BRUTE FORCE

unit - 3

There are several methods to design an algorithm for a given problem. Brute-Force is one such technique. It is a straight forward method to solve a given problem based on the statement of the problem and definitions of the concepts involved. For example, if we want to compute x^n then without any other work, we will just multiply x by itself for n number of times. The definition based matrix multiplication algorithm, the consecutive integer checking algorithm for finding gcd etc are few examples of brute force.

We may come accross a question - in which situation, the brute force method is applicable? To answer, we can consider the following facts -

* Brute force algorithm is useful for solving small-size instances of a problem.
* It is useful for many important algorithms such as computing the sum of n numbers, finding largest element in list etc.

* For the problems like searching, sorting, matrix multiplication, string matching etc, the brute-force method results in reasonable algorithms of some practical value with no limitation on input size.
* Suppose, only few instances of a problem must be solved. Then, there is no meaning in designing most efficient algorithm by spending much time. In such case, brute-force gives a simple methodology to solve the problem with reasonable speed.
* Brute-force can be used as a yard-stick with which other alternate problems can be solved & efficient algorithms can be choosen.

Now, we will discuss some of the algorithms that designed using brute-force technique.

Selection Sort :

This is a sorting algorithm where we compare the first element of the list with all other elements. If first element is found to be greater than the compared element, then they are exchanged. In the second iteration the second element -

is compared with all other elements starting from third element. The process continues till the sorted list is available.
So, if there are n elements in the list, there will be $n-1$ iterations. Let us number the iterations from 0 to $n-2$. Then, at the i^{th} iteration, the $(n-i-1)^{th}$ element is compared with the last $(n-i)$ elements and exchange happens if required.

The algorithm is as below –

```
ALGORITHM Selection(A[0..n-1])
// Sorting a list by selection sort
// Input: An array A[0..n-1]
// Output: Array A[0..n-1] in ascending order
for i ← 0 to n-2 do
    pos ← i
    for j ← i+1 to n-1 do
        if A[j] < A[pos]
            pos ← j
    swap A[i] and A[pos].
```

Chetana Hegde
9448301894

Analysis:

1. The parameter is obviously, the input size n.
2. The basic operation is comparison.
3. The number of times the basic operation gets executed depends only on input size.

So,

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [n-1-(i+1)+1]$$

$$= \sum_{i=0}^{n-2} (n-i-1)$$

$$= n \sum_{i=0}^{n-2} 1 - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1$$

$$= n(n-2-0+1) - \frac{(n-2)(n-1)}{2} - (n-2-0+1)$$

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

$$= (n-1)\left\{ n - \frac{n-2}{2} - 1 \right\}$$

$$= (n-1)\left\{ \frac{n}{2} \right\}$$

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

$$\therefore C(n) \in \Theta(n^2).$$

[NOTE: Find proper c_1, c_2 and n_0].

Bubble Sort :-

In this algorithm, the first element is compared with the second element and if first element is greater than second, they are exchanged. Then second element is compared with the third & so on. At the end of first iteration, the largest element will be placed in its proper position. In the second iteration again we start with comparing first and second & so on till the last-but-one element. The process is continued till we get the sorted list.

Thus, for a list of n elements, we need (n-1) iterations. The algorithm is as below –

```
ALGORITHM Bubble (A[0..n-1])
// Sorting an array using bubble sort method.
// Input : An array A[0...n-1]
// Output : A sorted array.
for i ← 0 to n-2 do
    for j ← 0 to n-2-i do
        if A[j+1] < A[j]
            swap A[j] and A[j+1]
```

Analysis:

1. The parameter is the input size n.
2. The basic operation is comparison.
3. The basic operation depends only on the input size n.

And, there is one comparison for each value of i and for each value of j.

$$\therefore C(n) = \sum_{i=0}^{n-2} \sum_{j=0}^{n-2-i} 1$$

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

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{n(n-1)}{2} \quad \text{(As in selection sort)}.$$

Chetana Hegde
9448830189

$$\therefore C(n) \in \Theta(n^2)$$

<u>NOTE</u>: We can observe that the complexity of both selection sort and bubble sort algorithm is $\Theta(n^2)$. But, in the worst case, selection sort requires $(n-1)$ exchanges and bubble sort requires $\frac{n(n-1)}{2}$ exchanges. Thus, there is a difference between these algorithms.

Sequential Search

We have discussed the algorithm and analysis of sequential search. In that, the key element is compared with every element till ~~either~~ match found or list is terminated. We can change the algorithm slightly by putting the key element at the end of existing list so that it always be a successful search. The algorithm is as below –

ALGORITHM Sequential(A[0..n], K)
// Sequential search by putting key at the end.
// Input: Array of n elements & key K.
// Output: The position of the first element
//   in A[0..n-1] whose value is equal
//   to K or -1 if no such element exists.

```
A[n] ← K
i ← 0
while A[i] ≠ K do
    i ← i+1
if i < n
    return i
else
    return -1
```

String Matching

This problem involves with ~~predicting~~ checking whether one particular string exists in other string. i.e. We will consider one string of length 'n' and call it as 'text'. Then take one more string of length 'm' (where $m \leq n$) and call it as 'pattern'. Then our problem is to check whether pattern exists in text or not. The procedure for this is as below—

Take the first character of 'text' and the first character of 'pattern'. If they matches, consider second character of 'text' and second character of 'pattern'. If they matches, take third pair & so on. Do this process until either all the character matches or an unmatched pair is found. If there is an unmatched pair, the start the process by taking second character of 'text' and first character of 'pattern', third character of 'text' and second character of 'pattern' & so on. Thus, at the i^{th} step, we will be comparing-

$$t_i = p_0, \quad t_{i+1} = p_1, \dots, t_{i+j} = p_j, \dots, t_{i+m-1} = p_{m-1}.$$

(where, t_i is i^{th} character of 'text' and
p_j is j^{th} character of pattern.)

<u>Note</u> that for the 'text' of length n and the 'pattern' of length m, we need to consider only first (n-m) characters of 'text'. Because, after (n-m) have been completed in text, there will not be sufficient number (i.e. m) of characters remaining in the 'text' to match with 'pattern'. ~~The~~ So, there is no meaning in checking thereafter.

Consider an example to illustrate this-

Let our text be-

~~Hello, How are you?~~
~~$t_0\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ t_8\ t_9\ t_{10}\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t_{16}$~~

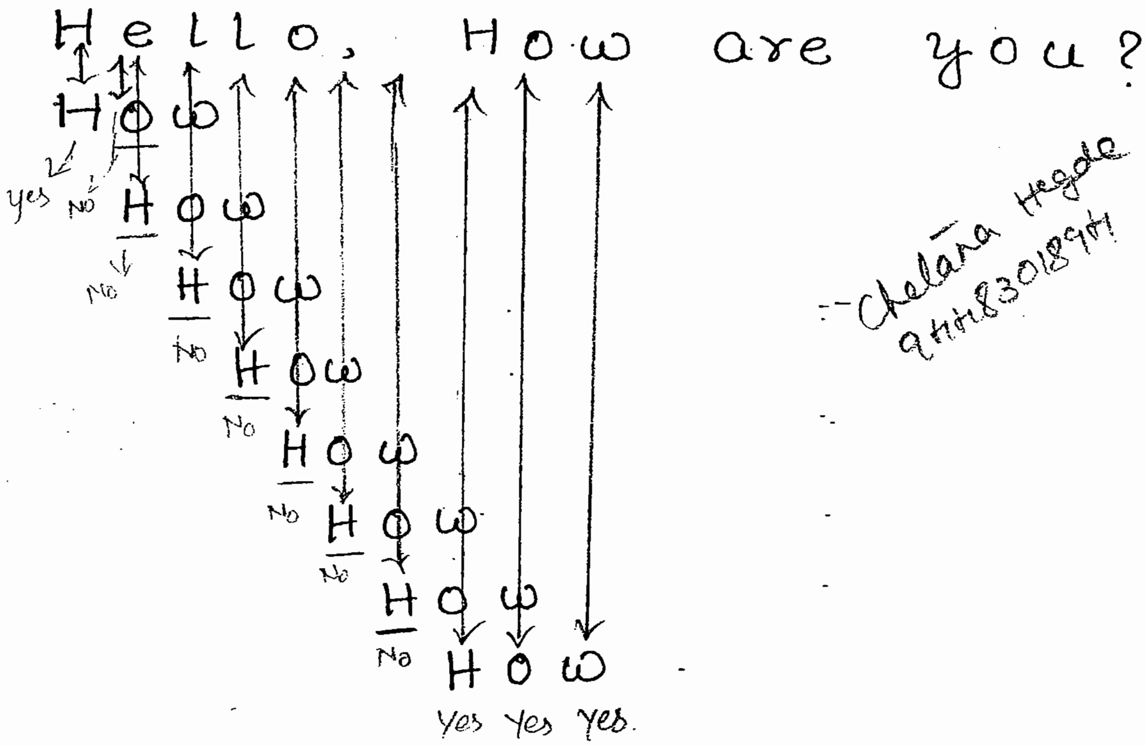
Hello, How are you?
$t_0\ t_1\ t_2\ t_3\ t_4\ t_5\ t_6\ t_7\ t_8\ t_9\ t_{10}\ t_{11}\ t_{12}\ t_{13}\ t_{14}\ t_{15}\ t_{16}\ t_{17}\ t_{18}$

Let the pattern be -

How
$p_0 \; p_1 \; p_2$

So, $n = 19$ and $m = 3$.

The procedure is as shown —



-- Chetana Hegde
9448301894

Thus, by this straight-forward method we can check whether the given pattern exists in a text or not. If match is found, the position of the first character in the text that starts the first matching substring is returned otherwise, -1 is returned.

ALGORITHM StringMatch(T[0..n-1], P[0..m-1])

// Implements string matching by brute-force

// Input: An array T[0..n-1] of length 'n' is text.

// An array P[0..m-1] of length m is pattern.

// Output: The position of first character in the

// text that starts the first matching

// substring if successful, & -1 otherwise

```
for i ← 0 to n-m do
    j ← 0
    while j<m and P[j] == T[i+j] do
        j ← j+1
    if j == m
        return i
return -1.
```

~~Analysis (Worst-Case)~~

~~1. There are two parameters m & n.~~

~~2. The basic operation is comparison~~

Analysis :-

1. There are two parameters m & n.
2. The basic operation is comparison.

3. As the time complexity of the algorithm not only depends on the input size m and n, but it depends on the position of occurence of pattern in text or the non-occurence also. So, we go for both best case and worst case analysis.

Best-Case: If the pattern of length m is found in the text at the very first position only i.e if first m characters of the text matches with the pattern, then that will be best possibility. For this reason, we have to compare first m characters.

So, $C_{best}(n) = m$

As $C_{best}(n) \nless m$, we can easily say that

$C_{best}(n) \geq m$

$\therefore C_{best}(n) \in \Omega(m)$.

i.e The time complexity in best case analysis of string matching is m.

Worst-Case:

In the algorithm, the comparison of P[j] and T[i+j] is done once for

each value of j starting from 0 to m.
Moreover, this is done for each value of i ranging from 0 to n-m.

Thus,

$$C(n) = \sum_{i=0}^{n-m-1} \sum_{j=0}^{m-1} 1$$

$$= \sum_{i=0}^{n-m-1} (m-1-0+1)$$

$$= \sum_{i=0}^{n-m-1} m$$

$$= m \sum_{i=0}^{n-m-1} 1$$

$$= m \cdot (n-m-1-0+1)$$

$$= m(n-m)$$

$$= mn - m^2$$

$$\simeq mn, \quad \text{for } n >> m.$$

Thus, $C(n) \in \Theta(mn)$,
i.e. time complexity of string matching at worst case is of order mn.

Chetana Hegde
9448301894

Exhaustive Search

The problems involving combinatorial objects such as permutations, combinations, subsets of a set etc. requires finding an element with a special property in a domain that grows exponentially with an instance size. Such problems will usually be optimization problems i.e. finding out the element that maximizes or minimizes some characteristic like transportation cost, profit after production etc.

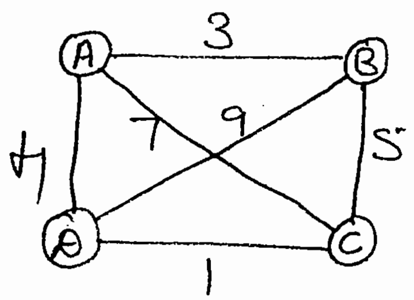
Exhaustive search is a brute-force technique for solving combinatorial problems. This technique generates each and every element of the problem's domain that are satisfying the constraints of the problem. Then the desired element as per the requirement of the problem will be selected. The implementation of exhaustive search requires an algorithm for generating certain combinatorial objects. But few of the problems that uses exhaustive search are discussed here viz. travelling salesman problem, knapsack problem and assignment problem.

Travelling Salesman Problem :-

This problem involves finding the shortest path through 'n' cities that visits each city exactly once before returning to the starting city. This can be thought of as finding the shortest Hamiltonian circuit of a weighted graph, where vertices of the graph being cities and the distance between the cities is considered as weights.

We know that a Hamiltonian circuit is a sequence of $n+1$ vertices $v_0 v_1 \ldots v_n v_0$ with first and last vertices of the sequence being same and other $n-1$ vertices are being distinct.

Consider an example of TSP which can be solved by exhaustive search technique. Below shown is a graph representing the cities A, B, C and with their mutual distances -



~~If we assume that the salesman starts~~ with the city A, then the possible routes with the distances are given below—

$A \to B \to C \to D \to A$ dist. = 13

$A \to B \to D \to C \to A$ dist. = 20

$A \to D \to C \to B \to A$ dist. = 13

$A \to D \to B \to C \to A$ dist. = 25

$A \to C \to B \to D \to A$ dist. = 25

$A \to C \to D \to B \to A$ dist. = 20

Having all possible routes from A we can observe that the first and third routes results in optimal distance i.e. 13. So, salesperson can opt either

$A \to B \to C \to D \to A$ or

$A \to D \to C \to B \to A$.

It can easily observed that there will be $(n-1)!$ permutations or routes for n cities. So, the time complexity of TSP is of order $(n-1)!$, as we will choose optimal route only after finding all these $(n-1)!$ routes. Thus, $C(n) \in O((n-1)!)$

Knapsack Problem

The problem is stated as— There is a knapsack of capacity W. There are 'n' items of weights $w_1, w_2, \ldots, w_n$ and values $v_1, \ldots v_n$. We have to find the most valuable subset of the items that fit into the knapsack.
i.e. our problem is to

maximize $\sum_{i=1}^{n} v_i a_i$

subject to the constraint - $\sum_{i=1}^{n} w_i \cdot a_i \le W$.

Here a_i denotes the i^{th} item to be put into the knapsack.

To illustrate, consider an example—

Let the capacity of knapsack, W = 30
Let there be three items (i.e n=3) A, B & C.
Let their weights be 18, 18 and 22.
Let their values be 41, 28 & 50.
Now, we have to find the subset of a set {A, B, C} so that-

* total weight ~~must~~ can be at the most 30
* total value of selected items is maximum.

The exhaustive search approach to this problem suggests to pick-up all possible subsets of the given set & to compute the total weights of all these subsets with the values & then select the feasible subset.

So, for the given example, we proceed as given below–

Subsets	Total weight	Total value	Feasible ?
{ }	0	0	Yes
{A}	18	41	Yes
{B}	8	28	Yes
{C}	22	50	Yes
{A, B}	18+8=26	41+28=69	Yes
{A, C}	18+22=40	41+50=91	No
{B, C}	8+22=30	28+50=78	Yes
{A, B, C}	18+8+22 =48	41+28+50 =119	No

Here, the weights for the subsets {A, C} and {A, B, C} are exceeding the capacity of the knapsack. So, they are not feasible.

Out of other remaining feasible subsets, the one with maximum value is the subset $\{B, C\}$, which is the solution of the problem.

Note that for any knapsack problem with n items, we have to find out the possible subsets. As we know, there will be 2^n subsets for a set containing n elements. So, for solving knapsack problem, we have to find out 2^n subsets. This means that the time complexity of this algorithm is $\Omega(2^n)$.

Chetana Hegde
9448301894

Assignment Problem

Assignment problem involves assigning n different jobs to n different people. There will be cost incurred for assigning a job to a person. We have to assign the jobs so that the total cost is minimum. The cost incurred for assigning i^{th} person to the j^{th} job is denoted by C_{ij}, for $i = 1, 2, \ldots, n$ & $j = 1, 2, \ldots, n$

~~Consider one example for illustration-~~

So, our problem is to

minimize $\sum_{i=1}^{n}\sum_{j=1}^{n} c_{ij}\, x_{ij}$

Subjected to constraints -

$$x_{ij} = \begin{cases} 1, & \text{if } i^{th} \text{ person is assigned } j^{th} \text{ job} \\ 0, & \text{otherwise.} \end{cases}$$

& $\sum_{j=1}^{n} x_{ij} = 1$, i.e. exactly one job is done by i^{th} person.

Now, consider one example for illustration -

	J1	J2	J3	J4
P1	7	10	13	8
P2	9	2	5	12
P3	3	15	4	9
P4	13	5	10	7

Here, J1, J2, J3, J4 are jobs and P1, .., P4 are persons. The values are costs & such a matrix is known as cost matrix.

The exhaustive search techniques says that we have ~~to~~ to find out ~~several~~ all possible n-tuples ~~of~~ like (J1, ..., Jn) for a problem of n jobs. Then find out the cost incurred for all these tuples and select the one feasible tuple which results in minimum cost.

Hence, for the example, we have tuples like-

(J1, J2, J3, J4) $\longrightarrow$ $7+2+4+7 = 20$

(J1, J2, J4, J3) $\rightarrow$ $7+2+9+10 = 28$

(J1, J3, J2, J4) $\rightarrow$ $7+5+15+7 = 34$

(J1, J3, J4, J2) $\longrightarrow$ $7+5+9+5 = 26$

(J1, J4, J2, J3) $\rightarrow$ $7+12+15+10 = 44$

(J1, J4, J3, J2) $\rightarrow$ $7+12+4+5 = 28$

⋮ etc ⋮

As, in this example we are having 4 jobs, there will be $4! = 24$ tuples. We have to find one sequence with minimum cost.

Thus, in general, to get feasible solution for an assignment problem with n jobs, we have to find out n! number of permutations or n-tuples. Hence, the time complexity of the assignment problem will be $O(n!)$.

NOTE: Apart from expensive algorithm like exhaustive search technique, we have 'Hungarian method' for solving assignment problem.

Unit II

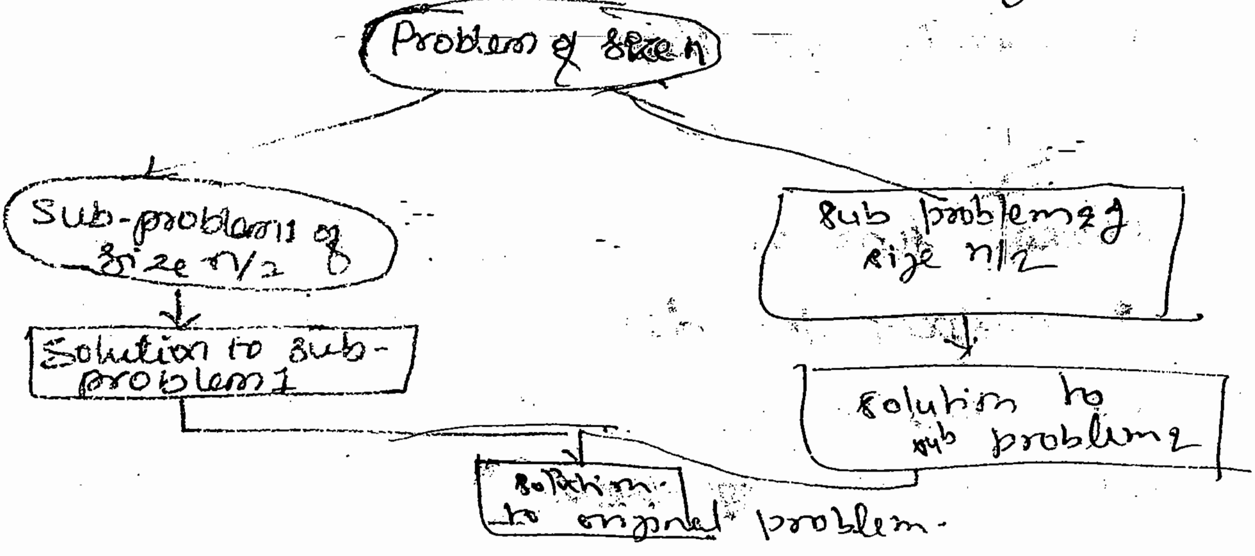
(3) DIVIDE-AND-CONQUER

Chetana Hegde
9448301894

We know that some of the problems can be straight-away solved using brute-force technique. But, in many cases, brute-force fails. So, let us study the problems which falls under divide-and-conquer. The general plan for this technique is as below –

1. An instance of a given problem is divided into several smaller instances of same type of problem and of equal size.
2. These smaller problems are solved, usually by recursive method.
3. The solutions of all these sub problems are combined to get the solution of original problem.

The pictorial representation can be given as –



Note that, even though, divide-and-conquer is one of the best design techniques, it is not necessarily more efficient than brute-force in many of the situations. But, if the problem suits the criteria of DAC, then definitely, it will yield an efficient algorithm.

Note also that, DAC is ideally suited for parallel ~~algo~~ computation problems, ~~we~~ even though, we apply the technique on sequential problems.

For finding the time complexity of any DAC problem, we will proceed as further. Assume that a problem of size 'n' is divided into 'a' number of subproblems each of size 'n/b'. Then, the time complexity function is given by-

$$T(n) = a.T(n/b) + f(n),$$

where f(n) is a function denoting the time spent for dividing the problem into subproblems and for combining the solutions of these.

The above relation is known as *general divide-and-conquer recurrence*. It is easily observed that $T(n)$ depends on the values of constants 'a' and 'b' and also on the order of growth of $f(n)$. This is observed in a theorem as—

Master Theorem: If $f(n) \in \Theta(n^d)$, $d \geq 0$ in the recurrence relation, $T(n) = a.T(n/b) + f(n)$, then,

$$T(n) \in \begin{cases} \Theta(n^d) & , \text{ if } a < b^d \\ \Theta(n^d . \log n) & , \text{ if } a = b^d \\ \Theta(n^{\log_b a}) & , \text{ if } a > b^d. \end{cases}$$

NOTE: The above results hold for O and Ω notations also.

Note also that, the above theorem will give only the order of growth of $T(n)$. If we want exact time complexity, then we must go for substitution method only.

To illustrate the use of above theorem let us consider a simple example of finding the sum of n numbers. If we apply DAC technique in this

It says that the problem is divided into two parts ~~to~~ each with $n/2$ (if n is even, otherwise sizes will be $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$) elements then ~~to~~ solve the problem. Finally, two sums are added to get the solution. So, the recurrence relation is given by—

$$T(n) = \begin{cases} 0 & , \text{ if } n = 1 \\ T(n/2) + T(n/2) + 1 & , \text{ otherwise.} \end{cases}$$

Here, two $T(n/2)$ terms indicates the time required for finding the sum of first half ~~and~~ elements & second half elements. The term 1 is for ~~final~~ adding two sums finally.

Now, if we use, substitution method—

$$\begin{aligned}
T(n) &= 2 \cdot T(n/2) + 1 \\
&= 2 \cdot \{2 \cdot T(n/4) + 1\} + 1 \\
&= 2^2 \cdot T(n/2^2) + 2 + 1 \\
&= 2^2 \cdot \{2 \cdot T(n/2^3) + 1\} + 2 + 1 \\
&= 2^3 \cdot T(n/2^3) + 2^2 + 2 + 1 \\
&= \cdots \\
&= 2^k \cdot T(n/2^k) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1
\end{aligned}$$

~~As, for solving recurrence relations,~~ we assume $n = 2^k$, always, we will get-

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

$$= 2^k . T(1) + 2^{k-1} + 2^{k-2} + \dots + 2 + 1$$

$$= 2^k . 0 + 2^{k-1} + \dots + 2 + 1$$

$$= \frac{1.(2^{k}-1)}{2-1}$$

Chetana Hegde
9448301894

$$= 2^k - 1$$

$$= n - 1$$

$\simeq n$, for large value of n.

$\therefore T(n) \in \Theta(n)$.

Now, let us apply master theorem on the equation,

$$T(n) = \begin{cases} 0 & , \quad n = 1 \\ T(n/2) + T(n/2) + 1 & , \quad \text{otherwise.} \end{cases}$$

i.e. $T(n) = 2 . T(n/2) + 1$. (Here, $f(n) = 1 \in \Theta(n^0)$)

Here, $a = 2$, $b = 2$ and $d = 0$

By applying $a > b^d$

i.e. $2 > 2^0$

we will get,

~~$T(n) \in \Theta(n^{\log_2 2}$~~

$T(n) \in \Theta(n^{\log_2 2}) \Rightarrow T(n) \in \Theta(n$

Merge Sort :-

Merge sort is a best example of DAC technique. This sorting technique sorts a given array A[0.. n-1] by dividing it into two halves A[0.. ⌊n/2⌋-1] and A[⌊n/2⌋.. n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted array. The merging of two sorted arrays can be done as below- Two indexes are initialized to point the first elements of two arrays. Now their positional values are compared and smaller element is copied into a third resulting array. Now, the index of the array from which we copied an element, is increased by one position and again comparison is done between two arrays. This process is continued till either of two arrays is completed. Then the remaining elements of non-completed array are copied into resulting array.

ALGORITHM Mergesort$(A[0..n-1])$
// Sorts array $A[0..n-1]$ by recursive mergesort
// Input : Array $A[0..n-1]$.
// Output : Array $A[0..n-1]$ of sorted.

```
if n > 1
    copy A[0.. ⌊n/2⌋ - 1] to B[0.. ⌊n/2⌋ - 1]
    Copy A[⌊n/2⌋ .. n-1] to C[0.. ⌈n/2⌉ - 1]
    MergeSort (B[0.. ⌊n/2⌋ - 1])
    MergeSort (C[0.. ⌈n/2⌉ - 1])
    Merge (B, C, A)
```

ALGORITHM Merge$(B[0..p-1], C[0..q-1], A[0..p+q-1])$
// Merges two sorted arrays into one sorted array.
// Input : Two sorted lists $B[0..p-1]$ & $C[0..q-1]$.
// Output : Sorted list $A[0..p+q-1]$

```
i ← 0
j ← 0
k ← 0
while i < p and j < q do
    if B[i] ≤ C[j]
        A[k] ← B[i]
        i ← i+1
    else
        A[k] ← C[j]
        j ← j+1

    k ← k+1
```

if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else
    copy $B[i..p-1]$ to $A[k..p+q-1]$

Analysis:

1. The parameter is n.
2. The basic operation is comparison.

The recurrence relation can be given as-

$$C(n) = \begin{cases} 0 & , \text{ if } n=1 \\ C(n/2) + C(n/2) + C_{merge}(n) & , n > 1 \end{cases}$$

Here, two terms $C(n/2)$ indicates the time required for sorting two halves of the given array and $C_{merge}(n)$ denotes the time required for merging two arrays. During merge process, at every step, there is one comparison i.e $B[i] \le C[j]$. After each comparison, the elements to be processed is reduced by one. In the worst case, neither of two arrays becomes empty before the other one contains just one element.

This happens when smaller elements come from the alternating arrays. Thus, in such situation,

$$C_{merge}(n) = n-1.$$

Chetana Hegde
9448301894

So, we have-

$$C(n) = \begin{cases} 0 & , \quad n=1 \\ 2.C(n/2) + n-1 & , \quad n>1. \end{cases}$$

For the sake of simplicity, we assume $n = 2^k$ & proceed as further-

$$C(n) = 2.C(n/2) + n-1$$

$$= 2.\{2.C(n/4) + \frac{n}{2} - 1\} + (n-1)$$

$$= 2^2.C(n/2^2) + 2(n/2 - 1) + (n-1)$$

$$= 2^2.\{2.C(n/2^3) + (\frac{n}{4} - 1)\} + 2(n/2 - 1) + (n-1)$$

$$= 2^3.C(n/2^3) + 2^2(\frac{n}{2^2} - 1) + 2(n/2 - 1) + (n-1)$$

$$\vdots$$

$$= 2^k.C(n/2^k) + 2^{k-1}(\frac{n}{2^{k-1}} - 1) + \cdots + 2(n/2 - 1) + (n-1)$$

$$= 2^k.C(1) + (\underbrace{n + n + \cdots + n}_{k \text{ times}}) - \{2^{k-1} + 2^{k-2} + \cdots + 2^2 + 2^1 + 2^0\}$$

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

$= nk - (n-1)$

$= n(k-1) + 1$

$C(n) \approx k.n$ for large n.

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

$\therefore C(n) \approx n.\log_2 n$

$\therefore \boxed{C(n) \in \Theta(n \log_2 n)}$

NOTE: If we solve the recurrence relation :-

$$C(n) = \begin{cases} 0 & ; \quad n=1 \\ 2.C(n/2) + (n-1) & ; \quad n>1 \end{cases}$$

using master theorem, then -

$a = 2$, $b = 2$ & $f(n) = n-1$.

For large n, $f(n) = n-1$
$\approx n$

$\therefore$ we can take $f(n) \in \Theta(n^1)$, so that $d = 1$.

$\therefore b^d = 2^1 = 2$

$\Rightarrow a = b^d$

$\therefore C(n) \in \Theta(n^1.\log_2 n)$

$C(n) \in \Theta(n \log n)$

Quick Sort :-

It is a technique that devides the given array into based on the values of elements. After such a partition, all the elements before one particular element called pivot, are less than pivot and all other elements after pivot are greater than the pivot. The technique is again imposed on these two sub arrays, as the position of pivot is already fixed. The process is continued till the entire array gets sorted.

Thus, after first partition, the array of n elements may look like this -

$$\underbrace{A[0], \ldots, A[s-1]}_{\text{smaller than } A[s]},\ A[s],\ \underbrace{A[s+1] \ldots A[n-1]}_{>= A[s]}$$

The procedure of partitioning the given array is as explained below -

ⓐ Usually, the first element of the array is treated as key. The position of second element will be first index variable i and the position of last element will be the index variable j.

ⓑ Now, the index variable i is incre... by one till the value stored at ...

position i is greater than the key element.

(c) Similarly, j is decremented by one till the value stored at j is smaller than the pivot.

(d) Now, the two elements A[i] & A[j] are interchanged. Again ~~from~~ the current positions of i and j are incremented and decremented respectively and exchanges are made appropriately if required.

(e) This process ends when the index pointers meet or crossover.

(f) Now, the whole array is divided into two parts such that one part is containing the elements less than the pivot and the other part containing the elements greater than the pivot

(g) The above procedure is applied on both the sub-arrays. At the end, each sub array will be containing one element and by that time, the given array will be sorted.

ALGORITHM QuickSort(A[l..r])
// Sorts a subarray by quicksort
// Input: A subarray A[l..r] of A[0..n-1]
// Output: The subarray A[l..r] sorted in ascending order.

```
if l < r
    s ← Partition(A[l..r])
    QuickSort(A[l..s-1])
    QuickSort(A[s+1..r])
```

ALGORITHM Partition(A[l..r])
// Partitions a subarray by using first element as pivot.
// Input: A subarray A[l..r] of A[0..n-1], l < r.
// Output: A partition of A[l..r], with the split
// position as returned value.

```
p ← A[l]
i ← l;
j ← r+1
repeat
    repeat i ← i+1
        until A[i] ≥ p
    repeat j ← j-1
        until A[j] ≤ p
    swap(A[i], A[j])
until i ≥ j
swap(A[i], A[j])
swap(A[l], A[j])
return j
```

Chetana Hegde
9448301894

Analysis:

1. The parameter is input size n.
2. The basic operation is comparison of pivot with other positional elements.
3. The time complexity depends not only on n but also on the value of pivot element. Because, after partition, where exactly the pivot lies or what will be the sizes of subarrays plays an important role. So, we have to consider various efficiencies.

Best Case:

When the partition algorithm divides array into two equal parts i.e. the pivot element will be placed exactly at the middle of the array, then ~~hat~~ that will be the best situation.

So, if $C_b(n)$ is the time taken for an array of n elements,

The number of comparisons made before partition.

$$C_{best}(n) = \begin{cases} C(n/2) + C(n/2) + n, & n > 1 \\ 0, & n = 1 \end{cases}$$

$$\therefore C_{best}(n) = 2C(n/2) + n$$

$$= 2\{2\,C(n/4) + n/2\} + n$$

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

$$= 2^2 \{2 . C(n/8) + n/4\} + n + n$$

$$= 2^3 . C(n/2^3) + n + n + n$$

$$= \vdots$$

$$= 2^k C(n/2^k) + \underbrace{n + \dots + n}_{(k \text{ times})}$$

$$= 2^k . C(1) + nk$$

$$= n . \log_2 n \quad , \text{ as } 2^k = n \ \& \ C(1) = 0.$$

$$\therefore C_{best}(n) \in \Theta(n \log_2 n).$$

Worst-Case:

If the partition algorithm divides the given array into two parts of extremely different sizes, then that will be a worst case. That is, the pivot element will be at first position only, so that the subarray of smaller elements contains zero elements and the subarray of greater elements is of size $n-1$. Thus, by applying this logic on both the sub-arrays recursively, it is easily observed that worst case occurs when the given array is already sorted.

So, we will get —

$$
\begin{aligned}
C_{worst}(n) &= C(0) + C(n-1) + n, \quad n > 1 \\
&= C(n-1) + n \\
&= \{C(n-2) + n-1\} + n \\
&= C(n-3) + (n-2) + (n-1) + n \\
&= \vdots \\
&= C(n-n) + \cancel{(n-1)}\ 1 + 2 + \dots + (n-2) + (n-1) + n \\
&= 0 + 1 + 2 + \dots + n \\
&= \frac{n(n+1)}{2}
\end{aligned}
$$

$$\therefore C_{worst}(n) \approx n^2/2 \le n^2$$

$$\therefore C_{worst}(n) \in \Theta(n^2).$$

Chetana Hegde
9441830189

<u>Average Case:</u>

The ~~key~~ pivot element may be placed at any arbitrary position in the array. Then that will be the average situation. Consider that the pivot element is placed at the position k. Then, (k-1) elements are there in the left sub-array and (n-k) elements are there in the ~~...~~ right subarray. Now, if C(n) is the ime required for sorting entire array, C(k-1) & C(n-k) are the time required left & right subarrays respectively.

Thus, the average total time is given by -

$$C_{Avg}(n) = \frac{1}{n}\sum_{k=1}^{n}\left[C(k-1) + C(n-k)\right] + f(n),$$ for left and right subarray.

Here, $f(n)$ is no. of comparisons made before partition.

Now, we have to calculate the average time for comparison. If array is divided as two sub arrays of sizes 0 and $n-1$ then, total comparisons $= n$.

If array sizes are 1 & $n-2$, comparisons $= n-1$.

Continuing in this way, we get, the average number of comparisons as -

$$\frac{1}{n}\{n + (n-1) + (n-2) + \cdots + 3 + 2 + 1\}$$

$$\therefore f(n) = \frac{1}{n}\cdot\frac{n(n+1)}{2} = \frac{n+1}{2} \approx n+1$$

Considering $\frac{1}{2}$ as constant term & ignoring it, we get -

$$C_{Avg}(n) = \begin{cases} (n+1) + \frac{1}{n}\sum_{k=1}^{n}\left[C(k-1) + C(n-k)\right], & n > 1 \\ 0, & n = 0 \text{ or } 1 \end{cases}$$

Now, let us do the ~~forwo~~ backword substitution for the equation -

$$C_A(n) = (n+1) + \frac{1}{n}\sum_{k=1}^{n}\left[C(k-1) + C(n-k)\right]$$

$n \times$ Eq ① gives –

$$n.C(n) = n(n+1) + \sum_{k=1}^{n} [C(k-1) + C(n-k)]$$

$$= n(n+1) + [C(0) + C(1) + \dots + C(n-1)]$$
$$+ [C(n-1) + C(n-2) + \dots + C(1) + C(0)]$$

$$n.C(n) = n(n+1) + 2\{C(0) + C(1) + \dots + C(n-1)\} \quad \text{— (2)}$$

Replacing n by $(n-1)$ in the above equation –

$$(n-1).C(n-1) = (n-1).n + 2\{C(0) + C(1) + \dots + C(n-2)\} \quad \text{— (3)}$$

Now, ②–③ gives –

$$n.C(n) - (n-1).C(n-1)$$
$$= n^2 + n - n^2 + n + 2.C(n-1)$$
$$= 2n + 2.C(n-1)$$

$$\therefore n.C(n) = (n-1).C(n-1) + 2n + 2.C(n-1)$$
$$= C(n-1)\{n-1+2\} + 2n$$
$$n.C(n) = (n+1)\,C(n-1) + 2n \quad \text{— (4)}$$

Dividing the equation ④ by $n(n+1)$ we get –

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1} \quad \text{— (5)}$$

Replacing n by $(n-1)$ in the above equation we get -

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

Putting this value in (5) -

$$\frac{C(n)}{n+1} = \left\{\frac{C(n-2)}{n-1} + \frac{2}{n}\right\} + \frac{2}{n+1}$$

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

$$= \vdots$$

$$= \frac{C(n-n)}{1} + \frac{2}{2} + \frac{2}{3} + \cdots + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{C(0)}{1} + 2\left\{\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n+1}\right\}$$

$$= 0 + 2 \cdot \sum_{k=2}^{n+1} \frac{1}{k}$$

$$\therefore \frac{C(n)}{n+1} = 2\sum_{k=2}^{n+1} \frac{1}{k}$$

Here the functions $\sum_{k=2}^{n+1} \frac{1}{k}$ takes the discrete values.

$$\therefore \sum_{k=2}^{n+1} \frac{1}{k} \leq \int_2^{n+1} \frac{1}{k} \cdot dk$$

$$= \log_e k \Big|_2^{n+1}$$ {Note that base of log is 'e'}

$$\therefore \sum_{k=2}^{n+1} \frac{1}{k} \le (0.6930)\{\log_2(n+1) - \log_2 2\}$$

$$= (0.6930)\{\log_2(n+1) - 1\}$$

~~Thus~~, [To convert log from base 'e' to '2' we have to multiply with the factor 0.6930]

$$\cancel{C_A(n) \le 2(n+1)(1.4429)\{\log_2(n+1) - 1\}}$$

$$\cancel{= (2.8858)(n+1)\{\log_2(n+1) - 1\}}$$

$$\cancel{\approx n \log_2 n}$$, ~~for large value of n.~~

~~Thus $C_{Avg}(n) \in \Theta(n \log_2 n)$.~~

Thus, $C_A(n) \le 2(n+1)(0.6930)\{\log_2(n+1) - 1\}$

$$= (1.386)(n+1)\{\log_2(n+1) - 1\}$$

$$\approx (1.386)\, n.\log_2 n$$, for large n.

$$\therefore C_A(n) \in \Theta(n \log_2 n)$$

Chetana Hegde
9448301894

Binary Search:-

This is a very efficient searching algorithm on a sorted list. Here, the key element is searched with the middle element of the array. If they are equal, the position of middle element is returned and algorithm stops. If the key is found to be greater than the middle element, then the searching technique is applied on second half of the array, otherwise on the first half of the array. i.e.

$$\underbrace{A[0] \ldots A[m-1]}_{\substack{\text{search here} \\ \text{if } K < A[m]}} \quad \underset{\substack{\uparrow \\ K}}{A[m]} \quad \underbrace{A[m+1] \ldots A[n-1]}_{\substack{\text{search here if} \\ K > A[m]}}$$

We can use either recursive or non-recursive algorithm for it. The non-recursive algorithm is given below-

ALGORITHM BinarySearch(A[0..n-1], K)
// Implements non-recursive binary search
// Input: An array A[0..n-1] sorted in ascending
//   order & a search key K.
// Output: The position of array's element that
//   is equal to K, otherwise -1.

$l \leftarrow 0$

$r \leftarrow n-1$

while $l \leq r$ do
    $m \leftarrow \lfloor (l+r)/2 \rfloor$
    if $K = A[m]$
        return m
    else if $K < A[m]$
        $r \leftarrow m-1$
    else
        $l \leftarrow m+1$
return -1

Analysis:-

1. The parameter is input size n.
2. The basic operation is comparison of key with array elements.
3. As, the time complexity not only depends on 'n', but also on the possible position of key, we will go for all the efficiency classes.

Best Case:

Best possibility is the key is present exactly in the middle of the given array. For such a case, only one comparison is required.

So, $C_{best}(n) = 1 \leq n$

$\Rightarrow \quad C_{best}(n) \in \Omega(1).$

Worst case:

The worst case occurs if key is not found or it is found ~~at~~ in the last subarray. For both the cases, we have to search in all possible subarrays & each time the array size being reduced to the half of the previous size. Thus, if $C(n)$ is the total time required for search,

$$C_{worst}(n) = \begin{cases} C_w(\lfloor n/2 \rfloor) + 1 & , \quad n > 1 \\ 1 & , \quad n = 1 \end{cases}$$

Here, $C_w(\lfloor n/2 \rfloor)$ is the time required for searching either of the subarray and the term '1' is for comparing the key with middle element.

As, we will assume $n = 2^k$, $\lfloor n/2 \rfloor = n/2$.

So, $C_w(n) = C_w(n/2) + 1$

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

$= C_w(n/2^3) + 1 + 1 + 1$

$= \vdots$

$= C_w(n/2^k) + \underbrace{1 + 1 + 1 + \cdots + 1}_{(k \text{ times})}$

$= C_w(1) + k$

$= 1 + k$

$= 1 + \log_2 n$

$\therefore C_{worst}(n) \approx \log_2 n$, for large n.

$\therefore C_{worst}(n) \in \Theta(\log_2 n)$.

Average Case:

Let us discuss the number of comparisons required based on array size. As we will assume $n = 2^k$, we will consider only those situations where n is a power of 2. It is easily observed that -

If there are 2^0 items, 1 comparisons

- - - - - 2^1 - - - - , 2 - - - -

- - - - - 2^2 - - - - , 3 - - - -

- - - - - 2^3 - - - - , 4 - - - -

⋮

- - - - - 2^{c-1} - - - - , c - - - -

~~As, for large value of k, $2^k \approx 2^k - 1$, for the time-being, let us assume that $n = 2^k - 1$. (This is because, $\sum_{i=0}^{c-1} 2^i = 2^c - 1$).~~

Now, let us consider the average of all these possibilities.

$$C_{Avg}(n) = \frac{1}{n} \cdot \sum_{c=1}^{k} c \cdot 2^{c-1}$$

Multiplying 2 on both sides-

$$2 \cdot C_{Avg}(n) = \frac{1}{n} \cdot \sum_{c=1}^{k} c \cdot 2^{c}$$

Use the standard formula-

$$\sum_{i=1}^{n} i \cdot 2^{i} = (n-1)\, 2^{n+1} + 2$$

Now, $2 \cdot C_{Avg}(n) = \frac{1}{n}\left\{(k-1)\, 2^{k+1} + 2\right\}$

$$\Rightarrow 2 \cdot C_{Avg}(n) = \frac{2}{n}\left\{(k-1)\, 2^{k} + 1\right\}$$

$$\Rightarrow C_{Avg}(n) = \frac{1}{n}\left\{(k-1)(n\cancel{+1}) + 1\right\} \qquad \cancel{(\because 2^k = n+1)}$$

$$= (k-1)\cancel{\left(1+\frac{1}{n}\right)} + \frac{1}{n}$$

$$\approx k-1 \qquad \left(\because \text{As } n \to \infty,\ \frac{1}{n} \to 0\right)$$

$$\therefore C_{Avg}(n) = \log_2(n\cancel{+1}) - 1 \qquad \cancel{\left(\because 2^k = n+1 \Rightarrow k = \log(n+1)\right)}$$

$$\approx \log_2 n$$

$$\therefore C_{Avg}(n) \in \Theta(\log n)$$

Binary Tree Traversals & Properties :-

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees. viz. left subtree, T_L and right subtree, T_R. As the definition of binary tree itself divides it into two parts, many problems can be solved by applying divide-&-conquer strategy.

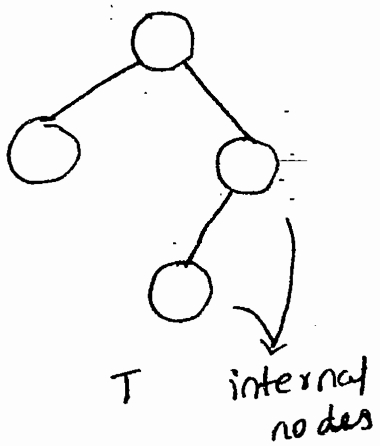
For illustration, let us consider a recursive algorithm for finding height of a binary tree. We know that height of a binary tree is the length of a longest path from the root to a leaf. So, this also can be given as the maximum of the heights of left subtree & right subtree plus 1.

```
ALGORITHM Height(T)
// Computes the height of binary tree recursively.
// Input : A binary tree T.
// Output : The height of T.
if T = Φ
    return -1
else
  return max { Height (T_L), Height (T_R)} + 1
```

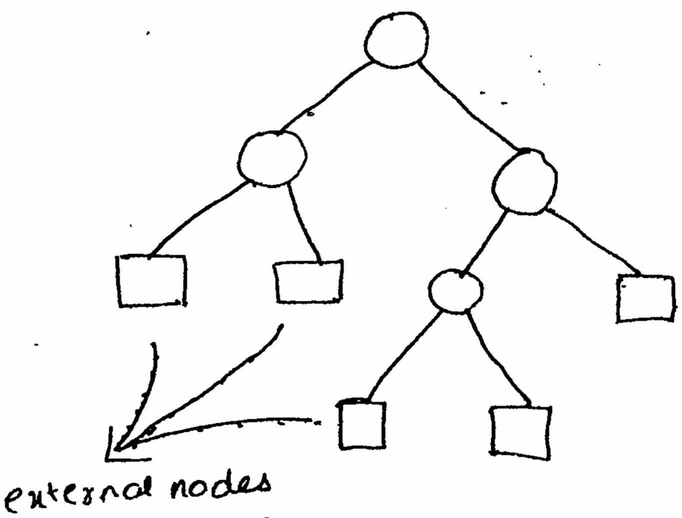
Let $n(T)$ be the number of nodes in T. Now, as every time we check for $T = \Phi$, and if this comparison fails, we add 1 to the maximum, the total number of comparisons is equal to the total number of additions, $A(n(T))$.

$$\therefore\ A(n(T)) = \begin{cases} A(n(T_L)) + A(n(T_R)) + 1, & n(T) > 0 \\ 0, & n(T) = 0 \end{cases}$$

But, here, we can observe that, the comparison is most frequently done operation but not the addition. Because, for empty tree, there will be comparison but not addition. So, for analysis purpose, we will draw the extension of tree by replacing empty subtrees by special nodes. For ex -



T



Extension of T

Now, it is easily observed that the algorithm makes one addition for internal node & one comparison for every internal & external node. And, a Btree with n nodes will be having $(n+1)$ external nodes.

ie $x = n+1$

$\therefore$ The number of comparisons -

$$C(n) = n + x$$
$$= n + n + 1$$
$$= 2n + 1.$$

And, number of additions is

$$A(n) = n.$$

NOTE: By using DAC, we can find the time complexities of inorder, preorder and post order tree traversals.

Multiplication of Large Integers :-

For many modern scientific applications, the manipulation of very large integers are required. Such integers can not be stored in normal computer's word. So, they require special ~~alg~~ methodologies for working on it.

Here, let us consider a multiplication of two n-digit numbers. In a normal way, the procedure requires n^2 multiplications. But, we can reduce the number of multiplications by slightly increasing the number of additions.

To illustrate, consider two 2-digit numbers 25 and 13.

Now,

$$25 = 2 \times 10^1 + 5 \times 10^0$$
$$13 = 1 \times 10^1 + 3 \times 10^0$$

Chetana Hegde
9441830189

Now,

$$25 \times 13 = (2 \times 10^1 + 5 \times 10^0)(1 \times 10^1 + 3 \times 10^0)$$
$$= (2 \times 1) 10^2 + [(2+5) \times (1+5) - (2 \times 1) - (5 \times 3)] 10^1 + (5 \times 3) 10^0$$
$$= 325$$

The actual methodology is as below –

If $a = a_1 a_0$ & $b = b_1 b_0$ are 2-digit integers, then, their product c is

$$c = a \times b$$

$$= c_2 . 10^2 + c_1 . 10^1 + c_0$$

where,

$$c_2 = a_1 * b_1$$

$$c_0 = a_0 * b_0$$

$$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$$

Thus, here ~~insetes~~ instead of 4 multiplications, we have only 3.

Now, consider two n-digit integers a & b, where n is even number. Divide the integers in the middle. Denote first half of 'a' as 'a_1' & second half as 'a_0'. Similarly for b also.

$$\therefore \quad a = a_1 10^{n/2} + a_0 \quad \&$$

$$b = b_1 10^{n/2} + b_0$$

Now, $c = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$$= c_2 10^n + c_1 10^{n/2} + c_0$$

where, $C_2 = a_1 * b_1$

$C_0 = a_0 * b_0$

$C_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$.

Now, we can apply same strategy for computing C_2, C_1 & C_0 recursively, if n is a power of 2.

As, multiplication of n-digit numbers takes three multiplication of $n/2$-digit numbers, we have,

$$M(n) = \begin{cases} 3 \cdot M(n/2), & n > 1 \\ 1, & n = 1. \end{cases}$$

$$\therefore M(n) = 3\, M(n/2)$$
$$= 3 \{ 3 \cdot M(n/2^2) \}$$
$$= 3^2\, M(n/2^2)$$
$$\vdots$$
$$= 3^k\, M(n/2^k)$$
$$= 3^k$$

$$\therefore M(n) = 3^{\log_2 n}$$
$$= n^{\log_2 3} \qquad (\because a^{\log_b c} = c^{\log_b a})$$
$$\approx n^{1.585}$$
$$< n^2$$

Thus, the number of multiplications reduced.

Strassen's Matrix Multiplication

Matrix multiplication is usually done by brute-force technique, which will take 8 multiplications and 4 additions for 2x2 matrices. An algorithm developed by V. Strassen for matrix multiplication will reduce the number of multiplications and so, reducing the execution time. The formula is as below -

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_3 + m_6 \end{bmatrix}$$

Here,
$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11})$$
$$m_2 = (a_{10} + a_{11}) * b_{00}$$
$$m_3 = a_{00} * (b_{01} - b_{11})$$
$$m_4 = a_{11} * (b_{10} - b_{00})$$
$$m_5 = (a_{00} + a_{01}) * b_{11}$$
$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01})$$
$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11})$$

Thus, Strassen's algorithm takes only 7 multiplications & 18 additions for 2x2 matrix.

Now, let us apply this for $n \times n$ matrix where n is a power 2. Let A & B be two such matrices. Then, their product C can be divided into four $n/2 \times n/2$ matrices.
i.e.

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} * \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Here $C_{00} = M_1 + M_4 - M_5 + M_7$ etc.
We can achieve this by recursively applying the Strassen's strategy on each of the four $n/2 \times n/2$ matrices.

Let $M(n)$ be the total multiplications required for two $n \times n$ matrices. We have—

$$M(n) = \begin{cases} 7 \cdot M(n/2), & n > 1 \\ 1, & n = 1 \end{cases}$$

$$\therefore M(n) = 7 \cdot M(n/2)$$
$$= 7\{7 \cdot M(n/2^2)\}$$
$$= 7^2 M(n/2^2)$$
$$\vdots$$
$$= 7^k \cdot M(1) = 7^k$$

i.e., $M(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx n^{2.807}$

$< n^3$, obviously.

Chetana Hegde
9448301894