Complexity and Performance

Typically answering about the performance of the code requires the answer to be along several axis.

Performance is measured along resource consumption and code consumes a variety of resources.

Improving code performance beyond a certain point involves tradeoffs meaning that you have to consume more of one resource to help consume less of another resource.

Measures of performance

Time

Space (Memory or disk space)

Network (How much data it transfers/network bandwidth)

Time:

The amount of processing or number of operations code has to perform to accomplish its objective

Space:

Memory needed by code to store information at run time as well as disk space needed for persistent storage

Network:

The bandwidth code uses to pass information to client or other machines

**Efficient code uses fewer resources along all these axes**

**Code can also be more performant when it uses the resources we have in plenty rather than those we lack**.

Is it a time critical system or a space critical system.

**WHAT IS COMPLEXITY?**

Complexity is a measure of how resource requirements change as **the size of the problem gets larger.**

Complexity affects performance.

The higher the complexity, the lower the performance.

As the size of the input to our code becomes larger, how much worse is the performance.

Very high complexity means that as the size of the input grows, the performance degrades rapidly.

Types of operations:

Arithmetic Operations

Read Operations

Assignment

Write operations

Test Operations

When we talk about the complexity, we do not worry about the exact number of operations. But we are concerned by the number of changes based on the input size. How performance changes based on input size. We also focus on the worst case performance. What is the maximum number of operations that might have to be performed based on the input.

**How to measure complexity?**

Big O notation.

Expresses the complexity of an algorithm.

N is the size of the input. The complexity of an algorithm is O(N) if it increases linearly when N increases.

O(N2) = The algorithm increases in complexity, quadratically. If N becomes 2N, complexity becomes 4N2.

**IMPORTANT:**

Lower Order Terms and Constants do not matter while expressing complexity. The assumption is that N is very large compared to constant time operations.

**Time Complexity**

It is not measured in time, as if you ran the same code in a computer that is twice as fast, it would complete twice as fast too. This doesn’t mean the code is better but means the computer is better.

**Space Complexity**

While a code runs fast, may take a lot of space. If preserving memory space is the priority, that code is better.

**Big O (Worse Case)**

Best Case for running a code is Omega

Average Case for running a code is Theta

Worse Case for running a code is O

**Linear Time**

O(n)

Graph of O(n) will always be Number of Operations against the Size of Input

**Drop Constants**

Look at the following code:

def print\_items(n):

for i in range(n):

print(i)

for j in range(n):

print(j)

In total for print\_items(10) you will have 20 operations occurring. We can write the complexity as O(2n) but we can instead drop the constants and write it as O(n)

**Quadratic Time**

O(n2)

def print\_items(n):

    for i in range(n):

        for j in range(n):

            print(i,j)

print\_items(10)

The above will result in 100 operations for 10 items. Hence it is O(n2)

def print\_items\_2(n):

    l = 0

    for i in range(n):

        for j in range(n):

            for k in range(n):

                l += 1

                print(i,j,k)

    print(l)

print\_items\_2(10)

The above will result in 1000 operations for 10 items. It is O(n3) but we call it as O(n2)

**Drop Non Dominants**

def print\_items\_3(n):

    for i in range(n):

        for j in range(n):

            print(i,j)

    for k in range(n):

        print(k)

In the above code, we run 100 operations for 10 items in the first 2 loops, but the final loop only runs 10 times. Hence the dominant term is 100 as the n2 term is always dominant. Hence the other term (10) is dropped out.

**O(1) functions**

Constant time for all operations

**O(logn) functions**

Finding an element in a sorted list

1. First we split the list into 2
2. Check in the two halves
3. Then find which half has it
4. Then repeat step 2 and 3 in second half.

For a list of size 8, we had only 3 steps in total to do. This is equal to log2(8) hence the complexity is O(logn)

The real power of such algorithms is when N is extremely large. Log21073741824 requires only 31 steps to find the item

On the complexity graph, the graph of O(logn) is almost as flat as O(1).

A picture containing diagram

Description automatically generated

Another possibility is O(nlogn) which is some sorting algorithms. It lies between n2 and n. This is also the most efficient you can make a sorting algorithm (Merge Sort, Quick Sort).

**Different Terms for Inputs**

If a function has different inputs **a** and **b**, then we cannot say that the complexity is O(n) as a and b are different. Hence the complexity is O(a+b).

If we has a nested for loop, where the input for each for loop is **a** and **b** respectively, the complexity is going to be O(a\*b)

**Big O of Lists**

Check BigOLists.py

**Wrap Up**

When n=100

O(1) = 1

O(logn) = 7

O(n) = 100

O(n2) = 10000

When n = 1000

O(1) = 1

O(logn) = 10

O(n) = 1000

O(n2) = 1000000

**Terminologies:**

**O(n2) =** Loop within a Loop

O(n) = Proportional

O(logn) = Divide and Conquer

O(1) = Constant

**BigOcheatsheet.com**