Algorithm Analysis (also called Asymptotic Analysis. Big-Oh Notation)

What is an algorithm?
• An algorithm is a sequence of instructions or a set of rules that are followed to accomplish a task.

e.g. making coffee requires a series of step (algorithm)

(https://en.wikipedia.org/wiki/Algorithm)

Asymptotic analysis
• Asymptotic analysis attempts to estimate the resource consumption of an algorithm.

• This will allow you to compare the merits of two alternative approaches to a problem you need to solve, and also to determine whether a proposed solution will meet required resource constraints before you invest money and time coding. Analysis is done before coding.

• How do you compare two algorithms for solving some problem in terms of efficiency?
  o We could implement both algorithms as computer programs and then run them on a suitable range of inputs, measuring how much of the resources in question each program uses. This approach is often unsatisfactory.
  o The absolute running time of an algorithm cannot be predicted, since this depends on the programming language used to implement the algorithm, the computer the program runs on, other programs running at the same time, the quality of the operating system, and many other factors. We need a machine-independent notion of an algorithm’s running time. The critical resource for a program is most often its running time. Asymptotic analysis measures the efficiency of an algorithm, or its implementation as a program, as the input size becomes large.

• Measure the amount of resources needed
  o time
  o space

• The current state-of-the-art in analysis is finding a measure of an algorithm’s relative running time, as a function of how many items there are in the input, i.e., the number of symbols required to reasonably encode the input, which we call n. We count the number of abstract operations as a function of n (size).
**The Constant Function**

Let’s take a constant function: \( f(n) = c; \) for some fixed constant \( c. \)

In other words, it does not matter what the value of \( n \) is - \( f(n) \) will always be equal to the constant value \( c. \)

As simple as it is, the constant function is useful in algorithm analysis because it characterizes the number of steps needed to do a basic operation on a computer, like

- adding two numbers,
- assigning a value to a variable, or
- comparing two numbers.

Time, \( T(n) = 1 \)

**Time Complexity: \( O(1) \)**

**The Logarithm Function**

\( f(n) = \log_b n, \) for some constant \( b > 1 \)

This function is defined as the inverse of a power, as follows:

\[ x = \log_b n \text{ if and only if } b^x = n. \] //b is the base of the algorithm

The most common base for the logarithm function in computer science is 2 as computers store integers in binary. That is, for us, \( \log n = \log_2 n. \)

Computing the logarithm function exactly for any integer \( n \) involves the use of calculus, but we can use an approximation that is good enough for our purposes without calculus.

For a positive integer, \( n, \) we repeatedly divide \( n \) by \( b \) and stop when we get a number less than or equal to 1. The number of divisions performed is equal to \( \lceil \log_b n \rceil. \) We give below examples of the computation of \( \lceil \log_b n \rceil \) by repeated divisions:

\[ \lceil \log_4 64 \rceil = 3, \text{ because } ((64/4)/4)/4 = 1; \]
\[ \lceil \log_2 12 \rceil = 4, \text{ because } (((12/2)/2)/2)/2 = 0.75 \leq 1. \]

**Time Complexity: \( \log n \)**
The Linear Function

\( f(n) = n \).

That is, given an input value \( n \), the linear function \( f \) assigns the value \( n \) itself.

This function arises in algorithm analysis any time we have to do a single basic operation for each of \( n \) elements.

e.g. when \( n \) doubles, so does the running time.

**Example 1: Printing each element of an array**

```java
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

Here, \( n = a.length \)

- 1 initialization of \( I \) (constant)
- \( n \) comparisons of \( i \) against \( a.length \)
- \( n \) increments of \( i \)
- \( n \) array indexing operations (to compute \( a[i] \))
- \( n \) invocations of \( System.out.println \)

\( T(n) = 4n+1 \).

All operations are not created equal.

But overall, \( T(n) = n \).

**Example 2**

Imagine a case with doubly-nested loops where only the outer loop is dependent on the problem size \( n \), and the inner loop always executes a constant number of times, say 3 times:

```java
for (int i = 0; i < n; i++) {
    for (int j = 0; j < 3; j++) {
        // these statements are executed \( O(n) \) times
    }
}
```

**Time Complexity: \( n \)**
The N-Log-N Function
\[ f(n) = n \log n \]
the function that assigns to an input \( n \) the value of \( n \) times the logarithm base-two of \( n \).

This function grows a little more rapidly than the linear function and a lot less rapidly than the quadratic function; therefore, we would greatly prefer an algorithm with a running time that is proportional to \( n \log n \), than one with quadratic running time this arises when algorithms solve a problem by breaking it up to smaller problems, solving them independently and then combining the solutions.

For example, the fastest possible algorithms for sorting \( n \) arbitrary values require time proportional to \( n \log n \).

Time Complexity: \( n \log n \)

The Quadratic Function
\[ f(n) = n^2 \]
given an input value \( n \), the function \( f \) assigns the product of \( n \) with itself (in other words, “\( n \) squared”).

Why the quadratic function appears in the analysis of algorithms?
many algorithms that have nested loops, where the inner loop performs a linear number of operations and the outer loop is performed a linear number of times. Thus, in such cases, the algorithm performs \( n \cdot n = n^2 \) operations.

nested loops where the first iteration of a loop uses one operation, the second uses two operations, the third uses three operations, and so on.

Practical when \( n \) is small.

Example 1
If you have nested loops, and the outer loop iterates \( i \) times and the inner loop iterates \( j \) times, the statements inside the inner loop will be executed a total of \( i \times j \) times. This is because the inner loop will iterate \( j \) times for each of the \( i \) iterations of the outer loop. This means that if both the outer and inner loop are dependent on the problem size \( n \), the statements in the inner loop will be executed \( O(n^2) \) times:

for ( int i = 0; i < n; i++ )
{
    for ( int j = 0; j < n; j++ )
    {
        //these statements are executed \( O(n^2) \) times
    }
}
Example 2

```java
for ( int i = 0; i < n / 2; i++ ) {
    for ( int j = 0; j < n / 3; j++ ) {
        // these statements are also executed O(n^2) times since both loops loop O(n) times, and O(n) * O(n) = O(n^2)
    }
}
```

**Time Complexity:** \( n^2 \)

*The Cubic Function*

\[ f(n) = n^3 \]

which assigns to an input value \( n \) the product of \( n \) with itself three times.

The cubic function appears less frequently in the context of algorithm analysis than the constant, linear, and quadratic functions previously mentioned, but it does appear from time to time.

Similarly, if you have triply-nested loops, all of which are dependent on the problem size \( n \), the statements in the innermost loop will be executed \( O(n^3) \) times:

```java
for ( int i = 0; i < n; i++ ) {
    for ( int j = 0; j < n; j++ ) {
        for ( int k = 0; k < n; k++ ) {
            // these statements are executed \( O(n^3) \) times
        }
    }
}
```

**Time Complexity:** \( n^3 \)

*The Exponential Function*

\[ f(n) = b^n \]

where \( b \) is a positive constant, called the base, and the argument \( n \) is the exponent.

The “Traveling Salesman Problem” – this is classic problem in c.s, and its brute-force solution is exponential in the number of cities. It is this: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

**Time Complexity:** depends on the exact algorithm, but is approximately \( 2^n \)
If we have a loop that starts by performing one operation and then doubles the number of operations performed with each iteration, then the number of operations performed in the $n$th iteration is $2^n$.

**Time Complexity:** $2^n$

**Summary**

To sum up in order, each of the seven common functions used in algorithm analysis.

<table>
<thead>
<tr>
<th>constant</th>
<th>logarithm</th>
<th>linear</th>
<th>$n$-log-$n$</th>
<th>quadratic</th>
<th>cubic</th>
<th>exponential</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\log n$</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n^2$</td>
<td>$n^3$</td>
<td>$a^n$</td>
</tr>
</tbody>
</table>

Ideally, we would like our algorithms to run in linear or $n$-log-$n$ time.

---

*Disclaimer: Notes adapted from previous CS 231 lecture materials at Colby College.*