SCPC 2회 1차예선 풀이

서울대학교 컴퓨터공학부 18학번 김동현

테스트 케이스 그룹

- 2회 예선부터는 각 문제의 테스트 케이스가 그룹별로 나뉘어 있고, 각 그룹에 배당된 점수는 그 그룹의 테스트 케이스를 모두 맞히면 받을 수 있습니다.
- 그런데 SCPC는 테스트 케이스가 하나의 파일로 묶여 있어서, 일부 그룹만 해결하는 코드를 냈을 경우에는 제한 시간 내에 내놓은 출력만 가지고 채점을 수행합니다.
- 콘솔 입출력 함수는 버퍼를 사용하기 때문에, printf 등의 함수로 출력을 요청한다고 해서 바로 출력이 되는 것이 아닙니다. 만약 큰 데이터에서 제한 시간 내에 종료하지 않는 코드를 낸다면, 출력이 버퍼에 남아 있는 상태에서 실행이 종료될 수 있습니다.

출력 버퍼 관리하기

- `setbuf(stdout, NULL)`
  - `stdout` (표준 출력) 에 해당하는 buffer를 제거합니다.
  - 맨 처음에 한 번 실행해주면 이후 모든 출력이 buffer 없이 바로 출력됩니다.
  - `cin` / `cout`을 사용하는 경우 (아마) 제대로 작동하지 않습니다.
- 버퍼 직접 비우기
  - `printf` 사용 경우 : `fflush(stdout);`
  - `cout` 사용 경우 :
    `cout << std::endl` (줄바꿈 + flush)
    `cout << std::flush` (버퍼만 비움)
- 만점 코드를 짜면 이런 거 안 해도 됩니다 ^^;;

1번 - 3N+1

- 양의 정수 N에 대해 F(N)을 다음과 같이 정의합니다.

$$F(N) = \begin{cases} \frac{N}{2}, & N \text{ even} \\ 3N+1, & n \text{ odd} \end{cases}$$

- 어떤 정수 N에 대해, N → F(N)을 반복적으로 적용했을 때 N이 1이 되기까지 걸리는 단계 수를 g(N)이라고 합시다.
  - Ex) 10 → 5 → 16 → 8 → 4 → 2 → 1이므로 g(10) = 6
- 정수 K (1 ≤ K ≤ 63) 가 주어지면, g(N)=K를 만족하는 N의 최솟값과 최댓값을 각각 구하세요.

1번 - 3N+1

- N의 최댓값부터 생각해 봅시다.
- N→F(N)의 과정에서 수는 커지거나(3N+1) 줄어듭니다(N/2).
- 어떤 N에서 시작해서 N → N/2만 K번 반복해서 1이 되었다면, 그 N이 g(N)=K인 N 중 최댓값일 것입니다.
- 최댓값은 2^K가 됩니다.
  - K가 최대 63이라 C++의 경우 unsigned long long 자료형을 사용해야 합니다.

1번 - 3N+1

- 최솟값은 조금 더 어려운 것 같습니다.
- 잘 모르겠으니 코드를 짜서 실험을 해 봅시다..?

```cpp
int main() {
  ios::sync_with_stdio(false);
  cin.tie(nullptr);

  int cnt = 0;
  for(int i = 1; cnt < 64; i++) {
    int c = 0, x = i;
    while(x > 1) {
      if(x & 1) x = 3 * x + 1;
      else x /= 2;
      c++;
    }
    if(c <= 63 && !ans[c]) {
      cout << c << " : " << i << '\n';
      ans[c] = i;
      cnt++;
    }
  }
```

```
43 : 540
56 : 569
51 : 641
59 : 758
62 : 1010
57 : 1138
```

1 ≤ K ≤ 63에 대한 모든 답이
1138 이하의 정수에서
모두 나타납니다!

1번 - 3N+1

- 약간 어이 없는 풀이이긴 하지만, 아무튼 N을 1부터 시작해서 하나씩 늘려 나가면서 g(N)을 계산하다 보면 제한 조건 내의 모든 K에 대해 최솟값을 구할 수 있습니다.
- 지금 같은 경우에는 답을 구하는 속도가 매우 빠르기 때문에 제출하는 코드에서 답을 실시간으로 구해도 되지만, 조금 더 오래 걸리는 경우에는 로컬에서 미리 답을 구한 뒤, 결과값만 배열에 저장해서 출력해도 됩니다.
  - 이런 방법을 DB(DataBase)라고 부르기도 합니다.

1번 - 3N+1

- 전처리(각 K에 대한 최솟값 구하기)를 수행하는 코드입니다.
- N에 해당하는 변수(i)를 1부터 하나씩 시도해 보면서 모든 K에 대해 답을 다 구할 때까지 반복문을 수행합니다.
- 각 N에 대해서는 시뮬레이션을 한 단계씩 수행합니다.
- 각 TC를 처리하기 전에 딱 한 번 실행하면 됩니다.

```
int cnt = 0;
for(int i = 1; cnt < 64; i++) {
  int c = 0, x = i;
  while(x > 1) {
    if(x & 1) x = 3 * x + 1;
    else x /= 2;
    c++;
  }
  if(c <= 63 && !ans[c]) {
    ans[c] = i;
    cnt++;
  }
}
```

1번 - 3N+1

- 각 TC에 대한 답을 출력하는 코드입니다.
- 미리 구해놓은 답을 출력하면 끝입니다.

- 리터럴(상수) 값 뒤에 UL 접미사를 붙이면 그 상수를 unsigned long long type으로 취급합니다.
  - 정수의 기본 type은 int입니다.

```
void solve() {
  int k;
  cin >> k;
  cout << ans[k] << ' ' << (1UL << k) << '\n';
}
```

2번 – 징검다리

- 1~N까지 번호가 붙은 N개의 돌이 순서대로 놓여 있습니다.
- 출발지는 1번 돌 바로 전에 있습니다. (0번 돌이라고 생각)
- 몇 개의 돌에는 지뢰가 있어 밟을 수 없습니다.
  - 출발지와 N번 돌에는 지뢰가 없습니다.
- 한 번에 최대 K칸 떨어진 돌로 점프할 수 있습니다.
- 두 번의 연속한 점프를 같은 칸만큼 뛰면 저격수에게 저격당합니다.
- 저격당하지 않고 출발지에서 N번 돌까지 가는 경우의 수를 구하세요.
- $1 \leq N \leq 50{,}000$, $1 \leq K \leq 100$

2번 - 징검다리

- 경우의 수를 구하라니까 DP가 하고 싶어집니다.
- DP 식부터 정의를 해 봅시다.
- 인접한 두 번의 점프를 같은 칸 만큼 뛰면 안 되니까 현재 위치와 바로 전에 몇 칸 뛰었는지를 parameter로 넣으면 될 것 같습니다.
- `D[i][j]`를 "현재 `i`번 칸에 있고, 바로 전에 `j`칸만큼 뛴 경우의 수"로 정의합시다.
- 답은 `D[n][*]` 들의 합입니다.
- 기저 상태는 `D[0][0]=1`, `D[0][i]=0` (`i` 〉 0) 으로 정의할 수 있습니다.

2번 – 징검다리

- D[i][j]를 어떻게 구할 수 있을까요?
- 현재 i번째 칸에 있고 마지막으로 뛴 점프가 j칸짜리 점프이니 바로 전 상태는 D[i-j][k] 꼴일 것입니다.
- k는 j와 같지만 않으면 어떤 수든 다 가능합니다.
- 즉, D[i][j] = D[i-j][0]+D[i-j][1]+…+D[i-j][j-1]+ D[i-j][j+1]+…+D[i-j][min(K,i-j)] 입니다.

- j > min(i,K)이거나 i번째 칸에 지뢰가 있을 경우에는 D[i][j] = 0 입니다(정의되지 않습니다).

2번 – 징검다리

- 이 식 그대로 DP 항 하나를 계산하는 데는 O(K) 만큼의 시간이 듭니다.
- 항을 총 O(NK)개 계산해야 하니 O(NK^2)입니다.
- 그대로 하면 시간초과가 날 것 같습니다….
- DP 식을 살펴보면, `D[i-j][*]`들 중 하나만 빼고 다 더하는 형태입니다.
- `S[i] = D[i][0]+D[i][1]+ … +D[i][min(k,i)]`라고 합시다.
- `D[i][j] = S[i-j] – D[i-j][j]` 가 됩니다.
- 이제 `D[i][j]`의 각 항을 O(1)에 계산할 수 있습니다!
- `S[i]`는 `D[i][j]`를 하나 구할 때마다 누적해주면 됩니다.

2번 – 징검다리

- M (모듈러) 변수는 다음과 같이 코드 맨 위에 선언해 둘 수 있습니다.
  - `#define M 1000000009`
  - `const int M = int(1e9)+9;`
- C(C++)에서는 음수에 % 연산을 취하면 음수가 나올 수 있습니다. 따라서 뺄셈이 들어간 식에 모듈러를 취할 때에는 양수가 되도록 해 주어야 합니다.

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

  vint v(n + 1, 0);
  for(int i = 0; i < m; i++) {
    int x;
    cin >> x;
    v[x] = 1;
  }

  d[0][0] = s[0] = 1;
  int ans = 0;
  for(int i = 1; i <= n; i++) {
    s[i] = 0;
    for(int j = 1; j <= i && j <= k; j++) {
      if(v[i]) d[i][j] = 0;
      else d[i][j] = (s[i - j] - d[i - j][j] + M) % M;
      s[i] = (s[i] + d[i][j]) % M;
      if(i == n) ans = (ans + d[i][j]) % M;
    }
  }

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

3번 – 바이러스

- V개의 정점과 E개의 양방향 간선으로 이루어진 그래프가 주어집니다.
  - $1 \le V \le 100$, $1 \le E \le V(V-1)/2$
- 정점을 0개 이상 없애서 남는 그래프가 다음 조건을 만족하도록 만들려고 합니다. (정점을 하나 없앨 때마다 그 정점에 붙은 간선이 같이 없어진다고 합시다)
  - 각 정점은 자신과 연결된 정점이 최소 K개 있어야 합니다.
  - 각 정점은 남은 정점 중 자신과 연결되지 않은 정점이 최소 L개 있어야 합니다.
- 위 조건을 만족하면서 없애는 정점의 개수가 최소일 때, 그 때 없어진 정점들의 번호 합을 구하세요.
  - 답이 여러 가지일 경우, 번호 합이 최소인 때의 답을 구하세요.

3번 – 바이러스

- 처음 주어진 그래프에서 각 정점이 주어진 조건을 만족하는지 살펴봅시다.
- 지금 상태에서 연결된 정점이 K개 이하이거나 연결되지 않은 정점이 L개 이하인 정점은 앞으로 무슨 짓을 해도 절대 조건을 만족할 수 없습니다.
  - 정점을 하나 지워도 남아 있는 각 정점에 대해 연결된 정점의 수와 연결되지 않은 정점의 수는 늘어날 수 없기 때문입니다.
- 즉, 조건을 만족하지 않는 정점이 하나라도 있다면 그 정점은 일단 지우고 생각해도 좋습니다.

3번 – 바이러스

- 조건을 만족하지 않는 정점 하나를 찾아서 지우는 것을 반복하다가, 더 이상 그런 정점이 존재하지 않으면 그 때 멈추면 될 것 같습니다.
- 이 때 최종 상태가 유일할까요?

> 가장 적은 정점을 없애면서 조건을 만족시킨 서로 다른 최종 상태가 2개 있다고 합시다.
> 각각에 대해 남은 정점의 집합을 A, B라고 합시다. $A \neq B$ 이므로, $|A \cup B| > \max\{|A|, |B|\}$입니다.
> 또한, $A \cup B$에 속한 각 정점은 A (또는 B) 에서 이미 조건을 만족하므로 $A \cup B$에서도 역시 조건을 만족합니다. 따라서, $A \cup B$ 역시 조건을 만족시키는 집합이며, A나 B보다 더 적은 정점을 없애면서 조건을 만족시킨 최종 상태가 되므로 가정에 모순이 됩니다.

- 최종 상태는 유일하게 결정됩니다!

3번 – 바이러스

- 조건을 만족하지 않는 정점을 찾는 것을 어떻게 구현할까요?
- 인접 행렬을 사용하면 O(N^2)만에 조건을 만족하지 않는 정점을 하나 찾고, O(N)만에 그 정점에 연결된 간선을 인접행렬에서 제거할 수 있습니다. 총 시간복잡도는 O(N^3)입니다.
- 아마 이렇게만 짜도 N이 100 이하라서 통과할 듯 하지만, 저는 왠지 모르게 O(N^2logN) 코드를 짰습니다.

- O(N+M) 내지는 O((N+M)logN)에 풀어 보려고 시도를 해 보았는데, 잘 안 되는 듯 합니다. 혹시 방법을 발견하시면 알려 주시기 바랍니다….

3번 – 바이러스

- 그래프 입력을 받는 부분입니다.
- 인접 리스트에서 간선의 삭제를 빠르게 구현하기 위해 `std::set`을 사용하였습니다.

```
int k, l, n, m;
cin >> k >> l >> 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);
}
```

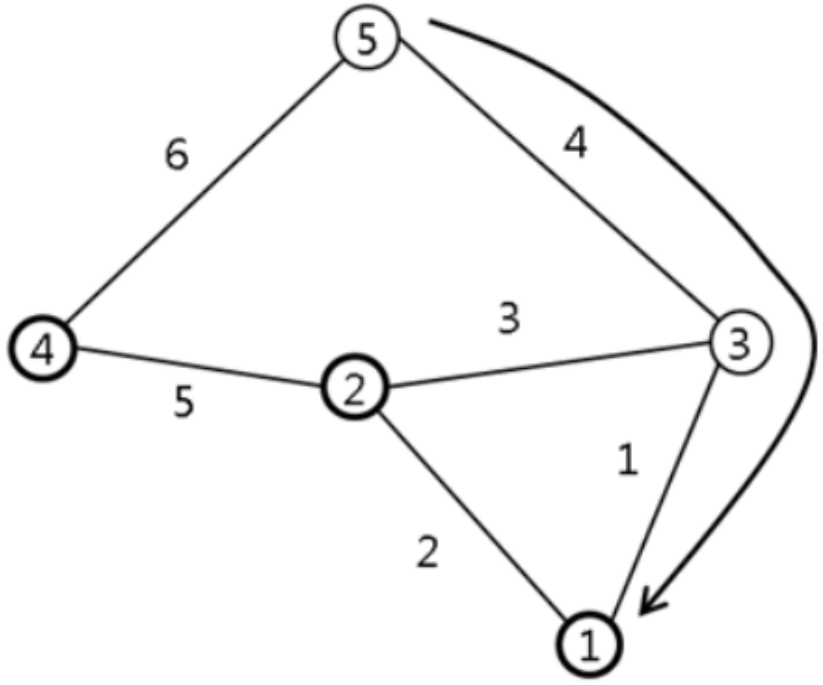
3번 – 바이러스

- 조건을 만족하지 않는 정점을 찾아서 그래프에서 지우는 것을 반복합니다.
- 양방향 간선이므로 자신과 인접한 정점들의 인접 리스트에서 자신의 번호를 지워 주면 됩니다.

```
int sz = n, ans = 0;
vector<int> chk(n + 1, 0);
while(true) {
  int x = 0;
  for(int i = 1; i <= n; i++) {
    if(chk[i]) continue;
    int deg = e[i].size();
    if(deg < k || deg >= sz - 1) { x = i; break; }
  }
  if(!x) break;
  ans += x;
  chk[x] = 1;
  sz--;
  for(int y : e[x]) e[y].erase(x);
}
cout << ans << '\n';
```

4번 - 대피소

- 정점 N개, 간선 M개인 연결된 무방향 가중치 그래프가 주어집니다.
  - 1 ≤ N ≤ 100,000, 1 ≤ M ≤ 500,000
- 정점들 중 K개는 대피소입니다.
- 모든 정점에서 (가장 가까운 대피소까지의 거리)의 합과 (가장 가까운 대피소의 번호)의 합을 구하세요.
  - 가장 가까운 대피소가 2개 이상이라면 번호가 가장 작은 것을 택합니다.
  - 대피소로 지정된 정점은 자기 자신 (거리 0)이 제일 가까운 대피소입니다.



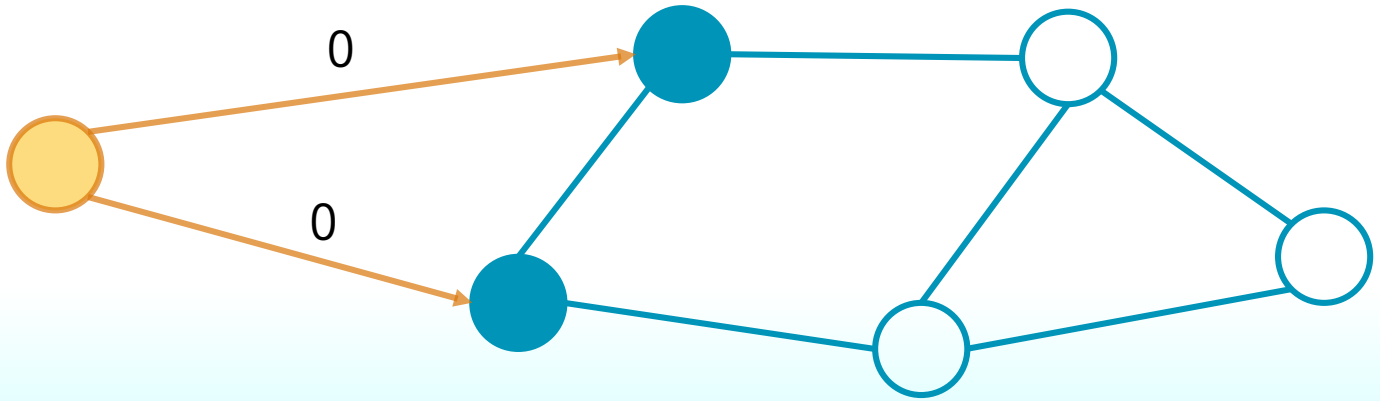
4번 – 대피소

- 그래프가 매우 큽니다.
- 다익스트라 알고리즘 한 번 겨우 돌릴 수 있을 정도의 제한입니다.
- 대피소가 여러 개 일 수 있는데 다익스트라 한 번으로 모두 찾을 수 있을까요?
- 가장 가까운 대피소의 “번호”는 어떻게 구할까요? 같은 거리인 대피소가 여러 개 있을 때 가장 작은 번호를 어떻게 구할까요?

- C++ STL을 잘 이용하면 다익스트라 알고리즘을 조금만 응용해서 아주 쉽게 할 수 있습니다.

4번 – 대피소

- 가능한 시작점이 여러 개인 다익스트라는 시작점이 한 개인 다익스트라와 전혀 다르지 않습니다.
- 처음에 heap에 정점을 넣을 때 출발점들을 한 번에 다 넣어주면 됩니다.
- 가상의 출발점 노드를 하나 만들고, 거기서 길이 0짜리 (단방향) 간선을 각 대피소에 이어주었다고 생각하면 편합니다.



4번 – 대피소

- 각 정점에서 가장 가까운 대피소까지의 거리와 그 번호를 동시에 저장해야 할 때는 `std::pair`를 이용하면 편합니다.
- 다익스트라 알고리즘에서 거리를 (정점부터의 거리, 시작 정점 번호)의 pair로 정의하면 문제를 한 번에 해결할 수 있습니다!
  - pair 두 개를 비교할 때는 첫 번째 원소를 먼저 비교하고 같을 경우에만 두 번째 원소를 비교합니다. 즉, 다익스트라 알고리즘을 수행한 뒤 각 정점의 최단 거리 배열에는 (가장 가까운 대피소까지의 거리, 가장 가까운 대피소들 중 번호가 가장 작은 대피소의 번호) 의 순서쌍이 들어있게 됩니다.
- 말로 하면 좀 복잡한데, 코드를 보면 매우 짧습니다.

4번 – 대피소

- 입력을 받는 부분입니다.
- 인접 리스트의 각 원소는 (도착 정점, 간선 가중치)의 pair입니다.

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

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

vint v(k);
for(int &x : v) cin >> x;
```

4번 – 대피소

- 다익스트라 알고리즘의 수행 부분입니다.
- 모든 대피소에 대해 (0, 대피소 정점 번호) 를 초기화 해 줍니다.
- 다익스트라 알고리즘에서 거리 갱신이 일어날 때는 오른쪽 코드와 같이 거리 값은 더해 주고, 시작 정점 번호는 그대로 전달해 주면 됩니다.

```
priority_queue<dijk, vector<dijk>, greater<dijk>> pq;
vpii d(n + 1, pii(int(2e9), 0));
for(int &x : v) {
  d[x] = pii(0, x);
  pq.emplace(d[x], x);
}

while(!pq.empty()) {
  dijk c = pq.top();
  pq.pop();
  if(c.x != d[c.y]) continue;

  for(pii &i : e[c.y]) {
    pii nd = pii(c.x.x + i.y, c.x.y);
    if(d[i.x] > nd) {
      d[i.x] = nd;
      pq.emplace(nd, i.x);
    }
  }
}
```

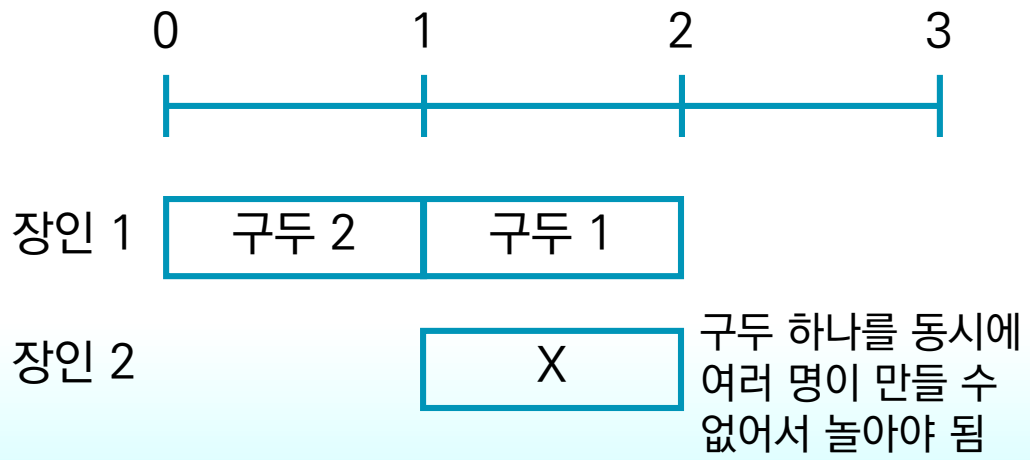
5번 – 구두제작

- N켤레의 구두를 K명의 장인들이 만들려고 합니다.
  - $1 \le N \le 200, 1 \le K \le 100$
- 각 구두는 a_i 시간에 주문이 들어와서 f_i 시간까지 완성되어야 하며, p_i 만큼의 제작 시간을 들여야 완성됩니다.
- 각 장인은 s_j 시간부터 e_j 시간까지 일할 수 있으며, 매 순간 하나의 구두 제작에 참여할 수 있습니다. 각 장인은 1초 단위로 제작할 구두를 하나씩 선택해서 작업할 수 있습니다.
- 모든 시간 값은 0 이상 100 이하입니다.
- 모든 구두를 주문 사항에 맞추어 완성할 수 있는지 여부를 출력하세요.

5번 – 구두제작

- 뭔가 그리디가 될 것 같이 생겼습니다.
- Deadline first로 그리디를 해 봅시다.
- 각 시간대마다 현재 일할 수 있는 장인들에게 아직 작업이 남은 구두 중 마감 기한이 빠른 것 부터 하나씩 배정을 해 봅시다.
- 그럴 듯 하죠? 예제도 잘 나옵니다.
- 이제 내면 64점을 받습니다.
- 이런 케이스에서 틀립니다 →

```
1
2 2
0 3 2
0 2 1
0 2
1 2
```



5번 – 구두제작

- 그리디 기준을 바꿔 보고 해도 잘 안 되는 것 같습니다….
- 다른 방법이 있을까요?
- 이런 식으로 “작업”을 배당하는 문제 같은 경우에는 최대 유량 문제(일명 플로우)로의 변환을 생각해 볼 수 있습니다.
- 최대 유량 문제로 변환하려면 문제 상황에 해당하는 그래프를 먼저 구축해야 합니다.
- 그래프를 어떻게 구축할 수 있을까요?

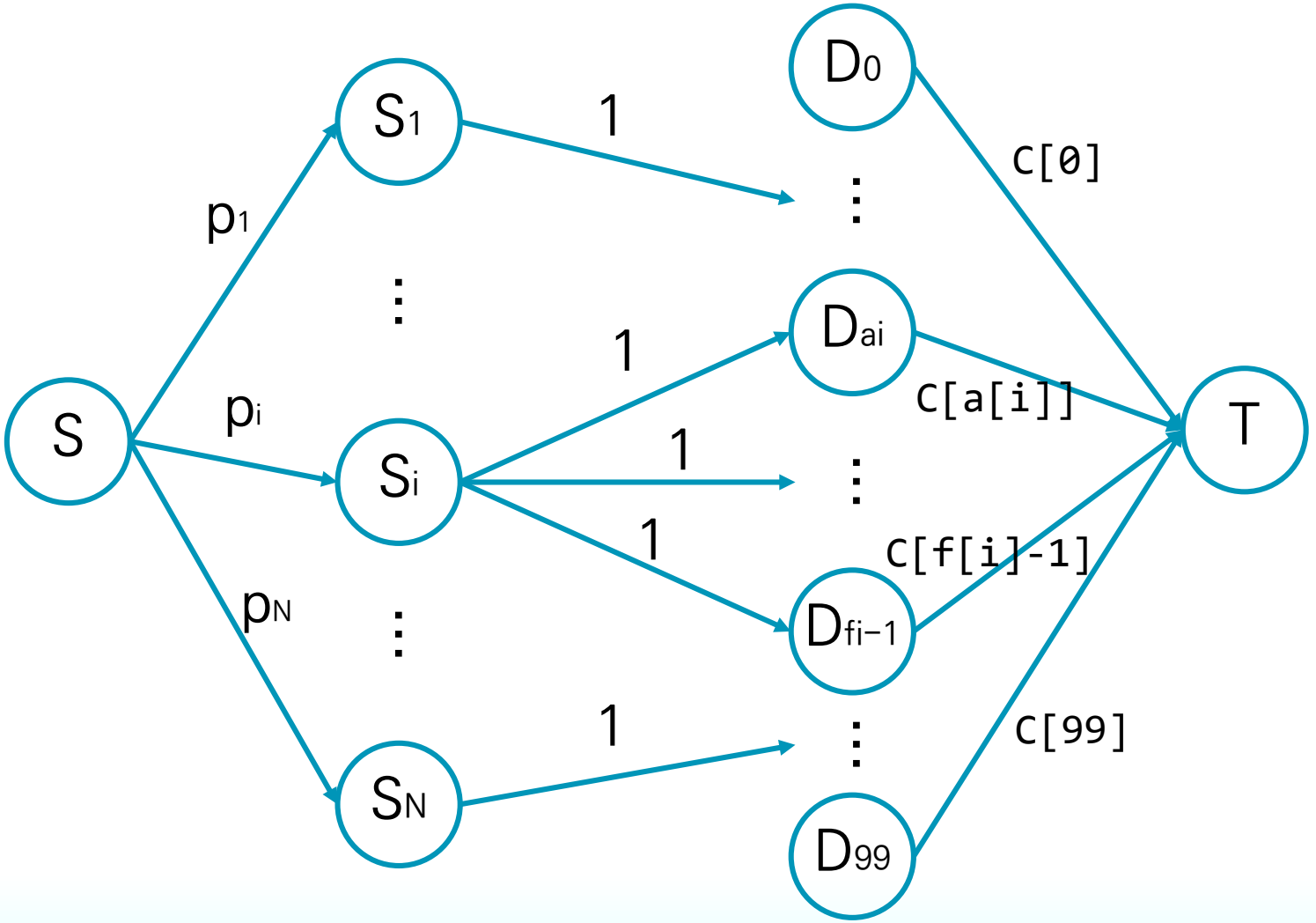
https://justicehui.github.io/ps/2019/09/10/BOJ1420/

5번 – 구두제작

- 우선, 장인에 관한 정보는 각 시간대마다 몇 명이 일 할 수 있는지만 중요합니다. `C[i]`를 시간 (`i ~ i+1`)에 일할 수 있는 장인의 수라고 합시다.
- 이제 문제는 다음과 같이 바뀝니다.
  - 각 신발을 a_i ~ f_i 시간 사이에 있는 시간대 중 p_i개의 시간대에 배정해야 합니다.
  - 시간 `i ~ i+1`에는 최대 `C[i]`개의 신발을 배정할 수 있습니다.
  - 모든 신발을 필요한 시간대만큼 다 배정할 수 있는지 여부를 판단해야 합니다.
- 노드와 간선을 적절히 설정해서 그래프의 각 유량이 작업 배정 하나에 대응되도록 해 봅시다.

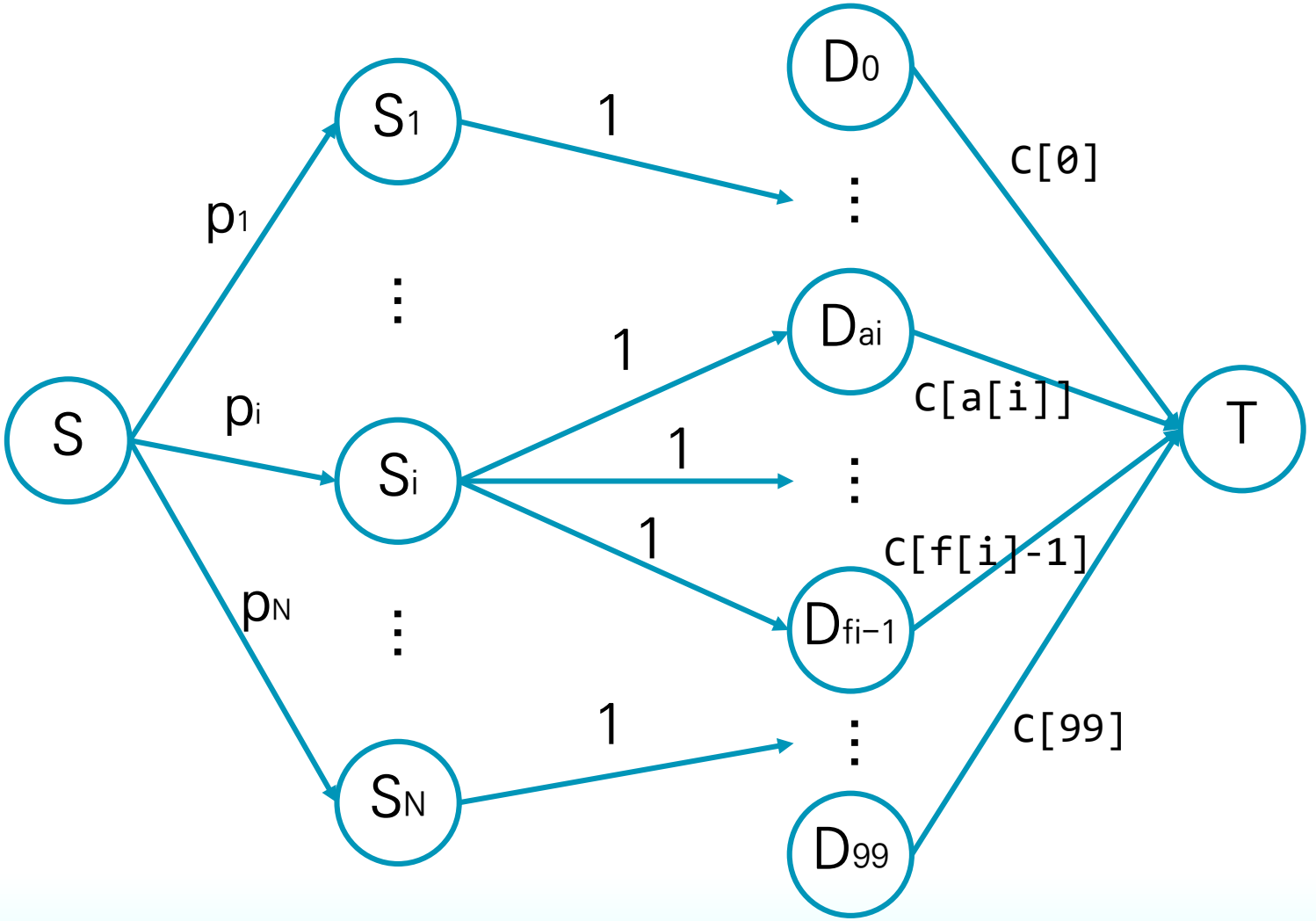
5번 – 구두제작

- 이런 그래프를 만들면 됩니다.
- S_i는 각 신발에 대응하는 정점, D_j는 각 시간대에 대응하는 정점입니다.
- S (source) 에서 S_i로 필요한 작업량 (p_i) 만큼의 유량을 가진 간선을 잇습니다.
- S_i에서는 $[a_i, f_i-1]$ 구간에 있는 각 D_j 정점에 유량 1짜리 간선을 잇습니다.
- D_j에서는 T(sink)로 그 때 일하는 장인의 수 (`c[j]`) 만큼의 유량을 가진 간선을 잇습니다.



5번 – 구두제작

- 이 그래프에서 S → T로 흘릴 수 있는 최대 유량을 구해서 그것이 모든 p 값들의 합과 같으면 모든 신발을 제한 시간 내에 만들 수 있는 것입니다.
- 이제 주어진 그래프에서 최대 유량을 어떻게 구할 수 있는지 알아봅시다.



5번 – 구두제작

- 그래프의 최대 유량을 구하는 알고리즘은 다양하게 있습니다.
- 가장 빠르다고 알려진 알고리즘은 Dinic 알고리즘인데, 조금 복잡합니다.
  - 저도 잘 모릅니다..
- 기본적으로 유량을 흘릴 수 있는 경로를 찾아서 흘려주는 것을 더 이상 못 할 때까지 반복하면 최대 유량을 구할 수 있습니다. (Ford-Fulkerson 알고리즘)
  - 그 경로를 DFS로 찾으면 O(FE) (F는 최대 유량)
  - 경로를 BFS로 찾으면 O(VE^2)
    (이것을 특별히 Edmond-Karp 알고리즘이라고 부릅니다.)
- DFS로 구현하는 것이 약간 더 간단합니다. (사실 별 차이 없습니다)

5번 - 구두제작

- 이 문제 같은 경우는 그래프의 간선 수와 최대 유량 모두 최대 100^2 정도가 될 수 있습니다.
- O(EF)나 O(VE^2)이나 이론상으로는 시간 초과가 날 거 같습니다…?
- 보통 최대 유량 알고리즘은 대부분의 경우 최악 시간 복잡도보다 훨씬 빨리 동작합니다. 그래서 보통 문제 제한을 가지고 시간 복잡도를 계산하는 대신 적당히 느낌으로 “이건 Edmond-Karp로 되겠구나”, “이건 Dinic을 써야 되겠구나”, 내지는 “이건 Flow로 푸는 게 아니구나” 등을 깨닫게 된다고 합니다….

5번 – 구두제작

- 입력 및 그래프 구축입니다.
- 최대 유량 알고리즘 등은 namespace를 따로 정의해서 빼놓는 식으로 구현하는 것이 편합니다.
- T는 100(최대 시간)입니다.
- 0번부터 T−1번 정점이 각 시간대, T번부터 (T+n−1)번 정점이 각 신발, T+n번이 source, T+n+1번이 sink입니다.

```
int n, k;
cin >> n >> k;
Flow::init(T + n + 2);

int psum = 0;
for(int i = 0; i < n; i++) {
  int x, y, z;
  cin >> x >> y >> z;
  psum += z;
  Flow::add(T + n, T + i, z);
  for(; x < y; x++) Flow::add(T + i, x, 1);
}

vint m(T, 0);
for(int i = 0; i < k; i++) {
  int x, y;
  cin >> x >> y;
  for(; x < y; x++) m[x]++;
}
for(int i = 0; i < T; i++) Flow::add(i, T + n + 1, m[i]);

cout << (Flow::get(T + n, T + n + 1) == psum) << '\n';
```

5번 – 구두제작

- 최대 유량 알고리즘의 초기화 및 간선 추가 부분입니다.
- 각 간선은 (도착 정점, 현재 남은 유량, 역간선 번호) 를 가집니다.
- 역간선을 만드는 이유는 이미 흘린 유량을 “취소하고” 다른 쪽으로 흘려주는 것이 더 좋을 때가 있기 때문입니다.

```
namespace Flow {
  struct Edge { int x, cap, rev; };
  vector<vector<Edge>> e(N);
  vint v(N);
  int n, snk;

  void init(int n_) {
    n = n_;
    for(int i = 0; i < n; i++) e[i].clear();
  }

  void add(int x, int y, int c) {
    e[x].push_back({y, c, e[y].size()});
    e[y].push_back({x, 0, int(e[x].size()) - 1});
  }
```

5번 – 구두제작

- 최대 유량 알고리즘의 핵심 구현 부분입니다.
- get 함수에서는 source와 sink가 주어지면 유량을 더 이상 흘릴 수 없을 때까지 반복적으로 유량을 흘려 줍니다.
- dfs 함수는 0 초과의 유량을 흘릴 수 있는 경로를 찾아서 거기로 유량을 흘려 줍니다.

```
int dfs(int x, int c) {
  v[x] = 1;
  if(x == snk) return c;
  for(Edge &i : e[x]) {
    if(v[i.x] || !i.cap) continue;
    int cur = dfs(i.x, min(i.cap, c));
    if(cur > 0) {
      i.cap -= cur;
      e[i.x][i.rev].cap += cur;
      return cur;
    }
  }
  return 0;
}

int get(int s, int t) {
  snk = t;
  int ans = 0;
  while(true) {
    fill(all(v), 0);
    int cur = dfs(s, int(1e9));
    if(!cur) break;
    ans += cur;
  }
  return ans;
}
```