Time Complexity

What you will learn in this lecture?

- Algorithm Analysis
- Type of Analysis
- Big O Notation
- Determining Time Complexities Theoretically
- Time complexity of some common algorithms

Introduction

An important question while programming is: How efficient is an algorithm or piece of code?

Efficiency covers lots of resources, including:

1. CPU (time) usage
2. memory usage
3. disk usage
4. network usage

All are important but we are mostly concerned about CPU time. Be careful to differentiate between:

1. **Performance:** how much time/memory/disk/etc. is actually used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.
2. **Complexity:** how do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger. Complexity affects performance but not vice-versa. The time required by a function/method is proportional to the number of "basic operations" that it performs.

Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system. changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Analysis

To analyze a given algorithm, we need to know, with which inputs the algorithm takes less time (i.e. the algorithm performs well) and with which inputs the algorithm takes a long time.

Three types of analysis are generally performed:

• **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.

• **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.

• **Average case:** The average case gives an idea about the average running time of the given algorithm.

There are two main complexity measures of the efficiency of an algorithm:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Big-O notation

We can express algorithmic complexity using the **big-O** notation. For a problem of size N:

- A constant-time function/method is *"order 1"*: **O(1)**
- A linear-time function/method is *"order N"*: **O(N)**
- A quadratic-time function/method is *"order N squared"*: **O(N^2)**

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be **O(g)** (read big-oh of g), if there is a constant **c** and a natural n_0 such that **f (n) ≤ cg(n)** for all **n > n_0.**

Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).

Examples:

- **$5n^2 + 15 = O(n^2)$**, since **$5n^2 + 15 \leq 6n^2$**, for all **n > 4**.
- **5n2 + 15 = $O(n^3)$**, since **$5n^2 + 15 \leq n^3$**, for all **n > 6**.
- **O(1)** denotes a constant.

Although we can include constants within the big-O notation, there is no reason to do that. Thus, we can write **O(5n + 4) = O(n)**.

Note: The **big-O** expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time function/method, which will be faster than a quadratic-time function/method).

Determining Time Complexities Theoretically

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
...
statement k;
```

The total time is found by adding the times for all statements:

```
totalTime = time(statement1) + time(statement2) +..+
time(statementk)
```

2. `if-else` statements

```
if (condition):
    #sequence of statements 1
else:
    #sequence of statements 2
```

Here, either **sequence 1** will execute, or **sequence 2** will execute. Therefore, the worst-case time is the slowest of the two possibilities:

```
max(time(sequence 1), time(sequence 2))
```

For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be **O(N)**.

3. `for` loops

```
for i in range N:
    #sequence of
statements
```

Here, the loop executes **N** times, so the sequence of statements also executes **N** times. Now, assume that all the statements are of the order of **O(1)**, then the total time for the **for** loop is **N * O(1)**, which is **O(N)** overall.

4. Nested loops

```
for i in range N:
    for i in range
M:
        #statements
```

The outer loop executes **N** times. Every time the outer loop executes, the inner loop executes **M** times. As a result, the statements in the inner loop execute a total of **N * M** times. Assuming the complexity of the statement inside the inner loop to be **O(1)**, the overall complexity will be **O(N * M).**

Sample Problem:

What will be the Time Complexity of following while loop in terms of 'N' ?

```
while N>0:
    N =
N//8
```

We can write the iterations as:

Iteration Number	Value of N
1	N
2	N//8
3	N//64
...	...
k	$N//8^k$

We know, that in the last i.e. the **k^{th}** iteration, the value of **N** would become **1**, thus, we can write:

```
N//8^k = 1
=> N = 8^k
=> log(N) = log(8^k)
=> k*log(8) =
log(N)
=> k =
log(N)/log(8)
=> k = log₈(N)
```

Now, clearly the number of iterations in this example is coming out to be of the order of $\log_8(N)$. Thus, the time complexity of the above while loop will be $O(\log_8(N))$.

Qualitatively, we can say that after every iteration, we divide the given number by 8, and we go on dividing like that, till the number remains greater than 0. This gives the number of iterations as $O(\log_8(N))$.

Time Complexity Analysis of Some Common Algorithms

Linear Search

Linear Search time complexity analysis is done below-

Best case- In the best possible case:

- The element being searched will be found in the first position.

- In this case, the search terminates in success with just one comparison.
- Thus in the best case, the linear search algorithm takes **O(1)** operations.

Worst Case- In the worst possible case:

- The element being searched may be present in the last position or may not present in the array at all.
- In the former case, the search terminates in success with **N** comparisons.
- In the latter case, the search terminates in failure with **N** comparisons.
- Thus in the worst case, the linear search algorithm takes **O(N)** operations.

Binary Search

Binary Search time complexity analysis is done below-

- In each iteration or each recursive call, the search gets reduced to half of the array.
- So for **N** elements in the array, there are **$\log_2 N$** iterations or recursive calls.

Thus, we have-

- Time Complexity of the Binary Search Algorithm is **$O(\log_2 N)$**.
- Here, **N** is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Big-O Notation Practice Examples

Example-1 Find upper bound for **f(n) = 3n + 8**

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$

$\therefore 3n + 8 = O(n)$ with $c = 4$ and $n_0 = 8$

Example-2 Find upper bound for **$f(n) = n^2 + 1$**

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for **$f(n) = n^4 + 100n^2 + 50$**

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$