Dynamic Programming

- **Principle of optimality**
- **0/1 Knapsack**
- **Largest Common Subsequence**
- **Travelling Salesperson Problem**
- **Multistage Graph problem (using Forward computation)**

(Ref. Horowithz Sahni and Parag Dave)

Dynamic Programming

- Dynamic programming is typically applied to optimization problem.
- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions

Why Dynamic Programming?

- **Divide-and-Conquer** : a top-down approach.
  partitions a problem into independent subproblems

- **Greedy method :** only works with the local information

- **Dynamic programming :**a bottom-up approach.
  Solutions for smaller instances are stored in a table for later use.

Comparison with divide-and-conquer

- Divide-and-conquer algorithms split a problem into separate subproblems, solve the subproblems, and combine the results for a solution to the original problem
  - Example: Quicksort
  - Example: Mergesort
  - Example: Binary search
- Divide-and-conquer algorithms can be thought of as top-down algorithms
- In contrast, a dynamic programming algorithm proceeds by solving small problems, then combining them to find the solution to larger problems
- Dynamic programming can be thought of as bottom-up

Comparison with Greedy Approach

- Greedy and Dynamic Programming are methods for solving optimization problems.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- Dynamic Programming provides efficient solutions for some problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need only show that the principle of optimality applies to the problem.

Elements of Dynamic Programming …

- Principle of optimality

  In an optimal sequence of decisions or choices, each subsequence must also be optimal.

- Memorization (for overlapping sub-problems)
- avoid calculating the same thing twice,
- usually by keeping a table of know results that fills up as sub-instances are solved.

Example: Fibonacci numbers

Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 24

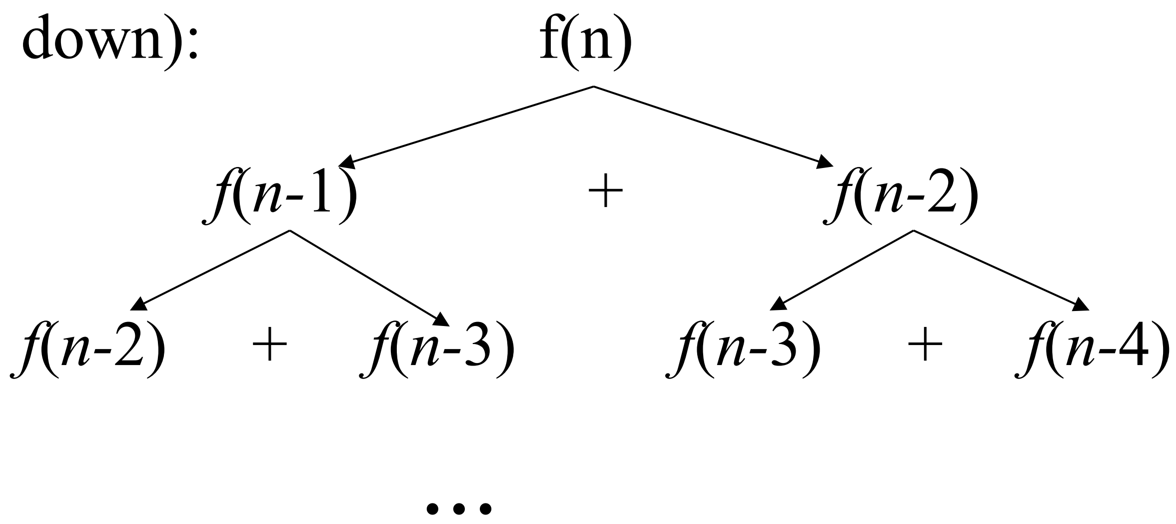
Computing the n^{th} fibonacci number using **bottom-up** iteration:

- $f(0) = 0$
- $f(1) = 1$
- $f(2) = 0+1 = 1$
- $f(3) = 1+1 = 2$
- $f(4) = 1+2 = 3$
- $f(5) = 2+3 = 5$
-
-
-
- $f(n-2) = f(n-3)+f(n-4)$
- $f(n-1) = f(n-2)+f(n-3)$
- $f(n) = f(n-1) + f(n-2)$

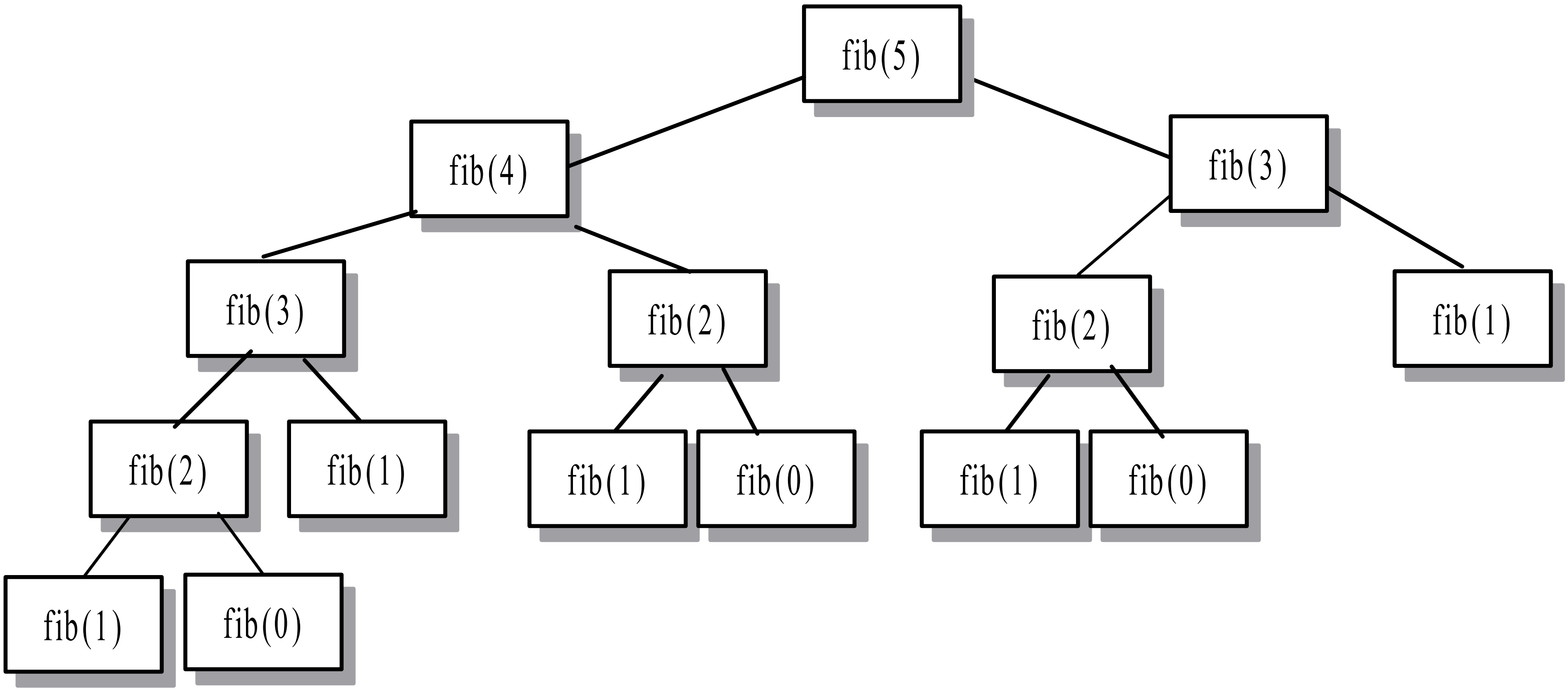
- Recall definition of Fibonacci numbers:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n\text{-}1) + f(n\text{-}2) \qquad \textit{for } n \geq 2$$

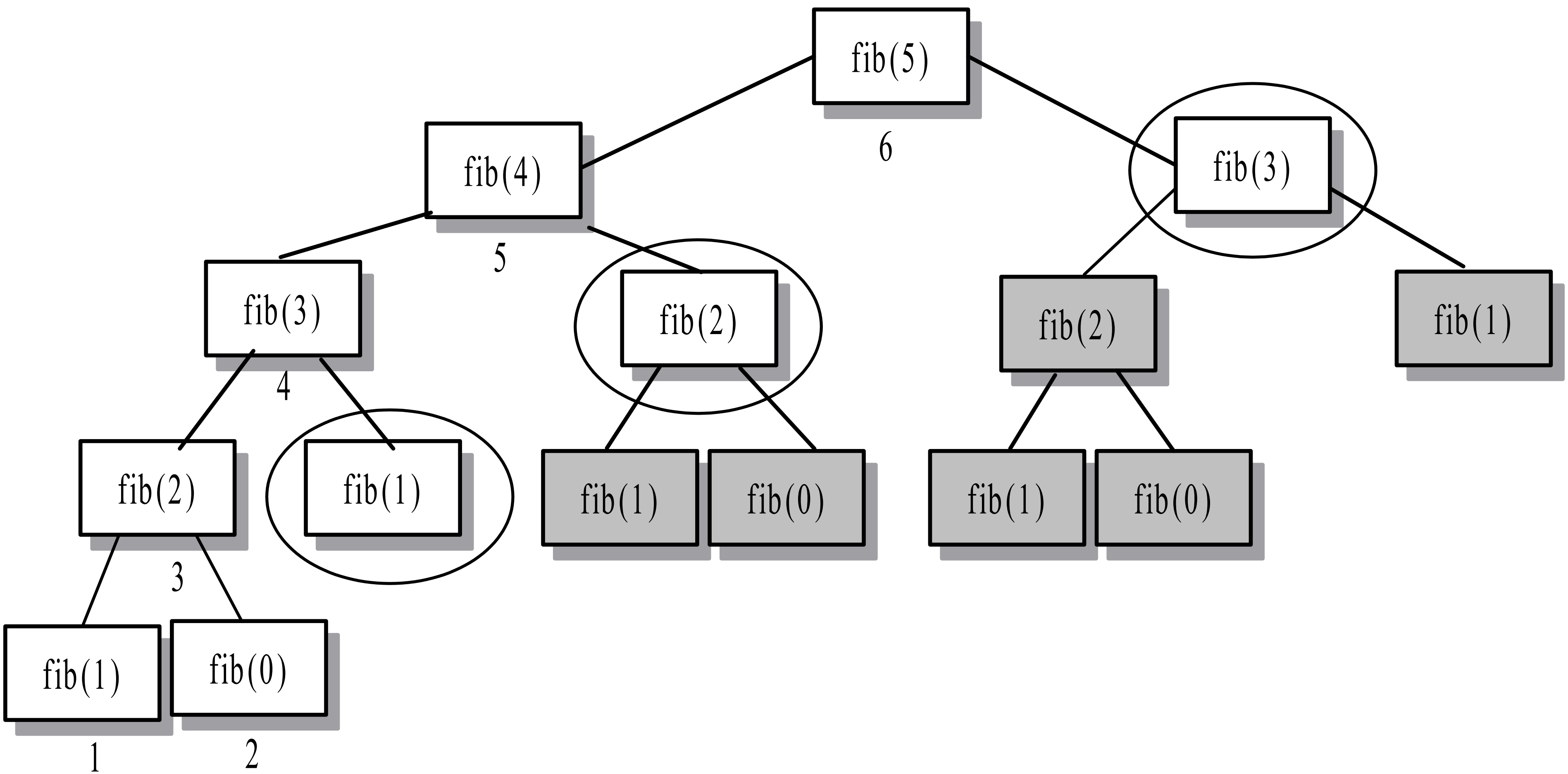
- Computing the n^{th} Fibonacci number recursively (top-down):



Recursive calls for fib



fib Using Dynamic Programming



Knapsack 0-1 Problem

- The difference between this problem ar
  one is that you CANNOT take a fraction
    - You can either take it or not.
    - Hence the name Knapsack 0-1 problem.

Knapsack 0-1 Problem

- As we did before we are going to solve the problem in terms of sub-problems.
  - So let's try to do that…
- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots, I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$ in any regular pattern.
  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that point with an example:

Item	**Weight**	**Value**
I_0	**3**	**10**
I_1	**8**	**4**
I_2	**9**	**9**
I_3	**8**	**11**

- **The maximum weight the knapsack can hold is 20.**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it build's upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- So our recursive formula for subproblems:

$$B[k, w] = B[k-1, w], \text{ if } w_k > w$$
$$= \max\{B[k-1, w], B[k-1, w-w_k] + v_k\}, \text{ otherwise}$$

- this means that the best subset of S_k that has total weight w is:
  1) The best subset of S_{k-1} that has total weight w, or
  2) The best subset of S_{k-1} that has total weight $w-w_k$ plus the item k

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.

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

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

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
   B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
   B[i,0] = 0
}
for i = 1 to n {
   for w = 0 to W {
            if w_i <= w {  //item i can be in the solution
                     if v_i + B[i-1,w-w_i] > B[i-1,w]
                              B[i,w] = v_i + B[i-1,w- w_i]
                     else
                              B[i,w] = B[i-1,w]
            }
            else B[i,w] = B[i-1,w] // w_i > w
   }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

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
```

Knapsack 0-1 Example

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				
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 1$

$w - w_i = -1$

if w_i <= w //item i can be in the solution

  if v_i + B[i-1,w-w_i] > B[i-1,w]

    B[i,w] = v_i + B[i-1,w- w_i]

  else

    B[i,w] = B[i-1,w]

else **B[i,w] = B[i-1,w]** // w_i > w

Knapsack 0-1 Example

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			
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w-w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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		
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 3}$

$w - w_i = 1$

if $w_i \le w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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	
2	0					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 4}$

$w\text{-}w_i = 2$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

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					
3	0					
4	0					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 5}$

$w\text{-}w_i = 3$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

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	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 1}$

$w\text{-}w_i = -2$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 2}$

$w\text{-}w_i = -1$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else **B[i,w] = B[i-1,w]** // w_i > w

Knapsack 0-1 Example

Items:

1: (2,3)

$\boxed{\text{2: (3,4)}}$

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					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 3}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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	
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 4}$

$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 5}$

$w\text{-}w_i = 2$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

Knapsack 0-1 Example

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					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 1..3}$

$w\text{-}w_i = -3..-1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

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	5	
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 4}$

$w\text{-}w_i = 0$

if w_i <= w //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

if w_i <= w //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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	5	7
4	0					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 5}$

$w - w_i = 1$

if $w_i \le w$ //item i can be in the solution

  if $v_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = v_i + B[i-1,w- w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

if $w_i \le w$ //item i can be in the solution

  if $v_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = v_i + B[i-1,w- w_i]$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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	5	7
4	0	0	3	4	5	

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 1..4}$

$w\text{-}w_i = -4..-1$

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else B[i,w] = B[i-1,w] // w_i > w

if w_i <= w //item i can be in the solution

if v_i + B[i-1,w-w_i] > B[i-1,w]

B[i,w] = v_i + B[i-1,w- w_i]

else

B[i,w] = B[i-1,w]

else **B[i,w] = B[i-1,w]** // w_i > w

Knapsack 0-1 Example

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	5	7
4	0	0	3	4	5	7

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 5}$

$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$\mathbf{B[i,w] = B[i-1,w]}$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

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	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 B[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 $B[i, k] \neq B[i-1, k]$ then

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

  $i = i-1$, $k = k-w_i$

  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

$i = 4$
$k = 5$
$v_i = 6$
$w_i = 5$
$\mathbf{B[i,k] = 7}$
$B[i-1,k] = 7$

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

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

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

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

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

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

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

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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

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

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

$i = 1$
$k = 2$
$v_i = 3$
$w_i = 2$
$\mathbf{B[i,k] = 3}$
$B[i-1,k] = 0$
$k - w_i = 0$

k = 0, so we're DONE!

The optimal knapsack should contain:

Item 1 and Item 2

Knapsack 0-1 Problem – Run Time

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

$\mathbf{O(\mathit{W})}$

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

$\mathbf{O(\mathit{n})}$

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

Repeat *n* times

$\mathbf{O(\mathit{W})}$

What is the running time of this algorithm?

O(*n*W*)

Remember that the brute-force algorithm takes: $\mathbf{O(2^n)}$

Brute-force search or **exhaustive search**, also known as **generate and test**, is a very general problem solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

- Consider the problem having weights and profits are:
- Weights: {3, 4, 6, 5}
- Profits: {2, 3, 1, 4}
- The weight of the knapsack is 8 kg
- The number of items is 4

The 0-1 knapsack problem

- A solution to the knapsack problem may be obtained by making a sequence

of decisions on the variables *x1*, *x2*, ... , *xn. A decision on variable x; involves deciding which of the values 0 or 1 is to be assigned to it.*

- Let $f_j(X)$ the value of an optimal solution to KNAP(l,j, X). Since the

principle of optimality holds, we obtain

$$f_n(M) = \max\{f_{n-1}(M), f_{n-1}(M - w_n) + p_n\}$$

- For arbitrary $f_i(X), i > 0$ *quation generalizes to*

$$f_i(X) = \max\{f_{i-1}(X), f_{i-1}(X - w_i) + p_i\}$$

- Equation may be solved $f_n(M)$ *y beginning with the knowledge*

$f_0(X) = 0$ or all *X and* $f_i(x) = -\infty, x < 0. f_1, f_2, \ldots, f_n$ *y be successively*

computed using equation 2.

The 0-1 knapsack problem

(Ref. Horowithz Sahni , page no-305)

- Consider the knapsack instance n = 3, (w1, w2,w3) = (2, 3, 4), (p1, p2, p3) = (1, 2, 5) and M = 6.

Initially compute

$$S^0 = \{(0,0)\}$$

$$S^i_1 = \{(P, W) | (P - p_i, W - w_i) \in S^{i-1}\}$$

Where S^i m… together S^{i-1} and s^i_1.

Purging Rule: If s_{i+1} contains (P_j, W_j) and (P_k, W_k); these two pairs such that $P_j <= P_k$ and $W_j >= W_k$, then (P_j, W_j) can be eliminated . This purging rule is also called as dominance rule. In short, remove the pair with less profit and more weight.

The 0-1 knapsack problem

$S^0 = \{(0, 0)\}; S_1^1 = \{(1, 2)\}$
$S^1 = \{(0, 0), (1, 2)\}; S_1^2 = \{(2, 3), (3, 5)\}$
$S^2 = \{(0, 0), (1, 2), (2, 3), (3, 5)\}; S_1^3 = \{(5, 4), (6, 6), (7, 7), (8, 9)\}$
$S^3 = \{(0, 0), (1, 2), (2, 3), (5, 4), (6, 6), (7, 7), (8, 9)\}.$

Note that the pair (3, 5) has been eliminated from S^3 as a result of the purging rule stated above. □

The 0-1 knapsack problem

procedure *DKP*(p, w, n, M)
    $S^0 \leftarrow \{(0, 0)\}$
    for $i \leftarrow 1$ **to** $n - 1$ **do**
        $S_1^i \leftarrow \{(P1, W1) \mid (P1 - p_i, W1 - w_i) \in S^{i-1}$ and $W1 \leq M\}$
        $S^i \leftarrow MERGE\_PURGE(S^{i-1}, S_1^i)$
    repeat
    $(PX, WX) \leftarrow$ *last tuple in* S^{n-1}
    $(PY, WY) \leftarrow (P1 + p_n, W1 + w_n)$ where $W1$ is the largest W in
        any tuple in S^{n-1} such that $W + w_n \leq M$
    //trace back for $x_n, x_{n-1}, \ldots, x_1$//
    if $PX > PY$ **then** $x_n \leftarrow 0$
                else $x_n \leftarrow 1$
    endif
    trace back for $x_{n-1}, \ldots, x_1$
end *DKP*

Largest/Longest Common Subsequence (LCS)

(Ref. Parag Dave, page no-285)

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, “’acefg”, .. etc are subsequences of “abcdefg”.

Problem:
Given two sequences X = <x1,x2,...xm> and Y = <y1,y2,...yn> , Find the longest sub-sequence Z = <z1,....zk> that is common to X and Y.

For example:
If X = < A,B,C,B,D,A,B> and Y = <B,D,C,A,B,A>
then some common sub-sequences are:
{A} {B} {C} {D} {A,A} {B,B} {B,C,A} {B,C,A} {B,C,B,A } {B,D,A,B}

From which {B,C,B,A } {B,D,A,B} are the Longest Common sub-sequences.

$c[i, j]$ = length of LCS for $X[i]$ and $Y[j]$.

C[i, j]= length of LCS for X[i] and Y[j]
➔

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_i \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_i \end{cases}$$

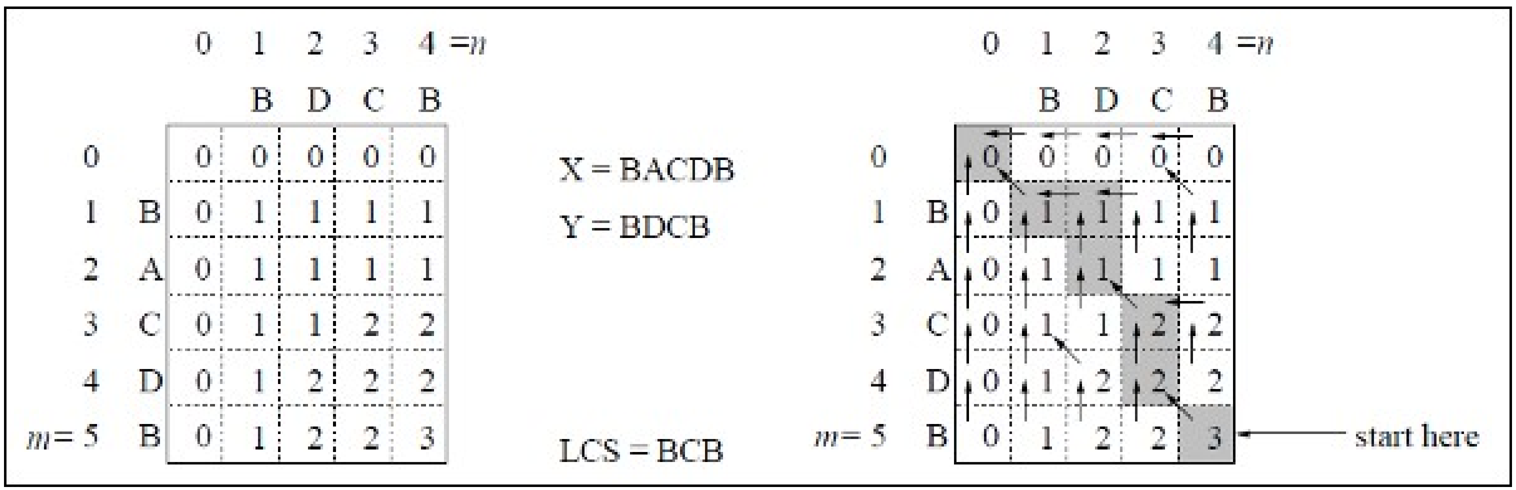
```
LCS-Length(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
        do c[i, 0] ← 0
for j ← 0 to n
        do c[0, j] ← 0
for i ← 1 to m
        do for j ← 1 to n
                do if x_i = y_j
                        then c[i, j] ← c[i − 1, j − 1] + 1
                             b[i, j] ← "↖"
                        else if c[i − 1, j] ≥ c[i, j − 1]
                                then c[i, j] ← c[i − 1, j]
                                     b[i, j] ← "↑"
                                else c[i, j] ← c[i, j − 1]
                                     b[i, j] ← "←"
return c and b
```

PRINT-LCS (b, x, i, j)

1. if i=0 or j=0
2. then return
3. if b [i,j] = ' ↖ '
4. then PRINT-LCS (b,x,i-1,j-1)
5. print x_i
6. else if b [i,j] = ' ↑ '
7. then PRINT-LCS (b,X,i-1,j)
8. else PRINT-LCS (b,X,i,j-1)

Largest/Longest Common Subsequence (LCS)

Example:
Given two strings are ***X* = *BACDB*** and ***Y* = *BDCB*** find the longest common subsequence.



Travelling Salesman Problem

(Ref. Horowithz Sahni , page no-319)

In the traveling salesman problem, a map of cities is given to the salesman and he has to visit all the cities only once and return to his starting point to complete the tour in such a way that the length of the tour is the shortest among all possible tours for this map. Clearly starting from a given city, the salesman will have a total of (n-1)! Different sequences:

If n = 2, A and B, there is no choice.

If n = 3, i.e. he wants to visit three cities

inclusive of the starting point, he has 2! Possible routes and so on.

Travelling Salesman Problem

(Ref. Horowithz Sahni , page no-319)

The Dynamic Programming proceeds as follows:-

Step-1

Consider the given travelling salesman problem in which he wants to find that route which has shortest distance.

Step-2

Consider set of 0element, such that

$g(2, \Phi) = c21$ $g(3, \Phi) = c31$ $g(4, \Phi) = c41$

Step-3

After completion of step-2, consider sets of 1 elements, such that

Set {2}: $g(3,\{2\}) = c32 + g(2, \Phi) = c32 + c21$

$g(4,\{2\}) = c42 + g(2, \Phi) = c42 + c21$

Set {3}: $g(2,\{3\}) = c23 + g(3, \Phi) = c23 + c31$

$g(4,\{3\}) = c43 + g(3, \Phi) = c43 + c31$

Set {4}: $g(2,\{4\}) = c24 + g(4, \Phi) = c24 + c41$

$g(3,\{4\}) = c34 + g(4, \Phi) = c34 + c41$

Step-4

After completion of step-3, consider sets of 2 elements, such that

Set {2,3}: $g(4,\{2,3\}) = \min \{c42 + g(2,\{3\}), c43 + g(3,\{2\})\}$

Set {2,4}: $g(3,\{2,4\}) = \min \{c32 + g(2,\{4\}), c34 + g(4,\{2\})\}$

Set {3,4}: $g(2,\{3,4\}) = \min \{c23 + g(3,\{4\}), c24 + g(4,\{3\})\}$

Step-5

After completion of step-4, Find the length of an optimal tour:

$f = g(1,\{2,3,4\}) = \min \{ c12 + g(2,\{3,4\}), c13 + g(3,\{2,4\}), c14 + g(4,\{2,3\}) \}$

Step-6

After completion of step-5, Find the Optimal TSP tour

Travelling Salesman Problem

(Ref. Horowithz Sahni , page no-319)

Solve the TSP problem for given Distance matrix.

Distance matrix

$$\begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$$

g(2, Φ) = c21 = 1
g(3, Φ) = c31 = 15
g(4, Φ) = c41 = 6

k = 1, consider sets of 1 element:

Set {2}: g(3,{2}) = c32 + g(2, Φ) = c32 + c21 = 7 + 1 = 8
g(4,{2}) = c42 + g(2, Φ) = c42 + c21 = 3 + 1 = 4

Set {3}: g(2,{3}) = c23 + g(3, Φ) = c23 + c31 = 6 + 15 = 21
g(4,{3}) = c43 + g(3, Φ) = c43 + c31 = 12 + 15 = 27

Set {4}: g(2,{4}) = c24 + g(4, Φ) = c24 + c41 = 4 + 6 = 10
g(3,{4}) = c34 + g(4, Φ) = c34 + c41 = 8 + 6 = 14

k = 2, consider sets of 2 elements:

Set {2,3}: g(4,{2,3}) = min {c42 + g(2,{3}), c43 + g(3,{2})} = min {3+21, 12+8}= min {24, 20}= 20
Set {2,4}: g(3,{2,4}) = min {c32 + g(2,{4}), c34 + g(4,{2})} = min {7+10, 8+4}= min {17, 12} = 12
Set {3,4}: g(2,{3,4}) = min {c23 + g(3,{4}), c24 + g(4,{3})} = min {6+14, 4+27}= min {20, 31}= 20

Length of an optimal tour:
f = g(1,{2,3,4}) = min { c12 + g(2,{3,4}), c13 + g(3,{2,4}), c14 + g(4,{2,3}) }
= min {2 + 20, 9 + 12, 10 + 20}
= min {22, 21, 30} = 21

Successor of node 1: c13 + g(3,{2,4}) = 3
Successor of node 3: = c34 + g(4,{2})} =4
Successor of node 4: g(4,{2}) =2
Successor of node 2: back to staring node 1

Optimal TSP tour: 1 →3 → 4 →2→1 with minimum cost= 21

Dynamic Programming

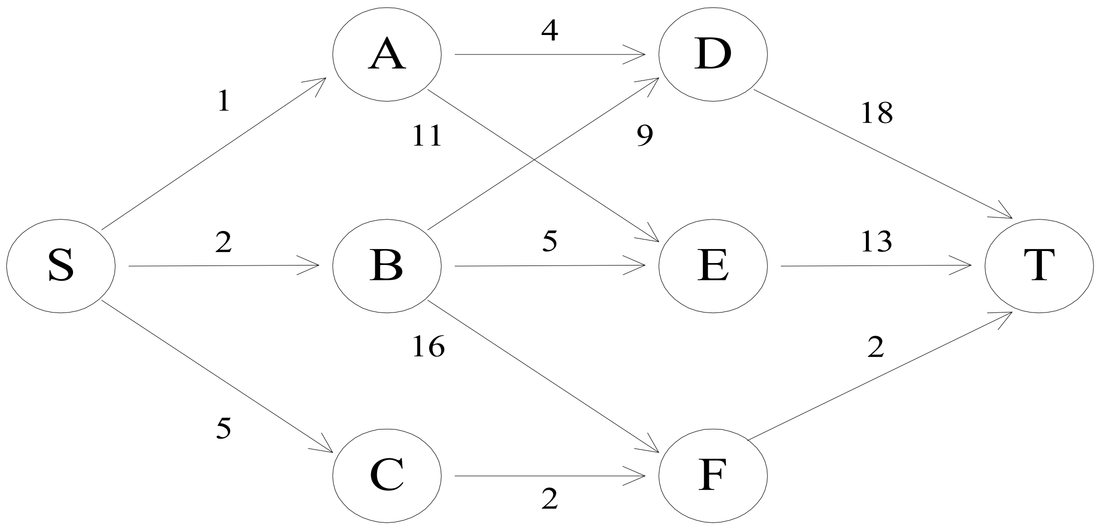
- Dynamic Programming is an algorithm design method that can be used when the solution to a problem may be viewed as the result of a sequence of decisions

Multi-stage graph

- A multistage graph is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$.
- $<u, v>$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some i, $1 \leq i \leq k$.
- The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$
- s and t are the vertices in V_1 and V_k respectively.
- The vertex s is the source and t is the sink
- The multi stage graph is to find a minimum cost path from s to t.

The shortest path in multistage graphs

- e.g.

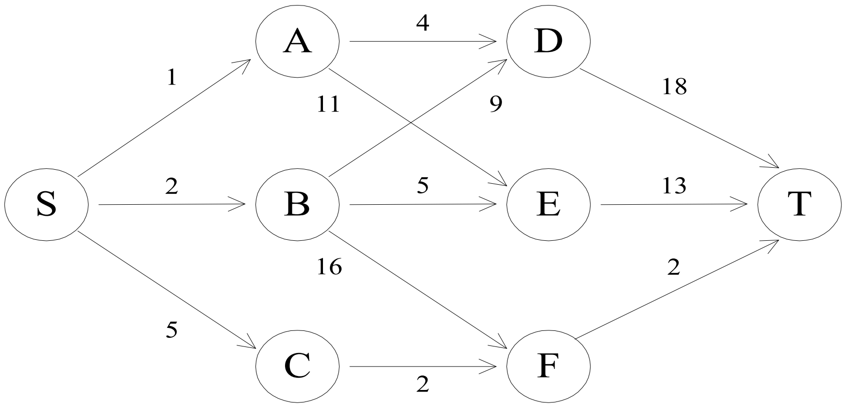


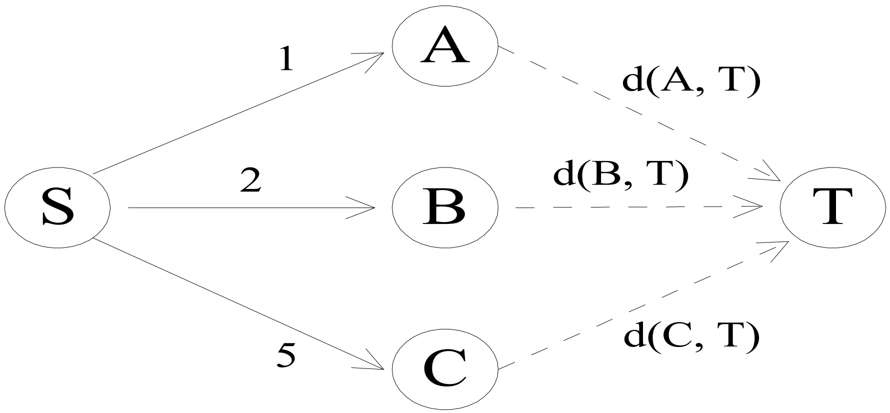
- The greedy method can not be applied to this case: (S, A, D, T) 1+4+18 = 23.
- The real shortest path is:

  (S, C, F, T) 5+2+2 = 9.

Dynamic programming approach

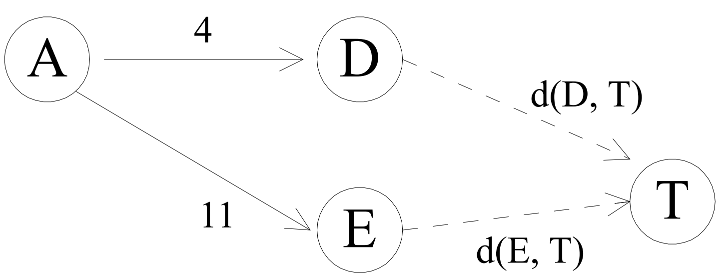
- Dynamic programming approach ([forward approach]()):



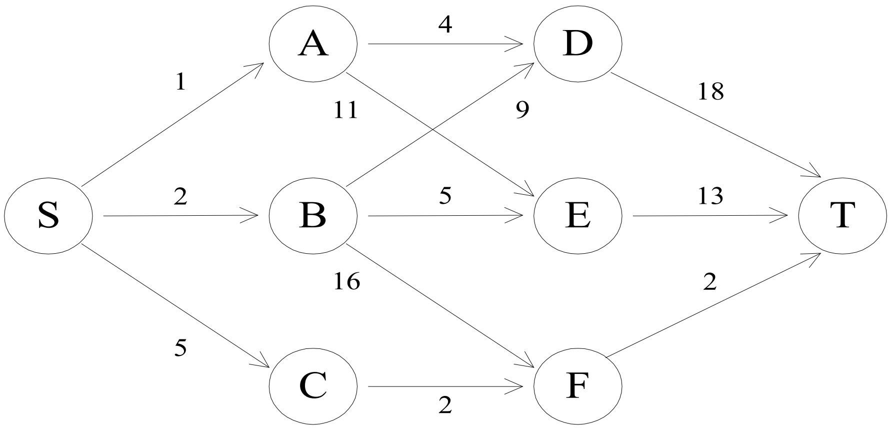


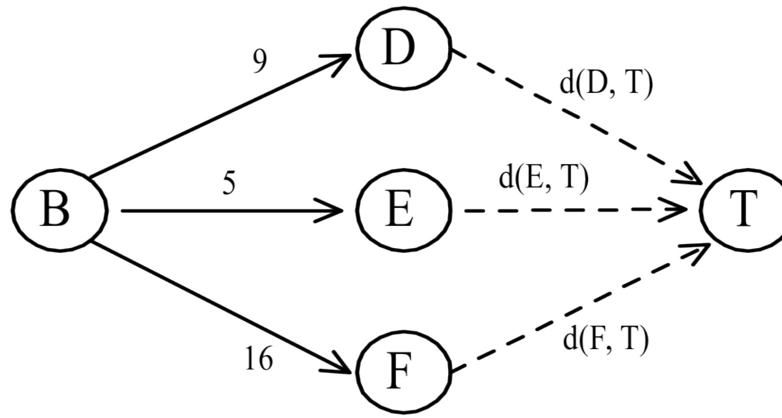
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}

- d(A,T) = min{4+d(D,T), 11+d(E,T)}
  = min{4+18, 11+13} = 22.



- d(B, T) = min{9+d(D, T), 5+d(E, T), 16+d(F, T)}
  = min{9+18, 5+13, 16+2} = 18.





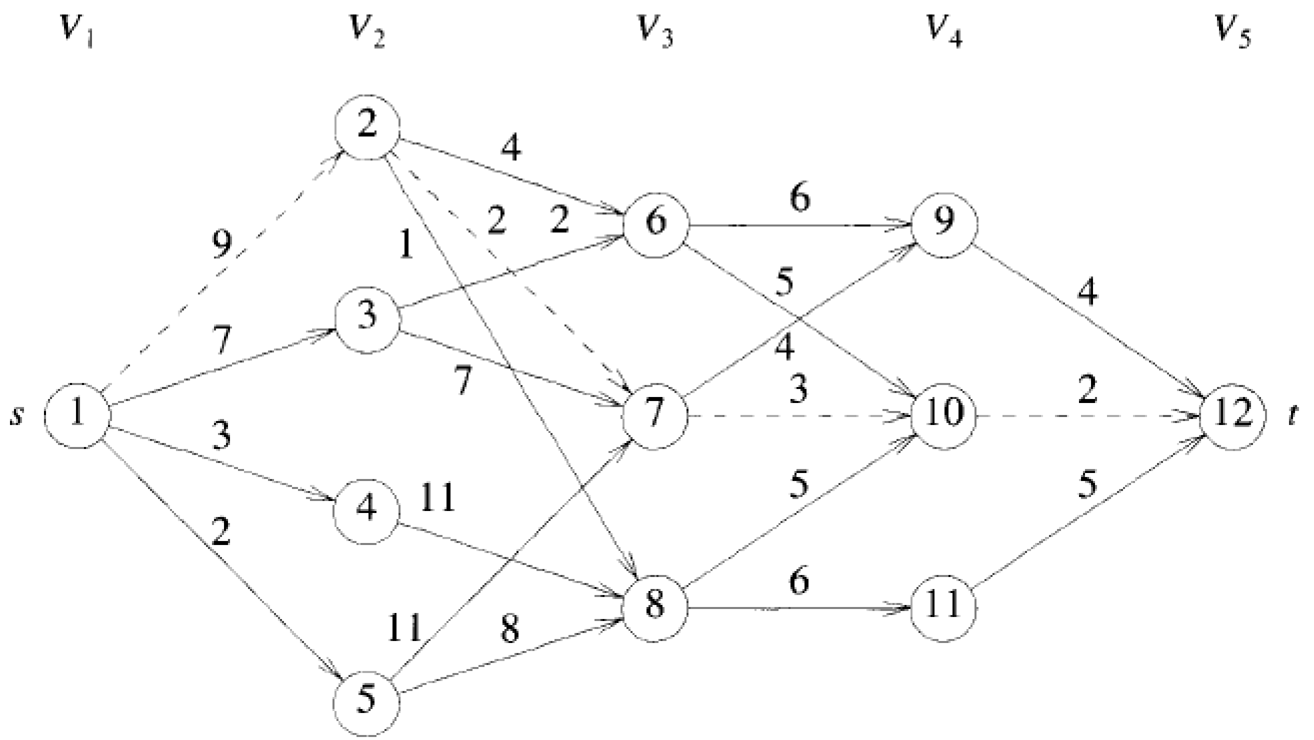
- d(C, T) = min{ 2+d(F, T) } = 2+2 = 4
- d(S, T) = min{1+d(A, T), 2+d(B, T), 5+d(C, T)}
  = min{1+22, 2+18, 5+4} = 9.
- The above way of reasoning is called backward reasoning.

```
Algorithm Fgraph(G,k,n,p)
//input is k stage graph G=(V,E) with n vertices
//indexed in order of stages
// E set of edges, c[i][j] is cost of <i,j>
// p[1..k] is a minimum cost path
{
   fcost[n] = 0.0;
For j= n-1 to 1 step -1 do
{ // compute fcost[j]
   Let r be the vertex such that <j, r> is an edge
   of G and c[j][r] + fcost[r] is minimum;
   fcost[j] = c[j][r] + fcost[r];
   d[j]=r ;
}
p[1]=1; p[k] = n;
for j= 2 to k-1 do p[j] = d[p[j-1]];
}
```

Complexity

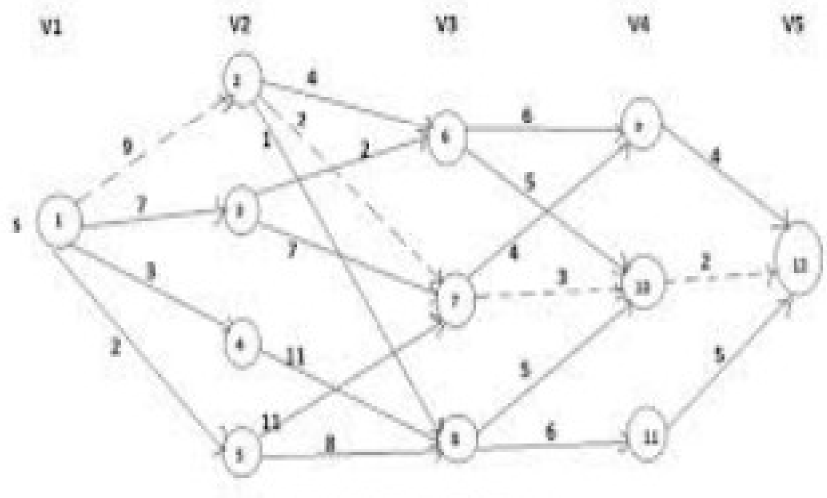
- G represented using adjacency list
- Vertex r can be found in time proportional to the degree of vertex j.
- If G has |E| edges, the total time required is $\Theta(|v| + |E|)$
- Additional space required for cost[],d[],p[]

Multi-stage graph



Multistage Graph-forward approach

- $Cost(i, j) = \min \{ c(j, l) + cost\ (i+1, l) \}\ l \in V_{i+1}\ ,\ \langle j,l \rangle \in E$



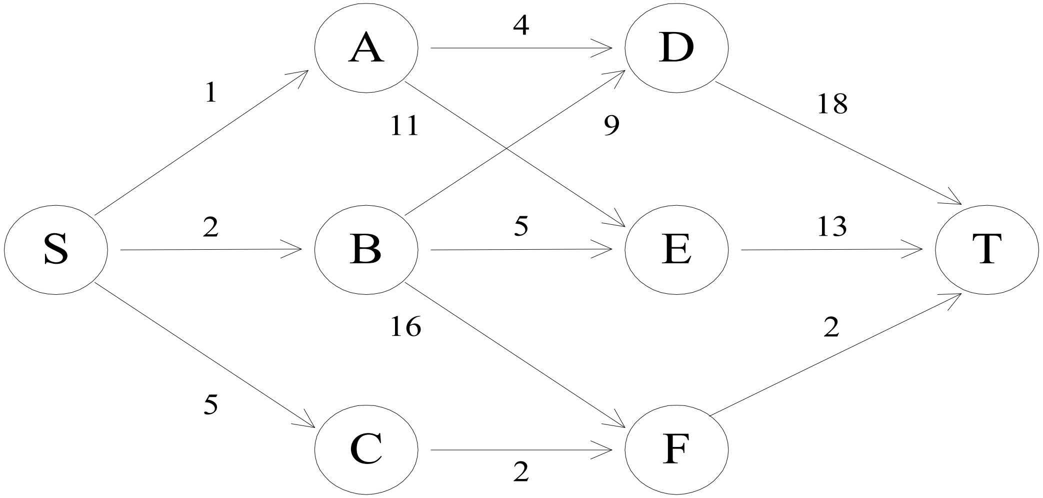
MULTI STAGE GRAPH

- Cost(5,12)=0;
- Cost(4,9)=4+ Cost(5,12)=4;
- Cost(4,10)=2+ Cost(5,12)=2;
- Cost(4,11)=5+ Cost(5,12)=5;
- Cost(3,6)=min{6+cost(4,9), 5+cost(4,10)}=min{10,7}=7
- Cost(3,7)=min{4+cost(4,9), 3+cost(4,10)}=min{8,5}=5
- Cost(3,8)=min{5+cost(4,10),6+cost(4,11)}=min{7,11}=7
- Cost(2,2)=min{4+cost(3,6),2+cost(3,7),1+cost(3,8)}=min{11,7,8}=7
- Cost(2,3)=min{2+cost(3,6),7+cost(3,7)}=min{9,12}=9
- Cost(2,4)=11+cost(3,8)=18
- Cost(2,5)=min{11+cost(3,7), 8+cost(3,8)}=min{16,15}=15
- Cost(1,1)=min{9+cost(2,2),7+cost(2,3),3+cost(2,4),2+cost(2,5)}
  =min{16,16,21,17}=16

Shortest path 1-2-7-10-12

$Cost(i, j)=\min \{ c(j, l) + cost(i+1,l)\}\ l \in V_{i+1},\ \langle j,l\rangle \in E$

Backward approach (forward reasoning)

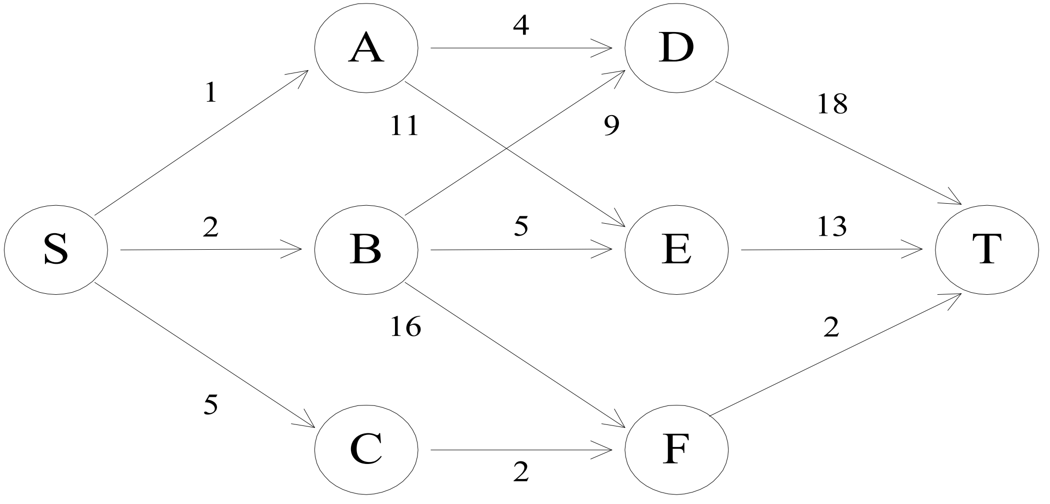


- d(S, A) = 1
  d(S, B) = 2
  d(S, C) = 5
- d(S,D)=min{d(S,A)+d(A,D), d(S,B)+d(B,D)}
  = min{ 1+4, 2+9 } = 5
  d(S,E)=min{d(S,A)+d(A,E), d(S,B)+d(B,E)}
  = min{ 1+11, 2+5 } = 7
  d(S,F)=min{d(S,B)+d(B,F), d(S,C)+d(C,F)}
  = min{ 2+16, 5+2 } = 7

- $d(S,T) = \min\{d(S, D)+d(D, T), d(S,E)+ d(E,T), d(S, F)+d(F, T)\}$

  $= \min\{ 5+18, 7+13, 7+2 \}$

  $= 9$



```
Algorithm Bgraph(G,k,n,p)
//input is k stage graph G=(V,E) with n vertices
//indexed in order of stages
// E set of edges, c[i][j] is cost of <i,j>
// p[1..k] is a minimum cost path
{
    bcost[1] = 0.0;
For j=2 to n do
{ // compute bcost[j]
    Let r be the vertex such that <r, j> is an edge
    of G and  bcost[r] + c[r][j] is minimum;
    bcost[j] = bcost[r] + c[r][j];
d[j]=r ;
}
p[1]=1; p[k] = n;
for j=  k-1 to 2 do p[j] = d[p[j+1]];
}
```

Principle of optimality

- Principle of optimality: Suppose that in solving a problem, we have to make a sequence of decisions $D_1, D_2, \ldots, D_n$. If this sequence is optimal, then the last k decisions, $1 < k < n$ must be optimal.
- e.g. the shortest path problem

  If $i, i_1, i_2, \ldots, j$ is a shortest path from i to j, then $i_1, i_2, \ldots, j$ must be a shortest path from i_1 to j
- In summary, if a problem can be described by a multistage graph, then it can be solved by dynamic programming.

Dynamic programming

- Forward approach and backward approach:
  - Note that if the recurrence relations are formulated using the forward approach then the relations are solved backwards . i.e., beginning with the last decision
  - On the other hand if the relations are formulated using the backward approach, they are solved forwards.
- To solve a problem by using dynamic programming:
  - Find out the recurrence relations.
  - Represent the problem by a multistage graph.

The End