# SORTING

→ Arranging elements of an array in ascending order.

- Classification; 1) Number of Comparisions
  2) No. of Swaps 3) Memory Usage
  4) Recursion 5) Stability & Adaptability

  → Internal Sort (In-Sort)
  (use main memory exclusively)

  → External Sort (Out-Sort)
  (uses external memory)



## 1) BUBBLE SORT (sinking sort)

→ Traverse through the list in adjacent manner one by one if wrongly placed swaps them. Peforms traversing till each element is placed correctly at end after one looping.

Time Complexity: $O(n^2)$ (Best) Worst Case)
Space " : $O(1)$ auxillary

Implementation;

```
void bubbleSort ( int a[], int size)
{
    for ( int step = 0 ; step < size - 1 ; step++)
    {
        for( int j = 0 ; j < size - step - 1 ; j++)
        {
            if ( a[j] > a[j+1])
            // swap ; ( swaps element )
        }}}
```



* In improved version best case can be resolved to O(n) by adding a variable swapped if false, inner loop will not execute further

```
→ void bubbleSort (int a[], int size){
    int i, j, swapped = 1 , temp;
  for ( i = size - 1; i >= 0 && swapped ; i--) {
        swapped = 0;
        for ( j = 0 ; j < i - 1 ; j++) {
            if ( a[j] > a[j+1] ) {
                swap ;
                swapped = 1;
3333
```

} if cond. executes on one looping the swapped = 1 else the list is sorted

- **SELECTION SORT** ; → Takes smallest element & exchange it with first element in the list & continues
  → Similarly, takes largest element & exchange it with last element
  → Improves on bubble sort by making one exchange for every pass through the list.



**Advantages** ; • easy implementation
• In-place sorting (i.e. no additional space required)



**Dis-advantage** ; • Time → $O(n^2)$

**Implementation** ;

```
void SelectionSort (int a[], int size){
    int i, j, min, temp;
    for(i=0; i<size-1; i++){
        min=i
        for(j=i+1; j<size; j++){
            if(a[j] < a[min])
                min=j;
        }
    temp = a[min];
    a[min] = a[i];
    a[i] = temp;
}}
```

OR

```
void selectionSort (int a[], int size){
  int i,j,max,temp;
  for(i=size-1; i>=0; i--){
    max=i;
    for(j=i-1;j>=0;j--){
      if (a[j]>a[max])
        max=j;
    }
    temp = a[max];
    a[max] = a[i];
    a[i]= temp;
  }
```



INSERTION SORT, picks element 1-by-1 and places it to right position where it belongs in sorted list.

- Simple & effecient comparision sort.
- Each iteration removes an element from I/P list and inserts it into sorted sublist.
- In-place - requires a constant amount O(1) of additional memory.

Implementation;

```
void insertionSort (int a[], int size) {
for(int i = 1; i <= size -1; i++) {
  int key = a[i]; j = i;
  while (a[j-1] > key && j>=1){
        a[j] = a[j-1];
        j--;
  }
  a[j] = key; }}
```



Example;

4 9 11 45 27

(Sorted Sublist) (inserted to sorted Sublist)

→ 4 9 11 45 45 (27<45 copy 45 at A[4])

→ 4 9 11 45 45 (next is 11, 27>11, so copy 27 at A[3])

→ 4 9 11 27 45

(2)

→ i=1, j=1 ( 3 8 4 2 9 1 )

(3>8)x → ( 3 8 4 2 9 1 )

i=2=j

key=4 { (8>4) → ( 3 8 8 2 9 1 ) (copy 8 at a[2])
j=1
(3>4)x → ( 3 4 8 2 9 1 ) (Insert 4 at a[1]

key=2 {
i=3=j
(8>2) → ( 3 4 8 8 9 1 ) (copy 8 at a[3])
j=2
(4>2) → ( 3 4 4 8 9 1 ) (copy 4 at a[2])
j=1
(3>2) → ( 3 3 4 8 9 1 ) (copy 3 at a[1])
j=0 → ( 2 3 4 8 9 1 ) (Insert 2 at a[0])

i=9=j
key=9 (8>9)x ( 2 3 4 8 9 1 )

Key=1 {
i=5=j
(9>1) ( 2 3 4 8 9 9 ) (copy 9 at a[5])
j=4
(8>1) ( 2 3 4 8 8 9 ) (copy 8 at a[4])
j=3
(4>1) ( 2 3 4 4 8 9 ) (copy 4 at a[3])
j=2
(3>1) ( 2 3 3 4 8 9 ) (copy 3 at a[2])
j=1
(2>1) ( 2 2 3 4 8 9 ) (copy 2 at a[1])
j=0

( 1 2 3 4 8 9 ) (Inserts 9 at a[0])

- Insertion sort is used when data is nearly sorted or when i/p size is small and due to its stability (low overhead)

→ Time Complexity → $T(n) = \theta(1) + \theta(2) + \cdots \theta(n-1)$

$T(n) = \theta\left(\frac{n(n-1)}{2}\right)$

$T(n) = \theta(n^2)$ [Best/worst case]

→ Space Complexity → $O(n^2)$ total, $O(1)$ auxillary



- Shell Sort: (Diminishing increment sort)
  ↳ n-gap insertion sort
  - Shell sort makes several passes & uses various gaps b/w adjacent elements, variation in shell sort is to avoid comparing adjacent elements until last step, so effectively performs insertion sort.
  - Let the gap be of $\frac{size}{2}$ & decrement of $\frac{1}{2}$ by every looping.
  - Swap the values
  - at gap = 1, shell sort will perform insertion sort

Example;

gap = 3

34 1 42 8 23 6

34 ——— 8 (Sort 8 & 34)

1 ——— 23

42 ——— 6 (Sort 42 & 6)

gap = 1

perform insertion sort

→ 8 1 6 34 23 42

→ 8 8 6 34 23 42

→ 1 8 8 34 23 42

→ 1 6 8 34 34 42

→ 1 6 8 23 34 42 (Sorted)



Complexity; Space; O(1) auxillary

Time; O(nlog²n) (best case)

O(n²) (worst case)

- Also known as inse n-gap insertion sort.

Sahitya Bhawan

- Merge Sort;

Implementation:

```
void ShellSort (int a[], int size) {
    int i, j, h, v;
    for(h=1; h = size
    while (h < size/3)
    h = 3*h + 1;          } used to calculate gap b/w elements
    for ( ; h>0; h=(h-1)/3 ){
        for( i=h; i<size; i++){
            v = a[i]; j=i;
            while( a[j-h] > v && j>h-1){
                a[j] = a[j-h];
                j -= h;
            }
            a[j] = v;
        }
    }
}
```

Performing Insertion sort at h gap in given array

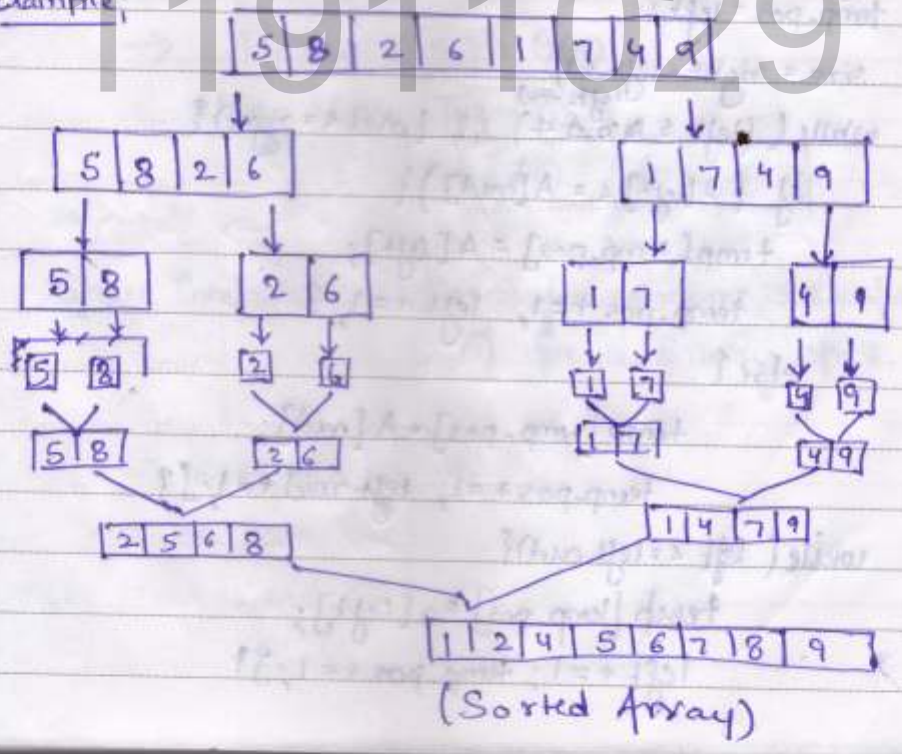
- In place algorithm.
- Average Complexity; $O(n^2 \log^2 n)$ or $O(n^{3/2})$
  for best case; $O(n)$

---

- **MERGE SORT**; is an example of divide & conquer strategy.
  - Divides the array into equal halves & then combine them in sorted way.
  - Recursive algo that splits array into $\frac{1}{2}$.



Example;




(Sorted Array)

Implementation; void Mergesort(int a[], int temp[], int left, int right)
{
int mid;
if (right > left){
mid = (right + left)/2;
Mergesort(A, temp, left, mid);
Mergesort(A, temp, mid+1, right);
Merge(A, temp, left, mid+1, right);}}
void Merge(int a[], int temp[], int left, int mid, int right){
int i, size, temp_pos; int left_end = mid-1;
temp_pos = left;
size = right - left + 1;
while((left <= left_end) && (mid <= right){
if(A[left] <= A[mid]){
temp[temp_pos] = A[left];
temp_pos += 1; left += 1;}
else{
temp[temp_pos] = A[mid];
temp_pos += 1; mid += 1;}}
while(left <= left_end){
temp[temp_pos] = a[left];
left += 1; temp_pos += 1;}}

```
while(mid <= right){
    temp[temp_pos] = a[mid];
    mid++; temp_pos++;}
for(i=0; i<=size; i++){
    a[right] = temp[right];
    right = right-1;
```



- Out place algorithm as it needs temp array to assign sorted values.

→ $T(n) = 2T(\frac{n}{2}) + \theta(n)$

Using Master Theorem;

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

Space Complexity; $O(\log n)$ for runtime stack space
$O(n)$ for auxillary space.

# Quick Sort

- famous comparision based algorithm.
- sorts using divide & conquer strategy.
- No additional storage required as it is in-place algorithm unlike mergeSort

- The only disadvantage is worst-case time-Complexity is $O(n^2)$.



→ Divide & Conquer Strategy;

- first select the first element of list as pivot value.
- The pivot value roles to assist the spilliting of list into 2-sublists.
- After sorting, the pivot value position is referred as partition point.
- All the elements in first sublist must be smaller than pivot value &,
- In 2nd list must be greater than pivot value

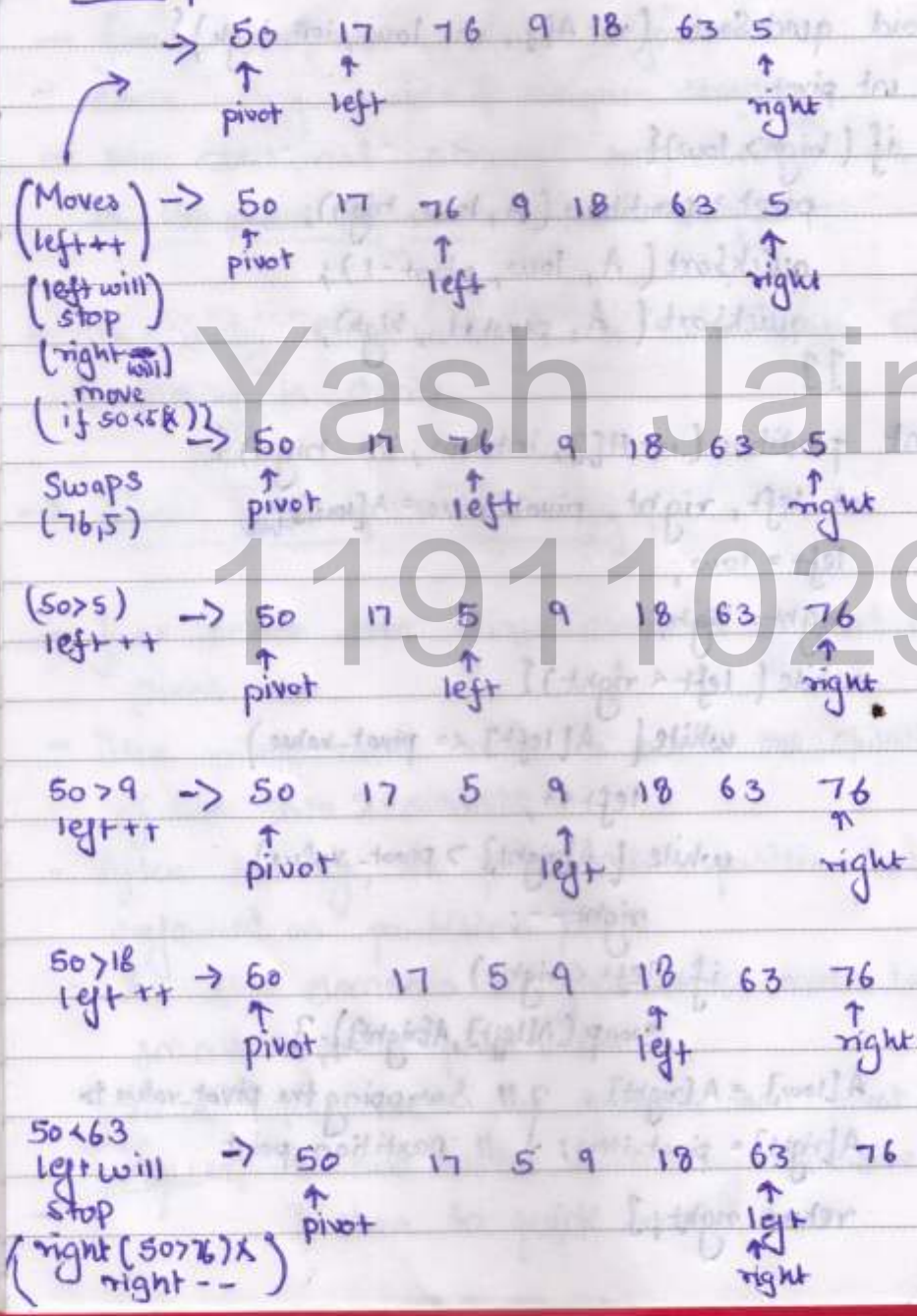
Conquer; The two sublist sorted by recursive calls & then to quick sort

Sanitya Bhawan

## ALGO. IMPLEMENTATION:

```
void quickSort (int A[], int low, int high){
  int pivot;
  if (high> low){
      pivot = partition (A, low, high);
      quickSort( A, low, pivot-1);
      quickSort( A, pivot+1 , high);
  }}
int partition( int A[], int low, int high) {
  int left, right, pivot_value = A[low];
  left = low;
  right = high;
  while( left < right){
        while( A[left] <= pivot_value)
            left++;
        while ( A[right] > pivot_value)
            right--;
        if (left < right)
            swap (A[left], A[right]); }
  A[low] = A[right]; } // Swapping the pivot_value to
  A[right] = pivot_item; } // partition point
  return right; }
```

Example;



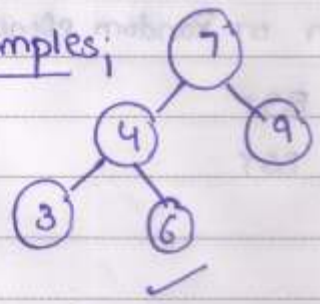
# Binary Search Trees; (BST)

- In Binary Trees there is no node restrictions i.e. why worst case complexity of search is $O(n)$.
- whereas in BST imposing some restrictions on node we make worst case complexity of search $O(\log n)$.

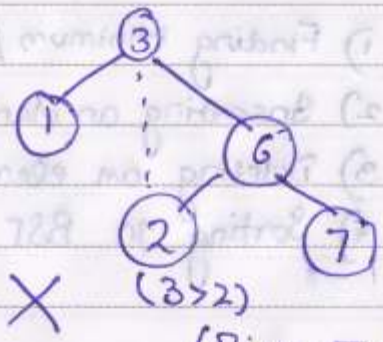


- BST Property; follows that left subtree data should be less than root data
  - & right subtree data should be greater than root data.
  - Implies on all nodes in BST.

Examples;




(BST)




(3>2)

(Binary Tree)

- **BST Declaration;**

```
Struct BST {
    int data;
    Struct BST * left;
    Struct BST * right;
} *root;
```

Structure of BST




- **Operations on BST;**

1) Finding Minimum / Maximum or Random element in BST.
2) Inserting an element in BST
3) Deleting an element in BST
4) Sorting in BST