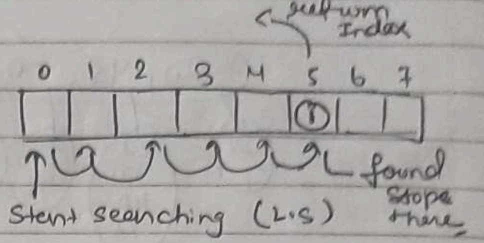
Searching and Sorting

- Linear search →
- Binary search




```
int LS ( int a[ ] , int size , int v)
         or int *a   (8)
         array      size of array   search value
{
   for (i = 0;  i < size,  i++)
   {
       if  (a[i] = v)
            return  i ;          } O(n)
   }
If NOT (Never Matches)
       return -1 ;
}
```

- It is Simple
- It works both on Array as well as LinkedList.

Binary Search → Assumption, Input data must be in Sorted manner.

Data →	3	9	12	17	45	64	73	81	85	86
Indexes →	0	1	2	3	4	5	6	7	8	9

l (position) . . . . . . u   m   l . . . . . . u

find $m = l + u/2$

$= 0 + 9/2 = 4$

If (a[m] == v)
↳ Done

If not

If (v < a[m])
{ u = m - 1
}

or If (v > a[m])
{
$l = m+1$
}

Now Do repetation of these whole code.

```
int BS (int *a, int l, int u, int v)
        array    beg pos.  end pos.  value
```

⊙ first. Insert l and u by the user.

```
while (l <= u)
{
    m = l+u/2 ;

    if (a[m] == v)
        return m;      — found case

    if (v < a[m])
        u = m-1
    else
        l = m+1
}
return -1;     — NOT found case
```

O(log n)

Sorting.

I Insertion Sort. $O(n^2)$

↳ I have some sorted data and if an element come my prev. few data are sorted and that new element you had to take a decision where to Insert

2 · 5 7 4

Array →

0	1	2	3	4	i = 5
5	7	10	13	14	8

j, (j+1) '8', j, (i-1)

14 = temp = 8

If you not store '8' after compare when it is shuffled '8' get deleted

① first element doesn't compare with any thing.

```
if (a[j] > temp)
{
    a[j+1] = a[j]      } shifting condn
}

a[j+1] = temp          } placed condn
```

code :-

```
Void Insert_Sort (int *a, int size)
{   int tmp, j;
    for (int i = 1; i < size, i++)      // To do too many element sorts.
    {                                   // i = 1: 1st element is sorted
        tmp = a[i];
        for (j = i-1; j >= 0; j--)
        {
            if (a[j] > tmp)
            {
                a[j+1] = a[j]
            }
```

$O(n^2)$

$$\left(\frac{1+2+3+\cdots n}{\frac{n(n+1)}{2}} \right)$$

```
        else
        {
            break;
        }
    } // Inner for loop end
    a[j+1] = temp;
} // End of for loop.
```

⊙ If our data is already in a sorted manner then, Insertion sort performs with $O(n)$. bcz at least this for loop has to be continue for checking ~~else~~ every element Insertion.

Best case = $O(n)$
Worst case = $O(n^2)$

Bubble-Sort $O(n^2)$

Size - 8 (n)

Iteration		0	1	2	3	4	5	6	7	
		2	9	6	4	1	5	7	3	
i=0	1	2	6	4	1	5	7	3	9	(n-1)
i=1	2	2	4	1	5	6	3	7	9	(n-2)
i=2	3	2	1	4	5	3	6	7	9	
i=3	4	1	2	4	3	5	6	7	9	
i=4	5	1	2	3	4	5	6	7	9	
i=5	6									
i=6	7	1	2	3	4	5	6	7	9	

j=0

By writing this you are reducing the half of the moment (Almost Half)

$\Rightarrow n + (n-1) + (n-2) + \dots$

$\Rightarrow \frac{n(n+1)}{2}$

$= O(n^2)$

~~By data~~

```
for ( i=0 ; i < size-1 ; i++ )
{
    for ( j=0 ; j < size-i-1 ; j++ )
    {
        flag = 0;  // If no swap take place
        if ( a[j] > a[j+1] )
        {
            swap ( &a[j], &a[j+1] )          O(n²)
            flag = 1 ;  // swap take place
        }
    }
    if ( flag == 0 )
    {
        break;
    }
}
```

Think there are 1000 of data but after 100 data ~~swap~~ it is already Sorted, then no need to go further. So, (flag == 0)

It reduce the complexity.

Best Case → O (n) < already Sorted / flag code take place

Page No.

Selection Sort. $O(n^2)$

Ex:- 2 9 6 1 4 7 3

1 9 6 2 4 7 3 ⊙ In this case every time, comparision happens

1 2 9 6 4 7 3 ⊙ Here, too many swap occurs.

1 2 3 9 6 7 4

:

In case of Bubble sort last side sorting happens but in Selection sort sorting done in the front.

Another method → 1 9 6 2 4 7 3

compar — minm

└ find minm among these then, compare it

└, It is better bcz finding the minm take O(n) time.

Here Only one time swap take place.

⊙ Here, we are also comparing too many time but not swapping.

size = 8

	0	1	2	3	4	5	6	7
Ex:- i=1	2	9	6	1	4	7	3	8

i j min

└, minm then swaps with '3' to '0' index

i=2 1 9 6 2 4 7 3 8

└, minm

i=3 1 2 6 9 4 7 3 8

└, minm

i=4 1 2 3 9 4 7 6 8

└, minm

i=5 1 2 3 4 9 7 6 8

i=6 1 2 3 4 6 7 9 8 — no swap

i = 7 ① ② ③ ④ ⑥ ~~⑧~~ ⑦ 9 8

Done → ① ② ③ ④ ⑥ ⑦ ⑧ ⑨

```
                         ┌→ selected pos.
O(n) ——— for ( i = 0;  i < size - 1;  i++ )
         {
                 min = i + 1;        // pos.
(n-1)  ——————  for ( j = i+2; j < size; j++ )
 n-2            {
 n-3                if (a[j] < a[min])
  :     finding     {
 ①      min            min = j;
                    }
                }
```

But Here we are comparing 1 less element

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

```
         if (a[min] < a[i])
O(n)     {
             swap (& a[min], & a[i])
         }
}
```

If

```
min = 1
for (j = i+1; j<size, j++)
{
   ==
}
if (min != i)
{
   Swap (&a[min],
         &a[i]);
}
```

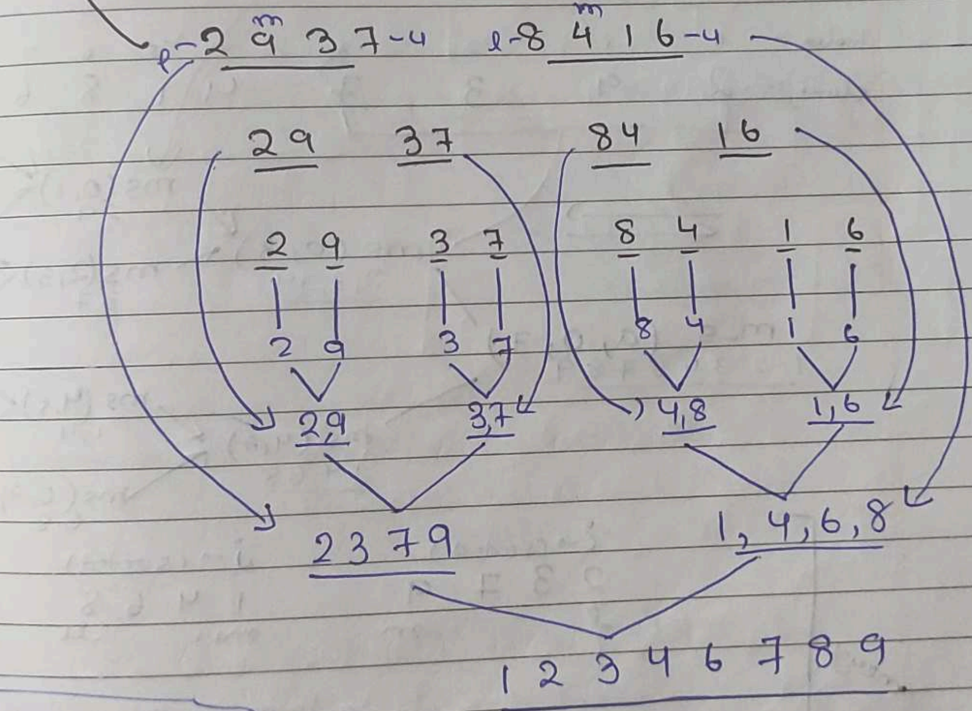
Page No.

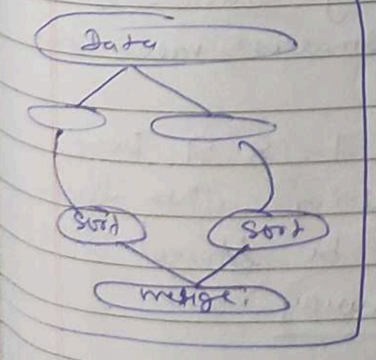
Merge Sort. O(n log n)

l - 0 1 2 3 4 5 6 7 -4 $m = \frac{l+4}{2}$

2 9 3 7 8 4 1 6

If I give you 'n' such data, then you divide this part into two different parts then, short those both have and then give it to me and I merge them both sorted element (arrays) (repeate this)





$O(n \log n)$ — merge-sort (int *a, int l, int u)

{ if (l < u)

{ $m = \frac{(l+u)}{2}$;

merge-sort (a, l, m);

recursive fn — merge-sort (a, m+1, u); log(n)

merge (a, l, m, u); O(n)

}

Index :- 0 1 2 3 4 5 6 7

element :- 2 9 3 7 4 1 8 6

m-s (a, 0, 7)
1 2 3 4 6 7 8 9

ms (0, 3)
2, 3, 7 9

ms (0, 1)
2 9

ms (0,0) 2

ms (1,1) 9

ms (2, 3)
3 7

ms (2,2) 3

ms (3,3) 7

ms (4, 7)
1 4 6 8

ms (4, 5)
1, 4

ms (4,4) 4

ms (5,5) 1

ms (6, 7)
6, 8

ms (6,6) 8

ms (7,7) 6

merge

i → (sorted) j → (sorted)

2 3 7 9 1 4 6 8

l m m+1 u

Among this 'i' & 'j' which ever is smallest value that come first.

→ 1 2 3 4 6 7 8 9 Done.

① After comparing the last result has to be stored in another array

Out_array | 1 | 2 | 3 | 4 | . | | |
k=0

O(n) - merge (int *a, int l, int m, int u)
{
int i, j, k = 0;
find array to store { int *Out_array = malloc (sizeof (int) * (u - l + 1))
i = l ; j = m+1;

While (i <= m && j <= u)
{
if (a[i] < a[j])
{
Out_array [k++] = a[i];
}
else
{
Out_array [k++] = a[j++];
}
}
(No comparision take place then) While (i <= m)
{
'i' value left out || Out [k++] = a[i++];
}
While (j <= u)
{
'j' value left out || Out_array [k++] = a[j++];
}

} // end of merge function
for (x = 0 ; x < K ; x ++)
{
a [l + x] = Out_array [x];

O(n) Space Complexity
}
but Quick Sort will not need

Quick Sort $O(n \log n)$

swap.

5	7	2	9	8	1	12	10	13	4

l — pivot — u

Choose a pivot then,

To achive this

l ⟶
u ⟵
l u
swap

______ (pivot) ______
less greater

5 4 2 9 8 1 12 10 13 7
l — l → l swap u ← u ← u ← u ← u

5 4 2 1 8 9 12 10 13 7
swap → 4 — u

1 4 2 5 8 9 12 10 13 7

Pivot | (Again do same) | (it's pos. is fix) | Pivot | (Again do same)

Repeat

① [a[l] <= pivot
l++;]

② [a[u] > pivot
u--;]

③ [$l_{pos.} < u_{pos.}$
If (yes)
↳ Swap (them)]

Swap (u, pivot pos. val)

Now, that pivot pos. is fix.

```
Q-sort (int *a, int l, int u)
{   if (l < u)   // at least two element present
int pos = partition (a, l, u)  ← O(n)
        Q-sort (a, l, pos-1);
        Q-sort (a, pos+1, u);
}   }
```

usually $O(n \log n)$

worst case ; $O(n^2)$

Page No.

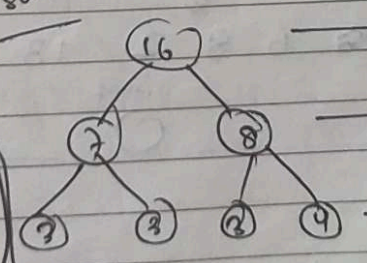
O(n) int partision (int *a, int i, int j)

```
{
        int l , u , piv ;
            l = i    ; u = j
                piv = a[i] ;
        while ( l < u
        {
            while ( a[l] <= piv && l<j)
            {
                l++ ;
            }
            while ( a[u] > piv)
            {
                u-- ;
            }
            if ( l<u)
            {
                Swap (&a[l], &a[u]);
            }
        } // end of while loop

        Swap (&a[i] , &a[u]) ;
        return u; // partising pos.
}
```

no extra space is needed in Quick Sort but In Merge sort in every label we need a extra memory space

In Quick sort



16 (1) reduce 2^0

15 (2) reduce 2^1

13 (4) reduce 2^2

But in merge sort no one element is reducing label wise i.e All elements are participating

but In Quick sort all element are not participating

Pivot

ex:- 1 3 7 11 15 17 n

$O(n \times n)$

$\Rightarrow O(n^2)$

Pivot

(1) 3 7 11 15 17 (n-1)

(1) (3) 7 11 15 17 (n-2)

Here, Only one element got reduced, If all are already sorted

- In ~~[illegible]~~ this 5 4 3 2 1
  - ↳ also take $O(n^2)$
  - ↳ If you want to make it $O(n\log n)$ then make partition somehow in middle

Ex:- 2 6 8 9 13 15 → It take '6' steps.

↳ To reduce it's complexity use Randomized Quick Sort

Since, you know the 'l' & 'u' while choosing the pivot don't choose the first element as pivot

↳ r = Random (i, j)

swap (&a[i], a[r])

piv = a[i]

9 6 8 2 13 15

pivot

n log n

Quick Sort

Also known as partition-Exchange sort. an efficient sorting algo.

Work on Divide & Conquer.

	0	1	2	3	4	5	6	7	8
ex:- arr	15	9	7	13	12	16	4	18	11

Pivot

i = 0 Pivot 12 J = 8

first choose any element from this array called pivot.

choose - first | last | Middle

Left hand side of pivot all are small from pivot and Right hand side are bigger than pivot.

(Increment i and Decrement J)

```
while (i <= j)
{
  while (pivot > arr[i])
  {
     i++;
  }
  while (arr[j] > pivot)
  {
     j--;
  }
  if (i <= j)
  {
     swap;
     i++;
     j--;
  }
}
```

if both false then swap both I & j

(11, 9, 7, 13, 12, 16, 4, 18, 15)

then decrease j and Incr i

Pivot

11, 9, 7, 4, 12, 16, 13, 18, 15

divide both and again same thing

```
Class Quicksort {
        p s v m ( — )
        {
            int [] arr = {15, 9, 7, 13, 12, 16, ...
            int length = arr.length;
            Quicksort m = new Quicksort();
            m.quicksort_Recursion (arr, 0, len...
            m.printArray (arr);

    int partition ( int [] arr, int low, int high)
    {
        int pivot = arr [low + high / 2];
        while ( low <= high)
        {
            while (arr[low] < pivot)
            {
                low ++;
            }
            while (arr [high] > pivot)
            {
                high --;
            }
            if ( low <= high)
            {
                int temp = arr[low];
                arr [low] = arr[high];
                arr [high] = temp;
                low ++;
                high --;
            }
        }
        return low;
    }
}
```

```
Void quick_SortRecursion (int [] arr, int low, int high)
{
    int pi = partition (arr, low, high)
        if ( low < pi -1)   bcz we don't need the pivot element
        {
left //      quicksortRecursion (arr, low, pi-1)
        }
        if (pi < high)
        {
Right //     quicksortRecursion (arr, pi, high);
        }
}

Void printArray (int[] arr)
{
    for (int i : arr)
        { sop (i);
        }
}
```