

## 알고리즘-12주차-Dynamic Programming

이름 : 유형곤

학번 : 201902722

사용 언어 : Java

### <점수표>

RANK	TEAM	SCORE	1-SUBSUM  [1 POINT]	2-KNAPSACK  [3 POINTS]	3-CHANGE  [3 POINTS]	4-MRDRILLER  [3 POINTS]
2	201902722	10 79	4 2 tries	7 1 try	16 1 try	32 1 try

문제 1 : SubSum

문제 : 배열의 최대 부분 합을 구하는 문제

해결 방법 : DP를 활용해 부분 합 알고리즘의 시간복잡도를 낮추면 되는데, 여기서의 겹치는 부분문제는 - 이전까지의 최대 부분 합을 알 경우, 그 다음 인덱스까지의 최대 부분 합을  $O(1)$ 만에 알아낼 수 있으므로 이전까지의 최대 부분 합을 구하는 것이 겹친다고 볼 수 있다. 좀 더 단순하게 말하자면  $arr[k+1]$ 까지의 최대 부분 합을 구하는 문제는  $arr[k]$ 의 최대 합을 구하는 문제를 포함한다.

시간복잡도 :  $O(N)$

```
d[0] = arr[0];
int answer = d[0];
for(int i = 1; i < n; i++) {
    d[i] = Math.max(d[i-1] + arr[i], arr[i]);
    answer = Math.max(answer, d[i]);
}
```

### <SubSum 소스코드>



```
<terminated> MaxSum [Java Application] C:\Program Files\Java\jre1.8.0_211\
10
10 -4 3 1 5 6 -35 12 21 -1
33
|
```

### <SubSum 실행결과>

문제 2 : Knapsack

문제 : 0-1 Knapsack 알고리즘으로 배낭에 채워서 얻을 수 있는 최대 가치를 구하는 문제  
해결방법 : 지금까지 저장된 무게가  $w$ 일 때, 현재 물건을 포함시키거나 아닌 경우 가치가 더 큰 경우를 선택 의사코드로 표현하면 :

i: from 0 to  $n-1$  //  $i$ 번째 물건

w : from 0 to  $W$  //지금까지 가방에 담긴 무게가  $0 \sim W$

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w-w_i] + v_i)$$

시간복잡도 :  $O(nw)$

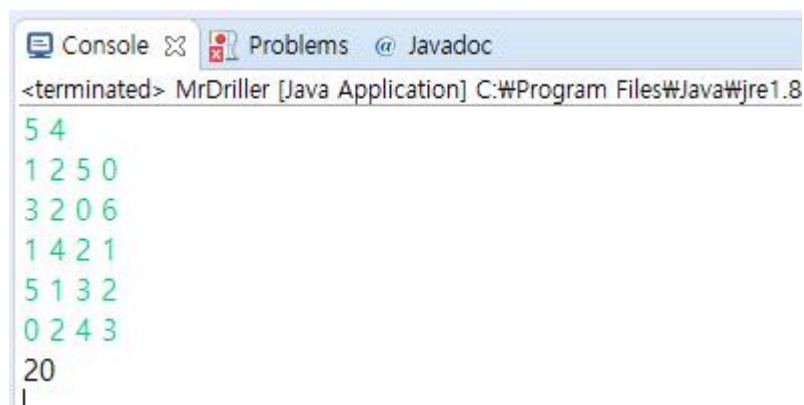
의사코드에 모두 나와있으므로 설명 생략.

```
static int n, k;
static int weight[], value[], d[][];
public static int pack(int capacity, int item) {
    if(item == n+1) {
        return 0;
    }

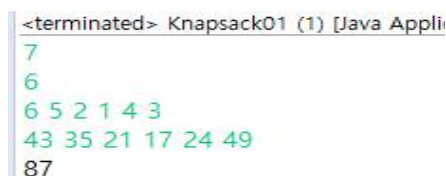
    int ret = d[capacity][item];
    if(ret > 0) {
        return ret;
    }

    ret = pack(capacity, item + 1); //not picking this item
    if(capacity >= weight[item]) {
        ret = Math.max(ret, pack(capacity - weight[item], item + 1) + value[item]);
    }
    d[capacity][item] = ret;
    return ret;
}
```

<Knapsack 소스코드>



```
<terminated> MrDriller [Java Application] C:\Program Files\Java\jre1.8
5 4
1 2 5 0
3 2 0 6
1 4 2 1
5 1 3 2
0 2 4 3
20
|
```



```
<terminated> Knapsack01 (1) [Java Appli
7
6
6 5 2 1 4 3
43 35 21 17 24 49
87
```

<Knapsack 실행결과>

### 문제 3 : Change

문제 : 동전  $n$ 개로 돈을 거슬러줄 때 가장 작은 동전의 개수

해결 방법 :  $d[1] \sim d[k-1]$ 까지의 정답을 구했을 때,  $d[k]$ 는

for coin in coins:

$d[k] = \min(d[k], d[k - \text{coin}] + 1)$

으로 정의할 수 있습니다. (즉  $d$ [현재 돈에서 각각의 코인을 뺀 경우 거스름돈의 개수]의 최솟값을 구하면 됩니다.) Improved Recursion와 다른 점은, 이전에는 재귀를 사용해 Top-Down 방식으로 해결해 “남은 돈”이 0이 될 때까지 재귀를 수행한다면, Bottom-Up 방식에서는 1~ $n$ 원에 대한 정답을 미리 구해놓습니다.

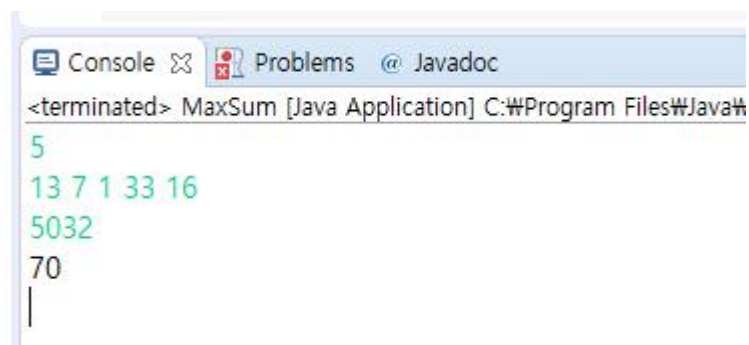
시간복잡도 :  $O(cn)$  ( $n$  : 구하려는 금액,  $c$  : 동전의 개수)

```
for(int i = 1; i <= money; i++) {
    d[i] = Integer.MAX_VALUE;
}

for(int i = 0; i < coin.length; i++) {
    d[coin[i]] = 1;
}

for(int i = 1; i <= money; i++) {
    for(int j = 0; j < coin.length; j++) {
        if(i >= coin[j]) {
            d[i] = Math.min(d[i], d[i-coin[j]] + 1);
        }
    }
}
```

<Change 소스코드>



```
Console Problems @ Javadoc
<terminated> MaxSum [Java Application] C:\Program Files\Java\
5
13 7 1 33 16
5032
70
|
```

<Change 실행결과>

문제 4 : Mr.Driller

문제 : 주어진 맵에서 주어진 조건으로 맵을 탐색할 때의 최대 이익을 구하는 문제  
(RGB 거리와 매우 유사) 주어진 맵은 m by n 행렬

해결방법 : k번째 줄의 i번째 블록에서의 최대 이익은 다음과 같이 정의할 수 있다.

$d[k][i] = \max(d[k-1][i-1], d[k-1][i], d[k-1][i+1]) + \text{money}[i][k]$

단,  $(i-1) \geq 0 \ \&\& \ (i+1) < n$

시간복잡도 :  $O(mn)$

각각의 행에 대하여  $O(m)$  ,  $d[k][i]$ 를 구하려면  $O(3n) = O(n)$ 번의 연산이 필요하다.

따라서 시간복잡도는  $O(mn)$

```
for(int i = 0; i < n; i++) {
    d[0][i] = map[0][i];
}

int dx[] = {1, -1, 0};
for(int i = 1; i < m; i++) {
    for(int j = 0; j < n; j++) {
        for(int k = 0; k < 3; k++) {
            int x = dx[k] + j;
            if(0 <= x && x < n) {
                d[i][j] = Math.max(d[i][j], d[i-1][x]);
            }
        }
        d[i][j] += map[i][j];
    }
}

int answer = 0;
for(int i = 0; i < n; i++) {
    answer = Math.max(answer, d[m-1][i]);
}
```

<Mr.Driller 소스코드>



```
<terminated> MrDriller [Java Application] C:\Program Files\Java\
4 4
1 2 5 0
3 2 0 6
1 4 2 1
5 1 3 2
16
```

<Mr.Driller 실행결과>