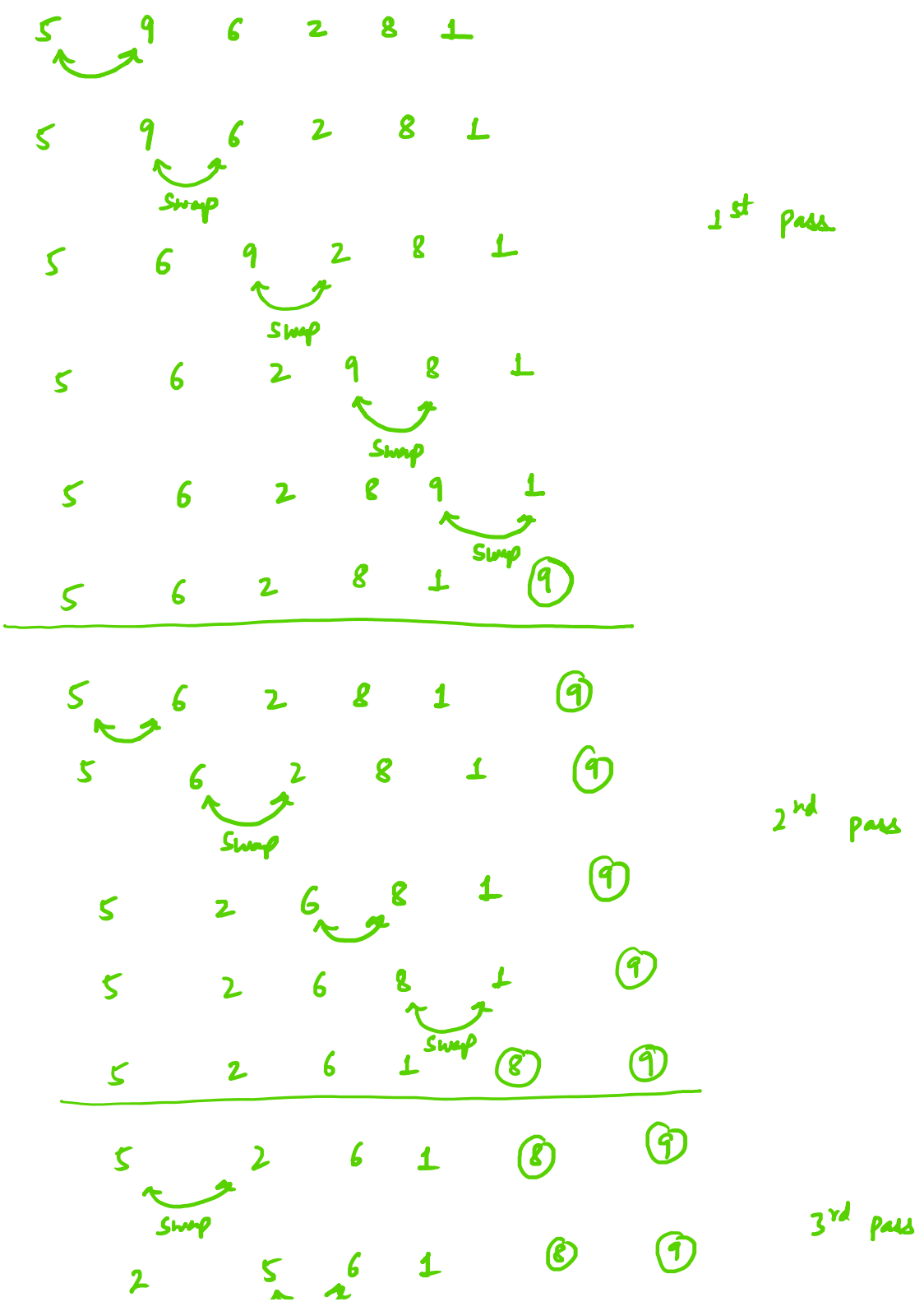
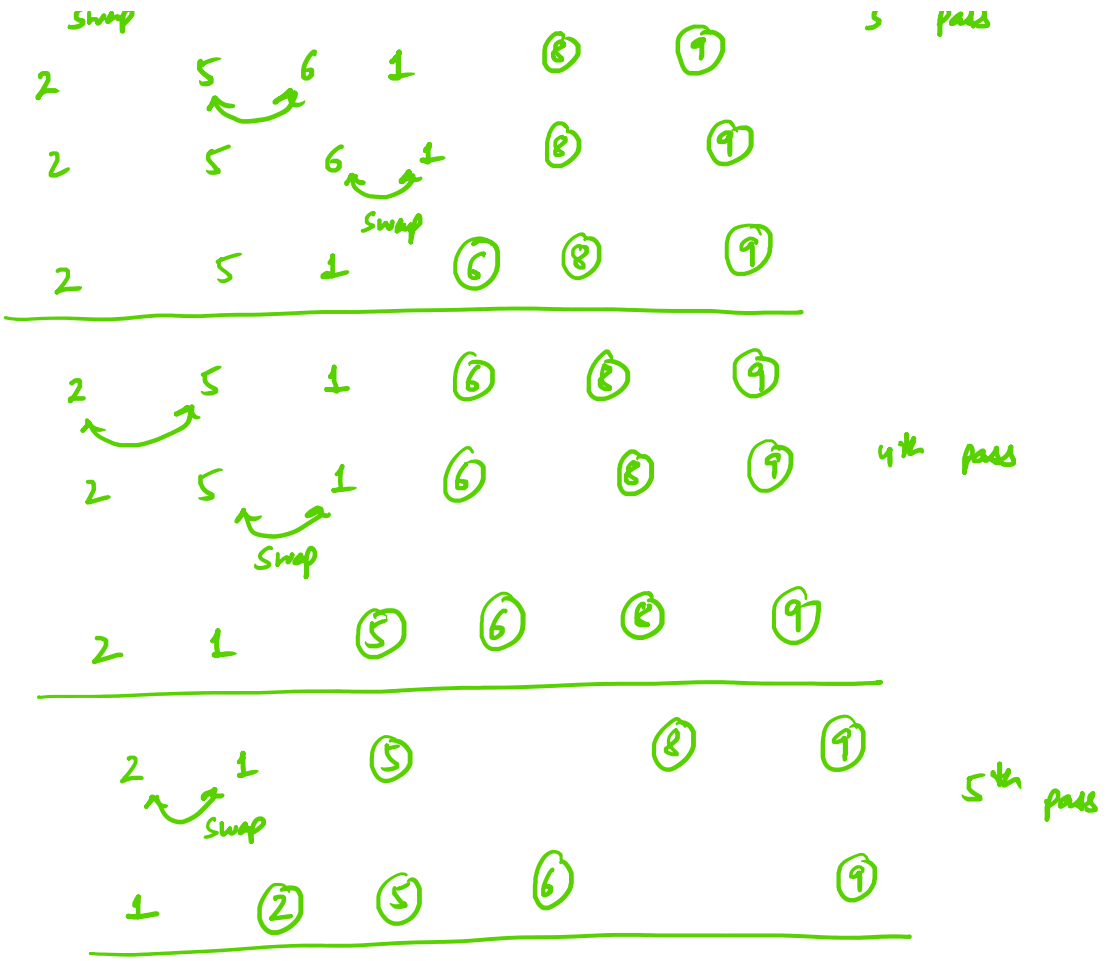
Sorting :- Arrangement of data in either ascending or descending order.

1) Bubblesort :- Bubbles rises up in water

B - biggest element should be placed at last.

eg:- A = { 5, 9, 6, 2, 8, 1 } , n = 6





For n sized data, we require atmost n-1 passes

```
void bubble_sort (int a[], int n)
{
    int i, j, temp;
    for ( i = 1; i < n ; i++)
        for (j=0 ; j< n-i ; j++)
            if( a[j] > a[j+1] )
            {
                temp = a[j];
                a[j] = a[j+1];
                a[j+1]= temp;
            }
}
```

Time Complexity
= Comparison time
+ Swapping time

$\sum_{i=1}^{n-1} n-i \simeq n^2$

$\Omega(n^2)$

$O(n^2)$

$\theta(n^2)$

Improved Version

```
void improbable_sort (int a[], int n)
```

```
{
    int i,j, temp, flag = 1;
    for ( i= 1; i <n && flag == 1; i++)
    {
        flag = 0;
          for (j=0; j< n-i ; j++)
                if( a[j] > a[j+1])
                {
                   temp = a[j];
                    a[j] = a[j+1];
                    a[j+1]= temp;
                 } flag = 1;
    }
```

1, 2, 3, 4, 5

$\Omega(n)$

$O(n^2)$

$\theta(n^2)$

In-place sorting algorithm – No extra space required

Stable sorting algorithm – If 2 elements are same, then the order of these elements doesn't change after sorting

eg:- 5, 2, 8, 6, 2*, 9

2, 2*, 5, 6, 8, 9 → stable

2*, 2, 5, 6, 8, 9 → unstable

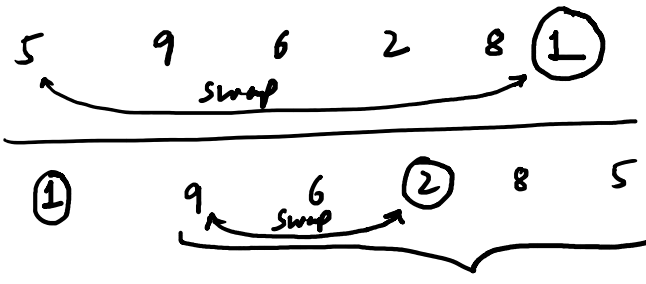
Selection Sort :- S – select the smallest element

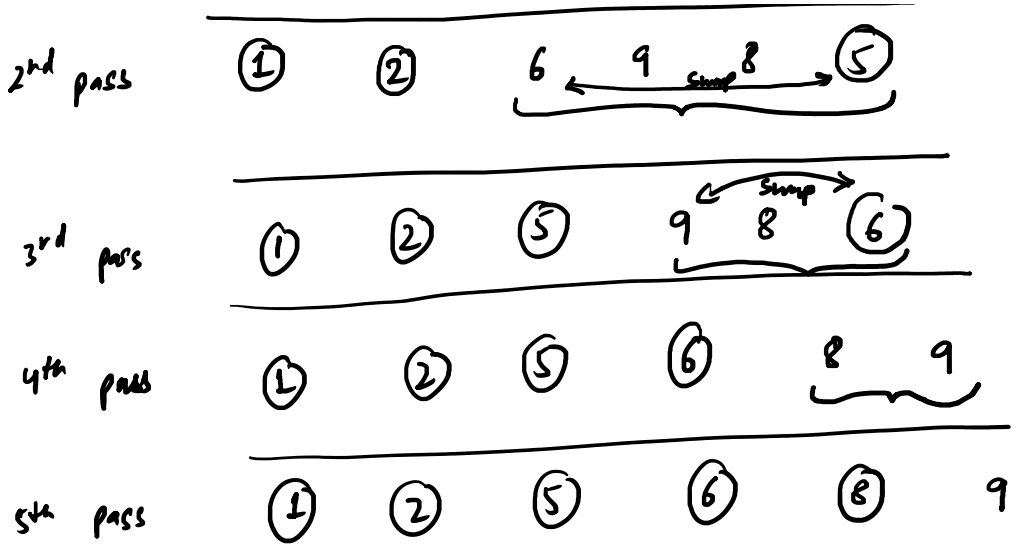
ith pass : starting i-1 th elements are sorted.

a[i .. n-1] → select the smallest element. Lets say its index is K

swap a[i] ↔ a[k]

A = { 5, 9, 6, 2, 8, 1}, n = 6





Selection sort requires least no. of swaps

Max. no. of swaps $= n-1$

```
void selection_sort (int a[], int n)
{
    int i, j, temp, k;
    for (i = 0; i < n-1; i++)
    {
        k = i;
        for( j = i+1; j < n; j++)
            if (a[j] < a[k])
                k = j;

        if (k != i)
        {
            temp = a[i];
            a[i] = a[j];
        }   a[j] = temp;
}
```

$\Omega(n^2)$

$O(n^2)$

$\theta(n^2)$

In-place sorting algorithm

Not stable

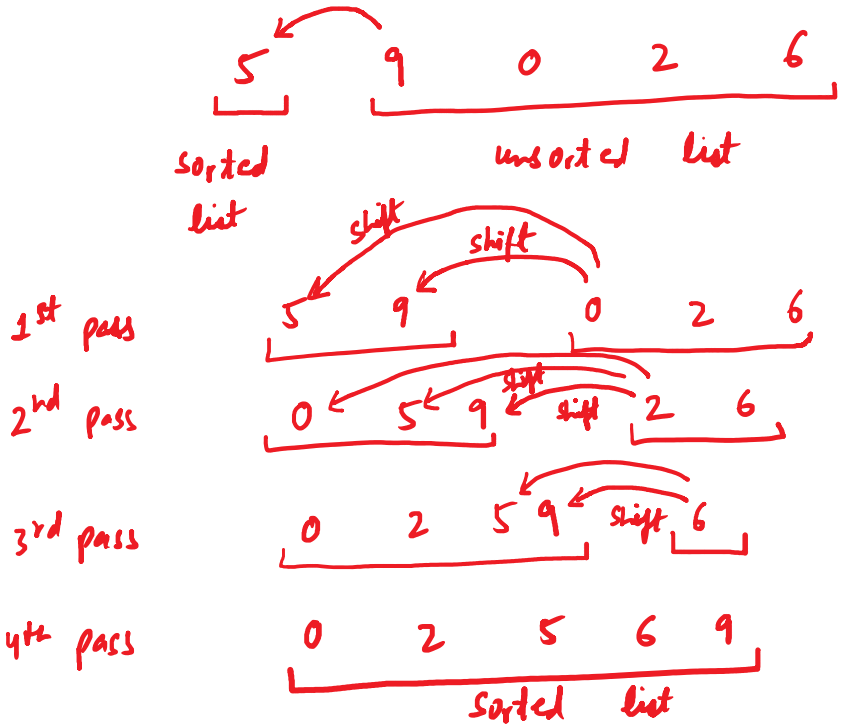
1, 8, 6, 9, 4, 1*

Insertion Sort :- Insert the element at the correct location in the sorted list.

2 types of list → Unsorted list
→ Sorted list

After the i-th pass, starting i elements are placed at correct locations.

A = { 5, 9, 0, 2, 6 } , n = 6




```
void insertion_sort (int a[], int n)
{
    int i, j, temp;
    for (i=1; i<n; i++)
    {
        temp = a[i];
        for( j= i-1; j>=0 && temp < a[j] ;j--)
            a[j+1] = a[j];
    }   a[j+1] = temp;
}
```

$\Theta(n^2)$ $O(n^2)$ $\Omega(n)$

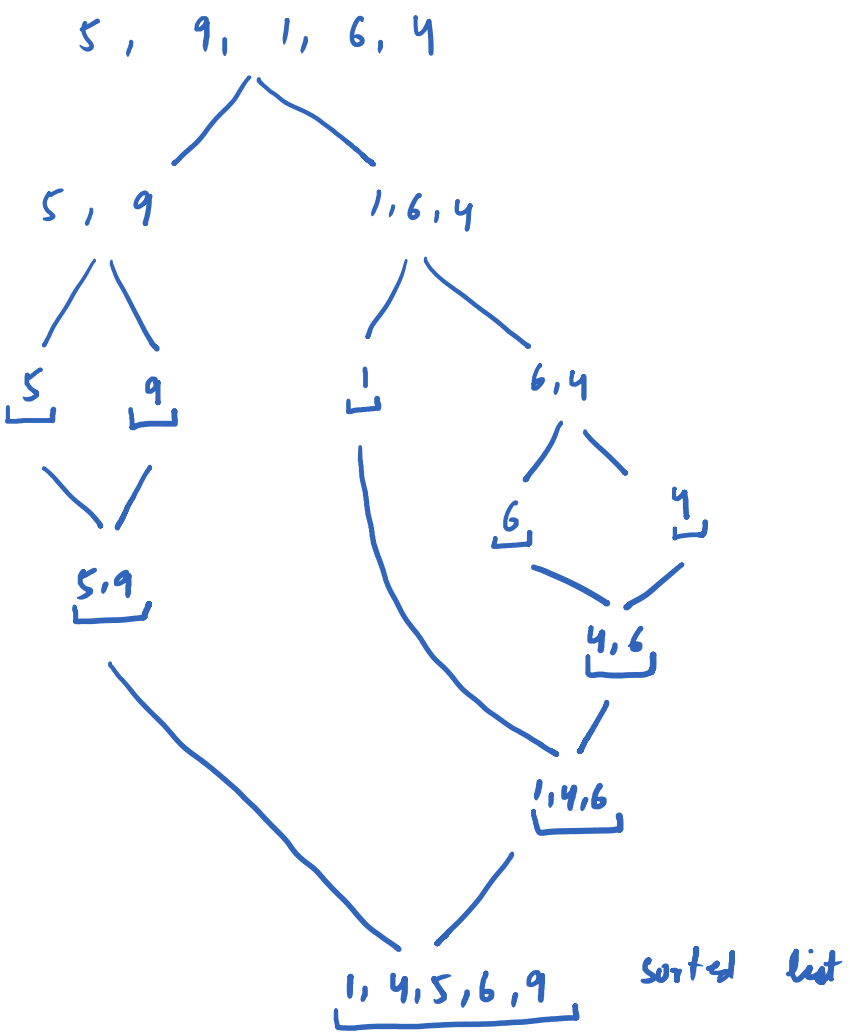
Insertion sort must be prefered when dealing with already or almost sorted data.

In-place sorting algorithm, Stable sorting algorithm.

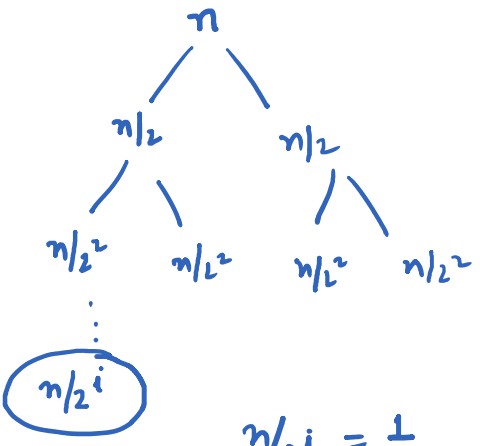
Merge sort :- 2-way merge sort

Merging of sorted lists External Sorting

Divide & Conquer strategy




```
void mergesort (int a[], int lb, int ub)
{
    int k;
    if (lb < ub)
    {
        k = (lb+ub)/2;
        mergesort(a, lb, k);
        mergesort(a, k+1, ub);
        merge(a, lb, k, ub);
    }
}

void merge (int a[], int lb, int k, int ub)
{
```



$n/2^i = 1$

$i = \log_2 n$

```
int i = lb, j = k+1, c[MAX], m = 0;
while (i <= k && j <= ub)
{
    if (a[i] <= a[j])
        c[m++] = a[i++];
    else
        c[m++] = a[j++];
}
while (i <= k)
    c[m++] = a[i++];
while (j <= ub)
    c[m++] = a[j++];
for (i = lb, j = 0; i <= ub; i++, j++)
    a[i] = c[j];
}
```

Not data dependant

Rather structure dependant

$\Omega(n \log_2 n)$

$O(n \log_2 n)$

$\theta(n \log_2 n)$

Stable sort

Not In-place

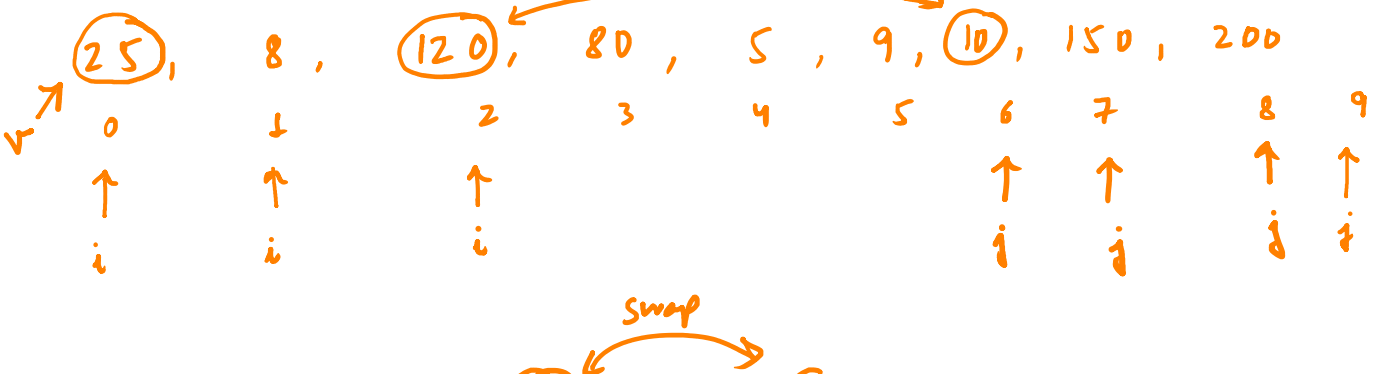
Extra space is Required

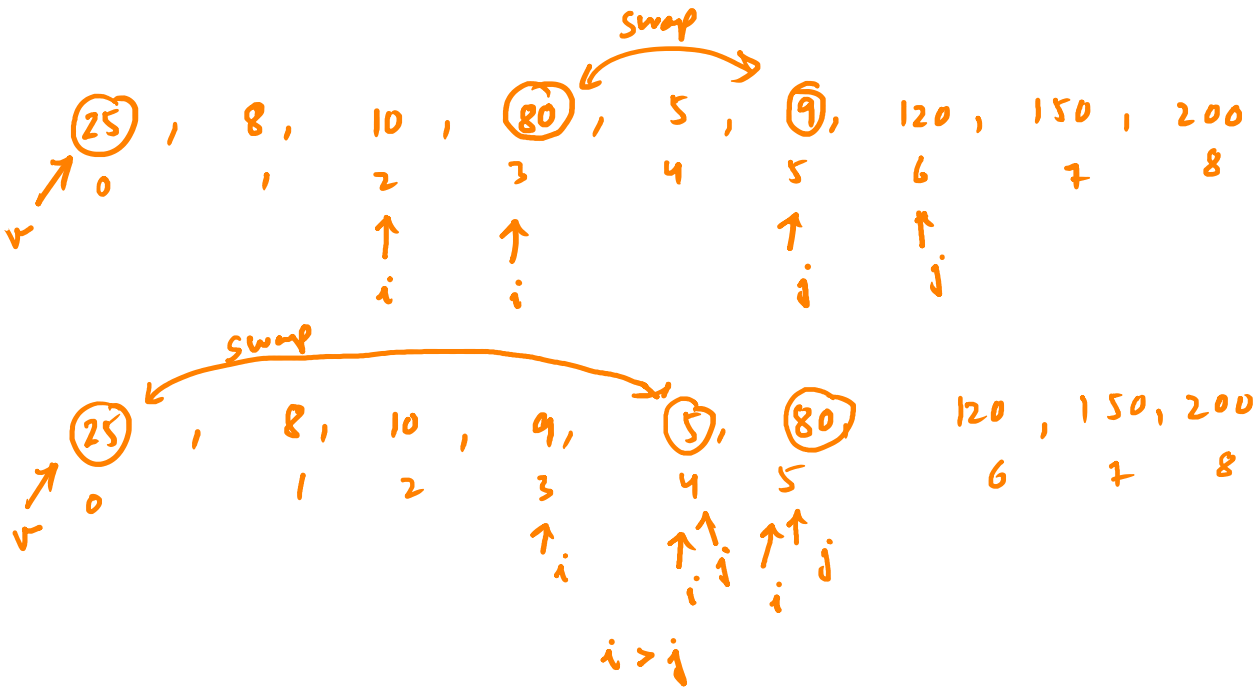
Quicksort :- Fastest internal sorting algorithm

1) Pick the pivot element, v in a[]
2) Arrange the array in such a way that all elements larger than v are placed in RHS of v & all elements smaller than v are placed in LHS of v.
3) Lets say K is the index of v.

a[0 .. K-1], v, a[K+1, ..., n-1]

Solve them recursively swap






```
void quicksort (int a[], int lb, int ub)
 {
    int j;
    if (lb < ub)
    {
       j = partition (a, lb, ub);
       quicksort (a, lb, j-1);
       quicksort (a, j+1, ub);
    }
 }
int partition (int a[], int lb, int ub)
{
  int v = a[lb], i = lb, j = ub+1, temp;
  do
  {
```

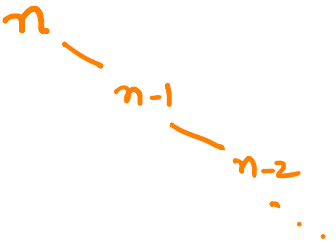
```
do
  i++;
while ( i <= ub && a[i] < v);

do
  j--;
while ( a[j] > v);

if (i<j)
{
  temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
} while (i < j);
a[lb] = a[j];
a[j] = v;
return j;
}
```

Worst cases

(1) Data is sorted in ascending order

(10) 20, 30, 40, 50

right list



$\simeq n^2$

(2) Data is sorted in descending order

(50), 40, 30, 20, 10 →

10 20, 30 40, 50

$\simeq n^2$

$O(n^2)$ $\quad \theta(n \log_2 n)$ $\quad \Omega(n \log_2 n)$

Each pivot element is the median of data

In-place sorting algorithm.

Not stable 10, 9, 9*, 8, 20

Recurrence Functions :-

$T(n)$ ← time taken to solve the algorithm over the n-sized data.

Binary Search : $T(n) = T(n/2) + \theta(1)$

$$T(n) = T(n/2) + c$$
$$T(n/2) = T(n/2^2) + c$$
$$T(n/2^2) = T(n/2^3) + c$$
$$\vdots$$
$$+ \quad T(n/2^{i-1}) = T(n/2^i) + c$$

$$T(n) = T(n/2^i) + ci$$

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

$$i = \log_2 n$$

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

$$\boxed{T(n) \simeq \theta(\log_2 n)}$$

Merge Sort :-

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

$$T(n) = 2T(n/2) + cn$$
$$2T(n/2) = 2^2 T(n/2^2) + cn$$
$$2^2 T(n/2^2) = 2^3 T(n/2^3) + cn$$
$$\vdots$$
$$= 2^i T(n/2^i) + cn$$

$$T(n) = 2^i T(n/2^i) + cni$$

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

$$i = \log_2 n$$

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

$$T(n) = n\,T(1) + cn\log_2 n$$

$$T(n) \simeq \theta(n \log_2 n)$$

Quicksort:- Best case → pivot is median

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

Average case

$$T(n) = T(i) + T(n-i) + \theta(n)$$

↓ if data is sorted

$$T(n) = T(n-1) + \theta(n) \simeq \theta(n^2)$$