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.

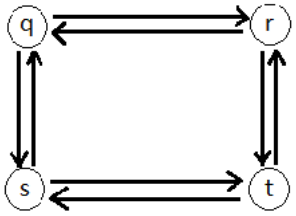


Figure 1: Sample Graph.

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. [i.e: Detailed discussion for fibonacchi problem solve in DP will be in section 2.1]

```
/* 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

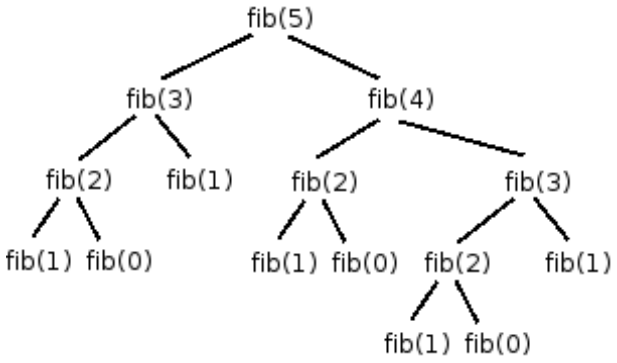


Figure 2: Fibonacci tree

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.

2.1 Overview on Dynamic programming solving Fibonacci number problem

The Fibonacci numbers are the numbers in the following integer sequence.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 141, ...
In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation.

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

with seed values $F_0 = 0$ and $F_1 = 1$ Write a function int **fib(int n)** that returns F_n . For example, if n = 0, then fib() should return 0. If n = 1, then it should return 1. For n > 1, it should return $F_{n-1} + F_{n+2}$.
Following are different methods to get the nth Fibonacci number

Method 1 (Use recursion)

A simple method that is a direct recusrive implementation mathematical recurance relation given above.

```
#include<stdio.h>
int fib(int n){
if (n <= 1) return n;
return fib(n-1) + fib(n-2); // recursive call of fib(n) here
}
int main (){
int n;

/* Scan n as nth number */
 print(fib(n)) // returns the value from recursive function

return 0;
}
```

Time Complexity: T(n) = T(n-1) + T(n-2) which is exponential.
We can observe that this implementation does a lot of repeated work (see the recursion tree from Fig:2 on page 3). So this is a bad implementation for nth Fibonacci number.
Extra Space: O(n) if we consider the fuinction call stack size, otherwise O(1).

Method 2 (Use Dynamic Programming)

We can avoid the repeated work done is the method 1 by storing the Fibonacci numbers calculated so far.

```
int fib(int n){
/* Declare an array to store fibonacci numbers. */
int f[n+1];
int i;
```

```
/* 0th and 1st number of the series are 0 and 1*/
f[0] = 0;
f[1] = 1;
for (i = 2; i <= n; i++){
/* Add the previous 2 numbers in the series and store it */
f[i] = f[i-1] + f[i-2];
}
return f[n];
}
int main (){
int n ;
print( fib(n))
return 0;
}
```

Time Complexity: O(n)
Extra Space: O(n)

Method 3 (Space Otimized Method 2)

We can optimize the space used in method 2 by storing the previous two numbers only because that is all we need to get the next Fibannaci number in series.

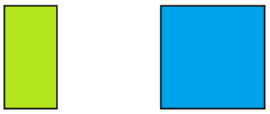
```
/* Optimized version function of method 2*/
int fib(int n){
int a = 0, b = 1, c, i;
if( n == 0)
return a;
for (i = 2; i <= n; i++){
c = a + b;
a = b;
b = c;
}
return b;
}

int main (){
int n;
/* scan n*/
/*printf(fib(n))*/
getchar();
return 0;
}
```

Time Complexity: O(n)
Extra Space: O(1)
This problem can also be solved in O(logN) complexity using optimized Matrix method and

3 Tiling Problem

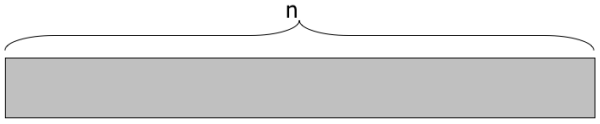
In tiling problem, the problem states that,
Given a 2*N board where N is any integer value, how many ways can you fill it up with given constant size tiles?

Available tiles 2x1 or 2x2.



Now when n = 3 there are 5 possible ways to fill up the board with given tiles.

consider board of width n:

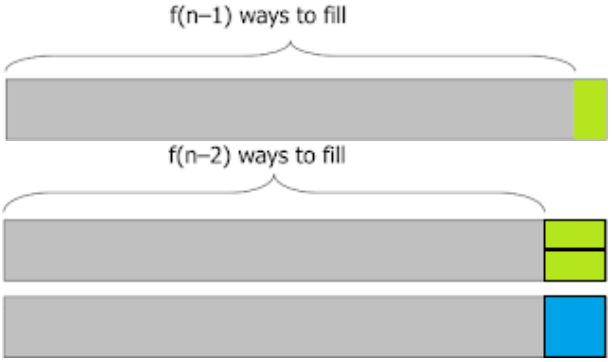


So, possible ways to fill the rightmost column :
1. By one 2×1 vertically.

2. By two of 2×1 horizontally.

3. By one 2×2 either horizontally or vertically(since both are same).

Let f(n) be the number of ways of filling a board of size 2×n.
So, f(n) = f(n-1) + f(n-2) + f(n-2)



Now, let's determine the base cases.
1. f(0) = 1, because if we have to find f(1) then , f(1) = f(0) + f(-1) + f(-1) , since f(-1) not exist, so f(-1) = 0 then f(1) = f(0) + 0 + 0 => f(1) = f(0) since there is one way to fill the board of size 2×1 by using one 2×1 tile. It means that f(0) must be equal to 1.
2. f(1) = 1

Algorithm/Code:

```
int ways[x];

int filling_tiles(int n){
   if(ways[n] != -1) return ways[n];
   return (ways[n] = f(n-1) + 2*f(n-2));
}


int main(){
   ways[0];
   ways[1];
   for(int i =2; i < x ; i++) ways[i] = -1;
//cout << f(x) ;
return f(x)
```

Complexity: O(n)

Another problem:

Suppose this time N = 3 and a tile can be either placed horizontally 1x2 or vertically, 2x1.
For instance, below is a board A. How many ways can it be filled up using tile B(2x1 or 1x2 both are same just rotated)?

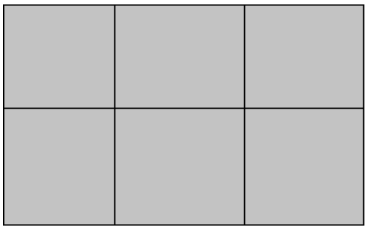


Figure 4: Board A - 2x3

Here N= 3 , so the board in 2x3. and given tile(s) is 2x1 or 1x2

Figure 5: tile B - 1x2

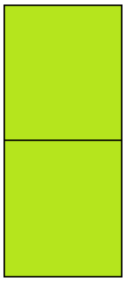


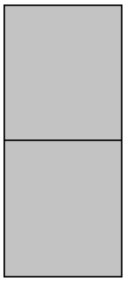
Figure 6: tile B - 2x1

2x1 tiles can be placed vertically or horizontally into 2xN tiles. You can place the tiles B on A by the following combinations.

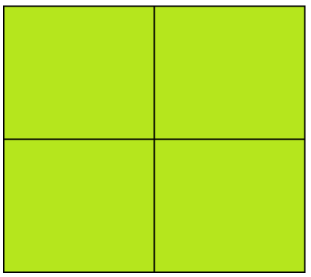
- All of (2x1) tiles Horizontally
- (1x2) - Twice vertically and (2x1) - once horizontally
- (2x1) - Twice horizontally and (1x2) - once vertically

That's all the combination possible for three tiles.

What if N was 1?
The 2x1- board would be as follows:



And to overlap all of it, tile-B could be used vertically once only. So, for N = 1 , answer would be one.
What if N was 2?
Then the figure(2x2) would be,



The following possible ways to fill up figure(2x2) using tiles(2x1) are :
–Twice Horizontally
–Twice Vertically
Apart from these two combination, there aren't any other options. So, for N's value 2, 2 combinations are possible. Because, so far we are getting linear values, the case still remains ambiguous on whether we would always get linear value answers.
Now, let's see the combinations for N's value 4.

- All tiles piled horizontally
- All tiles piled vertically
- Two horizontally and two vertically
- Two vertically and two horizontally.
- First and last horizontally and middle two vertically

Those are all the combinations possible for N's value 4. Notice for 4, the value is 5. If nicely presented, the data would look like below :

N	1	2	3	4
Number of Ways	1	2	3	5

So, as more tiles added, combinations won't increase linearly. But it seems for N's value 3, answer could be achieved by adding previous two data together. And same case follows for N's value 4. We could achieve 5's combination using previous two values.
For this problem, we have base case 1 and 2 for which we could get combination of 1 and 2 respectively. And a function of N can be formed by following way:

$$N(n) = N(n-1) + N(n-2)$$

Here n is the integer describe 2*n tiles. And n's data lies in data of n-1 and n-2.
Algorithm/Pseudo code is provided below:
Variables :
n : n is the integer which describes how far horizontally our input explores. 2*n.
Dp[i] : for any integer number i,

Here, Dp[i] stores the value of ith integer combination of way tiling input

Code:

```
Filling-Tiles (n)
{
    if(n is 1)
return 1; //base case one
    If (n is 2)
       return 2; // base case 2
   if (n is 3)
   return 3; //base case 3
else {

     Filling-Tiles (n) = Filling-Tiles (n-1) + Filling-Tiles (n-2);
     //This recursively calls for the value of n-1 and n-2 and sends the
        summation as answer;
        }
}
```

Time Complexity: O(n)
Memory Complexity: O(n)

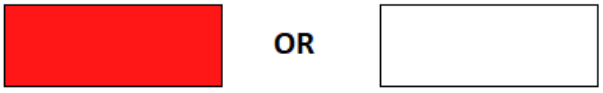
4 Flag/Coloring Problem

Coloring flag is a common dynamic programming problem for beginners. It can be solved using basic math too but for beginners DP (short for dynamic programming) learners, it's better to avoid math in this case.

The problem statement is as follows below:

Given three colors, white, red and blue, how many way can you fill up a sequence, provided the length of the sequence? Also, you can't have any 2 same colors in adjacent cells. As in, the tiles may not be Red and Red, or White and White. It has to be Red and White or White and Red. Also to make the task more difficult, you may put **Blue** only between **Red** and **White** or White and Red. The length of the sequence of to be colored will be provided in integer number n.

For instance, if N is 1, then how many ways can you fill up the one cell?



Possible answer is red or white. So for N equals to 1, the answer is 2.

What if N is 2?

We can either choose Red and White combination or choose White and Red combination.

Figure 7: Red and White combination

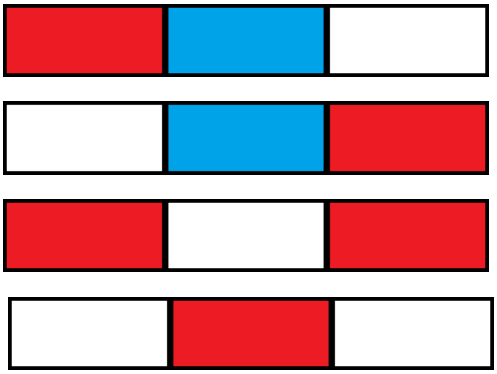
Figure 8: White and Red combination

The primary step of solving a problem using DP is understanding what the problem is about and what it's asking. Then figuring out what could be the base cases, or constant cases. And these cases could be set as default cases. These cases could be used to solve for any test case may be given. For this particular problem, for N = 1, answer will be 2 and for N = 2 answer will be 2. The two best cases which could potentially help further figure out a solution.

Now, **what would be the result if N was 3?**

For sequence of 3 cells, blue must be used between Red and White or White and Red color. So, that's already two of the possible combinations we could have. But getting a combination of Red and White or White and Red is acceptable too where two flags/blocks of the same color are not adjacent. So, Red-Red-white or white-white-red is not acceptable.

So, below are the possible combinations we should come up with for N's value 3.



We figuared the combinations out above manually. Now, to demonstrate why base cases are important, let's further explore what combinations we could form for N's value 4.

The possible combination for a sequence of four cells should be like following:

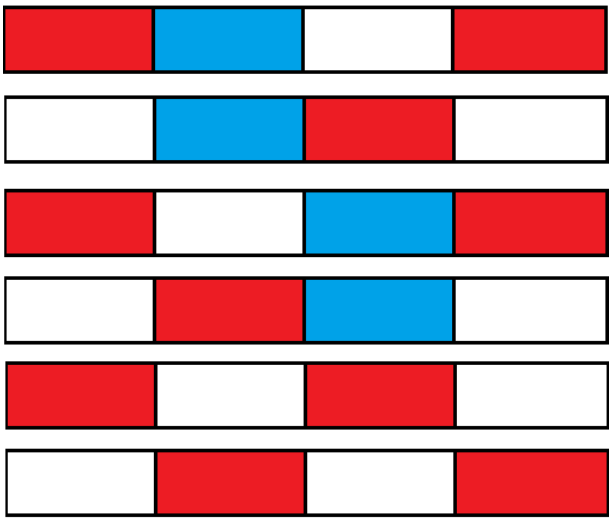


Figure 9: All possible combination for N = 4

Now, if we check the table of the results for different values of n, we see -

N	1	2	3	4
Possible no. of combinations	1	2	4	6

Dynamic programming is very important to programmers because it saves time and memory. Data can be stored to use later. Previously stored data can be used to find new data. For instance, for the two bases above, 1 and 2, answers are 2 and 2 respectively. For N's value 3, answer is 4. Which, as manually demonstrated, was 4 but it can also be achieved by adding previous two answers. The answer found for 4 manually was 6. You could add previous two values, 4 and 2 to get the value for 4.

So, it can be deducted that, for this particular problem we can have the formula below:

$$N(4) = N(3) + N(2)$$

$$N(3) = N(3) + N(2)$$

...

...

Let's call this a function of N. For n^{th} value of N, solve for function N(n) where

$$N(n) = N(n-1) + N(n-2)$$

Dynamic programming is a very important part of Algorithm 1 course. And most programmers prefer this method if given the choice. You can solve these type of problems using recursion or loop. But it is better to use recursion.

Pseudo code is provided below:

Variables :
n : nth sequence for which, find number of combinations
dp[i] : for any integer number i, dp[i] stores the combination which could be used to color i cells.

Code:

```
Flag-Color(n)
{
    if(n is 1 or n is 2)
    return 2;
      // this is the base case
Else {

     Flag-Color (n) = Flag-Color(n-1) + Flag-Color(n-2);
     //This recursively calls for the value of n-1 and n-2 and sends the
        summation as answer;
        }
}
```

Time Complexity: O(n)
Memory Complexity: O(n)