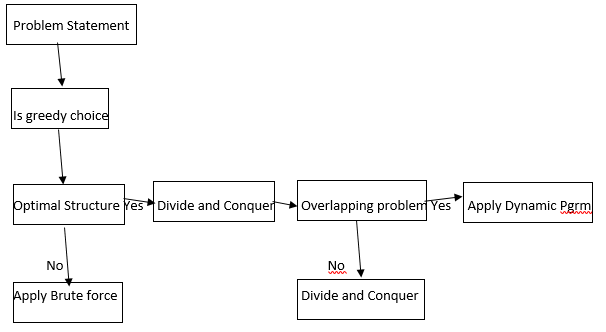
## Dynamic Programming:

The procedure of solving an overlapping problem in real life is called a Dynamic programming.

It is similar to divide and conquer with more optimization.

The problem ill broken down into smaller pieces. All the solution will be stored and re-computed to get final solution.



For Ex: 1+2+3+4+5 =15; For Ex: 1+2+3+4+5+6 = 21; Example 1 answer +6 = 21

**Why learning Dynamic programming:**

Breaking down solution helps in easier solutions instead of getting a solution of a bigger problem.

There will overlapping of multiple smaller problems helping to solve the bigger problem easily. So Dynamic programming is important for any programmer to solve the problem.

**Optimal sub-structure:**

Any similar problem that occurs again and again to get to an optimal solution is called optimal sub-structure. Ex: Fibonacci

**Overlapping sub-problem:**

Any problem which consist of many overlapping sub-problems for finding its solution multiple times.

Ex: finding Fib(6) knowing the answer of Fib(4).

This is the only difference between Divide and Conquer and Dynamic programming.

# Top-Down approach:

This is approach were the problem is broken down into sub-problems

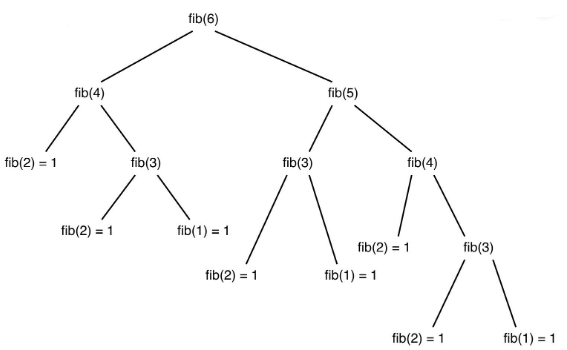
Then sub-problems were solved and remembered and reused to solve the big problem

**Ex: Finbonacci series algorithm:**

Fibonacci(6) = Fib(5) + Fib (4)

Simalarly Fib (5)= 4 and 3.

So in dynamic programming we memorize the values to get the solution.



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 1 |  |  |  | 5+4 |

**Algorithm:**

Fib(memory[], n)

If n less than 1; return 0

If n=2; return 1

**If memory[n] ==0**

Memory[n] = Fib(memory[], n-1)+ Fib(memory[], n-2)

Return memory[n]

Break the problem till top and go till down.

# Bottom-Up approach:

This approach the evaluation happens from the smallest input and increase input argument value

All the computed values will be stored

For solving bigger problem, smaller solution will be re-used

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 |
| 0 | 1 |  |  |  |  |

Algorithm:

Fib(n)

If n less than 1; return 0

If n=2 ; return 1

F[0]=0 F[1]=1

Loop 3 to n

Add values F [i]= F[i-2]+F[i-3]

Return F[n]

Time complexity: O(n) Space Complexity = O(n)

## Top-down vs Bottom-up

Easy of Algorihm: Top down is easy to code

Run time: Bottom up is faster.

Space: Bottom up has to stack required.

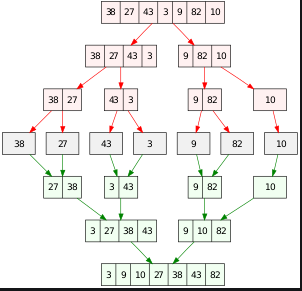
When to use: Top-up is used for quick solution. Bottom up is used for efficient solutions.

# Example for Dynamic programming

Optimal sub-structure

Overlapping sub-problem

**Mergesort**



# House thief:

N house. Thief cannot steal from two adjacent house. Find maximum.

1. 8,9,2,20,4,3,1
2. 7,5,2,18,4

**Divide and Conquer:**

MaxMoney(Networth, currIndex)

If currIndex >= Networth.Length; then return 0

Steal = Networth[currIndex] + MaxMoney(Networth, currIndex+2)

Skip = MaxMoney(Networth, currIndex+1)

Return maximum (Steal, Skip)

**Top-Down:**

MaxMoneyTD(DP, Networth, currIndex)

If currIndex >= Networth.Length; then return 0

**If currIndex = 0**

Steal = Networth[currIndex] + MaxMoney(Networth, currIndex+2)

Skip = MaxMoney(Networth, currIndex+1)

**DP[currIndex] =maximum (Steal, Skip)**

Return (DP[currIndex])

**Bottom-up**

MaxSteal (Wealth)

DP[]= new Wealth.length +2

DP[Wealth.length]= 0

Loop Wealth.length to 0

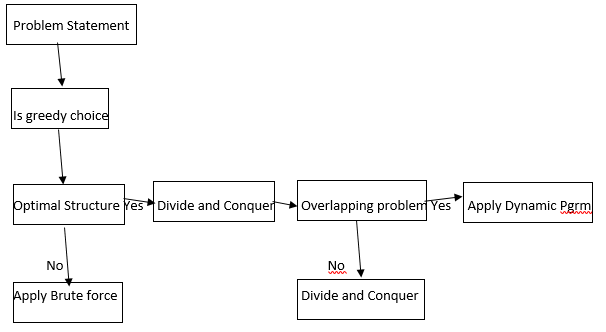
DP[Loop]= max(Wealth+ DP[i+2]+ DP[i+1])

Return DP[0]

## Number Factor:

Problem: Number of ways to express N using different integers.

Ex: 5 Using 1, 3, 4 ; Answer is 6 -> {4,1},{3,1,1},{1,4},{1,3,1},{1,1,3},{1,1,1,1,1}



**WaystoN(N)**

If (n==0)|| (n==1) || (n==2) ; Return 1 //{}, {1}, {1,1}

If (n==3) ; return 2 //{111},{3}

Int sub1= WaystoN(N-1)

Int sub3= WaystoN(N-3)

Int sub4= WaystoN(N-4)

Return sub1+sub3+sub4;

**Algorithm (Top Down)**

WaystoN(N)

**Dp[]= new ar[N+1]**

If (n==0)|| (n==1) || (n==2) ; Return 1 //{}, {1}, {1,1}

If (n==3) ; return 2 //{111},{3}

**If (Dp[n]=0)**

Int sub1= WaystoN(N-1)

Int sub3= WaystoN(N-3)

Int sub4= WaystoN(N-4)

**Dp[n]=sub1+sub3+sub4;**

Return dp[n]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Algorithm (Bottom up)**

For n=4 to n:

Dp[i]= dp[i-1]+ dp[i-3]+ dp[i-4]

Return dp[n]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**Convert One string into another:**

Given S1 and S2; Need S2 to be converted to S1; Minimum count of characters to convert to S1;

Ex: S1= “Cat” ; S2 = “Bat” ; Output : 1 ;

Divide and Conquer+ Top Down :

FindMinOperation(S1, S2, i1, i2)

**If dp[i1][i2]==null**

If (i1 = S1.length); return S2.length-i2;

If (i2 = S2.length); return S1.length-i1;

If(s1.charAt(i1)=S2.charAt(i2)); return(FindMaxOperation(S1, S2, i1+1, i2+1)

C1= 1+ return(FindMaxOperation(S1, S2, i1+1, i2)

C2= 1+ return(FindMaxOperation(S1, S2, i1, i2+1)

C3= 1+ return(FindMaxOperation(S1, S2, i1+1, i2+1)

**dp[i1][i2]=1+ math.min(C1, math.min(C2,C3))**

**Return dp[i1][i2]**

**Bottom Up:**

FindMinOperation(S1, S2)

dp[][]== arr[s1.length+1][s2.length+1]

Loop 0 to S1.length

Dp[i1][0]=i1

Loop 0 to S2.length

Dp[0][i2]=i2

Loop 1 to s1.Length

Loop 1 to s2.Length

If (s1.charAt(i1-1)=S2.charAt(i2-1))

Dp[i1][i2]= dp[i1-1][i2-1]

Else Dp[i1][i2]= 1+ math.min(dp[i1-1][i2], dp[i1-1][i2-1])

Return dp[0][0]