Group 8

Dynamic Programming

Recursive Definition

$$P(C) = \mathrm{Max}(p_0 + P(C - w_0), p_1 + P(C - w_1), \ldots, p_{n-1} + P(C - w_{n-1}))$$

1) Recursive Definition of Function P(C)

P() : maximum profit function

C : capacity weight in the knapsack

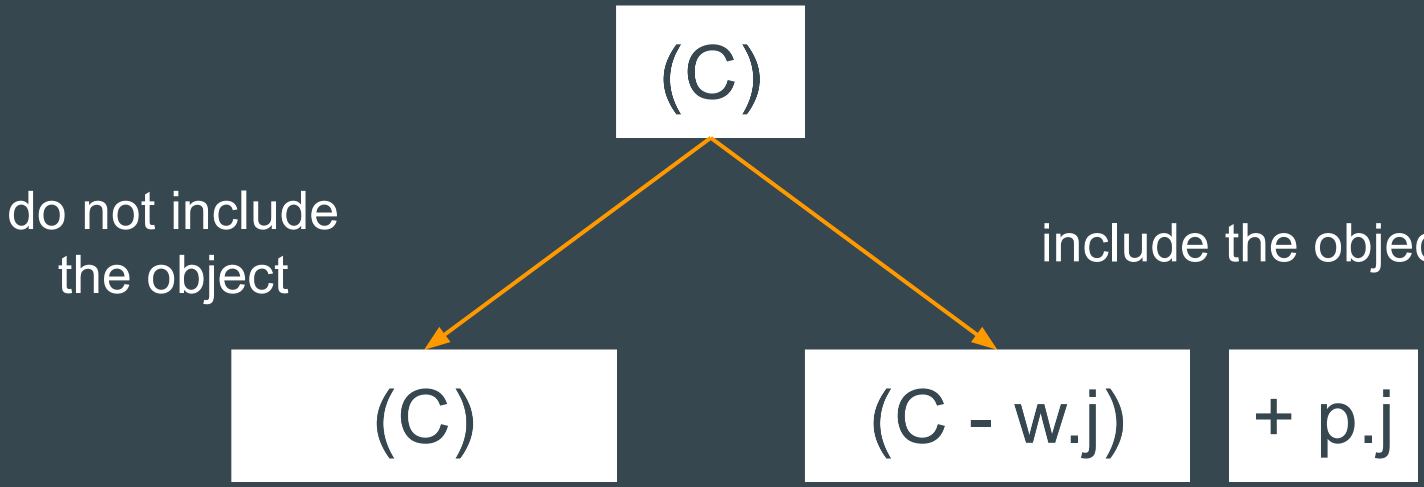
n : number of objects

j : subset of n objects

when C is 0 OR j is 0 $\longrightarrow$ $P(C, 0) = P(0, j) = 0$

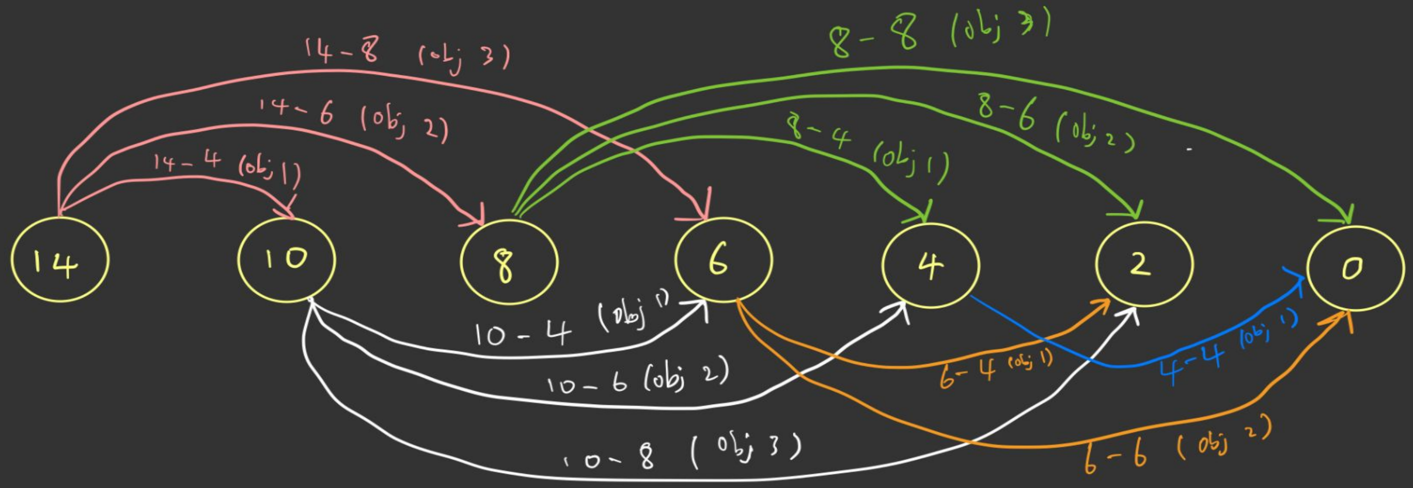
1) Recursive Definition of Function P(C)

```
for(j = 1; j < n; j++)
    if(w.j < C)
```



$$P(C) = max(P(C), P(C - w.j) + p.j)$$

2) Subproblem Graph



	1	2	3
w_i	4	6	8
p_i	7	6	9

- Current knapsack capacity minus away the weight of the object

3) Dynamic Programing Bottom Up Approach

1. Create array of profits with C+1 as size

```
int* prft = new int[max_wt+1];
```

2. Set the first elements in the array to 0

```
prft[0] = 0;
```

3) Dynamic Programing Bottom Up Approach

3. While traversing through profit array, traverse through each profit possible, default set it to be the same as before.

```
for(int i=1;i<=max_wt;i++){
    prft[i] = prft[i-1];
```

4. Check if the object can be contained in the knapsack given the current capacity i.

```
for(int i=1;i<=max_wt;i++){
    prft[i] = prft[i-1];
    for(int j = 0; j < item_type; j++){
        if(wt[j] <= i){...}
    }
}
```

3) Dynamic Programing Bottom Up Approach

5. While traversing object j, compare the current value of array[i] to the value of array[i-w[j]] + p[j] if the current capacity i is able to accommodate the object (i>w[j]).

```
int take_j = p[j]+prft[i-wt[j]];
if( take_j > prft[i])
    prft[i] = take_j;
```

3) Dynamic Programing Bottom Up Approach

```
int unlimitedKnapSack(int* wt, int* p, int max_wt, int item_type){
    int* prft = new int[max_wt+1];
    prft[0] = 0;
    for(int i=1;i<=max_wt;i++){
        prft[i] = prft[i-1];
        for(int j = 0; j < item_type; j++){
            if(wt[j] <= i){
                int take_j = p[j]+prft[i-wt[j]];
                if( take_j > prft[i])
                    prft[i] = take_j;
            }
        }
    }
    return prft[max_wt];
}
```

4) Running Result

a) P(14)

	1	2	3
w_i	4	6	8
p_i	7	6	9

Capacity	3
0	0
2	0
4	7
6	7
8	14
10	14
12	21
14	21

Can put 1 object of weight 4

Can put 2 objects of weight 4

Can put 3 objects of weight 4

Profit: 21

b) P(14)

	1	2	3
w_i	5	6	8
p_i	7	6	9

Capacity	3
0	0
4	0
5	7
6	7
7	7
8	9
9	9
10	14
11	14
12	14
13	16
14	16

Can put 1 object of weight 5

Can put 1 object of weight 8

Can put 2 objects of weight 5

Can put 1 object of weight 5 and 1 object of weight 8

Profit: 16

Thank You!