Analisis de Algoritmos - Algoritmos de Ordenamiento

Rodolfo Armando Jaramillo Ruiz
22 de Febrero de 2023

Preambulo

Para todas las pruebas use diez listas con las siguientes cantidades de elementos:

$$[10, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000]$$

Es decir, una lista con 10 elementos, luego una lista con 100 elementos, etc. Cree una función en c++ que produce diez listas cada una con estas cantidades de elementos, para cada uno de los tres casos: el mejor, el peor y el promedio en función de una variable *char* que representa si la lista será ascendente, descendente, o aleatoria.

```
vector<int> lists(char caseType, int n) {
    vector<int> arr(n);
    switch (caseType) {
        case 'A':
            for (int i = 1; i <= n; i++) {
                arr[i - 1] = i;
            }
            break;
        case 'D':
            for (int i = n; i >= 1; i--) {
                arr[n - i] = i;
            }
            break;
        case 'R':
            srand(time(NULL));
            for (int i = 0; i < n; i++) {
                arr[i] = rand() % 100;
            }
            break;
    }
    return arr;
}
```

La función principal es la siguiente junto con las librerías usadas:

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <vector>
#include <string>
#include <iomanip>

using namespace std;

int main() {
    ofstream file;
    char type;
    string fileName;
    for (int n : {10, 100, 250, 500, 750, 1000, 2500, 5000, 7500, 10000}) {
        for (char caseType : {'A', 'D', 'R'}) {
            vector<int> arr = lists(caseType, n);
            auto start = chrono::high_resolution_clock::now();
            arr = Sort(arr, n); //Algoritmo de ordenamiento
            auto end = chrono::high_resolution_clock::now();
            auto duration = chrono::duration_cast<chrono::duration<double, std::milli>>(
    end - start);

            switch (caseType) {
                case 'A':
                    fileName = "insertion-best.csv";
                    break;
                case 'D':
                    fileName = "insertion-worst.csv";
                    break;
```

```
                case 'R':
                    fileName = "insertion-mean.csv";
                    break;
                default:
                    break;
            }

            file.open(fileName, ios::app);
            file << n << "," << fixed << setprecision(4) << duration.count() << endl;
            file.close();

        }
    }
    return 0;
}
```

Al final de obtienen tres archivos *csv*, uno para cada caso, donde la primer columna es el tamaño de la lista a ordenas y la segunda el tiempo (en milisegundos con cuatro decimales de precisión) que tardó la función en ordenarla. Estos archivos se procesa para obtener las respectivas gráficas para cada caso.

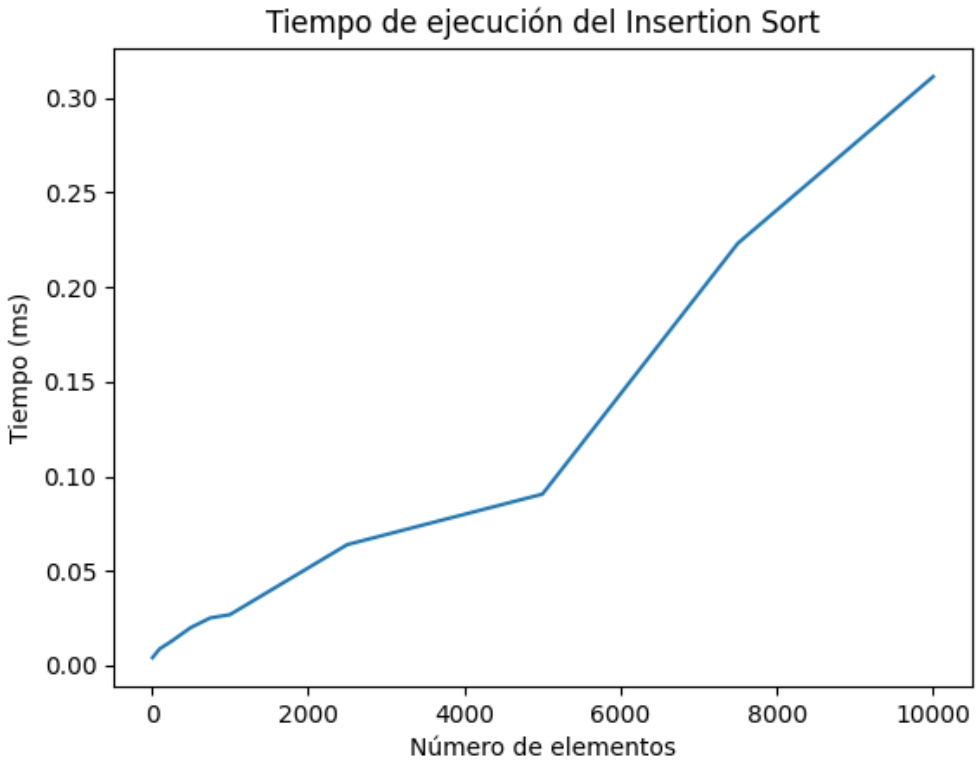
Insertion Sort

El código implementado en c++ para el *insertion sort* fue el siguiente.

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

Mejor de los casos:

Elementos	Tiempo (ms)
10.0	0.0042
100.0	0.0088
250.0	0.0128
500.0	0.0201
750.0	0.0252
1000.0	0.0269
2500.0	0.0639
5000.0	0.0906
7500.0	0.2231
10000.0	0.3112

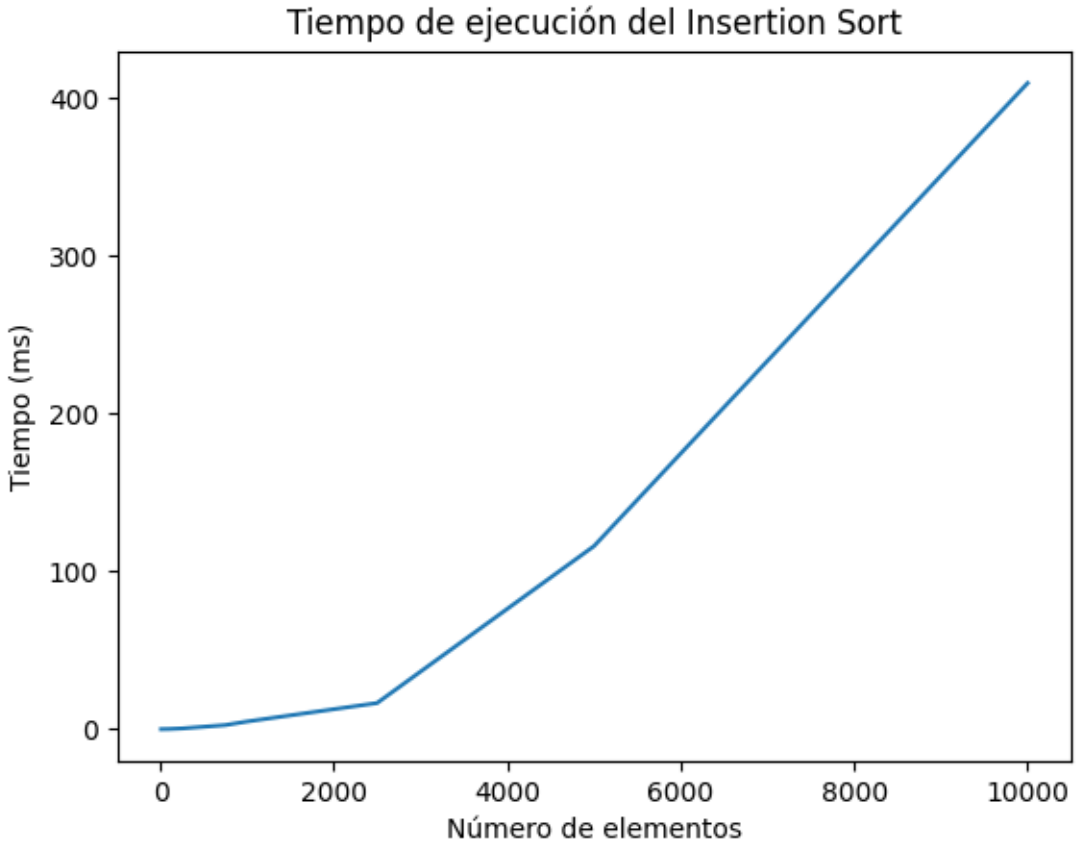


Peor de los casos:

Elementos	Tiempo (ms)
10.0	0.0062
100.0	0.1407
250.0	0.7957
500.0	3.347
750.0	4.7632
1000.0	9.5901
2500.0	48.8384
5000.0	166.7427
7500.0	355.3732
10000.0	808.4235

Caso promedio:

Elementos	Tiempo (ms)
10.0	0.0053
100.0	0.102
250.0	0.4539
500.0	1.5203
750.0	2.5767
1000.0	4.8427
2500.0	16.5499
5000.0	115.9915
7500.0	262.7784
10000.0	409.5359



Selection Sort

El código del *selection sort* es el siguiente:

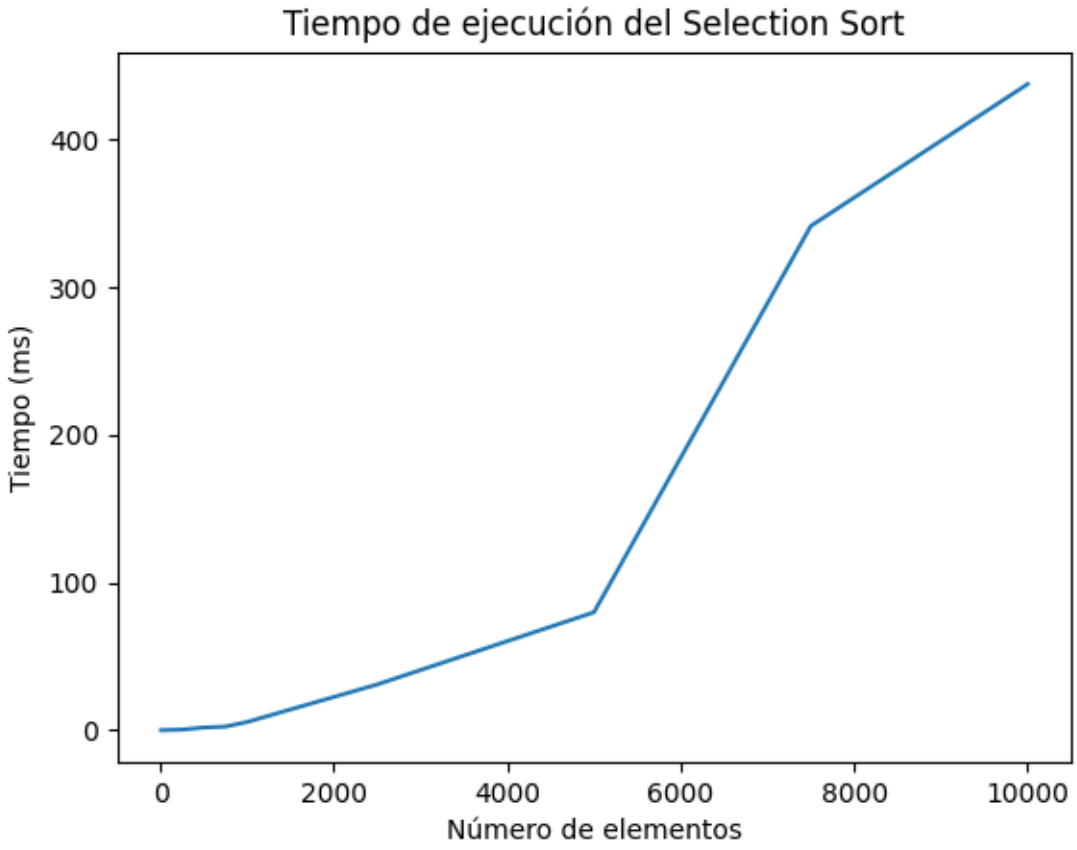
```
vector<int> selectionSort(vector<int> arr, int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        swap(arr[min_idx], arr[i]);
    }
    return arr;
}
```

Mejor de los casos:

Elementos	Tiempo (ms)
10.0	0.0054
100.0	0.1076
250.0	0.3198
500.0	1.318
750.0	3.434
1000.0	7.665
2500.0	32.2334
5000.0	95.3243
7500.0	141.6628
10000.0	312.4596

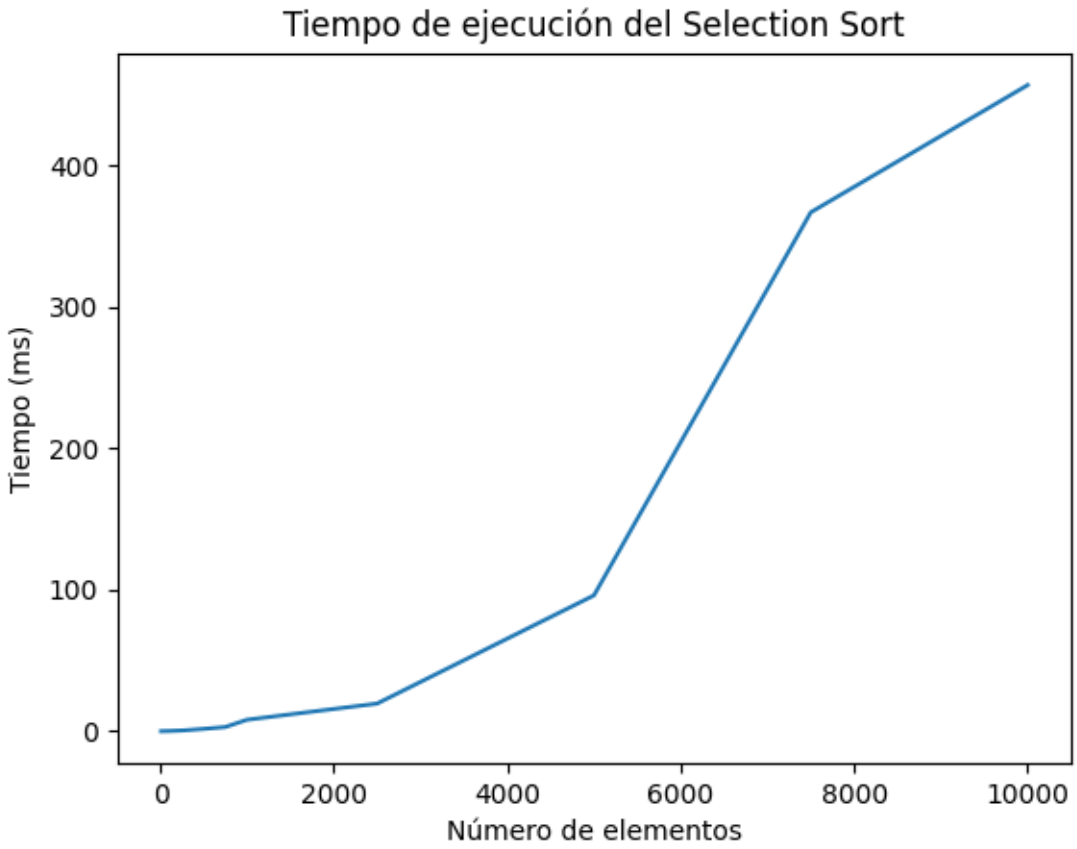
Peor de los casos:

Elementos	Tiempo (ms)
10.0	0.0045
100.0	0.1236
250.0	0.3599
500.0	1.7597
750.0	2.3899
1000.0	5.4949
2500.0	30.9228
5000.0	79.8443
7500.0	341.5719
10000.0	437.7391



Caso promedio:

Elementos	Tiempo (ms)
10.0	0.0053
100.0	0.1257
250.0	0.4704
500.0	1.603
750.0	2.9108
1000.0	8.0599
2500.0	19.4927
5000.0	95.977
7500.0	366.8033
10000.0	456.7511


Tiempo de ejecución del Selection Sort
Tiempo (ms)
0
100
200
300
400
0
2000
4000
6000
8000
10000
Número de elementos

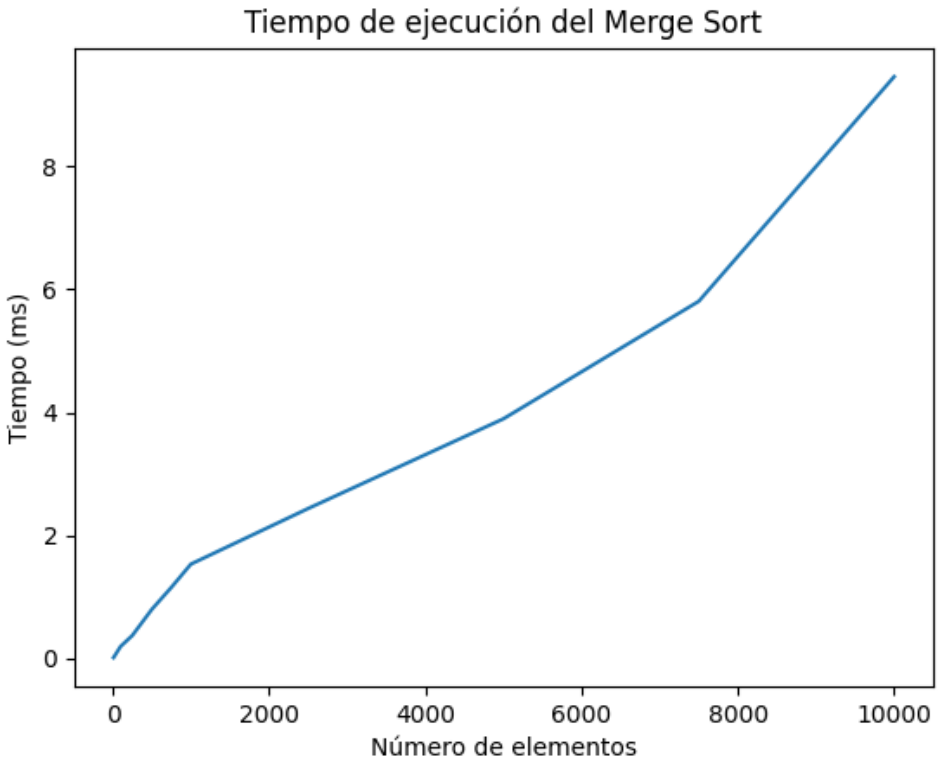
Merge Sort

El código del *merge sort* es el siguiente:

```
void merge(vector<int> &arr, int left, int middle, int right) {
    int i, j, k;
    int n1 = middle - left + 1;
    int n2 = right - middle;

    vector<int> leftArr(n1);
    vector<int> rightArr(n2);

    for (i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (j = 0; j < n2; j++) {
        rightArr[j] = arr[middle + 1 + j];
    }

    i = 0;
    j = 0;
    k = left;

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}

void mergeSort(vector<int> &arr, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSort(arr, left, middle);
        mergeSort(arr, middle + 1, right);

        merge(arr, left, middle, right);
    }
}
```

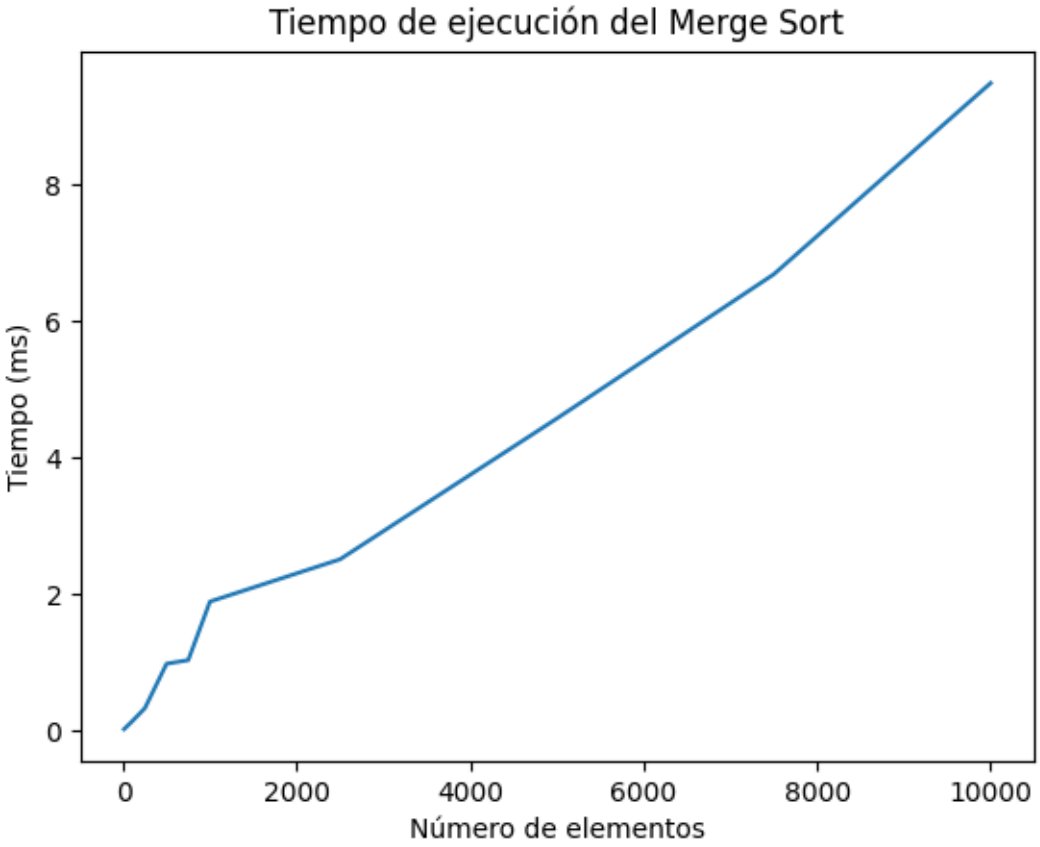
Mejor de los casos:

Elementos	Tiempo (ms)
10.0	0.0146
100.0	0.196
250.0	0.374
500.0	0.799
750.0	1.1566
1000.0	1.5346
2500.0	2.438
5000.0	3.8967
7500.0	5.8069
10000.0	9.4576



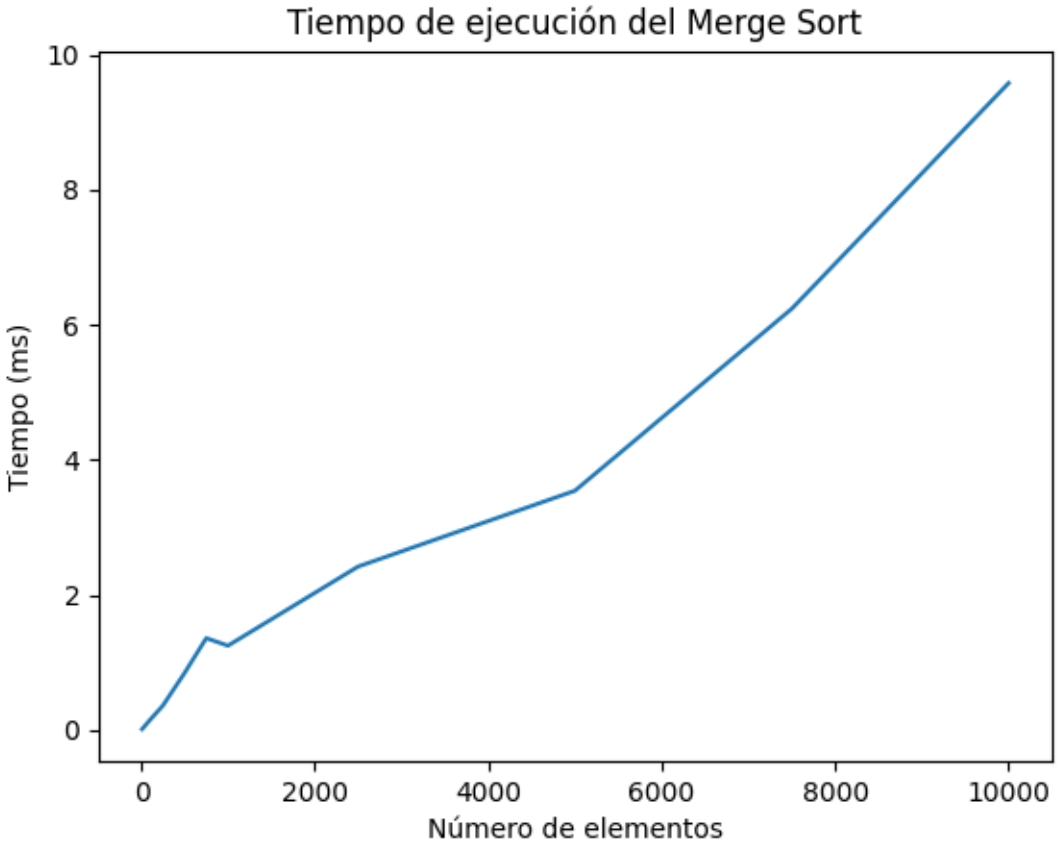
Peor de los casos:

Elementos	Tiempo (ms)
10.0	0.0287
100.0	0.1394
250.0	0.3363
500.0	0.9868
750.0	1.04
1000.0	1.8973
2500.0	2.5168
5000.0	4.5799
7500.0	6.6916
10000.0	9.4922



Caso promedio:

Elementos	Tiempo (ms)
10.0	0.0146
100.0	0.1487
250.0	0.3637
500.0	0.8465
750.0	1.3614
1000.0	1.2497
2500.0	2.4205
5000.0	3.5432
7500.0	6.2368
10000.0	9.5768


Tiempo de ejecución del Merge Sort
Tiempo (ms)
Número de elementos
0
2
4
6
8
10
0
2000
4000
6000
8000
10000

T(n) del Selection Sort

El algoritmo del *Selection Sort* es el siguiente

```
for i <- 1 to n-1 do
  min j <- i;
  min x <- A[i]
  for j <- i+1 to n do
    if A[j] < min x then
      min j <- i
      min x <- A[j]
  A[min j] <- A[i]
  A[i] <- min x
```

El algoritmo ejecuta dos bucles aninados, el primero o exterior se ejecuta n veces mientras que el interior se ejecuta $n-1$ en la primera iteración del bucle exterior, reduciéndose en uno en cada iteración de este. Así hasta que solo le queda un elemento por ordenar. La función de coste de esta forma se puede expresar como

$$T(n) = c_1(n-1) + c_2(n-2) + \cdots + c_{n-1}$$

Donde c_i representa el costo de la operación en la i-ésima iteración del bucle interior, con n la longitud del la lista. La operación más costosa en el Selection Sort es encontrar el mínimo elemento en la lista, que se realiza en cada iteración del bucle interior. Esta operación tiene un costo de n comparaciones. Por lo tanto, podemos expresar el costo total del Selection Sort como:

$$T(n) = n - 1 + 2(n-2) + 3(n-3) + ... + (n-1)$$

Que es pues la serie aritmética que se expresa como:

$$T(n) = \frac{n^2}{2} + \frac{n}{2}$$

Funciones f(n)

Las gráficas de las funciones

$$f(n) = log(n) \quad f(n) = n \quad f(n) = nlog(n)$$

$$f(n) = n^2 \quad f(n) = n^3$$