I) Divide & conquer :-

i) Bubble Sort :-

Algorithm :-

```
algorithm bubblesort(A, n)
{
    temp;
    for i := 0 to n-1 do
    {   for j := 0 to n-i-1 do
        {   if arr[j] > arr[j+1]
            {   temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
}
```

→ Best case :- $O(n)$

→ Worst & Average case : $O(n^2)$

ex1

(5 1 4 2 8) $\xrightarrow{5>1}$ (swap) (1 5 4 2 8)

(1 5 4 2 8) $\xrightarrow{5>4}$ (swap) (1 4 5 2 8)

(1 4 5 2 8) $\xrightarrow[\text{swap}]{5>2}$ (1 4 2 5 8)

(1 4 2 5 8) $\longrightarrow$ (1 4 2 5 8)

$5 > 8$ no swap

} 1st iteration

(1 4 2 5 8) $\longrightarrow$ (1 4 2 5 8)

(1 4 2 5 8) $\xrightarrow[\text{swap}]{4>2}$ (1 2 4 5 8)

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

} 2nd iteration

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

(1 2 4 5 8) $\longrightarrow$ (1 2 4 5 8)

} 3rd iteration

Done

ii) Quick Sort:

```
Algorithm Quicksort(A, low, high)
{
    if (low < high)
    {
        j := partition(low, high);
        Quicksort(low, j-1);
        Quicksort(j+1, high);
    }
}
Algorithm Partition(l, h)
{
    pivot := a[i];
    i := l;
    j := h+1;
    while (i < j) do
    {
        i++;
        while (a[i] < pivot) do
            i++;
```

```
        j--;
        while (a[j] > pivot) do
            j--
        if (i < j) then
            Interchange (i, j);
    }
    interchange (l, j);
    return j;
}
Algorithm interchange (x, y)
{
    temp := a[x];
    a[x] = a[y];
    a[y] = temp;
}
```

→ Best case :- $O(n \log n)$

→ Worst case :- $O(n^2)$

→ Average case :- $O(n \log n)$

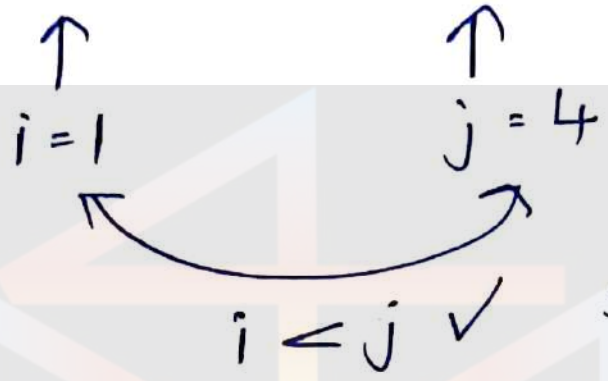
1.

ex: 8 9 7 6 5

pivote i j

0 1 2 3 4

i for greater & j for lesser



i < j ✓ swap i & j

8 5 7 6 9

pivote i j

j i j < i

swap pivote & j

6 5 7 8 9

combine

at the end → 5 6 7 8 9

6 5 7 8 9

↑ ↑ ↑

pivot i j

6 5 7

↑ ↑

j i

$j < i$ ✓

$p \in j$

5 6 7

=

$\therefore$ 5 6 7 8 9

=

iii) Merge Sort:

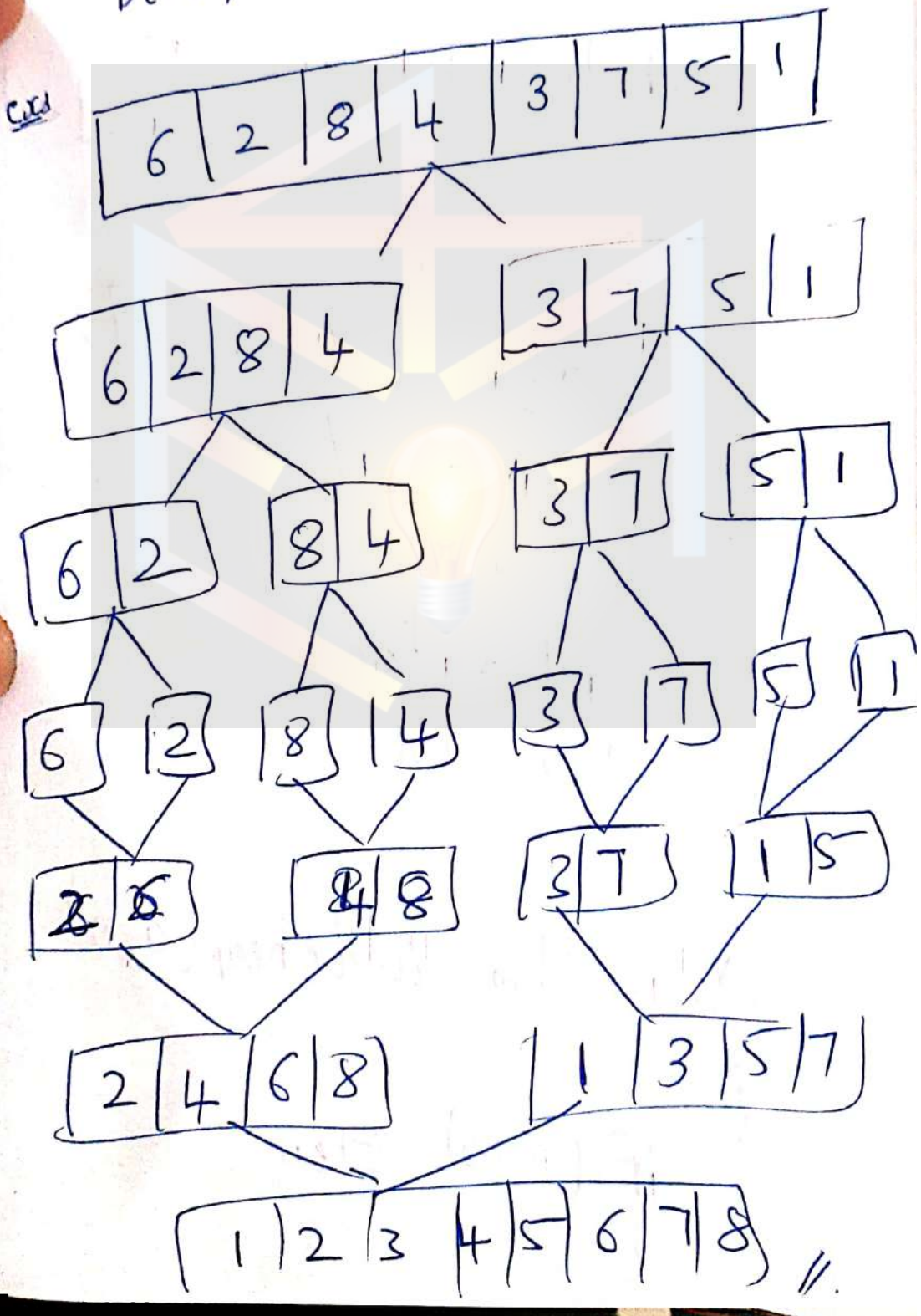
```
Algorithm mergesort(low, high)
{
  if (low < high) then
  {
    mid := (low + high)/2;
    mergesort(low, mid);
    mergesort(mid+1, high);
    // merge(low, mid, high);
  }
}
Algorithm Merge(low, mid, high)
{
  h := low;
  i := low;
  j := mid+1;
  while (h ≤ mid) and (j ≤ high) do
  {
    if (a[h] ≤ 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] := h[k];
}
}
```

Best, worst, average :- O(n log(n))

Ex:



//.

iv) heap :

```
Algorithm insert (a, n)
{
    i := n;
    item := a[n];
    while (i > 1 and a[i/2] < item)
    do
    {
        a[i] = a[i/2];
        i = i/2;
    }
    a[i] = item;
    return true;
}
Algorithm Delete heap(a, n)
{
    if (n = 0) then
    {
```

```
        write "Dedibion is not possible"
        return false;
    }
    else
    {
        x := a[1];
        a[1] := a[n];
        adjust (a, 1, n-1);
        return true;
    }
}
Algorithm adjust (a, i, n);
{
    j := 2i;
    item := a[i];
    while (j ≤ n)
    if ((j < n) and
            a[j] < a[j+1]))
    then
        j := j+1;
```

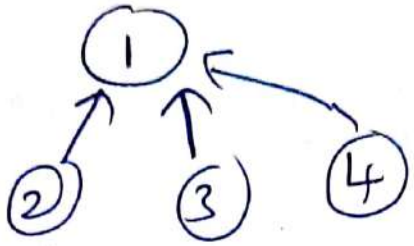
```
if(item >= a[j]) then
    break;
a[j/2] := a[j];
j = 2j;
a[j/2] = item;
}
Algorithm HeapSort(a, n)
{
    For i := 1 to n do
        insert(a, i);
    For i := n to step-1 do
        Deleteheap(a, i);
        a[i] := x;
}
```

Sets:

↳ collection of elements

$\{1, 2, 3, 4, 5\}$ ⟹ $S_1 = \{1, 2, 3, 4\}$

$S_2 = \{5\}$



=

→ In set trees we are linking children to parent instead of parent to children

→ 2 operations

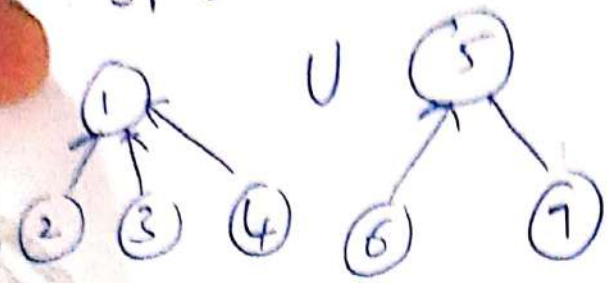
- Disjoined set union
- Find (i) — Given the element i we have to find the set containing i

If S_i & S_j are 2 disjoined set then $S_i \cup S_j$ is all elements in S_i or in S_j but not in both

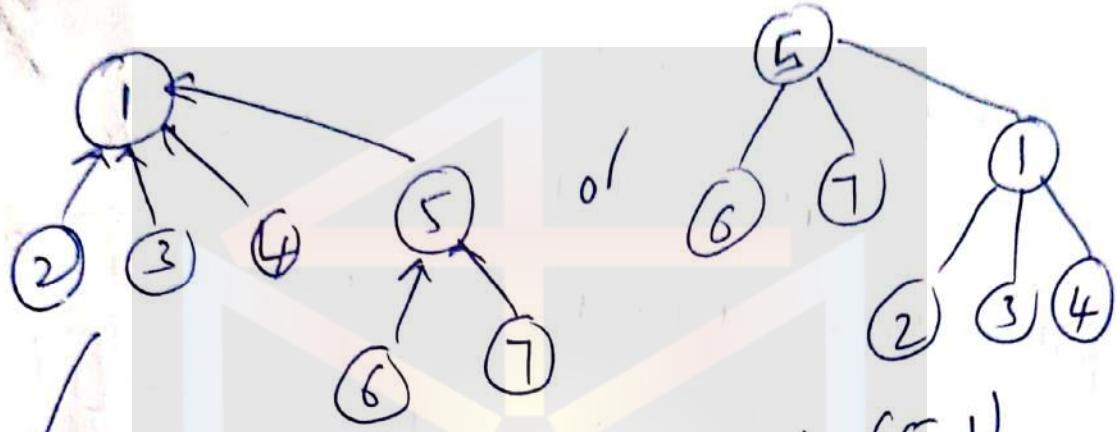
Find (3) - S_1

Find (5) - S_2

$S_1 \cup S_2$



make one tree as subtree of the other



or

Union (1,5)

Union (5,1)

Algorithm SimpleUnion (i, j)

```
{
    P[i] = j ; // or P[j] = i ;
}
```

1	2	3	4	5	6	7
-1	1	1	1	1	5	5

```
Algorithm SimpleFind(i)
{
    while (p[i] > 0) do
        i = p[i];
    return i;
}
```

7

collapsing find rule:

if i is a node & the path from i to it's root & p[i] ≠ root[i] then set p[i] to root[i]

```
Algorithm x collapsingFind(i)
{
    r := i;
    while (p[r] > 0) do
        r := p[r];
    while (i ≠ r) do
    {
        s := p[i];
        p[i] := r;
        i := s;
    }
}
```

return r;

}

Weighted union rule :-

If the no. of nodes in the tree with root i with < no. with root j then make j as parent of i otherwise make it as parent of j

Algorithm WeightedUnion(i,j)

// Union sets with roots i and j, i ≠ j, using // weighting rule p[i] = -count[i] & p[j] = -count[j];

{

temp := p[i] + p[j];

if (p[i] > p[j]) then

{

```
// i has fewer nodes
    p[i] := j;
    p[j] := temp;
  }
  else
  { // has fewer or equal nodes
    p[j] := i;
    p[i] := temp;
  }
}
```

#include <unistd.h>