9. Dynamic Programming #1

2017010698
수학과 오서영

동적계획법 (Dynamic Programming)

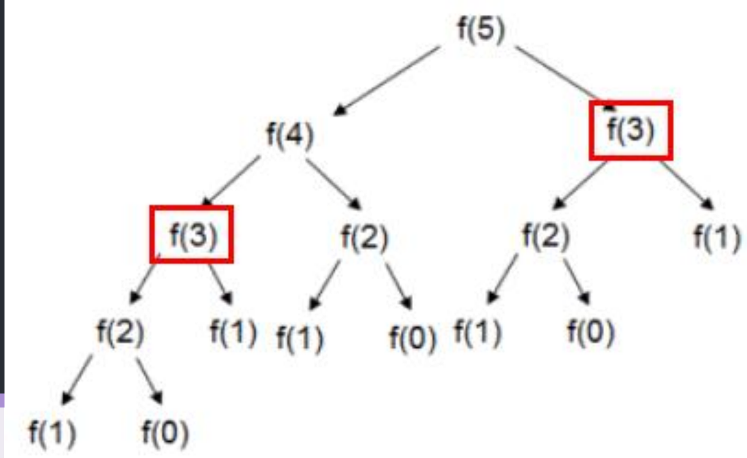
동적계획법 (Dynamic Programming)
: 전체 문제를 작은 문제로 단순화한 다음
점화식으로 만들어 재귀적인 구조를 활용해서
전체 문제를 해결하는 방식

-> 큰문제를 작은문제를 나눠서 푸는 기법

메모이제이션(Memoization)
: 이전에 계산해둔 값을 메모리 (배열 등) 에 저장해서
반복 작업을 줄이는 기법

메모이제이션(Memoization)

```python
def fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```



재귀함수를 이용하여 피보나치 수열 계산하기

: fib(5) 를 호출 -> 왼쪽에서 이미 f(3) 을 한번 호출했는데, 오른쪽에서도 또 호출하고 있다. DP의 아이디어는 계산하는 값들을 어디다 저장해뒀다가, 저런 식으로 중복되는 계산이 나오면 저장해뒀던 값을 쓰는것

-> 오른쪽 트리의 f(3) 아래쪽 부분은 계산할 필요가 없게 된다.

0-1 Knapsack

0-1 Knapsack

: 도둑이 보석가게에 배낭을 메고 침입했다.
배낭의 최대 용량은 W이며,
각 보석들의 무게와 가격은 알고 있다.
배낭이 찢어지지 않는 선에서 가격 합이 최대가 되도록 보석을 담는 방법은?

0-1 Knapsack

1) Brute-Force : 모든 경우의 수를 넣어본다

n개의 보석이 있다고 치면, n개 보석으로
만들 수 있는 가능한 부분집합의 수는 2^n개
-> **O(2^n)**

2) Greedy : 가격이 높은 보석, 혹은 (가격/무게) 의 값이 제일 높은 보석부터 먼저 골라서 넣는다.
빨간 보석을 먼저 고르고 그 다음 노란 보석을 고를 것이다. 그러면 10kg가 차고 가격의 합은 $16
그렇지만 다른 방법을 찾아보면, 왼쪽 보석 3개를 넣으면 10kg/$17이 된다는 걸 쉽게 알 수 있고, 1/2/3/4kg짜리를 하나씩 조합해서 넣으면 $19까지도 나온다. 즉 이 방법은 최적의 답을 보장하지 못한다.

0-1 Knapsack ->DP로 풀기

$$P[i,w]=\begin{cases} P[i-1,w] & \text{if } w_i > w \\ \max\{v_i + P[i-1, w-w_k], P[i-1,w]\} & \text{else} \end{cases}$$

P[i, w] : i개의 보석, 배낭의 무게 한도가 w일 때 **최적의 이익**

- i번째 보석이 배낭의 무게 한도보다 무거우면 넣을 수 없으므로 i번째 보석을 뺀 i-1개의 보석들을 가지고 구한 전 단계의 최적값을 그대로 가져온다
- 그렇지 않은 경우, i번째 보석을 위해 i번째 보석만큼의 무게를 비웠을 때의 최적값에 i번째 보석의 가격을 더한 값 or i-1개의 보석들을 가지고 구한 전 단계의 최적값 중 큰 것을 선택한다

0-1 Knapsack ->DP로 풀기

$$
\begin{aligned}
&if\ w_i \le W: \\
&\qquad if\ v_i + V[i-1, w-w_i] > V[i-1, w]: \\
&\qquad\qquad V[i,w] = v_i + V[i-1, w-w_i] \\
&\qquad else: \\
&\qquad\qquad V[i,w] = V[i-1,w] \\
&else: V[i,w] = V[i-1,w]
\end{aligned}
$$

- 1 2kg/$3
- 2 3kg/$4
- 3 4kg/$5
- 4 5kg/$6

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

0-1 Knapsack ->DP로 풀기

백준 온라인 저지 [12865번 문제]

```
# W: 배낭의 무게한도, wt: 각 보석의 무게, val: 각 보석의 가격, n: 보석의 수
def knapsack(W, wt, val, n):
    K = [[0 for x in range(W+1)] for x in range(n+1)]
    for i in range(n+1):
        for w in range(W+1):
            if i==0 or w==0:
                K[i][w] = 0
            elif wt[i-1] <= w:
                K[i][w] = max(val[i-1]+K[i-1][w-wt[i-1]], K[i-1][w])
            else:
                K[i][w] = K[i-1][w]
    return K[n][W]
```

(Longest Increasing Subsequence Problem, LIS)

최장 증가 부순서 문제
: 주어진 수열에서 오름차순으로 정렬된 가장 긴
부분 수열을 찾아라.
단, 부분 수열은 연속적이거나 유일할 필요는 없다.

- LIS는 앞에서부터 뒤로 숫자를 선택하며 부분수열을 구성해 나갈 때 증가하는 순서대로 숫자를 고르면서 고른 부분수열의 길이가 **최대길이**가 되도록 숫자를 선택
- LIS 보통 문제의 답은 한 수열에서 주어지는 LIS의 길이가 답이 된다.

(EX) A = (8, 3, 5, 10, 9, 12, 2, 15, 7)
증가 수열 : (8, 10, 12, 15), (3, 5, 9, 12, 15), (2, 7), ...
최장 증가 수열 : (3, 5, 9, 12, 15)

(Longest Increasing Subsequence Problem, LIS)

O(n^2)

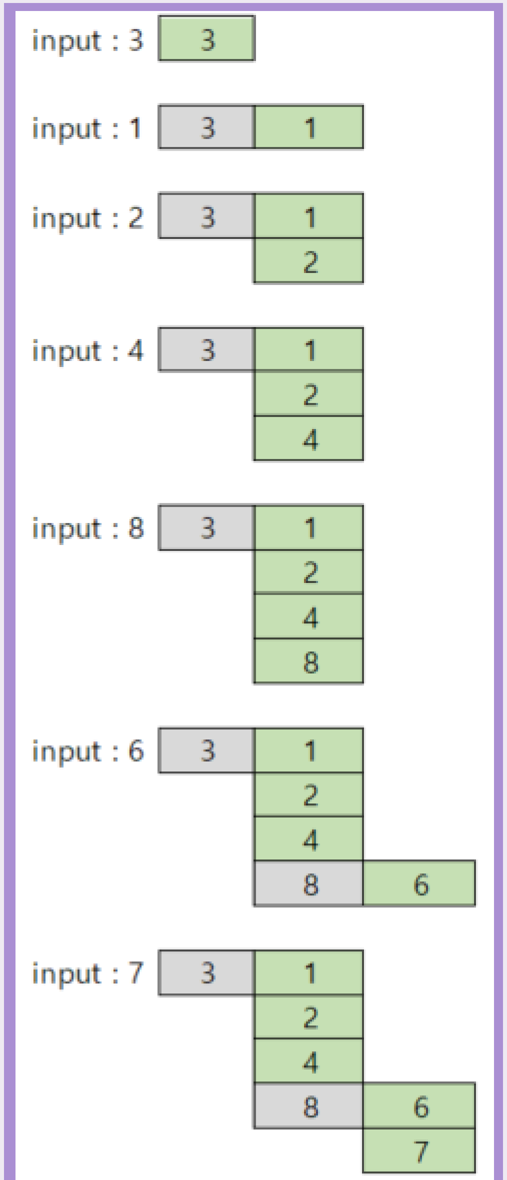
```
arr= [3,1,2,4,8,6,7]
n= len(arr)
dp= [1]* n

for i in range(1, n):
    for j in range(i):
        if arr[j] < arr[i]:
            dp[i]= max(dp[i], dp[j]+ 1)

Res = max(dp)
```

특정 시점에서 (i) 에서 맨 앞쪽까지 크기 비교를 수행한 뒤, 가장 큰 길이(DP 값)를 구해서 마지막에 자기자신(+1)을 더한다.

즉 D[i] = max(D[i-1], D[i-2], ... , D[2], D[[1]) + 1 이다.

(Longest Increasing Subsequence Problem, LIS)

임의의 배열에 입력이
들어올때마다 정렬된 형태를 유지하며
가장 큰값보다 큰입력이 들어올경우
새로 추가하고 그것이 아니라면
그 배열속의 그 값과 작거나 같고
제일 비슷한값을 이분탐색(lgN)
(lower_bound) 하여
그 인덱스의 값과 교환하는 방식
-> **O(nlongn)**



(Longest Increasing Subsequence Problem, LIS)

O(nlongn)

```
from bisect import bisect_left

arr= [3,1,2,4,8,6,7]
n= len(arr)
dp= [arr[0]]

for iin range(1, n):
    if dp[-1] < arr[i]:
        dp.append(arr[i])
    else:
        dp[bisect_left(dp, arr[i])]= arr[i]

res= max(dp)
```

Subset Sum

Subset Sum

: 배열의 일부 원소를 더해서 0이 되는 경우가
있는지 알아내는 프로그램

(EX) {3, -2, 5, 7, -3, 1}
-> **True**
{-2, 5, -3}을 부분집합으로 취해 더하면 0을 만드는 것이 가능

Subset Sum -> 1) binary operation

O(2^n)

```
bool naive(vector<int>& v) {
    int n = (int)v.size();
    for(int i=1;i<(1<<n);i++)
    {
        int s = 0;
        for(int j=0;j<n;j++) if((1<<j)&i) s+=v[j];
        if(s==0) return true;
    }
    return false; }
```

Subset Sum -> 2) dynamic programming

set[] = { 3, 24, 4, 12, 5, 2}

sum \ n	{} 0	{3} 1	{3,24} 2	{3,24,4} 3	{3,24,4,12} 4	{3,24,4,12,5} 5	{3,24,4,12,5,2} 6
0	true	true	true	true	true	true	true
1	false	false	false	false	false	false	false
2	false	false	false	false	false	false	true
3	false	true	true	true	true	true	true
4	false	false	false	true	true	true	true
5	false	false	false	false	false	true	true
6	false	false	false	false	false	false	true
7	false	false	false	true	true	true	true
8	false	false	false	false	false	true	true
9	false	false	false	false	false	true	true

기본 값 채우기

채우기 진행 방향

sum을 만들 수 있는지 여부

(n-1, sum)의 값이 true이므로 이 곳도 true.
{12}가 포함되지 않아도 이미 4을 만들 수 있다는 뜻

(n-1, sum - set[n])의 값이 true이므로 이 곳도 true.
{2}이 포함되면 비로소 6이 된다는 뜻

Subset Sum -> 2) dynamic programming

O(n*m)

```
bool isSubsetSum(int set[], int n, int sum) {
    // The value of subset[i][j] will be true
    // if there is a subset of set[0..j-1] with sum equal to i
    bool** subset = new bool*[sum + 1];
    for (int i = 0; i <= sum; ++i)
        subset[i] = new bool[n + 1];
     // If sum is 0, then answer is true
    for (int i = 0; i <= n; i++)
        subset[0][i] = true;
    // If sum is not 0 and set is empty, then answer is false
    for (int i = 1; i <= sum; i++)
        subset[i][0] = false;
    // Fill the subset table in botton up manner
    for (int i = 1; i <= sum; i++) {
        for (int j = 1; j <= n; j++) {
            subset[i][j] = subset[i][j - 1];
            if (i >= set[j - 1])
                subset[i][j] = subset[i][j] || subset[i - set[j - 1]][j - 1]; }}
```

Reference

[1] Dynamic Programming: 배낭 채우기 문제 (Knapsack Problem),
https://gsmesie692.tistory.com/113
[2] [알고리즘] 최장공통부순서(LCS, Longest Common Subsequence),
https://cholong-cholong.tistory.com/8?category=780036
[3] subset sum problem,
https://greatzzo.tistory.com/39