Data Structures and Algorithms in

Nutshell

Sajak Basnet

**What is Data? And its types in Computer science?**

In a narrower sense, data refers specifically to raw facts, numbers, characters, symbols, or values that are collected, stored and processed by a computer system.

Data, in its raw form, encompasses information collected from various sources like sensors, user input, databases, and software systems. It can take on different representations depending on its type, such as numeric values, text strings, or symbols. Computer programs manipulate data using specific algorithms and operations tailored to its type, enabling tasks like arithmetic operations on numeric data or string manipulation on text. Stored in computer memory or storage devices, efficient organization and retrieval of data are pivotal for software functionality and user value. As the foundation of computer science and programming, data underpins nearly all software applications and systems, driving their operations and outcomes.

In the context of computer science and programming, data can be broadly categorized into several types based on its characteristics and usage. Here are some common types of data:

1. **Primitive Data Types**: These are the basic building blocks of data and include:

- Integer: Whole numbers without decimal points (e.g., 5, -10, 100).

- Floating-point: Numbers with decimal points (e.g., 3.14, -0.5, 2.0).

- Character: Single characters (e.g., 'A', 'b', '$').

- Boolean: Represents true or false values.

2. **Composite Data Types**: These types consist of collections of primitive data types and include:

- Arrays: Ordered collections of elements of the same data type.

- Lists: Ordered collections of elements that may or may not be of the same data type.

- Sets: Unordered collections of unique elements.

- Dictionaries (Maps or Hash Tables): Collections of key-value pairs, where each key is associated with a value.

- Tuples: Ordered collections of elements of different data types.

3. **String**: A sequence of characters, often used to represent text data.

4. **Files and Streams**: Data stored in files or streams, which can include text, binary data, images, audio, video, etc.

5. **Structured Data**: Data organized into tables or records, commonly found in databases.

6. **Unstructured Data**: Data that does not have a predefined data model or organization, such as text documents, emails, social media posts, etc.

7. **Numeric Data**: Data that consists of numerical values and can be used for mathematical operations and calculations.

8. **Date and Time Data**: Data representing dates, times, durations, and intervals.

Understanding the types of data is crucial for effectively representing and manipulating information within computer programs and systems.

**What are Algorithms?**

In simple words algorithms are step-by-step instruction for solving certain problems.

In technical sense, algorithms are the required and detailed step to solve computation problems. These computational problems are specifically those that can be solved by computers.

Algorithms have several key traits that make them effectively for solving problem. Here are some of the most important ones:

* **Input and output :** An algorithmalways takes input (data it operates on) and gives specific output (result).
* **Definite and Unambiguous steps:** The instruction in an algorithm are clear and leave no room for misinterpretation. Each step should have a single outcome.
* **Finiteness:** An algorithm terminates after finite number of steps, meaning it eventually reaches an end and doesn’t run forever.
* **Effectiveness:** An algorithm must be practical and effieient in solving the problem its’s designed for.
* **Generality:** In some cases, algorithms can be designed to solve a range of similar problems, not just one specific instance.

**What are Data Structures? and its types.**

Data Structures is the organized format of data. Think of it as a container that holds some thing in structured and well-organized format. And in more technical terms Data structures can be explained as the format to store data some common Data structures include array, stack, queue, graphs, and trees and so on. These are some of the most common data structures in computer science. These data structures are used to store data. Each data structure has its own unique properties hence differs from one another. Furthermore, the above Data structures I mentioned are classified into two types which includes, Linear Data structure and non-linear data structure. linear data structure stores data in linear fashion (one dimension) and likewise, non-linear data structure stores in non-linear fashion (two dimension) which makes non-linear ds a bit more complex type of data structure to implement. On these data structures various algorithms can be implemented. For example, traversing, sorting and so on.

Data structures can be classified into several types based on how they organize and store data. The main types of data structures include:

1. **Primitive Data Structures**: These are the basic data structures provided by programming languages, such as integers, floating-point numbers, characters, and Booleans.

2. **Linear Data Structures**:

- **Arrays**: Contiguous blocks of memory where elements are stored sequentially and accessed by index.

- **Linked Lists**: Elements are stored in nodes, each containing a reference to the next node in the sequence.

- **Stacks**: Follows the Last-In, First-Out (LIFO) principle, where elements are added and removed from the same end.

- **Queues**: Follows the First-In, First-Out (FIFO) principle, where elements are added at the rear and removed from the front.

3. **Non-Linear Data Structures**:

- **Trees**: Hierarchical structures with nodes organized in a parent-child relationship. Examples include binary trees, binary search trees, AVL trees, and B-trees.

- **Graphs**: Collections of nodes (vertices) connected by edges, allowing for complex relationships between elements.

4. **Hashed Data Structures**:

- **Hash Tables**: Uses a hash function to map keys to values, providing efficient lookup, insertion, and deletion operations.

5. **Specialized Data Structures:**

**- Heaps**: Specialized tree-based data structures used for priority queue operations.

- **Tries (Prefix Trees):** Efficient for storing and retrieving strings based on their prefixes.

- **Bloom Filters**: Space-efficient probabilistic data structures used to test whether an element is a member of a set.

This is just the basic overview of different Data Structures later we will discuss each in detail.

**Concept of Abstract Data Type (ADT)**

An Abstract Data Type (ADT) is a theoretical concept in computer science that defines a set of operations on a data type without specifying how these operations are implemented. In other words, an ADT focuses on the behaviour and functionality of a data type rather than its implementation details.

An ADT encapsulates data and the operations that can be performed on that data, providing a clear interface for interacting with the data. It allows programmers to work with data structures at a higher level of abstraction, without being concerned about the underlying implementation.

Here's an example of an Abstract Data Type:

Consider the ADT of a Stack.

A stack is a collection of elements with two primary operations:

1. **Push**: Adds an element to the top of the stack.

2. **Pop**: Removes and returns the element at the top of the stack.

In addition to these primary operations, a stack typically includes other operations such as:

3. **Peek**: Returns the element at the top of the stack without removing it.

4. **isEmpty**: Checks if the stack is empty.

5. **Size**: Returns the number of elements in the stack.

Here's how the operations would be defined for the abstract data type "Stack":

class Stack:

    def \_\_init\_\_(self):

        self.stack = []

    def push(self, item):

        self.stack.append(item)

    def pop(self):

        if not self.is\_empty():

            return self.stack.pop()

    def peek(self):

        if not self.is\_empty():

            return self.stack[-1]

    def is\_empty(self):

        return len(self.stack) == 0

    def size(self):

        return len(self.stack)

In this example, the Stack ADT is defined with its operations: **push**, **pop**, **peek**, **is\_empty**, and **size**. Users of the stack data structure can utilize these operations without needing to know how they are implemented internally. The implementation details can vary, for example, the stack can be implemented using a list, linked list, or array, but the interface remains consistent regardless of the underlyingimplementation. This abstraction allows for flexibility and modularity in software design.

**Algorithm Efficiency and Complexity**

As we have already discussed what algorithms are now let’s understand what algorithm efficiency and complexity means.

Algorithm Efficiency and Complexity are two sides of the same coin when it comes to analysing how well an algorithm performs.

Algorithm Efficiency simply tells how well an algorithm utilizes computing resources like time and memory to solve a problem. An efficient algorithm solves the problem quickly and uses minimal memory.

And likewise, Algorithm Complexity is a tool used to measure efficiency of an algorithm in relation to the size of the input data (often termed n). It helps to understand how algorithm’s performance scales as the input grows.

Now let’s talk about time and space analysis of an algorithm.

Time and space analysis or simpley time and space complexities. Understanding these complexities is crucial for evaluation an algorithm’s efficiency and choosing the best tool for the job.

Imagine a scenario where You’re at a restaurant with a massive menu. A brute force approach would be to meticulously scan every single dish , taking a considerable amount of time (linear search, O(n)). But what if you have dietary restrictions? You can significantly reduce your search time by filtering based on those criteria (binary search, O(log n)). This analogy beautifully illustrates the concept of time complexity.

Time Complexity: Its tool to measure the execution time of an algorithm relative to the input size (often denoted by n). It tells us how the algorithm’s workload scales as the problem grows bigger. Think of it as the time it takes you to finish your exam, which depends on the number of questions (input size). We express time complexity using Big O Notation which represents the upper bound of an algorithm’s execution time. It ignores constant factors and lower-order terms, focusing on the dominant factor as the input grows infinitely large.

Common Time Complexities:

1. O(1): Constant time : the execution times remains constant regardless of the input size.
2. O(n): Linear Time: the execution time grow proportionally to the input size.
3. O(n^2): Quadratic time: The execution time grow quadratically with the input size. Most common example include nested loops.
4. O(log n): Logarithmic Time: The execution time grow logarithmically with the input size. This is a much faster growth rate compared to linear or quadratic.

Space Complexity: this refers to the amount of additional memory an algorithm requires, besides the input data itself, to execute. Just like with your exam, you might need rough paper (space) to perform some rough calculations, but the number of questions (input) doesn’t affect how much rough paper you need. Space complexity is also expressed using Big O Notation, focusing on the extra space needed as the input size increases.

Let’s try to understand with example:

Consider two sorting algorithms: Bubble Sort and Merge Sort.

* **Bubble Sort (Time Complexity: O(n^2), Space Complexity: O(1))**
  + It repeatedly iterates through the list, comparing adjacent elements and swapping them if they're in the wrong order. In the worst case, it might need to perform n\*(n-1)/2 comparisons, leading to a quadratic time complexity. However, it only uses constant additional space for temporary variables.
* **Merge Sort (Time Complexity: O(n log n), Space Complexity: O(n))**
  + It works by recursively dividing the list into halves, sorting them independently, and then merging the sorted halves. This divide-and-conquer approach leads to a more efficient time complexity of O(n log n). However, it requires additional space (proportional to the input size) to store the temporary sub-lists during the merge process.

**Choosing the Right Algorithm:**

The choice between algorithms often hinges on a trade-off between time and space complexity. For massive datasets, prioritizing an algorithm with a lower time complexity (like Merge Sort) might be more crucial, even if it requires more space. But for smaller datasets or situations with limited memory, a space-efficient algorithm (like Bubble Sort) might be preferred, even if it's slower.

**Conclusion:**

Time and space complexity analysis equips you with the tools to assess an algorithm's efficiency and make informed decisions when tackling computational problems. Remember, the best algorithm choice depends on the specific problem, data size, and resource constraints. So, the next time you encounter an algorithm, don't just take its functionality at face value, delve deeper and analyse its time and space complexity!