**STELLA MWIHAKI**

**SCT221-0832/2021**

**ASSIGNMENT**

Q1) Recursive Algorithm for Extracting Subsentence :def extract subsentence(sentence, start, end):

# Base case: if start index exceeds end index, return empty string

if start > end:

return ""

# Base case: if start index is greater than the length of the sentence, return empty string

if start >= len(sentence):

return ""

# Base case: if end index is less than 0, return empty string

if end < 0:

return ""

# Base case: if end index exceeds the length of the sentence, adjust it to the maximum index

if end >= len(sentence):

end = len(sentence) - 1

# Find the indices of the start and end of the next word

while start < len(sentence) and sentence[start] == ' ':

start += 1

word\_ start = start

while end >= 0 and sentence[end] == ' ':

end -= 1

word\_ end = end

# Extract the word between start and end indices

subsentence = sentence[word\_ start : word\_ end + 1]

# Recur for the next word

return subsentence + extract\_ subsentence(sentence, end + 1, end)

# Example usage:

sentence = "This is a sample sentence"

start\_ index = 5

end\_ index = 13

result = extract\_ subsentence(sentence, start\_ index, end \_index)

print(result) # Output: "is a sample"

Recurrence Relation: Let ( T(n) ) be the time complexity of the algorithm, where ( n ) is the length of the sentence. The recurrence relation can be expressed as: [ T(n) = T(n-1) + O(1) ]Time Complexity Analysis using Tracing Tree Method: When tracing the recursive calls on a tree: At each level of recursion, one character is processed and the recursion progresses towards the end of the sentence. The depth of the recursion tree is ( n ), where ( n ) is the length of the sentence. Therefore, the time complexity is ( O(n) ).Q2) Algorithm for Circular Shifting: def circular\_ shift(array, k):

n = len(array)

result = [0] \* n

for i in range(n):

result[(i + k) % n] = array[i]

return result

# Example usage:

A = [5, 15, 29, 35, 42]

k = 2

result = circular\_ shift(A, k)

print(result) # Output: [35, 42, 5, 15, 29]

Recurrence Relation: Let ( T(n) ) be the time complexity of the algorithm, where ( n ) is the number of elements in the array. The recurrence relation for the iterative algorithm is: [ T(n) = O(1) ]Time Complexity Analysis using Iterative Method: The algorithm iterates through each element of the array once, performing constant-time operations. Therefore, the time complexity is ( O(n) ).