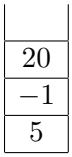
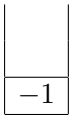
PRG - ETSInf. TEORÍA. Curso 2014-15. Recuperación.
23 de junio de 2015. Duración: 2 horas y 15 minutos.

1. 2.5 puntos Dada una `PilaIntEnla p` cualquiera, eventualmente vacía, se quiere escribir un método estático **recursivo** (se supone que en una clase distinta de `PilaIntEnla`), tal que elimine los elementos positivos de la pila, dejando el resto en el mismo orden en el que se encontraban inicialmente. El método debe retornar la suma de los elementos eliminados.

   Ejemplo:

   La pila  debe quedar  , y se debe retornar 25.

   Se pide:

   a) Perfil del método.

   b) Caso base y caso general.

   c) Implementación en Java.

Solución:

a)
```
/** Elimina los elementos positivos de p, siendo p.talla()>=0, y devuelve su suma.*/
 public static int filtrarPos(PilaIntEnla p)
```

b)
- Caso base, `p` está vacía: No hay ningún elemento que eliminar. Trivialmente, se devuelve 0 como suma de los elementos eliminados.
- Caso general, `p` no está vacía, tiene al menos un elemento: Se reduce el problema desapilando la cima de `p`. Recursivamente, se eliminan los elementos positivos de la pila reducida, obteniéndose la suma de dichos elementos. Si el elemento que se había desapilado es ≤ 0 se vuelve a apilar en `p`, y si no su valor incrementa la suma a devolver.

c)
```
public static int filtrarPos(PilaIntEnla p) {
    if (p.esVacia()) return 0;
    else {
        int x = p.desapilar();
        int sumaResult = filtrarPos(p);
        if (x <= 0) p.apilar(x)
        else sumaResult +=  x;
        return sumaResult;
    }
}
```

2. 3 puntos El siguiente método recibe un array `a` cuyas componentes son listas con PI de enteros, y busca si el entero `x` aparece en alguna de estas listas:

```
/** Busca en qué lista a[i], 0 <= i < a.length, aparece x.
 *  Si no se encuentra en ninguna, devuelve -1.
 */
public static int buscar(ListaPIIntEnla[] a,int x) {
    boolean encontrado = false;
    int i = 0;
    while (i<a.length && !encontrado) {
        ListaPIIntEnla l = a[i];
        l.inicio();
        while (!l.esFin() && l.recuperar() != x)
```

```
                l.siguiente();
            if (!l.esFin()) encontrado = true; else i++;
        }
        if (encontrado) return i; else return -1;
    }
```

Se desea analizar el coste temporal del método, teniendo en cuenta que todas las operaciones de la clase `ListaPIIntEnla` son $\Theta(1)$. Por simplificar, supondremos que todas las listas de `a` tienen `a.length` elementos.

Se pide:

a) Indicar cuál es el tamaño o talla del problema, así como la expresión que la representa.

b) Indicar si existen diferentes instancias significativas para el coste temporal del algoritmo e identificarlas si es el caso.

c) Elegir una unidad de medida para la estimación del coste (pasos de programa, instrucción crítica) y, de acuerdo con ella, obtener una expresión matemática, lo más precisa posible, del coste temporal del método, distinguiendo el coste de las instancias más significativas en caso de haberlas.

d) Expresar el resultado anterior utilizando notación asintótica.

Solución:

a) La talla es $n =$ `a.length`, que determina el volumen de elementos a tratar en el problema.

b) Se trata de un problema de búsqueda, lista a lista, y en cada lista, elemento a elemento. Por lo tanto, para una misma talla sí que presenta instancias distintas.

El caso mejor se da cuando `x` se encuentra en la primera posición de la primera lista revisada, `a[0]`.

El caso peor cuando todos los elementos de todas las listas componentes de `a` son diferentes de `x`.

c) Medida del coste tomando como unidad el paso de programa:

Todas las instrucciones que se ejecutan en el método (salvo los bucles) tienen un coste del orden de un paso de programa. Así pues, el coste del caso mejor es $T^m(n) = 1$ *p.p.* El coste del caso peor es $T^p(n) = 1 + n \cdot (1 + n) = 1 + n + n^2$ *p.p.*, dado que en tal caso el bucle más externo realiza n pasadas, todas ellas del mismo coste $1 + n$ *p.p.*

Medida del coste tomando como unidad la instrucción crítica:

Todas las instrucciones que se ejecutan en el método (salvo los bucles) tienen un coste del mismo orden constante, por lo tanto se puede tomar como instrucción crítica la evaluación de la guarda `!l.esFin() && l.recuperar()!=x`, que es la que más se repite. En el caso mejor sólo se ejecutará una vez. En el caso peor se repetirá, para cada una de las n listas, un total de $n + 1$ veces (tantas veces como elementos hay en la lista, más la evaluación final que detecta el final de la lista). La función de coste temporal, considerando la instrucción crítica de coste unitario, en el caso mejor será $T^m(n) = 1$ *i.c.* y en el caso peor $T^p(n) = n \cdot (n + 1) = n + n^2$ *i.c.*

d) En notación asintótica: $T^m(n) \in \Theta(1)$ y$T^p(n) \in \Theta(n^2)$. Por lo tanto, $T(n) \in \Omega(1)$ y $T(n) \in O(n^2)$.

3. 1.5 puntos Se define el índice de masa corporal de una persona (IMC) como su peso (kg) dividido por la altura (m) al cuadrado. Se dispone de un fichero de texto en el que cada línea contiene el dni (una cadena de caracteres), peso y altura de una persona (ambos números reales), separados por espacios en blanco.

Se pide: Implementar un método que a partir de dicho fichero, genere otro en el que aparezca, además de los datos anteriores, una cuarta columna con el IMC. Tanto el nombre del fichero origen, como el nombre que se desea para el fichero destino deberán ser parámetros (de tipo `String`) del método.

Supondremos que el formato de los datos del fichero de entrada es correcto. En el caso de que haya algún problema al abrir los ficheros, se debe escribir en la salida estándar el mensaje `"Error abriendo fichero"`.

Solución:

```
public static void imc(String fichIn,String fichOut) {
    try {
        Scanner sc = new Scanner(new File(fichIn)).useLocale(Locale.US);
        PrintWriter pw = new PrintWriter(new File(fichOut));
        while (sc.hasNext()) {
            String dni = sc.next();
            double peso = sc.nextDouble();
            double altura = sc.nextDouble();
            double imc = peso / (altura * altura);
            pw.println(dni + " " + peso + " " + altura + " " + imc);
        }
        sc.close();
        pw.close();
    } catch (FileNotFoundException e) {
        System.out.println("Error abriendo fichero");
    }
}
```

4. 3 puntos **Se pide:** En la clase `ListaPIIntEnla`, implementar un método de instancia con perfil:

```
public void borrarAtras()
```

que, con un coste como mucho lineal, permita el borrado hacia atrás, esto es, que borre el elemento anterior al punto de interés.

En caso de que la lista esté vacía o, en general, el punto de interés se encuentre al inicio, deberá lanzar la excepción `NoSuchElementException` con el mensaje `PI al inicio`.

En caso contrario, tras el borrado, el punto de interés (`PI`) permanece inalterado sobre el mismo elemento, y el anterior al punto de interés (`antPI`) retrocede una posición.

Ejemplo: Si la lista es `2 3 1 8 9` (el `PI` está sobre el 8), tras el borrado debe quedar `2 3 8 9`.

NOTA: En la solución sólo se permite acceder a los atributos de la clase, quedando terminantemente prohibido el acceso a sus métodos.

Solución:

```
/** Elimina el elemento anterior al punto de interés. Si la lista está vacía
 *  o el punto de interés está al inicio, lanza NoSuchElementException. */
public void borrarAtras() {
    if (antPI == null) throw new NoSuchElementException("PI al inicio");
    if (antPI == primero) {
        primero = antPI.siguiente;
        antPI = null;
    }
    else {
        NodoInt aux = primero;
        while (aux.siguiente != antPI)
            aux = aux.siguiente;
        aux.siguiente = antPI.siguiente;
        antPI = aux;
    }
    talla--;
}
```

Notar que el coste depende, además de la talla n de la lista, de la posición del `PI`. En el caso peor, `PI` al final de la lista, $T^p(n) \in \Theta(n)$.