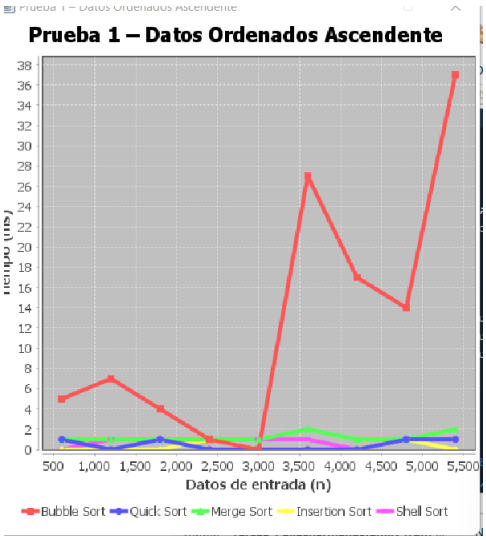
Facultad de ciencias.
Manejo de datos
Grupo 9160

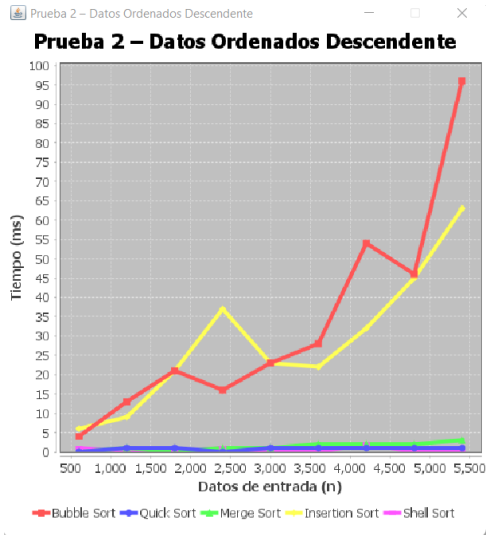
Equipo:
Pollos hermanos family
Integrantes:
Herrera León Jorge Alberto
Limón Cruz Andrés
Juan Mendoza Ilse Montserrat
Fouilloux Ramírez Jesús Gael
Juárez Pacheco Carlos Fernando

Imágenes de las graficas

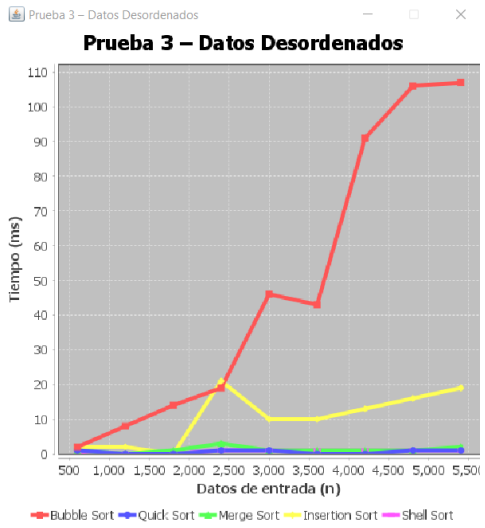
1



2



3



1.Preguntas

1.1.¿Cuál es el mejor y el peor algoritmo para datos ordenados de forma ascendente?

R=Insertion Sort es el mejor algoritmo para datos ordenados de forma ascendente y Bubble Sort es el peor algoritmo para datos ordenados de forma ascendente.

1.2.¿Cuál es el mejor y el peor algoritmo para datos ordenados de forma descendente?

R=Shell Sort es el mejor algoritmo para datos ordenados de forma descendente e Insertion Sort es el peor algoritmo para datos ordenados de forma descendente.

1.3.¿Cuál es el mejor y el peor algoritmo para datos desordenados?

R=Bubble Sort es el peor algoritmo para datos desordenados y shell sort

2.Funcionamiento, implementación (en Java para valores numéricos) y complejidad (mejor caso, caso promedio y peor caso) de los algoritmos Radix Sort y Heap Sort. ¿Alguno de ellos resulta mejor que los algoritmos revisados en el curso?

Radix Sort

El método de ordenamiento Radix Sort también llamado ordenamiento por residuos 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).

Funcionamiento:

El algoritmo ordena utilizando un algoritmo de ordenación estable, las letras o dígitos de forma individual, partiendo desde el que está más a la derecha (menos significativo) y hasta el que se encuentra más a la izquierda (el más significativo). Nota: a cada letra o dígito se le asigna una llave o código representado por un número entero, el cual se utiliza para el ordenamiento de cada elemento que conforma el valor original.

Ventajas:

-Radix Sort es estable, preservando la orden de elementos iguales.

-Radix Sort funciona en un tiempo lineal, en comparación de varios otros métodos de ordenamiento.

-El tiempo de ordenar cada elemento es constante, ya que no se hacen comparaciones entre elementos.

-Radix Sort es particularmente eficiente cuando se tratan con grandes grupos de números cortos.

Desventajas

-Radix Sort no funciona tan bien cuando los números son muy largos, ya que el total de tiempo es proporcional a la longitud del número más grande y al número de elementos a ordenar.

Complejidad:

-Para algún caso promedio su complejidad temporal es de **O(n + b)** dónde **b** es el rango de entrada, pero sí hay **d** dígitos en el elemento máximo "max", entonces la complejidad se vuelve **O(d*(n + b))**.
-Para el peor de los casos la complejidad es del orden de **[Big O]: O(n)**.
-Para el mejor de los casos la complejidad es del orden de **[Big Omega]: O(n)**. Que termina siendo la misma complejidad que en el peor de los casos.

Implementación:

```
// Implementación de radix en java
import java.io.*;
import java.util.*;

class Radix {

    // Una función de utilidad para obtener el valor máximo en arr[]
    static int getMax(int arr[], int n)
    {
      int mx = arr[0];
      for (int i = 1; i < n; i++)
          if (arr[i] > mx)
              mx = arr[i];
      return mx;
  }

  // Una función para hacer un conteo sort de arr[] según el dígito representado por exp.
  static void countSort(int arr[], int n, int exp)
  {
      int output[] = new int[n]; // matriz de salida
      int i;
      int count[] = new int[10];
      Arrays.fill(count, 0);

      // Almacenar el recuento de ocurrencias en count[]
      for (i = 0; i < n; i++)
          count[(arr[i] / exp) % 10]++;

      // Cambiar count[i] para que count[i] ahora contenga la posición actual de este dígito en output[]
      for (i = 1; i < 10; i++)
          count[i] += count[i - 1];

      // Construir la matriz de salida
      for (i = n - 1; i >= 0; i--) {
          output[count[(arr[i] / exp) % 10] - 1] = arr[i];
          count[(arr[i] / exp) % 10]--;
      }
```

```
        // Copiar la matriz de salida en arr[], de modo que arr[] ahora contiene números ordenados según el dígito actual.
        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // La función principal que ordena arr[] de tamaño n usando Radix sort
    static void radixsort(int arr[], int n)
    {
        // Encontrar el número máximo para saber el número de dígitos
        int m = getMax(arr, n);

        // Hacer un conteo sort para cada dígito. Hay que considerar que, en lugar de saltar el número de dígito, exp es el saltado.
        // exp es 10^i donde i es el número de dígito actual.
        for (int exp = 1; m / exp > 0; exp *= 10)
            countSort(arr, n, exp);
    }

    // Una función de utilidad para imprimir una matriz.
    static void print(int arr[], int n)
    {
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }

    /*Código de controlador*/
    public static void main(String[] args)
    {
        int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
        int n = arr.length;

         // Llamada de la función
        radixsort(arr, n);
        print(arr, n);
    }
}
```

Heap Sort

Para el método de Ordenamiento HeapSort, se toman las mejores características de los dos algoritmos de ordenamiento basados en comparación (MergeSort e InsertionSort) para crear un nuevo algoritmo llamado HeapSort.

Para poder realizar un ordenamiento mediante HeapSort, es necesario primero generar primero la estructura Heap(montículo).

Funcionamiento:

El algoritmo HeapSort, consiste en remover el mayor elemento que es siempre la raíz del Heap, una vez seleccionado el máximo, lo intercambiamos con el último elemento del vector, reducimos la cantidad de elementos del Heap y nos encargamos de volver a acomodarlo para que vuelva a ser un Heap.

Ventajas:

-Funciona efectivamente con datos desordenados.
-Su desempeño es en promedio tan bueno como el Quicksort.
-No utiliza memoria adicional.

Desventajas:

-No es estable, ya que se comporta de manera ineficaz con datos del mismo valor.
-Este método es mucho más complejo que los demás.

Complejidad:

-Para algún caso promedio la altura de un árbol binario completo con **n** elementos está en max **logn**. Por lo tanto, la función **heapify()** puede tener un máximo de **logn** comparaciones cuando un elemento se mueve de la raíz a la hoja. La función de construcción de heap se llama para **n/2** elementos, lo que hace que la complejidad de tiempo total para la primera etapa sea **n/2*logn** o **T(n) = nlogn**. **Heapsort()** toma el peor tiempo de **logn** para cada elemento, y **n** elementos hacen que su complejidad de tiempo también sea **nlogn**. Tanto el tiempo de complejidad para la construcción de heap y el de heapsort son agregadas y nos dan como resultado la complejidad cómo **nlogn**. Por lo tanto, la complejidad temporal total es del orden de **[Big Theta]: O(nlogn)**.
-Para el peor de los casos la complejidad temporal es de **[Big O]: O(nlogn)**.
-Para el mejor de los casos la complejidad temporal es de [**Big Omega]: O(nlogn)**. Es lo mismo que la complejidad temporal del peor de los casos.

Implementación

```
for (int i = 0; i < arr.length; i++) {
    // índice del elemento hijo
    int index = i;
    while (index != 0) {
        int parent = (index - 1) / 2;
        if (arr[index] <= arr[parent]) break;
        // Intercambiar el elemento hijo con su respectivo padre
        int temp = arr[index];
        arr[index] = arr[parent];
```

```
                arr[parent] = temp;
                index = parent;
            }
        }
    }

    // Para eliminar item de la parte superior del árbol binario -> arr[0]
    public void removeTopItem(int count) {
        int a = arr[0];
        arr[0] = arr[count];
        arr[count] = a;
        int index = 0;
        count--;
        // Para rehacer el árbol binario
        while (true) {
            int leftChild = index * 2 + 1;
            int rightChild = index * 2 + 2;

            // Checar el perímetro
            if (rightChild > count) break;
            if (arr[index] > arr[leftChild] && arr[index] > arr[rightChild]) break;

            // Para obtener un padre más grande
            int parentGreat = arr[rightChild] > arr[leftChild] ? rightChild : leftChild;

            // Intercambiar el item actual con su respectivo padre
            int temp = arr[index];
            arr[index] = arr[parentGreat];
            arr[parentGreat] = temp;
            index = parentGreat;
        }
    }

    // Ordenar usando heap
    public int[] heapSort() {
        // Hacer un heap
        makeHeap();
// Ordenación
        for (int i = arr.length - 1; i >= 0; i--) {
            removeTopItem(i);
        }
        return arr;
    }
}
```

¿Alguno de ellos resulta mejor que los algoritmos revisados en el curso?
-Si tenemos log 2 n bits para cada dígito, el tiempo de ejecución de Radix resulta ser mejor que el de Quick Sort para una amplia gama de números de entrada. Los factores constantes ocultos en la notación asintótica son más altos para Radix Sort y Quick-Sort usa cachés de hardware de manera más efectiva. Además, la clasificación de Radix utiliza la clasificación de conteo como una subrutina y la clasificación de conteo ocupa espacio adicional para clasificar los números.
-Heapsort es conveniente cuando se trata de ordenar arreglos estáticos grandes a diferencia de otros métodos como el Quicksort y el Mergesort. Heapsort compite primariamente con Quicksort, otro método de ordenamiento muy eficiente para propósitos en general.