## Construct binary tree from preorder & inorder traversal

```
def buildTree(pre, in):
    if not pre or not ino:
        return None
    root_val= preorder[0]
    root= TreeNode(root_val)
    root_idx= ~~inorde~~ ino.index(
     root_val)
    root.left= buildTree(pre[1:
      root_idx+1], ino[:root_idx])
    root.right= buildTree(pre[root_idx+1:],
                 ino[root_idx+1:])
    return root
```

## Validate binary search tree (check if it's a BST)

```
def isBST(root):
    def validate(node, low, high):
        if not node:
            return True
        if not (low <node.val< high):
            return False
        return (validate(node.left, low,
                   node.val) and
                 validate(node.right,
                   node.val, high))
    return validate(root, float('-inf'),
                  float('inf'))
```

## kth smallest element in a BST

```
def kthSmallest(root, k):
    stack=[]
    while True:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        k -=1
        if k == 0:
            return root.val
        root = root.right
```

## Lowest Common Ancestor

```
def lca(root, p, q)
    cur = root
    while cur:
        if p.val > cur.val & q.val > cur.val:
            cur = cur.right
        elif p.val<cur.val & q.val<cur.val:
            cur = cur.left
        else:
            return cur
```

## Implement Trie (Prefix tree)

```
class TrieNode:
    def __init__(self):
        self.children= {}
        self.isEndOfWord = False

class Trie:
    def __init__(self):
        self.root= TrieNode()
    def insert(self, word):
        node= self.root
        for char in word:
            if char not in node.children:
                node.children[char] =
                      TrieNode()
            node = node.children[char]
        node.isEndOfWord= True
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node= node.children[char]
        return node.isEndOfWord
    def startsWith(self, prefix):
        node= self.root
        for char in prefix:
            if char not in node.children:
                return False
            node= node.children[char]
        return True
```

## Design Add & Search Words Data Structure

```
class TrieNode:
    def __init__(self):
        self.children= {}
        self.isEndOfWord = False
class WordDictionary:
    def __init__(self):
        self.root= TrieNode()
    def addWord(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char]= TrieNode()
            node = node.children[char]
        node.isEndOfWord= True
    def search(self, word):
        def dfs(node, i):
            if i == len(word):
                return node.isEndOfWord
            char = word[i]
            if char == '.':
                for child in node.children.values():
                    if dfs(child, i+1):
                        return True
                return False
            else:
                if char not in node.children:
                    return False
                return dfs(node.children[ch-
                      ar], i+1)
        return dfs(self.root, 0)
```

## Dynamic Programming (DP)

### coin change (min. coins to make amt)

```
def coinchange(coins, amt):
    dp= [float('inf')] * (amt+1)
    dp[0]=0
    for i in range(1, amt +1):
        for coin in coins:
            if i - coin >= 0:
                dp[i] = min(dp[i], dp[i-coin]
                        +1)
    return dp[amt] if dp[amt] != float('
    inf') else -1
```

### climbing stairs (nbr. of ways using 1 or 2 steps)

```
def climbStairs(n):
    dp =[0] * (n+1)
    dp[0], dp[1] = 1, 1
    for i in range(2, n+1):
        dp[i] = dp[i-2] + dp[i-1]
    return dp[n]
```

### Length of Longest increasing sub-sequence

```
def lengthOfLIS(nums):
    cnt= len(nums)
    dp= [1]* cnt
    dp[0] = 1
    for i in range(cnt):
        for j in range(i):
            if nums[j] < nums[i]:
                dp[i] =max(dp[i], dp[j] +1)
    return max(dp)
```

### Longest common subsequence of 2 strings

```
def lcs(a, b):
    cnta = len(a)
    cntb = len(b)
    dp = [[0] * (cntb+1) for i in range(cnta+1)]
    for i in range(1, cnta+1):
        for j in range(1, cntb+1):
            if a[i-1] == b[j-1]:
                dp[i][j] = max(dp[i][j], 1 +
                              dp[i-1][j-1])
            else:
                dp[i][j] = max(dp[i-1][j], dp[i]
                              [j-1])
    return max(max(el) for el in dp))
```

### Word break problem

```
def wordBreak(s, wordList):
    wordSet = set(wordList):
    dp = [False] * (len(s)+1)
    dp[0] = True
    for i in range(1, len(s)+1):
        for j in range(i):
            if dp[j] and s[j:i] in wordSet:
                dp[i] = True
                break
    return dp[-1]
```

### Combination sum

```
def combinationSum(candidates, target):
    result =[]
    def backtrack(remaining, combination, start)
        if remaining == 0:
            result.append(list(combination))
            return
        for i in range(start, len(candidates)):
            if candidates[i] > remaining:
                continue
            combination.append(candidates[i])
```