Learning Objectives

Learners will be able to…

- **Define space complexity**
- **Compare and contrast Arrays and ArrayLists**
- **Analyze insertion and merge sorting algorithms**

Space Complexity Analysis

Introduction to Space Complexity Analysis

After delving into time complexity, it's essential to understand another crucial aspect of algorithm analysis, **Space Complexity**. Space complexity provides us with a measure of the amount of memory an algorithm needs to run to completion. It's as crucial as time complexity, especially in systems with limited memory resources or where memory cost is a significant factor.

info

The **space complexity** of an algorithm quantifies the amount of space or memory taken by an algorithm to execute as a function of the length of the input. It includes the memory required for the input data itself and any additional space needed for the algorithm to transform the input and produce the output.

Space complexity is sometimes overlooked in favor of time complexity, but it's particularly important in today's computing world. As datasets become larger, an algorithm that was previously feasible could become impractical due to high memory requirements. So understanding space complexity is essential to ensure our solutions scale well.

In this assignment, we will explore different algorithms, their space complexity, and how to analyze them. Furthermore, we'll also discuss the trade-offs between time and space complexity, as optimizing for one often comes at the expense of the other. **Please note:** more often than not, when you are asked about the complexity of an algorithm (unless specifically ask for space) they are referring to time complexity or both.

Fundamental Concepts

Before we get into different data structures and the space complexity of some sorting algorithms, we need to cover some fundamental concepts first. This will help us understand and quantify the space complexity of algorithms.

Fixed Space and Variable Space

Space complexity deals with memory usage of an algorithm. We can categorize this into two ideas:

1. **Fixed Space**: This is the space required to store certain data and variables, which does not change during the execution of the algorithm. For instance, simple variables and constants used in the algorithm fall into this category.

2. **Variable Space**: This is the space needed by variables that dynamically change during the execution of the algorithm. It includes things like dynamic data structures, recursion stack space, etc.

The total space required by an algorithm is the sum of the fixed space and the variable space.

Consider the following Java code snippet:

```java
public class SpaceComplexity {

    public static void main(String[] args) {
        int n = 100;     // fixed space
        int[] array = new int[n];    // variable space

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

In this program, the integer `n` takes up a fixed space, while the `array` is variable space, which depends on the value of `n`.

When calculating space complexity, we usually focus on the variable space as it is the one that grows with the input size.

Big O Notation for Space Complexity

Just like time complexity, we use Big O notation to represent space complexity. This allows us to express the space complexity of an algorithm in relation to the size of the input.

In the example above, the space complexity of the program is $O(n)$. That's because the amount of space used by the array scales linearly with the input `n`.

In the following sections, we will look into more complex examples and also compare the space complexities of two commonly used data structures in Java: Arrays and ArrayLists. This will provide us with a practical understanding of how data structure choices can impact the memory utilization of our programs.

Auxiliary Space

Space complexity is often confused for auxiliary space. Auxiliary space is any additional storage space used to assist in the execution of a program.

$$\text{Space Complexity} = \text{Auxiliary space} + \text{Space use by input values}$$

Space complexity is dependent upon auxiliary space. As auxiliary space increases, so does space complexity.

Diving Deeper

To fully understand the significance of auxiliary space, let's take a look at an example:

```java
public class AuxiliarySpace {

    public static int findSum(int[] array) {
        int sum = 0;    // auxiliary space

        for (int i = 0; i < array.length; i++) {
            sum += array[i];
        }

        return sum;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};    // space used by input

        int sum = findSum(array);

        System.out.println("Sum = " + sum);
    }
}
```

In the `findSum` method above, the space used by the `array` is not considered auxiliary space, as it's part of the input to the method. However, the `sum` variable, which is used to keep track of the total sum of the array, is considered auxiliary space because it's extra space used by the algorithm, independent of the input size.

The concept of auxiliary space becomes especially important when we discuss algorithms that might take in large amounts of data as input, but only use a small, fixed amount of extra space to perform their operations.

In this example, no matter how large our input array is, our `findSum` function only ever needs a single integer's worth of extra space to perform its calculation. Hence, the auxiliary space complexity is $O(1)$, which means it uses constant extra space.

Being able to minimize auxiliary space can be a significant advantage when dealing with large data, as it allows for more memory-efficient algorithms. Understanding the difference between space complexity and auxiliary space complexity helps you write algorithms that not only perform well but are also resource-efficient.

In the next section, we'll take this concept further, comparing the space complexities and auxiliary space used by two commonly used data structures in Java: Arrays and ArrayLists.

Array vs ArrayList

Comparing Space Complexity

One of the essential factors to consider while choosing the right data structure is its space complexity. Understanding how much space a data structure consumes can significantly impact the overall efficiency of an algorithm. Let’s compare Arrays and ArrayLists in Java to understand this better.

Arrays in Java

An array in Java is a static data structure that holds a fixed number of values of a single type. The length of the array is established when the array is created, and it cannot be changed.

Here’s an example of an array. The variable `array` is instantiated with enough memory to hold five integers. The program then iterates over the array and assigns a random integer (0-99) to each element. Finally, the program uses the `.add()` method to add a sixth element to the array.

```java
import java.util.Random;

public class ArrayVsArrayList {

    public static void main(String args[]) {
        Random rand = new Random();
        int[] array = new int[5];

        for (int i = 0; i < 5; i++) {
          array[i] = rand.nextInt(100);
        }

        array.add(0);
    }
}
```

As expected, this code produces an error. Arrays have a fixed length, and trying to invoke a method to change its size causes a problem for the compiler. Once you instance an array with a given size, it cannot change. Because of its fixed length, we can say the space complexity for an array is $O(n)$, where n is the number of elements in the array.

ArrayLists in Java

On the other hand, an ArrayList in Java is a dynamic data structure, which means it can grow or shrink as needed. When elements are added to an ArrayList, its capacity grows automatically.

Here's an example of how an ArrayList. The same actions are performed on it as the array example from above.

```java
import java.util.ArrayList;
import java.util.Random;

public class ArrayVsArrayList {

    public static void main(String args[]) {
        Random rand = new Random();
        ArrayList<Integer> numbers = new ArrayList<Integer>(5);

        for (int i = 0; i < 5; i++) {
          numbers.add(rand.nextInt(100));
        }

        numbers.add(0);
    }
}
```

When an element is added to an ArrayList, if it is full, a new, larger array is created, and the old array is copied into the new one. Therefore, in addition to the space required for the elements themselves, extra space is required during the resizing operation. The space complexity of an ArrayList is generally $O(n)$, but the auxiliary space complexity during the resize operation is $O(n)$ as well, making ArrayLists less memory efficient than arrays during insertions when resizing is required.

While both Arrays and ArrayLists have a space complexity of $O(n)$, ArrayLists require more auxiliary space because of the resizing operations. Therefore, if the size of the data structure is known in advance and will not change, an array would be a more memory-efficient choice. However, if you need a flexible size data structure, ArrayList provides this functionality at the cost of additional memory. The choice between using an Array or an ArrayList often comes down to a trade-off between memory efficiency and flexibility. Understanding these trade-offs is crucial for effective programming.

Insertion sort

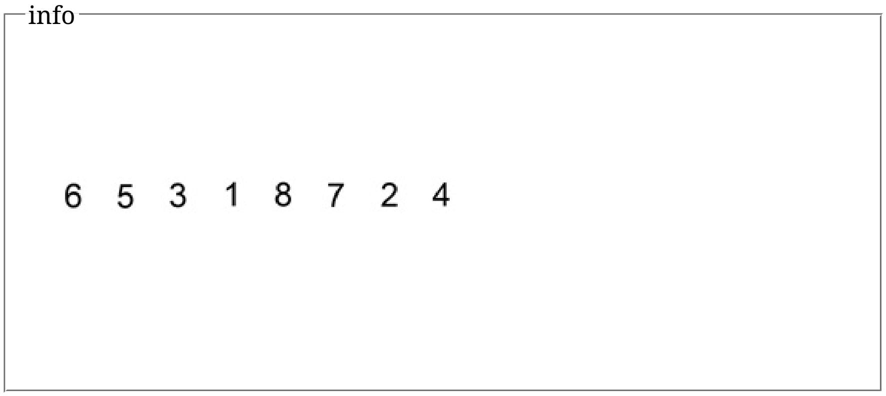
In order to better help visualize the space complexities. We are going to compare the space complexities of two sorting algorithms: **insertion sort** and **merge sort**.

When interacting with **insertion sort**, we use only a constant amount of additional memory apart from the input array, the space complexity is $O(1)$. A constant Big O value in terms of space complexity means that our algorithm will take the same amount of space regardless of the input size. In **Merge sort**, we require an auxiliary array to store the merged subarray. The size of this auxiliary array is at most n, and as a result, the space complexity of Merge sort is $O(n)$.

Insertion Sort

An insertion sort is a simple sorting algorithm that works by repeatedly inserting an unsorted element into a sorted portion of an array until the entire array is sorted. A visual representation is provided below:

info



Space Complexity in Context

Insertion sort is an efficient sorting algorithm for small data sets. It sorts the input list by treating the leftmost part of the array as a sorted segment. It then iterates through the rest of the list and inserts the element in its correct position in the sorted segment.

Let’s take a look at a simple implementation of Insertion Sort in Java:

```
public class InsertionSort {

    // Method to perform insertion sort on the given array
    public static void sort(int[] arr) {
        int n = arr.length;

        // Traverse through the array from the second element
        for (int i = 1; i < n; i++) {
            int key = arr[i]; // Store the current element to be
        inserted into the sorted part
            int j = i - 1; // Initialize the pointer to the last
        element of the sorted part

            // Shift elements greater than the key to the right
            // to create space for inserting the key in the
        correct position
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            // Insert the key into the correct position in the
        sorted part
            arr[j + 1] = key;
        }
    }
```

Space Complexity of Insertion Sort

In terms of space complexity, insertion sort is quite efficient. The space complexity of an algorithm is determined by the amount of memory it requires in relation to the size of the input. In the case of insertion sort:

- **Fixed Space**: This includes variables and constants. In the implementation above, the fixed space is occupied by variables such as `n`, `i`, `key`, and `j`.
- **Variable Space**: This includes dynamically allocated space. In this case, there's no dynamically allocated space, as we're not using any additional data structures, like arrays or linked lists.

Therefore, the total space complexity of insertion sort is the sum of the fixed and variable space. Given that the variable space is zero, and the fixed space does not depend on the size of the input array, we can conclude that the Insertion Sort algorithm has a **constant space complexity**, denoted as $O(1)$. This means the space requirement does not change with the size of the input array.

Understanding the space complexity of sorting algorithms like insertion sort is crucial. Even though the insertion sort might not be the most efficient in terms of time complexity, especially for larger data sets, it's very space-efficient because it sorts the list in place, without needing additional storage, making it an excellent choice when memory is a consideration.

Merge Sort

Merge sort involves breaking down an unsorted array into n subarrays, each consisting of one element, which can be considered sorted in a straightforward manner. Then, we merge these sorted subarrays together repeatedly, forming new sorted subarrays, until we end up with only one sorted array of size n.

A visual representation is provided below:

info

6 5 3 1 8 7 2 4

Link

Space Complexity in Context

Merge sort is another common sorting algorithm that leverages the divide-and-conquer approach to sort an array or list of elements. We will delve more into the divide and conquer in a bit. In short, this means that an algorithm divides the current problem into subproblems. It then solves the subproblems in order to solve the whole problem. The algorithm works by repeatedly splitting the list into two halves until we have sublists with one element each. We then merge these sublists back together such that they are sorted.

Here is a simple implementation of merge sort:

```java
public class MergeSort {

    // Merges two subarrays of arr[].
    // First subarray is arr[left..mid].
    // Second subarray is arr[mid+1..right].
    private static void merge(int arr[], int left, int mid, int
        right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temporary arrays to store the two subarrays
        int leftArray[] = new int[n1];
        int rightArray[] = new int[n2];
        // Copy data to temporary arrays
        for (int i = 0; i < n1; ++i)
            leftArray[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            rightArray[j] = arr[mid + 1 + j];

        // Merge the two temporary arrays back into
        arr[left..right]

        int i = 0, j = 0; // Initialize pointers for the two
        temporary arrays
        int k = left; // Initialize pointer for the merged array

        // Compare elements from leftArray and rightArray and
        put the smaller one into arr
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements from leftArray, if any
        while (i < n1) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        // Copy remaining elements from rightArray, if any
        while (j < n2) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    // Main method that sorts arr[left..right] using merge()
    public static void sort(int arr[], int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            // Sort the first half and the second half
        separately
            sort(arr, left, mid);
            sort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }
```

Space Complexity of Merge Sort

The space complexity of an algorithm involves the amount of memory it needs in proportion to the input size. In the case of Merge Sort:

- **Fixed Space**: This includes variables and constants. In the above implementation, the fixed space is occupied by variables such as `left`, `right`, `mid`, `n1`, `n2`, `i`, `j`, and `k`.

- **Variable Space**: This includes dynamically allocated space. Here, the variable space is used by the temporary arrays `leftArray` and `rightArray`.

As such, the total space complexity of Merge Sort is the sum of the fixed and variable space. Since the variable space in this algorithm scales with the size of the input array (specifically, we need to create temporary arrays for each element in the array), merge sort has a **linear space complexity**, denoted as $O(n)$.

This means that the space required by the merge sort algorithm increases linearly with the size of the input. Although merge sort is efficient in terms of time complexity ($O(nlogn)$), its space complexity is higher than that of some other sorting algorithms, such as insertion sort, which has a space complexity of $O(1)$. Understanding this trade-off is critical when dealing with large datasets and limited memory resources.

Formative Assessment 1

Formative Assessment 2