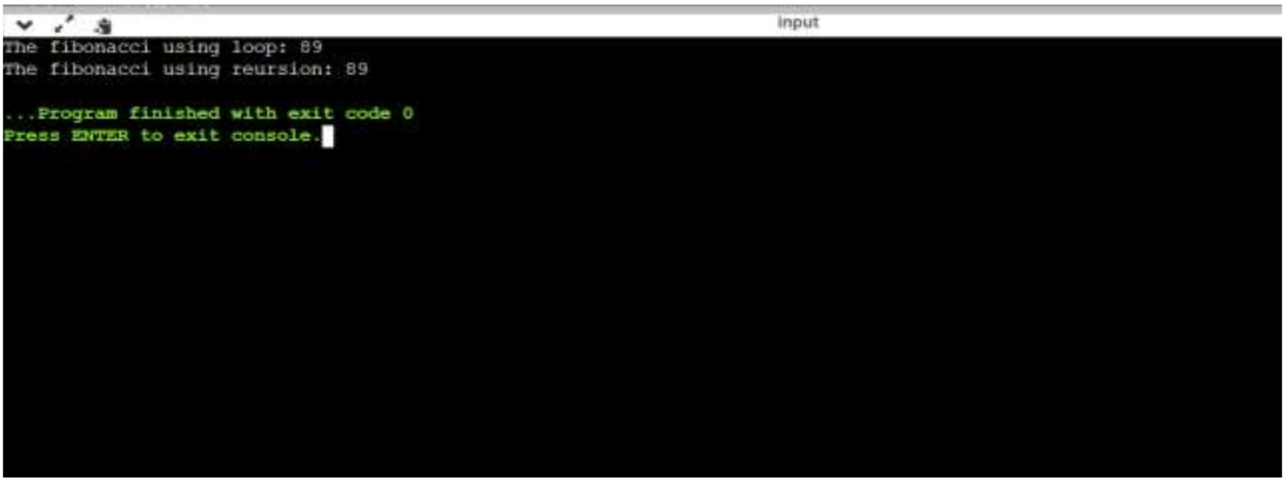
# Experiment No:1

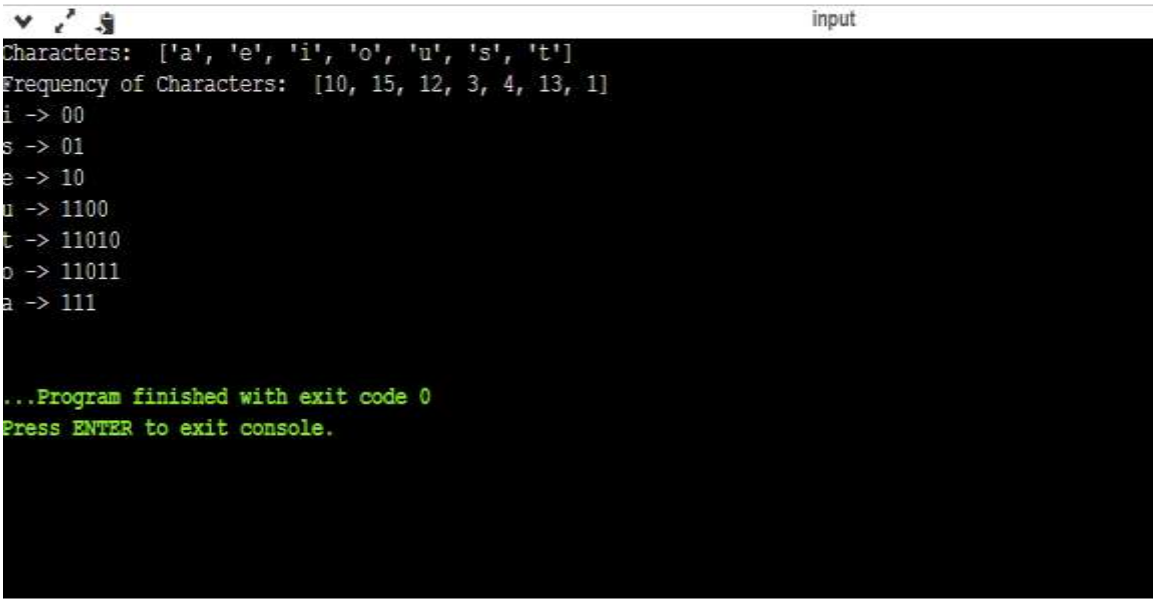
```
#include <iostream>

using namespace std;

int fibonacci_recursion(int n){
        if(n<=2)
                return 1;
        else
                return fibonacci_recursion(n-1) + fibonacci_recursion(n-2);
}

int fibonacci_using_loop(int n){
        if(n<=2)
                return 1;
        int i, last, nextToLast, result;
        last = 1;
        nextToLast = 1;
        result = 1;
        for(i=3; i<=n; i++){
                result = last + nextToLast;
                nextToLast = last;
                last = result;
        }
        return result;
}

int main()
{
  cout<<"The fibonacci using loop: "<<fibonacci_using_loop(11);
  cout<<"\nThe fibonacci using reursion: "<<fibonacci_recursion(11);
  return 0;
}
```



```
The fibonacci using loop: 89
The fibonacci using reursion: 89

...Program finished with exit code 0
Press ENTER to exit console.
```

# Experiment No:2

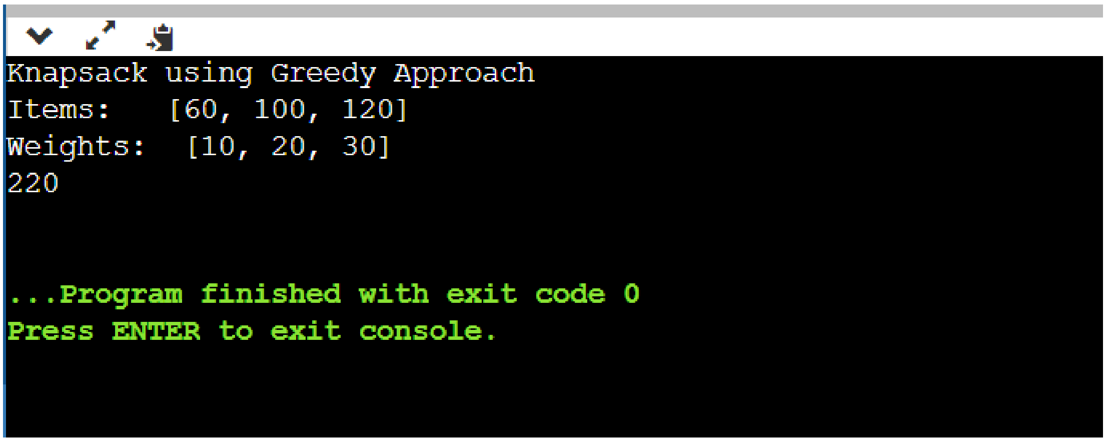
```
import heapq
class node:
	def __init__(self, freq, symbol, left=None, right=None):
		# frequency of symbol
		self.freq = freq

		# 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.left):
		printNodes(node.left, newVal)
	if(node.right):
		printNodes(node.right, newVal)
	if(not node.left and not node.right):
		print(f"{node.symbol} -> {newVal}")

chars = ['a', 'e','i','o','u','s','t']
freq = [10, 15, 12, 3, 4, 13, 1]
nodes = []

for x in range(len(chars)):
	heapq.heappush(nodes, node(freq[x], chars[x]))

while len(nodes) > 1:
	left = heapq.heappop(nodes)
	right = heapq.heappop(nodes)
	left.huff = 0
	right.huff = 1
	newNode = node(left.freq+right.freq, left.symbol+right.symbol, left, right)
	heapq.heappush(nodes, newNode)
printNodes(nodes[0])
```



```
Characters:  ['a', 'e', 'i', 'o', 'u', 's', 't']
Frequency of Characters:  [10, 15, 12, 3, 4, 13, 1]
i -> 00
s -> 01
e -> 10
u -> 1100
t -> 11010
o -> 11011
a -> 111


...Program finished with exit code 0
Press ENTER to exit console.
```

# Experiment No:3

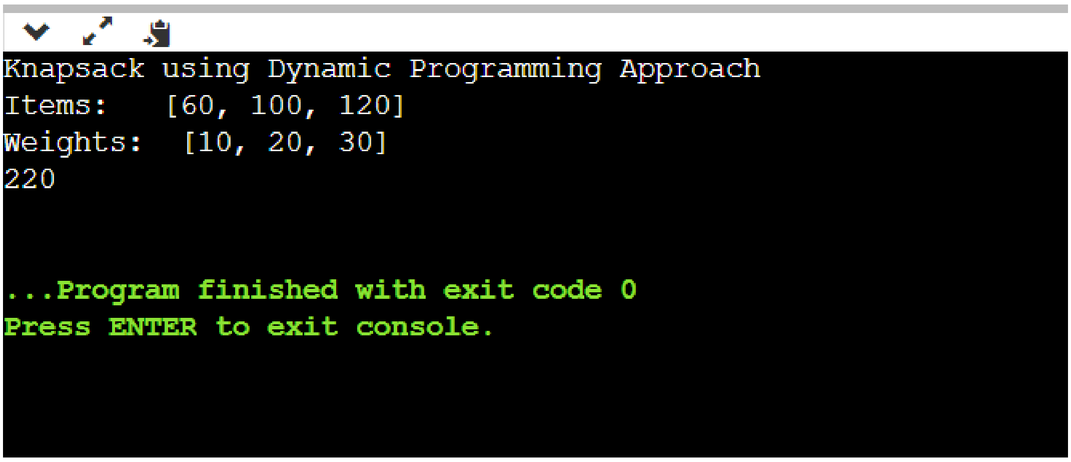
```
def knapSack(W,  wt, val, n):

    # Base Case
    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, n-1))

#Driver Code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print("Knapsack using Greedy Approach")
print("Items: \t",val)
print("Weights: ",wt)
print(knapSack(W, wt, val, n))
```



```
Knapsack using Greedy Approach
Items:    [60, 100, 120]
Weights:  [10, 20, 30]
220


...Program finished with exit code 0
Press ENTER to exit console.
```

# Experiment No:4

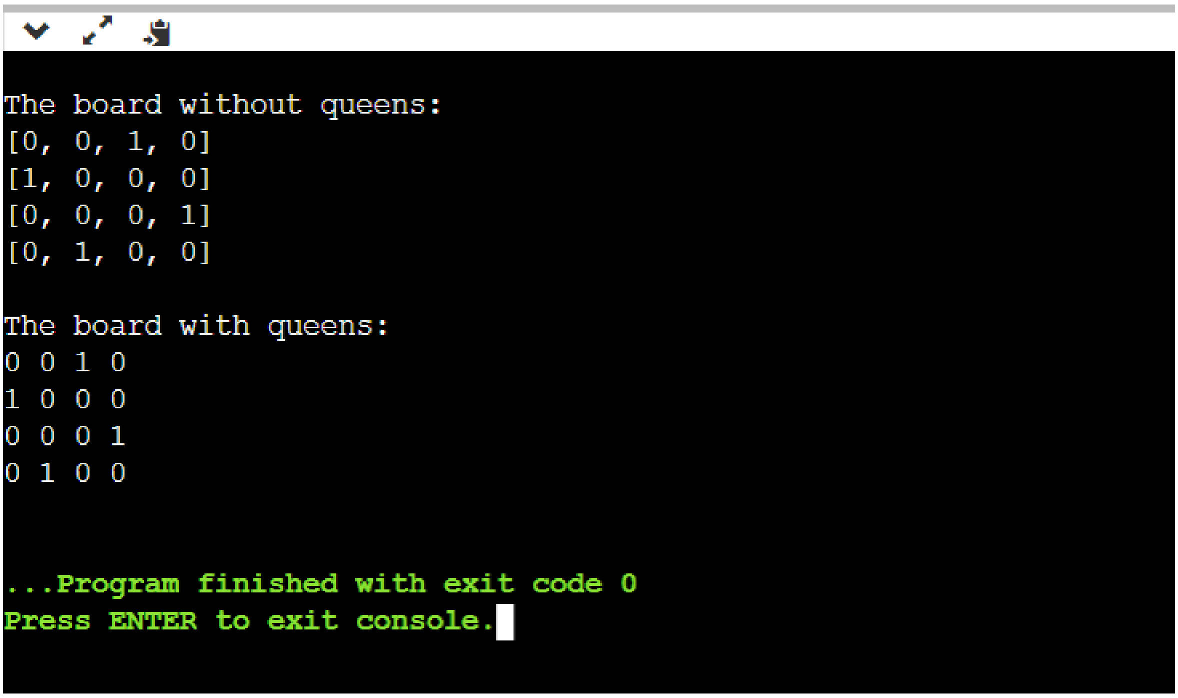
```
def knapSack(W, wt, val, n):
  K = [[0 for x in range(W+1)] for y in range(2)]
  for i in range(n + 1):
     for w in range(W + 1):
        if (i == 0 or w == 0):
           K[i % 2][w] = 0
        elif (wt[i - 1] <= w):
           K[i % 2][w] = max(
              val[i - 1]
              + K[(i - 1) % 2][w - wt[i - 1]],
              K[(i - 1) % 2][w])

        else:
           K[i % 2][w] = K[(i - 1) % 2][w]

  return K[n % 2][W]

# Driver Code
if __name__ == "__main__":

  val = [60, 100, 120]
  wt = [10, 20, 30]
  W = 50
  n = len(val)
  print("Knapsack using Dynamic Programming Approach")
  print("Items: \t",val)
  print("Weights: ",wt)
  print(knapSack(W, wt, val, n))
```



```
Knapsack using Dynamic Programming Approach
Items:    [60, 100, 120]
Weights:  [10, 20, 30]
220


...Program finished with exit code 0
Press ENTER to exit console.
```

# Experiment No:5

```
# Python program to solve N Queen Problem using backtracking
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):
			# Place this queen in board[i][col]
			board[i][col] = 1

			# recur to place rest of the queens
			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

    print("The board without queens:")
    for i in board:
      print(list(i))
    print("\nThe board with queens:")
    printSolution(board)
    return True

print()
solveNQ()
```



```
The board without queens:
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]

The board with queens:
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0


...Program finished with exit code 0
Press ENTER to exit console.
```