Examen parcial 1 de Análisis y Diseño de Algoritmos. Fecha: 23 de Septiembre de 2017.

Nombre del alumno: **David Alfonso Velasco Sedano**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

I.- ¿Cuál es el método de ordenamiento que mejor cumple con cada característica citada? Fundamenta tu respuesta de manera breve y clara. *Valor: 30 puntos. (5 puntos por pregunta)*

1. Es de fácil programación y adecuado cuando los movimientos son más costosos que las comparaciones. Funciona con cualquier tipo de dato comparable, no sólo con enteros.

**Nombre del algoritmo.**

Heap sort

Fundamentación.

Esté algoritmo consume mucho tiempo a la hora de acomodar un árbol. La comparación simplemente se basa en ver si un elemento es menor o mayor al elemento actual. Mientras que, a la hora de mover un punto intermedio, debemos acomodar todo el árbol en base al nuevo elemento.

Siguiendo con las condiciones, el heap sort también se puede usar para acomodar palabras (hacer árboles de caracteres), cómo un ejemplo adicional que no está limitado simplemente a números.

1. Basa su eficiencia en que la distancia entre los elementos a intercambiar crece de manera exponencial conforme se alejan del elemento 0 de la lista.

**Nombre del algoritmo.**

Insertion sort

Fundamentación.

Dado que su segunda anidación de comparación recorre desde el punto actual hasta la posición inicial (0). Entre mayor sea el número de elementos a ordenar, mayor va ser el recorrido que constantemente tendrá que atravesar.

1. Se cumple el predicado *isSorted(array, 0, k)* en su *k-*ésima iteración, donde la función *isSorted(array, left, right)* devuelve verdadero si el sub-arreglo que va de *left* a *right* está ordenado.

**Nombre del algoritmo.**

Binary tree sort

Fundamentación.

Dado que asumimos que el árbol va estar balanceado, si comprobamos que un sector lo está, podemos asumir que el resto del árbol lo va estar.

1. Termina de procesar los espacios de búsqueda pequeños antes que los grandes.

**Nombre del algoritmo.**

Merge sort

Fundamentación.

La manera en que funciona esté algoritmo sub-dividiendo el arreglo hasta llegar a arreglos de tamaño 1 ó 2. Con ese arreglo pequeño, hace la validación y realiza los movimientos necesarios. Una vez terminado, combina los sub-arreglos pequeños a su estado anterior para realizar una comparación “correctiva”. Va así hasta regresar a su tamaño original.

1. Ofrece la máxima eficiencia para ordenar enteros positivos, pero inadecuado para cadenas de texto.

**Nombre del algoritmo.**

Merge Sort

Fundamentación.

1. Es una opción eficiente cuando cada elemento a ordenar puede ser partido en M unidades tal que M es mucho menor que el número de elementos.

**Nombre del algoritmo.**

Quick Sort

Fundamentación.

Esté algoritmo funciona en base a generar particiones pequeñas que fácilmente se van ordenando y comparando. Las particiones son menores al número inicial de elementos.

II.- Considere el siguiente algoritmo. *Valor: 5 + 10 + 10 puntos*.

**public** **static** **int**[] processingData(**int**[] array) {

**for** (**int** i = 0; i < array.length/2; i++) {

**boolean** swapped = **false**;

**for** (**int** j = i; j < array.length - i - 1; j++) {

**if** (array[j] < array[j+1])

Utils.*swap*(array, j, j+1);

}

**for** (**int** j = array.length - 2 - i; j > i; j--) {

**if** (array[j] > array[j-1]) {

Utils.*swap*(array, j, j-1);

swapped = **true**;

}

}

**if**(!swapped) **break**;

}

**return** array;

}

1. En términos generales, qué hace el algoritmo y cómo lo hace.

Acomoda de mayor a menor los elementos de un arreglo mediante fuerza bruta, ya que hace una comparación con todos los elementos.

1. Mediante un análisis *a priori*, determine el número comparaciones y swaps que ejecuta para un arreglo de tamaño N. Para el mejor y peor caso.

|  |  |  |
| --- | --- | --- |
|  | Comparaciones | Swaps |
| Mejor | 2N - 2 | 0 |
| Peor | (2N-2)^(N/2) | (2N-2)^(N/2) |

1. Demuestre que la complejidad en el mejor caso es mejor que cuadrática mediante una contradicción.

Nuestra formula es 2N-2. Vamos a tratar de comprobar que 2N-2 es menor o igual al orden de O(N^2). Por lo que al negarlo vamos a asumir que 2N-2 debe ser mayor a N^2 (2N-2 > N^2).

|  |
| --- |
| g(N) > O(f(N)) |
| g(N) = 2N-2 |
| f(N) = N^2 |
| 2N-2 > K(N^2) |
| (2N-2) / (N^2) > K |
| N = 3 |
| K < (2(3)-2) / (3^2) |
| K < (4/9) |
| K = 3/9 |
| 2(3)-2 > (3/9)((3)^2) |
| 4 > 3 |

Dado que 4 no es mayor a 3, comprobamos que 2N-2 debe ser menor o igual a N^2. Haciendo el mejor caso mejor a una solución cuadrática.

III.- Diseñe un algoritmo del tipo *divide y vencerás* de ordenamiento basado en el siguiente procedimiento:

* + Si el que el inicial intercambiarlos. Valor 10+ 10+ 5
  + Si hay 3 o más elementos valor al final es más pequeño en la lista entonces:
    1. Llamada recursiva con los primeros 2/3 de la lista
    2. Llamada recursiva con los últimos 2/3 de la lista
    3. Otra vez llamada recursiva con los primeros 2/3 de la lista.

**public** **static** **void** prob3ADA(**int**[] array, **int** start, **int** end) {

**if**(array[start] > array[end])

Utils.*swap*(array, start, end);

**if**((end - start) > 1) {

**int** index = ((end-start) \* 2) / 3;

*prob3ADA*(array, start, start + index);

*prob3ADA*(array, end - index, end);

*prob3ADA*(array, start, start + index);

}

}

1. Muestra la gráfica comparativa de este algoritmo con el merge sort, (en una sola gráfica)
2. ¿Cuál es la complejidad obtenida a posteriori?

|  |  |
| --- | --- |
|  | Formula |
| Comparaciones | N^2 |

IV- Codifique un método **boolean** isHeap(**int**[] array, boolean max) que determine si un arreglo es un montículo. El segundo parámetro “max” si es verdadero indica que el padre debe ser mayor que los hijos, si es falso significa que el padre debe ser menor que sus hijos. *Valor: 10 puntos*.

public static boolean isHeap(int[] array, boolean max) {

int rightChild = 0, leftChild = 0;

for(int i = 0; i < array.length; i++) {

leftChild = i \* 2 + 1;

rightChild = i \* 2 + 2;

if(leftChild >= array.length || rightChild >= array.length)

return true;

if(max) {

if(array[i] < array[leftChild] ||

array[i] < array[rightChild])

return false;

} else {

if(array[i] > array[leftChild] ||

array[i] > array[rightChild])

return false;

}

}

return true;

}

V.- Implemente un método recursivo int **search(Node234 n, int value)** que busque un número en un árbol 2,3,4. Si encuentra el valor regresar el índice o -1 en caso de no encontrarlo. Basarse en el código visto en clase. *Valor: 15 puntos.*

public static int search(Node234 node, int value) {

for(int i = 0; i < node.getType()-1; i++) {

if(node.getValue(i) == value ) {

return node.getIndex(0);

}

}

if(node.isLeaf())

return -1;

if(node.getType() == 2) {

if(value < node.getValue(0)) {

return search(node.getChild(0), value);

} else if(node.children.size() > 1) {

return search(node.getChild(1), value);

}

} else {

if(value < node.getValue(0)) {

return search(node.getChild(0), value);

} else if(node.children.size() > 1 && value > node.getValue(0)

&& value < node.getValue(1)) {

return search(node.getChild(1), value);

} else if(node.children.size() > 2 && value > node.getValue(1)) {

return search(node.getChild(2), value);

}

}

return -1;

}

VI.- El siguiente método recursivo construye un fractal como el que se muestra en la figura. Diseñe un método iterativo equivalente (con el mismo nombre y declaración). *Valor: 25 puntos.*

**public** **class** Dibujo {

**public** **static** **void** dibujar(Graphics g, **int** x1, **int** y1, **double** angle, **int** depth) {

**if** (depth == 0)

**return**;

**int** x2 = x1 + (**int**) (Math.cos(Math.toRadians(angle)) \* depth \* 5.0);

**int** y2 = y1 + (**int**) (Math.sin(Math.toRadians(angle)) \* depth \* 5.0);

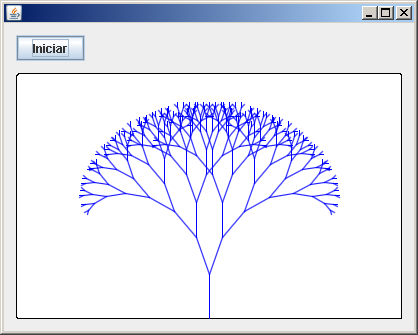
g2.drawLine(x1, y1, x2, y2);

dibujar(g2, x2, y2, angle - 20, depth - 1);

dibujar(g2, x2, y2, angle + 20, depth - 1);

}

}

 Dibujo.dibujar( jPanel1.getGraphics(),800/2, 600, -90, 9);

Usar el estilo del código fuente mostrado, o copiar + pegar de Eclipse rescatando los colores originales.

public static void dibujar(Graphics g2, int x1, int y1, double angle, int depth) {

    LinkedList<Integer> myList = new LinkedList<Integer>();

    LinkedList<Double> angleList = new LinkedList<Double>();

    myList.add(x1);

    myList.add(y1);

    myList.add(depth);

    angleList.add(angle);

    while(!myList.isEmpty()) {

    x1 = myList.pop();

    y1 = myList.pop();

    depth = myList.pop();

    angle = angleList.pop();

    if(depth == 0)

    continue;

    int x2 = x1 + (int) (Math.cos(Math.toRadians(angle)) \* depth \* 5.0);

        int y2 = y1 + (int) (Math.sin(Math.toRadians(angle)) \* depth \* 5.0);

        g2.drawLine(x1, y1, x2, y2);

        myList.add(x2);

        myList.add(y2);

        myList.add(depth-1);

        angleList.add(angle + 20);

        myList.add(x2);

        myList.add(y2);

        myList.add(depth-1);

        angleList.add(angle - 20);

    }

    }