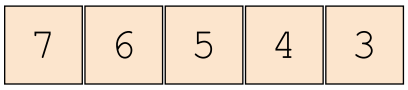
Analyzing Algorithms

We start our journey into the world of complexity by analyzing the insertion sort algorithm.

Insertion Sort Analysis

Let's start with the example of an array that contains 5 elements sorted in the worst possible order, i.e. descending when we want to sort the array in an ascending order. We'll use the insertion sort algorithm as the guinea pig for our analysis.



Array Sorted in Descending Order

We'll not delve into explaining the algorithm itself as our focus is on analyzing algorithm performance. The coursework assumes that readers have an understanding of the fundamental algorithms used in everyday computing tasks, or they can do a quick web-search to get up to speed on the algorithm under discussion. Let's proceed with the implementation of the algorithm first:

```
class Demonstration {

    static void insertionSort(int[] input) {
        for (int i = 0; i < input.length; i++) {
            int key = input[i];
            int j = i - 1;
            while (j >= 0 && input[j] > key) {
                if(input[j] > key){
                    int tmp = input[j];
                    input[j] = key;
                    input[j + 1] = tmp;
                    j--;
                }
            }
        }
    }

    static void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++)
            System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    public static void main( String args[] ) {
        int[] input = new int[] {7, 6, 5, 4, 3, 2, 1};
        insertionSort(input);
        printArray(input);
    }
}
```

Below, we extract out the meat of the algorithm.

```
        for (int i = 0; i < input.length; i++) {
            int key = input[i];
            j = i - 1;
            while (j >= 0 && input[j] > key) {
                if (input[j] > key) {
                    int tmp = input[j];
                    input[j] = key;
                    input[j + 1] = tmp;
                    j--;
                }
            }
        }
```

Analyzing the program

For simplicity, let's assume that each code statement gets executed as a single machine instruction and we measure the total cost of running the program in terms of total number of instructions executed. This gives us a simple formula to measure the running time of our program expressed below:

Running Time** = **Instructions executed in For loop** + **Instructions executed in While loop

Analyzing the outer for loop

Let's start with the outer for loop and see how many instructions would get executed for each iteration of the outer loop. For now, forget about the inner while loop; we'll analyze it separately from the outer for loop.

Code Statement	Cost
1. for (int i = 0; i < input.length; i++) {	For the first iteration, the initialization statement is executed but not for any subsequent iteration. Similarly, the increment statement is executed for every iteration except the first one. The inequality check is made for every iteration. Note that when the loop exits, the increment and inequality check are executed again. Imagine a 3 element array then the instructions executed are 1st iteration : initialization and inequality 2nd iteration : increment and inequality 3rd iteration : increment and inequality 4th iteration : increment and inequality So we can conclude that we'll execute ***2 x (n + 1)*** instructions when an array contains ***n*** elements. Cost per iteration is 2 instructions and we have *n + 1* iterations.
2. int key = input[i];	executed once per iteration
3. int j = i - 1;	executed once per iteration
lines 4 - 11 skipping inner while loop	
12. }	
Total cost per iteration	**4 instructions**

Total cost for n iterations	**[2 x (n + 1) + 2n] instructions**

Analyzing the inner while loop

Code Statement	Cost
4. while (j >= 0 && input[j] > key) {	Loop starts with j = 0. The two conditions for the loop get tested once per iteration of the loop. However, there will be one additional execution of these conditions, when the loop exits. So if the loop runs for 3 iterations then the loop conditions get tested 4 times. Cost = 2 *per iteration* and 2 for the final exit of loop. Note for simplicity, we'll assume that both the conditions get tested even if the first is false i.e. we don't short-circuit.
5. if (input[j] > key) {	executed once per iteration of loop
6. int tmp = input[j];	executed once per iteration of loop
7. input[j] = key;	executed once per iteration of loop
8. input[j + 1] = tmp;	executed once per iteration of loop
9. j--;	executed once per iteration of loop
10. }	--
11. }	--
Total cost per iteration of	= 7 instructions

inner loop	= 7 instructions
Total cost for n iterations of inner loop	= **(n x 7)** + **2** instructions

Note that with the above analysis, we got a handle on the number of instructions that get executed ***per iteration*** of each loop given an input size of *n*. We can compute the instruction count for all iterations and sum them up to get the execution count for the entire run. In later sections, we'll compute a generalization that'll allow us to count the number of executions for the entire algorithm with a single formula.

It is also important to be cognizant that we are considering each line of code as a machine instruction, which isn't true. A line of code may result in several individual machine instructions; however, this shouldn't affect our analysis as long as we maintain the same yardstick when performing analysis across algorithms. Think of it as expressing the speed of an athlete in km/hour, miles/hour, centimeters/minutes etc. Whichever metric you choose for your representation, it will not affect the speed with which the athlete runs.