Algorithm Analysis

Algorithm Analysis

Objectives

- The purpose of algorithm analysis is, in general , to determine the performance of the algorithm in terms of time taken and storage requirements to solve a given problem.

- An other objective can be to check whether the the algorithm produces consistent, reliable, and accurate outputs for all instances of the problem .It may also ensured that algorithm is robust and would prove to be failsafe under all circumstances.

- The common metrics that are used to gauge the performance are referred to as *time efficiency*, *space efficiency*, and *correctness*.

Algorithm Analysis

Input Size

- The performance is often expressed in terms *problem size*, or more precisely, by the number of data item items processed by an algorithm.

- The key parameters used in the analysis of algorithms for some common application types are:

    i. Count of data items in data collections such as arrays, queues

    ii. Number of nodes in trees and linked lists

    iii. Number of vertices and number of edges in a graph

    iv. Number of rows and columns in an input table

    v. Character count in an input text block

Analysis of Algorithm

Time Efficiency

- The *time efficiency* determines how fast an algorithm solves a given problem. Quantitatively, it is the measures of time taken by the algorithm to produce the output with a given input. The time efficiency is denoted by a mathematical function of the input size.

- Assuming that an algorithm takes *T(n)* time to process *n* data items. The function *T(n)* is referred to as the *running time* of the algorithm. It is also known as *time complexity*. The time complexity can depend on more than one parameter. For example, the running time of a graph algorithm can depend both on the number of vertices and edges in the graph.

- The running time is an important indicator of the behavior of an algorithm for varying inputs. It tells, for example, whether the time taken to process would increase in direct proportional to input size, would increase four fold if size is doubled, or increase exponentially.

- It would be seen that that time efficiency is the most significant metric for algorithm analysis. For this reason, the main focus of algorithm analysis would be on determining the time complexity.

Time Efficiency

Approaches

The time efficiency can be determined in several ways . The common methods are categorized as *empirical*, *analytical*, and *visualization*

- In empirical approach , the running time is determined experimentally. The performance is measured by testing the algorithm with inputs of different sizes

- The analytical method uses *mathematical* and *statistical* techniques to examine the time efficiency. The running time is expressed as mathematical function of input size

- The visualization technique is sometimes used to study the behavior and performance of an algorithm by generating graphics and animation, based on interactive inputs .

Empirical Analysis

Methodology

The empirical methodology broadly consists of following steps

1) The algorithm is coded and run on a computer. The running times are measured, by using some timer routine , with inputs of different sizes..

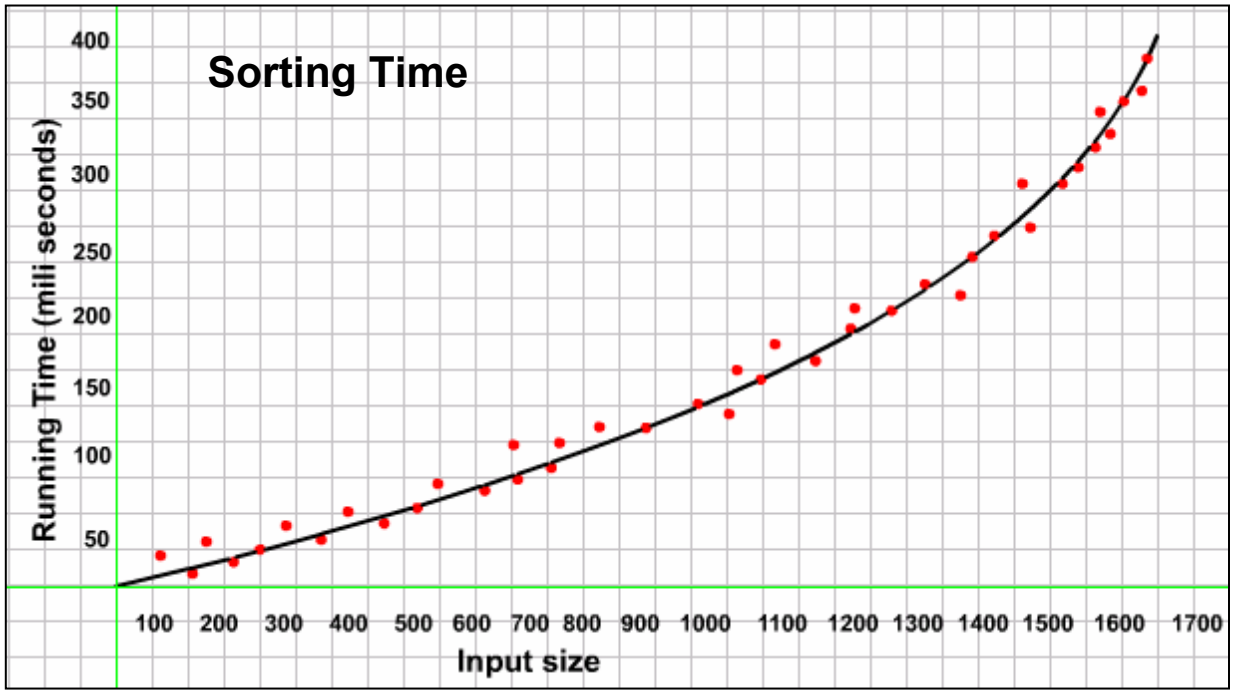
2) The output is logged and analyzed with the help of *graphical* and *statistical* tools to determine the behavior and growth rate of running time in terms of input size.

3) A *best curve* is fitted to depict trend of the algorithm in terms input sizes

Empirical Analysis

Example

• The graph illustrates the results of an *empirical analysis*. The running times of a sorting algorithm are plotted against the number input size. The measured values are shown in red dots. The graph shows the *best-fit curve* to scattered points.



• The analysis indicates that time increases roughly in proportion to the square of input size, which means that doubling the input size increases the running time four fold.

Empirical Analysis

Limitations

The empirical analysis provides estimates of running times in a real life situation. It has, however, several limitations, because the running time crucially depends on several factors. Some key factors that influence the time measurements are:

- Hardware types *(CPU speed, IO throughput, RAM size etc.)*
- Software environment *(Compiler, Programming Language etc.)*
- Program design *(Conventional, Structured, Object Oriented)*
- Composition of data set sets (*Choice of data values and the range of input)*

Analytical Analysis

Methodology

In analytic approach the running time is estimated by studying and analyzing the *basic* or *primitive operations* involved in an algorithm. Broadly, the following methodology is adopted:

- The code for the algorithm is examined to identify basic operations
- The number of times each basic operation is executed, for a given input, is determined.
- The running time is estimated by taking into consideration the frequency and cost of significant operations
- The total time is expressed as a function of the input size

Analytical Analysis

Basic Operations

• The code for an algorithm, generally, consists of a mix of following basic operations:.

i. *Assigning value to a variable*

ii. Comparing a pair of data items

iii. Incrementing a variable

iv. Performing arithmetic and floating point operations

v. Moving a data item from one storage location to another location

vi. Calling a procedure

vii. Returning a value

viii. Accessing an array element

• The time taken by the basic operation to complete a single step is often referred to as the *cost* of the operation. Some operations are relatively more expensive than others. For example, operations involving data movement and arithmetical computations are costly compared to logical or assignment operations .

Running Time Classification

Worst, Best, Average Cases

- We have seen that the running time of an algorithm depends on the *input size*. For some applications, the running time also depends on the *order* in which the data items are input to the algorithm.

- Let k denote *all possible orderings of input of size n*, and $T_1(n), T_2(n), .. T_k(n)$ be the running times for each instance **.** We can formally, define the ***best, worst*** and ***average*** running times, $T_{best}(n)$, $T_{worst}(n)$, $T_{average}(n)$, as follows

Best Case: In this case the algorithm has *minimum* running time.

: $T_{best}(n) = minimum(T_1, T_2, \ldots T_k)$

This is also called the *optimistic* time

Worst Case*:* In this case the algorithm has *maximum* running time

$T_{worst}(n) = maximum(T_1, T_2, \ldots T_k)$

This is also known as *pessimistic* time

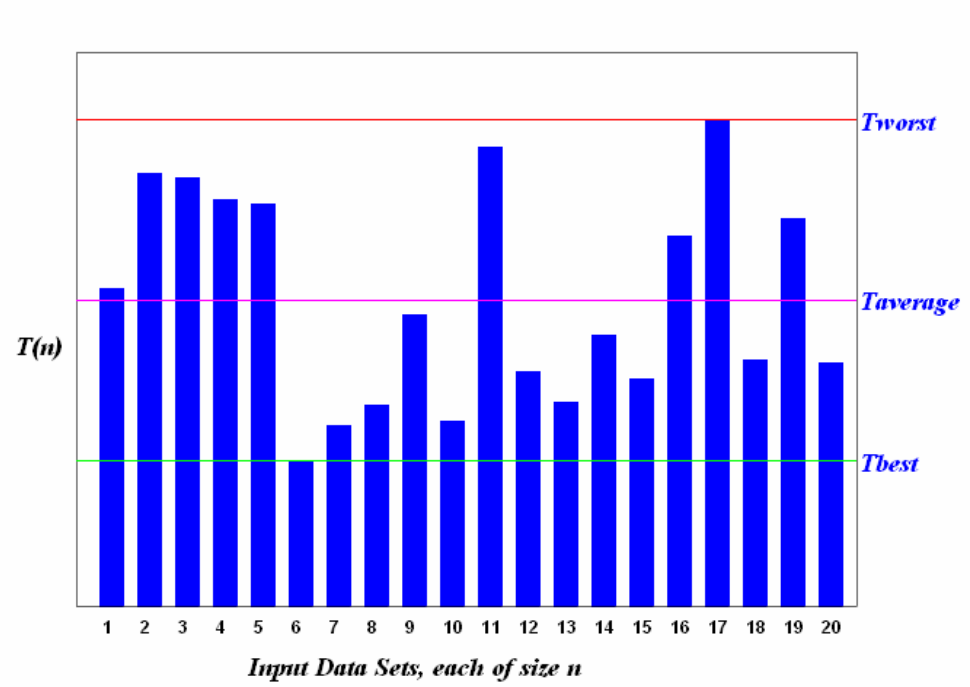
Average Case: The *average* running time is the *average of running times for all possible ordering of inputs of the same size:*

$T_{average}(n) = (T_1 + T_2 + \ldots + T_k) / k$

Best, Worst, Average Cases

Example

• The graph shows *best*, *worst*, and *average* running times of an algorithm. In this example, times are shown for *20* inputs of *same size* but *in different order*. The algorithm takes *minimum time* to process *Input* #6 , and *maximum time* to process *Input* #17 , which are referred to as *best* and *worst* times. The *average* of all the running times for *20* inputs is also shown.



• For an accurate analysis , *all possible arrangements* of input should be considered to identify the best, worst, and average running times

Algorithm Analysis

Space Efficiency

• The space efficiency determines total memory requirement of RAM and disk storage to run an algorithm with given input.

• The space requirement consists of the amount of real storage needed to execute the algorithm code and the storage to hold the application data. The algorithm code occupies a fixed amount of space, which is independent of the input size. The storage requirement for the application depends on the nature of data structure used to provide faster and flexible access to stored information . For an arrays and linked lists, for example, the space requirement is directly proportional to the input size.

• For most algorithms, the space efficiency is not of much concern. However, some algorithms require extra storage to hold results of intermediate computations. This is the case, for example, with merge sort and dynamic programming techniques.

Algorithm Correctness

Loop Invariant Method

- There are no standard methods for proving the correctness of a given algorithm. However, many useful algorithms consist of one or more *iterative computations*, using loop structures. The correctness of such algorithms can be formally established by a technique called *Loop Invariant Method* .

- A *loop invariant* is set of *conditions* and *relationships* that remain true *prior to*, *during* , and *after* the execution of a loop.

- The loop invariant condition / statement depends on the nature of problem being analyzed In a *sorting problem*, for example, the condition might be the order of keys in a *sub-array*, which should remain in ascending/descending order ,prior to, and after the execution of each iteration

Algorithm Correctness

Formal Proof

The loop invariant method establishes the correctness of algorithm in three steps, which are known as *initialization, maintenance*, and *termination* *(* Reference*: T. Cormen et al)* At each step, the *loop invariant* is examined. If the loop conditions at each of the steps hold true, then algorithm is said be *correct*.

Initialization: Loop invariant is true prior to execution of *first iteration* of the loop

Maintenance: If loop invariant is assumed to be true at *some iteration***,** it remains true *after the next iteration*

Termination: After the termination of the loop, the invariant holds true for the problem size

Algorithm Analysis

Visualization

Algorithm visualization techniques are used to study

- *Studying inner working of an algorithm through trace of basic operations*
- *Illustrating algorithm steps with animations*
- *Studying algorithm performance with interactive inputs*
- *Counting primitive operations for analysis*
- *Doing simulations with a variety of data sets*
- *Reporting on the performance*