# **1B Java I/O Fundamentals**

**Primitive Data Types**

1. byte: 8-bit signed integer.
2. short: 16-bit signed integer.
3. int: 32-bit signed integer.
4. long: 64-bit signed integer.
5. float: Single-precision 32-bit floating point.
6. double: Double-precision 64-bit floating point.
7. char: 16-bit Unicode character.
8. Boolean Represents a value of true or false.

**Text Files:** The lecture defines a text file as a sequence of characters organized into lines, with each line terminated by a newline character (\n). All data in a text file, including numeric data, is stored in textual form. The file is terminated with an End-of-File (EOF) character.

**I/O Streams:** Java utilizes streams as the primary mechanism for input and output. An application *receives* data through an **input stream** and *sends* data through an **output stream**. Streams provide a consistent model for interacting with various data sources and destinations.

**Byte Streams vs. Character Streams:** The lecture introduces the distinction between byte and character streams. Byte streams work with 8-bit bytes, while character streams handle character values using Unicode.

**Byte streams** are considered low-level and are primarily used for the most primitive I/O. All other stream types are built upon byte streams. InputStream and OutputStream are the base classes for byte streams. Examples include FileInputStream and FileOutputStream.

**Character streams** are preferred for handling text data because they automatically handle the translation between Unicode and local character sets. Reader and Writer are the base classes for character streams. Examples include **FileReader** and **FileWriter.**

**Stream Types & Usage**

**Byte Streams:**

Byte streams are suitable for binary data and primitive I/O. The example provided, CopyBytes, demonstrates reading from FileInputStream and writing to FileOutputStream byte by byte, using an int to store a byte value (using the last 8 bits).

**Example:** The CopyBytes program copies the content of "xanadu.txt" to "outagain.txt" byte-by-byte.

FileInputStream in = null;

FileOutputStream out = null;

*try {*

*in = new FileInputStream("xanadu.txt");*

*out = new FileOutputStream("outagain.txt");*

*int c;*

*while ((c = in.read()) != -1) {*

*out.write(c);*

*}*

*} finally { // Ensure streams are closed even if an error happens*

*if (in != null) {*

*in.close();*

*}*

*if (out != null) {*

*out.close();*

*}*

*}*

**Important Note:** Closing streams, particularly within a finally block, is critical to prevent **resource leaks.**

**Quote:** "Always Close Streams. Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs."

**Character Streams:**

Character streams handle Unicode characters and automatically convert between Unicode and the local character set making them ideal for text files. The example provided, CopyCharacters, reads from FileReader and writes to FileWriter character by character, storing a character value in the last 16 bits of an int.

Character streams can be "wrappers" around byte streams. For example FileReader uses FileInputStream and FileWriter uses FileOutputStream. InputStreamReader and OutputStreamWriter bridge byte streams to character streams.

**Quote:** "Character streams are often 'wrappers' for byte streams... handles translation between characters and bytes."

**Line-Oriented I/O:**

Character streams can operate on lines rather than just individual characters. BufferedReader and PrintWriter are introduced for line-based input/output. The CopyLines example demonstrates reading lines using BufferedReader.readLine() and writing lines using PrintWriter.println().

**Quote:** "Character I/O usually occurs in bigger units than single characters. One common unit is the line... A line terminator can be a carriage-return/line-feed sequence ('\r\n'), a single carriage-return ('\r'), or a single line-feed ('\n')."

**Example:** CopyLines reads and writes whole lines of text

BufferedReader inputStream = null;

PrintWriter outputStream = null;

*try {*

*inputStream = new BufferedReader(new FileReader("xanadu.txt"));*

*outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));*

*String l;*

*while ((l = inputStream.readLine()) != null) {*

*outputStream.println(l);*

*}*

*} finally {*

*if (inputStream != null) {*

*inputStream.close();*

*}*

*if (outputStream != null) {*

*outputStream.close();*

*}*

*}*

**Buffered Streams**

* **Efficiency:** Buffered streams improve efficiency by reducing direct interaction with the underlying operating system and hardware.
* **Mechanism:** Buffered streams utilize a memory buffer to temporarily hold data. Buffered input streams read from the buffer; when the buffer is empty, the native I/O API is called. Buffered output streams write to the buffer; when the buffer is full, the native I/O API is called.
* **Wrapping:** Unbuffered streams can be converted to buffered streams using the wrapping pattern (passing an unbuffered stream to the constructor of a buffered stream). Example:

inputStream = new BufferedReader(new FileReader("xanadu.txt"));

outputStream = new BufferedWriter(new FileWriter("characteroutput.txt"));

* **Classes:** Four buffered stream classes: BufferedInputStream, BufferedOutputStream, BufferedReader, and BufferedWriter.
* **Flushing:** Buffers can be manually flushed using the flush() method, or certain events can trigger an automatic flush (e.g., PrintWriter.println() with autoflush).
* **Quote:** "It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer."
* **Buffer Definition:** A buffer is a region of memory used to temporarily hold data while it's being moved, usually when there is a difference between data rates, preventing data loss or poor bandwidth utilization.

**Error Handling (IOException)**

**Checked Exception:** I/O operations can throw IOException, a checked exception. This means that any code that may throw or propagate an IOException must either catch it or list it in the method's throws clause.

**Common Causes:** IOException can occur due to various issues:

File doesn't exist.

File cannot be found.

File doesn't contain the expected data type.

**Error Handling:** Code blocks are wrapped in try-catch blocks.

*try {*

*FileReader fr = new FileReader("/…/test.txt");*

*BufferedReader br = new BufferedReader(fr);*

*String s = br.readLine();*

*while (s!=null) { // not end of file??*

*System.out.println(s);*

*s = br.readLine();*

*}*

*br.close(); // close buffered reader stream*

*fr.close(); // close the file stream*

*} catch (IOException e) { // catch error*

*System.out.println(e); // print error message*

*}*

* **Quote:** "An IOException is a checked exception which means it either must be caught by a method, or must be listed in the throws clause of any method that may throw or propagate it."

**Text File Input and Output (Practical Examples)**

**Reading from a file:** Examples show how to use FileReader, BufferedReader, and readLine() to read text files line by line. The examples also show how to convert the read string value to integer using Integer.parseInt().

Example: Reading integers from a file:

*FileReader fr = new FileReader(fileName);*

*BufferedReader br = new BufferedReader(fr);*

*String c = br.readLine();*

*while (c != null) { // while not end of file*

*int num = Integer.parseInt(c);*

*System.out.println(num);*

*c = br.readLine();*

*}*

*br.close();*

*fr.close();*

**Writing to a file:** The lecture shows how to write data to files using FileWriter, BufferedWriter, and PrintWriter. It covers writing character-by-character, using write() method, or using println() for writing entire lines of text including strings and numbers.

Example of writing a sequence of numbers (1-10) to a file, each on its own line:

*FileWriter fw = new FileWriter(“test.txt”);*

*BufferedWriter bw = new BufferedWriter(fw);*

*PrintWriter pw = new PrintWriter(bw);*

*for (int i=1; i <= 10; i++) {*

*pw.println(i);*

*}*

*pw.close(); bw.close(); fw.close();*

*Handling Unknown Number of Elements: The standard pattern for reading a file when the number of lines (or elements) is not known in advance is to read until readLine() returns null (end of file).*

*Reusable Method: The material includes how to create reusable method to write an integer array to a file:*

*public static void writeToFile ( int[] nums, String fname){*

*int total = 0;*

*try {*

*FileWriter fw = new FileWriter(fname);*

*BufferedWriter bw = new BufferedWriter(fw);*

*PrintWriter pw = new PrintWriter(bw);*

*for (int i = 0; i < nums.length; i++) {*

*pw.println(nums[i]);*

*}*

*pw.close();*

*bw.close();*

*fw.close();*

*} // end*

*catch (IOException e) {*

*System.out.println(e);*

*}*

*}*

**Multiple File Manipulation**

The lecture emphasizes that it is often necessary to work with multiple files simultaneously, such as reading from several files and writing to others. The scenario mentioned is reading from two files, "electric.txt" and "gas.txt," and writing to a "total.txt" file.

The final task is to create two files, “electric.txt” and “gas.txt”, populate them with random values, and write the sum of corresponding values from each to a file called Total.txt.

# **1B Quiz**

1. **What is a text file, and how is it structured in terms of lines?**

A text file is a sequence of characters organized into lines, where each line is terminated by a new-line character (\n). All data, including numbers, is stored in textual form within these files.

1. **Explain the difference between byte streams and character streams in Java. Why is it generally preferred to use character streams when dealing with text files?**

Byte streams operate on 8-bit bytes and are used for low-level I/O, while character streams handle Unicode character data, taking care of the translation between internal formats and local character sets. Character streams are generally preferred for text files because they handle character encoding automatically and adapt to different locales.

1. **Describe the purpose of input streams and output streams in the context of file processing. Provide examples of classes used for each.**

Input streams are used by an application to receive data, reading data from a source such as a file or the keyboard. Output streams are used to send data to a destination such as a file or the screen. FileInputStream and FileReader are examples of input stream classes, while FileOutputStream and FileWriter are examples of output stream classes.

1. **What is the importance of using a finally block when working with I/O streams? Explain why closing streams is essential.**

The finally block is used to guarantee that resources like I/O streams are closed, regardless of whether errors occur. This prevents resource leaks, where open streams may tie up system resources. It ensures that close() is always called.

1. **What is the function of the read() method in FileInputStream and FileReader? How do the return types of each method differ?**

The read() method in FileInputStream reads a single byte from the input stream, returning an int that represents the byte read (or -1 if the end of the stream is reached). FileReader also reads one character at a time, which is also represented as an int that can be cast to a character, and will also return -1 at the end of the stream.

1. **Explain what buffered I/O is and why it is more efficient than unbuffered I/O. Name two buffered stream classes used for byte streams and two for character streams.**

Buffered I/O uses a buffer, which is a region of memory, to store data temporarily. Instead of direct interaction with the OS, buffered streams read from or write to a memory buffer, reducing the overhead of multiple I/O requests. BufferedInputStream and BufferedOutputStream are used for byte streams, while BufferedReader and BufferedWriter are used for character streams.

1. **What does it mean to "flush" a buffer? When is it useful to manually flush a buffer, and why are some stream classes set to "autoflush"?**

To "flush" a buffer means to write any data remaining in the buffer to its destination immediately. It is useful to manually flush a buffer to ensure that data is written to the destination at critical points. Some output classes, such as PrintWriter, have an autoflush option that will cause the buffer to be flushed with every println or format method call.

1. **Explain the role of the BufferedReader class and how the readLine() method works for reading lines of text.**

The BufferedReader class is used for efficient reading of text from a character-input stream, using a buffer. The readLine() method reads a line of text, terminated by a newline character (\n), and returns a String that contains the line (or null if the end of the stream is reached).

1. **What is an IOException, and why is it considered a checked exception? Describe how to handle an IOException.**

An IOException is a checked exception thrown by I/O operations that fail or are interrupted, indicating problems with file operations or communication. It is checked, meaning it must be caught or declared in the method's throws clause. You can handle an IOException by enclosing I/O code in a try block and catching it in a catch block.

1. **Describe the process of converting a string read from a file into an integer value. What method is used for this?**

To convert a string read from a file to an integer, you must first read the number as a String, and then use the Integer.parseInt() method, which takes a String argument and returns the parsed integer value.

# 1B Glossary

**Text File:** A file where all data, including numbers, is stored in character or textual form. Organized into lines, with each line terminated by a newline character.

**Byte Stream:** A stream that handles input and output of 8-bit bytes. Used for low-level I/O, often as a base for other streams. Classes descend from InputStream and OutputStream.

**Character Stream:** A stream that handles input and output of character data. Manages character encoding and translation. Classes descend from Reader and Writer.

**Input Stream:** A stream used by an application to receive data from a source, such as a file or keyboard.

**Output Stream:** A stream used by an application to send data to a destination, such as a file or screen.

**EOF (End-of-File):** A special invisible character that marks the end of a file, indicating there is no more data to read.

**FileInputStream:** A byte stream class used for reading byte data from a file.

**FileOutputStream:** A byte stream class used for writing byte data to a file.

**FileReader:** A character stream class used for reading character data from a file.

**FileWriter:** A character stream class used for writing character data to a file.

**BufferedReader:** A character stream class that reads character data from a buffer, improving efficiency when reading text.

**BufferedWriter:** A character stream class that writes character data to a buffer, improving efficiency when writing text.

**PrintWriter:** A character stream class that provides print and println methods for formatting and writing data, and can have autoflush features.

**readLine():** A method of BufferedReader that reads an entire line of text, up to the next newline character, from a stream.

**Buffer:** A region of memory used to temporarily store data while it is being moved between a source and a destination.

**Buffered I/O:** Using buffers to improve the efficiency of I/O operations by reducing the number of direct interactions with the underlying OS.

**Flushing:** The act of forcing the contents of a buffer to be written to its destination, even if the buffer is not full.

**Autoflush:** An option for some buffered output classes that automatically flush the buffer when certain events occur, such as println() or format() methods being invoked.

**IOException:** A checked exception thrown by I/O operations that fail, such as file not found, read error, or write error.

**Integer.parseInt():** A method of the Integer class that parses a String value as a signed decimal integer and returns the corresponding int.

FAQ:

**1. What are the primary benefits of using files for data storage in Java applications?**

Files provide a way to store data persistently, even after an application terminates. This allows applications to read data from a file when needed and to save data for future use. Text files, where data is stored in character form, are particularly useful as they are human-readable and can be easily created and edited using standard text editors.

**2. What are I/O streams and how do they work in Java?**

I/O streams are a fundamental concept for handling data flow in Java applications. An input stream is used to read data from a source (like a file or keyboard) into an application, and an output stream is used to write data from an application to a destination (like a file or screen). Streams present a simple model of a sequence of data, allowing a program to read or write one item at a time, irrespective of how they work internally.

**3. What are the key differences between byte streams and character streams? When should each be used?**

Byte streams handle I/O in terms of 8-bit bytes and are descended from InputStream and OutputStream classes. Character streams handle I/O in terms of 16-bit Unicode characters and are descended from Reader and Writer classes. Character streams are preferred when dealing with text because they automatically handle character encoding, adapting to the local character set and are ready for internationalization. Byte streams are used for more primitive I/O tasks. All other stream types are built on byte streams.

**4. How are FileInputStream, FileOutputStream, FileReader, and FileWriter used for file I/O?**

FileInputStream and FileOutputStream are byte stream classes used for reading and writing bytes to files respectively. FileReader and FileWriter are character stream classes for reading and writing characters to files. These classes provide a basic way to read/write from files. However, these classes often are wrapped by other classes like BufferedReader and PrintWriter.

**5. What role do BufferedReader, BufferedWriter, and PrintWriter play in I/O operations?**

BufferedReader is used to read text from a character input stream, buffering characters to make reading more efficient by reducing calls to the underlying system. BufferedWriter performs similarly for writing data to output streams. PrintWriter provides convenient methods like print and println for writing various types of data to a character output stream, often a buffered output stream, making it easier to write different types of data into the file.

**6. Why is it important to close streams after use, and how can you ensure that they are closed even if errors occur?**

Closing streams is crucial to avoid resource leaks, particularly with file I/O. If streams are not closed, resources used by the operating system could remain locked and unavailable. Java uses the finally block to ensure that streams are always closed, even if exceptions are thrown during I/O operations. This is especially important as file opening can result in an IOException if the file does not exist, is not accessible, or there is some other problem. The stream variable may be null and so it is important to check whether a stream object exists before attempting to close it.

**7. What is buffering and why is it important in I/O operations?**

Buffering is a technique where data is temporarily held in a memory region called a buffer. Buffered streams minimize direct interaction with the operating system for each read or write by working with the buffer instead. This is much more efficient than unbuffered I/O, as disk access or network activity is slow. Buffered input streams read data into a buffer, and the underlying operating system API is called only when the buffer is empty. Similarly buffered output streams write data into a buffer, and the underlying system API is called only when the buffer is full.

**8. How can programs handle IOException during I/O operations and what types of errors do these exceptions typically indicate?**

IOException is a checked exception that must be caught or declared in the throws clause of a method. It signals that an I/O error has occurred such as file not found, unable to open the file for writing or reading, or the file is the wrong format. Proper exception handling can prevent program crashes and gives users useful error messages. By implementing a try-catch block around the code, it ensures that these errors can be handled and outputted to the user.

# **2A Data Structures**

**Big O Notation**

**Definition:** **Big O notation** is a mathematical notation used in computer science to describe the *asymptotic behavior* of functions. Essentially, it tells you how the runtime or space requirements of an algorithm *scale* as the input size grows. It is "a symbolism used in complexity theory, computer science, and mathematics to describe the asymptotic behavior of functions."

**Purpose:** It's used for approximating formulas for analysis of algorithms, and for the definitions of terms in complexity theory (e.g. polynomial time). It helps analyze the efficiency of algorithms. It's about *complexity* (how resource requirements scale) rather than *performance* (actual resource usage on a specific machine). "Complexity affects performance but not the other way around."

**Ignoring Constants and Lower-Order Terms:** Big O notation focuses on the dominant term and ignores constants and lower-order terms because, for sufficiently large input sizes, these become insignificant.

**Example: T(n) = 4n^2 - 2n + 2 is simplified to O(n^2).**

**Common Classes:**

* **O(1): Constant time**
* **O(log n): Logarithmic time**
* **O(n): Linear time**
* **O(n^2): Quadratic time**
* **O(n^c): Polynomial time**
* **O(c^n): Exponential time**

**Formal Definition:** T(N) is O(F(N)) if for some constant c and for values of N greater than some value n0: T(N) <= c \* F(N). This means F(N) is an upper bound on the complexity.

**Little o Notation:** f(x) = o(g(x)) means that f grows much slower than g and is insignificant in comparison.

**Other Related Notations:** The document also introduces Omega (Ω), Theta (Θ), and little omega (ω) notations for more precise analysis, but emphasizes that Big O is the most common.

**Practical Application:Sequence of Statements:** Adding the times for all statements. Simple statements are O(1).

**If-Then-Else:** Worst-case time is the slower of the two blocks (max(time(block 1), time(block 2))).

**Loops:** for I in 1 .. N loop ... end loop; is O(N) if the statements inside are O(1).

**Nested Loops:** for I in 1 .. N loop for J in 1 .. M loop ... end loop; end loop; is O(N\*M). If M = N, then it's O(N^2).

**Function/Procedure Calls:** The complexity of the call is included in the overall complexity.

**Arrays**

**Definition:** An array is an indexed collection of data elements of the same type, stored in consecutive memory locations. "An array is an indexed collection of data elements of the same type."

**Indexing:** Array elements are numbered, typically starting at 0.

**Primitive Types & Memory:** Arrays of primitive types (byte, short, int, long, float, double, char, boolean) store the values directly. A Java byte is 8 bits, and an int is 4 bytes (32 bits). The source material notes, "I'm going to say that the strangest thing about the Java platform is that the byte type is signed.“ This was a language design choice.

**Area Overhead**: The slides display the memory overhead of an array containing 100 integers, and an array containing 100,000 integers.

**Big O Notation of Array Operations:Access (Read):** O(1) - Fastest operation because of contiguous memory.

"arr[5] directly retrieves the value at index 5 in O(1) time."

**Search:** O(n) - Linear search in the worst case (unsorted array). O(log n) if sorted (using binary search).

"Searching for X in [10, 20, 30, 40, 50] might require looking at every element."

**Insertion:** O(1) at the end (if space available), O(n) at the beginning or middle (requires shifting elements).

"To insert at index 0, all elements must shift right → O(n)."

**Deletion:** O(1) from the end, O(n) from the beginning or middle (requires shifting elements).

"Deleting arr[0] shifts all elements left → O(n)."

**Space Complexity:** O(n)

**Arrays as Objects (Java):** In Java, arrays are objects, and the array name is an object reference variable. Arrays must be instantiated using 'new'.

**Pass-by-Reference (Arrays in Java):** When an array is passed as a parameter to a method, the reference to the array is passed. Changes to array elements within the method *do* affect the original array.

**Linked Lists**

**Definition:** A linked list is a dynamic data structure where elements (nodes) are linked using pointers, not contiguous memory.

**Big O Notation of Linked List Operations:Access (Read):** O(n) - Requires traversing the list from the head.

**Search:** O(n) - Linear search.

**Insertion:** O(1) at the beginning (if head pointer is available). O(1) at the end (if tail pointer exists). O(n) in the middle.

**Deletion:** O(1) from the beginning. O(1) from the end (if doubly linked list with tail pointer). O(n) from the middle.

**Space Complexity:** O(n) - Each node requires extra memory for pointers.

**When to Use Arrays vs. Linked Lists:Arrays:** Fast access by index, memory efficiency, fixed or rarely changing size. "You need fast access to elements using an index (O(1))."

**Linked Lists:** Frequent insertions/deletions, unknown size in advance, memory fragmentation is not an issue. "You need frequent insertions/deletions, especially at the beginning or middle."

**Two-Dimensional Arrays**

**Definition:** An array of arrays, often visualized as a table with rows and columns.

**Declaration (Java):** int[][] scores = new int[2][4]; (2 rows, 4 columns).

**Access:** Elements are referenced using two index values: int value = scores[1][3];

**"Row Major" (Java):** Java treats 2D arrays as "array of arrays", so it processes rows first.

**Length:** data.length gives the number of rows; data[0].length gives the number of columns.

**Processing:** Typically done with nested for loops.

for (int r = 0; r < data.length; r++) {

for (int c = 0; c < data[0].length; c++) {

data[r][c] = 0;

}

}

**HashSets**

**Definition:** A set of values with no duplicate elements.

**HashSet Implementation:** Implemented using a hash table, providing fast performance for insertion, deletion, and lookups. The underlying data structure is a HashMap.

**Basic Operations:** add, remove, contains, isEmpty, size, clear, iterator.

**Unordered:** Elements are not sorted.

**Efficiency:** Optimized for searching.

**Big O Notation (HashSet Operations):** The time complexity for adding, removing, and searching in a HashSet is O(1) (constant time) on average.

**Hashing:** Information is stored by using a mechanism called hashing where a key is transformed into its hash code. "In hashing, the informational content of a key is used to determine a unique value, called its hash code." The hash code is then used as the index at which the data is stored.

**Hash Function Properties:** Deterministic, fast to compute, low collision rate, distributes keys evenly.

**Collision Resolution:** Techniques like linear probing and quadratic probing are used to handle collisions. "Techniques exist for collision resolution such as: Linear probing; Quadratic probing"

**Java Usage:**Use generics: Set<Integer> numbers = new HashSet<>();

Adding elements: names.add("Tom");

Removing elements: names.remove("Mary");

Checking if empty: names.isEmpty()

Getting size: name.size()

Checking if contains: names.contains("Mary")

Iterating: Using an Iterator.

**Set Operations:Subset:** s1.containsAll(s2) (checks if s2 is a subset of s1).

**Union:** s1.addAll(s2) (transforms s1 into the union of s1 and s2).

**Intersection:** s1.retainAll(s2) (transforms s1 into the intersection of s1 and s2).

**Set Difference:** s1.removeAll(s2) (transforms s1 into the (asymmetric) set difference of s1 and s2).

This briefing document provides a solid overview of the main concepts covered in the provided resources.

# 2A Quiz

**Question 1: Big O Notation**

Which of the following is NOT a use of Big O notation?

a) Approximating formulas for analysis of algorithms

b) Defining terms in complexity theory

**c) Describing the actual time used when a program is run**

d) Describing the asymptotic behavior of function

**Question 2: Array Time Complexity**

What is the time complexity for accessing an element (read operation) in an array?

a) O(n)

b) O(log n)

**c) O(1)**

d) O(n2)

**Question 3: Linked List vs. Array**

When should you prefer using a linked list over an array?

a) When you need fast access to elements using an index

b) When memory efficiency is crucial

**c) When you need frequent insertions/deletions, especially at the beginning or middle**

d) When the size is fixed or changes rarely

**Question 4: Two-Dimensional Arrays**

How do you declare a 2D array of integers with 2 rows and 4 columns in Java?

a) int[][] a = new int[10][11];

b) int[][] a = new int[10, 11];

c) int[11][10] a = new int[][];

**d) int[][] a = new int[11][10];**

**Question 5: HashSets**

Which of the following statements about HashSets is true?

a) HashSets maintain the order of elements

b) HashSets allow duplicate elements

**c) HashSets are implemented using a hash table**

d) The time complexity for adding, removing, and searching in a HashSet is O(n)

**Question 6: HashSet Operations**

Given two HashSets, set1 and set2, which operation transforms set1 into the intersection of set1 and set2?

a) set1.addAll(set2)

b) set1.removeAll(set2);

**c) set1.retainAll(set2);**

d) set1.containsAll(set2);

# 2A Glossary

**Algorithm Complexity** The measure of how the resource requirements of an algorithm grow as the size of the problem increases [1]. Complexity affects performance, but performance does not affect complexity [1].

**Arrays** An indexed collection of data elements of the same type, stored in consecutive memory cells [2, 3].

**Asymptotic Behaviour** Describes the limiting behavior of a function, especially as the argument tends towards infinity [4].

**Big O Notation** A symbolism used to describe the asymptotic behavior of functions [4, 5]. It indicates how fast a function grows or declines [4]. Big O notation is used for approximating formulas for analysis of algorithms, and for the definitions of terms in complexity theory [6].

**Collision (in Hashing)** Occurs when different keys produce the same hash code, resulting in them mapping to the same index in a hash table [7].

**Constant Time** An algorithm where the number of operations performed remains the same regardless of the input size, denoted as O(1) [8, 9].

**Data Structures** Ways of organizing and storing data in a computer so that it can be used efficiently. Examples from the source include Arrays and Linked Lists [2, 10].

**Hashing** A technique to store information by transforming the informational content of a key into a unique value, called its hash code, which is then used as the index at which the data associated with the key is stored [11].

**HashSet** A class in the Java Collections Framework that implements the Set interface. It is an unordered collection of unique elements [12]. The basic operations of a HashSet are add, remove, and contains [13].

**Linked List** A dynamic data structure where elements (nodes) are linked using pointers, not requiring contiguous memory [10].

**Linear Time** An algorithm where the number of operations performed is directly proportional to the size of the input, denoted as O(N) [9, 14].

**Object Reference Variable** In Java, the name of an array is an object reference variable, and the array itself must be instantiated using ‘new’ [15, 16].

**Pass-by-Reference** Passing a reference to a variable to a function, allowing the function to potentially change the contents of the original variable [15].

**Pass-by-Value** Passing the value of a variable to a function/method [15].

**Quadratic Time** An algorithm where the number of operations performed is proportional to the square of the input size, denoted as O(N2) [9, 17].

**Set** A generic set of values with no duplicate elements [18].

**Space Complexity** The amount of memory space required by an algorithm, often expressed in Big O notation [10].

**Time Complexity** The amount of time required by an algorithm to complete, typically expressed using Big O notation [19].

**Two-Dimensional Array** An array of arrays, often visualized as a table with rows and columns [20, 21]. In Java, it is declared by specifying the size of each dimension separately, such as int[][] scores = new int[22][17]; [21].

# **2B Advanced Data Structures**

**2. Core Concepts**

**Data Structures:** The lecture introduces the concept of data structures as a way of organizing and storing data in a computer.

It contrasts primitive types with more complex nested data types:

Complex data types are nested data structures composed of primitive data types. These data structures can also be composed of other complex types."

**Abstract Data Types (ADTs):** The material introduces the concept of ADTs and presents some common ADTs:

* List
* Queue
* Double-ended Queue
* Stack
* Associative Array
* Set
* Priority Queue

It notes the common data structures that are used to implement these abstract types (e.g. Lists are often implemented with Array or Linked List structures).

**Complex Data Types:** These are nested data structures formed from primitive types or other complex types. Examples include struct, array/list, map, and union. These are supported by languages like Python, C++, and Java.

**3. Arrays**

**Definition:** Arrays are introduced as a familiar data structure for storing data.

They are characterized by direct access using location and the ability to compute position based on element size.

Crucially we can go straight to an element as long as we know its location We say we “access an element”

Direct access based on the location in memory

**Advantages:**

* Direct access to elements is fast given the index.
* Useful when the number of elements is known in advance.
* Good for static storage of elements.

**Disadvantages:**

* Fixed size at runtime: "An array is a static data structure. length of the array cannot be altered at run time".
* Inefficient insertions and deletions: "Arrays can be problematic when working with Insertions and/or Deletions"
* Inserting in the middle requires shifting elements to make space.
* Deleting in the middle requires shifting elements to close the gap.
* There are problems of extra values (not fully cleaned up) in arrays after deleting an element.

**4. Linked Lists**

**Definition:** Linked lists are introduced as a dynamic alternative to arrays. Elements (nodes) are not stored contiguously in memory, but each contains a pointer to the next node. "A linked list is a dynamic data structure. length can be increased and decreased at run time. the elements may be kept at any location but still be connected to each other".

**Advantages:**

* Dynamic size: "The number of nodes in a list is not fixed and can grow and shrink on demand".
* Efficient insertions and deletions: "Linked lists are often used because of their efficient insertion and deletion." Adding/removing from the head is particularly straightforward.
* Flexibility: Can implement stacks, queues, and other ADTs.

**Disadvantages:**

* No direct access: "One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements". Elements must be accessed sequentially by traversing the list.
* Accessing elements requires traversal which is slow compared to array access.

**Linked Lists in Java:**

The document introduces how to use Linked Lists using the Java Collection framework ( java.util.\*). It covers basic operations like:

* Declaration and adding elements (with examples showing adding different numeric types and strings).
* Adding elements at a specific index.
* Adding an entire collection of items at a specific position.
* Retrieving elements with get(), getFirst(), and getLast().
* Updating elements with set().
* Removing elements with remove(), removeAll(), and clear().
* Iterating through the list.
* Using contains(), indexOf(), and lastIndexOf() for searching.
* Sorting using Collections.sort().
* Copying a list using Collections.copy().
* Shuffling a list using Collections.shuffle().
* Reversing a list using Collections.reverse().
* Extracting a portion of a list using subList().

Other common methods like size(), removeFirst(), removeLast(), and swap().

**Key Takeaway:** Linked lists excel in situations where the size of the data collection is dynamic and insertions/deletions are frequent. However, they are not suitable when direct, random access to elements is required.

**5. Stacks**

**Definition:** Stacks are introduced as an abstract data type (ADT) with restricted access, following a Last-In-First-Out (LIFO) principle.

The distinguishing characteristic of a stack is that the addition or removal of items takes place at the same end. This end is commonly referred to as the "top."

The end opposite to it is known as the "bottom".

The principle by which a stack is ordered is called LIFO

**Analogy:** The text uses the example of a stack of books, where you can only add or remove from the top.

**Implementation:** Stacks can be implemented using arrays or linked lists.

**Characteristics:**

* LIFO (Last-In, First-Out) ordering.
* Operations occur at the "top" of the stack.
* Stack overflow and underflow conditions.

**Stack Types:** The document notes that there are various stack types:

* FILO (First In, Last Out)
* FIFO (First In, First Out)
* LIFO (Last In, First Out)
* LILO (Last In, Last Out)

**Key Takeaway:** Stacks are ideal for situations requiring LIFO access patterns, like function call stacks, reversing data, and undo functionality.

**6. Doubly Linked Lists**

The document briefly mentions that doubly linked lists are a type of list where there are pointers to both the previous and next nodes.

It is also possible to create a doubly linked lists where pointers go both ways.

This allows us to traverse the list in both directions, and it also makes operations such as deletion easier.".

The document explains that having pointers both ways makes deletion easier because we don't need to keep track of previous nodes to perform the deletion.

# 2B Quiz

**Question 1:**

**Which of the following data structures is a dynamic data structure where the length can be increased and decreased at run time?**

Array

Stack

Linked List

Queue

**Answer:** (c) Linked List

**Question 2:**

**What is a key advantage of using linked lists compared to arrays?**

Direct access to individual elements

Fixed length

Efficient insertion and deletion of elements

Elements are kept at consecutive memory locations

**Answer:** (c) Efficient insertion and deletion of elements

**Question 3:**

**What does LIFO stand for, and to which data structure does it apply?**

Last-In, First-Out, applies to Queues

First-In, Last-Out, applies to Stacks

Last-In, First-Out, applies to Stacks

First-In, First-Out, applies to Arrays

**Answer:** (c) Last-In, First-Out, applies to Stacks

**Question 4:**

**Which data structure allows you to traverse the list in both directions?**

Singly Linked List

Array

Doubly Linked List

Stack

**Answer:** (c) Doubly Linked List

**Question 5:**

**What is a potential problem when attempting to add an element to a full stack?**

Stack underflow error

Stack overflow error

NullPointerException

ArrayIndexOutOfBoundsException

**Answer:** (b) Stack overflow error

**Question 6:**

**If you want to access a particular item in a linked list, where do you have to start?**

The tail

A random node

The head

The middle

**Answer:** (c) The head

**Question 7:**

**What is the primary characteristic of a stack**?

Elements are accessed randomly**.**

Elements are added and removed from the same end.

Elements are stored in a linear, contiguous memory block.

Elements are ordered based on their value.

**Answer:** (b) Elements are added and removed from the same end

**Question 8:**

**Which of the following is NOT a basic operation that can be performed on a LinkedList in Java, according to the source?**

adding

retrieving

updating

deleting

none of the above

**Answer:** (e) none of the above

# 2B Glossary

**Abstract Data Type (ADT)** An abstract data type is a data structure that can be implemented with a variety of other data structures [1]. Common ADTs include Lists, Queues, Double-ended Queues (Deques), Stacks, Associative Arrays (Dictionaries, Maps, Hashes), Sets, and Priority Queues [2].

**Arrays** Useful for storing data when the number of elements is known [3]. Arrays allow direct access to an element based on its location in memory [3]. However, arrays can be problematic when working with insertions and/or deletions [3]. The length of arrays is fixed [4]. Arrays are static data structures, so the length of the array cannot be altered at run time and all the elements are kept at consecutive memory locations

**Complex Data Types** Nested data structures composed of primitive data types or other complex types, such as struct(row), array/list, map and union [2].

**Doubly Linked Lists** A type of linked list where nodes have pointers to both the next and previous nodes, allowing traversal in both directions and making deletion operations easier [6, 7].

**Linked Lists** A dynamic data structure where elements are stored in separate objects called nodes [5, 8]. Each node contains data and a reference to the next node in the list [5, 6]. The length of a linked list can be increased and decreased at run time, and elements may be kept at any location but are still connected to each other [5, 9]. Linked lists are often used because of their efficient insertion and deletion [5, 6]. A disadvantage of linked lists is that they do not allow direct access to individual elements [10].

**LIFO (Last-In, First-Out)** The principle by which a stack is ordered, where the last item added to the stack is the first one removed

**Nodes** Individual objects in a linked list that hold data and a reference to the next node in the list [5, 6].

**Stack** An abstract data type that restricts the addition or removal of elements to one end, called the "top" [1, 7]. Stacks follow the LIFO principle [1]. Stacks can be implemented with linked lists or arrays

Top of Form

**What is Big O notation and why is it important?**

Big O notation is a mathematical notation used to describe the asymptotic behavior (growth or decline) of functions, especially in the context of computer science. It provides a way to classify algorithms according to how their resource requirements (e.g., time, memory) grow as the input size increases. It's important because it allows us to compare the efficiency of different algorithms and predict their performance for large inputs, abstracting away machine and compiler specific characteristics.

**How do I determine the Big O complexity of a piece of code?**

Determining the Big O complexity involves analyzing the number of basic operations performed by the code as a function of the input size. For sequences of statements, add the times of each statement. For if-then-else statements, consider the worst-case time of either the 'then' or 'else' block. For loops, multiply the number of loop iterations by the complexity of the statements within the loop. Nested loops multiply complexities (e.g., a nested loop iterating N times each has O(N\*N) = O(N^2) complexity if the statements inside are constant time). Function/procedure calls should be included into the overall complexity calculation by their own complexity, where the loops execute N times and each function/procedure call is complexity O(N) then the overall result is O(N^2).

**What are some common Big O complexities, and how do they compare?**

Common Big O complexities, ordered from fastest to slowest growth rate, include:

***O(1):*** *Constant time (the best). The execution time does not depend on the input size.*

***O(log n):*** *Logarithmic time. The execution time increases logarithmically with the input size.*

***O(n):*** *Linear time. The execution time increases linearly with the input size.*

***O(n log n):*** *The execution time grows slightly faster than linear time.*

***O(n2):*** *Quadratic time. The execution time increases with the square of the input size.*

***O(nc):*** *Polynomial time. The execution time increases to the power of the input size.*

***O(cn):*** *Exponential time (the worst). The execution time increases exponentially with the input size.*

Smaller complexities are more efficient for large inputs, but may have constant factors that make them slower for smaller problems.

**What are arrays, and what are their key characteristics?**

Arrays are indexed collections of data elements, where all elements must be of the same type. Arrays store elements in contiguous memory locations, allowing for fast access to elements based on their index. Arrays have a fixed size that must be determined when the array is created. In Java, the size is dynamic, but it cannot be changed after instantiation without creating a new array.

**What are the Big O complexities of common array operations?**

Common array operations have the following Big O complexities:

***Access (Read):*** *O(1) - Very fast due to direct access using the index.*

***Search:*** *O(n) - Requires checking each element in the worst case (linear search). Can be O(log n) if the array is sorted and binary search is used.*

***Insertion:*** *O(n) at the beginning or middle (requires shifting elements), O(1) at the end if there's space.*

***Deletion:*** *O(n) from the beginning or middle (requires shifting elements), O(1) from the end.*

***Space Complexity:*** *O(n) as the amount of memory consumed will increase linearly to the number of elements.*

**How do arrays differ from linked lists, and when should I use one over the other?**

Arrays use contiguous memory, offering O(1) access but slower insertion/deletion in the middle. Linked lists use dynamic memory with pointers, leading to O(n) access but faster O(1) insertion/deletion at the beginning. Arrays are preferred when fast indexed access and memory efficiency are important, and the size is relatively fixed. Linked lists are favored when frequent insertions/deletions are needed, especially at the beginning or middle, and the size is dynamic.

**What are HashSets and how do they work?**

A HashSet is a collection that contains no duplicate elements and doesn't maintain any specific order. It's implemented using a hash table, which allows for fast performance when searching for a specific element. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. This makes the time complexity for adding, removing, and searching in a HashSet is O(1) (constant time), making it efficient even with large amounts of data.

**What are some practical uses of common Set operations like Union, Intersection, and Difference?**

Set operations are useful for combining, comparing, and filtering data in various ways. **Union** combines two sets to create a new set containing all unique elements from both sets. **Intersection** finds the common elements present in both sets. **Difference** creates a set with elements present in the first set but not in the second set. Examples include database operations (finding customers who purchased product A OR product B), data analysis (identifying overlapping user groups), and algorithm design (optimizing search spaces).

Bottom of Form

convert\_to\_textConvert to source

# **3A Sorting Algorithms**

Sorting is a fundamental operation in computer science, essential for efficient searching, data analysis, and database management.

Sorting is a fundamental operation used in many algorithms and applications, including searching, data analysis, and database management.

Sorted lists are more efficient to search

**2. Types of Sorting Algorithms:**

The document categorizes sorting algorithms into:

* **Comparison-Based Sorting Algorithms:** These algorithms compare elements to determine their order.
  + **Bubble Sort:** Repeatedly swaps adjacent elements if they are in the wrong order. It's "slow and inefficient but simple."
  + **Selection Sort:** Repeatedly selects the smallest/largest element and moves it to the correct position. It's a "slight improvement on bubble sort speed."
  + **Insertion Sort:** Builds a sorted array one element at a time by inserting elements into their correct position. It's "best of the three but slightly more complex."
  + **Merge Sort:** Uses a divide-and-conquer approach to recursively sort and merge sub-lists. Big O time complexity: O(n log n). "Efficient for large datasets" and a "Guaranteed 𝑂(𝑛 log𝑛) performance."
  + **Quick Sort:** Selects a pivot and partitions elements into those smaller and larger, then recursively sorts them.
  + **Heap Sort:** Uses a heap data structure to repeatedly extract the maximum or minimum element.
* **Non-Comparison-Based Sorting Algorithms:** These algorithms do not rely on element comparisons.
  + **Counting Sort:** Counts occurrences of each element and uses the count to place elements in the correct order.
  + **Radix Sort:** Sorts numbers digit by digit, from least significant to most significant.
  + **Bucket Sort:** Divides elements into buckets, sorts them individually, and then combines them.

**3. Algorithm Complexity (Big O Notation):**

Sorting algorithms have different time complexities that affect their efficiency, especially with larger datasets.

**O(n²):** Quadratic Time Complexity (e.g., Bubble Sort, Selection Sort, Insertion Sort).

* Suitable for small datasets but inefficient for large datasets.

**O(n log n):** Log-Linear Time Complexity (e.g., Merge Sort, Quick Sort, Heap Sort).

* More efficient and widely used for larger datasets.

**O(n):** Linear Time Complexity (e.g., Counting Sort, Radix Sort, Bucket Sort). "Used for special cases when constraints allow."

Big-O notation describes the upper bound of an algorithm’s growth rate, ignoring constants and lower-order terms.

Rules for Big-O notation include: Drop constants and Drop lower-order terms.

**Bubble Sort:**

* Works by repeatedly comparing adjacent elements and swapping them if necessary. The largest element "bubbles" to the end of the list with each pass.
* **Time Complexity: O(n²).** It is inefficient because it "compares every pair multiple times, even if the array is already mostly sorted."
* The algorithm consists of two nested loops. The outer loop runs n times (for each element in the list). The inner loop runs up to 𝑛−1 times in the worst case.
* "We can improve the algorithm by noting if a swap is not performed during a pass through the list, then the list must be sorted and we can abort the sort early.

**Selection Sort:**

* Works by repeatedly finding the smallest element in the unsorted portion and swapping it with the first unsorted element.
* Improves upon Bubble Sort by reducing the number of swaps.
* **Time Complexity: O(n²).**
* "Each shift of data moves an item into its final correct position."
* The number of swaps is much less as only one swap is made on each pass (N-1 passes)

**Insertion Sort:**

* Builds the sorted array one element at a time by inserting each element into its correct position within the already sorted portion.
* "This algorithm is the most efficient of the three we have considered."
* **Time Complexity: Best case O(n) (already sorted),** **Worst/Average case O(n²).** "In fact, if the list is already sorted or nearly sorted insertion sort is very efficient."
* Number of copies is roughly same as number of comparisons but a copy is not as time consuming as a swap.

**Sorting Lists of Objects (Comparable Interface):**

* Java provides a "natural" ordering through the Comparable interface.
* Any Java object that can be ordered should implement this interface. It consists of the compareTo(Object o) method.
* The compareTo method returns an integer: 0 if equal, <0 if less than, >0 if greater than.
* Standard Java classes like String implement the Comparable interface.
* The code samples shows how to modify Bubble Sort, Selection Sort, and Insertion Sort to sort Comparable objects.

**Merge Sort:**

* Uses a divide and conquer strategy: Divide the array into two halves, Recursively sort each half, Merge the sorted halves into a single sorted array.
* **Big O time complexity: O(n log n).**
* **Space Complexity O(n) (extra space for merging).**
* Advantages:
  + Efficient for large datasets
  + Guaranteed 𝑂(𝑛 log𝑛) performance
  + Stable sort (preserves the order of equal elements)
  + Works well with linked lists"
* Disadvantages:
  + Uses extra space 𝑂(𝑛)
  + Can be slower for small arrays due to recursion overhead

**Choosing a Sorting Algorithm:**

Criteria include:

* Simplicity
* Efficiency
* Memory usage
* Type of data being sorted.

Simple algorithms are easy to understand and implement but are usually quite inefficient.

Efficiency is usually the primary criterion when choosing an algorithm, especially if large amounts of data is being sorted."

Insertion sort is the best of the elementary sort algorithms. For random data, it's about twice as fast as Bubble Sort and faster than Selection Sort. For nearly sorted data, it's exceptionally fast. All three have the same Big O time complexity.

# 3A Quiz

**Question 1:**

**Which of the following is NOT a comparison-based sorting algorithm?**

1. Bubble Sort
2. Selection Sort
3. Counting Sort
4. Insertion Sort

**Answer: C**

**Question 2:** **What is the time complexity of Bubble Sort?**

1. O(n)

b) O(n log n)

c) O(n²)

d) O(log n)

**Answer: O(n2)**

**Question 3:** **Which sorting algorithm repeatedly selects the smallest/largest element and moves it to the correct position?**

1. Insertion Sort
2. Bubble Sort
3. Selection Sort
4. Merge Sort

**Answer: SELECTION SORT**

**Question 4:** **Which of the following is an advantage of Merge Sort?**

1. Uses extra space O(n)
2. Inefficient for large datasets
3. Unstable sort
4. Not suitable for linked lists

**Answer: USES EXTRA SPACE**

**Question 5:** **What is the key idea behind Insertion Sort?**

1. Repeatedly swapping adjacent elements
2. Dividing the array into subarrays
3. Building a sorted array one element at a time
4. d) Selecting the smallest element

**Answer: BUILDING A SORTED ARRAY ONE ELEMENT AT A TIME**

**Question 6:** **What is the purpose of the Comparable interface in Java when sorting objects?**

1. To define a natural ordering for objects

b) To specify the sorting algorithm to be used

c) To handle primitive data types

d) To create a copy of the object

**Answer: TO DEFINE A NATURAL ORDERING FOR OBJECTS**

**Question 7:** **In Bubble Sort, after each pass, what happens to the largest item?**

1. It stays in the same position.

b) It moves to the beginning of the list

c) It "bubbles" to the end of the list

d) It is removed from the list.

**Answer: IT BUBBLES TO END OF THE LIST**

**Question 8: How many comparisons are made in the first pass of Bubble Sort for a list of 'n' items?**

1. n

b) n-1

c) n/2

d) log n

**Answer: N-1**

**Question 9:** **Which sorting algorithm is known to be simple but inefficient?**

1. Merge Sort

b) Quick Sort

c) Bubble Sort

d) Insertion Sort

**Answer: BUBBLE**

**Question 10:** **Which of the following factors should be considered when choosing a sorting algorithm?**

1. Simplicity

b) Efficiency

c) Memory usage

d) All of the above

**Answer: ALL**

# 3A Glossary

**Big O Notation**: Describes the upper bound of an algorithm’s growth rate, ignoring constants and lower-order terms [1]. It focuses on asymptotic growth [1].

**Bubble Sort**: A simple but inefficient sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. It has a time complexity of O(n²) [2-6]. In Bubble sort, the largest number "bubbles" to the top of the list with each pass [5, 7].

**Comparison-Based Sorting Algorithms**: Algorithms that determine the order of elements by comparing them [2, 8]. Examples include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort [2, 8].

**Counting Sort**: A non-comparison-based sorting algorithm that counts the occurrences of each element and uses the counts to place elements in the correct order [9]. It has a linear time complexity, O(n) [3, 9].

**Insertion Sort**: A sorting algorithm that builds a sorted array one element at a time by inserting elements into their correct position [2, 10, 11]. It is more efficient than Bubble Sort and Selection Sort [4, 12, 13].

**Merge Sort**: A divide-and-conquer sorting algorithm that recursively splits an array into smaller subarrays, sorts them, and then merges them back together in the correct order [8, 14]. It has a time complexity of O(n log n) and requires O(n) extra space [3, 14, 15]. Merge sort is efficient for large datasets and guarantees 𝑂(𝑛 log𝑛) performance [15].

**Non-Comparison-Based Sorting Algorithms**: Algorithms that do not rely on element comparisons [9]. Examples include Counting Sort, Radix Sort, and Bucket Sort [9].

**Quadratic Time Complexity**: Represented as O(n²), it is a time complexity suitable for small datasets but inefficient for large datasets [3]. Bubble Sort, Selection Sort, and Insertion Sort have quadratic time complexity [3].

**Radix Sort**: A non-comparison-based sorting algorithm that sorts numbers digit by digit, from least significant to most significant [9].

**Selection Sort**: A sorting algorithm that repeatedly selects the smallest/largest element from the unsorted portion of the array and swaps it with the first unsorted element [2, 16]. It has a time complexity of O(n²) [3]. Selection sort reduces the number of swaps required compared to bubble sort [17].

**Sorting**: The process of arranging elements of a list or array in a specific order, typically in ascending or descending order [2].

**Time Complexity**: A measure of the amount of time required to execute an algorithm as a function of the input size [3].

**What is a sorting algorithm and why are they important?**

Sorting algorithms are methods used in computer science to arrange elements of a list or array in a specific order, such as ascending or descending. They are crucial because sorted data makes searching, data analysis, and database management much more efficient. Instead of having to comb through a jumbled mess, one can easily find specific data points when organized in order, which drastically speeds up computation.

**What are the differences between bubble sort, selection sort, and insertion sort?**

These are all comparison-based algorithms, meaning they compare elements to determine their order. Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order, with the largest number "bubbling" to the end of the list. Selection sort finds the smallest element in the unsorted portion of the list and places it at the beginning, repeating this process until the entire list is sorted. Insertion sort builds a sorted list one element at a time, inserting each unsorted element into its correct position within the sorted portion. They differ in how they find the correct position of an element, their number of swaps and comparisons, and their overall efficiency, with insertion sort generally being the most efficient of the three.

**What does time complexity mean in the context of sorting algorithms?**

Time complexity, often expressed in Big O notation (e.g., O(n), O(n^2), O(n log n)), describes how the runtime of an algorithm scales with the size of the input (n). It doesn't measure the exact time but indicates the growth rate of the execution time. Algorithms with O(n^2) complexity, like bubble, selection, and insertion sort, are generally less efficient for larger datasets compared to O(n log n) algorithms, such as Merge Sort. O(n) algorithms are more efficient when the constraints of the data allow it.

**Why is bubble sort considered inefficient even though it's simple?**

Bubble sort is inefficient due to its O(n²) time complexity. This means that the number of comparisons and swaps increases quadratically with the size of the input, making it slow for large datasets. It also compares every pair multiple times, even if the array is already mostly sorted. Even in the best case, it takes O(n) time with an optimized version which is still relatively slow when compared to algorithms like Insertion Sort.

**How do selection sort and insertion sort improve on bubble sort's efficiency?**

Selection sort improves upon bubble sort by reducing the number of swaps. While the number of comparisons for selection sort is similar to bubble sort (both are O(n^2)), selection sort only performs one swap per pass by first finding the smallest element in the unsorted part of the array. Insertion sort, while having the same time complexity on average and in the worst case as both bubble and selection, does not perform swaps. Instead, it performs shifts of data which are less time consuming. In general, it also has a better best case scenario than bubble sort.

**How can the sorting algorithms be modified to handle objects rather than just integers?**

Sorting algorithms like bubble sort, selection sort, and insertion sort can sort lists of objects by using the Comparable interface in Java. Objects that can be ordered should implement this interface, which contains the compareTo(Object o) method. This method determines the relative order of two objects, returning 0 if they are equal, a negative value if the object is less than, or a positive value if the object is greater than another object. The sorting algorithms will then use this method in place of a less than or greater than check on integers.

**What are the main principles behind merge sort and how does it improve upon the earlier sorting algorithms?**

Merge sort is a divide-and-conquer algorithm. It splits the input array into halves, recursively sorts these halves, and then merges them back together into a single sorted array. Merge sort has a time complexity of O(n log n) which is more efficient than bubble, selection, and insertion sorts, making it well-suited for large datasets. It also has a guaranteed performance and is a stable sort which means it preserves the order of equal elements.

**When would you choose merge sort over bubble sort, selection sort, or insertion sort, and what are its drawbacks?**

Merge sort is preferable over bubble, selection, or insertion sort for larger datasets where efficiency is key. Its consistent O(n log n) time complexity and stability make it a strong choice. However, merge sort's drawbacks include its extra O(n) space requirement for merging and the overhead from recursion, which can make it slower for very small datasets. It's also preferable for linked lists where merging is efficient.

Top of Form

# **3B Search Algorithms**

Search Algorithms

**Core Concept:** The fundamental problem is to find a specific "target key" within a collection of items.

The document highlights the need to "understand the requirement to search a list of items for a target key."

**Classification:** Search algorithms are broadly divided into:

**Uninformed (Blind) Search:** These algorithms explore possibilities systematically without prior knowledge of the data being searched.

The document states: "Uninformed (Blind) Search Algorithms - These do not use domain-specific knowledge and explore possibilities systematically."

**Informed (Heuristic) Search:** These algorithms use specific knowledge about the data to find the target more efficiently. This document doesn't delve deeply into heuristic search algorithms.

The document states: "Informed (Heuristic) Search Algorithms - These use problem-specific knowledge to find solutions efficiently."

**Linear Search (Sequential Search)**

**Process:** Compares the target key with each item in the list sequentially until a match is found or the end of the list is reached.

The process is described as: "Beginning with the first item in the list compare each item with the target key until a match occurs or the end of the list is reached"

**Time Complexity:** **O(n)** meaning the time it takes grows linearly with the size (n) of the list.

**Use Cases:** Suitable for unsorted lists or when the list size is relatively small.

**Implementation:**

The provided code examples show the algorithm for searching both arrays of integers and arrays of String objects:

Integers use the equality operator ==:

public int linearSearch(int items[], int key) {

// iterate over each element in array

for(int i = 0; i < items.length; i++) {

if (key == items[i]) {

return i; // found it so return position

}

}

return -1 // not found so return -1

}

String objects use the .equals() method for equality comparison:

public int linearSearch(String items[], String key) {

// iterate over each element in array

for (int i = 0; i < items.length; i++) {

if (key.equals(items[i])) {

return i; // found it so return position

}

}

return -1; // not found so return -1

}

**Analysis:** The document highlights linear search inefficiency for large lists. With an example, it notes a best-case search of 1ms, a worst-case of 16.6 minutes, and an average case of 8.3 minutes when searching 1 million entries.

"This example serves to show inefficiency of sequential search when searching long lists."

**Binary Search**

**Requirements:**

The list **must be sorted** in ascending or descending order.

Direct access to items is required (e.g., arrays).

The document states: "...**the list must be ordered** and we must have direct access to any item in the list (i.e. array access)."

**Process:** Uses the "divide and conquer" approach, repeatedly dividing the search space in half by comparing the target key with the middle element of the current search range.

The document says, "Compare the target key with the key of the item at the middle of the list. If this is the item then report success. If this is not the required item then determine whether the sought item is in the first half of the list or the second and then search the appropriate half. Repeat this process until the key is found or the list cannot be divided further."

**Time Complexity:** **O(log n)** meaning the time required grows logarithmically with the size of the list. This makes it significantly faster than linear search for large lists.

**Implementation:**

The algorithm is implemented for searching an array of integers: ```java public int binarySearch(int items[], int key) { int left = 0, right = items.length-1;

while (left <= right) {

int pivot = (left + right) / 2;

if (key == items[pivot]) {

return pivot; // found

} else if (key > items[pivot]) {

left = pivot + 1; // search right

} else {

right = pivot - 1; // search left

}

}

return -1; // not found

}

```

**Comparison of Objects:** When dealing with objects, the .compareTo() method of the Comparable interface is used for ordering, and .equals() is used for equality checks.

The document states: "Not only do we need to be able to test for equality using the objects equals method but we need to be able to test for ordering. (i.e. is one object less than or greater than another)."

**Analysis:** Binary search is much more efficient than linear search on sorted data. Using the same 1 million entry example, it only takes 20 passes.

"In fact, it only takes 20 passes to search 1million entries."

Best case: 1ms, Worst case: 20ms, Average case: 10ms.

**Graph Data Structures**

**Definition:** A graph is a data structure consisting of nodes (vertices) and edges (connections) that link pairs of nodes.

The document defines it as: "A graph is a data structure that consists of a set of nodes (vertices) and a set of edges (connections) that link pairs of nodes."

**Types of Graphs:**

**Directed Graph (Digraph):** Edges have a direction (one-way connections).

The document states: "Directed Graph (Digraph) Edges have a direction (one-way connections)."

**Undirected Graph:** Edges have no direction (two-way connections).

The document states: "Undirected Graph Edges have no direction (two-way connections)."

**Weighted Graph:** Edges have associated weights (costs, distances, etc.).

The document states: "Weighted Graph Edges have weights (costs, distances, etc.)."

**Graph Representations:**

**Adjacency Matrix:** A 2D array where rows and columns represent vertices, and cells indicate the presence or weight of an edge between them.

The document says: "In the adjacency matrix, vertices of the graph represent rows and columns."

**Adjacency List:** A list where each vertex is associated with a list of its adjacent vertices.

The document states: "An adjacency list is nothing but a linked list and each node in the list represents a vertex."

**Graph Traversal**

**Breadth-First Search (BFS):** Explores the graph level by level using a queue.

The document states: "Breadth-first (BFS) technique uses a queue to store the nodes of the graph. In BFS we traverse the graph breadth-wise. This means we traverse the graph level wise."

The steps for the algorithm are outlined: "Step 1: Begin with the root node and insert it into the queue...Step 4: Now add all the adjacent nodes of the root node to the queue and repeat steps 2 to 4 for each node."

**Depth-First Search (DFS):** Explores the graph depth-wise, going as deep as possible along each branch before backtracking, using a stack.

The document states: "DFS technique starts with a root node and then traverses the adjacent nodes of the root node by going deeper into the graph. In the DFS technique, the nodes are traversed depth-wise until there are no more children to explore."

The steps for the algorithm are outlined: "Step 1: Start with the root node and insert it into the stack...Step 3: For node marked as ‘visited’ (or in visited list), add the adjacent nodes of this node that are not yet marked visited, to the stack."

**Real-World Application: The Traveling Salesman Problem (TSP)**

**Problem Definition:** Given a set of cities and distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city.

The document defines it as: "Given a set of cities and the distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city."

**Graph Representation:**

Cities are represented as nodes (vertices).

Paths between cities are represented as edges, with weights representing distances or travel costs.

The document states: "Nodes (Vertices) = Cities, Edges = Paths between cities (with weights representing distances or travel costs)"

**Complexity:** TSP is an NP-hard problem. There is no known efficient exact solution for large inputs.

The document states: "There is no efficient exact solution for large inputs because finding the best solution requires exponential time in the worst case."

**Solution Approaches:**

**Brute Force:** Tries all possible routes, which is highly inefficient for larger problems, with a time complexity of O(n!).

The document says that brute force "Tries every possible route and picks the shortest. Time Complexity: 𝑂(𝑛!) (Factorial growth, very slow for large n)."

**Dynamic Programming (Held-Karp):** Uses memoization to reduce redundant calculations.

**Approximation Algorithms (Heuristics):** Used for large problems where an exact solution is infeasible (e.g., Nearest Neighbor Algorithm).

**Summary & Key Takeaways** \* **Searching is a key requirement:** The document emphasizes that the ability to search a list for a target is very important. \* The document states: "The need to be able to search a list of items for a particular target key." \* **Linear search** works for unordered lists but is inefficient for large datasets. \* **Binary search** is much faster but requires an ordered (sorted) list. \* When comparing objects we need to use the .compareTo() and .equals() methods \* Graphs are a valuable data structure for representing networks. \* BFS and DFS are essential techniques for exploring graphs. \* Problems such as the Traveling Salesman Problem are important real world challenges. This briefing document should provide a comprehensive overview of the key ideas presented in the provided excerpts from "Search Algorithms.pdf." Let me know if you have any further questions.

# 3B Quiz

**1:** **What are the two main categories of search algorithms?**

a) Linear and Binary

b) Informed and Uninformed

c) Simple and Complex

d) Ordered and Unordered

**Answer:** b) **Informed and Uninformed**

**2:** **Which search algorithm requires the list to be ordered?**

1. Linear Search
2. Binary Search
3. Breadth-First Search
4. Depth-First Search

**Answer:** b) **Binary Search**

**Question 3:** **What is the time complexity of Linear Search in the worst-case scenario for a list of n items?**

a) O(1)

b) O(log n)

c) O(n)

d) O(n^2)

**Answer:** c) **O(n)**

**Question 4:** **What is the return value of a search algorithm if the key is not found?**

a) 0

b) 1

c) -1

d) null

**Answer:** **-1**

**Question 5:** **Which method is used to test for ordering when searching an array of objects in Java?**

1. ==
2. equals()
3. compareTo()
4. toString()

**Answer:** c**)** **compareTo()**

**Question 6:** **What data structure does Breadth-First Search (BFS) use to store nodes of a graph?**

1. Stack
2. Queue
3. Linked List
4. Array

**Answer:** b) **Queue**

**Question 7:** **What data structure does Depth-First Search (DFS) use to store nodes that are being traversed?**

1. Queue

b) Stack

c) Tree

d) Graph

**Answer:** **Stack**

.

**Question 8: In graph terminology, what are the connection links between vertices called?**

a) Nodes

b) Edges

c) Paths

d) Roots

**Answer:** **Edges**

**Question 9:** **What distinguishes a directed graph (digraph) from an undirected graph?**

a) The number of vertices

b) The weight of the edges

c) The direction of the edges

d) The color of the vertices

**Answer:** **The direction of the edges**

**Question 10:** **What is the goal of the Traveling Salesman Problem (TSP)**

a) Find the longest possible route that visits each city

b) Find the shortest possible route that visits each city exactly once and returns to the starting city

c) Find the most scenic route between cities

d) Visit as many cities as possible within a limited time frame

**Answer:** b) **Find the shortest possible route that visits each city exactly once and returns to the starting city**

Top of Form

Bottom of Form

# 3B Glossary

**Adjacency List:** A linked representation of a graph where each node in the list represents a vertex, and the presence of an edge between two vertices is indicated by a pointer from the first vertex to the second [1]. For an undirected graph, the total lengths of adjacency lists are usually twice the number of edges, while for a directed graph, the total length of the adjacency lists equals the number of edges [2, 3]. In a weighted graph, the adjacency list includes a space denoting the weight of each node [3].

**Adjacency Matrix:** A linear representation of graphs that stores the mapping of vertices and edges in a matrix format [4]. If a graph has N vertices, the adjacency matrix has a size of NxN [4]. For an undirected graph, the intersection Mij is equal to 1 if there is an edge [5]. For a directed graph, Mij is 1 only if there is a clear edge directed from Vi to Vj [5]. In a weighted graph, the weight is specified in the adjacency matrix instead of '1' to represent the edge's weight [5].

**BFS (Breadth-First Search):** A graph traversal technique that uses a queue to store nodes and explores the graph breadth-wise, level by level [6, 7]. It begins with the root node, adds it to a visited list, and enqueues its adjacent nodes [7, 8].

**Binary Search:** A search algorithm that requires the list to be ordered and allows direct access to any item in the list [9]. It works by comparing the target key with the middle item in the list and then searching either the first or second half of the list, depending on whether the sought item is in the first half or the second, repeating until the key is found or the list cannot be divided further [9, 10]. The algorithm can be implemented for both arrays of integers [11] and arrays of objects, using the compareTo method for ordering [12-14].

**Comparable Interface:** A Java interface that provides a "natural" ordering for objects [12]. Any Java object that can be ordered should implement this interface, which consists of the compareTo(Object o) method [12, 13]. The method returns 0 if both objects are equal, <0 if the object is less than object o, and >0 if the object is greater than object o [13].

**DFS (Depth-First Search):** A technique to traverse a tree or graph. It starts with a root node and explores as far as possible along each branch before backtracking [15]. DFS uses a stack data structure to store the nodes being traversed [15, 16].

**Directed Graph (Digraph):** A graph where the edges have a specific direction, originating from one vertex and ending at another [17-19].

**Edges:** Connections that link pairs of nodes (vertices) in a graph [17, 18].

**Graph:** A data structure consisting of a set of nodes (vertices) and a set of edges (connections) that link pairs of nodes [17, 18]. Graphs can be directed or undirected and weighted or unweighted [17].

**Informed (Heuristic) Search Algorithms:** Search algorithms that utilize problem-specific knowledge to efficiently find solutions [20].

**Linear (Sequential) Search:** An algorithm that searches for a target key by comparing each item in the list until a match is found or the end of the list is reached [21]. It can be used on unordered lists [22]. When searching an array of objects, the equals method is used instead of the equality operator == [23].

**Nodes (Vertices):** The points in a graph that are connected by edges [18].

**The Traveling Salesman Problem (TSP):** Given a set of cities and the distances between them, the problem seeks the shortest possible route that visits each city exactly once and returns to the starting city [24, 25]. It can be represented as a weighted complete graph, where nodes are cities, and edges are paths between cities with weights representing distances [25, 26].

**Uninformed (Blind) Search Algorithms:** Search algorithms that do not use domain-specific knowledge and explore possibilities systematically [20].

**Undirected Graph:** A graph where edges have no direction, representing two-way connections [17, 18].

**Weighted Graph:** A graph in which a weight is associated with each edge, typically representing the distance or cost between two vertices [17, 19, 27].

FAQ:

**What is the fundamental purpose of search algorithms, and what are the main categories they fall into?**

Search algorithms are used to locate specific items within a collection of data, such as arrays, trees, or graphs. They are broadly categorized into two types: *uninformed (blind) search* algorithms, which systematically explore all possibilities without any prior knowledge about the data, and *informed (heuristic) search* algorithms, which use problem-specific knowledge to find solutions more efficiently.

**How do linear search and binary search differ, and when is each appropriate to use?**

*Linear search* examines each item in a list sequentially until the target item is found or the end of the list is reached. It has a time complexity of O(n), meaning the time taken increases linearly with the number of items. Linear search can be used with any list, ordered or unordered. *Binary search* is a much faster algorithm, with a time complexity of O(log n), which makes it suitable for large lists, but requires the list to be sorted in ascending or descending order. It works by repeatedly dividing the search space in half.

**How does searching a list of simple types (like integers) differ from searching a list of objects?** When searching lists of simple data types such as integers or doubles, you can use the equality operator (==) to compare values. However, when searching for objects, you must use the equals() method to ensure proper comparison of object content rather than memory addresses. Further, if using a binary search on a list of objects, the objects must implement the Comparable interface which enables comparison for ordering as well.

**Explain the concepts of Breadth-First Search (BFS) and Depth-First Search (DFS) in the context of graph traversal.**

*Breadth-First Search (BFS)* explores all the neighbors of a node before moving to the next level of the graph. It uses a queue data structure to keep track of which nodes to visit next, visiting all nodes on the same level before proceeding to the next level. *Depth-First Search (DFS)* explores as far as possible along each branch before backtracking. It uses a stack data structure to keep track of the nodes to explore, visiting the deepest unexplored node at each step. BFS is useful for finding the shortest path in an unweighted graph, whereas DFS can be used for cycle detection and topological sorting, among other tasks.

**What are adjacency matrices and adjacency lists, and how are they used to represent graphs?**

*Adjacency matrices* are a way to represent graphs using a two-dimensional matrix. The rows and columns represent vertices, and an entry in the matrix indicates the presence or absence of an edge between vertices. *Adjacency lists* represent a graph as a list where each vertex is associated with a list of its adjacent vertices. Adjacency lists are often more efficient than adjacency matrices for sparse graphs (graphs with fewer edges).

**How can graphs be used to model real-world scenarios, and what example was given?**

Graphs are versatile data structures used to represent various real-world networks. The provided materials specifically mention road maps and navigation, where cities or locations are nodes, and the roads connecting them are edges. Edges can be weighted with the distance or travel time. Another application mentioned is the Traveling Salesman Problem which can be represented with cities as nodes and distances as the weighted edges.

**What is the Traveling Salesman Problem (TSP), and why is it considered challenging?**

The Traveling Salesman Problem (TSP) involves finding the shortest possible route that visits each city in a given set exactly once and returns to the starting city. It's challenging because finding the optimal solution requires checking a vast number of possible routes (n!, where n is the number of cities), making it computationally infeasible for large numbers of cities. There is no known polynomial time solution.

**What are some common approaches to solving the Traveling Salesman Problem (TSP), given its computational complexity?**

Because finding an exact solution to the TSP is impractical for large problems, a few common approaches are used to address the problem. *Brute force* which examines every possible path to find the shortest is an exact, but very slow solution. *Dynamic programming* uses memoization to improve speed over brute force, but is still slow for large inputs. *Approximation algorithms* use heuristics like the Nearest Neighbor algorithm to quickly find good, but not optimal, solutions.

# 4A Algorithmic Analysis

**Algorithm Analysis Fundamentals:**

* **Algorithms vs. Data Structures:** An algorithm is a "step-by-step procedure for performing tasks in a finite amount of time," while a data structure is a "systematic way of organizing and accessing data.
* **Goals of Analysis:** To understand algorithm constraints, trade-offs between execution time and memory usage, provide guidance for algorithm selection, and build a repertoire of standard algorithms.
* **Comparing Algorithms:** The primary goal is to determine how the running time of an algorithm depends on the size of the input. "Our Primary interest is in running time (time complexity) of algorithms and data structure operations… We are interested in determining the dependency between the running time and the size of the input." Secondary interest lies in space complexity (memory usage).

**Time Complexity and Performance Factors:**

* **Running Time Dependency:** The time taken by an algorithm depends on the number of computational steps and the time taken for each step.
* **Focus on Computational Steps:** Because the time for each step depends on many external factors (computer performance, compiler efficiency etc.) we can use computational steps as our measure of performance: "We will therefore analyse the computational steps"

**Linear vs. Quadratic Solutions:**

**Example: Package Delivery:** Demonstrates how different approaches to the same problem can lead to vastly different complexities.

* Solution 1 (deliver all, return) is linear (O(N)), while
* Solution 2 (deliver one, return) is quadratic (O(N^2)).
* Solution 1 : N + N = 2N (Linear Solution) - GOOD…
* Solution 2: 2 \* (1+2+…+N) = N2 + N (Quadratic Solution) - BAD
* **Importance of Scaling:** Highlights that a linear solution is preferable as the input size (N) increases.

**The Traveling Salesman Problem:**

* **Brute-Force Approach:** Trying all permutations leads to a factorial time complexity of O(n!), making it impractical for even moderately sized problems. "The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute-force search). The running time for this approach lies within a polynomial factor of O(n!), the factorial of the number of cities, so this solution becomes impractical for 20 cities i.e. it doesn’t scale."

**Analyzing Code Complexity:**

* **Counting Statements:** A simple example illustrates how to analyze a while loop and derive a complexity function (f(n) = 4n + 3 in the example).
* **Rules for Analyzing Code:** Covers consecutive statements (add), If/Else (take the larger branch), For loops (multiply by the number of iterations), and nested loops (analyze inside out and multiply).
* **Examples:** Code examples are analyzed including consecutive statements, if/else statements and nested loops

**Big-O Notation:**

* **Simplifying Complexity:** Big-O notation provides an upper bound on the asymptotic growth of an algorithm, focusing on the dominant term as input size grows. "Big-O makes no attempt to provide exact running times for an algorithm, but estimates how fast the execution time grows as the size of the input grows."
* **Dropping Constants and Lower Order Terms:** Big-O simplifies analysis by "drop[ping] lower order terms (constants)" and "drop[ping] the constant coefficient of the highest order term." For example, f(N) = 4N + 3 simplifies to O(N).
* **Crossover Point:** Highlights that for smaller input sizes, an algorithm with a larger constant factor but lower complexity might outperform one with a smaller constant factor but higher complexity. "Note how for small values of N the value in the third column (10N) overwhelms the quantity in the second column (0.01N2)… Crossover Point: at N=1000 the time taken by both terms is roughly equal. But after that the term 0.01N2 overwhelms 10N"
* **Common Complexity Classes:** Provides a table of common complexities (O(1), O(log N), O(N), O(N log N), O(N^2), O(N^3), O(2^N)) with examples.
* **Best, Worst, and Average Cases:** Introduces the concepts of best-case, worst-case, and average-case analysis. Typically, worst-case behavior is analyzed because it's easier to evaluate, though average-case can be more useful.

**Limitations of Big-O:**

* **Not for Small Inputs:** Inappropriate for small amounts of input, where simpler algorithms are often better.
* **Ignores Large Constants:** Large constants can impact performance in excessively complex algorithms. "e.g. In a complex algorithm 2N log N is probably better than 1000N even though growth rate is larger"
* **Simplifies Operations:** Doesn't differentiate between more expensive and less expensive operations.
* **Assumes Infinite Memory:** Assumes infinite memory, which can be a problem with large datasets.

**Key Ideas and Facts:**

* Algorithm analysis focuses on understanding how runtime scales with input size.
* Time complexity is the primary focus, with space complexity as a secondary concern.
* Linear algorithms are generally more efficient than quadratic algorithms as input size increases.
* Big-O notation provides a simplified representation of algorithm complexity, focusing on the dominant term.
* Big-O has limitations and may not be appropriate for all scenarios.
* Worst-case analysis is commonly used, but average-case analysis can be more useful in certain situations.

# 4A Quiz

**Question 1:**

**Given the following functions representing the number of instructions required for the execution of an algorithm, determine the Big-O time complexities**

1. 5n + n2 – 2
2. 7
3. 3 + 4 log2 N
4. 4n + 10 log n + 25

**Answer:**

* O(N2) [2]
* O(1)
* O(log N)
* O(N)

**Question 2:**

**Estimate the Big-O complexity order for an algorithm that calculates the minimum element in an array. Given an array [ 3, 5, 2, 7, 9, 12, 1, 6 ] the basic algorithm is:**

lowest = first element

FOR each remaining element

IF current is less than lowest THEN

lowest = current

ENDIF

ENDFOR

**Answer:** O(N)

**Question 3:**

**function a() { Statement1 Statement2 }**

**If Statement1 is O(n) and Statement2 is O(n log n), then what is the overall time complexity of function a?**

**Answer:** O(n log n)

**Question 4:**

**function b() { If (condition) Statement1 else Statement2 }**

**If Statement1 is O(n2) and Statement2 is O(n), then what is the overall time complexity of function b?**

**Answer:** O(N2)

**Question 5:**

**Analyze the complexity of the following code snippets**

* **5a)**

1. int i=0;
2. while (i<n) {
3. sum++;
4. i++;
5. }
6. int y=0;
7. while (y<n) {
8. total++;
9. y++;
10. }

* **5b)**

1. int i=0;
2. while (i<n) {
3. int y=0;
4. while (y<n) {
5. total++;
6. y++;
7. }
8. i++;
9. }

* **5c)**

1. int i=0;
2. while ( i<n ) {
3. int y=0;
4. while ( y<n ) {
5. int j=0;
6. while ( j<n ) {
7. total++;
8. j++;
9. }
10. y++;
11. }
12. i++;
13. }

* **5d)** Place the following growth rates in order – lowest to highest: N, 1, N2, N3, N log N, 2N, log N

**Answer:**

* **5a)** O(N) [9]
* **5b)** O(N2) [10]
* **5c)** O(N3)
* **5d)** 1, log N, N, N log N, N2, N3, 2N

**Question 6:**

**An algorithm takes 0.5ms for an input size of 100. How long will it take for an input size of 500 if the running time is:**

* **a) Linear**
* **b) Quadratic**
* **c) Cubic**

**Answer:**

* a) Linear: 2.5ms
* b) Quadratic: 12.5ms
* c) Cubic: 62.5ms

**Question 7:**

**When comparing two algorithms with different complexity analyses (e.g., a quadratic algorithm of 0.01N2 and a linear algorithm of 10N), what is meant by the "crossover point"?**

**Answer:**

The crossover point is where the performance of both algorithms is roughly equal [11]. For values of N smaller than the crossover point, the algorithm with a larger order may perform better due to smaller constants. However, beyond the crossover point, the algorithm with the smaller order will always be more efficient, as the differences between N2 and N increases

Top of Form

Bottom of Form

# 4A Glossary

* **Algorithm:** A step-by-step procedure for performing tasks in a finite amount of time
* **Data Structure:** A systematic way of organizing and accessing data .
* **Time Complexity:** The running time of an algorithm, typically the primary interest when measuring efficiency. It describes the dependency between the running time and the size of the input
* **Space Complexity:** The memory usage required by an algorithm to solve a problem
* **Linear Algorithm:** An algorithm where the time taken is directly proportional to the amount of input [4].
* **Quadratic Solution:** An algorithm where the time taken is proportional to the square of the input size
* **Big-O Notation:** A convention used to represent different complexity classes. It estimates how fast the execution time grows as the size of the input grow. Big-O notation simplifies the analysis of complexity expressions and defines an upper bound on the asymptotic growth of a complexity class
* **Complexity Class:** A classification of algorithms based on their growth rate, represented using Big-O notation
* **Computational Steps:** The number of steps an algorithm takes to solve a problem [9]. Analyzing computational steps helps in determining the time complexity of an algorithm
* **Performance Factors:** Elements that affect the time an algorithm takes to solve a problem, such as the number of computational steps and the time taken to complete each step
* **Crossover Point:** When comparing two algorithms with different complexity analyses, the crossover point is where the performance of both algorithms is roughly equal
* **Growth Rate:** The rate at which an algorithm's resource consumption (e.g., time or space) increases as the input size grows
* **Best Case:** The scenario where an algorithm performs most efficiently.
* **Worst Case:** The scenario where an algorithm performs least efficiently. Usually algorithms are analyzed based on their worst-case behaviour
* **Average Case:** The expected performance of an algorithm over all possible inputs

Key concepts related to Big-O Notation:

* **Terms:** The parts of a function that are added together
* **Coefficients:** The leading constants of each term in a function

When using Big-O notation:

* Drop lower order terms (constants)
* Drop the constant coefficient of the highest order term

**Big O Examples:**

* f(N) = N2+N, Big-O notation is O(N2)
* f(N) = 5N2+4N, Big-O notation is O(N2)
* f(N) is logbN, Big-O notation is O(log N)
* f(N) is 3N2 + 3N + 2, Big-O notation is O(N2)

**What are algorithms and data structures, and why is their analysis important?**

Algorithms are step-by-step procedures for performing tasks within a finite amount of time, while data structures are systematic ways of organizing and accessing data. Analyzing these is crucial because it helps us understand their constraints, the trade-offs between execution time and memory usage, and ultimately guides us in choosing the most efficient approach for a specific purpose.

**What are the primary and secondary interests when measuring the efficiency of an algorithm?**

The primary interest is the running time or time complexity, which examines the relationship between the algorithm's execution time and the size of the input. The secondary interest is memory usage, or space complexity, which refers to the amount of memory required to solve a problem, especially relevant on devices with limited resources.

**How do we determine the time complexity of an algorithm, and what factors influence it?**

Time complexity is determined by analyzing the computational steps of an algorithm. The number of steps is calculated by examining the code. While the actual time taken for each step depends on factors like computer performance and compiler efficiency, the analysis concentrates on counting the number of operations performed, as this provides a constant factor regardless of the specific implementation environment.

**What is Big-O notation, and what is its purpose?**

Big-O notation is a mathematical notation used to describe the limiting behavior of a function when the argument tends towards a particular value or infinity. In algorithm analysis, it's used to classify algorithms based on how their execution time or memory usage grows as the input size increases. It simplifies the analysis of complexity expressions by focusing on the dominant term and ignoring constants and lower-order terms, providing an upper bound on the asymptotic growth of the algorithm's complexity.

**What are some common Big-O complexity classes, and what do they signify?**

Common Big-O complexity classes include O(1) (constant), O(log N) (logarithmic), O(N) (linear), O(N log N), O(N^2) (quadratic), O(N^3) (cubic), and O(2^N) (exponential). These classes represent how the algorithm's running time scales with the input size. For instance, O(N) means the running time grows linearly with the input size, while O(N^2) means it grows quadratically.

**Why do we focus on the worst-case scenario when analyzing algorithms?**

While it's ideal to consider best, worst, and average-case scenarios, analyzing the worst-case behavior is often easier to determine. It provides a guarantee on the upper bound of the algorithm's running time, ensuring that the algorithm will not perform worse than this bound, regardless of the input. However, in some situations, the average-case behavior might be more useful if it's possible to evaluate it effectively.

**What is meant by the "crossover point" when comparing algorithms with different complexity classes?**

The "crossover point" refers to the input size (N) at which the performance of two algorithms with different complexity classes becomes roughly equal. For small input sizes, an algorithm with a larger complexity class but smaller constants might perform better. However, beyond the crossover point, the algorithm with the lower complexity class will eventually outperform the other as the input size continues to increase.

**What are some limitations of using Big-O notation for algorithm analysis?**

Big-O notation simplifies analysis by ignoring constants and lower-order terms. While effective in classifying algorithms into complexity classes, it has limitations. It's not appropriate for small input sizes, where the constants may dominate. Additionally, large constants ignored by Big-O can impact performance in excessively complex algorithms. Analysis also assumes infinite memory, which isn't practical for large datasets. It also does not differentiate between the cost of different operations.

# 4B Recursion

Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the *same* problem. This is achieved through functions that call themselves from within their own code.

"In computer science, recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem."

**Breaking Down Problems:** Recursion is about breaking down a large problem into smaller, self-similar subproblems.

"The best way to solve a problem is breaking down a problem into smaller problems and solving these smaller problems...Recursion is a technique that solves a problem by solving a smaller problem of the same type."

**Flow of Control:**

* The document emphasizes the flow of control in recursive function calls. When a method calls itself, the flow of control passes to a new instance of the method.
* Upon completion of that instance, control returns to the point immediately following the recursive call in the previous instance.

**Essential Elements of Recursion:**

A valid recursive method MUST meet two conditions:

1. **Recursive Call:** There must be a line in the method which is a further call to the method itself.
2. **Conditional Call (Base Case):** This call MUST be a CONDITIONAL CALL. There must be something that causes the sequence of calls to the method to cease, preventing an infinite loop.

"There MUST be a line in the method which is a further call to the method itself... This call MUST be a CONDITIONAL CALL...There must be something that causes the sequence of calls to the method to cease."

**Infinite Loops:** The document strongly warns against infinite loops. A recursive function without a base case will continuously call itself, leading to a stack overflow and program crash.

"We could end up in an infinite loop i.e. we keep on calling the method that calls itself, which calls itself, which calls itself infinitly unless there was something that stops further calls to the method"

**Key Steps for Recursion:** The document highlights two key steps when constructing a recursive function.

1. **Base cases:** Always have at least one case that can be solved without using recursion.
2. **Make progress:** Any recursive call must make progress toward a base case.

**Visualizing Recursion:** Analogies, such as Russian nesting dolls, are used to help visualize the process of recursion. However, it is cautioned that these analogies can be flawed, as they might equate simpler with smaller which isn't always true.

**Examples:** Several examples are provided to illustrate recursion, including:

Counting Russian dolls.

String reversal (demonstrated with "cat" becoming "tac").

Calculating the factorial of a number.

**Factorial Example:** Factorials are expressed in terms of smaller factorials:

"N! = N \* (N-1) \* (N-2) \* (N-3) \* .... \* 3 \* 2 \* 1"

"8! = 8 \* 7!"

**Recursion vs. Iteration:** The document compares recursion with iteration (looping).

Iteration simply 'jumps back' to the beginning of the loop while recursion is based upon calling the same function over and over.

Recursive solutions can be less efficient due to the overhead of function calls (memory allocation, parameter copying, branching).

Recursion may simplify the solution and result in shorter, more easily understood code.

"Iteration can be used in place of recursion... Recursive solutions are often less efficient... Recursion may simplify the solution... shorter, more easily understood source code"

**Three-Question Method for Writing Recursive Functions:** A useful guide for writing recursive functions involves answering three questions:

1. **The Base-Case Question:** Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?
2. **The Smaller-Caller Question:** Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?
3. **The General-Case Question:** Assuming that the recursive call(s) work correctly, does the whole function work correctly?

**Anagrams:** Anagram generation is presented as another application of recursion. The method involves recursively anagramming the rightmost N-1 letters, rotating the word, and repeating.

**Fibonacci Numbers:** The Fibonacci sequence is defined recursively (each number is the sum of the two preceding ones).

**When to Use Recursion:** Recursion is particularly useful when the depth of recursive calls is relatively "shallow" and when it leads to clearer, simpler code.

**Advantages of Recursion:**

* Recursion makes it easier to write simple and elegant programs
* Directly reflects the abstract solution strategy (algorithm)
* Reduces the cost of maintaining the software.

**Disadvantages of Recursion:**

* Recursion makes it easier to write inefficient programs
* Simple, elegant recursive algorithms can still be inefficient.

**Recursion and Backtracking:**

* Recursion and backtracking are not synonymous
* Backtracking relies on recursion. Backtracking uses recursion to explore all possibilities, and "undo" bad choices.
* Not all recursion is backtracking, but all backtracking problems use recursion

# 4B Quiz

**Question 1: What is recursion in computer science?**

a) A method of defining an infinite set of objects.

b) A method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem

c) A technique for creating infinite loop

d) A way to avoid using functions in code.

**Answer:** b)

**Question 2:** **According to the source, what are the key steps for recursion?**

a) Base cases and infinite loops.

b) Making progress and conditional calls.

c) **Base cases** and **making progress**

d) Calling other methods and returning values.

**Answer:** c)

**Question 3:** **What is the primary danger of not having a conditional call in a recursive method?**

a) The program will crash.

b) The program will run faster.

c) The program will enter an **infinite loop**

d) The program will return an incorrect value.

**Answer:** c)

**Question 4:** **What are the benefits of using recursion?**

a) Recursive functions are always more efficient.

b) Recursive functions are clearer, simpler, and shorter

c) Recursive functions are harder to understand

d) Recursive functions increase the cost of maintaining software.

**Answer:** b)

**Question 5: What is the relationship between recursion and iteration?**

a) They are the same thing.

b) Iteration is always more efficient than recursion.

c) Iteration can be used in place of recursion

d) Recursion is based upon jumping back to the beginning of a loop.

**Answer:** c)

**Question 6: What is the "Base-Case Question" when writing a recursive function?**

a) Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?

b) Does each recursive call involve a larger case of the original problem?

c) Is the function call more expensive than a jump?

d) Does the function simplify the solution?

**Answer:** a)

**Question 7:** **What is the purpose of backtracking, and what is its relation to recursion?**

a) Backtracking is the same as recursion.

b) Backtracking avoids using recursion.

c) Backtracking uses recursion to explore all possibilities

d) Backtracking is more efficient than recursion.

**Answer:** c)

**Question 8:** **How does the source explain recursion using the concept of Russian dolls?**

a) By stating that Russian nesting dolls are a flawed analogy because simpler does not always equate to smaller

b) By explaining how finding the smallest doll is similar to a recursive algorithm

c) By opening dolls to find a smaller doll inside until you find the smallest doll that cannot be opened

d) All of the above.

**Answer:** d)

Recursion Study Guide

Quiz

1. Define recursion in computer science.
2. Explain the **flow of control** in a program that uses recursion.
3. What are the two essential conditions for a recursive method to avoid an infinite loop?
4. Describe how the "Russian doll" analogy illustrates recursion and why some consider it flawed.
5. Why is the base case important in recursion?
6. What are the advantages and disadvantages of using recursion over iteration?
7. Explain the concept of the call stack in the context of recursion.
8. Provide an example of a problem that is well-suited for a recursive solution.
9. Explain how backtracking and recursion relate to one another.
10. Explain how recursion can help reduce the cost of software maintenance.

Quiz Answer Key

1. Recursion is a method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem, typically implemented using functions that call themselves.
2. Control starts in the main method and passes to other methods until a method calls itself. The flow is then passed to a new instance of that method, creating a stack of calls. When a base case is reached, control returns to the calling method until the main method is reached.
3. First, there must be a line in the method that calls the method itself. Second, the call MUST be a CONDITIONAL CALL, with something that causes the sequence of calls to the method to cease.
4. The "Russian doll" analogy represents recursion by showing how each doll contains a smaller version of itself, similar to how a recursive function solves a problem by solving a smaller instance. The analogy is flawed because it equates simpler with smaller; some recursive problems don't necessarily get "smaller" in a physical sense but become simpler versions of the original problem.
5. The base case is the non-recursive condition that stops the recursive calls and provides a direct solution. Without it, the recursive function would call itself indefinitely, resulting in a stack overflow error.
6. Recursion can lead to clearer, simpler, and shorter code, making it easier to understand and maintain. However, it can be less efficient in terms of time and space due to function call overhead and the need to store intermediate results on the call stack.
7. The call stack is a data structure that stores information about active function calls, including the return address, parameters, and local variables. In recursion, each recursive call adds a new frame to the stack, and when a base case is reached, the frames are popped off the stack in reverse order, returning control to the calling functions.
8. Factorial calculation is well-suited for a recursive solution, where the factorial of a number N is defined as N multiplied by the factorial of (N-1), with the base case being the factorial of 0 (which is 1).
9. Backtracking is a problem-solving technique that uses recursion to explore all possible solutions by trying different paths and undoing (backtracking) choices that lead to dead ends. Therefore, backtracking relies on recursion to function.

Because recursive functions are clearer, simpler, shorter, and easier to understand than their non-recursive counterparts, the program directly reflects the abstract solution strategy (algorithm). As such, there is a reduction in the cost of maintaining the software.

# 4B Glossary

**Base Case:** A non-recursive way out of a function that allows the routine to work correctly for a simple, non-recursive case [1, 2]. It is a case that can be solved without using recursion [1].

**Backtracking:** A technique that uses recursion to explore all possibilities, undoing bad choices to find solutions by exploring different paths [3, 4]. Backtracking relies on recursion to explore all possibilities [3].

**Conditional Call:** A call to a method that must be conditional to avoid infinite loops, ensuring the sequence of method calls will eventually cease [1, 5].

**Factorial:** The product of all integers from 1 to a given number N, denoted as N! [6]. For example, 5! = 5 \* 4 \* 3 \* 2 \* 1 [6, 7].

**Infinite Loop:** A situation where a method calls itself repeatedly without stopping, which occurs if there isn't a condition to terminate the recursive calls

**Iteration:** An iterative algorithm uses a looping construct. Iteration can be used in place of recursion

**Making Progress:** Each recursive call must make progress toward a base case. Any recursive call must make progress toward a base case

**Recursion:** A method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem. It involves functions that call themselves from within their own code

**Recursive Algorithm:** An algorithm that solves a problem by solving a smaller problem of the same type.

**Recursive Call:** When a method contains a line of code that calls itself, introducing another occurrence of the method.

**Towers of Hanoi:** The minimal number of moves required to solve a Tower of Hanoi puzzle is 2n − 1, where n is the number of disks.

# 5A Advanced Algorithm Design

**1. Algorithm Design and Analysis**

**Definition:** An algorithm is defined as "a solution or mathematical formula to solve a problem, this can usually involve a number of steps." The course focuses on more complex algorithms and algorithm building.

**Algorithm Types Covered:**

* Recursion
* Searching
* Sorting
* Hashing
* Divide and Conquer
* Brute Force
* Greedy
* Backtracking
* Dynamic Programming
* Shortest Path Algorithms (Dijkstra's and A\*)

**Brute Force:**

**Concept:** A straightforward approach that "tries every possible solution until a satisfactory solution is found."

* Useful for small inputs or as a baseline.
* **Example:** Finding the maximum element in an unsorted array by comparing each element with every other (O(n^2) time complexity).
* **Pros:** Guaranteed to find a solution, generic, ideal for simpler problems.
* **Cons:** Inefficient, relies on computing power, slow.
* **Example Code (Java):**

public static int findMax(int[] arr) {

*int max = arr[0];*

*for (int i = 0; i < arr.length; i++) {*

*for (int j = i + 1; j < arr.length; j++) {*

*if (arr[j] > max) {*

*max = arr[j];*

*}*

*}*

*}*

*return max;*

*}*

**Greedy Algorithms:**

**Concept:** "An algorithmic paradigm that follows the problem-solving heuristic of making the locally optimal choice at each stage of the algorithm, with the hope of finding a global optimum solution."

* Well-suited for optimization problems.
* Do not guarantee optimal solutions in all cases.
* **Example:** Cashier's algorithm (making change with the minimum number of coins).
* **Backtracking:**
* **Concept:** "A general algorithmic technique that involves exploring all possible solutions to a problem by incrementally building and testing partial solutions." Used for optimization and decision problems with constraints.
* If a partial solution is invalid, the algorithm backtracks.
* Can be computationally expensive.
* **Example:** N-Queens problem (placing N queens on an N×N chessboard so that no two queens attack each other.)

**Dynamic Programming:**

**Concept:** "A method of solving problems by breaking them down into smaller subproblems and solving each subproblem only once, storing the solution and reusing it whenever it is needed again."

* Used for optimization problems.
* Avoids redundant work by storing solutions to subproblems in a table or array.
* **Types:** Bottom-up and top-down.
* **Examples:** Longest common subsequence, knapsack problem, Fibonacci sequence.
* **Example Code (Java - Fibonacci):**public static int fibonacci(int n) {
* int[] array = new int[n + 1];
* array[0] = 0;
* array[1] = 1;
* for (int i = 2; i <= n; i++) {
* array[i] = array[i-1] + array[i-2];
* }
* return array[n];
* }

**Shortest Path Algorithms:**

**Concept:** Finding the shortest path between two points in a graph.

* **Graphs:** Collection of vertices (nodes) connected by edges (links). Edges have weights representing cost or distance.
* **Algorithms:**Dijkstra's Algorithm: Finds the shortest path in a weighted graph with non-negative edge weights. "Dijkstra's Algorithm basically starts at the node that you choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph."
* A\* Algorithm: A modification of Dijkstra's that uses a heuristic function to estimate the distance to the destination.

**Interfaces and Inheritance in Java**

**Inheritance:**

* A fundamental concept in OOP where a new class (subclass) inherits characteristics (attributes and behaviors) from an existing class (superclass).
* "Inheritance facilitates code reuse, promotes code organization, and supports the concept of hierarchical classification."

**Types:** Single, Multilevel, Hierarchical, Multiple (not directly supported in Java for classes), Hybrid.

* Java doesn't support multiple inheritance for classes, but it supports multiple inheritance through interfaces.
* Uses the extends keyword.

**Interfaces:**

* A reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types.
* "Interfaces are used to define a contract or a set of methods that a class must implement."
* Achieves abstraction and multiple inheritance-like behavior.
* Uses the implements keyword. A class can extend one class but implement many interfaces.
* **Example:**interface Bicycle {
* void changeCadence(int newValue);
* void changeGear(int newValue);
* void speedUp(int increment);
* void applyBrakes(int decrement);
* }
* class ACMEBicycle implements Bicycle { ... }
* "Implementing an interface allows a class to become more formal about the behavior it promises to provide."

**Software Testing**

**Definition:** "Software testing is a process of evaluating a software application or system to detect differences between expected and actual results."

**Goals:**

* Identifying defects
* Validating requirements
* Ensuring quality
* Improving user satisfaction

**Testing Stages:**

**Unit Testing:** Testing individual units or components in isolation. "One of the biggest benefits of this testing phase is that it can be run every time a piece of code is changed, allowing issues to be resolved as quickly as possible." White-box testing is usually used.

**Integration Testing:** Testing the interactions between integrated components. Designed to find interface defects between modules/functions.

**System Testing:** Testing the entire system as a whole. "System testing is undertaken by independent testers who haven’t played a role in developing the program. This testing is performed in an environment that closely mirrors production."

**Acceptance Testing:** Testing to determine if the software meets acceptance criteria and is ready for deployment. User tests the system to find out whether the application meets their business’ needs.

**Regression Testing:** Ensuring that changes do not introduce new defects.

**Performance Testing:** Evaluating performance, scalability, and responsiveness.

**Security Testing:** Identifying and mitigating security vulnerabilities.

**Manual Testing:**

* Done in person, by clicking through the application.
* "Manual testing is a natural part of software development and can be as simple as logging all tests e.g. I have a database, can the user log in – pass."
* **Advantages:** Minimal initial investment, allows human intuition.
* **Disadvantages:** Time-consuming, prone to errors, less efficient for large projects.

**Automated Testing:**

* Performed by a machine executing a test script.
* More robust and reliable than manual tests.
* Requires a testing framework (e.g., PHPUnit, Mocha, RSpec, JUnit).

**Manual vs. Automated Testing**

* The choice depends on project requirements, resources, timeline, complexity, and cost considerations. "Often, a combination of both manual and automated testing approaches yields the best results, leveraging the strengths of each approach to ensure the delivery of high-quality, reliable software products."
* **QA Testing (Quality Assurance Testing):**
* Ensuring the quality and reliability of software.
* Involves test planning, test case development, test execution, defect reporting and tracking, documentation, and continuous improvement.

**Smoke Testing:**

* Basic tests that check the basic functionality of an application. "Smoke tests are meant to be quick to execute, and their goal is to give you the assurance that the major features of your system are working as expected."
* Useful right after a new build or deployment.

**Alpha and Beta Testing:**

**Pecha Kucha:**

* A presentation technique from Japan that follows strict rules: an oral presentation is visually supported by 20 images that are displayed every 20 seconds.
* for developing efficient, maintainable, and reliable software.

# 5A Quiz

1. **What type of algorithm tries every possible solution until a satisfactory one is found?**

a) Greedy Algorithm

b) Dynamic Programming

c) **Brute Force Algorithm**

d) Backtracking Algorithm

1. **What is the time complexity of a brute force algorithm to find the maximum element in an unsorted array?**

a) O(n)

b) **O(n2)**

c) O(log n)

d) O(n log n)

1. **Which algorithmic paradigm makes the locally optimal choice at each stage with the hope of finding a global optimum solution?**

a) Dynamic Programming

b) Backtracking

c) **Greedy Algorithm**

d) Brute Force Algorithm

1. **Which algorithm explores all possible solutions to a problem by incrementally building and testing partial solutions?**

a) Dynamic Programming

b) Greedy Algorithm

c) **Backtracking**

d) Brute Force Algorithm

1. **Which method solves problems by breaking them down into smaller subproblems, solving each only once and storing the solutions?**

a) Greedy Algorithm

b) Backtracking

c) Brute Force Algorithm

d) **Dynamic Programming**

1. **What is the primary goal of software testing?**

a) To write code quickly

b) **To identify defects and ensure the software meets requirements**

c) To design user interfaces

d) To manage project timelines

1. **Which type of testing involves testing individual units or components of the software in isolation?**

a) Integration Testing

b) System Testing

c) Acceptance Testing

d) **Unit Testing**

1. **Which type of testing is conducted to determine whether the system is ready for release?** a) Integration Testing

b) System Testing

c) **Acceptance Testing**

d) Unit Testing

1. **What type of testing is performed by a person clicking through the application?**

a) **Manual Testing**

b) Automated Testing

c) Unit Testing

d) System Testing

**What is the purpose of smoke tests?**

a) To find every possible bug in an application

b) To test the performance of an application under heavy load

c) **To check the basic functionality of an application quickly**

d) To test the security of an application

# 5A Glossary

**Algorithm** A solution or mathematical formula to solve a problem, often involving a number of steps

**Brute Force Algorithm** A straightforward approach that tries every possible solution until a satisfactory one is found. It is guaranteed to find the correct solution but can be inefficient

**Greedy Algorithm** An algorithmic paradigm that makes the locally optimal choice at each stage, hoping to find a global optimum. It's efficient but doesn't guarantee an optimal solution.

**Backtracking** A technique that explores all possible solutions by incrementally building and testing partial solutions. It backtracks when a solution is invalid.

**Dynamic Programming** A method of solving problems by breaking them down into smaller subproblems, solving each subproblem only once, and storing the solutions for reuse.

**Shortest Path Problem** Finding the shortest path between two points in a graph, commonly used in transportation and network design.

**Dijkstra's Algorithm** An algorithm for finding the shortest path in a weighted graph with non-negative edge weights.

**Interface** A reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types [20]. It defines a contract that a class must implement

**Inheritance** A mechanism in OOP that allows a new class to inherit characteristics of an existing class, promoting code reuse and organization.

**Software Testing** Evaluating a software application to detect differences between expected and actual results

**Unit Testing** Testing individual units or components of the software in isolatioN.

**Integration Testing** Testing the interactions and interfaces between integrated components.

**System Testing** Testing the entire system as a whole to validate its behavior and functionality.

**Acceptance Testing** Testing conducted to determine whether the software meets the acceptance criteria and is ready for deployment.

**Regression Testing** Testing to ensure that changes or updates do not introduce new defectS.

**Performance Testing** Evaluating the performance, scalability, and responsiveness of the software under various conditions.

**Security Testing** Identifying and mitigating security vulnerabilities.

**Manual Testing** Testing done in person, by clicking through the application or interacting with the software and APIs with the appropriate tooling.

**Automated Testing** Testing performed by a machine that executes a test script written in advance.

**QA Testing** Quality Assurance testing, ensuring the quality and reliability of software products.

**Smoke Testing** Basic tests that check the major features of a system are working as expected.

Top of Form

Top of Form

Top of Form