Dynamic Programming

- Dynamic Programming (commonly referred to as DP) is an algorithmic technique for solving a problem by recursively breaking it down into simpler subproblems and using the fact that the optimal solution to the overall problem depends upon the optimal solution to it's individual subproblems.
- The technique was developed by Richard Bellman in the 1950s.
- DP algorithm solves each subproblem just once and then remembers its answer, thereby avoiding re-computation of the answer for similar subproblem every time.
- It is the most powerful design technique for solving optimization related problems.
- It also gives us a life lesson - Make life less complex. There is no such thing as big problem in life. Even if it appears big, it can be solved by breaking into smaller problems and then solving each optimally.

The idea is very simple, If you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. If the given problem can be broken up in to smaller sub-problems and these smaller subproblems are in turn divided in to still-smaller ones, and in this process, if you observe some overlapping subproblems, then its a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem.

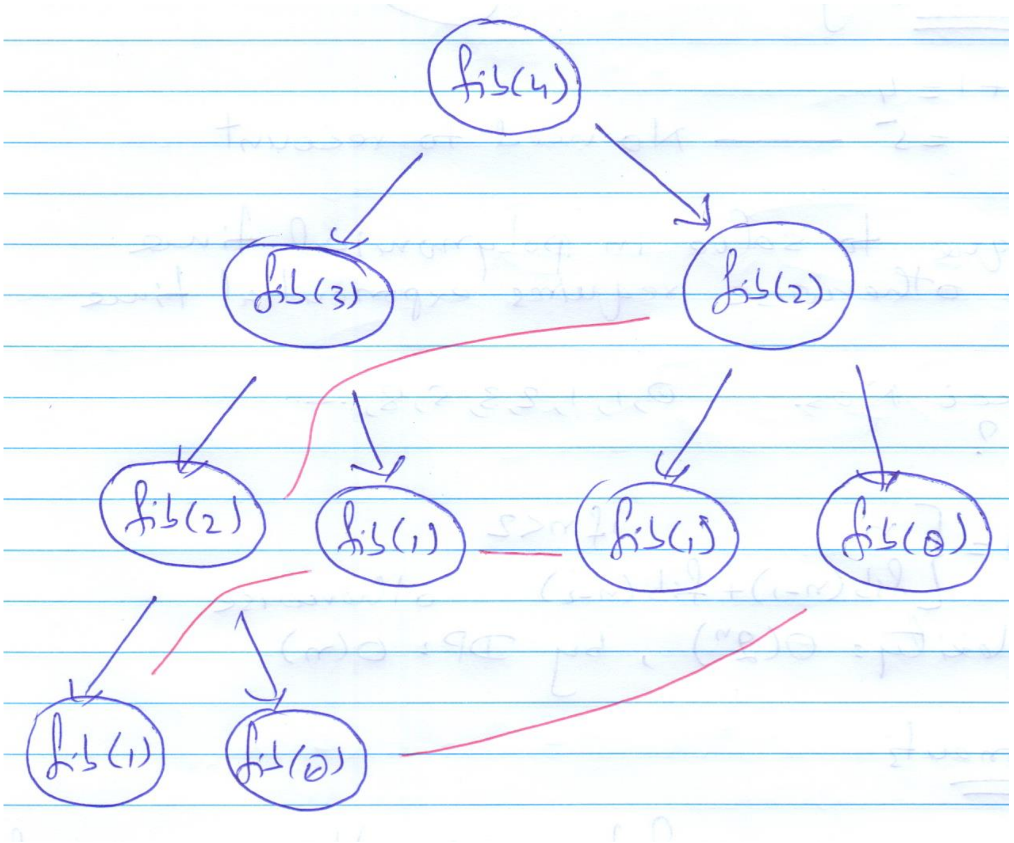
Example:

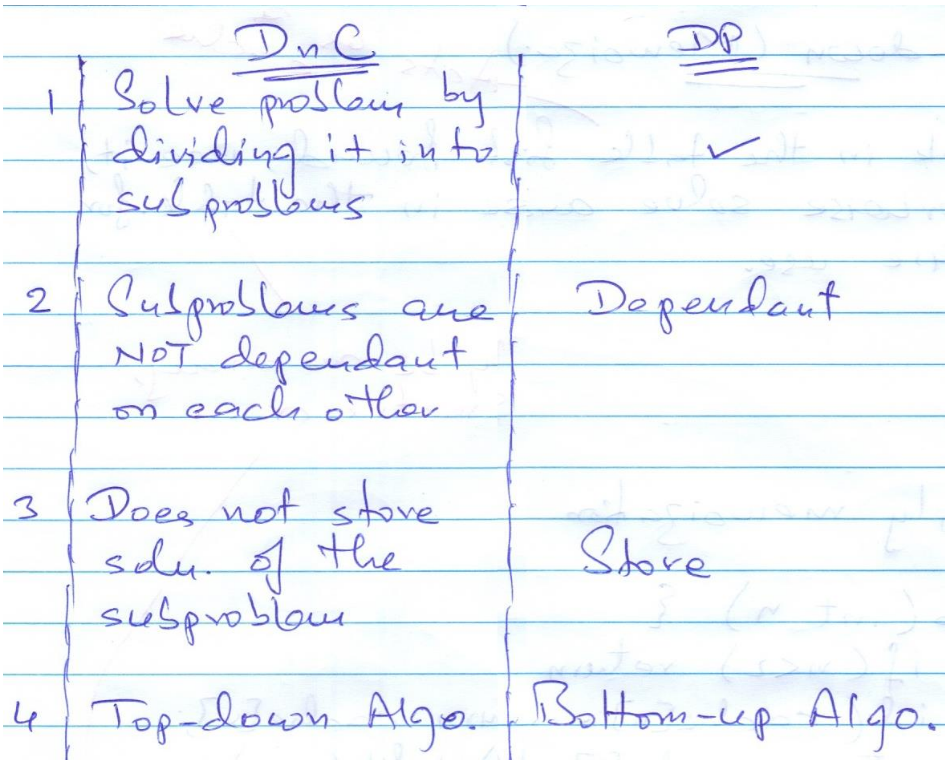
$$fib(n) = \begin{cases} n & if\ n < 2 \\ fib(n-1) + fib(n-2) & otherwise \end{cases}$$

Complexity: *$O(2^n)$* but with DP it is *$O(n)$*

Characteristics:

1. Optimal substructure property: An optimal solution can be obtained by using optimal solutions if its subproblems.
2. Overlapping subproblems: Finding the solution requires the same subproblem again and again.





	DnC	DP
1	Solve problem by dividing it into subproblems	✓
2	Subproblems are NOT dependant on each other	Dependant
3	Does not store soln. of the subproblem	Store
4	Top-down Algo.	Bottom-up Algo.

Methods:

1. Top-Down (Memoization): Look in the table first. If found, use it, otherwise, solve and story in the table for future use.
2. Bottom-Up (Tabulation): Solve subproblem first and Fill-up all entries in the table.

Maybe an array, 2D matrix or …

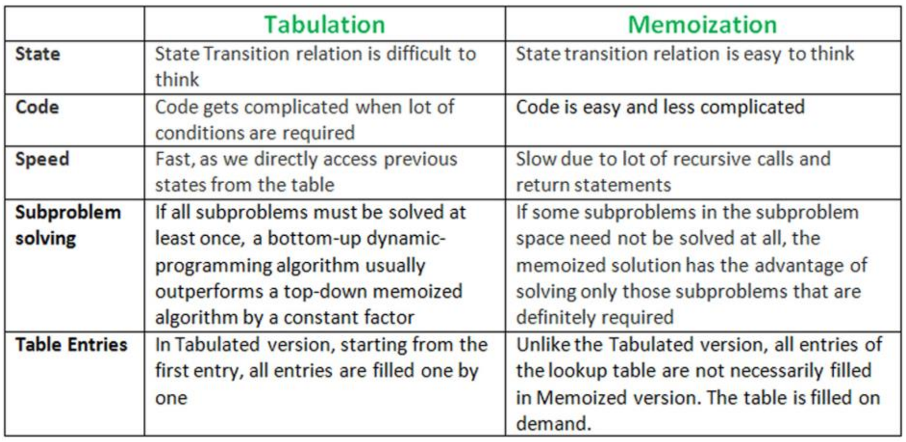
```
fib(int n) {
    if(n<2)             return n;
    if( cache[n]… )     return cache[n];
    return cache[n] = fib(n-1) + fib(n-2);
}
```

```
fib(n) {
    int cache[] = new int[n+1];
    cache[0] = 0;        Base Cases
    cache[1] = 1;
    for( int=2 ; i<=n ; i++ )
        cache[i] = cache[i-1] + cache[i-2];
    return cache[n];
}
```

Tabulation Vs Memoization

If the original problem requires all subproblems to be solved, tabulation usually outperformes memoization by a constant factor. This is because tabulation has no overhead for recursion and can use a preallocated array rather than, say, a hash map.

If only some of the subproblems need to be solved for the original problem to be solved, then memoization is preferrable since the subproblems are solved lazily, i.e. precisely the computations that are needed are carried out.



	Tabulation	Memoization
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.