# Team Note of (tmp)

roonm813, noonmap, gusah009

Compiled on October 9, 2020

## Contents



# 1 General

## 1.1 BFS

```
typedef struct{
  int self;
  vector<int> node; //인접한 노드들
}Node

Node nodes[N];

void BFS(int start){

  bool *visit = new bool[N];
  queue<Node> queue(N);

  //1. start node를 큐에 넣는다.
  queue.push(start);

  //2. 큐가 빌 때까지 계속한다.
  while(!queue.empty()){
    // 2-1. 큐에서 node 꺼냄.
    Node cur = queue.back();
    queue.pop();

    // 2-2. 큐에 (아직 방문 안 한) 인접한 node들을 넣음.
    for(int i=0; i<cur.node.size(); i++){
      Node nxt = cur.node[i];
      if( visit[ nxt.self ] == false ){
        visit[ nxt.self ] = true;
```

```
        queue.push( nxt );
      }
    }
  }
}
```

## 1.2 DFS with recusiveCall

```
/*
  Date : 2019 10 04
  Author : roonm813
  Subject : ACM-ICPC Team Note - DFS with recursive call
  Time Complexity : O(V+E)
*/

class Node{
public:
  //add field which you need
  vector<int> neighbors;
  bool visited;
  Node():visited(false){}
};


void DFS(int here, Node* nodes){
  cout << "DFS visits" << here << endl;
  for(int i = 0; i < nodes[here].neighbors.size(); i++){
    int next = nodes[here].neighbors.at(i);
    if(graphs[next].visited == false)
      DFS(next, nodes)
  }
}

int main(){
  //N = number of Node
  Node nodes[N + 1];
  //initailization edge infomation.

  //DFS all
  //만약 connected graph가 아닌 경우를 대비해 visited가 아니라면 해당 node를 시작점으로 방문해야함.
  for(int i = 1; i < N + 1; i++){
    if(graphs[N]->visited == false)
      DFS(i, graphs);
  }
  return 0;
}
```

## 1.3 DFS with stack

```
/*
  Date : 2019 10 04
  Author : roonm813
  Subject : ACM-ICPC Team Note - DFS with while loop. using Stack
  Time Complexity : O(V+E)
*/

/*  which is faster : implement DFS using recursive function calls or using stack?
  Implementation with stack is slightly faster than with recursive function! but almost same..
*/
class Node{
public:
  //add field which you need
  vector<int> neighbors;
  bool visited;
  Node():visited(false){}
};



void DFS(int start, Node* nodes){
    stack<int> nextVisits;
    nextVisits.push(start);
  nodes[start].visited = true;

    while(!nextVisits.empty()){
        int here = nextVisits.top();
```

```
        nextVisits.pop();

        for(int i = 0; i < nodes[here].neighbors.size(); i++){
            int next = nodes[here].neighbors.at(i);
            if(nodes[next].visited == false){
                nextVisits.push(next);
                nodes[next].visited = true;
            }
        }
    }
}


int main(){
  //N = number of Node
  Node* nodes[N + 1];
  //intialization edge information

  //DFS all
  //만약 connected graph가 아닌 경우를 대비해 visited가 아니라면 해당
  node를 시작점으로 방문해야함.
  for(int i = 1; i < N + 1; i++){
    if(graphs[N]->visited == false)
      DFS(i, graphs);
  }
  return 0;
}
```

## 1.4 Bellman Ford

```
// O(VE)
// 정점의 개수
int V;
// 그래프의 인접 리스트. (연결된 정점 번호, 간선 가중치) 쌍을 담는다.
vector<pair<int, int> > adj[MAX_V];
// 음수 사이클이 있을 경우 텅 빈 배열을 반환
vector<int> bellmanFord(int src) {
  // 시작점을 제외한 모든 정점까지의 최단 거리 상한을 INF로 둔다.
  vector<int> upper(V, INF);
  upper[src] = 0;
  bool updated;
  // V번 순회한다
  for (int it = 0; it < V; it++) {
    updated = false;
    for (int here = 0; here < V; here++) {
      for (int i = 0; i < adj[here].size(); i++) {
        int there = adj[here][i].first;
        int cost = adj[here][i].second;
        // (here, there) 간선을 따라 완화를 시도한다.
        if (upper[there] > upper[here] + cost) {
          // 성공
          upper[there] = upper[here] + cost;
          updated = true;
        }
      }
      // 모든 간선에 대해 완화가 실패했을 경우 V-1번 돌 필요도 없이 곧장
      ㅈ오료한다.
      if (!updated) break;
    }
    // V번째 순회에서도 완화가 성공했다면 음수 사이클이 있다는 뜻이다.
    if (updated) upper.clear();
    return upper;
  }
}
```

## 1.5 Dijkstra

```
// O(|E||log|V|)
// 알고스팟 최단거리문제에서 발췌함
// 마이너스로 저장하기 때문에 주의해야 됨!
FOR(i,101)
  FOR(j,101)
    adj[i][j] = -INF;

// adj[1][1] = 0;
// priority queue로 (최단거리, 시작지점, 끝지점) 저장
// 길이가 짧은 순으로 돌릴 예정이므로 마이너스로 저장.
// Ex) pq.top() =>  -1, -2, -3 vs 3, 2, 1 ...
pq.push(make_tuple(0,1,1)); // 1번부터 시작!
while(!pq.empty()) {
```

```
  int b = get<0>(pq.top()); // 최단거리
  int x = get<1>(pq.top()); // 시작지점
  int y = get<2>(pq.top()); // 끝지점
  if (x <= 0 || x > N || y <= 0 || y > M) { // 조건에서 벗어나면
    pq.pop();
    continue;
  }
  if (adj[x][y] >= b) { // 이미 설정된 최단거리보다 크면! (마이너스임)
    pq.pop();
    continue;
  }
  adj[x][y] = b; // 최단거리로 설정 후
  pq.pop();
  for (int i = 0; i < 4; i++) { // 인접한 노드를 전부 탐색하면서
  최단거리 재설정
    int move_x = x + dir[i][0];
    int move_y = y + dir[i][1];
    int move_b = (board[move_x][move_y] == true) ? b - 1 : b;
    pq.push(make_tuple(move_b, move_x, move_y));
  }
}
// 거의 최단거리의 경우 한번 최단거리를 구한 후 최단거리인 노드를 삭제하고
// 한번 더 다익스트라를 돌리면 된다!
```

## 1.6 Floyd Warshall

```
// 정점의 개수
int V;
// 그래프의 인접 행렬 표현
// adj[u][v] = u에서 v로 가는 간선의 가중치.
int adj[MAX_V][MAX_V];
// 플로이드의 모든 쌍 알고리즘
void floyd() {
  for (int i = 0; i < V; i++) adj[i][i] = 0;
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        adj[i][j] = min(adj[i][j], adj[i][k] + adj[k][j]);
}
// ====================
// 실제 경로 구하기
// 간선이 없으면 아주 큰 값을 넣는다.
int adj[MAX_V][MAX_V];
// via[u][v] = u에서 v까지 가는
// 최단 경로가 경유하는 점 중 가장 번호가 큰 정점
int via[MAX_V][MAX_V];
void floyd2() {
  for (int i = 0; i < V; i++) adj[i][j] = 0;
  memset(via, -1 ,sizeof(via));
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        if (adj[i][j] > adj[i][k] + adj[k][j]) {
          via[i][j] = k;
          adj[i][j] = adj[i][k] + adj[k][j];
        }
}
// u에서 v로 가는 최단경로를 계산해 path에 저장한다.
void reconstruct(int u, int v, vector<int>& path) {
  // 기저 사례
  if (via[u][v] == -1) {
    path.push_back(u);
    if (u != v) path.push_back(v);
  } else {
    int w = via[u][v];
    reconstruct(u, w, path);
    path.pop_back(); // w가 중복으로 들어가므로 지운다.
    reconstruct(w, v, path);
  }
}
```

## 1.7 Ford Fulkerson

```
const int INF = 987654321;
int V;
// capacity[u][v] = u에서 v로 보낼 수 있는 용량
// flow[u][v] = u에서 v로 흘러가는 유량 (반대 방향인 경우 음수)
int capacity[MAX_V][MAX_V], flow[MAX_V][MAX_V];
// flow[][]를 계산하고 총 유량을 반환한다.
int networkFlow(int source, int sink) {
```

```
  //flow를 0으로 초기화 한다.
  memset(flow, 0, sizeof(flow));
  int totalFlow = 0;
  while(1) {
    // 너비 우선 탐색으로 증가 경로를 찾는다.
    vector<int> parent(MAX_V, -1);
    queue<int> q;
    parent[source] = source;
    q.push(source);
    while(!q.empty() && parent[sink] == -1) {
      int here = q.front(); q.pop();
      for (int there = 0; there < V; there++) {
        // 잔여 용량이 남아 있는 간선을 따라 탐색한다.
        if (capacity[here][there] - flow[here][there] > 0 &&
        parent[there] == -1) {
          q.push(there);
          parent[there] = here;
        }
      }
      // 증가 경로가 없으면 종료한다.
      if (parent[sink] == -1) break;
      // 증가 경로를 통해 유량을 얼마나 보낼 지 결정한다.
      int amount = INF;
      for (int p = sink; p != source; p = parent[p]) {
        amount = min(capacity[parent[p]][p] - flow[parent[p]][p]);
      }
      // 증가 경로를 통해 유량을 보낸다.
      for (int p = sink; p != source; p = parent[p]) {
        flow[parent[p]][p] += amount;
        flow[p][parent[p]] -= amount;
      }
      totalFlow += amount;
    }
    return totalFlow;
  }
}
```

## 1.8 KMP

```
// 총 문자열에서 중복되는 주기문을 찾는 문제에서 발췌했습니다.
int answer = 0;
int m = str.length(); // m은 총 문자열의 길이
int j = 0;
vector<int> pi(m);
for (int i = 1; i < m; i++)
{
  if (j == 0 && str[i] == str[j])
  { // 같은게 생겼다? -> 합리적의심 시작
    answer = i;
  }
  // 합리적 의심중에 의심이 실패한다면
  while (j != 0 && str[i] != str[j])
  { // str[i] == str[j]가 될 때 까지 pi를 이용해 되돌아가기
    j = pi[j - 1];
    answer = i - j;
  }
  if (str[i] == str[j])
  { // 합리적 의심이 지속된다면
    pi[i] = ++j;
  }
  if (j == 0)
  { // 같은게 없어졌다? -> 반복 안하는중
    answer = 0;
  }

  if (answer != 0)
  { // answer가 0이 아니라는 건 합리적 의심중이라는 뜻
    if (pi[i] % answer == 0)
    { // answer 크기만큼 pi가 증가중이다 -> 반복중
      I.push_back(i + 1);
      N.push_back(pi[i] / answer + 1);
    }
  }
}
/*
예시입니다
str
```

```
pi

abababababab
0012345678910

aaaaaa
012345

ababcababc
0012012345

aab aab aab aab
010 123 456 789

aaba aaba
0101 2234
*/

// ========================================================
// pi가 이미 나와있다면 pi배열 설정만 해주면 됩니다.
// N에서 자기 자신을 찾으면서 나타나는 부분 일치를 이용해 pi[] 계산
// pi[i] = N[..i]의 접미사도 되고 접두사도 되는 문자열의 최대 길이
vector<int> getPartialMatch(const string &N)
{
  int M = N.size();
  vector<int> pi(M, 0);
  //KMP로 자기 자신을 찾는다
  //N을 N에서 찾는다.
  //begin = 0이면 자기 자신을 찾아버리니까 안됨!
  int begin = 1, matched = 0;
  //비교할 문자가 N의 끝에 도달할 때까지 찾으면서 부분 일치를 모두
  기록한다
  while (begin + matched < M) {
    if (N[begin + matched] == N[matched])     {
      matched++;
      pi[begin + matched - 1] = matched;
    }
    else     {
      if (matched == 0)
        begin++;
      else {
        begin += matched - pi[matched - 1];
        matched = pi[matched - 1];
      }
    }
  }
  return pi;
}

vector<int> kmpSearch2(const string &H, const string &N) {
  int n = H.size(), m = N.size();
  vector<int> result;
  vector<int> pi = getPartialMatch(N);
  //현재 대응된 글자의 수
  int matched = 0;
  //짚더미의 각 글자를 순회
  for (int i = 0; i < n; i++) {
    //matched번 글자와 짚더미의 해당 글자가 불일치할 경우,
    //현재 대응된 글자의 수를 pi[matched-1]로 줄인다
    while (matched > 0 && H[i] != N[matched])
      matched = pi[matched - 1];
    //글자가 대응될 경우
    if (H[i] == N[matched]) {
      matched++;
      if (matched == m) {
        //문제에서 인덱스는 0이 아닌 1부터 시작
        result.push_back(i - m + 2);
        matched = pi[matched - 1];
      }
    }
  }
  return result;
}
```

## 1.9 Bipartite Matching

```
#include <iostream>
#include <vector>
#include <cstring>
#define MAX 1001
```

```cpp
using namespace std;

int A_size, B_size; //A그룹의 크기, B그룹의 크기
vector<int> A_adj[MAX]; //A그룹 => B그룹 연결선
int A[MAX];  //A->B, a에 매칭되는 B의 element 번호
int B[MAX]; //B->A , b에 매칭되는 A의 element 번호

bool goBipartiteMatching(int a, bool* callstack){
    callstack[a] = true; //무한루프 방지를 위해서, 해당 매칭에서 이미 방문한 것은 체크한다.
    for(int idx = 0; idx < A_adj[a].size(); idx++){ //A와 연결된 B의 모든 원소 방문
        int b = A_adj[a][idx];
        //b에 짝이 없다면 바로 매칭
        if(B[b] == -1){
            A[a] = b;
            B[b] = a;
            return true;
        }
        //이미 b에 짝이 있다. 하지만 B[b]가 이번 스텝에서 방문되지 않았고, B[b]에 새로운 원소를 찾을 수 있다면 a-b매칭 가능
        if(callstack[B[b]] == false && goBipartiteMatching(B[b], callstack)){
            A[a] = b;
            B[b] = a;
            return true;
        }
    }
    return false;
}

int main(){
    cin >> A_size;
    cin >> B_size;
    A_size++; //A의 원소가 1번부터 n번이라서, index도 1~n을 사용할 수 있게 size하나 증가시킴
    B_size++; //B의 원소가 1번부터 n번이라서 index도 1~n을 사용할 수 있기 size하나 증가시킴
    int adj_count, tmp;  //adj입력받기
    for(int i = 1; i < A_size; i++){
        cin >> adj_count;
        for(int j = 0; j < adj_count; j++){
            cin >> tmp;
            A_adj[i].push_back(tmp);
        }
    }
    memset(A, -1, sizeof(int)*(A_size));
    memset(B, -1, sizeof(int)*(B_size));

    //A의 i번이 매칭되지 않았다면 매칭시도
    bool callStack[A_size];
    for(int i = 1; i < A_size; i++){
        if(A[i] == -1){
            memset(callStack, false, sizeof(bool)*A_size);
            goBipartiteMatching(i, callStack);
        }
    }
    return 0;
}
```

# 2 Data Structure

## 2.1 Disjoint Set

```cpp
#include <iostream>
#include <map>
#include <utility>
using namespace std;

//map<int, int> disjoint_set; // map<self value, root value>
int *disjoint_set;

void make_set(int set_size){
    disjoint_set = new int[set_size+1];
    for(int s=0; s<set_size+1; s++){
        disjoint_set[s] = -1; // init: no root
    }
}

int find_root(int a){
```

```
    int root = a;
    while(disjoint_set[root] != -1){
        root = disjoint_set[root];
    }
    if(disjoint_set[a] != -1) disjoint_set[a] = root; //optimizing
    return root;
}

void union_set(int a, int b){
    int a_root = find_root(a);
    int b_root = find_root(b);
    if(a_root == b_root) return;    // root가 같으면 합칠 필요 없다.
    disjoint_set[b_root] = a_root;  // root가 다르면 b set을 a set에
    붙여준다.
}

bool find_set(int a, int b){
    int a_root = find_root(a);
    int b_root = find_root(b);
    if(a_root == b_root) return true;   // a와 b는 같은 set이다.
    return false;                       // a와 b는 disjoint set이다.
}

int main(){
    ios::sync_with_stdio(false);
    cin.tie(0); // 입력이 100,000*3 정도 되는 큰 입력이면,
    sync_with_stdio 보다 cin.tie()가 더 중요하다고 함!
    cout.tie(0);

    // 두 node a,b의 set을 합치거나 같은 set에 있는지 확인하는 문제.

    int N, M; // 초기 집합의 개수, 연산(Question)의 개수
    cin >> N >> M;

    make_set(N); // disjoint set을 초기화한다.

    for(int m=0; m<M; m++){
        int question, a, b;
        cin >> question >> a >> b;
        switch(question){
        case 0: // union
            union_set(a, b);
            break;
        case 1: //find
            cout << (find_set(a, b) ? "YES" : "NO") << '\n';
            break;
        }
    }
    return 0;
}
```

## 2.2 Segment Tree

```
#include <vector>
#include <iostream>
using namespace std;

class SegTree{
private:
    //완전이진트리 뒷쪽에 빈 값을 채우기 위함, invalid한 값을 넣어야한다.

    //현재는 구간곱을 구하는 예제라서 곱셈의 항등원인 1을 넣은 것
    const int inValid = 1;
    int size; //tree 배열을 저장하는 전체 사이즈
    int leaf; //leaf node가 시작되는 index
    int* data;

    //n보다 크면서 가장 가까운 2의 거듭제곱 * 2한 값 계산
    int calSize(int n){
        int m = 1;
        n = n - 1;
        while(n > 0){
            m = m << 1;
            n = n >> 1;
        }
        return m*2;
    }

public:
```

```
    //생성자: values를 leaf노드로 옮기고 위쪽 트리를 계산해서 채움
    SegTree(vector<int>& values){
        size = calSize(values.size());
        data = new int[size];
        leaf = size/2;
        //leaf노드에 값 옮기기
        for(int i = 0; i < values.size(); i++){
            data[leaf + i] = values[i];
        }
        //완전 이진트리 남는 자리에 invalid한 값 넣기
        for(int i = leaf + values.size(); i < size; i++){
            data[i] = inValid;
        }
        //root노드(index==1)까지 부분 결과값 채우기
        for(int i = leaf - 1; i > 0; i--){
            //현재는 구간곱을 구하는 예제라서 곱셈을 하지만,
            //찾는 값에 따라서 다른 연산으로 대체해야함.
            data[i] = data[i*2] * data[i+2 + 1];
        }
    }

    //index에 newValue라는 새로운 값이 들어오는 경우
    void update(int index, int newValue){
        int indexOfTree = leaf + index;
        data[indexOfTree] = newValue;
        //root노드로 올라가면서 관련 값들 모두 갱신하기
        indexOfTree = indexOfTree / 2;
        while(indexOfTree > 0){
            //갱신되는 값 계산도 경우에 따라서 다른 연산으로 구해야함.
            //현재는 구간곱을 구하는 예제라서 곱셈을 했음.
            data[indexOfTree]  = data[indexOfTree*2] *
            data[indexOfTree*2 + 1];
            indexOfTree = indexOfTree / 2;
        }
    }

    //[start, end] 까지의 구간값을 rLeft(range_Left),
    rRight(range_Right) 범위내에서 구함.
    int select(int start, int end, int rLeft=-1, int rRight=-1){
        //rLeft범위가 안 주어지면 전체 범위에서 진행
        //rLeft = Leaf노드의 시작, start = tree에서의 start index로
        변환
        if(rLeft == -1){
            rLeft = size / 2;
            start = rLeft + start;
        }
        //rRight의 범위가 안 주어지면 전체 범위에서 진행
        //rRight = Leaf노드의 끝, end = tree에서의 end index로 변환
        if(rRight == -1){
            rRight = size - 1;
            end = rLeft + end;
        }
        //[start, end], [rLeft, rRight]가 겹치는 구간이 없는 경우
        if(end < rLeft || rRight < start){
            return inValid;
        }
        //[rLeft, rRight]가 [start, end]에 포함되는 경우
        if(start <= rLeft && rRight <= end){
            int gap = rRight - rLeft; //현재 구간 크기
            int index = rLeft; //현재 구간 index
            while(gap > 0){ //gap이 0이 될때까지 계속 트리를 타고
            올라간다.
                gap = gap >> 1;
                index = index >> 1;
            }
            return data[index]; //구간에 해당되는 값반환
        }
        //그 외의 모든 경우
        int mid = (rLeft + rRight) / 2;
        //range를 다시 2개로 나누어서 [start, end]에 포함되는 range
        찾을때까지 진행
        //적절한 연산으로 찾은 2개의 값을 비교해야한다. 여기서는 구간곱
        예제라서 곱셈을 쓴 것
        return select(start, end , rLeft, mid) * select(start, end,
        mid+1, rRight);
    }
};

int main(){
```

```
    vector<int> arr = {1, 2, 3, 4, 5};
    SegTree tree(arr);
    tree.update(2, 6); //arr[2] = 6 update
    cout << tree.select(1, 4) << endl; //구간곱(arr[1] ~ arr[4])
    tree.update(4, 2);  //arr[4] = 2 update
    cout << tree.select(2, 4) << endl;  //구간곱(arr[2] ~ arr[4])
    return 0;
}
```

# 3 NumberTheory

## 3.1 Greatest Common Divider

```
// a > b여야한다.
int gcd (int a, int b){
  int c;
  //b==0이라면 a를 출력하고 종료한다.
  while(b){
    c = a % b;
    a = b;
    b = c;
  }
  return a;
}
```

## 3.2 Extended Eculid

```
#include <bits/stdc++.h>
using namespace std;

long long gcd(long long a, long long b){
    if(a%b==0)
        return b;
    return gcd(b, a%b);
}

//확장유클리드 알고리즘 ax + by = 1인 방정식의 해(x, y)를 반환한다.
//단, gcd(a, b) == 1 이어야만 한다, 그렇지 않다면 해가 존재하지 않는 것
//또한 반드시 a > b여야한다.
pair<long long, long long> extend_euclid(long long a, long long b){
    if(b == 1){
        return make_pair(0ll, 1ll);
    }
    long long q = -1*a/b;
    long long r = a%b;
    if(r == 1){
        return make_pair(1ll, q);
    }
    pair<long long, long long> res = extend_euclid(b, r);
    return make_pair(res.second, res.second*q + res.first);
}

int main(){
    //a^(-1) mod n , a의 법n에서의 역원 구하기
    //nx + ay = 1 에서 y가 a의 법n의 역원이다.
    long long n, a;
    cin >> n >> a;
    long long mult_inverse;

    //gcd(n, a) != 1 이라면 ax + ny = 1의 해가 없음으로 역원이 존재하지
    않는다//베주항등식 참고
    if(gcd(n, a) != 1)
        mult_inverse = -1;
    else{
        //nx + ay = 1에서 해인 (x, y)반환
        pair<long long, long long> res = extend_euclid(n, a);
        //pair의 2번째인자인 second, 즉 y가 역원에 해당한다.
        long long tmp = res.second;
        //법 n에 맞춰서 만약 음수값이거나 n보다 큰 값이면, 0 <= y < n으로
        값을 조정한다.
        while(tmp < 0)
            tmp += n;
        while(tmp > n)
            tmp -= n;
        mult_inverse = tmp;
    }
    cout << mult_inverse << endl;
    return 0;
}
```

### 3.3 Square and Multiply method

```
#include <iostream>
typedef long long ll;
using namespace std;

/*보다 빠른 n^k를 위하여
    5^12 = (5^6)^2 = ((5^3)^2)^2  = ((5^2 * 5)^2)^2
    12 / 2 = 6 + 0  square
    6 / 2 = 3 + 0   square
    3 / 2 = 1 + 1   multiply
*/
ll get_power(int n, int k){
    bool bin[64];
    int i = 0;
    int power = k;
    while( power > 0 ){
        if(power % 2)
            bin[i] = 1;
        else
            bin[i] = 0;
        i++;
        power = power/2;
    }

    i = i - 2;
    ll res = n;
    while(i >= 0){
        res = res * res;
        cout << "[" << i << "], res = " << res << "(sqaure)" <<
        endl;
        if(bin[i]){
            res = res * n;
            cout << "[" << i << "], res = " << res << "(multiply)"
            << endl;
        }
        i--;
    }
    return res;
}

int main(){
    int a, b;
    cin >> a >> b;
    ll num = get_power(a, b);
    cout << a << "^(" << b << ") = " << num << endl;
    return 0;
}
```

# 4 Geometry

## 4.1 ConvexHull

```
#include <iostream>
#include <utility>
#include <algorithm>

using namespace std;

#define NMAX 100000
typedef long long LL;
typedef pair<LL, LL> Point;

int N;  // num of point
Point point[NMAX+1]; //pair<x, y>
Point convex_hull[NMAX]; //save points in convex hull
int ans = 0; // convex hull count

int ccw(Point &a, Point &b, Point &c){ // 반시계 방향(ccw)를 찾는
알고리즘
    LL cross = a.first*b.second + b.first*c.second +
    c.first*a.second
                - b.first*a.second - c.first*b.second -
                a.first*c.second;
    if(cross > 0) return 2;          // ccw
    else if(cross == 0){             // parallel
        return 1;
    }
    else return -1;                  // cw
}
```

```
bool cmp(Point& a, Point& b){ // 탐색 시작점을 기준으로, 기울기가 작은
순서대로 정렬하기 위한 비교 함수
    LL agrad = (a.second-point[0].second)*(b.first-point[0].first);
    LL bgrad = (b.second-point[0].second)*(a.first-point[0].first);
    if(agrad > bgrad) return 0;
    else if(agrad == bgrad && a.first > b.first) return 0; //
    기울기가 같으면 x값이 작은 순서대로 정렬
    return 1;
}
void convexHull(){
    // 첫 두 점을 convexhull에 넣는다.
    convex_hull[0] = point[0];
    convex_hull[1] = point[1];
    ans = 2;
    // State Space에 있는 point들을 탐색하며 convex hull을 찾는다.
    for(int p=2; p<N+1; p++){
        LL cross = ccw(convex_hull[ans-2], convex_hull[ans-1],
        point[p]); // cross product
        // ccw인 경우, point를 convex hull에 넣는다.
        if( cross == 2){
            convex_hull[ans] = point[p];
            ans++;
        }
        // parallel인 경우, 이전 point를 삭제, 현재 point를 추가.
        // 문제 조건에서 변에 여러 점이 있는 경우, 양 끝점만 개수에
        포함되기 때문.
        else if(cross == 1){
            convex_hull[ans-1] = point[p];
        }
        // cw인 경우, ccw가 될때까지 이전 point들을 convex_hull에서
        제외한다.
        else{
            bool add_point_flag = false;
            while(ans > 2){
                ans--;
                cross = ccw(convex_hull[ans-2], convex_hull[ans-1],
                point[p]);
                if(cross == 2){ // ccw인 경우, 현재 point 추가.
                    convex_hull[ans] = point[p];
                    ans++;
                    add_point_flag = true;
                    break;
                }
                else if(cross == 1){ // parallel인 경우, 이전 point
                삭제, 현재 point 추가.
                    convex_hull[ans-1] = point[p];
                    add_point_flag = true;
                    break;
                }
            }
            // 만약 시작점과 convex hull의 두번째 point만 남았는데
            여전히 cw인 경우,
            // 두번째 point를 지워주고 현재 point를 넣는다.
            if(!add_point_flag){
                convex_hull[ans-1] = point[p];
            }
        }
    }
}

int main(){
    ios::sync_with_stdio(false);

    // Example
    cin >> N; // point의 개수
    for(int i=0; i<N; i++){
        cin >> point[i].first >> point[i].second;
        // 편의를 위해 point의 범위 >0 으로 맞춰준다.
        point[i].first += 40001;
        point[i].second += 40001;
    }
    sort(point, point+N);       // x, y 최소값 찾기 --> 탐색 시작점을
    찾음.
    sort(point+1, point+N, cmp);// point[0]에 대한 기울기 오름차순
    point[N] = point[0];        // convex hull 계산의 편의를 위해,
    시작점을 마지막 원소에 넣어줌.

    convexHull();               // convex hull과 convex hull의 point
    개수를 구한다.
```

```
    cout << ans-1; // 마지막 원소 == 시작점이었음으로, 중복되는 시작점을
    빼줌.

    return 0;
}
```

## 4.2 CrossLine Detection

```
#include <iostream>

using namespace std;

typedef long long LL;
typedef struct{
    LL x, y;
}Point;

class Line{
private:
    Point s, e;
public:
    Line(int x1, int y1, int x2, int y2){
      s = {x1, y1};
      e = {x2, y2};
    }
    LL getAreaOfTwoTriangles(Line &ano){
        // self 선분의 각각의 끝점(s, e)에서
        // ano 선분의 양 끝점(s, e)으로 선분을 연결해서 나오는 삼각형 2
        개의 넓이 합을 구함
        LL x = this->s.x;
        LL y = this->s.y;
        LL cross_s = ano.s.x*y + x*ano.e.y + ano.e.x*ano.s.y
                   - x*ano.s.y - ano.e.x*y - ano.s.x*ano.e.y;
        x = this->e.x;
        y = this->e.y;
        LL cross_e = ano.s.x*y + x*ano.e.y + ano.e.x*ano.s.y
                   - x*ano.s.y - ano.e.x*y - ano.s.x*ano.e.y;
        cross_s = cross_s < 0 ? -cross_s : cross_s;
        cross_e = cross_e < 0 ? -cross_e : cross_e;
        return cross_s + cross_e; // 실수형을 피하기위해 /2를 안해서
        넓이의 2배인 것에 주의
    }
};

bool checkLineCross(Line &a, Line &b){
    // 1. ano 선분의 각각의 끝점에서 self 선분의 양 끝점으로 선분을
    연결했을 때 나오는 삼각형 2개의 넓이 합을 구함
    // 2. self 선분의 각각의 끝점에서 ano의 선분의 양 끝점으로 선분을
    연결해서 나오는 삼각형 2개의 넓이 합을 구함
    // 3. 두 합이 같으면 cross
    // 4. 아니면 not cross
    LL a_area = a.getAreaOfTwoTriangles(b);
    LL b_area = b.getAreaOfTwoTriangles(a);
    return a_area == b_area;
}

int main(){
    // Example
    Line l1(1,1,5,5);
    Line l2(1,5,5,1);
    cout << (checkLineCross(l1, l2) ? "cross" : "not cross") <<
    '\n';

    Line l3(1,1,5,5);
    Line l4(6,10,10,6);
    cout << (checkLineCross(l3, l4) ? "cross" : "not cross") <<
    '\n';

    return 0;
}
```