**EDWIN KIMATHI   
SCT221-0762/2021**

**Design and Analysis of Algorithms**

**ICS 2301**

**Assignment**

**Question 1**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

char \*extract\_substring(char \*sentence, int start, int end) {

if (start >= end || start < 0 || end >= strlen(sentence)) {

return NULL; // Handle invalid indices or empty substring

}

int sub\_len = end - start + 1;

char \*substring = (char \*)malloc(sub\_len \* sizeof(char) + 1); // Allocate memory for substring + null terminator

if (substring == NULL) {

return NULL; // Handle memory allocation failure

}

int i = 0;

while (start < end) {

substring[i++] = sentence[start++];

}

substring[i] = '\0'; // Add null terminator

return substring;

}

int main() {

char sentence[] = "This is a sample sentence";

int start = 5;

int end = 11;

char \*substring = extract substring (sentence, start, end);

if (substring! = NULL) {

printf("Extracted substring: %s\n", substring);

free(substring); // Free allocated memory

} else {

printf("Error: Invalid indices or memory allocation failure\n");

}

return 0;

}

**Recurrence relation**

Let T (n) rep the rime complexity of the algorithm, where n is size of the input sentence. Recurrence relation can be represented as: T(n) = T(n-k) + O (k)

Where:

k is the number of characters processed in one recursive call

O(k) rep time complexity of extracting a word of length k

**Time Complexity Analysis (Tracing Tree):**

1. The base case has a constant time complexity of O(1).
2. The recursive case involves copying a character (O(1)) and a recursive call.
3. In the worst case (no overlap between start and end), the recursion continues until the base case is reached.
4. The total number of recursive calls is equal to the length of the substring (n).
5. Each call involves a constant amount of work (O(1)).

Therefore, the time complexity of the algorithm is T(n) = O(n), which is linear in the length of the substring being extracted.

**Question 2**

#include <stdio.h>

void circular\_shift(int A[], int n, int k) {

k = k % n; // Handle k greater than or equal to n (full rotations)

// Reverse the entire array

for (int i = 0, j = n - 1; i < j; i++, j--) {

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

// Reverse the first k elements

for (int i = 0, j = k - 1; i < j; i++, j--) {

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

// Reverse the remaining elements (n-k to n-1)

for (int i = k, j = n - 1; i < j; i++, j--) {

int temp = A[i];

A[i] = A[j];

A[j] = temp;

}

}

int main() {

int A[] = {5, 15, 29, 35, 42};

int n = sizeof(A) / sizeof(A[0]);

int k = 2;

circular\_shift(A, n, k);

printf("Shifted array: ");

for (int i = 0; i < n; i++) {

printf("%d ", A[i]);

}

printf("\n");

return 0;

}

**Explanation:**

1. The circular shift function takes the array A, its size n, and the number of positions to shift k as input.
2. It handles the case where k is greater than or equal to n (full rotations) by taking the modulo with n.
3. It uses three separate loops to achieve the circular shift:
   * The first loop reverses the entire array. This effectively moves the elements that need to be shifted to the end.
   * The second loop reverses the first k elements. This puts the elements that were at the end (originally to be shifted) at the beginning in the correct order.
   * The third loop reverses the remaining elements (n-k to n-1). This completes the circular shift by placing the remaining elements after the shifted elements.

**Time Complexity Analysis:**

* Each loop iterates n/2 times (ignoring the constant overhead of loop control variables) in the worst case.
* There are three loops.
* Therefore, the total time complexity is T(n) = 3 \* (n/2) which simplifies to O(n).

This iterative approach achieves a linear time complexity of O(n), making it efficient for circularly shifting arrays in C.