Korea Aerospace University  
Team: Koala

Coach : Prof. Inbok Lee  
Contestant : Suhwan Cheon, Taehyun Kim, Dabeen Jeong

# Table of Contents

Table of Contents 1

Algorithm tip 2

Data Structures 3

Binary Index Tree 3

Trie 4

Suffix array 5

Graph Algorithms 6

Topology Sort(위상 정렬) 6

Disjoint Set 6

Kruskal MST 7

Floyd-Warshall 8

Dijkstra 9

Bellman-Ford 9

Eulerian curcuit(오일러 서킷) 10

2-sat, scc 10

SubNode counting 11

Bipartite Matching 12

Maximum Flow(Edmonds-Karp)

Mathematical 11

큰 수 곱셈 11

FFT(Fast Fourier Transform) 11

페르마 소정리 11

Geometry 14

다각형 넓이 구하기 14

ccw 14

벡터 클래스, 다각형 클리핑 15

점과 선분 사이의 거리(binary search) 16

Convex Hull 18

Dynamic Programming 23

행렬 거듭제곱 dp 23

Bitmasking dp(TSP) 23

Quantize dp(분할) 23

String 23

LCS(Longest Common Subsequence) 24

# Algorithm tip

1. 체계적인 접근을 위한 질문들

“알고리즘 문제 해결 전략” 책에서 발췌

* 비슷한 문제를 풀어본 적이 있던가?
* 단순한 방법에서 시작할 수 있을까? (brute force)
* 내가 문제를 푸는 과정을 수식화할 수 있을까? (예제를 직접 해결해보면서)
* 문제를 단순화할 수 없을까?
* 그림으로 그려볼 수 있을까?
* 수식으로 표현할 수 있을까?
* 문제를 분해할 수 있을까?
* 뒤에서부터 생각해서 문제를 풀 수 있을까?
* 순서를 강제할 수 있을까?
* 특정 형태의 답만을 고려할 수 있을까? (정규화)

2. dp 설계 팁

- n이 10,000 정도 된다 -> 2차원 10,000 \* 10,000 dp는 메모리 초과로 불가능하므로 1차원 dp이거나 그리디일 것임.

- 작은 문제가 중복이 되어서 중복된 문제의 정답이 모두 동일하다.(Optimal Substructure)

- 정답을 한 번 구했으면 어딘가에 정답을 메모한다.(Memoization)

# Data Structures

## Binary Index Tree(구간 합 트리)

구간의 합/곱/최소/최대를 log(n)에 구해줄 수 있는 자료구조

build() : 초기 tree 생성

update(idx,val) : idx 값 val로 변경, 트리를 올라가며 구간의 값도 갱신(log(n))

query(l,r) : [l,r) 구간의 합을 구해줌.(log(n))

const int MAX = 2e6;

const int mod = 1e9 + 7;

ll tree[MAX];

void build() {

for (int i = n - 1; i > 0; i--) {

tree[i] = tree[i << 1] \* tree[(i << 1) | 1];

tree[i] %= mod;

}

}

void update(int idx, int val) {

idx += n - 1;

tree[idx] = val;

while (idx > 1) {

tree[idx / 2] = tree[idx] \* tree[idx ^ 1];

tree[idx / 2] %= mod;

idx >>= 1;

}

}

int query(int l, int r) { //[ㅣ,r)구간의 합 또는 곱을 구해줄 수 있음. 여기선 곱

ll ret = 1;

l += n - 1, r += n - 1;

while (l < r) {

if (l & 1) {

ret \*= tree[l];

ret %= mod;

l += 1;

}

if (r & 1) {

r -= 1;

ret \*= tree[r];

ret %= mod;

}

l >>= 1; r >>= 1;

}

return ret;

}

## Trie(트라이)

문자열을 저장하고 효율적으로 탐색하기 위한 트리 형태의 자료구조

Insert : 트라이에 문자열 추가

Find : 트라이에 해당 문자열이 있는지 확인

int t; //test case

int n; //call number

const int NUMBER = 10; // '0' ~ '9'

int toNumber(char ch) { return ch - '0'; }

struct TrieNode {

TrieNode\* children[NUMBER];

bool terminal;

//생성, 소멸자

TrieNode() : terminal(false) {

memset(children, 0, sizeof(children));

}

~TrieNode() {

for (int i = 0; i < NUMBER; i++)

if (children[i]) delete children[i];

}

//트라이에 번호 추가

void insert(const char\* key) {

if (\*key == 0) terminal = true;

else {

int next = toNumber(\*key);

if (children[next] == NULL)

children[next] = new TrieNode();

children[next]->insert(key + 1);

}

}

//트라이에 번호 일치 확인

TrieNode\* find(const char\* key) {

if (\*key == 0) return this;

int next = toNumber(\*key);

if (children[next] == NULL) return NULL;

return children[next]->find(key + 1);

}

};

int main()

{

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

cin >> t;

while (t--) {

cin >> n;

vector<string>input; //전화번호 길이 순 정렬

for (int i = 0; i < n; i++) {

char buf[11];

cin >> buf;

input.push\_back(buf);

}

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

TrieNode\* trie = new TrieNode();

bool check = false;

for (int i = 0; i < input.size(); i++) {

TrieNode\* node = trie->find(input[input.size() - 1 - i].c\_str());

if (node != NULL) check = true;

trie->insert(input[input.size() - 1 - i].c\_str());

}

if (check) cout << "NO\n";

else cout << "YES\n";

}

}

## Suffix Array & LCP(Longest Common Prefix)

Suffix array : 사전순으로 정렬된 모든 suffix의 시작 인덱스 배열

LCP: 인접한 접미사들이 겹치는 길이.

가장 긴 공통 문자열 = suffixarray[i~max(LCP[i])]

O(n\*log(n)^2)

const int MAX = 1e6;

string S;

int n, gap, suffix\_array[MAX], group[MAX];

int temp[MAX];

int LCP[MAX];

bool comp(int i, int j) {

//같은 그룹일 때

if (group[i] != group[j]) return group[i] < group[j];

//그룹이 다를 때

i += gap; j += gap;

return (i < n && j < n) ? (group[i] < group[j]) : (i > j);

}

void build\_suffix\_array() {

n = S.size();

//초기화

for (int i = 0; i < n; i++) {

suffix\_array[i] = i;

group[i] = S[i]; //처음엔 첫 글자로 비교하기 위해

}

for (gap = 1;; gap \*= 2) {

// gap 글자만큼만 보고 정렬

sort(suffix\_array, suffix\_array + n, comp);

memset(temp, 0, sizeof(int)\*MAX);

//그룹 넘버링

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

temp[i + 1] = temp[i] + comp(suffix\_array[i], suffix\_array[i + 1]);

for (int i = 0; i < n; i++) group[suffix\_array[i]] = temp[i]; //복사

if (temp[n - 1] == n - 1) break; //모두 그룹이 나눠졌으면 종료

}

}

void build\_LCP() {

for (int i = 0, k = 0; i < n; i++, k = max(k - 1, 0)) {

if (group[i] == n - 1) continue;

for (int j = suffix\_array[group[i] + 1]; S[i + k] == S[j + k]; k++); //j = 다음 그룹의 suffix array

LCP[group[i] + 1] = k;

}

}

int main() {

cin >> S;

build\_suffix\_array();

build\_LCP();

//print();

}

# Graph Algorithms

## Topology Sort(위상 정렬)

Directed acyclic graph에서 가능한 위상 정렬 순서를 구해줌.

순서는 res에 담기고 true를 반환

그러한 순서가 존재하지 않을 때 false를 반환.

Bfs와 들어가는 간선의 차수를 이용하여 구현.

const int Max = 501;

int inDegree[Max];

int n;

vector<int>a[Max];

vector <int>res;

bool tpsort() {

queue<int>q;

for (int i = 1; i <= n; i++)

if (inDegree[i] == 0)

q.push(i);

for (int i = 1; i <= n; i++) {

if (q.empty())

return false;

int from = q.front();

q.pop();

res.push\_back(from);

for (int i = 0; i < a[from].size(); i++) {

int to = a[from][i];

inDegree[to] -= 1;

if (inDegree[to] == 0)

q.push(to);

}

}

return true;

}

## 유니온 파인드(Disjoint-set)

교집합이 없는 집합들로 분류.

Find(idx) : idx번째의 부모를 리턴.

Merge(idx1,idx2) : 부모가 다르면 idx1의 집합을 idx2의 집합에 귀속시킴.

const int MAX;

class Set {

private:

int parent[MAX];

public:

Set() {

for (int i = 0; i < MAX; i++)

parent[i] = i;

}

int find(int idx) {

if (idx == parent[idx]) return idx;

return parent[idx] = find(parent[idx]);

}

void merge(int idx1, int idx2) {

int p1, p2;

p1 = find(idx1); p2 = find(idx2);

if (p1 == p2) return;

parent[p1] = p2;

}

};

## Kruskal MST

최소스패닝트리의 total edge cost를 구해줌

Edge cost로 정렬하고 disjoint set으로 연결 여부를 확인하며 node들을 n-1개의 간선으로 연결하면 mst가 완성되고 그 비용을 구할 수 있음.

#include <bits/stdc++.h>

#define pii pair<int,int>

#define pipi pair<pair<int,int>,pair<int,int> >

#define fs first

#define sc second

#define sorta(a) sort(a.begin(),a.end());

typedef long long ll;

using namespace std;

//typedef vector<vector<int> > matrix;

#define pll pair<ll,ll>

#define ppl pair<ll,pair<ll,ll>>

const int INF = (int)1e9 + 10;

const int MAX = 100001;

int V, E;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// 여기에 disjoint set class 선언

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

cin >> V >> E;

vector<ppl> edges;

for (int i = 0; i < E; i++) {

int a, b, c;

cin >> a >> b >> c;

edges.push\_back({ c,{a,b} });

}

//sort by cost

sorta(edges);

//use disjoint set to check if node is connected

Set \*s = new Set();

ll ansCost = 0;

//\*\*\*iterate for all edges ('E' times) \*\*\*

for (int i = 0; i < E; i++) {

int n1, n2, c;

n1 = edges[i].sc.fs; n2 = edges[i].sc.sc; c = edges[i].fs;

int p1, p2;

p1 = s->find(n1); p2 = s->find(n2);

if (p1 == p2) continue;

s->merge(n1, n2);

ansCost += c;

}

cout << (ll)cost << "\n";

}

## Floyd-Warshall

모든 노드에서 모든 노드로의 최단 거리를 구해줌.

O(n^3)

const int MAX = 101;

int graph[MAX][MAX];

int V, E;

void floyd() {

for (int k = 1; k <= V; k++) {

for (int i = 1; i <= V; i++) {

for (int j = 1; j <= V; j++) {

if (k == i || k == j || i == j) continue;

graph[i][j] = min(graph[i][j], graph[i][k] + graph[k][j]);

}

}

}

}

## Dijkstra

어떤 노드에서 다른 모든 노드로의 최단거리를 구해줌

Priority\_queue를 이용해서 그리디하게 최단거리 갱신.

const int MAX = 10100;

int n, m;

vector<pii> graph[MAX];

vector<int> mincost; //src에서 각 노드까지 최대 비용.

bool visited[MAX];

int trace[MAX]; //추적?

//가중치 그래프에서 출발지에서 각 노드까지의 최단 거리.

void mincost\_bfs(int src) { //==dijkstra algorithm ==priority bfs

priority\_queue<pii, vector<pii>, greater<pii> > pq; pq.push({ 0,src }); //use min heap.

//각 노드까지 거리를 무한대로 초기화.

mincost = vector<int>(n, 1e9);

mincost[src] = 0;

while (not pq.empty()) {

int here = pq.top().sc, cost = pq.top().fs; pq.pop();

//갱신할 비용이 현재 비용보다 크면 그건 고려할 필요도 없다.

if (mincost[here] < cost) continue;

for (int i = 0; i < graph[here].size(); i++) {

int nxt = graph[here][i].sc, nxtcost = cost + graph[here][i].fs;

if (nxtcost < mincost[nxt]) {

mincost[nxt] = nxtcost;

pq.push({ nxtcost,nxt });

trace[nxt] = here;

}

}

}

}

## BellmanFord

음의 간선이 있을 때 최단 경로를 찾는 알고리즘

만약 음의 사이클이 있다면, 비어있는 벡터를 리턴하게 된다.

#include <iostream>

#include <vector>

#include <cstring>

#include <algorithm>

using namespace std;

vector<pair<int, int> >adj[510];

int tc, n, m, w;

const int INF = 987654321;

vector<int> bellmanFord(int src) {

vector<int> upper(n + 1, INF);

upper[src] = 0;

bool updated = false;

for (int iter = 0; iter < n; iter++) {

updated = false;

for (int here = 1; here <= n; here++) {

for (int i = 0; i < adj[here].size(); i++) {

int there = adj[here][i].first;

int cost = adj[here][i].second;

if (upper[there] > upper[here] + cost) {

upper[there] = upper[here] + cost;

updated = true;

}

}

}

//모든 간선에 대해서 완화 실패시 곧장 종료한다.

if (!updated) break;

}

//n번째 순회에 대해서도 완화가 성공했다면 음수 사이클이 존재한다.

if (updated) upper.clear();

return upper;

}

int main()

{

scanf("%d", &tc);

while (tc--) {

//input data

for (int i = 0; i < 510; i++) adj[i].clear();

scanf("%d %d %d", &n, &m, &w);

for (int i = 0; i < m; i++) {

int s, e, t;

scanf("%d %d %d", &s, &e, &t);

adj[s].push\_back(make\_pair(e, t));

adj[e].push\_back(make\_pair(s, t));

}

for (int i = 0; i < w; i++) {

int s, e, t;

scanf("%d %d %d", &s, &e, &t);

adj[s].push\_back(make\_pair(e, -t));

}

vector<int> ans = bellmanFord(1);

if (ans.size() == 0) printf("YES\n");

else printf("NO\n");

}

return 0;

}

## Eulerian Curcuit(오일러 서킷)

한붓그리기.

오일러 서킷 존재 조건: 모든 노드의 차수가 짝수 or 2개의 노드의 차수가 홀수이고 다른 모든 노드의 차수가 짝수

Dfs 이용하여 재귀적으로 서브서킷들을 합쳐나가는 방식으로 구현.

Ans[0] != ans.back()이면 존재하지 않는 것.

const int MAX = 1100;

int graph[MAX][MAX];

int n;

vector<int> ans;

void dfs(int here) {

for (int to = 0; to < n; to++) {

if (to == here) continue;

if (graph[here][to] == 0) continue;

graph[here][to]--;

graph[to][here]--;

dfs(to);

}

ans.push\_back(here); //반환시 추가하는게 핵심.

}

## 2-SAT, SCC

SCC를 만들고, 위상정렬을 통해 2-SAT 해결

Tarjan 알고리즘으로 SCC만들기(위상정렬이 역순으로 됨)

SCC가 만들어 졌을 때, 같은 SCC안에 Xn과 ~Xn이 있는지 없는지 판단(없다면 SATISABLE)

#include <iostream>

#include <vector>

#include <cstring>

#include <stack>

#include <algorithm>

using namespace std;

int n, m, a, b, cnt;

// 그래프의 인접 리스트 표현

vector<vector<int>> adj(20004);

// 각 정점의 컴포넌트 번호. 컴포넌트 번호는 0부터 시작하며

// 같은 강결합 컴포넌트에 속한 정점들의 컴포넌트 번호가 같다.

vector<int> sccID;

// 각 정점의 발견 순서

vector<int> discovered;

// 정점의 번호를 담는 스택

stack<int> st;

int sccCounter, vertexCounter;

// here를 루트로 하는 서브트리에서 역방향 간선이나 교차 간선을

// 통해 갈 수 있는 정점 중 최소 발견 순서를 반환한다.

// (이미 SCC로 묶인 정점으로 연결된 교차 간선은 무시한다.)

int scc(int here) {

// 이 코드에서 here과 there은 u,v 즉 연결된 정점이라고 생각하면 편하다

int ret = discovered[here] = vertexCounter++;

// 스택에서 here를 넣는다. here의 후손들은 모두 스택에서 here 후에 들어간다.

st.push(here);

for (int i = 0; i < adj[here].size(); ++i) {

int there = adj[here][i];

// (here, there)가 트리 간선

if (discovered[there] == -1) ret = min(ret, scc(there)); // 아직 방문하지 않았다면

// there가 무시해야 하는 교차 간선이 아니라면

else if (sccID[there] == -1) {

ret = min(ret, discovered[there]);

}

}

// here에서 부모로 올라가는 간선을 끊어야 할지 확인한다.

if (ret == discovered[here]) {

//here를 루트로 하는 서브트리에 남아 있는 정점들을 전부 하나의 컴포넌트로 묶는다.

while (true) {

int t = st.top();

st.pop();

sccID[t] = sccCounter; // sccID의 각 index에는 각 정점이 몇 번째 컴포넌트인지 들어있다.

cnt = sccCounter;

if (t == here) break;

}

++sccCounter;

}

return ret;

}

// tarjan의 SCC알고리즘

vector<int> tarjanSCC() {

// 배열들을 전부 초기화

sccID = discovered = vector<int>(adj.size(), -1);

// 카운터 초기화

sccCounter = vertexCounter = 0;

// 모든 정점에 대해 scc() 호출

for (int i = 1; i <= 2 \* n; i++)if (discovered[i] == -1) scc(i);

return sccID;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cin >> n >> m;

for (int i = 0; i < m; i++) {

cin >> a >> b;

// a u b 라면 ~a -> b 와 ~b -> a 이 두 간선을 추가해야 한다.

adj[a > 0 ? n + abs(a) : abs(a)].push\_back(b > 0 ? b : n + abs(b));

adj[b > 0 ? n + abs(b) : abs(b)].push\_back(a > 0 ? a : n + abs(a));

}

// tarjan 알고리즘으로 구했기 때문에 위상정렬의 역순으로 구해져 있다.

vector<int> temp = tarjanSCC();

// x와 ~x가 한 scc안에 있는지 없는지 판단 -> 없다면 SAT

// 간단한 정리 : p -> q라는 명제에서 p가 거짓이면 q는 참이든 거짓이든 상관없음. but p가 참이면 q도 반드시 참이어야 함!

// 그렇다면 한 scc안에 하나라도 true라면 전부 true여야 함. -> 이 때 x와 ~x가 같은 scc안에 있다면 위의 조건 불만족

// 즉, 한 scc안에 자신과 반대되는 명제가 없다면 SAT

int stop = 0;

for (int i = 1; i <= n; i++) {

if (temp[i] == temp[n + i]) {

stop = 1;

break;

}

}

if (stop == 1) {

cout << "0" << "\n";

return 0;

}

else cout << "1" << "\n";

// 이 문제의 해결방법

// 위상정렬 -> 먼저 만나는 xi 또는 ~xi 를 false로 설정! (역은 true로 설정)

// 즉, 먼저 위상정렬 된 상태로 바꿔야 한다.

vector<pair<int, int>>correct\_scc;

for (int i = 1; i <= 2 \* n; i++) correct\_scc.push\_back(make\_pair(temp[i], i));

sort(correct\_scc.begin(), correct\_scc.end(), greater<pair<int, int>>()); // 정렬을 통해 드디어 제대로 된 위상정렬 순서를 얻었다!

vector<int>ans(2 \* n + 1, -1);

for (int i = 0; i < correct\_scc.size(); i++) {

int x = correct\_scc[i].second;

if (ans[x] == -1) { // 아직 true인지 false인지 모를 때 (방문하지 않았을 때), 자신은 false, 역은 true

ans[x] = 0;

ans[x <= n ? n + x : x - n] = 1;

}

}

for (int i = 1; i <= n; i++) {

cout << ans[i] << " ";

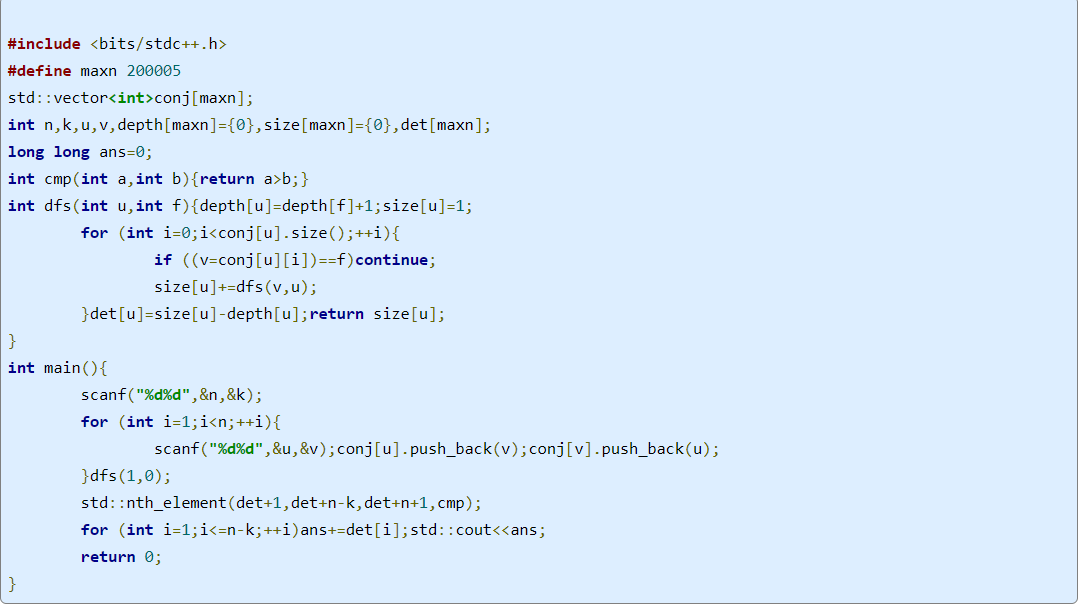
}

cout << "\n";

}

## Subnode counting

트리에서 각 노드의 서브노드 개수 빠르게 구하기



## Bipartite Matching

이분 매칭

이분 그래프에서 A 그룹의 정점에서 B 그룹의 정점으로 간선을 연결 할 때,

A그래프의 하나의 정점이 B그래프의 하나의 정점만 가지도록 구성된 것.

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

int c[55][55]; //capacity

int f[55][55]; //flow

vector<int>v[55];

int main()

{

cin.sync\_with\_stdio(0); cin.tie(0);

int n;

cin >> n;

for (int i = 0; i < n; i++)

{

char s, t; int val;

cin >> s >> t >> val;

int src, dst;

if ('A' <= s && s <= 'Z') src = s - 'A';

else src = s - 'a' + 26;

if ('A' <= t && t <= 'Z') dst = t - 'A';

else dst = t - 'a' + 26;

c[src][dst] += val; c[dst][src] += val;

v[src].push\_back(dst);

v[dst].push\_back(src);

}

int total = 0, src = 0, dst = 'Z' - 'A';

while (1)

{

int dist[55];

memset(dist, -1, sizeof(dist));

queue<int> q;

q.push(src);

while (!q.empty())

{

int here = q.front();

q.pop();

for (int i = 0; i < v[here].size(); i++)

{

int next = v[here][i];

if (dist[next] != -1) continue;

if (c[here][next] - f[here][next] > 0)

{

q.push(next);

dist[next] = here;

if (next == dst) break;

}

}

}

if (dist[dst] == -1) break;

int flow = 987654321;

for (int i = dst; i != src; i = dist[i]) {

flow = min(flow, c[dist[i]][i] - f[dist[i]][i]);

}

for (int i = dst; i != src; i = dist[i]) {

f[dist[i]][i] += flow;

f[i][dist[i]] -= flow;

}

total += flow;

}

cout << total << "\n";

return 0;

}

## Maximum Flow (by Edmonds-Karp Algorithm)

네트워크 플로우

용량을 갖는 그래프에서 두 정점 사이에 얼마나 많은 유량을 보낼 수 있는 가

#define MAX 52

#define INF 987654321

using namespace std;

int main()

{

int n;

int c[MAX][MAX]; // capacity :: i->j로 가는 간선 용량

int f[MAX][MAX]; // flow :: i->j로 현재 흐르고 있는 유량

memset(c, 0, sizeof(c));

memset(f, 0, sizeof(f));

vector<int> adj[MAX];

// 입력

scanf("%d", &n);

for (int i = 0; i < n; i++)

{

char chfrom, chto;

int val;

// \n 때문에 " %c로 받는다.

scanf(" %c %c %d", &chfrom, &chto, &val);

// 입력받은 문자를 0~51사이 값으로 변환

int from, to;

if ('A' <= chfrom && chfrom <= 'Z')

from = chfrom - 'A';

else

from = chfrom - 'a' + 26;

if ('A' <= chto && chto <= 'Z')

to = chto - 'A';

else

to = chto - 'a' + 26;

c[from][to] += val; // 용량을 추가해준다.

c[to][from] += val;

adj[from].push\_back(to);

adj[to].push\_back(from); // 역방향 간선 추가

}

int totalFlow = 0, S = 0, T = 'Z' - 'A'; // S :: source, T :: sink

// 증가 경로를 못 찾을 때까지

while (1)

{

int prev[MAX];

memset(prev, -1, sizeof(prev));

queue<int> q;

// 싱크를 처음 push 해준다.

q.push(S);

while (!q.empty())

{

int cur = q.front();

q.pop();

for (int i = 0; i < adj[cur].size(); i++)

{

int next = adj[cur][i];

// next를 방문하였다면 continue

if (prev[next] != -1)

continue;

// 아직 흐를 수 있는 유량이 있다면

// 즉, cur -> next로 갈 수 있는 유량이 있다면

if (c[cur][next] - f[cur][next] > 0)

{

q.push(next);

// next의 경로 추적을 위해 저장

prev[next] = cur;

// 만약 next가 sink면 break

if (next == T)

break;

}

}

}

// source -> sink의 간선이 없다면 break;

if (prev[T] == -1)

break;

int flow = INF;

// 지금까지 구한 경로 상에서 가장 flow가 낮은 곳을 구한다.

for (int i = T; i != S; i = prev[i])

flow = min(flow, c[prev[i]][i] - f[prev[i]][i]);

// 해당 경로에 flow를 보낸다.

// 그리고 역방향으로는 flow를 빼준다.

for (int i = T; i != S; i = prev[i])

{

f[prev[i]][i] += flow;

f[i][prev[i]] -= flow;

}

totalFlow += flow;

}

printf("%d\n", totalFlow);

return 0;

}

# Mathematical

## 큰 수 곰셈

큰 수의 곱셈을 vector를 이용하여 오버플로우 없이 할 수 있음.

O(n^2)

//낮은 자리수부터 벡터에 담아야 계산 구현하기 편함

vector<int> stoVec(string s) {

vector<int> ret;

for (int i = s.size() - 1; i >= 0; i--)

ret.push\_back(s[i] - '0');

return ret;

}

//자리 올림 처리

void normalize(vector<int>& C) {

C.push\_back(0);

for (int i = 0; i < C.size() - 1; i++) {

if (C[i] < 0) {

int borrow = (abs(C[i] + 9)) / 10;

C[i + 1] -= borrow;

C[i] += 10 \* borrow;

}

else {

C[i + 1] += (C[i] / 10);

C[i] %= 10;

}

}

while (C.size() > 1 && C.back() == 0) C.pop\_back();

}

vector<int> multiply(vector<int>& A, vector<int>& B) {

vector<int> C(A.size() + B.size() + 1, 0);

for (int i = 0; i < A.size(); i++) {

for (int j = 0; j < B.size(); j++) {

C[i + j] += A[i] \* B[j];

}

}

normalize(C);

return C;

}

int main() {

string a, b;

cin >> a >> b;

vector<int> A = stoVec(a);

vector<int> B = stoVec(b);

vector<int> C = multiply(A, B); // 낮은 자리수부터 담겨있다.

for (int i = C.size() - 1; i >= 0; i--)

cout << C[i];

cout << "\n";

}

## FFT(Fast Fourier Transform)

고속 푸리에 변환 – 다항식의 곱셈을 O(nlogn)에 가능!

Multyply의 결과(ret)가 다항식의 곱셈 결과임

#include <iostream>

#include <complex>

#include <vector>

#include <cstring>

#include <string>

#include <algorithm>

using namespace std;

typedef complex<double> base;

typedef long long ll;

const double PI = acos(-1);

void fft(vector<base>& a, bool inv = false) {

int n = a.size(), j = 0;

// 분할 정복 기법, 이때 n은 무조건 2^n 승임

vector<base> roots(n / 2);

for (int i = 1; i < n; i++) {

int bit = (n >> 1);

while (j >= bit) {

j -= bit;

bit >>= 1;

}

j += bit;

if (i < j) swap(a[i], a[j]);

}

double ang = 2 \* acos(-1) / n \* (inv ? -1 : 1);

for (int i = 0; i < n / 2; i++) {

roots[i] = base(cos(ang \* i), sin(ang \* i));

}

for (int i = 2; i <= n; i <<= 1) {

int step = n / i;

for (int j = 0; j < n; j += i) {

for (int k = 0; k < i / 2; k++) {

base u = a[j + k], v = a[j + k + i / 2] \* roots[step \* k];

a[j + k] = u + v;

a[j + k + i / 2] = u - v;

}

}

}

if (inv) for (int i = 0; i < n; i++) a[i] /= n;

}

vector<ll> multiply(vector<ll>& v, vector<ll>& w) {

vector<base> fv(v.begin(), v.end()), fw(w.begin(), w.end());

// n이 무조건 2^n 이여야 하기 때문에 변환!

int n = 2; while (n < v.size() + w.size()) n <<= 1;

fv.resize(n); fw.resize(n);

fft(fv, 0); fft(fw, 0);

for (int i = 0; i < n; i++) fv[i] \*= fw[i];

fft(fv, 1);

vector<ll> ret(n);

for (int i = 0; i < n; i++) ret[i] = (ll)round(fv[i].real());

return ret;

}

int main() {

ios\_base::sync\_with\_stdio(false);

cin.tie(NULL);

cout.tie(NULL);

vector<ll> a, b;

vector<ll> ret = multiply(a, b);

}

## 페르마 소정리

매우 큰 이항계수를 구하기 위한 방법

nCk = n! / (k! \* (n – k)!), 분자, 분모를 각각 A, B라 하면

페르마 소정리에 의해 (AB^-1) % p = ((A % p)(B^(p-2) % p)) % p

시간 복잡도 : O(lgn)

#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

ll n, k;

const ll MOD = 1000000007;

ll f[4000001];

int main()

{

ios\_base::sync\_with\_stdio(0); cin.tie(0);

cin >> n >> k;

f[0] = 1;

f[1] = 1;

for (ll i = 2; i <= n; i++)

{

f[i] = (i \* f[i - 1]) % MOD;

}

// a = f[n]

// b = f[k] \* f[n-k]

ll a = f[n];

ll b = (f[k] \* f[n - k]) % MOD;

//b ^ (p-2)를 구하는 과정 필요

ll c = 1;

int cnt = MOD - 2;

while (cnt > 0)

{

if (cnt % 2 == 1)

{

c = (c \* b) % MOD;

}

b = (b \* b) % MOD;

cnt /= 2;

}

ll ans = (a \* c) % MOD;

cout << ans << "\n";

return 0;

}

# Geometry

## 다각형 넓이 구하기

다각형을 외부의 한 점과 다각형의 인접한 두 꼭지점들을 한 방향으로 돌리면서 외적

한 것을 더하면 다각형의 면적을 구할 수 있음.

typedef long long ll;

#define pll pair<ll,ll>

vector<pll> a(n); //다각형을 이루는 꼭지점들의 좌표(점벡터). 정수 좌표에 대해서 구현.

for (int i = 0; i < n; i++) {

cin >> a[i].fs >> a[i].sc;

}

ll tot = 0;

//외적을 다 합한 뒤에 절대값 취하고 2로 나누면 넓이.

for (int i = 0; i < n; i++) {

tot += (a[i].fs\*a[(i + 1) % n].sc - a[i].sc\*a[(i + 1) % n].fs);

}

## Ccw

세 점 a,b,c를 가르키는 세 벡터가 있을 때,

a,b,c를 순서대로 잇는 두 선분은 b에서 어느방향으로 꺽을지

ccw(a,b) : 원점에서 벡터b가 벡터 a의 반시계 방향이면 양수, 시계방향이면 음수, 평행이면 0을 리턴

ccw(p,a,b): 점 p를 기준으로 벡터 b가 벡터 a의 반시계 방향이면 양수, 시계방향이면 음수, 평행이면 0을 리턴

double ccw(VECTOR a,VECTOR b){

return a.cross\_product(b);

}

double ccw(VECTOR p,VECTOR a,VECTOR b){

return ccw(a-p,b-p);

}

## 벡터 클래스, 다각형 클리핑

<내적의 용도>

식 : a(\*)b = a.x\*b.x + a.y+b.y , '(\*)' means inner product

= norm(a) \* norm(b) \*cos(theta)

1. 벡터의 사이각 구하기( theta = acos((a(\*)b)/(norm(a)\*norm(b))) )

2. 벡터의 직각 여부 확인 (직각 iff 내적값 ==0)

< 외적의 용도>

식 : a x b = a.x\*b.y -a.y\*b.x; , 'x' means cross product

= norm(a) \* norm(b) \*sin(theta) ,

theta: a에서 b까지 반시계 방향으로의 각도

1. 두 벡터가 이루는 면적 계산하기 ::외적/2 ==두 벡터와 원점이 이루는 삼각형의 넓이

2. 두 벡터의 방향 판별 : a x b =-(b x a) ~> CCW check 가능

<점과 선 사이의 거리>

Perpendicular(p,a,b) : 점 p에서 (a,b) 직선에 내린 수선의 발을 구한다.

pointToLine(p,a,b) : 점 p와 (a,b)가 이루는 직선 사이의 거리를 구한다.

<두 직선의 교차점 구하기>:: bool lineIntersection(a,b,c,d,ret)

두 직선의 교차점

입력받는 a,b,c,d를 b-a로,d-c로 만든게 B,D.

ret : 구하려는 교차점

<두 다각형의 교집합 구하기>

polygon intersection(polygon p,다각형을 이루는 꼭지점의 좌표들)

::sutherland-hodgman alogrithm

:: 다각형1의 선분을 포함하는 직선으로 반평면 만들어가면서 그 반평면에 포함 안 되는 다각형2의 꼭지점들 걸러내기

==> 최종적으로 남는 것이 다각형1과 다각형2의 교집합 꼭지점들.

(다각형 클리핑(잘라내기)라고 함 )

struct VECTOR {

double x, y;

explicit VECTOR(double x\_ = 0, double y\_ = 0) : x(x\_), y(y\_)

{}

VECTOR operator + (const VECTOR& rhs) const { //벡터 덧셈

return VECTOR(x + rhs.x, y + rhs.y);

}

VECTOR operator - (const VECTOR& rhs) const { //벡터 뺄셈

return VECTOR(x - rhs.x, y - rhs.y);

}

VECTOR operator \* (double rhs) const { //벡터 실수배)

return VECTOR(x\*rhs, y\*rhs);

}

double norm() const { return hypot(x, y); } //벡터 길이

VECTOR normalize() const { return VECTOR(x / norm(), y / norm()); }

//x축의 양의 방향으로부터 이 벡터까지 반시계 방향으로 잰 각도

double polar() const { return fmod(atan2(y, x) + 2 \* PI, PI); }

//atan2 : arctan , fmod: 실수에 대한 나머지연산

//내적(inner product,dot product)

double dot\_product(const VECTOR& rhs) const { return x \* rhs.x + y \* rhs.y; }

//외적(cross product)

double cross\_product(const VECTOR& rhs) const { return x \* rhs.y - rhs.x\*y; }

//이 벡터(this class,오브젝트)를 rhs에 사영한 결과를 반환

//사영 = norm(a) \*cos(theta) = ( a(\*)b ) / norm(b)

VECTOR project(const VECTOR& rhs) const {

VECTOR r = rhs.normalize(); //길이는 1로, 방향만 남기기.

return r \* r.dot\_product(\*this);

}

};

//수선의 발

VECTOR perpendicular(VECTOR p,VECTOR a,VECTOR b){

return a+(p-a).project(b-a);

}

//점과 직선 사이의 거리

double pointToLine(VECTOR p,VECTOR a,VECTOR b){

return (p-perpendicular(p,a,b)).norm();

}

bool lineIntersection(VECTOR a, VECTOR b, VECTOR c, VECTOR d, VECTOR& ret) {

double determinant = (b - a).cross\_product(d - c);

//check if parrallel

if (fabs(determinant) < DBL\_EPSILON) return false;

//These are parrallel

ret = a + (b - a)\*((c - a).cross\_product(d - c) / determinant);

return true; //found intersection.

}

typedef vector<VECTOR> polygon;

//왼쪽 반평면(반시계방향 순회 기준)과 다각형의 교집합을 구한다.

//(a,b)를 포함하는 직선으로 다각형 p를 자른 뒤, (a,b)의 왼쪽에 있는 부분

polygon cutPoly(const polygon& p, const VECTOR& a, const VECTOR& b) {

int n = p.size();

//p의 각 점이 반평면 내에 있는지 여부.

vector<bool> inside(n, false);

for (int i = 0; i < n; i++) inside[i] = ccw(a, b, p[i]) >= 0; //직선(a,b)의 왼편에 있는지 여부::반평면 내에 있는지.

polygon ret;

for (int from = 0; from < n; from++) {

int to = (from + 1) % n;

//반평면 내에 있는점은 항상 결과 다각형에 포함.

if (inside[from]) ret.push\_back(p[from]);

//(p[from],p[to]) 선분이 직선(a,b)와 교차하면 교차점을 결과 다각형에 포함시킨다.

if (inside[from] != inside[to]) {

VECTOR cross; // cross: 구하려는 교차점

assert(lineIntersection(p[from], p[to], a, b, cross)); // false when parallel

ret.push\_back(cross);

}

}

return ret;

}

polygon intersection(const polygon& p, double x1, double y1, double x2, double y2) {

VECTOR a(x1, y1), b(x2, y1), c(x2, y2), d(x1, y2); //직사각형

polygon ret = cutPoly(p, a, b);

ret = cutPoly(ret, b, c);

ret = cutPoly(ret, c, d);

ret = cutPoly(ret, d, a);

return ret;

}

## 점과 선분 사이의 거리 by binary search

Case 1: 수선의 발이 선분 위에 떨어질 때 -> 거리 = 수선의 발 벡터 길이

Case 2: 수선의 발이 선분 밖에 있을 때 -> 거리 = 양 끝점 중 가까운 곳과의 거리

double pointToSegment2(VECTOR p, VECTOR a, VECTOR b) {

for (int i = 0; i < 100; i++) {

double pa = (p - a).norm(), pb = (p - b).norm();

if (pa > pb) a = (a + b)\*0.5;

else b = (a + b)\*0.5;

}

return (p - a).norm();

}

## Convex Hull

볼록껍질: 주어진 점들을 모두 포함하는 최소 크기의 다각형

볼록껍질 찾는 알고리즘 중 하나인 gift wrapping algorithm.

:가장 왼쪽 아래 점을 시작으로 시계방향으로 돌리면서 가장 왼쪽에 있으면서 가장 먼 점을 택하면서 껍질을 만들어간다.

polygon giftWrap(const vector<VECTOR>& points)

:points에 있는 점들을 모두 포함하는 최소의 볼록 다각형을 찾는다.

찾은 다각형의 모든 꼭지점을 담은 polygon을 반환

O(N^2)

polygon giftWrap(const vector<VECTOR>& points) { //O(n^2)

int n = points.size();

polygon hull;

//start : 가장 왼쪽 아래 점(볼록껍질에 항상 포함되는 점)

VECTOR start = \*min\_element(all(points));

//비교자가 VECTOR 구조체에 구현되어 있다.

hull.push\_back(start);

while (true) {

//ccw(p,a,b) : 점 p를 기준으로 벡터 b가 벡터 a의 반시계 방향이면 양수, 시계방향이면 음수, 평행이면 0을 리턴

VECTOR ph = hull.back(), nxt = points[0];

for (int i = 1; i < n; i++) {

double cross = ccw(ph, nxt, points[i]);

double dist1, dist2;

dist1 = (nxt - ph).norm();

dist2 = (points[i] - ph).norm();

//더 왼쪽에 있는 것이 nxt가 되도록.

if (cross > 0) nxt = points[i];

//ph,nxt,points[i]가 일직선 상에 있으면(외적==0) 거리가 더 먼 것이 nxt가 되도록.

else if (cross == 0 && dist1 < dist2) {

nxt = points[i];

}

}

if (nxt == start) break;

hull.push\_back(nxt);

}

return hull;

}

//다각형과 다각형의 경계 위에 있지 않은 점 q가 주어질 때, q가 다각형의 내부/외부에 있는지 판별

bool isInside(VECTOR q, const polygon& p) {

int crosses = 0;

for (int i = 0; i < p.size(); i++) {

int j = (i + 1) % p.size();

//(p[i],p[j])가 q에서 오른쪽으로 가는 반직석을 세로로 가로지르는가.

if ((p[i].y > q.y) != (p[j].y > q.y)) {

//

double atX = (p[j].x - p[i].x) \* (q.y - p[i].y) / (p[j].y - p[i].y) + p[i].x; //비례식 :dx1:dy1 = dx2: dy2

if (q.x < atX) crosses++;

//x좌표 마저 내부이면 교차점으로 카운트.

}

}

if (crosses % 2) return true; //홀수번 교차 -> 내부 점

else return false; //짝수번 교차 -> 외부 점

}

//두 다각형이 서로 닿거나 겹치는지 여부 반환.

//한 점이라도 겹치면 return true

//isInside 함수만 써서 모든 변이 아닌 모든 점들을 검사하면 한 점만 겹치는 경우를 처리 못함.

bool polygonIntersects(const polygon& p, const polygon& q) {

int n = p.size(), m = q.size();

//p,q의 아무 점이나 잡았는데 포함되어 있으면 ==> 두 다각형은 겹친다.

if (isInside(p[0], q) || isInside(q[0], p)) return true;

//두 다각형의 모든 변들끼리 서로 겹치는지 검사.

for (int i = 0; i < n; i++) {

for (int j = 0; j < m; j++) {

f (segmentIntersects(p[i], p[(i + 1) % n], q[j], q[(j + 1) % m])) return true;

}

}

return false; //not intersects.

}

int main() {

int n, a, b, c;

polygon p1, p2;

for (int i = 0; i < n; i++) {

cin >> c >> a >> b;

if (c == 0) {

p1.push\_back(VECTOR(a, b));

}

else {

p2.push\_back(VECTOR(a, b));

}

bool isintersect = polygonIntersects(p1, p2);

}

# Dynamic Programming

## 행렬 거듭제곱 dp

10^18의 dp 점화식을 적절히 더하고 빼서 행렬로 표현하면

분할정복을 이용해 행렬 곱셈을 log 수준으로 해낼 수 있다.

typedef long long ll;

typedef vector<vector<ll> > matrix;

const ll mod = 1000000007;

matrix operator \* (const matrix& A, const matrix& B) {

int n = A.size();

matrix C(n, vector<ll>(n));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

for (int k = 0; k < n; k++) {

C[i][j] += A[i][k] \* B[k][j];

}

C[i][j] %= mod;

}

}

return C;

}

matrix I;

void make\_Identity(int n) {

I = matrix(n, vector<ll>(n));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

if (i == j) I[i][j] = 1;

else I[i][j] = 0;

}

}

}

matrix pow(matrix& A, ll m) {

if (m == 0) return I;

if (m % 2 > 0) return pow(A, m - 1)\*A;

//else

matrix half = pow(A, m / 2);

return half \* half;

}

int main() {

ll n; cin >> n;

matrix C(2, vector<ll>(2));

//C[0][0] = 0, C[0][1] = 1, C[1][0] = 1, C[1][1] = 1;

make\_Identity(2);

matrix M\_i\_1 = pow(C, n - 1);

ll ans = M\_i\_1[1][1];

cout << ans << '\n';

}

## Bitmasking dp (TSP problem)

N<=16에 대한 TSP

방문여부를 정수 하나의 bit로 표현

args : here = 현재위치 , visited = 방문 여부

return : here에서 시작하는 남은 부분 경로의 최소 길이 반환

const int MAX = 16, inf = 1e9;

int n, src; //방문지 수,출발지

int graph[MAX][MAX]; //방문지간 거리

int dp[MAX][1 << MAX]; //(현재 위치,이진수로 방문여부표시한 정수)

int shortestPath(int here, int visited) {

//base case

if (visited == (1 << n) - 1) return graph[here][src];

//(1<<n)-1 ==111...11 :: meaning visited all nodes.

int& ret = dp[here][visited];

if (ret != -1) return ret;

ret = inf;

for (int nxt = 0; nxt < n; nxt++) {

//방문한 노드면 skip

if (visited&(1 << nxt)) continue;

int candidatePath = graph[here][nxt] + shortestPath(nxt, visited | (1 << nxt)); //bit연산으로 방문처리하는 모습.

ret = min(ret, candidatePath);

}

return ret;

}

## Quantize dp(분할)

냅색 dp

int solve(int from, int parts) {

if (from == n) return 0;

if (parts == 0) return inf;

int& ret = cache[from][parts];

if (ret != -1) return ret;

ret = inf;

for (int partSize = 1; from + partSize <= n; partSize++)

//다양한 크기의 조각들로 나눠보며 최소값을 찾아서 cache[from][parts]에 저장한다.

ret = min(ret, minError(from, from + partSize - 1) + solve(from + partSize, parts - 1));

return ret;

}

# String

## LCS(Longest Common Subsequence)

Longest Common Subsequence(공통 부분 수열) 길이구하기

const int MAX = 5050;

int dp[MAX][MAX]; //lcs 길이 저장용

string s, p; //두 문자열 s,p

//두 문자열으로 2차원 테이블 만들었을 때,

for (int i = 0; i < s.size(); i++) {

for (int j = 0; j < p.size(); j++) {

//운좋게 딱 맞았네. 그럼 두 인덱스를 모두++ 하고 길이는 1증가하면 되지.

if (s[i] == p[j]) dp[i + 1][j + 1] = dp[i][j] + 1;

else {//다르네. 그럼 테이블의 위에서 오거나 왼쪽에서 온 것 중에 최댓값.

dp[i + 1][j + 1] = max(dp[i + 1][j], dp[i][j + 1]);

}

lcs = max(lcs, dp[i + 1][j + 1]);

}

}