Universidad ICESI

Sean Quintero - Bryan Guapacha

Paola Osorio - John Kennedy

Análisis de complejidad temporal de algoritmos de ordenamiento.

Insertion sort:

```
public static void insertionSortImperative(int[] input) {
        for (int i = 1; i < input.length; i++) {
            int key = input[i];
            int j = i - 1;
            while (j >= 0 && input[j] > key) {
                input[j + 1] = input[j];
                j = j - 1;
            }
            input[j + 1] = key;
        }
    }
```

Instrucción	Veces que se repite (Big O)
1. for (int i = 1; i < input.length; i++) {	n
2. int key = input[i];	n-1
3. int j = i - 1;	n-1
4. while (j >= 0 && input[j] > key) {	(n*(n-1))/2
5. input[j + 1] = input[j];	((n*(n-1))/2)-1
6. j = j - 1;	((n*(n-1))/2)-1
7. input[j + 1] = key;	n-1
Total:	n^2

Radix sort:

```
private static int findMaximumNumberIn(int[] arr) {

        return Arrays.stream(arr).max().getAsInt();
```

Instrucción	Veces que se repite (Big O)
1.return Arrays.stream(arr).max().getAsInt();	1
Total:	O(1)

```
private static int calculateNumberOfDigitsIn(int number) {
    return (int) Math.log10(number) + 1; // valid only if number > 0
```

Instrucción	**Veces que se repite (Big O)**
1. return (int) Math.log10(number) + 1; // valid only if number > 0	1
Total:	O(1)

```
private static void applyCountingSortOn(int[] numbers, int placeValue) {
    int range = 10; // radix or the base

    int length = numbers.length;
    int[] frequency = new int[range];
    int[] sortedValues = new int[length];

    for (int i = 0; i < length; i++) {
      int digit = (numbers[i] / placeValue) % range;
      frequency[digit]++;
    }

    for (int i = 1; i < range; i++) {
      frequency[i] += frequency[i - 1];
    }

    for (int i = length - 1; i >= 0; i--) {
      int digit = (numbers[i] / placeValue) % range;
      sortedValues[frequency[digit] - 1] = numbers[i];
      frequency[digit]--;
    }
    System.arraycopy(sortedValues, 0, numbers, 0, length);
}
```

Instrucción	**Veces que se repite (Big O)**
1. int range = 10; // radix or the base	1
2. int length = numbers.length;	1
3. int[] frequency = new int[range];	1
4. int[] sortedValues = new int[length];	1
5. for (int i = 0; i < length; i++) {	m+1
6. int digit = (numbers[i] / placeValue) % range;	m
7. frequency[digit]++;	m
8. for (int i = 1; i < range; i++) {	m+1
9. frequency[i] += frequency[i - 1];	m
10. for (int i = length - 1; i >= 0; i--) {	m+ 1
11. int digit = (numbers[i] / placeValue) % range;	m

```
   }
```

12. sortedValues[frequency[digit] - 1] = numbers[i];	m
13. frequency[digit]--;	m
14. System.arraycopy(sortedValues, 0, numbers, 0, length);	1
Total:	O(m)

```
public static void radixSort(int numbers[]) {
        int maximumNumber = findMaximumNumberIn(numbers);

        int numberOfDigits = calculateNumberOfDigitsIn(maximumNumber);

        int placeValue = 1;

        while (numberOfDigits-- > 0) {
            applyCountingSortOn(numbers, placeValue);
            placeValue *= 10;
        }
    }
```

Instrucción	Veces que se repite (Big O)
1. int maximumNumber = findMaximumNumberIn(numbers);	1
2. Int numberOfDigits = calculateNumberOfDigitsIn(maximumNumber);	1
3. int placeValue = 1;	1
4. while (numberOfDigits-- > 0) {	n + 1
5. applyCountingSortOn(numbers, placeValue);	n * O(m)
6. placeValue *= 10;	n
Total:	O(mn)