# Pre-Class Work 15: Runtime Analysis

## Introduction to Runtime Analysis [Background Reading]

# □ Motivation

There is often more than one way to solve a computer science problem. Naturally, this leads us to ask the question, which way is the most optimal? Well, this largely depends on what you define as "optimal".

Speaking in terms of programming, some may define the most optimal program as the one that's easiest to read or the one that handles the most input mismatch. Generally speaking though, when computer scientists talk about the most optimal program, they are referring to performance. For extremely large inputs, we want our program to run as fast as possible.

How then can we distinguish between two programs and determine which solution has better performance? Your first instinct might be to just run the program and time it on a stopwatch □. However, there are a couple of issues with this approach:

1. We introduce human error due to reaction time in starting and stopping a stopwatch.
2. Depending on what programming language the program was written in, that will have an effect on how fast the program will get executed.
3. Different computers have different hardware which can affect speed and memory capacity.

Simply put, there is too much variability with just running the program and timing it. As computer scientists, we want a reliable and portable way to evaluate an algorithm that is independent of the machine, language, coding tricks, etc. This leads us into the territory of **runtime analysis**.

# □ What is Runtime Analysis?

**Runtime Analysis** refers to the overall process of characterizing code with a **complexity class**. To understand what a **complexity class** is, it's important to know how to analyze your code and quantify the operations in a code snippet.

When we are performing **runtime analysis**, we are not referring to the actual execution time (milliseconds or seconds). Instead, we are referring to the total number of basic operations that the program executes as a function of the size of the input.

Basic operations are a loose definition but they refer to simple code constructs like assigning a

variable, checking a condition, arithmetic, accessing an array index, etc. All basic operations take some form of **constant** time, meaning these operations are "fast".

Consider the following snippet of code:

```
public void example1() {
    int x = 0;
    if (x < 5) {
        x = x + 1;
    }
}

public void example2(int n) {
    for (int i = 0; i < n; i++) {
        example1();
    }
}
```

If we take a look at `example1`, we see that there are 3 basic operations. We see `int x = 0` which initializes a variable. We see `x < 5` which is a conditional check. We also see `x = x + 1` which is variable reassignment. This all adds up to 3 basic operations.

If you want to get technical, `x = x + 1` could be considered 2 basic operations since it involves variable reassignment and arithmetic. However, needing to know the exact number of basic operations is seldom important when performing runtime analysis.

Let's now take a look at `example2`. Notice how initializing the for-loop also takes 2 basic operations! First, we have `int i = 0` which initializes a variable. Then we make a have a conditional check `i < n` . The most interesting thing to note is that in `example2`, it takes a parameter `n`. Let's see what happens as `n` increases.

- `n = 1`
    - First, we initialize the for-loop (2 operations).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `1 < 1` (1 operation).
    - The method finishes executing.
- `n = 2`
    - First, we initialize the for-loop (2 operations).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `1 < 2` (1 operation).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `2 < 2` (1 operation).

    - The method finishes executing.
- `n = 3`
    - First, we initialize the for-loop (2 operations).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `1 < 3` (1 operation).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `2 < 3` (1 operation).
    - Now we execute `example1()` (3 operations).
    - We updated `i` with `i++` (1 operation).
    - We check `i < n` which is `3 < 3` (1 operation).
    - The method finishes executing.

We can come up with a mathematical equation to determine how many basic operations get executed as `n` increase. We observe that `5 * n + 2` will give us the number of basic operations in `example2` for any value of `n`. Now, we don't really care about keeping track of these number of operations. What we are concerned with is how to characterize the code with a **complexity class**.

# □ Main Points

- **Runtime analysis** is the process of characterizing code with a **complexity class**. We can analyze runtimes by quantifying the number of *basic operations* code will execute as a function of the size of the input.
- *Basic operations* are simple code constructs that take constant time (they will take the same amount of time regardless of where they are in your code).
    - Some basic operations include assigning variables, checking for conditions, arithmetic operations, and array indexing.
- In **runtime analysis**, we don't need to count the e*xact* number of basic operations, but rather should understand how the number of basic operations will scale with input size.
- **Runtime analysis** can give us a reliable and portable way to evaluate an algorithm that is independent of the machine, language, coding tricks, etc.

## Complexity Classes [Background Reading]

**In all code snippets, you can assume that the method won't throw any exceptions / the input is valid.**

# Constant

If a program has a **constant time complexity**, it means that the program will always run in the same number of operations, regardless of the input size. Consider the following code snippet below:

```
public static int constantTimeMethod(int[] arr) {
    int result = arr[0];
    return result;
}
```

In this code, the method will always take the same amount of time to obtain the first element. Even if `arr` contained a million elements, the number of operations that this method takes will always remain the same.

Let's take a look at another example:

```
public static int constantTimeMethod(int[] arr) {
    int result = 0;
    for (int i = 0; i < 100; i++) {
        result += arr[0];
    }
    return result;
}
```

In this code, we are adding the first element in the array 100 times before returning it. It might be tempting to think that this method is something other than **constant time** but it actually is still **constant**! You might think well what if `arr.length` was 500? Again, regardless of the length of `arr`, this method will always take the same number of operations. Even if `arr.length` is 100 or even 1000000, the for-loop is not dependent on the input size so it will always take the same amount of time to execute, making the method **constant time**.

# Linear

If a program has a **linear time complexity**, it means that the program's number of operations grows **linearly** with the input size. **Linear** growth means that the number of operations increases by a fixed amount depending on the size of the input. Consider the following code snippet below:

```
public static int linearTimeMethod(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
    }
    return sum;
}
```

In this code, the method loops over the entire array and keeps a running sum of each element. Notice now that if `arr.length` varied, it would affect the number of times that the for-loop would execute. The more elements there are in the array, the number of operations **linearly** increases.

Let's take a look at another example:

```
public static int linearTimeMethod(int[] arr) {
    int sum = 0;
    int product = 1;
    for (int i = 0; i < arr.length; i++) [
        sum += arr[i];
        product *= arr[i];
    }

    return product - sum;
}
```

In this code, the method loops over the entire array and keeps a running sum while also keeping track of the running product. Notice how the body inside the for-loop are still **constant time** operations, meaning they don't depend on the input size. Thus, this method is still **linear** since as the number of elements in `arr` increase, the number of operations will still **linearly** scale alongside.

Let's take a look at one more example:

```
public static int linearTimeMethod(int[] arr) {
    int maxNum = 0;
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
        if (maxNum < arr[i]) {
            maxNum = arr[i];
        }
    }

    for (int i = 0; i < arr.length; i++) [
        sum += maxNum - arr[i]l
    }

    return sum;
}
```

In this code, the method loops over the entire array and keeps track of the maximum number it has seen. Then we have another for-loop which sums up the elements subtracted from the maximum number seen. Even though there are two for-loops in this example, if you added up the total

operations, you would notice that the number of operations would still increase **linearly** as you varied `arr.length`.

# Quadratic

If a program has a **quadratic time complexity**, it means that the program's number of operations grows directly proportional to the squared size of the input size $f(n) = n^2$. Consider the following code snippet below:

```
public static boolean quadraticTimeMethod(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            if (i != j && arr[i] == arr[j]) {
                return true;
            }
        }
    }

    return false;
}
```

In this code, the method is searching for two values that are the same. For each element in `arr`, we iterate over the entirety of `arr` again. The outer for-loop is looping for the length of the array and at each element, it creates an inner for for-loop which also loops for the length of array.

- When `i = 0`, we iterate `j` for `arr.length` times.
- When `i = 1`, we iterate `j` for `arr.length` times.
- When `i = 2`, we iterate `j` for `arr.length` times.
- ...and so forth.

Thus, the total number of operations that this code executes is somewhere within the ballpark of `arr.length * arr.length` which we can simply say is **quadratic time**.

# Big-O

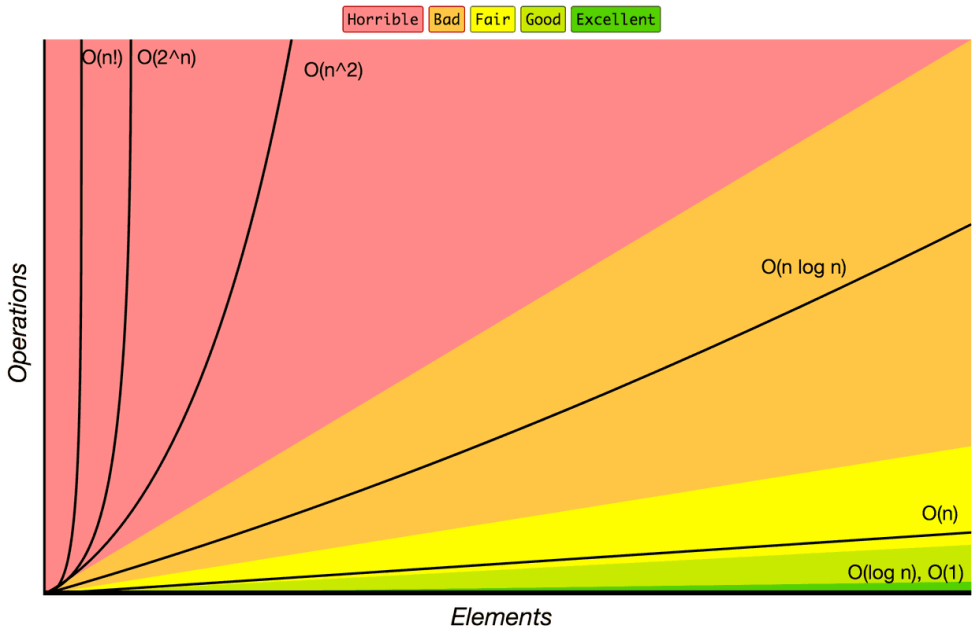
There is a notation called **Big-O** (pronounced "big oh") which computer scientists use to denote what **complexity class** a program falls into. Traditionally, Big-O uses the term *n* to denote input size. For example, if a program takes **constant time**, we would notate that as $O(1)$. If a program takes **linear time**, we would notate that as $O(n)$. Finally, if a program takes **quadratic time**, we would notate that as $O(n^2)$.

# Additional Complexity Classes

While we've introduced **constant**, **linear**, and **quadratic complexity**, they are not the only ones that you will come across. From fastest to slowest, some other **complexity classes** you'll see are as follows:

- Constant: $O(1)$
- Logarithmic: $O(log(n))$
- Linear: $O(n)$
- Log-linear: $O(nlog(n))$
- Quadratic: $O(n^2)$
- Cubic: $O(n^3)$
- Exponential: $O(2^n)$
- Factorial: $O(n!)$

It might be hard to visualize how these complexity class compares to one another so take a look at the graph below to see how the relationship between operations and elements differ between each complexity class:




As we can see in the graph above, with many elements and a few operations, $O(log(n))$ and $O(1)$ have a good and almost excellent runtime. With many elements and slightly more operations, $O(n)$ has a fair runtime. With a moderate number of elements and a moderate number of operations, $O(nlog(n))$ has a bad runtime.

With only some elements and many operations, $O(n^2)$ already has a bad runtime. And with even fewer elements and more operations, $O(2^n)$ has an even worse runtime, with the worst runtime being $O(n!)$!

For the purposes of this course, all you need to worry about is **constant**, **linear**, and **quadratic complexity**!

# Worst-Case Time Complexity

Now that you have an understanding of how to use **complexity classes** to characterize code, what should you be looking for when you are doing **runtime analysis**? When you are doing **runtime analysis**, you should be looking for the worst-case time complexity. In other words, what are the maximum number of operations that your algorithm will take on its most challenging input size.

Consider the following code snippet below:

```
public static void boolean find(int[] arr, int k) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

In this method, it returns true if `arr` contains the element `k` and false otherwise. What might be the best-case scenario for this method? The best-case would be if `k` is at the front of `arr`.

For example:

```
// Element is at the front
arr = [5, 7, 3, 2, 11, 4, 4, 6, 0, 16]
k = 5
```

Since the element we are looking for is directly at the start of `arr`, the for-loop never iterates through the entirety of `arr` so regardless of the size of `arr`, in the best-case, it would be considered **constant time**. However, would it be entirely accurate to say that this method is $O(1)$? Not really. We can't assume that all inputs given to us will always be the best-case inputs.

Instead, we as computer scientists like to think of the worst-case. What would be the worst-case scenario for the method? The worst-case would be if `k` is at the end of `arr` or if `k` doesn't exist in `arr`.

For example:

```
// Element exists
arr = [5, 7, 3, 2, 11, 4, 4, 6, 0, 16]
k = 16
```

```
// Element doesn't exist
```

```
arr = [5, 7, 3, 2, 11, 4, 4, 6, 0, 16]
k = 8
```

In this example, the element we are looking for is either directly at the end of `arr` or the element doesn't exist at all in `arr`. If you varied the size of `arr` and always put `k` at the end of `arr` or set `k` to be an element not in `arr`, then the for-loop will always need to iterate through `arr` to find that element.

Thus, in the worst-case, this method would take **linear time** to run.

You might wonder why we care about worst-case over best-case. When you write code, you cannot always rely on the inputs being the best-case inputs. As observed above, the element could be anywhere in the array! It could be in the front, the middle, the end, or maybe not even exist in the array at all! However, we should always anticipate the worst-case inputs being given to our program.

The reason why worst-case inputs are important when doing **runtime analysis** is because it guarantees that the program will finish. Therefore, it is a more accurate **runtime analysis** to say that the method is $O(n)$ since it guarantees that the program will take **linear time** in the worst-case.

# □ Main Points

- Example **time complexities**:
  - A program with **constant time complexity**, $O(1)$, will always run in the same number of operations, regardless of its input size.
  - A program with **linear time complexity**, $O(n)$, y has the number of operations grow linearly with the input size.
  - A program with **quadratic time complexity**, $O(n^2)$, usually has nested loops and has the number of operations grow proportionally to the square of the input size.
  - Some other time complexities include **logarithmic** $(O(log(n)))$, **log-linear** $(O(nlog(n)))$, **cubic** $(O(n^3))$, **exponential** $(O(2^n))$, and **factorial** $(O(n!))$
- We use **big-O notation** to standardize the way we describe the upper bound, or **worst-case scenario**, of the complexity of an algorithm in terms of input size.
- We focus on **worst-case scenarios** so we can understand how a program will behave under challenging input conditions and because it guarantees that the program will finish in a certain amount of time.
- When we design algorithms, we want to design them with the lowest complexity class we can so we can create more efficient and scalable programs, especially when we're working with a lot of data or our program needs to consume a lot of resources.

## Runtime Analysis for Worst-Case Complexity [Discussion Questions]

# Runtime Analysis Practice

Apply your understanding of **runtime analysis** to identify the worst-case time complexity. For each question, let `n` be the length of the input `arr`. Answer each question in terms of `n`.

**Question 1**

Suppose you are given the following code snippet:

```
public static int mystery1(int[] arr) {
    if (arr.length > 0) {
        return arr.length * 2;
    } else {
        return arr.length * -1;
    }
}
```

Apply your knowledge of runtime analysis to assess worst-case time complexity.

- $O(1)$
- $O(n)$
- $O(n^2)$

**Question 2**

Suppose you are given the following code snippet:

```
public static void mystery2(int[] arr) {
    int result = 0;
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            result += arr[i] * arr[j];
        }
    }
    return result;
}
```

Apply your knowledge of runtime analysis to assess worst-case time complexity.

- $O(1)$
- $O(n)$
- $O(n^2)$

**Question 3**

Suppose you are given the following code snippet:

```
public static boolean mystery3(int[] arr, int k) {
    for (int i = arr.length - 1; i >= 0; i--) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

Apply your knowledge of runtime analysis to assess worst-case time complexity.

- $O(1)$
- $O(n)$
- $O(n^2)$

**Question 4**

Suppose you are given the following code snippet:

```
public static int mystery4(int[] arr, int k) {
    if (arr.length != 0) {
        int sum = 0;
        for (int i = 0; i < 10000; i++) {
            sum += (arr[0] * k);
        }
        return sum;
    } else {
        return -1;
    }
```

```
}
```

Apply your knowledge of runtime analysis to assess worst-case time complexity.

- $O(1)$
- $O(n)$
- $O(n^2)$

**Question 5**

Suppose you are given the following code snippet:

```
public static void mystery5(int[] arr) {
    int result = 0;
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j+=2) {
            if (i != j) {
                for (int k = 1; k < 5; k++) {
                    result += arr[i] * arr[j];
                }
            }
        }
    }
    return result;
}
```

Apply your knowledge of runtime analysis to assess worst-case time complexity.

- $O(1)$
- $O(n)$
- $O(n^2)$

## Is Distinct [Programming Question]

Now that you have familiarity with applying runtime analysis to identify worst-case complexity, let's apply this knowledge to write a more optimized program.

Write a method `isDistinct()` that takes an integer array `arr` and returns `true` if `arr` is **distinct**, and `false` otherwise. An integer array is defined as **distinct** if all elements are unique.

For example, if `arr` contained the elements `[1, 2, 3]`, then `isDistinct(arr)` should return true. However, if `arr` contained the elements `[1, 2, 1]`, then `isDistincts(arr)` should return false. You may assume `arr` is non-null.

Consider the working solution below:

```
public boolean isDistinct(int[] arr) {
    // The solution below takes O(N^2) time
    // which exceeds the time limit on the given tests.
    // Try hitting the "Mark" button to see what happens.
    for (int i = 0; i < arr.length; i++) {
        for (int j = i + 1; j < arr.length; j++) {
            if (arr[i] == arr[j]) {
                return false;
            }
        }
    }
    return true;
}
```

You have been supplied with a working solution but as it stands, the current solution exceeds the time limit on the given tests.

Try inputting the solution into the scaffold and hitting the "Mark" button to see what tests pass and which ones fail (you should see the test cases only pass for 2/4 of tests).

**Task:** Your task is to write a more optimized solution, one that runs in a faster time complexity than the current worst-case time complexity.

***HINT:*** What data structure helps with eliminating duplicates?