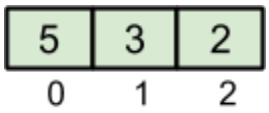
Representación de los números por vectores (v2)

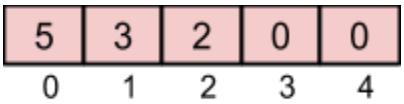
Los números enteros que la plataforma Java permite representar por tipos primitivos, como hemos visto, tienen una longitud máxima de 64 bits. Por ello, si nos ceñimos a los números naturales, el rango de valores posibles es desde 0 hasta 2^{63}-1.
El **objetivo** de esta práctica es **definir un conjunto de funciones que permiten trabajar con números naturales de "tamaño ilimitado"** (bueno, limitado por la cantidad de memoria que tengamos en la máquina).
Para ello representaremos un número natural por el vector que forman sus dígitos. Además, como en la representaciones de números que usamos en la vida real, nuestros vectores solamente contendrán las cifras **realmente significativas** del número que estamos representando.
Tomemos como ejemplo el número 235. El array que lo representa será:



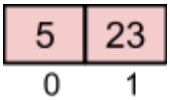
Fijaos en las siguientes propiedades:

- El número queda escrito "al revés", ya que las unidades están en la posición 0 del vector, las decenas en la 1, y las centenas en la 2.
- El número de elementos del array coincide con el de cifras que tiene 235 que son 3.

Por ello, la siguiente representación sería inválida:



ya que en el vector tenemos ceros en posiciones no significativas del número (sería el equivalente a escribir 00235).
Otro caso inválido sería:



ya que en una de las posiciones tenemos un número que no se corresponde con un dígito válido (entre 0 y 9, inclusivamente).

Fijaos en que, en esta representación, el valor 0 se corresponde con un array de una posición que contiene el valor 0 (su único dígito significativo).

Todas las funciones que haremos supondrán que los arrays que reciben están bien construidos y han de devolver arrays bien construidos. Es decir:

- todas sus posiciones contienen dígitos válidos
- no contiene dígitos no significativos

Además, aunque Java permite que modifiquemos dentro de una función el array que nos pasen como parámetro (ya vimos en clase que *los parámetros que se corresponden con arrays se pasan por referencia*), **en ninguna función modificaremos el parámetro pasado** sino que siempre devolveremos arrays "nuevos" creados por nuestra función. Sí, de entrada parece "más complicado" pero, como veréis, usar las funciones así definidas será "mucho más fácil".

Las funciones zero y one

```
public int[] zero() { ¿? }
public int[] one()  { ¿? }
```

Estas dos funciones son tan simples como parece: la primera devuelve el vector que se corresponde con el número 0; y, la segunda, el que se corresponde con el número 1.

La función de comparación equals

```
public boolean equals(int[] number1, int[] number2) { ¿? }
```

Esta función compara dos vectores de enteros (que son representaciones válidas de dos números) y devuelve un booleano indicando si son iguales o no.
En su implementación solamente podemos usar el acceso a la longitud de los arrays, y el acceso a cada una de sus posiciones (es decir, **no es válido el uso de funciones definidas en la biblioteca estándar como Arrays.equals**).

La suma de dos números

```
public int[] add(int[] number1, int[] number2) { ¿? }
```

Esta es una de las funciones que os generará alguna complicación, ya que hasta que no realicéis la suma, no sabréis exactamente cuántas posiciones necesitaréis. Por ejemplo: al sumar 235 + 683 = 918, necesitáis un vector con tres posiciones; pero al sumar 235 + 783 = 1018, necesitáis cuatro.

PISTA: Existen dos formas de resolver este caso:

- reservar el máximo de posiciones que harán falta y, si después de hacer la suma vemos que nos hemos pasado, copiar la parte significativa del resultado en un vector de las dimensiones adecuadas.
- reservar lo mínimo necesario y, si al realizar la suma vemos que nos hemos quedado cortos, crear un array del tamaño adecuado y copiar la parte ya sumada y añadir el dígito que falta.
- fijaos en que con las tres funciones descritas anteriormente podemos escribir código como el que sigue:

  ```
  if ( equals(number, zero()) ) return one();
  ```

  que sería la versión, usando arrays, del código:

  ```
  if ( number == 0 ) return 1;
  ```

El desplazamiento hacia la izquierda

```
public int[] shiftLeft(int[] number, int positions) { ¿? }
```

En esta función tenemos la garantía de que `positions >= 0`.

Esta función desplaza el número hacia la izquierda tantas posiciones como indica el segundo parámetro, añadiendo ceros.

Por ejemplo, dada la representación del número 235:

- si `positions` es 2, el resultado es la representación del número 23500
- si `positions` es 0, el resultado es la representación del número 235

NOTA: Si el número que nos dan es la representación del número 0, el resultado será la representación del número 0, independientemente del valor de `positions`.

Multiplicación de un número por un dígito

```
public int[] multiplyByDigit(int[] number, int digit) { ¿? }
```

Esta función calcula el número resultante de multiplicar el array que representa el número por el dígito. Tened en cuenta que tenemos garantizado que todos los datos son correctos, es decir:

- number es un vector válido
- 0 <= digit <= 9

Esta operación tiene los mismos problemas que la suma respecto el tamaño de array que se requiere para representar el resultado. Por ejemplo, 345 * 2 = 690 y 345 * 3 = 1035.

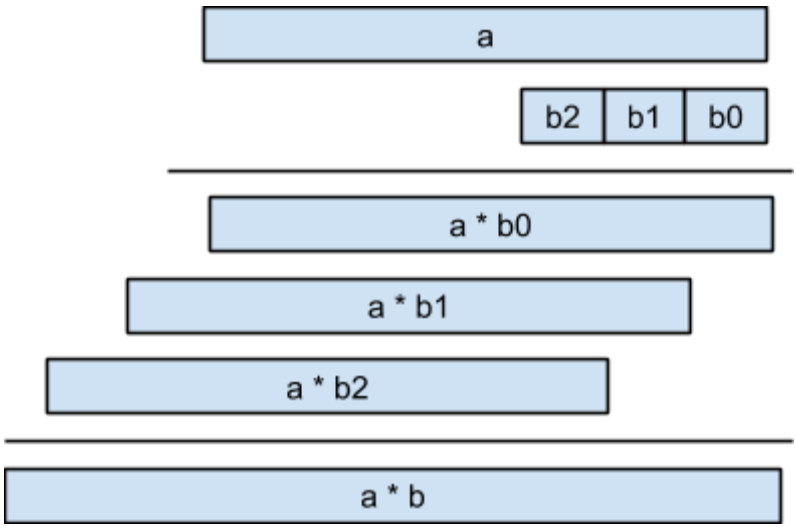
PISTA: Puede ser útil tratar por separado el caso cuando o bien el número representado por `number` o bien `digit` son 0.

Multiplicación de dos números

```
public int[] multiply(int[] a, int[] b]) { ¿? }
```

Esta función utilizará algunas de las funciones definidas anteriormente. Para entender qué funciones utilizará y cómo, es conveniente recordar cómo realizamos la multiplicación de dos números “a mano”.

Supongamos que queremos multiplicar dos números a y b, donde b tiene tres dígitos. Para ello haríamos lo siguiente[1]:



Es decir, las funciones involucradas son:

- sumar dos números
- multiplicar un número natural por un dígito (por ejemplo, a * b0)
- desplazar un número hacia la izquierda (añadiendo ceros por la derecha).

que son, precisamente, las funciones que hemos definido anteriormente (además de la que nos permite obtener el número 0, para inicializar el acumulador de las sumas parciales).

El factorial

```
public int[] factorial(int[] number) { ¿? }
```

En este punto podemos implementar la función que calcula el factorial de un número. El factorial consiste en el producto de todos los números naturales positivos iguales o menores que el número dado.
Es decir:

- 5! = 1 * 2 * 3 * 4 * 5
- 0! = 1

PISTA: Tratad el caso del cero como un caso particular y luego haced un bucle para tratar los números a partir del 1. Recordad que las operaciones que tenéis definidas sobre lo números son:

- zero
- one
- add
- equals
- multiply

[1] He dibujado el número en el sentido habitual, es decir, con las unidades a la derecha. Recordad que en los vectores el sentido es el contrario y dibujamos la posición 0, que en nuestra representación se corresponde con las unidades, a la izquierda.

El proyecto que os entregamos para completar

Además de aprender conceptos de programación y el lenguaje Java, es bueno que empecemos a adquirir buenas costumbres que nos ayuden a poner en práctica los conocimientos adquiridos.
Una de las cosas que debemos aprender es a probar sistemáticamente el código que realizamos. En prácticas posteriores veremos herramientas que nos ayudan a hacerlo, pero, de momento, usaremos los conocimientos de programación que tenemos.
De cara a que podáis probar la práctica, en el proyecto que os entregamos, hemos añadido un conjunto de funciones que ponen a prueba las funciones que debéis implementar.

La función run

La función run que os entregamos es la siguiente:

```
public void run() {
    testFromString();
    testAsString();
    //testZero();
    //testOne();
    //testEquals();
    //testAdd();
    //testShiftLeft();
    //testMultByDigit();
    //testMultiply();
    //testFactorial();
}
```

Como véis, llama a unas cuantas funciones auxiliares que, como su nombre indica, comprueban diferentes funciones. Inicialmente sólo están habilitadas las dos primeras, ya que son dos funciones auxiliares que os damos y que os ayudan a realizar tests y a mostrar los resultados de vuestras funciones.

La función fromString

La función fromString permite construir la representación de un número (un vector de dígitos) a partir de una String. Es decir:

- fromString(“235”) -> {5, 3, 2}
- fromString(“0”) -> {0}

La función asString

Es la inversa de la anterior y, dado un vector de dígitos que representa un número, devuelve la cadena de caracteres (en el orden ‘correcto’ para ser escrito). Es decir:

- asString(new int[] {5, 3, 2}) -> “235”
- asString(new int[] {0}) -> “0”

La función testFromString

Esta función contiene tests sobre la función fromString. Escribe en la consola al iniciar y acabar los tests y, en caso de que alguno falle, indica el error.

```
    private void testFromString() {
        println("Inicio de las pruebas de fromString");
        if ( ! Arrays.equals(new int[] {5, 2}, fromString("25")) ) {
            println("Error en el caso \"25\"");
        }
        if ( ! Arrays.equals(new int[] {1}, fromString("1")) ) {
            println("Error en el caso \"1\"");
        }
        println("Final de las pruebas de fromString");
    }
```

Para comprobar que la llamada a fromString es correcta, se compara el resultado de la función con el valor previsto. La comparación se realiza con la función predefinida Arrays.equals para que no dependa de si habéis implementado correctamente la función equals o no.
La función testAsString es similar.

Las funciones de prueba para vuestro código

Cada vez que implementéis una nueva función (por ejemplo, add), **sustituiréis la línea del `throw`** en el fichero del proyecto

```
    public int[] add(int[] num1, int[] num2) {
        throw new UnsupportedOperationException("Not implemented yet");
    }
```

por vuestra solución. Después, **descomentaréis su prueba asociada en la función run y las funciones de test asociadas**.
Por ejemplo, en el caso de la función add debéis descomentar la línea

```
  //testAdd()
```

del main, y las funciones:

```
  private boolean checkAdd(String number1, String number2, String result) {
    return Arrays.equals(add(fromString(number1), fromString(number2)),
                         fromString(result));
  }

    private void testAdd() {
        println("Inicio de las pruebas de add");
```

```
        if ( ! checkAdd("1", "1", "2") ) {
            println("Error en la suma 1 + 1 = 2");
        }
        if ( ! checkAdd("5", "5", "10") ) {
            println("Error en la suma 5 + 5 = 10");
        }
        if ( ! checkAdd("99", "999", "1098") ) {
            println("Error en la suma 99 + 999 = 1098");
        }
        if ( ! checkAdd("999", "99", "1098") ) {
            println("Error en la suma 999 + 99 = 1098");
        }
        if ( ! checkAdd("5", "0", "5") ) {
            println("Error en la suma 5 + 0 = 5");
        }
        println("Final de las pruebas de add");
    }
```

La primera de ellas es una función auxiliar y la segunda la que contiene los tests. Si quisiéramos añadir un nuevo test, añadimos un nuevo if a la función para el caso dado.

Escribir el resultado de una llamada concreta a una función

Si solamente queremos escribir el resultado de una llamada a una función, podemos hacerlo de la siguiente manera:

```
    println(asString(add(fromString(“12”), fromString(“54”))));
```

que debería escribir 66 en la consola.

Lo que debéis entregar

Debéis entregar el directorio que contiene el proyecto IntelliJ con vuestra solución (comprimido con **ZIP**) y un informe en **PDF** que explique (p.e. usando diagramas cuando sea necesario) cómo habéis diseñado vuestra solución. En el anexo encontraréis ejemplos de cómo documentar una función.

Para los diagramas, simplemente podéis escanear los que habéis hecho a mano antes de poneros a programar.

La práctica se realiza de forma individual. Las prácticas detectadas como copiadas o no realizadas por el alumno (y aquí se incluyen todo tipo de academias o similares), tendrán una calificación de 0 y no serán recuperables.

Una vez hayáis acabado la práctica, debéis mirar vuestra solución y considerar si puede mejorarse. Nos hemos de acostumbrar a que lo primero que obtenemos no tiene por qué ser la

mejor solución. Hemos de arreglar el código para que se entienda, buscando los nombres más adecuados para las variables y funciones auxiliares.

Podéis añadir más casos de prueba para tener más certeza de que vuestra práctica se comporta correctamente.

Si aún así os quedara tiempo, podéis intentar buscar soluciones alternativas a la suma, la multiplicación, etc., e intentar pensar si son “mejores” que las que teníais hasta ahora.

Si aún queréis más, podéis ir más allá del enunciado y plantearnos, por ejemplo, el dividir un número representado por un int[] por un dígito. Y luego, si aún queremos intentar más cosas, dividir dos números, etc.

La nota de la práctica solamente considerará la calidad de la solución al enunciado propuesto, por lo que las posibles ampliaciones solamente podrán tener efecto a la hora de decidir posibles matrículas y, lo que es más importante, a la hora de aprender más.

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 caràcter 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.
- es por ello que 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 caràcter 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

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.