Section 5

Divide and Conquer

DIVIDE & CONQUER

In this section we concentrate on Divide & conquer technique, which is one of the important technique to solve some problems very effectively. It is probably the best known general algorithm design technique.

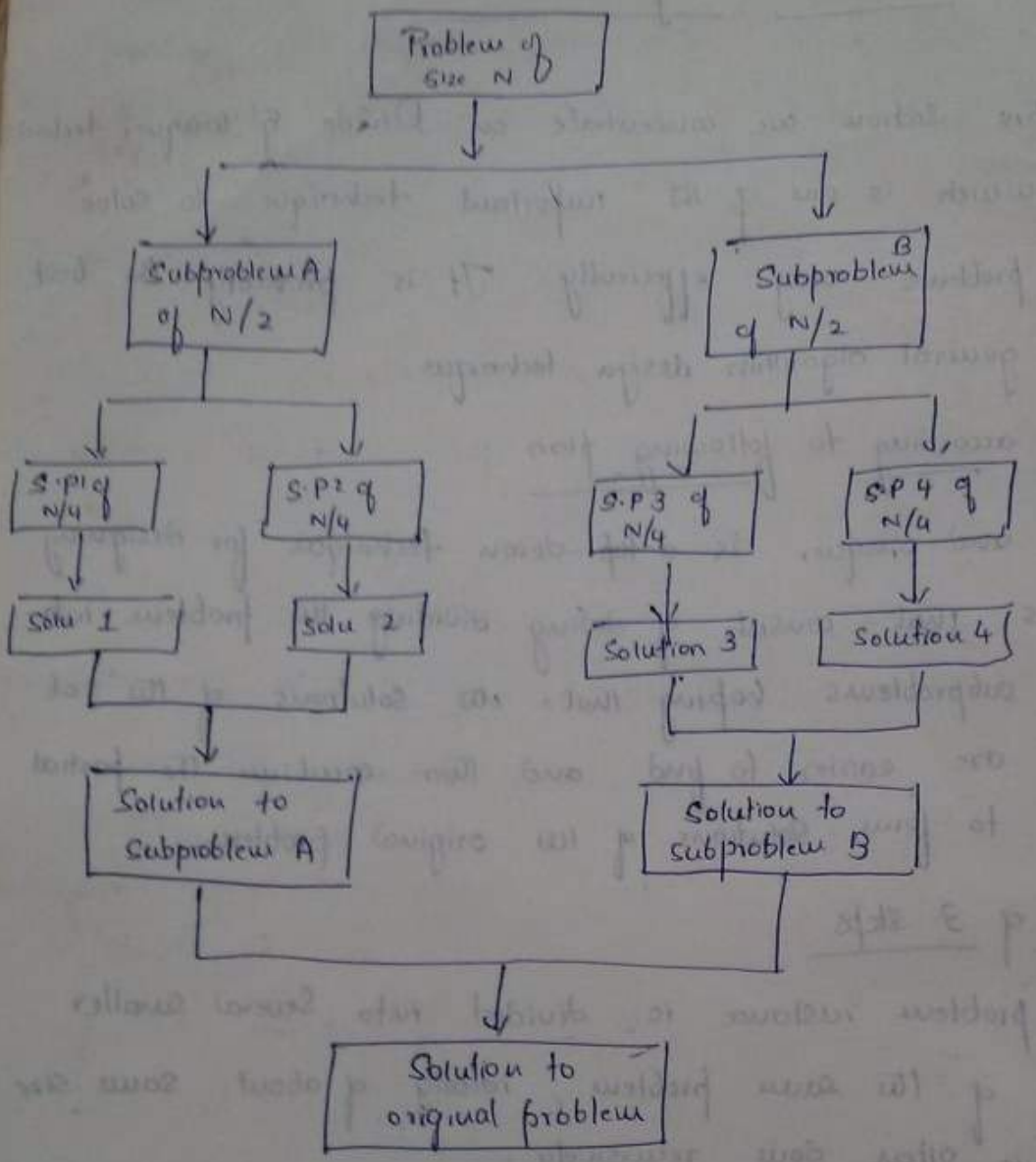
Works according to following plan :

Divide and conquer is a top-down technique for designing algorithms that consists of diving dividing the problem into smaller subproblems hoping that the solutions of the sub problems are easier to find and then combine the partial solutions to form solutions of the original problem.

Consists of 3 steps :

1. An problem instance is divided into several smaller instances of the same problem, ideally of about same size.
2. This is often done recursively.
2. The smaller instances are solved (recursively or sometimes a different algorithm is employed when instances become small enough)
3. The solutions obtained for smaller problem are combined to get a solution to the original instance.

Divide & conquer technique



Note: Every divide and conquer is not necessarily more efficient when compared to an algorithm designed using brute force. But remember that divide and conquer approach yields some of the most important & efficient algorithms in the field of computer science.

— This technique are ideally suited for parallel computations so that all the subproblems can be solved simultaneously and later all solutions can be merged into single solution.

Time comp

The most fa
instance of
n/b.
More general
instances o
solved [a
b, to siz
for running

$T(n)$

where $f(n)$
problem i

· Equation

The order
the values
growth of
by followi

Master th

in recurre

$T(n)$

Time complexity of Divide and conquer method

In most typical case of divide & conquer, a problems instance of size 'n' is divided into two instances of size n/b.

More generally, an instance of size 'n' can be divided into 'b' instances of size n/b, with a of them needing to be solved. [a ≥ 1 and b > 1] Assuming size 'n' is power of b, to simplify our analysis we get the following recurrence for running time T(n):-

$$T(n) = a\,T(n/b) + f(n), \quad \text{——— (1)}$$

where f(n) is a function that shows time spent on dividing problem into smaller ones and combining their solution.

Equation (1) is called general divide and conquer recurrence. The order of growth of its solution T(n) depends on the values of the constants a and b and the order of growth of the function f(n). The efficiency is simplified by following theorem.

Master theorem: If $f(n) \in \Theta(n^d)$ with $d \geq 0$ in recurrence equation (1) then,

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

For example, the recurrence equation for the number of additions $A(n)$ made by divide & conquer sum-computation algorithm on the inputs of size $n = 2^k$ is

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

$2A/2$

No of addition required in left and right array

1 addition to add last two elements

Thus for this example, comparing with general divide & conquer recurrence relation we get,

$$T(n) = aT(n/b) + f(n)$$

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

$\Rightarrow$ $a = 2$, $b = 2$ and $f(n) = 1$

but we know $f(n) \in \Theta(n^d)$

$$f(n) = n^d$$

$$= n^0 = 1$$

$\therefore d = 0$

Now $2 > 2^0$ $\quad [\text{i.e. } a > b^d]$

$2 > 1$

$$\therefore A(n) \in \Theta\left(n^{\log_b a}\right) = \Theta\left(n^{\log_2 2}\right) = \Theta(n)$$

Merge sort

It is a

divide and

It divides

ie A

- Sorts
- Merges

Algorithm

// Sorts

// Input

// output

if n >

copy

Copy

Mergeso

Mergeso

Merge

algorith

// Me

// Inpu

// Sor

Merge Sort:

It is a perfect example of a successful application of the divide and conquer technique.

- It divides given array $A[0 \dots n-1]$ into 2 halves ie $A[0 \dots \lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor \dots n-1]$.
- Sorts each of them recursively
- Merges the two sorted arrays into a single sorted one.

Algorithm: MergeSort $(A[0 \dots n-1])$

// Sorts array $A[0 \dots n-1]$ by recursive mergesort

// Input: An array $A[0 \dots n-1]$ of orderable elements

// output: Array $A[0 \dots n-1]$ sorted in nondecreasing order.

if $n > 1$

copy $A[0 \dots \lfloor n/2 \rfloor - 1]$ to $B[0 \dots \lfloor n/2 \rfloor - 1]$

Copy $A[\lfloor n/2 \rfloor \dots n-1]$ to $C[0 \dots \lceil n/2 \rceil - 1]$

Mergesort $(B[0 \dots \lfloor n/2 \rfloor - 1])$

Mergesort $(C[0 \dots \lceil n/2 \rceil - 1])$

Merge (B, C, A)

algorithm Merge $(B[0 \dots p-1], C[0 \dots q-1], A[0 \dots p+q-1])$

// Merges two sorted arrays into one

// Input: Arrays $B[0 \dots p-1]$ & $C[0 \dots q-1]$ both sorted

// Sorted array $A[0 \dots p+q-1]$ of elements B & C

$i \leftarrow 0$, $j \leftarrow 0$; $k \leftarrow 0$

while $i < p$ and $j < q$ do

if $B[i] \leq C[j]$

$A[k] \leftarrow B[i]$; $i \leftarrow i+1$

$A[k] \leftarrow C[j]$; $j \leftarrow j+1$

$k \rightarrow k+1$;

if $i = p$

copy $C[j \cdots q-1]$ to $A[k \cdots p+q-1]$

else copy $B[i \cdots p-1]$ to $A[k \cdots p+q-1]$

Effeciency

Number of comparisions is

$$C(n) = 2C(n/2) + C_{merge}(n) \quad \text{for } n > 1 \qquad C(1) = 0$$

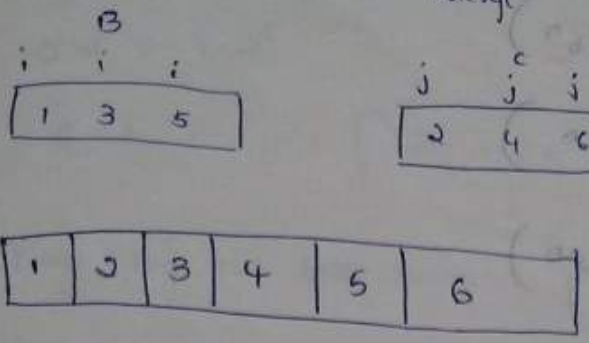
$2C(n/2)$ ↓ : No of comparisions required to sort left and right parts of array

$C_{merge}(n)$ ↘ : No of key comparisions performed during the merging stage

At each step one comparision is made, after every step the total number of elements will be reduced by one.

In Worst case, neither of two arrays become empty before the other one [i.e smaller elements come from alternate arrays] $\therefore C_{merge}(n) = n-1$ in worst case.

ex



No of comparisions

1 & 2
3 & 2
3 and 4
5 and 4
5 and 6

$\therefore$ Total number of comparisions are $5 \cdot [6-1]$

$$C_{worst}(n) = 2C_{worst}(n/2) + n-1 \quad \text{for } n>1$$

$$C_{worst}(1) = 0$$

Time complexity using masters theorem:

The general recurrence relation for divide & conquer is

$$T(n) = aT(n/b) + f(n)$$

where as recurrence relation for merge sort is

$$C(n) = 2C(n/2) + C_{merge}(n)$$

$a = 2, \; b = 2, \quad f(n) = n = n^1$

$\therefore d = 1$

$a = 2 \quad b^d = 2^1 = 2$

$\therefore a = b^d$

$\Rightarrow C(n) = \Theta(n^d \log_b n)$

$= \Theta(n^1 \log_2 n)$

$= \Theta(n \log_2 n)$

Find same using substitution method.

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

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

put $n = 2^k$,

$C(2^k) = 2C(2^k/2) + c2^k$

$= 2C(2^{k-1}) + c2^k$

$= 2\left[2C(2^{k-2}) + c2^{k-1}\right] + c2^k$

$= 2^2 C(2^{k-2}) + 2 \cdot c2^{k-1} + c2^k$

$= 2^2 C(2^{k-2}) + 2c2^k$

$\vdots$

$2^3 C(2^{k-3}) + 3c2^k$

$2^k T(2^{k-k}) + k \cdot c \cdot 2^k$

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

$= 2^k c(1) + k \cdot c \cdot 2^k$

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

$= c \cdot k \cdot n \quad [\because n = 2^k]$

Taking log on both sides for $n = 2^k$

$k \log_2 2 = \log_2 n$

$k = \log_2 n$

$\Rightarrow c \cdot n \cdot \log_2 n$

$\Rightarrow \in \Theta(n \log_2 n)$

Merge sort:

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

Let $n = 2^k$

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

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

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

$= 2^2 \cdot [2 \cdot T(2^{k-3}) + 2^{k-2} - 1] + 2^k - 2 + 2^k - 1$

$= 2^3 \cdot T(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1$

$= 2^3 T(2^{k-3}) + 3 \cdot 2^k - 2^2 - 2 - 1$

$= 2^3 \cdot T(2^{k-3}) + 3 \cdot 2^k - (2^3 - 1)$

$= 2^k \cdot T(2^{k-k}) + 3 \cdot k \cdot 2^k - (2^k - 1)$

$= 0 + k \cdot 2^k - 2^k + 1$

$= 2^k (k-1) + 1$

$n = 2^k$

$= \log_2^n \cdot n - n + 1$