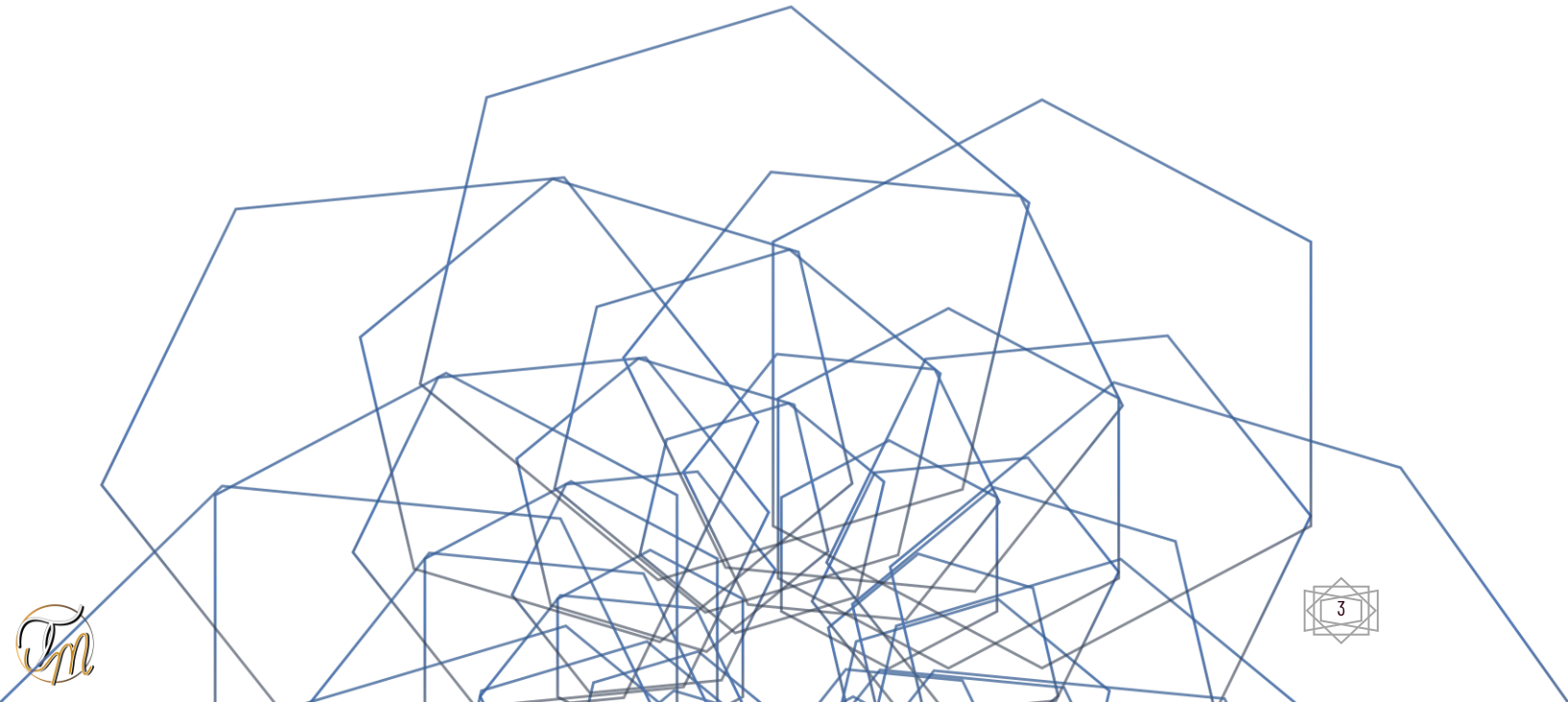
Práctica Número 2

Análisis Temporal

Equipo:

- Jeon Jeong Paola
- Lemus Ruiz Mariana Elizabeth
- López López Oscar Manuel
- Mora Ayala José Antonio

CONTENIDO



INTRODUCCIÓN

Tomando en base el análisis de la primera práctica, pudimos observar los diferentes funcionamientos de los algoritmos de ordenamiento. Para esta práctica pondremos a prueba 5 diferentes algoritmos de búsqueda, analizaremos cómo es que trabaja cada algoritmo y también trabajaremos con el análisis de casos de las complejidades que cada algoritmo puede presentar.

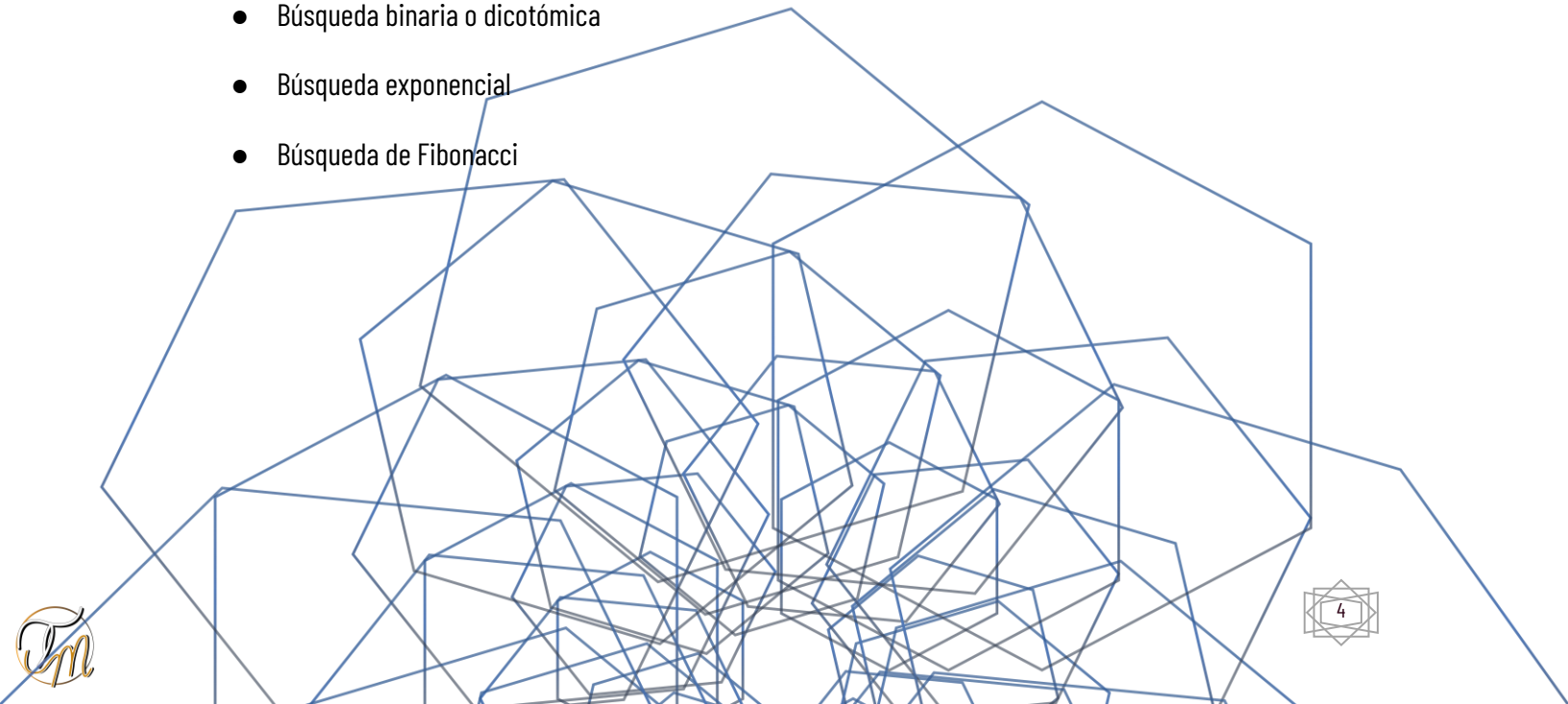
Otro aspecto importante para este trabajo, es utilizar códigos no recursivos y emplear los conocidos hilos, esto es para que se puedan mejorar los tiempos de búsqueda para cada método. Para poder realizar experimentos, utilizaremos el archivo el cuál contiene 10,000,000 números para poder comprobar el funcionamiento de los 5 métodos distintos en la práctica, para cada método igual tendremos que registrar el tiempo que tarda para calcular cuánto se tarda para poder ejecutar cada algoritmo con un "n" tamaño de problema y poder analizar cúal es mejor y por qué.

Por último, utilizaremos gráficas comparativas que ayudan a observar las diferencias para cada método y sacar conclusiones si los datos que sacamos teóricamente, se acercan a los datos que obtuvimos por medio de los experimentos que realizamos durante el desarrollo de esta práctica.

DEFINICIÓN DEL PROBLEMA

Programas los 5 métodos de búsqueda que se piden en la práctica y utilizar la no recursividad y los hilos para analizar la complejidad de los casos para los algoritmos de:

- Búsqueda lineal o secuencial
- Búsqueda de un árbol binario
- Búsqueda binaria o dicotómica
- Búsqueda exponencial
- Búsqueda de Fibonacci



ANÁLISIS TEÓRICO

BÚSQUEDA LINEAL

Descripción

Dado un arreglo de tamaño n, se quiere encontrar un elemento k, para eso, la búsqueda lineal iterará sobre todo su arreglo empezando en la primera posición y terminando en la última, y comparará el elemento actual con el número a buscar, si lo encuentra, regresa el índice de la iteración actual.

Pseudocódigo

```
Algoritmo BúsquedaLineal(A, n, aBuscar)
{
   para i = 0 hasta n - 1 hacer
      si(A[i] == aBuscar)
         regresar i;

   regresar -1;
}
```

Implementación en código

```
int busquedaLineal(int *A, int n, int aBuscar)
{
      int i;
      for(i = 0; i < n; i++)
      {
            if(A[i] == aBuscar)
                  return i;
      }

      return -1;
}
```

Análisis

Para este código utilizaremos como operaciones básicas a:

- Comparaciones con los elementos de $A[\,]$.

Veamos el código:

```
int busquedaLineal(int *A, int n, int aBuscar)
{
	int i;
	for(i = 0; i < n; i++)
	{
		if(A[i] == aBuscar)
			return i;
	}

	return -1;
}
```

Sea j el índice del elemento a buscar, es decir $A[j]$, podemos observar que la variable i se va a iterar hasta llegar a j.

$$valores\ de\ i:$$

$$i = 0,1,2,3,\ldots,n-1$$

Expresemos a i en función de las x operaciones:

$$i = x - 1 \ \ e \ \ i\ debe\ llegar\ hasta\ j$$

$$x - 1 = j$$

$$x = j + 1$$

Mejor Caso

El mejor caso de esta búsqueda es cuando el elemento a buscar se encuentra en la primera posición, es decir $j = 0$:

$$x = 0 + 1$$

$$x = 1$$

Peor caso

El peor caso lo obtenemos cuando el elemento a buscar se encuentra en la última posición del arreglo o no se encuentra en este, es decir $j = n - 1$:

$$x = n - 1 + 1$$

$$x = n$$

Caso medio

Es claro que el elemento $A[j]$ puede ocupar cualquier posición en el arreglo, en otras palabras, $j = 0,1,2,\ldots,n-1$ más el caso donde el elemento no se encuentre (tenemos $n+1$ eventos distintos), así tenemos que el caso promedio está definido por:

$$Caso\ medio = \sum_{k=1}^{h} O_k P_k$$

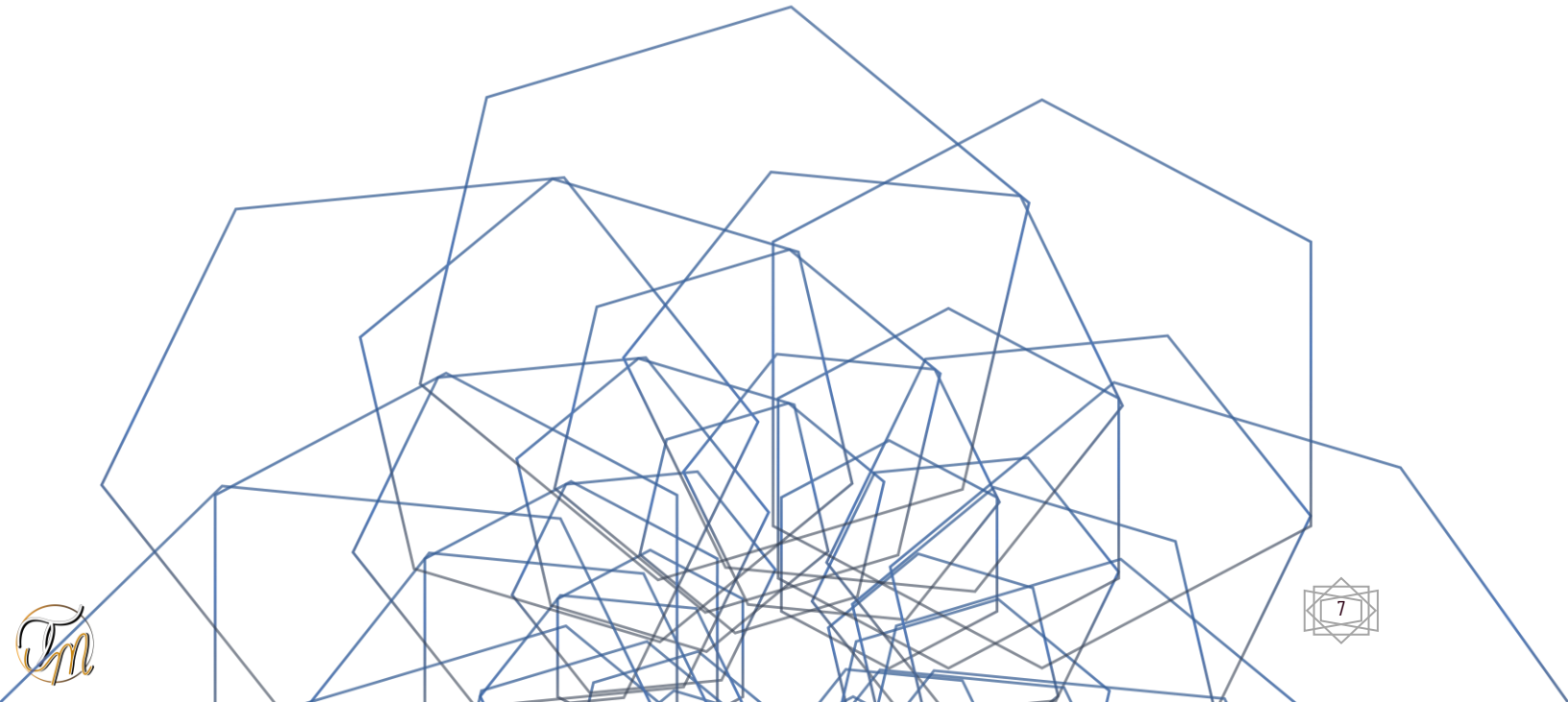
Supongamos que cada evento tiene la misma probabilidad, entonces:

$$P_i = \frac{1}{n+1}$$

Y cada O_k tiene por valor:

$$O_k = x(k) = k + 1$$

$$Caso\ medio = \frac{n}{n+1} + \sum_{k=0}^{n-1} (k+1)\left(\frac{1}{n+1}\right)$$



ÁRBOL BINARIO DE BÚSQUEDA

Descripción

Dado un árbol binario, recorreremos sus nodos dependiendo de si el valor a buscar es máyor, menor o igual a la raíz actual, si es igual devolvemos verdadero, si es mayor, entonces buscaremos por la derecha de la raíz y si es menor, buscaremos por la izquierda.

Pseudocódigo

```
Algoritmo ÁrbolBinarioDeBúsqueda(raiz, aBuscar)
{
   mientras raiz != NULL hacer
      si(raiz.valor == aBuscar)
         regresar verdadero;
      si no, si(raiz.valor < aBuscar)
         raiz = raiz.derecha;
      si no
         raiz = raiz.izquierda;

   regresar falso;
}
```

Implementación en código

```
int arbolBinarioBusqueda(Node *raiz, int aBuscar)
{
      while(raiz != NULL)
      {
            if(raiz->valor == aBuscar)
                  return 1;
            else if(raiz->valor < aBuscar)
                  raiz = raiz->der;
```

```
            else
                  raiz = raiz->izq;
      }

      return 0;
}
```

Análisis

Para este código utilizaremos como operaciones básicas a:

- Comparaciones con el valor de $raiz$.
- Retornos de la función
- Asignaciones de $raiz$.

Veamos el código:

```
int arbolBinarioBusqueda(Node *raiz, int aBuscar)
{
      while(raiz != NULL)
      {
            if(raiz->valor == aBuscar)
                  return 1;
            else if(raiz->valor < aBuscar)
                  raiz = raiz->der;
            else
                  raiz = raiz->izq;
      }

      return 0;
}
```

Recordemos que la altura de un árbol binario está dada por el camino que realiza el árbol hasta su último nodo

$$si\ el\ árbol\ tiene\ una\ altura\ h\ \Rightarrow\ el\ árbol\ tiene\ al\ menos\ h+1\ elementos$$

Ahora, sea x el nivel en el que se encuentra nuestro elemento a buscar, al iniciar la búsqueda en la raíz el algoritmo bajará x niveles hasta llegar al elemento, por lo que podemos notar un comportamiento lineal del algoritmo

$$niveles\ que\ bajará\ el\ árbol = x, \qquad x = 1,2,\dots,n$$

Las operaciones que se realizan en cada nivel que el árbol no encuentra a nuestro elemento son 2 comparaciones y una asignación de $raiz$.

$$\Rightarrow\ en\ x-1\ descensos\ se\ cumplirán\ estas\ operaciones$$

$$\therefore cont = 3(x-1)$$

Luego, en el último descenso, si el elemento se encuentra en el árbol, solamente se ejecutará el primer if y el retorno dentro de este; por otra parte, si el elemento no se encuentra en el arreglo, se ejecutarán las tres operaciones de los otros descensos más el retorno en falso de la función:

$$cont = 3x - 3 + 2 + (2)$$

$$cont = 3x - 1 + (2)$$

Mejor Caso

El mejor caso de esta búsqueda es cuando el elemento a buscar se encuentra en el primer nivel, es decir $x = 1$:

$$cont = 3(1) - 1$$

$$cont = 2$$

Peor caso

El peor caso lo obtenemos cuando tenemos un árbol en el que todos sus elementos están ordenados ascendente/descendentemente y el elmento a buscar no se encuentra en el árbol pero es mayor/menor al último nodo. Es decir $x = n$:

$$cont = 3x - 1 + (2)$$

$$cont = 3n - 1 + 2$$

$$cont = 3n + 1$$

Caso medio

Es claro que el nivel x puede tomar cualquier valor en el rango $[1, n]$ (más el caso donde no se encuentra en el árbol), así tenemos que el caso promedio está definido por:

$$Caso\ medio = \sum_{k=1}^{h} O_k P_k$$

Supongamos que cada evento tiene la misma probabilidad, entonces:

$$P_i = \frac{1}{n+1}$$

Y cada O_k tiene por valor:

$$O_k = cont(k) = 3k - 1$$

$$Caso\ medio = \frac{3n+1}{n+1} + \sum_{k=1}^{n} (3k-1)\left(\frac{1}{n+1}\right)$$

Búsqueda Binaria

Descripción

Dado un arreglo ordenado ascendentemente, la búsqueda binaria consiste en agarrar al primer y último elemento del arreglo para dividirlo a la mitad, se comparan los tres límites con el número a buscar, luego, si ninguno resulta igual al número a buscar, se compara este último con el punto medio; si es menor, entonces el límite superior pasará a tomar el valor del punto medio; si no es mayor que el punto medio, entonces el límite inferior toma el valor del punto medio. Una vez que tengamos los nuevos valores del límite inferior y/o superior, se vuelve a realizar la división y por ende el proceso de la búsqueda binaria sobre el nuevo rango.

Pseudocódigo

```
Algoritmo BúsquedaBinaria(A, inicio, fin, aBuscar)
{
   izquierda = inicio;
   derecha = fin;

   i = 1;
   mientras(Centro != izquierda && centro != derecha)
      centro = (izquierda + derecha) / 2;

            si(A[izquierda] == aBuscar)
                  regresar izquierda;
            si(A[derecha] == aBuscar)
                  regresar derecha;
            si(A[centro] == aBuscar)
                  regresar centro;

            si(A[centro] > aBuscar)
                  derecha = centro;
            si no
                  izquierda = centro;

   regresar -1;
}
```

Implementación en código

```
int busquedaBinaria(int *A, int inicio, int fin, int aBuscar)
{
	int izquierda = inicio;
	int derecha = fin;
	int centro = -1;

	while(centro != izquierda && centro != derecha)
	{
		centro = (izquierda + derecha) / 2;

		if(A[izquierda] == aBuscar)
			return izquierda;
		if(A[derecha] == aBuscar)
			return derecha;
		if(A[centro] == aBuscar)
			return centro;

		if(A[centro] > aBuscar)
			derecha = centro;
		else
			izquierda = centro;
	}

	return -1;
}
```

Análisis

Para este código utilizaremos como operaciones básicas a:

- Comparaciones con los elementos de $A[\ ]$.
- Comparaciones entre $centro$ e $izquierda/derecha$.
- Asignaciones de $centro, izquierda$ y $derecha$.
- Valores regresados de la función $busquedaBinaria$.

Veamos el bloque principal del código:

```
int busquedaBinaria(int *A, int inicio, int fin, int aBuscar)
{
	int izquierda = inicio;
	int derecha = fin;
	int centro = -1;

	while(centro != izquierda && centro != derecha)
	{
		centro = (izquierda + derecha) / 2;

		if(A[izquierda] == aBuscar)
			return izquierda;
		if(A[derecha] == aBuscar)
			return derecha;
		if(A[centro] == aBuscar)
			return centro;

		if(A[centro] > aBuscar)
			derecha = centro;
		else
			izquierda = centro;
	}

	return -1;
}
```

El arreglo posee $n$elementos (es decir, está indexado de 0 a $n-1$). Definamos a $j$como el índice del número a buscar $A[j]$.

Observamos que en cada iteración del ciclo `while` el bloque realiza 3 comparaciones: una en el límite inferior del rango, otra en el centro y otra en el límite superior, luego, si no se encontró el valor, divide el rango a la mitad y vuelve a repetir el proceso. El rango de búsqueda está dado por la diferencia entre el límite superior y el límite inferior (y el primer rango abarca todo el arreglo), veamos cómo cambia el rango a través de las iteraciones.

$$Rango_1 = (n-1) - 0$$

$$Rango_2 = [n-1]/2$$

$$Rango_3 = [n-1]/4$$

$$\vdots$$

En general:

$$Rango_x = [n-1]/2^{x-1}$$

Hasta que el límite superior, inferior o su mitad sea igual a j.

Podemos notar que el número de iteraciones se puede encontrar si conocemos el índice j y el $Rango_x$:

$$2^{x-1} = (n-1)/Rango_x$$

$$log_2(2^{x-1}) = log_2([n-1]/Rango_x)$$

$$x - 1 = log_2\left[\frac{n-1}{Rango_x}\right] = log_2(n-1) - log_2(Rango_x)$$

$$x = log_2(n-1) - log_2(Rango_x) + 1$$

Este es el número de iteraciones en el ciclo `while` , sin embargo como en la última iteración alguno de los tres if's se cumplirá (si $A[j]$ se encuentra en el arreglo), el número de ciclos whiles que ejecutará todas las operaciones son:

$$x = log_2(n-1) - log_2(Rango_x)$$

Ahora veamos cuántas operaciones posee este bloque:

Al inicio tenemos 3 asignaciones que siempre se ejecutarán.

$$cont = 3$$

Luego, el inicio del ciclo `while` consiste de las 2 comparaciones `centro != izquierda` y `centro != derecha`, estas se ejecutarán $2(log_2(n-1) - log_2(Rango_x) + 1)$ veces o $2(log_2(n-1) - log_2(Rango_x) + 2)$ veces si el número a buscar no se encuentra en el arreglo.

$$cont = 3 + 2log_2(n-1) - 2log_2(Rango_x) + 2 + (2)$$

Dentro del ciclo while se ejecutarán 2 asignaciones y 4 comparaciones en cada iteración en que las condiciones de los if's no se cumplan, y como vimos anteriormente, estas iteraciones se ejecutan $log_2(n-1) - log_2(Rango_x)$ veces.

$$cont = 2log_2(n-1) + 5 - 2log_2(Rango_x) + (2) + 6log_2(n-1) - 6log_2(Rango_x)$$

Finalmente, en la última iteración el número de operaciones variará de 3 (1 comparación, 1 asignación y 1 retorno) hasta 7 (el caso donde el número a buscar no está en el arreglo y por ende se ejecuta toda la iteración para que en la siguiente las condiciones del `while` ya no se cumpla y se ejecute la instrucción `return -1`).

$$cont = 8log_2(n-1) + 5 - 8log_2(Rango_x) + (2) + l$$

$$donde\ l \in [3, 7]$$

Mejor caso

El mejor caso de la búsqueda binaria es cuando el elemento a buscar se encuentra en la primera posición del arreglo, es decir, que el rango será el original y en la primera iteración se obtendrá al número buscado:

$$cont = f(n) = 8log_2(n-1) + 5 - 8log_2(n-1) + l$$

$$donde\ l = 3, pues\ en\ la\ primera\ condición\ devuelve\ verdadero$$

$$f(n) = 5 + 3 = 8$$

Peor caso

El peor caso es cuando el número a buscar no se encuentra en el arreglo y además dicho número es mayor que el último elemento en el arreglo (es decir que $j > n-1$, sin embargo como la búsqueda binaria tiene como límite $n-1$, supongamos que $j = n-1$ y luego sumaremos las operaciones extras de esta búsqueda para el caso donde el elemento no se encuentra en el arreglo), así la búsqueda binaria llegará a su última iteración, es decir, el menor $Rango_x$ posible, y como los valores de $Rango_x$ se dividen a la mitad, el menor valor posible es $Rango_x = 1$:

$$cont = 8log_2(n-1) + 5 - 8log_2(Rango_x) + (2) + l$$

$$donde\ l = 7, pues\ estamos\ hablando\ del\ peor\ caso$$

$$cont = 8log_2(n-1) + 5 - 8log_2(1) + 2 + 7$$

$$cont = f(n) = 8log_2(n-1) + 14$$

Caso medio

Ya vimos dos 'caminos' a tomar en cuenta para el caso medio, sin embargo este algoritmo tiene infinidad de variantes distintas que distinguir caminos claros muy complicado, lo que sí sabemos es que $Rango_x$ varía desde $n-1$ hasta 1:

$$\Longrightarrow \quad Rango_x = n-1, \frac{n-1}{2}, \frac{n-1}{4}, \ldots, 1$$

Como vimos anteriormente, $Rango_x$está dado por:

$$Rango_x = [n-1]/2^{x-1}$$

Encontremos el valor máximo de x, es decir cuando el rango vale 1:

$$1 = [n-1]/2^{x-1}$$

$$2^{x-1} = [n-1]$$

$$log_2(2^{x-1}) = log_2[n-1]$$

$$x = log_2(n-1) + 1$$

$$\therefore habrá\ log_2(n-1)+1\ instancias\ del\ rango$$

También supondremos que cada uno de los caminos tiene la misma probabilidad de ocurrencia:

$$\#total\ de\ instancias = 2 + log_2(n-1) + 1$$

$$\therefore P(j) = 1/(2 + log_2(n-1) + 1)$$

La O_x para nuestro caso supuesto es igual a $cont$ sustituyendo $Rango_x = [n-1]/2^{x-1}$, y $l = 4$:

$$O_x = cont = 8log_2(n-1) + 5 - 8log_2([n-1]/2^{x-1}) + 4$$

$$O_x = 8log_2(n-1) + 9 - 8log_2([n-1]/2^{x-1})$$

Y x ocupará el rango $[1, log_2(n-1)+1]$.

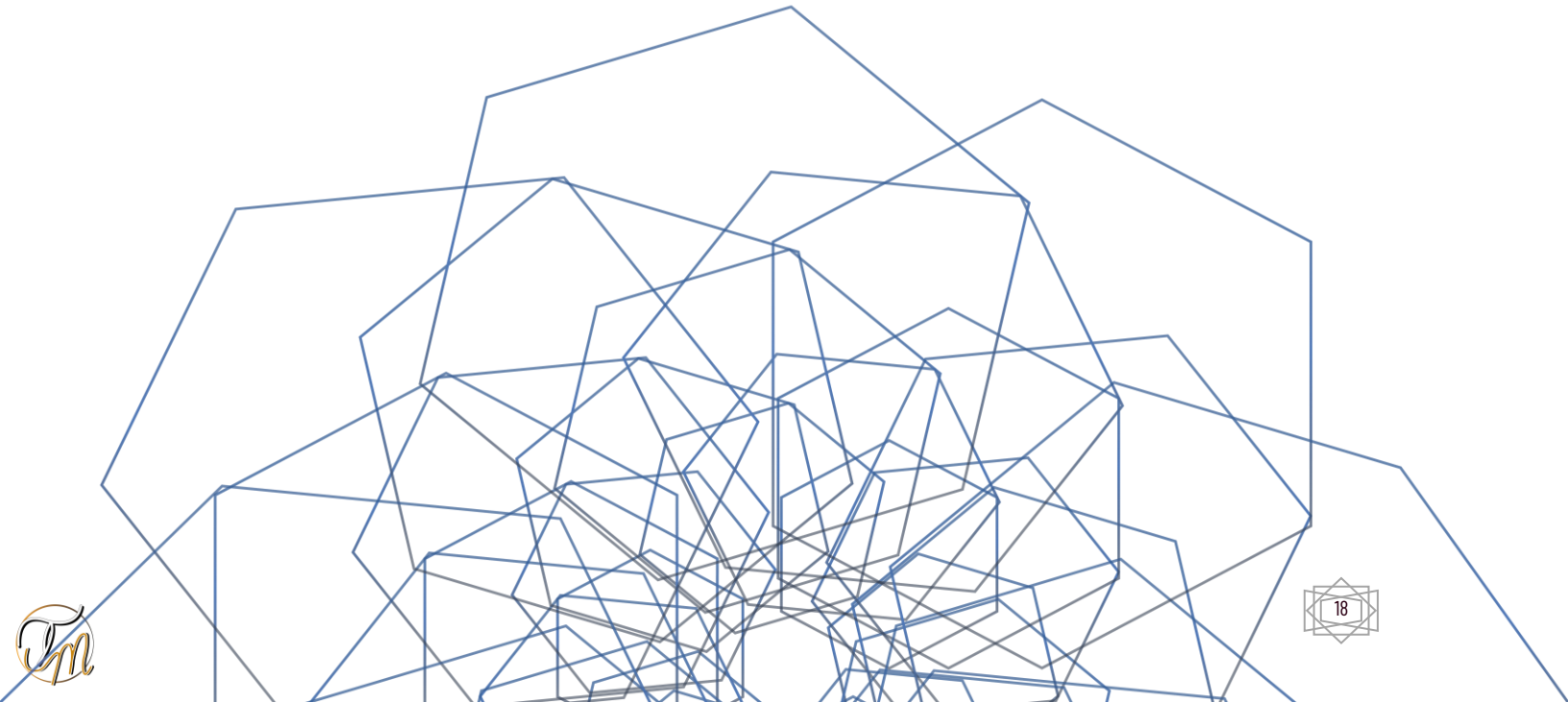
El caso medio está dado por:

$$Caso\ medio = \sum_{k=1}^{h} O_k P_k$$

Sustituyendo:

$$Caso\ medio = (8log_2(n-1)+14)(P_j) + \sum_{x=1}^{log_2(n-1)+1} O_x P_j$$

$$donde\ P_j = 1/(2+log_2(n-1)+1)\ y\ O_x$$
$$= 8log_2(n-1) + 9 - 8log_2([n-1]/2^{x-1})$$



Búsqueda Exponencial

Descripción

Dado un arreglo ordenado ascendentemente, realizaremos la búsqueda en dos pasos:

- Primero encontraremo el rango en el arreglo donde el elemento puede estar presente
- Y después aplicaremos una búsqueda binaria en ese rango

Para encontrar el rango primero compararemos al último elemento del primer sub arreglo de tamaño 1 en el arreglo original con el valor que estamos buscado, luego haremos los mismo para un sub arreglo de tamaño dos a partir del siguiente elemento, luego con un subarreglo de tamaño cuatro y así sucesivamente hasta que el último elemento del sub arreglo actual sea mayor o igual al número que buscamos; una vez que tengamos a este elemento, utilizaremos su índice i y al índice $i/2$ (pues en la última iteración el elemento con ese índice no era mayor o igual al elemento que buscamos) para realizar una búsqueda binaria sobre ese rango.

Pseudocódigo

```
Algoritmo BúsquedaExponencial(A, n, aBuscar)
{
   si(A[0] == aBuscar)
      regresar 0;

   i = 1;
   mientras(i < n && A[i] <= aBuscar)
      i = i * 2;

   regresar BúsquedaBinaria(A, i / 2, menor(i, n - 1), aBuscar);
}
```

Implementación en código

```
int busquedaExponencial(int *A, int n, int aBuscar)
{
```

```
	if(A[0] == aBuscar)
		return 0;

	int i = 1;
	while(i < n && A[i] <= aBuscar)
		i *= 2;

	return busquedaBinaria(A, i / 2, menor(i, n-1), aBuscar);
}

int busquedaBinaria(int *A, int inicio, int fin, int aBuscar)
{
	int izquierda = inicio;
	int derecha = fin;
	int centro = -1;

	while(centro != izquierda && centro != derecha)
	{
		centro = (izquierda + derecha) / 2;

		if(A[izquierda] == aBuscar)
			return izquierda;
		if(A[derecha] == aBuscar)
			return derecha;
		if(A[centro] == aBuscar)
			return centro;

		if(A[centro] > aBuscar)
			derecha = centro;
		else
			izquierda = centro;
	}

	return -1;
}
```

Análisis

Para este código utilizaremos como operaciones básicas a:

- Comparaciones con los elementos de $A[\ ]$.

- Asignación/incrementos de i.
- Comparaciones entre $centro$ e $izquierda/derecha$.
- Asignaciones de $centro, izquierda$ y $derecha$.
- Valores regresados de la función $busquedaBinaria$.

Veamos la primer parte del código:

```
int busquedaExponencial(int *A, int n, int aBuscar)
{
	if(A[0] == aBuscar)
		return 0;

	int i = 1;
	while(i < n && A[i] <= aBuscar)
		i *= 2;

	return busquedaBinaria(A, i / 2, menor(i, n-1), aBuscar);
}
```

Iniciamos el conteo de operaciones $cont = 0$.

Para comenzar, identificamos dos operaciones que siempre se realizan al inicio del bloque: la comparación `A[0] == aBuscar` y `int i = 1`.

$$cont = 2$$

Veamos cuántas veces se repite el bucle `while`, para eso, analicemos el comportamiento de `i`:

$$Sea\ j = índice\ del\ elemento\ a\ buscar$$

$$i = 1, 2, 4, 8, \ldots \quad hasta\ i < n\ ó\ i \leq j$$

Describamos a i en función de los k saltos que da:

$$f(k) = 2^{k-1}\ y\ f(k) < n\ ó\ f(k) \leq j$$

Sin perder generalidad, simplemente supongamos que:

$$f(k) \leq j < n$$

$$\therefore \ f(k) = j + comp$$

será el valor que romperá el ciclo y nos dará el número de saltos hasta llegar a ese va

y donde comp: es el valor que hace entera a la función, en otras palabras:

$$2^{k-1} = j + comp, \qquad 2^{k-2} < j \leq 2^{k-1} \ y \ 0 \leq comp \leq (2^{k-2} - 1) = \frac{2^{k-1}}{2} - 1$$

¿Cómo obtenemos a $comp$ en términos de j?

$$como: 0 \leq comp \leq \frac{2^{k-1}}{2} - 1$$

$$entonces \ \Rightarrow \ log_2(j + comp) = techo(log_2(j))$$

$$\therefore \ 2^{techo(log_2(j))} = j + comp$$

$$\Rightarrow \ comp = 2^{techo(log_2(j))} - j$$

Resolvamos para k:

$$2^{k-1} = j + comp$$

$$log_2(2^{k-1}) = log_2(j + comp)$$

$$k - 1 = log_2(j + comp)$$

$$k = log_2(j + comp) + 1$$

Así, concluimos que las comparaciones `A[i] <= aBuscar` se repetirán $log_2(j + comp) + 2$ veces (la extra es cuando la condición `i < n` o la `A[i] <= aBuscar` ya no se cumple), mientras que los incrementos `i *= 2` lo harán $log_2(j + comp) + 1$ veces.

$$cont = 2 + log_2(j + comp) + 2 + log_2(j + comp) + 1$$

$$cont = 2log_2(j + comp) + 5$$

Después se aplica la búsqueda binaria en el rango de $i/2$ a i, pero como i está expresado en términos de los k saltos y ya encontramos el valor de k, sustituimos para ver el rango real:

$$Rango = (\frac{i}{2}\ a\ i)$$

$$Rango = (\frac{2^{k-1}}{2}\ a\ 2^{k-1}) = (2^{k-2}\ a\ 2^{k-1})$$

$$si\ k = log_2(j + comp) + 1$$

$$Rango = (2^{log_2(j+comp)+1-2}\ a\ 2^{log_2(j+comp)+1-1})$$

$$Rango = (2^{log_2(j+comp)-1}\ a\ 2^{log_2(j+comp)})$$

$$Rango = (\frac{j + comp}{2}\ a\ j + comp)$$

Una vez que ya conocemos el rango de la búsqueda binaria, analicemos su código:

```
int busquedaBinaria(int *A, int inicio, int fin, int aBuscar)
{
	int izquierda = inicio;
	int derecha = fin;
	int centro = -1;

	while(centro != izquierda && centro != derecha)
	{
		centro = (izquierda + derecha) / 2;

		if(A[izquierda] == aBuscar)
			return izquierda;
		if(A[derecha] == aBuscar)
			return derecha;
		if(A[centro] == aBuscar)
			return centro;

		if(A[centro] > aBuscar)
			derecha = centro;
		else
			izquierda = centro;
	}
```

```
        return -1;
}
```

Observamos que en cada iteración del ciclo `while` el bloque realiza 3 comparaciones: una en el límite inferior del rango, otra en el centro y otra en el límite superior, luego, si no se encontró el valor, divide el rango a la mitad y vuelve a repetir el proceso, veamos cómo cambia el rango a través de las iteraciones:

$$Rango_1 = (j + comp) - \frac{j + comp}{2}$$

$$Rango_2 = \left[(j + comp) - \frac{j + comp}{2}\right]/2$$

$$Rango_3 = \left[(j + comp) - \frac{j + comp}{2}\right]/4$$

$$\vdots$$

En general:

$$Rango_x = \left[\frac{j + comp}{2}\right]/2^{x-1}$$

Hasta que el límite superior, inferior o su mitad sea igual a j.

Podemos notar que el número de iteraciones se puede encontrar si conocemos el índice j y el $Rango_x$:

$$2^{x-1} = \left[\frac{j + comp}{2}\right]/Rango_x$$

$$log_2(2^{x-1}) = log_2\left(\left[\frac{j + comp}{2}\right]/Rango_x\right)$$

$$x - 1 = log_2\left[\frac{j + comp}{2Rango_x}\right] = log_2(j + comp) - log_2(2Rango_x)$$

$$x = log_2(j + comp) - log_2(Rango_x)$$

Este es el número de iteraciones en el ciclo `while`, sin embargo como en la última iteración alguno de los tres if's se cumplirá (si $A[j]$ se encuentra en el arreglo), el número de ciclos whiles que ejecutará todas las operaciones son:

$$x = log_2(j + comp) - log_2(Rango_x) - 1$$

Ahora veamos cuántas operaciones posee este bloque:

Al inicio tenemos 3 asignaciones que siempre se ejecutarán.

$$cont = 2log_2(j + comp) + 5 + 3$$

Luego, el inicio del ciclo `while` consiste de las 2 comparaciones `centro != izquierda` y `centro != derecha`, estas se ejecutarán $2(log_2(j + comp) - log_2(Rango_x))$ veces o $2(log_2(j + comp) - log_2(Rango_x) + 1)$ veces si el número a buscar no se encuentra en el arreglo.

$$cont = 2log_2(j + comp) + 8 + 2log_2(j + comp) - 2log_2(Rango_x) + (2)$$

Dentro del ciclo while se ejecutarán 2 asignaciones y 4 comparaciones en cada iteración en que las condiciones de los if's no se cumplan, y como vimos anteriormente, estas iteraciones se repiten $log_2(j + comp) - log_2(Rango_x) - 1$ veces.

$$cont = 4log_2(j + comp) + 8 - 2log_2(Rango_x) + (2) + 6log_2(j + comp) - 6log_2(Rango_x) - 6$$

Finalmente, en la última iteración el número de operaciones variará de 3 (1 comparación, 1 asignación y 1 retorno) hasta 7 (el caso donde el número a buscar no está en el arreglo y por ende se ejecuta toda la iteración para que en la siguiente las condiciones del `while` ya no se cumpla y se ejecute la instrucción `return -1`).

$$cont = 10log_2(j + comp) + 2 - 8log_2(Rango_x) + (2) + l$$

$$donde\ l \in [3, 7]$$

Mejor caso

El mejor caso de la búsqueda exponencial es cuando el elemento a buscar se encuentra en la primera posición del arreglo, de esta forma el algoritmo solo ejecutará una operación.

$$\Longrightarrow f(n) = 1$$

Peor caso

El peor caso es cuando el número a buscar no se encuentra en el arreglo y además dicho número es mayor que el último elemento en el arreglo (es decir que $j > n - 1$, sin embargo como la búsqueda binaria tiene como límite $n - 1$, supongamos que $j = n - 1$ y luego sumaremos las operaciones extras de esta búsqueda para el caso donde el elemento no se encuentra en el arreglo), así primero se ejecutarán todos los posibles incrementos de i y

después la búsqueda binaria llegará a su última iteración, es decir, el menor $Rango_x$ posible, y como los valores de $Rango_x$ se dividen a la mitad, el menor valor posible es $Rango_x = 1$:

$$sustituyamos\ en\ cont\ para\ j = n - 1$$

$$recordemos\ que\ comp = 2^{techo(log_2(j))} - j$$

$$Tenemos\ que:$$

$$cont\ = 10log_2(j + comp) + 2 - 8log_2(Rango_x) + 2 + l$$

$$donde\ l = 7, pues\ estamos\ hablando\ del\ peor\ caso$$

$$cont\ = 10log_2(j + comp) + 11 - 8log_2(Rango_x)$$

$$cont\ = 10log_2\left(j + 2^{techo(log_2(j))} - j\right) + 11 - 8log_2(1)$$

$$cont\ = 10log_2\left(2^{techo(log_2(j))}\right) + 11$$

$$cont\ = 10techo[log_2(j)] + 11$$

$$cont = f(n) = 10techo[log_2(n-1)] + 11$$

Caso medio

Ya vimos dos 'caminos' a tomar en cuenta para el caso medio, sin embargo este algoritmo tiene infinidad de variantes distintas que distinguir caminos claros muy complicado, lo que sí sabemos es que $Rango_x$ varía desde $\frac{j+comp}{2}$ hasta 1:

$$\Longrightarrow\ Rango_x = \frac{j + comp}{2}, \frac{\frac{j + comp}{2}}{2}, \frac{\frac{j + comp}{2}}{4}, \ldots, 1$$

Como vimos anteriormente, $Rango_x$está dado por:

$$Rango_x = \left[\frac{j + comp}{2}\right] / 2^{x-1}$$

Encontremos el valor máximo de x, es decir cuando el rango vale 1:

$$1 = \left[\frac{j + comp}{2}\right] / 2^{x-1}$$

$$2^{x-1} = \left\lceil \frac{j + comp}{2} \right\rceil$$

$$log_2(2^{x-1}) = log_2 \left\lceil \frac{j + comp}{2} \right\rceil$$

$$x = log_2(\frac{j + comp}{2}) + 1$$

$$x = log_2(j + comp)$$

$$\therefore habrá\ log_2(j + comp)\ instancias\ del\ rango$$

También supondremos que cada uno de los caminos tiene la misma probabilidad de ocurrencia:

$$\#total\ de\ instancias = 2 + log_2(j + comp)$$

$$\therefore P(j) = 1/(2 + log_2(j + comp))$$

La O_x para nuestro caso supuesto es igual a $cont$ sustituyendo $Rango_x = \left\lceil \frac{j+comp}{2} \right\rceil /2^{x-1}$, y $l = 4$:

$$O_x = cont = 10log_2(j + comp) + 2 - 8log_2(Rango_x) + 4$$

$$O_x = 10log_2(j + comp) + 6 - 8log_2(\left\lceil \frac{j + comp}{2} \right\rceil /2^{x-1})$$

Y x ocupará el rango $[1, log_2(j + comp)]$.

El caso medio está dado por:

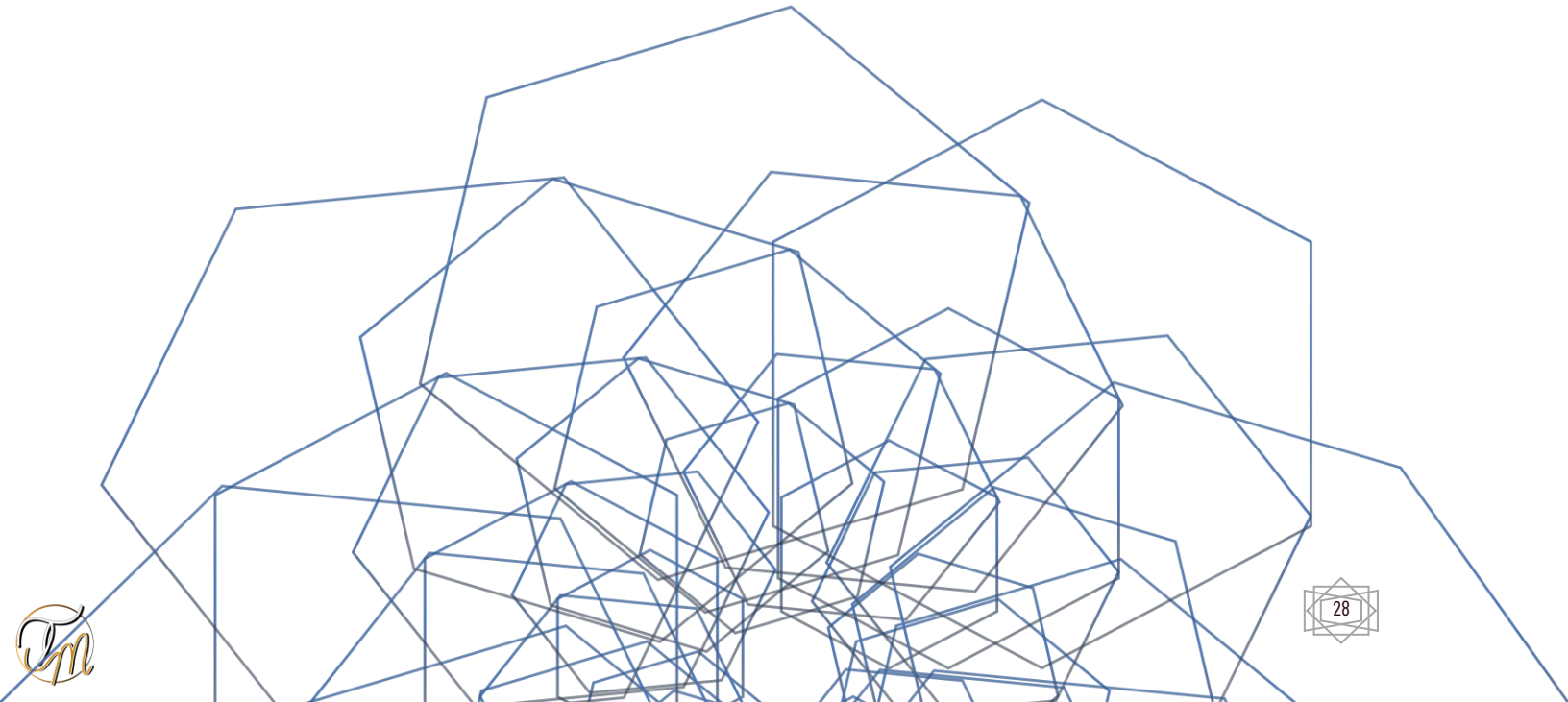
$$Caso\ medio = \sum_{k=1}^{h} O_k P_k$$

Sustituyendo:

$$Caso\ medio = 1(P_j) + (10techo[log_2(n-1)] + 11)(P_j) + \sum_{x=1}^{log_2(j+comp)} O_x P_j$$

$$donde\ P_j = 1/(2 + log_2(j + comp))\ y\ O_x$$

$$= 10log_2(j + comp) + 6 - 8log_2(\left[\frac{j + comp}{2}\right]/2^{x-1})$$



Búsqueda de Fibonacci

Descripción

Dado un arreglo ordenado, se segmenta en subarreglos de una longitud que coincide con los valores de la serie Fibonacci. Posteriormente se empieza a comparar el valor a buscar con los límites de los subarreglos empezando por el más grande, si el numero a buscar es menor a su límite inferior o a la mitad del rango, se vuelve a dividir el arreglo en elementos más pequeños que siguen respetando la serie de Fibonacci

Implementación en código

```
int busquedaFibo(int *A, int n, int aBuscar)
{
	int fiboM2 = 0;
	int fiboM1 = 1;
	int fibo = fiboM2 + fiboM1;

	while(fibo < n)
	{
		fiboM2 = fiboM1;
		fiboM1 = fibo;
		fibo = fiboM2 + fiboM1;
	}

	int comp = -1;

	while(fibo > 1)
	{
		int i = menor(comp + fiboM2, n-1);

		if(A[i] == aBuscar)
			return i;
		else if(A[i] < aBuscar)
		{
			fibo = fiboM1;
			fiboM1 = fiboM2;
			fiboM2 = fibo - fiboM1;
			comp = i;
```

```
            }
            else
            {
                  fibo = fiboM2;
                  fiboM1 -= fiboM2;
                  fiboM2 = fibo - fiboM1;
            }
      }

      if(fiboM1 && A[comp + 1] == aBuscar)
            return comp + 1;

      return -1;
}
```

Análisis

Para este código utilizaremos como operaciones básicas a:

- Asignaciones de $fibo, fiboM1, fiboM2$ y $comp$.
- Comparaciones con los elementos de $A[\,]$.
- Valores retornados por la función.

Veamos la primer parte del código:

```
      int fiboM2 = 0;
      int fiboM1 = 1;
      int fibo = fiboM2 + fiboM1;

      while(fibo < n)
      {
            fiboM2 = fiboM1;
            fiboM1 = fibo;
            fibo = fiboM2 + fiboM1;
      }
```

```
int comp = -1;
```

Vemos que esta primera parte tenemos los dos primeros términos de la serie de Fibonacci y después continúa esta serie hasta que el valor de esta supere o sea igual a n, sin embargo como n puede variar, el número de veces que se repite el ciclo es una variable, así:

$$sea\ x = \#\ de\ veces\ que\ se\ repite\ el\ ciclo\ while$$

$$\Rightarrow el\ valor\ de\ la\ serie\ de\ fibonacci\ será\ F(x+2)$$

Recordemos que la serie de Fibonacci está dada por:

$$F(n) = F(n-1) + F(n-2)$$

Y que podemos aproximar a los valores $F(n-1)$ y $F(n-2)$ como:

$$F(n-1) \simeq \frac{2}{3}F(n)$$

$$F(n-2) \simeq \frac{1}{3}F(n)$$

Entonces, si sabemos que el código dentro del ciclo `while` se repetirá x veces, y en cada iteración se realizan 3 asignaciones, además de que tenemos otras 4 asignaciones que son estáticas (siempre se realizan), el número de operaciones totales en este bloque es de:

$$cont = 3x + 4$$

Ahora veamos el segundo bloque de este algoritmo:

```
while(fibo > 1)
{
	int i = menor(comp + fiboM2, n-1);

	if(A[i] == aBuscar)
		return i;
	else if(A[i] < aBuscar)
	{
```

```
                fibo = fiboM1;
                fiboM1 = fiboM2;
                fiboM2 = fibo - fiboM1;
                comp = i;
        }
        else
        {
                fibo = fiboM2;
                fiboM1 -= fiboM2;
                fiboM2 = fibo - fiboM1;
        }
}

if(fiboM1 && A[comp + 1] == aBuscar)
        return comp + 1;

return -1;
```

Nuevamente, el número de iteraciones que dará este ciclo es variable pero si nos fijamos bien podemos observar que a diferencia de x, este ciclo tiene un límite definido de iteraciones, pues `fibo` está constantemente decreciendo y debe llegar a 1 para que el ciclo se detenga, veamos las dos formas en que puede decrementar de valor `fibo` en las iteraciones empezando por la primera:

En `fibo = fiboM1` puede decrecer al valor aproximado $\frac{2}{3}F(x+2)$

Ó en `fibo = fiboM2` puede decrecer al valor aproximado $\frac{1}{3}F(x+2)$

Si en la primera iteración no se encontró el elemento, en la segunda ahora se puede decrementar (principalmente) a:

En `fibo = fiboM1` puede decrecer al valor aproximado $\frac{4}{9}F(x+2)$

Ó en `fibo = fiboM2` puede decrecer al valor aproximado $\frac{1}{9}F(x+2)$

En general, podemos expresar el valor de `fibo` de a acuerdo a la $y-ésima$ iteración:

$$fibo = (\frac{2}{3})^{y-1}F(x+2)$$

$$\text{ó } fibo = (\frac{1}{3})^{y-1}F(x+2)$$

Despejemos a y:

$$(\frac{3}{2})^{y-1} = \frac{F(x+2)}{fibo}$$

$$\text{ó } 3^{y-1} = \frac{F(x+2)}{fibo}$$

$$log_{\frac{3}{2}}((\frac{3}{2})^{y-1}) = log_{\frac{3}{2}}(\frac{F(x+2)}{fibo})$$

$$\text{ó } log_3(3^{y-1}) = log_3(\frac{F(x+2)}{fibo})$$

$$y = log_{\frac{3}{2}}(\frac{F(x+2)}{fibo}) + 1$$

$$\text{ó } y = log_3(\frac{F(x+2)}{fibo}) + 1$$

Si se cumple el primer caso, las operaciones que se realizarán en cada iteración (mientras no se encuentre el elemento a buscar) serán dos comparaciones y cinco asignaciones.

En el segundo caso serán dos comparaciones y cuatro asignaciones.

Finalmente, en la iteración donde se encuentre el número se realizan una asignación, una comparación y un retorno.

$$cont = 3x + 4 + 7(log_{\frac{3}{2}}(\frac{F(x+2)}{fibo})) + 3$$

$$\text{ó } cont = 3x + 4 + 6(log_3(\frac{F(x+2)}{fibo})) + 3$$

$$cont = 3x + 7(log_{\frac{3}{2}}(\frac{F(x+2)}{fibo})) + 7$$

$$\text{ó } cont = 3x + 6(log_3(\frac{F(x+2)}{fibo})) + 7$$

Mejor caso

El mejor caso ocurre cuando el elemento a buscar es el primero que compra en la primera iteración del `while(fibo > 1)`, es decir cuando $fibo = F(x+2)$, ya que no se ha asignado a `fiboM1` o a `fiboM2`:

$$cont = 3x + 7(log_{\frac{3}{2}}(\frac{F(x+2)}{F(x+2)})) + 7$$

$$\text{ó } cont = 3x + 6(log_3(\frac{F(x+2)}{F(x+2)})) + 7$$

$$cont = 3x + 7(log_{\frac{3}{2}}(1)) + 7$$

$$\text{ó } cont = 3x + 6(log_3(1)) + 7$$

$$cont = 3x + 7(0) + 7$$

$$\text{ó } cont = 3x + 6(0) + 7$$

$$cont = 3x + 7 \; para \; cualquiera \; de \; los \; dos \; casos$$

Peor caso

El peor caso ocurre cuando el elemento a buscar se encuentra en el máximo número de iteraciones, es decir, cuando $fibo = 1$, o definitivamente no se encuentra, ya que en ambos casos el código ejecuta dos instrucciones extra, una comparación, y un retorno:

$$cont = 3x + 7(log_{\frac{3}{2}}(\frac{F(x+2)}{1})) + 9$$

$$\text{ó } cont = 3x + 6(log_3(\frac{F(x+2)}{1})) + 9$$

$$cont = 3x + 7(log_{\frac{3}{2}}(F(x+2))) + 9$$

$$\text{ó } cont = 3x + 6(log_3(F(x+2))) + 9$$

Caso Medio

Podemos observar que principalmente tenemos dos caminos, y a su vez, el número de iteraciones en el ciclo principal de estos caminos dependen del valor actual de $fibo$, teniendo un valor mínimo de una iteración y un máximo cuando $fibo = 1$, veamos cuántas iteraciones representan este caso para cada ciclo:

$$y = log_{\frac{3}{2}}(\frac{F(x+2)}{1}) + 1$$

$$\text{ó } y = log_{3}(\frac{F(x+2)}{1}) + 1$$

$$y = log_{\frac{3}{2}}(F(x+2)) + 1$$

$$\text{ó } y = log_{3}(F(x+2)) + 1$$

Por lo que el número total de casos es la suma de estos dos valores más los dos casos que implican que el número a buscar no se encuentre en el arreglo:

$$\#\ de\ instancias\ totales = log_{\frac{3}{2}}(F(x+2)) + log_{3}(F(x+2)) + 4$$

Si suponemos que cada caso tiene la misma probabilidad, entonces:

$$P_i = 1/(log_{\frac{3}{2}}(F(x+2)) + log_{3}(F(x+2)) + 4)$$

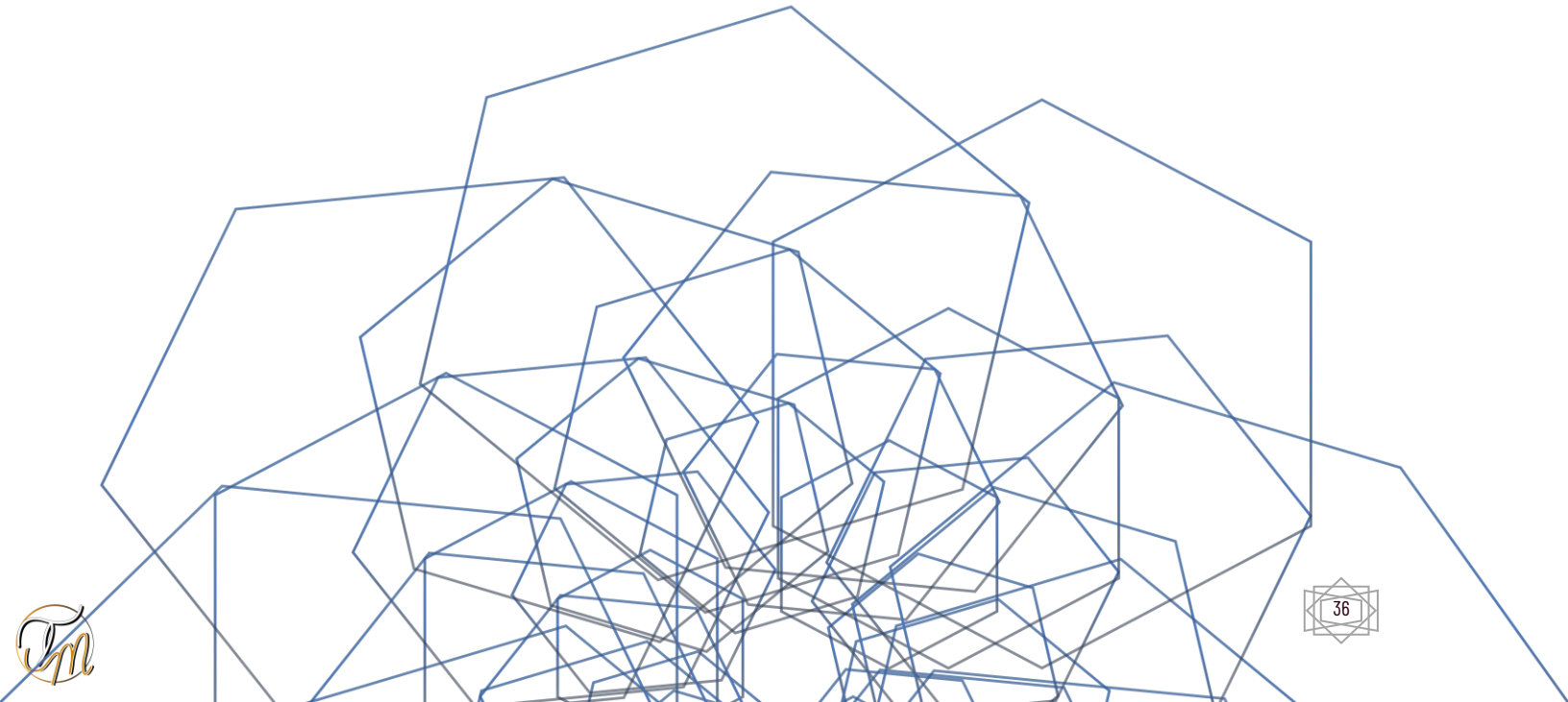
Finalmente, expresamos al caso medio como:

$$Caso\ medio = \sum_{i=1}^{k} O_i P_i$$

$$Caso\ medio = \sum_{i=1}^{log_{\frac{3}{2}}(F(x+2))+1} \left[3x + 7(log_{\frac{3}{2}}(\frac{F(x+2)}{fibo_i})) + 7\right] P_i$$

$$+ \sum_{i=1}^{log_{3}(F(x+2))+1} \left[3x + 6(log_{3}(\frac{F(x+2)}{fibo_i})) + 7\right] P_i$$

$$+\left[3x + 7(log_{\frac{3}{2}}(F(x+2))) + 9\right] + [3x + 6(log_3(F(x+2))) + 9]$$



IMPLEMENTACIÓN

En esta sección mostraremos la implementación de los pseudocódigos a programas (sin haber implementado la parte de los hilos)

BÚSQUEDA LINEAL O SECUENCIAL

```
// // Busqueda Lienal
//Recibimos un arreglo que contiene los numeros del archivo 10 millones
// inicio : Inicio del arreglo
//Un entero "final" para especificar el tamaño del arreglo
//Un entero llamado elemento el cual tiene el número que deseamos encontrar,
//dentro del arreglo
// *aviso: Bandera que nos ayudara a determianr si el elemento ya fue
encontrado o no
void BusquedaLineal(int *A, int inicio, int final, int elem, int *aviso)
{
    //- Variables para el ciclo
    int n;
    for (n = inicio; n < final; n++)
    {
        // Cuando aviso haya cambiado de valor terminara este algoritmo de
busqueda
        if (*aviso >= 0)
        {
            break;
        }
        //Nos ayuda a comparar las posiciones que se encuentran en el arreglo
        //con la posición del elemento que se desea encontrar
        if (A[n] == elem)
        {
            // printf("El elemento fue encontrado en la posicion numero: %d
\n", n);
            *aviso = elem;
        }
    }
}
```

Búsqueda en un árbol binario

```
//Busqueda en Arbol
void BusquedaEnArbol(int * arreglo, int n, int valorABuscar, int * aviso)
{
    // Variable para "almacenar" el árbol
    arbol t;
    // Inicializando arbol
    Iniciar(&t);
    // Rellenamos el arbol
    int i;
    // Variable para poder controlar el ciclo
    for(i=0;i < n; i++)
    {
        NuevoNodo(&t,arreglo[i]);
    }
    // Lanzamos la búsqueda sin hilos
    BuscaValor(&t, valorABuscar, aviso);
}
```

Búsqueda binaria o dicotómica

```
void BusquedaBinaria(int *A, int inicio, int final, int elem, int *aviso)
{
    // -Variables para algoritmo de busqueda
    int mitad;
    while (inicio <= final)
    {
        mitad = (inicio + final) / 2;
        if (A[mitad] < elem)
        {
            inicio = mitad + 1;
        }

        else
        {
            final = mitad - 1;
        }

        if (A[mitad] == elem)
        {
            *aviso = elem;
            // printf("Enontraste el emento en la posicion: %d\n", mitad);
            break;
        }
    }
}
```

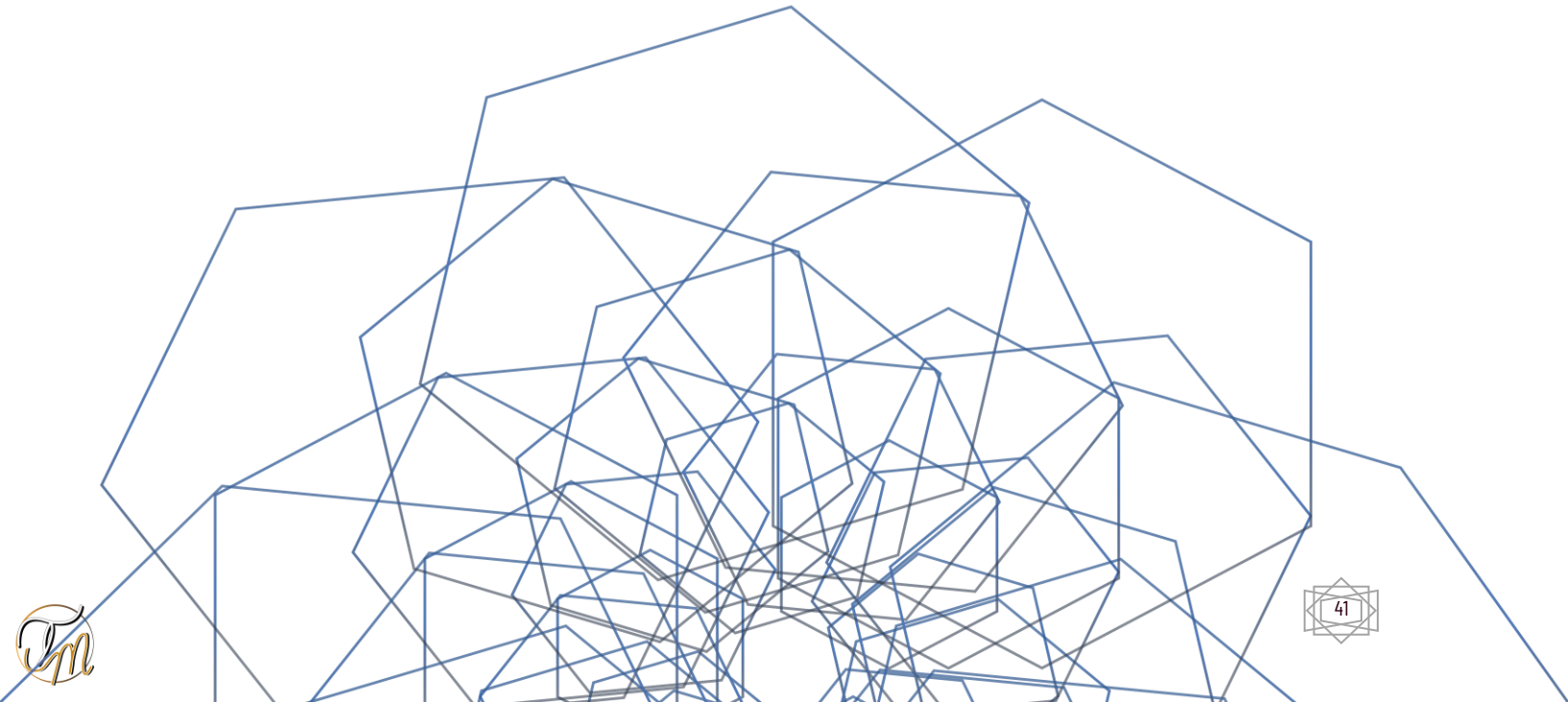
Búsqueda exponencial

```
int BusquedaExponencial(int *A, int n, int elem, int *aviso)
{
    // -Variables para algoritmo de busqueda
    int inicio, final, mitad, i;
    i = 1;
    final = n;
    while (i < final && A[i] <= elem)
    {
        i = i * 2;
    }

    inicio = (i / 2);
    final = (n - 1);

    while (inicio <= final)
    {
        mitad = (inicio + final) / 2;
        if (A[mitad] == elem)
        {
            *aviso = mitad;
            return mitad;
        }
        if (A[mitad] > elem)
        {
            final = mitad - 1;
        }
        else
        {
            inicio = mitad + 1;
        }
    }
    *aviso=-1;
    return -1;
}
```

Búsqueda de Fibonacci

```
// //          Busqueda con Serie de Fibbonacci          // //
//Función para encontrar el mínimo de dos números
int min(int x, int y) { return (x <= y) ? x : y; }

//Regresa "i" si el elemento se encuentra en el arreglo
//Si el elemento no se encuentra, regresa un -1

//Declaramos un arreglo
//Un entero "n" para especificar el tamaño del arreglo
//Un entero llamado n el cual tiene el número que deseamos encontrar,
//dentro del arreglo
int fibMonaccianSearch(int *A, int x, int n)
{
    //Inicializamos los primeros dos números de la serie
    //Estos números siempre son continuos
    int fibMMm2 = 0;
    int fibMMm1 = 1;

    //La suma de los números anteriores dan el número que sigue en la serie
    int fibM = fibMMm2 + fibMMm1;

    //Mientras fibM es menor al tamaño del arreglo,
    //Los valores se irán "recorriendo" de variable
    //Y la suma de los números anteriores,
    //serán el valor de fibM
    while (fibM < n)
    {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    //Nos marca el rango de la izquierda donde no se encuentra el elemento deseado
    int offset = -1;

    while (fibM > 1)
    {
        int i = min(offset + fibMMm2, n - 1);

        //Si el elemento es mayor al valor del índice,
        //Lo valores se irán recorriendo y el "offset"
        //se cambiará al valor del índice,
        if (A[i] < x)
        {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
```

```
        }

        //Si el elemento es mayor a la posición del arreglo en la posiciónde fibMm2
        //el arreglo se dividirá después de la posición i+1
        else if (A[i] > x)
        {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        //Si el elemento es encontrado, devolvemos "i"
        else
            return i;
    }

    //Comparamos el último valor del arreglo con el elemento
    if (fibMMm1 && A[offset + 1] == x)
        return offset + 1;

    //Si el elemento no es encontrado, devolvemos un -1
    return -1;
}
```



ACTIVIDADES Y PRUEBAS

Especificaciones del equipo usado

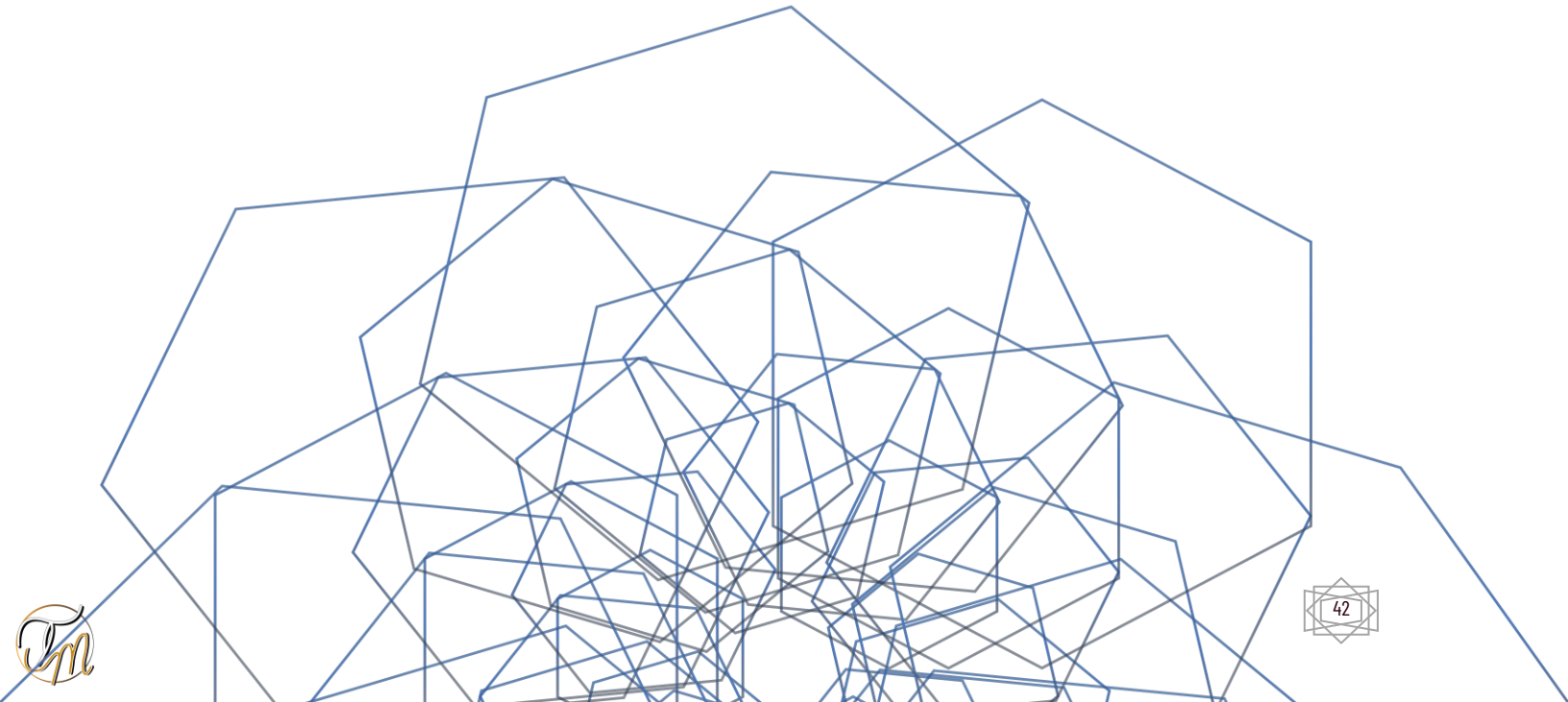
Para la realización de los pruebas de todos lo métodos de de búsqueda, utilizamos el mismo equipo de cómputo, pues de esta forma las comparativas proporcionadas en este documento tendrán una relación justo, ya que habrán sido sometidas a pruebas con las mismas especificaciones computacionales, claro que los resultados pueden variar en comparación de otros precisamente por esa cuestión, el rendimiento de cada equipo computacional.

A continuación, se listan las especificaciones del ordenador usado:

- Marca HP Notebook modelo g3 240
- 4GB ram DDR3
- SSD 120 GB
- Intel Celeron (2 núcleos)

Sistema operativo

El sistema operativo con el cual fueron robados todos los algoritmos en cuestión fue una distribución de Linux llamada: **Elementary OS.**



TIEMPOS DE BÚSQUEDA PROMEDIO (SIN HILOS)

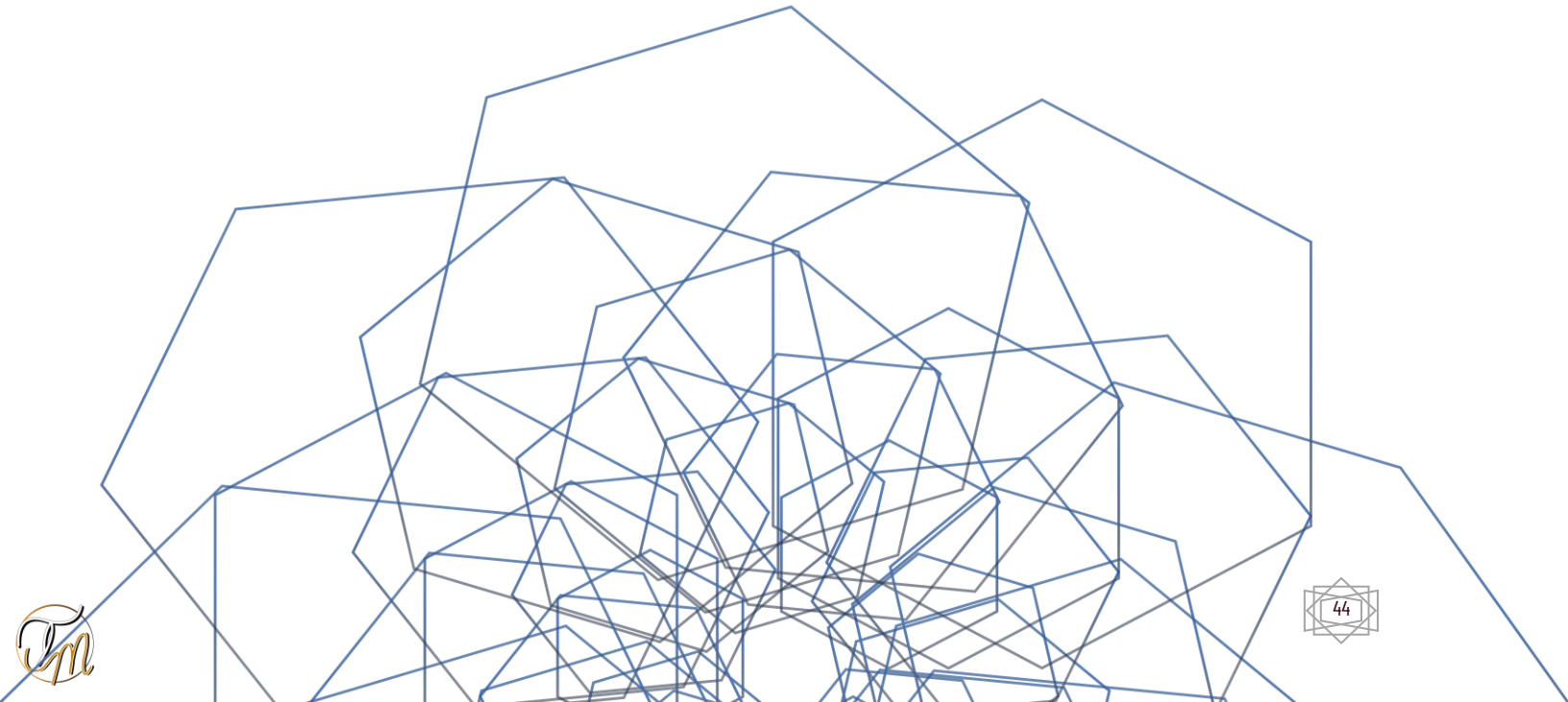
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	1,000,000	0.014422976 s
Búsqueda de árbol	1,000,000	121.8601917 s
Búsqueda binaria	1,000,000	0.00000354085 s
Búsqueda exponencial	1,000,000	0.00000396967 s
Búsqueda de Fibonacci	1,000,000	0.00000355882 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	2,000,000	0.01085474 s
Búsqueda de árbol	2,000,000	2,863.3152 s
Búsqueda binaria	2,000,000	0.00000339746 s
Búsqueda exponencial	2,000,000	0.00000413656 s
Búsqueda de Fibonacci	2,000,000	0.00000380277 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	3,000,000	0.01903196 s
Búsqueda de árbol	3,000,000	3521.8776 s
Búsqueda binaria	3,000,000	0.000004971025 s
Búsqueda exponencial	3,000,000	0.000004315375 s
Búsqueda de Fibonacci	3,000,000	0.00000425577 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	4,000,000	0.01903196 s
Búsqueda de árbol	4,000,000	4331.9095 s
Búsqueda binaria	4,000,000	0.00000376701 s
Búsqueda exponencial	4,000,000	0.0000059128 s
Búsqueda de Fibonacci	4,000,000	0.00000391 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	5,000,000	0.02372143 s
Búsqueda de árbol	5,000,000	5328.2487 s
Búsqueda binaria	5,000,000	0.00000344515 s
Búsqueda exponencial	5,000,000	0.00000436306 s
Búsqueda de Fibonacci	5,000,000	0.0000039458 s



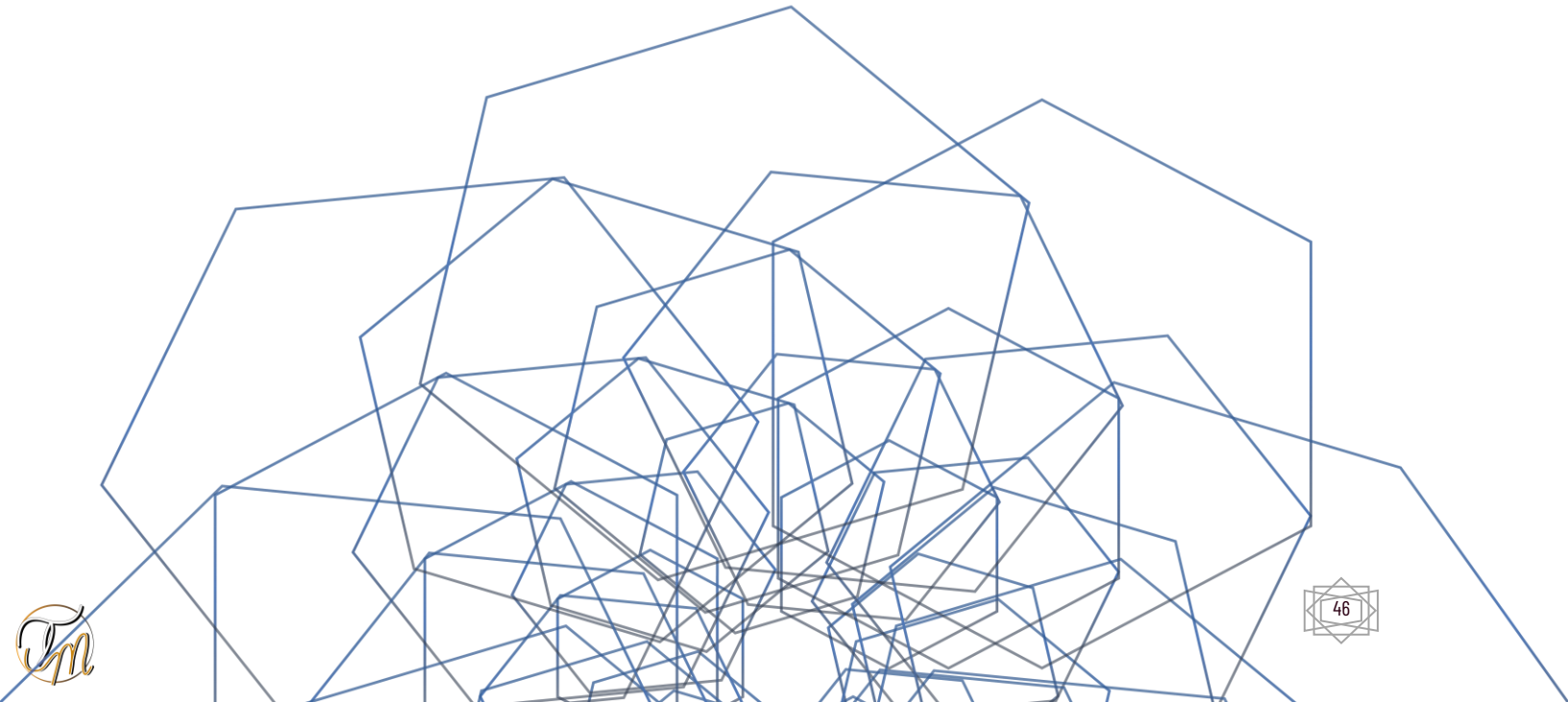
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	6,000,000	0.02843812 s
Búsqueda de árbol	6,000,000	6553.7459 s
Búsqueda binaria	6,000,000	0.0000034332 s
Búsqueda exponencial	6,000,000	0.0000056674 s
Búsqueda de Fibonacci	6,000,000	0.000006032 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	7,000,000	0.03490495 s
Búsqueda de árbol	7,000,000	8061.1075 s
Búsqueda binaria	7,000,000	0.0000039696 s
Búsqueda exponencial	7,000,000	0.0000049471 s
Búsqueda de Fibonacci	7,000,000	0.0000002026 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	8,000,000	0.03926047 s
Búsqueda de árbol	8,000,000	9915.1622 s
Búsqueda binaria	8,000,000	0.00000371933 s
Búsqueda exponencial	8,000,000	0.00000495910 s
Búsqueda de Fibonacci	8,000,000	0.00000426769 s

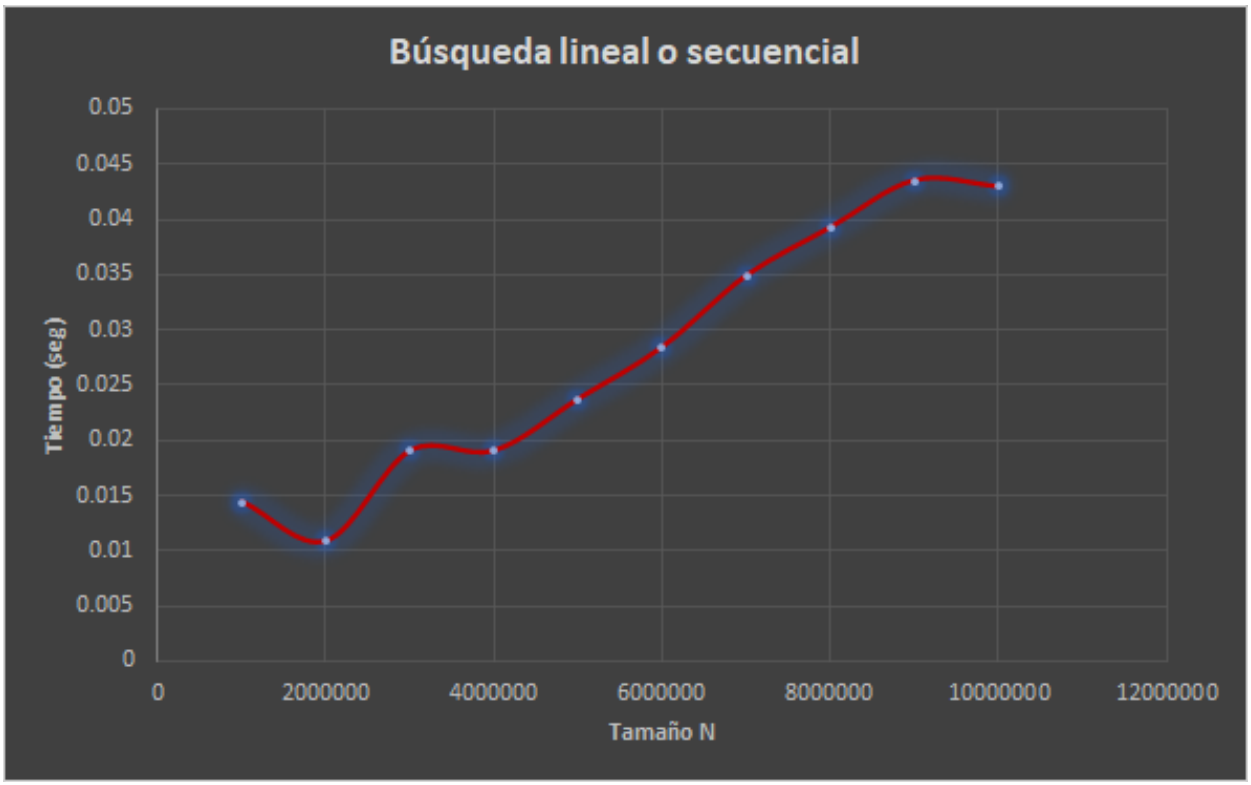
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	9,000,000	0.0435162 s
Búsqueda de árbol	9,000,000	12,195.6496 s
Búsqueda binaria	9,000,000	0.0000041484 s
Búsqueda exponencial	9,000,000	0.0000048637 s
Búsqueda de Fibonacci	9,000,000	0.0000041008 s

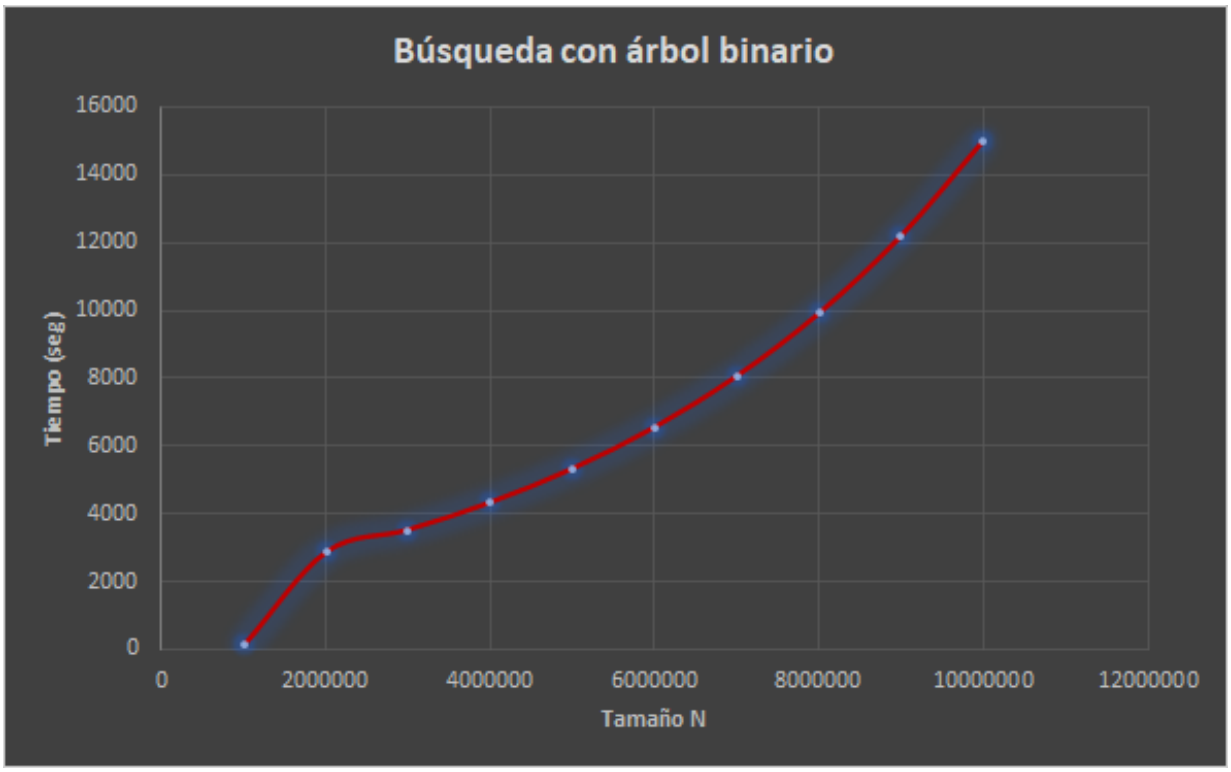
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	10,000,000	0.0429584 s
Búsqueda de árbol	10,000,000	15,000.649 s
Búsqueda binaria	10,000,000	0.00000429198 s
Búsqueda exponencial	10,000,000	0.00000486639 s
Búsqueda de Fibonacci	10,000,000	0.15923 s

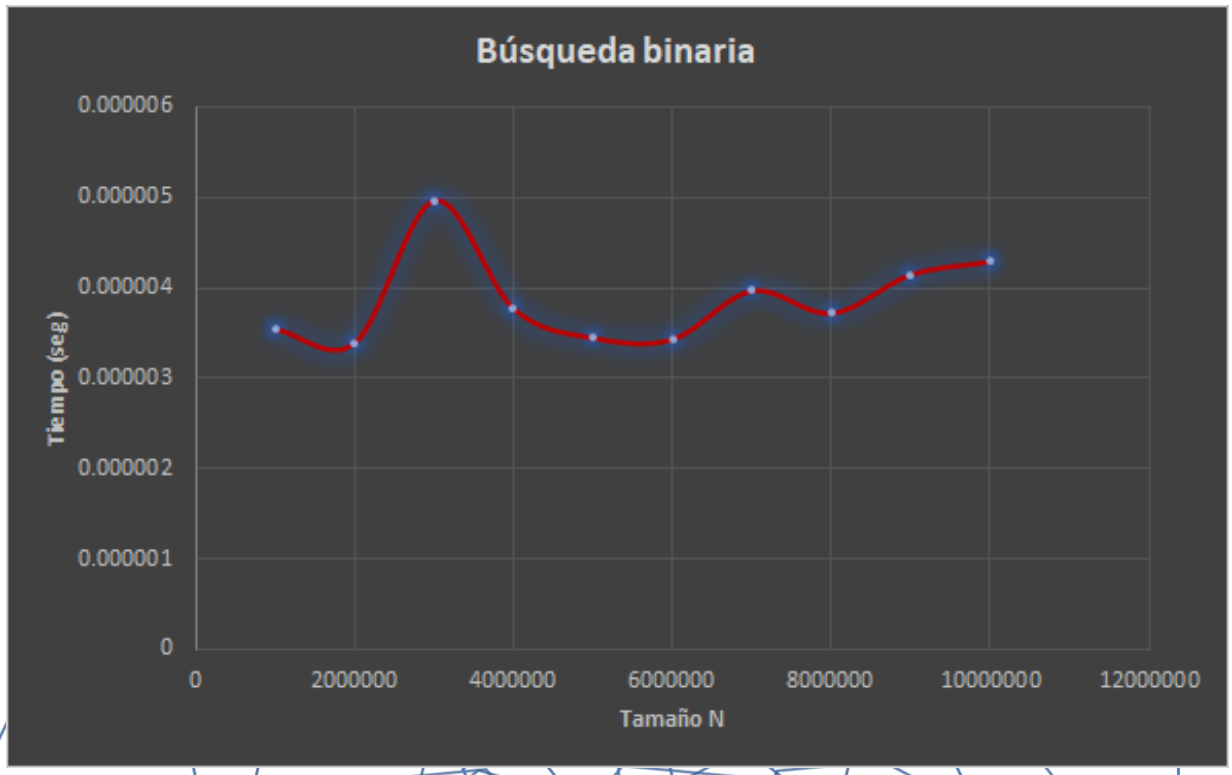


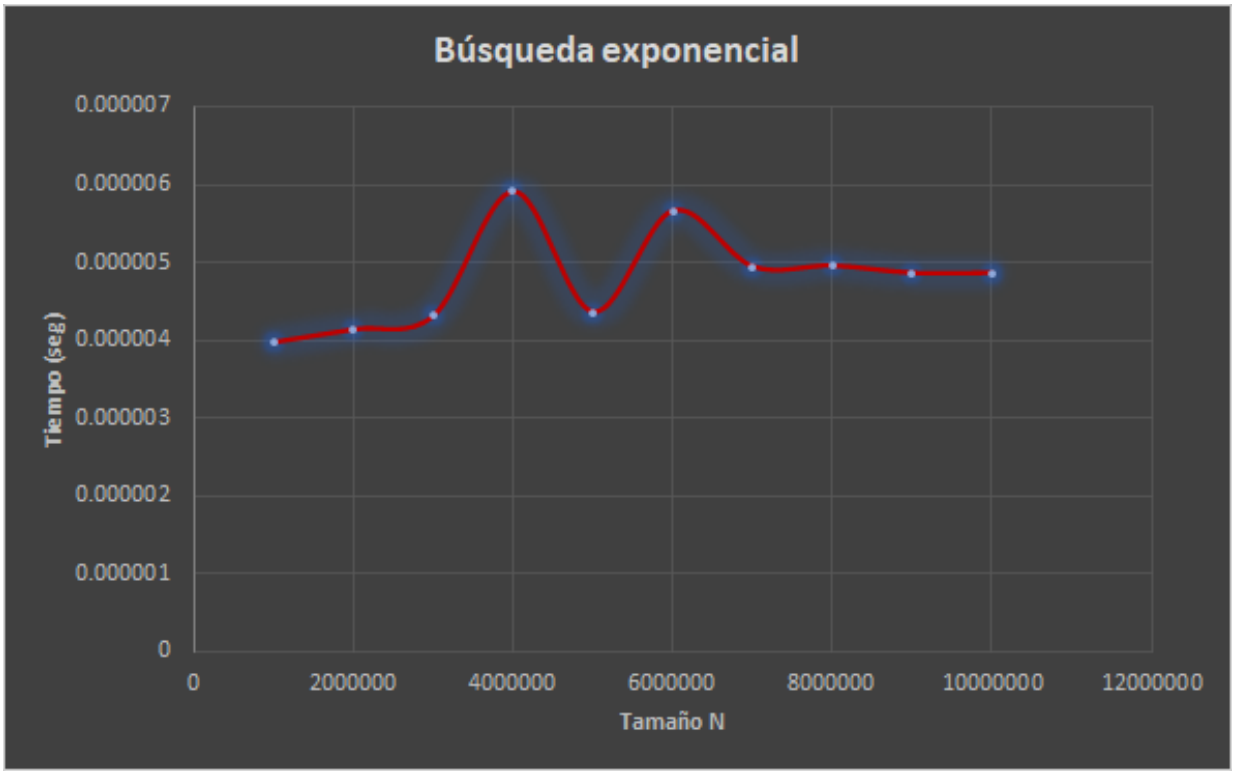
GRÁFICAS DEL COMPORTAMIENTO TEMPORAL

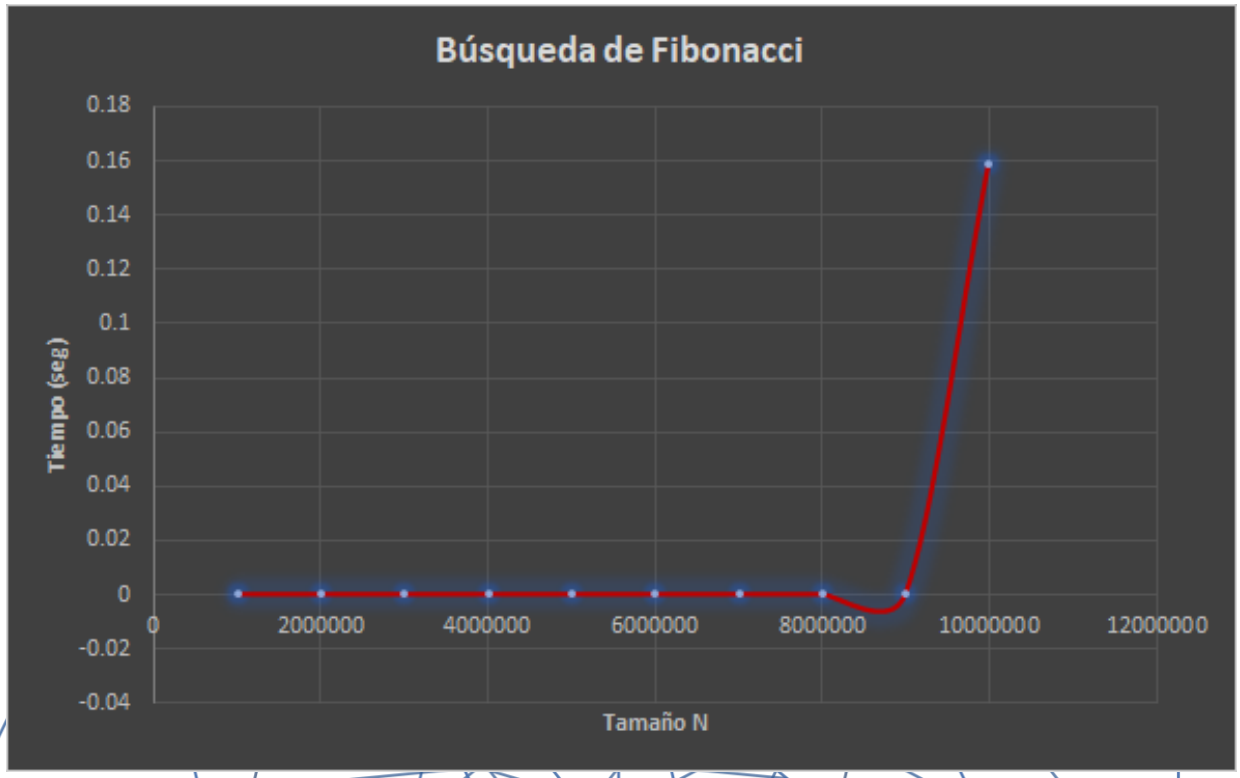
Para realizar las gráficas, tomamos los datos del tiempo promedio (seg) como los valores para nuestro eje y y los tamaños del arreglo (N), los cuales son: 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000, los tomamos como nuestros valores para el eje x.




Búsqueda con árbol binario
Tiempo (seg)
16000
14000
12000
10000
8000
6000
4000
2000
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N


Búsqueda binaria
Tiempo (seg)
0.000006
0.000005
0.000004
0.000003
0.000002
0.000001
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N


Búsqueda exponencial
0.000007
0.000006
0.000005
0.000004
0.000003
0.000002
0.000001
0
Tiempo (seg)
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N


Búsqueda de Fibonacci
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
-0.02
-0.04
Tiempo (seg)
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N

APROXIMACIONES POLINOMIALES

Busqueda Lineal

Logaritmica	Polinomial
f(x)=14670629.4186ln(x)-197227238.9695	f(x)= x^7(0.014) + x^6(0.0108) - x^5(0.019) + x^4(0.019031) - x^3(0.0237) + x^2(0.0284) - x0.0349 + 0.03926047

Búsqueda de Arbol

Logaritmica	Polinomial
f(x)=5638.903150*ln(x)-79632.205435	f(x)= x^6(121.86) - x^5(2863.31) + x^4(4331.90) - x^3(6553.74) + x^2(8061.10) + x9915.16 + 12195.64

Búsqueda Binaria

Logaritmica	Polinomial
f(x)=157.341450*ln(x)+1456.993018	f(x)= x^8(3540.85) - x^7(3397.46) + x^6(4971.025) - x^5(3767.01) + x^4(3445.15) - x^3(3433.2) + x^2(3969.6) - x3719.33 + 4148.4

Búsqueda Exponencial

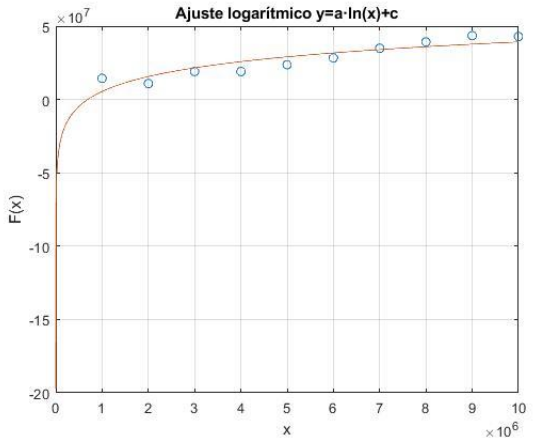
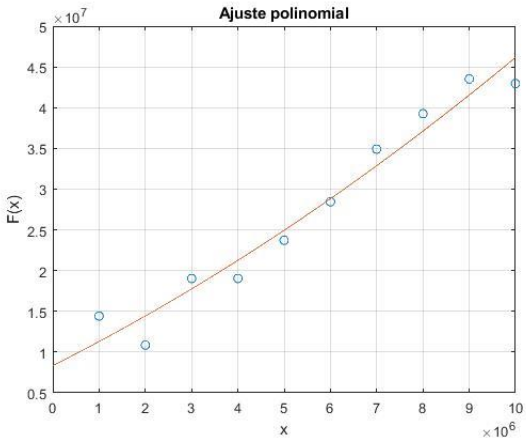
Logaritmica	Polinomial
f(x)=477.575170*ln(x)+-2519.184047	f(x)= x^9(3969.66) + x^8(4136.55) - x^7(4315.37) + x^6(5912.80) - x^5(4363.06) + x^4(5667.40) - x^3(4947.09) + x^2(4959.09) + x4863.70-4866.38

Búsqueda de Fibonacci

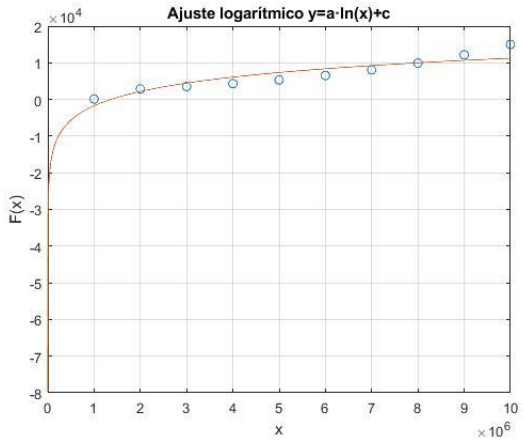
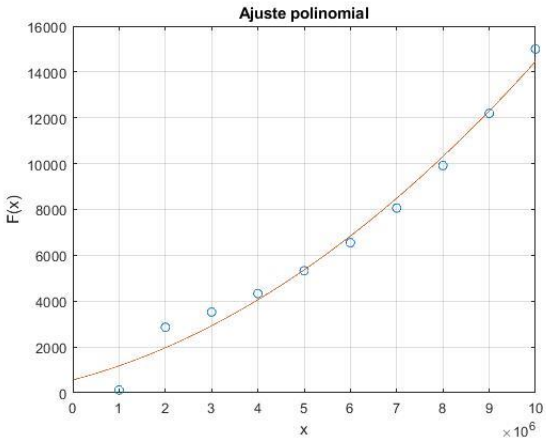
Logaritmica	Polinomial
f(x)=26081876.700*ln(x)+-383803177.9335	f(x)= x^7(3558.82) + x^6(3802.77) - x^5(6032 202.6) + x^4(3945.8) - x^3(3910) + x^2(4267.69) - x4267.69 - 159230

GRÁFICAS APROXIMACIONES

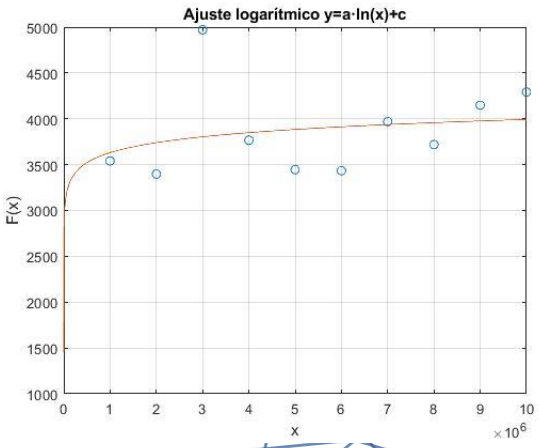
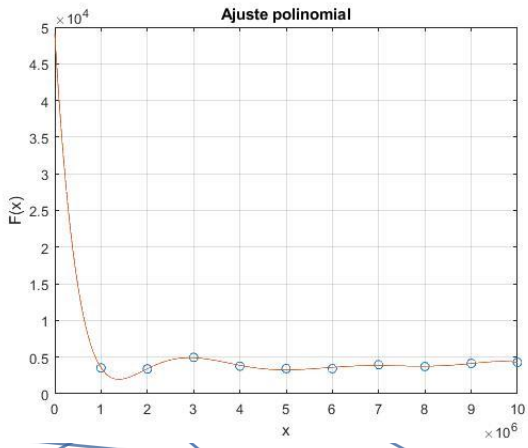
Busqueda Lineal

Logaritmica	Polinomial
	

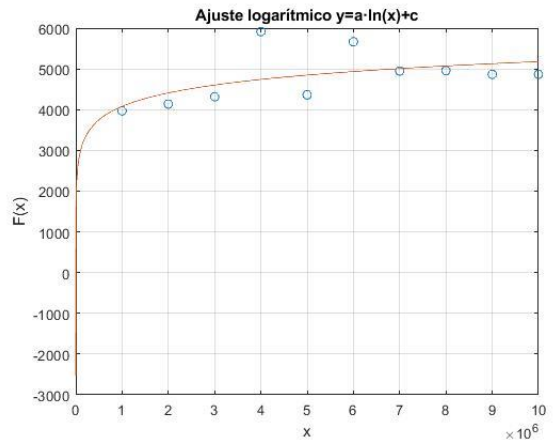
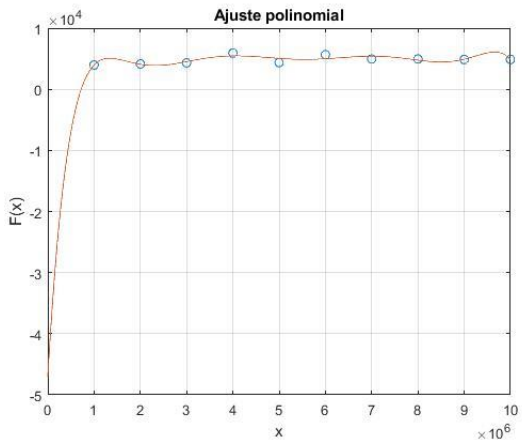
Búsqueda de Arbol

Logaritmica	Polinomial
	

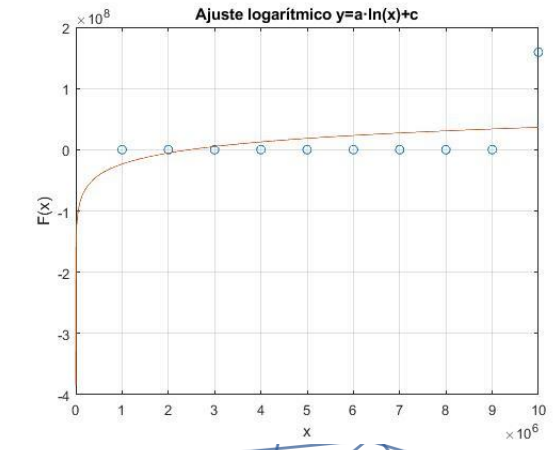
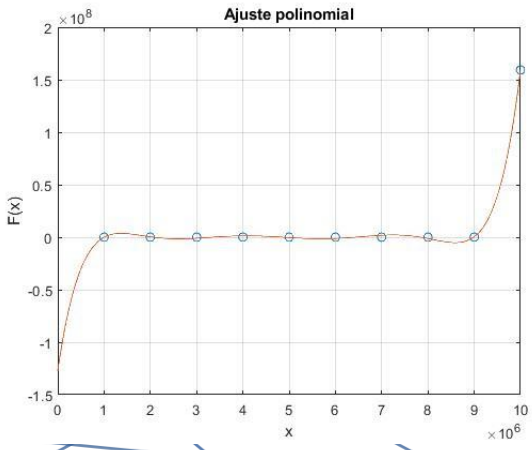
Búsqueda Binaria

Logaritmica	Polinomial
	

Búsqueda Exponencial

Logaritmica	Polinomial
	

Búsqueda de Fibonacci

Logaritmica	Polinomial
	

IDEA DE MEJORA MEDIANTE EL USO DE HILOS

ESTRUCTURAS AUXILIARES CREADAS PARA EL CONTROL DE DATOS DE LOS HILOS

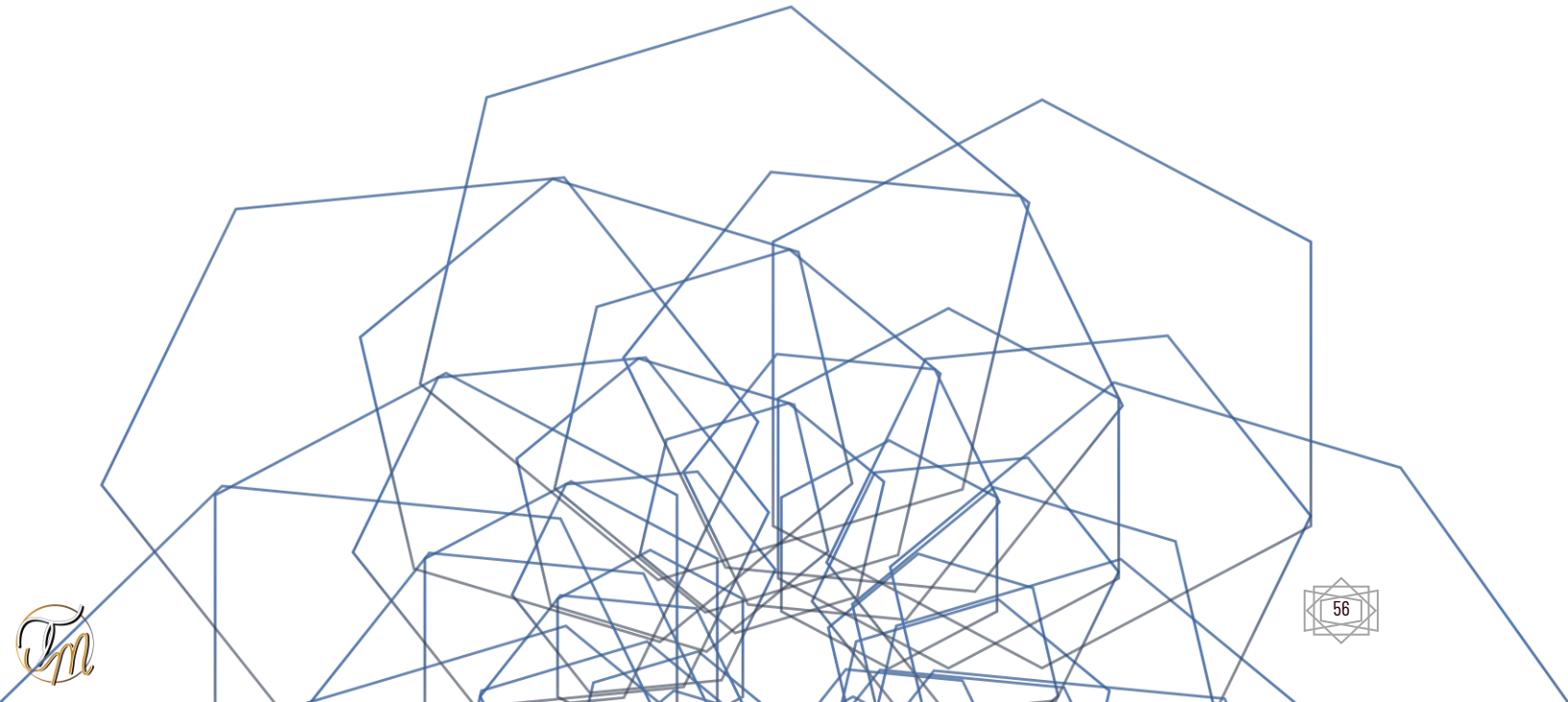
```
//  //Estrucutras auxiliares para la ejecucion de los códigos
//  //Nos ofrecen la alternativa de almacenar los valores dependiento de cada
uno de los hilos
//  //De tal forma que podremos ALMACENAR LOS PARAMETROS MANDADOS A LAS
UNCIONES CON HILOS
typedef struct auxiliarBusquedaLineal
{
    int *arrelgo;
    int valorABuscar;
    int inicio;
    int final;
    int *encontrado;
} AuxiliarLineal;
typedef struct auxiliarBusquedaBinaria
{
    int *arrelgo;
    int valorABuscar;
    int inicio;
    int final;
    int *encontrado;
} AuxiliarBinaria;

typedef struct auxiliarBusquedaEnArbol{
    arbol t;
    int valorABuscar;
    int * encontrado;
}AuxiliarArbol;
```

BÚSQUEDA LINEAL HILOS

```
void BusquedaLinealHilos(int *A, int valorABuscar, int inicio, int final, int *
aviso,int hilos){
    int parts = hilos;
    pthread_t hilo[parts];
    int medio = final/parts;
    for (int i = 0; i < parts; i++)
    {
        AuxiliarBinaria *f = (AuxiliarBinaria *)malloc(sizeof(AuxiliarBinaria))
;
        f->arrelgo=A;
        f->valorABuscar=valorABuscar;
        f->inicio=i*medio;
        f->final=((f->inicio)+medio)-1;
        f->encontrado=aviso;
        pthread_create(&hilo[i],NULL,lanzarBusquedaLineal,(void*)f);
    }

    for (int i = 0; i < parts; ++i)
    {
        pthread_join(hilo[i],NULL);
    }
```

Búsqueda Binaria Hilos

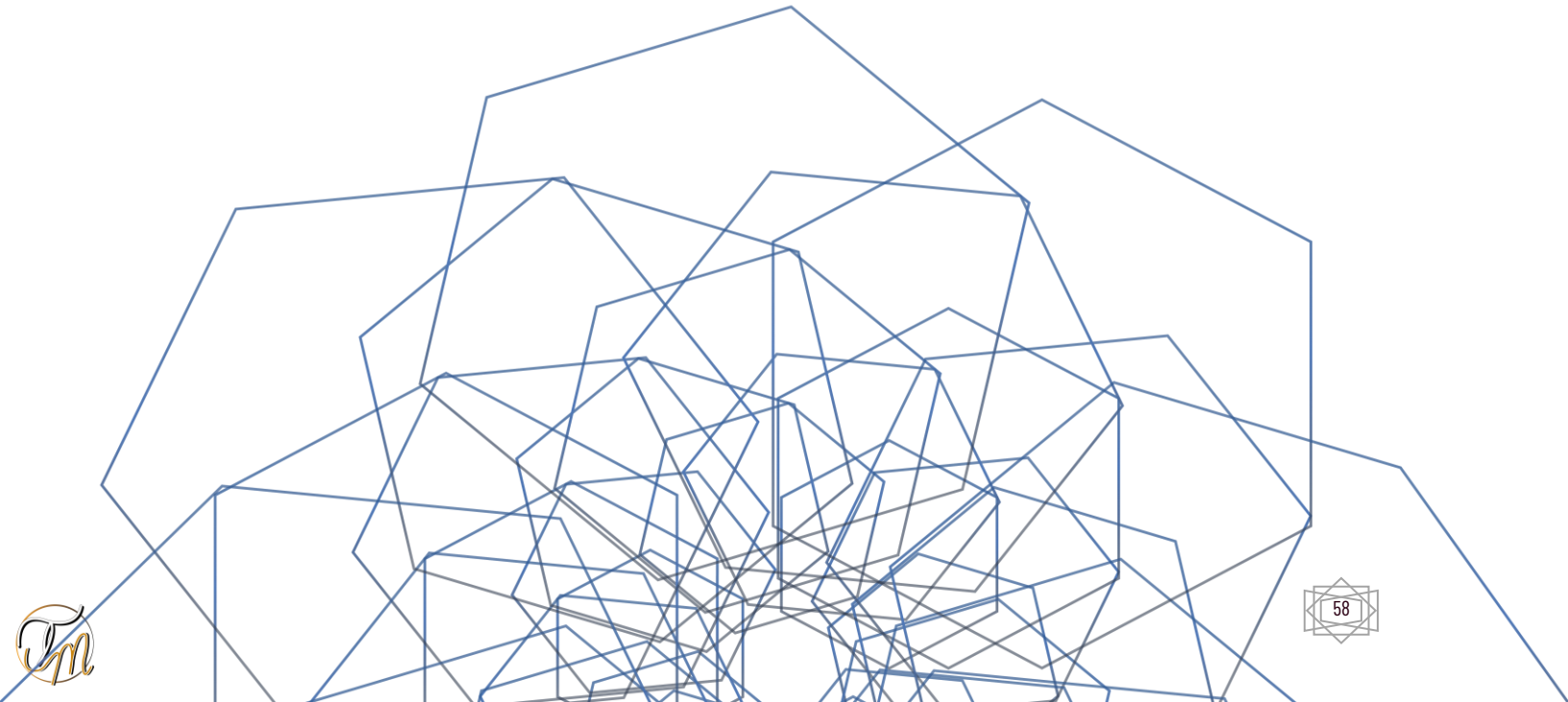
```
void BusquedaBinariaHilos(int *A, int valorABuscar, int inicio, int final, int *avi
so,int hilos){
    int parts = hilos;
    pthread_t hilo[parts];
    int medio = final/parts;
    for (int i = 0; i < parts; i++)
    {
        AuxiliarBinaria *f = (AuxiliarBinaria *)malloc(sizeof(AuxiliarBinaria));
        f->arrelgo=A;
        f->valorABuscar=valorABuscar;
        f->inicio=i*medio;
        f->final=((f->inicio)+medio)-1;
        f->encontrado=aviso;
        pthread_create(&hilo[i],NULL,lanzarBusquedaBinaria,(void*)f);
    }

    for (int i = 0; i < parts; ++i)
    {
        pthread_join(hilo[i],NULL);
    }

}
```



BÚSQUEDA EN UN ÁRBOL BINARIO CON HILOS

```
void BusquedaEnArbolHilos(int * arreglo, int n, int valorABuscar, int * aviso)
{
    // Variable para "almacenar" el árbol
    arbol t;
    // Creamos nuestro árbol
    Iniciar(&t);
    // Rellenamos el arbol
    int i;
    for(i=0;i < n; i++)
    {
        NuevoNodo(&t,arreglo[i]);
    }
    /* Parte de hilos */
    // Comprobamos si nuestro valor a buscar
    // no está en la raíz:
    if((*t).valor == valorABuscar)
    {
        *aviso = (*t).valor;
    }
    else
    {
        // Si no está, comenzamos la
        // repartición de los subarboles
        pthread_t *thread;
        thread = malloc(2*sizeof(pthread_t));
        // Creamos los auxiliares
        AuxiliarArbol * izq = (AuxiliarArbol *)malloc(sizeof(AuxiliarArbol));
        izq->t = (*t).izquierdo;
        izq->valorABuscar = valorABuscar;
        izq->encontrado = aviso;


        AuxiliarArbol * der = (AuxiliarArbol *)malloc(sizeof(AuxiliarArbol));
        der->t = (*t).derecho;
        der->valorABuscar = valorABuscar;
        der->encontrado = aviso;

        // Creamos los hilos
        if(pthread_create(&thread[0], NULL, procesarBusquedaArbol, (void *)izq) !=
0)
        {
            perror("El thread no  pudo crearse [Arbol]\n");
            exit(-1);
        }
        if(pthread_create(&thread[1], NULL, procesarBusquedaArbol, (void *)der) !=
0)
        {
            perror("El thread no  pudo crearse [Arbol]\n");
```

```
            exit(-1);
        }
        // Esperamos a los hilos
        int i;
        for (i=0; i<2; i++) pthread_join (thread[i], NULL);
        free(thread);
    }
}
```



FIBONACCI HILOS

```c
//Librerias incluidas
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
// Algoritmo de Busqueda por Fibonacci con la implementacion de hilos
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "./Tiempo/tiempo.c"

/* Prototipo de la funcion de lectura del archivo de los 10 millones de numeros */
int *LeerArchivo(int *A, int n);
struct argumentos {
    int *arr;
    int fib2;
    int fib1;
    int n;
    int target;
};

int min(int x, int y) { return (x<=y)? x : y; }

/*
    Funcion: fibonacci
    Esta realiza una busqueda Fibonacci en el intervalo dado
    Lo que se recibe: (Struct) args que contienen los numeros Fibonacci donde inicia
la busqueda,
    el numero a buscar y el arreglo donde buscar.
    Imprime si ha encontrado el numero
*/
void *fibonacci(void *ags){
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medici□n
de tiempos
    struct argumentos *args = (struct argumentos*) ags;
    int status = 0, offset = 0;
    // Inizializa numeros fibonacci
    int fibMMm2 = args->fib2; // (m-2)No. Fibonacci
    int fibMMm1 = args->fib1; // (m-1)No. Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m Fibonacci
    int n = args->n;
    int x = args->target;
    uswtime(&utime0, &stime0, &wtime0);
    while (fibM > 1)
        {
            // Revisa si fibMm2 esta en una locacion valida
            int i = min(offset+fibMMm2, n-1);
```

```
            // Si x es maqyor que el valor de index fibMn2, corta el subarray

            if ((args->arr[i]) < x)
            {
                fibM = fibMMm1;
                fibMMm1 = fibMMm2;
                fibMMm2 = fibM - fibMMm1;
                offset = i;
            }

            // Si x es menor que el valor index fibMm2, corta el subarray despues de
i+1
            else if ((args->arr[i]) > x)
            {
                fibM = fibMMm2;
                fibMMm1 = fibMMm1 - fibMMm2;
                fibMMm2 = fibM - fibMMm1;
            }

            // Elemento encontrado retorna S
            else
            {
        uswtime(&utime1, &stime1, &wtime1);
                printf("Encontrado\n");
                printf("\n");
                printf("%d\n",x);
                printf("real (Tiempo total)  %.10e s\n", wtime1 - wtime0);
                printf("////////////////////////////////////////////////");
                pthread_exit((void*)&status);

            }
        }

        // compara el ultimo elemento con x
        if(fibMMm1 == 1 && (args->arr[offset+1]) == x)
            printf("Encontrado\n");

    pthread_exit((void*)&status);
}

/*
    main
    Recibe: Tama?o de problema (argv[1]), Numero a buscar (argv[2]).
    Entonces realiza la busqueda Fibonacci utilizando dos hilos.
    Y finalmente devuelve el tiempo que tarda en ejecutarse el algoritmo.
    Nota: argv[1] DEBE SER MAYOR A 0.
*/
void main(int argc, char *argv[]){
    struct argumentos *margs = NULL;
    pthread_t id1 = 0,id2 = 0;
    int n = 0, target = 0, *numeros, i = 0;
```

```
    int eid1 = -1, eid2 = -1;

    int fibMMm2 = 0; // (m-2)No. Fibonacci
    int fibMMm1 = 1; // (m-1)No. Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m Fibonacci

    n = atoi(argv[1]);
    target = atoi(argv[2]);

    numeros = malloc(sizeof(int)*n);
    margs = malloc(sizeof(struct argumentos));

    margs -> n = n;
    margs -> target = target;

    LeerArchivo(numeros,n);

    margs-> arr = numeros;
    /*Generamos los numeros fibonacci, para el tama□o de problema
    el □ndice m□ximo alcanzado en la serie fibonacci ser□ el 36
    por lo tanto, se propone la creacion de dos hilos, uno busca
    en □ndice < 18 y otro en □ndice > 18*/
    i = 0;
    // fibM va a guardar el mas peque□o
    while (fibM < n)
    {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
        i++;
        if(i==18){
            //Lanzamos un hilo que busque aqui
            margs -> fib2 = fibMMm2;
            margs -> fib1 = fibMMm1;
            pthread_create(&id1,NULL,fibonacci,(void*)margs);
        }
    }
    //Alcanzo los ultimos numeros, lanzamos otro hilo para que busque por alla
    margs -> fib2 = fibMMm2;
    margs -> fib1 = fibMMm1;
    pthread_create(&id2,NULL,fibonacci,(void*)margs);
    pthread_join(id1,(void*)&eid1);
    pthread_join(id2,(void*)&eid2);

    printf("\n");
    printf("n=%i\n",n);
    printf("x=%i\n",target);
    return;
}
```

```c
/* Funcion para leer y almacenar n cantidad de numeros del archivo 10millones dentro de un arreglo (con memoria previamente asignada)
    Recibe:
        *A: Arreglo de elementos vacio con la memoria suficiente para almacenar n cantidad de elementos
         n: cantidad de elementos que seran leidos del archivo y almacenados en el arreglo
 */
int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}
```

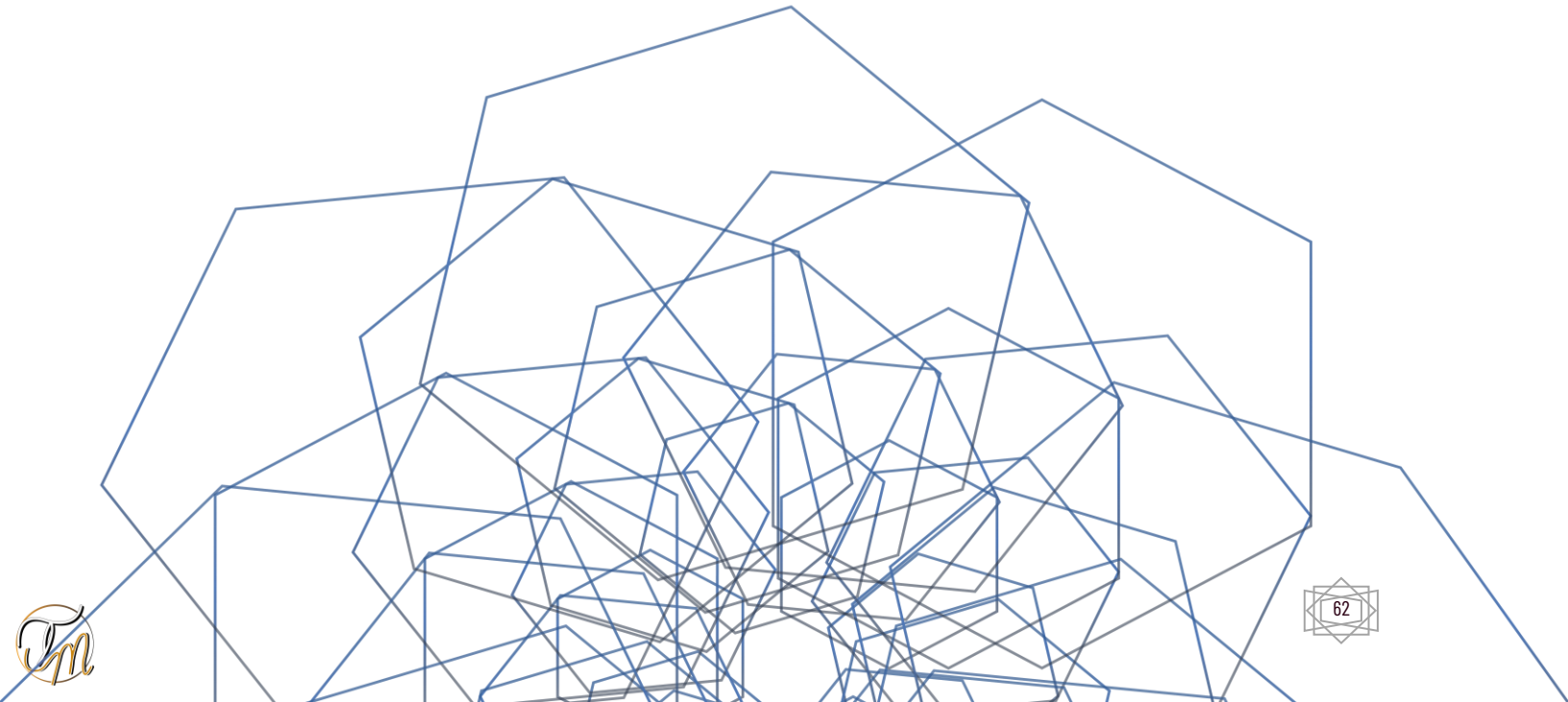


TABLA COMPARATIVA HILOS

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	1,000,000	0.00341104 s
Búsqueda de árbol	1,000,000	120.81449 s
Búsqueda binaria	1,000,000	0.00010546 s
Búsqueda exponencial	1,000,000	0.00309586 s
Búsqueda de Fibonacci	1,000,000	0.0000011563 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	2,000,000	0.0053881 s
Búsqueda de árbol	2,000,000	483.46671 s
Búsqueda binaria	2,000,000	0.0000867 s
Búsqueda exponencial	2,000,000	0.00184273 s
Búsqueda de Fibonacci	2,000,000	0.0000009059 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	3,000,000	0.00764948 s
Búsqueda de árbol	3,000,000	502.80537 s
Búsqueda binaria	3,000,000	0.000097567 s
Búsqueda exponencial	3,000,000	0.0020225 s
Búsqueda de Fibonacci	3,000,000	0.00000092114 s

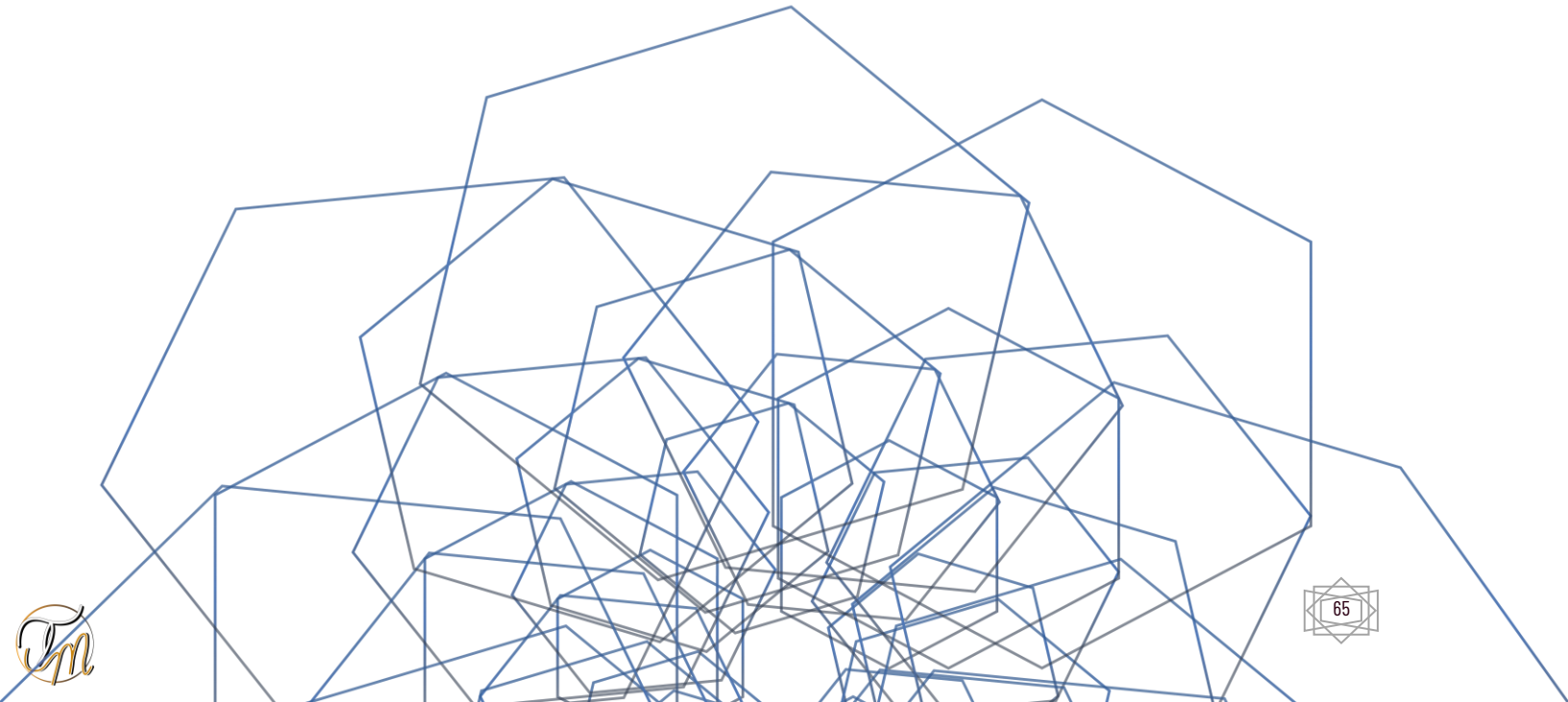
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	4,000,000	0.0097284 s
Búsqueda de árbol	4,000,000	522.91759 s
Búsqueda binaria	4,000,000	0.000098659 s
Búsqueda exponencial	4,000,000	0.00244307 s
Búsqueda de Fibonacci	4,000,000	0.0000009328 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	5,000,000	0.0123962 s
Búsqueda de árbol	5,000,000	543.83429 s
Búsqueda binaria	5,000,000	0.000099764 s
Búsqueda exponencial	5,000,000	0.0024675 s
Búsqueda de Fibonacci	5,000,000	0.0000009446 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	6,000,000	0.014676 s
Búsqueda de árbol	6,000,000	565.58766 s
Búsqueda binaria	6,000,000	0.00010088 s
Búsqueda exponencial	6,000,000	0.00249217 s
Búsqueda de Fibonacci	6,000,000	0.00001502 s

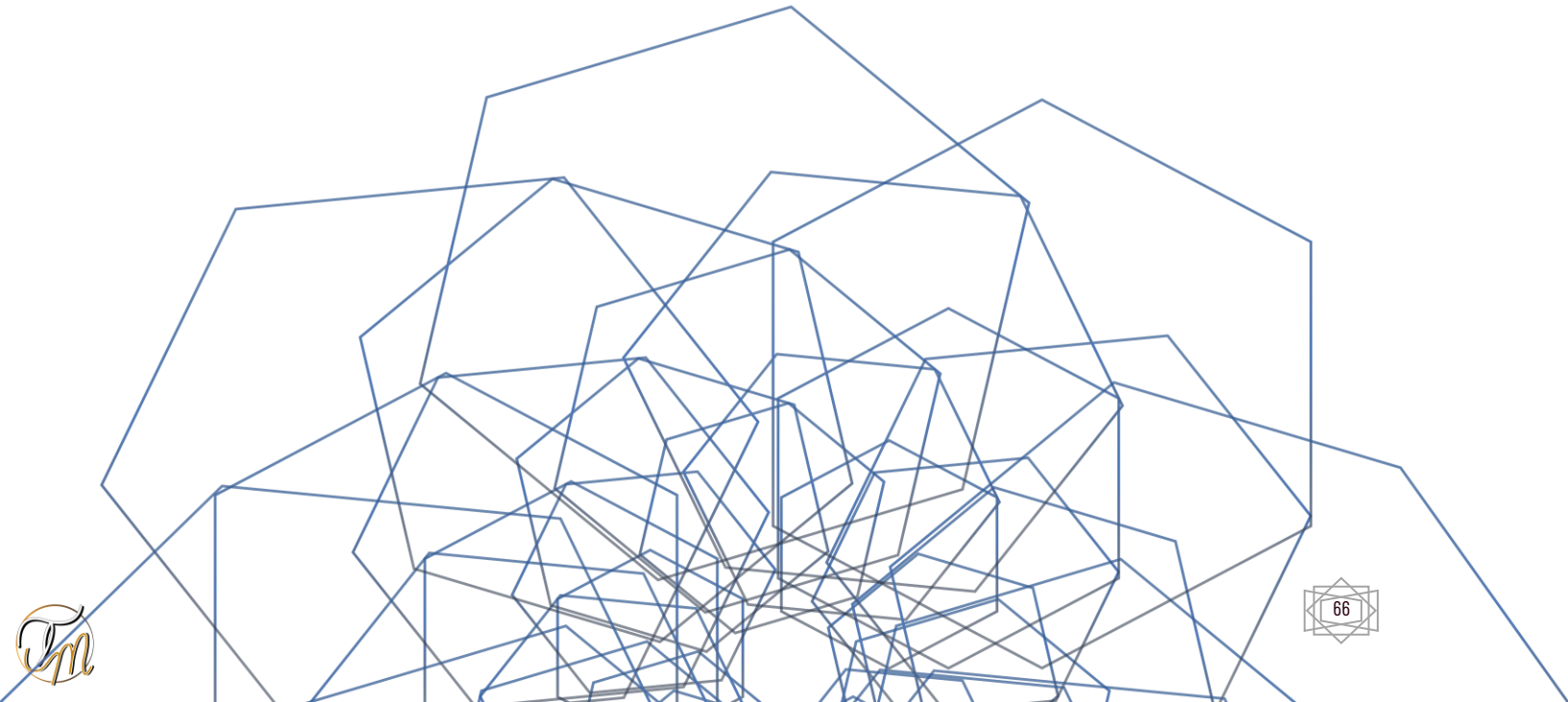
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	7,000,000	0.0168761 s
Búsqueda de árbol	7,000,000	588.21117 s
Búsqueda binaria	7,000,000	0.000102011 s
Búsqueda exponencial	7,000,000	0.00251709 s
Búsqueda de Fibonacci	7,000,000	0.0000015211 s

TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	8,000,000	0.0187193 s
Búsqueda de árbol	8,000,000	611.73962 s
Búsqueda binaria	8,000,000	0.000103154 s
Búsqueda exponencial	8,000,000	0.00198597 s
Búsqueda de Fibonacci	8,000,000	0.0000015515 s



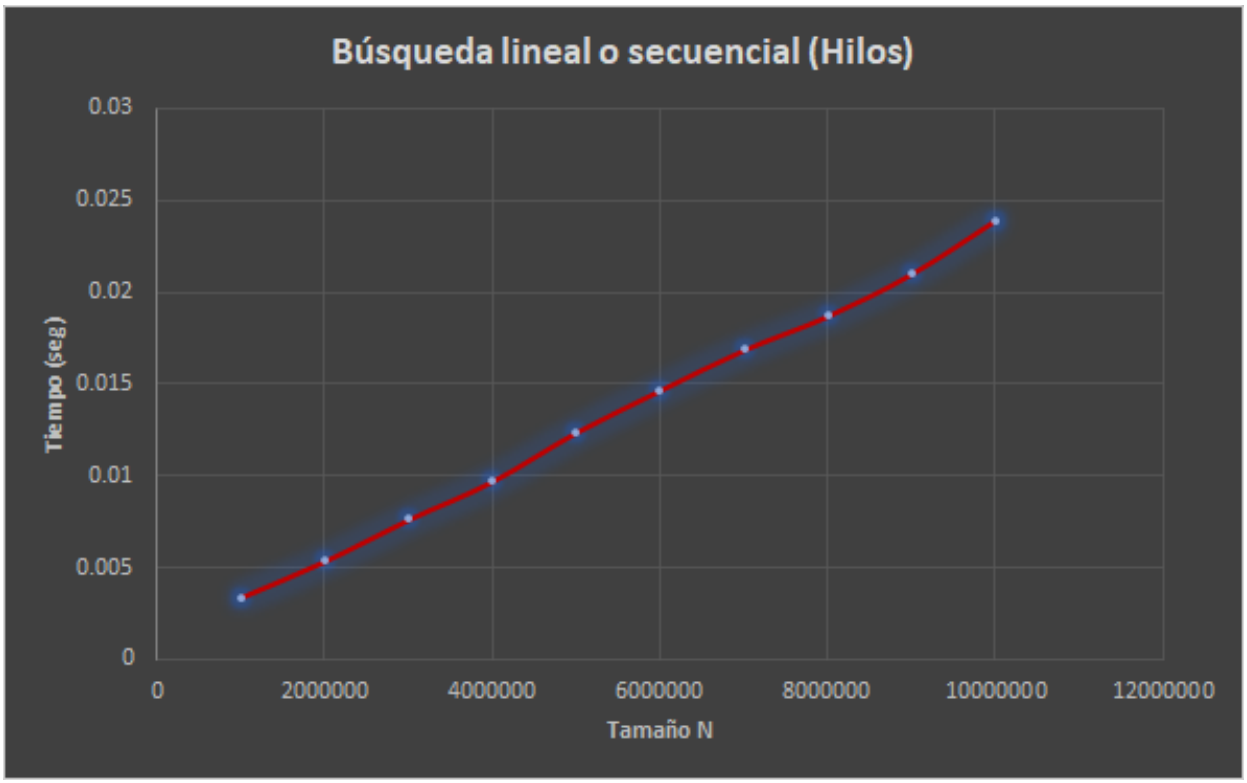
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	9,000,000	0.02101357 s
Búsqueda de árbol	9,000,000	636.2092 s
Búsqueda binaria	9,000,000	0.000104309 s
Búsqueda exponencial	9,000,000	0.0025842 s
Búsqueda de Fibonacci	9,000,000	0.0000015825 s

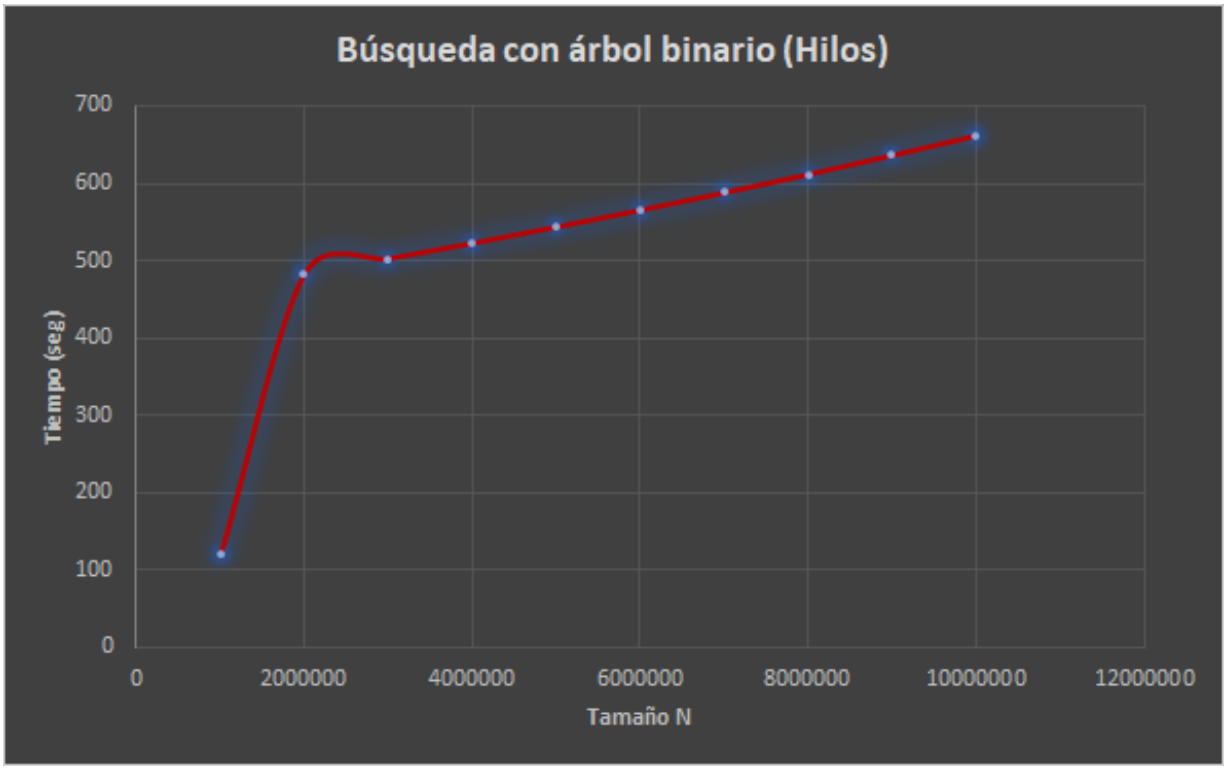
TIEMPOS DE BÚSQUEDA PROMEDIO PARA CADA N		
ALGORITMO	TAMAÑO DE N	TIEMPO PROMEDIO DE BÚSQUEDA
Búsqueda lineal	10,000,000	0.02392023 s
Búsqueda de árbol	10,000,000	661.65757 s
Búsqueda binaria	10,000,000	0.000105478 s
Búsqueda exponencial	10,000,000	0.00261004 s
Búsqueda de Fibonacci	10,000,000	0.0000016142 s

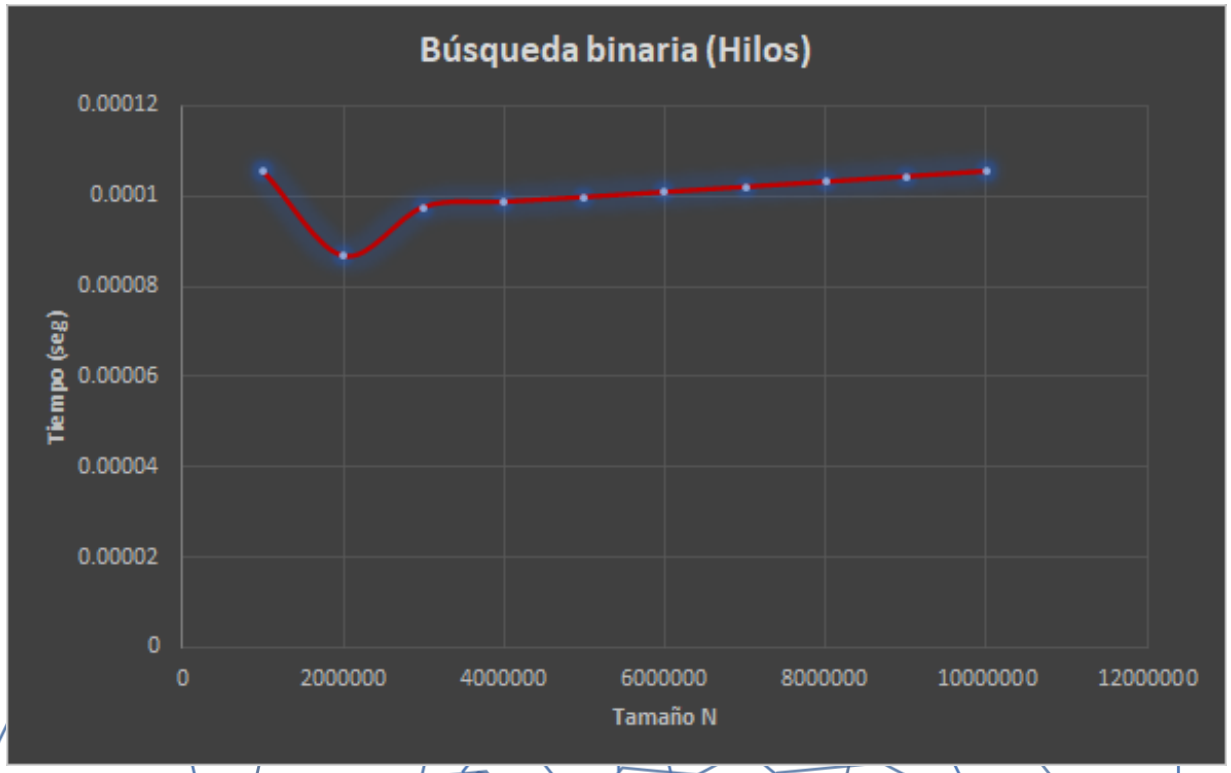


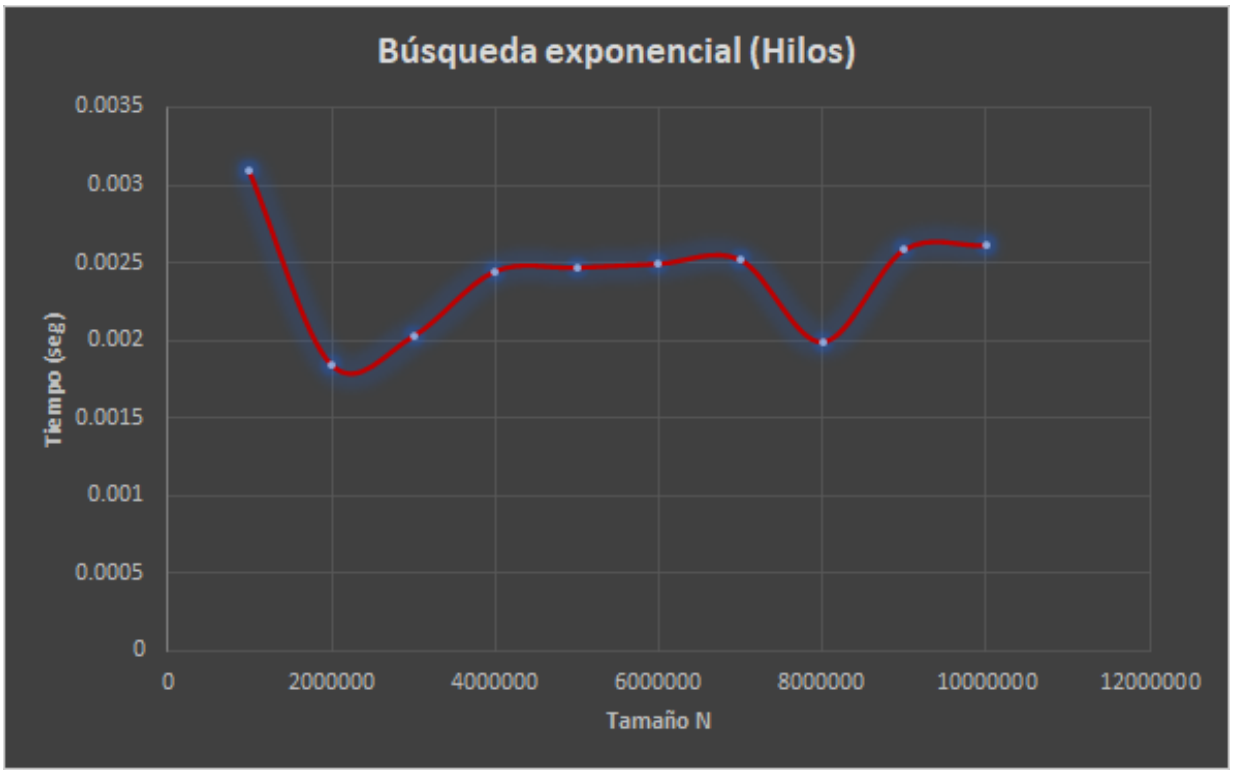
GRÁFICAS DEL COMPORTAMIENTO TEMPORAL

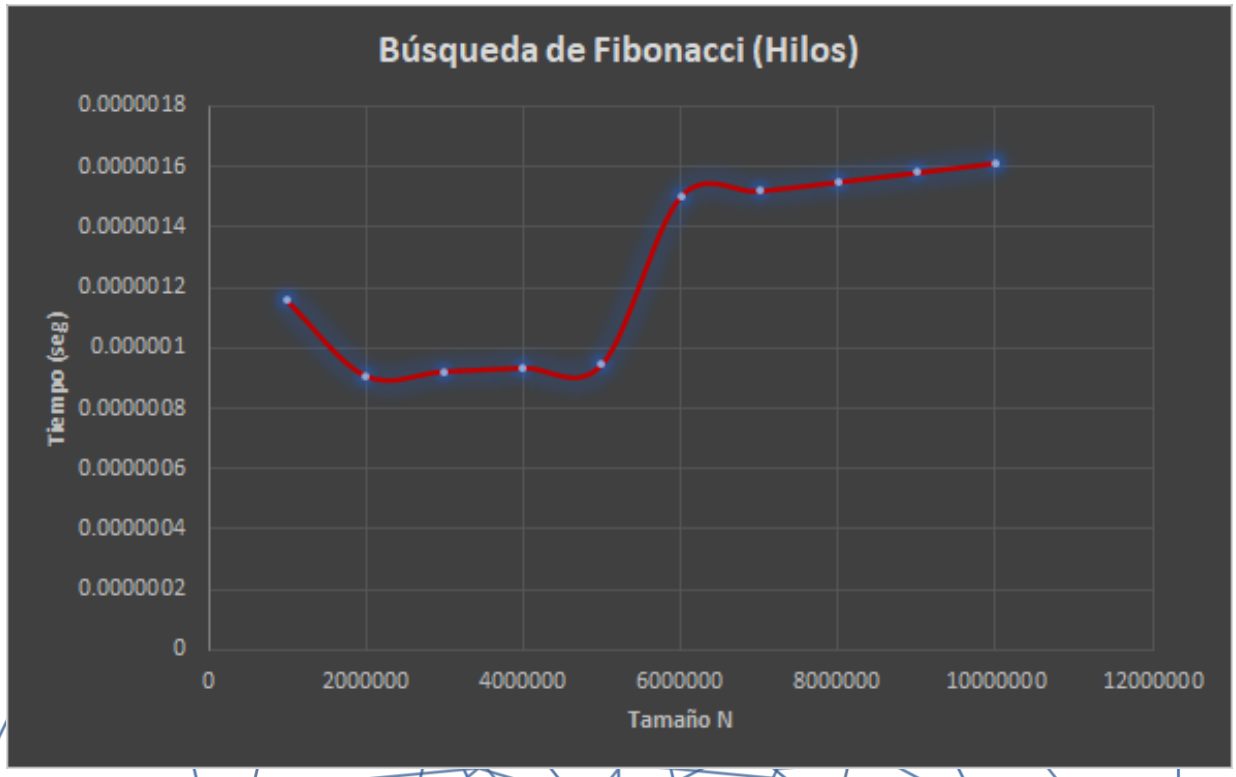
Para realizar las gráficas, tomamos los datos del tiempo promedio (seg) como los valores para nuestro eje y y los tamaños del arreglo (N), los cuales son: 1000000, 2000000, 3000000, 4000000, 5000000, 6000000, 7000000, 8000000, 9000000, 10000000, los tomamos como nuestros valores para el eje x.




Búsqueda con árbol binario (Hilos)
Tiempo (seg)
700
600
500
400
300
200
100
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N


Búsqueda binaria (Hilos)
Tiempo (seg)
0.00012
0.0001
0.00008
0.00006
0.00004
0.00002
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N

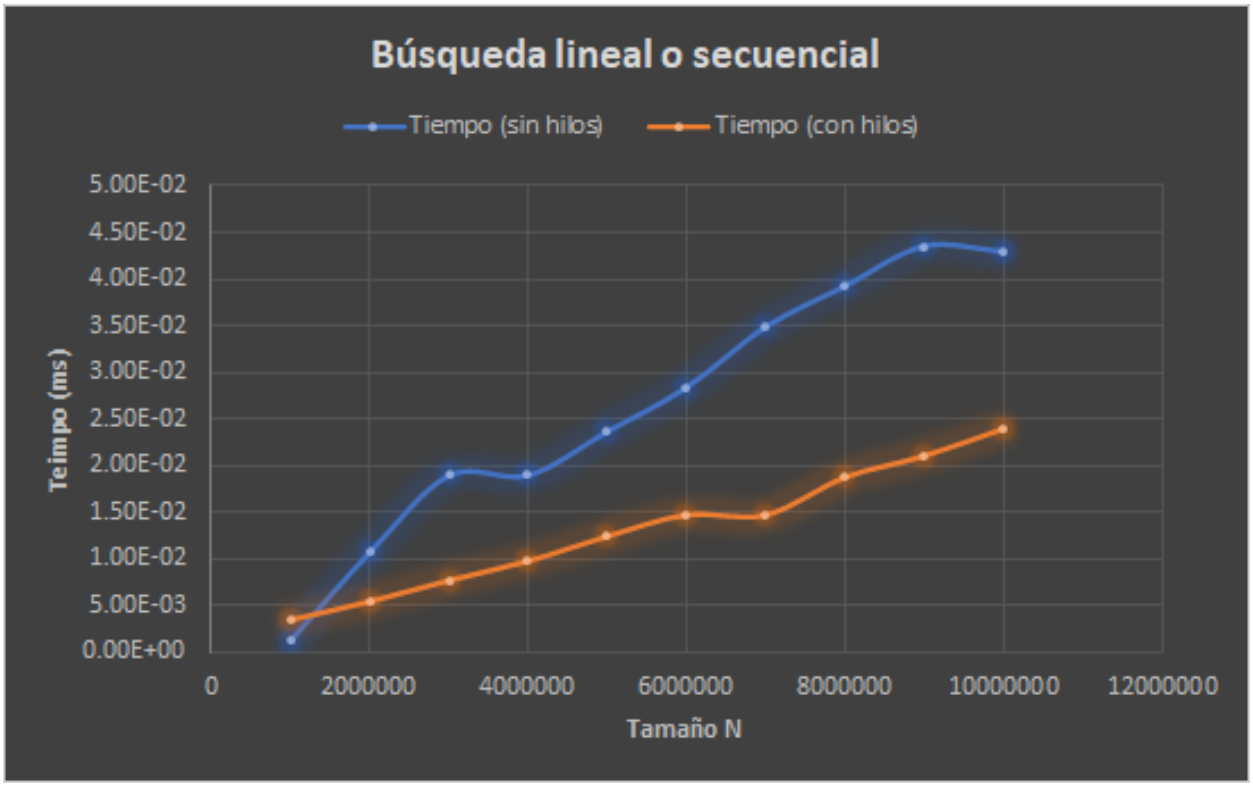

Búsqueda exponencial (Hilos)
Tiempo (seg)
0.0035
0.003
0.0025
0.002
0.0015
0.001
0.0005
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N


Búsqueda de Fibonacci (Hilos)
Tiempo (seg)
0.0000018
0.0000016
0.0000014
0.0000012
0.000001
0.0000008
0.0000006
0.0000004
0.0000002
0
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N

MEJORA UTILIZANDO HILOS EN LOS ALGORITMOS

BÚSQUEDA LINEAL O SECUENCIAL

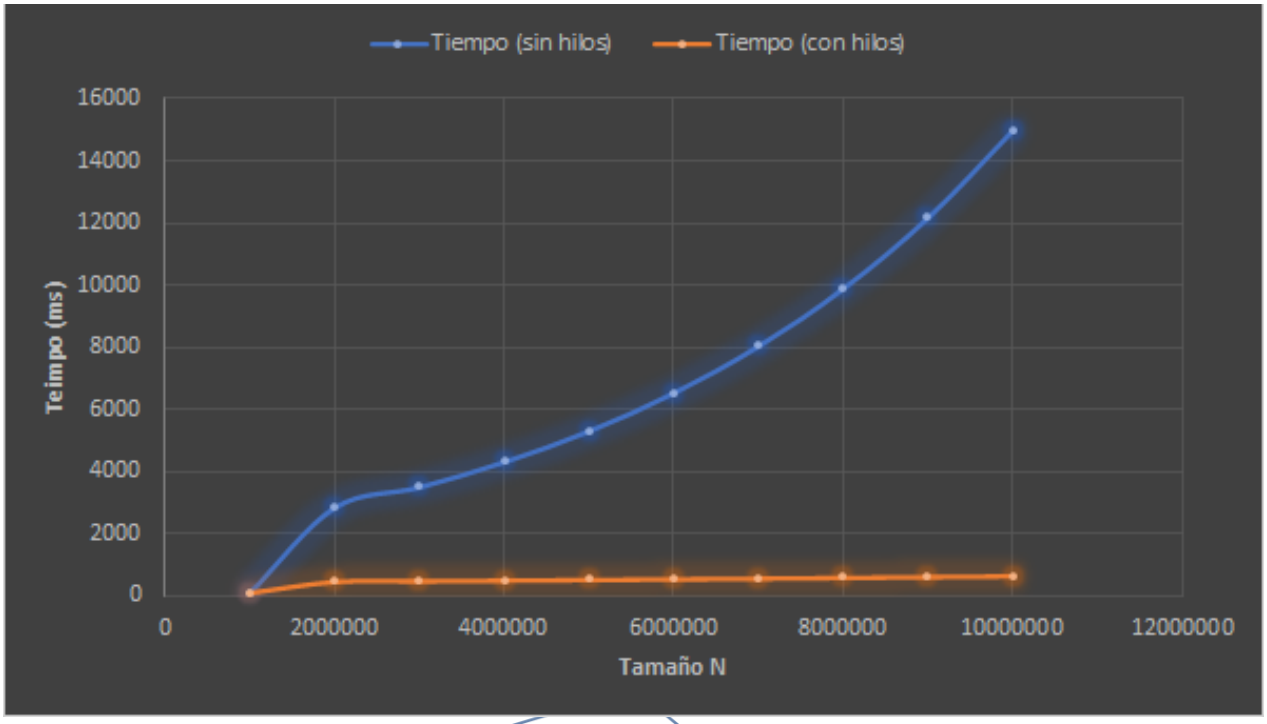
Tamaño de N	Tiempo real (versión sin hilos)	Tiempo real (versión con hilos)	Mejora
1000000	14.4229 ms	3.41104 ms	14.422976 ms/3.41104 ms =4.22 Mejora 422%
2000000	10.85474 ms	5.3881 ms	10.85474 ms/5.3881 ms =2.01 Mejora 201%
3000000	19.03196 ms	7.64948 ms	19.03196 ms/7.64948 ms =2.48 Mejora 248%
4000000	19.03196 ms	9.7284 ms	19.03196 ms/9.7284 ms =1.95 Mejora 195%
5000000	23.72143 ms	12.3962 ms	23.72143 ms/12.3962 ms =1.91 Mejora 191%
6000000	28.43812 ms	14.676 ms	28.43812 ms/14.676 ms =1.93 Mejora 193%
7000000	34.90495 ms	14.676 ms	34.90495 ms/16.8761 ms =2.06 Mejora 206%
8000000	39.26047 ms	18.7193 ms	39.26047 ms/18.7193 ms =2.09 Mejora 209%
9000000	43.5162 ms	21.01357 ms	43.5162 ms/21.01357 ms =2.07% Mejora 207
10000000	42.9584 ms	23.92023 ms	42.9584 ms/23.92023 ms =1.79 Mejora 179%



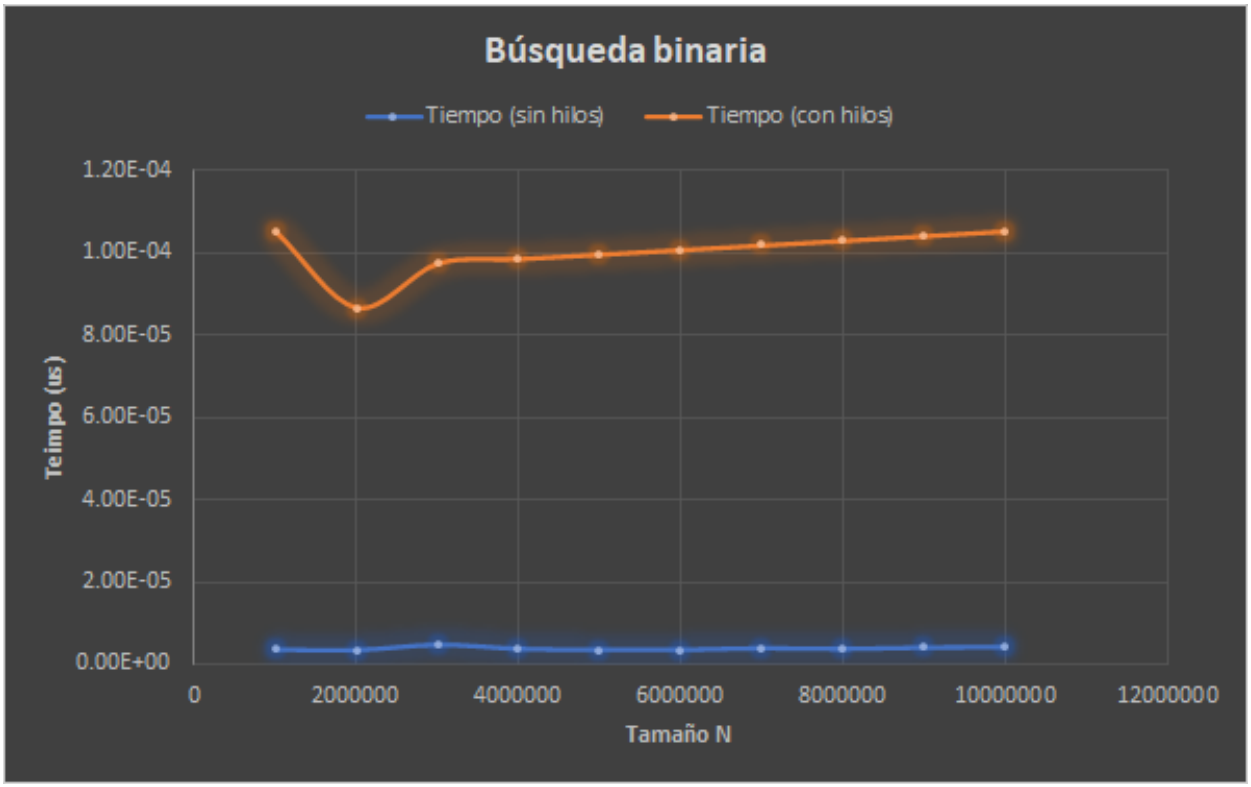
BÚSQUEDA CON ÁRBOL BINARIO

Tamaño de N	Tiempo real (versión sin hilos)	Tiempo real (versión con hilos)	Mejora
1000000	121.8601 s	120.8144 s	121.8601 s/120.8144 s =1.00
2000000	2,863.3152 s	483.46671 s	Mjora 100% 2,863.3152 s/483.46671 s =5.92
3000000	3521.8776 s	502.8053 s	Mejora 592% 3521.8776 s/502.80537 s =7.00
4000000	4331.9095 s	522.9175 s	Mejora 700% 4331.9095 s/522.9175 s =8.28
5000000	5328.2487 s	543.83429 s	Mejora 828% 5328.2487 s/543.83429 s =9.79
6000000	6553.7459 s	565.58766 s	Mejora 979% 6553.7459 s/565.58766 s =11.58
7000000	8061.1075 s	588.21117 s	Mejora 1158% 8061.1075 s/588.21117 s

8000000	9915.1622 s	611.73962 s	=13.70 Mejora 1370% 9915.1622 s/611.7396 s =16.20
9000000	12,195.6496 s	636.2092 s	Mejora 1620% 12,195.6496 s/636.2092 s =19.61
10000000	15,000.649 s	661.65757 s	Mejora 1961% 15,000.649 s/661.65757 s =22.67 Mejora 2267



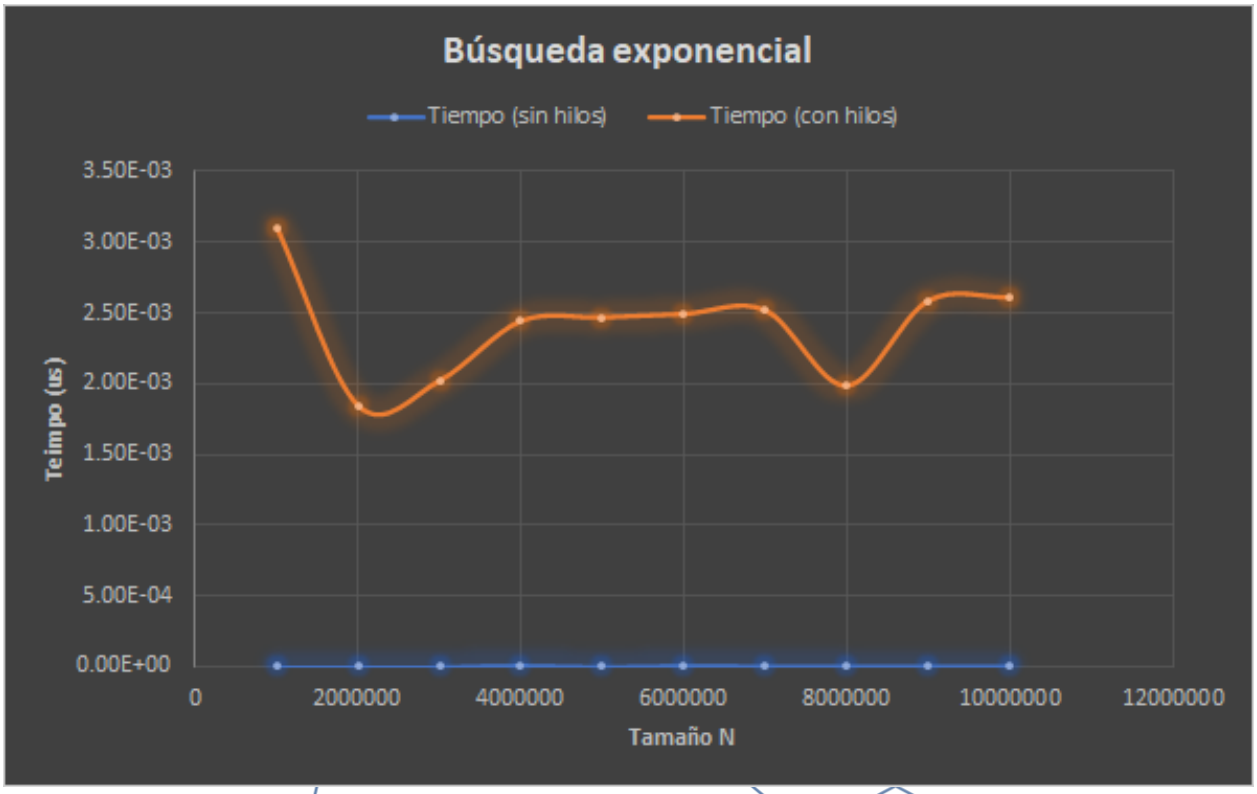
BÚSQUEDA BINARIA			
Tamaño de N	Tiempo real (versión sin hilos)	Tiempo real (versión con hilos)	Mejora
1000000	3.54085 us	105.46 us	3.54085 us/105.46 us =0.035 Mejora 3.5%
2000000	3.39746 us	86.7 us	3.39746 us/86.7 us =0.039% Mejora 3.9%
3000000	4.971025 us	97.567 us	4.971 us/97.567 us =0.050 Mejora 5.0%
4000000	3.76701 us	98.659 us	3.76701 us/98.659 us =0.038 Mejora 3.8%
5000000	3.44515 us	99.764 us	3.44515 us/99.764 us =0.034 Mejora 3.4%
6000000	3.4332 us	100.88 us	3.4332 us/100.88 us =0.034 Mejora 3.4%
7000000	3.9696 us	102.011 us	3.9696 us/102.011 us =0.038 Mejora 3.8%
8000000	3.71933 us	103.154 us	3.71933 us/103.154 us =0.036 Mejora 3.6%
9000000	4.1484 us	104.309 us	4.1484 us/104.309 us =0.039 Mejora 3.9%
10000000	4.29198 us	105.478 us	4.2919 us/105.478 us =0.040 Mejora 4.0%



BÚSQUEDA EXPONENCIAL

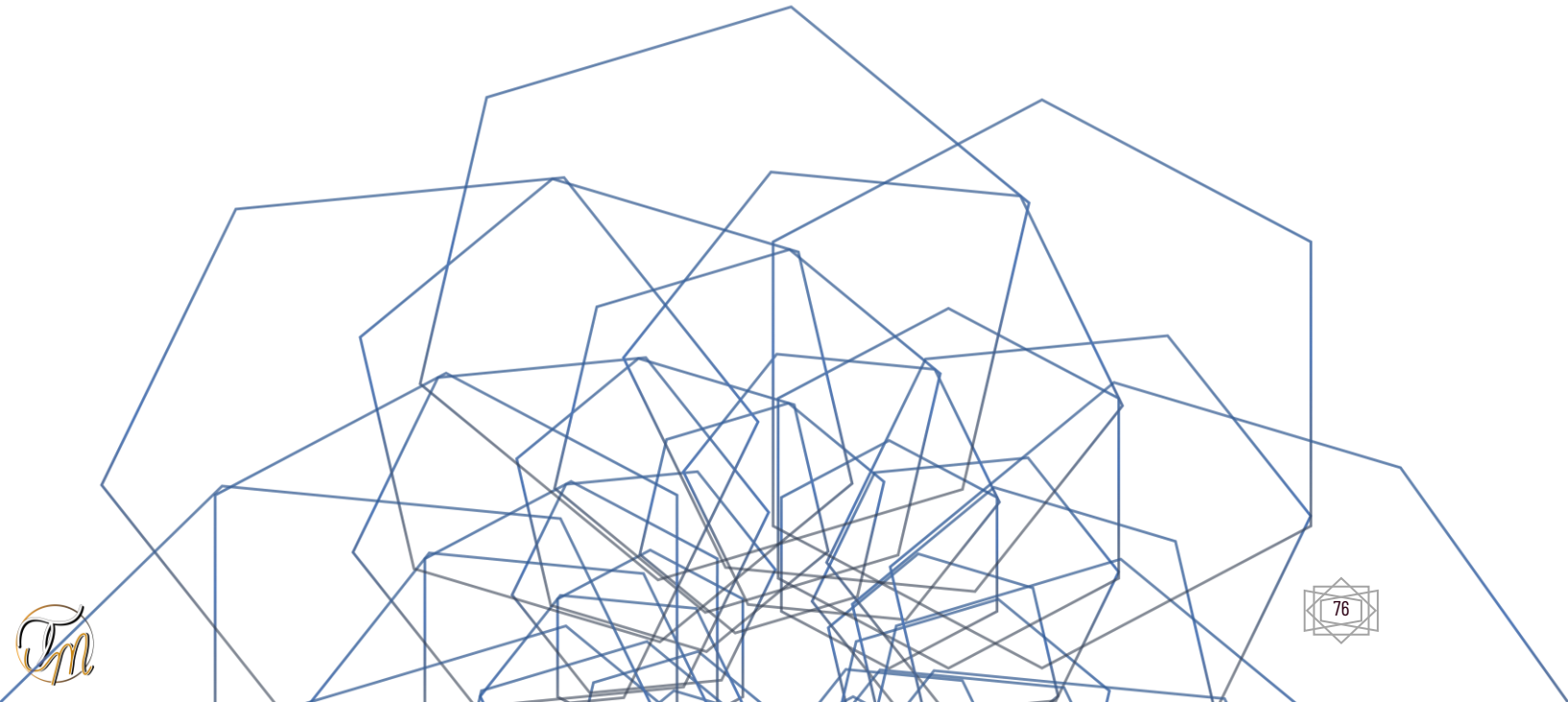
Tamaño de N	Tiempo real (versión sin hilos)	Tiempo real (versión con hilos)	Mejora
1000000	3.96967 us	3095.86 us	3.96967 us/3095.86 us =0.00128 Mejora 0.128%
2000000	4.13656 us	1842.73 us	4.13656 us/1842.73 us =0.00224 Mejora 0.224%
3000000	4.3153 us	2022.5 us	4.3153 us/2022.5 us =0.00213 Mejora 0.213%
4000000	5.9128 us	2443.07 us	5.9128 us/2443.07 us =0.00241 Mejora 0.241%
5000000	4.363 us	2467.5 us	4.363 us/2467.5 us =0.00176 Mejora 0.176%
6000000	5.6674 us	2492.17 us	5.6674 us/2492.17 us =0.00227 Mejora 0.227%
7000000	4.9471 us	2517.09 us	4.9471 us/2517.09 us

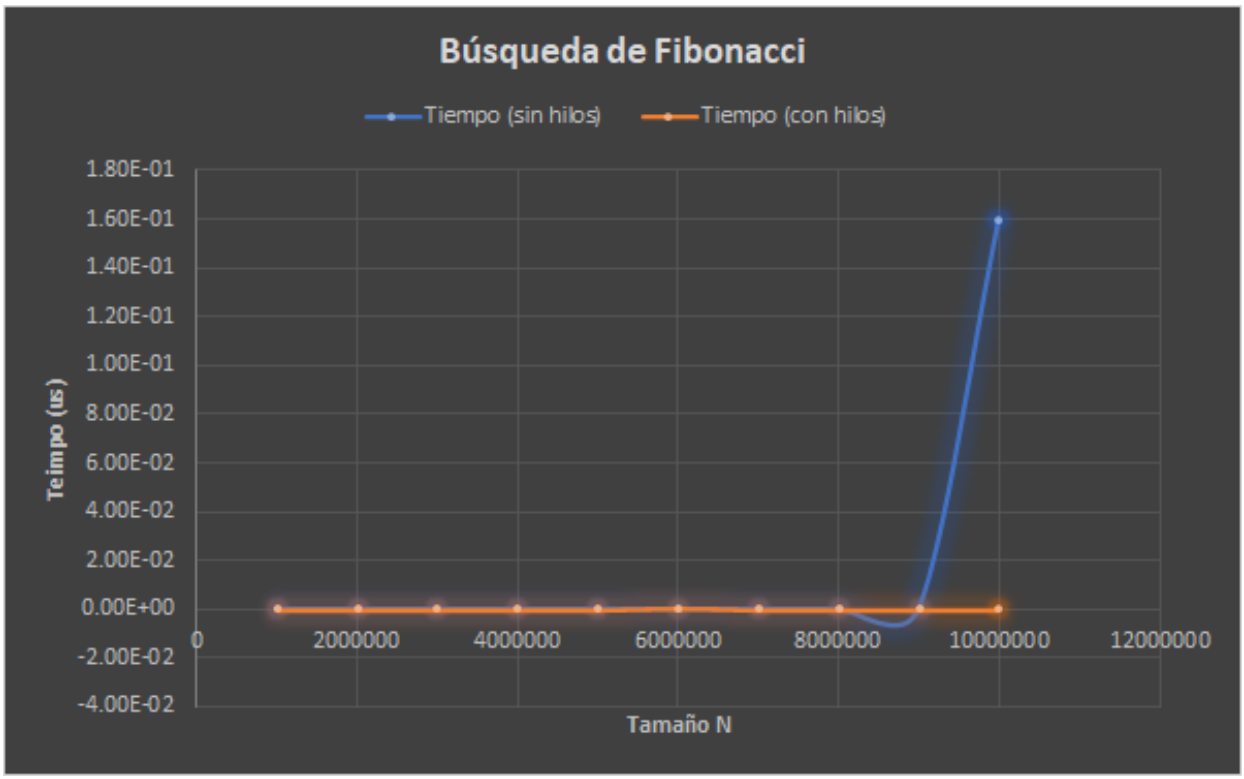
8000000	4.9591 us	1985.97 us	=0.00196 Mejora 0.196% 4.95910 us/1985.97 us =0.00249
9000000	4.8637 us	2584.2 us	Mejora 0.249% 4.8637 us/2584.2 us =0.00188
10000000	4.86639 us	2610.04 us	Mejora 0.188% 4.86639 us/2610.04 us =0.00186 Mejora 0.186%



BÚSQUEDA DE FIBONACCI			
Tamaño de N	Tiempo real (versión sin hilos)	Tiempo real (versión con hilos)	Mejora
1000000	3.5588 us	1.1563 us	3.5588 us/1.1563 us =3.07

2000000	3.80277 us	0.9059 us	Mejora 307% 3.8027 us/0.9059 us =4.19
3000000	4.2557 us	0.9211 us	Mejora 419% 4.2557 us/0.9211 us =4.61
4000000	3.91 us	0.9328 us	Mejora 461% 3.91 us/0.9328 us =4.19
5000000	3.9458 us	0.9446 us	Mejora 419% 3.9458 us/0.9446 us =4.17
6000000	6.032 us	15.02 us	Mejora 417% 6.032 us/15.02 us =0.401
7000000	0.2026 us	1.5211 us	Mejora 40.1% 0.2026 us/1.5211 us =0.133
8000000	4.2676 us	1.5515 us	Mejora 13.3% 4.2676 us/1.5515 us =2.73
9000000	4.1008 us	1.5825 us	Mejora 273% 4.1008 us/1.5825 us =2.59
10000000	0.15923 s	0.0000016142 s	Mejora 259% 0.15923 s/0.0000016142 s =98513.011




Búsqueda de Fibonacci
Tiempo (sin hilos)
Tiempo (con hilos)
1.80E-01
1.60E-01
1.40E-01
1.20E-01
1.00E-01
8.00E-02
6.00E-02
4.00E-02
2.00E-02
0.00E+00
-2.00E-02
-4.00E-02
Teimpo (us)
0
2000000
4000000
6000000
8000000
10000000
12000000
Tamaño N

CUESTIONARIO

- **¿Cuál de los 5 algoritmos es más difícil de implementar?**

  Búsqueda de árbol binario

- **¿Cuál de los 5 algoritmos es el más fácil de implementar?**

  Búsqueda lineal o secuencial

- **¿Cuál de los 5 algoritmos fue el más difícil de modelar en su variante con hilos?**

  La opinión en común de todo el equipo resultó ser que el algoritmo más complejo de modelar con respecto a su variante en hilos era el de Fibonacci, pues consideramos que era un poco más complejo de entender con respecto a lo que lógica conlleva, tomo un poco más de tiempo comprender la forma en que funcionaba

- **¿Cuál de los 5 algoritmos en su variante con hilos resultó ser más rápido?¿Por qué?**

  Búsqueda lineal o secuencial, a parte de que sin hilos es uno de los algoritmos más rápidos de realizar, a la hora de implementar un hilos este realiza dos comparaciones a la par, y cuando se encuentra el elemento que se quiere encontrar en una mitad del arreglo, para la otra mitad, ya no es necesario seguir con el procedimiento.

- **¿Cuál de los 5 algoritmos en su variante con hilos no representa ninguna ventaja?¿Por qué?**

  Desde nuestro punto de vista y con respecto a la experimentación no se nota una mejora significativa con el algoritmo de búsqueda mediante árbol binario, pues a pesar de que se divide para ir buscando en las ramas, este sigue realizando lo mismo parte de un punto y va bajando continuamente hasta encontrar el elemento buscado, en caso de que uno de los 2 hilos no lo haya encontrado parará pero el otro continuará hasta el final

- **¿Cuál algoritmo tiene menor complejidad temporal?**

  Búsqueda lineal o secuencial

- **¿Cuál algoritmo tiene mayor complejidad temporal?**

  Búsqueda exponencial y binaria

- **¿El comportamiento experimental de los algoritmos era esperado?¿Por qué?**

Parece ser que si, realmente las pruebas demostraron lo que habíamos inferido previo a la realización de las mismas con el uso de los conocimientos adquiridos y las investigaciones realizadas para comprender el funcionamiento de cada uno de los algoritmos de búsqueda

- **¿Sus resultados experimentales difieren mucho de los análisis teóricos que realizó?¿A qué se debe?**

No , consideramos que nuestro análisis resulto tener una buena aproximación a la realidad con lo que obtuvimos gracias a la ejecución de los códigos

- **En la versión con hilos, usar n hilos dividió el tiempo en n? ¿Lo hizo n veces más rápido?**

Si, realmente nos sorprendio el porcentaje de mejora que pudimos obtener en la mayoría de los algoritmos, pues se ve tanto en números como al momento de realizar la ejecución del mismo, ya que el equipo no tardaba mucho en proporcionar el archivo de salida que solicitábamos con los tiempos y nor pareció increíble la mejora, en su mayoría las pruebas fueron hechas usando solo 2 hilos pero al momento de agregar mas pudismo ver mejora

- **¿Cuál es el % de mejora que tiene cada uno de los algoritmos en su variante con hilos? ¿Es lo que esperabas? ¿Por qué?**

-Lineal: 225% en promedio

-Árbol: 1197% en promedio

-Binario: 3.8%

-Exponencial: 0.202% en promedio

-Fibonacci: 10112% en promedio

Al analizar los porcentajes de mejora, con el algoritmo de Fibonacci, no pensamos en la cantidad de porcentaje que nos salió, mientras que en del árbol igual nos salió una cantidad de porcentaje muy alto, pero por otro lado, el porcentaje del algoritmo de binario y exponencial, nos salieron cantidades muy pequeñas.

- **Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?**

El uso del mismo equipo con la misma cantidad de hilos y los mismos valores que tomaría en el arreglo.

Debido a que el equipo no es de un excelente rendimiento se le proporcionaban intervalos de “descanso” de 1 hora entre cada ejecución de script de los algoritmos

- **Si solo se realizará el análisis teórico de un algoritmo antes de implementarlo, podrías asegurar cual es el mejor?**

Realizando un análisis previo se podrían determinar bastantes cosas y tal vez llegar a dar una afirmación que pueda llegar a ser lo suficientemente certera, aunque no garantiza al 100% nada, pues se requiere de un análisis muy profundo, el cual se trató de realizar en el presente documento y consideramos que llegamos a resultados muy satisfactorios con respecto a esto compete

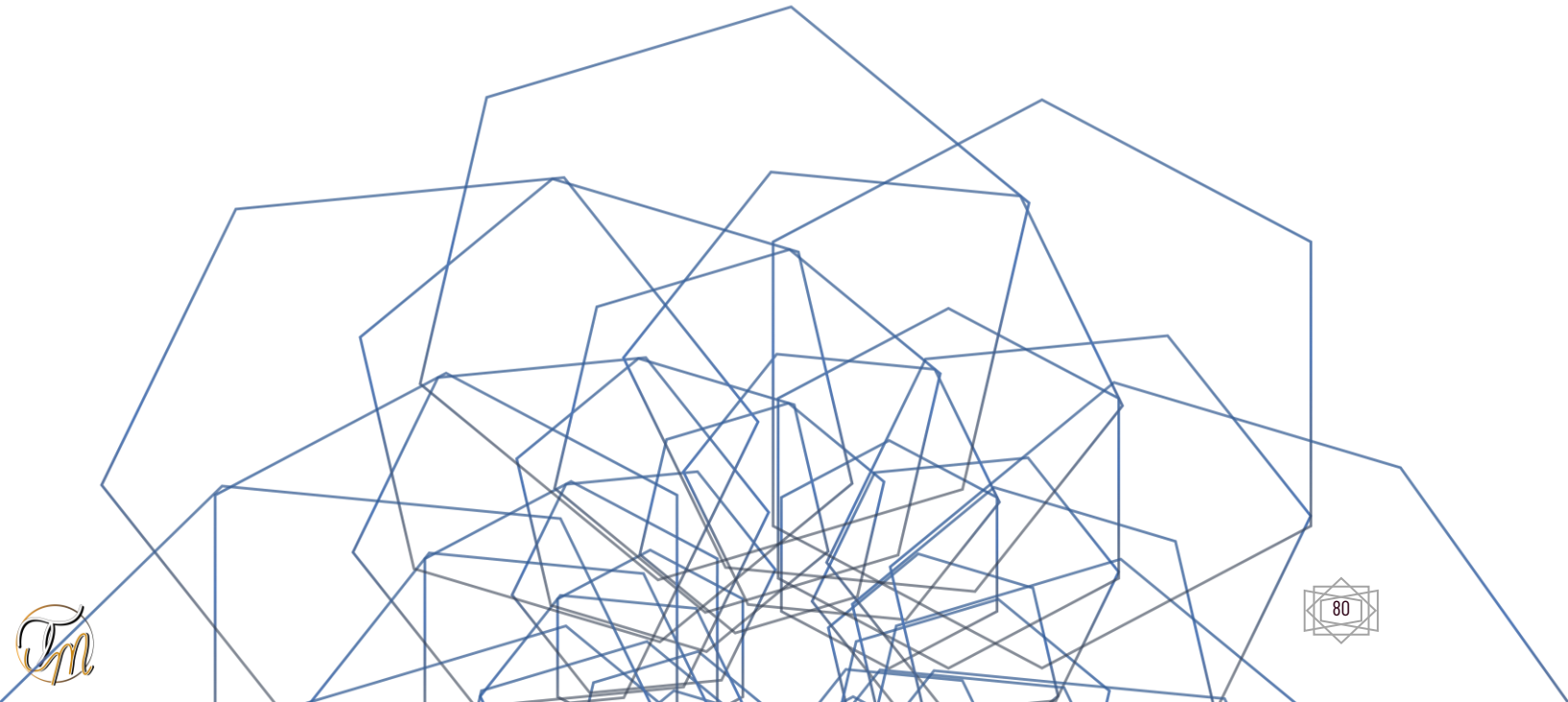
- **¿Qué tan difícil fue realizar el análisis teórico de cada algoritmo?**

El análisis teórico fue complejos debido a que se debían recurrir a varios conocimientos matemáticos de cierta forma un poco más sofisticados, pues como tal no estamos acostumbrados a tratar con ellos o bien practicarlos en el día a día, los comprendemos, claro, pero como tal no los fundamentamos, pero mediante documentación y apoyo mutuo consideramos haber llegado a un buen análisis y un buen establecimiento con respecto a lo teórico compete.

La parte de analizar como tal los algoritmos no fue muy compleja exceptuando la de Fibonacci, la cual para la mayoría del equipo resultó ser la más compleja de poder entender, pues como observamos es un algoritmo que tiene en su estructura otro algoritmo de secuencia numérica.

- **¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?**

Recomendamos que se planteen un tiempo para poder analizar los algoritmos como debe ser, de igual forma leer la documentación pertinente y crear previo un plan de trabajo desarrollo de código para poder realmente ir viendo resultados y no solo ir “picando piedra” y llegar a un resultado deficiente



PRUEBAS CON N

METODO	FORMULA	50000000	100000000	500000000	1000000000	5000000000
Lineal	$j = n - 1$	49999999	999999999	4999999999	99999999999	49999999999
Binaria	$f(n) = 8log_2(n-1) + 14$	218.603398	226.603398	245.178823	253.178823	271.754248
Exponencial	$f(n) = 10techo[log_2(n-1)] + 11$	266.754247	276.754247	299.973529	309.973529	333.192809
Árbol	$cont = 3n + 1$	150000001	300000001	1500000001	3000000001	1.5E+10
Fibonacci	$cont = 3x + 6(log_3(F(x+2))) + 9$	150000105.8	300000110	1500000118	3000000122	1.5E+10

OBTENCIÓN DE CONSTANTE MULTIPLICATIVA PARA CADA ALGORITMO

LINEAL

Para este algoritmo tenemos que,

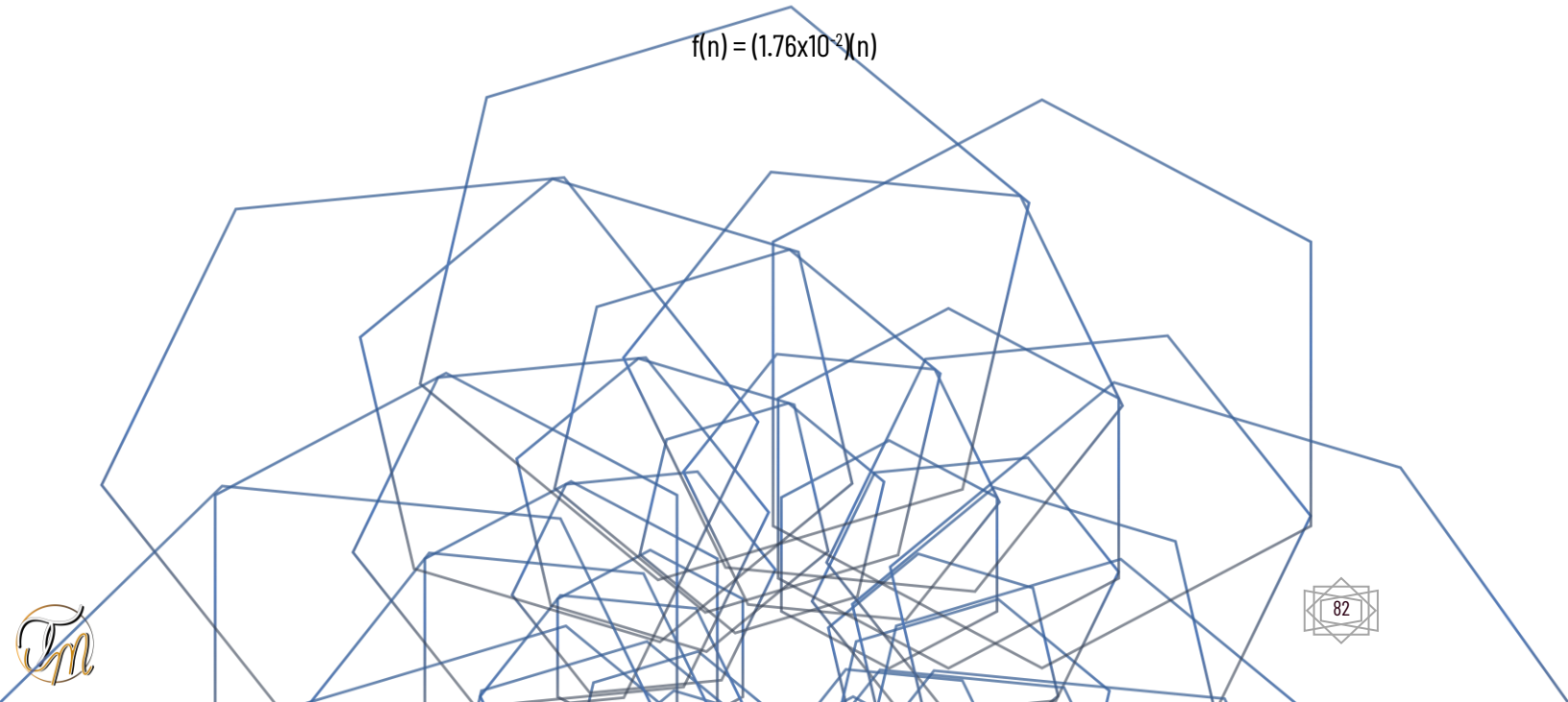
$$x = n$$

Realizamos los siguientes cálculos:

Tamaño de problema									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Número de operaciones									
4	7	10	13	16	19	22	25	28	31
Tiempo promedio obtenido de la ejecución del algoritmo									
0.0144229 7 s	0.0108 5474 s	0.0190 3196 s	0.0190 3196 s	0.0237 2143 s	0.0284 3812 s	0.0349 0495 s	0.0392 6047 s	0.043 5162 s	0.042 9584 s
Tiempo por operación básica (Tiempo promedio/número de operaciones)									
3.605E-03	1.550E-03	1.903E-03	1.463E-03	1.482E-03	1.496E-03	1.586E-03	1.570E-03	1.554 E-03	1.385 E-03
Tiempo promedio por operación básica									
1.76E-02									

Tomamos el valor del tiempo promedio que el algoritmo necesita para poder calcular el tiempo promedio para equis tamaño de n, entonces tenemos,

$$f(n) = (1.76x10^{-2})(n)$$



Binaria

Para este algoritmo tenemos que,

$$f(n)=8\log(2n-1)+14$$

Realizamos los siguientes cálculos:

Tamaño de problema									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Número de operaciones									
4	7	10	13	16	19	22	25	28	31
Tiempo promedio obtenido de la ejecución del algoritmo									
0.000003540855 s	0.000003397464971025 s	0.000004376701 s	0.000003445150 s	0.000003433239696 s	0.000003719330 s	0.000004148429198 s			
Tiempo por operación básica (Tiempo promedio/número de operaciones)									
885E-09	485.385E-09	376.7e-09	289.769E-09	215.312E-09	180.684E-09	156.045E-09	148.64E-09	148.142E-09	138.419E-09
Tiempo promedio por operación básica									
3.02E-06									

Tomamos el valor del tiempo promedio que el algoritmo necesita para poder calcular el tiempo promedio para equis tamaño de n, entonces tenemos,

$$f(n) = (3.02x10\text{-}6)(8\log 2n\text{-}1+14)$$

FIBONACCI

Para este algoritmo tenemos que,

$$f(n)=3x+6(\log 3(F(x+2)))+9$$

Realizamos los siguientes cálculos:

Tamaño de problema									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Número de operaciones									
18	22	26	30	34	38	42	45	49	52
Tiempo promedio obtenido de la ejecución del algoritmo									
3.55882 E-06	3.80277E -06	4.25577 E-06	0.00000 391	3.9458 E-06	0.000006 032	2.026E-07	4.26769 E-06	4.1008E -06	0.15923
Tiempo por operación básica (Tiempo promedio/número de operaciones)									
1.97712E -07	409.0450 28	1.63683 E-07	1.30333E -07	1.16053 E-07	1.58737E-07	4.82381 E-09	9.48376 E-08	8.36898 E-08	0.003062 115
Tiempo promedio por operación básica									
40.90480911									

Tomamos el valor del tiempo promedio que el algoritmo necesita para poder calcular el tiempo promedio para equis tamaño de n, entonces tenemos,

$$f(n) =(40.90480911)[3x+6(\log_3(F(x+2)))+9]$$

ARBOL

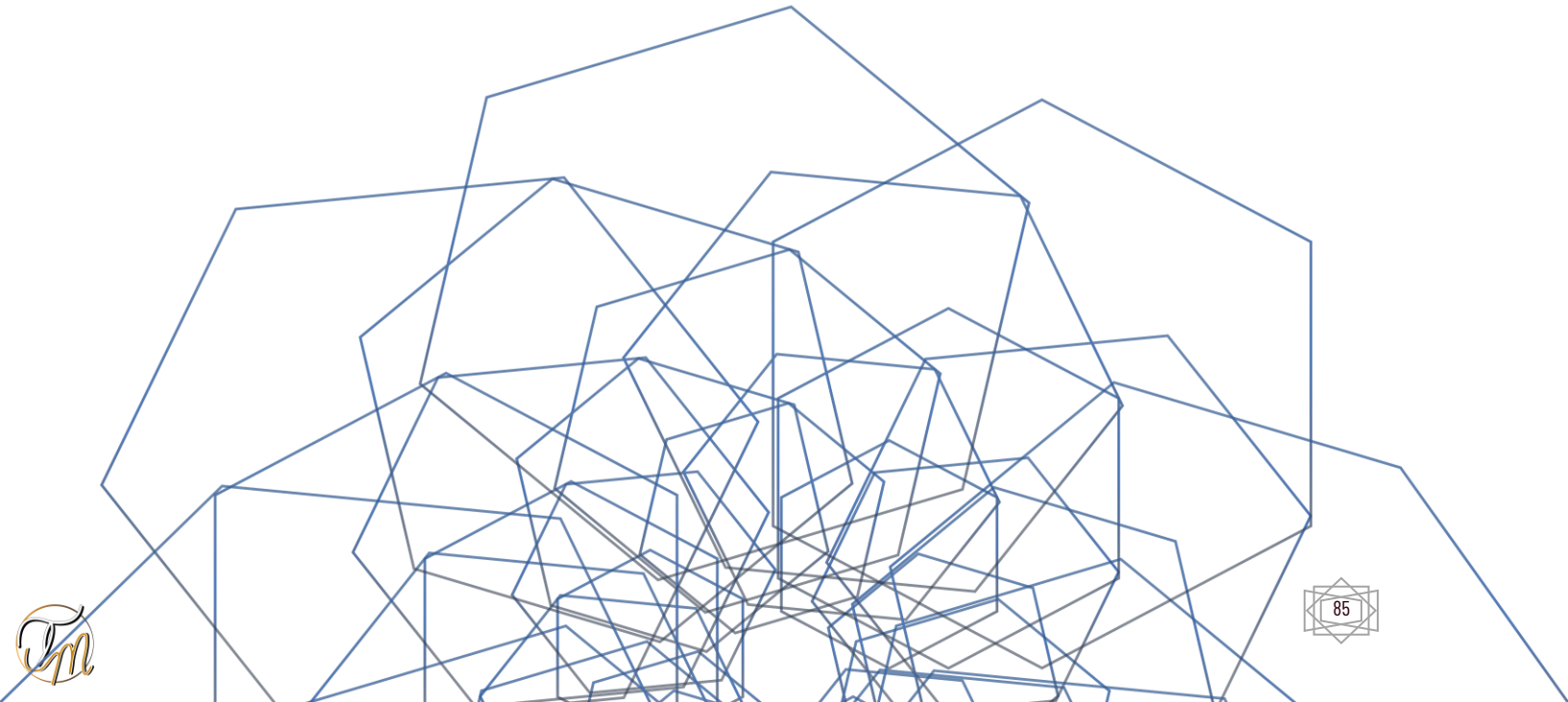
Para este algoritmo tenemos que,

$$f(n) = 3n+1$$

Tamaño de problema									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Número de operaciones									
4	7	10	13	16	19	22	25	28	31
Tiempo promedio obtenido de la ejecución del algoritmo									
121.8601917	2,863.3152	3521.8776	4331.9095	5328.2487	6553.7459	8061.1075	9915.1622	12195.6496	15000.649
Tiempo por operación básica (Tiempo promedio/número de operaciones)									
30.46504793	409.045028	352.18776	333.2238077	333.0155438	344.9339947	366.4139773	396.606488	435.5589143	483.8919032
Tiempo promedio por operación básica									
348.5342465									

Tomamos el valor del tiempo promedio que el algoritmo necesita para poder calcular el tiempo promedio para equis tamaño de n, entonces tenemos,

$$f(n) =(348.53)(3n+1)$$



EXPONENCIAL

Tamaño de problema									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Número de operaciones									
1M	2M	3M	4M	5M	6M	7M	8M	9M	10M
Tiempo promedio obtenido de la ejecución del algoritmo									
0.0000039696 7 s	0.0000041365 6 s	0.0000431537 5 s	0.0000059128 s	0.0000043630 6 s	0.0000056674 s	0.0000049471 s	0.0000049591 0 s	0.0000048637 s	0.0000048663 9 s
Tiempo por operación básica (Tiempo promedio/número de operaciones)									
992.25E-09	590.857E-09	431.5E-09	454.769E-09	272.685E-09	298.263E-09	224.863E-09	225.409E-09	173.678E-09	156.967E-09
Tiempo promedio por operación básica									
3.82E-06									

Tomamos el valor del tiempo promedio que el algoritmo necesita para poder calcular el tiempo promedio para equis tamaño de n, entonces tenemos,

$$f(n) = (3.82x10^{-6})(10techo(\log_2(n-1))+11)$$

CONCLUSIONES

A través de esta práctica pudimos reafirmar la importancia sobre la complejidad de los algoritmos y lo que significa el análisis de estos mismos para poder observar el comportamiento que cada algoritmo puede tener. Por otra parte, observar el cambio que un hilo puede generar en un algoritmo, ayuda a analizar de qué forma podemos hacer un algoritmo más eficiente o no, si implementar un nuevo "camino" nos beneficia o no a nosotros como programadores. Por último, a comparación de la práctica anterior, esta fue un poco más difícil y más larga de realizar por el análisis que se debía de realizar en cada punto de la práctica.

Jeon Jeong Paola

Esta práctica me sirvió para corroborar todo el estudio, que incluso podría ser considerado una ciencia, del análisis de algoritmos, pues la complejidad que radica encontrar el comportamiento de un algoritmo dado involucra muchas matemáticas y un raciocinio crítico, por lo que ahora comprendo cómo es que se decide cuál es la mejor solución a un problema al que le han encontrado diversas formas de solucionarse.

López López Oscar Manuel

Este enfoque resalta la importancia de los logaritmos de búsqueda porque ayudan a prevenir el posicionamiento de sitios que utilizan malas prácticas, además, cada algoritmo tiene pautas que permiten a los sitios lograr este posicionamiento, obligándolos a mejorar la calidad. Igualmente no solo se destaca la utilización de los algoritmos de búsqueda, si no de igual manera la utilización e implementación de hilos, pues a partir del uso de estos sabemos que aumentan la eficiencia de la comunicación entre programas en ejecución y con la realización de esta práctica nos muestra que podemos implementar los mismos en cualquier algoritmo.a

Lemus Ruiz Mariana Elizabeth

Esta práctica me ayudo muchísimo a poder realizar una implementación acerca de los conocimientos adquiridos hasta el momento dentro de la Unidad de Aprendizaje de Análisis de Algoritmos, pues me parece que los análisis que se tuvieron que llevar a cabo con respecto a los algoritmos correspondientes fueron de una dificultad media y a mi parecer logramos proporcionar un buen análisis mediante el trabajo en equipo y mediante la documentación solicitada. Me pareció una muy buena práctica para la implementación del conocimiento adquirido dentro de Sistemas Operativos, pues la implementación de hilos resulta ser muy importante y muy eficiente para la resolución de problemas y la posible reducción de tiempos cunado es necesario, aunque debemos saber implementarlo de forma correcta para no generar un fallo dentro de nuestro programa

Mora Ayala José Antonio

ANEXOS

A continuación, se presentan los programas involucrados en el desarrollo de este archivo mostrando el código de forma completa

La forma de ejecución del programa principal **MainLin.c debe ser realizada de la siguiente forma:**

./MainLin (tamaño del arreglo) (Opcion de algoritmo) (cantidad de hilos)

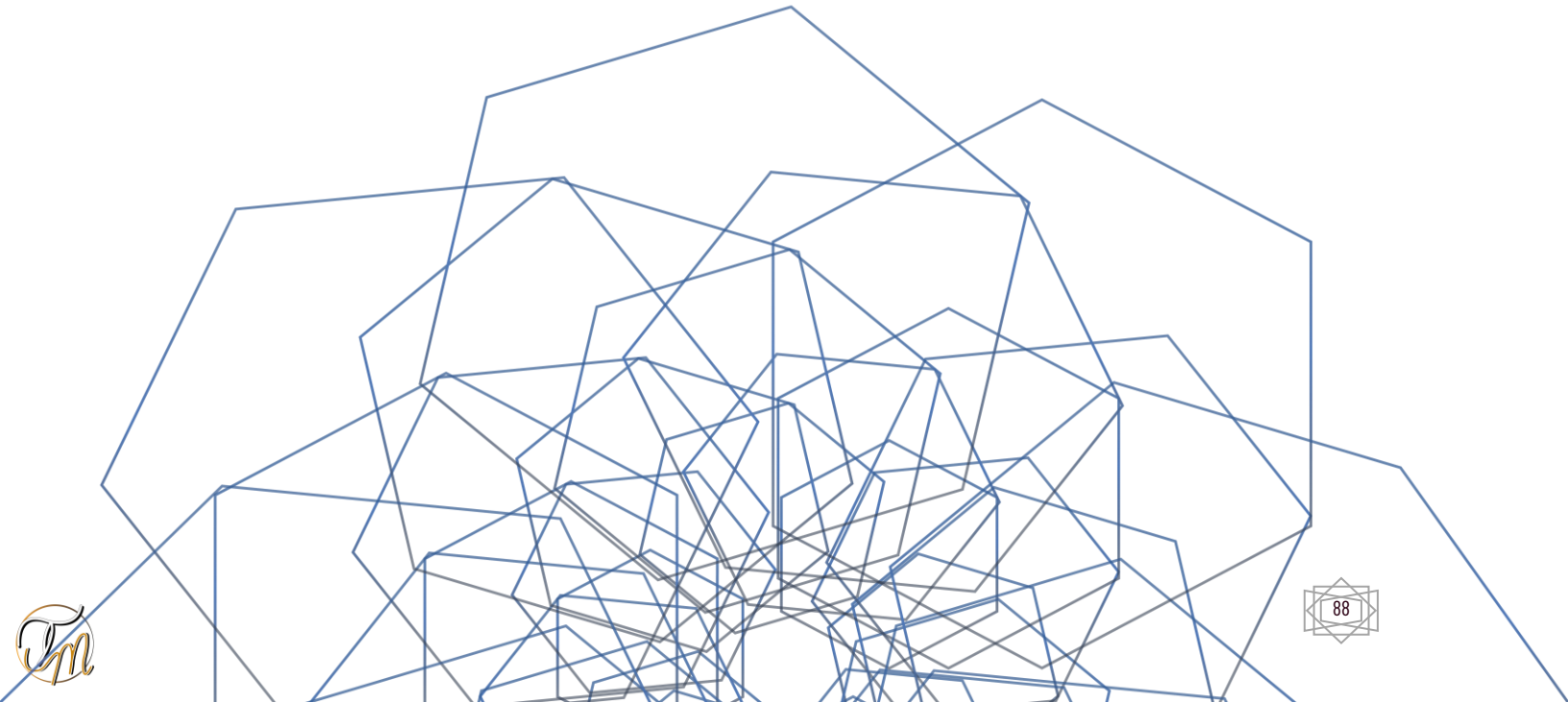
- Para las opciones 2 y 4 se debe elegir un valor mayor a 0
- Para las opciones 1,3,5,6,7,8 puede ser cualquier valor, de preferencia 0

1. Busqueda Lineal
2. Busqueda Lineal Hilos
3. Busqueda Binaria
4. Busqueda Binaria Hilos
5. Busqueda Fibonacci
6. Busqueda Arbol
7. Busqueda Arbol Hilos
8. Busqueda Exponencial

La búsqueda exponencial con Hilos y Fibonacci con Hilos se encuentran en documentos aparte.

Ejecución exponencial Hilos: ./exponencial (tamaño de n) (hilos)

Ejecución Fibonacci Hilos: ./FibonacciHilos (tamaño de n) (valor a buscar)



Archivo FuncionesH.h

```
//  3CM12
//  Analisis de Algoritmos
//  Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
//  Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
//  Cabeceras de funciones
#ifndef FuncionesH
#define FuncionesH

 void BusquedaLinealHilos(int * arreglo, int valorABuscar, int inicio, int final, int *
encontrado);
 void BusquedaLineal(int * arreglo, int valorABuscar, int inicio, int final, int *
encontrado);

/*  Mis hilos
************Busqueda Lineal Hilos*************
Recibe: Arreglo, tamaño del arreglo, elemento a buscar, (bandera para saber cuando el
emeento haya sido encontrado)
Devuelve: vacio
Funcion: Realizar la busqueda de un elemento dado mediante la opcion de Hilos
Uso: BusquedaLineal(A,0,n,valor,*aviso) */

void BusquedaLineal(int *A, int inicio ,int final, int elemn, int *aviso);
void BusquedaLinealHilos(int * A, int valorABuscar, int inicio, int final, int * aviso,
int hilos);
/*   Busqueda Binaria
Recibimos un arreglo que contiene los numeros del archivo 10 millones
 inicio : Inicio del arreglo
Un entero "final" para especificar el tamaño del arreglo
Un entero llamado elemento el cual tiene el número que deseamos encontrar,
dentro del arreglo
 *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no */
void BusquedaBinaria(int *A,int inicio, int final,int elem,int *aviso);

/* ***********************************BUSQUEDA BINARIA
HILOS*************************************
Recibimos un arreglo que contiene los numeros del archivo 10 millones
 inicio : Inicio del arreglo
Un entero "final" para especificar el tamaño del arreglo
Un entero llamado elemento el cual tiene el número que deseamos encontrar,
dentro del arreglo
 *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no */
void BusquedaBinariaHilos(int * A, int valorABuscar, int inicio, int final, int *
aviso, int hilos);

/* *************BUSQUEDA ARBOL*************
Recibimos un arreglo que contiene los numeros del archivo 10 millones
 inicio : Inicio del arreglo
Un entero "final" para especificar el tamaño del arreglo
Un entero llamado elemento el cual tiene el número que deseamos encontrar,
dentro del arreglo
```

```
 *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no */
void BusquedaEnArbol(int * arreglo, int n, int valorABuscar, int * aviso);
/* *********BUSQUEDA EN ARBOL HILOS*************
Recibimos un arreglo que contiene los numeros del archivo 10 millones
inicio : Inicio del arreglo
Un entero "final" para especificar el tamaño del arreglo
Un entero llamado elemento el cual tiene el número que deseamos encontrar,
dentro del arreglo
 *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no */
void BusquedaEnArbolHilos(int * arreglo, int n, int valorABuscar, int * aviso);
 /* Funcion para devolver el minimo de dos numeros */
int min(int x, int y);

 /* Recibe arreglo, valor a buscar y tamaño del arreglo */
int fibMonaccianSearch(int *A, int x, int n);
/* *********BUSQUEDAEXPONENCIAL*************
Recibimos un arreglo que contiene los numeros del archivo 10 millones
inicio : Inicio del arreglo
Un entero "final" para especificar el tamaño del arreglo
Un entero llamado elemento el cual tiene el número que deseamos encontrar,
dentro del arreglo
 *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no */
int BusquedaExponencial(int *A, int n, int elem, int *aviso);

 /* Prototipos para afcilitar ejecucion de Hilos */
void * lanzarBusquedaLineal(void* busqueda);
void * lanzarBusquedaBinaria(void* busqueda);
void * procesarBusquedaArbol(void* busqueda);
/*
 Funcion para leer el archivo 10millones ordenados
 Recibe : Arreglo de elementos con memoria suficiente para alojar la n cantidad de
elementos
 n: Cantidad de elementos que seran asignados al arreglo */
int *LeerArchivo(int *A, int n);
/* *********QUICKSORT*************
Algoritmo de ordenamiento auxiliar para proporrcionar los datos de los numeros en el
orden correcto (numeros a buscar solicitados)
Recibe:
  A:Arreglo de elementos
  primera posicion
  ultimo: Tamaño del arreglo (o hasta donde se quieran obtener los numeros ordenados)
Devuelve:
  Arreglo Ordenado */
int* Quicksort2(int *A, int primero, int ultimo);
#endif
```

FuncionesH.c

```c
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
// Implementacion de funciones
#include "FuncionesH.h"
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "../FuncionesHilos/Arbol.c"

//  //Estrucutras auxiliares para la ejecucion de los códigos
//  //Nos ofrecen la alternativa de almacenar los valores dependiento de cada uno de los hilos
//  //De tal forma que podremos ALMACENAR LOS PARAMETROS MANDADOS A LAS UNCIONES CON HILOS
typedef struct auxiliarBusquedaLineal
{
    int *arrelgo;
    int valorABuscar;
    int inicio;
    int final;
    int *encontrado;
} AuxiliarLineal;
typedef struct auxiliarBusquedaBinaria
{
    int *arrelgo;
    int valorABuscar;
    int inicio;
    int final;
    int *encontrado;
} AuxiliarBinaria;

typedef struct auxiliarBusquedaEnArbol{
    arbol t;
    int valorABuscar;
    int * encontrado;
}AuxiliarArbol;


// // Busqueda Lienal
//Recibimos un arreglo que contiene los numeros del archivo 10 millones
// inicio : Inicio del arreglo
//Un entero "final" para especificar el tamaño del arreglo
//Un entero llamado elemento el cual tiene el número que deseamos encontrar,
//dentro del arreglo
// *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no
void BusquedaLineal(int *A, int inicio, int final, int elem, int *aviso)
{
```

```
    //- Variables para el ciclo
    int n;
    for (n = inicio; n < final; n++)
    {
        // Cuando aviso haya cambiado de valor terminara este algoritmo de busqueda
        if (*aviso >= 0)
        {
            break;
        }
        //Nos ayuda a comparar las posiciones que se encuentran en el arreglo
        //con la posición del elemento que se desea encontrar
        if (A[n] == elem)
        {
            // printf("El elemento fue encontrado en la posicion numero: %d \n", n);
            *aviso = elem;
        }
    }
}
// // Busqueda Lineal Hilos
//Recibimos un arreglo que contiene los numeros del archivo 10 millones
// inicio : Inicio del arreglo
//Un entero "final" para especificar el tamaño del arreglo
//Un entero llamado elemento el cual tiene el número que deseamos encontrar,
//dentro del arreglo
// *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no
void BusquedaLinealHilos(int *A, int valorABuscar, int inicio, int final, int *aviso)
{

    // - Variables para el manejo del algoritmo
    int puntomedio = (final - inicio) / 2;
    if (puntomedio > 0)
    {
        //- Creacion de los hilos que contendran partes del arreglo en cuestion:
        pthread_t *hilo;
        hilo = malloc(2 * sizeof(pthread_t));
        //- Repartiendo el arreglo mediante el uso de las estructuras que creamos
        AuxiliarLineal *der = (AuxiliarLineal *)malloc(sizeof(AuxiliarLineal));
        der->arrelgo = A;
        der->valorABuscar = valorABuscar;
        der->inicio = puntomedio + 1;
        der->final = final;
        der->encontrado = aviso;
        AuxiliarLineal *izq = (AuxiliarLineal *)malloc(sizeof(AuxiliarLineal));
        izq->arrelgo = A;
        izq->valorABuscar = valorABuscar;
        izq->inicio = inicio;
        izq->final = puntomedio;
        izq->encontrado = aviso;

        // -Creando los Hilos
        //- La funcion naturalmente devuelve 0 en caso de que el hilo se haya creado
```

```
de forma exitosa, de lo contrario devolvera otro valor
        if (pthread_create(&hilo[0], NULL, lanzarBusquedaLineal, (void *)izq) != 0)
        {
            perror("El hilo nos e pudo crear");
            exit(-1);
        }
        if (pthread_create(&hilo[1], NULL, lanzarBusquedaLineal, (void *)der) != 0)
        {
            perror("El hilo nos e pudo crear");
            exit(-1);
        }
        // - Variables manejo de ciclo for
        int i;
        // - Hace que los hilos terminen y posteriormente se vuelvan a unir al
programa principal (Hasta que los hilos acaben acaba todo)
        // -Si el main termina primero tendra que mantenerse en espera
        for (i = 0; i < 2; i++)
            pthread_join(hilo[i], NULL);
        free(hilo);
    }
}

// // Busqueda Binaria
//Recibimos un arreglo que contiene los numeros del archivo 10 millones
// inicio : Inicio del arreglo
//Un entero "final" para especificar el tamaño del arreglo
//Un entero llamado elemento el cual tiene el número que deseamos encontrar,
//dentro del arreglo
// *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no

void BusquedaBinaria(int *A, int inicio, int final, int elem, int *aviso)
{
    // -Variables para algoritmo de busqueda
    int mitad;
    while (inicio <= final)
    {
        mitad = (inicio + final) / 2;
        if (A[mitad] < elem)
        {
            inicio = mitad + 1;
        }

        else
        {
            final = mitad - 1;
        }

        if (A[mitad] == elem)
        {
            *aviso = elem;
            // printf("Enontraste el emento en la posicion: %d\n", mitad);
```

```
            break;
        }
    }
}

// // Busqueda Binaria Hilos // //
// // Busqueda Binaria
//Recibimos un arreglo que contiene los numeros del archivo 10 millones
// inicio : Inicio del arreglo
//Un entero "final" para especificar el tamaño del arreglo
//Un entero llamado elemento el cual tiene el número que deseamos encontrar,
//dentro del arreglo
// *aviso: Bandera que nos ayudara a determianr si el elemento ya fue encontrado o no

void BusquedaBinariaHilos(int *A, int valorABuscar, int inicio, int final, int *aviso)
{

    // - Variables para el manejo del algoritmo
    int puntomedio = (final - inicio) / 2;
    if (puntomedio > 0)
    {
        //- Creacion de los hilos que contendran partes del arreglo en cuestion:
        pthread_t *hilo;
        hilo = malloc(2 * sizeof(pthread_t));
        //- Repartiendo el arreglo mediante el uso de las estructuras que creamos
        AuxiliarBinaria *der = (AuxiliarBinaria *)malloc(sizeof(AuxiliarBinaria));
        der->arrelgo = A;
        der->valorABuscar = valorABuscar;
        der->inicio = puntomedio + 1;
        der->final = final;
        der->encontrado = aviso;
        AuxiliarBinaria *izq = (AuxiliarBinaria *)malloc(sizeof(AuxiliarBinaria));
        izq->arrelgo = A;
        izq->valorABuscar = valorABuscar;
        izq->inicio = inicio;
        izq->final = puntomedio;
        izq->encontrado = aviso;

        // -Creando los Hilos
        //- La funcion naturalmente devuelve 0 en caso de que el hilo se haya creado
de forma exitosa, de lo contrario devolvera otro valor
        if (pthread_create(&hilo[0], NULL, lanzarBusquedaBinaria, (void *)izq) != 0)
        {
            perror("El hilo nos e pudo crear");
            exit(-1);
        }
        if (pthread_create(&hilo[1], NULL, lanzarBusquedaBinaria, (void *)der) != 0)
        {
            perror("El hilo nos e pudo crear");
            exit(-1);
        }
```

```
        // - Variables manejo de ciclo for
        int i;
        // - Hace que los hilos terminen y posteriormente se vuelvan a unir al
programa principal (Hasta que los hilos acaben acaba todo)
        // -Si el main termina primero tendra que mantenerse en espera
        for (i = 0; i < 2; i++)
            pthread_join(hilo[i], NULL);
        free(hilo);
    }
}

// //          Busqueda con Serie de Fibbonacci           // //
//Función para encontrar el mínimo de dos números
int min(int x, int y) { return (x <= y) ? x : y; }

//Regresa "i" si el elemento se encuentra en el arreglo
//Si el elemento no se encuentra, regresa un -1

//Declaramos un arreglo
//Un entero "n" para especificar el tamaño del arreglo
//Un entero llamado n el cual tiene el número que deseamos encontrar,
//dentro del arreglo
int fibMonaccianSearch(int *A, int x, int n)
{
    //Inicializamos los primeros dos números de la serie
    //Estos números siempre son continuos
    int fibMMm2 = 0;
    int fibMMm1 = 1;

    //La suma de los números anteriores dan el número que sigue en la serie
    int fibM = fibMMm2 + fibMMm1;

    //Mientras fibM es menor al tamaño del arreglo,
    //Los valores se irán "recorriendo" de variable
    //Y la suma de los números anteriores,
    //serán el valor de fibM
    while (fibM < n)
    {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    //Nos marca el rango de la izquierda donde no se encuentra el elemento deseado
    int offset = -1;

    while (fibM > 1)
    {
        int i = min(offset + fibMMm2, n - 1);

        //Si el elemento es mayor al valor del índice,
```

```
        //Lo valores se irán recorriendo y el "offset"
        //se cambiará al valor del índice,
        if (A[i] < x)
        {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        }

        //Si el elemento es mayor a la posición del arreglo en la posiciónde fibMm2
        //el arreglo se dividirá después de la posición i+1
        else if (A[i] > x)
        {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        //Si el elemento es encontrado, devolvemos "i"
        else
            return i;
    }

    //Comparamos el último valor del arreglo con el elemento
    if (fibMMm1 && A[offset + 1] == x)
        return offset + 1;

    //Si el elemento no es encontrado, devolvemos un -1
    return -1;
}


//Busqueda en Arbol
void BusquedaEnArbol(int * arreglo, int n, int valorABuscar, int * aviso)
{
    // Variable para "almacenar" el árbol
    arbol t;
    // Inicializando arbol
    Iniciar(&t);
    // Rellenamos el arbol
    int i;
    // Variable para poder controlar el ciclo
    for(i=0;i < n; i++)
    {
        NuevoNodo(&t,arreglo[i]);
    }
    // Lanzamos la búsqueda sin hilos
    BuscaValor(&t, valorABuscar, aviso);
}
```

```
void BusquedaEnArbolHilos(int * arreglo, int n, int valorABuscar, int * aviso)
{
    // Variable para "almacenar" el árbol
    arbol t;
    // Creamos nuestro árbol
    Iniciar(&t);
    // Rellenamos el arbol
    int i;
    for(i=0;i < n; i++)
    {
        NuevoNodo(&t,arreglo[i]);
    }
    /* Parte de hilos */
    // Comprobamos si nuestro valor a buscar
    // no está en la raíz:
    if((*t).valor == valorABuscar)
    {
        *aviso = (*t).valor;
    }
    else
    {
        // Si no está, comenzamos la
        // repartición de los subarboles
        pthread_t *thread;
        thread = malloc(2*sizeof(pthread_t));
        // Creamos los auxiliares
        AuxiliarArbol * izq = (AuxiliarArbol *)malloc(sizeof(AuxiliarArbol));
        izq->t = (*t).izquierdo;
        izq->valorABuscar = valorABuscar;
        izq->encontrado = aviso;


        AuxiliarArbol * der = (AuxiliarArbol *)malloc(sizeof(AuxiliarArbol));
        der->t = (*t).derecho;
        der->valorABuscar = valorABuscar;
        der->encontrado = aviso;

        // Creamos los hilos
        if(pthread_create(&thread[0], NULL, procesarBusquedaArbol, (void *)izq) != 0)
        {
            perror("El thread no  pudo crearse [Arbol]\n");
            exit(-1);
        }
        if(pthread_create(&thread[1], NULL, procesarBusquedaArbol, (void *)der) != 0)
        {
            perror("El thread no  pudo crearse [Arbol]\n");
            exit(-1);
        }
        // Esperamos a los hilos
        int i;
        for (i=0; i<2; i++) pthread_join (thread[i], NULL);
```

```
        free(thread);
    }
}

int BusquedaExponencial(int *A, int n, int elem, int *aviso)
{
    // -Variables para algoritmo de busqueda
    int inicio, final, mitad, i;
    i = 1;
    final = n;
    while (i < final && A[i] <= elem)
    {
        i = i * 2;
    }

    inicio = (i / 2);
    final = (n - 1);

    while (inicio <= final)
    {
        mitad = (inicio + final) / 2;
        if (A[mitad] == elem)
        {
            *aviso = mitad;
            return mitad;
        }
        if (A[mitad] > elem)
        {
            final = mitad - 1;
        }
        else
        {
            inicio = mitad + 1;
        }
    }
    *aviso=-1;
    return -1;
}


// Nos ayudará a lanzar la búsqueda en árbol por cada hilo
void *lanzarBusquedaLineal(void *busqueda)
{
    AuxiliarLineal *l = (AuxiliarLineal *)busqueda;
    BusquedaLineal(l->arrelgo, l->inicio, l->final, l->valorABuscar, l->encontrado);
}

void *lanzarBusquedaBinaria(void *busqueda)
{
    AuxiliarBinaria *a = (AuxiliarBinaria *)busqueda;
    BusquedaBinaria(a->arrelgo, a->inicio, a->final - 1, a->valorABuscar, a-
```

```
>encontrado);
}

void * procesarBusquedaArbol(void* busqueda)
{
    AuxiliarArbol * b = (AuxiliarArbol *)busqueda;
    BuscaValor(&((*b).t), (*b).valorABuscar, (*b).encontrado);
}

// .Lectura del Archivo
// .Recibe:  Arreglo con memoria suficiente para almacenar n cantidad de enteros
// .Devuelve: Arreglo con los valores insertados dentro de el

int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}

int *Quicksort2(int *A, int primero, int ultimo)
{
    int piv, i, j, central, aux;
    central = (primero + ultimo) / 2;
    piv = A[central], i = primero, j = ultimo;
    do
    {
        while (A[i] < piv)
        {
            i++;
        }
        while (A[j] > piv)
        {
            j--;
        }
        if (i <= j)
        {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
            j--;
```

```
        }
    } while (i <= j);
    if (primero < j)
    {
        Quicksort2(A, primero, j);
    }
    if (primero < ultimo)
    {
        Quicksort2(A, i, ultimo);
    }
}
```

MainLin.c

```
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "./Funciones/FuncionesH.c"
#include "./Tiempo/tiempo.c"
// numeros = Arreglo leido
// n = tamaño del arreglo

// Tomara los argumentos que se coloquen al momento de ejecutar, en caso de ejecutar
sin argumentos
// pedira que ingreses una opcion del 1 al 9 y los elementos a ordenar por defecto
sera de 100 numeros

int main(int argc, char const *argv[])
{
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medición
de tiempos
    int i, l, m, j;                                        //Variables para loops
    int n, opc, *A;                                        //Variables de : (tamaño de
problema), (opcion), (Arreglo dinamico) respectivamente
    int Arreglo[20] = {322486, 14700764, 3128036, 6337399, 61396, 10393545,
2147445644, 1295390003, 450057883, 187645041, 1980098116, 152503, 5000, 7500, 214826,
1843349527, 1321906174, 2109248666, 2147470852, 0};
    // Aviso que nos servira de apoyo para saber cuando un elemento ya haya sido
encontrado
    // En todos los casos cuando aviso>0 el elemento habra sido encontrado
    int aviso = -1;
    // Tomando los valores de los argumentos de ejecucion
    n = atoi(argv[1]);
```

```
    // numero=atoi(argv[2]);
    opc = atoi(argv[2]);
    // Asignacion de memoria del arreglo con respecto al tamaño de problema solicitado
    A = (int *)malloc(n * sizeof(int));
    // Lectura y asignacion de valores al arreglo en cuestion
    LeerArchivo(A, n);
    //Implementacion de un Algoritmo de Ordenamiento para ir evaluando la busqueda del
arreglo de forma ordenada y facilitar recoleccion de datos
    Quicksort2(Arreglo, 0, 20);

    // Ciclo para poder proporcionar un menu de opcion y seleccionar el algoritmo
deseado
    // Durante el ciclo de acuerdo a la opcion seleccionada proporcionara los
resultados en caso de que el valor del
    // aviso haya cambiado
    do
    {
        printf("///////////////Valor de n %d//////////////////\n\n", n);
        // scanf("%d", &opc);
        switch (opc)
        {
        case 1:
            for (m = 0; m < 20; m++)
            {
                uswtime(&utime0, &stime0, &wtime0);
                BusquedaLineal(A, 0, n, Arreglo[m], &aviso);
                uswtime(&utime1, &stime1, &wtime1);

                if (aviso > 0)
                {
                    printf("Busqueda Lineal numero: %d\n", Arreglo[m]);
                    printf("real:%.10e s\n", wtime1 - wtime0);
                    printf("\n");
                }
                else
                {
                    printf("NoE: %d\n", Arreglo[m]);
                    printf("real  %.10e s\n", wtime1 - wtime0);
                    printf("\n");
                }

                aviso = -1;
            }

            break;
        case 2:
            for (j = 0; j < 20; j++)
            {
                uswtime(&utime0, &stime0, &wtime0);
                BusquedaLinealHilos(A, Arreglo[j], 0, n, &aviso);
                uswtime(&utime1, &stime1, &wtime1);
```

```
            if (aviso > 0)
            {
                printf("Lineal Hilos %d\n", Arreglo[j]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[j]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        break;
    case 3:
        printf("Metodo de Busqueda Binaria");
        for (m = 0; m < 20; m++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            BusquedaBinaria(A, 0, n, Arreglo[m], &aviso);
            uswtime(&utime1, &stime1, &wtime1);

            if (aviso > 0)
            {
                printf("Binaria%d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        break;
    case 4:
        printf("Metodo de Busqueda Binaria Hilos");
        int h;
        h = 0;
        for (h = 0; h < 20; h++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            BusquedaBinariaHilos(A, Arreglo[h], 0, n, &aviso);
            uswtime(&utime1, &stime1, &wtime1);
```

```
            if (aviso > 0)
            {
                printf("BH: %d\n", Arreglo[h]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[h]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        break;
    case 5:
        printf("Metodo de Fibo\n");
        for (m = 0; m < 20; m++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            aviso = fibMonaccianSearch(A, Arreglo[m], n);
            uswtime(&utime1, &stime1, &wtime1);

            if (aviso > 0)
            {
                printf("Busqueda Fibo %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[m]);
                printf("real  %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        // BusquedaEnArbol(A, n, elem,&encontrado);
        break;

    case 6:
        printf("Metodo de Arbol \n");
        for (m = 0; m < 20; m++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            BusquedaEnArbol(A, n, Arreglo[m], &aviso);
            uswtime(&utime1, &stime1, &wtime1);

            if (aviso > 0)
```

```
            {
                printf("Busqueda Arbol elemento : %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        break;
    case 7:
        for (m = 0; m < 20; m++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            BusquedaEnArbolHilos(A, n, Arreglo[m], &aviso);
            uswtime(&utime1, &stime1, &wtime1);

            if (aviso > 0)
            {
                printf("Busqueda Arbol Hilos elemento: %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
            else
            {
                printf("NoE: %d\n", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }

            aviso = -1;
        }
        break;
    case 8:
        for (m = 0; m < 20; m++)
        {
            uswtime(&utime0, &stime0, &wtime0);
            BusquedaExponencial(A, n, Arreglo[m], &aviso);
            uswtime(&utime1, &stime1, &wtime1);

            if (aviso > 0)
            {
                printf("Busqueda Exponencial: %d ", Arreglo[m]);
                printf("real %.10e s\n", wtime1 - wtime0);
                printf("\n");
            }
```

```
                else
                {
                    printf("NoE: %d ", Arreglo[m]);
                    printf("real %.10e s\n", wtime1 - wtime0);
                    printf("\n");
                }

                aviso = -1;
            }
            break;

        default:
            break;
        }
        opc = 0;
    } while (opc != 0);

    return 0;
}
```

FibonacciHilos.c

```
//Librerias incluidas
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
// Algoritmo de Busqueda por Fibonacci con la implementacion de hilos
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "./Tiempo/tiempo.c"

/* Prototipo de la funcion de lectura del archivo de los 10 millones de numeros */
int *LeerArchivo(int *A, int n);
struct argumentos {
    int *arr;
    int fib2;
    int fib1;
    int n;
    int target;
};

int min(int x, int y) { return (x<=y)? x : y; }

/*
    Funcion: fibonacci
    Esta realiza una busqueda Fibonacci en el intervalo dado
    Lo que se recibe: (Struct) args que contienen los numeros Fibonacci donde inicia
la busqueda,
```

```
    el numero a buscar y el arreglo donde buscar.
    Imprime si ha encontrado el numero
*/
void *fibonacci(void *ags){
    double utime0, stime0, wtime0, utime1, stime1, wtime1; //Variables para medici□n
de tiempos
    struct argumentos *args = (struct argumentos*) ags;
    int status = 0, offset = 0;
    // Inizializa numeros fibonacci
    int fibMMm2 = args->fib2; // (m-2)No. Fibonacci
    int fibMMm1 = args->fib1; // (m-1)No. Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m Fibonacci
    int n = args->n;
    int x = args->target;
    uswtime(&utime0, &stime0, &wtime0);
    while (fibM > 1)
        {
            // Revisa si fibMm2 esta en una locacion valida
            int i = min(offset+fibMMm2, n-1);
            // Si x es maqyor que el valor de index fibMn2, corta el subarray

            if ((args->arr[i]) < x)
            {
                fibM = fibMMm1;
                fibMMm1 = fibMMm2;
                fibMMm2 = fibM - fibMMm1;
                offset = i;
            }

            // Si x es menor que el valor index fibMm2, corta el subarray despues de
i+1
            else if ((args->arr[i]) > x)
            {
                fibM = fibMMm2;
                fibMMm1 = fibMMm1 - fibMMm2;
                fibMMm2 = fibM - fibMMm1;
            }

            // Elemento encontrado retorna S
            else
            {
        uswtime(&utime1, &stime1, &wtime1);
                printf("Encontrado\n");
                printf("\n");
                printf("%d\n",x);
                printf("real (Tiempo total)  %.10e s\n", wtime1 - wtime0);
                printf("////////////////////////////////////////////////");
                pthread_exit((void*)&status);

            }
        }
```

```
        // compara el ultimo elemento con x
        if(fibMMm1 == 1 && (args->arr[offset+1]) == x)
            printf("Encontrado\n");

    pthread_exit((void*)&status);
}

/*
    main
    Recibe: Tama□o de problema (argv[1]), Numero a buscar (argv[2]).
    Entonces realiza la busqueda Fibonacci utilizando dos hilos.
    Y finalmente devuelve el tiempo que tarda en ejecutarse el algoritmo.
    Nota: argv[1] DEBE SER MAYOR A 0.
*/
void main(int argc, char *argv[]){
    struct argumentos *margs = NULL;
    pthread_t id1 = 0,id2 = 0;
    int n = 0, target = 0, *numeros, i = 0;
    int eid1 = -1, eid2 = -1;

    int fibMMm2 = 0; // (m-2)No. Fibonacci
    int fibMMm1 = 1; // (m-1)No. Fibonacci
    int fibM = fibMMm2 + fibMMm1; // m Fibonacci

    n = atoi(argv[1]);
    target = atoi(argv[2]);

    numeros = malloc(sizeof(int)*n);
    margs = malloc(sizeof(struct argumentos));

    margs -> n = n;
    margs -> target = target;

    LeerArchivo(numeros,n);

    margs-> arr = numeros;
    /*Generamos los numeros fibonacci, para el tama□o de problema
    el □ndice m□ximo alcanzado en la serie fibonacci ser□ el 36
    por lo tanto, se propone la creacion de dos hilos, uno busca
    en □ndice < 18 y otro en □ndice > 18*/
    i = 0;
    // fibM va a guardar el mas peque□o
    while (fibM < n)
    {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
        i++;
        if(i==18){
            //Lanzamos un hilo que busque aqui
```

```
            margs -> fib2 = fibMMm2;
            margs -> fib1 = fibMMm1;
            pthread_create(&id1,NULL,fibonacci,(void*)margs);
        }
    }
    //Alcanzo los ultimos numeros, lanzamos otro hilo para que busque por alla
    margs -> fib2 = fibMMm2;
    margs -> fib1 = fibMMm1;
    pthread_create(&id2,NULL,fibonacci,(void*)margs);
    pthread_join(id1,(void*)&eid1);
    pthread_join(id2,(void*)&eid2);

    printf("\n");
    printf("n=%i\n",n);
    printf("x=%i\n",target);
    return;
}

/* Funcion para leer y almacenar n cantidad de numeros del archivo 10millones dentro
de un arreglo (con memoria previamente asignada)
    Recibe:
        *A: Arreglo de elementos vacio con la memoria suficiente para almacenar n
cantidad de elementos
         n: cantidad de elementos que seran leidos del archivo y almacenados en el
arreglo
 */
int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}
```

EXPONENCIAL.C

```c
//Librerias incluidas
// 3CM12
// Analisis de Algoritmos
// Autores: Mora Ayala Jose, Antonio, Lopez Lopez Oscar Manual
// Jeon Jeong Paola, Lemus Ruiz Mariana ELizabeth
// Algoritmo de Busqueda por algoritmo exponencial con la implementacion de hil
os
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "./Tiempo/tiempo.c"

/* Prototipo de la funcion de lectura del archivo de los 10 millones de numeros
 */
int *LeerArchivo(int *A, int n);
int exponencialBusqueda(int * arreglo, int n, int valorABuscar, int parts);
void *exponencialH(void *args);
int *Quicksort2(int *A, int primero, int ultimo);
// Estructura auxiliar exponencial
typedef struct argumentos {
    int *arr;
    int elemento;
    int izq;
    int der;
    int *aviso;
}expo;

int main(int argc, char const *argv[])
{

    // Variables para medición de tiempos
    double utime0, stime0, wtime0, utime1, stime1, wtime1;
    int n; // n determina el tamaño del algorito dado por argumento al ejecutar
    int m; // Variables para loops
    int hilos;
    n=atoi(argv[1]);
```

```
    hilos= atoi(argv[2]);
    // Variables
    int x;               // India el valor a buscar.
    int found;       // Indica si encontró el valor.
    // Asignacion de memoria del arreglo con respecto al tamaño de problema sol
icitado
    int *A;
    int Arreglo[20] = {322486,14700764,3128036,6337399,61396,10393545,214744564
4,1295390003,450057883,187645041,1980098116,152503,5000,7500,214826,1843349527,
1321906174,2109248666,2147470852,0};
    A = (int *)malloc(n* sizeof(int));
    Quicksort2(Arreglo,0,20);
    // Lectura y asignacion de valores al arreglo en cuestion
    LeerArchivo(A, n);


    for (m = 0; m < 20; m++)
            {
                printf("VALOR DE N: %d: \n", n);
                uswtime(&utime0, &stime0, &wtime0);
                found=exponencialBusqueda(A,n,Arreglo[m],hilos);
                uswtime(&utime1, &stime1, &wtime1);

                if (found >= 0)
                {
                    printf("Busqueda Exponencial ENCONTRO: %d\n",Arreglo[m]);
                    printf("real:%.10e s\n", wtime1 - wtime0);
                    printf("\n");
                }else{
                    printf("NoE: %d\n",Arreglo[m]);
                    printf("real  %.10e s\n", wtime1 - wtime0);
                    printf("\n");
                }

                found = -1;
            }
    exit (0);
}
```

```c
/* Funcion para leer y almacenar n cantidad de numeros del archivo 10millones d
entro de un arreglo (con memoria previamente asignada)

    Recibe:
        *A: Arreglo de elementos vacio con la memoria suficiente para almacenar
 n cantidad de elementos
         n: cantidad de elementos que seran leidos del archivo y almacenados en
 el arreglo
 */
int *LeerArchivo(int *A, int n)
{
    int i;
    FILE *numeros;
    numeros = fopen("10millones.txt", "r");
    if (numeros == NULL)
    {
        puts("Error en la apertura del archivo");
    }
    for (i = 0; i < n; i++)
    {
        fscanf(numeros, "%d", &A[i]);
    }
    fclose(numeros);
}

/*
    Funcion: fibonacci
    Esta realiza una busqueda Fibonacci en el intervalo dado
    Lo que se recibe: (Struct) args que contienen los numeros Fibonacci donde i
nicia la busqueda,
    el numero a buscar y el arreglo donde buscar.
    Imprime si ha encontrado el numero
*/
void *exponencialH(void *args){
    expo *exponencial=(expo*)args;
    int izq=exponencial->izq;
    int der=exponencial->der;
    int i=izq+1;
```

```
    if (exponencial->arr[izq]==exponencial->elemento )
    {
        *exponencial->aviso=exponencial->elemento;
        pthread_exit(NULL);
    }
    while (i<=der && exponencial->arr[i] <= exponencial->elemento)
    {
        i=i*2;
    }

    if (i<=der)
    {
        der=i;
    }

    izq=i/2;

    while (izq <= der && *exponencial->aviso==-1)
    {
        //Dividiendo en mitades
        int mitad = (izq+der)/2;
        //Revisamos si el elemento de encuentra en la posicion del centro actua
l
        if (exponencial->arr[mitad] == exponencial->elemento)
        {
            *exponencial->aviso=0;
            pthread_exit(NULL);
        }

        if (exponencial->arr[mitad] < exponencial->elemento)
        {
            izq=mitad+1;
        }else{
            der=mitad-1;
        }

    }
        pthread_exit(NULL);
}
```

```
int exponencialBusqueda(int * arreglo, int n, int valorABuscar, int parts){
    pthread_t ids[parts]; // Número de hilos
    int found = -1;             // Encontró el hilo o no (0 -> false)
    // Variable auxiliar para cálculo de los rangos de los hilos
    int tamT = n / parts;
    // Variable auxiliar para divisibles no exactos
    int mod = n % parts;

    for (int i = 0; i < parts; i++)
    {


        // Creación de memoria para
        expo *d = malloc(sizeof(expo));
        d->arr = arreglo;
        d->elemento = valorABuscar;
        // Variable inicio del rango
        d->izq = i * tamT;
        // Variable fin del rango
        d->der = (d->izq + tamT) - 1;
        // Condicional para ajustar el rango
        if (i == parts - 1)
        {
            // Asignación del valor de rango
            d->der += mod;
        }
        // Comunicación entre hilos
        d->aviso = &found;

        // Lanza los hilos
        pthread_create(&ids[i], NULL, exponencialH, (void *)d);
    }

    for (int i = 0; i < parts; i++)
    {
        // Fin/terminar hilos
        pthread_join(ids[i], NULL);
    }
```

```
    return found;


}

int *Quicksort2(int *A, int primero, int ultimo)
{
    int piv, i, j, central, aux;
    central = (primero + ultimo) / 2;
    piv = A[central], i = primero, j = ultimo;
    do
    {
        while (A[i] < piv)
        {
            i++;
        }
        while (A[j] > piv)
        {
            j--;
        }
        if (i <= j)
        {
            aux = A[i];
            A[i] = A[j];
            A[j] = aux;
            i++;
            j--;
        }
    } while (i <= j);
    if (primero < j)
    {
        Quicksort2(A, primero, j);
    }
    if (primero < ultimo)
    {
        Quicksort2(A, i, ultimo);
    }
}
```

SCRIPT

A continuación se presenta uno de los 10 script implementados para cada algoritmo, el cual facilita la ejecución, cada uno recibe como argumento un valor especifico de n el numero de opción que corresponde a su algoritmo y la cantidad de hilos, en este caso 0 porque no se dispondrá de ninguno.

Exponencial.sh

```
#!/bin/bash
echo Exponencial
cd ..
gcc MainLin.c -lm -o MainLin -pthread
./MainLin 1000000 8 0 >Exponencial.txt
./MainLin 2000000 8 0 >>Exponencial.txt
./MainLin 3000000 8 0 >>Exponencial.txt
./MainLin 4000000 8 0 >>Exponencial.txt
./MainLin 5000000 8 0 >>Exponencial.txt
./MainLin 6000000 8 0 >>Exponencial.txt
./MainLin 7000000 8 0 >>Exponencial.txt
./MainLin 8000000 8 0 >>Exponencial.txt
./MainLin 9000000 8 0 >>Exponencial.txt
./MainLin 10000000 8 0 >>Exponencial.txt
```

SCRIPT HILOS

Situación contraria a uno con hilos, el tercer argumento tendrá que ser un numero mayor a 0

```
#!/bin/bash
echo Lineal Hilos
cd ..
gcc MainLin.c -lm -o MainLin -pthread
./MainLin 1000000 2 2 >LinealH.txt
./MainLin 2000000 2 2 >>LinealH.txt
./MainLin 3000000 2 2 >>LinealH.txt
./MainLin 4000000 2 2 >>LinealH.txt
./MainLin 5000000 2 2 >>LinealH.txt
./MainLin 6000000 2 2 >>LinealH.txt
./MainLin 7000000 2 2 >>LinealH.txt
./MainLin 8000000 2 2 >>LinealH.txt
./MainLin 9000000 2 2 >>LinealH.txt
./MainLin 10000000 2 2 >>LinealH.txt
```