Sorting Algorithms

Santiago Toll Leyva
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv16a.stoll@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 14, 2019

Tarea 2: Algoritmos de ordenamiento

Teoria

Los algoritmos de ordenamiento tienen la funcion de acomodar una serie de elementos en un orden dado. Estos algoritmos necesitan una entrada; la funcion de estos algoritmos es regresar una salida que contenga los elementos de la entrada organizados en base al orden especificado.

Planteamiento del problema

Ordenar una serie de elementos puede llegar a ser un proceso tardado. Dependiendo de la magnitud de la entrada y la cantidad de elementos que deberan ser ordenados estos algoritmos aumentan su costo.

Solucion del problema

Ya que estos algoritmos siempre tendran un costo se decidio evaluar cinco de estos utilizando tres tipos distintos de entrada. Cada entrada es un vector que contiene 500 elementos. El primer vector contiene todos los elementos ordenados de manera ascendente; el segundo vector contiene todos los elementos ordenados de manera descendente; el tercer vector contiene elementos generados de manera aleatoria.

Asi mismo se utilizaron cinco diferentes algoritmos de ordenamiento para evaluar cual es el mas eficiente en cada caso. Los algoritmos utilizados son:

- Bubble sort
- Quick sort
- Heap sort
- Insertion sort
- Merge Sort

Para evaluar el tiempo que tarda cada algoritmo en ordenar estos vectores se itero diez veces por cada elemento para poder obtener un promedio del tiempo en ejecucion. Este proceso se repite varias veces utilizando vectores de distintos tamaños(vacio hasta 500).

Para no realizar esto de manera manual, se creo una funcion de benchmarking la cual hace exactamente lo descrito en el parrafo anterior.

Codigo

Funciones utilizadas para generar vectores.

Listing 1. Vector Generation Functions C++

```cpp
/*****************************************************************/
/* Vector Generator functions                                    */
/*****************************************************************/
vector<int> ascendingVector(int size) {
  vector<int> returnVector;
  for (int i = 0; i < size; i++) {
    returnVector.push_back(i);
  }
  return returnVector;
}

vector<int> descendingingVector(int size) {
  vector<int> returnVector;
  for (int i = size; i > 0; i--) {
    returnVector.push_back(i);
  }
  return returnVector;
}

vector<int> randomVector(int size,int min, int max) {
  vector<int> returnVector;
  std::srand(std::time(0));
  for (int i = 0; i < size; i++) {
    int random =  min + rand() % ((max + 1)-min); //Generate a random numbver between min and

    returnVector.push_back(random);
  }
  return returnVector;
}
```

Algoritmo Bubble sort.

Listing 2. Bubble Sort Algorithm C++

```cpp
/*****************************************************************/
/* Bubble Sort functions                                         */
/*****************************************************************/
void bubbleSort(vector<int>& vector) {
  if (vector.size() < 2) {
    return;
  }
  int size = vector.size(); //Save the vector size to make this efficient.
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size-1; j++) {
      if (vector[j] > vector[j + 1]) {
        std::swap(vector[j], vector[j + 1]); // Swap the positions.
      }
    }
  }
}
```

Algoritmo Quick sort.

Listing 3. Quick Sort Algorithm C++

```
/**********************************************************************/
/* Quick Sort functions                                               */
/**********************************************************************/
int partition(vector<int>& vector, int low, int high) {
  int pivot = vector[high]; // Find the pivot for this iteration.
  int i = low - 1;
  for (int j = low; j < high; j++) {
    if (vector[j] < pivot) {
      i++;
      std::swap(vector[i], vector[j]); // Swap the positions
    }
  }
  std::swap(vector[i + 1], vector[high]); // Swap the positions
  return(i + 1);
}

void quickSort(vector<int>& vector, int low, int high) {
  if (vector.size() < 2) {
    return;
  }
  if (low < high) {
    int partitionIndex = partition(vector, low, high);
    //Recursivity to travel the binary tree
    quickSort(vector, low, partitionIndex - 1);
    quickSort(vector, partitionIndex + 1, high);
  }
}
void quickSortCall(vector<int>& vector) {
  quickSort(vector, 0, vector.size() - 1);
}
```

Algoritmo Heap sort.

Listing 4. Heap Sort Algorithm C++

```cpp
/*********************************************************************/
/* Heap sort functions                                               */
/*********************************************************************/
void heapify(vector<int>& vector, int root, int size) {
  int largest = root;
  int leftChild = (2 * root) + 1;
  int rightChild = (2 * root) + 2;
  //Check if children are smaller than root
  //Left child
  if (leftChild < size && vector[leftChild] > vector[largest]) {
    largest = leftChild;
  }
  //Right child
  if (rightChild < size && vector[rightChild] > vector[largest]) {
    largest = rightChild;
  }
  //If root is not the largest
  if (largest != root) {
    std::swap(vector[root], vector[largest]);
    //Make this recursive
    heapify(vector, largest, size);
  }
}

void heapSort(vector<int>& vector) {
  if (vector.size() < 2) {
    return;
  }
  int size = vector.size();
  //Rearrange array
  for (int i = size / 2 - 1; i >= 0; i--) {
    heapify(vector, i, size);
  }
  //Extract element from heap
  for (int i = size-1;i>=0;i--)
  {
  std::swap(vector[0], vector[i]);
  heapify(vector, 0, i);
  }
}
```

Algoritmo Insertion sort.

Listing 5. Insertion Sort Algorithm C++

```cpp
/*******************************************************************/
/* Insertion sort                                                   */
/*******************************************************************/
void insertionSort(vector<int>& vector) {
  if (vector.size() < 2) {
    return;
  }
  int size = vector.size();
  //Number that is being looked for
  int key;
  //Current index being checked
  int current;
  for (int i = 1; i < size; i++) {
    //Assign key
    key = vector[i];
    //Set current index
    current = i - 1;
    //Move elements
    while (current >= 0 && vector[current] >key) {
      vector[current + 1] = vector[current];
      current = current - 1;
    }
    //Assign new key
    vector[current + 1] = key;
  }
}
```

Algoritmo Merge sort.

Listing 6. Merge Sort Algorithm C++

```
/**********************************************************************/
/* Merge Sort functions                                               */
/**********************************************************************/
void merge(vector<int>& inputVector, int left, int middle, int right) {
  int leftIndex; //Index of the first subarray.
  int rightIndex; //Index of the second subarray.
  int mergeIndex; //Index of the merged subarray.

  int leftSize = middle - left + 1;
  int rightSize = right - middle;

  vector<int> leftVector, rightVector; //Temporary vectors.

  for (leftIndex = 0; leftIndex < leftSize; leftIndex++) {
    leftVector.push_back(inputVector.at(left + leftIndex));
  }
  for (rightIndex = 0; rightIndex < rightSize; rightIndex++) {
    rightVector.push_back(inputVector.at(middle +1 + rightIndex));
  }
  //Set the indices.
  leftIndex = 0;
  rightIndex = 0;
  mergeIndex = left;

  //Merge the temporary vectors into original.
  while (leftIndex < leftSize && rightIndex < rightSize)
  {
    if (leftVector[leftIndex] <= rightVector[rightIndex]) {
      inputVector[mergeIndex] = leftVector[leftIndex];
      leftIndex++;
    }
    else {
      inputVector[mergeIndex] = rightVector[rightIndex];
      rightIndex++;
    }
  }
  //Copy remaining elements(if any)
  while (leftIndex < leftSize)
  {
    inputVector[mergeIndex] = leftVector[leftIndex];
    leftIndex++;
    mergeIndex++;
  }
  while (rightIndex < rightSize)
  {
    inputVector[mergeIndex] = rightVector[rightIndex];
    rightIndex++;
    mergeIndex++;
  }
}
void mergeSort(vector<int>& vector, int left, int right) {
  if (vector.size() < 2) {
    return;
  }
  if (left < right) {
```

```
    int middle = left + (right - left) / 2;
    //Recursivity
    mergeSort(vector, left, middle);
    mergeSort(vector, middle + 1, right);

    merge(vector, left, middle, right);
  }
}
void mergeSortCall(vector<int>& vector) {
  mergeSort(vector, 0, vector.size() - 1);
}
```

Algoritmo de Benchmarking.

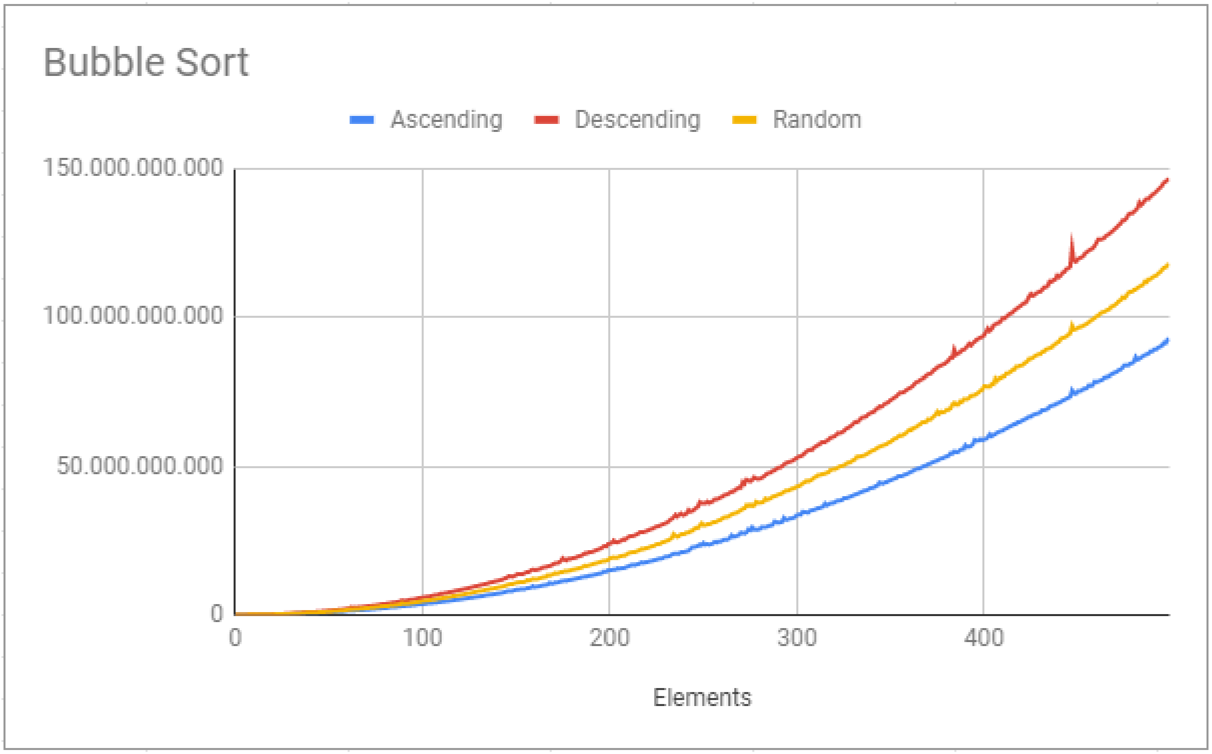
Listing 7. Benchmarking Algorithm C++

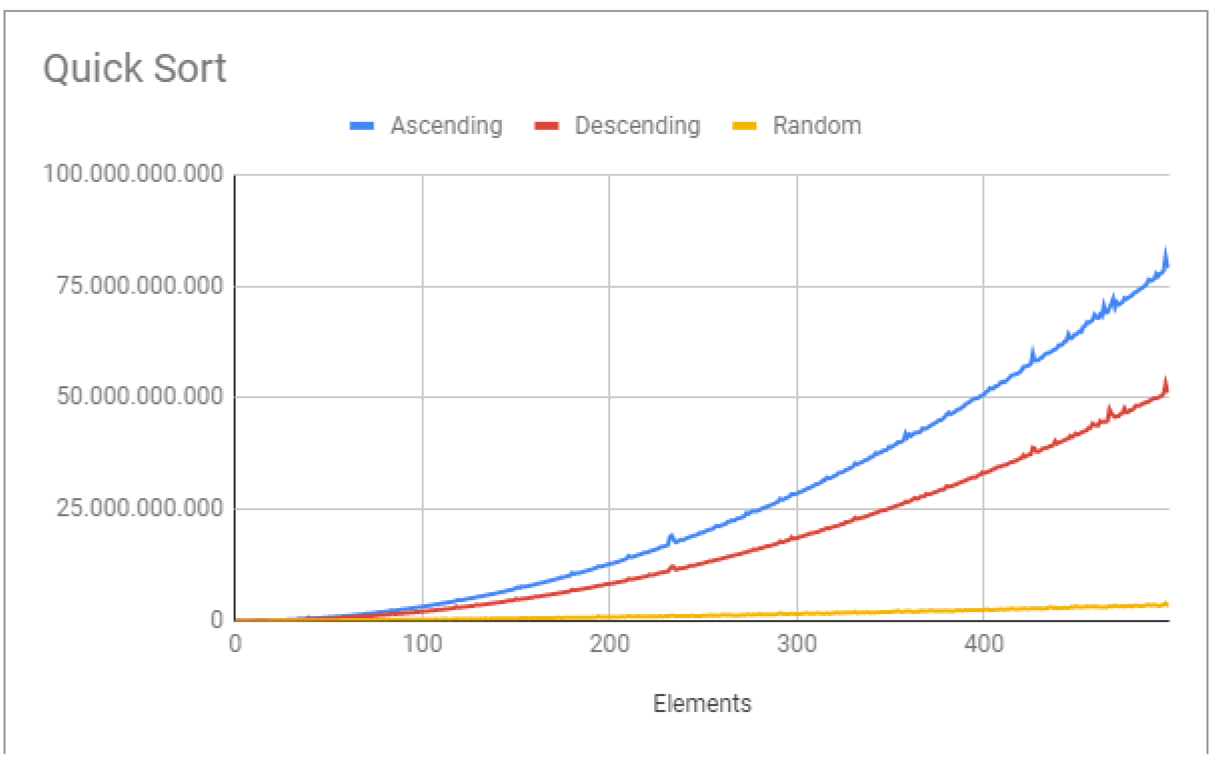
```cpp
/********************************************************************/
/* Benchmarking functions                                           */
/********************************************************************/
template <typename ...Args>
void benchmark(int testSize, int iterations, std::function<void(vector<int>&)> func, string fil
  //Create vectors that will be used for benchmarking.
  const vector<int> bestVector = ascendingVector(testSize);
  const vector<int> worstVector = descendingingVector(testSize);
  const vector<int> averageVector = randomVector(testSize, 0, 9);
  vector<int> usedVector;

  //Create duration variables for each case.
  duration<float, std::micro> duration;

  float bestDuration  = 0;
  float worstDuration = 0;
  float averageDuration = 0;

  //Create start and end time so it doesn't happen on every loop.
  auto startTime = high_resolution_clock::now();
  auto endTime = high_resolution_clock::now();

  //Initialize file stream.
  std::ofstream file;
  string fileText;

  //Write function name at file start.
  fileText += fileName;
  fileText += "\n";
  fileText += "Elements";
  fileText += ", ";
  fileText += "Best";
  fileText += ", ";
  fileText += "Worst";
  fileText += ", ";
  fileText += "Average";
  fileText += "\n";

  //Iterate case for every input size up to test size.
  for (int element =0; element < testSize;element++)
  {
    //Iterate to get average amount of time it takes to execute function.
    for (int iteration =0; iteration < iterations; iteration++)
    {
      //Testing best case.
      ///Set the current vector.
      usedVector = bestVector;
      usedVector.resize(element);
      startTime = high_resolution_clock::now();
      func(usedVector);
      endTime = high_resolution_clock::now();
      duration = (endTime - startTime);
      bestDuration += duration.count();

      //Testing worst case.
```

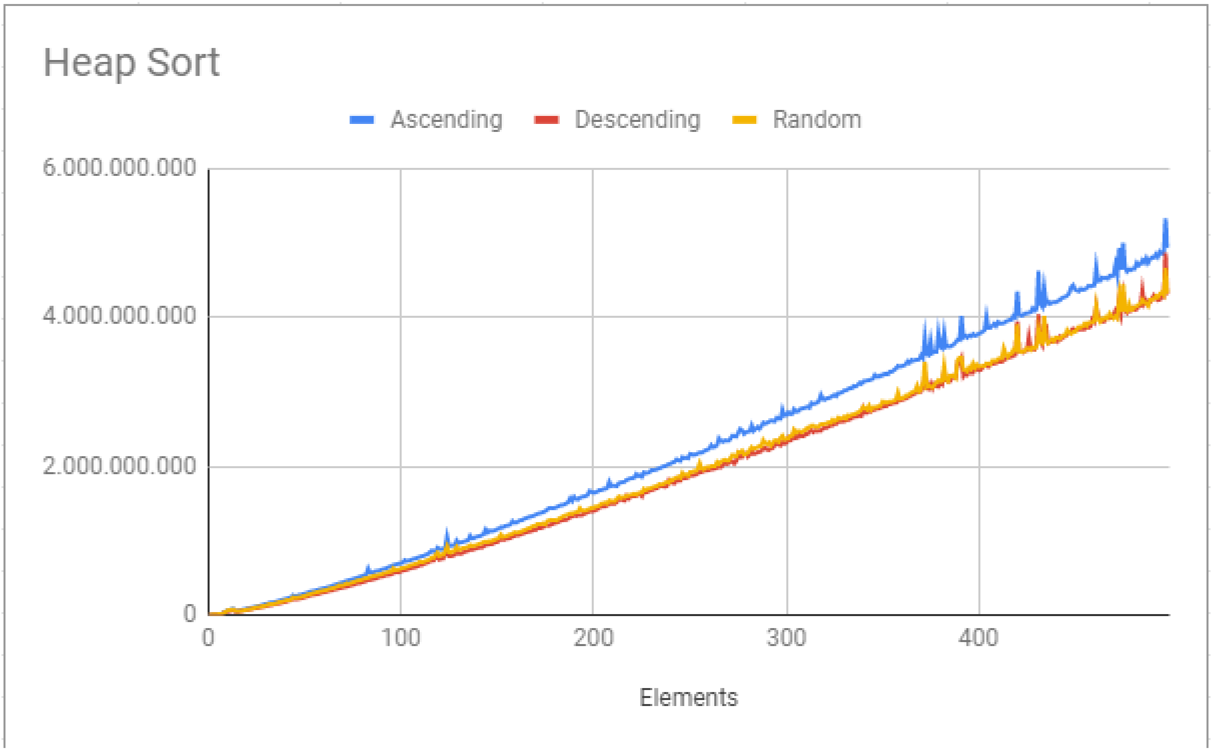
```
      ///Set the current vector.
      usedVector = worstVector;
      usedVector.resize(element);
      startTime = high_resolution_clock::now();
      func(usedVector);
      endTime = high_resolution_clock::now();
      duration = (endTime - startTime);
      worstDuration += duration.count();

      //Testing average case.
      ///Set the current vector.
      usedVector = averageVector;
      usedVector.resize(element);
      startTime = high_resolution_clock::now();
      func(usedVector);
      endTime = high_resolution_clock::now();
      duration = (endTime - startTime);
      averageDuration += duration.count();
    }
    //Get average time.
    bestDuration /= iterations;
    worstDuration /= iterations;
    averageDuration /= iterations;

    //Write duration on file.
    fileText += std::to_string(element);
    fileText += ", ";
    fileText += std::to_string(bestDuration);
    fileText += ", ";
    fileText += std::to_string(worstDuration);
    fileText += ", ";
    fileText += std::to_string(averageDuration);
    fileText += "\n";
  }
  file.open(fileName);
  file.clear();
  file << fileText;
  file.close();

}
```

Benchmark Results

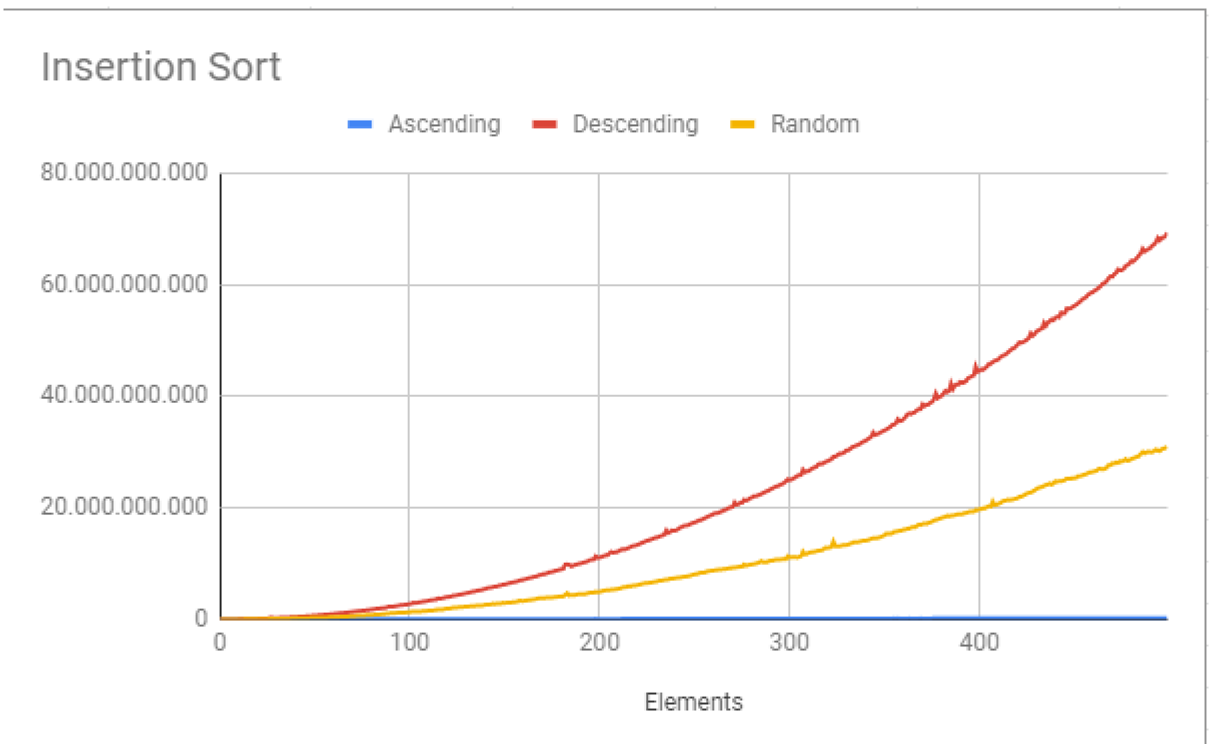
Nota: El eje X representa el numero de elementos en el vector mientras que el eje Y representa el tiempo(microsegundos). En estas graficas se observan los resultados de cada algoritmo utilizando diferentes entradas.



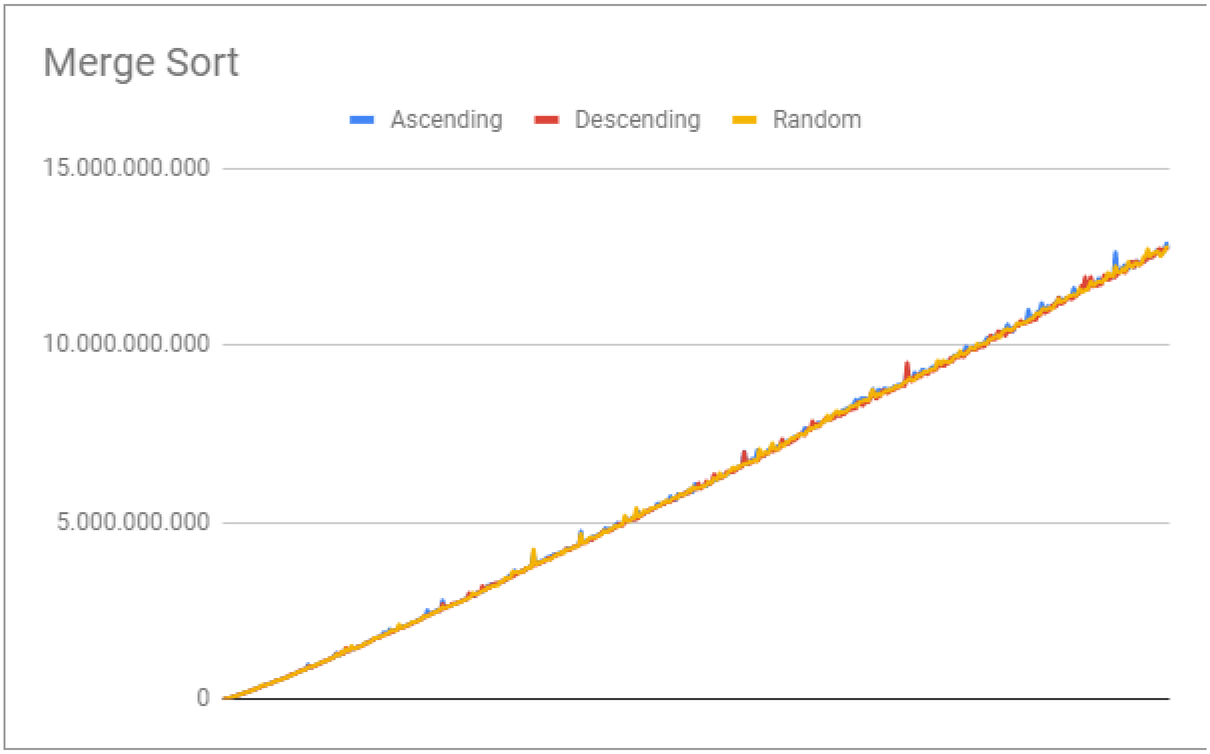




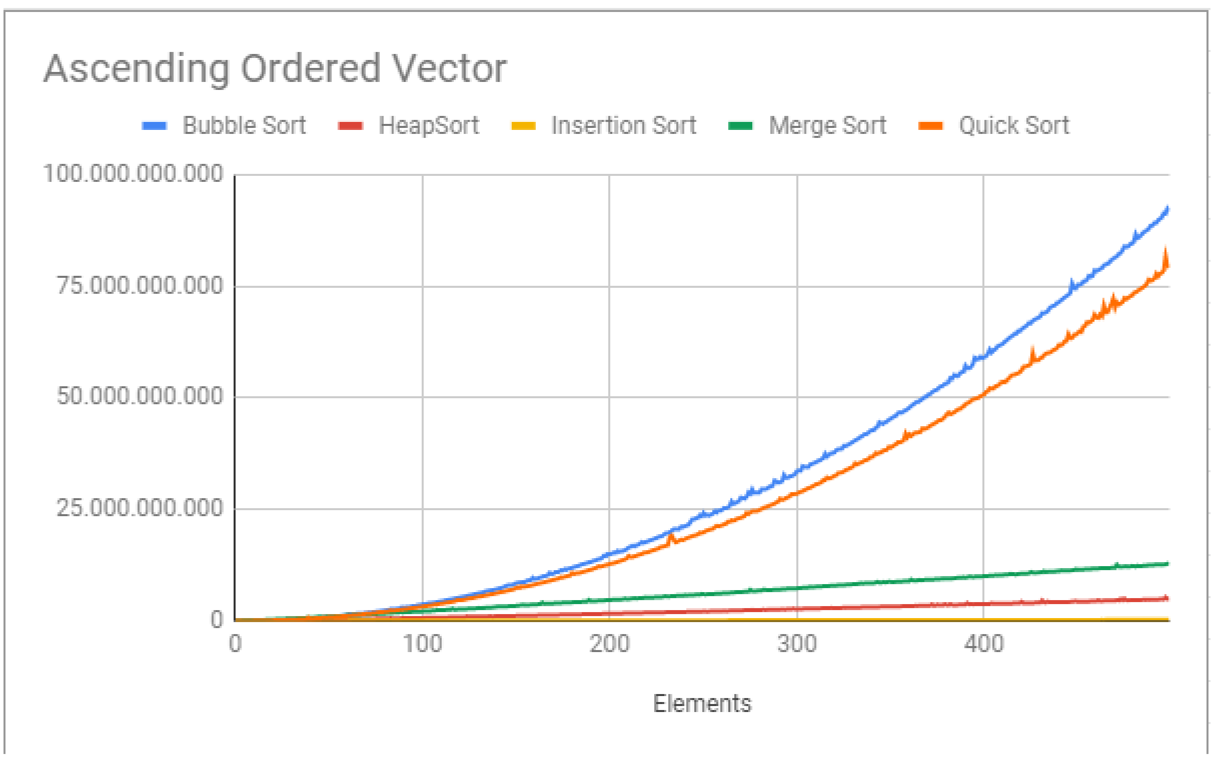
Heap Sort
Ascending
Descending
Random
6.000.000.000
4.000.000.000
2.000.000.000
0
0
100
200
300
400
Elements

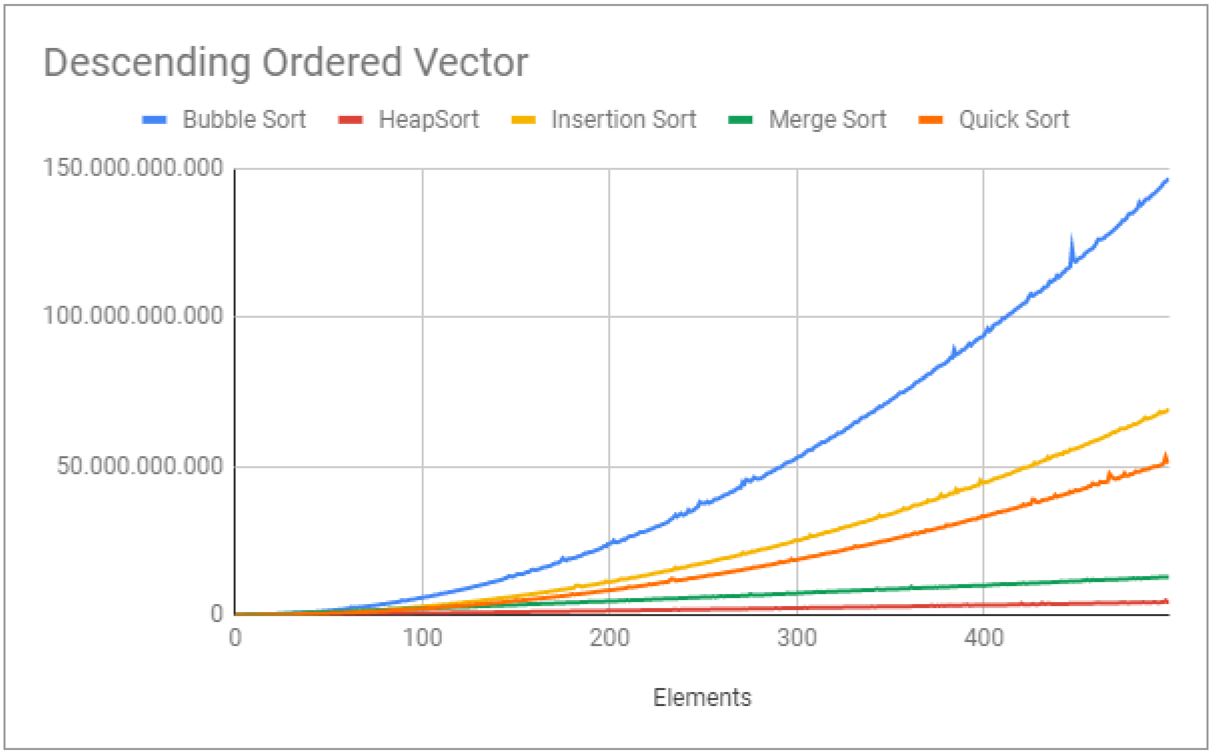


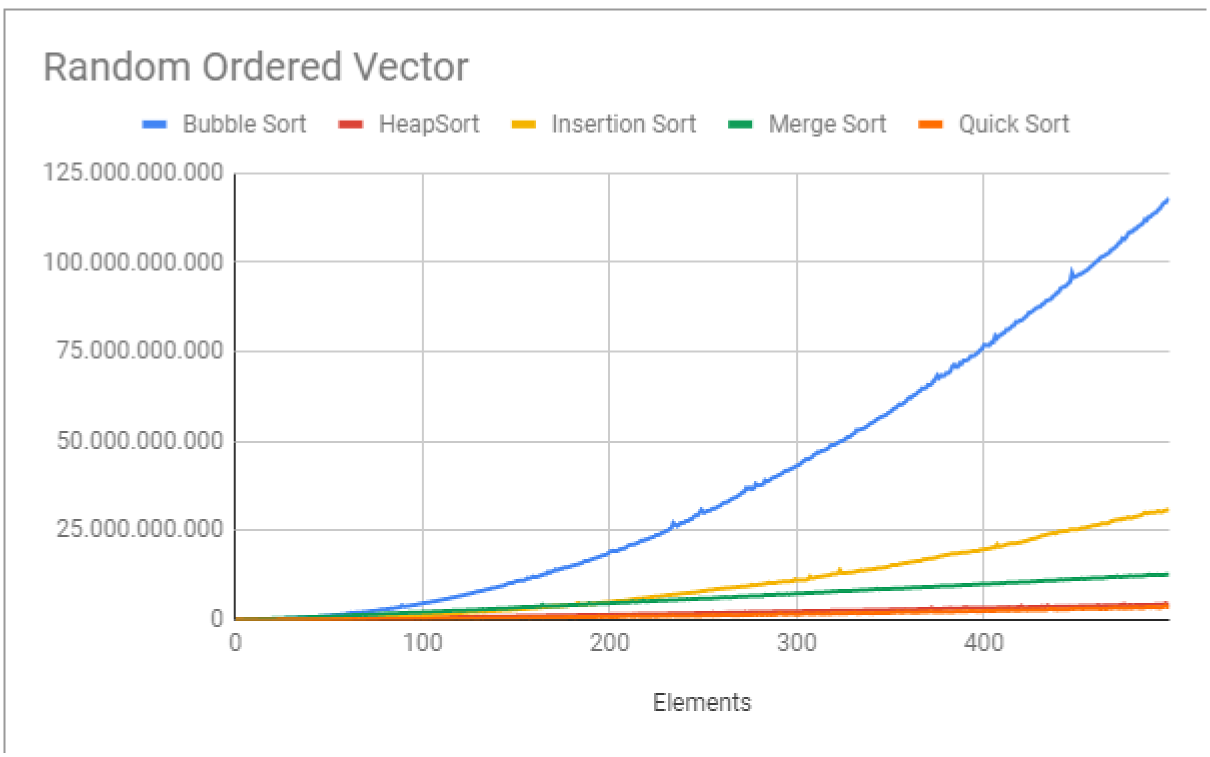
Insertion Sort
Ascending
Descending
Random
80.000.000.000
60.000.000.000
40.000.000.000
20.000.000.000
0
0
100
200
300
400
Elements



En estas graficas se compara los algoritmos utilizando el mismo vector de entrada.




Descending Ordered Vector
Bubble Sort
HeapSort
Insertion Sort
Merge Sort
Quick Sort
150.000.000.000
100.000.000.000
50.000.000.000
0
0
100
200
300
400
Elements


Random Ordered Vector
Bubble Sort
HeapSort
Insertion Sort
Merge Sort
Quick Sort
125.000.000.000
100.000.000.000
75.000.000.000
50.000.000.000
25.000.000.000
0
0
100
200
300
400
Elements