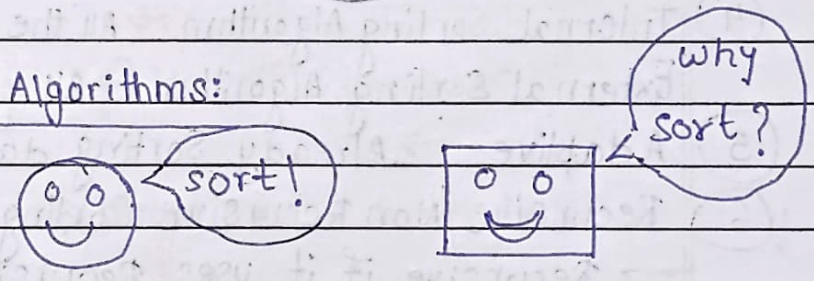
Lec-48 Introduction to Sorting Algorithms:




1 9 8 2 7 .....

Two ways of sorting

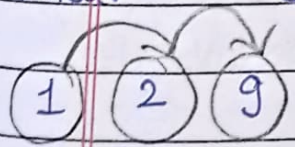
1) Ascending order ⟶ 1, 2, 7, 8, 9

2) Descending order ⟶ 9, 8, 7, 2, 1

Sort by: Newest!
Top Rated!

Swiggy/Zomato ⟶ Restaurants → Sort by Rating → Algo ✓
→ Sort by Price → Algo ✓

Linear Binary




20 71 33 ⟶ yes, 9 is present

✓ ⟶ ↘ n elements

Lec-49. **Criteria for Analysis of Sorting Algorithms:**

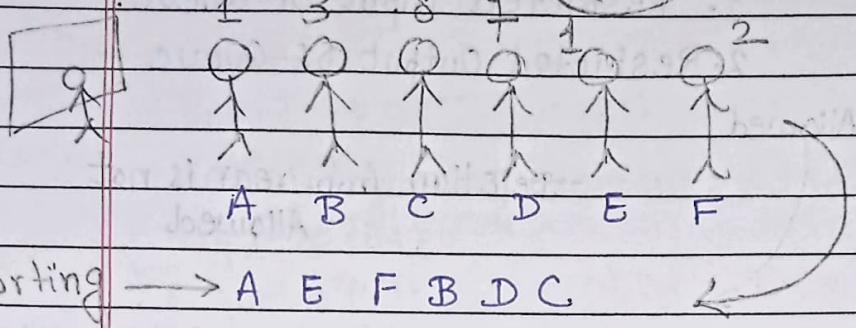
**Analysis Criteria for Sorting Algorithm**

Green House
Date
Page No.

(1) Time Complexity → $O(n^2)$ $O(n \log n)$ ✓ Memory Size C

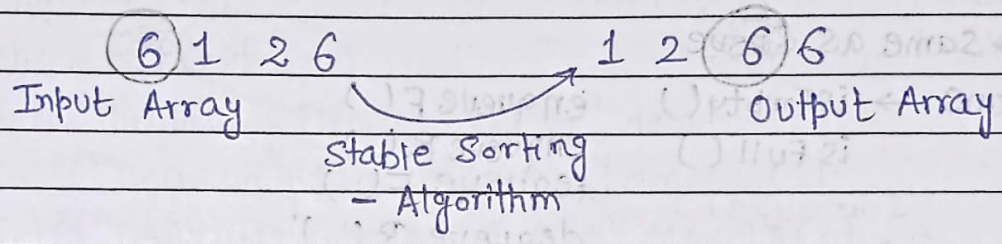
(2) Space Complexity → Inplace Sorting Algorithms → 9, 9K

(3) Stability → (6) 1 2 6 → 1 2 (6) 6




Sorting → A E F B D C

→ Algo 1 is a Stable Sorting Algorithm

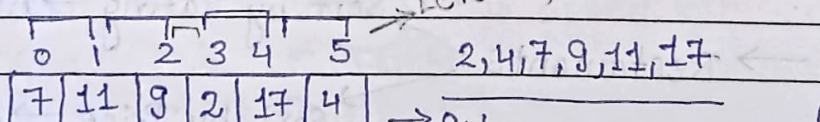



(4) Internal Sorting Algorithm → All the data is loaded into the memory
External Sorting Algorithm → All " " " not loaded " " "

(5) Adaptive → Already Sorted Sorting data takes less time.

(6) Recursive / Non Recursive Sorting Algorithm
→ Recursive if it uses Recursion.

Lec-50 **Bubble Sort Algorithm in Hindi** No. of Passes = n-1




1st Pass
7 9 11 2 17 4 → 1,2
7 9 2 11 17 4 → 2,3 → 5 comparisons, 5 possible swaps
7 9 2 11 17 4 → 3,4
7 9 2 11 4 (17) → 4,5 ⇒ 1st Pass Completed

Lighter Elements ← → Heavier Elements

i → 0 1 2 3 4 5

2nd Pass → 7 9 2 11 4 17 → 0,1
7 2 9 11 4 17 → 1,2
7 2 9 11 4 17 → 2,3
7 2 9 4 11 17 → 3,4

→ 4 Comparisions
→ 4 possible swaps
(n-2) Comp.

Green House
Date
Page No.

3rd pass

2 7 9 4 11 17 → 0,1
2 7 9 4 11 17 → 1,2
2 7 4 9 11 17 → 2,3

→ 3 Comparisions
→ 3 possible swaps

4th Pass

2 7 4 9 11 17 → 0,1
2 4 7 9 11 17 → 1,2

→ 2 Comparision
2 possible swap

5th Pass

2 4 7 9 11 17

1 Comp.
→ 1 Comparison 1 possible swap
→ Sorted Array ☺

Total Number of Comparisons:

Time Complexity:

$1+2+3+4+\ldots\ldots+(n-1) = \frac{n(n+1)}{2} = O(n^2)$

Time complexity of Bubble sort.

0 1 2 3
7 8 7 7 → 0,1
7 7 8 7 → 1,2
7 7 7 8 → order is Same as the order in Input Array
→ Hence, Stable.

1 2 3 4 → 3 passes

Imp
→ By Default it is Not Adaptive, but can be made Adaptive.
→ Not Recursive
→ O(n) → For already Sorted ✓
→ Space Complexity → O(1) No Extra Space required

Sec-51 Bubble Sort Program in C: Also known as Sinking Sort or Exchange Sort

B.C : O(N)
W.C : O(N²)

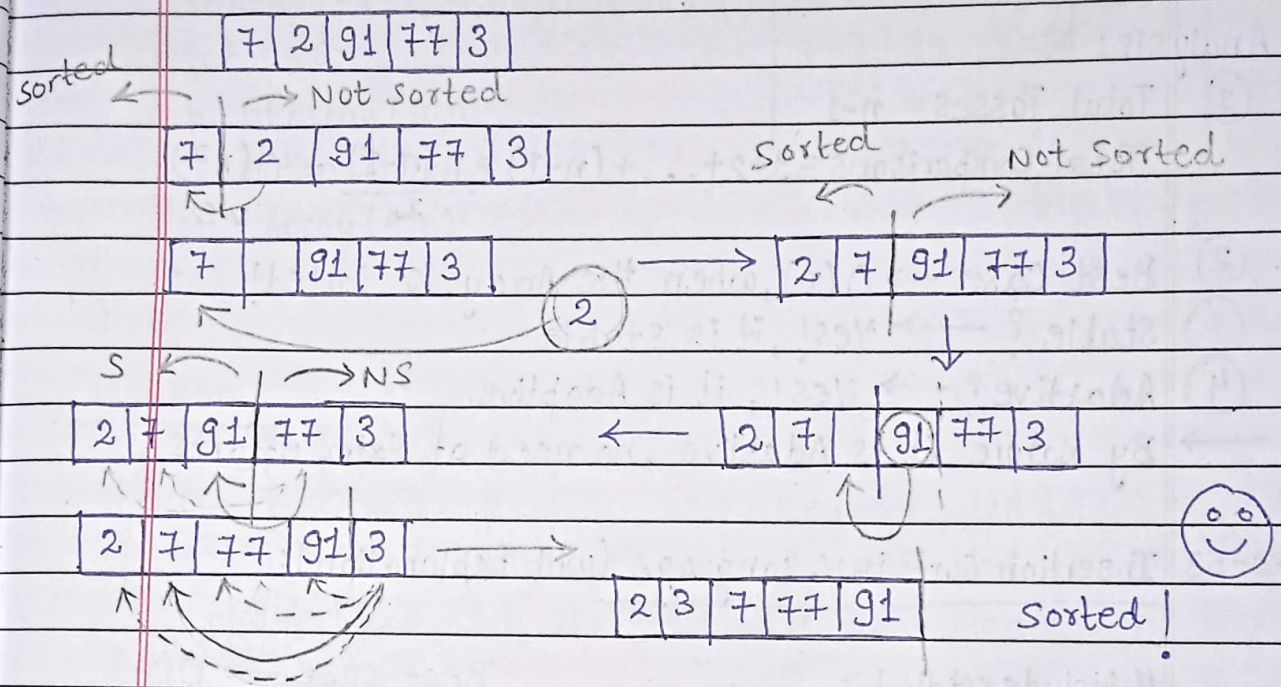
```
#include<stdio.h>

void printArray(int *A, int n){
    for(int i=0; i<n; i++){
        printf("%d", A[i]);
    }
    printf("\n");
}
```

```
void bubbleSort(int *A, int n){
    int temp;
    int isSorted=0;
    for(int i=0; i<n-1; i++)  //for number of passes
    {
        printf("working on pass number %d \n", i+1);
        for(int j=0; j<n-1-i; j++)  //for Comparison in each pass
        {
            if( A[j] > A[j+1] ){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
    }
}

void bubbleSortAdaptive(int *A, int n){
    int temp;
    int isSorted = 0;
    for(int i=0; i<n-1; i++)  //for number of passes
    {
        printf("working on pass number %d \n", i+1);
        isSorted = 1;
        for(int j=0; j<n-1-i; j++)  //for Comparison in each pass
        {
            if(A[j] > A[j+1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                isSorted = 0;
            }
        }
        if(isSorted){
            return 0;
        }
    }
}

int main(){
    int A[] = {1,2,5,6,12,54,625,7,23,9,987};
```

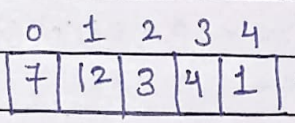
```
int n= 11;
printArray(A,n);          // printing the Array before Sorting
bubbleSortAdaptive(A,n);  // Function to Sort the Array
printArray(A,n);          // printing the Array After Sorting
return 0;
}
```

Green House
Date
Page No.

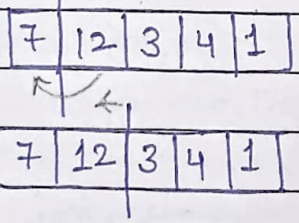
## Lec-52 Insertion Sort Algorithm:



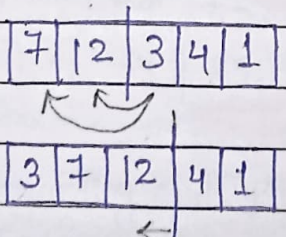

### Insertion Sort:



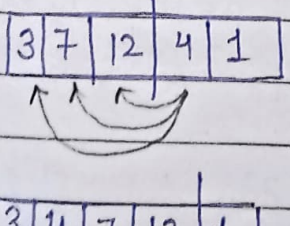

1st Pass:



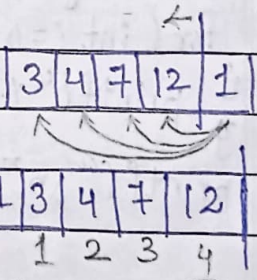

2nd Pass:




3rd Pass:




4th Pass:




Total → n-1 passes

| Time Complexity |
| --- |
| BC → O(n) |
| WC → O($n^2$) |

when Array is Already Sorted

Array is Not Sorted

Total PS/PC = $1+2+3+\ldots+(n-1)$

$= \frac{n(n-1)}{2} = O(n^2)$ → Time complexity (WC)

Green House

Date

Page No.

If Array Already Sorted

| 1 | 3 | 4 | 7 | 12 |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |

Total PC = $1+1+1+\ldots n-1 = O(n)$ (BC)

$= (n-1) = O(n)$

Analysis:

(1) Total Passes = n-1

Total Comparisons = $1+2+\ldots+(n-1) = \frac{n(n-1)}{2} = O(n^2)$ WC

(2) Best Case → O(n), when the Array is Sorted!

(3) Stable ? → Yes!, it is stable

(4) Adaptive ? → Yes!, it is Adaptive.

→ By Nature it is Adaptive, no need of Extra efforts!

## Lec-53 Insertion Sort in C language (with Explanation):

Best case → O(n)

Worst case → $O(n^2)$

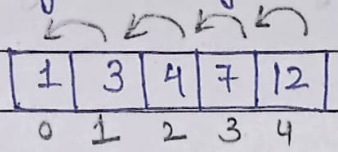
```
#include<stdio.h>

void printArray(int* A, int n){
  for(int i=0; i<n; i++)
  {
    printf("%d", A[i]);
  }
  printf("\n");
}

void insertionSort(int *A, int n){
    int key, j;
  // Loop for passes
  for(int i=1; i<=n-1; i++)
  {
    key = A[i];
    j = i-1;
  // Loop for each pass
```

Total PS/PC = $1+2+3+\ldots+(n-1)$

$= \frac{n(n-1)}{2} = O(n^2)$ → Time complexity (WC)

Green House
Date
Page No.

If Array Already Sorted




Total PC = $1+1+1+\ldots n-1 = O(n)$ (BC)

$= (n-1) = O(n)$

Analysis:

(1) Total Passes = $n-1$

Total Comparisons = $1+2+\ldots+(n-1) = \frac{n(n-1)}{2} = O(n^2)$ WC

(2) Best Case → $O(n)$, when the Array is Sorted!

(3) Stable ? → Yes!, it is stable

(4) Adaptive ? → Yes!, it is Adaptive.

→ By Nature it is Adaptive, no need of Extra efforts!

Lec-53 Insertion Sort in C language (with Explanation):

Best case → $O(n)$

Worst case → $O(n^2)$

```
#include<stdio.h>

void printArray(int* A, int n){
  for(int i=0; i<n; i++)
  {
    printf("%d", A[i]);
  }
  printf("\n");
}

void insertionSort(int *A, int n){
    int key, j;
  // Loop for passes
  for(int i=1; i<=n-1; i++)
  {
    key= A[i];
    j= i-1;
  // Loop for each pass
```

```
while(j>=0 && A[j]>key){
    A[j+1]=A[j];
    j--;
}
A[j+1]=key;
}
int main(){
int A[]={12,54,65,7,23,9};
int n=6;
printArray(A,n);
insertionSort(A,n);
printArray(A,n);
return 0;
}
```

A[j] < key → For descending order sorting

Green House

Date

Page No.

| 12 54 65 7 23 9 |
|---|
| 7 9 12 23 54 65 |

Try Run!

-1 | 0 1 2 3 4 5 i=1, key=54, j=0

12, | (54), 65, 7, 23, 9 → ~~i=2, key=65, j=1~~

12, | (54), 65, 7, 23, 9 → 1st pass done (i=1)!

12, 54, | (65), 7, 23, 9 → i=2, key=65, j=1

12, 54, | (65), 7, 23, 9 → 2nd pass done (i=2)!

12, 54, 65, | (7), 23, 9 → i=3, key=7, j=2

12 54, 65, | (65), 23, 9 → i=3, key=7, j=1

12, 54, (54) | 65, 23, 9 → i=3, key=7, j=0

12, (12) 54, | 65, 23, 9 → i=3, key=7, j=-1

(7) 12, 54, | 65, 23, 9 → i=3, key=7, j=-1 → 3rd pass done!

Fast Forwarding and 4th, 5th pass will give:

07, 12, 54, 65, | (23), 09 → i=4, key=23, j=3

07, 12, (23), 54, | 65, 09 → After the 4th pass

07, 12, 23, 54, 65, | (09) → i=5, key=09, j=4

07, (09), 12, 23, 54, 65 | → After the 5th pass

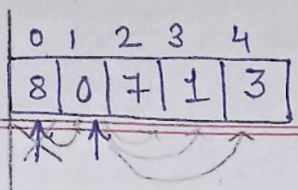
Lec-54 Selection Sort Algorithm:

Total passes → (n-1)

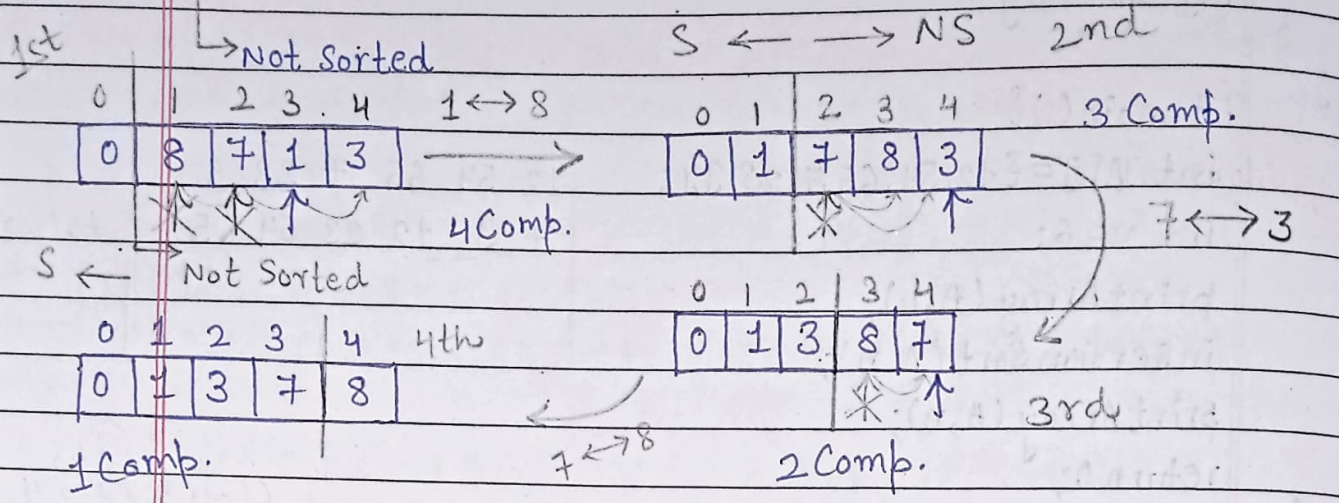
Green House

Date

Page No.




Length of Array = 5



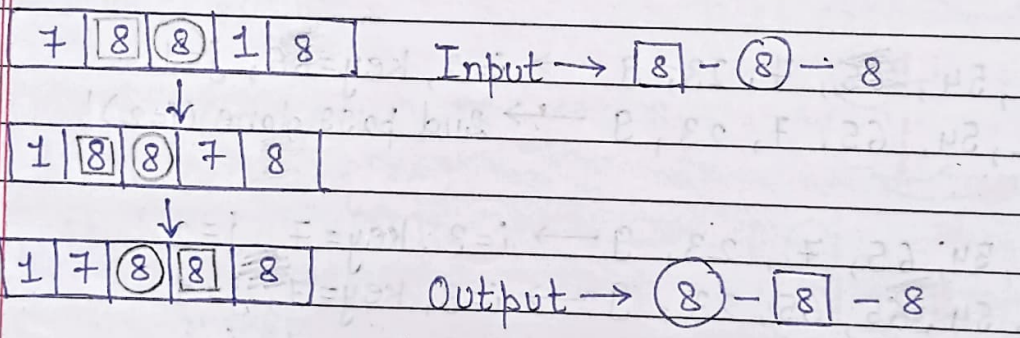

Total Possible Comparisons: $1+2+3+\ldots+(n-1)$

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

Max. Swaps = (n-1)

$T = T_1(n-1) + T_2\left(\frac{n(n-1)}{2}\right) \rightarrow O(n^2)$

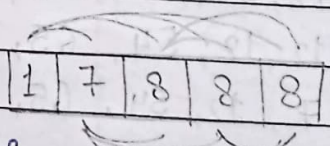
Is Selection Sort Adaptive? And Stable?




→ Selection Sort is not Stable!

→ Selection Sort is not Adaptive!

→ Sorting in Minimum Number of Swaps.




Lec-55 Selection Sort Program in C:

```
#include<stdio.h>

void printArray(int *A, int n){
```

```
for(int i=0; i<n; i++)
{
printf("%d", A[i]);
}
printf("\n");
}
```

Green House
Date
Page No.

```
void SelectionSort (int * A, int n){
 int indexofMin, temp;
printf("Running Selection Sort.... \n");
for(int i=0; i<n-1; i++)
{
 indexofMin = i;
for(int j= i+1; j<n; j++)
{
 if(A[j] < A[indexofMin]){
  indexofMin=j;
 }
}
// swap A[i] and A[indexofMin]
 temp=A[i];
A[i]= A[indexofMin];
A[indexofMin] = temp;
}
}
```

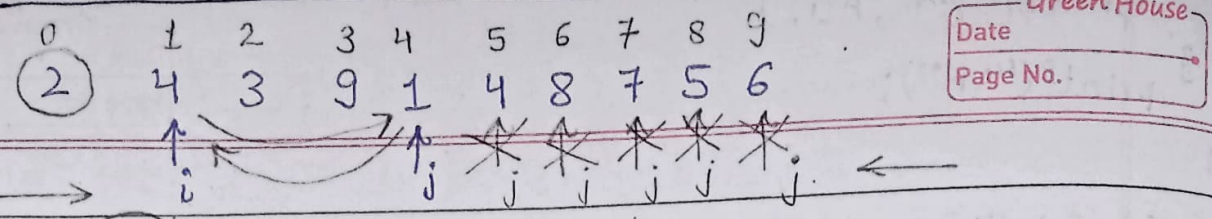
| |
|---|
| 3 5 2 13 12 |
| Running Selection Sort.... |
| 2 3 5 12 13 |

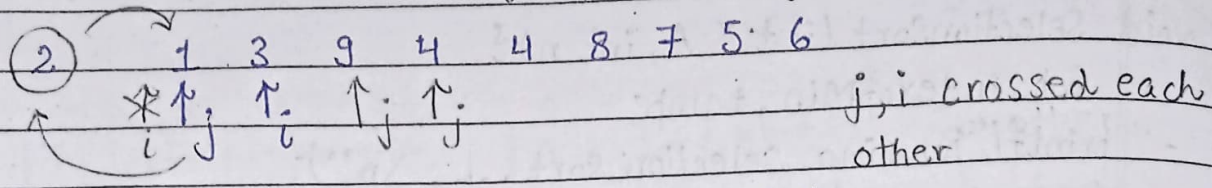
```
int main(){
 int A[]={3,5,2,13,12};
 int n=5;
 printArray(A,n);
 SelectionSort(A,n);
 printArray(A,n);
 return 0;
}
```

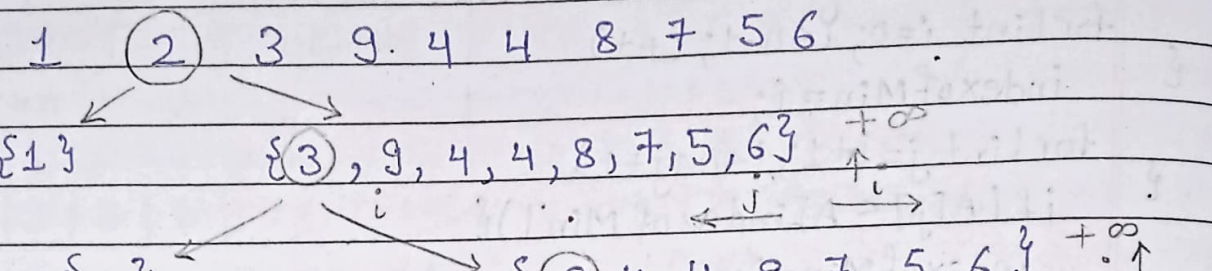
Best case $\rightarrow O(n^2)$
Worst case $\rightarrow O(n^2)$
Average case $\rightarrow O(n^2)$

# Lec-56. Quick Sort Algorithm in Hindi (with Code in C)

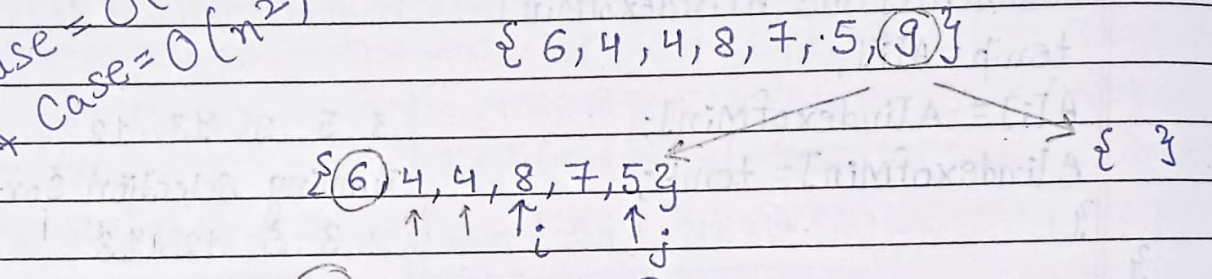
Green House
Date
Page No.




Best Case = O(n log n)
Worst Case = O(n²)




Sorted Array,

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |

**Steps:**

1. i = low
2. j = High
3. pivot = low
4. i++ untill element > pivot is found.

5. j-- until element ≤ pivot is found
6. Swap A[i] & A[j] and repeat 4 & 5 steps until (j≤i)
7. Swap pivot & A[j].

This Algorithm follows → Divide and Conquere strategy

## Quick Sort Program in C:

```
#include<stdio.h>
void printArray(int *A, int n)
{
  for(int i=0; i<n; i++){
    printf("%d ", A[i]);
  }
  printf("\n");
}
int partition(int A[], int low, int high)
{
  int pivot=A[low];
  int i=low+1;
  int j=high;
  int temp;
  do
  {
    while(A[i]<=pivot)
    {
      i++;
    }
    while(A[j]>pivot)
    {
      j--;
    }
    if(i<j){
      temp=A[i];
      A[i]=A[j];
      A[j]=temp;
    }
  }while(i<j);
  //Swap A[low] and A[j]
  temp=A[low];
  A[low]=A[j];
  A[j]=temp;
  return j;
}
void quickSort(int A[], int low, int high)
{
  int partitionIndex;
  if(low<high)
  {
    partitionIndex=partition(A,low,high);
    quickSort(A,low,partitionIndex-1);
    quickSort(A,partitionIndex+1,high);
  }
}
int main(){
  int A[]={9,4,4,8,7,5,6};
  int n=7;
  printArray(A,n);
  quickSort(A,0,n-1);
  printArray(A,n);
  return 0;
}
```

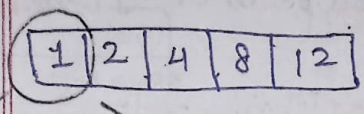
| |
|---|
| 9 4 4 8 7 5 6 |
| 4 4 5 6 7 8 9 |

Index of pivot after partition

Sort right subarray

Sort left Subarray

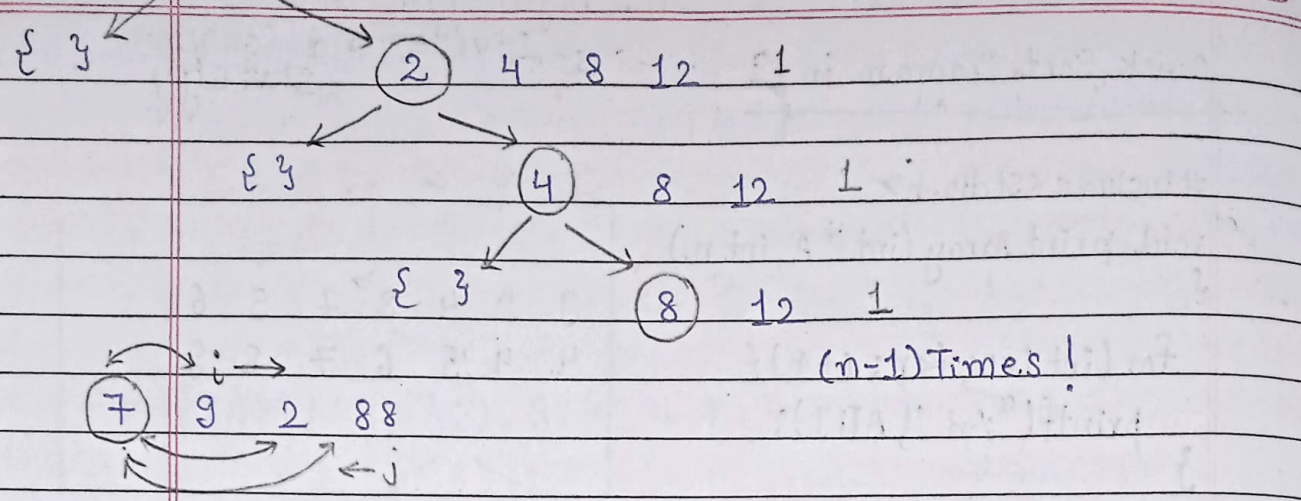
Lec-57 Analysis of QuickSort Sorting Algorithm:




1

Worst Case! → $O(n^2)$

→ Already Sorted Elements.





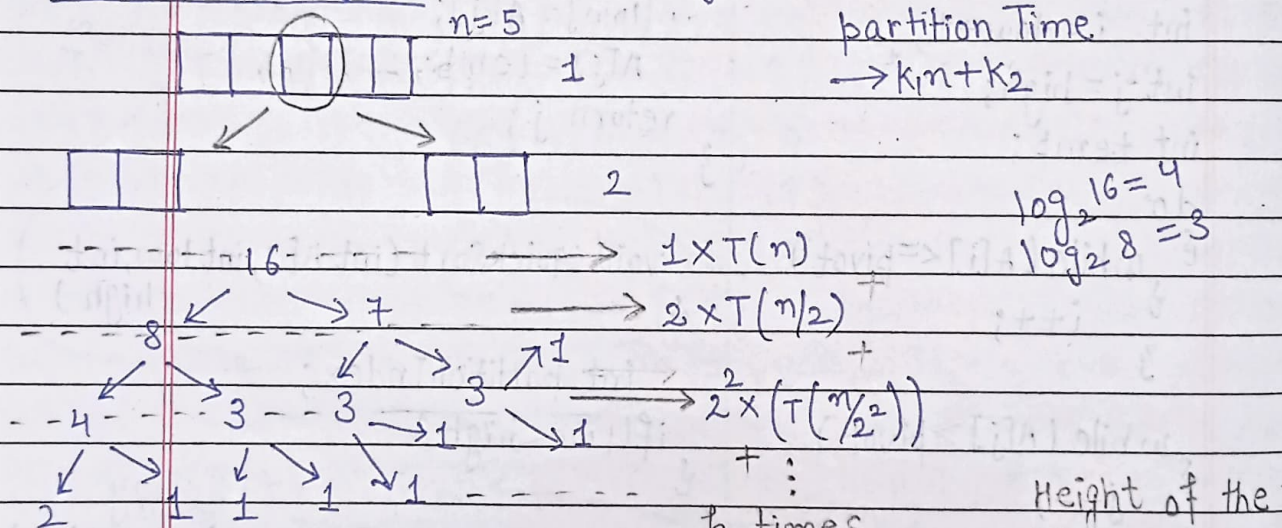


No. of Comparisons → Partition = Some linear function of n

Algo → $K_1 n + K_2$

$T = (n-1) \times K_1 n \times K_2 \rightarrow O(n^2)$

Best Case Analysis! → $O(n \log n)$




$\log_2 16 = 4$

$\log_2 8 = 3$

$1 \times T(n) + 2 \times T(n/2) + 2^2 \times T(n/2^2) + \vdots$

Total Time $= (K_1 n + K_2) + 2\left(K_1 \frac{n}{2} + K_2\right) + 2^2\left(K_1 \frac{n}{2^2} + K_2\right)$

$= K_1 n + 2\left(K_1 \frac{n}{2}\right) + 2^2\left(K_1 \frac{n}{2^2}\right) + K + \dots + h$

$= h \times K_1 n + k \simeq O(n \log n)$

Average Case Time Complexity → $O(n \log n)$

Is QuickSort stable?

(2) 8 9 12 2 → QuickSort is not Stable!
↑i ↑j → QuickSort is an inplace Algorithm

Green House
Date
Page No.

(2) 2 9 12 8
↑j ↑i ↑j ↑j

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 8 | 9 | 12 | 2 |

→

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 2 | 2 | 8 | 9 | 12 |

Not stable!

2 (2) 9 12 8

{2}

(9) 12 8 → (9) 8 12 → 8 (9) 12
↑i ↑j ↑j ↑i

Lec-58 **MergeSort Algorithm in Data Structure:**

i=j=k=0

Copy

| 7 | 9 | 18 | 19 | 22 |
|---|---|---|---|---|

A Sorted Array 1 size → m

| 1 | 6 | 9 | 11 |
|---|---|---|---|

B Sorted Array 2 size → n

Copied

| 1 | 6 | 7 | 9 | 9 | 11 | 18 | 19 | 22 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

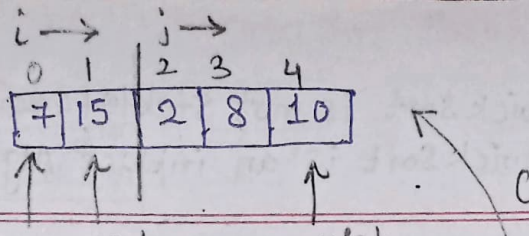
C Sorted Array 3 size → (m+n)

```
void Merge (A[],B[], C[], m, n) {
  int i,j, k;
  i=j=k=0;
  while(i<m && j<n) {
    if( A[i] < B[j] ){
      C[k] = A[i]
      i++; k++;
    }
  else {
      c[k]=B[j];
      j++; k++;
  }
  }
  while(i<m){
    C[k] = A[i];
    k++; i++;
  }
```

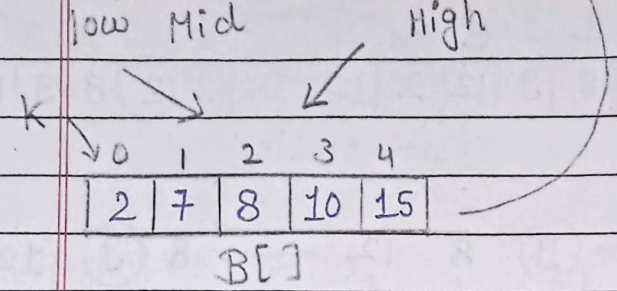
Copy all remaining elements from A to C

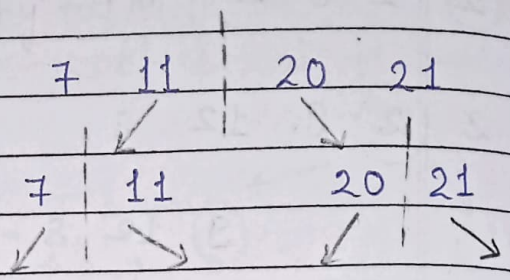
```
  while(j<n){
    C[k]=B[j]
    k++;
    j++;
  }
  }
```

from B to C




Copy to A[]




```
void Merge (A[], mid, low, high){
    int i, j, k;
    int B[high+1];
    i=low;
    j=mid+1;
    k=low;
    while (i<=mid && j<=high){
    if (A[i]<A[j]){
        B[k]=A[i];
        i++; k++;
    }
else {
    B[k]=A[j];
    j++; k++;
    }
}
while (i<=mid){
    B[k]=A[i];
    k++; i++;
}
while (j<high){
    B[k]=A[j];
    k++; j++;
}
}

for(int i=low; i<=high; i++){
    A[i]=B[i];
}
}
```

## Recursive Merge Sort

```
void MergeSort(int A[], int
{                   low, int high)
  if (low<high)
{
  mid=(low+high)/2;
MergeSort(A, low, mid);
MergeSort(A, mid+1, high);
 Merge(A, low, mid, high);
}
}
```

Note

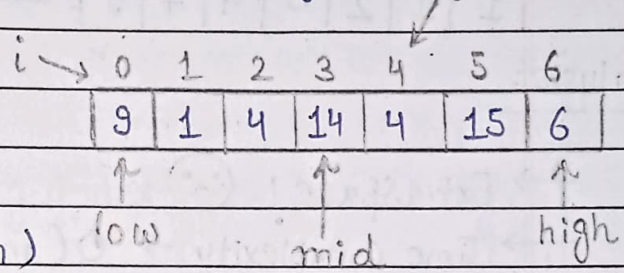
Ideal Size for Array B[]
→ high-low+1;

Rec-59 Merge Sort Source Code in C

```
#include<stdio.h>
void printArray(int *A, int n)
{
  for(int i=0; i<n; i++)
  {
    printf("%d ", A[i]);
  }
  printf("\n");
}
void merge(int A[], int mid, int low, int high)
{
  int i, j, k, B[100];
  i=low;
  j=mid+1;
  k=low;
→ while(i<=mid && j<=high)
  {
    if(A[i]<A[j])
    {
      B[k]=A[i];
      i++; k++;
    }
    else {
      B[k]=A[j];
      j++; k++;
    }
  }
→ while(i<=mid){
    B[k]=A[i];
    k++; i++;
  }
→ while(j<=high){
    B[k]=A[j];
    k++; j++;
  }
→ for(int i=low; i<=high; i++){
    A[i]=B[i];
  }}
void mergeSort(int A[], int low,
               int high){
  int mid;
  if(low<high){
    mid=(low+high)/2;
→   mergeSort(A, low, mid);
→   mergeSort(A, mid+1, high);
→   merge(A, mid, low, high);
  }
}
int main(){
  int A[]={9,1,4,14,4,15,6};
  int n=7;
  printArray(A,n);
→ mergeSort(A,0,6);
  printArray(A,n);
  return 0;
}
```

Output:

| | | | | | | |
|---|---|---|---|---|---|---|
| 9 | 1 | 4 | 14 | 4 | 15 | 6 |
| 1 | 4 | 4 | 6 | 9 | 14 | 15 |



Lec-60 **Count Sort Algorithm:**

Green House
Date
Page No.

Count Sort ⟶ One of the Fastest

Given Array

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 3 | 1 | 9 | 7 | 1 | 2 | 4 |

size of Array = n = 7 ⟶ Array

i

Max = 9 ⟶ m

Count (10)

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |

Size ⟶ Max+1

j

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 1 | 2 | 3 | 4 | 7 | 9 |

⟶ Sorted Array!

**Analysis:**

⟶ Extra Space! :(

⟶ Time Complexity → $O(m+n) \simeq O(n)$ ⟶ Linear Time!

⟶ But at the Cost of Extra Space!

**C Code for Count Sort Algorithm:**

```
#include <stdio.h>
#include <limits.h>
#include <stdlib.h>

void printArray(int *A, int n){
for(int i=0; i<n; i++){
  printf("%d ", A[i]);
}
printf("\n");
}

int maximum(int A[], int n){
  int max=INT_MIN;
for(int i=0; i<n; i++){
  if(max<A[i]){
    max=A[i];
  }
```

```
}
return max;
}
void CountSort(int* A, int n){
int i, j;

// Find the maximum element in A
int max = maximum(A,n);
// Create the Count Array
int* count = (int*)malloc((max+1)*sizeof(int));
// Initialize the array element to 0
for(i=0; i<max+1; i++){
   count[i] = 0;
}
// Increment the Corresponding Index in the Count Array
for(i=0; i<n; i++){
count[A[i]] = count[A[i]]+1;
}
i=0; // counter for Count Array
j=0; // Counter for given Array A
while(i<=max){
   if(count[i]>0){
    A[j] = i;
   count[i] = count[i]-1;
   j++;
}
else{
    i++;
}
}
}
int main(){
   int A[] = {9,1,4,14,4};
   int n=5;
printArray(A,n);
CountSort(A,n);
printArray(A,n);
return 0;  }
```

Output:

| | | | | |
|---|---|---|---|---|
| 9 | 1 | 4 | 14 | 4 |
| 1 | 4 | 4 | 9 | 14 |