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 rápida (Quick):*

El ordenamiento rápido (quicksort en inglés) es un algoritmo basado en la técnica de divide y vencerás, que permite, en promedio, ordenar n elementos en un tiempo proporcional a n log n. Esta es la técnica de ordenamiento más rápida conocida. Fue desarrollada por C. Antony R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento (los algoritmos recursivos son en general más lentos que los iterativos, y consumen más recursos).

Para los casos en este ordenamiento tenemos:

Best case	$\Omega(n \log n)$
Worst case	$\mathcal{O}(n^2)$
Avarage case	$\theta(n \log n)$

Best case análisis: $T(n) = 2T\left(\frac{n-1}{2}\right) + \mathcal{O}(n))$

Donde el método maestro nos dice: $T(n) = \mathcal{O}(n \log n)$

Worst case análisis: $T(n) = T(n-1) + \mathcal{O}(n)$
Donde el método maestro nos dice: $T(n) = \mathcal{O}(n^2)$

Avarage case análisis: $T(n) = \mathcal{O}(n \log n)$

Implementación de código en C++:

```
void
quickSort(std::vector<int>& v, unsigned int low, unsigned int high)
{
  if(high == std::numeric_limits<int>::max())
  {
    high = v.size() - 1;
  }
  // no need to sort a vector of zero or one elements
  if(low >= high)
    return;

  // select the pivot value
  unsigned int pivotIndex = (low + high) / 2;

  // partition the vector
  pivotIndex = partition(v, low, high, pivotIndex);

  // sort the two sub vectors
  if(low < pivotIndex)
    quickSort(v, low, pivotIndex);
  if(pivotIndex < high)
    quickSort(v, pivotIndex + 1, high);
}
```

```
int
partition(std::vector<int>& v,
          unsigned int start,
          unsigned int stop,
          unsigned int position)
{
  std::swap(v[start], v[position]);

  // partition values
  unsigned int low = start + 1;
  unsigned int high = stop;
  while(low < high)
    if(v[low] < v[start])
      low++;
    else if(v[--high] < v[start])
      std::swap(v[low], v[high]);

  // then swap pivot back into place
  std::swap(v[start], v[--low]);
  return low;
}
```

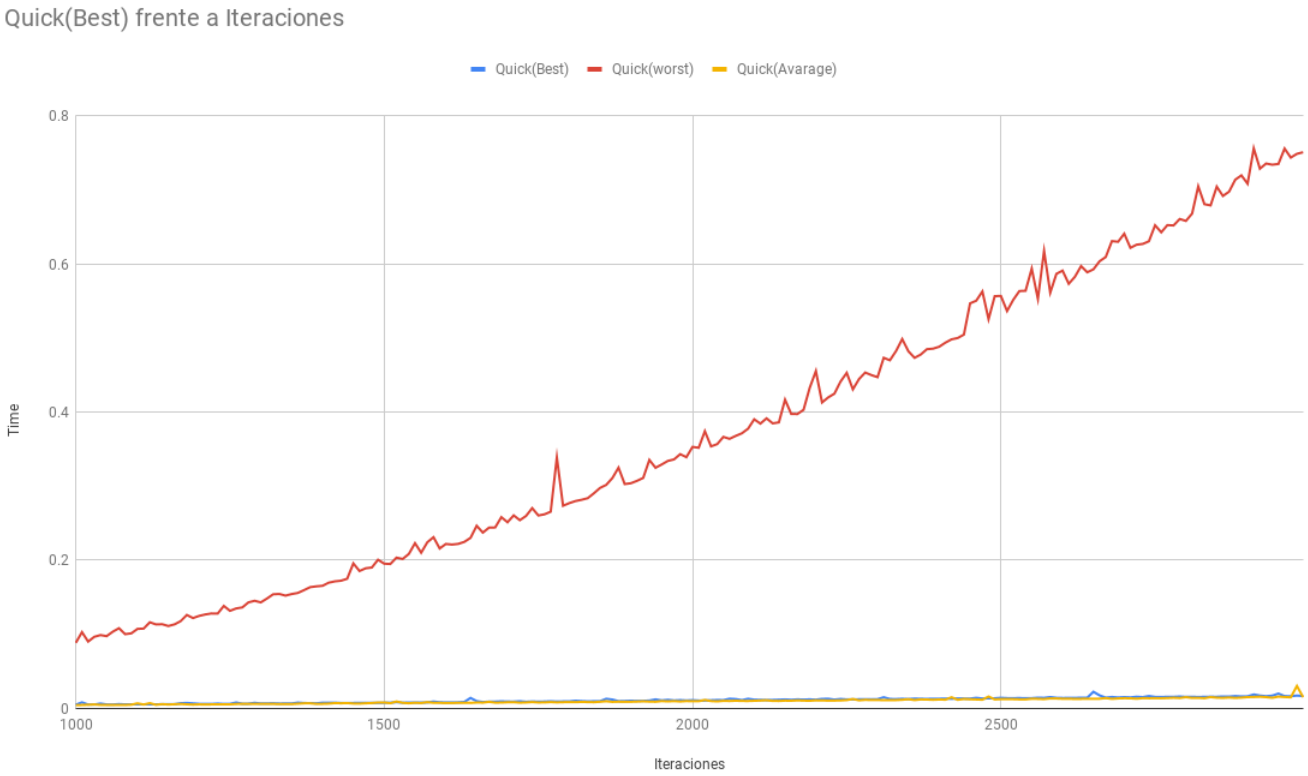

Fig. 1. Los tiempos tomados en algoritmo Quick en diferentes casos.

B. ***Solución mezcla (Merge):***

El ordenamiento por mezcla es un algoritmo recursivo que divide continuamente una lista por la mitad. Si la lista está vacía o tiene un solo ítem, se ordena por definición (el caso base). Si la lista tiene más de un ítem, dividimos la lista e invocamos recursivamente un ordenamiento por mezcla para ambas mitades. Una vez que las dos mitades están ordenadas, se realiza la operación fundamental, denominada mezcla. La mezcla es el proceso de tomar dos listas ordenadas más pequeñas y combinarlas en una sola lista nueva y ordenada

Para los casos en este ordenamiento tenemos:

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

En base al tiempo de complejidad $T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$
podemos deducir con fácilidad que en los 3 casos la complejodad se volvería:
$T(n) = \mathcal{O}(n \log n)$

Implementación de código en C++:

```
std::vector<int>
mergeSort(std::vector<int>& m)
{
  if(m.size() <= 1)
  {
    return m;
  }

  std::vector<int> left, right, result;
  int middle = ((int) m.size() + 1) / 2;

  for(int i = 0; i < middle; i++)
  {
    left.push_back(m[i]);
  }

  for(int i = middle; i < (int) m.size(); i++)
  {
    right.push_back(m[i]);
  }

  left = mergeSort(left);
  right = mergeSort(right);
  result = merge(left, right);

 m = result;

  return result;
}
```

```
std::vector<int>
merge(std::vector<int>& left, std::vector<int>& right)
{
  std::vector<int> result;
  while((int) left.size() > 0 || (int) right.size() > 0) {
    if((int) left.size() > 0 && (int) right.size() > 0) {
      if((int) left.front() <= (int) right.front()) {
        result.push_back((int) left.front());
        left.erase(left.begin());
      }
      else {
        result.push_back((int) right.front());
        right.erase(right.begin());
      }
    }
    else if((int) left.size() > 0) {
      for(int i = 0; i < (int) left.size(); i++)
        result.push_back(left[i]);
      break;
    }
    else if((int) right.size() > 0) {
      for(int i = 0; i < (int) right.size(); i++)
        result.push_back(right[i]);
      break;
    }
  }
  return result;
}
```

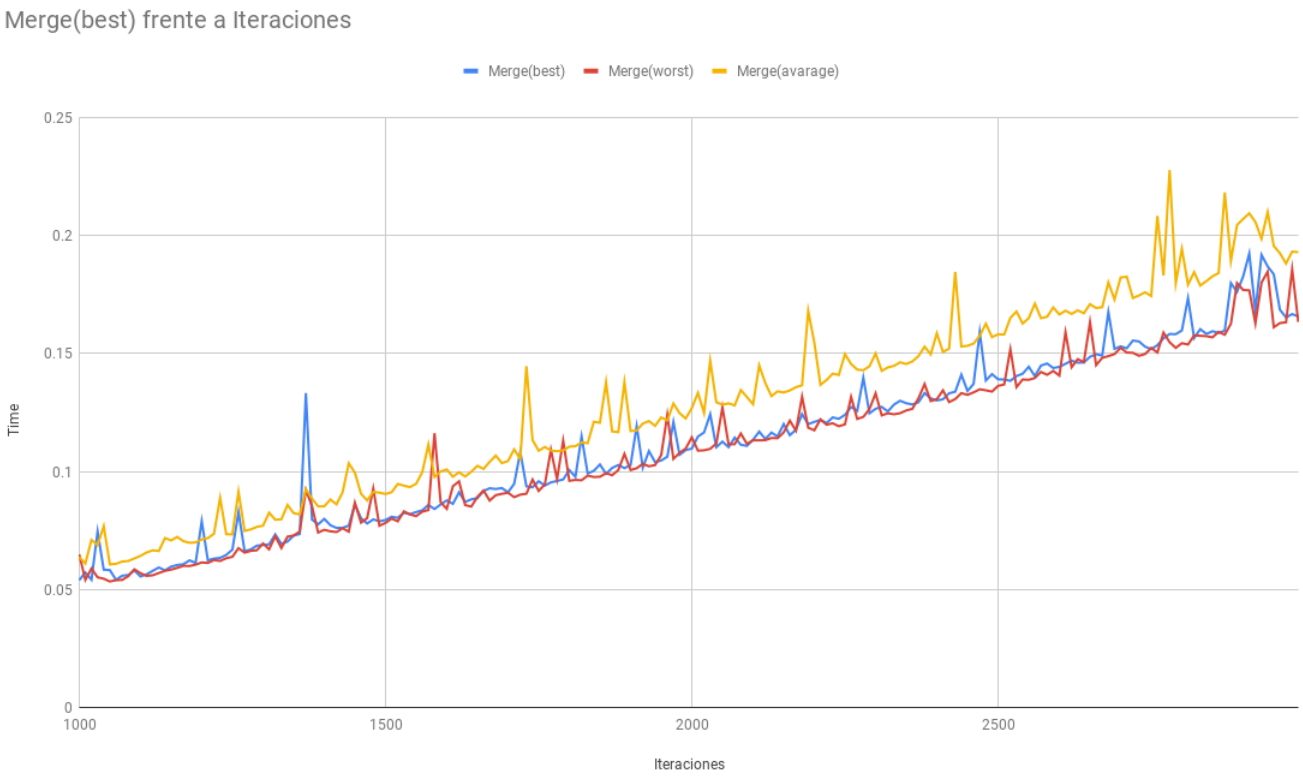

Fig. 2. Los tiempos tomados en algoritmo Merge en diferentes casos.

II. Búsqueda

Un algoritmo de búsqueda es aquel que está diseñado para localizar un elemento concreto dentro de una estructura de datos. Consiste en solucionar un problema de existencia o no de un elemento determinado en un conjunto finito de elementos, es decir, si el elemento en cuestión pertenece o no a dicho conjunto, además de su localización dentro de éste.

A. *Solución lineal (linear search):*

Es un método para encontrar un valor objetivo dentro de una lista.Ésta comprueba secuencialmente cada elemento de la lista para el valor objetivo hasta que es encontrado o hasta que todos los elementos hayan sido comparados.
Búsqueda lineal es en tiempo el peor, y marca como máximo n comparaciones, donde n es la longitud de la lista. Si la probabilidad de cada elemento para ser buscado es el mismo, entonces la búsqueda lineal tiene una media de n/2 comparaciones, pero esta media puede ser afectado si las probabilidades de búsqueda para cada elemento varían.

Para los casos en esta búsqueda tenemos:

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

Tomando en cuenta los casos, se puede concluir rápidamente que el tiempo de complejidad es $\mathcal{O}(n)$ como el mismo nombre de la búsqueda lo describe, lineal.

Implementación de código en C++:

```
int
linearSearch(std::vector<int>& vec, int x)
{
  int i = 0;
  for(auto& it : vec)
  {
    if(it == x)
    {
      return i;
    }

    ++i;
  }

  return -1;
}
}
```

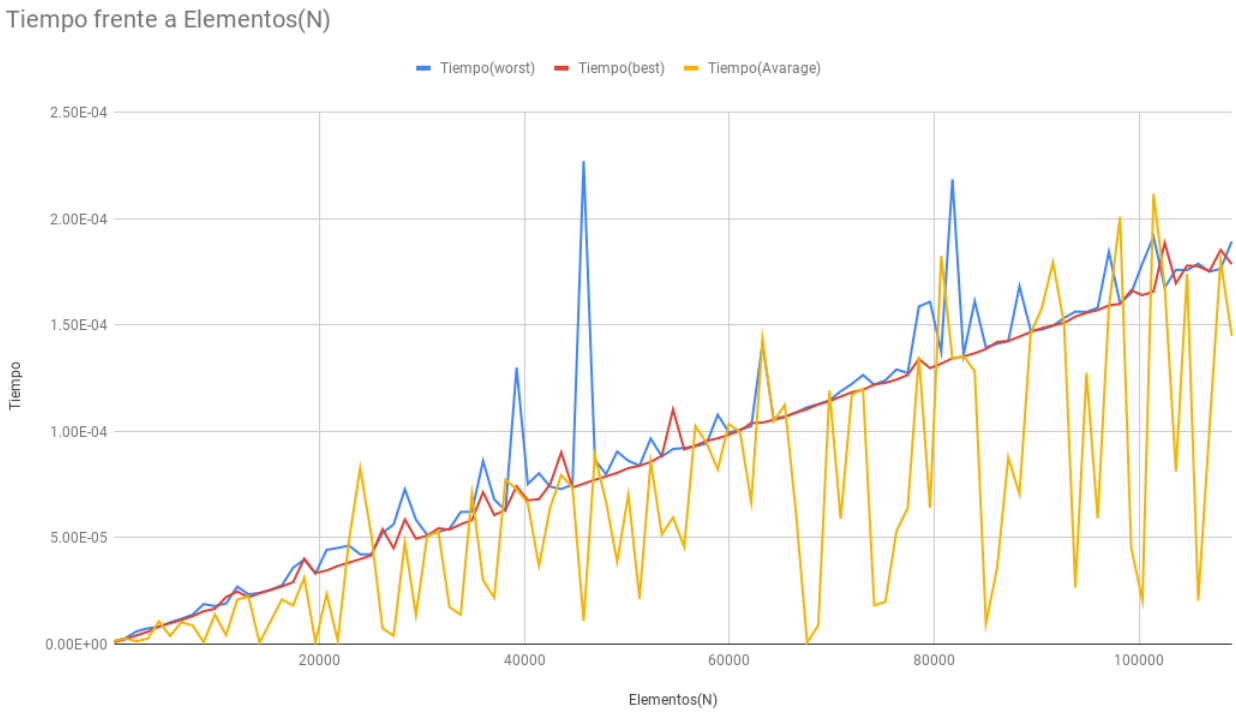

Fig. 3. Tiempos tomados en la búsqueda; siendo best ordenado ascendente; worst ordenado descendente; avarage completamente random. Esto explica el comportamiento y podemos notar facilmente el tiempo $\mathcal{O}(n)$

B. Solución binaria (binary search):

La búsqueda binaria es un algoritmo eficiente para encontrar un elemento en una lista ordenada de elementos. Funciona al dividir repetidamente a la mitad la porción de la lista que podría contener al elemento, hasta reducir las ubicaciones posibles a solo una. Usamos la búsqueda binaria en el juego de adivinar en la lección introductoria.

Para los casos en esta búsqueda tenemos:

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

Tomando en cuenta los casos, se puede concluir rápidamente que el tiempo de complejidad es $\mathcal{O}(\log n)$. Es decir, no importa el número de entrada el tiempo siempre será recurrente.

Implementación de código en C++:

```
if (right == std::numeric_limits<int>::max())
{
  right = numbers.size() - 1;
}

if(right >= left)
{
  int mid = left + (right - left) / 2;

  if(numbers[mid] == x)
    return mid;

  if(numbers[mid] > x)
    return binarySearch(numbers, x, left, mid - 1);

  return binarySearch(numbers, x, mid + 1, right);
}

return -1;
```

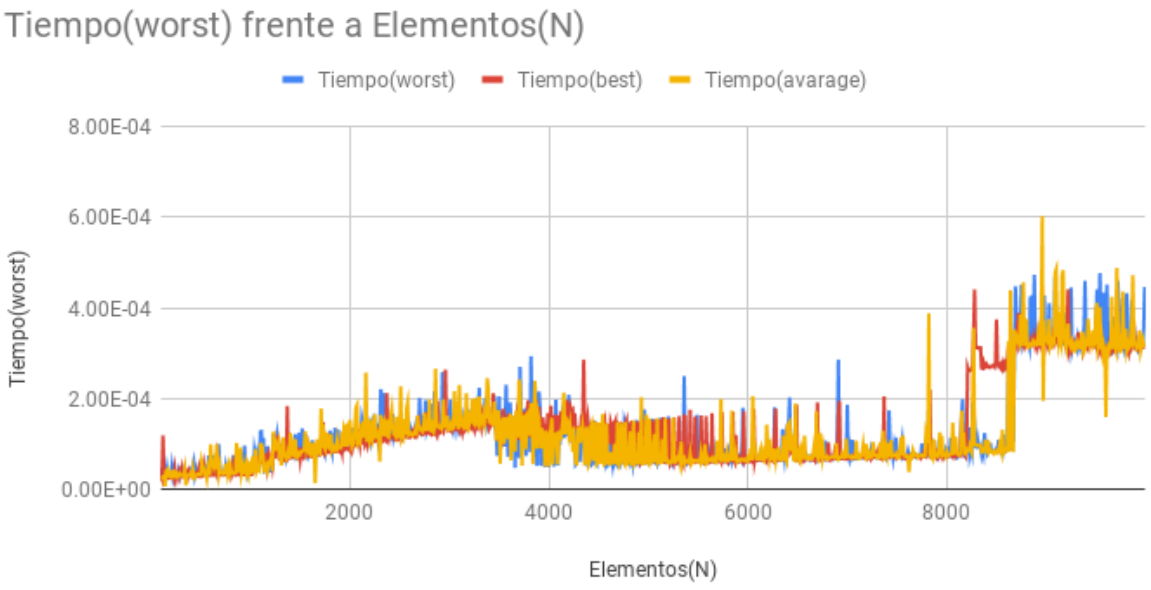

Fig. 4. Aunque no parezca obvio a simple vista, se realizaron más de 900 iteraciones, y el tiempo no varió, siempre quedó en milisegundos, sin importar el número de entrada.