In [ ]:
```
ITERATIVE ALGORITHM
Algorithm Fibonacci(n)
//Compute the nth Fibonacci Number
{
    if (n<=1) then
        write(n);
    else
    {
        fnm2 = 0; fnm1 = 1;
        for i=2 to n do
        {
            fn = fnm1 + fnm2;
            fnm2 = fnm1; fnm1 = fn;
        }
        write(fn);
    }
}
```

In [4]:
```
#Iterative Program
nterms = int(input("Enter number of terms "))

n1, n2 = 0, 1

if nterms <= 1:
    print(n1)
else:
    print(n1)
    print(n2)
    for i in range(nterms-2):
        nth = n1 + n2
        n1 = n2
        n2 = nth
        print(nth)
```

```
Enter number of terms 5
0
1
1
2
3
```

In [ ]:
```
Recursive Algorithm
Algorithm rFibonacci(n)
{
    if (n <= 1)
        return n;
    else
        return rFibonacci(n - 1) + rFibonacci(n - 2);
}
```

In [5]:
```
#Recursive Program
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

n = int(input("Enter Number of Terms : "))
```

```
for i in range(n):
    print(fibonacci(i))
```

```
Enter Number of Terms : 6
0
1
1
2
3
5
```

In [ ]:
```
GREEDY-HUFFMAN-CODE(C)
  min_queue.build(C)

  while min_queue.length > 1
    z = new node
    z.left = min_queue.extract()
    z.right = min_queue.extract()
    z.freq = z.left.freq + z.right.freq
    min_queue.insert(z)

  return min_queue.extract()
```

In [5]:
```
import heapq

class node:
        def __init__(self, freq, symbol, left=None, right=None):
                # frequency of symbol
                self.freq = freq
                # symbol name (character)
                self.symbol = symbol
                # node left of current node
                self.left = left
                # node right of current node
                self.right = right
                # tree direction (0/1)
                self.huff = ''

        def __lt__(self, nxt):
                return self.freq < nxt.freq

def printNodes(node, val=''):
        newVal = val + str(node.huff)
        # if node is not an edge node
        # then traverse inside it
        if(node.left):
                printNodes(node.left, newVal)
        if(node.right):
                printNodes(node.right, newVal)
                # if node is edge node then
                # display its huffman code
        if(not node.left and not node.right):
                print(f"{node.symbol} -> {newVal}")

# characters for huffman tree
chars = ['a', 'b', 'c', 'd', 'e', 'f']

# frequency of characters
freq = [ 4, 7, 12, 14, 43, 54]

# list containing unused nodes
nodes = []

# converting characters and frequencies
# into huffman tree nodes
for x in range(len(chars)):
        heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:
```

```
        # sort all the nodes in ascending order
        # based on their frequency
        left = heapq.heappop(nodes)
        right = heapq.heappop(nodes)

        # assign directional value to these nodes
        left.huff = 0
        right.huff = 1

        # combine the 2 smallest nodes to create
        # new node as their parent
        newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)

        heapq.heappush(nodes, newNode)

# Huffman Tree is ready!
printNodes(nodes[0])
```

```
f -> 0
d -> 100
a -> 10100
b -> 10101
c -> 1011
e -> 11
```

In [ ]:

In [ ]:
```
Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)
for i = 1 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    if weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```

In [1]:
```
def fractional_knapsack(value, weight, capacity):
    # index = [0, 1, 2, ..., n - 1] for n items
    index = list(range(len(value)))
    # contains ratios of values to weight
    ratio = [v/w for v, w in zip(value, weight)]
    # index is sorted according to value-to-weight ratio in decreasing order
    index.sort(key=lambda i: ratio[i], reverse=True)

    max_value = 0
    fractions = [0]*len(value)
    for i in index:
        if weight[i] <= capacity:
            fractions[i] = 1
            max_value += value[i]
            capacity -= weight[i]
        else:
            fractions[i] = capacity/weight[i]
            max_value += value[i]*capacity/weight[i]
            break

    return max_value, fractions


n = int(input('Enter number of items: '))
value = input('Enter the values of the {} item(s) in order: '
              .format(n)).split()
value = [int(v) for v in value]
weight = input('Enter the positive weights of the {} item(s) in order: '
               .format(n)).split()
weight = [int(w) for w in weight]
capacity = int(input('Enter maximum weight: '))

max_value, fractions = fractional_knapsack(value, weight, capacity)
print('The maximum value of items that can be carried:', max_value)
print('The fractions in which the items should be taken:', fractions)
```

```
Enter number of items: 3
Enter the values of the 3 item(s) in order: 24 15 25
Enter the positive weights of the 3 item(s) in order: 15 10 18
Enter maximum weight: 20
The maximum value of items that can be carried: 31.5
The fractions in which the items should be taken: [1, 0.5, 0]
```

In [ ]:

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
    c[0, w] = 0
for i = 1 to n do
    c[i, 0] = 0
    for w = 1 to W do
        if wi ≤ w then
            if vi + c[i-1, w-wi] then
                c[i, w] = vi + c[i-1, w-wi]
            else c[i, w] = c[i-1, w]
        else
            c[i, w] = c[i-1, w]
```

In [4]:

```
def knapSack(W, wt, val, n):
    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 = [70, 90, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))
```

```
210
```

In [ ]:
```
N-Queen using Backtracking Algorithm
IS-ATTACK(i, j, board, N)
  // checking in the column j
  for k in 1 to i-1
    if board[k][j]==1
      return TRUE

  // checking upper right diagonal
  k = i-1
  l = j+1
  while k>=1 and l<=N
    if board[k][l] == 1
      return TRUE
    k=k+1
    l=l+1

  // checking upper left diagonal
  k = i-1
  l = j-1
  while k>=1 and l>=1
    if board[k][l] == 1
      return TRUE
    k=k-1
    l=l-1

  return FALSE

N-QUEEN(row, n, N, board)
  if n==0
    return TRUE

  for j in 1 to N
    if !IS-ATTACK(row, j, board, N)
      board[row][j] = 1

      if N-QUEEN(row+1, n-1, N, board)
        return TRUE

      board[row][j] = 0 //backtracking, changing current decision
  return FALSE
```

In [2]:
```python
global N
N = 4

def printSolution(board):
        for i in range(N):
                for j in range(N):
                        print(board[i][j], end=' ')
                print()

def isSafe(board, row, col):

        # Check this row on left side
        for i in range(col):
                if board[row][i] == 1:
                        return False

        # Check upper diagonal on left side
```

```
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
                if board[i][j] == 1:
                        return False

        # Check lower diagonal on left side
        for i, j in zip(range(row, N, 1), range(col, -1, -1)):
                if board[i][j] == 1:
                        return False

        return True

def solveNQUtil(board, col):

        if col >= N:
                return True

        for i in range(N):

                if isSafe(board, i, col):

                        board[i][col] = 1


                        if solveNQUtil(board, col + 1) == True:
                                return True


                        board[i][col] = 0


        return False


def solveNQ():
        board = [ [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0],
                        [0, 0, 0, 0]
                        ]

        if solveNQUtil(board, 0) == False:
                print ("Solution does not exist")
                return False

        printSolution(board)
        return True

# driver program to test above function
solveNQ()
```

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Out[2]: True