



IV. Dynamic Programming

Those who cannot remember the past
are condemned to repeat it.

-Dynamic Programming

Fundamental TIPS

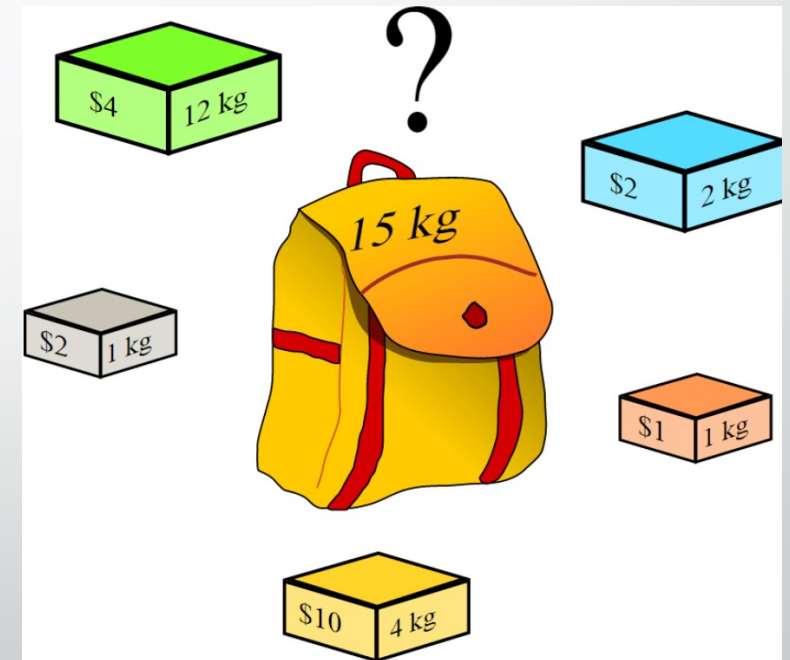
- 📌 The key skills that you have to develop in order to master DP are the abilities to determine the **problem states** and to determine the relationships or **transitions** between current problems and their sub-problems.
- 📌 prepare to see/use lots of recursion and **recurrence relations**!
- 📌 DP problems with small input size constraints may already be solvable with **recursive backtracking**
- 📌 If you are new to the DP technique, you can start by assuming that (the 'top-down') DP is a kind of 'intelligent' or 'faster' recursive backtracking.
- 📌 When is it likely to use DP?
 - DP is typically applied to **Optimization Problems** (e.g., maximizing && minimizing) and *counting* problems.
 - E.g., if you encounter a problem that stats "minimize this" / "maximize that" / "count the ways to do sthg"

What is Dynamic Programming?

- DP, like the *divide-and-conquer* method, it solves problems by combining the solutions of subproblems!
 - 📌 **"Optimal Substructure"**!
 - "Programming", in the DP context, refers to "Dynamically Tracking/Planning transitions among states/sub-problems" – R. Bellman
 - Definition:
 - "A given problem has the Optimal Substructure Property if an optimal solution of the given problem can be obtained by using optimal solutions of its subproblems."
 - Examples?
- What's the difference, then, between DP and Div&Conq?
 - Solving the problems recursively?
 - Of course, NO
 - 📌 DP is applicable when the subproblems are not independent
 - When subproblems share sub-subproblems!
 - 📌 This is called "**Overlapping Subproblems**"!
 - DP solves every subproblem just once and saves the answer in a table, thus avoiding useless re-computations 📌 "**Memoization**"

The Optimal Substructure in the Knapsack Problem

- Maximizing the Total Value of Items to take constrained by the Knapsack "Size-Limit"
- The Knapsack problem has various real-world applications across different fields. Here are some examples:
 - **Resource Allocation:**
 - For example, in *project management*, it can be used to allocate limited resources (such as budget, manpower, or equipment) to maximize project outcomes.
 - **Resource Scheduling:**
 - For example, in *airline or freight logistics*, it can be employed to optimize the loading of cargo onto limited-capacity aircraft or trucks, considering weight, space, and other constraints.
 - **Portfolio Optimization:**
 - In *finance*, it can be used to optimize portfolio selection.
 - Given a set of investments with different expected returns and risks, and a limited investment budget, the problem is to select a combination of investments that maximizes the expected return while staying within the budget.



Knapsack Real-world Applications cont.

- **Cutting Stock Problem:**

- In **manufacturing** and material optimization, the Knapsack problem can be used to solve cutting stock problems.
- Given a set of orders for different-sized items and a limited amount of stock material (such as rolls of fabric or metal sheets), the problem is to determine the most efficient cutting patterns to minimize waste and fulfill all orders.

- **Bin Packing:**

- The Knapsack problem is closely related to the bin packing problem. In **logistics and transportation**, it can be applied to optimize the packing of items into containers, trucks, or shipping containers, aiming to minimize the number of containers used or maximize space utilization.

- **Advertising Campaign Optimization:**

- In marketing and advertising, the Knapsack problem can be used to optimize the allocation of **advertising budget** across various channels or media platforms, considering the expected returns (such as conversion rates or customer reach) associated with each channel and budget constraints.

- **DNA Sequencing:**

- The Knapsack problem has been used in **bioinformatics** for **DNA sequencing**.
- The problem involves selecting a subset of DNA fragments for sequencing, considering their lengths and costs, while aiming to obtain the maximum coverage or the most informative sequences.



- **etc.**

Last Word about Knapsack Problem

- These are just a few examples that highlight the diverse range of real-world applications of the Knapsack problem.
- The **problem's versatility** and ability to **model various optimization scenarios** make it a valuable tool in **decision-making** and **resource allocation in many industries**.

Knapsack Problem

Optimal Sub-structure – Cont.

- For the Knapsack problem to exhibit an optimal substructure, this stipulates that the optimal solution (with a problem with size n) can be constructed from the optimal solutions to its smaller subproblems.
- **How?**
 - To understand how the Knapsack problem exhibits optimal substructure, let's consider a scenario where we have a knapsack with a capacity of W and a set of n items with values (v_1, v_2, \dots, v_n) and weights (w_1, w_2, \dots, w_n) .
- Now, let's assume we are at the last item, item n . We have two choices:
 -   either include item n in the knapsack or exclude it.
- If we include item n in the knapsack, we need to consider the remaining $(n-1)$ items to fill the remaining capacity $(W - w_n)$.
 - In this case, the total value of the knapsack would be v_n plus the optimal value of the $(n-1)$ items with a capacity of $(W - w_n)$. This forms a **subproblem** with $(n-1)$ items and a reduced capacity..

Knapsack Problem

Optimal Sub-structure – Cont.

- If we exclude item n from the knapsack, we only need to consider the remaining $(n-1)$ items with the same capacity of W .
 - In this case, the total value of the knapsack would be the optimal value of the $(n-1)$ items with the same capacity.
- To find the optimal solution to the Knapsack problem for n items and a capacity of W , we compare the total value obtained by including item n ($v_n +$ optimal value of $(n-1)$ items with capacity $(W - w_n)$) and the total value obtained by excluding item n (optimal value of $(n-1)$ items with capacity W). We choose the option that yields the maximum value.
- This *recursive decision-making process* demonstrates the optimal substructure property:
 - The optimal solution to the original problem (n items and capacity W) can be constructed by making optimal choices at each step, considering the optimal solutions to the smaller subproblems ($(n-1)$ items and reduced capacity).
- By using dynamic programming techniques, such as **memoization** or **bottom-up tabulation**, we can efficiently solve the Knapsack problem by storing the optimal solutions to the subproblems and reusing them when needed, avoiding redundant computations.
- 🗨️ You will get a chance to implement this! – Just understand the “CORE” of DP!, 🗨️ “Optimality Substructure”
- 🗨️ I have “Demonstrated Code” for You 😊

Knapsack – Code / Memoization

```
// R. Abid,
#include <stdio.h>

#define MAX_ITEMS 100
#define MAX_CAPACITY 1000

int values[MAX_ITEMS];
int weights[MAX_ITEMS];
int memo[MAX_ITEMS][MAX_CAPACITY];

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

int knapsack(int item, int capacity) {
    if (item < 0) // no more items,
        return 0;

    if (memo[item][capacity] != -1) // already computed,
        return memo[item][capacity];

    if (weights[item] > capacity) // cannot be included,
        memo[item][capacity] = knapsack(item - 1, capacity);
    else
        memo[item][capacity] = max(knapsack(item - 1, capacity),
                                    values[item] + knapsack(item - 1, capacity - weights[item]));

    return memo[item][capacity];
}
```

Knapsack – Code / Memoization – Cont.

```
}

int main() {
    int numItems, capacity;

    printf("Enter the number of items: ");
    scanf("%d", &numItems);

    printf("Enter the values and weights of each item:\n");
    for (int i = 0; i < numItems; i++)
        scanf("%d %d", &values[i], &weights[i]);

    printf("Enter the capacity of the knapsack: ");
    scanf("%d", &capacity);

    // Initialize memoization table with -1
    for (int i = 0; i < MAX_ITEMS; i++) {
        for (int j = 0; j < MAX_CAPACITY; j++) {
            memo[i][j] = -1;
        }
    }

    int maxVal = knapsack(numItems - 1, capacity);

    printf("Maximum value that can be obtained: %d\n", maxVal);

    return 0;
}
```

ACI

Knapsack – Code / Tabulation

```
// R. Abid,
#include<stdio.h>

// Function to find the maximum of two integers
int max(int a, int b) {
    return (a > b) ? a : b;
}

// Function to solve the Knapsack problem using dynamic programming
int knapsack(int W, int wt[], int val[], int n) {
    int i, w;
    int K[n+1][W+1];

    // Build the bottom-up table K[][] in a tabular manner
    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];
        }
    }
    return K[n][W];
}

// Main function
int main() {
    int val[] = {1, 2, 10, 60, 5, 12};
    int wt[] = {2, 30, 5, 10, 20, 30};
    int W = 50;
    int n = sizeof(val)/sizeof(val[0]);

    printf("Maximum value that can be obtained: %d\n", knapsack(W, wt, val, n));
    return 0;
}
```

Re-Capitulation

DP - The basic Idea

1. Find a naïve exponential-time recursive algorithm
 2. Speed up the algorithm by storing solutions to subproblems
 3. Speed it up further by solving subproblems in a more efficient order
- -----
 - 🚩👉 DP is not an individual algorithm!, it is a framework/**paradigm** for developing algorithms!! ... surprisingly efficient ones!

Brief History of DP

- Dynamic programming was invented by Richard E. Bellman at the RAND Corporation circa 1950.
- The word “**programming**” is not used in its modern sense but in the sense of finding an **optimal schedule** or program of activities.
- I can/will send you relevant papers
- **Bellman explains the name:**
 - *The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word, research... His face would suffuse, he would turn red, and he would get violent if people used the term, research, in his presence. You can imagine how he felt, then, about the term, mathematical... I thought dynamic programming was a good name. It was something not even a Congressman could object to.*

Practice 1

Binomial Coefficients

UVA#1649

- A binomial coefficient $C(n, k)$ can be defined as the coefficient x^k in the expression $(1 + x)^n$
- In mathematics, the binomial coefficients are the positive integers that occur as coefficients in the binomial theorem
- For example, the fourth power of $1 + x$ is:
- Write a function that takes 2 parameters n and k and returns the value of the binomial coefficient $C(n, k)$
- Make sure you work out:
 - *Optimal Substructure*
 - *Overlapping sub-problems*

$$\begin{aligned}(1 + x)^4 &= \binom{4}{0}x^0 + \binom{4}{1}x^1 + \binom{4}{2}x^2 + \binom{4}{3}x^3 + \binom{4}{4}x^4 \\ &= 1 + 4x + 6x^2 + 4x^3 + x^4,\end{aligned}$$

Practice 1

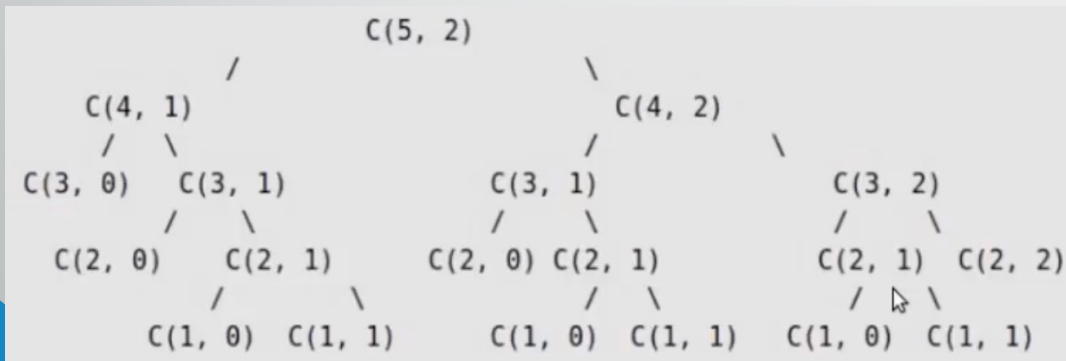
Binomial Coefficients - Discussion

1. Optimal Sub-Structure – Recursion?

- $C(n, k) = C(n-1, k-1) + C(n-1, k)$ for $n > k > 0$
 - How? – next slide
- $C(n, 0) = C(n, n) = 1$

2. Overlapping Subproblems?

- Delineate $C(5, 2)$ - for instance



• Step 1: Let's write simple-recursive Algorithm

- ```
int C(int n, int k) {
 • If (n == k) or (k == 0)
 • return 1;
 • else
 • return (C(n-1, k-1) + C(n-1, k));
}
```

### • Step 2: Memoization

- Like any other DP algorithm, re-computations of the same subproblems can be avoided by constructing a temporary  $C[][]$  in **bottom up** Approach.
  - Top-down vs. Bottom-up?
  - What's the trend?
    - Hints:
      - Do you always need to compute/visit ALL subproblems?
      - Recursion vs. Iterative

$$C(n, k) = C(n-1, k-1) + C(n-1, k) \text{ for } n > k > 0$$

- “Here is perhaps a simple proof”:
  - $C(n, k)$  is the number of ways to select  $k$  objects from  $n$  distinct ones.
  - Now let us say we have marked a specific object and want to count the selections with and without this object:
    - You can select  $k$  objects without the marked one in  $C(n-1, k)$  ways,
    - and
    - including the marked one in  $C(n-1, k-1)$

# Cont - Practice 1

## Binomial Coefficients – Discussion

- Code both versions –

# TIPs on Memoization Vs. Tabulation

|                           | Tabulation                                                                                                                                                        | Memoization                                                                                                                                                                   |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>State</b>              | State Transition relation is difficult to think                                                                                                                   | State transition relation is easy to think                                                                                                                                    |
| <b>Code</b>               | Code gets complicated when lot of conditions are required                                                                                                         | Code is easy and less complicated                                                                                                                                             |
| <b>Speed</b>              | Fast, as we directly access previous states from the table                                                                                                        | Slow due to lot of recursive calls and return statements                                                                                                                      |
| <b>Subproblem solving</b> | If all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms a top-down memoized algorithm by a constant factor | If some subproblems in the subproblem space need not be solved at all, the memoized solution has the advantage of solving only those subproblems that are definitely required |
| <b>Table Entries</b>      | In Tabulated version, starting from the first entry, all entries are filled one by one                                                                            | Unlike the Tabulated version, all entries of the lookup table are not necessarily filled in Memoized version. The table is filled on demand.                                  |



# RE-CAPITULATION

## *Tabulation Vs. Memoization*

## *Bottom-Up Vs. Top-down*

## Fibonacci Nbrs as an illustration

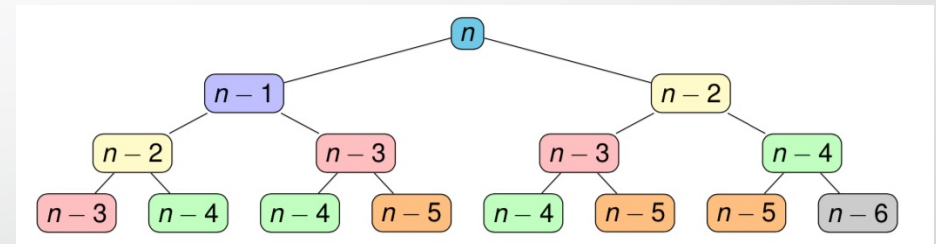
- Recursive Solution:

**RecFib( $n$ )**

1. if  $n \leq 0$
2.     return 0
3. if  $n = 1$
4.     return 1
5. return RecFib( $n - 1$ ) + RecFib( $n - 2$ )

- However, RecFib( $n$ ) has running time exponential in  $n$
- Exercise: Prove it!

- Inefficient: Infinitely recomputes solutions to subproblems



- Solution: DP / Memoization

**MemoFib( $n$ )**

1. if  $n \leq 0$
2.     return 0
3. if  $n = 1$
4.     return 1
5. if  $F[n]$  undefined
6.      $F[n] \leftarrow \text{MemoFib}(n - 1) + \text{MemoFib}(n - 2)$
7. return  $F[n]$

# Improving the Algorithm

## Tabulation – Bottom-Up

- Something unnatural about this algorithm: the numbers are requested from the top down, but filled from the bottom up!

### MemoFib( $n$ )

```
1. if $n \leq 0$
2. return 0
3. if $n = 1$
4. return 1
5. if $F[n]$ undefined
6. $F[n] \leftarrow \text{MemoFib}(n-1) + \text{MemoFib}(n-2)$
7. return $F[n]$
```

- That is, the  $F$  array is computed in the order  $F[0]$ ,  $F[1]$ , .....,  $F[n]$
- This leads to unnecessarily large number of recursive calls being made

- We can get rid of the recursion by simply computing the Fib numbers in ascending orders

### AscFib( $n$ )

```
1. $F[0] \leftarrow 0$
2. $F[1] \leftarrow 1$
3. for $i = 2$ to n
4. $F[i] \leftarrow F[i-1] + F[i-2]$
5. return $F[n]$
```

- This may be the natural algorithm one would come up with when first looking at the problem, but the point is that here we found it almost completely mechanically from the original recurrence!

# Final TIP

## Bottom-Up vs. Top-down

- Although this problem was very simple, it illustrates the basic concepts behind DP:
  1. Start out with a problem which can be presented recursively in terms of overlapping subproblems.
  2. Write a naïve recursive algorithm based on this presentation.
  3. Memoize the recursive algorithm
  4. Finally, restructure the algorithm to obtain a “bottom-up” algorithm which computes solutions in an efficient order, with no recursion.

# Relevant problem:

## UVA 1649

### 1649 Binomial coefficients

Gunnar is quite an old and forgetful researcher. Right now he is writing a paper on security in social networks and it actually involves some combinatorics. He wrote a program for calculating binomial coefficients to help him check some of his calculations.

A binomial coefficient is a number

$$\binom{n}{k} = \frac{n!}{k!(n-k)!},$$

where  $n$  and  $k$  are non-negative integers.

Gunnar used his program to calculate  $\binom{n}{k}$  and got a number  $m$  as a result. Unfortunately, since he is forgetful, he forgot the numbers  $n$  and  $k$  he used as input. These two numbers were a result of a long calculation and they are written on one of many papers lying on his desk. Instead of trying to search for the papers, he tried to reconstruct the numbers  $n$ ,  $k$  from the output he got. Can you help him and find all possible candidates?

#### Input

On the first line a positive integer: the number of test cases, at most 100. After that per test case:

- one line with an integer  $m$  ( $2 \leq m \leq 10^{15}$ ): the output of Gunnar's program.

# Relevant problem: UVA 1649 – Cont,

## Output

Per test case:

- one line with an integer: the number of ways of expressing  $m$  as a binomial coefficient.
- one line with all pairs  $(n, k)$  that satisfy  $\binom{n}{k} = m$ . Order them in increasing order of  $n$  and, in case of a tie, order them in increasing order of  $k$ . Format them as in the sample output.


## Sample Input

```
2
2
15
```

## Sample Output

```
1
(2,1)
4
(6,2) (6,4) (15,1) (15,14)
```





# UVA 1649

## Solution Discussion

# Practice 2

## LCS

- 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.
  - For example, "abc", "abg", "bdf", "aeg", "acefg", .. etc are subsequences of "abcdefg"
- Applications:
  - Bioinformatics! (DNA Sequencing)
  - Data comparison: diff program (Difference between 2 files)
  - Linguistics
  - Versioning Control Systems
  - Authentication!
  - Etc.

# LCS Promising Applications

## *Genomics!*

- DNA Sequences can be viewed as a String of A, C, G, and T characters, which represent nucleotides.
- Finding the similarities between 2 DNA sequences is an important computation performed in Bioinformatics
  - For instance, when comparing the DNA of different organisms, such as alignments can highlight the location where the organisms have identical DNA patterns.
- In Genomics, analyzing the Genome sequence will allow detecting vulnerability for diseases, produce personalized medicine, discovering evolutionary history and individuals ancestral roots, etc.
- DNA sequencing is the most appealing “**killer app.**” on the horizon!:
  - alike “DNA ancestry” costing about 100\$ by now!, it is expected to fall down to a couple of 10\$ in the couple of decades ahead!
  - Everybody will be interested in getting his Genome sequenced!
  - Big Data Explosion!

# LCS - Genomics

- Finding the best alignment between two DNA strings involves minimizing the number of changes to convert one string to the other
- A brute-force search would take exponential time: WE can do much better using DP!
- Computational biology [Needleman-Wunsch, 1970's] – Simple measure of genome similarity
  - $n - \text{length}(\text{LCS}(x,y))$  is called the “edit distance”

```
X: ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
 | | | | | | | | | | | | |
 G TC GT CG G AAGCCGGCCGAA
GTCGT CGGAA GCCG GC C G AA
 | | | | | | | | | | | |
Y: GTCGTTCGGAATGCCGTTGCTCTGTAA
```

# LCS

- **Brute Force**

- Check every subsequence of  $x[1 \dots m]$  to see if it is also a subsequence of  $y[1 \dots n]$
- Analysis
  - $(2^{\text{pow } m})$  subsequences
  - exponential time.

- **Recursion / DP**

$$c[i, j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max \{c[i-1, j], c[i, j-1]\} & \text{otherwise.} \end{cases}$$

- Base case:  $c[i, j] = 0$  if  $i = '\text{\textbackslash}0'$  or  $j = '\text{\textbackslash}0'$
- What about Memoization and Bottom-Up?



# LCS

## UVA#10405

### 10405 Longest Common Subsequence

Given two sequences of characters, print the length of the longest common subsequence of both sequences.

Sequence 1:



Sequence 2:



For example, the longest common subsequence of the following two sequences 'abcdgh' and 'aedfhr' is 'adh' of length 3.

#### Input

Input consists of pairs of lines. The first line of a pair contains the first string and the second line contains the second string. Each string is on a separate line and consists of at most 1,000 characters

#### Output

For each subsequent pair of input lines, output a line containing one integer number which satisfies the criteria stated above.

# LCS

## UVA#10405 – Cont,

### Sample Input

```
bcacbcabbaccbab
bccabccbbabacbc
a1b2c3d4e
zz1yy2xx3ww4vv
abcdgh
aedfhr
abcdefghijklmnopqrstuvwxyz
a0b0c0d0e0f0g0h0i0j0k0l0m0n0o0p0q0r0s0t0u0v0w0x0y0z0
abcdefghijklmnopzyxwvutsrqpo
opqrstuvwxyzabcdefghijklmn
```

### Sample Output

```
11
4
3
26
14
```

# LCS – Memoization

```
// R. Abid,
#include<stdio.h>
#include<string.h>

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

int LCS(char *X, char *Y, int m, int n, int memo[][100]) {
 if (m == 0 || n == 0)
 return 0;
 if (memo[m][n] != -1)
 return memo[m][n];
 if (X[m - 1] == Y[n - 1])
 return memo[m][n] = 1 + LCS(X, Y, m - 1, n - 1, memo);
 else
 return memo[m][n] = max(LCS(X, Y, m - 1, n, memo), LCS(X, Y, m, n - 1, memo));
}

int findLCS(char *X, char *Y, int m, int n) {
 int memo[100][100];
 memset(memo, -1, sizeof(memo)); // Initialize memoization table with -1
 return LCS(X, Y, m, n, memo);
}

int main() {
 char X[] = "ABC-UM6P-BEN-GUERIR-TODAY";
 char Y[] = "ACDF-UM6P-DAY";
 int m = strlen(X);
 int n = strlen(Y);
 int length = findLCS(X, Y, m, n);
 printf("Length of LCS: %d\n", length);
 return 0;
}

-- INSERT --
```

# LCS – Tabulation

```
// R. Abid,
#include <stdio.h>
#include <string.h>

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

int LCS_length(char* X, char* Y, int m, int n) {
 int dp[m + 1][n + 1];
 int i, j;

 // Building the table in a bottom-up manner
 for (i = 0; i <= m; i++) {
 for (j = 0; j <= n; j++) {
 if (i == 0 || j == 0)
 dp[i][j] = 0;
 else if (X[i - 1] == Y[j - 1])
 dp[i][j] = dp[i - 1][j - 1] + 1;
 else
 dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
 }
 }
 return dp[m][n];
}

int main() {
 char X[] = "ABCD-youNameIT-UM6P";
 char Y[] = "ACDF-NaIT-M6";
 int m = strlen(X);
 int n = strlen(Y);
 int lcs_length = LCS_length(X, Y, m, n);
 printf("Length of LCS: %d\n", lcs_length);
}
```

*The key always: Formulating a relation among the states*

## Practice 3 -

- **Problem:**
  - Given 3 numbers {1, 3, 4}, and a number N, we need to tell the total number of ways we can form the number 'N' using the sum of the given three numbers. (allowing repetitions and different arrangements).
- Example: for  $n = 5$ , the answer is 6
- 5

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

$$= 1+1+3$$

$$= 1+3+1$$

$$= 3+1+1$$

$$= 1+4$$

$$= 4+1$$



# Solution -

- Define subproblems
  - Let  $D_n$  be the number of ways to write  $n$  as the sum of 1,3,4
- Find the recurrence
  - Consider one possible solution
$$n = x_1 + x_2 + \dots + x_m$$
  - If  $x_m = 1$ , the rest of the terms must sum to  $n-1$
  - Thus, the number of sums that end with  $x_m = 1$  is equal to  $D_{n-1}$
  - Take other cases into account ( $x_m = 3, x_m = 4$ )
- Recurrence is then
$$D_n = D_{n-1} + D_{n-3} + D_{n-4}$$
- Solve the base cases
  - $D_0 = 1$
  - $D_n = 0$  for all negative  $n$
  - Alternatively, can set:  $D_0 = D_1 = D_2 = 1$ , and  $D_3 = 2$

# Implementation

- $D[0] = D[1] = D[2] = 1;$
- $D[3] = 2;$
- $\text{for}(i = 4; i \leq n; i++)$ 
  - $D[i] = D[i-1] + D[i-3] + D[i-4];$
- Very short!
- Extension: solving this for huge  $n$ ,
  - say  $n \approx 10^{12}$

ACM Competitive Programming Training (C) R. Abid @UM6P

```
// R. Abid,
#include <stdio.h>

// Declare a MEMO -
int Dn(int n) {
 if (n < 0)
 return 0;
 if (n == 0 || n == 1 || n == 2)
 return 1;
 return
 Dn(n-1) + Dn(n-3) + Dn(n-4);
}

/* Tabulation
D[0] = D[1] = D[2] = 1;
D[3] = 2;
for(i = 4; i <= n; i++)
 D[i] = D[i-1] + D[i-3] + D[i-4];
*/

int main() {
 int n;

 printf("\n Enter n: ");
 scanf("%d", &n);
 printf("-->: %d", Dn(n));
 return 0;
}
```

# Exercises / Assignments

- Wedding Shopping UVA#11450
- Classical Problems
  - MAX 1D Range Sum – UVA#507
    - MAX 2D Range Sum – UVA#108
  - LIS (Longest Increasing Subsequence) – UVA#11499
  - 0-1 Knapsack – UVA#10130
  - Coin Change – UVA#674

# It was a pleasure meeting you ...

***✍ Keep on the Hard Work!***



- and looking forward to successfully continue on coaching you:

-  **✍ Future UM6P Morocco Champions!**

*Yours,  
R. Abid,*