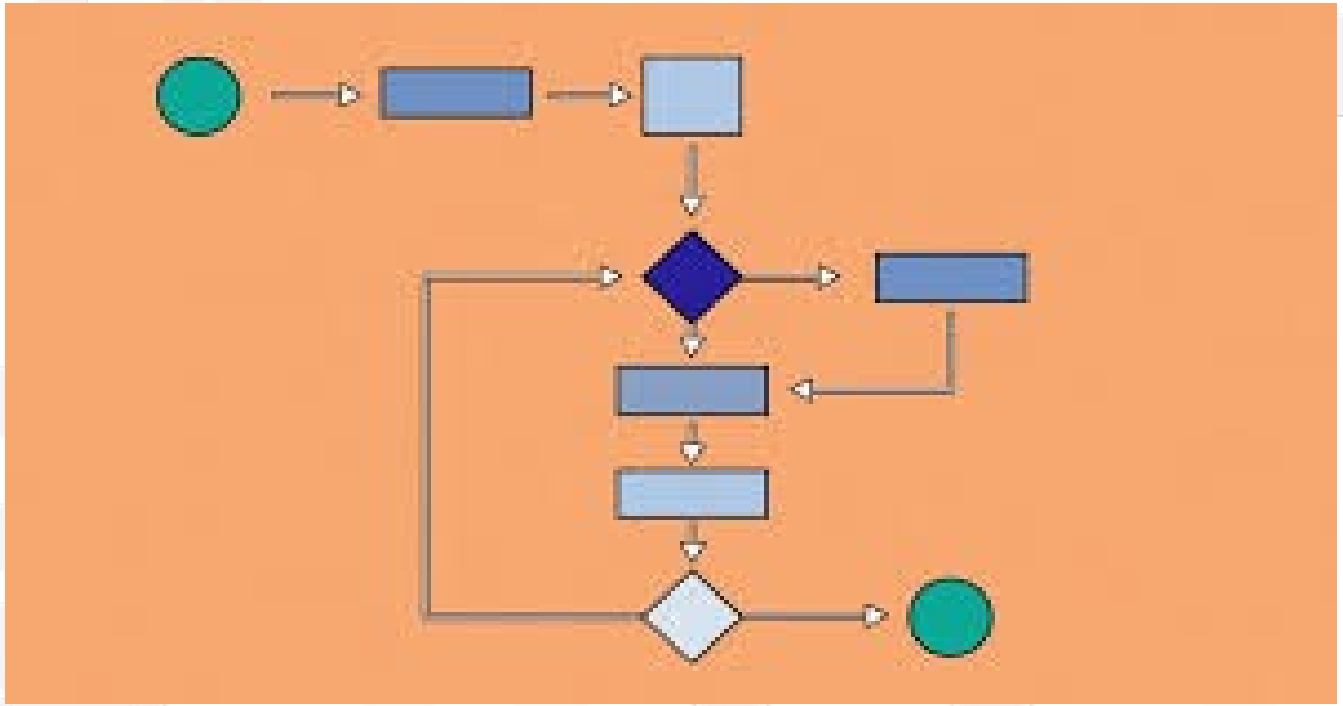
Implementing Sorting and Searching

Session Goals

By the end of this session, you will be able to demonstrate how to:

- **Perform sorting**
- **Perform efficient searching**
- **Apply divide and conquer**

Context Setting



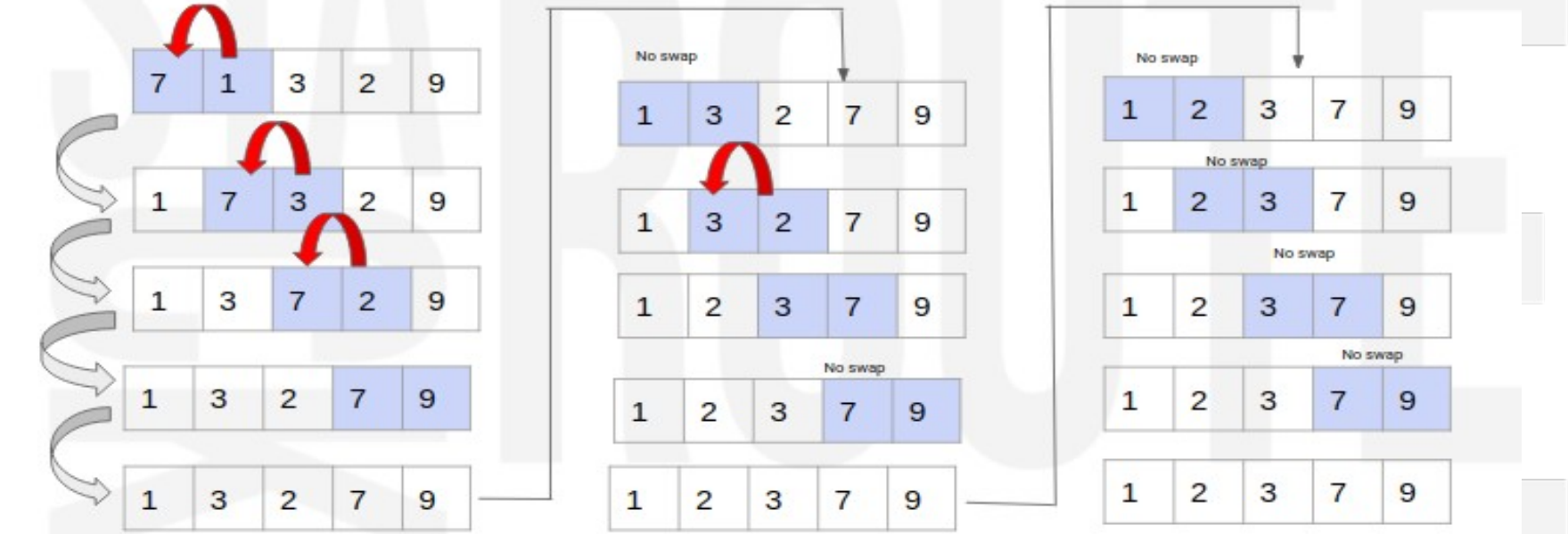
Let Us Try to Find Out

Why do we need Algorithms?
How to implement Sorting?
How to implement Searching?

Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)	O(log(n))
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n+k)

Bubble Sort



Bubble Sort Demo

```
public class BubbleSortDemo {
   public static void main(String args[]){
       int  input[]={70,1,3,20,6,34,94,39};
       bubbleSort(input);}
   public static int[] bubbleSort(int input[]){
       for (int i = 0; i < input.length; i++) {
           for (int j = 0; j < input.length-1-i; j++) {
               if(input[j]>input[j+1]){
                   int temp=input[j];
                   input[j]=input[j+1];
                   input[j+1]=temp; }}
         System.out.print("Iteration "+(i+1)+": ");
           printArray(input);}
       return input;}
   public static void printArray(int input[]){
       for (int i = 0; i <input.length; i++) {
           System.out.print(input[i]+" ");}
       System.out.println();   }}
```

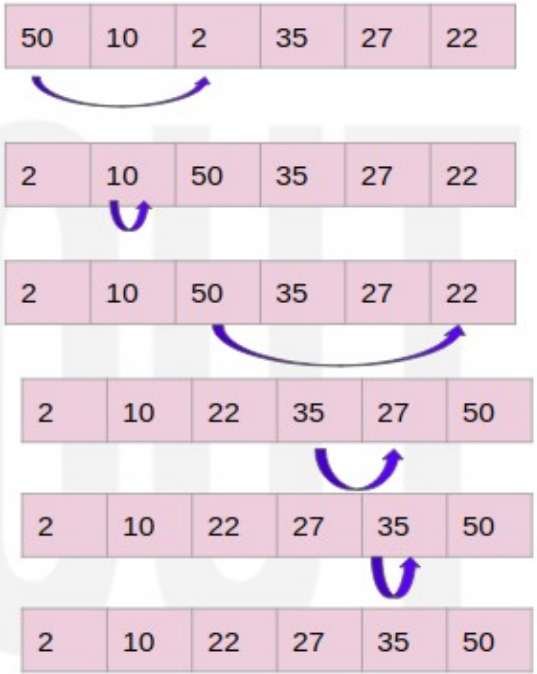
Output :

Iteration 1: 1 3 20 6 34 70 39 94
Iteration 2: 1 3 6 20 34 39 70 94
Iteration 3: 1 3 6 20 34 39 70 94
Iteration 4: 1 3 6 20 34 39 70 94
Iteration 5: 1 3 6 20 34 39 70 94
Iteration 6: 1 3 6 20 34 39 70 94
Iteration 7: 1 3 6 20 34 39 70 94
Iteration 8: 1 3 6 20 34 39 70 94

Selection Sort

Selection sort algorithm

- Find the minimum element in the list.
- Swap minimum element with current element.
- Repeat the whole process until array is fully sorted.



Selection Sort Demo

```
public class SelectionSortDemo {
   public static int[] selectionSort(int[] input){
       for (int i = 0; i < input.length - 1; i++){
           int index = i;
           for (int j = i + 1; j < input.length; j++)
               if (input[j] < input[index])
                   index = j;
           int smallerNumber = input[index];
           input[index] = input[i];
           input[i] = smallerNumber;}
       return input;}
   public static void main(String a[]){
       int[] input = {50,10,2,35,27,22};
       System.out.println("Array before Sorting : ");
       System.out.println(Arrays.toString(input));
       input = selectionSort(input);
       System.out.println("********************");
       System.out.println("Array after Sorting : ");
       System.out.println(Arrays.toString(input));   }}
```

Output:

Array before Sorting :

[50, 10, 2, 35, 27, 22]

****

Array after Sorting :

[2, 10, 22, 27, 35, 50]

Quick Sort

Quick sort

- Uses divide and conquer
- Also known as partition exchange sort

In quick sort,

- Choose a pivot and divide into two sublists. One sublist will contain elements lower than pivot and other will have elements greater than pivot.
- Sort the sublist is recursively

Quick Sort Demo

```
public class QuickSortDemo {
   private static int input[];
   public static void sort(int[] a) {
       if (a == null || a.length == 0){
           return;}
       input = ar;
       quickSort(0, input.length-1);}
    private static void quickSort(int left, int
right) {
       int i = left;
       int j = right;
int pivot = input[left+(right-left)/2];
       while (i <= j) {
while (input[i] < pivot) { i++; }
           while (input[j] > pivot) {j--;}
           if (i <= j) {
               exchange(i, j);i++;
               j--;}}
```

Quick Sort Demo Contd.

```
if (left < j){quickSort(left, j);}
        if (i < right){quickSort(i, right);}}
    private static void exchange(int i, int j) {
        int temp = input[i];
        input[i] = input[j];
        input[j] = temp;}
    public static void main(String a[]){
        int[] input = {3,21,65,74,7,234,11,19,4,8,9};
           System.out.println("Array Before Sorting :
"); System.out.println(Arrays.toString(input));
sort(input);
System.out.println("**********************************
****");
         System.out.println("Array After Sorting : ");
     System.out.println(Arrays.toString(input));
    }
}
```

Output:

Array Before Sorting :

[3, 21, 65, 74, 7, 234, 11, 19, 4, 8, 9]

****

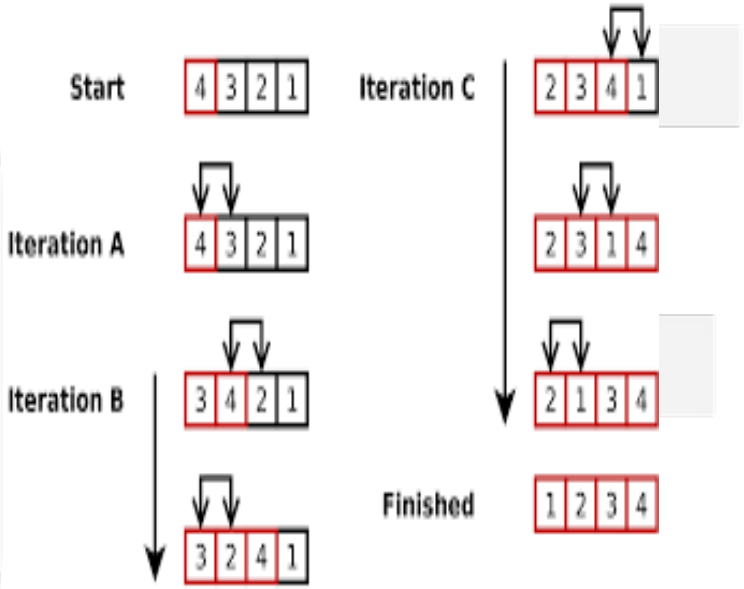
Array After Sorting :

[3, 4, 7, 8, 9, 11, 19, 21, 65, 74, 234]

Insertion Sort

Insertion sort:

- Compares values at index with all its prior elements.
- A value is placed at the index where there are no lesser value to the elements.
- When you reach last element,we get a sorted array.



Insertion Sort Demo

```
public class InsertionSortDemo {
   public static void main(String args[]) {
       int  input[]={90,56,74,1,56,5,19};
       insertionSort(input);}
   public static int[] insertionSort(int input[]){
       for (int i = 1; i < input.length; i++){
           int sortValue = input[i];
           int j;
for ( j = i; j > 0 && input[j - 1] > sortValue; j--) {
               input[j] = input[j - 1];}
input[j] = sortValue;
           System.out.print("Iteration "+(i)+": ");
           printArray(input);}
       return input;}
   public static void printArray(int input[]){
       for (int i = 0; i <input.length; i++) {
           System.out.print(input[i]+" ");}
       System.out.println();}}
```

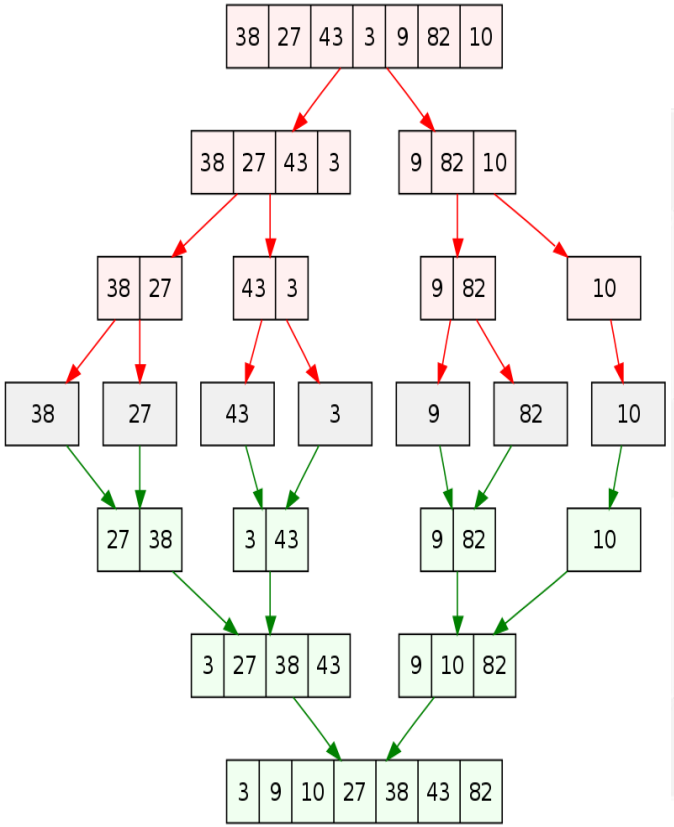
Output:

```
Iteration 1: 56 90 74 1 56 5 19
Iteration 2: 56 74 90 1 56 5 19
Iteration 3: 1 56 74 90 56 5 19
Iteration 4: 1 56 56 74 90 5 19
Iteration 5: 1 5 56 56 74 90 19
Iteration 6: 1 5 19 56 56 74 90
```

Merge Sort

Merge Sort works as listed below:

- First, divide list into sublist of about half size in each iteration until each sublist has only one element
- Then, merge each sublist repeatedly to create sorted list.



Merge Sort Demo

```
public class MergeSortDemo {
   static int input[] = {90,56,74,1,56,5,19};
   public static void main(String args[]) {
       System.out.println("Before sorting:");
       printArray(input, 0, input.length - 1);
       System.out.println("*****************************");
       mergeSort(0, input.length - 1);
       System.out.println("*****************************");
       System.out.println("Array After sorting:");
       printArray(input, 0, input.length - 1);
   }
   // Recursive algorithm for merge sort
   public static void mergeSort(int start, int end) {
       int mid = (start + end) / 2;
       if (start < end) {
                   mergeSort(start, mid);
                   mergeSort(mid + 1, end);
                   merge(start, mid, end);}}
```

Merge Sort Demo Contd.

```
public static void printArray(int input[], int start, int end) {
        for (int i = start; i <= end; i++) {
            System.out.print(input[i] + " ");}
        System.out.println();
    }
     private static void merge(int start, int mid, int end) {
        int[] temp = new int[input.length];
        int tempIndex = start;
        System.out.print("Before Merging:  ");
        printArray(input, start, end);
        int startIndex = start;
        int midIndex = mid + 1;
        while (startIndex <= mid && midIndex <= end) {
            if (input[startIndex] < input[midIndex]) {
                temp[tempIndex++] = input[startIndex++];
            } else {
                temp[tempIndex++] = input[midIndex++];
            }}
```

Merge Sort Demo Contd.

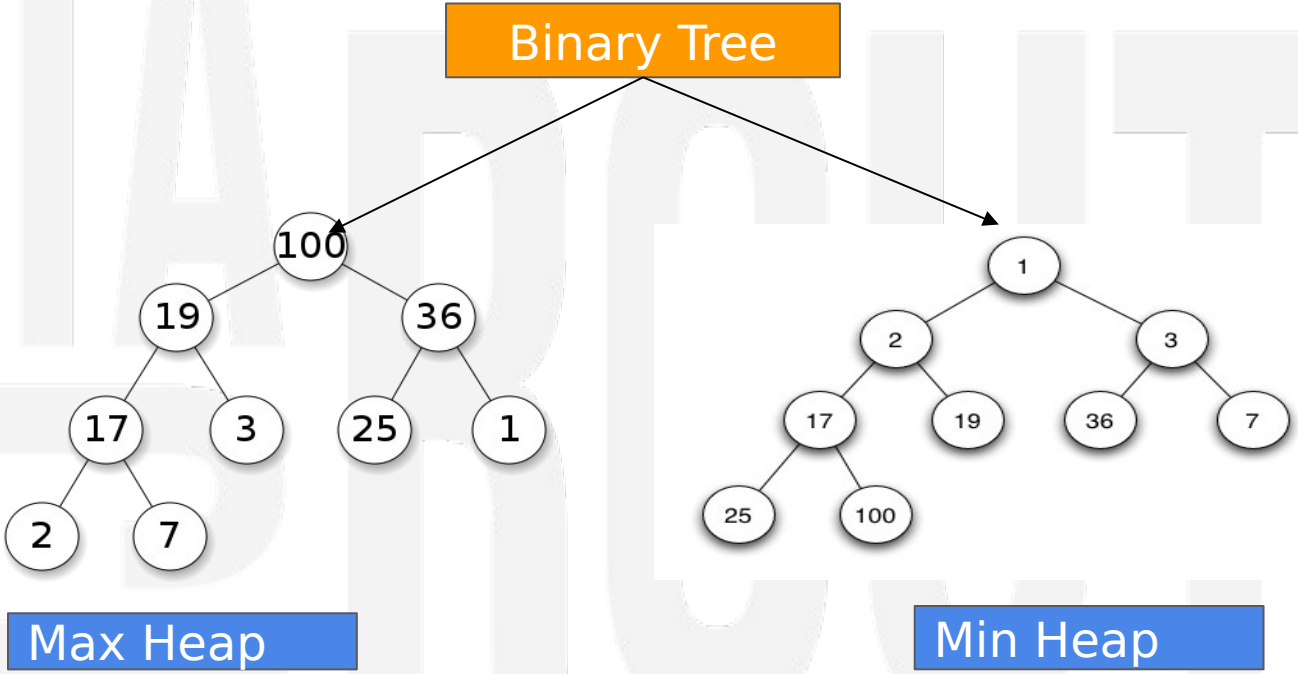
```
//Copy remaining elements
        while (startIndex <= mid) {
            temp[tempIndex++] = input[startIndex++];
        }
        while (midIndex <= end) {
            temp[tempIndex++] = input[midIndex++];
        }

// Copy temp to actual inputay after sorting
        for (int i = start; i <= end; i++) {
            input[i] = temp[i];
        }
        System.out.print("After merging:   ");
        printArray(temp, start, end);
       // System.out.println();
    }
}
```

Output:

```
Before sorting:
90 56 74 1 56 5 19
****************************
Before Merging:  90 56
After merging:   56 90
Before Merging:  74 1
After merging:   1 74
Before Merging:  56 90 1 74
After merging:   1 56 74 90
Before Merging:  56 5
After merging:   5 56
Before Merging:  5 56 19
After merging:   5 19 56
Before Merging:  1 56 74 90 5 19 56
After merging:   1 5 19 56 56 74 90
****************************
Array After sorting:
1 5 19 56 56 74 90
```

Heap Sort



Heapifying Process

- Heapify is the process of converting a binary tree into a Heap data structure.
- Heapify process is used to build max-heap.

Heap Sort Demo

```
public class HeapSortDemo {
   public static void buildHeap(int []input) {
for(int i=(input.length-1)/2; i>=0; i--){
          heapify(input,i,input.length-1);}}
   public static void heapify(int[] input, int i,int size) {
       int left = 2*i+1;
       int right = 2*i+2;
       int max;
       if(left <= size && input[left] > input[i]){
           max=left;} else {max=i;}
       if(right <= size && input[right] > input[max]) {
           max=right;} if(max!=i) {
           exchange(input,i, max);
           heapify(input, max,size);}}
public static void exchange(int[] input,int i, int j) {
       int t = input[i];
       input[i] =input[j];
       input[j] = t;}
```

Heap Sort Demo Contd.

```
    public static int[] heapSort(int[] input) {
        buildHeap(input);
        int sizeOfHeap=input.length-1;
        for(int i=sizeOfHeap; i>0; i--) {
            exchange(input,0, i);
            sizeOfHeap=sizeOfHeap-1;
            heapify(input, 0,sizeOfHeap);}
        return input;}
    public static void main(String[] args) {
        int[] input={1,10,16,19,3,5};
        System.out.println("Before Heap Sort : ");
        System.out.println(Arrays.toString(input));
        input=heapSort(input);
        System.out.println("*****************************");
        System.out.println("After Heap Sort : ");
        System.out.println(Arrays.toString(input));
    }}
```

Output:

Before Heap Sort :

[1, 10, 16, 19, 3, 5]

****

***

After Heap Sort :

[1, 3, 5, 10, 16, 19]

Searching Algorithms

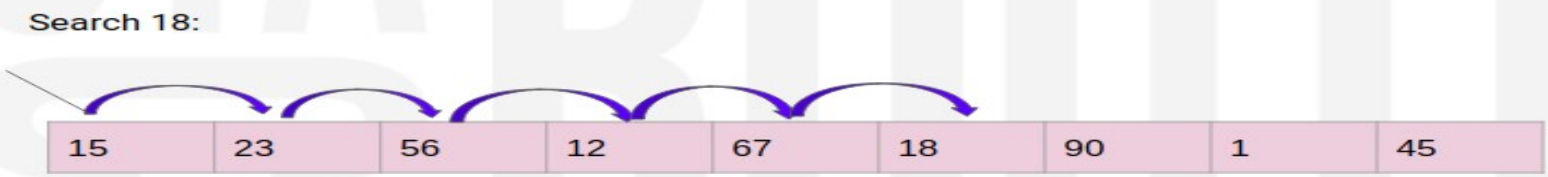
Searching approaches:

- Sequential Search
- Interval Search

Linear Search Algorithm

Steps :

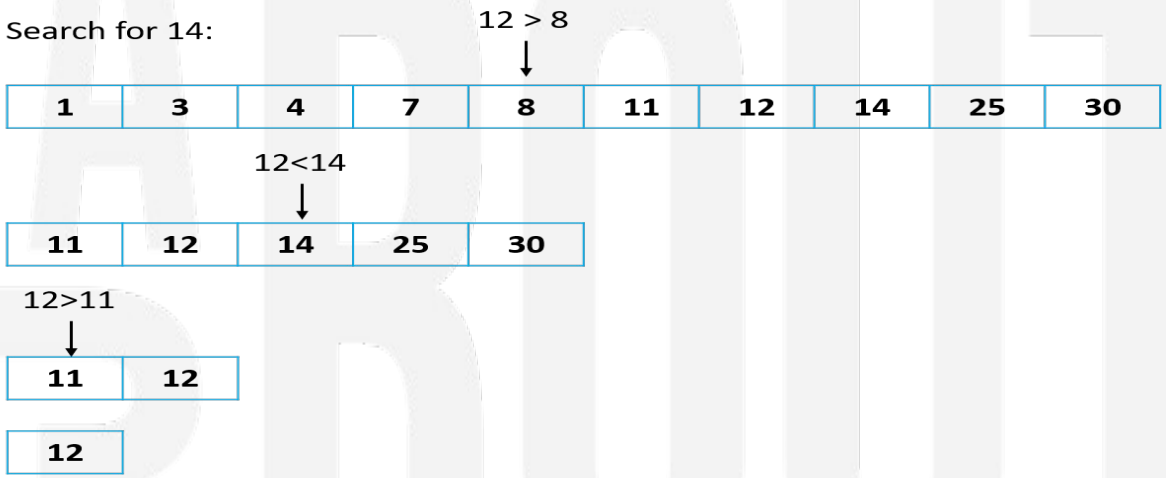
- Start from the leftmost element of arr[] and one by one compare y with each element of array[]
- If y matches with an element, return the index.
- If y doesn’t match with any of elements, return -1.



Linear Search Demo

```
class LinearSearchDemo{
   public static int linearSearch(int input[], int x){
        int n = input.length;
        for(int i = 0; i < n; i++){
            if(input[i] == x)
                return i;}
        return -1;}
   public static void main(String args[]){
        int input[] = { 56,12,13,8,98,87 };
        int x =98;
        int result = linearSearch(input, x);
        if(result == -1)
            System.out.print("Element is not present in array");
        else
            System.out.print("Element is present at index " + result);
   }}
Output:
Element is present at index 4
```

Binary Search Algorithm



Binary Search Demo

```
class BinarySearchDemo {
int binarySearch(int input[], int left, int right, int x){
        if (right >= left) {
            int mid = left + (right - left) / 2;
            if (input[mid] == x)
                if (input[mid] > x)
                    return binarySearch(input, left, mid - 1, x);
            return binarySearch(input, mid + 1, right, x);}
        return -1;}
public static void main(String args[]){
        BinarySearchDemo ob = new BinarySearchDemo();
        int input[] = { 10,56,78,5,45,23 };
        int n = input.length;
        int x = 45;
        int result = ob.binarySearch(input,0,n-1,x);
        if (result == -1)
            System.out.println("Element not present");
        else
            System.out.println("Element found at index " + result);}}
```

Output: Element found at index 4

Jump Search Algorithm

- It is similar to linear search but instead of searching each element linearly we will jump in the interval sqrt(array(length))
- Searching continues until we reach an element greater than current element or end of the array.
- It requires the collections to be sorted

Jump Search Demo

```
class JumpSearchDemo {
   public static int jumpSearch(int[] myArray, int elementToSearch) {
       int length = myArray.length;
       int jumpStep = (int) Math.sqrt(myArray.length);
       int previousStep = 0;
       while (myArray[Math.min(jumpStep, length) - 1] <
elementToSearch) {
           previousStep = jumpStep;
           jumpStep += (int) (Math.sqrt(length));
           if (previousStep >= length)return -1;}
       while (myArray[previousStep] < elementToSearch) {
           previousStep++;
           if (previousStep == Math.min(jumpStep, length))return -1;}
       if (myArray[previousStep] == elementToSearch)
           return previousStep;
       return -1;}
public static void main(String[] args) {
   int[] input = {1,5,9,23,45,67,88,102,198};
   int index = jumpSearch(input, 45);
   System.out.println("Element found at index " + index);}}
```

Output:

Element found at index 4

Interpolation Search Algorithm

- It is improved variant of binary search.
- This search is appropriate if we know that the data is sorted and uniformly distributed.
- It uses interpolation formula to find the best probable place where the element can be found in the array
- For this formulae to be effective the search array should be large otherwise it performs like Linear Search

Depth First Search (DFS) Algorithm

- It is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph)
- It explores as far as possible along each branch before backtracking

Breadth First Search Algorithm

- It is an algorithm for traversing or searching tree or graph data structures.
- The algorithm starts at the tree root(or some arbitrary node of a graph sometimes referred to as a 'search key’),
- It explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.

DFS and BFS Difference

Key	BFS	DFS
Definition	BFS, stands for Breadth First Search.	DFS, stands for Depth First Search.
Data structure	BFS uses Queue to find the shortest path.	DFS uses Stack to find the shortest path.
Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
Suitability for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.

Searching Algorithms

	Time Complexity
Linear Search	O (n)
Binary Search	O (log (n))
Jump Search	O (√ n)
Interpolation Search	O (log (log n))-Best | O (n)-Worst
Exponential Search	O (log (n))
Sequential search	O (n)
Depth-first search (DFS)	O (|V| + |E|)
Breadth-first search (BFS)	O (|V| + |E|)

Key TakeAways

By the end of this session, you should be able to demonstrate how to:

- Perform sorting
- Perform efficient searching
- Apply divide and conquer

Thank You!