75.41 Algoritmos y Programación II Curso 4
Ordenamiento

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

9 de junio de 2020

1. Introducción

Uno de los problemas más comunes en el manejo de conjunto de datos es el ordenar dichos datos. Estos algoritmos están dentro de los más comunes en la construcción de aplicaciones informáticas. Existen varias razones para estudiar los algoritmos de ordenamiento en detalle, entre ellos , saber determinar cual utilizar según el caso, poder realizar modificaciones que permitan ajustar el algoritmo al problema, entre otros.

1.1. Aspectos Generales

Antes de considerar algoritmos específicos se discutirá alguna terminología y suposiciones básicas sobre los algoritmos de ordenamiento:

- Se van a considerar métodos de ordenamiento cuyos ítems contienen **claves**. Las **claves**, que son solo parte de estos ítems, son utilizadas para controlar el ordenamiento.
- El objetivo de un método de ordenamiento es del de reorganizar los ítems de forma tal que sus claves *estén ordenadas de acuerdo a una regla bien definida(normalmente orden numérico o alfabético)*. Las característica especificas de las claves o los ítems pueden variar ampliamente según el problema, pero la noción abstracta de poner las claves y su información asociada en orden es lo que caracteriza al problema del ordenamiento.
- Si el archivo que tiene que ser ordenado cabe completamente en la memoria, entonces el método de ordenamiento será denominado **ordenamiento interno**. Si el archivo esta en disco o en memoria secundaria el método se denomina **ordenamiento externo**. La principal deferencia entre estos dos métodos es que en un método de ordenamiento interno cualquier ítem puede ser accedido fácilmente, mientras que en un método de ordenamiento externo es accedido secuencialmente o por lo menos en grandes bloques.
- Se van a considerar tanto los arreglos como las listas enlazadas. El problema de ordenar arreglos y el problema de ordenar listas enlazadas son ambos de interés: durante el proceso de desarrollo de los algoritmos hay tareas que se realizan mejor con memoria reservada secuencialmente; y otras tareas que se realizan mejor con memoria enlazada. La mayoría de los métodos clásicos de ordenamiento son los suficientemente abstractos para ser implementados eficientemente ya sea por arreglos o por listas enlazadas. Existen otros métodos que son mejores en particular en un caso o en otro.
- Los algoritmos de ordenamiento pueden diferenciarse entre:
  - **No Adaptativos** si solo usan operaciones de comparación o intercambio por ejemplo el método de inserción; la secuencia de operaciones que realizan es independiente del orden de los datos.
  - **Adaptativo** es aquel que realiza diferentes secuencias de operaciones dependiendo de las salidas de las comparaciones.
- La primera característica de estudio de un algoritmo de ordenamiento es su tiempo de ejecución. En segundo lugar la cantidad de memoria utilizada por el algoritmo de ordenamiento es el segundo lugar de importancia a considerar.

2. Algoritmos de Ordenamiento Comparativos

Estos algoritmos utilizan las relaciones comparación entre los datos para poder determinar el orden relativo entre ellos.

2.1. Quicksort

Probablemente este sea el algoritmo mas utilizado que cualquier otro. El algoritmo básico fue inventado en 1960 por C.A.R. Hoare y ha sido estudiado por muchísimas personas desde su creación.

El algoritmo de ordenamiento Quicksort tiene la deseable característica de utilizar una pequeña pila auxiliar, solo requiere alrededor de $nlogn$ operaciones en promedio para ordenar N elemento, y posee un ciclo interno extremadamente corto.

Su desventaja es que no es un algoritmo de ordenamiento estable, es decir no preserva su orden relativo si la cantidad de claves se duplica, es decir que en el peor de los casos se necesitan N^2 operaciones.

2.2. El Algoritmo Básico

El Quicksort es un algoritmo de la familia**divide y conquista**. Trabaja particionando un arreglo en dos partes, después ordena cada una de las partes independientemente. El punto crucial del método es el **proceso de partición**, el cual reordena el arreglo de forma tal que las siguientes tres condiciones se cumplan:

- el elemento a[i] esta en su lugar definitivo en el arreglo para algún valor de i.
- ninguno de los elementos a[1],...,a[i-1] es mayor que a[i].
- ninguno de los elementos en a[i+1] ... a[r] es menor que a[i].

2.2.1. Explicación abstracta del funcionamiento de QuickSort

1. Se elige un elemento v de la lista L de elementos al que se le llama pivote.
2. Se particiona la lista L en tres listas:
   - L1 - que contiene todos los elementos de L menos v que sean menores o iguales que v
   - L2 - que contiene a v
   - L3 - que contiene todos los elementos de L menos v que sean mayores o iguales que v
3. Se aplica la recursión sobre L1 y L3
4. Se unen todas las soluciones que darán forma final a la lista L finalmente ordenada. Como L1 y L3 están ya ordenados, lo único que tenemos que hacer es concatenar L1, L2 y L3

Aunque este algoritmo parece sencillo, hay que implementar los pasos 1 y 3 de forma que se favorezca la velocidad de ejecución del algoritmo.

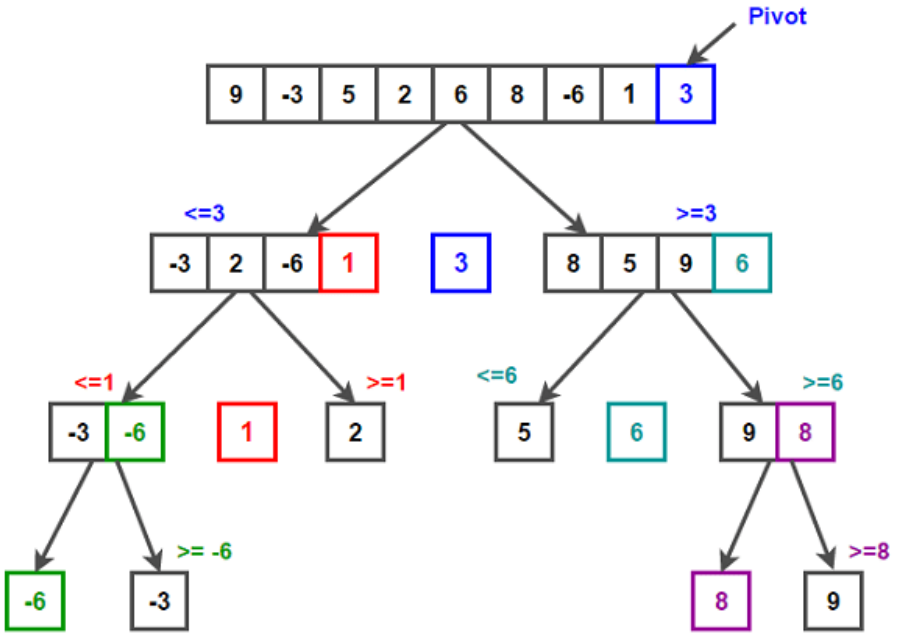

Figura 1

2.3. Implementación

```
/* C implementation QuickSort */
#include<stdio.h>

// A utility function to swap two elements
void swap(int* a, int* b) {
        int t = *a;
        *a = *b;
        *b = t;
}

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */

int partition (int arr[], int low, int high) {
        int pivot = arr[high];    // pivot
        int i = (low - 1);  // Index of smaller element

        for (int j = low; j <= high- 1; j++) {
                // If current element is smaller than or
                // equal to pivot
                if (arr[j] <= pivot){
                        i++;    // increment index of smaller element
                        swap(&arr[i], &arr[j]);
                }
        }
        swap(&arr[i + 1], &arr[high]);
        return (i + 1);
}

/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low  --> Starting index,
high  --> Ending index */
void quickSort(int arr[], int low, int high) {

        if (low < high) {
/* pi is partitioning index, arr[p] is now
at right place */
                int pi = partition(arr, low, high);

// Separately sort elements before
// partition and after partition
                quickSort(arr, low, pi - 1);
                quickSort(arr, pi + 1, high);
        }
}

/* Function to print an array */
void printArray(int arr[], int size) {
        int i;
        for (i=0; i < size; i++)
                printf("%d ", arr[i]);
        printf("n");
}

// Driver program to test above functions
```

```
int main() {
        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = sizeof(arr)/sizeof(arr[0]);
        quickSort(arr, 0, n-1);
        printf("Sorted array: n");
        printArray(arr, n);
        return 0;
}
```

2.4. Análisis de la Complejidad

Este algoritmo en el peor delos casos se comporta con una complejidad de $O(n^2)$, ese es el caso en el que la partición está mas desbalanceada, es decir, la primer partición es de $(n-1)$ elementos. Esto sucede cuando el pivot elegido es el elemento más pequeño o más grande. Con lo cual lo que sucede es que se van a hacer $(n-1)$ llamadas recursivas anidadas, si se realiza la suma de $(n-1)$ cuyo resultado es $fracn(n-1)/2$ por lo cual, en este escenario el Quicksort tiene una complejidad $O(n^2)$ en el peor de los casos.

Pero en un caso promedio en el cual el algoritmo genera particiones de $n/2$ el tiempo promedio de ordenar las sublistas es $O(n/2)$. Con este razonamiento se obtiene que la ecuación de recurrencia es:

$$T(n) = \begin{cases} n/2 & si\ n = 0 \\ 2T(n/2) + n/2 & si\ n > 1 \end{cases}$$

Para resolver esta ecuación se utiliza el Teorema Maestro, se obtiene que $a = 2$ y $b = 2$. Por ende $log_b\ a = 1$, que implica que $f(n) = n^{log_b\ a}$ siendo como válido el caso 2 del mismo teorema, $O(n\ log\ n)$

3. Merge Sort

Fue desarrollado en 1945 por John Von Neumann. Es un algoritmo de la familia divide y conquista. Trabaja particionando un arreglo en dos partes, llamándose recursivamente hasta que el arreglo tenga longitud 1 o cero. Una vez alcanzada el caso base mezcla cada arreglo de longitud 1 en forma ordenada hasta obtener el arreglo de longitud original ordenado.

3.1. El Algoritmo Básico

Dividir al encontrar el número q de la posición a medio camino entre p y r. Este paso se realiza de la misma manera en que se encuentra el punto medio en la búsqueda binaria: sumar p y r , dividir entre 2 y redondea hacia abajo.

Vencer se realiza al ordenar de manera recursiva los subarreglos en cada uno de los dos subproblemas creados por el paso de dividir. Es decir, ordenar de manera recursiva el subarreglo array[p..q] y ordenar de manera recursiva el subarreglo array[q+1..r].

Combinar al mezclar los dos subarreglos ordenados de regreso en un solo subarreglo ordenado array[p..r].

3.1.1. Explicación Abstracta del Funcionamiento de Merge Sort

Conceptualmente, el ordenamiento por mezcla funciona de la siguiente manera:

- Si la longitud de la lista es 0 ó 1, entonces ya está ordenada. En otro caso:
- Dividir la lista desordenada en dos sublistas de aproximadamente la mitad del tamaño.
- Ordenar cada sublista recursivamente aplicando el ordenamiento por mezcla.
- Mezclar las dos sublistas en una sola lista ordenada.

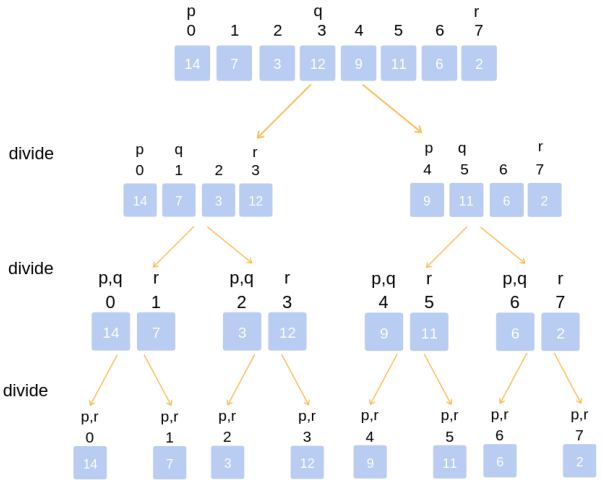

Figura 2: divide

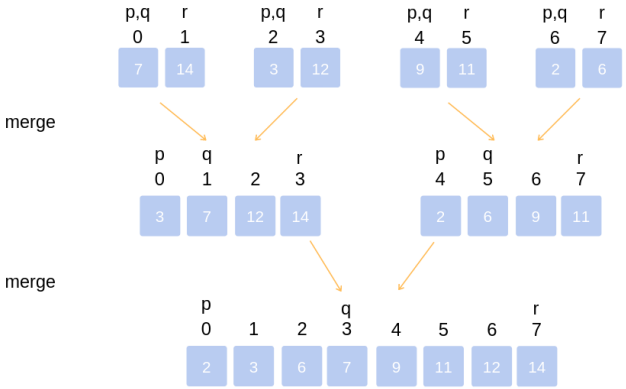

Figura 3: conquistar

3.2. Implementación

```
/ example of merge sort in C/C++
// merge function take two intervals
// one from start to mid
// second from mid+1, to end
// and merge them in sorted order

void merge(int *Arr, int start, int mid, int end) {
// create a temp array
        int temp[end - start + 1];

// crawlers for both intervals and for temp
        int i = start, j = mid+1, k = 0;

// traverse both arrays and in each iteration add smaller of both elements in temp
        while(i <= mid && j <= end) {
        if(Arr[i] <= Arr[j]) {
                temp[k] = Arr[i];
                k += 1; i += 1;
        }
        else {
                temp[k] = Arr[j];
                k += 1; j += 1;
        }
}
```

```
// add elements left in the first interval
        while(i <= mid) {
                temp[k] = Arr[i];
                k += 1; i += 1;
        }

// add elements left in the second interval
        while(j <= end) {
                temp[k] = Arr[j];
                k += 1; j += 1;
        }

// copy temp to original interval
        for(i = start; i <= end; i += 1) {
                Arr[i] = temp[i - start]
        }
}

// Arr is an array of integer type
// start and end are the starting and ending index of current interval of Arr

void mergeSort(int *Arr, int start, int end) {

        if(start < end) {
                int mid = (start + end) / 2;
                mergeSort(Arr, start, mid);
                mergeSort(Arr, mid+1, end);
                merge(Arr, start, mid, end);
        }
}
```

4. Algoritmos de Ordenamiento No Comparativos

Algoritmos que utilizan otras estrategias para logra el ordenamiento del set de datos que no son exclusivamente las comparación entre elementos.

4.1. Bucket Sort

Este es un algoritmo de ordenamiento que trabaja mediante la distribución de los elementos de un arreglo en contenedores llamados "Buckets" o baldes, posteriormente cada bucket se ordena utilizando algún método de ordenamiento. *Se utiliza cuando el set de datos esta distribuido uniformemente sobre un rango de valores* . La Idea Básica es la siguiente:

1. Se crean N buckest vacíos
2. Recorrer el arreglo original poniendo el elemento correspondiente en el bucket que lo debe contener.
3. ordenar cada bucket que no este vació.
4. Visitar los buckets en orden y poner todos los elementos nuevamente en el arreglo:

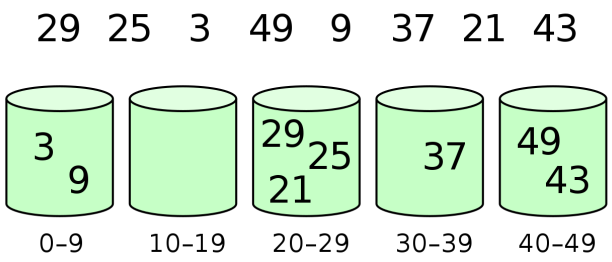

Figura 4: distribuir

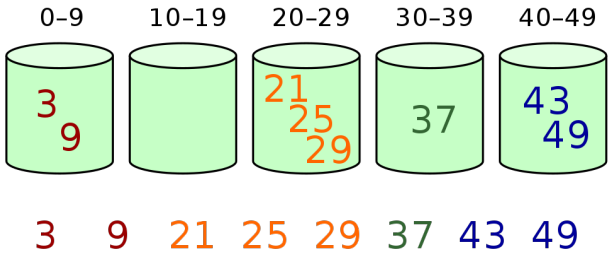

Figura 5: Agrupar, Ordenar y Juntar

4.1.1. Números Flotantes

Si los elementos a ordenar fueran número flotantes, este algoritmo también podría utilizarse. Para ello se plantea el siguiente arreglo:

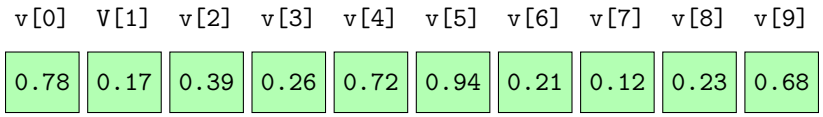

Figura 6: Arreglo v[]

El algoritmo no varia en nada de la versión anterior , excepto la forma en que se distribuyen los elementos en los buckets:

1. Se crean N buckest vacíos, en este caso $N = 10$.
2. Se recorre el arreglo y se inserta el elemento $v[i]$ en el $bucket[\lfloor N * v[i] \rfloor]$
3. Se ordenan los buckets individuales
4. se concatenan los buckets no vacíos ordenados.

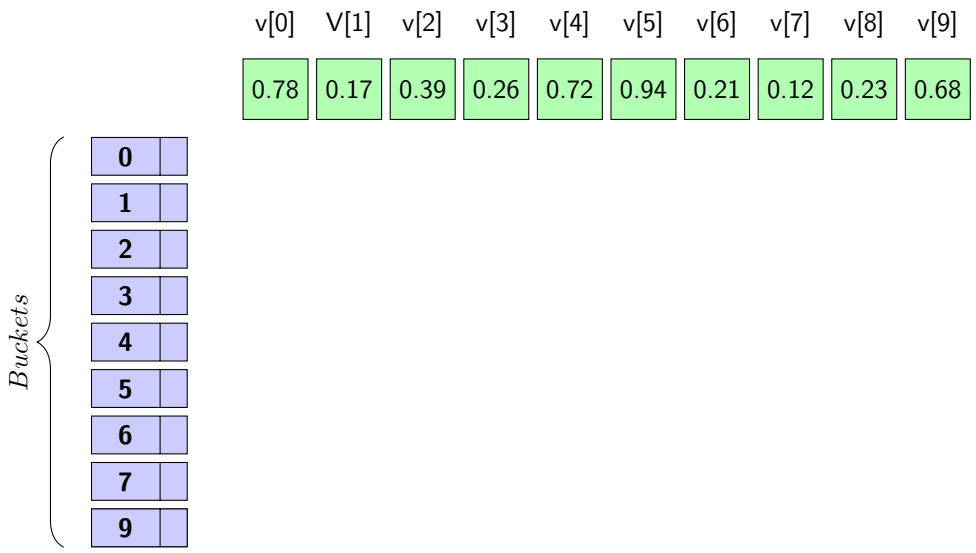

Figura 7: Arreglo v con los Buckets

A partir de esta configuración se recorre el arreglo y según la fórmula establecida $bucket[\lfloor N * v[i] \rfloor]$ se asigna cada elemento a un bucket:

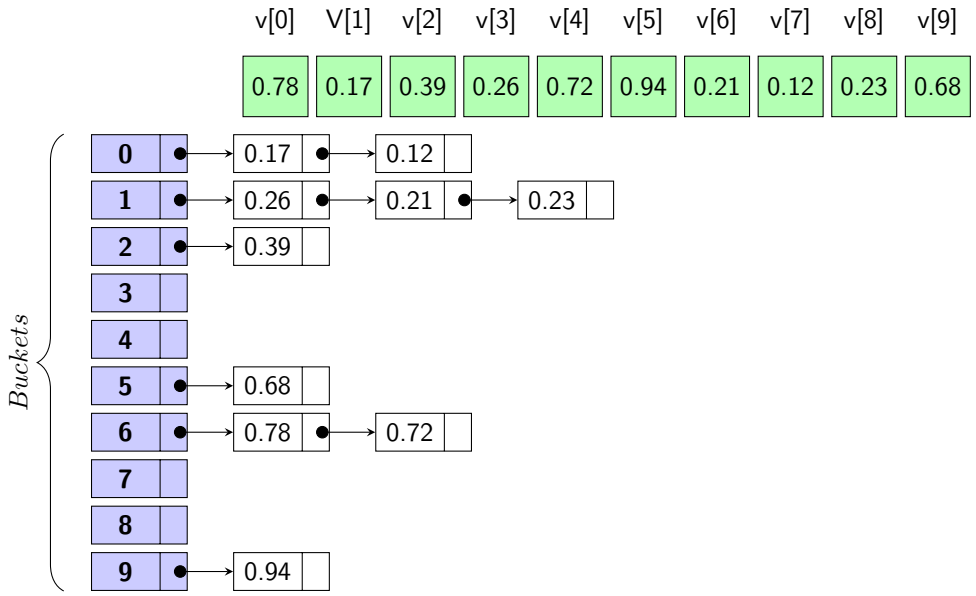

Figura 8: Disperción de los elementos del arreglo en los buckets

A continuación se recorren los buckets no vacíos y se ordenan según algún criterio:

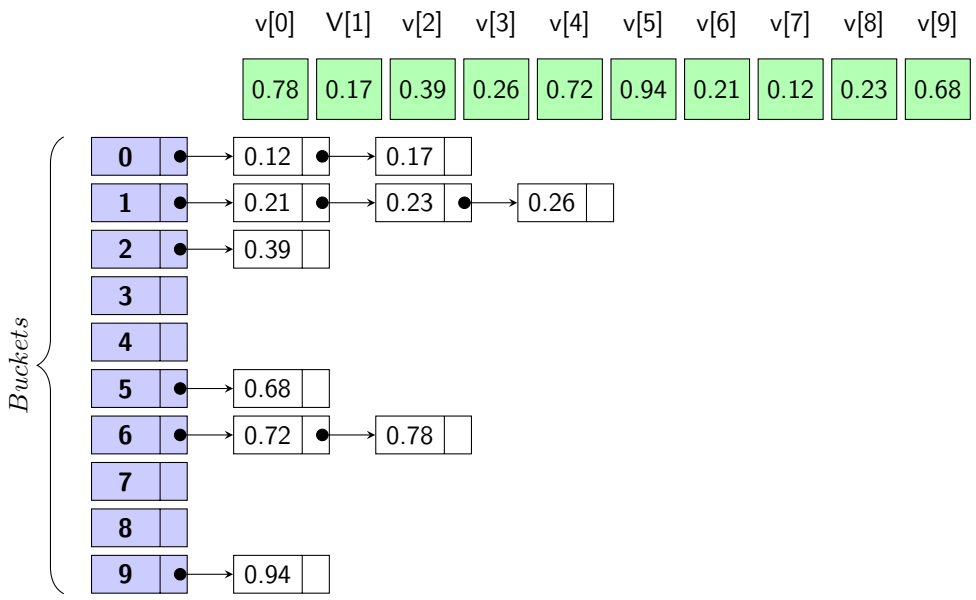

Figura 9: Ordenamiento de cada Bucket

Finalmente se recorren los buckets no vacíos y se concatenan para volver a generar el vector:

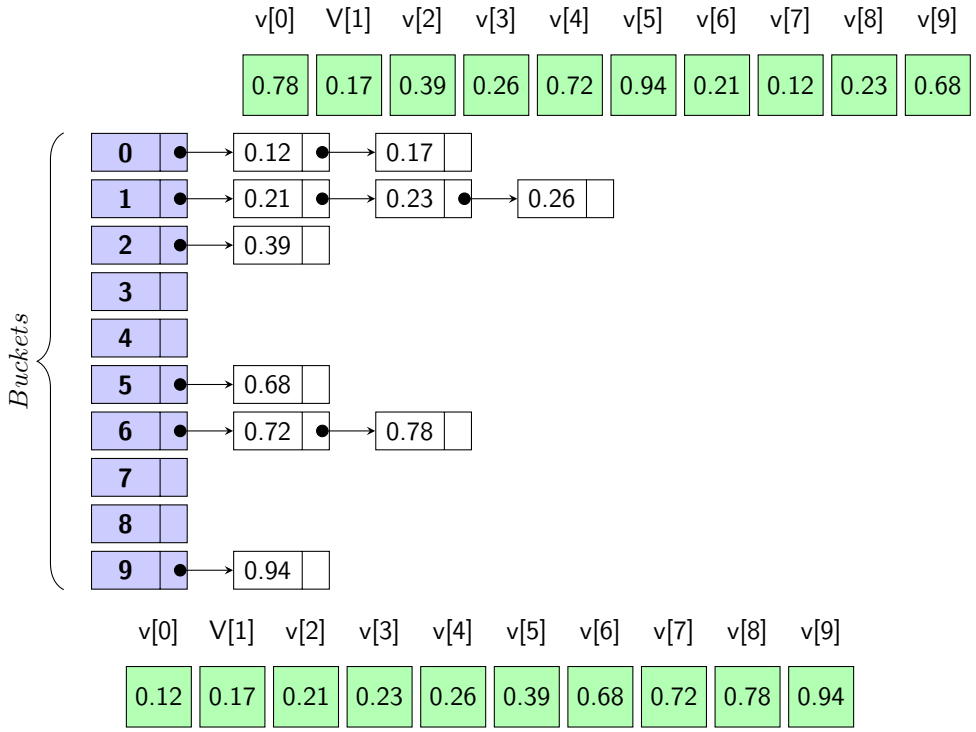

Figura 10: Se recorren los buckets no vacíos y se genera el nuevo arreglo ordenado

Finalmente el arreglo ordenado resulta:

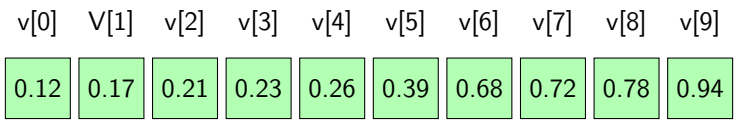

Figura 11: Lista de Adyacencia

4.1.2. Análisis de la Complejidad

Este algoritmo en el peor delos casos se comporta con una complejidad de $O(n^2)$.

4.2. Radix Sort

Este es otro método de ordenamiento no comparativo, que sirve para ordenar números ya que estos pueden representarse como cadenas de strings. La idea detrás de del método es agrupar los números según el valor de sus dígitos. Esto puede hacerse con el dígito menos significativo o el digito más significativo. Sea el vector:

v[0]	170
v[1]	045
v[2]	394
v[3]	090
v[4]	802
v[5]	066
v[6]	072
v[7]	188
v[8]	075
v[9]	002

Figura 12: Vector Desordenado

El Algoritmo se implementará partiendo del digito más significativo al más significativo.

Se realiza ordena el arreglo teniendo únicamente el primer digito menos significativo.

v[0]	170
v[1]	090
v[2]	802
v[3]	072
v[4]	002
v[5]	394
v[6]	045
v[7]	075
v[8]	066
v[9]	188

Se realiza ordena el arreglo teniendo únicamente el segundo digito menos significativo.

v[0]	002
v[1]	802
v[2]	045
v[3]	066
v[4]	072
v[5]	075
v[6]	170
v[7]	188
v[8]	394
v[9]	090

Se realiza ordena el arreglo teniendo únicamente el digito más significativo.

v[0]	002
v[1]	045
v[2]	066
v[3]	072
v[4]	075
v[5]	090
v[6]	170
v[7]	188
v[8]	394
v[9]	802

Figura 13: Vector Desordenado

4.3. Counting Sort

Referencias