14.7. Exploring Selection Sort

- Easy to implement the Selection sort
- Selection sort works well for small data sets.
- Selection sort is an in-place sorting algorithm.
- Selection sort is not a Stable sorting algorithm.

Algorithm

1) Find the minimum value in the list
2) Swap it with the value in the current position
3) Repeat this process for all the elements until the entire array is sorted

- This algorithm is called selection sort since it repeatedly selects the smallest element.

Complexity of Selection Sort:

Time Complexity - Best	$O(n^2)$
Time Complexity - Average	$O(n^2)$
Time Complexity - Worst	$O(n^2)$
Space Complexity	O(1)

14.7.1. Selection Sort Implementation

Lab13.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
```

```
public class Lab13 {

	public static void selectionSort(int arr[]) {

		int n = arr.length;

		for (int i = 0; i < n-1; i++) {
			int min_index = i;
			for (int j = i + 1; j < n; j++) {
				if (arr[j] < arr[min_index])
					min_index = j;
			}

			int temp = arr[min_index];
			arr[min_index] = arr[i];
			arr[i] = temp;

		}
	}

	public static void main(String[] args) {

		int arr[] = { 10, 5, 30, 25, 15, 20 };

		System.out.println(Arrays.toString(arr));
		selectionSort(arr);
		System.out.println(Arrays.toString(arr));
	}
}
```

14.8. Exploring Insertion Sort

- Insertion sort is a simple and efficient comparison sort.
- Insertion sort is an in-place sorting algorithm.
- Insertion sort is a Stable sorting algorithm.
- In this algorithm, each iteration
  - ✓ Removes an element from the input data,
  - ✓ Inserts it into the correct position in the already-sorted list until no input elements remain.

Algorithm

1) Consider some part is already Sorted
2) Take i^{th} element and Insert it in the Correct position
3) Repeat this process for all the elements until the entire array is sorted

- This algorithm is called Insertion Sort since it repeatedly Inserts the Element in Correct position in the Sorted part.

Complexity of Insertion Sort:

Time Complexity - Best	O(n)
Time Complexity - Average	$O(n^2)$
Time Complexity - Worst	$O(n^2)$
Space Complexity	O(1)

14.8.1. Insertion Sort Implementation

Lab14.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
```

```
**/
public class Lab14 {

	public static void insertionSort(int arr[]) {

		int n = arr.length;

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

			arr[j+1] = key;

		}

	}

	public static void main(String[] args) {

		int arr[] = { 10, 5, 30, 25, 15, 20 };

		System.out.println(Arrays.toString(arr));
		insertionSort(arr);
		System.out.println(Arrays.toString(arr));
	}
}
```

14.9. Exploring Merge Sort

- Merge sort is an example of the divide and conquer strategy.
- Merging is the process of combining two sorted arrays into one bigger sorted array
- Merge sort is Quick sort's complement
- Merge sort accesses the data in a sequential manner
- This algorithm is used for sorting a linked list
- erge sort divides the list into two parts; then each part is conquered individually.
- Merge sort starts with the small arrays and finishes with the largest one.
- Merge sort is not an in-place sorting algorithm.
- Merge sort is a Stable sorting algorithm.

Algorithm

1) Divide the Input List into two parts.
2) Solve each part recursively.
3) After solving the sub problems, they are merged by scanning the resultant sub problems.

This algorithm is called Mrge Sort since it is merging two sorted arrays to make one bigger sorted array.

Complexity of Merge Sort:

Time Complexity - Best	O(nlogn)
Time Complexity - Average	O(nlogn)
Time Complexity - Worst	O(nlogn)
Space Complexity	O(n)

14.9.1. Merge two Sorted Arrays

Lab15.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab15 {
	public static void mergeArrays(int a[],int b[]) {
		int m = a.length;
		int n = b.length;

		int i=0;
		int j=0;
		while(i<m && j<n) {
			if(a[i]<=b[j]) {
				System.out.println(a[i]);
				i++;
			}
			else {
				System.out.println(b[j]);
				j++;
			}
		}
		while(i<m ) {
				System.out.println(a[i]);
				i++;
		}
		while( j<n) {
				System.out.println(b[j]);
				j++;
		}
	}
```

```
        public static void main(String[] args) {

                int a[] = { 10, 20, 50 };
                int b[] = { 5, 15, 50 };
                mergeArrays(a,b);

        }
}
```

14.9.2. Implement Merge Function

Lab16.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

public class Lab16 {

        public static void merge(int arr[], int low, int mid, int high) {

                int m = mid - low + 1;
                int n = high - mid;

                int left[] = new int[m];
                for (int i = 0; i < m; i++) {
                        left[i] = arr[low + i];
                }

                int right[] = new int[n];
                for (int j = 0;j < m; j++) {
                        right[j] = arr[mid + j+1];
                }
```

```
            int i = 0;
            int j = 0;
            int k=low;

            while (i < m && j < n) {
                if (left[i] <= right[j]) {
                    arr[k] = left[i];
                    i++;
                    k++;
                } else {
                    arr[k] = right[j];
                    j++;
                    k++;
                }
            }
            while (i < m) {
                arr[k] = left[i];
                i++;
                k++;
            }
            while (j < n) {
                arr[k] = right[j];
                j++;
                k++;
            }
        }
        public static void main(String[] args) {
            int arr[] = { 10, 20, 50, 5, 15, 50 };
            int low = 0;       int high = 5;      int mid = 2;

            System.out.println(Arrays.toString(arr));
            merge(arr, low, mid, high);
            System.out.println(Arrays.toString(arr));
        }
}
```

14.9.3. Merge Sort Algorithm

Lab17.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab17 {
	public static void merge(int arr[], int low, int mid, int high) {

		int m = mid - low + 1;
		int n = high - mid;

		int left[] = new int[m];
		for (int i = 0; i < m; i++) {
			left[i] = arr[low + i];
		}

		int right[] = new int[n];
		for (int j = 0;j < n; j++) {
			right[j] = arr[mid + j+1];
		}

		int i = 0;
		int j = 0;
		int k=low;

		while (i < m && j < n) {
			if (left[i] <= right[j]) {
				arr[k] = left[i];
				i++;
				k++;
			} else {
				arr[k] = right[j];
```

```
                        j++;
                        k++;
                }
        }

        while (i < m) {
                arr[k] = left[i];
                i++;
                k++;
        }

        while (j < n) {
                arr[k] = right[j];
                j++;
                k++;
        }

    }
    public static void mergeSort(int arr[], int low, int high) {
        if(high>low) {
            int mid = low + (high - low)/2;
            mergeSort(arr,low,mid);
            mergeSort(arr,mid+1,high);
            merge(arr, low, mid, high);

        }
    }
    public static void main(String[] args) {
        int arr[] = { 10, 35, 20, 15, 5, 25 };
        int low = 0;            int high = 5;
        System.out.println(Arrays.toString(arr));
        mergeSort(arr, low,  high);
        System.out.println(Arrays.toString(arr));
    }
}
```

14.10. Exploring Quicksort

- Quick sort is an example of the divide and conquer strategy.
- It is one of the famous algorithms among comparison-based sorting algorithms
- It is also called partition exchange sort.
- It uses recursive calls for sorting the elements,
- Quick sort is an in-place sorting algorithm.
- Quick sort is a Non-Stable sorting algorithm.

Algorithm

- **The recursive algorithm consists of 4 steps:**
  1) If there are one or no elements in the array to be sorted, return.
  2) Pick an element in the array to serve as the "pivot" point.
  3) Split the array into two parts – one with elements larger than the pivot and the other with elements smaller than the pivot.
  4) Recursively repeat the algorithm for both halves of the original array.

Complexity of Quick Sort:

Time Complexity - Best	O(n logn)
Time Complexity - Average	O(n logn)
Time Complexity - Worst	$O(n^2)$
Space Complexity	O(n)

14.10.1. Partitioning the given Array

Lab18.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
```

```
* @Company: Java Learning Center
* */
public class Lab18 {
	public static void simplePartition(int arr[], int low,  int high, int pivot) {

		int n = high - low+1;
		int temp[] = new int[n];
		int k=0;

		for(int i=low;i<=high;i++) {
			if(arr[i]<=arr[pivot]) {
				temp[k++]=arr[i];
			}
		}

		for(int i=low;i<=high;i++) {
			if(arr[i]>arr[pivot]) {
				temp[k++]=arr[i];
			}
		}

		for(int i=low;i<=high;i++) {
			arr[i] = temp[i-low];
		}
	}

	public static void main(String[] args) {
		int arr[] = { 5,13,6,9,12,4,8,7};
		int low = 0;
		int high = arr.length-1;
		int pivot = high;
		System.out.println(Arrays.toString(arr));
		simplePartition(arr, low, high,pivot);
		System.out.println(Arrays.toString(arr));
	}
}
```

14.10.2. Lomuto Partition

Lab19.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab19 {

	public static int lomutoPartition(int arr[], int low,  int high) {

		int pivot = arr[high];
		int i=low-1;

		for(int j=low;j<=high-1;j++) {
			if(arr[j]<pivot) {
				i++;
				int temp= arr[i];
				arr[i]=arr[j];
				arr[j]=temp;
			}
		}

		int temp= arr[i+1];
		arr[i+1]=arr[high];
		arr[high]=temp;

		return (i+1);
	}

	public static void main(String[] args) {

		int arr[] = { 5,13,6,9,12,4,8,7};
		int low = 0;
```

```
                int high = arr.length-1;

                System.out.println(Arrays.toString(arr));
                int pivotIndex= lomutoPartition(arr, low, high);
                System.out.println(Arrays.toString(arr));
                System.out.println(pivotIndex);
        }
}
```

14.10.3. Hoares Partition

Lab20.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab20 {

        public static int hoaresPartition(int arr[], int low, int high) {

                int pivot = arr[low];
                int i = low - 1;
                int j = high + 1;

                while (true) {
                        do {
                                i++;
                        } while (arr[i] < pivot);

                        do {
                                j--;
                        } while (arr[j] > pivot);
```

```
                if (i >= j) {
                        return j;
                }

                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
        }
    }

    public static void main(String[] args) {

        int arr[] = {5,3,8,4,2,7,1,10};
        int low = 0;
        int high = arr.length-1;

        System.out.println(Arrays.toString(arr));
        int pivotIndex = hoaresPartition(arr, low, high);
        System.out.println(Arrays.toString(arr));
        System.out.println(pivotIndex);
    }
}
```

14.10.4. Quick Sort with Lomuto Partition

Lab21.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab21 {
    public static int lomutoPartition(int arr[], int low,  int high) {
```

```
            int pivot = arr[high];
            int i=low-1;

            for(int j=low;j<=high-1;j++) {
                if(arr[j]<pivot) {
                    i++;
                    int temp= arr[i];
                    arr[i]=arr[j];
                    arr[j]=temp;
                }
            }

            int temp= arr[i+1];
            arr[i+1]=arr[high];
            arr[high]=temp;

            return (i+1);
        }

        public static void quickSort(int arr[], int low,  int high) {
            if(low<high) {
                int pivotIndex=lomutoPartition(arr, low, high);
                quickSort(arr, low, pivotIndex-1);
                quickSort(arr, pivotIndex+1, high);
            }
        }

        public static void main(String[] args) {
            int arr[] = {8,4,7,9,3,10,5};
            int low = 0;
            int high = arr.length-1;
            System.out.println(Arrays.toString(arr));
            quickSort(arr,low,high);
            System.out.println(Arrays.toString(arr));
        }
}
```

14.10.5. Quick Sort with Hoares Partition

Lab22.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab22 {
	public static int hoaresPartition(int arr[], int low, int high) {

		int pivot = arr[low];
		int i = low - 1;
		int j = high + 1;

		while (true) {
			do {
				i++;
			} while (arr[i] < pivot);

			do {
				j--;
			} while (arr[j] > pivot);

			if (i >= j) {
				return j;
			}

			int temp = arr[i];
			arr[i] = arr[j];
			arr[j] = temp;
		}
	}
```

```
	public static void quickSort(int arr[], int low,  int high) {

		if(low<high) {
			int pivotIndex=hoaresPartition(arr, low, high);
			quickSort(arr, low, pivotIndex);
			quickSort(arr, pivotIndex+1, high);
		}

	}

	public static void main(String[] args) {

		int arr[] = {8,4,7,9,3,10,5};
		int low = 0;
		int high = arr.length-1;

		System.out.println(Arrays.toString(arr));
		quickSort(arr,low,high);
		System.out.println(Arrays.toString(arr));

	}
}
```

14.11. Exploring Cycle Sort

- Cycle Sort does the Sorting with minimum memory writes and can be usefull for cases where the memory writes are costly.
- Selection sort is an in-place sorting algorithm.
- Selection sort is not a Stable sorting algorithm.

Complexity of Cycle Sort:

Time Complexity - Best	$O(n^2)$
Time Complexity - Average	$O(n^2)$
Time Complexity - Worst	$O(n^2)$
Space Complexity	O(1)

14.11.1. Cycle Sort with Distinct elements

Lab23.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */
public class Lab23 {

	public static void cycleSort(int arr[]) {

		int n = arr.length;
		for (int cycle = 0; cycle < n - 1; cycle++) {
			int item = arr[cycle];
			int position = cycle;
			for (int j = cycle + 1; j < n; j++) {
				if (arr[j] < item)
					position++;
			}
```

```
                int temp = item;
                item = arr[position];
                arr[position] = temp;

                while (position != cycle) {
                        position = cycle;
                        for (int j = cycle + 1; j < n; j++) {
                                if (arr[j] < item)
                                        position++;
                        }

                         temp = item;
                        item = arr[position];
                        arr[position] = temp;
                }
        }
    }

    public static void main(String[] args) {

        int arr[] = { 20,40,50,10,30 };

        System.out.println(Arrays.toString(arr));
        cycleSort(arr);
        System.out.println(Arrays.toString(arr));

    }
}
```

14.12. Exploring Heap Sort

- Basic Idea of Heap Sort is based on Selection Sort
- Heap Sort is an Improvement of Selection Sort

In Selection Sort

a) We find the Max Element
b) We swap Max Elemen with Last element

- In Selection Sort, we do Linear Search to find the max element.
- That why Time Complexity with Selection Sort is $O(n^2)$

In Heap Sort

a) We build Max Heap
b) We swap Root with Last element - So the Max Element goes to End.
c) Reduce the Heap Size
d) Heapify the Array
e) Repeat Steps b,c,d

- Heap Sort does the Optimization over the selection sort. Uses the same Idea but for finding the Max Element, Uses Max Heap Data Structures instead of Linear Search
- That why Time Complexity with Heap Sort is O(n logn)

Complexity of Heap Sort:

Time Complexity - Best	O(nlogn)
Time Complexity - Average	O(nlogn)
Time Complexity - Worst	O(nlogn)
Space Complexity	O(1)

14.12.1. Heap Sort Implementation

Lab24.java

```
package com.jlcindia.sorting;

import java.util.Arrays;
/*
* @Author : Srinivas Dande
* @Company: Java Learning Center
* */

public class Lab24 {
	public static void heapify(int arr[], int n, int i) {

		int largest = i;
		int left = 2 * i + 1;
		int right = 2 * i + 2;

		if (left < n && arr[left] > arr[largest])
			largest = left;

		if (right < n && arr[right] > arr[largest])
			largest = right;

		if (largest != i) {
			int temp = arr[i];
			arr[i] = arr[largest];
			arr[largest] = temp;

			// Recursively heapify the affected sub-tree
			heapify(arr, n, largest);
		}
	}
```

```
	public static void buildHeap(int arr[]) {

		int n = arr.length;
		for (int i = n / 2 - 1; i >= 0; i--) {
			heapify(arr, n, i);
		}
	}

	public static void heapSort(int arr[]) {

		int n = arr.length;
		buildHeap(arr);
		for (int i = n - 1; i >= 0; i--) {
			int temp = arr[0];
			arr[0] = arr[i];
			arr[i] = temp;

			heapify(arr, i, 0);
		}
	}

	public static void main(String[] args) {

		int arr[] = { 10, 15, 50, 5, 20,60,2 };

		System.out.println(Arrays.toString(arr));
		heapSort(arr);
		System.out.println(Arrays.toString(arr));
	}
}
```