Unit-4

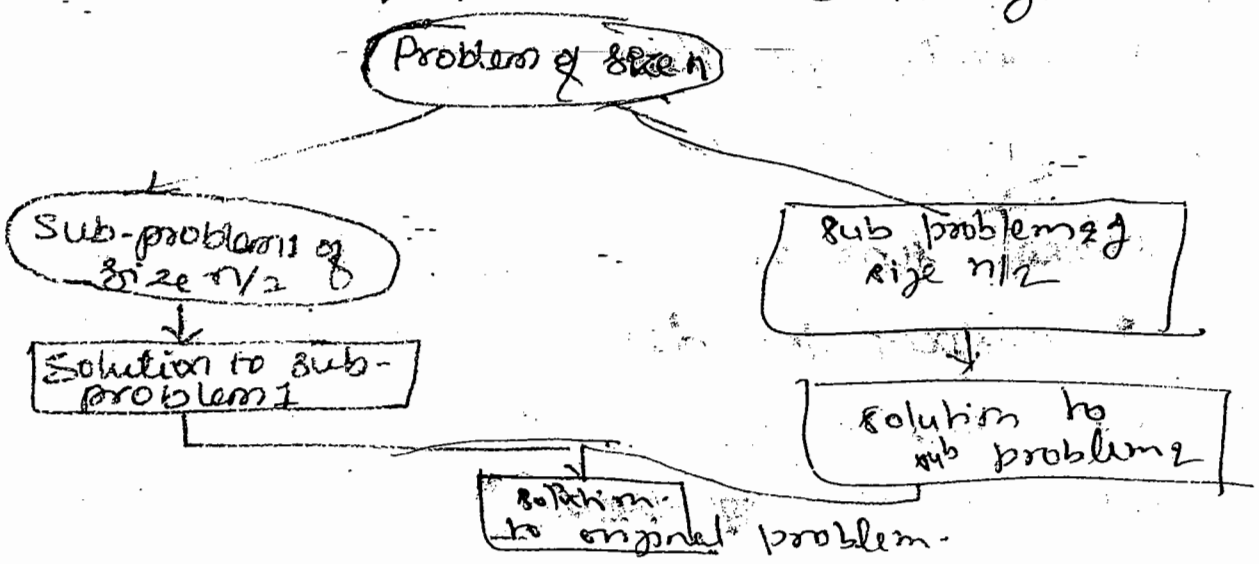
③ DIVIDE-AND-CONQUER

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 devide-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 computation problems, 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 each with $n/2$ (if n is even, otherwise sizes will be $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$) elements then 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 elements & second half elements. The term 1 is for adding two sums finally.

Now, if we use, substitution method-

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

$$= 2.\{2.T(n/4) + 1\} + 1$$

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

$$= 2^2.\{2.T(n/2^3) + 1\} + 2 + 1$$

$$= 2^3.T(n/2^3) + 2^2 + 2 + 1$$

$$= \cdots$$

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

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

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

$$= 2^k \cdot T(1) + 2^{k-1} + 2^{k-2} + \cdots + 2 + 1$$

$$= 2^k \cdot 0 + 2^{k-1} + \cdots + 2 + 1$$

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

$$= 2^k - 1$$

$$= n - 1$$

$$\simeq n, \text{ for large values of } n.$$

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

Chetana Khaire
9444830189

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

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

By applying $a \geq b^d$

i.e. $2 \geq 2^0$

we will get,

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

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 $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor .. n-1]$, 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. $B[i] \leq C[j]$. 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
9448183018@gmail.com

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 -

(a) 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.

(b) 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 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 procedures 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

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 ~~on~~ not only on n but also on the values 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 that will be the best situation.

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

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

(n → The number of comparisons made before partition.)

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

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

$$= 2^2 \cdot C(n/2^2) + n + n$$

$$= 2^2 \{2 \cdot C(n/2) + n/2\} + n + n$$

$$= 2^3 \cdot C(n/2^3) + n + n + n$$

$$= \vdots$$

$$= 2^k C(n/2^k) + n + \cdots + n \quad (k \text{ times})$$

$$= 2^k \cdot C(1) + nk$$

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

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

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 sub-array 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—

$$C_{worst}(n) = C(0) + C(n-1) + n, \quad n > 1$$

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

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

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

$$= \vdots$$

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

$$= 0 + 1 + 2 + \cdots + n$$

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

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

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

Average Case:

The 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 sub array. Now, if $C(n)$ is the time required for sorting entire array, $C(n-1)$ & $C(n-k)$ are the time required left & right sub arrays respectively.

Thus, the average total time is given by-

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

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-

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

Now, let us do the ~~forced~~ backward substitution for the equation-

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

$n \times$ Eq ① gives –

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

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

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

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

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

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 function $\sum_{k=2}^{n+1} \frac{1}{k}$ 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'}

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

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

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

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

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

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

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

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

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

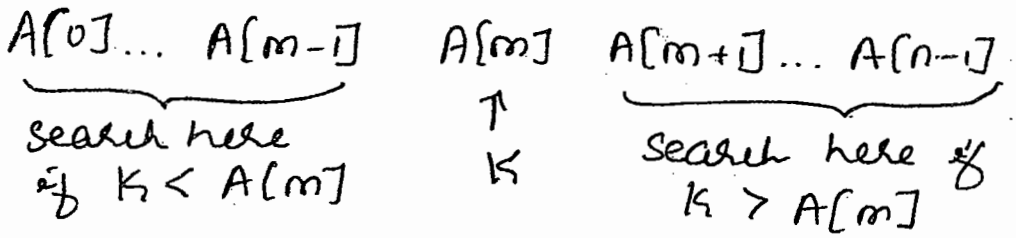
$$\approx (1.386)\, n \log_2 n \text{ , for large } n.$$

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

Chetana Hegde
9844830189

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. ie.



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 ← 0
r ← n-1
```

```
while l ≤ r do
    m ← ⌊(l+r)/2⌋
    if K = A[m]
        return m
    else if K < A[m]
        r ← m-1
    else
        l ← 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, $C_{best}(n) = 1 \leq n$

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

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,

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

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

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

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

$= \vdots$

$= C_w(n/2^k) + 1 + 1 + 1 + \cdots + 1$ (k 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 –

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

Use the standard formula –

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

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

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

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

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

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

$\therefore C_{Avg}(n) = \log_2(n+1) - 1$

$\approx \log_2 n$

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

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.

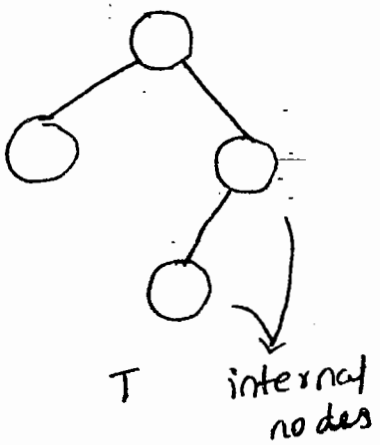
For illustration, let us consider a recursive algorithm for finding height of a binary tree. We know that height of a binary tree is the length of a longest path from the root to a leaf. So, this also can be given as the maximum of the heights of left subtree & right subtree plus 1.

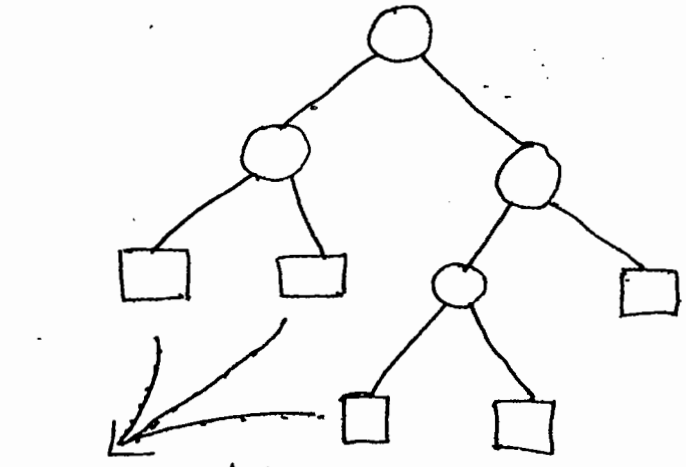
```
ALGORITHM Height(T)
// Computes the height of binary tree recursively.
// Input : A binary tree T.
// Output : The height of T.
if T = Φ
    return -1
else
    return max{Height(T_L), Height(T_R)} + 1
```

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. Forex.





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 B.tree with n nodes will be having (n+1) external nodes.

i.e. $x = n+1$

∴ 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 postorder 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 data 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, $25 = 2 \times 10^1 + 5 \times 10^0$

$13 = 1 \times 10^1 + 3 \times 10^0$

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

$$c = a \times b$$
$$= c_2 \cdot 10^2 + c_1 \cdot 10^1 + c_0$$

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 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.

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

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

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,

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

$\therefore M(n) = 3\, M(n/2)$

$= 3\{3 \cdot M(n/2^2)\}$

$= 3^2 M(n/2^2)$

$\vdots$

$= 3^k M(n/2^k)$

$= 3^k$

$\therefore M(n) = 3^{\log_2 n}$

$= n^{\log_2 3}$ $\quad (\because a^{\log_b c} = c^{\log_b a})$

$\simeq n^{1.585}$

$< n^2$

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 2×2 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 –

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

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

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 2×2 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. ie.

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

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—

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

$$\therefore M(n) = 7 \cdot M(n/2)$$

$$= 7\{7 \cdot M(n/2^2)\}$$

$$= 7^2 M(n/2^2)$$

$$\vdots$$

$$= 7^k \cdot M(1) = 7^k$$

$\therefore M(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx n^{2.807}$

$< n^3$, obviously.

Chelana Hegde
9th8301894