Dynamic Programming - II

The Coin-row problem

The Coin row problem

- There is a row of n coins whose values are some positive integers c_1, c_2 ,...,c_n, not necessarily distinct.
- The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.
  - e.g. consider a coin row as 5, 1, 2, 10, 6, 2
  - Then, the selection of the coins leading to maximum amount of money subject to the constraint given is as follows:

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

The Coin row problem: Designing the recurrence

- Let F (n) be the maximum amount that can be picked up from the row of n coins and the c_n i.e. c_1, c_2 ,...,c_n be the values(the denominations) of the coins in a row.
- To derive a recurrence for F(n), we partition all the allowed coin selections into two groups:
  1. Group 1: those that include the last coin
     - the largest amount we can get from the Group 1 is equal to $c_n + F(n-2)$ i.e. the value of the n^{th} coin plus the maximum amount we can pick up from the first n−2 coins.
  2. Group 2: those without it
     - the maximum amount we can get from the second group is equal to F(n−1) i.e. when we do not select the coin n.
- Thus, we have the following recurrence subject to the obvious initial conditions:

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1,$$
$$F(0) = 0, \qquad F(1) = c_1.$$

The Coin row problem: Algorithm

ALGORITHM CoinRow(C[1..n])

/*Applies the recurrence designed bottom up to find the maximum amount of money that can be picked up from a coin row without picking two adjacent coins. */

Input: Array C[1..n] of positive integers indicating the coin values

Output: The maximum amount of money that can be picked up

```
1. F[0]←0;
2. F[1]←C[1]
3. for i←2 to n
4. do
5.    F[i]←max(C[i] + F[i-2], F[i-1])
6.    return F[n]
```

The Coin row problem: Dry run on an input

- Applying the algorithm on a coin row as 5, 1, 2, 10, 6, 2
- Note that we have to backtrace to compute every optima
  - i.e. to find the coins with the maximum total value found, we need to backtrace the computations to see which of the two possibilities — c_n + F(n−2) OR F(n−1) - produced the maxima in formula
- What is the complexity of the algorithm?
  - takes Θ(n) time and Θ(n) space.

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

The Coin row problem: Tutorial Problem

- Applying the algorithm on a coin row as 7, 2, 1, 12, 5 of the coin-row problem. Draw the table as shown here for this instance.

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Tutorial Exercise1

- Suppose you are managing a consulting team of expert computer hackers and each week you have to choose a job for them to undertake. Now, as per the company policy, the set of possible jobs is divided into two categories: (1) low-stress (e.g. setting up a website for a class at the school) OR (2) high-stress (e.g. protecting nation's most valuable secrets).
- The basic question each week is whether to take on a low-stress job OR a high-stress one. You as a consultant are supposed to help/assign in selection of the jobs.
- The nature of the job is related to the emoluments: if one takes a low-stress job, one gets a revenue of $l_i>0$ rupees; whereas if one takes a high-stress job, one gets a revenue of $h_i>0$ rupees.

….continued

Tutorial Exercise1…

- The job selection, however has some conditions:
  - in order for a team to take on a high-stress job in week *i*, it is required that they do no job in week *(i-1)* – of either type i.e. they need one full week's time in preparing for the crushing stress level.
  - On the other hand, it is permissible for them to do a low-stress job in week *i*, even if they have done a job (of either type) in week *(i-1)*. However, for the first week it is okay to choose a high-stress job.
- So, given a sequence of *n* weeks, a plan is specified by a choice of "low-stress", "high-stress" OR "none" for each of these n weeks, adhering to the above conditions.

….continued

Tutorial Exercise1…

- **The Problem:** Given sets of values $l_1, l_2, l_3, l_4 \ldots . l_n$ and $h_1, h_2, h_3, h_4 \ldots . h_n$, find a plan of maximum value. Such plan would be the optimal plan.
- e.g. consider the following input. What is the maximum value in this input with appropriate job selection ?

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

- The value of this plan would be 70.

Tutorial Exercise1...: An initial attempt

Algorithm FirstAttemptStressJob

```
1. for i = 1 to n
2.    if h_{i+1} > l_i+l_{i+1} then
3.       Output "Choose no job in week i"
4.       Output "Choose a high-stress job in week i + 1"
5.       Continue with iteration i+2
6. Else
7.    Output "Choose a low-stress job in week i"
8.    Continue with iteration i+1
9. Endif
10.End
```

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

Tutorial Exercise1: Dynamic Progg Solution outline

Coin Changing Problem

The Coin changing problem

- to devise an algorithm for paying a given amount to a customer using the smallest possible number of coins.
- Two variations
  - Given a fixed denomination, unlimited supply of coins
  - Given different series of denominations, but limited supply.
- Two approaches
  - Greedy approach works ?
  - Dynamic programming approach is required

Algorithm MakeChange(SetofCoins n)

```
// Algorithm MakeChange(SetofCoins n) /* Make change for n units using the
   least possible number of coins.
1. const C = {100, 25, 10, 5}
2. S ← ∅ {S - the set to hold the solution}
3. s ← 0 {s - the sum of items in S}
4. while s ≠ n do
5.      x ← the largest item in C such that s + x ≤ n
6.     if there is no such item then
7.                        return "no solution found"
8.      S ← S U { a coin value x}
9.      s ← s + x
10. return S.
```

- Which design approach does this algorithm use. Why ?
- Limitations ?

The Dynamic Programming solution: Approach1

- Approach to use ?
- Let F(n) be the minimum number of coins whose values add up to n.
- It is convenient to define F(0) = 0.
- How to obtain the amount n, using the coin of denomination dj ?

The Dynamic Programming solution: Approach1

- Again two choices here :
  1. Either use the denomination d_j and then the amount n can only be obtained by adding one coin of denomination d_j to the amount $n- d_j$ for j = 1, 2,...,m such that $n \geq d_j$
  2. OR we may not use d_j but whatever amount we had arrived at before considering d_j could be giving the minimum number of coins.
- Let us first try to understand this….
- Denominations available are say 1, 3 and 4 and the maximum amount is 6.

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

- So, minimum of 1 and 2 above must be used……

The Coin Changing Problem : Designing the recurrence

- Thus, given that F(n) be the minimum number of coins whose values add up to n.
  - we can consider all such denominations and select the one minimizing F(n−dj) + 1
  - Since 1 is a constant, we can, of course, find the smallest F (n−dj) first and then add 1 to it.
  - Hence, we have the following recurrence

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$
$$F(0) = 0.$$

  - We can compute F(n) by filling a one-row table left to right in the manner similar to the way it was done above for the coin-row problem, but computing a table entry here requires finding the minimum of up to m numbers.

The Coin-changing problem: Approach#1

- Denominations available are say 1, 3 and 4 and the maximum amount is 6.

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$
$$F(0) = 0.$$

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

The CC problem: Approach#1: Another illustration

- Let us assume now, the denominations available are say 1, 3 and 4 and 5 and again the maximum amount is 10.
- Apply the recurrence and prepare the table as before.

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

The CC problem: Approach#1: Another illustration

- Let us assume now, the denominations available are say 1, 3 and 4 and 5 and again the maximum amount is 10.
- Apply the recurrence and prepare the table as before.

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0										

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1									

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2								

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1							

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	1						

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	1					

The CC problem: Approach#1: Another illustration

- Let us assume now, the denominations available are say 1, 3 and 4 and 5 and again the maximum amount is 10.
- Apply the recurrence and prepare the table as before.

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	1					

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	1	2				

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	5	2	2			

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	5	2	2	2		

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	1	5	2	7	2	2	

n=	0	1	2	3	4	5	6	7	8	9	10
F(j)=	0	1	2	1	4	1	2	7	8	2	2

The Coin-changing problem: Approach#1: Algorithm

ALGORITHM ChangeMaking(D[1..m], n)

/* Applies dynamic programming to find the minimum number of coins of denominations $d_1 < d_2 < \ldots < d_m$ where $d_1 = 1$ that adds up to a given amount n */

Input: Positive integer n and array D[1..m] of increasing positive integers indicating the coin denominations where D[1] = 1

Output: The minimum number of coins F(n) that add up to n

D[3]	1	3	4

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

```
1. F[0] = 0
2. for LocalAmount=1 to n
3.     do  temp=∞;  j=1
4.     while j≤m and LocalAmount≥D[j]
5.           do temp = min(F[LocalAmount - D[j]], temp)
6.           j←j + 1
7.     F[LocalAmount]←temp + 1
8. return F[n]
```

The Coin-changing problem: Approach#2

- Setup a table c[1…n, 0..N]
  - rows: denominations available viz. **1 ≤ i ≤ n**
  - columns: : amount to be paid is j viz. **0 ≤ j ≤ N**

	Amount	0	1	2	3	4	5	6	7	8
0	$d_0=0$	0	0	0	0	0	0	0	0	0
1	$d_1=1$	0	1	2	3	4	5	6	7	8
2	$d_2=4$	0								
3	$d_3=6$	0								

The Coin-changing problem: Approach#2…

- Setup a table c[1…n, 0..N]
  - rows: denominations available viz. **1 ≤ i ≤ n**
  - columns: : amount to be paid is j viz. **0 ≤ j ≤ N**
- let **c[n, N]** - the minimum number of coins of denominations **n** required for the amount **N**
- **c[i,0]** – has to be zero for every **i**
- To pay an amount **j** using coins of denomination **1 to i :**
  - either do not use any coins of denomination **i**
  - OR use the current denomination d_i, but then use the appropriate number of coins by looking up the amount that shows the difference between the current amount j and the d_i
- Then, the objective is to minimize the number of coins used and so choose minimum of the two.

The Coin-changing problem: Approach#2:Designing the recurrence

- Then how could the recurrence be specified ?

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

- e.g. if we have to pay an amount of 8 here, which coins would be selected ?

	Amount	0	1	2	3	4	5	6	7	8
0	d_0=0	0	0	0	0	0	0	0	0	0
1	d_1=1	0	1	2	3	4	5	6	7	8
2	d_2=4	0								
3	d_3=6	0								

- Note that the table is filled using the recurrence as above i.e.

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

The Coin-changing problem: Approach#2:Designing the recurrence…

- Now, let us fill the third row….
- Specifically, consider making up the amount 8, an interesting case.

	Amount	0	1	2	3	4	5	6	7	8
0	$d_0=0$	0	0	0	0	0	0	0	0	0
1	$d_1=1$	0	1	2	3	4	5	6	7	8
2	$d_2=4$	0	1	2	3	1	2	3	4	2
3	$d_3=6$	0								

- Note that the table is filled using the recurrence as above i.e.

$$c[i, j] = \min \{1 + c[i, j-di], c[i-1, j]\}$$

The Coin-changing problem: Approach#2…

- The complete answer as shown ……using the recurrence

$$c[i, j] = \min \{1 + c[i, j-di], c[i-1, j]\}$$

	Amount	0	1	2	3	4	5	6	7	8
0	$d_0=0$	0	0	0	0	0	0	0	0	0
1	$d_1=1$	0	1	2	3	4	5	6	7	8
2	$d_2=4$	0	1	2	3	1	2	3	4	2
3	$d_3=6$	0	1	2	3	1	2	1	2	2

-

The Coin-changing problem: Approach#2 Algorithm

```
Algorithm coins(N)
// making change for N units using coinages from d[1..n]
1. d[1..n] = [1,4,6]
2. for i=1 to n
3.    for j=1 to N
4.     if i=1 && j < d[i] then c[i,j]= ∞
5.     elseif i=1 then c[i,j]= 1 + c[1,j-d[1]]
6.     elseif j < d[i] then c[i,j]=c[i-1, j]
7.     else min{c[i-1,j], 1 + c[1,j-d[1]]}
8. return c[n,N]
```

The Coin-changing problem: Approach#2: Another illustration…

- Populate the following table using the denominations now, as shown and then consider paying an amount of 8 and 10 here. Which coins would be selected in each cases ?

Amount	0	1	2	3	4	5	6	7	8	9	10
d_0=0	0	0	0	0	0	0	0	0	0	0	0
d_1=1	0	1	2	3	4	5	6	7	8	8	8
d_2=4	0										
d_3=5	0										
d_3=6	0										

  - Note that the table is filled using

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

The Coin-changing problem: Approach#2: Another illustration…

- Populate the following table using the denominations now, as shown and then consider paying an amount of 8 and 10 here. Which coins would be selected in each cases ?

Amount	0	1	2	3	4	5	6	7	8	9	10
d_0=0	0	0	0	0	0	0	0	0	0	0	0
d_1=1	0	1	2	3	4	5	6	7	8	8	8
d_2=4	0	1	2	3	1	2	3	4	2	2	2

  - Note that the table is filled using

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

The Coin-changing problem: Approach#2: Another illustration…

- Populate the following table using the denominations now, as shown and then consider paying an amount of 8 and 10 here. Which coins would be selected in each cases ?

Amount	0	1	2	3	4	5	6	7	8	9	10
d_0=0	0	0	0	0	0	0	0	0	0	0	0
d_1=1	0	1	2	3	4	5	6	7	8	8	8
d_2=4	0	1	2	3	1	2	3	4	2	2	2
d_3=5	0	1	2	3	1	1	2	3	2	2	2
d_3=6	0										

  - Note that the table is filled using

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

The Coin-changing problem: Approach#2: Another illustration…

- Populate the following table using the denominations now, as shown and then consider paying an amount of 8 and 10 here. Which coins would be selected in each cases ?

Amount	0	1	2	3	4	5	6	7	8	9	10
d_0=0	0	0	0	0	0	0	0	0	0	0	0
d_1=1	0	1	2	3	4	5	6	7	8	8	8
d_2=4	0	1	2	3	1	2	3	4	2	2	2
d_3=5	0	1	2	3	1	1	2	3	2	2	2
d_3=6	0	1	2	3	1	1	1	2	2	2	2

  - Note that the table is filled using

```
c[i, j] = min {1+ c[i, j-di], c[i-1, j]}
```

The Coin-collecting problem

The Coin Collecting Problem

- The Problem Description:
  - Several coins are placed in cells of an $n \times m$ board, with no more than one coin per cell.
  - A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell.
  - On each step, the robot can move either one cell to the right or one cell down from its current location.
  - When the robot visits a cell with a coin, it always picks up that coin.
- Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

The Coin Collecting Problem: Designing the recurrence

- Let F (i, j) be the largest number of coins the robot can collect and bring to the cell (i, j) in the i^{th} row and j^{th} column of the board.
- From which other cells can it reach this cell, under consideration ?
- It can reach this cell either from the adjacent cell (i−1, j) above it or from the adjacent cell (i, j−1) to the left of it.
- Which cell out of these two should be used ?
- The largest numbers of coins that can be brought to these cells are F (i−1,j) and F (i, j−1), respectively.
- Are there any other adjacent cells ?
  - Of course, there are no adjacent cells above the cells in the first row, and there are no adjacent cells to the left of the cells in the first column.

The Coin Collecting Problem: Designing the recurrence...

- For the adjacent cells above the cells in the first row, and adjacent cells to the left of the cells in the first column, we assume that F (i−1,j) and F (i, j−1) are equal to 0 for their nonexistent neighbors.
- Therefore, the largest number of coins the robot can bring to cell (i, j) is the maximum of these two numbers plus one possible coin at cell (i, j) itself.
- In other words, we have the following formula for

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \quad 1 \le j \le m$$
$$F(0, j) = 0 \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \text{ for } 1 \le i \le n,$$

here c_{ij} = 1 if there is a coin in cell (i, j), and c_{ij} = 0 otherwise.

- Using these formulas, we can fill in the n × m table of F (i, j) values either row by row or column by column, as is typical.

The Coin Collecting Problem: Algorithm

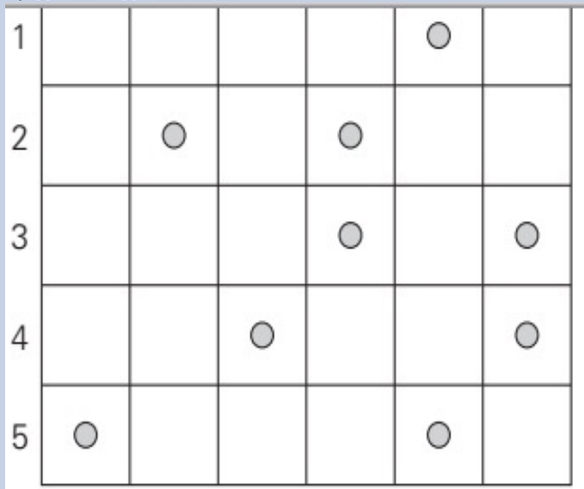
ALGORITHM RobotCoinCollection(C[1..n, 1..m])

- /* Applies dynamic programming to compute the largest number of coins a robot can collect on an n × m board by starting at (1, 1) and moving right and down from upper left to down right corner */
- Input: Matrix C[1..n, 1..m] whose elements are equal to 1 and 0 for cells with and without a coin, respectively
- Output: Largest number of coins the robot can bring to cell (n, m)

```
1.  F[1, 1]←C[1, 1];
2.  for j←2 to m
3.        do F[1, j ]←F[1, j−1] + C[1, j ]
4.  for i←2 to n
5.        do  F[i, 1]←F[i−1, 1] + C[i, 1]
6.        for j←2 to m
7.              do  F[i, j ]←max(F[i−1, j ], F[i, j−1]) + C[i, j ]
8.  return F[n, m]
```

The Coin Collecting Problem

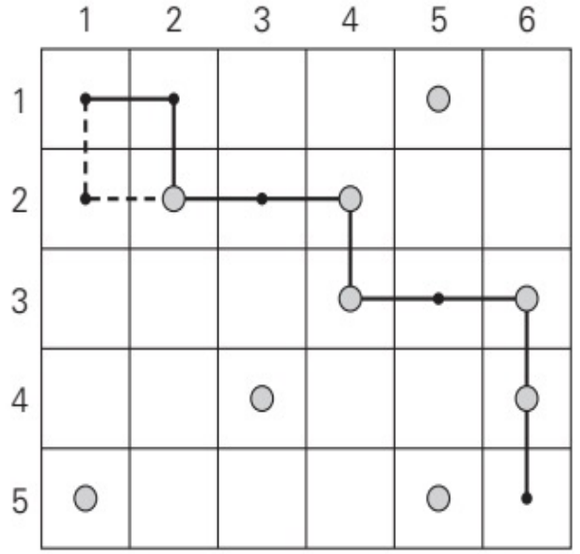
- Tracing the computations backward makes it possible to get an optimal path:
  - if F (i−1, j) > F (i, j−1), an optimal path to cell (i, j) must come down from the adjacent cell above it;
  - if F (i−1, j) < F (i, j−1), an optimal path to cell (i, j) must come from the adjacent cell on the left; and
  - if F (i−1, j) = F (i, j−1), it can reach cell (i, j) from either direction.
  - This yields two optimal paths as in 8.3c,



(a)

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

(b)



(c)

Characteristics of Dynamic Programming : A "Re-Review"

- Dynamic programming
  - is a technique that allows us to break up difficult problems into a sequence of easier subproblems
    - the subproblems are then evaluated by stages.
  - has the power to determine the optimal solution over say a one-year time horizon by breaking the problem into 12 smaller one-month horizon problems and to solve each of these optimally.
  - hence, it uses a multistage approach.
- Thus, dynamic programming differs from linear programming in two ways.
  - First, there is no algorithm (as in the simplex method) that can be programmed to solve all problems.
  - Second, linear programming is a method that gives single-stage (one time period) solutions.

Characteristics of Dynamic Programming : A "Re-Review"...

- Solving problems with dynamic programming involves four steps:
  I. Divide the original problem into subproblems called stages.
  II. Solve the last stage of the problem for all possible conditions or states.
  III. Working backward from the last stage, solve each intermediate stage. This is done by determining optimal policies from that stage to the end of the problem (last stage).
  IV. Obtain the optimal solution for the original problem by solving all stages sequentially.

0/1 Knapsack Problem

The 0/1 Knapsack problem

- Knapsack problem.
  - Given n objects and a "knapsack."
  - Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
  - Knapsack has capacity of W kilograms.
  - Goal: fill knapsack so as to maximize total value.

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

- Ex: Let W = 11, then
  - { 3, 4 } has value 40
- Greedy
  - repeatedly add item with maximum ratio v_i / w_i.
  - Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

The 0/1 Knapsack problem

- The greedy approach does not work
- consider an instance of **i** items, **1≤i ≤n**,
  - weights $w_1, w_2, w_3 \ldots w_n$,
  - values $v_1, v_2, v_3 \ldots . v_n$,
  - knapsack capacity **j** with **1≤j≤W,**
  - **V[i,j]** be the value of an optimal solution to this instance
    - i.e. the value of the most valuable subset of first i items that fit into the knapsack of capacity j.

Dynamic Programming: False Start

- Define
  - OPT = v(i) = max profit subset of items 1, …, i.
    - Case 1: OPT does not select item i.
      - OPT selects best of { 1, 2, …, i-1 }
    - Case 2: OPT selects item i.
      - accepting item i does not immediately imply that we will have to reject other items
      - without knowing what other items were selected before i, we don't even know if we have enough room for i
  - Conclusion. Need more sub-problems!

DP: Adding a New Variable

- Define
  - OPT = v(i, j) = max profit subset of items 1, …, i with weight limit j.
    - Case 1: OPT does not select item i.
      - OPT selects best of { 1, 2, …, i-1 } using weight limit j
    - Case 2: OPT selects item i.
      - new weight limit = $j - w_i$
      - OPT selects best of { 1, 2, …, i–1 } using this new weight limit
  - How to arrive at the recurrence to be used ?

DP: Adding a New Variable : Further Exp

- Two cases to consider
  - if the item **`i`** is not included in the above subset, the optimal solution is **`V[i-1,j]`**
  - if the item **`i`** is included in the above subsets, the optimal subset is made of
    - this item and
    - an optimal subset of first **`i-1`** items that fit into the knapsack of capacity **`j-w`**$_i$.
  - However, what is to be returned is the max of the two i.e.

    Max{(v_i + **`V[i-1,j-w`**$_i$**`]`**), **`v(i-1,j-w`**$_i$**`)`**}
- Therefore, the optimal solution among all feasible subsets of the first i items is the maximum of the above two values i.e.

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\left\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\right\} & \text{otherwise} \end{cases}$$

The 0/1 Knapsack problem : DP Table

- The table shows the values of V[i,j] where
  - i is the number of items that fit into the knapsack of capacity j

	0	$j-w_i$	j	W
0	0	0	0	0
i-1	0	V[i-1,j-W_i]	V[i-1, j]	
i	0		V[i, j]	
n	0			goal

The 0/1 Knapsack problem : DP Illustration

- Consider W = 5, n=5,v_i and w_i as shown below

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

The 0/1 Knapsack problem : DP Illustration

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

			1	2	3	4	5
		0	0	0	0	0	0
1	W_1=2, v_1=12	0					
2	W_2=1, v_2=10	0					
3	W_3=3, v_3=20	0					
4	W_4=2, v_4=15	0					

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

The 0/1 Knapsack problem : DP Illustration

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

			1	2	3	4	5
		0	0	0	0	0	0
1	W_1=2, v_1=12	0	0	12	12	12	12
2	W_2=1, v_2=10	0					
3	W_3=3, v_3=20	0					
4	W_4=2, v_4=15	0					

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

The 0/1 Knapsack problem : DP Illustration

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

			1	2	3	4	5
		0	0	0	0	0	0
1	W_1=2, v_1=12	0	0	12	12	12	12
2	W_2=1, v_2=10	0	10	12	22	22	22
3	W_3=3, v_3=20	0					
4	W_4=2, v_4=15	0					

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

The 0/1 Knapsack problem : DP Illustration

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

			1	2	3	4	5
		0	0	0	0	0	0
1	W_1=2, v_1=12	0	0	12	12	12	12
2	W_2=1, v_2=10	0	10	12	22	22	22
3	W_3=3, v_3=20	0	10	12	22	30	32
4	W_4=2, v_4=15	0					

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

The 0/1 Knapsack problem : DP Illustration

item	Weight	value
1	2	12
2	1	10
3	3	20
4	2	15

			1	2	3	4	5
		0	0	0	0	0	0
1	W_1=2, v_1=12	0	0	12	12	12	12
2	W_2=1, v_2=10	0	10	12	22	22	22
3	W_3=3, v_3=20	0	10	12	22	30	32
4	W_4=2, v_4=15	0	10	15	25	30	37

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

Knapsack Problem: Dynamic Progg. Algorithm

Algorithm DPKnapsack(w[1..n], v[1..n], W)

```
/* Input: Two nonnegative integer lists W[1…n] and V[1..n] indicating the weights and the values of the
   items being considered and a nonnegative integer W indicating the knapsack's capacity. Output: The value
   of an optimal feasible subset of the first i items. Note: Uses as global variables a table  V[0…n, 0…W]
   whose entries are initialized with -1's  except for row 0 and column 0 initialized with 0's */
```

```
1.  for j = 0 to W do
2.      V[0,j] == 0
3.  for i = 0 to n do
4.      V[i,0] = 0
5.  for i = 1 to n do
6.      for j = 1 to W do
7.          if (j - w_i) >=0
8.              V[i,j] = max{V[i-1,j], v[i] + V[i-1,j-w[i]]}
            else
9.              V[i,j] := V[i-1,j];
10. return V[n,W]
```

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \ \ if \ \ j - w_i < 0$$

Tutorial Exercise6

- Solve the following instance of the 0/1 knapsack problem using dynamic programming

Weight	1	2	3	2
Profit	20	15	25	12

- The capacity of the knapsack is W = 5
- Use the following recurrence

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

Tutorial Exercise6…solution ??

- Verify whether the following formulation is correct or not ?

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=1, v_1=20	1	0	20	20	20	20	20
W_2=2, v_2=15	2	0	20	20	35	35	35
w_3=3, v_3=25	3	0	20	20	35	45	45
w_4=2, v_4=12	4	0	20	20	35	45	47

Tutorial Exercise7

- Solve the following instance of the 0/1 knapsack problem using dynamic programming

Weight	1	2	5	6	7
Profit	1	6	18	22	28

- The capacity of the knapsack is W = 11
- Use the following recurrence, again

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

Tutorial Exercise7

OPT: { 4, 3 }
value = 22 + 18 = 40

W + 1 →

W = 11

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ w_1=1, v_1=1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ w_2=2, v_2=6 }	0	1	6	7	7	7	7	7	7	7	7	7
{ w_3=5, v_3=18 }	0	1	6	7	7	18	19	24	25	25	25	25
{ w_4=6, v_4=22 }	0	1	6	7	7	18	22	24	28	29	29	40
{ w_5=7, v_5=28 }	0	1	6	7	7	18	22	28	29	34	34	40

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i)\ +\ v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Overview and Comments

Reviewing the algorithm approaches seen

- Dynamic programming deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems.
- The direct top-down approach i.e. the recursive approach
  - to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once
  - hence is very inefficient (typically, exponential or worse).
- Let us try to understand this further…..

Problems/Subproblems call structure

- Consider a recursive algorithm A known for a problem P. Then,
  - the relationship that exists between the problem instance of P and its subproblems can be characterized by a DAG, whose
    - vertices
      - all input instances of the problem
    - directed edges i.e. I $\rightarrow$ J
      - all pairs such that when algorithm is invoked on instance I, it makes a recursive call to instance J.
- E.g. again draw the call-graph of fib-iters(6)

Problems/Subproblems call structure (contd)

- Basically a graph traversal procedure like DFS, but doesn't color the vertex.
  - Therefore, a memory less graph traversal
  - in a recursive call, it traverses every path in the DAG

- This means that…….
  - if using a recursive strategy to solve a problem with input instance P,
    - it is to be solved for all the vertices that are reachable from P, in the subproblem graph.
  - What if there are multiple paths to a subproblem in DAG?
    - the procedure will take exponential no of paths in DAG.

Summarizing.....(contd)

- What if we aren't able to determine the topological order (dependency graph)
  - from the knowledge of the subproblem?
    - invoke the DFS and note the reverse topological order at the finish/postorder time.
- A recursive algorithm explores every edge in the sense of making the recursive call while
  - the DFS skeleton only explores edges to undiscovered vertices and
  - it checks the other edges.

The Classic Dynamic Programming Approach

- The classic dynamic programming approach, on the other hand, works bottom up i.e.
  - it fills a table with solutions to all smaller subproblems, but each of them is solved only once.
- All the approaches in algorithms seen till now follow this style….
- Is there any limitation of the basic dynamic programming approach ?

Reviewing the algorithm approaches seen

- An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.
- Therefore, there is a need to combine the strengths of the top-down and bottom-up approaches.
- That is, to design a method that solves only subproblems that are necessary and does so only once.
- Such a method is based on using memory functions or the memoization based approach.

DP version of a recursive algorithm

- def: A dynamic programming version of a given recursive algorithm A, denoted as DP(A)
  - is a procedure that, given a top-level problem P to solve,
    - performs a DFS on the subproblem graph for A(P) and
    - as the intermediate solutions are found for the subproblems, they are recorded in the ADT dictionary viz. **soln**…..(i.e**. Dict soln**)
- a process called memo…ization.

Memoization

- Recall that
  - the operations on ADT **Dict** are **create, member, insert** and **store**.
- We shall devise an algorithm for this process viz algorithm memo-ization(A, DP(A)).
  - What is expected out of memo-ization(A, DP(A))?

- Memoization can be be applied
  - only to functions which are referentially transparent i.e.
    - those that do not have side effects

Algorithm Memoization

- Before any recursive call on subproblem **Q**, check the dictionary **soln** to see if the solution for **Q** has been stored.
  - If no solution has been stored, go ahead with the recursive call,
  - thereby treating **Q** as a white vertex and treating **P→Q** as a tree edge.
  - IF a solution has been stored for **Q**,
  - retrieve the stored solution and do not make the recursive call, thereby treating Q as a black vertex.
  - Just before returning the solution for **P**,
  - store it in the dictionary **soln**……effectively coloring the vertex **P** black.
- /* The subproblem graph must be acyclic, because vertices are not colored gray */

Fibonacci – Dynamic Programming version

```
Algorithm fibDP(soln, k)
1.  if (k<2)
2.    fib = k;
3.  else
4.    if (member(soln, k-1) == false)
5.       f1= fibDP(soln, k-1);
6.    else f1= retrieve(soln, k-1);
7.    if (member(soln, k-2) == false)
8.       f2= fibDP(soln, k-2);
9.    else f2= retrieve(soln, k-2);
10.   fib = f1 + f2
11. store(soln, k, fib)
12. return fib
```

Fibonacci – Dynamic Programming version

Algorithm DP(fib(n)):

```
1.    Dict soln = create(n);
3.          return fibDP(soln,n);
4.    fibDP(soln, k)
```

0/1 Knapsack revisited with Memoization.

What is the approach based on ?

- Now, for the knapsack problem let us first verify which are the values computed but not used i.e. the some of the solutions are not necessary to compute the solution to the given problem.

$$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i) + v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

Weight	1	2	3	2
Profit	20	15	25	12

The capacity of the knapsack is W = 5

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1=1, v_1=20$	1	0	20	20	20	20	20
$W_2=2, v_2=15$	2	0	20	20	35	35	35
$w_3=3, v_3=25$	3	0	20	20	35	45	45
$w_4=2, v_4=12$	4	0	20	20	35	45	47

Computations done but not used

- $$OPT = v(i,\ j) = \begin{cases} 0 & \text{if } i = 0 \\ v(i-1,\ j) & \text{if } w_i > j \\ \max\{v(i-1,\ j),\quad v(i) + v(i-1,\ j-w_i)\} & \text{otherwise} \end{cases}$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1=1$, $v_1=20$	1	0	20	20	20	20	20
$W_2=2$, $v_2=15$	2	0	20	20	35	35	35
$w_3=3$, $v_3=25$	3	0	20	20	35	45	45
$w_4=2$, $v_4=12$	4	0	20	20	35	45	47

Cell no, value and $(j-w_i)$	Formula used in the computations
V(4,5) = 47, (5-2) ≥ 0	max{V(3.5), 12+ V(3,3)} i.e. max{45, 12+35} i.e. 47
V(3,5) = 45, (5-3) ≥ 0	max{V(2,5), 25+ V(2,2)} i.e. max{35, 25+20} i.e. 45
V(3,3) = 35, (3-3) ≥ 0	max{V(2,3), 25+ V(2,0)} i.e. max{35, 25+0} i.e. 35
V(2,5) = 35, (5-2)) ≥ 0	max{V(1,5), 15+ V(1,3)} i.e. max{20, 15+20} i.e. 35
V(2,3) = 35, (3-2) ≥ 0	max{V(1,3), 15+ V(1,1)} i.e. max{20, 15+20} i.e. 35
V(2,2) = 20, (2-2)) ≥ 0	max{V(1,2), 15+ V(1,0)} i.e. max{20, 15+0} i.e. 20
V(1,3) = 20, (3-1) ≥ 0	max{V(0,3), 20+ V(0,0)}i.e. max{20, 20+0} i.e. 20
V(1,1) = 20, (1-1) ≥ 0	max{V(0,1), 20+ V(0,0)}i.e. max{0, 20+0} i.e. 20

Computations done but not used

- Thus, the computations that are not shown in red here, are the ones, are not necessary to compute the solution to the given problem.

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
$W_1=1$, $v_1=20$	1	0	20	20	20	20	20
$W_2=2$, $v_2=15$	2	0	20	20	35	35	35
$w_3=3$, $v_3=25$	3	0	20	20	35	45	45
$w_4=2$, $v_4=12$	4	0	20	20	35	45	47

Cell no, value and $(j-w_i)$	Formula used in the computations
V(4,5) = 47, (5-2) ≥ 0	max{V(3.5), 12+ V(3,3)} i.e. max{45, 12+35} i.e. 47
V(3,5) = 45, (5-3) ≥ 0	max{V(2,5), 25+ V(2,2)} i.e. max{35, 25+20} i.e. 45
V(3,3) = 35, (3-3) ≥ 0	max{V(2,3), 25+ V(2,0)} i.e. max{35, 25+0} i.e. 35
V(2,5) = 35, (5-2)) ≥ 0	max{V(1,5), 15+ V(1,3)} i.e. max{20, 15+20} i.e. 35
V(2,3) = 35, (3-2) ≥ 0	max{V(1,3), 15+ V(1,1)} i.e. max{20, 15+20} i.e. 35
V(2,2) = 20, (2-2)) ≥ 0	max{V(1,2), 15+ V(1,0)} i.e. max{20, 15+0} i.e. 20
V(1,3) = 20, (3-1) ≥ 0	max{V(0,3), 20+ V(0,0)}i.e. max{20, 20+0} i.e. 20
V(1,1) = 20, (1-1) ≥ 0	max{V(0,1), 20+ V(0,0)}i.e. max{0, 20+0} i.e. 20

Computations done but not used: Another illustration

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=2, v_1=12	1	0	0	12	12	12	12
W_2=1, v_2=10	2	0	10	12	22	22	22
w_3=3, v_3=20	3	0	10	12	22	30	32
w_4=2, v_4=15	4	0	10	15	25	30	37

Cell no, value and (j-w_i)	Formula used in the computations
V(4,5) = 37, (5-2) ≥ 0	max{V(3.5), 15+ V(3,3)} i.e. max{32, 15+22} i.e. 37
V(3,5) = 32, (5-3) ≥ 0	max{V(2,5), 20+ V(2,2)} i.e. max{22, 20+12} i.e. 32
V(3,3) = 22, (3-3) ≥ 0	max{V(2,3), 20+ V(2,0)} i.e. max{22, 20+0} i.e. 22
V(2,5) = 22, (5-1)) ≥ 0	max{V(1,5), 10+ V(1,3)} i.e. max{12, 10+12} i.e. 22
V(2,3) = 22, (3-1) ≥ 0	max{V(1,3), 10+ V(1,2)} i.e. max{12, 10+12} i.e. 22
V(2,2) = 12, (2-1)) ≥ 0	max{V(1,2), 10+ V(1,0)} i.e. max{12, 10+0} i.e. 12
V(1,3) = 12, (3-2) ≥ 0	max{V(0,3), 12+ V(0,0)}i.e. max{0, 12+0} i.e. 12
V(1,1) = 0, (1-2) !≥ 0	V(0,1) i.e = 0.

Computations done but not used: Another illustration

$$V[i,j] = \max\{V[i-1,j],\ vi + V[i-1, j-wi]\}, \quad if \quad j - w_i >= 0$$
$$= V[i-1,j] \quad if \quad j - w_i < 0$$

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=2, v_1=12	1	0	0	12	12	12	12
W_2=1, v_2=10	2	0	10	12	22	22	22
w_3=3, v_3=20	3	0	10	12	22	30	32
w_4=2, v_4=15	4	0	10	15	25	30	37

Cell no, value and $(j-w_i)$	Formula used in the computations
V(4,5) = 47, (5-2) ≥ 0	max{V(3.5), 12+ V(3,3)} i.e. max{45, 12+35} i.e. 47
V(3,5) = 45, (5-3) ≥ 0	max{V(2,5), 25+ V(2,2)} i.e. max{35, 25+20} i.e. 45
V(3,3) = 35, (3-3) ≥ 0	max{V(2,3), 25+ V(2,0)} i.e. max{35, 25+0} i.e. 35
V(2,5) = 35, (5-2)) ≥ 0	max{V(1,5), 15+ V(1,3)} i.e. max{20, 15+20} i.e. 35
V(2,3) = 35, (3-2) ≥ 0	max{V(1,3), 15+ V(1,1)} i.e. max{20, 15+20} i.e. 35
V(2,2) = 20, (2-2)) ≥ 0	max{V(1,2), 15+ V(1,0)} i.e. max{20, 15+0} i.e. 20
V(1,3) = 20, (3-1) ≥ 0	max{V(0,3), 20+ V(0,0)}i.e. max{20, 20+0} i.e. 20
V(1,1) = 20, (1-1) ≥ 0	max{V(0,1), 20+ V(0,0)}i.e. max{0, 20+0} i.e. 20

Computations done but not used: Another illustration

- Thus, the computations that are not shown in red here, are the ones, are not necessary to compute the solution to the given problem.

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=2, v_1=12	1	0	0	12	12	12	12
W_2=1, v_2=10	2	0	10	12	22	22	22
w_3=3, v_3=20	3	0	10	12	22	30	32
w_4=2, v_4=15	4	0	10	15	25	30	37

Cell no, value and $(j-w_i)$	Formula used in the computations
V(4,5) = 47, (5-2) ≥ 0	max{V(3.5), 12+ V(3,3)} i.e. max{45, 12+35} i.e. 47
V(3,5) = 45, (5-3) ≥ 0	max{V(2,5), 25+ V(2,2)} i.e. max{35, 25+20} i.e. 45
V(3,3) = 35, (3-3) ≥ 0	max{V(2,3), 25+ V(2,0)} i.e. max{35, 25+0} i.e. 35
V(2,5) = 35, (5-2)) ≥ 0	max{V(1,5), 15+ V(1,3)} i.e. max{20, 15+20} i.e. 35
V(2,3) = 35, (3-2) ≥ 0	max{V(1,3), 15+ V(1,1)} i.e. max{20, 15+20} i.e. 35
V(2,2) = 20, (2-2)) ≥ 0	max{V(1,2), 15+ V(1,0)} i.e. max{20, 15+0} i.e. 20
V(1,3) = 20, (3-1) ≥ 0	max{V(0,3), 20+ V(0,0)}i.e. max{20, 20+0} i.e. 20
V(1,1) = 20, (1-1) ≥ 0	max{V(0,1), 20+ V(0,0)}i.e. max{0, 20+0} i.e. 20

How to devise the memoization based approach?

- Maintain a table of the kind that would have been used by a bottom-up dynamic programming algorithm.
- Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated.
- Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first:
  - if this entry is not "null," it is simply retrieved from the table;
  - otherwise, it is computed by the recursive call whose result is then recorded in the table.
- The algorithm is as shown in the next diagram

DP approach with memoization algorithm

```
ALGORITHM   MKnapsack(i, j)
//Implementn of the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//  items being considered and a nonnegative integer j indicating the knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights [1...n], Values [1...n]
//and table V[0…n, 0…W] whose entries are initialized with -1's except for
//row 0 and column 0 initialized with 0's

1. if  V[i, j] < 0
2.       if j < Weights[i]
3.               values ← MKnapsack(i-1, j)
4.      else
5.               values ← max(MKnapsack(i-1, j),
6.                        Values[i] +  MKnapsack(i-1, j-Weights[i])
7.      V[i, j]← Value
8. return V[i, j]
```

The DP approach with memoization

	i & j	0	1	2	3	4	5
	0	0	0	0	0	0	0
w_1=2, v_1=12	1	0	0	12	12	12	12
w_2=1, v_2=10	2	0	-	12	22	-	22
w_3=3, v_3=20	3	0	-	-	22	-	32
w_4=2, v_4=15	4	0	-	-	-	-	37

- Compare this with the table that we computed earlier about the values computed but not used – again shown next.

	i and j	0	1	2	3	4	5
	0	0	0	0	0	0	0
W_1=2, v_1=12	1	0	0	12	12	12	12
W_2=1, v_2=10	2	0	10	12	22	22	22
w_3=3, v_3=20	3	0	10	12	22	30	32
w_4=2, v_4=15	4	0	10	15	25	30	37

Knapsack Problem: Running Time

- Running time. $\Theta(n W)$.
  - Not polynomial in input size!
  - "Pseudo-polynomial."
  - Decision version of Knapsack is NP-complete.

- Knapsack approximation algorithm.
  - There exists a polynomial algorithm that produces a feasible solution that has value within 0.01% of optimum.

Tutorial Exercise7

- Movers and Packers own and operate their Air Transport Service, which ships cargo by plane to most large cities in the India. The remaining capacity for one of the flights from New Delhi to Chennai is 10 tons. There are four different items that M&P can ship between New Delhi to Chennai. Each item has a weight in tons, a net profit in thousands of rupees, and a total number of that item that is available for shipping. This information is presented below in table. /* to be completed*/

ITEM	WEIGHT	PROFIT/UNIT	NUMBER AVAILABLE
1	1	3	6
2	4	9	1
3	3	8	2
4	2	5	2

Tutorial Exercise8

- Suppose the semester is coming to an end and you have to work on the assigned projects of n courses in a total of H hours. Each course is to be graded on a scale of 1 to g with a higher grade being the better one. You diligently examined the nature of each project problem, amount of work needed, the psychology of the instructor and how much work your partners are doing in the project, to come up with the following estimate functions.

  For all $1 \leq i \leq n$, you have a function

  $f_i : \{0, \ldots\ldots\ldots, H\} \rightarrow \{1, \ldots\ldots\ldots\ldots, g\}$

  such that $f_i(k)$ gives the grade you are likely to get if you spend k hours working on the project for course i.

  Your goal, of course, is to maximize the total of all the grades that you get.

  Design an algorithm to do so. Assume that all the quantities n, g, H are positive natural numbers and the functions f_i are non-decreasing i.e. you do not get a worse grade if you put in more time.