Introduction

This chapter lays the groundwork for the material covered in the course and sets out the expectations for the readers.

The Interview Room

You're in an interview room, and you've just completed white-boarding the interviewer's question. Inside, you feel ecstatic; It's your dream company and you nailed the question. However, just when you think the coast is clear, the interviewer follows up with a question on the *time* or *space* complexity of your solution. The blank drawn across your face gives away your ineptness at analyzing algorithm complexity in space or time. In this common interview situation, most computer science graduates are still able to muster up a reasonable answer. For candidates lacking the formal education, however, reasoning about algorithms in terms of the ***big Oh*** notation becomes a ***big uh oh!***.

The intended readership of this course includes folks who are graduates of boot-camps, career-switchers into programming jobs or just established

industry veterans who want a quick revision on the topic of complexity theory. The course uses layman's terms to describe the concepts commonly used in the industry while intentionally avoiding the intricate mathematical gymnastics often involved in analyzing algorithms so that the subject material is comprehensible for readers at all skill levels. At the same time, this course may not deliver great value to individuals who already have a firm grasp of algorithms and data-structures or to those that hold a degree in computer science.

Why does complexity matter?

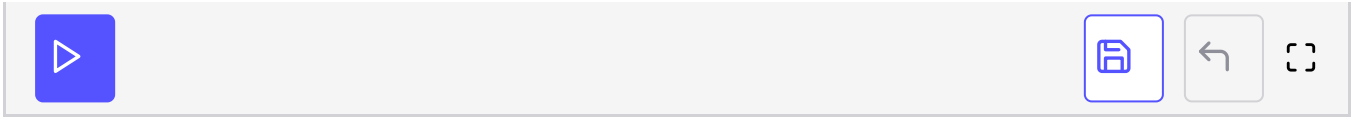
If you are working on a class assignment or a customer application with a few dozen users, you can take the liberty to ignore the space and time complexity of your solution. However, the moment you step into the professional world and write software with strict SLAs (service level agreements) or for millions of users, your choice of algorithm and data-structures starts to matter. A well designed and thought out software will scale elegantly and set itself apart from poorly written competitors.

Usually, the *101* example used to teach complexity is sorting of integers. Below are three example programs that sort integers using different sorting algorithms.

Bubble Sort

```
import java.util.Random;
class Demonstration {

    static int SIZE = 10000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        bubbleSort(input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void bubbleSort(int[] input) {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE - 1; j++) {
                if (input[j] > input[j + 1]) {
                    int tmp = input[j];
                    input[j] = input[j + 1];
                    input[j + 1] = tmp;
                }
```

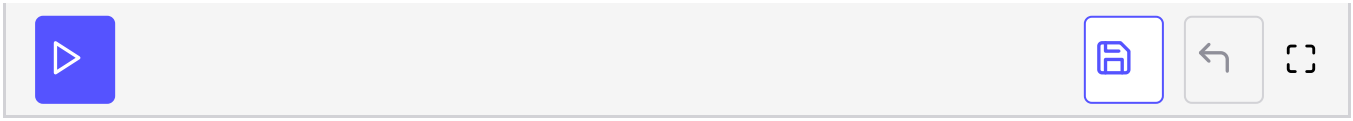
```
            }
        }
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
}
```



Merge Sort

```
import java.util.Random;
class Demonstration {

    static int SIZE = 5000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];
    static int[] scratch = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void mergeSort(int start, int end, int[] input) {

        if (start == end) {
            return;
        }

        int mid = (start + end) / 2;

        // sort first half
        mergeSort(start, mid, input);

        // sort second half
        mergeSort(mid + 1, end, input);

        // merge the two sorted arrays
        int i = start;
        int j = mid + 1;
        int k;

        for (k = start; k <= end; k++) {
            scratch[k] = input[k];
        }

        k = start;
        while (k <= end) {

            if (i <= mid && j <= end) {
```

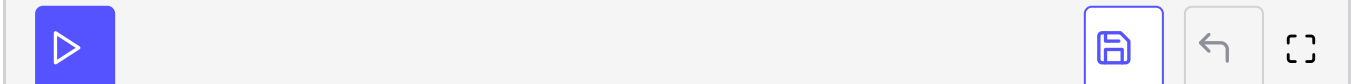
```
                input[k] = Math.min(scratch[i], scratch[j]);

                if (input[k] == scratch[i]) {
                    i++;
                } else {
                    j++;
                }
            } else if (i <= mid && j > end) {
                input[k] = scratch[i];
                i++;
            } else {
                input[k] = scratch[j];
                j++;
            }
            k++;
        }
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(1000);
        }
    }
}
```



Merge Sort

Dual Pivot Quicksort

```
import java.util.Random;
import java.util.Arrays;

class Demonstration {

    static int SIZE = 5000;
    static Random random = new Random(System.currentTimeMillis());
    static int[] input = new int[SIZE];

    public static void main( String args[] ) {
        createTestData();
        long start = System.currentTimeMillis();
        Arrays.sort(input);
        long end = System.currentTimeMillis();
        System.out.println("Time taken = " + (end - start));
    }

    static void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }
}
```



Java's Dual Pivot Quicksort

The code implementations above run on datasets of 5000 elements. Below are some crude tests (run on a Macbook) for the three algorithms. All times are in milliseconds

Size	Bubble Sort	Merge Sort	Java's util.Arrays.sort
5000	90	9	4
10000	299	15	17
100000	24719	26	33
1000000	2474482	150	113

The results tabulated above should provide plenty of motivation for software developers to increase their understanding of algorithmic performance. Bubble sort progressively degrades in performance as the input size increases, whereas the other two algorithms perform roughly the same. We'll have more to say about their performances in the next chapter.

Time & Space

The astute reader would notice the use of an additional array in the merge sort code, named *scratch* array. This is the classic trade-off in the computer science realm. Throwing more space at a problem allows us to bring down the execution time, and vice versa. Generally, time and space have an inversely proportional relationship with each other; if space increases then execution time decreases, and if execution time increases then space decreases.