Polinomios (v2)

Un polinomio es una expresión consistente en constantes (llamadas coeficientes) y variables que involucran solamente las operaciones de suma, resta, multiplicación y exponenciación con exponentes no negativos.

Ejemplos de polinomios son:

- $x^2 - 4x + 7$
- $x^3 + 2xyz^2 - yz + 1$

En el primer caso se trata de un polinomio en una variable (x) y, en el segundo, de un polinomio en tres variables (x, y, z)

En esta práctica **nos centraremos en polinomios en una variable** que, por convención, denominaremos x cuyos **coeficientes** son valores **enteros**.

Un polinomio en una variable puede expresarse siempre de la siguiente manera:

$$a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x^1 + a_0 = \sum_{i=0}^{n} a_i x^i$$

donde las a_i son constantes y x es la variable y las expresiones $a_i x^i$ se denominan términos (o monomios).

- Por tanto, un polinomio puede ser, o bien cero, o bien la suma de un número finito de términos no cero.
- Cada término consiste en el producto de una constante entera diferente de cero (coeficiente) por la variable elevada a un exponente no negativo.
    - El exponente al que se eleva la indeterminada en un término es el **grado** de ese término.
- El **grado del polinomio** se corresponde con el del término de grado mayor**.**

Por ejemplo, $x^2 - 4x + 7$ es un polinomio de grado 2, y 7 es un polinomio de grado cero (es igual a $7x^0$).

Representación expandida de un polinomio

La forma más inmediata de representar un polinomio en una variable es mediante un array en el que almacenaremos todos sus coeficientes. Para un polinomio de

grado n necesitaremos espacio para $n+1$ **coeficientes que, en nuestro casi, serán enteros**.

A esta representació la denominaremos, **representación expandida del polinomio** y colocaremos **el coeficiente** a_i **del polinomio en la posición** i **del array**.

Por ejemplo, si queremos representar $x^2 - 4x + 7$, como se trata de un polinomio de grado 2, necesitaremos un array de 3 posiciones:

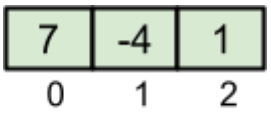


Figura 1: Representación expandida de $x^2 - 4x + 7$

En esta representación se cumple que:

- O bien el polinomio es cero, representado por un array de tamaño cero.
- O bien no lo es y el coeficiente del término de grado mayor no es cero y, por tanto, `a[a.length-1] != 0`.

¿Qué problema tiene esta representación? En polinomios en los que muchos coeficientes son cero, malgasta mucho espacio. Por ejemplo, el polinomio $x^{1000} - 1$ requiere de un vector de 1001 enteros, en los que solamente las posiciones 0 y 1000 tienen valores diferentes de 0.

Representación comprimida de un polinomio

Para resolver este problema, ya que el objetivo es no usar espacio para aquellos coeficientes que son cero, y como ya no podremos asociar el coeficiente y el grado del término al que pertenece vía la posición que ocupa en el array, tendremos que guardar, para cada término, tanto su coeficiente como su grado.

Por tanto, en vez de tener un array tendremos una matriz de dos columnas y tantas filas como coeficientes no cero tenga el polinomio (excepto para el polinomio cero que, como antes será especial, y consistirá en una matriz de cero filas y dos

columnas); en dicha matriz, **la columna 0 indicará el grado y la columna 1 el coeficiente**.

Además, de cara a simplificar las operaciones, añadiremos el requisito de que los términos (filas de la matriz) estén ordenados en orden de grado creciente.

Con esta representación, el polinomio $x^{1000} - 1$ se representa con una matriz 2x2 de enteros:

	0	1
0	0	-1
1	1000	1

Figura 2: representación comprimida de $x^{1000} - 1$

lo que es un ahorro de espacio considerable. En cambio, con esta nueva representación, necesitaremos el doble de espacio para el polinomio $x^2 - 4x + 7$.

	0	1
0	0	7
1	1	-4
2	2	1

Figura 3: representación comprimida de $x^2 - 4x + 7$

La práctica

La práctica consistirá en **implementar diferentes operaciones que trabajen sobre polinomios representados usando los dos métodos descritos anteriormente**. Como siempre, y especialmente para las operaciones complejas, además de las operaciones que se piden explícitamente en el enunciado, podéis (más bien, debéis) definir las operaciones auxiliares que creáis conveniente.

MUY IMPORTANTE:

- **Si no se dice explícitamente lo contrario, las funciones que reciben polinomios, ya sean en forma expandida como comprimida, pueden suponer que los polinomios están bien construidos.**

- **Si no se dice expresamente lo contrario, las funciones NO modifican los parámetros, arrays o matrices, que se les pasan.**

Funciones de cálculo de los tamaños

Implementaremos una función para, dado un polinomio representado de forma comprimida, calcular el tamaño del array que se necesitan para almacenarlo de forma expandida.

1. `public int expandedSize(int[][] compressed) { ?? }`

Por ejemplo, si nos dan la matriz mostrada en la figura 3, el resultado deberá ser 3.

De forma dual, implementaremos una función tal que, dado un polinomio expandido, calculará el tamaño de la matriz (su número de filas ya que siempre tendrá dos columnas) que se necesitan para representarlo de forma comprimida.

2. `public int compressedSize(int[] expanded) { ?? }`

Por ejemplo, si se aplicara al array de 1001 posiciones que representa de forma extendida el polinomio $x^{1000} - 1$, devolvería 2.

Funciones de creación

Estas dos funciones crean los arrays (o matrices) dados las tamaños que se les pasan:

3. `public int[] createExpanded(int expandedSize) { ?? }`
4. `public int[][] createCompressed(int compressedSize) { ?? }`

NOTA: Podéis suponer que, como precondición, está garantizado que el entero que reciben como parámetro es no negativo.

Funciones de copia

Estas funciones copian los elementos de un polinomio (fromXXXXX) representado de una manera, expandido o comprimido, sobre otro (toXXXXX) representado de la forma opuesta.

Ambas funciones suponen, como precondición, que el polinomio destino tiene el tamaño adecuado y, obviamente, modifican su contenido. Por tanto, en su implementación, podéis dar por seguro que eso se cumple (es responsabilidad de quien llama a la función que se cumpla)[1].

5. `public void copyTo(int[] fromExpanded, int[][] toCompressed) { ?? }`
6. `public void copyTo(int[][] fromCompressed, int[] toExpanded) { ?? }`

Funciones de conversión

Estas dos funciones convierten la representación de un polinomio. **No realizan el trabajo directamente sino que utilizan las funciones definidas anteriormente.**

7. `public int[][] compress(int[] expanded) { ?? }`
8. `public int[] expand(int[][] compressed) { ?? }`

Funciones de evaluación

Una de las operaciones que podemos realizar con un polinomio en una variable es evaluarlo para un valor dado de esa variable. Por ejemplo, el polinomio $x^2 - 4x + 7$ evaluado para $x = 2$ es $2^2 - 4 * 2 + 7 = 4 - 8 + 7 = 3$.

[1] Lo razonable en Java sería lanzar un error, pero cómo hacerlo forma parte del temario de la siguiente asignatura del plan de estudios: Estructuras de Datos.

Para evaluar los polinomios en representación comprimida, puede ser útil definir la siguiente función auxiliar (NOTA: no uséis Math.pow, que trabaja con doubles y después redondeéis).

```
9. public int pow(int base, int exponent) { ?? }
```

Tendremos, por tanto, dos funciones de evaluación, una para polinomios representados de forma expandida y otra para polinomios representados de forma comprimida.

```
10.  public int evaluate(int[] expanded, int x) { ?? }
```

De cara a evaluar el polinomio forma eficiente, os puede ser interesante la siguiente igualdad:

$$a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x^1 + a_0 = (\ldots ((((a_n)\,x) + a_{n-1})\,x + a_{n-2})\,x + \ldots.) + a_0$$

Es decir, por ejemplo:

$$x^2 - 4x + 7 = (((1) * x) + (-4) * x) + 7$$

Es el llamado algoritmo de Horner y evita el uso de una operación costosa como es la exponenciación en la evaluación del polinomio expandido.

```
11.  public int evaluate(int[][] compressed, int x) { ?? }
```

No se aceptará la solución de implementar una función de evaluación convirtiendo el primer polinomio a la otra representación y luego llamar a la otra función de evaluación. Las dos funciones de evaluación han de ser independientes.

Suma de dos polinomios

Definid también dos operaciones para calcular la suma de polinomios, tanto en representación expandida como comprimida. Tened en cuenta que, al hacer la suma, podría suceder que el resultado requiriera menos espacio que cualquiera de los polinomios que se están sumando (ya que los términos del mismo grado con coeficientes de igual valor absoluto pero signo contrario se anulan al sumar).
Por ello, para resolver el problema lo que haréis es:

- Primero calcular la suma término a término, sin tener en cuenta que el polinomio resultante podría no ser válido.
    - El resultado de esta operación puede no ser un polinomio válido
- Después, normalizar el resultado, de forma que ocupe el espacio que le corresponde.
    - El parámetro de esta operación puede no corresponderse a un polinomio válido

Dichas operaciones de normalización, a su vez, podrían descomponerse en:

- Calcular el tamaño real para el polinomio
- Crear, si es necesario, espacio para un polinomio de ese tamaño
- Copiar, si es necesario, el polinomio a normalizar sobre este polinomio

Y, finalmente, las de la suma:

```
12.  public int[] add(int[] expanded1, int[] expanded2) { ?? }
13.  public int[][] add(int[][] compressed1, int[][] compressed2) {
     ?? }
```

No se aceptará la solución de implementar una función de suma convirtiendo los polinomios de entrada a la otra representación, llamar a la otra función de suma y convertir el resultado. Las dos funciones de suma han de ser independientes.

El proyecto que os proporcionamos

Además de aprender algunos conceptos de programación orientada a objetos y el lenguaje Java, es necesario que empecéis a adquirir buenas costumbres en la manera de resolver problemas de programación.
Una de las más relevantes es la de saber probar sistemáticamente el código que realizamos. En posteriores prácticas presentaremos una biblioteca de clases que nos ayudará en la confección de estos juegos de prueba y, en esta práctica, os proporcionaremos un conjunto de funciones que ejecutarán vuestras soluciones y comprobarán que los resultados son los esperados. Es conveniente que estudiéis su código con atención, ya que puede mostraros ejemplos que os pueden resultar útiles, tanto para entender mejor el funcionamiento de las funciones que deberéis implementar, como para aprender algunos elementos de programación en Java.

Definición de constantes

Como las operaciones sobre los polinomios comprimidos han de acceder a las diferentes columnas de la matriz, cero para el grado y uno para el coeficiente, de cara a que quede claro, especialmente para nosotros cuando estamos programando, si se está accediendo a un grado o un coeficiente, se han definido las constantes

```
private static final int DEGREE = 0;
private static final int COEFFICIENT = 1;
```

para que, usándolas, quede más claro si accedemos a un grado o un coeficiente. Por ejemplo, acceder al grado del término que ocupa la posición i de la matriz sería `compressed[i][DEGREE]`, que es mucho más claro que `compressed[i][0]`, lo que ayuda a evitar errores de programación (y ayuda a que el código sea más legible).

Aunque los veremos más adelante en el tema de programación orientada a objetos, los modificadores que hemos añadido a las definición tienen el significado siguiente:

- **private**: estas constantes solamente son usables en la clase en la que están definidas
- **static**: las variables son compartidas por todas las instancias de la clase y no hay un par de variables para cada instancia
- **final**: una vez hemos asignado un valor a la variable, éste ya no puede modificarse (para variables de tipos primitivos esto las convierte en constantes) y, para comunicar este hecho, el nombre de las variables está escrito en mayúsculas (siguiendo las convenciones de nombres de Java)

La función run

La función que os entregamos es:

```
public void run() {
    testExpandedSize();
    testCompressedSize();
    testCreateExpanded();
    testCreateCompressed();
    testCopyToFromExpandedToCompressed();
    testCopyToFromCompressedToExpanded();
    testCompress();
    testExpand();
    testPow();
    testEvaluateExpanded();
    testEvaluateCompressed();
    testAddExpanded();
    testAddCompressed();
}
```

Esta función llama a todas las funciones que prueban vuestras soluciones. **Inicialmente están comentadas**, ya que el código de vuestras funciones no existe; así que, **una vez implementéis una función, deberéis descomentar la llamada a la función de test correspondiente**. De esta manera se harán llamadas a vuestro código para comprobar si el resultado es el esperado, mostrando para cada tests que se realiza, OK si ha ido bien, ERROR en caso contrario.
Las funciones que debéis implementar (y que salen en el enunciado), las tenéis implementadas de manera que, si las llamáis, se produce un error de ejecución. **Lo que debéis hacer es sustituir esa implementación por la vuestra.**

Debéis seguir el orden de implementación del enunciado y no pasar a la siguiente función hasta que las anteriores pasen si error sus pruebas. Tened el cuenta que en la implementación de una función podéis usar las funciones definidas anteriormente.

Estructura de una función de test

Cada función de `testXXX` se apoya en una función de `checkXXX` que es la que recibe los parámetros de la función que queremos comprobar y el resultado que esperamos, llama a la función que comprueba, compara el resultado obtenido con el esperado e informa del resultado de dicha comparación.
Por ejemplo, la función `check` para el caso de `compressedSize` es:

```
public void checkCompress(int[] expanded, int[][] expected) {
    int[][] compressed = compress(expanded);
    if (! matrixEquals(compressed, expected)) {
        printlnError("compress of " + stringify(expanded)
                + " returns " + stringify(compressed)
                + " but should be " + stringify(expected));
    } else {
        printlnOk("compress of " + stringify(expanded));
    }
```

```
}
```

Y la función de `testXXX` realiza diferentes pruebas llamando a la función `checkXXX`. La correspondiente a `compressedSize` es:

```
public void testCompressedSize() {
   printlnInfo("BEGIN testCompressedSize");
   checkCompressedSize(EXPANDED_ZERO, 1);
   checkCompressedSize(EXPANDED_LEFT_ZEROS, 1);
   checkCompressedSize(EXPANDED_MIDDLE_ZEROS, 3);
   checkCompressedSize(EXPANDED_NON_ZEROS, 5);
   printlnInfo("END testCompressedSize");
   printBar();
}
```

Polinomios definidos

De cara a no tener que definir polinomios en cada uno de los tests, se han definido una serie de polinomios que se usarán en varios de los tests. Por ejemplo:

```
public int[] EXPANDED_ZERO
     = new int[] {0};
public int[] EXPANDED_LEFT_ZEROS
     = new int[] {0, 0, 0, 0, 42};
public int[] EXPANDED_MIDDLE_ZEROS
     = new int[] {42, 0, 12, 0, 24};
public int[] EXPANDED_NON_ZEROS
     = new int[] {42, 25, 12, 18, 24};
```

Funciones auxiliares

También se han definido funciones auxiliares para mostrar texto en color, comparar matrices (los arrays los compararemos con una función predefinida en Java) y para convertir tanto arrays como matrices en cadenas de caracteres (para mostrar los mensajes de error).

IMPORTANTE: No podéis usar las funciones que se han añadido para hacer los tests en vuestra solución.

¿Qué debéis entregar?

Un directorio **comprimido con ZIP** (no se aceptarán RAR, 7z, etc.) que contenga:

- el proyecto IntelliJ (es decir, el directorio del proyecto).
- un informe en **PDF** (no se aceptará DOC, DOCX, ODF, etc.) que expliqui cómo habéis diseñado cada una de las funciones. Os podéis (en algunos casos diría que debéis) ayudar de diagramas, que podéis hacer a mano y después escanear. En el apéndice del enunciado tenéis ejemplos de cómo documentar funciones para usar como referencia.

Recordad que la práctica **se realiza de forma individual**.

Criterios de evaluación

Cada una de las funciones se evaluará teniendo en cuenta:

- funcionamiento correcto
- estructura del código

Las puntuación de cada función será:

- `public int expandedSize(int[][] compressed) { 2 }`
- `public int compressedSize(int[] expanded) { 2 }`

- `public int[] createExpanded(int expandedSize) { 1 }`
- `public int[][] createCompressed(int compressedSize) { 1 }`
- `public void copyTo(int[] fromExpanded, int[][] toCompressed) { 2 }`
- `public void copyTo(int[][] fromCompressed, int[] toExpanded) { 2 }`
- `public int[][] compress(int[] expanded) { 1 }`
- `public int[] expand(int[][] compressed) { 1 }`
- `public int pow(int base, int exponent) { 1 }`
- `public int evaluate(int[] expanded, int x) { 2 }`
- `public int evaluate(int[][] compressed, int x) { 2 }`
- `public int[] add(int[] expanded1, int[] expanded2) { 3 }`
- `public int[][] add(int[][] compressed1, int[][] compressed2) { 5 }`

Dónde, para cada función, el **25% de los puntos** se corresponde con la valoración de su documentación en el **informe**[2]. En dicho informe podéis, en algunos casos complejos, usar diagramas para ayudaros en las explicaciones. No hace falta que los hagáis en alguna aplicación: podéis realizarlos a mano, escanearlos e incluirlos en el documento; eso sí, aseguraos de que se vean correctamente.

Como se ha indicado, al corregir, no solamente se tendrá en cuenta que la función se ejecute correctamente, sino que su código sea entendible, los nombres de las variables y funciones auxiliares sean adecuados, esté bien indentado, etc. etc. Ello quiere decir que **se espera que no entreguéis la primera solución que habéis conseguido hacer funcionar, sino que dediquéis tiempo, de hecho, bastante tiempo, a mejorar ese código que funciona**. Disponer de un conjunto de pruebas os ayudará a detectar casos en las modificaciones que hacéis rompen alguna cosa.

En el documento Doing-it-right, en el apartado “Novell”, podéis encontrar una descripción de lo que se espera de vosotros y en el informe de evaluación de las prácticas del año pasado, podéis encontrar ejemplos de errores comunes a evitar.

[2] En el anexo tenéis ejemplos de documentación.

Anexo: Cómo documentar una función

La documentación de una función, principalmente, es una descripción de los **porqués**, de las **decisiones** que habéis tomado en su diseño. También es el lugar donde comentar elementos no evidentes del código y otras posibles soluciones que se han descartado.

Obviamente, si consideráis que un diagrama os puede ayudar (p.e. como el que se muestra en el enunciado para sugerir un posible diseño de la operación de multiplicación), podéis añadirlo sin problemas.

Ejemplo1:

Diseñad e implementad una función tal que, dado un caràcter, decida si éste es una vocal, es decir, la función con signatura:

```
public boolean isVocal(char c)
```

NOTA: No hace falta que consideréis las vocales acentuadas o con diéresis

Explicación del diseño:

- he resuelto el problema construyendo una cadena de caracteres con todas las vocales y buscando el carácter dado en dicha cadena. Si encuentro el carácter es que éste es una vocal y, si no, no.

Código:

- no hace falta que lo pongáis en la documentación, ya que está en la práctica, aunque si os ayuda a redactar el informe, no hay problema alguno.
- fijaos en que el código, en caso de ponerse, va después de la explicación de su diseño, ya que es la consecuencia de éste, de pensar la solución.
- por ello es buena práctica ir tomando notas sobre qué pensamos mientras solucionamos el problema para luego usar dichas notas como base del informe.

```
public boolean isVocal(char c) {
     String vocals = "AEIOUaeiou";
     int i = 0;
     while (i < vocalsChars.length() && vocals.charAt(i) != c) {
         i += 1;
     }
     return i < vocalsChars.length();
 }
```

Explicación de algún aspecto no evidente en el código:

- es importante resaltar la condición del bucle en la que los términos del && han de colocarse en este orden para evitar el acceso a una posición fuera del String en caso de que el caràcter dado no sea una vocal
- al salir del bucle la condición (i < vocals.length()) nos indica si c es una vocal o no

- si se sale del bucle con (i < vocals.length()) cierto, es que se ha salido porque (vocals.charAt(i) == c), es decir, que c es una vocal (la que ocupa la posición i)
- si se sale del bucle con (i < vocals.length()) falso, es que en ningún momento se ha encontrado una posición del String que fuera igual a c, por lo que c no es una vocal.

Otras posibilidades:

- otra posibilidad hubiera sido la de usar un array de caracteres pero su inicialización hubiera sido más farragosa.

```
char[] vocals = new char[] {‘A’, ‘B’, ....};
```

- también hubiera podido hacer una única disyunción en la que se va comprobando si el carácter dado es igual a cada una de las vocales. La he descartado porque me ha parecido poco elegante.

Ejemplo 2:

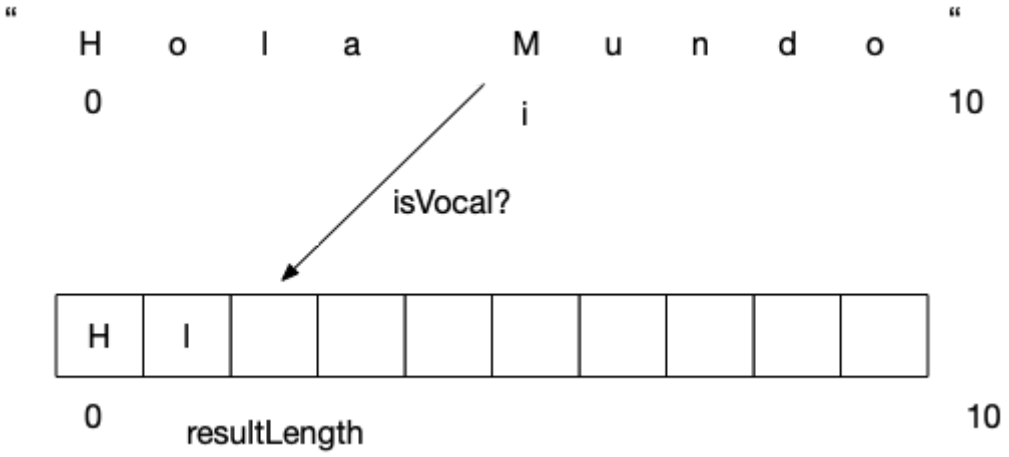
Diseñad e implementad una función tal que dado un String, retorne otro que contenga los caracteres de la cadena original que no sean vocales.

```
public String removeVocals(String str)
```

Explicación:

- se necesitará hacer un recorrido de los caracteres de la cadena original y, en caso de que el carácter no sea una vocal, añadirlo a la cadena de salida
- para añadir un carácter a la cadena de salida necesitaré un array de caracteres para ir guardándolos conforme se van encontrando
  - como la cadena de salida nunca será más larga que al de entrada, puedo usar la longitud de la cadena de entrada como tamaño del array
- también necesitaré una variable que indique el número de caracteres que he guardado en el array

Es decir, en un momento de la ejecución tenemos:



Dónde (i) es la variable que usamos para recorrer la cadena de caracteres de entrada y resultLength indica el número de no vocales que hemos encontrado hasta el momento.

Código:

```
public String removeVocals(String str) {
       char[] resultChars = new char[str.length()];
       int resultLength = 0;
       for (int i = 0; i < str.length(); i++) {
           char current = str.charAt(i);
           if (!isVocal(current)) {
               resultChars[resultLength] = current;
               resultLength += 1;
           }
       }
       return new String(resultChars, 0, resultLength);
}
```

Otras posibilidades:

- Se podría hacer un primer recorrido del String original para calcular el número de no vocales que contiene para saber exactamente la dimensión del array en el que guardarlos. Hemos preferido la solución presentada porque solamente realiza un recorrido de la cadena, pero si hay muchas más vocales que no vocales, podría tener sentido hacerlo así.