COURSEWARE

Complexity

Contents

Overview

In programming, we use the Big O notation to describe the complexity of an algorithm.
Big O more specifically describes the worst-case scenario, and is used to describe the execution time of an algorithm.
With Big O notation we can express the runtime in terms of *how quickly it grows relative to the input, as the input gets larger*.

- *How quickly the runtime grows* simply refers to using the Big O notation to describe the complexity of our program.
- *Relative to the input* means that we can express how long our program will take to run, in its worst-case scenario, depending on the input.
  So if the program is linear the runtime would grow "on the order of the size of the input" or (O(n)) in Big O notation.
  This simply means that our worst-case scenario is the size of the input, or "n".
- *As the input gets larger* refers to the fact that we may have steps that take a long time when "n" is small, but when "n" gets larger the speed of the program would be faster compared to other implementations.

Tutorial

Constant Time

```
public void printFirstInt(int[] arrayOfInts) {
    System.out.println(arrayOfInts[0]);
}
```

The above example runs in constant time, or O(1) in Big O notation.
This means that no matter what, whether the array is size 1 or size 1000, the method will always only run one step as we are simply printing the first item of an array.

Linear Time

```
public void printAllInts(int[] arrayOfInts) {
    for(int i : arrayOfInts) {
        System.out.println(i);
    }
}
```

The above example runs in linear time, or O(n) in Big O notation.
This means that the runtime of the code is going to depend on the size of "n" or in our example `arrayOfInts`.
The larger `arrayOfInts` is, the longer the code will take to execute as we are running an operation on each item in the array.

Quadratic Time

```
public int[] bubbleSort(int[] arrayOfInts) {
    for(int i = 0; i < arrayOfInts.length; i++) {
        for(int j = 0; j < arrayOfInts.length; j++) {
            if(arrayOfInts[j] > arrayOfInts[i]) {
                int temp = arrayOfInts[i];
                arrayOfInts[i] = arrayOfInts[j];
                arrayOfInts[j] = temp;
            }
        }
    }
    return arrayOfInts;
}
```

The above example runs in quadratic time, or O(n^2) in Big O notation.
This means that because we have a nested for loop the array is running the outer loop "n" times and the inner loop is running "n" times for each iteration of the outer loop.
This means that if we have 10 integers in the array, we would be performing 100 iterations.
If we were to have a third nested loop the Big O notation would change to O(n^3).

Worst Case Scenario

As mentioned above when describing the complexity of our code using Big O notation we give the worst-case scenario.
This is because some programs could have multiple notations applied depending on where the element we want in the array is.

```
public boolean hasNeedle(int[] haystack, int needle) {
    for(int i = 0; i < haystack.length; i++) {
        if(haystack[i] == needle) {
            return true;
        }
    }
    return false;
}
```

In the above example, we are searching an array for a specific integer that will match the integer of `needle`.
The best-case scenario for this would be that the integers match at the first index of `haystack` which would make the Big O notation for this method O(1).
However there is a chance that the two won't match at all, or they won't match until the last index, which is our worst-case scenario and this would make the Big O notation O(n).
Therefore we describe the complexity of the code as O(n).

Exercises

There are no exercises for this module.