1 Setup

1.1 header.h

```
#pragma once  // Delete this when copying this file
#include <bits/stdc++.h>
using namespace std;

#define ll long long
#define ull unsigned ll
#define ld long double
#define pl pair<ll, ll>
#define pi pair<int, int>  // use pl where possible/necessary
#define vl vector<ll>
#define vi vector<int>  // change to vl where possible/necessary
#define vb vector<bool>
#define vvi vector<vi>
#define vvl vector<vl>
#define vpl vector<pl>
```

```
#define vpi vector<pi>
#define vld vector<ld>
#define vvpi vector<vpi>
#define in_fast(el, cont) (cont.find(el) != cont.end())
#define in(el, cont) (find(cont.begin(), cont.end(), el) != cont.end())
#define all(x) x.begin(), x.end()
#define rall(x) x.rbegin(), x.rend()

constexpr int INF  = 2000000010;
constexpr ll LLINF = 9000000000000000010LL;

// int main() {
//   ios::sync_with_stdio(false);  // do not use cout + printf
//   cin.tie(NULL);
//   cout << fixed << setprecision(12);
//   return 0;
// }
```

1.2 Bash for c++ compile with header.h

```
#!/bin/bash
if [ $# -ne 1 ];then echo "Usage: $0 <input_file>"; exit 1;fi
f="$1";d=code/;o=a.out
[ -f $d/$f ] || { echo "Input file not found: $f"; exit 1; }
g++ -I$d $d/$f -o $o && echo "Compilation successful. Executable '$o' created." || echo "Compilation failed."
```

1.3 Bash for run tests c++

```
g++ $1/$1.cpp -o $1/$1.out
for file in $1/*.in; do diff <($1/$1.out < "$file") "${file%.in}.ans"; done
```

1.4 Bash for run tests python

```
for file in $1/*.in; do diff <(python3 $1/$1.py < "$file") "${file%.in}.ans"; done
```

1.4.1 Aux. helper C++

```cpp
#include "header.h"

int main() {
    // Read in a line including white space
    string line;
    getline(cin, line);
    // When doing the above read numbers as follows:
    int n;
    getline(cin, line);
    stringstream ss(line);
    ss >> n;

    // Count the number of 1s in binary represnatation of a number
    ull number;
    __builtin_popcountll(number);
}

// __int128
using lll = __int128;
ostream& operator<<( ostream& o, __int128 n ) {
  auto t = n<0 ? -n : n; char b[128], *d = end(b);
  do *--d = '0'+t%10, t /= 10; while (t);
  if(n<0) *--d = '-';
  o.rdbuf()->sputn(d,end(b)-d);
  return o;
}
```

1.4.2 Aux. helper python

```python
from functools import lru_cache

# Read until EOF
while True:
    try:
        pattern = input()
    except EOFError:
        break

@lru_cache(maxsize=None)
def smth_memoi(i, j, s):
    # Example in-built cache
    return "sol"

# Fast I
import io, os
def fast_io():
    finput = io.BytesIO(os.read(0,
        os.fstat(0).st_size)).readline
    s = finput().decode()
    return s

# Fast O
import sys
def fast_out():
    n = 5
    sys.stdout.write(str(n)+"\n")
```

2 Python

2.1 Graphs

2.1.1 BFS

```python
from collections import deque
def bfs(g, roots, n):
    q = deque(roots)
    explored = set()
    distances = [0 if v in roots else float('inf') for v in range(n)]

    while len(q) != 0:
        node = q.popleft()
        if node in explored: continue
        explored.add(node)
        for neigh in g[node]:
            if neigh not in explored:
                q.append(neigh)
                distances[neigh] = distances[node] + 1
    return distances
```

2.1.2 Dijkstra

```python
from heapq import *
def dijkstra(n, root, g):  # g = {node: (cost, neigh)}
  dist = [float("inf")]*n
  dist[root] = 0
  prev = [-1]*n

  pq = [(0, root)]
  heapify(pq)
  visited = set([])

  while len(pq) != 0:
    _, node = heappop(pq)

    if node in visited: continue
    visited.add(node)

    # In case of disconnected graphs
```

```python
    if node not in g:
      continue

    for cost, neigh in g[node]:
      alt = dist[node] + cost
      if alt < dist[neigh]:
        dist[neigh] = alt
        prev[neigh] = node
        heappush(pq, (alt, neigh))
  return dist
```

2.1.3 Topological Sort

```python
#Python program to print topological sorting of a DAG
from collections import defaultdict

#Class to represent a graph
class Graph:
    def __init__(self,vertices):
        self.graph = defaultdict(list) # dictionary containing adjacency List
        self.V = vertices #No. of vertices

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A recursive function used by topologicalSort
    def topologicalSortUtil(self,v,visited,stack):

        # Mark the current node as visited.
        visited[v] = True

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Push current vertex to stack which stores result
        stack.insert(0,v)

    # The function to do Topological Sort. It uses recursive
    # topologicalSortUtil()
    def topologicalSort(self):
        # Mark all the vertices as not visited
        visited = [False]*self.V
        stack =[]

        # Call the recursive helper function to store Topological
        # Sort starting from all vertices one by one
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i,visited,stack)

        # Print contents of stack
        return stack

    def isCyclicUtil(self, v, visited, recStack):

        # Mark current node as visited and
        # adds to recursion stack
        visited[v] = True
        recStack[v] = True

        # Recur for all neighbours
        # if any neighbour is visited and in
        # recStack then graph is cyclic
        for neighbour in self.graph[v]:
            if visited[neighbour] == False:
                if self.isCyclicUtil(neighbour, visited, recStack) == True:
                    return True
            elif recStack[neighbour] == True:
                return True

        # The node needs to be popped from
        # recursion stack before function ends
        recStack[v] = False
        return False

    # Returns true if graph is cyclic else false
    def isCyclic(self):
        visited = [False] * (self.V + 1)
        recStack = [False] * (self.V + 1)
        for node in range(self.V):
            if visited[node] == False:
                if self.isCyclicUtil(node, visited, recStack) == True:
                    return True
        return False
```

2.1.4 Kruskal (UnionFind)

```python
class UnionFind:
    def __init__(self, n):
        self.parent = [-1]*n

    def find(self, x):
        if self.parent[x] < 0:
            return x
        self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def connect(self, a, b):
        ra = self.find(a)
        rb = self.find(b)
        if ra == rb:
            return False
        if self.parent[ra] > self.parent[rb]:
            self.parent[rb] += self.parent[ra]
            self.parent[ra] = rb
        else:
            self.parent[ra] += self.parent[rb]
            self.parent[rb] = ra
        return True

# Full MST is len(spanning==n-1)
def kruskal(n, edges):
    uf = UnionFind(n)
    spanning = []
    edges.sort(key = lambda d: -d[2])
    while edges and len(spanning) < n-1:
        u, v, w = edges.pop()
        if not uf.connect(u, v):
            continue
        spanning.append((u, v, w))
    return spanning

# Example
edges = [(1, 2, 10), (2, 3, 20)]
```

2.2 Num. Th. / Comb.

2.2.1 nCk % prime

```python
# Note: p must be prime and k < p
def fermat_binom(n, k, p):
    if k > n:
        return 0
    # calculate numerator
    num = 1
    for i in range(n-k+1, n+1):
        num *= i % p
    num %= p
    # calculate denominator
    denom = 1
    for i in range(1,k+1):
        denom *= i % p
    denom %= p
    # numerator * denominator^(p-2) (mod p)
    return (num * pow(denom, p-2, p)) % p
```

2.2.2 Sieve of E. $O(n)$ so actually faster than C++ version, but more memory

```python
MAX_SIZE = 10**8+1
isprime = [True] * MAX_SIZE
prime = []
SPF = [None] * (MAX_SIZE)

def manipulated_seive(N):  # Up to N (not included)
  isprime[0] = isprime[1] = False
  for i in range(2, N):
    if isprime[i] == True:
      prime.append(i)
      SPF[i] = i
    j = 0
    while (j < len(prime) and
      i * prime[j] < N and
        prime[j] <= SPF[i]):
      isprime[i * prime[j]] = False
      SPF[i * prime[j]] = prime[j]
      j += 1
```

2.3 Strings

2.3.1 LCS

```python
def longestCommonSubsequence(text1, text2):  # O(m*n) time, O(m) space
    n = len(text1)
    m = len(text2)

    # Initializing two lists of size m
    prev = [0] * (m + 1)
    cur = [0] * (m + 1)

    for idx1 in range(1, n + 1):
        for idx2 in range(1, m + 1):
            # If characters are matching
            if text1[idx1 - 1] == text2[idx2 - 1]:
                cur[idx2] = 1 + prev[idx2 - 1]
            else:
                # If characters are not matching
                cur[idx2] = max(cur[idx2 - 1], prev[idx2])

        prev = cur.copy()

    return cur[m]
```

2.3.2 KMP

```python
class KMP:
    def partial(self, pattern):
        """ Calculate partial match table: String -> [Int]"""
        ret = [0]
        for i in range(1, len(pattern)):
            j = ret[i - 1]
            while j > 0 and pattern[j] != pattern[i]: j = ret[j - 1]
            ret.append(j + 1 if pattern[j] == pattern[i] else j)
        return ret

    def search(self, T, P):
        """KMP search main algorithm: String -> String -> [Int]
        Return all the matching position of pattern string P in T"""
        partial, ret, j = self.partial(P), [], 0
        for i in range(len(T)):
            while j > 0 and T[i] != P[j]: j = partial[j - 1]
            if T[i] == P[j]: j += 1
            if j == len(P):
                ret.append(i - (j - 1))
                j = partial[j - 1]
        return ret
```

2.3.3 Edit distance

```python
def editDistance(str1, str2):
  # Get the lengths of the input strings
  m = len(str1)
  n = len(str2)

  # Initialize a list to store the current row
  curr = [0] * (n + 1)

  # Initialize the first row with values from 0 to n
  for j in range(n + 1):
    curr[j] = j

  # Initialize a variable to store the previous value
  previous = 0

  # Loop through the rows of the dynamic programming matrix
  for i in range(1, m + 1):
    # Store the current value at the beginning of the row
    previous = curr[0]
    curr[0] = i

    # Loop through the columns of the dynamic programming matrix
    for j in range(1, n + 1):
      # Store the current value in a temporary variable
      temp = curr[j]

      # Check if the characters at the current positions in str1 and str2 are the same
      if str1[i - 1] == str2[j - 1]:
        curr[j] = previous
      else:
        # Update the current cell with the minimum of the three adjacent cells
        curr[j] = 1 + min(previous, curr[j - 1], curr[j])

      # Update the previous variable with the temporary value
      previous = temp

  # The value in the last cell represents the minimum number of operations
  return curr[n]
```

2.3.4 Bitstring Slower than a set for many elements, but hashable

```python
def add_element(bit_string, index):
    return bit_string | (1 << index)

def remove_element(bit_string, index):
    return bit_string & ~(1 << index)

def contains_element(bit_string, index):
    return (bit_string & (1 << index)) != 0
```

2.4 Other Algorithms

2.4.1 Rotate matrix

```python
def rotate_matrix(m):
    return [[m[j][i] for j in range(len(m))] for i in range(len(m[0])-1,-1,-1)]
```

2.5 Geometry

2.5.1 Convex Hull

```
def vec(a,b):
    return (b[0]-a[0],b[1]-a[1])
def det(a,b):
    return a[0]*b[1] - b[0]*a[1]

def convexhull(P):
    if (len(P) == 1):
        return [(p[0][0], p[0][1])]

    h = sorted(P)
    lower = []
    i = 0
    while i < len(h):
        if len(lower) > 1:
            a = vec(lower[-2], lower[-1])
            b = vec(lower[-1], h[i])
            if det(a,b) <= 0 and len(lower) > 1:
                lower.pop()
                continue
        lower.append(h[i])
        i += 1

    upper = []
    i = 0
    while i < len(h):
        if len(upper) > 1:
            a = vec(upper[-2], upper[-1])
            b = vec(upper[-1], h[i])
            if det(a,b) >= 0:
                upper.pop()
                continue
        upper.append(h[i])
        i += 1

    reversedupper = list(reversed(upper[1:-1:]))
    reversedupper.extend(lower)
    return reversedupper
```

2.5.2 Geometry

```

def vec(a,b):
    return (b[0]-a[0],b[1]-a[1])

def det(a,b):
    return a[0]*b[1] - b[0]*a[1]

    lower = []
    i = 0
    while i < len(h):
        if len(lower) > 1:
            a = vec(lower[-2], lower[-1])
            b = vec(lower[-1], h[i])
            if det(a,b) <= 0 and len(lower) > 1:
                lower.pop()
                continue
        lower.append(h[i])
        i += 1

    # find upper hull
    # det <= 0 -> replace
    upper = []
    i = 0
    while i < len(h):
        if len(upper) > 1:
            a = vec(upper[-2], upper[-1])
            b = vec(upper[-1], h[i])
            if det(a,b) >= 0:
                upper.pop()
                continue
        upper.append(h[i])
        i += 1
```

2.6 Other Data Structures

2.6.1 Segment Tree

```
N = 100000  # limit for array size
tree = [0] * (2 * N)  # Max size of tree

def build(arr, n):  # function to build the tree
    # insert leaf nodes in tree
    for i in range(n):
        tree[n + i] = arr[i]

    # build the tree by calculating parents
    for i in range(n - 1, 0, -1):
        tree[i] = tree[i << 1] + tree[i << 1 | 1]

def updateTreeNode(p, value, n):  # function to update a tree node
    # set value at position p
    tree[p + n] = value
    p = p + n

    i = p  # move upward and update parents
    while i > 1:
        tree[i >> 1] = tree[i] + tree[i ^ 1]
        i >>= 1

def query(l, r, n):  # function to get sum on interval [l, r)
    res = 0
    # loop to find the sum in the range
    l += n
    r += n
    while l < r:
        if l & 1:
            res += tree[l]
            l += 1
        if r & 1:
            r -= 1
            res += tree[r]
        l >>= 1
        r >>= 1
    return res
```

2.6.2 Trie

```
class TrieNode:
    def __init__(self):
        self.children = [None]*26
        self.isEndOfWord = False

class Trie:
    def __init__(self):
        self.root = self.getNode()

    def getNode(self):
        return TrieNode()

    def _charToIndex(self,ch):
        return ord(ch)-ord('a')


    def insert(self,key):
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])
            if not pCrawl.children[index]:
                pCrawl.children[index] = self.getNode()
            pCrawl = pCrawl.children[index]
        pCrawl.isEndOfWord = True

    def search(self, key):
        pCrawl = self.root
        length = len(key)
        for level in range(length):
            index = self._charToIndex(key[level])
            if not pCrawl.children[index]:
                return False
            pCrawl = pCrawl.children[index]

        return pCrawl.isEndOfWord
```

3 C++

3.1 Graphs

3.1.1 BFS

```cpp
#include "header.h"
#define graph unordered_map<ll, unordered_set<ll>>
vi bfs(int n, graph& g, vi& roots) {
    vi parents(n+1, -1); // nodes are 1..n
    unordered_set<int> visited;
    queue<int> q;
    for (auto x: roots) {
        q.emplace(x);
        visited.insert(x);
    }
    while (not q.empty()) {
        int node = q.front();
        q.pop();

        for (auto neigh: g[node]) {
            if (not in(neigh, visited)) {
                parents[neigh] = node;
                q.emplace(neigh);
                visited.insert(neigh);
            }
        }
    }
    return parents;
}
vi reconstruct_path(vi parents, int start, int goal) {
    vi path;
    int curr = goal;
    while (curr != start) {
        path.push_back(curr);
        if (parents[curr] == -1) return vi(); // No path, empty vi
        curr = parents[curr];
    }
    path.push_back(start);
    reverse(path.begin(), path.end());
    return path;
}
```

3.1.2 DFS Cycle detection / removal

```cpp
#include "header.h"
void removeCyc(ll node, unordered_map<ll, vector<pair<ll, ll>>>& neighs, vector<bool>& visited,
vector<bool>& recStack, vector<ll>& ans) {
    if (!visited[node]) {
        visited[node] = true;
        recStack[node] = true;
        auto it = neighs.find(node);
        if (it != neighs.end()) {
            for (auto util: it->second) {
                ll nnode = util.first;
                if (recStack[nnode]) {
                    ans.push_back(util.second);
                } else if (!visited[nnode]) {
                    removeCyc(nnode, neighs,
                        visited, recStack, ans);
                }
            }
        }
    }
    recStack[node] = false;
}
```

3.1.3 Dijkstra

```cpp
#include "header.h"
vector<int> dijkstra(int n, int root, map<int, vector<pair<int, int>>>& g) {
  unordered_set<int> visited;
  vector<int> dist(n, INF);
    priority_queue<pair<int, int>> pq;
    dist[root] = 0;
    pq.push({0, root});
    while (!pq.empty()) {
        int node = pq.top().second;
        int d = -pq.top().first;
        pq.pop();

        if (in(node, visited)) continue;
        visited.insert(node);

        for (auto e : g[node]) {
            int neigh = e.first;
            int cost = e.second;
            if (dist[neigh] > dist[node] + cost)
            {
                dist[neigh] = dist[node] + cost;
                pq.push({-dist[neigh], neigh});
            }
        }
    }
    return dist;
}
```

3.1.4 Floyd-Warshall

```cpp
#include "header.h"
// g[i][j] = infty if not path from i to j
// if g[i][i] < 0, i is contained in a negative cycle
void warshall(vvl g) {
    for (int i=0; i<g.size(); ++i) {
        for (int j=0; j<g.size(); ++j) {
            for (int k=0; k<g.size(); ++k) {
                if (g[i][k] < LLINF and g[k][j] < LLINF and g[i][j] > g[i][k] + g[k][j]) {
                    g[i][j] = g[i][k] + g[k][j];
}}}}}
```

3.1.5 Kruskal Minimum spanning tree of undirected weighted graph

```cpp
#include "header.h"
#include "disjoint_set.h"
// O(E log E)
pair<set<pair<ll, ll>>, ll>  kruskal(vector<tuple<ll, ll, ll>>& edges, ll n) {
    set<pair<ll, ll>> ans;
    ll cost = 0;

    sort(edges.begin(), edges.end());
    DisjointSet<ll> fs(n);

    ll dist, i, j;
    for (auto edge: edges) {
        dist = get<0>(edge);
        i = get<1>(edge);
        j = get<2>(edge);

        if (fs.find_set(i) != fs.find_set(j)) {
            fs.union_sets(i, j);
            ans.insert({i, j});
            cost += dist;
        }
    }
    return pair<set<pair<ll, ll>>, ll> {ans, cost};
}
```

3.1.6 Hungarian algorithm

```cpp
#include "header.h"

template <class T> bool ckmin(T &a, const T &b) {
    return b < a ? a = b, 1 : 0; }
/**
 * Given J jobs and W workers (J <= W), computes the minimum cost to assign each
 * prefix of jobs to distinct workers.
 * @tparam T a type large enough to represent integers on the order of J *
 * max(|C|)
```

```
 * @param C a matrix of dimensions JxW such that C[j][w] = cost to assign j-th
 * job to w-th worker (possibly negative)
 *
 * @return a vector of length J, with the j-th entry equaling the minimum cost
 * to assign the first (j+1) jobs to distinct workers
 */
template <class T> vector<T> hungarian(const vector<vector<T>> &C) {
    const int J = (int)size(C), W = (int)size(C[0]);
    assert(J <= W);
    // job[w] = job assigned to w-th worker, or -1 if no job assigned
    // note: a W-th worker was added for convenience
    vector<int> job(W + 1, -1);
    vector<T> ys(J), yt(W + 1);  // potentials
    // -yt[W] will equal the sum of all deltas
    vector<T> answers;
    const T inf = numeric_limits<T>::max();
    for (int j_cur = 0; j_cur < J; ++j_cur) {  // assign j_cur-th job
        int w_cur = W;
        job[w_cur] = j_cur;
        // min reduced cost over edges from Z to worker w
        vector<T> min_to(W + 1, inf);
        vector<int> prv(W + 1, -1);  // previous worker on alternating path
        vector<bool> in_Z(W + 1);    // whether worker is in Z
        while (job[w_cur] != -1) {   // runs at most j_cur + 1 times
            in_Z[w_cur] = true;
            const int j = job[w_cur];
            T delta = inf;
            int w_next;
            for (int w = 0; w < W; ++w) {
                if (!in_Z[w]) {
                    if (ckmin(min_to[w], C[j][w] - ys[j] - yt[w]))
                        prv[w] = w_cur;
                    if (ckmin(delta, min_to[w])) w_next = w;
                }
            }
            // delta will always be non-negative,
            // except possibly during the first time this loop runs
            // if any entries of C[j_cur] are negative
            for (int w = 0; w <= W; ++w) {
                if (in_Z[w]) ys[job[w]] += delta, yt[w] -= delta;
                else min_to[w] -= delta;
            }
            w_cur = w_next;
        }
        // update assignments along alternating path
        for (int w; w_cur != W; w_cur = w) job[w_cur] = job[w = prv[w_cur]];
        answers.push_back(-yt[W]);
    }
    return answers;
}
```

3.1.7 Suc. shortest path Calculates max flow, min cost

```
#include "header.h"
// map<node, map<node, pair<cost, capacity>>>
#define graph unordered_map<int, unordered_map<int, pair<ld, int>>>
graph g;
const ld infty = 1e60l;  // Change if necessary
ld fill(int n, vld& potential) {  // Finds max flow, min cost
  priority_queue<pair<ld, int>> pq;
  vector<bool> visited(n+2, false);
  vi parent(n+2, 0);
  vld dist(n+2, infty);
  dist[0] = 0.l;
  pq.emplace(make_pair(0.l, 0));
  while (not pq.empty()) {
    int node = pq.top().second;
    pq.pop();
    if (visited[node]) continue;
    visited[node] = true;
    for (auto& x : g[node]) {
      int neigh = x.first;
      int capacity = x.second.second;
      ld cost = x.second.first;
      if (capacity and not visited[neigh]) {
        ld d = dist[node] + cost + potential[node] - potential[neigh];
        if (d + 1e-10l < dist[neigh]) {
          dist[neigh] = d;
          pq.emplace(make_pair(-d, neigh));
          parent[neigh] = node;
  }}}}

  for (int i = 0; i < n+2; i++) {
    potential[i] = min(infty, potential[i] + dist[i]);
  }
  if (not parent[n+1]) return infty;
  ld ans = 0.l;
  for (int x = n+1; x; x=parent[x]) {
    ans += g[parent[x]][x].first;
    g[parent[x]][x].second--;
    g[x][parent[x]].second++;
  }
  return ans;
}
```

3.1.8 Bipartite check

```
#include "header.h"
int main() {
    int n;
    vvi adj(n);

    vi side(n, -1);    // will have 0's for one side 1's for other side
    bool is_bipartite = true;   // becomes false if not bipartite
    queue<int> q;
    for (int st = 0; st < n; ++st) {
        if (side[st] == -1) {
            q.push(st);
            side[st] = 0;
            while (!q.empty()) {
                int v = q.front();
                q.pop();
                for (int u : adj[v]) {
                    if (side[u] == -1) {
                        side[u] = side[v] ^ 1;
                        q.push(u);
                    } else {
                        is_bipartite &= side[u] != side[v];
                    }
}}}}}
```

3.1.9 Find cycle directed

```
#include "header.h"
int n;
const int mxN = 2e5+5;
vvi adj(mxN);
vector<char> color;
vi parent;
int cycle_start, cycle_end;
bool dfs(int v) {
    color[v] = 1;
    for (int u : adj[v]) {
        if (color[u] == 0) {
            parent[u] = v;
```

```
            if (dfs(u)) return true;
        } else if (color[u] == 1) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
    }
    color[v] = 2;
    return false;
}
void find_cycle() {
    color.assign(n, 0);
    parent.assign(n, -1);
    cycle_start = -1;
    for (int v = 0; v < n; v++) {
        if (color[v] == 0 && dfs(v))break;
    }
    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        reverse(cycle.begin(), cycle.end());

        cout << "Cycle␣Found:␣";
        for (int v : cycle) cout << v << "␣";
        cout << endl;
    }
}
```

3.1.10 Find cycle undirected

```
#include "header.h"
int n;
const int mxN = 2e5 + 5;
vvi adj(mxN);
vector<bool> visited;
vi parent;
int cycle_start, cycle_end;
bool dfs(int v, int par) { // passing vertex and its parent vertex
    visited[v] = true;
    for (int u : adj[v]) {
        if(u == par) continue; // skipping edge to parent vertex
        if (visited[u]) {
            cycle_end = v;
            cycle_start = u;
            return true;
        }
        parent[u] = v;
        if (dfs(u, parent[u]))
            return true;
    }
    return false;
}
void find_cycle() {
    visited.assign(n, false);
    parent.assign(n, -1);
    cycle_start = -1;
    for (int v = 0; v < n; v++) {
        if (!visited[v] && dfs(v, parent[v]))
            break;
    }
    if (cycle_start == -1) {
        cout << "Acyclic" << endl;
    } else {
        vector<int> cycle;
        cycle.push_back(cycle_start);
        for (int v = cycle_end; v != cycle_start; v = parent[v])
            cycle.push_back(v);
        cycle.push_back(cycle_start);
        cout << "Cycle␣Found:␣";
        for (int v : cycle) cout << v << "␣";
        cout << endl;
    }
}
```

3.1.11 Tarjan's SCC

```
#include "header.h"

struct Tarjan {
  vvi &edges;
  int V, counter = 0, C = 0;
  vi n, l;
  vector<bool> vs;
  stack<int> st;
  Tarjan(vvi &e) : edges(e), V(e.size()), n(V, -1), l(V, -1), vs(V, false) {}
  void visit(int u, vi &com) {
    l[u] = n[u] = counter++;
    st.push(u);
    vs[u] = true;
    for (auto &&v : edges[u]) {
      if (n[v] == -1) visit(v, com);
      if (vs[v]) l[u] = min(l[u], l[v]);
    }
    if (l[u] == n[u]) {
      while (true) {
        int v = st.top();
        st.pop();
        vs[v] = false;
        com[v] = C;  //<== ACT HERE
        if (u == v) break;
      }
      C++;
    }
  }
  int find_sccs(vi &com) {  // component indices will be stored in 'com'
    com.assign(V, -1);
    C = 0;
    for (int u = 0; u < V; ++u)
      if (n[u] == -1) visit(u, com);
    return C;
  }
  // scc is a map of the original vertices of the graph to the vertices
  // of the SCC graph, scc_graph is its adjacency list.
  // SCC indices and edges are stored in 'scc' and 'scc_graph'.
  void scc_collapse(vi &scc, vvi &scc_graph) {
    find_sccs(scc);
    scc_graph.assign(C, vi());
    set<pi> rec;  // recorded edges
    for (int u = 0; u < V; ++u) {
      assert(scc[u] != -1);
      for (int v : edges[u]) {
        if (scc[v] == scc[u] ||
          rec.find({scc[u], scc[v]}) != rec.end()) continue;
        scc_graph[scc[u]].push_back(scc[v]);
        rec.insert({scc[u], scc[v]});
      }
    }
  }
  // Function to find sources and sinks in the SCC graph
  // The number of edges needed to be added is max(sources.size(), sinks.())
  void findSourcesAndSinks(const vvi &scc_graph, vi &sources, vi &sinks) {
    vi in_degree(C, 0), out_degree(C, 0);
    for (int u = 0; u < C; u++) {
      for (auto v : scc_graph[u]) {
        in_degree[v]++;
        out_degree[u]++;
      }
    }
    for (int i = 0; i < C; ++i) {
      if (in_degree[i] == 0) sources.push_back(i);
      if (out_degree[i] == 0) sinks.push_back(i);
    }
  }
};
```

3.1.12 SCC edges

Prints out the missing edges to make the input digraph strongly connected

```
#include "header.h"
const int N=1e5+10;
int n,a[N],cnt[N],vis[N];
vector<int> hd,tl;
int dfs(int x){
    vis[x]=1;
    if(!vis[a[x]])return vis[x]=dfs(a[x]);
    return vis[x]=x;
}
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i]);
        cnt[a[i]]++;
    }
    int k=0;
    for(int i=1;i<=n;i++){
        if(!cnt[i]){
            k++;
            hd.push_back(i);
            tl.push_back(dfs(i));
        }
    }
    int tk=k;
    for(int i=1;i<=n;i++){
        if(!vis[i]){
            k++;
            hd.push_back(i);
            tl.push_back(dfs(i));
        }
    }
    if(k==1&&!tk)k=0;
    printf("%d\n",k);
    for(int i=0;i<k;i++)printf("%d %d\n",tl[i],hd
        [(i+1)%k]);
    return 0;
}
```

3.1.13 Find Bridges

```
#include "header.h"
int n; // number of nodes
vvi adj; // adjacency list of graph
vector<bool> visited;
vi tin, low;
int timer;
void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] > tin[v])
                IS_BRIDGE(v, to);
        }
    }
}
void find_bridges() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs(i);
    }
}
```

3.1.14 Articulation points

(i.e. cut off points)

```
#include "header.h"
int n; // number of nodes
vvi adj; // adjacency list of graph
vector<bool> visited;
vi tin, low;
int timer;
void dfs(int v, int p = -1) {
    visited[v] = true;
    tin[v] = low[v] = timer++;
    int children=0;
    for (int to : adj[v]) {
        if (to == p) continue;
        if (visited[to]) {
            low[v] = min(low[v], tin[to]);
        } else {
            dfs(to, v);
            low[v] = min(low[v], low[to]);
            if (low[to] >= tin[v] && p!=-1)
                IS_CUTPOINT(v);
            ++children;
        }
    }
    if(p == -1 && children > 1)
        IS_CUTPOINT(v);
}
void find_cutpoints() {
    timer = 0;
    visited.assign(n, false);
    tin.assign(n, -1);
    low.assign(n, -1);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs (i);
    }
}
```

3.1.15 Topological sort

```
#include "header.h"
int n; // number of vertices
vvi adj; // adjacency list of graph
vector<bool> visited;
vi ans;
void dfs(int v) {
    visited[v] = true;
    for (int u : adj[v]) {
        if (!visited[u]) dfs(u);
    }
    ans.push_back(v);
}
void topological_sort() {
    visited.assign(n, false);
    ans.clear();
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) dfs(i);
    }
    reverse(ans.begin(), ans.end());
}
```

3.1.16 Bellmann-Ford

Same as Dijkstra but allows neg. edges

```
#include "header.h"
// Switch vi and vvpi to vl and vvpl if necessary
void bellmann_ford_extended(vvpi &e, int source,
    vi &dist, vb &cyc) {
  dist.assign(e.size(), INF);
  cyc.assign(e.size(), false); // true when u is
      in a <0 cycle
  dist[source] = 0;
  for (int iter = 0; iter < e.size() - 1; ++iter)
      {
    bool relax = false;
    for (int u = 0; u < e.size(); ++u)
      if (dist[u] == INF) continue;
      else for (auto &e : e[u])
        if(dist[u]+e.second < dist[e.first])
          dist[e.first] = dist[u]+e.second, relax
              = true;
    if(!relax) break;
  }
  bool ch = true;
  while (ch) {        // keep going untill no
      more changes
    ch = false;       // set dist to -INF when in
        cycle
```

```
    for (int u = 0; u < e.size(); ++u)
      if (dist[u] == INF) continue;
      else for (auto &e : e[u])
        if (dist[e.first] > dist[u] + e.second
          && !cyc[e.first]) {
          dist[e.first] = -INF;
          ch = true;    //return true for cycle
              detection only
          cyc[e.first] = true;
        }
  }
}
```

3.1.17 Ford-Fulkerson Basic Max. flow

```
#include "header.h"
#define V 6  // Num. of vertices in given graph

/* Returns true if there is a path from source 's
    ' to sink
't' in residual graph. Also fills parent[] to
    store the
path */
bool bfs(int rGraph[V][V], int s, int t, int
    parent[]) {
  bool visited[V];
  memset(visited, 0, sizeof(visited));
  queue<int> q;
  q.push(s);
  visited[s] = true;
  parent[s] = -1;

  // Standard BFS Loop
  while (!q.empty()) {
    int u = q.front();
    q.pop();

    for (int v = 0; v < V; v++) {
      if (visited[v] == false && rGraph[u][v] >
          0) {
        if (v == t) {
          parent[v] = u;
          return true;
        }
        q.push(v);
        parent[v] = u;
        visited[v] = true;
      }
    }
  }
  return false;
}

// Returns the maximum flow from s to t in the
    given graph
int fordFulkerson(int graph[V][V], int s, int t)
    {
  int u, v;
  int rGraph[V]
      [V];
  for (u = 0; u < V; u++)
    for (v = 0; v < V; v++)
      rGraph[u][v] = graph[u][v];

  int parent[V]; // This array is filled by BFS
      and to
        // store path
  int max_flow = 0; // There is no flow initially
  while (bfs(rGraph, s, t, parent)) {
    int path_flow = INT_MAX;
    for (v = t; v != s; v = parent[v]) {
      u = parent[v];
      path_flow = min(path_flow, rGraph[u][v]);
    }

    for (v = t; v != s; v = parent[v]) {
      u = parent[v];
      rGraph[u][v] -= path_flow;
      rGraph[v][u] += path_flow;
    }
    max_flow += path_flow;
  }
  return max_flow;
}
```

3.1.18 Dinic max flow $O(V^2E)$, $O(Ef)$

```

using F = ll; using W = ll; // types for flow and
    weight/cost
struct S{
    const int v;            // neighbour
    const int r;    // index of the reverse edge
    F f;            // current flow
    const F cap;    // capacity
    const W cost;    // unit cost
    S(int v, int ri, F c, W cost = 0) :
        v(v), r(ri), f(0), cap(c), cost(cost) {}
    inline F res() const { return cap - f; }
};
struct FlowGraph : vector<vector<S>> {
    FlowGraph(size_t n) : vector<vector<S>>(n) {}
    void add_edge(int u, int v, F c, W cost = 0){
         auto &t = *this;
        t[u].emplace_back(v, t[v].size(), c, cost
            );
        t[v].emplace_back(u, t[u].size()-1, c, -
            cost);
    }
    void add_arc(int u, int v, F c, W cost = 0){
        auto &t = *this;
        t[u].emplace_back(v, t[v].size(), c, cost
            );
        t[v].emplace_back(u, t[u].size()-1, 0, -
            cost);
    }
    void clear() { for (auto &E : *this) for (
        auto &e : E) e.f = 0LL; }
};
struct Dinic{
    FlowGraph &edges; int V,s,t;
    vi l; vector<vector<S>::iterator> its; //
        levels and iterators
    Dinic(FlowGraph &edges, int s, int t) :
        edges(edges), V(edges.size()), s(s), t(t)
        , l(V,-1), its(V) {}
    ll augment(int u, F c) { // we reuse the same
         iterators
        if (u == t) return c; ll r = 0LL;
        for(auto &i = its[u]; i != edges[u].end()
        ; i++){
            auto &e = *i;
            if (e.res() && l[u] < l[e.v]) {
                auto d = augment(e.v, min(c, e.
                    res()));
                if (d > 0) { e.f += d; edges[e.v
                    ][e.r].f -= d; c -= d;
                    r += d; if (!c) break; }
        }    }
        return r;
    }
    ll run() {
        ll flow = 0, f;
        while(true) {
            fill(l.begin(), l.end(),-1); l[s]=0;
                // recalculate the layers
            queue<int> q; q.push(s);
            while(!q.empty()){
                auto u = q.front(); q.pop(); its[
                    u] = edges[u].begin();
                for(auto &&e : edges[u]) if(e.res
                    () && l[e.v]<0)
                    l[e.v] = l[u]+1, q.push(e.v);
            }
            if (l[t] < 0) return flow;
            while ((f = augment(s, INF)) > 0)
                flow += f;
        }    }
};
```

3.1.19 Edmonds-Karp Max flow $O(VE^2)$

```
/**
```

```
 * Description: Flow algorithm with guaranteed complexity $O(VE^2)$. To get edge flow values, compare
 * capacities before and after, and take the positive values only.
 */

template<class T> T edmondsKarp(vector<unordered_map<int, T>>&
  graph, int source, int sink) {
  assert(source != sink);
  T flow = 0;
  vi par(sz(graph)), q = par;

  for (;;) {
    fill(all(par), -1);
    par[source] = 0;
    int ptr = 1;
    q[0] = source;

    rep(i,0,ptr) {
      int x = q[i];
      for (auto e : graph[x]) {
        if (par[e.first] == -1 && e.second > 0) {
          par[e.first] = x;
          q[ptr++] = e.first;
          if (e.first == sink) goto out;
        }
      }
    }
    return flow;
out:
    T inc = numeric_limits<T>::max();
    for (int y = sink; y != source; y = par[y])
      inc = min(inc, graph[par[y]][y]);

    flow += inc;
    for (int y = sink; y != source; y = par[y]) {
      int p = par[y];
      if ((graph[p][y] -= inc) <= 0) graph[p].erase(y);
      graph[y][p] += inc;
    }
  }
}
```

3.2 Dynamic Programming

3.2.1 Longest Incr. Subseq.

```
#include "header.h"
template<class T>
vector<T> index_path_lis(vector<T>& nums) {
  int n = nums.size();
  vector<T> sub;
    vector<int> subIndex;
  vector<T> path(n, -1);
  for (int i = 0; i < n; ++i) {
     if (sub.empty() || sub[sub.size() - 1] < nums[i]) {
    path[i] = sub.empty() ? -1 : subIndex[sub.size() - 1];
    sub.push_back(nums[i]);
    subIndex.push_back(i);
     } else {
    int idx = lower_bound(sub.begin(), sub.end(), nums[i]) - sub.begin();
    path[i] = idx == 0 ? -1 : subIndex[idx - 1];
    sub[idx] = nums[i];
    subIndex[idx] = i;
     }
  }
  vector<T> ans;
  int t = subIndex[subIndex.size() - 1];
  while (t != -1) {
     ans.push_back(t);
     t = path[t];
  }
  reverse(ans.begin(), ans.end());
  return ans;
}
// Length only
template<class T>
int length_lis(vector<T> &a) {
  set<T> st;
  typename set<T>::iterator it;
  for (int i = 0; i < a.size(); ++i) {
    it = st.lower_bound(a[i]);
    if (it != st.end()) st.erase(it);
    st.insert(a[i]);
  }
  return st.size();
}
```

3.2.2 0-1 Knapsack

```
#include "header.h"
// given a number of coins, calculate all possible distinct sums
int main() {
  int n;
  vi coins(n);  // all possible coins to use
  int sum = 0;        // sum of the coins
  vi dp(sum + 1, 0);           // dp[x] = 1 if sum x can be made
  dp[0] = 1;                   // sum 0 can be made
  for (int c = 0; c < n; ++c)         // first iteration: sums with first
    for (int x = sum; x >= 0; --x)       // coin, next first 2 coins etc
      if (dp[x]) dp[x + coins[c]] = 1;  // if sum x valid, x+c valid
}
```

3.2.3 Coin change

Number of coins required to achieve a given value

```
#include "header.h"
// Returns total distinct ways to make sum using n coins of
// different denominations
int count(vi& coins, int n, int sum) {
    // 2d dp array where n is the number of coin
    // denominations and sum is the target sum
    vector<vector<int> > dp(n + 1, vector<int>(sum + 1, 0));
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {

            // without using the current coin,
            dp[i][j] += dp[i - 1][j];

            // using the current coin
            if ((j - coins[i - 1]) >= 0)
                dp[i][j] += dp[i][j - coins[i - 1]];
        }
    }
    return dp[n][sum];
}
```

3.3 Trees

3.3.1 Tree diameter

```
#include "header.h"
const int mxN = 2e5 + 5;
int n, d[mxN];  // distance array
vi adj[mxN];  // tree adjacency list
void dfs(int s, int e) {
  d[s] = 1 + d[e];     // recursively calculate the distance from the starting node to each node
  for (auto u : adj[s]) { // for each adjacent node
    if (u != e) dfs(u, s);  // don't move backwards in the tree
  }
}
```

```cpp
int main() {
  // read input, create adj list
  dfs(0, -1);                  // first dfs call to find farthest node from arbitrary node
  dfs(distance(d, max_element(d, d + n)), -1);  // second dfs call to find farthest node from that one
  cout << *max_element(d, d + n) - 1 << '\n';  // distance from second node to farthest is the diameter
}
```

3.3.2 Tree Node Count

```cpp
#include "header.h"
// calculate amount of nodes in each node's subtree
const int mxN = 2e5 + 5;
int n, cnt[mxN];
vi adj[mxN];
void dfs(int s = 0, int e = -1) {
  cnt[s] = 1;  // count leaves as one
  for (int u : adj[s]) {
    dfs(u, s);
    cnt[s] += cnt[u];  // add up nodes of the subtrees
  }
}
```

3.4 Numerical

3.4.1 Template (for this section)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define rep(i, a, b) for(int i = a; i < (b); ++i)
#define all(x) begin(x), end(x)
#define sz(x) (int)(x).size()
typedef long long ll;
typedef pair<int, int> pii;
typedef vector<int> vi;
```

3.4.2 Polynomial

```cpp
#include "template.cpp"

struct Poly {
  vector<double> a;
  double operator()(double x) const {
    double val = 0;
    for (int i = sz(a); i--;) (val *= x) += a[i];
    return val;
  }
  void diff() {
    rep(i,1,sz(a)) a[i-1] = i*a[i];
    a.pop_back();
  }
  void divroot(double x0) {
    double b = a.back(), c; a.back() = 0;
    for(int i=sz(a)-1; i--;) c = a[i], a[i] = a[i+1]*x0+b, b=c;
    a.pop_back();
  }
};
```

3.4.3 Poly Roots

```cpp
/**
 * Description: Finds the real roots to a polynomial.
 * Usage: polyRoots({{2,-3,1}},-1e9,1e9) // solve x^2-3x+2 = 0
 * Time: O(n^2 \log(1/\epsilon))
 */
#include "Polynomial.h"
#include "template.cpp"

vector<double> polyRoots(Poly p, double xmin, double xmax) {
  if (sz(p.a) == 2) { return {-p.a[0]/p.a[1]}; }
  vector<double> ret;
  Poly der = p;
  der.diff();
  auto dr = polyRoots(der, xmin, xmax);
  dr.push_back(xmin-1);
  dr.push_back(xmax+1);
  sort(all(dr));
  rep(i,0,sz(dr)-1) {
    double l = dr[i], h = dr[i+1];
    bool sign = p(l) > 0;
    if (sign ^ (p(h) > 0)) {
      rep(it,0,60) { // while (h - l > 1e-8)
        double m = (l + h) / 2, f = p(m);
        if ((f <= 0) ^ sign) l = m;
        else h = m;
      }
      ret.push_back((l + h) / 2);
    }
  }
  return ret;
}
```

3.4.4 Golden Section Search

```cpp
/**
 * Description: Finds the argument minimizing the function $f$ in the interval $[a,b]$
 * assuming $f$ is unimodal on the interval, i.e. has only one local minimum and no local
 * maximum. The maximum error in the result is $eps$. Works equally well for maximization
 * with a small change in the code. See TernarySearch.h in the Various chapter for a
 * discrete version.
 * Usage:
  double func(double x) { return 4+x+.3*x*x; }
  double xmin = gss(-1000,1000,func);
 * Time: O(\log((b-a) / \epsilon))
 */
#include "template.cpp"

/// It is important for r to be precise, otherwise we don't necessarily maintain the inequality a < x1 < x2 < b.
double gss(double a, double b, double (*f)(double)) {
  double r = (sqrt(5)-1)/2, eps = 1e-7;
  double x1 = b - r*(b-a), x2 = a + r*(b-a);
  double f1 = f(x1), f2 = f(x2);
  while (b-a > eps)
    if (f1 < f2) { //change to > to find maximum
      b = x2; x2 = x1; f2 = f1;
      x1 = b - r*(b-a); f1 = f(x1);
    } else {
      a = x1; x1 = x2; f1 = f2;
      x2 = a + r*(b-a); f2 = f(x2);
    }
  return a;
}
```

3.4.5 Hill Climbing

```cpp
/**
 * Description: Poor man's optimization for unimodal functions.
 */
#include "template.cpp"

typedef array<double, 2> P;

template<class F> pair<double, P> hillClimb(P start, F f) {
  pair<double, P> cur(f(start), start);
  for (double jmp = 1e9; jmp > 1e-20; jmp /= 2) {
    rep(j,0,100) rep(dx,-1,2) rep(dy,-1,2) {
      P p = cur.second;
      p[0] += dx*jmp;
      p[1] += dy*jmp;
```

```
      cur = min(cur, make_pair(f(p), p));
    }
  }
  return cur;
}
```

3.4.6 Integration

```
/**
 * Description: Simple integration of a function over an interval using
 *  Simpson's rule. The error should be proportional to $h^4$, although in
 *  practice you will want to verify that the result is stable to desired
 *  precision when epsilon changes.
 */
#include "template.cpp"

template<class F>
double quad(double a, double b, F f, const int n = 1000) {
  double h = (b - a) / 2 / n, v = f(a) + f(b);
  rep(i,1,n*2)
    v += f(a + i*h) * (i&1 ? 4 : 2);
  return v * h / 3;
}
```

3.4.7 Integration Adaptive

```
/**
 * Description: Fast integration using an adaptive Simpson's rule.
 * Usage:
  double sphereVolume = quad(-1, 1, [](double x) {
  return quad(-1, 1, [\&](double y) {
  return quad(-1, 1, [\&](double z) {
  return x*x + y*y + z*z < 1; });});});
 * Status: mostly untested
 */
#include "template.cpp"

typedef double d;
#define S(a,b) (f(a) + 4*f((a+b) / 2) + f(b)) * (b-a) / 6

template <class F>
d rec(F& f, d a, d b, d eps, d S) {
  d c = (a + b) / 2;
  d S1 = S(a, c), S2 = S(c, b), T = S1 + S2;
  if (abs(T - S) <= 15 * eps || b - a < 1e-10)
    return T + (T - S) / 15;
  return rec(f, a, c, eps / 2, S1) + rec(f, c, b, eps / 2, S2);
}
template<class F>
d quad(d a, d b, F f, d eps = 1e-8) {
  return rec(f, a, b, eps, S(a, b));
}
```

3.5 Num. Th. / Comb.

3.5.1 Basic stuff

```
#include "header.h"
ll gcd(ll a, ll b) { while (b) { a %= b; swap(a, b); } return a; }
ll lcm(ll a, ll b) { return (a / gcd(a, b)) * b; }
ll mod(ll a, ll b) { return ((a % b) + b) % b; }
// Finds x, y s.t. ax + by = d = gcd(a, b).
void extended_euclid(ll a, ll b, ll &x, ll &y, ll &d) {
  ll xx = y = 0;
  ll yy = x = 1;
  while (b) {
    ll q = a / b;
    ll t = b; b = a % b; a = t;
    t = xx; xx = x - q * xx; x = t;
    t = yy; yy = y - q * yy; y = t;
  }
  d = a;
}
// solves ab = 1 (mod n), -1 on failure
ll mod_inverse(ll a, ll n) {
  ll x, y, d;
  extended_euclid(a, n, x, y, d);
  return (d > 1 ? -1 : mod(x, n));
}
// All modular inverses of [1..n] mod P in O(n) time.
vi inverses(ll n, ll P) {
  vi I(n+1, 1LL);
  for (ll i = 2; i <= n; ++i)
    I[i] = mod(-(P/i) * I[P%i], P);
  return I;
}
// (a*b)%m
ll mulmod(ll a, ll b, ll m){
  ll x = 0, y=a%m;
  while(b>0){
    if(b&1) x = (x+y)%m;
    y = (2*y)%m, b /= 2;
  }
  return x % m;
}
// Finds b^e % m in O(lg n) time, ensure that b < m to avoid overflow!
ll powmod(ll b, ll e, ll m) {
  ll p = e<2 ? 1 : powmod((b*b)%m,e/2,m);
  return e&1 ? p*b%m : p;
}
// Solve ax + by = c, returns false on failure.
bool linear_diophantine(ll a, ll b, ll c, ll &x, ll &y) {
  ll d = gcd(a, b);
  if (c % d) {
    return false;
  } else {
    x = c / d * mod_inverse(a / d, b / d);
    y = (c - a * x) / b;
    return true;
  }
}

// Description: Tonelli-Shanks algorithm for modular square roots. Finds $x$ s.t. $x^2 = a \pmod p$ ($-x$ gives the other solution). O(\log^2 p) worst case, O(\log p) for most $p$
ll sqrtmod(ll a, ll p) {
  a %= p; if (a < 0) a += p;
  if (a == 0) return 0;
  assert(powmod(a, (p-1)/2, p) == 1); // else no solution
  if (p % 4 == 3) return powmod(a, (p+1)/4, p);
  // a^(n+3)/8 or 2^(n+3)/8 * 2^(n-1)/4 works if p % 8 == 5
  ll s = p - 1, n = 2;
  int r = 0, m;
  while (s % 2 == 0)
    ++r, s /= 2;
  /// find a non-square mod p
  while (powmod(n, (p - 1) / 2, p) != p - 1) ++n;
  ll x = powmod(a, (s + 1) / 2, p);
  ll b = powmod(a, s, p), g = powmod(n, s, p);
  for (;; r = m) {
    ll t = b;
    for (m = 0; m < r && t != 1; ++m)
      t = t * t % p;
    if (m == 0) return x;
    ll gs = powmod(g, 1LL << (r - m - 1), p);
    g = gs * gs % p;
    x = x * gs % p;
    b = b * g % p;
  }
}
```

3.5.2 Mod. exponentiation Or use pow() in python

```
#include "header.h"
```

```
ll mod_pow(ll base, ll exp, ll mod) {
  if (mod == 1) return 0;
    if (exp == 0) return 1;
    if (exp == 1) return base;

  ll res = 1;
  base %= mod;
  while (exp) {
    if (exp % 2 == 1) res = (res * base) % mod;
    exp >>= 1;
    base = (base * base) % mod;
  }

  return res % mod;
}
```

3.5.3 GCD Or math.gcd in python, std::gcd in C++

```
#include "header.h"
ll gcd(ll a, ll b) {
  if (a == 0) return b;
  return gcd(b % a, a);
}
```

3.5.4 Sieve of Eratosthenes

```
#include "header.h"
vl primes;
void getprimes(ll n) {  // Up to n (not included)
    vector<bool> p(n, true);
    p[0] = false;
    p[1] = false;
    for(ll i = 0; i < n; i++) {
        if(p[i]) {
            primes.push_back(i);
            for(ll j = i*2; j < n; j+=i) p[j] =
                false;
}}}
```

3.5.5 Fibonacci % prime

```
#include "header.h"
const ll MOD = 1000000007;
unordered_map<ll, ll> Fib;
ll fib(ll n) {
    if (n < 2) return 1;
    if (Fib.find(n) != Fib.end()) return Fib[n];
    Fib[n] = (fib((n + 1) / 2) * fib(n / 2) + fib
        ((n - 1) / 2) * fib((n - 2) / 2)) % MOD;
    return Fib[n];
}
```

3.5.6 nCk % prime

```
#include "header.h"
ll binom(ll n, ll k) {
    ll ans = 1;
    for(ll i = 1; i <= min(k,n-k); ++i) ans = ans
        *(n+1-i)/i;
    return ans;
}
ll mod_nCk(ll n, ll k, ll p ){
    ll ans = 1;
    while(n){
        ll np = n%p, kp = k%p;
        if(kp > np) return 0;
        ans *= binom(np,kp);
        n /= p; k /= p;
    }
    return ans;
}
```

3.5.7 Chin. rem. th.

```
#include "header.h"
#include "elementary.cpp"
// Solves x = a1 mod m1, x = a2 mod m2, x is
    unique modulo lcm(m1, m2).
// Returns {0, -1} on failure, {x, lcm(m1, m2)}
    otherwise.
pair<ll, ll> crt(ll a1, ll m1, ll a2, ll m2) {
  ll s, t, d;
  extended_euclid(m1, m2, s, t, d);
  if (a1 % d != a2 % d) return {0, -1};
  return {mod(s*a2 %m2 * m1 + t*a1 %m1 * m2, m1 *
      m2) / d, m1 / d * m2};
}

// Solves x = ai mod mi. x is unique modulo lcm
    mi.
// Returns {0, -1} on failure, {x, lcm mi}
    otherwise.
pair<ll, ll> crt(vector<ll> &a, vector<ll> &m) {
  pair<ll, ll> res = {a[0], m[0]};
  for (ull i = 1; i < a.size(); ++i) {
    res = crt(res.first, res.second, mod(a[i], m[
        i]), m[i]);
    if (res.second == -1) break;
  }
  return res;
}
```

3.5.8 Derangements

Permutations of a set such that none of the elements appear in their original position:

$$!n = (n-1)(!(n-1) + !(n-2)) = \left[\frac{n!}{e}\right] \tag{1}$$

$$!n = 1 - e^{-1},\ n \to \infty \tag{2}$$

3.6 Strings

3.6.1 Z alg. KMP alternative

```
#include "../header.h"
void Z_algorithm(const string &s, vi &Z) {
  Z.assign(s.length(), -1);
  int L = 0, R = 0, n = s.length();
  for (int i = 1; i < n; ++i) {
    if (i > R) {
      L = R = i;
      while (R < n && s[R - L] == s[R]) R++;
      Z[i] = R - L; R--;
    } else if (Z[i - L] >= R - i + 1) {
      L = i;
      while (R < n && s[R - L] == s[R]) R++;
      Z[i] = R - L; R--;
    } else Z[i] = Z[i - L];
  }
}
```

3.6.2 KMP

```
#include "header.h"
void compute_prefix_function(string &w, vi &
    prefix) {
  prefix.assign(w.length(), 0);
  int k = prefix[0] = -1;

  for(int i = 1; i < w.length(); ++i) {
    while(k >= 0 && w[k + 1] != w[i]) k = prefix[
        k];
    if(w[k + 1] == w[i]) k++;
    prefix[i] = k;
  }
}
void knuth_morris_pratt(string &s, string &w) {
  int q = -1;
  vi prefix;
  compute_prefix_function(w, prefix);
  for(int i = 0; i < s.length(); ++i) {
    while(q >= 0 && w[q + 1] != s[i]) q = prefix[
        q];
    if(w[q + 1] == s[i]) q++;
    if(q + 1 == w.length()) {
```

```
      // Match at position (i - w.length() + 1)
      q = prefix[q];
    }
  }
}
```

3.6.3 Aho-Corasick

Also can be used as Knuth-Morris-Pratt algorithm

```
#include "header.h"

map<char, int> cti;
int cti_size;
template <int ALPHABET_SIZE, int (*mp)(char)>
struct AC_FSM {
  struct Node {
    int child[ALPHABET_SIZE], failure = 0, match_par = -1;
    vi match;
    Node() { for (int i = 0; i < ALPHABET_SIZE; ++i) child[i] = -1; }
  };
  vector<Node> a;
  vector<string> &words;
  AC_FSM(vector<string> &words) : words(words) {
    a.push_back(Node());
    construct_automaton();
  }
  void construct_automaton() {
    for (int w = 0, n = 0; w < words.size(); ++w, n = 0) {
      for (int i = 0; i < words[w].size(); ++i) {
        if (a[n].child[mp(words[w][i])] == -1) {
          a[n].child[mp(words[w][i])] = a.size();
          a.push_back(Node());
        }
        n = a[n].child[mp(words[w][i])];
      }
      a[n].match.push_back(w);
    }
    queue<int> q;
    for (int k = 0; k < ALPHABET_SIZE; ++k) {
      if (a[0].child[k] == -1) a[0].child[k] = 0;
      else if (a[0].child[k] > 0) {
        a[a[0].child[k]].failure = 0;
        q.push(a[0].child[k]);
      }
    }
    while (!q.empty()) {
      int r = q.front(); q.pop();
      for (int k = 0, arck; k < ALPHABET_SIZE; ++k) {
        if ((arck = a[r].child[k]) != -1) {
          q.push(arck);
          int v = a[r].failure;
          while (a[v].child[k] == -1) v = a[v].failure;
          a[arck].failure = a[v].child[k];
          a[arck].match_par = a[v].child[k];
          while (a[arck].match_par != -1
              && a[a[arck].match_par].match.empty())
            a[arck].match_par = a[a[arck].match_par].match_par;
        }
      }
    }
  }
  void aho_corasick(string &sentence, vvi &matches){
    matches.assign(words.size(), vi());
    int state = 0, ss = 0;
    for (int i = 0; i < sentence.length(); ++i, ss = state) {
      while (a[ss].child[mp(sentence[i])] == -1)
        ss = a[ss].failure;
      state = a[state].child[mp(sentence[i])]
          = a[ss].child[mp(sentence[i])];
      for (ss = state; ss != -1; ss = a[ss].match_par)
        for (int w : a[ss].match)
          matches[w].push_back(i + 1 - words[w].length());
    }
  }
};
int char_to_int(char c) {
  return cti[c];
}
int main() {
  ll n;
  string line;
  while(getline(cin, line)) {
    stringstream ss(line);
    ss >> n;

    vector<string> patterns(n);
    for (auto& p: patterns) getline(cin, p);

    string text;
    getline(cin, text);

    cti = {}, cti_size = 0;
    for (auto c: text) {
      if (not in(c, cti)) {
        cti[c] = cti_size++;
      }
    }
    for (auto& p: patterns) {
      for (auto c: p) {
        if (not in(c, cti)) {
          cti[c] = cti_size++;
        }
      }
    }

    vvi matches;
    AC_FSM <128+1, char_to_int> ac_fms(patterns);
    ac_fms.aho_corasick(text, matches);
    for (auto& x: matches) cout << x << endl;
  }

}
```

3.6.4 Long. palin. subs

Manacher - $O(n)$

```
#include "header.h"
void manacher(string &s, vi &pal) {
  int n = s.length(), i = 1, l, r;
  pal.assign(2 * n + 1, 0);
  while (i < 2 * n + 1) {
    if ((i&1) && pal[i] == 0) pal[i] = 1;
    l = i / 2 - pal[i] / 2; r = (i-1) / 2 + pal[i] / 2;

    while (l - 1 >= 0 && r + 1 < n && s[l - 1] == s[r + 1])
      --l, ++r, pal[i] += 2;

    for (l = i - 1, r = i + 1; l >= 0 && r < 2 * n + 1; --l, ++r) {
      if (l <= i - pal[i]) break;
      if (l / 2 - pal[l] / 2 > i / 2 - pal[i] / 2)
        pal[r] = pal[l];
      else {  if (l >= 0)
          pal[r] = min(pal[l], i + pal[i] - r);
        break;
      }
    }
    i = r;
} }
```

3.6.5 Bitstring

Slower than an unordered set for many elements, but hashable

```
#include "../header.h"

template<size_t len>
struct pair_hash { // To make it hashable (pair<int, bitset<len>>)
```

```
    std::size_t operator()(const std::pair<int, std::bitset<len>>& p) const {
        std::size_t h1 = std::hash<int>{}(p.first);
        std::size_t h2 = std::hash<std::bitset<len>>{}(p.second);
        return h1 ^ (h2 << 1);
    }
};
#define MAXN 1000
std::bitset<MAXN> bs;
// bs.set(idx) <- set idx-th bit (1)
// bs.reset(idx) <- reset idx-th bit (0)
// bs.flip(idx) <- flip idx-th bit
// bs.test(idx) <- idx-th bit == 1
// bs.count() <- number of 1s
// bs.any() <- any bit == 1
```

3.7 Geometry

3.7.1 essentials.cpp

```
#include "../header.h"
using C = ld; // could be long long or long double
constexpr C EPS = 1e-10;  // change to 0 for C=ll
struct P {    // may also be used as a 2D vector
  C x, y;
  P(C x = 0, C y = 0) : x(x), y(y) {}
  P operator+ (const P &p) const { return {x + p.x, y + p.y}; }
  P operator- (const P &p) const { return {x - p.x, y - p.y}; }
  P operator* (C c) const { return {x * c, y * c}; }
  P operator/ (C c) const { return {x / c, y / c}; }
  C operator* (const P &p) const { return x*p.x + y*p.y; }
  C operator^ (const P &p) const { return x*p.y - p.x*y; }
  P perp() const { return P{y, -x}; }
  C lensq() const { return x*x + y*y; }
  ld len() const { return sqrt((ld)lensq()); }
  static ld dist(const P &p1, const P &p2) {
    return (p1-p2).len(); }
  bool operator==(const P &r) const {
    return ((*this)-r).lensq() <= EPS*EPS; }
};
C det(P p1, P p2) { return p1^p2; }
C det(P p1, P p2, P o) { return det(p1-o, p2-o); }
C det(const vector<P> &ps) {
  C sum = 0; P prev = ps.back();
  for(auto &p : ps) sum += det(p, prev), prev = p;
  return sum;
}
// Careful with division by two and C=ll
C area(P p1, P p2, P p3) { return abs(det(p1, p2, p3))/C(2); }
C area(const vector<P> &poly) { return abs(det(poly))/C(2); }
int sign(C c){ return (c > C(0)) - (c < C(0)); }
int ccw(P p1, P p2, P o) { return sign(det(p1, p2, o)); }

// Only well defined for C = ld.
P unit(const P &p) { return p / p.len(); }
P rotate(P p, ld a) { return P{p.x*cos(a)-p.y*sin(a), p.x*sin(a)+p.y*cos(a)}; }
```

3.7.2 Two segs. itersec.

```
#include "header.h"
#include "essentials.cpp"
bool intersect(P a1, P a2, P b1, P b2) {
  if (max(a1.x, a2.x) < min(b1.x, b2.x)) return false;
  if (max(b1.x, b2.x) < min(a1.x, a2.x)) return false;
  if (max(a1.y, a2.y) < min(b1.y, b2.y)) return false;
  if (max(b1.y, b2.y) < min(a1.y, a2.y)) return false;
  bool l1 = ccw(a2, b1, a1) * ccw(a2, b2, a1) <= 0;
  bool l2 = ccw(b2, a1, b1) * ccw(b2, a2, b1) <= 0;
  return l1 && l2;
}
```

3.7.3 Convex Hull

```
#include "header.h"
#include "essentials.cpp"
struct ConvexHull {  // O(n lg n) monotone chain.
  size_t n;
  vector<size_t> h, c;  // Indices of the hull are in `h`, ccw.
  const vector<P> &p;
  ConvexHull(const vector<P> &_p) : n(_p.size()), c(n), p(_p) {
    std::iota(c.begin(), c.end(), 0);
    std::sort(c.begin(), c.end(), [this](size_t l, size_t r) -> bool { return p[l].x != p[r].x ? p[l].x < p[r].x : p[l].y < p[r].y; });
    c.erase(std::unique(c.begin(), c.end(), [this](size_t l, size_t r) { return p[l] == p[r]; }), c.end());
    for (size_t s = 1, r = 0; r < 2; ++r, s = h.size()) {
      for (size_t i : c) {
        while (h.size() > s && ccw(p[h.end()[-2]], p[h.end()[-1]], p[i]) <= 0)
          h.pop_back();
        h.push_back(i);
      }
      reverse(c.begin(), c.end());
    }
    if (h.size() > 1) h.pop_back();
  }
  size_t size() const { return h.size(); }
  template <class T, void U(const P &, const P &, const P &, T &)>
  void rotating_calipers(T &ans) {
    if (size() <= 2)
      U(p[h[0]], p[h.back()], p[h.back()], ans);
    else
      for (size_t i = 0, j = 1, s = size(); i < 2 * s; ++i) {
        while (det(p[h[(i + 1) % s]] - p[h[i % s]], p[h[(j + 1) % s]] - p[h[j]]) >= 0)
          j = (j + 1) % s;
        U(p[h[i % s]], p[h[(i + 1) % s]], p[h[j]], ans);
      }
  }
};
// Example: furthest pair of points. Now set ans = 0LL and call
// ConvexHull(pts).rotating_calipers<ll, update>(ans);
void update(const P &p1, const P &p2, const P &o, ll &ans) {
  ans = max(ans, (ll)max((p1 - o).lensq(), (p2 - o).lensq()));
}
int main() {
  ios::sync_with_stdio(false);  // do not use cout + printf
  cin.tie(NULL);

  int n;
  cin >> n;
  while (n) {
    vector<P> ps;
      int x, y;
    for (int i = 0; i < n; i++) {
          cin >> x >> y;
          ps.push_back({x, y});
```

```
        }

        ConvexHull ch(ps);
        cout << ch.h.size() << endl;
        for(auto& p: ch.h) {
            cout << ps[p].x << " " << ps[p].y << endl;
        }
    cin >> n;
  }

  return 0;
}
```

3.8 Other Algorithms

3.8.1 2-sat

```
#include "../header.h"
#include "../Graphs/tarjan.cpp"
struct TwoSAT {
  int n;
  vvi imp; // implication graph
  Tarjan tj;

  TwoSAT(int _n) : n(_n), imp(2 * _n, vi()), tj(imp) { }

  // Only copy the needed functions:
  void add_implies(int c1, bool v1, int c2, bool v2) {
    int u = 2 * c1 + (v1 ? 1 : 0),
      v = 2 * c2 + (v2 ? 1 : 0);
    imp[u].push_back(v);    //  u =>  v
    imp[v^1].push_back(u^1);  // -v => -u
  }
  void add_equivalence(int c1, bool v1, int c2, bool v2) {
    add_implies(c1, v1, c2, v2);
    add_implies(c2, v2, c1, v1);
  }
  void add_or(int c1, bool v1, int c2, bool v2) {
    add_implies(c1, !v1, c2, v2);
  }
  void add_and(int c1, bool v1, int c2, bool v2) {
    add_true(c1, v1); add_true(c2, v2);
  }
  void add_xor(int c1, bool v1, int c2, bool v2) {
    add_or(c1, v1, c2, v2);
    add_or(c1, !v1, c2, !v2);
  }
  void add_true(int c1, bool v1) {
    add_implies(c1, !v1, c1, v1);
  }

  // on true: a contains an assignment.
  // on false: no assignment exists.
  bool solve(vb &a) {
    vi com;
    tj.find_sccs(com);
    for (int i = 0; i < n; ++i)
      if (com[2 * i] == com[2 * i + 1])
        return false;

    vvi bycom(com.size());
    for (int i = 0; i < 2 * n; ++i)
      bycom[com[i]].push_back(i);

    a.assign(n, false);
    vb vis(n, false);
    for(auto &&component : bycom){
      for (int u : component) {
        if (vis[u / 2]) continue;
        vis[u / 2] = true;
        a[u / 2] = (u % 2 == 1);
      }
    }
    return true;
  }
};
```

3.8.2 Matrix Solve

```
#include "header.h"
#define REP(i, n) for(auto i = decltype(n)(0); i < (n); i++)
using T = double;
constexpr T EPS = 1e-8;
template<int R, int C>
using M = array<array<T,C>,R>;  // matrix
template<int R, int C>
T ReducedRowEchelonForm(M<R,C> &m, int rows) {    // return the determinant
  int r = 0; T det = 1;             // MODIFIES the input
  for(int c = 0; c < rows && r < rows; c++) {
    int p = r;
    for(int i=r+1; i<rows; i++) if(abs(m[i][c]) > abs(m[p][c])) p=i;
    if(abs(m[p][c]) < EPS){ det = 0; continue; }
    swap(m[p], m[r]);   det = -det;
    T s = 1.0 / m[r][c], t; det *= m[r][c];
    REP(j,C) m[r][j] *= s;         // make leading term in row 1
    REP(i,rows) if (i!=r){ t = m[i][c]; REP(j,C) m[i][j] -= t*m[r][j]; }
    ++r;
  }
  return det;
}
bool error, inconst;  // error => multiple or inconsistent
template<int R,int C> // Mx = a; M:R*R, v:R*C => x:R*C
M<R,C> solve(const M<R,R> &m, const M<R,C> &a, int rows){
  M<R,R+C> q;
  REP(r,rows){
    REP(c,rows) q[r][c] = m[r][c];
    REP(c,C) q[r][R+c] = a[r][c];
  }
  ReducedRowEchelonForm<R,R+C>(q,rows);
  M<R,C> sol; error = false, inconst = false;
  REP(c,C) for(auto j = rows-1; j >= 0; --j){
    T t=0; bool allzero=true;
    for(auto k = j+1; k < rows; ++k)
      t += q[j][k]*sol[k][c], allzero &= abs(q[j][k]) < EPS;
    if(abs(q[j][j]) < EPS)
      error = true, inconst |= allzero && abs(q[j][R+c]) > EPS;
    else sol[j][c] = (q[j][R+c] - t) / q[j][j];   // usually q[j][j]=1
  }
  return sol;
}
```

3.8.3 Matrix Exp.

```
#include "header.h"
#define ITERATE_MATRIX(w) for (int r = 0; r < (w); ++r) \
              for (int c = 0; c < (w); ++c)
template <class T, int N>
struct M {
  array<array<T,N>,N> m;
  M() { ITERATE_MATRIX(N) m[r][c] = 0; }
  static M id() {
    M I; for (int i = 0; i < N; ++i) I.m[i][i] = 1; return I;
  }
  M operator*(const M &rhs) const {
    M out;
    ITERATE_MATRIX(N) for (int i = 0; i < N; ++i)
        out.m[r][c] += m[r][i] * rhs.m[i][c];
    return out;
  }
  M raise(ll n) const {
    if(n == 0) return id();
    if(n == 1) return *this;
```

```
    auto r = (*this**this).raise(n / 2);
    return (n%2 ? *this*r : r);
  }
};
```

3.8.4 Finite field For FFT

```
#include "header.h"
#include "../Number Theory/elementary.cpp"
template<ll p,ll w> // prime, primitive root
struct Field { using T = Field; ll x; Field(ll x=0) : x{x} {}
  T operator+(T r) const { return {(x+r.x)%p}; }
  T operator-(T r) const { return {(x-r.x+p)%p}; }
  T operator*(T r) const { return {(x*r.x)%p}; }
  T operator/(T r) const { return (*this)*r.inv(); }
  T inv() const { return {mod_inverse(x,p)}; }
  static T root(ll k) { assert( (p-1)%k==0 );  // (p-1)%k == 0?
    auto r = powmod(w,(p-1)/abs(k),p);      // k-th root of unity
    return k>=0 ? T{r} : T{r}.inv();
  }
  bool zero() const { return x == 0LL; }
};
using F1 = Field<1004535809,3 >;
using F2 = Field<1107296257,10>; // 1<<30 + 1<<25 + 1
using F3 = Field<2281701377,3 >; // 1<<31 + 1<<27 + 1
```

3.8.5 Complex field For FFR

```
#include "header.h"
const double m_pi = M_PIf64x;
struct Complex { using T = Complex; double u,v;
  Complex(double u=0, double v=0) : u{u}, v{v} {}
  T operator+(T r) const { return {u+r.u, v+r.v}; }
  T operator-(T r) const { return {u-r.u, v-r.v}; }
  T operator*(T r) const { return {u*r.u - v*r.v, u*r.v + v*r.u}; }
  T operator/(T r) const {
    auto norm = r.u*r.u+r.v*r.v;
    return {(u*r.u + v*r.v)/norm, (v*r.u - u*r.v)/norm};
  }
  T operator*(double r) const { return T{u*r, v*r}; }
  T operator/(double r) const { return T{u/r, v/r}; }
  T inv() const { return T{1,0}/ *this; }
  T conj() const { return T{u, -v}; }
  static T root(ll k){ return {cos(2*m_pi/k), sin(2*m_pi/k)}; }
  bool zero() const { return max(abs(u), abs(v)) < 1e-6; }
};
```

3.8.6 FFT

```
#include "header.h"
#include "complex_field.cpp"
#include "fin_field.cpp"
void brinc(int &x, int k) {
  int i = k - 1, s = 1 << i;
  x ^= s;
  if ((x & s) != s) {
    --i; s >>= 1;
    while (i >= 0 && ((x & s) == s))
      x = x &~ s, --i, s >>= 1;
    if (i >= 0) x |= s;
  }
}
using T = Complex;  // using T=F1,F2,F3
vector<T> roots;
void root_cache(int N) {
  if (N == (int)roots.size()) return;
  roots.assign(N, T{0});
  for (int i = 0; i < N; ++i)
    roots[i] = ((i&-i) == i)
      ? T{cos(2.0*m_pi*i/N), sin(2.0*m_pi*i/N)}
      : roots[i&-i] * roots[i-(i&-i)];
}
void fft(vector<T> &A, int p, bool inv = false) {
  int N = 1<<p;
  for(int i = 0, r = 0; i < N; ++i, brinc(r, p))
    if (i < r) swap(A[i], A[r]);
//  Uncomment to precompute roots (for T=Complex). Slower but more precise.
//  root_cache(N);
//          , sh=p-1     , --sh
  for (int m = 2; m <= N; m <<= 1) {
    T w, w_m = T::root(inv ? -m : m);
    for (int k = 0; k < N; k += m) {
      w = T{1};
      for (int j = 0; j < m/2; ++j) {
//        T w = (!inv ? roots[j<<sh] : roots[j<<sh].conj());
        T t = w * A[k + j + m/2];
        A[k + j + m/2] = A[k + j] - t;
        A[k + j] = A[k + j] + t;
        w = w * w_m;
      }
    }
  }
  if(inv){ T inverse = T(N).inv(); for(auto &x : A) x = x*inverse; }
}
// convolution leaves A and B in frequency domain state
// C may be equal to A or B for in-place convolution
void convolution(vector<T> &A, vector<T> &B, vector<T> &C){
  int s = A.size() + B.size() - 1;
  int q = 32 - __builtin_clz(s-1), N=1<<q;  // fails if s=1
  A.resize(N,{}); B.resize(N,{}); C.resize(N,{});
  fft(A, q, false); fft(B, q, false);
  for (int i = 0; i < N; ++i) C[i] = A[i] * B[i];
  fft(C, q, true); C.resize(s);
}
void square_inplace(vector<T> &A) {
  int s = 2*A.size()-1, q = 32 - __builtin_clz(s-1), N=1<<q;
  A.resize(N,{}); fft(A, q, false);
  for(auto &x : A) x = x*x;
  fft(A, q, true); A.resize(s);
}
```

3.8.7 Polyn. inv. div.

```
#include "header.h"
#include "fft.cpp"
vector<T> &rev(vector<T> &A) { reverse(A.begin(), A.end()); return A; }
void copy_into(const vector<T> &A, vector<T> &B, size_t n) {
  std::copy(A.begin(), A.begin()+min({n, A.size(), B.size()}), B.begin());
}

// Multiplicative inverse of A modulo x^n. Requires A[0] != 0!!
vector<T> inverse(const vector<T> &A, int n) {
  vector<T> Ai{A[0].inv()};
  for (int k = 0; (1<<k) < n; ++k) {
    vector<T> As(4<<k, T(0)), Ais(4<<k, T(0));
    copy_into(A, As, 2<<k); copy_into(Ai, Ais, Ai.size());
    fft(As, k+2, false); fft(Ais, k+2, false);
    for (int i = 0; i < (4<<k); ++i) As[i] = As[i]*Ais[i]*Ais[i];
    fft(As, k+2, true); Ai.resize(2<<k, {});
    for (int i = 0; i < (2<<k); ++i) Ai[i] = T(2) * Ai[i] - As[i];
```

```
  }
  Ai.resize(n);
  return Ai;
}
// Polynomial division. Returns {Q, R} such that A = QB+R, deg R < deg B.
// Requires that the leading term of B is nonzero.
pair<vector<T>, vector<T>> divmod(const vector<T> &A, const vector<T> &B) {
  size_t n = A.size()-1, m = B.size()-1;
  if (n < m) return {vector<T>(1, T(0)), A};

  vector<T> X(A), Y(B), Q, R;
  convolution(rev(X), Y = inverse(rev(Y), n-m+1), Q);
  Q.resize(n-m+1); rev(Q);

  X.resize(Q.size()), copy_into(Q, X, Q.size());
  Y.resize(B.size()), copy_into(B, Y, B.size());
  convolution(X, Y, X);

  R.resize(m), copy_into(A, R, m);
  for (size_t i = 0; i < m; ++i) R[i] = R[i] - X[i];
  while (R.size() > 1 && R.back().zero()) R.pop_back();
  return {Q, R};
}
vector<T> mod(const vector<T> &A, const vector<T> &B) {
  return divmod(A, B).second;
}
```

3.8.8 Linear recurs.

Given a linear recurrence of the form

$$a_n = \sum_{i=0}^{k-1} c_i a_{n-i-1}$$

this code computes a_n in $O(k \log k \log n)$ time.

```
#include "header.h"
#include "poly.cpp"
// x^k mod f
vector<T> xmod(const vector<T> f, ll k) {
  vector<T> r{T(1)};
  for (int b = 62; b >= 0; --b) {
    if (r.size() > 1)
      square_inplace(r), r = mod(r, f);
    if ((k>>b)&1) {
      r.insert(r.begin(), T(0));
      if (r.size() == f.size()) {
        T c = r.back() / f.back();
        for (size_t i = 0; i < f.size(); ++i)
          r[i] = r[i] - c * f[i];
        r.pop_back();
      }
    }
  }
  return r;
}
// Given A[0,k) and C[0, k), computes the n-th term of:
// A[n] = \sum_i C[i] * A[n-i-1]
T nth_term(const vector<T> &A, const vector<T> &C, ll n) {
  int k = (int)A.size();
  if (n < k) return A[n];

  vector<T> f(k+1, T{1});
  for (int i = 0; i < k; ++i)
    f[i] = T{-1} * C[k-i-1];
  f = xmod(f, n);

  T r = T{0};
  for (int i = 0; i < k; ++i)
    r = r + f[i] * A[i];
  return r;
}
```

3.8.9 Convolution

Precise up to 9e15

```
#include "header.h"
#include "fft.cpp"
void convolution_mod(const vi &A, const vi &B, ll MOD, vi &C) {
  int s = A.size() + B.size() - 1; ll m15 = (1LL<<15)-1LL;
  int q = 32 - __builtin_clz(s-1), N=1<<q;  // fails if s=1
  vector<T> Ac(N), Bc(N), R1(N), R2(N);
  for (size_t i = 0; i < A.size(); ++i) Ac[i] = T{A[i]&m15, A[i]>>15};
  for (size_t i = 0; i < B.size(); ++i) Bc[i] = T{B[i]&m15, B[i]>>15};
  fft(Ac, q, false); fft(Bc, q, false);
  for (int i = 0, j = 0; i < N; ++i, j = (N-1)&(N-i)) {
    T as = (Ac[i] + Ac[j].conj()) / 2;
    T al = (Ac[i] - Ac[j].conj()) / T{0, 2};
    T bs = (Bc[i] + Bc[j].conj()) / 2;
    T bl = (Bc[i] - Bc[j].conj()) / T{0, 2};
    R1[i] = as*bs + al*bl*T{0,1}, R2[i] = as*bl + al*bs;
  }
  fft(R1, q, true); fft(R2, q, true);
  ll p15 = (1LL<<15)%MOD, p30 = (1LL<<30)%MOD; C.resize(s);
  for (int i = 0; i < s; ++i) {
    ll l = llround(R1[i].u), m = llround(R2[i].u), h = llround(R1[i].v);
    C[i] = (l + m*p15 + h*p30) % MOD;
  }
}
```

3.8.10 Partitions of n

Finds all possible partitions of a number

```
#include "header.h"
void printArray(int p[], int n) {
  for (int i = 0; i < n; i++)
    cout << p[i] << " ";
  cout << endl;
}

void printAllUniqueParts(int n) {
  int p[n]; // An array to store a partition
  int k = 0;  // Index of last element in a partition
  p[k] = n; // Initialize first partition as number itself

  // This loop first prints current partition then generates next
  // partition. The loop stops when the current partition has all 1s
  while (true) {
    printArray(p, k + 1);

    // Find the rightmost non-one value in p[]. Also, update the
    // rem_val so that we know how much value can be accommodated
    int rem_val = 0;
    while (k >= 0 && p[k] == 1) {
      rem_val += p[k];
      k--;
    }

    // if k < 0, all the values are 1 so there are no more partitions
    if (k < 0) return;

    // Decrease the p[k] found above and adjust the rem_val
    p[k]--;
    rem_val++;

    // If rem_val is more, then the sorted order is violated. Divide
    // rem_val in different values of size p[k] and copy these values at
```

```
    // different positions after p[k]
    while (rem_val > p[k]) {
      p[k + 1] = p[k];
      rem_val = rem_val - p[k];
      k++;
    }

    // Copy rem_val to next position and
        increment position
    p[k + 1] = rem_val;
    k++;
  }
}
```

3.8.11 Ternary search

```
/**
 * Description:
 * Find the smallest i in $[a,b]$ that maximizes
     $f(i)$, assuming that $f(a) < \dots < f(i) \
     ge \dots \ge f(b)$.
 * To reverse which of the sides allows non-
     strict inequalities, change the < marked
     with (A) to <=, and reverse the loop at (B).
 * To minimize $f$, change it to >, also at (B).
 * Usage:
  int ind = ternSearch(0,n-1,[\&](int i){return a
      [i];});
 * Time: O(\log(b-a))
 */
#include "../Numerical/template.cpp"

template<class F>
int ternSearch(int a, int b, F f) {
  assert(a <= b);
  while (b - a >= 5) {
    int mid = (a + b) / 2;
    if (f(mid) < f(mid+1)) a = mid; // (A)
    else b = mid+1;
  }
  rep(i,a+1,b+1) if (f(a) < f(i)) a = i; // (B)
  return a;
}
```

3.9 Other Data Structures

3.9.1 Disjoint set (i.e. union-find)

```
template <typename T>
class DisjointSet {
    typedef T * iterator;
    T *parent, n, *rank;
    public:
        // O(n), assumes nodes are [0, n)
        DisjointSet(T n) {
            this->parent = new T[n];
            this->n = n;
            this->rank = new T[n];

            for (T i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 0;
            }
        }

        // O(log n)
        T find_set(T x) {
            if (x == parent[x]) return x;
            return parent[x] = find_set(parent[x
                ]);
        }

        // O(log n)
        void union_sets(T x, T y) {
            x = this->find_set(x);
            y = this->find_set(y);

            if (x == y) return;

            if (rank[x] < rank[y]) {
                T z = x;
                x = y;
                y = z;
            }

            parent[y] = x;
            if (rank[x] == rank[y]) rank[x]++;
        }
};
```

3.9.2 Fenwick tree (i.e. BIT) eff. update + prefix sum calc.

```
#include "header.h"
#define maxn 200010
int t,n,m,tree[maxn],p[maxn];

void update(int k, int z) {
    while (k <= maxn) {
        tree[k] += z;
        k += k & (-k);
    }
}

int sum(int k) {
    int ans = 0;
    while(k) {
        ans += tree[k];
        k -= k & (-k);
    }
    return ans;
}
```

3.9.3 Fenwick2d tree

```
#include "header.h"
template <class T>
struct FenwickTree2D {
  vector< vector<T> > tree;
  int n;
  FenwickTree2D(int n) : n(n) { tree.assign(n +
      1, vector<T>(n + 1, 0)); }
  T query(int x1, int y1, int x2, int y2) {
    return query(x2,y2)+query(x1-1,y1-1)-query(x2
        ,y1-1)-query(x1-1,y2);
  }
  T query(int x, int y) {
    T s = 0;
    for (int i = x; i > 0; i -= (i & (-i)))
      for (int j = y; j > 0; j -= (j & (-j)))
        s += tree[i][j];
    return s;
  }
  void update(int x, int y, T v) {
    for (int i = x; i <= n; i += (i & (-i)))
      for (int j = y; j <= n; j += (j & (-j)))
        tree[i][j] += v;
  }
};
```

3.9.4 Trie

```
#include "header.h"
const int ALPHABET_SIZE = 26;
inline int mp(char c) { return c - 'a'; }

struct Node {
  Node* ch[ALPHABET_SIZE];
  bool isleaf = false;
  Node() {
    for(int i = 0; i < ALPHABET_SIZE; ++i) ch[i]
        = nullptr;
  }

  void insert(string &s, int i = 0) {
    if (i == s.length()) isleaf = true;
    else {
      int v = mp(s[i]);
      if (ch[v] == nullptr)
        ch[v] = new Node();
```

```cpp
      ch[v]->insert(s, i + 1);
    }
  }

  bool contains(string &s, int i = 0) {
    if (i == s.length()) return isleaf;
    else {
      int v = mp(s[i]);
      if (ch[v] == nullptr) return false;
      else return ch[v]->contains(s, i + 1);
    }
  }

  void cleanup() {
    for (int i = 0; i < ALPHABET_SIZE; ++i)
      if (ch[i] != nullptr) {
        ch[i]->cleanup();
        delete ch[i];
      }
  }
};
```

3.9.5 Treap

A binary tree whose nodes contain two values, a key and a priority, such that the key keeps the BST property

```cpp
#include "header.h"
struct Node {
  ll v;
  int sz, pr;
  Node *l = nullptr, *r = nullptr;
  Node(ll val) : v(val), sz(1) { pr = rand(); }
};
int size(Node *p) { return p ? p->sz : 0; }
void update(Node* p) {
  if (!p) return;
  p->sz = 1 + size(p->l) + size(p->r);
  // Pull data from children here
}
void propagate(Node *p) {
  if (!p) return;
  // Push data to children here
}
void merge(Node *&t, Node *l, Node *r) {
  propagate(l), propagate(r);
  if (!l)     t = r;
  else if (!r)  t = l;
  else if (l->pr > r->pr)
      merge(l->r, l->r, r), t = l;
  else  merge(r->l, l, r->l), t = r;
  update(t);
}
void spliti(Node *t, Node *&l, Node *&r, int index) {
  propagate(t);
  if (!t) { l = r = nullptr; return; }
  int id = size(t->l);
  if (index <= id) // id \in [index, \infty), so move it right
    spliti(t->l, l, t->l, index), r = t;
  else
    spliti(t->r, t->r, r, index - id), l = t;
  update(t);
}
void splitv(Node *t, Node *&l, Node *&r, ll val) {
  propagate(t);
  if (!t) { l = r = nullptr; return; }
  if (val <= t->v) // t->v \in [val, \infty), so move it right
    splitv(t->l, l, t->l, val), r = t;
  else
    splitv(t->r, t->r, r, val), l = t;
  update(t);
}
void clean(Node *p) {
  if (p) { clean(p->l), clean(p->r); delete p; }
}
```

3.9.6 Segment tree

```cpp
#include "../header.h"
template <class T, const T&(*op)(const T&, const T&)>
struct SegmentTree {
  int n; vector<T> tree; T id;
  SegmentTree(int _n, T _id) : n(_n), tree(2 * n, _id), id(_id) { }
  void update(int i, T val) {
    for (tree[i+n] = val, i = (i+n)/2; i > 0; i /= 2)
      tree[i] = op(tree[2*i], tree[2*i+1]);
  }
  T query(int l, int r) {
    T lhs = T(id), rhs = T(id);
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
      if (  l&1 ) lhs = op(lhs, tree[l++]);
      if (!(r&1)) rhs = op(tree[r--], rhs);
    }
    return op(l == r ? op(lhs, tree[l]) : lhs, rhs);
  }
};
```

3.9.7 Lazy segment tree

Uptimizes range updates

```cpp
#include "../header.h"
using T=int; using U=int; using I=int;    // exclusive right bounds
T t_id; U u_id;
T op(T a, T b){ return a+b; }
void join(U &a, U b){ a+=b; }
void apply(T &t, U u, int x){ t+=x*u; }
T convert(const I &i){ return i; }
struct LazySegmentTree {
  struct Node { int l, r, lc, rc; T t; U u;
    Node(int l, int r, T t=t_id):l(l),r(r),lc(-1),rc(-1),t(t),u(u_id){}
  };
  int N; vector<Node> tree; vector<I> &init;
  LazySegmentTree(vector<I> &init) : N(init.size()), init(init){
    tree.reserve(2*N-1); tree.push_back({0,N}); build(0, 0, N);
  }
  void build(int i, int l, int r) { auto &n = tree[i];
    if (r > l+1) { int m = (l+r)/2;
      n.lc = tree.size();    n.rc = n.lc+1;
      tree.push_back({l,m});    tree.push_back({m,r});
      build(n.lc,l,m);     build(n.rc,m,r);
      n.t = op(tree[n.lc].t, tree[n.rc].t);
    } else n.t = convert(init[l]);
  }
  void push(Node &n, U u){ apply(n.t, u, n.r-n.l); join(n.u,u); }
  void push(Node &n){push(tree[n.lc],n.u);push(tree[n.rc],n.u);n.u=u_id;}
  T query(int l, int r, int i = 0) { auto &n = tree[i];
    if(r <= n.l || n.r <= l) return t_id;
    if(l <= n.l && n.r <= r) return n.t;
    return push(n), op(query(l,r,n.lc),query(l,r,n.rc));
  }
  void update(int l, int r, U u, int i = 0) { auto &n = tree[i];
    if(r <= n.l || n.r <= l) return;
    if(l <= n.l && n.r <= r) return push(n,u);
    push(n); update(l,r,u,n.lc); update(l,r,u,n.rc);
    n.t = op(tree[n.lc].t, tree[n.rc].t);
  }
};
```

3.9.8 Suffix tree

```cpp
#include "../header.h"
using T = char;
```

```cpp
using M = map<T,int>;   // or array<T, ALPHABET_SIZE>
using V = string;     // could be vector<T> as well
using It = V::const_iterator;
struct Node{
  It b, e; M edges; int link;   // end is exclusive
  Node(It b, It e) : b(b), e(e), link(-1) {}
  int size() const { return e-b; }
};
struct SuffixTree{
  const V &s; vector<Node> t;
  int root,n,len,remainder,llink; It edge;
  SuffixTree(const V &s) : s(s) { build(); }
  int add_node(It b, It e){ return t.push_back({b,e}), t.size()-1; }
  int add_node(It b){ return add_node(b,s.end()); }
  void link(int node){ if(llink) t[llink].link = node; llink = node; }
  void build(){
    len = remainder =  0; edge = s.begin();
    n = root = add_node(s.begin(), s.begin());
    for(auto i = s.begin(); i != s.end(); ++i){
      ++remainder; llink = 0;
      while(remainder){
        if(len == 0) edge = i;
        if(t[n].edges[*edge] == 0){     // add new leaf
          t[n].edges[*edge] = add_node(i); link(n);
        } else {
          auto x = t[n].edges[*edge];   // neXt node [with edge]
          if(len >= t[x].size()){     // walk to next node
            len -= t[x].size(); edge += t[x].size(); n = x;
            continue;
          }
          if(*(t[x].b + len) == *i){    // walk along edge
            ++len; link(n); break;
          }               // split edge
          auto split = add_node(t[x].b, t[x].b+len);
          t[n].edges[*edge] = split;
          t[x].b += len;
          t[split].edges[*i] = add_node(i);
          t[split].edges[*t[x].b] = x;
          link(split);
        }
        --remainder;
        if(n == root && len > 0)
          --len, edge = i - remainder + 1;
        else n = t[n].link > 0 ? t[n].link : root;
      }
    }
  }
};
```

3.9.9 UnionFind

```cpp
#include "header.h"
struct UnionFind {
  std::vector<int> par, rank, size;
  int c;
  UnionFind(int n) : par(n), rank(n, 0), size(n, 1), c(n) {
    for(int i = 0; i < n; ++i) par[i] = i;
  }
  int find(int i) { return (par[i] == i ? i : (par[i] = find(par[i]))); }
  bool same(int i, int j) { return find(i) == find(j); }
  int get_size(int i) { return size[find(i)]; }
  int count() { return c; }
  int merge(int i, int j) {
    if((i = find(i)) == (j = find(j))) return -1;
    --c;
    if(rank[i] > rank[j]) swap(i, j);
    par[i] = j;
    size[j] += size[i];
    if(rank[i] == rank[j]) rank[j]++;
    return j;
  }
};
```

4 Other Mathematics

4.1 Helpful functions

4.1.1 Euler's Totient Fucntion

$n = p_1^{k_1-1} \cdot (p_1 - 1) \cdot \ldots \cdot p_r^{k_r-1} \cdot (p_r - 1)$, where $p_1^{k_1} \cdot \ldots \cdot p_r^{k_r}$ is the prime factorization of n.

```cpp
# include "header.h"
ll phi(ll n) {  // \Phi(n)
    ll ans = 1;
    for (ll i = 2; i*i <= n; i++) {
        if (n % i == 0) {
            ans *= i-1;
            n /= i;
            while (n % i == 0) {
                ans *= i;
                n /= i;
            }
        }
    }
    if (n > 1) ans *= n-1;
    return ans;
}
vi phis(int n) {  // All \Phi(i) up to n
  vi phi(n + 1, 0LL);
  iota(phi.begin(), phi.end(), 0LL);
  for (ll i = 2LL; i <= n; ++i)
    if (phi[i] == i)
      for (ll j = i; j <= n; j += i)
        phi[j] -= phi[j] / i;
  return phi;
}
```

Formulas $\Phi(n)$ counts all numbers in $1, \ldots, n-1$ coprime to n.
$a^{\varphi(n)} \equiv 1 \mod n$, a and n are coprimes.
$\forall e > \log_2 m: \; n^e \mod m = n^{\Phi(m)+e \mod \Phi(m)} \mod m$.
$\gcd(m, n) = 1 \Rightarrow \Phi(m \cdot n) = \Phi(m) \cdot \Phi(n)$.

4.1.2 Pascal's trinagle

$\binom{n}{k}$ is k-th element in the n-th row, indexing both from 0

```cpp
#include "header.h"
void printPascal(int n) {
    for (int line = 1; line <= n; line++) {
        int C = 1; // used to represent C(line, i)
        for (int i = 1; i <= line; i++) {

            // The first value in a line is always 1
            cout << C << " ";
            C = C * (line - i) / i;
        }
        cout << "\n";
    }
}
```

4.2 Theorems and definitions

Fermat's little theorem

$$a^p \equiv a \mod p$$

Subfactorial

$$!n = n! \sum_{i=0}^{n} \frac{(-1)^i}{i!}$$

$$!(0) = 1,\ !n = n \cdot !(n-1) + (-1)^n$$

Binomials and other partitionings

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} = \prod_{i=1}^{k} \frac{n-i+1}{i}$$

This last product may be computed incrementally since any product of k' consecutive values is divisibleby $k'!$. Basic identities: The hockeystick identity:

$$\sum_{k=r}^{n} \binom{k}{r} = \binom{n+1}{r+1}$$

or

$$\sum_{k \le n} \binom{r+k}{k} = \binom{r+n+1}{n}$$

Also

$$\sum_{k=0}^{n} \binom{k}{m} = \binom{n+1}{m+1}$$

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

For $n, m \ge 0$ and p prime: write n, m in base p, i.e. $n = n_k p^k + \dots + n_1 p + n_0$ and $m = m_k p^k + \dots m_1 p + m_0$. Then by Lucas theorem we have $\binom{n}{m} \equiv \prod_{i=0}^{k} \binom{n_i}{m_i} \mod p$, with the convention that $n_i < m_i \implies \binom{n_i}{m_i} = 0$.

Fibonacci (See also number theory section)

$$\sum_{0 \le k \le n} \binom{n-k}{k} = F_{n+1}$$

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n$$

$\sum_{i=1}^{n} F_i = F_{n+2} - 1$, $\sum_{i=1}^{n} F_i^2 = F_n F_{n+1}$

$$\gcd(F_m, F_n) = F_{\gcd(m,n)}$$

$$\gcd(F_n, F_{n+1}) = \gcd(F_n, F_{n+2}) = 1$$

Bit stuff $a + b = a \oplus b + 2(a \& b) = a | b + a \& b$.
kth bit is set in x iff $x \mod 2^{k-1} \ge 2^k$, or iff $x \mod 2^{k-1} - x \mod 2^k \ne 0$ (i.e. $= 2^k$) It comes handy when you need to look at the bits of the numbers which are pair sums or subset sums etc.
$n \mod 2^i = n \& (2^i - 1)$.
$\forall k:\ 1 \oplus 2 \oplus \ldots \oplus (4k-1) = 0$

Stirling's numbers First kind: $S_1(n,k)$ count permutations on n items with k cycles. $S_1(n,k) = S_1(n-1,k-1) + (n-1)S_1(n-1,k)$ with $S_1(0,0) = 1$. Note:

$$\sum_{k=0}^{n} S_1(n,k) x^k = x(x+1)\ldots(x+n-1)$$

$$\sum_{k=0}^{n} S_1(n,k) = n!$$

Second kind: $S_2(n,k)$ count partitions of n distinct elements into exactly k non-empty groups.

$$S_2(n,k) = S_2(n-1,k-1) + kS_2(n-1,k)$$

$$S_2(n,1) = S_2(n,n) = 1$$

$$S_2(n,k) = \frac{1}{k!} \sum_{i=0}^{k} (-1)^{k-i} \binom{k}{i} i^n$$

4.3 Geometry Formulas

$$[ABC] = rs = \frac{1}{2} ab \sin\gamma$$

$$= \frac{abc}{4R} = \sqrt{s(s-a)(s-b)(s-c)} = \frac{1}{2} \left| (B-A, C-A)^T \right|$$

$$s = \frac{a+b+c}{2} \qquad 2R = \frac{a}{\sin\alpha}$$

cosine rule: $c^2 = a^2 + b^2 - 2ab\cos\gamma$

Euler: $1 + CC = V - E + F$

Pick: $\text{Area} = \text{itr pts} + \frac{\text{bdry pts}}{2} - 1$

$p \cdot q = |p||q|\cos(\theta) \qquad |p \times q| = |p||q|\sin(\theta)$

Given a non-self-intersecting closed polygon on n vertices, given as (x_i, y_i), its centroid (C_x, C_y) is given as:

$$C_x = \frac{1}{6A} \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$C_y = \frac{1}{6A} \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i)$$

$$A = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i) = \text{polygon area}$$

Inclusion-Exclusion For appropriate f compute $\sum_{S \subseteq T} (-1)^{|T \setminus S|} f(S)$, or if only the size of S matters, $\sum_{s=0}^{n} (-1)^{n-s} \binom{n}{s} f(s)$. In some contexts we might use Stirling numbers, not binomial coefficients!

Some useful applications:

Graph coloring Let $I(S)$ count the number of independent sets contained in $S \subseteq V$ ($I(\emptyset) = 1$, $I(S) = I(S \setminus v) + I(S \setminus N(v))$). Let $c_k = \sum_{S \subseteq V} (-1)^{|V \setminus S|} I(S)$. Then V is k-colorable iff $v > 0$. Thus we can compute the chromatic number of a graph in $O^*(2^n)$ time.

Burnside's lemma Given a group G acting on a set X, the number of elements in X up to symmetry is

$$\frac{1}{|G|}\sum_{g\in G}|X^g|$$

with X^g the elements of X invariant under g. For example, if $f(n)$ counts "configurations" of some sort of length n, and we want to count them up to rotational symmetry using $G=\mathbb{Z}/n\mathbb{Z}$, then

$$g(n)=\frac{1}{n}\sum_{k=0}^{n-1}f(\gcd(n,k))=\frac{1}{n}\sum_{k\|n}f(k)\phi(n/k)$$

I.e. for coloring with c colors we have $f(k)=k^c$.

Relatedly, in Pólya's enumeration theorem we imagine X as a set of n beads with G permuting the beads (e.g. a necklace, with G all rotations and reflections of the n-cycle, i.e. the dihedral group D_n). Suppose further that we had Y colors, then the number of G-invariant colorings Y^X/G is counted by

$$\frac{1}{|G|}\sum_{g\in G}|Y|^{c(g)}$$

with $c(g)$ counting the number of cycles of g when viewed as a permutation of X. We can generalize this to a weighted version: if the color i can occur exactly r_i times, then this is counted by the coefficient of $t_1^{r_1}\dots t_n^{r_n}$ in the polynomial

$$Z(t_1,\dots,t_n)=\frac{1}{|G|}\sum_{g\in G}\prod_{m\geq 1}(t_1^m+\dots+t_n^m)^{c_m(g)}$$

where $c_m(g)$ counts the number of length m cycles in g acting as a permutation on X. Note we get the original formula by setting all $t_i=1$. Here Z is the cycle index. Note: you can cleverly deal with even/odd sizes by setting some t_i to -1.

Lucas Theorem If p is prime, then:

$$\frac{p^a}{k}\equiv 0(\mod p)$$

Thus for non-negative integers $m=m_kp^k+\ldots+m_1p+m_0$ and $n=n_kp^k+\ldots+n_1p+n_0$:

$$\frac{m}{n}=\Pi_{i=0}^{k}\frac{m_i}{n_i}\mod p$$

Note: The fraction's mean integer division.

Catalan Numbers - Number of correct bracket sequence consisting of n opening and n closing brackets.
The number of ways to completely parenthesize $n+1$ factors.
The number of triangulations of a convex polygon with $n+2$ sides (i.e. the number of partitions of polygon into disjoint triangles by using the diagonals).
The number of ways to connect the $2n$ points on a circle to form n disjoint i.e. non-intersecting chords.

$$C_n=\frac{1}{n+1}\binom{2n}{n}$$

$$C_0=1,\ C_1=1,\ C_n=\sum_{k=0}^{n-1}C_kC_{n-1-k}$$

Narayana numbers The number of expressions containing n pairs of parentheses, which are correctly matched and which contain k distinct nestings.

$$N(n,k)=\frac{1}{n}\frac{n}{k}\frac{n}{k-1}$$