# TECHNO INDIA UNIVERSITY

Name: Rohan Ghosh

Batch: BCS2B

ID: 181001001122

Subject: Design Analysis of Algorithms

# Problem 1:

Implement Huffman's algorithm using Python. You need to implement the following:

1. Read a string input by the user
2. Find the frequency of each symbol present in the input.
3. Construct the Huffman tree.
4. Output the codewords for each symbol present in the string (encode the input string).

A typical execution of your code should look like this:

Enter the string to be encoded: good morning everybody
character Weight Huffman Code
o 4 01
n 2 000
r 2 001
y 2 100
" 2 1010
D 2 1100
e 2 1101
g 2 1110
b 1 10110
i 1 10111
m 1 11110
v 1 11111
---------------------------------

Construction of the Huffman tree may require use of a special data structure called Heap. A *heap* is a tree-like data structure where the child nodes have a sort-order relationship (either ascending or descending order) with the parents. A max-heap ensures that, in any sub tree, the parent is larger than or equal to both of its children. On the other hand, a min-heap requires the parent be less than or equal to any of its children, in any sub tree. Python's has a built-in module, heap q module which implements all necessary function relevant to a min-heap. A comprehensive description of Python's heap q module may be obtained in https://pymotw.com/2/heapq/. You are encouraged to study that module in detail, before solving this problem.

## ANSWER:

```
import heapq
from collections import defaultdict
def encode(frequency):
        heap = [[weight, [symbol, '' ]] for symbol, weight in
        frequency.items()]
        heapq.heapify(heap)
        while len(heap) > 1 :
                low = heapq.heappop(heap)
                high = heapq.heappop(heap)
                for value in low[ 1 :]:
                        value[ 1 ] = '0' + value[ 1 ]
                for value in high[ 1 :]:
                        value[ 1 ] = '1' +value[ 1 ]
                heapq.heappush(heap, [low[ 0 ] + high[ 0 ]] + low[ 1 :] +
high[ 1 :])
        return sorted(heapq.heappop(heap)[ 1 :], key= lambda
p:(len(p[ -1 ]), p))
```

```
string=input( "Enter the string to be encoded:" )

frequency = defaultdict(int)

for character in string:

    frequency[character] += 1

huff = encode(frequency)

print( "character" .ljust( 10 ) + "Weight" .ljust( 10 ) +
"HuffmanCode" )

for i in huff:

    print(i[ 0 ].ljust( 10 ) + str(frequency[i[ 0 ]]).ljust( 10 ) + i[ 1 ])
```

## <u>Output</u>:

```
C:\Users\PRATIK\Desktop>python check.py
Enter the string to be encoded:coding
character Weight    HuffmanCode
n         1         00
o         1         01
c         1         100
d         1         101
g         1         110
i         1         111

C:\Users\PRATIK\Desktop>
```

## PROBLEM 2:

Write Python code to implement Strassen's matrix multiplication algorithm. Generate random 32x32, 64x64, 128x128, and 256x256 matrices (in whatever way is convenient, use smallish integers). Run the implemented algorithm on the generated matrices and record the execution times.

| Typical run should look like this: **n** | **Run times in milliseconds** |
|---|---|
| 32 | 149 |
| 64 | 342 |
| 128 | 2444 |
| 256 | 20071 |

## ANSWER:

```
import numpy as np

import time

def split (matrix) :

        row, col = matrix.shape

        row2, col2 = row// 2 , col// 2

        return matrix[:row2, :col2], matrix[:row2, col2:],matrix[row2:,
 :col2], matrix[row2:, col2:]

def strassen(x, y):

        if len(x) == 1 :
```

```
        return x * y
    a, b, c, d = split(x)
    e, f, g, h = split(y)

    p1 = strassen(a, f - h)
    p2 = strassen(a + b, h)
    p3 = strassen(c + d, e)
    p4 = strassen(d, g - e)
    p5 = strassen(a + d, e + h)
    p6 = strassen(b - d, g + h)
    p7 = strassen(a - c, e + f)

    c11 = p5 + p4 - p2 + p6
    c12 = p1 + p2
    c21 = p3 + p4
    c22 = p1 + p5 - p3 - p7
    c = np.vstack((np.hstack((c11, c12)), np.hstack((c21,c22))))
    return c

start_time = time.time()
```

```
mat1 = np.random.randint( 4 , size= ( 32 , 32 ))
mat2 = np.random.randint( 4 , size= ( 32 , 32 ))
k = strassen(mat1, mat2)
end_time = time.time()
diff1 = end_time - start_time

start_time2 = time.time()
mat2 = np.random.randint( 4 , size= ( 64 , 64 ))
mat3 = np.random.randint( 4 , size= ( 64 , 64 ))
k2 = strassen(mat2, mat3)
end_time2 = time.time()
diff2 = end_time2 - start_time2

start_time3 = time.time()
mat4 = np.random.randint( 4 , size= ( 128 , 128 ))
mat5 = np.random.randint( 4 , size= ( 128 , 128 ))
k3 = strassen(mat4, mat5)
end_time3 = time.time()
diff3 = end_time3 - start_time3
start_time4 = time.time()
```

```
mat6 = np.random.randint( 4 , size= ( 256 , 256 ))

mat7 = np.random.randint( 4 , size= ( 256 , 256 ))

k4 = strassen(mat6, mat7)

end_time4 = time.time()

diff4 = end_time4 - start_time4


print( 'for dimension = 32, runtime = ' , diff1* 1000 )

print( 'for dimension = 64, runtime = ' , diff2* 1000 )

print( 'for dimension = 128, runtime = ' , diff3* 1000 )

print( 'for dimension = 256, runtime = ' , diff4* 1000 )
```

**Output**:

```
start_time2 = time.time()
mat2 = np.random.randint( 4 , size= ( 64 , 64 ))
mat3 = np.random.randint( 4 , size= ( 64 , 64 ))
k2 = strassen(mat2, mat3)
end_time2 = time.time()
diff2 = end_time2 - start_time2

start_time3 = time.time()
mat4 = np.random.randint( 4 , size= ( 128 , 128 ))
mat5 = np.random.randint( 4 , size= ( 128 , 128 ))
k3 = strassen(mat4, mat5)
end_time3 = time.time()
diff3 = end_time3 - start_time3

start_time4 = time.time()
mat6 = np.random.randint( 4 , size= ( 256 , 256 ))
mat7 = np.random.randint( 4 , size= ( 256 , 256 ))
k4 = strassen(mat6, mat7)
end_time4 = time.time()
diff4 = end_time4 - start_time4

print( 'for dimension = 32, runtime = ' , diff1* 1000 )
```

```
for dimension = 32, runtime =  187.5
for dimension = 64, runtime =  1314.453125
for dimension = 128, runtime =  9227.5390625
for dimension = 256, runtime =  65301.7578125
[Finished in 76.4s]
```

## PROBLEM 3:

Implement n-queens problem using Python and test your code for 4-queens problem.

## ANSWER:

```
class QueenChessBoard :
    def __init__ (self, size) :
        self.size = size
        self.columns = []
```

```
    def place_in_next_row (self, column) :
        self.columns.append(column)
    def remove_in_current_row (self) :
        return self.columns.pop()
    def is_this_column_safe_in_next_row(self, column):
        row = len(self.columns)
        for queen_column in self.columns:
            if column == queen_column:
                return False
        for queen_row, queen_column in 
enumerate(self.columns):
            if queen_column - queen_row == column - row:
                return False
        for queen_row, queen_column in 
enumerate(self.columns):
            if ((self.size - queen_column) - queen_row == 
(self.size - column) - row):
                return False
        return True


    def display (self) :
```

```
        for row in range(self.size):
            for column in range(self.size):
                if column == self.columns[row]:
                    print( 'Q' , end= ' ' )
                else :
                    print( '.' , end= ' ' )
            print()



def solve_queen (size) :
    board = QueenChessBoard(size)
    number_of_solutions = 0
    row = 0
    column = 0
    while True :
        while column < size:
            if board.is_this_column_safe_in_next_row(column):
                board.place_in_next_row(column)
                row += 1
                column = 0
```

```
                break
            else :
                column += 1
        if (column == size or row == size):
            if row == size:
                board.display()
                print()
                number_of_solutions += 1
                board.remove_in_current_row()
                row -= 1
        try :
            prev_column = board.remove_in_current_row()
        except IndexError:
            break
        row -= 1
        column = 1 + prev_column
    print( 'Number of solutions:' , number_of_solutions)

n = int(input( 'Enter n: ' ))
solve_queen(n)
```

## OUTPUT:

```
C:\Users\PRATIK\Desktop>python check.py
Enter n: 1
Q

Number of solutions: 1
```

```
Enter n: 4
. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Number of solutions: 2
```