Algorithm specification :-

Algorithm can be described in many ways.

(i) Natural language like English.

(ii) Graphic representation called flowcharts.

(iii) Pseudocode that resembles C.

Representation ⇒

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces { and }

   A compound statement can be represented as a block.

   The body of a procedure also forms a block.

   Statements are delimited by ;
3. An identifier begins with a letter. compound data types (integer, float, char, boolean) can be formed with records.

Example.

```
node = record
{ datatype-1 data1;
        ⋮
  datatype-n datan;
  node *link;
}
```

link ⇒ pointer to the record type node.

Individual data items of a record can be accessed with →

4. Assignment of values to variable is done using the assignment statement. <variable> := <expression> ;
5. There are two boolean values true and false for the logical operators and, or, and not and relational operators <, <=, =, ≠, ≥, and >.
6. Multidimensional array representation [and]. Example A[i, j]
7. Following looping statements are employed. for, while & repeat-until

```
while (condition) do
{
   (statement 1)
        ⋮
   (statement n)
}
```

```
for variable := value1 to value2
{
   (statement 1)
        ⋮
   (statement n)
}
```

```
repeat
   (statement 1)
        ⋮
   (statement n)
until (condition)
```

8. A Conditional statement,

if (condition) then (statement)

if (condition) then (statement1) else (statement2)

case statement,

```
case
{
  :(condition 1): (statement 1)
        ⋮
  :(condition n): (statement n)
  :else: (statement n+1)
}
```

9. Input ⇒ read, Output ⇒ write.

10. An algorithm consists of a heading and a body. The heading takes of the form. Algorithm Name (<parameter list>)

Example: To find and return the maximum of n given numbers,

```
Algorithm Max (A, n)
{
    Result := A[1];
    for i=2 to n do
        if A[i] > Result then Result := A[i];
    return Result;
}
```

Algorithm specification :-

Algorithm can be described in many ways.

(i) Natural language like English.

(ii) Graphic representation called flowcharts.

(iii) pseudocode that resembles C.

Representation =>

1. Comments begin with // and continue until the end of line.
2. Blocks are indicated with matching braces { and }

   A compound statement can be represented as a block.

   The body of a procedure also forms a block.

   Statements are delimited by ;
3. An identifier begins with a letter. compound data types (integer, float, char, boolean) can be formed with records.

Example.

```
node = record
{ datatype-1  data1;
        ⋮
  datatype-n  datan;
  node * link;
}
```

link => points to the record type node.

Individual data items of a record can be accessed with →

4. Assignment of values to variable is done using the assignment statement. <variable> := <expression>;
5. There are two boolean values true and false for the logical operators and, or, and not and relational operators <, <=, =, ≠, ≥, and >.
6. Multidimensional array representation [and]. Example A[i, j]
7. Following looping statements are employed. for, while & repeat-until

```
while (condition) do
{
  (statement 1)
      ⋮
  (statement n)
}
```

```
for variable := value1 to value2
{
  (statement 1)
      ⋮
  (statement n)
}
```

```
repeat
  (statement 1)
      ⋮
  (statement n)
until (condition)
```

8. A conditional statement,

if (condition) then (statement)

if (condition) then (statement1) else (statement2)

case statement,

```
case
{
   :(condition 1): (statement 1)
                 .
                 .
                 .
   :(condition n): (statement n)
   :else: (statement n+1)
}
```

9. Input ⇒ read, Output ⇒ write.

10. An algorithm consists of a heading and a body. The heading takes of the form. Algorithm Name (<parameter list>)

Example: To find and return the maximum of n given numbers.

```
Algorithm Max (A, n)
{
   Result := A[1];
   for i:=2 to n do
       if A[i] > Result then Result := A[i];
   return Result;
}
```

Step count : (line count / operation count)

The idea is to count the number of instructions or steps that are used by a given algorithm to perform the given tasks. The first step is to find steps per execution and frequency (or number of times) of statements.

For, non executable statements - no of steps for execution is 0.

Executable Statements . " " " is 1.

Specific ways of measuring step counts per execution are as follows

1. Declarative Statements with no initializations have a Step

2. comments, braces such as begin/ end/ endif/ while/ end for case all having a step count of 0.

3. Expressions have a step count of 1.

4. Assignment statement, function invocation, return statements and other statements such as break/ continue/ go to all have a step count of 1.

Frequency – no of times an instruction is executed.

Algorithm:

```
Algorithm simple (A,B,C)
Begin
    A = B+1
    C = A+2
    D = A+B
end.
```

Step no	program	steps per execution	Frequency	Total
1	Algorithm Simple(A,B,C)	0	–	–
2	Begin	0	–	–
3	A = B+1	1	1	1
4	C = A+2	1	1	1
5	D = A+B	1	1	1
6	End	0	–	–
		Total		3

```
Algorithm sum()
Begin
    Sum = 0.0          — 1
    for i = 1 to n do  — n+1
      Sum = Sum + 1    — n
    End for            — 0
  Return sum           — 1
end.                   — 0
```

Step count

$T(n) = 2n+3$ => Time complexity

Operation count:

The idea is to count the no. of operation instead of step for algorithm analysis.

operations ⇒ 1) Elementary (basic) 2) Non-elementary (non-basic)

Basic operations:

1) Assignment
2) Comparison
3) Arithmetic
4) Logical

Non-basic operations
- Sorting
- Finding maximum or minimum of an array.

steps required for performing an operation count is follows,

1. Count the no. of basic operations of a program and express it as a formula.
2. Simplify the formula.
3. Represent time complexity as a function of operation count.

Ex:1) Algorithm Swap(a,b) — 0
Begin — 0
temp=a — c_1
a=b — c_2
b=a — c_3
end — 0

$T(n) = c_1 + c_2 + c_3$

Analysis as Dominant Operation

Counting of all the operation of an entire algorithm may be a difficult task. It would be much simpler if the count of only the dominant operation taken into account.

2) Algorithm Sample() — 0.
begin — 0
for k=1 to n do — $(n+1)*c_1$
A = A x k. — $n*c_2$
end for — 0
end — 0

$= (n+1)*c_1 + n*c_2$

Ex: for i=1 to n do
for j=1 to n do
A = B x C
End for
End for

Sln $count = \sum_{i=1}^{n} \sum_{j=1}^{n} 1$

$= \sum_{i=1}^{n} n^2$

Line count, operation count.

To count the number of operations is also known as to analyze the algorithm complexity. The idea to have a rough idea how many operation are in the worst case needed to execute the algorithm on an input of size n, which give the upperbound of the computational resources required for that algorithm.

Example:

S1:	prod ← 0	F	1
S2:	for i ← 0 to n-1 do (for(i=0;i++))		inital - 1, inital/incre } 2×n
S3:	prod ← prod + A[i] * B[i]		5 × n
S4:	return prod.		1

(+1) Line count

⇒ Line 1 is one operation (Assigning value) = 1

⇒ Loop initializing is one op (Assigning a value) = 1

⇒ Line 3 is five op per iteration (Mul, add, 2 array ref, assign) = 5n

⇒ Loop incrementation is two ops (Add & Assign) = 2n

⇒ Loop termination test is one op (i<n) each time, (ie) done n+1 time (n success, 1 failure) = n+1

⇒ Return is one operation = 1

Total = 1+1+5n+2n+(n+1)+1

Total = 8n+4

Algorithm Design Technic (Approaches)

Design of an algorithm.

1. understanding the problem.
2. Decision making on.
   - capabilities of Computational devices.
   - choices of either exact or approximate problem solving method.
   - Data structure.
   - Algorithmic strategies.

3. Specification of algorithm.
4. Algorithmic verification.
5. Analysis of algorithm.
6. Implementation or coding of algorithm.

Algorithm design Techniques:-

1. Bruteforce - Straightforward & native approach.
2. Divide and conquer - problem is divided into smaller instances.
3. Dynamic programming - The results of smaller recurring instances are obtained to solve the problem. (subproblems)

0/1

4. Greedy technique - To solve the problem locally (optimal) decisions made.

Knapsack

5. Backtracking - This method is based on the trail and error. If we want to solve the problem the desired solution is chosen from the finite set S.

Designing an algorithm and Analysis:

Analysis of algorithm help us determine whether the algorithm is useful or not. Generally an algorithm is analyzed based on its execution time (Time complexity) and amount of space (Space complexity).

Space complexity => the space complexity of an algorithm is the amount of memory it needs to completion.

Time complexity => the time complexity of an algorithm is the amount of computer time it needs to run to completion.

linear search

Worst, Avg
Best, Case

$W(n) = \max_{1 \le i \le K} \{T_i(K)\}$

$B(n) = \min_{1 \le i \le K} \{..\}$

$$A(n) = \sum_{i=1}^{k} P_i \times T_i(n)$$

$$= 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n}$$

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

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

Asymtotic Notations (O, Ω, Θ) based on order of growth.

⇒ The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows to compare the relative performance of algorithms.

⇒ Asymtotic - how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound.

⇒ Usually an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymtotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0, 1, 2, \ldots\}$.

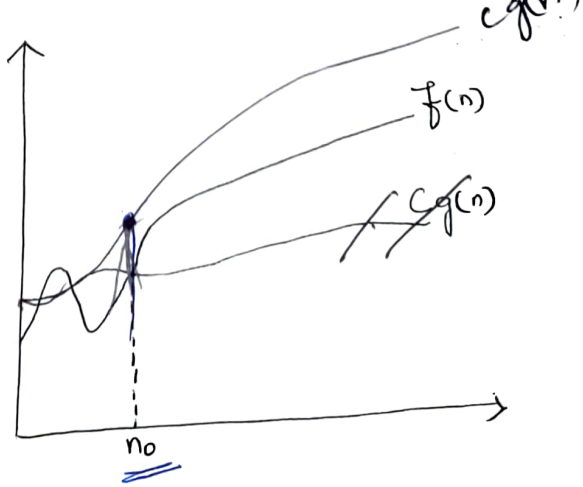
To choose the best algorithm, we need to check efficiency of each algorithm the efficiency can be measured by computing time complexity of each algorithm.

Asymtotic notation is a shorthand way to represent the time complexity.

Various notations are Ω, Θ and O (Best, Average and Worst).

① Big-oh notation (O)

The Big-oh notation is denoted by O. It is a method of representing the upper bound of algorithm's running time. Using big-oh notation we can give largest amount of time taken by the algorithm to complete.



Definition:

Let $F(n)$ and $g(n)$ be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input

$n > n_0$

Similarly c is some constant such that $c>0$, we can write

$$F(n) \leq c * g(n)$$

F(n) is big oh of g(n). It is also denoted as $F(n) \in O(g(n))$. (ie) F(n) is the less than g(n) if g(n) is multiple of some constant c.

Example: consider the function $F(n) = 2n+2$ and $g(n) = n^2$.

Find constant c, so that $F(n) \leq c\, g(n)$.

If $n=1$ $\quad F(n) = 2n+2 = 2(1)+2$ $\boxed{F(n)=4}$

$g(n) = n^2 = (1)^2$ $\boxed{g(n)=1}$

(ie) $F(n) > g(n)$.

If $n=2$ $\quad F(n) = 2(2)+2 = 4+2$ $\boxed{F(n)=6}$

$g(n) = n^2 = (2)^2$ $\boxed{g(n)=4}$

(ie) $F(n) > g(n)$

If $n=3$ $\quad F(n) = 2(3)+2 = 6+2$ $\boxed{F(n)=8}$

$g(n) = n^2 = (3)^2$ $\boxed{g(n)=9}$

(ie) $F(n) < g(n)$ is true.

Hence we conclude that for $n \geq 2$, we obtain

$$F(n) < g(n).$$ //

(2) Big-Omega notation. (Ω)

Used to represent the lower bound of algorithm's running time. Using this we can denote shortest amount of time taken by algorithm.

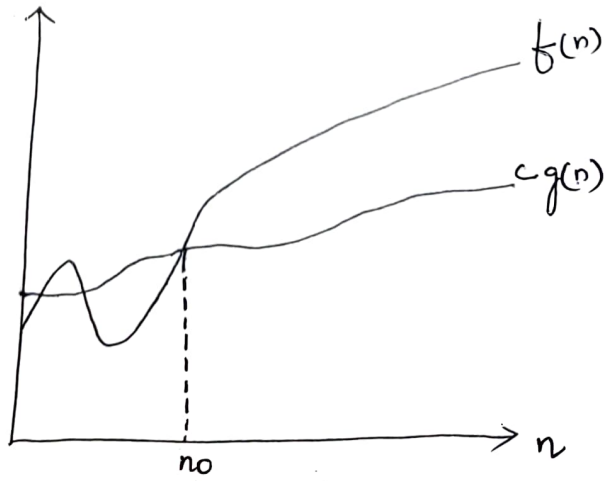
Definition

A function F(n) is said to be in $\Omega(g(n))$ if F(n) is bounded below by some positive constant multiple of g(n) such that

$$\boxed{F(n) \geq c * g(n)}$$

For all $n \geq n_0$

It is denoted as $F(n) \in \Omega(g(n))$.



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

Example: Consider $F(n) = 2n^2 + 5$ and $g(n) = 7n$

$n=0$ $\quad F(n) = 2(0)^2 + 5 \Rightarrow \boxed{F(n) = 5}$

$g(n) \leq 7(0) \Rightarrow \boxed{g(n) = 0}$

(ie) $F(n) > g(n)$

$n=1$ $\quad F(n) = 2(1)^2 + 5 \Rightarrow \boxed{F(n) = 7}$

$g(n) = 7(1) \Rightarrow \boxed{g(n) = 7}$

(ie) $F(n) = g(n)$

$n=3$ $\quad F(n) = 2(3)^2 + 5 \Rightarrow \boxed{F(n) = 23}$

$g(n) = 7(3) \Rightarrow \boxed{g(n) = 21}$

(ie) $F(n) > g(n)$

Thus for $n > 3$ we get $F(n) > c * g(n)$

(3) Big-Theta Notation (Θ)

Running time is between upper bound and lower bound.

Definition:

Let $F(n)$ and $g(n)$ be two non-negative functions. there are two positive constant namely c_1 and c_2 such that

~~$c_1 \leq g(n) \leq c_2 g(n)$~~ $\quad c_1 g(n) \leq f(n) \leq c_2 g(n)$

Thus we can say that $F(n) \in \Theta(g(n))$ $\quad \Omega \quad \Theta$

Example :-

If $F(n) = 2n + 8$ and $g(n) = 7n$

Where $n \geq 2$

Similarly $F(n) = 2n + 8$

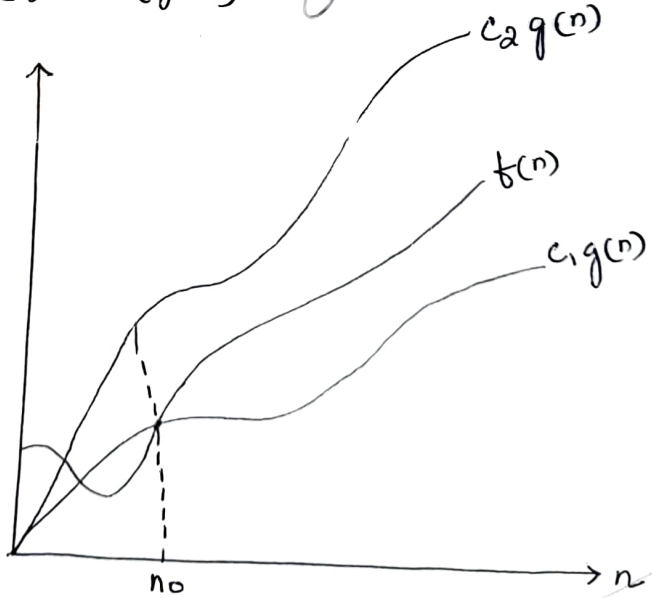
$g(n) = 7n$

(ie) $5n < 2n + 8 < 7n$ For $n \geq 2$

$c_1 = 1 \quad c_2 = 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$

Assume



$f(n) = \Theta(g(n))$ $\quad c_1 = 1 \quad c_2 = 2$

$* \quad 7 \leq 2n + 8 \leq 14$

$c_1 = 2 ; c_2 = 3$

$14 \leq 2n + 8 \leq 21$

Theta notation is more precise with big-oh and big-omega notations.

Recurrence Relation - Substitution Method

A recurrance is an equation or inequality that describes a functio in terms of its value on smaller inputs.

Example, the worst case running time $T(n)$ of the merge-sort procedure could be described by the recurrence,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2)+\Theta(n) & \text{if } n>1 \end{cases}$$, whose solution is $T(n) = O(n \log n)$.

Three methods to solve recurrences,

1. Substitution method ⇒ guess a bound and then use mathematical induction to prove the guess correct.
2. Recursion tree method ⇒ converts the recurrence into a tree whose nodes represents the cost incurred at various levels of the recursion.
3. Master method ⇒ provides bounds for recurrences of the form,

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n)$ is given function.

Substitution Method

The substitution method for solving recurrences entails two steps.

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

Substitution method can be used to establish either upper or lower bounds on a recurrence.

There are two types of substitution,

1. Forward substitution
2. Backward substitution.

Forward Substitution => This method makes use of an initial condition in the initial term and value for the next term is generated.

This process is continued until some formula is guessed.

Example: Consider a recurrence relation.

$T(n) = T(n-1) + n$ with initial condition $T(0) = 0$.

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

If $n=1$ then $T(1) = T(0) + 1$

$= 0 + 1$

$\boxed{T(1) = 1}$

If $n=2$ then $T(2) = T(1) + 2$

$= 1 + 2$

$\boxed{T(2) = 3}$

If $n=3$ then $T(3) = T(2) + 3$

$= 3 + 3$

$\boxed{T(3) = 6}$

By observing above generated equations we can derive a formula,

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

Denote $T(n)$ in terms of big-oh notation as follows,

$\boxed{T(n) = O(n^2)}$.

Backward Substitution Method.

Values are substituted recursively in order to derive some formula.

Example, consider a recurrence relation,

$T(n) = T(n-1) + n$ with initial condition $T(0) = 0$. ——①

$T(n-1) = T(n-1-1) + (n-1)$ ——②

Putting eq(2) in equation(1) we get,

$T(n) = T(n-2) + (n-1) + n$ ——③

Let,

$T(n-2) = T(n-2-1) + (n-2)$ ——④

Putting eq(4) in equation(3) we get,

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

$\vdots$

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

If $k=n$ then

$T(n) = T(0) + 1 + 2 + \cdots + n$

$T(n) = 0 + 1 + 2 + \cdots + n$

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

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

Denote $T(n)$ is terms of big-oh notation as

$T(n) \in O(n^2)$.

Problem: Solve the recurrence equation $T(n) = T(n-1)+1$ with $T(0)=0$ a initial condition. Also find big oh notation.

Solution: Let $T(n) = T(n-1)+1$

By backward substitution.

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

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

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

$$\boxed{T(n) = T(n-2)+2}$$

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

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

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

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

$$\vdots$$

$$\boxed{T(n) = T(n-k)+k} \quad \text{---(i)}$$

If $k=n$, then equ(i) becomes

$$T(n) = T(0)+n$$

$$= 0+n$$

$$\boxed{T(n)=n}$$

$T(n)$ in terms of big-oh notation as $T(n) = O(n)$.

Problem: Solve the recurrence relation.

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

Solution:

With $T(1)=1$ as initial condition.

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

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

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

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

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

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

$$= 8T\left(\frac{n}{8}\right) + 3n$$

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

$$\vdots$$

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

→

→ If we assume $2^k = n$

$k = \log_2 n$

$$T(n) = n \cdot T\left(\frac{n}{n}\right) + \log n \cdot n$$

$$= nT(1) + n \cdot \log n$$

$$= n + n\log n$$

(ie) $T(n) \simeq n\log n$.

In terms of big-oh notation.

$$\boxed{T(n) = O(n\log n)}$$

Recurrence Relation - Recursion Tree method

Recursion tree ⇒ Each node represent the cost of a single subproblem.

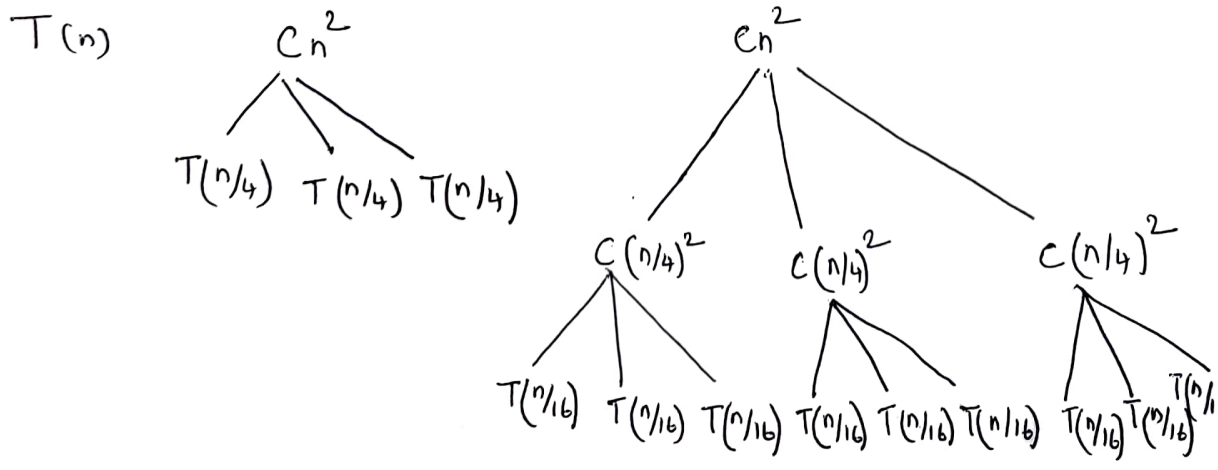
⇒ Sum the cost within each level of the tree to obtain a set of per-level costs and then sum all the per-level costs to determine the total cost of all levels of the recursion.

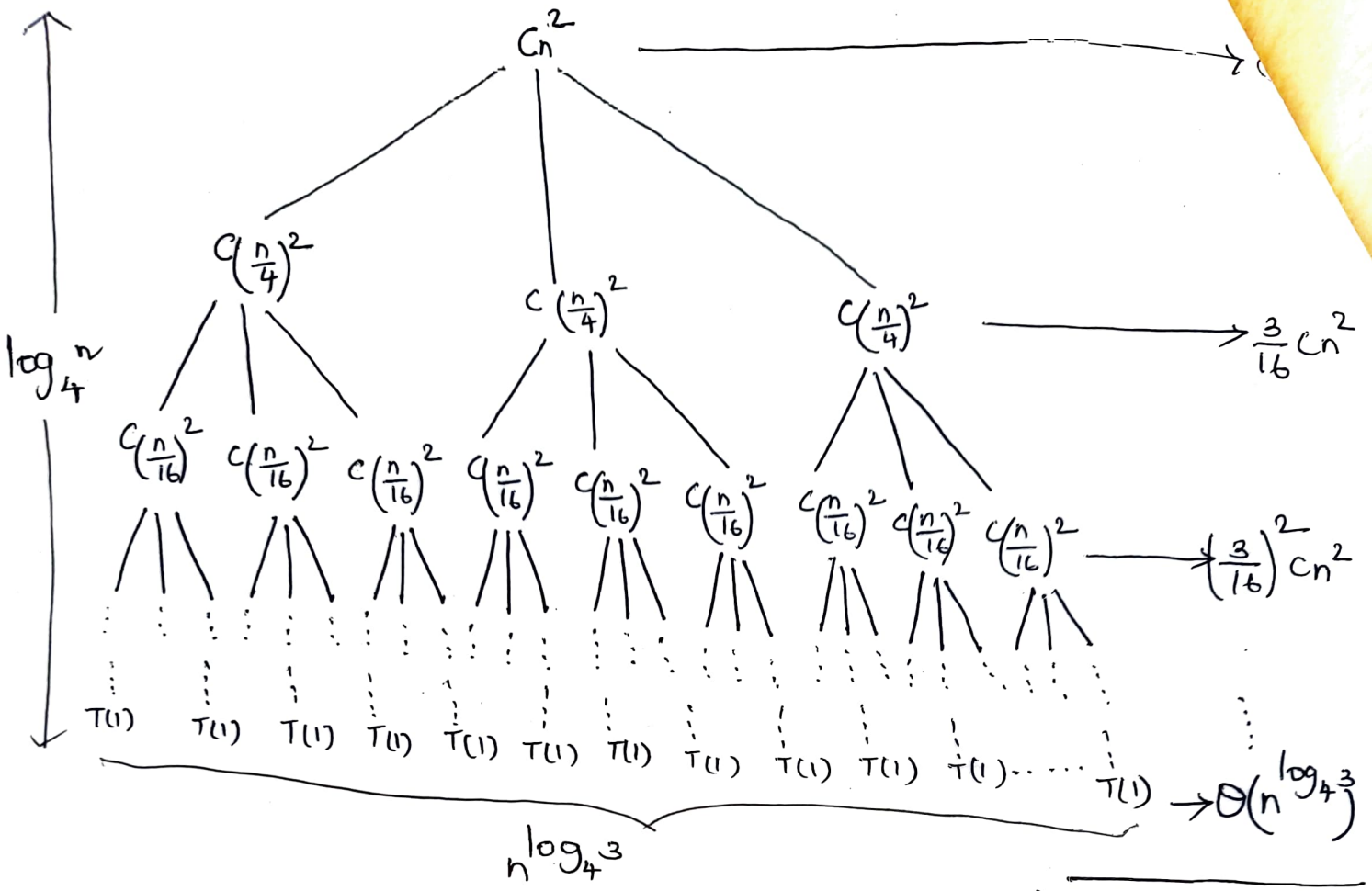
⇒ Recursion trees are particularly useful when the recurrence describes the running time of a divide-and-conquer algorithm.

Example: Consider the recurrence equation,

$$T(n) = 3T(n/4) + cn^2, c>0$$

Recursion Tree,





Total: $O(n^2)$

Total cost $T(n) = cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + \cdots + (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3})$

$$= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{(\frac{3}{16})^{\log_4 n} - 1}{(\frac{3}{16}) - 1} cn^2 + \Theta(n^{\log_4 3})$$

$$= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{1}{1-(3/16)} cn^2 + \Theta(n^{\log_4 3})$$

$$= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3})$$

$$= O(n^2)$$

Recurrence Relation - Master's theorem.

Master's theorem provides bounds for recurrence of the form.

$T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$ and $f(n)$ is a given function where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$ then
   $T(n) = \Theta\left(n^{\log_b a}\right)$
2. If $f(n) = \Theta\left(n^{\log_b a}\right)$ then $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Case(1) function $n^{\log_b a} > T(n) = \Theta\left(n^{\log_b a}\right)$

Case(2) Two functions are same (ie) $T(n) = \Theta\left(n^{\log_b a} \lg n\right) = \Theta(f(n) \lg n)$

Case(3) function $f(n)$ is larger, the solution is $T(n) = \Theta\left(n^{\log_b a}\right)$

Example:

$$T(n) = 9T(n/3) + n$$

$a = 9$, $b = 3$

$f(n) = n$.

$$n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$$

Master theorem,

$$f(n) = O\left(n^{\log_3 9 - \epsilon}\right)$$

where, $\epsilon = 1$

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

Ex-1

$a = 8$ $b = 2$ $c = 1$

$k = 2$

① $a < b^k \Rightarrow 8 < 2^2$

② (NO) $a = b^k \Rightarrow 8 = 2^2$

③ $a > b^k$ ✓

$\Theta\left(n^{\log_b a}\right)$

$\Theta\left(n^{\log_2 8}\right)$

$\Theta(n^3)$.

Example:

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

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

$f(n) = n\log n$

$a = 2, b = 2$

$\log_2^2 = 1$

Master theorem,

$f(n) = \theta(n^{\log_2^2} \log_n^1), k=1$

$\therefore T(n) = \theta(n^{\log_b^a} \log^{k+1} n)$

$= \theta(n^{\log_2^2} \log_n^2)$

$= \theta(n^1 \log^2 n)$

$\boxed{T(n) = \theta(n\log^2 n)}$

solve: $T(n) = 8T(n/2) + n^2$

Sln $T(n) = 8T(n/2) + n^2$

Here $f(n) = n^2$

$a = 8, b = 2$

$\log_2^8 = 3$

Master theorem,

$f(n) = O(n^{\log_b^a - \epsilon})$

$= O(n^{\log_2^8 - \epsilon})$

$f(n) = O(n^{3-\epsilon})$

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

$= \theta(n^{\log_2^8})$

$\boxed{T(n) = \theta(n^3)}$

$n^k \log n$

Pg - 127

Mathematical Induction:

Mathematical Induction can be used to prove a sequence of statements indexed by the positive integers. For example, it can be used to prove that,

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad \text{for all } n \geq 1$$

Here, the sequence of statements is

$$\text{Statement 1}: \sum_{i=1}^{1} i = \frac{1(1+1)}{2}$$

$$\text{Statement 2}: \sum_{i=1}^{2} i = \frac{2(2+1)}{2}$$

$$\text{Statement 3}: \sum_{i=1}^{3} i = \frac{3(3+1)}{2}$$

$$\vdots$$

$$\text{Statement n}: \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

(Statement 1-3 are obtained by everywhere replacing n by 1 in the original equation, then n by 2 and n by 3.)

Suppose we wish to use mathematical induction to prove a sequence of statements $S(1), S(2), \ldots$ we must,

> **Basis step**: Prove that $S(1)$ is true
>
> **Inductive step**: Assume that $S(n)$ is true and prove that $S(n+1)$ is true for all $n \geq 1$

Problem: Prove that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \geq 1$.

Solution:

Basis step: The equation is true for $n=1$,

$$\sum_{i=1}^{1} i = \frac{1(1+1)}{2}$$

$$= \frac{2}{2}$$

$$= 1$$

The truth is immediate since both sides are equal.

Inductive step: Assume that the equation is true for n and prove that it is true for n+1.

Assume, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Prove that, $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i\right) + (n+1)$$

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

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

$$\frac{n^2+2n+n+2}{2} = \frac{n^2+n+2n+2}{2}$$

$$\frac{n^2+3n+2}{2} = \frac{n^2+3n+2}{2}$$

L.H.S = R.H.S.

Hence proved (ie) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ for all $n \geq 1$.

Problem: prove that if a and $r \neq 1$ are real numbers

$$\sum_{i=0}^{n} ar^i = \frac{a(r^{n+1}-1)}{r-1}$$ for all $n \geq 0$ [Geometric sum]

Solution:

Basis step: Since n=0, the Basis step becomes,

$$\sum_{i=0}^{0} ar^i = \frac{a(r^{0+1}-1)}{r-1}$$

$$ar^0 = \frac{a(r-1)}{r-1}$$

$$a = a$$

Since both expressions are equal to a, the equation is true for n=0.

Inductive step: We must assume that the inequality is true for n and prove that it is true for n+1,

Assume $\sum_{i=0}^{n} ar^i = \frac{a(r^{n+1}-1)}{r-1}$

prove that $\sum_{i=0}^{n+1} ar^i = \frac{a(r^{n+2}-1)}{r-1}$

$$\sum_{i=0}^{n+1} ar^i = \sum_{i=0}^{n} ar^i + ar^{n+1}$$

$$= \frac{a(r^{n+1}-1)}{r-1} + ar^{n+1}$$

$$= \frac{ar^{n+1} - a + ar^{n+1}}{r-1}$$

$$= \frac{ar^{n+1} - a + (r-1)ar^{n+1}}{r-1}$$

$$= \frac{ar^{n+1} - a + r.ar^{n+1} - ar^{n+1}}{r-1}$$

$$= \frac{-a + rar^{n+1}}{r-1}$$

$$\sum_{i=0}^{n+1} ar^i = \frac{a(r^{n+2}-1)}{r-1}$$

Hence proved, (ie) $\sum_{i=0}^{n} ar^i = \frac{a(r^{n+1}-1)}{r-1}$ for all $n \geq 0$.

Problem: prove that $2n+1 \leq 2^n$ for all $n \geq 3$.

Solution:

Basis step: Since n=3, the basis step becomes.

$$2 \cdot 3 + 1 \leq 2^3$$

$$6 + 1 \leq 8$$

$$7 \leq 8.$$

Condition is true for n=3.

Inductive step: we must assume that the inequality is true for n and prove that it is true for n+1,

Assume $2n+1 \leq 2^n$

Prove that $2(n+1)+1 \leq 2^{n+1}$

$$2(n+1)+1 \leq 2^{n+1}$$

$$(2n+2+1) \leq 2^{n+1}$$

Here, case n is "within" case n+1 in the sense that

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

Note $2 \leq 2^n$ for $n \geq 1$, we obtain

$$\begin{aligned} 2(n+1)+1 = (2n+1)+2 &\leq 2^n + 2 \\ &\leq 2^n + 2^n \\ &= 2^{n+1} \end{aligned}$$

∴ Hence proved.

Exercises:

Use mathematic induction to prove the following.

1. $\sum_{i=1}^{n} (2i-1) = n^2$
2. $\sum_{i=1}^{n} i(i+1) = \frac{n(n+1)(n+2)}{3}$
3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
4. $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$