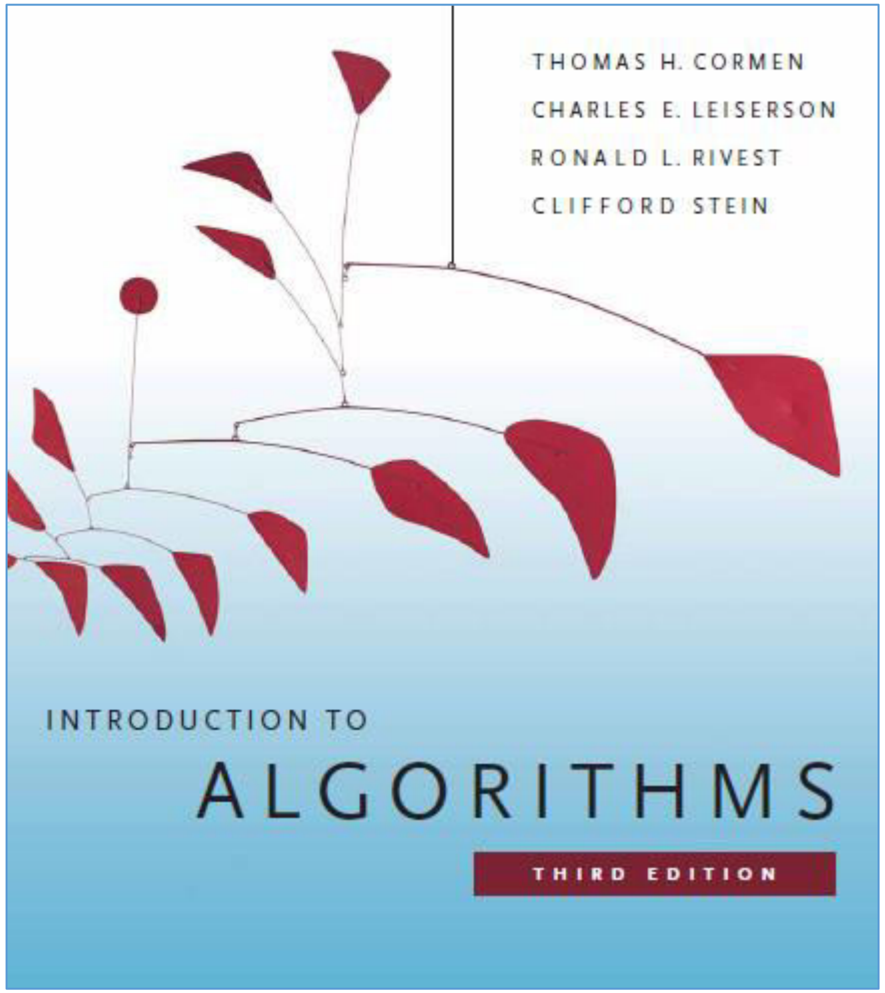
Algorithms Dr. Md Nakib Hayat Chowdhury



Dynamic Programming

- LCS

Dynamic Programming (DP)

- Dynamic Programming is a strategy for designing algorithm.
  - » A meta-technique, not an algorithm (like divide & conquer)
  - » “Programming” in this context refers to a tabular method
  - » not to writing computer code.

Basic Principle

- Used when problem breaks down into recurring small subproblems
- Used when the solution can be recursively described in terms of solutions to subproblems (optimal substructure).
- Algorithm finds solutions to subproblems and stores them in memory for later use.

Dynamic Programming vs Divide and Conquer

Dynamic programming

Dynamic programming solves problems by combining the solutions to sub-problems.

Dynamic Programming vs Divide and Conquer

Are they same?

Dynamic Programming vs Divide and Conquer

- No!
- Divide-and-conquer algorithms partition the problem into **disjoint subproblems**, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- In contrast, dynamic programming applies when the **subproblems overlap**, that is, when subproblems share sub-subproblems.

Dynamic Programming vs Divide and Conquer

Overlapping Subproblems

- In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common sub-subproblems.
- A dynamic-programming algorithm solves each sub-subproblem **just once** and then saves its answer in a table.
- Avoids the work of re-computing the answer every time it solves each sub-subproblem.

Dynamic Programming: When Apply?

We typically apply dynamic programming to **optimization problems**.

- Such problems can have many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Using Dynamic Programming for Optimal Solution

- There are two approach to develop a dynamic programming solution:

  - **Top-Down (Memoization):** In this **recursive** approach, we write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level; if not, the procedure computes the value in the usual manner. Here we start from the main problem and then break it down.

  - **Bottom-Up (Tabulation):** In this **iterative** approach, we start from the base case and build up the solution. When solving a particular subproblem, we have already solved all of the smaller subproblems its solution depends upon, and we have saved their solutions. We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.

Dynamic Programing

Every DP problem can be solved in two ways

- Iteratively
- Recursively

Longest Common Subsequence

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

Longest Common Subsequence

- What are the Subsequences of “ABC”
- Total way = 2^n
- Common Subsequence between two string and LCS

Examples:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

- Brut force approach (if n > 25)

LCS : “HELLOM” and “HMLLD”

- Start with first element

HELLOM

HMLLD

- If they are same: include them into LCS

LCS : “HELLOM” and “HMLLD”

- Recursively solve the rest string

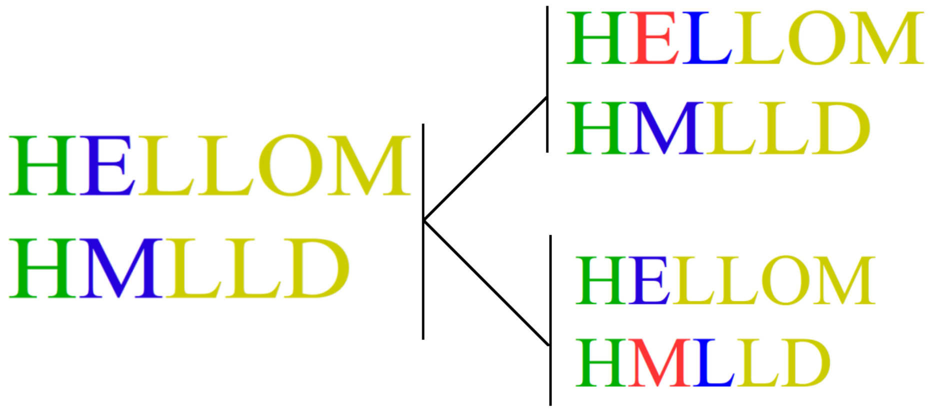
HELLOM

HMLLD

- We have sorter strings “ELLOM” and “MLLD”. Solve it recursively.
- Here E and M are not same. So, we have to drop one.

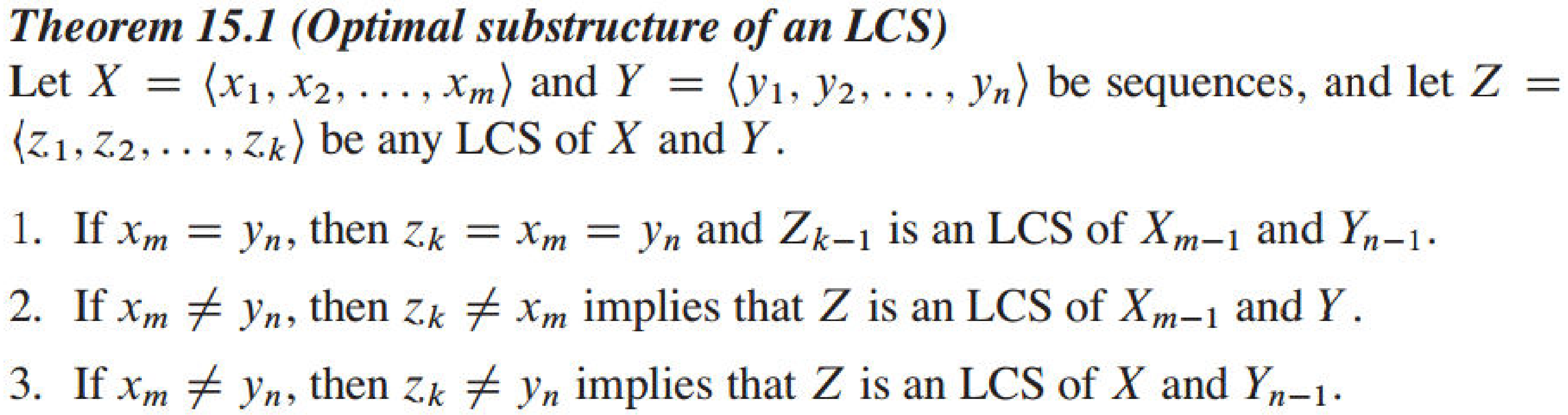
LCS : “HELLOM” and “HMLLD”

- Once we have to drop E and next we drop M.



- LCS(“ELLOM”, “MLLD”) ->Max[LCS(“LLOM”, “MLLD”) and LCS(“ELLOM”, “LLD”)]

Longest Common Subsequence



Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Longest Common Subsequence: Base Case

- Let, x[m] represented by x[i] where i is 1 to m.
- Let, y[n] represented by y[j] where j is 1 to n.
- C[i, j] represents LCS.
- If i or j is 0, then, one of the sequences has length 0, and so the LCS has length 0.

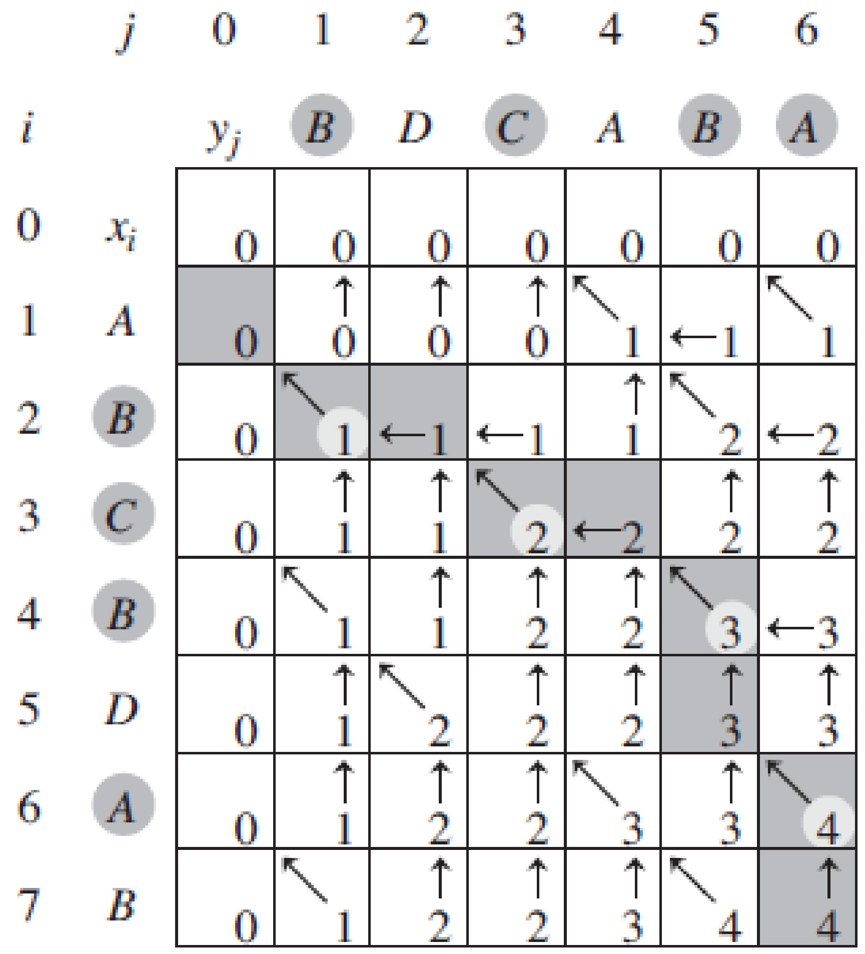
Longest Common Subsequence: Base Case

- Let, x[m] represented by x[i] where i is 1 to m.
- Let, y[n] represented by y[j] where j is 1 to n.
- C[i, j] represents LCS.

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

Longest Commc

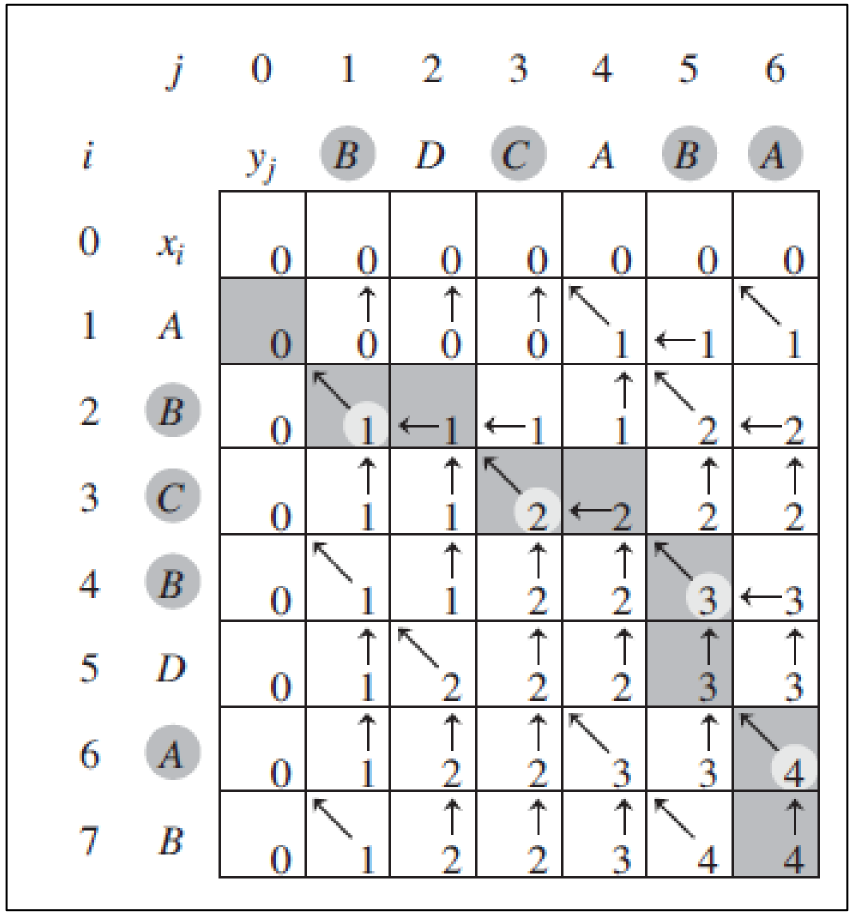
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$



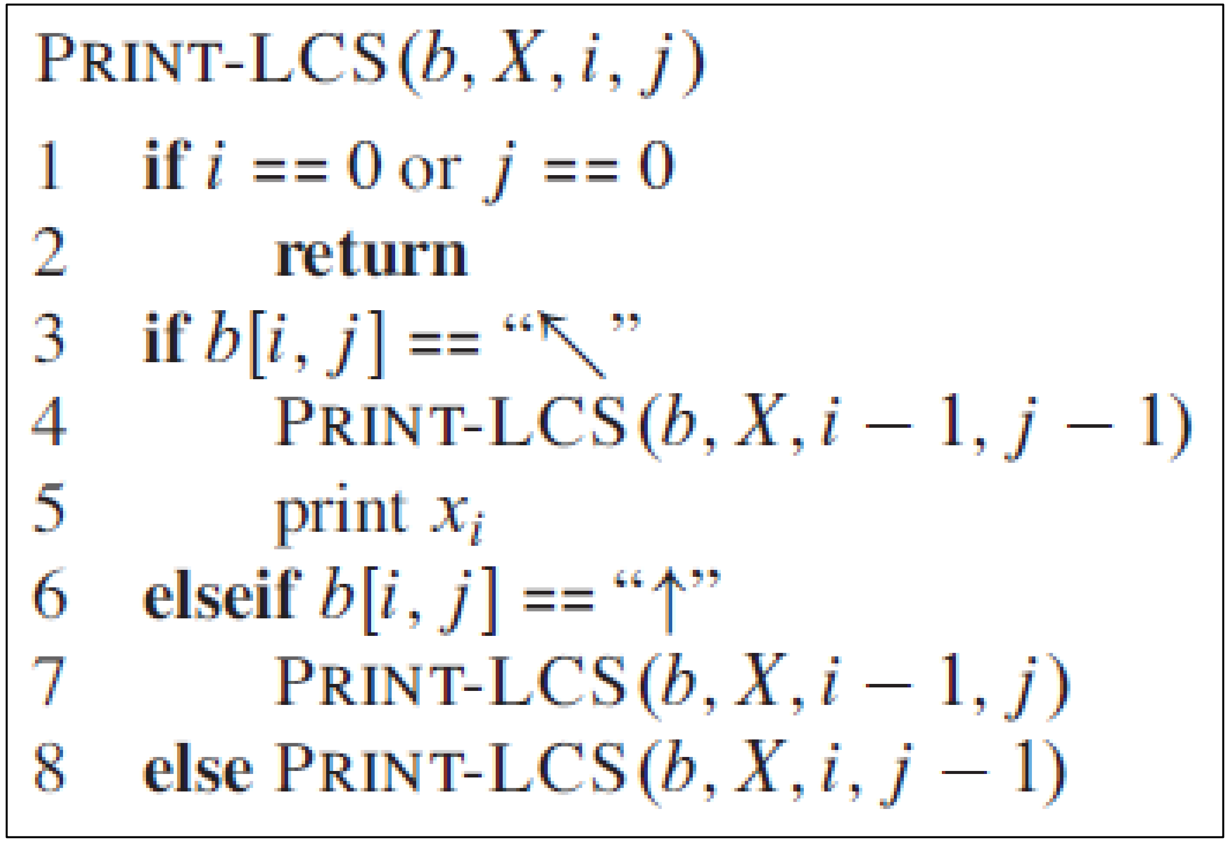
i	j	0	1	2	3	4	5	6
	y_j		B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

```
LCS-LENGTH($X, Y$)
$m = X.length$
$n = Y.length$
let $b[1..m, 1..n]$ and $c[0..m, 0..n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "↖"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] =$ "↑"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] =$ "←"
return $c$ and $b$
```

Longest Common Subsequence

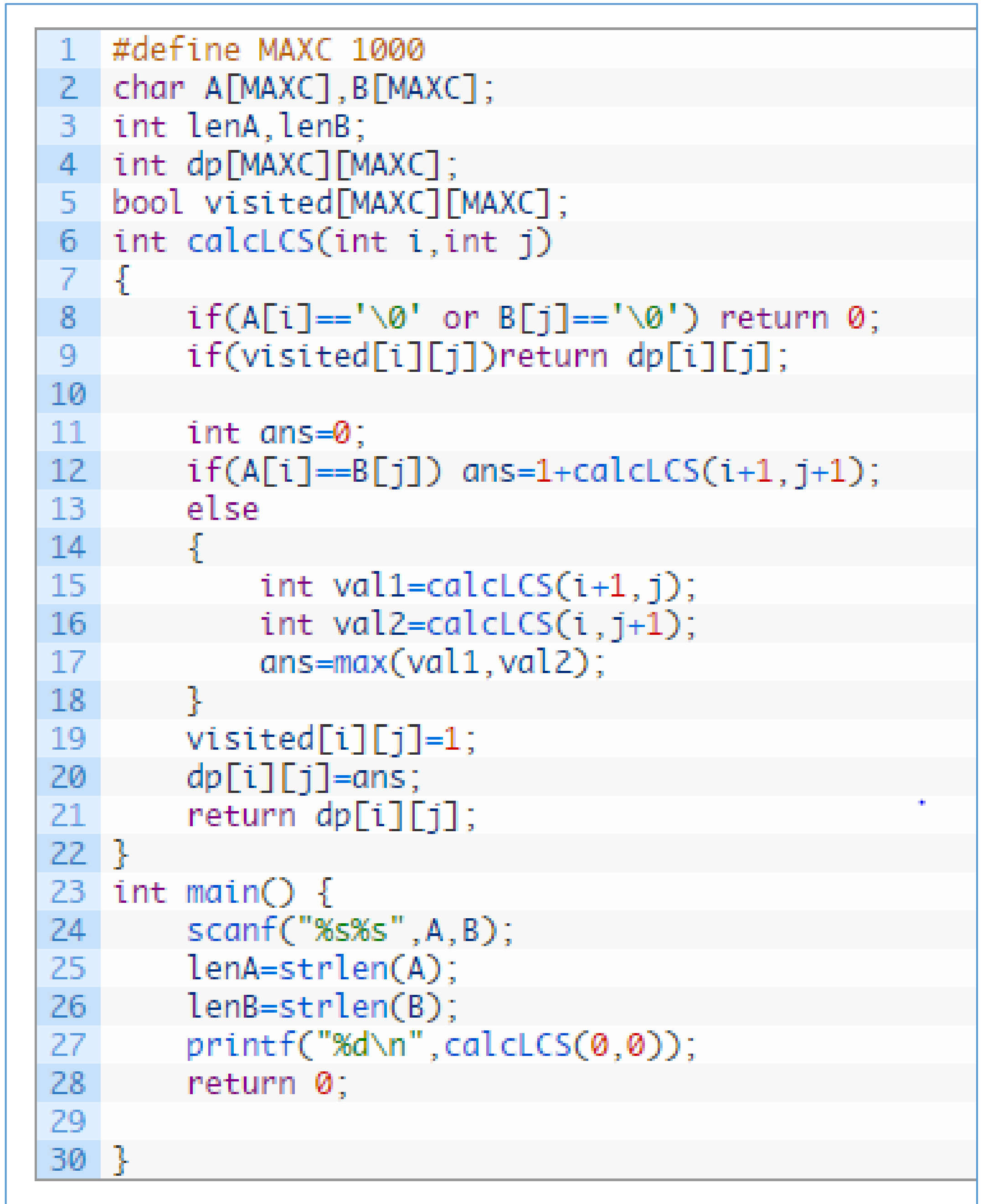


	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4



```
PRINT-LCS($b, X, i, j$)
if $i == 0$ or $j == 0$
        return
if $b[i, j] ==$ "↖"
        PRINT-LCS($b, X, i - 1, j - 1$)
        print $x_i$
elseif $b[i, j] ==$ "↑"
        PRINT-LCS($b, X, i - 1, j$)
else PRINT-LCS($b, X, i, j - 1$)
```

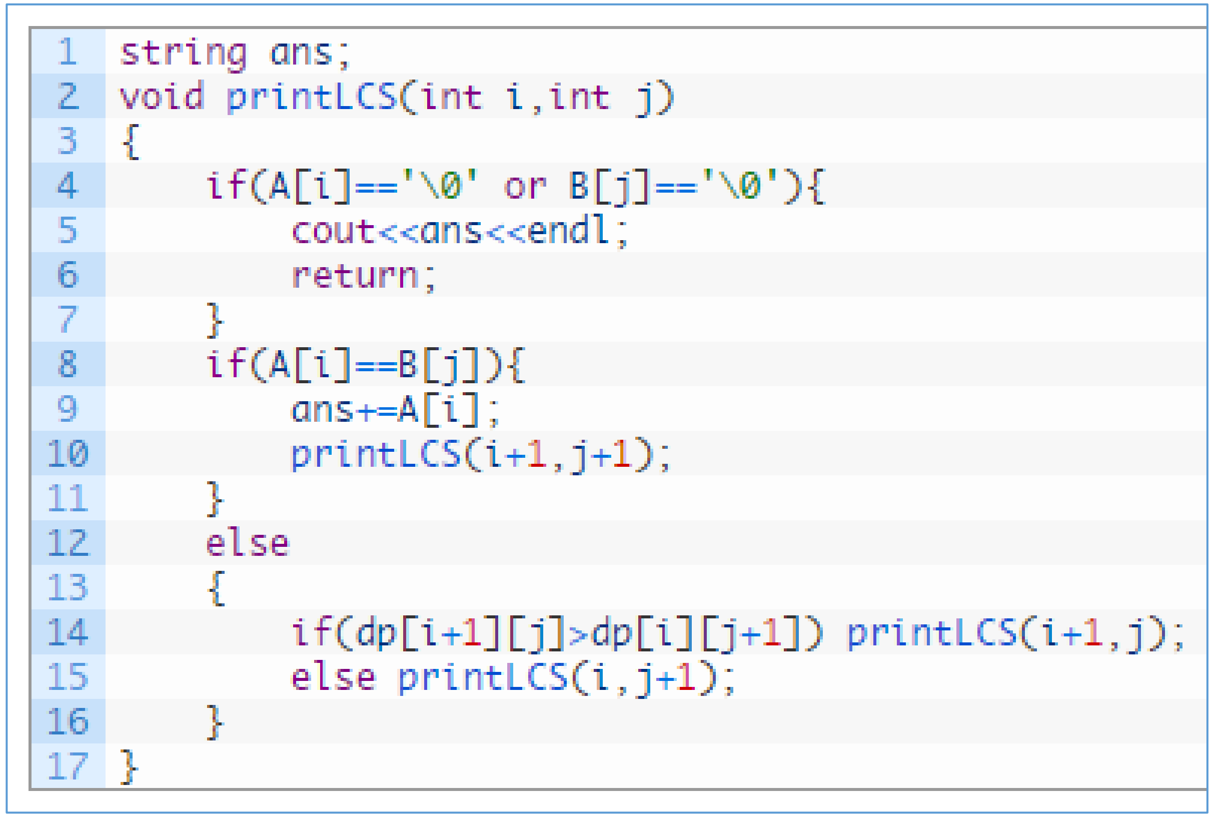
Longest Common Subsequence : Recursviely



```
#define MAXC 1000
char A[MAXC],B[MAXC];
int lenA,lenB;
int dp[MAXC][MAXC];
bool visited[MAXC][MAXC];
int calcLCS(int i,int j)
{
    if(A[i]=='\0' or B[j]=='\0') return 0;
    if(visited[i][j])return dp[i][j];

    int ans=0;
    if(A[i]==B[j]) ans=1+calcLCS(i+1,j+1);
    else
    {
        int val1=calcLCS(i+1,j);
        int val2=calcLCS(i,j+1);
        ans=max(val1,val2);
    }
    visited[i][j]=1;
    dp[i][j]=ans;
    return dp[i][j];
}
int main() {
    scanf("%s%s",A,B);
    lenA=strlen(A);
    lenB=strlen(B);
    printf("%d\n",calcLCS(0,0));
    return 0;

}
```



```
string ans;
void printLCS(int i,int j)
{
    if(A[i]=='\0' or B[j]=='\0'){
        cout<<ans<<endl;
        return;
    }
    if(A[i]==B[j]){
        ans+=A[i];
        printLCS(i+1,j+1);
    }
    else
    {
        if(dp[i+1][j]>dp[i][j+1]) printLCS(i+1,j);
        else printLCS(i,j+1);
    }
}
```

Longest Common Subsequence :

```
int main()
{   int m=0;
   int n=0;
   cout<<"The length of 1st sequence:";
   cin>>m;
   cout<<"The first sequence:";
   char s1[m];
   for(int i=0;i<m;i++)
   {
      cin>>s1[i];
   }
   cout<<"The length of 2nd sequence:";
   cin>>n;
   cout<<"The second sequence:";
   char s2[n];
   for(int j=0;j<n;j++)
   {
      cin>>s2[j];
   }
   lcsAlgo(s1,s2,m,n);
}
```

```
#include<bits/stdc++.h>
using namespace std;
void lcsAlgo(char *s1,char *s2,int m,int n)
{
   int lcsTable[m+1][n+1];
   for(int i=0;i<=m;i++)
   {for(int j=0;j<=n;j++)
      {if(i==0 || j==0)
         {
            lcsTable[i][j]=0;
         }
         else if(s1[i-1]==s2[j-1])
         {
            lcsTable[i][j]=lcsTable[i-1][j-1]+1;
         }
         else
         {
            lcsTable[i][j]=max(lcsTable[i-1][j],lcsTable[i][j-1]);
         }  } }
int lcs_length=lcsTable[m][n];
```

Longest Common Subsequence :

```
#include<bits/stdc++.h>
using namespace std;
void lcsAlgo(char *s1,char *s2,int m,int n)
{
   int lcsTable[m+1][n+1];
   for(int i=0;i<=m;i++)
   {for(int j=0;j<=n;j++)
      {if(i==0 || j==0)
         {
            lcsTable[i][j]=0;
         }
         else if(s1[i-1]==s2[j-1])
         {
            lcsTable[i][j]=lcsTable[i-1][j-1]+1;
         }
         else
         {
            lcsTable[i][j]=max(lcsTable[i-1][j],lcsTable[i][j-1]);
         }  } }
int lcs_length=lcsTable[m][n];
char lcs[lcs_length];
   lcs[lcs_length]='\0';
   int i=m,j=n;
   while(i>0 && j>0)
   {  if(s1[i-1]==s2[j-1])
      {
         lcs[lcs_length-1]=s1[i-1];
         i--;
         j--;
         lcs_length--;
      }
      else if(lcsTable[i-1][j]>lcsTable[i][j-1])
         i--;
      else
         j--; }
   lcs_length=lcsTable[m][n];
   cout<<"The first sequence:";
   cout<<s1<<endl;
   cout<<"The second sequence:";
   cout<<s2<<endl;
   cout<<"The LCS:"<<lcs<<endl;
   cout<<"LCS length:"<<lcs_length;}
```