Counting vs Bucket vs Radix 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. Counting Sorting

Este algoritmo sólo puede trabajar con número enteros. Está diseñado para ordenar los número por medio de una serie de sumas con respecto a la posición de éstos. No es eficaz con rangos muy grandes.

B. Radix Sorting

Este algoritmo de comportamiento lineal realiza comporaciones a partir los dígitos de un número, comenzando con el menos significativo. Radix utiliza el algoritmo de counting como una subrutina dentro de sus procesos.

C. Bucket Sorting

El algoritmo de bucket utiliza un método similar al comportamiento de una tabla hash. Es un algoritmo eficaz con los número flotantes.

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. Counting Sorting

```
void
countingSort(std::vector<int> & _vector)
{
  int size = _vector.size();

  std::vector<int> indexed_list;
  indexed_list.resize(size, 0);

  for (int index = 0; index < size; ++index)
  {
    indexed_list[index]++;
  }

  std::vector<int>::iterator it;
  int indexed_size = indexed_list.size();
  for (int index = 1; index < indexed_size; ++index)
  {
    indexed_list[index] += indexed_list[index - 1];
  }

  std::vector<int> original_vector(_vector);
  for (int index = 0; index < size; ++index)
  {
    int i = original_vector[index] - 1;
    int position = indexed_list[i];
    indexed_list[i]--;

    _vector[position - 1] = original_vector[index];
  }

  return;
}
```

B. Radix Sorting

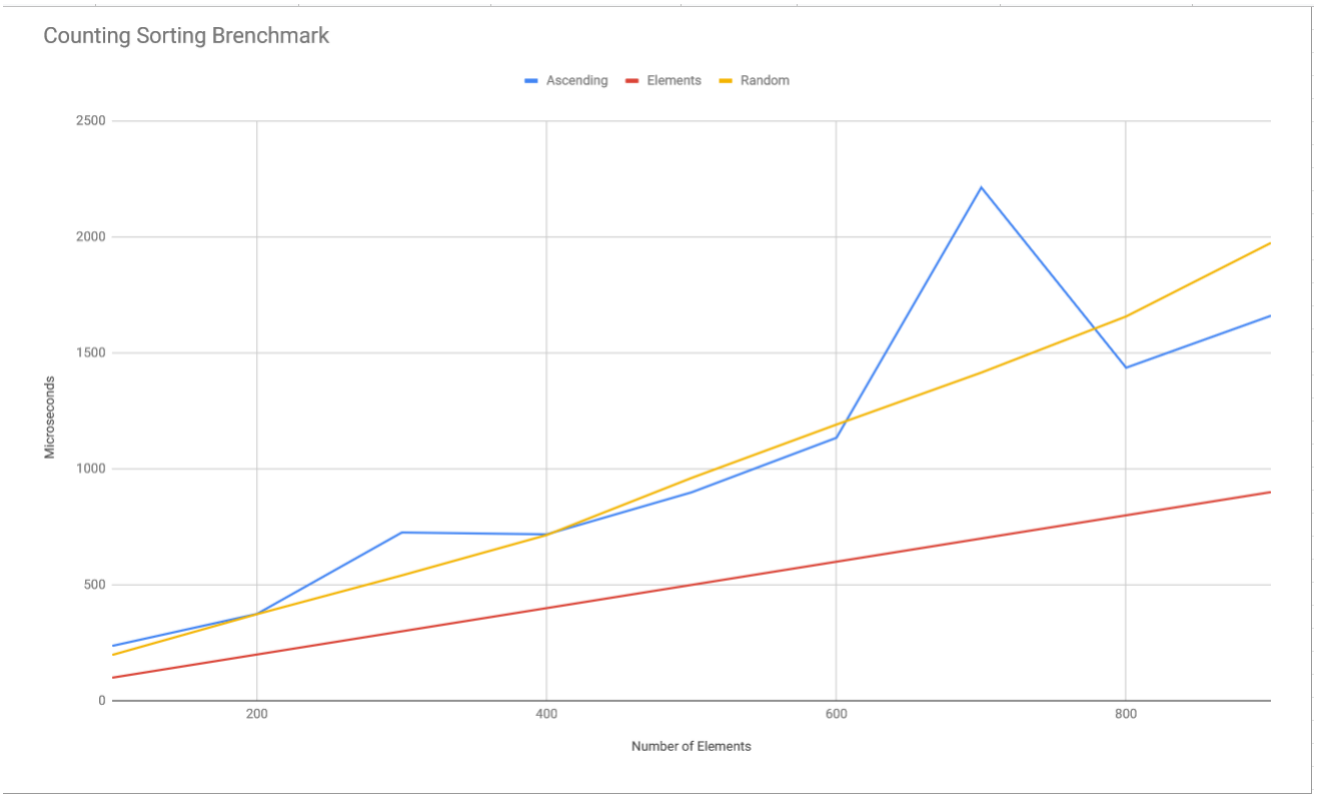
```
void
countingSortByDigit(std::vector<int> & _vector, int & _exp)
{
  std::vector<int> origin(_vector);

  int size = _vector.size();
  int count[10] = { 0 };

  for (int index = 0; index < size; ++index)
  {
    count[(origin[index] / _exp) % 10]++;
  }

  for (int index = 1; index < 10; ++index)
  {
    count[index] += count[index - 1];
  }

  for (int index = size - 1; index >= 0; --index)
```

```
  {
    int j = (origin[index] / _exp) % 10;

    _vector[count[j] - 1] = origin[index];
    count[j]--;
  }

  return;
}

void
radixSort(std::vector<int> & _vector)
{

  int max_num = getMax(_vector);

  for (int exp = 1; (max_num / exp) > 0; exp *= 10)
  {
    countingSortByDigit(_vector, exp);
  }

  return;
}
```

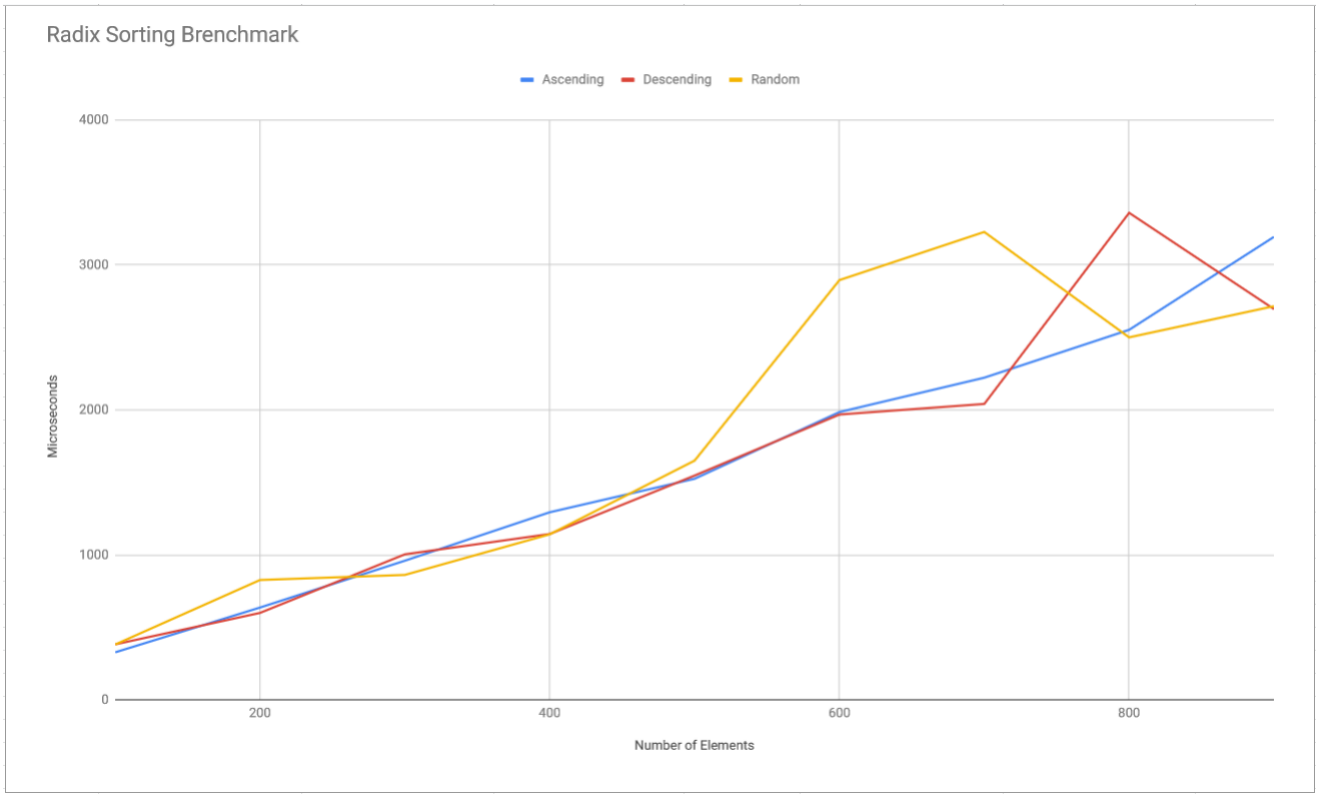
C. Bucket Sorting

```
void
bucketSort(std::ve       ctor<float> & _vector)
{
  int size = _vector.size();

  std::vector<std::vector<float>> buckets;
  buckets.resize(size);

  for (int index = 0; index < size; ++index)
  {
    int bucket_idx = size * _vector[index];
    buckets[bucket_idx].push_back(_vector[index]);
  }

  for (int index = 0; index < size; ++index)
  {
    std::sort(buckets[index].begin(), buckets[index].end(), std::less<float>());
  }

  int arr_index = 0;
  for (int bucket_idx = 0; bucket_idx < size; ++bucket_idx)
  {
    for (int j = 0; j < buckets[bucket_idx].size(); ++j)
    {
      _vector[arr_index++] = buckets[bucket_idx][j];
    }
  }

  return;
}
```

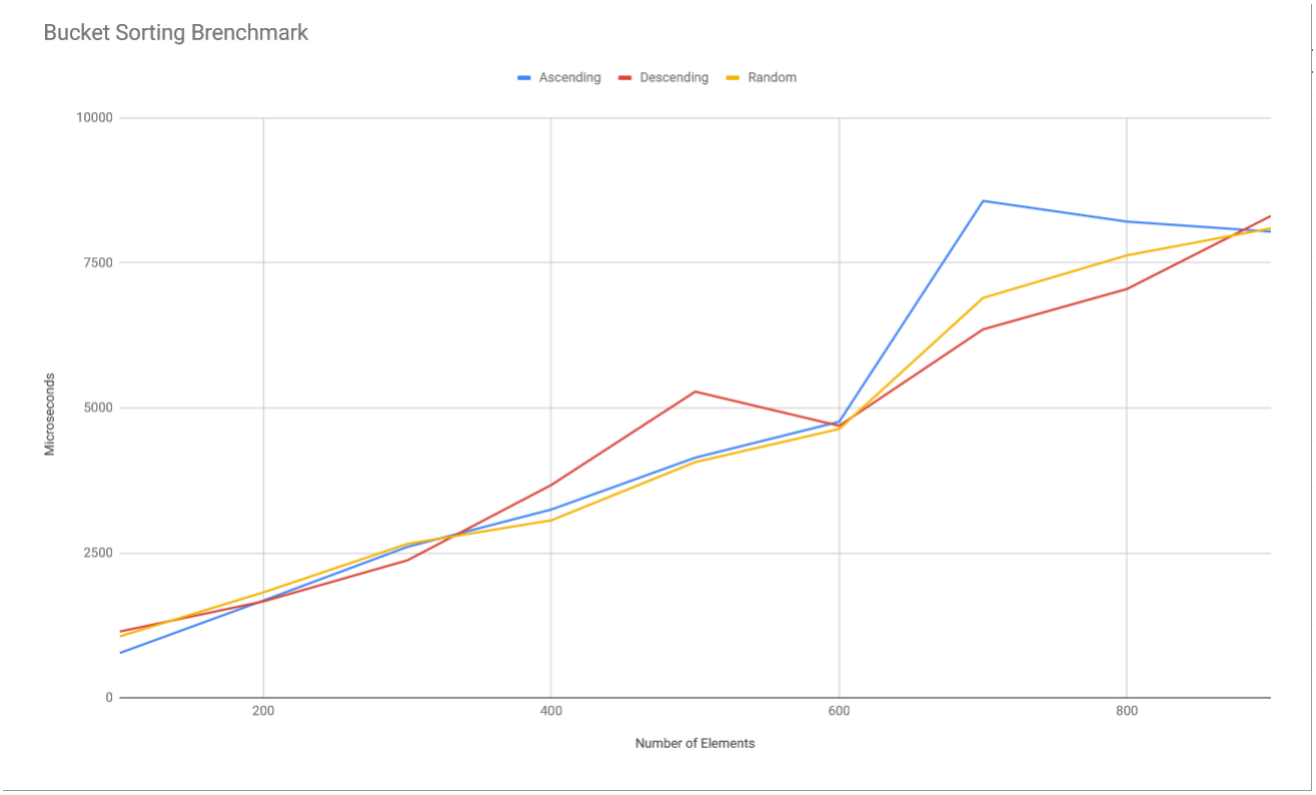
VI. BrenchMark


Counting Sorting Brenchmark
Ascending
Elements
Random
2500
2000
1500
1000
500
0
Microseconds
200
400
600
800
Number of Elements

Counting Sorting				
Elements	Best	Worst	Average	
100	237	192	198	
200	375	375	374	
300	726	539	541	
400	718	719	715	
500	899	1020	961	
600	1134	1072	1191	
700	2213	1638	1415	
800	1436	1679	1657	
900	1660	1630	1974	


Radix Sorting Brenchmark
Ascending
Descending
Random
4000
3000
2000
1000
0
Microseconds
200
400
600
800
Number of Elements

Radix Sorting				
Elements	Best	Worst	Average	
100	328	383	382	
200	637	600	827	
300	960	1004	862	
400	1294	1144	1142	
500	1525	1547	1650	
600	1985	1968	2895	
700	2222	2041	3227	
800	2552	3359	2500	
900	3193	2695	2715	


Bucket Sorting Brenchmark
Ascending
Descending
Random
10000
7500
5000
2500
0
Microseconds
200
400
600
800
Number of Elements

Bucket Sorting			
Elements	Best	Worst	Average
100	772	1143	1059
200	1679	1664	1820
300	2604	2372	2656
400	3248	3671	3062
500	4142	5280	4065
600	4761	4691	4636
700	8568	6354	6897
800	8211	7050	7630
900	8039	8309	8096