ITT201 Data Structures

Module 1 : Introduction to Data Structures

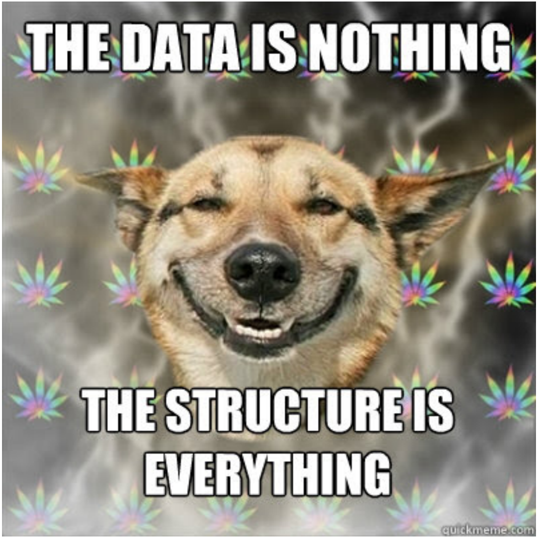
Anoop S K M
(skmanoop@gmail.com)
Department of Information Technology
Govt. Engg. College, Sreekrishnapuram, Palakkad

Acknowledgements

- All the pictures are taken from the Internet using Google search.
- Wikipedia also referred.

Lecture 01

Welcome to Data Structures!



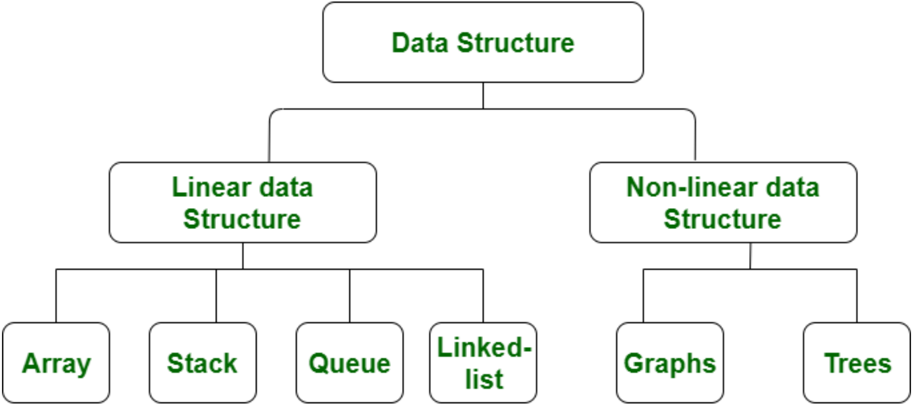
Data Structure

Data Structure

A data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data, i.e., it is an algebraic structure about data.

- is a particular way of organizing data in computer
- data can be used efficiently and effectively

Types of Data Structure



What all are there in store?

Syllabus

Module 1: Introduction to data structures (9 Hours)	
Data Structures-Introduction and Overview- Arrays, Algorithm/Program Development, , Searching and Sorting.	
Module 2: Linked lists (10 Hours)	
Linked lists, singly linked list, Doubly linked list, Circular linked list, Applications of linked list, Dynamic Memory management.	
Module 3 : Stacks and Queues (9 Hours)	
Stack, Applications of stacks, Queues, Types of queues	
Module 4 : Trees and graphs (10 Hours)	
Trees, Binary Tree Traversals, Binary tree Applications, Graph, and Graph Applications.	
Module 5 : Hash Table (7 Hours)	
Hash Tables, Different Hash Functions, Collision Resolution Techniques, closed hashing and Open Hashing (Separate Chaining).	

Text Books

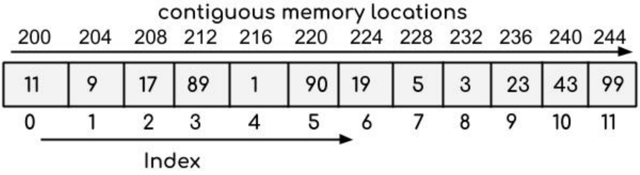
T1. Samanta D., Classic Data Structures, Prentice Hall India, 2/e, 2009.

T2. Ellis horowitz, Sartaj Sahni, Fundamentals of Data structures, Galgotia Booksource

1-D Integer Array- Properties with Example

Properties :

- All elements of same type
- The number of elements is fixed
- stored in contiguous memory locations



- size of the array is 12
- 200 is the starting address
- int uses 4 bytes

Declaration and Initialization of Arrays

- int a[5];
  - integer array of size 5 with garbage values
- int a[5] = {1,2,3,4,5}; OR int a[ ] = {1,2,3,4,5};
  - integer array of size 5 with initial values from 1 to 5
- int a[5] = {1,2};
  - integer array of size 5 with initial values 1,2,0,0,0
- int a[5] = {1,2,3,4,5,6,7};
  - same as int a[5] = {1,2,3,4,5};
  - warning: excess elements in array initializer
  - warnings and errors are different
- int a[5];
  for(int i=0; i<5; i++) {a[i]= i+1;}
  - a[0] =1, a[1] = 2, a[2] = 3, a[3] = 4 & a[4] = 5

Searching

- Trying to find something by looking or otherwise seeking carefully and thoroughly.

Searching in Array- Linear Search

- Assume an array A[ ] storing n different numbers
- Task: Accept a value , check if the value is present in the array

```
int val;
scanf("%d",&val);
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    }
}
```

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only
- Worst Case : *val* present at the last index : Compare *val* with all *n* elements

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only
- Worst Case : *val* present at the last index : Compare *val* with all *n* elements Any other Worst Cases?

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only
- Worst Case : *val* present at the last index : Compare *val* with all n elements Any other Worst Cases?
- Worst Case :*val* not present : Compare *val* with all n elements

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only
- Worst Case : *val* present at the last index : Compare *val* with all n elements Any other Worst Cases?
- Worst Case :*val* not present : Compare *val* with all n elements

What about Average Case number of comparisons ?

Search time - Number of comparisons

```
for (int i = 0; i < n; i++) {
    if (A[i] == val) {
        printf("Found %d at index %d\n", val, i);
        break;
    } }
```

Comparison Times :

- Best Case : *val* at index 0 : Compare *val* with 0^{th} element only
- Worst Case : *val* present at the last index : Compare *val* with all n elements Any other Worst Cases?
- Worst Case :*val* not present : Compare *val* with all n elements

What about Average Case number of comparisons ?

$$\text{Avg. No. of Comparisons} = \frac{1+2+3+\ldots+n}{n}$$

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

Lab Experiment #1- Linear Search

- Accept the size of the array
- Accept the elements
- Ask for the item to be searched
- Print either Found or Not Found

Lecture 02

We Saw & Will See

Till Now We Saw...

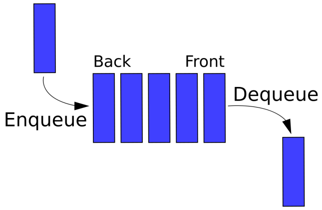
- Module 1 : Introduction to Data Structures
  - Linear/Non-linear DS
  - Searching - Linear Search

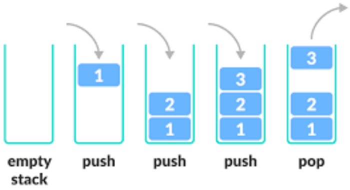
Today We Will See...

- Abstract Data Type & Concrete Data Type
- Searching - Binary Search

Abstract Data Types (ADT)

- a mathematical model of a data structure
- specifies the type of data stored, the operations supported on them, and the types of parameters of the operations
- specifies what each operation does, but not how it does it
- can be implemented using one of many data structures
- examples : Stack, Queue





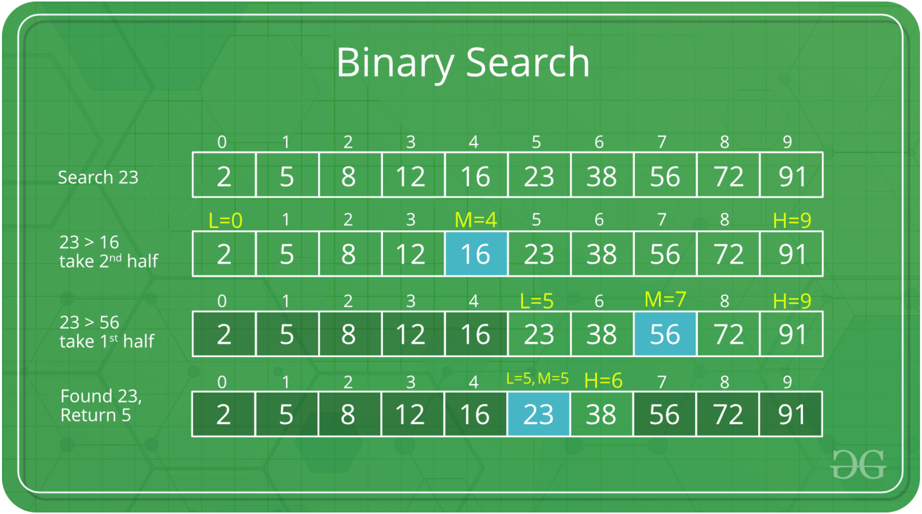
Concrete Data Types(CDT)

- Boolean, Integer, Floating Point, User defined Structures
- Arrays, linked lists, trees, graphs
- It is a specialized solution-oriented data type that represents a well-defined single solution domain concept.
- A concrete data type is a data type whose representation is known and relied upon by the programmers who use the data type.

Searching in Dictionary

- Search in dictionary for "floccinaucinihilipilification" !
- Do we usually search from page 1 ?
- We search at somewhat middle of the book.
- So need to search only in the first part of the book.
- This is the idea behind - Binary Search!

Binary Search Example



Binary Search

- Compare element to be searched with the element at the middle. If Matches, then report Found and Exit.
- else decide which part of the array is relevant and repeat the above step until the size of the part is more than one.
- report Not Found and Exit

```
\\ Assume int A[n] contains data
\\ & key the value to be searched
low = 0; high = n-1;
while (low <= high) {
   mid = (low + high) / 2;
   if (A[mid] == key) {
      printf("Found at %d", mid);
      return; }
   if (key > A[mid]) {
      low = mid+1; }
   else {
      high = mid-1; }
}
printf("Not Found");
```

Linear Search Vs Binary Search

	Linear Search	**Binary Search**
Input	Data in any order	Data should be Sorted
Searching	From Start or End	From Middle
Single Search	Reduces the Search space by one	Reduces the Search space by half
Time Complexity	$O(n)$	$O(\log n)$

Lecture 03

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Linear/Non-linear DS
  - Searching - Linear Search & Binary Search
  - Abstract Data Type & Concrete Data Type

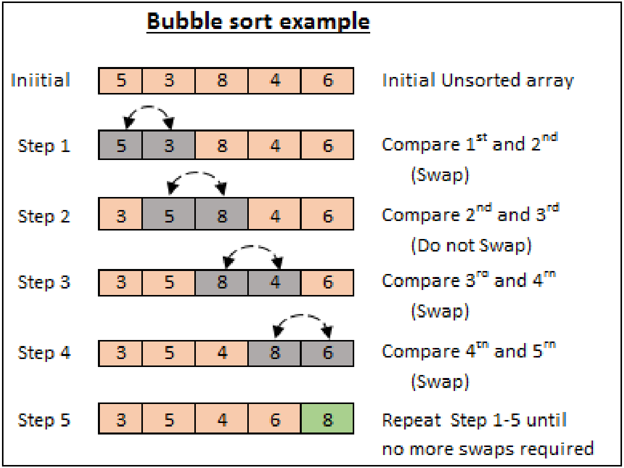
Today We Will See...

- Sorting - Bubble Sort
- Algorithm to Pseudocode and then to Program

Sorting

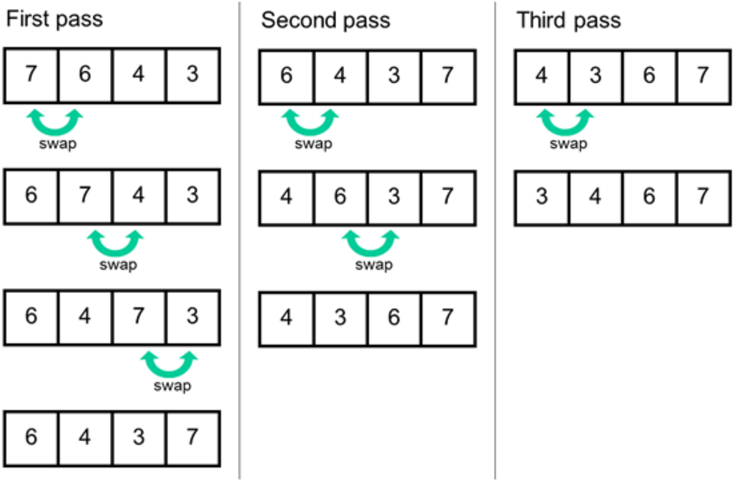
- Sorting - arranging things in order.
- Why Sorting is important ?
- What if the words in a Dictionary were UnSorted ?
- Sorting makes Searching easier.
- Linear Search Time(O(n)) Vs Binary Search Time(O($\log n$)).
- Sorting Numbers - ascending(non decreasing) or descending(non increasing)

Bubble Sort-One pass Example



- First pass ensures the largest gets moved to extreme right.
- i.e. largest number moved to index n-1
- After Second pass, second largest at index n-2
- In general, after i passes largest i values at correct positions
- Total passes needed is ~~n~~?

Bubble Sort-All passes Example

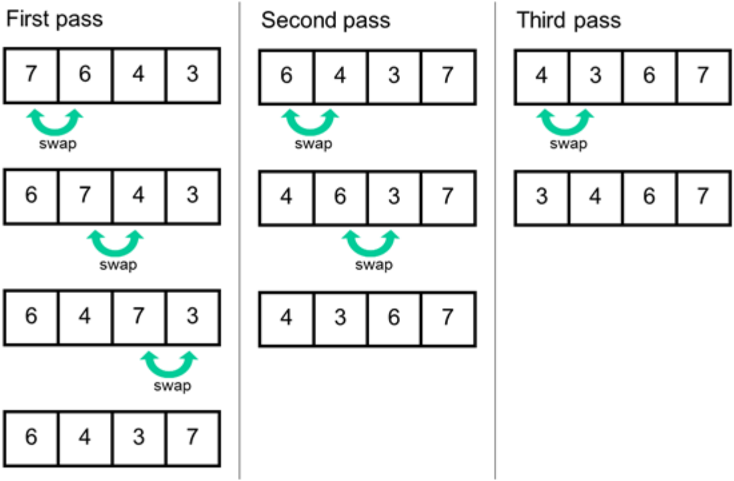


Algorithm, Pseudocode, Program

- Algorithms are generally written in plain English
- Pseudo-codeis written in a format that is similar to the structure of a high-level programming language
- Program write a code in a particular programming language.

Let us see an example : Bubble Sort

Bubble Sort-All passes Example



Bubble Sort : Algorithm & Pseudo-code

Algorithm:

- Step 1 : Accept the contents of the array of size n
- Step 2: Compare the first two elements
  - if first element > second element swap them
- Step 3: continue the Step 2 comparison process, with the second and third element, then third and forth element and so on till the last two elements
- Step 4: Restart from Step 2 the whole process starting with the first two elements until the array is sorted

Pseudo code:

```
for i=0 to n-1
 accept A[i]
for i=1 to n-1
 for j=0 to n-2
  if A[j] > A[j+1] then
   Swap( A[j] and A[j+1])
```

Pseudo code & Program

Pseudo code:

```
for i=0 to n-1
  accept A[i]
for i=1 to n-1
 for j=0 to n-2
   if A[j] > A[j+1] then
     Swap( A[j] and A[j+1])
```

```
int main()
{
 int n,temp;
 scanf("%d",&n);
 int A[n];
 for(int i=0; i<n; i++)
   scanf("%d",&A[i]);
 for(int i=1;i<n;i++){
   for(int j=0;j<n-1;j++){
     if(A[j] > A[j+1]){
       temp = A[j];
       A[j] = A[j+1];
       A[j+1] = temp;
     }
   }
 }
}
```

Let's Try Bubble Sort in Lab!

Lab#03 in replit

- Accept size of the array (n)
- Accept the array elements
- Do Bubble Sort
- Print the elements in the sorted array

Lecture 04

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Linear/Non-linear DS
  - Searching - Linear Search & Binary Search
  - Abstract Data Type & Concrete Data Type
  - Algorithm to Pseudocode and then to Program
  - Sorting: Bubble Sort

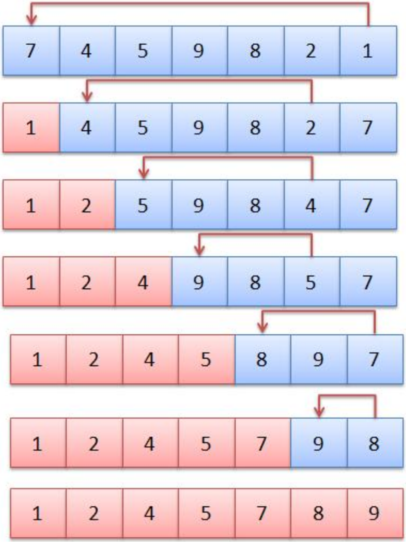
Today We Will See...

- Sorting - Selection Sort

Selection Sort on array of size n

- Select the minimum and place at index 0
- Select the second minimum and place at index 1
- Continue the process $n-1$ times

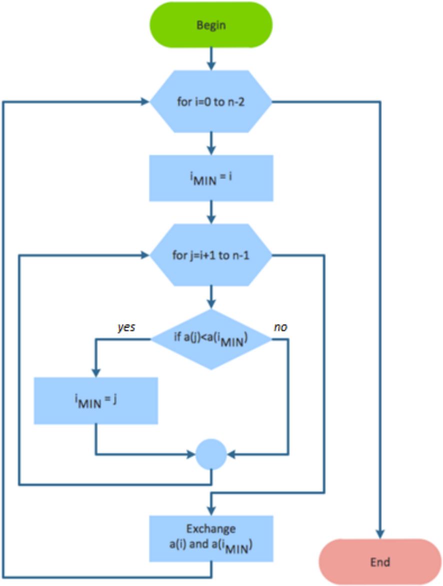
Selection Sort Example



Select the minimum from an array

- Algorithm for finding the minimum from an array .
- Try to find the index of the minimum valued element in an array a[n]

Selection Sort Flow Chart



Selection Sort : Algorithm & Pseudo-code

Algorithm:

- Step 1 : Accept the contents of the array of size n
- Step 2: $i = i_{min} = 0$ i.e. the current minimum at index 0
  - starting from index $i_{min} + 1$ upto $n - 1$, find the index of the smallest element and update i_{min}
- Step 3: Swap elements at index i and index i_{min}
- Step 4: Restart from Step 2 the whole process with i incremented by 1, till $i \leq n - 2$

Pseudo code:

`Pseudo code in next slide`

Selection Sort Pseudo Code

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

Lecture 05

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Linear/Non-linear DS
  - Searching - Linear Search & Binary Search
  - Abstract Data Type & Concrete Data Type
  - Algorithm to Pseudocode and then to Program
  - Sorting: Bubble Sort, Selection Sort

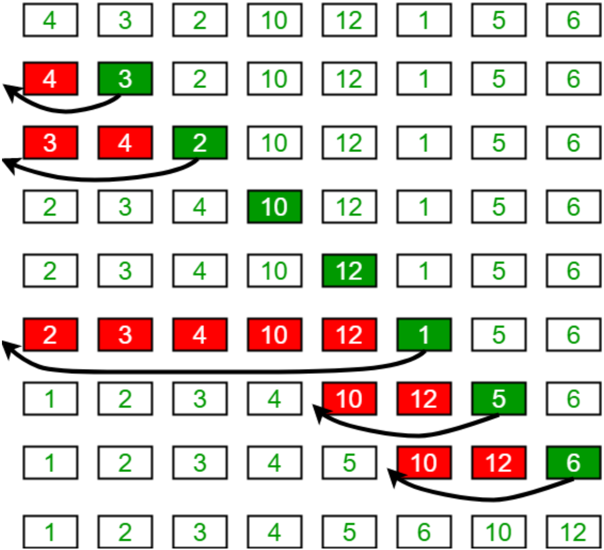
Today We Will See...

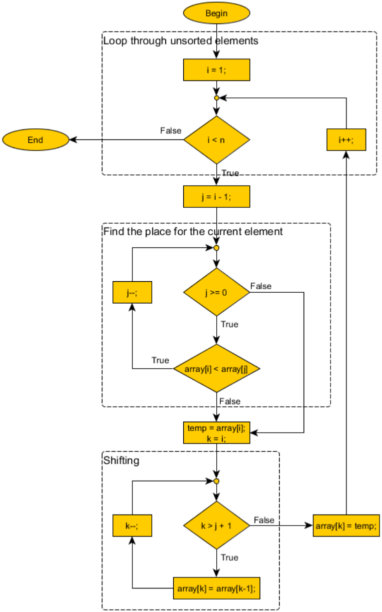
- Sorting - Insertion Sort

Card Picking and Arranging

- A player picks one by one card and inserts at correct position comparing the cards already in hand.
- comparison is done usually from the beginning/end

Insertion Sort Execution Example




Begin
Loop through unsorted elements
i = 1;
False
End
i < n
i++;
True
j = i - 1;
Find the place for the current element
j--;
j >= 0
False
True
True
array[i] < array[j]
False
temp = array[i];
k = i;
Shifting
k--;
k > j + 1
False
array[k] = temp;
True
array[k] = array[k-1];

Pseudo code of insertion sort

```
INSERTION-SORT(A)
for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j − 1].
    i = j − 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

Lecture 06

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
    - Linear/Non-linear DS
    - Searching - Linear Search & Binary Search
    - Abstract Data Type & Concrete Data Type
    - Algorithm to Pseudocode and then to Program
    - Sorting: Bubble Sort, Selection Sort, Insertion Sort Example

Today We Will See...

- Sorting - Insertion Sort Pseudo code and Program

Insertion Sort Pseudo Code

```
for(int i=1 ;i < n; i++){
        int key = a[i];
        int j = i-1;
        while( j>=0 && a[j] > key){
                a[j+1] = a[j];
                j--;
        }
        a[j+1] = key;
}
```

Insertion Sort Pseudo Code I

```
int main()
{
        int n;
        scanf("%d",&n);
        int a[n];
        for(int i = 0; i < n; i++)
                scanf("%d",&a[i]);
        for(int i=1 ;i < n; i++){
                int key = a[i];
                int j = i-1;
                while( j>=0 && a[j] > key){
                        a[j+1] = a[j];
                        j--;
                }
                a[j+1] = key;
        }
```

Insertion Sort Pseudo Code II

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

Lecture 07

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Linear/Non-linear DS
  - Searching - Linear Search & Binary Search
  - Abstract Data Type & Concrete Data Type
  - Algorithm to Pseudocode and then to Program
  - Sorting: Bubble Sort, Selection Sort, Insertion Sort

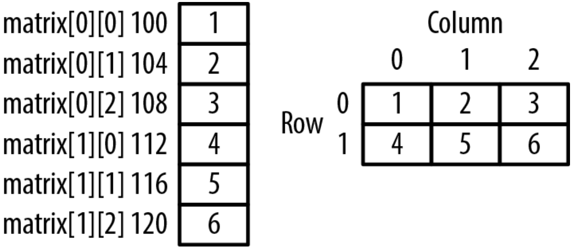
Today We Will See...

- 2 D Array
- Sparse Matrix

2-D Array indices

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

2-D Array storage in Memory



2-D Array declarations

- int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
- int arr[2][3] = {1, 2, 3, 4, 5, 6 };
- int arr[ ][3] = {1, 2, 3, 4, 5, 6 };
    - size of first dimension can be empty. No others can be empty.
- int arr[ ][ ] = {1, 2, 3, 4, 5, 6 };
- int arr[2][ ] ={1, 2, 3, 4, 5, 6 };

Sparse Matrix

```
[[ 0.  0. 14.  0.  0.  0. 15.  0.  0.  0.]
 [16.  0. 14.  0.  0. 14.  0. 13.  0.  0.]
 [15.  0.  0. 13.  0. 16.  0.  0. 13. 16.]
 [ 0.  0. 16.  0.  0.  0. 14.  0.  0.  0.]
 [ 0. 11.  0.  0.  0.  0.  0. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0. 15.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0. 20.  0.]
 [ 0.  0.  0. 14.  0. 13.  0.  0. 13.  0.]
 [ 0.  0.  0.  0. 18.  0.  0.  0. 16.  0.]
 [ 0.  0. 15.  0.  0. 14.  0.  0.  0. 19.]]
```

- a matrix in which most of the elements are zero.
- if most elements are non zero - Dense Matrix
- $Density = \frac{\text{No. of Non-Zero Elements}}{\text{Total No. of Elements}}$
- $Sparsity = \frac{\text{No. of Zero Elements}}{\text{Total No. of Elements}} = 1 - Density$

Sparse Matrix into 3 tuple

$$\begin{bmatrix} 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Rows	Columns	values
5	6	5
0	2	8
1	1	7
2	5	5
3	1	3
4	4	1

- Imagine $A[100][100]$ with very high sparsity say 90%
- Inefficient (space) if we use a 100×100 matrix
- No. of Rows, No. of Columns, No. of Non-zero elements - in first row

Sparse Matrix into 3 tuple- Algorithm 0

$$\begin{bmatrix} 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Rows	Columns	values
5	6	5
0	2	8
1	1	7
2	5	5
3	1	3
4	4	1

- Identify the size of matrix
- identify the number of non zero elements
- Create a new 2D array to store the details of non zero elements
    - row number, column number, value

Lecture 08

We Saw & Will See

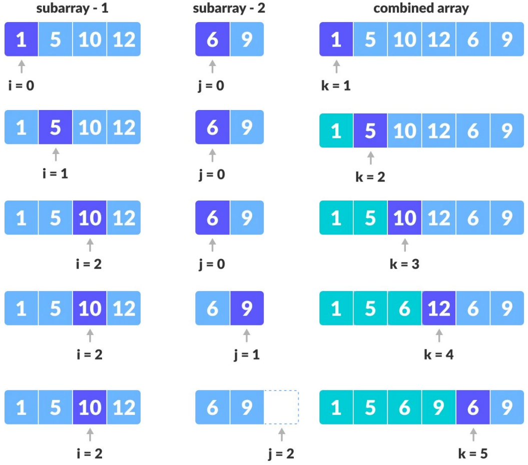
Till Now We Saw...

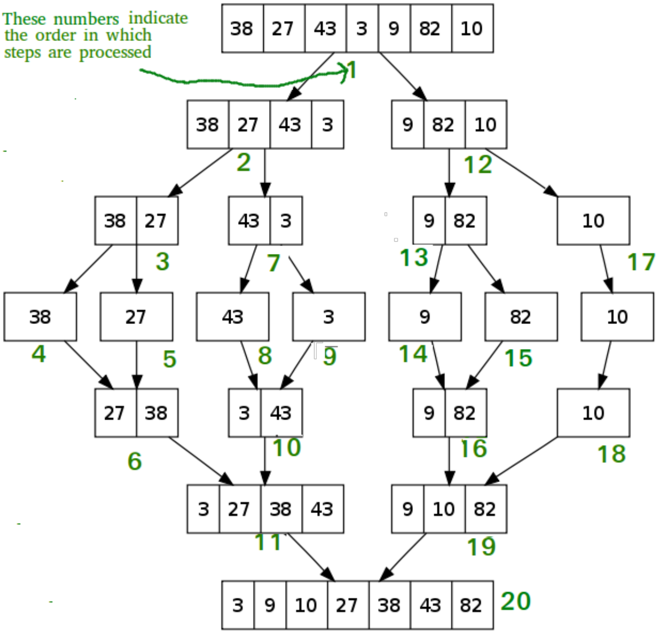
- Module 1 : Introduction to Data Structures
  - Concept, Classification - ADT, CDT, Linear/Non-linear DS
  - Searching - Linear and Binary Searches
  - Sorting - Bubble, Selection, Insertion
  - Sparse Matrices

Today We Will See...

- Merge in Merge Sort

Merge Two Sorted Arrays



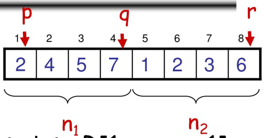

These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort

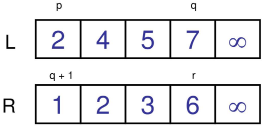
- it is a Divide and Conquer technique!
- divides the input array into two halves
- calls itself for the two halves
- then merges the two sorted halves.
- merge(arr, p, q, r) is a key process that assumes that arr[p..q] and arr[q+1..r] are sorted and merges the two sorted sub-arrays into one

Merge - Pseudocode

Alg.: MERGE(A, p, q, r)



1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1 \,..\, n_1 + 1]$ and the next n_2 elements into $R[1 \,..\, n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \le R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$



18

Lecture 09

We Saw & Will See

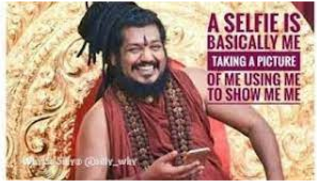
Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Concept, Classification - ADT, CDT, Linear/Non-linear DS
  - Searching - Linear and Binary Searches
  - Sorting - Bubble, Selection, Insertion
  - Sparse Matrices
  - Merge in Merge Sort

Today We Will See...

- Recap of Recursion
- Merge Sort Algorithm

Recap :Recursion - Properties



- A function which calls itself is called Recursion
- base case - for termination
- without base case - infinite loop

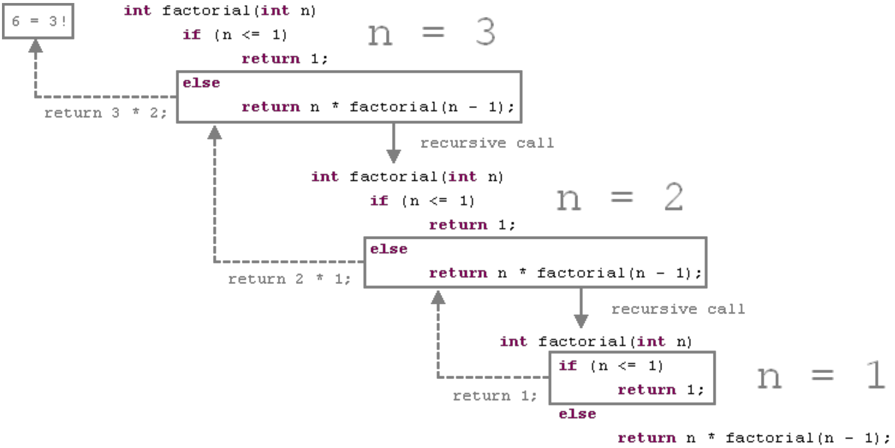
Iterative Method

```
int factorial(int);
int main()
{
 int n,f;
 scanf("%d", &n);
 f = factorial(n);
 printf("%d", f);
 return 1;
}
int factorial(int n)
{
 int ff=1;
 while(n>0){
  ff = ff*n; n--;
 }
 return ff;
}
```

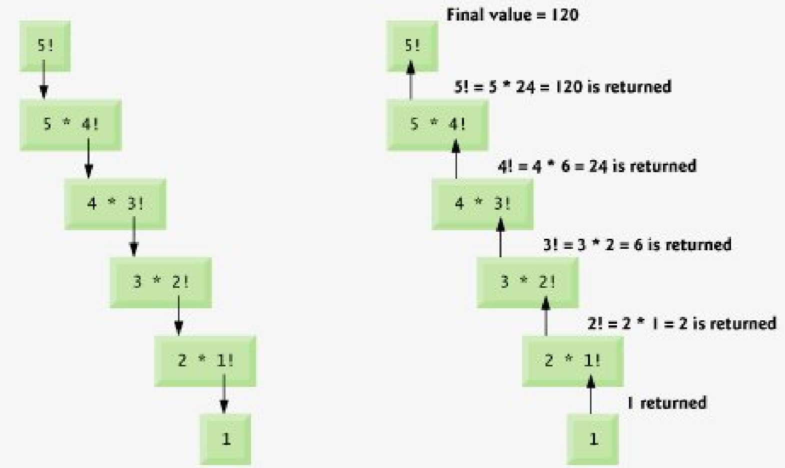
Recursive Method

```
int factorial(int);
int main()
{
   int n,f;
   scanf("%d", &n);
   f = factorial(n);
   printf("%d", f);
   return 1;
}
int factorial(int n)
{
   if(n<=1)
    return 1;
   else
    return n* factorial(n-1);
}
```

Factorial Example



Factorial Example - Pictorial



(a) Sequence of recursive calls.

(b) Values returned from each recursive call.

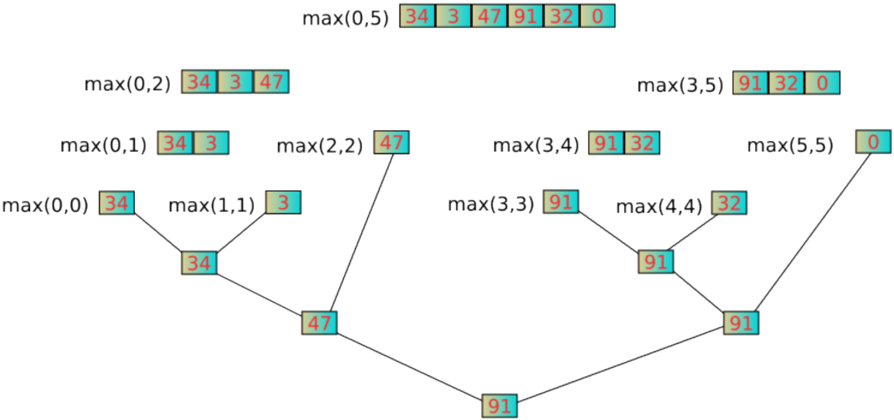
Find Maximum in Array : Algorithms

Iterative Method

- fix the first item in array as current maximum
- compare maximum with all values from 2^{nd} to last index one by one
- at any comparison, we find a value greater than current maximum, update the current maximum with this value
- Return current maximum upon reaching end

Recursive Method

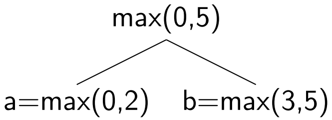
- Recursively get the maximum from first half and second half
- Compare both and return the maximum
- if only one item return it

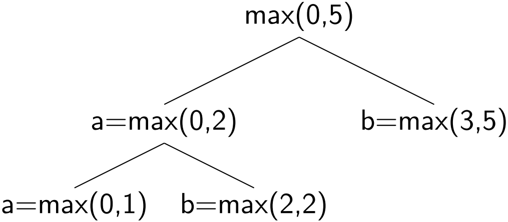

max(0,5) 34 3 47 91 32 0
max(0,2) 34 3 47
max(3,5) 91 32 0
max(0,1) 34 3
max(2,2) 47
max(3,4) 91 32
max(5,5) 0
max(0,0) 34
max(1,1) 3
max(3,3) 91
max(4,4) 32
34
91
47
91
91

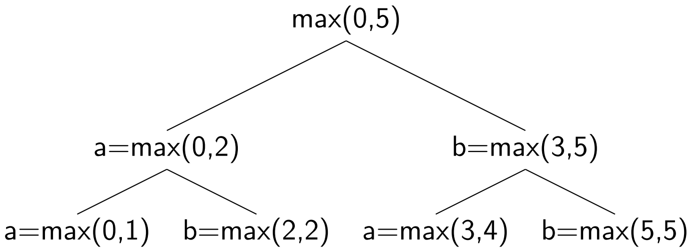
Find Maximum in Array using Recursion

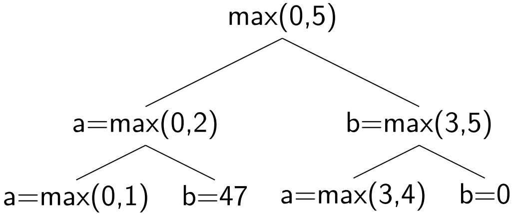
```
#include<stdio.h>
int arr[6]={34,3,47,91,32,0};
int max(int , int);
int main()
{
 int x;
 x = max(0,5);
 printf("%d\n",x);
 return 1;
}
```

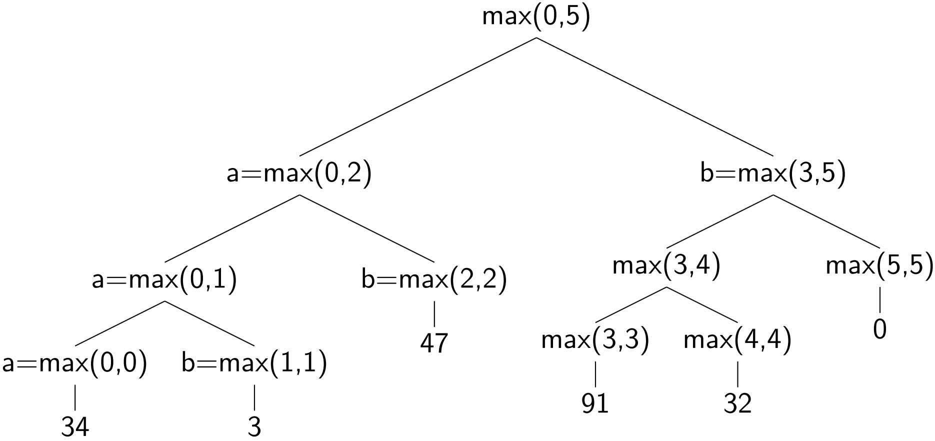
```
int max(int i, int j)
{
 int a,b;
 if(i==j)
   return arr[i];
 else{
   a = max(i,i+(j-i)/2);
   b = max(i+(j-i)/2+1,j);
 }
 if(a>b)
   return a;
 else
   return b;
}
```

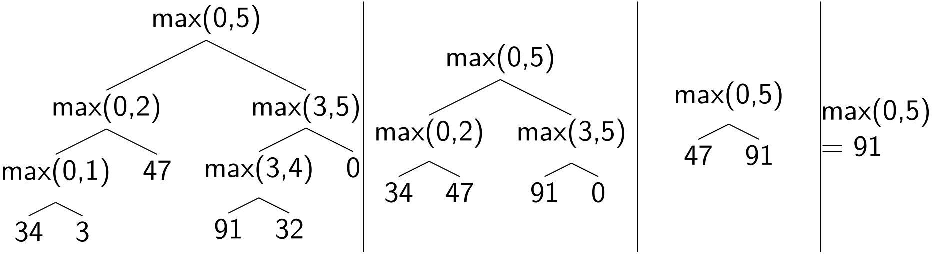

max(0,5)
a=max(0,2)
b=max(3,5)


max(0,5)
a=max(0,2)
b=max(3,5)
a=max(0,1)
b=max(2,2)


max(0,5)
a=max(0,2)
b=max(3,5)
a=max(0,1)
b=max(2,2)
a=max(3,4)
b=max(5,5)


max(0,5)
a=max(0,2)
b=max(3,5)
a=max(0,1)
b=47
a=max(3,4)
b=0

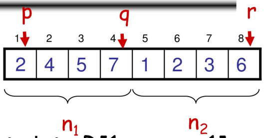

max(0,5)
a=max(0,2)
b=max(3,5)
a=max(0,1)
b=max(2,2)
max(3,4)
max(5,5)
a=max(0,0)
b=max(1,1)
47
max(3,3)
max(4,4)
0
34
3
91
32


max(0,5)
max(0,2)
max(3,5)
max(0,1)
47
max(3,4)
0
34
3
91
32
max(0,5)
max(0,2)
max(3,5)
34
47
91
0
max(0,5)
47
91
max(0,5)
= 91

Merge Sort Algorithm

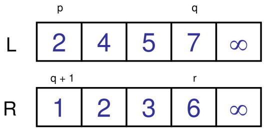
```
MERGE-SORT(A, p, r)
if p < r
    q = ⌊(p + r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q + 1, r)
    MERGE(A, p, q, r)
```

Merge - Pseudocode



Alg.: MERGE(A, p, q, r)

1. Compute n_1 and n_2
2. Copy the first n_1 elements into $L[1\,..\,n_1 + 1]$ and the next n_2 elements into $R[1\,..\,n_2 + 1]$
3. $L[n_1 + 1] \leftarrow \infty$; $R[n_2 + 1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. **for** $k \leftarrow p$ **to** r
6. **do if** $L[\,i\,] \leq R[\,j\,]$
7. **then** $A[k] \leftarrow L[\,i\,]$
8. $i \leftarrow i + 1$
9. **else** $A[k] \leftarrow R[\,j\,]$
10. $j \leftarrow j + 1$



18

Lecture 10

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Concept, Classification - ADT, CDT, Linear/Non-linear DS
  - Searching - Linear and Binary Searches
  - Sorting - Bubble, Selection, Insertion ($O(n^2)$)
  - Sparse Matrices
  - Sorting - Merge Sort ($O(n \log n)$)

Today We Will See...

- Partition in Quick Sort

Partitioning based on Pivot

- k^{th} smallest in a sorted array will be at k^{th}position

Partitioning based on Pivot

- k^{th} smallest in a sorted array will be at k^{th}position
- All elements $< k$ will occupy the first $k-1$positions

Partitioning based on Pivot

- k^{th} smallest in a sorted array will be at k^{th}position
- All elements $< k$ will occupy the first $k - 1$positions
- All elements $> k$ will occupy the last $n - k$ positions

Partitioning based on Pivot

- k^{th} smallest in a sorted array will be at k^{th}position
- All elements $< k$ will occupy the first $k-1$positions
- All elements $> k$ will occupy the last $n-k$ positions
- Aim of Partition
    - find the correct position of pivot

Partitioning based on Pivot

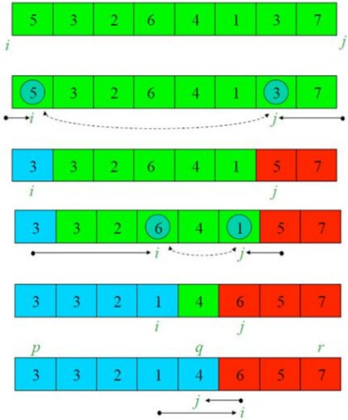
- k^{th} smallest in a sorted array will be at k^{th}position
- All elements $< k$ will occupy the first $k-1$positions
- All elements $> k$ will occupy the last $n-k$ positions
- Aim of Partition
  - find the correct position of pivot
  - all elements less than pivot are on its left

Partitioning based on Pivot

- k^{th} smallest in a sorted array will be at k^{th}position
- All elements $< k$ will occupy the first $k - 1$positions
- All elements $> k$ will occupy the last $n - k$ positions
- Aim of Partition
  - find the correct position of pivot
  - all elements less than pivot are on its left
  - all elements greater than or equal to are on its right

Quick sort- Partition Step

Hoare's Partitioning Algorithm - Ex1 (pivot=5)



Termination: $i = 6; j = 5,$ i.e., $i = j + 1$

Analysis of Algorithms 8

Partition Excercise

$$10, 4, 8, 15, 7, 99, 34, 43, 12, 6, 29, 3$$

- Take any number of your liking as Pivot
- Swap it with the first element
- Do Partition with respect to the selected Pivot

Partition Algorithm- partition(A, lo, hi)

```
pivot = A[lo]
i = lo - 1  // Initialize left index
j = hi + 1  // Initialize right index
while(true){
  do
     i = i + 1;
  while(A[i] < pivot) //Find in left side a value>pivot
  do
     j = j - 1;
  while (A[j] > pivot) //Find in right side a value<pivot
  if i >= j then
     return j
  swap A[i] with A[j]
}
```

Lecture 11

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Concept, Classification - ADT, CDT, Linear/Non-linear DS
  - Searching - Linear and Binary Searches
  - Sorting - Bubble, Selection, Insertion ($O(n^2)$)
  - Sparse Matrices
  - Sorting - Merge Sort ($O(n \log n)$)
  - Partition in Quick Sort

Today We Will See...

- Quick Sort Algorithm

Quick Sort Algorithm

```
quicksort( A, low, high)
{
    // base condition
    if (low >= high) {
        return;
    }
    // rearrange elements across pivot
    split = partition(A, low, high);

    quicksort(A, low, split);

    quicksort(A, split + 1, high);
}
```

Partition 0 - 9

4	2	5	14	11	8	15	13	19	10
0	1	2	3	4	5	6	7	8	9

Partition 0 - 1

2	4	5	14	11	8	15	13	19	10
0	1	2	3	4	5	6	7	8	9

Partition 3 - 9

2	4	5	8	10	19	13	15	11	14
0	1	2	3	4	5	6	7	8	9

Partition 5 - 9

2	4	5	8	10	13	11	14	15	19
0	1	2	3	4	5	6	7	8	9

Partition 5 - 6

2	4	5	8	10	11	13	14	15	19
0	1	2	3	4	5	6	7	8	9

Partition 8 - 9

2	4	5	8	10	11	13	14	15	19
0	1	2	3	4	5	6	7	8	9

2	4	5	8	10	11	13	14	15	19
0	1	2	3	4	5	6	7	8	9

Lecture 12

We Saw & Will See

Till Now We Saw...

- Module 1 : Introduction to Data Structures
  - Concept, Classification - ADT, CDT, Linear/Non-linear DS
  - Searching - Linear and Binary Searches
  - Sorting - Bubble, Selection, Insertion ($O(n^2)$)
  - Sparse Matrices
  - Sorting - Merge Sort, Quick Sort ($O(n \log n)$)

Today We Will See...

- Analysis of Sorting Algorithms

Running Time Complexity of Sorting Algorithms

- The time for Sorting Algorithm depends mainly on the number of comparisons made.
- We consider Best Case, Worst Case and Average Case time complexities
- All the methods that we saw are based on comparison
- Let $T(n)$ represent the time taken for an n input values

Running Time Complexity - Bubble Sort

Number of Comparisons : First Round $n-1$ comparisons, Second Round $n-2$ comparisons... so on. Second last Round have 2 comparisons. Last Round has 1 comparison.

- Best Case : $O(n)$ by checking initially if the array is sorted
- Worst Case & Average Case :

$$
\begin{aligned}
T(n) &= (n-1) + (n-2) + \ldots + 2 + 1 \\
&= \frac{n(n-1)}{2} \\
&= \frac{n^2}{2} - \frac{n}{2} \\
&= O(n^2)
\end{aligned}
$$

Running Time Complexity - Selection Sort

Number of Comparisons : First Round $n-1$ comparisons, Second Round $n-2$ comparisons... so on. Second last Round have 2 comparisons. Last Round has 1 comparison.

- Best Case : $O(n)$ by checking initially if the array is sorted
- Worst Case & Average Case :

$$\begin{aligned} T(n) &= (n-1)+(n-2)+\ldots+2+1 \\ &= \frac{n(n-1)}{2} \\ &= \frac{n^2}{2}-\frac{n}{2} \\ &= O(n^2) \end{aligned}$$

Running Time Complexity - Insertion Sort

Comparison may vary according to the next value to be inserted. First element inserted without any comparison. Second element needs 1 comparison. Third element needs upto 2 comparisons. ...so on

- Best Case : $O(n)$, one comparison for each element inserted
- Worst Case : $i-1$ comparisons for the i^{th} element insertion

$$\begin{aligned} T(n) &= 0 + 1 + \ldots (n-2) + (n-1) \\ &= \frac{n(n-1)}{2} \\ &= O(n^2) \end{aligned}$$

- Average Case: Each i^{th} element on an average needs $\frac{i}{2}$ comparisons. So it takes half the time with respect to Worst Case.

$$T(n) = \frac{1}{2} * \frac{n(n-1)}{2} = \frac{n(n-1)}{4} = O(n^2)$$

Running Time Complexity -Merge Sort & Quick Sort

- Both Merge and Quick sorts uses a divide and conquer technique. The recurrence relation of its running time can be written as below.

Let $T(1) = c_1$ be the constant time taken for doing sorting on single element(i.e. $n = 1$.

$$
\begin{aligned}
T(n) &= 2T(\frac{n}{2}) + O(n) \\
&\leq 2T(\frac{n}{2}) + cn \\
&\leq 2(2T\frac{n}{4}) + c\frac{n}{2}) + cn \\
&\leq 4T(\frac{n}{4}) + 2cn && = 2^2T(\frac{n}{2^2}) + 2cn \\
&\leq 8T(\frac{n}{8}) + 3cn && = 2^3T(\frac{n}{2^3}) + 3cn \\
&\dots \\
&\leq 2^iT(\frac{n}{2^i}) + i \times cn
\end{aligned}
$$

$$T(n) \leq 2^i T(\frac{n}{2^i}) + i \times cn$$

We know $T(1) = c_1$, a constant. We now find value of i, for which $\frac{n}{2^i} = 1$.

$$2^i = n \text{ or } i = \log n$$

Putting value of i in the above equation we get

$$\begin{aligned} T(n) &\leq 2^{\log n} T(\frac{n}{2^{\log n}}) + i \times cn \\ &\leq nT(1) + \log n \times cn \\ &\leq n \times c_1 + cn \times \log n \\ &= O(n \log n) \end{aligned}$$

Time Complexity comparison of Sorting Algorithms

Algorithm	Data Structure	Time Complexity		
		Best	Average	Worst
Quicksort	Array	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	Array	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	Array	O(n)	O(n^2)	O(n^2)
Insertion Sort	Array	O(n)	O(n^2)	O(n^2)
Select Sort	Array	O(n^2)	O(n^2)	O(n^2)
Bucket Sort	Array	O(n+k)	O(n+k)	O(n^2)