Divide and Conquer

This technique divides a given problem into smaller instances of the same problem. Solves the smaller problems and combine solutions to solve the given problem.

For a given large data set.

* DIVIDE: Partition the data set into smaller sets
* Solve the problems for the smaller sets
* CONQUER: Combine the results of the smaller sets.

Algorithm Divide_and_Conquer (x).

If x is small

return solve (x)

else

Decompose x into smaller sets $x_0, x_1, \ldots x_n$.

for i ← 0 to n-1 in step of 1.

y_i = Divide_and Conquer (x_i)

Combine y_i's (We have solved y for x)

return y.

Three conditions that make D & C worthwhile

1. It must be possible to decompose an instance into sub instances.
2. It must be efficient to recombine the results.
3. Sub instances should be about the same size.

Some examples of Divide and Conquer :-

① Binary search :-

Array, Size, Element to be searched

Algorithm BinSearch (a, n, x)

// Array is in nondecreasing order.

```
{
    low ← 1 ;  high ← n ;
    while ( low ≤ high ) do
    {
        mid ← ⌊(low + high)/2⌋ ;
        if (x < a[mid])
            then  high ← mid - 1 ;
        else if  (x > a[mid])
            then  low ← mid + 1;
        else
            return mid;         ( ~~successfull~~ search successful)
    }
    return 0 ;   ( Unsuccessful search)
}
```

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

for successful search -

Best case : $\Theta(1)$

Average case : $\Theta(\log_2 n)$

Worst case : $\Theta(\log_2 n)$

for unsuccessful search -

~~$\Theta(\log_2 n)$~~ $\Theta(\log_2 n)$

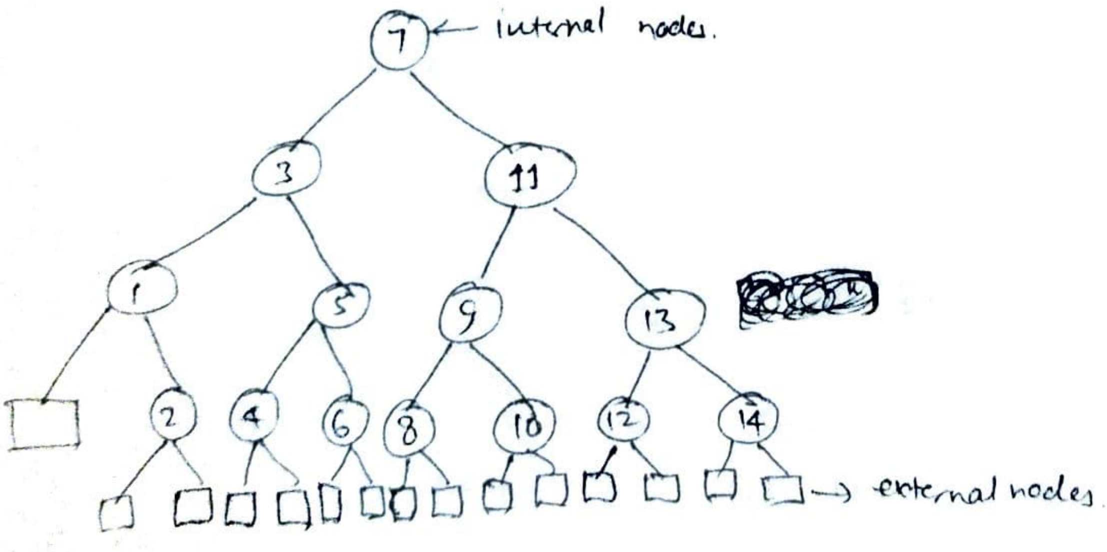
Theorem :- If n is in the range $[2^{k-1}, 2^k)$ then binary search makes atmost k comparisons for a successful search and either k+1 or k comparisons for an unsuccessful search.) ~~(i.e. time for a successful search is $O(\log_2$ and for unsuccessful search is $\Theta(\log_2 n))$~~

Sol:- Consider a binary decision tree for n element all successful search ends at the circular node while all unsuccessful search ends at a square node.

If $2^{k-1} \leq n < 2^k$, then all circular nodes are at level 1, 2, ... k.

While all squares nodes are at levels k and k+1.

So for the successful search k comparisons required and for unsuccessful search k or k+1 comparisons are required.



(2) Merge sort:-

~~The merge sort~~

- The procedure merge sort sorts the elements in the sub array A [low, high].
- If low ≥ high the sub array has at most one element and is therefore already sorted.
- Otherwise we divide it into A [low, mid] and A [mid+1, high]

Algorithm MergeSort (low, high)
// a[low : high] is a global array to be sorted.

{

if (low < high) then

// Divide it into subproblems

mid ← ⌊(low + high)/2⌋;

~~MergeSort~~
// Solve the subproblems.

MergeSort (A, low, mid);

MergeSort (A, mid+1, high);

// Combine the solutions.

Merge (low, mid, high).

}end if.

}end algorithm

Algorithm Merge (low, mid, high). Page 147 (HS)

// a[low, high] is a global array containing two sorted subsets
// in a[low, mid] and in a[mid+1, high]. The goal is to
// merge these two sets into a single set residing in
// a [low, high. b[ ] is an auxiliary global array.

{

h ← low; i ← low ; j ← mid+1;

while ((h ≤ mid) and (j ≤ high)) do

{

if (a[h] ≤ a[j]) then

{

b[i] ← a[h];

h ← h+1;

}

else

{

b[i] ← a[j];

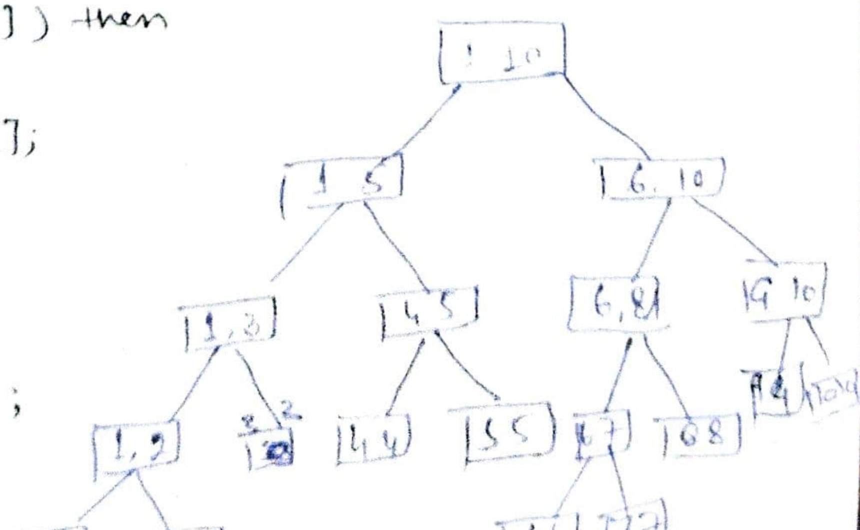
j ← j+1;

}

i ← i+1;

}

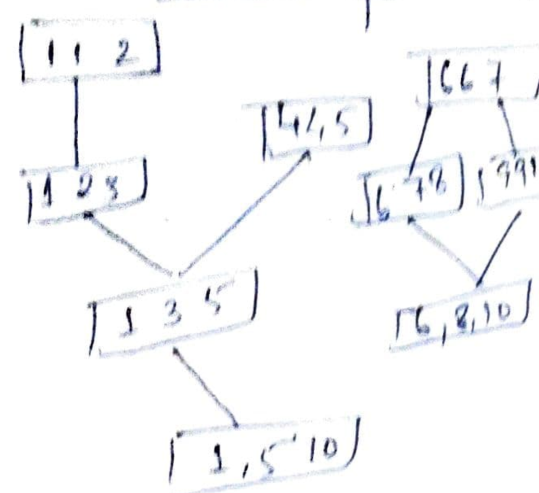
Ref : Tree of call of mergesort



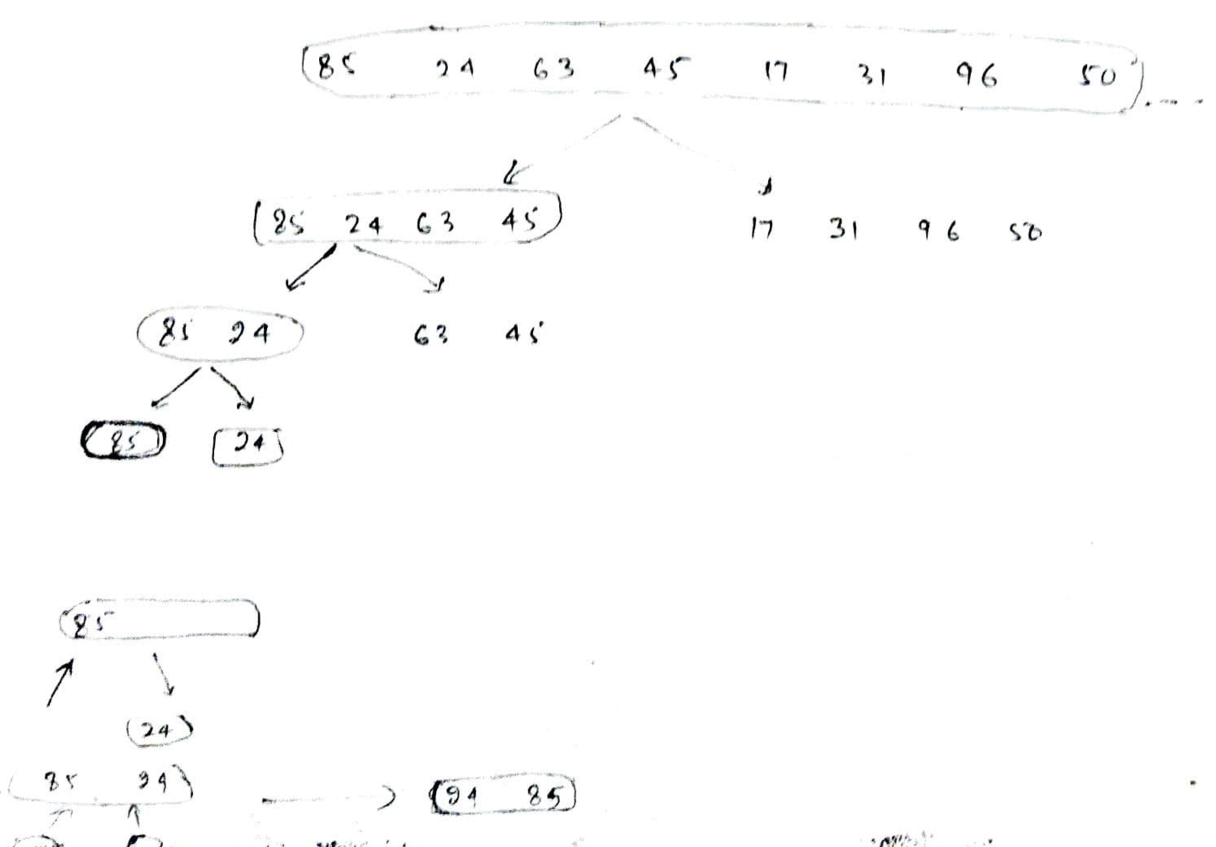
```
if (h > mid) then
    for k ← j to high do
    {
        b[i] ← a[k];
        i ← i+1;
    }
else
    for k ← h to mid do
    {
        b[i] ← a[k];
        i ← i+1;
    }

    for k ← low to high do
        a[k] ← b[k];
}
```

Tree of call of Merge



Example:-



~~Q) the time for the merging~~

The computing time for merge sort is described by the recurrence relation :-

$$T(n) = \begin{cases} a & n = 1, \quad a \text{ is a constant} \\ 2T(n/2) + cn, & n > 1, \quad c \text{ is a constant} \end{cases}$$

where n is a power of $2, n = 2^k$

<u>Solution</u> :-

$$T(n) = 2T(n/2) + cn$$

$$= 2(2T(n/4) + cn/2) + cn$$

$2^2\left[2T\left(\frac{n}{2^3}\right) + c\cdot\frac{n}{2}\right] + 2cn =$ $$2^2 T(n/2^2) + 2.cn.$$ $2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{cn}{2}\right] + 2c$

$$= 2^3 T(n/2^3) + 3cn.$$ $= 2^3 T\left(\frac{n}{2^3}\right) + c$

$$\vdots$$

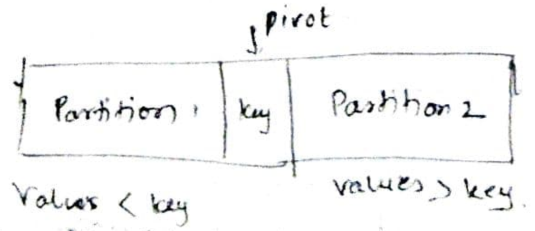
$$= 2^k T(1) + kcn$$

$$= an + cn\log_2 n$$

Therefore complexity $T(n) = O(n \log_2 n)$

③ <u>Quick sort</u> :-

Similar to Merge Sort but by choosing a pivot point, with lesser numbers on left and greater on right. ~~It partitions the array of numbers to be sorted, but does not need to keep a temporary array~~. It is faster than merge sort because it does not go through the merge phases.



Algorithm QuickSort (a, p, r)   In book Algorithm QuickSort (p, q)

```
if p < r then        // More than one element.
    q ← Partition (a, p, r)
    QuickSort (a, p, q-1);
    Quicksort (a, q+1, r);
end if.
end algorithm.
```

```
Algorithm Partition (a, p, r)
    x ← A[p]
    i ← p-1.
    j ← r+1.
    while TRUE
        repeat j ← j-1.
        until A[j] ≤ x.
        repeat i ← i+1.
        until A[i] ≥ x
        if i<j
            then exchange A[i] ↔ A[j]
            else return j.
```

```
Algorithm Interchange (a, i, j)
// Exchange a[i] with a[j]
    P ← a[i] ;
    a[i] ← a[j];
    a[j] ← P ;
end algorithm.
```

// Rearranging the elements of an Array is referred to as partitioning.

```
Algorithm quicksort(a, m, p)
{ if (m<p).
    { q ← partition(a, m, p+1);
      quicksort(a, m, q-1)   no of elements +1
      quicksort(a, q+1, p)
    }
}
```

(a, m, p) Page 156 (H.S.).

r = p+1

J = P+1

```
Algorithm Partition(a, m, p)
{
    v = a[m]; i = m; j = p;
    repeat
    {
        repeat
            i = i+1;
        until (a[i] ≥ v);
        repeat
            j = j-1
        until (a[j] ≤ v);
        if (i<j) then interchange(a, i, j)
    } until (i ≥ j)
    a[m] = a[j];
    a[j] = v;
    return j;
}
```

```
Algorithm Interchange(a, i, j)
{ p = a[i]
  a[i] = a[j]
  a[j] = p;
}
```

Algorithm' Partition the Array.

Analysis of Quick sort :-

- Worst case : If unbalanced partitioning.

  One region with one element and the other with n-1 elements. If this happen in every step i.e. when array is already sorted.
  then complexity = $O(n^2)$

- Best case : if balanced partitioning :-

  Two regions, each with $n/2$ elements.

  then computing time for quick sort is :-

$$T(n) = \begin{cases} a & n = 1. \\ 2T(n/2) + cn, & n > 1. \end{cases}$$

~~same as~~ This recurence relation have;

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

- Averag case closer to the best case than the worst case.

Example

key	low								high
45	26	77	14	68	61	97	39	99	90
		low					high		
45	26	77	14	68	61	97	39	99	90
			low			high			
45	26	39	14	68	61	97	77	99	90
			high	low					
45	26	39	14	68	61	97	77	99	90
14	26	39	45	68	61	97	77	99	90

$\underbrace{14 \quad 26 \quad 39}_{\text{value} < 45}$ 45 $\underbrace{68 \quad 61 \quad 97 \quad 77 \quad 99 \quad 90}_{\text{value} > 45}$

Now apply quick sort procedure on each of these subarrays until the entire array is sorted.

Strassen's matrix multiplication.

Let A and B be two $n \times n$ matrices. The product $C = A.B$ is also $n \times n$ and can be obtained as:-

$$C(i,j) = \sum_{1 \le k \le n} A(i,k)\, B(k,j)$$

for all i and j between 1 and n

```
for(i=0; i<n; i++)
for(j=0; j<n; j++)
{ c[i][j]=0;
for(k=0; k<n; k++)
c[i][j]=c[i][j]+
a[i][k]*
b[k][j]
```

To compute $C(i,j)$ we need n multiplications

As C has n^2 elements we can say matrix multiplication algorithm is $O(n^3)$.

n^3

~~We will take square~~ ~~n is power of 2 and square~~

~~matrix~~ The divide and conquer strategy suggests another way to compute the product of two $n \times n$ matrices. We assume that n is a power of 2.

~~The divide and conquer strategy suggests another way to compute the product of two n×n matrices.~~

Imagine that ~~each~~ A and B are each partitioned into four square submatrices, each submatrix having dimensions $n/2 * n/2$. Then product is:-

In case n is not a power of 2, then enough rows & columns of zeros are added to both A and B so that resulting dimensions are of power of 2

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$

$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$

$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$

$$C_{22} = A_{21} B_{12} + A_{22} B_{22}$$

```
Algo MM(A, B, n)
{ if (n≤2)
  { c = Applying the formula
  }
  else
  { mid = n/2
    MM(A11, B11, n/2) + MM(A12, B21, n/2)
    MM(A11, B12, n/2) + MM(A12, B22, n/2)
    matrix add
  }
}
```

8 multiplications + 4 addition of $n/2 * n/2$ matrices.

Since two $n/2 * n/2$ matrices can be added in time cn^2 for some constant c. Hence overall computing time $T(n)$ of the resulting divide & conquer algorithm is given by the recurrence.

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2. \end{cases}$$

$\cancel{T(\dots)}$

$T(n) = O(n^3)$

where b and c are constants.

Although complexity is same but number of computations is reduced. but no improvement over conventional method

Volker Strassen's method:-

Since matrix multiplications $[O(n^3)]$ are more expensive than matrix additions $[O(n^2)]$. In this method there are 7 multiplications and 18 additions or subtractions.

First compute seven $n/2 * n/2$ matrices P, Q, R, S, T, U, V and then C_{ij} by these matrices:

$$P = (A_{11} + A_{22}).(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22}) B_{11}$$

$$R = A_{11} (B_{12} - B_{22})$$

$$S = A_{22} (B_{21} - B_{11})$$

$$T = (A_{11} + A_{12}) B_{22}.$$

$$U = (A_{21} - A_{11}) (B_{11} + B_{12})$$

$$V = (A_{12} - A_{22}) (B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

The resulting recurrence relation for T(n) is

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

where a and b are constants.

Working with this formula, we get

$$T(n) = an^2\left[1 + 7/4 + (7/4)^2 + \cdots + (7/4)^{k-1}\right] + 7^k T(1)$$

$$\le cn^2\,(7/4)^{\log_2 n} + 7^{\log_2 n} \qquad c \text{ a constant}$$

$$= cn^{\log_2 4 + \log_2 7 - \log_2 4} + n^{\log_2 7}$$

$$= O(n^{\log_2 7})$$

$$\simeq O(n^{2.81}).$$

$$\le cn^2\, n^{\log_2(7/4)} + 7^{\log_2 n}$$

$$\le cn^{\log_2 4 + \log_2 7 - \log_2 4} + 7^{\log_2 n}$$

$$\le cn^{\log_2 7} + n^{\log_2 7}$$

$$\simeq O(n^{\log_2 7})$$

$$\simeq O(n^{2.81}).$$

$$T(n) = 7T\left(\frac{n}{2}\right) + an^2$$

$$= 7\left(7T\left(\frac{n}{4}\right) + a\left(\frac{n}{2}\right)^2\right) + an^2$$

$$= 7^2 . T\left(\frac{n}{4}\right) + \frac{7}{4}an^2 + an^2$$

$$= 7^3 T\left(\frac{n}{8}\right) + 7^2 a\left(\frac{n}{4}\right)^2 + \frac{7}{4}an^2 + an^2$$

$$= 7^3 T\left(\frac{n}{8}\right) + \left(\frac{7}{4}\right)^2 an^2 + \frac{7}{4}an^2 + an^2$$

$$= 7^k T\left(\frac{n}{2^k}\right) + \left(\frac{7}{4}\right)^{k-1} an^2 + \cdots + \left(\frac{7}{4}\right)^2 an^2 + \frac{7}{4}an^2 + an^2$$

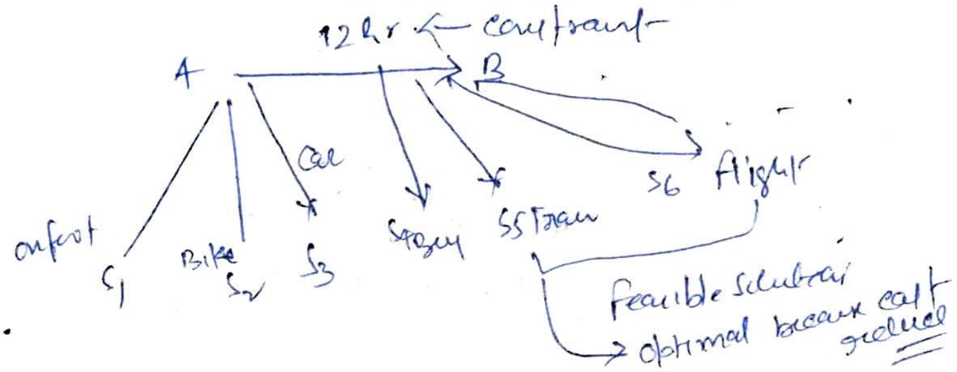
$$= 7^{\log_2 n} T(1) + an^2\left[1 + \left(\frac{7}{4}\right) + \left(\frac{7}{4}\right)^2 + \cdots + \left(\frac{7}{4}\right)^{k-1}\right]$$

$$\leq cn^2\left(\frac{7}{4}\right)^{\log_2 n} + 7^{\log_2 n}$$

$\frac{ar^n}{r-1}$

our own method for solving a Problem

greedy Approach Solving
↳ optimization Problem
↳ Problem which requires either minimum result or maximum result.



Feasible → satisfying given constraint

A solution which is feasible and giving minimum cost (best result) then this solution is known as optimal solution

↳ there is only one ~~maximum~~ optimal solution not more than one.

↳ Greedy
Dynamic programming
Branch & Bound method
} Approaches are different

Size

Algo greedy (a, n)

$n = 5$

$a = $ | a_1 | a_2 | a_3 | a_4 | a_5 |

```
{  for i = 1 to n do
     x = select(a);
     if feasible(x) then
       solution = solution + x;
```