Quiz 4 Assignment

MINHAJUL ABEDIN BHUIYAN
ID: 210042148
Department: Computer Science & Engineering
Program: Software Engineering
ISLAMIC UNIVERSITY OF TECHNOLOGY
Data Structure (CSE 4303)

Topic: Introduction to data structures,
concepts of efficiency, elementary
data structures: arrays,
records, pointers, examples of random accessing.

31 December, 2023

Abstract

This article provides a formal yet accessible examination of data structures, introducing the core principles of efficiency and essential structures such as arrays, records, and pointers. Through real-world examples, the concept of random accessing is elucidated, offering readers a practical understanding. The LaTeX-crafted document employs a structured approach, guiding individuals through key concepts, operations, and solutions. Whether one is new to the subject or seeking a comprehensive review, this article endeavors to present data structures in a clear, formal, and widely understandable manner.

Story/Realistic problem Scenario

The Algorithmic Alchemist In the dynamic city of Algorithmica, financial virtuoso Alice stumbled upon an ancient manuscript promising untold riches through a cryptic sequence of numbers. In a quest fueled by curiosity and ambition, she unleashed a powerful program to decipher the hidden code and unlock the maximum subarray sum.

Input-Output

Input	Output
8 -1 3 -2 5 3 -5 2 2	9

With a fusion of programming prowess and mathematical magic, Alice's algorithm unraveled the secret, exposing a maximum subarray sum of '9'. As the news spread, Alice became the talk of the town—an Algorithmic Alchemist, transforming numerical mysteries into wealth through the art of programming.

Basic understanding of assigned topic

Introduction:

Data structures are essential in computer science for efficiently organizing and storing data, impacting the performance of algorithms. This overview focuses on key concepts in C++.

Efficiency Concepts:

Time Complexity: Measures algorithm speed relative to input size (Big O notation).
Space Complexity: Examines memory usage efficiency (Big O notation).

Elementary Data Structures:

Arrays: Contiguous memory storage for elements of the same type.
Example:

```
    int numbers[5] = {1, 2, 3, 4, 5};
    int element = numbers[2];  // Random access
```

Records (Structures): Groups different data types under one name.
Example:

```
    struct Student
    {
        int ID;
        std::string name;
        float grade;
    };

    int main()
    {
       Student myStudent = {123, "John Doe", 85.5};
    }
```

Pointers: Store memory addresses for dynamic memory use.
Example:

```
    int* intPointer;
    int x = 10;
    intPointer = &x;  // Point to variable x
```

Example of Random Access:

```
#include <iostream>

int main()
{
    int myArray[5] = {10, 20, 30, 40, 50};
    int element = myArray[2];  // Random access
    std::cout << "Element at index 2: " << element << std::endl;
    return 0;
}
```

Operations supported

Arrays

Initialization: An array is a collection of elements of the same data type stored in contiguous memory locations. It's initialized by specifying the data type and the number of elements.

```
int numbers[5] = {1, 2, 3, 4, 5};
```

Accessing Elements: Elements in an array are accessed using their index. The index starts from 0, so `numbers[2]` refers to the third element.

```
int element = numbers[2];  // Accessing the third element
```

Modification: Values in an array can be modified by assigning new values to specific indices.

```
numbers[2] = 10;  // Modifying the third element
```

Traversal: Traversal involves accessing each element of the array. A loop is commonly used for this purpose.

```
for (int i = 0; i < 5; ++i)
{
    // Access elements using numbers[i]
}
```

Records (Structures)

Declaration: A structure (or record) allows grouping different data types under a single name. It's declared using the `struct` keyword.

```
struct Student
{
    int ID;
    std::string name;
```

```
    float grade;
};
```

Initialization: Instances of a structure are created by specifying values for each member.

```
Student myStudent = {123, "John Doe", 85.5};
```

Accessing Members Individual members of a structure are accessed using the dot notation.

```
int studentID = myStudent.ID;
myStudent.grade = 90.0;
```

Pointers

Declaration: A pointer is a variable that stores the memory address of another variable. It's declared by appending `*` to the data type.

```
int* intPointer;
```

Initialization: A pointer is initialized by assigning the address of a variable to it.

```
int x = 10;
intPointer = &x;  // intPointer now holds the address of x
```

Dynamic Memory Allocation: Pointers are commonly used for dynamic memory allocation with `new`.

```
int* dynamicInt = new int;
```

Solution

Input:

An integer n: the size of the array.
An array of integers $x_1, x_2, \ldots, x_n$.

Output:

The maximum subarray sum.

a) Intuition behind the solution:

The problem can be efficiently solved using Kadane's Algorithm, which involves iterating through the array and maintaining a running sum of the subarray. The key is to reset the sum to 0 whenever it becomes negative, ensuring that we always consider the maximum subarray sum.

b) Solution steps:

```cpp
#include <iostream>
#include <bits/stdc++.h>
using namespace std;

int maxSubarraySum(int arr[], int n)
{
    int maxSum = INT_MIN;
    int currentSum = 0;

    for (int i = 0; i < n; ++i)
    {
        currentSum += arr[i];

        // If the current sum becomes negative, reset it to 0.
        if (currentSum < 0)
        {
            currentSum = 0;
        }

        // Update the maximum sum if the current sum is greater.
        if (currentSum > maxSum)
        {
            maxSum = currentSum;
        }
    }

    return maxSum;
}

int main()
{
    int n;
    cin >> n;

    int arr[n];
    for (int i = 0; i < n; ++i)
    {
```

```
        std::cin >> arr[i];
    }

    int result = maxSubarraySum(arr, n);
    cout << result << std::endl;

    return 0;
}
```

c) Complexity Analysis:

Time Complexity: $O(n)$ - The algorithm iterates through the array once.
Space Complexity: $O(1)$ - The algorithm uses a constant amount of space regardless of the input size.
This algorithm is efficient and suitable for large arrays, making it a practical solution for the given problem.

Conclusion Comments

The Kadane's Algorithm for finding the maximum subarray sum in a given array is a highly efficient and widely-used approach. It is particularly useful in scenarios where the goal is to optimize the identification of the contiguous subarray with the maximum sum. Here are some points to consider:

Where it can be used:

- **Financial Analysis:** Kadane's Algorithm is employed in financial applications where analyzing time-series data, such as stock prices, requires identifying periods of maximum profitability or loss.
- **Signal Processing:** In signal processing, it is utilized for detecting patterns in signals, where the maximum subarray sum corresponds to the most significant signal strength.
- **Dynamic Programming:** Kadane's Algorithm is a classic example of dynamic programming and is applied in various problem-solving scenarios where optimal substructure is present.
- **Data Mining:** The algorithm finds applications in data mining tasks where identifying patterns or trends within a dataset is crucial.

Where it can't be used:

- **Non-contiguous Subarrays:** If the problem requires identifying the maximum sum over non-contiguous subarrays, Kadane's Algorithm is not suitable. It specifically targets contiguous subarrays.
- **All Negative Numbers:** If the array consists entirely of negative numbers, the algorithm might not provide the desired result, as the maximum subarray sum would then be the single largest negative number.
- **Global Optimization:** In scenarios where the goal is to optimize a global function considering non-contiguous elements, other algorithms or approaches might be more appropriate.

In summary, Kadane's Algorithm is a powerful tool for solving problems related to finding the maximum subarray sum in contiguous sequences. However, it is essential to assess the problem requirements to determine whether this algorithm aligns with the specific needs of the task at hand.

Other Problem Links

1. Problem 1: Traffic Lights
2. Problem 2: Room Allocation
3. Problem 3: Nested Ranges Check

References

- https://www.geeksforgeeks.org/
- https://www.w3schools.com/cpp/cpp_arrays.asp
- https://www.openai.com
- https://cses.fi/problemset/