MEMORIA - PRÁCTICA 1
Ingeniería algorítmica y ordenación

Fecha de entrega:
15 - Octubre - 2023

Autores y NIP:
Pilar Fierro Casanova (819658)
Óscar Brizuela García (820773)

Profesores:
Elvira Mayordomo
Ubaldo Ramón

Grado:
Cuarto curso de Ingeniería Informática

Asignatura:
Algoritmia para problemas difíciles

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Universidad
Zaragoza

RESUMEN

Este documento recoge el trabajo realizado para llevar a cabo la práctica 1 de la asignatura de Algoritmia para Problemas Difíciles, donde se cuenta con una descripción de los aspectos más significativos del código fuente junto con una explicación de las pruebas realizadas y los resultados obtenidos.

Se ha abordado la implementación de tres algoritmos de ordenación de vectores: QuickSort, RadixSort y BubbleSort. Cada uno de ellos emplea una técnica de ordenación distinta, lo que implica diferentes implementaciones y costes de tiempo.

Por otro lado, las pruebas realizadas han sido tanto con conjuntos de datos generados aleatoriamente como con datos de casos reales. Los resultados obtenidos muestran la eficacia y rendimiento de cada algoritmo.

Por último, se recoge una pequeña explicación de la organización de las tareas entre los integrantes del equipo y la información de los recursos de fuentes externas empleadas.

ÍNDICE

1. CÓDIGO FUENTE ENTREGADO

Antes de comentar en detalle la implementación del código fuente proporcionado, es necesario recordar que el objetivo de la práctica es la implementación de tres algoritmos de ordenación de vectores. En este caso, además de implementar QuickSort y RadixSort tal y como se solicitaba en el guión de la práctica, se ha elegido BubbleSort como algoritmo opcional.

El algoritmo QuickSort está basado en la técnica divide y vencerás, permitiendo en promedio ordenar *n* elementos con una complejidad algorítmica temporal del orden de *O(n*log(n))*. Su funcionamiento consiste en escoger un elemento de dicho conjunto y emplearlo como pivote, dividiendo el problema en subproblemas más pequeños en función de si es menor o mayor que dicho pivote. El coste de ordenación varía en función de qué criterio se emplea para escoger dicho pivote y si el vector se encuentra inicialmente ya ordenado.

Respecto al algoritmo RadixSort, éste ordena un conjunto de *n* enteros procesando sus dígitos de manera individual comenzando por el menos significativo. Esta lista se va reorganizando en base al dígito que se ha comprobado colocando el entero en la posición correspondiente. Este proceso se repite de manera consecutiva hasta realizarse con todos los dígitos de todos los elementos del conjunto de datos. La complejidad algorítmica para este algoritmo en promedio es del orden de *O(d * n)*, siendo *d* el número de dígitos del elemento con más dígitos del vector y *n* el número de elementos del vector.

Por último, se ha optado por la implementación del algoritmo de BubbleSort como el tercero, el cual ordena mediante repetidas comparaciones del elemento visitado con los elementos contiguos. Como se puede intuir, existen algoritmos de ordenación más óptimos que este, pues el hecho de que se haga estas sucesivas comparaciones lo lleva a tener una complejidad algorítmica temporal del orden de $O(n^2)$.

1.1. Ficheros fuente entregados

- **bubbleSort.cpp**: fichero con la función correspondiente a la implementación del algoritmo de BubbleSort.
- **dnis_a_txt.txt**: fichero en el cual cada línea es un identificador de persona física (DNI ó NIE). Estos identificadores están solamente compuestos por los números del identificador original, puesto que en el caso de los DNIs se ha eliminado la letra del final y en el caso de los NIEs se han eliminado las letras tanto del final como del principio.
- **dnis.py**: fichero que lee el fichero .txt de entrada correspondiente con el contenido íntegro del PDF que se ha utilizado como fuente de datos real, extrae los IDs (DNIs y NIEs) y los escribe en un fichero de formato .txt, a razón de un ID por línea.
- **ejecutar1.sh**: script que contiene el código correspondiente para compilar y ejecutar de manera automática.
- **funcionarios.pdf**: documento pdf del que se han obtenido los DNIs y NIEs para las pruebas con casos reales
- **includes.hpp**: fichero con los includes necesarios y comunes en el resto en su gran mayoría.
- **main_reals.cpp**: fichero principal con las funciones necesarias para la ejecución de los 3 algoritmos en función del tamaño de los vectores con todos los casos con datos reales. Por otro lado, cuenta con las estructuras de datos y las funciones necesarias para extraer los diferentes IDs del fichero "dnis_a_txt.txt" y utilizarlos posteriormente para poder ser ordenados.

- **main.cpp**: fichero principal, que se ejecuta en primer lugar tras la ejecución del script de Bash. En este fichero se encuentran las funciones necesarias para ejecutar los 3 algoritmos con todos sus casos posibles y haciendo crecer el tamaño de los vectores a ordenar, así como las funciones necesarias para escribir los tiempos de ejecución en los ficheros de salida y sus respectivas medias. Por último, también se encuentra el código necesario para cerrar los ficheros previamente abiertos y para sacar por pantalla pequeños ejemplos del funcionamiento de los 3 algoritmos con casos normales y de entradas limitadas.
- **quickSort.cpp**: fichero con las funciones necesarias para la implementación del algoritmo de QuickSort. Además de estas funciones, también se implementan en este fichero todas las funciones de distintas particiones necesarias para elegir posteriormente distintos pivotes, así como las funciones para generar vectores con determinados elementos. Por último, también se implementa en este fichero una función que realiza QuickSort para ordenar un vector de manera decreciente, que será posteriormente utilizada en uno de los casos peores de este algoritmo.
- **radixSort.cpp**: fichero con las funciones necesarias para la implementación del algoritmo de RadixSort
- **utils.cpp**: fichero con funciones auxiliares para conseguir el funcionamiento de la práctica. Aquí se implementan las funciones, que serán llamadas en el fichero principal, necesarias para ejecutar los 3 diferentes algoritmos en base a distintos casos. Además, aquí también se encuentran implementadas otras funciones requeridas por el main para la creación de ficheros de entrada y salida o la salida por pantalla de resultados, con el objetivo de que el usuario pueda realizar los experimentos de forma sencilla e interactiva.
- **LEEME.txt**: fichero con una breve descripción de cómo se encuentra organizado el código y qué se puede o no hacer.

2. IMPLEMENTACIÓN DE LOS ALGORITMOS

2.1. QuickSort

En el fichero *quickSort.cpp* se encuentran las funciones que permiten conseguir el comportamiento de dicho algoritmo ya explicado anteriormente.

En primer lugar se cuentan con unas funciones auxiliares que permiten completar un vector con elementos aleatorios, con un elemento concreto y buscar el elemento mediano de un vector tomando como referencia el primer elemento, el último y el de la posición de en medio.

```
11  void generateRandomIntArray(int array[], int n) {
12      srand(time(NULL));              // Genera una semilla aleatoria
13      for (int i = 0; i < n; i++) {   // Recorre el array
14          array[i] = rand() % 101;    // Genera un número aleatorio entre 0 y 100
15      }
16  }
```

Figura 1. Función *generateRandomIntArray* que completa el array con n elementos aleatorios

```
22  void generateIntArrayWithSameElements(int array[], int n, int element) {
23      for (int i = 0; i < n; i++) {   // Recorre el array
24          array[i] = element;         // Completa todos los componentes con el entero "element"
25      }
26  }
```

Figura 2. Función *generateIntArrayWithSameElements* que completa el array con n elementos con valor *element*

```
32  int findMedian(int arr[], int start, int end) {
33      int middle = start + (end - start) / 2;                 //
34      int elements[3] = {arr[start], arr[middle], arr[end]};  //
35
36      if (elements[0] > elements[1]) {        // Si el elemento
37          swap(elements[0], elements[1]);     // Se intercambiar
38      }
39      if (elements[1] > elements[2]) {        // Si el elemento
40          swap(elements[1], elements[2]);     // Se intercambiar
41      }
42      if (elements[0] > elements[1]) {        // Si el elemento
43          swap(elements[0], elements[1]);     // Se intercambiar
44      }
45
46      return elements[1];                     // Devuelve el me
47  }
```

Figura 3. Función *findMedian* que busca el emento medio entre el primer elemento, el segundo y el de la posición en medio del vector.

Este algoritmo emplea un elemento del vector como pivote. La elección de dicho pivote es un aspecto crucial para su rendimiento ya que junto con el vector a ordenar se pueden dar varios casos, lo que influye en su coste de ordenación. Entre las opciones de elección de pivote implementadas, se encuentran:

Las que emplean el elemento de más a la izquierda, coincidiendo con la posición *start*:

```
116  int partitionLeftMost(int arr[], int start, int end) {
117
118      int pivot = arr[start];             // Primer elemento como pivote
119      int k = end;                        // Se guarda en una variable el último elemento del array
120      for (int i = end; i > start; i--) { // Se recorre el array de derecha a izquierda
121          if (arr[i] > pivot){            // Si el elemento es mayor que el pivote
122              swap(arr[i], arr[k--]);     // Se intercambia el elemento con el último elemento del array
123          }
124      }
125      swap(arr[start], arr[k]);           // Se intercambia el pivote con el último elemento del array
126      return k;                           // Se devuelve el índice del pivote
127  }
```

Figura4. Función *partitionLeftMost* que realiza la partición del vector empleando como pivote el elemento situado más a la izquierda coincidiendo con la posición *start*.

Las que emplean el último elemento, coincidiendo con la posición *end*:

```
135  int partitionRightMost(int arr[], int start, int end) {
136      int pivot = arr[end];               // Se obtiene el pivote como el último elemento del array
137      int i = (start - 1);                // Se obtiene el índice del elemento más pequeño
138
139      for (int j = start; j <= end - 1; j++) { // Se recorre el array de izquierda a derecha
140          if (arr[j] <= pivot) {          // Si el elemento es menor o igual que el pivote
141              i++;                        // Se incrementa el índice del elemento más pequeño
142              swap(arr[i], arr[j]);       // Se intercambia el elemento más pequeño con el elemento actual
143          }
144      }
145      swap(arr[i + 1], arr[end]);         // Se intercambia el pivote con el elemento más pequeño
146      return (i + 1);                     // Se devuelve el índice del elemento más pequeño
147  }
```

Figura 5. Función *partitionRightMost* que realiza la partición del vector empleando como pivote el elemento situado más a la derecha coincidiendo con la posición *end*.

Las que emplean el elemento del medio:

```
155  int partitionMiddlePivot(int arr[], int start, int end) {
156      int middle = start + (end - start) / 2;     // Se obtiene el índice del elemento que está en el medio
157      int pivot = arr[middle];                    // Se obtiene el pivote como el elemento del medio
158      return particion(arr, start, end, pivot);   // Se devuelve el índice del pivote
159
160  }
```

Figura 6. Función *partitionRightMost* que realiza la partición del vector empleando como pivote el elemento situado en el medio.

Las que emplean el elemento mediano entre el primer elemento, el último y el del medio:

```
168  int partitionMedianPivot(int arr[], int start, int end) {
169
170      int leftmost = arr[start];              // Se obtiene el elemento más a la izquierda
171      int rightmost = arr[end];               // Se obtiene el elemento más a la derecha
172      int middle = start + (end - start) / 2; // Se obtiene el elemento que está en el medio
173      int middleElement = arr[middle];        // Se obtiene el elemento que está en el medio
174      int pivot, i = start, j = end;          // Se declara el pivote, el índice i y el índice j
175
176      pivot = findMedian(arr, start, end);    // Se obtiene el pivote como el elemento mediano en
177      return particion(arr, start, end, pivot);   // Se devuelve el índice del pivote
178  }
```

Figura 7. Función *partitioMedianPivot* que realiza la partición del vector empleando como pivote el elemento situado en el medio.

Las que emplean un elemento aleatorio:

```
101  int partition_r(int arr[], int start, int end)
102  {
103      srand(time(NULL));                          // Genera una semilla aleatoria
104      int random = start + rand() % (end - start);    // Genera un número aleatorio entre start y end
105      swap(arr[random], arr[end]);                // Intercambia el elemento en posición aleatoria
106
107      return partition_random(arr, start, end);       // Devuelve el índice del pivote
108  }
```

Figura 8. Función *partition_r* que realiza la partición del vector empleando como pivote un elemento aleatorio del vector.

Para contar con un código más ordenado en el fichero, se ha implementado una función que en función del valor *opción* del parámetro realizará uno de estos casos de elección de pivote:

```
204  void quickSort(int arr[], int start, int end, int opcion) {
205      if (start < end){
206          int pivotIndex;
207          if(opcion == 1){                                    // Opción 1
208              pivotIndex = partitionLeftMost(arr, start, end);
209              quickSort(arr, start, pivotIndex - 1, opcion);
210              quickSort(arr, pivotIndex + 1, end, opcion);
211
212          }else if(opcion == 2){                              // Opción 2
213              pivotIndex = partitionRightMost(arr, start, end);
214              quickSort(arr, start, pivotIndex - 1, opcion);
215              quickSort(arr, pivotIndex + 1, end, opcion);
216
217          }else if(opcion == 3){                              // Opción 3
218              pivotIndex = partitionMiddlePivot(arr, start, end);
219              quickSort(arr, start, pivotIndex - 1, opcion);
220              quickSort(arr, pivotIndex, end, opcion);
221
222          }else if(opcion == 4){                             // Opción 4
223              pivotIndex = partitionMedianPivot(arr, start, end);
224              quickSort(arr, start, pivotIndex - 1, opcion);
225              quickSort(arr, pivotIndex, end, opcion);
226
227          }else if(opcion == 5){                             // Opción 5
228              pivotIndex = partition_r(arr, start, end);
229              quickSort(arr, start, pivotIndex - 1, opcion);
230              quickSort(arr, pivotIndex + 1, end, opcion);
231          }else{
232              cout << "Opcion no valida" << endl;
233          }
234      }
235  }
```

Figura 9. Función *quickSort* que realiza las llamadas a las funciones correspondientes en función del tipo de elección de pivote deseado.

Por otro lado, se cuenta con una función que realiza el algoritmo de Quicksort con un vector ordenado decrecientemente, esto permite poder realizar posteriormente las pruebas correspondientes, recogidas en este documento en el apartado correspondiente (apartado 3.5)

```
241  void quickSortReverse(int arr[], int start, int end) {
242      if (start < end) {
243          int pivotIndex = partitionForReversed(arr, start, end); // Se
244          quickSortReverse(arr, start, pivotIndex - 1);    // Se ordena
245          quickSortReverse(arr, pivotIndex + 1, end);      // Se ordena
246      }
247  }
```

Figura 10. Función *quickSortReverse* que realiza el algoritmo con un vector ordenado de manera decreciente.

2.2. RadixSort,

Este algoritmo ordena el vector de enteros procesando los dígitos de manera individual. Para ello se cuenta con 3 funciones que facilitan su implementación:

```
11  int generateRandomNumber(int n) {
12      int min = 1;
13      for (int i = 1; i < n; i++) {
14          min *= 10;
15      }
16      int max = min * 10 - 1;
17      return min + rand() % (max - min + 1);
18  }
```

Figura 11. Función *generateRandomNumber* que permite generar un número aleatorio con n dígitos.

Con el siguiente algoritmo podemos obtener el elemento máximo del vector, y por ello se tendrá como referencia a la hora de saber cuántos dígitos como máximo habrá que procesar para ordenar el vector.

```
24  int getMax(int arr[], int n) {
25      int mx = arr[0];
26      for (int i = 1; i < n; i++){
27          if (arr[i] > mx){
28              mx = arr[i];
29          }
30      }
31      return mx;
32  }
```

Figura 12. Función *getMax* que permite encontrar el elemento máximo de dicho vector

La siguiente función permite realizar la ordenación del array en función del exponente.

```
37  void countSort(int arr[], int n, int exp) {
38      int output[n];                      // Array de salida
39      int i, count[10] = { 0 };           // Índice "i" y ar
40
41      for (i = 0; i < n; i++){            // Almacena el núm
42          count[(arr[i] / exp) % 10]++;
43      }
44
45      for (i = 1; i < 10; i++){           // Cambia "count[i
46          count[i] += count[i - 1];
47      }
48
49      for (i = n - 1; i >= 0; i--) {      // Construye el ar
50          output[count[(arr[i] / exp) % 10] - 1] = arr[i];
51          count[(arr[i] / exp) % 10]--;
52      }
53
54      for (i = 0; i < n; i++){            // Copia el array
55          arr[i] = output[i];
56      }
57
58  }
```

Figura 13. Función *countSort* que ordena el array según el exponente

Por último la función principal de este algoritmo, que recorre el vector de manera recursiva siguiendo el método de ordenación de quicksort.

```
64  void radixsort(int arr[], int n) {
65      int m = getMax(arr, n);             // Encue
66
67      // En vez de pasar el número de dígito, se p
68      for (int exp = 1; m / exp > 0; exp *= 10){ /
69          countSort(arr, n, exp);
70      }
71  }
```

Figura 14. Función *radixSort*

2.3. BubbleSort

Para la implementación de este algoritmo se cuenta con una función que tiene como parámetros un vector de enteros *arr* y un entero *n*. Su funcionamiento se basa en recorrer el vector de manera iterativa comparando los elementos contiguos e intercambiando su posición si el elemento de la derecha es menor que el de la izquierda.

Si se realiza dicho intercambio de posiciones se cambia el valor de la variable booleana a *true* y en caso de que en alguna iteración no se encuentre a *true* se finaliza la ejecución de la función ya que el vector estará ordenado.

```
12  void bubbleSort(int arr[], int n) {
13      int i, j;
14      bool swapped;
15      for (i = 0; i < n - 1; i++) {
16          swapped = false;
17          for (j = 0; j < n - i - 1; j++) {
18              if (arr[j] > arr[j + 1]) {
19                  swap(arr[j], arr[j + 1]);
20                  swapped = true;
21              }
22          }
23
24          if (swapped == false){
25              break;
26          }
27      }
28  }
```

Figura 15. Función *bubbleSort* del fichero *bubbleSort.cpp*

3. ANÁLISIS DE LOS RESULTADOS

Para realizar las pruebas, se han realizado con vectores de datos enteros, unos generados aleatoriamente y otros obtenidos de casos reales.

La representación y almacenamiento de los datos de los vectores de enteros generados aleatoriamente como la función generateRandomIntArray del fichero quickSort.cpp el cual de almacenan en un arreglo en memoria y la variable array[] se pasa a la función por parámetro como puntero, es decir, se modifica directamente en memoria donde se encuentra almacenado el vector. En el main principal se declara el vector con el tamaño deseado para posteriormente realizar la llamada a la función.

Para el caso de los datos reales, se ha empleado como fuente de datos un documento en formato PDF del Ministerio de Trabajo, Migraciones y Seguridad Social del Gobierno de España en el que se encuentran todas las personas admitidas en el Cuerpo General Auxiliar de la Administración del Estado en 2018. En este caso a diferencia de los vectores con datos generados aleatoriamente, se declaran estáticamente en el fichero main_reals.cpp, es decir, se fija el tamaño del vector inicialmente y es más eficiente en cuanto a la asignación de memoria.

A continuación se describen en detalle las pruebas realizadas con cada uno de los algoritmos:

3.1. QuickSort

3.1.1. Casos peores

Las modalidades del algoritmo “QuickSort” y las correspondientes entradas que se han elegido para la realización de los experimentos con los casos peores han sido los siguientes:

- Emplear como pivote el elemento de más a la izquierda y un vector de enteros ya ordenado como entrada. En este caso, tras la partición, tendremos dos vectores extremadamente desequilibrados, pues uno no tendrá ningún elemento mientras que el otro tendrá todos los demás (n - 1, puesto que el pivote no corresponde a ninguno de los dos subvectores).

- Emplear como pivote el elemento de más a la derecha y un vector de enteros ya ordenado de forma decreciente como entrada. En este caso, tras la partición, tendremos de nuevo dos vectores extremadamente desequilibrados, pues uno tendrá n - 1 elementos (por la misma razón que en el caso peor anterior) mientras que el otro subvector no tendrá ningún elemento.

- Independientemente de la elección del pivote, tener como entrada un vector con todos los elementos iguales. En este escenario, el elemento del pivote no puede dividir el vector original de entrada en dos subvectores, lo que hace que la complejidad temporal de este algoritmo aumente significativamente.

En los 3 casos peores la complejidad es del orden de $O(n^2)$.

3.1.2. Caso medio

Los casos medios de este algoritmo tienen lugar cuando el vector de entrada con elementos aleatorios distribuidos de forma relativamente equitativa, como es en la mayoría de los casos. Por tanto, la elección del pivote (el elemento de más a la izquierda, el de en medio o el de más a la derecha) no tiene tanta importancia, siempre que la implementación sea la correcta. En estos casos, la complejidad algorítmica de QuickSort en cuanto al tiempo es del orden de O(n * log(n)).

Cabe destacar que el elemento que se ha elegido para los experimentos en estos "casos medios" ha sido el elemento de más a la izquierda.

3.1.3. Casos con nuevas versiones del algoritmo

Además de experimentos en los que se presenta el escenario para que aparezcan los casos peores y medios de este algoritmo, también se han realizado experimentos con distintas versiones de "quickSort". En estas versiones se cambia la forma en la que se escoge el pivote para realizar la partición de las 3 siguientes maneras:

- Utilizar como pivote el elemento del medio
- Utilizar como pivote el elemento mediano entre el elemento de más a la izquierda, el de más a la derecha y el del medio
- Utilizar como pivote un elemento aleatorio

En estas 3 nuevas versiones, la complejidad temporal media también es del orden de O(n * log(n)), siempre y cuando el vector de entrada tenga elementos aleatorios distribuidos de forma relativamente equitativa, como se ha llevado a cabo en los experimentos realizados.

3.2. RadixSort:

3.2.1. Caso peor:

El peor caso utilizando el algoritmo de RadixSort tiene lugar cuando todos los elementos excepto uno del vector de entrada tienen el mismo número de dígitos. El elemento restante tiene un número significativamente más grande de dígitos que los demás.

En las pruebas realizadas, por razones de almacenamiento de los datos de tipo int en C++, para probar este caso peor se ha utilizado un vector con todos los elementos de 2 dígitos, salvo un elemento, que cuenta con 9 dígitos. Todos estos elementos son generados aleatoriamente, independientemente del número de dígitos. La razón por la que este caso está categorizado como el peor posible al utilizar RadixSort es porque, como ya se ha comentado brevemente en el primer punto de este documento, el coste se ve condicionado al número de cifras del elemento más grande.

La complejidad algorítmica para este algoritmo es del orden de $O(d * n)$, siendo d el número de dígitos del elemento con más dígitos del vector y n el número de elementos del vector.

3.2.2. Caso medio:

El caso medio utilizando el algoritmo RadixSort es aquel en el que el vector de entrada cuenta con números aleatorios de un número de cifras similar entre sí distribuidos de forma relativamente equitativa. En los experimentos realizados, se ha utilizado como entrada un vector de enteros con elementos de 2, 3 y 4 cifras.

Cada tipo de estos elementos en el vector tiene las mismas probabilidades de aparecer (o, dicho de otro modo, aproximadamente un tercio de los elementos son números de 2 dígitos, otro tercio son sumerios de 3 dígitos y el último tercio son números de 4 dígitos).

3.3. BubbleSort:

Debido a las iteraciones necesarias para la ordenación de los elementos de un vector haciendo uso del algoritmo de BubbleSort, tanto el caso peor como los casos medios tienen un coste algorítmico del orden de $O(n^2)$. Tan solo en el caso mejor, en el cual el vector de entrada ya está ordenado, tiene un coste lineal. Por tanto, se ha decidido no hacer distinción en cuanto a casos medios y peores en el análisis de los tiempos al utilizar este algoritmo.

3.4. Realización de los experimentos

3.4.1. Experimentos con elementos aleatorios

Para llevar a cabo cada experimento, en el caso de que en el experimento se hayan utilizado los algoritmos QuickSort o BubbleSort se han ejecutado 20 iteraciones con 1000 elementos, otras 20 iteraciones con 10000 elementos y otras 20 iteraciones con 100000 elementos. En el caso de RadixSort, sin embargo, no fue posible realizar las 20 iteraciones con 100000 elementos (al menos en VSCode), probablemente debido a los recursos de cómputo disponibles en nuestras máquinas locales.

De cada conjunto de 20 iteraciones se ha obtenido la media. Todas las medias obtenidas son los tiempos, en milisegundos, facilitados en la tabla ….

3.4.2. Experimentos con elementos provenientes de fuentes reales

Los experimentos con datos provenientes de fuentes reales se han realizado de la misma manera que con los datos aleatorios salvo que, debido a las estructuras de datos utilizadas, no se han podido llevar a cabo las iteraciones correspondientes a las ordenaciones de vectores con 100000 elementos en nuestras máquinas locales.

3.4.3. Parámetros de los experimentos

Experimento 1-. Algoritmo: QuickSort. Caso: promedio. Pivote: elemento de más a la izquierda. Entrada: vector de enteros aleatorios distribuidos equitativamente

Experimento 2-. Algoritmo: QuickSort. Caso: nueva versión. Pivote: elemento del medio. Entrada: vector de enteros aleatorios distribuidos equitativamente

Experimento 3-. Algoritmo: QuickSort. Caso: nueva versión. Pivote: elemento mediano entre el elemento de más a la izquierda, el de más a la derecha y el del medio. Entrada: vector de enteros aleatorios distribuidos equitativamente

Experimento 4-. Algoritmo: QuickSort. Caso: nueva versión. Pivote: elemento aleatorio. Entrada: vector de enteros aleatorios distribuidos equitativamente

Experimento 5-. Algoritmo: QuickSort. Caso: peor. Pivote: elemento de más a la izquierda. Entrada: vector de enteros aleatorios ya ordenados

Experimento 6-. Algoritmo: QuickSort. Caso: peor. Pivote: elemento de más a la derecha. Entrada: vector de enteros aleatorios ya ordenados decrecientemente

Experimento 7-. Algoritmo: QuickSort. Caso: peor. Pivote: elemento de más a la izquierda. Entrada: vector de enteros con todos los elementos iguales

Experimento 8-. Algoritmo: RadixSort. Caso: peor. Entrada: vector de enteros aleatorios con todos los elementos de 2 dígitos salvo uno, que tiene 9 dígitos

Experimento 9-. Algoritmo: RadixSort. Caso: promedio. Entrada: vector de enteros con todos los elementos de 2, 3 ó 4 dígitos.

Experimento 10-. Algoritmo: BubbleSort. Caso: promedio. Entrada: vector de enteros aleatorios

3.5. Resultados de los experimentos

Los resultados a analizar son aquellos obtenidos tras la ejecución del código el VisualStudio Code. En el servidor remoto Hendrix, a pesar de funcionar correctamente, los tiempos son notablemente mayores, por lo que realizar los experimentos de manera local el VSCode fue mucho más sencillo y rápido.

Algoritmo, versión, caso y entrada del experimento (ver leyenda)	Vectores con enteros aleatorios		Vectores con enteros de fuentes reales	
	Número de elementos	Tiempo promedio (en ms)	Número de elementos	Tiempo promedio (en ms)
1-. Algoritmo QuickSort → average	1000	0.17520	1000	0.10905
	10000	3.22865	10000	2.01785
	100000	203.04000		
2-. Algoritmo QuickSort → newVersion 1	1000	0.09995	1000	0.11995
	10000	1.65565	10000	1.83390
	100000	16.65050		
3-. Algoritmo QuickSort → newVersion 2	1000	0.12345	1000	0.18495
	10000	1.98700	10000	1.98795
	100000	18.11750		
4-. Algoritmo QuickSort → newVersion 3	1000	0.04995	1000	0.19990
	10000	7.33895	10000	2.53640
	100000	526.91000		
5-. Algoritmo QuickSort → worst 1	1000	0.55600	1000	4.29930
	10000	6.97975	10000	506.73500
	100000	237.26000		
6-. Algoritmo QuickSort → worst 2 (*)	X		X	
7-. Algoritmo QuickSort → worst 3	1000	0.10005	1000	0.12915
	10000	0.88285	10000	1.80110
	100000	11.80550		

	Vectores con enteros aleatorios		Vectores con enteros de fuentes reales	
8-. Algoritmo RadixSort → worst	1000	0.25125	1000	0.05000
	10000	2.02675	10000	1.10010
9-. Algoritmo RadixSort → average	1000	0.07650	1000	0.10060
	10000	0.97135	10000	0.45635

	Vectores con enteros aleatorios		Vectores con enteros de fuentes reales	
10-. Algoritmo BubbleSort	1000	0.00000	1000	0.05000
	10000	0.00000		
	100000	424.83930	10000	0.05145

* En el experimento 6 no se han logrado obtener los tiempos correspondientes debido a que el programa colapsaba. A pesar de que con 10 y 100 elementos no había ningún problema (el vector de entrada se pasaba al algoritmo ya ordenado decrecientemente y este lo ordenaba de manera creciente utilizando como pivote el elemento de más a la derecha), al utilizar 1000 y 10000 elementos no se ejecutaba correctamente. Se ha decidido dejarlo constante en la tabla puesto que, al fin y al cabo, se intentó la realización de este experimento.

4. CONCLUSIONES DE LOS RESULTADOS

Observando los resultados recogidos en las tablas anteriores, se concluye que el rendimiento que los tiempos de ejecución en los algoritmos de QuickSort y RadixSort son bajos en comparación con BubbleSort, el cual es significativamente menos eficiente con tiempos de ejecución en conjuntos de datos más grades.

Analizando los resultados del algoritmo de QuickSort, se aprecia que en todas las ejecuciones conforme aumenta el tamaño de datos en el vector, aumenta el tiempo de ejecución. Por otro lado, los resultados de newVersion 1 y newVersion 2 tienen medias de ejecución más rápidas que la primera versión y de manera notoria con la newVersion 3. Con ello se puede concluir que que es importante la elección de la implementación del algoritmo.

La versión implementada del QuickSort en esta práctica tiene limitaciones ya que ha colapsado en el experimento 6 al tratar de ordenar vectores de 1000 y 10000 elementos, lo que indica que puede no ser una opción viable para grandes conjuntos de datos.

5. ASIGNACIÓN DE TAREAS

Para llevar a cabo la realización de la práctica entre los integrantes de la pareja, se ha realizado una asignación de tareas inicialmente donde se define el alcance de la práctica identificando los objetivos y requisitos, repartiéndose de manera equitativa.

Primero se ha llevado a cabo la implementación de los distintos algoritmos y se han realizado pruebas individuales con vectores de 10 elementos enteros. Una vez funcionaba correctamente en el entorno de trabajo de Visual Studio Code, se ha procedido a la implementación completa que se encuentra en los ficheros fuente entregados.

Tras la realización de las pruebas, cuyos datos se encuentran recogidos en esta memoria, se ha procedido a la documentación del código para posteriormente ser puesto en marcha en hendrix y realizar las correcciones necesarias.

Una vez funciona correctamente y se hacen las comprobaciones correspondientes, se realiza la documentación de la memoria y se procede a la preparación de los ficheros fuente entregados.

Se ha trabajado con un ambiente positivo entre los miembros de la pareja de prácticas, sin embargo se han identificado aspectos que requieren mejorar en las siguientes prácticas. Uno de ellos es la gestión del tiempo, que a pesar de realizar las tareas con tiempo suficiente de margen, se considera que una inversión inicial del tiempo en una planificación más detallada podría resultar en mejoras significativas.

6. WEBGRAFÍA

Algoritmo QuickSort:

https://www.baeldung.com/cs/quicksort-time-complexity-worst-case
https://www.geeksforgeeks.org/cpp-program-for-quicksort/
https://www.genbeta.com/desarrollo/implementando-el-algoritmo-quicksort
https://www.geeksforgeeks.org/implement-quicksort-with-first-element-as-pivot/
http://cidecame.uaeh.edu.mx/lcc/mapa/PROYECTO/libro9/mtodo_quick_sort.html#:~:text=Quicksort%20es%20un%20algoritmo%20basado,tiempo%20proporcional%20a%20n%20log%20n.

Algoritmo RadixSort:

https://www.geeksforgeeks.org/radix-sort/
https://sites.google.com/a/uabc.edu.mx/radix-sort/algoritmo#:~:text=El%20ordenamiento%20radix%20se%20empieza,hacerse%20con%20todos%20los%20d%C3%ADgitos.
https://sites.google.com/a/uabc.edu.mx/radix-sort/algoritmo

Algoritmo BubbleSort:

https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm#:~:text=The%20bubble%20sort%20algorithm%20is,complexity%20of%20O(1).