SCPC 4회 1차예선 풀이

서울대학교 컴퓨터공학부 18학번 김동현

1번 – 버스 타기

- N명의 선수들이 버스를 탄다.
- 각 선수들은 정수로 나타나는 실력 값을 가진다.
- 두 선수의 실력 값이 K 이하로 차이나는 경우 두 선수는 같은 버스에 타지 않는다.
- 각 버스의 정원은 무제한이라고 할 때, 모든 선수를 태울 수 있는 최소의 버스 대수를 구하여라.
- $1 \leq N \leq 200{,}000$

1번 - 버스 타기

- 가장 실력 값이 낮은 선수부터 봅시다.
- 그 선수는 일단 버스 하나에 타야 합니다.
- 그 버스에 태울 다른 선수들을 어떻게 뽑으면 좋을까요?
- 최대한 많이 태울 수 있다면 좋으니, 그 다음 선수를 바로 전에 태운 선수와 실력 값이 K 초과로 차이나면서 실력 값이 가장 작은 선수로 하면 좋을 것 같습니다.
- 버스를 하나 새로 도입할 때마다 가장 실력 값이 낮은 선수부터 태우고 위에 말한 대로 다음에 태울 선수를 반복적으로 선택해주면 됩니다.

1번 - 버스 타기

- 이 방법을 구현하려면 set을 사용해야 합니다.
- 사실, 자료구조 지식을 요구하지 않는 조금 더 쉬운 풀이가 있습니다.
  - 구현이 쉽다는 뜻입니다.
- 각 선수들의 실력 값을 오름차순 정렬해서 순서대로 A_1, A_2, ⋯, A_n이라고 합시다.
- 문제의 답은 $|A_i - A_j| \le K$인 (i, j)들 중 $(j - i + 1)$의 최댓값입니다.

1번 - 버스 타기

- (1) 답이 그것보다 작을 수 없음
- $|A_i - A_j| \le K$이고 $(j - i + 1)$이 최대인 (i, j)를 하나 잡았다고 합시다.
- 아래 그림과 같이, K 범위 내에 $(j - i + 1)$개의 실력 값이 모여 있습니다.
- 버스를 $(j - i + 1)$대보다 적게 준비하면, 비둘기집의 원리에 의해 이들 중 두 명은 같은 버스를 타야 하고, 이는 제한 조건에 위배됩니다.
- 따라서, 최소 $(j - i + 1)$대의 버스가 필요합니다.

1번 - 버스 타기

- (2) 가능한 버스 배치가 존재함
- $|A_i - A_j| \le K$인 (i, j) 중 (j − i + 1)의 최댓값을 M이라고 합시다.
- 버스를 M대 준비하고, A_1은 1번 버스, A_2는 2번 버스, …, A_M은 M번 버스, A_{M+1}은 다시 1번 버스, … 이런 식으로 순서대로 버스를 태웁시다.
- 이렇게 태우면 같은 버스에 탄 두 선수는 무조건 인덱스 차이가 M 이상 나기 때문에 M의 정의에 의해 실력 값이 K 초과로 차이가 나게 됩니다. 즉, M대의 버스로 모든 선수를 태울 수 있습니다.

1번 - 버스 타기

- 우리가 구해야 하는 것은 필요한 버스 대수 뿐이므로 매우 간단하게 구해 줄 수 있습니다.
- Two pointer 기법을 사용하면 O(N)에 가능합니다.

```cpp
void solve() {
  int n, k;
  cin >> n >> k;

  vint v(n);
  for(int &x : v) cin >> x;
  sort(all(v));

  int ans = 0;
  for(int i = 0, j = 0; i < n; i++) {
    while(j < n - 1 && v[j + 1] - v[i] <= k) j++;
    ans = max(ans, j - i + 1);
  }

  cout << ans << '\n';
}
```

2번 – 회문인 수의 합

- 회문(回文)인 수란, 거꾸로 써도 원래와 똑같은 수를 뜻한다.
  - 1, 232, 10001, 2345432, …
- 자연수 N이 주어지면, N을 최소 개수의 회문인 수들의 합으로 표현하였을 때 그 개수와 각 수들을 출력하라.
  - 단, 3개 초과로 필요하다면 -1을 출력하라.
- $1 \le N \le 10{,}000$

2번 - 회문인 수의 합

- (TMI) 임의의 자연수 N과 임의의 g ≥ 5에 대해 N을 g진법으로 회문인 수 3개의 합으로 표현할 수 있다고 합니다.
  - https://www.acmicpc.net/problem/18793
  - https://arxiv.org/pdf/1602.06208.pdf
  - 많이 심심하시면 읽어보세요..

- 물론 이 문제는 제한이 매우 작으니 훨씬 쉽게 풀 수 있습니다.

2번 - 회문인 수의 합

- 10,000 이하의 자연수 중 (10진법으로) 회문인 수는 200개 정도밖에 없습니다.
- 따라서, 모든 3개의 조합을 다 시도해 보아도 시간이 충분합니다.
- 회문인 수는 미리 구해서 전역 배열 같은 곳에 넣어두면 됩니다.

```
for(int i = 1; i < N; i++) {
  string s = to_string(i);
  string t = s;
  reverse(all(t));
  if(s == t) {
    pal.push_back(i);
    pchk[i] = 1;
  }
}
```

C++ 기준으로 회문인 수는
이렇게 체크할 수 있습니다.

2번 – 회문인 수의 합

- 저는 세제곱 말고 제곱 코드를 한 번 짜 보았습니다.

- 1번보다 2번이 쉬운 거 같습니다..

```
int n;
cin >> n;

if(pchk[n]) {
  cout << "1 " << n << '\n';
  return;
}

for(int x : pal) {
  if(x < n && pchk[n - x]) {
    cout << "2 " << max(x, n - x) << ' ' << min(x, n - x) << '\n';
    return;
  }
}

for(int x : pal) {
  for(int y : pal) {
    if(x + y < n && pchk[n - x - y]) {
      vint v = {x, y, n - x - y};
      sort(all(v));
      cout << "3 " << v[0] << ' ' << v[1] << ' ' << v[2] << '\n';
      return;
    }
  }
}

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

3번 - 우주 정거장

- 그래프를 아래의 두 단계에 걸쳐서 만들려고 한다.
  - 1단계 : 최초 구성을 만든다. 아무 연결 그래프를 만들 수 있다.
  - 2단계 : A-B 간선이 있을 때, 새로운 정점 X를 추가하여 A와 B에 각각 연결한다.
- 정점 N개, 간선 M개로 된 그래프의 최종 모양이 하나 주어진다.
- 그 그래프를 만들 수 있는 최초 구성들 중, 정점의 개수가 가장 적은 최초 구성의 정점 개수를 구하여라.
- $2 \le N \le 200{,}000$, $1 \le M \le 400{,}000$

3번 - 우주 정거장

- 최초 구성에서 최종 구성으로 갈 때 하는 연산은 간선 A-B가 있을 때 정점 X와 간선 X-A, X-B를 추가하는 연산 한 종류 뿐입니다.
- 이 작업을 반대로 해 나가면 그래프의 최초 구성을 찾을 수 있을 것 같습니다. 즉, Degree가 2인 정점에 대해 그 정점과 연결된 두 정점 간에 간선이 있다면 해당 정점을 그래프에서 제거하는 작업을 반복합니다.
- 그런데 이 작업을 아무렇게나 해도 될까요? 정점을 잘못 떼어 버려서 최소 개수의 정점을 남기지 못하는 경우가 있을까요?

3번 – 우주 정거장

- 어떤 정점 X가 어느 순간에 “제거 가능”해졌다면, 다른 정점을 건드려서 그 정점이 다시 “제거 불가능”해지는 경우가 있을까요?
- 그런 경우가 있긴 한데, 그 경우에 대해 자세히 살펴봅시다.
  - X를 가만히 놔뒀으므로 ‘제거 불가능해졌다’라는 것은 X에 연결된 정점 둘 중 하나가 없어졌다는 뜻입니다.
  - X–A, X–B 간선이 있었고, A가 없어졌다고 해 봅시다.
  - 그 순간 A는 X와 B하고만 연결이 되어 있었어야 합니다. 즉, A와 X는 완전히 똑같은 상황에 놓인 정점입니다! 둘 중 무엇을 제거해도 똑같습니다.
- 해당 경우를 빼면 한번 제거 가능해진 정점은 언제라도 제거할 수 있으니, 아무 순서로 제거를 수행해도 상관이 없습니다!

3번 – 우주 정거장

- 이런 식으로 그래프에서 특정 조건을 만족하는 정점을 제거하는 것을 반복적으로 수행하는 알고리즘은 C++ STL의 queue와 set을 사용하여 구현할 수 있습니다.
- queue는 현재 제거해야 할 정점 후보들을 관리하는 데 사용됩니다.
  - Queue에 넣어진 정점이 어떤 이유로 제거 불가능해졌을 수 있습니다.
  - 제거 가능 → 제거 불가능을 거친 정점은 무조건 끝까지 남아있게 됩니다. (Degree가 1이 되었기 때문)
- set은 간선의 삭제를 빠르게 처리하기 위해 사용됩니다.
  - 인접 리스트의 구현에서 각 정점이 vector<int> 대신 set<int>를 가집니다.

3번 – 우주 정거장

- 큐에서 뽑을 때 해당 정점을 실제로 지울 수 있는지 체크를 해야 합니다.
- 각 정점은 최대 한 번 큐에 들어갑니다.
- 시간 복잡도는 O(M+NlogN) 입니다.

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

vector<set<int>> e(n + 1);
for(int i = 0; i < m; i++) {
  int x, y;
  cin >> x >> y;
  e[x].insert(y);
  e[y].insert(x);
}

queue<int> q;
for(int i = 1; i <= n; i++) if(e[i].size() == 2) q.push(i);

int ans = n;
while(!q.empty()) {
  int x = q.front();
  q.pop();

  if(e[x].size() < 2) continue;
  int y = *e[x].begin();
  int z = *e[x].rbegin();
  if(!e[y].count(z)) continue;

  ans--;
  e[y].erase(x);
  e[z].erase(x);
  if(e[y].size() == 2) q.push(y);
  if(e[z].size() == 2) q.push(z);
  e[x].clear();
}
cout << ans << '\n';
```

4번 – 선형배치

- 정점 N개와 간선 M개로 이루어진 그래프가 주어진다.
- 그래프의 각 정점들을 수직선 위에 1의 간격으로 배치하려고 한다.
- 정점들을 잘 배치해서 간선 길이의 합이 최소가 되도록 하라.
- 그래프는 아래와 같은 방법으로 만들어진다.
  - 정점들을 미리 어떤 순서로 늘어놓고, 인접한 정점들 간에 우선 간선을 잇는다.
  - 2 이상의 k에 대해, k만큼 떨어진 두 정점 간에는 0.8^k 확률로 간선을 잇는다.
- 5 ≤ N ≤ 100, 4 ≤ M ≤ 1,000

4번 – 선형배치

- 점수 기준이 좀 이상합니다.
- 일단 결정론적으로 가장 좋은 답을 내놓는 알고리즘은 찾기 힘든 것 같습니다.
- 만점 기준이 “출제진이 찾은 답”이라니 참 막막합니다….
- 다행히도(?) 경험상 기준이 아주 빡빡하지는 않습니다.
- 보통 “좀 열심히 하면 만점이 나오는 정도”로 기준을 잡는 듯 합니다.
  - 말이 참 애매합니다….

4번 - 선형배치

- 이런 류의 문제를 푸는 기본은 “그럴듯한 그리디”를 하나 짜는 것입니다.
- 정점을 왼쪽부터 하나씩 배치한다고 합시다.
- 현재 정점 몇 개를 이미 배치했다고 합시다.
- 그 다음으로 올 정점은 어떤 게 좋을까요?
  - (1) 이미 배치한 정점과 간선이 많이 이어져 있을수록 좋습니다.
  - (2) 아직 배치하지 않은 정점과는 조금 이어져 있을수록 좋습니다.
- (배치된 인접한 정점 수) - (배치가 안 된 인접한 정점 수) 의 값이 가장 큰 정점을 매번 새로 추가하는 그리디를 짜 봅시다.
- 첫 번째 정점은 N개의 정점 각각을 한 번씩 시도해보면 됩니다.

4번 – 선형배치

- 이걸 짜서 내면 놀랍게도 196점 정도가 나옵니다. (만점 200점)
- 조금만 더 하면 될 것 같습니다….
- 제가 처음에 짰던 코드는 기준값이 최대인 정점이 여러 개 있을 때 가장 처음 발견한 정점을 골랐는데, 최대인 정점들 중 하나를 랜덤하게 고르는 코드로 수정하고 시도 횟수를 30N번으로 늘리니 (각 시작점당 30번씩) 199.99점이 나왔습니다..?
- 이제 진짜 조금만 더 하면 됩니다.

4번 – 선형배치

- 그리디 방식으로 한계에 부딪히면 그 다음으로 해볼 수 있는 것이 Local Search입니다.
- 기본적으로 답에 약간의 수정을 가해 본 뒤 더 좋아지면 그 쪽으로 가는 것을 반복하는 알고리즘입니다.
- 보통 랜덤한 두 원소를 잡아서 서로 자리를 바꿔보는 방법을 사용합니다.
- 마지막에 랜덤 swap을 2000번씩 시도해 보도록 수정한 결과 200점을 받을 수 있었습니다.

4번 – 선형배치

- 입력 부분 및 eval 함수를 정의하는 부분입니다.
- eval 함수는 정점 배치를 주면 그 배치에서 간선 길이의 합을 반환합니다.
- 이런 문제를 풀 때 답을 계산하는 함수는 (굳이 출력을 요구하지 않더라도) 일단 짜 놓고 시작하는 것이 좋습니다.

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

vector<vint> e(n);
for(int i = 0; i < m; i++) {
  int x, y;
  cin >> x >> y;
  e[x].push_back(y);
  e[y].push_back(x);
}

vint q(n);
auto eval = [&](vint &p) {
  for(int i = 0; i < n; i++) q[p[i]] = i;
  int ret = 0;
  for(int i = 0; i < n; i++) {
    for(int j : e[i]) {
      if(i < j) ret += abs(q[i] - q[j]);
    }
  }
  return ret;
};
```

4번 – 선형배치

- 그리디 알고리즘의 구현 부분입니다.
- N이 작으므로 약간 naive하게 구현하였습니다.
- 시도 횟수를 늘리고 싶다면 알고리즘을 효율적으로 작성하면 됩니다.

```
p.clear();
fill(all(chk), 0);
p.push_back(st % n);
chk[st % n] = 1;

for(int t = n - 1; t--; ) {
  int mx = -n;
  vint cur;
  for(int i = 0; i < n; i++) {
    if(chk[i]) continue;
    int cnt = 0;
    for(int j : e[i]) cnt += (2 * chk[j] - 1);
    if(mx <= cnt) {
      if(mx < cnt) cur.clear();
      mx = cnt;
      cur.push_back(i);
    }
  }
  shuffle(all(cur), mt);
  p.push_back(cur[0]);
  chk[cur[0]] = 1;
}
```

4번 - 선형배치

- Local Search 부분입니다.
- 정점 두 개 자리를 바꾸어 보고, 답이 더 좋아지면 바뀐 채로 두고, 그렇지 않으면 원래대로 돌립니다.
- rand() 함수는 되도록 쓰지 않는 것이 좋습니다. mt19937을 씁시다.

```
mt19937 mt(9949 * n + m);
uniform_int_distribution<int> rnd(0, n - 1);

    int tt = 2000;
    while(tt--) {
      int x, y;
      do {
        x = rnd(mt);
        y = rnd(mt);
      } while(x == y);

      swap(av[x], av[y]);
      int t = eval(av);
      if(ans > t) ans = t;
      else swap(av[x], av[y]);
    }
```

4번 – 선형배치

- 사실 앞에서 설명했던 방법은 제가 며칠 전에 다시 풀어볼 때 사용했던 방법이고, 대회 당시에는 다른 방법으로 만점을 받았습니다.
- Simulated Annealing이라고 불리는 기법이 있습니다.
- 기본적으로는 Local Search와 유사하게 답에 약간의 변형을 가해 보는 식인데, 차이점은 더 나빠지는 경우에도 가끔씩 가본다는 것입니다.
- ‘온도’라는 개념을 도입하여, 초기에는 온도가 높아서 답이 나빠지는 쪽으로 자주 가지만, 갈수록 온도가 식어서 좋아지는 쪽으로만 향하게 됩니다.

4번 – 선형배치

- 전체 코드를 GitHub에서 확인할 수 있습니다.
- 대회 당시 제출한 코드를 그대로 넣은 거라서 지금 코드와 조금 다르게 생겼습니다.
- 시도 횟수를 최대한으로 하기 위해서는, 배치에 약간의 변형을 가했을 때 (예를 들어, 정점 두 개 자리를 바꾸었을 때) 답의 변화를 빠르게 계산하는 것이 필요합니다.
  - 이 문제에서는 그래프에서 두 정점 위치를 바꿨을 때 길이가 바뀌는 간선은 위치가 바뀐 두 정점 중 하나에 붙어 있던 간선밖에 없다는 사실을 이용하면 됩니다.

5번 - Lights to Stage

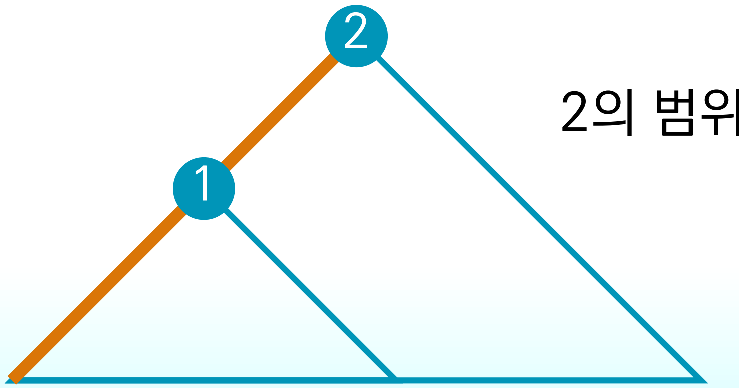
- 무대는 (0, 0) - (N, 0) 을 잇는 선분입니다.
- 무대 위, y좌표가 양수인 부분에 전기선이 놓여 있습니다.
  - 전기선은 M개의 선분으로 나눌 수 있습니다.
  - 선분 끝점의 좌표는 모두 정수입니다.
  - 인접한 두 선분은 끝점을 공유하며, 각 선분의 기울기는 +1 또는 −1입니다.
- 전기선의 어느 부분에 전등을 놓으면 그 전등은 좌우 45° 각도로 아래를 비춥니다.
- 전등을 최대 L개 설치하여 무대를 모두 비추고자 할 때, 놓은 전등들의 y 좌표 중 최댓값을 최소화하세요.

5번 – Lights to Stage

- 최댓값을 최소화하라고 하면 답에 대해 이분탐색을 하라는 뜻으로 읽으시면 됩니다.
- 그런데 답을 기약분수 형태로 출력하라고 합니다. 이런 상황에서 이분탐색을 할 수 있을까요…?
- 사실, 답은 정수 k에 대해 $\frac{k}{2}$ 꼴입니다. 즉, 모든 좌표에 2를 곱하면 답이 무조건 정수가 됩니다.
  - 전등을 놓는 선분들의 기울기가 +1 또는 −1이고, 끝 점 좌표가 모두 정수라서 성립하는 사실입니다.
  - 증명은 생략합니다. (저는 대회 칠 당시 대충 때려맞혀서 풀고 맞았습니다..)

5번 – Lights to Stage

- 아무튼 이제 답이 정수라는 가정 하에 문제를 풀어 봅시다.
- 답이 X 이하인지, 즉 y좌표가 X 이하인 곳에만 전등을 놓아 무대 전체를 밝힐 수 있는지 판단한다고 합시다.
- 우선, 각 선분마다 전등을 2개 이상 놓을 필요가 없다는 사실을 알 수 있습니다. 각 선분에서 (높이 제한에 걸리지 않는 선에서) 전등을 위로 최대한 끌어올리면 그보다 아래에 놓았을 때보다 무조건 좋기 때문입니다.



2의 범위가 1의 범위를 포함.

5번 - Lights to Stage

- 각 선분마다 전등을 놓을 위치가 0개(선분이 너무 높이 있을 경우) 또는 1개이므로, 전등 위치의 후보는 O(M)개 나옵니다.
- 각 위치에 전등을 놓았을 때 무대에서 밝혀지는 구간을 쉽게 알 수 있으므로, 결국 O(M)개의 구간들 중 최대 L개를 선택해 [0, N]을 완전히 덮을 수 있는가라는 문제를 풀면 됩니다.
- 이것은 좌표압축+DP 또는 Greedy를 통해 풀 수 있습니다.
- Greedy가 코딩이 조금 더 편한 듯 합니다.

5번 – Lights to Stage

- 입력 및 이분 탐색 부분입니다.
- f(x)는 답이 x 이하가 될 수 있다면 true, 아니면 false를 반환합니다.
- 처음에 모든 좌표에 2를 곱하고 시작했으므로 마지막에는 다시 2로 나누는 작업이 필요합니다.
- 무대 전체를 밝히는 것이 아예 불가능할 때도 있는데, 이것도 처리해야 합니다.

```
int m, k;
ll n;
cin >> n >> m >> k;
n *= 2;
m++;

vpll v(m);
for(pll &p : v) {
  cin >> p.x >> p.y;
  p.x *= 2;
  p.y *= 2;
}
```

```
ll l = 0, r = ll(3e12);
while(l < r) {
  ll x = (l + r) / 2;
  if(f(x)) r = x;
  else l = x + 1;
}

if(l > ll(2e12)) { cout << "-1\n"; }
else {
  int odd = l % 2;
  cout << (l / (2 - odd)) << ' ' << (1 + odd) << '\n';
}
```

5번 - Lights to Stage

- f 함수의 앞부분입니다.
- 전기선을 이루는 각 선분마다 그 선분에서 전등을 놓을 위치를 구한 뒤, 그 위치에 해당하는 무대 구간을 구합니다.
- 기울기가 +1일 때와 −1일 때 케이스를 나누기 싫어서 lo(낮은 쪽 끝점)와 hi(높은 쪽 끝점) 변수를 도입하였습니다.

```cpp
auto f = [&](ll x) {
  vpll w;
  for(int i = 0; i + 1 < m; i++) {
    pll lo = (v[i].y < v[i + 1].y ? v[i] : v[i + 1]);
    pll hi = (v[i].y < v[i + 1].y ? v[i + 1] : v[i]);
    if(lo.y > x) continue;

    pll cur;
    if(hi.y > x) cur = pll(v[i].x + abs(x - v[i].y), x);
    else cur = hi;
    w.emplace_back(cur.x - cur.y, cur.x + cur.y);
  }
  sort(all(w));
```

5번 - Lights to Stage

- f 함수의 뒷부분입니다.
- 무대를 왼쪽부터 커버해 나가는 그리디 알고리즘입니다.
- 매번 전등을 새로 고를 때 마다,
  - 현재까지 커버한 가장 오른쪽 x좌표를 포함하며 (빼먹는 구간이 없으며)
  - 오른쪽으로 가장 많이 뻗어나가는

  구간을 고릅니다.
- 전등을 몇 개 썼는지도 체크합니다.

```
ll lst = 0;
int cnt = 0;
for(int i = 0; i < w.size() && cnt < k; ) {
  ll mx = lst;
  while(i < w.size() && w[i].x <= lst) {
    mx = max(mx, w[i].y);
    i++;
  }
  if(mx == lst) return false;
  cnt++;
  lst = mx;
  if(lst >= n) return true;
}
return false;
```