Laboratorio A.E.D. Laboratorio 4

Guillermo Román
guillermo.roman@upm.es

Marina Álvarez
marina.alvarez@upm.es

Lars-Åke Fredlund
larsake.fredlund@upm.es

Julio García
juliomanuel.garcia@upm.es

Manuel Carro
manuel.carro@upm.es

Tonghong Li
tonghong.li@upm.es

Normas

- Fechas de entrega y penalización asociada:

Hasta el miércoles 16 de noviembre, 23:59 horas	0 %
Hasta el jueves 17 de noviembre, 23:59 horas	20 %
Hasta el viernes 18 de noviembre, 23:59 horas	40 %

  Después la puntuación máxima será 0
- Se comprobará plagio y se actuará sobre los detectados.
- Usad las horas de tutoría para preguntar sobre programación – son oportunidades excelentes para aprender.

Entrega

- Todos los ejercicios de laboratorio se deben entregar a través de

  `http://deliverit.fi.upm.es`

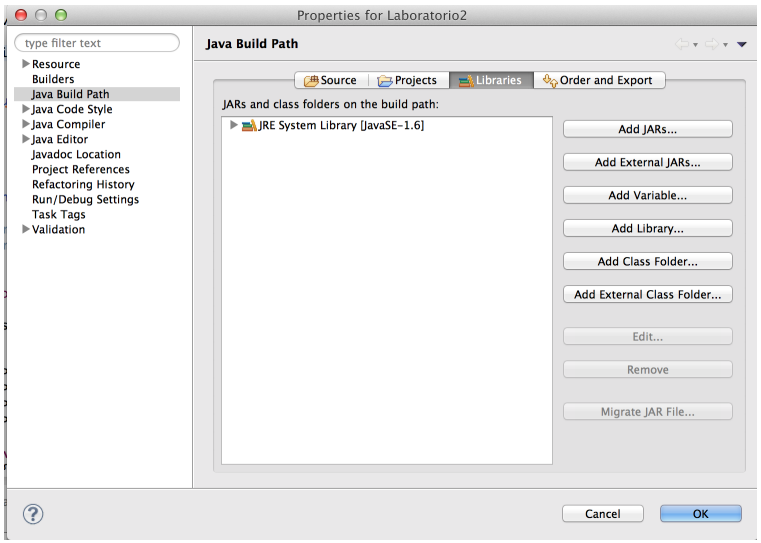
- Los ficheros que hay que subir es `Utils.java`, `Explorador.java`.

Configuración previa

- Arrancad Eclipse
- Podéis utilizar cualquier versión reciente de Eclipse. Es suficiente con que instaléis la *Eclipse IDE for Java Developers*.
- Cambiad a "Java Perspective".
- Debéis tener instalado al menos Java JDK 8.
- Cread un proyecto Java llamado `aed`:
  - Seleccionad separación de directorios de fuentes y binarios.
  - **No debéis elegir la opción de crear el fichero** `module-info.java`
- Cread un *package* `aed.recursion` en el proyecto `aed`, dentro de `src`
- Aula Virtual → AED → Laboratorios → Laboratorio 4 → `Laboratorio4.zip`; descomprimidlo
- Contenido de Laboratorio4.zip:
  - `Lugar.java`, `Punto.java`, `TesterLab4.java`, `PuntoCardinal.java`, `MyInteger.java`, `Explorador.java`, `Utils.java`

Configuración previa

- Importad al paquete `aed.recursion` los fuentes que habéis descargado (`Lugar.java`, `Punto.java`, `TesterLab4.java`, `PuntoCardinal.java`, `MyInteger.java`, `Explorador.java`, `Utils.java`)
- Si no lo habéis hecho, añadid al proyecto `aed` la librería `aedlib.jar` que tenéis en Moodle (en Laboratorios).

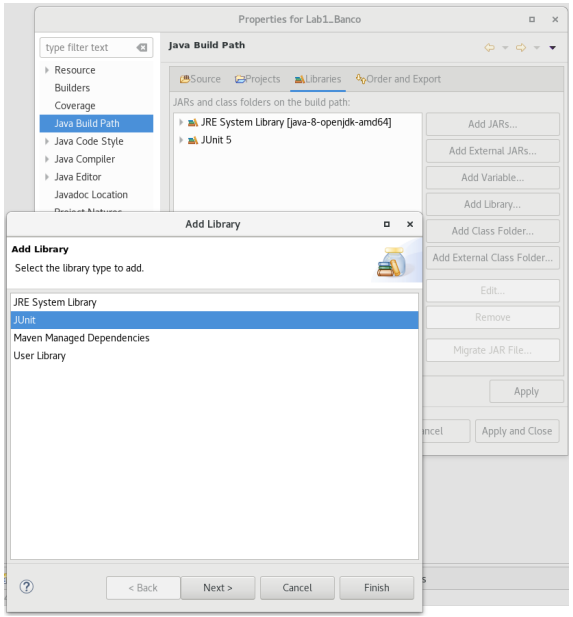


Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda
- Usad la opción "Add External JARs...".
- Si vuestra instalacion distingue `ModulePath` y `ClassPath`, instalad en `ClassPath`

Configuración previa

- Si no lo habéis hecho, añadid al proyecto `aed` la librería `JUnit 5`



Para ello:

- Project → Properties → Java Build Path. Se abrirá una ventana como la de la izquierda;
- Usad la opción "Add Library..." → Seleccionad "Junit" → Seleccionad "JUnit 5"
- Si vuestra instalacion distingue `ModulePath` y `ClassPath`, instalad en `ClassPath`

- En la clase `TesterLab4` tenéis las pruebas, para ejecutarlas, abrid el fichero `TesterLab4`, pulsando el botón derecho sobre el editor, seleccionar "Run as..." → "JUnit Test"
- NOTA: Si al ejecutar no aparece la vista "JUnit", podéis incluirla en "Window" → "Show View" → "Java" → "JUnit"

Documentación de la librería aedlib.jar

- La documentación de la API de `aedlib.jar` esta disponible en `http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/`
- Tambien se puede añadir la documentación de la librería a Eclipse (*no es obligatorio*): en el "Package Explorer": "Referenced Libraries" → aedlib.jar y elige la opción "Properties". Se abre una ventana donde se puede elegir "Javadoc Location" y ahí se pone como "javadoc location path:"

  `http://costa.ls.fi.upm.es/~entrega/aed/docs/aedlib/`

  y presionar el buton "Apply and Close"

Tarea para hoy: programar con `recursión`

- Las clases `Utils` y `Explorador` contienen cuatro métodos a completar: `multiply`, `findBottom`, `jointMultiSets` (en `Utils`), y `explora` (en `Explorador`).
- Es **obligatorio** usar recursión en la implementación de estos métodos
- En la clase `Utils`: está **prohibido** usar bucles `for`, `for-each`, `while`, `do-while`, o iteradores
- En la clase `Explorador`: sólo está **permitido** el uso de bucles para explorar los caminos que salen de un `Lugar` (ver después)
- Está **permitido** (y **será necesario**) añadir métodos auxiliares para implementar correctamente los métodos mencionados
- **NO está permitido** añadir nuevos atributos a clases
- **NO está permitido** crear nuevas estructuras de datos como pilas, colas, etc. Únicamente está permido crear nuevas `NodePositionList` para los resultados a devolver por el método `joinMultiSets` y/o sus métodos auxiliares.

Tarea para hoy: programar con `recursión`

- Para poder entregar es necesario completar al menos tres de los cuatro métodos a programar.
- Maxima puntuación por entregar soluciones perfectas:
  - Multiplicación, findBottom y joinMultiSets: 10
  - Método explora: 3 puntos

Tarea: implementar un algoritmo de multiplicación

- Implementa *recursivamente* el siguiente algoritmo para multiplicar dos enteros a y b:
  - $sign := -1$ si $a < 0$ y 1 si no
  - $sum := 0$
  - while $a \neq 0$
    - ★ $sum := sum + b$ si $a \ mod \ 2 \neq 0$
    - ★ $a := a/2$
    - ★ $b := b * 2$
  - `return` $sign * sum$
- Concretamente hay que implementar el método

```
public static int multiply(int a, int b)
```

  usando recursión, es decir, **sin** usar bucles `while`, `for`, etc.
- Como siempre, el uso de métodos auxiliares está permitido si se considera necesario

Un ejemplo

- Multiplicamos $a = 579$ y $b = 342$:

a	b	sum	
579	342	342	porque $a \; mod \; 2 \neq 0$
289	684	$342 + 684$	porque $a \; mod \; 2 \neq 0$
144	1368	$342 + 684$	
72	2736	$342 + 684$	
36	5472	$342 + 684$	
18	10944	$342 + 684$	
9	21888	$342 + 684 + 21888$	porque $a \; mod \; 2 \neq 0$
4	43776	$342 + 684 + 21888$	
2	87552	$342 + 684 + 21888$	
1	175104	$342 + 684 + 21888 + 175104$	porque $a \; mod \; 2 \neq 0$
0	350208	$342 + 684 + 21888 + 175104$	

- El resultado es $342 + 684 + 21888 + 175104 = 198018$

Tarea: implementar `findBottom`

- Implementad el método

```
public <E extends<Comparable>> static int
  findBottom(IndexedList<E> list)
```

  que devuelve un indice en `list` que corresponde a un "hoyo"

- Un "hoyo" es un elemento de `list` que no es mayor que sus vecinos. Si i es el índice de un hoyo y tiene dos vecinos se cumple:

```
list.get(i-1) >= list.get(i) <= list.get(i+1)
```

- Por ejemplo:
  - En la lista `[1,2,3,4]` el elemento `1` es el único hoyo
  - En la lista `[1,4,5,3,3]` hay tres hoyos: `1` (que solo tiene un vecino), y los dos elementos `3`
- Si hay múltiples hoyos, `findBottom` puede devolver el índice de cualquiera de ellos. Si no hay un hoyo, el método debe devolver `-1`.
- La lista `list` nunca contiene elementos `null`.

Una solución eficiente

- Haciendo un recorrido sobre todos los elementos de la lista para buscar un hoyo implica una complejidad asintótica $O(n)$
- Sin embargo para obtener la máxima puntuacion habría que conseguir una implementación con complejidad asintótica $O(\log n)$
- En clase habéis visto cómo el algoritmo de "búsqueda binaria" (binary search), para encontrar un elemento en una lista ordenada, consigue un complejidad asintótica de $O(\log n)$: en cada paso el algoritmo reduce el intervalo para buscar a la mitad siguiendo la técnica de "divide y vencerás".

Divide y Vencerás

- Un poquito de teoría: las tareas de encontrar un elemento en una lista ordenada, o encontrar un hoyo, son problemas que se puede resolver usando la estrategia de "divide y vencerás" o "divide and conquer". Segun wikipedia:

  > *"El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original."*

- Si logramos expresar una tarea como un algoritmo "divide y vencerás" es probable que obtengamos un algoritmo eficiente.

Implementación eficiente de `findBottom`

- Podemos ver el problema como una variante de esquema de "binary search":
  `findBottom(E[] array, int starts, int ends)`
  donde `starts` y `ends` delimitan un *subarray* del array.
  1. Si el tamaño del subarray es 1, su índice es el de un hoyo.
  2. Si el subarray tiene tamaño dos, el menor elemento es un hoyo.
  3. Si el subarray tiene tamaño 3 o más, iremos al punto medio del intervalo, veremos si es un hoyo y decidiremos qué intervalo seleccionamos comparando el valor en el punto medio con sus vecinos.

Tarea: implementar `joinMultiSets`

Implementar el método

```
public static <E extends Comparable<E>>
  NodePositionList<Pair<E,Integer>>
  joinMultisets(NodePositionList<Pair<E,Integer>> l1,
                NodePositionList<Pair<E,Integer>> l2)
```

- El método recibe dos listas `l1` y `l2`. Cada lista representa multisets mediante pares `Pair(Element,Count)`. `Count` indica el número de elementos `Element` en el multiset.
- **Importante**: las listas `l1` y `l2` están ordenadas. Si `Element1` $<$ `Element2`, entonces `Pair(Element1,Count1)` aparece antes de `Pair(Element2,Count2)`.
- El método `joinMultisets` deber devolver una lista **nueva** que representa la unión de los multisets `l1` y `l2`.
- **Obligatorio**: la lista devuelta por `joinMultisets` también tiene que estar ordenada en orden creciente.

Multisets/Multiconjuntos

- Un "multiset" se comporta como un set, excepto que los multisets admiten elementos repetidos
- Ejemplo: el multiset $\{1,3,1,2\}$ contiene dos enteros 1, un 3, y un 2
- Sería facil representar un multiset con una lista:

$$\{1,4,9,9,4,9\} \Rightarrow \texttt{[1,4,9,9,4,9]}$$

- Sin embargo vamos a usar una representación más eficiente: una lista de pares: `Pair(Element,Count)` – `Count` indica cuantas occurencias de `Element` hay en el multiset.

$$\{1,4,9,9,4,9\} \Rightarrow \texttt{[Pair(1,1),Pair(9,3),Pair(4,2)]}$$

Observaciones

- Se puede asumir que las listas nunca contienen elementos `null`
- Para comparar dos elementos de tipo `E` se puede usar el método `compareTo` ya que `E` implementa el interfaz `Comparable`
- Recuerda los invariantes de la representación de multisets con una lista de pares:
  - `Count > 0` para todos los elementos `Pair(Element,Count)` de una lista.
  - No puede haber dos pares `Pair(Element,Count1)` `Pair(Element,Count2)` con el mismo elemento `Element`.

  y la nueva invariante sobre la ordenación:
  - Un par `Pair(Element1,Count1)` precede a un par `Pair(Element2,Count2)` si y sólo si `Element1` $\leq$ `Element2`.

Ejemplo

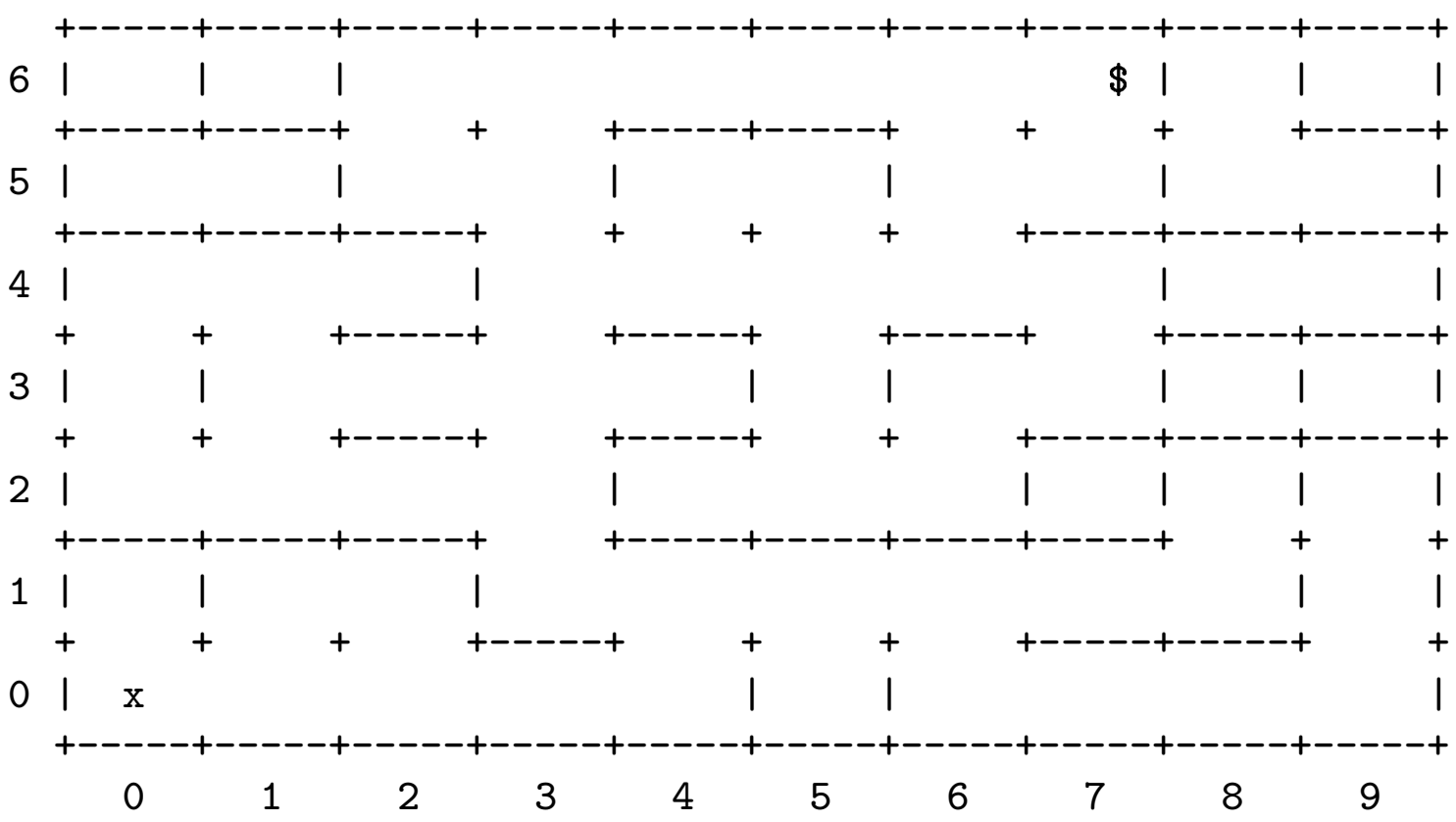
Asumimos que s es un multiset $\{1,1,2,2,3\}$ y t es un multiset $\{4,4,1\}$:

```
s.toString();       =>   [Pair(1,2),Pair(2,2),Pair(3,1)]
t.toString();       =>   [Pair(1,1),Pair(4,2)]

PositionList<Pair<Integer,Integer>> joined =
  RecursiveMethods.joinMultisets(s,t);

joined.toString(); =>   [Pair(1,3),Pair(2,2),
                         Pair(3,1),Pair(4,2)]
```

Un reto: `explora`: explorar un laberinto usando recursión




```
x = posicion inicial
$ = lugar del tesoro
```

Un reto: `explora`: explorar un laberinto usando recursión

- **Fichero a modificar:** `Explorador.java`
- **Objetivo:** desarrollar un método `explora()` en la clase `Explorador` que recorra un laberinto sistemáticamente y encuentre un "tesoro".
  - El laberinto esta compuesto por "lugares", implementados como objetos de la clase `Lugar`
  - En un lugar:
    - ★ Puede haber un tesoro (un objeto, no nulo).
    - ★ Cada lugar puede estar marcado (para detectar que ya ha sido visitado).
    - ★ Cada lugar puede tener caminos hacia otros lugares.
  - El metodo `explora()` debe devolver un par con el tesoro encontrado y UN *camino* (una lista de lugares) que lleva al tesoro:

```
Pair<Object,PositionList<Lugar>> explora(Lugar inicialLugar) { ... }
```

  - Si no hay ningún tesoro alcanzable, el método devolverá `null`.
  - En otro caso, el camino debe contener al menos el lugar desde el que arranca la búsqueda.

La clase Lugar

```
public class Lugar {
  public boolean tieneTesoro()       // Devuelve true si el lugar tiene un tesoro

  public Object getTesoro()          // Devuelve el tesoro (un objeto) o null

  public Iterable<Lugar> caminos()   // Devuelve los lugares conectados
                                     // con el lugar (del objeto)

  public void marcaSueloConTiza()    // Permite marcar el ''suelo'' en el lugar
                                     // con ''tiza''

  public boolean sueloMarcadoConTiza()   // Esta marcado el suelo con tiza?

  public String toString()               // Para imprimir el lugar

  public void printLaberinto()           // Imprime todo el laberinto
}
```

Requisitos generales

- Evitad código repetido o muy similar.
- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar.
- Debe ejecutar `TesterLab4` correctamente sin mensajes de error.
- Nota: una ejecución sin mensajes de error `no` significa que el método sea correcto (es decir, que funcione bien para cada posible entrada).
- Las entregas se comprueban manualmente antes de dar la nota final.