Lecture 9-DYNAMIC PROGRAMMING

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So, we can say that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
- Dynamic algorithms use memorization.

Example

The following computer problems can be solved using dynamic programming approach −

- Fibonacci number series
- Knapsack problem

is an algorithm design technique for *optimization problems.*Like divide and conquer, DP solves problems by combining solutions to sub-problems. Unlike divide and conquer, sub-problems are not independent Sub-problems may share sub-sub-problems.

The term Dynamic Programming comes from Control Theory, not computer science. Programming refers to the use of tables (arrays) to construct a solution.

In dynamic programming we usually reduce time by increasing the amount of space. We solve the problem by solving sub-problems of increasing size and saving each optimal solution in a table (usually). The table is then used for finding the optimal solution to larger problems. Time is saved since each sub-problem is solved only once.

Consider again the recursive function for computing the nth Fibonacci number.

```
int Fibr(int n)
{ if (n <= 1) return 1; // Base case
return Fibr(n-1) + Fibr(n-2); // Recursive call
}
```

The cost of this algorithm (in terms of function calls) is the size of the nth Fibonacci number itself, which our analysis showed to be exponential (2 to power N), Why is this so expensive? It is expensive primarily because two recursive calls are made by the function, and they are largely redundant. That is, each of the two calls is recomputing most of the series, as is each sub-call, and so on. Thus, the smaller values of the function are being recomputed a huge number of times. If we could eliminate this redundancy, the cost would be greatly reduced. One way to accomplish this goal is to keep a table of values, and first check the table to see if the computation can be avoided. Here is a straightforward example of doing so.

```
int Fibrt(int n, int* Values)
 { // Assume Values has at least n slots, and all
   // slots are initialized to 0
if (n <= 1) return 1; // Base case
 if (Values[n] != 0)
return Values[n];
 Values[n] = Fibr(n-1, Values) + Fibr(n-2, Values);
return Values[n];
}
```

This version of the algorithm will not compute a value more than once, so its cost should be linear. Of course, we didn't actually need to use a table. Instead, we could build the value by working from 0 and 1 up to n rather than backwards from

n down to 0 and 1. Going up from the bottom we only need to store the previous two values of the function, as is done by our iterative version.

```
int Fibi(int n)
{ int past, prev, curr;
past = prev = curr = 1; // curr holds Fib(i)
for (int i=2; i<=n; i++) { // Compute next value
 past = prev; prev = curr; // past holds Fib(i-2)
curr = past + prev; // prev holds Fib(i-1)
}
 return curr; }
```

This issue of recomputing sub problems comes up frequently. In many cases, arbitrary sub problems (or at least a wide variety of sub problems) might need to be recomputed, so that storing sub results in a fixed number of variables will not work. Thus, there are many times where storing a table of sub results can be useful. Thisapproachtodesigninganalgorithmthatworksbystoringatableofresults for sub problems is called dynamic programming. The name is somewhat arcane, because it doesn't bear much obvious similarity to the process that is taking place of storing sub problems in a table. However, it comes originally from the field of dynamic control systems, which got its start before what we think of as computer programming. The act of storing precomputed values in a table for later reuse is referred to as "programming" in that field.

Dynamic programming is a powerful alternative to the standard principle of divide and conquer. In divide and conquer, a problem is split into sub problems, the sub problems are solved (independently), and the recombined into a solution for the problem being solved.

Dynamic programming is appropriate whenever the sub problems to be solved are overlapping in some way. Whenever this happens, dynamic programming can be used if we can find suitable way of doing the necessary book keeping. Dynamic programming algorithms are usually not implemented by simply using a table to store sub problems for recursive calls (i.e., going backwards as is done by Fibrt). Instead, such algorithms more typically implemented by building the table of sub problems from the bottom up. Thus, Fibi is actually closer in spirit to dynamic programming than is Fibrt even though it doesn't need the actual table.