- Height balance tree(AVL) Height $N_h = N_{h-1} + N_{h-2} + 1$, Is same as fibonacci So $height = \log_{\phi} n$
- In a Red black tree , 1) Each node is either red or black 2) Every leaf (Null nodes)is black. 3) The root is black 4) No red node has a red parent or a red child 4) Paths from the root to any leaf all pass through the same number of black nodes (Constant black depth).
- RBT-Insertion (Always make new Insertion as Red)
  If X's uncle is Red → Make X's parent and uncle black and X's grandparent red.
  If X's uncle is Black & and X is the right chil →rotate the edge between X & its parent in opposite of X(yield Case 3)
  If X's uncle is Black and and X is the lef child → rotate the edge between X's parent and X's Grand parent in Opposite of X. Color X's parent black and its old grandparent red.
- Complexity Insertion - 1) Insert.O($\log n$) + Color.O(1) + Fix Violation.($O(\log n) \times (Recolor.O(1) + Rotation.O(1)) = O(2 \log n)$
- ```
  int LCSubStr(char *X, char *Y, int m, int n) // Longest common substring
  {
      int LCSuff[m+1][n+1];
      int result = 0;
      for (int i=0; i<=m; i++)
      {
          for (int j=0; j<=n; j++)
          {
              if (i == 0 || j == 0)
                  LCSuff[i][j] = 0; //For simplicity . Doesn't mean anything

              else if (X[i-1] == Y[j-1])
              {
                  LCSuff[i][j] = LCSuff[i-1][j-1] + 1;
                  result = max(result, LCSuff[i][j]);
              }
              else LCSuff[i][j] = 0;
          }
      }
      return result;
  }
  ```
- ```
  def lcs(X, Y, m, n):  //Longest common subsequence

      if m == 0 or n == 0:
         return 0;
      elif X[m-1] == Y[n-1]:
         return 1 + lcs(X, Y, m-1, n-1);
      else:
         return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
  ```
- ```
  # Returns the maximum value that can be put in a knapsack of capacity W
  def knapSack(W, wt, val, n): #DP 0-1 KnapsackProblem.
      K = [[0 for x in range(W+1)] for x in range(n+1)]

      # Build table K[][] in bottom up manner
      for i in range(n+1):
          for w in range(W+1):
              if i==0 or w==0:
                  K[i][w] = 0
              elif wt[i-1] <= w:
                  K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]],  K[i-1][w])
              else:
                  K[i][w] = K[i-1][w]

      return K[n][W]

  #val = [60, 100, 120] , wt = [10, 20, 30] , W = 50 ,n = len(val)
  #print knapSack(W , wt , val , n)
  ```
- ```
  def minCoins(coins, m, V):  # Min Coin Change ,m is size of coins array
      if (V == 0): # base case
          return 0
      res = sys.maxsize  # Initialize result

      for i in range(0, m): # Try every coin that has smaller value than V
          if (coins[i] <= V):
              sub_res = minCoins(coins, m, V-coins[i])

              # Check for INT_MAX to avoid overflow and see if  result can minimized
              if (sub_res != sys.maxsize and sub_res + 1 < res):
                  res = sub_res + 1
      return res
   #print("Minimum coins required is",minCoins([9, 6, 5, 1], len(coins),11)) # 2
  ```