IMPORTANT - don't forget to use long long int

```
#include <bits/stdc++.h>
ios_base::sync_with_stdio(false);
cin >> variable;
getline(cin, line);
istringstream iss(line);
iss>>word;
```

1 Graph

1.1 DFS

```
void DFS(int start, const vector<vector<int>>& graph, vector<bool>& visited) {
    stack<int> s;
    s.push(start);
    while (!s.empty()) {
        int node = s.top();
        s.pop();
        if (!visited[node]) {
            visited[node] = true;
            for (auto it = graph[node].begin(); it != graph[node].end();++it) {
                if (!visited[*it]) {
                    s.push(*it);
                }
            }
        }
    }
}
void DFS(int node, const vector<vector<int>>& graph, vector<bool>& visited) {
    visited[node] = true;
    for (int adjacent : graph[node]) {
        if (!visited[adjacent]) {
            DFS(adjacent, graph, visited);
        }
    }
}
/*for topo sort push node after children then reverse the vec
for cycle detec set recStack=true before and check if there is a neighbour in recStack
for cycle, then reset recStack=false */
bool dfs(int v, vector<bool>& visited, vector<bool>& recStack, vector<int>& parent) {
    visited[v] = true;
    recStack[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) {
            parent[u] = v;
            if (dfs(u, visited, recStack, parent))
                return true;
        } else if (recStack[u]) {
            stack<int> cycle;
            int current = v;
            cycle.push(u);
            while (current != u) {
                cycle.push(current);
                current = parent[current];
            }
            cycle.push(u);
            return true;
        }
    }
    recStack[v] = false;
    return false;
}
```

1.2 BFS

```
void BFS(int start, const vector<vector<int>>& graph, vector<bool>& visited) {
    queue<int> q;
    q.push(start);
    visited[start] = true;
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        for (int adjacent : graph[node]) {
            if (!visited[adjacent]) {
                visited[adjacent] = true;
                q.push(adjacent);
            }
        }
    }
}
```

1.3 Shortest path

1.3.1 Positive weights : Dijkstra O((V + E) log V)

```
typedef pair<int, int> pii; // (distance, vertex)
const int INF = 1e9;
void Dijkstra(int start, const vector<vector<pii>>& graph) {
    priority_queue<pii, vector<pii>, greater<pii>> pq;
    vector<int> distances(graph.size(), INF);
    distances[start] = 0;
    pq.push({0, start});
    while (!pq.empty()) {
        int current_vertex = pq.top().second;
        pq.pop();
        for (const auto& edge : graph[current_vertex]) {
            int next_vertex = edge.first;
            int weight = edge.second;
            int distance = distances[current_vertex] + weight;
            if (distance < distances[next_vertex]) {
                distances[next_vertex] = distance;
                pq.push({distance, next_vertex});
            }
        }
    }
}
```

1.3.2 Positive and negative : Bellman–Ford O(EV)

```
typedef pair<int, int> pii; // Pair to store (vertex, weight)
const int INF = numeric_limits<int>::max();
bool BellmanFord(int start, const vector<vector<pii>>& graph, vector<int>& distances) {
    int n = graph.size();
    distances.assign(n, INF);
    distances[start] = 0;
    for (int i = 0; i < n - 1; ++i) {
        for (int u = 0; u < n; ++u) {
            for (const auto& edge : graph[u]) {
                int v = edge.first;
                int weight = edge.second;
                if (distances[u] != INF && distances[u] + weight < distances[v]) {
                    distances[v] = distances[u] + weight;
                }
            }
        }
    }
```

```
    // Check for negative-weight cycles
    for (int u = 0; u < n; ++u) {
        for (const auto& edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (distances[u] != INF && distances[u] + weight < distances[v]) {
                return false;
            }
        }
    }
    return true;
}
```

1.3.3 Between every pair : Floyd–Warshall O(V^3)

```
void FloydWarshall(vector<vector<int>>& graph) {
    int n = graph.size();
    vector<vector<int>> dist = graph;
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (dist[i][k] != INF && dist[k][j] != INF && ) {
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
                }
            }
        }
    }
}
```

1.4 Cycles and Eulerian Paths O(V + E)

```
int n, m;
int deg[N];
// non orient  : deg[u]++ ; deg[v]++
// orient  : deg[u]-- ; deg[v]++
vector<pair<int, int>> vs[N]; // (vertex, edge_index)
vector<int> path;
bool visited[M];
void dfs(int u) {
    for (auto& e : vs[u]) {
        if (!visited[e.second]) {
            visited[e.second] = true;
            dfs(e.first);
            path.push_back(e.second);
        }
    }
}
bool eulercycle(bool oriented) {
    // Check if all vertices have even degree for undirected graph
    // or if all vertices have zero net degree for directed graph
    for (int u = 0; u < n; u++) {
        if (oriented && deg[u] != 0)
            return false;
        else if (!oriented && deg[u] % 2 != 0)
            return false;
    }
    dfs(0);
    reverse(path.begin(), path.end());
    return true;
}
bool eulerpath(bool oriented) {
    int s = 0;
    for (int u = 0; u < n; u++) {
```

```
        if (oriented && deg[u] > deg[s]) s = u;
        else if (!oriented && deg[u] % 2 == 1) s = u;
    }
    dfs(s);
    reverse(path.begin(), path.end());
    return path.size() == 0;
}
```

2 Data Structures

2.1 Binary search – Dichotomy

```
int binarySearch(const vector<int>& arr, int target) {
    int left = 0;
    int right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) {
            return mid;}
        if (arr[mid] < target) {
            left = mid + 1;}
        else {
            right = mid - 1;}
    }
    return -1;
}
//Maximum x Such That f(x) = true (true true false)
int last_true(int lo, int hi, function<bool(int)> f) {
        lo--;
        while (lo < hi) {
                int mid = lo + (hi - lo + 1) / 2;
                if (f(mid)) {
                        lo = mid;
                } else {
                        hi = mid - 1;
                }
        }
        return lo;
}
```

2.1.1 Ternary search

```
//strictly increasing then strictly decreasing function f
double ternary_search(double l, double r) {
    double eps = 1e-9; //set the error limit here
    while (r - l > eps) {
        double m1 = l + (r - l) / 3;
        double m2 = r - (r - l) / 3;
        double f1 = f(m1); //evaluates the function at m1
        double f2 = f(m2) //evaluates the function at m2
        if (f1 < f2)
            l = m1;
        else
            r = m2;
    }
    return f(l); //return the maximum of f(x) in [l, r]
}
```

2.1.2 Sorting and stats

```
sort(begin, end, [cmp]);
min_element(begin, end, [cmp]);
```

```
max_element(begin, end, [cmp]);
nth_elemnt(begin, begin + nth, end, [cmp]); // put nth in place
random_shuffle(begin, end);
lower_bound(begin, end, val); // premier >= val si tri
upper_bound(begin, end, val); // premier > val si tri
```

2.1.3 Disjoint Set Union : Union Find O(ln(N))

```
const int N = 1000; // Maximum number of elements
int par[N]; // parent of each element
int siz[N]; // the size of each set
void initialize(int n) {
    iota(par, par + n, 0); // Set each element's parent to itself
    fill(siz, siz + n, 1); // Initialize the size of each set to 1
}
int find(int u) {
    if (par[u] == u) return u;
    return par[u] = find(par[u]);
}
bool union(int u, int v) {
    u = find(u);
    v = find(v);
    if (u == v) return false;
    if (siz[u] < siz[v]) swap(u, v);
    par[v] = u;
    siz[u] += siz[v];
    return true;
}
```

2.1.4 Fast Exponentiation O(ln(n))

```
int fastexp(int x, int n) {
    if (n == 0)
        return 1;
    int y = fastexp(x, n / 2);
    return (n & 1? x : 1) * y * y;
}
```

2.1.5 Sparse Table O(nlogn) implementation O(1) query

```
    SparseTable(const vector<int>& input) {
        n = input.size();
        int maxLog = log2(n) + 1;
        st.assign(n, vector<int>(maxLog));
        for (int i = 0; i < n; ++i) {
            st[i][0] = input[i];
        }
        for (int j = 1; (1 << j) <= n; ++j) {
            for (int i = 0; (i + (1 << j) - 1) < n; ++i) {
                st[i][j] = min(st[i][j-1], st[i + (1 << (j-1))][j-1]);
            }
        }
    }
    int query(int l, int r) {
        int j = log2(r - l + 1);
        return min(st[l][j], st[r - (1 << j) + 1][j]);
    }
```

2.1.6 Built in data structures

```
// 0-1 BFS O(E) like 0 BFS
vector<int> d(n, INF);
d[s] = 0;
deque<int> q;
q.push_front(s);
while (!q.empty()) {
    int v = q.front();
    q.pop_front();
    for (auto edge : adj[v]) {
        int u = edge.first;
        int w = edge.second;
        if (d[v] + w < d[u]) {
            d[u] = d[v] + w;
            if (w == 1)
                q.push_back(u);
            else
                q.push_front(u);
        }
    }
}
// Unordered Sets
uses hashing : insertions, deletions, and searches in O(1)
insert(x) erase(x) count(x), which returns 1 if the set contains x
// Ordered Sets
Insertions, deletions ,and searches require O(log n)
begin(), which returns an iterator to the lowest element in
the set, end(), which returns an iterator to the highest element in the set, lower_bound,
which returns an iterator to the least element greater than or equal to some element k,
and upper_bound, which returns an iterator to the least element strictly greater than some
element k.
The primary limitation of the ordered set is that we cant efficiently access the kth
largestelement in the set, or find the number of elements in the set greater than some x
set<int, greater<int>> v; store a set in reverse order
// Unordered Maps
count(key) which is either one or zero
erase(key)
O(1) with hashing
// Ordered Maps
lower_bound and upper_bound, returning the iterator pointing to the lowest entry not less
than the key, and the iterator pointing to the lowest entry strictly greater than the key.
// Multisets
sorted set that allows multiple copies of the same element.
count() O(log n + f) where f is the number of occurrences
begin(), end(), lower_bound(), and upper_bound()
To remove a value once, use ms.erase(ms.find(val)).
To remove all occurrences of a value, use ms.erase(val).
```

2.1.7 Prefix Sums

Process Q queries to find the sum of the elements between two indices a and b, app : Max Subarray Sum

2.1.8 Sorting

Comparators :
If object x is less than object y, return true
If object x is greater than or equal to object y, return false

```
bool cmp(const Edge &x, const Edge &y) { return x.w < y.w; }
sort(begin(v), end(v), cmp); (v vector)
or use sort(begin(v), end(v), [](const Edge &x, const Edge &y) { return x.w < y.w; });
```

2.2 Utilities

```
c - '0'; converts char to int
int i = stoi(word); string to int
//range over subsets
for (int i = 0; i < (1<<n); ++i) {
    for (int j = 0; j < n; ++j) {
        if (i & (1 << j)) { // j is in i }
    }
}
//permutations
void search() {
if (permutation.size() == n) {
// process permutation
} else {
for (int i = 0; i < n; i++) {
if (chosen[i]) continue;
chosen[i] = true;
permutation.push_back(i);
search();
chosen[i] = false;
permutation.pop_back();
}}}
```