Fundamental Data Structures and Algorithms 02 - Computational Complexity and Big O Notation

Unit 1: Foundations (continued)

1.2 Computational Complexity

In this module, we are interested in the design of good data structures and algorithms. As such we must have precise ways of analyzing them.

The primary analysis tool we will use in this module involves characterizing the running times of algorithms and data structure operations, with space usage also being of interest. Running time is a natural measure of "goodness," since time is a precious resource—computer solutions should run as fast as possible. In general, the running time of an algorithm or data structure operation increases with the input size, although it may also vary for different inputs of the same size. Also, the running time is affected by the hardware environment (e.g., the processor, clock rate, memory, disk) and software environment (e.g., the operating system, programming language) in which the algorithm is implemented and executed. All other factors being equal, the running time of the same algorithm on the same input data will be smaller if the computer has, say, a much faster processor or if the implementation is done in a program compiled into native machine code instead of an interpreted implementation.

Much of the discussion that follows will be framed by the following three guiding principles. The rational and importance of these principles should become clearer as we proceed. These principles are as follows:

- Worst case analysis. Make no assumptions on the input data.
- Ignore or suppress constant factors and lower order terms. At large inputs higher order terms dominate.
- Focus on problems with large input sizes.

Worst case analysis is useful because it gives us a tight upper bound that our algorithm is guaranteed not to exceed. Ignoring small constant factors, and lower order terms is really just about ignoring the things that, at large values of the input size, n, do not contribute, in a large degree, to the overall run time. Not only does it make our work mathematically easier, it also allows us to focus on the things that are having the most impact on performance.

1.2.1 Experimental Analysis using Searching Algorithms

We begin this chapter by discussing tools for performing experimental analysis, yet also limitations to the use of experiments as a primary means for evaluating algorithm efficiency.

The sorting problem is a problem that frequently arises in practice and provides fertile ground for introducing many concepts. Previously, we mentioned how dictionaries have to be sorted for it to be meaningful. The most obvious way is to sort the words alphabetically. It provides us a systemic way of searching for a particular word quickly as compared to say sorting them by the length of the word. While it is still possible to search words according to their lengths, it is becomes much more challenging to do so.

As many programs use sorting as an intermediate step, it is considered a fundamental concept in computer science. This has also led to many sorting algorithms being developed. Therefore, it is important for us to be able to analyze algorithms to determine their *computational complexity*, i.e., the amount of resources, in particular time and memory, required to execute them.

Consider the following problem: Given an array of n consecutive elements, write a function to search for an element x in that array.

A simple solution could follow the following algorithm:

1. Starting from the beginning of the array, check if the element in the array matches with x
2. If the element matches with x, return the index of the element in the array + 1 (this gives the number of steps taken to reach the element)
3. Otherwise, repeat steps 1 to 2 for the next element in the array
4. If x does not match with any of the elements in the array, return '-1'

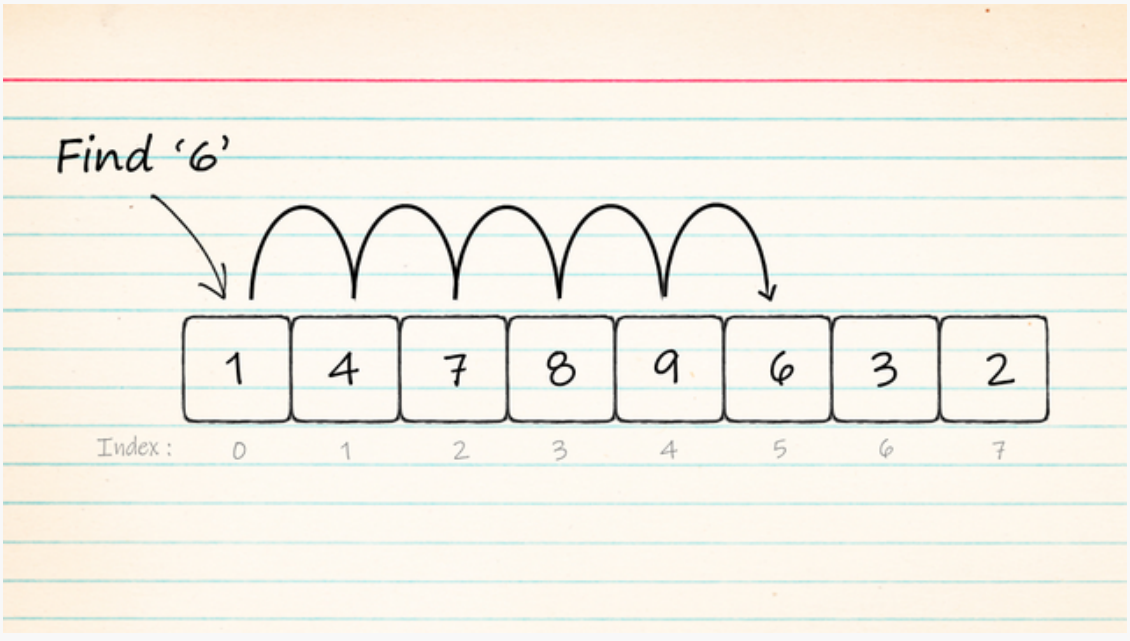

Fig 1.6. Visualization example of a basic search algorithm on a list or array structure.

Fig 1.6 shows how the algorithm works if we need to find the element 6 in an array consisting of an array with the elements $\{1, 4, 7, 8, 9, 6, 3, 2\}$. It starts from index 0 and works it way through the array one index at a time until it find the element 6. It will then return the number of steps taken to reach that element.

Resolving the algorithm in Python will yield the following result:

```
# Search Function 1
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i + 1
    return -1
```

This code simply scans the array for the element x from left to right. The best-case scenario is when x is already positioned at the start of the array. However, as this only happens $\frac{1}{n^{th}}$ of the time, it is more meaningful to determine the *average-case* or *worst-case* performance of the algorithm when making efficiency considerations. Similar to the best-case scenario, we can determine the worst-case performance of the algorithm if x is positioned at the end of the array, or if x is not even in the array. This is particularly useful if we have mission critical problems that we cannot afford to risk having any worst-case incidence.

Another good measure of algorithm is its average-case performance. Let's demonstrate an average-case performance by taking the average number of steps taken to reach a randomly selected element of array size $n = 10000$ after 1000 attempts. We repeat this over 10 observations as shown below.

```
import random

n = 10000   # sample size
obs = 10    # number of observations

arr = range(n)

for j in range(obs):
    totalSteps = 0
    attempts = 1000
    for k in range(attempts):
        totalSteps += search(arr, (random.sample(arr,1)[0] % n))
    print(j + 1, ". Average = ", totalSteps/attempts, "\n", sep="")
```

The corresponding results are as follows:

```
1. Average = 4991.086
2. Average = 5028.156
3. Average = 4938.414
4. Average = 4914.839
5. Average = 5172.611
6. Average = 4922.84
7. Average = 5031.356
8. Average = 5041.052
9. Average = 5073.320
10. Average = 4952.059
```

We can see that for a sample size of $n = 10000$, it will take an average of 5000 steps, roughly half the sample size, to reach a randomly selected number. We can repeat this for different n values, and the results will be similar.

Now, compare it with the following code:

```
# Search Function 2
def search(arr, x):
    count = 0
    left = 0
    right = n - 1
    while right >= left:
        count += 1
        middle = (left + right) // 2    # ensure to use integer and not float
        if x == arr[middle]:
            return count
        if x < arr[middle]:
            right = middle - 1
        else:
            left = middle + 1
    return count
```

This search algorithm works by dividing the search interval in half for each iteration. Running the above code using the same `arr` variable and driver code will yield the following results:

```
1. Average = 12.277
2. Average = 12.382
3. Average = 12.417
4. Average = 12.343
5. Average = 12.283
6. Average = 12.372
7. Average = 12.362
8. Average = 12.353
9. Average = 12.271
10. Average = 12.386
```

The results are significantly different in that it takes an average of only 12 steps to search for a particular element in an array of size 10000. Let's repeat the above 2 search functions for a couple more times but doubling the sample size n each time. We then summarize the results in the table below.

Array Size, n	**10000**	**20000**	**40000**	**80000**	**160000**	**320000**
Average Steps (est.) - Search Function 1	5000	10000	20000	40000	80000	160000
Average Steps (est.) - Search Function 2	12	13	14	15	16	17

The results shows that for the first search function, the average number of steps taken to search a particular element in the array doubles as the array size also doubles. This function is known as a *linear search* function, and its computational time is directly proportional to the size of the problem. On the other hand, despite doubling the array size, the second search function will only take only 1 additional step on average. This function is called the *binary search* function, and the computational time is proportional to the *logarithm of the problem size*. The examples above demonstrates the quantitative analyses of algorithms. For a visualization of the two algorithms, please proceed here.

1.2.2 Challenges of Experimental Analysis

While experimental studies of running times are valuable, especially when finetuning production-quality code, there are three major limitations to their use for algorithm analysis:

- Experimental running times of two algorithms are difficult to directly compare unless the experiments are performed in the same hardware and software environments.
- Experiments can be done only on a limited set of test inputs; hence, they leave out the running times of inputs not included in the experiment (and these inputs may be important).
- An algorithm must be fully implemented in order to execute it to study its running time experimentally.
  This last requirement is the most serious drawback to the use of experimental studies. At early stages of design, when considering a choice of data structures or algorithms, it would be foolish to spend a significant amount of time implementing an approach that could easily be deemed inferior by a higher-level analysis.

Very often we are more interested in the qualitative analysis of algorithms. As such, a common way to estimate an algorithm's computational complexity is by using the *big O notation*.

1.3 Big O Notation

The big O notation is a mathematical notation describing the upper limiting behavior of a function when the argument tends towards infinity. It aims to simplify the analysis of an algorithm's computational complexity by measuring its *rate of growth* or *order of the function* in terms of the problem size n. The principle of determining the big O notation is to only consider the highest-order term of a formula and ignore the lower-order terms, since the lower-order terms are relatively insignificant for large values of n. We also ignore the highest-order term's constant coefficient, since constant factors are less significant than the order.

For a concrete example, let us revisit the search algorithms. Note that following examples presented are merely to support in explaining the different notations. The big O notation extends to all algorithms and therefore, are **not** only limited to these.

1.3.1 Constant Notation: *O(1)*

```
def getFirstElement(arr):
    return arr[0]
```

The `getFirstElement` function will retrieve the first element of a given array, and it will always be completed in one step regardless of size of the array. Therefore, this function is considered to have a computational complexity of $O(1)$.

This is also a central behavior of Python's list class is that it allows access to an arbitrary element of the list using syntax, `arr[i]`, for integer index i. Because Python's lists are implemented as array-based sequences, references to a list's elements are stored in a consecutive block of memory. The i^{th} element of the list can be found, not by iterating through the list one element at a time, but by validating the index,
and using it as an offset into the underlying array. In turn, computer hardware supports constant-time access to an element based on its memory address. Therefore, we say that the expression `arr[i]` is evaluated in $O(1)$ time for a Python list.

1.3.2 Linear Notation: *O(n)*

```
# given that the size of array is equals to n
def linearSearch(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i + 1
    return -1
```

To determine the big O notation of the linear search algorithm, we have to consider the worst-case performance, i.e., for a given list of size n, it can take up to n iterations to find a particular element if the element is at the end of the list. Therefore, it can be said that the computational complexity of a linear search algorithm is $O(n)$, pronounced as 'O of n'. So, if the size of the problem is doubled, while although it will take up to $2n$ iterations to find the last element, the big O notation is still $O(n)$. Similarly, an algorithm that cycles 0.1 times of the problem size i.e. $0.1n$ is also considered $O(n)$.

1.3.3 Logarithmic Notation: *O(log n)*

```
# given that the size of array is equals to n
def binarySearch(arr, x):
    count = 0
    left = 0
    right = n - 1
    while right >= left:
        count+=1
        middle = (left + right) // 2
        if x == arr[middle]:
            return count
        if x < arr[middle]:
            right = middle - 1
        else:
            left = middle + 1
    return count
```

Previously, we have seen that for binary search, it only takes one additional step to find an element despite the size of the problem, n, doubling. This growth rate is what is known as a *logarithmic* one. To provide a clearer picture, we use the values found previously in the binary search algorithm to plot a graph of the average number of steps against the array size, n.

```
import matplotlib.pyplot as plt

n = [10000, 20000, 40000,80000,160000,320000]
steps = [12,13,14,15,16,17]

plt.plot(n, steps, 'b')
plt.xlabel('Array size, n')
plt.ylabel('Number of steps')
plt.title('Logarithmic Complexity')
plt.show()
```

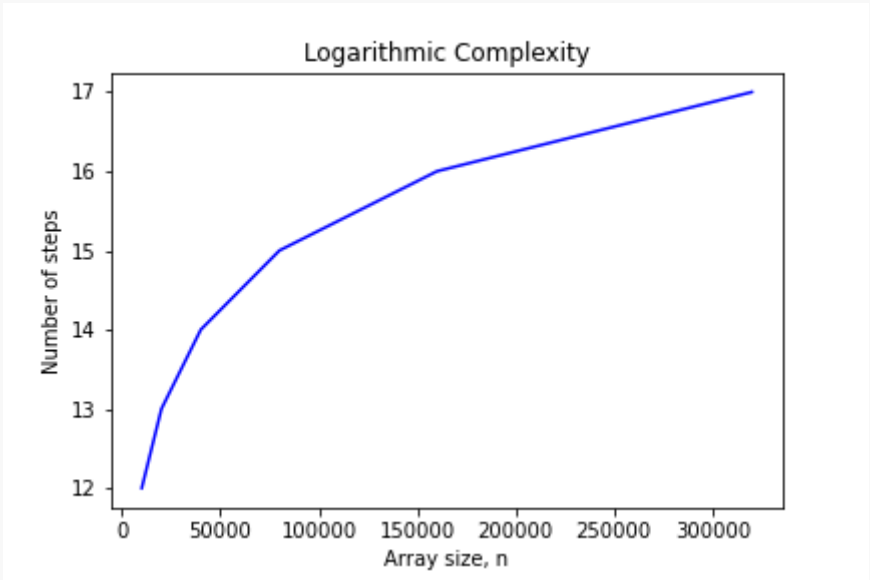

Fig 1.7. Logarithmic Complexity Graph.

We can see that a logarithmic algorithm is one whose rate of growth decreases as the problem size increases.

Also note that since many problems in computer science that repeatedly reduce the input size do so by half, it's not uncommon to use $\log n$ to imply $\log_2 n$ when specifying the run time of an algorithm.

1.3.4 Polynomial Notation: *O(n²)*, *O(n³)*, etc.

```
def searchSquareArray(arr, x, n): # where n is the size of the square array
    for row in range(n):
        for col in range(n):
            if arr[row][col] == x:
                return (row + 1), (col + 1)
    return -1
```

The function `searchSquareArray` above represents a search algorithm for finding the index of an element in a square array, where n is the size of the square array. It returns the row and column of the element in the square array.

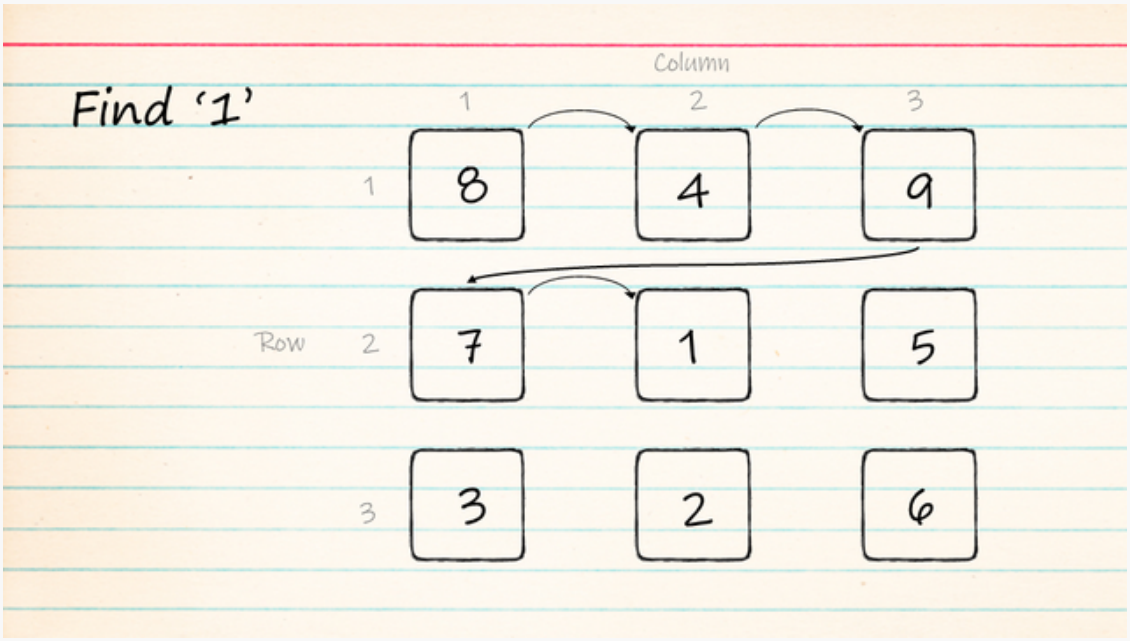

Fig 1.8. Visualization of a search algorithm in a square array.

This function is basically comprises of a nested `for` loop which iterates for 0 to n for each loop. In the worst case performance, where the element is in the last position, it will take a total of n^2 number of iterations. For example, in Fig 1.8, if we need to find the last element '6', we will need to to run a total of $n^2 = 3^2 = 9$ iterations. Overall, this is considered as a *quadratic* or $O(n^2)$ notation. Algorithms which are based on nested loops are more likely to have polynomial notations such as quadratic, or cubic, etc., depending on the level of nesting.

1.3.5 Characterizing Functions in Simplest Terms

Although, in most cases, the big O notation can be determined by simplifying terms. For example, while it is true that an algorithm whose growth rate is defined by the function $C(n) = 50000log(n) \ + \ 4n^2 \ + 0.7n \ + 830$ is $O(n^3)$, it more accurate to say that it is $O(n^2)$.

Consider, by way of analogy, a scenario where a hungry traveler driving along a long country road happens upon a local farmer walking home from a market. If the traveler asks the farmer how much longer he must drive before he can find some food, it may be truthful for the farmer to say, "certainly no longer than 12 hours," but it is much more accurate (and helpful) for him to say, "you can find a market just a few minutes drive up this road." Thus, even with the big O notation, we should strive as much as possible to tell the whole truth and describe it in the *simplest terms*.

1.3.6 Additional Information

Although we have gone through several big O notation types, there are still many other notations, such as $O(n^2)$, $O(n!)$, $O(n \log n)$, etc., that we did not cover. Note that not all possible notations will be covered.

A few words of caution about big O notation are in order at this point. First, note that the use of the big O notations notations can be somewhat misleading should the constant factors they "hide" be very large. For example, while it is true that the function $10^{100}n$ is $O(n)$, if this is the running time of an algorithm being compared to one whose running time is $10n \log n$, we should prefer the $O(n \log n)$ time algorithm, even though the linear-time algorithm is faster. This preference is because the constant factor, 10^{100} is believed by many astronomers to be an upper bound on the number of atoms in the observable universe. So we are unlikely to ever have a real-world problem that has this number as its input size. Thus, even when using the big O notation, we should at least be somewhat mindful of the constant factors and lower-order terms we are "hiding."

The observation above raises the issue of what constitutes a "fast" algorithm. Generally speaking, any algorithm running in $O(n \log n)$ time (with a reasonable constant factor) should be considered efficient. Even an $O(n^2)$ time function may be fast enough in some contexts, that is, when n is small. But an algorithm running in $O(2^n)$ time should almost never be considered efficient.