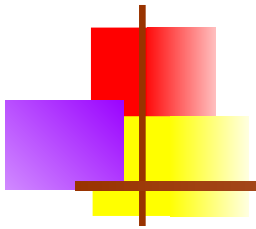
Dynamic Programming

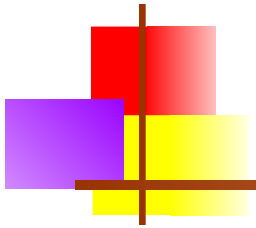
- **Principle of optimality**
- **0/1 Knapsack**
- **Largest Common Subsequence**
- **Travelling Salesperson Problem**

(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?

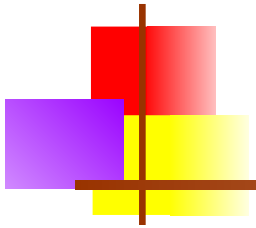
- **Divide-and-Conquer** : a top-down approach.

  partitions a problem into independent sub-problems

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

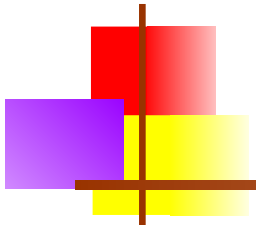
- **Dynamic programming :** a bottom-up approach.

  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.

Instances when greedy approach fails

There are tons of tasks where greedy algorithms fail.

e.g. **Coin changing problem**, if you have coins 1,6,8, then 12=6+6 is better than 12=8+1+1+1+1.

e.g. **Traveling salesman problem**

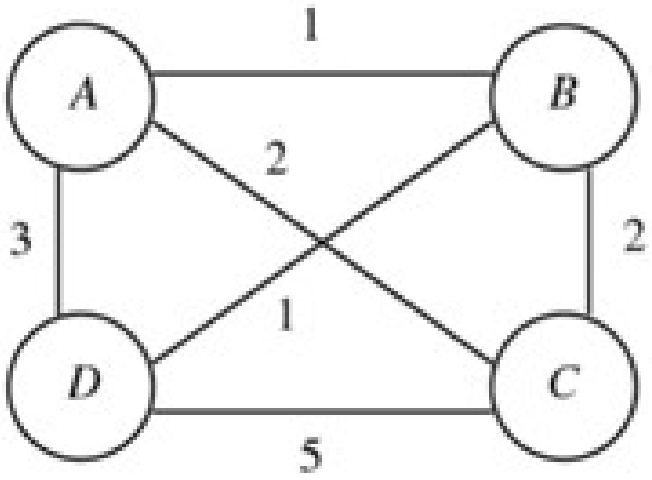
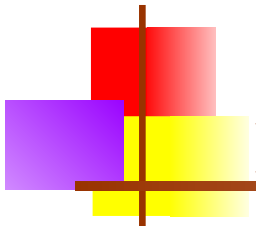


FIGURE 4.7 Example graph for TSP.

From A, the greedy cycle is $ABDCA$ of length 9, while $ACBDA$ has length 8.



Elements of Dynamic Programming …

- Principle of optimality

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

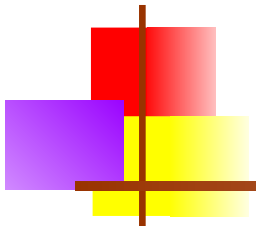
- Memorization (for overlapping sub-problems)
- avoid calculating the same thing twice,
- usually by keeping a table of known results that fills up as sub-instances are solved.

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:

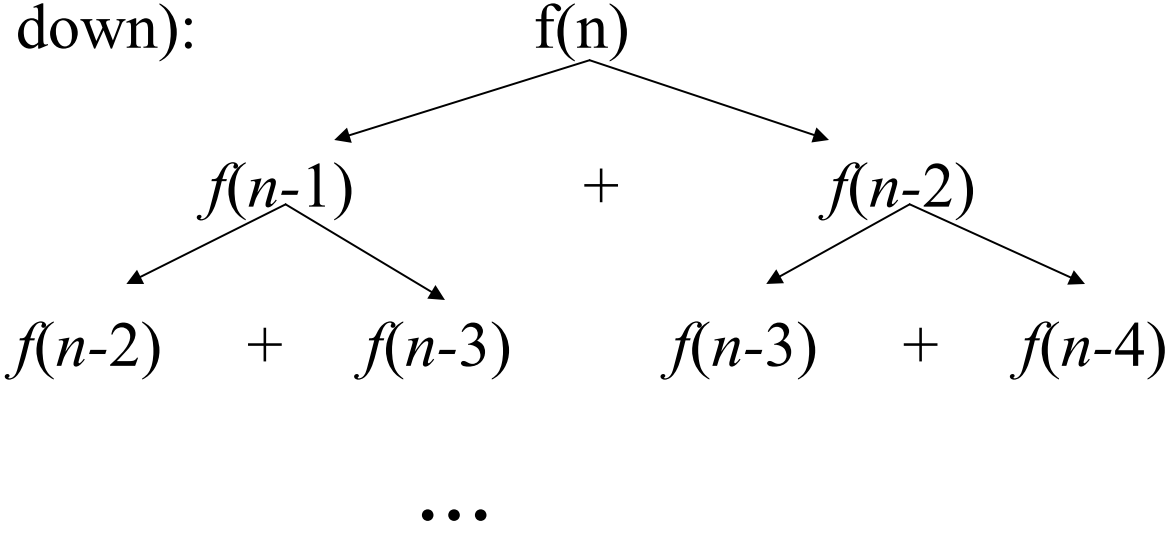
- $f(0) = 0$
- $f(1) = 1$
- $f(2) = 0+1 = 1$
- $f(3) = 1+1 = 2$
- $f(4) = 1+2 = 3$
- $f(5) = 2+3 = 5$
- .
- .
- .
- $f(n-2) = f(n-3)+f(n-4)$
- $f(n-1) = f(n-2)+f(n-3)$
- $f(n) = f(n-1) + f(n-2)$



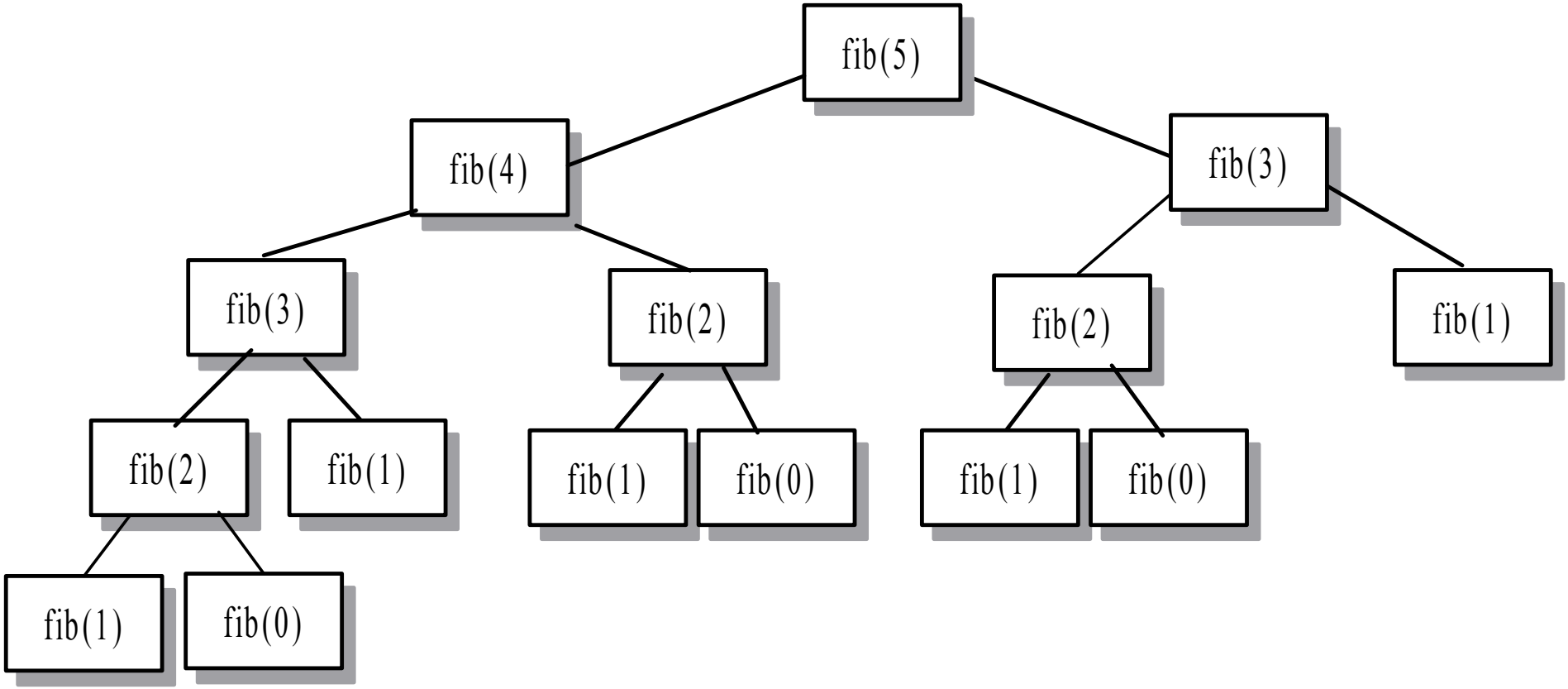
- Recall definition of Fibonacci numbers:

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

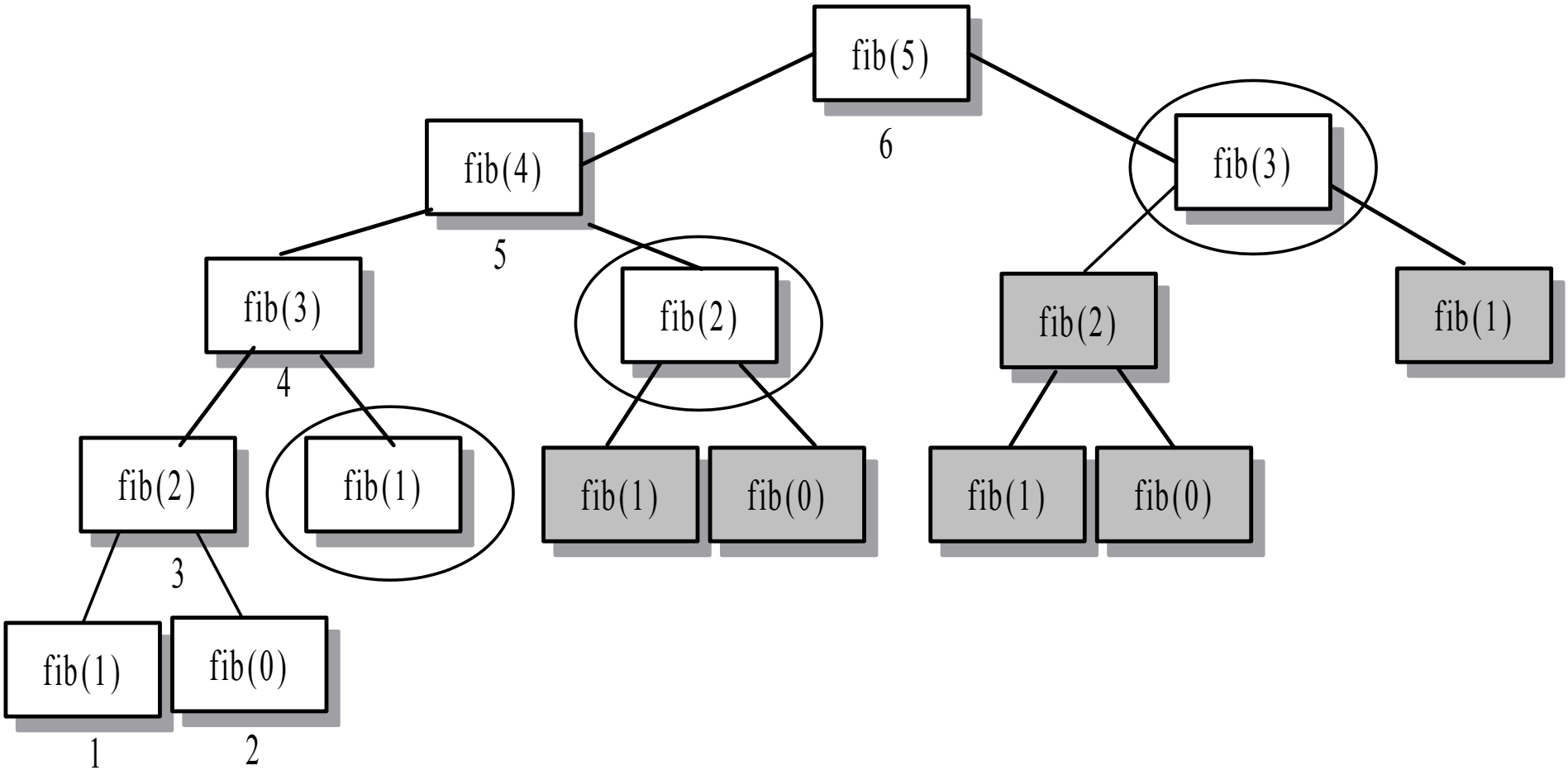
- Computing the n^{th} Fibonacci number recursively (top-down):



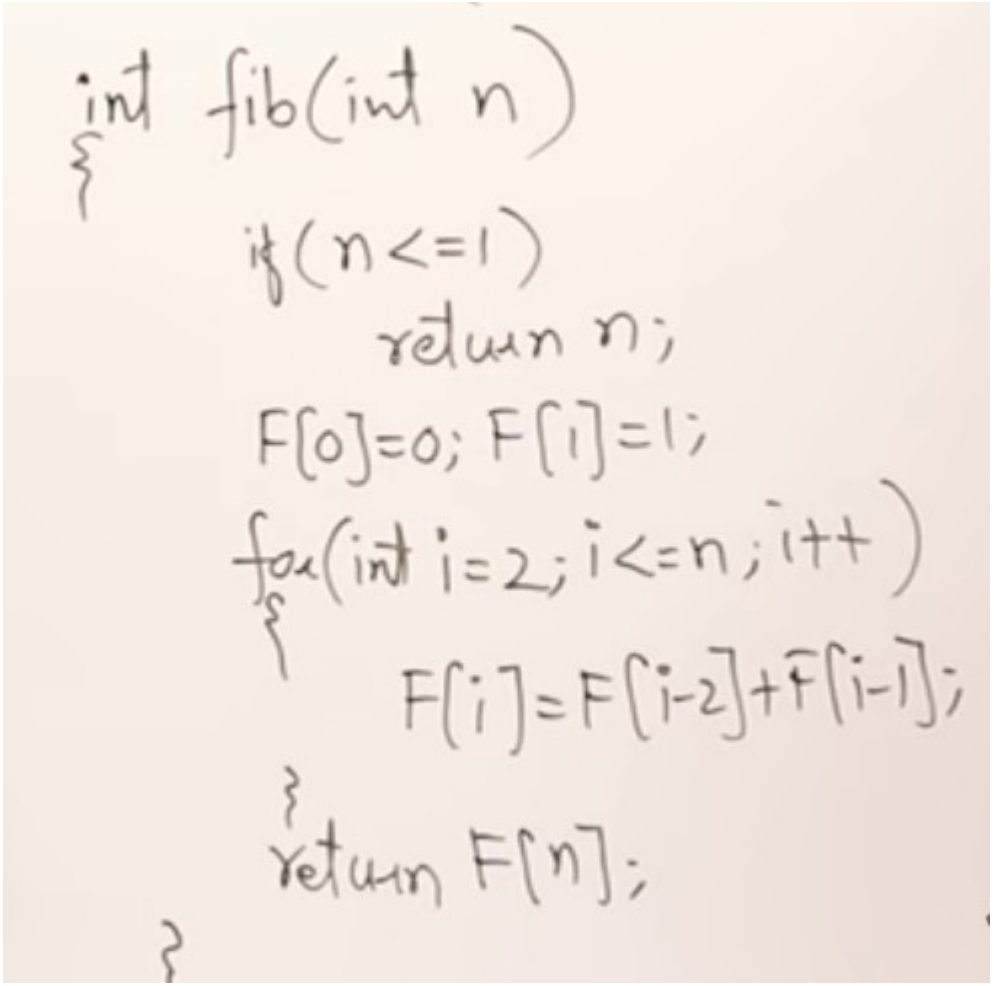
Recursive calls for fib



fib Using Dynamic Programming



fib Using Dynamic Programming without recursion (iterative)



```
int fib(int n)
{
    if(n<=1)
        return n;
    F[0]=0; F[1]=1;
    for(int i=2; i<=n; i++)
    {
        F[i]=F[i-2]+F[i-1];
    }
    return F[n];
}
```

Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - 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

The 2D array/matrix B[i,w]

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

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

Knapsack 0-1 Problem Recursive Formula

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

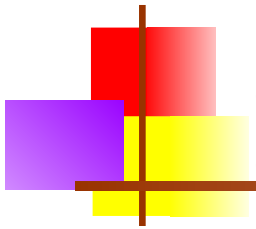
- The best subset of S_k that has the total weight w, either contains item k or not.

- **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:** $w_k \leq w$
  - Then the item *k* can 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					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 1$

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

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-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					

$i = 1$

$v_i = 3$

$w_i = 2$

w = 2

$w\text{-}w_i = 0$

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

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					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 3$

$w\text{-}w_i = 1$

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

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					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 4$

$w\text{-}w_i = 2$

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

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					

$i = 1$

$v_i = 3$

$w_i = 2$

$w = 5$

$w\text{-}w_i = 3$

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

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					

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

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]** // $w_i > w$

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

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					

$i = 2$

$v_i = 4$

$w_i = 3$

w = 2

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

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

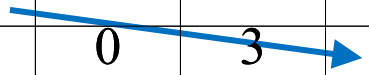
$B[i,w] = B[i\text{-}1,w]$

else **B[i,w] = B[i-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		
3	0					
4	0					



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

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

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					

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

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

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					

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

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

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					

$i = 3$
$v_i = 5$
$w_i = 4$
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 **$B[i,w] = B[i\text{-}1,w]$** // $w_i > w$

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					

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

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

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					

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

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

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	

$i = 4$
$v_i = 6$
$w_i = 5$
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

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

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

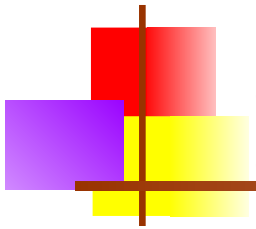
Knapsack 0-1 Example

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!!

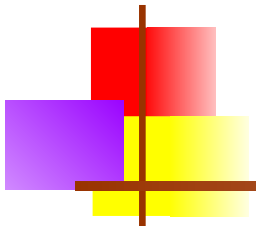
The max possible value that can be carried in this knapsack is ***$7***



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] ≠ 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?

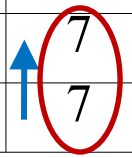
Knapsack 0-1 Algorithm Finding the Items

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

i = n , k = W
while i, k > 0
    if *B[i, k] ≠ 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

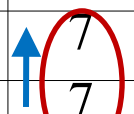
Knapsack 0-1 Algorithm Finding the Items

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

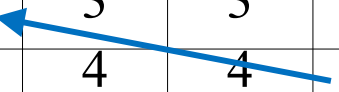
i = n , k = W
while i, k > 0
    if *B[i, k] ≠ B[i-1, k]* then
        mark the i^{th} item as in the knapsack
        i = i-1, k = k-w_i
    else
        i = i-1

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



$i = 2$
$k = 5$
$v_i = 4$
$w_i = 3$
B[i,k] = 7
B[i-1,k] = 3
$k - w_i = 2$

$i = n$, $k = W$
while i, k > 0
    if *B[i, k] ≠ 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
$v_i = 3$
$w_i = 2$
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
$v_i = 3$
$w_i = 2$
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 - 1

Weights : 2 3 3 4 6

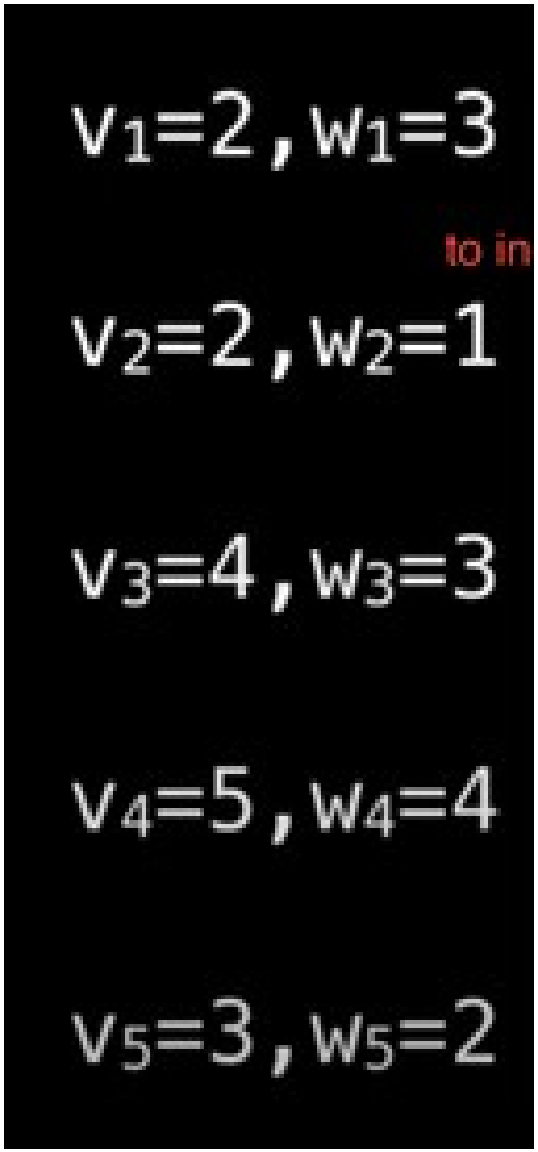
Values : 1 2 5 9 4

Knapsack Capacity (W) = 10

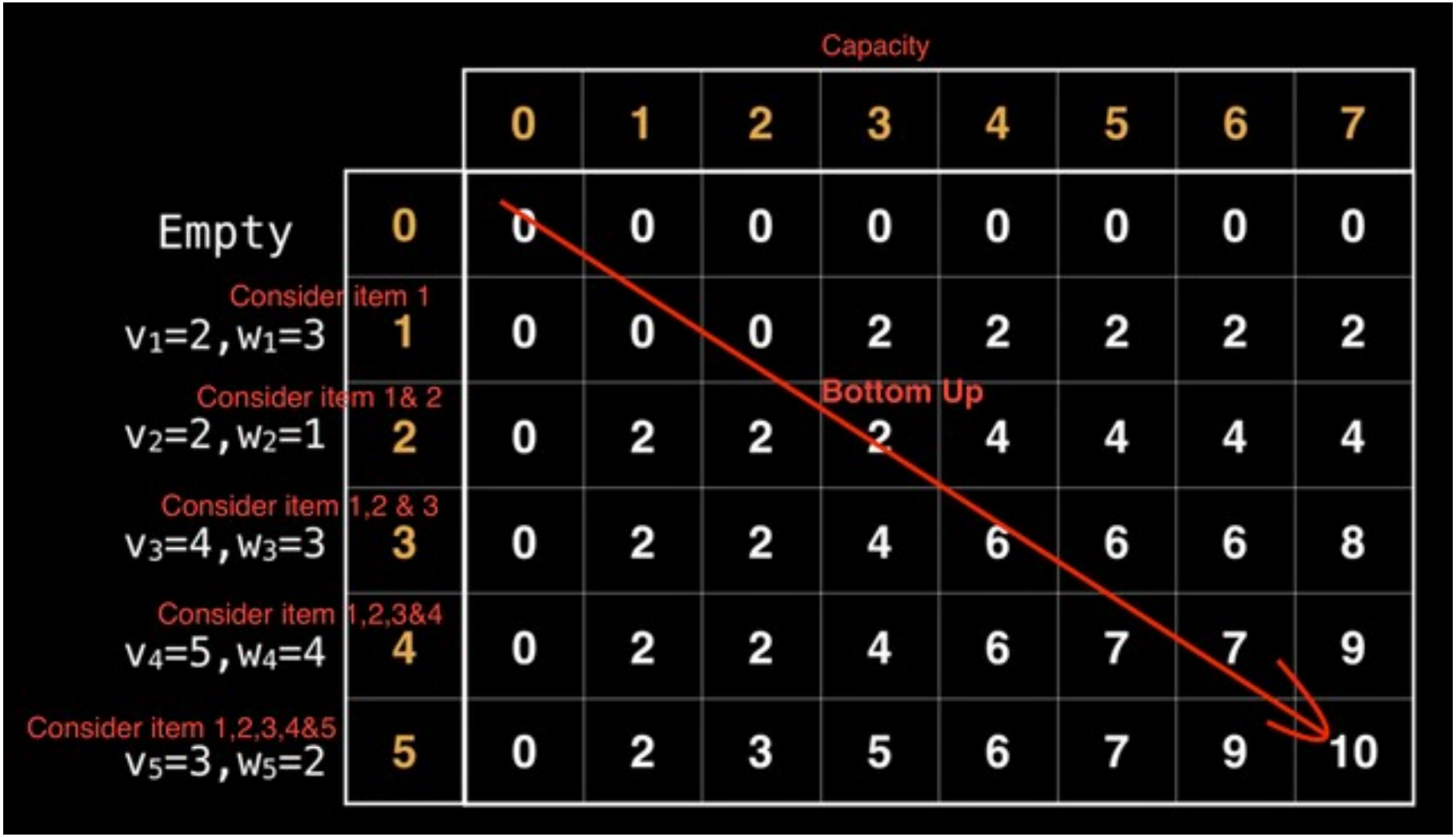
Knapsack 0-1 Solution Problem - 1

	0	1	2	3	4	5	6	7	8	9	10
2(1)	0	0	1	1	1	1	1	1	1	1	1
3(2)	0	0	1	2	2	3	3	3	3	3	3
3(5)	0	0	1	5	5	6	7	7	8	8	8
4(9)	0	0	1	5	9	9	10	14	14	15	16
6(4)	0	0	1	5	9	9	10	14	14	15	16

Knapsack 0-1 Problem – 2 (W=7)



Knapsack 0-1 solution – 2 (W=7)



Knapsack 0-1 Problem – Run Time

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

O(*W*)

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

O(*n*)

Repeat *n* times

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

O(*W*)

What is the running time of this algorithm?

O(*n*W*)**

Remember that the brute-force algorithm takes: **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.

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 $f_j(X)$ be 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 $f_i(X), i > 0$, *Equation generalizes to*

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

- Equation may be solved $f_n(M)$ by *beginning with the knowledge* $f_0(X) = 0$ for all *X and* $f_i(x) = -\infty, x < 0. f_1, f_2, \ldots, f_n$ *may 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

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

$$S_1^i = \{(P, W) \mid (P - p_i, W - w_i) \in S^{i-1}\}$$

Where S^i may now be obtained by merging 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

```
line  procedure DKP(p, w, n, M)
 1      S^0 ← {(0, 0)}
 2      for i ← 1 to n − 1 do
 3        S^i_1 ← {(P1, W1) | (P1 − p_i, W1 − w_i) ∈ S^{i−1} and W1 ≤ M}
 4        S^i ← MERGE_PURGE(S^{i−1}, S^i_1)
 5      repeat
 6      (PX, WX) ← last tuple in S^{n−1}
 7      (PY, WY) ← (P1 + p_n, W1 + w_n) where W1 is the largest W in
          any tuple in S^{n−1} such that W + w_n ≤ M
        //trace back for x_n, x_{n−1}, ..., x_1//
 8      if PX > PY then x_n ← 0
 9                 else x_n ← 1
10      endif
11      trace back for x_{n−1}, ..., x_1
12    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] ➔

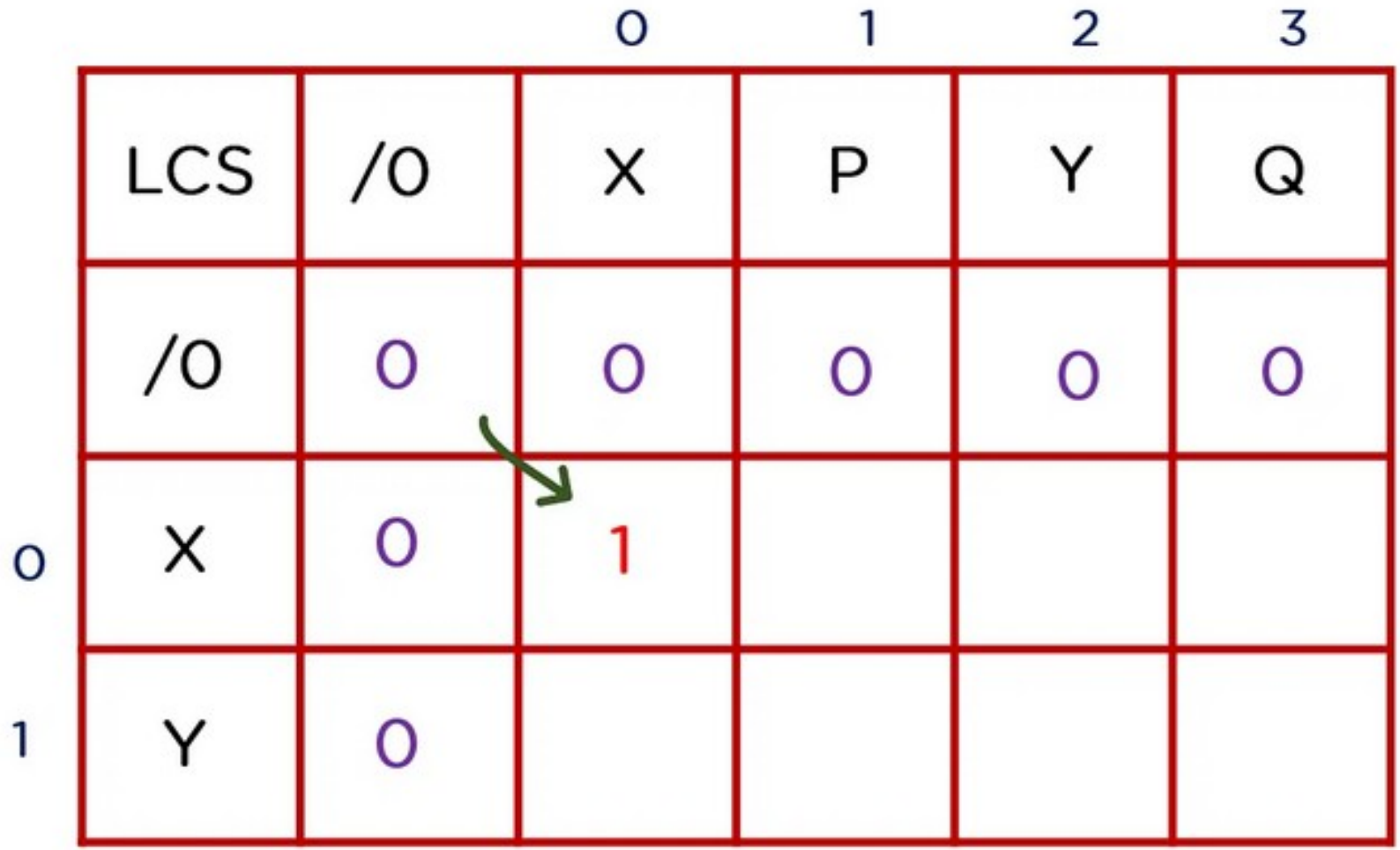
$$c[i,j] = \text{length of LCS for } X[i] \text{ and } Y[j].$$

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

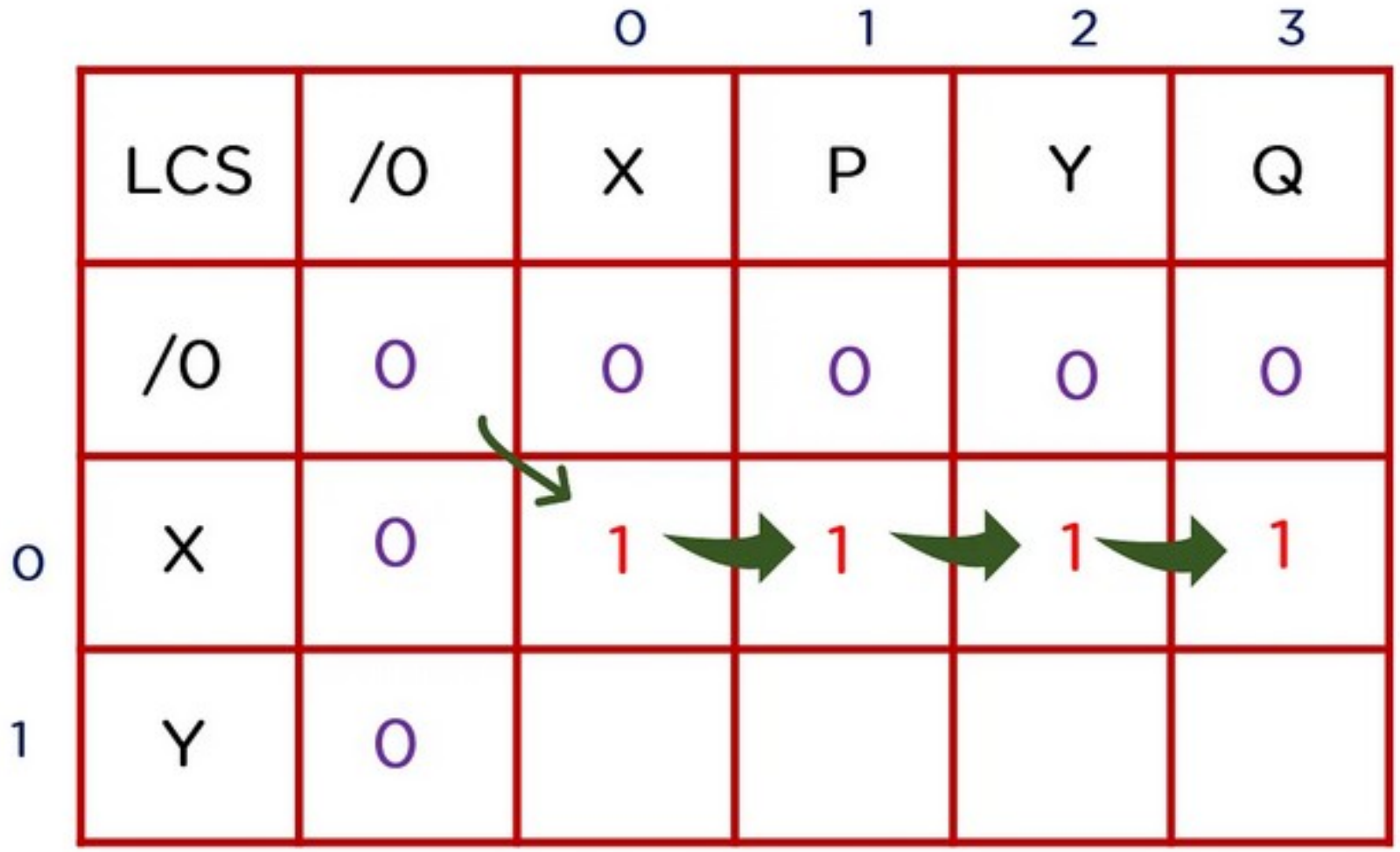
- Step 1: Make a table with the dimensions (x+1*y+1), where x and y are the lengths of strings A and B, respectively. Insert a null column and row to represent the end of the string. Now, insert zeros in all first rows and columns.

			0	1	2	3
	LCS	/0	X	P	Y	Q
	/0	0	0	0	0	0
0	X	0				
1	Y	0				

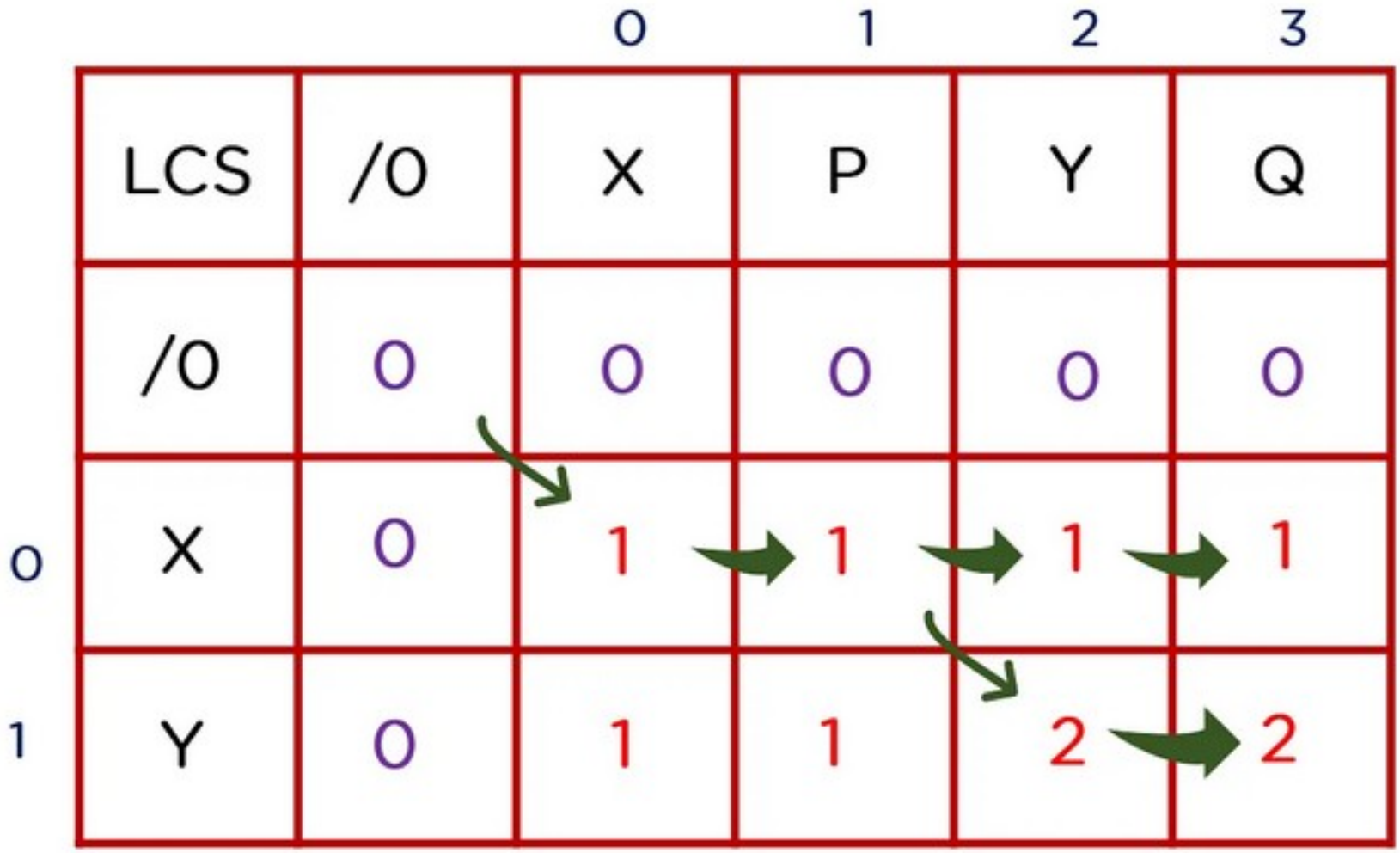
- Step 2: Use the logic given below to fill each cell in the table.
- Step 3: Fill the current cell by adding one to the diagonal element if the character equivalent to the current row and column matches. Also, draw an arrow to the cell on the diagonal.



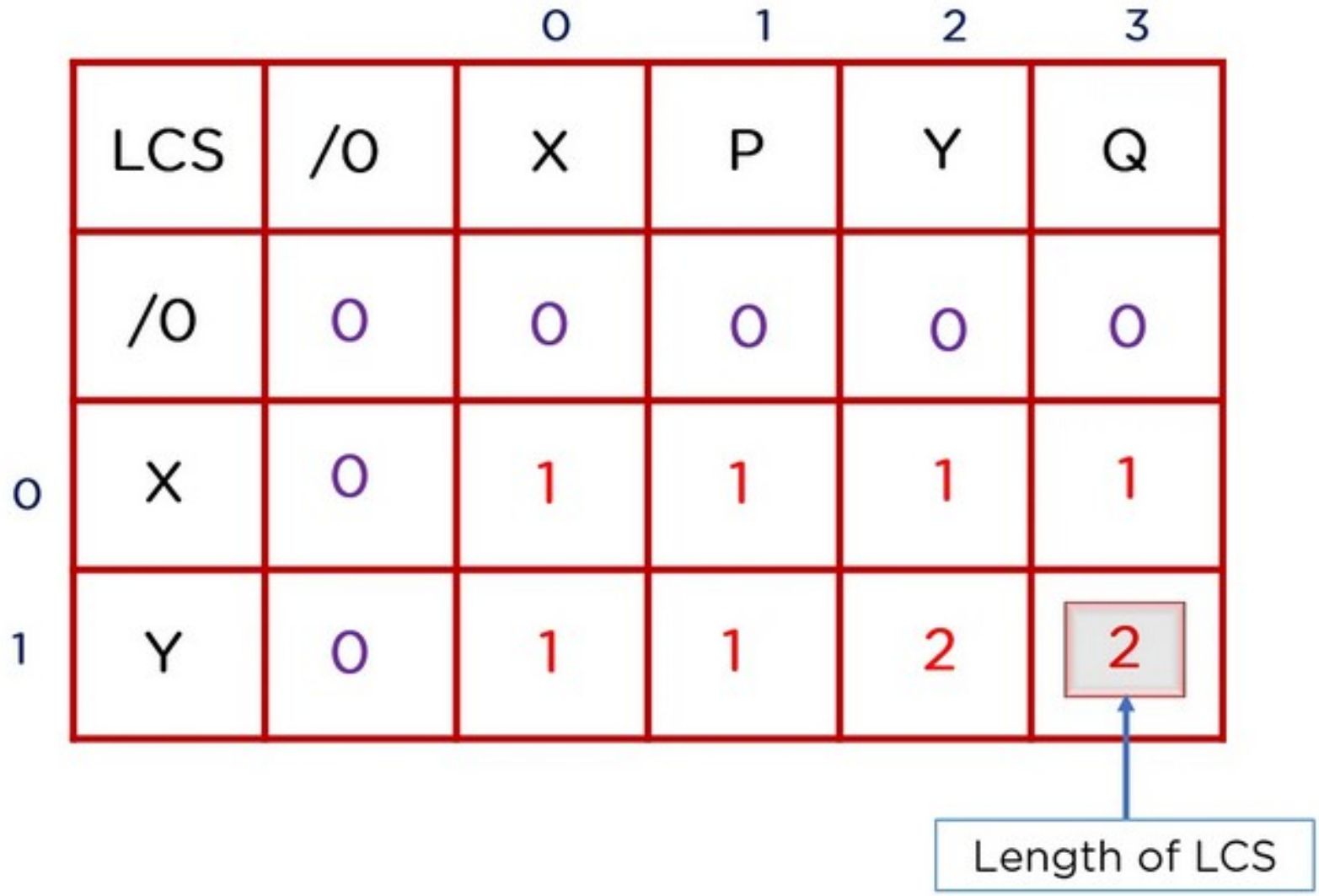
- Step 4: Otherwise, fill the current field with the largest value from the preceding column and row element. Draw an arrow to the cell with the highest value. If they're all the same, you can point at any of them.



- Step 5: Repeat steps from 2 to 4 until the whole table is filled.



- Step 6: The length of the longest common subsequence is reflected by the value in the final row and column.



Largest/Longest Common Subsequence (LCS)

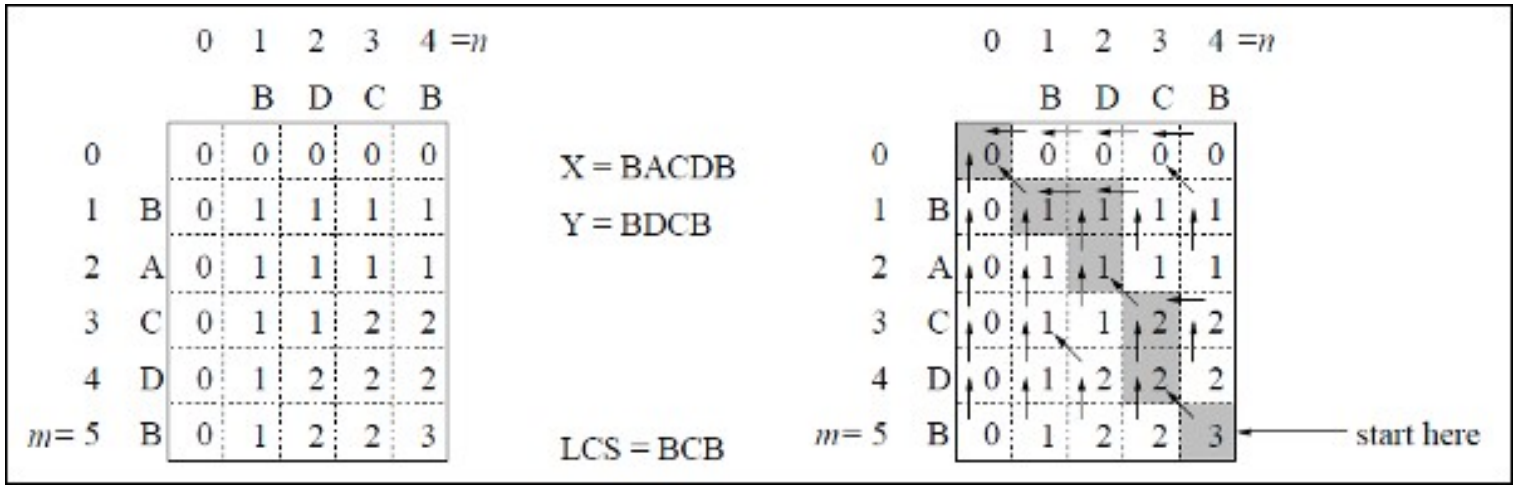
```
Algorithm LCS Length(x,y)

{ c[i,j] is maximum length of array }

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;
              else if c[i-1 ,  j] >= c[i , j-1] then
                    c[i , j] = c[i-1 , j];
            else
                    c[i , j] = c[i , j-1]

Return c and b;
```

Largest/Longest Common Subsequence (LCS)

Example:

Given two strings are $X = BACDB$ and $Y = BDCB$

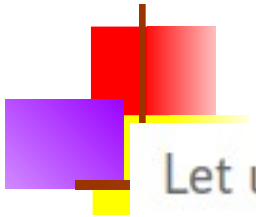
find the longest common subsequence.



Travelling Salesman Problem

Problem: Solve the traveling salesman problem with the associated cost adjacency matrix using dynamic programming.

	1	2	3	4	5
1	-	24	11	10	9
2	8	-	2	5	11
3	26	12	-	8	7
4	11	23	24	-	6
5	5	4	8	11	-



Let us start our tour from city 1.

Step 1: Initially, we will find the distance between city 1 and city {2, 3, 4, 5} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

$$\text{Cost}(2, \Phi, 1) = d[2, 1] = 24$$

$$\text{Cost}(3, \Phi, 1) = d[3, 1] = 11$$

$$\text{Cost}(4, \Phi, 1) = d[4, 1] = 10$$

$$\text{Cost}(5, \Phi, 1) = d[5, 1] = 9$$

Step 2: In this step, we will find the minimum distance by visiting 1 city as intermediate city.

$$Cost\{2, \{3\}, 1\} = d[2, 3] + Cost(3, f, 1)$$

$$= 2 + 11 = 13$$

$$Cost\{2, \{4\}, 1\} = d[2, 4] + Cost(4, f, 1)$$

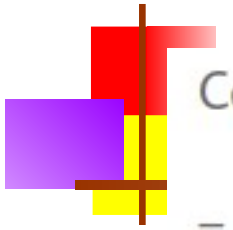
$$= 5 + 10 = 15$$

$$Cost\{2, \{5\}, 1\} = d[2, 5] + Cost(5, f, 1)$$

$$= 11 + 9 = 20$$

$$Cost\{3, \{2\}, 1\} = d[3, 2] + Cost(2, f, 1)$$

$$= 12 + 24 = 36$$



$$Cost\{3, \{4\}, 1\} = d[3, 4] + Cost(4, f, 1)$$

$$= 8 + 10 = 18$$

$$Cost\{3, \{5\}, 1\} = d[3, 5] + Cost(5, f, 1)$$

$$= 7 + 9 = 16$$

$$Cost\{4, \{2\}, 1\} = d[4, 2] + Cost(2, f, 1)$$

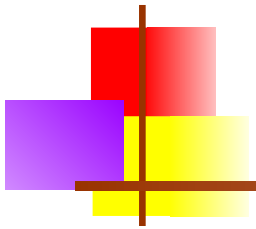
$$= 23 + 24 = 47$$

$$Cost\{4, \{3\}, 1\} = d[4, 3] + Cost(3, f, 1)$$

$$= 24 + 11 = 35$$

$$Cost\{4, \{5\}, 1\} = d[4, 5] + Cost(5, f, 1)$$

$$= 6 + 9 = 15$$



$$Cost\{5, \{2\}, 1\} = d[5, 2] + Cost(2, f, 1)$$

$$= 4 + 24 = 28$$

$$Cost\{5, \{3\}, 1\} = d[5, 3] + Cost(3, f, 1)$$

$$= 8 + 11 = 19$$

$$Cost\{5, \{4\}, 1\} = d[5, 4] + Cost(4, f, 1)$$

$$= 11 + 10 = 21$$

Step 3: In this step, we will find the minimum distance by visiting 2 cities as intermediate city.

$$Cost(2, \{3, 4\}, 1) = \min \{ d[2, 3] + Cost(3, \{4\}, 1), d[2, 4] + Cost(4, \{3\}, 1)]\}$$

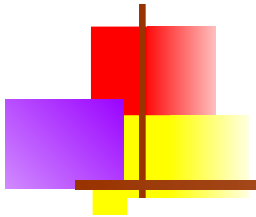
$$= \min \{ [2 + 18], [5 + 35] \}$$

$$= \min\{20, 40\} = 20$$

$$Cost(2, \{4, 5\}, 1) = \min \{ d[2, 4] + Cost(4, \{5\}, 1), d[2, 5] + Cost(5, \{4\}, 1)]\}$$

$$= \min \{ [5 + 15], [11 + 21] \}$$

$$= \min\{20, 32\} = 20$$



$$Cost(2, \{3, 5\}, 1) = \min \{ d[2, 3] + Cost(3, \{4\}, 1),\ d[2, 4] + Cost(4, \{3\}, 1)]\}$$

$$= \min \{ [2 + 18], [5 + 35] \}$$

$$= \min\{20, 40\} = 20$$

$$Cost(3, \{2, 4\}, 1) = \min \{ d[3, 2] + Cost(2, \{4\}, 1),\ d[3, 4] + Cost(4, \{2\}, 1)]\}$$

$$= \min \{ [12 + 15], [8 + 47] \}$$

$$= \min\{27, 55\} = 27$$

$$Cost(3, \{4, 5\}, 1) = \min \{ d[3, 4] + Cost(4, \{5\}, 1),\ d[3, 5] + Cost(5, \{4\}, 1)]\}$$

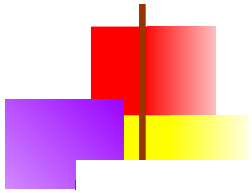
$$= \min \{ [8 + 15], [7 + 21] \}$$

$$= \min\{23, 28\} = 23$$

$$Cost(3, \{2, 5\}, 1) = \min \{ d[3, 2] + Cost(2, \{5\}, 1),\ d[3, 5] + Cost(5, \{2\}, 1)]\}$$

$$= \min \{ [12 + 20], [7 + 28] \}$$

$$= \min\{32, 35\} = 32$$



Cost(4, {2, 3}, 1) = min{ d[4, 2] + Cost(2, {3}, 1), d[4, 3] + Cost(3, {2}, 1)]}

= min { [23 + 13], [24 + 36] }

= min{36, 60} = 36

Cost(4, {3, 5}, 1) = min{ d[4, 3] + Cost(3, {5}, 1), d[4, 5] + Cost(5, {3}, 1)]}

= min { [24 + 16], [6 + 19] }

= min{40, 25} = 25

Cost(4, {2, 5}, 1) = min{ d[4, 2] + Cost(2, {5}, 1), d[4, 5] + Cost(5, {2}, 1)]}

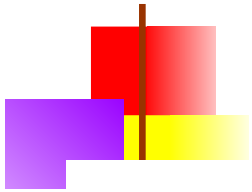
= min { [23 + 20], [6 + 28] }

= min{43, 34} = 34

Cost(5, {2, 3}, 1) = min{ d[5, 2] + Cost(2, {3}, 1), d[5, 3] + Cost(3, {2}, 1)]}

= min { [4 + 13], [8 + 36] }

= min{17, 44} = 17



$$Cost(5, \{3, 4\}, 1) = \min\{ d[5, 3] + Cost(3, \{4\}, 1), d[5, 4] + Cost(4, \{3\}, 1)]\}$$

$$= \min \{ [8 + 18], [11 + 35] \}$$

$$= \min\{26, 46\} = 26$$

$$Cost(5, \{2, 4\}, 1) = \min\{ d[5, 2] + Cost(2, \{4\}, 1), d[5, 4] + Cost(4, \{2\}, 1)]\}$$

$$= \min \{ [4 + 15], [11 + 47] \}$$

$$= \min\{19, 58\} = 19$$

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as intermediate city.

$$Cost(2, \{3, 4, 5\}, 1) = \min \{ d[2, 3] + Cost(3, \{4, 5\}, 1), d[2, 4] + Cost(4, \{3, 5\}, 1), d[2, 5] + Cost(5, \{3, 4\}, 1)\}$$

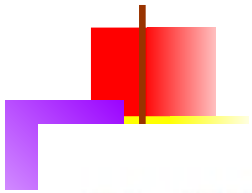
$$= \min \{ 2 + 23, 5 + 25, 11 + 36\}$$

$$= \min\{25, 30, 47\} = 25$$

$$Cost(3, \{2, 4, 5\}, 1) = \min \{ d[3, 2] + Cost(2, \{4, 5\}, 1), d[3, 4] + Cost(4, \{2, 5\}, 1), d[3, 5] + Cost(5, \{2, 4\}, 1)\}$$

$$= \min \{ 12 + 20, 8 + 34, 7 + 19\}$$

$$= \min\{32, 42, 26\} = 26$$



$$Cost(4, \{2, 3, 5\}, 1) = \min \{ d[4, 2] + Cost(2, \{3, 5\}, 1), d[4, 3] + Cost(3, \{2, 5\}, 1), d[4, 5] + Cost(5, \{2, 3\}, 1)\}$$

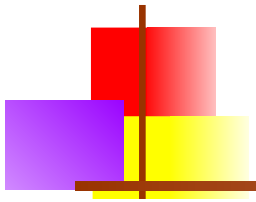
$$= \min \{23 + 30, 24 + 32, 6 + 17\}$$

$$= \min\{53, 56, 23\} = 23$$

$$Cost(5, \{2, 3, 4\}, 1) = \min \{ d[5, 2] + Cost(2, \{3, 4\}, 1), d[5, 3] + Cost(3, \{2, 4\}, 1), d[5, 4] + Cost(4, \{2, 3\}, 1)\}$$

$$= \min \{4 + 30, 8 + 27, 11 + 36\}$$

$$= \min\{34, 35, 47\} = 34$$



Step 5 : In this step, we will find the minimum distance by visiting 4 cities as an intermediate city.

Cost(1, {2, 3, 4, 5}, 1) = min { d[1, 2] + Cost(2, {3, 4, 5}, 1), d[1, 3] + Cost(3, {2, 4, 5}, 1), d[1, 4] + Cost(4, {2, 3, 5}, 1) , d[1, 5] + Cost(5, {2, 3, 4}, 1)}

= min { 24 + 25, 11 + 26, 10 + 23, 9 + 34 }

= min{49, 37, 33, 43} = 33

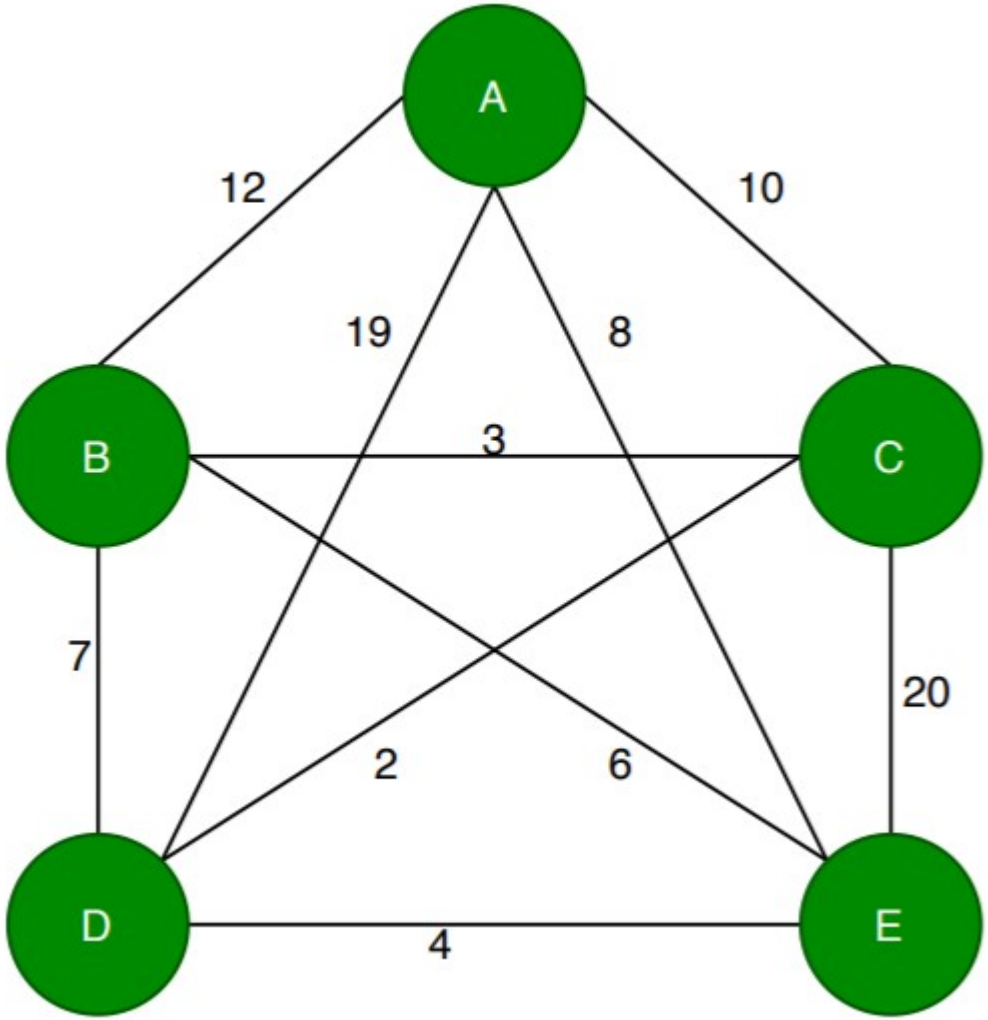
Thus, minimum length tour would be of 33.

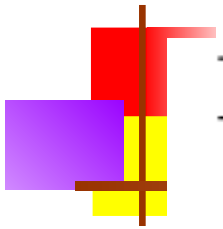
Trace the path:

- Let us find the path that gives the distance of 33.
- Cost(1, {2, 3, 4, 5}, 1) is minimum due to d[1, 4], so move from 1 to 4. Path = {1, 4}.
- Cost(4, {2, 3, 5}, 1) is minimum due to d[4, 5], so move from 4 to 5. Path = {1, 4, 5}.
- Cost(5, {2, 3}, 1) is minimum due to d[5, 2], so move from 5 to 2. Path = {1, 4, 5, 2}.
- Cost(2, {3}, 1) is minimum due to d[2, 3], so move from 2 to 3. Path = {1, 4, 5, 2, 3}.

All cities are visited so come back to 1. Hence the optimum tour would be 1 – 4 – 5 – 2 – 3 – 1.

Solve the given TSP using algorithm on next slide





Algorithm 1: Dynamic Approach for TSP

Data: s: starting point; N: a subset of input cities; $dist()$: distance among the cities

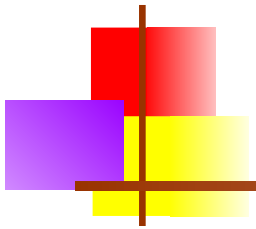
Result: $Cost$: TSP result

$Visited[N] = 0;$

$Cost = 0;$

Procedure TSP(N, s)

```
    Visited[s] = 1;
    if |N| = 2 and k ≠ s then
        Cost(N, k) =dist(s, k);
        Return Cost;
    else
        for j ∈ N do
            for i ∈ N and visited[i] = 0 do
                if j ≠ i and j ≠ s then
                    Cost(N, j) = min ( TSP(N − {i}, j) + dist(j, i))
                    Visited[j] = 1;
                end
            end
        end
    end
    Return Cost;
end
```

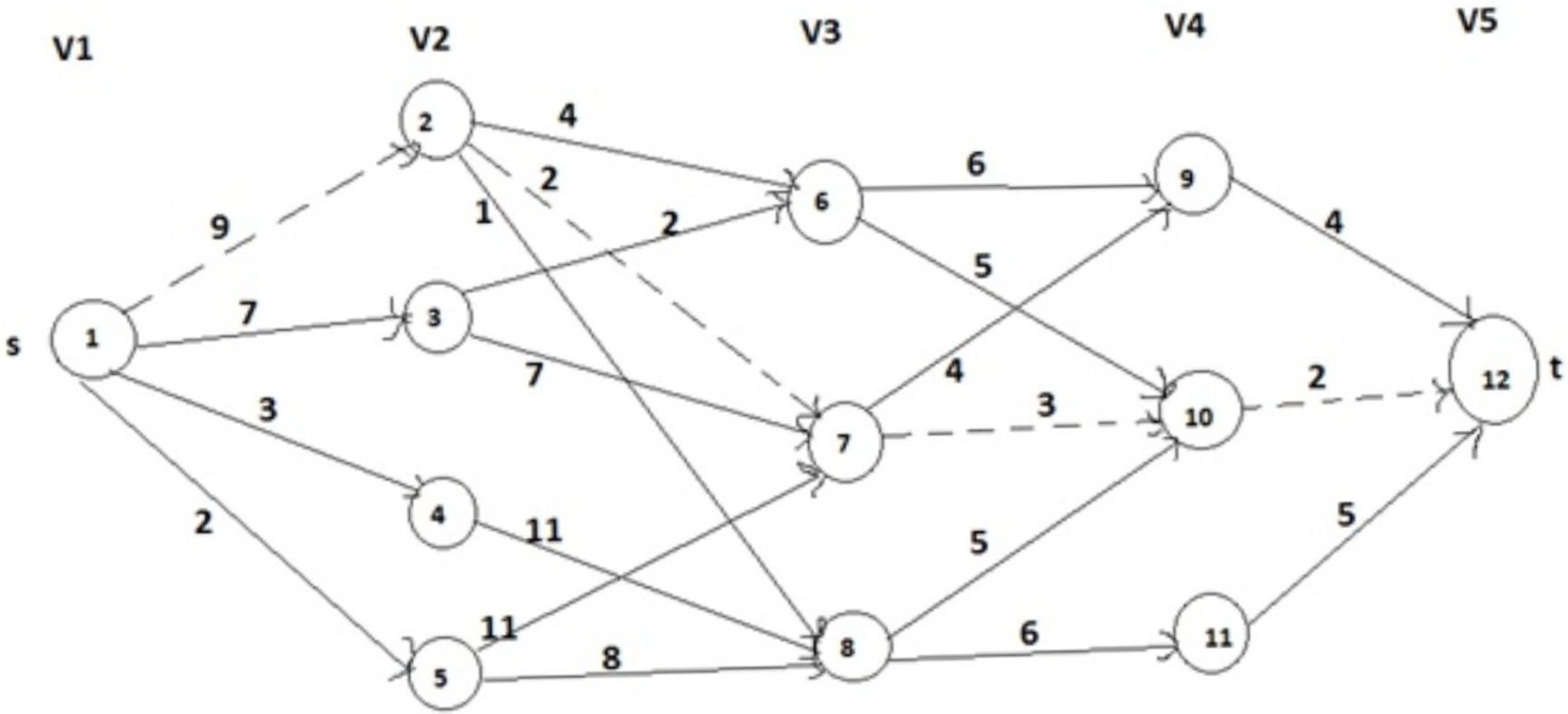


Time complexity of the algorithm on previous slide:

In the dynamic algorithm for TSP, the number of possible subsets can be at most $N*2^N$.

Each subset can be solved in O(N) times.

Therefore, the time complexity of this algorithm would be (N^2*2^N)



MULTI STAGE GRAPH