Los tipos integrales con signo (v1)

En Java, los diferentes tipos que representan números enteros se denominan tipos integrales. Se diferencian entre sí en el número de bits que utilizan para representar un número entero y en si se tiene en cuenta el signo del número entero.

Si queréis una descripción detallada de cómo funcionan, podéis consultar la especificación del lenguaje Java, que podéis encontrar en:

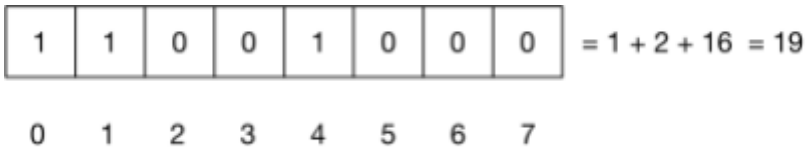
The Java Language Specification (Java 21 Edition)

En esta práctica, para simplificar, nos centraremos en los **tipos integrales con signo**: `byte`, `short`, `int` y `long` cuyos tamaños son 8, 16, 32 y 64 bits, respectivamente.

Por ello, en el esqueleto de la solución que os hemos proporcionado se tiene:

```
public int BYTE_SIZE = 8;
public int SHORT_SIZE = 16;
public int INTEGER_SIZE = 32;
public int LONG_SIZE = 64;
```

Para representar estos tipos integrales usaremos **arrays de enteros** en los que, en cada posición, tendremos el valor de un bit, es decir, arrays de ceros o unos. El bit menos significativo estará en la posición 0 del array, es decir:



se corresponde con el número binario 10011. Fijaos en que el número queda “invertido”, ya que al escribir los números, su dígito menos significativo queda a la derecha pero, al dibujar un array, solemos colocar la posición 0 a la izquierda.

En el proyecto os proporcionamos funciones para crear ceros (todos los bits a cero) de integrales de diferentes longitudes:

```
    public int[] zeroOfSize(int size) { return new int[size]; }

    public int[] zeroByte() { return zeroOfSize(BYTE_SIZE); }

    public int[] zeroShort() { return zeroOfSize(SHORT_SIZE); }

    public int[] zeroInteger() { return zeroOfSize(INTEGER_SIZE);
}
```

```
public int[] zeroLong() { return zeroOfSize(LONG_SIZE); }
```

Para simplificar las pruebas, también os proporcionamos una función para convertir un int[] (que solamente contiene ceros y unos) en un String, de manera que se pueda leer cómodamente (es decir, de izquierda a derecha):

```
public String toString(int[] num) {
    char[] chars = new char[num.length + num.length / 8];
    int iChars = 0;
    for (int iNum = num.length - 1; iNum >= 0; iNum--) {
        chars[iChars] = (char) ('0' + num[iNum]);
        iChars = iChars + 1;
        if (iNum != 0 && iNum % 8 == 0) {
            chars[iChars] = ' ';
            iChars = iChars + 1;
        }
    }
    return new String(chars, 0, iChars);
}
```

que, si la aplicamos sobre el array dado, devuelve: “00010011”

Si no comprendéis exactamente cómo funciona, no os preocupéis; más adelante detallaremos su diseño.

Funciones de test

En el proyecto os incluimos un conjunto de funciones de test que comprueban si las funciones que habéis implementado calculan correctamente sus resultados para algunos valores de sus parámetros. **Es muy importante que os acostumbréis a probar exhaustivamente vuestros programas**. Por ello, es muy interesante que estudiéis detenidamente el código de pruebas que se os proporciona, ya que os puede servir de inspiración para hacer pruebas en otros contextos.

En general, un conjunto de test nunca podrá garantizar que la función sea correcta al 100%. Pero cuantos más test diferentes se realicen, más seguridad tendremos en que la función no contiene errores. Eso sí, cuando una función no pasa algún test sabemos con certeza (si el test está correctamente diseñado) que no funciona.

Tened en cuenta que no solamente es importante que la función calcule correctamente sus resultados: el código ha de ser inteligible, los nombres de las variables han de ser adecuados, etc, etc. Tener test facilita escribir código entendible ya que, una vez hemos conseguido un código que funcione correctamente, podemos arreglarlo y disponemos de test que nos permiten comprobar, como siempre hasta cierto punto, que no hemos roto nada.

Hacer que la función calcule correctamente lo tenéis que considerar casi el punto de partida de vuestra solución, más que el punto de llegada. Es decir: no debéis entregar el primer código que habéis conseguido hacer que funcione, sino un código que funcione y que esté expresado de la mejor manera posible. Para poder hacer esto es indispensable disponer de test que comprueben que no rompemos nada al mejorar el código de nuestra solución.

La clase principal

Se ha optado por hacer que la clase que contiene todo el código sea extensión de `CommandLineProgram`. De esta manera, cuando lo ejecutemos, el resultado nos aparecerá en la ventana Run del IntelliJ. Además, las líneas correspondientes a pruebas correctas se escribirán en verde y las correspondientes a errores en rojo.

Estructura de la función `run`

La función `run` es la que ejecuta todas las funciones de prueba (una para cada apartado):

```
    public void run() {
        testToString();
//        testAllBits();
//        testCopy();
//        testNarrow();
//        testWiden();
//        testCast();
//        testAnd();
//        testOr();
//        testLeftShift();
//        testUnsignedRightShift();
//        testSignedRightShift();
    }
```

La única llamada que no está comentada es la que se corresponde con código que os proporcionamos nosotros. **A medida que vayáis completando vuestra solución de la práctica, iréis descomentando las llamadas correspondientes para ejecutar las pruebas correspondientes a cada función.**

Código correspondiente a cada apartado

En el proyecto se os proporciona la cabecera de cada una de las funciones que debéis implementar en cada apartado. **Podéis añadir las funciones auxiliares que queráis, pero no podéis cambiar las cabeceras de las funciones que se os piden y que se describen en este enunciado.**

Todas las funciones tienen la misma implementación, por ejemplo:

```
    public boolean allBits(int[] num) {
        throw new UnsupportedOperationException("apartado 1");
    }
```

Lo que debéis hacer es **eliminar la única línea de código y sustituirla por vuestra implementación.** Lo único que hace esta línea es dejar contento al compilador (el código Java es válido) pero da un error de ejecución cuando se la llama.

Estructura de una función de test

Cada función de test se corresponde con una función y prueba su funcionamiento para algunos valores de los parámetros. Por ejemplo, la correspondiente a la función toString sobre vectores de ceros de diferentes longitudes es:

```
public void testToString() {
    checkToString("00000000",
            "toString(zeroByte())",
            zeroByte());
    checkToString("00000000 00000000",
            "toString(zeroShort())",
            zeroShort());
    checkToString("00000000 00000000 00000000 00000000",
            "toString(zeroInteger())",
            zeroInteger());
    checkToString("00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000",
            "toString(zeroLong())",
            zeroLong());
    checkToString("10 11001010",
            "toString(new int[]{0, 1, 0, 1, 0, 0, 1, 1, 0, 1})",
            new int[]{0, 1, 0, 1, 0, 0, 1, 1, 0, 1});
    printBar();
}
```

Esta función realiza cuatro pruebas comprobando si las cadenas que obtenemos al llamar a `toString` sobre integrales de diferentes longitudes (`byte`, `short`, `int` o `long`) se corresponden con las cadenas dadas.

Por supuesto que podéis añadir nuevas pruebas para aumentar la confianza en vuestras soluciones. En caso de hacerlo, añadidas al final del método run y comentadas en el informe.

Función de checking

La función de checking que hace esto es la siguiente:

```
public void checkToString(
      String expected,
      String expression,
      int[] num) {

   String actual = toString(num);
   if (expected.equals(actual)) {
       println0k(expression);
   } else {
       printlnError("\"" + expression + "\" should be \"" +
expected + "\" but is \"" + actual + "\"");
   }
}
```

Todas las funciones de `checkXXXX` tienen una estructura similar. En este caso, la función recibe tres parámetros:

- el resultado que se espera obtener
- una cadena que representa lo que se está comprobando (que aparecerá en el mensaje que se mostrará al usuario)
- el número que, en este caso, se pasará a `toString`

La función llama a `toString` y obtiene el resultado real (variable actual) y comprueba si éste es igual o no al resultado esperado. En caso de que sea así, se muestra el mensaje de OK. En caso de no ser iguales, se muestra el mensaje de error indicando la expresión, el valor esperado y el valor obtenido.

NOTA: En la siguiente práctica, usaremos un framework de test que simplificará sobremanera la programación de test.

Ejecución de los test

Si ejecutamos el programa, se muestra:

```
OK: toString(zeroByte())
OK: toString(zeroShort())
OK: toString(zeroInteger())
OK: toString(zeroLong())
OK: toString(new int[]{0, 1, 0, 1, 0, 0, 1, 1, 0, 1})
```

Si hubiera algún error, se mostraría:

```
OK: toString(zeroByte())
ERROR: "toString(zeroShort())" should be "00000000 00000000" but is "0000
00000000"
OK: toString(zeroInteger())
```

```
OK: toString(zeroLong())
OK: toString(new int[]{0, 1, 0, 1, 0, 0, 1, 1, 0, 1})
```

Como vemos, parece ser que en vez de crear un integral de 16 bits lo hemos creado de 12. Para que se lean mejor los números binarios, la función `toString` añade espacios cada 8 bits, contando desde el bit menos significativo (el de más a la derecha).

Conforme vayáis avanzando en la práctica, es conveniente que estudiéis el código de prueba correspondiente a la función del apartado a resolver para entender el porqué de los resultados que muestran las pruebas para comprender mejor la función que debéis implementar.

En las siguientes secciones y apartados os describiremos el funcionamiento que han de tener las funciones que debéis implementar.

Funciones básicas

Apartado 1: `boolean allBits(int[] num)`

La primera función que se os pide ha de comprobar que, dado un array de enteros, éstos solamente son o bien cero o bien uno.

Una vez implementada, ya podréis utilizar las siguientes funciones:

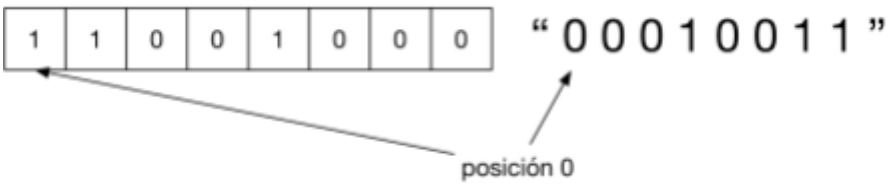
```
public boolean hasSize(int[] num, int size) {
   return num.length == size;
}

public boolean isByte(int[] num) {
   return hasSize(num, BYTE_SIZE) && allBits(num);
}

public boolean isShort(int[] num) {
   return hasSize(num, SHORT_SIZE) && allBits(num);
}

public boolean isInteger(int[] num) {
   return hasSize(num, INTEGER_SIZE) && allBits(num);
}

public boolean isLong(int[] num) {
   return hasSize(num, LONG_SIZE) && allBits(num);
}
```

Apartado 2: `void copy(String from, int[] to)`

Esta función copia los bits que hay en la cadena `from` en el array `to`, teniendo en cuenta lo siguiente:

- la función deja de copiar tanto si se ha acabado la cadena como si ya no quedan más posiciones libres en el vector.
- solamente se copian los bits que son '`0`' o '`1`' en la cadena. Si hay algún carácter diferente, éste se ignora.
- las posiciones `0` del array y de la cadena son diferentes: en el array la posición cero se corresponde con el bit de menor peso; en la cadena, con el de mayor peso.

Para entender este último punto, considerad la siguiente figura:



Una vez creada la función copy, podremos usarla para crear valores de los diferentes tipos integrales existentes en Java:

```
public int[] newByte(String from) {
    int[] num = zeroByte();
    copy(from, num);
    return num;
}

public int[] newShort(String from) {
    int[] num = zeroShort();
    copy(from, num);
    return num;
}

public int[] newInteger(String from) {
    int[] num = zeroInteger();
    copy(from, num);
    return num;
}

public int[] newLong(String from) {
    int[] num = zeroLong();
    copy(from, num);
    return num;
```

```
}
```

Conversiones

Apartado 3: `int[] narrow(int[] num, int toLength)`

Una de las operaciones que podemos realizar entre tipos integrales son las conversiones. En este apartado nos centraremos en las conversiones que se realizan hacia un tipo con el mismo o menor número de bits, es decir, dentro de la función narrow tendremos garantizada la precondición:

```
0 <= toLength <= num.length
```

Lo que sucede en este caso es que se pierden los bits más significativos del integral original.

En el caso de convertir al mismo número de bits se obtiene una copia (no se devuelve el original).

Apartado 4: `int[] widen(int[] num, int toLength)`

Vamos a tratar el caso opuesto al del apartado anterior: cuando la conversión se realiza hacia un tipo que tiene un mayor número de bits. Podría parecer que este caso es igual al anterior y que solamente debemos de copiar los bits existentes en el resultado. Si lo hacemos, nos olvidamos de un elemento importante: para que se mantenga el valor numérico, hemos de rellenar los bits de mayor peso del resultado, con el bit de signo del valor integral original.

Por ejemplo, dado el número “`0101`” si lo queremos expresar en 6 bits, su valor será “`000101`”. En ambos casos su valor numérico se corresponde con el número 5. ¿Cuál es la representación binaria del número -5? Usando complemento a 2, partimos de 5 (`0101`), cambiamos ceros por unos y unos por ceros (`1010`) y sumamos 1 (`1011`). ¿Qué pasaría si no rellenáramos los bits más significativos con el bit de signo que ahora es 1? Pues que al transformarlo a seis bits obtendríamos `001011` que se corresponde con el número 11, no con -5. En cambio, si rellenamos con 1s, obtenemos `111011` que es -5 expresado con 6 bits.

Ahora la precondición que podéis suponer es:

```
0 <= from.length <= toLength
```

y, como en la función anterior, **si se pide la misma longitud que la original, debemos retornar una copia, no el array original.**

Apartado 5: `int[] cast(int[] from, int toLength)`

Esta es muy fácil: dependiendo del tamaño de `from` y del tamaño deseado, haced `narrow` o `widen` según corresponda. En ningún caso, devolved el array original.
Una vez realizada esta función, ya podréis convertir entre cualquier tipo integral con las funciones:

```
public int[] toByte(int[] from) {
   return cast(from, BYTE_SIZE);
}

public int[] toShort(int[] from) {
   return cast(from, SHORT_SIZE);
}

public int[] toInteger(int[] from) {
   return cast(from, INTEGER_SIZE);
}

public int[] toLong(int[] from) {
   return cast(from, LONG_SIZE);
}
```

Operaciones booleanas

Apartado 6: `int[] and(int[] arg1, int[] arg2)`

La función que calcula el `and` de dos valores integrales bit a bit sería muy simple si no fuera por la letra pequeña:

- si ambos parámetros pueden representarse en 32 bits (`byte`, `short` o `integer`), se transforman ambos a integer y se realiza en `and` booleano posición a posición (el resultado tiene 32 bits)
- si al menos uno de ellos necesita 64 bits para representarse, se transforman ambos a long y se realiza un and booleano posición a posición (el resultado tiene 64 bits)

Apartado 7: `int[] or(int[] arg1, int[] arg2)`

La función que calcula el `or` de dos valores integrales bit a bit tiene la misma letra pequeña que la detallada en el apartado anterior pero, obviamente, realiza el or booleano posición a posición.

Operaciones de desplazamiento

Apartado 8: `int[] leftShift(int[] num, int numPos)`

Esta operación desplaza los bits del número `numPos` (>= 0) posiciones hacia la izquierda (hacia las posiciones más significativas del número). Las posiciones de la derecha se rellenan con ceros. Como siempre, la letra pequeña es importante:

- si el valor es representable como entero (32 bits), se transforma `num` a entero y se desplazan tantas posiciones como los 5 bits menos significativos de `numPos`, es decir, un número entre 0 y 31 (para obtenerlo podéis usar el operador `%`). El resultado obtenido es un número entero (32 bits).
- si el valor no es representable como entero (es un `long`), se desplazan tantas posiciones como los 6 bits menos significativos de numPos, es decir, un número entre 0 y 63. El resultado obtenido es un `long`.
- como se ha indicado, en ambos casos, las posiciones de la derecha se rellenan con ceros.

Apartado 9: `int[] unsignedRightShift(int[] num, int numPos)`

Esta función es como la anterior, pero desplazando en el sentido contrario, es decir, hacia la derecha (hacia los bits menos significativos). Las posiciones de la izquierda se rellenan siempre con ceros (por ello se dice que es `unsigned`, ya que no se trata el bit de signo de forma especial).

Apartado 10: `int[] signedRightShift(int[] num, int numPos)`

Finalmente, el último desplazamiento, será también hacia al derecha pero el valor con el que se rellenan las posiciones de la izquierda se corresponde con el bit de signo que tenía el número original (una vez promocionado a int o long según corresponda).

El informe de la práctica

El informe de la práctica constituye el **40% de la nota total**. Tendrá la siguiente estructura:

- **Portada** con el nombre de la práctica, nombre del alumno, DNI, Grupo de prácticas, fecha.
- Para cada función que se os pide, una **descripción de cómo la habéis diseñado** (en la siguiente sección del enunciado tenéis un ejemplo).
    - Como muchas funciones trabajan sobre arrays, el uso de diagramas que muestren los movimientos que se realizan, indicado los límites de las zonas que se “mueven”, y que permiten deducir los límites de un bucle for, son la mejor forma de explicar el diseño.

    - Pensad que esos diagramas **ya los deberíais estar haciendo antes de lanzaros a escribir código.** El informe, simplemente, es pasar a limpio y comentar **lo que ya habéis hecho antes.** Y, si no estáis acostumbrados a hacerlo, **deberías empezar cuanto antes.**
- Si es el caso, también podéis comentar posibles **diseños alternativos** de la función, si los habéis considerado y porqué los habéis descartado (o por qué habéis preferido el diseño elegido)
- Si habéis **añadido métodos de prueba**, comentadlo en el informe. ¿Que cosas comprueban?¿Cómo los habéis diseñado?¿Por qué los habéis considerado convenientes?
- Una sección final de **conclusiones** que responda a la pregunta: **¿qué he aprendido solucionando la práctica?**
- Una sección de **bibliografía**: apuntes, libros, páginas web, etc.

Como se ha indicado, además del código de vuestras funciones, para cada uno de los apartados os pedimos una descripción sobre cómo habéis pensado la solución. Puede contener texto, diagramas (los podéis realizar a mano y luego escanearlos), etc, etc.

En general, no debería de contener código Java (éste ya aparece en el código del proyecto). Si realmente pensáis en Java, decídnoslo, y os enviaremos un Blade Runner para comprobar si sois humanos o replicantes.

Especialmente **no debería contener descripciones línea a línea de código java.** Sabemos java y somos capaces de leer el código que nos proporcionáis y reconocer qué es una asignación, qué es un condicional, etc, etc.

Por ejemplo, considerad la función `toString`, que se os proporciona:

```
public String toString(int[] num) {
    char[] chars = new char[num.length + num.length / 8];
    int iChars = 0;
    for (int iNum = num.length - 1; iNum >= 0; iNum--) {
        chars[iChars] = (char) ('0' + num[iNum]);
        iChars = iChars + 1;
        if (iNum != 0 && iNum % 8 == 0) {
            chars[iChars] = ' ';
            iChars = iChars + 1;
        }
    }
    return new String(chars, 0, iChars);
}
```

Dicha función escribe, en forma de cadena, los bits correspondientes a un integral (representado por un array de enteros en el que la posición 0 se corresponde con el bit de menor peso). Además, para mejorar la legibilidad, cada 8 espacios contando desde el bit menos significativo, se dejará un espacio en blanco. Es decir, si el número es:

```
new int[] {0, 1, 0, 1, 0, 0, 1, 1, 0, 1}
```

el resultado de `toString` debería ser `"10 11001010"`.

¿Cómo se ha diseñado la solución?

Una descripción del diseño de la función podria ser la siguiente:

- Se ha de crear un `String`, pero para hacerlo, se necesitará crear primero un array para guardar los caracteres del `String`.
    - ¿Qué tamaño tiene el array? Como es preciso guardar tantos bits como hay en la `num`, serán necesarias `num.length` posiciones. Pero como cada 8 bits se añadirá un espacio, a este tamaño he de añadir `num.length / 8`.
    - A este array le denominamos `chars`.
- El orden de los bits en el array es diferente que en el `String` (la posición 0 del array contiene el bit menos significativo pero en el `String` esta posición está ocupada por el más significativo, ya que es el que aparece más a la izquierda).
    - Se ha de recorrer `num` y `result` en sentidos diferentes, por tanto se usarán dos índices: `iNum` para el array de enteros y `iChars` para el de caracteres.
        - En este caso no podemos calcular uno en función de otro pues tenemos espacios en el array de caracteres que no se corresponden con ninguna posición del array de números
    - Se recorrerá `num` de izquierda a derecha, es decir, desde el bit más significativo hasta el menos significativo (empezaré desde `num-length - 1` y se descenderá hasta haber tratado el `0`).
    - Por tanto, `iChars` comenzará desde `0` (primera posición del `String`).
- A cada paso, si en el array num hay un 0, en chars se coloca '`0`' y, si hay un 1, un '`1`'.
    - La expresión usada: `(char) ('0' + num[iNum])`, está explicada en los apuntes del tema de Programación Orientada a Objetos (página 13).
- ¿Dónde se añaden espacios? Imaginemos un `num` que tiene 20 posiciones. Su representación sería:

  ```
  (posiciones 19 a 16) + “ “ + (posiciones 15 a 8) + “ “ + (posiciones 7 a 0)
  ```

  Es decir, hay un espacio a la derecha de cada posición (`iNum`) que es múltiplo de 8, excepto del caso `iNum=0` que no tiene un espacio a la derecha. Por tanto, la condición es: `iNum != 0 && iNum % 8 == 0`.

Criterios de evaluación

- Cada apartado aporta 1 punto a la evaluación, el 40% del mismo se corresponde a su informe, el 60% a su implementación
- No solamente se tendrá en cuenta que calcule bien su resultado, sino que el código sea entendible, los nombres de las variables razonables y que, en caso de ser complicada, use funciones auxiliares, etc.

Formato de la entrega

Un fichero **ZIP** (no `rar`, `7z`, `tgz`, etc.) que contenga:

- El directorio del proyecto IntelliJ con vuestra solución (borrad el directorio **out** del proyecto antes de comprimirlo para que ocupe el mínimo de espacio).
- Un **PDF** (no `doc`, `docx`, `txt`, `odf`, etc.) con el informe, **dentro del directorio raíz del proyecto**.

La entrega se realizará vía **campus virtual** (en el apartado de **actividades**).

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 carácter, 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 carácter 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:

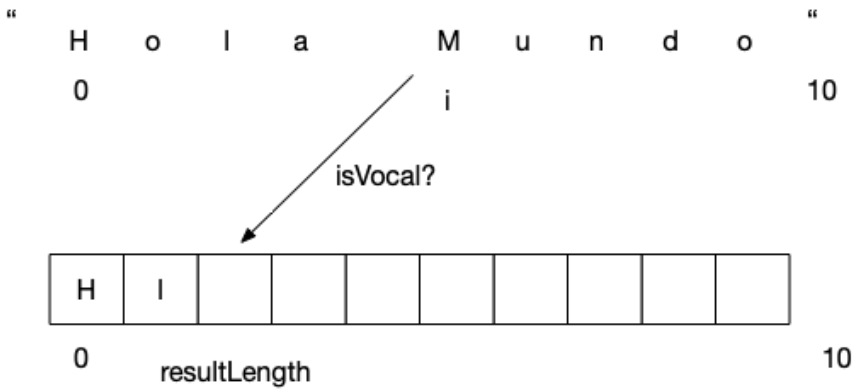
Diseñad e implementad una función tal que dado un String, retorne otro que contenga los caracteres de la cadena original que no sean vocales.

```
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 la 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:



Donde (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.