A.E.D. Laboratorio 4

Guillermo Román
guillermo.roman@upm.es

Julio García
juliomanuel.garcia@upm.es

Lars-Åke Fredlund
lfredlund@fi.upm.es

Tonghong Li
tonghong@fi.upm.es

Manuel Carro
mcarro@fi.upm.es

Nicolás Alonso Shirra
nicolas.alonso.shirra@upm.es

Normas

- Fechas de entrega y penalización:

Hasta el martes 10 de noviembre, 16:00 horas	0 %
Hasta el miércoles 11 de noviembre, 16:00 horas	20 %
Hasta el jueves 12 de noviembre, 16:00 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`

- El/los fichero(s) que hay que subir es/son `Utils.java`.
- La clase debe estar en el paquete `aed.recursion` .
- La documentación de la API de `aedlib.jar` está disponible en `http://costa.ls.fi.upm.es/teaching/aed/docs/aedlib/`

Normas

- El tema principal de este laboratorio es practicar el uso de recursión.
- Por ese motivo está:
  - **Prohibido** usar bucles `for`, `for-each`, `do-while` o `while`.
  - **Prohibido** definir atributos adicionales en las clases a entregar.

Tareas

- El laboratorio tiene 4 partes:
  - Combinar y ordenar dos listas (un "merge") (3 puntos)
  - Calcular el valor de una proposición (3 puntos)
  - Devolver un "pico" de un array de enteros (3 puntos)
  - Comprobar que un String está "equilibrado" (3 puntos)
- Notad que la puntuacíon máxima es 12 puntos.
- Hay que entregar una soluciones que, al menos, pasan los tests de 3 de las 4 tareas

Tarea 1: ordenar los elementos de dos listas

Implementad el método

```
static PositionList<E> merge(PositionList<E> l1,
                             PositionList<E> l2,
                             Comparator<E> cmp)
```

- Las dos listas `l1` y `l2` están ordenadas según el comparador `cmp`
- El método debe devolver una lista nueva, que contenga todos los elementos devueltos por las listas, *en orden* según `cmp`
- Las listas `l1` y `l2` no son `null`, y no contienen elementos `null`
- Ejemplo: `merge([1,1,3,9],[2,3,10],new AscComp())` devuelve la lista `[1,1,2,3,3,9,10]` dado la comparador `AscComp` que ordena los enteros en el orden "normal" (ascendiente)

Comparadores usados en las pruebas de `merge`

- Notad que el Tester llama a `merge` con tres distintos comparadores de enteros:
  - `AscComp` – los enteros están ordenados en orden ascendiente (p.e. $1, 2, 3, 4$)
  - `DescComp` – los enteros están ordenados en orden descendiente (p.e. $5, 4, 3, 2, 1$)
  - `AbsComp` – los enteros están ordenados en orden ascendiente según la relación: $n_1 < n_2$ si $abs(n_1) < abs(n_2)$, o si $abs(n_1) = abs(n_2)$ y $n_1 < n_2$. Notad que p.e. $abs(-5) = 5$. Ejemplo de enteros ordenados por el comparador en orden: $0, -1, 1, 2, -3, -5, 5$.
- IMPORTANTE: vuestra implementación debe funcionar con cualquier otro comparador

Tarea 2: calcular el valor de una proposición

Implementa el método

```
boolean calculate(PropTerm t, Map<String,Boolean> env)
```

que dado:

- una proposición `t`
- y un mapa `env` que asigna valores booleanos a los variables dentro `t`

calcula si el valor de la proposición `t` es `true` o `false`.

- Si el mapa `env` no contiene como clave el nombre de una variable que ocurre dentro `t` entonces el método `calculate` debería lanzar la excepción `IllegalArgumentException`.

¿Que es una proposición?

Una proposición ϕ tiene la forma

	var	una variable
|	$\neg\phi$	negación
|	$\phi_1 \wedge \phi_2$	conjunción (and)
|	$\phi_1 \vee \phi_2$	disyunción (or)
|	$\phi_1 \rightarrow \phi_2$	implicación (implies)

- Consulta por ejemplo Wikipedia para obtener mas información

Ejemplos

Asumiendo que `env` contiene los "entries" $\langle "x", true \rangle$ y $\langle "y", false \rangle$

`t`	`calculate(t,env)`
x	*true*
y	*false*
z	lanza `IllegalArgumentException`
$x \wedge y$	*false*
$x \vee y$	*true*
$x \rightarrow y$	*false*
$x \rightarrow (x \vee y)$	*true*
$y \wedge z$	lanza `IllegalArgumentException`

Trabajar con Proposiciones

El interfaz `PropTerm` proporciona métodos para analizar y descomponer proposiciones:

```
public interface PropTerm {
  public boolean isVar();       public boolean isNeg();
  public boolean isImplies();   public boolean isAnd();
  public boolean isOr();

  public String getVar();
  public PropTerm getOperand();
  public Pair<PropTerm,PropTerm> getOperands();
}
```

- El método `getVar()` devuelve el nombre de una variable
- `getOperand()` devuelve ϕ, dado una proposición $\neg\phi$
- `getOperands` devuelve un `Pair` (ϕ_1, ϕ_2) dada una proposición $\phi_1 \wedge \phi_2$ (ó $\phi_1 \vee \phi_2$ ó $\phi_1 \rightarrow \phi_2$).

Tarea 3: implementar `findPeak`

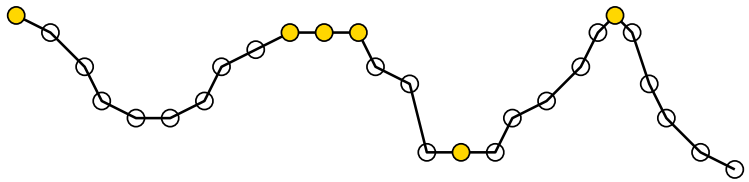
- Implementad el método `int findPeak(Integer[] arr)` que devuelve un índice en `arr` que se corresponde con un "pico"
- Un pico es un elemento de `arr` que no es menor que sus vecinos. Si i es el índice de un pico y tiene dos vecinos:

$$\texttt{arr}[i-1] \leq \texttt{arr}[i] \geq \texttt{arr}[i+1]$$

- Por ejemplo:
  - En el array `[1 2 3 4]` el elemento `4` es el único pico.
  - En el array `[4 3 2 5 5]` hay tres picos: `4` (que solo tiene un vecino), y los dos elementos `5`.
- Si hay múltiples picos, `findPeak` puede devolver el índice de cualquiera de ellos.
- Se puede asumir que el array nunca es vacío, por tanto, siempre hay al menos un pico.
- Es **obligatorio** implementar `findPeak` *eficientemente* – una búsqueda logarítmico.

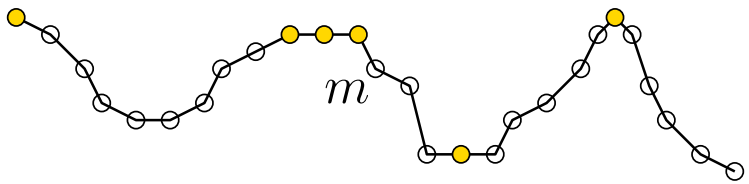
Dividiendo el problema: intuición geométrica

Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



Dividiendo el problema: intuición geométrica

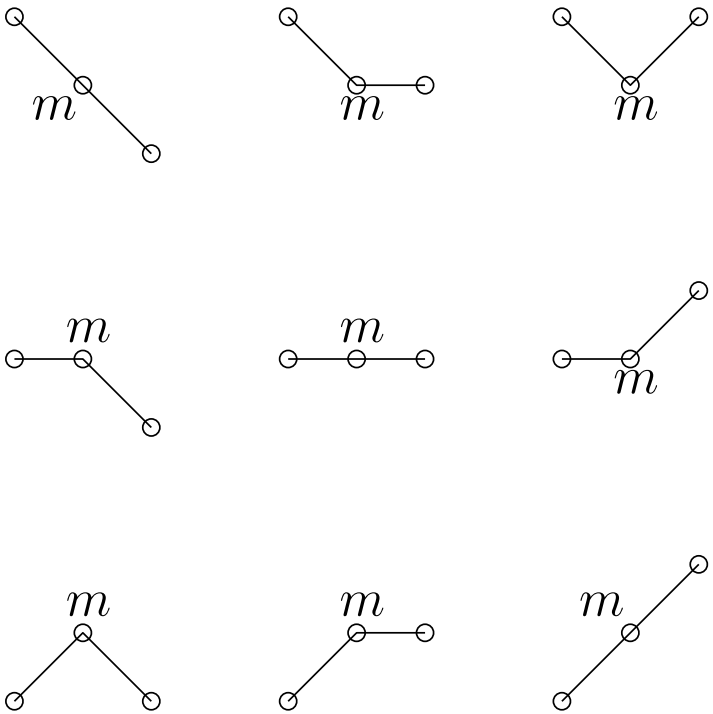
Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



- Seleccionar punto intermedio m.

Dividiendo el problema: intuición geométrica

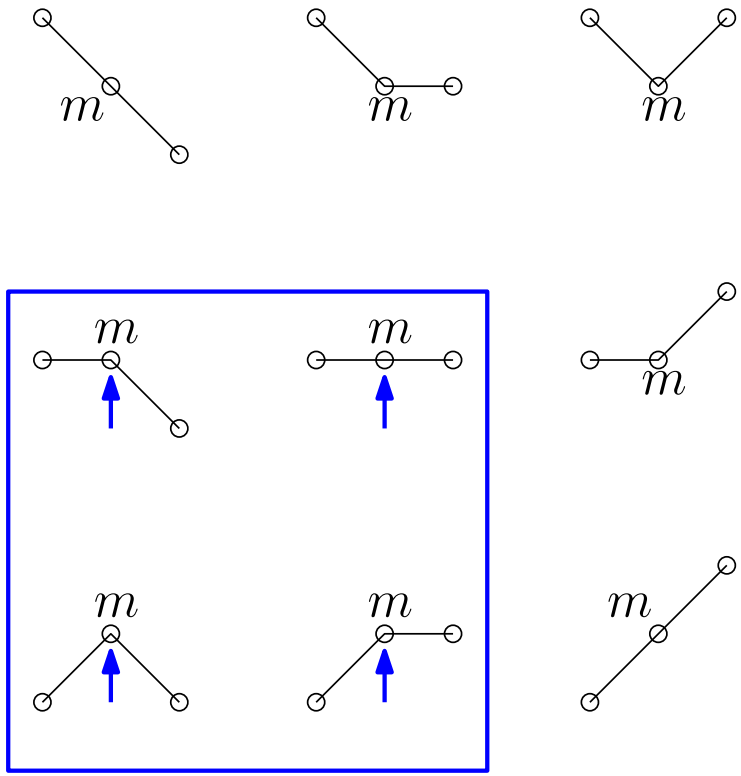
Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



- Seleccionar punto intermedio m.
- Examinar los vecinos

Dividiendo el problema: intuición geométrica

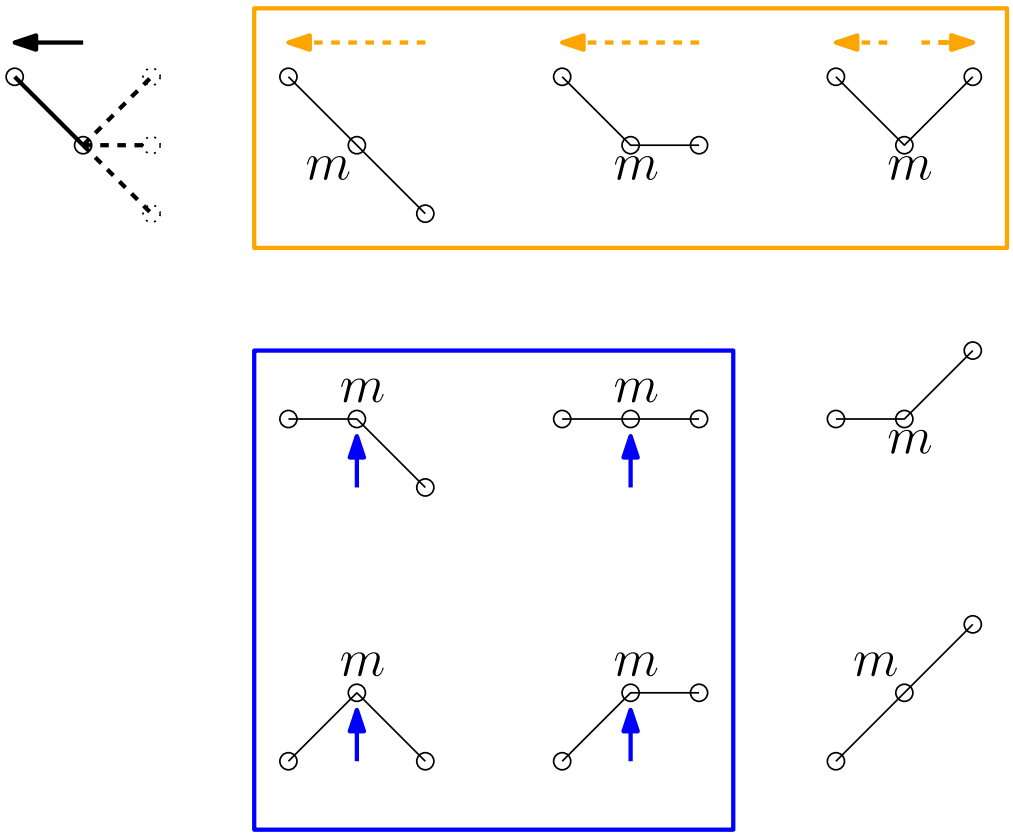
Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



- Seleccionar punto intermedio m.
- Examinar los vecinos
- Tenemos un pico $\Rightarrow$ acabado.

Dividiendo el problema: intuición geométrica

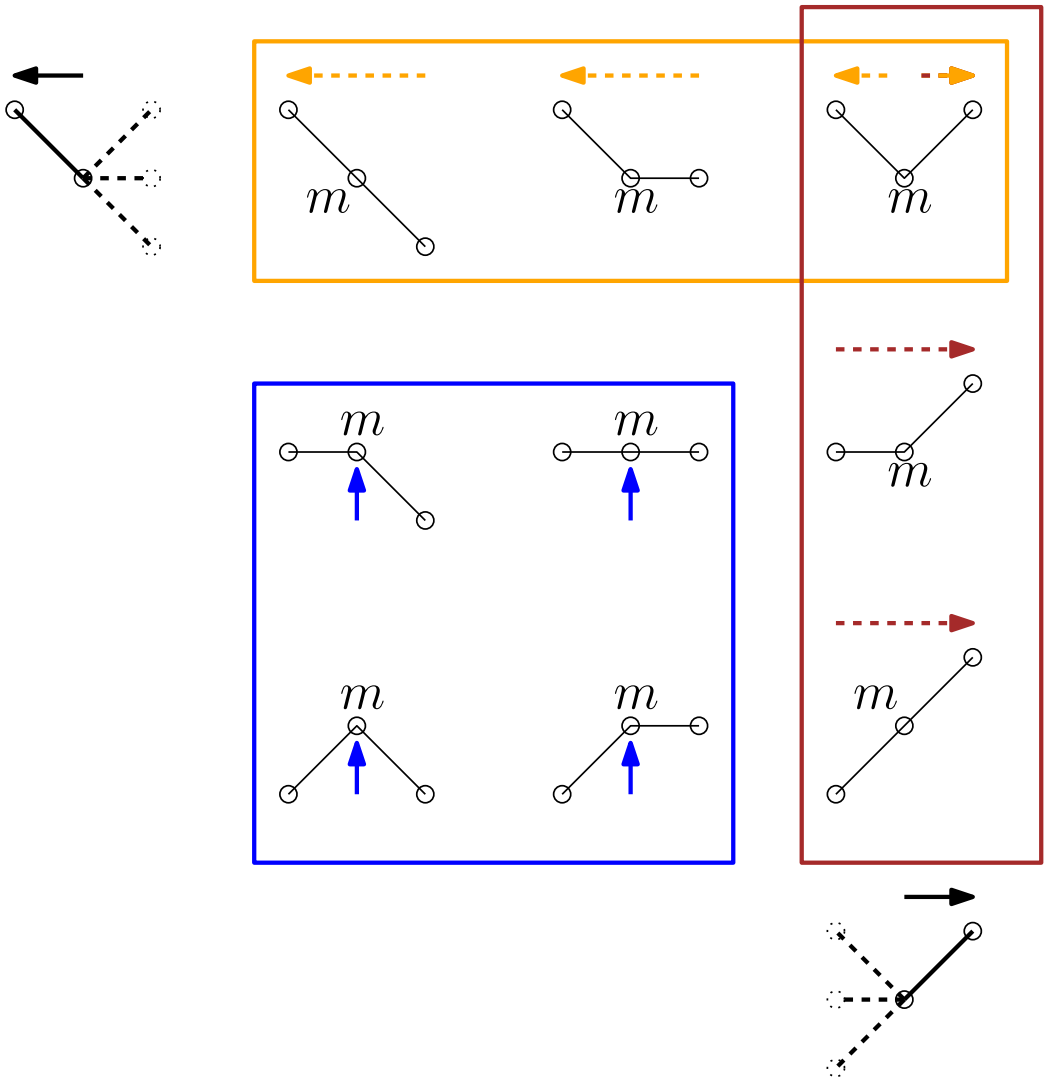
Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



- Seleccionar punto intermedio m.
- Examinar los vecinos
- Tenemos un pico $\Rightarrow$ acabado.
- Pico hacia la izquierda $\Rightarrow$ subdividir.

Dividiendo el problema: intuición geométrica

Barrido secuencial: $O(n)$. Queremos que los hagáis mucho más eficiente: $O(\log n)$.



- Seleccionar punto intermedio m.
- Examinar los vecinos
- Tenemos un pico $\Rightarrow$ acabado.
- Pico hacia la izquierda $\Rightarrow$ subdividir.
- Pico hacia la derecha $\Rightarrow$ subdividir.

Tarea 4: comprobar que un String está "equilibrado"

- Implementa: `boolean isBalanced(String s)`
  - El método comprueba que los caracteres '(', ')', '{', '}', '[' y ']' están equilibrados dentro de `s`

Ejemplo de funcionamiento:

```
isBalanced("");              // => true
isBalanced("ab");            // => true
isBalanced("(ab");           // => false
isBalanced("ab]");           // => false
isBalanced("(ab)");          // => true
isBalanced("(ab}");          // => false
isBalanced("{(a)b}");        // => true
isBalanced("[{(a)b}cd]");    // => true
isBalanced("a[b](c)d");      // => true
isBalanced("[(][)]");        // => false
```

Implementar `boolean isBalanced(String s)`

- Dado un String `s`, se puede acceder al carácter que ocupa la posición `i` dentro de `s` usando `s.charAt(i)`.
  Ejemplo: `"hola".charAt(2) => 'l'`
- Se recomienda definir algunos métodos auxiliares. Podrían ser útiles los siguientes métodos:
  - `boolean opens(char c)` que devuelve true si c es '(', '{', o '['
  - `boolean closes(char c)` que devuelve true si c es ')', '}', o ']'
  - `boolean matches(char c1, char c2)` que devuelve true si las caracteres c1 y c2 abren y cierran correctamente, como por ejemplo, si c1='{' y c2='}'.
- IMPORTANTE: La solución **no debe contener métodos auxiliares con parámetros que permitan almacenar múltiples elementos** – listas, arrays, pilas, colas, maps, ... – *usando recursión no es necesario*

Notas Generales

- El proyecto debe compilar sin errores y debe cumplirse la especificación de los métodos a completar
- Debe pasar todos los test `TesterLab4` correctamente sin mensajes de error
- **Nota:** una ejecución sin mensajes de error y que pase todas las pruebas **no** significa que la implementación sea correcta (es decir, que funcione bien para cada posible entrada)
- Todos los ejercicios se corrigen manualmente antes de dar la nota final

Evaluación

¡Seguir estos consejos os permitirá conseguir mejores resultados!

- Corrección
- Ausencia de código repetido con la misma funcionalidad (podéis usar métodos auxiliares para evitarlo)
- Concisión y claridad del código
- Legibilidad, incluida selección de nombres descriptivos para variables y métodos
- El código debe estar correctamente indentado y con comentarios *útiles* cuando lo veáis necesario
- Eficiencia:
  - Se valorará la complejidad computacional del código
  - Se valorará no iterar innecesariamente en los recorridos de las estructuras de datos