Analyzing algorithms

- To measure performance of algorithms, we typically use time and space complexity analysis.
- The idea is to measure order of growths in terms of input size.

  > Independent of the machine and its configuration, on which the algorithm is running on.

  >Shows a direct correlation with the number of inputs.

  >Can distinguish two algorithms clearly without ambiguity.

Time Complexity and Space Complexity

Time Complexity:

- The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

Time Complexity

- The valid algorithm takes a finite amount of time for execution.
- The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm.
- It is the time needed for the completion of an algorithm.
- To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Example

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <- A + B
return C
```

Example

- The addition of two scalar numbers requires one addition operation.
- the time complexity of this algorithm is constant, so $T(n) = O(1)$.

Space Complexity:

- Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution.
- The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
- The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.

Space Complexity:

- To estimate the memory requirement we need to focus on two parts:
- **(1) A fixed part:** It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.
- **(2) A variable part:** It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.

Example : Addition of two scalar variables

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <– A+B
return C
```

Example : Addition of two scalar variables

- The addition of two scalar numbers requires one extra memory location to hold the result.
- Thus the space complexity of this algorithm is constant, hence S(n) = O(1).

Asymptotic Notations

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- These notations provide a concise way to express the behaviour of an algorithm's time or space complexity as the input size approaches infinity.
- Rather than comparing algorithms directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.

Asymptotic Notations

- It enables comparisons of algorithms' efficiency by abstracting away machine-specific constants and implementation details, focusing instead on fundamental trends.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

Asymptotic Notations

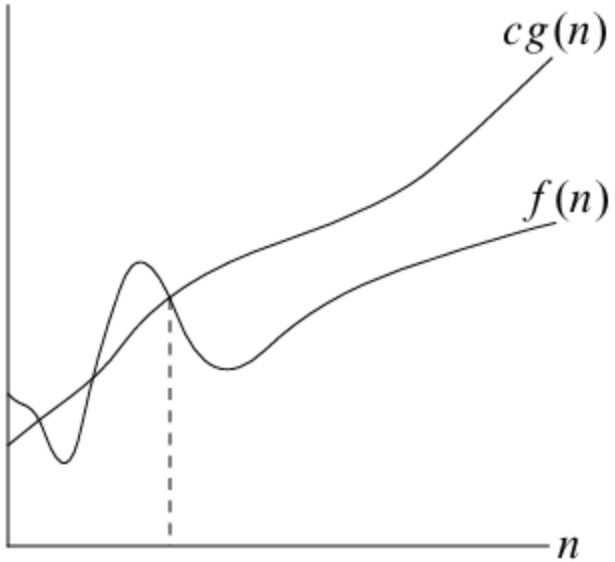
There are mainly three asymptotic notations:

1. *Big-O Notation (O-notation)*
2. *Omega Notation (Ω-notation)*
3. *Theta Notation (Θ-notation)*

O-notation

The Θ-notation asymptotically bounds a function from above and below. When we have only an ***asymptotic upper bound***, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-oh of g of n" or sometimes just "oh of g of n") the set of functions

$$O(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\} .$$



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

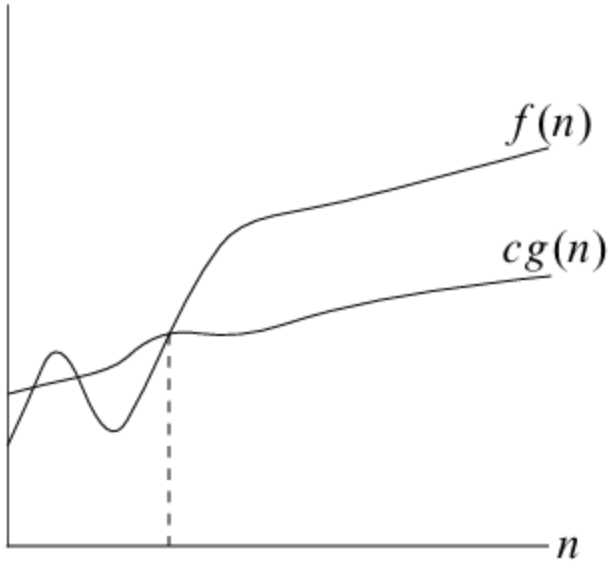
O-notation

- Worst case time complexity
- Maximum time

Ω-notation

Just as O-notation provides an asymptotic *upper* bound on a function, Ω-notation provides an ***asymptotic lower bound***. For a given function $g(n)$, we denote by $\Omega(g(n))$ (pronounced "big-omega of g of n" or sometimes just "omega of g of n") the set of functions

$$\Omega(g(n)) = \{f(n) : \text{ there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



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

Ω-notation

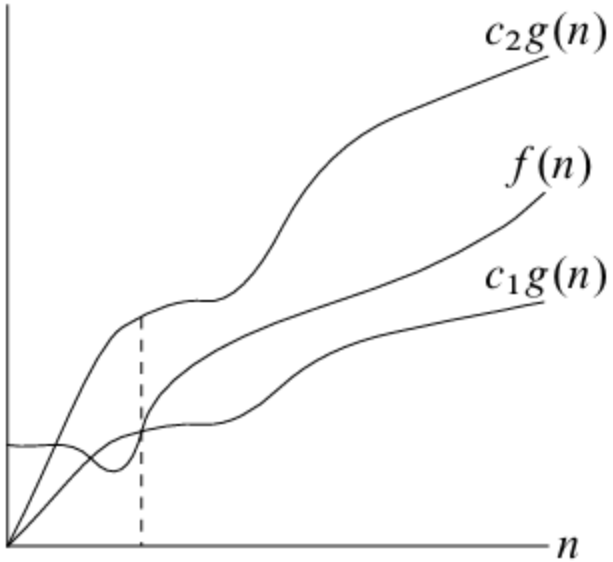
- Best case time complexity
- Minimum time

Θ-Notation

For a given function g(n)

we denote by $\Theta(g(n))$ the *set of functions*

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



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

Θ-Notation

- We say that g(n) is an asymptotically tight bound for f(n).
- Average case time complexity
- Exact time

o notation

$o(g(n)) = \{f(n) :$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

o notation

- If f(n)= o(g(n)), then f(n)=O(g(n)).
- If f(n)=O(g(n)), then it may or may not be true that f(n)=o(g(n)).
- Example:

  f(n)=n and g(n)=n

  For c=1, f(n)=O(g(n))

  For c=1, f(n)≠o(g(n))

ω-notation

By analogy, ω-notation is to Ω-notation as o-notation is to O-notation. We use ω-notation to denote a lower bound that is not asymptotically tight. One way to define it is by

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$.

Formally, however, we define $\omega(g(n))$ ("little-omega of g of n") as the set

$$\omega(g(n)) = \{f(n) : \text{ for any positive constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\} .$$

Constant Time Complexity O(1):

- The time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain a loop, recursion, and call to any other non-constant time function.
- i.e. set of non-recursive and non-loop statements
- In computer science, O(1) refers to constant time complexity, which means that the running time of an algorithm remains constant and does not depend on the size of the input.
- This means that the execution time of an O(1) algorithm will always take the same amount of time regardless of the input size.
- An example of an O(1) algorithm is accessing an element in an array using an index.

Constant Time Complexity O(1):

- A loop or recursion that runs a constant number of times is also considered O(1).
- For example, the following loop is O(1)

```
for (int i = 1; i <= c; i++)
{
 // some O(1) expressions
}
```

Linear Time Complexity O(n):

- The Time Complexity of a loop is considered as O(n) if the loop variables are incremented/decremented by a constant amount.
- Linear time complexity, denoted as O(n), is a measure of the growth of the running time of an algorithm proportional to the size of the input.
- In an O(n) algorithm, the running time increases linearly with the size of the input.
- For example, searching for an element in an unsorted array or iterating through an array and performing a constant amount of work for each element would be O(n) operations.
- In simple words, for an input of size n, the algorithm takes n steps to complete the operation.

Linear Time Complexity O(n):

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i = i + c) {
    // some O(1) expressions
   }

for (int i = n; i > 0; i = i - c) {
    // some O(1) expressions
   }
```

Quadratic Time Complexity $O(n^c)$:

- The time complexity is defined as an algorithm whose performance is directly proportional to the squared size of the input data, as in nested loops it is equal to the number of times the innermost statement is executed.
- Quadratic time complexity, denoted as O(n^2), refers to an algorithm whose running time increases proportional to the square of the size of the input.
- In other words, for an input of size n, the algorithm takes n * n steps to complete the operation
- For example, the following sample loops have $O(n^2)$ time complexity

Quadratic Time Complexity $O(n^c)$:

```
for (int i = 1; i <= n; i += c) {
    for (int j = 1; j <= n; j += c) {
        // some O(1) expressions
    }
}
```

Quadratic Time Complexity $O(n^c)$:

```
for (int i = n; i > 0; i -= c) {
    for (int j = i + 1; j <= n; j += c) {
        // some O(1) expressions
    }
}
```

Quadratic Time Complexity $O(n^c)$:

```
for (int i = n; i > 0; i -= c) {
    for (int j = i - 1; j > 0; j -= c) {
        // some O(1) expressions
    }
}
```

Logarithmic Time complexity: O(log n)

- The time Complexity of a loop is considered as O(Logn) if the loop variables are divided/multiplied by a constant amount.
- And also for recursive calls in the recursive function, the Time Complexity is considered as O(Logn).

Logarithmic Time complexity: O(log n)

```
for (int i = 1; i <= n; i *= c) {
    // some O(1) expressions
}

for (int i = n; i > 0; i /= c) {
    // some O(1) expressions
}
```

Logarithmic Time complexity: O(log n)

```
void recurse(int n)
{
    if (n <= 0)
        return;
    else {
        // some O(1) expressions
    }
    recurse(n/c);
  // Here c is positive integer constant greater than 1
}
```