CSE- 208
Algorithms Lab

Lab: 09
Dynamic Programming

Fahad Ahmed
Lecturer, Dept. of CSE
E-mail: fahadahmed@uap-bd.edu

Task-09

Problem:

Write a program for generating Fibonacci Number using DP:

- ✓ **Tabulation: Bottom Up**
- ✓ **Memorization: Top down**

Solve Maximum Subarray Sum Problem using Kadane's Algorithm (DP).

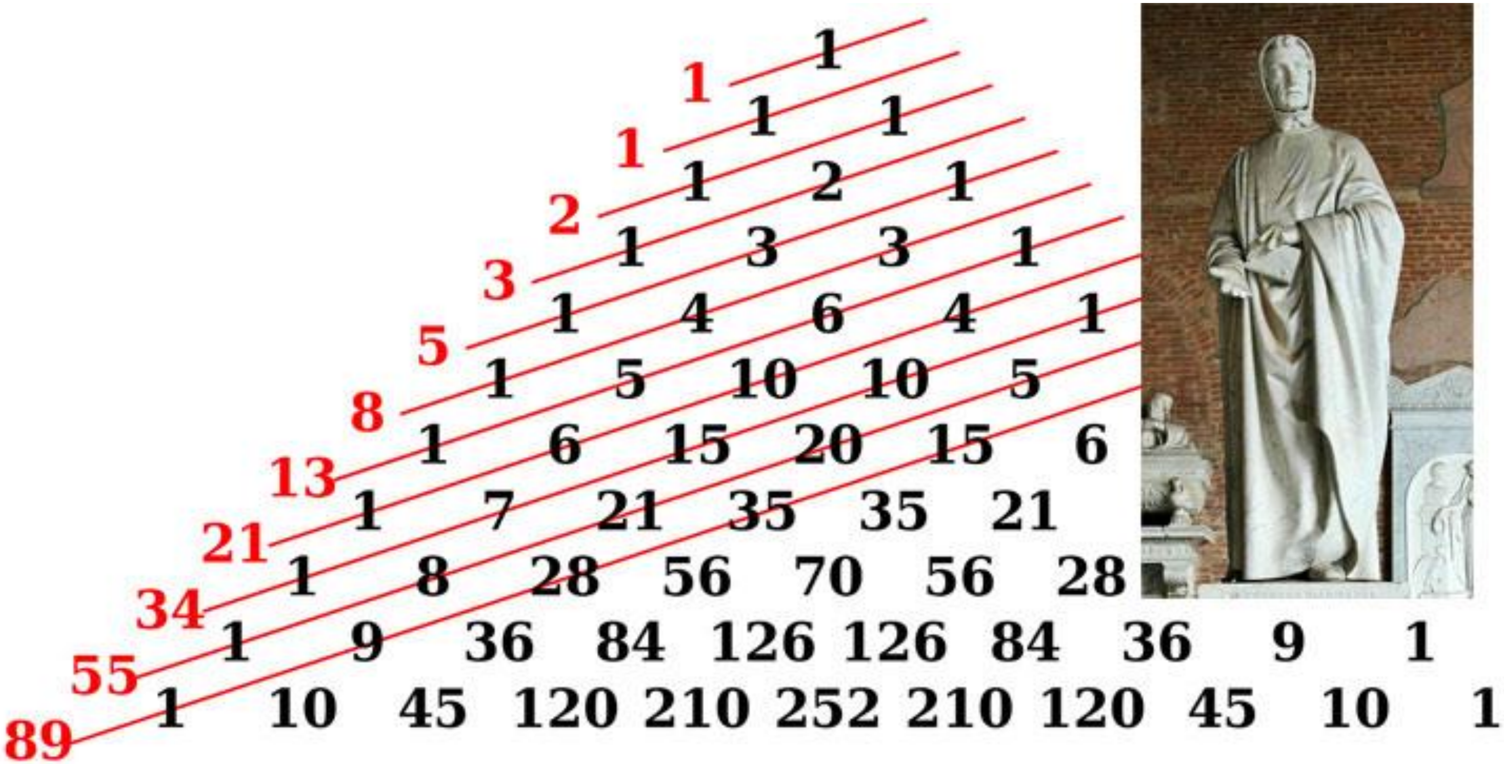
Task-09

Problem:

Implement the following algorithm using Dynamic Programming:

1. **0-1 Knapsack Problem**

Fibonacci Number



Let's think about Fibonacci number

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

n	0	1	2	3	4	5	6	7	8	9	10
F(n)	1	1	2	3	5	8	13	21	34	55	89

Pseudo code for the recursive algorithm:

```
Procedure F(n)
    if n==0 or n==1 then
        return 1
    else
        return F(n-1) + F(n-2)
```

- Time Complexity: $\Theta(2^n)$
- Is it a good algorithm?
- Is there any way to improve?

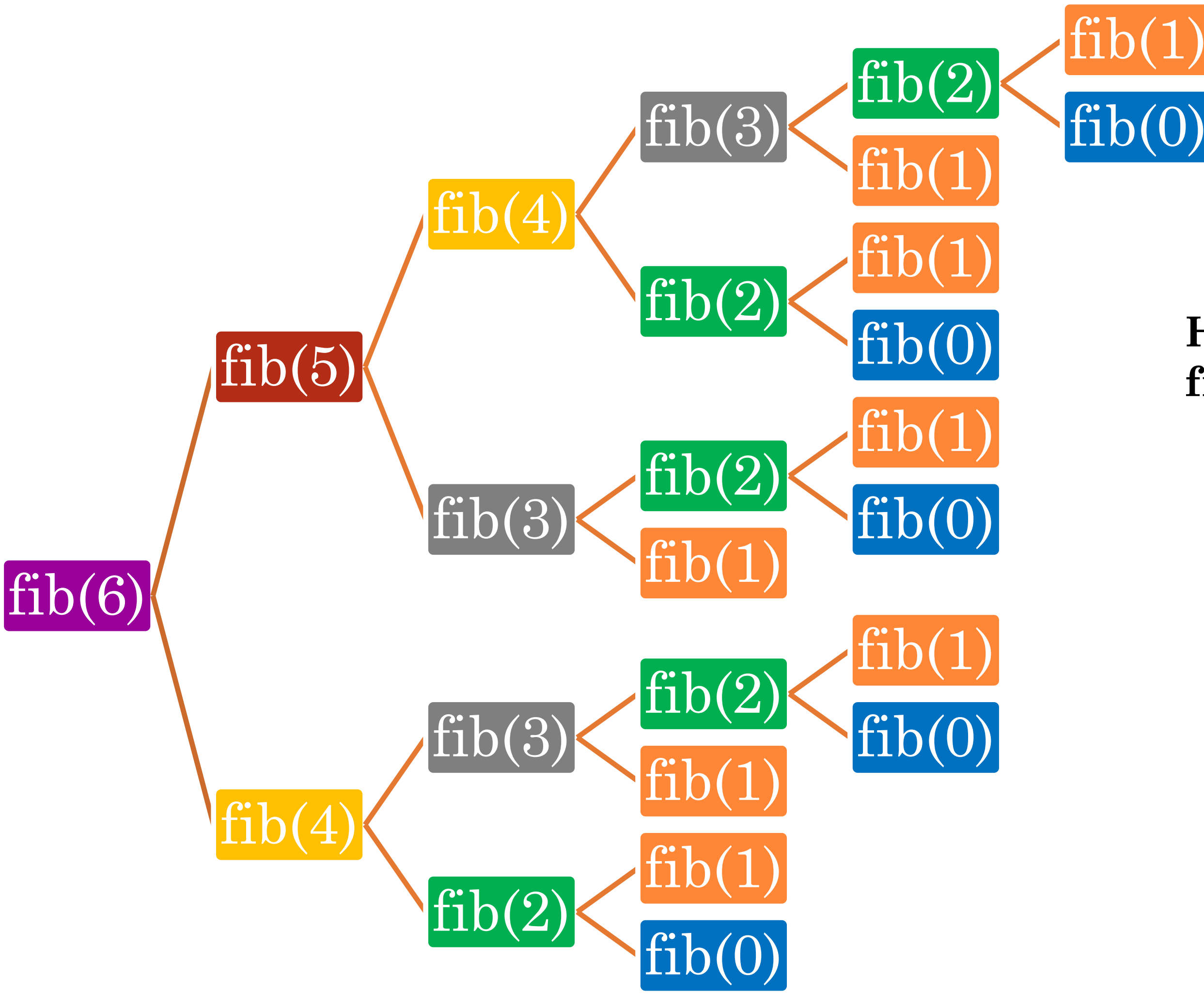
METHOD 1 (USE RECURSION)

```
#include<stdio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 6;
  printf("%d", fib(n));
  return 0;
}
```

Too many repeated work done

Fibonacci number – recursion tree for n=6



How many times each fib(n) is called?

Function	Count
fib(5)	1
fib(4)	2
fib(3)	3
fib(2)	5
fib(1)	7
fib(0)	5

Method 2 (Use Dynamic Programming)

Tabulation: Bottom Up

It starts from solving the lowest level sub-problem. The solution to the lowest level sub-problem will help to solve next level sub-problem, and so forth.

```
#include<stdio.h>
int fib(int n)
{// 1 extra to handle case, n = 0
  int f[n+1];
  int i;
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++)
      /* storing*/
      f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

We can avoid the repeated work done in **method 1** by storing the Fibonacci numbers calculated so far.

```
int main ()
{
  int n = 6;
  printf("%d", fib(n));
  return 0;
}
```

DP using memorization (Top down approach)

It starts from solving the highest-level sub-problems.

```
#include <bits/stdc++.h>
using namespace std;
int dp[10];
int fib(int n)
{
    if (n <= 1)
        return n;
    int first, second;

    if (dp[n - 1] != -1)
        first = dp[n - 1];
    else
        first = fib(n - 1);

    if (dp[n - 2] != -1)
        second = dp[n - 2];
    else
        second = fib(n - 2);
    // memoization
    return dp[n] = first + second;
}
```


```
int main()
{
    int n = 9;

    memset(dp, -1, sizeof(dp));

    cout << fib(n);

    return 0;

}
```

The Maximum Sub-Array Sum

Kadane's Algorithm

Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max_ending_here is used for this).

And keep track of maximum sum contiguous segment among all positive segments (max_so_far is used for this).

Each time we get a positive sum compare it with max_so_far and update max_so_far if it is greater than max_so_far

MAXIMUM SUBARRAY

Efficient solutions

Five solutions for this problem:-

1. Brute force approach I : Using 3 nested loops

2. Brute force approach II : Using 2 nested loops

3. Divide and Conquer approach : Similar to merge sort

4. **Dynamic Programming Approach : Kadanes's Algorithm**

Maximum Subarray– Kadane's Algorithm

```
Initialize:
    max_so_far = INT_MIN
    max_ending_here = 0

Loop for each element of the array
        (a) max_ending_here = max_ending_here + a[i]

        (b) if(max_so_far < max_ending_here)
                  max_so_far = max_ending_here

        (c) if(max_ending_here < 0)
                  max_ending_here = 0
return max_so_far
```

Maximum Subarray– Kadane's Algorithm

```
int maxSubArraySum(int a[], int size)
{
   int max_so_far = INT_MIN, max_ending_here = 0;

   for (int i = 0; i < size; i++)
   {
      max_ending_here = max_ending_here + a[i];
      if (max_so_far < max_ending_here)
         max_so_far = max_ending_here;

      if (max_ending_here < 0)
         max_ending_here = 0;
   }
   return max_so_far;
}
```

Notice that each element has been visited only once.

Time Complexity = O(n)

Maximum Subarray– Additional Requirements

Print the subarray with the maximum sum, we maintain indices whenever we get the maximum sum.

Time Complexity: *O(n)*

```
Input:
{-2, -3, 4, -1, -2, 1, 5, -3}
```

Output:
Maximum contiguous sum is 7
Starting index 2
Ending index 6

Task to think

Maximum Product Subarray:

Given an array that contains both positive and negative integers, find the product of the maximum product subarray.

Time complexity is O(n)

Solution Link: https://www.geeksforgeeks.org/maximum-product-subarray/?ref=lbp

Try also: **Using Two Traversals way**

{-2, -3, 4,} MSS=4 MPS=24

Task to think

Find the longest subarray in a binary array with an equal number of 0s and 1s

Problem+ Solution Link:
https://practice.geeksforgeeks.org/problems/largest-subarray-of-0s-and-1s/1

0-1 Knapsack

Knapsack 0-1 Problem – Recursive Formula

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- **First case:** $w_k > w$
  - Item *k* can't be part of the solution! If it was the total weight would be > w, which is unacceptable.

- **Second case:** $w_k \le w$
  - Then the item *k* can be in the solution, and we choose the case with greater value.

Knapsack 0-1 Problem

Consider the problem having weights and profits are:

- **Weights**: {2, 3, 4, 5}
- **Profits:** {3, 4, 5, 6}
- The weight of the knapsack is 5 kg
- The number of items is 4

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

// Initialize the base cases

for w = 0 to W

  B[0,w] = 0

for i = 1 to n

  B[i,0] = 0

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 1$
$w - w_i = -1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w - w_i = 0$

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 3}$
$w\text{-}w_i = 1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

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

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 4}$
$w - w_i = 2$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 5}$
$w-w_i = 3$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 1}$
$w - w_i = -2$

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

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			
3	0					
4	0					

i = 2
$v_i = 4$
$w_i = 3$
w = 2
$w - w_i = -1$

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

Items:
1: (2,3)
$\boxed{2: (3,4)}$
3: (4,5)
4: (5,6)

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		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w - w_i = 0$

Items:
1: (2,3)
$\boxed{2: (3,4)}$
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 4}$
$w\text{-}w_i = 1$

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

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					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 5}$
$w - w_i = 2$

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

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		
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 1..3}$
$w - w_i = -3..-1$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

KNAPSACK 0-1 EXAMPLE

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 4}$
$w-w_i = 0$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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					

i = 3
$v_i = 5$
$w_i = 4$
w = 5
$w - w_i = 1$

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 6}

Items:
1: (2,3)
2: (3,4)
3: (4,5)
$\boxed{\text{4: (5,6)}}$

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	

$i = 4$
$v_i = 6$
$w_i = 5$
w = 1..4
$w\text{-}w_i = -4..-1$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w - w_i = 0$

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack 0-1 Example

Weights: {2, 3, 4, 5}
Profits: {3, 4, 5, 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**

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Algorithm

- This algorithm only finds the max possible value that can be carried in the knapsack
  - The value in dp[n,W]

- To know the ***items*** that make this maximum value, we need to trace back through the table.

Knapsack 0-1 Algorithm Finding the Items

- Let i = n and k = W

  if dp[i, k] $\neq$ dp[i-1, k] then

  mark the i^{th} item as in the knapsack

  i = i-1, cpt = cpt-w_i

  else

  i = i-1 // Assume the i^{th} item is not in the knapsack

  // Could it be in the optimally packed knapsack?

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

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

$i = 4$
$k = 5$
$v_i = 6$
$w_i = 5$
dp[i,k] = 7
dp[i-1,k] = 7

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

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

i = 3
k = 5
$v_i = 5$
$w_i = 4$
dp[i,k] = 7
dp[i-1,k] = 7

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2

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

$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
$\mathbf{dp[i,k] = 7}$
$dp[i-1,k] = 3$
$k - w_i = 2$

Knapsack 0-1 Algorithm Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$dp[i,k] = 3$
$dp[i-1,k] = 0$
$k - w_i = 0$

Knapsack 0-1 Algorithm
Finding the Items

Items:

1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Knapsack:

Item 2
Item 1

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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$dp[i,k] = 3$
$dp[i-1,k] = 0$
$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:
Item 1 and Item 2

How to find the items that are in the sack?

```
while (n != 0)
  {
    if (dp[n][cpt] != dp[n - 1][cpt])
    {
      printf("\nPackage %d with Wt = %d and Val = %d\n",n,  wt[n-1], val[n-1] );
      cpt = cpt - wt[n-1];
    }
    n--;
  }
```

KNAPSACK 0-1 PROBLEM – RUN TIME

```
for w = 0 to W
    dp[0,w] = 0
```

O(*W*)

```
for i = 1 to n
    [i,0] = 0
```

O(*n*)

```
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

Repeat *n* times

O(*W*)

What is the running time of this algorithm?

O(*nW*) – *of course, W can be mighty big***

What is an analogy in world of sorting?

Remember that the brute-force algorithm takes: **$O(2^n)$**

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
1	2	1											
2	1	2											
3	7	3								A			
4	6	4											
5	12	6											

← Capacity

Any cell in the table represents the maximum value attained by choosing items from i items (not i^{th}) in a sack of capacity listed in the header. For example, the cell with value "A" represents that we can add items of total value "A" from 3 items and with a sack capacity=7 which is represented as T[3,7] = A

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(\ T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0											
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

← Capacity

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(\ T(i-1, c),\ \ v[i] + T(i-1, c - w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1											
2	1	2											
3	7	3											
4	6	4											
5	12	6											

Capacity

If i=0, no items are available, to put to the sack, the maximum value we can attain is 0.

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(\ T(i-1, c),\ \ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

If bag capacity is 0, we can't add anything into the sack. So, attained value is 0.

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(\ T(i-1, c),\ \ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

T[1,1] = Max(T[1-1,1], v_1 + T[1-1,1-1])
= Max(T[0,1], 2 + T[0,0])
= Max(0, 2+0) = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = max($T(i-1, c)$, $v[i] + T(i-1, c-w[i])$)

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$$
\begin{aligned}
T[1,1] &= \text{Max}(T[1-1,1],\ v_1 + T[1-1,1-1]) \\
&= \text{Max}(T[0,1],\ 2 + T[0,0]) \\
&= \text{Max}(0,\ 2+0) = 2
\end{aligned}
$$

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(\ T(i-1, c),\ v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2									
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$$\begin{aligned} T[1,1] &= \text{Max}(T[1\text{-}1,1],\ v_1 + T[1\text{-}1,1\text{-}1]) \\ &= \text{Max}(\ T[0,1],\ 2 + T[0,0]) \\ &= \text{Max}(0,\ 2+0) = 2 \end{aligned}$$

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(\ T(i-1, c),\ v[i] + T(i-1, c - w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2 Going w_1 cell back	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$$T[1,2] = \text{Max}(T[1-1,1],\ v_1 + T[1-1,2-1])$$
$$= \text{Max}(\ T[0,1],\ 2 + T[0,1])$$
$$= \text{Max}(0,\ 2+0) = 2$$

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2								
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

$$T[1,2] = \text{Max}(T[1-1,1],\ v_1 + T[1-1,2-1])$$
$$= \text{Max}(T[0,1],\ 2 + T[0,1])$$
$$= \text{Max}(0,\ 2+0) = 2$$

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

For any c >= 1,
$T[1,c] = \text{Max}(T[1-1,1], v_1 + T[1-1,c-1])$
$= \text{Max}(T[0,1], 2 + T[0,c-1])$
$= \text{Max}(0, 2+0) = 2$

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T(2,1)= T[2-1,1] as $w_2 > c$
= T[1,1] = 2

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T(2,1)= T[2-1,1] as $w_2 > c$
= T[1,1] = 2

KNAPSACK SIMULATION

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
T[i, c] = T[i-1,c]
else
T[i, c] = $\max(T(i-1,c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2									
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_2 cell back

$$T[2,2] = \text{Max}(T[2-1,2],\ v_2 + T[1-1,\ 2-2])$$
$$= \text{Max}(\ T[1,2],\ 1 + T[1,0])$$
$$= \text{Max}(2,\ 1) = 2$$

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

Capacity

T[2,2] = 2

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] = $\max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2								
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

Going w_2 cell back

T[2,3] = Max(T[2-1, 3], v_2 + T[2-1, 3-2])
= Max(T[1,3], 1+ T[1,1])
= Max(2, 3) = 3

Knapsack simulation

Assume,
Sack capacity, C = 10
Available Items
V = 7 2 1 6 12
W =3 1 2 4 6

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  T[i, c] $= \max(T(i-1, c), v[i] + T(i-1, c-w[i]))$

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

So, simplest version is compare 1) the cell above the current cell and 2) v_i + value of w_i cell backward in previous row. Populate the current cell with whichever value is bigger.

Populate the table with this logic.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Sack = {}
1. Start with 21 (Green cell) and compare with the one above it (16).
2. As 21 and 16 are not equal item# 5 is included in the sack.
3. Go 6(weight if item) units back in previous row which is the next cell to check.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

included

Sack = {5}

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

← Capacity

Not Included

Included

Weight Left: 10-6=4

Sack = {5}

1. Compare T[4,4] 9 (Green cell) with the one above it (9).
2. As both cell has same value item# 4 is not included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	
2	1	2	0	2	2	3	3	3	3	3	3	3	3	
3	7	3	0	2	2	7	9	9	10	10	10	10	10	Included
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	Included

Sack = {5, 3}

1. Compare T[3,4] 9 (Green cell) with the one above it (3).
2. As the cells have different values item# 3 is included in the sack.
3. Go 3 units back in previous row which is the next cell to check.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	2	2	2	2	2	2	2	2	2
2	1	2	0	2	2	3	3	3	3	3	3	3	3
3	7	3	0	2	2	7	9	9	10	10	10	10	10
4	6	4	0	2	2	7	9	9	10	13	15	15	16
5	12	6	0	2	2	7	9	9	12	14	15	19	21

Capacity

Not Included

Include
d

Not Included

Included

Weight Left: 4-3=1

Sack = {5, 3}

1. Compare T[2,1] 2 (Green cell) with the one above it (2).
2. As both cells have same values item# 2 is not included in the sack.

Knapsack simulation

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	Included
2	1	2	0	2	2	3	3	3	3	3	3	3	3	Not Included
3	7	3	0	2	2	7	9	9	10	10	10	10	10	Included
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	Included

Weight Left: 1-1=0

Sack = {5, 3, 1}

1. Compare T[1,1] 2 (Green cell) with the one above it (0).
2. As the cells have different values item# 1 is included in the sack.

KNAPSACK SIMULATION

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	2	1	0	2	2	2	2	2	2	2	2	2	2	**Included**
2	1	2	0	2	2	3	3	3	3	3	3	3	3	Not Included
3	7	3	0	2	2	7	9	9	10	10	10	10	10	**Included**
4	6	4	0	2	2	7	9	9	10	13	15	15	16	Not Included
5	12	6	0	2	2	7	9	9	12	14	15	19	21	**Included**

Sack = {5, 3, 1}
As we have reached the 0^{th} row, we are done with item selection. So, the sack contains **1, 3 and 5** item with value = 2+7+12 = 21

Knapsack simulation

```
for (int i = 1; i <= n; i++)
   {
       for (int j = 0; j <= cpt; j++)
       {
         if(wt[i-1]<=j)
             dp[i][j]=max((val[i-1]+dp[i - 1][j - wt[i - 1]]),dp[i-1][j]);

          else
          dp[i][j] = dp[i-1][j];

     }
}
printf("Max Value: %d",dp[n][cpt]);
```

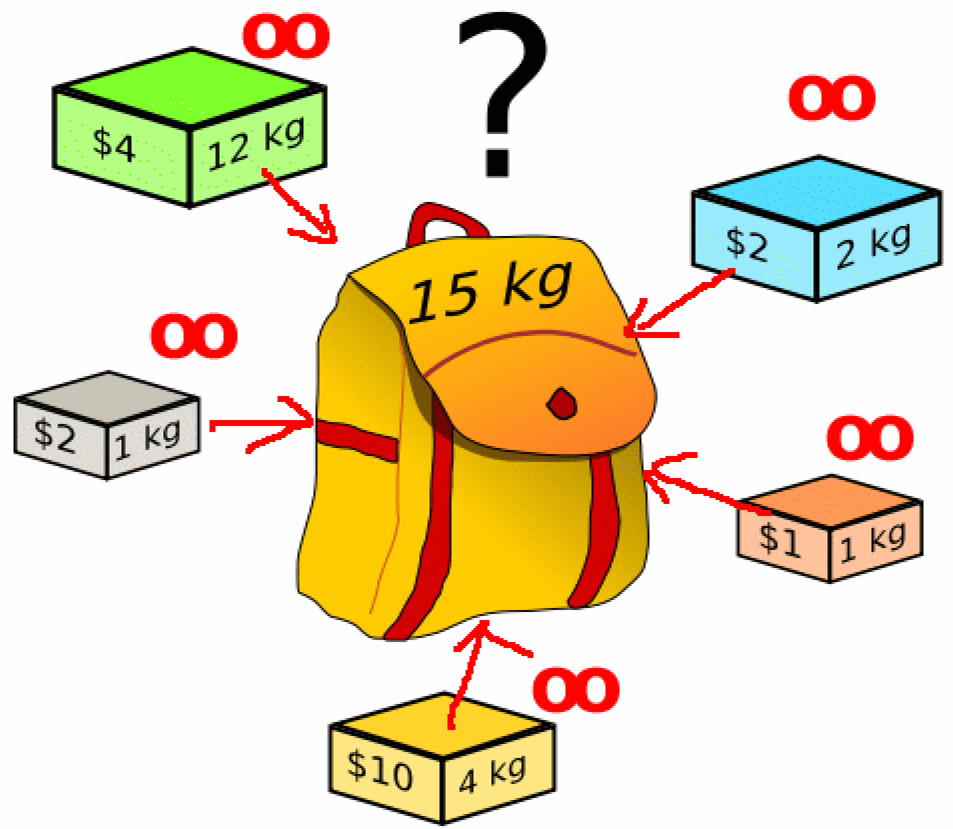
Knapsack space optimized simulation

```
int knapSack(int cpt, int wt[], int val[], int n)
{
    int dp[cpt + 1];
    memset(dp, 0, sizeof(dp));

   for (int i=0; i<n; i++)
          for (int j=0; j<=cpt; j++)
                 if (wt[i - 1] <= j)
              dp[j] = max(dp[j],dp[j - wt[i - 1]] + val[i - 1]);

    return dp[cpt];
}
```

Unbounded Knapsack
(Repetition of items allowed)



KNAPSACK UNBOUNDED

Given a knapsack weight **W** and a set of **n** items with certain value val_i and weight wt_i, we need to calculate the maximum amount that could make up this quantity exactly.

This is different from classical Knapsack problem, here we are allowed to use unlimited number of instances of an item.

Knapsack Unbounded

Example:
items: {Apple, Orange, Melon}
weights: {1, 2, 3}
profits: {15, 20, 50}
capacity: 5

Different Profit Combinations:
5 Apples (total weight 5) => 75 profit
1 Apple + 2 Oranges (total weight 5) => 55 profit
3 Apples + 1 Orange (total weight 5) => 65 profit
2 Apples + 1 Melon (total weight 5) => 80 profit
1 Orange + 1 Melon (total weight 5) => 70 profit

Best Profit Combination : 2 Apples + 1 Melon with 80 profit.

KNAPSACK SIMULATION - UNBOUNDED

- Can include multiple instances of the same resource

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0										
2	1	2	0										
3	7	3	0										
4	6	4	0										
5	12	6	0										

← Capacity

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i,c] = \max(T[i-1,c], v[i] + T[i,c-w[i]])$

KNAPSACK SIMULATION - **UNBOUNDED**

- Can include multiple instances of the same resource

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	2	1	0	2	4	6	8	10	12	14	16	18	20
2	1	2	0	2	4	6	8	10	12	14	16	18	20
3	7	3	0	2	4	7	9	11	14	16	18	21	23
4	6	4	0	2	2	7	9	11	14	16	18	21	23
5	12	6	0	2	2	7	9	11	14	16	18	21	23

← Capacity

if (w[i]>c)
  T[i, c] = T[i-1,c]
else
  $T[i, c] = \max(T[i-1, c], v[i] + T[i, c-w[i]])$

KNAPSACK SIMULATION - UNBOUNDED

How to find the items that are in the bag?

i	v_i	w_i	0	1	2	3	4	5	6	7	8	9	10	← Capacity
0	0	0	0	0	0	0	0	0	0	0	0	0	0	Not Included
1	2	1	0	2	4	6	8	10	12	14	16	18	20	**Included – 1 times**
2	1	2	0	2	4	6	8	10	12	14	16	18	20	Not Included
3	7	3	0	2	4	7	9	11	14	16	18	21	23	**Included – 3 times**
4	6	4	0	2	2	7	9	11	14	16	18	21	23	Not Included
5	12	6	0	2	2	7	9	11	14	16	18	21	23	Not Included

Sack = {3, 3, 3, 1}
As we have reached the 0^{th} row, we are done with item selection. So, the sack contains one quantity of item#1 **and 3 quantity of item#3** with value = 1*2+3*7 = 23

Knapsack simulation - Unbounded

```
for (int i = 1; i <= n; i++)
    {
        for (int j = 0; j <= cpt; j++)
        {
            if(wt[i-1]<=j)
                dp[i][j]=max((val[i-1]+dp[i][j - wt[i - 1]]),dp[i-1][j]);
            else
                dp[i][j] = dp[i-1][j];

            printf("%d    ",dp[i][j]);
        }
        printf("\n");
    }
```

Reference

- Chapter 15 (15.1 and 15.3) (Cormen)
- http://www.shafaetsplanet.com/?p=3638
- https://www.javatpoint.com/0-1-knapsack-problem
- https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

Lab Test: Up to Lab-7 (Greedy)

Section B2:

Section B1:

Section A2:

Thanks to All