && end<=b) return st[pos];
91
92     int middle = start + (end-start)/2;
93     st_value l = query(pos*2, start, middle, a, b);
94     st_value r = query(pos*2+1, middle+1, end, a, b);
95     return merge(l, r);
96 }
97
98 // -----
99
100 int main() {
101     int q;
102     cin >> n >> q;
103     vector<int> sequence;
104     sequence.push_back(0); // sentinel para sequence[0]
105
106     // construir sequencia de números
107     for (int i = 0; i < n; i++) {
108         int number;
109         cin >> number;
110         sequence.push_back(number);
111     }
112
113     vector<tuple<int,int,int>> queries;
114     set<int> query_set;
115
116     // construir vector de queries (para responder depois)
117     // construir query_set (para saber que números tenho de responder na segtree)
118     for (int i = 0; i < q; i++) {
119         tuple<int,int,int> query;

```

```

120     int a, b, k;
121     cin >> a >> b >> k;
122
123     get<0>(query) = a;
124     get<1>(query) = b;
125     get<2>(query) = k;
126
127     queries.push_back(query);
128     query_set.insert(k);
129 }
130
131 // construir um dicionário para ser mais fácil saber o índice dos queries
132 map<int,int> query_dictionary;
133 int counter = 0;
134 for (auto x : query_set) {
135     query_dictionary.insert({x, counter});
136     counter++;
137 }
138
139 // construir o array v[] para o build() da segtree
140 for (int i = 1; i <= n; i++) {
141     vector<int> v_k;
142     for (auto x : query_set) {
143         if (sequence[i] > x) {
144             v_k.push_back(1);
145         } else {
146             v_k.push_back(0);
147         }
148     }
149     v[i] = v_k;
150 }
151
152 build(1, 1, n);
153
154 for (auto x : queries) {
155     int a, b, k;
156     tie(a, b, k) = x;
157
158     vector<int> query_result = query(1, 1, n, a, b);
159     int index = query_dictionary[k];
160     int result = query_result[index];
161
162     cout << result << "\n";
163 }
164 return 0;
165 }
```

## Cumulative Sums

cpp

```

1 // Construir a matriz de somas cumulativas.
2 // Fixar duas linhas da matriz, e usar Kadane como se as colunas delimitadas por essas
3 // linhas fossem elementos únicos (a soma da coluna).
4
5 // soma de uma coluna j, desde i_start até i_end
6 int column(vector<vector<int>> & sums, int i_start, int i_end, int j) {
```

```

7     return sums[i_end][j] - sums[i_start-1][j] - sums[i_end][j-1] + sums[i_start-1][j-1];
8 }
9
10 // kadane numa dimensão 'n', delimitada por duas linhas 'a' e 'b'
11 int kadane(vector<vector<int>> & sums, int size, int a, int b) {
12     int current_sum = column(sums, a, b, 1);
13     int best_sum = current_sum;
14
15     for (int n = 2; n <= size; n++) {
16         current_sum = max(column(sums, a, b, n), current_sum + column(sums, a, b, n));
17         best_sum = max(best_sum, current_sum);
18     }
19
20     return best_sum;
21 }
22
23
24 int main() {
25     int n;
26     cin >> n;
27
28     // construir matriz e matriz de somas acumuladas
29     vector<vector<int>> cumSums(n + 1, vector<int>(n + 1, 0));
30
31     for (int i = 1; i <= n; i++) {
32         for (int j = 1; j <= n; j++) {
33             int num;
34             cin >> num;
35             cumSums[i][j] = num + cumSums[i][j-1];
36         }
37     }
38
39     for (int i = 1; i <= n; i++) {
40         for (int j = 1; j <= n; j++) {
41             cumSums[j][i] = cumSums[j][i] + cumSums[j-1][i];
42         }
43     }
44
45     int max_sum = -101;
46
47     // fixar linhas (a e b) e usar Kadane nessa dimensão
48     for (int a = 1; a <= n; a++) {
49         for (int b = a; b <= n; b++) {
50             int sum_at = kadane(cumSums, n, a, b);
51             if (sum_at > max_sum) {
52                 max_sum = sum_at;
53             }
54         }
55     }
56
57     cout << max_sum << "\n";
58     return 0;
59 }
```

## Binary Indexed Tree

```
1 // Guardar as arestas num dicionário, assim ficam ordenadas por 'n'.
2 //
3 // Uma aresta NiMi vai ter intersecções se as arestas anteriores forem do tipo NkMj, com
4 // k < i e j > i.
5 // k < i é garantindo porque estamos a percorrer as arestas ordenadas por 'n'.
6 // Só temos de verificar quantos valores j > i apareceram até agora.
7 //
8 // Ou seja, quantas arestas que começam em nós anteriores de N acabam em nós posteriores
9 // de M.
10 // Só precisamos de guardar as frequências dos 'm' que apareceram até agora (numa BIT) e,
11 // para cada aresta NM, perguntar quantos 'm' maiores que M apareceram.
12 //
13 // Ou seja, fazer um range query a uma BIT tree do tipo sum[l,r] (que é sum[0,r] -
14 // sum[0,l-1])
15 // Para os updates, temos de guardar os valores de 'm' num stack temporário e só fazer
16 // updates
17 // se o 'n' seguinte for diferente. Isto porque, por exemplo, as arestas 3,1 e 3,2 não se
18 // intersectam.
19
20 #define ll long long int
21
22 // Implementação da BIT
23
24 vector<ll> tree;
25 ll maxIdx;
26
27 ll read(ll idx) {
28     ll sum = 0;
29     while (idx > 0) {
30         sum += tree[idx];
31         idx -= (idx & -idx);
32     }
33     return sum;
34 }
35
36 void update(ll idx, ll val) {
37     while (idx <= maxIdx) {
38         tree[idx] += val;
39         idx += (idx & -idx);
40     }
41 }
42
43 int main() {
44     ll n, m, k;
45     cin >> n >> m >> k;
46
47     maxIdx = m;
48     multimap<ll,ll> roads;
```

cpp

```
49     tree.assign(m+1, 0);
50
51     for (ll i = 0; i < k; i++) {
52         ll n1, m1;
53         cin >> n1 >> m1;
54         roads.insert({n1, m1});
55     }
56
57     stack<ll> temp_update;
58     ll prev = 0;
59     ll crossing_sum = 0;
60
61     for (auto x : roads) {
62         if (x.first != prev) {
63             while (!temp_update.empty()) {
64                 update(temp_update.top(), 1);
65                 temp_update.pop();
66             }
67         }
68
69         ll temp_sum = sum(x.second + 1, maxIdx);
70         crossing_sum += temp_sum;
71
72         temp_update.push(x.second);
73         prev = x.first;
74     }
75 }
76 cout << crossing_sum << "\n";
77 return 0;
78 }
```

cpp

## Longest Common Subsequence

```
1 #define INF 1e6 + 4
2
3 vector<int> alice_seq;
4 vector<int> bob_seq;
5 vector<vector<int>> dp;
6
7 void receive_seq(vector<int> &seq, int sz) {
8     seq.push_back(-INF);
9     for (int i = 1; i <= sz; i++) {
10         int num;
11         cin >> num;
12         seq.push_back(num);
13     }
14 }
15
16 // longest common subsequence, recursivo com memoization
17 // i: índice na sequência da alice
18 // j: índice na sequência do bob
19 int lcs_rec(int i, int j) {
20     // caso base
21     // se uma ou ambas as sequências estiverem vazias.
22     // não pode ser zero porque podemos ter respostas negativas.
```

cpp

```

23 if (i == 0 || j == 0) {
24     return -INF;
25 }
26 // memoization
27 // verificar na tabela dp se a resposta já foi calculada
28 if (dp[i][j] != -INF) {
29     return dp[i][j];
30 }
31 // 3 hipóteses:
32 // 1) usar os números dos índices indicados: o nosso valor total vai incrementar da
multiplicação
33 // (ou vai ser melhor que um valor muito negativo), continuar a procurar em i-1, j-1
34 // 2) descartar o número 'j' do bob, continuar a procurar em i, j-1
35 // 3) descartar o número 'i' da alice, continuar a procurar em i-1, j
36 int use_both = max(alice_seq[i] * bob_seq[j] + lcs_rec(i-1, j-1), alice_seq[i] *
bob_seq[j]);
37 int use_alice = lcs_rec(i, j-1);
38 int use_bob = lcs_rec(i-1, j);
39 // retornar e guardar o valor em dp ao mesmo tempo
40 return dp[i][j] = max(use_both, max(use_alice, use_bob));
41 }
42
43 int main() {
44     int n;
45     cin >> n;
46     receive_seq(alice_seq, n);
47
48     int m;
49     cin >> m;
50     receive_seq(bob_seq, m);
51     // tabela de memoization, usar 'inf' por causa dos valores negativos
52     dp.assign(n+1, vector<int> (m+1, -INF));
53
54     cout << lcs_rec(n, m) << "\n";
55     return 0;
56 }

```

## Bellman-Ford

cpp

```

1 // Estrutura do Grafo do Professor Pedro Ribeiro
2 //
3 // Bellman-Ford aplicado quase directamente, trocar só a inicialização dos nós para -inf
4 // porque é um problema de maximização.
5 //
6 // Na verificação de ciclos, só retornar -1 se o relaxamento existir (obviamente) mas
também só se:
7 // 1) o nó relaxado consegue chegar ao nó final E
8 // 2) o nó inicial consegue chegar ao nó relaxado
9 // estas condições são verificadas com um DFS reaches_n(v, z) que verifica se um nó v
consegue
10 // chegar ao nó z.
11
12 #define INF LONG_LONG_MAX
13 typedef long long ll;
14

```

```

15 // Classe que representa um no
16 class Node {
17 public:
18     list<pair<int, int>> adj; // Lista de adjacencias
19     ll distance; // Distancia ao no origem da pesquisa
20     int parent;
21     int visited;
22 };
23
24 // Classe que representa um grafo
25 class Graph {
26 public:
27     int n; // Numero de nos do grafo
28     Node *nodes; // Array para conter os nos
29
30     Graph(int n) { // Constructor: chamado quando um objeto Graph for criado
31         this->n = n;
32         nodes = new Node[n+1]; // +1 se os nos começam em 1 ao invés de 0
33     }
34
35 ~Graph() { // Destructor: chamado quando um objeto Graph for destruido
36     delete[] nodes;
37 }
38
39 void addLink(int a, int b, int c) {
40     nodes[a].adj.push_back({b,c});
41 }
42
43 bool dfs(int v, int z) {
44     if (v == z) {
45         return true;
46     }
47     nodes[v].visited = true;
48     for (auto e : nodes[v].adj) {
49         if (!nodes[e.first].visited && dfs(e.first, z)) {
50             return true;
51         }
52     }
53     return false;
54 }
55
56 bool reaches_n(int v, int z) {
57     // cout << "testar se " << v << " chega a " << z << "\n";
58     for (int i = 1; i <= n; i++) {
59         nodes[i].visited = false;
60     }
61     bool reaches = dfs(v, z);
62     // cout << reaches << "\n";
63     return reaches;
64 }
65
66 // Bellman-Ford
67 ll bellman_ford() {
68
69     // inicializar nós a -inf, estamos à procura da pontuação máxima

```

```

70     for (int i = 1; i <= n; i++) {
71         nodes[i].distance = -INF;
72     }
73     // pontuação inicial
74     nodes[1].distance = 0;
75     nodes[1].parent = 0;
76
77     // percorrer todas as arestas V-1 vezes
78     for (int i = 1; i < n; i++) {
79         for (int v = 1; v <= n; v++) {
80             for (auto e : nodes[v].adj) {
81                 int neighbour = e.first;
82                 int score = e.second;
83                 if (nodes[v].distance + score > nodes[neighbour].distance) {
84                     nodes[neighbour].distance = nodes[v].distance + score;
85                     nodes[neighbour].parent = v;
86                 }
87             }
88         }
89     }
90
91     // detetar ciclos - fazer uma última iteração de todas as arestas
92     for (int v = 1; v <= n; v++) {
93         for (auto e : nodes[v].adj) {
94             int neighbour = e.first;
95             int score = e.second;
96             // retornar -1 só se o nó estiver envolvido no caminho de maior pontuação
97             if (reaches_n(neighbour, n) && reaches_n(1, v) && nodes[v].distance +
    score > nodes[neighbour].distance) {
98                 return -1;
99             }
100        }
101    }
102    return nodes[n].distance;
103 }
104 };
105
106 int main() {
107     int n, e, a, b, c;
108
109     cin >> n;
110     Graph g(n);
111     cin >> e;
112     for (int i=0; i<e; i++) {
113         cin >> a >> b >> c;
114         g.addLink(a, b, c);
115     }
116
117     cout << g.bellman_ford() << "\n";
118
119     return 0;
120 }
```

## Grafo Bipartido

cpp

```

1 // Usei a implementação de Edmonds-Karp do Professor Pedro Ribeiro de DAA.
2 //
3 // Só modifiquei a inserção de dados no grafo, para transformar os inputs desta maneira:
4 // O número de nós do grafo é 2n+2. 2n porque é um grafo bipartido.
5 // Por exemplo, para o caso de teste:
6 // 5
7 // 5
8 // 0 1
9 // 1 2
10 // 2 3
11 // 3 4
12 // 4 2
13 //
14 // 5 nós e 5 arestas transformam-se em 10 nós e 5 arestas:
15 //
16 //      1   6
17 //      2   7
18 //      3   8
19 //      4   9
20 //      5   10
21 //
22 // com arestas 1-7, 2-8, 3-9, 4-10, 5-8
23 //
24 // Para transformar este grafo num problema de fluxo máximo (com capacidades unitárias)
25 // temos de adicionar dois nós guarda:
26 //
27 //      1   6
28 //      2   7
29 //      0   3   8   11
30 //      4   9
31 //      5   10
32 //
33 // e arestas do nó 0 para os nós 1..5 e dos nós 6..10 para o nó 11.
34 //
35 // Depois é só resolver um problema de fluxo máximo, onde as capacidades das arestas são
36 // 1,
37 // e se o fluxo final for igual a 'n' então conseguimos distribuir todos os livros por
38 // todas
39 // as pessoas.
40 //
41 // Classe que representa um grafo
42 class Graph {
43 public:
44     int n; // Número de nos do grafo
45     vector<vector <int>> adj; // Lista de adjacencias
46     vector<vector <int>> cap; // Matriz de capacidades
47
48     Graph(int n) {
49         this->n = n;
50         adj.resize(n); // +1 se os nos comecam em 1 ao invés de 0
51         cap.resize(n);
52         for (int i=0; i<n; i++) cap[i].resize(n);
53     }
54
55     void addLink(int a, int b, int c) {
```

```

54     // adjacencias do grafo nao dirigido, porque podemos ter de andar no sentido
55     // contrario ao procurarmos caminhos de aumento
56     adj[a].push_back(b);
57     adj[b].push_back(a);
58     cap[a][b] = c;
59 }
60
61 // BFS para encontrar caminho de aumento
62 // devolve valor do fluxo nesse caminho
63 int bfs(int s, int t, vector<int> &parent) {
64     for (int i=0; i<n; i++) parent[i] = -1;
65
66     parent[s] = -2;
67     queue<pair<int, int>> q; // fila do BFS com pares (no, capacidade)
68     q.push({s, INT_MAX}); // inicializar com no origem e capacidade infinita
69
70     while (!q.empty()) {
71         // retornar primeiro no da fila
72         int cur = q.front().first;
73         int flow = q.front().second;
74         q.pop();
75
76         // percorrer nos adjacentes ao no atual (cur)
77         for (int next : adj[cur]) {
78             // se o vizinho ainda nao foi visitado (parent== -1)
79             // e a aresta respetiva ainda tem capacidade para passar fluxo
80             if (parent[next] == -1 && cap[cur][next]>0) {
81                 parent[next] = cur; // atualizar pai
82                 int new_flow = min(flow, cap[cur][next]); // atualizar fluxo
83                 if (next == t) return new_flow; // chegamos ao final?
84                 q.push({next, new_flow}); // adicionar a fila
85             }
86         }
87     }
88     return 0;
89 }
90
91 // Algoritmo de Edmonds-Karp para fluxo maximo entre s e t
92 // devolve valor do fluxo maximo (cap[][] fica com grafo residual)
93 int maxFlow(int s, int t) {
94     int flow = 0; // fluxo a calcular
95     vector<int> parent(n+1); // vetor de pais (permite reconstruir caminho)
96
97     while (true) {
98         int new_flow = bfs(s, t, parent); // fluxo de um caminho de aumento
99         if (new_flow == 0) break; // se nao existir, terminar
100
101        // imprimir fluxo e caminho de aumento
102        // cout << "Caminho de aumento: fluxo " << new_flow << " | " << t;
103
104        flow += new_flow; // aumentar fluxo total com fluxo deste caminho
105        int cur = t;
106        while (cur != s) { // percorrer caminho de aumento e alterar arestas
107            int prev = parent[cur];
108            cap[prev][cur] -= new_flow;

```

```

109            cap[cur][prev] += new_flow;
110            cur = prev;
111            // cout << " <- " << cur; // imprimir proximo no do caminho
112        }
113        // cout << endl;
114    }
115    return flow;
116 }
117 };
118
119 int main() {
120     int tests;
121     cin >> tests;
122
123     for (int t = 0; t < tests; t++) {
124         int n, e, a, b;
125         cin >> n;
126         // grafo bipartido com nós guarda
127         Graph g(2*n+2);
128         cin >> e;
129         for (int i=0; i<e; i++) {
130             cin >> a >> b;
131             // grafo bipartido com capacidades unitárias
132             g.addLink(a+1, b+1+n, 1);
133         }
134         // adicionar arestas dos nós guardas (0 e 2n+1)
135         for (int i = 1; i <= n; i++) {
136             g.addLink(0, i, 1);
137             g.addLink(i+n, 2*n+1, 1);
138         }
139         // resolver fluxo máximo no grafo bipartido
140         int flow = g.maxFlow(0, 2*n+1);
141         // se o fluxo for igual a 'n', conseguimos distribuir todos os livros
142         if (flow != n) {
143             cout << "NO\n";
144         }
145         else {
146             cout << "YES\n";
147         }
148     }
149     return 0;
150 }

```