Dynamic Programing 3

Today's Content

- 0/1 Knapsack
- 0/∞ Knapsack

Dynamic Programming

Solve a problem with the help of subproblem

If the sub problems are overlapping

store and reuse

0/1 Knapsack

1Q. Given N items each with a weigth & value. Find max value which can be obtained by picking items such that, total weight of all items <= K

NOTE 1 : Every item can be picked at max 1 time

NOTE 2 : We cannot take a part of item

N = 4 items K=50

items :	0	1	2	3
weight [] :	20	10	30	40
value [] :	100	60	120	150
	5	6	4	3.75

Idea 1 ⟶ Greedily choose items based on V/W

Picked items :	1	0	~~2~~	
weight	10	20	~~30~~	160
value	60	100	~~120~~	
	6	5	~~4~~	

Idea 2 ⟶ Choose the highest value first

Picked items :	3	1	
weight	40	10	210
value	150	60	

Actual ans

Picked items :	2	0	
weight	30	20	220
value	120	100	

N = 7 K = 15

	0	1	2	3	4	5	6
w[]	4	1	5	4	3	7	4
v[]	3	2	8	3	7	10	5

kp(0-6 , 15)

~~6~~ → kp(0-5 , 15)

6 ✓ → kp(0-5 , 11) + 5 v[6]

kp(0-5 , 15):

~~5~~ → kp(0-4 , 15)

5 ✓ → kp(0-4 , 8) + 10 v[5]

kp(0-5 , 11):

~~5~~ → kp(0-4 , 11)

5 ✓ → kp(0-4 , 4) + 10 v[5]

kp(0-4 , 15): ~~4~~ → kp(0-3 , 15); 4 ✓ → kp(0-3 , 12) + 7 v[4]

kp(0-4 , 8): ~~4~~ → kp(0-3 , 8); 4 ✓ → kp(0-3 , 5) + 7 v[4]

kp(0-4 , 11): ~~4~~ → kp(0-3 , 11); 4 ✓ → kp(0-3 , 8) + 7 v[4]

kp(0-4 , 4): ~~4~~ → kp(0-3 , 4); 4 ✓ → kp(0-3 , 1) + 7 v(4)

DP state

dp[N][K+1] → DP table

dp[i][cap] ⟶ from item 0 ...... i with max capacity as cap. what is my max value ?

Pseudo code

```
int solve (int W[], int V[], int K) {
    int N = W.length();
    int dp[][] = new int[N][K+1]     // -1 init
    return knapsack (W, V, N-1, K);
}
```

K+1 because indexing in an array starts from 0

i : I am at index i

cap : Current capacity or weight limit

NOTE : pass dp table as a parameter.

```
int knapsack (int W[], int V[], int i, int cap) {
    if (i < 0 || cap == 0) {
        return 0
    }
    if (dp[i][cap] != -1) {
        return dp[i][cap]
    }
    // items     0   1   2 ....  i-1    | i  |
    // weights   w0  w1  w2 ...  wi-1   | wi |
    int ans = knapsack (W, V, i-1, cap)  // dont pick ith item
    if (cap >= W[i]) {
        // pick ith item
        ans = max( ans,
              knapsack (W, V, i-1, cap - W[i]) + V[i])
    }
    dp[i][cap] = ans
    return ans
}
```

TC : no. of states * TC per state

$(N * K) * O(1)$

$\longrightarrow$ TC $O(N * K)$

SC : $O(N * K)$ $\longrightarrow$ due to dp table

0/∞ knapsack

2Q) Same as above question, we can pick an element as many times as we want.

$N = 7 \quad K = 15$

	0	1	2	3	4	5	6
w[]	4	1	5	4	3	7	4
v[]	3	2	8	3	7	10	5

dp(0-6, 15)

~~6~~ → dp(0-5, 15)

6 ✓ → dp(0-6, 11) + V[6] (5)

dp(0-6, 11) + V[6]

~~6~~ → dp(0-5, 11)

6 ✓ → dp(0-6, 7) + 5 (V[6])

Generalized

items : 0 1 2 3 4 ...... i-1 i

weight : w_0 w_1 w_2 w_3 .... w_{i-1} w_i

dp(i, cap)

~~i~~ → dp(i-1, cap)

i ✓ → dp(i, cap - w[i]) + v[i]

Pseudocode

```
int solve (int w[], int v[], int k) {
    int N = w.length();
    int dp[][]= new int[N][k+1]      // -1 init
    return knapsack∞ (w, v, n-1, k);
}
```

k+1 because indexing in an array starts from 0

i : I am at index i

cap : current capacity or weight limit

NOTE : pass dp table as a parameter.

```
int knapsack∞ (int w[], int v[], int i, int cap ) {
    if ( i<0 || cap ==0 ) {
        return 0
    }
    if (dp[i][cap] != -1) {
        return dp[i][cap]
    }
    // items    0  1  2 .... i-1  | i  |
    // weights  w0 w1 w2 ... wi-1 | wi |
    int ans = knapsack (w, v, i-1, cap)  // dont pick ith item
    if (cap >= w[i]) {
        // pick ith item
        ans = max ( ans,
                knapsack (w, v, i, cap - w[i]) + v[i])
    }
    dp[i][cap] = ans
} return ans
```

change from prev Q

Break 22:40

Fibonacci

	0	1	2	3	4	5	6	7	8	9	....
Fib :	0	1	1	2	3	5	8	13	21	34	

```
int fib ( int N) {
    if( N <= 1) { return N }
    return fib(N-1) + fib (N-2)
}
```

TC : $O(2^N)$
SC : $O(N)$

Iterative steps

1> DP state

dp[i] = i^{th} fibonacci no.

final ans = dp[N]

int dp[N+1] // init with -1

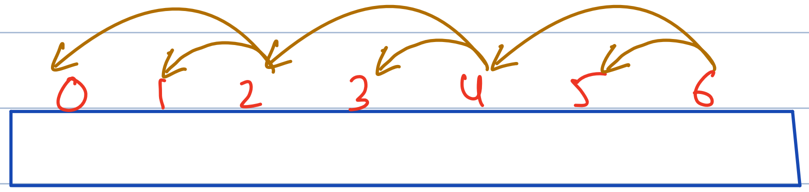
fib(N-1) + fib (N-2)

2> DP expression — Expression using dp table to find out our ans

dp[i] = dp[i-1] + dp[i-2]

3> Filling DP table iteratively

Fill the dp table in the reverse order of dependency



To resolve dependency we iterate from $l \longrightarrow r$

```
int fib ( int N) {
    int dp [N+1]

    for (i = 0 ; i <= N ; i++ ) {
        if ( i <= 1) {dp [i] = i }
        else {
            dp [i] = dp [i-1] + dp [i-2]
        }
    }
    return dp [N]
}
```

Handle edge cases

TC : O(N)

SC : O(N)

Ways to reach (0, 0) $\longrightarrow$ bottom right (N-1, M-1)

```
int ways (int i, int j) {
    if (i < 0 or j < 0) return 0
    if ( i == 0 and j == 0) return 1

    return ways (i-1, j) + ways (i, j-1)
}
```

Steps to convert into iterative code

1> **DP state**

dp[i][j] $\longrightarrow$ no. of ways to reach ith row j col A_{ij} cell.

final ans = dp[N-1][M-1]

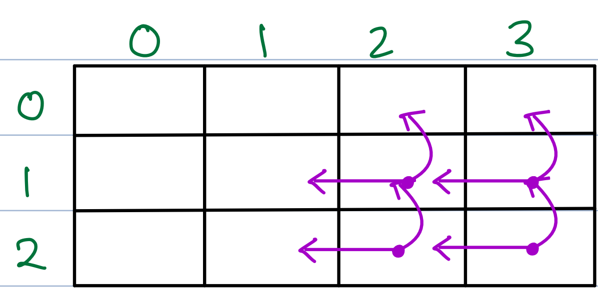
init dp[N][M]

2> **DP expression** ways (i-1, j) + ways (i, j-1)

dp[i][j] = dp[i-1][j] + dp[i][j-1]

3> Fill the DP table N = 3 M = 4

No. of ways to reach the cell 2, 3



dependency ↑ and ←

Left to Right
and Top to bottom.

iterate dp table ↓ and →

Top to bottom.
and Left to Right

	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10

	0	1	2	3
0	1	1	1	1
1	1	2	3	4
2	1	3	6	10

4) Code

```
int ways (int N , int M) {
    dp [N] [M]

    for (i=0 ; i<N ; i++) {
        for (j=0 ; j<M ; j++) {
            if (i==0 || j==0) {          Handle edge
                dp [i] [j] = 1              cases
            }
            else {
                dp [i] [j] = dp [i-1] [j] + dp [i] [j-1]
            }
        }
    }
    return dp [N-1] [M-1]
}
```

TC: $O(NM)$

SC: $O(NM)$