Asymptotic Analysis:

Accurate measurement of time complexity is possible with asymptotic notation. Asymptotic complexity gives an idea of how rapidly the space requirement or time requirement grow as problem size increases.

When there is a computing device that can execute 1000 complex operations per second. The size of the problem is that can be solved in a second or minute or an hour by algorithms of different asymptotic complexity.

In general, asymptotic complexity is a measure of algorithm not problem. Usually, the complexity of an algorithm is as a function relating the input length to the number of steps (time complexity) or storage location (space complexity). For example, the running time is expressed as a function of the input size 'n' as follows.

f(n) =n^4+1000n^2+10n+50(running time)

Limits and asymptotic notations

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$

For example, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C, C \geq 0$ implies f(n)= O(g(n)).

Description	Notation Used
c is positive or zero	O-notation
c is positive or ∞	Ω-notation
c is positive	Θ-notation
c is zero	o-Notation
c is ∞	ω-notation

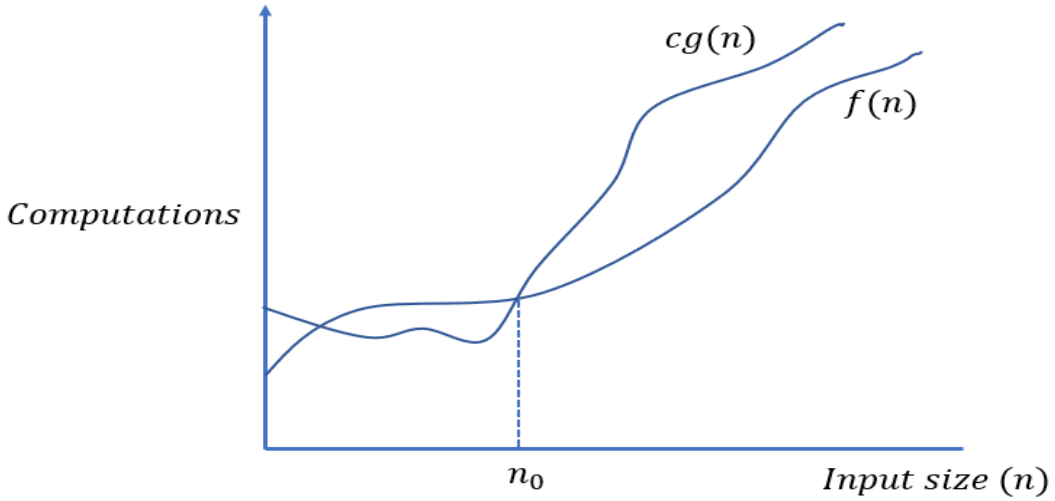
Big oh notation

Big oh notation is denoted by 'O' . It is sued to describe the efficiency of an algorithm. It is used to represent the upper bound of an algorithm running time. Using Big O notation, we can give largest amount of time taken by the algorithm to complete.

Definition: Let f(n) and g(n) be the two non -negative functions. We say that f(n) is said to be O(g(n)) if and only if there exists a positive constant 'c' and 'n_0' such that.

f(n)≤ c*g(n) ∀non-negative values of n, where n≥n_0 and $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C, \quad C \geq 0.$

Here g(n) is the upper bound for f(n).



Example:

Let $f(n)=2n^4+5n^2+2n+3$

$<= 2n^4+5n^4+2n^4+3n^4$

$<= (2+5+2+3)\ n^4$

$<=12n^4$

$f(n)=12n^4$

This implies $g(n)= n^4, n>=1$ ∴ $c=12$ and $n_0=1$

∴ $f(n)= O(n^4)$

The above function states that the function 'f' is almost 'c' times the function 'g' when 'n' is greater than or equal to n0. This notion provides an upper bound for the function 'f' i.e., the function g(n) is an upper bound on the value of f(n) for all n, where n≥n_0 .

Show that $4n^2 = O(n^3)$.

Solution By definition, we have

$0 \leq f(n) \leq cg(n)$

Substituting 4n2 as h(n) and n3 as g(n), we get

$0 \leq 4n^2 \leq cn^3$

Dividing by n^3

$0/n3 \leq 4n^2/n^3 \leq cn^3/n^3$

$0 \leq 4/n \leq c$

Now to determine the value of c, we see that 4/n is maximum when n=1.

Therefore, c=4.

To determine the value of n_0,

$0 \leq 4/n_0 \leq 4$

$0 \leq 4/4 \leq n0$

$0 \leq 1 \leq n0$

This means n0=1. Therefore, $0 \leq 4n^2 \leq 4n^3$, $\forall\ n \geq n0=1$.

Show that $400n3 + 20n2 = O(n^3)$.

Solution: By definition, we have

$0 \leq f(n) \leq cg(n)$

Substituting $400n^3 + 20n^2$ as f(n) and n^3 as g(n), we get

$0 \leq 400n^3 + 20n^2 \leq cn^3$

Dividing by n^3

$0/n^3 \leq 400n^3/n^3 + 20n^2/n^3 \leq cn^3/n^3$

$0 \leq 400 + 20/n \leq c$

Note that $20/n \rightarrow 0$ as $n \rightarrow \infty$, and 20/n is maximum when n = 1. Therefore,

$0 \leq 400 + 20/1 \leq c$

This means, c = 420

To determine the value of n_0,

$0 \leq 400 + 20/n0 \leq 420$

$-400 \leq 400 + 20/n0 - 400 \leq 420 - 400$

$-400 \leq 20/n_0 \leq 20$

$-20 \leq 1/n_0 \leq 1$

$-20\ n_0 \leq 1 \leq n_0$. This implies $n_0 = 1$.

Hence, $0 \leq 400n^3 + 20n^2 \leq 420n^3\ \forall\ n \geq n_0=1$.

Show that $10n^3 + 20^n \neq O(n^2)$.

Solution By definition, we have

$0 \leq f(n) \leq cg(n)$

Substituting 10n3 + 20n as h(n) and n2 as g(n), we get

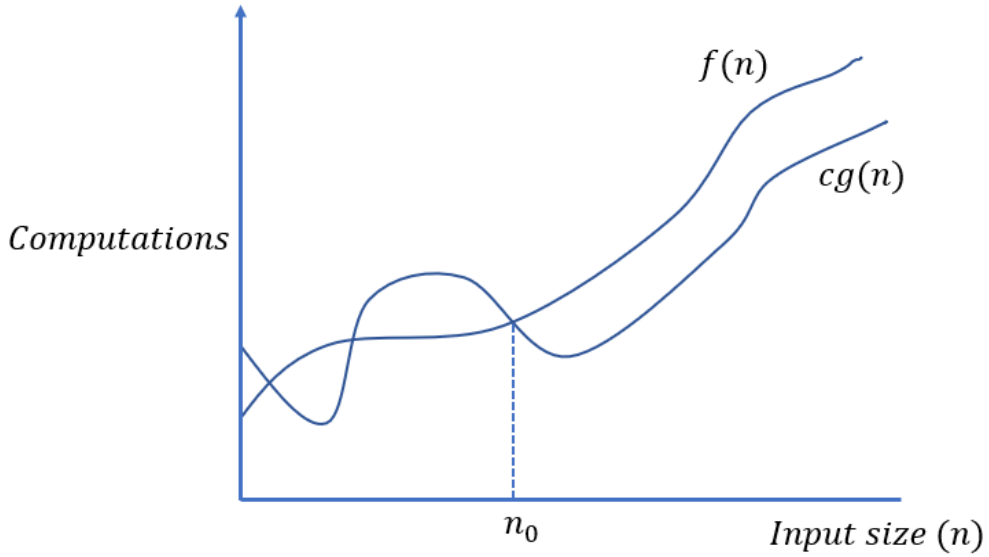
$0 \leq 10n^3 + 20n \leq cn^2$

Dividing by n^2
$0/n^2 \leq 10n^3/n^2 + 20n/n^2 \leq cn^2/n^2$
$0 \leq 10n + 20/n \leq c$
$0 \leq (10n^2 + 20)/n \leq c$
Hence, $10n^3 + 20n \neq O(n^2)$

Big Omega Notation: It is denoted by the symbol Ω. It is used to represent the lower bound of an algorithm running time. Using big omega notation, we can give shortest amount of time taken by the algorithm.

Definition: The function f(n) = Ω(g(n)) f(n) is omega of g(n) if and only if there exist positive constants 'c' and 'n0' such that

f(n)≥ c* g(n) for all n≥n0 and $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$.



Let f(n)= $2n^4+5n^2+2n+3$

≥$2n^4$(for example as $n \to \infty$ lower order terms are insignificant)

f(n)=$2n^4$, n≥1

g(n)=n^4, c=2 and n_0=1

f(n)= Ω(n^4)

- **Show that $5n^2 + 10n = \Omega(n^2)$.**

Solution By the definition, we can write

$0 \leq cg(n) \leq f(n)$
$0 \leq cn^2 \leq 5n^2 + 10n$

Dividing by n^2
$0/n^2 \leq cn^2/n^2 \leq 5n^2/n^2 + 10n/n^2$
$0 \leq c \leq 5 + 10/n$
Now, $\lim_{n\to\infty} \left(5 + \frac{10}{n}\right) = 5.$

Therefore, $0 \leq c \leq 5$.
Hence, $c = 5$
Now to determine the value of n_0
$0 \leq 5 \leq 5 + 10/n_0$
$-5 \leq 5 - 5 \leq 5 + 10/n_0 - 5$

$-5 \leq 0 \leq 10/n_0$
So $n_0 = 1$ as $\lim_{n\to\infty} \left(\frac{1}{n}\right) = 0.$

Hence, $5n^2 + 10n = \Omega(n^2)$ for $c=5$ and $\forall\ n \geq n_0=1$.

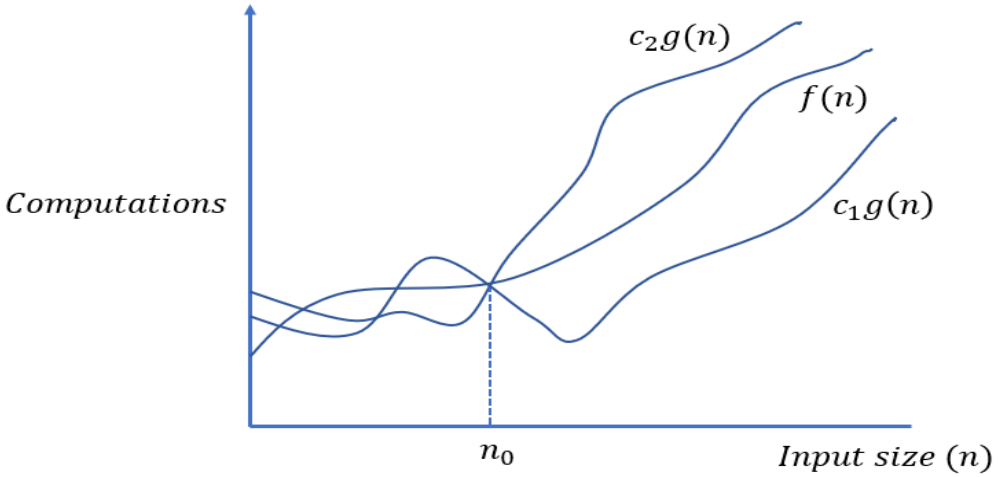
- **Show that $7n \neq \Omega(n^2)$.**

Solution By the definition, we can write
$0 \leq cg(n) \leq h(n)$
$0 \leq cn^2 \leq 7n$
Dividing by n^2, we get
$0/n^2 \leq cn^2/n^2 \leq 7n/n^2$
$0 \leq c \leq 7/n$
Thus, from the above statement, we see that the value of c depends on the value of n. There does not exist a value of n_0 that satisfies the condition as n increases. This could fairly be possible if $c = 0$ but it is not allowed as the definition by itself says that $\lim_{n\to\infty} \left(\frac{1}{n}\right) = 0.$

Big theta notation: The big theta notation is denoted by Θ. It is in between upper bound and lower bound of an algorithm running time.

Definition: Let f(n) and g(n) be the two non-negative functions. We say that f(n) is said to be Θ(g(n)) if and only if there exists a positive constants 'c1' and 'c2' such that.

c1*g(n) ≤f(n)≤c2g(n), for all non-negative values n where n≥n0 and $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C, C > 0.$



The above definition states that the function f(n) lies between 'c1' times the function g(n) and 'c2' times the function g(n) where 'c1' and 'c2' are positive constants. This notation provides both lower and upper bounds for the function f(n) i.e., g(n) is both lower and upper bounds on the value of f(n) for large n. In other words, Theta notation says that f(n) is both O(g(n)) and Ω(g(n)) for all n, where $n \geq n_0$.

$f(n)= 2n^4+5n^2+2n+3$

$\Longrightarrow 2n^4 <= 2n^4+5n^2+2n+3 <= 12n^4$

$\Longrightarrow 2n^4 <= f(n) <= 12n^4,\ n >= 1$

$\therefore g(n)=n^4$

$\therefore c1=2$,c2=12 and $n_0=1$

$\therefore f(n)= \Theta(n^4)$

Little Oh notation

Let f(n) and g(n) are the functions that map positive real numbers. We can say that the function f(n) is o(g(n)) if for any real positive constant c, there exists an integer constant n0 ≤ 1 such that f(n) > 0.

Mathematical Relation of Little o notation

Using mathematical relation, we can say that f(n) = o(g(n)) means,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

Example on little o asymptotic notation

If f(n) = n^2 and g(n) = n^3 then check whether f(n) = o(g(n)) or not.

$$\lim_{n\to\infty} \frac{n^2}{n^3}$$

$$= \lim_{n\to\infty} \frac{1}{n}$$

$$= \frac{1}{\infty}$$

$$= 0$$

The result is 0, and it satisfies the equation mentioned above. So, we can say that f(n) = o(g(n))

Little Omega Notation (ω): we use ω notation to denote a lower bound that is not asymptotically tight. Formally, however, we define $\omega(g(n))$ for any positive constant c>0 and there exists a value n0>0, such that $0 \leqslant c.g(n) < f(n)$.

For example, $\frac{n^2}{2} = \omega(n)$ *but* $\frac{n^2}{2} \neq \omega(n^2)$. The relation f(n)=ω(g(n)) implies that the following limit exists $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$. That is, f(n) becomes arbitrarily large relative to g(n) as n approaches infinity.

Example

Let us consider same function, f(n)=4. n^3+10.n^2+5.n+1

Considering g(n)=n^2

$$\lim_{n\to\infty} \frac{4n^3 + 10n^2 + 5n + 1}{n^2} = \infty.$$

Hence, the complexity of ***f(n)*** can be represented as ω (g(n)), i.e. $\omega(n^2)$.

Algorithm:

1. Set max_element to the first element of the array.

2. Iterate through the array:

   - If the current element is greater than max_element, update max_element.

3. Return max_element.

Now, let's analyze the time complexity of this algorithm in terms of the notations:

Big O Notation (O): The worst-case scenario occurs when the maximum element is at the end of the array or is the last element being checked.

In this case, the algorithm will have to iterate through all elements of the array once. Therefore, the time complexity is O(n), where **n** is the number of elements in the array.

Omega Notation (Ω): The best-case scenario occurs when the maximum element is at the beginning of the array. In this case, the algorithm only needs to check the first element and can return it immediately. Therefore, the time complexity is Ω (1)

Theta Notation (Θ): Since the algorithm's time complexity ranges from O(n) to Ω (1), we can conclude that its time complexity is Θ(n). This is because the algorithm's performance is bounded both from above and below by linear functions.

Little o Notation (o): The time complexity of the algorithm is not o (1) because it's not always constant time, especially when n grows large.

Little Omega Notation (ω): Similarly, the time complexity of the algorithm is not ω(n) because it's not always linear time, especially when the array is small.

So, for the FindMaximumElement algorithm:

- O(n)
- Ω (1)
- Θ (n)
- o (1) (False)

- ω(n) (False)

Apriori and Apostiari Analysis

Apriori analysis means, analysis is performed prior to running it on a specific system. This analysis is a stage where a function is defined using some theoretical model. Hence, we determine the time and space complexity of an algorithm by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.

Apostiari analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.

In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.
In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same.

Empirical Analysis

Empirical analysis is a crucial technique used to complement theoretical analysis by evaluating the actual performance of algorithms using real-world data and experimentation. It provides insights into how algorithms behave in practical scenarios, accounting for factors not always considered in theoretical models.

1. Validating Theoretical Analysis:

Theoretical analysis, using tools like Big O notation, predicts an algorithm's complexity (e.g., time and space complexity) based on its operations. Empirical analysis allows you to test these predictions with real-world data. By running the algorithm on various data sets of different sizes and types, you can see if the predicted complexity holds true in practice.

2. Comparing Algorithms:

Often, multiple algorithms can solve the same problem with different theoretical complexities. Empirical analysis lets you compare the actual performance of these algorithms by running them on the same data sets and measuring their execution times and resource usage (e.g., memory). This helps you choose the most efficient algorithm for your specific needs and data characteristics.

3. Identifying Practical Limitations:

Theoretical analysis typically focuses on idealized scenarios and may not capture all real-world factors that affect performance.
Empirical analysis helps you uncover these limitations. It can reveal how factors like:

- Hardware architecture: Different CPU architectures or cache sizes can impact performance.
- Coding implementation: Different coding styles or libraries might have varying efficiency.
- Data distribution: The algorithm's performance might depend on the specific distribution of the input data (e.g., sorted vs. random).

While theoretical analysis provides a foundational understanding of algorithm complexity, empirical analysis is instrumental in understanding the practical behavior of algorithms and making informed decisions when selecting and designing algorithms for real-world applications.

Probabilistic Analysis

Probabilistic analysis is a technique used to analyze the performance of algorithms by considering the probability distribution of the input data. Unlike traditional worst-case or best-case analysis, it provides a more realistic evaluation by accounting for the randomness or uncertainty often present in real-world data. This approach helps us understand the average-case performance, which is often more relevant than extreme scenarios.

Example: Linear Search

Consider the problem of searching for an element in an unsorted array using linear search. In the worst case, the algorithm needs to compare the element with every item in the array, resulting in a complexity of O(n). However, this only occurs when the element is at the end of the array.

Probabilistic analysis allows us to consider the probability of finding the element at each position. If we assume all positions are equally likely, the average number of comparisons becomes:

Average comparisons = (1/n) + (2/n) + ... + (n/n)
= (n + n-1 + ... + 1) / n
= n (n + 1) / 2n
= (n + 1) / 2

This translates to an average complexity of O(n), even though the worst-case remains O(n). This analysis provides a more accurate picture of the algorithm's performance in most practical scenarios, where elements are not necessarily placed in the worst possible order.

Amortized Analysis

Amortized analysis is a technique used to analyze the performance of algorithms that exhibit fluctuations in their running time for individual operations. It helps us understand the average cost of an operation over a sequence of operations, even if some operations are more expensive than others.

Here's an illustration using a resizing array:

Scenario: Imagine a data structure that initially holds 4 elements and allows adding new elements. When the array reaches its capacity, it needs to be resized to double its size to accommodate more elements.

Analysis:

Adding the first 4 elements: Each addition takes constant time (O(1)) as there's enough space. Adding the 5th element: This triggers a resize operation, which is expensive and takes O(n) time (copying existing elements to the new array). Looking at individual operations, the first 4 additions seem "cheap" at O(1), while the 5th operation is "expensive" at O(n). However, what about the average cost per operation?

Amortized analysis helps us analyze the entire sequence of operations and "spread" the cost of the expensive resize operation over all additions. Average cost per addition: We consider the total time taken for all 5 additions (constant time for the first 4 + O(n) for the resize) and divide it by the number of additions (5).

This gives us an average cost of: $(4 * O(1) + O(n)) / 5 \approx O(1) + O(n/5)$

Key points:

The expensive resize operation is "amortized" over all additions, meaning its cost is spread out. The average cost per addition becomes a constant (O(1)) with an additional term that decreases with the number of elements (O(n/5)), becoming less significant as the array grows.

This demonstrates how amortized analysis provides a more accurate picture of the performance of algorithms with fluctuating costs. It shows that even though some operations may be expensive, their impact on the average cost per operation might be smaller than it initially seems. Here are some other examples where amortized analysis is commonly used:
Dynamic stacks: Pushing and popping elements usually takes constant time, but occasionally a resize is needed, which is amortized over all operations.

Time and space trade-off. It refers to the relationship between how much memory (space) an algorithm or program uses and how long it takes to run (time). Often, these two factors work in opposition:

- **Less Space, More Time:** An algorithm can be designed to use minimal space, but this might mean it spends extra time searching through data or performing calculations.
- **More Space, Less Time:** Conversely, an algorithm can be designed to prioritize speed by using additional space to store pre-computed information or optimized data structures.

This trade-off is a constant consideration when designing algorithms and data structures. The ideal scenario is to find an efficient balance between space and time complexity.

Here are some real-world examples of this trade-off:

- **Compressed vs. Uncompressed Files:** Storing a file uncompressed takes up more space but allows for faster access. Compressing the file saves space but requires extra time to decompress it when needed.
- **Hash Tables vs. Linked Lists:** Hash tables offer very fast retrieval times by using more space to store additional data for quick lookups. Linked lists use less space but may require iterating through more elements to find the desired data.

Understanding this trade-off is crucial for programmers to make informed decisions when choosing algorithms and data structures for their specific tasks. They consider factors like:

- **Available Memory:** If memory is limited, a space-saving algorithm might be preferable, even if it takes a bit longer.
- **Speed Requirements:** If fast processing is critical, using more space for a faster algorithm might be worthwhile.

- **Data Size:** The size of the data being processed can also influence the choice. For massive datasets, space efficiency might become more important.

By carefully considering the time and space trade-off, programmers can create software that is both efficient and effective.

Analyzing Recursive Algorithms with Recurrence Relations

Recursive algorithms, where a function calls itself within its definition, are a powerful tool for solving problems. But analyzing their performance (time and space complexity) can be tricky. This is where recurrence relations come in.

What are Recurrence Relations?

A recurrence relation is an equation that defines the running time (or space complexity) of a recursive function in terms of its value on smaller inputs. It expresses how the complexity for a given input size n depends on the complexity for smaller sizes, typically n-1 or n/2.

Structure of a Recurrence Relation:

A recurrence relation typically consists of two parts:

1. **Base Case(s):** This specifies the complexity for the simplest input sizes. For example, the base case might state that the complexity for n = 1 is a constant value.
2. **Recursive Case:** This defines the complexity for larger inputs (n) in terms of the complexity for smaller inputs (n-k, where k is some constant).

Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.

2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

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

We have to show that it is asymptotically bound by O (log n).

Solution:

For T (n) = O (log n)
We have to show that for some constant c

T (n) ≤c log n.
Put this in given Recurrence Equation.
T (n) ≤c log($\frac{n}{2}$) + 1
≤c log+ 1 = c logn-c$\log_2$ 2+1
≤c logn for c≥1
Thus **T (n) =O logn**.

Example2:
Recurrence: T (1) = 1 and T(n) = 2T(n/2) + n for n > 1.

We guess that the solution is T(n) = O (n log n). So, we must prove that T(n) ≤ cn log n for some constant c. (We will get to n0 later, but for now let's try to prove the statement for all n ≥ 1.)

As our inductive hypothesis, we assume T(n) ≤ cn log n for all positive numbers less than n.

Therefore, T([n/2]) ≤ c[n/2] log(n/2), and

$$
\begin{aligned}
T(n) &\le 2(c[n/2]\log(n/2])) + n \\
&\le cn\log(n/2) + n \\
&= cn\log n - cn\log 2 + n \\
&= cn\log n - cn + n \\
&\le cn\log n \text{ (for } c \ge 1)
\end{aligned}
$$

Now we need to show the base case. This is tricky, because if T(n) ≤ cn log n, then T(1) ≤ 0, which is not a thing. So, we revise our induction so that we only prove the statement for n ≥ 2, and the base cases of the induction proof (which is not the same as the base case of the recurrence!) are n = 2 and n = 3. (We are allowed to do this because asymptotic notation only requires us to prove our statement for n ≥ n0, and we can set n0 = 2.)

We choose n = 2 and n = 3 for our base cases because when we expand the recurrence formula, we will always go through either n = 2 or n = 3 before we hit the case where n = 1.

So, proving the inductive step as above, plus proving the bound works for n = 2 and n = 3, suffices for our proof that the bound works for all n > 1. Plugging the numbers into the recurrence formula, we get T (2) = 2T(1) + 2 = 4 and T(3) = 2T(1) + 3 = 5. So now we just need to choose a c that satisfies those constraints on T (2) and T (3).

We can choose c = 2, because $4 \leq 2 \cdot 2 \log 2$ and $5 \leq 2 \cdot 3 \log 3$. Therefore, we have shown that $T(n) \leq 2n \log n$ for all $n \geq 2$, so $T(n) = O(n \log n)$.

Recursive Tree Method

A recursion tree is useful for visualizing what happens when a recurrence is iterated. It diagrams the tree of recursive calls and the amount of work done at each call.

Recursion trees can be useful for gaining intuition about the closed form of a recurrence, but they are not a proof (and in fact it is easy to get the wrong answer with a recursion tree, as is the case with any method that includes "..." kinds of reasoning). As we saw last time, a good way of establishing a closed form for a recurrence is to make an educated guess and then prove by induction that your guess is indeed a solution. Recurrence trees can be a good method of guessing.

Steps to Solve Recurrence Relations Using Recursion Tree Method-

Step-01:

Draw a recursion tree based on the given recurrence relation.
Step-02:

Determine-

- Cost of each level
- Total number of levels in the recursion tree
- Number of nodes in the last level
- Cost of the last level

Step-03:

Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.
Following problems clearly illustrates how to apply these steps.
Problem-01:

Solve the following recurrence relation using recursion tree method-

$T(n) = 2T(n/2) + n$

Solution-

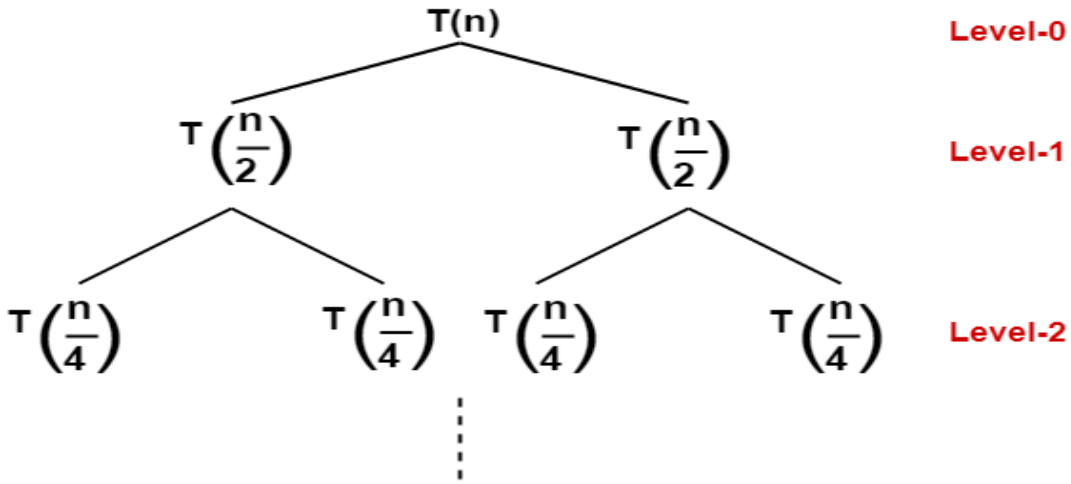
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

A problem of size n will get divided into 2 sub-problems of size n/2.
Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
At the bottom most layer, the size of sub-problems will reduce to 1.

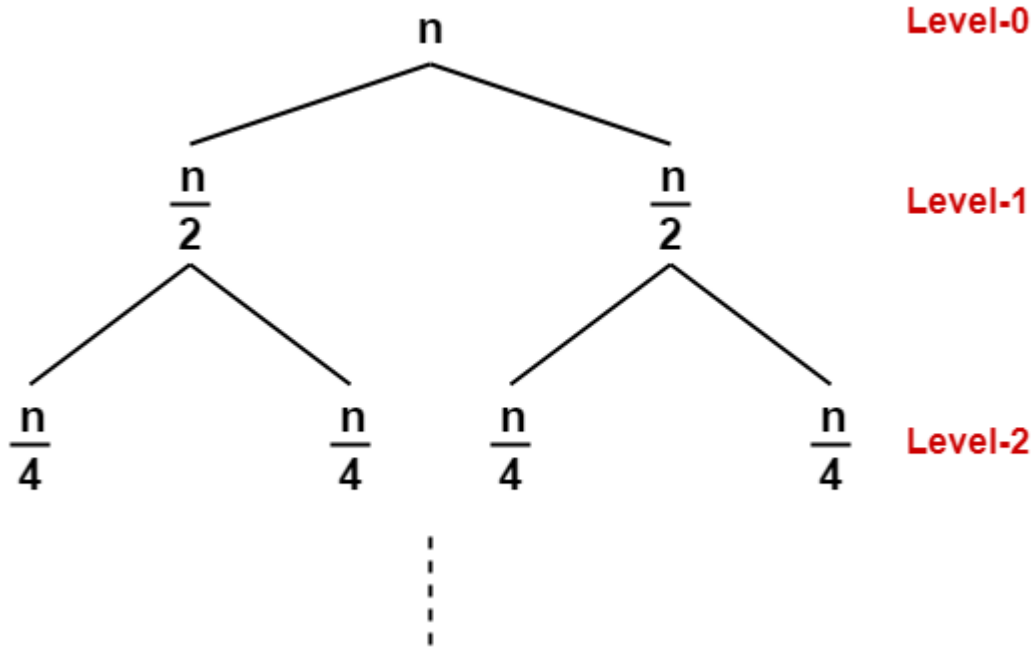
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/2 into its 2 sub-problems and then combining its solution is n/2 and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02:

Determine cost of each level-
Cost of level-0 = n
Cost of level-1 = n/2 + n/2 = n
Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.

Step-03:

Determine total number of levels in the recursion tree-
Size of sub-problem at level-0 = $n/2^0$
Size of sub-problem at level-1 = $n/2^1$
Size of sub-problem at level-2 = $n/2^2$

Continuing in similar manner, we have-
Size of sub-problem at level-i = $n/2^i$
Suppose at level-x (last level), size of sub-problem becomes 1. Then-

$n / 2^x = 1$
$2^x = n$
Taking log on both sides, we get-
$x\log 2 = \log n$
$x = \log_2 n$

∴ Total number of levels in the recursion tree = $\log_2 n + 1$

Step-04:

Determine number of nodes in the last level-
Level-0 has 2^0 nodes i.e. 1 node
Level-1 has 2^1 nodes i.e. 2 nodes
Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level-$\log_2$ n has $2^{\log_2 n}$ nodes i.e. n nodes

Step-05:

Determine cost of last level-
Cost of last level = n x T(1) = θ(n)

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_2 n \text{ levels}} + \theta(n)$$

$= n \times \log_2 n + \theta(n)$

$= n\log_2 n + \theta(n)$

$= \theta(n\log_2 n)$

Problem-02:

Solve the following recurrence relation using recursion tree method-
T(n) = T(n/5) + T(4n/5) + n

Solution-

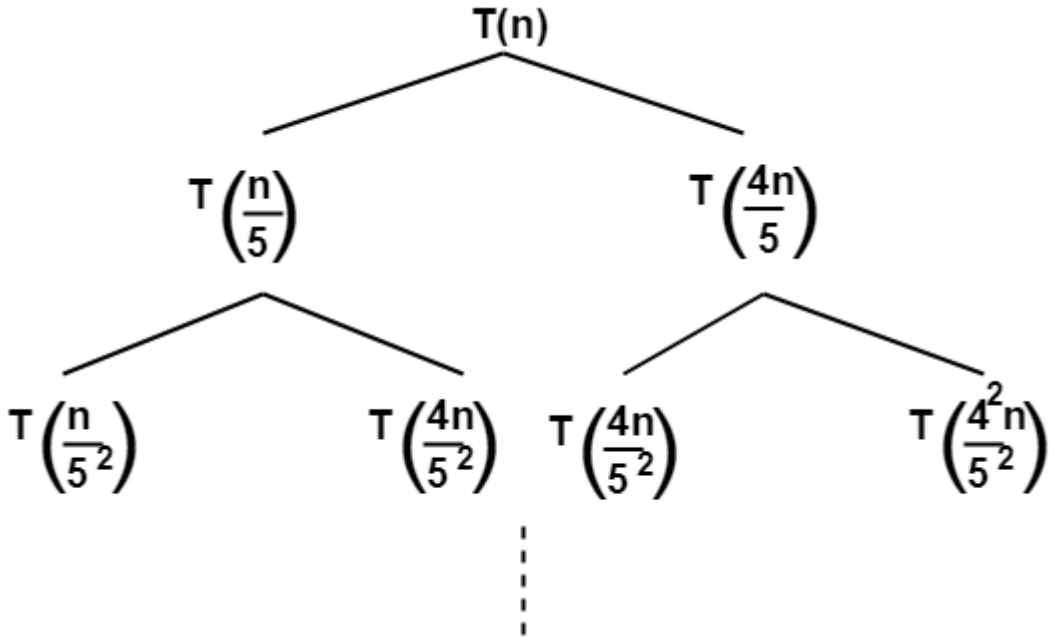
Step-01:

Draw a recursion tree based on the given recurrence relation.

The given recurrence relation shows-

A problem of size n will get divided into 2 sub-problems- one of size n/5 and another of size 4n/5.
Then, sub-problem of size n/5 will get divided into 2 sub-problems- one of size $n/5^2$ and another of size $4n/5^2$.
On the other side, sub-problem of size 4n/5 will get divided into 2 sub-problems- one of size $4n/5^2$ and another of size $4^2n/5^2$ and so on.
At the bottom most layer, the size of sub-problems will reduce to 1.

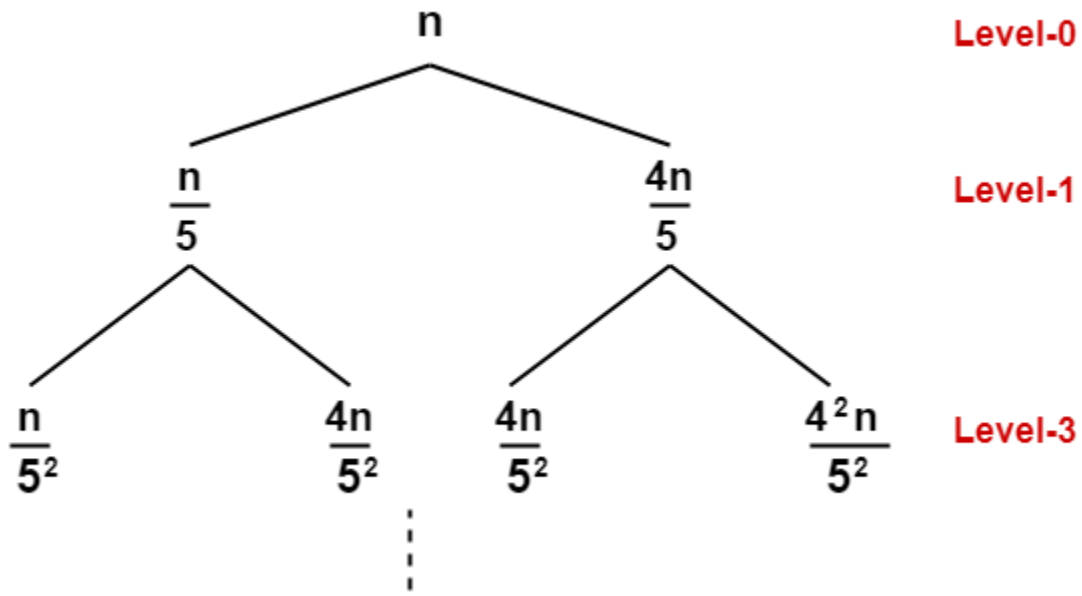
This is illustrated through following recursion tree-



The given recurrence relation shows-

- The cost of dividing a problem of size n into its 2 sub-problems and then combining its solution is n.
- The cost of dividing a problem of size n/5 into its 2 sub-problems and then combining its solution is n/5.
- The cost of dividing a problem of size 4n/5 into its 2 sub-problems and then combining its solution is 4n/5 and so on.

This is illustrated through following recursion tree where each node represents the cost of the corresponding sub-problem-



Step-02:

Determine cost of each level-
Cost of level-0 = n
Cost of level-1 = n/5 + 4n/5 = n
Cost of level-2 = $n/5^2 + 4n/5^2 + 4n/5^2 + 4^2n/5^2 = n$

Step-03:

Determine total number of levels in the recursion tree. We will consider the rightmost sub tree as it goes down to the deepest level-
Size of sub-problem at level-0 = $(4/5)^0n$
Size of sub-problem at level-1 = $(4/5)^1n$
Size of sub-problem at level-2 = $(4/5)^2n$

Continuing in similar manner, we have-
Size of sub-problem at level-i = $(4/5)^in$
Suppose at level-x (last level), size of sub-problem becomes 1. Then-

$(4/5)^xn = 1$
$(4/5)^x = 1/n$
Taking log on both sides, we get-
$x\log(4/5) = \log(1/n)$
$x = \log_{5/4}n$

∴ Total number of levels in the recursion tree = $\log_{5/4}n + 1$

Step-04:

Determine number of nodes in the last level-
Level-0 has 2^0 nodes i.e. 1 node
Level-1 has 2^1 nodes i.e. 2 nodes
Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-
Level-$\log_{5/4}n$ has $2^{\log_{5/4}n}$ nodes

Step-05:

Determine cost of last level-
Cost of last level = $2^{\log_{5/4}n} \times T(1) = \theta(2^{\log_{5/4}n}) = \theta(n^{\log_{5/4}2})$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{ n + n + n + ... \}}_{\text{For } \log_{5/4}n \text{ levels}} + \theta(n^{\log_{5/4}2})$$

$= n\log_{5/4}n + \theta(n^{\log_{5/4}2})$

$= \boldsymbol{\theta(n\log_{5/4} n)}$

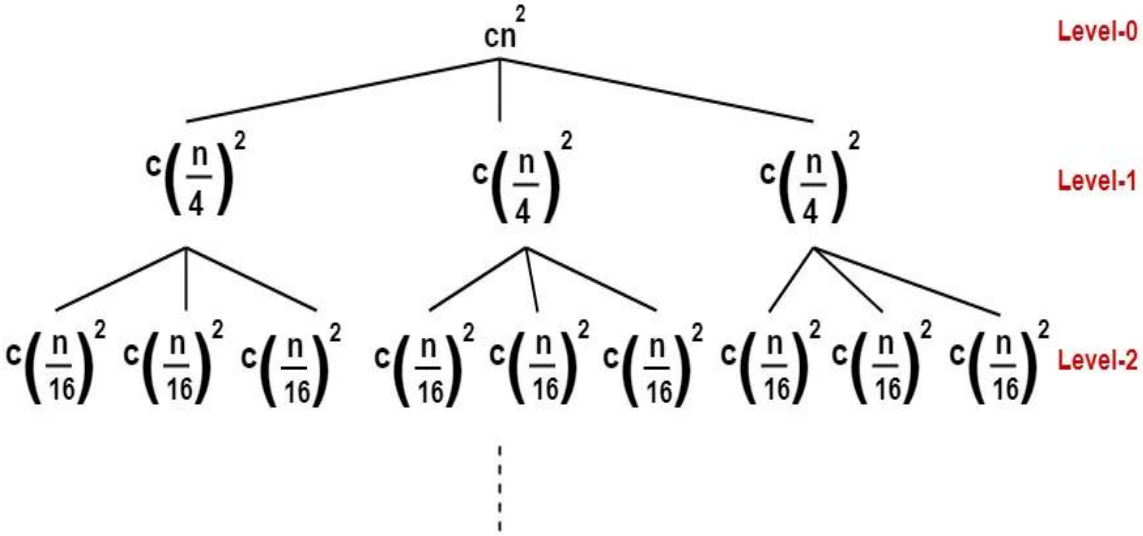
Problem-03:

Solve the following recurrence relation using recursion tree method-
$T(n) = 3T(n/4) + cn^2$

Solution-

Step-01:

Draw a recursion tree based on the given recurrence relation-



(Here, we have directly drawn a recursion tree representing the cost of sub problems)

Step-02:

Determine cost of each level-
Cost of level-0 = cn^2
Cost of level-1 = $c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$
Cost of level-2 = $c(n/16)^2 \times 9 = (9/16^2)cn^2$

Step-03:

Determine total number of levels in the recursion tree-
Size of sub-problem at level-0 = $n/4^0$
Size of sub-problem at level-1 = $n/4^1$
Size of sub-problem at level-2 = $n/4^2$

Continuing in similar manner, we have-
Size of sub-problem at level-i = $n/4^i$
Suppose at level-x (last level), size of sub-problem becomes 1. Then-

$n/4^x = 1$
$4^x = n$
Taking log on both sides, we get-
$x\log 4 = \log n$
$x = \log_4 n$

∴ Total number of levels in the recursion tree = $\log_4 n + 1$

Step-04:

Determine number of nodes in the last level-
Level-0 has 3^0 nodes i.e. 1 node
Level-1 has 3^1 nodes i.e. 3 nodes
Level-2 has 3^2 nodes i.e. 9 nodes

Continuing in similar manner, we have-
Level-$\log_4 n$ has $3^{\log_4 n}$ nodes i.e. $n^{\log_4 3}$ nodes

Step-05:

Determine cost of last level-
Cost of last level = $n^{\log_4 3}$ x $T(1) = \theta(n^{\log_4 3})$

Step-06:

Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\left\{ cn^2 + \frac{3}{16} cn^2 + \frac{9}{(16)^2} cn^2 + \ldots\ldots\ldots \right\}}_{\text{For } \log_4 n \text{ levels}} + \theta\left(n^{\log_4 3}\right)$$

$= cn^2 \{1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots\} + \theta(n^{\log_4 3})$

Now, $\{1 + (3/16) + (3/16)^2 + \ldots\ldots\ldots\}$ forms an infinite Geometric progression.

On solving, we get-

$= (16/13)\, cn^2 \{1 - (3/16)^{\log_4 n}\} + \theta(n^{\log_4 3})$
$= (16/13)\, cn^2 - (16/13)\, cn^2 (3/16)^{\log_4 n} + \theta(n^{\log_4 3})$
$= O(n^2)$

Master Theorem

Master's theorem is one of the many methods that are applied to calculate time complexities of algorithms.

Recurrence relations are equations that define the sequence of elements in which a term is a function of its preceding term. In algorithm analysis, the recurrence relations are usually formed when loops are present in an algorithm.

Problem Statement

Master's theorem can only be applied on decreasing and dividing recurring functions. If the relation is not decreasing or dividing, master's theorem must not be applied.

Master's Theorem for Dividing Functions

Consider a relation of type −

$T(n) = aT(n/b) + f(n)$

where, $a >= 1$ and $b > 1$,
n − size of the problem
a − number of sub-problems in the recursion
n/b − size of the sub problems based on the assumption that all sub-problems are of the same size.
f(n) − represents the cost of work done outside the recursion -> $\Theta (n^k \log^p n)$, where $k >= 0$ and p is a real number.

If the recurrence relation is in the above given form, then there are three cases in the master theorem to determine the asymptotic notations −

Case 1: if $a > b^k$, then $T(n)= \Theta (n^{\log_b a})$ [$\log_b a = \log a / \log b$]

Case 2: if $a = b^k$

i. If $p > -1$, then $T(n) = \Theta (n^{\log_b a} \log^{p+1} n)$
ii. If $p = -1$, then $T(n) = \Theta (n^{\log_b a} \log \log n)$
iii. If $p < -1$, then $T(n) = \Theta (n^{\log_b a})$

Case 3: if $a < b^k$,

if $p >= 0$, then $t(n) = \theta (n^k \log^p n)$.
if $p < 0$, then $t(n) = \theta (n^k)$

Master's Theorem for Decreasing Functions

Consider a relation of type −
$T(n) = aT(n-b) + f(n)$
where $a >= 1$ and $b > 1$, f(n) is asymptotically positive
Here,

n − size of the problem
a − number of sub-problems in the recursion
n-b − size of the sub problems based on the assumption that all sub-problems are of the same size.
f(n) − represents the cost of work done outside the recursion -> $\Theta(n^k)$, where k >= 0.

If the recurrence relation is in the above given form, then there are three cases in the master theorem to determine the asymptotic notations −

if a = 1, $T(n) = O(n^{k+1})$
if a > 1, $T(n) = O(a^{n/b} * n^k)$
if a < 1, $T(n) = O(n^k)$

Examples
Few examples to apply master's theorem on dividing recurrence relations −

Example 1
Consider a recurrence relation given as $T(n) = 8T(n/2) + n^2$

In this problem, a = 8, $b^k = 2^2$ and $f(n) = \Theta(n^k \log^p n) = n^2$,
giving us k = 2 and p = 0.
$a = 8 > b^k = 2^2 = 4$,
Hence, case 1 must be applied for this equation.
To calculate, $T(n) = \Theta(n\log_b a)$
$= n\log 2^8$
$= n(\log 8 / \log 2)$
$= n^3$
Therefore, $T(n) = \Theta(n^3)$ is the tight bound for this equation.

Example 2

Consider a recurrence relation given as $T(n) = 4T(n/2) + n^2$

In this problem, a = 4, b = 2 and $f(n) = \Theta(n^k \log^p n) = n^2$,
giving us k = 2 and p = 0.
$a = 4 = b^k = 2^2 = 4$, p > -1
Hence, case 2(i) must be applied for this equation.
To calculate, $T(n) = \Theta(n\log_b a \log^{p+1} n)$
$= n\log_2 4 \log^{0+1} n$
$= n^2 \log n$

Therefore, $T(n) = \Theta(n^2 \log n)$ is the tight bound for this equation.

Example 3

Consider a recurrence relation given as $T(n) = 2T(n/2) + n/\log n$

In this problem, $a = 2$, $b = 2$ and $f(n) = \Theta (n^k \log n\ p) = n/\log n$,
giving us $k = 1$ and $p = -1$.
$a = 2 = b^k = 2^1 = 2$, $p = -1$
Hence, case 2(ii) must be applied for this equation.
To calculate, $T(n) = \Theta (n \log_b{}^a \log \log n)$
$= n\log 2^2 \log \log n$
$= n.\log(\log n)$
Therefore, $T(n) = \Theta(n.\log(\log n))$ is the tight bound for this equation.