# Algorithm Analysis

## Algorithm Definition:

An algorithm is a step-by-step method of solving some problem. An algorithm has a finite set of instruction having the following characteristics:

- **Precision:** The steps are precisely stated.
- **Uniqueness**: The intermediary results of each step of execution are uniquely defined and depend only on the inputs and the results of the preceding steps
- **Finiteness**: The algorithm stops after a finitely many instructions have been executed.
- **Input**: The algorithm receives input
- **Output**: The algorithm produces output
- **Generality** : The algorithm applies to a set of inputs

Recursion:

An algorithm is recursive if it calls itself to do part of its work. For this to be successful, the call to itself must be on a smaller problem than the original one.
A recursive algorithm has two parts 1- the base case, which handles a simple input that can be solved without using recursion and the recursive part which contains one or more recursive calls to the algorithm where the parameters are in some sense closer to the base case.
Summations and recurrences:

Most programs contain loop constructs. When analyzing time cost for programs with loops, we need to add up the cost for each time the loop is executed. This is an example of summation.
The following is a list of useful summations, along with their closed-form solutions:

$$\sum_{i=}^{n} i \; : \frac{n(n+\;)}{2} \qquad (1)$$

$$\sum_{i=}^{n} i \; = \frac{2n^3 + \; n^2 + n}{6} \qquad (2)$$

$$\sum_{i=}^{n} i \; : n \log n \qquad (3)$$

$$\sum_{i=} a^i = \frac{1}{1-a} \; for \; 0 < a < \qquad (4)$$

$$\sum_{i=}^{n} \frac{i}{2i} = \; : - \frac{n+}{2^n} \qquad (5)$$

$$\sum_{i=}^{n} a^i = \frac{a^{n+} \; -}{a-} \; for \; a \neq \qquad (6)$$

Estimating

Estimating can be formalized by the following three-step process:

1. Determine the major parameters that affect the problem
2. Derive an equation that relates the parameters to the problem
3. Select values for the parameters and apply the equation to yield an estimated solution.

Introduction to Algorithm Analysis:

Algorithm analysis measures the efficiency of an algorithm or its implementation as a program, as the input size becomes large. Typically, we will analyze the tine required for an algorithm and the space required a data structure,

Of primary consideration when estimating an algorithm performance is the number of basic operations required by the algorithm to process an input of a certain size.

Size often refers to the number of inputs being processed.

A basic operation must have the property that its time to complete does not depends on the particular value of its operands. (adding, comparing two integers versus summing an array of n integers)

Consider a simple algorithm to solve the problem of finding the largest value in an array of n integers. The algorithm looks at each integer in turn, saving the position of the largest value seen so far. This algorithm is called the largest-value sequential search :

```
int largest(int array[], int n) {
int currlarge =0;
for (int i =1; i <n; i++)
        if (array[currlarge] <array[i]
                currlarge=I;
return currlarge; }
```

Here the size of the problem is n; the number of integers stored in array. The basic operation is to compare an integer's value to that of the largest seen so far. It is assumed that it takes a fixed amount of time to do one such comparison, regardless of the value of the two integers or their position in the array. Since the most important factor affecting the running time is the size of the input, for a given input size n we say that the time T to run the algorithm is a function of n or T(n).

Let us call *c* the time required to compare two integers. We are not concerned with the time required to increment variable *i*, or the time of actual assignments.
Therefore, a good approximation of the total time required to run `largest` is *cn*, since we must make n comparisons, with each one costing *c* time.
We say the function larges running time is expressed by the equation: T(n) = cn
The equation describes the growth rate for the running time of the largest value sequential search algorithm

The growth rate for an algorithm is the rate at which the cost of the algorithm grows as the size of the input grows.
A growth rate of cn is referred to as linear growth rate or running time.

An algorithm whose running-time equation has a highest order term containing a factor of $n^2$ is said to be quadratic.

If the term if the equation is $2^n$, we say that the algorithm has an exponential growth rate.

Best, Worst and Average Cases

**Best-Case Time:** The minimum time needed to execute the algorithm for a series of input all of size n.

**Worst-Case Time:** The maximum time needed to execute the algorithm for a series of input all of size n.

**Average-Case Time:** The average time needed to execute the algorithm for a series of input all of size n.

Often, we are more interested in how the best-case or the worst-case grows as the time grows.

For example, suppose that the worst case time of an algorithm is:
$T(n) = 60\ n^2 + 5\ n + 1$
For input of size n.
For large n, the term $60\ n^2$ is approximately equal to t(n). In this sense, t(n) grows like $60\ n^2$.

| n | $T(n) = 60\ n^2 + 5\ n+1$ | $60\ n^2$ |
|---|---|---|
| 10 | 6051 | 6000 |
| 100 | 600,501 | 600, 000 |
| 1000 | 60, 005,001 | 60,000,000 |
| 10,000 | 6, 000,050,001 | 6,000,000,000 |

When, we describe how the best case time and the worst case time grows as the size of the input increases, we only look for the dominant term and ignore the others, in other words, we can say that t(n) grows like $n^2$ as n increases.

We say in this case that t(n) is of order $n^2$ and we write:

$t(n) = \Theta(n^2)$ (read as t(n) is theta of $n^2$ )

Let f and g be functions with domain {1, 2, 3,…….}

## Big Oh Notation:

We write:

$f(n) = O(g(n))$

And we say that f(n) is of order at most g(n) if there exists a positive constant C1 such that

$|f(n)| \leq C1\,|g(n)|$

for all but a finitely many positive integers n.

## Omega Notation:

We write:

$f(n) = \Omega(g(n))$

And we say that f(n) is of order at least g(n) if there exists a positive constant C2 such that

$|f(n)| \geq C2\,|g(n)|$

for all but a finitely many positive integers n.

## Theta Notation:

We write:

$f(n) = \Theta(g(n))$

And we say that f(n) is of order g(n) if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

## Finding the Theta function of a function:

In order to find the Theta function of a function f(n), we need to find two constants C1 and C2 so that

$|f(n)| \leq C1\,|g(n)|$ *and* $|f(n)| \geq C2\,|g(n)|$

*with* $g(n) = n^2$

Example:

## Finding C1

$60\ n^2 + 5\ n + 1 \preceq 60\ n^2 + 5\ n^2 + n^2 = 66\ n^2$ for $n \geq 1$

So we can take C1 = 66

$60\ n^2 + 5\ n + 1 = O(n^2)$

## Finding C2

$60\ n^2 + 5\ n + 1 \geq 60\ n^2$

So we can take C2 = 60

$60\ n^2 + 5\ n + 1 = \Omega(n^2)$

Since $60\ n^2 + 5\ n + 1 = O(n^2)$ and $60\ n^2 + 5\ n + 1 = \Omega(n^2)$ we say that

$60\ n^2 + 5\ n + 1 = \Theta(n^2)$

This method can be used to show that a polynomial in n of degree k with non negative coefficient is $\Theta(n^k)$.

Classes of Problems:

- A problem that can be solved with polynomial worst-case complexity is called **tractable**.
- Problems of higher complexity are called **intractable.**
- Problems that no algorithm can solve are called **unsolvable**.

**Additional Resources:**

**http://www.cc.gatech.edu/~bleahy/cs1311/cs1311lecture23wdl.ppt.**