**Programación Avanzada, indice de contenidos Reflejo**

|  |  |
| --- | --- |
| Clase | Contenido |
| 1 | Presentación de la materia. |
| 2 | Lotes de prueba (Practica y teoría) |
| 3 | Clase de práctica, Scanner – PrintWriter. |
| 4 | Nomenclaturas, Tipos de Datos, Equals, Hashcode, etc. |
| 5 | Ejercicios luchadores japoneses, planteo, diseño, codificación. |
| 6 | Clase abstracta, herencia, interfaz |
| 7 | TP 1 Práctico de Lote de Prueba. |
| 8 | TP Teórico, Collections y Excepciones. |
| 9 | Complejidad computacional + TP Teórico. |
| 10 | Complejidad computacional. |
| 11 | Algoritmos de ordenamiento avanzados y montículo. |
| 12 | Prolog |
| 13 | Prolog |
| 14 | Prolog |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

**Segunda clase (Lotes de prueba):**

* Rol de preparador de prueba: Tengo que mostrar las posibles salidas para una misma entrada.
* Rol de desarrollador: Con una entrada muestra ALGUNA de las posibles salidas, solo UNA (sí lo especifica el enunciado).

Descripción del caso: Que no describa la entrada o la salida, sino que es lo que quiero probar.

**Lo importante es encontrar CASOS PARTICULARES, no casos “normales”**

Hay problemas con 6 casos son suficientes, y otros que se necesitan 20 casos por ejemplo.

En el transcurso del desarrollo (es decir después de las pruebas) me puedo dar cuenta que hay algún caso mas, lo puedo agregar entonces, no importa sí estoy en el desarrollo.

**Sí hay dudas, ver ese caso que hizo la duda.**

**El caso de prueba debe ser ATÓMICO**

**El Output es el ESPERADO no el que me dio por desarrollar el programar, puedo apoyarme con softwares auxiliares.**

El nombre del caso de prueba debe ser el mismo que el del archivo correspondiente.

**Salida Múltiple no es lo mismo que caso de empate**.

**Pueden existir problemas sin programa probador.**

**Tipos de salidas:**

* Salidas Complejas: Programa probador – Lote de prueba.
* Salidas Múltiples: Programa probador – Lote de prueba.
* Salidas Simples(únicas): Lote de prueba.
* Salidas no verificables: Lote de prueba.

**En salidas complejas o salidas múltiples, es mas recomendable programa probador.**

Nomenclaturas

* UnProyecto
* UnaClase (Singular)
* unObjeto
* unPaquete
* UNA\_CONSTANTE (final)

Las variables se utilizan en plural cuando se tiene un array de alumnos o un conjunto de algo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | PUBLIC | Default | PROTECTED | PRIVATE |
| Misma Clase | X | X | X | X |
| Mismo Package Sub-Clase | X | X | X |  |
| Mismo Package Otra Clase | X | X | X |  |
| Otro Package  Sub-Clase | X |  | X |  |
| Otro Package  Otra Clase | X |  |  |  |

**Java tiene dos tipos de datos**:

* Primitivos: int(32), long(64), short(16), float(32), double(64), char(8), boolean(8), byte(8).
* Referencia: Java no tiene PUNTEROS. Integer – Long – Short – Float – Double – Boolean – String. Todas pesan lo mismo una REFERENCIA A MEMORIA.

El int y el Integer ambos se pasan por copia, no referencia justo ese caso especial.

El final funciona como const, es decir no puedo modificar dentro de una función la referencia al objeto, pero sí el objeto en sí mismo.

Public static void incrementar(Object e)

{

e = new Object();

}

Public static void incrementar(final Object e)

{

e = new Object(); // NO PUEDO HACER ESTO.

E++; // Me modifica el objeto.

}

Java pasa los objetos por COPIA, de referencia o de valor, pero COPIA.

Siempre que se trate de objetos comparar con equals.

Object e1 = new Object();

Object e2 = e1;

if(e1 == e2) // Devuelve TRUE.

Utilizar el equals generado

hashCode generado también a la par con equals.

Object[] objetos = new Object[5];

Crea en el stack una referencia hacia la memoria reservada en el heap.

New Object[10] llama al constructor de ARRAYS no de OBJECT.

**Foreach explicado en clase.**

Clase Punto2D – Punto3D

Interface Punto

Punto2D = Punto3D

Ahora Punto2D es Punto3D

Punto(interface) P1 = New Punto2D();

Punto(interface) P2 = New Punto3D();

P1 = P2;

P2 = P1;

SE PUEDE

Punto[] vecPuntos = new Punto [4];

for(int i=0; i<4; i++)

vecPuntos[i] = new Punto3D(1, 1, 1);

SE PUEDE

POLIMORFISMO EN JAVA.

Las clases abstractas llevan constructor.

El SUPER siempre arriba, no puede existir nada arriba de SUPER.

**HACER EJERCICIO DE VEHICULO.**

Realización del TP1 Teórico.

**Explicación de Collection.**

ArrayList muy costoso para “resizear”.

LinkedList muy costoso para acceder a los elementos del “medio”.

Lo mejor es realizar List<String> list = new ArrayList<string>();

Porque puedo cambiar la implementación cuando deseo.

* IList
* ISet
* IQueue
* Imap

**Explicación de Exceptions**

Ver árbol de jerarquía de excepciones.

* Exception
  + RuntimeException.
  + IOException.

En RuntimeException tengo las excepciones que ocurren en tiempo de ejecución. Las excepciones que no son RuntimeException se deben tratar SI o SI digamos.

Try – Catch – Finally – Throw – Throws

**Se trata la excepción con try – catch se delega para arriba con throw.**

**Algoritmos de ordenamiento**

|  |  |  |  |
| --- | --- | --- | --- |
| Nombre | Complejidad | Estable | Sensible |
| Burbuja | O(n²) | SI | SI |
| Selección | O(n²) | NO | NO |
| Inserción | O(n²) | SI | SI |
| Shell | [O](https://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(*n*1.25) | NO |  |
| Quick | Promedio: O(n log n)  Peor: O(n²) | NO | SI |
| Merge | O(n log n) | SI |  |

* ***Tipos de Algoritmo***
* Algoritmo Sensible: Modifica su tiempo de ejecución según el tipo de entrada.
* Algoritmo No Sensible: Su tiempo de ejecución es independiente al tipo de entrada.
* Algoritmo Estable: Aquellos que teniendo clave repetida, mantiene su posición inicial igual a la final.
* Algoritmo No Estable: Aquello que no respetan la posición inicial igual que la final teniendo claves repetidas

Tanto para burbuja, inserción y selección, el **caso óptimo** es sí los elementos ya se encuentran ordenados y la complejidad computacional es de O(n).

Tanto para burbuja, inserción y selección, el **peor caso o caso desfavorable**, es sí los elementos se encuentran ordenados a la inversa, y la complejidad computacional es de O(n²)

[*novella.mhhe.com/sites/dl/free/844814077x/619434/A06.pdf*](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=2ahUKEwifpZaj-YXeAhVGjJAKHa_WDuQQFjABegQIBxAC&url=http%3A%2F%2Fnovella.mhhe.com%2Fsites%2Fdl%2Ffree%2F844814077x%2F619434%2FA06.pdf&usg=AOvVaw1QmmZeZgAOfM1uoNlFu5KC)

* **Inserción / Selección**: Para listas con pocos elementos.
* **QuickSort**: Para listas con muchos elementos.
* **MergeSort**: Para listas con muchísimos elementos.
* **ShellSort**: Suele variar mucho.
* **Burbuja**: El de peor rendimiento.

**Complejidad Computacional.**

* Toda pila tiene lógica computacional O(1)
* Toda búsqueda binaria tiene lógica computacional O(Log n)
* Toda búsqueda secuencial tiene lógica computacional O(n)
* Los sort de las colecciones y arrays (quicksort) tienen lógica computacional O(n log n)
* El algoritmo shell tiene lógica computacional O(n elevado a la 3/2)
* Después los algoritmos con lógica computacional O(n elevado).
* Después los algoritmos con lógica exponencial O(2 elevado a la n).
* El peor caso son los algoritmos con lógica computacional O(n!)

**Regla de la suma**

Cuando sumo complejidades, me quedo con la peor.

**Regla de la multiplicación**

Cuando hay una complejidad dentro de otra, se multiplican las complejidades.

**Practica de Complejidad Computacional**

1)

    a) 6+n^2 = O(n2)

    b) 4+n+2logn = O(n)

    c) n2 + 3\*2^n = O(2^n)

    d)

2)

    a) Falso

    b) Falso

    c) Falso

    d) Verdadero

    e) Falso

    f) Verdadero

    g) Verdadero

3)

    a) O(a² Log2a)

    b) O(n²)

Problemas 14, 15, 16, 17 son los que entran siempre en el parcial.

14)

    a)    10 = C (10000)²

        C = 10^-7

        30 = 10^-7 \*N²

        N = (30 – 10^-7)^1/2

    b)    T = 10^-7(30000)²

        T = 90

    c)    T = (10^-7) / 3 \* (30000)²

        T= 30

15) 20 = C \* (2000)²

    C = ¼ \* 10^-5

    T =  ¼ \* 10^-5 \* (5000)²

    T = 62,5

    40 =  ¼ \* 10^-5\* (N)²

    N = 4000

19) T1 = C1 \* N1²

    2T1 = C2 \*2N1^2

    2C1 = 4C2

    C2 = C1/2

20) 2T1 = C \* N \* Log N

**Algoritmos de ordenamiento avanzados.**

**Shell (O3/2)**

5 8 2 1 6 7 9 3 4 3 2 1

2 5 7

1 8 9

2 8

1 3

4 5

6

2 1 2 1 3 5 8 3 4 6 7 9

2 2 3 4 7 8

1 1 3 5 6 9

2 1 2 1 3 3 4 5 7 6 8 9

1 2 2

1 1 2 2 3 3 4 5 7

1 1 2 2 3 3 4 5 5 6 7 8 9

Algoritmos que trabajan con vectores k\_ordenados

2\_ordenado

Los elementos que distan dos están ordenados.

20 7 24 35 48 46

La idea es empezar con un K grande y empezar a ordenar por inserción.

Ki = ki-1

\* 3 +1

Ki = 1, 14, 13, 40 ...

Ver video para entender.

**Quicksort (NLogN)**

Sí un elemento tiene para el lado de las posiciones mas bajas del vector, elementos menores a el y para el

lado de las posiciones mas altas del vector, elementos mayores a el, se lo toma como pivote al numero y

se repite lo siguiente en forma recursiva.

2 6 5 3 8 7 1 0

2 6 5 0 8 7 1 3

2 1 5 0 8 7 6 3

2 1 0 3 8 7 6 5

8 7 6 5

8 5 6 7

6 5 8 7

6 5 7 8

Del elemento pivote elegido lo intercambio con el de la ultima posición (SIEMPRE AL PRINCIPIO)

**Fusión – Merge Sort (NLogN)**

2 8 5 3 9 4 1 7

2 8 5 3 9 4 1 7

2 8 5 3 9 4 1 7

2 8 5 3 9 4 1 7

2 8 3 5 4 9 1 7

2 3 5 8 1 4 7 9

1 2 3 4 5 6 7 8 9

Cuenta de distribución particular (N)

13 10 12 14 11

Claves no duplicadas

Claves consecutivas entre n y k.

10 11 12 13 14 (obvio jaja)

Cuenta de distribución general (KN)

Pocas claves consecutivas repetidas muchas veces.

B A B B C A C A

A B C

3 3 2

A B C

3 6 8

A – 1 = 2

- - A - - - - -

C – 1 = 7

- - A - - - - C

**Montículo TDA**

• Árbol Binario Casi Completo

• Las hojas se llenan de izquierda a derecha

• Si estamos en un montículo de máximo el padre tiene que ser mayor que los hijos.

• Si estamos en un montículo de mínimo el padre tiene que ser menor que los hijos.

La implementación es en un vector.

El elemento 0 no se usa.

En la raíz esta el elemento mas grande o mas chico.

0 59 27 45 20 3 15

HI = i \* 2

HD = i \* 2 + 1

Padre = i / 2

Cuando quiero sacar un elemento, el único elemento que puedo sacar es el elemento de la raíz.

Es una estructura de datos que me permite tener los datos ordenados.

Cada vez que agrego un elemento tengo que preguntar si cumple la condición del montículo.

Cuando quiero hacer una cola de prioridad lo puedo implementar sobre un montículo.

Cada vez que saco un elemento (de la raíz), tomo el ultimo elemento agregado lo asigno como raíz y

empiezo a preguntar.

**Programación Dinámica**

Se puede aplicar programación dinámica cuando la solución es recursiva.

• Ver la recursividad de abajo hacia arriba, guardo todo lo que aparece.

• Ver la recursividad de arriba hacia abajo, si tengo el número lo uso sino lo calculo.

Algoritmo greedy, es un algoritmo que en cada iteración busca la mejor solución.