CSE2011 Problem Solving, 2016 Spring

2013312343 이상헌

**Homework 6-1**

**1. 문제 이해**

(1) 문제

**Problem 6-1: Elevator Optimization 2 (50 points)**

We need to write an elevator optimization program.

The riders all enter their intended destination sat the beginning of the trip.

We limit the elevator to making at most k stops on any given run.

We assume that the penalty for walking up and down is same.

Management proposes to break ties among equal-cost solutions by given preference to stopping the elevator at the lowest floor.

Elevator does not necessary to stop at one of the floors the riders specified.

Ex) If riders specify floors 27 and 29, it can decided to stop at floor 28.

The aim is to select the floors to be stopped, so as to minimize the total number of floors people have to walk either up or down.

- Read the elevator optimization problem. We change the problem such that

1. People can walk down but cannot walk up; and

2. The 1stfloor is the base floor.

(2) 문제 정의

- 엘리베이터는 특정한 층에서만 멈춘다.

- 엘리베이터에 탈 수 있는 최대 정원, 엘리베이터가 멈추는 최대 층 수, 각 사람들의 최종 도착 층이 주어진다.

- 사람들은 멈춘 층에서 자신의 최종 도착 층까지 걸어가는데, 내려갈 수만 있으며 내려가는 층만큼의 cost가 주어진다.

- 모든 사람이 최종 도착 층까지 가는 데에 필요한 cost가 가장 적도록 하기 위해 엘리베이터가 멈춰야 하는 층을 출력하는 프로그램을 작성하라.

- Output : 각 사람마다 엘리베이터를 통해 내려야 하는 층.

**2. 문제 해결**

*Dynamic Programming*

- Dynamic Programming이란 어떠한 문제를 부분적으로 분할하여, 각 부분에 대해 최적의 답을 차례로 구하는 것으로써 문제 전체에 대한 최적의 해답을 얻는 알고리즘이다. Divide–and–conquer 기법과 다른 점은, dynamic programming은 Bottom-Up 기법이라는 점이다. 즉, 문제를 여러 부분 문제로 분할하고, 각 부분 문제의 해답을 구하여 이를 combine하는 divide-and-conquer 기법과는 다르게, dynamic programming은 전체 문제를 다양한 level의 subproblems로 나누고, 각 level의 해답을 통해 다음 level의 해답을 얻는 기법이다. Dynamic programming으로 문제를 해결할 때, 각 level의 해답을 얻을 때 사용한 중요한 정보들을 다음 level의 해답을 얻을 때 재사용되기 때문에, 반복적인 경험을 줄인다.

- 아이디어

먼저, 엘리베이터가 맨 꼭대기 층부터 아래로 내려간다고 역으로 가정하자. 이는 엘리베이터에 탑승한 사람이 내린 후, 최종 도착 층까지 올라갈 수 없고 내려갈 수만 있기 때문이다. 또한 엘리베이터가 멈추는 층의 수를 k라고 하고, 다음 멈추는 층을 선택하는 것을 각 단계라고 하자. 이와 같을 때, 어떠한 단계에서의 해답은(엘리베이터가 멈추는 층은) 이전 단계들의 모든 가능한 cost에 의존한다. 예를 들어, f1층에 멈춘 후 다음 멈추는 층을 f2라고 가정하면(단, f1>f2), f2층을 선택하는 것에서는 최종 도착 층이 f1층보다 위에 있는 사람들은 고려할 필요가 없다. 즉, 최종 도착 층이 f1층보다 아래에 있는 사람들만이 고려된 상태에서 f2층이 선택된다.

첫 번째로, m(i,j) 를 마지막 멈춘 층이 i층인 경우, 모든 사람들이 정확히 j단계만에 최종 도착 층에 도착하는 데에 드는 minimum cost라고 정의하자. 엘리베이터가 맨 꼭대기 층부터 아래로 내려온다고 가정하였기 때문에, (j+1)번째 멈추는 층은 j번째 멈추는 층보다 아래에 있다. 위에서 언급했듯이, 새로이 멈추는 층은 최종 도착 층이 바로 이전에 멈춘 층보다 아래에 있는 사람들만을 고려하여 선택된다. 따라서 모든 사람들을 각 멈추는 층 사이에 최종 도착 층이 있는 사람들로 나누어야 한다.

Figure . m(i,j)와 m(k,j-1).

i floor

k floor

walk(1,k)

walk(i,k)

walk(1,i)

j-1 stops

jth stops

특정 j단계의 해답을 찾기 위해 j단계와 1단계부터 j-1단계까지의 관계를 살펴보자. Figure 1과 같이 j-1단계에서 마지막으로 멈춘 층을 k층, j단계에서 멈출 층을 i층이라고 가정하자. 즉, m(i,j)와 m(k,j-1)이 존재한다고 가정하자. m(i,j)는 m(k,j-1)에서 새로이 i층이라는 추가로 멈추는 층이 있는 상태이다. 따라서 m(i,j)는 m(k,j-1)에서 최종 도착 층이 k층 밑에 있는 사람들에 대한 모든 cost를 빼고, 최종 도착 층이 i층과 k층 사이에 있는 사람들에 대한 cost, 최종 도착 층이 i층보다 밑에 있는 사람들에 대한 cost를 더한 값이 minimum인 i를 찾음으로써 만들어진다. 즉, 아래와 같은 식을 만족한다.

여기서 새로이 정의한 walk(a,b) 란, 최종 도착 층이 a층과 b층 사이에 존재하는 사람들이 a층 혹은 b층에서 내린 경우에 드는 cost를 의미한다.

실제로 엘리베이터는 맨 꼭대기 층에서부터 아래로 내려오지 않고, 1층부터 시작한다. 하지만 사람들은 내린 층에서 위로 올라가지 못하고, 꼭 아래로만 내려가야 하는, 원래 문제와 다른 점이 있기 때문에 위와 같은 가정을 하였다. 프로그램 구현 시, 이러한 가정은 검사의 순서가 맨 꼭대기 층에서부터 시작한다는 점으로 고려해야 한다. 하지만 결과는 멈춰야 하는 각 층 만을 출력하고, 또한 엘리베이터가 움직이는 데에 따른 cost 등이 없기 때문에 이와 같은 가정은 프로그램 구현 시에 크게 중요하지 않다.

- 이점

Dynamic programming의 가장 큰 이점은 각 단계에서 필요한 중요한 정보가 다음 단계에서도 쓰이고, 이러한 하나의 중요한 정보를 토대로 가장 정확한 결과를 도출할 수 있다는 것이다. 본 아이디어에서는 특정 단계와 그 단계 이전의 모든 단계들 내에서 도출된 cost들 사이의 식 하나만으로 모든 효율적인 경우를 검사하고, 정확한 결과를 도출해 낼 수 있다. 즉, 하나의 정보를 토대로 복잡하지 않은 단계들을 통해 input이 어떠한 경우에든 일반화하여 결과를 도출할 수 있다는 것이다.

- 한계

본 문제는 책에 이미 존재한 문제에서 가정을 약간 변형한 문제이다. 변형한 점은, 사람들이 멈춘 층에서부터 최종 도착 층까지 가는 방법에서, 올라갈 수 없고 오로지 내려갈 수만 있다는 점이다. 이와 같은 변형된 문제를 푸는 과정에서 본 아이디어에서는 엘리베이터의 방향을 맨 꼭대기 층에서부터 아래로 내려간다고 가정하였다. 만일, 문제의 변형 중에 엘리베이터의 움직임에 대한 cost가 작용하거나 다른 요소에 의해 이와 같은 가정을 했을 때 모순점이 생기는 경우, 본 아이디어의 한계점이 드러나고, 정확한 결과를 출력할 수 없을 수도 있다.

이외에 본 문제에 대한 해결 방안으로 한 알고리즘으로는 큰 한계점을 느낄 수 없었다.

**3. 코드 작성**

- 본 아이디어 및 설명을 바탕으로 코드를 작성하였다.

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#define NFLOORS 100

#define MAX\_RIDERS 50

#define MAX\_STOPS 50

int stops[MAX\_RIDERS];

int nriders;

int nstops;

int m[NFLOORS + 1][MAX\_RIDERS];

int p[NFLOORS + 1][MAX\_RIDERS];

int result[MAX\_STOPS];

int res[MAX\_RIDERS];

- “limits.h” 라는 헤더 파일은 integer의 최대값(INT\_MAX)을 사용하기 위함이다. 건물의 최대 층수(100층)와 입력 받을 수 있는 최대 인원수(50) 및 엘리베이터의 멈추는 횟수(50)를 상수로 정의하였다. 각 사람의 최종 도착 층수를 저장하는 integer 배열인 stops, 엘리베이터 탑승 인원을 저장하는 nriders, 엘리베이터가 멈추는 횟수를 저장하는 nstops 및 dynamic programming에서 사용되는, cost를 저장하는 2차원 배열 m과 이전 단계 층을 저장하는 배열 p, 결과를 저장하는 1차원 배열 result와 res는 모든 함수에서 쓰이기 때문에 전역 변수로 설정하였다.

int floors\_walked(int previous, int current) {

int nsteps = 0;

int i;

for (i = 0; i < nriders; i++) {

if ((stops[i] <= current) && (stops[i] > previous))

nsteps += (current - stops[i]);

}

return (nsteps);

}

- floors\_walked 함수는 위의 알고리즘 설명에서 정의한 walk(a,b)를 의미한다. previous와 current라는 각 매개변수는 아래와 위 층을 의미한다. 만일 어떤 사람의 최종 도착 층이 이 둘 층 사이에 존재한다면, cost는 위 층인 current에서 내려오는 층 수만큼 더해진다. 이 함수는 모든 사람들의 최종 도착 층을 검사하여, 계산한 cost를 return 한다.

void reconstruct\_path(int lastfloor, int stops\_to\_go) {

if (stops\_to\_go > 1)

reconstruct\_path(p[lastfloor][stops\_to\_go], stops\_to\_go - 1);

result[stops\_to\_go] = lastfloor;

}

- reconstruct\_path 함수는 재귀적 호출을 이용하여 엘리베이터가 멈추는 각 층을 result라는 배열에 저장하는 함수이다.

void printresult() {

int i, j;

for (i = 0; i < nriders; i++) {

for (j = 1; j <= nstops; j++) {

if (stops[i] <= result[j])

res[i] = result[j];

}

}

for (i = 0; i < nriders; i++)

printf("%d\n", res[i]);

}

- printresult 함수는 엘리베이터가 멈추는 층수를 저장한 result라는 배열과 stops 배열 내에 저장된 정보를 이용하여, 문제의 output 의도에 맞도록 출력하는 함수이다.

int main(void) {

int cost, laststop;

int i, j, k;

scanf("%d", &nriders);

scanf("%d", &nstops);

for (i = 0; i < nriders; i++) {

scanf("%d", &stops[i]);

}

for (i = 0; i <= NFLOORS; i++) {

m[i][0] = floors\_walked(0, INT\_MAX);

p[i][0] = 1;

}

- main 함수에서는 본격적으로 dynamic programming을 구현한다. cost 및 laststop은 각각 최소의 cost 및 가장 마지막으로 멈추는 층을 저장한다. nriders와 nstops 및 각 사람의 최종 도착 층수를 입력 받는다. cost를 저장하는 m 배열의 j=0 단계에서의 값을 walk(0,INT\_MAX)로 정의하고, p 배열의 j=0 단계에서의 값을 1로 정의한다.

for (j = 1; j <= nstops; j++) {

for (i = NFLOORS; i >= 1; i--) {

m[i][j] = INT\_MAX;

for (k = NFLOORS; k >= i; k--) {

cost = m[k][j - 1] - floors\_walked(1, k) + floors\_walked(i, k) + floors\_walked(1, i);

if (cost < m[i][j]) {

m[i][j] = cost;

p[i][j] = k;

}

}

}

}

- dynamic programming을 구현하는 코드이다. 각 단계에서(j=1 … nstops) 이전 단계들의 정보를 통해 새로운 cost를 알고리즘 설명에서 표현한 식을 통해 계산한다. 만일 새로운 cost가 앞서 계산한 j단계에서의 cost보다 작다면 이를 m에 저장하고, p에 k층을 저장한다.

laststop = 1;

for (i = 1; i <= NFLOORS; i++) {

if (m[i][nstops] < m[laststop][nstops])

laststop = i;

}

reconstruct\_path(laststop, nstops);

printresult();

}

- 마지막으로 엘리베이터가 멈추는 단계를 계산하여 laststop 변수에 저장한다. 이후 reconstruct\_path 및 printresult 함수를 이용하여 결과를 계산하고, 출력한다.

**4. 확인 및 분석**

- 본 아이디어를 이용하여 프로그램을 작성하였다. 이 프로그램이 정확한 결과값을 도출하는지를 알아보기 위해서 간단한 예시를 입력하고, 결과를 비교하였다.

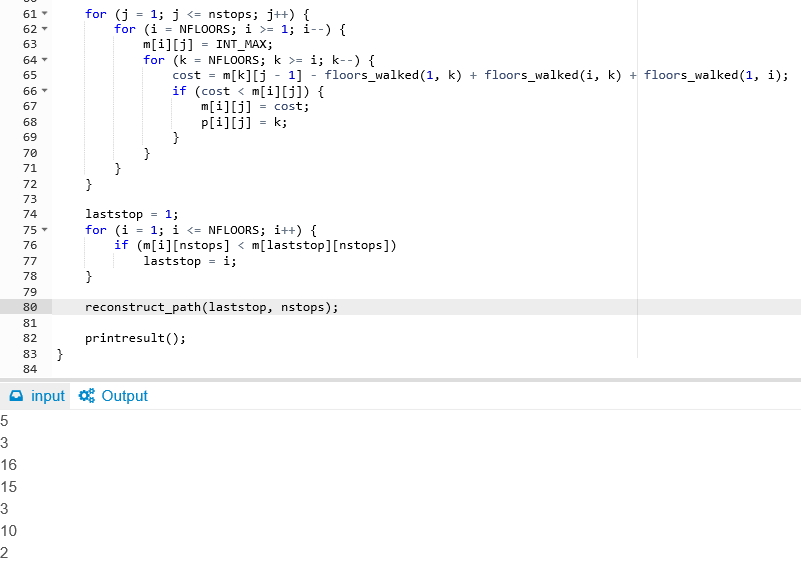


Figure . 예시 1 Input

Figure 2는 예시 1의 input 정보이다. 이는 문제 설명 ppt에 나와있는 예시 input의 순서만 바꾼 형태이다. 위와 같이 5명이 탑승하고, 3번의 멈춤을 요구하는 엘리베이터의 경우, 각 멈추는 층은 16층, 10층, 그리고 3층이다. 프로그램은 Figure 3과 같이 예상과 맞는 정확한 결과를 도출했다는 것을 알 수 있다.

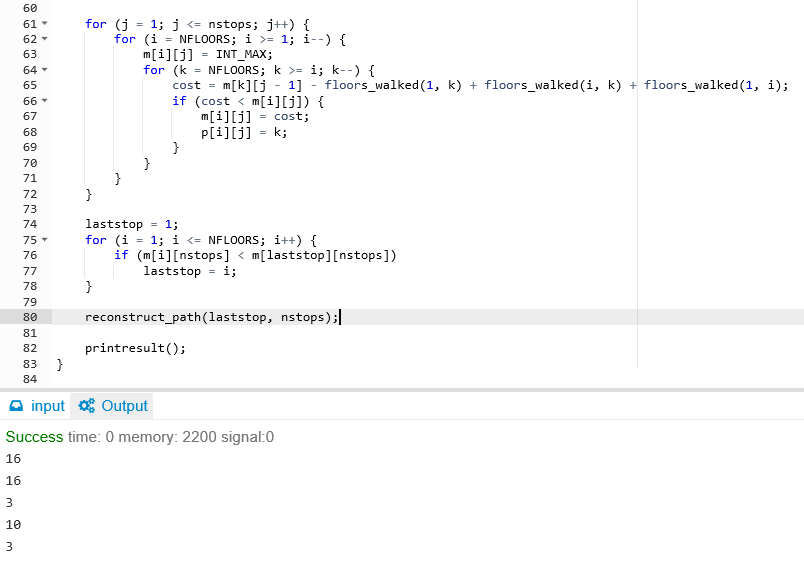


Figure . 예시 1 Output



Figure . 예시 2 Input

Figure 4는 예시 2의 input 정보이다. 위와 같이 7명이 탑승하고, 3번의 멈춤을 요구하는 엘리베이터의 경우, 각 멈추는 층은 10층, 12층, 15층 혹은 10층, 13층, 15층으로 2가지 경우에서 최소의 cost를 나타낸다. 프로그램은 Figure 5와 같이 예상과 맞는 정확한 결과를 도출했다는 것을 알 수 있다.



Figure . 예시 2 Output

**5. 고찰**

본 문제는 입력된 인원 및 멈추는 횟수가 정해져 있는 엘리베이터의 가장 효율적인 운행 방식을 결정, 출력하는 프로그램을 구현하는 것이다. 본 문제를 dynamic programming 기법을 이용하여 해결하였다. 기존에 책에 있는 문제와 다른 점은, 내린 사람이 최종 도착 층까지 이동하는 데에 아래로만 내려갈 수 있다는 제약이 존재한다는 점이다. 이와 같은 사실을 반영하기 위하여, 엘리베이터의 진행 방향을 맨 꼭대기에서 바닥 방향으로 가정하였다. 이 가정은 실제 프로그램 구현에서는 큰 문제가 되지는 않았다. 또한 시작 층수가 1층이라는 것은, 각 단계에서의 cost 계산 식에서 0층이 아닌 1층으로 기준잡음으로써 해결하였다. Dynamic Programming 기법을 통해 하나의 식(정보)만으로 각 단계의 효율적인 해답을 찾고, 이 해답을 통해 원하는 단계까지의 해답을 얻을 수 있었다.

**6. 참고문헌 및 사용**

- 이진규 교수님, [문제해결기법] CSE2011\_2016spring\_Lecture\_Note08, 2016

- <http://terms.naver.com/entry.nhn?docId=2691142&cid=42411&categoryId=42411>

- <http://ehclub.tistory.com/1555>