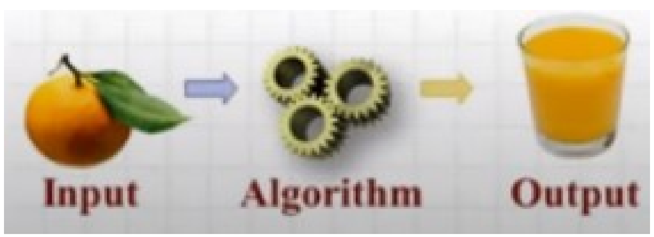
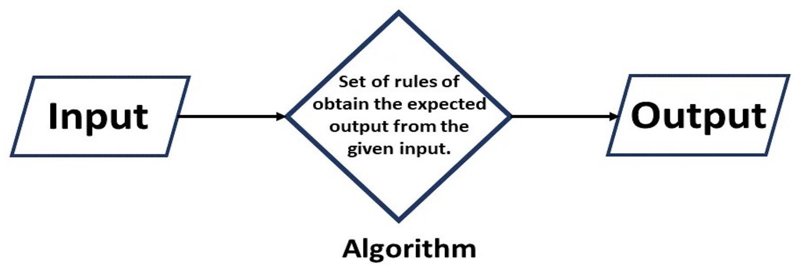
Design and Analysis of Algorithms

Algorithms

An algorithm is an effective step-by-step procedure for solving a problem in a finite number of steps.

algorithms are generally developed independently of underlying languages, which means that an algorithm can be implemented in more than one programming language.





Algorithms

Example of an Algorithm

Algorithm : Calculation of Simple Interest

Step 1: Start

Step 2: Read principle (P), time (T) and rate (R)

Step 3: Calculate I = P*T*R/100

Step 4: Print I as Interest

Step 5: Stop

Characteristics of an Algorithm

1) **Input**: It should take zero or more input.
2) **Output**: At the end of an algorithm, It should result at least one output.
3) **Unambiguity**: A perfect algorithm is defined as unambiguous, which means that its instructions should be clear and straightforward
4) **Finiteness**: An algorithm should have finite number of steps and it should end after a finite time.
5) **Effectiveness**: Because each instruction in an algorithm affects the overall process, it should be adequate.
6) **Language independence**: An algorithm must be language-independent, which means that its instructions can be implemented in any language and produce the same results.

Need of Algorithm

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. It is the best method of description without describing the implementation detail.
5. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
6. A good design can produce a good solution.
7. To understand the flow of the problem.
8. To measure the behaviour (or performance) of the methods in all cases (best cases, worst cases, average cases)
9. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
10. With the help of algorithm, we convert art into a

Analysis of algorithm

Algorithm is a combination or sequence of finite-state to solve a given problem. If the problem is having more than one solution or algorithm then the best one is decided by the analysis based on two factors.

1. CPU Time (Time Complexity)
2. Main memory space (Space Complexity)

Time Complexity: Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

Space Complexity: The space complexity can be understood as the amount of space required by an algorithm to run to completion.

Time Complexity

Time complexity of an algorithm can be calculated by using two methods:

1. Posteriori Analysis
2. Priori Analysis

Posteriori Analysis → Asymptop

1. It will give exact answer.
2. It is dependent on language of compiler and type of hardware.
3. It doesn't use asymptotic notations to represent the time complexity of an algorithm.

1. **A priori analysis** → Asymptotic
   It will give approximate answer
2. It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.

Types of algorithm

- Iterative algorithm
- Recursive algorithm

A( )
{
A( )
{

Time Analysis of Iterative algorithm (Priori)

Step1: Identify the executable statements used in the algorithm.

Step2: Calculate the frequency Count.

Step3: Add all the frequency count to find the frequency count of the entire algorithm. $f(n) =$

Step3: Convert the frequency count of algorithm into time complexity using Asymptotic notations.

Frequency

Ex: $i = 1 \rightarrow 1$

$n = 0 \rightarrow 1$

while($i < n$) $\rightarrow n$

{

$n = n + 1 \rightarrow n$

}

$F(n) = 2n + 2$

$O(n)$

$O \rightarrow$

$\Omega \rightarrow$

$\Theta \rightarrow$

Asymptotic Notation

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.
Asymptotic Notation is used to describe the running time of an algorithm - how much time an algorithm takes with a given input, n.

Types of Asymptotic notations

1. Big O notation (Worst Case)
2. Theta notation (Average Case)
3. Omega notation (Best Case)

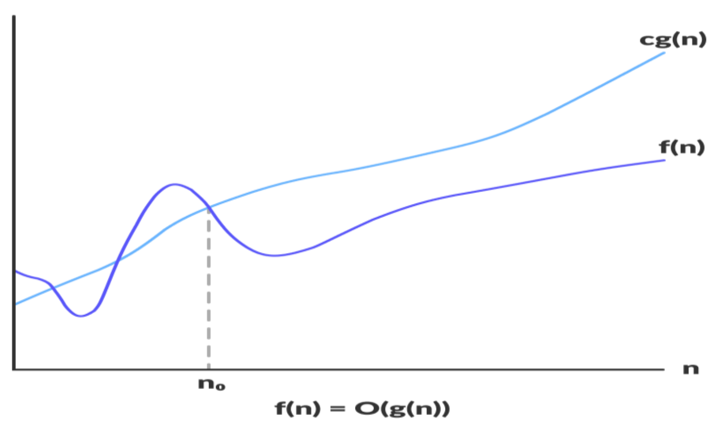
Big O notation

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.

A function **$f(n) = O(g(n))$** ,if there exists a positive integer n & n_0 and a positive constant c, such that **$f(n) \leq c.g(n)\ \forall\ n \geq n_0$, $n_0 \geq 1$, $c>0$.**

Hence function **g(n)** is an Asymptotic upper bound for function **f(n)** , as **g(n)** grow faster than **f(n).**

Example: **$f(n)=2n^2 + 5n + 1$**



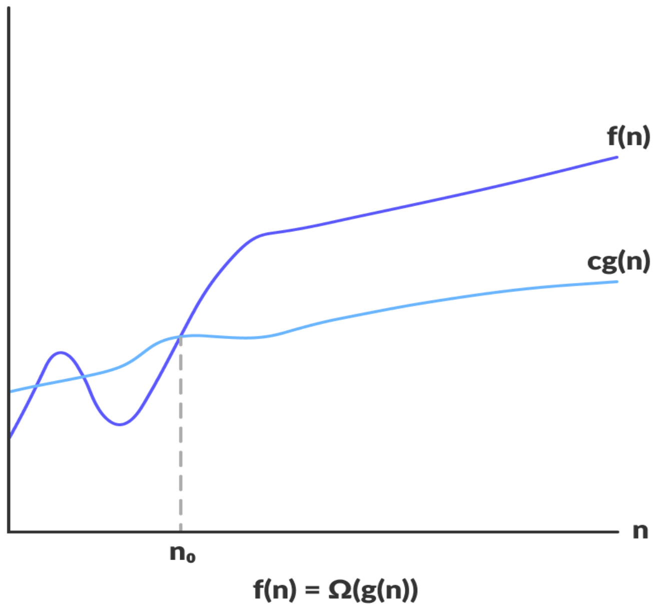
Omega Ω notation

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

A function **$f(n) = \Omega(g(n))$** ,if there exists a positive integer n & n_0 and a positive constant c, such that **$f(n) \geq c.g(n)\ \forall\ n \geq n_0$, $n_0 \geq 1$, c>0.**

Hence function **g(n)** is an Asymptotic lower bound for function **f(n).**

Example: **f(n)= 3n+2**



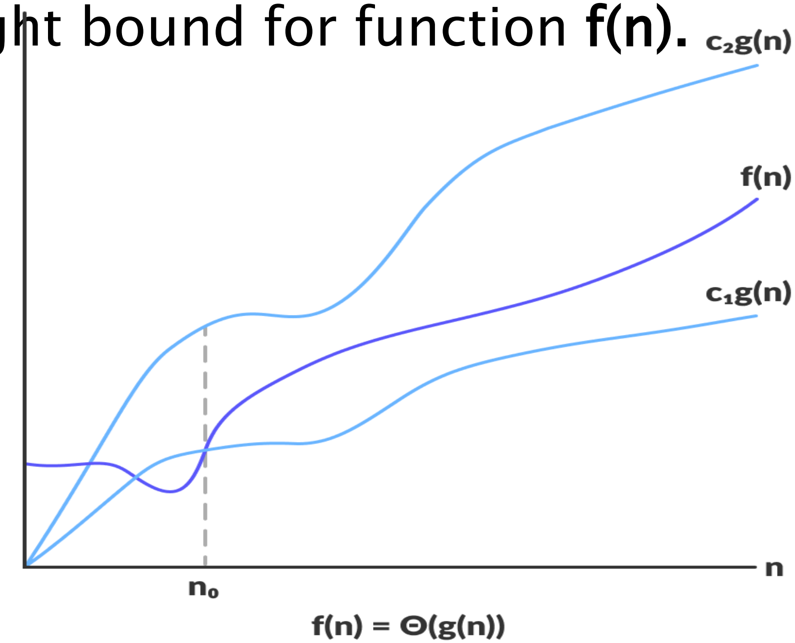
Theta notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

A function **$f(n) = \Theta(g(n))$** ,if there exists a positive integer n & n_0 and a positive constant c1 and c2, such that **$c1g(n) \le f(n) \le c2g(n)$,$\forall n \ge n_0$, $n_0 \ge 1$, $c>0$.**

Example: f(n)= 3n+2

Hence function **g(n)** is an Asymptotic tight bound for function **f(n).**



Little o notation

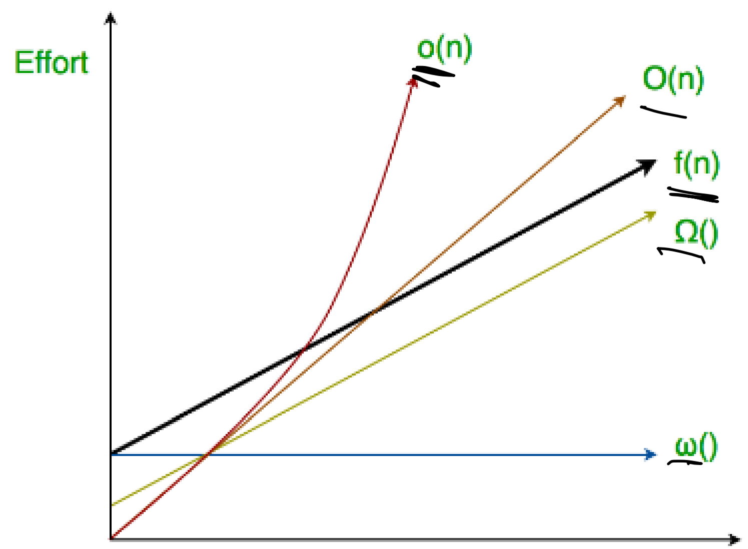
The set of functions f(n) are strictly smaller than cg(n) , meaning that little-o notation is a stronger upper bound than big-O notation. the little-o notation does not allow the function **f(n)** to have the same growth rate as **g(n).**

A function **f(n) = o(g(n))** ,if there exists a positive integer n & n_0 and a positive constant c, such that **f(n) < c.g(n) ∀ n≥n_0 , n_0 ≥ 1, c>0.**

In mathematical ter

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

Example: **f(n)=2n² + 5n + 1**



Little ω notation

O Ω ω

Little omega (ω) notation is used to describe a loose lower bound of f(n).

A function **f(n) = ω (g(n))** ,if there exists a positive integer n & n_0 and a positive constant c, such that **f(n) > c.g(n) ∀ n≥n_0 , n_0 ≥ 1, c>0.**

In mathematical terms:

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

Example: **f(n)=2n² + 5n + 1**

$g(n) = n$ $\Rightarrow \lim_{n\to\infty} \frac{2n^2+5n+1}{n}$

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

$2n^2+5n+1 = \omega(n, \log n, \log\log n \ldots)$

o

$f(n) = n^3 + n + 1$

$\cancel{g(n) = n^3}$ $\Rightarrow$ $f(n) = o(g(n))$ X

$g(n) = n^4$

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

$\Rightarrow$

$\lim_{n \to \infty} \frac{n^3 + n + 1}{n^3}$

$f(n) = o(n^4)$

$\lim_{n \to \infty} \left(\frac{n^3}{n^3} + \frac{n}{n^3} + \frac{1}{n^3} \right)$

$\lim_{n \to \infty} \left(1 + \frac{1}{n^2} + \frac{1}{n^3} \right) = 1$

X

$\Rightarrow$

Recurrence Relation

When an algorithm contain a recursive call to it self ,its running time can be describe by recurrence equation.

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the Procedures is described by the recurrence.

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

Example1 :

```
Void Sum( int n)
{
If(n>0)
{
Print(n);
Sum(n-1);
}
```

$\to T(n)$

1

1

$T(n-1)$

$T(n) = T(n-1)+2$

$$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+1 & n>0 \end{cases}$$

Example2 :

```
Void fun(int n)
{
If(n>0)
{
for(i=0;i<n; i++)
{
Print(n);
}
fun(n-1);
}
```

Example 3 :

```
Void fib (int n)
{
If(n<0)
{
Return 1
}
Else
{
Return fib(n-1) + fib(n-
    1);
}
}
```

$\to T(n)$

1

$\to T(n-1) + T(n-2) + 1$

$T(n) = T(n-1)+T(n-2)+2$

$$T(n) = \begin{cases} 1 & \text{if } n=0 \\ T(n-1)+T(n-2)+c & \text{if } n>0 \end{cases}$$

1. Iteration Method

Binary Search(arr, x, low, high)

repeat till low = high

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid])

low = mid + 1

Else

high = mid - 1

2. Recursive Method

Binary Search (arr, x, low, high) T(n)

if low > high

return False

Else

mid = (low + high) / 2 → c

if x == arr[mid]

return mid

else if x > arr[mid]

return binary Search(arr, x, mid + 1, high)

T(n/2)

else or

return binary Search(arr, x, low, mid - 1)

T(n/2)

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

There are four methods for solving Recurrence:

1. Substitution Method → Back Subs, For
2. Iteration Method
3. Recursion Tree Method
4. Master Method

$2T(\frac{k}{2}) + k \le ck\log k$

$\Rightarrow 2T(\frac{n}{2^2}) + \frac{n}{2} \le c\frac{n}{2}\log n/2$

$n = 2$

$2T(2/2) + 2 \le c\,2\log_2 2$

$\Rightarrow 2T(1) + 2 \le c\,2 \Rightarrow$

$2\times1 + 2 \le 2c$

$4 \le 2c$

$c \ge 2$

$n = 2, 3, 4, 5, \dots k$

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.

Example: $T(n) = 2T(n/2) + n$, $T(1) = 1$

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

$f(n) \le c\,g(n)$

(I) The solution $T(n) = O(n\log n)$

(II) $T(n) \le c\,n\log n$ — 1

$\Rightarrow (2T(n/2) + n \le c\,n\log n$ — 2

Put $n = 1$ $\Rightarrow 2T(1/2) + 1 \le c\,1\log 1 \Rightarrow 1 \le 0$ False

Example: $T(n) = 2T(n/2) + n$, $T(1) = 1$

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

$f(n) \le c\,g(n)$

Sol^n: Step 1 Guess $T(n) = O(n\log n)$

$T(n) \le cn\log n$ —— (1)

By Appl^n M's

put $n=1$

$T(1) \le c\,1\log 1$

$1 \le 0.c$ false

put $n=2$

$T(2) \le c\,2\log 2$

$2T(1)+2 \le 2c$

$4 \le 2c$ $c \ge 2$ true

$n = 2, 3, 4 \ldots\ldots k$ $\Rightarrow 2 \longrightarrow n \Rightarrow k = n/2$

$n=k$

$T(k) \le c\,k\log k$ —— (2)

$T(n/2) \le c\,\frac{n}{2}\log\frac{n}{2}$ —— (3)

$T(n) = 2\left[c\frac{n}{2}\log\frac{n}{2}\right] + n$

$T(n) = cn\left[\log\frac{n}{2}\right] + n$

$T(n) = cn\log n - cn\log 2 + n$

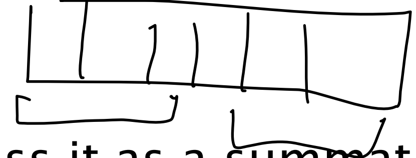
$T(n) = O(n\log n)$

H.W

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

$T(1) = 1$

Iteration Methods



It means to expand the recurrence and express it as a summation of terms of n and initial condition.

$T(n) = T(n/2) + 1$, $T(1) = 1$ —— (I)

$T(n/2) = T(n/4) + 1$

$T(n) = T(n/4) + 1 + 1$

$T(n) = T(\frac{n}{2^2}) + 2$ —— (II)

$T(n/4) = T(\frac{n}{8}) + 1$

$T(n) = T(\frac{n}{8}) + 2 + 1$

$T(n) = T(\frac{n}{2^3}) + 3$

⋮

$T(n) = T(\frac{n}{2^k}) + k$ —— (III)

$T(n) = T(1) + k$

$T(n) = 1 + k$ —— (IV)

Let assume

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log_2 n = \log_2 2^k$

$k = \log_2 n$

$T(n) = 1 + \log_2 n$

$T(n) = O(\log_2 n)$

Iteration Methods

$T(n) = T(n-1) + n$, $T(0) = 1$

Sol^n -

$T(n) = T(n-1) + n$ — ①

put $n = n-1$

$T(n-1) = T(n-2) + (n-1)$ — ②

from eq^n ①

$T(n) = T(n-2) + (n-1) + n$ — 2.a

put $n = n-2$

$T(n-2) = T(n-3) + n-2$ — ③

from eq^n

$T(n) = T(n-3) + (n-2) + (n-1) + n$

$\vdots$

$T(n-k) + (n-(k-1)) + \cdots + n$ — ④

from eq^n (IV)

$T(n) = T(0) + (n+1) + \cdots + n$

$\therefore n-k = 0$

$n = k$

put $n = k$

$1+2+3+4 \cdots + k$

$\Rightarrow \dfrac{k(k+1)}{2}$

$T(n) = O(n^2)$

Iteration Methods

H.W

$T(n) = T(n-1) + n^2, \ T(0) = 1$

Iteration Methods H.W

$T(n) = 2T(n-1)$, $T(0) = 1$

Iteration Methods

H.W

T (n) = T(n–1) + log n, T(0) =1

$T(n) = T(n-1) + \log n$ —— ①

put $n = n-1$ in eqⁿ ①

$T(n-1) = T(n-2) + \log(n-1)$ —— *

put $T(n-1)$ in eqⁿ ①

$T(n) = T(n-2) + \log(n-1) + \log n$ —— ②

put $n = n-2$ in eqⁿ ②

$T(n-2) = T(n-3) + \log(n-2)$ —— **

$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n$ —— ③

~~k~~n

⋮

$T(n) = T(n-k) + \log(n-(k-1)) + \log(n-(k-2)) + \dots \log n$ —— ④

From eqⁿ ④

$T(n) = \boxed{T(0)} + \log(n-k+1) + \log(n-k+2) + \dots + \log n$

$\because n-k=0$

$T(n) = 1 + \log 1 + \log 2 + \log 3 + \dots \log n$

$T(n) = 1 + \log(1*2*3*\dots n)$

$T(n) = 1 + \log n!$

upper Bound $n! = n^n$

$\Rightarrow T(n) = \Theta(\log n^n)$

$T(n) = O(n \log n)$

Iteration Methods

$T(n) = nT(n-1) + 1$, $T(0) = 1$

Sol^n:

$T(n) = nT(n-1) + 1$ —— ①

put $n = n-1$

① $T(n-1) = (n-1)T(n-2) + 1$

from eq^n ①

$T(n) = n[(n-1)T(n-2) + 1] + 1$

$T(n) = n(n-1)T(n-2) + n + 1$ —— ②

put $n = n-2$ in eq^n ①

$T(n-2) = (n-2)T(n-3) + 1$

$T(n) = n(n-1)[(n-2)T(n-3) + 1] + n + 1$

$T(n) = n(n-1)(n-2)T(n-3) +$

$T(n) = n(n-1)(n-2)T(n-3) + n(n-1) + n + 1$

$T(n) = n(n-1)(n-2)T(n-3) + n^2 + 1$

$T(n) = n(n-1)(n-2)\ldots(n-k-1)T(n-k) + n^{k-1} + 1$

$T(n) = n(n-1)(n-2)\ldots T(0) + n^{k-1} + 1$

$T(n) = n(n-1)(n-2)\ldots 1 + n^{k-1} + 1$

$T(n) = n! + n^{k-1} + 1$ —— ③

$\because n - k = 1$

$\Rightarrow k = n - 1$

$T(n) = n! + n^{n-1-1} + 1$

$T(n) = n! + n^{n-2} + 1$

$T(n) = O(n^n)$

Iteration Methods

$T(n) = T(\sqrt{n}) + 1, n \geq 2$

$\because \sqrt{n} = n^{\frac{1}{2}}$

$T(2) = 1$

$T(n) = T(n^{\frac{1}{2}}) + 1 \quad \text{—— (I)}$

put $n = n^{\frac{1}{2}}$

I, 485, 486, 487, 490, 491, 492, 494, 6, 504, 505, 506, 509, 510, 512, 515, 516, 520, 521, 52

D' 181, 185, 188, 190, 198, 210, 222, 231,

Lateral Present

10, 13, 37, 005

D 183, 184, 187, 190, 192, 193, 196, 197, 198, 199, 202, 203, 210, 217, 219, 220, 221, 222, 224, 225, 226, 227, 231, 232, 233, 235, 6, 237,

D 492, 499, 504, 505, 509, 510, 522,

Present

63, 28, 45, 08, 09, 32, 48, T004, 36, 40, 026, 12, Shivangi

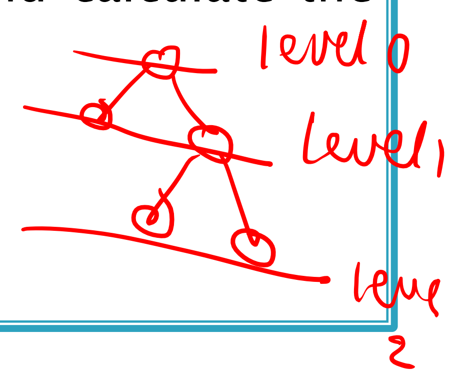
Recursion Tree method

1. The Recursion Tree Method is a way of solving recurrence relations. In this method, a recurrence relation is converted into recursive trees.

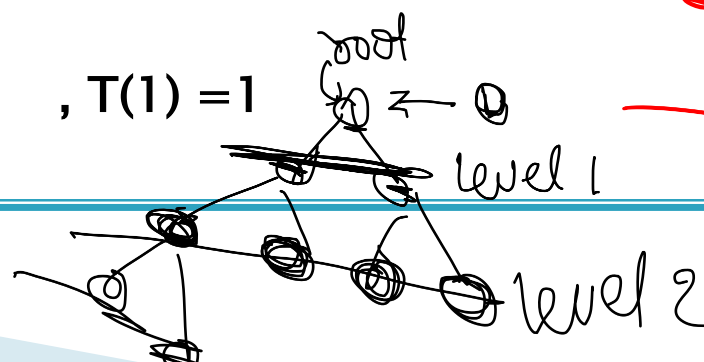
2. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.
3. It is useful when divide and conquer algorithm is used.

Steps to solve recurrence relation using recursion tree method:

- ➢Draw a recursive tree for given recurrence relation
- ➢Calculate the cost at each level and count the total no of levels in the recursion tree.
- ➢Count the total number of nodes in the last level and calculate the cost of the last level
- ➢Sum up the cost of all the levels in the recursive tree



Example: $T(n) = 2T(n/2) + n$, $T(1) = 1$

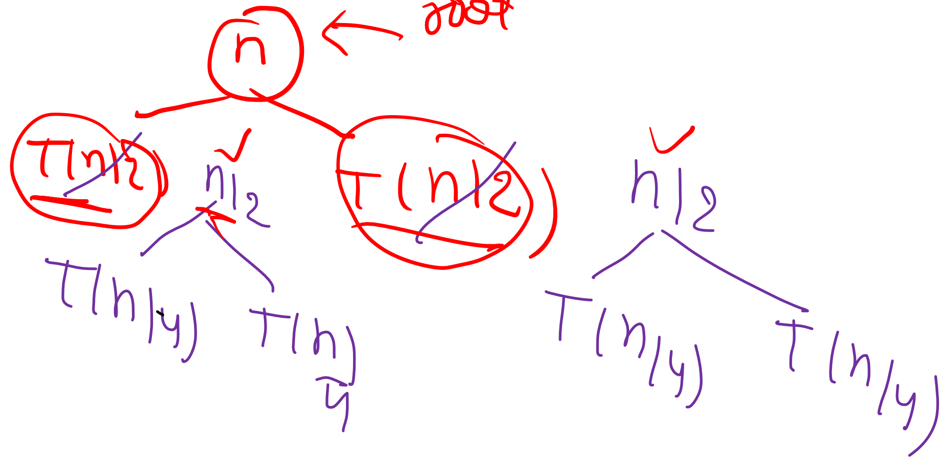


Recursion Tree method

$T(n) = 2T(n/2) + n$, $T(1) = 1$

$f(n)$

root



put $n = n/2$

$T(\frac{n}{2}) = 2T(n/4) + \frac{n}{2}$

put $n = n/4$

$T(n/4) = 2T(n/8) + \frac{n}{4}$

$\Rightarrow T(\frac{n}{2^k})$

$T(1) \Rightarrow$

$\because \frac{n}{2^k} = 1$

$\Rightarrow n = 2^k$

$\Rightarrow \log_2 n = \log_2 2^k$

$\Rightarrow k = \log_2 n$

Recursion Tree method

$$T(n) = 2T(n/2) + n^2, \quad T(1) = 1$$

$T(n) = 2T(n/2) + n^2$, $T(1) = 1$

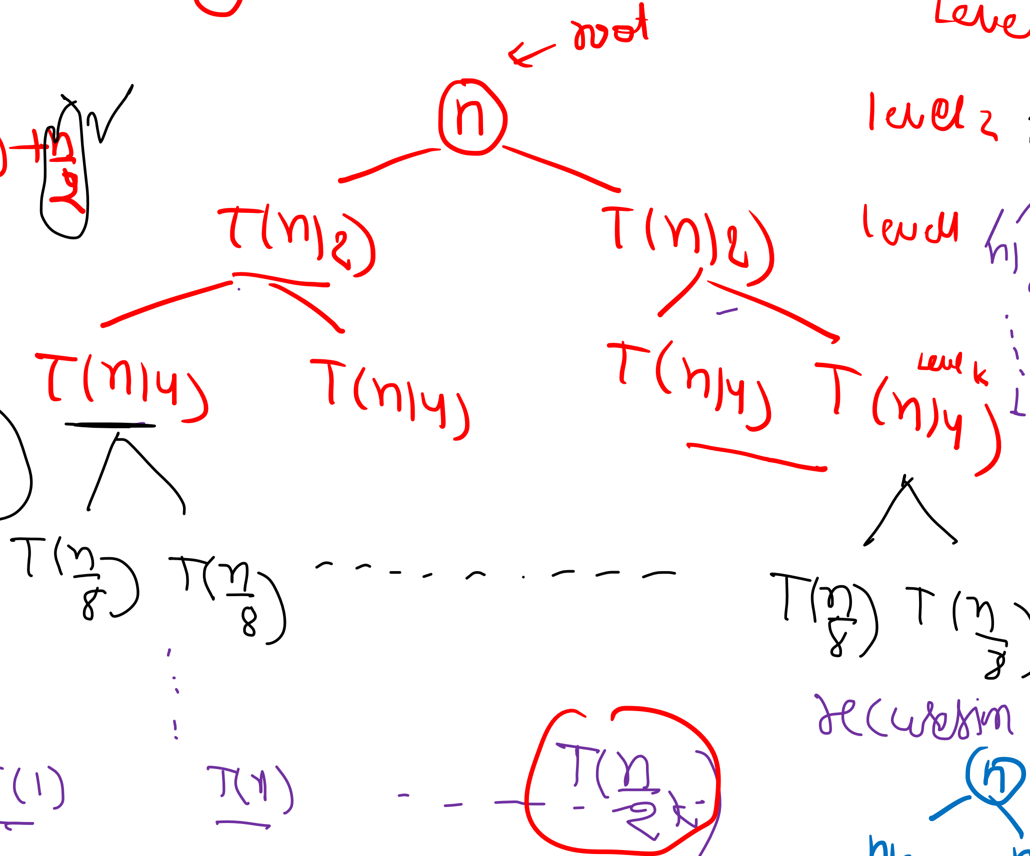
$f(n) = n$

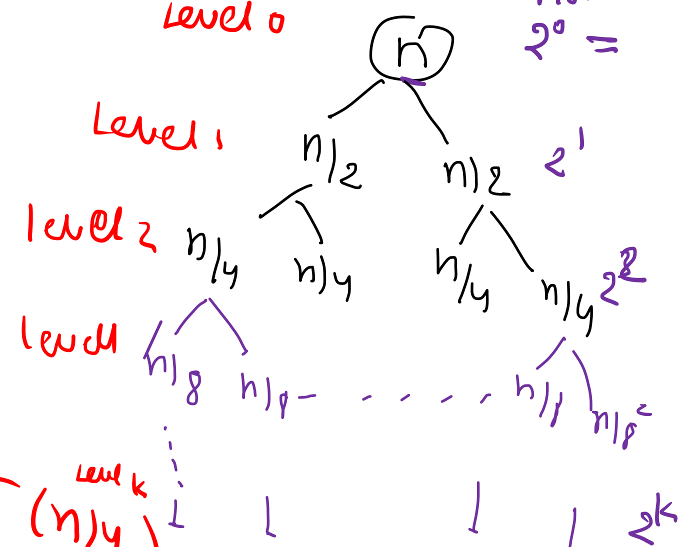
put $n = \frac{n}{2}$

$T(n/2) = 2T(n/4) + \frac{n}{2}$

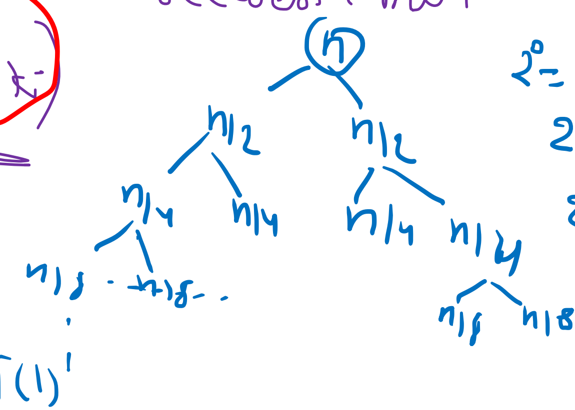
put $n = \frac{n}{4}$ in (1)

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$





recursion meth



no. of node	cost
$2^0 = 1$	$1 \cdot n = n$
$2^1 = 2$	$\Rightarrow 2 \cdot \frac{n}{2} = n$
$2^2 = 4$	$\Rightarrow 4 \cdot \frac{n}{4} = n$
$2^3 = 8$	
2^k	$\Rightarrow 2^k \cdot \frac{n}{2^k} = n$

$\Rightarrow n + n + n + n \ldots$

$\Rightarrow kn$

$T(n) = O(n \log n)$

Master Method

Master's Theorem is the best method to quickly find the algorithm's time complexity from its recurrence relation.

This theorem can be applied to decreasing as well as dividing functions.

In this ,Problem is divided into **'a'** sub problems , each of size **n/b** where **a** and **b** are positive constants. the cost of dividing the problem and combining the results of the sub problems is described by the function **f(n)**.

Master's Method for Dividing Functions

$$T(n) = aT(n/b) + f(n), \quad \text{with } a \geq 1,\ b>1 \text{ and } f(n) \geq 0$$

where,

n = size of input

n's , n log n

Master Method

Cases

① Big-oh $f(n) \le c\,g(n)$

② Omega $f(n) \ge \Omega(g(n))$

③ Theta $c_1 g(n) \le f(n) \le c_2 g(n)$

- Theorem 1: If $f(n) = O(n^{\log_b a - \varepsilon}), \exists \varepsilon > 0$,

  then, $T(n) = \Theta(n^{\log_b a})$
- Theorem 2: If $f(n) = \Theta(n^{\log_b a})$,

  then $T(n) = \Theta(n^{\log_b a} logn)$
- Theorem 3: If $f(n) = \Omega(n^{\log_b a + \varepsilon}), \exists \varepsilon > 0$,

  then $T(n) = \Theta(f(n))$

  iff the Regularity condition holds

  $a.f(\frac{n}{b}) \leqslant C.f(n)$ for C < 1.

Master Method

Example

Solve $T(n) = 4T(n/2) + n$, using master Method.

Master Method

Example

Solve $T(n) = T(n/2) + 1$, using master Method.

Master Method

Example

Solve $T(n) = 2T(n/2) + n^4$, using master Method.

Cases When Master Method Fails

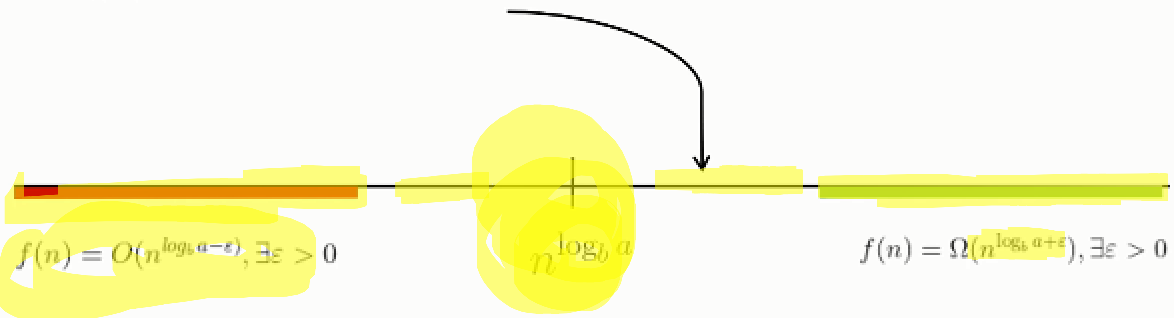
Master Method for non polynomial f(n)

- Note that the three cases do not cover all the possibilities for f(n). There is a gap between case 1 and 2 when f(n) is smaller than $n^{\log_b a}$ but not polynomially smaller.

- Similarly there is a gap between case 2 and case 3 where f(n) is larger than $n^{\log_b a}$ but not polynomially larger. That is failed to satisfy the regularity condition.

f(n) falls into the gap :

- f(n) falls into the gap :



Extension of Master Method

Master's theorem solves recurrence relations of the form–

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta\left(n^k \log^p n\right)$$

Here, a >= 1, b > 1, k >= 0 and p is a real number.

Master Theorem Cases

To solve recurrence relations using Master's theorem, compare **a** with **b^k**.

Case–01:

If $a > b^k$, then $T(n) = \theta\,(n^{\log_b a})$

Case–02:

If $a = b^k$ and

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

Extension of Master Method

Master Theorem Cases

Case-03:

If $a < b^k$ and

If $p < 0$, then $T(n) = O(n^k)$

If $p >= 0$, then $T(n) = \theta(n^k \log^p n)$

Example of Extension of Master Method

$T(n) = 3T(n/2) + n^2$

$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ —— 1

compare with R.E

$a = 3, \; b = 2, \; k = 2, \; p = 0$

$b^k \Rightarrow 2^2 = 4$

III $a < b^k$

I $p >= 0$

$T(n) = \Theta(n^k \log^p n)$

$\Rightarrow T(n) = \Theta(n^2 \log^0 n)$

$T(n) = \Theta(n^2)$

Example of Extension of Master Method

$T(n) = 2T(n/2) + n \log n$ —— (1)

Solⁿ: $T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$ —— (2)

compare eqⁿ (1) & (2)

$a = 2$, $b = 2$, $k = 1$ $p = 1$

$\because b^k = 2^1 = 2$

Case-II: $a = b^k$

$p >= -1$ $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$ —— (3)

$T(n) = \Theta(n^{\log_2 2} \log^1 n)$

$= \Theta(n \log^2 n)$

Example of Extension of Master Method

$T(n) = 2T(n/4) + n^{0.51}$ —— 1

$\cancel{\frac{n}{\log n}} \Rightarrow n \log^{-1} n$

$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ —— 2

compare Eq^n 1 & 2

$a = 2, \quad b = 4, \quad k = 0.51, \quad \underline{p = 0}$

$\therefore b^k = 4^{0.51} = \underline{2.2}$

Case-3 $\quad a < b^k$

$p >= 0, \quad T(n) = \Theta(n^k \log^p n)$

$\Rightarrow T(n) = \Theta(n^{0.51} \log^0 n)$

$= \underline{\Theta(n^{0.51})}$

Example of Extension of Master Method

$T(n) = \sqrt{2}T(n/2) + \log n$ —— (1)

$T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k \log^p n)$ —— (2)

compare eqn (1) & eqn (11)

$a = \sqrt{2}, \quad b = 2, \quad k = 0, \quad p = 1$

$\because b^k = 2^0 = 1$

Case 1 $a > b^k$

then $T(n) = \Theta(n^{\log_b a})$

$= \Theta(n^{\log_2 \sqrt{2}})$

Example of Extension of Master Method

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

Example of Extension of Master Method

$T(n) = 2T(n/2) + n / \log^2 n$

$T(n) = T(9\,n/10) + n$

$T(n) = 4T(n/2) + n^2\sqrt{n}$

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

Example of Extension of Master Method

$T(n) = 2T(n/4) + \sqrt{n}$

Example of Extension of Master Method

$T(n) = \sqrt{2}T(n/2) + \log n$

Example of Extension of Master Method

$T(n) = 3T(n/3) + n/2$

Master's Theorem for Decreasing Functions

$T(n) = T(n-b) + f(n)$, where $f(n)=\theta(n^k)$

Here a, b, and k are constants that satisfy the following conditions:

$a>0$, $b>0$,$k>=0$

1. If $a<1$ then $T(n) = O(n^k)$
2. If $a=1$ then $T(n) = O(n^{k+1})$
3. if $a>1$ then $T(n) = O(n^k a^{n/b})$

Master's Theorem for Decreasing Functions

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

Master's Theorem for Decreasing Functions

$$T(n)=1/2T(n-1)+n^2$$