KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

Website: www.ktunotes.in

Module 1 → Introduction to Algorithm: Analysis: Characteristics of Algorithm, Criteria for Analysing Algorithms, Time and Space Complexity: Best, Worst and Average Case Complexities, Asymptotic Notations → Big-oh, Big Omega(Ω), Big-Theta (Θ), Little-oh (o) and Little Omega (ω) and their properties. Classifying functions by their asymptotic growth rate, Time and Space Complexity Calculation of simple algorithm.

Analysis of Recursive Algorithms: Recurrence Equations, Solving Recurrence Equations → Iteration method, Recursion Tree Method, Substitution Method and Master's Theorem (Proof Not Required).

→ System Life Cycle: The system life cycle is a series of stages that are worked through during the development of a new information system. A lot of time and money can be wasted if a system is developed that does not work properly or do exactly what is required of it.

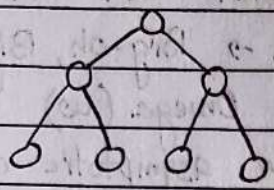
There are 5 phases in System Life Cycle:

I. Requirement → Understanding the information you are given (inputs) and what results you are to produce (outputs). It is a rigorous description of the input and output which covers all cases.

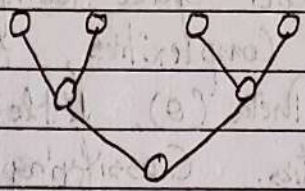
II. Analysis → In this phase, we begin to break the problem down into manageable pieces. Two approaches are:

Bottom-Up Approach: It is an older, unstructured strategy, starting the design with specific modules and build them into more complex structures, ending at the top.

Top Down Approach : Program will serve and uses this end product to divide the program into manageable segments.



Top down Approach



Bottom-up Approach

III: Design → There are several data objects such as maze, a polynomial or a list of names.
For each object, there will be some basic operations to perform on it such as print the maze, add two polynomial, or find a name in the list.
Assume that these operations already exist in the form of procedures.
Write algorithm which solves the problem according to the requirements.
Use a notation which is natural to the way you wish to describe the order of processing.

IV. Refinement and Coding → Chose representations for data objects. (a maze as a two dimensional array of zeroes and ones, a polynomial as one dimensional array of degree and co-efficients, a list of names possibly as an array.
Write algorithms for each of the operations on these objects.

The order in which you do this may be crucial, because once you chose a representation, the resulting algorithm may be inefficient.

V. Verification → This phase consists of developing correctness proofs for the program, testing the program and removing errors.

Correctness Proof : Mathematical techniques are time consuming for large projects.

Testing: Requires the working code and set of test data. This data should be developed carefully so that it includes all possible scenarios.

Error Removal : If done properly, the correctness proofs and system test will indicate erroneous codes.

→. Data Structure :→ Data structure is the structural representation of logical relationships between elements of data. We can say that data structure is a collection of data elements which are characterized by accessing functions. The accessing functions are used to store and retrival individual data items.

Data: Value or set of values.
Eg: 34, abc, 10/10/2020.

Entity: Something that has certain attributes or properties which may be assigned a value
Eg: EMPLOYEE.

Information : Meaningful or processed data.

Data	Meaningful
30	Age

Data Manipulation: Storage, alter, update, retrieve.

Linear → Static (array), Dynamic (Linked list, queue, stack)

Non-linear → Trees, Graph, Tables, Sets.

→ **Algorithm →** Algorithm is a step by step procedure which defines a set of instructions to be executed in certain order to get the desired output. An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.
Algorithms are generally created independent of underlying languages i.e. an algorithm can be implemented in more than one programming languages.

Characteristics : The following are the characteristics of an algorithm :

- **Unambiguous →** Algorithm should be clear and unambiguous. Each of its steps or phases, and their input / output should be clear and must lead to only one meaning.
- **Input →** An algorithm should have 0 or more well defined inputs.
- **Output →** An algorithm should have 1 or more well defined outputs and should match the desired results.
- **Finiteness →** Algorithm must terminate after a finite number of steps.
- **Feasibility →** Should be feasible with the available resources.
- **Independent →** An algorithm have step by step directions

which is independent of any programming code.

- Efficiency → Algorithm should efficiently use the resources available to the system.

The computational time and memory used by the algorithm should be as less as possible.

Eg. I. Start
II. Enter two numbers
III. Sum of numbers
IV. Print the sum
V. Stop.

Analysis of an Algorithm:

To understand how good an algorithm is, depends on:

- Correctness (algorithm definition criteria 1 & 2)
- Execution time (time complexity)
- amount of memory (space complexity)
- simplicity and clarity (criteria 3 and 5)
- optimality. (Best, average and worst case).

A complete analysis of the running time of an algorithm involves the following steps:

- Implement the algorithm completely.
- Determine the time required for each basic operation.
- Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
- Develop a realistic model for the input to the program.
- Analyze the unknown quantities, assuming the modelled i/p.
- Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

→ Difference between Priori and Posteriori

	Priori Analysis	Posteriori Analysis
i)	It is an absolute analysis.	It is a relative analysis.
ii)	It is independent of language of compiler and types of h/w.	It is dependent on language of compiler and type of hardware.
iii)	It will give approximate answer.	It will give exact answer.
iv)	It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.	It does not use asymptotic notations to represent the time complexity of an algorithm.
v)	The time complexity of an algorithm using a priori analysis is same for every system.	The time complexity of an algorithm using a posteriori analysis differ from system to system.
vi)	If the algorithm running faster, credit goes to the programmer.	If the time taken by the program is less, then the credit will go to compiler and hardware.

→ Performance Analysis : Performance of an algorithm is a process of making evaluative judgement about algorithms. Performance of an algorithm means predicting the ~~about~~ resources which are required to an algorithm to perform its task.

For instance, if we want to go from city "A" to city "B", there can be many ways of doing this. We can go by

flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us.

Similarly, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve problem, we analyze them and pick the one which is best suitable for our requirements.

In this, we compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement etc.

Generally, the performance of an algorithm depends on the following elements:

a) Whether that algorithm is providing the exact solution for the problem?
b) Whether it is easy to understand?
c) Whether it is easy to implement?
d) How much space / memory is required to solve the problem?
e) How much time it takes to solve the problem? etc.

Performance analysis of an algorithm is performed by using the following measures:

- Space required to complete the task of that algorithm (space complexity). It includes program space and data space.
- Time required to complete the task of that algorithm (Time complexity).

Space Complexity → Space complexity of an algorithm represents the amount of memory/ space required by the algorithm in its lifecycle.
The space required by an algorithm is equal to the sum of the following two components:

a) Fixed Part: A fixed part that is a space required to store certain data and variables that are independent of the size of the problem.
For example, simple variables and constants used, program size etc.

b) Variable Part: (Auxiliary Space), A variable part is a space required by variables, whose size depends on the size of the problem.
For example, dynamic memory allocation, recursion stack space etc.

Space Complexity $S(P)$ of any algorithm (P) is:
$S(P) = C + SP(I)$.
where C is the fixed part and $SP(I)$ is the variable part of the algorithm which depends on instance characteristic.

Time Complexity → Time Complexity of an algorithm represents the amount of time required by the algorithm to run to completion.
Time requirements can be defined as a numerical function(T_n) where T(n) can be measured as the number of steps, provided each step consumes constant time.
n → size of input.

The time complexity also depends on the amount of data input to an algorithm; we can calculate the order of magnitude for the time required.

Total time required = Compile time + execution time.
$T(P) = C + IP(I)$.
Constant depends on i/p.

The more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm.

Space Complexity can be obtained by using two methods:

I. Frequency Count : Frequency count is defined as the number of times the statement is executed in the program.
Eg A. ---
x = x+2

The frequency count of the statement in the program segment A is 1.
(It gets executed once).

Program Segment B.

```
---
for K=1 to n do
  x = x+2
end
end
---
```

The frequency count of the statement is n, since the loop in which the statement is embedded executes n(n>=1) times.

Program Segment C

```
---
for j = 1 to n do
    for x=1 to n do
      x = x+2
    end
  end.
```

The statement is executed $n^2 (n>=1)$ times, since the statement is embedded in a nested for loop, executing n times each.

II. Tabular Method:

- define steps
- determine the frequency of steps based on 'n' (total no. of time)
- add up the steps
- reduced.

* Calculation of Time Complexity :

- Comments & Declarative Statements → count is zero.
- Expressions and assignment statements → Count is 1.
- Iteration statements → It depends on the expressions present in the program.

Eg. for loop, Do-while, while statement.

- Switch statements → It consists of a header followed by a number of cases. Header Switch is given count 1 since it has an expression, then count for cases depends on condition.

- If else statements → It also depends on the expressions present in the statement.

- Function Invocation → It is given a step count of one.

- Memory Management Statements → Eg malloc, calloc, → count of 1.

- Function Statements → Count of zero since these are count in their invocations as said in function invocation.

- Jump statements → Eg Continue, break, return etc. → count of 1.

[ORDER OF MAGNITUDE OF AN ALGORITHM : SUM OF OCCURENCES OF STATEMENTS CONTAINED IN IT]

→ Cases for Analysis of Algorithm :

I. Worst - Case Complexity : The worst - case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n.
For example, The linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list.

II. **Best-Case Complexity :** The best-case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n.
For example, the best case for a simple linear search on an array occurs when the desired element is the first in the list.

III. **Average Case Complexity :** The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n.
For example, the linear search on a list of n elements. The average case is assumed when the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only $n/2$ elements.

→. **Difference b/w Recursive and Iterative Algorithm.**

	Recursive Algorithm	Iterative Algorithm.
i)	Function calls itself.	A set of instructions repeatedly executed.
ii)	For functions.	For loops.
iii)	Through base case, where there will be no function call.	When the termination condition for the iterator ceases to be satisfied.
iv)	Used when code size needs to be small & time complexity is not an issue.	Used when time complexity needs to be balanced against an expanded code size.
v)	Smaller code size.	Larger code size.
vi)	Very high time complexity.	Relatively lower time complexity.

⇒ Examples of Space Complexity :

1.
```
algorithm swap (a, b)
{
    temp = a;
    a = b;
    b = temp;
}
```
1 ← 1 ← 1 ← f(n)=3

a → 1, b → 1, temp → 1, S(P) = 3

T(n) = O(1).

2.
```
def calculate(a, b, c)
    return (a+b)/c
```
a → 1, b → 1, c → 1

S(P) = 3

T(n) = O(1).

3.
```
def sum (a, n)
    s = 0
for i in range (n)
    s = s + a[i]
    return s.
```
s → 1, i → 1, n → 1, a → n

S(P) = n + 3. T(n) = O(n).

4.
```
int fact (int n)
{
    if (n == 0)
        return 1
    else
        return (n * fact (n-1));
}
```
n → 1, fact → n, S(P) = n+1, T(n) = O(n)

⇒ for each function call, 1 stack memory is used. no. of function calls is 'n'. fixed space n = 1. variable space : for each recursive call is 'n'.

5.
```
int fact (int n)
{
    int i, f = 1;
    if (n == 0)
```

```
    f = 1;
  else
    for (i = 1; i <= n; i++)
      f = f * i;
    return (f);
  }
```

$i \to 1$, $f \to 1$, $n \to 1$

return (1) → 1.

$S(P) = 4$

$T(n) = O(1)$.

Recursive :

return (n * fact (n-1))

fact (5) → 1
fact (4) → 1
fact (3) → 1
fact (2) → 1
fact (1) → 1

Iterative :

```
fact(5)
{
  return (5 * fact (4))
}.
```

⇒ Examples for Time Complexity

1.

```
Alg Sum (A, n)
{
    s = 0;          — 1
    for (i = 0; i < n; i++)   → n+1
      s = s + A[i];           → n
    return s;       → 1.
}
```

$S(P) = n+3$

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

2.

```
Alg A()
{
  for (i=0; i<n; i++)   → n+1
    printf ("Hi");      → n.
}
```

$f(n) = 2n + 1.$

$T(n) = O(n)$

3. Alg A(n, m)

{

for (i=0; i<n; i++) → n+1

for (j=0; j<m; j++) → n (m+1)

x = x+2 → n * m.

}

$f(n) = n+1 + n(m+1) + nm$

$= n+1 + nm + n + nm$

$= 2n + 2nm + 1$

$\therefore O(nm)$

$S(P) = 5.$

4. Alg add (A, B, n)

{

for (i=0; i<n; i++) → n+1

for (j=0; j<n; j++) → n * (n+1)

C[i, j] = A[i, j] + B[i, j] → n * n.

} n^2 n^2 n^2

$f(n) = n+1 + n^2 + n + n^2$

$= 2n^2 + 2n + 1$

$= O(n^2).$

$S(P) = 3n^2 + 3.$

5. Alg Multiply (A, B, n)

{

for (i=0; i<n; i++) → n+1

for (j=0; j<n; j++) → n (n+1)

{

n^2 C[i, j] = 0; → n * n.

for (k=0; k<n; k++) → n * n (n+1)

C[i,j] = C[i,j] + A[i,K] * B[K,j]; → n*(n*n)
}
}

n^2 n^2

$f(n) = n(n+1) + n^2 + n^2(n+1) + n^3$

$= 2n^3 + 3n^2 + 2n + 1$

$\therefore O(n^3)$.

$S(P) = 3n^2 + 4$.

6.
```
A()
{
for(i=0; i<n; i=i+2) → n/2 + 1
printf("Hi"); → n/2.
}
```

$f(n) = n+1$

$\therefore O(n)$

* for (i=0; i<n; i++) → n+1
  stmt; → n+1
* for (i=0; i<n; i=i*2) → log n + 1
  stmt; → log n.

$n^{1/2} = \sqrt{n}$.

for (i=0; i<n; i=i+10) → n/10 + 1
stmt; → n/10.

$f(n) = n/5 + 1$.

$\therefore O(n)$.

7.
```
for (i=0; i<n; i++) → n+1
  for(j=0; j<i; j++)
    stmt;
```

$\Rightarrow 1 + 2 + 3 + \dots + n$.

$f(n) = \frac{n(n+1)}{2} = \frac{n^2+n}{2}$

$\therefore O(n^2)$

8. `for (i=1; i<n; i++)`
`for (j=1; j<=1; j++)`
`for (k=1; k<=100; k++)`
`printf("Hi");`

i → 1 2 3 ... n
j → 1 2 3 ... n
k → 100 200 300 ... n*100

$T(n) = 100 + 200 + \dots + n*100$
$= 100(1+2+\dots+n)$
$= \frac{100(n(n+1))}{2}$
$= O(n^2)$

9. `for (i=1; i<n; i++)`
`for (j=1; j<i^2; j++)`
`for (k=1; k<=n/2; k++)`

i → 1 2 3 ... n
j → 1 4 9 ... n^2
k → $\frac{n}{2}$ $\frac{n*4}{2}$ $\frac{n*9}{2}$ ... $\frac{n*n^2}{2}$

$T(n) = \frac{n}{2} + \frac{n*4}{2} + \frac{n*9}{2} + \dots + \frac{n*n^2}{2}$

$= \frac{n}{2}(1+2^2+3^2+\dots+n^2)$

$\therefore O(n^4) = \frac{n}{2}\left[\frac{n(n+1)(2n+1)}{6}\right]$

10. `for (i=0; i*i<n; i++)`
`stmt;`

$[i^2 <= n]$ $\therefore O(\sqrt{n})$
$i = n,\ i = \sqrt{2}$

11. `p=0;`
`for (i=1; p<=n; i++)`
`p = p+i;`

i → 1 2 3 4 5 ... k
p → 1 3 6 10 15 ... $\frac{k(k+1)}{2}$

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

$\frac{k^2+k}{2} = n \Rightarrow k^2 = n$
$k = \sqrt{2}$.

$\therefore O(\sqrt{n})$.

OR

```
p=0; i=1;
while (p<=n)
{
i++;
p = p+i;
}
```

i<n	i++	O(n)
i*i<n		$O(n^2)$
i+2		$O(\sqrt{n})$

$i*3 \rightarrow 3^k = n$

12. for (i=1; i<n; i=(i*2)) $k = \log_3 n$
    stmt;

$i \rightarrow 1\ 2\ 4\ 8\ 16 \ldots$
stmt $\rightarrow 2^0\ 2^1\ 2^2\ 2^3\ 2^4 \ldots 2^k$
$T(n) = 2^k = n$
$k = \log_2 n$

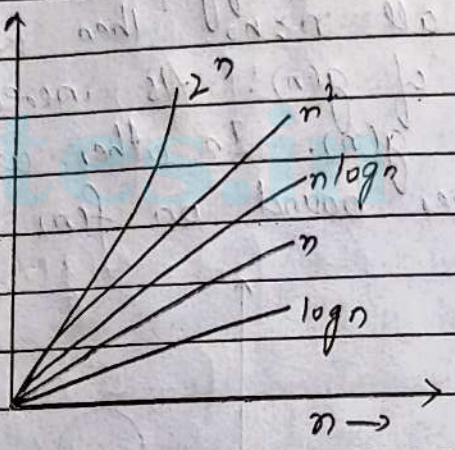
13. for (i=n; i>1; i=i/2)
    stmt;

$n \rightarrow 2^{1^2}\ 4^{2^2}\ 8^{2^3}\ 16^{2^4}\ 32^{2^5} \ldots n$
$1\ 2\ 4\ 8$
$1\ 2\ 4$
$1\ 2$
1

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

Various Time Complexities

Linear → O(n)
Quadratic → $O(n^2)$
Exponential → $O(x^n)$
Polynomial → $O(n^x)$
logrithmic → O(log n)
Constant → O(1)

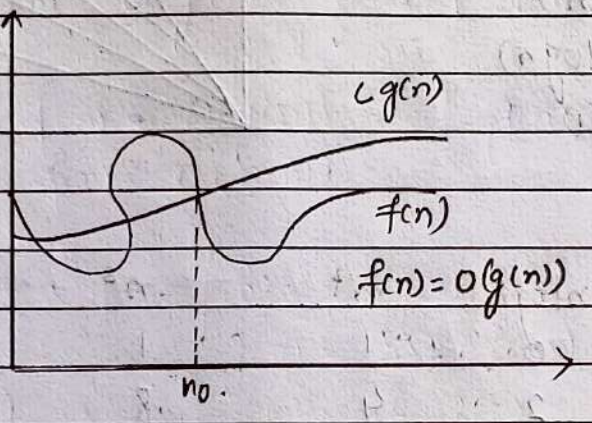


For- Values

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294963

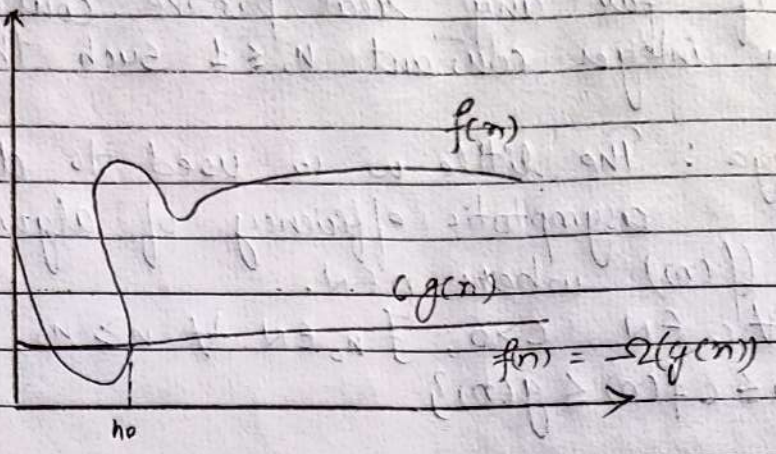
→. **Asymptotic Notation:** Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting values.
Complexity of an algorithm is usually a function of n. Commonly used asymptotic notations to calculate the running time complexity of an algorithm.

I. **Big-Oh:** $f(n) = O(g(n))$, if and only if there are two positive constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for all $n \geq n_0$. If $f(n)$ is non-negative we can simplify the last condition to $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. Then we can say that $f(n)$ is big O of $g(n)$. As increases, $f(n)$ grows no faster than $g(n)$. In other words, $g(n)$ is an asymptotic upper bound on $f(n)$.

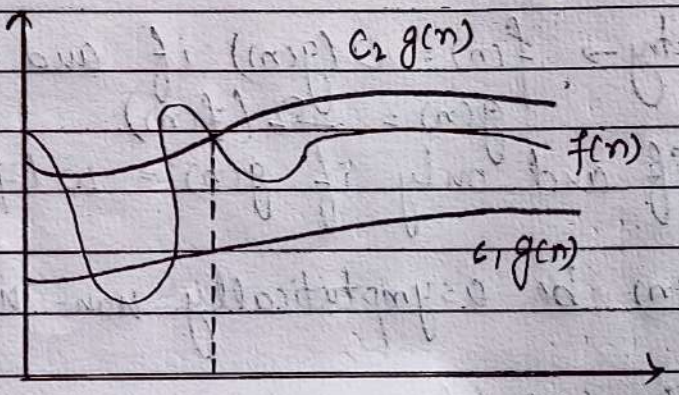


II. **Big Ω:** $f(n) = \Omega(g(n))$, if and only if there are two positive constants c and n_0 such that $|f(n)| \geq c|g(n)|$ for all $n \geq n_0$. If $f(n)$ is non-negative, we can simplify the last condition to $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$, then we can say that "$f(n)$ is omega of $g(n)$". As n increases.

$f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an asymptotic lower bound on $f(n)$.



III. Big-Θ: $f(n) = \Theta(g(n))$, if and only if there are three positive constants c_1, c_2 and n_0 such that $c_1 |g(n)| \leq f(n) \leq c_2 |g(n)|$ for all $n \geq n_0$. If $f(n)$ is non-negative, we can simplify the last condition to $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$. Then we can say that "$f(n)$ is theta of $g(n)$". As n increases, $f(n)$ grows at the same rate as $g(n)$. In other words, $g(n)$ is an asymptotically tight bound on $f(n)$.



IV. Little o: It is used to describe an upper bound that cannot be tight. In other words, loose upper bound of $f(n)$.

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 $n_0 \geq 1$ such that $f(n) > 0$.

V. Little Omega : The little ω is used to describe the asymptotic efficiency of algorithms. It is written $\omega(f(n))$ where $n \in N$.

$\{g(n) : \forall c \in N, c > 0; \exists n_0 \in N \ \forall n \geq n_0, 0 \leq c f(n) \leq g(n)\}$.

Properties of Asymptotic Notation :

I. Reflexivity → If $f(n)$ is $O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n)$ is $\Theta(g(n))$.

II. Symmetry → $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

III. Transitivity → $f(n) = O(g(n))$ and $g(n) = O(h(n))$, $f(n) = O(h(n))$.

IV. Transpose Symmetry → $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

$f(n) = O(g(n))$ if and only if $g(n) = \omega(f(n))$.

V. Let $f(n)$ and $g(n)$ be asymptotically non-negative functions. Then

$\max(f(n), g(n)) = \Theta(f(n) + g(n))$

VI If $f(n)$ is $O(g(n))$ then $h(n) * f(n)$ is $O(h(n) * g(n))$.

VII. If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then

$f(n) + g(n) = O[\max(g(n), h(n))]$

$f(n) * g(n) = O(g(n) * h(n))$.

→. **Recursive Functions:** A recursive function can be defined as a routine that calls itself directly or indirectly.
Mathematically, recursion helps to solve a few puzzles easily.

One critical requirement of recursive functions is the termination point or base case. Every recursive progr must have a base case to make sure that the function will terminate. Missing base case results in unexpected behaviour.

```
int fact (int n)
{
  if (n==0)  ⟶ n+1
  return (1)  ⟶ 1
else
  return (n * fact (n-1))  ⟶ 2n
}
```

S(P) = n+2
T(P) = O(n) → 3n+2

*. Sum of natural no.

```
int Sum (int n)
{ if (n==1)  ⟶ n
  return (1)  ⟶ 1
else
  return (n + sum(n-1))  ⟶ 2n
}
```

S(P) = n+2
T(n) = 3n+1.

*. Recursive Function for fibonacci series and gcd, also find S(P) and T(P).

```
int fib (int a, int b, int n)
{
  int (n == 1)
  return a;
  else if (n == 2)
  return b;
  else
  return fib (a, b, n-1) + fib (a, b, n-2);
}
```

→. Recurrence Relation: 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.

There are four methods for solving recurrence relation:

I. Substitution Method: The substitution method consists of two main steps:

a) Guess the solution.
b) Use the mathematical induction to find the boundary condition and shows that the guess is correct.

Eg 1.
```
void fun (int n)
{
  if (n>0)  ⟶ 1
  {
  printf ("Hi");  ⟶ 1
```

fun(n-1); → T(n-1)

}

} T(n) = T(n-1) + 2.

Recurrence Relation: $T(n) = T(n-1) + 2$ — ④.

Put n = n-1. in ④.

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

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

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

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

Put n = n-2. in ④.

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

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

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

$T(n) = T(n-3) + 3*2$.

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

$= T(0) + 2n$

$= 1 + 2n$.

$= O(n)$.

2. int fun (int n)

{

if (n<=1)

return n;

return 2 * fun (n-1);

}

n<=1	n>1
1	1
1	0
0	T(n-1) + 1 → for arithmetic
2	T(n-1) + 2

Recurrence Relation : $T(n) = \begin{cases} 2 & n <= 1 \\ [T(n-1)+2] & n > 1 \end{cases}$.

Put $n = n-1$.

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

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

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

for $T(n-1)$ substitute
$T(n-1-1)+2$
$T(n-1-1)+2+2$
$T(n-2)+2+2$

Put $n = n-2$.

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

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

$T(n) = T(n-3) + 2*3$.

$T(n) = T(n-K) + 2*K$.

Assume : $n-K = 1 \Rightarrow K = n-1$.

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

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

$= 2 + 2n-2$

$= 2n$

$= O(n)$.

3.
```
void Test (int n)
{
if (n>0)                    → 1
{
for(i=0; i<n; i++)          → n+1
printf ("Hi");              → n
Test (n-1)                  → T(n-1)
}
}
```

$T(n) = T(n-1) + \underbrace{2n+2}_{n}$.

Rewrence Relation: $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+2 & n>0 \end{cases}$

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

for $T(n) \rightarrow T(n-1)$

& $n-1 \rightarrow (n-1-1)$

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

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

Put eq. ② in ①.

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

So, $T(n) = [T(n-1)-1) + (n-1)] + n$

$T(n) = T(n-2) + (n-1) + n \rightarrow$ Put this in $T(n-1)+n$

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

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

$\vdots$

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

$= T(0) + 1 + 2 + 3 + \dots + (n-2)$

$= \left[1 + \frac{n(n+1)}{2}\right] \rightarrow$ sum of natural no.

$= O(n^2)$.

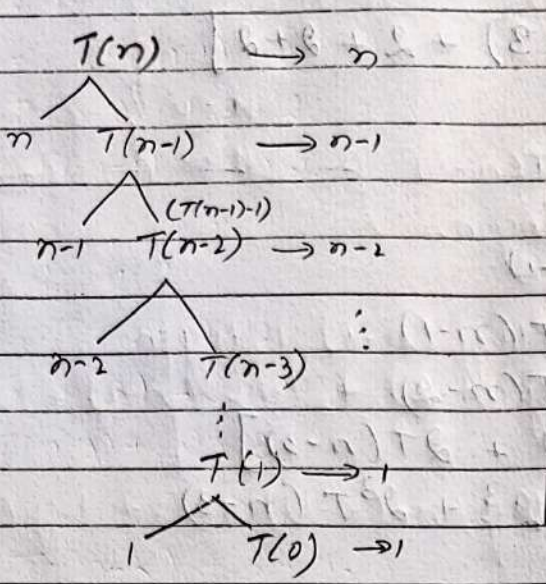
II. Recursion Tree Method: It is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

In this, each root and child represents the cost of a single subproblem.

We sum the costs with each of the levels of the tree to obtain a set of per-level costs and then sum per-level costs to determine the total cost of all level of the recursion.

A recursion tree is best used to generate a good guess, which can be verified by the substitution method.

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



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

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

$= O(n^2)$.

4.

```
int fun (int n)
{
  if (n<=1)
    return n;
  return fun(n-1) + fun(n-1);
}
```

	$n <= 1$	$n > 1$
if	1	1
return n;	1	0
return fun(n-1) + fun(n-1);	0	$T(n-1) + T(n-1) + 1$
	2	$2 * T(n-1) + 2$

Recursion Relation: $T(n) = \begin{cases} 2 & n <= 1 \\ 2T(n-1) + 2 & n > 1 \end{cases}$

$T(n) = 2T(n-1) + 2$ — (*) doubt.

Put $n = n-1$

$T(n-1) = 2[2T(n-1-1) + 2]$ — (1).

Put (1) in (*).

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

$T(n) = 2[2T(n-2) + 2 + 2]$ — (2).

Put (3) in (4).

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

doubt.

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

$\vdots$

$T(n) = 2^k$

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

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

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

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

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

$\vdots$

$= 2 + 2^2 + 2^3 + \cdots + 2^k + 2^k T(n-k)$

Assume $n-k=1 \Rightarrow k = n-1$.

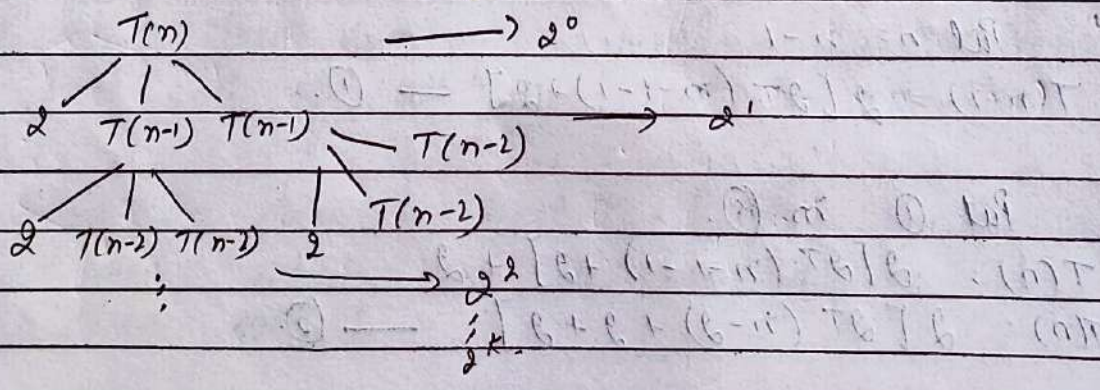
$T(n) = 2 + 2^2 + 2^3 + \cdots + 2^k + 2^k T(n-(n-1))$

$= 2 + 2^2 + 2^3 + \cdots + 2^k + 2^k \cdot 2$

$= 2 + 2^2 + 2^3 + \cdots + 2^k + 2^{k+1}$.

$G(P) = \frac{a(r^n-1)}{r-1} = 2[1 + 2^1 + 2^2 + \cdots + 2^{k-1} + 2^k]$

$= 2\left[\frac{2^k-1}{2-1}\right] = 2(2^k-1) = O(2^n)$



$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^k = \frac{2^k - 1}{2-1} = 2^k - 1 \Rightarrow O(2^n)$.

5. $T(n) = T(n-1) + n^4$.

$n = n-1$.

$T(n-1) = [T(n-2) + (n-1)^4] + n^4$

$T(n-2) = T(n-3) + (n-2)^4 + (n-1)^4 + n^4$

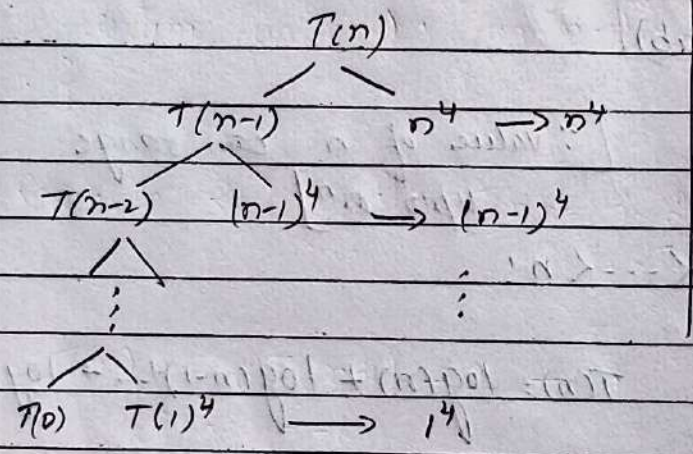
$T(n-K) = T(n-K) + (n-(K-1))^4 + \cdots + (n-1)^4 + n^4$

Assume $n-K=0$; $K=n$.

$T(n) = T(n-n) + 1^4 + 2^4 + \cdots + (n-1)^4 + n^4$

$= T(0) + \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$

$T(n) = O(n^5)$



$T(n) = n^4 + (n-1)^4 + \cdots + 1^4$

$= \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$

$= O(n^5)$.

8. void Test (int n)
{
  if (n > 0) ⟶ 1
  {
    for(i=1; i<n; i = i*2) ⟶ log(n+1)

printf("Hi"); → log n
Test(n-1); → T(n-1).
}
}

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

$T(n) = T(n-1) + \log n.$

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

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

$= T(n-K) + \log(n-(K-1)) + \cdots + \log(n-1) + \log n$

Assume $n-K = 0$; $K = n$.

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

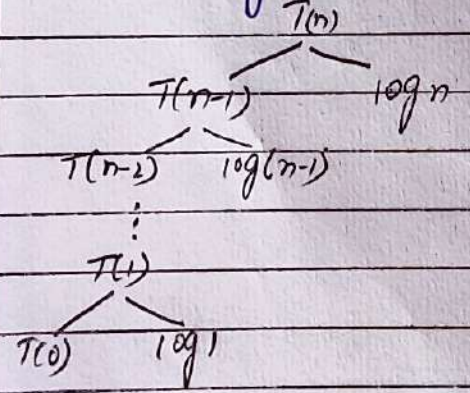
$= T(0) + \log(1*2*\cdots*(n-1)*n).$

$\boxed{\log a + \log b = \log(ab)}$

$= 1 + \log(n!)$ [∵ value of n can range upto n^n]

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

$n < n\log n < n^2 < n^3 < \cdots < n!$



$T(n) = \log(n) + \log(n-1) + \cdots + \log 1.$

$T(n) = \log(n*n-1*n\cdots*1)$

$= \log(n!)$

$= O(n \log n).$

7. Fibonacci Series.

$$T(n) = \begin{cases} 3 & n <= 1 \\ 3 + T(n-1) + T(n-1) & n > 1 \end{cases}$$

```
Alg fib(n)
{
if (n==0 || n==1)  ⟶ 2.
  return n;  ⟶ 1.
return fib(n-1) + fib(n-2);  ⟶ T(n-1) + T(n-2) + 1.
}
```

* **Dividing Functions :**

1. $T(n) = 2T\left(\frac{n}{2}\right) + n \; ; \quad T(1) = 1$

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

$= 2 \times 2T\left(\frac{n}{4}\right) + \left(2 \times \frac{n}{2}\right) + n$

$= 2^2 T\left(\frac{n}{4}\right) + n + n$

$= 2^2 T\left(\frac{n}{4}\right) + 2n$

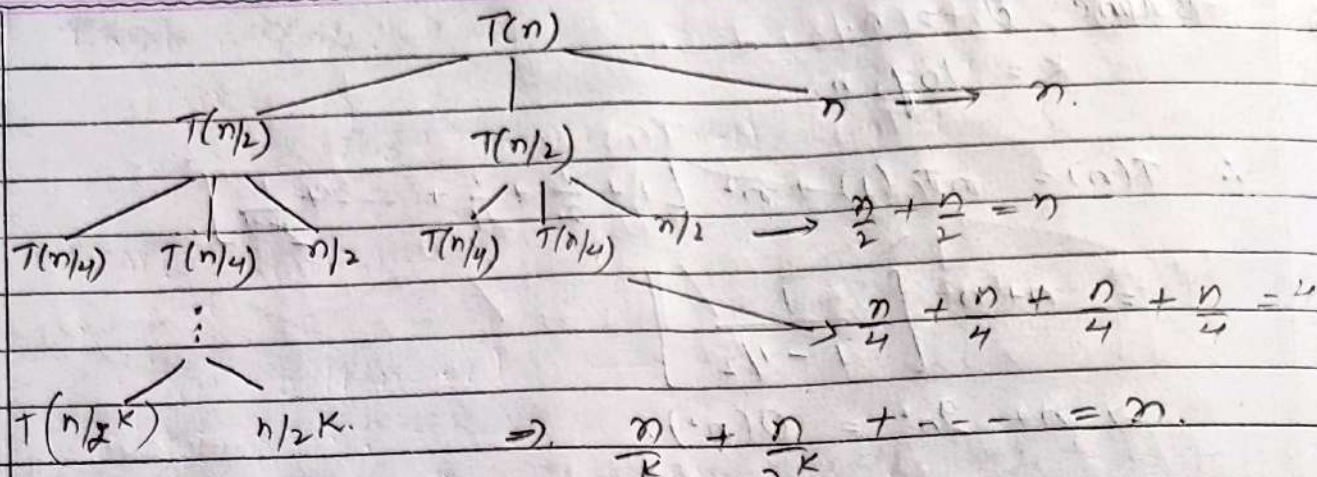
$T\left(\frac{n}{4}\right) = 2^2\left(2T\left(\frac{n}{8}\right) + \frac{n}{4}\right) + 2n$

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

$\vdots$

$= 2^k T\left(\frac{n}{2^k}\right) + kn.$

```
void Test (int n)
{
if (n>1)
  for(i=0; i<n; i++)
printf("Hi");
Test (n/2);
Test (n/2);
}}
```



$T(n) = n + n + \dots + n.$

$= Kn.$

Put $K = \log_2 n.$

$\therefore T(n) = n \log_2 n.$

$= O(n \log n).$

$\left[\frac{n^2}{2^3} + \frac{n^2}{2^2} + \frac{n^2}{2^1} + \frac{n^2}{2^0} \right]$

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

$= 2\left[2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right] + n^2.$

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

$= 2^2\left[2T\left(\frac{n}{2^3}\right) + \left(\frac{n}{4}\right)^2\right] + \frac{n^2}{2} + n^2.$

$= 2^3 T\left(\frac{n}{2^3}\right) + 2^2 + \left(\frac{n}{4}\right)^2 + \frac{n^2}{2} + n^2.$

$\vdots$

$T(n) = 2^k T\left(n/2^k\right) + \frac{n^2}{2^{k-1}} + \dots + \frac{n^2}{2} + n^2.$

Assume $2^k = n$.

$k = \log_2 n$.

$\therefore T(n) = nT(1) + n^2\left[1 + \frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^{k-1}}\right]$.

$= n + n^2\left[\frac{1}{1 - 1/2}\right]$

$\Rightarrow n + 2n^2 = O(n^2)$.

3. $T(n) = 3T\left(\frac{n}{4}\right) + n$.

$= 3\left[3T\left(\frac{n}{16}\right) + \frac{n}{4}\right] + n$.

$= 3^2 T\left(\frac{n}{4^2}\right) + \frac{3n}{4} + n$.

$= 3^2\left[3T\left(\frac{n}{4^3}\right) + \left(\frac{n}{4^2}\right)\right] + \frac{3n}{4} + n$.

$= 3^3 T\left(n/4^3\right) + 3^2 n/4^2 + 3n/4 + n$.

$= 3^3 T\left(\frac{n}{4^3}\right) + n\left(\frac{3^2}{4^2} + \frac{3}{4} + 1\right)$

$T(n) = 3^k T\left(n/4^k\right) + n\left[\frac{3^{k-1}}{4^{k-1}} + \cdots + \frac{3^2}{4^2} + \frac{3}{4} + 1\right]$

$= 3\log_4 n + 4n$.

Assume $4^k = n = O(n)$

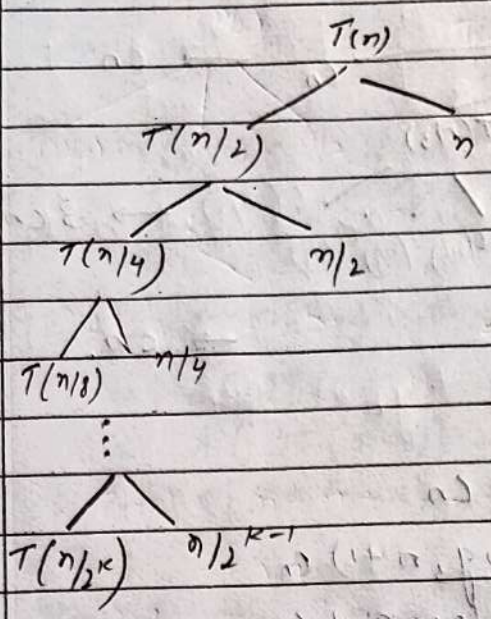
$k = \log_4 n$.

$a = 1, r = 3/4$

GP $= \frac{a}{1-r}$

$= \frac{1}{1 - 3/4} = \frac{1}{1/4} = 4$

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

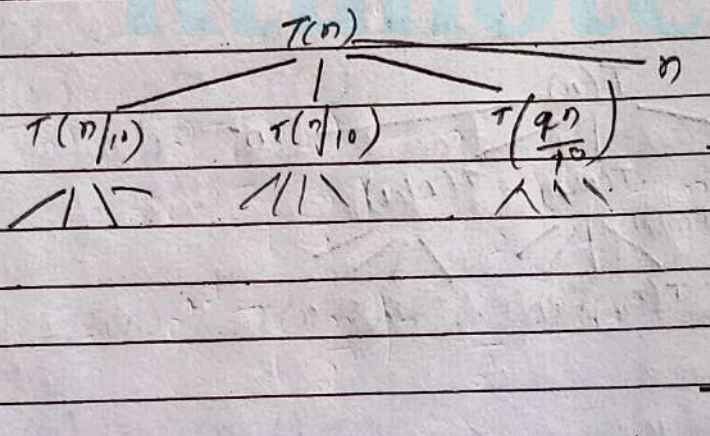


$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}}$$

$$= n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}}\right]$$

$$\Rightarrow n[1+1] = 2n = O(n)$$

5. $T(n) = 2T\left(\frac{n}{2}\right) + T\left(\frac{9n}{10}\right) + n$.



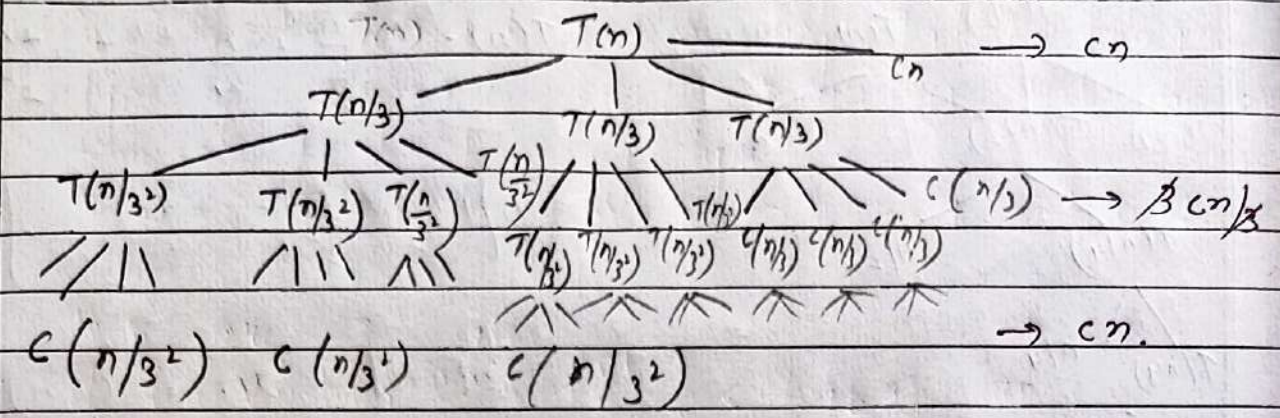
$$T(n) = n + \left(\frac{11}{10}\right) + n\left(\frac{11}{10}\right)^2 + \cdots$$

$$= n\left[1 + \frac{11}{10} + \left(\frac{11}{10}\right)^2 + \cdots\right]$$

$$= n\left[\frac{1}{1 - 11/10}\right]$$

$$= 10(n) \Rightarrow O(n)$$

6. $T(n) = 3T(n/3) + cn$.

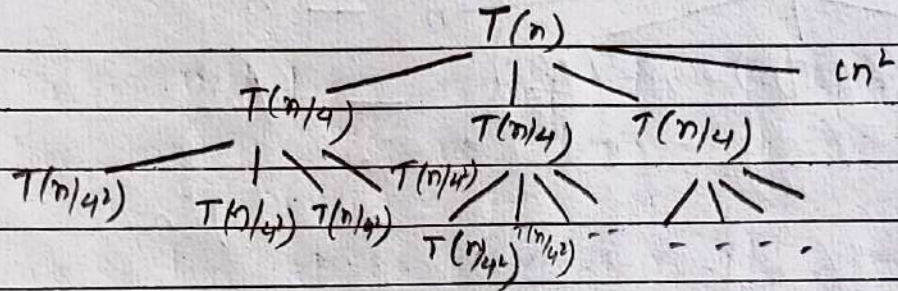


$T(n) = (k+1)c$ | $= (k+1)cn$

$= O(n \log_3 n)$ | $= (\log_3 n + 1)cn$

$= cn \log_3 n + cn$.

7. $T(n) = 3T(n/4) + cn^2$.



III. **Iteration Method:** It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Eg. 1 $T(n) = 1 \quad \text{if } n = 1$
$= 2T(n-1) \quad \text{if } n > 1.$

$T(n) = 2T(n-1)$
$= 2[2T(n-2)] = 2^2 T(n-2)$
$= 4[2T(n-3)] = 2^3 T(n-3)$
$= 8[2T(n-4)] = 2^4 T(n-4). \quad -①.$

Repeat the procedure for i times.

$T(n) = 2^i T(n-i)$
Put $n-i = 1$ or $i = n-1$ in ①.
$T(n) = 2^{n-1} T(1)$
$= 2^{n-1} \cdot 1 \quad \{T(1) = 1 \text{ ... given}\}$
$= 2^{n-1}$

Eg 2. $T(n) = T(n-1) + 1$ and $T(1) = \Theta(1)$

$T(n) = T(n-1) + 1$
$= (T(n-2)+1)+1) = (T(n-3)+1)+1+1$
$= T(n-4) + 4 = T(n-5) + 1 + 4$
$= T(n-5) + 5 = T(n-K) + K$

where $K = n-1$
$T(n-K) = T(1) = \Theta(1)$
$T(n) = \Theta(1) + (n-1) = 1 + n - 1 = n = \Theta(n)$

IV. Master's Theorem:

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type:

$T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$.

There are 3 cases:

i. if $f(n) = O(n^c)$ where $c < \log_b a$ Then
$T(n) = \Theta(n^{\log_b a})$

ii) if $f(n) = \Theta(n^c)$ where $c = \log_b a$ Then
$T(n) = \Theta(n^c \log n)$

iii) if $f(n) = \Omega(n^c)$ where $c > \log_b a$ Then
$T(n) = \Theta(f(n))$

Master method is mainly derived from recurrence tree method. If we draw recurrence tree of $T(n) = aT(n/b) + f(n)$, we can see that the work done at root is $f(n)$ and work done at all leaves is $\Theta(n^c)$ where c is $\log_b a$. And the height of recurrence tree is $\log_b n$.

Master's Theorem for ~~Dividing~~ Decreasing Function:

Used to directly calculate the time complexity function of 'decreasing' recurrence relations of the form:

$T(n) = aT(n-b) + f(n)$.
$f(n) = O(n^k)$; $k \geq 0$

The value of 'a' decides the time complexity function for the 'decreasing' recurrence relation

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

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

Where n = input size or the size of the problem.

a = count of subproblems in the decreasing recursive fn.

$n-b$ = size of each subproblem

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

$a > 0, \quad b > 0, \quad K >= 0.$

Case I: If $a = 1$; $T(n) = O(n^{K+1})$ OR $O(n^k \cdot n) = O(n \cdot f(n))$

Case II: If $a > 1$; $T(n) = O(n^K \, a^{n/b})$

Case III: If $a < 1$; $T(n) = O(n^K)$

Examples: I. $T(n) = T(n-1) + 1 \longrightarrow n^K = K = 0$

$O(n)$.

II. $T(n) = T(n-1) + n$

$n^K \Rightarrow K = 1, \; b = 1$

$O(n^2)$

III. $T(n) = T(n-1) + \log n$

$a = 1, \; b = 1, \; f(n) = \log n$

$O(n \log n)$

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

V. $T(n) = T(n-1) + n^4$ $O(n^5)$

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

$a = 2, \quad b = 1, \quad f(n) = 1$

$O(n^0 2^{n/1}) = O(2^n)$.

VII. $T(n) = T(n-2) + 1$ $O(n)$

VIII. $T(n) = T(n-100) + n$ $O(n^2)$

IX. $T(n) = 3T(n-1) + 1$

$a = 3, \quad b = 1, \quad f(n) = 1$

$O(n^0 3^{n/1}) = O(3^n)$

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

$a = 2, \quad b = 1, \quad f(n) = n$

$O(n \cdot 2^{n/1}) = O(n \cdot 2^n)$.

XI. $T(n) = 3T(n-2) + 1$ $O(3^{n/2})$.

Master's Theorem for Dividing Functions

To calculate $\log_b a$ and compare it with $f(n)$ to decide the final time complexity of recurrence relation $T(n)$.

The idea behind Master's algorithm is based on the computation of $n^{\log_b a}$ and comparing it $f(n)$. The time complexity function comes out to be the one over-riding the other.

$T(n) = aT(n/2) + f(n)$; $a \geq 1$, $b > 1$

$f(n) = \Theta(n^k \log^p n)$: $k \geq 0$

$p \rightarrow$ real no.

* $T(n) = aT\left(\frac{n}{b}\right) + f(n)$; $a \geq 1$, $b \geq 1$.

$f(n) = \Theta(n^k \log^p n)$, $k \geq 0$. $p \rightarrow$ real no.

Case I: $\log_b a > k$ OR $a > b^k$ =

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

Case II: $\log_b a = k$ OR $a = b^k$

i) $p > -1$; $T(n) = \Theta(n^k \log^{p+1} n)$

ii) $p = -1$; $T(n) = \Theta(n^k \log(\log n))$

iii) $p < -1$; $T(n) = \Theta(n^k)$

Case III: $\log_b a < k$ OR $a < b^k$.

i) $p \geq 0$; $T(n) = \Theta(n^k \log^p n)$

ii) $p < 0$; $T(n) = \Theta(n^k)$.

Examples:

Case 1: I. $T(n) = 2T(n/2) + 1 \rightarrow \Theta(n)$

II. $T(n) = 4T(n/2) + 1 \rightarrow \Theta(n^2)$

III. $T(n) = 4T(n/2) + n \rightarrow \Theta(n^2)$

IV. $T(n) = 2T(n/2) + n^2 \rightarrow \Theta(n^3)$

V. $T(n) = 16T(n/2) + n^2 \rightarrow \Theta(n^4)$

Case III : I. $T(n) = T(n/2) + n \rightarrow \Theta(n)$

II. $T(n) = 2T(n/2) + n^2 \rightarrow \Theta(n^2)$

III. $T(n) = 2T(n/2) + n^2 \log n \rightarrow \Theta(n^2 \log n)$

IV. $T(n) = 4T(n/2) + n^3 \log^2 n \rightarrow \Theta(n^3 \log^2 n)$

V. $T(n) = 2T(n/2) + n^2 / \log n \rightarrow \Theta(n^2)$

Case II : I $T(n) = T(n/2) + 1$ $\Theta(\log n)$

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

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

$T(n) = 4T(n/2) + n^2$ $\Theta(n^2 \log n)$

$T(n) = 2T(n/2) + (n \log n)^2$ $\Theta(n^2 \log^3 n)$

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

$T(n) = 2T\left(\frac{n}{2}\right) + n / \log^2 n$ $\Theta(n)$

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

1. $T(n) = 7T(n/2) + n^2$

$a = 7, \quad b = 2, \quad K = 2.$
$P = 0.$

$f(n) = \Theta(n^K \log^P n)$
$a > b^K$
$a < b^K$
$a = b^K.$

$T(n) = \Theta(n^{\log_b a})$
$= \Theta(n^{\log_2 7})$

2. $T(n) = 2T(n/2) + n \log n$ ($K=1$, $P=K$)

$a = 2, \quad b = 2$

$a = b^K \Rightarrow 2 = 2$
$\because a = b^K$ and $P > -1$

$T(n) = \Theta(n^K \log^{P+1} n)$
$= \Theta(n \log^2 n)$

3. $T(n) = 2T(n/4) + \sqrt{n}$
$a = 2, \quad b = 4, \quad K = 1/2, \quad P = 0$

$a < b, \quad P > -1.$
$T(n) = \Theta(\sqrt{n} \log n)$

4. $T(n) = 2T(n/2) + n/\log n$
$a = 2, \quad b = 2, \quad K = 1, \quad P = -1$

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

5. $T(n) = 3T(n/2) + n^2$
$a = 3, \quad b = 2, \quad K = 2, \quad P = 0$

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