Unit -1

UNIT-1

Algorithm :→ Algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produce some values, or set of values as output.

An algorithm is thus a sequence of computational steps that transform the input into the output.

All algorithms must satisfy the following criteria:

1) Input :
2) output
3) Definiteness : Each instruction is clear and unambiguous.
4) Finiteness : The algorithm terminates after a finite no. of steps.
5) Effectiveness : Every inst must be very basic, so that it can be carried out effectively.

Analysis of algorithms or performance analysis refers to the task of determining how much computing time and storage an algorithm requires.

Time complexity of algorithm: Time taken by the algorithm to run its instructions.

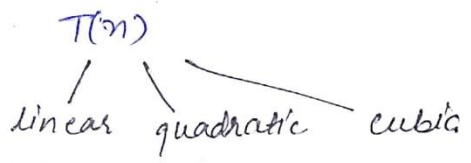
It is denoted by $T(n)$; whereas n is size of dataset.

$T(n)$: maximum time taken by an algorithm (2) to solve any instance of size n.

size: no. of values in the data set.

It is also the measure of goodness of any algorithm.

If $T(n) = 5$ that means no instance of size n will take time more than 5.



If an algo has time complexity of linear form than it is better than the algo having quadratic time complexity.

for example: Algo for Matrix Multiplication :-

```
1.  for i = 1 to n
2.      for j = 1 to n
3.          c[i,j] = 0
4.      for k = 1 to n
            c[i,j] = c[i,j] + A[i,k] * B[k,j]
5.      end
6.  end
7.  end
```

algo. of inst. $\rightarrow 2+n(b+3)$ steps.

$2+n(4\times4+2+1)$

$2+n(21)$

$\rightarrow 2+n(2+21n+3)$

$= 2+2n+21n^2+3n = 21n^2+5n+2$

$\rightarrow 2+n(21n^2+5n+2+3)$

It will have "cubic" complexity.

Growth of functions :→

We are concerned with running time of the algorithm.

⇒ How to interpret the time taken by the algo.

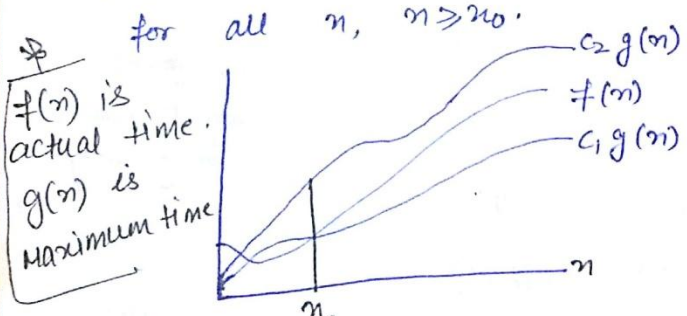
Asymtatic Notation :- The notation we use to describe the asymptotic running time of an algorithm are defined in terms of function whose domains are the set of natural numbers $N = \{0, 1, 2 \cdots\}$.

a line that continually approaches a given curve, but does not meet it at finite distance.

Such notations are convenient for describing the worst-case running time function $T(n)$.

$\Theta(n)$: f and g are two non-negative functions having positive values

$f(n) = \Theta(g(n))$ iff there exist 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, $n \geq n_0$.



$f(n)$ is actual time.
$g(n)$ is maximum time

Minimum no. of elements

This notation always generates the average case time complexity i.e average time required by the algorithm.

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

eg → 1→ function $5n\log n + 10n$.

$5n\log n + 10n = \Theta(n \log n)$

when n becomes larger than $5n\log n$ becomes more than $10n$.

larger

2. eg → $3n+2 = \Theta(n)$

as $3n+2 \geq 3n$ for $n \geq 2$

$3n+2 \leq 4n$ for $n \geq 2$, $c_1 = 3$, $c_2 = 4$, $n_0 = 2$

$\therefore$ $3n+2 = \Theta(n)$.

3. $6 * 2^n + n^2 = \Theta(2^n)$

4. $10n^3 + 5n^2 + 17 = \Theta(n^3)$

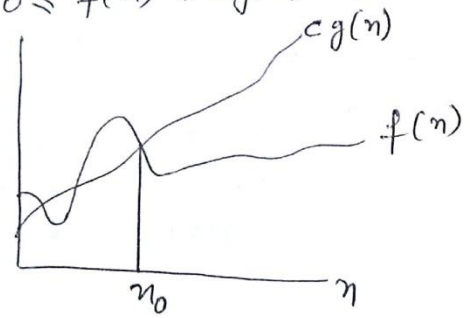
as $10n^3 \leq 10n^3 + 5n^2 + 17 \leq (10+5+17)n^3$,

2. Big-oh Notation :- $f(n) = O(g(n))$ iff these exist positive constants c and n_0 such that (size of problem)

$f(n) \leq c * g(n)$ for all n, $n \geq n_0$.

When we have only an asymptotic upper bound we use O- notation.

$0 \leq f(n) \leq c\,g(n)$ for all $n \geq n_0$.



It gives worst-case complexity.

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

eg → $6 * 2^n + n^2 = O(2^n)$

as $6 * 2^n + n^2 \leq 7 * 2^n$

Take $n = 3$

$6 * 8 + 9 \leq 7 * 8$

$57 \not\leq 56$

Take

$6 * 2^n + n^2 < 6 * 2^n$

Put $n = 2$

$24 + 4 \leq 6 * 4$

$28 \not\leq 24$

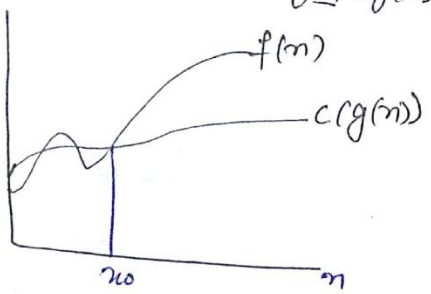
$\therefore$ we take $7 * 2^n$

$\therefore$ $6 * 2^n + n^2 \leq 7 * 2^n$ for $n \geq 4$

eg → ~~$3n+3 = O(n^2)$ as $3n+3 \leq 3n^2$ for $n \geq 2$~~

3. big-omega notation :- This provides asymptotic lower bound.

$\Omega(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$



This gives best case complexity.

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

eg → $3n+2 = \Omega(n)$ as $3n+2 \geq 3n$ for $n \geq 1$

eg → $10n^2+4n+2 \geq n^2$ for $n \geq 1$ $\therefore 10n^2+4n+2 = \Omega(n^2)$

eg → $6*2^n+n^2 = \Omega(n^2)$ also $6*2^n+n^2 = \Omega(2^n)$.

as $6*2^n+n^2 \geq n^2$

and as well as $6*2^n+n^2 = \Omega(n)$ also.

4. o-notation :→ $o(g(n)) = \{ f(n) :$ for any positive constant $c > 0$, there exist a constant $n_0 > 0$ such that $0 \leq f(n) \leq c(g(n))$ for all $n \geq n_0 \}$.

we use o-notation to denote an upper bound that is not asymptotically tight.

The function $f(n) = o(g(n))$ iff

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

5. **Little Omega ! (ω)** The function $f(n) = \omega(g(n))$ as $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$

$\omega(g(n)) \Rightarrow$ [little omega of g of n]

$\omega(g(n)) = \{ f(n)$: for any positive constant $c>0$, there exist a constant $n_0>0$ such that $0 \le c\,g(n) < f(n)$ for all $n \ge n_0 \}$

$\Rightarrow$ This provides a lower bound which is not asymptoti-cally tight.

for eg:- $\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \Omega(n)$.

$\frac{n^2}{2} = \Omega(n^2)$.

4. **o-Notation !** This provides an upper bound which is not asymptotically tight.

The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not asymptotically tight.

$\Rightarrow$ we use o-notation to denote an upper bound that is not asymptotically tight.

$o(g(n)) = \{ f(n)$: for any positive constant $c>0$, there exist a constant $n_0>0$ such that $0 \le f(n) \le c\,g(n)$ for all $n \ge n_0 \}$.

for example: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$.

eg→ $3n+2 = O(n)$

& $3n+2 = o(n^2)$ as $3n+2 \le 3n^2$

This gives asymptotically lower bound

Q:→ Prove that $n! = O(n^n)$

$f(n) = n!$

$n! = n(n-1)(n-2)(n-3)\cdots 1$

$f(n) \leq c \cdot g(n)$

$n(n-1)(n-2)(n-3)\cdots 1 \leq n.n.n.n \cdots$ n times

$n! \leq n^n$

H.P

Q:→ $\log n! = O(n \log n)$

$\log [n(n-1)(n-2)(n-3)\cdots 1] = \log n!$

$f(n) \leq c \cdot g(n)$

$\log (n(n-1)(n-2)\cdots 1) \leq \log (n^n)$

$\leq n \log n$

H.P.

Q:- $2^{n+1} = O(2^n)$

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

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

$2^{n+1} \leq c \cdot 2^n$

$2^n \cdot 2 \leq c \cdot 2^n$

$c \geq 2$ then $2^{n+1} \leq O(2^n)$

Q:- $2^{2n} \neq O(2^n)$

$f(n) = 2^{2n}$

$2^{2n} \leq c \cdot g(n)$

$2^{2n} \leq c \cdot 2^{2n}$

$2^{2n} \leq 2^n \cdot 2^n$

$c \geq 2^n$

While c is a constant no. ∴ $c \neq 2^n$ [∴ $2^n \neq O(2^n)$]

Recurrences :→ A function calls itself to solve a particular problem is called Recursion.

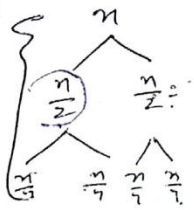
Recurrence :- A recurrence is an equation or inequality that describes a function in terms of its value or smaller inputs.

for eg: write a recurrance relation to find the factorial of a number:

$$f(n) = \begin{cases} 1 & \text{if } n=1 \\ n * fact(n-1) & \text{otherwise} \end{cases}$$

We define Recurrence relation as:

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ a.T\left(\frac{n}{b}\right) + f(n) & \text{if } n>1 \end{cases}$$



where a is no. of subproblems and b is size of the subproblem.

f(n): cost for dividing the problem into subproblems and finding the solution for the subproblems.

⇒ we use ~~three~~ four Methods for solving the recurrence relation:

1) Master Method (for Divide & Conquer techniques)
2) Substitution Method
3) Iteration /4) Recurrence Tree Method.

↳ 1. **Master Method** : This method provides bounds for recurrences as $T(n) = aT(n/b) + f(n)$, $a \geq 1$, $b > 1$

The Master Method requires three cases for solving the relation.

where, a sub-problems are created, each of which is 1/b the size of original problem, & in which divide & conquer steps take f(n) ti

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$
$T(n) = \Theta(n^{\log_b a})$ [$f(n) <$]

2. If $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$ [$f(n) =$

3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$ then
$T(n) = \Theta f(n)$. [$f(n) =$]

eq⇒ 1: $T(n) = 9T(\frac{n}{3}) + n$

$a = 9, \quad b = 3, \quad f(n) = n$

find $n^{\log_b a} \Rightarrow n^{\log_3 9} = n^2$

$f(n) < n^{\log_b a}$ ∴ case 1 valid

∴ $T(n) = \Theta(n^2)$

eq 2: $T(n) = T(\frac{2n}{3}) + 1$

$a = 1, \quad b = \frac{3}{2}, \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

$f(n) = n^{\log_b a} \log n$ case 2 valid

eq 3: $T(n) = 3T(\frac{n}{4}) + n \log n$

$a = 3, \quad b = 4, \quad f(n) = n \log n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

$f(n) > n \log n$

∴ $= \Theta(n \log n)$

eq 4: $T(n) = 4T(\frac{n}{2}) + n^3$

$a = 4, \quad b = 2, \quad f(n) = n^3$

$n^{\log_2 4} \Rightarrow n^2$

$f(n) > n^{\log_b a}$

∴ $T(n) = \Theta(n^3)$

→ Iteration Method :→ substitute the given function again and again.

— eq (i)

Eg 1:→ $T(n) = T\left(\frac{n}{2}\right) + c$ if $n>1$

$= 1$ if $n=1$

$T(n) = T\left(\frac{n}{2}\right) + c$ [but $n = \frac{1}{2}$

$= T\left(\frac{n}{4}\right) + c + c$ [Put the value of n as $\frac{n}{4}$ in eq (i)

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

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

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

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

$= T(1) + kc$ [$1 = \frac{n}{2^k}$, $2^k = n$, $k = \log_2 n$]

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

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

Eg 2:→ $T(n) = 2T\left(\frac{n}{2}\right) + n$ if $n>1$, if $n=1$

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

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

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

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

$= 8T\left(\frac{n}{8}\right) + 3n \Rightarrow 2^3 T\left(\frac{n}{2^3}\right) + 3n$

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

{ Put $n = \frac{1}{2}$, $T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$ — Put the value of $T\left(\frac{n}{2}\right)$ in $T(n)$ }

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

$$= 2^k T\left(\frac{n}{2^k}\right) + kn \qquad \left[\because n = 2^k, \therefore \frac{n}{n} = 1\right]$$

$$= nT(1) + kn \qquad \left[\because \frac{n}{2^k} = 1\right]$$

$$= n + n\log_2 n$$

$$= O(n \log_2 n) \text{ Ans.} \qquad \left[\frac{n}{2^k} = 1,\ n = 2^k,\ k = \log_2 n\right]$$

Recursion Tree Method :→

In a recursion tree, each node represents the cost of a single subproblem.

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

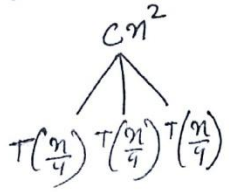
eg → $T(n) = \overset{\text{No. of subproblems}}{3}T\left(\frac{n}{4}\right) + \underbrace{cn^2}_{\text{cost}}$

⇒ we assume that n is exact power of 4.

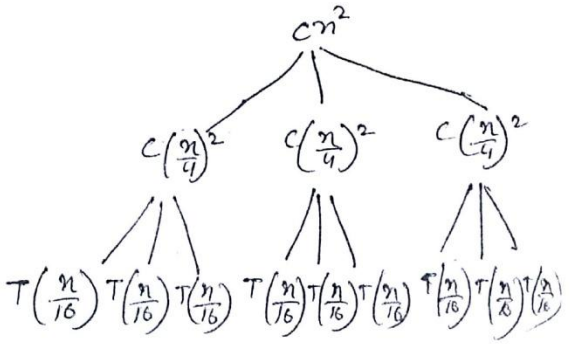
$T(n)$ is expanded into an equivalent tree representing the recurrence.

$T(n)$

(a)



(b)



(c)

∴ ↳ cn^2 term at the root represents the cost at the top level of recursion.

↳ Three subtrees of the root represents the costs incurred by the subproblems of size $\frac{n}{4}$.

↳ Part c shows this process carried one step further by expanding each node with cost $T(\frac{n}{4})$ from (b).

↳ The cost for each of three children of the root is $c(\frac{n}{4})^2$.

↳ The subproblem size for a node at depth i is $\frac{n}{4^i}$.

↳ Boundary condition is when $\boxed{\frac{n}{4^i} = 1}$

↳ $\frac{n}{4^i} = 1$, $n = 4^i$, $i = \log_4 n$

Tree has total $\log_4 n + 1$ levels.

↳ Now we determine the cost at each level.

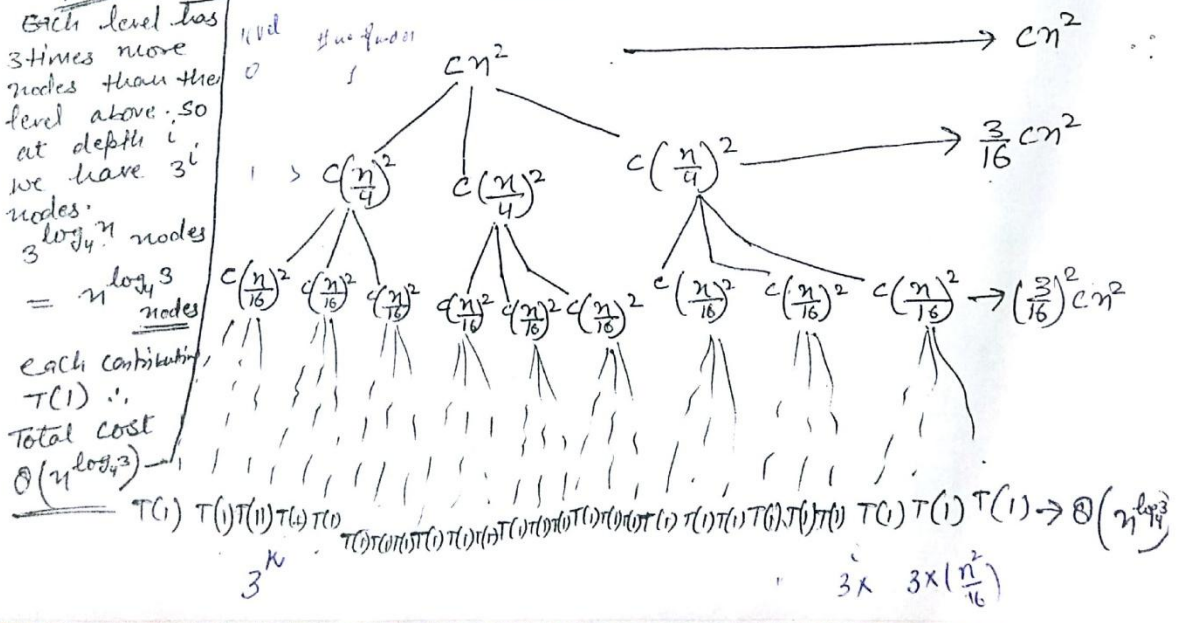
↳ Each level has three times more nodes than the level above.

↳ No. of nodes at depth i is 3^i.

∴ The last level, at depth i $[i = \log_4 n]$ has 3^i nodes means $3^{\log_4 n}$ nodes.

$= n^{\log_4 3}$ nodes

NOTE:
Each level has 3 times more nodes than the level above. So at depth i we have 3^i nodes.
$3^{\log_4 n}$ nodes
$= n^{\log_4 3}$ nodes
each contributing $T(1)$ ∴
Total cost $\Theta(n^{\log_4 3})$



↳ Last level has $n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} T(1)$, which is $\Theta(n^{\log_4 3})$.

⇒ Adding the cost at each level for the entire tree

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

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

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

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

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

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

eg → Solve by ~~Substitution~~ Iteration Method:

$$T(n) = 2T\left(\frac{n}{2}\right) + n\log n \quad \text{if } n > 1$$
$$= 1 \quad \text{if } n = 1$$

$$T(n) = 2T\left(\frac{n}{2}\right) + n\log n \quad \text{— (i)}$$

$$\Rightarrow 2\left[2T\left(\frac{n}{4}\right) + \frac{n}{2}\log\frac{n}{2}\right] + n\log n \quad \text{— (ii)}$$

{ Put the value of $T\left(\frac{n}{2}\right)$ in eq (i)

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

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

{ Put the value of $\frac{n}{4}$ in eq. (i)

$$= 2^3 T\left(\frac{n}{8}\right) + n\log\frac{n}{4} + n\log\frac{n}{2} + n\log n$$

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

$= 2^k T\left(\frac{n}{2^k}\right) + n\left[\{\log_2 n - 0\} + \{\log_2 n - 1\log_2 2\} + \{\log_2 n - 2\log_2 2\} + \cdots \{\log_2 n - k - 1\}\right]$

$\log\frac{n}{4} = \log n - \log 4 = \log_2 n - \log_2 2^2 = \log_2 n - 2\log_2 2 = \log_2 n - 2$

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

$\therefore$ this is possible only when $n = 2^k$.

$\therefore = n\,T(1) + n\left[k\log_2 n - [1+2+3+\cdots k]\right]$

$= n + n\left[k\log_2 n - \frac{k(k+1)}{2}\right]$

$= n + n\left[k\log_2 n - \frac{k^2}{2}\right]$

$\left[\frac{n}{2^k} = 1,\ n = 2^k,\ k = \log_2 n\right]$

$= n + n\left[\log_2 n(\log_2 n) - \frac{(\log_2 n)^2}{2}\right]$

$= n + n\left[(\log_2 n)^2 - \frac{(\log_2 n)^2}{2}\right]$

$= n + n(\log_2 n)^2 - \frac{n(\log_2 n)^2}{2}$

$= O\left(n(\log_2 n)^2\right)$

Q:- Solve By Iteration Method:

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

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

$= 3\left[3T\left(\frac{n}{16}\right) + \frac{n}{4}\right] + n$ [Put the value of $T\left(\frac{n}{4}\right)$]

$= 9T\left(\frac{n}{16}\right) + \frac{3n}{4} + n$

~~$= 9\left[9T\left(\frac{n}{16^2}\right) + \frac{n}{64}\right] + \frac{3n}{4}$~~ [Put the value of $T\left(\frac{n}{16}\right)$]

$= 8T\left(\frac{n}{16}\right) + \frac{5n}{64} + \frac{5n}{16}$

$= 8T\left(\frac{n}{16}\right) + \frac{85n}{64}$

$= 3^i T\left(\frac{n}{4^i}\right) + n + \frac{3n}{4} + \frac{9n}{16}$

$=$ Assume $\frac{n}{4^i} = 1$ since $T(1) = 1$

$n = 4^i$

$i = \log_4 n$

$\Rightarrow$ Put the value of i

$T(n) = 3^{\log_4 n} T(1) + n + \frac{3n}{4} + \frac{9n}{16}$

Note:-
$a^{\log_b n} = n^{\log_b a}$

$n + \frac{n \cdot 3}{4} + \frac{n \cdot 3^2}{4^2} + \cdots + \frac{n \cdot 3^i}{4^i}$

$= \sum_{i=0}^{\log_4 n} \frac{3^i}{4^i} \times n + \Theta\left(n^{\log_4 3}\right)$

$T(n) \leq \sum_{i=0}^{\log_4 n} n \times \left(\frac{3}{4}\right)^i + \Theta\left(n^{\log_4 3}\right)$

$\leq n \times \frac{\left(\frac{3}{4}\right)^0}{1 - \frac{3}{4}} + \Theta\left(n^{\log_4 3}\right)$

$\leq 4n + \Theta\left(n^{\log_4 3}\right)$

$\leq 4n + \Theta(n)$ as $\log_4 3 < 1$

$T(n) \leq O(n)$ [as $O(n) + O(n) = O(n)$]

Substitution Method :-

(1) How to make a good guess?

→ Recurrence Relation is of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Case 1: If $a = b = 2$ then
$T(n) = O(f(n) \log n)$ is a good guess.

Case 2: If $a = 1$, $b =$ any value then
$T(n) = O(\log n)$ is a good guess

Case 3: If $a \neq 1$, $b > a$ then
$T(n) = O[f(n) \cdot n]$ is a good guess.

Q:→ $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{if } n > 1 \end{cases}$

Ans:- Assume that $T(n) = O(n^2)$

as $f(n) \leq c \cdot g(n)$

To prove ∴ $T(n) \leq c \cdot n^2$

Put the value of $T(n)$ in eq.(i)

$T(n) \leq T(n-1) + n$ — eq(i)

$\leq c \cdot (n-1)^2 + n$

$\leq c[n^2 + 1 - 2n] + n$

$\leq cn^2 + c - 2cn + n$

$T(n) \leq cn^2$

Now apply Mathematical Induction to Check for Basic Condition and check the value of c.

When $n = 1$, $T(1) \leq c \cdot 1^2$, by Choosing $c = 1$

$1 \leq 1$

4. Substitution Method :→ This method consists of 5 guessing an asymptotic (upper or lower) bound on the solution, and trying to prove it by induction.

∴ It follow two steps:

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

eg→ $T(n) = 2T(\lfloor n/2 \rfloor) + n$ — (1)

Solution :→ This is similar to the recurrances we have seen before. Therefore we guess the solution as $T(n) = O(n \log n)$.

↳ our method to prove that $T(n) \leq cn\log n$ (↳ $f(n)$ $g(n)$)

$f(n) \leq c \cdot g(n)$ for $c > 0$, we need to prove that $T(n) \leq cn\log n$.

↳ Let us assume that this bound holds for $\lfloor \frac{n}{2} \rfloor$

i.e $T(\lfloor n/2 \rfloor) \leq c \lfloor n/2 \rfloor \log \lfloor n/2 \rfloor$.

Substituting into Recurrance Relation eq (1)

$$\begin{aligned} T(n) &\leq 2(c \lfloor n/2 \rfloor (\log \lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &\leq cn \log n. \end{aligned}$$

[as $\log_2^2 = 1$]

This holds as long as $c \geq 1$.

⇒ Mathematical induction is used to check that solution is hold true for boundary conditions.

⇒ we use the base case for Inductive proof.

↳ Boundary condition is when $n=1$.

i.e $T(1)=1$.

Now check for this

$T(1) \leq c\,1\log 1$ as $\log 1$ is 0 $\therefore T(1)=0$

But Acc. to Mathematical Induction, if the Inductive proof is true for n, it should be true for 1 also.

$\therefore$ The Base case Proof fails to hold.

↳ To remove this difficulty, we use the advantage of threshold value 'n_0' by requiring us to prove

$T(n) \leq cn\log n$ for $n \geq n_0$.

where n_0 is a constant of our choosing.

Thus we replace $T(1)$ by $T(2)$ & $T(3)$ as the base case for Inductive proof., letting $n_0=2$

Put the values in eq (i)

$T(2) = 4$, $T(3) = 5$.

Inductive proof $T(n) \leq cn\log n$

$\therefore T(2) \leq c\,2\log 2 \Rightarrow 4 \leq c \cdot 2$

& $T(3) \leq c\,3\log 3$.

Take c as large example to solve this condition

for $c \geq 2$ suffice for the base case of $n=2$ & $n=3$ to hold.

eg2: $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$

guess — $O(n\log n)$

[added '17' does not affect the Recurrence.

Q:→ Q1- Show $T(n) = T(\lfloor \frac{n}{5} \rfloor) + 1$ is $O(\log n)$

⇒ Here $a = 1$, $b =$ any value, then

To prove: $T(n) \leq c \cdot \log n$

$$T(n) \leq c \cdot \log \frac{n}{2} + 1$$
$$\leq c \log n - c \log 2 + 1$$
$$\leq c \log n - c + 1$$
$$\leq c \log n \quad \text{for } c \geq 1$$

$\therefore T(n) = O(\log n)$. for $c \geq 1$

Q1→ $T(n) = 2T(\sqrt{n}) + 1$

[always assume $\log_2 n = n$ for these type of questions]

⇒ let $m = \log_2 n$

$\therefore 2^m = n$

i.e $\sqrt{n} = 2^{m/2}$

$\therefore T(2^m) = 2T(2^{m/2}) + 1$ [as $n = 2^m$]

changing the recurrence to $S(m) = T(2^m)$

$$S(m) = 2S(\frac{m}{2}) + 1$$

Now the solution is $O(\log m)$

$T(n) = O(\lg \lg n)$.

5. **Changing Variables :→** Sometimes a little algebraic manipulation can make a recurrence more difficult.

↳ Recurrences can be reduced to simpler ones by changing variables.

eg→ Solve $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$

Solution :→ To solve this we assume that $\log n = m$

$\therefore n = 2^m$

$\therefore T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$ becomes

$T(2^m) = 2T(2^{m/2}) + m$ [as $2^m = n$, $\sqrt{n} = 2^{m/2}$]

Let $S(m) = T(2^m)$ [Renaming]

$S(m) = 2S(\frac{m}{2}) + m$

We have already solved these types of recurrences $\therefore$ we can say that

$S(m) = O(m \log m)$

$= O(\log n \log \log n)$ Ans.

Data structure for disjoint sets :→.

→ Some applications involve grouping n elements into a collection of disjoint sets.

→ Disjoint sets :- If they have no elements in common.

→ Two main operations on disjoint sets are:- FIND and UNION.

→ A disjoint set data structure maintains a collection $'S' = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

→ Each set is identified by a representative which is some member of the set.

→ There may be some prespecified rule for choosing the representative, such as choosing the smallest member in the set etc.

→ Let 'x' denotes an object (element of the set).

We wish to support the following operations:→

① MAKE-SET(x):
It creates a new set whose only member is x.
Since the sets are disjoint, we require that 'x' not already in some other set.

② UNION(x,y):
Unites the dynamic sets that contain x & y say S_x and S_y into a new set that is union of these two sets.

→ S_x & S_y assumed to be disjoint prior to UNION operation, & thus they are removed from 'S' after their UNION.

3) FIND_SET(x) :- Returns a pointer to the representative of the set containing x.

We analyze the running time of disjoint-set data structure in terms of two parameters:

↳ n & m.

* n :→ No. of MAKE_SET operations.

m :- No. of MAKE_SET, UNION, FIND operations.

→ Since the sets are disjoint, each UNION operation reduces the number of sets by one.

∴ with 'n-1' UNION operations, we are left with only one set.

→ Also to form 'n' sets, n MAKE_SET operations and as 'm' includes MAKE_SET & UNION operations

∴ $m \geq n$. [More No. of operations in m as compared to n].

Application :—

↳ Determining the connected components of an undirected graph.

→ CONNECTED COMPONENTS (G)

```
1. for each vertex u ∈ V[G]
2.     do MAKE_SET(u)
3. for each edge (u,v) ∈ E[G]
4.     do if FIND_SET(u) ≠ FIND_SET(v)
5.         then UNION(u,v)
```

NOTE:

V[G] :- set of vertices.

E[G] : set of edges.

SAME-COMPONENT answers the queries about whether two vertices are in the same connected component.

1. if FIND-SET(u) = FIND-SET(q)
2. then return TRUE
3. else return FALSE.

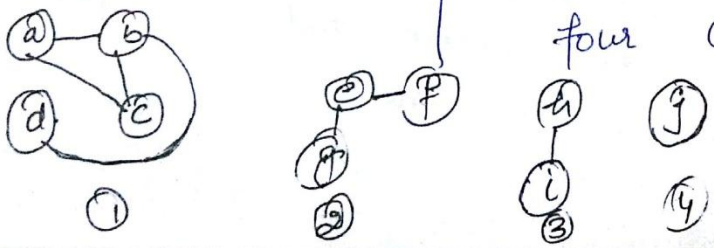
Q:- find the connected components of the graph having vertices, V = {a, b, c, d, e, f, g, h, i, j} and edges processed in order {(b,d), (e,g), (a,c), (h,i), (a,b), (e,f), (b,c)}

Ans :→

Edge processed	Collection of disjoint sets.
	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j}
(b,d)	{a} {b,d} {c} {e} {f} {g} {h} {i} {j}
(e,g)	{a} {b,d} {c} {e,g} {f} {h} {i} {j}
(a,c)	{a,c} {b,d} {e,g} {f} {h} {i} {j}
(h,i)	{a,c} {b,d} {e,g} {f} {h,i} {j}
(a,b)	{a,c,b,d} {e,g} {f} {h,i} {j}
(e,f)	{a,c,b,d} {e,f,g} {h,i} {j}
(b,c)	as (b,c) belongs to same set ∴ NO UNION.

{a,c,b,d}, {e,f,g}, {h,i}, {j}.

four Connected Components.



Q:- $V = \{a, b, c, d, e, f, g, h, i, j, k\}$

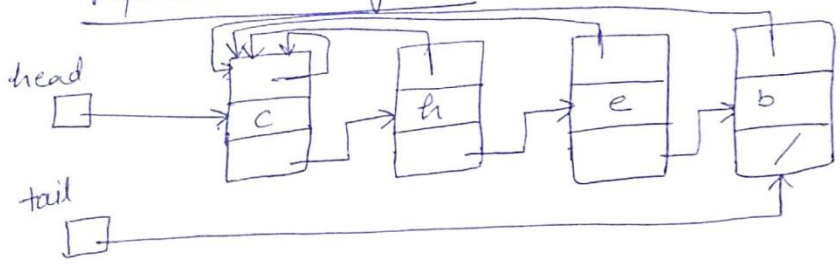
$E = \{(d,i), (f,k), (g,i), (b,g), (a,h), (i,j), (d,k), (b,j), (d,f), (g,j), (a,e), (i,d)\}$

⇒ LINKED-LIST REPRESENTATION OF DISJOINT SETS :→

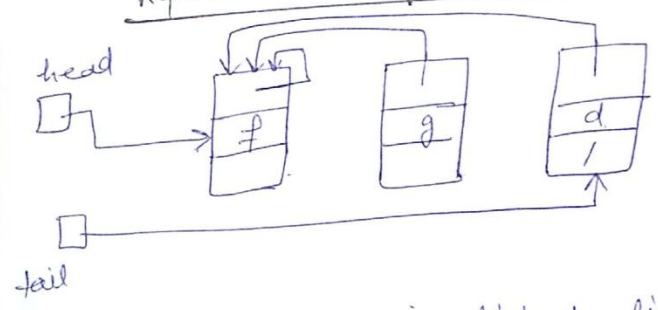
$S_1 = \{c, h, e, b\}$

$S_2 = \{f, g, d\}$

Representation of S_1 :→



Representation of S_2 :-



(1) first object in linked list serves as its set's representative.

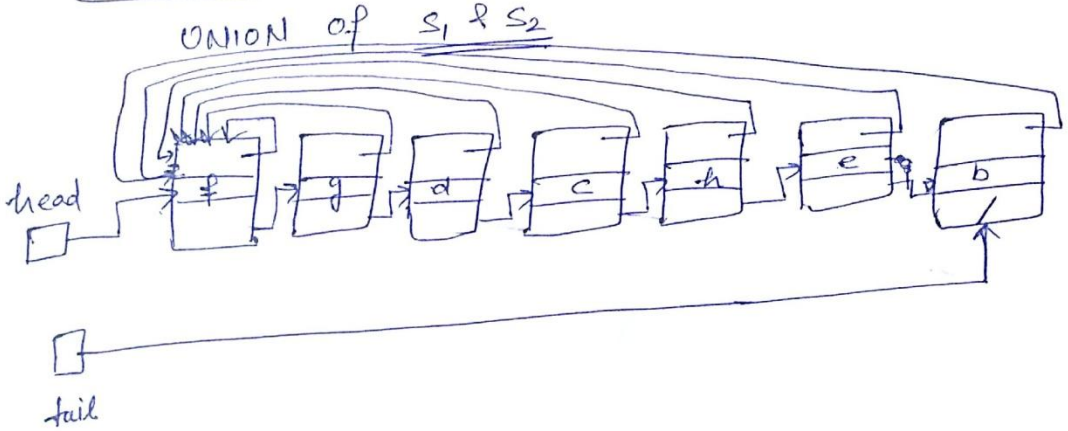
(2) Each object contains a set member, a pointer to the object containing the next set member, a pointer back to the representative.

(3) Head points to representative and tail points to last object in the list.

MAKE_SET(x) - Require O(1) time, create a new link list whose only member is x.

FIND_SET(x) :- Require O(1) time, as return the pointer from 'x' back to representative.

⇒ Simple Implementation of UNION :→



→ We perform UNION(x,y) by appending x' list on the end of y's list.

→ we use 'tail' pointer of y's list to find where to append x's list.

→ Representative is, representive of the set containing y.

↳ we must update the pointer for each object orginally on x's list.

Path Compression :→ very simple and effective.

We use path compression during FIND-SET operations to make each node the find path points directly to the root.

Path compression does not change any rank.

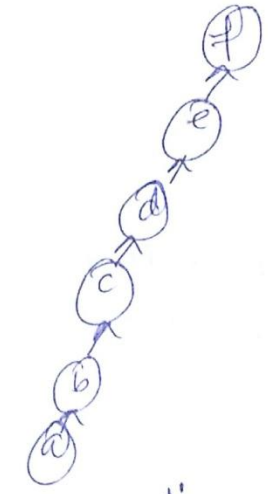
We perform three operations as:

1. MAKE-SET → simply creates a tree.
2. UNION → causes the root of one tree to point to the root of other.
3. FIND-SET →, by following parents pointers until we find root of the tree.

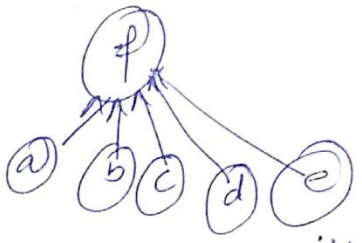
```
MAKE-SET(x)
1.  P[x] ← x
2.  rank[x] ← 0

UNION(x,y)
1.  LINK( FIND-SET(x), FIND-SET (y))

LINK (x,y)
1.  if rank[x] > rank[y]
2.      then P[y] ← x
3.      else P[x] ← y
4.           if rank[x] = rank[y]
5.           then rank[y] ← rank[y] + 1.
```

FIND-SET procedure with path compression is simple:

```
FIND-SET(x)
1. if x ≠ P[x]
2.     then P[x] ← FIND-SET(P[x])
3. return P[x].
```



(i)



(ii)

each node has a pointer to parent.

(iii) after executing FIND_SET(a)

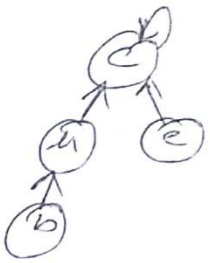
each node on the find path points to the root node.

It will take linear time $O(n)$.

Disjoint forest :→ (for faster implementation) we represent sets by rooted trees, with each node containing one member and each tree representing one set.

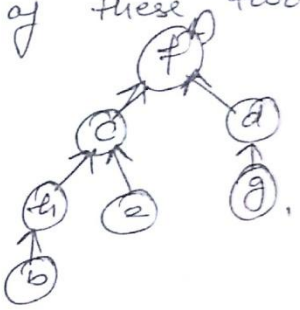
① ↳ each member points to its parent.

set $\{c, h, e, b\}$, $\{f, d, g\}$



(f, g)

Union of these two sets:



root of one tree to point to the root of the other.

⇒ Heuristics to improve the running time.

We can achieve a running time better than linked list implementation.

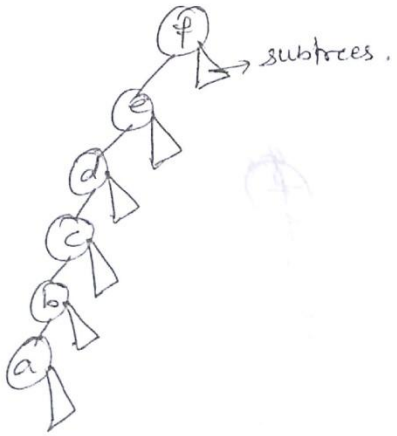
↳ Union by Rank

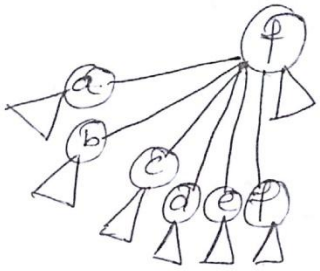
↳ Path Compression

↳ Union by Rank :— we maintain a rank, that is an upper bound on the height of the node.

In this the root with smaller rank is made to point to the root with larger rank during UNION operation.

↳ Path Compression :→ We use it during FIND-SET operation to make each node on the find path point directly to the root.





Quick Sort :→

Quicksort is based upon divide and conquer paradigm:

Divide: Partition the array $A[p \ldots r]$ into two subarrays $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that each element in $A[p \ldots q-1]$ is less than $A[q]$ and each element in $A[q+1 \ldots r]$ is greater than $A[q]$.

Conquer: Sort the two subarrays

Combine: Since the subarrays are sorted :. No need to combine the entire array, Now $A[p \ldots r]$ is sorted.

```
QUICKSORT(A, P, r)
1.  if  p<r
2.      then  q ← PARTITION(A, P, r)
3.            QUICKSORT(A, P, q-1)
4.            QUICKSORT(A, q+1, r)

PARTITION(A, P, r)
1.   x ← A[r]
2.   i ← P-1
3.   for j ← p to r-1
4.       doif A[j] ≤ x
5.           then i ← i+1
6.                exchange A[i] ↔ A[j]
7.   exchange  A[i+1] ⇔ A[r]
8.   return i+1.
```

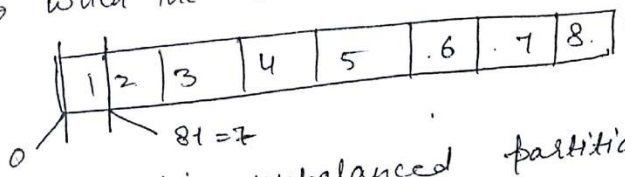
Performance Analysis of Quicksort :-

The running time of quicksort depends on whether the partitioning is balanced or unbalanced.

1. Worst case Partitioning :→

The worst case behaviour for quicksort occurs when the partitioning produces one subproblem with $n-1$ elements and one with 0 elements.

for eg→ when the list is already sorted.



Let us assume that this unbalanced partitioning arises in each recursive call:

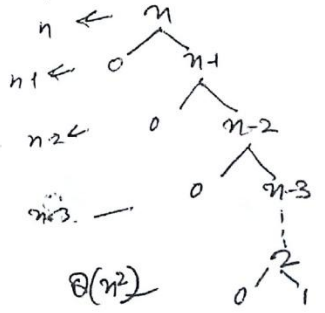
Partitioning call cost $\Theta(n)$ time.

$T(0) \rightarrow$ Time taken for array of size zero.

$$T(0) = \Theta(1).$$

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

Solve it by substitution Method, Recursion tree Method,



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

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

2. Best case Partitioning :-

PARTITION produces two subproblems, each of size no more than $\frac{n}{2}$. Since one is of size $\frac{n}{2}$ and one of size $\lceil \frac{n}{2} \rceil - 1$.

In this case quicksort runs much faster.

↳ In this case, the recurrence for the running time is -

$$T(n) \leq 2T\left(\frac{n}{2}\right) + \Theta(n)$$

Now it satisfies case 2 of master Method.

$$a = 2,\ b = 2,\quad n^{\log_2 2} = n$$

$$\& \quad f(n) = n$$

$$\therefore \quad f(n) = n^{\log_b a}$$

$\therefore$ complexity is $O(n \log n)$

By Iteration Method: $T(n) = 2T\left(\frac{n}{2}\right) + cn$

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

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

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

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

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

$$\Rightarrow \frac{n}{2^i} = 1, \quad n = 2^i \Rightarrow \boxed{i = \log_2 n}$$

$$= n \cdot T(1) + i \cdot n \quad [\text{By putting the}$$

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

$$= \Omega(n \log_2 n) \quad \text{Ans.}$$

⇒ Balanced Partitioning :→ The average case running time of quicksort is much closer to the best case than to the worst case.

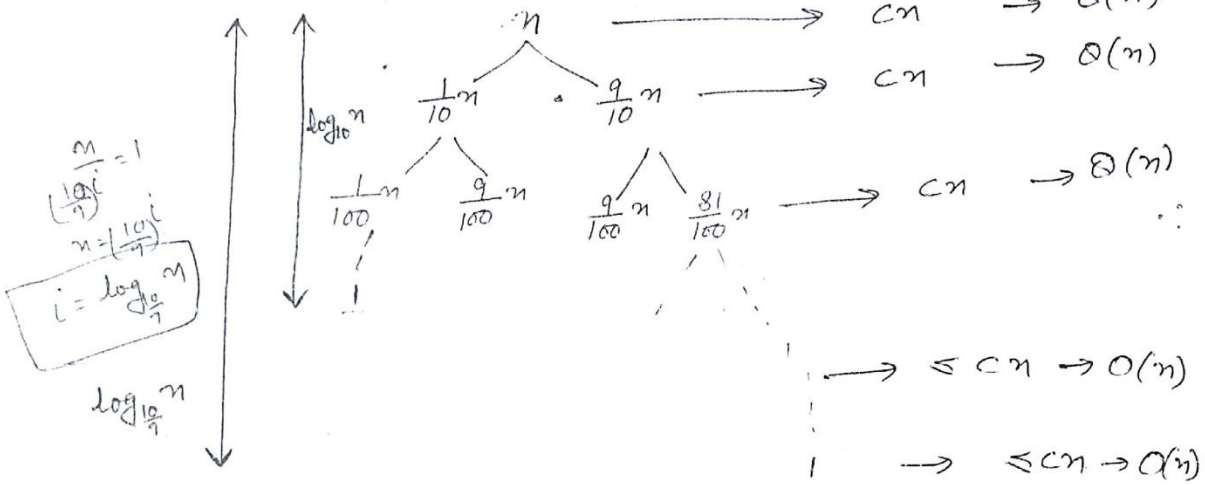
In best case, we have $\frac{n}{2}, \frac{n}{2}$ partitioning.

In worst case, we have $n-1, 0$ partitioning.

In average case, we have partitioning somewhere b/w best case & worst case. In best case we

have good split & in worst case, we have bad split, but in average case, we have combination of good & bad splits.

Suppose, the partitioning algorithm always produces a 9-to-1 proportional split, which at its first blush seems quite unbalanced.



We then obtain the recurrence

$$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

In recursion tree, every level of the tree has cost cn, until a boundary condition is reached.

at depth $\log_{10} n = \Theta(\log n)$.

(ch-h)

$\frac{n}{b^i} = 1$

$n = b^i$

$i = \log_b n$.

levels have cost at most cn.

Recursion terminates at depth $\log_{\frac{10}{9}} n = \Theta(\log n)$

at this level we have cost cn.

∴ the total cost of quicksort is $\Theta(n \log n)$

In average case, we take random number for split.

Even if take 99 to 1 split, this also yields $O(n \log n)$ running time.

⇒ Reason is that any split of constant proportionality yields a recursion tree of depth $\Theta(\log n)$ and level at cost at each level is cn.

∴ the total cost is $\Theta(n \log n)$.

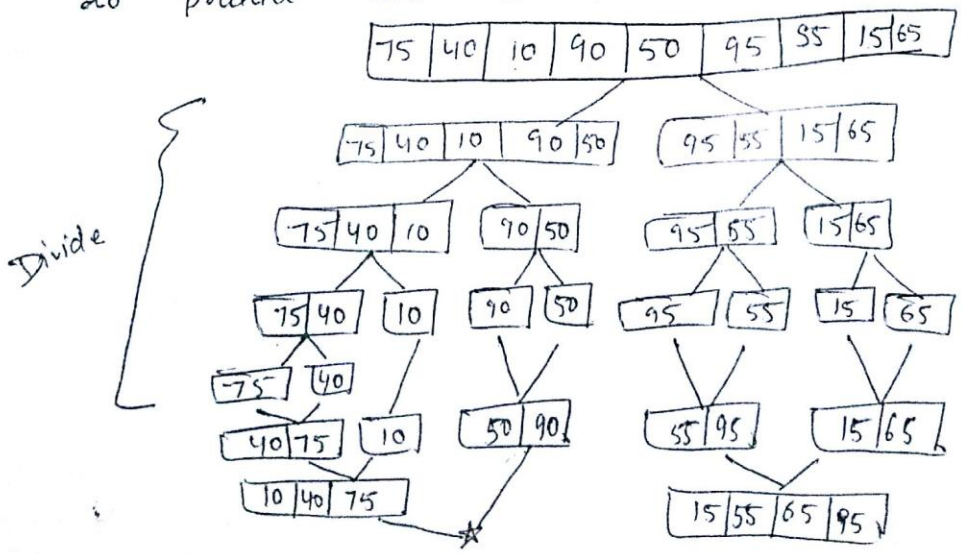
→ Merge Sort :→

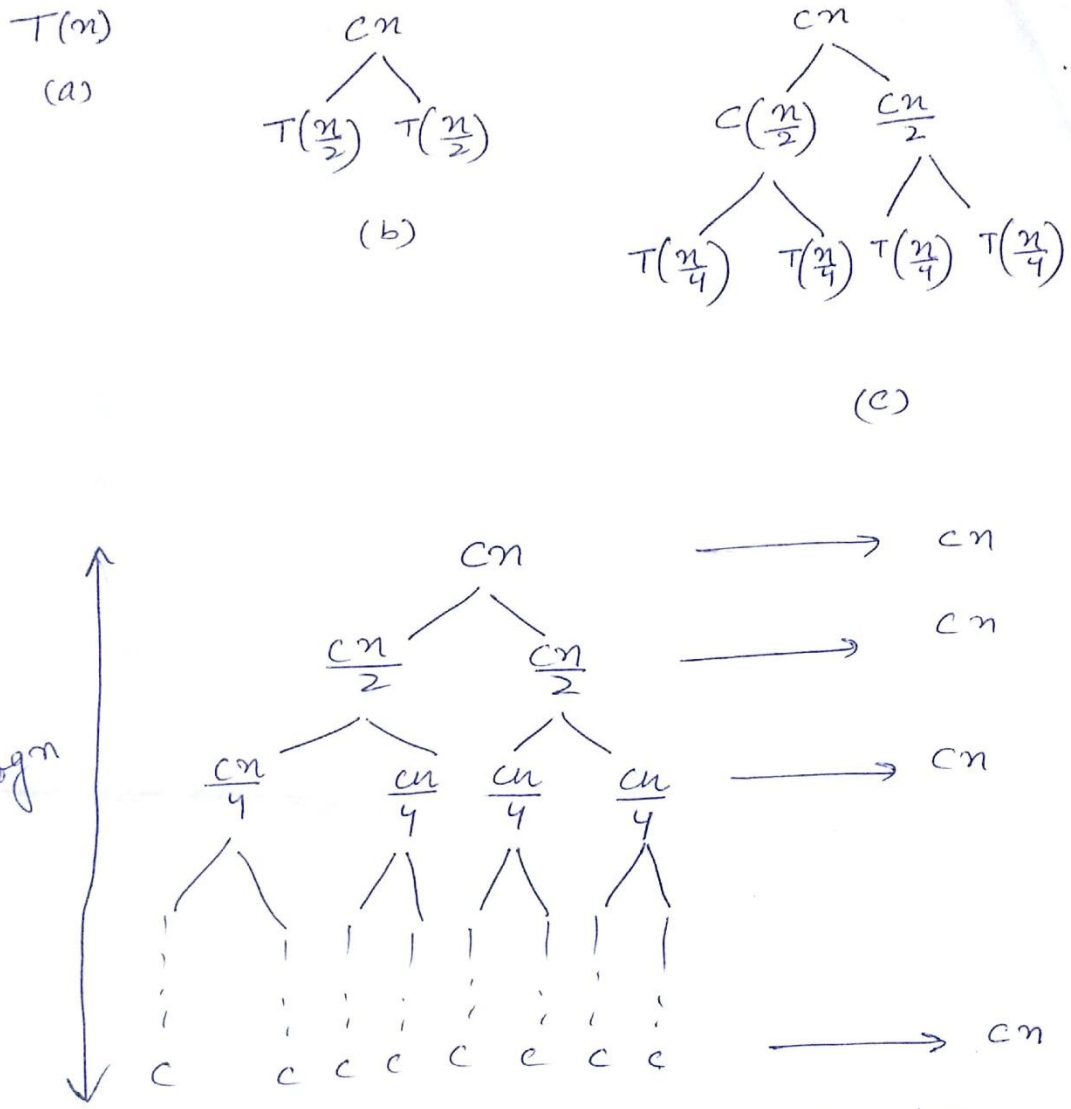
Merge sort algorithm closely follows divide & conquer paradigm in the following manner:

1) Divide: Divide n element sequence to be sorted into 2 subsequences of $n/2$ elements each.

2) Conquer: Sort the two subsequences recursively using merge sort.

3) Combine: Merge the two sorted subsequences to produce the sorted answers.





→ Add cost across each level of the tree.

$\frac{cn}{2} + \frac{cn}{2} = cn$ → level 2. and so on.

→ Level 'i' below the top has 2^i nodes.

each contributing $\frac{cn}{2^i}$ or $c\left(\frac{n}{2^i}\right)$.

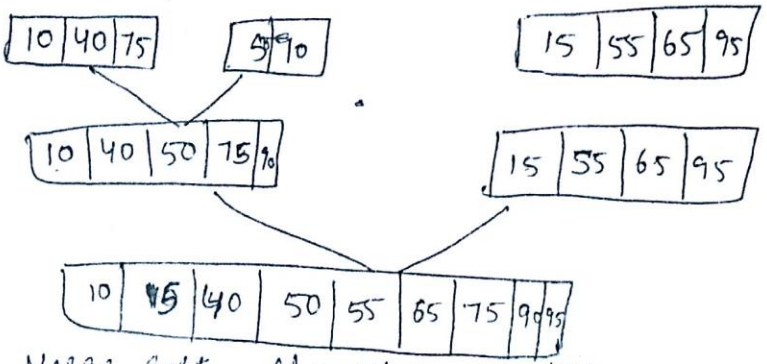
→ Therefore at i^{th} level, below the has

Total cost. $2^i \cdot c\left(\frac{n}{2^i}\right) = \underline{\underline{cn}}$.

or at the bottom level, these are n nodes

each contributing a cost of c

∴ Total cost = $\underline{\underline{cn}}$.



How Merge Sort Algorithm works:

↳ Key operation is to Merging the two sorted sequences in the combine step.

↳ for Merging we use Merge (A, p, q, r).
A is array. p, q, r are such that $p \le q < r$.

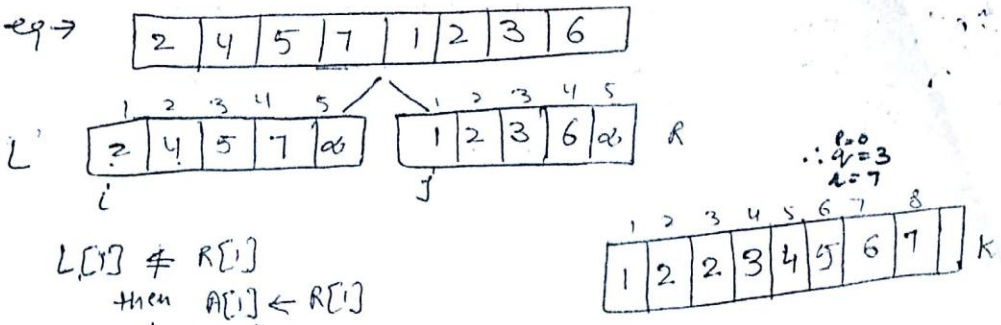
↳ 'Merge' assumes that the subarrays A[p..q] and A[q+1..r] are in sorted orders.

algo :→

```
MERGE (A, p, q, r)
1.  n1 ← q − p + 1        → compute the length of A[p..q]
2.  n2 ← r − q            → compute the length of A[q+1..r]
3.  create arrays L[1..n1+1] and R[1..n2+1]   [create two subarrays of length n1+1 & n2+1]
4.  for i ← 1 to n1
5.      do L[i] ← A[p+i−1]                    [copy the subarray A[p..q] in L[1 to n1]]
6.  for j ← 1 to n2
7.      do R[j] ← A[q+j]                      [copy the subarray A[q+1..r] in to R[1 to n2]]
8.  L[n1+1] ← ∞           → [e.g. put the sentinels at the end of arrays]
9.  R[n2+1] ← ∞
10. i ← 1
11. j ← 1
12. for k ← p to r
13.     do if L[i] ≤ R[j]
14.         then A[k] ← L[i]
15.              i ← i + 1
16.         else A[k] ← R[j]
17.              j ← j + 1
```

[Merge the subarrays into one sorted array]

eg →



$L[1] \not\leq R[1]$
then $A[1] \leftarrow R[1]$
$j++$, $j=2$

$L[1] \leq R[2]$
$A[2] \leftarrow L[1]$
$i++$, $i=2$

$L[2] \not\leq R[2]$
$\therefore A[3] \leftarrow R[2]$, $j++$, $j=3$

$L[2] \not\leq R[3]$
$4 \not\leq 3$
$\therefore A[4] = R[3]$, $j++$, $j=4$

$L[2] \leq R[4]$
$4 \leq 6$
$A[5] = L[2]$, $i++$, $i=3$

$L[3] \leq R[4]$
$5 \leq 6$, $i++$, $i=4$
& $A[6] = L[3]$

$L[4] \not\leq R[4]$
$\therefore A[7] = R[j]$, $j++$, $j=5$

$L[4] \leq R[5]$
$\therefore A[8] = L[4]$, $i++$, $i=5$

$L[5] = \infty$ & $R[5] = \infty$.

MERGE-SORT (A, P, R)

1. if $p < r$ → [the array has more than 1 element]
2. then $q \leftarrow \lfloor (p+r)/2 \rfloor$
3. MERGESORT (A, P, q)
4. MERGESORT (A, q+1, r)
5. MERGE (A, P, q, r) .

Analysis of Merge-Sort Algorithm :→

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise} \end{cases}$$

$D(n)$: time to divide the problem
$C(n)$: Time to combine the solutions.

Although the pseudocode for MergeSort works correctly if the no. of elements are not even, we assume that:

Original problem size is a power of 2, and in each step, it divide the subproblem in $1/2$. That is in every divison it generates two subsequences of size $\frac{n}{2}$.

Divison takes constant time.

$\therefore T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ → time to combine the solutions as we apply merge procedure on n elements $\therefore \Theta(n)$.

i.e $T(n) = 2T\left(\frac{n}{2}\right) + Cn.$

⇒ By Master Method, according to Case No. 2

$$f(n) = n$$
$$n^{\log_b a} = n^{\log_2 2} = n$$
$$n = n$$

$\therefore T(n) = (n \cdot \log n)$ Ans.

⇒ By Recusion tree Method :→

The 'cn' term is the root of the tree.

For convenience we assume that n is exact power of 2.

Total No. of levels are: i Merging Sort

Now $\frac{n}{2^i} = 1$

$n = 2^i$

$i = \log_2 n$

There are $\log_2 n + 1$ No. of levels as for base case $\log_2 1 = 0$. for $n = 1$

At this level we have zero No. of nodes

∴ No. of levels = $\log_2 n + 1$

This gives us correct no. of levels.

→ Now each level cost cn.

∴ $\log_2 n + 1$ levels cost $cn(\log_2 n + 1)$

$= \boxed{cn \log_2 n + cn}$

Ignoring the low order term:

$T(n) = \Theta(n \log_2 n)$. Ans.

Medians & Order Statistics

⇒ The i^{th} order statistic of a set of n elements is the i^{th} smallest element.
i.e. 5^{th} order statistics of a set of n elements where $n > 5$ is the 5^{th} smallest element.

e.g:- Minimum of a set of elements is 1^{st} order statistic ($i = 1$) &
Maximum of a set of elements is n^{th} order statistic ($i = n$)

⇒ Median: informally midpoint of the set.
When n is odd, Median $= \frac{(n+1)}{2}$

When n is even, we have two medians given by $i = n/2$ and $i = n/2 + 1$.

Thus we can say that medians are represented generally as $i = \lfloor (n+1)/2 \rfloor$ (lower median) and $i = \lceil (n+1)/2 \rceil$ (upper median)

⇒ To determine the minimum of a set of n elements we need (n-1) comparisons.
e.g:- Let us have an array A[n] having n elements

```
MINIMUM (A)
1.  min ← A[1]
2.  for i ← 2 to length[A]
3.      do if min > A[i]
4.          then min ← A[i]
5.  return min
```

⇒ Similarly for finding the Maximum number, we need (n-1) comparisons as well.

...ne running time in both the cases will be ...

To find simultaneous Minimum & Maximum of n elements, we can find the Minimum & Maximum independently using $(n-1) + (n-1) = (2n-2)$ comparisons

OR

We can maintain the minimum & maximum elements seen thus far.

(1) If we process each element of input by comparing it with current maximum & current minimum, we have a cost of 2 comparison per element.

(2) If we process the input elements in pairs, by first comparing the input elements with each other and then comparing the smaller element to the current minimum & the larger element to the current maximum; thus having a cost of 3 comparison for 2 elements.

(3) Setting up the initial values for current minimum & current maximum depends on whether n is even or odd.

↳ If n is odd, we set both the minimum & maximum to the value of the first element, and then process the rest of the elements in pairs.

↳ If n is even, we perform the 1st comparison on the first two elements to determine the initial values for minimum & maximum & then process the rest of the elements in pairs.

P.T.O.

Taking expected values, we have :-

$$E[T(n)] \leq E\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] -$$

$$= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n)$$

We have, $\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \leq \lceil n/2 \rceil \end{cases}$

If n is even, each term from $T(\lceil n/2 \rceil)$ upto $T(n-1)$ appears exactly twice in the summation.

If n is odd, all these terms appear twice & $T(\lfloor n/2 \rfloor)$ appear once.

Thus we have

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n) \quad -$$

We solve this recurrence by substitution —

Let us assume that $T(n) \leq cn$ satisfies the given recurrence for some constant c.

We assume that $T(n) = O(1)$ for n less than some constant.

We also pick a constant a such that the function described by $O(n)$ term above is bounded from above by 'an' for all $n > 0$.

Taking expected values, we have :-

$$E[T(n)] \leq E\left[\sum_{k=1}^{n} X_k \cdot T(\max(k-1, n-k)) + O(n)\right] -$$

$$= \sum_{k=1}^{n} E[X_k \cdot T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{k=1}^{n} E[X_k] \cdot E[T(\max(k-1, n-k))] + O(n)$$

$$= \sum_{k=1}^{n} \frac{1}{n} \cdot E[T(\max(k-1, n-k))] + O(n)$$

We have, $\max(k-1, n-k) = \begin{cases} k-1 & \text{if } k > \lceil n/2 \rceil \\ n-k & \text{if } k \leq \lceil n/2 \rceil \end{cases}$

If n is even, each term from $T(\lceil n/2 \rceil)$ upto $T(n-1)$ appears exactly twice in the summation.

If n is odd, all these terms appear twice & $T(\lfloor n/2 \rfloor)$ appear once.

Thus we have

$$E[T(n)] \leq \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} E[T(k)] + O(n) \quad -$$

We solve this recurrence by substitution —

Let us assume that $T(n) \leq cn$ satisfies the given recurrence for some constant c.

We assume that $T(n) = O(1)$ for n less than some constant.

We also pick a constant a such that the function described by $O(n)$ term above is bounded from above by 'an' for all $n > 0$.

we have

$$T(n) \le \frac{2}{n} \sum_{k=\lfloor n/2 \rfloor}^{n-1} ck + an$$

$$= \frac{2c}{n} \left(\sum_{k=1}^{n-1} k - \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k \right) + an$$

$$= \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{(\lfloor n/2 \rfloor)(\lfloor n/2 \rfloor - 1)}{2} \right) + an$$

$$\le \frac{2c}{n} \left(\frac{n(n-1)}{2} - \frac{(n/2 - 2)(n/2 - 1)}{2} \right) + an$$

$$= \frac{2c}{n} \left(\frac{n^2 - n}{2} - \frac{n^2/4 - 3n/2 + 2}{2} \right) + an$$

$$= \frac{c}{n} \left(\frac{3n^2}{4} + \frac{n}{2} - 2 \right) + an$$

$$= c \left(\frac{3n}{4} + \frac{1}{2} - 2 \right) + an$$

$$\le \frac{3cn}{4} + \frac{c}{2} + an$$

$$= cn - \left(\frac{cn}{4} - \frac{c}{2} - an \right)$$

If we want to prove that the last expression is at most cn, we must have

$$\frac{cn}{4} - \frac{c}{2} - an \ge 0$$

$$\Rightarrow \frac{cn}{4} - an \ge \frac{c}{2}$$

$$\Rightarrow n \left(\frac{c}{4} - a \right) \ge \frac{c}{2}$$

$$\Rightarrow n \ge \frac{2c}{c - 4a}$$

So, if we assume that $T(n) = O(1)$ for $n < 2c'/(c - 4a)$, we have $T(n) = O(1)$ for $n < 2c/(c - 4a)$ we have

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

⇒ Thus, we can conclude that any order statistic, & in particular the median can be determined on average in linear time.

Similarly, the no. of elements that are less than x is at least $\frac{3n}{10} - 6$.

$\Rightarrow$ Thus, in the worst case, SELECT is called recursively on at most $\frac{7n}{10} + 6$ elements in step 5.

Steps 1, 2 & 4 take $O(n)$ time.
Step 3 takes time $T(\lceil n/5 \rceil)$
Step 5 takes time at most $T\left(7n/10 + 6\right)$

We assume that an input of 140 or fewer elements require $O(1)$ time.
Thus,

$$T(n) \leq \begin{cases} \Theta(1) & \text{if } n \leq 140 \\ T(\lceil n/5 \rceil) + T\left(7n/10 + 6\right) + O(n) & \text{if } n > 140 \end{cases}$$

We will show that $T(n) \leq cn$ for some large constant c & all $n > 0$.
We assume that $T(n) \leq cn$ for all $n \leq 140$

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c\left(7n/10 + 6\right) + an \\ &\leq cn/5 + c + 7cn/10 + 6c + an \\ &= \frac{9cn}{10} + 7c + an \qquad (7c + 6c = 6c) \\ &= cn + (an + 6c - cn/10) \end{aligned}$$

which is at most cn if

$$-\frac{cn}{10} + 6c + an \leq 10$$

Since n
Choosing $c \geq 20$ a will satisfy the inequality
The worst case running time of SELECT is therefore linear.

Strassen Matrix Multiplication

Divide-and-Conquer algorithsm for matrix multiplication

$$A = \begin{matrix} A^{11} & A^{12} \\ A^{21} & A^{22} \end{matrix} \quad B = \begin{matrix} B^{11} & B^{12} \\ B^{21} & B^{22} \end{matrix} \quad C = A \times B = \begin{matrix} C^{11} & C^{12} \\ C^{21} & C^{22} \end{matrix}$$

Formulas for C^{11}, C^{12}, C^{21}, C^{22}:

$$C^{11} = A^{11}B^{11} + A^{12}B^{21} \qquad C^{12} = A^{11}B^{12} + A^{12}B^{22}$$

$$C^{21} = A^{21}B^{11} + A^{22}B^{21} \qquad C^{22} = A^{21}B^{12} + A^{22}B^{22}$$

The First Attempt Straightforward from the formulas above (assuming that N is a power of 2):

MMult(A, B, N)

- If $N = 1$ Output $A \times B$

* Else

3. Compute $A^{11}, B^{11}, \ldots, A^{22}, B^{22}$ % by computing $M = N/2$

- $X_1 \leftarrow MMULT(A^{11}, B^{11}, N/2)$
- $X_2 \leftarrow MMULT(A^{12}, B^{21}, N/2)$
- $X_3 \leftarrow MMULT(A^{11}, B^{12}, N/2)$
- $X_4 \leftarrow MMULT(A^{12}, B^{22}, N/2)$
- $X_5 \leftarrow MMULT(A^{21}, B^{11}, N/2)$
- $X_6 \leftarrow MMULT(A^{21}, B^{21}, N/2)$
- $X_7 \leftarrow MMULT(A^{21}, B^{12}, N/2)$
- $X_8 \leftarrow MMULT(A^{22}, B^{22}, N/2)$
- $C^{11} \leftarrow X_1 + X_2$
- $C^{12} \leftarrow X_3 + X_4$
- $C^{21} \leftarrow X_5 + X_6$
- $C^{22} \leftarrow X_7 + X_8$
- Output C
- End If

Analysis: The operations on line 3 take constant time. The combining cost (lines 12–15) is $\Theta(N^2)$ (adding two $\frac{N}{2} \times \frac{N}{2}$ matrices takes time $\frac{N^2}{4} = \Theta(N^2)$). There are 8 recursive calls (lines 4–11). So let $T(N)$ be the total number of mathematical operations performed by MMult(A, B, N), then

$$T(N) = 8T\left(\frac{N}{2}\right) + \Theta(N^2)$$

The Master Theorem gives us

$$T(N) = \Theta(N^{\log_2(8)}) = \Theta(N^3)$$

So this is not an improvement on the "obvious" algorithm given earlier (that uses N^3 operations).

Strassen's algorithm is based on the following observation:

$$C^{11} = P_5 + P_4 - P_2 + P_6 \qquad C^{12} = P_1 + P_2$$
$$C^{21} = P_3 + P_4 \qquad C^{22} = P_1 + P_5 - P_3 - P_7$$

where

$$P_1 = A^{11}(B^{12} - B^{22})$$
$$P_2 = (A^{11} + A^{12})B^{22}$$
$$P_3 = (A^{21} + A^{22})B^{11}$$
$$P_4 = A^{22}(B^{21} - B^{11})$$
$$P_5 = (A^{11} + A^{22})(B^{11} + B^{22})$$
$$P_6 = (A^{12} - A^{22})(B^{21} + B^{22})$$
$$P_7 = (A^{11} - A^{21})(B^{11} + B^{12})$$

Exercise Verify that $C^{11}, \ldots, C^{22}$ can be computed as above.

The above formulas can be used to compute $A \times B$ recursively as follows:

Strassen(A, B)

1. If $N = 1$ Output $A \times B$
2. Else
3. Compute $A^{11}, B^{11}, \ldots, A^{22}, B^{22}$ % by computing $M = N/2$
4. $P_1 \leftarrow$ STRASSEN$(A^{11}, B^{12} - B^{22})$
5. $P_2 \leftarrow$ STRASSEN$(A^{11} + A^{12}, B^{22})$
6. $P_3 \leftarrow$ STRASSEN$(A^{21} + A^{22}, B^{11})$
7. $P_4 \leftarrow$ STRASSEN$(A^{22}, B^{21} - B^{11})$
8. $P_5 \leftarrow$ STRASSEN$(A^{11} + A^{22}, B^{11} + B^{22})$
9. $P_6 \leftarrow$ STRASSEN$(A^{12} - A^{22}, B^{21} + B^{22})$
10. $P_7 \leftarrow$ STRASSEN$(A^{11} - A^{21}, B^{11} + B^{12})$
11. $C^{11} \leftarrow P_5 + P_4 - P_2 + P_6$
12. $C^{12} \leftarrow P_1 + P_2$
13. $C^{21} \leftarrow P_3 + P_4$
14. $C^{22} \leftarrow P_1 + P_5 - P_3 - P_7$
15. Output C
16. End If

Analysis: The operations on line 3 take constant time. The combining cost (lines 11–14) is $\Theta(N^2)$. There are 7 recursive calls (lines 4–10). So let $T(N)$ be the total number of mathematical operations performed by Strassen(A, B), then

$$T(N) = 7T\left(\frac{N}{2}\right) + \Theta(N^2)$$

The Master Theorem gives us

$$T(N) = \Theta(N^{\log_2(7)}) = \Theta(N^{2.8})$$

The best current upper bound for multiplying two matrices of size $N \times N$ is $O(N^{2.32})$ (by using similar idea, but instead of dividing a matrix into 4 quarters, people divide them into a bigger number of submatrices).