**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 |
| 1. int key = input[i]; | n-1 |
| 1. int j = i - 1; | n-1 |
| 1. while (j >= 0 && input[j] > key) { | (n\*(n-1))/2 |
| 1. input[j + 1] = input[j]; | ((n\*(n-1))/2)-1 |
| 1. j = j - 1; | ((n\*(n-1))/2)-1 |
| 1. input[j + 1] = key; | n-1 |
| **Total:** | n² |

**Merge sort:**

public static void mergeSort(int[] a, int n) {

if (n < 2) {

return;

}

int mid = n / 2;

int[] l = new int[mid];

int[] r = new int[n - mid];

for (int i = 0; i < mid; i++) {

l[i] = a[i];

}

for (int i = mid; i < n; i++) {

r[i - mid] = a[i];

}

*mergeSort*(l, mid);

*mergeSort*(r, n - mid);

*merge*(a, l, r, mid, n - mid);

}

public static void merge(int[] a, int[] l, int[] r, int left, int right) {

int i = 0, j = 0, k = 0;

while (i < left && j < right) {

if (l[i] <= r[j]) {

a[k++] = l[i++];

} else {

a[k++] = r[j++];

}

}

while (i < left) {

a[k++] = l[i++];

}

while (j < right) {

a[k++] = r[j++];

}

}

|  |  |
| --- | --- |
| **Instrucción** | **Veces que se repite (Big O)** |
| 1. if (n < 2) { | 1 |
| 1. return; | 1 |
| 1. int mid = n / 2; | 1 |
| 1. int[] l = new int[mid]; | 1 |
| 1. int[] r = new int[n - mid]; | 1 |
| 1. for (int i = 0; i < mid; i++) { | n + 1 |
| 1. l[i] = a[i]; | n |
| 1. for (int i = mid; i < n; i++) { | n + 1 |
| 1. r[i - mid] = a[i]; | n |
| 1. *mergeSort*(l, mid); | log2 n |
| 1. *mergeSort*(r, n - mid); | log2 n |
| 1. *merge*(a, l, r, mid, n - mid); | O(n) |
| **Total:** | O(nlogn) |

|  |  |
| --- | --- |
| **Instrucción** | **Veces que se repite (Big O)** |
| 1. int i = 0, j = 0, k = 0; | 1 |
| 1. while (i < left && j < right) { | n+ 1 |
| 1. if (l[i] <= r[j]) { | n |
| 1. a[k++] = l[i++]; | n |
| 1. while (i < left) { | n+ 1 |
| 1. a[k++] = l[i++]; | n |
| 1. while (j < right) { | n+ 1 |
| 1. a[k++] = r[j++]; | n |
| **Total:** | O(n) |

**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 |
| 1. int length = numbers.length; | 1 |
| 1. int[] frequency = new int[range]; | 1 |
| 1. int[] sortedValues = new int[length]; | 1 |
| 1. for (int i = 0; i < length; i++) { | m+1 |
| 1. int digit = (numbers[i] / placeValue) % range; | m |
| 1. frequency[digit]++; | m |
| 1. for (int i = 1; i < range; i++) { | m+1 |
| 1. frequency[i] += frequency[i - 1]; | m |
| 1. for (int i = length - 1; i >= 0; i--) { | m+ 1 |
| 1. int digit = (numbers[i] / placeValue) % range; | m |
| 1. sortedValues[frequency[digit] - 1] = numbers[i]; | m |
| 1. frequency[digit]--; | m |
| 1. 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 |
| 1. Int numberOfDigits = calculateNumberOfDigitsIn(maximumNumber); | 1 |
| 1. int placeValue = 1; | 1 |
| 1. while (numberOfDigits-- > 0) { | n + 1 |
| 1. applyCountingSortOn(numbers, placeValue); | n \* O(m) |
| 1. placeValue \*= 10; | n |
|  |  |
| **Total:** | O(mn) |

}