Project 3: Dynamic Programming

Group 9, Lab Group SS1
CZ2002

Unbounded Knapsack Problem

(Unlimited == Unbounded)

Subproblem Graph

	0	1	2
wi	4	6	8
pi	7	6	9

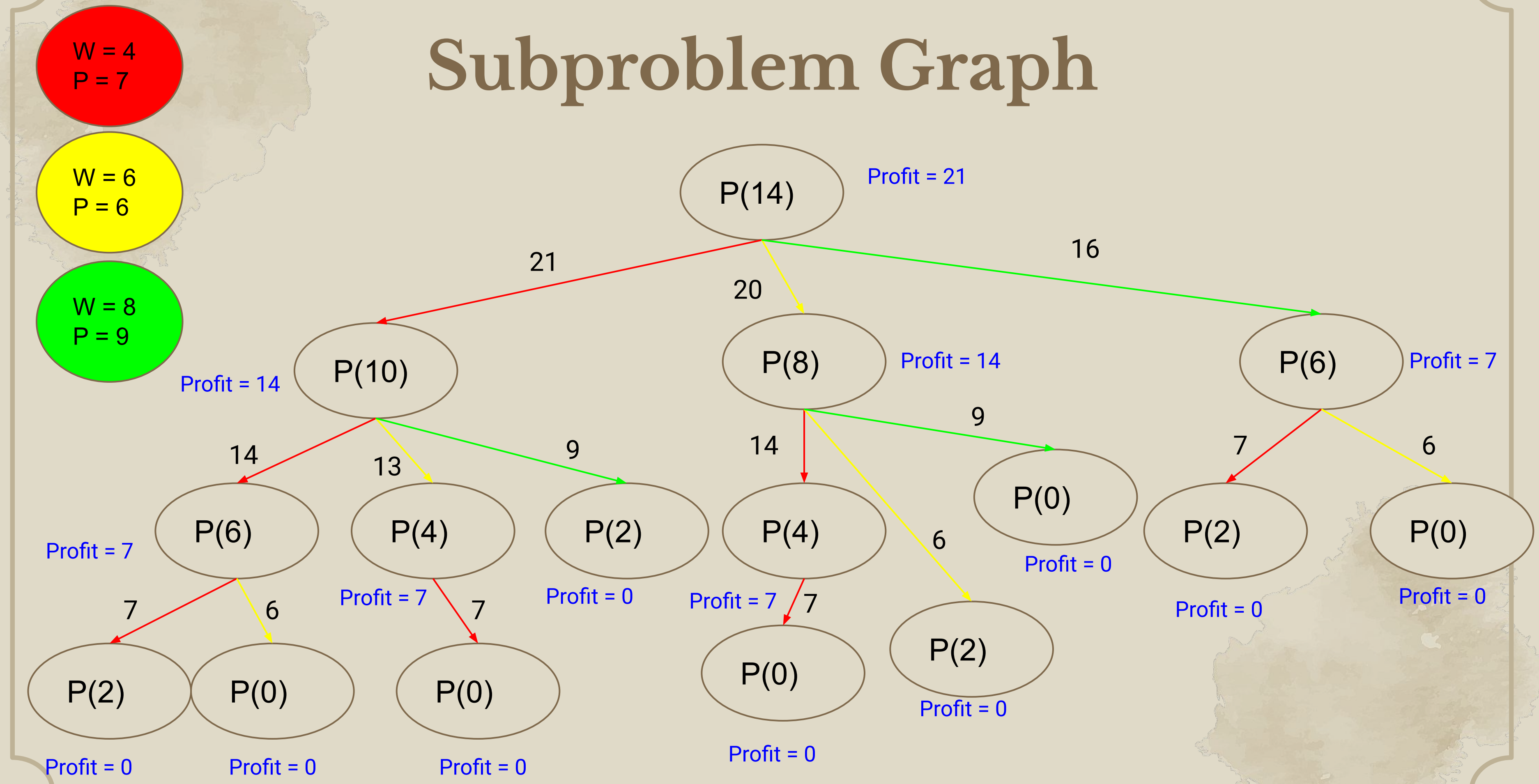
Definitions:

- **P(C): Maximum profit at particular C value (Dependent on `w[i]` and `p[i]`)**
- **C: Capacity of knapsack (Dependent on `w[i]`)**

Mechanics:

1. When C = 0 or can no longer be reduced by any of the weight, P(C) = 0
2. Return to previous state, save the profit of the weight that led to dead end.
3. Continue down alternative paths with other weights and repeat (1) and (2)
4. Compare the profits and save the higher profit.
5. Rinse and repeat 1-5 until function returns back to starting point with max profit

Subproblem Graph



Dynamic Programming Algorithm

1. Create a new array `soln` of size C+1. The array indicates the maximum profit we can achieve with a knapsack capacity.

   `soln[i]` = max profit with a knapsack of capacity i

   Aim is to find `soln[C]`

   Initialise the array to zeros

2. We iterate over all the elements available for each knapsack capacity between 1 to C and determine if it can be used to achieve a greater profit
3. The recurrence relation would be

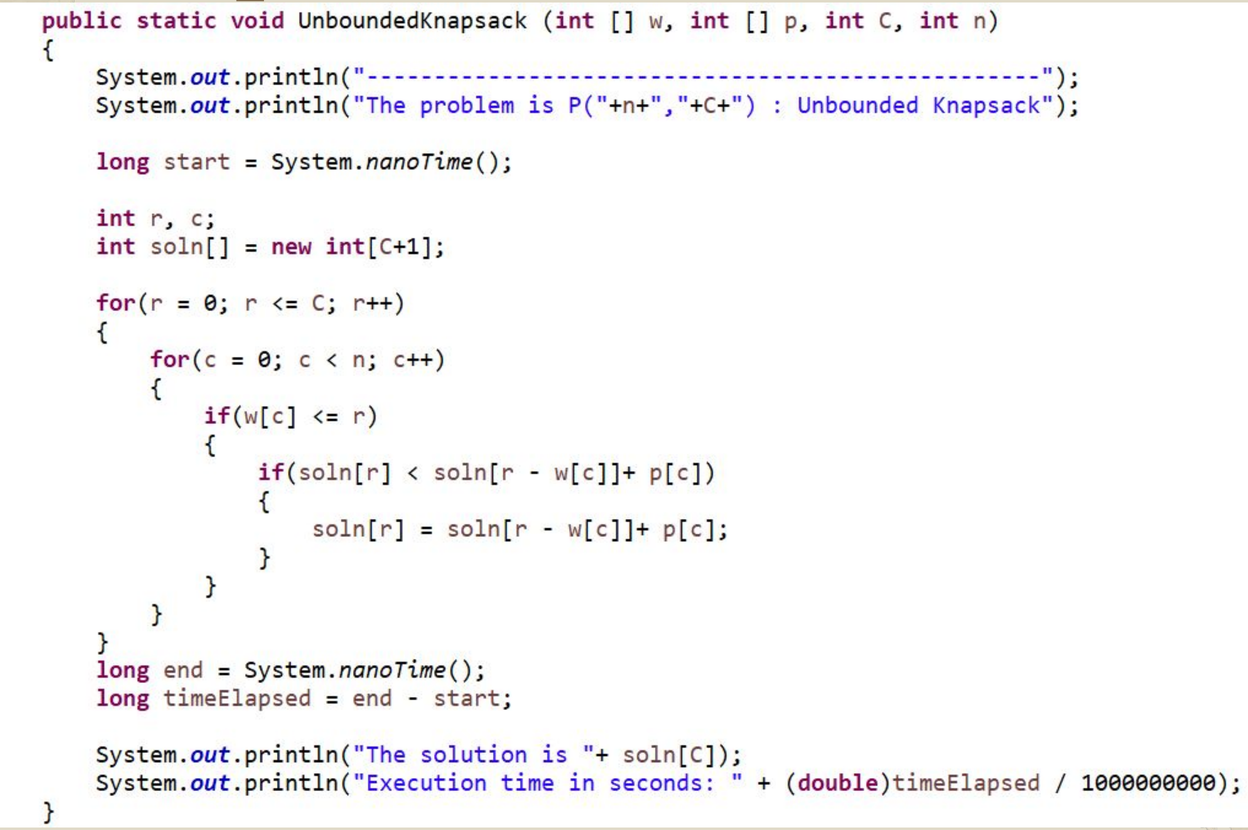
```
soln[i] = max(soln[i], soln[i-w[j]] + p[j])
```

if `w[j] < i` => item j is taken

Dynamic Programming Recurrence Relation

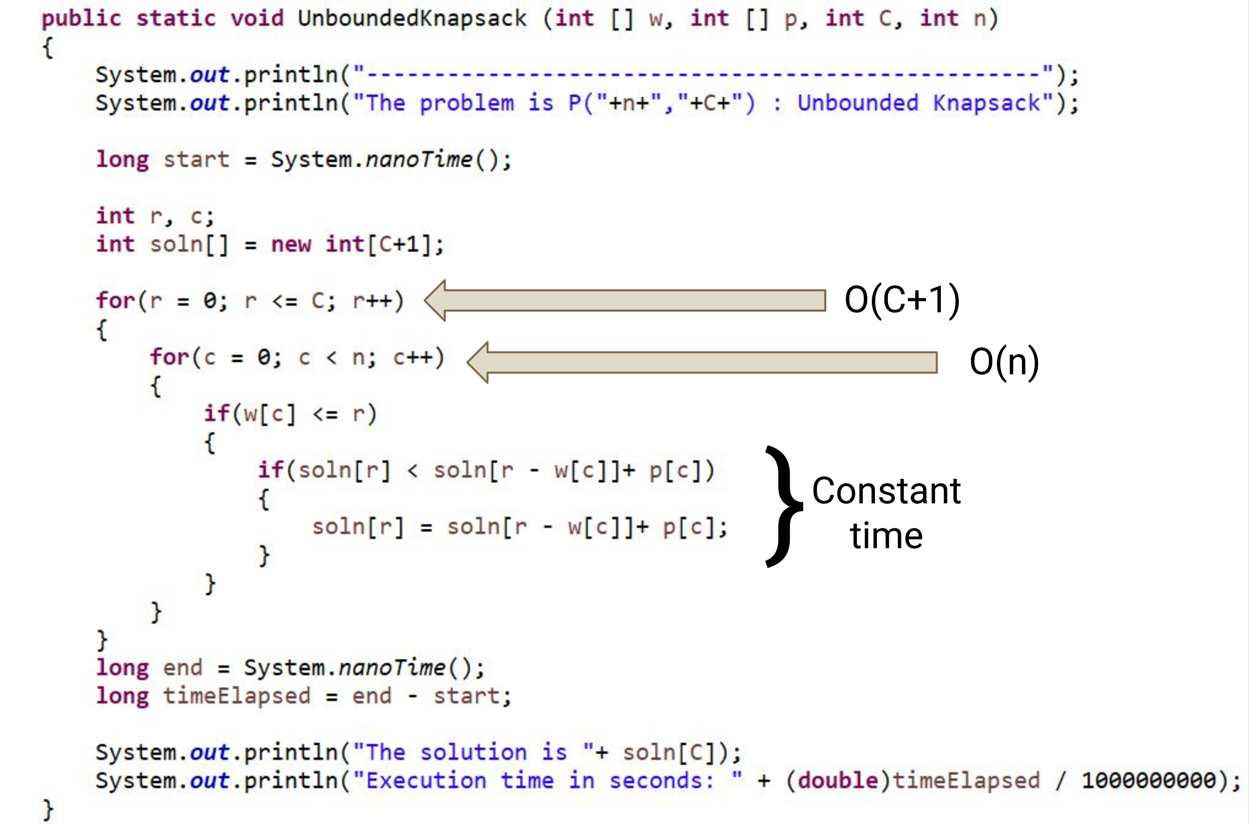
```
soln[i] = max(soln[i], soln[i-w[j]] + p[j])
```

Implementation in code



```
public static void UnboundedKnapsack (int [] w, int [] p, int C, int n)
{
    System.out.println("--------------------------------------------------");
    System.out.println("The problem is P("+n+","+C+") : Unbounded Knapsack");

    long start = System.nanoTime();

    int r, c;
    int soln[] = new int[C+1];

    for(r = 0; r <= C; r++)
    {
        for(c = 0; c < n; c++)
        {
            if(w[c] <= r)
            {
                if(soln[r] < soln[r - w[c]]+ p[c])
                {
                    soln[r] = soln[r - w[c]]+ p[c];
                }
            }
        }
    }
    long end = System.nanoTime();
    long timeElapsed = end - start;

    System.out.println("The solution is "+ soln[C]);
    System.out.println("Execution time in seconds: " + (double)timeElapsed / 1000000000);
}
```

Complexity Analysis



```
public static void UnboundedKnapsack (int [] w, int [] p, int C, int n)
{
    System.out.println("--------------------------------------------------");
    System.out.println("The problem is P("+n+","+C+") : Unbounded Knapsack");

    long start = System.nanoTime();

    int r, c;
    int soln[] = new int[C+1];

    for(r = 0; r <= C; r++)
    {
        for(c = 0; c < n; c++)
        {
            if(w[c] <= r)
            {
                if(soln[r] < soln[r - w[c]]+ p[c])
                {
                    soln[r] = soln[r - w[c]]+ p[c];
                }
            }
        }
    }
    long end = System.nanoTime();
    long timeElapsed = end - start;

    System.out.println("The solution is "+ soln[C]);
    System.out.println("Execution time in seconds: " + (double)timeElapsed / 1000000000);
}
```

O(C+1)

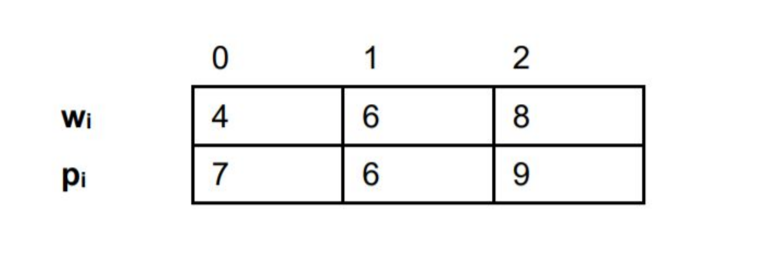
O(n)

Constant time

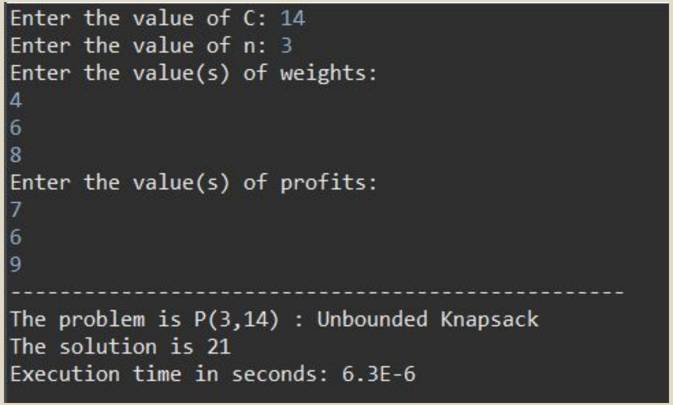
Total time complexity = O(Cn)

Total space complexity = O(C)

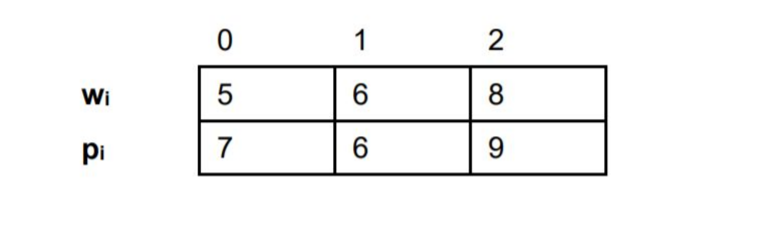
Results using Dynamic Programming



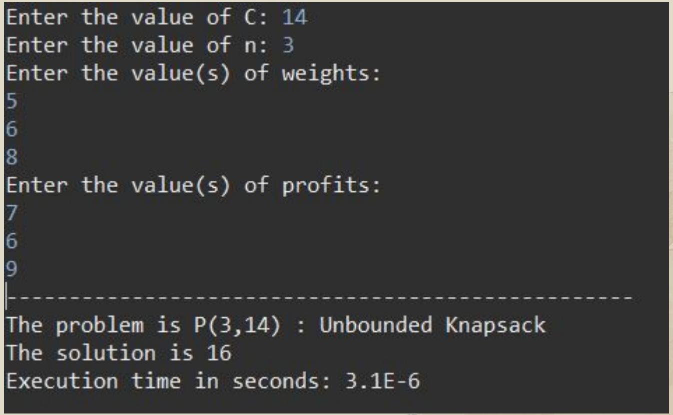
	0	1	2
w_i	4	6	8
p_i	7	6	9



```
Enter the value of C: 14
Enter the value of n: 3
Enter the value(s) of weights:
4
6
8
Enter the value(s) of profits:
7
6
9
--------------------------------------------------
The problem is P(3,14) : Unbounded Knapsack
The solution is 21
Execution time in seconds: 6.3E-6
```



	0	1	2
w_i	5	6	8
p_i	7	6	9



```
Enter the value of C: 14
Enter the value of n: 3
Enter the value(s) of weights:
5
6
8
Enter the value(s) of profits:
7
6
9
--------------------------------------------------
The problem is P(3,14) : Unbounded Knapsack
The solution is 16
Execution time in seconds: 3.1E-6
```

The End
Q&A