**O Notation**

Big O Notation is a tool used to describe the time complexity of algorithms. It shows us **how the time it takes to run your function grows as your input grows**.

**Why is it important?**

- It is important to have a precise vocabulary when discussing code performance.

- It is useful when discussing trade-offs between different algorithms

- When our code slows down, or crashes, it's important to identify parts of the code that are inefficient that can help us point to the problem.

**- And very important, this topic comes up in technical job interviews!**

**Different O Notations:**

1. **O(1)**: A function *f*(*n*) is in this class if there is a constant K>0 such that f(n)≤K for all n≥1. An algorithm whose worst-case complexity function is in this class is said to run in ***constant time***.

**In other words, the input size does not affect the time it takes to run the function.**

Example of a function that is O(1)

A math equations on a white background

Description automatically generated

No matter what the value of num is, it would not make the function run longer. The run time is constant.

A graph with numbers and a red line

Description automatically generated

2. **O(*n*)**: Algorithms in this class run in ***linear time***. Any increase in the size of the problem results in a proportionate increase in the running time of the algorithm. This complexity is characteristic of algorithms like *sequential search* that make a **single pass** over their input.

Example of a function that is O(n):

A screenshot of a computer program

Description automatically generated

As the size of the input increases, the time it takes to run the function increases linearly.

A graph with a red line

Description automatically generated

3. **O(log *n*)**: Algorithms in this class run in ***logarithmic time***. Because log *n* grows much slower than *n*, a huge increase in the size of the problem results in a small increase in the running time of the algorithm. This complexity is characteristic of search problems that eliminate half of the search space with each operation. The *binary search* algorithm is in this class.

Example of a function that is O(log n)

A screenshot of a computer code

Description automatically generated

A graph with a red line

Description automatically generated

4. **O(n^2)**: This class is called ***quadratic time***. This performance is characteristic of algorithms that make multiple passes over the input data using two nested loops. An increase in the size of the problem causes a much greater increase in the running time of the algorithm. The worst-case complexity functions of *bubble sort*, *selection sort*, *insertion sort*, and *Quicksort* all lie in this

class.

Example of a function that is O(n^2):

A screenshot of a computer program

Description automatically generated

A graph with a red line

Description automatically generated

5. **O(*n* log *n*)**: This class is called **“*n* log *n*” *time***. An increase in the size of the problem results in a slightly greater increase in the running time of the algorithm. The average case complexity of *Quicksort* lies in this class. Two well-known sorting algorithms, *Mergesort* and *Heapsort*, have worst case complexity functions that also lie in this class. We will discuss the *Heapsort*

algorithm in Chapter 22.