

# DYNAMIC PROGRAMMING

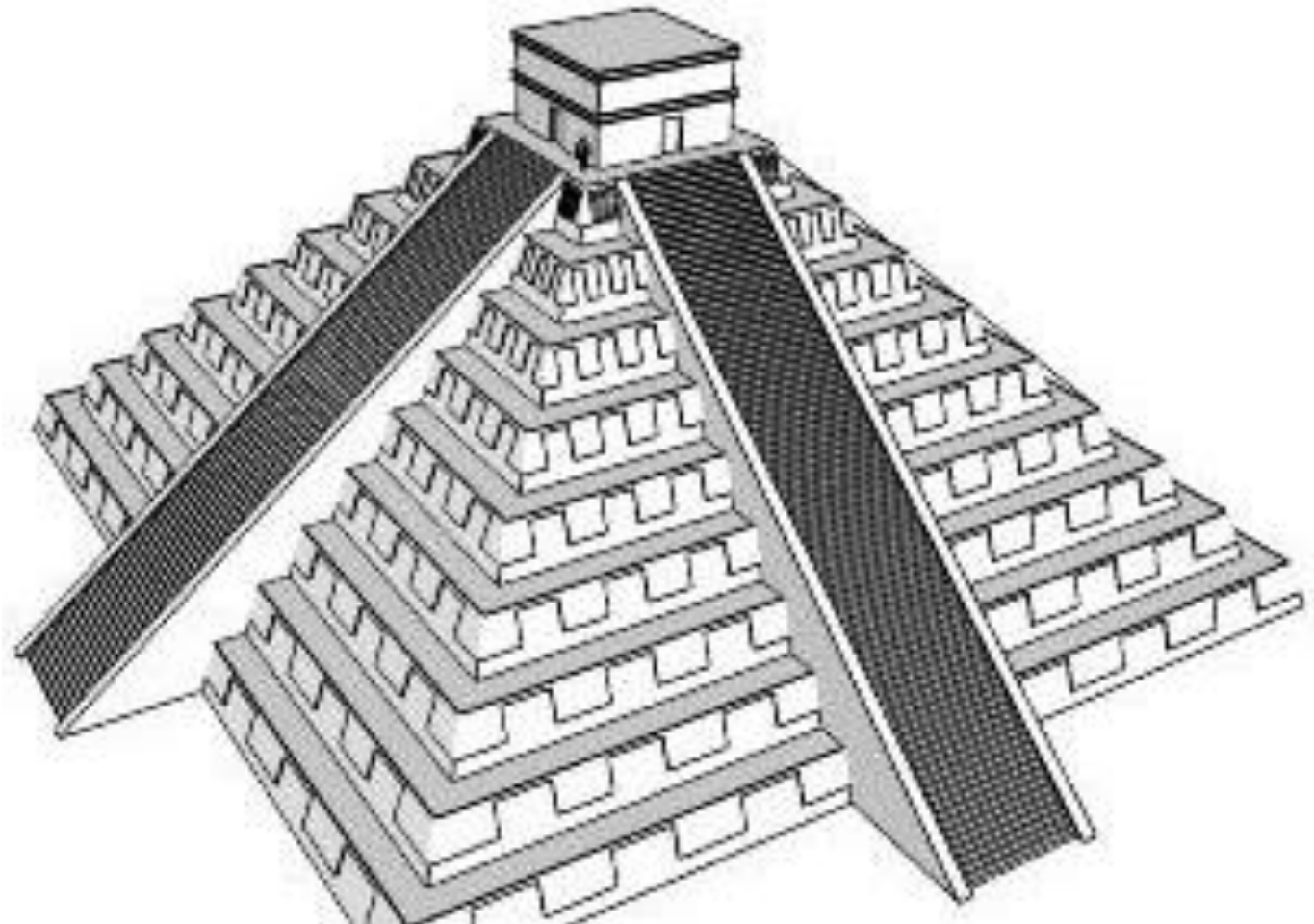
ALGORITHMS AND DATA STRUCTURES



# DYNAMIC PROGRAMMING

## DP content

- Dynamic programming
  - Design DP algorithm and strategies
- Fibonacci number calculation
- Coin-row problem
- Coin exchange problem
- The Longest Common subsequence
- Robot problem
- Knapsack problem



# WHAT IS THE DYNAMIC PROGRAMMING

## Dynamic programming

Dynamic programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a given problem's solution to solutions of its smaller subproblems

given  $1 + 1 + 1 + 1 + 1 = 5$

Calculate this  $\left\{ \begin{array}{l} 1 + 1 + 1 + 1 + 1 + \mathbf{1} = ? \\ 1 + 1 + 1 + 1 + 1 + \mathbf{1} + \mathbf{1} = ? \end{array} \right.$

### Ordinary approach

$$\begin{array}{l} 1 + 1 + 1 + 1 + 1 = 5 \\ 1 + 1 + 1 + 1 + 1 + 1 = 6 \\ 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7 \end{array}$$

### Dynamic programming

$$1 + 1 + 1 + 1 + 1 = 5$$

$$5 + 1 = 6$$

$$6 + 1 = 7$$

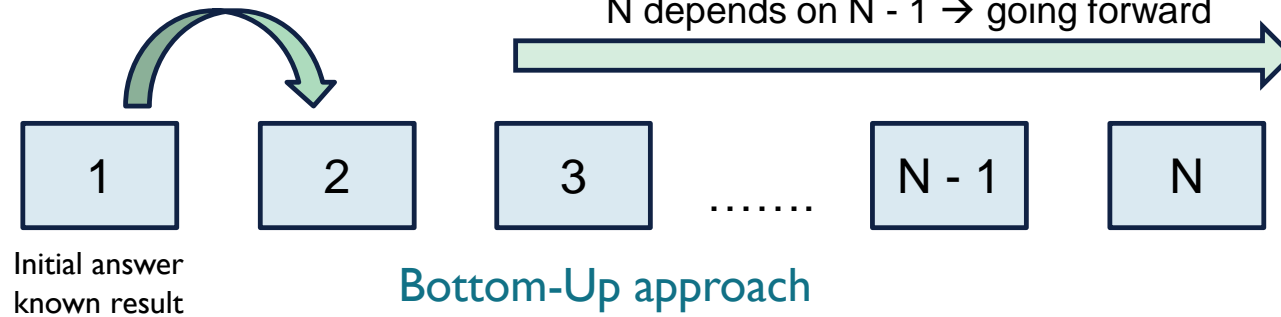
Initial state

Recurrent relation

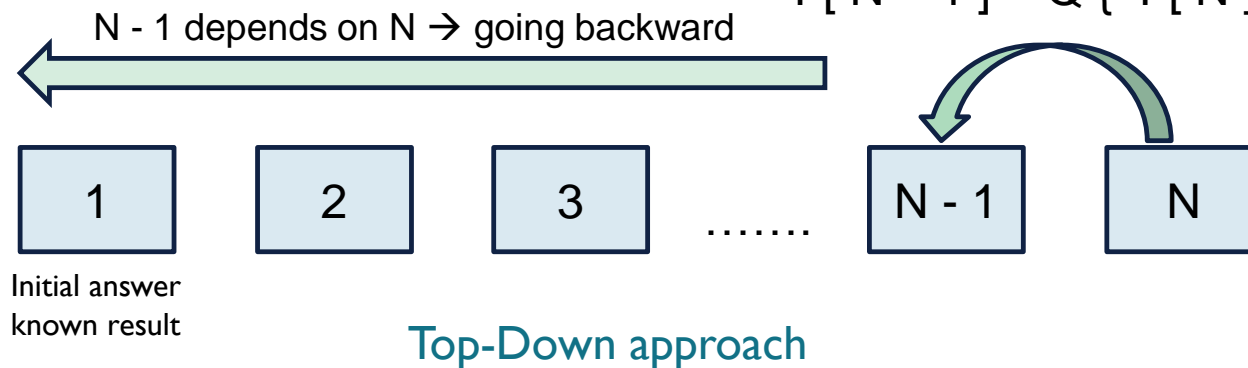
$$F[n] = 1 + F[n-1]$$

# HOW TO SOLVE DP TASKS

$$F[N] = Q\{F[N-1] \dots\}$$



$$F[N-1] = Q\{F[N] \dots\}$$



## Dynamic programming Modeling strategy

- Calculate result for the initial state
  - The initial state is the start or end point – the result must be set.
- Find **recurrent** dependency
  - This is a formula that describes dependency from the past → to the next step.
  - While implementation instead of recursive calling the function itself use the data structure like array to store n-th – state and the loop for calculating the next state.



# FIBONACCI NUMBERS COMPUTATION

DYNAMIC PROGRAMMING VS RECURSIVE APPROACH



# RECURSIVE APPROACH

## Fibonacci numbers formula

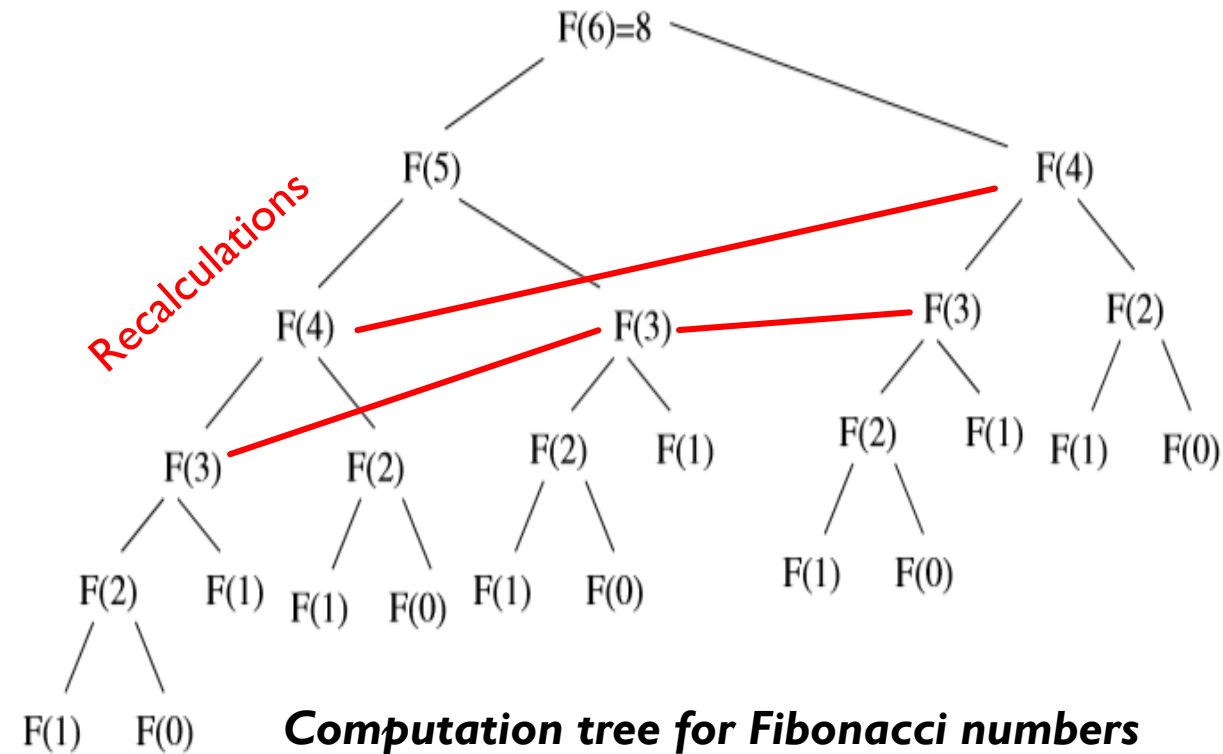
$$F_n = F_{n-1} + F_{n-2}$$

```
long fib(int n) {
    if (n == 0)
        return(0);

    if (n == 1)
        return(1);

    return(fib(n - 1) + fib(n - 2));
}
```

*Recursive implementation*



```
#define MAXN 1000000000
```

```
long fib_dp(int n) {
```

```
    int i; // counter
```

```
    long f[MAXN + 1]; // cache computed fib
```

```
    f[0] = 0;
```

```
    f[1] = 1;
```

```
    for (i = 2; i <= n; i++) {
```

```
        f[i] = f[i - 1] + f[i - 2];
```

```
    }
```

```
    return(f[n]);
```

```
}
```

*Dynamic Programming  
(Bottom → UP)*

**Caching F[ MAXN+1 ]**

**F[8]**

**F[7] F[6]**

**F[5] F[4]**

**F[3] F[2]**

**init: F[1] = 1 F[0] = 0**

# FIBONACCI SOLVING

## Algorithm

- Top-down Fibonacci DP solution will record the each Fibonacci calculation in a table so it won't have to re-compute the value again when you need it, a simple table-lookup is enough (memorization), whereas Bottom-up DP solution will build the solution from smaller numbers

```
int DP_Top_Down(int n) {  
  
    if (n == 1 || n == 2)  
        return 1;  
  
    if (memo[n] != 0)  
        return memo[n];  
  
    memo[n] = DP_Top_Down(n-1) + DP_Top_Down(n-2);  
  
    return memo[n];  
}
```

Dynamic programming Top→Down



# COIN ROW PROBLEM

DYNAMIC PROGRAMMING



## COIN-ROW PROBLEM

*Coin-row problem* There is a row of  $n$  coins whose values are some positive integers  $c_1, c_2, \dots, c_n$ , not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.



# ANALYZING THE COIN-ROW PROBLEM

Let  $F(n)$  be the maximum amount that can be picked up from the row of  $n$  coins. To derive a recurrence for  $F(n)$  :

## Initial values

$$F(0) = 0$$

$$F(1) = c_1.$$

## Recurrent formula

Partition all allowed coins into two groups

Includes the last coin

$$c_n + F(n - 2)$$

Not includes the last coin

$$F(n - 1)$$

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

# COIN-RAW ALGORITHM

//Applies formula bottom up to find  
 // the maximum amount of money  
 //that can be picked up from a coin  
 // row without picking two adjacent coins  
 //Input: Array  $C [1..n]$  of positive integers  
 // indicating the coin values  
 //Output: The maximum amount of money  
 //that can be picked up

$F[0] \leftarrow 0;$

$F[1] \leftarrow C[1]$

for  $i \leftarrow 2$  to  $n$  do

$F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$

return  $F[n]$

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

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

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

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

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

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

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

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

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

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

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

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



# COIN CHANGE PROBLEM

DYNAMIC PROGRAMMING





## THE TASK

*Change-making problem* Consider the general instance of the following well-known problem. Give change for amount  $n$  using the minimum number of coins of denominations  $d_1 < d_2 < \dots < d_m$ .

## ANALYZING A PROBLEM

Assuming availability of unlimited quantities of coins for each of the  $m$  denominations  $d_1 < d_2 < \dots < d_m$  where  $d_1 = 1$ .

$F(n)$  be the minimum number of coins whose values add up to  $n$

The amount  $n$  can only be obtained by adding one coin of denomination  $d_j$  to the amount  $n - d_j$  for  $j = 1, 2, \dots, m$  such that  $n \geq d_j$

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

$$F(0) = 0$$

Therefore, we can consider all such denominations and select the one minimizing  $F(n - d_j) + 1$ . Since 1 is a constant, we can, of course, find the smallest  $F(n - d_j)$

# ALGORITHM

**ALGORITHM**  $F[0] \leftarrow 0$  *ChangeMaking*( $D[1..m], n$ )  
//Applies dynamic programming to find the minimum number of coins  
//of denominations  $d_1 < d_2 < \dots < d_m$  where  $d_1 = 1$  that add up to a  
//given amount  $n$   
//Input: Positive integer  $n$  and array  $D[1..m]$  of increasing positive  
// integers indicating the coin denominations where  $D[1]=1$   
//Output: The minimum number of coins that add up to  $n$

```
for  $i \leftarrow 1$  to  $n$  do
     $temp \leftarrow \infty$ ;  $j \leftarrow 1$ 
    while  $j \leq m$  and  $i \geq D[j]$  do
         $temp \leftarrow \min(F[i - D[j]], temp)$ 
         $j \leftarrow j + 1$ 
     $F[i] \leftarrow temp + 1$ 
return  $F[n]$ 
```

$F[0] = 0$

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

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

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

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

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

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

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

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

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

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

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

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

$n$	0	1	2	3	4	5	6
$F$	0	1	2	1	1	2	<b>2</b>

# COIN CHANGE PROBLEM

```
#include <iostream>
using namespace std;
```

```
int CoinChangeDynamic(int amount, int d[], int size, int C[])
{
    C[0] = 0;
    for (int j = 1; j <= amount; j++) {
        C[j] = INT_MAX;
        for (int i = 0; i < size; i++) {
            if (j >= d[i] && 1 + C[j - d[i]] < C[j]) {
                C[j] = 1 + C[j - d[i]];
            }
        }
    }
    return C[amount];
}
```

```
int main()
{
    int d[] = { 1, 5, 10, 25 };
    int amount = 67;
    int size = sizeof(d) / sizeof(d[0]);
    int *C = new int[amount + 1];
    int ans = CoinChangeDynamic(amount, d, size, C);
    cout << "Minimal # of coins = " << ans << endl;

    delete[] C;

    return 0;
}
```

# COIN COLLECTING PROBLEM (ROBOT WALK PROBLEM)

DYNAMIC PROGRAMMING





# COIN COLLECTING PROBLEM

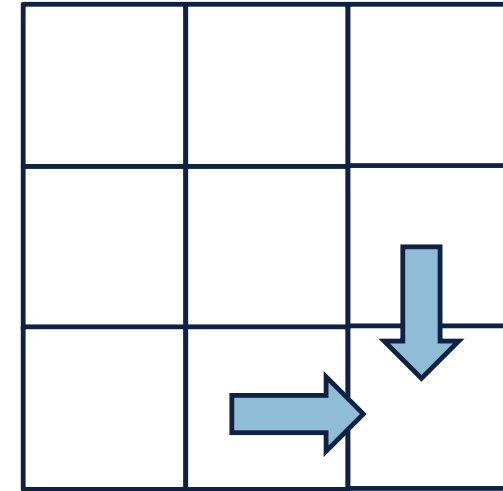
Several coins are placed in cells of an  $n \times m$  board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right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



# ANALYZING AN ALGORITHM

$F(i, j)$  largest number of coins the robot can collect and bring to the cell  $(i, j)$

- Reach to cell  $(i, j)$  from
  - $(i - 1, j)$
  - $(i, j - 1)$



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

$$F(0, j) = 0 \text{ for } 1 \leq j \leq m \text{ and } F(i, 0) = 0 \text{ for } 1 \leq i \leq n,$$

# ALGORITHM

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

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	<b>5</b>

$F[1, 1] \leftarrow C[1, 1]$ ; **for**  $j \leftarrow 2$  **to**  $m$  **do**  $F[1, j] \leftarrow F[1, j - 1] + C[1, j]$

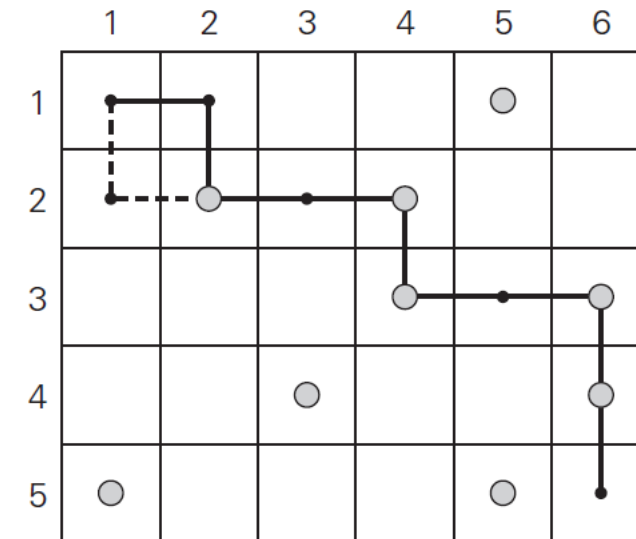
**for**  $i \leftarrow 2$  **to**  $n$  **do**

$F[i, 1] \leftarrow F[i - 1, 1] + C[i, 1]$

**for**  $j \leftarrow 2$  **to**  $m$  **do**

$F[i, j] \leftarrow \max(F[i - 1, j], F[i, j - 1]) + C[i, j]$

**return**  $F[n, m]$



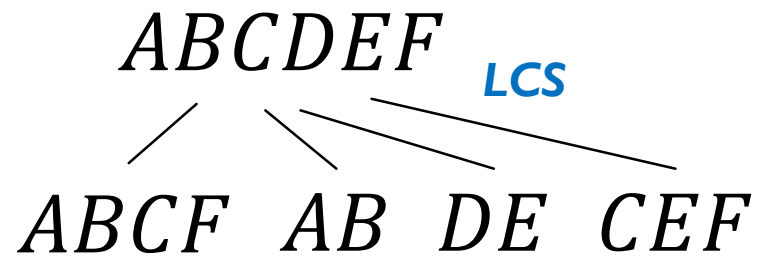


# THE LONGEST COMMON SUBSEQUENCE (LCS)

DYNAMIC PROGRAMMING

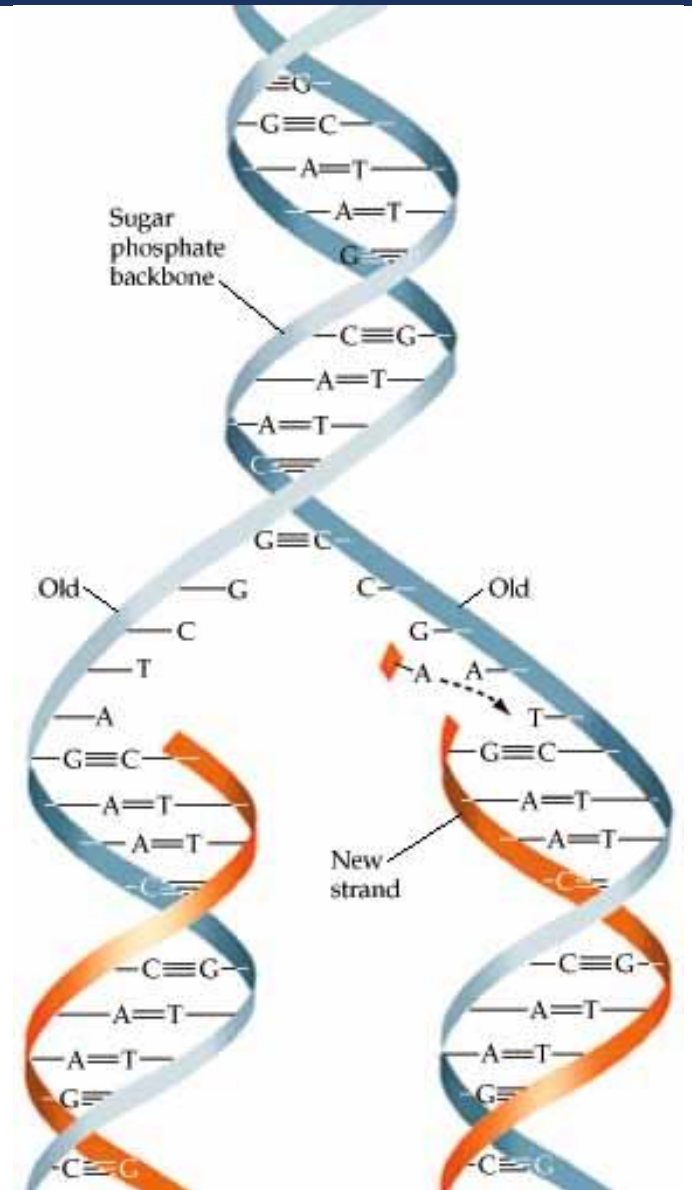


# THE LONGEST COMMON SUBSEQUENCE



LCS (“**ABCDGH**” and “**AEDFHR**”) “**ADH**” : length 3.

LCS (“**AGGTAB**” and “**GXTXAYB**”) “**GTAB**”: length 4.





# LCS

## The Longest Common Subsequence (LCS)

char  $X[0..m-1]$ ,  $Y[0..n-1]$     The length of strings  $X$  and  $Y$

$LCS(X[0..m-1], Y[0..n-1])$     Function that returns LCS

$$LCS(X[0..m-1], Y[0..n-1]) = \begin{cases} X[m-1] == Y[n-1], & 1 + LCS(X[0..m-2], Y[0..n-2]) \\ X[m-1] != Y[n-1], & MAX ( LCS(X[0..m-2], Y[0..n-1]), \\ & LCS(X[0..m-1], Y[0..n-2]) \\ & ) \end{cases}$$

If The last char is matches:  $X[m-1] == Y[n-1]$   
     then:

$$LCS(X[0..m-1], Y[0..n-1]) = 1 + LCS(X[0..m-2], Y[0..n-2])$$

If the last chat do not matches:  $X[m-1] != Y[n-1]$   
     then:

$$LCS(X[0..m-1], Y[0..n-1]) = MAX ( LCS([0..m-2], Y[0..m-1]), LCS(0..m-1], Y[0..m-2]))$$

## LCS

$$X[m-1] == Y[n-1]$$

“AGGTAB” and “GXTXAYB”.

$$\text{LCS}(\text{“AGGTAB”}, \text{“GXTXAYB”}) = 1 + \text{LCS}(\text{“AGGTA”}, \text{“GXTXAY”})$$

$$X[m-1] != Y[n-1]$$

“ABCDGH” and “AEDFHR”.

$$\text{LCS}(\text{“ABCDGH”}, \text{“AEDFHR”}) = \text{MAX} \left( \begin{array}{l} \text{LCS}(\text{“ABCDG”}, \text{“AEDFHR”}), \\ \text{LCS}(\text{“ABCDGH”}, \text{“AEDFH”}) \end{array} \right)$$

# LCS

```
#include<iostream>
using namespace std;

// length of LCS for X[0..m-1], Y[0..n-1]
int lcs(char *X, char *Y, int m, int n)
{
    if (m == 0 || n == 0)
        return 0;
    if (X[m - 1] == Y[n - 1])
        return 1 + lcs(X, Y, m - 1, n - 1);
    else
        return max(lcs(X, Y, m, n - 1), lcs(X, Y, m - 1, n));
}

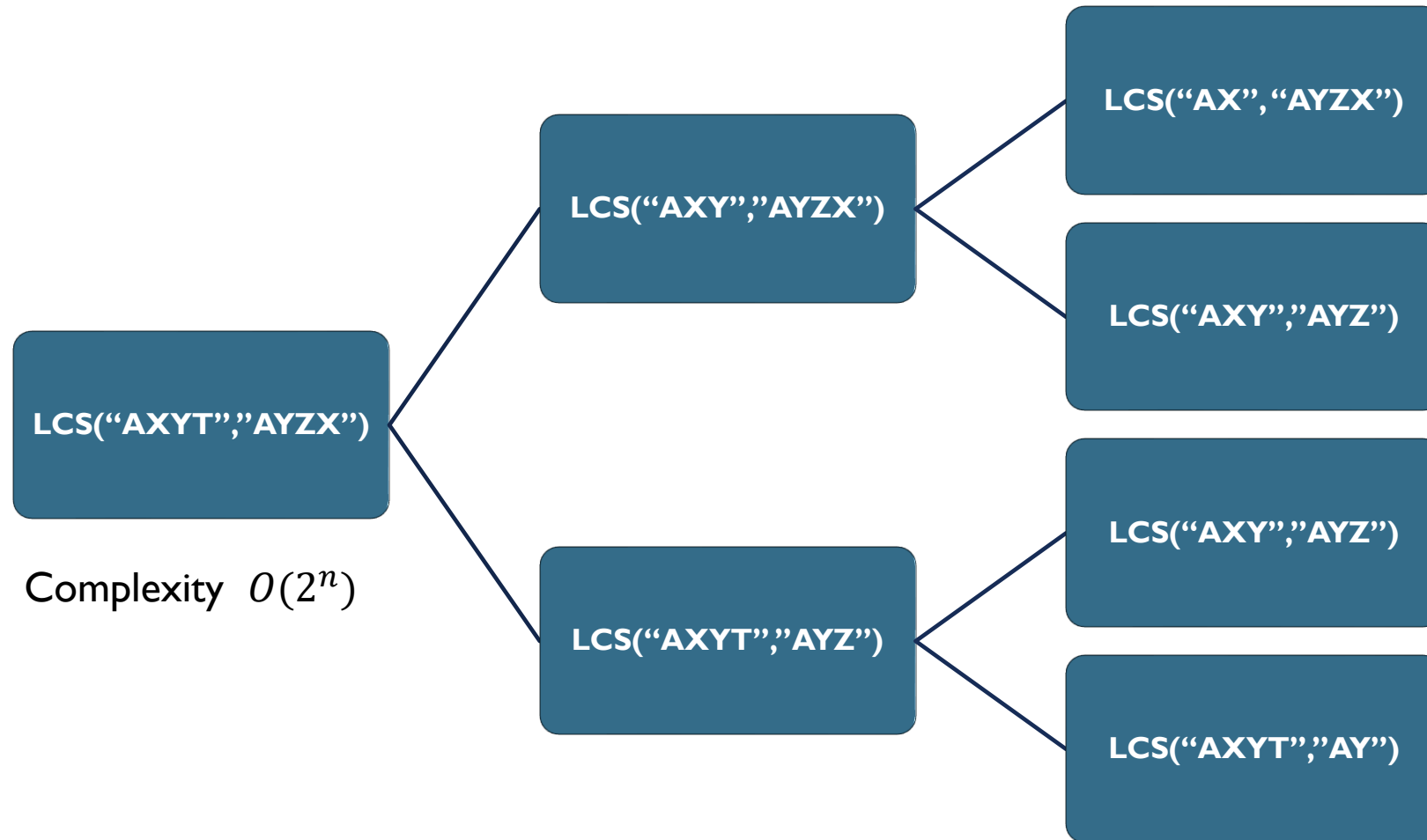
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```
int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    cout<<"Length of LCS is : "<< lcs(X, Y, m,
n)<<endl;
    system("pause");
    return 0;
}
```

# LCS: CALCULATING



# LCS

```
#include<iostream>
/* Returns length of LCS for X[0..m-1], Y[0..n-1] */
int lcs(char *X, char *Y, int m, int n)
{
    int **L = new int* [m+1];
    for (int i = 0; i < m+1 ;i++) {
        L[i] = new int[n+1];
    }
    //int L [m + 1][n + 1];
    int i, j;
    // L[m+1][n+1] in bottom up fashion.
    // L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]
    for (i = 0; i <= m; i++){
        for (j = 0; j <= n; j++){
            if (i == 0 || j == 0)
                L[i][j] = 0;

            else if (X[i - 1] == Y[j - 1])
                L[i][j] = L[i - 1][j - 1] + 1;

            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }
    /* L[m][n] contains length of LCS for X[0..n-1] and Y[0..m-1] */
    int result = L[m][n];
    for (int z = 0; z <= m; z++) {
        delete[] L[z];
    }
    delete[] L;
    return result;
}
```

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int main()
{
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    printf("Length of LCS is %d\n", lcs(X, Y, m,
n));

    system("pause");
    return 0;
}
```



# KNAPSACK PROBLEM

DYNAMIC PROGRAMMING



# KNAPSACK PROBLEM

## Task

- Given weights & values of  $n$  items. put these items in a knapsack of capacity  $W$  to get the max of total value in the knapsack

value[0.. $n-1$ ], weight[0.. $n-1$ ] such that sum of the weight of this subset is smaller than or equal to  $W$ .



# KNAPSACK PROBLEM

Value[ ] = {60, 100, 120};

Weight[ ] = {10, 20, 30};



**$W = 50$**

Weight = 10; Value = 60;

Weight = 20; Value = 100;

Weight = 30; Value = 120;

Weight = (20+10); Value = (100+60)

Weight = (30+10); Value = (120+60);

Weight = (30+20); Value = (120+100);

Weight = (30+20+10) > 50

**Result: 220**

# KANPSACK PROBLEM

```
#include<iostream>
using namespace std;

int max(int a, int b) { return (a > b) ? a : b; }

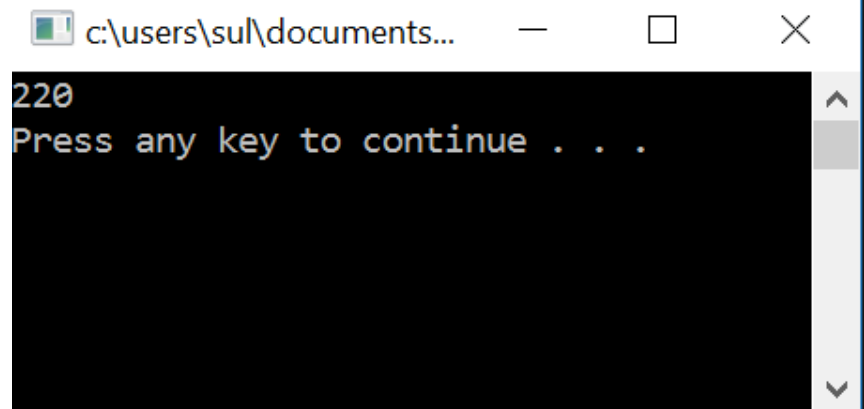
int knapSack(int W, int wt[], int val[], int n){
    if (n == 0 || W == 0)
        return 0;
    // If weight of the nth item is higher than capacity W
    if (wt[n - 1] > W)
        return knapSack(W, wt, val, n - 1);

    //maximum of two cases:
    // (1) nth item included
    // (2) not included
    else
        return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
                    knapSack(W, wt, val, n - 1));
}
```

```
int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) / sizeof(val[0]);

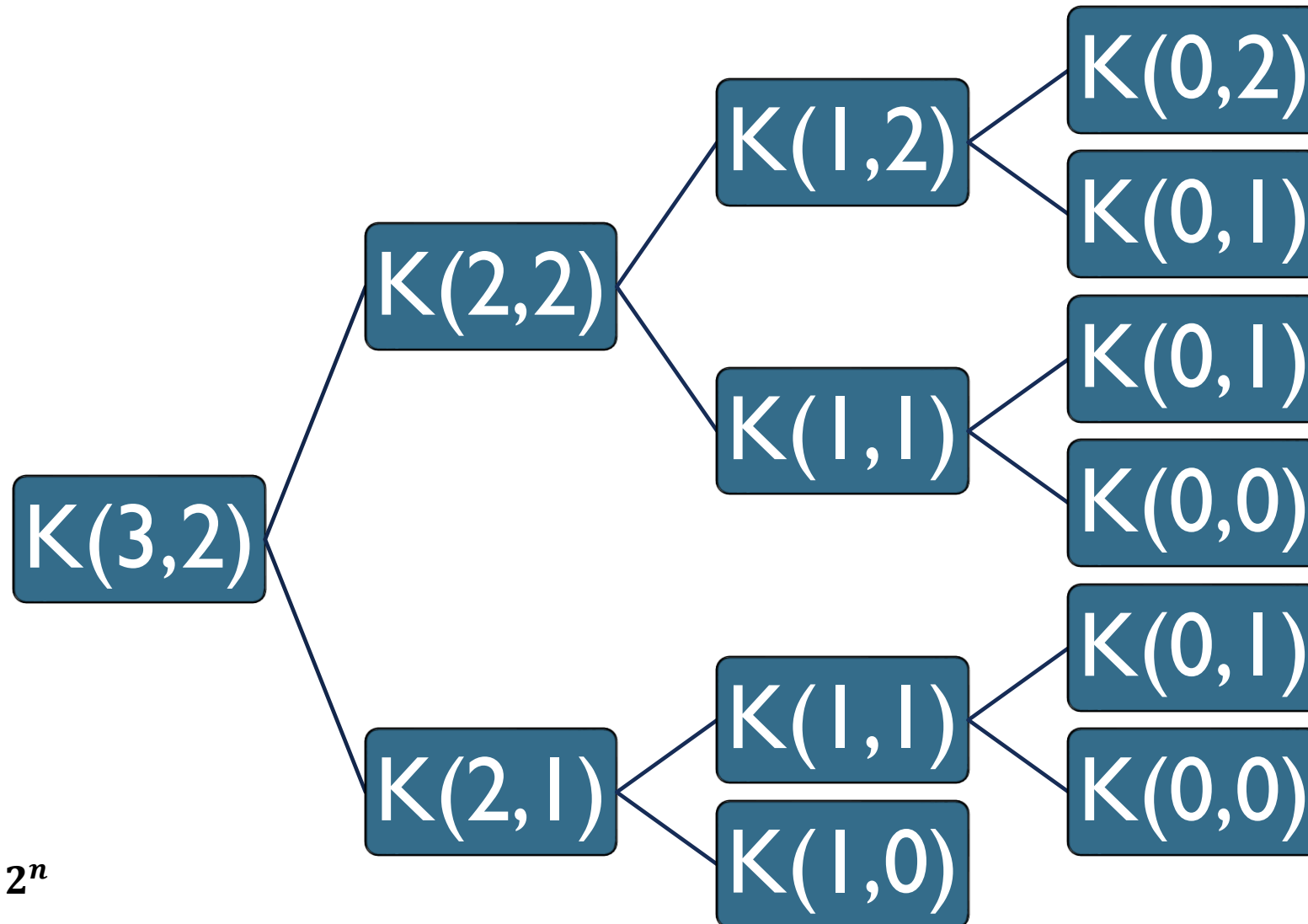
    cout<< knapSack(W, wt, val, n)<< endl;

    system("pause");
    return 0;
}
```



The screenshot shows a Windows command prompt window with the title bar 'c:\users\sul\documents...'. The window contains the output '220' followed by the prompt 'Press any key to continue . . .'. The window has standard Windows controls (minimize, maximize, close) in the top right corner.

# KNAPSACK PROBLEM



**Recursive :  $2^n$**



# KNAPSACK PROBLEM

```
#include<iostream>
using namespace std;

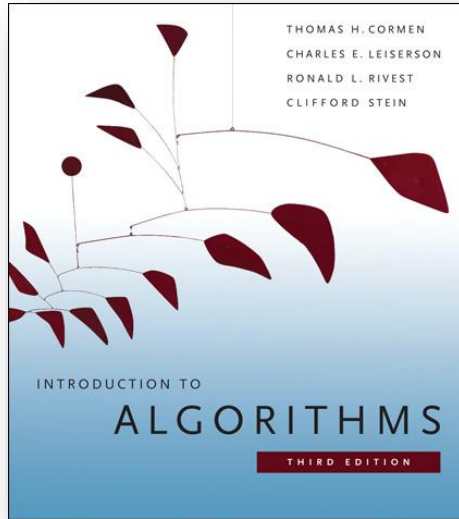
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
    int **K = new int*[n + 1];
    for (int i = 0; i < W + 1; i++) {
        K[i] = new int[W + 1];
    } //dynamic allocation int K[n + 1][W + 1];
    // Build table K[][] from bottom to up
    for (i = 0; i <= n; i++)
    {
        for (w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] + K[i - 1][w - wt[i - 1]],
                               K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
}
```

```
int result = K[n][W];
for (int i = 0; i < n + 1; i++) {
    delete[] K[i];
}delete [] K;
return result;
}

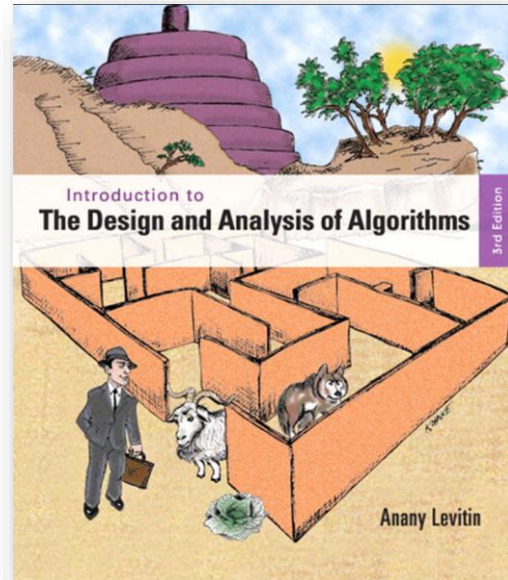
int main()
{
    int val[] = { 60, 100, 120 };
    int wt[] = { 10, 20, 30 };
    int W = 50;
    int n = sizeof(val) /
sizeof(val[0]);
    cout << knapSack(W, wt, val, n)
        << endl;
    int x;
    cin >> x;
    system("pause");
    return 0;
}
```

DP :  $O(n * W)$

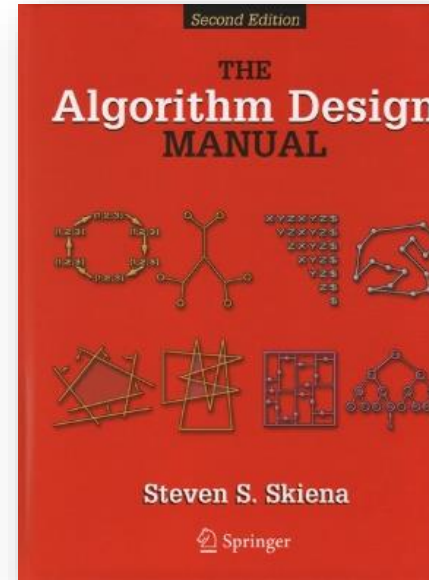
# LITERATURE



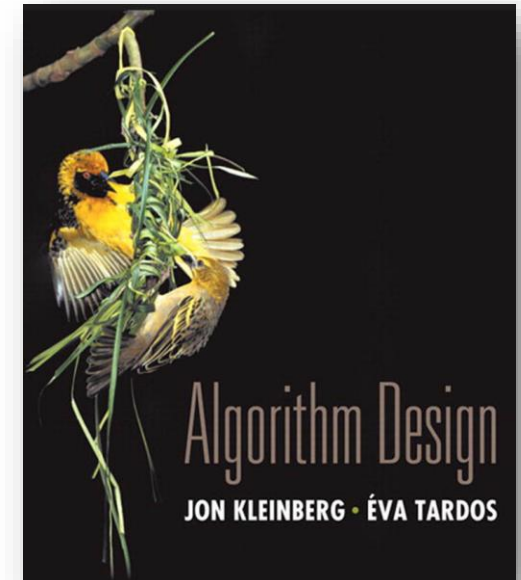
Thomas H. Cormen  
Introduction to Algorithms  
Chapter 15 Dynamic programming  
Page 359.



Anany Levitin  
Introduction to The Design and Analysis of  
Algorithms  
Chapter 8: Dynamic programming  
Page 283.



Steven S. Skiena  
The Algorithms Design Manual  
Chapter 8 Dynamic programming  
Page 273



Jon Kleinberg, Eva Tardos  
Algorithm Design  
Chapter 6 Dynamic programming  
Page 251