CPCS 223

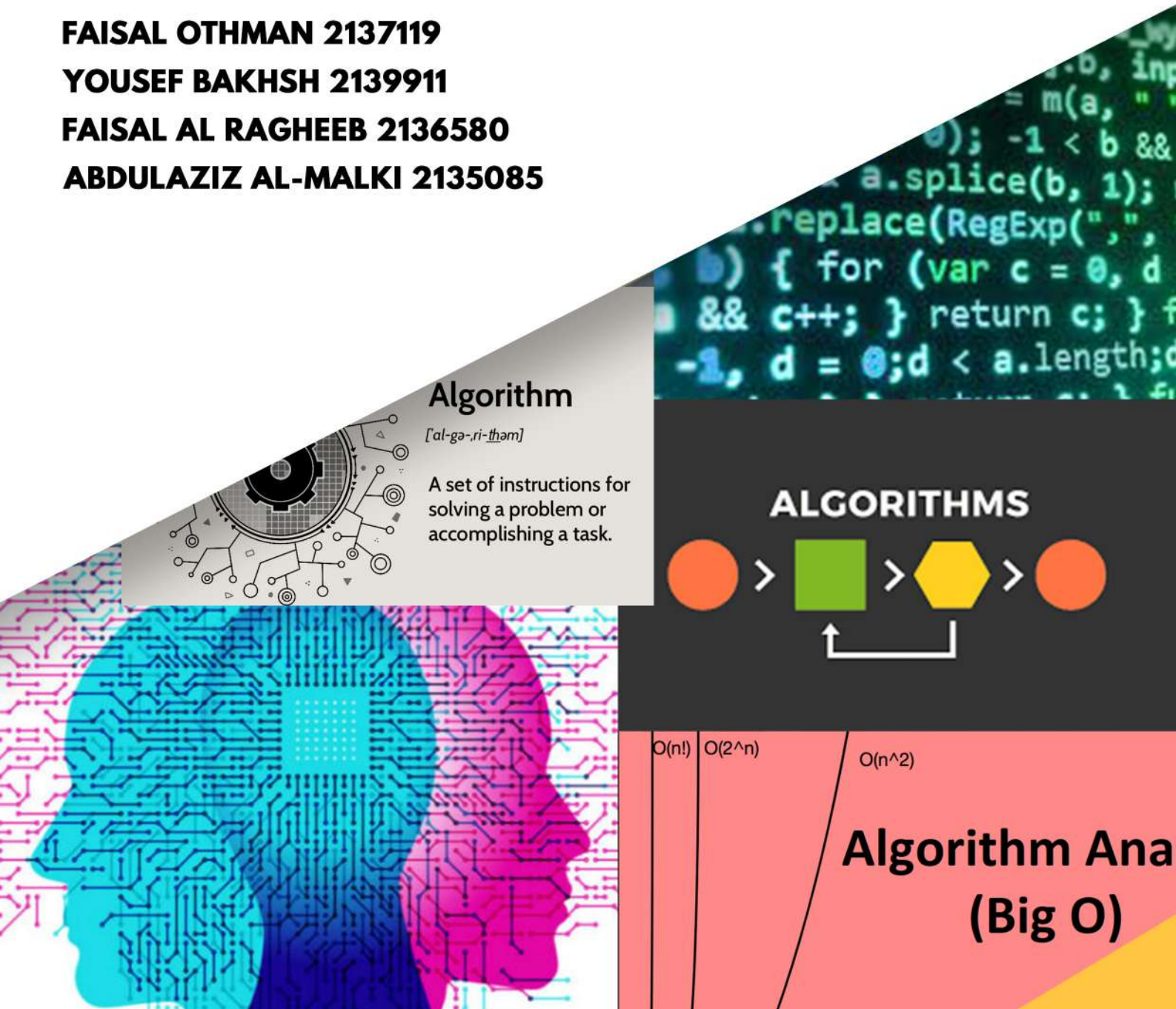
Theoretical and Empirical analysis study

FAISAL OTHMAN 2137119

YOUSEF BAKHSH 2139911

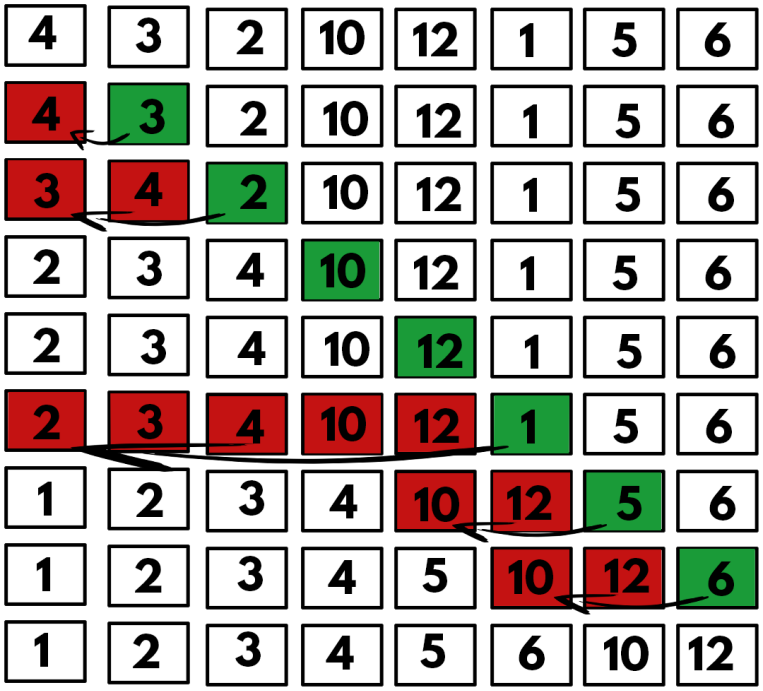
FAISAL AL RAGHEEB 2136580

ABDULAZIZ AL-MALKI 2135085



Insertion Sort

Insertion sort is a simple sorting algorithm that is similar to the way you sort playing cards in your hands. The array is split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.



Time complexities of Insertion sort:

Best case $\Omega(n)$:

- Best case of insertion sort is when the array is already sorted.

Average case $\Theta(n^2)$:

- Average case of insertion sort is when elements are in a random order (not sorted).

Worst case $O(n^2)$:

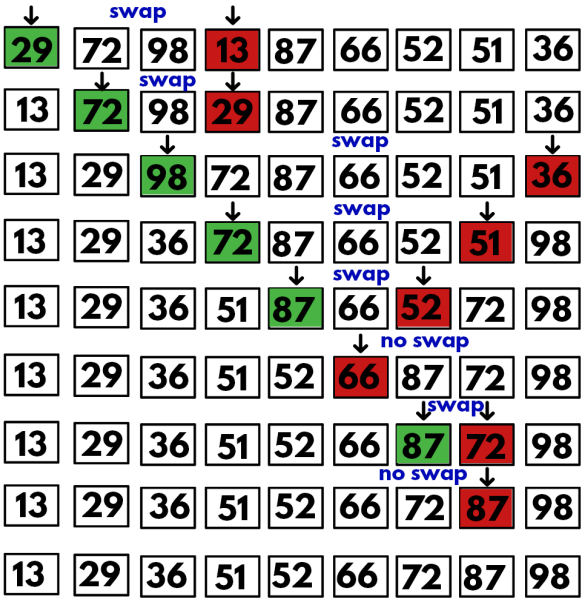
- Worst case of insertion sort is when elements are required to be sorted in descending order when it is in ascending order, and vice versa.

Insertion sort java code

```java
public static void insertionSort(int[] array) {
    // Start measuring execution time
    long startTime = System.nanoTime();


    // Print content of array
    // before sorting
    System.out.println("Array elements before sorting (Insertion Sort)");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();

    for (int i = 1; i < array.length; i++) {
        int current = array[i];
        int j = i-1;
        while ((j > -1) && (array[j] > current)) {
            array[j+1] = array[j];
            j--;
        }
        array[j+1] = current;
    }

    // Print content of array
    // after sorting
    System.out.println("Array elements after sorting");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();


    // End measuring execution time
    long stopTime = System.nanoTime();
    long executionTime = stopTime - startTime;
    System.out.println("\nExecution Time (Insertion Sort): " + executionTime + " ns");
}
```

Selection Sort

Selection sort is a straightforward and efficient sorting algorithm that works by continually picking the smallest (or biggest) element from the unsorted section of the list and shifting it to the sorted portion. The method takes the smallest (or biggest) element from the unsorted section of the list and swaps it with the first member of the unsorted portion on a regular basis. This step is continued for the remainder of the unsorted list until the entire list is sorted.



Time complexities of Selection sort

Best case $\Omega(n^2)$:

- Best case of selection sort is when the array is already sorted. In this case, swapping at each step is avoided, but the time spent to search for the smallest/biggest element is O(n).

Average case $\Theta(n^2)$:

- Average case of selection sort is when the array is not sorted in any specific order order (randomly ordered)

Worst case $O(n^2)$:

- Worst case of selection sort is when the array is already sorted (with one swap) but the smallest element is the last element.

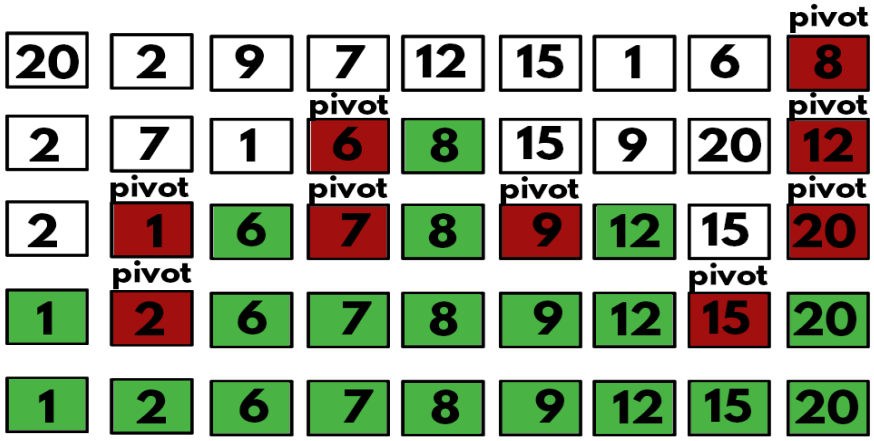
Selection sort java code

```java
public static void selectionSort(int[] array) {
    // Start measuring execution time
    long startTime = System.nanoTime();


    // Print content of array
    // before sorting
    System.out.println("\nArray elements before sorting (Selection Sort)");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();

    // traverse unsorted array
    for (int i = 0; i < array.length-1; i++) {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < array.length; j++)
            if (array[j] < array[min_idx])
                min_idx = j;
        // swap minimum element with compared element
        int temp = array[min_idx];
        array[min_idx] = array[i];
        array[i] = temp;

    }

    // Print content of array
    // after sorting
    System.out.println("Array elements after sorting");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();


    // End measuring execution time
    long stopTime = System.nanoTime();
    long executionTime = stopTime - startTime;
    System.out.println("\nExecution Time (Selection Sort): " + executionTime + " ns");
}
```

Quick Sort

QuickSort implements the Divide and Conquer concept. It selects one element as a pivot and splits the specified array around it. There are several quickSort variants that select pivot in various ways.

A partition is the main procedure of quickSort. Given an array and an array member x as the pivot, the goal of partitioning is to place x in its right position in a sorted array and to place all smaller items (less than x) before x and all bigger elements (greater than x) after x.



Time complexities of Quicksort:

Best case: Ω(nlogn)

- In Quicksort, the best case happens when the pivot is the middle element.

Average case : Θ(nlogn)

- it occurs when the array is mixed not ascending or descending.

Worst case: $O(n^2)$

- when the pivot is the smallest or largest element.

Quicksort java code

```java
public static void quickSort1(int[] array) {
    // Start measuring execution time
    long startTime = System.nanoTime();


    // Print content of array
    // before sorting
    System.out.println("\nArray elements before sorting (Quick Sort)");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();

    // call quick sort method
    quickSort(array, 0, array.length - 1);

    // Print content of array
    // after sorting
    System.out.println("Array elements after sorting");
    for(int i = 0; i < array.length; i++)
        System.out.print(array[i] + " ");
    System.out.println();


    // End measuring execution time
    long stopTime = System.nanoTime();
    long executionTime = stopTime - startTime;
    System.out.println("\nExecution Time (Quick Sort): " + executionTime + " ns");
}


public static void quickSort(int[] array, int begin, int end) {
    if (begin < end) {
        int partitionIndex = partition(array, begin, end);

        quickSort(array, begin, partitionIndex-1);
        quickSort(array, partitionIndex+1, end);
        }
}
```

Partition

```
public static int partition(int array[], int begin, int end) {
    int pivot = array[end];
    int i = (begin-1);

    for (int j = begin; j < end; j++) {
        if (array[j] <= pivot) {
            i++;
            int Temp = array[i];
            array[i] = array[j];
            array[j] = Temp;
    }
    }

    int Temp = array[i+1];
    array[i+1] = array[end];
    array[end] = Temp;

    return i+1;
    }
```

Comparison of all three sorting algorithms

Value of N	Run	Insertion Sort (Running time in Nanoseconds)	Selection Sort (Running time in Nanoseconds)	Quick Sort (Running time in Nanoseconds)
N=10	1	1500	2100	23700
	2	1500	2100	3600
	3	1900	2200	3400
	4	1700	2200	18900
	5	1600	2100	3600
Average		1,640 ns	2,140 ns	10,640 ns
N=100	1	36700	61600	20000
	2	39200	61300	19100
	3	46700	61600	20200
	4	58500	104400	65900
	5	39700	61900	20700
Average		44,160 ns	70,160 ns	29,180 ns
N=1000	1	2426700	3019700	239500
	2	1638800	2440400	265600
	3	1876800	2857900	249700
	4	1682400	2470800	531000
	5	1662900	2457800	260800
Average		1,857,520 ns	2,649,320 ns	309,320 ns

Value of N	Run	Insertion Sort (Running time in Nanoseconds)	Selection Sort (Running time in Nanoseconds)	Quick Sort (Running time in Nanoseconds)
N=10000	1	38933800	34698800	1407200
	2	40440900	35354800	1742300
	3	39787700	32323400	1446100
	4	29894500	35059300	1462300
	5	38773600	33295000	1726700
Average		37,566,100 ns	34,146,260 ns	1,556,920 ns
N=100000	1	2515884700	2513278900	10188900
	2	2492070700	2486668800	9979000
	3	2463104500	2466675700	10614100
	4	2459265300	2474010600	12652100
	5	2456755900	2472468400	9582500
Average		2,477,416,220 ns	2,482,620,480 ns	10,603,320 ns
Average Time		503,377,128 ns	503,897,672 ns	2,501,876 ns

Conclusion

To conclude, Quick Sort is the fastest sorting algorithm out of 3 given algorithms. However, after analyzing execution times of Quick Sort, we noticed that Quick Sort takes longer than Selection and Insertion sort algorithms to sort arrays that are small in size ($n \leq 10$). We also noticed that as the input size increases, Selection sort and Insertion sort execution time grew significantly larger than that of Quick sort.

Data analysis

n = 10

- Insertion sort: **1,640 ns**
- Selection sort: **2,140 ns**
- Quick sort: **10,640 ns**

n = 1000

- Insertion sort: **1,857,520 ns**
- Selection sort: **2,649,320 ns**
- Quick sort: **309,320 ns**

n = 100000

- Insertion sort: **2,477,416,220 ns**
- Selection sort: **2,482,620,480 ns**
- Quick sort: **10,603,320 ns**

The data above shows that Quicksort was the fastest sorting algorithm for larger input sizes while Insertion and Selection sort were faster for input sizes of $n \leq 10$. But as the input size increased, Quick sort appeared to be the fastest sorting algorithm out of the other two.

Algorithm Analysis

Analysis of algorithms must be independent of the machine, operating system, programming language, compiler, etc. The data recorded during these trials all depend on the machine we used, the operating systems, the programming language and the compiler.

The machine we used is a gaming laptop. Gaming laptops tend to have more powerful components than normal laptops. Those components include the processor, memory, and many more components.

The programming language that we used is Java. Java is considered to be very slow compared to other languages such as C++ and C. This is because Java runs on a virtual machine unlike C++ and C, which runs on the native machine. This makes languages like C++ and C much faster than Java.