Ordenamientos y Búsqueda en Arreglos

Accel Israel Magaña Rodríguez
Universidad de Artes Digitales

Guadalajara, Jalisco

Email: idv17a.amagana@uartesdigitales.edu.mx

Profesor: Efraín Padilla

Mayo 18, 2019

I. Ordenamientos

Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar esa información de manera eficiente es deseable que aquella esté ordenada. Existen varios métodos para ordenar las diferentes estructuras de datos básicas.

*A. **Solución por cuentas (counting):***

El ordenamiento por cuentas (counting sort en inglés) es un algoritmo de ordenamiento en el que se cuenta el número de elementos de cada clase para luego ordenarlos. Sólo puede ser utilizado por tanto para ordenar elementos que sean contables (como los números enteros en un determinado intervalo, pero no los números reales, por ejemplo)..

Para los casos en este ordenamiento tenemos:

Best case	$\Omega(n+k)$
Worst case	$\mathcal{O}(n+k))$
Avarage case	$\theta(n+k)$

Implementación de código en C++:

```
void countingSort(std::vector<int>& vec)
{
  int maxValue = vec.size();
  int* counting = new int[maxValue];
  int k = 0;
  for(int i = 0; i < maxValue; i++)
    counting[i] = 0;
  for(int i = 0; i < vec.size(); i++)
    counting[vec[i]]++;
  for(int i = 0; i < maxValue; i++)
  {
    int count = counting[i];
    for(int j = 0; j < count; j++)
    {
      vec[k] = i;
      k++;
    }
  }
}
```

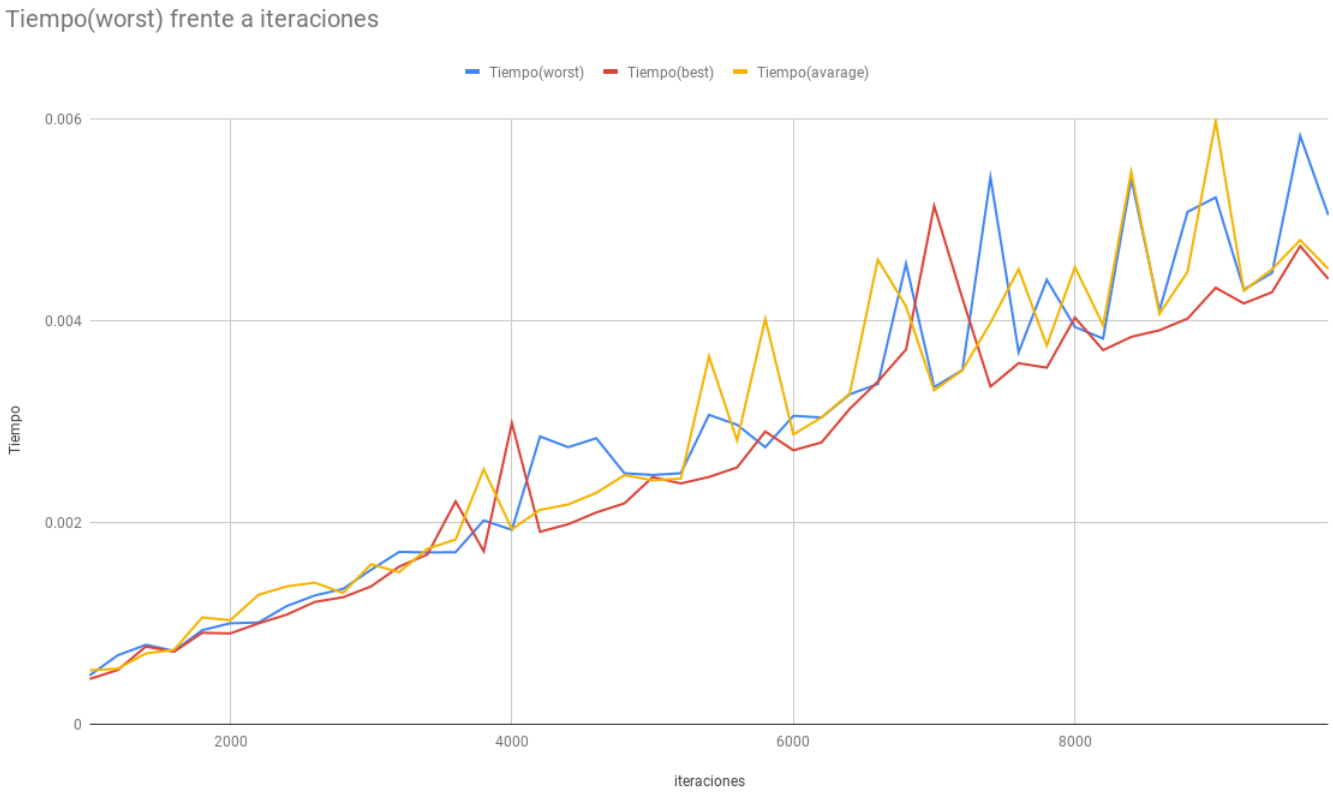

Fig. 1. Aunque en la a simple vista parece que está creciendo, tomemos en cuenta que el algoritmo está corriendo mas de 9mil veces y el tiempo máximo no sube exponencial, por lo que los tiempos son los indicados anteriormente.

B. ***Solución Radix:***

En informática, el ordenamiento Radix (radix sort en inglés) es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, radix sort no está limitado sólo a los enteros.

Para los casos en este ordenamiento tenemos:

Best case	$\Omega(d(n+k'))$
Worst case	$\mathcal{O}(d(n+k'))$
Avarage case	$\theta(d(n+k'))$

Implementación de código en C++:

```
std::vector<int> radixHelper(std::vector<int>& arr, int spot)
{
  std::vector<int> count(10, 0);

  std::vector<int> retVal(arr.size(), 0);

  for(int i = 0; i < arr.size(); ++i)//Fill count array
  {
    int temp = arr[i];
    for(int j = 0; j < spot; ++j)
    {
      temp /= 10;
    }
    count[temp % 10]++;
  }

  for(int i = 1; i < count.size(); ++i)//Sum count array
  {
    count[i] += count[i - 1];
  }

  for(int i = arr.size() - 1; i >= 0; --i)//Find which values match what spots //complicated part
  {
    int temp = arr[i];
    for(int j = 0; j < spot; ++j)
    {
      temp /= 10;
    }

    temp = temp % 10;

    for(int j = 0; j < count.size(); ++j)
    {
      if(j == temp)
      {
        retVal[--count[j]] = arr[i];
      }
    }
  }
  return retVal;
}
```

```
void radixSort(std::vector<int>& arr)
{
  for(int i = 0; i < 10; ++i)
  {
    arr = radixHelper(arr, i);
  }
}
```

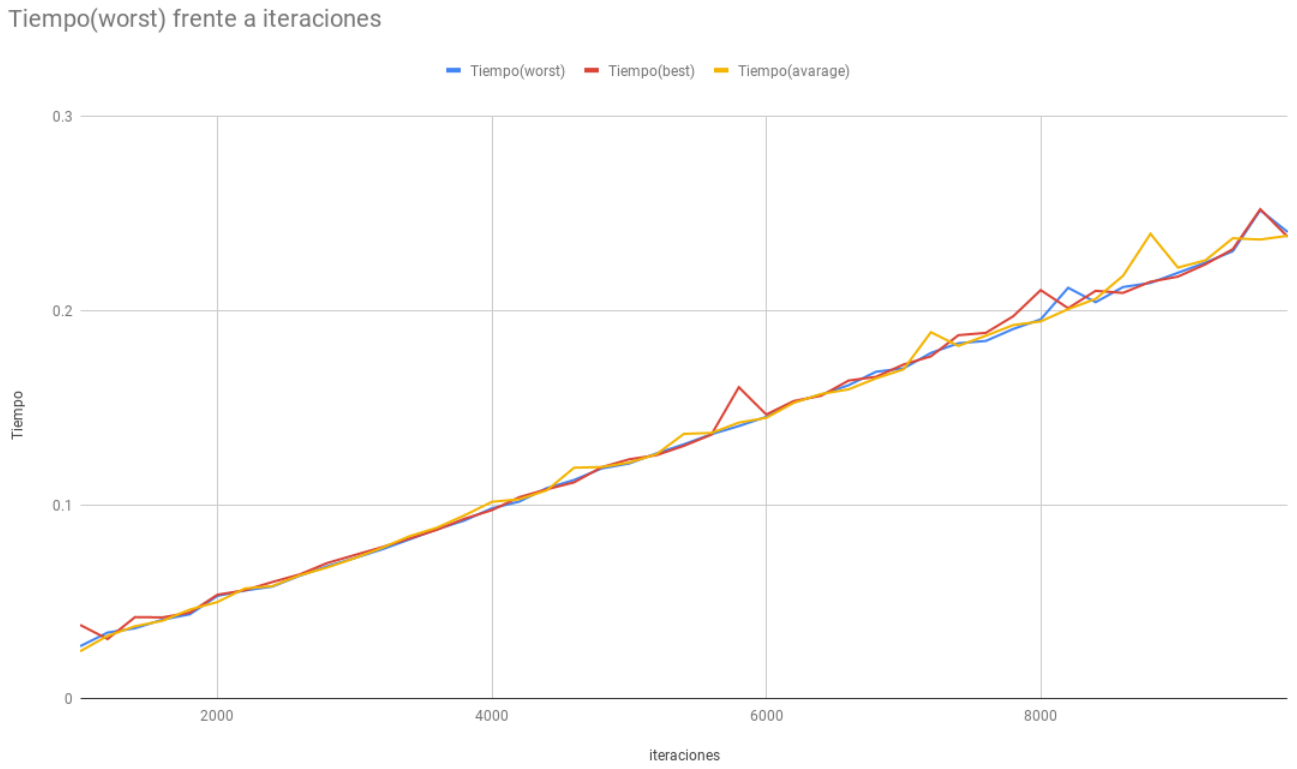

Fig. 2. Aunque en la a simple vista parece que está creciendo, tomemos en cuenta que el algoritmo está corriendo mas de 9mil veces y el tiempo máximo no sube exponencial, por lo que los tiempos son los indicados anteriormente.

*C. **Solución Cubeta (bucket):***

El algoritmo por cubetas es también denominado bucket sort o bin sort. Este algoritmo corre en tiempo lineal [esto es O(n)], debido a que el algoritmo asume que la entrada consiste de elementos en un rango pequeño, esto es, divide la entrada en n subconjuntos uniformes (cubetas) para después distribuir cada uno de los elementos en dichos subconjuntos. Así para producir la salida es únicamente necesario ordenar los elementos dentro de la cubeta y posteriormente en base al orden de cada cubeta listar cada elemento.

Para los casos en este ordenamiento tenemos:

Best case	$\Omega(n))$
Worst case	$\mathcal{O}(n)$
Avarage case	$\theta(n)$

Implementación de código en C++:

```
void BucketSort(std::vector<int>& data) {
  int minValue = data[0];
  int maxValue = data[0];

  for(int i = 1; i < data.size() -1; i++)
  {
    if(data[i] > maxValue)
      maxValue = data[i];
    if(data[i] < minValue)
      minValue = data[i];
  }

  int bucketLength = maxValue - minValue + 1;
  std::vector<int>* bucket = new std::vector<int>[bucketLength];

  for(int i = 0; i < bucketLength; i++)
  {
    bucket[i] = std::vector<int>();
  }

  for(int i = 0; i < data.size() - 1; i++)
  {
    bucket[data[i] - minValue].push_back(data[i]);
  }

  int k = 0;
  for(int i = 0; i < bucketLength; i++)
  {
    int bucketSize = bucket[i].size();

    if(bucketSize > 0)
    {
      for(int j = 0; j < bucketSize; j++)
      {
        data[k] = bucket[i][j];
        k++;
      }
    }
  }
}
```

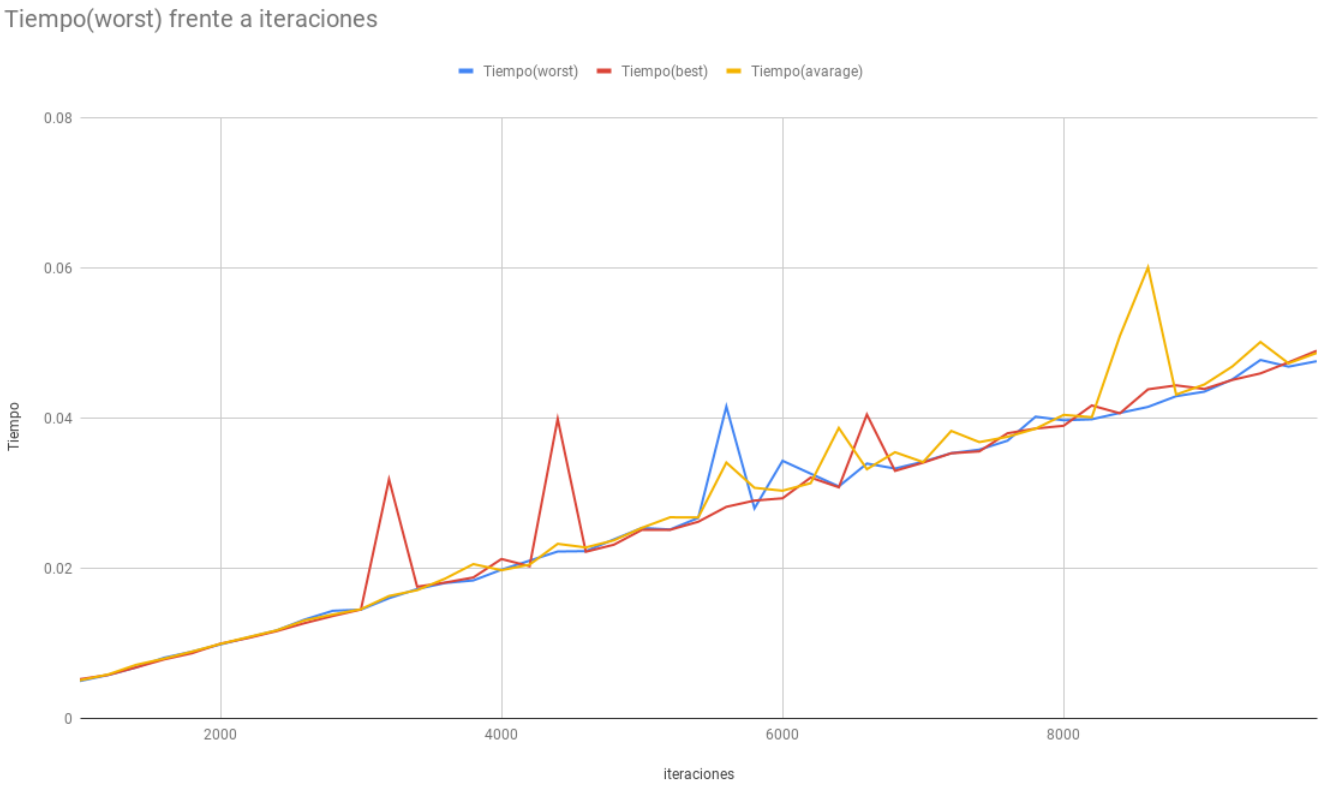

Fig. 3. Aunque en la a simple vista parece que está creciendo, tomemos en cuenta que el algoritmo está corriendo mas de 9mil veces y el tiempo máximo no sube exponencial, por lo que los tiempos son los indicados anteriormente.