CENTRO DE CIENCIAS BÁSICAS

DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURAS COMPUTACIONALES

2° "A"

ANÁLISIS DE EFICIENCIA DE LOS MÉTODOS DE ORDENAMIENTO INTERNO

Profesor: Francisco Javier Luna Rosas

Alumnos:

Espinoza Sánchez Joel Alejandro

Gómez Garza Dariana

Macías Soto Valeria

Fecha de Entrega: Aguascalientes, Ags.**, 11** de junio de 2019

Índice

Capítulo 1: Introducción

La programación surge como un método para optimizar las tareas que vienen a partir de la mecanización y la repetición de procesos. De ahí se concibe el término de “algoritmo” el cual, de manera muy informal se define como un procedimiento a partir de una entrada para obtener un resultado.

Muchos algoritmos se han creado para la resolución de distintos problemas, algunos problemas muy sencillos, otros más complicados, pero todos definidos a partir de la complejidad computacional en función del tiempo que le toma a cierto algoritmo realizar todo el proceso y obtener la conclusión que se espera. Esto se puede observar con los algoritmos hechos para ordenar colecciones de datos, los muy conocidos “métodos de ordenamiento”.

A partir de los algoritmos de ordenamiento, se puede observar la complejidad computacional de cada uno, es decir, el tiempo que le toma a cada uno realizar el ordenamiento de un número determinado de datos y así, obtener cuál método de ordenamiento es el más eficiente en cuestión del tiempo.

En la presente investigación, se realizará un análisis de eficiencia de los métodos de ordenamiento Bubble, Insertion, Selection, Shell, Quick, Merge, Radix y Heap para encontrar cuál es el más eficiente según una cantidad determinada de datos, para después, procesar la información y encontrar el algoritmo más veloz de entre los ocho ya mencionados. Para ello, se ha investigado y programado cada método de ordenamiento en el lenguaje de programación C.

Asimismo, se estudiaron los tiempos de cada método con la biblioteca de funciones “`<time.h>`” con diferentes casos, diferentes procesos y elementos a ordenar. Finalmente, con tests estadísticos se probarán y demostrarán los órdenes de complejidad de cada algoritmo para determinar cuál es el algoritmo más eficiente de los ya presentados.

Capítulo 2: Generalidades

Métodos de Ordenamiento

La ordenación de datos es uno de los procesos más importantes en cuanto a programación se refiere, este método utiliza diferentes tipos de algoritmos para facilitar técnicas de ejecución como la posibilidad de una búsqueda jerarquizada u otros complejos estructurales más avanzados que en la programación existen, tales como el uso de árboles binarios en los grafos al momento de programar un algoritmo de este tipo.

Los algoritmos son herramientas que se han usado para facilitar muchos trabajos debido a la mecanización y repetición de estos. Esta serie de instrucciones sigue una secuencia lógica que finalmente arroja un resultado esperado.

Sin embargo, un algoritmo debe respetar las cualidades de ser simple, claro y adecuado para el manejo de datos. Aho (1983) definía a un algoritmo como "una meta más objetiva, lo cual no significa que sea más importante, es la eficiencia en tiempo de ejecución".

Algoritmos

La palabra algoritmo fue creada por un matemático persa del siglo IX llamado *Al-Khwarizmi* que produjo el primer libro conocido de álgebra: "*Al-Kitab alMukhtasar fi Hisab al-Jabr wa l-Muqabala*" (Compendio de cálculo por reintegración y comparación), el cual pasó a reinventarse y usarse en la actualidad para formar el nuevo compendio algebraico "Álgebra" hecho por el cubano Aurelio Baldor (como referencia al uso del previo, *Al-Khwarizmi* aparece en la portada de la edición original).

El nombre álgebra viene directo del nombre al-Jabr del título en el libro. Como los escolares diseminaron el trabajo de Al-Khwarizmi en latín durante el medioevo, la traducción de su nombre, "algorism", fue utilizada para describir cualquier método de cálculo sistemático o automático, pues en su obra, describía el proceso sistemático y mecanizado de diferentes procedimientos del álgebra, tales como el algoritmo de la división, el algoritmo de la multiplicación, algoritmos de fracciones, productos notables, entre otros.

Ejemplos de algoritmos pueden hallarse desde el primer algoritmo grabado y luego encontrado por la civilización moderna, que viene de Shuruppak. Los sumerios, dejaron tablas de arcilla que datan aproximadamente del 2500 A.C. y que ilustraban un método repetitivo para medir equitativamente la cosecha de granos entre un número variable de hombres. El método descrito utilizaba herramientas de precisión para medir.

Algunos algoritmos que fueron desarrollados muchos años atrás siguen teniendo un impacto en nuestra actualidad, en el mundo tecnológico de hoy: muchos sitios web,

routers inalámbricos, y lugares donde nombres de usuarios y contraseñas deben ser encriptados utilizando un algoritmo que fue concebido hace dos mil años por Euclides de Alejandría, un matemático griego.

El algoritmo de Euclides es usado por la industria para derivar los patrones rítmicos de la música. Además, Euclides, en su trabajo llamado Elementos, incluyó un algoritmo que encontraba el divisor más largo entre dos números diferentes.

La secuencia de Fibonacci es otro algoritmo importante que es reconocido actualmente, es una secuencia que establece que cada número es la suma de los dos que lo preceden: **1, 1, 2, 3, 5, 8, 13, 21, 34,** etc. A medida que la secuencia progresa, la proporción de los números y su inmediato predecesor convergen hacia la proporción áurea de 1,618, también conocida como la medida de oro.

Gottfried Leibniz fue un matemático que concebía el lenguaje del cálculo definido solamente por dos figuras: 0 y 1. Leibniz desarrolló este sistema para expresar todos los números y operaciones de la aritmética (suma, resta, multiplicación y división) en el lenguaje binario de 1's y 0's. Él definió este lenguaje en su documento de 1703, "Explicación de la aritmética binaria".

Leibniz y Newton descubrieron el cálculo en paralelo. Leibniz fue quien desarrolló las notas para las funciones integrales y derivadas que todos los estudiantes aprenden hoy en día. El cálculo y los algoritmos tienen historias estrechamente entrelazadas.

Leibniz avanzó en la ciencia algorítmica de tres maneras diferentes. Fue uno de los instigadores fundadores del cálculo y, introdujo el método de construcción de algoritmos para expresar difíciles soluciones en una serie de sencillos bloques binarios y su tercera contribución recae en el enlace buscado entre los fragmentos simples del lenguaje que revelan las emociones humanas. Descubrió que si algo tan complicado como la existencia humana podía ser reducido a dos absolutos: Dios y la nada o 0 y 1, ¿por qué el lenguaje no podía ser deconstruido en una manera en que párrafos, oraciones, cláusulas y palabras pudieran ser cernidas para mayor entendimiento?

El filósofo y matemático especulaba que los humanos pronunciaban palabras y frases que encajaban en sus percepciones y emociones individuales. El sistema binario es el responsable de la existencia de todos los lenguajes de programación que conocemos actualmente, ya que no son más que vehículos que permiten escribir algoritmos fácilmente. Pero también es el sistema que refuerza los chips y circuitos de nuestras computadoras para correr esos algoritmos.

En 1965, Edmonds definió un "buen algoritmo como uno con un tiempo de ejecución polinómico". Esto condujo al surgimiento de uno de los conceptos más importantes y ramas de estudio de la Computación llamada la Teoría de la Complejidad Computacional: la NP-completitud y su pregunta fundamental, si P=NP.

La Teoría de la Complejidad Computacional

La Teoría de la Complejidad Computacional nace de la idea de medir el tiempo y espacio como una función de la longitud de la entrada se originó a principios de los 60's por Hartmanis y Stearns, después de descubrir el fallo de la máquina de Turing al medir el tiempo y la memoria requerida por una computadora.

La Teoría de la Complejidad Computacional trata de clasificar los problemas que pueden, o no pueden ser resueltos con una cantidad determinada de recursos. Tratar de dar respuesta a la siguiente pregunta: *¿Qué hace a algunos problemas computacionalmente difíciles y a otros sencillos?* Estudia la eficiencia de los algoritmos estableciendo su efectividad de acuerdo con el tiempo de corrida y al espacio requerido en la computadora, ayudando a evaluar la viabilidad de la implementación práctica en tiempo y costo.

Esta pregunta influye mucho al pensar en un método para ordenar, por medio de una computadora, una colección de datos dada y cuál de ellos es "*computacionalmente sencillo*" y por lo tanto, más eficiente y útil, por lo que a continuación se presentarán los métodos de ordenamiento a tratar, cómo se definen y en qué bases se sustenta cada algoritmo

Capítulo 3: Tipos de Ordenación

Todos los métodos por evaluar serán por ordenación interna, es decir, tipos de ordenaciones de un conjunto de datos que se encuentran almacenados en una estructura que es la memoria volátil.

Existen numerosos métodos que ordenan, también llamados por su análoga en inglés, "*sort*", los elementos de un arreglo. Los métodos pueden agruparse según la característica principal (intercambio, inserción o selección) de la operación que realiza para ordenar los datos. Todos los métodos utilizan operaciones básicas para realizar la ordenación de los elementos de un arreglo, la comparación y el intercambio de datos mismos.

Bubble Sort

El Bubble Sort es un sencillo algoritmo de ordenamiento. Funciona "revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado" (Joyanes, 2005). Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada.

Este algoritmo obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el método del intercambio directo. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillos de implementar.

Una manera simple de expresar el ordenamiento de burbuja en pseudocódigo es la siguiente:

Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso, de n términos numerados del 0 al $n-1$; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta $n-i$, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja se define como dos términos seguidos de la lista, j y $j+1$, que se comparan: si el primero es mayor que el segundo sus valores se intercambian. Esta comparación se repite en el centro de los dos bucles, dando lugar a la postre a una lista ordenada.

Puede verse que el número de repeticiones solo depende de n y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Esta es una característica de este algoritmo.

La siguiente función en C describe el pseudocódigo anterior:

```
void BubbleSort(int arreglo[], int tam)
{
   int cont=0,aux;

   do
   {
      if(arreglo[cont]>arreglo[cont+1])
      {
         aux=arreglo[cont];
         arreglo[cont]=arreglo[cont+1];
         arreglo[cont+1]=aux;
         cont=0;
      }
      else
      {
         cont++;
      }
   }
   while(cont<tam);
}
```

Para entender los parámetros de cada función y ver el funcionamiento completo del programa experimental que se usó durante la investigación, véase el anexo 1.

Insertion Sort

El Insertion Sort es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria, que también tiene algunas bases en el método anterior.

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, "cuando hay k elementos ordenados de menor a mayor, se toma el elemento $k+1$ y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño)" (Joyanes, 2005). En este punto se inserta el elemento $k+1$ debiendo desplazarse los demás elementos.

Por lo que el código en C para una función que realice el procedimiento anterior es el siguiente:

```
void InsertionSort(int arreglo[], int tam)
{
```

```
    for(int cont=1; cont<tam; cont++)
    {
        int aux1 = arreglo[cont];
        int aux2 = cont;
        while(aux2>0&&arreglo[aux2-1]>aux1)
        {
            arreglo[aux2] = arreglo[aux2-1];
            aux2--;
        }
        arreglo[aux2]=aux1;
    }
}
```

Selection Sort

El Selection Sort es un algoritmo similar al Bubble Sort y al Insertion Sort. Su funcionamiento es el siguiente:

1. Buscar el mínimo elemento de la lista
2. Intercambiarlo con el primero
3. Buscar el siguiente mínimo en el resto de la lista
4. Intercambiarlo con el segundo

Por lo que puede resumirse en “buscar el mínimo elemento entre una posición i y el final de la lista para posteriormente intercambiar el mínimo con el elemento de la posición i” (Joyanes, 2005).

De esta manera el código en C de la función en cuestión es el siguiente:

```
void SelectionSort(int arreglo[], int tam)
{
    int aux,min;
    for(int cont1=0; cont1<tam; cont1++)
    {
        min=cont1;
        for(int cont2=cont1+1; cont2<tam; cont2++)
        {
            if(arreglo[cont2]<arreglo[min])
            {
                min=cont2;
            }
        }
        aux=arreglo[cont1];
        arreglo[cont1]=arreglo[min];
        arreglo[min]=aux;
    }
}
```

Shell Sort

El Shell Sort es un algoritmo de ordenamiento denominado Shell en honor a su inventor Donald Shell. Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución. El Shell Sort es una generalización del Insertion Sort, teniendo en cuenta dos observaciones:

1. El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
2. El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. “Esto permite que un elemento haga ‘pasos más grandes’ hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños” (Moreno, 2016).

El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados. Para ejemplificar este método de ordenamiento, se propone la siguiente función en C:

```
void ShellSort(int arreglo[], int tam)
{
   int saltos, aux;
    saltos=tam/2;

      while(saltos>0)
      {
         for(int i=saltos+1;i<=tam;i++)
         {
            int j=i-saltos;
            while(j>=0)
            {
                  if(arreglo[j]>=arreglo[j+saltos])
               {
                  aux=arreglo[j];
                     arreglo[j]=arreglo[j+saltos];
                     arreglo[j+saltos]=aux;
               }
               else
               {
                  j=0;
               }
               j=j-saltos;
            }
         }
         saltos=saltos/2;
```

```
        }
}
```

Quick Sort

El Quick Sort es un algoritmo de ordenamiento creado por el científico británico en computación C. A. R. Hoare. El algoritmo tiene una alta complejidad al momento de programarlo, pues requiere algunos elementos “pivote”, sin embargo, su velocidad teórica es superior a los anteriores.

El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del arreglo de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño.
- En el peor caso, el pivote termina en un extremo de la lista. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del arreglo, y el arreglo que le pasamos está ordenado, siempre va a generar a su izquierda un arreglo vacío, lo que es ineficiente.

No es extraño que la mayoría de las optimizaciones que se aplican al algoritmo se centren en la elección del pivote.

De esta manera, el código en C para este método es el siguiente:

```
void QuickSort(int arreglo[],int first,int last)
{
   int cont1,cont2,pivot,aux;
```

```
    if(first<last)
    {
        pivot=first;
        cont1=first;
        cont2=last;

        while(cont1<cont2)
        {
            while(arreglo[cont1]<=arreglo[pivot]&&cont1<last)
            {
                cont1++;
            }
            while(arreglo[cont2]>arreglo[pivot])
            {
                cont2--;
            }
            if(cont1<cont2)
            {
                aux=arreglo[cont1];
                arreglo[cont1]=arreglo[cont2];
                arreglo[cont2]=aux;
            }
        }

        aux=arreglo[pivot];
        arreglo[pivot]=arreglo[cont2];
        arreglo[cont2]=aux;
        QuickSort(arreglo,first,cont2-1);
        QuickSort(arreglo,cont2+1,last);
    }
}
```

Merge Sort

El Merge Sort es un tipo de ordenamiento desarrollado en 1945 por John Von Neumann.

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

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

El ordenamiento por mezcla incorpora dos ideas principales para mejorar su tiempo de ejecución:

1. Una lista pequeña necesitará menos pasos para ordenarse que una lista grande.
2. Se necesitan menos pasos para construir una lista ordenada a partir de dos listas también ordenadas, que a partir de dos listas desordenadas. Por ejemplo, sólo será necesario entrelazar cada lista una vez que están ordenadas.

Debido a esto, el ordenamiento por Merge, al proponerlo en C, se utilizaron dos funciones, las cuales son las siguientes:

```
void MergeSort(int arr[], int l, int r)
{
   if (l < r)
    {
        int m = l+(r-l)/2;

        MergeSort(arr, l, m);
        MergeSort(arr, m+1, r);

        PreMerge(arr, l, m, r);
    }
}

void PreMerge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
```

```
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Radix Sort

El Radix Sort es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, Radix Sort no está limitado sólo a los enteros.

La mayor parte de los ordenadores digitales representan internamente todos sus datos como representaciones electrónicas de números binarios, por lo que procesar los dígitos de las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente. Existen dos clasificaciones de Radix Sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD).

- ✓ Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo.
- ✓ Radix sort MSD trabaja en sentido contrario.

Las representaciones de enteros que son procesadas por los algoritmos de ordenamiento se les llama a menudo "claves", que pueden existir por sí mismas o asociadas a otros datos. Radix Sort LSD usa típicamente el siguiente orden: claves cortas aparecen antes que las claves largas, y claves de la misma longitud son ordenadas de forma léxica. Esto coincide con el orden normal de las representaciones de enteros, como la secuencia "1, 2, 3, 4, 5, 6, 7, 8, 9, 10".

Radix sorts MSD usa orden léxico, que es ideal para la ordenación de cadenas de caracteres, como las palabras o representaciones de enteros de longitud fija. Una secuencia como "b, c, d, e, f, g, h, i, j, ba" será ordenada léxicamente como "b, ba, c, d, e, f, g, h, i, j".

Si se usa orden léxico para ordenar representaciones de enteros de longitud variable, entonces la ordenación de las representaciones de los números del 1 al 10 será "1, 10, 2, 3, 4, 5, 6, 7, 8, 9", como si las claves más cortas estuvieran justificadas a la izquierda y rellenadas a la derecha con espacios en blanco, para hacerlas tan largas como la clave más larga, para el propósito de este ordenamiento, cabe destacar que este método no funciona para la estructura de datos debido a que los ciclos `for` que se implementaran marcaran error debido a las matrices bidimensionales.

El ordenamiento Radix se traduce en C de la siguiente manera:

```
void RadixSort(int arreglo[], int tam)
{
   int max = arreglo[0];
   for (int i=1;i<tam;i++)
   {
      if (arreglo[i] > max)
      {
         max = arreglo[i];
      }
   }
   int m=max;

    for (int exp=1;m/exp>0;exp*=10)
   {
      int salida[tam];
      int i,count[10]={0};

       for (i=0;i<tam;i++)
       {
          count[(arreglo[i]/exp)%10]++;
       }

       for (i=1;i<10;i++)
```

```
        {
          count[i]+=count[i-1];
        }

        for (i=tam-1;i>=0;i--)
        {
            salida[count[(arreglo[i]/exp)%10]-1]=arreglo[i];
            count[(arreglo[i]/exp)%10]--;
        }

        for (i=0;i<tam;i++)
        {
          arreglo[i]=salida[i];
        }
    }
}
```

Heap Sort

El Heap Sort, también conocido como ordenamiento por montículos consiste en almacenar todos los elementos del vector a ordenar en un montículo o en inglés, heap, y luego extraer el nodo que queda como nodo raíz del montículo (cima) en sucesivas iteraciones obteniendo el conjunto ordenado. Basa su funcionamiento en una propiedad de los montículos, por la cual, la cima contiene siempre el menor elemento (o el mayor, según se haya definido el montículo) de todos los almacenados en él.

El algoritmo, después de cada extracción, recoloca en el nodo raíz o cima, la última hoja por la derecha del último nivel, lo cual destruye la propiedad heap del árbol, pero inmediatamente realiza un proceso de "descenso" del número insertado de forma que se elige a cada movimiento el mayor de sus dos hijos, con el que se intercambia. Este intercambio, realizado sucesivamente "hunde" el nodo en el árbol restaurando la propiedad montículo del árbol y dejando paso a la siguiente extracción del nodo raíz.

El algoritmo, en su implementación habitual, tiene dos fases. Primero una fase de construcción de un montículo a partir del conjunto de elementos de entrada, y después, una fase de extracción sucesiva de la cima del montículo. La implementación del almacén de datos en el heap, pese a ser conceptualmente un árbol, puede realizarse en un vector de forma fácil.

Cada nodo tiene dos hijos y por tanto, un nodo situado en la posición i del vector, tendrá a sus hijos en las posiciones $2 \times i$, y $2 \times i + 1$ suponiendo que el primer elemento del vector tiene un índice 1, es decir, la cima ocupa la posición inicial del vector y sus dos hijos la posición segunda y tercera, y así, sucesivamente. Por tanto, en la fase de ordenación, el intercambio ocurre entre el primer elemento del vector

(la raíz o cima del árbol, que es el mayor elemento de éste) y el último elemento del vector que es la hoja más a la derecha en el último nivel. El árbol pierde una hoja y por tanto reduce su tamaño en un elemento. El vector definitivo y ordenado, empieza a construirse por el final y termina por el principio.

Para convertir en montículo el arreglo, se utilizará una función definida como `heapify` y para el ordenamiento, con su nombre habitual en la siguiente propuesta de código en C:

```
void HeapSort(int arreglo[], int tam)
{
      int aux;
      for (int i = tam / 2 - 1; i >= 0; i--)
            heapify(arreglo,tam, i);

      for (int i=tam-1; i>=0; i--)
      {
            aux=arreglo[0];
            arreglo[0]=arreglo[i];
            arreglo[i]=aux;

            heapify(arreglo,i, 0);
      }
}

void heapify(int arreglo[],int tam, int i)
{
      int largest = i,aux;
      int d = 2*i + 1;
      int iz = 2*i + 2;

      if (iz < tam && arreglo[iz] > arreglo[largest])
            largest = iz;

      if (d < tam && arreglo[d] > arreglo[largest])
            largest = d;

      if (largest != i)
      {
            aux=arreglo[i];
            arreglo[i]=arreglo[largest];
            arreglo[largest]=aux;

            heapify(arreglo,tam, largest);
      }
}
```

Capítulo 4: Análisis de algoritmos

Las técnicas matemáticas básicas para analizar algoritmos son fundamentales para el tratamiento de temas avanzados de computación y analizan medios para formalizar el concepto de que un algoritmo es significativamente más eficiente que otros. El análisis de algoritmos “es una parte muy importante de las ciencias de la computación para poder analizar los requisitos de tiempo y espacio de un algoritmo para ver si existe dentro de límites aceptables” (Joyanes, 2005).

Eficiencia de un Algoritmo

Para considerar la eficiencia de un algoritmo, se puede evaluar bajo dos rasgos muy importantes:

- Evaluación de la memoria
- Evaluación del tiempo

Para evaluar un algoritmo, en cuanto a términos de memoria se refiere, Joyanes (2005) considera los siguientes factores como alta importancia al momento de realizar dicho análisis:

- ✓ Su facilidad de codificación y depuración.
- ✓ Su funcionamiento correcto para cualquier posible valor de los datos de entrada.
- ✓ Inexistencia de otro algoritmo que resuelva el problema utilizando menos recursos (tiempo en ejecutarse y memoria consumida). El recurso espacio y el recurso tiempo suelen ser contrapuestos.

La complejidad en relación al tiempo de un programa, según Eslava (2016) “es la cantidad de tiempo que se necesita para su ejecución, para lo cual es necesario considerar el "principio de la invarianza" (la eficiencia de dos implementaciones distintas de un mismo algoritmo difiere tan sólo en una constante multiplicativa)”.

El enfoque matemático para esta evaluación considera el consumo de tiempo por parte del algoritmo como una función del total de sus datos de entrada. Se define el tamaño de un ejemplar de datos de entrada, como el número de elementos necesarios. Para la evaluación de un algoritmo es importante considerar que el tiempo que tarda un programa en ejecutarse depende del tamaño del ejemplar de entrada. Hay que tener en cuenta que el tiempo de ejecución puede depender también de la entrada concreta.

Supóngase que se tratará de ordenar un ejemplar de datos de entrada compuesto por n elementos por el método de la burbuja. Considerando las acotaciones anteriores, sea $T(n)$ el tiempo de ejecución de un programa con entrada de tamaño n, así será posible valorar $T(n)$ como el número de sentencias, en el caso particular de esta investigación, en C, ejecutadas por el programa, y la evaluación, explicada por Moreno (2014) se podrá efectuar desde diferentes puntos de vista:

Peor caso: Se puede hablar de $T(n)$ como el tiempo para el peor caso, de modo que indica el tiempo peor que se puede tener. Este análisis es perfectamente adecuado para algoritmos cuyo tiempo de respuesta sea crítico.

Mejor caso: Se habla de $T(n)$ como el tiempo para el mejor caso. Indica el tiempo mejor que se puede tener.

Caso medio: Se puede computar $T(n)$ como el tiempo medio de ejecución del programa sobre todas las posibles ejecuciones de entradas de tamaño n. El tiempo de ejecución medio es a veces una medida más realista del rendimiento en la práctica, pero es, normalmente, mucho más difícil de calcular que el tiempo de ejecución en el caso peor.

Para efectos de esta investigación, se omitirán las pruebas que se puedan realizar para probar la eficiencia de cada método según la memoria, pero se hará énfasis en la evaluación del tiempo, tomando en cuenta que se realizarán evaluaciones para obtener una ponderación del caso medio, peor caso y mejor caso de cada método de ordenamiento, realizando diez evaluaciones para así obtener una media de los datos obtenidos. Para expresar la complejidad computacional obtenida a partir de los datos anteriores se empleará la Notación O-grande.

La Notación O-grande

El tiempo de ejecución de un programa se escribe normalmente utilizando la notación "O-grande" que está diseñada para expresar factores constantes tales como:

- Número medio de instrucciones máquina que genera un compilador determinado.
- Número medio de instrucciones máquina por segundo que ejecuta una computadora específica.

Así, se dirá que “un algoritmo determinado emplea un tiempo $O(n)$ que se lee ‘O grande de n’ o bien ‘O de n’ y que informalmente significa ‘algunos tiempos constantes n ‘” (Joyanes, 2005).

Para poder comprender mejor esta función que se está tratando de construir a partir de una n dada como la masa de datos a analizar y el tiempo que le tomará ejecutar el programa, se deben de incluir implícitamente los conceptos del cálculo matemático diseñados para la asociación de dos conjuntos por medio de una función, lo cual se hará a continuación.

“Sea $f(n) = T(n)$ el tiempo de ejecución de algún programa, medido como una función de la entrada de tamaño n. Sea $f(n)$ una función definida sobre números naturales. Se dice "$f(n)$ es $O(g(n))$", si $f(n)$ es menor o igual que una constante de tiempo c multiplicada por $g(n)$, excepto posiblemente para algunos valores pequeños de n. De modo más riguroso, se dice que $f(n) \in O(g(n))$ si existe un

entero n_0 y una constante real $c > 0$ tal que si un natural $n \geq n_0$ se tiene que $f(n) \leq c \cdot g(n)$" (Joyanes, 2005).

Esto quiere decir que, específicamente, la notación $f(n) = O(g(n))$ significa que se establece una igualdad que relaciona al valor absoluto tal que $|f(n)| \leq c|g(n)|$ para $n \geq n_0$. Por consiguiente, se deduce que $|g(n)|$ es un límite superior para $|f(n)|$. La función que se suele considerar es la más próxima que actúa como límite de $f(n)$ y así, se puede trabajar y deducir otras propiedades bajo su definición formal, la cual afirma que $f(n) = O(g(n))$ si existen constantes $c > 0$ y n Es decir $f(n)$ es $O(g(n))$ si

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c \geq 0$$

Los tipos de complejidad más conocidos son los siguientes:

Notación	Nombre
$O(1)$	Orden constante
$O(\log\log n)$	Orden sublogarítmico
$O(\log n)$	Orden logarítmico
$O(\sqrt{n})$	Orden sublineal
$O(n)$	Orden lineal o de primer orden
$O(n \cdot \log n)$	Orden lineal logarítmica
$O(n^2)$	Orden cuadrático o segundo orden
$O(n^3)$	Orden cúbica o tercer orden
$O(n^c)$	Orden potencial fija
$O(c^n), \quad n > 1$	Orden exponencial
$O(n!)$	Orden factorial
$O(n^n)$	Orden potencial exponencial

Capítulo 5: Experimentación

A modo de práctica empírica, este capítulo presenta la experimentación que se llevó a cabo para determinar la eficacia de cada algoritmo y definir el algoritmo óptimo según diferentes situaciones.

Objetivo: Conocer qué algoritmo es más eficaz para distintas masas de información a procesar.

Pregunta de Investigación: ¿Qué algoritmo de ordenamiento es el más eficiente para jerarquizar información?

Método (Variables):

Dependiente: El tiempo de ordenamiento de los elementos.

Independiente: El tipo de algoritmo de ordenamiento utilizado.

Controlada: La cantidad de datos a ordenar.

Procedimiento:

1.- Se codificará cada código junto con un cronómetro

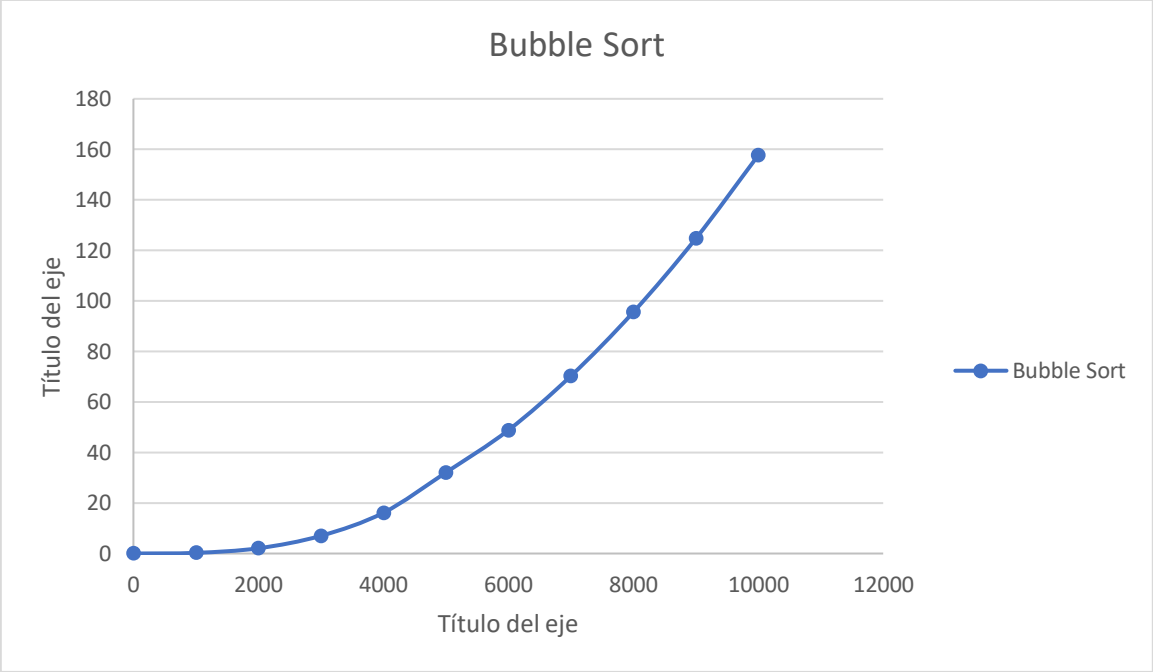
2.- Se probará por separado cada algoritmo en cada caso, iniciando con 100 elementos, y terminando donde se crea conveniente y pueda predecirse una función para cada algoritmo, donde el tope máximo serán 10, 000 datos.

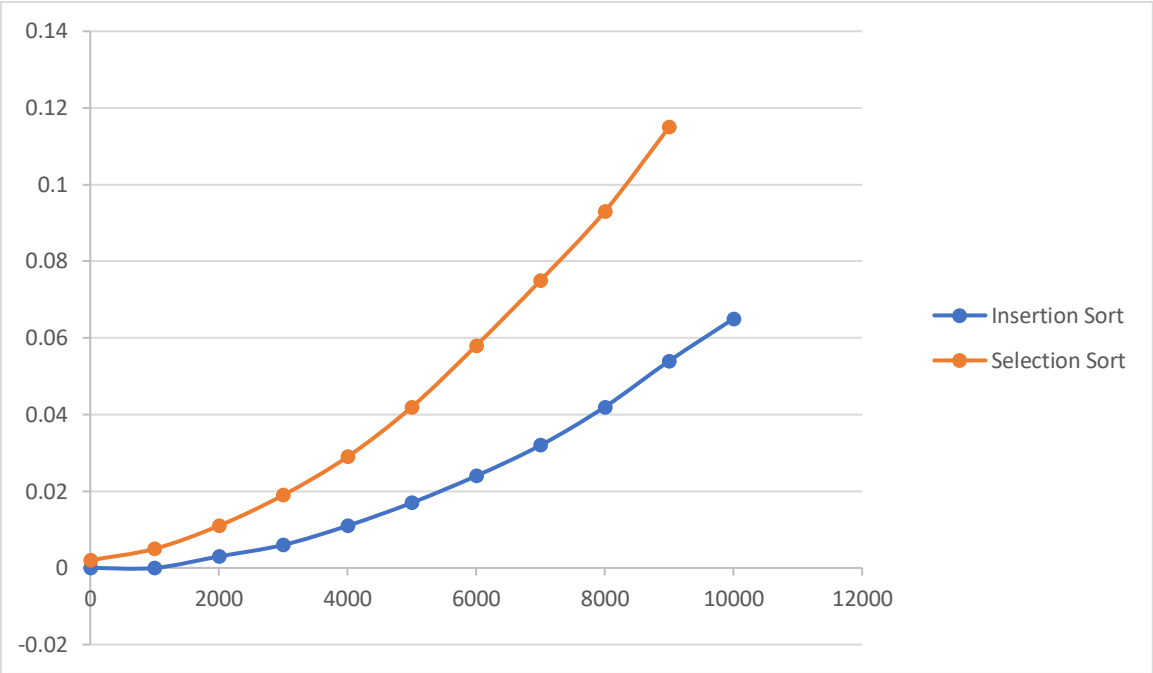
Obtención y Procesamiento de Datos: Para el procesamiento de datos, se organizó la información en tablas que relacionan los datos donde se promediaron todos los intentos y después se promediaron los mejores casos, casos promedios y peores casos, obteniendo los datos resultantes, información con la cual se graficaron los elementos visuales que se encuentran a continuación.

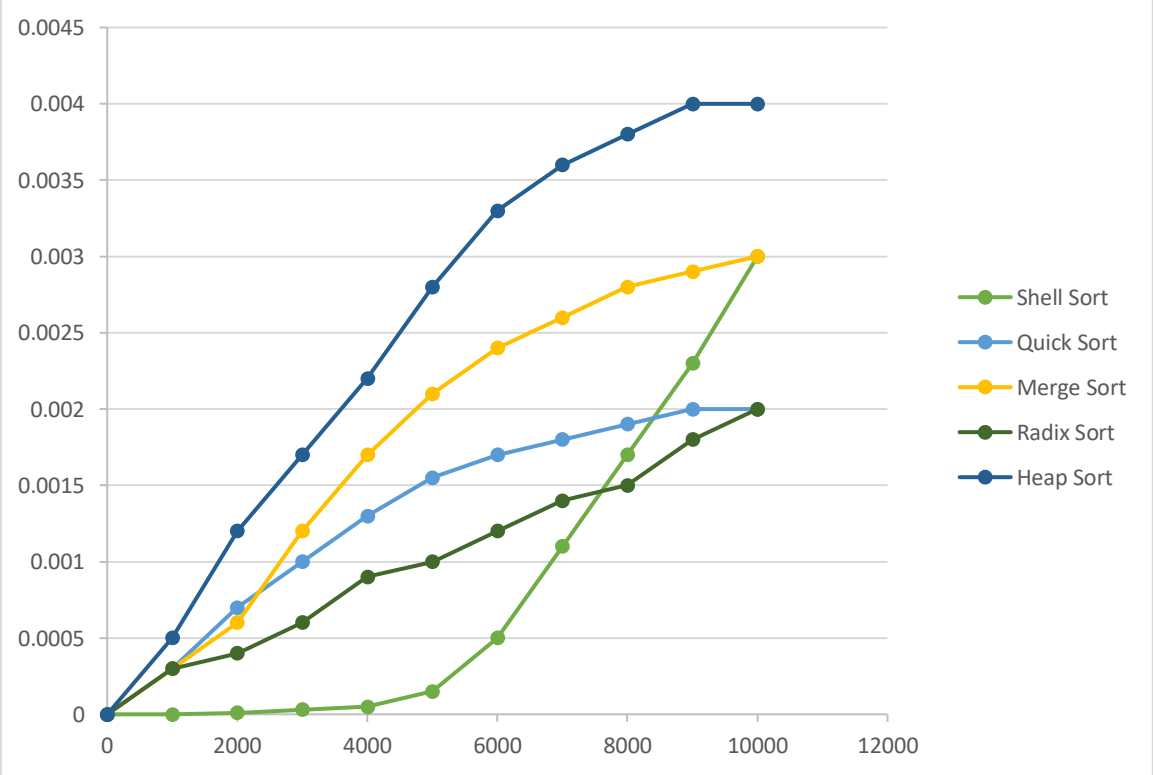
Bubble Sort		**Insertion Sort**		**Selection Sort**		**Shell Sort**	
1,000	0.261	1,000	0	1,000	0.002	1,000	0
2,000	2.082	2,000	0.003	2,000	0.005	2,000	0.00001
3,000	7.016	3,000	0.006	3,000	0.011	3,000	0.00003
4,000	16.133	4,000	0.011	4,000	0.019	4,000	0.00005
5,000	32.067	5,000	0.017	5,000	0.029	5,000	0.00015
6,000	48.808	6,000	0.024	6,000	0.042	6,000	0.0005
7,000	70.253	7,000	0.032	7,000	0.058	7,000	0.0011
8,000	95.540	8,000	0.042	8,000	0.075	8,000	0.0017
9,000	124.667	9,000	0.054	9,000	0.093	9,000	0.0023
10,000	157.639	10,000	0.065	10,000	0.115	10,000	0.003
Quick Sort		**Merge Sort**		**Radix Sort**		**Heap Sort**	
1,000	0.0003	1,000	0.0003	1,000	0.0003	1,000	0.0005
2,000	0.0007	2,000	0.0006	2,000	0.0004	2,000	0.0012
3,000	0.001	3,000	0.0012	3,000	0.0006	3,000	0.0017

4,000	0.0013	4,000	0.0017	4,000	0.0009	4,000	0.0022
5,000	0.00155	5,000	0.0021	5,000	0.001	5,000	0.0028
6,000	0.0017	6,000	0.0024	6,000	0.0012	6,000	0.0033
7,000	0.0018	7,000	0.0026	7,000	0.0014	7,000	0.0036
8,000	0.0019	8,000	0.0028	8,000	0.0015	8,000	0.0038
9,000	0.002	9,000	0.0029	9,000	0.0018	9,000	0.004
10,000	0.002	10,000	0.003	10,000	0.002	10,000	0.004

A partir de los datos anteriores, se realizaron las siguientes gráficas







Pueden verse algunas gráficas muy similares a gráficas lineales, cuadráticas, exponenciales o logarítmicas, sin embargo, ¿cómo se demuestra que éstas se apegan más a un tipo de gráfica? Para ello, se empleará una prueba de estadística llamado regresión. Esta prueba permite, bajo distintas fórmulas, realizar regresión lineal, cuadrática, logarítmica, exponencial, entre muchos otros tipos, arrojando un

valor de qué tan certero se encontraría cada conjunto de datos de ser considerada como una gráfica al 100% de cierto tipo, ya sea lineal, logarítmica, cuadrática, exponencial, entre otros tipos. Este valor que arroja se le llama “r” que es el coeficiente de correlación entre las variables. El coeficiente de correlación va desde 0 a 1, donde 0 significa que hay correlación nula y 1 es una correlación completa entre los datos. Evidentemente, entre más se acerque el coeficiente de correlación a 1, es más fiable creer en dicha regresión

Los cálculos de las pruebas de regresión se realizaron en una calculadora graficadora modelo fx-CG10 y a continuación se presenta una tabla con cada método de ordenamiento evaluado en cada tipo de regresión y el valor del coeficiente de correlación obtenido según la prueba.

Método	Regresión lineal $O(n)$	Regresión cuadrática $O(n^2)$	Regresión logarítmica $O(n \log n)$
Bubble Sort	0.962	0.9998	0.8383
Insertion Sort	0.943	0.9996	0.8094
Selection Sort	0.917	0.9992	0.8426
Shell Sort	0.952	0.9991	0.8197
Quick Sort	0.901	0.8240	0.9993
Merge Sort	0.898	0.8174	0.9990
Radix Sort	0.9995	0.8385	0.8073
Heap Sort	0.910	0.8439	0.9998

Como puede observarse, la prueba de correlación tiene valores altísimos y extremadamente cercanos a 1 en ciertos métodos según la prueba hecha, por lo que, bajo las deducciones de la tabla anterior, puede deducirse que los métodos de ordenamiento tienen un comportamiento de la siguiente forma:

Bubble Sort	$O(n^2)$
Insertion Sort	$O(n^2)$
Selection Sort	$O(n^2)$
Shell Sort	$O(n^2)$
Quick Sort	$O(n \log n)$
Merge Sort	$O(n \log n)$
Radix Sort	$O(nk)$
Heap Sort	$O(n \log n)$

De aquí se concluye que los algoritmos más eficientes de ordenamiento son el ordenamiento por método Quick, Merge y Heap, mientras que los algoritmos por el método Bubble, Insertion y Selection poseen una complejidad mayor y llegan a tomar más tiempo para el ordenamiento de datos. Claro, esto, hay que aclararse que, bajo un promedio entre los mejores, peores y casos promedio. Cuando estos

se analizan por separado, otros métodos pueden resultar más útiles que los ya mencionados aquí (véase anexo 2 para observar los tiempos por separado).

Capítulo 6: Conclusiones

Joel Alejandro Espinoza Sánchez: Bajo la presente investigación, el equipo pudo comprobar cuál algoritmo era el más eficiente y experimentar, así como comprender por qué lo son. Tuvimos la oportunidad de analizar cada algoritmo de ordenamiento mientras lo construíamos para el proyecto y así entendimos qué elementos hacen que un algoritmo de ordenamiento tenga una mayor o una menor complejidad computacional y qué significa cada tipo diferente de complejidad, pues observamos que una complejidad computacional de orden cuadrático es menos eficiente que una complejidad computacional de orden logarítmico, pues así, las complejidades logarítmicas consiguen un menor tiempo de respuestas, tales como el Quick Sort o el Shell Sort. Métodos muy útiles para ordenar grandes masas de datos.

Dariana Gómez Garza: En esta investigación hecha por mis compañeros de equipo y yo, pudimos darnos cuenta de la eficacia que hay entre algunos métodos de ordenación y qué tan útiles son para nuestras prácticas de programación. Pudimos notar una gran diferencia en cuanto a la cantidad de datos que podría ordenar y el tiempo que se tardó cada método, básicamente la eficacia que tiene cada uno para hacer su trabajo. En nuestro trabajo, encontramos que hay varios tipos de métodos y nosotros tan sólo conocemos algunos, nos falta experimentar con muchísimos más.

Cuando hicimos la prueba, para nuestra sorpresa, el más rápido y el que (por ahora) ha soportado mayor cantidad de datos es el método Quicksort, nos asombramos con toda la cantidad de datos que éste inofensivo método pudo ordenar en tan poco tiempo. Y el que más nos decepcionó, a pesar de su facilidad al implementarlo en un código, fue el Bubble sort ☹.

Valeria Macías Soto: Con base a la investigación realizada para el proyecto, pudimos aprender un poco más acerca de los algoritmos, y su utilidad. Además de entender mejor el funcionamiento de diferentes tipos de métodos de ordenamiento que, al medir sus tiempos de ordenamiento y realizar un análisis sobre los datos obtenidos en la experimentación, concluimos que los algoritmos de ordenamiento más eficaces son los que poseen un orden logarítmico, por lo tanto son métodos de ordenamiento que pueden ordenar mayor cantidad de datos que los algoritmos menos eficientes (que son los de orden cuadrático), ya que estos tomarán un tiempo demasiado largo para ordenar un número menor de datos. Debido a esto, el programar los códigos aumenta su complejidad cuando su eficacia es mejor; por lo tanto, dependerá de las condiciones de los datos que se requieren ordenar, de los gustos y necesidades del programador, para tomar la decisión del algoritmo que se utilizará para realizar el programa.

Referencias Bibliográficas

- Joyanes, A. L. (2005). *Estructuras de datos en c*. Retrieved from https://ebookcentral.proquest.com
- Joyanes, A. L., Castillo, S. A., & Sánchez, G. L. (2005). *C algoritmos, programación y estructuras de datos*. Retrieved from https://ebookcentral.proquest.com
- Eslava, M. V. J. (2016). *Aprendiendo a programar paso a paso con c*. Retrieved from https://ebookcentral.proquest.com
- Moreno, P. J. C. (2014). *Programación*. Retrieved from https://ebookcentral.proquest.com

Anexos

Anexo 1: Código de C completo:

```
#include <stdio.h>
#include <stdlib.h>
#include <locale.h>
#include <windows.h>
#include <time.h>

int data10000000[10000000];

void gotoxy(int x,int y);
void CasoMejor(int arreglo[], int tam);
void CasoPromedio(int arreglo[], int tam);
void CasoPeor(int arreglo[], int tam);
void Imprimir(int arreglo[], int tam);
void BubbleSort(int arreglo[], int tam); //100 a 10,000
void InsertionSort(int arreglo[], int tam); //100 a 100,000
void SelectionSort(int arreglo[], int tam); //100 a 100,000
void ShellSort(int arreglo[], int tam); //100 a 10,000,000
void QuickSort(int arreglo[],int first,int last); //100 a 10,000,000
void MergeSort(int arr[], int l, int r); // 100 a 500,000
   void PreMerge(int arr[], int l, int m, int r);
//void DistributionSort(); Radix es un tipo de Distribution
void RadixSort(int arreglo[], int tam); // 100 a 500,000
void HeapSort(int arreglo[], int tam); //100 a 10,000,000
   void heapify(int arreglo[],int tam, int i);

main()
{
   setlocale(LC_ALL,"");
   srand(time(NULL));

   printf("Comenzando evaluación...\n\n");

   Sleep(3000);

   float tiempoglobal;
   clock_t inicioglobal,finglobal;
   inicioglobal=clock();

   float tiempo;
```

```
   clock_t inicio,fin;

   int caso=1,sort=1,renglon=3,verif=0;
   long int tam=100;

   gotoxy(0,2);
   printf("CASO MEJOR:");
   renglon++;
   do
   {
      if(caso==1)
      {
         CasoMejor(data10000000,tam);
         if(sort==1)
         {
            if(tam<=100000)
            {
               inicio=clock();
               BubbleSort(data10000000,tam);
               fin=clock();
               tiempo=((float)fin-(float)inicio)/CLK_TCK;
               gotoxy(0,renglon);
               printf("Tiempo del Bubble Sort para %d datos:
%f\n",tam,tiempo);
               renglon++;
            }
            if(tam==100000)
            {
               sort=2;
               tam=100;
               renglon++;
            }

         }
         if(sort==2)
         {
            if(tam<=100000)
            {
               inicio=clock();
               InsertionSort(data10000000,tam);
               fin=clock();
               tiempo=((float)fin-(float)inicio)/CLK_TCK;
```

```
                gotoxy(0,renglon);
                printf("Tiempo del Insertion Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==100000)
            {
                sort=3;
                tam=100;
                renglon++;
            }
        }
        if(sort==3)
        {
            if(tam<=200000)
            {
                inicio=clock();
                SelectionSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Selection Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==200000)
            {
                sort=4;
                tam=100;
                renglon++;
            }
        }
        if(sort==4)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                ShellSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
```

```
                printf("Tiempo del Shell Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==10000000)
            {
                sort=5;
                tam=100;
                renglon++;
            }
        }
        if(sort==5)
        {
            if(tam<=30000)
            {
                inicio=clock();
                QuickSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Quick Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==30000)
            {
                sort=6;
                tam=100;
                renglon++;
            }
        }
        if(sort==6)
        {
            if(tam<=500000)
            {
                inicio=clock();
                MergeSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Merge Sort para %d datos:
%f\n",tam,tiempo);
```

```
                renglon++;
            }
            if(tam==500000)
            {
                sort=7;
                tam=100;
                renglon++;
            }
        }
        if(sort==7)
        {
            if(tam<=500000)
            {
                inicio=clock();
                RadixSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Radix Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==500000)
            {
                sort=8;
                tam=100;
                renglon++;
            }
        }
        if(sort==8)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                HeapSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Heap Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
```

```
            if(tam==10000000)
            {
                sort=1;
                caso=2;
                tam=100;
                renglon++;
                gotoxy(0,renglon);
                printf("CASO PROMEDIO:");
                renglon++;
            }
        }
    }
    if(caso==2)
    {
        CasoPromedio(data10000000,tam);
        if(sort==1)
        {
            if(tam<=5000)
            {
                inicio=clock();
                BubbleSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Bubble Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==5000)
            {
                sort=2;
                tam=100;
                renglon++;
            }

        }
        if(sort==2)
        {
            if(tam<=200000)
            {
                inicio=clock();
                InsertionSort(data10000000,tam);
```

```
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Insertion Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==200000)
            {
                sort=3;
                tam=100;
                renglon++;
            }
        }
        if(sort==3)
        {
            if(tam<=200000)
            {
                inicio=clock();
                SelectionSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Selection Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==200000)
            {
                sort=4;
                tam=100;
                renglon++;
            }
        }
        if(sort==4)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                ShellSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
```

```
                gotoxy(0,renglon);
                printf("Tiempo del Shell Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==10000000)
            {
                sort=5;
                tam=100;
                renglon++;
            }
        }
        if(sort==5)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                QuickSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Quick Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==10000000)
            {
                sort=6;
                tam=100;
                renglon++;
            }
        }
        if(sort==6)
        {
            if(tam<=500000)
            {
                inicio=clock();
                MergeSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
```

```
                printf("Tiempo del Merge Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==500000)
            {
                sort=7;
                tam=100;
                renglon++;
            }
        }
        if(sort==7)
        {
            if(tam<=500000)
            {
                inicio=clock();
                RadixSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Radix Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==500000)
            {
                sort=8;
                tam=100;
                renglon++;
            }
        }
        if(sort==8)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                HeapSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Heap Sort para %d datos:
%f\n",tam,tiempo);
```

```
                renglon++;
            }
            if(tam==10000000)
            {
                sort=1;
                caso=3;
                tam=100;
                renglon++;
                gotoxy(0,renglon);
                printf("CASO PEOR:");
                renglon++;
            }
        }
    }
    if(caso==3)
    {
        CasoPeor(data10000000,tam);
        if(sort==1)
        {
            if(tam<=5000)
            {
                inicio=clock();
                BubbleSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Bubble Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==5000)
            {
                sort=2;
                tam=100;
                renglon++;
            }
        }
        if(sort==2)
        {
            if(tam<=200000)
            {
                inicio=clock();
```

```
                InsertionSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Insertion Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==200000)
            {
                sort=3;
                tam=100;
                renglon++;
            }
        }
        if(sort==3)
        {
            if(tam<=200000)
            {
                inicio=clock();
                SelectionSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Selection Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==200000)
            {
                sort=4;
                tam=100;
                renglon++;
            }
        }
        if(sort==4)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                ShellSort(data10000000,tam);
                fin=clock();
```

```
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Shell Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==10000000)
            {
                sort=5;
                tam=100;
                renglon++;
            }
        }
        if(sort==5)
        {
            if(tam<=30000)
            {
                inicio=clock();
                QuickSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Quick Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==30000)
            {
                sort=6;
                tam=100;
                renglon++;
            }
        }
        if(sort==6)
        {
            if(tam<=500000)
            {
                inicio=clock();
                MergeSort(data10000000,0,tam-1);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
```

```
                printf("Tiempo del Merge Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==500000)
            {
                sort=7;
                tam=100;
                renglon++;
            }
        }
        if(sort==7)
        {
            if(tam<=500000)
            {
                inicio=clock();
                RadixSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Radix Sort para %d datos:
%f\n",tam,tiempo);
                renglon++;
            }
            if(tam==500000)
            {
                sort=8;
                tam=100;
                renglon++;
            }
        }
        if(sort==8)
        {
            if(tam<=10000000)
            {
                inicio=clock();
                HeapSort(data10000000,tam);
                fin=clock();
                tiempo=((float)fin-(float)inicio)/CLK_TCK;
                gotoxy(0,renglon);
                printf("Tiempo del Heap Sort para %d datos:
%f\n",tam,tiempo);
```

```
                renglon++;
            }
            if(tam==10000000)
            {
                verif=1;
                renglon++;
            }
        }
    }
    if(tam>=10000000&&tam<100000000)
    {
        tam=tam+10000000;
    }
    if(tam>=1000000&&tam<10000000)
    {
        tam=tam+1000000;
    }
    if(tam>=100000&&tam<1000000)
    {
        tam=tam+100000;
    }
    if(tam>=10000&&tam<100000)
    {
        tam=tam+10000;
    }
    if(tam>=1000&&tam<10000)
    {
        tam=tam+1000;
    }
    if(tam>=100&&tam<1000)
    {
        tam=tam+100;
    }
}
while(verif==0);
}

void gotoxy(int x,int y)
{
    HANDLE hcon;
     hcon = GetStdHandle(STD_OUTPUT_HANDLE);
     COORD dwPos;
```

```
    dwPos.X = x;
    dwPos.Y= y;
    SetConsoleCursorPosition(hcon,dwPos);
}

void CasoMejor(int arreglo[], int tam)
{
   int cont=0;
   do
   {
      arreglo[cont]=cont+1;
      cont++;
      if(cont==tam-2)
      {
         arreglo[tam-2]=tam;
         arreglo[tam-1]=tam-1;
         break;
      }
   }
   while(cont<tam);
}

void CasoPromedio(int arreglo[], int tam)
{
   int cont=0;
   srand(time(NULL));
   do
   {
      arreglo[cont]=1+rand()%(10000000);
      cont++;
   }
   while(cont<tam);
}

void CasoPeor(int arreglo[], int tam)
{
   int cont=0;
   do
   {
      arreglo[cont]=tam-cont;
      cont++;
   }
```

```
   while(cont<tam);
}

void Imprimir(int arreglo[], int tam)
{
   int cont=0;
   do
   {
      printf("[%d]\n",arreglo[cont]);
      cont++;
   }
   while(cont<tam);
}

void BubbleSort(int arreglo[], int tam)
{
   int cont=0,aux;

   do
   {
      if(arreglo[cont]>arreglo[cont+1])
      {
         aux=arreglo[cont];
         arreglo[cont]=arreglo[cont+1];
         arreglo[cont+1]=aux;
         cont=0;
      }
      else
      {
         cont++;
      }
   }
   while(cont<tam);
}

void InsertionSort(int arreglo[], int tam)
{
   for(int cont=1; cont<tam; cont++)
   {
      int aux1 = arreglo[cont];
      int aux2 = cont;
      while(aux2>0&&arreglo[aux2-1]>aux1)
```

```
      {
         arreglo[aux2] = arreglo[aux2-1];
         aux2--;
      }
      arreglo[aux2]=aux1;
   }
}

void SelectionSort(int arreglo[], int tam)
{
   int aux,min;
   for(int cont1=0; cont1<tam; cont1++)
   {
      min=cont1;
      for(int cont2=cont1+1; cont2<tam; cont2++)
      {
         if(arreglo[cont2]<arreglo[min])
         {
            min=cont2;
         }
      }
      aux=arreglo[cont1];
      arreglo[cont1]=arreglo[min];
      arreglo[min]=aux;
   }
}

void ShellSort(int arreglo[], int tam)
{
   int saltos, aux;
    saltos=tam/2;

      while(saltos>0)
      {
         for(int i=saltos+1;i<=tam;i++)
         {
            int j=i-saltos;
            while(j>=0)
            {
                  if(arreglo[j]>=arreglo[j+saltos])
               {
                  aux=arreglo[j];
```

```
                    arreglo[j]=arreglo[j+saltos];
                    arreglo[j+saltos]=aux;
                }
                else
                {
                   j=0;
                }
                j=j-saltos;
            }
         }
         saltos=saltos/2;
      }
}

void QuickSort(int arreglo[],int first,int last)
{
   int cont1,cont2,pivot,aux;

   if(first<last)
   {
      pivot=first;
      cont1=first;
      cont2=last;

      while(cont1<cont2)
      {
         while(arreglo[cont1]<=arreglo[pivot]&&cont1<last)
         {
            cont1++;
         }
         while(arreglo[cont2]>arreglo[pivot])
         {
            cont2--;
         }
         if(cont1<cont2)
         {
            aux=arreglo[cont1];
            arreglo[cont1]=arreglo[cont2];
            arreglo[cont2]=aux;
         }
      }
```

```
      aux=arreglo[pivot];
      arreglo[pivot]=arreglo[cont2];
      arreglo[cont2]=aux;
      QuickSort(arreglo,first,cont2-1);
      QuickSort(arreglo,cont2+1,last);
   }
}

void MergeSort(int arr[], int l, int r)
{
   if (l < r)
    {
        int m = l+(r-l)/2;

        MergeSort(arr, l, m);
        MergeSort(arr, m+1, r);

        PreMerge(arr, l, m, r);
    }
}

void PreMerge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;

    int L[n1], R[n2];

    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
```

```
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void RadixSort(int arreglo[], int tam)
{
   int max = arreglo[0];
   for (int i=1;i<tam;i++)
   {
      if (arreglo[i] > max)
      {
         max = arreglo[i];
      }
   }
   int m=max;

    for (int exp=1;m/exp>0;exp*=10)
   {
      int salida[tam];
      int i,count[10]={0};
```

```
        for (i=0;i<tam;i++)
        {
           count[(arreglo[i]/exp)%10]++;
        }

        for (i=1;i<10;i++)
        {
           count[i]+=count[i-1];
        }

        for (i=tam-1;i>=0;i--)
        {
            salida[count[(arreglo[i]/exp)%10]-1]=arreglo[i];
            count[(arreglo[i]/exp)%10]--;
        }

        for (i=0;i<tam;i++)
        {
           arreglo[i]=salida[i];
        }
    }
}

void HeapSort(int arreglo[], int tam)
{
    int aux;
    for (int i = tam / 2 - 1; i >= 0; i--)
        heapify(arreglo,tam, i);

    for (int i=tam-1; i>=0; i--)
    {
        aux=arreglo[0];
        arreglo[0]=arreglo[i];
        arreglo[i]=aux;

        heapify(arreglo,i, 0);
    }
}

void heapify(int arreglo[],int tam, int i)
{
    int largest = i,aux;
```

```
    int d = 2*i + 1;
    int iz = 2*i + 2;

    if (iz < tam && arreglo[iz] > arreglo[largest])
       largest = iz;

    if (d < tam && arreglo[d] > arreglo[largest])
       largest = d;

    if (largest != i)
    {
       aux=arreglo[i];
       arreglo[i]=arreglo[largest];
       arreglo[largest]=aux;

       heapify(arreglo,tam, largest);
    }
}
```

Anexo 2: Tablas de datos de los tiempos

En cada caso y con cada dato, la ejecución del programa fue elaborada 10 veces para realizar un promedio. Los promedios son los siguientes:

Bubble Sort			
Datos	Caso Mejor	Caso promedio	Caso peor
0	0	0	0
100	0.000000	0.000000	0.001000
200	0.000000	0.002000	0.004000
300	0.001000	0.008000	0.013000
400	0.000000	0.018000	0.028000
500	0.001000	0.033000	0.051000
600	0.001000	0.056000	0.087000
700	0.001000	0.089000	0.140000
800	0.003000	0.136000	0.207000
900	0.002000	0.186000	0.294000
2,000	0.004000	0.261000	0.393000
3,000	0.029000	2.082000	
4,000	0.025000	7.016000	
5,000	0.030000	16.132999	
6,000	0.042000		
7,000	0.056000		
8,000	0.075000		
9,000	0.095000		
10,000	0.116000		
20,000	0.465000		
30,000	1.043000		
40,000	1.851000		
50,000	2.905000		
60,000	4.166000		
70,000	5.673000		
80,000	7.452000		
90,000	9.463000		
100,000	11.60900		

Insertion Sort			
Datos	Caso Mejor	Caso promedio	Caso peor
0	0	0	0
100	0.000000	0.000000	0.000000

200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.000000	0.000000	0.000000
800	0.000000	0.001000	0.001000
900	0.000000	0.000000	0.001000
2,000	0.000000	0.003000	0.001000
3,000	0.000000	0.000000	0.012000
4,000	0.000000	0.006000	0.022000
5,000	0.000000	0.011000	0.033000
6,000	0.001000	0.017000	0.047000
7,000	0.000000	0.032000	0.064000
8,000	0.000000	0.043000	0.083000
9,000	0.000000	0.053000	0.107000
10,000	0.000000	0.065000	0.131000
20,000	0.000000	0.258000	0.517000
30,000	0.000000	0.588000	1.161000
40,000	0.001000	1.029000	2.054000
50,000	0.000000	1.612000	3.222000
60,000	0.000000	2.314000	4.638000
70,000	0.000000	3.181000	6.316000
80,000	0.000000	4.159000	8.228000
90,000	0.001000	5.223000	10.44400
100,000	0.000000	6.453000	12.91000
200,000	0.001000	25.768999	51.546001
300,000	0.001000		
400,000	0.002000		
500,000	0.001000		
600,000	0.002000		
700,000	0.002000		
800,000	0.002000		
900,000	0.003000		
1,000,000	0.004000		
2,000,000	0.007000		
3,000,000	0.010000		
4,000,000	0.013000		
5,000,000	0.016000		
6,000,000	0.020000		
7,000,000	0.023000		

8,000,000	0.027000		
9,000,000	0.030000		
10,000,000	0.035000		

Selection sort			
Datos	Caso Mejor	Caso promedio	Caso peor
0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.001000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.001000
700	0.000000	0.000000	0.001000
800	0.001000	0.001000	0.000000
900	0.001000	0.001000	0.001000
2,000	0.005000	0.005000	0.005000
3,000	0.001000	0.011000	0.010000
4,000	0.018000	0.019000	0.018000
5,000	0.029000	0.029000	0.027000
6,000	0.042000	0.042000	0.040000
7,000	0.056000	0.058000	0.054000
8,000	0.074000	0.075000	0.070000
9,000	0.093000	0.093000	0.087000
10,000	0.115000	0.115000	0.110000
20,000	0.463000	0.458000	0.435000
30,000	1.029000	1.033000	0.979000
40,000	1.832000	1.829000	1.741000
50,000	2.864000	2.860000	2.713000
60,000	4.113000	4.106000	3.912000
70,000	5.605000	5.594000	5.345000
80,000	7.317000	7.290000	6.963000
90,000	9.279000	9.313000	8.77200
100,000	11.423000	11.457000	10.87500
200,000	45.717999	45.751999	43.780998

Sell Sort			
Datos	Caso Mejor	Caso promedio	Caso peor

0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.000000	0.000000	0.000000
800	0.000000	0.001000	0.000000
900	0.000000	0.000000	0.000000
2,000	0.001000	0.001000	0.000000
3,000	0.001000	0.001000	0.001000
4,000	0.000000	0.000000	0.000000
5,000	0.001000	0.001000	0.000000
6,000	0.001000	0.001000	0.001000
7,000	0.000000	0.001000	0.000000
8,000	0.000000	0.002000	0.000000
9,000	0.000000	0.002000	0.000000
10,000	0.001000	0.002000	0.001000
20,000	0.001000	0.004000	0.001000
30,000	0.002000	0.007000	0.002000
40,000	0.002000	0.010000	0.003000
50,000	0.003000	0.014000	0.004000
60,000	0.003000	0.016000	0.006000
70,000	0.004000	0.021000	0.007000
80,000	0.005000	0.025000	0.007000
90,000	0.005000	0.027000	0.008000
100,000	0.005000	0.031000	0.009000
200,000	0.012000	0.071000	0.020000
300,000	0.018000	0.112000	0.031000
400,000	0.023000	0.166000	0.040000
500,000	0.030000	0.206000	0.051000
600,000	0.040000	0.285000	0.065000
700,000	0.045000	0.325000	0.076000
800,000	0.050000	0.407000	0.085000
900,000	0.057000	0.454000	0.100000
1,000,000	0.063000	0.513000	0.108000
2,000,000	0.132000	1.349000	0.230000
3,000,000	0.211000	2.623000	0.361000
4,000,000	0.274000	3.859000	0.486000

5,000,000	0.362000	5.864000	0.621000
6,000,000	0.439000	7.671000	0.762000
7,000,000	0.504000	9.182000	0.881000
8,000,000	0.573000	11.507000	1.011000
9,000,000	0.671000	15.721000	1.143000
10,000,000	0.752000	18.764999	1.319000

Quick Sort			
Datos	Caso Mejor	Caso promedio	Caso peor
0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.001000	0.000000	0.000000
800	0.001000	0.001000	0.000000
900	0.001000	0.000000	0.001000
1,000	0.001000	0.000000	0.001000
2,000	0.004000	0.000000	0.005000
3,000	0.009000	0.000000	0.010000
4,000	0.017000	0.000000	0.017000
5,000	0.025000	0.001000	0.027000
6,000	0.036000	0.001000	0.039000
7,000	0.049000	0.000000	0.054000
8,000	0.065000	0.001000	0.068000
9,000	0.081000	0.001000	0.087000
10,000	0.100000	0.001000	0.107000
20,000	0.401000	0.002000	0.430000
30,000	0.899000	0.004000	
40,000		0.005000	
50,000		0.006000	
60,000		0.007000	
70,000		0.008000	
80,000		0.010000	
90,000		0.010000	
100,000		0.012000	
200,000		0.025000	

300,000		0.038000	
400,000		0.052000	
500,000		0.067000	
600,000		0.080000	
700,000		0.097000	
800,000		0.115000	
900,000		0.133000	
1,000,000		0.154000	
2,000,000		0.386000	
3,000,000		0.680000	
4,000,000		1.050000	
5,000,000		1.489000	
6,000,000		1.996000	
7,000,000		2.584000	
8,000,000		3.233000	
9,000,000		3.949000	
10,000,000		4.742000	

Merge Sort			
Datos	**Caso Mejor**	**Caso promedio**	**Caso peor**
0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.000000	0.000000	0.000000
800	0.000000	0.000000	0.000000
900	0.000000	0.000000	0.000000
2,000	0.000000	0.000000	0.001000
3,000	0.000000	0.000000	0.000000
4,000	0.000000	0.000000	0.000000
5,000	0.000000	0.001000	0.000000
6,000	0.000000	0.001000	0.000000
7,000	0.001000	0.001000	0.001000
8,000	0.001000	0.001000	0.001000
9,000	0.001000	0.002000	0.001000
10,000	0.001000	0.001000	0.001000
20,000	0.002000	0.003000	0.001000

30,000	0.003000	0.005000	0.003000
40,000	0.004000	0.006000	0.004000
50,000	0.004000	0.008000	0.005000
60,000	0.006000	0.009000	0.006000
70,000	0.007000	0.012000	0.006000
80,000	0.008000	0.013000	0.008000
90,000	0.008000	0.015000	0.009000
100,000	0.010000	0.016000	0.010000
200,000	0.020000	0.034000	0.020000
300,000	0.031000	0.056000	0.031000
400,000	0.043000	0.071000	0.044000
500,000	0.055000	0.090000	0.054000

Radix Sort			
Datos	**Caso Mejor**	**Caso promedio**	**Caso peor**
0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.000000	0.000000	0.000000
800	0.000000	0.000000	0.000000
900	0.000000	0.000000	0.000000
2,000	0.000000	0.000000	0.000000
3,000	0.000000	0.000000	0.001000
4,000	0.001000	0.000000	0.001000
5,000	0.001000	0.000000	0.000000
6,000	0.001000	0.001000	0.000000
7,000	0.000000	0.000000	0.001000
8,000	0.000000	0.001000	0.001000
9,000	0.000000	0.001000	0.001000
10,000	0.001000	0.001000	0.001000
20,000	0.002000	0.002000	0.002000
30,000	0.003000	0.003000	0.003000
40,000	0.004000	0.004000	0.004000
50,000	0.005000	0.005000	0.005000
60,000	0.006000	0.006000	0.006000
70,000	0.007000	0.008000	0.008000

80,000	0.008000	0.008000	0.008000
90,000	0.009000	0.008000	0.009000
100,000	0.012000	0.010000	0.012000
200,000	0.023000	0.019000	0.024000
300,000	0.034000	0.029000	0.035000
400,000	0.047000	0.039000	0.047000
500,000	0.057000	0.048000	0.058000

Heap Sort			
Datos	**Caso Mejor**	**Caso promedio**	**Caso peor**
0	0	0	0
100	0.000000	0.000000	0.000000
200	0.000000	0.000000	0.000000
300	0.000000	0.000000	0.000000
400	0.000000	0.000000	0.000000
500	0.000000	0.000000	0.000000
600	0.000000	0.000000	0.000000
700	0.000000	0.000000	0.000000
800	0.000000	0.000000	0.000000
900	0.000000	0.000000	0.000000
2,000	0.000000	0.000000	0.000000
3,000	0.000000	0.001000	0.001000
4,000	0.001000	0.000000	0.000000
5,000	0.001000	0.001000	0.000000
6,000	0.001000	0.001000	0.001000
7,000	0.001000	0.001000	0.001000
8,000	0.001000	0.001000	0.001000
9,000	0.001000	0.002000	0.001000
10,000	0.002000	0.001000	0.002000
20,000	0.003000	0.003000	0.003000
30,000	0.005000	0.006000	0.005000
40,000	0.006000	0.008000	0.006000
50,000	0.008000	0.011000	0.007000
60,000	0.010000	0.001200	0.009000
70,000	0.011000	0.014000	0.011000
80,000	0.013000	0.018000	0.012000
90,000	0.015000	0.019000	0.015000
100,000	0.017000	0.023000	0.017000
200,000	0.360000	0.047000	0.036000

300,000	0.056000	0.074000	0.052000
400,000	0.075000	0.104000	0.072000
500,000	0.093000	0.013000	0.091000
600,000	0.114000	0.160000	0.111000
700,000	0.134000	0.188000	0.129000
800,000	0.154000	0.217000	0.149000
900,000	0.175000	0.249000	0.170000
1,000,000	0.195000	0.282000	0.193000
2,000,000	0.408000	0.614000	0.396000
3,000,000	0.623000	1.001000	0.613000
4,000,000	0.849000	1.419000	0.831000
5,000,000	1.078000	1.855000	1.055000
6,000,000	1.305000	2.321000	1.286000
7,000,000	1.529000	2.815000	1.508000
8,000,000	1.755000	3.280000	1.733000
9,000,000	1.997000	3.802000	1.966000
10,000,000	2.233000	4.335000	2.198000