THOMAS ADEWUMI
UNIVERSITY,
OKO, KWARA STATE
Science | Technology | Medicine

Faculty of Computing and Applied Sciences
Department of Mathematical and Computing Science

SWE 208 – Design and Analysis of Computer Algorithms

Credit Hours:	**2**
Contact Hours:	**24**
Status:	**Core**
Semester:	**Second**
Pre-requisite:	**Nil**

Lecturer: O. J. Olabode

Course Description

Algorithm design is the specification of the logical steps required in solving a computational problem in an efficient way with minimum time and memory space. Analysis of algorithms has to do with the processes of estimating the amount of resources required for implementing an algorithm. Since there can be more than one algorithm for the same problem, an analysis of each algorithm will enable one to make an informed decision in choosing a better of two algorithms for solving a similar problem.

Course Objectives

- To learn classic algorithms
- To device correct and efficient algorithms for solving a given problem
- To know how to express algorithms
- To know how to analyze algorithms
- To learn how to write clear algorithms

Learning Outcome

At the end of the course, students will be able to:

- Understand the meaning of an algorithm
- Know how to express algorithms
- Know how to analyze algorithms
- Demonstrate familiarity with major algorithms and data structures
- Design algorithms for some computational problems
- Analyze the asymptotic performance of algorithms

Course Content

- Introduction to algorithm and its importance
- Mathematical foundation: growth functions, complexity analysis of algorithms, summations, recurrences, sorting algorithms
- Algorithm design: divide-and-conquer approach, greedy approach
- Graph algorithms: graph searching, topological sort, minimum spanning tree, shortest paths, backtracking and its applications in games
- String matching
- Theory of NP-completeness
- Turing machines and halting problem

Lecture Notes 1

i. What Is an Algorithm and Why Are They Important
ii. What are the Importance of Computer Algorithms

What Is An Algorithm?

An algorithm is a set of step-by-step procedures, or a set of rules to follow, for completing a specific task or solving a particular problem. Algorithms are all around us. The recipe for baking a cake, the method we use to solve a long division problem, and the process of doing laundry are all examples of an algorithm. Here's what baking a cake might look like, written out as a list of instructions, just like an algorithm:

1. Preheat the oven
2. Gather the ingredients
3. Measure out the ingredients
4. Mix together the ingredients to make the batter
5. Grease a pan
6. Pour the batter into the pan
7. Put the pan in the oven
8. Set a timer
9. When the timer goes off, take the pan out of the oven
10. Enjoy!

Algorithmic programming is all about writing a set of rules that instruct the computer how to perform a task. A computer program is essentially an algorithm that tells the computer what specific steps to execute, in what specific order, in order to carry out a specific task. Algorithms are written using particular syntax, depending on the programming language being used.

Types of Algorithms

Algorithms are classified based on the concepts that they use to accomplish a task. While there are many types of algorithms, the most fundamental types of computer science algorithms are:

1. Divide and conquer algorithms – divide the problem into smaller sub-problems of the same type; solve those smaller problems, and combine those solutions to solve the original problem.
2. Brute force algorithms – try all possible solutions until a satisfactory solution is found.
3. Randomized algorithms – use a random number at least once during the computation to find a solution to the problem.
4. Greedy algorithms – find an optimal solution at the local level with the intent of finding an optimal solution for the whole problem.
5. Recursive algorithms – solve the lowest and simplest version of a problem to then solve increasingly larger versions of the problem until the solution to the original problem is found.
6. Backtracking algorithms – divide the problem into sub-problems, each which can be attempted to be solved; however, if the desired solution is not reached, move backwards in the problem until a path is found that moves it forward.
7. Dynamic programming algorithms – break a complex problem into a collection of simpler sub-problems, then solve each of those sub-problems only once, storing their solution for future use instead of re-computing their solutions.

Where are Algorithms Used in Computer Science?

Algorithms are used in every part of computer science. They form the field's backbone. In computer science, an algorithm gives the computer a specific set of instructions, which allows the computer to do everything, be it running a calculator or running a rocket. Computer programs are, at their core, algorithms written in programming languages that the computer can understand. Computer algorithms play a big role in how social media works: which posts show up, which ads are seen, and so on. These decisions are all made by algorithms. Google's programmers use algorithms to optimize searches, predict what users are going to type, and more. In problem-solving, a big part of computer programming is knowing how to formulate an algorithm.

Why are Algorithms Important to Understand?

Algorithmic thinking, or the ability to define clear steps to solve a problem, is crucial in many different fields. Even if we're not conscious of it, we use algorithms and algorithmic thinking all the time. Algorithmic thinking allows students to break down problems and conceptualize solutions in terms of discrete steps. Being able to understand and implement an algorithm requires students to practice structured thinking and reasoning abilities.

How to Write an Algorithm?

An algorithm can be thought of as the plan or blueprint for solving a problem. Some steps for creating an algorithm are as follows:

1. Determine the goal of the algorithm. This should explain what the algorithm will accomplish.
2. Analyze current and historical information pertaining to the problem at hand.

3. Create a rough algorithm that models the steps of the mathematical computation.

4. Begin fine-tuning the algorithm by refining the steps involved.

5. Continue running the algorithm to ensure its correctness.

6. If possible, prove the algorithm using a mathematical proof.

Sorting Algorithms

A sorting algorithm is an algorithm that puts elements of a list in a certain order, usually in numerical or lexicographical order. Sorting is often an important first step in algorithms for the students of private engineering colleges in Jaipur that solves more complex problems. There are a large number of sorting algorithms, each with their own benefits and costs. Below, we will focus on some of the more famous sorting algorithms.

1. **Linear sort** — Find the smallest element in the list to be sorted, add it to a new list, and remove it from the original list. Repeat this until the original list is empty.
2. **Bubble sort** — Compare the first two elements in the list, and if the first is greater than the second, swap them. Repeat this with every pair of adjacent elements in the list. Then, repeat this process until the list is fully sorted.
3. **Insertion sort** — Compare each element in the list to all the prior elements until a smaller element is found. Swap these two elements. Repeat this process until the list is fully sorted.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the below mentioned characteristics −

1. **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their input/outputs should be clear and must lead to only one meaning.
2. **Input** − An algorithm should have 0 or more well defined inputs.
3. **Output** − An algorithm should have 1 or more well defined outputs, and should match the desired output.
4. **Finiteness** − Algorithms must terminate after a finite number of steps.
5. **Feasibility** − Should be feasible with the available resources.
6. **Independent** − An algorithm should have step-by-step directions which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (*do, for, while*), flow-control (*if-else*) etc. These common constructs can be used to write an algorithm.

We write algorithms in step by step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example:

Sum of Two Numbers:

step 1 – START ADD

step 2 – get values of **a** & **b**

step 3 – c ← a + b

step 4 – display c

step 5 – STOP

Algorithm Analysis

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation, as mentioned below –

- ***A priori*** **analysis** – This is theoretical analysis of an algorithm. Efficiency of algorithm is measured by assuming that all other factors e.g. processor speed, are constant and have no effect on implementation.
- ***A posterior*** **analysis** – This is empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn here **a priori** algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. Running time of an operation can be defined as no. of computer instructions executed per operation.

Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the Algorithm X are the two main factors which decide the efficiency of X.

- **Time Factor** − The time is measured by counting the number of key operations such as comparisons in sorting algorithm
- **Space Factor** − The space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and / or storage space required by the algorithm in terms of n as the size of input data.

Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function **T(n)**, where **T(n)** can be measured as the number of steps, provided each step consumes constant time. For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is **T(N) = c*n**, where **c** is the time taken for addition of two bits. Here, we observe that **T(N)** grows linearly as input size increases.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. Space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example simple variables & constant used, program size etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example dynamic memory allocation, recursion stack space etc.

Space complexity **S(I)** of any algorithm **I** is **S(I) = C + bS(P)** Where b>C is the fixed part and **S(P)** is the variable part of the algorithm which depends on instance characteristic **P**.

Following is a simple example that tries to explain the concept:

Algorithm: SUM(A,B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B and C and one constant. Hence **S(I) = 1+3**. Now space depends on data types of given variables and constant types and it will be multiplied accordingly.

Example:

Now use an example to learn how to write algorithms

Problem: Create an algorithm that multiplies two numbers and display the output

Step 1 – Start
Step 2 – declare three integers x, y & z
Step 3 – define values of x & y
Step 4 – multiply values of x & y
Step 5 – store result of step 4 to z
Step 6 – print z
Step 7 – Stop

Algorithms instruct programmers on how to write code. In addition, the algorithm can be written as:

Step 1 – Start mul
Step 2 – get values of x & y
Step 3 – z ← x * y
Step 4 – display z
Step 5 – Stop

In algorithm design and analysis, the second method is typically used to describe an algorithm. It allows the analyst to analyze the algorithm while ignoring all unwanted definitions easily. They can see which operations are being used and how the process is progressing. It is optional to write step numbers. To solve a given problem, you create an algorithm. A problem can be solved in a variety of ways.

Recalling important mathematical background

- - Indices/exponents,
- - series, bounding summations,
- - proofs by mathematical induction

Lecture Notes 2

Classic Algorithm

- **Methods of expressing algorithms**
- **Algorithm Analysis – what and why?**
- **Measures of efficiency of an algorithm**
- **Four different algorithms for the same computational problem – finding the minimum of a set numbers**

Classic Algorithms

Classic algorithms are fundamental to computer science and problem-solving. Some well-known ones include:

1. **Binary Search**: Efficiently finds the position of a target value within a sorted array.
2. **Quicksort**: A fast sorting algorithm that uses divide and conquer.
3. **Merge Sort**: Another efficient sorting algorithm that also uses divide and conquer.
4. **Breadth-First Search (BFS**): Traverses a graph level by level.
5. **Depth-First Search (DFS):** Traverses a graph by going as far as possible along each branch before backtracking.
6. **Dijkstra's Algorithm:** Finds the shortest path between nodes in a graph with non-negative edge weights.
7. **Floyd-Warshall Algorithm:** Finds the shortest paths in a weighted graph with positive or negative edge weights.
8. **A Search Algorithm:** Finds the shortest path in a graph, using a heuristic to guide the search.

These are just a few examples, but there are many more classic algorithms with various applications in computer science and beyond.

Methods of expressing algorithms

There are several methods for expressing algorithms, each with its own advantages and best use cases:

These examples demonstrate how the same algorithm can be expressed using different methods, each suited to different purposes and audiences.

- **Natural Language**: Algorithms can be expressed in plain language, such as English, using step-by-step instructions. This method is often used for explaining algorithms to people who may not be familiar with programming languages or formal notation. While it's easy to understand, it can sometimes lack precision and may be open to interpretation.

e.g. "To find the maximum number in a list:

1. Start with assuming the first number in the list is the maximum.
2. Compare this number with each subsequent number in the list.
3. If the current number is greater than the assumed maximum, update the maximum to be this number.
4. Repeat step 2 and 3 until all numbers in the list have been checked.
5. The final assumed maximum is the maximum number in the list.“

- **Pseudocode**: Pseudocode is an informal, high-level description of the operating principle of a computer program or other algorithm. It uses a mixture of natural language and some syntax from a programming language to represent the logic of the algorithm without getting bogged down in the details of a specific programming language. Pseudocode is often used as a planning tool before writing actual code.

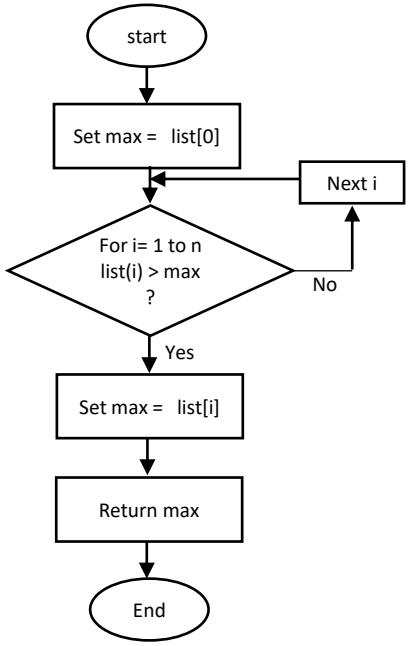
e.g. Function FindMaximum(list)

```
    max = list[0]
    for each number in list:
        if number > max:
            max = number
    return max
```

Flowcharts: Flowcharts are graphical representations of algorithms. They use different shapes to represent different types of steps or decisions, with arrows indicating the flow of control through the algorithm. Flowcharts are useful for visualizing the flow of an algorithm and can be helpful for understanding complex processes, especially for people who are more visually oriented.

e.g. To find maximum number in a list using flow chart

The flowchart consists of shapes like rectangles, diamonds, and arrows. The rectangle shapes represent process steps, the diamonds represent decision points, and the arrows indicate the flow of control. Here's a textual representation:



- **Structured English**: Structured English is a method of expressing algorithms using a limited subset of natural language along with a specific syntax for expressing control structures like loops and conditionals. It's similar to pseudocode but tends to be more formal and structured.

  e.g.

  Start

  Set max to the first number in the list

  For each number in the list

  If the number is greater than max

  Set max to the number

  End of loop

  Return max

  End

- **Programming Languages**: Ultimately, algorithms are implemented in programming languages to be executed by computers. Programming languages provide precise syntax and semantics for expressing algorithms in a way that can be directly executed by a computer. Examples of programming languages commonly used for expressing algorithms include Python, Java, C++, and many others.

e.g. in python

```
def find_maximum(lst):
    max_num = lst[0]
    for num in lst:
        if num > max_num:
            max_num = num
    return max_num
```

Algorithm Analysis

Algorithm analysis is the process of evaluating the efficiency and performance characteristics of algorithms. It involves studying how algorithms behave in terms of their time complexity, space complexity, and other factors such as their scalability and resource usage.

Here's a breakdown of key aspects of algorithm analysis:

- **Time Complexity**: Time complexity refers to the amount of time an algorithm takes to complete as a function of the size of its input. It's often expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time with respect to the input size. Time complexity analysis helps us understand how an algorithm's performance scales as the input size increases.

  For example, consider a linear search algorithm, which sequentially checks each element of the array until it finds the target element or reaches the end of the array. The time complexity of this algorithm is O(n), where n is the size of the array. This means that the time taken by the algorithm grows linearly with the size of the input array.

- **Space Complexity**: Space complexity refers to the amount of memory an algorithm requires to execute as a function of the size of its input. It considers factors such as the size of data structures used by the algorithm and any additional memory required for auxiliary variables. Like time complexity, space complexity is also expressed using Big O notation.

  For example, the space complexity of the linear search algorithm is O(1), meaning it requires a constant amount of extra space regardless of the size of the input array. This is because it only requires a few additional variables (e.g., loop counters, the target element) and does not depend on the size of the input array.

- **Asymptotic Analysis**: Asymptotic analysis focuses on how the performance of an algorithm scales with respect to the size of its input as the input size approaches infinity. It disregards constant factors and lower-order terms, focusing instead on the dominant term that determines the algorithm's growth rate. This provides a high-level understanding of an algorithm's efficiency and allows for comparisons between different algorithms.

  For example, the dominant term in the time complexity of the linear search algorithm is O(n), indicating that its performance grows linearly with the size of the input array. Asymptotically, the algorithm's performance is proportional to the size of the input.

- **Empirical Analysis**: In addition to theoretical analysis, empirical analysis involves running algorithms on actual inputs and measuring their performance in practice. This can provide insights into how well theoretical predictions match real-world performance and can help identify practical considerations such as implementation details and hardware constraints.

  Empirical analysis involves running the linear search algorithm on arrays of different sizes and measuring the time taken to find the target element. By plotting the input size against the execution time, we can observe the algorithm's empirical performance and verify whether it aligns with the theoretical time complexity analysis.

Why is algorithm analysis necessary?

Algorithm analysis is necessary for several reasons:

- **Efficiency**: Efficient algorithms are crucial for optimizing resource usage, such as time and memory. By analyzing algorithms, we can identify inefficiencies and improve them to make better use of resources.
- **Performance Comparison**: Algorithm analysis allows us to compare different algorithms and choose the most suitable one for a particular problem. Understanding the time and space complexity of algorithms helps in selecting the most efficient solution.
- **Scalability**: As the size of input data grows, the performance of algorithms may vary significantly. Algorithm analysis helps us understand how algorithms scale with input size, enabling us to predict their behavior for larger datasets.
- **Resource Constraints**: In many applications, resources like time and memory are limited. Algorithm analysis helps in designing algorithms that meet these constraints, ensuring that the solution remains feasible within available resources.
- **Problem-solving**: Analyzing algorithms deepens our understanding of problem-solving techniques. It helps us recognize common patterns and strategies, making it easier to devise solutions for new problems.
- **Optimization**: Algorithm analysis often leads to optimization opportunities. By identifying bottlenecks and inefficiencies, we can refine algorithms to improve their performance and reduce resource usage.
- **Real-world Applications**: In real-world applications, efficient algorithms can have a significant impact on user experience and system performance. Algorithm analysis ensures that algorithms meet performance requirements in practical scenarios.
- **Algorithm Design**: Understanding algorithm analysis principles informs the design of new algorithms. It provides insights into how to structure algorithms to achieve desired performance characteristics.
- In summary, algorithm analysis is necessary for optimizing algorithms, comparing different solutions, understanding scalability, meeting resource constraints, enhancing problem-solving skills, optimizing real-world applications, and guiding the design of new algorithms. It is a fundamental aspect of computer science and plays a crucial role in the development of efficient and effective software systems.

Measures of Efficiency of an Algorithm

- The efficiency of an algorithm can be measured using various metrics, including:
- **Time Complexity**: Time complexity measures the amount of time an algorithm takes to complete as a function of the size of its input. It provides an estimate of the worst-case, average-case, or best-case running time of the algorithm. Time complexity is often expressed using Big O notation, which describes the upper bound on the growth rate of the algorithm's running time.
- **Space Complexity**: Space complexity measures the amount of memory an algorithm requires to execute as a function of the size of its input. It considers factors such as the size of data structures used by the algorithm and any additional memory required for auxiliary variables. Like time complexity, space complexity is also expressed using Big O notation.
- **Auxiliary Space Complexity**: Auxiliary space complexity specifically measures the extra space required by an algorithm beyond the input size. It excludes space taken up by the input itself. Auxiliary space complexity is important for understanding the memory usage of an algorithm, especially in constrained environments.
- **Best, Worst, and Average Case Analysis**: Algorithms may exhibit different performance characteristics depending on the input they receive. Best-case analysis considers the minimum amount of time or space required by an algorithm for any input. Worst-case analysis considers the maximum amount of time or space required. Average-case analysis considers the expected performance over all possible inputs, often assuming some probability distribution for input data.
- **Asymptotic Analysis**: Asymptotic analysis focuses on how the performance of an algorithm scales with respect to the size of its input as the input size approaches infinity. It disregards constant factors and lower-order terms, focusing instead on the dominant term that determines the algorithm's growth rate. This provides a high-level understanding of an algorithm's efficiency and allows for comparisons between different algorithms.
- **Empirical Analysis**: In addition to theoretical analysis, empirical analysis involves running algorithms on actual inputs and measuring their performance in practice. This can provide insights into how well theoretical predictions match real-world performance and can help identify practical considerations such as implementation details and hardware constraints.

By considering these measures of efficiency, we can evaluate algorithms and make informed decisions about their suitability for specific tasks, taking into account factors such as time, memory, scalability, and practical performance.

Lecture Notes 3

Algorithm Analysis Overview

- **RAM model of computation**
- **Concept of input size**
- **Three complexity measures and their analyses**

RAM Model of Computation

- The RAM (Random Access Machine) model of computation is a theoretical framework used in computer science and algorithm analysis to study the efficiency and behavior of algorithms. It serves as a simplified abstraction of modern computer architectures and provides a common basis for analyzing algorithmic complexity.
- In the RAM model, computation is performed sequentially, and the memory is organized into a sequence of addressable cells, each capable of storing a single word of data. Here are some key aspects of the RAM model:
- **Instructions**: The RAM model supports a set of basic instructions or operations, such as arithmetic operations (addition, subtraction, multiplication, division), memory access (read, write), control flow (conditional branching, looping), and function calls.
- **Addressing**: Each memory cell in the RAM model is assigned a unique address, allowing direct access to any memory location in constant time. This property is referred to as "random access," meaning that any memory cell can be accessed directly without having to traverse other cells sequentially.
- **Arithmetic Operations**: Arithmetic operations in the RAM model typically have unit cost, meaning they are assumed to execute in constant time regardless of the size of the operands. This assumption simplifies the analysis of algorithms by focusing on the number of arithmetic operations performed.
- **Memory Model**: The RAM model assumes an unbounded amount of memory available for computation. This abstraction allows for the analysis of algorithms without being constrained by specific memory limitations, such as the size of physical RAM in a real computer system.

- **Complexity Analysis**: In the RAM model, the complexity of an algorithm is typically measured in terms of the number of basic operations executed, such as arithmetic operations, memory accesses, or comparisons. This provides a standardized framework for comparing the efficiency of algorithms independent of specific hardware architectures or programming languages.

While the RAM model provides a useful theoretical framework for analyzing algorithms, it is important to recognize its simplifications and limitations. In practice, real-world computing systems may have additional complexities, such as caching, pipelining, and parallelism, which can affect the performance of algorithms differently. Nonetheless, the RAM model remains a valuable tool for theoretical analysis and understanding algorithmic complexity.

Concept of Input Size

The concept of input size refers to the measure of the amount of data or the characteristics of the input that an algorithm processes. It's a fundamental aspect of algorithm analysis as the performance of an algorithm often depends on the size of the input it operates on.

The input size can vary depending on the problem being solved and the specific representation of the input data. Here are some examples of how input size can be defined for different types of problems:

- **Arrays and Lists**: For algorithms that operate on arrays or lists, such as sorting or searching algorithms, the input size typically corresponds to the number of elements in the array or list.
- **Graphs**: For algorithms that operate on graphs, such as graph traversal or shortest path algorithms, the input size can be defined in terms of the number of vertices (nodes) and edges in the graph.
- **Strings**: For algorithms that process strings, such as pattern matching or string manipulation algorithms, the input size can be defined in terms of the length of the string.
- **Matrices**: For algorithms that operate on matrices, such as matrix multiplication or matrix inversion algorithms, the input size can be defined in terms of the dimensions of the matrix (number of rows and columns).
- **Complex Data Structures**: For algorithms that operate on complex data structures, such as trees or heaps, the input size may depend on various factors, including the depth, breadth, or other characteristics of the data structure.

When analyzing the performance of algorithms, it's important to consider how the input size affects the algorithm's time complexity, space complexity, and other performance characteristics. Typically, algorithm analysis involves studying how these characteristics scale with respect to changes in the input size. This helps in understanding the efficiency and scalability of algorithms and in making informed decisions about algorithm selection and optimization.

Complexity Analysis and Their Measures

Complexity measures are used to analyze algorithms and understand their performance characteristics. Here are some common complexity measures and how they are analyzed:

- **Time Complexity**: Time complexity measures the amount of time an algorithm takes to run as a function of the size of its input. It is typically expressed using Big O notation, which provides an upper bound on the growth rate of the algorithm's running time. To analyze time complexity:
  - Identify the basic operations performed by the algorithm.
  - Determine the number of times each basic operation is executed in terms of the input size.
  - Express the total number of basic operations as a function of the input size.
  - Simplify the expression and identify the dominant term to determine the overall time complexity.
- **Space Complexity**: Space complexity measures the amount of memory (space) an algorithm requires to run as a function of the size of its input. It is also expressed using Big O notation. To analyze space complexity:
  - Identify the memory used by the algorithm, including data structures, auxiliary variables, and recursive calls.
  - Determine how the memory usage grows with the input size.
  - Express the total memory usage as a function of the input size.
  - Simplify the expression and identify the dominant term to determine the overall space complexity.
- **Worst-Case, Best-Case, and Average-Case Analysis**: Algorithms may perform differently depending on the characteristics of the input data. Analyzing worst-case, best-case, and average-case scenarios helps understand the algorithm's behavior under different conditions:
  - Worst-case analysis determines the maximum time or space required for any input.
  - Best-case analysis determines the minimum time or space required for any input.
  - Average-case analysis considers the expected performance over all possible inputs, often using probability distributions.

- **Amortized Analysis**: Some algorithms have variable performance depending on the sequence of operations rather than just the input size. Amortized analysis provides an average cost per operation over a sequence of operations. It is used to analyze algorithms with alternating expensive and cheap operations.
- **Average-Case Analysis with Randomized Algorithms**: Some algorithms use randomness to achieve better average-case performance. Analyzing the average-case complexity of randomized algorithms involves considering the probability distributions of inputs and the expected performance over all possible random outcomes.
- **Empirical Analysis**: In addition to theoretical analysis, empirical analysis involves running algorithms on actual inputs and measuring their performance in practice. It provides insights into how well theoretical predictions match real-world performance and can help identify practical considerations such as implementation details and hardware constraints.

By using these complexity measures and analysis techniques, we can gain a deeper understanding of algorithmic efficiency, scalability, and behavior under different conditions. This understanding is essential for designing, analyzing, and optimizing algorithms for various problem domains.

Lecture Notes 4

Asymptotic Analysis (AA) of an Algorithm

- **Meaning of AA**
- **Motivation for AA**
- **Analyzing algorithms using AA**
- **Asymptotic notation**
- **Standard complexity classes: big “O”, little “o”**

Meaning of Asymptotic Analysis

Asymptotic analysis is a method used to evaluate the behavior of algorithms as the size of their input approaches infinity. It focuses on understanding how an algorithm's performance scales with the size of its input. The key idea is to identify the most significant factors affecting the algorithm's efficiency and to ignore less significant details.

The primary goal of asymptotic analysis is to describe the limiting behavior of an algorithm's time or space complexity as the input size grows arbitrarily large. This allows us to make general statements about the efficiency and scalability of an algorithm without getting bogged down in the specifics of individual cases.

Key Aspects of Asymptotic Analysis

- **Focus on Dominant Terms**: Asymptotic analysis focuses on identifying the dominant term or terms in the expression representing an algorithm's time or space complexity. The dominant term is the one that grows the fastest as the input size increases and determines the overall behavior of the algorithm.
- **Big O Notation**: Asymptotic analysis often uses Big O notation to express the upper bound on the growth rate of an algorithm's complexity. Big O notation provides a concise way to describe how an algorithm's performance scales with input size. For example, O(n^2) represents quadratic growth, O(n log n) represents near-linearithmic growth, and O(1) represents constant time complexity.
- **Worst-Case Analysis**: Asymptotic analysis typically focuses on worst-case time or space complexity, which represents the maximum resources an algorithm requires for any input. Worst-case analysis provides a conservative estimate of an algorithm's performance and ensures that the algorithm behaves acceptably even in the most unfavorable scenarios.
- **Average-Case and Best-Case Analysis**: While worst-case analysis is common in asymptotic analysis, algorithms may perform differently under different conditions. Average-case and best-case analyses consider the expected or minimum performance of an algorithm over all possible inputs. These analyses provide additional insights into an algorithm's behavior but may be more complex to determine.
- **Simplifications**: Asymptotic analysis often simplifies the analysis by ignoring constant factors, lower-order terms, and specific details of the algorithm's implementation. This simplification allows for a focus on the fundamental characteristics of the algorithm's efficiency and scalability.

Overall, asymptotic analysis provides a powerful tool for comparing and evaluating algorithms, understanding their efficiency, scalability, and behavior under different conditions, and making informed decisions about algorithm selection and optimization. It is widely used in computer science and algorithm analysis to analyze the performance of algorithms in a rigorous and systematic manner.

Motivation for Asymptotic Analysis

Asymptotic analysis is a powerful tool used in mathematics and computer science for understanding the behavior of functions as their input grows towards infinity or some other limit. There are several motivations for studying asymptotic analysis:

- **Simplicity**: Asymptotic analysis allows us to simplify complex functions by focusing on their dominant behavior as the input size grows large. This simplification makes it easier to understand and analyze the overall performance or behavior of algorithms, especially in cases where precise analysis is impractical or impossible.
- **Algorithmic efficiency**: Asymptotic analysis helps in evaluating the efficiency of algorithms in terms of their time and space complexity. By understanding how the resource requirements of an algorithm grow with input size, we can compare different algorithms and choose the most efficient one for a given problem.
- **Algorithm design**: Asymptotic analysis provides insights into the design of algorithms. It helps in identifying inefficiencies and bottlenecks in algorithm implementations, leading to the development of improved algorithms with better performance characteristics.
- **Predictive power**: Asymptotic analysis allows us to make predictions about the behavior of algorithms or functions for very large inputs without having to actually execute them. This predictive power is valuable for estimating the scalability of algorithms and understanding how they will perform as the problem size increases.
- **Optimization**: By understanding the asymptotic behavior of functions, we can often identify opportunities for optimization. For example, we may discover that certain parts of an algorithm have negligible impact on overall performance and can be optimized away, leading to faster or more efficient implementations.

Overall, asymptotic analysis plays a crucial role in both theoretical and practical aspects of mathematics and computer science, helping researchers and practitioners understand, analyze, and optimize algorithms and functions.

Analyzing Algorithm Using Asymptotic Analysis

Lecture Notes 5

Order of Growth of Some Important Functions

- **Logarithm function**
- **Polynomial function**
- **Exponential function**
- **Factorial function**
- **Asymptotic efficiency classes**

What Is An Algorithm?

An algorithm is a set of step-by-step procedures, or a set of rules to follow, for completing a specific task or solving a particular problem. Algorithms are all around us. The recipe for baking a cake, the method we use to solve a long division problem, and the process of doing laundry are all examples of an algorithm. Here's what baking a cake might look like, written out as a list of instructions, just like an algorithm: