**Data Structures & Algorithms**

**Interface** – each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.

**Implementation** – implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

**Characteristics of a Data Structure**

**Correctness:** data structure implementation should implement its interface correctly

**Time complexity:** Running time or the execution time of operations of data structure must be as small as possible

**Space complexity:** memory usage of a data structure operation should be as little as possible

**Need for Data Structure**

Three common problems for modern day applications:

**Data Search**: Consider an inventory of 1million items of a store. If the application is to search an item, it has to search an item in 1 million items every time, slowing down the search. As data grows, search will become slower

**Processor Speed**: processor speed although being high, falls limited if the data grows to billion records

**Multiple Requests**: As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data

**Basic Terminology**

**Data** – values or set of values

**Data Item** – refers to single unit of values

**Group Items** – Data items that are divided into sub items

**Elementary Items** – Data items that cannot be divided

**Attribute and Entity** – an entity is that which contains certain attributes or properties, which may be assigned values

**Entity Set** – entities of similar attributes form an entity set

**Field** – a single elementary unit of information representing an attribute of an entity

**Record** – a collection of field values of a given entity

**File** – a collection of records of the entities in a given entity set

**Algorithm**

Some important categories of algorithms:

**Search** – algorithm to search an item in a data structure

**Sort** – algorithm to sort items in a certain order

**Insert** – algorithm to insert item in a data structure

**Update** – algorithm to update an existing item in a data structure

**Delete** – algorithm to delete an existing item from a data structure

**Characteristics of an Algorithm**

**Unambiguous** – algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning

**Input** – an algorithm should have 0 or more well-defined inputs

**Output** – an algorithm should have 1 or more well-defined outputs, and should match the desired output

**Finiteness** – algorithms must terminate after a finite number of steps

**Feasibility** – should be feasible with the available resources

**Independent** – an algorithm should have step-by-step directions, which should be independent of any programming code

**Writing an Algorithm**

All programing languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc.; these common constructs can be used to write an algorithm

Example:

Problem – design an algorithm to add two number sand display the result

Step1: start

Step2: declare three ints a, b, c

Step3: define values of a & b

Step4: add values of a & b

Step5: store output of step4 to c

Step6: print c

Step7: stop

OR

Step1: start add

Step2: get values of a & b

Step3: c = a & b

Step4: display c

Step5: stop

Algorithm is designed to get a solution of a given problem. The problem can be solved in more than one ways. Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution

**Algorithm Analysis**

Two different analysis stages:

**A Priori Analysis** – this is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation

**A Posterior Analysis** – this is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required are collected.

**Algorithm Complexity**

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

**Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm

**Space Factor** – Space is measured by counting the maximum memory space required by the algorithm

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

**Space Complexity**

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

A fixed part that is a space required to store certain data and variables, which are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

A variable part is a space required by variables, whose size depends on t size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Example: Algorithm: SUM(A, B)

Step1: Start Step2: C=A+B+10 Step3: Stop

Three variables A, B, C and one constant; Hence S(P) = 1 + 3.

**Time Complexity**

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

E.g.: addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c \* n, where c is the time taken for the addition of two bits. Here, T(n) grows linearly as the input size increases.

**Asymptotic Analysis**

Asymptotic analysis of an algorithm refers to defining the mathematical bound/framing of its run-time performance. Asymptotic analysis is input bound i.e.: if there is no input to the algorithm, it is concluded to work in a constant time.

Three commonly used asymptotic notations: O Notation, Ω Omega Notation, θ Theta Notation

**Big O Notation, O**

The notation O(n) is the formal way to express the upper bound of an algorithm’s running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

O – upper bound, worst case time measure

Omega – lower bound, best case time measure

Theta – express both the lower bound and the upper bound

**Common Asymptotic Notations**

Constant O(1)

Logarithmic O(log n)

Linear O(n)

N log n O(n log n)

Quadratic O(n^2)

Cubic O(n^3)

Polynomial n^ (O(1))

Exponential 2^ (O(n))

**Greedy Algorithms**

In greedy algorithm approach, decisions are made from the given solution domain. Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally, greedy algorithms do not provide globally optimized solutions.

Example: **Counting Coins**

Count to desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. E.g. provided coins of 1, 2, 5, 10, and asked to count 18, the greedy procedure would be to – select one 10 (remaining 8), select one 5(remaining 3), select 2 (remaining 1), and select one 1.

In case if the coins were 1, 7, 10, and the count was also 18, the same approach would work, but it may not be optimum for counts like 15. The greedy approach would be to use one 10 and five 1s, but the same issue can be solved by taking two 7s and one 1.

Therefore, the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern

Most networking algorithms use greedy approach. Some are:

Traveling Salesman Problem, Prim’s Minimal Spanning Tree Algorithm, Kruskal’s minimal Spanning Tree Algorithm, Dijkstra’s Minimal Spanning Tree algorithm, Graph – Map Coloring, Graph – Vertex Cover, Knapsack Problem, Job Scheduling Problem

**Divide and Conquer**

With divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When dividing sub-problems into even small sub-problems, a stage where no more division is possible can be eventually reached. Those “atomic” smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of the original problem

Examples

Merge sort, quick sort, binary search, Strassen’s Matrix Multiplication, Closest pair (points)

**Dynamic Programming**

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. Unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used with problems that can be divided into similar sub-problems so that their results can be re-used. Mostly, the algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

**Comparison** – In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for an overall optimization of the problem.

In contrast to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use memorization to remember the output of already solved sub-problems.

Examples

Fibonacci number series, Knapsack problem, Tower of Hanoi, All pair shortest path by Floyd-Warshall, Shortest path by Dijkstra, Project Scheduling

Dynamic programming can be used in both top-down and bottom-up manner

**Data Structure**

Atomic – Definition should define a single concept

Traceable – Definition should be able to be mapped to some data element

Accurate – Definition should be unambiguous

Clear and Concise

Two data types: Built-in data type, Derived data type

**Build-in data type**

Data types for which a language has built-in support

Integers, Boolean, floating, char and strings, etc.

**Derived data type**

Data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. Normally built by the combination of primary or built-in data types and associated operations on them.

List, array, stack, queue

**Basic operations** – traversing, searching, insertion, deletion, sorting, merging