CSE2011 Problem Solving, 2016 Spring

2013312343 이상헌

**Homework 4-2**

**1. 문제 이해**

(1) 문제

**Problem 4-2: Shoemaker’s problem (page 94)**

- A shoemaker has N orders from customers which he must satisfy. The shoemaker can work on only one job in each day, and jobs usually take several days. For the ith job, the integer Ti (1 ≤ Ti ≤ 1, 000) denotes the number of days it takes the shoemaker to finish the job.

- But popularity has its price. For each day of delay before starting to work on the ith job, the shoemaker has agreed to pay a fine of Si (1 ≤ Si ≤ 10, 000) cents per day. Help the shoemaker by writing a program to find the sequence of jobs with minimum total fine.

**Input**

- The input begins with a single positive integer on a line by itself indicating the number of the test cases, followed by a blank line. There is also a blank line between two consecutive cases.

- The first line of each case contains an integer reporting the number of jobs N, where 1 ≤ N ≤ 1000. The ith subsequent line contains the completion time Ti and daily penalty Si for the ith job.

**Output**

- For each test case, your program should print the sequence of jobs with minimal fine. Each job should be represented by its position in the input. All integers should be placed on only one output line and each pair separated by one space. If multiple solutions are possible, print the first one in lexicographic order.

- The output of two consecutive cases must be separated by a blank line.

(2) 중요 정보

- 반복 수, 일의 개수, 각 일이 완료하는 데 걸리는 날 및 1일 당 드는 비용을 입력 받는다.

- 1일 당 드는 비용은 특정 일이 시작되기 전까지 걸리는 날에 적용된다.

- 일은 1개 이상 1000개 이하이고, 비용은 1 이상 10000 이하이다.

(3) 문제 정의

- input 정보를 통해 최소의 비용이 발생하는 일의 순서를 정하는 알고리즘을 만들고, 결과를 출력하는 프로그램을 구현하라.

**2. 문제 해결**

*BackTracking.*

- Backtracking 알고리즘이란 모든 경우의 수를 효과적으로 고려하는 방법이다. 하나부터 시작하여 각 단계에서 포함될 수 있는 후보들을 선출하고, 각 후보에 대해 같은 방법을 재귀적으로 반복한다. 일련의 역할이 정해져 있는 함수들이 포함된 코드 툴을 통해 backtracking을 효과적으로 구현할 수 있다.

- 아이디어

각 일에 대해 가능한 모든 우선순위의 조합과 각 조합에서 발생하는 비용을 비교, 최적을 조사한다. 시행 과정을 일의 개수만큼의 단계로 나누고, 각 단계에서 결과에 포함되는 일의 우선 순위를 1, 2, 3, … 등의 순서로 정의한다. (1의 순위가 높은 순위로, 가장 먼저 시행된다.) 또한 각 단계에서 결과에 포함될 수 있는 일의 후보를 정한다. 후보는 각 단계에서 결과에 입력되기 전에 정의된다.

예를 들어 첫 번째 단계에서는 모든 일이 후보로 등록될 수 있다. 이들 중에 결과에 포함되는 일은 단 한 개로, 포함되는 일의 우선순위는 1이 된다. 두 번째 단계에서는 첫 번째 단계에서 결과에 포함된 일을 제외한 나머지 일들이 후보로 등록될 수 있고, 첫 번째 단계와 같은 방법으로 이들 중에 단 한 개의 일이 결과에 포함되고, 우선순위는 2가 된다. 이와 같이 각 단계에서 결과에 하나의 일만이 포함되어, 모든 일들이 포함되는 경우에는 backtrack을 종료하고, 발생하는 비용을 계산한다.

모든 일이 포함되었다고 해서 본 알고리즘이 종료되는 것이 아니다. 한 단계 전으로 돌아가서 그 단계에서의 후보들 중 다른 후보의 일이 결과에 먼저 포함되는 경우도 생각한다. 이와 같이 재귀 적인 방법을 통해 가능한 모든 우선순위 조합의 경우를 확인하고, 비용을 계산한다.

예를 들어, A, B, C, D의 4개의 일이 있다고 가정하자. 각 단계에서는 Table 1과 같이 후보를 먼저 등록하고, 그 후보들 중에 단 하나의 일이 결과에 포함된다. 이와 같이 일의 수만큼의 단계를 반복하면, 하나의 우선순위 조합이 나타난다. 이후에 전단계로 돌아가서 다른 후보를 결과에 포함시킨다. Table 1에는 두 가지 겨로가 조합이 나타났다. 즉, A,B,C,D의 우선순위 순서가 (1, 2, 4, 3), (3, 1, 4, 2)가 될 수 있다. 각 조합에 따라 발생하는 비용을 계산한다.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 우선순위 (단계) | 1 | | 2 | | 3 | | 4 | |
| 결과 | A | B | B | D | D | A | C | C |
| 후보 | A,B,C,D | A,B,C,D | B, C, D | A, C, D | C, D | A, C | C | C |

Table 1. 각 단계에서 후보 및 결과에 포함되는 일.

각 일의 우선순위가 결정된 경우에 대해 발생하는 비용을 계산하는 방법을 알아보자. Input 값은 각 일에 대해 일이 완료될 때까지 걸리는 날 및 1일 당 발생하는 비용이다. N 개의 일에 대해 각 일의 완료 날짜를 a[n], 1일당 비용을 f[n]이라 하자. 또한 임의로, n을 그 일의 우선순위라고 정하자. 총 발생하는 비용은 다음 수식과 같다.

총 발생 비용 = f[1] \* 0 + f[2] \* (0 + a[1]) + f[3] \* (0 + a[1] + a[2]) + … + f[n] \* (0 + a[1] + a[2] + … + a[n-1])

즉, 총 발생 비용 = 이다. 단, a[0] = 0으로 가정하자.

- 이점

Backtracking 기법의 가장 큰 장점은 정확성이다. Backtracking은 가능한 모든 경우의 수를 고려하고, 그 중에서 문제의 의도에 맞는 경우를 저장하기 때문에, 모든 알고리즘 중에서 가장 정확한 결과를 나타낸다. 또한 이미 존재하는 여러 함수들이 포함된 코드 툴을 이용하여 간단하고, 체계적인 코딩 및 구현을 가능하게 한다.

**3. 코드 작성**

- 본 아이디어 및 설명을 바탕으로 코드를 작성하였다.

int is\_a\_solution(int a[], int k, int day[], int fine[], int num\_task) {

return (k == num\_task);

}

- is\_a\_solution 함수는 backtracking 과정 중에 모든 단계가 완료된, 즉 모든 일의 우선순위가 정해진 상태인지를 check하는 함수이다. K는 backtracking 과정에서의 단계를 의미하는데, 만일 마지막 단계라면 더 이상 포함될 수 있는 일이 없다는 뜻으로, backtracking 과정을 완료할 수 있다.

void process\_solution(int a[], int day[], int fine[], int num\_task, int order[], int \*min) {

int i = 0;

for (i = 0; i < num\_task; i++) {

order[i] = a[i];

}

}

- process\_solution 함수는 결론적으로 문제에서 원하는 답을 저장 혹은 출력 등으로 결과를 저장하는 함수이다. 가장 적은 비용이 발생하는 우선순위 조합을 order에 저장하여, main 함수에서 출력한다.

void construct\_candidates(int a[], int k, int day[], int fine[], int num\_task, int c[], int \*ncandidates) {

int i, j = 0, l, check;

if (k == 1) {

for (i = 0; i < num\_task; i++)

c[i] = i;

}

else {

for (i = 0; i < num\_task; i++) {

check = 1;

for (l = 0; l < k - 1; l++) {

if (a[l] == i)

check = 0;

}

if (check) {

c[j] = i;

j++;

}

}

}

\*ncandidates = num\_task - k + 1;

}

- construct\_candidates 함수는 각 k 단계에서 후보를 선정하는 함수이다. 후보는 각 단계에서 결과에 포함될 수 있는 일들을 의미한다. K가 1인 경우, 즉 첫 번째 단계에서의 후보는 모든 일이 선정된다. 그 다음 단계부터는, 이미 결과에 포함되어 있는 일은 후보에 포함될 수 없기 때문에 제외시킨다. 후보의 개수는 항상 일의 개수 – 단계 수 + 1 이다.

void backtrack(int a[], int k, int num\_task, int day[], int fine[], int order[], int \*min) {

int \*c = calloc(num\_task, sizeof(int));

int ncandidates, i;

if (is\_a\_solution(a, k, day, fine, num\_task)) {

if (calculate\_fine(day, fine, num\_task, a) < \*min) {

\*min = calculate\_fine(day, fine, num\_task, a);

process\_solution(a, day, fine, num\_task, order, min);

}

}

else {

k = k + 1;

construct\_candidates(a, k, day, fine, num\_task, c, &ncandidates);

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

a[k - 1] = c[i];

backtrack(a, k, num\_task, day, fine, order, min);

}

}

}

- backtrack 함수는 backtracking을 실제로 진행하는 함수이다. 먼저, 후보를 저장하는 c라는 int형 배열과 후보의 개수를 저장하는 ncandidates를 선언한다.

- 단계를 시작하기에 앞서 만일 단계가 마지막 단계, 즉 모든 일이 결과에 포함되어 backtrack 과정이 완료된 경우에는, is\_a\_solution 함수를 통해 추출한다. 이러한 경우에는 저장된 우선순위 조합에 따라 발생하는 비용을 계산하고, 기존에 있던 가장 작은 비용을 저장하는 min 보다 작은 경우에는 process\_solution 함수를 통해 우선순위 조합 결과를 저장한다.

- backtrack이 완료되지 않은 경우에는 단계를 진행한다. 먼저, 본 단계에서 결과에 포함될 수 있는 후보를 선정하고 (construct\_candidates 함수), 후보 중에 하나의 일을 결과에 포함시킨 뒤에 다음 단계로 진행한다. 다음 단계 진행은 backtrack 함수를 재귀 호출하는 것으로 한다. 결국 backtrack 과정이 모두 끝난다면, 다시 이 단계로 돌아와서 다른 후보의 일을 결과에 포함시킨 뒤에 같은 backtrack 과정을 진행한다.

int calculate\_fine(int day[], int fine[], int num\_task, int order[]) {

int tot\_fine = 0, i, j, k;

for (i = 0; i < num\_task; i++) {

for (j = 0; j < num\_task; j++) {

if (order[j] == i)

k = j;

}

for (j = 0; j < num\_task; j++) {

if (j != k && order[j] > order[k])

tot\_fine += day[k] \* fine[j];

}

}

return tot\_fine;

}

- calculate\_fine 함수는 특정 우선순위를 토대로 일을 진행한 경우에 발생하는 총 비용을 계산하는 함수이다. 앞서 설명한 수식을 토대로 발생하는 비용을 tot\_fine 변수에 저장하고, 이를 리턴 한다.

**4. 확인 및 분석**

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

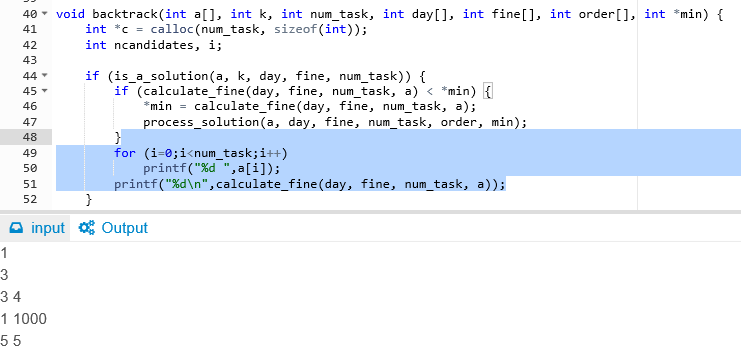


Figure 1. 예시 1 Input.

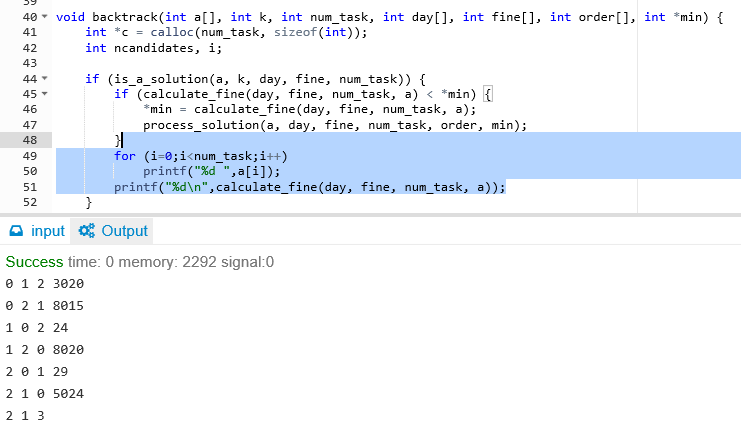


Figure 2. 예시 1 Output.

- 첫 번째 예시는 Figure 1과 같이 1번의 반복, 3개의 일과 각각의 일에 대해 (완료 날짜, 하루당 비용)을 (3, 4), (1, 1000), (5,5) 라는 input을 입력하였다. 이에 대한 결과는 Figure 2와 같이 나타난다.

- 여기서, 각 backtrack 단계의 완료 상태에서의 우선순위 조합을 보기 위해서, 각 backtrack 과정의 완료일 때마다 일 순서에 따른 우선순위 및 발생하는 비용을 출력하도록 하였다. (Figure 2에서 드래그한 부분이 예시 분석을 위해 추가된 코드이다.) 우선순위가 0부터 시작하는 것은 main 함수에서 1씩 더하며 출력하기 때문에 문제되지 않는다.

- Figure2를 보면, 모든 경우의 수를 탐색했고 각각의 경우의 수에 따라 발생하는 비용 또한 정확하다는 것을 검산을 통해 알 수 있다. 또한 가장 적은 비용이 발생하는 우선순위 조합을 저장하여, 정확한 결과를 도출했다는 것을 알 수 있다.

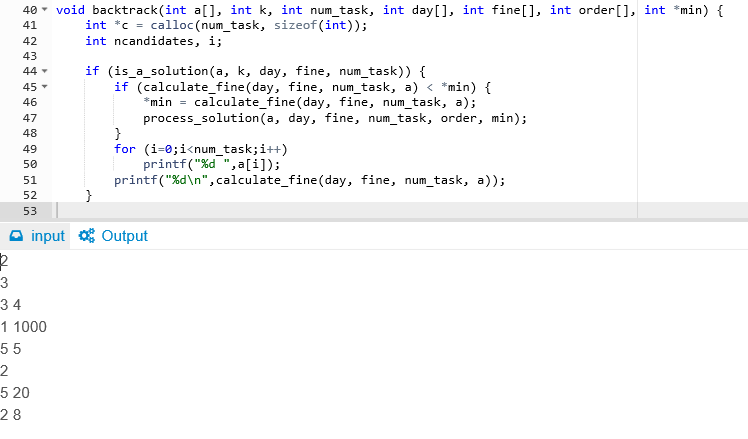


Figure 3. 예시 2 Input.

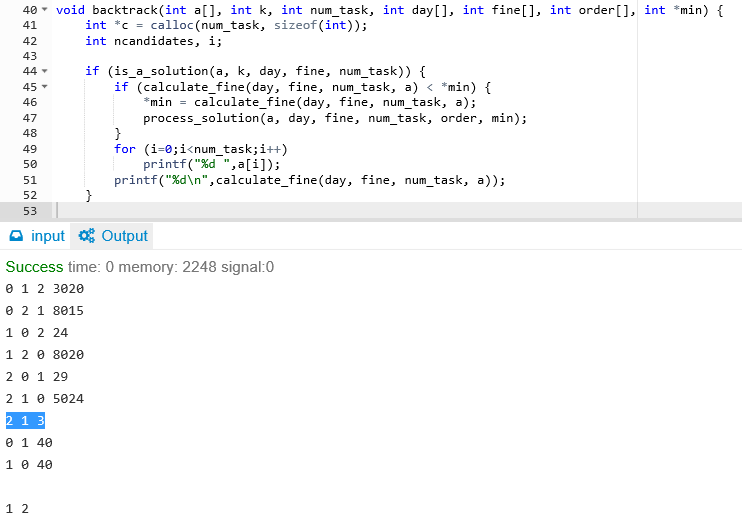


Figure 4. 예시 2 Output.

- 두 번째 예시는 Figure 3과 같이 2번의 반복을 입력하였다. 첫 번째 반복에서는 3개의 일과 각각의 일에 대해 (완료 날짜, 하루당 비용)을 (3, 4), (1, 1000), (5,5) 라는 input을 입력하였고, 두 번째 반복에서는 2개의 일과 각각의 일에 대해 (5, 20), (2, 8) 라는 input을 입력하였다. 이에 대한 결과는 Figure 4와 같이 나타난다.

- Figure4를 보면, 반복이 있는 input에서도 각각의 반복에 대해 모든 경우의 수를 탐색하여 가장 적은 비용이 발생하는 결과를 정확히 출력했다는 것을 알 수 있다. 결과와 결과 사이에는 한 줄의 공백이 생성되는 것도 문제의 의도와 맞다.

**5. 고찰**

- 본 문제는 완료될 때까지 걸리는 일수 및 일이 시작되기 전에 하루당 발생하는 비용이 다른, 여러 개의 일들에 대해 가장 작은 비용이 발생하도록 하는 일의 우선순위를 정하는 프로그램을 구현하는 것이다. Backtracking 기법을 이용하여 본 문제를 해결하는 프로그램을 성공적으로 구현하였다. Backtracking 기법은 모든 경우의 수를 판단하고, 비교하기 때문에 가장 정확한 결과를 도출하는 알고리즘 중 하나이다. 하지만 모든 경우의 수를 고려해야 한다는 점에서, 높지 않은 효율성을 나타낸다.

- Backtracking 과정에서 각 단계에 조건문 등을 통해 문제의 solution이 될 수 없는 경우를 미리 삭제하여, 그 이후의 단계를 고려하지 않아도 되도록 수정한다면 효율적인 문제가 조금이나마 개선될 것이다. 예를 들어, 본 문제에서는 완료되지 않은 각 단계에서도, 지금까지의 우선순위에 따른 비용을 계산하고, min과 비교하는 부분을 생성한다. 만일 단계가 모두 완료되지 않았지만 min보다 큰 비용이 발생한다면, 이후의 단계에서의 경우의 수는 고려할 필요가 없을 것이다.

**6. 참고문헌 및 사용**

- [www.ide.com](http://www.ide.com) : C언어 구현 및 실행.

- 이진규 교수님, [문제해결기법] CSE2011\_2016spring\_Lecture\_Note06, 2016