



Examen 1 - CI3641

Joao Pinto 17-10490

1. Pregunta 1

Para esta pregunta se seleccionó **Javascript**. **Javascript** es un lenguaje de programación de alto nivel que conforma la especificación **ECMAScript**. A lo largo de esta pregunta nos estaremos refiriendo al Javascript que implementa **ECMAScript 2018**. Junto con *HTML* y *CSS*, este lenguaje es uno de los componentes esenciales de la web moderna pues permite el desarrollo de páginas interactivas del lado del cliente. La mayoría de exploradores web tienen su propia implementación del lenguaje, a estas implementaciones se les denomina **Javascript Engine**. En la mayoría de engines, **Javascript** es implementado utilizando un compilador *Just-in-time (JIT)*. **Javascript** es un lenguaje multi-paradigma, admite programación funcional, programación imperativa y programación dirigida por eventos.

Si bien **Javascript** fue pensado para la web, existen entornos de desarrollo para utilizar este lenguaje en diferentes ámbitos, uno de estos es **Node.js**, que permite el uso del lenguaje fuera del explorador web haciendo uso del **V8 Javascript Engine** de **Chrome**. **Node.js** ha permitido el uso de este lenguaje en un amplio abanico de aplicaciones, desde aplicaciones *Backend Web*, pasando por aplicaciones móviles y llegando hasta programación de micro-controladores.

1.1. Parte a)

1.1.1. Alcance y asociaciones

El alcance de una variable en **Javascript** está determinado por la forma en la que se declara. Se pueden declarar variables con **var**, **let** o **const**.

Variables declaradas con **let** o con **const** tendrán alcance estático, la diferencia entre **let** y **const** es que **const** no permite reasignaciones (intuitivamente es una constante). Tanto **let** como **const** no permiten redeclaraciones dentro de su mismo bloque, sin embargo

pueden redeclararse en bloques hijos.

Cuando una variable se declara con `let` o `const` en algun punto de un bloque, esta variable no puede ser usada hasta que se le asigne un valor (incluso si en algun bloque padre esta ya estaba inicializada), si se intenta usar una variable declarada `let` o `const` pero sin valor, se lanzara una excepcion. Por ejemplo, el siguiente programa lanza la excepcion `Cannot access 'x' before initialization`:

```
1 let x = 0;
2
3 if (true) {
4   console.log(x);
5   let x = 1;
6   console.log(x);
7 }
8
9 console.log(x);
```

Al remover la linea 4 el programa ejecuta correctamente e imprime 1 y luego 0.

Variables declaradas con `var` tienen alcance dentro del bloque de `function` padre mas cercano (*El bloque global se considera como un bloque de function*). A diferencia de `let` y `const`, las variables declaradas por `var` pueden ser evaluadas sin que se les haya asignado un valor. Por ejemplo:

```
1 var x = 0;
2 var y = 0;
3
4 function P() {
5   console.log(y);
6
7   if (false) {
8     var y = 1;
9     console.log(y);
10  }
11
12  console.log(x);
13
14  function Q() {
15    console.log(y);
16  }
17  Q();
18 }
19
20 P();
```

Este programa se ejecuta correctamente e imprime `undefined`, 0 y `undefined`

El uso de `var` puede llegar a ser bastante confuso, del ejemplo vemos que un bloque que nunca se ejecuta afecta en gran medida la salida del programa. Este efecto, en proyectos mas

grandes produce numerosos bugs difíciles de *debuggear*.

`let` y `const` fueron añadidas al lenguaje a partir de **ECMAScript 2015**. A partir de ese momento se recomienda **fuertemente** no utilizar `var`, se debe utilizar `let` y `const` exclusivamente, ya que estos funcionan de manera mucho más natural y se adaptan muy bien al uso que se le da al lenguaje, en particular se adaptan muy bien al paradigma de programación dirigida por eventos, que representa una de las bases de **Node.js**. El único escenario en donde se puede considerar utilizar `var` es cuando se requiere compatibilidad con *engines* que aún no implementan **ECMAScript 2015**, sin embargo, muy pocos *engines* presentan esta situación, así que en general se debe evitar el uso de `var`.

La asociación en **Javascript** es **profunda** (*Deep binding*) y en el momento de creación de la función, esto último como consecuencia de que las funciones sean de primer orden en este lenguaje. Esto viene de maravilla pues resulta natural para el programador y no se producen efectos de borde no deseados, de hecho la asociación **profunda** cumple un valor esencial en que **Javascript** sea utilizado para programación orientada a eventos. Ya que la asociación **profunda** asegura que el estado de las funciones usadas como *callbacks* para los *events* sea exactamente el presente en el momento de definición de la función, esto es muy importante considerando que una función *callback* puede ejecutarse en cualquier momento arbitrario.

1.1.2. Módulos, importación y exportación de nombres

Javascript posee manejo nativo de módulos utilizando los *keywords* `export` e `import`.

`export` permite exportar nombres del espacio global de nombres del módulo. Se pueden marcar los nombres que se quiere exportar con `export` (*Opción 1*). Como alternativa, se puede seleccionar la lista de nombres a exportar con `export` (*Opción 2*)

```
1 // Opcion 1:
2 export const PI = 3.1432;
3 export class Tree {
4   constructor(value = null, left = null, right = null) {
5     this.value = value;
6     this.left = left;
7     this.right = right;
8   }
9 }
10
11 // Opcion 2:
12 const PI = 3.1432;
13 class Tree {
14   constructor(value = null, left = null, right = null) {
15     this.value = value;
16     this.left = left;
17     this.right = right;
18   }
19 }
```

```
20
21 export { PI, Tree };
```

`import` permite importar nombres exportados por un modulo al espacio global de nombres. Se pueden importar nombres con un alias (namespace) (*Opcion 1*). Tambien se pueden importar los la lista de nombres deseados (*Opcion 2*), en este caso se le puede colocar un alias a cada nombre importado por separado.

```
1 // Opcion 1:
2 import * as alias from './module.js';
3 // Opcion 2:
4 import { Tree as T, PI } from './module.js';
```

Como podemos ver de los ejemplos, **Javascript** permite modulos como gestor y como librerias. Cabe destacar, que **Javascript** ejecuta los modulos al momento de importacion, sin embargo, esto se hace una sola vez por cada modulo independientemente de cuantos `imports` se le hagan a un modulo.

Algunos *engines* no soportan completamente `export` e `import`. Entre ellos el popular **Node.js**. En estos casos se suelen utilizar librerias para el manejo de modulos. En **Node.js** se utiliza por defecto **CommonJS** que provee la funcion `require` y el objeto `module.exports` (entre otras funcionalidades). Algunas otras librerias para el manejo de modulos son **RequireJS** y **Webpack**.

Otro interesante metodo de introduccion de nombres que posee **Javascript** son los *destructuring assignments*. Estos permiten *desempacar* valores de arrays y propiedades de objetos en variables dentro del espacio de nombre actual.

```
1
2 // ----- Arrays
3 const B = [1, 2, 3];
4
5 const [a, ...b] = B;
6 console.log(a); // 1
7 console.log(b); // [2, 3]
8
9 const [one, , three] = B;
10 console.log(one); // 1
11 console.log(three); // 3
12
13 // ----- Objects
14 const obj = {
15   id: 1,
16   password: '12345',
17   email: 'foo@bar.com',
18   bills: ['primero', 'segundo', 'tercero'],
19   names: {
20     first: 'Pepe',
21     last: 'Grillo'
22   }
23 }
```

```

22   }
23 };
24
25 const { id, email } = obj;
26 console.log(id); // 1
27 console.log(email); // foo@bar.com
28
29 // ... y renombrando
30 const { password, id: foo, ...rest } = obj;
31 console.log(password); // 12345
32 console.log(foo); // 1
33 console.log(rest); // { id: 1, ...}
34
35 // ----- Nested
36 const {
37   bills: [first],
38   names: { last: lastName }
39 } = obj;
40 console.log(first); // primero
41 console.log(lastName); // Grillo

```

Para los *Arrays* esto permite asignar nombres locales a los elementos del arreglo. El (...) indica que el resto del *Array* se asociara al nombre que sigue los (...). Tambien se pueden ignorar posiciones no deseadas colocando , vacias. Para los *Objects* esto permite asignar nombres locales a las propiedades del objeto. El (...) tiene el mismo significado que en el caso de *Arrays*. Podemos tambien renombrar los atributos que obtenemos del objeto (Vea el bloque de codigo anterior en la linea 30). Tambien es interesante destacar que esto se puede hacer de forma anidada (Vea el bloque de codigo anterior en la linea 36).

1.1.3. Control de flujo

1.1.3.1. if else

```

1 if (cond) THEN_STATEMENT
2 else ELSE_STATEMENT

```

Sentencia *if-then-else* del lenguaje. Si la expresion `cond` es *truthy* se ejecuta `THEN_STATEMENT`, de lo contrario se ejecuta `ELSE_STATEMENT`. Una expresion `expr` es *truthy* si y solo si `Boolean(expr) === true`. Si una expresion no es *truthy*, se dice que es *falsy*. Las expresiones `undefined`, `null`, `false`, `0`, `NaN` y `''` (string vacio) son *falsy*.

`THEN_STATEMENT` y `ELSE_STATEMENT` pueden ser cualquier sentencia valida del lenguaje, desde bloques (denotados con `{}`) hasta otros *if-then-else*. Notese que esto nos permite encadenar varios *if-then-else*.

```

1 if (cond1) THEN_STATEMENT
2 else if (cond2) ELSEIF_STATEMENT
3 else ELSE_STATEMENT

```

1.1.3.2. switch case

```

1 switch (SWITCH_EXPRESSION) {
2     SWITCH_BODY
3 }

```

SWITCH_BODY consiste en zero o mas clausulas del estilo:

```

1 case CASE_EXPRESSION:
2     STATEMENTS // secuencia de statements

```

Y opcionalmente una clausula default

```

1 default:
2     DEFAULT_STATEMENTS

```

`switch` evalua `SWITCH_EXPRESSION` y, de existir, salta a la primera clausula de `SWITCH_BODY` que cuyo `CASE_EXPRESSION` sea igual al valor de `SWITCH_EXPRESSION`. Cuando se salta a una clausula se ejecuta la secuencia `STATEMENTS`, una vez ejecutada la secuencia, se salta a la siguiente clausula dentro de `SWITCH_BODY` (de existir). Usualmente la ultima expresion de cada `STATEMENTS` es `break`, de forma que el bloque `switch` no salte a las siguientes clausulas.

Si no existe ninguna clausula dentro de `SWITCH_BODY` cuyo `CASE_EXPRESSION` sea igual al valor de `SWITCH_EXPRESSION` entonces se salta a la clausula `default`. Si no existe clausula `default` entonces no se ejecuta nada.

1.1.3.3. ciclos while

```

1 while (cond) {
2     STATEMENTS
3 }

```

`while` ejecuta repetitivamente la secuencia `STATEMENTS` mientras `cond` sea *truthy*.

1.1.3.4. ciclos do-while

```

1 do {
2     STATEMENTS
3 } while (cond);

```

`do-while` ejecuta la secuencia `STATEMENTS`, luego, mientras `cond` sea *truthy* ejecuta repetitivamente secuencia `STATEMENTS`. La diferencia entre `while` y `do-while` es que `while` ejecuta **por lo menos 0 veces** la secuencia `STATEMENTS`, mientras que `do-while` ejecuta **por lo menos 1 vez** la secuencia `STATEMENTS`.

1.1.3.5. ciclos for

```
1 for (INITIALIZATION; CONDITION; POST_ITERATION) {  
2   STATEMENTS  
3 }
```

`for` ejecuta la secuencia `STATEMENTS` repetidas veces, la cantidad de repeticiones viene dada por la expresiones `INITIALIZATION`, `CONDITION` y `POST_ITERATION`. En `INITIALIZATION` se declaran las variables que viviran dentro del ciclo. `CONDITION` es la condicion de repeticion, mientras `CONDITION` sea *Truthy* sea seguira ejecutando el ciclo. `POST_ITERATION` es una expresion que se ejecuta al final de cada iteracion. Notese que cualquier ciclo `for` es equivalente al siguiente ciclo `while`:

```
1 INITIALIZATION  
2 while (CONDITION) {  
3   STATEMENTS  
4   POST_ITERATION  
5 }
```

Si `INITIALIZATION`, `CONDITION` y `POST_ITERATION` son la instruccion vacia, se produce un ciclo infinito.

```
1 for (;;) {  
2   // ciclo infinito  
3 }
```

1.1.3.6. ciclos for-of

```
1 for (ELEMENT_VARIABLE of ITERABLE) {  
2   STATEMENTS  
3 }
```

`for-of` permite iterar sobre cualquier *iterable* (como los `Array` y los `Set`). `ITERABLE` es una expresion que debe evaluar a un *iterable*, `ELEMENT_VARIABLE` es una expresion de declaracion de variable, la variable declarada tomara el valor de cada elemento en `ITERABLE`, un valor por iteracion. Se realiza una iteracion por cada elemento en `ITERABLE`. `STATEMENTS` es la secuencia ejecutada en cada iteracion.

```
1 const iterable = ['hello', 'world'];  
2 for (const elem of iterable) {  
3   console.log(elem);  
4 }  
5 // Output:  
6 // 'hello'  
7 // 'world'
```

1.1.3.7. ciclos for-in

```
1 for (ELEMENT_VARIABLE in OBJECT) {  
2   STATEMENTS  
3 }
```

`for-in` permite iterar sobre las propiedades cualquier `Object` (esto incluye los `Array`). `OBJECT` es una expresion que debe evaluar a un `Object`, `ELEMENT_VARIABLE` es una expresion de declaracion de variable, la variable declarada tomara el valor de cada propiedad de `OBJECT`, una propiedad por iteracion. Se realiza una iteracion por cada propiedad en `OBJECT`. `STATEMENTS` es la secuencia ejecutada en cada iteracion. `for-in` itera sobre las **propiedades** de un objeto, no sobre sus valores.

```
1 const arr = ['a', 'b', 'c'];  
2 arr.propKey = 'property value';  
3  
4 for (const key in arr) {  
5   console.log(key);  
6 }  
7  
8 // Output:  
9 // '0'  
10 // '1'  
11 // '2'  
12 // 'propKey'
```

1.1.3.8. continue

`continue` solo puede ser utilizado dentro de bloques `while`, `do-while`, `for`, `for-of` y `for-in`. Cuando se ejecuta `continue`, se descarta la iteracion actual y se continua a la siguiente. Por ejemplo:

```
1 const lines = [  
2   'Normal line',  
3   '# Comment',  
4   'Another normal line',  
5 ];  
6 for (const line of lines) {  
7   if (line.startsWith('#')) continue;  
8   console.log(line);  
9 }  
10 // Output:  
11 // 'Normal line'  
12 // 'Another normal line'
```


1.1.3.9. break

`break` tiene dos versiones, una con un operando y otra sin operando.

La version `break` sin operando solo puede ser utilizado dentro de bloques `while`, `do-while`, `for`, `for-of` y `for-in`. Al ejecutarse, el programa se sale del bloque del ciclo padre mas cercano, es decir, detiene el ciclo. Por ejemplo:

```
1 for (const x of ['a', 'b', 'c']) {
2   console.log(x);
3   if (x === 'b') break;
4   console.log('---')
5 }
6
7 // Output:
8 // 'a'
9 // '---'
10 // 'b'
```

La version que tiene un operando se demomina *labeled statement break*. Su operando es un *label* (una etiqueta).

```
1 my_label: { // labeled block
2   // ...
3   break my_label; // labeled break
4 }
```

Al ejecutar `break my_label` el programa se sale del bloque marcado con `my_label`. Esta expresion lanza una excepcion si no existe ningun bloque padre que este marcado con `my_label`. Se utiliza usualmente cuando se desea detener ciclos anidados. Notese que los bloques pueden ser cualquier tipo de bloque de **Javascript** valido, no esta limitado a ciclos.

```
1 loop: for (let i = 0; i < 10000; i++) {
2   for (let j = 0; j < 500; j++) {
3     if (j === 2) break loop;
4     console.log(j);
5   }
6 }
7
8 // Output:
9 // 0
10 // 1
```

1.1.3.10. Bloques try-catch


```
1 try {  
2   TRY_STATEMENTS  
3 } catch (error) {  
4   CATCH_STATEMENTS  
5 }
```

`try-catch` ejecuta la secuencia `TRY_STATEMENTS`, si durante esa ejecución se lanza una excepción `excep` entonces se le asigna `excep` a `error`, se descarta el bloque `try` y se salta al bloque `catch` para ejecutar la secuencia `CATCH_STATEMENTS`. `try-catch` permite ejecutar código que pudiera generar excepciones sin tener que detener la ejecución del programa.

1.1.4. Orden de evaluación


En **JavaScript** el orden de evaluación es de izquierda a derecha, tanto para las expresiones como para las funciones y sus argumentos. Los argumentos de las funciones son evaluados antes de ejecutar la función.

También vale la pena mencionar que **JavaScript** posee cortocircuito para los operadores booleanos.


En esta dirección  se puede encontrar la lista completa de operadores de **JavaScript** ordenados por nivel de precedencia. También se indica la asociatividad de cada operador.

1.2. Parte b)


1.2.1. Factorial

A partir de **ECMAScript 2015** algunos *engines* poseen optimización de *recursion de cola*, por esta razón, realizaremos dos implementaciones, una directa de la definición de factorial y otra utilizando recursión de cola. En el archivo *factorial.js*  en la línea 4 y en la línea 12 tenemos las respectivas implementaciones.

1.2.2. Producto de matrices

Para esta pregunta nos inclinaremos hacia el uso del lenguaje bajo el paradigma funcional, esto con la finalidad de generar un código limpio, sin embargo, utilizaremos algunas herramientas de control de flujo imperativas como el ciclo *for..of*, aunque evitaremos utilizarlas, y de utilizarlas trataremos de utilizarlas de la forma más compacta posible y evitando los efectos de borde. Nos apoyaremos mucho en las *arrow functions* y en la función *map* (*miembro de Array*). Debido al sistema de tipos de este lenguaje, es necesario validar que *A* y *B* sean matrices válidas, para ello utilizaremos *arrow functions* con cortocircuito a modo de predicados. Definiremos también una función auxiliar que nos permitiera calcular la transpuesta de una matriz y otra función auxiliar para calcular la suma de los elementos de un *Array*. En el archivo *product.js*  en la línea 4 tendremos la implementación del procedimiento solicitado.

2. Pregunta 2

La respuesta a esta pregunta se encuentra en una presentacion en formato `.pdf`. En el archivo ***pregunta-2.pdf***  se encuentra dicha presentacion.

3. Pregunta 3

Para esta pregunta hemos seleccionado **Python** (version 3.8). Para la implentacion, tendremos una clase **BuddySystem** que recibe en su constructor el numero de bloques a manejar. referenciamos un bloque por un numero entero $0 \leq k < n$ con n el numero de bloques del sistema.





Adicionalmente tendremos la clase **Partition** que guarda la direccion de inicio y de fin de la particion, el nombre (de existir) y el espacio usado de esa particion (de existir) (esto para poder representar aquellos *best fit* que no llenen por completo la particion), esta clase tendra un metodo para obtener una representacion **string** de la particion y un metodo para obtener el tamaño de la particion.



Cada instancia de **BuddySystem** guarda una lista de bloques libres por cada posible **orden**. Un orden es un valor representativo del tamaño de las particiones, en concreto, **orden** de una particion *par* se define como $\lceil \log_2(par.size()) \rceil$. Notese que todo orden *ord* cumple que $0 \leq ord \leq \lceil \log_2(n) \rceil$ con n el numero de bloques del sistema.

Cada instancia de **BuddySystem** tambien almacena un diccionario de las particiones reservadas, este diccionario es indexado por el nombre de las particiones.

BuddySystem tiene 3 metodos fundamentales, **allocate**, **deallocate** y **representation**.

allocate recibe **name** y **size**, en este metodo se reserva una particion de tamaño **size** con nombre **name** realizando las divisiones de particiones (subparticiones) necesarias. **deallocate** recibe **name**, en este metodo se libera la particion con nombre **name** realizando los *BuddyMerge* necesarios para preservar el *BuddySystem*. **representation** retorna una representacion **string** de las particiones libres y las particiones reservadas.

En el archivo ***buddy.py***  esta la implementacion de **BuddySystem** y **Partition**. Para las especificaciones de IO se implemento el archivo ***main.py*** . Este archivo es el encargo de crear la instancia de **BuddySystem** de cada ejecucion, tambien es el que manipula el sistema dependiendo del comando introducido por el usuario. En el archivo ***README.md***  se encuentran las **instrucciones para ejecutar *main.py*** .

Para las pruebas unitarias se decidio utilizar **unittest**, el modulo de pruebas unitarias por defecto de **Python**. Para la medicion del cobertura se utilizo **coverage**, que es un bastante sencillo de integrar con **unittest**. En el archivo ***buddy.test.py***  se encuentran las pruebas unitarias del *BuddySystem*. En el archivo ***README.md***  se encuentran las **in-**


trucciones para ejecutar las pruebas unitarias. Se hace mucho énfasis en las **instrucciones para ejecutar** debido a que el código se organizó utilizando módulos de **Python**. Se hizo de esta manera para poder utilizar plenamente **unittest** y **coverage**.

```
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 % coverage run -m unittest pregunta-3.tests.buddy_test
...
-----
Ran 3 tests in 0.001s

OK
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 % coverage report
Name                               Stmts   Miss  Cover
-----
pregunta-3/__init__.py              0      0   100%
pregunta-3/src/__init__.py          0      0   100%
pregunta-3/src/buddy.py            112      9    92%
pregunta-3/tests/__init__.py        0      0   100%
pregunta-3/tests/buddy_test.py      42      0   100%
-----
TOTAL                             154      9    94%
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 %
```

Aca se deja un screenshot de la salida de las pruebas unitarias y de la medición de la cobertura.

4. Pregunta 4

La respuesta a esta pregunta se encuentra en una presentación en formato **.pdf**. En el archivo ***pregunta-4.pdf***  se encuentra dicha presentación.

El source code utilizado en esta pregunta se encuentra en el archivo ***misterio.py*** .

5. Pregunta 5


Dado que $X = 4, Y = 9, Z = 0$ tenemos que $\alpha = 6 \wedge \beta = 7$. Estaremos trabajando con $F_{6,7}$.

5.1. Parte a)

Implementamos la fórmula de $F_{6,7}$, esto es:

$$F(n) = \begin{cases} n & \text{si } 0 \leq n < 42 \\ F(n-7) + F(n-14) + F(n-21) + F(n-28) + F(n-35) + F(n-42) & \text{si } n \geq 42 \end{cases}$$

con $F_{6,7}(n) = F(n)$

En el archivo ***recursion.c***  en la **línea 21** tenemos la implementación directa solicitada.

5.2. Parte b)


Expandimos evaluaciones de $F_{6,7}$ con el fin de obtener una intuición de su comportamiento. En particular nos enfocamos en aquellos n que comparten mismo valor de $n \bmod 7$:

$$\begin{aligned}
 F(42) &= F(35) + F(28) + F(21) + F(14) + F(7) + F(0) \\
 &\quad \dots \\
 F(49) &= F(42) + F(35) + F(28) + F(21) + F(14) + F(7) \\
 &\quad \dots \\
 F(56) &= F(49) + F(42) + F(35) + F(28) + F(21) + F(14) \\
 &\quad \dots
 \end{aligned}$$

En este caso tenemos $n \bmod 7 = 0$. Rápidamente notamos la dependencia entre $F(n)$ y $F(n-7)$ (para $n > 42$), si bien esto lo dice claramente la definición podemos ver algo más valioso, esto es que, entre $F(n)$ y $F(n-7)$ ocurre un desplazamiento de los términos de la suma.


$F(n_0 - 35)$ se vuelve $F(n_1 - 42)$, $F(n_0 - 28)$ se vuelve $F(n_1 - 35)$, $F(n_0 - 21)$ se vuelve $F(n_1 - 28)$, $F(n_0 - 14)$ se vuelve $F(n_1 - 21)$, $F(n_0 - 7)$ se vuelve $F(n_1 - 14)$ y claramente $F(n_0)$ se vuelve $F(n_1 - 7)$.

Este comportamiento ocurre para todo valor de $m = n \bmod 7$, la diferencia es que los casos iniciales serán desplazados m unidades. Es decir que el caso base sería $F(42 + m)$. Notese que debido a la definición de F y a la definición de m , $F(42 + m)$ es una suma que no requiere calcular F , ya que $42 + m - 7k$ (con $1 \leq k \leq 6$) siempre es menor que 42. Esto nos dice que es posible realizar una implementación *Bottom-Up* de $F(n)$ partiendo de $42 + m - 7k$ (con $1 \leq k \leq 6$) y "avanzando" hasta $F(n)$.

Utilizando lo anterior, implementamos el procedimiento solicitado. Para ello utilizamos un procedimiento (auxiliar) de cola y otro procedimiento que llama al procedimiento auxiliar con ciertos valores iniciales. En el archivo ***recursion.c***  en la **línea 33** tenemos la implementación del procedimiento auxiliar. En el mismo archivo en la **línea 54** tenemos la implementación del procedimiento principal.

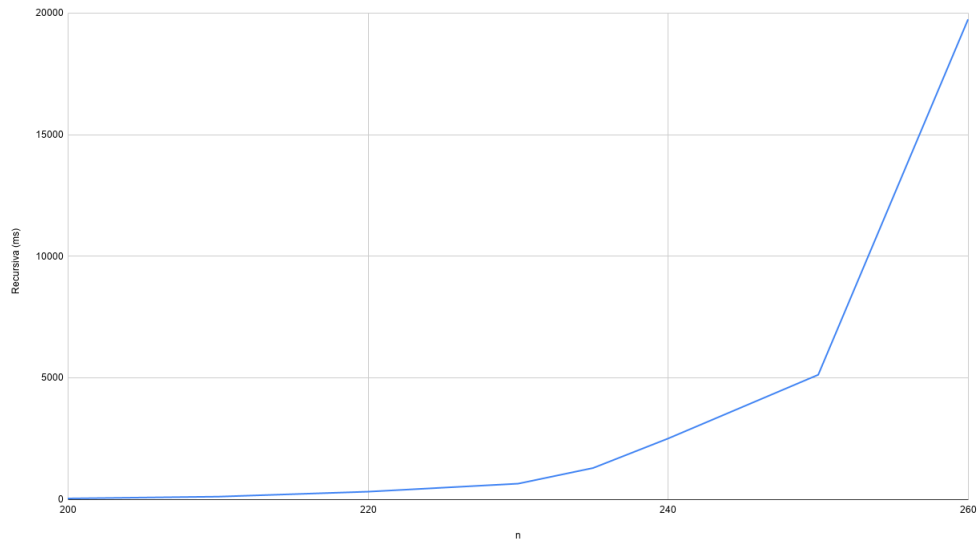
5.3. Parte c)

En el procedimiento de la parte b) se colocó un comentario por cada componente del procedimiento, estos comentarios comienzan por una letra mayúscula que los identifica. Dentro del procedimiento a implementar en esta parte también se colocarían comentarios en cada componente del procedimiento, sin embargo, en este caso la letra mayúscula por la que comienza cada comentario indica con cuál componente del procedimiento de la parte b) se asocia cada componente.

En el archivo *recursion.c*  en la **línea 63** tenemos la version iterativa solicitada.

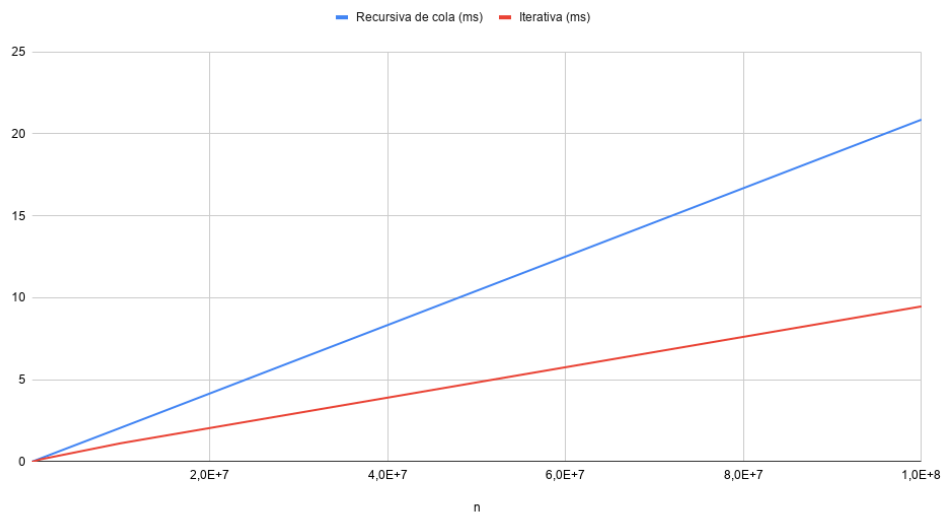
5.4. Analisis de tiempo de ejecución

Recursiva (ms) frente a n



Podemos notar inmediata que la version directa de $F_{6,7}$ es extremadamente ineficiente, el tiempo de ejecucion muestra un comportamiento exponencial con respecto de n, para cuando $n = 260$ el procedimiento tarda cerca de 20 segundos!

Recursiva de cola (ms) y Iterativa (ms)



Por otro lado, como era de esperarse, tanto la version de cola como la version iterativa tienen comportamiento lineal con respecto de n, para cuando $n = 10^8$ los procedimientos


tardan cerca de 20ms y 10ms respectivamente. Como tambien era de esperarse, la version iterativa es ligeramente mas rapida que la de cola.



Podemos concluir que en terminos de eficiencia, la version iterativa y la version de cola son muy superiores a la version directa. Adicionalmente podemos recomendar el uso de la version iterativa por encima de la version recursiva, sin embargo podemos tambien destacar que el uso de la version de cola sigue siendo relativamente eficiente considerando que la diferencia entre la version iterativa y la de cola no es tan significativa.



6. Pregunta 6

Para esta pregunta, al igual que en la pregunta 3, hemos seleccionado **Python** (version 3.8). Para la implementacion tendremos una clase **ExpressionTree** que representa a una expresion aritmetica por medio de un arbol binario. Esta clase tendra un metodo **infix** que retorna una representacion **string** de la expresion en orden infijo. Implementaremos el metodo **__call__**, cuando se llame un **ExpressionTree** se retornara el valor de la evaluacion de la expresion que representa.

Implemetaremos tambien dos funciones **postfixToExpressionTree** y **prefixToExpressionTree** que respectivamente construyen un **ExpressionTree** a partir de una expresion en orden *post-fijo* o *prefijo*.

En el archivo **expression.py**  se encuentran las implementaciones de **ExpressionTree**, **prefixToExpressionTree** y **postfixToExpressionTree**.

Para las especificaciones de IO se implemento el archivo **main.py** . En el archivo **README.md**  se encuentran las **instrucciones para ejecutar el programa**.

Para las pruebas unitarias, al igual que en la pregunta 3, se decidio utilizar **unittest** y **coverage**. En el archivo **expression_test.py**  se encuentran las pruebas unitarias de **ExpressionTree**, **prefixToExpressionTree** y **postfixToExpressionTree**. En el archivo **README.md**  se encuentran las **instrucciones para ejecutar los tests** y medir la cobertura.

```
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 % coverage run -m unittest pregunta-6.tests.expression_test
...
-----
Ran 3 tests in 0.002s

OK
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 % coverage report
Name                               Stmts  Miss  Cover
-----
pregunta-6/__init__.py              0      0   100%
pregunta-6/src/__init__.py          0      0   100%
pregunta-6/src/expression.py       65      1    98%
pregunta-6/tests/__init__.py        0      0   100%
pregunta-6/tests/expression_test.py 64      0   100%
-----
TOTAL                               129      1    99%
pintojoao@Joaos-MacBook-Pro CI3641-exam-1 %
```

Aca se deja un screenshot de la salida de las pruebas unitarias y de la medicion de la cobertura.

7. Reto

Con la motivacion de implementar la funcion *jaweno* de forma de que funcione eficientemente en tiempo, utilizamos la **aproximacion de Stirling** 🌀. Esta aproximacion dice lo siguiente:

$$\log_2 n! = n \log_2 n - n \log_2 e + \mathcal{O}(\log_2 n)$$

En otras palabras:

$$\log_2 n! \simeq n \log_2 n - n \log_2 e$$

Teniendo esta expresion, desarrollamos el termino logaritmico de *jaweno* y, luego de manipular, y aplicar la definicion de *fibonacci* obtenemos:

$$\lfloor \log_2 \left(\frac{f_{n+1}}{f_n} \right) \rfloor \simeq \lfloor f_{n+1} \log_2 f_{n+1} - f_{n-1} \log_2 f_{n-1} - f_n \log_2 f_n \rfloor$$

Utilizando esta expresion, para calcular el valor aproximado de *jaweno*(*n*) solo hace falta calcular *f_{n+1}* (y en el proceso obtener *f_n* y *f_{n-1}*) y evaluar la aproximacion.

Implementamos el programa en C++. La implementacion esta en el archivo **reto.cpp** 🌀. Debido a que el término de error de la aproximacion es de orden logaritmico, el programa aproxima bastante bien los valores de *jaweno* para valores de *n* suficientemente grandes. Aca una tabla comparativa entre *jaweno* y el programa implementado:

Aprox	0	0	2	2	4	7	12	20	32	52	85	138	223	361	585	946	1532	2479	4011	6490	10501
Jaweno	0	0	1	1	3	5	10	17	29	49	81	134	219	357	580	941	1526	2473	4005	6483	10494
N	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Esta aproximacion tiene la ventaja de ser muy rapida. Para *n* > 100 sigue por debajo del segundo de tiempo de ejecucion. La implementacion se logro utilizando **246** caracteres (incluyendo caracteres vacios).

Como nota personal. Ha sido un placer participar en este reto. Me siento orgulloso de haber llegado hasta aca, sin embargo, debido a que **solo logre una aproximacion** a *jaweno* y a que lo hice utilizando **246** caracteres, me gustaria admitir que me siento como cuando *Las islas Bahamas* entran al mundial. El que entendió, entendió ❤️.