Basics

1 Definition

In computer science, an algorithm is a set of well-defined instructions or rules designed to perform a specific task or solve a particular problem. Think of it as a recipe in cooking: it provides step-by-step directions on how to accomplish something.

2 Proof

On every valid input, the algorithm produces the output satisfies the required input/output relationship. However, this requires proof.

2.1 Proofs by Counterexample

Proof by counterexample is a method of disproving a statement or proposition in mathematics. The approach is straightforward: if you want to show that a general statement (like "all swans are white") is false, you only need to find a single instance or case (a counterexample) where the statement does not hold true (like finding a black swan).

2.2 Proofs by Contradiction

In this approach, you start by assuming the opposite of what you want to prove. Then, through logical reasoning, you demonstrate that this assumption leads to a contradiction, a situation where two opposing statements are both true, or where the assumption leads to an absurd or impossible conclusion.

2.3 Inductive Proofs

Inductive proofs, commonly used in mathematics, are a method of proving statements or theorems that are generally formulated as "for all integers n", where n is a natural number. This method is particularly effective for proving statements about sequences or series, mathematical properties, and patterns that continue indefinitely. The process involves two main steps:

1. **Base Case:** First, you establish the truth of the statement for the initial value of n, usually $n = 1$. This step is crucial as it serves as the foundation for the inductive process.

2. **Inductive Step:** Next, you assume that the statement is true for some arbitrary natural number $n = k$ (this is called the **inductive hypothesis**), and then you use this assumption to prove that the statement must also be true for the next number in the sequence, $n = k + 1$

2.4 If and Only If Proofs

An "if and only if" proof in mathematics is used to establish a biconditional statement, where **two conditions are both necessary and sufficient** for each other. In formal terms, a statement "A if and only if B" means that A is true exactly when B is true, and vice versa. Such a statement is often abbreviated as "**A iff B**".

To prove this statement, two things need to be proved:

1. **If A, then B**
2. **If B, then A**

3 Running Time

Simple operation only takes 1 step, such as plus, minus, equal, if and so on. However, **loop and subroutine calls** are not simple operations. They depend upon the size of the data and the contents. The **running time** is defined as the number of steps taken by the algorithm, given input size. **The time complexity** of an algorithm $T(n)$ associates **the maximum (worst-case)** amount of time taken on input of size n, where T is a function mapping non-negative integers (problem sizes) to real numbers (number of steps).

3.1 Asymptotic Order of Growth

1. **Upper bounds (worst-case)**: the maximum number of steps taken on any instance of size n. In math notation, $T(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have:

$$T(n) \leq c \cdot g(n) \tag{1}$$

   Or we can say T grows slower or equal to g.

2. **Lower bounds (best-case)**: the minimum number of steps taken on any instance of size n. In math notation, $T(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have:

$$T(n) \geq c \cdot g(n) \tag{2}$$

   Or we can say T grows faster or equal to g.

3. **Tight bounds (average-case)**: the average number of steps taken on any instance of size n. In math notation, $T(n)$ is $\Theta(g(n))$ if $T(n)$ is both $O(g(n))$ and $\Omega(g(n))$. Or we can say T and g have the same rates.

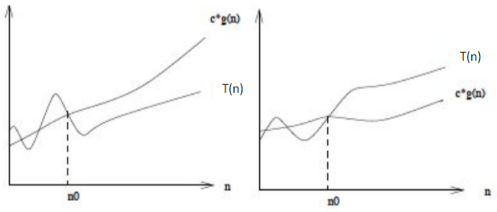

Figure 1: Upper case and lower case

3.2 Properties

3.2.1 Transitivity

1. If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
2. If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$
3. If $f \in \Theta(g)$ and $g \in \Theta(h)$ then $f \in \Theta(h)$

3.2.2 Additivity

1. If $f \in O(h)$ and $g \in O(h)$ then $f + g \in O(h)$
2. If $f \in \Omega(h)$ and $g \in \Omega(h)$ then $f + g \in \Omega(h)$
3. If $f \in \Theta(h)$ and $g \in \Theta(h)$ then $f + g \in \Theta(h)$

4 Polynomial vs Logarithmic vs Exponential

4.1 Polynomial

The polynomial time is defined as $O(n^d)$ for some constant d independent of the input size n. For a polynomial model:

$$a_0 + a_1 n + ... + a_d n^d \tag{3}$$

The time complexity will be $\Theta(n^d)$ if $a_d > 0$.

4.2 Logarithmic

Logarithmic time complexity, denoted as $O(\log n)$, is a term used in computer science to describe an algorithm that has its running time increase logarithmically in relation to the size of the input data set. Notice that the base does not matter because:

$$log_b n = \frac{log_a n}{log_a b} \tag{4}$$

Therefore for any constants $a, b > 0$:

$$O(log_a n) = O(log_b n) \tag{5}$$

4.3 Exponential

Exponential time complexity, usually defined as $O(b^n)$, where b is a constant base greater than 1, refers to an algorithm whose running time will be multiplied by b with each additional element in the input data set. In other words, the growth of the runtime is proportional to a fixed power of the size of the input.

4.4 Interrelations

1. Logarithms grow **slower** than every polynomial for every $d > 0$, no matter how small d is. In other words:

$$\log n \in O(n^d) \tag{6}$$

   This means logarithmic time complexity is also a polynomial time complexity.

2. Exponentials grow **faster** than every polynomial for every $b > 1$ no matter how close to 1, and every $d > 0$ no matter how big. In other words:

$$n^d \in O(b^n) \tag{7}$$

3. Stirling's Approximation:

$$ln(n!) = nln(n) - n + O(ln(n)) \tag{8}$$

$$n! \approx \sqrt{(}2\pi n)(\frac{n}{e})^n \tag{9}$$

$$\log_2(n!) = n\log_2 n - n\log_2 e + O(\log_2 n) \tag{10}$$