## **Evaluación asincrónica 1** de Estrategias Algorítmicas, ITESO, Luis Gatica

### Fecha de entrega: viernes 4 de marzo de 2022

Nombre: Rodríguez Castro, Carlos Eduardo; Cordero Hernández, Marco Ricardo

# Parte 1: análisis *a priori*

1. Indica qué clase de algoritmo es el algoritmo que resuelve cada uno de estos problemas. En caso de conocer soluciones sofisticadas a algunos de ellos, ignóralas. Justifica en uno o un par de enunciados tu respuesta para cada caso. Pon especial atención en la tarea a cumplir: que la solución sea ingenua no significa que otra “gran tarea” que no le fue solicitada (por ejemplo, construir o imprimir en vez de simplemente contar).
   1. Dado un arreglo de N números enteros, **cuenta** el número de impares positivos existentes en tal arreglo.

***N*** : Se deben recorrer todos los números del arreglo para determinar si Arr[pos] cumple con la condición dada. La instrucción que debe verificar es constante (*c < N*).

* 1. Dada una palabra de N caracteres, **determina** si termina en vocal o no.

***1*** : Únicamente se verifica el último carácter; se consideran mayúsculas y minúsculas en términos de equivalencia (*int* ⇔ *char*) y solamente se realiza esta comparación.

* 1. Dada una matriz de N x N números reales, **encuentra** el número más grande de la segunda columna.

***N*** : Se deben recorrer todos los números de la segunda columna de la matriz (Matrix[x][1]) contando con un valor inicial (Matrix[0][1]) y realizar comparaciones (constante) para ir reemplazando ese valor predefinido hasta que se acaben los valores, es decir, la única condicionante del algoritmo es la dimensión **N**.

* 1. Dado un arreglo de N números enteros, **imprime** todos los pares (𝑎, 𝑏), tales que 𝑎, 𝑏 ∈ 𝑎𝑟𝑟𝑒𝑔𝑙𝑜, 𝑎 es un número impar y 𝑏 es un número par.

***Mejor caso 🡪 N*** : El arreglo no contiene números impares, de tal manera que no se ingresa a un segundo ciclo y únicamente se realiza una comprobación.

***Peor caso 🡪 N2*** : El arreglo no contiene números pares, ingresando a un segundo ciclo pero no imprimiendo nada.

***Caso promedio 🡪 N2*** : El arreglo contiene tanto números pares como impares, imprimiendo los pares indicados.

* 1. **Calcula** el número de veces que hay que dividir un número entero N entre 2 hasta llegar a 1.

***log2(N)*** : dado que el problema considera un número que se irá dividiendo entre dos, esto es logarítmico.

* 1. Dadas dos matrices de números reales cada una, **calcula** .

***N2*** : Dado que los arreglos tienen una dimensión simétrica, se accede a ellos de manera directa mediante el uso de dos ciclos que deben recorrer su totalidad.

* 1. Dado un arreglo de números enteros, **calcula** el promedio de los primeros 50 elementos.

***1*** : Dado que siempre se considera un valor constante, la clasificación también lo es.

* 1. Dado un conjunto de *N* objetos, **construye** todos los subconjuntos diferentes de objetos, tal que .

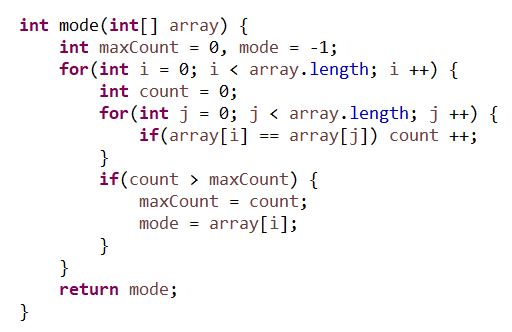
*2N*: dado que se analiza el conjunto potencia sin tomar en cuenta el subconjunto vacío.

* 1. Dado un conjunto de *N* objetos, **cuenta** el número de subconjuntos diferentes de objetos, tal que .

***1*** : ya que solo se está contando la cantidad de subconjuntos, solo se realiza una operación aritmética con el fin de determinar el resultado.

* 1. Dada una formación de *N* niños, **imprime** todos los distintos acomodos que pueden tener todos los niños en la formación.

***N!*** : dado que se analiza una permutación de N tomados N a la vez, se obtiene que la operación siempre será factorial.

1. Efectúa un análisis **a priori** del siguiente algoritmo que calcula la **moda** (el número con más ocurrencias) de un arreglo de enteros, y exprese el **tiempo de ejecución** en términos de los siguientes costos:
   1. *c*1: costo de una inicialización de variables, o de una comparación, movimiento o cualquier operación aritmética entre **índices** y **conteos**.
   2. *c*2: costo de un movimiento entre **datos**.
   3. *c*3: costo de una comparación entre **datos**.

|  |  |  |  |
| --- | --- | --- | --- |
| **Instrucción** | **Costo** | **Veces (iguales)** | **Veces (distintos)** |
| maxCount = 0 | C1 | 1 | 1 |
| mode = -1 | C1 | 1 | 1 |
| i = 0; | C1 | 1 | 1 |
| i < array.length | C3 | n + 1 | n + 1 |
| i ++ | C1 | n | n |
| count = 0; | C1 | n | n |
| j = 0; | C1 | n | n |
| j < array.length | C3 | n2 + n | n2 + n |
| j ++ | C1 | n2 | n2 |
| array[i] == array[j] | C3 | n2 | n2 |
| count ++; | C1 | n2 | n |
| count > maxCount | C3 | n | n |
| maxCount = count; | C2 | 1 | 1 |
| mode = array[i]; | C2 | 1 | 1 |

*Considere dos casos: cuando todos los elementos del arreglo son el mismo y cuando son todos distintos.*

*Tiguales(n)* =

*Tdistintos(n)* =

1. Expresa el tiempo de ejecución de este método en términos de la instrucción que más se repita. ¿Qué clase de algoritmo es?

void sumAndMult(int[] array) {

int sum = 0;

int product = 1;

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

for (int j = 0; j < 12; j++) { // Línea más repetida

array[i] \*= 1.0083;

}

}

}

1. Expresa el tiempo de ejecución de este método en términos del número de impresiones. ¿Qué clase de algoritmo es?

void printOrderedPairs(int[] array) {

for (int i = 0; i < array.length; i++) { // N + 1

for (int j = i + 1; j < array.length; j++) { // (N^2 - N / 2) - 1

System.*out*.println(array[i] + "," + array[j]); // (N^2 – N) / 2

}

}

}

1. Expresa el tiempo de ejecución de este método en términos del número de sumas. ¿Qué clase de algoritmo es?

int divide(int a, int b) {

int count = 0;

int sum = b;

while(sum <= a) {

sum += b;

count ++;

}

return count;

}

1. Dado un arreglo A *ordenado* y un número entero X, implementar una función en Java que determine si existen dos números en A que sumen exactamente X.

Por ejemplo, para A = {1, 4, 5, 7, 9, 12, 13},

Si X = 13, la función devuelve *true* porque 4 + 9 = 13.

Si X = 15, la función devuelve *false*.

La complejidad temporal esperada para este ejercicio es cuadrática.

    public static boolean **SumaEnteros1**(int[] *nums*, int *x*) {

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

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

                if (*nums*[i] + *nums*[j] == *x*) return true;

            }

        }

        return false;

    }

1. Implemente una segunda función que resuelva el problema anterior, pero con una complejidad temporal menor. Recuerde que constante < logarítmica < lineal < cuasilineal < cuadrática…

    public static boolean **SumaEnteros2**(int[] *nums*, int *x*) {

        int l = 0, r = *nums*.length - 1, sumHolder;

        while (l != r) {

            sumHolder = *nums*[l] + *nums*[r];

            if (sumHolder > *x*) r--;

            else if (sumHolder < *x*) l++;

            else return true;

        }

        return false;

    }

1. Esboce en un párrafo un análisis del algoritmo anterior que muestre que su complejidad temporal es menor que cuadrática.

Dado que se ha planteado el trabajar con punteros a extremos cambiantes del arreglo mediante el uso de la ventana deslizante, como peor caso en donde no se encuentre el entero objetivo, el arreglo se recorrerá una única vez y se irán verificando los pares de elementos simultáneamente. Dicho esto, al depender la ejecución únicamente del tamaño del arreglo, la complejidad de este algoritmo es ***lineal***, lo cual es menor que cuadrático.

1. Implemente una función que reciba una *n* entera positiva y devuelva una matriz de *n*\**n* enteros rellenada con ceros a excepción de las dos diagonales principales (rellenadas con unos). Por ejemplo, para *n* = 5, la matriz se vería así:

10001  
01010  
00100  
01010  
10001

    public static int[][] **fill**(int *n*) {

        int[][] matrix = new int[*n*][*n*];

        for (int i = 0; i < (*n* / 2) + 1; i++) {

            for (int j = 0; j < (*n* / 2) + 1; j++) {

                int curVal = (j == i) ? 1 : 0;

                matrix[i][j] = curVal;

                matrix[i][*n* - j - 1] = curVal;

                matrix[*n* - i - 1][j] = curVal;

                matrix[*n* - i - 1][*n* - j - 1] = curVal;

            }

        }

        return matrix;

    }

La esencia del algoritmo anterior funcionaría en lenguajes en donde el manejo de arreglos se vincule directamente con la memoria y demás componentes de este tipo de estructura de datos, sin embargo, al tener la posibilidad de trabajar con un lenguaje como Java, el cual rellena automáticamente los datos de un arreglo de enteros con 0’s, se puede tomar ventaja de esta característica para reducir la complejidad del algoritmo, de manera que quedaría de la siguiente manera.

    public static int[][] **fill**(int *n*) {

        int[][] matrix = new int[*n*][*n*];

        for (int i = 0; i < (*n* / 2) + 1; i++) {

            matrix[i][i] = 1;

            matrix[i][*n* - i - 1] = 1;

            matrix[*n* - i - 1][i] = 1;

            matrix[*n* - i - 1][*n* - i - 1] = 1;

        }

        return matrix;

    }

1. Escriba en un párrafo un análisis de la complejidad temporal de la función anterior. No considere el costo de construir la matriz usando *new int[][]*. Recuerde que inicialmente las posiciones no asignadas explícitamente contienen ceros.

Para el primer algoritmo propuesto, se crean dos ciclos para iterar a través de las filas y columnas de la matriz con el fin de rellenarlas, de manera que se opera a través de índices y no de el recorrido de la misma matriz. Dado que se puede ver la matriz como un conjunto de cuatro secciones que son simétricas, basta con recorrer una “esquina” de esta y espejearlo.

Ignorando la comparación entre índices y asignación a las posiciones de la matriz, se obtiene

es decir, tiempo cuadrático.

Para el segundo algoritmo propuesto, la complejidad se verá reducida, puesto que únicamente se necesita recorrer la mitad del tamaño del nuevo arreglo, rellenando solamente las posiciones en donde se requieran los 1’s. Se utiliza la misma lógica de la esquina y el espejeo.

De igual manera, ignorando las asignaciones, se obtiene

es decir, tiempo lineal.

|  |  |
| --- | --- |
| Criterio de evaluación | Puntos |
| Ejercicio 1. Clasificación y justificación correctas ( 2 pts. c/u de los 10) | 20 |
| Ejercicio 2. Análisis correcto y bien detallado para los dos casos | 25 |
| Ejercicio 3. Respuesta y justificación correctas | 5 |
| Ejercicio 4. Respuesta y justificación correctas | 5 |
| Ejercicio 5. Respuesta y justificación correctas | 5 |
| Ejercicio 6. Algoritmo e implementación correctos | 10 |
| Ejercicio 7. Algoritmo optimizado y bien implementado | 15 |
| Ejercicio 8. Análisis y justificación correctos | 5 |
| Ejercicio 9. Algoritmo correcto en función bien implementada | 5 |
| Ejercicio 10. Análisis y justificación correctos | 5 |
| TOTAL | **100** |

# Parte 2: análisis *a posteriori*

1. Modifica el análisis **a posteriori** del algoritmo ingenuo de suma de sumas para que en lugar de analizar un solo arreglo de tamaño *n* para cada *n* revisada, promedie el conteo para *n*/10 arreglos de tamaño *n* para cada *n* distinta. Por ejemplo, cuando *n=5000*, hay que generar *n/10=500* arreglos distintos y para cada uno contar el total de operaciones relevantes al ejecutar *suma de sumas* ingenuo. En lugar de desplegar cada conteo, se promediarán los 500 y se desplegará ese promedio, que será el dato tabulado y graficado en Excel. Cuando *n=10000*, se generarán *n/10=1000* arreglos distintos y se desplegará el promedio de esas 1000 cuentas, y así sucesivamente. Esto amplía la muestra estadística utilizada para el análisis.

**Para este ejercicio utilice límites e incrementos distintos de los originales: haga un análisis para cada *n* de 1000 a 5000 con incremento de 100.**

Código de clase (implementación y análisis a posteriori)

public class SumaDeSumasModificado {

    private static float sumCount;

    public static int **randomInt**(int *min*, int *max*) {

        return (int)(*min* + (*max* - *min*) \* Math.**random**());

    }

    public static int[] **randomArray**(int *n*, int *min*, int *max*) {

        int[] arr = new int[*n*];

        for (int i = 0; i < *n*; i++) {

            arr[i] = **randomInt**(*min*, *max*);

        }

        return arr;

    }

    public static int **sumaDeSumas**(int[] *arr*) {

        int sum = 0, sumParcial; sumCount = 0;

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

            sumParcial = 0;

            for (int j = 0; j <= i; j++) {

                sumParcial += *arr*[j];

                sumCount++;

            }

            sum += sumParcial;

        }

        return sum;

    }

    public static void **main**(String[] *args*) {

        for (int n = 1000; n <= 5000; n += 100) {

            float sumPromedio = 0;

            for (int i = 0; i < (n / 10); i++) {

**sumaDeSumas**(**randomArray**(n, -10, 10));

                sumPromedio += sumCount;

            }

            sumPromedio /= (n / 10);

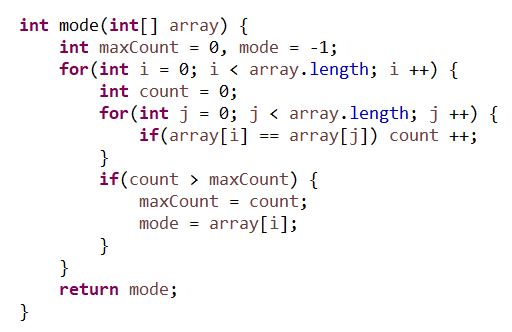
            System.out.**printf**("%d\t%.3f\n", n, sumPromedio);

        }

    }

}

1. Efectúa un análisis **a posteriori** del siguiente algoritmo que calcula la **moda** (el número con más ocurrencias) de un arreglo de enteros, y exprese el **tiempo de ejecución** contando las siguientes operaciones:



* 1. *movimientos* entre **datos** (c2 en la tarea 1)
  2. *comparaciones* entre **datos** (c3 en la tarea 1)

*Considera tres casos: cuando todos los elementos del arreglo son el mismo, cuando son todos distintos y cuando son aleatorios\*.*

*Tiguales(n)*

*Tdistintos(n)*

*Taleatorios\*(n)*

*¿Coincidió el análisis a posteriori de los primeros dos casos con el análisis a priori?¿Por qué?*

Para ambos casos en el análisis a posterior, la ecuación resultante fue , la cual, despreciando las constante con valor sumamente pequeño y los órdenes inferiores, se obtiene , lo cual coincide con el análisis a priori.

Adicional a que el análisis a posteriori se realizó correctamente, dado que se puede replicar este simulación con facilidad y exactitud, los resultados se pueden comparar y compaginar de manera que la teoría analítica cumple con la práctica.

Ecuación para elementos iguales:

Ecuación para elementos distintos:

Ecuación para elementos aleatorios:

Código de clase (implementación y análisis a posteriori)

public class moda {

    static double comparaciones, movimientos;

    static int **mode**(int[] *array*) {

        int maxCount = 0, mode = -1;

        comparaciones = 0; movimientos = 0;

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

            int count = 0;

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

                comparaciones++;

                if (*array*[i] == *array*[j]) count++; *// c\_3*

            }

            comparaciones++;

            if (count > maxCount) { *// c\_3*

                maxCount = count; *// c\_2*

                mode = *array*[i]; *// c\_2*

                movimientos += 2;

            }

        }

        return mode;

    }

    static int[] **fillArr**(int *n*, int *val*) {

        int[] arr = new int[*n*];

        for (int i = 0; i < *n*; i++) {

            arr[i] = *val*;

        }

        return arr;

    }

    static int **randomInt**(int *min*, int *max*) {

        return (int)(*min* + (*max* - *min*) \* Math.**random**());

    }

    static int[] **randomArray**(int *n*, int *min*, int *max*) {

        int[] arr = new int[*n*];

        for (int i = 0; i < *n*; i++) {

            arr[i] = **randomInt**(*min*, *max*);

        }

        return arr;

    }

    static int[] **generateRangeArr**(int *limInf*, int *limSup*) { *// límites inclusivos [limInf, limSup]*

        int[] arr = new int[*limSup* + 1 - *limInf*];

        for (int i = *limInf*; i <= *limSup*; i++) {

            arr[i - *limInf*] = i;

        }

        return arr;

    }

    public static void **main**(String[] *args*) {

        for (int n = 1000; n <= 5000; n += 100) {

            float promCompAleatorio = 0, promMovAleatorio = 0;

**mode**(**fillArr**(n, n));

            double movIgual = movimientos;

            double compIgual = comparaciones;

**mode**(**generateRangeArr**(1, n));

            double movDistinto = movimientos;

            double compDistinto = comparaciones;

            for (int i = 0; i < (n / 10); i++) {

**mode**(**randomArray**(n, -n, n));

                promCompAleatorio += comparaciones;

                promMovAleatorio += movimientos;

            }

            promCompAleatorio /= (n / 10);

            promMovAleatorio /= (n / 10);

            System.out.**printf**("%d\t%f\t%f\t%f\t%f\t%.3f\t%.3f\n",

            n,compIgual, movIgual, compDistinto, movDistinto,

            promCompAleatorio, promMovAleatorio);

*// n, compIgual, movIgual, compDistinto, movDistinto, promCompAleatorio, promMovAleatorio*

        }

    }

}

1. Codifica el algoritmo de inserción en Java. Efectúa un análisis **a posteriori** del mismo considerando por separado *movimientos* y *comparaciones* entre datos (como en el ejercicio 2).

*Considera tres casos: cuando el arreglo está ordenado no descendentemente (1,2,3,4,5,6,7,… pero también 1,2,3,3,4,5,5,6,…), cuando el arreglo está ordenado en reversa (descendentemente: n, n-1, n-2, n-3, …, 3, 2, 1) y cuando son aleatorios\*.*

*Tno descendentemente(n)*

*Tdescendentemente(n)*

*Taleatorios\*(n)*

*¿Coincidió el análisis a posteriori de los primeros dos casos con el análisis a priori iniciado en clase?*

El análisis a priori realizado anteriormente dictaminó que la complejidad de este algoritmo estaría resultando en una complejidad cuadrática. Para los casos en donde el arreglo se encontraba en orden descendente o desorden, el análisis a posteriori determinó que el mejor ajuste para los datos generados resultaría en una expresión cuadrática, es decir, los resultados son congruentes.

Ecuación para arreglos ordenados:

Ecuación para arreglos con orden inverso:

Ecuación para arreglos aleatorios:

Código de clase (implementación y análisis a posteriori)

public class InsertionSort {

    static double comparaciones, movimientos;

    public static int **randomInt**(int *min*, int *max*) {

        return (int)(*min* + (*max* - *min*) \* Math.**random**());

    }

    public static int[] **randomArray**(int *n*, int *min*, int *max*) {

        int[] arr = new int[*n*];

        for (int i = 0; i < *n*; i++) {

            arr[i] = **randomInt**(*min*, *max*);

        }

        return arr;

    }

    static int[] **generateRangeArr**(int *limInf*, int *limSup*, boolean *reverse*) { *// límites inclusivos [limInf, limSup]*

        int[] arr = new int[*limSup* + 1 - *limInf*];

        for (int i = *limInf*; i <= *limSup*; i++) {

            arr[i - *limInf*] = (*reverse*) ? *limSup* - i + 1 : i;

        }

        return arr;

    }

    public static void **sort**(int[] *arr*) {

        comparaciones = 0; movimientos = 0;

        for (int p = 1; p < *arr*.length; p++) {

            int pivot = *arr*[p]; *// c\_2*

            movimientos++;

            int i = p - 1;

            while (i >= 0 && *arr*[i] > pivot) { *// c\_2*

                comparaciones++;

*arr*[i + 1] = *arr*[i]; *// c\_3*

                movimientos++;

                i--;

            } comparaciones++;

*arr*[i + 1] = pivot; *// c\_3*

            movimientos++;

        }

    }

    public static void **main**(String[] *args*) {

        for (int n = 1000; n <= 5000; n += 100) {

            double promCompAleatorio = 0, promMovAleatorio = 0;

**sort**(**generateRangeArr**(1, n, false));

            double movOrdenado = movimientos;

            double compOrdenado = comparaciones;

**sort**(**generateRangeArr**(1, n, true));

            double movInverso = movimientos;

            double compInverso = comparaciones;

            for (int i = 0; i < (n / 10); i++) {

**sort**(**randomArray**(n, -n, n));

                promCompAleatorio += comparaciones;

                promMovAleatorio += movimientos;

            }

            promCompAleatorio /= (n / 10);

            promMovAleatorio /= (n / 10);

            System.out.**printf**("%d\t%f\t%f\t%f\t%f\t%.3f\t%.3f\n",

            n,compOrdenado, movOrdenado, compInverso, movInverso,

            promCompAleatorio, promMovAleatorio);

*// n, compOrdenado, movOrdenado, compInverso, movInverso, promCompAleatorio, promMovAleatorio*

        }

    }

}

|  |  |
| --- | --- |
| Criterio de evaluación | Puntos |
| Ejercicio 1. Código y salida correctos | 10 |
| Ejercicio 1. Gráfica de dispersión con título, línea de tendencia y ecuación | 10 |
| Ejercicio 2. Código y salida correctos | 10 |
| Ejercicio 2. 6 gráficas de dispersión con título, línea de tendencia y ecuación | 10 |
| Ejercicio 2. Respuesta argumentada a las preguntas en cursiva | 10 |
| Ejercicio 2. Ecuaciones finales resultantes para los 3 casos | 10 |
| Ejercicio 3. Código y salida correctos | 10 |
| Ejercicio 3. 6 gráficas de dispersión con título, línea de tendencia y ecuación | 10 |
| Ejercicio 3. Respuesta argumentada a las preguntas en cursiva | 10 |
| Ejercicio 3. Ecuaciones finales resultantes para los 3 casos | 10 |
| TOTAL | **100** |

# Parte 3: análisis de BubbleSort

1. Calendario

   Descripción generada automáticamenteImplementa en Java el algoritmo de ordenamiento BubbleSort. El comportamiento que se espera del algoritmo se puede apreciar en la corrida siguiente:

    public static void **bubbleSort**(int[] *arr*) {

        for (int i = 0; i < *arr*.length - 1; i++) {

            boolean swapped = false;

            for (int j = 0; j < *arr*.length - i - 1; j++) {

                if (*arr*[j] > *arr*[j + 1]) {

                    int temp = *arr*[j];

*arr*[j] = *arr*[j + 1];

*arr*[j + 1] = temp;

                    swapped = true;

                }

            }

            if (!swapped) break;

        }

    }

1. Mediante un análisis a priori, calcula el tiempo de ejecución del BubbleSort para cada uno de los siguientes costos, considerando el mejor (ordenado) y el peor caso (invertido):
   1. *c*1: costo de un movimiento entre **datos** en el arreglo o copiados de él.
   2. *c*2: costo de una comparación entre **datos** en el arreglo o copiados de él.
2. Mediante un análisis a posteriori, calcula el tiempo de ejecución del BubbleSort para cada uno de los costos anteriores, considerando el caso promedio. Sigue la metodología que se desarrolló en la Sesión 4 y la Tarea 2. Incluye en la respuesta los gráficos de dispersión de Excel con las ecuaciones correspondientes y con la misma escala.

Escribe en la tabla siguiente los resultados obtenidos en los pasos 2 y 3.

|  |  |  |
| --- | --- | --- |
| **Caso** | **Movimientos** | **Comparaciones** |
| Mejor | 0 | N |
| Peor |  |  |
| Promedio |  |  |

1. ¿A qué conjunto(s) pertenece cada una de las siguientes funciones? [notación asintótica]
2. ¿Con cuáles propiedades cumple el algoritmo BubbleSort?

Justifique cada respuesta de forma breve y clara.

* 1. El subarreglo *A*[1 .. *i* – 1] siempre está ordenado. 🡪 Suponiendo que cambios entre elementos del arreglo ocurren, el orden se puede alterar y un par [Aj, Aj+1] (desde el ciclo interno) puede encontrarse en orden inverso.
  2. El subarreglo *A*[1 .. *i* – 1] tiene los mismos elementos que el original. 🡪 Dado que una cantidad da arbitraria de swaps tendrá lugar en el algoritmo, no se puede asegurar que los mismos elementos del arreglo se encuentren en sus mismas posiciones originales.
  3. El subarreglo *A*[1 .. *i* – 1] tiene los elementos definitivos. 🡪 La manera en la que funciona BubbleSort es precisamente “burbujeando” los elementos más grandes al final del arreglo, de forma que los elementos menos van cambiando su posición hacía el inicio del arreglo, por ello, no se les puede considerar definitivos si no hasta la finalización de la ejecución.
  4. El subarreglo *A*[*n* – *i* + 1 .. *n*] siempre está ordenado. 🡪 Dado que los elementos van subiendo en el arreglo para quedar establecidos, estas porciones del arreglo estarán ordenadas tomando en cuenta el modo de operación de BubbleSort.
  5. El subarreglo *A*[*n* – *i* + 1 .. *n*] tiene los mismos elementos que el original. 🡪 Similar a la propiedad analizada anteriormente, dado que existen una serie de swaps, no se puede asegurar que los elementos originales se encuentren en sus posiciones originales, a menos que el arreglo ya se encontrase ordenado.
  6. El subarreglo *A*[*n* – *i* + 1 .. *n*] tiene los elementos definitivos 🡪 Nuevamente, analizando el comportamiento de BubbleSort, conforme se vaya iterando sobre el arreglo, los elementos mayores subirán hasta las últimas posiciones posibles y ya no se necesitará modificarlos.
  7. Es estable. 🡪 Considerando comparaciones entre una única propiedad de los objetos pertenecientes al arreglo completo, los datos con identificadores similares y propiedades distintas quedarán en el mismo orden que el arreglo original.

|  |  |
| --- | --- |
| Criterio de evaluación | Puntos |
| Ejercicio 1. Código fuente claro y correcto | 20 |
| Ejercicio 2. Análisis a priori, ambos casos | 10 + 10 |
| Ejercicio 3. Análisis a posteriori (2 gráficas y ecuaciones) | 15 + 15 |
| Ejercicio 4. ¿A qué conjunto pertenece cada función? [Notación asint.] | 9 |
| Ejercicio 5. Propiedades de *BubbleSort* (y justificaciones breves) | 7 + 14 |
| TOTAL | **100** |

# Parte 4: análisis de heapsort

Después de un semestre de tomar clases con otros profesores, Selena cometió el error de volver a inscribirse con su *profesor pedagógico*. En esta ocasión usted debe ayudar a Selena y sus amigos con sus conocimientos sobre algoritmos de ordenamiento y análisis de algoritmos.

En esta tarea, el profesor quiere que queden claros (o reforzados) los siguientes temas:

* La implementación de la interfaz *Comparable*
* *Heap-sort*
* La propiedad de estabilidad en algoritmos de ordenamiento
* El análisis a posteriori de caso *promedio*

Para eso, el profesor de Selena propone que los estudiantes implementen en Java una clase *Borrego* con sólo dos atributos: *peso* en kilogramos (entero) y *nombre*. Esta clase implementará la interfaz *Comparable*. En una clase *Heapsort* deberán implementar un método *heapsort* que reciba un arreglo de objetos comparables y los ordene utilizando ese algoritmo. Notemos que el método deberá operar con todo tipo de objetos que implementen la interfaz *Comparable*, no sólo con *Borrego*. ¡Ayude a Selena a terminar su tarea!

# Descripción

## 1. Clase *Borrego*

La clase debe implementar, a su vez, la interfaz *Comparable*.

**class** Borrego **implements** Comparable<Borrego>

Los atributos (privados) son:

* *kilos* (entero)
* *nombre* (cadena)

Los métodos son:

* Un constructor que recibe valores para ambos atributos
* *toString()*
* *compareTo()*

El segundo método debe devolver una cadena con el siguiente formato:

"%d kilos: borrego %s. ¡Mbaaah!", kilos, nombre

El tercer método es heredado de la interfaz *Comparable*.

**public** **int** compareTo(Borrego b)

Este método tomará al atributo *kilos* como único criterio para comparar borregos entre sí: por ejemplo, si el borrego *A* tiene menos kilos que el borrego *B*, *A* es menor que *B*. Para efectos de esta comparación, si dos borregos tienen el mismo número de kilos contarán como "iguales".

Es responsabilidad del estudiante investigar cómo debe operar la función *compareTo*.

public class Borrego implements Comparable<Borrego> {

    private int kilos;

    private String nombre;

*// Constructor*

    public **Borrego**(int *kilos*, String *nombre*) {

        try {

**setKilos**(*kilos*);

**setNombre**(*nombre*);

        } catch (BorregoException *e*) {

            System.out.**printf**("No fue posible crear al borrego {%d, \"%s\"}.\nError: %s",

*kilos*, *nombre*, e.**getMessage**());

        }

    }

*// Setters*

    private void **setKilos**(int *kilos*) throws BorregoException {

        if (*kilos* < 0) throw new **BorregoException**("Kilos inválidos.");

        this.kilos = *kilos*;

    }

    private void **setNombre**(String *nombre*) throws BorregoException {

        if (*nombre*.**length**() == 0) throw new **BorregoException**("Nombre inválido.");

        this.nombre = *nombre*;

    }

*// Impresión de borrego*

    @Override

    public String **toString**() { *// Formateo de cadena*

        return String.**format**("%d kilos: borrego %s. ¡Mbaaah!",

        kilos, nombre);

    }

*// Minimizado a ternario*

    public int **compareTo**(Borrego *borrego*) {

        return (this.kilos > *borrego*.kilos) ? 1 :

        (this.kilos < *borrego*.kilos) ? -1 : 0;

    }

}

## 2. Clase *Heapsort*

Esta clase debe implementar el algoritmo de ordenamiento. Para esto, implemente los siguientes métodos:

**private** **static** <T **extends** Comparable<? **super** T>> **boolean** greaterThan(T a, T b)

**private** **static** **int** left(**int** k)

**private** **static** **int** right(**int** k)

**private** **static** <T **extends** Comparable<? **super** T>> **void** maxHeapify(T a[], **int** root, **int** heapSize)

**public** **static** <T **extends** Comparable<? **super** T>> **void** heapsort(T a[])

El primer método deberá llamar al método *compareTo()* del objeto *a*, y después servirá para contar las comparaciones en el análisis a posteriori. Éste es el único método que podrá llamar a *compareTo*(), de forma que todas las comparaciones sean contabilizadas.

El segundo y el tercer método deben calcular el índice del hijo izquierdo o derecho partiendo del nodo con índice *k*.

El cuarto método implementará el algoritmo visto en clase, sirviéndose de los métodos anteriores.

El quinto método también implementará el algoritmo visto en clase, sirviéndose del cuarto método. Este método incluirá lo que en clase apareció como *buildMaxHeap*().

NOTA: tal como la parametrización lo implica, el método *heapsort()* debe operar con cualquier objeto *Comparable* y no sólo con objetos *Borrego*.

Antes de continuar, compruebe que su implementación funciona.

public class Heapsort {

    private static <T extends Comparable<? super T>> boolean **greaterThan**(T *a*, T *b*) {

       return (*a*.**compareTo**(*b*) == 1) ? true : false;

    }

    private static int **left**(int *k*) { return (2\**k*) + 1; }

    private static int **right**(int *k*) { return (2\**k*) + 2; }

    private static <T extends Comparable<? super T>> void **swap**(T *a*[], int *A*, int *B*) {

        T temp = *a*[*A*]; *a*[*A*] = *a*[*B*]; *a*[*B*] = temp;

    }

    private static <T extends Comparable<? super T>> void **buildMaxHeap**(T *a*[]) {

        for (int root = (int)Math.**floor**(*a*.length/2); root >= 0; root--) {

**maxHeapify**(*a*, root, *a*.length);

        }

    }

    private static <T extends Comparable<? super T>> void **maxHeapify**(T *a*[], int *root*, int *heapSize*) {

        int left = **left**(*root*);

        int right = **right**(*root*);

        int max = *root*;

        if (left < *heapSize* && **greaterThan**(*a*[left], *a*[max])) max = left;

        if (right < *heapSize* && **greaterThan**(*a*[right], *a*[max])) max = right;

        if (max != *root*) {

**swap**(*a*, *root*, max);

**maxHeapify**(*a*, max, *heapSize*);

        }

    }

    public static <T extends Comparable<? super T>> void **heapsort**(T *a*[]) {

**buildMaxHeap**(*a*);

        for (int heapSize = *a*.length; heapSize > 1; heapSize--) {

**swap**(*a*, 0, heapSize - 1);

**maxHeapify**(*a*, 0, heapSize - 1);

        }

    }

}

## 3. Estabilidad del algoritmo

Ejecute *heapsort()* para varios arreglos de *Borrego* y despliegue en salida estándar los arreglos ya ordenados con el fin de determinar si el algoritmo es estable o no. Justifique su respuesta y haga capturas de pantalla que la respalden.

Pistas:

* Para este ejercicio necesita arreglos donde haya al menos un par de objetos *Borrego* con el mismo valor en *kilos* pero distinto valor en *nombre*. (¿Por qué?)
* Imprima el arreglo mostrando un *Borrego* por línea y llamando a su método *toString()*. Recuerde que los atributos de *Borrego* son privados.

La primera de las pistas indica que se ha de contar con al menos dos borregos con pesos similares y nombres distintos; recordando lo que significa que un algoritmo sea estable, contar con este par de objetos es útil para verificar si esta propiedad se estaría cumpliendo, de manera que la precedencia de sus posiciones no se vean alteradas una vez que el ordenamiento se haya llevado a cabo.

    public static void **main**(String[] *args*) {

        Borrego[] borregos = {

            new **Borrego**(21, "C"), new **Borrego**(23, "A"),

            new **Borrego**(23, "B"), new **Borrego**(25, "D"),

            new **Borrego**(23, "F"), new **Borrego**(22, "E")

        };

        System.out.**println**("Antes de sort");

**printArr**(borregos);

        Heapsort.**heapsort**(borregos);

        System.out.**println**("\nDespués de sort");

**printArr**(borregos);

    }

Un texto con letras negras

Descripción generada automáticamente

        Borrego[] borregos = {

            new **Borrego**(21, "C"), new **Borrego**(21, "A"),

            new **Borrego**(21, "B"), new **Borrego**(21, "D"),

            new **Borrego**(21, "F"), new **Borrego**(21, "E")

        };

Un conjunto de letras blancas en un fondo blanco

Descripción generada automáticamente con confianza media

Dados los casos anteriores mostrados, se puede concluir con facilidad que este algoritmo **no es estable**, puesto que el orden original de los datos satelitales no se conserva.

Después de esto, cambie la implementación de *compareTo()* en *Borrego* para que tome el *nombre* como segundo criterio de comparación.

    public int **compareTo**(Borrego *borrego*) { *// método modificado*

        return (this.kilos > *borrego*.kilos) ? 1 :

               (this.kilos < *borrego*.kilos) ? -1 :

               (this.nombre.**compareTo**(*borrego*.nombre) > 0) ? 1 :

               (this.nombre.**compareTo**(*borrego*.nombre) < 0) ? -1 : 0;

    }

Texto en fondo blanco

Descripción generada automáticamente

­­

## 4. Análisis a posteriori

Modifique el código de la clase *Heapsort* para contar (por separado) *comparaciones* entre elementos del arreglo y *movimientos* donde se involucren datos que estaban en el arreglo de entrada, fueron copiados de él o fueron calculados a partir de elementos en él.

Pistas:

* Cada *swap* implica 3 movimientos
* Cada llamada a *compareTo()* cuenta como una comparación

Nota: la clase *Borrego* no debe ser modificada para este objetivo, sino sólo la clase *Heapsort*. ¡Separe las responsabilidades de su código!

Ahora corra el algoritmo para arreglos de tamaño *n=1000* hasta *n=5000* con incremento de *10*:

* Generando en cada caso *n/10* arreglos de *Borrego* con *kilos* aleatorios y el mismo *nombre*.
* Calculando el promedio de *movimientos* y de *comparaciones* (por separado) para esos *n/10* arreglos.
* Despliegue la información con el siguiente formato:

"%d\t%.1f\t%d\t%.1f\n", n, promedioMovimientos, n, promedioComparaciones

Copie esos datos, péguelos tal cual en Excel y genere dos *gráficas de dispersión*:

1. Promedio de *movimientos* en función de *n*.
2. Promedio de *comparaciones* en función de *n*.

Agregue a estas gráficas la ecuación de tendencia y el valor de ajuste estadístico *R2*.

public class prueba { *// driver para probar heapSort junto con Borrego*

    static float promedioMovimientos, promedioComparaciones;

    public static <T> void **printArr**(T[] *arr*) {

        for (T elem : *arr*) System.out.**println**(elem);

    }

    public static Borrego **generarBorrego**(int *min*, int *max*) { *// mínimo y máximo de edad (inclusivo)*

        return new **Borrego**((int)(*min* + (*max* - *min* + 1) \* Math.**random**()), "genérico");

    }

    public static Borrego **generarBorrego**(int *max*) { *// valore desde 0 hasta máximo (inclusivo) para edad*

        return new **Borrego**((int)((*max* + 1) \* Math.**random**()), "genérico");

    }

    public static void **procesarArreglo**(Borrego[][] *borregos*, int *n*) {

        for (int i = 0; i < *n* / 10; i++) {

            for (int j = 0; j < *n*; j++) {

*borregos*[i][j] = **generarBorrego**(20);

            }

            Heapsort.**heapsort**(*borregos*[i]);

            promedioMovimientos += Heapsort.movimientos;

            promedioComparaciones += Heapsort.comparaciones;

        }

    }

    public static void **main**(String[] *args*) {

        for (int n = 1000; n <= 5000; n += 10) {

            promedioMovimientos = 0; promedioComparaciones = 0;

            Borrego[][] borregos = new Borrego[n / 10][n];

**procesarArreglo**(borregos, n);

            promedioMovimientos /= (n / 10); promedioComparaciones /= (n/10);

            System.out.**printf**("%d\t%.1f\t%d\t%.1f\n", n, promedioMovimientos, n, promedioComparaciones);

        }

    }

}

public class Heapsort { *// modificado para conteo*

    public static int movimientos, comparaciones;

    private static <T extends Comparable<? super T>> boolean **greaterThan**(T *a*, T *b*) {

        comparaciones++;

        return (*a*.**compareTo**(*b*) == 1) ? true : false;

    }

    private static int **left**(int *k*) { return (2\**k*) + 1; }

    private static int **right**(int *k*) { return (2\**k*) + 2; }

    private static <T extends Comparable<? super T>> void **swap**(T *a*[], int *A*, int *B*) {

        movimientos += 3;

        T temp = *a*[*A*]; *a*[*A*] = *a*[*B*]; *a*[*B*] = temp;

    }

    private static <T extends Comparable<? super T>> void **buildMaxHeap**(T *a*[]) {

        for (int root = (int)Math.**floor**(*a*.length/2); root >= 0; root--) {

**maxHeapify**(*a*, root, *a*.length);

        }

    }

    private static <T extends Comparable<? super T>> void **maxHeapify**(T *a*[], int *root*, int *heapSize*) {

        int left = **left**(*root*);

        int right = **right**(*root*);

        int max = *root*;

        if (left < *heapSize* && **greaterThan**(*a*[left], *a*[max])) max = left;

        if (right < *heapSize* && **greaterThan**(*a*[right], *a*[max])) max = right;

        if (max != *root*) {

**swap**(*a*, *root*, max);

**maxHeapify**(*a*, max, *heapSize*);

        }

    }

    public static <T extends Comparable<? super T>> void **heapsort**(T *a*[]) {

        movimientos = 0; comparaciones = 0;

**buildMaxHeap**(*a*);

        for (int heapSize = *a*.length; heapSize > 1; heapSize--) {

**swap**(*a*, 0, heapSize - 1);

**maxHeapify**(*a*, 0, heapSize - 1);

        }

    }

    public static <T extends Comparable<? super T>> boolean **isSorted**(T *a*[]) {

        for (int i = 0; i < *a*.length - 1; i++) {

            if (**greaterThan**(*a*[i], *a*[i + 1])) return false;

        }

        return true;

    }

}

Entregables y rúbrica

1. Clase *Borrego* (15%)
   1. Implementación de la clase en general 5%
   2. Implementación original de *compareTo()* 10%
2. Clase *Heapsort* (40%)
   1. *greaterThan()* 5%
   2. *left()* y *right()* 5%
   3. *maxHeapify()* 15%
   4. *heapsort()* 15%
3. Estabilidad del algoritmo (20%)
   1. Respuesta sobre la estabilidad (o no) 5%
   2. Justificación de la respuesta 5%
   3. Capturas de pantalla que la evidencien 5%
   4. Código de *compareTo()* modificado 5%
4. Análisis a posteriori (25%)
   1. Código de clase *Heapsort* modificado 5%
   2. Conteos correctos 5%
   3. Código invocador del algoritmo (*main*) 5%
   4. Gráficas de dispersión con ecuación y *R2* 10%

**Valor porcentual por cada parte:**

* Parte 1 (análisis a priori): 20%
* Parte 2 (análisis a posteriori): 20%
* Parte 3 (análisis de bubblesort): 30%
* Parte 4 (análisis de heapsort): 30%

Por ejemplo: si un estudiante obtiene todos los puntos de la primera parte, la mitad de los puntos de la segunda, ningún punto en la tercera y todos los de la cuarta, su puntaje final sería: 20% + 10% + 0% + 30% = 60% del total de la evaluación asincrónica 1.