University of Dhaka

Department of Computer Science and Engineering

Design and Analysis of Algorithm

Dynamic Programming- DP

Submitted To:

Hasnain Heickal
Asst. Professor
Department of Computer
Science and Engineering

Submitted By :

Sakib Hasan
Roll : 149

Tammana Sultana
Roll : 61

1 What is Dynamic Programming?

Dynamic programming is an **optimization** approach that transforms a complex problem into a sequence of simpler problems; its essential characteristic is the multistage nature of the optimization procedure. More so than the optimization techniques described previously, dynamic programming provides a general framework for analyzing many problem types. Within this framework a variety of optimization techniques can be employed to solve particular aspects of a more general formulation. Usually creativity is required before we can recognize that a particular problem can be cast effectively as a dynamic program; and often subtle insights are necessary to restructure the formulation so that it can be solved effectively.

In easier words, Dynamic problem(DP) refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

1.1 General Concepts

Let us learn about some general concepts before diving in to dynamic programming.

- ◇ Algorithm strategies
  - – Approach to solving a problem
  - – May combine several approaches
- ◇ Algorithm structures
  - – Iterative ⇒ execute action in loop
  - – Recursive ⇒ reapply action to sub-problem(s)
- ◇ Problem types
  - – Decision ⇒ find Yes/No answer
  - – Satisfying ⇒ find any satisfactory solution
  - – Optimization ⇒ find best solutions (vs. cost metric)

Now, dynamic programming is an algorithmic strategy where we try to break the original problem into smaller similar sub-problems and try to find optimal solution in recursive manner which refers to optimization. DP is based on remembering past results, so we also have to store already calculated data from smaller sub-problems solution. DP is generally used to solve **'Optimization Problems'**.

1.2 Approach -

(i) Divide problem into smaller sub-problems

- Sub-problems must be of same type
- Sub-problems must overlap

(ii) Solve each sub-problem recursively

- May simply look up solution

(iii) Combine solutions into to solve original problem

(iv) Store solution to problem (**'Tabulation'** or **'Memoization'**)

1.3 Two key ingredients of dynamic programming

1. Optimal substructures
2. Overlapping sub-problems

Optimal substructures :

A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.
For example, the Shortest Path problem has following optimal substructure property: If a node x lies in the shortest path from a source node u to destination node v then the shortest path from u to v is combination of shortest path from u to x and shortest path from x to v. The standard All Pair Shortest Path algorithms like Floyd–Warshall and Bellman–Ford are typical examples of Dynamic Programming.

On the other hand, the Longest Path problem doesn't have the Optimal Substructure property. Here by Longest Path we mean longest simple path (path without cycle) between two nodes. Consider the following unweighted graph. There are two longest paths from q to t: q→r→t and q→s→t. Unlike shortest paths, these longest paths do not have the optimal substructure property. For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

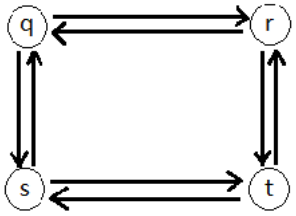

Figure 1: Sample Graph.

Overlapping sub-problems :

Like Divide and Conquer, Dynamic Programming combines solutions to sub-problems. Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed. So Dynamic Programming is not useful when there are no common (overlapping) subproblems because there is no point storing the solutions if they are not needed again. For example, Binary Search doesn't have common subproblems. If we take an example

of following recursive program for Fibonacci Numbers, there are many subproblems which are solved again and again.

```
/* simple recursive program for Fibonacci numbers */
int fib(int n)
{
   if ( n <= 1 )
      return n;
   return fib(n-1) + fib(n-2);
}
```

Recursion tree for execution of fib(5) We can see that the function fib(3) is being called

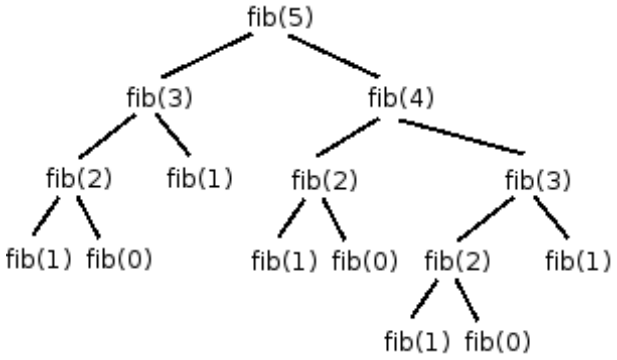

Figure 2: Fibonacci tree

2 times. If we would have stored the value of fib(3), then instead of computing it again, we could have reused the old stored value. There are following two different ways to store the values so that these values can be reused:

(a) Memoization (Top Down)

(b) Tabulation (Bottom Up)

So, to use DP, subproblems must be dependent/overlapping. Otherwise, a divide-and-conquer approach is the choice.

1.4 Tabulation vs Memoization

****Tabulation – Bottom Up Dynamic Programming****

As the name itself suggests starting from the bottom and cumulating answers to the top.Let's describe a state for our DP problem to be dp[x] with dp[0] as base state and dp[n] as our destination state. So, we need to find the value of destination state dp[n].If we start our transition from our base state i.e dp[0] and follow our state transition relation to reach our destination state dp[n], we call it Bottom Up approach as it is quite clear that we started our transition from the bottom base state and reached the top most desired state.

```
// Bottom up version to find factorial x.
int dp[max];

// base case
int dp[0] = 1;
for (int i = 1; i< =n; i++)
{
    dp[i] = dp[i-1] * i;
}
```

The above code clearly follows the bottom-up approach as it starts its transition from the bottom-most base case dp[0] and reaches its destination state dp[n]. Here, we may notice that the dp table is being populated sequentially and we are directly accessing the calculated states from the table itself and hence, we call it tabulation method.

**** Memoization Method – Top Down Dynamic Programming****

If we need to find the value for some state say dp[n] and instead of starting from the base state that dp[0] we ask our answer from the states that can reach the destination state dp[n] following the state transition relation, then it is the top-down fashion of DP. So, we start from dp[n] and to calculate that we go down to dp[n-1] ... ... until we are done calculating and storing dp[0] in this manner. Let's have a look into the code of finding factorial x in this method.

```
// Top down version to find factorial x.
// To speed up we store the values of calculated states

// initialized to -1
int dp[n]

// return fact x!
int solve(int x)
{
    if (x==0)
        return 1;
    if (dp[x]!=-1)
        return dp[x];
    return (dp[x] = x * solve(x-1));
}
```

As we can see we are storing the most recent solution such a way that if next time we got a call from the same state we simply return it from the memory. So, this is why we call it memoization as we are storing the most recent state values.

	Bottom-Up	Top Down
State	State Transition relation is difficult to think	State transition relation is easy to think
Code	Code gets complicated when lot of conditions are required	Code is easy and less complicated
Speed	Fast, as we directly access previous states from the table	Slow due to lot of recursive calls and return statements
Subproblem solving	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
Table Entries	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.

Figure 3: Comparison between Bottom-up and Top down approach.

2 Examples of Dynamic Programming Problems

Basic problems -

1. Fibonacci numbers
2. Tiling Problems
3. Coin change problem
4. Subset sum problem
5. Longest Common Sub-sequence
6. Longest Repeated Subsequence
7. Longest Increasing Subsequence
8. LCS (Longest Common Subsequence) of three strings
9. Warshall's All pairs shortest path
10. Bellman Ford's Single Source Shortest Path

Advanced problems -

1. Matrix chain multiplication
2. BitMasking
3. Digit DP

and so on. Here, we will be discussing about Tiling problem first.