

INSTITUTO POLITÉCNICO NACIONAL  
ESCUELA SUPERIOR DE CÓMPUTO

ANÁLISIS DE ALGORITMOS

---

## Práctica 2: Pruebas a Posteriori (Algoritmos de Búsqueda)

---

**EQUIPO:**

CompilandoConocimiento

**INTEGRANTES:**

Morales López Laura Andrea  
Ontiveros Salazar Alan Enrique  
Rosas Hernández Óscar Andrés

**PROFESOR:**

Franco Martínez Edgardo  
Adrián



# Índice

<b>1. Introducción</b>	<b>2</b>
1.1. Métodos . . . . .	2
1.2. Definición . . . . .	2
1.3. Algoritmos de Búsqueda . . . . .	2
<b>2. Planteamiento del Problema</b>	<b>3</b>
<b>3. Diseño de la Solución</b>	<b>4</b>
3.1. Búsqueda Lineal . . . . .	4
3.2. Búsqueda Binaria . . . . .	4
3.3. Búsqueda con BTS . . . . .	5
<b>4. Implementación de la Solución</b>	<b>6</b>
4.1. Búsqueda Lineal . . . . .	7
4.2. Búsqueda Lineal Paralela . . . . .	8
4.3. Búsqueda Binaria . . . . .	9
4.4. Búsqueda Binaria Paralela . . . . .	10
4.5. Búsqueda con BTS . . . . .	11
<b>5. Actividades y Pruebas</b>	<b>12</b>
5.1. Comparativas Individuales . . . . .	12
5.2. Comparativas Globales . . . . .	12
5.3. Preguntas . . . . .	12
<b>6. Errores Detectados</b>	<b>16</b>
<b>7. Posibles Mejoras</b>	<b>17</b>
<b>8. Conclusiones</b>	<b>18</b>

## 1. Introducción

Uno de los típicos problemas dentro de un curso de programación es la búsqueda. Estos algoritmos son la base de muchos otros, además de que tenemos con ellos unas ideas interesantes a usar en otro tipo de algoritmos, como búsqueda binaria y derivados, las estructuras de datos y los algoritmos concurrentes.

### 1.1. Métodos

### 1.2. Definición

### 1.3. Algoritmos de Búsqueda

## 2. Planteamiento del Problema

Con base en el ordenamiento obtenido a partir del archivo de entrada de la práctica 1 que tiene 10 millones de números diferentes; realizar la búsqueda de elementos bajo tres métodos de búsqueda, realizar el análisis teórico y experimental de las complejidades, así como encontrar las cotas de los algoritmos.

1. Búsqueda lineal o secuencial
2. Búsqueda binaria o dicotómica
3. Búsqueda en un árbol binario de búsqueda
4. Implementación de las tres búsquedas con Threads

## 3. Diseño de la Solución

### 3.1. Búsqueda Lineal

---

**Algorithm 1** LinealSearch

---

```
1: procedure LINEALSEARCH( $A, n, NumberToSearch$ )
2:   for  $i \leftarrow 0$  hasta  $n$  do
3:     if  $A[i] = NumberToSearch$  then
4:       regresa  $i$ 
5:     end if
6:   end for
7:   regresa -1
8: end procedure
```

---

### 3.2. Búsqueda Binaria

---

**Algorithm 2** BinarySearch

---

```
1: procedure BINARYSEARCH( $A, n, NumberToSearch$ )
2:    $inicio \leftarrow 0$ 
3:    $final \leftarrow n$ 
4:   while  $inicio \leq final$  do
5:      $medio \leftarrow inicio + ((final - inicio) / 2)$ 
6:     if  $A[medio] = NumberToSearch$  then
7:       regresa  $medio$ 
8:     end if
9:     if  $A[medio] > NumberToSearch$  then
10:       $final \leftarrow medio - 1$ 
11:    end if
12:     if  $A[medio] < NumberToSearch$  then
13:       $inicio \leftarrow medio + 1$ 
14:    end if
15:   end while regresa -1
16: end procedure
```

---

### 3.3. Búsqueda con BTS

---

**Algorithm 3** SearchBTS

---

```
1: procedure SEARCHBTS( $A, n, \text{NumberToSearch}$ )
2:   arbol  $\leftarrow$  CreaArbol( $A, n$ )
3:   nodo  $\leftarrow$  Raiz del árbol
4:   while nodo no sea nulo do
5:     if nodo.informacion = NumberToSearch then
6:       nodo.indice
7:     end if
8:     if NumberToSearch < nodo.informacion then
9:       nodo  $\leftarrow$  nodo.derecha
10:    end if
11:    if NumberToSearch > nodo.informacion then
12:      nodo  $\leftarrow$  nodo.izquierda
13:    end if
14:  end whileregresa -1
15: end procedure
```

---

## 4. Implementación de la Solución

## 4.1. Búsqueda Lineal

```
1  /*=====
2  LINEAL SEARCH
3  =====*/
4  /**
5   * Is just lineal search ...
6   *
7   * @param Data          A pointer to the array of int to sort
8   * @param DataSize       The size of the Data array
9   * @param NumberToSearch The number to search
10  * @return               Nothing... I'm modifying the raw data
11  */
12 int LinealSearch(int Data[], int DataSize, int NumberToSearch) {    //== LINEAL SEARCH ==
13
14     for (int i = 0; i < DataSize; ++i) {                             //For each item in Data
15         if (Data[i] == NumberToSearch)                               //If find it
16             return i;                                                //Go
17     }
18
19     return -1;                                                        //Not found
20 }
21
```



## 4.2. Búsqueda Lineal Paralela

```

1  /*=====
2  /*===== PARALELL LINEAL SEARCH =====
3  /*=====*/
4
5  /**
6   * Is just lineal search ... but I divide the search to N workers
7   *
8   * @param Data          A pointer to the array of int to sort
9   * @param DataSize       The size of the Data array
10  * @param NumberToSearch The number to search
11  * @return               Nothing...I'm modifying the raw data
12  */
13
14
15  /*=====
16  /*===== PARALELL AUXILIAR FUNCTIONS =====
17  /*=====*/
18
19  typedef struct LinealSearchDataStruct {                //Parameters to the threads
20      int Initial;                                     //Initial index to found
21      int Final;                                       //Final index to found
22      int *Data;                                       //Pointer to teh data
23      int NumberToSearch;                             //What I'm searching
24      int *FoundIt;                                   //Flag
25  } LinealSearchData;
26
27  void* LinealSearchRange(void* Parameters) {            //Thread Function
28      LinealSearchData* Data = (LinealSearchData*) Parameters; //Get the Parameters
29      int Initial = Data->Initial;                       //Unwrap the data
30      int Final = Data->Final;                           //Unwrap the data
31
32      for (int i = Initial; i<=Final && *Data->FoundIt==-1; ++i){ //For each item in Data
33          if (Data->Data[i] == Data->NumberToSearch){         //If find it
34              *Data->FoundIt = i;                             //Now we have an index :)
35              break;
36          }
37      }
38
39      return NULL;                                         //I found nothing :(
40  }
41
42
43  /*=====
44  /*===== PARALELL MAIN FUNCTIONS =====
45  /*=====*/
46  int ParalellLinealSearch
47  (int Data[], int DataSize, int ToSearch, int NumOfWorkers) { //== 'LINEAL' SEARCH ==
48
49      pthread_t* Workers =
50          (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t)); //Now get the array worker
51
52      LinealSearchData* Parameters = (LinealSearchData*)
53          malloc(NumOfWorkers*sizeof(LinealSearchData)); //Now create the parameters data
54
55      int SizeOfSearch = DataSize / NumOfWorkers;           //Get the size of the search
56
57      int FoundIt = -1;                                     //Found it flags
58      for (int i = 0; i < NumOfWorkers; ++i) {              //For each worker:
59
60          Parameters[i].Initial = i * SizeOfSearch;          //Get the initial index to search
61          Parameters[i].Final = (i == NumOfWorkers - 1)?
62              (i + 1) * SizeOfSearch - 1: DataSize - 1;      //Get the final index to search
63
64          Parameters[i].Data = Data;                         //Put the data
65          Parameters[i].NumberToSearch = ToSearch;           //Put the data
66          Parameters[i].NumberToSearch = ToSearch;           //Put the data
67          Parameters[i].FoundIt = &FoundIt;                  //Put the data
68
69          pthread_create
70              (&Workers[i], NULL, LinealSearchRange, &Parameters[i]); //Get the thread working!
71      }
72
73      for (int i = 0; i < NumOfWorkers; ++i)                //For each worker
74          pthread_join(Workers[i], NULL);                   //Now wait to the worker
75
76      return FoundIt;                                       //Return the result
77  }
78  }

```

### 4.3. Búsqueda Binaria

```

1  /*=====
2  /*                               BINARY SEARCH                               */
3  /*=====*/
4  /**
5   * Is just binary search ...
6   *
7   * @param Data           A pointer to the array of int to sort
8   * @param DataSize       The size of the Data array
9   * @param NumberToSearch The number to search
10  * @return               Nothing...I'm modifying the raw data
11  */
12
13 int BinarySearch(int Data[], int DataSize, int NumberToSearch) { //== BINARY SEARCH ==
14     int Initial = 0, Final = DataSize; //Variables that we need
15
16     while (Initial <= Final) { //While find make sense
17
18         int Middle = Initial + ((Final - Initial) / 2); //Find a new SearchPosition
19
20         if (Data[Middle] == NumberToSearch) //If all ok!
21             return Middle; //If we find it!
22         else if (Data[Middle] > NumberToSearch) //If we need to go to side
23             Final = Middle - 1; //Find the new final position
24         else //If we need to go to side
25             Initial = Middle + 1; //Find the new initial position
26     }
27
28     return -1;
29 }

```

## 4.4. Búsqueda Binaria Paralela

```

1  /*=====
2  PARALELL AUXILIAR FUNCTIONS
3  =====*/
4  typedef struct BinarySearchDataStruct {
5      int Initial;
6      int Final;
7      int *Data;
8      int NumberToSearch;
9      int *FoundIt;
10 } BinarySearchData;

11 //Parameters to the threads
12 //Initial index to found
13 //Final index to found
14 //Pointer to teh data
15 //What I'm searching
16 //Flag

17 void* BinarySearchRange(void* Parameters) {
18     BinarySearchData* Data = (BinarySearchData*) Parameters;
19     //Thread Function
20     //Get the Parameters
21     //Unwrap the data
22     //Unwrap the data
23     int Initial = Data->Initial;
24     int Final = Data->Final;
25     int Middle;
26
27     while(Initial <= Final && *Data->FoundIt==0){
28         Middle = Initial + ((Final - Initial) / 2);
29         //Find a new SearchPosition
30
31         if (Data->Data[Middle] == Data->NumberToSearch){
32             //If we find it!
33             *Data->FoundIt = Middle;
34             break;
35         }
36
37         if (Data->Data[Middle] > Data->NumberToSearch)
38             //If we need to go to side
39             //Find the new final position
40             //If we need to go to side
41             //Find the new initial position
42             Final = Middle - 1;
43         else
44             Initial = Middle + 1;
45     }
46
47     return NULL;
48     //I found nothing :(
49 }

50 /*=====
51 PARALELL MAIN FUNCTIONS
52 =====*/
53 int ParalellBinarySearch
54 (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {
55     //===== 'BINARY' SEARCH =====
56
57     pthread_t* Workers =
58         (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));
59     //Now get the array worker
60
61     BinarySearchData* Parameters = (BinarySearchData*)
62         malloc(NumOfWorkers*sizeof(BinarySearchData));
63     //Now create the parameters data
64
65     int SizeOfSearch = DataSize / NumOfWorkers;
66     //Get the size of the search
67
68     int FoundIt = -1;
69     //Found it flags
70     for (int i = 0; i < NumOfWorkers; ++i) {
71         //For each worker:
72
73         Parameters[i].Initial = i * SizeOfSearch;
74         Parameters[i].Final = (i == NumOfWorkers - 1)?
75             (i + 1) * SizeOfSearch - 1: DataSize - 1;
76         //Get the initial index to search
77         //Get the final index to search
78
79         Parameters[i].Data = Data;
80         Parameters[i].NumberToSearch = ToSearch;
81         Parameters[i].NumberToSearch = ToSearch;
82         Parameters[i].FoundIt = &FoundIt;
83         //Put the data
84         //Put the data
85         //Put the data
86         //Put the data
87
88         pthread_create
89             (&Workers[i], NULL, BinarySearchRange, &Parameters[i]); //Get the thread working!
90     }
91
92     for (int i = 0; i < NumOfWorkers; ++i)
93         pthread_join(Workers[i], NULL);
94     //For each worker
95     //Now wait to the worker
96
97     return FoundIt;
98     //Return the result
99 }

```

## 4.5. Búsqueda con BTS

```

1  /*=====
2  BST SEARCH
3  =====*/
4  /**
5   * Just search like in a BTS, note that a node save the number and
6   * the index of the number in the original array
7   *
8   * @param Tree      A pointer to a node; the root of a tree
9   * @param ToSearch   The number to search
10  * @return           The index of the number in the Original Array
11  */
12 int SearchWithBST(Node* Tree, int ToSearch) {           //== BST SEACH ==
13     Node **NewNode = &Tree;                             //Let start at root
14     while (*NewNode != NULL) {                             //While are not at a leaf
15         if ((*NewNode)->NodeItem == ToSearch)             //If we found it!
16             return (*NewNode)->Index;                     //return the index
17         NewNode = (ToSearch < (*NewNode)->NodeItem)?       //We have to move right
18             &((*NewNode)->Left): &((*NewNode)->Right);    //Move left or right
19     }
20     return -1;                                             //Not found!
21 }
22
23
24
25

```

## 5. Actividades y Pruebas

### 5.1. Comparativas Individuales

### 5.2. Comparativas Globales

### 5.3. Preguntas

- **¿Cuál de los 3 algoritmos es más fácil de implementar?**

El más sencillo (o en algoritmia conocido coloquialmente como la bruta) es la búsqueda lineal. Es la solución más sencilla con apenas 3 o 4 líneas de código.

Además de ser sencilla de implementar es en la que puedes cometer un error con menor probabilidad a la hora de definir los índices o variables de apoyo.

- **¿Cuál de los 3 algoritmos es el más difícil de implementar?**

El más difícil de implementar en la búsqueda usando un Binary Search Tree, porque nos obliga a diseñar algoritmos que son comúnmente recursivos en algo iterativo.

- **¿Cuál de los 3 algoritmos es el más difícil de implementar en su variante con hilos?**

Curiosamente la respuesta creo que tiene que ir para la versión con hilos de la búsqueda lineal.

Por un lado el trabajar con hilos siempre complica las cosas, sobretodo porque involucra problemas de sincronización y el manejo de variables que pueden ser posiblemente accedidas por varios hilos. Aquí hay una consideración importante que hay que hacer: Solo estamos buscando una variable, por lo que no me tengo que preocupar con la memoria: el primer hilo que me mande una respuesta, esa la tomaré y todos los demás hilos al notar el cambio en la variable se matan.

- **¿Cuál de los 3 algoritmos en su variante con hilos resulto ser mas rápido?**

La búsqueda binaria, es cierto, que fue más rápida la versión lineal, pero aún así la versión con hilos es endemoniadamente rápida.

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

La búsqueda binaria. Es uno de los algoritmos más famosos por tener un  $O(\log_2(n))$  aunque en ciertos casos, CON UN ÁRBOL BALANCEADO la búsqueda con árbol puede tener esa complejidad.

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

Sin duda alguna ese premio tiene que ir para la búsqueda lineal, en el peor de los casos tenemos una búsqueda de  $O(n)$ .

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

Sí, al menos en rangos generales de los algoritmos se esperaba y se ve la gran diferencia entre los algoritmos de búsqueda rápidos y la búsqueda lineal.

Por otro lado a simple vista uno puede pensar que una variante con hilos siempre será más rápida que una versión que no es concurrente, pero esto no tiene porque ser así, por ejemplo en la búsqueda binaria es más que obvio que como solo estamos haciendo serie de comparaciones entonces el añadir hilos no mejora nada y solo sobrecarga al sistema operativo con su creación.

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

No, como continuación de la respuesta anterior, si se analizaban con algo de tranquilidad los algoritmos se podía llegar a unas muy buenas estimaciones.

■ **¿Los resultados experimentales de las implementación con hilos de los algoritmos realmente tardaron  $\frac{F(t)}{\#hilos}$  de su implementación sin hilos?**

Jajaja, si hablamos de la búsqueda lineal, claro, es decir, no fue exactamente esa cantidad sobretodo por los grandes márgenes de error que tenemos a la hora de medir tiempos tan pequeños pero el punto es que sí, casi casi tenemos una mejora linealmente correspondiente con la cantidad de hilos, después de todo dividimos el trabajo entre la cantidad de hilos.

Pero en los otros dos al ser esencialmente una gran cantidad de comparaciones en lineal entonces no podemos llegar a prácticamente ninguna mejora, incluso un desempeño un poco peor.

En general, para  $n$  hilos tenemos una nueva función complejidad:

- Búsqueda Lineal:  $f_h(t) = \frac{f(t)}{\text{Número de hilos}}$
- Búsqueda Binaria:  $f_h(t) = 1f(t)$
- Búsqueda BST:  $f_h(t) = 1f(t)$

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

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

Sí, usamos una PC con las siguientes características:

- HP EliteDesk 700 G1 SFF
- 8GB de memoria RAM, DDR3 SDRAM non-ECC, 1600MHz
- Procesador Intel(R) Core(TM) i5-4590 (4° generación), CPU @ 3.30GHz (4 CPUs), ~3.3GHz. Intel vPro Technology
- 1TB de disco duro HDD, Serial ATA-600 6Gb/s, 7200 rpm

Y el software fue el siguiente:

- Sistema operativo Linux
- Distribución Elementary OS 0.4
- Compilador GCC con soporte para C11
- Python 3.6 con las bibliotecas `matplotlib` y `numpy`
- No había ninguna aplicación abierta al momento de ejecutar los algoritmos

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

Con un análisis básico, es decir, con el uso de cotas, rápidamente podríamos descartar a la búsqueda lineal SI ES QUE NUESTRO ARREGLO ESTUVIERA ORDENADO; y ya que el BST requiere otro tiempo inicial para la creación del árbol, sería fácil decantarse por la sencilla búsqueda binaria.

Pero si tenemos un arreglo que no está ordenado las cosas cambian, ahí depende mucho del tamaño del problema y de la cantidad de hilos que podemos correr eficientemente lo que decanta la balanza por BST o un búsqueda lineal.

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

- Búsqueda Lineal: Sencillo, de esos que son ejemplos básicos en clase.
- Búsqueda Binaria: Algo más compleja, sobretodo por el cálculo de los valores límites, pero la verdad es que no hay mucho más allá.
- Búsqueda BST: Igualmente, el peor caso en un árbol BALANCEADO es bastante sencillo de obtener.

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

- Hagan sus scripts jóvenes, pues aunque parezca más tedioso y crean que es más rápido anotar toda la información solicitada (que es mucha) a mano, es más eficiente por si se requiere cambiar algún algoritmo o los valores de entrada. Al menos que el script guarde los tiempos en algún archivo de texto con un formato que quieran.
- Prueben sus algoritmos con arreglos pequeños antes de buscar en los 10 millones para ver si realmente los programaron bien.
- Hagan sus programas generales, es decir, que reciban cualquier tamaño de subarreglo y el algoritmo a usar.
- Usen Linux, pues en Windows no están las bibliotecas para medir los tres tiempos.
- Cuando corran sus algoritmos, procuren cerrar todas las tareas en segundo plano para que los tiempos obtenidos sean lo más apegados a la realidad posible.



## 6. Errores Detectados

Por un lado, no hemos detectado errores al momento de correr los algoritmos que no hallamos sabido como resolver, bueno, si, solo uno.

El Tiempo.

A la hora de medir el tiempo de los algoritmos tenemos un grave problema, y es que a la hora de usar las funciones estándar proporcionadas por el profesor para medir el tiempo tenemos que sus mediciones son incorrectas en el sentido de que sabemos que la ecuación para calcular el porcentaje de uso de la CPU esta dada por:

$$CPUWall = (UserTime + SysTime) / RealTime$$

Pero al momento de intentar medir este porcentaje con búsquedas ridículamente rápida tenemos que nos dan valores sobre el 100% lo que indica que las mediciones de tiempo son imprecisas si trabajamos con intervalos pequeños de tiempo

Por otro lado no “errores” en si, pero cosas que podrían ocasionar un error en los programas son:

- Que no se sigan las indicaciones de los parámetros de consola, es decir:
  - Darle mas casos de los que tengo en el archivo
  - Darle mas tamaño para el arreglo que números
  - Darle rutas incorrectas a la hora de abrir los ficheros
- Que no se ingresen los parámetros de consola
- Que no se tenga configurada la versión de python3.6 sino la 3.5 que es la estándar en este momento
- Que no existan los directorios donde estarán las salidas y las gráficas

## 7. Posibles Mejoras

Las posibles mejoras que podemos hacer a estos algoritmos son sencillas:

- Podemos crear una interfaz de entrada de comandos mucho mas sencilla, como una que no requiera pasar la cantidad de casos a buscar, sino un simple path al archivo, así como un path al archivo de datos del arreglo
- Podemos asegurarnos de los errores que podría traer la entrada de datos, por ejemplo al asegurarnos de que estamos leyendo enteros o que el path existe
- Podemos usar una documentación acorde al lenguaje, mas exactamente usar algo como el estándar para documentar en C++11/14
- Podríamos haber hecho la organización del script Make.py mucho mas documentada separar en funciones el código

## 8. Conclusiones

- **Alan:**
- **Laura:**
- **Óscar:**

Gracias a esta practica pudimos entender mucho más a fondo los algoritmos de búsqueda, primeramente al implementarlo, al pasar de sus respectivos diseños en pseudo-código (o PSeint) a implementaciones en c11, casi todos pudieron ser modelados completamente como una función solo dependiente de sus valores de entrada pero tuvimos además de eso que armar algunas estructuras auxiliares (como BST-árbol binario de búsqueda y un pequeño stack-pila), y al momento de intentar crear funciones paralelas tuvimos que hacer tanto otra función auxiliar y una maestra, así como la declaración de diversas estructuras para enviarlas a las estructuras como parámetros.

Además usamos una característica importante de las versiones modernas de C, como son las funciones inline que nos permiten la modularidad que nos dan las funciones sin tener que perder rendimiento entre los cambios de contexto, pues inline nos permite crear funciones que el compilador puede optimizar de gran manera.

Usamos Python como lenguaje de script para automatizar todo el proceso de corrimiento y de recolección de datos.

Además es importante puntualizar que todas las conclusiones posteriores están bajo los resultados obtenidos, y si bien en cierto que buscamos crear un ambiente de pruebas lo más estable y parejo es importante recordar que estamos tratando con un conjunto, grande si, pero también con una distribución casi normal, con lo cual estamos hablando de que los algoritmos están siendo expuestos y testeados bajo condiciones que no representan como se comportarían bajo una distribución completamente aleatoria, sobretodo hablando del comportamiento en BTS pues la creación de un árbol balanceado a partir de información aleatoria es otro problema completamente diferente, uno que por cierto usa un AVL-RebBlack Tree

Pero dejando en claro estas limitaciones por el tipo de entrada que recibían los algoritmos podemos ver conclusiones muy interesantes, sobre todo hay que tener en cuenta que los ejes que usamos no están acordes, por lo que a primera vista las gráficas pueden dar la apariencia de rectas, pero no nos engañemos.

Podemos también ver por las comparativas como es que casi dictado por su nombre la búsqueda lineal, sus tiempos, se aproximan de manera casi perfecta a un recta. La gran sorpresa con los modelos teóricos es la búsqueda paralela, igual hablando de una cantidad minúscula de tiempo tenemos que es mucho mas rápido simplemente buscar el arreglo que esperar a lanzar los hilos, terminando con tiempos casi iguales.

Creo que a nadie le sorprende como es que son los tiempos de la búsqueda binaria, y porque tiene su gran fama, este algoritmo tiene un comportamiento mas bien digno de un  $\log_2(n)$  y eso es en el peor, caso, definitivamente es tu mejor opción para trabajar sobre datos ordenados y como cabía esperar su versión en hilos no mejoro nada.

Un trato casi igual se puede dar al BTS, pero hay que hacer una importante nota, y es que no estamos tomando el tiempo que toma la creación del árbol, eso si que podría hacer que

los tiempos cambiaran considerablemente, sobretodo como dije antes si no implementamos un árbol autobalanceable.

Finalmente creo que es importante escribir porque es que sus versiones con hilos no hicieron casi nada, para empezar aunque para un algoritmo de ordenamiento el tamaño de problema es enorme, para un algoritmo de búsqueda es ridículo, por lo que el tiempo de la creación de hilos puede ser una gran desventaja, también que para los casos de los 2 últimos algoritmos no ayudan en nada.

Pero en nada, por ejemplo en la búsqueda binaria, lanzar por ejemplo 4 búsquedas binarias no hace nada porque en el primer intento 3 de ellas saldrán porque el mismo algoritmo las insta a buscar mas allá de sus limites, por lo tanto mucho ayuda no nos proporcionan y solo sobrecargan al sistema.

Creo que la gran beneficiada teórica es la búsqueda lineal, pero claro, hablamos sobre la búsqueda de tamaños de problema mucho mas grandes para que se puede notar una diferencia.

Al final concluyo que me voy sin sorpresas con respecto a BinarySearch, pero me voy sorprendido por como es que fue el desempeño de los algoritmos en paralelo.

Hay mucho mas en el análisis de algoritmos que la notación  $O()$ , mucho más.