

**Tabla de actividades propuesta**

|  |  |
| --- | --- |
| Actividad | Responsable |
| Codificar bubble sort | Daniel |
| Codificar insertion sort | Ernesto |
| Codificar quick sort | Daniel |
| Codificar merge sort | Ernesto |
| Codificar heap sort | Daniel |
| Codificar generación de archivos | Ernesto |
| Codificar lectura de archivos | Daniel |
| Realizar pruebas | Daniel |
| Documentar pruebas experimentales | Daniel |
| Documentar los resultados teoricos | Ernesto |
| Elaboración del documento | Ernesto |
| Revisión del documento | Daniel |

**Justificación**

En el presente trabajo se mostrará la comparación entre varios algoritmos de ordenamiento, tanto teórica como experimentalmente.

Se utilizarán archivos con datos en formato número entero, los tamaños de los archivos son los siguientes:

* Archivo pequeño: 10 enteros.
* Archivo mediano: 100 enteros.
* Archivo grande: 1,000 enteros.
* Archivo extra grande: 100,000 enteros.

Para mayor comodidad y exactitud de las muestras, se han generado previamente a la entrega de este reporte, con lo cual, la veracidad del tiempo de ejecución tendrá un margen de error reducido, ya que se ejecutarán y ordenarán los mismos números. Se añadirán los archivos correspondientes a la entrega.

Las pruebas se realizaron en una máquina virtual con SO tipo Linux (Kali), con un Ryzen 3 y 8 GB de RAM.

**Bubble sort**

|  |
| --- |
| //=============================== BUBBLE SORT ==================================  *void* bubbleSort(*int* arr[], *int* n){  // Iterando sobre al arreglo  for (*int* i = 0; i < n - 1; i++){  // Bandera para revisar si el arreglo ya esta ordenado  *bool* estaOrdenado = true;  // Se comparan los elementos adyacentes  // Si el primero es mas grande  // Se intercambia y se pone la bandera en falso  for (*int* j = 0; j < n - i - 1; j++){  if (arr[j] > arr[j + 1]){  *int* temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  estaOrdenado = false;  }  }  // Si el arreglo esta ordenado romper el for  if (estaOrdenado)  break;  }  } |

Caso promedio de tiempo de ejecución teórico.

**Insertion sort**

|  |
| --- |
| //============================== INSERTION SORT ================================  void insertionSort(int arr[], int n){  // iterando sobre el arreglo  for (int i = 1; i < n; i++){  // obtener el elemento actual  int actual = arr[i];  // compara el elemento actual con los elementos previos  // y moverlos a la derecha hasta encontrar su posicion correcta  // para el elemento actual  int j = i - 1;  while (j >= 0 && arr[j] > actual)  {  arr[j + 1] = arr[j];  j--;  }  // insertar el elemento actual en la posicion correcta  arr[j + 1] = actual;  }  } |

Caso promedio de tiempo de ejecución teórico.

**Quick sort**

|  |
| --- |
| //================================ QUICK SORT ==================================  *int* particion(*int* arr[], *int* inicio, *int* fin){  *int* pivote = arr[inicio];  *int* contador = 0;  for (*int* i = inicio + 1; i <= fin; i++) {  if (arr[i] <= pivote)  contador++;  }  // Dando al pivote su posicion correcta  *int* indicePivote = inicio + contador;  swap(arr[indicePivote], arr[inicio]);  // ordenando las partes izq y der del pivote  *int* i = inicio, j = fin;  while (i < indicePivote && j > indicePivote) {  while (arr[i] <= pivote) {  i++;  }  while (arr[j] > pivote) {  j--;  }  if (i < indicePivote && j > indicePivote) {  swap(arr[i++], arr[j--]);  }  }  return indicePivote;  }  *void* quickSort(*int* arr[], *int* inicio, *int* fin){  // caso base  if (inicio >= fin)  return;  // particionando el arreglo  *int* p = particion(arr, inicio, fin);  // ordenando la parte izq  quickSort(arr, inicio, p - 1);  // ordenando la parte der  quickSort(arr, p + 1, fin);  } |

Caso promedio de tiempo de ejecución teórico.

**Merge sort**

|  |
| --- |
| //============================== MERGE SORT ====================================  void merge(int arr[], int const izq, int const mid,  int const der)  {  auto const subArregloUno = mid - izq + 1;  auto const subArregloDos = der - mid;  // Creamos arreglos temporales  auto \*arregloIzq = new int[subArregloUno],  \*arregloDer = new int[subArregloDos];  // Copiamos datos a los arreglos temportales arregloIzq[] y arregloDer[]  for (auto i = 0; i < subArregloUno; i++)  arregloIzq[i] = arr[izq + i];  for (auto j = 0; j < subArregloDos; j++)  arregloDer[j] = arr[mid + 1 + j];  auto indiceSubArregloUno  = 0, // Indice inicial del primer sub arreglo  indiceSubArregloDos  = 0; // Indice inicial del segundo sub arreglo  int indiceArregloMezclado  = izq; // Indice inicial del arreglo mezclado  // Mezclar los arreglos temporales al arreglo mezclado[izq..der]  while (indiceSubArregloUno < subArregloUno  && indiceSubArregloDos < subArregloDos){  if (arregloIzq[indiceSubArregloUno]  <= arregloDer[indiceSubArregloDos]){  arr[indiceArregloMezclado]  = arregloIzq[indiceSubArregloUno];  indiceSubArregloUno++;  }  else{  arr[indiceArregloMezclado]  = arregloDer[indiceSubArregloDos];  indiceSubArregloDos++;  }  indiceArregloMezclado++;  }  // copiar los elementos restantes de  // izq[], si es que hay  while (indiceSubArregloUno < subArregloUno){  arr[indiceArregloMezclado]  = arregloIzq[indiceSubArregloUno];  indiceSubArregloUno++;  indiceArregloMezclado++;  }  // copiar los elementos restantes de  // der[], si es que hay  while (indiceSubArregloDos < subArregloDos) {  arr[indiceArregloMezclado]  = arregloDer[indiceSubArregloDos];  indiceSubArregloDos++;  indiceArregloMezclado++;  }  delete[] arregloIzq;  delete[] arregloDer;  } |

Caso promedio de tiempo de ejecución teórico.

**Heap sort**

|  |
| --- |
| //=============================== HEAP SORT ====================================  *void* heapea(*int* arr[], *int* n, *int* i){  // encuentra el elemento mas grande entre la raiz, hijo izq e hijo der  *int* masGrande = i;  *int* izq = 2 \* i + 1;  *int* der = 2 \* i + 2;  if (izq < n && arr[izq] > arr[masGrande]){  masGrande = izq;  }  if (der < n && arr[der] > arr[masGrande]){  masGrande = der;  }  // si el elemento mas grande no es la raiz, cambiar los elementos y heapear el subarbol  if (masGrande != i)  {  *int* temp = arr[i];  arr[i] = arr[masGrande];  arr[masGrande] = temp;  heapea(arr, n, masGrande);  }  }  *void* heapSort(*int* arr[], *int* n)  {  // construye el monticulo  for (*int* i = n / 2 - 1; i >= 0; i--){  heapea(arr, n, i);  }  // Ordenando el arreglo  for (*int* i = n - 1; i >= 0; i--){  // mover la raiz actual al final  *int* temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  // heapear los elementos restantes  heapea(arr, i, 0);  }  } |

Caso promedio de tiempo de ejecución teórico.

**Generación de archivos**

|  |
| --- |
| // Genera un entero en el rango [1, n]  *int* generaEnteroRandom(*int* n) {  return rand() % n + 1;  }  // Crea un archivo con el numero de enteros especificado  *void* crearArchivo(string nombreArchivo, *int* numeroDeEnteros) {  // Semilla para numero random  srand(time(NULL));  ofstream out(nombreArchivo);  // Escribe el numero de enteros en el archivo  out << numeroDeEnteros << endl;  // Escribe los enteros random en el archivo  for (*int* i = 0; i < numeroDeEnteros; i++) {  out << generaEnteroRandom(numeroDeEnteros) << endl;  }  out.close();  } |

**Lectura de archivos**

|  |
| --- |
| // Extrae datos del archivo y los guarda en un array  *void* extraeDatos(string nombreArchivo, *int*\* datos, *int* numeroDeEnteros) {  ifstream in(nombreArchivo);  // Read the integers from the file and store them in the array  for (*int* i = 0; i < numeroDeEnteros; i++) {  in >> datos[i];  }  in.close();  } |

**Código completo**

|  |
| --- |
| //Algoritmos de ordenamiento  #include <bits/stdc++.h>  #include <iostream>  #include <fstream>  #include <cstdlib>  #include <ctime>  using namespace std;  const *int* TAMS[] = {10, 100, 1000, 100000};  // Imprime el arreglo  *void* imprimeArreglo(*int* arr[], *int* n)  {  for (*int* i = 0; i < n; i++)  cout << arr[i] << " ";  cout << endl;  }  //============================== PARA ARCHIVOS =================================  // Genera un entero en el rango [1, n]  *int* generaEnteroRandom(*int* n) {  return rand() % n + 1;  }  // Crea un archivo con el numero de enteros especificado  *void* crearArchivo(string nombreArchivo, *int* numeroDeEnteros) {  // Semilla para numero random  srand(time(NULL));  ofstream out(nombreArchivo);  // Escribe el numero de enteros en el archivo  out << numeroDeEnteros << endl;  // Escribe los enteros random en el archivo  for (*int* i = 0; i < numeroDeEnteros; i++) {  out << generaEnteroRandom(numeroDeEnteros) << endl;  }  out.close();  }  // Extrae datos del archivo y los guarda en un array  *void* extraeDatos(string nombreArchivo, *int*\* datos, *int* numeroDeEnteros) {  ifstream in(nombreArchivo);  // Read the integers from the file and store them in the array  for (*int* i = 0; i < numeroDeEnteros; i++) {  in >> datos[i];  }  in.close();  }  //=============================== HEAP SORT ====================================  *void* heapea(*int* arr[], *int* n, *int* i){  // encuentra el elemento mas grande entre la raiz, hijo izq e hijo der  *int* masGrande = i;  *int* izq = 2 \* i + 1;  *int* der = 2 \* i + 2;  if (izq < n && arr[izq] > arr[masGrande]){  masGrande = izq;  }  if (der < n && arr[der] > arr[masGrande]){  masGrande = der;  }  // si el elemento mas grande no es la raiz, cambiar los elementos y heapear el subarbol  if (masGrande != i)  {  *int* temp = arr[i];  arr[i] = arr[masGrande];  arr[masGrande] = temp;  heapea(arr, n, masGrande);  }  }  *void* heapSort(*int* arr[], *int* n)  {  // construye el monticulo  for (*int* i = n / 2 - 1; i >= 0; i--){  heapea(arr, n, i);  }  // Ordenando el arreglo  for (*int* i = n - 1; i >= 0; i--){  // mover la raiz actual al final  *int* temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  // heapear los elementos restantes  heapea(arr, i, 0);  }  }  //=============================== BUBBLE SORT ==================================  *void* bubbleSort(*int* arr[], *int* n){  // Iterando sobre al arreglo  for (*int* i = 0; i < n - 1; i++){  // Bandera para revisar si el arreglo ya esta ordenado  *bool* estaOrdenado = true;  // Se comparan los elementos adyacentes  // Si el primero es mas grande  // Se intercambia y se pone la bandera en falso  for (*int* j = 0; j < n - i - 1; j++){  if (arr[j] > arr[j + 1]){  *int* temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  estaOrdenado = false;  }  }  // Si el arreglo esta ordenado romper el for  if (estaOrdenado)  break;  }  }  //============================== MERGE SORT ====================================  *void* merge(*int* arr[], *int* const izq, *int* const mid,  *int* const der)  {  *auto* const subArregloUno = mid - izq + 1;  *auto* const subArregloDos = der - mid;  // Creamos arreglos temporales  *auto* \*arregloIzq = new *int*[subArregloUno],  \*arregloDer = new *int*[subArregloDos];  // Copiamos datos a los arreglos temportales arregloIzq[] y arregloDer[]  for (*auto* i = 0; i < subArregloUno; i++)  arregloIzq[i] = arr[izq + i];  for (*auto* j = 0; j < subArregloDos; j++)  arregloDer[j] = arr[mid + 1 + j];  *auto* indiceSubArregloUno  = 0, // Indice inicial del primer sub arreglo  indiceSubArregloDos  = 0; // Indice inicial del segundo sub arreglo  *int* indiceArregloMezclado  = izq; // Indice inicial del arreglo mezclado  // Mezclar los arreglos temporales al arreglo mezclado[izq..der]  while (indiceSubArregloUno < subArregloUno  && indiceSubArregloDos < subArregloDos){  if (arregloIzq[indiceSubArregloUno]  <= arregloDer[indiceSubArregloDos]){  arr[indiceArregloMezclado]  = arregloIzq[indiceSubArregloUno];  indiceSubArregloUno++;  }  else{  arr[indiceArregloMezclado]  = arregloDer[indiceSubArregloDos];  indiceSubArregloDos++;  }  indiceArregloMezclado++;  }  // copiar los elementos restantes de  // izq[], si es que hay  while (indiceSubArregloUno < subArregloUno){  arr[indiceArregloMezclado]  = arregloIzq[indiceSubArregloUno];  indiceSubArregloUno++;  indiceArregloMezclado++;  }  // copiar los elementos restantes de  // der[], si es que hay  while (indiceSubArregloDos < subArregloDos) {  arr[indiceArregloMezclado]  = arregloDer[indiceSubArregloDos];  indiceSubArregloDos++;  indiceArregloMezclado++;  }  delete[] arregloIzq;  delete[] arregloDer;  }  // inicio es para el indice izq y fin es  // para el indice der del sub arreglo  // del arreglo a ordenar  *void* mergeSort(*int* arr[], *int* const inicio, *int* const fin)  {  if (inicio >= fin)  return; // caso base  *auto* mid = inicio + (fin - inicio) / 2;  mergeSort(arr, inicio, mid);  mergeSort(arr, mid + 1, fin);  merge(arr, inicio, mid, fin);  }  //================================ QUICK SORT ==================================  *int* particion(*int* arr[], *int* inicio, *int* fin){  *int* pivote = arr[inicio];  *int* contador = 0;  for (*int* i = inicio + 1; i <= fin; i++) {  if (arr[i] <= pivote)  contador++;  }  // Dando al pivote su posicion correcta  *int* indicePivote = inicio + contador;  swap(arr[indicePivote], arr[inicio]);  // ordenando las partes izq y der del pivote  *int* i = inicio, j = fin;  while (i < indicePivote && j > indicePivote) {  while (arr[i] <= pivote) {  i++;  }  while (arr[j] > pivote) {  j--;  }  if (i < indicePivote && j > indicePivote) {  swap(arr[i++], arr[j--]);  }  }  return indicePivote;  }  *void* quickSort(*int* arr[], *int* inicio, *int* fin){  // caso base  if (inicio >= fin)  return;  // particionando el arreglo  *int* p = particion(arr, inicio, fin);  // ordenando la parte izq  quickSort(arr, inicio, p - 1);  // ordenando la parte der  quickSort(arr, p + 1, fin);  }  //============================== INSERTION SORT ================================  *void* insertionSort(*int* arr[], *int* n){  // iterando sobre el arreglo  for (*int* i = 1; i < n; i++){  // obtener el elemento actual  *int* actual = arr[i];  // compara el elemento actual con los elementos previos  // y moverlos a la derecha hasta encontrar su posicion correcta  // para el elemento actual  *int* j = i - 1;  while (j >= 0 && arr[j] > actual)  {  arr[j + 1] = arr[j];  j--;  }  // insertar el elemento actual en la posicion correcta  arr[j + 1] = actual;  }  }  *int* main() {  // Vars donde se guardaran los datos  *int*\* datosPeque = new *int*[TAMS[0]];  *int*\* datosMed = new *int*[TAMS[1]];  *int*\* datosGra = new *int*[TAMS[2]];  *int*\* datosEG = new *int*[TAMS[3]];  // Nombre de los archivos  string nombreArchivoPeque = "peque.txt";  string nombreArchivoMed = "med.txt";  string nombreArchivoGra = "grande.txt";  string nombreArchivoEG = "eg.txt";  /\*crearArchivo(nombreArchivoPeque, TAMS[0]);  crearArchivo(nombreArchivoMed, TAMS[1]);  crearArchivo(nombreArchivoGra, TAMS[2]);  crearArchivo(nombreArchivoEG, TAMS[3]);\*/    // Extraccion de los datos de los archivos  extraeDatos(nombreArchivoPeque, datosPeque, TAMS[0]);  extraeDatos(nombreArchivoMed, datosMed, TAMS[1]);  extraeDatos(nombreArchivoGra, datosGra, TAMS[2]);  extraeDatos(nombreArchivoEG, datosEG, TAMS[3]);  //Se toma el tiempo  *clock\_t* inicio, fin;    /\* Tomamos el inicio del reloj para tomar la medida del tiempo de  ejecucion de los algoritmos de ordenamiento.\*/  inicio = clock();    //EJECUCIONES  //========================= ARCHIVO PEQUE ==============================  //quickSort(datosPeque, 0, TAMS[0] - 1);  /\*mergeSort(datosPeque, 0, TAMS[0] - 1);  bubbleSort(datosPeque, TAMS[0]);  insertionSort(datosPeque, TAMS[0]);  heapSort(datosPeque, TAMS[0]);\*/  heapSort(datosPeque, TAMS[0]);  imprimeArreglo(datosPeque, TAMS[0]);  //========================= ARCHIVO MED ================================  //quickSort(datosMed, 0, TAMS[1] - 1);  /\*mergeSort(datosMed, 0, TAMS[1] - 1);  bubbleSort(datosMed, TAMS[1]);  insertionSort(datosMed, TAMS[1]);  heapSort(datosMed, TAMS[1]);  heapSort(datosMed, TAMS[1]);  imprimeArreglo(datosMed, TAMS[1]);\*/  //========================= ARCHIVO GRA ================================  //quickSort(datosGra, 0, TAMS[2] - 1);  /\*mergeSort(datosGra, 0, TAMS[2] - 1);  bubbleSort(datosGra, TAMS[2]);  insertionSort(datosGra, TAMS[2]);  heapSort(datosGra, TAMS[2]);  heapSort(datosGra, TAMS[2]);  imprimeArreglo(datosGra, TAMS[2]);\*/  //========================= ARCHIVO EG =================================  //quickSort(datosEG, 0, TAMS[3] - 1);  /\*mergeSort(datosEG, 0, TAMS[3] - 1);  bubbleSort(datosEG, TAMS[3]);  insertionSort(datosEG, TAMS[3]);  heapSort(datosEG, TAMS[3]);  heapSort(datosEG, TAMS[3]);  imprimeArreglo(datosEG, TAMS[3]);\*/  // Obtenemos el final del reloj  fin = clock();    // Calcula el tiempo que tomo hacer la operacion encerrada entre inicio y fin  *double* tiempo\_tomado = *double*(fin - inicio) / *double*(CLOCKS\_PER\_SEC);  cout << "Tiempo de ejecucion : " << fixed  << tiempo\_tomado << setprecision(5);  cout << " seg " << endl;    return 0;  } |

**Suposiciones teóricas de ejecución**

Las suposiciones que se tienen con la información obtenida durante el curso en la UEA “Análisis y diseño de algoritmos”, concluimos lo siguiente:

* Los algoritmos *bubble sort* e *insertion sort*, serán los algoritmos que tomen más tiempo al ejecutarse ordenando el archivo extra grande, ya que tienen un tiempo de ejecución teórico de , por otra parte, al ordenar los archivos pequeño y mediano no se tendrá un margen dramático en comparación a los demás algoritmos.
* Los algoritmos *quick sort*, *merge sort*, y *heap sort,* serán algoritmos que tendrán una diferencia mínima al ordenar el arreglo con los datos del archivo extra grande, debido a su tiempo de ejecución teórico de , al ordenar los datos pequeño y mediano no se tendrá una ejecución más rápida, simplemente terminarán más rápido que los algoritmos con un tiempo de ejecución mayor que estos.

**Resultados experimentales**

. Ejecución con el archivo pequeño.

|  |  |
| --- | --- |
| Ejecución con el archivo pequeño | |
| Algoritmo | **Tiempo de ejecución (seg)** |
| Bubble sort | 0.000330 |
| Insertion sort | 0.000278 |
| Quick sort | 0.000266 |
| Merge sort | 0.000104 |
| Heap sort | 0.000439 |



Ilustración . Ejecución con bubble sort.



Ilustración . Ejecución con insertion sort.



Ilustración . Ejecución con quick sort.



Ilustración . Ejecución con merge sort.



Ilustración . Ejecución con heap sort.

. Ejecución con el archivo mediano.

|  |  |
| --- | --- |
| Ejecución con el archivo mediano | |
| Algoritmo | **Tiempo de ejecución (seg)** |
| Bubble sort | 0.000117 |
| Insertion sort | 0.000476 |
| Quick sort | 0.000107 |
| Merge sort | 0.000287 |
| Heap sort | 0.000112 |

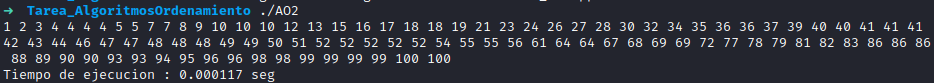


Ilustración . Ejecución con bubble sort.

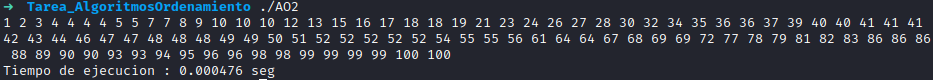


Ilustración . Ejecución con insertion sort.

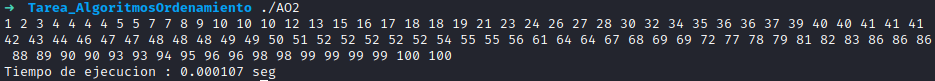


Ilustración . Ejecución con quick sort.

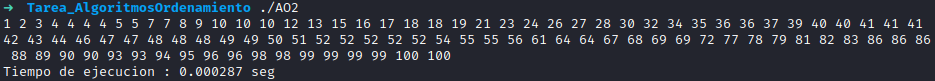


Ilustración . Ejecución con merge sort.

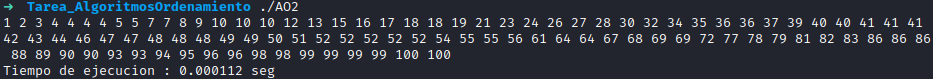


Ilustración . Ejecución con heap sort.

. Ejecución con el archivo grande.

|  |  |
| --- | --- |
| Ejecución con el archivo grande | |
| Algoritmo | **Tiempo de ejecución (seg)** |
| Bubble sort | 0.002833 |
| Insertion sort | 0.001185 |
| Quick sort | 0.000344 |
| Merge sort | 0.000298 |
| Heap sort | 0.000716 |

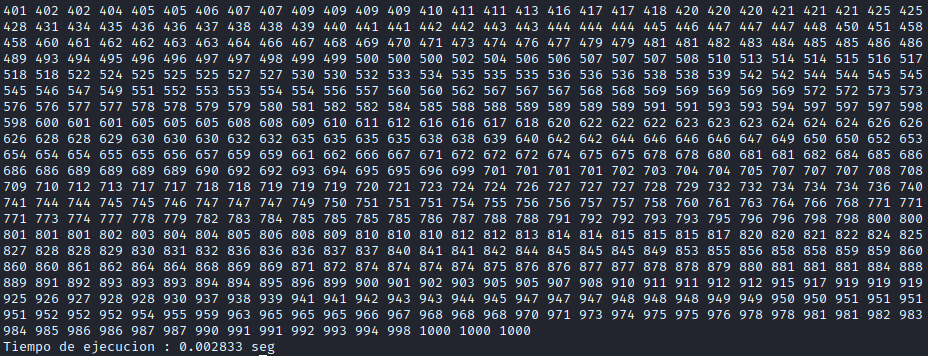


Ilustración . Ejecución con bubble sort.

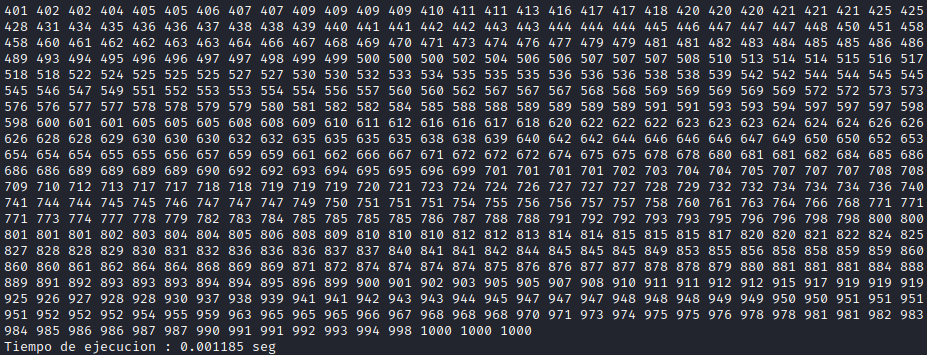


Ilustración . Ejecución con insertion sort.

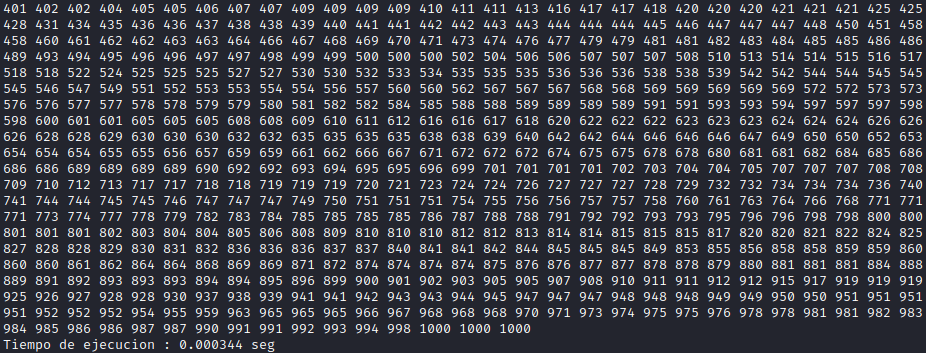


Ilustración . Ejecución con quick sort.

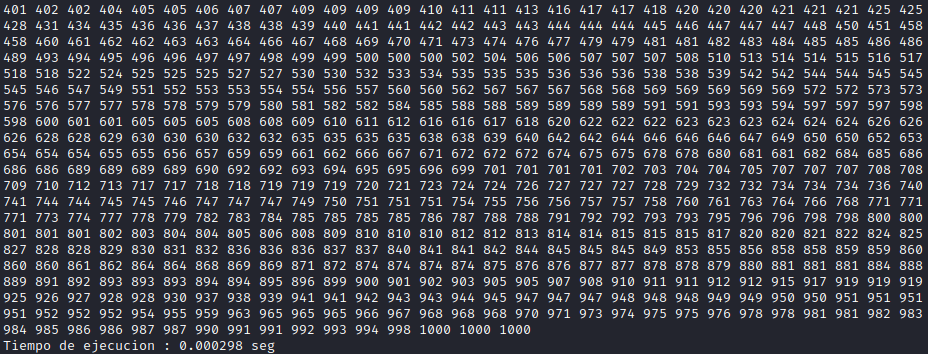


Ilustración . Ejecución con merge sort.

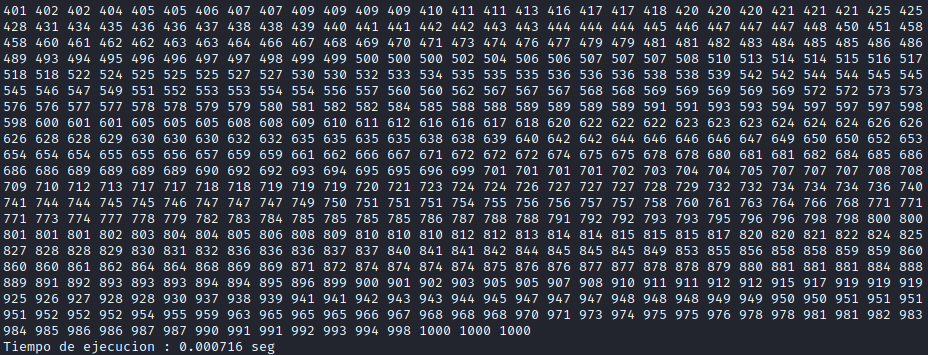


Ilustración . Ejecución con heap sort.

. Ejecución con el archivo extra grande.

|  |  |
| --- | --- |
| Ejecución con el archivo extra grande | |
| Algoritmo | **Tiempo de ejecución (seg)** |
| Bubble sort | 34.501171 |
| Insertion sort | 10.320607 |
| Quick sort | 0.043013 |
| Merge sort | 0.039318 |
| Heap sort | 0.044555 |

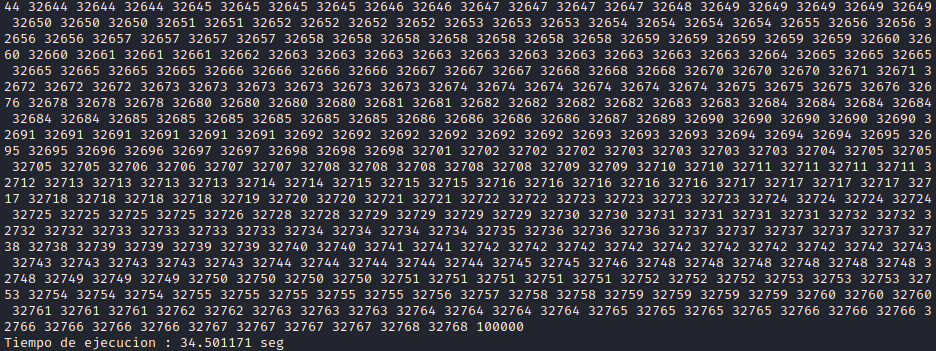


Ilustración . Ejecución con bubble sort.

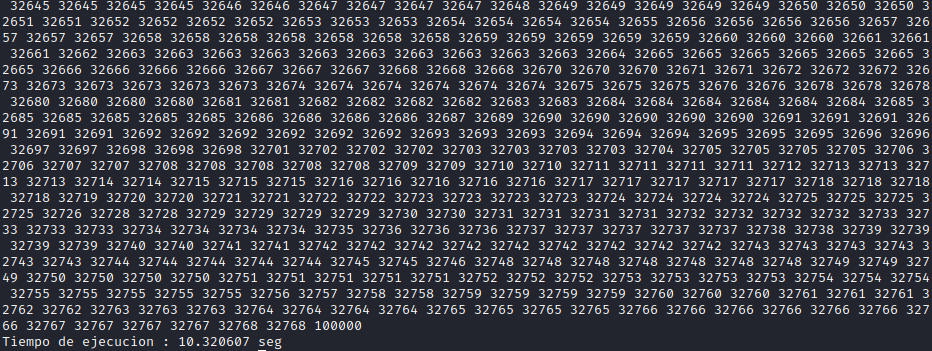


Ilustración . Ejecución con insertion sort.

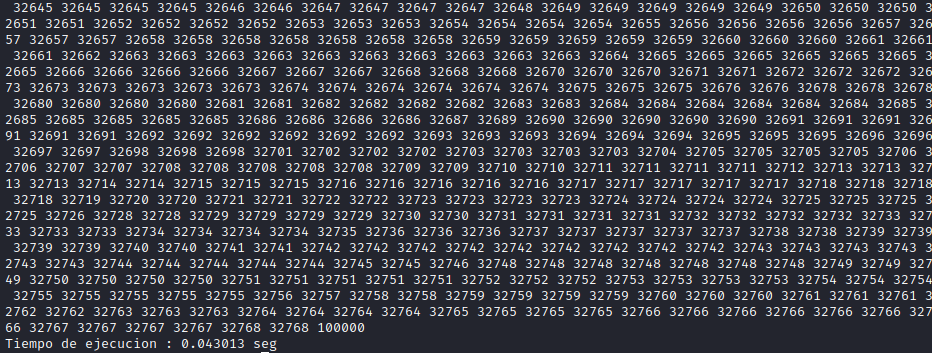


Ilustración . Ejecución con quick sort.

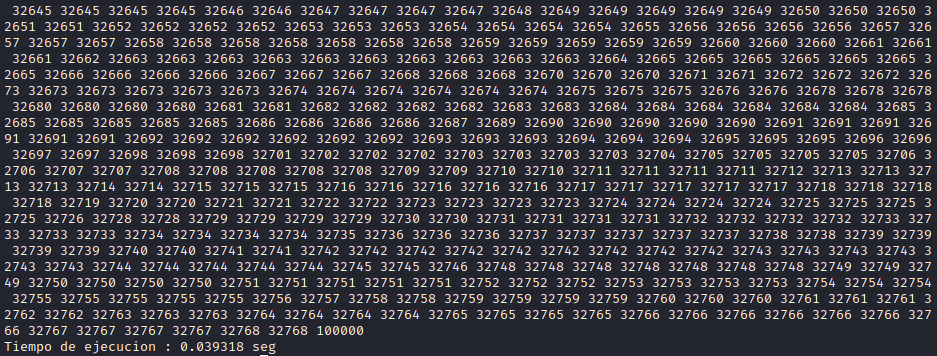


Ilustración . Ejecución con merge sort.

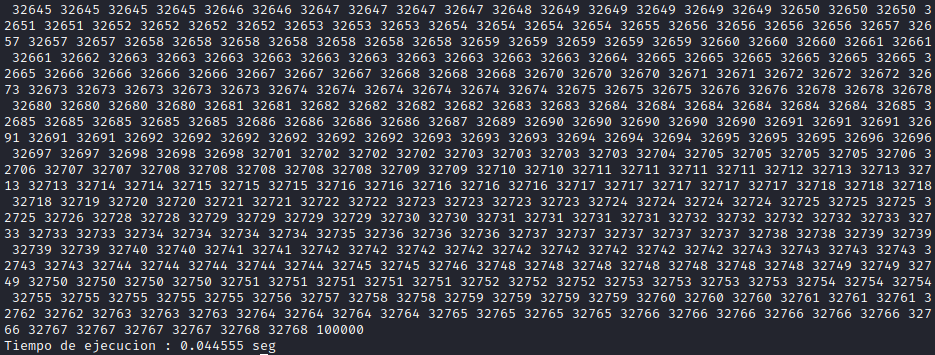


Ilustración . Ejecución con heap sort.

**Comparación de ejecución**

Como podemos notar, el algoritmo *merge sort* es el que mejor se desempeñó al ordenar los 10 números. Tanto los algoritmos *insertion sort* y *quick sort* están muy parejos, les siguen los algoritmos *bubble sort* y *heap sort*, este último es el que más se ha tardado en ordenar el archivo pequeño.

El algoritmo *quick sort* es el que mejor se ha desempeñado al ordenar los 100 números. Le siguen los algoritmos *bubble sort* y *heap sort*, nos parece interesante que el algoritmo *bubble sort* se ha desempeñado mejor que el algoritmo *merge sort*, ya que este y el algoritmo de *insertion sort* son los que peor se han desempeñado.

El algoritmo *merge sort* es el que mejor se desempeñó al ordenar el archivo con los 1000 números, le siguen los algoritmos *quick sort* y *heap sort*, por último le siguen los algoritmos *insertion sort* y *bubble sort*, ahora comenzamos a ver el comportamiento teórico del algoritmo *bubble sort*, puesto que ahora se nota una diferencia notable en comparación con los otros algoritmos de ordenamiento.

El algoritmo *merge sort* es el que mejor se desempeñó al ordenar el archivo con los 100,000 números, le siguen *quick sort* y *heap sort*, por último le siguen los algoritmos *insertion sort* y *bubble sort*, notamos que estos últimos han tomado un tiempo considerablemente mayor, siendo el algoritmo *bubble sort* el que peor se desempeñó, con una diferencia del 199.545% en comparación con el algoritmo *merge sort* (el de mejor desempeño).

**Tabla de actividades resultante**

|  |  |  |
| --- | --- | --- |
| Actividad | Responsable | Tiempo tomado |
| Codificar bubble sort | Daniel | 5 min |
| Codificar insertion sort | Ernesto | 3 min |
| Codificar quick sort | Daniel | 20 min |
| Codificar merge sort | Ernesto | 15 min |
| Codificar heap sort | Daniel | 25 min |
| Codificar generación de archivos | Ernesto | 2 min |
| Codificar lectura de archivos | Daniel | 5 min |
| Realizar pruebas | Daniel | 10 min |
| Documentar pruebas experimentales | Daniel | 30 min |
| Documentar los resultados teoricos | Ernesto | 5 min |
| Elaboración del documento | Ernesto | 60 min |
| Revisión del documento | Daniel | 30 min |

**Conclusiones**

* Flores Machuca Ernesto: para concluir cabe mencionar que se ha analizado detalladamente el funcionamiento y la eficiencia de cada método de ordenamiento, y se ha observado que algunos metodos funcionan mejor, o son mas convenientes utilizarlos cuando el arreglo de elementos están semiordenados, como es el caso de insertion sort y bubble sort, para los metodos como quick sort y merge sort es mejor utilizarlos cuando nuestro arreglo se encuentre totalmente desordenado, ya que son algoritmos que utilizan la técnica de divide y vencerás y de esta forma llegan a ser mas eficientes, finalmente el ultimo método de ordenamiento heap sort es muy interesante el funcionamiento ya que este utiliza una estructura de árbol, y como se sabe, es eficiente encontrar los elementos e ir poniendo el mas grande hasta la raíz, para que de esta forma al estar ordenandolo se vaya agregando a la última posición del arreglo y cada vez son menos elementos en el árbol y de esta forma podemos ir ordenando eficientemente todo nuestro arreglo
* Gutiérrez Ramos Alberto Daniel: Con esta tarea teórico-práctica, he comprobado la eficiencia en tiempo de ejecución de estos algoritmos de ordenamiento. Con lo que he llegado a la conclusión de que los algoritmos *bubble sort* e *insertion sort* se desempeñan igual o mejor al ordenar cantidades considerablemente “pequeñas”, mientras que los algoritmos *quick sort, merge sort* y *heap sort* se desempeñan mejor con una cantidad mayor de datos, puesto que su implementación se beneficia de esto, con cantidades pequeñas de datos, es indistinto usar *bubble sort* o *quick sort*. También está el asunto de la eficiencia de espacio en el tiempo de ejecución, ya que *merge sort* se desempeñó mejor que los demás algoritmos en este ejercicio experimental, ya que dicho algoritmo ocupa arreglos auxiliares para mezclar datos. En cuanto al trabajo colaborativo, prefiero trabajar individual, pero fue un ejercicio interesante poder colaborar con un compañero y resolver la problemática.