**Algorithms**

Lab 3: To implement Time complexity of Merge and Quick Sort

Merge sort:

procedure merge(arr1, arr2, original)

i := 0

j := 0

k := 0

while (i < length(arr1) and j < length(arr2))

if arr1[i] < arr2[j]

original[k] := arr1[i]

k := k + 1

i := i + 1

else

original[k] := arr2[j]

k := k + 1

j := j + 1

end while

while (i < length(arr1))

original[k] := arr1[i]

k := k + 1

i := i + 1

end while

while (j < length(arr2))

original[k] := arr2[j]

k := k + 1

j := j + 1

end while

end procedure

procedure sorting(a)

if length(a) = 0 or length(a) = 1

return

end if

mid := length(a) // 2

a1 := a[0:mid]

a2 := a[mid:]

sorting(a1)

sorting(a2)

merge(a1, a2, a)

end procedure

-----------------------------------------------------------------------------------------------

Quick sort:procedure partition(arr, s, e)

pivot := arr[s]

c := 0

for i := s to e

if arr[i] < pivot

c := c + 1

end if

end for

arr[s+c], arr[s] := arr[s], arr[s+c]

pivotin := s + c

i := s

j := e

while i < j

if arr[i] < pivot

i := i + 1

elif arr[j] >= pivot

j := j - 1

else

arr[i], arr[j] := arr[j], arr[i]

i := i + 1

j := j - 1

end if

end while

return pivotin

end procedure

procedure quickSort(arr, s, e)

if s >= e

return

end if

p := partition(arr, s, e)

quickSort(arr, s, p-1)

quickSort(arr, p+1, e)

end procedure

-----------------------------------------------------------------------------------------------------

Lab 4: Analyse the time complexity of Strassen Matrix Multiplication

procedure strassen\_matrix\_multiply(A, B)

if length(A) = 1

return [[A[0][0] \* B[0][0]]]

end if

size := length(A) // 2

# Split matrices into quarters

A11 := [row[0:size] for row in A[0:size]]

A12 := [row[size:] for row in A[0:size]]

A21 := [row[0:size] for row in A[size:]]

A22 := [row[size:] for row in A[size:]]

B11 := [row[0:size] for row in B[0:size]]

B12 := [row[size:] for row in B[0:size]]

B21 := [row[0:size] for row in B[size:]]

B22 := [row[size:] for row in B[size:]]

# Calculate products

P1 := strassen\_matrix\_multiply(matrix\_add(A11, A22), matrix\_add(B11, B22))

P2 := strassen\_matrix\_multiply(matrix\_add(A21, A22), B11)

P3 := strassen\_matrix\_multiply(A11, matrix\_sub(B12, B22))

P4 := strassen\_matrix\_multiply(A22, matrix\_sub(B21, B11))

P5 := strassen\_matrix\_multiply(matrix\_add(A11, A12), B22)

P6 := strassen\_matrix\_multiply(matrix\_sub(A21, A11), matrix\_add(B11, B12))

P7 := strassen\_matrix\_multiply(matrix\_sub(A12, A22), matrix\_add(B21, B22))

# Calculate result quarters

C11 := matrix\_add(matrix\_sub(matrix\_add(P1, P4), P5), P7)

C12 := matrix\_add(P3, P5)

C21 := matrix\_add(P2, P4)

C22 := matrix\_add(matrix\_sub(matrix\_add(P1, P3), P2), P6)

# Combine result quarters into the final matrix

result := [C11[i] + C12[i] for i in range(size)] + [C21[i] + C22[i] for i in range(size)]

return result

end procedure

procedure matrix\_add(A, B)

return [[A[i][j] + B[i][j] for j in range(length(A[i]))] for i in range(length(A))]

end procedure

procedure matrix\_sub(A, B)

return [[A[i][j] - B[i][j] for j in range(length(A[i]))] for i in range(length(A))]

end procedure

------------------------------------------------------------------------------------------------------------------

Lab 5: Implement the time complexity of Heap Sort

procedure heapify(arr, n, i)

largest := i

left\_child := 2 \* i + 1

right\_child := 2 \* i + 2

if left\_child < n and arr[left\_child] > arr[largest]

largest := left\_child

end if

if right\_child < n and arr[right\_child] > arr[largest]

largest := right\_child

end if

if largest ≠ i

swap(arr[i], arr[largest])

heapify(arr, n, largest)

end if

end procedure

procedure heap\_sort(arr)

n := length(arr)

# Build a max heap

for i := n // 2 - 1 to 0 step -1

heapify(arr, n, i)

end for

# Extract elements one by one

for i := n - 1 to 1 step -1

swap(arr[i], arr[0])

heapify(arr, i, 0)

end for

end procedure

----------------------------------------------------------------------------------------------------------------------

Lab 6: To implement Huffman Coding and analyse its time complexity. To implement Minimum Spanning Tree and analyse its time complexity.

Huffmans code:

class HuffmanNode:

function \_\_init\_\_(char, frequency)

self.char := char

self.frequency := frequency

self.left := None

self.right := None

end function

function \_\_lt\_\_(other)

return self.frequency < other.frequency

end function

end class

function build\_huffman\_tree(frequencies)

priority\_queue := [HuffmanNode(char, freq) for char, freq in frequencies.items()]

sort(priority\_queue, key=lambda x: x.frequency)

while length(priority\_queue) > 1

left := pop(priority\_queue, 0)

right := pop(priority\_queue, 0)

merged\_node := HuffmanNode(None, left.frequency + right.frequency)

merged\_node.left := left

merged\_node.right := right

append(priority\_queue, merged\_node)

sort(priority\_queue, key=lambda x: x.frequency)

end while

return priority\_queue[0]

end function

function build\_huffman\_codes(node, current\_code, codes)

if node

if node.char is not None

codes[node.char] := current\_code

end if

build\_huffman\_codes(node.left, concatenate(current\_code, '0'), codes)

build\_huffman\_codes(node.right, concatenate(current\_code, '1'), codes)

end if

end function

function huffman\_encode(data, frequencies)

root := build\_huffman\_tree(frequencies)

codes := {}

build\_huffman\_codes(root, '', codes)

encoded\_data := concatenate([codes.get(char, '') for char in data])

return encoded\_data, root

end function

function huffman\_decode(encoded\_data, root)

decoded\_data := ''

current\_node := root

for bit in encoded\_data

if bit == '0'

current\_node := current\_node.left

else

current\_node := current\_node.right

end if

if current\_node.char is not None

decoded\_data := concatenate(decoded\_data, current\_node.char)

current\_node := root

end if

end for

return decoded\_data

end function

Minimum spanning tree:

class Graph:

function \_\_init\_\_(vertices)

self.vertices := vertices

self.graph := create\_matrix(vertices, vertices, 0)

end function

function add\_edge(u, v, weight)

self.graph[u][v] := weight

self.graph[v][u] := weight

end function

function min\_key(key, mst\_set)

min\_val := infinity

min\_index := -1

for v := 0 to self.vertices - 1

if key[v] < min\_val and not mst\_set[v]

min\_val := key[v]

min\_index := v

end if

end for

return min\_index

end function

function prim\_mst

key := [infinity] \* self.vertices

parent := [-1] \* self.vertices

mst\_set := [false] \* self.vertices

key[0] := 0 # Start with the first vertex

for \_ := 0 to self.vertices - 2

u := min\_key(key, mst\_set)

mst\_set[u] := true

for v := 0 to self.vertices - 1

if self.graph[u][v] > 0 and not mst\_set[v] and key[v] > self.graph[u][v]

key[v] := self.graph[u][v]

parent[v] := u

end if

end for

end for

return parent

end function

end class

-------------------------------------------------------------------------------------------------------------

Lab 7: To implement Dijkstra’s algorithm and analyse its time complexity and to implement Bellman Ford algorithm and analyse its time complexity.

Dijkstra:

class Graph:

function \_\_init\_\_(vertices)

self.vertices := vertices

self.graph := create\_matrix(vertices, vertices, 0)

end function

function add\_edge(u, v, weight)

self.graph[u][v] := weight

self.graph[v][u] := weight

end function

function dijkstra(start)

# Initialize distance array with infinity and source vertex distance as 0

distance := [infinity] \* self.vertices

distance[start] := 0

# Initialize set to keep track of visited vertices

visited := set()

while length(visited) < self.vertices

# Find the vertex with the minimum distance value

min\_distance := infinity

min\_vertex := -1

for v := 0 to self.vertices - 1

if distance[v] < min\_distance and v not in visited

min\_distance := distance[v]

min\_vertex := v

end if

end for

# Add the selected vertex to the visited set

add(visited, min\_vertex)

# Update the distance values of the adjacent vertices

for v := 0 to self.vertices - 1

if self.graph[min\_vertex][v] > 0 and v not in visited

new\_distance := distance[min\_vertex] + self.graph[min\_vertex][v]

if new\_distance < distance[v]

distance[v] := new\_distance

end if

end if

end for

end while

return distance

end function

end class

Bellman ford:

class Graph:

function \_\_init\_\_(vertices)

self.vertices := vertices

self.edges := []

end function

function add\_edge(u, v, weight)

add(self.edges, (u, v, weight))

end function

function bellman\_ford(start)

distance := [infinity] \* self.vertices

distance[start] := 0

for \_ := 0 to self.vertices - 2

for edge in self.edges

u, v, weight := edge

if distance[u] ≠ infinity and distance[u] + weight < distance[v]

distance[v] := distance[u] + weight

end if

end for

end for

for edge in self.edges

u, v, weight := edge

if distance[u] ≠ infinity and distance[u] + weight < distance[v]

raise ValueError("Graph contains a negative cycle")

end if

end for

return distance

end function

end class

---------------------------------------------------------------------------------------------------------------

Lab 8: Implement N Queen's problem using Back Tracking.

function is\_safe(board, row, col, n)

for i := 0 to col - 1

if board[row][i] == 1

return False

end if

end for

for i, j := row, col to 0, -1

if board[i][j] == 1

return False

end if

end for

for i, j := row, col to n - 1, -1

if board[i][j] == 1

return False

end if

end for

return True

end function

function solve\_n\_queens\_util(board, col, n)

if col >= n

return True

end if

for i := 0 to n - 1

if is\_safe(board, i, col, n)

board[i][col] := 1

if solve\_n\_queens\_util(board, col + 1, n)

return True

end if

board[i][col] := 0

end if

end for

return False

end function

function solve\_n\_queens(n)

board := create\_matrix(n, n, 0)

if not solve\_n\_queens\_util(board, 0, n)

print("Solution does not exist")

return False

end if

end function

----------------------------------------------------------------------------------------------------------

Lab 9: To implement Matrix Chain Multiplication and analyse its time complexity. To implement Longest Common Subsequence problem and analyse its time complexity.

Matrix Chain Multiplication:

function matrix\_chain\_multiplication(p)

n := length(p) - 1

m := create\_matrix(n + 1, n + 1, 0)

s := create\_matrix(n + 1, n + 1, 0)

for length := 2 to n + 1

for i := 1 to n - length + 2

j := i + length - 1

m[i][j] := infinity

for k := i to j - 1

cost := m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j]

if cost < m[i][j]

m[i][j] := cost

s[i][j] := k

end if

end for

end for

end for

return m, s

end function

function print\_optimal\_parentheses(s, i, j)

if i == j

print("A" + i)

else

print("(")

print\_optimal\_parentheses(s, i, s[i][j])

print\_optimal\_parentheses(s, s[i][j] + 1, j)

print(")")

end if

end function

function run\_experiment(matrix\_dimensions)

start\_time := current\_time()

m, s := matrix\_chain\_multiplication(matrix\_dimensions)

end\_time := current\_time()

elapsed\_time := end\_time - start\_time

print("Matrix Dimensions:", matrix\_dimensions)

print("Minimum number of multiplications:", m[1][length(m) - 1])

print("Optimal Parentheses:", print\_optimal\_parentheses(s, 1, length(s) - 1))

print("Time taken:", elapsed\_time)

print("\n")

end function

----------------------------------------------------------------------------------------------------------

Longest Common Subsequence:

function longest\_common\_subsequence(X, Y)

m, n := length(X), length(Y)

lcs\_table := create\_matrix(m + 1, n + 1, 0)

for i := 0 to m

for j := 0 to n

if i == 0 or j == 0

lcs\_table[i][j] := 0

else if X[i - 1] == Y[j - 1]

lcs\_table[i][j] := lcs\_table[i - 1][j - 1] + 1

else

lcs\_table[i][j] := max(lcs\_table[i - 1][j], lcs\_table[i][j - 1])

end if

end for

end for

lcs := []

i, j := m, n

while i > 0 and j > 0

if X[i - 1] == Y[j - 1]

append(lcs, X[i - 1])

i := i - 1

j := j - 1

else if lcs\_table[i - 1][j] > lcs\_table[i][j - 1]

i := i - 1

else

j := j - 1

end if

end while

return lcs

end function

----------------------------------------------------------------------------------------------------------

**Lab 10**: To implement naïve String Matching algorithm, Rabin Karp algorithm and Knuth Morris Pratt algorithm and analyse its time complexity.

Naive String Matching:

function naive\_string\_matching(text, pattern)

occurrences := []

n, m := length(text), length(pattern)

start\_time := current\_time()

for i := 0 to n - m

if text[i:i + m] == pattern

append(occurrences, i)

end if

end for

end\_time := current\_time()

elapsed\_time := end\_time - start\_time

return occurrences, elapsed\_time

end function

Rabin karp:

function rabin\_karp(text, pattern, d=256, q=101)

occurrences := []

n, m := length(text), length(pattern)

start\_time := current\_time()

h\_pattern := sum(ord(pattern[i]) \* d^(m - 1 - i) % q for i := 0 to m - 1) % q

h\_window := sum(ord(text[i]) \* d^(m - 1 - i) % q for i := 0 to m - 1) % q

for i := 0 to n - m

if h\_pattern == h\_window and text[i:i + m] == pattern

append(occurrences, i)

end if

if i < n - m

h\_window := (d \* (h\_window - ord(text[i]) \* d^(m - 1)) + ord(text[i + m])) % q

end if

end for

end\_time := current\_time()

elapsed\_time := end\_time - start\_time

return occurrences, elapsed\_time

end function

Knauth Morris pratt:

def compute\_prefix\_function(pattern):

m = len(pattern)

pi = [0] \* m

k = 0

for q in range(1, m):

while k > 0 and pattern[k] != pattern[q]:

k = pi[k - 1]

if pattern[k] == pattern[q]:

k += 1

pi[q] = k

return pi

def knuth\_morris\_pratt(text, pattern):

occurrences = []

n, m = len(text), len(pattern)

start\_time = time.time()

pi = compute\_prefix\_function(pattern)

q = 0

for i in range(n):

while q > 0 and pattern[q] != text[i]:

q = pi[q - 1]

if pattern[q] == text[i]:

q += 1

if q == m:

occurrences.append(i - m + 1)

q = pi[q - 1]

end\_time = time.time()

elapsed\_time = f"{end\_time - start\_time:.6f}"

return occurrences, elapsed\_time