ICOM2021 – 2do Parcial

1 de noviembre de 2021

Notas:

1. Uso de prácticos: **se pueden utilizar los trabajos prácticos propios realizados.**

Problema 1: Uso de tries

Una característica común en las aplicaciones de navegación satelital (GPS) es contar con una facilidad de búsqueda de destinos (claves). En tal facilidad el usuario ingresa el nombre del sitio deseado utilizando un teclado virtual táctil en la pantalla del equipo.

Para eliminar la posibilidad de búsquedas de destinos que no se encuentran en la base de datos del sistema se desea diseñar un teclado en donde, a un dado momento, solo se encuentren habilitadas las teclas que permiten expandir el prefix actual (destino parcialmente ingresado) en un nuevo prefix o claves existentes en la base de datos del programa.

Dada la estructura de Trie y sus funciones de manipulación que se encuentran ya implementadas en el archivo **`trie.cpp`** se solicita implementar las siguientes funciones:

a. Función que debe poner **`enabledKeys[i] = true`** o **`false`**, indicando si la tecla **`i`** debe estar habilitada o no, dado el **`prefix`** enviado. Prototipo: (respetarlo!)

```
void trieGetEnabledKeys(const Trie *trie, const char *prefix, bool enabledKeys[ALPHABET_SIZE]);
```

b. Función que debe retornar una lista de claves que comparten el **`prefix`** dado. Prototipo:

```
using ListWord = list<string>;
ListWord trieGetKeys(const Trie_t *trie, const char *prefix);
```

En **`trie.cpp`** hay un main preparado para leer un archivo de calles y probar el código.

Problema 2: Infraestructura de simulación

Se propone implementar una infraestructura simple para realizar simulaciones de distintos modelos interactuando entre sí.

En el archivo **`Simulator.cpp`** se encuentra un esqueleto de la infraestructura propuesta en donde se destacan los siguientes UDSs:

`Variable`: Define una variable numérica, con un nombre asociado.

`VarManager`: Define un administrador de variables, donde los modelos, en la etapa de *creación*, deben registrar sus variables propias para que sean visibles desde otros modelos. Además brindan un servicio para ganar acceso a una variable a través de un nombre, realizado desde otro modelo en la etapa de *conectado* de los modelos.

`Model`: Define la clase base de todos los modelos, y como tal define la interfase abstracta que debe poseer cualquier modelo "instanciable"

`Grifo`, **`Tanque`**, **`BombaRiego`**, **`AspersorRiego`**: Modelos de prueba, deben implementar la interfase **`Model`**, tener algunas variables que deben ser registradas en el **`VarManager`** en la etapa de creación del modelo, y hacer uso de variables externas (de otros modelos) tomadas del **`VarManager`** en la etapa de *connect*.

`FakeMonitorModel`: Esta clase es también un modelo, pero su funcionalidad es observar a todas las variables del sistema.

`Simulator`: Modela el contexto de simulación. Es quien tiene los modelos registrados y es el que en la **`prepare`**, manda a preparar y *connectar* los modelos previamente registrados.

`Scheduler`: Tipo interno al **`Simulator`**, con la funcionalidad de avanzar la simulación de los distintos modelos.

Se solicita implementar los métodos/clases marcados con **`// TODO`**

Problema 3: Claves públicas RSA

El sistema de claves públicas RSA permite la transmisión segura de datos. Está basado en la dificultad no-polinomial[1] de encontrar los factores primos de un número. Cada usuario debe generar una clave pública e y otra privada d a partir de un número n grande (e, d y n son números enteros).

Codificación y decodificación

Supongamos que Roberto quiere enviar un mensaje m a Alicia, donde m es un número entero. Para cifrarlo, simplemente calcula:

$$c = m^e \,\%\, n$$

($\%$ es la operación módulo, resto de la división entera) utilizando la clave pública e de Alicia y le envía el mensaje codificado c.

Alicia para decodificarlo calcula:

$$m = c^d \,\%\, n$$

utilizando su clave privada d, que solo ella conoce.

e, d y n deben ser obtenidos de manera que las operaciones que realizan Roberto y Alicia sean inversas una de la otra y que conociendo solo e y n sea muy difícil calcular d.

a) Implementar una función que calcule el exponente modular de un número:

$$x^{n_1} \,\%\, n_2$$

```
long exponente_modular(long x, long n1, long n2);
```

En general n_1 va a ser un número grande, por lo que va a ser necesario realizar las multiplicaciones de a una y utilizar que

$$x^s \,\%\, n_2 = (x * x^{s-1}) \,\%\, n_2 = ((x \,\%\, n_2) * (x^{s-1} \,\%\, n_2)) \,\%\, n_2$$

Ayuda: se puede verificar que esta expresión funciona adecuadamente utilizando la siguiente igualdad válida para cualquier número primo $p > 2$.

$$2^{p-1} \,\%\, p = 1,$$

por ejemplo: $2^6 \% 7 = (1 + 7 * 9) \% 7 = 1$.

b) Implementar la función cifra que reciba un **`string`** y retorne un **`MensajeCifrado`** cifrando carácter a carácter utilizando un e y n dados.

```
using MensajeCifrado = vector<long>;
MensajeCifrado cifra(const string &s, long e, long n);
```

c) Implementar una función descifra que reciba un **`MensajeCifrado`** y retorne un **`string`** con el mensaje descifrado a partir de un d y n dados.

```
string descifra(const MensajeCifrado &msg, long e, long n);
```

d) Utilice la función **`descifra`** para descifrar el siguiente **`MensajeCifrado`** utilizando $d = 1553$ y $n = 6767$.

```
MensajeCifrado mensaje {4162, 5524, 789, 1445, 4537, 3698, 964, 1273, 4252, 1412, 3698, 6600, 5526,
                        5526, 2759, 2757, 2333};
```

[1] Complejidad NP: la cantidad de operaciones necesarias crece exponencialmente con la cantidad de bits del número.

Generación de claves

En las aplicaciones, n se calcula como el producto de dos números primos distintos obtenidos al azar.

e) Implementar una función que retorne un número primo al azar en el intervalo $[30: 200]$.

```
long rand_primo();
```

Ayuda: todos los números menores a 341 que satisfacen $2^{p-1} \% p = 1$ son primos, así que es posible utilizar esta igualdad para saber si un número es primo o no.

Utilizar esta función para obtener dos números primos distintos p y q.

La clave pública e tiene que ser un número coprimo con $r = (p-1) * (q-1)$. Se recuerda que dos números son coprimos si el máximo común divisor entre ambos es 1, esto es: $mcd(e, r) = 1$.

f) Implementar una función que retorne un número e en el intervalo $[17: r)$ cuidando que e y r sean coprimos.

```
long coprimo(long r);
```

Ayuda 1: utilizar la función creada en la práctica 6, ejercicio 2.
Ayuda 2: comenzar con 17 e ir aumentándolo de a 2 hasta que sea coprimo con r.

Para calcular la clave privada d hay que obtener el inverso módulo r de e, esto es resolver la ecuación:

$$(e * d) \% r = 1, \qquad (1)$$

Para ello se puede utilizar el algoritmo de Euclides extendido (una extensión del algoritmo del ejercicio 2 de la práctica 6) que permite resolver ecuaciones del tipo:

$$a * x + b * y = mcd(a, b),$$

Obteniendo x e y que son enteros y $mcd(a, b)$ que es el máximo común divisor de a y b. Como para el algoritmo de Euclides:

- Si $b = 0$ la solución es trivial: $x = 1$, $y = 0$ y $mcd(a, b) = a$.
- Si $b \neq 0$ y tenemos la solución para b y $a\%b$:

$$mcd(b, a\%b) = b * x' + (a\%b) * y'$$

se puede demostrar que $x = y'$ e $y = x' - (a/b) * y'$ son una solución de:

$$mcd(a, b) = a * x + b * y.$$

g) Implementar la función:

```
Terna eulerext(long a, long b);
```

que dados a y b retorne $\{x, y, mcd(a, b)\}$ en una estructura

```
struct Terna {
    long x;
    long y;
    long mcd;
};
```

Ayuda: en pseudo-código es:

- Si $b = 0$ retornar la solución trivial $\{1,0, a\}$.
- Si $b \neq 0$:
  - Llamar a `eulerext(b, a % b)` para obtener x', y' y mcd.
  - Actualizar $x = y'$ e $y = x' - (a/b) * y'$.
  - Retornar $\{x, y, mcd\}$.

La ecuación (1) se puede escribir como:

$$e * d + q * r = mcd(e, r)$$

ya que $mcd(e, r) = 1$ por construcción.

En base a esto:

h) Implementar la función:

```
Claves genera_claves();
```

que retorne una estructura

```
struct Claves {
    long e;
    long d;
    long n;
};
```

En caso que d sea negativo, calcular $d = (d + r)\%r$. Verificar que $(e * d)\%r = 1$.

Ejemplo: para $p = 67$, $q = 101$, $n = 6767 = (67 * 101)$, se obtienen $r = 6600 = (66 * 100)$, $e = 17$ y $d = 1553$.

El interés de esta codificación es que si no se conocen p y q es muy difícil de calcular d.

i) Utilizar las claves generadas y las funciones implementadas para codificar y decodificar un **`string`**.