(2) Dynamic Programming : Some examples of algorithms using dynamic programming are:

(a) Evaluating Fibonacci numbers by Memoizing the Recursion

(b) Evaluating Fibonacci numbers by Iteration.

(c) Bellman-Ford Algorithm for finding Single Source Shortest Paths in a Weighted Directed Graph.

(d) All-Pairs Shortest Paths problem using Matrix Multiplication Algorithm.

(e) Flyod-Warshall Algorithm for finding All Pairs Shortest Paths in a Weighted Directed Graph.

One characteristic of Dynamic Programming is Overlapping Subproblems

In Dynamic Programming, we save our computational effort by avoiding the ~~so~~ solution of subproblems that are overlapping. If a given subproblem appears more than once in the recursion tree, ~~then~~ we solve it only once, ~~The other~~ and save ~~it in memory~~ the solution in memory for future use. When we encounter the same subproblem again, instead of solving the same subproblem again, we just reuse the solution that we have already saved.

There are two approaches for dynamic programming.

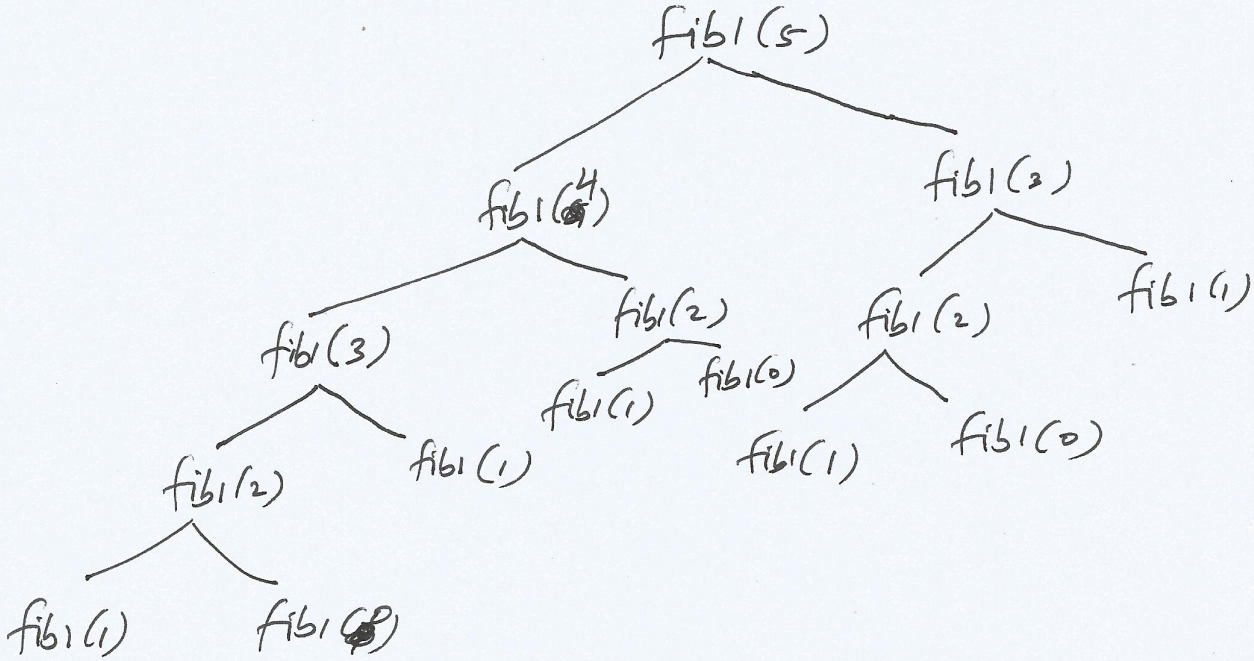
① Top-Down Approach: Memoizing the Recursion.
We solve a subproblem only once and save the result in a global table (this is called memoizing the recursion). Next time when we encounter the same subproblem, we just consult the table for solution.

② Bottom-up Approach: Iteratively solving the subproblems starting from the smallest subproblems, and then proceeding to solve larger subproblems. The solutions to the subproblems are saved in a table. This is the most common method for dynamic programming.

Example of Dynamic Programming: Evaluating the Fibonacci Numbers: We first consider the divide and conquer solution.

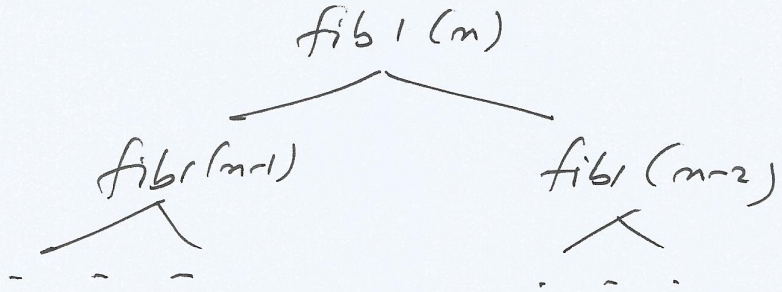
```
long int fib1(long int n)
{  long int a,b;
   if((n==0)||(n==1))
   { return 1; }
   a = fib1(n-1);
   b = fib1(n-2);
   return (a+b);
}
```

Example of Recursion Tree for fib1(5)



We can easily see that the recursion tree has many overlapping sub problems. For example, fib1(0) is called 3 times, fib1(1) is called 5 times, fib1(2) is called 3 times, fib1(3) is called 2 times.

Complexity of fib1(n):



Let $T(n)$ be the time complexity of fib1(n).

We have: $T(n) = T(n-1) + T(n-2) + 1$

$> T(n-1) + T(n-2)$

$T(1) = T(0) = 1$

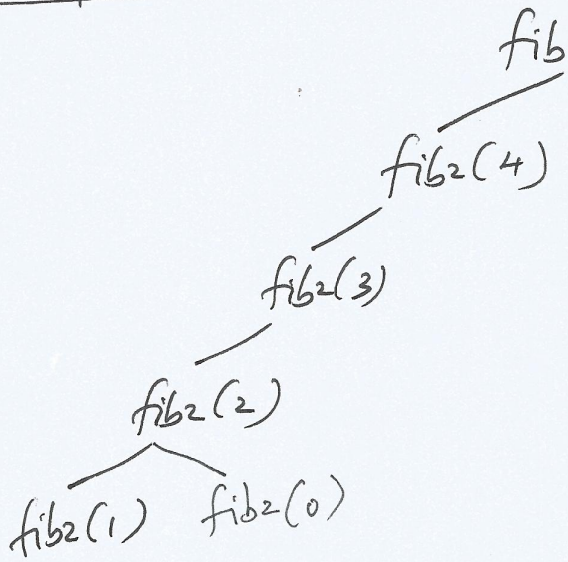
$\Rightarrow T(n) > F_n = \Theta\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$

$\Rightarrow T(n) = \Omega\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$

Top-Down Approach by Memoizing the Recursion (Dynamic Programming):

```
long int fib2(long int n)
{   long int a,b;
    if((n==0)||(n==1))
    {  TABLE[n]=1;
       return 1;
    }
    if(TABLE[n-1]>0)
    {   a = TABLE[n-1]; }
    else
    { a= fib2(n-1);
      TABLE[n-1]=a;
    }
    if (TABLE[n-2]>0)
    { b = TABLE[n-2]; }
    else
    {   b= fib2(n-2);
        TABLE[n-2]= b;
    }
    return(a+b);
}
```

Example of Recursion Tree for fib2(5):



fib2(n) has time complexity O(n).

Bottom-up Approach using Iteration (Dynamic Programming):

```
long int fib3( long int n)
{ long int i;
  TABLE = ( long int *) malloc((n * sizeof (long int)));
  if ((n==0)||(n==1))
  { TABLE (n) = 1;
    return 1;
  }
  TABLE[0]= 1;
  TABLE[1]= 1;
  for( i=2; i<=n; i++)
  { TABLE(i) = TABLE[i-1]+ TABLE[i-2]; }
  return TABLE(n);
}
```

Time complexity of fib3(n) is O(n).

The 0/1 Knapsack Problem

We are given n items $I_1, I_2, \ldots, I_i, \ldots, I_n$
having weights $w_1, w_2, \ldots, w_i, \ldots, w_n$
and profits $p_1, p_2, \ldots, p_i, \ldots, p_n$

A knapsack capacity W is given. Our objective is to choose the items (here we cannot choose fraction of an item) in such a way that the selected items fit into the knapsack (total weight of the selected items is less than or equal to W), and the profit is maximized. Let x_i be a binary variable such that $x_i = 1$ if we have selected I_i, and $x_i = 0$ otherwise. It will contribute $w_i x_i$ weight and $p_i x_i$ profit. Total weight of selected items will be $\sum_{i=1}^{n} w_i x_i$ and total profit of selected items will be $\sum_{i=1}^{n} p_i x_i$. We can write an Integer Linear Programming (ILP) formulation of the 0/1 knapsack problem as follows:

Maximize $\sum_{i=1}^{n} p_i x_i$ subject to

$$\sum_{i=1}^{n} w_i x_i \leq W \quad \text{and}$$

$$x_i \in \{0,1\} \qquad \forall i \in [1..n]$$

Example:

	I_1	I_2	I_3
$w_i =$	3	2	2
$p_i =$	7	4	4

$W = 4$

The optimal solution is $\{I_2, I_3\}$ with profit (8) = opt

Here the greedy algorithm of the fractional knapsack problem is no longer applicable as the most valuable item has ($w_1 = 3$, $p_1 = 7$) < opt.

Let $opt(i, w)$ denote the value of the optimal solution using a subset of the items $\{I_1, I_2, \ldots, I_i\}$ and maximum available weight w. We consider an optimal solution O, and identify two cases depending on whether or not $I_n \in O$:

① If $n \notin O$, then $opt(n, W) = opt(n-1, W)$.

② If $n \in O$, then $opt(n, W) = p_n + opt(n-1, W - w_n)$

For a general i:

$\Rightarrow$ If $W < w_i$ then $opt(i, W) = opt(i-1, W)$. otherwise

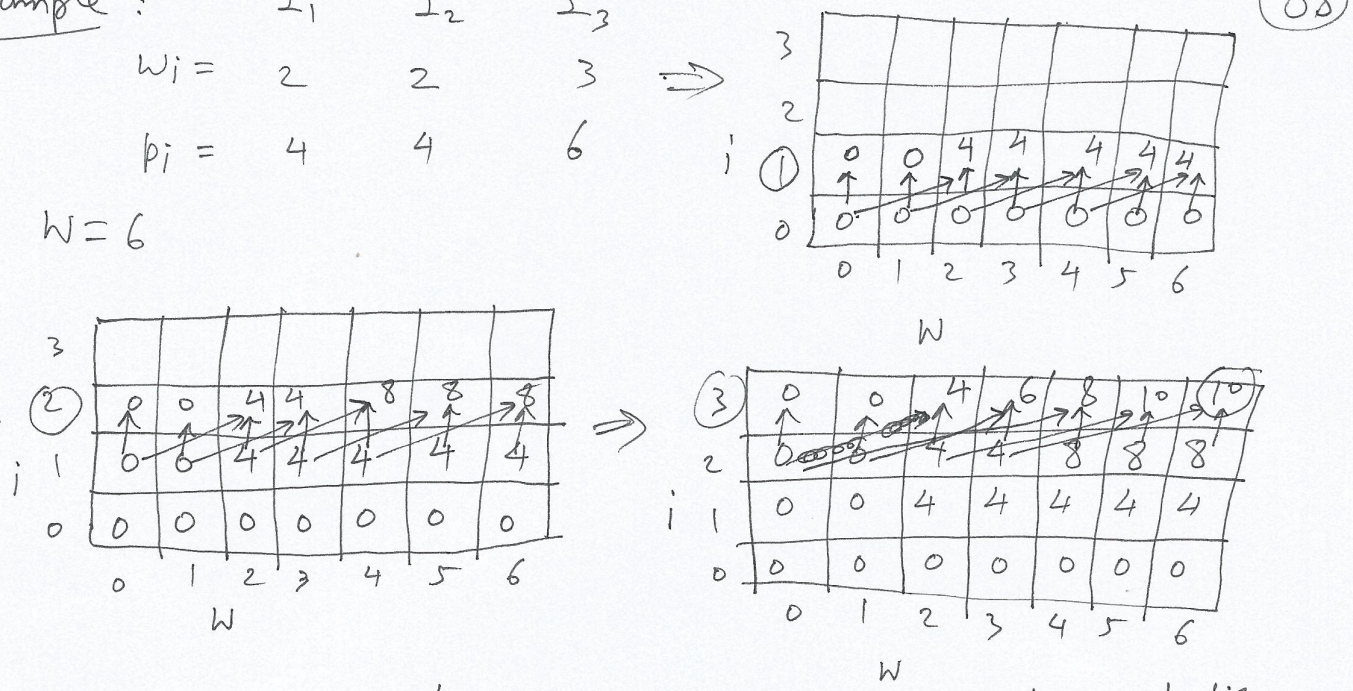
$opt(i, w) = \max(opt(i-1, w), p_i + opt(i-1, W-w_i))$.

```
1   01Knapsack(n, W)
2       Array M[0...n, 0... W]
3       Initialize M[0,w] = 0 for each w=0,1,...,W
4       For i=1,2,... n
5           For w=0, ...,W
6               If (w < wi)
7                   M[i,w] ← M[i-1,w]
8               else
9                   M[i,w] ← max(M[i-1,w], pi+M[i-1,w-wi])
10              Endif
11          EndFor
12      EndFor
13      Return M[n,W]
```

Example:

	I_1	I_2	I_3
$w_i =$	2	2	3
$p_i =$	4	4	6

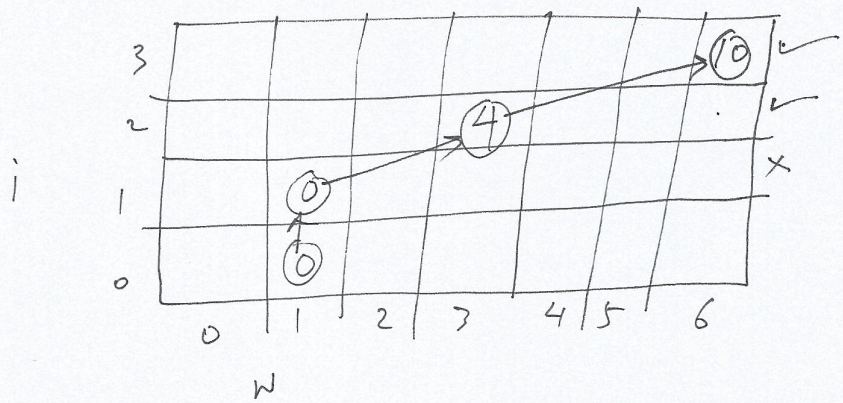
$W = 6$



complexity of 0-1 knapsack: $\Theta(n(W))$ → value not size ⇒ Pseudo-Polynomial Time Algorithm

---

Given a table M of the optimal values of the subproblems the optimal solution (the optimal set of items) can be found in $\Theta(n)$ time. For getting the optimal solution, we just trace back the arrows from $M[n, W]$:



$\{I_2, I_3\}$ is the optimal Solution.

If arrow is vertical: $M[i-1, w] \rightarrow M[i, w]$
then do not select I_i

If arrow is not vertical: $M[i-1, w-w_i] \rightarrow M[i, w]$
then select I_i.