Design and Analysis of Algorithms

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About this Tutorial

An Algorithm is a sequence of steps to solve a problem. Design and Analysis of Algorithm is very important for designing algorithm to solve different types of problems in the branch of computer science and information technology.

This tutorial introduces the fundamental concepts of Designing Strategies, Complexity analysis of Algorithms, followed by problems on Graph Theory and Sorting methods. This tutorial also includes the basic concepts on Complexity theory.

Audience

This tutorial has been designed for students pursuing a degree in any computer science, engineering, and/or information technology related fields. It attempts to help students to grasp the essential concepts involved in algorithm design.

Prerequisites

The readers should have basic knowledge of programming and mathematics. The readers should know data structure very well. Moreover, it is preferred if the readers have basic understanding of Formal Language and Automata Theory.

Copyright & Disclaimer

Table of Contents

Basics of Algorithms

1. DAA – Introduction

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

An algorithm is the best way to represent the solution of a particular problem in a very simple and efficient way. If we have an algorithm for a specific problem, then we can implement it in any programming language, meaning that the **algorithm is independent from any programming languages**.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient. However, one has to keep in mind that both time consumption and memory usage cannot be optimized simultaneously. If we require an algorithm to run in lesser time, we have to invest in more memory and if we require an algorithm to run with lesser memory, we need to have more time.

Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

Characteristics of Algorithms

The main characteristics of algorithms are as follows:

- Algorithms must have a unique name
- Algorithms should have explicitly defined set of inputs and outputs
- Algorithms are well-ordered with unambiguous operations
- Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudocode

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

The running time can be estimated in a more general manner by using Pseudocode to represent the algorithm as a set of fundamental operations which can then be counted.

Difference between Algorithm and Pseudocode

An algorithm is a formal definition with some specific characteristics that describes a process, which could be executed by a Turing-complete computer machine to perform a specific task. Generally, the word "algorithm" can be used to describe any high level task in computer science.

On the other hand, pseudocode is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it. Writing a pseudocode has no restriction of styles and its only objective is to describe the high level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

```
Algorithm: Insertion-Sort
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop
```

Here is a pseudocode which describes how the high level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i ← 1 to length(A)
    x ← A[i]
    j ← i
```

```
    while j > 0 and A[j-1] > x
        A[j] ← A[j-1]
        j ← j - 1
    A[j] ← x
```

In this tutorial, algorithms will be presented in the form of pseudocode, that is similar in many respects to C, C++, Java, Python, and other programming languages.

2. DAA – Analysis of Algorithms

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. The term **"analysis of algorithms"** was coined by Donald Knuth.

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Most algorithms are designed to work with inputs of arbitrary length. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps, known as **time complexity**, or volume of memory, known as **space complexity**.

The Need for Analysis

In this chapter, we will discuss the need for analysis of algorithms and how to choose a better algorithm for a particular problem as one computational problem can be solved by different algorithms.

By considering an algorithm for a specific problem, we can begin to develop pattern recognition so that similar types of problems can be solved by the help of this algorithm.

Algorithms are often quite different from one another, though the objective of these algorithms are the same. For example, we know that a set of numbers can be sorted using different algorithms. Number of comparisons performed by one algorithm may vary with others for the same input. Hence, time complexity of those algorithms may differ. At the same time, we need to calculate the memory space required by each algorithm.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the time and size required (the size of memory for storage while implementation). However, the main concern of analysis of algorithms is the required time or performance. Generally, we perform the following types of analysis:

- **Worst-case**: The maximum number of steps taken on any instance of size **a**.
- **Best-case**: The minimum number of steps taken on any instance of size **a**.
- **Average case**: An average number of steps taken on any instance of size **a**.
- **Amortized**: A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider time as well as space complexity as the program may run on a system where memory is limited but adequate space is available or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort.** Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

3. DAA – Methodology of Analysis

To measure resource consumption of an algorithm, different strategies are used as discussed in this chapter.

Asymptotic Analysis

The asymptotic behavior of a function $f(n)$ refers to the growth of $f(n)$ as **n** gets large.

We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs.

A good rule of thumb is that the slower the asymptotic growth rate, the better the algorithm. Though it's not always true.

For example, a linear algorithm $f(n) = d * n + k$ is always asymptotically better than a quadratic one, $f(n) = c.n^2 + q$.

Solving Recurrence Equations

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs. Recurrences are generally used in divide-and-conquer paradigm.

Let us consider $T(n)$ to be the running time on a problem of size **n**.

If the problem size is small enough, say $n < c$ where **c** is a constant, the straightforward solution takes constant time, which is written as $\theta(1)$. If the division of the problem yields a number of sub-problems with size $\frac{n}{b}$.

To solve the problem, the required time is $a.T(n/b)$. If we consider the time required for division is $D(n)$ and the time required for combining the results of sub-problems is $C(n)$, the recurrence relation can be represented as:

$$T(n) = \begin{cases} \theta(1) & if\ n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$$

A recurrence relation can be solved using the following methods:

- **Substitution Method** – In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.

- **Recursion Tree Method** – In this method, a recurrence tree is formed where each node represents the cost.

- **Master's Theorem** – This is another important technique to find the complexity of a recurrence relation.

Amortized Analysis

Amortized analysis is generally used for certain algorithms where a sequence of similar operations are performed.

- Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.

- Amortized analysis differs from average-case analysis; probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

It is not just a tool for analysis, it's a way of thinking about the design, since designing and analysis are closely related.

Aggregate Method

The aggregate method gives a global view of a problem. In this method, if **n** operations takes worst-case time $\boldsymbol{T(n)}$ in total. Then the amortized cost of each operation is $\boldsymbol{T(n)/n}$. Though different operations may take different time, in this method varying cost is neglected.

Accounting Method

In this method, different charges are assigned to different operations according to their actual cost. If the amortized cost of an operation exceeds its actual cost, the difference is assigned to the object as credit. This credit helps to pay for later operations for which the amortized cost less than actual cost.

If the actual cost and the amortized cost of **i^th^** operation are $\boldsymbol{c_i}$ and $\boldsymbol{\widehat{c_i}}$, then

$$\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$$

Potential Method

This method represents the prepaid work as potential energy, instead of considering prepaid work as credit. This energy can be released to pay for future operations.

If we perform $\boldsymbol{n}$ operations starting with an initial data structure $\boldsymbol{D_0}$. Let us consider, $\boldsymbol{c_i}$ as the actual cost and $\boldsymbol{D_i}$ as data structure of **i^th^** operation. The potential function ϕ maps to a real number $\phi(\boldsymbol{D_i})$, the associated potential of $\boldsymbol{D_i}$. The amortized cost $\boldsymbol{\widehat{c_i}}$ can be defined by

$$\widehat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$$

Hence, the total amortized cost is

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_i + \phi(D_i) - \phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0)$$

Dynamic Table

If the allocated space for the table is not enough, we must copy the table into larger size table. Similarly, if large number of members are erased from the table, it is a good idea to reallocate the table with a smaller size.

Using amortized analysis, we can show that the amortized cost of insertion and deletion is constant and unused space in a dynamic table never exceeds a constant fraction of the total space.

In the next chapter of this tutorial, we will discuss Asymptotic Notations in brief.

4. DAA – Asymptotic Notations & Apriori Analysis

In designing of Algorithm, complexity analysis of an algorithm is an essential aspect. Mainly, algorithmic complexity is concerned about its performance, how fast or slow it works.

The complexity of an algorithm describes the efficiency of the algorithm in terms of the amount of the memory required to process the data and the processing time.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space.**

Time Complexity

It's a function describing the amount of time required to run an algorithm in terms of the size of the input. "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

Space Complexity

It's a function describing the amount of memory an algorithm takes in terms of the size of input to the algorithm. We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.

Space complexity is sometimes ignored because the space used is minimal and/or obvious, however sometimes it becomes as important an issue as time.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by $\mathbf{T(n)}$, where **n** is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm. Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O**: Big Oh
- **Ω**: Big omega
- **ϴ**: Big theta
- **o**: Little Oh
- **ω**: Little omega

O: Asymptotic Upper Bound

'O' (Big Oh) is the most commonly used notation. A function $\mathbf{f(n)}$ can be represented is the order of $\boldsymbol{g(n)}$ that is $\boldsymbol{O(g(n))}$**,** if there exists a value of positive integer **n** as $\mathbf{n_0}$ and a positive constant **c** such that:

$$\boldsymbol{f(n)} \leq \boldsymbol{c.g(n)} \text{ for } \boldsymbol{n > n_0} \text{ in all case.}$$

Hence, function $\boldsymbol{g(n)}$ is an upper bound for function $\boldsymbol{f(n)}$**,** as $\boldsymbol{g(n)}$ grows faster than $\boldsymbol{f(n)}$.

Example

Let us consider a given function, $\boldsymbol{f(n) = 4.n^3 + 10.n^2 + 5.n + 1}$.

Considering $\boldsymbol{g(n) = n^3}$,

$$\boldsymbol{f(n) \leq 5.g(n)} \text{ for all the values of } \boldsymbol{n > 2}.$$

Hence, the complexity of $\boldsymbol{f(n)}$ can be represented as $\boldsymbol{O(g(n))}$**, i.e.** $\boldsymbol{O(n^3)}$.

Ω: Asymptotic Lower Bound

We say that $\boldsymbol{f(n) = \Omega(g(n))}$ when there exists constant **c** that $\boldsymbol{f(n) \geq c.g(n)}$ for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n***, ***f*** will never go below ***g***.

Example

Let us consider a given function, $\boldsymbol{f(n) = 4.n^3 + 10.n^2 + 5.n + 1}$.

Considering $\boldsymbol{g(n) = n^3}$, $\boldsymbol{f(n) \geq 4.g(n)}$ for all the values of $\boldsymbol{n > 0}$.

Hence, the complexity of $\boldsymbol{f(n)}$ can be represented as $\boldsymbol{\Omega(g(n))}$, i.e. $\boldsymbol{\Omega(n^3)}$.

θ: Asymptotic Tight Bound

We say that $f(n) = \theta(g(n))$ when there exist constants $\mathbf{c_1}$ and $\mathbf{c_2}$ that $c_1.g(n) \leq f(n) \leq c_2.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function ***g*** is a tight bound for function ***f***.

Example

Let us consider a given function, $\boldsymbol{f(n) = 4.n^3 + 10.n^2 + 5.n + 1}$.

Considering $\boldsymbol{g(n) = n^3}$, $\boldsymbol{4.g(n) \leq f(n) \leq 5.g(n)}$ for all the large values of **n**.

Hence, the complexity of $f(n)$ can be represented as $\Theta(g(n))$, i.e. $\Theta(n^3)$.

O - Notation

The asymptotic upper bound provided by **O-notation** may or may not be asymptotically tight. The bound $2.n^2 = O(n^2)$ is asymptotically tight, but the bound $2.n = O(n^2)$ is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define $o(g(n))$ (little-oh of g of n) as the set $f(n) = o(g(n))$ for any positive constant $c > 0$ and there exists a value $n_0 > 0$, such that $0 \leq f(n) \leq c.g(n)$.

Intuitively, in the **o-notation**, the function $f(n)$ becomes insignificant relative to $g(n)$ as **n** approaches infinity; that is,

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = 0$$

Example

Let us consider the same function, $f(n) = 4.n^3 + 10.n^2 + 5.n + 1$.

Considering $g(n) = n^4$,

$$\lim_{n\to\infty}\left(\frac{4\,n\text{^}3\ +\ 10\,n\text{^}2\ +\ 5\,n\ +\ 1}{n\text{^}4}\right) =\ 0$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$, i.e. $o(n^4)$.

ω – Notation

We use **ω-notation** to denote a lower bound that is not asymptotically tight. Formally, however, we define $\omega(g(n))$ (little-omega of g of n) as the set $f(n) = \omega(g(n))$ for any positive constant $c > 0$ and there exists a value $n_0 > 0$, such that $0 \leq c.g(n) < f(n)$.

For example, $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$. The relation $f(n) = \omega(g(n))$ implies that the following limit exists

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right) = \quad \infty$$

That is, $f(n)$ becomes arbitrarily large relative to $g(n)$ as **n** approaches infinity.

Example

Let us consider same function, $f(n) = 4.n^3 + 10.n^2 + 5.n + 1$.

Considering $g(n) = n^2$,

$$\lim_{n \to \infty} \left(\frac{4\,\mathrm{n}^3 + 10\,\mathrm{n}^2 + 5\,\mathrm{n} + 1}{n^2} \right) = \infty$$

Hence, the complexity of $f(n)$ can be represented as $o(g(n))$**, i.e.** $\omega(n^2)$.

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**