1. Introduction

In simple terms, an algorithm is a step-by-step method for completing a task within a limited amount of time, while a data structure is a systematic way to organize and access data. These concepts are fundamental to computing. This course focuses on the principles and methods for designing and implementing correct and efficient data structures and algorithms.

1.1. Data Structure

What is Data? Data can be defined as the elementary value or collection of values that convey some meaning. For example, a student's name and their ID are pieces of data about the student.

What is a Record? A record can be defined as a collection of various data items that typically describe an entity. For example, a student's record might include their name, address, course, and marks. These items are grouped together to form a complete record of the student.

What is Abstract Data Type? An Abstract Data Type (ADT) is a way to describe what a data structure does without specifying how it does it. It's like a blueprint that tells you what operations you can perform on the data and what rules it follows, but not how these operations are implemented. For example, an ADT for a stack would include operations like push (add an item), pop (remove an item), and peek (look at the top item) without explaining how these operations are coded. In fact, it is just a high-level definition of a data structure.

A data structure is a way to store and organize data so that it can be used efficiently. As the name indicates, it organizes data in memory. Data structures can be seen as a set of algorithms that we can use in any programming language to structure data in memory.

There are two main types of data structures:

1. **Primitive Data Structures (provided by the language makers):** These are the basic data types that are built into a programming language and are very specific to the language. They include:
    - **Integers (int):** Used to store whole numbers.
    - **Floating-point numbers (float, double):** Used to store numbers with decimal points.
    - **Characters (char):** Used to store individual characters.
    - **Pointers:** Used to store memory addresses.
2. **Non-Primitive Data Structures(user-defined):** These are more complex data structures that are derived from primitive data types. Informally, we can look at a non-primitive data structure in this way: it contains two components: a **container** (to store data) and a **set of functions**. A container is usually composed of one or more primitive data structures and the functions help us perform operations on our data set, such as inserting, deleting, and retrieving data, ... We can also use these functions to understand our data set better. These functions allow us to perform specific operations

on our data set or prevent us from performing particular actions. This distinction makes different data structures suitable and efficient for specific applications. They can be further divided into:

- **Linear Data Structures:** These organize data in a sequential manner. Each element is connected to the next, forming a linear sequence. Examples include:
    - **Arrays:** A collection of elements identified by index or key.
    - **Linked Lists:** A series of connected nodes where each node contains data and a reference to the next node.
    - **Stacks:** A collection of elements that follows the Last In, First Out (LIFO) principle.
    - **Queues:** A collection of elements that follows the First In, First Out (FIFO) principle.
- **Non-Linear Data Structures:** In these structures, each element can be connected to multiple elements, forming a hierarchical or graph-like structure. Examples include:
    - **Trees:** A hierarchical structure with a root element and child elements, forming a parent-child relationship.
    - **Graphs:** A collection of nodes connected by edges, which can represent various relationships between elements.

What is the need for data structures?

As applications become more complex and data grows daily, several issues can arise. Processors may struggle to handle billions of files, slowing down operations. For example, searching for an item in a store with a million items can be very slow if the application has to check each item one by one. Additionally, when thousands of users search data simultaneously, servers may fail to process all the requests efficiently.

Data structures help solve these problems by organizing data in a way that allows for quick and efficient retrieval, eliminating the need to search through all items each time.

Advantages of Data Structures:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it improves efficiency.
- **Reusability:** Data structures provide reusability, meaning that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by the ADT also provides a level of abstraction. The client cannot see the internal workings of the data structure, so they do not have to worry about the implementation.

1.2. Data Structures and Algorithm

An algorithm is a set of instructions designed to perform calculations or solve problems, especially by a computer. It is not a complete program or code; it is just the logic for solving a problem. This logic can be represented either through a flowchart or pseudocode.

Characteristics of an Algorithm:

1. **Input:** An algorithm takes some input values. We can provide these input values as needed.
2. **Output:** An algorithm produces at least one output after processing the input.
3. **Unambiguity:** An algorithm should be clear and straightforward. Each instruction must be easy to understand and follow.
4. **Finiteness:** An algorithm must have a limited number of steps. It should eventually come to an end.
5. **Effectiveness:** Each step of the algorithm must be effective, meaning it has a direct impact on solving the problem.

These characteristics ensure that algorithms are practical and efficient solutions for computational problems.

We learn algorithms and data structures simultaneously because they are deeply interconnected and essential for efficient problem-solving in computer science. Here's why:

1. **Efficiency:** Algorithms depend on data structures to manage and organize data efficiently. Choosing the right data structure can significantly improve the performance of an algorithm.
2. **Problem-Solving:** Many problems require specific data structures for optimal solutions. Understanding both helps in selecting the best approach to store and manipulate data.
3. **Foundation:** Both are fundamental concepts in computer science. A solid grasp of data structures enhances your ability to design effective algorithms, and vice versa.
4. **Application:** Real-world applications often require the combined use of algorithms and data structures. For example, searching and sorting operations rely heavily on the interaction between the two (without having a sorted data structure, you cannot use Binary Search Algorithm).

By learning them together, you develop a more comprehensive understanding of how to design, analyze, and implement efficient solutions to complex problems.

Approaches in Algorithms:

1. **Brute Force Algorithm:** This approach applies a general logic structure to design an algorithm. Also known as an exhaustive search algorithm, it searches all possible solutions to find the required one. There are two types:
    - **Optimizing:** Finds all solutions to a problem and then selects the best one.
    - **Satisficing:** Stops once the best solution is found.

2. **Divide and Conquer:** This method breaks down a problem into smaller sub-problems, solves each sub-problem, and then combines their solutions to produce the final output. It allows for handling complex problems more efficiently by dividing them into manageable parts.

3. **Greedy Algorithm:** This paradigm makes an optimal choice at each step with the hope of finding the best overall solution. It is easy to implement and has a faster execution time, but it rarely provides the optimal solution in all cases.

4. **Dynamic Programming:** This technique involves breaking down a problem into smaller overlapping sub-problems and solving each sub-problem only once. It stores the solutions to sub-problems in a table to avoid redundant computations, resulting in improved efficiency.

5. **Backtracking:** Backtracking is a systematic way of searching for solutions to problems by trying various options and backtracking from those choices that do not satisfy the problem's constraints. It is particularly useful for problems where a sequence of decisions needs to be made, and not all decisions lead to a valid solution.

2. Algorithm Analysis

The performance of an algorithm can be measured in two factors:

1. **Time Complexity**: This refers to the amount of time required to complete the execution of an algorithm. Time complexity is mainly calculated by counting the number of steps needed to finish execution.
2. **Space Complexity**: An algorithm's space complexity is the amount of memory required to solve a problem and produce an output. Space complexity is calculated as the sum of auxiliary space and input size.

2.1. Time Complexity

Running time is a natural measure for evaluating performance, since time is a precious resource. It is an important consideration in economic and scientific applications, since everyone expects computer applications to run as fast as possible. The running time of an algorithm or data structure operation typically depends on a number of factors, so what should be the proper way of measuring it? We will discuss the three methodologies as follows, noting that the first two are not practical:

1. **Timer:** If an algorithm has been implemented, we can study its running time by executing it on various test inputs and recording the actual time spent in each execution. Such measurements can be taken in an accurate manner by using system calls that are built into the language or operating system for which the algorithm is written. So, to measure the efficiency of an algorithm, we need to write the code, run a series of test cases, and use a timer to record how long the algorithm takes to execute. This method helps estimate the algorithm's efficiency in practical terms. However, this approach has some challenges: the running time is affected by the hardware environment (processor, clock rate, memory, disk, etc.) and software environment (operating system, programming language, compiler, interpreter, etc.) in which the algorithm is implemented, compiled, 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 run on a virtual machine.

2. **Counting operations:** Another method to estimate efficiency is to count the fundamental operations the algorithm performs. These operations include mathematical operations, comparisons, setting values, and retrieving values. By determining how many of these operations the algorithm uses as a function of the input size, we can get a sense of its efficiency. This approach abstracts away the specifics of hardware and provides a more consistent measure of performance. This method of measuring efficiency depends on the specific implementations of algorithms and can be unclear about which operations should be counted, making it difficult to apply consistently. We will discuss this method further.
3. **Order of Growth:** Order of growth describes the behavior of an algorithm in terms of how its running time or space requirements grow as the input size increases. It helps in abstractly comparing different algorithms and understanding their efficiency. This concept leads to the classification of algorithms into some classes, such as O(n), O(log n), and O(n^2), which describe their efficiency in a standardized way. We will discuss this method further.

As we mentioned in "Timer" method, experimental studies on running times are valuable but constrained. They rely on a limited range of test inputs that must be representative. Comparing the efficiency of algorithms requires identical hardware and software environments. Plus, implementing and executing an algorithm is necessary for experimental runtime analysis. While experimentation has an important role to play in algorithm analysis, it alone is not sufficient. Therefore, in addition to experimentation, we desire an analytic framework that

a) Takes into account all possible inputs
b) Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment
c) Can be performed by studying a high-level description of the algorithm without actually implementing it or running experiments on it.

Given that an algorithm comprises mathematical, logical, and memory access operations, counting the number of operations appears to be a promising method for algorithm analysis. However, before delving into this approach, it's essential to discuss several components of this methodology, namely:

1. A language for describing algorithms: **Pseudocode**
2. A computational model that algorithms execute within: **Random Access Machine and Primitive Operations**
3. A metric for measuring algorithm running time: **Asymptotic bounds**
4. An approach for characterizing running times

2.1.1. Pseudocode

Programmers are often asked to describe algorithms in a way that is intended for human eyes only. Such descriptions are not computer programs, but are more structured than usual prose. They also facilitate the high-level analysis of a data structure or algorithm. We call these descriptions pseudocode. Note that the pseudocode

is more compact than an equivalent actual software code fragment would be. In addition, the pseudocode is easier to read and understand.

Example. The array-maximum problem is the simple problem of finding the maximum element in an array A storing n integers. To solve this problem, we can use an algorithm called arrayMax, which scans through the elements of A using a for loop.

Algorithm arrayMax(A, n):
  Input: An array A storing $n \geq 1$ integers.
  Output: The maximum element in A.
  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $currentMax < A[i]$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudo-code is a mixture of natural language and high-level programming constructs that describe the main ideas behind a generic implementation of a data structure or algorithm. There really is no precise definition of the pseudocode language, however, because of its reliance on natural language. At the same time, to help achieve clarity, pseudocode mixes natural language with standard programming language constructs. The programming language constructs we choose are those consistent with modern high-level languages such as Python, C++, and Java. These constructs include the following:

- **Expressions:** We use standard mathematical symbols to express numeric and Boolean expressions. We use the left arrow sign (←) as the assignment operator in assignment statements (equivalent to the = operator in Java) and we use the equal sign (=) as the equality relation in Boolean expressions (equivalent to the "==" relation in Java).
- **Method declarations: Algorithm** name(param1, param2, . . .) declares a new method "name" and its parameters.
- **Decision structures: if** condition **then** true-actions [else false-actions]. We use indentation to indicate what actions should be included in the true-actions and false-actions, and we assume Boolean operators allow for short-circuit
evaluation.
- **While-loops: while** condition **do** actions. We use indentation to indicate what actions should be included in the loop actions.
- **Repeat-loops**: **repeat** actions **until** condition. We use indentation to indicate what actions should be included in the loop actions.
- **For-loops: for** variable-increment-definition **do** actions. We use indentation to indicate what actions should be included among the loop actions.
- **Array indexing:** A[i] represents the ith cell in the array A. We usually index the cells of an array A of size n from 1 to n, as in mathematics, but sometimes we instead such an array from 0 to n − 1, consistent with Java.
- **Method calls:** object.method(args) (object is optional if it is understood).
- **Method returns: return** value. This operation returns the value specified to the method that called this one.

When we write pseudocode, we must keep in mind that we are writing for a human reader, not a computer. Thus, we should strive to communicate high-level ideas, not low-level implementation details. At the same time, we should not gloss over important steps. Like many forms of human communication, finding the right balance is an important skill that is refined through practice.

2.1.2. Primitive Operations

As we noted above, experimental analysis is valuable, but it has its limitations. If we wish to analyze a particular algorithm without performing experiments on its running time, we can take the following more analytic approach directly on the high-level code or pseudocode. We define a set of high-level primitive operations that are largely independent from the programming language used and can be identified also in the pseudocode. Primitive operations include the following:

- Assigning a value to a variable
- Calling a method
- Performing an arithmetic operation (for example, adding two numbers)
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method.

Specifically, a primitive operation corresponds to a low-level instruction with an execution time that depends on the hardware and software environment but is nevertheless constant. Instead of trying to determine the specific execution time of each primitive operation, we will simply count how many primitive operations are executed, and use this number t as a high-level estimate of the running time of the algorithm. This operation count will correlate to an actual running time in a specific hardware and software environment, for each primitive operation corresponds to a constant-time instruction, and there are only a fixed number of primitive operations. The implicit assumption in this approach is that the running times of different primitive operations will be fairly similar. Thus, the number, t, of primitive operations an algorithm performs will be proportional to the actual running time of that algorithm.

This approach of counting basic operations leads to a computational model called the Random Access Machine (RAM). In this model, a computer is seen as a CPU connected to a bank of memory cells, each storing a word (a number, a string, or an address). The term “random access” means the CPU can access any memory cell with one basic operation. To keep it simple, we assume there are no limits on the size of numbers in memory and that the CPU can perform any basic operation in a constant number of steps, regardless of input size. Thus, an accurate bound on the number of primitive operations an algorithm performs corresponds directly to the running time of that algorithm in the RAM model.

Example: We now show how to count the number of primitive operations executed by an algorithm, using as an example algorithm arrayMax, whose pseudocode was given back. We do this analysis by focusing on each step of the algorithm and counting the primitive operations that it takes

```
Algorithm arrayMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
  currentMax ← A[0]
  for i ← 1 to n − 1 do
      if currentMax < A[i] then
          currentMax ← A[i]
  return currentMax
```

Initializing the variable currentMax to A[0] corresponds to two primitive operations (indexing into an array and assigning a value to a variable) and is executed only once at the beginning of the algorithm. Thus, it contributes two units to the count. At the beginning of the for loop, counter i is initialized to 1. This action corresponds to executing one primitive operation (assigning a value to a variable). Before entering the body of the for loop, condition i < n is verified. This action corresponds to executing one primitive instruction (comparing two numbers). Since counter i starts at 1 and is incremented by 1 at the end of each iteration of the loop, the comparison i < n is performed n times. Thus, it contributes n units to the count. The body of the for loop is executed n − 1 times (for values 1, 2, . . . , n – 1 of the counter). At each iteration, A[i] is compared with currentMax (two primitive operations, indexing and comparing), A[i] is possibly assigned to currentMax (two primitive operations, indexing and assigning), and the counter i is incremented (two primitive operations, summing and assigning). Hence, at each iteration of the loop, either four or six primitive operations are performed, depending on whether A[i] ≤ currentMax or A[i] > currentMax. Therefore, the body of the loop contributes between 4(n − 1) and 6(n − 1) units to the count. Returning the value of variable currentMax corresponds to one primitive operation, and is executed only once. To summarize, the number of primitive operations $t(n)$ executed by algorithm arrayMax is at least: $2 + 1 + n + 4(n - 1) + 1 = 5n$ and at most: $2 + 1 + n + 6(n - 1) + 1 = 7n - 2$.

2.1.3. Asymptotic Notation

We have gone into a lot of detail to evaluate the running time of a simple algorithm like arrayMax. Doing this for more complicated algorithms would be very impractical. Generally, each step in pseudocode (or each line of code in a high-level language) corresponds to a small number of basic operations that don't depend on the input size. Therefore, we can simplify our analysis by estimating the number of these basic operations, roughly, by counting the ***steps*** in the pseudocode (or the lines of code executed). Luckily, there is a notation that helps us understand the main factors affecting an algorithm's running time without needing to count every single basic operation exactly.

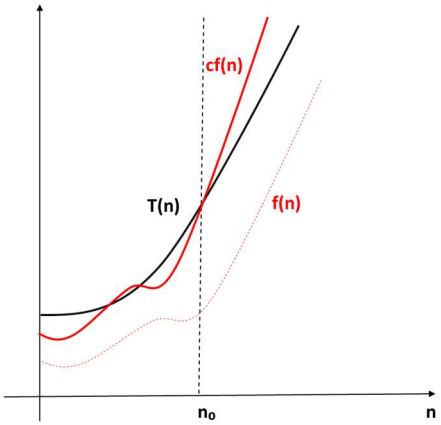
There are three different ways to estimate the time complexity of an algorithm, T(n):

1. **Big-Oh(upper bound):** This classification allows us to say asymptotically that T(n) is less than or equal to a particular function, up to a constant factor.
2. **Big-Omega(lower bound)**: This classification allows us to say asymptotically that T(n) is greater than or equal to a particular function, up to a constant factor.
3. **Big-Theta(tight bound)**: State that T(n) is asymptotically equal to a particular function, up to a constant factor.

Note. When we encounter a quantity that is unknown and cannot be precisely counted, we resort to estimation. Estimations can be provided as upper bounds, lower bounds, or tight bounds. For instance, consider the number of students in my next semester class. It could be at most 120(upper bound), at least 15(lower bound), or, with additional information, a tight bound like 40-45 students can be determined.

Let $T(n)$ be the total number of primitive operations of our algorithm as a function of input size (n), and $f(n)$ be a function mapping nonnegative integers to real numbers.

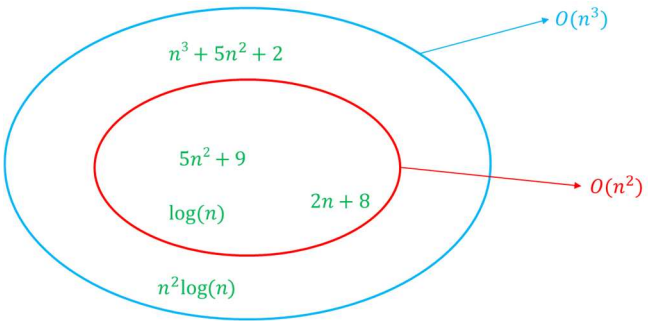
1. **Big-Oh:** We say that $T(n)$ is $O(f(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $T(n) \leq cf(n)$ for every integer $n \geq n_0$.



Example. $T(n) = 5n^2 + 9$ is $O(n^3)$. We need a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 + 9 \leq cn^3$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 14$ and $n_0 = 1$. So $14n^3$ is an upper bound for $T(n)$ for large values of n.

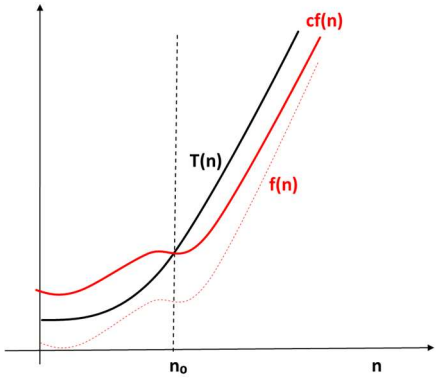
Example. $T(n) = 5n^2 + 9$ is $O(n^2)$. We need a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 + 9 \leq cn^2$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 6$ and $n_0 = 3$, but there are other possibilities as well. Note that $6n^2$ is an upper bound for $T(n)$ for large values of n.

From the above examples, we can infer that both $6n^2$ and $14n^3$ are upper bounds for $T(n)$ for large values of n. However, we obviously prefer the smaller upper bound. See the following figure.



Assume that my classroom has a capacity of 80 students, and I don't know exactly how many students I will have in my class. If I am asked how many students will register in my class, I could answer 'at most 1000 students' or 'at most 80 students.' Obviously, both answers are correct, but the latter is more realistic, and we prefer it to the former. So, when you are asked to find an upper bound for your algorithm's time complexity, try to find the smallest possible one.

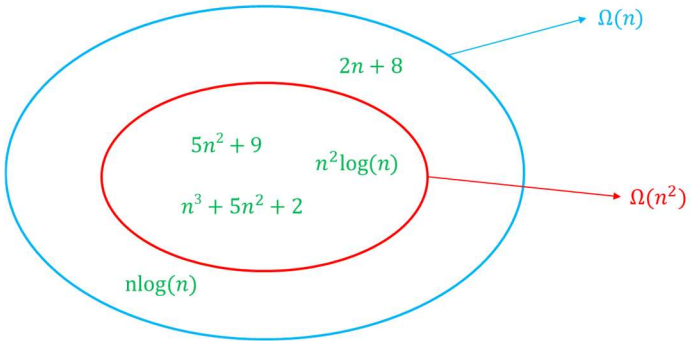
2**. Big-Omega**: We say that $T(n)$ is $\Omega(f(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $T(n) \geq cf(n)$ for every integer $n \geq n_0$.



Example. $T(n) = 5n^2 + 9$ is $\Omega(n)$. We need a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 + 9 \geq cn$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 1$ and $n_0 = 1$. So n is a lower bound for $T(n)$ for large values of n.

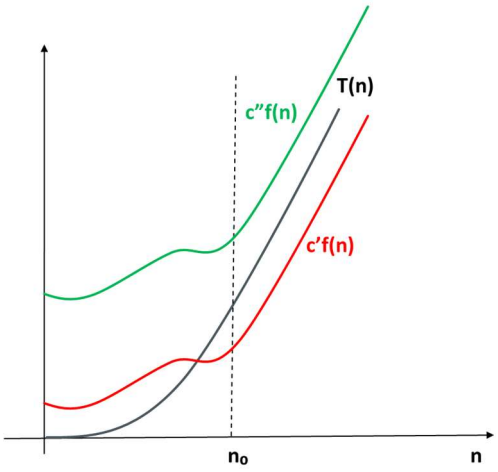
Example. $T(n) = 5n^2 + 9$ is $\Omega(n^2)$. We need a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $5n^2 + 9 \geq cn^2$ for every integer $n \geq n_0$. It is easy to see that a possible choice is $c = 5$ and $n_0 = 1$. Note that $5n^2$ is a lower bound for $T(n)$ for large values of n.

From the above examples, we can infer that both $5n^2$ and n are lower bounds for $T(n)$ for large values of n. However, we obviously prefer the greater lower bound. See the following figure.



3. **Big-Theta:** We say that $T(n)$ is $\Theta\left(f(n)\right)$, if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$; that is, there are real constants $c' > 0$ and $c'' > 0$, and an integer constant $n_0 \geq 1$ such that $c'f(n) \leq T(n) \leq c''f(n)$ for every integer $n \geq n_0$.

Example. $T(n) = 5n^2 + 9$ is $\Theta(n^2)$ because it is both $O(n^2)$ and $\Omega(n^2)$.



As you can see in the figure, the growth of T(n) is bounded between two functions, $c'f(n)$ and $c''f(n)$. This means that its behavior is very predictable.

Fact: Any polynomial, $a_k n^k + a_{k-1}n^{k-1} + \cdots + a_0$, is always $\Theta(n^k)$.

Example. Time complexity of arrayMax algorithm is $\Theta(n)$**.**

Asymptotic notation highlights the importance of algorithm design by showing how algorithms that grow slower (like $O(nlog\ n)$) perform better over time than those with faster growth rates (like $O(n^2)$), even if the constant factor for the faster algorithm is worse.

2.1.4. Characterizing Running Times

If you are able to count all primitive operations and calculate the time complexity as a function of input size (n), You can represent the time complexity using big-Oh notation by identifying the dominant term. To achieve this, simply drop all constants and lower-order terms. For example, in the arrayMax algorithm, after counting all primitive operations, we obtained $T(n) = 7n - 2$. By dropping constants and lower-order terms, we arrive at $T(n) = O(n)$.

However, it is often neither possible nor practical to count the exact number of primitive operations. To circumvent this, instead of examining the program's statements one by one, we directly represent the time

complexity in big-Oh notation. For this purpose, focus only on loops, function calls, and recursion, which have the most significant impact on the code's execution time. Consequently, we can identify the dominant term without painstakingly counting each one. Now, let's look at an example to understand this approach better. We'll illustrate how to employ big-Oh notation to analyze two algorithms that solve the same problem but have different running times.

Example. Maximum Subarray Problem: We are given an array of integers and asked to find the subarray whose elements have the largest sum. See the example. That is, given array $A = [a_1, a_2, \ldots, a_n]$ (indexed from 1), find indices j and k that maximize the sum

$$s_{j,k} = a_j + a_{j+1} + \cdots + a_k = \sum_{i=j}^{k} a_i$$

For example if A = [-2, -4, 3, -1, 5, 6, -7, -2, 4, -3, 2], j=3 and k=6 will be the output, $s_{3,6} = 13$.

a) Our first algorithm computes the maximum of every possible subarray summation, $s_{j,k}$, of A separately.

Algorithm MaxsubSlow(A):
Input: An n-element array A of numbers, indexed from 1 to n.
Output: The maximum subarray sum of array A.

```
m ← 0     // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s ← 0     // the next partial sum we are computing
        for i ← j to k do
            s ← s + A[i]
        if s > m then
            m ← s
return m
```

Analyzing the running time of the MaxsubSlow algorithm: the outer loop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the inner-most loop, for index i, will iterate at most n times. Thus, the running time of the MaxsubSlow algorithm is $O(n^3)$. b) We can design an improved algorithm for the maximum subarray problem by observing that we are wasting a lot of time by recomputing all the subarray summations from scratch in the inner loop of the MaxsubSlow algorithm. There is a much more efficient way to calculate these summations. The crucial insight is to consider all the prefix sums, which are the sums of the first t integers in A for $t = 1, 2, \ldots, n$. That is, consider each prefix sum, S_t, which is defined as

$$S_t = a_1 + a_2 + \cdots + a_t = \sum_{i=1}^{t} a_i$$

If we are given all such prefix sums, then we can compute any subarray summation, $s_{j,k}$, in constant time using the formula: $s_{j,k} = S_k - S_{j-1}$.

Analyzing the running time of the Maxsubfaster algorithm: the outerloop, for index j, will iterate n times, its inner loop, for index k, will iterate at most n times, and the steps inside that loop will only take O(1) time in each iteration. Thus, the total running time of the MaxsubFaster algorithm is $O(n^2)$, which improves the running time of the MaxsubSlow algorithm by a linear factor.

```
Algorithm MaxsubFaster(A):
  Input: An n-element array A of numbers, indexed from 1 to n.
  Output: The maximum subarray sum of array A.
S_0 ← 0     // the initial prefix sum
for i ← 1 to n do
    S_i ← S_{i−1} + A[i]
m ← 0     // the maximum found so far
for j ← 1 to n do
    for k ← j to n do
        s = S_k − S_{j−1}
        if s > m then
            m ← s
return m
```

3. Best case/Worst case/ Average case Time complexity

As we mentioned in the Maxarray algorithm, the time complexity can differ based on the test case even if the input size is always n. In the Maxarray algorithm, the time complexity can be at least $5n$ and at most $7n-2$ depending on the outcome of the condition in the **if** statement. However, in either case, it is $O(n)$. Sometimes, this difference can be even bigger; for example, you can have a test case of size n where the algorithm runs in $O(1)$ time, while for another test case of the same size n, it will run in $O(n)$ time. For instance, let L be a list of n numbers. We are given a number t, which is in L. To find the location of t in L, we can use the following algorithm:

```
Algorithm SequentialSearch(t, L)
Input: array L and number t
Output: index i such that L[i]=t
    for i=0 to n-1
        if t == L[i] then
            return i
return -1
```

If you are looking for the first element in L, this algorithm runs in $O(1)$ time, while if you look for the last element, it will run in O(n) time.

There are three approaches to evaluate the time complexity based on the different test cases:

1. **Best case:** The best case time complexity of an algorithm represents the lower bound on its cost, indicating the minimum amount of time or computational steps required for completion under the most favorable conditions. This scenario is determined by the "easiest" input of size n, where the algorithm performs optimally, such as searching for the first element in a list. Understanding the best case time complexity provides a benchmark or goal for optimization, guiding efforts to achieve the most efficient behavior the algorithm can exhibit.

2. **Worst case:** The worst case time complexity of an algorithm serves as the upper bound on its cost, representing the maximum amount of time or computational steps required for completion under the most adverse conditions. This scenario is determined by the "most difficult" input of

size n, where the algorithm performs least efficiently, such as searching for the last element in a list. Understanding the worst case time complexity provides a guarantee for all inputs, ensuring that the algorithm will never exceed a certain level of performance, regardless of the input characteristics.

3. **Average case:** The average case time complexity of an algorithm represents the expected cost for processing random inputs (all of size n), providing a measure of its efficiency across a broad spectrum of possible input data of size n. This scenario relies on a model for what constitutes "random" input, which can vary depending on the specific problem domain and characteristics of the algorithm. By analyzing the average case time complexity, we gain insight into the algorithm's typical performance under typical or average input conditions. This information allows us to predict the algorithm's behavior in real-world scenarios and make informed decisions regarding its suitability for specific tasks.

Note: To calculate the Average time complexity we need to know concepts from probability theory. Random variables are like special kinds of variables that change depending on the result of an experiment. They're usually numbers. For example, if you flip a coin, the result (heads or tails) can be a random variable, denoted by X. The probability of getting heads is 0.5, which is written as $Pr(X = head) = .5$.

When we're talking about how long something takes, like a computer program, we might use a random variable to describe it. We're often interested in what usually happens, or the "expected" value. For instance, if we flip a fair coin many times, we'd expect it to land heads about half the time and tails the other half. This idea of "expected" value helps us understand how things behave on average. The expected value of a discrete random variable X is defined as $E(X) = \sum_x x\ Pr(X = x)$, where the summation is defined over the range of X and $Pr(X = x)$ represents the probability of event x occurring.

For example, to calculate the average time complexity of the SequentialSearch algorithm, a random variable can represent the number of comparisons made between the target number t and elements in the list L. Each comparison is like a random event, as its outcome depends on the contents of the list and the value of t. Let's define the random variable X as the number of comparisons required until we find the target number t in the list L. X can take on different values based on the position of t in L. If t is located at the i-th position in L (which is index i-1, because indexing starts from 0), then X=i, because we need to check i elements before finding t. Now, to calculate the average time complexity, we need to find the expected value of X, denoted as $E[X]$, which represents the average number of comparisons. By definition of $E[X]$, we have: $E[X] = \sum_{i=1}^{n} iPr(X = i)$. Where $P(X$=$i)$ is the probability that t is located at the i-th position in L (i.e., index $i - 1$). Since each index is equally likely if t is present in L, $Pr(X = i) = 1/n$ for $1 \leq i \leq n$. Therefore, we have:

$$E[X] = \sum_{i=1}^{n} iPr(X = i) = \sum_{i=1}^{n} \frac{i}{n} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}.\frac{n(n+1)}{2} = \frac{n+1}{2}.$$

So, the average time complexity of the SequentialSearch algorithm, when we know L contains t, is approximately (n+1)/2, which is $O(n)$. This means that on average, the algorithm will perform about (n+ 1)/2 comparisons before finding the target t.

4. Exercises (with Solutions)

4.1. Easy: Calculate the time complexity of the following algorithms. The Python code for the algorithms is provided. (But you need to calculate the time complexity of given pseudocode not Python codes. In some cases they are not the same.)

1. EXERCISE 1: MULTIPLICATION USING ADDITION

```
Algorithm func1(a, b)
    // Functionality: Multiplies two numbers using repeated addition.
    // Time Complexity: O(n)
    result ← 0
    for i ← 0 to b - 1 do
        result ← result + a
    return result
// Sample function call: func1(5, 3) -> Output: 15
```

```python
# Exercise 1
def func1(a, b):
    """
    Functionality: Multiplies two numbers using repeated addition.
    Time Complexity: O(n)
    """
    result = 0
    for i in range(0, b):
        result += a
    return result
# Sample function call: func1(5, 3) -> Output: 15
```

This function multiplies two numbers, 'a' and 'b', using repeated addition. Specifically, it initializes a result variable to 0 and adds 'a' to it 'b' times.

Time Complexity Analysis: Since the function performs a constant time operation ('a' additions) 'b' times, the time complexity is O(b), making it linear with respect to 'b'. This is straightforward without the need for Master's theorem or recurrence relations.

2. EXERCISE 2: EXPONENTIATION

```
Algorithm func2(a, b)
    // Functionality: Computes a^b (a raised to the power of b) using a loop.
    // Time Complexity: O(n)
    result ← 1
    while b > 0 do
        result ← result * a
        b ← b - 1
    return result
// Sample function call: func2(2, 4) -> Output: 16
```

```python
# Exercise 2
def func2(a, b):
    """
    Functionality: Computes a^b (a raised to the power of b) using a loop.
    Time Complexity: O(n)
    """
    result = 1
    while b > 0:
        result = result * a
        b = b - 1
    return result
# Sample function call: func2(2, 4) -> Output: 16
```

This function calculates 'a' raised to the power of 'b' (a^b) using a loop. It initializes the result to 1 and multiplies it by 'a', 'b' times.

Time Complexity Analysis: The loop runs 'b' times, with each iteration performing a multiplication operation, resulting in a time complexity of O(b). This linear time complexity reflects the direct proportionality to the exponent 'b'.

3. EXERCISE 3: SUM OF DIGITS

```
Algorithm func3(num)
    // Functionality: Calculates the sum of the digits of a number.
    // Time Complexity: O(log n)
    result ← 0
    while num > 0 do
        result ← result + (num mod 10)
        num ← num // 10 // integer division
    return result
// Sample function call: func3(1234) -> Output: 10
```

```python
# Exercise 3
def func3(num):
    """
    Functionality: Calculates the sum of the digits of a number.
    Time Complexity: O(log n)
    """
    result = 0
    while num > 0:
        result = result + num % 10
        num //= 10 # integer division -> 3//2 = 1
    return result
# Sample function call: func3(1234) -> Output: 10
```

Calculates the sum of the digits of a number by continuously dividing the number by 10 and adding the remainder to a result variable.

Time Complexity Analysis: The number of operations is proportional to the number of digits in 'num', which can be represented by log10(num). Therefore, the time complexity is O(log n), where 'n' is the value of 'num'. This logarithmic complexity arises because the input size decreases exponentially with each division by 10.

4. EXERCISE 4: INTERSECTION COUNT USING BINARY SEARCH

```
Algorithm func4(a, b)
    // Functionality: Finds the count of intersecting elements between two arrays using
binary search.
    // Time Complexity: O(n log n)
    b.sort() // Time complexity of this function is O(N log N) where N = len(b).
    intersect ← 0
    for x in a do
        if binary_search(b, x) ≥ 0 then
            intersect ← intersect + 1
    return intersect

Algorithm binary_search(arr, target)
    low ← 0
    high ← len(arr) - 1
    while low ≤ high do
        mid ← (low + high) // 2
        if arr[mid] = target then
            return mid
        else if arr[mid] < target then
            low ← mid + 1
        else
            high ← mid - 1
    return -1

// Sample function call: func4([1, 2, 3, 4], [2, 4, 6, 8]) -> Output: 2
```

```python
def func4(a, b):
    """
    Functionality: Finds the count of intersecting elements between two arrays using binary
search.
    Time Complexity: O(n log n)
    """
    b.sort()#Time complexity of this function is O(Nlog N) where N=len(b).
    intersect = 0
    for x in a:
        if binary_search(b, x) >= 0:
            intersect = intersect + 1
    return intersect

def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
```

```
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
# Sample function call: func4([1, 2, 3, 4], [2, 4, 6, 8]) -> Output: 2
```

This function finds the count of intersecting elements between two arrays. It sorts one array (neglected in complexity analysis) and then performs a binary search for each element of the other array within the sorted array.

Time Complexity Analysis: Assuming 'n' is the length of array 'a' and 'm' is the length of array 'b', the binary search for each element of 'a' in 'b' results in a time complexity of O(n log m), since binary search has a complexity of O(log m) and it is performed 'n' times. So the total time complexity is O(mlog m)+O(nlogm)=O(max{n,m}log m). the first term is for sorting b.

5. EXERCISE 5: SQUARE ROOT BY SEQUENTIAL SEARCH

```
Algorithm func5(n)
    // Functionality: Finds the square root of 'n', if it is a perfect square, by
sequential search.
    // Time Complexity: O(sqrt(n))
    guess ← 1
    while guess * guess ≤ n do
        if guess * guess = n then
            return guess
        guess ← guess + 1
    return None

// Sample function call: func5(16) -> Output: 4
```

```
# Exercise 5
def func5(n):
    """
    Functionality: Finds the square root of 'n', if it is a perfect square, by sequential
search.
    Time Complexity: O(sqrt(n))
    """
    guess = 1
    while guess * guess <= n:
        if guess * guess == n:
            return guess
        guess = guess + 1
    return None
# Sample function call: func5(16) -> Output: 4
```

Finds the square root of 'n', if n is a perfect square number, by sequentially checking each positive integer number starting from 1 until the square of the number is equal to 'n'.

Time Complexity Analysis: The time complexity is O(sqrt(n)) because the operation is repeated until the guess squared is greater than or equal to 'n'. The complexity is directly proportional to the square root of 'n', illustrating a square root time complexity.

6. EXERCISE 6: CHECK IF ARRAY IS SORTED

```
Algorithm func6(arr)
    // Functionality: Checks if an array is sorted in ascending order.
    // Time Complexity: O(n)
    for i ← 1 to len(arr) - 1 do
        if arr[i - 1] > arr[i] then
            return False
    return True

// Sample function call: func6([1, 2, 3, 4, 5]) -> Output: True
```

```python
# Exercise 6
def func6(arr):
    """
    Functionality: Checks if an array is sorted in ascending order.
    Time Complexity: O(n)
    """
    for i in range(1, len(arr)):
        if arr[i-1] > arr[i]:
            return False
    return True
# Sample function call: func6([1, 2, 3, 4, 5]) -> Output: True
```

Checks if an array is sorted in ascending order by comparing each pair of adjacent elements, returning False if a pair is found in the wrong order.

Time Complexity Analysis: The time complexity is O(n), where 'n' is the length of the array. This linear complexity arises because each element is compared only once with its next element, requiring a single pass through the array.

7. EXERCISE 7: DUPLICATE EACH ELEMENT IN AN ARRAY

```
Algorithm func7(arr)
    // Functionality: Duplicates each element in an array.
    // Time Complexity: O(n)
    result ← new Array(len(arr) * 2)
    index ← 0
    for i=0 to len(arr) do
        item=arr[i]
        result[index] ← item
        index ← index + 1
        result[index] ← item
        index ← index + 1
    return result

// Sample function call: func7([1, 2, 3]) -> Output: [1, 1, 2, 2, 3, 3]
```

```python
# Exercise 7
def func7(arr):
    """
    Functionality: Duplicates each element in an array.
    Time Complexity: O(n)
    """
    result = []
    for item in arr:
        # Note: append is a built-in function in Python which is used to append an element to
an array at the end. Its time complexity is O(1)
        result.append(item) # O(1)
        result.append(item) # O(1).
    return result
# Sample function call: func7([1, 2, 3]) -> Output: [1, 1, 2, 2, 3, 3]
```

Duplicates each element in an array by iterating through the array and appending each element twice to a new array.
Time Complexity Analysis: The time complexity is O(n) because we have one for loop containing constant operations.

8. EXERCISE 8: ROTATE AN ARRAY

```
Algorithm func8(arr, k)
    // Functionality: Rotates an array to the right by 'k' steps.
    // Time Complexity: O(n)
    arr_len ← custom_len(arr)
    k ← k mod arr_len
    // Initialize a new array of the same length as 'arr'
    custom_arr ← new Array(arr_len)
    // Copy elements from 'arr' to 'rotated_arr' with the right rotation
    for i ← 0 to arr_len - 1 do
        mystery_index ← (i + k) mod arr_len
        custom_arr[mystery_index] ← arr[i]
    return custom_arr
// Sample function call: func8([1, 2, 3, 4, 5], 2) -> Output: [4, 5, 1, 2, 3]
```

```python
# Exercise 8
def func8(arr, k):
    """
    Functionality: Rotates an array to the right by 'k' steps.
    Time Complexity: O(n)
    """
    arr_len = len(arr)
    k = k % arr_len
    # Initialize a new array of the same length as 'arr'
    custom_arr = [0] * arr_len
    # Copy elements from 'arr' to 'rotated_arr' with the right rotation
    for i in range(arr_len):
        mystery_index = (i + k) % arr_len
        custom_arr[mystery_index] = arr[i]
    return custom_arr
# Sample function call: func8([1, 2, 3, 4, 5], 2) -> Output: [4, 5, 1, 2, 3]
```

Rotates an array to the right by 'k' steps. This involves moving each element to its new position, with the rotation considering wrapping around the array.

Time Complexity Analysis: The time complexity is O(n), where 'n' is the size of the array. We have only a for loop that repeats n times. Note that despite the rotation, each element is moved exactly once to its new position, ensuring a single linear pass through the array.

9. EXERCISE 9: COUNT VOWELS

```
Algorithm func9(s)
    // Functionality: Counts the number of vowels in a string.
    // Time Complexity: O(n)
    s ← s.lower() //O(n)
    count ← 0
    for i=0 to len(s) do
        if is_vowel(s[i], vowels) then
            count ← count + 1
    return count

Algorithm is_vowel(char, vowels)
    // Functionality: Checks if a character is a vowel.
    // Time Complexity: O(1)
    if char = 'a' or char = 'e' or char = 'i' or char = 'o' or char = 'u' then
        return true
    return false

// Sample function call: func9("hello world") -> Output: 3
```

```python
# Exercise 9
def func9(s):
    """
    Functionality: Counts the number of vowels in a string.
    Time Complexity: O(n)
    """
    s = s.lower() #O(n)
    vowels = 'aeiou'
    count = 0
    for char in s:
        if char in vowels: #O(1)
            count += 1
    return count
# Sample function call: func9("hello world") -> Output: 3
```

Counts the number of vowels in a given string by iterating through the string and checking if each character is a vowel.

Time Complexity Analysis: The time complexity is O(n), with 'n' being the length of the string s. We have only a for loop that repeats n times. Inside for loop, we have a function call which runs in a constant time (containing 5 comparisons).

10. EXERCISE 10: LAST ELEMENT OF AN ARRAY

```
Algorithm func10(arr)
    // Functionality: Returns the last element of an array or None if the array is empty.
    // Time Complexity: O(1)
    arr_len ← len(arr)
    if arr_len > 0 then
        return arr[arr_len - 1]
    else
        return None

// Sample function call: func10([1, 2, 3, 4, 5]) -> Output: 5
```

```python
# Exercise 10
def func10(arr):
    """
    Functionality: Returns the last element of an array or None if the array is empty.
    Time Complexity: O(1)
    """
    arr_len = len(arr)
    if arr_len > 0:
        return arr[arr_len - 1]
    else:
        return None
# Sample function call: func10([1, 2, 3, 4, 5]) -> Output: 5
```

Returns the last element of an array or None if the array is empty. This operation does not involve iteration and directly accesses the last index.

Time Complexity Analysis: The time complexity is O(1) since accessing an element at a known index in an array is a constant time operation, independent of the array's size.
Note: The time complexity of the len() operation on lists is constant because it doesn't depend on the size of the list. The length attribute is maintained and updated in constant time as elements are added or removed.

11. EXERCISE 11: Calculate average of an array

```
Algorithm custom_sum(arr)
    // Functionality: Computes the sum of elements in an array.
    // Time Complexity: O(n)
    total ← 0
    arr_len=len(arr)
    for i=0 to arr_len-1 do
        num=arr[i]
        total ← total + num
    return total

Algorithm func11(arr)
    // Functionality: Computes the average of elements in an array.
    // Time Complexity: O(n)
    total ← custom_sum(arr)
    count ← len(arr)
```

```
    if count > 0 then
        return total / count
    else
        return 0

// Sample function call: func11([1, 2, 3, 4, 5])
```

```python
# Exercise 11
def custom_sum(arr):
    total = 0
    for num in arr:
        total += num
    return total
def func11(arr):
    total = custom_sum(arr)
    count = len(arr)
    if count > 0:
        return total / count
    else:
        return 0
# Sample function call: func11([1, 2, 3, 4, 5])
```

Calculates the average of an array by summing all elements and then dividing by the number of elements. This involves a single pass to compute the total followed by a division.

Time Complexity Analysis: The time complexity is O(n), with 'n' being the number of elements in the array. Summing the elements requires a linear pass, and the division is a constant time operation.

12. EXERCISE 12: REPLACE NEGATIVES WITH ZERO

```
Algorithm func12(arr)
    // Functionality: Replaces each negative number in an array with 0.
    // Time Complexity: O(n), as it iterates through each element once.
    A ← new Array(len(arr))
    for i ← 0 to len(arr) - 1 do
        A[i] ← max(0, arr[i])
    return A
// Sample function call: func12([-1, 2, -3, 4, -5]) -> Output: [0, 2, 0, 4, 0]
```

```python
# Exercise 12
def func12(arr):
    """
    Functionality: Replaces each negative number in an array with 0.
    Time Complexity: O(n), as it iterates through each element once.
    """
    A = []
    for num in arr:
        A.append(max(0, num))
    return A
# Sample function call: func12([-1, 2, -3, 4, -5]) -> Output: [0, 2, 0, 4, 0]
```

Iterates through an array and replaces each negative number with 0. This modification is applied directly to each element based on its value.

Time Complexity Analysis: The time complexity is O(n), as we have a for loop.

13. EXERCISE 13: FILTER EVEN NUMBERS

```
Algorithm func13(arr)
    // Functionality: Filters an array to only include even numbers.
    // Time Complexity: O(n), as it iterates through each element once.
    index ← 0
    arr_len=len(arr)
    for i=0 in arr_len-1 do
        x=arr[i]
        if x % 2 = 0 then
            A[index] ← x
            index ← index + 1
    return A

// Sample function call: func13([1, 2, 3, 4, 5, 6]) -> Output: [2, 4, 6]
```

```python
# Exercise 13
def func13(arr):
    """
    Functionality: Filters an array to only include even numbers.
    Time Complexity: O(n), as it iterates through each element once.
    """
    A = []
    for x in arr:
        if x % 2 == 0:
            A.append(x) # Neglect any complexity for this line of code (assume it to be
O(1)).
    return A
# Sample function call: func13([1, 2, 3, 4, 5, 6]) -> Output: [2, 4, 6]
```

Creates a new array containing only the even numbers from the original array. It checks each element's divisibility by 2.

Time Complexity Analysis: The time complexity is O(n) because it requires iterating through the entire array once, performing a constant time check on each element.

14. Let find2D be an algorithm to find an element x in an n × n array A. The algorithm find2D iterates over the rows of A and calls the algorithm $\mathrm{SequentialSearch}(\mathrm{t}, \mathrm{L})$, on each one, until x is found or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? Is this a linear-time algorithm? Why or why not?

No. Time complexity is $O(n^2)$. Because in the worst case we need to search all rows(n rows) and for each one sequentialSearch runs in O(n) time.

4.2. Medium

15. An n-degree polynomial p(x) is an equation of the form:

$$p(x) = \sum_{i=0}^{n} a_i x^i$$

where x is a real number and each a_i is a constant.

a. Describe a simple $O(n^2)$-time method for computing p(x) for a particular value of x.

b. Describe a simple $O(n)$-time method for computing p(x) for a particular value of x.

c. Consider now a rewriting of p(x) as

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + xa_n) \cdots)))$$

which is known as **Horner's method**. Using the big-Oh notation, characterize the number of multiplications and additions this method of evaluation uses.

a)

```
Algorithm compute_polynomial_slow(a, x)
Input: array a of size n+1 containing coefficients
    result ← 0
    for i ← 0 to n do
      p=1;
      for j ← 0 to i-1 do
              p=p*x
      term ← a[i] * p
      result ← result + term
    return result
```

Time complexity is quadratic because of nested loops. The exact number of multiplications is: $\sum_0^n i = \frac{n(n+1)}{2}$ because to calculate each x^i we need to do i multiplications in the inner "for loop". The number of additions is n+1, because we have one addition in each iteration of outer "for loop". (and for loop iterates n+1 times)

b)

```
Algorithm compute_polynomial_fast(a, x)
Input: array a of size n containing coefficients
    result ← 0
    p ← 1
    for i ← 0 to n do
      term ← a[i] * p
      p←p*x
      result ← result + term
    return result
```

Time complexity is linear because we have constant number of operations inside the loop. The exact number of multiplications is: $2(n + 1)$ because there are two multiplications inside the for loop. The number of additions is n+1.

c)

```
Algorithm horner_evaluation(a, x)
    result ← 0
    for i ← n to 0 step -1 do
        result ← a[i] + x * result
    return result
```

Time complexity is linear because we have constant number of operations inside the loop. The exact number of multiplications is: n+1 because there is one multiplication inside the for loop. The number of additions is also n+1.

16. An array A contains n − 1 unique integers in the range [0, n − 1]; that is, there is one number from this range that is not in A. Design an O(n)-time algorithm for finding that number. You are allowed to use only O(1) additional space besides the array A itself.

To find the missing number in an array A, we can add all numbers up and compare the result with the sum of all numbers from 0 to n-1 to find the missing number. Obviously we are using O(1) additional space and the algorithm is linear.

```
Algorithm find_missing_number(A)
    n ← len(A) + 1
    sum_expected ← (n * (n-1))/ 2
    sum_actual ← 0
    for i=0 to n-2 do
        sum_actual ← sum_actual + A[i]
    missing_number ← sum_expected - sum_actual
    return missing_number
```

Note(cycle-sort method): There is another approach to solve this problem in linear time and O(1) additional space. In this approach we try to bring all numbers i to location i except for n-1 if A contains it. If n-1 is in A, it will be at the location of the missing element. If n-1 is not in A, we will return n-1.

```
Algorithm missing_number(A)
    n ← len(A)
    i ← 0
    result ← n-1
    while i < n do
        if A[i] == n-1 then
            result ← i
            i ← i + 1
        else if i == A[i] then
            i ← i + 1
        else
            index ← A[i]
            A[i] ← A[index]
            A[index] ← index
    return result
```

17. Suppose that each row of an n × n array A consists of 1's and 0's such that, in any row of A, all the 1's come before any 0's in that row. Assuming A is already in memory, describe a method running in O(n) time (not $O(n^2)$ time) for finding the row of A that contains the most 1's.

```
Algorithm find_max_ones_row(L,n)
    i ← 0
    res ← -1
    max ← -1
    for j ← 0 to n- 1 do
        while i < n and L[j][i] = 1 do
            i ← i + 1
        if i-1 > max then
            max ← i-1
            res ← j
    return res
```

The time complexity is linear despite having two nested loops. We start from the first row and find the first 0, then we go to the next row and if we are on a 0 we go to the next row. Otherwise we continue in that row to find 1. See the following figure. So in the worst case scenario, we are checking n elements since we never go back to the previous columns.

1	1	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	0
1	1	1	1	1	0
1	1	1	1	0	0

18. Describe a method for finding both the minimum and maximum of n numbers using fewer than 3n/2 comparisons.

```
Algorithm  maxmin(L)
min_val ← L[0]
max_val ← L[0]
n ← len(L)
if n % 2 = 0 then
    i ← 0
else
    i ← 1
while i < n - 1 do
    if L[i] > L[i + 1] then
        cmax ← L[i]
        cmin ← L[i + 1]
    else
        cmin ← L[i]
        cmax ← L[i + 1]
    if cmax > max_val then
        max_val ← cmax
    if cmin < min_val then
        min_val ← cmin
    i ← i + 2
return max_val, min_val
```

The while loop iterates n/2 times and we have 3 comparisons inside it.