Data Structures and Algorithms

To develop an understanding of data structures and algorithms requires:

1. Understanding how information is arranged in memory
2. Familiarity with the algorithms manipulating the information within the data structure
3. The performance characteristics of the data structure so that when called upon a suitable data structure can be selected.

Learns

* Basic Principles of algorithm design
* How to represent well-known data structures in Python
* How to implement well-known algorithms in Python
* How to transform new problems into well-known algorithmic problems with efficient solutions
* How to analyse algorithms and python programs using both mathematical tools and basic experiments and benchmarks.
* How to understand several classical algorithms and data structures in depth, and be able to implement these efficiently in Python.

Does not cover:

* Numerical algorithms
* Parallel algorithms
* Multi-core algorithms

Chapters

**Data types**

* Primitive data types (system-defined)
* User-defined data types

**Data Structures**

* Linear Data Structures – Linked Lists, Stacks, Queues
* Non-linear Data Structures – Trees and Graphs

**Abstract Data Types**

User-defined data types are defined with their operations (primitives operations are already defined by the system). To simplify problem solving data structures are combined with their operations called ADT. ADTs consist of 2 parts:

1. Declaration of Data
2. Declaration of Operations

Commonly used ADTs include linked lists, stacks, queues, priority queues, binary trees, dictionaries, disjoint sets, hash tables, graphs etc etc…

Essentially an ADT describe the wrapper around a data structure, concerning itself more with the operations that can be performed on the underlying data (create, delete, etc) than how those methods are implemented.

**Algorithms**

A step-by-step list of instructions to solve a given problem.

Analysis of algorithms helps establish how fast, reliable, easy to implement etc. an algorithm is.

This is normally concerned with speed or ‘running time analysis', this is how long processing increases as the size of the problem increases (input size).

Algorithms can be compared via various objective measures:

* Execution times (not good as specific to individual machines)
* Statements executed (also not good as statement exectuions vary between languages and code)

**Rates of growth (Big O notation – Upper Bound) (Omega Notation – Lower Bound) (Theta Notation - Average)**

|  |  |  |
| --- | --- | --- |
| Time Complexity | Name | Example |
| 1 | Constant | Add element to front of list |
| logn | Logarithmic | Element in sorted array |
| n | Linear | Element in unsorted array |
| nlogn | Linear logarithmic | Divide and conquer |
| n^2 | Quadratic | Shortest path |
| n^3 | Cubic | Matrix multiplication |
| 2n | exponential | Towers of Hanoi |

analysis of an algorithm is usually given in terms of the worst case, best case and average scenario.

Ǝ -> there exists

⟹ -> implies that

∴ -> therefore

Generally:

* A loop is a constant O(n)
* A nested loop is O(n^2)
* If else O(n)
* Logarithmic complexity O(logn)

**Recursion**

Any function calling itself is recursive.

Recursive functions must terminate.

Examples:

* Fibonacci Series
* Merge Sort, Quick Sort
* Binary Search
* Tree Traversal (InOrder, PreOrder, PostOrder)
* Graph Traversals (Depth first search, breadth first search)
* Divide and conquer algorithms
* Towers of Hanoi
* Backtracking

Linked Lists

Stacks

Queues

Trees

Priority Queues

Graph Algorithms

Disjoint Set ADT

Sorting Algorithms

Searching Algorithms

Symbols Tables

Hashing

String Algorithms

Algorithm Design Techniques

Greedy Algorithms

Divide and Conquer

Dynamic Programming

Complexity Classes