	Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia	

Laboratorios de computación salas A y B

Profesor: M.I. EDGAR TISTA GARCÍA

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS 2

Grupo: 8

No de Práctica(s): 3] ALGORITMOS DE ORDENAMIENTO PARTE 3

Integrante(s): ADOLFO ROMÁN JIMENEZ

No. de Equipo de cómputo empleado: TRABAJO EN CASA

No. de Lista o Brigada:

Semestre: 2022 - 1

Fecha de entrega: 5 DE OCTUBRE DE 2021

CALIFICACIÓN: _____

Practica 3

Algoritmos de Ordenamiento Parte 3

Román Jimenez Adolfo

October 5, 2021

1 Objetivo

El estudiante identificará la estructura de los algoritmos de ordenamiento Counting Sort y Radix-sort

2 Objetivo de Clase

El alumno implementará casos particulares de estos algoritmos para entender mejor su funcionamiento a nivel algorítmico.

3 Desarrollo

3.1 Ejercicio 0: Menu y Auxiliares

El menu creado para el Ejercicio 0, es un menu sencillo que solamente incluye las opciones de los algoritmos de ordenamiento a que la practica solicita.

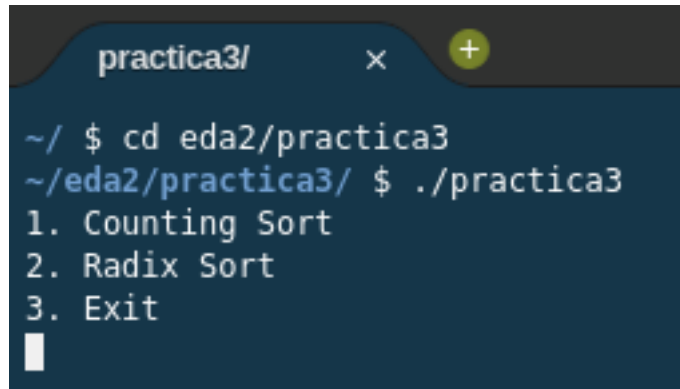
El menu contiene las opciones:

- Counting Sort
- Radix
- Exit

Este menu esta contenido dentro del archivo `options.c` bajo la funcion `menu` y contiene como argumento un apuntador que almacenara la informacion obtenida por el usuario en una variable entera externa `option` para escoger la opcion solicitada por el usuario al momento de probar el algoritmo deseado.

El menu esta contenido dentro de un *while-loop* que rompe el ciclo hasta que el usuario escoge una de las 3 opciones numericas validas, mientras no lo haga, este seguira ejecutandose.

Al momento de escoger una opcion valida, esta se almacena dentro de una variable interna `n` que despues se deposita dentro del apuntador.



```
practica3/ × +
~/ $ cd eda2/practica3
~/eda2/practica3/ $ ./practica3
1. Counting Sort
2. Radix Sort
3. Exit
█
```

Ejercicio 0: Menu

3.2 Ejercicio 1: Counting Sort

Para programar el algoritmo de ordenamiento *CountingSort()* me base directamente en el pseudocodigo contenido dentro del video de la clase y lo trate de transcribir de la mejor forma posible a como ahi se mostraba, aunque tuve que agregar algunos detalles para que mi codigo funcionara de forma mas eficiente.

El algoritmo de *CountingSort()*, se encuentra dentro del archivo `counting.c` y consta de 4 funciones que estan ordenadas en forma descendente a como aparecen en la siguiente lista:

- *findMax()*
- *fillZeros()*
- *restore()*
- *countingSort()*

De forma general, *CountingSort()* obtiene el arreglo a ordenar y utiliza la funcion *findMax()* para iterar dentro de este arreglo y obtener el valor mayor del arreglo. Despues, crea un arreglo de la longitud de ese valor maximo y utiliza *fillZeros()* para igualar todas las posiciones dentro del arreglo creado a 0's. Acto seguido, el programa itera sobre el arreglo originar para contabilizar el numero de veces que se repiten los elementos en este y registrar ese numero de veces dentro del nuevo arreglo creado, para despues sumar los elementos de este arreglo de forma recursiva conforme a cada una de las casillas del arreglo y por ultimo usa la funcion *restore()* para reescribir los elementos ordenados en un nuevo arreglo previamente creado.

A continuacion se ofrece una explicacion mas detallada sobre el funcionamiento de cada una de las funciones del algoritmo por orden de uso.

3.2.1 optionOne()

Despues de que se escoge la opcion deseada, dentro de la funcion *main()* el codigo se dirige hace un operador ternario que evalua el valor de **option** y cuando el valor es 1, entonces se dirige a ejecutar la funcion *optionOne()*.

La funcion *optionOne()* esta contenida dentro del archivo `options.c` y es la que proporciona todos las variables necesarias a *countingSort()* para operar.

Como el ejercicio solicita que *CountingSort* opere con una arreglo de 20 elementos, primeramente se declara una variable de tipo entero y se le asigna un valor de 20 que es el número de elementos que el ejercicio recomienda para la práctica. Acto seguido, se declaran dos arreglos de caracteres estos y estos arreglos tendrán el tamaño de la variable `size`, por lo que podrán almacenar un

máximo de 20 caracteres cada uno.

El primer arreglo lleva por nombre **arr** en este arreglo se ingresarán os caracteres que se pretende ordenar, el segundo arreglo lleva por nombre **sorted** y es dentro de este arreglo donde se escribirán los caracteres ya ordenados.

Después de esto se invoca a la función *charGet()* que toma dos argumentos. El primero un arreglo de caracteres y el segundo, una variable de tipo entero. La función *charGet()* primeramente imprime una leyenda que invita al usuario a ingresar los 20 caracteres que pretende ordenar. Después de esto se declara una variable de tipo **char** y acto seguido se activa un *for-loop* el cual comienza en 0 y termina hasta que itera hasta la ultima posicion del indice que es menor a 20. El *for-loop* utiliza la función *getChar()* que deposita en la variable **a** y despues evalúa si la variable no incluye un salto de línea; si lo incluye, entonces de se deshace del salto de línea y pasa al siguiente elemento que finalmente deposita la variable contenida en el indice correspondiente al arreglo.

Cuándo la funcion alcanza un total de 20 iteraciones finaliza su ejecución.

Después de esto se utiliza una función para imprimir el arreglo de la cual hablaremos después y se procede a ingresar a la funcion del algoritmo *countingSort()*.

3.2.2 countingSort()

El algoritmo *countingSort()* usa tres argumentos los cuales son: el arreglo de caracteres donde se encuentran los elementos en desorden, una variable de tipo entero que contiene la longitud del arreglo y otro arreglo de caracteres que es dónde se ordenarán los elementos.

La función comienza declarando un arreglo de dos enteros que servirá para depositar el valor minimo y maximo, el cual constituria el rango de los elementos del arreglo. Asimismo, declara una variable de tipo entero **elem** qué después funcionara como una variable auxiliar.

Después de esto, el programa invoca a la función *findMax()* y cuando está finaliza ya se encuentran depositados en el arreglo auxiliar, dentro de el índice 0, el rango y en el índice 1, el valor mínimo en el arreglo.

Después de esto se declara otro arreglo de tipo entero **count** y qué tiene posee una longitud del valor contenido en la variable **max** la cual representa el total rango entre los elementos mínimo y máximo del arreglo original que pretendemos ordenar.

Dado esto se procede a llenar con ceros las posiciones del nuevo arreglo por medio de la funcion *fillZeros()* para su uso posterior.

Una vez que el arreglo ha sido llenado con ceros, se declara un *for-loop* que itera desde el número 0 hasta 1 antes de llegar al tamaño del arreglo, esto quiere decir que sus ciclos irán desde el índice 0 hasta el 19 que son los índices donde se contienen los caracteres de el arreglo que pretendemos ordenar. Ya dentro de este *for-loop* se toma cada uno de los caracteres en **arr** y se les hace un *casting* a tipo entero que convierte su valor a código ASCII.

Después se resta ese valor en código ASCII al valor contenido en **min** y esto da como resultado el índice al que el elemento corresponde dentro del arreglo **count** por lo que se procede a sumar 1 a ese índice y es de esta forma cómo se registra que la frecuencia los caracteres existentes dentro en el arreglo a ordenar. Esto se hace con todos los demás elementos y así se obtiene una “lista” de los elementos existentes y su frecuencia.

Cuando este proceso termina Se imprime el arreglo.

Después de esto se procede a iterar dentro de los elementos del arreglo **count**; se comienza desde el índice número 1 hasta 1 antes del valor en **max**, no hay que olvidar que el valor en **max** corresponde al rango de los elementos, por lo que esto es igual a la longitud del arreglo en **count**. Ya dentro del ciclo se suma el valor del elemento anterior con el elemento siguiente comenzando desde el primer índice y así sucesivamente hasta llegar al último, lo que nos indica el valor de los índices donde se depositaran los elementos ya ordenados al momento de escribirlos es el arreglo auxiliar previamente creado.

Cuando este proceso finaliza entonces se declara un *for-loop* que lea el arreglo **arr** pero ahora desde el último elemento hasta el primero y qué será en dónde se terminen de ordenar los caracteres de forma ascendente.

Una vez que *countingSort()* finaliza, sale hacia el *scope* de *optionOne()* y es ahí cuando se imprime la salida del archivo pero ahora en **sorted** que muestra los elementos de caracteres ya ordenados.

3.2.3 findMax()

Dentro de la función *findMax*, se evalúan tanto el mayor elemento dentro del arreglo como el de menor valor.

findMax utiliza 3 argumentos, que son: un arreglo de caracteres que es el que contiene a **arr** y los elementos que pretendemos ordenar, un apuntador que “apunta” hacia los índices de memoria del arreglo depositado un nivel arriba en *countingSort()* y una variable de tipo entero **size** que incluye el tamaño del arreglo **arr**.

Primeramente se declaran las variables enteras **max** y **min** y se les asigna el valor en el índice 0 de **arr**, por medio de un *casting* que convierte el valor del carácter de **ASCII** a **int**.

Finalizado esto, se itera dentro de **arr** y cada carácter bajo el que se itera se deposita su valor entero en una variable **b** la cual evalúa si el valor en curso es mayor que **max** o menor que **min** según sea el caso, y asigna los valores a cada una de las variables en caso de que la respuesta haya sido afirmativa.

Cuando se termina de iterar, se resta el valor de **max** menos el de **min** que nos da el valor del rango del arreglo y se deposita en el índice 0 del arreglo auxiliar **r** y de igual manera se deposita el valor en **min** dentro del índice 1 del arreglo auxiliar **r**

Es aquí donde finaliza la función.

3.2.4 fillZeroes()

La función *fillZeroes()* integra dos argumentos que son un apuntador de tipo entero y una variable entera **size** que contiene el tamaño del arreglo y lo único que hace es que por medio del apuntador, opera un ciclo que finaliza en 1 menos que **size** y a través del cual se deposita el número 0 a cada uno de los elementos del arreglo.

La función finaliza cuando el ciclo *for* finaliza.

3.2.5 restore()

La función *restore()* toma 4 argumentos para su funcionamiento: un argumento de tipo entero, un arreglo de tipo entero **count** que proporciona los índices, el arreglo auxiliar donde se imprimirán los elementos ya ordenados, el carácter que esté en ese momento iterando en el *loop* que funciona al final de *countingSort()* y el valor en *min*

3.3 Ejecucion

Aquí se adjuntan las capturas de pantalla del funcionamiento de *countingSort()*

Cada una de las pantallas muestran los elementos ingresados por el usuario en dos distintas formas, la impresión de esos elementos a pantalla, el primer arreglo bajo el cual se obtienen la frecuencia de los elementos, el segundo arreglo donde se indican los índices y para finalizar el arreglo ordenado de forma alfabética-ascendente.

```

1. Counting Sort
2. Radix Sort
3. Exit
1
Ingrese 20 caracteres:
C
B
A
C
F
C
G
D
J
I
D
H
H
E
A
J
B
B
F
C
C B A C F C G D J I D H H E A J B B F C
2 3 4 2 1 2 1 2 1 2
2 5 9 11 12 14 15 17 18 20
A A B B B C C C C D D E F F G H H I J J
1. Counting Sort
2. Radix Sort
3. Exit

```

Ejercicio 1: countingSort() ejecutandose


```
~/eda2/practica3/ $ ./practica3
1. Counting Sort
2. Radix Sort
3. Exit
1
Ingresa 20 caracteres:
IHIDDBDFHJGIGDEFDBJA
I H I D D B D F H J G I G D E F D B J A
1 2 0 5 1 2 2 2 3 2
1 3 3 8 9 11 13 15 18 20
A B B D D D D E F F G G H H I I I J J
```

Ejercicio 1: countingSort() ejecutandose

3.4 Ejercicio 2: Radix Sort

Programar el algoritmo de *radixSort()* fue muy entretenido por la forma en que este ordena los elementos y cual es el proceso que usa para que todos ellos queden ordenados en forma ascendente.

Este ejercicio se hizo tomando en cuenta que se ingresarian al menos 1 secuencia de 15 elementos de enteros cuyo rango seria desde 1111 hasta 4444 usando unicamente los digitos 1,2,3 y 4 para cada uno de ellos, lo que nos da un total de 4^4 o 256 opciones diferentes para ingresar como elementos a este algoritmo o si repetimos elementos, entonces puede existir una infinidad.

Una de las caracteristas de *radixSort()* asi mismo, es que (al menos esta version en particular) no usa funciones alternativas para hacer el trabajo, sino que todo se declara dentro de la propia funcion.

En los siguientes incisos, describiremos la forma en que este algoritmo funciona en este programa.

3.4.1 optionTwo()

Para comenzar a preparar el terreno para *radixSort()* se comienza por ingresar a funcion la *optionTwo()* cuando se escoje la opcion 2 del menu principal.

Ya dentro de la funcion, declaramos una variable entera **size** que servira para ingresar la longitud del arreglo que el usuario desee crear y se imprime una leyenda solicitando que el usuario ingrese una cantidad minima de datos a ingresar que sean de al menos 15.

Despues de esto, se escanea el dato ingresado dentro de un *while-loop* que no se rompe sino hasta que el usuario ingresa un valor mayor o igual a 15 y cuando esto sucede se procede a imprimir otro mensaje en pantalla que solicita ingresar los elementos en funcion del tamano solicitado anteriormente. Una vez que se obtienen todos los elementos, se imprime el arreglo y se procede a ingresar a la funcion que contiene el algoritmo *radixSort()*.

3.4.2 radixSort()

La funcion *radixSort()* usa dos argumentos que son, un arreglo de numeros enteros **arr** que es donde se encuentran los datos a ordenar y el tamano de tal arreglo que ya fue proporcionado por el usuario.

La funcion *radixSort()* que yo programe inicia con una variable entera **last** que se inicializa en 4 y que rompera terminara con el algoritmo cuando su valor sea 0.

Despues se abre un *do-while loop* que utiliza memoria dinamica para declarar los arreglos **arrOne**, **arrTwo**, **arrThree** y **arrFour** con un tamano de 4 bytes cada uno.

Asi mismo se declaran 5 variables enteras que son **one**, **two**, **three**, **four** y **digit** que se inicialzan en 0 cada una.

Asi, se comienza un *for-loop* que va desde 0 hasta **size** y se ingresa a un *switch* que utiliza a **last** como argumento y contiene 4 casos.

La variable **last** existe para que en este *switch* tome el digito correspondiente a cada elemento del arreglo, por lo que al valer 4 inicialmente, significa que tomara las unidades de las cifras a ordenar, cuando valga 3 tomara el digito de las decenas, 2 el de las centenas y 1 el de los miles.

Asi cada caso dentro del *switch* corresponde a la forma en que lograra esto y en el caso 4 lo unico que hara para obtener las unidades de la cifra sera obtener el **% 10** de la cifra que proporcionara el primer digito. Para obtener el segundo digito en el momento en que **last** valga 3, entonces calculara el **% 100** que proporcionara los ultimos dos digitos del numero (decenas y unidades), dividira entre 10 lo que dara un numero flotante (con decimal) y al final se hara un *casting* a variable entera que nos proporcionara unicamente la cifra entera del numero flotante y asi obtendremos el digito para las decenas.

El mismo proceso para las centenas pero en este caso el modulo se calcula sobre **1000** y la division sobre **100**.

Para el caso de los miles, unicamente se divide el elemento entre **1000** y se obtiene la parte entera de la misma forma.

El dígito obtenido en el proceso se deposita en la variable **digit**,

Después la variable **digit** se deposita en un segundo *switch* que provee de acceso a cada uno de los arreglos dinámicos creados.

Dependiendo del número que **digit** vaya acarreado, ingresa al arreglo correspondiente. Si **digit** vale 1, entonces ingresa al área de **arrOne**, si vale 2 al área de **arrTwo** y así dependiendo del caso.

En el momento en que se ingresa a cualquier caso, se encuentra la variable entera correspondiente a cada uno **one**, **two**, **three** o **four** y se aumenta el valor de la variable correspondiente al momento de ingresar, inmediatamente después, el arreglo pertinente *resizes* (se hace de un nuevo tamaño) con la ayuda de *realloc()* que se multiplica por el valor que tenga la variable y se procede a depositar a la cifra correspondiente en el arreglo en el índice de la variable entera menos 1.

Esto sucede hasta que el *for-loop* inicial itera sobre todos los elementos del arreglo a ordenar y por lo tanto, deposita cada uno de los elementos en los arreglos correspondientes al dígito con el que se este trabajando.

Después el programa ingresa a una serie de *loops* que están destinados a reacomodar todos los elementos en su arreglo original **arr** y esto sucede por medio de los valores en las variables enteras **one**, **two**, **three** y **four**.

El primer *loop* comienza en 0 hasta menos de **one** para extraer todos los elementos de **arrOne**, una vez que finaliza, ingresa al segundo *loop* pero ahora este comienza desde el valor en **one** hasta el valor de **one + two**, pero al momento de tomar el elemento del arreglo **arrTwo** se hace desde el índice **i - (one + two)** que da el valor **0** y después **1**, **2** y así sucesivamente mientras va progresando el *loop*. De esta forma se pueden obtener los elementos desde el índice 0 en el arreglo temporal, pero depositarlos en el índice correspondiente del arreglo original y que no se sobrescriban.

Esto sucede a través de otros 2 *loops* hasta que terminan de iterar.

Acto seguido se resta 1 a **last** y se procede a usar la función *free()* para liberar la memoria de cada uno de los arreglos creados y por último se imprime el arreglo con el nuevo orden numérico como la práctica lo requiere.

Así el proceso se repite hasta que **last** es igual a 0 y cuando llega a este valor, significa que ya ha iterado sobre los 4 dígitos que componen los números y el arreglo está ordenado.

Por último se imprime el arreglo ordenado y se sale de la función.

3.4.3 Ejecucion

En la siguiente captura se muestra como se ejecuta *radixSort()* con una serie de 15 cifras que no estan ordenadas.

Se imprime cada arreglo modificado desde el originalmente no esta ordenado, hasta el que ya contiene todos los elementos en orden.

```
~/eda2/practica3/ $ ./practica3
1. Counting Sort
2. Radix Sort
3. Exit
2
Ingresa un tamano (minimo 15) para el arreglo:
15
Ingresa elementos a ordenar:

3122
3144
3334
2334
3322
1411
3113
1243
2211
4213
3211
4332
2432
1212
2412

3122 3144 3334 2334 3322 1411 3113 1243 2211 4213 3211 4332 2432 1212 2412
1411 2211 3211 3122 3322 4332 2432 1212 2412 3113 1243 4213 3144 3334 2334
1411 2211 3211 1212 2412 3113 4213 3122 3322 4332 2432 3334 2334 1243 3144
3113 3122 3144 2211 3211 1212 4213 1243 3322 4332 3334 2334 1411 2412 2432
1212 1243 1411 2211 2334 2412 2432 3113 3122 3144 3211 3322 3334 4213 4332
```

Ejercicio 3: Funcionamiento radixSort()

4 Conclusion

Algo que me gusto mucho de esta practica fue el hecho de que pude programar las funciones de los algoritmos y uno de ellos de tal forma que no tuve que buscar mucho sobre su funcionamiento en codigo como tal, sino desde el ejemplo de la prueba de escritorio en los videos me pude imaginar como seria el funcionamiento (principalmente de Radix) y a partir de ahi comence a idear una forma de programar todo el sistema para que el algoritmo pudiera resultar en algo que proporcionara una respuesta correcta.

Probablemente esta version de Radix no sea la aceptada o correcta pues tiene en general muchos *bugs* ya que fue hecha pensando en una prueba especifica de acuerdo a la practica, pero no esta hecha "a prueba de balas" aun.

Estuve pensando en crear mi propio algoritmo de ordenamiento, pero desafortunadamente me di cuenta que ya fue inventado. Originalmente pense que se parecia mucho a *Counting Sort* pero despues vi que el que es practicamente lo mismo que lo que yo pense se llama *Bucket/Bin Sort*, pero algo que me he dado cuenta es que uno de los problemas es que el arreglo en el que inicialmente se implementa la informacion es estatico, cuando este deberia de ser dinamico, de tal forma que al momento que eliminas informacion del arreglo original, creas otro nodo en el arreglo ordenado y asi la complejidad espacial es constante y la temporal puede llegar a ser lineal, pero el problema es la forma en la que los archivos se ingresan primeramente.

Esa idea se me ocurrio cuando vi el video de un hombre que ordenaba frutas pasandolas por un puente de tubos que se va cerrando y de esa manera van las frutas cayendo automaticamente en funcion de su tamano.

Si se pudiera implementar un sistema de ordenamiento asi seria muy eficiente.

Me parece asi, que se cumplen los objetivos de la clase sobre implementacion de estos algoritmos y el conocimiento de estos.

Gracias por leer mi practica!



Hombre ordenando frutas por medio de un puente de acuerdo a su tamaño