Greedy Algorithm

Fractional Knapsack

Greedy Algorithm

A greedy algorithm is a mathematical process that looks for simple, easy to complex one, multiply step problem by designed which next step will be most benefits.

Characteristics of greedy algorithm

=> Make a sequence of choices

=> Each choice is the one that seems best so far, only depends on what's been done so far

=> Choice produces a smaller problem to be solved

Fractional Knapsack

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number. So we must consider weights of items as well as their values.

Item	1	2	3
Value	8	6	5
Weight	1	3	5

Fractional Knapsack

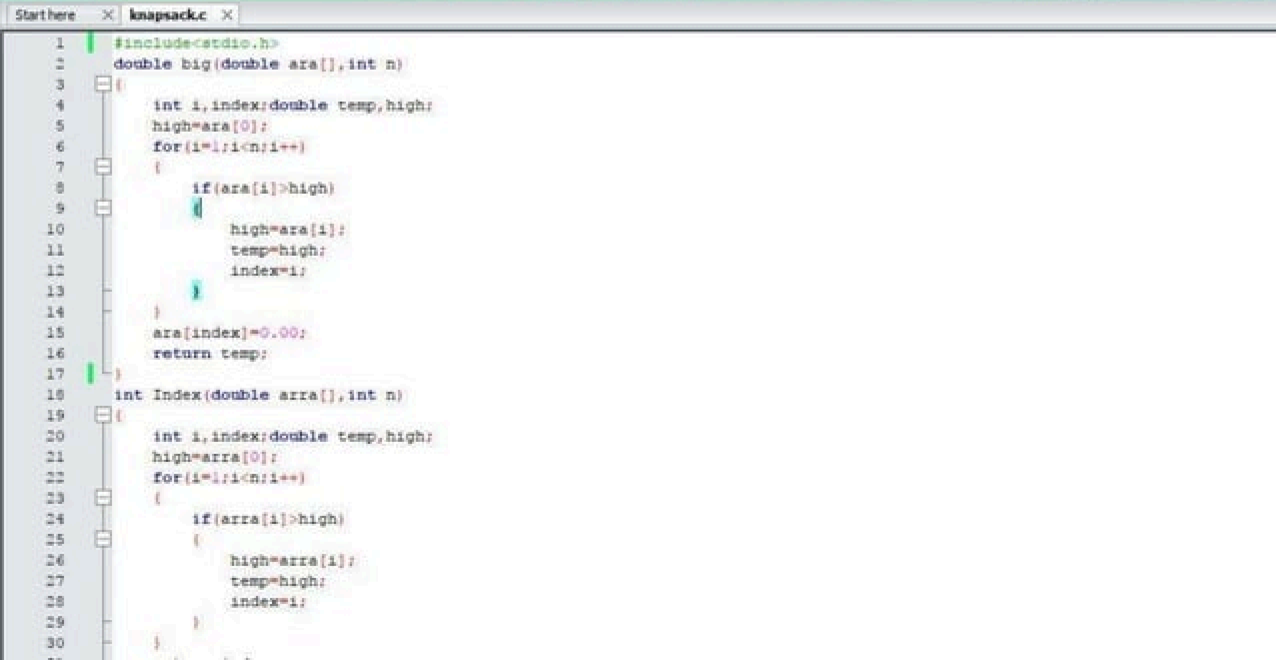
Pseudo code:

Input: Set S of items w, benefits bi and weight w_i, Weight W

Output: Amount x_i of each item *i* to maximize benefits

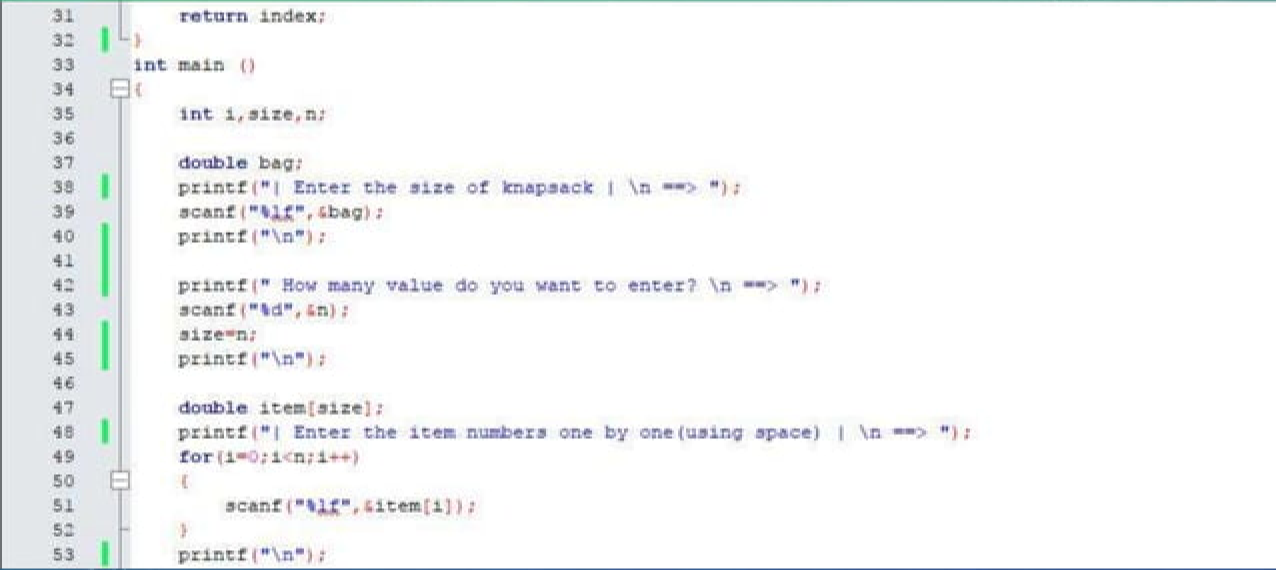
```
$X_i$ = 0, w = 0
For each item i in S
$V_i = \frac{b_i}{w_i}$
While w<W
//remove item i with highest $V_i$
$X_i$ = min ($w_i$, W − w)
w = w + min ($w_i$, W − w)
```

Code Implementation (Part-1)



```c
#include<stdio.h>
double big(double ara[],int n)
{
    int i,index;double temp,high;
    high=ara[0];
    for(i=1;i<n;i++)
    {
        if(ara[i]>high)
        {
            high=ara[i];
            temp=high;
            index=i;
        }
    }
    ara[index]=0.00;
    return temp;
}
int Index(double arra[],int n)
{
    int i,index;double temp,high;
    high=arra[0];
    for(i=1;i<n;i++)
    {
        if(arra[i]>high)
        {
            high=arra[i];
            temp=high;
            index=i;
        }
    }
```

Code Implementation (Part-2)



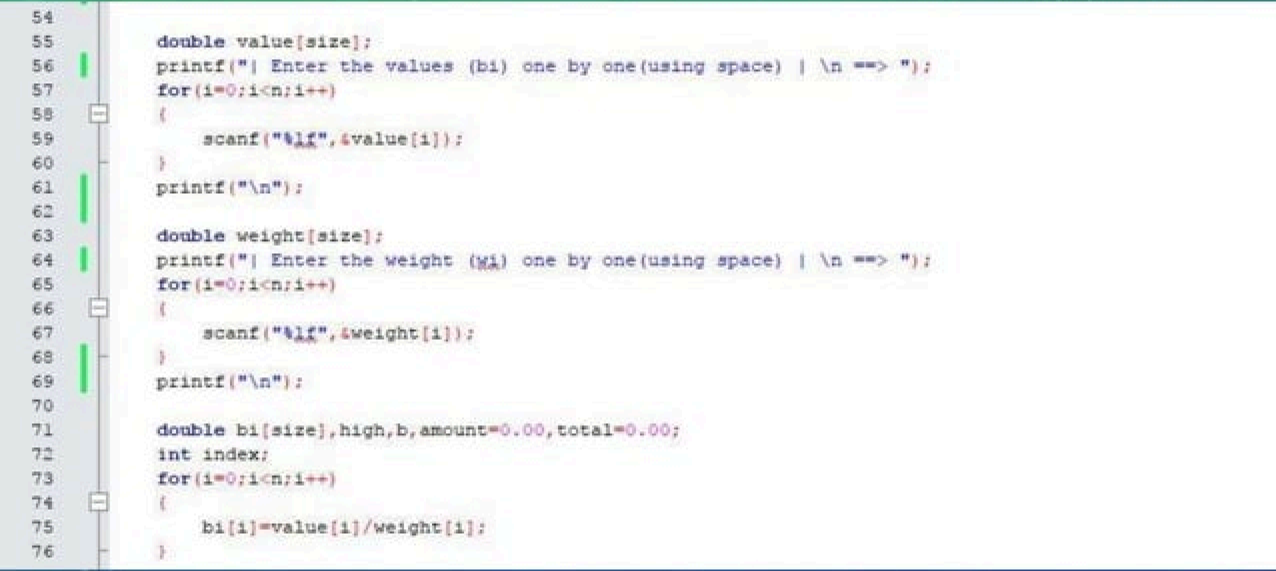
```
    return index;
}
int main ()
{
    int i,size,n;

    double bag;
    printf("| Enter the size of knapsack | \n ==> ");
    scanf("%lf",&bag);
    printf("\n");

    printf(" How many value do you want to enter? \n ==> ");
    scanf("%d",&n);
    size=n;
    printf("\n");

    double item[size];
    printf("| Enter the item numbers one by one(using space) | \n ==> ");
    for(i=0;i<n;i++)
    {
        scanf("%lf",&item[i]);
    }
    printf("\n");
```

Code Implementation (Part-3)

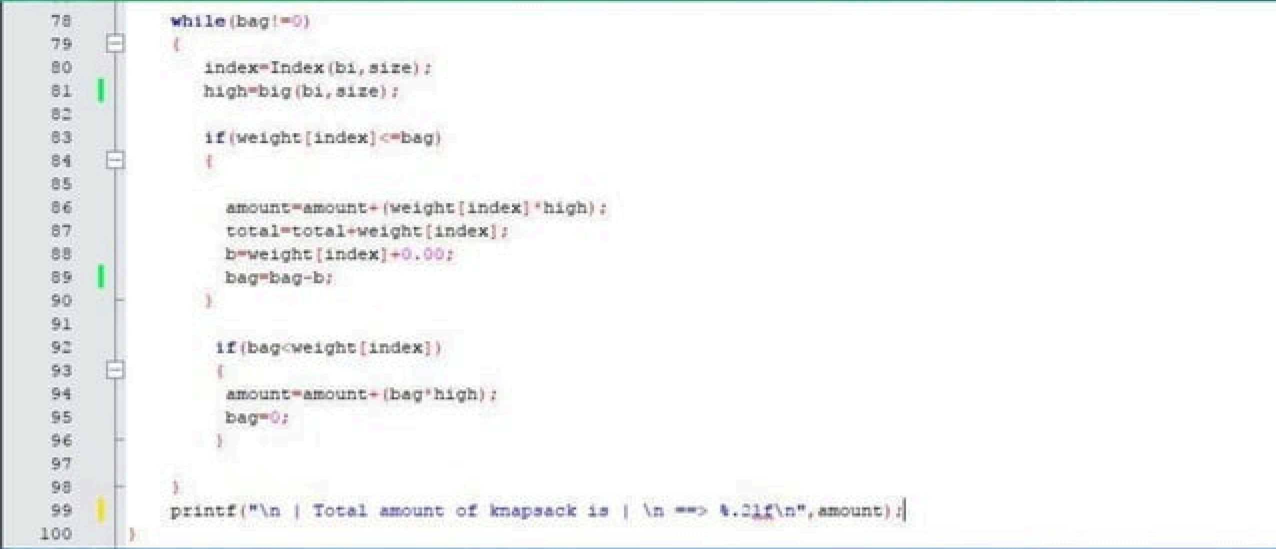


```
    double value[size];
    printf("| Enter the values (bi) one by one(using space) | \n ==> ");
    for(i=0;i<n;i++)
    {
        scanf("%lf",&value[i]);
    }
    printf("\n");

    double weight[size];
    printf("| Enter the weight (wi) one by one(using space) | \n ==> ");
    for(i=0;i<n;i++)
    {
        scanf("%lf",&weight[i]);
    }
    printf("\n");

    double bi[size],high,b,amount=0.00,total=0.00;
    int index;
    for(i=0;i<n;i++)
    {
        bi[i]=value[i]/weight[i];
    }
```

Code Implementation (Part-4)



```
    while(bag!=0)
    {
       index=Index(bi,size);
       high=big(bi,size);

       if(weight[index]<=bag)
       {

         amount=amount+(weight[index]*high);
         total=total+weight[index];
         b=weight[index]+0.00;
         bag=bag-b;
       }

        if(bag<weight[index])
        {
         amount=amount+(bag*high);
         bag=0;
        }

    }
    printf("\n | Total amount of knapsack is | \n ==> %.2lf\n",amount);
}
```

Example:

Items	1	2	3	4	5
Value (b_i)	16	32	17	32	40
Weight (w_i)	5	6	2	6	4
$X_i = \frac{b_i}{w_i}$	3.2	5.3	8.5	5.3	10

Here, The size of knapsack is = 13 kg

Item 5: 4 kg = (4x10) tk = 40 tk

Remaining (13 - 4) kg = 9 kg

Item 3: 2 kg = (2x8.5) tk = 17 tk

Remaining (9 - 2) kg = 7 kg

Item 2: 6 kg = (6x5.3) tk = 32 tk

Remaining (7 - 6) kg = 1 kg

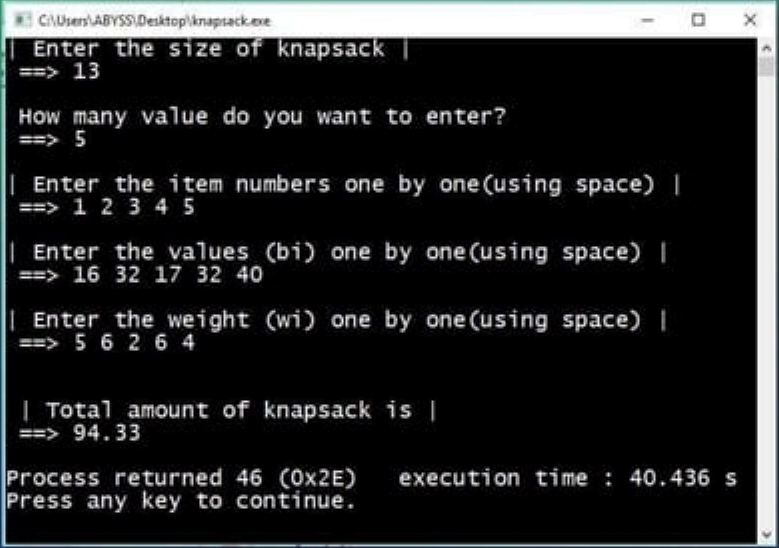
Item 4: 1 kg = (1x5.3) tk =5.3 tk

Remaining (1 - 1) kg = 0 kg

Total amount of knapsack is = (40+17+32+5.3) tk

= 94.3 tk

Code Implementation Output Screenshot:



```
| Enter the size of knapsack |
 ==> 13

 How many value do you want to enter?
 ==> 5

| Enter the item numbers one by one(using space) |
 ==> 1 2 3 4 5

| Enter the values (bi) one by one(using space) |
 ==> 16 32 17 32 40

| Enter the weight (wi) one by one(using space) |
 ==> 5 6 2 6 4


 | Total amount of knapsack is |
 ==> 94.33

Process returned 46 (0x2E)   execution time : 40.436 s
Press any key to continue.
```

Dynamic Programming

0/1 Knapsack

0 / 1 knapsack algorithm

The knapsack problem or rucksack problem is a problem in combinatorial optimization: given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.

Dynamic programming

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler sub problems, solving each of those sub problems just once, and storing their solutions.

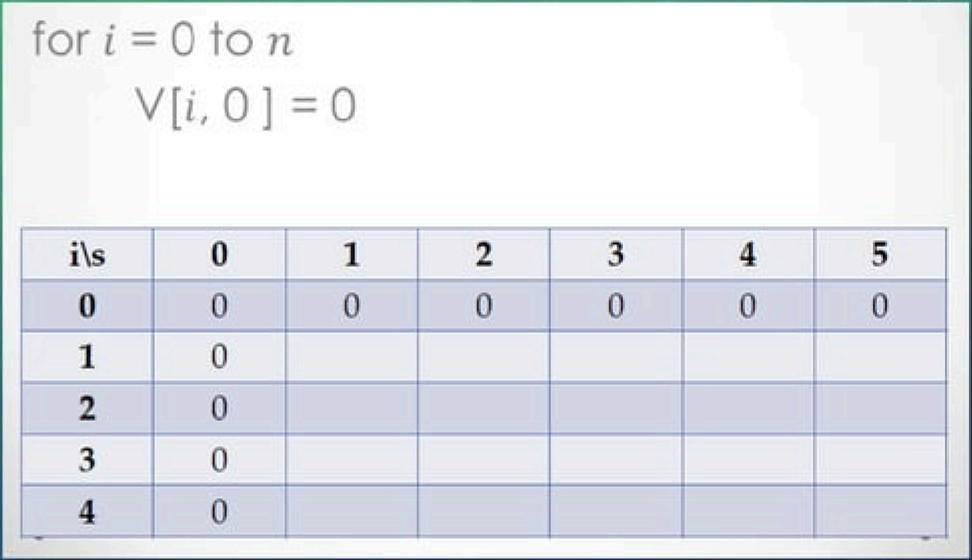
Example

Number	01	02	03	04
Weight	02	03	04	05
Value	03	04	05	06

i\s	0	1	2	3	4	5
0						
1						
2						
3						
4						

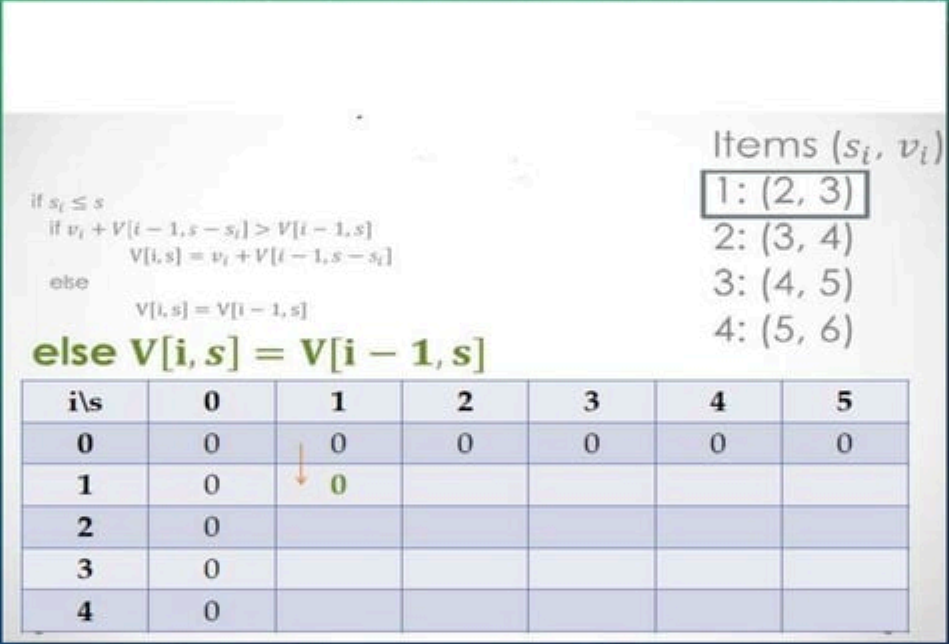
```
for $s$ = 0 to $S$
        V[ 0, $s$ ] = 0
```

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						



```
for $i = 0$ to $n$
    V[$i$, 0 ] = 0
```

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					



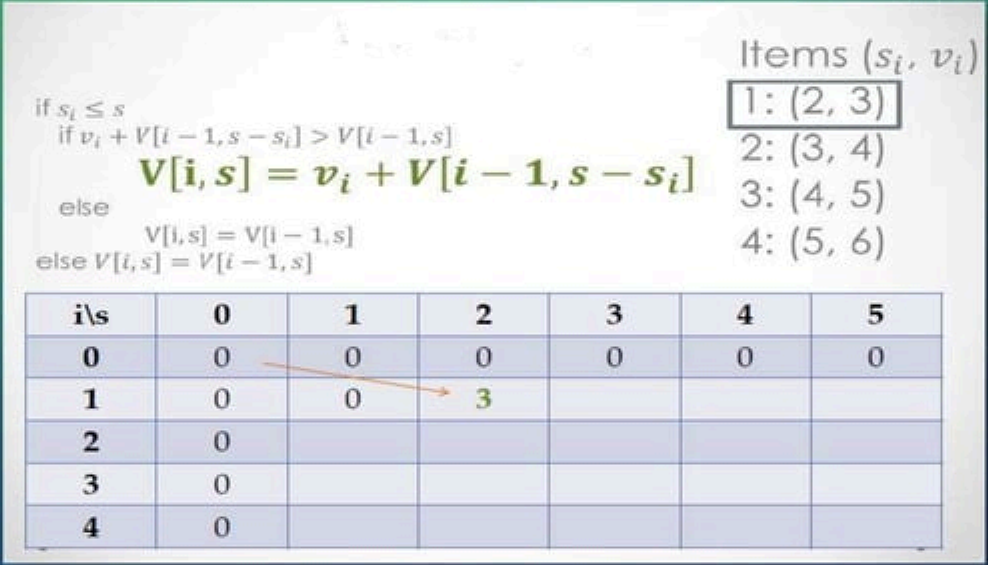
```
if $s_i \le s$
    if $v_i + V[i-1, s-s_i] > V[i-1, s]$
            $V[i, s] = v_i + V[i-1, s-s_i]$
    else
            $V[i, s] = V[i-1, s]$
```

Items (s_i, v_i)

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

else $\mathbf{V[i, s] = V[i-1, s]}$

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	↓ **0**				
2	0					
3	0					
4	0					



```
if $s_i \le s$
  if $v_i + V[i-1, s-s_i] > V[i-1, s]$
        $\mathbf{V[i,s] = v_i + V[i-1, s-s_i]}$
  else
        $V[i,s] = V[i-1,s]$
else $V[i,s] = V[i-1,s]$
```

Items (s_i, v_i)

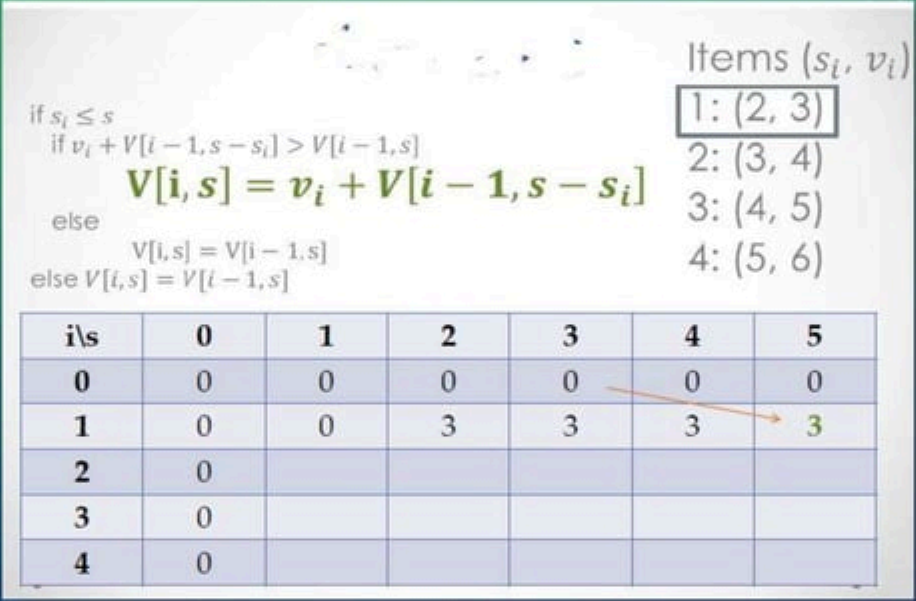
$\boxed{1\text{: }(2, 3)}$

2: (3, 4)

3: (4, 5)

4: (5, 6)

i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					



```
if $s_i \le s$
  if $v_i + V[i-1, s-s_i] > V[i-1, s]$
      $\mathbf{V[i,s]} = v_i + V[i-1, s-s_i]$
  else
      $V[i,s] = V[i-1,s]$
else $V[i,s] = V[i-1,s]$
```

Items (s_i, v_i)

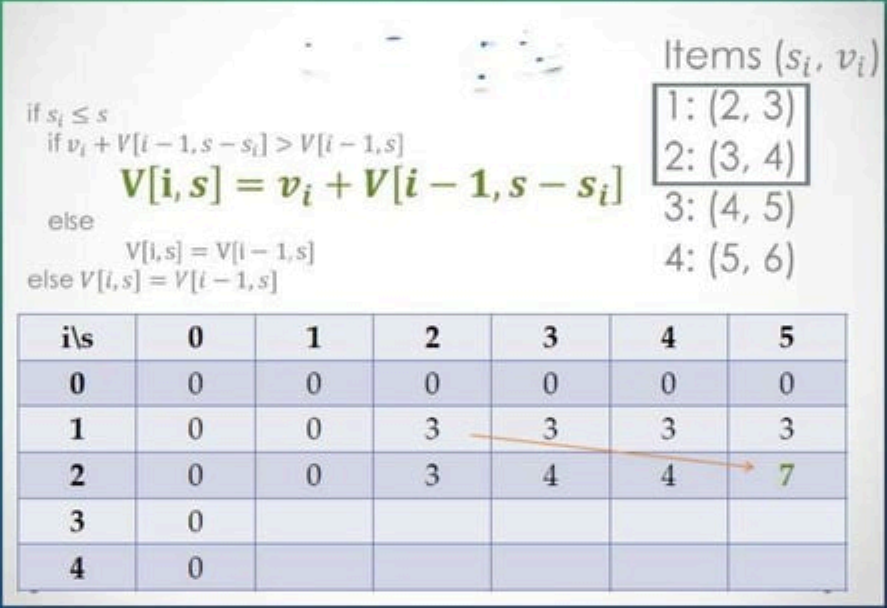
$\boxed{1: (2, 3)}$

2: (3, 4)

3: (4, 5)

4: (5, 6)

i\s	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					



```
if $s_i \le s$
  if $v_i + V[i-1, s-s_i] > V[i-1, s]$
        $V[i,s] = v_i + V[i-1, s-s_i]$
  else
        $V[i,s] = V[i-1,s]$
else $V[i,s] = V[i-1,s]$
```

Items (s_i, v_i)

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

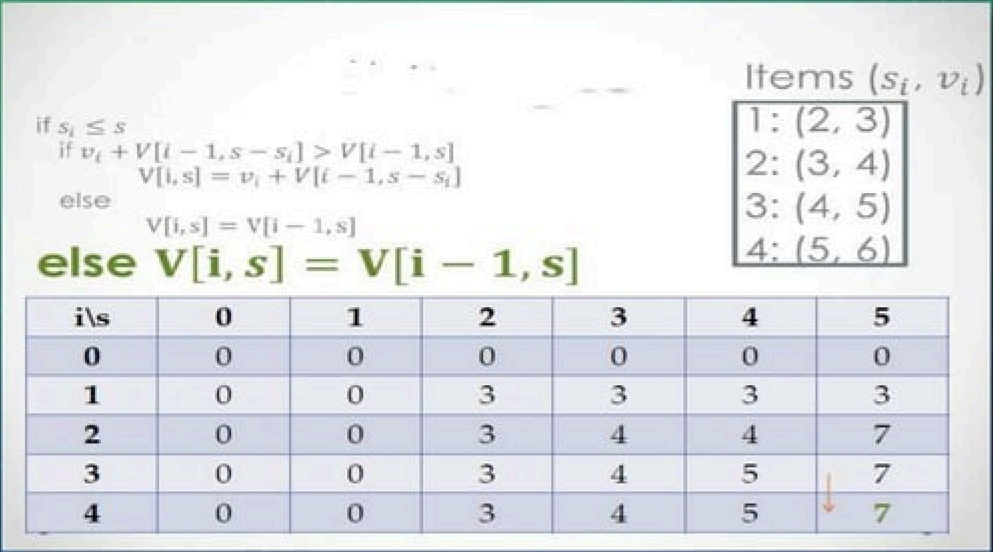
i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

```
if $s_i \le s$
    if $v_i + V[i-1, s-s_i] > V[i-1, s]$
            $V[i,s] = v_i + V[i-1, s-s_i]$
    else
            $V[i,s] = V[i-1, s]$
else $V[i,s] = V[i-1, s]$
```

Items (s_i, v_i)

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

i\s	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					



```
if $s_i \le s$
  if $v_i + V[i-1, s-s_i] > V[i-1, s]$
        $V[i,s] = v_i + V[i-1, s-s_i]$
  else
        $V[i,s] = V[i-1, s]$
else $V[i, s] = V[i-1, s]$
```

Items (s_i, v_i)

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

i\s	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

Items (s_i, v_i)

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

```
$i = n, k = S$
while $i, k > 0$
if V$[i, k] \neq V[1 - i, k]$
        $i = i - 1, k = k - s_i$
else
        $i = i - 1$
```

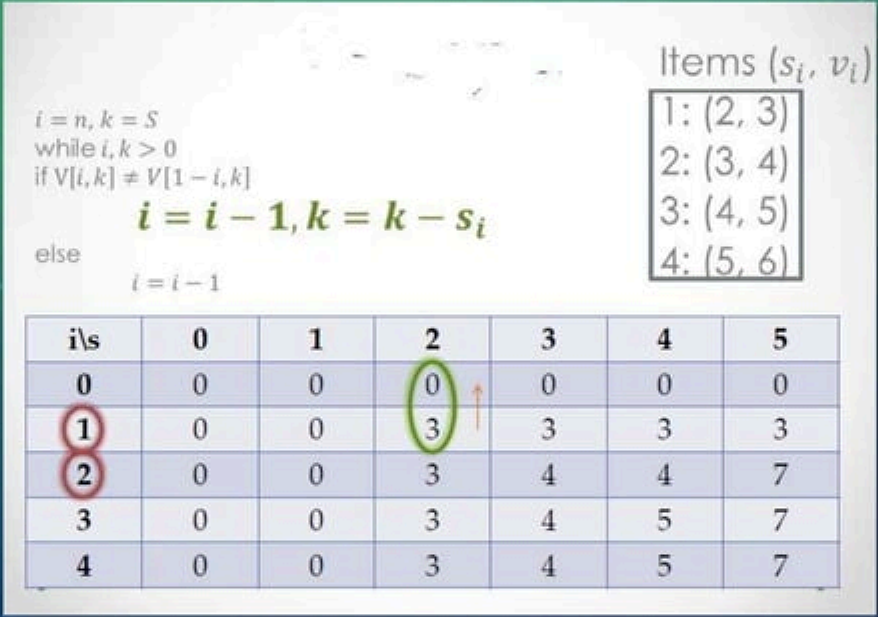
i\s	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

```
$i = n, k = S$
while $i, k > 0$
if $V[i, k] \neq V[1 - i, k]$
        $i = i - 1, k = k - s_i$
else
        $i = i - 1$
```

Items (s_i, v_i)

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

i\s	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



Items (s_i, v_i)

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

```
$i = n, k = S$
while $i, k > 0$
if V$[i,k] \neq V[1-i,k]$
        $\boldsymbol{i = i - 1, k = k - s_i}$
else
        $i = i - 1$
```

i\s	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

Longest Common Subsequence (LCS)

Definition

Longest common subsequence (LCS) of 2 sequences is a subsequence, with maximal length, which is common to both the sequences. Given two sequence of integers, and , find any one longest common subsequence.

Dynamic programming
1) It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
2) Algorithm finds solutions to subproblems and stores them in memory for later use
3) More efficient than "*brute-force methods*", which solve the same subproblems over and over again

LCS Length Algorithm

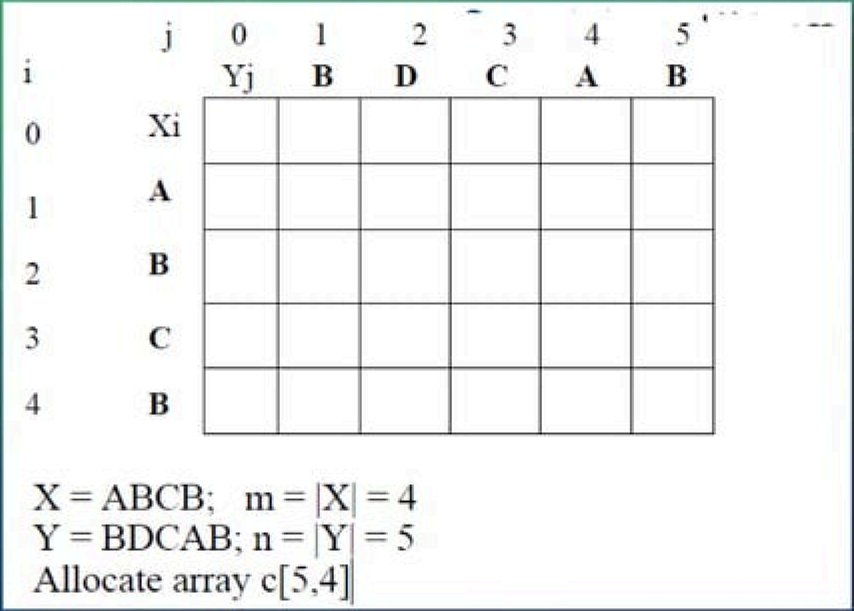
```
LCS-Length(X, Y)
1. m = length(X) // get the # of symbols in X
2. n = length(Y) // get the # of symbols in Y
3. for i = 1 to m c[i,0] = 0 // special case: Y0
4. for j = 1 to n c[0,j] = 0 // special case: X0
5. for i = 1 to m // for all Xi
6. for j = 1 to n // for all Yj
7. if ( Xi == Yj )
8. c[i,j] = c[i-1,j-1] + 1
9. else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

LCS Example

We'll see how LCS algorithm works on the following example:

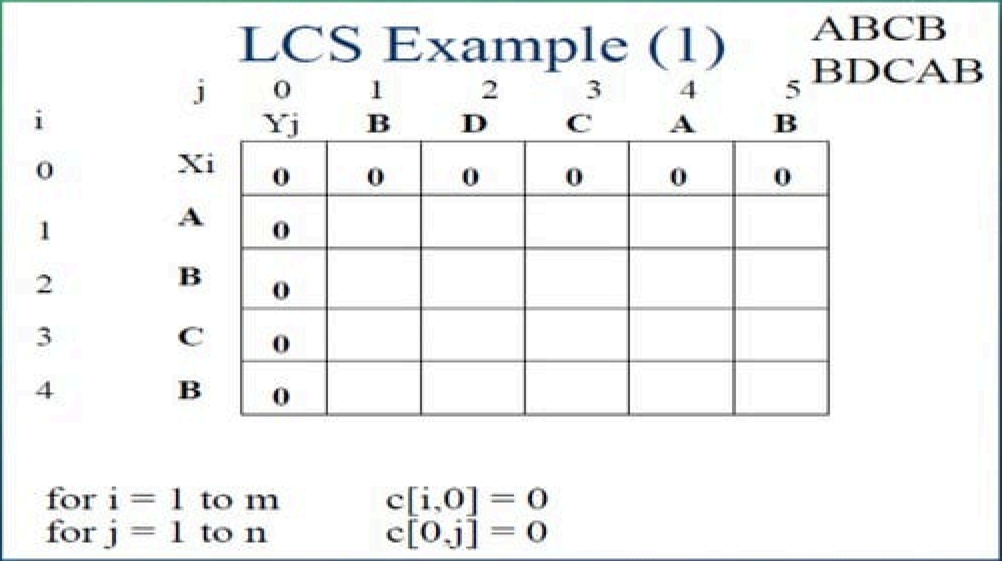
X = ABCB

Y = BDCAB



	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

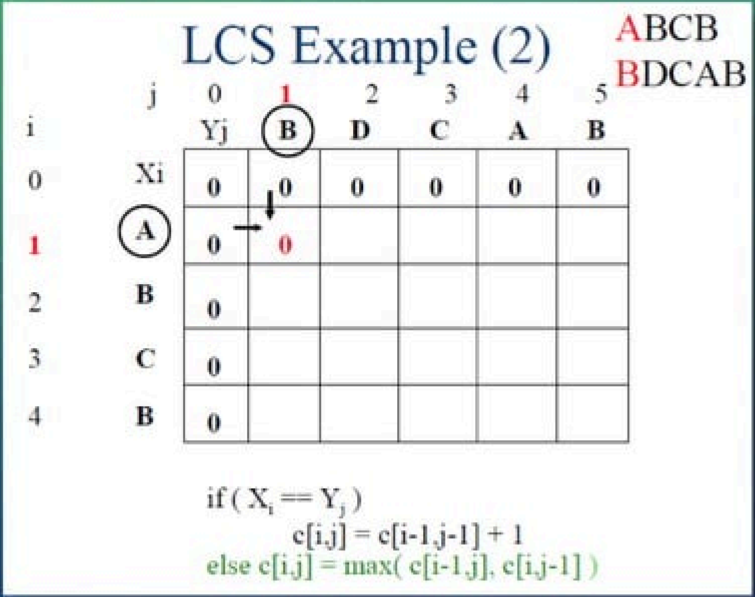


LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

```
for i = 1 to m        c[i,0] = 0
for j = 1 to n        c[0,j] = 0
```

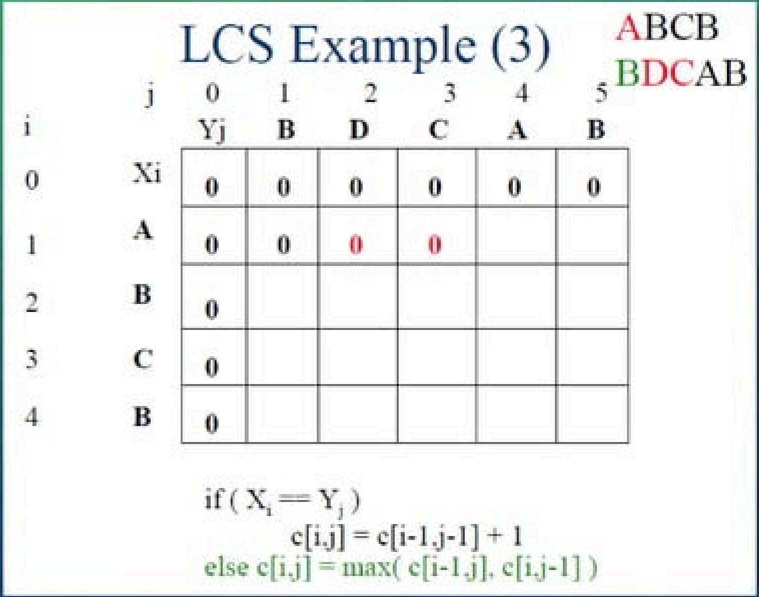


LCS Example (2)

ABCB
BDCAB

i		j	0	1	2	3	4	5
i			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0				
2	B		0					
3	C		0					
4	B		0					

```
if ( $X_i == Y_j$ )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

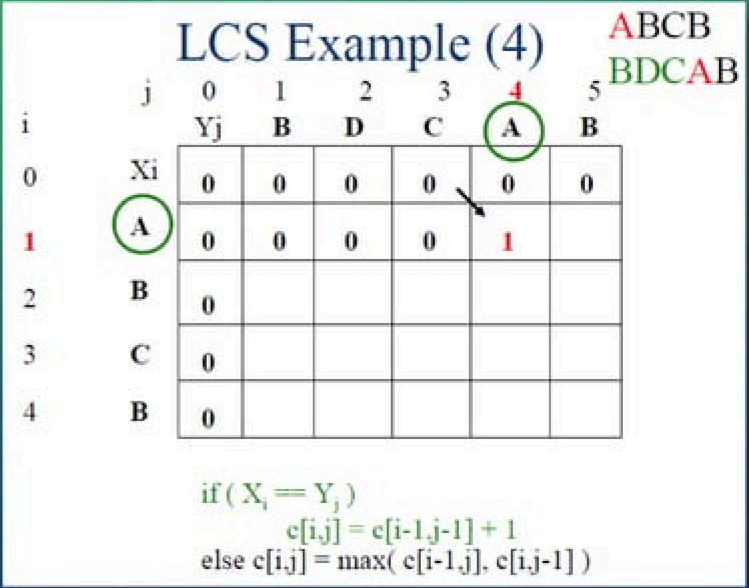


LCS Example (3)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi	**0**	**0**	**0**	**0**	**0**	**0**
1	**A**	**0**	**0**	**0**	**0**		
2	**B**	**0**					
3	**C**	**0**					
4	**B**	**0**					

```
if ( $X_i == Y_j$ )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

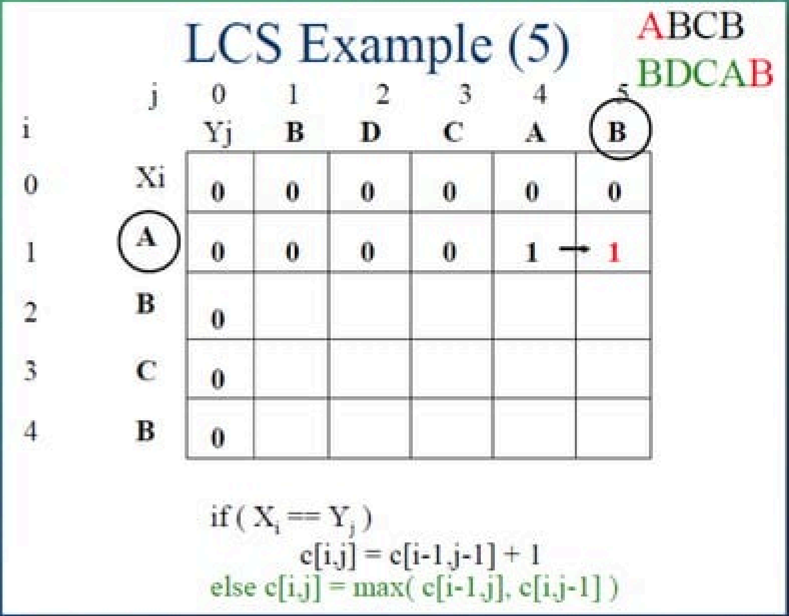


LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

```
if ( $X_i == Y_j$ )
        $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```

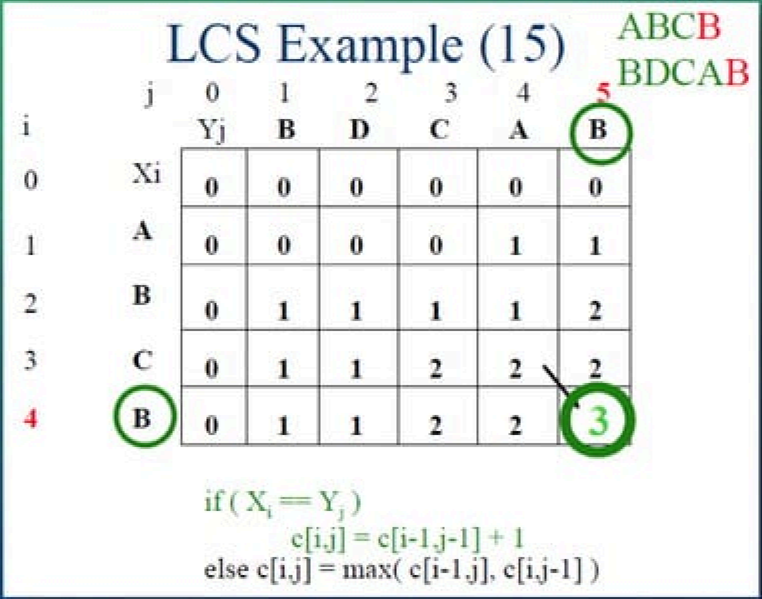


LCS Example (5)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1 →	1
2	B	0					
3	C	0					
4	B	0					

```
if ( $X_i$ == $Y_j$ )
        c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max( c[i-1,j], c[i,j-1] )
```

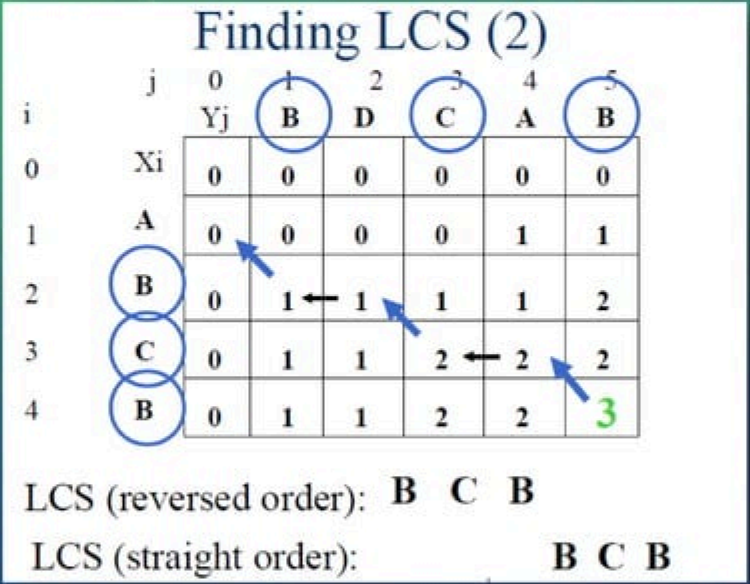


LCS Example (15)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

```
if ( $X_i == Y_j$ )
        $c[i,j] = c[i-1,j-1] + 1$
else $c[i,j] = \max( c[i-1,j], c[i,j-1] )$
```



Finding LCS (2)

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

LCS (reversed order): **B C B**

LCS (straight order): **B C B**

Huffman Code

ALGORITHM

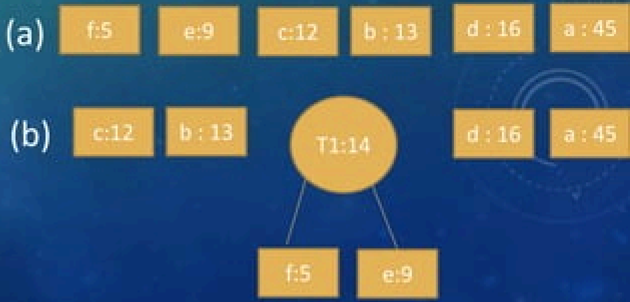
- n = |C|
- Q = C
- for i =1 to n-1
- do allocate a new node z
- left [z] = x = EXTRACT-MIN (Q)
- right [z] = y = EXTRACT-MIN (Q)
- f[z] = f[x] +f[y]
- INSERT(Q, Z)
- return EXTRACT-MIN (Q)

MAJOR STEPS

- 1. Prepare the frequency table
- 2. Construct the binary tree.
- 3. Extract the Huffman Code from the tree.

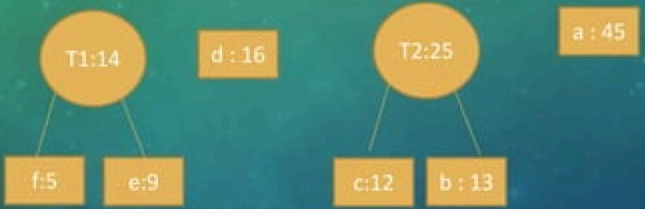
a	b	c	d	e	f
45	13	12	16	9	5

Frequency Table

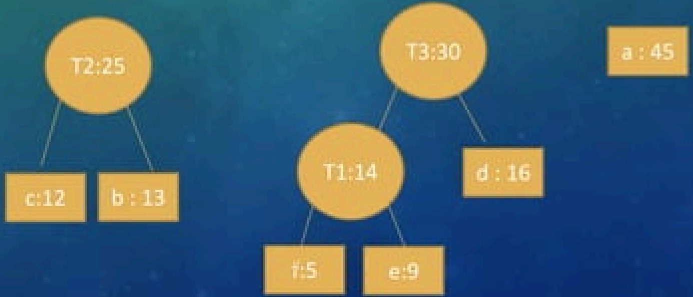


MAJOR STEPS

(c)

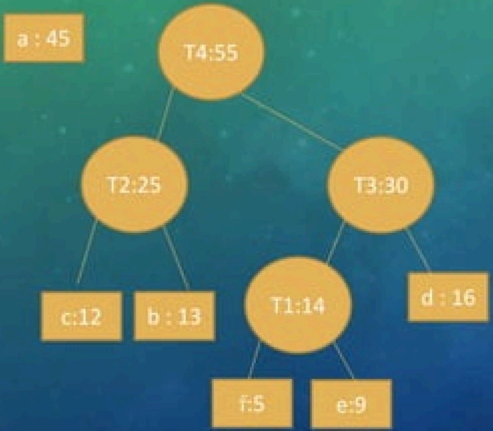


(d)

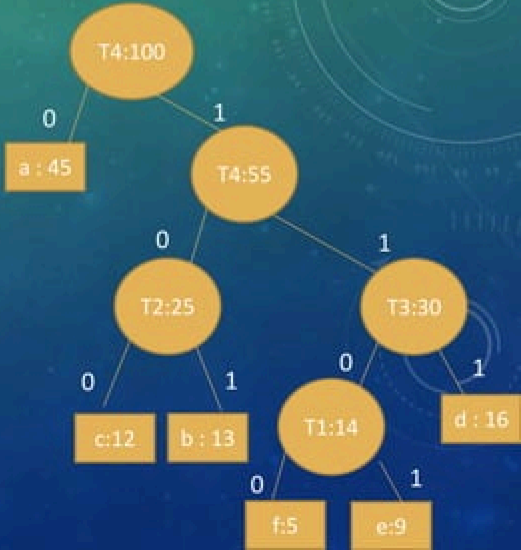


MAJOR STEPS

(e)



(f)



Thank you!