

Práctica 4: Diseño modular de programas C++ con vectores

4.1. Introducción

Los diferentes lenguajes de programación disponen de uno o más tipos de datos predefinidos para trabajar con información numérica entera. En el caso de C++ hay predefinidos, entre otros, los tipos `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned`, `unsigned long` y `unsigned long long`. Independientemente del tipo de dato que se use, la magnitud de los datos enteros que se pueden representar está limitada. Por ejemplo, cuando un compilador en concreto opta por representar los enteros con 32 bits (lo cual sucede en el caso de MinGW y GCC, los compiladores que estamos utilizando en este curso en Windows y en GNU/Linux/macOS, respectivamente), solo es posible representar los enteros del intervalo $[-2\,147\,483\,648, 2\,147\,483\,647]$ (*enteros con signo*) o los naturales del intervalo $[0, 4\,294\,967\,295]$ (*enteros sin signo*), es decir, todos los enteros o naturales con un máximo de 9 dígitos y solo algunos enteros o naturales de 10 dígitos.

En esta práctica se va a romper esa barrera de los 10 dígitos y se va a trabajar con números naturales que puedan tener decenas, centenas o incluso un número mayor de dígitos.

En primer lugar, vas a desarrollar un módulo denominado «naturales-grandes» que facilite una colección de funciones para trabajar con números naturales cuya magnitud, definida en tiempo de compilación, pueda ser tan grande como se desee. Posteriormente, desarrollarás y probarás al menos uno de los tres programas de aplicación que se proponen. Estos problemas trabajan con naturales cuyo valor puede desbordar muy ampliamente la magnitud de los tipos de datos enteros predefinidos en C++.

4.2. Representación de naturales grandes mediante vectores

En esta práctica, **se propone la representación de números naturales mediante vectores de enteros sin signo**. Cada vector tendrá `NUM_DIGITOS` componentes (constante definida con el valor 1000 en el código que suministrado) y en él se almacenarán los dígitos en base 10 del número natural a representar. En la componente indexada por 0 del vector se almacenará el dígito correspondiente a las unidades de ese número natural, en la componente indexada por 1, las decenas, en la indexada por 2, las centenas, y así sucesivamente.

Las componentes correspondientes a órdenes de magnitud superior a la de la cifra más significativa del número natural a representar tendrán valor cero.

Considera, a modo de ejemplo, la representación del número natural 1 742 863 427 043 573, que consta de 16 cifras. Con el esquema propuesto, su representación sería la siguiente:

| | | | | | | | | | | | | | | | | | | | | | |
|-----|-----|-----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 999 | 998 | ... | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | ... | 0 | 0 | 0 | 1 | 7 | 4 | 2 | 8 | 6 | 3 | 4 | 2 | 7 | 0 | 4 | 3 | 5 | 7 | 3 |

El vector que representa al número natural 1 742 863 427 043 573 tiene NUM_DIGITOS (1000) componentes. Puede observarse que en la componente indexada por 0 del vector se almacena el dígito menos significativo (el valor 3); en la componente indexada por 1, las decenas (el valor 7); y así sucesivamente. La componente más significativa de este número natural de 16 cifras se almacena en la componente de índice 15 (el valor 1). A partir de esa componente, las restantes componentes (índices 16 a 999) almacenan el valor 0.

4.3. Trabajo a desarrollar en esta práctica

Es conveniente realizar un trabajo suficiente con anterioridad a la sesión de prácticas que te corresponda, con el objeto de sacar el máximo rendimiento a dicha sesión.

El código fuente de los programas a desarrollar en esta práctica se localizará en un directorio denominado «practica4», ubicado dentro de tu carpeta «Programacion1». Para que el programa de pruebas descrito en la Sección 4.3.1 compile correctamente, el directorio «practica4» tiene que estar al mismo nivel que el directorio «practica3», es decir, ambos directorios tienen que estar en el mismo directorio (la carpeta «programacion1»). En la Sección 4.3.1 se muestra un esquema en el que se indica cómo tiene que quedar la organización de estos directorios.

En el repositorio <https://github.com/progl-eina/practica4> tienes el código de partida para esta práctica, que contiene el módulo de biblioteca «naturales-grandes» con el que se va a trabajar, así como los esqueletos de los programas a desarrollar y un programa que hace pruebas de unidad. Todos ellos ya están configurados para ser compilados, ejecutados y depurados a través de los ficheros «Makefile» necesarios y a través de la definición de tareas de Visual Studio Code.

Puedes descargar el área de trabajo completa (botón `Code` `Download ZIP` de la web del repositorio), y descomprimirla en tu directorio «programacion1» como «practica4» (recuerda borrar el sufijo «-main» que añade GitHub al preparar el fichero comprimido).

4.3.1. Tarea 1. Definición del módulo de biblioteca «naturales-grandes»

Descripción y organización del código de partida

Abre en Visual Studio Code el directorio «practica4», con la opción de menú `File` `Open Folder...` y eligiendo la carpeta «practica4». Recuerda que si no la abres de esta manera, no verás las tareas configuradas que te proporcionamos y que te facilitará el trabajo de compilación, ejecución y depuración. Una vez abierta, si seleccionas el botón `Explorer` (el primero de los del panel de la izquierda) tienes que ver el nombre del directorio «practica4» en mayúsculas y, colgando de él, los directorios «.vscode», «src», «test» y el fichero «Makefile». Estos elementos son, en concreto:

Directorio «src»: Contiene el código fuente con el que tendrás que trabajar, que consta de un módulo de biblioteca (el módulo «naturales-grandes», que se describe más adelante) y los esqueletos de tres módulos principales («fibonacci-main.cpp», «lychrel-main.cpp» y «primera-potencia-main.cpp») en cuyas implementaciones tendrás que utilizar el módulo «naturales-grandes». El comportamiento de estos programas se especifica a continuación en las tareas 2 a 4.

Directorio «test»: Contiene el código fuente de un programa de pruebas de unidad de las funciones del módulo «naturales-grandes». Tienes más detalles en la Sección 4.3.1.

El fichero «Makefile»: Contiene las reglas necesarias para que la herramienta «make» invoque al compilador de C++ de forma que se generen los programas «primera-potencia», «fibonacci», «lychrel» y el programa de prueba del módulo «naturales-grandes», cuyo fuente está en el directorio «test».

Directorio «.vscode»: Contiene los ficheros de configuración de las tareas de Visual Studio Code que permiten compilar, ejecutar y depurar cada uno de los programas con los que se trabajará en esta práctica. Los detalles sobre su contenido los puedes volver a revisar en el guion de la práctica 3.

Descripción del módulo «naturales-grandes»

El módulo «naturales-grandes» facilita a otros módulos las siguientes funciones y procedimientos para trabajar con números naturales de una gran magnitud:

- Procedimiento `void copiar(const unsigned original[], unsigned copia[])`, que permite copiar el número natural grande representado por el vector `original` al vector `copia`. Como los vectores no tienen disponible el operador de asignación, utilizaremos esta función como sustituta y así poder copiar el valor de un número natural grande representado en un vector a otro vector.
- Función `bool sonIguales(const unsigned a[], const unsigned b[])`, que permite comprobar si el número natural grande representado por el vector `a` es igual al número natural grande representado por el vector `b`. De forma análoga al caso del procedimiento `copiar`, como los vectores no tienen disponible el operador de comparación, utilizaremos esta función como sustituta y así poder comprobar si los números naturales grandes representados por los dos vectores de entrada son el mismo o no.
- Función `unsigned numCifras(const unsigned natural[])`, que devuelve el número de cifras del número natural grande representado por el vector `natural` (esto es, el número de cifras con las que se escribe ese número natural grande sin ceros a la izquierda).
- Procedimiento `void escribirEnPantalla(const unsigned natural[])`, que escribe en la pantalla el número natural grande representado por el vector `natural` (sin ceros a la izquierda).
- Procedimiento `void convertir(const unsigned numero, unsigned naturalGrande[])`, que permite transformar un natural representable como `unsigned` (el valor del parámetro `numero`) en un vector de enteros que almacena la secuencia de sus dígitos (`naturalGrande`).
- Función `unsigned valor(const unsigned naturalGrande[])`, que devuelve el valor numérico de un número natural grande representado por un vector que almacena la secuencia de sus dígitos (`naturalGrande`).

Nota: No todos los valores de un vector que representa un número natural grande son representables como datos de tipo `unsigned`. En concreto, esta función solo producirá un resultado correcto cuando el número natural grande representado por `naturalGrande` sea inferior o igual al mayor `unsigned` representable, que es $2^{32} - 1 = 4\,294\,967\,295$. En otros casos, el valor devuelto por la función no está definido (es decir, puede ser cualquier cosa).

- Procedimiento `void sumar(const unsigned a[], const unsigned b[], unsigned suma[])`, que permite sumar dos números naturales grandes representados mediante dos vectores (`a` y `b`), asignando el resultado al tercer parámetro (`suma`), también representado como un vector de enteros sin signo.

- Procedimiento **void** `calcularImagen(const unsigned natural[], unsigned imagen[])`, que permite calcular la imagen especular de un número natural grande representado mediante el vector de enteros sin signo `natural`, asignando el resultado al parámetro `imagen`, también representado como un vector de enteros sin signo.
- Función **bool** `esCapicua(const unsigned natural[])`, que permite determinar si un número natural grande representado mediante el vector de enteros sin signo `natural` es o no es capicúa. Recuerda que un número es capicúa si se lee igual de izquierda a derecha que de derecha a izquierda (por ejemplo, los números 13 355 331, 123 454 321 y 44 555 544 son capicúa).

El fichero de interfaz del módulo «naturales-grandes» es el siguiente:

```
const unsigned NUM_DIGITOS = 1000;

/*
 * Pre: «original» almacena la representación de un número natural. La
 *       componente original[0] representa las unidades de «original», la
 *       componente original[1] las decenas de «original» y así, sucesivamente.
 *       Las componentes de índices superiores al correspondiente al dígito más
 *       significativo almacenan el valor 0. «original» y «copia» tienen
 *       «NUM_DIGITOS» componentes.
 * Post: Tras ejecutar este procedimiento, «copia» almacena una representación
 *       idéntica a la del número natural correspondiente a la de «original».
 */
void copiar(const unsigned original[], unsigned copia[]);

/*
 * Pre: «a» y «b» almacenan las representaciones de sendos números naturales.
 *       Las componentes a[0] y b[0] representan las unidades del natural «a» y
 *       «b», respectivamente, las componentes a[1] y b[1], las decenas y así
 *       sucesivamente. Las componentes de índices
 *       superiores a los correspondientes a los dígitos más significativos
 *       almacenan el valor 0. Las componentes de
 *       índices superiores almacenan el valor 0. «a» y «b» tienen «NUM_DIGITOS»
 *       componentes.
 * Post: Devuelve el valor booleano true si «a» y «b» representan el mismo
 *       número natural y false en caso contrario.
 */
bool sonIguales(const unsigned a[], const unsigned b[]);

/*
 * Pre: «natural» almacena la representación de un número natural. La
 *       componente natural[0] representa las unidades de «natural», la
 *       componente natural[1] las decenas de «natural» y así, sucesivamente.
 *       Las componentes de índices superiores al correspondiente al dígito más
 *       significativo almacenan el valor 0. «natural» tiene «NUM_DIGITOS»
 *       componentes.
 * Post: Devuelve el número de cifras de «natural» cuando se escribe en base 10
 *       sin ceros a la izquierda.
 */
unsigned numCifras(const unsigned natural[]);

/*
 * Pre: «natural» almacena la representación de un número natural grande. Sea
 *       «n» el número de dígitos de «natural». La componente natural[0]
 *       representa las unidades de «natural», la componente natural[1] las
 *       decenas de «natural» y así, sucesivamente. Las componentes de índices
 *       superiores al correspondiente al dígito más significativo
```

```
*      almacenan el valor 0. «natural» tiene «NUM_DIGITOS» componentes.
* Post: Tras ejecutar este procedimiento, se ha escrito en la pantalla el
*      número natural representado por «natural» (sin ceros a la izquierda).
*/
void escribirEnPantalla(const unsigned natural[]);

/*
* Pre: «naturalGrande» tiene «NUM_DIGITOS» componentes.
* Post: Tras ejecutar este procedimiento, «naturalGrande» contiene la secuencia
*       de dígitos equivalente a «numero».
*       La componente naturalGrande[0] almacena las unidades de «numero», la
*       componente naturalGrande[1], las decenas de «numero» y así
*       sucesivamente. Las componentes de índices superiores al correspondiente
*       al dígito más significativo almacenan el valor 0.
*/
void convertir(const unsigned numero, unsigned naturalGrande[]);

/*
* Pre: «naturalGrande» almacena los dígitos de un número natural lo
*       suficientemente pequeño como para ser representado por un dato de tipo
*       «unsigned». En la componente naturalGrande[0] se almacena el dígito
*       que representa las unidades de ese número, en la componente
*       naturalGrande[1], las decenas y así sucesivamente. Las componentes de
*       índices superiores al correspondiente al dígito más significativo
*       almacenan el valor 0. «naturalGrande» tiene «NUM_DIGITOS» componentes.
* Post: Devuelve el valor numérico del natural almacenado en «naturalGrande».
*/
unsigned valor(const unsigned naturalGrande[]);

/*
* Pre: «a» y «b» almacenan las representaciones de sendos números naturales.
*       Las componentes a[0] y b[0] representan las unidades del natural «a» y
*       «b», respectivamente, las componentes a[1] y b[1], las decenas y así
*       sucesivamente. Las componentes de índices superiores a los
*       correspondientes a los dígitos más significativos almacenan el valor 0.
*       La suma de los números representados por «a» y «b» tiene menos de
*       «NUM_DIGITOS» dígitos.
*       Los vectores «a», «b» y «suma» tienen «NUM_DIGITOS» componentes.
* Post: Tras ejecutar este procedimiento, «suma» almacena la representación de
*       un número natural equivalente a la suma de los naturales representados
*       por «a» y «b». Sea «s» el número de dígitos de suma. La componente
*       suma[0] representa las unidades de la suma, la componente suma[1]
*       representa las decenas y así sucesivamente.
*/
void sumar(const unsigned a[], const unsigned b[], unsigned suma[]);

/*
* Pre: «natural» almacena la representación de un número natural. La
*       componente natural[0] representa las unidades de «natural», la
*       componente natural[1] las decenas de «natural» y así sucesivamente. Las
*       componentes de índices superiores al correspondiente al dígito más
*       significativo almacenan el valor 0. «natural» e «imagen» tienen
*       «NUM_DIGITOS» componentes.
* Post: Tras ejecutar este procedimiento, «imagen» almacena la representación
*       del número natural correspondiente a la imagen especular de «natural».
*/
void calcularImagen(const unsigned natural[], unsigned imagen[]);

/*
```

```
* Pre: «natural» almacena la representación de un número natural. La
* componente natural[0] representa las unidades de «natural», la
* componente natural[1] las decenas de «natural» y así sucesivamente. Las
* componentes de índices superiores al correspondiente al dígito más
* significativo almacenan el valor 0. «natural» tiene «NUM_DIGITOS»
* componentes.
* Post: Devuelve «true» si y solo si «natural» es un número capicúa.
*/
bool esCapicua(const unsigned natural[]);
```

En esta práctica, tienes que trabajar con números naturales grandes representados a través de sus dígitos, por lo que **tendrás que adaptar el código de funciones similares a las funciones realizadas en la práctica 3**. El esquema basado en convertir el número natural grande a un dato de tipo `unsigned` a través de la función `valor`, realizar la operación deseada (cálculo de la suma, de la imagen o determinación de si es capicúa) con los operadores y funciones definidos para datos de tipo `unsigned` y volver a convertir el resultado a un vector con la función `convertir` no va a funcionar con carácter general. Como se ha comentado anteriormente, esto es debido a que la función `valor` solo puede devolver valores correctos cuando el número natural grande tiene 9 dígitos o menos. En el resto de los casos, se producirán comportamientos no definidos (normalmente, desbordamientos).

Descripción de un programa de pruebas para el módulo «naturales-grandes»

En el directorio «test» se encuentra la mayor parte del código de un programa que va a servir para facilitar el desarrollo y la realización de pruebas del módulo denominado «naturales-grandes».

El programa consta de cuatro módulos organizados en siete ficheros:

1. El módulo objeto de las pruebas de este programa, «naturales-grandes», que se encuentra en el directorio «practica4/src» y cuyo fichero de implementación tienes que completar.
2. El módulo «naturales-grandes-test», formado también por un fichero de interfaz y otro de implementación. Ambos ficheros se encuentran en el directorio «practica4/test». Este módulo contiene funciones que permitirán comprobar el correcto funcionamiento de las funciones del módulo «naturales-grandes» que implementarás.
3. El módulo principal correspondiente al fichero «naturales-grandes-test-main.cpp», que se ubica también en el directorio «practica4/test». Este módulo especifica las pruebas concretas que se hacen de las funciones del módulo «naturales-grandes».
4. El módulo «testing-prog1», ya utilizado en la práctica 3. Está compuesto por los ficheros «testing-prog1.hpp» y «testing-prog1.cpp», ubicados en el directorio «practica3/test/testing-prog1».

Para que este programa se compile correctamente, las ubicaciones y nombres de los directorios correspondientes a las prácticas 3 y 4 y sus subdirectorios tienen que ser exactamente los indicados en los enunciados de las prácticas:

```
programacion1/
├── ...
├── practica3/
│   ├── ...
│   └── test/
│       ├── ...
│       └── testing-prog1/
│           ├── testing-prog1.hpp
│           └── testing-prog1.cpp
└── practica4/
    ├── ...
    ├── src/
    │   ├── naturales-grandes.hpp
    │   ├── naturales-grandes.cpp
    │   └── ...
    └── test/
        ├── naturales-grandes-test.hpp
        ├── naturales-grandes-test.cpp
        └── naturales-grandes-test-main.cpp
    └── ...
```

Si no es así, ajusta las ubicaciones y nombres de los directorios hasta que el programa de pruebas se pueda compilar y ejecutar.

Puedes compilar el programa de pruebas ejecutando el comando `make naturales-grandes-test` en la terminal de Visual Studio Code (`mingw32-make naturales-grandes-test` si estás en Windows) o ejecutando la tarea “Compilar tests del módulo «naturales-grandes»” del menú **Terminal** **Run Tasks...**. Para ejecutar el programa, puedes ejecutar la orden `bin/naturales-grandes-test` en la terminal de Visual Studio Code una vez que el programa ha sido compilado sin errores, o lanzar la tarea “Ejecutar tests del módulo «naturales-grandes»” desde el menú **Terminal** **Run Tasks...**.

Inicialmente, el programa «naturales-grandes-test» compilará con varias advertencias debido a que el código del fichero de implementación «naturales-grandes.cpp» no está completo. Al ser ejecutado, también informará de errores en los resultados de todas las pruebas realizadas. El número de errores detectados en las pruebas se irá reduciendo hasta llegar a cero conforme implementes las funciones del módulo correctamente.

En el fichero «naturales-grandes-test-main.cpp» puedes ver los casos de prueba que ejecuta el programa de pruebas. Para facilitar la escritura de este programa, los números naturales grandes se han expresado como cadenas de caracteres¹. Esto implica que, si lo necesitas, puedes añadir fácilmente nuevos casos de prueba que te ayuden a detectar o corregir errores que pueda haber en tus implementaciones de las funciones del módulo «naturales-grandes».

El programa de pruebas inicializa intencionadamente todas las componentes de los vectores que sirven de argumento a parámetros de salida de las funciones que tenéis que implementar con un valor determinado, de forma que el propio programa de pruebas puede detectar si tu código inicializa adecuadamente todas las componentes de los vectores.

Un mensaje indicando que el resultado calculado ha sido «???...???902» indica, precisamente, que no has inicializado adecuadamente todas las componentes del vector. Cada signo de interrogación representa, en ese caso, una componente que no has inicializado.

¹En el módulo «naturales-grandes-test» se han definido funciones que convierten estas cadenas de caracteres a los vectores de dígitos que se utilizan en el módulo «naturales-grandes». No es necesario comprender el código de estas funciones para compilar y utilizar el programa de pruebas. No obstante, las cadenas de caracteres se explicarán próximamente en clase de teoría.

De forma similar, cuando el programa de pruebas muestra un resultado incorrecto en el que en una o varias componentes hay un dato entero que no está entre 0 y 9, lo indicará escribiendo el valor de esa componente entre paréntesis.

Así, por ejemplo, un mensaje indicando que el resultado calculado ha sido «12(10)5(12)» indicaría que tu vector almacena el siguiente resultado:

| | | | | | | | | | | | | | | | | | | | | | |
|-----|-----|-----|----|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|----|---|----|
| 999 | 998 | ... | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | ... | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 10 | 5 | 12 |

Por último, ten en cuenta que el programa de pruebas no hace pruebas del procedimiento `escribirEnPantalla`.

4.3.2. Tarea 2. Potencias de 2

Diseña un programa interactivo que, al ser ejecutado, itere el siguiente tipo de diálogo con el usuario hasta que este responda con un 0 o un número negativo. En cada iteración, el programa ha de pedir al usuario que determine un número de dígitos y el programa escribe el exponente y valor de la primera potencia de 2 cuyo número de dígitos es mayor o igual que el número de dígitos introducidos por el usuario. El programa ha de ser capaz de trabajar con valores de potencias de 2 de hasta NUM_DIGITOS dígitos.

```
Número de dígitos (0 o negativo para acabar): 1
1 es 2 elevado a la 0.ª potencia
y es la primera potencia de 2 de 1 dígitos.

Número de dígitos (0 o negativo para acabar): 2
16 es 2 elevado a la 4.ª potencia
y es la primera potencia de 2 de 2 dígitos.

Número de dígitos (0 o negativo para acabar): 3
128 es 2 elevado a la 7.ª potencia
y es la primera potencia de 2 de 3 dígitos.

Número de dígitos (0 o negativo para acabar): 4
1024 es 2 elevado a la 10.ª potencia
y es la primera potencia de 2 de 4 dígitos.

Número de dígitos (0 o negativo para acabar): 10
1073741824 es 2 elevado a la 30.ª potencia
y es la primera potencia de 2 de 10 dígitos.

Número de dígitos (0 o negativo para acabar): 21
147573952589676412928 es 2 elevado a la 67.ª potencia
y es la primera potencia de 2 de 21 dígitos.

Número de dígitos (0 o negativo para acabar): 0
```

Este programa debe ser desarrollado completando el fichero «primera-potencia-main.cpp» del directorio «src». Tendrás que hacer uso del módulo «naturales-grandes», por lo que has de utilizar una

cláusula de inclusión `#include "naturales-grandes.hpp"` en el fichero «primera-potencia-main.cpp».

Podrás compilar el programa ejecutando el comando `make primera-potencia` en la terminal de Visual Studio Code (`mingw32-make primera-potencia` si estás en Windows) o ejecutando la tarea “Compilar proyecto «primera-potencia»” del menú `Terminal >> Run Tasks...`. Para ejecutar el programa, puedes ejecutar la orden `bin/primera-potencia` en la terminal de Visual Studio Code una vez que el programa ha sido compilado sin errores, o lanzar la tarea “Ejecutar proyecto «primera-potencia»” desde el menú `Terminal >> Run Tasks...`.

Observa que este problema es más sencillo de lo que parece inicialmente, puesto que para generar potencias de dos, la única operación matemática que hace falta es la suma:

$$\begin{aligned}1 + 1 &= 2 = 2^1 \\2 + 2 &= 4 = 2^2 \\4 + 4 &= 8 = 2^3 \\8 + 8 &= 16 = 2^4 \\16 + 16 &= 32 = 2^5 \\&\dots\end{aligned}$$

Por ello, las funciones definidas en el módulo «naturales-grandes» son suficientes para resolver este problema.

4.3.3. Tarea 3. Números de Fibonacci

La sucesión de Fibonacci es una sucesión infinita de números naturales cuyo primer término se define como 0 y cuyo segundo término como 1. Los restantes términos son iguales a la suma de los dos que le preceden. Estos son los primeros términos de esta sucesión infinita:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

Diseña un programa interactivo que, al ser ejecutado, itere el siguiente diálogo con el usuario hasta que este responda con un 0 o un número negativo. En cada iteración, pide al usuario que determine las posiciones del término inicial y final de la sucesión de Fibonacci que debe presentar a continuación. El programa ha de ser capaz de ser capaz de calcular términos de la sucesión de Fibonacci de hasta `NUM_DIGITOS` dígitos.

```
Términos inicial y final (0 o negativo para acabar): 1 10
1. 0
2. 1
3. 1
4. 2
5. 3
6. 5
7. 8
8. 13
9. 21
10. 34

Términos inicial y final (0 o negativo para acabar): 40 60
40. 63245986
41. 102334155
...
59. 591286729879
60. 956722026041

Términos inicial y final (0 o negativo para acabar): 200 205
200. 173402521172797813159685037284371942044301
201. 280571172992510140037611932413038677189525
202. 453973694165307953197296969697410619233826
203. 734544867157818093234908902110449296423351
204. 1188518561323126046432205871807859915657177
205. 1923063428480944139667114773918309212080528

Términos inicial y final (0 o negativo para acabar): 0
```

Este programa debe ser desarrollado completando el fichero «fibonacci-main.cpp» del directorio «src».

También tendrás que hacer uso del módulo «naturales-grandes», por lo que has de utilizar una cláusula de inclusión `#include "naturales-grandes.hpp"` en el fichero «fibonacci-main.cpp».

Puedes compilar el programa de pruebas ejecutando el comando `make fibonacci` en la terminal de Visual Studio Code (`mingw32-make fibonacci` si estás en Windows) o ejecutando la tarea “Compilar proyecto «fibonacci»” del menú `Terminal >> Run Tasks...`. Para ejecutar el programa, puedes ejecutar la orden `bin/fibonacci` en la terminal de Visual Studio Code una vez que el programa ha sido compilado sin errores, o lanzar la tarea “Ejecutar proyecto «fibonacci»” desde el menú `Terminal >> Run Tasks...`.

4.3.4. Tarea 4. Números de Lychrel

Dado un número natural n , vamos a considerar un proceso aritmético consistente en calcular la imagen especular de n y sumársela al propio n , proceso que repetiremos con las sumas resultantes hasta obtener un número capicúa.

Se denomina *número de Lychrel* a un número natural que *nunca* produce un número capicúa cuando se le aplica reiteradamente el proceso descrito en el párrafo anterior.

Así, por ejemplo, 56 no es un número de Lychrel, puesto que $56 + 65 = 121$, que es un número capicúa. El número 58 tampoco es número de Lychrel, puesto que $58 + 85 = 143$, que, en este caso, no es capicúa; pero repitiendo el proceso con el número resultante (143), se obtiene el número 484 ($= 143 + 341$), que sí es capicúa.

Alrededor del 80 % de los números naturales por debajo de 10 000 producen un número capicúa en 4 iteraciones o menos, y en torno al 90 % lo producen en 7 o menos iteraciones. Por ejemplo, el número 89 necesita 24 iteraciones del proceso hasta que se convierte en un capicúa².

No se ha demostrado todavía que existan los números de Lychrel en base decimal, aunque el número 196 es el menor natural que podría serlo.

Escribe un programa que solicite al usuario un número natural y que muestre el proceso de obtener un número capicúa a través del proceso iterativo ilustrado previamente de sumarlo con su imagen especular y repetir el proceso. El programa debe terminar cuando se obtenga un número capicúa o un resultado que alcance los NUM_DIGITOS dígitos.

Se muestran a continuación resultados de la ejecución del programa con varias entradas de usuario:

```
Escriba un número natural: 22
Iteración 0: 22

22 no es un número de Lychrel.
```

```
Escriba un número natural: 56
Iteración 0: 56
Iteración 1: 56 + 65 = 121

56 no es un número de Lychrel.
```

```
Escriba un número natural: 58
Iteración 0: 58
Iteración 1: 58 + 85 = 143
Iteración 2: 143 + 341 = 484

58 no es un número de Lychrel.
```

²Wikipedia. «Lychrel number» *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Lychrel_number. Consultado el 5 de noviembre de 2020.

Escriba un número natural: 89

Iteración 0: 89

Iteración 1: $89 + 98 = 187$

Iteración 2: $187 + 781 = 968$

Iteración 3: $968 + 869 = 1837$

Iteración 4: $1837 + 7381 = 9218$

Iteración 5: $9218 + 8129 = 17347$

Iteración 6: $17347 + 74371 = 91718$

Iteración 7: $91718 + 81719 = 173437$

Iteración 8: $173437 + 734371 = 907808$

Iteración 9: $907808 + 808709 = 1716517$

Iteración 10: $1716517 + 7156171 = 8872688$

Iteración 11: $8872688 + 8862788 = 17735476$

Iteración 12: $17735476 + 67453771 = 85189247$

Iteración 13: $85189247 + 74298158 = 159487405$

Iteración 14: $159487405 + 504784951 = 664272356$

Iteración 15: $664272356 + 653272466 = 1317544822$

Iteración 16: $1317544822 + 2284457131 = 3602001953$

Iteración 17: $3602001953 + 3591002063 = 7193004016$

Iteración 18: $7193004016 + 6104003917 = 13297007933$

Iteración 19: $13297007933 + 33970079231 = 47267087164$

Iteración 20: $47267087164 + 46178076274 = 93445163438$

Iteración 21: $93445163438 + 83436154439 = 176881317877$

Iteración 22: $176881317877 + 778713188671 = 955594506548$

Iteración 23: $955594506548 + 845605495559 = 1801200002107$

Iteración 24: $1801200002107 + 7012000021081 = 8813200023188$

89 no es un número de Lychrel.

```

Escriba un número natural: 196
Iteración 0: 196
Iteración 1: 196 + 691 = 887
Iteración 2: 887 + 788 = 1675
...
Iteración 2390: (....) + (....) = 9296693438763189453267027569211549676094124929...
35791109058730693081374197672272299294124906858461030746317623538923678243867028
Iteración 2391: 929669343876318945326702756921154967609412492903272276790384081287137
8499110875362280765646062177301188454494216660995096689632366906051125197124537875507
1518089756816370476743455470487571345924995011098002716797111668856959359967180386446
8297945247632215914899591115502905031533642534948003710431451980697307992958891417901
9659016508630541583319498708536934957426274517733685005252863491402387323903726038541
4024613614759801567520631086390088971092475325954466258524731411299641985052049814846
8944242548973752795024949023599320422642585266446051357430116888010359112502576619795
8506405431414474161631042269420409525935150948633771657152476052874481789392327624492
6914619857019882408895928981270698025413400640984854524633512950831551109590940862222
5851449802764469398167005306964896611069771729980110949953954316479497545533667517261
8756890814180457873542180252114960866323698670148906672238445498010378216163655808315
35791109058730693081374197672272299294124906858461030746317623538923678243867028 + 82
0768342876329835326713647030164858609421492992272276791473180396037850901197535138085
5636161287301089454483227660984107689632366806941125208124537875408141809865781627157
6633554579497461345935994901108992717796011669846960350076189396446720894415852222680
4909590115513805921533642545848904600431452089607218982959880428891075891641962944267
2329398718447825067425175617733684905153952590402496224013616147441413450460585979166
7520521195301088861103475315064466258524622402399532094942059725737984524244986484189
4025058914699211413742585266445952357429017988009368013602576510895741631642041458306
2730932378320419436825250058633771547262475943963580789491338514503680561095691097141
9885929970379608915413401730084943524633513050920551119599841951223674254979286446830
8176995395965886611179761720089011059942954317578407455434767407361865798081517055787
3542179152115060966323698669059906661249445488110377126064656708226357801199487317821
80483097672272309294214906769451129657207623549813678343966929 = 17504376867526487806
5341640395131982621883398589554455358185726168317570081228507136616212822234646022779
0897744432197920437926473371299225040524907575091529361884146326420533770100499850326
9186098991220699543459312333870391971004336978289355068894061544427198091812310167109
5306728508079690831086290407030452697591877184679304179329282599842556488974269847600
2485145013546736991040681608180488354791734218598281591182206195932350411522816911778
3219595064101893251704935381369917407999410954058487894849988385946890500079382985318
3638517053289200371485913487601972712610515313069159227218518290572243619746477408289
6276040100726754320441495199683806257888366613899637202308139308538287818589516503069
4082680237106979804926702600175210222919078281344625939995956289377063440007029307832
2224953345006912200989690863405790500096843492462374148716293510157470843594042300218
3264739733920881333348789098612075534222831251654171571229007462475261857295344544608
588339813627912160403525247088737356587833957

196 podría ser un número de Lychrel.




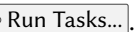
```

Como la escritura de los números naturales grandes en la pantalla es lenta (sobre todo, si la terminal que se utiliza es la integrada en el Visual Studio Code), **el programa «lychrel» debe mostrar los números que se suman en cada iteración y el resultado de la suma solo cuando los sumandos tengan menos de 20 dígitos o cuando tengan más de (NUM_DIGITOS - 1) dígitos. En el resto**

de los casos, debe escribir solo el número de la iteración seguido de unos puntos suspensivos (aunque, evidentemente, el programa siga calculando imágenes especulares, sumando y comprobando si el resultado es un número capicúa). Se muestra un ejemplo a continuación:

```
...
Iteración 39: 3603815405135183953 + 3593815315045183063 = 7197630720180367016
Iteración 40: 7197630720180367016 + 6107630810270367917 = 13305261530450734933
Iteración 41: ...
Iteración 42: ...
...
```

Este programa debe ser desarrollado en el fichero de nombre «lychrel-main.cpp» del directorio «src». Como en los dos problemas anteriores, tendrás que hacer uso del módulo «naturales-grandes».

Puedes compilar el programa de pruebas ejecutando el comando `make lychrel` en la terminal de Visual Studio Code (`mingw32-make lychrel` si estás en Windows) o ejecutando la tarea “Compilar proyecto «lychrel»” del menú  . Para ejecutar el programa, puedes ejecutar la orden `bin/lychrel` en la terminal de Visual Studio Code una vez que el programa ha sido compilado sin errores, o lanzando la tarea “Ejecutar proyecto «lychrel»” desde el menú  .

4.4. Entrega de la práctica

Antes del **sábado 19 de noviembre a las 18:00**, se deberán haber subido a Moodle el fichero «naturales-grandes.cpp» y **al menos uno** de los ficheros correspondientes a los módulos principales de los programas solicitados en las tareas 2, 3 y 4 («primera-potencia-main.cpp», «fibonacci-main.cpp» o «lychrel-main.cpp»). Si has realizado alguna otra tarea más, puedes entregar también los ficheros correspondientes.

El fichero «naturales-grandes.hpp» **no** hay que subirlo a Moodle, puesto que no debe ser **modificado** (ni añadiendo funciones en el mismo, ni modificando las cabeceras de las ya existentes). Si lo modificas, es muy probable que el resultado de la corrección que realicemos los profesores probablemente termine en error de compilación en todos y cada uno de los programas correspondientes a esta práctica, lo que acarreará una calificación de 0.