Examen parcial 1 de Análisis y Diseño de Algoritmos. Fecha: Viernes, 25 de Septiembre de 2015.
Nombre del alumno: Angel de Jesus Bañuelos Sahagun Aciertos: _____

I.- ¿Cuál es el método de ordenamiento que mejor cumple con cada característica citada? Fundamenta tu respuesta de manera breve. Te puedes apoyar con una porción del algoritmo que ilustre tu elección. *Valor: 40 puntos.*

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. Selección.
   Fundamentación. La implementación de este algoritmo es muy sencilla, y funciona prácticamente para cualquier tipo de dato comparable, Este algoritmo en el mejor de sus casos solo hace comparaciones y ningún movimiento. Y en el peor de los casos realiza ($1.5N\epsilon\ O(n^2)$)

2. 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. Burbuja.
   Fundamentación. Ese elemento lleva a la derecha los elementos mayores y el peor de los casos es cuando el arreglo está invertido que es cuadrático pero **no exponencial**, este es el algoritmo más lento de ordenamiento que vimos.

3. Se cumple el predicado *isSorted(array, 0, k)* en su *k*-ésima iteración, tal que *isSorted(array, left, right)* devuelve verdadero si el sub-arreglo que va de *left* a *right* está ordenado.
   Nombre del algoritmo. Quicksort.
   Fundamentación. Se basa en particionar el arreglo en dos, selecciona un pivote y lo posiciona donde corresponde ordenando la parte izquierda y después la derecha. No es muy eficiente cuando se hace una mala elección del pivote, pero generalmente cuando el arreglo esta invertido u casi-ordenado.

4. Termina de procesar los espacios de búsqueda pequeños antes que los grandes.
   Nombre del algoritmo. MergeSort
   Fundamentación. Porque va realizando divisiones sucesivas del espacio de búsqueda hasta llegar a un arreglo trivial, que por definición ya está ordenado, va sucesivamente hace arriba. Teniendo cada vez arreglos ordenados.

5. Se destaca porque podemos saber de antemano si el arreglo ya quedó ordenado.
   Nombre del algoritmo. Burbuja
   Fundamentación. Va buscando el elemento mayor, con su adyacente si en la primera pasada no realizo ningún intercambio termina la ejecución, para poder cumplir esta propiedad se tiene que agregar una bandera en el ciclo interno para concluir la ejecución.

6. Basa su funcionamiento en colocar un elemento en su posición definitiva y tratar de manera separada los elementos que lo rodean.
   Nombre del algoritmo. Heap Sort
   Fundamentación. Su funcionalidad de base en colocar el elemento mas grande siempre al principio creando un montículo, y al final intercambiar el elemento del elemento 0 al elemento n-1, bloquea la posición y trabaja con los demás sin afectar al que ya dejo bloqueado.

7. Ofrece la máxima eficiencia para ordenar enteros positivos, pero inadecuado para cadenas de texto.
   Nombre del algoritmo. Ordenamiento Por Conteo.
   Fundamentación. Funciona cuando las claves son números enteros. En el caso de cadenas de texto se tendrían que mapear los valores a índices del arreglo y se tendría que calcular corrimiento, lo que disminuiría considerablemente el desempeño.

8. Es una opción eficiente cuando cada elemento puede ser partido en M unidades tal que M es mucho menor que el número de elementos.
   Nombre del algoritmo. Radix Sort

Fundamentación. Porque basa su funcionamiento en partir cada elemento de la lista en componentes atómicos tomando por unidades, decenas, etc… lo que va ocasionando que cada que se mueva a listas, estas listas estén parcialmente ordenadas.

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

```
void go(int array[]) {
   int ms = 0, mi = -1;
   for(int i = 1; i <= array.length; i *= 2) {
      int s = 0;
      for(int b = i - 1; b < array.length; b += i) {
         s += array[b];
      }
      if(mi < 0 || s > ms) {
         ms = s;
         mi = i;
      }
   }
   printf("%d, %d\n", mi, ms);
}
```

1) En términos generales, ¿qué imprime?
2) Mediante un análisis *a priori*, determine el número de sumas que ejecuta para un arreglo de tamaño N. Por simplicidad, suponga que el último valor de *i* será exactamente N.
3) Demuestre que la complejidad de este algoritmo es mejor que la quasi-lineal mediante una contradicción.

1) El algoritmo realiza la sumatoria de los diferentes posiciones del arreglo, buscando el elementos mayores a la suma y guardando el valor, así como guardando la posición del arreglo que genero la sumatoria. Tomando incrementos de dos. Y realizando la suma con un elemento menor.

2) El algoritmo se ejecuta $\frac{n}{2} + 1 + \frac{n}{4} + 1 + \frac{n}{8} + 1 + \cdots \frac{n}{k} + 1$ Por lo tanto, la complejidad es: $LogN \in O(logN)$

3) Demostrar: $LogN \in O(logN)$

$$g(N) \leq c0\, f(N)\ para\ todo\ \ N > N0$$

g(n) = log2N ≤ O(log2N) para todo N > n0
N=16, n0 = 15
C0 = 10
Sustituyendo.
Log2(17) >10 log2(17)
Desarrollando tenemos:
3.9 > 39 por lo que concluimos que por contradicción cumple.

III.- El siguiente algoritmo calcula la máxima suma de los datos de algún sub-arreglo de un arreglo que contiene números enteros (positivos, negativos, cero). Sin embargo, conforme el tamaño arreglo crece, esta solución comienza a ser inviable por su lentitud. *Valor: 25 puntos.*

```
int sum(int[] array, int start, int end) {
   int sum = 0;
   for(int i = start; i <= end; i ++) sum += array[i];
   return sum;
}

int maxSum(int[] array) {
   int maxSum = 0;
   for(int i = 0; i < array.length - 1; i ++) {
      for(int j = i + 1; j < array.length; j ++) {
         int s = sum(array, i, j);
         if(s > maxSum) maxSum = s;
      }
   }
   return maxSum;
}
```

Diseñe un algoritmo del tipo *divide y vencerás* que solucione el problema.

a. Puede comprobar su precisión comparando el resultado obtenido con el del algoritmo mostrado.
b. El algoritmo dividirá siempre el espacio de búsqueda en dos mitades hasta llegar a la trivialidad.
c. Calcula la suma obtenida en la primera mitad.
d. Calcula la suma obtenida en la segunda mitad.
e. Calcula la suma obtenida de un sub-arreglo que ocupe las dos mitades:
    i. Encuentra la suma máxima del sub-arreglo que termina en la primera mitad.
    ii. Encuentra la suma máxima del sub-arreglo que comienza en la segunda mitad.
    iii. Suma los dos resultados anteriores.
f. Devuelve la mayor de las tres sumas.

R. Para ejecutar el código llamar al método `maxSumRec(int[] array)`

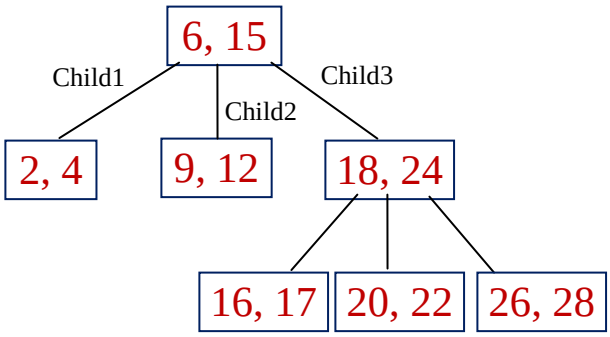
```
private static int maxSumRec(int[] a, int min, int max) {
        int maxSumLeft = 0;
        int maxSumRight = 0;
        int sum = 0;
        int maxSum = 0;
        int middle = (min + max) / 2;
        if (min == max) {
            return a[min] > 0 ? a[min] : 0;
        }
        //c. Calcula la suma obtenida en la primera mitad
        int sumLeft = maxSumRec(a, min, middle);
        //d. Calcula la suma obtenida en la segunda mitad.
        int sumRight = maxSumRec(a, middle + 1, max);

        //i. Encuentra la suma máxima del sub-arreglo que termina en la primera mitad.
        for (int i = middle; i >= min; i--) {
            sum += a[i];
            if (sum > maxSumLeft) {
                maxSumLeft = sum;
            }
        }
        //ii.       Encuentra la suma máxima del sub-arreglo que comienza en la segunda mitad.
        for (int j = middle + 1; j <= max; j++) {
            maxSum += a[j];
            if (maxSum > maxSumRight) {
```

```
            maxSumRight = maxSum;
        }
    }
    //f. Devuelve la mayor de las tres sumas
    return maxOfTree(sumLeft, sumRight, maxSumLeft + maxSumRight);

}

public static int maxSumRec(int a[]) {
    return maxSumRec(a, 0, a.length - 1);

}

private static int maxOfTree(int left, int right, int maxSum) {
    if (left < right && left < maxSum) {
        if (right < maxSum) {
            return maxSum;
        } else {
            return right;
        }
    } else if (right < left && right < maxSum) {
        if (left < maxSum) {
            return maxSum;
        } else {
            return left;
        }
    } else {
        if (left < right) {
            return right;
        } else {
            return left;
        }
    }
}
```

IV- Codifique un método **`boolean isHeap(int[] array)`** que determine si un arreglo es un montículo. *Valor: 10 puntos*.

```
/**
     * Cada nodo tiene un valor comparable tal que ningún nodo tiene un valor
     * más grande que el de su padre.
     *
     * Está balanceado: cada nodo tiene 2 hijos, excepto los de los últimos dos
     * niveles.
     *
     * Está alineado a la izquierda: si un nodo sólo tiene un hijo, debe ser el
     * izquierdo.
     *
     * @param array to validate if it is heap
     * @return
     */
    public static boolean isHeap(int[] array) {
        int parent = 0;
        int childL = 1, childR = 2;
        for (int i = 1; i < array.length; i++) {
            // Ningun nodo tiene un valor mas grande que el de su padre
            if (array[parent] < array[childL] || ((childR < array.length) && array[parent] <
array[childR])) {
                return false;
            }
            parent = i;
            childL = ((parent + 1) * 2) - 1;
            childR = childL + 1;
            if (childL > (array.length - 1)) {
                break;
            }
        }
        return true;
    }
```

V.- Implemente un método recursivo **`search(Node n, int value)`** que busque un número en un árbol 3-ario como el de la figura. Si lo encuentra deberá imprimir "Encontrado" y terminar. Si no lo encuentra, sólo terminar. Por simplicidad, suponga que el árbol está balanceado: todos los nodos tienen 3 hijos excepto las hojas. Además, todos los nodos guardan dos valores. OJO: La clase Node ya existe. *Valor: 15 puntos.*



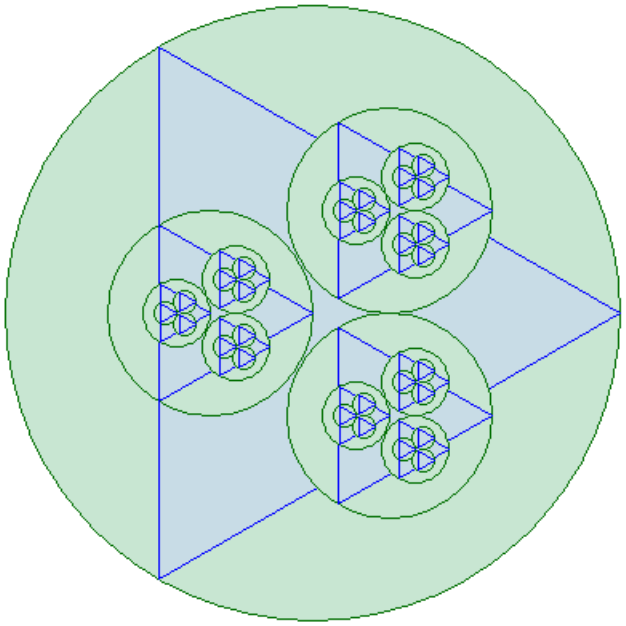
Node
+ child1: Node + child2: Node + child3: Node + value1: int + value2: int

Codigo en la siguiente hoja.

```java
/**
     *
     * @param node element to look into
     * @param value to search
     * @return
     */
    public String search(Node node, int value) {
        if (node == null) {
            return "";
        }
        if (node.getValues()[0] == value || node.getValues()[1] == value) {
            return "Encontrado";
        } else {
            if (value < node.getValues()[0]) {
                if (node.getChild1() != null) {
                    return search(node.getChild1(), value);
                }
            } else if (value > node.getValues()[0] && value < node.getValues()[1]) {
                if (node.getChild2() != null) {
                    return search(node.getChild2(), value);
                }
            } else {
                if (node.getChild3() != null) {
                    return search(node.getChild3(), value);
                }
            }
        }
        return "";
    }
```

VI.- El siguiente método recursivo construye un fractal como el que se muestra en la figura. Diseñe un método iterativo equivalente. *Valor: 25 puntos.*

```
// Recibe <x, y> del centro y radio del círculo
void drawFractal(int x, int y, int radius) {
  if(radius <= 4) return;
  drawCircle(x, y, radius);
  drawTriangle(x, y, radius);
  drawFractal(x + radius / 4, y - radius / 3,
              radius / 3); // Superior derecho
  drawFractal(x + radius / 4, y + radius / 3,
              radius / 3); // Inferior derecho
  drawFractal(x - radius / 3, y,
              radius / 3); // Izquierdo
}
drawFractal(400, 300, 192); // El de la figura
```



```
public void drawFractalIterative(int x, int y, int radius) {
    LinkedList<int[]> draws = new LinkedList();
    int[] d = {x, y, radius};
    draws.add(d);
    int level = 0;
    int newR = radius / 3;

    while (newR >= 4) {
        level++;
        int parent = 0;
        for (int h = 0; h < level - 1; h++) {
            parent += Math.pow(3, h);
        }
        for (int n = 1; n <= Math.pow(3, level - 1); n++) {
            int x1 = draws.get(parent)[0];
            int x2 = x1;
            int x3 = x1;
            int y1 = draws.get(parent)[1];
            int y2 = y1;
            int y3 = y1;

            int nRadius = draws.get(parent)[2];
            x1 = x1 + nRadius / 4;
            y1 = y1 - nRadius / 3;
            int[] a = {x1, y1, newR};
            draws.add(a);

            x2 = x2 + nRadius / 4;
            y2 = y2 + nRadius / 3;
            int[] b = {x2, y2, newR};
            draws.add(b);

            x3 = x3 - nRadius / 3;
            int[] c = {x3, y3, newR};
            draws.add(c);
            parent++;
        }
```

```
            newR = newR / 3;
        }

        while(!draws.isEmpty()){
            int[] poll = draws.pollLast();
            drawCircle(poll[0], poll[1], poll[2]);
            drawTriangle(poll[0], poll[1], poll[2]);
        }

    }
```