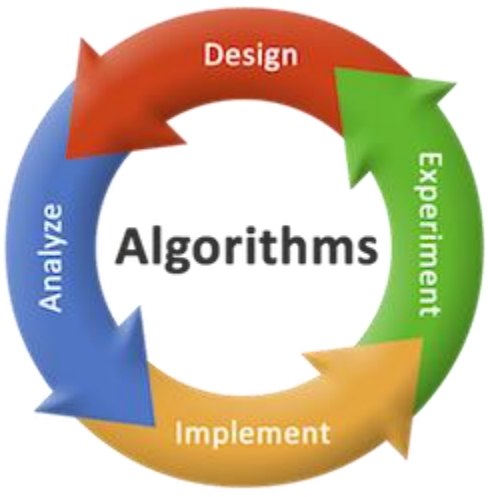
DAA Tutorial



Our DAA Tutorial is designed for beginners and professionals both.

Our DAA Tutorial includes all topics of algorithm, asymptotic analysis, algorithm control structure, recurrence, master method, recursion tree method, simple sorting algorithm, bubble sort, selection sort, insertion sort, divide and conquer, binary search, merge sort, counting sort, lower bound theory etc.

What is Algorithm?

A finite set of instruction that specifies a sequence of operation is to be carried out in order to solve a specific problem or class of problems is called an Algorithm.

Why study Algorithm?

As the speed of processor increases, performance is frequently said to be less central than other software quality characteristics (e.g. security, extensibility, reusability etc.). However, large problem sizes are commonplace in the area of computational science, which makes performance a very important factor. This is because longer computation time, to name a few mean slower results, less through research and higher cost of computation (if buying CPU Hours from an external party). The study of Algorithm, therefore, gives us a language to express performance as a function of problem size.

DAA Algorithm

The word algorithm has been derived from the Persian author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics. The word is taken based on providing a special significance in computer science. The algorithm is understood as a method that can be utilized by the computer as when required to provide solutions to a particular problem.

An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step.

An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

An algorithm unravels the computational problems to output the desired result. An algorithm can be described by incorporating a natural language such as English, Computer language, or a hardware language.

Characteristics of Algorithms

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

Pseudocode

Pseudocode refers to an informal high-level description of the operating principle of a computer program or other algorithm. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

Advantages of Pseudocode

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.
- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.
- It can easily detect an error before transforming it into a code.

Disadvantages of Pseudocode

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.
- It does not depict the design.

Difference between Algorithm and the Pseudocode

An algorithm is simply a problem-solving process, which is used not only in computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution. It not only helps in simplifying the problem but also to have a better understanding of it.

However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax. It encompasses semi-mathematical statements.

Problem: Suppose there are 60 students in the class. How will you calculate the number of absentees in the class?

Pseudo Approach:

1. Initialize a variable called as **Count** to zero, **absent** to zero, **total** to 60
2. FOR EACH Student PRESENT DO the following:
   Increase the **Count** by One
3. Then Subtract **Count** from **total** and store the result in **absent**
4. Display the number of absent students

Algorithmic Approach:

1. Count <- 0, absent <- 0, total <- 60
2. REPEAT till all students counted
   Count <- Count + 1

3. absent <- total - Count
4. Print "Number absent is:" , absent

Need of Algorithm

1. To understand the basic idea of the problem.

2. To find an approach to solve the problem.

3. To improve the efficiency of existing techniques.

4. To understand the basic principles of designing the algorithms.

5. To compare the performance of the algorithm with respect to other techniques.

6. It is the best method of description without describing the implementation detail.

7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.

8. A good design can produce a good solution.

9. To understand the flow of the problem.

10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)

11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.

12. With the help of algorithm, we convert art into a science.

13. To understand the principle of designing.

14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Analysis of algorithm

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which we can analysis the algorithm:

- **Space Complexity:** The space complexity can be understood as the amount of space required by an algorithm to run to completion.
- **Time Complexity:** Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

Let's understand it with an example.

Suppose there is a problem to solve in Computer Science, and in general, we solve a program by writing a program. If you want to write a program in some programming language like C, then before writing a program, it is necessary to write a blueprint in an informal language.

Or in other words, you should describe what you want to include in your code in an English-like language for it to be more readable and understandable before implementing it, which is nothing but the concept of Algorithm.

In general, if there is a problem **P1**, then it may have many solutions, such that each of these solutions is regarded as an algorithm. So, there may be many algorithms such as $\mathbf{A_1, A_2, A_3}$, ..., $\mathbf{A_n}$.

Before you implement any algorithm as a program, it is better to find out which among these algorithms are good in terms of time and memory.

It would be best to analyze every algorithm in terms of **Time** that relates to which one could execute faster and **Memory** corresponding to which one will take less memory.

So, the Design and Analysis of Algorithm talks about how to design various algorithms and how to analyze them. After designing and analyzing, choose the best algorithm that takes the least time and the least memory and then implement it as a program in C.

In this course, we will be focusing more on time rather than space because time is instead a more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we are running an algorithm on a particular platform, we are more or less stuck with the performance that platform can give us in terms of speed.

However, on the other hand, memory is relatively more flexible. We can increase the memory as when required by simply adding a memory card. So, we will focus on time than that of the space.

The running time is measured in terms of a particular piece of hardware, not a robust measure. When we run the same algorithm on a different computer or use different programming languages, we will encounter that the same algorithm takes a different time.

Generally, we make three types of analysis, which is as follows:

- **Worst-case time complexity:** For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of n.
- **Average case time complexity:** For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.
- **Best case time complexity:** For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

Complexity of Algorithm

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

Typical Complexities of an Algorithm

- **Constant Complexity:**
  It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.
- **Logarithmic Complexity:**
  It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.
  For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.
- **Linear Complexity:**
  - It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements.
    For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.
  - It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem.
    For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

- **Quadratic Complexity:** It imposes a complexity of **O(n^2)**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
  If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity. For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.
- **Cubic Complexity:** It imposes a complexity of **O(n^3)**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
  For example, if there exist 100 elements, it is going to execute 1,000,000 steps.
- **Exponential Complexity:** It imposes a complexity of **O(2^n), O(N!), O(n^k), ...**. For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
  For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits. The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them. Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem.

How to approximate the time taken by the Algorithm?

So, to find it out, we shall first understand the types of the algorithm we have. There are two types of algorithms:

1. **Iterative Algorithm:** In the iterative approach, the function repeatedly runs until the condition is met or it fails. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the function calls itself until the condition is met. It integrates the branching structure.

However, it is worth noting that any program that is written in iteration could be written as recursion. Likewise, a recursive program can be converted to iteration, making both of these algorithms equivalent to each other.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in the recursive program, we use recursive equations, i.e., we write a function of F(n) in terms of F(n/2).

Suppose the program is neither iterative nor recursive. In that case, it can be concluded that there is no dependency of the running time on the input data size, i.e., whatever is the input size, the running time is going to be a constant value. Thus, for such programs, the complexity will be **O(1)**.

For Iterative Programs

Consider the following programs that are written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an integer i and a for loop running from i equals 1 to n. Now the question arises, how many times does the name get printed?

```
1. A()
2. {
3. int i;
4. for (i=1 to n)
5. printf("Edward");
6. }
```

Since i equals 1 to n, so the above program will print Edward, n number of times. Thus, the complexity will be **O(n)**.

Example2:

```
1. A()
2. {
3. int i, j:
4. for (i=1 to n)
5. for (j=1 to n)
6. printf("Edward");
```

```
7. }
```

In this case, firstly, the outer loop will run n times, such that for each time, the inner loop will also run n times. Thus, the time complexity will be **$O(n^2)$**.

Example3:

```
1. A()
2. {
3. i = 1; S = 1;
4. while (S<=n)
5. {
6. i++;
7. SS = S + i;
8. printf("Edward");
9. }
10. }
```

As we can see from the above example, we have two variables; i, S and then we have while S<=n, which means S will start at 1, and the entire loop will stop whenever S value reaches a point where S becomes greater than n.

Here i is incrementing in steps of one, and S will increment by the value of i, i.e., the increment in i is linear. However, the increment in S depends on the i.

Initially;

i=1, S=1

After 1st iteration;

i=2, S=3

After 2nd iteration;

i=3, S=6

After 3rd iteration;

i=4, S=10 … and so on.

Since we don't know the value of n, so let's suppose it to be k. Now, if we notice the value of S in the above case is increasing; for i=1, S=1; i=2, S=3; i=3, S=6; i=4, S=10; …

Thus, it is nothing but a series of the sum of first n natural numbers, i.e., by the time i reaches k, the value of S will be k(k+1)/2.

To stop the loop, $\frac{k(k+1)}{2}$ has to be greater than n, and when we solve this equation, we will get $\frac{k^2+k}{2} > n$. Hence, it can be concluded that we get a complexity of **$O(\sqrt{n})$** in this case.

For Recursive Program

Consider the following recursive programs.

Example1:

```
1. A(n)
2. {
3. if (n>1)
4. return (A(n-1))
5. }
```

Solution;

Here we will see the simple Back Substitution method to solve the above problem.

T(n) = 1 + T(n-1) …Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) ...Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) ...Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

T(n)= 1 + 1+ T(n-2) = 2 + T(n-2) ...Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = ... = k + T(n-k) ...Eqn. (5)

Now, according to Eqn. (1), i.e. T(n) = 1 + T(n-1), the algorithm will run until n>1. Basically, n will start from a very large number, and it will decrease gradually. So, when T(n) = 1, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for k = n-1, the T(n) will become.

Step5: Substitute k = n-1 in eqn. (5)

T(n) = (n-1) + T(n-(n-1)) = (n-1) + T(1) = n-1+1

Hence, T(n) = n or **O(n).**

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

3. Dynamic Programming: Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is particularly helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to **Optimization Problems**.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

6. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn't work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

7. Randomized Algorithm: A randomized algorithm uses a random number at least once during the computation make a decision.

Example 1: In Quick Sort, using a random number to choose a pivot.

Example 2: Trying to factor a large number by choosing a random number as possible divisors.

Loop invariants

This is a justification technique. We use loop invariant that helps us to understand why an algorithm is correct. To prove statement S about a loop is correct, define S concerning series of smaller statement S_0 S_1....S_k where,

- The initial claim so is true before the loop begins.
- If S_{i-1} is true before iteration i begin, then one can show that S_i will be true after iteration i is over.
- The final statement S_k implies the statement S that we wish to justify as being true.