# Merge vs Quick Algorithms

Max Maldonado
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv16a.msolano@uartesdidgitales.edu.mx

Profesor: Efraín Padilla

Mayo 09, 2019

## I. Definición del Problema

En éste ejercicio comparamos dos algoritmos de ordenamiento de valores enteros. Los algoritmos que se estudian en este documento son Bubble e Insertion. El problema reside en la necesidad de observar el rendemiento de ambos algoritmos en tres casos específicos: el mejor caso, el peor caso y el caso promedio. El mejor caso radica en introducir un vector ya ordenado dentro del algoritmo. El peor caso radica en introducir un vector ordenado de forma inversa dentro del algoritmo. El caso promedio radica en introducir un vector de orden aleatorio dentro del algoritmo.

## II. Inputs y Outputs

| | |
|---|---|
| Input | Lista de N números enteros cuyo orden es desconocido. |
| Output | Lista de N números enteros ordenados de ascendentemente. |

## III. Solución

### A. Merge Sorting

Este algoritmo genera un arbol binario con los elementos de la lista. Una vez se ha llegado al final de las ramas, realiza operaciones de comparación y ”une” las hojas, de allí su nombre.

### B. Quick Sorting

Este algoritmo sigué la filosofía de ”Divide and Conquer”. Es una algoritmo que realiza comparaciones a partir partir los elementos de una lista en dos, recursivamente.

## IV. Best / Worst Complexity

### A. Best

El mejor escenario consiste en una cadena de números que ya se encuentran ordenados (Ascendente).

### B. Worst

El peor escenario consiste en una cadena de números que está ordenada de manera opuesta (Descendente).

## V. Code

### A. Merge Sorting

```
void
mergeSortA(std::vector<int> & _vector, int & _low, int & _midle, int & _high)
{

  int left_size = _midle - _low + 1;

  int i, j, k;
```

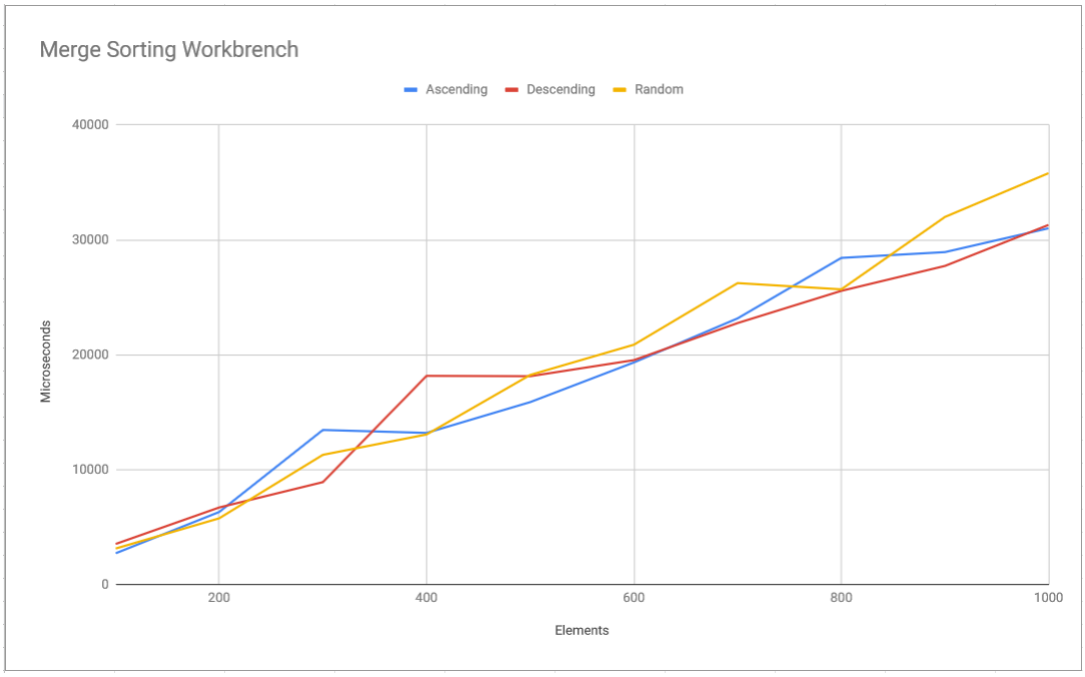
```
  int right_size = _high - _midle;

  std::vector<int> v_left;
  std::vector<int> v_right;

  for (int index = 0; index < left_size; ++index)
  {
    v_left.push_back(_vector[_low + index]);
  }

  for (int index = 0; index < right_size; ++index)
  {
    v_right.push_back(_vector[_midle + index + 1]);
  }

  i = 0;
  j = 0;

  for (k = _low; i < left_size && j < right_size; ++k)
  {
    if (v_left[i] < v_right[j])
    {
      _vector[k] = v_left[i++];
    }
    else
    {
      _vector[k] = v_right[j++];
    }
  }

  while (i < left_size)
  {
    _vector[k++] = v_left[i++];
  }

  while (j < right_size)
  {
    _vector[k++] = v_right[j++];
  }

  return;
}

void
mergeSort(std::vector<int> & _vector, int & _low, int & _high)
{
  int middle;

  if (_low < _high)
  {
    middle = (_low + _high) / 2;

    mergeSort(_vector, _low, middle);

    int middle_plus = middle + 1;
    mergeSort(_vector, middle_plus, _high);

    mergeSortA(_vector, _low, middle, _high);
```

```
  }

  return;
}
```

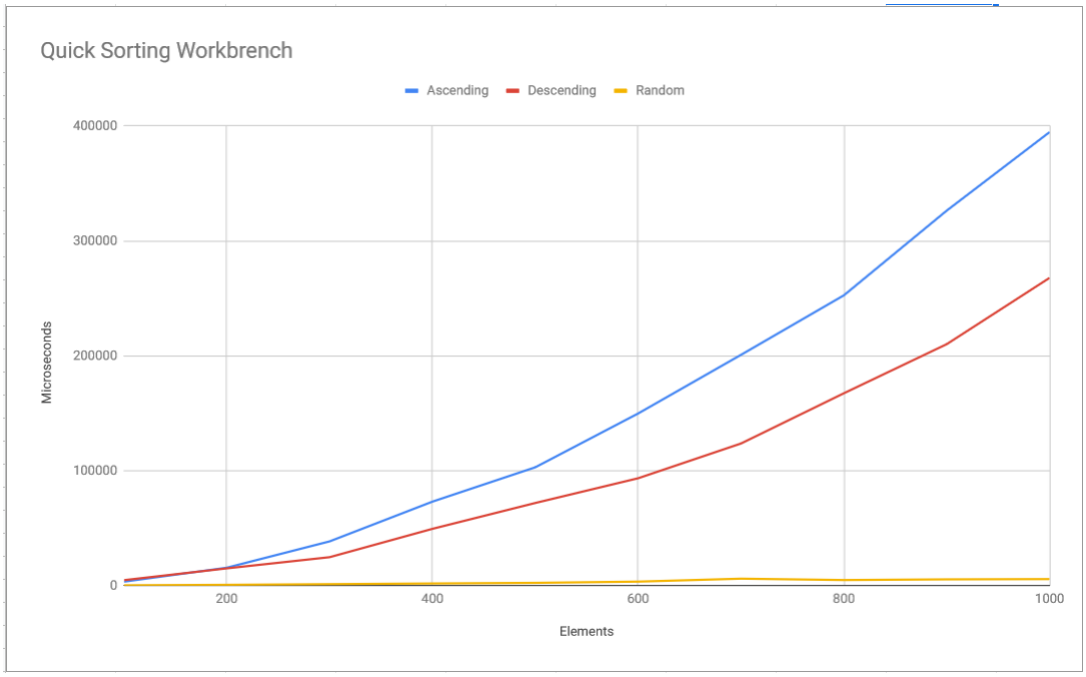
### B. Quick Sorting

```
int
partition(std::vector<int> & _vector, const int & _min, const int & _max)
{
  int pivot = _vector[_max];

  int i = (_min - 1);
  for (int j = _min; j < _max; j++)
  {
    if (_vector[j] < pivot)
    {
      i++;
      std::swap(_vector.at(i), _vector.at(j));
    }
  }

  std::swap(_vector.at(i + 1), _vector.at(_max));
  return i + 1;
}

void
quickAscSorting(std::vector<int> & _vector, const int & _min, const int & _max)
{
  if (_min < _max)
  {
    int pi = partition(_vector, _min, _max);

    quickAscSorting(_vector, _min, pi - 1);
    quickAscSorting(_vector, pi + 1, _max);
  }
}
```

## VI. BrenchMark


Merge Sorting Workbrench
Ascending
Descending
Random
40000
30000
20000
10000
0
Microseconds
200
400
600
800
1000
Elements

| Merge Sorting | | | |
|---|---|---|---|
| Elements | Best | Worst | Average |
| 100 | 2732 | 3544 | 3147 |
| 200 | 6331 | 6723 | 5781 |
| 300 | 13468 | 8929 | 11304 |
| 400 | 13211 | 18175 | 13075 |
| 500 | 15892 | 18145 | 18256 |
| 600 | 19343 | 19544 | 20887 |
| 700 | 23177 | 22770 | 26247 |
| 800 | 28440 | 25571 | 25703 |
| 900 | 28943 | 27741 | 32002 |
| 1000 | 31018 | 31313 | 35814 |


Quick Sorting Workbrench
Ascending
Descending
Random
400000
300000
200000
100000
0
Microseconds
200
400
600
800
1000
Elements

| Quick Sorting | | | |
|---|---|---|---|
| Elements | Best | Worst | Average |
| 100 | 3533 | 4863 | 384 |
| 200 | 15692 | 15118 | 814 |
| 300 | 38607 | 24855 | 1391 |
| 400 | 73198 | 49533 | 1980 |
| 500 | 103082 | 72022 | 2522 |
| 600 | 149914 | 93546 | 3597 |
| 700 | 200949 | 123787 | 6156 |
| 800 | 252736 | 167501 | 4966 |
| 900 | 326458 | 210232 | 5558 |
| 1000 | 394762 | 267928 | 5788 |