Unit-6

Dynamic programming

Dynamic Programming is a technique in computer programming that helps to efficiently solve a class of problems that have overlapping subproblems and optimal substructure property.
If any problem can be divided into subproblems, which in turn are divided into smaller subproblems, and if there are overlapping among these subproblems, then the solutions to these subproblems can be saved for future reference. In this way, efficiency of the CPU can be enhanced. This method of solving a solution is referred to as dynamic programming.

Such problems involve repeatedly calculating the value of the same subproblems to find the optimum solution.

Dynamic Programming *(DP) is a technique that solves some particular type of problems in* Polynomial Time. *Dynamic Programming solutions are faster than the exponential brute method and can be easily proved their correctness.*

To dynamically solve a problem, we need to check two necessary conditions:

- ✓ **Overlapping Subproblems:** When the solutions to the same subproblems are needed repetitively for solving the actual problem. The problem is said to have overlapping subproblems property.
- ✓ **Optimal Substructure Property:** If the optimal solution of the given problem can be obtained by using optimal solutions of its subproblems then the problem is said to have Optimal Substructure Property.

Steps to solve a Dynamic programming problem:

- ✓ Identify if it is a Dynamic programming problem.
- ✓ Decide a state expression with the Least parameters.
- ✓ Formulate state and transition relationship.
- ✓ Do tabulation (or memoization).

Step 1: How to classify a problem as a Dynamic Programming Problem?

- ✓ Typically, all the problems that require maximizing or minimizing certain quantities or counting problems that say to count the arrangements under certain conditions or certain probability problems can be solved by using Dynamic Programming.
- ✓ All dynamic programming problems satisfy the overlapping subproblems property and most of the classic Dynamic programming problems also satisfy the optimal substructure property. Once we observe these properties in a given problem be sure that it can be solved using Dynamic Programming.

Step 2: Deciding the state

Dynamic Programming problems are all about the **state** and its **transition**. This is the most basic step which must be done very carefully because the state transition depends on the choice of state definition you make.
State:
A state can be defined as the set of ***parameters*** *that can uniquely identify a certain position or standing in the given problem. This set of parameters should be as small as possible to reduce state space.*
Example:
In our famous Knapsack problem, we define our state by two parameters **index** and **weight** i.e DP[index][weight]. Here DP[index][weight] tells us the maximum profit it can make by taking items from range 0 to index having the capacity of sack to be weight. Therefore, here the parameters **index** and **weight** together can uniquely identify a subproblem for the knapsack problem.
The first step to solving a Dynamic Programming problem will be deciding on a state for the problem after identifying that

the problem is a Dynamic Programming problem. As we know Dynamic Programming is all about using calculated results to formulate the final result.
So, our next step will be to find a relation between previous states to reach the current state.

Step 3: Formulating a relation among the states

This part is the hardest part of solving a Dynamic Programming problem and requires a lot of intuition, observation, and practice.

Example:
Given 3 numbers {1, 3, 5}, The task is to tell the total number of ways we can form a number **N** using the sum of the given three numbers. (Allowing repetitions and different arrangements).
The total number of ways to form 6 is: 8
1+1+1+1+1+1
1+1+1+3
1+1+3+1
1+3+1+1
3+1+1+1
3+3
1+5
5+1

The steps to solve the given problem will be:

- ✓ We decide a state for the given problem.
- ✓ We will take a parameter **N** to decide the state as it uniquely identifies any subproblem.
- ✓ DP state will look like **state(N)**, state(N) means the total number of arrangements to form **N** by using {1, 3, 5} as elements. Derive a transition relation between any two states.
- ✓ Now, we need to compute state(N).

How to Compute the state?
As we can only use 1, 3, or 5 to form a given number **N.** Let us assume that we know the result for N = 1,2,3,4,5,6
Let us say we know the result for:
state (n = 1), state (n = 2), state (n = 3) ……… state (n = 6)
Now, we wish to know the result of the state (n = 7). See, we can only add 1, 3, and 5. Now we can get a sum total of 7 in the following 3 ways:

1) Adding 1 to all possible combinations of state (n = 6)
Eg : [(1+1+1+1+1+1) + 1]
[(1+1+1+3) + 1]
[(1+1+3+1) + 1]
[(1+3+1+1) + 1]
[(3+1+1+1) + 1]
[(3+3) + 1]
[(1+5) + 1]
[(5+1) + 1]

2) Adding 3 to all possible combinations of state (n = 4);
[(1+1+1+1) + 3]
[(1+3) + 3]
[(3+1) + 3]

3) Adding 5 to all possible combinations of state(n = 2)
[(1+1) + 5]
(Note how it sufficient to add only on the right-side – all the add-from-left-side cases are covered, either in the same state, or another, e.g. [1+(1+1+1+3)] is not needed in state (n=6) because it's covered by state (n = 4) [(1+1+1+1) + 3])
Now, think carefully and satisfy yourself that the above three cases are covering all possible ways to form a sum total of 7;
Therefore, we can say that result for
state (7) = state (6) + state (4) + state (2)
OR
state (7) = state (7-1) + state (7-3) + state (7-5)
In general,
state(n) = state(n-1) + state(n-3) + state(n-5)
Below is the implementation of the above approach:

```
int solve(int n)
{

    if (n < 0)
        return 0;
    if (n == 0)
        return 1;

    return solve(n-1) + solve(n-3) + solve(n-5);
}
```

Time Complexity: $O(3^N)$, As at every stage we need to take three decisions and the height of the tree will be of the order of n.

Auxiliary Space: O(N), The extra space is used due to the recursion call stack.

The above code seems exponential as it is calculating the same state again and again. So, we just need to add memorization.

Step 4: Adding memorization or tabulation for the state

This is the easiest part of a dynamic programming solution. We just need to store the state answer so that the next time that state is required, we can directly use it from our memory.

Adding memorization to the above code

```
int dp[MAXN];
int solve(int n)
{
  if (n < 0)
      return 0;
  if (n == 0)
      return 1;

  if (dp[n]!=-1)
      return dp[n];

return dp[n] = solve(n-1)+solve(n-3)+solve(n-5);
}
```

Time Complexity: O(n), As we just need to make 3n function calls and there will be no repetitive calculations as we are returning previously calculated results.

Auxiliary Space: O(n), The extra space is used due to the recursion call stack.

Another way is to add tabulation and make the solution iterative.

Recursion vs Dynamic Programming

Dynamic programming is mostly applied to recursive algorithms. This is not a coincidence; most optimization problems require recursion and dynamic programming is used for optimization.

But not all problems that use recursion can use Dynamic Programming. Unless there is a presence of overlapping subproblems like in the Fibonacci sequence problem, a recursion can only reach the solution using a divide and conquer approach.

That is the reason why a recursive algorithm like Merge Sort cannot use Dynamic Programming, because the subproblems are not overlapping in any way.

Greedy Algorithms vs Dynamic Programming

Greedy Algorithms are similar to dynamic programming in the sense that they are both tools for optimization.
However, greedy algorithms look for locally optimum solutions or in other words, a greedy choice, in the hopes of finding a global optimum. Hence greedy algorithms can make a guess that looks optimum at the time but becomes costly down the line and do not guarantee a globally optimum.

Dynamic programming, on the other hand, finds the optimal solution to subproblems and then makes an informed choice to combine the results of those subproblems to find the most optimum solution.

Different Approaches of Dynamic Programming

There are two approaches to formulate a dynamic programming solution:

Top-Down Approach

The top-down approach follows the memorization technique. It consists of two distinct events: recursion and caching. 'Recursion' represents the process of computation by calling functions repeatedly, whereas 'caching' represents the storing of intermediate results.

Advantages

- ✓ Easy to understand and implement
- ✓ Solves the subproblem only if the solution is not memorized.
- ✓ Debugging is easier.

Disadvantages

- ✓ Uses recursion, which takes up more memory space in the call stack, degrading the overall performance.
- ✓ Possibility of a stack overflow error.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

Key points

- ✓ We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- ✓ We use for loop to iterate over the sub-problems.
- ✓ The bottom-up approach is also known as the tabulation or table filling method.

Key Differences: Top-Down vs Bottom-Up Approach

Top-Down Approach	**Bottom-Up Approach**
Uses memorization technique	Uses tabulation technique
Recursive nature	Iterative nature

Structured programming languages such as COBOL, Fortran, C, and others mostly use this technique	Object-oriented programming languages such as C++, C#, and Python mostly use this technique
This approach uses decomposition to formulate a solution	This approach uses composition to develop a solution
A lookup table is maintained and checked before computation of any subproblem	The solution is built from a bottom-most case using iteration

Coin Change Problem:

Given an integer array of **coins[ ]** of size **N** representing different types of currency and an integer **sum**, The task is to find the number of ways to make **sum** by using different combinations from **coins[]**.

Note: Assume that you have an infinite supply of each type of coin.

Examples:

Input: *sum = 4, coins[] = {1,2,3},*

Output: *4*

Explanation: *there are four solutions: {1, 1, 1, 1}, {1, 1, 2}, {2, 2}, {1, 3}.*

Input: *sum = 10, coins[] = {2, 5, 3, 6}*

nput: *sum = 10, coins[] = {2, 5, 3, 6}*

Output: *5*

Explanation: *There are five solutions:*
{2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}.

Coin Change Problem using Recursion:

Solve the Coin Change is to traverse the array by applying the recursive solution and keep finding the possible ways to find the occurrence.

Follow the below steps to Implement the idea:

- We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.
- If we are at coins[n-1], we can take as many instances of that coin (unbounded inclusion) i.e **count(coins, n, sum – coins[n-1])**; then we move to coins[n-2].
- After moving to coins[n-2], we can't move back and can't make choices for coins[n-1] i.e **count(coins, n-1, sum)**.
- Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(coins, n, sum – coins[n-1]) + count(coins, n-1, sum);**

Follow the below steps to Implement the idea:

- We have 2 choices for a coin of a particular denomination, either i) to include, or ii) to exclude.
- If we are at coins[n-1], we can take as many instances of that coin (unbounded inclusion) i.e **count(coins, n, sum – coins[n-1])**; then we move to coins[n-2].
- After moving to coins[n-2], we can't move back and can't make choices for coins[n-1] i.e **count(coins, n-1, sum)**.
- Finally, as we have to find the total number of ways, so we will add these 2 possible choices, i.e **count(coins, n, sum – coins[n-1]) + count(coins, n-1, sum);**

Below is the Implementation of the above approach.

- C++

```
// Recursive C++ program for
// coin change problem.
#include <bits/stdc++.h>
using namespace std;

// Returns the count of ways we can
// sum coins[0...n-1] coins to get sum "sum"
int count(int coins[], int n, int sum)
```

```
{

    // If sum is 0 then there is 1 solution
    // (do not include any coin)
    if (sum == 0)
        return 1;

    // If sum is less than 0 then no
    // solution exists
    if (sum < 0)
        return 0;

    // If there are no coins and sum
    // is greater than 0, then no
    // solution exist
    if (n <= 0)
        return 0;

    // count is sum of solutions (i)
    // including coins[n-1] (ii) excluding coins[n-1]
    return count(coins, n - 1, sum)
           + count(coins, n, sum - coins[n - 1]);
}

// Driver code
int main()
{
    int i, j;
    int coins[] = { 1, 2, 3 };
    int n = sizeof(coins) / sizeof(coins[0]);
    int sum = 4;

    cout << " " << count(coins, n, sum);

    return 0;
}

// This code is contributed by shivanisinghss2110
```

Output

```
 4
```

Time Complexity: O(2^{sum})
Auxiliary Space: O(target)
Since the same sub-problems are called again, this problem has the Overlapping Subproblems property. So the Coin Change problem has both properties (see this and this) of a dynamic programming problem. Like other typical Dynamic Programming(DP) problems, recomputations of the same subproblems can be avoided by constructing a temporary array table[][] in a bottom-up manner.

Coin Change By Using Dynamic Programming:

Follow the below steps to Implement the idea:

- Using 2-D vector to store the Overlapping subproblems.
- Traversing the whole array to find the solution and storing in the memoization table.
- Using the memoization table to find the optimal solution.

Below is the implementation of the above Idea.

- C++

```
// C++ program for coin change problem

#include <bits/stdc++.h>

using namespace std;

int count(int coins[], int n, int sum)
{
    int i, j, x, y;

    // We need sum+1 rows as the table
    // is constructed in bottom up
    // manner using the base case 0
    // value case (sum = 0)
    int table[sum + 1][n];

    // Fill the entries for 0
    // value case (sum = 0)
    for (i = 0; i < n; i++)
        table[0][i] = 1;

    // Fill rest of the table entries
    // in bottom up manner
    for (i = 1; i < sum + 1; i++) {
        for (j = 0; j < n; j++) {
            // Count of solutions including coins[j]
            x = (i - coins[j] >= 0) ? table[i - coins[j]][j]
                                     : 0;

            // Count of solutions excluding coins[j]
            y = (j >= 1) ? table[i][j - 1] : 0;
            // total count
            table[i][j] = x + y;
        }
    }
    return table[sum][n - 1];
}

// Driver Code
int main()
{
    int coins[] = { 1, 2, 3 };
    int n = sizeof(coins) / sizeof(coins[0]);
    int sum = 4;
    cout << count(coins, n, sum);
    return 0;
}

// This code is contributed
// by Akanksha Rai(Abby_akku)
```

Output

```
4
```

Time Complexity: O(M*sum)
Auxiliary Space: O(M*sum)

Coin change Using the Space Optimized 1D array:

Follow the below steps to Implement the idea:

- Initialize with a linear array **table** with values equal to 0.
- With **sum = 0**, there is a way.
- Update the level wise number of ways of coin till the **ith** coin.
- Solve till **j <= sum.**

Below is the implementation of the above Idea.

- C++

```
#include <bits/stdc++.h>

using namespace std;

// This code is
int count(int coins[], int n, int sum)
{
    // table[i] will be storing the number of solutions for
    // value i. We need sum+1 rows as the table is
    // constructed in bottom up manner using the base case
    // (sum = 0)
    int table[sum + 1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // Pick all coins one by one and update the table[]
    // values after the index greater than or equal to the
    // value of the picked coin
    for (int i = 0; i < n; i++)
        for (int j = coins[i]; j <= sum; j++)
            table[j] += table[j - coins[i]];
    return table[sum];
}

// Driver Code
int main()
{
    int coins[] = { 1, 2, 3 };
    int n = sizeof(coins) / sizeof(coins[0]);
    int sum = 4;
    cout << count(coins, n, sum);
    return 0;
}
```

Output

```
4
```

Time Complexity: O(N*sum)
Auxiliary Space: O(sum)

Coin change using the Top Down (Memoization) Dynamic Programming:

Follow the below steps to Implement the idea:

- Creating a 2-D vector to store the Overlapping Solutions
- Keep Track of the overlapping subproblems while Traversing the array **coins[]**
- Recall them whenever needed

Below is the implementation using the Top Down Memoized Approach

- C++

```cpp
#include <bits/stdc++.h>
using namespace std;

int coinchange(vector<int>& a, int v, int n,
               vector<vector<int> >& dp)
{
    if (v == 0)
        return dp[n][v] = 1;
    if (n == 0)
        return 0;
    if (dp[n][v] != -1)
        return dp[n][v];
    if (a[n - 1] <= v) {
        // Either Pick this coin or not
        return dp[n][v] = coinchange(a, v - a[n - 1], n, dp)
                          + coinchange(a, v, n - 1, dp);
    }
    else // We have no option but to leave this coin
        return dp[n][v] = coinchange(a, v, n - 1, dp);
}
int32_t main()
{
    int tc = 1;
    // cin >> tc;
    while (tc--) {
        int n, v;
        n = 3, v = 4;
        vector<int> a = { 1, 2, 3 };
        vector<vector<int> > dp(n + 1,
                                vector<int>(v + 1, -1));
        int res = coinchange(a, v, n, dp);
        cout << res << endl;
    }
}
// This Code is Contributed by
// Harshit Agrawal NITT
```

Output

```
4
```

Time Complexity: O(N*sum)
Auxiliary Space: O(N*sum)
This article is contributed by: **Mayukh Sinha.** Please write comments if you find anything incorrect, or if you want to share more information about the topic discussed above.

Road Cutting problem

Problem Statement

Given a rod of length n inches and an array of prices that includes prices of all pieces of size smaller than n. determine the maximum profit you could obtain from cutting up the rod and selling its pieces.

For example,

length[] = [1, 2, 3, 4, 5, 6, 7, 8]

price[] = [1, 5, 8, 9, 10, 17, 17, 20]

Rod length: 4

Best: Cut the rod into two pieces of length 2 each to gain revenue of 5 + 5 = 10

Cut	Profit
4	9

```
1, 3                    (1 + 8) = 9

2, 2                    (5 + 5) = 10

3, 1               (8 + 1) = 9
```

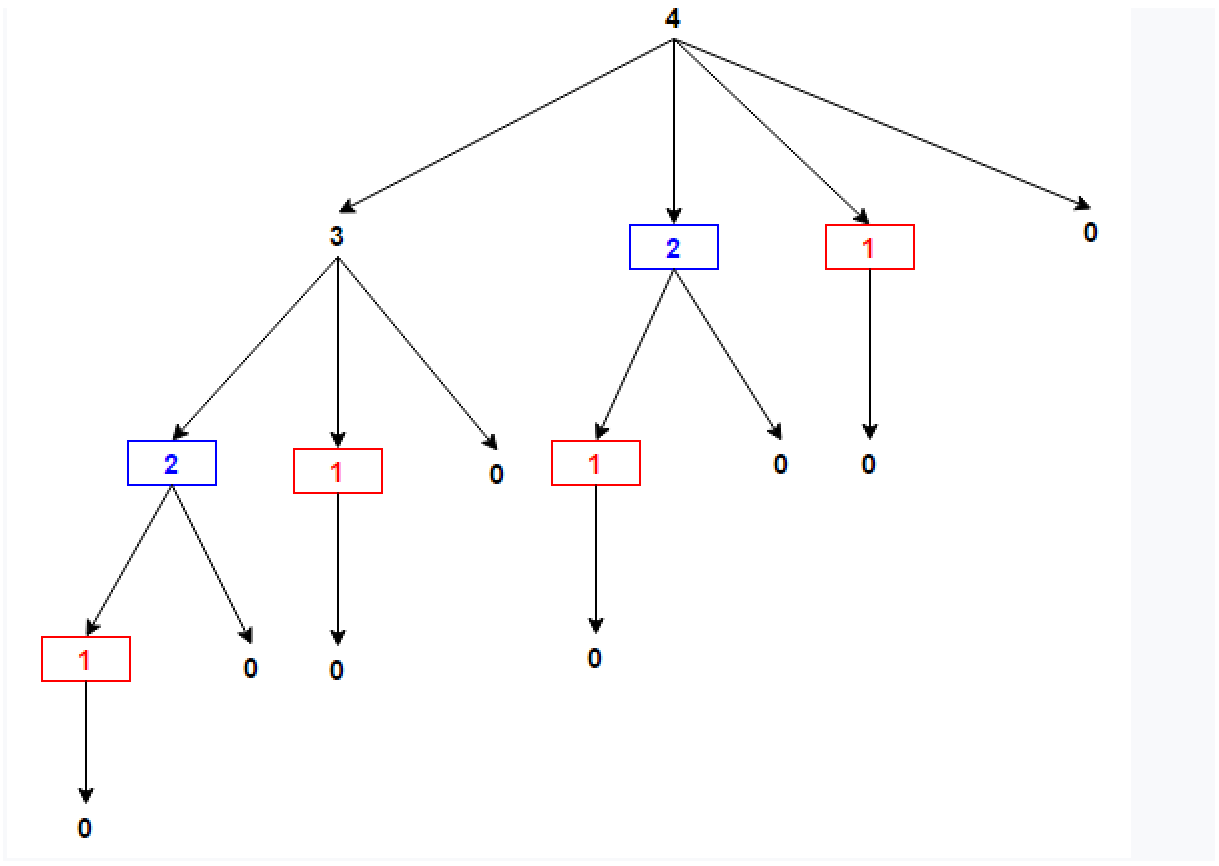
```
1, 1, 2        (1 + 1 + 5) = 7

1, 2, 1        (1 + 5 + 1) = 7

2, 1, 1        (5 + 1 + 1) = 7

1, 1, 1, 1     (1 + 1 + 1 + 1) = 4
```

We are given an array price [], where the rod of length i has a value price[i-1]. The idea is simple – one by one, partition the given rod of length n into two parts: i and n-i. Recur for the rod of length n-i but don't divide the rod of length i any further. Finally, take the maximum of all values. This yields the following recursive relation:

roadcut(n) = max {price [i – 1] + roadcut(n – i) } where 1 <= i <= n

The time complexity of the above solution is O(n^n) and occupies space in the call stack, where n is the rod length.

We have seen that the problem can be broken down into smaller subproblems, which can further be broken down into yet smaller subproblems, and so on. So, the problem has optimal substructure. Let's consider a recursion tree for the rod of length 4.



As we can see, the same subproblems (highlighted in the same color) are getting computed repeatedly. So, the problem also exhibits overlapping subproblems. We know that problems with optimal substructure and overlapping subproblems can be solved by dynamic programming, where subproblem solutions are memorized rather than computed and again.

We will solve this problem in a bottom-up manner. In the bottom-up approach, we solve smaller subproblems first, then solve larger subproblems from them. The following bottom-up approach computes T[i], which stores maximum profit achieved from the rod of length i for each 1 <= i <= n. It uses the value of smaller values i already computed.

CPP Code implementation:

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

int rodCut(int price[], int n)
{
    int T[n + 1];
    for (int i = 0; i <= n; i++)
    {
        T[i] = 0;
    }
    for (int i = 1; i <= n; i++)
    {

        for (int j = 1; j <= i; j++)
         {
            T[i] = max(T[i], price[j - 1] + T[i - j]);
        }
    }
     return T[n];
}

int main()
{
    int price[] = { 1, 5, 8, 9, 10, 17, 17, 20 };

    int n = 4;

    cout << "Profit is " << rodCut(price, n);

    return 0;

    }
```

Egg dropping problem

Problem Statement

You are given N eggs, and building with K floors from 1 to K. Each egg is identical, you drop an egg and if an egg breaks, you cannot drop it again. You know that there exists a floor F with 0 <= F <= K such that any egg dropped at a floor higher than F will break, and any egg dropped at or below floor F will not break. Each move, you may take an egg (if you have an unbroken one) and drop it from any floor X (with 1 <= X <= K). Your goal is to know with certainty what the value of F is.

What is the minimum number of moves that you need to know with certainty what F is, regardless of the initial value of F?

Example with explanation:

Input: T=1 N=1 K=2

Output: 2

Drop the egg from floor 1.

If it breaks, we know with certainty that F = 0.

Otherwise, drop the egg from floor 2.

If it breaks, we know with certainty that F = 1.

If it didn't break, then we know with certainty F = 2.

Hence,

```
we needed 2 moves in the worst case to know what F is with
certainty.

Input: T=1 N=2 K=100

Output: 14

Minimum number of trials that we would need is 14

to find the threshold floor F.

Here will declare the state of our dynamic programming as
dp[n][k] where n is the number of eggs we have and k is the
floor we have at any instant of the moment.

There are base cases,

dp[0][i]=0 if there is no egg then simply the answer is 0.

dp[1][i]=i if we have 1 egg then in the worst case we have to
attempt all the floor to get the min number of attempts in the
worst case.

We will see both of the approach top-down and bottom-up approach.

Initially, we will fill all dp[n][k] with -1 for memorization
approach.

So, that if it is calculated then return it immediately without
calculating it again.

CPP Code implementation using Top-Down Approach
```

```cpp
#include <bits/stdc++.h>
using namespace std;

typedef long long ll;
ll dp[1004][1004];

ll Drop(ll n, ll k)
{
    if (k == 0 || k == 1)
        return k;
    if (n <= 0)
        return 0;
    if (dp[n][k] != -1)
```

```
        return dp[n][k];
    ll ans = INT_MAX;
    for (ll i = 1; i <= k; i++)
    {

 ll subres = max(Drop(n - 1, i - 1), Drop(n, k - i));
     ans = min(ans, subres);
    }

    dp[n][k] = 1 + ans;
    return dp[n][k];
}

int main()
{
      cout << "Enter number of eggs and floors: ";
      ll n, k;
      cin >> n >> k;
  for (ll i = 0; i <= n; i++)
   for (ll j = 0; j <= k; j++)
        dp[i][j] = -1;

        dp[0][0] = 0;

        for (ll i = 1; i <= k; i++) {
            dp[0][i] = 0;
            dp[1][i] = i;
        }

        ll res = Drop(n, k);

        cout << "Minimum number of attempts: ";
        cout << res << "\n";

    return 0;

}
```

0/1 Knapsack problem

Problem Statement

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

0/1 knapsack problem

Fractional knapsack problem

We will discuss learn about the 0/1 knapsack problem.

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by the dynamic programming.

Cpp code implementation:

```
#include <bits/stdc++.h>
using namespace std;

int knapSack(int W, int wt[], int val[], int n)
{
    int dp[W + 1];
    memset(dp, 0, sizeof(dp));
    for (int i = 1; i < n + 1; i++) {
        for (int w = W; w >= 0; w--) {

            if (wt[i - 1] <= w)

   dp[w] = max(dp[w],dp[w - wt[i - 1]] + val[i - 1]);
        }
    }
    return dp[W];
}

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);
    return 0;

}
```

Longest common Sub Sequence problem

Problem Statement

Find the length of the longest increasing subsequence inside a given array. The longest increasing subsequence is a subsequence within an array of numbers with an increasing order. The numbers within the subsequence have to be unique and in ascending order.

Also, the items of the sequence do not need to be consecutive.

Algorithm

✓ Start with a recursive approach where you calculate the value of the longest increasing subsequence of every possible subarray from index zero to index i, where i is lesser than or equal to the size of the array.

- ✓ To turn this method into a dynamic one, create an array to store the value for every subsequence. Initialise all the values of this array to 0.
- ✓ Every index i of this array corresponds to the length of the longest increasing subsequence for a subarray of size i.
- ✓ Now, for every recursive call of findLIS(arr, n), check the nth index of the array. If this value is 0, then calculate the value using the method in the first step and store it at the nth index.
- ✓ Finally, return the maximum value from the array. This is the length of the longest increasing subsequence of a given size n.

CPP code Implementation:

```
#include <bits/stdc++.h>
using namespace std;


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 main()
{
    char X[] = "AGGTABQWEWA";
    char Y[] = "GXTXAYBQE";

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

    cout<<"Length of LCS is "<< lcs( X, Y, m, n ) ;

    return 0;

        }
```