Sorting Algorithms

Santiago Toll Leyva
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv16a.stoll@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 30, 2019

Tarea 4: 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 evaluarlos 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.

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).

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 Bucket sort.

Listing 2. Bucket Sort Algorithm C++

```
/*****************************************************************/
/* Bucket Sort                                                   */
/*****************************************************************/
void bucketSort(vector<float> & inputVector)
{
  if (inputVector.size() < 2) {
    return;
  }
  int size = inputVector.size();
  vector<vector<float>> buckets;
  buckets.resize(size);

  for (int i = 0; i < size; i++)
  {
    int bucketIndex = size * inputVector[i];
    buckets[bucketIndex].push_back(inputVector[i]);
  }

  for (int i = 0; i < size; i++)
    std::sort(buckets[i].begin(), buckets[i].end());

  int index = 0;
  for (int i = 0; i < size; i++)
    for (int j = 0; j < buckets[i].size(); j++)
      inputVector[index++] = buckets[i][j];
}
```

Algoritmo Radix sort.

Listing 3. Radix Sort Algorithm C++

```
/*****************************************************************/
/* Radix Sort                                                    */
/*****************************************************************/
void radixCountSort(vector<int>& inputVector, int exp) {
  int size = inputVector.size();
  vector<int> output; // output array
  output.resize(size);
  int i, count[10] = { 0 };

  // Store count of occurrences in count[]
  for (i = 0; i < size; i++)
    count[(inputVector[i] / exp) % 10]++;

  // Change count[i] so that count[i] now contains actual
  //  position of this digit in output[]
  for (i = 1; i < 10; i++)
    count[i] += count[i - 1];

  // Build the output array
  for (i = size - 1; i >= 0; i--)
  {
    output[count[(inputVector[i] / exp) % 10] - 1] = inputVector[i];
    count[(inputVector[i] / exp) % 10]--;
  }

  // Copy the output array to arr[], so that arr[] now
  // contains sorted numbers according to current digit
  for (i = 0; i < size; i++)
    inputVector[i] = output[i];
}

void radixsort(vector<int>& inputVector)
{
  if (inputVector.size() < 2) {
    return;
  }
  int size = inputVector.size();
  int max = *max_element(inputVector.begin(), inputVector.end());

  for (int exp = 1; max / exp > 0; exp *= 10) {
    radixCountSort(inputVector, exp);
  }

}
```

Algoritmo Counting Sort.

Listing 4. Counting Search C++

```
1  /*****************************************************************/
2  /* Counting Sort                                                 */
3  /*****************************************************************/
4  void countSort(vector<int>& inputVector)
5  {
6    if (inputVector.size() < 2) {
7      return;
8    }
9    int max = *max_element(inputVector.begin(), inputVector.end());
10   int min = *min_element(inputVector.begin(), inputVector.end());
11   int range = max - min + 1;
12
13   vector<int> count(range), output(inputVector.size());
14   for (int i = 0; i < inputVector.size(); i++)
15     count[inputVector[i] - min]++;
16
17   for (int i = 1; i < count.size(); i++)
18     count[i] += count[i - 1];
19
20   for (int i = inputVector.size() - 1; i >= 0; i--)
21   {
22     output[count[inputVector[i] - min] - 1] = inputVector[i];
23     count[inputVector[i] - min]--;
24   }
25
26   for (int i = 0; i < inputVector.size(); i++)
27     inputVector[i] = output[i];
28 }
```

Algoritmos de Benchmarking.

Se utilizaron dos funciones para hacer benchmarking ya que el algoritmo Bucket Sort utiliza valores flotantes. Estas dos funciones hacen lo mismo, pero operan diferentes tipos de entrada.

Listing 5. Benchmarking Algorithm C++

```
/*****************************************************************/
/* 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);
```

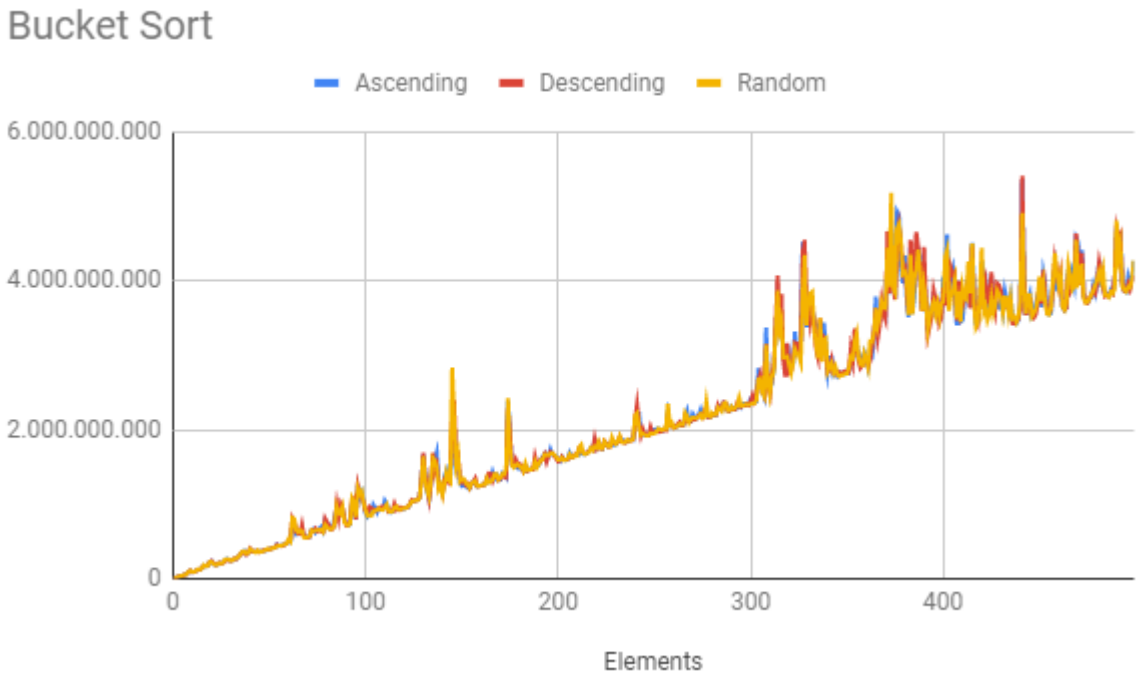
```
53          bestDuration += duration.count();
54
55          //Testing worst case.
56          ///Set the current vector.
57          usedVector = worstVector;
58          usedVector.resize(element);
59          startTime = high_resolution_clock::now();
60          func(usedVector);
61          endTime = high_resolution_clock::now();
62          duration = (endTime - startTime);
63          worstDuration += duration.count();
64
65          //Testing average case.
66          ///Set the current vector.
67          usedVector = averageVector;
68          usedVector.resize(element);
69          startTime = high_resolution_clock::now();
70          func(usedVector);
71          endTime = high_resolution_clock::now();
72          duration = (endTime - startTime);
73          averageDuration += duration.count();
74      }
75      //Get average time.
76      bestDuration /= iterations;
77      worstDuration /= iterations;
78      averageDuration /= iterations;
79
80      //Write duration on file.
81      fileText += std::to_string(element);
82      fileText += ", ";
83      fileText += std::to_string(bestDuration);
84      fileText += ", ";
85      fileText += std::to_string(worstDuration);
86      fileText += ", ";
87      fileText += std::to_string(averageDuration);
88      fileText += "\n";
89    }
90    file.open(fileName);
91    file.clear();
92    file << fileText;
93    file.close();
94
95  }
96
97
98  void f_benchmark(int testSize, int iterations, std::function<void(vector<float>&)> func, string
99    //Create vectors that will be used for benchmarking.
100   const vector<float> bestVector = randomFloatVector(testSize, 0, 255);
101   const vector<float> worstVector = randomFloatVector(testSize, 0, 255);
102   const vector<float> averageVector = randomFloatVector(testSize, 0, 1);
103
104
105   vector<float> usedVector;
106
107   //Create duration variables for each case.
108   duration<float, std::micro> duration;
109
110   float bestDuration = 0;
```

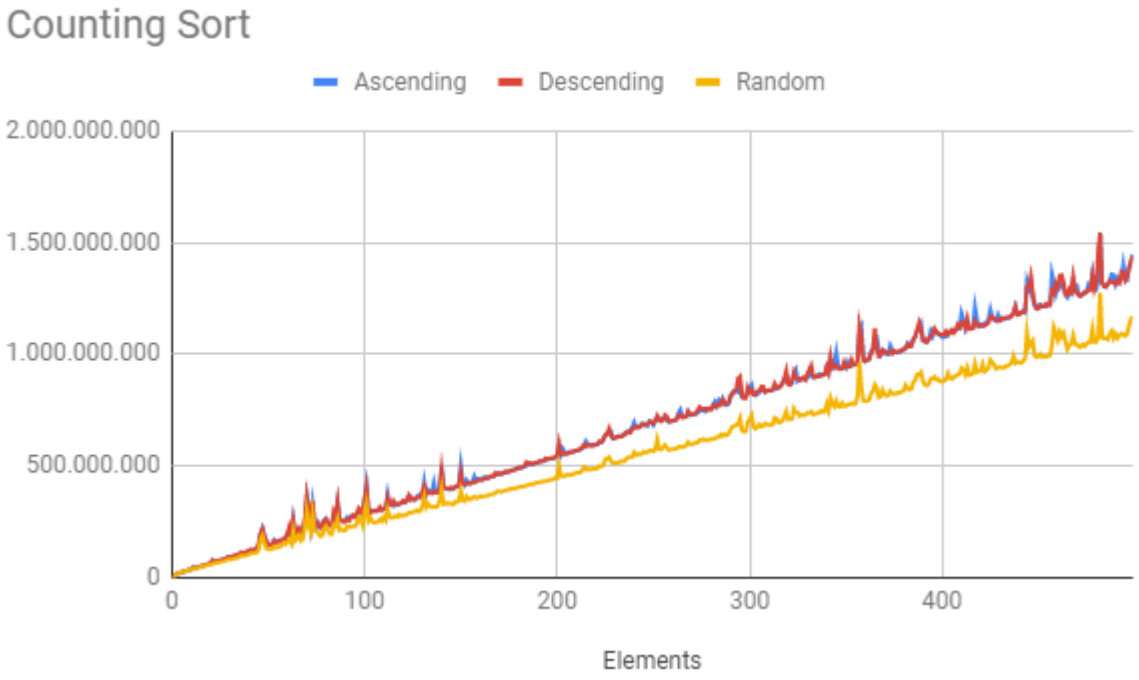
```
111     float worstDuration = 0;
112     float averageDuration = 0;
113 
114     //Create start and end time so it doesn't happen on every loop.
115     auto startTime = high_resolution_clock::now();
116     auto endTime = high_resolution_clock::now();
117 
118     //Initialize file stream.
119     std::ofstream file;
120     string fileText;
121 
122     //Write function name at file start.
123     fileText += fileName;
124     fileText += "\n";
125     fileText += "Elements";
126     fileText += ", ";
127     fileText += "Best";
128     fileText += ", ";
129     fileText += "Worst";
130     fileText += ", ";
131     fileText += "Average";
132     fileText += "\n";
133 
134     //Iterate case for every input size up to test size.
135     for (int element = 0; element < testSize; element++)
136     {
137       //Iterate to get average amount of time it takes to execute function.
138       for (int iteration = 0; iteration < iterations; iteration++)
139       {
140         //Testing best case.
141         ///Set the current vector.
142         usedVector = bestVector;
143         usedVector.resize(element);
144         startTime = high_resolution_clock::now();
145         func(usedVector);
146         endTime = high_resolution_clock::now();
147         duration = (endTime - startTime);
148         bestDuration += duration.count();
149 
150         //Testing worst case.
151         ///Set the current vector.
152         usedVector = worstVector;
153         usedVector.resize(element);
154         startTime = high_resolution_clock::now();
155         func(usedVector);
156         endTime = high_resolution_clock::now();
157         duration = (endTime - startTime);
158         worstDuration += duration.count();
159 
160         //Testing average case.
161         ///Set the current vector.
162         usedVector = averageVector;
163         usedVector.resize(element);
164         startTime = high_resolution_clock::now();
165         func(usedVector);
166         endTime = high_resolution_clock::now();
167         duration = (endTime - startTime);
168         averageDuration += duration.count();
```

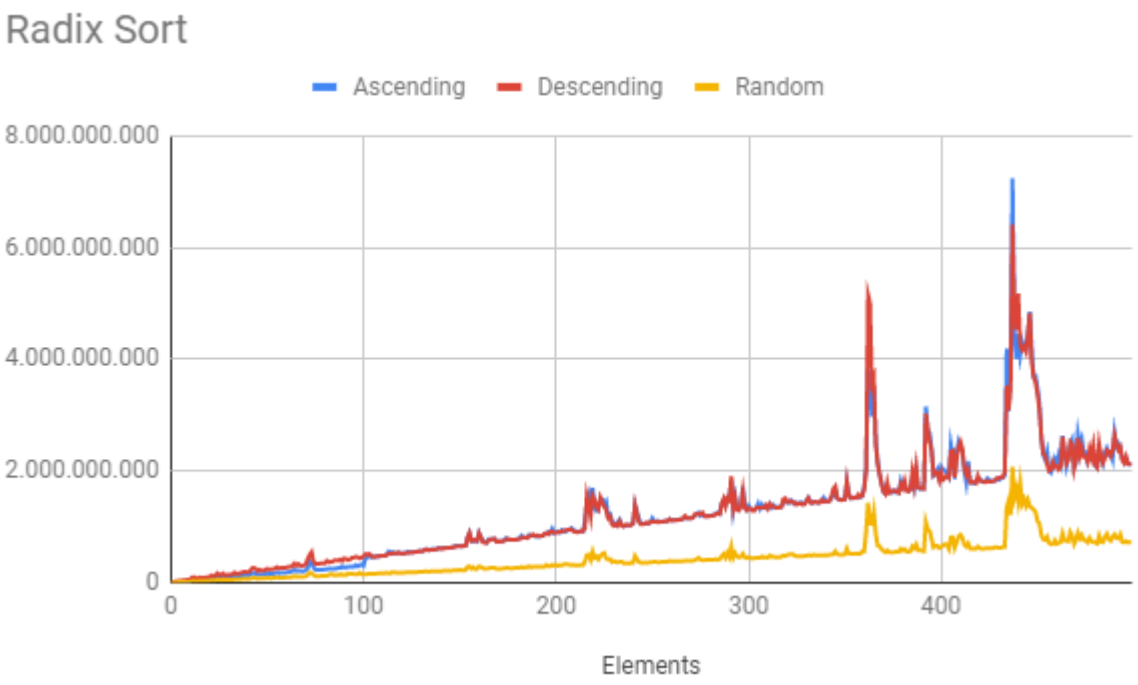
```
169
170
171         }
172         //Get average time.
173         bestDuration /= iterations;
174         worstDuration /= iterations;
175         averageDuration /= iterations;
176
177         //Write duration on file.
178         fileText += std::to_string(element);
179         fileText += ", ";
180         fileText += std::to_string(bestDuration);
181         fileText += ", ";
182         fileText += std::to_string(worstDuration);
183         fileText += ", ";
184         fileText += std::to_string(averageDuration);
185         fileText += "\n";
186       }
187       file.open(fileName);
188       file.clear();
189       file << fileText;
190       file.close();
191
192   }
```

Benchmark Results

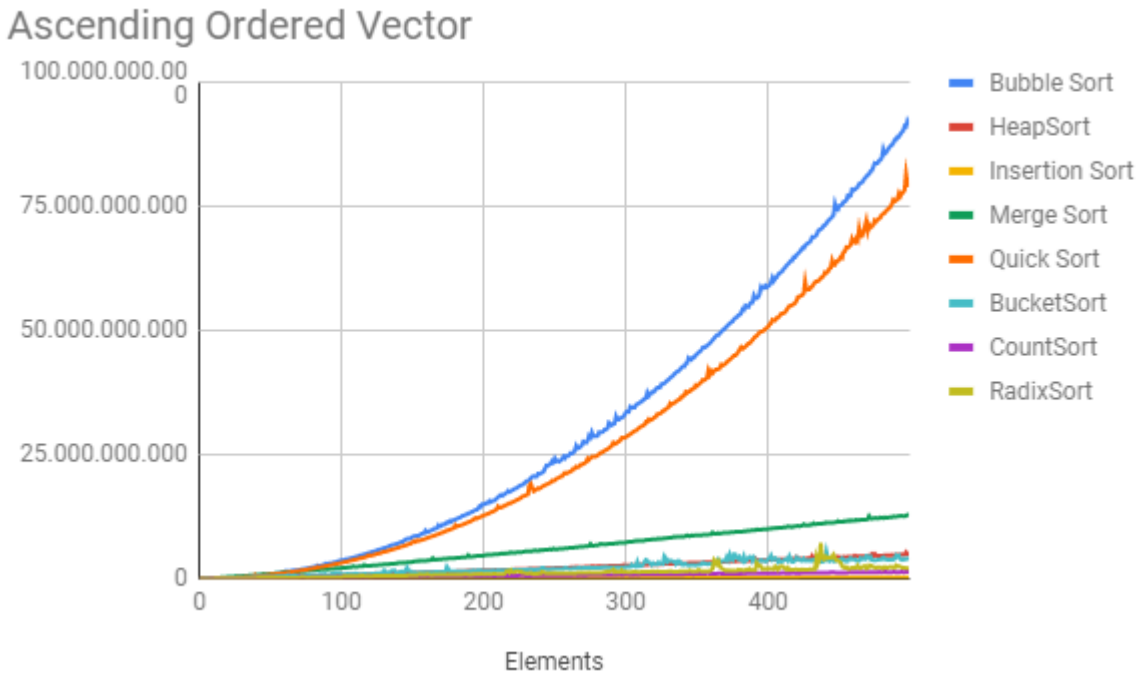
Nota: El eje X representa el numero de elementos en el vector mientras que el eje Y representa el tiempo(microsegundos). En estas graficas se observan los resultados de cada algoritmo utilizando diferentes entradas.

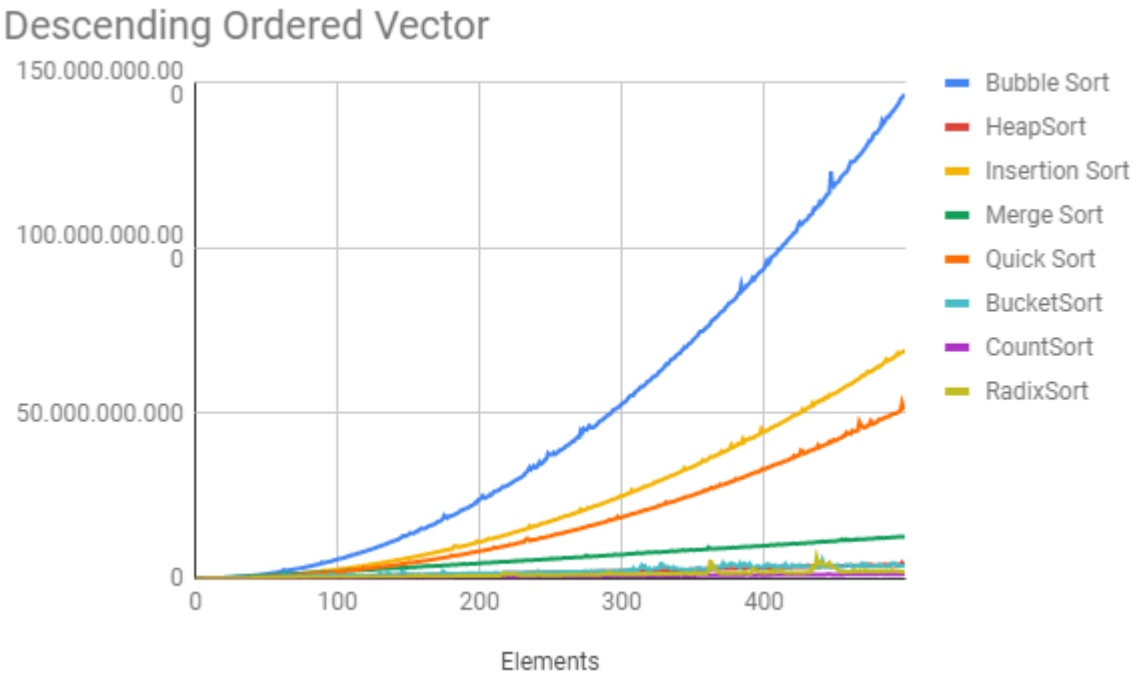


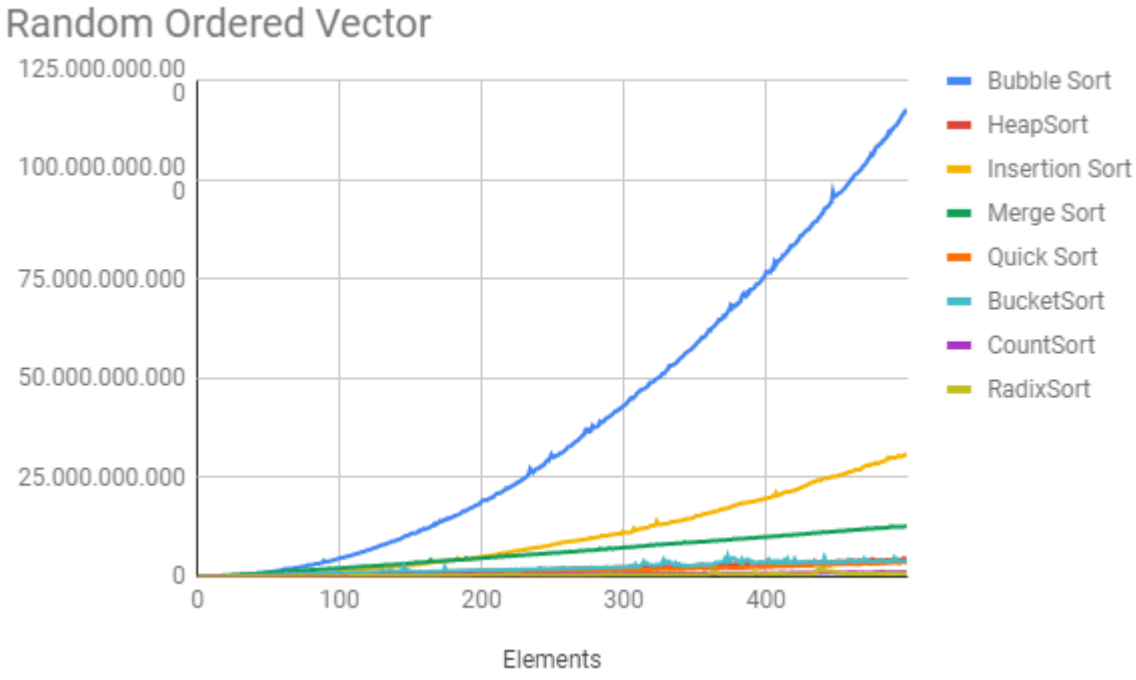




En las siguientes graficas se comparan todos los algoritmos de ordenamiento que se han trabajado hasta el momento. Se puede observar el comportamiento de los algoritmos y evaluar cual es mas conveniente dependiendo el caso especifico para el que se quiere utilizar.




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


Random Ordered Vector
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
Bubble Sort
HeapSort
Insertion Sort
Merge Sort
Quick Sort
BucketSort
CountSort
RadixSort