In [1]:

```
def fib(n):
    if n <= 1:
        return n

    else:
        return (fib(n - 1) + fib(n - 2))

n = int(input("Enter length of fibonacci series: "))

print("Fibonacci series(recursive) of length", n, "is: ")
for i in range (0, n):
    print(fib(i), end = "\t")

print("\n\nFibonacci series(non-recursive) of length", n, "is:\n0\t1", end = "\t")
n1 = 0
n2 = 1
for i in range (2, n):
    if n <= 2:
        break

    else:
        n_next = n1 + n2
        print(n_next, end = "\t")

    n1 = n2
    n2 = n_next
```

```
Enter length of fibonacci series: 10
Fibonacci series(recursive) of length 10 is:
0       1       1       2       3       5       8       13      21
34

Fibonacci series(non-recursive) of length 10 is:
0       1       1       2       3       5       8       13      21
34
```

In [2]:

```
class Nodes:
    def __init__(self, freq, symbol, left = None, right = None):
        self.freq = freq
        self.symbol = symbol
        self.left = left
        self.right = right

        self.huff = ""

def print_nodes(node, val = ""):

    new_val = val + str(node.huff)

    if node.left:
        print_nodes(node.left, new_val)
    if node.right:
        print_nodes(node.right, new_val)

    if not node.left and not node.right:
        print(f"{node.symbol} -> {new_val}")

chars = ['a', 'b', 'c', 'd', 'e', 'f']
freq = [5, 9, 12, 13, 16, 45]

nodes = [Nodes(freq[x], chars[x]) for x in range (len(chars))]

while (len(nodes) > 1):

    nodes = sorted(nodes, key = lambda x: x.freq)

    left = nodes[0]
    right = nodes[1]

    left.huff = 0
    right.huff = 1

    new_nodes = Nodes(left.freq + right.freq, left.symbol + right.symbol, left, right)

    nodes.remove(left)
    nodes.remove(right)
    nodes.append(new_nodes)

print("Characters: ", f'[{", ".join(chars)}]')
print("Frequency: ", freq, "\n\nHuffman Encoding: ")
print_nodes(nodes[0])
```

```
Characters:  [a, b, c, d, e, f]
Frequency:  [5, 9, 12, 13, 16, 45]

Huffman Encoding:
f -> 0
c -> 100
d -> 101
a -> 1100
b -> 1101
e -> 111
```

In [3]:

```
class item_value:
    def __init__(self, wt, val, ind):
        self.val = val
        self.wt = wt
        self.ind = ind

        self.cost = val // wt

    def __lt__(self, other):
        return (self.cost < other.cost)

def fractional_knapsack(wt, val, capacity):

    i_val = [item_value(wt[i], val[i], i) for i in range (len(wt))]
    i_val.sort(reverse = True)

    total_value = 0

    for i in i_val:
        cur_wt = i.wt
        cur_val = i.val

        if ((capacity - cur_wt) >= 0):
            capacity -= cur_wt
            total_value += cur_val

        else:
            fraction = capacity / cur_wt
            capacity = int(capacity - (fraction * cur_wt))
            total_value += cur_val * fraction

    return total_value

if __name__ == "__main__":
    wt = [10, 40, 20, 30]
    val = [60, 40, 100, 120]
    capacity = 50

    max_value = fractional_knapsack(wt, val, capacity)
    print("Maximum value of knapsack is: ", max_value)
```

```
Maximum value of knapsack is:  240.0
```

In [4]:

```
def knapSack(W, wt, val, n):

    if n == 0 or W == 0 :
        return 0

    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)

    else:
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1), knapSack(W, wt, val, r

val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print (knapSack(W, wt, val, n))
```

```
220
```

In [5]:

```
class NQBacktracking:
    def __init__(self, x, y):
        self.ld = [0] * 30
        self.rd = [0] * 30
        self.cl = [0] * 30

        self.x = x
        self.y = y

    def printSolution(self, board):
        print("N Queen Backtracking Solution:\nGiven initial position os 1st queen at r
              "and column:", self.y, "\n")

        for line in board:
            print(" ".join(map(str, line)))

    def solveNQUtil(self, board, col):
        if col >= N:
            return True
        if col == self.y:
            return self.solveNQUtil(board, col + 1)

        for i in range (N):
            if i == self.x:
                continue
            if (self.ld[i - col + N - 1] != 1 and self.rd[i + col] != 1) and self.cl[i]
                board[i][col] = 1
                self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 1

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

                board[i][col] = 0
                self.ld[i - col + N - 1] = self.rd[i + col] = self.cl[i] = 0

        return False

    def solveNQ(self):
        board = [[0 for _ in range (N)] for _ in range (N)]
        board[self.x][self.y] = 1

        self.ld[self.x - self.y + N - 1] = self.rd[self.x + self.y] = self.cl[self.x] =

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

        self.printSolution(board)
        return True

if __name__ == "__main__":
    N = 8
    x, y = 3, 2

    NQBt = NQBacktracking(x, y)
    NQBt.solveNQ()
```

```
N Queen Backtracking Solution:
Given initial position os 1st queen at row: 3 and column: 2

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