Unit 4- Dynamic Programming

Srinidhi H,
Assistant Professor,
Dept of CSE, MSRIT

Topics

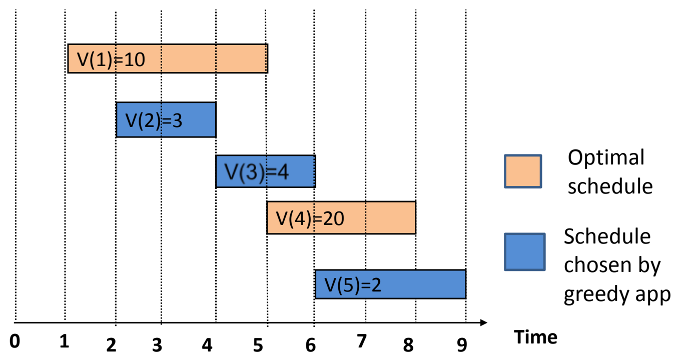
1. Weighted interval scheduling

a. Problem and Goal

We have *n* requests labeled 1, . . . , *n*, with each request *i* specifying a start time s_i and a finish time f_i. Each interval *i* now also has a *value*, or *weight* v_i. Two intervals are *compatible* if they do not overlap.
The **goal** of our current problem is to select a subset $S \subseteq \{1, . . . , n\}$ of mutually compatible intervals, so as to maximize the sum of the values of the selected intervals, $\Sigma_{i \in S} v_i$.

The greedy algorithm discussed in the previous lectures cannot be used to satisfy the goal of the problem as shown below with an example.

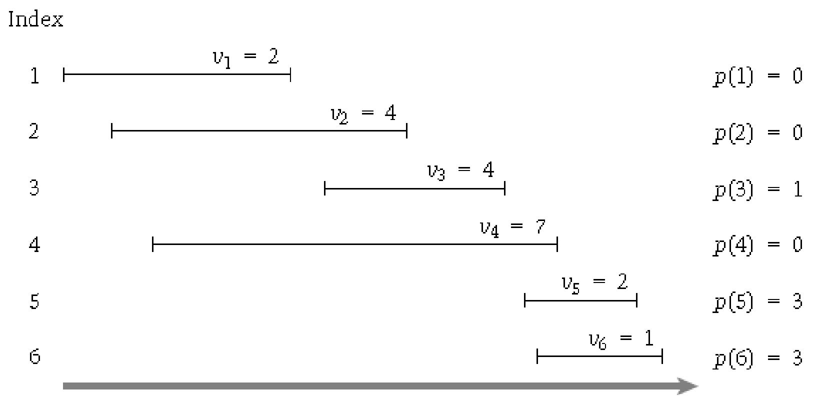
Greedy does not work



Greedy approach takes job 2, 3 and 5 as best schedule and makes profit of 7. While optimal schedule is job 1 and job4 making profit of 30 (10+20). Hence greedy will not work

b. Compute predecessor of a request

We define *p(j)*, for an interval *j*, to be the largest index *i* < *j* such that intervals *i* and *j* are disjoint i.e. interval i doesn't overlap with j .



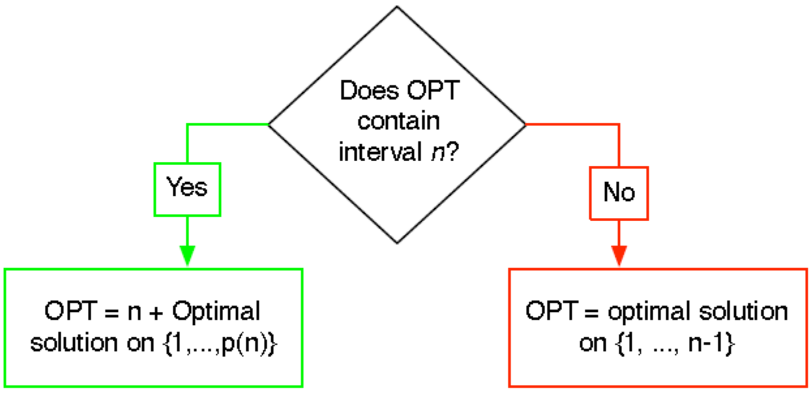
c. Designing the Algorithm for Weighted interval scheduling based on Dynamic Programming approach.

Let us consider there is an optimal schedule OPT for the given set of requests and their values.

Let ‘n’ be the last interval in the given set of requests. If we want to find whether an interval ‘n’ belongs to OPT there are **2 choices** left. They are:

i) Interval ‘n’ belongs to the OPT. If it belongs then we have to continue for further intervals that are compatible with interval ‘n’ using p(n) calculated for the nth interval.

ii) If it does not belong then, consider the set of intervals from set of {1,…,n-1}.


Does OPT contain interval n?
Yes
No
OPT = n + Optimal solution on {1,...,p(n)}
OPT = optimal solution on {1, ..., n-1}

The **recurrence relation** of a request ‘j’ in the optimal set of requests can be denoted as OPT(j) given by,

$$OPT(j) = \max \begin{cases} v_j + OPT(p(j)) & j \text{ in OPT solution} \\ OPT(j-1) & j \text{ not in solution} \\ 0 & j = 0 \end{cases}$$

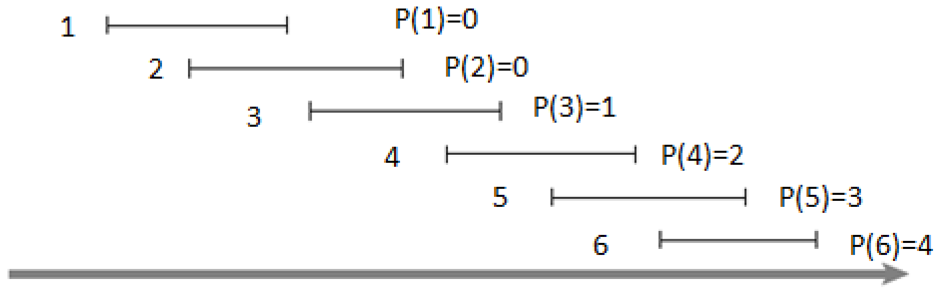
d. Algorithm implementing recurrence relation

//Purpose: To find the optimal weights for weighted interval scheduling problem.
//Input: 1….n requests each having start time ‘s’ and finish time ‘f’ and weight ‘v’
//**Output**: Optimal weight of the given set of ‘n’ intervals.

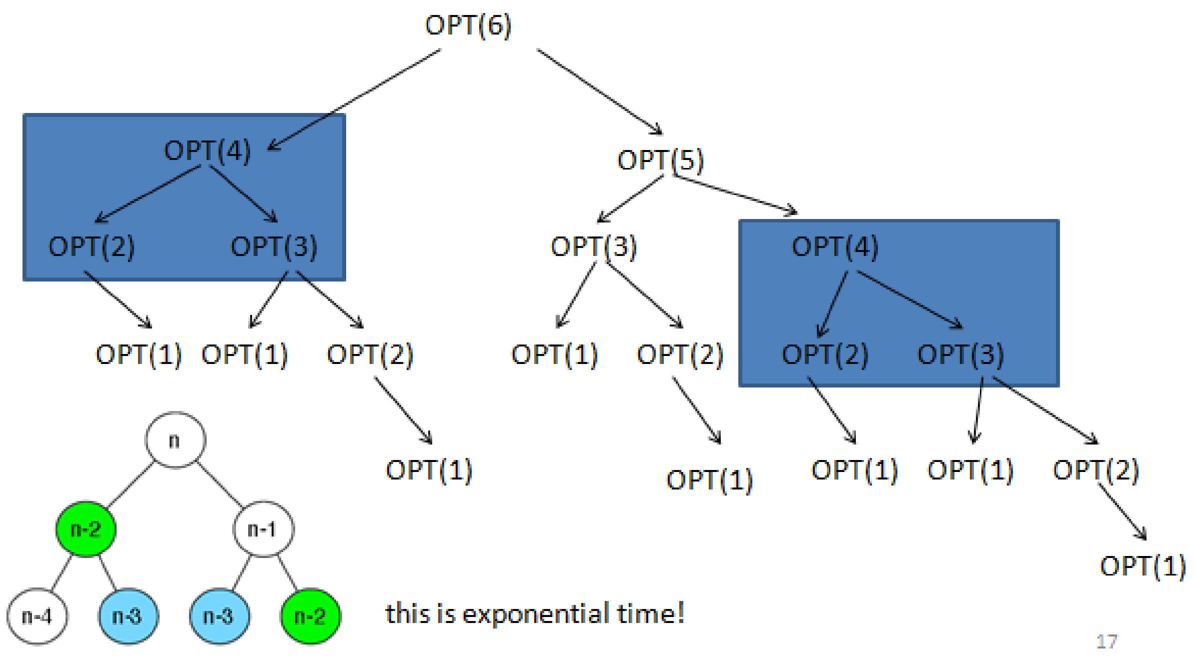
```
Sort the intervals according to their finish times
f1≤f2≤f3…..≤fn
Compute p(1), p(2)…….,p(n)
j=nth interval
Compute-Opt (j):
      If j = 0
            Return 0
      Else
            Return max( v[j] + Compute-Opt (p[j]),Compute-Opt
(j-1))
```

The algorithm takes an exponential amount of time.

Example



A recursive tree for the above set of intervals is as shown below.



e. Memoization

The exponential time of the recursive procedure for weighted interval scheduling can be reduced using Memoization technique.

We can store the value of Compute-Opt in a globally accessible place the first time we compute it and then simply use this precomputed value in place of all future recursive calls.This technique of saving values that have already been computed is referred to as *memoization*.

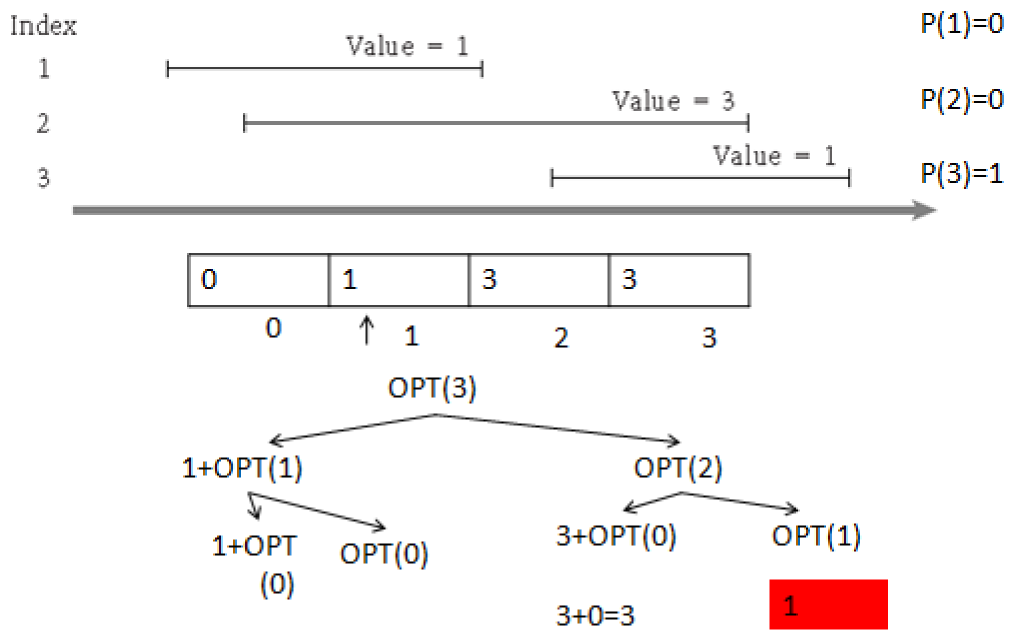
We implement the above strategy in the more "intelligent" procedure MCompute-Opt. This procedure will make use of an array $M[0 \ldots n]$; $M[j]$ will start with the value "empty," but will hold the value of Compute-Opt*(j)* as soon as it is first determined. To determine OPT*(n)*, we invoke M-Compute- Opt*(n)*.

f. Memoization algorithm

//**Purpose**: To find the optimal weights for weighted interval scheduling problem.
//**Input:** 1….n requests each having start time 's' and finish time 'f' and weight 'v'
//**Output**: Optimal weight of the given set of 'n' intervals.

```
M-Compute-Opt(j)
    If j = 0 then
        Return 0
    Else if M[j] is not empty then
        Return M[j]
    Else
        Define M[j] = max($v_j$+M-Compute-Opt(p(j)),
                              M-Compute-Opt(j − 1))
        Return M[j]
    Endif
```

Example



g. Find the intervals that belong to the output set

Given the global array M we can find the intervals that belong to the output set using the following relation and algorithm.

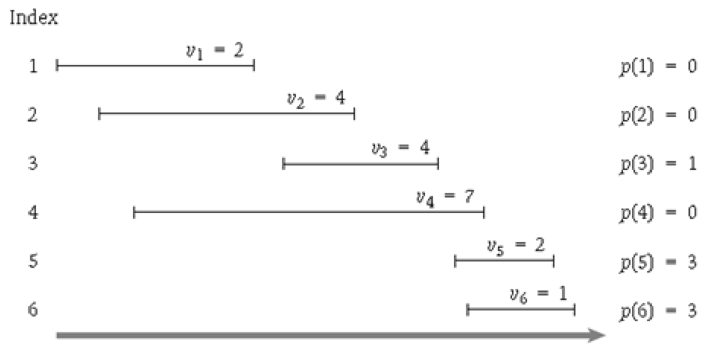
Relation: v_j + OPT$(p(j)) \geq$ OPT$(j - 1)$.

Algorithm:

//**Purpose**: To find the intervals belonging to the output set for weighted interval scheduling problem.
//**Input:** 1….n requests each having start time ‘s’ and finish time ‘f’ and weight ‘v’
//**Output**: The intervals belonging to the output set for weighted interval scheduling problem.

```
Find-Solution(j)
{
   if (j > 0)
    if (v_j + M[p(j)] >= M[j-1])
      print j
      Find-Solution(p(j))
   else
      Find-Solution(j-1)
}
```

Example: (Refer to the slide no 34.)



N	Global array M	$v_j + M[p(j)]$	M[j-1]	Output set containing the intervals
5	0 | 2 | 4 | 6 | 7 | 8	2+6=8	7	{5} Find_solution(3)
3	0 | 2 | 4 | 6 | 7 | 8	4+2=6	4	{5,3} Find_solution(1)
1	0 | 2 | 4 | 6 | 7 | 8	2+0=2	0	{5,3,1} Find_solution(0)

h. Iterative procedure for Weighted interval scheduling

We can also iterative procedure to calculate the global array ‘M’ and use the find-solution method to find the max weight possible and intervals in the output set respectively.

Iterative algorithm
//**Purpose**: To find the optimal weights for weighted interval scheduling problem.
//**Input:** 1….n requests each having start time ‘s’ and finish time ‘f’ and weight ‘v’

//**Output**: Optimal weight of the given set of ‘n’ intervals.

```
Iterative-Compute-Opt
$M[0] = 0$
For $j = 1, 2, . . . , n$
      $M[j] = \max(v_j + M[p(j)], M[j - 1])$
Endfor
```

Example: Refer to the slide 26-32.

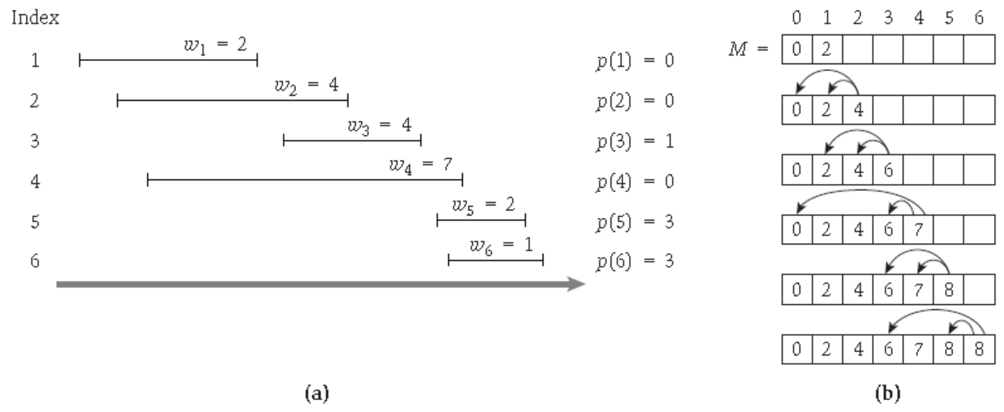


Figure 6.5 Part (b) shows the iterations of `Iterative-Compute-Opt` on the sample instance of Weighted Interval Scheduling depicted in part (a).

i. Analysis

1. The time complexity of the memorization algorithm for Weighted interval scheduling is O(n) *(assuming the input intervals are sorted by their finish times).*

Proof: The time spent in a single call to M-Compute-Opt is *O(*1*)*, excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls ever issued to M-Compute-Opt. Since the implementation itself gives no explicit upper bound on this number of calls, we try to find a bound by looking for a good measure of “progress.”

The most useful progress measure here is the number of entries in *M* that are not “empty.” Initially this number is 0; but each time the procedure invokes the recurrence, issuing two recursive calls to M-Compute-Opt, it fills in a new entry, and hence increases the number of filled-in entries by 1. Since *M* has only *n* + 1entries, it follows that there can be at most *O(n)* calls to M-Compute- Opt, and hence the running time of M-Compute-Opt*(n)* is *O(n)*, as desired.

2. The time complexity of the iterative algorithm for Weighted interval scheduling is O(n) *(assuming the input intervals are sorted by their finish times).*
3. *Given the array M of the optimal values of the sub-problems,* Find- Solution *returns an optimal solution in O(n) time.*
4. Sorting intervals according to finish time O(n logn).
5. To compute p(j) for each request j it takes O(logn).

Hence the overall time complexity of the Weighted interval scheduling problem is O(nlogn)

2. **Principles of Dynamic programming**

(i) There are only a polynomial number of subproblems.
(ii) The solution to the original problem can be easily computed from the solutions to the subproblems. (For example, the original problem may actually *be* one of the subproblems.)
(iii) There is a natural ordering on subproblems from "smallest" to "largest," together with an easy-to-compute recurrence (as in weighted interval scheduling) that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.

3. **Subset-sum problem**

a. **Problem and Goal:** We are given *n* items {1, . . . , *n*}, and each has a given nonnegative weight *wi* (for *i* = 1, . . . , *n*). We are also given a bound *W*. We would like to select a subset *S* of the items so that $\Sigma_{i\in S}\, w_i \leq W$ and, subject to this restriction, $\Sigma_{i\in S}\, w_i$ is as large as possible. We will call this the ***Subset Sum Problem*.**

b. **Designing the algorithm**

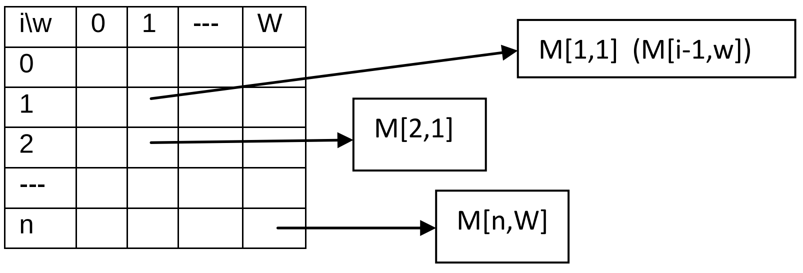
Let us consider and optimal solution OPT(n, W) = **max** profit subset of items **1, …, n** with weight limit ***W***. There can be 2 cases if we consider an nth item as follows

– **Case 1:** ***OPT*** does not select item ***n i***.e. n∉OPT
  - ***OPT*** selects **best** of **{ 1, 2, …, n-1 }** using weight limit ***W***
– **Case 2:** **OPT** selects item ***n***. (***i***.e. n∈OPT)
  - **new weight limit** = **W – w_n**
  - **OPT** selects **best** of **{ 1, 2, …, n–1 }** using **this new weight limit**

So, the recurrence relation can be obtained as below considering the above cases.

$$OPT(n,W) = \begin{cases} 0 & \text{if } n = 0 \\ OPT(n-1,W) & \text{if } w_n > W \\ \max\{OPT(n-1,W), w_n + OPT(n-1, W - w_n)\} & \text{otherwise} \end{cases}$$

We need two dimensional table of (n,W) to fill in the values in order to get the maximum subset of weights of the items. It is as shown below.



i\w	0	1	---	W
0				
1				
2				
---				
n				

c. Algorithm for Subset-Sum

//Purpose: To find the maximum weight from the given n items and their weights w_i
//Input: A set of items 1,2,…….n, with, $w_1,\ldots,w_N$, capacity W
//Output: Max weight M[n,W]

```
for w = 0 to W
   M[0, w] = 0
for i = 0 to n
   M[i, 0] = 0
for i = 1 to n            // n items
   for w = 1 to W         // weights from 1 to max cap W
      if ($w_i$ > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], $w_i$ + M[i-1, w-$w_i$ ]}
   endfor
endfor
return M[n, W]
```

d. Example for a subset-sum problem.

Items	Weights
1	2
2	2
3	3

W=5

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	2	2	2	2
2	0	0	2	2	4	4
3	0	0	2	3	4	5

i	w_i	w	M[i,w]
1	2	1	= M[0,1]=0
		2	=Max(M[0,2],2+M[0,0]) =Max(0,2) =2
		3	=Max(M[0,3],2+M[0,1]) =Max(0,2) =2
		4	=Max(M[0,4],2+M[0,2]) =Max(0,2) =2
		5	=Max(M[0,5],2+M[0,3]) =Max(0,2) =2
2	2	1	= M[1,1]=0

		2	=Max(M[1,2],2+M[1,0]) =Max(2,2) =2
		3	=Max(M[1,3],2+M[1,1]) =Max(2,2) =2
		4	=Max(M[1,4],2+M[1,2]) =Max(2,4) =4
		5	=Max(M[1,5],2+M[1,3]) =Max(4,4) =4
3	3	1	= M[2,1]=0
		2	=M[2,2]=2
		3	=Max(M[2,3],3+M[2,0]) =Max(2,3) =3
		4	=Max(M[2,4],3+M[2,1]) =Max(4,2) =2
		5	=Max(M[2,5],3+M[2,2]) =Max(4,5) =5

4. Knapsack problem

a. **Problem and Goal:** We are given n items $\{1, . . . , n\}$, and each has a given nonnegative weight *wi* (for $i = 1, . . . , n$). We are also given a bound of Knapsack capacity W. We would like to select a subset S of the items so that $\Sigma_{i\in S}\, w_i \leq W$ and, subject to this restriction, $\Sigma_{i\in S}\, v_i$ is as large as possible. We will call this the *Knapsack Problem*.

b. Designing the algorithm

The approach for designing the algorithm is same as subset-sum problem but we have consider value here and not the weight to achieve maximum profit.

The recurrence relation is as shown below for the knapsack problem,

$$OPT(n,W) = \begin{cases} 0 & \text{if } \ n = 0 \\ OPT(n-1,W) & \text{if } \ w_n > W \\ \max\{OPT(n-1,W), v_n + OPT(n-1, W - w_n)\} & \text{otherwise} \end{cases}$$

c. Algorithm for Knapsack

//Purpose: To find the maximum value or profit possible from the given n items and their weights w_i
//Input: A set of items 1,2,…….n, with, $w_1,\dots,w_N,$ and values $v_1,v_2\dots\dots,v_n$ with knapsack capacity W
//Output: Max weight M[n,W]

```
for w = 0 to W
   M[0, w] = 0
for i = 0 to n
   M[i, 0] = 0
for i = 1 to n             // n items
   for w = 1 to W          // weights from 1 to max cap W
      if ($w_i$ > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], $v_i$ + M[i-1, w-$w_i$ ]}
   endfor
endfor
return M[n, W]
```

d. Example for a Knapsack problem.

Items	Weights	Values
1	2	20
2	2	10
3	3	30

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

Knapsack capacity W=5

I	w_i	w	M[i,w]
1	2	1	= M[0,1]=0
		2	=Max(M[0,2],20+M[0,0]) =Max(0,20) =20
		3	=Max(M[0,3],20+M[0,1]) =Max(0,20) =20
		4	=Max(M[0,4],20+M[0,2]) =Max(0,20) =20
		5	=Max(M[0,5],20+M[0,3]) =Max(0,20) =20

2	2	1	= M[1,1]=0
		2	=Max(M[1,2],10+M[1,0]) =Max(20,10) =20
		3	=Max(M[1,3],10+M[1,1]) =Max(20,10) =10
		4	=Max(M[1,4],10+M[1,2]) =Max(20,30) =30
		5	=Max(M[1,5],10+M[1,3]) =Max(20,30) =30
3	3	1	= M[2,1]=0
		2	=M[2,2]=20
		3	=Max(M[2,3],30+M[2,0]) =Max(20,30) =30
		4	=Max(M[2,4],30+M[2,1]) =Max(30,30) =30
		5	=Max(M[2,5],30+M[2,2]) =Max(30,50) =50

e. Find Items in the Knapsack

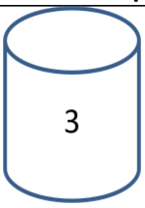
In order to find the items that belong to the knapsack, we use the global array M[n,W] computed and find the items using the following algorithm.

Algorithm: Find_items_in_knapsack()
// **Input:** Global array M[n,W] giving the maximum value achievable for 'n' set of items and knapsack capacity 'W'.
//**Output:** The items 'i' that belong to the Knapsack.

```
i = n , k = W
while  i, k > 0
      if $M[i, k] \neq M[i-1, k]$ then
            mark the $i^{th}$ item as in the knapsack
            $i = i-1$, $k = k-w_i$
      else
            $i = i-1$
```

f. Example to find the items in the knapsack

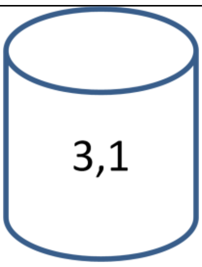
Items	Weights	Values
1	2	20
2	2	10
3	3	30

i	k	Global array M[n,W]	Items in the knapsack
3	5	(array below)	

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	20	20	20	20
2	0	0	20	20	30	***<u>30</u>***
3	0	0	20	30	30	***<u>50</u>***

i	k	Global array M[n,W]	Items in the knapsack
2	3	(array below)	

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	***20***	20	20	20
2	0	0	***20***	20	30	30
3	0	0	20	30	30	50

i	k	Global array M[n,W]	Items in the knapsack
1	2	(array below)	

i / w	0	1	2	3	4	5
0	0	0	***0***	0	0	0
1	0	0	***20***	20	20	20
2	0	0	20	20	30	30
3	0	0	20	30	30	50

i	k	Global array M[n,W]	Items in the knapsack
0	0		**STOP**

g. Analysis

The running time complexity of subset-sum problem and Knapsack problem O(nW).