Learning Objectives

- **Compare Linear and Binary search**
- **Analyze coding problems in five steps**
- **Learn to write more efficient code**
- **Use Big O analysis to determine algorithm runtime**

Algorithm Analysis

Introduction to Algorithm Analysis

After learning about data structures and their operations, we'll now shift our focus towards **Algorithm Analysis**. The goal of algorithm analysis is to predict the resources that the algorithm requires. It can be both time and space.

Often in computer science, there are many ways to solve a problem. All these solutions, or algorithms, can vary in their efficiency. Some algorithms might run faster, while others might use less memory. To make an informed choice about which algorithm to use, we need to understand how to analyze and compare algorithms based on their efficiency. This is the essence of algorithm analysis.

In the process of algorithm analysis, we are mostly interested in the efficiency with respect to time and space:

1. **Time Complexity**: This refers to the total count of operations an algorithm will perform in its lifetime. As the size of the input data increases, the running time of the algorithm also grows. We express time complexity as a function of the input size (n), often using **Big O notation**.
2. **Space Complexity**: This is the amount of memory space the algorithm needs to run to completion. Just like time complexity, space complexity is also expressed as a function of the input size (n), often using Big O notation.

In this part of the course, we will learn about the concept of **Big O notation**, which is a theoretical measure of the execution of an algorithm or the amount of space it requires. It provides us with an upper bound of the complexity in the worst-case scenario, helping us to understand the worst-case scenario for an algorithm.

We will go through different complexities like $O(1)$, $O(n)$, $O(logn)$, $O(n^2)$, etc., and understand their impact on the running time and space of an algorithm. We will also learn how to calculate the time and space complexity of an algorithm and compare different algorithms.

Big O

Big O is probably going to be one of the most important topics in your computer science career. Previously, we talked about how we wanted to optimize our code. By *optimize*, we mean how we make our code more efficient (or run faster). Turns out, there is a metric we use to describe the efficiency of algorithms and that is ***Big O***. Understanding this will help better determine when your algorithm is more efficient, running faster, taking less space, etc.

definition

Big O notation is a way of describing how the performance of an algorithm or program changes as the size of the input grows. It is a mathematical notation used to represent the time complexity or space complexity of an algorithm.

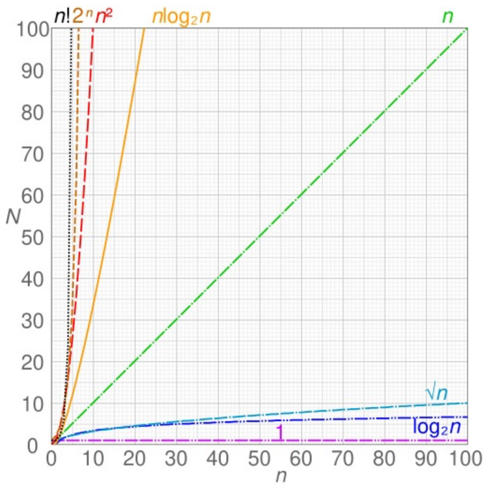
Time Complexity

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm. It's not about the time in seconds or milliseconds it takes for your program to run. Its primary focus is the number of operations relative to the size of the input. Suppose each elementary operation takes the same amount of time to perform. If each operation takes the same amount of time to perform, then we can think of it as the **more operations** performed by an algorithm, the **longer** that algorithm took to get to completion. We can think of the number of operations and the completion time as being related by a constant factor. In short, if one increases the other one does as well.

An algorithm's running time may vary with different inputs of the same size. The most common way to tackle this dilemma is to instead focus on the **worst case** scenario. Given an example, what's the longest it can take to perform all the operations it needs to in relation to its size. Sometimes, you will encounter cases where you are looking for the average case complexity, but that will be specified.

Time complexity can be measured by the number of elementary operation performed. If told to look for time complexity, we are looking for something in relation to the worst case scenario unless told otherwise. When we evaluate complexity of algorithms, we count the number of computation steps, or of memory locations.

The time complexity is generally expressed as a function of the size of the input. More specifically, it is based on the behavior of the complexity when the input size increases. The time complexity is commonly expressed using *Big O* notation, typically $O(n)$, $O(n\ log\ n)$, $O(2^n)$, etc., where n is the size of the input. The image below is a representation of some of the common O notations.



Common runtimes

- $O(log\ n)$, $O(n\ log\ n)$, $O(n)$, $O(n^2)$, $O(2^n)$
- $O(1)$

Let's start focusing on the purple line at the bottom. $O(1)$ is **constant** time which occurs when the algorithm does not depend on the size of the input. Regardless of how much we change our array, the function will take the same amount of operations to get the answer. For example, imagine we are told to write a function that returns the first element of the array. Regardless of how large the array is, it will only take *one* step to do.

$O(n)$ or **linear** time is when the algorithm is proportional to the size of the input. In the image above, it is represented by a green line. We can see that the size of the input (**n** on the x-axis) is directly correlated with the number of operations (**N** on the y-axis). In the graph, everything below the green line is considered to be more optimal. Another way to think of it is as the size of the array increases there is less change in terms of how many operation will be taken. Ideally, we want something below the linear time (green line) and avoid any times above it since they would take too long (making them much less efficient).

Searching

Searching is a fundamental operation in computer science. Given an array of elements, a common task might be to find if a particular value exists within this array. We have different strategies(algorithms) to perform this operation each strategy with different time complexities, and the two most common are **Linear Search** and **Binary Search**. Each of these techniques uses a different approach and has different performance characteristics.

Linear Search

The simplest approach to searching an array is with a linear search. As the name suggests, this algorithm works by checking each element in the array one-by-one, starting at the first element and continuing until it finds a match or until it has checked all elements. This method is straightforward and doesn't require the array to be sorted.

Here's a simple Java implementation of a linear search. The `target` variable represents the search target, and the `main` method prints the index of the target or that it was not found. Copy and paste the following in the top left panel:

```java
public class LinearSearch {

    public static int linearSearch(int[] array, int target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i] == target) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        int target = 3;

        int result = linearSearch(array, target);

        if (result == -1)
            System.out.println("Element not found in the
        array");
        else
            System.out.println("Element found at index: " +
        result);
    }
}
```

In this code, the `linearSearch` function iterates through each element in the array. If it finds the target element, it returns the index of the element. If it does not find the target, it returns `-1`.

Binary Search

Binary search is a more efficient algorithm than linear search, but it requires the array to be sorted first. Binary search works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the algorithm continues the search on the lower half. Otherwise, it continues on the upper half.

Here’s a simple Java implementation of a binary search. Copy and paste the following in the bottom left panel:

```java
public class BinarySearch {

    public static int binarySearch(int[] array, int target) {
        int left = 0;
        int right = array.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Check if target is present at the mid
            if (array[mid] == target)
                return mid;

            // If target greater, ignore left half
            if (array[mid] < target)
                left = mid + 1;

            // If target is smaller, ignore right half
            else
                right = mid - 1;
        }

        // if we reach here, then element was not present
        return -1;
    }

    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};
        int target = 3;

        int result = binarySearch(array, target);

        if (result == -1)
            System.out.println("Element not found in the
        array");
        else
            System.out.println("Element found at index: " +
        result);
    }
}
```

In this code, the `binarySearch` function repeatedly checks the middle element of the array (or sub-array) until it finds the target or the search interval is empty. The beauty of binary search lies in its ability to cut the search space in half with each iteration.

However, keep in mind that binary search only works if the array is sorted. If it's not, you'll need to sort it first or use a different search algorithm.

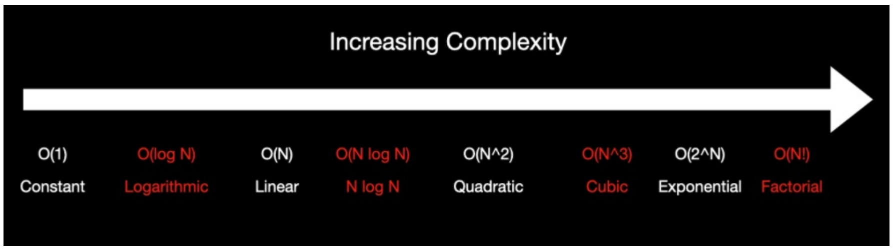
Through these two search methods, we can begin to see how different approaches to the same problem can greatly affect the efficiency of our programs. We'll further explore these topics when we dive deeper into time complexity and Big O notation.

Linear vs. Binary Search

Comparing Linear and Binary Searches

Linear Search
In the worst case scenario, we have to check every single element in our array (see the animation on the top-left). The **Big O** for Linear Search is $O(n)$, linear time. The algorithm complexity increases as the size of our array increases. The algorithm takes an additional step for each additional element added to our array.

Binary Search
In the worst case scenario, we are cutting our list in half with each iteration (see the animation on the bottom-left). The "halving" with each iteration can be expressed as the mathematical operation **logarithm**. The **Big O** for Binary Search is $log\ n + 1$. We can ignore the $+1$ because it is negligible in the grand scheme of time complexity. Binary Search is so efficient because doubling the length of our array means the algorithm would only do one additional step.



When dealing with *Big O*, we drop the non-dominant terms (yes, including constants). *Big O* notation only describes the growth rate of algorithms in terms of mathematical function, rather than the actual running time of algorithms.

- $O(n^2 + n)$ is the same as $O(n^2)$
- $O(n + log\ n)$ is the same as $O(n)$
- $O(log\ n + 323123)$ is the same $O(log\ n)$

Using the image above, in terms of *Big O*, it is more efficient for us to use Binary Search instead of Linear Search because it is more efficient as our input size becomes larger. Remember that $log\ n$ is smaller than n. While linear search is straightforward and works well on small, unsorted arrays, binary search is far more efficient for larger, sorted arrays. However, binary search requires the array to be sorted, which might be a limitation if you frequently add new elements to the array. Linear search does not have this limitation.

Problem Solving

When tackling a problem in computer science and for the rest of this course, always do these **five steps:**

1. **Read/Listen**
2. **Think of Examples**
3. **BRUTE FORCE**
4. **Optimize**
5. **Test**

We are going to employ these five steps with the following prompt prompt below.

Prompt:

Create a function given a sorted array of `n` strings, that will return the location of a given element.

Read/Listen

This goes with paying close attention to the problem we are trying to solve.

- What is our goal? What are we looking for? What do we want to make happen?
- What do we start with? What information is given? What can we extract from what we are given?

Example: We are given an array of strings, and we want to return the index at which the element of the search target.

Think of Examples

This goes with picturing how our code works. It helps us visualize what we are about to do. Think of special cases and normal examples.

- Normal case: search for `"cat"` given the following `["bat", "cat", "dog", "eagle", "fish"]`
- Special cases: search for `"cat"` given `[]` or `["ape", "donkey", "monkey", "raccoon", "snake"]`

BRUTE FORCE

This goes with thinking of the easiest or most straightforward way to obtain the solution. Let’s start by using a linear search, since this is the easiest and most straightforward way of search an array.

```java
public class ProblemSolving {

    public static int search(String[] array, String target) {
        for (int i = 0; i < array.length; i++) {
            if (array[i].equals(target)) {
                return i;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        String[] array = {"ape", "bat", "cat", "dog", "eagle"};
        String target = "dog";

        int result = search(array, target);

        if (result == -1)
            System.out.println("Element not found in the
        array");
        else
            System.out.println("Element found at index: " +
        result);
    }
}
```

Optimize

The fourth thing is to **optimize**. In other words, pay attention to how to make your code better. What does a better code look like? For that, let's compare linear search and binary search.

Prompt:

Create a function given a sorted array of `n` strings, that will return the location of a given element.

Linear vs. Binary Search

The two approaches for searching for an element that we have covered are linear (or sequential) search and binary (or half-interval) search. Let's go over the differences.

How they start:

- Linear : start from the beginning of our array
- Binary : start from the middle of our array

Best case:

- Linear : the first element is what we are looking for
- Binary : the middle element is what we are looking for

Worst case:

- Linear : last element of the array
- Binary : the value is the first or last element of the array

What Do We Mean by "Best" and "Worst" Case?

We are going to think of **best case** as the situation where our program will take the least number of steps to return the result. On the other hand, the **worst case** situation will take the most number of steps to return the result.

Looking above, the worst case for binary and the best case for linear are the same. Does that mean that linear is a better search function? No, it does not. As a matter of fact, more often than not, binary search will be faster than linear search for a given large array. We will not know in advance where the search target is located in the array. We need to choose the code that on average will be faster, regardless of what the array actually looks like.

Given the discussion above, we know on average that a binary search is more efficient than a linear search. As such, let’s optimize and update our `search` method to use a binary search algorithm.

```java
public static int search(String[] array, String target) {
    int left = 0;
    int right = array.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        int result = target.compareTo(array[mid]);

        if (result == 0)
            return mid;
        if (result > 0)
            left = mid + 1;
        else
            right = mid - 1;
    }
    return -1;
}
```

Test

The fifth and final step is to test your work. This does not mean that you write up a formal unit test (though thorough testing is never a bad idea), but you should at least give your code a few sample inputs to which you already know the answer. The previous search target was `"dog"`, which we know is an element of the test array. Now, let's try a test case where we know the algorithm should not find the search target.

```java
String target = "fish";
```

Run the code again. You should see a message stating the element is not found in the array.

Back in the **Think of Examples** step, we talked about special cases like an empty array. Let's update our code to see how it handles an empty array.

```java
String[] array = {};
String target = "fish";
```

Just as before, we should see the message about the element not being in the list.

Finally, let's consider a few more edge cases. What happens if we search for `"Dog"` instead of using a lowercase letter? Update the values for `array` and `target` so they match the example below.

```java
String[] array = {"ape", "bat", "cat", "dog", "eagle"};
String target = "Dog";
```

Because Java is case sensitive, it should print the message that `"Dog"` is not in the array.

Ignoring capitalization is a common search practice, and we want to do the same with our search algorithm. Find the line of code that calculates the `result` variable. We want the string comparison to force the search target and element to be uppercase.

```java
int result =
        target.compareToIgnoreCase(array[mid].toUpperCase());
```

Java provides us with the `compareToIgnoreCase()` method that compares the values of two strings while ignoring case. As long as the spelling matches, the algorithm should return the index of search target.

challenge

Try these variations:

Update `target` to have the different spellings below. The search algorithm should still return the index `3` despite the odd capitalizations.

- `"DOG"`, `"dOG"`, and `"dOg"`

Formative Assessment 1

Formative Assessment 2