Student’s Name

DOK MIYOM CHOL SCT221-0448/2021

Institutional Affiliation

JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

Instructor’s Name

CHARLES MWABU

Course Unit

Design and Analysis of Algorithm

Date

3/4/2024

**Q1)** Write an efficient recursive algorithm that takes a sentence, starting index, and ending index. The algorithm should then return a sentence that contains words between the starting and ending indices. Write the recurrence relation of your algorithm and find time complexity using the tracing tree method.

**The algorithm to solve the problem.**

1. Initialize an empty string to store the result.

2. Start iterating through the characters of the sentence from the starting index.

3. If the current character is not a space and it falls within the specified range (starting and ending indices), add it to the result string.

4. If the current character is a space, check if the previous character was part of a word (i.e., the result string is not empty). If so, add a space to separate words in the result string.

5. Recur with the next index until reaching the ending index.

6. Return the result string.

**Here's the recurrence relation for the algorithm:**

Let **T (*n*)** be the time complexity of the algorithm, where n represents the number of characters in the sentence.

***T* (*n*) = *T* (*n*−1) + *O* (1)**

The time complexity of the algorithm calculated using the tracing tree method is as follows:

Tracing Tree:

**T (*n*)**

**T (*n*-1)**

2

**T (*n-2)***

3

**T (*n-3)***

4

**T (*n-4)***

From the tracing tree, we can observe that each level of the tree adds a constant time operation ***(O (1)).*** The height of the tree is ***n***. Therefore, the time complexity is the sum of the operations at each level, which is ***O*** (*n*).

So, the time complexity of the algorithm using the tracing tree method is ***O*** (*n*).

#include <iostream>

#include <string>

using namespace std;

// Recursive function to extract words between starting and ending indices

string extractWords(const string& sentence, int start, int end) {

// Base case: if starting index exceeds ending index or sentence is empty

if (start > end || start >= sentence.length()) {

return "";

}

string result = ""; // Initialize an empty string to store the result

// If the current character is not a space and falls within the range,

// add it to the result string

if (sentence[start] != ' ' && start <= end) {

result += sentence[start];

}

// If the current character is a space and the previous character was part of a word,

// add a space to separate words

if (sentence[start] == ' ' && start <= end && start > 0 && sentence[start - 1] != ' ') {

result += ' ';

}

// Recur with the next index

return result + extractWords(sentence, start + 1, end);

}

int main() {

string sentence;

int start, end;

// Input sentence, starting index, and ending index

cout << "Enter a sentence: ";

getline(cin, sentence);

cout << "Enter starting index: ";

cin >> start;

cout << "Enter ending index: ";

cin >> end;

// Extract words between starting and ending indices

string result = extractWords(sentence, start, end);

// Output the extracted words

cout << "Words between indices " << start << " and " << end << ": " << result << endl;

return 0;

}

Q2) Write an efficient algorithm that takes an array A [...) of sorted integers and returns an array with elements that have been circularly shifted k positions to the right. For example, a sorted array A = [5, 15, 29, 35, 42] is converted to A [35, 42, 5, 15, 27, 29] after circularly shifted 2 positions, while the same array A = [5, 15, 29, 35, 42] is converted to A [27, 29, 35, 42, 5, 15] after circularly shifted 4 positions. Write the recurrence relation of your solution and find the time complexity of your algorithm using an iterative method.

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

void reverseArray(vector<int>& arr, int start, int end) {

while (start < end) {

swap(arr[start], arr[end]);

start++;

end--;

}

}

vector<int> circularShift(vector<int>& arr, int n1, int n2, int k) {

int n = n2 - n1 + 1;

k %= n; // to handle cases where k is larger than n

// Step 1: Reverse the entire range

reverseArray(arr, n1, n2);

// Step 2: Reverse the first k elements within the range

reverseArray(arr, n1, n1 + k - 1);

// Step 3: Reverse the remaining elements within the range

reverseArray(arr, n1 + k, n2);

return arr;

}

int main() {

vector<int> A = {5, 15, 29, 35, 42};

int n1 = 0; // Start index

int n2 = A.size() - 1; // End index

int k1 = 2;

int k2 = 4;

cout << "Original Array: ";

for (int i = n1; i <= n2; ++i) {

cout << A[i] << " ";

}

cout << endl;

vector<int> result1 = circularShift(A, n1, n2, k1);

cout << "Circularly shifted by " << k1 << " positions: ";

for (int i = n1; i <= n2; ++i) {

cout << result1[i] << " ";

}

cout << endl;

vector<int> result2 = circularShift(A, n1, n2, k2);

cout << "Circularly shifted by " << k2 << " positions: ";

for (int i = n1; i <= n2; ++i) {

cout << result2[i] << " ";

}

cout << endl;

return 0;

}

**SOLVING THE RECURRENCE RELATIONS**

Given: T(n) = 3\*T(n/2) + n

Let's expand the recurrence relation recursively:

T(n) = 3T(n/2) + n = 3(3T(n/4) + n/2) + n = 3^2 \* T(n/4) + 3n/2 + n = 3^2 \* (3T(n/8) + n/4) + 3n/2 + n = 3^3 \* T(n/8) + 3^2 \* n/4 + 3\*n/2 + n ...

After k steps, we have: T(n) = 3^k \* T(n/2^k) + n \* (1 + 3/2 + (3/2)^2 + ... + (3/2)^(k-1))

We stop when n/2^k = 1, which happens when k = log₂n.

So, T(n) = 3^log₂n \* T(1) + n \* (1 + 3/2 + (3/2)^2 + ... + (3/2)^(log₂n-1))

Since T(1) is a constant, and the sum in the second term is a geometric series, we can solve it:

T(n) = O(3^log₂n) + O(n \* (1 + (3/2) + (3/2)^2 + ... + (3/2)^(log₂n-1)))

Using the formula for the sum of a finite geometric series:

T(n) = O(3^log₂n) + O(n \* ((1 - (3/2)^log₂n) / (1 - 3/2)))

Since (3/2)^log₂n = n^(log₂3 - 1) and (1 - (3/2)^log₂n) / (1 - 3/2) is a constant, we have:

T(n) = O(3^log₂n) + O(n \* (1 - n^(log₂3 - 1)) / (1 - 3/2)) = O(n^log₂3) + O(n) = O(n^log₂3)

So, the time complexity of the given recurrence relation T(n) = 3\*T(n/2) + n is O(n^log₂3).