CHAPTER

1

INTRODUCTION

The objective of this chapter is to explain the importance of the analysis of algorithms, their notations, relationships and solving as many problems as possible. Let us first focus on understanding the basic elements of algorithms, the importance of algorithm analysis, and then slowly move toward the other topics as mentioned above. After completing this chapter, you should be able to find the complexity of any given algorithm (especially recursive functions).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of this equation. The important thing that we need to understand is that the equation has names (*x* and *y*), which hold values (data). That means the *names* (*x* and *y*) are placeholders for representing data. Similarly, in computer science programming we need something for holding data, and *variables* is the way to do that.

1.2 Data Types

In the above-mentioned equation, the variables *x* and y can take any values such as integral numbers (10, 20), real numbers (0.23, 5.5), or just 0 and 1. To solve the equation, we need to relate them to the kind of values they can take, and *data type* is the name used in computer science programming for this purpose. A *data type* in a programming language is a set of data with predefined values. Examples of data types are: integer, floating point, unit number, character, string, etc.

Computer memory is all filled with zeros and ones. If we have a problem and we want to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers provide us with data types. For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes, etc. This says that in memory we are combining 2 bytes (16 bits) and calling it an *integer*. Similarly, combining 4 bytes (32 bits) and calling it a *float*. A data type reduces the coding effort. At the top level, there are two types of data types:

- System-defined data types (also called *Primitive* data types)
- User-defined data types

System-defined data types (Primitive data types)

Data types that are defined by system are called *primitive* data types. The primitive data types provided by many programming languages are: int, float, char, double, bool, etc. The number of bits allocated for each primitive data type depends on the programming languages, the compiler and the operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types, the total available values (domain) will also change.

For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 32,767 (-2^{15} *to* $2^{15}-1$). If it takes 4 bytes (32 bits), then the possible values are between -2,147,483,648 and +2,147,483,647 (-2^{31} *to* $2^{31}-1$). The same is the case with other data types.

User defined data types

If the system-defined data types are not enough, then most programming languages allow the users

to define their own data types, called *user – defined data types*. Good examples of user defined data types are: structures in *C/C + +* and classes in *Java*. For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the name *"newType"*. This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {
    int data1;
    float data 2;
    ...
    char data;
};
```

1.3 Data Structures

Based on the discussion above, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. A *data structure* is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures:* Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially. *Examples:* Linked Lists, Stacks and Queues.
2) *Non – linear data structures:* Elements of this data structure are stored/accessed in a non-linear order. *Examples:* Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general, user defined data types are defined along with their operations.

To simplify the process of solving problems, we combine the data structures with their operations and we call this *Abstract Data Types* (ADTs). An ADT consists of *two parts:*

1. Declaration of data

2. Declaration of operations

Commonly used ADTs *include:* Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack, etc.

While defining the ADTs do not worry about the implementation details. They come into the picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

1.5 What is an Algorithm?

Let us consider the problem of preparing an *omelette*. To prepare an omelette, we follow the steps given below:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            1. If yes, then go out and buy.
            2. If no, we can terminate.
3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

An algorithm is the step-by-step unambiguous instructions to solve a given problem.

In the traditional study of algorithms, there are two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how much resources (in terms of memory and time) does it take to execute the).

Note: We do not have to prove each step of the algorithm.

1.6 Why the Analysis of Algorithms?

To go from city *"A"* to city *"B"*, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

1.7 Goal of the Analysis of Algorithms

The goal of the *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

1.8 What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

1.9 How to Compare Algorithms

To compare algorithms, let us define a *few objective measures:*

Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

1.10 What is Rate of Growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us

assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

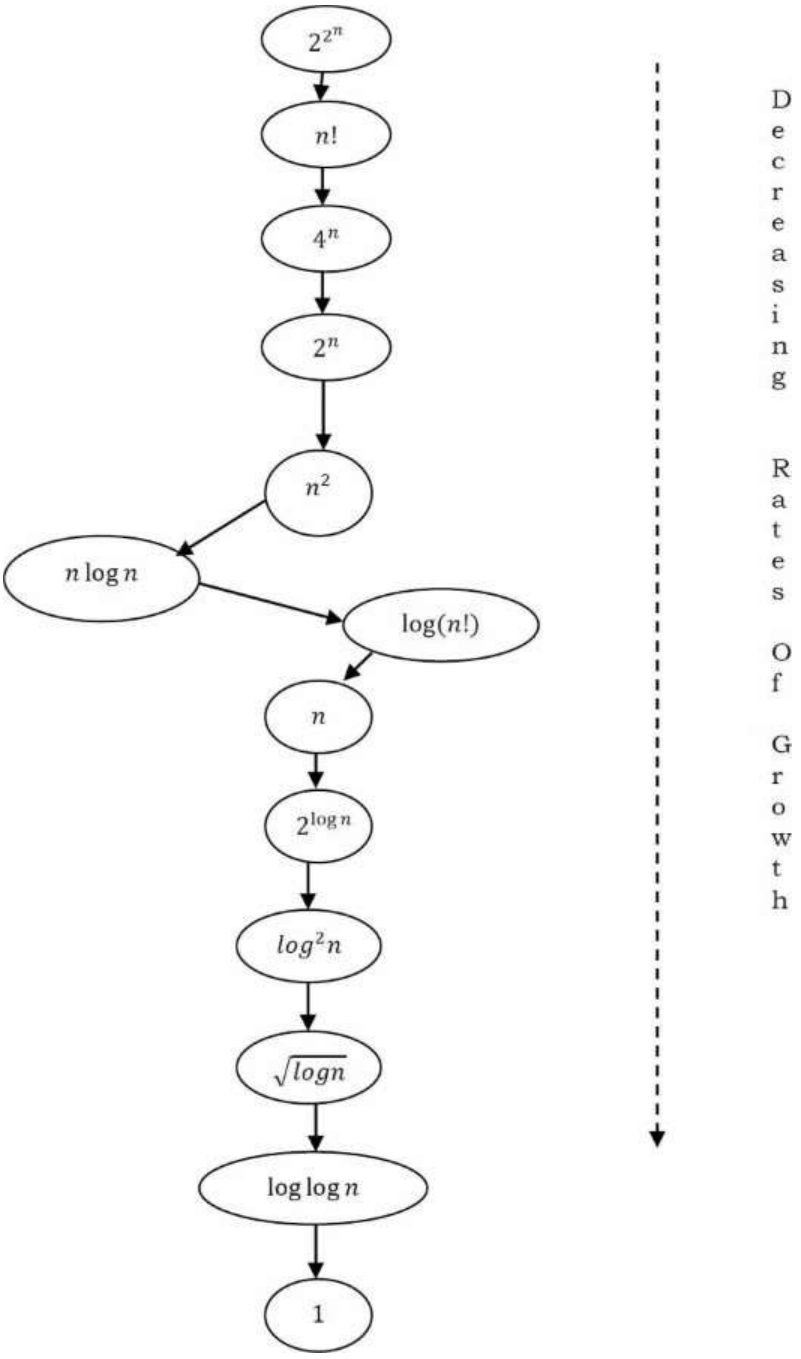
$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle in terms of function, and for a given function ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example, in the case below, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate to n^4 since n^4 is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 Commonly Used Rates of Growth

The diagram below shows the relationship between different rates of growth.



Below is the list of growth rates you will come across in the following chapters.

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

1.12 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing wel1) and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for the algorithm. To analyze an algorithm we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- **Worst case**
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - Assumes that the input is random.

Lower Bound <= Average Time <= Upper Bound

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$$f(n) = n^2 + 500, \text{ for worst case}$$
$$f(n) = n + 100n + 500, \text{ for best case}$$

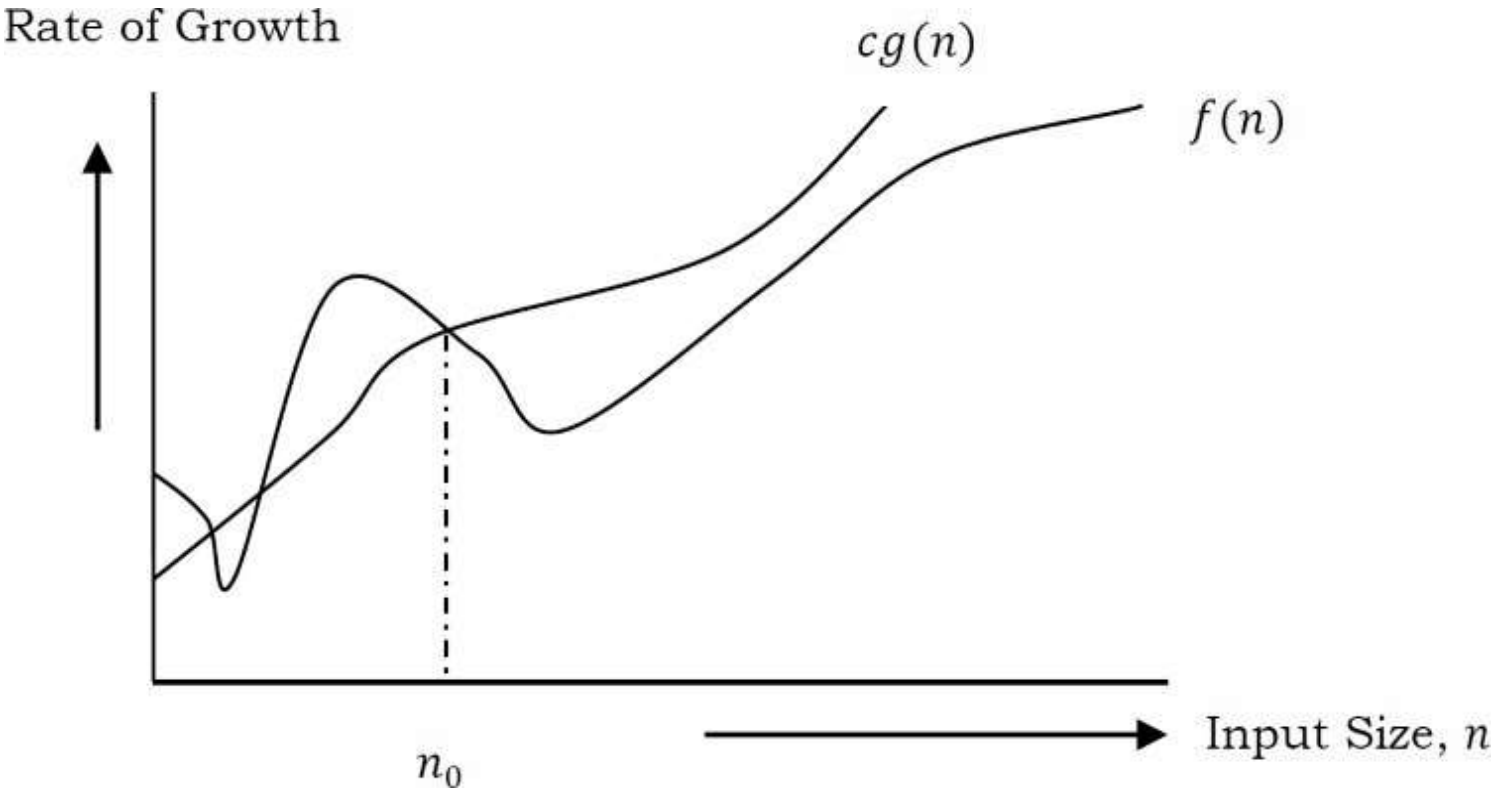
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

Having the expressions for the best, average and worst cases, for all three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function $f(n)$.

1.14 Big-O Notation [Upper Bounding Function]

This notation gives the *tight* upper bound of the given function. Generally, it is represented as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$. For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 *is* $g(n)$. That means $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.



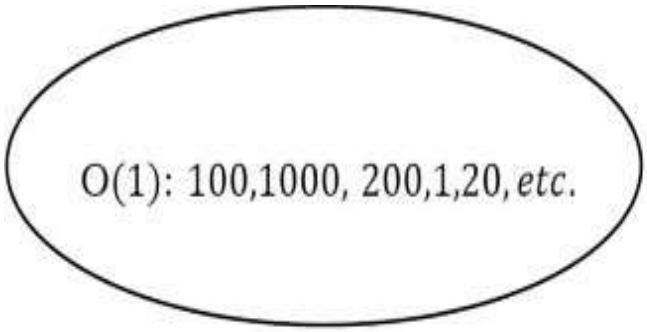
Let us see the O–notation with a little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n > n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth $g(n)$ which is greater than or equal to the given algorithms' rate of growth /(n).

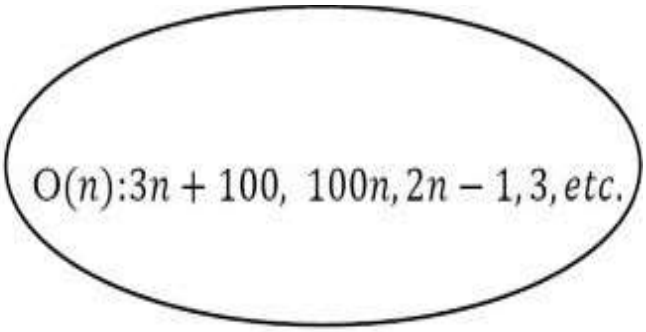
Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0, the rate of growth could be different. n_0 is called threshold for the given function.

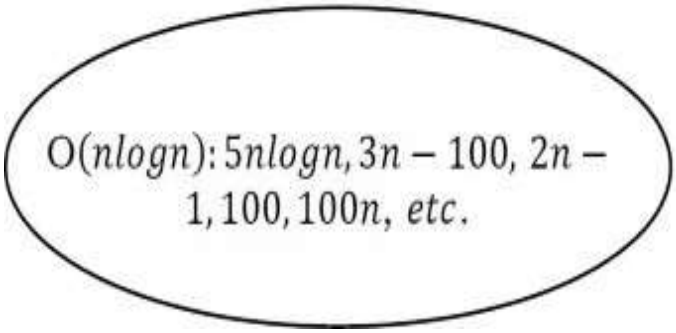
Big-O Visualization

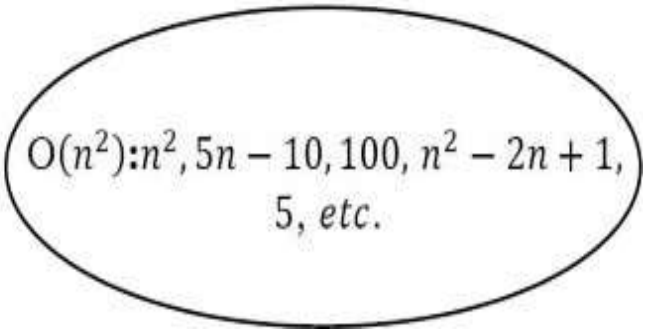
$O(g(n))$ is the set of functions with smaller or the same order of growth as $g(n)$. For example; $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$, etc.

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care about the rate of growth.









Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \le 4n$, for all $n \ge 8$

$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n > 1$
$\therefore 2n^3 - 2n^2 = O(n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \leq n$, for all $n \geq 1$
$\therefore n = O(n)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n > 1$
$\therefore 410 = O(1)$ with $c = 1$ and $n_0 = 1$

No Uniqueness?

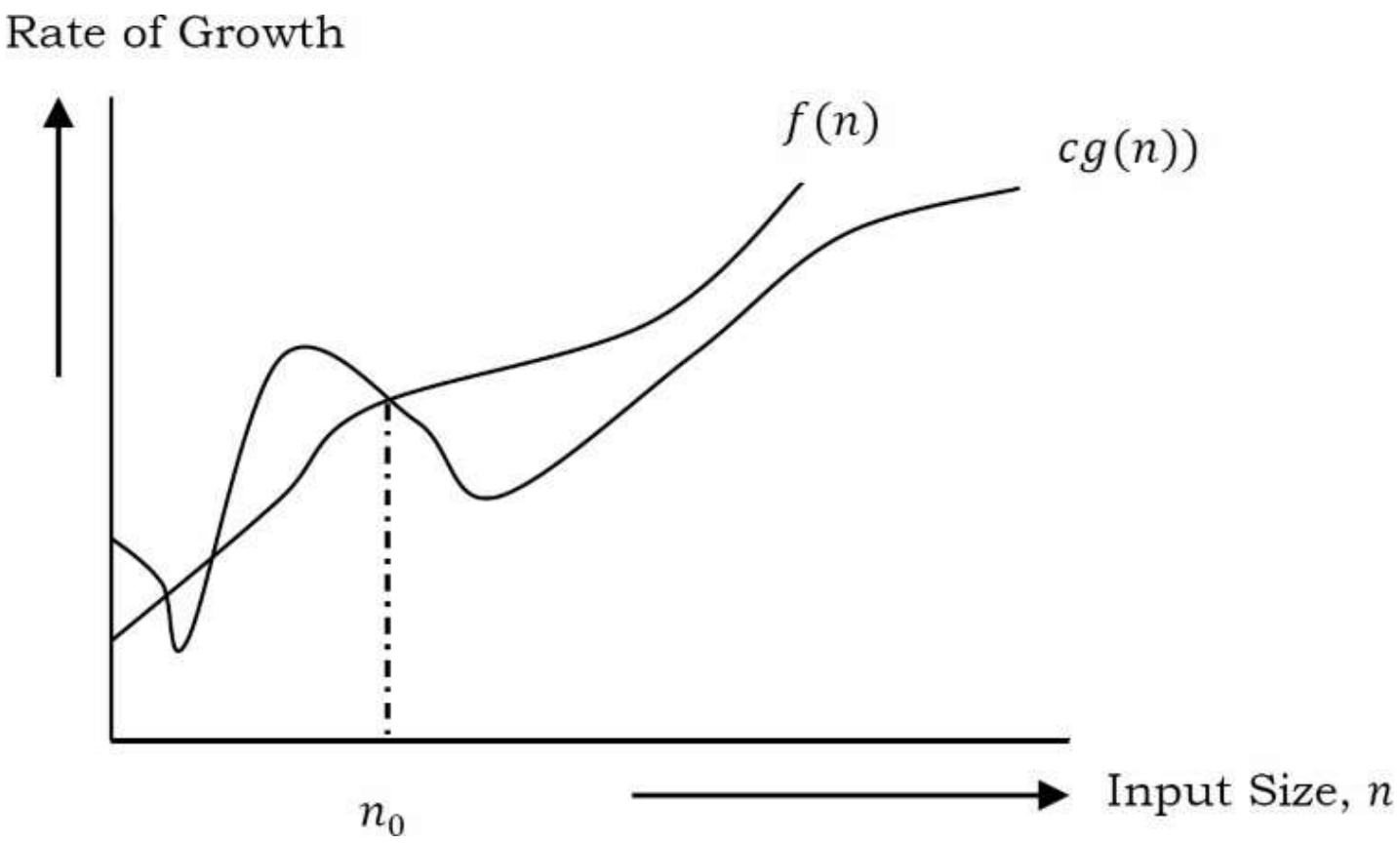
There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n$, for all $n > 1$, $n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Q Notation [Lower Bounding Function]

Similar to the O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



The Ω notation can be defined as Ω(g(*n*)) = {f(n): there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$}. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give the largest rate of growth $g(n)$ which is less than or equal to the given algorithm's rate of growth $f(n)$.

Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$.

Solution: ∃ c, n_0 Such that: $0 \le cn^2 \le 5n^2 \Rightarrow cn^2 \le 5n^2 \Rightarrow c = 5$ and $n_0 = 1$

$\therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$

Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$.

Solution: ∃ c, n_0 Such that: $0 \le cn^2 \le 100n + 5$

$100n + 5 \le 100n + 5n(\forall n \ge 1) = 105n$

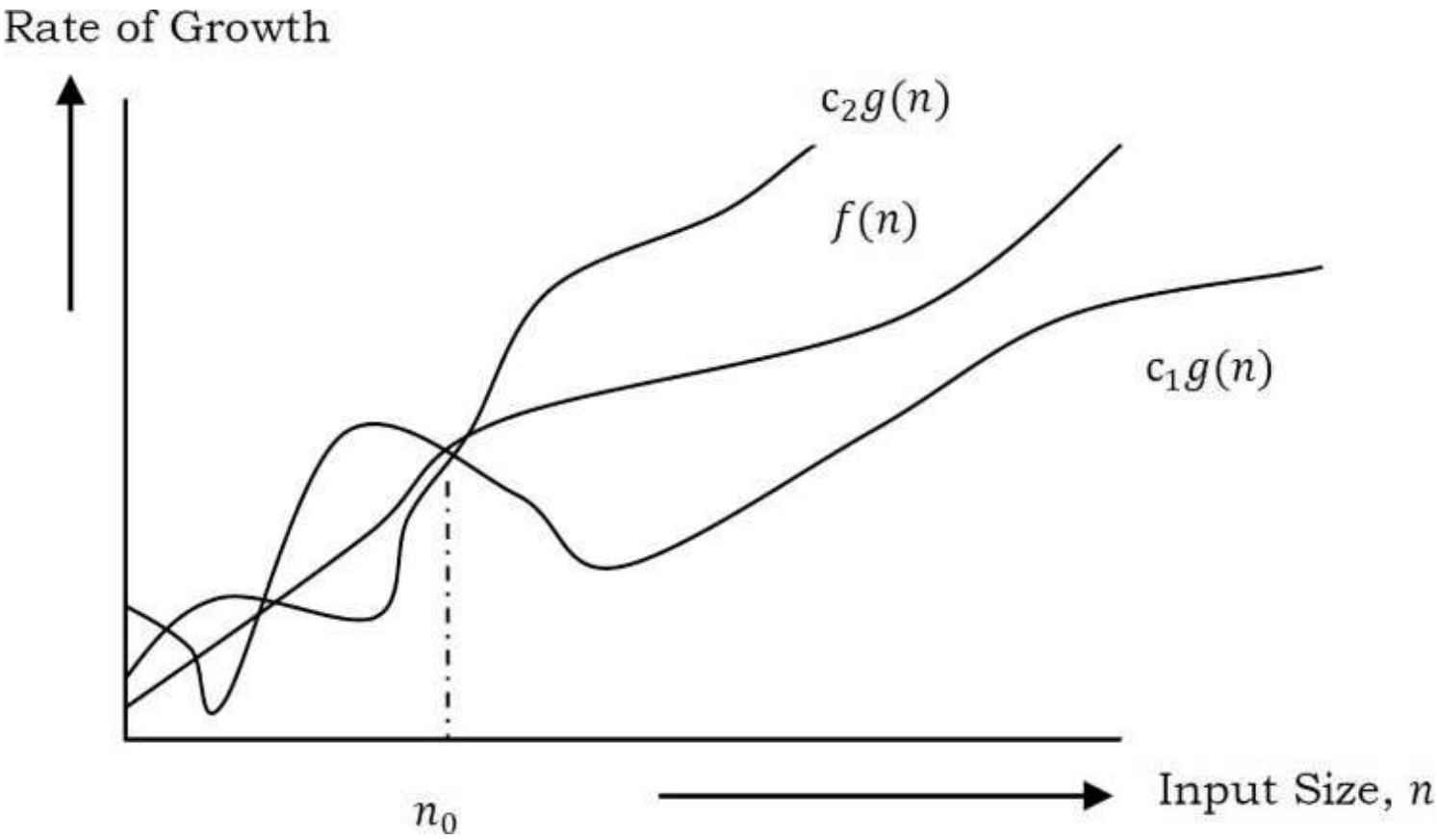
$cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

⇒ Contradiction: n cannot be smaller than a constant

Example-3 $2n = \Omega(n)$, $n^3 = \Omega(n^3)$, $= O(logn)$.

1.16 Theta-Θ Notation [Order Function]



This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth.

As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in the best case is $g(n) = O(n)$.

In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same. For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same. In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case, refer to the *Sorting* chapter).

Now consider the definition *of* Θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Θ Examples

Example 1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$ for all, $n \geq 2$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 2$

Example 2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example 3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example 4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \leq n \leq c_2 logn \Rightarrow c_2 \geq \frac{n}{\log n}$, $\forall n \geq n_0$ – Impossible

1.17 Important Notes

For analysis (best case, worst case and average), we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm), getting the upper bound (O) and lower bound (Ω) and average running time (Θ) may not always be possible. For example, if we are discussing the best case of an algorithm, we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ).

In the remaining chapters, we generally focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

1.18 Why is it called Asymptotic Analysis?

From the discussion above (for all three notations: worst case, best case, and average case), we can easily understand that, in every case for a given function $f(n)$ we are trying to find another function $g(n)$ which approximates $f(n)$ at higher values of n. That means $g(n)$ is also a curve which approximates $f(n)$ at higher values of n.

In mathematics we call such a curve an *asymptotic curve*. In other terms, $g(n)$ is the asymptotic

curve for $f(n)$. For this reason, we call algorithm analysis *asymptotic analysis*.

1.19 Guidelines for Asymptotic Analysis

There are some general rules to help us determine the running time of an algorithm.

1) **Loops:** The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total time = a constant $c \times n = c\,n = O(n)$.

2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executes n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executes n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
        k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
     // another if : constant + constant (no else part)
     if(!list[n].equals(otherList.list[n]))
         //constant
        return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is O(*logn*) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i=1; i<=n;)
   i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4,8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$, and at $(k + 1)^{th}$ step we come out of the *loop*. Taking logarithm on both sides, gives

$$log(2^k) = logn$$
$$klog2 = logn$$
$$k = logn \qquad //\text{if we assume base-2}$$

Total time = O(*logn*).

Note: Similarly, for the case below, the worst case rate of growth is O(*logn*). The same discussion holds good for the decreasing sequence as well.

```
for (i=n; i>=1;)
    i = i/2;
```

Another example: binary search (finding a word in a dictionary of n pages)

- Look at the center point in the dictionary
- Is the word towards the left or right of center?
- Repeat the process with the left or right part of the dictionary until the word is found.

1.20 Simplyfying properties of asymptotic notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n)), (g_1(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\ f_2(n)$ is in $O(g_1(n)\ g_1(n))$.

1.21 Commonly used Logarithms and Summations

Logarithms

$$\log x^y = y \log x \qquad \log n = \log_{10}^n$$
$$\log xy = \log x + \log y \qquad \log^k n = (\log n)^k$$
$$\log \log n = \log(\log n) \qquad \log \frac{x}{y} = \log x - \log y$$
$$a^{\log_b^x} = x^{\log_b^a} \qquad \log_b^x = \frac{\log_a^x}{\log_a^b}$$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \log n$$

Other important formulae

$$\sum_{k=1}^{n} \log k \approx n \log n$$
$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

1.22 Master Theorem for Divide and Conquer Recurrences

All divide and conquer algorithms (also discussed in detail in the *Divide and Conquer* chapter) divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, a merge sort algorithm [for details, refer to *Sorting* chapter] operates on two sub-problems, each of which is half the size of the original, and then performs O(*n*) additional work for merging. This gives the

running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it. If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1, b > 1, k \geq 0$ and p is a real number, then:

1) If a > b^k, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If a= b^k
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1.23 Divide and Conquer Master Theorem: Problems & Solutions

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$
Solution: $T(n) = 3T(n/2) + n^2 => T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$
Solution: $T(n) = 4T(n/2) + n^2 => T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$
Solution: $T(n) = T(n/2) + n^2 => \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$
Solution: $T(n) = 2^n T(n/2) + n^n =>$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$
Solution: $T(n) = 16T(n/4) + n => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + nlogn$

Solution: $T(n) = 2T(n/2) + nlogn => T(n) = \Theta(nlog^2n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$
Solution: $T(n) = 2T(n/2) + n/logn => T(n) = \Theta(nloglogn)$ (Master Theorem Case 2. b)

Problem-8 $T(n) = 2T(n/4) + n^{051}$
Solution: $T(n) = 2T(n/4) + n^{051} => T(n) = \Theta(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$
Solution: $T(n) = 0.5T(n/2) + 1/n$ => Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$
Solution: $T(n) = 6T(n/3) + n^2logn => T(n) = \Theta(n^2logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2logn$
Solution: $T(n) = 64T(n/8) - n^2logn$ => Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$
Solution: $T(n) = 7T(n/3) + n^2 => T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$
Solution: $T(n) = 4T(n/2) + logn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$
Solution: $T(n) = 16T(n/4) + n! => T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$
Solution: $T(n) = \sqrt{2}T(n/2) + logn => T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$
Solution: $T(n) = 3T(n/2) + n => T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$
Solution: $T(n) = 3T(n/3) + \sqrt{n} => T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$
Solution: $T(n) = 4T(n/2) + cn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$
Solution: $T(n) = 3T(n/4) + nlogn => T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$
Solution: $T(n) = 3T(n/3) + n/2 => T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

1.24 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \le 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b \ge 0, k \ge 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if } a < 1 \\ O(n^{k+1}), & \text{if } a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if } a > 1 \end{cases}$$

1.25 Variant of Subtraction and Conquer Master Theorem

The solution to the equation $T(n) = T(\alpha n) + T((1 - \alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

1.26 Method of Guessing and Confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then *prove* it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound $T(n) < cnlogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, c\, log\sqrt{n} + n \\ &= n.c.\frac{1}{2}.logn + n \\ &\leq cnlogn \end{aligned}$$

The last inequality assumes only that $1 \leq c.\frac{1}{2}.logn$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.\, k\, \sqrt{n}\, log\sqrt{n} + n \\ &= n.\, k\, log\sqrt{n} + n \\ &= n.k.\frac{1}{2}.logn + n \\ &\geq knlogn \end{aligned}$$

The last inequality assumes only that $1 \geq k.\frac{1}{2}.logn$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that $\Theta(nlogn)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \quad \geq n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n} + n \\ &= n.\, c + n \\ &= n\,(c + 1) \\ &\nleq cn \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(nlogn)$ is too big. So, we need something bigger than n and smaller than $nlogn$. How about $n\sqrt{logn}$?

Proving the upper bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ c.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Proving the lower bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\not\geq knlog\sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{logn})$ doesn't work. What else is between *n* and *nlogn?* How about *nloglogn?* Proving upper bound for *nloglogn:*

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn\text{-}c.n + n \\ &\leq cnloglogn, \text{ if } c \geq 1 \end{aligned}$$

Proving lower bound for *nloglogn:*

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn\text{-}k.n + n \\ &\geq knloglogn, \text{ if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \leq cnloglogn$, if $c \geq 1$ and $T(n) \geq knloglogn$, if $k \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(nloglogn)$.

1.27 Amortized Analysis