UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ESTRUCTURAS DE DATOS Y ALGORITMOS I

M. en I. Fco. Javier Rodríguez García

Práctica 3

Algoritmos de ordenamiento III

<francisco.rodriguez@ingenieria.unam.edu>

Versión 1.0, 08/09/22

Tabla de contenido

1. OBJETIVOS

1.1. Objetivos generales

Al finalizar la práctica:

1. Habrás escrito en el lenguaje C una versión del algoritmo ***Radixsort*** para ordenar una lista de números enteros.
2. Habrás escrito en el lenguaje C una versión del algoritmo ***Countingsort*** para ordenar una lista de números enteros.

1.2. Atributos de egreso CACEI

Tabla 1. Atributos de egreso aplicables

Atributo	Sub-atributo	Aplicación
A1. Identificar, formular y resolver problemas de Ingeniería en Computación aplicando las ciencias básicas y los principios de la ingeniería.	**A1-CD2** Expresa el fenómeno asociado mediante un modelo matemático adecuado.	Pseudocódigos
	A1-CD3 Aplica la técnica correspondiente de la ingeniería para la resolución del problema.	Programación en un lenguaje de computadora

1.3. Resultados de aprendizaje

Tabla 2. Resultados de aprendizaje

Resultado	Sub-resultado	Aplicación
1. Conocimiento y comprensión	1.1 Conocimiento y comprensión de las matemáticas y otras ciencias básicas inherentes a su especialidad de inteniería, en un nivel que permita adquirir el resto de las competencias del título.	Algoritmos de búsqueda y programación en un lenguaje de computadora.
5. Aplicación práctica de la ingeniería	Comprensión de las técnicas aplicables y métodos de análisis, proyecto e investigación y sus limitaciones en el ámbito de su especialidad.	Diferencia en la eficiencia de algoritmos y sus aplicaciones.

2. ALGORITMO RADIXSORT

Los algoritmos que hemos estudiado hasta el momento (y muchos otros) están basados en comparar un elemento contra otro. De manera teórica lo más rápido que se puede ordenar de esta forma es **O(n log n)**.

¿Pero qué tal que existiera un algoritmo mejor, **O(kn)**, con $k << n$ (k es el *radix* o base, y n el número de elementos en la lista), que supere incluso a *Quicksort*? Si estamos dispuestos a aceptar algunas restricciones, entonces este algoritmo existe y se llama *Radixsort*. Sin embargo, para lograr este nivel de eficiencia debemos conocer sus limitaciones:

- El **rango** de los elementos a ordenar debe ser fijo, o conocerse antes de ejecutar el algoritmo:
  - Si es una lista de enteros, entonces que todos estén en el rango `[a, b]` para todas las corridas. Por ejemplo, en todas las corridas el rango debe ser `[0, 1000]`.
  - Si es una lista de palabras, entonces debe existir una longitud máxima de la palabra para todas las corridas.
- Derivado del punto anterior, puede ordenar números reales de punto fijo, pero no aquellos con mantisa. El número de decimales debe ser fijo.
- Requiere memoria extra en forma de *colas* (o en inglés, *queues*). Se necesitan tantas *colas* como indique la base (más abajo lo explico), y el tamaño de las *colas* debe ser al menos tan grande como el número de elementos de la lista. Para evitar el desperdicio de memoria vamos a utilizar *colas* basadas en *listas enlazadas*. Aunque no es la mejor implementación, es simple y nos permitirá concentrarnos en el funcionamiento del algoritmo. No obstante, podríamos usar en su lugar *arreglos dinámicos*. Ambas alternativas , eventualmente, utilizan llamadas a la función de biblioteca de C, `malloc()` (y a efectos prácticos, cualquier otro lenguaje diferente de C terminaría llamando a `malloc()`), por lo que usar a esta función no debería impedirte usar al algoritmo.

En este algoritmo a las *colas* se les dice *buckets* (en español, *cubeta*).

Sin embargo, debes saber que existe una versión *in situ*, es decir, que no usa memoria extra. Tarea, ¿cómo se llama dicha versión?

¿Para qué se usa?

A pesar de las restricciones impuestas para lograr la eficiencia ya mencionada, este algoritmo se utiliza para:

- Ordenar de manera *lexicográfica*. En un diccionario, ¿qué palabra va primero, *Rodrigo*,

Rodriguez, o *Rodrigues*?

- Si el problema a mano permite conocer *a priori* el rango numérico de la lista, entonces éste algoritmo se podría utilizar en lugar de los ya estudiados.
- Puede ser paralelizado con cierta facilidad. (En la última parte del curso veremos Algoritmos paralelos.)
- Clasificadores (tarjetas perforadas, código de barras, cartas, naipes). Esta es la razón por la cual fue inventado por Herman Hollerith.

Radixsort

El algoritmo *Radixsort* utiliza el valor posicional de las cifras individuales en cada uno de los valores de una lista para llevar a cabo el ordenamiento durante varias *rondas*. A diferencia de otros algoritmos, como ya mencioné, éste no usa comparaciones.

Radix

En español, *radix* significa *base numérica*. Si lo que queremos ordenar es una lista de números naturales, entonces su *radix* (base) será 10. Si fueran números hexadecimales, su *radix* sería 16. Si fueran palabras, entonces su *radix* sería 27 (para español y 26 para inglés). En fórmula de la *cota superior*, **O(kn)**, la *k* es el número de elementos en la base o *radix*. Por ejemplo, si el *radix* es 10, entonces la *k* es 10.

Ten en cuenta que debe existir una *cola* (o *bucket*) por cada valor en la base. Por ejemplo, para base 10 tendríamos 10 *colas*, mientras que para la base 27, serían 27.

Y van a existir tantas rondas como valores posicionales tengamos en los valores de la lista. Si el valor máximo en la lista es 630, entonces tendremos tres rondas: una para las unidades, una para las decenas y una para las centenas. Si el valor máximo fuera 8765, entonces tendremos 4 rondas: una para las unidades, otra para las decenas, otra para las centenas, y una para los millares.

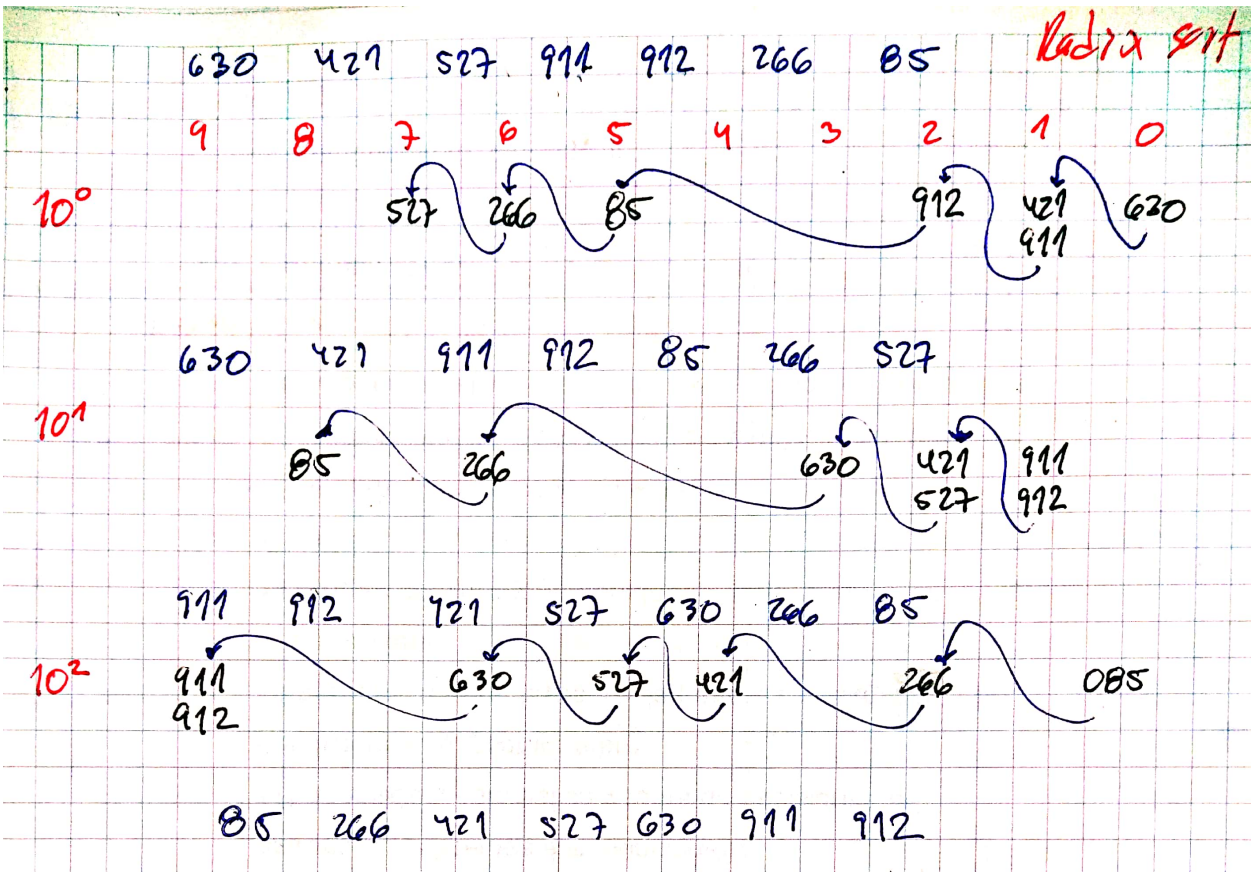
En cada ronda determinaremos en cuál cola escribir el elemento dependiendo de la cifra posicional de cada valor. Para el elemento 630 en base 10, en la primera ronda dicho valor se escribiría en la cola correspondiente a 0. Y así para todos los elemento. Al terminar dicha ronda todas las *colas* se deben *recolectar*, comenzando en la *cola* con menor valor posiciona, que en este ejemplo es la correspondiente al `0`, escribiendo los elementos en la lista original.

Durante la segunda ronda, el mismo valor se escribiría en la cola correspondiente a 3; y finalmente, en la última ronda, el valor se escribiría en la cola correspondiente a 6.

En general, después de escribir todos los valores de la lista en una ronda dada, los debemos *recolectar*, empezando por la cola 0, sacando valor por valor y escribíendolos en la lista original. Al final de la última ronda tendremos la lista ordenada en forma ascendente.

Ejemplo

Si tomamos al conjunto de números `{630, 421, 527, 911, 912, 266, 85}` para ordenarlo, la representación gráfica de las 3 rondas nos queda así:



El pseudocódigo de este algoritmo es el siguiente. El ciclo más externo se mueve sobre los valores posicionales (unidades, decenas, etc). El ciclo interno escribe en la *cola* correspondiente el elemento. Aquí la función `subKey()` determina a cuál *cola* pertenece el elemento. Por ejemplo, si estamos en la ronda de las decenas, y la cantidad es `634`, entonces la función devolverá el valor `3`, el cual corresponde a las decenas en dicha cantidad. Este valor, `3`, se guarda en la variable `whichQ`, la cual se pasa a la función para *encolar* `Enqueue()` junto con el valor a insertar. Con esto, el valor `3`, en la ronda de las decenas, se inserta en la *cola* de los `30`s.

Una vez terminado el ciclo interno el siguiente paso es la *recolección*, la cual es ejecutada por la función `Enqueue()`.

Listado 1. Pseudocódigo para el algoritmo Radix sort.

```
// list:   la lista a ordenar
// tam:    el número de elementos en la lista
// numPos: el número de valores posicionales: 10 tiene 2 valores, 100 tiene 3 valores,
//         1000 tiene 4 valores, y así sucesivamente
// radix:  la base numérica. Si en un problema en particular la base siempre será 10,
//         entonces podríamos quitar este argumento
Radixsort( list, tam, numPos, radix )
{
    Crear tantas colas, queues[], como lo indique radix

    Para i = 1 hasta numPos] {
        Para j = 0 hasta tam) {
            whichQ = subKey( list[ j ], i, radix )
            Enqueue( queues[ whichQ ], list[ j ] )
        }

        collect( list, queues, radix )
    }
}

subKey( val, pos, radix ) : Int
{
    divisor = 10 elevado a la (pos-1)
    Devuelve ( val / divisor) MOD radix
}

collect( list[], queues[], radix )
{
    index = 0
    Para i = 0 hasta radix) {
        Mientras queues[ i ] no esté vacía {
            val = Dequeue( queues[ i ] )
            list[ index ] = val
            ++index
        }
    }
}
```

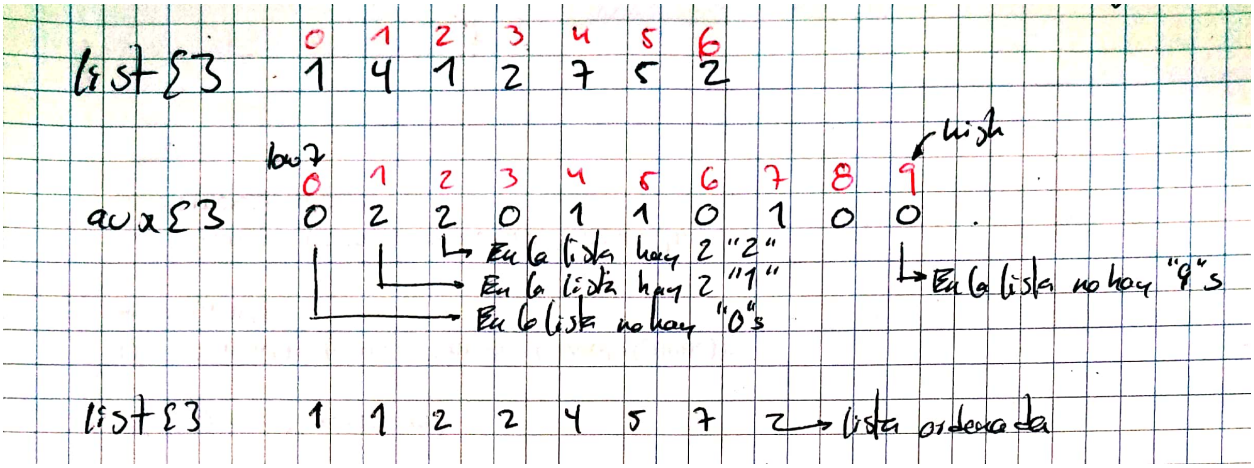
3. ALGORITMO COUNTINGSORT

Este algoritmo "cuenta" la frecuencia de aparición de los valores en la lista original a ordenar, y guarda dicha frecuencia en una lista auxiliar (y por lo tanto, no es un algoritmo *in situ*). Luego, cada índice con frecuencia diferente a cero lo escribe de regreso en la lista original tantas veces como lo indique su frecuencia.

Este algoritmo no es de uso general debido a algunos inconvenientes que presenta.

Ejemplo

El ordenamiento del conjunto de números `{1, 4, 4, 2, 7, 5, 2}` con este algoritmo nos queda así.



3.1. Versión simple (para enteros)

Listado 2. Pseudocódigo para el algoritmo Counting sort.

```
1 CountingSort( list, elems, low, high )
2 {
3    capacity = high + 1
4
5    Crea un arreglo aux[] de capacity elementos
6
7    Llena a aux[] con ceros
8
9    Para i = 0 hasta elems-1 {
10      pos = list[ i ]
11      aux[ pos ]++
12   }
13
14   j = 0
15
16   Para value = 0 hasta capacity-1 {
17      Para reps = aux[ value ] hasta 1 {
18         list[ j ] = value
19         ++j
20      }
21   }
22
23   Devuelve la memoria de aux[]
24 }
```

3.2. Versión completa (para enteros y tipos compuestos)

Listado 3. Pseudocódigo para el algoritmo CountingSort completo

```
CountingSort( list, elems, low, high ): List
{
   capacity = high + 1

   Crea un arreglo aux[] de capacity elementos

   Llena a aux[] con ceros

   Crea un arreglo output[] de elems elementos

   Para i = 0 hasta elems-1
   {
      pos = list[ i ]
      aux[ pos ] += 1
   }

   Para i = 1 hasta capacity-1
   {
      aux[ i ] += aux[ i - 1 ]
   }

   Para i = elems-1 hasta 0
   {
      j = key( list[ i ] )
      aux[ j ] -= 1
      output[ aux[ j ] ] = list[ i ]
   }

   Devuelve output
}
```

4. PRE-LABORATORIO

1. Investiga cómo utilizar al generador de números pseudo aleatorios de C para obtener valores dentro de un rango, por ejemplo, números en el rango del 5 al 25 inclusive, `[5, 25]`.
2. Escribe dos funciones en C, `max()` y `min()` que devuelvan, respectivamente, el valor máximo y mínimo de una lista. Guíate con sus firmas:

   ```
   int max( int list[], size_t elems ); // devuelve el valor máximo de la lista
   int min( int list[], size_t elems ); // devuelve el valor mínimo de la lista
   ```

3. Completa un programa funcional similar a los ya vistos para el algoritmo *Radixsort* a partir del pseudocódigo. Nómbralo radixsort.c. Utiliza el módulo `Queue` que recibiste junto con esta práctica.
4. Completa un programa funcional similar a los ya vistos para el algoritmo *Countingsort* a

partir del pseudocódigo. Nómbralo countingsort.c.

5. IN-LABORATORIO

5.1. Radixsort

1. Para compilar deberás utilizar una instrucción de compilación similar a la ya mencionada. Por ejemplo, si el archivo donde está la función `main()` se llama radixsort.c, la instrucción es:

```
gcc -Wall -std=c99 -o radix_sort.out radixsort.c Queue.c DLL.c
```

2. Crea un arreglo de 5000 elementos. Usa a la función de biblioteca `rand()` para llenarlo con números enteros aleatorios en el rango `[0, 4000)`.
3. Escribe una función de activación `Radixsort10()` que libere al usuario de los detalles de implementación de *Radixsort* para un `radix=10`, y un valor posicional equivalente al valor máximo de la lista `3999`:

```
void Radixsort10( int list[], size_t elems );
```

   Esto es, el *cliente* en la función `main()` llamará a `Radixsort10()` con únicamente la lista y su tamaño, y ésta función a su vez llamará a a `Radixsort()` con los parámetros necesarios. Para que esto funcione da por hecho que el valor máximo de la lista **nunca** estará por encima de `9999`, y que por supuesto, son la lista es de valores en base 10.

5.2. Countingsort

1. Crea un arreglo de 5000 elementos enteros. Usa a la función de biblioteca `rand()` para llenarlo con números enteros aleatorios en el rango `[5, 25]`. Para que la programación del algoritmo sea más fácil, asume que el rango comienza en 0. Esto es, las entradas del arreglo de la [0] a la [4] quedarán vacías.
2. Escribe una función de activación `CountingSort()` que libere al usuario de los detalles de implementación de `counting_sort()`:

```
void CountingSort( int list[], size_t elems );
```

   Esto es, el *cliente* en la función `main()` llamará a `CountingSort()` con únicamente la lista y

su tamaño, y ésta función a su vez llamará a a `counting_sort()` con los parámetros necesarios. La función `CountingSort()` deberá calcular el valor máximo y mínimo de la lista. Y para lograrlo deberás escribir dos funciones auxiliares:

```
int max( int list[], size_t elems ); // devuelve el valor máximo de la lista
int min( int list[], size_t elems ); // devuelve el valor mínimo de la lista
```

6. POST-LABORATORIO

6.1. RadixSort

6.1.1. General

1. Crea una variable global, `g_contador`, de tipo entero.
2. Incrementa `g_contador` inmediatamente después de la línea 14 (u operación similar en código C) en el Listado 1.
3. Incrementa `g_contador` inmediatamente después de la línea 34 (u operación similar en código C) en el Listado 1.
4. Imprime el valor de `g_contador` luego de realizar un ordenamiento.
5. Explica la eficiencia en tiempo **O(kn)** de este algoritmo.

6.1.2. Tipos compuestos

1. Crea un *tipo compuesto* sencillo:

```
typedef struct
{
   int barcode;
   float price;
   char name[32];
} Product;
```

2. Modifica tu algoritmo para que reciba una lista de productos y los ordene basado en el código de barras. Asume códigos de barras de 5 dígitos (99999).
3. En tu driver program:
   a. Crea una lista de 10 productos con códigos de barras en el rango `[1000, 100000)`.
   b. Ordena la lista utilizando el algoritmo del reactivo anterior basado en el código de

barras.

c. Imprime la lista de productos ordenada con todos sus campos.

6.2. CountingSort

1. Crea una variable global, `g_contador`, de tipo entero.
2. Incrementa `g_contador` inmediatamente después de la línea 11 (u operación similar en código C) en el Listado 2.
3. Incrementa `g_contador` inmediatamente después de la línea 19 (u operación similar en código C) en el Listado 2.
4. Imprime el valor de `g_contador` luego de realizar un ordenamiento.
5. Explica la eficiencia en tiempo **O(k+n)** de este algoritmo.
6. Modifica a la función `counting_sort()` para que sea capaz de procesar de forma eficiente listas con rangos que empiecen alejados del 0. Por ejemplo, una lista de 1000 elementos en el rango (500,1000]. Para que sea más fácil, considera que siempre serán rangos positivos.

7. Entregables

(pend)

8. Bibliografía

[JOYANES05]

Joyanes A., Luis; Zahonero M., Ignacio. *Programación en C, Metodología, algoritmos y estructura de datos*. 2da ed. Mc Graw Hill. ESPAÑA: 2005.

[KOFFMAN08]

Koffman, Elliot B., Wolfgang, Paul A. T. *Estructura de datos con C++, Objetos, abstracciones y diseño*. McGraw Hill. MÉXICO: 2008.

[NYHOFF92]

Nyhoff, Larry R. *Data structures and program design in Pascal*. 2nd ed. Macmillan Publishing. USA:1992.

[SHAFFER11]

Shaffer, Clifford A. *Data structures and algorithm analysis in C++*. 3rd ed. Dover Publications Inc. USA: 2011.

[WEISS13]

Weiss, Mark A. *Estructuras de datos en Java*. 4ta ed. Pearson Educación, S.A. SPAIN: 2013.

Referencia del lenguage C

http://www.cplusplus.com/reference/