**Advanced Coding Practice HW Problem #4**

**20191599 송경호**

**A) Train Rearrangement**

**A-1) 문제 인식 및 1번 풀이**

**스케치, 그림, 도표, 라인이(가) 표시된 사진

자동 생성된 설명**

위와 같은 T자 형태의 기찻길에 임의의 번호로 이루어지는 기차가 (a) → (b) → (c)로 진행한다. 이때 (c)에 작은 번호를 가진 기차부터 도착할 수 있는지 확인하는 문제이다. 예를 들어 (a)에서 3214 번호를 가진 기차가 출발할 때 #3 (a)to(b) → #2 (a)to(b) → #1 (a)to(b)to(c) → #2 (b)to(c) → #3 (b)to(c) → #4 (a)to(b)to(c)의 과정을 거쳐 (c)에 1234 번호를 가진 기차가 도착할 수 있다.

주목해야할 점은 (a)와 (b)는 LIFO 특성을 가진 stack 자료구조처럼 작동한다는 것이다. 즉 해당 문제는 임의의 값이 들어있는 (a) stack에서 (b) stack을 거쳐 (c)에 작은 번호의 수부터 삽입이 가능한지 여부를 확인하는 문제로 해석할 수 있다.

임의의 값 K가 (a) stack에 들어있을 때, (c)에 작은 번호의 수부터 삽입이 불가능한 경우는 쉽게 생각할 수 있다. 간단하게 가장 왼쪽 수를 stack의 top이라고 할 때, (a)에 231이 들어있는 경우를 생각한다. 1을 (c)에 삽입하기 위해 2와 3을 (b)에 삽입하면 다음과 같은 상황이 된다.

|  |  |
| --- | --- |
| empty | **(a)** |
| 3 2 | **(b)** |
| 1 | **(c)** |

그러나 기차는 단방향으로만 움직일 수 있기 때문에 위 상황에서 어떠한 방법으로도 2를 (c)에 3보다 먼저 삽입할 수 없다. 따라서 임의의 숫자 K에 대하여 항상 작은 수부터 (c)에 삽입하는 것은 불가능하다.

**A-2) 2번 풀이 및 알고리즘**

(a) stack에 들어있는 값은 최초에 정해지므로 값들이 머무를 곳은 (b) stack 밖에 없다. 즉, (b) stack의 순서는 (c)로도 이어진다. 따라서 문제의 조건을 만족시키기 위해선 절대로 (b) stack에 서 작은 값 위에 큰값이 쌓여서는 안된다. 이 점에 주목하여 알고리즘을 구상한다. (a)와 (b)는 각 각 stack으로 구현하여 top연산, pop연산, push연산이 가능하다.

1. (c)에 들어갈 숫자를 target으로 칭한다. (a)의 숫자들 중 가장 작은 수일 것이다. 또한 (a)의 pop값을 (b)에 push한다.

2. (b)의 top을 확인한다. 만약 top이 target이라면 (b)를 pop하고 다음 target을 업데이트 한다. 다시 2) 절차를 수행하며 이는 (b)의 top이 target이 아닐 때 까지 반복한다.

3. (a)의 pop값을 (b)의 top과 비교한다. 이때 (a)의 pop값이 (b)의 top보다 큰 경우 해당 기차는 (c)에 작은 순서로 삽입할 수 없다. 만약 (b)의 top이 더 큰 경우 (a)의 pop값을 (b)에 push한다. (b)가 empty일 경우, (a)의 값을 무조건 push한다.

4. (a)가 empty일 때까지 2. 3. 절차를 반복한다.

위 알고리즘을 통해 기차가 (c)에 올바르게 삽입 가능한지 확인할 수 있다.

**A-3) 시간복잡도, 공간복잡도 및 고려사항**

기차의 길이를 n이라고 할 때, (a) stack과 (b) stack을 합쳐서 전체 기차를 갖고 있어야 하므로 공간복잡도는 O(n)이 된다.

또한 기차가 문제의 조건에 성립하는 경우 모든 기차의 칸을 확인하게 되므로 시간복잡도 역시 O(n)이 된다.

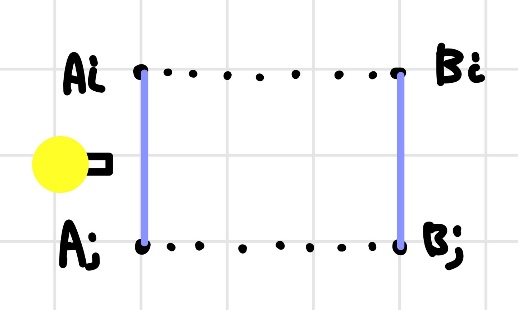
만약, (b)에서 다시 (a)로의 삽입이 가능하다면 더 복잡한 경우가 발생하겠지만 기차의 흐름이 단방향이기 때문에 직관적인 알고리즘이 설계되었다.

**B) Agony of Engineer**

**B-1) 문제 인식 및 아이디어 구상**

분리된 두 공간 A, B에 각 각 N개의 연결점이 있다. A의 점을 A1부터 An, B의 점을 B1부터 Bn이라고 했을 때, Ai와 Bi가 서로 연결되어 N개의 연결이 있는 형태이다. 이때 A1부터 An 그리고 B1부터 Bn이 임의의 위치에 있을 때 각 점을 특정하여 연결관계를 최소한의 A와 B의 이동을 통해 알아내는 것이 목표이다.

엔지니어는 전구와 스위치 그리고 전선을 이용하여 다음과 같이 점간의 연결을 확인할 수 있다.



위 그림을 보면 알 수 있듯이 2개 이상의 짝이 연결된 전체의 회로에 존재하면 전구에 불이 들어온다. 즉 위의 상태에서 각 Ai와 Aj를 구분할 수는 없지만 적어도 해당 쌍이 연결되어 있다는 점을 확인할 수 있다.

알고리즘 구상 시 편의를 위해 A를 차례대로 A1부터 An으로 정렬되어 있다고 가정한다. 단, Bi의 위치는 최초에 특정할 수 없다.

**B-2) 알고리즘**

*1) Ai와 Ai+1을 연결한다. 최초에 i=1이므로 A1과 A2를 연결한다.*

*2) 아직 연결 상태를 모르는 B를 모두 연결한다. 해당 상태에서 전구에 불을 확인하면 반드시 불이 켜지게 된다.*

*3) B의 점을 하나씩 제외해가며 불이 꺼지는 시점을 찾는다. 예를 들어 최초의 경우 B1부터 Bn까지가 연결되어 있을 것이다. 여기서 B1만 뺸 경우, B2만 뺀 경우 … 를 진행한다. 특정 점을 제외해서 불이 꺼졌을 때 해당 점은 Bi와 Bi+1 중 하나가 될 것이다.*

*4)*

*4-1) Ai의 점을 제외하고 Ai+2점을 추가한다. 만약 이때 불이 들어온다면 앞서 제외된 B의 점은 Ai와 연결된 것을 확인할 수 있다. 두 점의 연결을 표시하고 다시 2)로 넘어가 과정을 반복한다.*

*4-2) 만약 위 과정에서 불이 들어오지 않았다면 Ai+1과 앞서 제외된 B의 점이 연결된 것이다. Ai를 다시 추가하고 Ai+1을 제외한다. 두 점의 연결을 표시하고 다시 2)로 넘어간다.*

**B-3) 시간복잡도, 공간복잡도 및 고려사항**

시작하는 건물을 A라고 가정했을 때, A → B → A의 과정을 1번의 왕복이라고 한다면 총 n-1번의 왕복을 통해 모든 전선의 연결을 확인할 수 있다. 따라서 단순히 건물 사이의 왕복 횟수를 연산으로 시간복잡도를 계산한다면 O(n)의 시간복잡도를 가지게 된다.

그러나 회로를 구성하여 전구를 확인하는 과정을 한번의 연산이라고 한다면 3) 과정에서 전구에 불이 들어오는 지 최악의 경우 n번, n-1번, …, 1번의 연산을 수행해야 한다. 이 경우의 시간복잡도는 O(n2)이 된다.