SCPC 3회 1차예선 풀이

서울대학교 컴퓨터공학부 18학번 김동현

1번 - 괄호

- 세 종류의 괄호 문자들로 이루어진 문자열이 있습니다.
- 올바른 괄호 문자열을 다음과 같이 정의합니다:
  - 빈 문자열은 올바른 괄호 문자열입니다.
  - A가 올바른 괄호 문자열일 때, (A), {A}, [A] 역시 올바른 괄호 문자열입니다.
  - A, B가 올바른 괄호 문자열일 때, AB 역시 올바른 괄호 문자열입니다.
- 주어진 문자열의 연속한 부분 문자열 중 가장 긴 올바른 괄호 문자열의 길이를 구하세요.
- 1 ≤ (문자열 길이) ≤ 10^6

1번 - 괄호

- 전체 문자열이 올바른 괄호 문자열인지 아닌지 판단하는 것은 스택으로 간단하게 O(N)에 할 수 있습니다.
- 여는 괄호가 나오면 스택에 넣고, 닫는 괄호가 나오면 현재 스택의 맨 위에 자신과 짝이 맞는 모양의 여는 괄호가 있는지 검사하면 됩니다.
- 이 방법을 약간 응용해서 이 문제에도 써먹을 수 있습니다.

1번 - 괄호

- 문자열의 각 문자가 올바른 괄호 문자열의 일부인지 아닌지를 체크한다고 합시다.
- 앞에서부터 순서대로 보면서, 여는 괄호가 나오면 일단 스택에 넣습니다.
- 닫는 괄호가 나오면, 스택의 맨 위에 짝이 맞는 여는 괄호가 있는지 먼저 봅니다.
  - 짝이 맞는 괄호가 있다면, 스택에서 빼면서 이 두 문자를 올바른 괄호 문자열의 일부로 체크합니다.
  - 짝이 안 맞는다면, 스택에 있는 문자들을 (체크하지 않고) 다 빼버립니다. 지금 본 닫는 괄호 역시 체크하지 않습니다.

1번 - 괄호

- 올바른 괄호 문자열을 지나는 동안에는 알고리즘에서 “짝이 안 맞는” 경우에 절대 들어가지 않으므로 해당하는 문자들이 모두 체크됩니다.
- 중간에 이상한 경우를 만나면 바로 스택을 비워 버리므로, 올바르지 않은 괄호가 체크될 일은 없습니다.
- 문자열을 한 번 쭉 훑은 뒤, 가장 길게 연속해서 체크되어 있는 문자들의 총 길이를 출력하면 됩니다.

1번 - 괄호

- 스택에 문자 자체를 넣는 것이 아니라 인덱스를 넣으면 구현이 편합니다.

```
void solve() {
  string s;
  cin >> s;

  int n = s.length();
  vector<int> chk(n);
  stack<int> st;

  static char mat[1 << 8];
  mat[')'] = '(';
  mat['}'] = '{';
  mat[']'] = '[';

  for(int i = 0; i < n; i++) {
    char c = s[i];
    if(c == '(' || c == '{' || c == '[') st.push(i);
    else {
      if(!st.empty() && mat[c] == s[st.top()]) {
        chk[st.top()] = chk[i] = 1;
        st.pop();
      } else {
        while(!st.empty()) st.pop();
      }
    }
  }

  for(int i = 1; i < n; i++) chk[i] *= (chk[i - 1] + 1);
  cout << *max_element(all(chk)) << '\n';
}
```

2번 - 주식거래

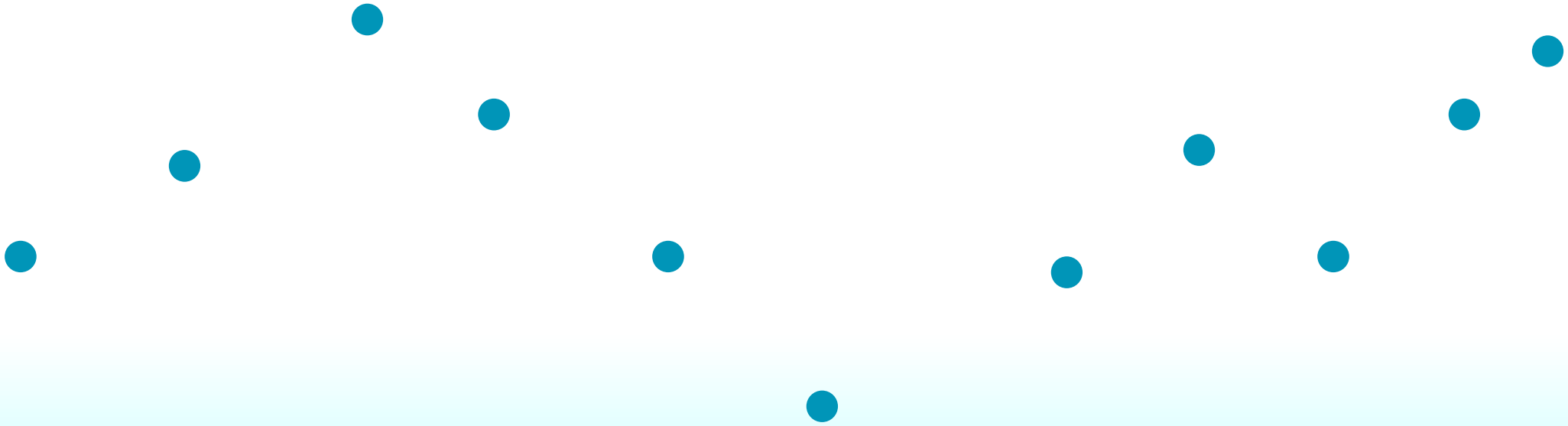
- N일 간의 주식 가격이 주어집니다.
  - $1 \leq N \leq 200,000$
- 다음과 같은 방식으로 주식 거래를 하려고 합니다.
  - 하루에는 주식을 사거나, 가만히 있거나, 주식을 팔 수 있습니다.
  - 주식은 최대 1주만 소유 가능합니다.
  - 주식은 산 가격보다 더 비싼 값에 팔아야 합니다.
  - 주식을 살 때는 바로 직전에 판 값보다는 싸게 사야 합니다. (첫 번째로 사는 주식은 제외)
- N일이 끝난 시점에는 주식을 가지고 있지 않아야 한다고 할 때, 가능한 최대 거래 횟수를 구하세요.

2번 - 주식거래

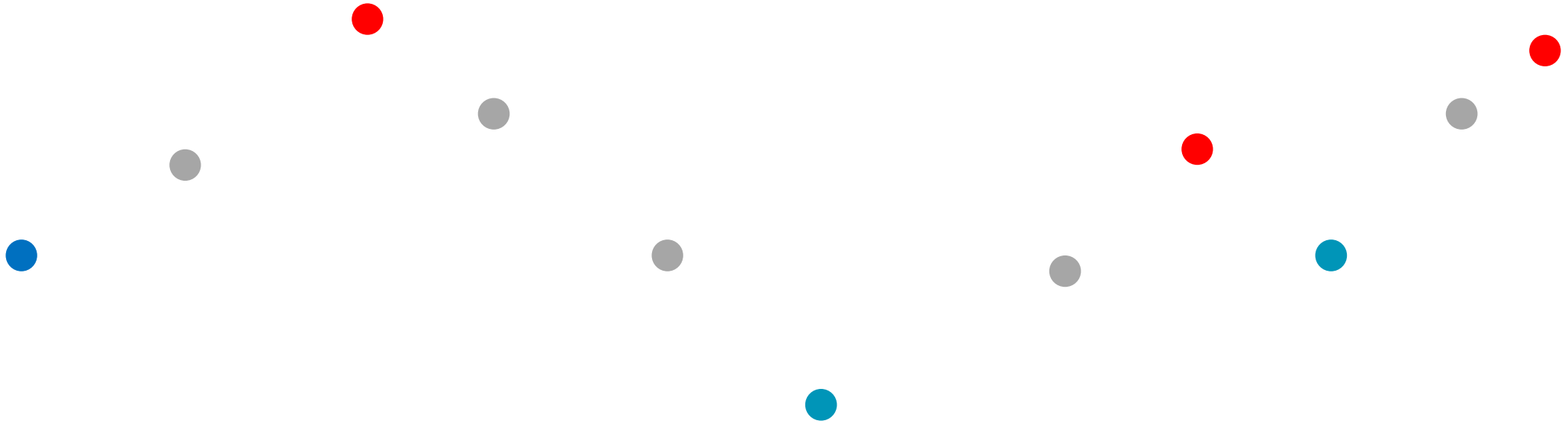
- 증가 → 감소 → 증가 → 감소 → … 를 반복하는 최대 길이의 부분 수열을 구하는 문제입니다.
- DP를 해 볼까요?
  - O(N^2)은 크게 어렵지 않습니다.
  - 세그먼트 트리 등의 자료구조를 이용해 최적화를 하면 O(NlogN)이 됩니다.
- 1차예선 2번인데 이게 의도된 풀이일 거 같지는 않습니다….
- 매우 쉽고 간단한 O(N) 풀이가 있습니다.

2번 - 주식거래

- 인접한 두 날짜의 가격 변동은 증가하거나, 감소하거나, 똑같거나 셋 중 하나입니다. 이 중 똑같은 경우는 그냥 값 하나를 지워버린다고 생각하면 무시할 수 있습니다.
- 결국 주어진 수열은 증가와 감소를 반복하는 형태가 됩니다.



2번 - 주식거래



- 파란 점 (감소 → 증가) 마다 사고, 빨간 점 (증가 → 감소) 마다 팔면 거래를 최대한 많이 할 수 있습니다.
- 주식을 못 판 채로 끝내면 안 됨에 유의해주세요.

2번 - 주식거래

- 코드는 매우 간단합니다.
- 입력을 배열에 저장할 필요도 없습니다!

```
void solve() {
  int n;
  cin >> n;

  int pv, sgn = -1, ans = 1;
  for(int i = 0; i < n; i++) {
    int x;
    cin >> x;
    if(i) {
      if(pv == x) continue;
      int nsgn = (x > pv) - (x < pv);
      if(sgn != nsgn) {
        ans++;
        sgn = nsgn;
      }
    }
    pv = x;
  }

  cout << (ans / 2 * 2) << '\n';
}
```

3번 – 전광판

- N × M 격자의 각 칸에 전구가 놓여 있습니다.
  - 1 ≤ N, M ≤ 100
- 각 전구는 스위치 2개에 연결되어 있습니다.
- 각 전구는 초기에 꺼져 있거나 켜져 있으며, 연결된 스위치를 누를 때마다 그 상태가 toggle됩니다.
- 어떤 스위치들의 집합을 골라서 그것들을 한 번씩 눌렀을 때 결과적으로 모든 전구가 켜지도록 만들고 싶습니다.
- 그럴 수 있다면 어떤 스위치들을 눌러야 하는지 출력하고, 그럴 수 없다면 “Impossible”을 출력하세요.

3번 - 전광판

- 입력 형식이 좀 귀찮은데, 우선 정리를 좀 해 봅시다.
- R 첨자가 붙은 스위치와 C 첨자가 붙은 스위치는 각각 총 NM개 존재합니다. 각 스위치에 번호를 잘 매겨 봅시다.
- 저는 R_{ij} → $iM + j$, C_{ij} → $NM + iM + j$ 와 같이 번호를 매겼습니다.
  - 입력에 나타나지 않는 스위치가 중간에 껴 있음을 유의해야 합니다.
- 나타나는 스위치만 골라서 따로 번호를 매길 수도 있는데, 이러면 전처리가 조금 더 귀찮아집니다.

3번 - 전광판

- 이제 각 전구가 2개의 스위치와 연결되어 있다는 조건에 대해 생각해봅시다.
- 각 스위치는 누를지 말지에 해당하는 논리 변수로 생각할 수 있습니다.
- 각 전구는 스위치 2개에 해당하는 값이 같은지 또는 다른지에 해당하는 식입니다.
  - 전구가 초기에 켜져 있었다면, 스위치 2개는 같은 값을 가져야 합니다.
  - 전구가 초기에 꺼져 있었다면, 스위치 2개는 다른 값을 가져야 합니다.
- 이 문제는 2-SAT으로 환원할 수 있고, O(스위치 개수)에 풀립니다 ^^..
- 다행히도 이 문제 같은 경우는 더 쉬운 풀이가 있습니다.

3번 - 전광판

- 각 스위치에 해당하는 논리 변수를 p_i라고 합시다.
- p_i, $\sim p_i$에 해당하는 정점을 스위치마다 2개씩 만듭시다.
- 각 전구에 해당하는 식에 따라서 다음과 같이 간선을 만듭시다.
  - 전구가 켜져 있다면, $p_u - p_v$와 $\sim p_u - \sim p_v$를 각각 잇습니다.
  - 전구가 꺼져 있다면, $p_u - \sim p_v$와 $\sim p_u - p_v$를 각각 잇습니다.
- 이렇게 만든 그래프에서 아래 조건을 만족하도록 각 논리 변수에 참/거짓을 배당하면 됩니다.
  - 같은 컴포넌트에 속한 정점들은 모두 같은 값을 가집니다.
  - p_i와 $\sim p_i$에 해당하는 정점은 서로 반대 값을 가집니다.

3번 - 전광판

- 어떤 i에 대해 p_i와 $\sim p_i$가 같은 컴포넌트에 속해 있다면 조건을 절대 만족시킬 수 없으니 Impossible입니다.
- 그런 i가 없다면, 각 컴포넌트에 대해 서로 마주보는 컴포넌트 (식이 대칭적이므로 컴포넌트들은 항상 둘씩 짝지어집니다) 각각에 참, 거짓을 채우면 됩니다.
- 마지막에 참으로 표시된 스위치들을 모두 출력해주면 됩니다.
- 입력에 안 들어온 스위치에 참을 채우지 않도록 주의해주세요.

3번 - 전광판

- 입력을 받고 그래프를 만드는 부분입니다.
- Union-Find (분리 집합) 구조를 사용했습니다.
- 각 스위치 i에 대해 i번 정점은 p_i, (i+2nm)번 정점은 ~p_i를 나타냅니다.

```
int n, m;
cin >> n >> m;

int k = n * m;
vint p(4 * k);
iota(all(p), 0);
auto idx = [&](int x, int y){ return x * m + y; };
function<int(int)> f = [&](int x){ return p[x] = (x == p[x] ? x : f(p[x])); };
auto u = [&](int x, int y){ p[f(y)] = f(x); };

for(int i = 0; i < n; i++) {
  for(int j = 0; j < m; j++) {
    int x, y, z;
    cin >> x >> y >> z;
    y = idx(i, y);
    z = idx(z, j) + k;
    if(x) {
      u(y, z);
      u(y + 2 * k, z + 2 * k);
    } else {
      u(y, z + 2 * k);
      u(y + 2 * k, z);
    }
  }
}
```

3번 - 전광판

- 논리 값을 채우고 답을 출력하는 부분입니다.
- ostringstream을 사용해서 답을 그때그때 만들면서도 Impossible 판단을 끝내기 전에 미리 출력하지 않도록 합니다.
- 출력 형식에 유의하세요.

```
ostringstream sout;
vint vis(4 * k, 0);
for(int t = 0; t < 2; t++) {
  for(int i = 0; i < n; i++) {
    for(int j = 0; j < m; j++) {
      int x = idx(i, j) + t * k;
      int nx = x + 2 * k;

      if(f(x) == f(nx)) {
        cout << "Impossible\n";
        return;
      }

      if(!vis[f(x)] && !vis[f(nx)]) vis[f(x)] = -1;
      if(!vis[f(x)]) vis[f(x)] = -vis[f(nx)];
      if(!vis[f(nx)]) vis[f(nx)] = -vis[f(x)];

      if(vis[f(x)] > 0) {
        if(t == 0) sout << 'R' << (i / 10) << (i % 10) << (j / 10) << (j % 10) << ' ';
        else sout << 'C' << (j / 10) << (j % 10) << (i / 10) << (i % 10) << ' ';
      }
    }
  }
}
cout << sout.str() << '\n';
```

4번 - Monotone

- N개의 정점으로 이루어진 단순다각형이 주어집니다.
  - $1 \le N \le 50,000$
- 어떤 단순다각형이 단조다각형이라는 것은, 어떤 방향 d가 존재해서 d에 평행한 모든 직선과 단순다각형의 교집합이 없거나, 점 하나거나, 선분 하나인 조건을 만족하는 것을 뜻합니다.
- 다각형이 주어지면 그 다각형이 단조다각형인지 아닌지 판단하세요.
  - 정점은 다각형의 둘레를 따라 반시계 방향으로 주어집니다.
  - 단조다각형이라면, 단조성을 만족시키는 방향이 실수오차를 초과하는 어떤 범위로 존재함이 보장됩니다.

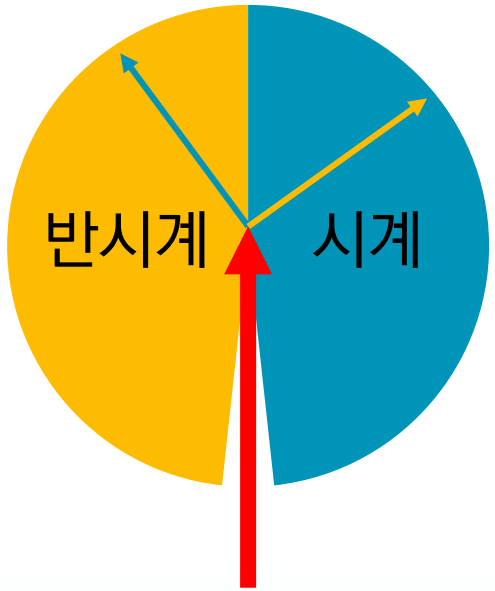
4번 - Monotone

- 드디어(?) 기하 문제가 등장했습니다.
- 문제 풀이에 앞서, 기하 문제에서 밥 먹듯이 쓰이는 함수 하나에 대해 소개하겠습니다.
- 일명 ccw라 불리는 함수입니다.
- 2차원 벡터 a, b가 주어지면 a → b 가 반시계방향이면 1, 시계방향이면 −1, a와 b가 평행하면 0을 반환하는 함수입니다.
- 구현은 이렇게 합니다 :

```
auto ccw = [](pll a, pll b) {
  ll t = a.x * b.y - b.x * a.y;
  return (t > 0) - (t < 0);
};
```

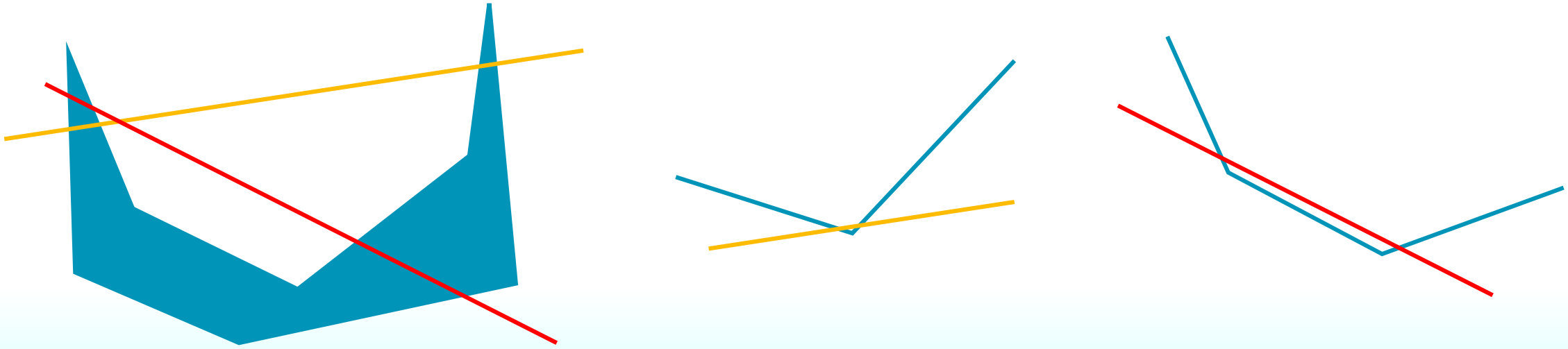
4번 - Monotone

- 시계 / 반시계의 (엄밀하지 않은) 정의는 아래 그림과 같습니다.
  - 왼쪽으로 돌면 반시계
  - 오른쪽으로 돌면 시계



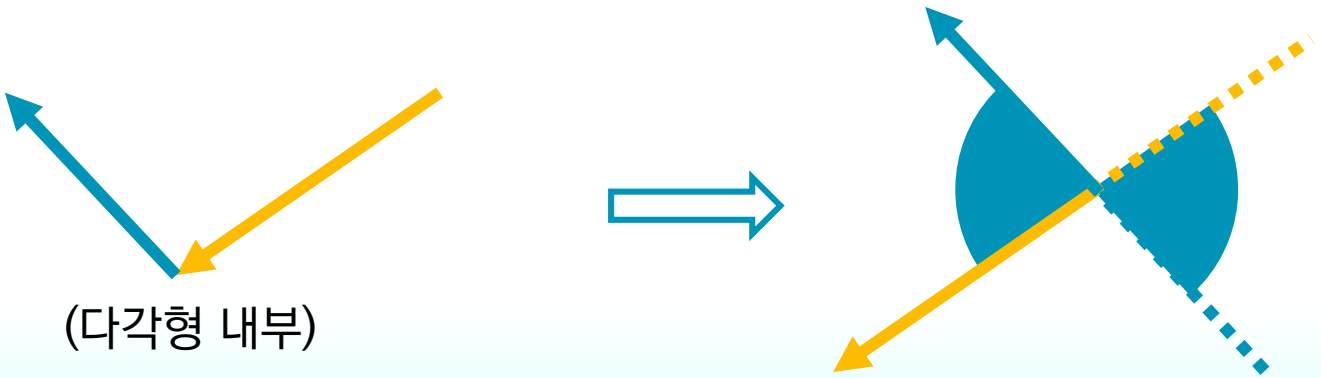
4번 - Monotone

- 단조적이지 않은, 즉 다각형과 두 번 이상에 걸쳐 만나는 직선에 대해 생각해 봅시다.
- 단순 다각형이므로, 어느 한 쪽으로 쭉 내리다 보면 결국 아래 그림과 같이 점 하나 또는 선분 하나에 아주 가까이 갈 수 있습니다.



4번 - Monotone

- 점 하나에 해당하는 경우부터 살펴봅시다.
- 다각형의 각 점에 대해, 그 점에 걸려 있는 내각이 180° 이하일 때는 단조성을 해칠 수 없습니다.
- 내각이 180° 초과일 때는 방향이 아래 그림에 해당하는 각도에 있으면 그 점에 의해 단조성이 깨집니다. (경계는 포함하지 않음)



(다각형 내부)

4번 - Monotone

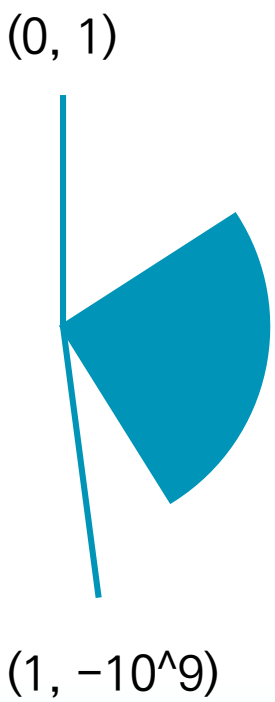
- 선분에 해당하는 경우는 어떻게 될까요?
- 점에 해당하는 경우와 비슷하게 처리할 수 있지만, 좀 귀찮습니다….
  - 다각형의 인접한 두 선분이 평행하면 합쳐 주어야 하는 등의 처리가 있습니다.
- 이쯤에서 문제에 달려있던 이상한 조건을 살펴봅시다.
  - 단조다각형이라면, 단조성을 만족시키는 방향이 실수오차를 초과하는 어떤 범위로 존재함이 보장됩니다.
- 이게 무슨 뜻인가 하면, 아까 다각형의 각 점에서 얻어냈던 불가능한 각도 구간에서 경계를 포함해도 답을 똑같이 구할 수 있다는 것입니다.
- 놀랍게도, 경계를 포함하면 선분 Case가 모두 같이 처리됩니다!

4번 - Monotone

- 이제 우리가 할 일은 여러 개의 각도 구간이 있을 때, 아무런 구간에도 포함되지 않는 각도의 범위가 존재하는지를 판별하는 것입니다.
- 각도가 원 전체에 걸쳐 있으면 처리가 곤란합니다….
- 다행히도, 이 문제의 경우 방향을 180° 돌려도 똑같습니다. 즉, 180° 범위만 택해서 남길 수 있습니다.
- 방향을 (x, y)라고 두었을 때, (x 〉 0) 또는 (x = 0 and y 〉 0)을 만족하는 방향만 고려합시다.
  - STL pair로 나타냈을 때 (0, 0)보다 큰 방향만 남기는 것입니다.

4번 – Monotone

- 각 점에 대한 각도 구간은 아래 두 가지 꼴 중 하나로 나타납니다.



4번 - Monotone

- 방향들은 ccw 순서대로 정렬하면 (a → b가 반시계 방향이면 a가 b보다 앞에 오도록) 수직선 위의 점들과 똑같이 생각할 수 있습니다.
- 각 구간의 시작점에 +1, 끝점에 -1을 놓고 누적합을 한다고 생각하면 됩니다.
- 어느 범위에 누적합이 0인 것을 발견한다면 그 범위에 속하는 모든 방향은 단조성을 만족시킵니다. 즉, 답이 “YES”입니다.
- 그런 범위를 찾을 수 없다면, 즉 모든 각도에 대해 그 각도를 포함하는 구간이 적어도 하나 있다면 답은 “NO” 입니다.
- 구현 디테일에 많은 주의가 필요합니다.

4번 - Monotone

- 정의 및 입력 부분입니다.
- Event 타입은 이후 누적합의 계산을 위해 쓰입니다.
- O, S는 각각 원점, 각도의 시작점을 나타내는 2차원 vector입니다.
- 연산자 오버로딩을 통해 두 vector의 뺄셈을 간단하게 쓸 수 있습니다.

```
using event = pair<pll, int>;
pll operator-(const pll &a, const pll &b){ return pll(a.x - b.x, a.y - b.y); }
const pll O = pll(0, 0);
const pll S = pll(1, -ll(1e9));

void solve() {
  int n;
  cin >> n;

  vpll v(n + 2);
  for(int i = 0; i < n; i++) cin >> v[i].x >> v[i].y;
  v[n] = v[0];
  v[n + 1] = v[1];
```

4번 - Monotone

- 주어진 다각형에서 각도 구간들을 뽑아내는 코드입니다.
- 180° 범위에 속하는 각도만 남깁니다.
- 180° 범위의 맨 끝에 도달할 경우 의도적으로 -1에 해당하는 점을 넣지 않습니다.

```
vector<event> ev;
for(int i = 1; i <= n; i++) {
  pll s = v[i + 1] - v[i];
  pll e = v[i] - v[i - 1];
  if(ccw(s, e) <= 0) continue;

  if(s < 0) s = 0 - s;
  if(e < 0) e = 0 - e;

  if(ccw(s, e) > 0) {
    ev.emplace_back(s, 1);
    if(e.x > 0) ev.emplace_back(e, -1);
  } else {
    ev.emplace_back(S, 1);
    ev.emplace_back(e, -1);
    ev.emplace_back(s, 1);
  }
}
```

4번 - Monotone

- 누적합을 계산하여 단조다각형 여부를 판단하는 코드입니다.
- 시작할 때 또는 중간에 0이 나타나는 범위가 있다면 바로 “YES”를 출력합니다.

```
sort(all(ev), [&](event a, event b){
  int t = ccw(a.x, b.x);
  if(t != 0) return t > 0;
  return a.y > b.y;
});
if(ev.empty() || ev[0].x != S) { cout << "YES\n"; return; }

int c = 0;
for(auto &x : ev) {
  c += x.y;
  if(!c){ cout << "YES\n"; return; }
}
cout << "NO\n";
```

5번 - Covernent

- 정확히 2N개의 리프가 있는 정점 M개짜리 트리가 있습니다. 간선에는 가중치가 존재합니다.
  - $1 \le N \le 500$, $2N \le M \le 2{,}000$
- 리프에는 1~2N까지의 번호가 붙어 있으며, 각 i ($1 \le i \le N$)에 대해 i번 리프와 N+i번 리프 중 하나만 선택할 수 있습니다.
- 리프들을 선택하면, 트리 상에서 선택된 리프들이 모두 하나의 컴포넌트로 이어지도록 최소한의 간선만 남기게 됩니다.
- 리프들을 잘 선택하여 남긴 간선의 가중치 합이 가장 크도록 했을 때, 그 때의 가중치 합을 출력하세요.

5번 - Covernent

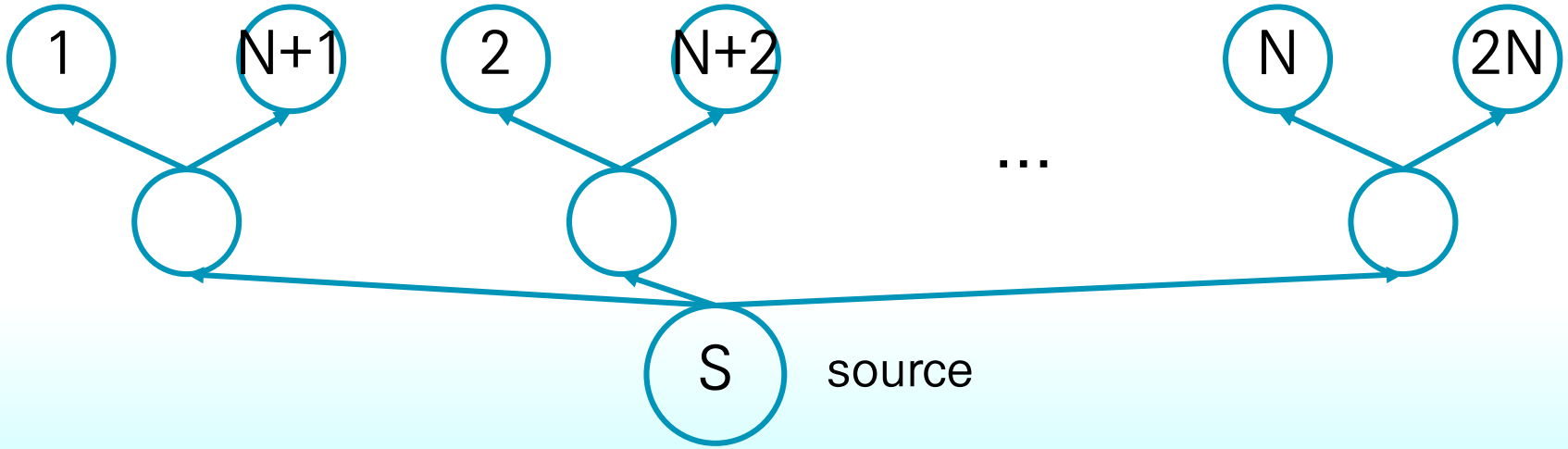
- 조건이 참 묘합니다..
- DP도 그리디도 잘 안 되는 것 같습니다.
- 결론부터 말하면, MCMF (Min Cost Max Flow) 로 풀리는 문제입니다.
- 어떻게 떠올리는지는 저도 잘 모르겠습니다….

5번 - Covernent

- MCMF란 2차 예선 5번 때 나왔던 Max Flow 알고리즘의 확장판입니다.
- 각 간선마다 1의 유량을 흘릴 때 일정한 cost가 소모됩니다. MCMF는 Min Cost Max Flow이므로, Max Flow들 중에 가장 cost가 적게 소모되는 Flow를 구하는 알고리즘입니다.
- 이 문제에서 그래프를 어떻게 설계해야 MCMF로 모델링할 수 있을지 생각을 해 봅시다.

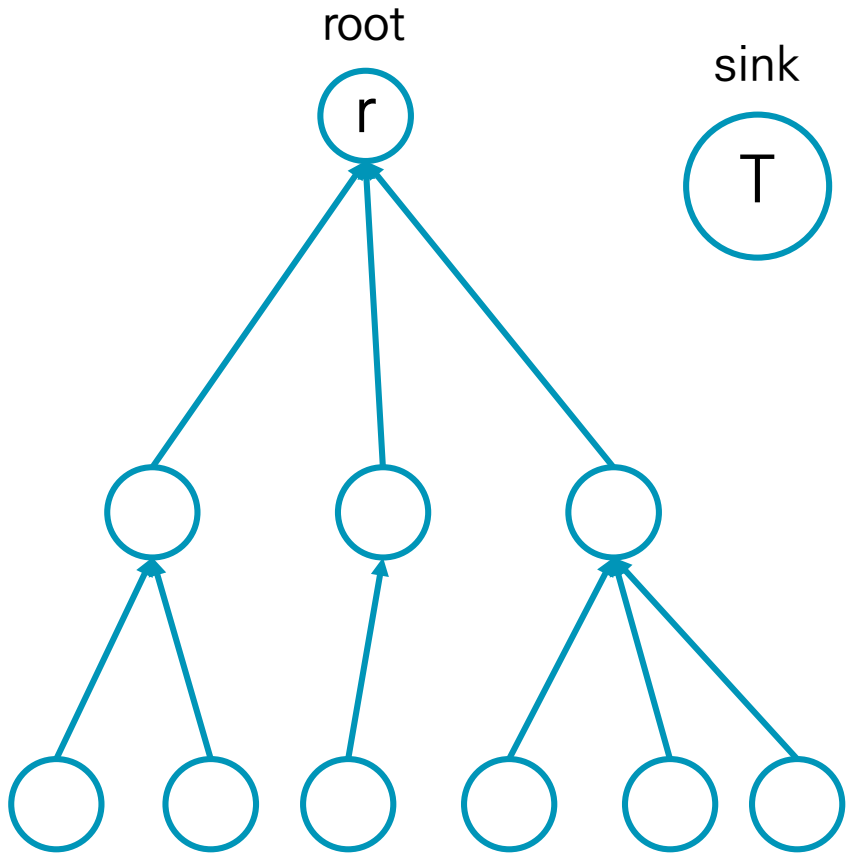
5번 - Covernent

- 주어진 트리에는 루트가 없으니, 뭔가 애매합니다.
- 아무 노드를 하나 루트로 잡고, 그 노드를 포함하는 트리 중에서 간선 가중치 합이 최대인 트리를 만들도록 해 봅시다.
- 우선 리프를 둘씩 짝지어 놓고 둘 중 하나를 선택해야 하니, 다음과 같이 더미 노드를 N개 만들어서 유량 1, 비용 0인 간선들을 이어 봅시다.



5번 - Covernent

- 이제 트리 내부의 각 간선은 루트를 향해 가는 방향으로 유량은 1, 비용은 -(그 간선 가중치) 로 정해서 이어 봅시다.
- 또한, 트리의 각 노드에서는 sink로 빠져나가는 유량 N (사실상 무한대), 비용 0의 간선을 잇습니다.
- 이렇게 만든 그래프에서 MCMF 알고리즘을 수행하면 최종적으로 구한 총 비용에 -1을 곱한 값이 루트를 포함하는 트리들 중 최대 가중치가 됩니다.



5번 - Covernent

- 문제가 하나 있는데, 루트를 어떤 노드로 해야 할지 모르기 때문에 MCMF를 총 O(M)번 해야 한다는 것입니다.
- 이것은 아주 간단한 아이디어로 해결 가능합니다.
- 1번과 N+1번 노드 둘 중에 하나는 무조건 골라야 하기 때문에, 이 둘을 각각 루트로 해 보면 2번만 MCMF를 돌려도 됩니다!
- 아까 각 리프에 연결해 준 더미 노드는 1번과 N+1번 쌍을 제외한 2(N-1)개의 노드들에 대해서만 해 주면 됩니다.

5번 - Covernent

- MCMF 알고리즘은 말로 하면 매우 간단합니다.
- s → t로 유량을 흘릴 수 있는 경로 중 cost 합이 가장 작은 경로 (최단경로)를 찾아서 흘려 주는 것을 더 이상 유량을 흘릴 수 없을 때까지 반복하면 됩니다.
- cost는 음수가 될 수 있기 때문에, Bellman-Ford 내지는 SPFA (Bellman-Ford의 개선된 버전. 대부분 이걸 씁니다) 알고리즘을 사용해서 최단경로를 찾아야 합니다.
- 자세한 것은 코드를 보시는 게 빠를 거 같습니다..

5번 - Covernent

- MCMF 알고리즘 전체 코드는 GitHub에 올려 놓은 것을 보시면 될 것 같습니다.
- MCMF 그래프를 구축하고 답을 출력하는 부분만 간단히 보겠습니다.
- Lambda 함수를 재귀함수로 짜려면 오른쪽의 f 함수와 같이 정의하면 됩니다.

```
ll ans = 0;
for(int r = 1; r <= n + 1; r += n) {
  MCMF::init(m + n + 2);

  function<void(int, int)> f = [&](int x, int y){
    for(pil &i : e[x]) {
      if(i.x == y) continue;
      MCMF::add(i.x, x, 1, -i.y);
      f(i.x, x);
    }
  };
  f(r, 0);

  for(int i = 2; i <= n; i++) {
    MCMF::add(m + n + 1, m + i, 1, 0);
    MCMF::add(m + i, i, 1, 0);
    MCMF::add(m + i, n + i, 1, 0);
  }
  for(int i = 1; i <= m; i++) {
    MCMF::add(i, m + n + 2, n, 0);
  }

  ans = max(ans, -MCMF::get(m + n + 1, m + n + 2));
  MCMF::clear();
}
cout << ans << '\n';
```