UNIT - 2

DIVIDE & CONQUER

General method :

In divide & conquer method, a given problem is divided into smaller sub-problems. These sub problems are solved independently. Combining the solutions of all the sub problems into a solution of whole

* If the sub problems are large enough then divide & conquer is reapply.
* The generated sub problems are usually of some type has the original problem.
* A control abstraction were divide & conquer is given below :

using control abstraction a flow of control of a procedure is given :

Algorithm :

```
Algorithm dac(P)
{
  if small[P] then
  return s[P]
  else
  { divide P into smaller instances P1, P2, ---, Pk
    k≥1;
    Apply dac to each of these sub problems;
    return combine [dac(P1), dac(P2) ---, dac(P(k))];
  }
}
```

* The computing time of the above procedure is given by the recurrance relation.

$$T(n) = \begin{cases} g(n) & \text{if } n \text{ is small} \\ T(n_1) + T(n_2) + \cdots T(n_k) + f(n) & n \text{ is large} \end{cases}$$

* where T(n) is the time for divide & conquer of size n. g(n) is the computing time requiring small inputs. The

f(n) is the time requiring in dividing the problems & combining the solutions of sub problems.

* The applications of this method are :

1) Binary search
2) Merge sort
3) Quick sort
4) Strassens Matrix multiplication.

Binary search :

1) In the Binary search method the list of elements in the ascending orders are taken as the input. Here a key element is given to search in the list.
2) The list is divided into two parts initially by finding the mid position and the mid = $\frac{start + end}{2}$
3) All the elements in the left sub list are less than the mid value. All the elements in the right sub list are greater than the mid value
4) The key element is now compared with the mid value. then one of the three conditions may be true.

* if key is less than a[mid] then left sub list is taken and again the same process is repeated for searching the element in the left sub list. it means that again in the left sub list the mid position will be found and the key element is compared with the mid value and again three cases will occur. the same process is repeated till the element is found. in any case

* if key is greater than a[mid] then right sub list is taken and again same process is repeated for searching the in the right sub list. it means that again right sub list mid position will be found and the key

T(n-k)

element is composed with the mid element and again three cases will occur the same process is repeated in any case till the element is found.

* if key is equal to a[mid] then return the mid position

Ex: Given list is a[ ] =

-15 -6 0 7 9 23 54 82 101 112 125 131 142 151

find the keys : key1 = 151, key2 = -14, and key3 = 9.

Ans: key 1 = 151

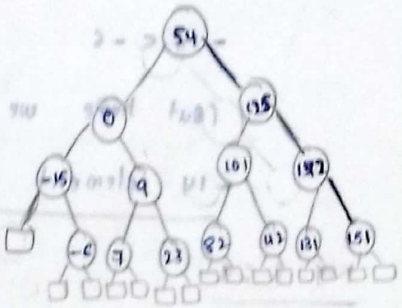
$$mid = \frac{s+e}{2} = \frac{0+13}{2} = \frac{13}{2} = 6$$

i) if (key == 151 == a[mid]

151 = 54

151 > 54 so RSL

$$mid = \frac{7+13}{2} = 10$$



ii) 151 == a[mid] (a[10])

151 > 125 so RSL

$$mid = \frac{10+13}{2} = 12$$

iii) 151 == a[12]

151 > 142 so RSL

$$mid = \frac{13+13}{2} = \frac{26}{2}$$

iv) 151 == a[13]

151 = 151 (element found)

return index ⟹ return 13

key 2 = -14 :

$$mid = \frac{s+e}{2} = \frac{0+13}{2} = 6.$$

i) if -14 == a[mid]

-14 == a[6]

-14 < 54 so LSL

$mid = \frac{0+5}{2} = 2.$

ii) $-14 = = a[2]$

$-14 < 0$ so LSL

$mid = \frac{0+1}{2} = 0$

iii) $-14 = = a[0]$

$-14 > -15$ so RSL.

$mid = \frac{1+1}{2} = 1.$

iv) $-14 = = a[1]$

$-14 < -6$

(But here we already checked -15)

-14 element not found

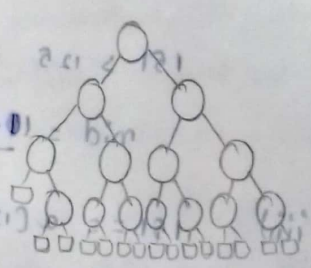
Key 3 : 9

$mid = \frac{s+e}{2} = \frac{0+13}{2} = 6$

i) $9 = = a[mid]$

$9 = = a[6]$

$9 < 54$ so move to LSL



$mid = \frac{0+5}{2} = 2.$

ii) $9 = = a[mid]$

$9 = = a[2]$

$9 > 0$ so move to RSL

$mid = \frac{3+4}{2} = 3.$

iii) $9 = = a[mid]$

$9 = = a[3]$

$9 > 7$ so move to RSL

$mid = \frac{4+4}{2} = 4.$

iv) $q == a[4]$

$q = q$ (element found)

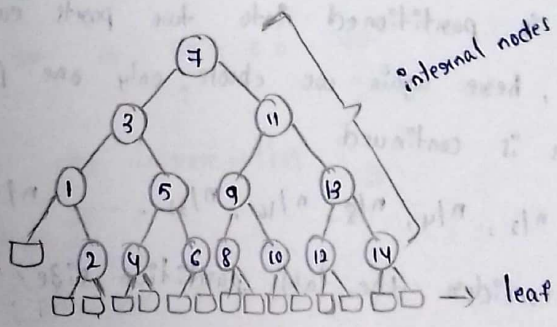
return element index.

∴ return 4.

Algorithm for Binary search:

```
Algorithm  Binary search (a, n, x)
{
    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 if ( x == a[mid] )
            return index (mid) ;
        else
            print (" element not found" );
    }
}
```

Binary Decision tree:



* A Binary decision tree traces the way in which these values are produced in binary search.
* The first comparision is x with a[7]

if x is less than a [7] the next comparision is a[3] similary if x is greater than a[7] then the next com-

-parison is with a[1] each path through the tree represents a sequence of comparison in the binary search method.

* If 'x' is present then the algorithm will end at one of the circular (internal) nodes. If 'x' isn't present then the algorithm will terminate at one of the square nodes.

* circular nodes are called internal nodes and square nodes are referred to as leaf nodes.

Analysis of Binary search:

The analysis of Binary search depends on the position of the element.

Best case: the best case occurs when the position of the search element is present in the middle of all the elements. It takes only one unit of time. So the time taken to search for an element in the best case is "O(1)"

worst case: It occurs when the position of the search element is starting (or) ending of all the positions of elements. In order to find out the search element we have to partition all the elements into two parts the size of the each ~~path~~ part is $n/2$. we only choose one part based on the search element. Again the selected path is partitioned into two parts each having the size $n/4$, here again we choose only one part like this the process is continued

$$n, n/2, n/4, n/8, n/16, n/32 \dots n/2^k$$

Let us consider the last partition size is 1.

i.e., $\frac{n}{2^k} = 1$ applying log on both sides

$$k = \log_2 n$$

So the time taken to search the element in the worst case $O(\log n)$

Average case:

The position of search element may be at the middle (or) at the starting (or) at the ending the average time taken to search a key element is overally $O(\log n)$

* The time taken for unsuccessful search is $O(\log n)$

Quick sort:

In this (~~merge~~) method the array of elements are taken as the input. Here one element is choosen as the pivot, the pivot can be the first element (or) the last element (or) the middle element (or) any random element in the list.

* Usually the first element in the list taken as the pivot. The correct position of the pivot in the list is determined by comparing the pivot element with remaining elements in the list.
* The position of the pivot element is find in such a way that all the elements left of the pivot element are less than the pivot and all the elements which are right of the pivot element are greater than the pivot element
* Once the pivot is placed in its correct position, then the list is divided into two parts (or) sublists. Again in each sublist the above procedure is repeated.

Procedure to find the pivot in the list:

1) Lets take an array of n elements a[1] to a[n]. The first element in the array a[1] is taken as pivot, let us say i, j indicates starting and ending positions of the array
   i.e., i=1 and j=n
2) Increment i ~~until~~ every time if a[i] ≤ pivot
   once a[i] > pivot stop incrementing i
3) Decrement j ~~until~~ every time if a[j] > pivot
   once a[j] ≤ pivot stop decrementing j
4) check the positions of i and j if (i<j) then swap a[i] and a[j] and then repeat the above steps from (2)
5) If i and j are crossed i.e., if (i≥j) then swap pivot and a[j], once the pivot is swapped then the

list is divided into two parts.

6) Repeat the same process from step 1 to step 5 for each and every sub list.

Algorithm quicksort (a, lb, ub)

{

if (lb < ub)

{

loc = partition (a, lb, ub);

quicksort (a, lb, loc-1);

quicksort (a, loc+1, ub);

}

}

Algorithm partition (a[], lb, ub)

{

p = a[lb];

start = lb, end = ub;

while (start < end)

{ while (a[start] ≤ p && start < end)

{ start = start+1;

}

while (a[end] > p)

{ end = end-1;

}

if (start < end)

{ swap (a[start], a[end]);

}

}

a[lb] = a[end];

a[end] = pivot;

return end;

}

lb ub

4

Quick sort :

30	20	10	50	60	40
i (pivot) (start)					j (end)

consider pivot = 30

* ~~while~~ ~~until~~ a[i] ≤ pivot do i++
* while ~~until~~ a[j] > pivot do ~~i++~~ j--

i < j swap a[i] & a[j]

i ≥ j swap a[j] & pivot

30	20	10	50	60	40
i,p					j

a[i] = p ⇒ i++

~~30~~	20	10	50	60	40
p	i				j

a[i] < p ⇒ i++

30	20	10	50	60	40
p		i			j

a[i] < p ⇒ i++

30	20	10	50	60	40
p			i		j

a[i] > p stop incrementing i

a[j] > p ⇒ j--

30	20	10	50	60	40
p			i	j	

a[j] > p ⇒ j--

30	20	10	50	60	40
p			i,j		

a[j] > p ⇒ j--

30	20	10	50	60	40
p		j	i		

a[j] < p so stop decrementing j

as we stopped incrementing i and decrementing j

Comparing i and j i = 4, j = 3.

(i > j) so swap a[j] & pivot.

then 10 20 30 50 40 60

30 is placed in its position then it divides total list into two parts.

* consider Left sub list.

10 20
i,p j

a[i] = pivot ⇒ i++

10 20
p i,j

a[i] > pivot so stop incrementing i

a[j] > pivot ⇒ j--

10 20
p,j i

a[j] = pivot so stop decrementing j

Comparing i and j i = 2, j = 1

(i > j) so swap a[j] & pivot means here a[j] and pivot are equal then there is no swapping.

10 20

* consider right sub list

50 60 40
p,i j

a[i] = pivot ⇒ i++

50 60 40
p i j

a[i] > pivot so stop incrementing i

* [a[i] > pivot ~~sign~~

50 60 ~~40~~
p ~~i,j~~ i] x

* [a[i]<p so stop decrementing j

comparing i and j i=2 and j,

a[j] < pivot so stop decrementing j

50	60	40
p	i	j

compare i and j i=5 and j=6

(i<6) then swap a[i] and a[j]

50	40	60
p	i	j

a[i] < pivot ⇒ i++

50	40	60
p	i,j	

a[i] > pivot ⇒ stop incrementing i

a[j] > pivot ⇒ j--

50	40	60
p	j	i

a[j] < pivot stop decrementing j

compare i and j i=6 and j=5

(i>j) so swap a[j] & pivot.

40 [50] 60

50 is placed in its position and there is only one element in left and right sub lists no need to repeat the process.

∴ The total list is

10 20 30 40 50 60

* If the list is divided into two parts and there is only 1 element in the left and right sub lists no need of repeating process.

* An element p (pivot element) only divides list into two sub lists swapping a[i] & a[j] don't divide the list into two parts.

Ex2: 65 70 75 80 85 60 55 50 45

sort the elements using quick sort

Ans:

65 70 75 80 85 60 55 50 45
i, p j

a[i] ≤ pivot ⇒ i++

65 70 75 80 85 60 55 50 45
p i j

a[i] > pivot ⇒ stop incrementing i

a[j] < pivot ⇒ stop decrementing j

compare i and j i=1, j=8

(i<j) swap a[i] & a[j]

65 45 75 80 85 60 55 50 70
p i j

a[i] < pivot ⇒ i++

65 45 75 80 85 60 55 50 70
p i j

a[i] > pivot ⇒ stop incrementing i

a[j] > pivot ⇒ j--

65 45 75 80 85 60 55 50 70
p i j

a[j] < pivot ⇒ stop decrementing j

(i<j) swap a[i] & a[j]

65 45 50 80 85 60 55 75 70
p i j

a[i] < pivot ⇒ i++

65 45 50 80 85 60 55 75 70
p i j

a[i] > pivot ⇒ stop incrementing i

a[j] > pivot ⇒ j--

65 45 50 80 85 60 55 75 70
p i j

a[j] < pivot ⇒ stop decrementing j

(i<j) swap a[i] & a[j]

65	45	50	55	85	60	80	75	70
p			i			j		

a[i] < pivot ⇒ i++

65	45	50	55	85	60	80	75	70
p				i		j		

a[i] > pivot ⇒ stop incrementing i

a[j] > pivot ⇒ j--

65	45	50	55	85	60	80	75	70
p				i	j			

a[j] < pivot ⇒ stop decrementing j

(i<j) swap a[i] & a[j]

65	45	50	55	60	85	80	75	70
p				i	j			

a[i] < pivot ⇒ i++

65	45	50	55	60	85	80	75	70
p					i,j			

a[i] > pivot ⇒ stop incrementing i

a[j] > pivot ⇒ j--

65	45	50	55	60	85	80	75	70
p				j	i			

a[j] < pivot ⇒ stop decrementing j

Compare i, and j i=5, j=4

(i>j) swap a[j] & pivot

60	45	50	55	65	85	80	75	70

↓

pivot element is placed in it's position and list is divided into left and right sub lists and again repeat the process.

consider left sub list

```
60   45   50   55
p,i            j
```

a[i] = pivot ⇒ i++

```
60   45   50   55
p    i         j
```

a[i] < pivot ⇒ i++

```
60   45   50   55
p         i    j
```

a[i] < pivot ⇒ i++

```
60   45   50   55
p              i,j
```

a[i] < pivot ⇒ (but i reaches end of the left sub list, so stop incrementing i)

a[j] < pivot ⇒ stop decrementing j

(i=j) swap a[j] & pivot

```
55   45   50   [60]
```

(pivot element is placed so repeat the process)

there is no right sub list apply the process for left sub list

```
55   45   50
p,i       j
```

a[i] = p ⇒ increment i

```
55   45   50
p    i    j
```

a[i] < pivot ⇒ i++

```
55   45   50
p         i,j
```

i reached end of the list stop incrementing i

a[j] < pivot ⇒ stop decrementing j

compare i and j (i=j) swap a[j] & pivot

50 45 [55]

pivot element is placed correct the process

left list : 50 45
p i j

a[i] = pivot => i++

50 45
p i j

a[i] < pivot (i reached end so stop incrementing i)

a[j] < pivot => stop decrementing j

(i=j) swap a[j] & pivot

45 [50]

pivot element is placed

the left sub list has only one element no need to re-peat the process.

left sub list : 45 50 55 60

Now consider the right sub list of first pivot element

85 80 75 70
p i j

a[i] = pivot => i++

85 80 75 70
p i j

a[i] < pivot => i++

85 80 75 70
p i j

a[i] < pivot => i++

85 80 75 70
p i,j

i reached end stop incrementing i

a[j] < pivot => stop decrementing j

compare i and j (i=j) swap a[j] & pivot

70 80 75 [85]

pivot placed consider left sub list

70 80 75
p i j

a[i] = pivot => i++

70 80 75
p i j

a[i] > pivot => stop incrementing i

a[j] > pivot => j--

70 80 75
p i,j

a[j] > pivot => j--

70 80 75
p,j i

a[j] = pivot => stop decrementing j

(i > j) swap a[j] & pivot.

70 80 75

pivot placed consider right sub list.

80 75
p,i j

a[i] = pivot -> i++

80 75
p i,j

a[i] < pivot (but i reached end stop incrementing i)

a[j] < pivot (stop decrementing j)

(i = j) swap a[j] & pivot.

75 80

Right sub list is : 70 75 80 85

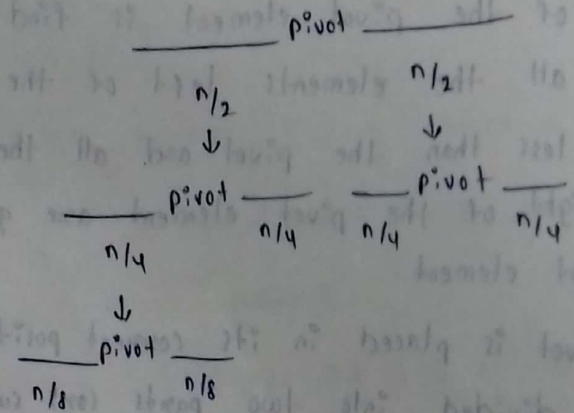
sorted list is : 45 50 55 60 65 70 75 80 85

Analysis of quick sort:

The running time of the quick sort depends on whether the partition is balanced (or) unbalanced.

* A very good partition splits an array into 2 parts.

i.e., If there are 'n' elements in the array

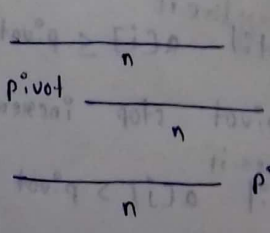


Like wise in each step the whole number of elements to be sorted are 'n' which will take the time $\log_2 n$

The best case time complexity is $O(n \log_2 n)$

* A bad partition splits an array into very different sized arrays.

i.e., If there are 'n' elements in an array



It means in each step there are almost 'n' elements which takes time of n elements sorting.

The worst case time complexity $O(n^2)$

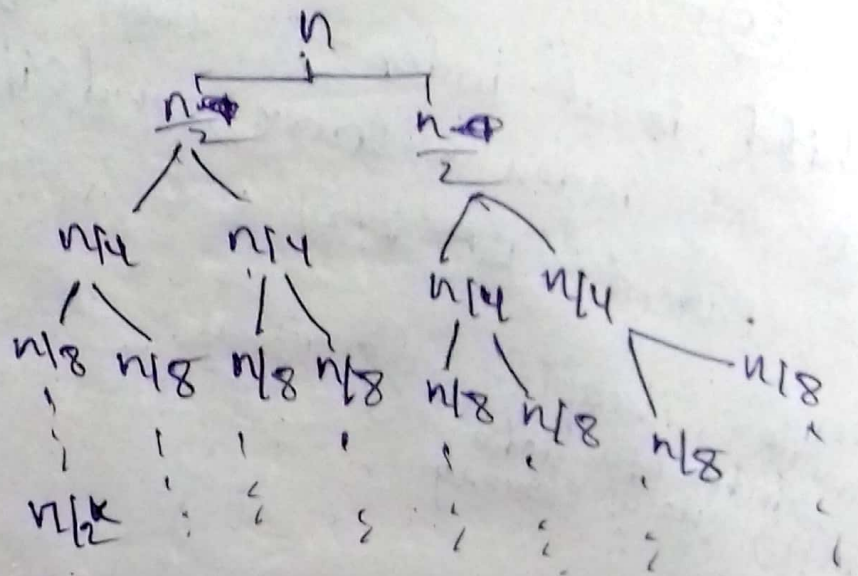
* An average partition splits in both ways in different levels.

∴ The average time complexity is $O(n \log_2 n)$

to the height of the binary tree).

Quick Sort: Analysis:

Best case: Best case occurs if the list is always divided into two equal parts. say n elements are there in the list.



the partition gets stopped whenever the list contains only one element i.e $n/2^k = 1$

$n = 2^k$ apply log

$\log n = k \log 2$

$k = \frac{\log n}{\log 2} = \log_2 n$

(or)

```
quicksort()
{ if(start<end)
 {loc=partition(a,lb,ub); - n
  quicksort(a,lb,loc-1); → T(n/2)
  quicksort(a,loc+1,ub); → T(n/2)
 }
}
```

$T(1) = C_1$

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

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

$= 8T(n/8) + 3Cn$

$= 2^k T(n/2^k) + kCn$

$= 2^{\log_2 n} \cdot T(1) + Cn \log_2 n$

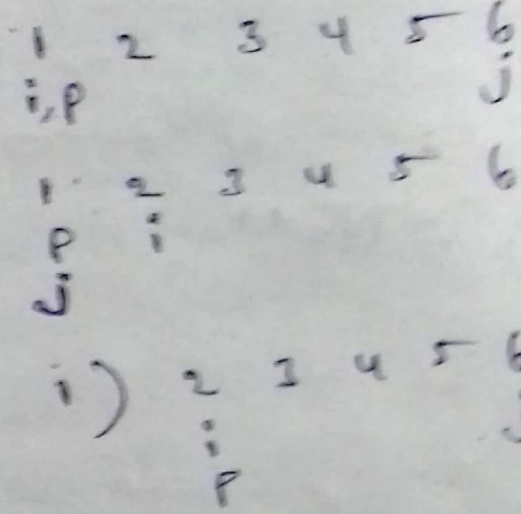
$= nC_1 + Cn \log_2 n = O(n \log_2 n)$

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

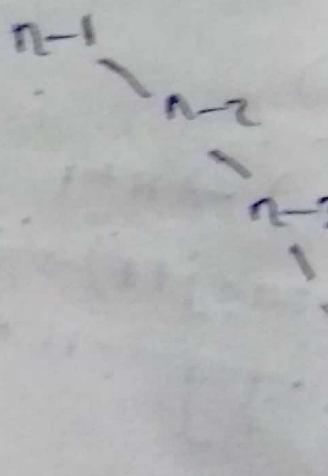
$k = \log_2 n$

$O(n \log_2 n)$

worst case :- worst case occurs when the taken i/p is in ascending or descending order



Each time the list is divided into two parts such that one part contains only one element and the another part contains n-1 elements.



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

→ To convert the worst case to best case, select middle element as pivot.

Merge sort:

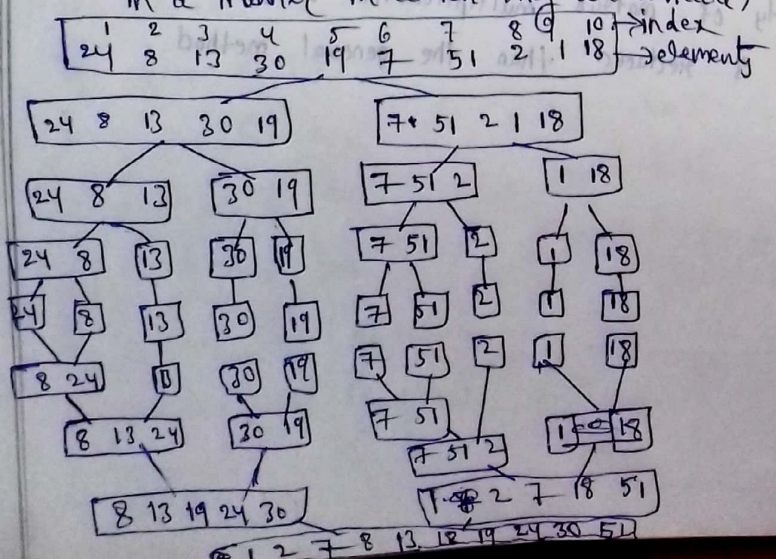
In this method list of elements are given, we have to arrange the elements in the order, it follows the divide & conquer technique.

* Here the list is divided into two equal sub lists by finding the mid position.

$$\text{where } mid = \frac{low + high}{2}$$

* The elements in the left sub list are start to mid, mid+1 to end elements are taken in the right sub list.
* Consider the left sub list repeat the same process as above, that is divide left sub lists into two equal sub sub lists such that start, mid are the elements of the list and mid+1, end are the elements in the left right sub list. For each and every sublists the above process is repeated until the sub sub list contain single element.
* The same procedure is repeated for right sub list also until it is divided into a sub sub list contain a single element.
* Consider the two adjacent singleton ~~sub~~ sub lists, merge them by arranging the elements in order.
* ~~Merge~~ the Repeat the process until the entire list is merged into a single list. (The sublists are merged in a manner in which they are divided).

Ex:



Algorithm for merge sort:

```
Algorithm Merge sort (low, high)
{
  if (low < high) then
  {
     mid := (low + high) / 2 ;
     Merge sort (low, mid) ;
     Merge sort (mid+1, high);
     Merge (low, mid, high);
  }
}
Algorithm Merge (low, mid, high)
{
  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 ;
  }
  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] ; }
```

To copy the leftover elements in left sublist to b array

To copy the leftover elements in right sublist to b array

Ex Tracing:

24 8 13 30 19 7 54 2 1 18 sort the elements using merge sort.

Ans: Given

24	8	13	30	19	7	51	2	1	18
1	2	3	4	5	6	7	8	9	10

Mergesort (1, 10)

if (1<10) true

{

mid = $\frac{1+10}{2}$ = 5

Mergesort (1,5); —— I

Mergesort (6, 10); —— II

Merge (1, 5, 10); —— III

}

I: if (1<5) true

mid = $\frac{1+5}{2}$ = 3.

Mergesort (1,3); —— IA

Mergesort (4,5); —— IB

Merge (1, 3, 5); —— IC

IA: if (1<3) true

mid = $\frac{1+3}{2}$ = 2

Mergesort (1,2); —— IAa

Mergesort (3,3); —— IAb

Merge (1, 2, 3); —— IAc

IAa: if (1<2) true

mid = $\frac{1+2}{2}$ = 1

Mergesort (1,1); —— IAa'

Mergesort (2,2); —— IAa''

Merge (1, 1, 2); —— IAa'''

IAa': if (1<1) false

IAa'': if (2<2) false

JAa:

Merge (1, 1, 2)

{

h = 1 ; i = 1 ; j = 2 ;

while ((1 ≤ 1) and (2 ≤ 2)) true

{

if (24 ≤ 8) false

else

b[1] = 8 ; j = 3 ;

i = 2 ;

while ((1 ≤ 1) and (3 ≤ 2)) false

}

if (h > mid) ⇒ (1 > 1) false

else

for k = h to mid

⇒ for k = 1 to 1

b[2] = 24 ; i = 3 ;

for k = low to high ⇒ for k = 1 to 2

a[1] = 8

a[2] = 24

JAb: Merge sort (3, 3)

if (3 < 3) false

JAc: Merge (1, 2, 3)

{

h = 1 ; i = 1 ; j = 3 ;

while ((h ≤ mid) and (j ≤ high))

while ((1 ≤ 2) and (3 ≤ 3)) true

{ ~~if (a[h] ≤ a[mid]) ⇒ if (8 ≤ 24) true~~

if (a[h] ≤ a[j]) ⇒ if (8 ≤ 13) true

b[1] = 8

h = 2 ; i = 2.

while ((2 ≤ 2) and (3 ≤ 3) true

if (a[h] ≤ a[j]) ⇒ if (24 ≤ 13) false

else

b[i] = a[j] → b[2] = a[3]

b[2] = 13 j = 4

i = 3;

} while ((i ≤ 2) and (4 ≤ 3)) false

}

if (h > mid) → (2 > 2) false

else

for k = h to mid

k = 2 to 2

(b[i]) b[3] = a[2] (a[k])

b[3] = 24;

for k = low to high ⇒ for k = 1 to 3

a[1] = 8

a[2] = 13

a[3] = 24

IB: Mergesort (4, 5)

if (4<5) true

mid = $\frac{4+5}{2}$ = 4

Mergesort (4, 4) — IB_a

Mergesort (5, 5) — IB_b

Merge (4, 4, 5) — IB_c

IBa: Mergesort (4, 4)

if (4<4) false

IBb: Mergesort (5, 5)

if (5<5) false

IBc: Merge (4, 4, 5)

h = 4; i = 4; j = 5;

while ((h ≤ mid) and (j ≤ high))

while ((4≤4) and (5≤5)) true.

if (a[h] ≤ a[j]) ⇒ if (a[4] ≤ a[5])

if (30 ≤ 19) false

else

{

b[4] = a[5] ;

b[4] = 19 ;

j = 6 ;

}

i = 5 ;

while ((4≤4) and (6≤5)) false

}

if (4 > 4) false

else

{ for k = h to mid (for k = 4 to 4)

b[5] = a[4] ;

b[5] = 30 ;

i = 6 ;

}

for k = low to high ⇒ for k = 4 to 5

a[4] = 19 ;

a[5] = 30 ;

IC: Merge (1, 3, 5)

{

h = 1 ; i = 1 ; j = 4

while ((h ≤ mid) && (j ≤ high))

{

while ((1≤3) && (4≤5))

if (a[h] ≤ a[j]) ⇒ (8 ≤ 19) true

{

b[1] = 8 ;

h = 2 ;

i = 2 ;

}

while ((2≤3) and (4≤5))

if (13 ≤ 19) true

{ b[2] = 13 ;

h = 3 ;

i = 3 ;

}

while ((3≤3) and (4≤5)) true

if (24 ≤ 19) false

else

{ b[3] = 19 ;

j = 5 ;

i = 4 ;

}

while ((3≤3) and (5≤5)) true

if (24 ≤ 30) true

{ b[4] = 24 ;

h = 4 ;

i = 5 ;

}

while ((4≤3) and (5≤5)) false

}

if (4>3) true

{ for k = j to high

for k = 5 to 5

{ b[5] = a[5] ;

b[5] = 30 ;

i = 6 ;

}

}

for k = low to high

⇒ for k = 1 to 5

a[1] = 8 ;

a[2] = 13 ;

a[3] = 19 ;

a[4] = 24 ;

a[5] = 30 ;

II. Mergesort (6,10)

(6<10) true

mide $\frac{6+10}{2} = 8$

Mergesort (6,8) — IIA

Mergesort (9,10) — IIB

Mergesort (6,8,10) — IIC

IIA: Mergesort (6,8)

(6<8) true

mid = $\frac{6+8}{2} = 7$

Mergesort (6,7) → IIAa

Mergesort (8,8) → IIAb

Merge (6,7,8) → IIAc

IIAa: Mergesort (6,7)

(6<7) true

mid = $\frac{13}{2} = 6$.

Mergesort (6,6) → IIAai

Mergesort (7,7) → IIAaii

Merge (6,6,7) → IIAaiii

IIAai: if (6<6) false

IIAaii: if (7<7) false

IIAaiii: Merge (6,6,7)

{

h=6; i=6; j=7;

while ((h≤mid) and (j≤high))

{ while ((6≤6) and (7≤7)) true

if (a[h] ≤ a[j])

⇒ (51 ≤ 51) true

{

b[6] = 7;

h=7;

} i=7;

merge sort Analysis:-

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

$$T(n) = 2T(n/2) + O(n) \quad \text{---(1)}$$

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

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

$$= 2^2 (2T(n/8) + n/4) + 2n$$

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

$$T(n) = 2^k T(n/2^k) + kn$$

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$$

$$T(n) = n T(1) + \log_2 n \cdot n$$

$$T(n) = n + n \log n$$

$$\boxed{T(n) = O(n \log n)}$$

Best, worst Avg cases $O(n \log n)$

Strassens Matrix multiplication:

General method:

In order to multiply the two matrices, the no. of columns of the first matrix is equal to the no. of rows of the second matrix. The resultant matrix order is the no. of rows of the first matrix and the no. of columns of the second matrix.

Algorithm:

```
for i = 1 to n do
    for j = 1 to n do
        C[i][j] = 0;
        for k = 1 to n do
            C[i][j] = C[i][j] + a[i][k] * b[k][j];
```

The time complexity of above algorithm is $O(n^3)$
As it is having the more time complexity divide & Conquer technique is used in matrix multiplication.

Divide & Conquer:

Here we have to take the matrices of order "n x n" where 'n' is the powers of 2.

i.e., we have to take the matrices of order

2x2, 4x4, 8x8, 16x16 ----

* Let us consider the 4x4 matrix multiplication with 4x4 matrix
* In this method, the nxn matrix is divided into sub matrices of order $n/2 \times n/2$

Let's see how divide & conquer is applied for 4x4 matrix:

$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array}\right] \qquad B = \left[\begin{array}{cc|cc} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ \hline b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{array}\right]$$

$$C_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad C_2 = \begin{bmatrix} a_{13} & a_{14} \\ a_{23} & a_{24} \end{bmatrix} \quad C_3 = \begin{bmatrix} a_{31} & a_{32} \\ a_{41} & a_{42} \end{bmatrix}$$

$$C_4 = \begin{bmatrix} a_{33} & a_{34} \\ a_{43} & a_{44} \end{bmatrix}$$

* Here the nxn matrix is divided into sub matrices of order $n/2 \times n/2$
* The no. of multiplication operations for each sub-matrix is 8 and the no. of addition operations is 4.
* The recurrance relation is

$$T(n) = 8\,T(n/2) + O(n^2) \quad [n \geq 2]$$

$T(n)$ = constant where $n < 2$.

Here, $8T(n/2)$ is the time taken for multiplication and $O(n^2)$ is the time complexity for addition. On solving the above recurrance relation we get the time complexity $O(n^3)$

* Here we get the time complexity same as the general method. In order to reduce the time complexity strassens method is used with divide & conquer technique.

Strassens method:

Strassens introduces some formulae to perform the matrix multiplication. The formulae are as follows:

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

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

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

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

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

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

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

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

$C_{12} = R + T$

$C_{21} = Q + S$

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

In the above formulas the no.of multiplication operation is 7 and the no.of addition and subtraction operation is 18. Though the additions and subtractions are increased than the divide & conquer, The major operation in matrix multiplications is multiplication.

* The recurrance relation is

$$T(n) = 7T(n/2) + cn^2$$

$$T(n) = 7T(n/2) \Rightarrow T(n/2) = 7T(n/4)$$

$$T(n/4) = 7T(n/8)$$

$$T(n) = 7(7T(n/4))$$

$$= 7^2(7T(n/8))$$

$$= 7^3(T(n/8)$$

$T(n) = 7^3 T(n/2^3)$ in 3rd iteration.

After k^{th} iteration.

$$T(n) = 7^k T(n/2^k)$$

where $n = 2^k$

$$T(n) = 7^k T(1)$$

$$= 7^{\log_2 n}$$

$$= n^{\log_2 7}$$

$$= n^{2.8}$$

Hence the time complexity is $O(n^{2.8})$

* $n^{2.8}$ is slightly less than the n^3 hence the time complexity of matrix multiplication wing strassens method is reduced than the general method.

merge sort :-

Recurrence Relation :-

The time complexity of the recursive algorithms are expressed using an equation called as recurrence equation. The Recurrence relations are solved in 3 ways.

- Substitution method
- tree method
- master's method

(1) Substitution Method :-

$$T(n) = \begin{cases} T(n-1)+1 & n>0 \\ 1 & n=0 \end{cases}$$

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

$$T(n-1)+1+1$$

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

$$T(n-3)+3$$

$$\vdots$$

$$T(n-k)+k$$

$n-k=0$

$n=k$

$T(0)+k$

$= 1+k = 1+n = O(n)$

(2)

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

we can write it as $O(n)$

$$T(n) = \begin{cases} T(n-1)+n, & n>0 \\ 1 & n=0 \end{cases}$$

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

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

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

$= T(n-3)+3n-3$

$\vdots$

$= T(n-k)+kn-k$

Say $n-k=0 \Rightarrow k=n$

$= T(0)+n^2-n = 1+n^2-n = O(n^2)$

④ $T(n) = \begin{cases} 1, & n=0 \\ T(n-1)+\log n, & n>0 \end{cases}$

$T(n) = T(n-1) + \log n$

$= T(n-2) + \log(n-1) + \log n$

$= T(n-3) + \log(n-2) + \log(n-1) + \log n$

$= T(n-3) + \log(n-2) + \log(n-1) + \log n$

$= T(n-k) + \log(n-k-1) + \log(n-(k-2)) + \log n$

$n-k=0 \Rightarrow n=k$

$= T(0) + \log(n-n+1) + \log(n-n+2) + \log n$

$= 1 + \log(1) + \log 2 + \log n$

$= 1 + \log(1*2*3 \cdots *n)$

$= 1 + \log(n!)$

$= 1 + \log(n^n)$

$= O(n \log n)$

$\log a + \log b = \log ab$

$\therefore n! \simeq n^n$

⑤ $T(n) = \begin{cases} 2T(n-1)+1, & n>0 \\ 1, & n=0 \end{cases}$

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

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

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

$= 8T(n-3) + 4 + 2 + 1$

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

$n-k=0 \Rightarrow n=k$

$= 2^n T(0) + 2^n - 1$

$= 2^{n+1} - 1 = O(2^n)$

2^n

⑤ $T(n) = \begin{cases} T(n/2)+1, & n>1 \\ 1, & n=1 \end{cases}$

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

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

$$= T(n/8) + 1 + 2$$

$$= T(n/16) + 1 + 3$$

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

$$= T(n/2^k) + k$$

$$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$$

$$= T(n/n) + \log_2 n$$

$$= 1 + \log_2 n$$

$$= O(\log_2 n)$$

(6) $$T(n) = \begin{cases} 2T(n/2) + n & ; n > 1 \\ 1 & ; n = 1 \end{cases}$$

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

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

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

$$= 4(2T(n/8) + n/4) + 2n/2 + n$$

$$= 8T(n/8) + 4n/4 + 2n$$

$$= 8T(n/8) + 3n$$

$$= 2^k T(n/2^k) + kn$$

let $n/2^k = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$$= nT(n/n) + n(\log_2 n)$$

$$= n + n\log n$$

$$= O(n\log n)$$

(7). $T(n) = \begin{cases} T(\sqrt{n}) + 1 & n > 2 \\ 1 & n = 2 \end{cases}$

$T(n) = T(\sqrt{n}) + 1$

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

$= T(n^{1/4}) + 1 + 1$

$= T(n^{1/8}) + 1 + 2$

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

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

let $n^{1/2^k} = 1 \Rightarrow n$

let $n = 2^m$

$T(2^m) = T(2^{m/2^k}) + k$

$m/2^k = 1 \Rightarrow m = 2^k \Rightarrow k = \log_2 m$

$n = 2^m \Rightarrow m = \log_2 n$

$k = \log_2(\log_2 n)$

$T(n) = O(\log \log n)$

master Theorem :- $T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$

for $a \geq 1, b \geq 1, k \geq 0$ & p is a real number.

case 1 : $a > b^k$ then $T(n) = \Theta(n^{\log_b a})$

case 2 : $a = b^k$

i. if $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

ii. if $p = -1$ then $T(n) = \Theta(n^{\log_b a} \log \log n)$

iii. if $p < -1$ then $T(n) = \Theta(n^{\log_b a})$

case 3 : $a < b^k$

i. if $p \geq 0$ then $T(n) = \Theta(n^k \log^p n)$

ii. if $p < 0$ then $T(n) = O(n^k)$

Ex 1: $T(n) = 9T(n/3) + n$

$a = 9,\ b = 3,\ k = 1,\ p = 0$

$a > b^k$

so $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$

(2) $T(n) = 3T(n/4) + cn^2$

$a = 3,\ b = 4,\ k = 2\ \ p = 0$

$a < b^k$ & $p = 0$ then $T(n) = \Theta(n^k \log^p n)$

$T(n) = \Theta(n^2 \log^0 n) = \Theta(n^2)$

(3) $T(n) = 3T(n/4) + n \log n$

$a = 3,\ b = 4,\ k = 1,\ p = 1$

$a < b^k$ & $p = 1$ then ~~$T(n) = \Theta(n^k \log^p n)$~~

~~$T(n) = \Theta(n \log^1 n) = \Theta(n)$~~

~~if $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$~~

~~if~~ then $T(n) = \Theta(n^k \log^p n)$

$T(n) = \Theta(n \log^1 n) = \Theta(n \log n)$

(4) $T(n) = T(\frac{2n}{3}) + 1$

$T(n) = T(\frac{2}{3} \cdot \frac{2n}{3}) + 1 + 1$

$= T(\frac{2^2}{3^2} \cdot n) + 1 + 1$

$= T(\frac{2}{3})^2 \cdot \frac{2}{3} n) + 1 + 1 + 1$

$= T(\frac{2^3}{3^3} n) + 3 = T((\frac{2}{3})^3 n) + 3)$ x

$T(n) = T(\frac{2n}{3}) + 1 \Rightarrow T(\frac{n}{3/2}) + 1$

$a = 1,\ b = 3/2,\ k = 0,\ p = 0$

$a = b^k$ & $p > -1$ then $T(n) = \Theta(n^{\log_b a} \log^{p+1} n)$

$T(n) = \Theta(n^{\log_{3/2} 1} \log^1 n) = \Theta(n^{\log_{3/2} 1} \log n)$