Divide and conquer

Divide & conquer is a very interesting algorithmic strategy. In this strategy the big problem is can be broken down into smaller sub problems and the solutions of these sub problems gives the final solution of the problem.

General Method : In divide & conquer method the given problem is

- Divided into smaller sub problems.
- These are solved independently.
- combining all the solutions of the sub problems into a solution of the whole.
- If the sub problems are large enough then divide and conquer is reapplied. Here recursive algorithms are used.

control Abstraction of Divide and conquer

general method of control abstraction is as shown below.

```
Algorithm Divide and conquer(P)
{ if 'P' is small then
    return solution of (P);
  else
  { Divide 'P' into smaller instances of
    P1, P2, .... Pn where n >= 1
    Apply Divide and conquer to the the
                                sub problems .
```

return combine (Divide & Conquer(P_1), Divide & Conquer(P_2), ... Divide & Conquer(P_n))

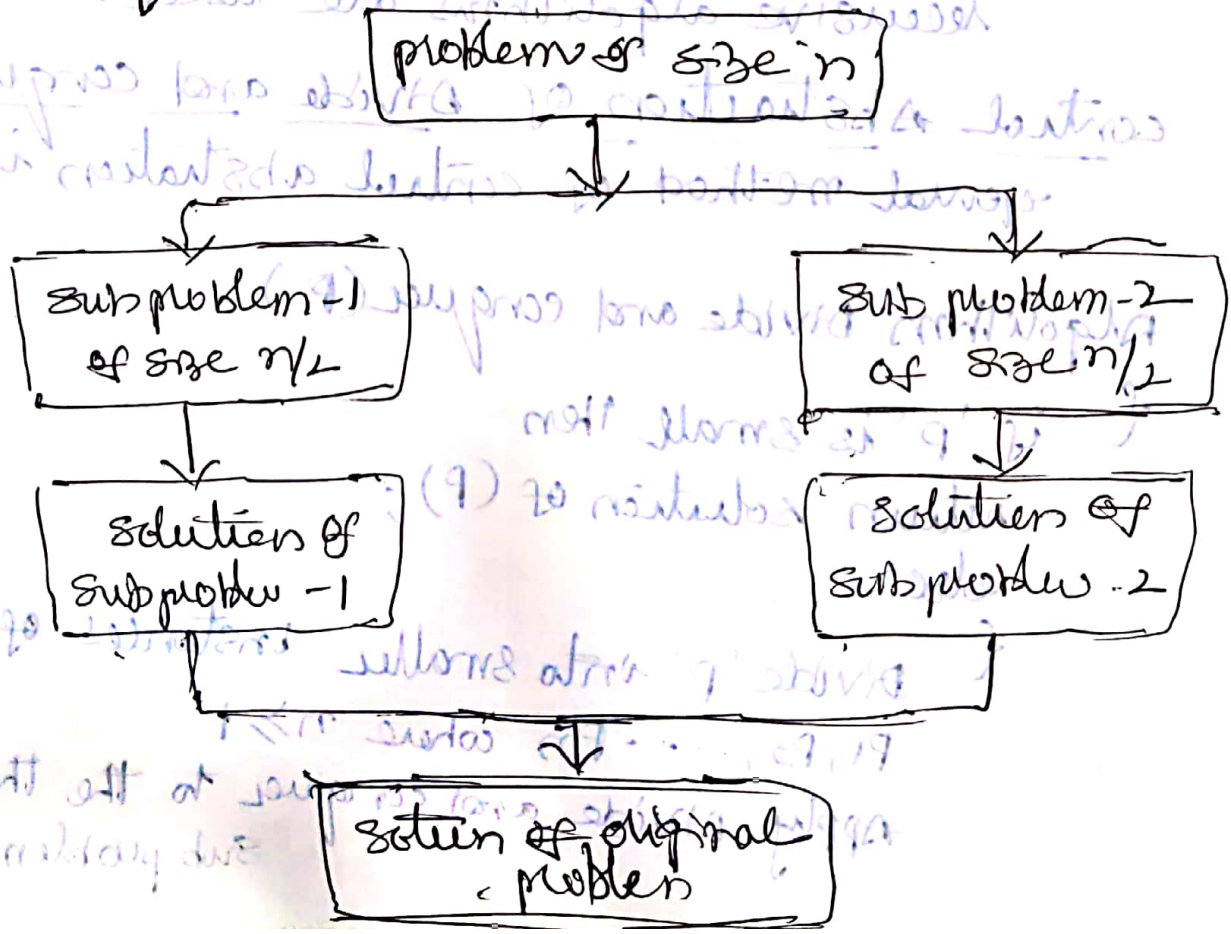
}

}

The computing time of Divide & Conquer is described by recurrence relation as

$$T(n) = \begin{cases} g(n) & \text{otherwise} \\ T(n_1) + T(n_2) + \cdots + T(n_r) + f(n) & \end{cases}$$

where $T(n)$ is the time for Divide & Conquer of size 'n', $g(n)$ is the computing time required to solve small inputs, $f(n)$ is the time required in Divide and Conquer problem 'p'.



Binary search method : Time complexity.

Worst case : In worst case time complexity, the algorithm compares all the elements for searching the desired element.

In this method, one comparison is made and based on the comparison array is divided each time, $n/2$ sublists. Hence worst case complexity

$$C_{worst}(n) = C_{worst}(n/2) + 1 \longrightarrow (1)$$

$C_{worst}(n/2)$ → time required to compare top & bottom sublist

$+1$ → one comparison is made with middle elem

Also

$$C_{worst}(1) = 1 \longrightarrow (2)$$

Assume $n = 2^k$ then eqn (1) becomes

$$C_{worst}(2^k) = C_{worst}(2^k/2) + 1$$

$$C_{worst}(2^k) = C_{worst}(2^{k-1}) + 1 \longrightarrow (3)$$

using backward substitution method we get

$$C_{worst}(2^{k-1}) = C_{worst}(2^{k-1-1}) + 1$$

$$C_{worst}(2^{k-1}) = C_{worst}(2^{k-2}) + 1 \longrightarrow (4)$$

substitute $C_{worst}(2^{k-1})$ in eqn (3) we have

$$C_{worst}(2^k) = [C_{worst}\,2^{k-2} + 1] + 1$$

$$C_{worst}(2^k) = [C_{worst}\,2^{k-2}] + 2$$

upto 'k' times we have

$$C_{worst}(2^k) = C_{worst}(2^{k-k}) + k$$

$$= C_{worst}(2^0) + k$$

$$C_{worst}(2^k) = C_{worst}(1) + k$$

From eq^n (2) we have

$C_{worst}(2^k) = 1 + k$

$\therefore C_{worst}(2^k) = 1 + \log_2 n$

$C_{worst}(n) = \log_2 n$ for $n > 1$

$\therefore$ The worst case time complexity of Binary search method is $\Theta(\log_2 n)$

Assume $n = 2^k$.

Taking log on both sides

$\log n = \log(2^k)$

$\log n = k \log 2$

$\therefore k = \log_2 n$

Average case :- For average case of Binary search consider the input values of 'n'.

consider the list as 11, 22, 33, 44, 55, 66, 77.

if n=1 then only one element '11' is there ie 1 → 11

$\therefore$ one comparison is required to search the key element.

if n=2 then two comparisons are made is to search '22'.

if n=4 then $m = (0+3)/2 = 1.5 = 1$

$\therefore A[1] = 22$ Here Key elent is '44'

Then $A[m] <$ Key ie $22 < 44$ then bottom of the list is considered. $\therefore m = (2+3)/2 = 5/2 = 2.5 = 2$

Again $A[m] <$ Key $\therefore A[m] = 33$

ie $33 < 44$ again consider bottom of the list

$\therefore m = (3+3)/2 = 6/2 = 3$

$\therefore A[m] = 44$

$\therefore A[m] =$ Key ie $44 = 44$

$\therefore$ The search is successful thus a total of '3' comparisons are made to search the Key '44'.

Quicksort method : Time complexity

Best case : For the best case time complexity is

$$C(n) = C(n/2) + C(n/2) + n \longrightarrow (1)$$

$C(n/2)$ → time req. to sort left sub array.
$C(n/2)$ → time req. to sort right sub array.
n → two req. for partitioning sub array.

and

$$C(1) = 0 \longrightarrow (2)$$

By using substitution method we get best case time complexity

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

Assume $n = 2^k$ then

$$C(2^k) = 2C(2^k/2) + 1 \cdot 2^k$$
$$C(2^k) = 2C(2^{k-1}) + 2^k \longrightarrow (3)$$

Using backward substitution method we get

$$C(2^{k-1}) = 2C(2^{k-1-1}) + 2^{k-1}$$
$$= 2C(2^{k-2}) + 2^{k-1}$$

substitute $C(2^{k-1})$ in eqn (3) we have.

$$C(2^k) = 2\left[2C(2^{k-2}) + 2^{k-1}\right] + 2^k$$
$$= 2^2 \cdot C[2^{k-2}] + 2^k + 2^k$$
$$= 2^2 \cdot C[2^{k-2}] + 2^k + 2^k$$
$$= 2^2 \cdot C[2^{k-2}] + 2 \cdot 2^k$$

upto 'k' times we can write

$$C(2^k) = 2^k \cdot C[2^{k-k}] + k \cdot 2^k$$
$$C(2^k) = 2^k \cdot C[2^0] + k \cdot 2^k$$
$$= 2^k \cdot C(1) + k \cdot 2^k$$

But $c(1)=0$ we have

$$c(2^k) = 2^k \cdot (0) + k \cdot 2^k$$

$$\therefore c(2^k) = k \cdot 2^k$$

Assume $2^k = n$ we get

$$c(n) = k \cdot n$$

$$= \log_2 n \cdot n$$

$$= n \cdot \log_2 n$$

$2^k = n$

Taking log on both sides

$\log(2^k) = \log(n)$

$k \log 2 = \log n$

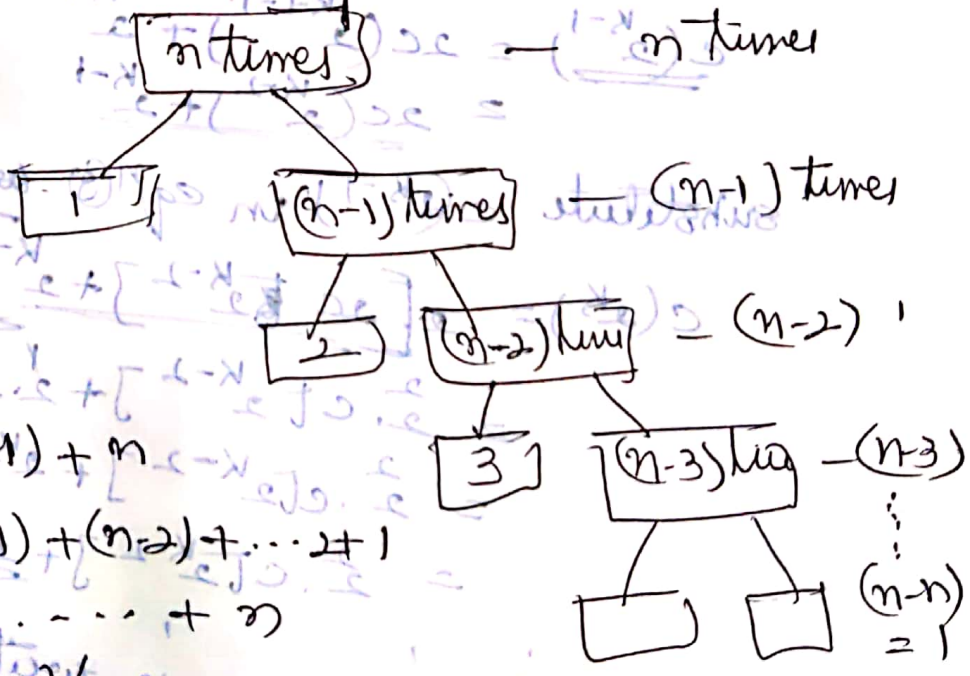
$k = \log_2 n$

∴ The best case time complexity of Quick sort is

$$\Theta = n \cdot \log_2 n.$$

⟹

Worst case time complexity: The worst case of quick sort occurs when pivot is min or max of all the elements. This is graphically represented as



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

$$\therefore c(n) = n + (n-1) + (n-2) + \cdots + 2 + 1$$

But $1 + 2 + 3 + \cdots + n$

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

$$= \frac{n^2+n}{2} = \frac{1}{2} \cdot n^2 + n/2$$

$$= \frac{1}{2} \cdot n^2$$

∴ The time complexity of worst case is $\Theta(n^2)$

⟹

Quick Sort Method :

This method partitioned the list into two sub lists

consider the list of following elements

44, 22, 77, 11, 66, 55, 33, 99, 88

pivot is the 1st element of the list ie

pivot = 44

i and j are the index variable of 2nd and last value of the list ie.

(44) (22), 77, 11, 66, 55, 33, 99, (88)

pivot A[i] A[j]

First compare pivot with A[i]

(i) if A[i] < pivot ⇒ i = i + 1

(ii) if A[j] > pivot ⇒ j = j - 1

At this point A[i] & A[j] - interchanged.

process is repeated with 'i' is fixed 'i' is incremented. when i > j Key is placed in final position. By interchanging pivot and A[j]

The encircled entries denote the keys being compared.

(44) (22) 77 11 66 55 33 99 (88)

pivot A[i] A[j]

(44) 22 (77) 11 66 55 33 99 88

~~(44) 22 77 (11) 66 55 33 99 88~~

(44) 22 77 11 66 55 33 99 (88)

(44) 22 77 11 66 55 (33) 99 88

(44) 22 77 11 66 55 33 99 88

Interchange 33 and 77

(44) 22 33 (11) 66 55 77 99 88

(44) 22 33 11 (66) 55 77 99 88

(44) 22 33 11 66 (55) 77 99 88

Interchange 11 and 44

11 22 33 44 66 55 77 99 88

list is partitioned into {11, 22, 33, 44} and {66, 55, 77, 99, 88}

same process applied to 2nd sublist

(66) (55) 77 99 (88)

pivot A[i] A[j]

Quick Sort Method :-

~~This method partitioned the list into two sub lists.~~

~~i and j – index variables with initial values 2nd and 4th respectively.~~

~~pivot is the 1st element of the list.~~

Example 2 :

Sort the below elements by using Quick Sort. Illustrate each step on the sorting.

33 44 55 22 11 55 66 24 10 6

The encircled entries denote the keys being compared.

(33) (44) 55 22 11 55 66 24 10 (6)
pivot [i] [j]

(33) (44) 55 21 11 55 66 24 10 6

(33) 44 55 21 11 55 66 24 10 (6)

Interchange 6 and 44

(33) 6 (55) 21 11 55 66 24 10 44

(33) 6 55 21 11 55 66 24 (10) 44

Interchange 10 and 55

(33) 6 10 (21) 11 55 66 24 55 44

(33) 6 10 21 (11) 55 66 24 55 44

(33) 6 10 21 11 (55) 66 24 55 44

(33) 6 10 21 11 55 66 (24) 55 44

Interchange 24 and 55

(33) 6 10 21 11 24 (66) 55 55 44

(33) 6 10 21 11 (24) 66 55 55 44

Interchange 24 and 33

(24) (6) 10 21 11 33 66 55 55 44

(24) 6 (10) 21 11 33 66 55 55 44

(24) 6 10 (21) 11 33 66 55 55 44

(24) 6 10 21 (11) 33 66 55 55 44

(24) 6 10 21 11 (33) 66 55 55 44

(24) 6 10 21 11 33 66 55 55 44

(24) 6 10 21 11 33 66 55 (55) 44

(24) 6 10 21 11 33 66 (55) 55 44

(24) 6 10 21 11 33 66 55 55 44

Algorithm :

This algorithm is performed using two functions Quick and partition.

(1). Algorithm Quick (A[0..n-1], low, high)

```
if (low < high) then
        // split the array into two sub arrays
    m ← partition ( A[low .. high] )
        // 'm' is the mid of the array.
    Quick (A[low .. (m-1)])
    Quick (A[(mid+1) .. high])
```

This algorithm call to partition algorithm as shown below :

The partition algorithm performs the arrangement of elements in ascending order.

(2). Algorithm partition (A[low .. high])

Here 'low' is the left most index of the array

'high' is the right most index of the array

```
pivot ← A[low]
    i ← low
    j ← high+1
    while (i <= j) do
    {   while (A[i] <= pivot) do
            i ← i+1 ;
        while (A[j] >= pivot) do
            j ← j-1 ;
        if (i <= j) then
            swap ( A[i], A[j]) .
    }
    swap (A[low], A[j])
        // when 'i' crosses 'j' swap
        A[low] and A[j]
    return j
```

Here arrange the elements that are less than pivot are at left side of pivot.

All the elements that are greater than pivot