Instituto Politécnico Nacional
Escuela Superior de Cómputo

Análisis de Algoritmos

Práctica 2: Análisis temporal y notación de orden (Algoritmos de Búsqueda)

Equipo:
CompilandoConocimiento

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

Profesor:
Franco Martínez Edgardo Adrián

Índice

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. Definición

Un **algoritmo de búsqueda** se usa para obtener información guardada en alguna estructura de datos. Escoger el algoritmo apropiado frecuentemente depende de en qué estructura estemos buscando, así como de la distribución de los datos.

1.2. Algoritmos de Búsqueda

1.2.1. Búsqueda Lineal

Una **Búsqueda lineal** es aquella que busca un dato secuencialmente hasta encontrarlo dentro de una colección de datos.

Esta búsqueda es la más intuitiva, simplemente si buscamos por ejemplo una persona en una fila de personas únicamente vamos visualizando una por una las personas hasta encontrar a la que buscamos.

1.2.2. Árbol Binario de Búsqueda

Una **Búsqueda de Árbol Binario** Sea A un árbol binario R la raíz e hijos izquierdo H_I y derecho H_D. [1]

Decimos que A es un árbol binario de búsqueda ABB si y solo si se satisfacen las dos condiciones al mismo tiempo:

- H_I es vacío o R es mayor que todo elemento de H_I y H_D es un ABB.
- H_D es vacío o R es menor que todo elemento de H_D y H_D es un H_D.

Siguiendo en la misma linea de buscar a una persona, lo siguiente que podemos hacer es ordenar a las personas como un árbol, con su raíz y respectivos hijos, de ahí solo usamos el parámetro de orden para buscar a la persona y decidir si nos vamos a la izquierda o a la derecha.

Suena que el primero es mejor, pero resulta que una vez la información se encuentre dentro del árbol las búsquedas son más rápidas pues no tengo que buscar en cada uno, sino me salto algunos (reduciendo a la mitad, más específicamente).

1.2.3. Búsqueda Binaria

Una **Búsqueda Binaria** es aquella que realiza aproximaciones realizando comparaciones en una colección de datos previamente ordenada con el fin de dividir el problema a razón de la mitad hasta encontrar el dato buscado.

Esta vez tendremos en el ejemplo a las personas ordenadas con dicho parámetro, buscamos la de en medio, y comparamos con la persona que estoy buscando.

Sucesivamente vas eliminando mitades hasta que encuentres el dato.

Por ejemplo, si comenzaste con 8 el siguiente bloque será de 4, el siguiente de 2 y así sucesivamente.

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 ← 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 Lineal Paralela

Algorithm 2 LinealSearch

```
1: procedure LinealSearchParallel(A, n, NumberToSearch) Encontrado ← -1
2:     for cada Hilo do
3:         for i ← 0 hasta n ó Encontrado ≠ 0 do
4:             if A[i] = NumberToSearch then
5:                 Encontrado ← i
6:             end if
7:         end for
8:     end for
       Regresa encontrado
9: end procedure
```

3.3. Búsqueda Binaria

Algorithm 3 BinarySearch

```
1: procedure BinarySearch(A, n, NumberToSearch)
2:     inicio ← 0
3:     final ← n − 1
4:     while inicio ≤ final do
5:         medio ← (inicio + final) / 2
6:         if A[medio] = NumberToSearch then
7:             regresa medio
8:         end if
9:         if A[medio] > NumberToSearch then
10:            final ← medio -1
11:        end if
12:        if A[medio] > NumberToSearch then
13:            inicio ← medio + 1
14:        end if
15:    end while
16:    regresa -1
17: end procedure
```

3.4. Búsqueda con BST

Algorithm 4 SearchBST

```
1: procedure SearchBST(A, n, NumberToSearch)
2:     arbol ← CreaArbol(A, n)
3:     nodo ← Raiz del árbol
4:     while nodo no sea nulo do
5:         if nodo.informacion = NumberToSearch then
6:             regresa nodo.indice
7:         end if
8:         if NumberToSearch <nodo.informacion then
9:             nodo ← nodo.izquierda
10:        end if
11:        if NumberToSearch >nodo.informacion then
12:            nodo ← nodo.derecha
13:        end if
14:    end while
15:    regresa -1
16: end procedure
```

4. Implementación de la Solución

4.1. Búsqueda Lineal

```
/*====================================================================
==================      LINEAL SEARCH           ====================
====================================================================*/
/**
 * Is just lineal search ...
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */
int LinealSearch(int Data[], int DataSize, int NumberToSearch) {    //=== LINEAL SEARCH ======

    for (int i = 0; i < DataSize; ++i) {                            //For each item in Data
        if (Data[i] == NumberToSearch)                              //If find it
            return i;                                               //Go
    }

    return -1;                                                      //Not found

}
```

4.2. Búsqueda Lineal Paralela

```
/*================================================================
============      PARALELL LINEAL SEARCH          ==================
================================================================*/

/**
 * Is just lineal search ... but I divide the search to N workers
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */


/*================================================
=======    PARALELL AUXILIAR FUNCTIONS     ==========
================================================*/

typedef struct LinealSearchDataStruct {                             //Parameters to the threads
    int Initial;                                                    //Initial index to found
    int Final;                                                      //Final index to found
    int *Data;                                                      //Pointer to teh data
    int NumberToSearch;                                             //What I'm searching
    int *FoundIt;                                                   //Flag
} LinealSearchData;

void* LinealSearchRange(void* Parameters) {                         //Thread Function
    LinealSearchData* Data = (LinealSearchData*) Parameters;        //Get the Parameters
    int Initial = Data->Initial;                                    //Unwrap the data
    int Final = Data->Final;                                        //Unwrap the data

    for (int i = Initial; i<=Final && *Data->FoundIt==-1; ++i){     //For each item in Data
        if (Data->Data[i] == Data->NumberToSearch){                 //If find it
            *Data->FoundIt = i;                                     //Now we have an index :)
            break;
        }
    }

    return NULL;                                                    //I found nothing :(
}


/*================================================
=======       PARALELL MAIN FUNCTIONS      ==========
================================================*/
int ParalellLinealSearch
    (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'LINEAL' SEARCH ======

    pthread_t* Workers =
        (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));      //Now get the array worker

    LinealSearchData* Parameters = (LinealSearchData*)
                    malloc(NumOfWorkers*sizeof(LinealSearchData));  //Now create the parameters data

    int SizeOfSearch = DataSize / NumOfWorkers;                     //Get the size of the search

    int FoundIt = -1;                                               //Found it flags
    for (int i = 0; i < NumOfWorkers; ++i) {                        //For each worker:

        Parameters[i].Initial = i * SizeOfSearch;                   //Get the initial index to search
        Parameters[i].Final = (i == NumOfWorkers - 1)?
            (i + 1) * SizeOfSearch - 1: DataSize - 1;               //Get the final index to search

        Parameters[i].Data = Data;                                  //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        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

```
/*====================================================================
==================      BINARY SEARCH           =====================
====================================================================*/

/**
 * Is just binary search ...
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */

int BinarySearch(int Data[], int DataSize, int NumberToSearch) {    //=== BINARY SEARCH ======
    int Initial = 0,  Final = DataSize;                             //Variables that we need

    while (Initial <= Final) {                                      //While find make sense

        int Middle = Initial + ((Final - Initial) / 2);             //Find a new SearchPosition

        if (Data[Middle] == NumberToSearch)                         //If all ok!
            return Middle;                                          //If we find it!
        else if (Data[Middle] > NumberToSearch)                     //If we need to go to side
            Final = Middle - 1;                                     //Find the new final position
        else                                                        //If we need to go to side
            Initial = Middle + 1;                                   //Find the new initial position
    }

    return -1;
```

4.4. Búsqueda Binaria Paralela

```

/*================================================
=======    PARALELL AUXILIAR FUNCTIONS     ==========
================================================*/

typedef struct BinarySearchDataStruct {                         //Parameters to the threads
    int Initial;                                                //Initial index to found
    int Final;                                                  //Final index to found
    int *Data;                                                  //Pointer to teh data
    int NumberToSearch;                                         //What I'm searching
    int *FoundIt;                                               //Flag
} BinarySearchData;

void* BinarySearchRange(void* Parameters) {                     //Thread Function
    BinarySearchData* Data = (BinarySearchData*) Parameters;    //Get the Parameters
    int Initial = Data->Initial;                                //Unwrap the data
    int Final = Data->Final;                                    //Unwrap the data
    int Middle;

    while(Initial <= Final && *Data->FoundIt==-1){

        Middle = Initial + ((Final - Initial) / 2);             //Find a new SearchPosition

        if (Data->Data[Middle] == Data->NumberToSearch){        //If we find it!
            *Data->FoundIt = Middle;
            break;
        }

        if (Data->Data[Middle] > Data->NumberToSearch)          //If we need to go to side
            Final = Middle - 1;                                 //Find the new final position
        else                                                    //If we need to go to side
            Initial = Middle + 1;                               //Find the new initial position
    }

    return NULL;                                                //I found nothing :(
}


/*================================================
=======       PARALELL MAIN FUNCTIONS      ==========
================================================*/
int ParalellBinarySearch
    (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'BINARY' SEARCH ======

    pthread_t* Workers =
        (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));      //Now get the array worker

    BinarySearchData* Parameters = (BinarySearchData*)
        malloc(NumOfWorkers*sizeof(BinarySearchData));              //Now create the parameters data

    int SizeOfSearch = DataSize / NumOfWorkers;                     //Get the size of the search

    int FoundIt = -1;                                               //Found it flags
    for (int i = 0; i < NumOfWorkers; ++i) {                        //For each worker:

        Parameters[i].Initial = i * SizeOfSearch;                   //Get the initial index to search
        Parameters[i].Final = (i == NumOfWorkers - 1)?
            (i + 1) * SizeOfSearch - 1: DataSize - 1;               //Get the final index to search

        Parameters[i].Data = Data;                                  //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        Parameters[i].NumberToSearch = ToSearch;                    //Put the data
        Parameters[i].FoundIt = &FoundIt;                           //Put the data

        pthread_create
            (&Workers[i], NULL, BinarySearchRange, &Parameters[i]); //Get the thread working!
    }
```

```
68
69      for (int i = 0; i < NumOfWorkers; ++i)                    //For each worker
70          pthread_join(Workers[i], NULL);                       //Now wait to the worker
71
72      return FoundIt;                                           //Return the result
```

4.5. Búsqueda con BST

```
/*====================================================================
==================           BST SEARCH         =====================
====================================================================*/
/**
 * Just search like in a BTS, note that a node save the number and
 * the index of the number in the original array
 *
 * @param Tree      A pointer to a node; the root of a tree
 * @param ToSearch  The number to search
 * @return          The index of the number in the Original Array
 */
int SearchWithBST(Node* Tree, int ToSearch) {                       //=== BST SEACH ======
    Node **NewNode = &Tree;                                         //Let start at root

    while (*NewNode != NULL) {                                      //While are not at a leaf

        if ((*NewNode)->NodeItem == ToSearch)                       //If we found it!
            return (*NewNode)->Index;                               //return the index

        NewNode = (ToSearch < (*NewNode)->NodeItem)?                //We have to move right
            &((*NewNode)->Left): &((*NewNode)->Right);              //Move left or right
    }

    return -1;                                                      //Not found!
}
```

5. Tabla de Datos Experimentales

5.1. Búsqueda lineal

Tamaño de la búsqueda (n)	Tiempo (s)
100	5.400170000000001e-06
1000	3.874295000000001e-06
5000	1.1908989999999998e-05
10000	2.2184885000000004e-05
50000	0.00010020734
100000	0.00021497010499999997
200000	0.00039399861
400000	0.00081955194
600000	0.00181578397
800000	0.0014500617999999997
1000000	0.001965355875
2000000	0.004773724085
3000000	0.005916869640000001
4000000	0.008346164219999999
5000000	0.00933980942
6000000	0.01170313358
7000000	0.01420212985
8000000	0.015153932575000001
9000000	0.015440154079999999
10000000	0.018738114834999996
50000000	0.09303440340855015
100000000	0.185854079170421
500000000	0.9284114852653876
1000000000	1.856608242884096
5000000000	9.282182303833762

5.2. Búsqueda lineal con hilos

Tamaño de la búsqueda (n)	Tiempo (s)
100	0.00022928715000000008
1000	0.000106847285
5000	0.0008980274050000003
10000	0.00035208463000000004
50000	0.0008391499450000001
100000	0.001796519755
200000	0.0006333112800000001
400000	0.0011380434100000001
600000	0.0017311692299999998
800000	0.0030428648
1000000	0.005923843385
2000000	0.0051276445399999985
3000000	0.01036624908
4000000	0.013133704665
5000000	0.011837184425000001
6000000	0.013794159905
7000000	0.014990472800000001
8000000	0.017036199555
9000000	0.01907159092
10000000	0.018732559674999996
50000000	0.10094525918253451
10000000	0.20054521362894814
500000000	0.9973448492002571
1000000000	1.9933443936643935
5000000000	9.961340749377484

5.3. Búsqueda binaria

Tamaño de la búsqueda (n)	Tiempo (s)
100	1.3232400000000001e-06
1000	1.5020350000000003e-06
5000	1.3232250000000006e-06
10000	1.466285e-06
50000	1.7881400000000006e-06
100000	2.062315e-06
200000	1.764295e-06
400000	1.4424350000000003e-06
600000	1.7046900000000005e-06
800000	1.6450850000000001e-06
1000000	1.6450899999999999e-06
2000000	1.7642999999999998e-06
3000000	1.8835000000000003e-06
4000000	1.8239e-06
5000000	2.002705e-06
6000000	2.31266e-06
7000000	3.94582e-06
8000000	2.0742249999999996e-06
9000000	2.229205e-06
10000000	2.4199299999999995e-06
50000000	7.217890080254385e-06
100000000	1.2851688440977715e-05
500000000	5.792207532676436e-05
1000000000	0.00011426005893399766
5000000000	0.0005649639277918641

5.4. Búsqueda binaria con hilos

Tamaño de la búsqueda (n)	Tiempo (s)
100	0.00012367962999999997
1000	9.262561500000001e-05
5000	0.000301599505
10000	0.0009054541549999998
50000	0.00029178858000000003
100000	0.00012454987
200000	0.00012365580999999998
400000	0.000122082235
600000	0.00012693404499999997
800000	0.00011880397
1000000	0.00011785031
2000000	9.514092999999998e-05
3000000	0.00010029078
4000000	0.00011854171999999997
5000000	0.00011415481000000003
6000000	0.00010844468999999999
7000000	0.000120401375
8000000	0.00011092423499999999
9000000	7.904768e-05
10000000	0.000103008745

En este caso no fue posible interpolar para valores de n más grandes, porque obteníamos tiempos negativos.

5.5. Búsqueda en BST

Tamaño de la búsqueda (n)	Tiempo (s)
100	1.1801850000000004e-06
1000	1.81199e-06
5000	1.3947550000000004e-06
10000	3.4570800000000006e-06
50000	1.88352e-06
100000	2.7895049999999996e-06
200000	1.84774e-06
400000	1.6927700000000001e-06
600000	2.1219000000000004e-06
800000	1.80006e-06
1000000	1.75237e-06
2000000	2.4199350000000002e-06
3000000	1.9788549999999996e-06
4000000	2.61067e-06
5000000	2.18151e-06
6000000	2.038475e-06
7000000	2.062309999999999e-06
8000000	2.1576699999999997e-06
9000000	3.6597200000000004e-06
10000000	2.8252500000000006e-06
50000000	6.068298921840185e-06
100000000	1.0188839583180624e-05
500000000	4.3153164873904126e-05
1000000000	8.43585714873085e-05
5000000000	0.0004140018243945436

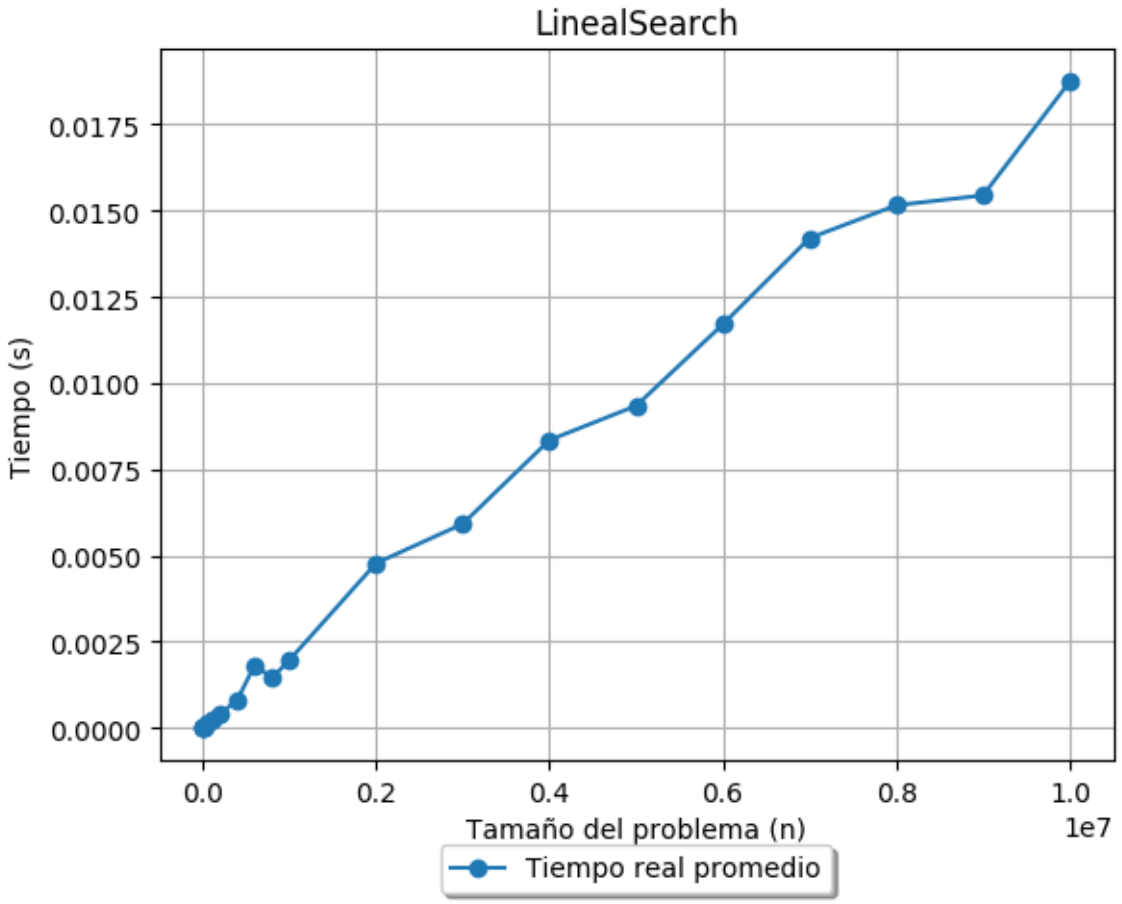
5.6. Resultados de la búsqueda en el arreglo completo

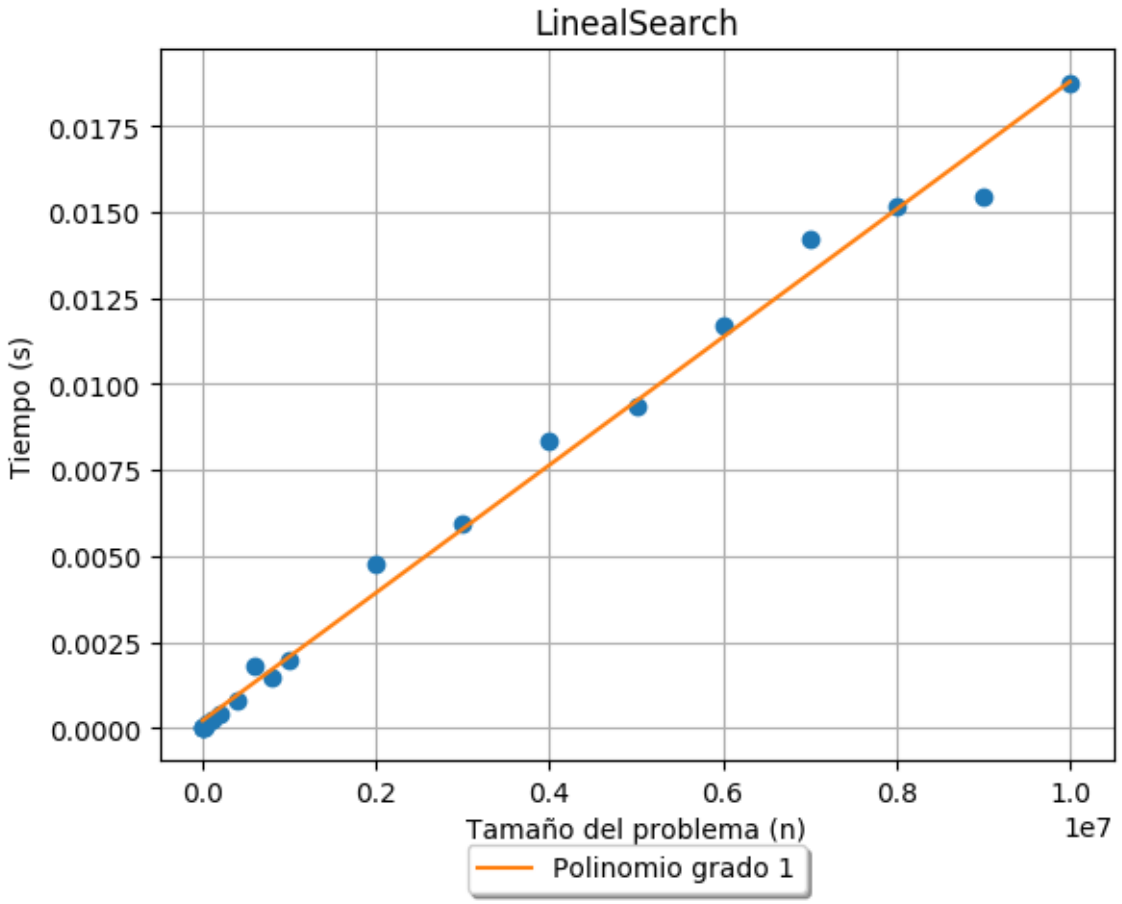
Número	Encontrado
322486	sí
14700764	no
3128036	sí
6337399	sí
61396	no
10393545	sí
2147445644	no
1295390003	sí
450057883	no
187645041	no
1980098116	no
152503	no
5000	no
1493283650	sí
214826	no
1843349527	sí
1360839354	no
2109248666	no
214747085	no
0	sí

6. Actividades y Pruebas

6.1. Comparativas Individuales

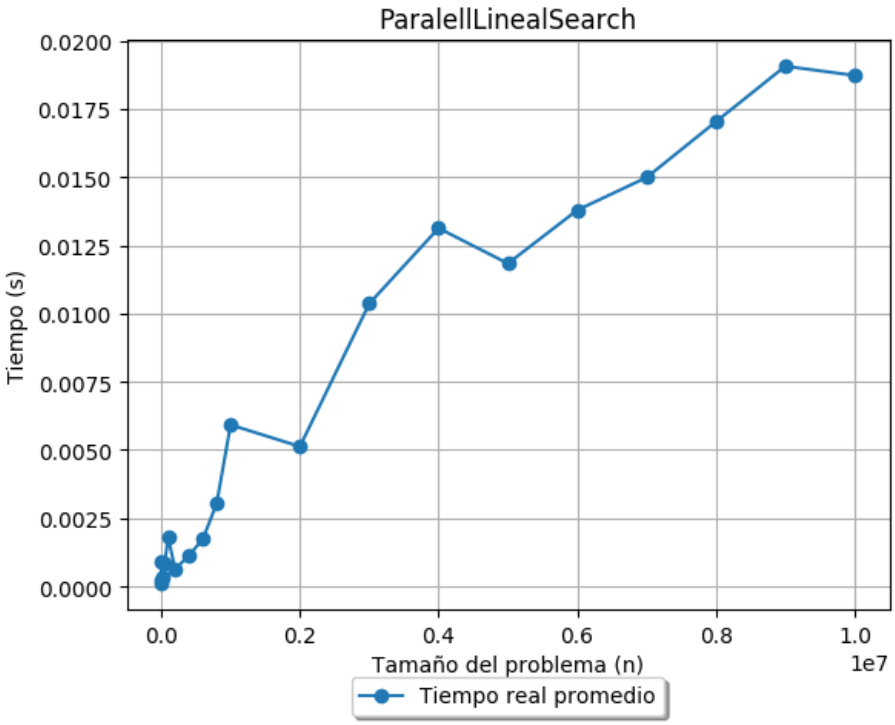
6.1.1. Búsqueda lineal

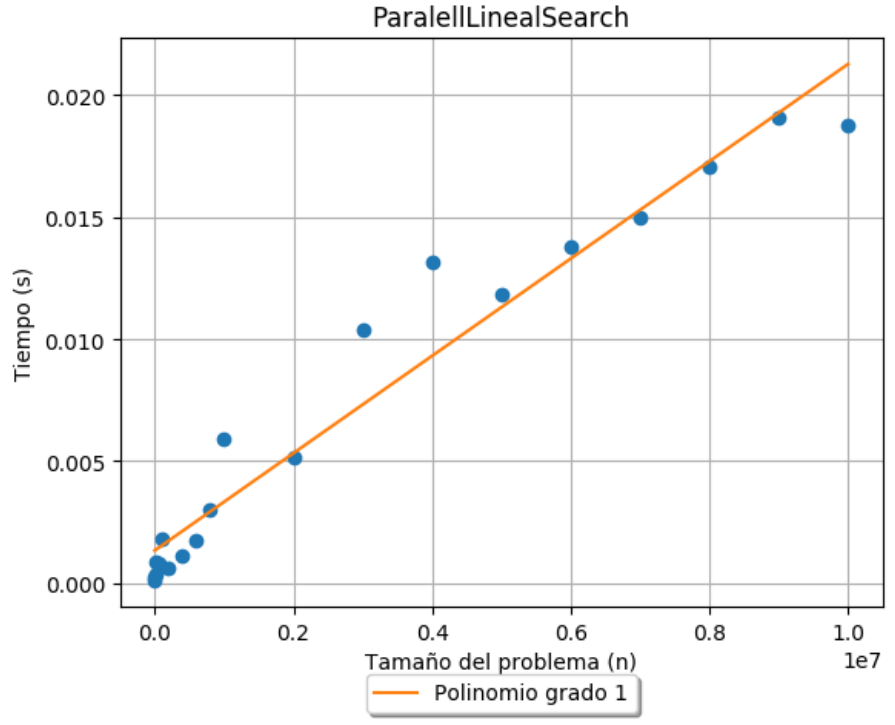




Polinomio aproximado: $1.85 \times 10^{-9}x + 2.14 \times 10^{-4} \in O(n)$.

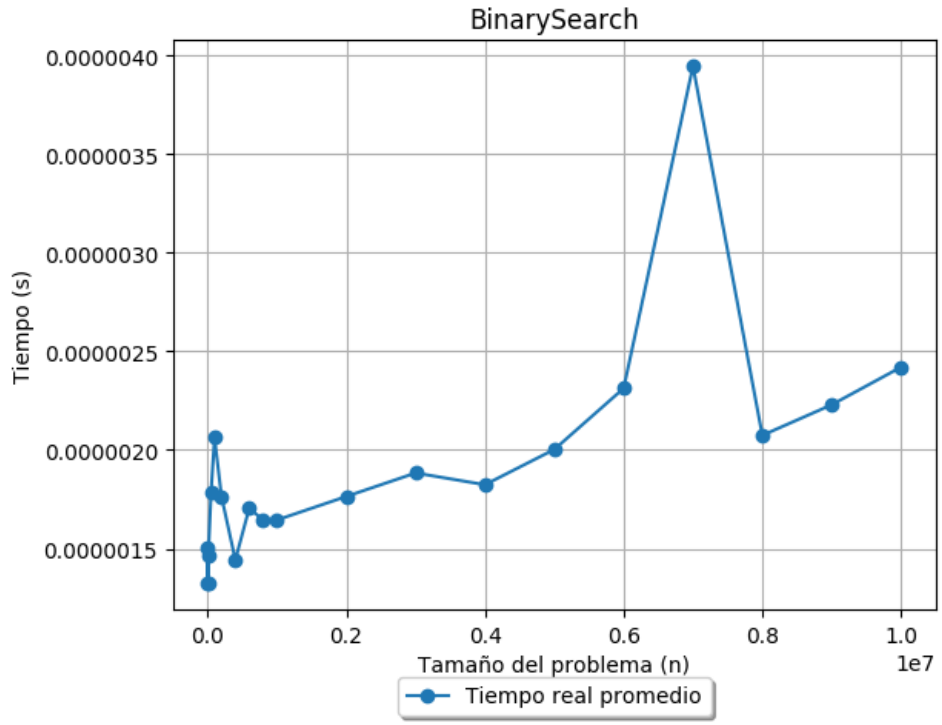
6.1.2. Búsqueda lineal con hilos

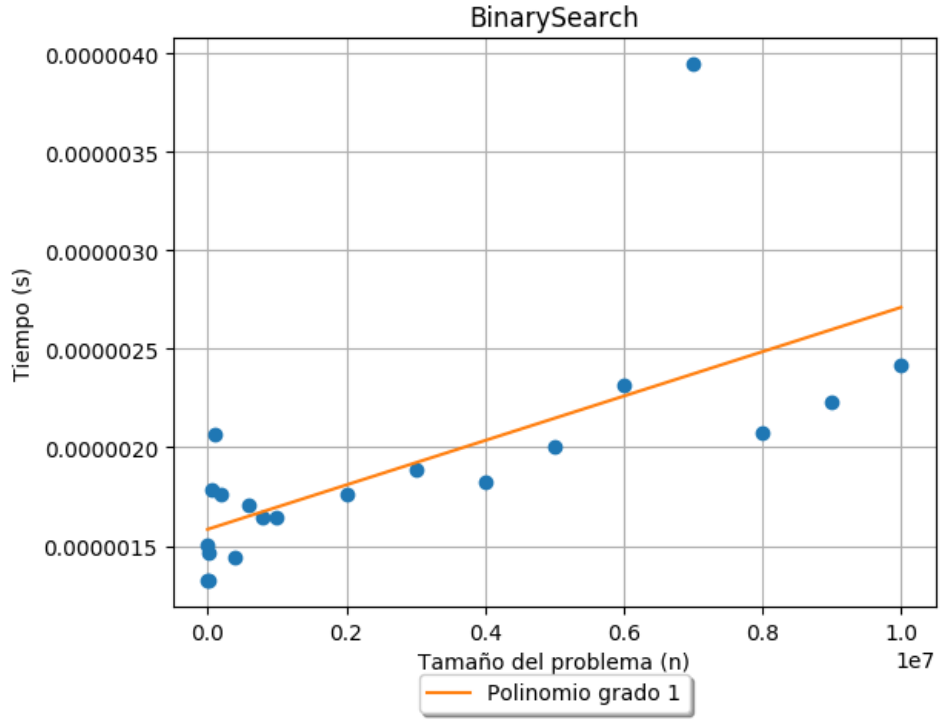




Polinomio aproximado: $1.991 \times 10^{-9}x + 1.345 \times 10^{-3} \in O(n)$.

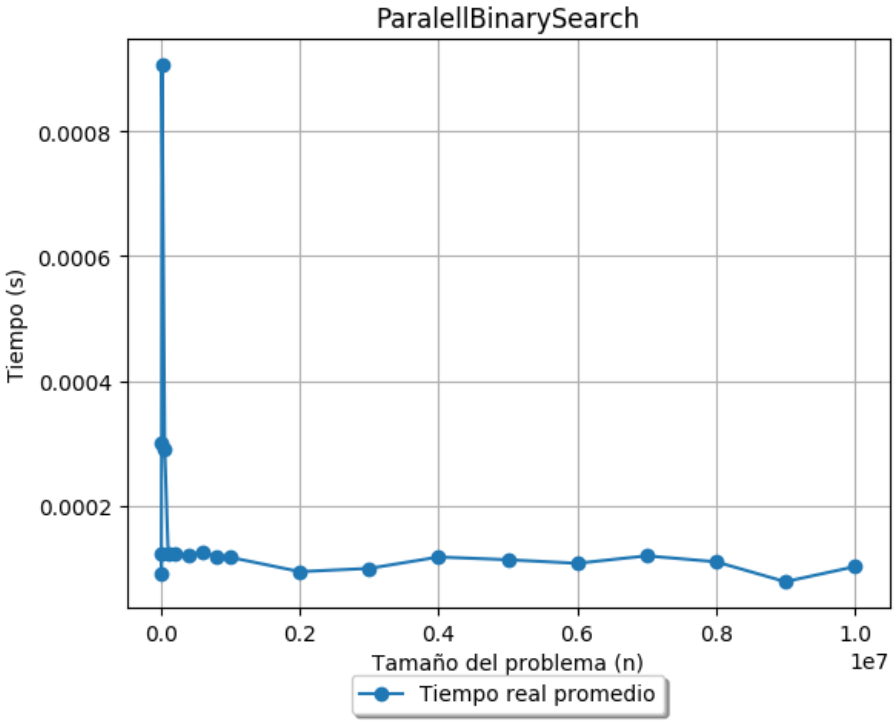
6.1.3. Búsqueda binaria

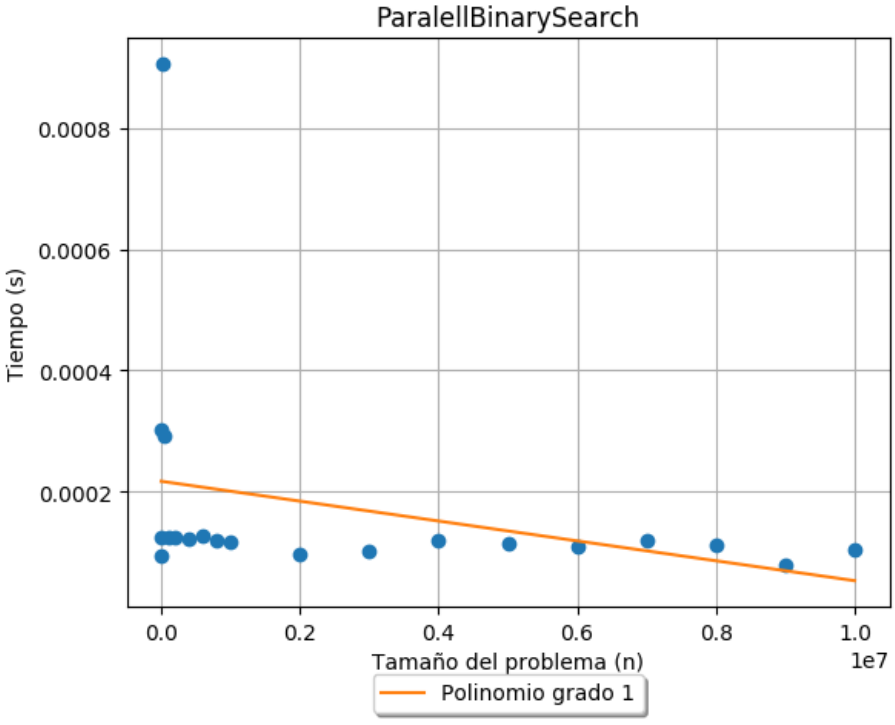




Polinomio aproximado: $1.126 \times 10^{-13}x + 1.584 \times 10^{-6} \in O(n)$.

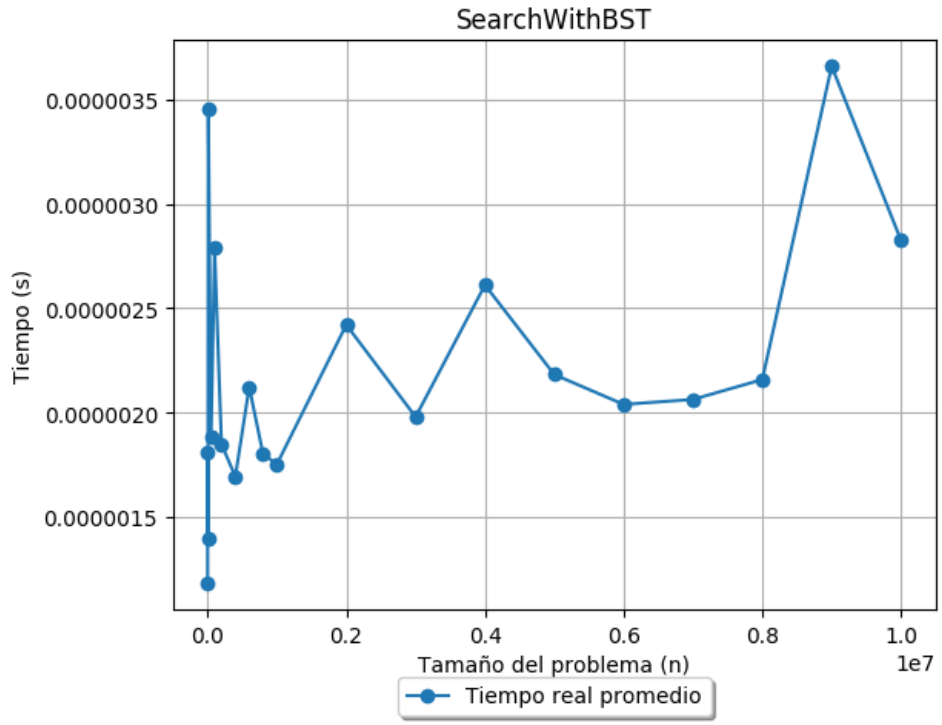
6.1.4. Búsqueda binaria con hilos

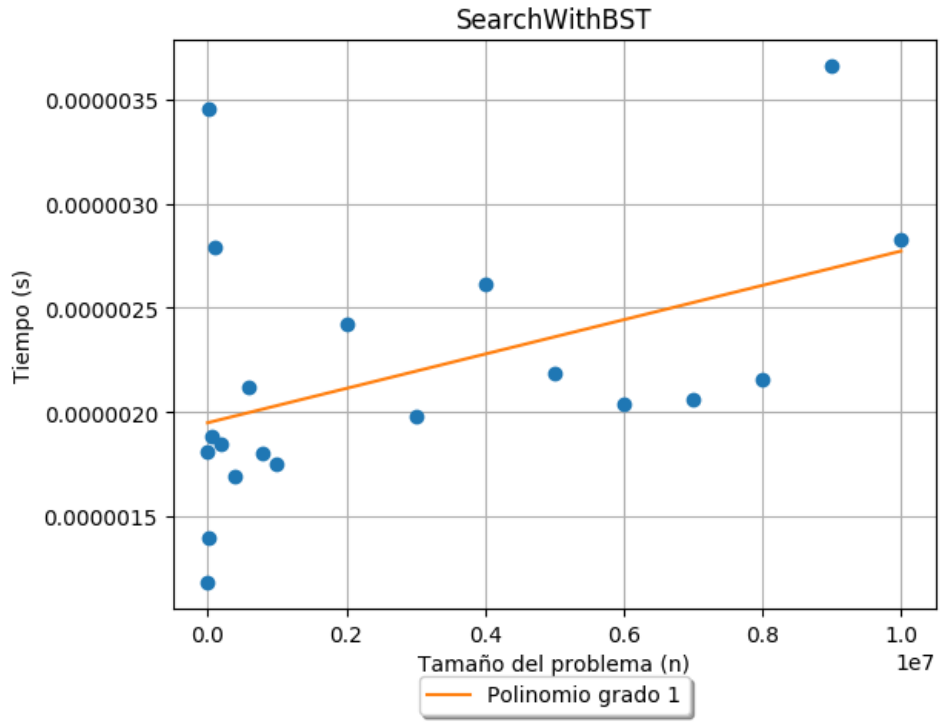




Polinomio aproximado: $-1.641 \times 10^{-11}x + 2.168 \times 10^{-4} \in O(n)$. No resultó ser una aproximación válida.

6.1.5. Búsqueda en BST

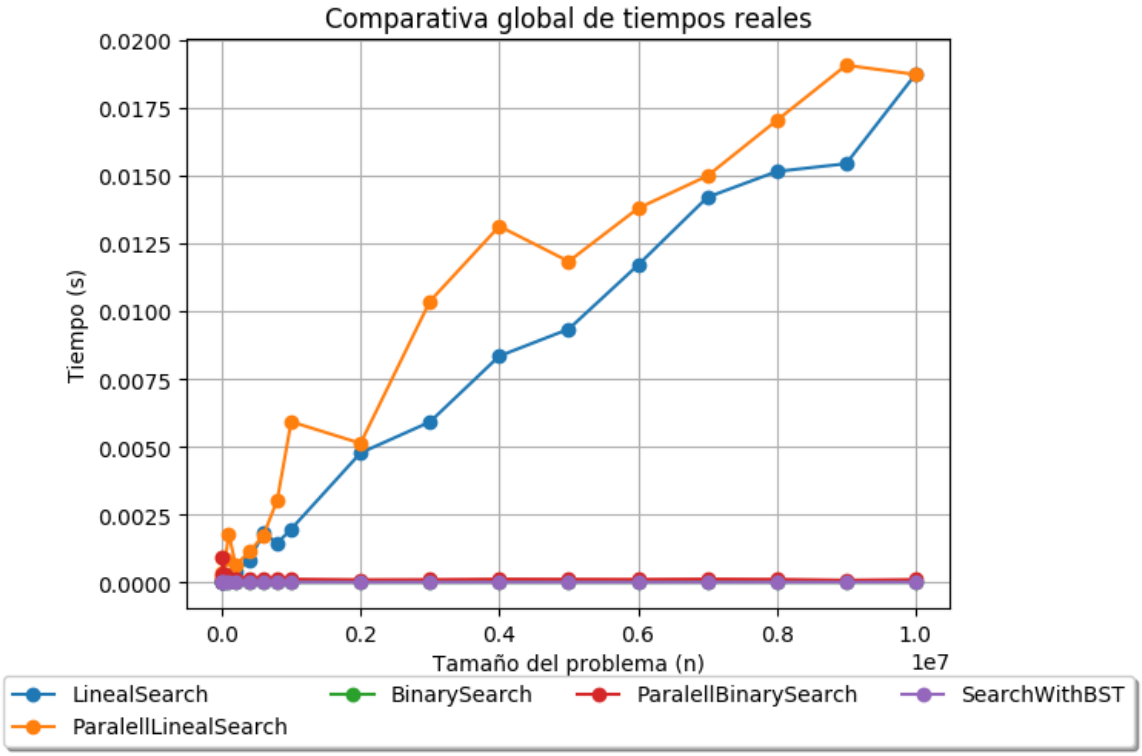


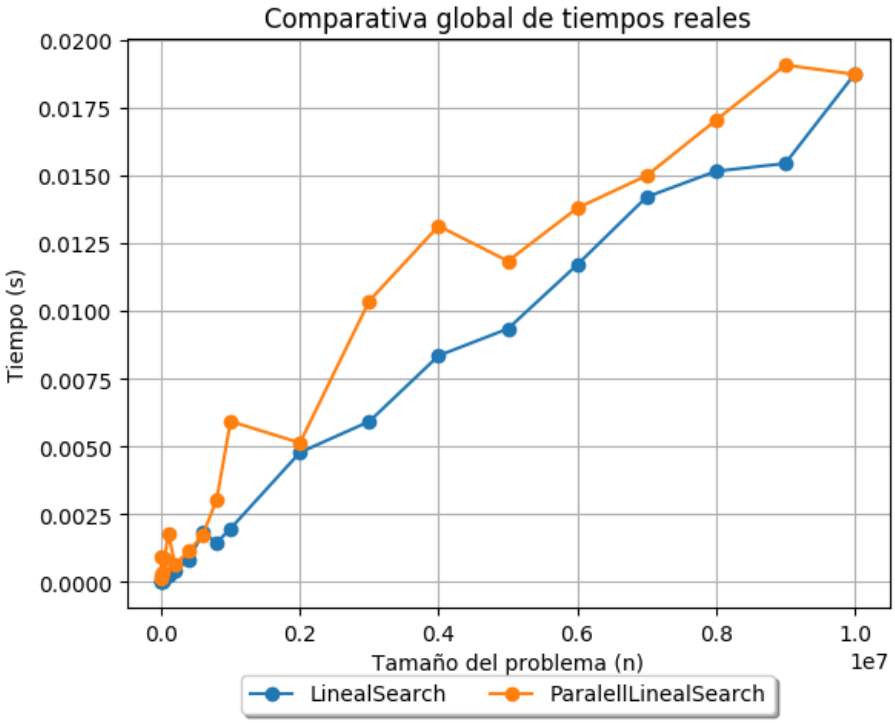


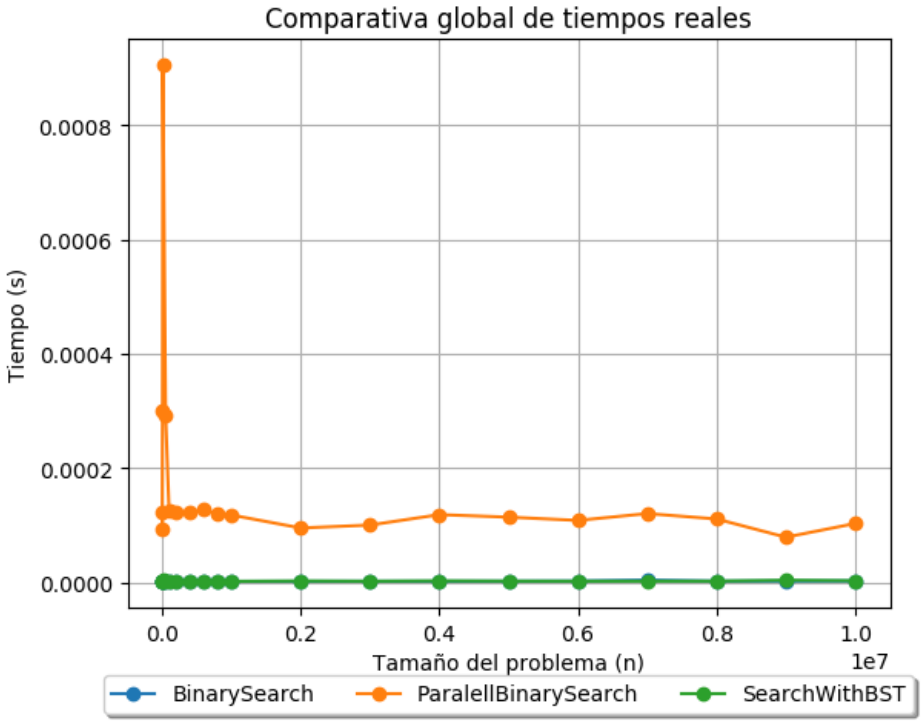
Polinomio aproximado: $8.241 \times 10^{-14}x + 1.947 \times 10^{-06}$

6.2. Comparativas Globales

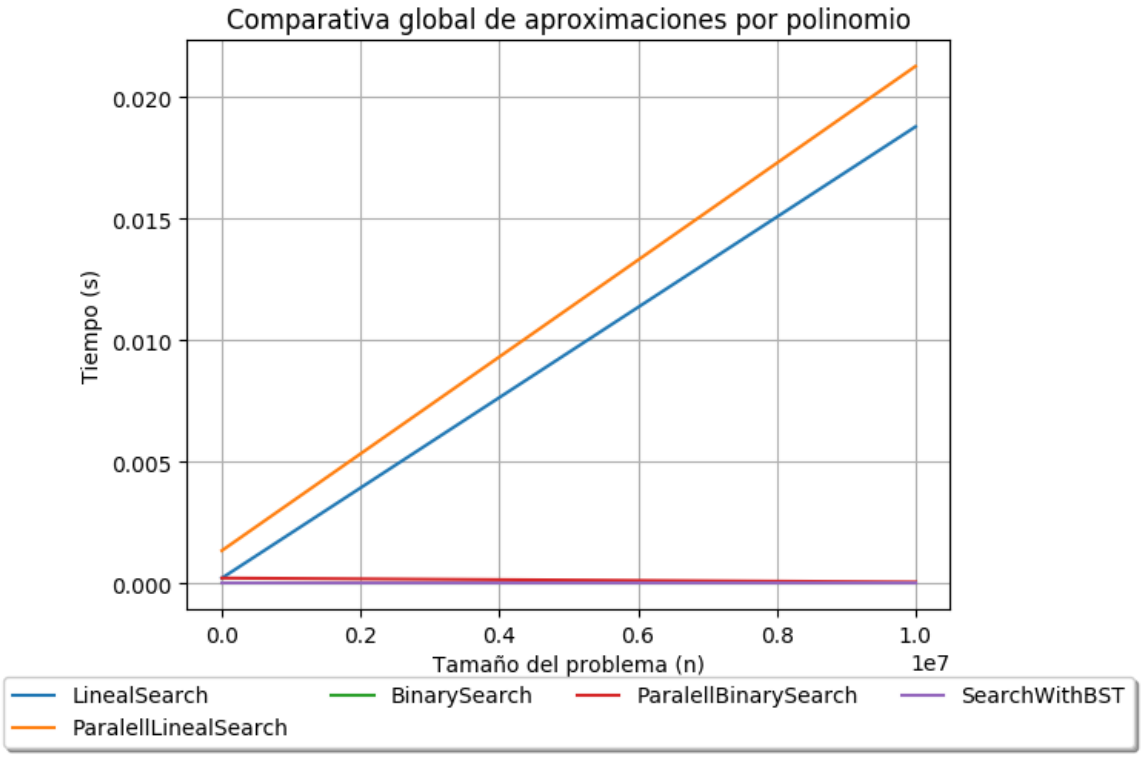
6.2.1. Por tiempo real

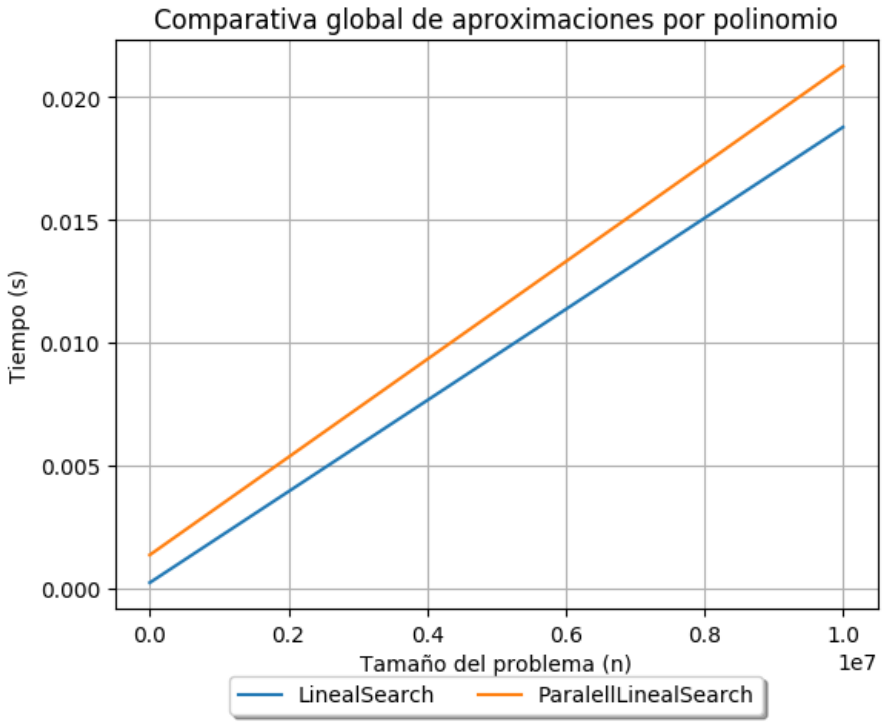


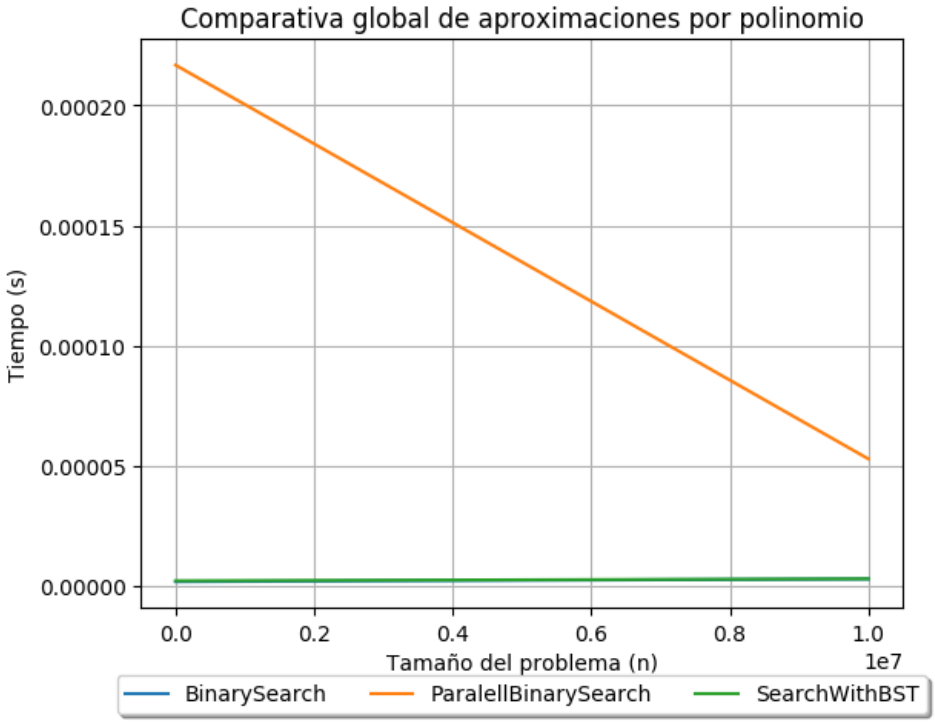



Comparativa global de tiempos reales
Tiempo (s)
0.0008
0.0006
0.0004
0.0002
0.0000
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
BinarySearch
ParalellBinarySearch
SearchWithBST

6.2.2. Por polinomio






Comparativa global de aproximaciones por polinomio
Tiempo (s)
0.00020
0.00015
0.00010
0.00005
0.00000
0.0
0.2
0.4
0.6
0.8
1.0
1e7
Tamaño del problema (n)
BinarySearch
ParalellBinarySearch
SearchWithBST

6.3. Análisis teórico

6.3.1. Búsqueda lineal

- **Mejor caso:** el elemento a buscar se encuentra en la primera posición.

  Complejidad: $\Omega(1)$.

- **Caso medio:** supongamos que cada posición (incluyendo una más allá de la última) tiene la misma probabilidad de contener al elemento buscado. Como hay $n+1$ posiciones, la probabilidad de cada una es $\dfrac{1}{n+1}$. El número de comparaciones en la i-ésima posición es i, por lo tanto la función de complejidad del caso medio es:

  $$f_m(t) = \sum_{i=1}^{n+1} \frac{i}{n+1} = \frac{(n+1)(n+2)}{2(n+1)} = \frac{n+2}{2}$$

  Complejidad: $\Theta(n)$.

- **Peor caso:** el elemento no se encuentra en el arreglo.

  Complejidad: $O(n)$.

6.3.2. Búsqueda binaria

- **Mejor caso:** el elemento se encuentra justo en la mitad del arreglo.

  Complejidad: $\Omega(1)$.

- **Peor caso:** el elemento no se encuentra en el arreglo, por lo que tuvimos que buscar en aproximadamente $\log_2 n$ segmentos (pues se van reduciendo a la mitad).

  Complejidad: $O(\log n)$.

- **Caso medio:** $\Theta(\log n)$.

6.3.3. Búsqueda en BST

- **Mejor caso:** el elemento se encuentra en la raíz del árbol.

  Complejidad: $\Omega(1)$.

- **Peor caso:** el elemento no se encuentra en el árbol. Si el árbol está balanceado, la complejidad es $O(\log n)$, pues a lo más descenderemos $\log_2 n$ niveles del árbol. Si el árbol se degeneró a una línea, la complejidad es entonces $O(n)$, pues tendremos que visitar cada elemento como si se tratara de una búsqueda lineal.

- **Caso medio:** el mismo que en la búsqueda binaria en caso de que el árbol esté balanceado: $\Theta(\log n)$. Si no lo está, la complejidad es $\Theta(n)$.

6.3.4. Tiempo de ejecución de instrucciones básicas

Con base en los tiempos obtenidos experimentalmente y el análisis teórico de los algoritmos, concluimos que a la computadora le toma aproximadamente entre 10^{-8} y 10^{-7} segundos en realizar comparaciones entre números enteros.

6.4. 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 sincronicidad 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 un 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?**

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

  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 escencialemente 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: $f_h(t) = 1f(t)$

- **¿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
  - Compliador 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.

7. 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 medicaciones 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:
  - Darme mas casos de los que tengo en el archivo
  - Darme mas tamaño para el arreglo que números
  - Darme 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

8. Posibles Mejoras

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

- Podemos crear una interfaz de entrada de comandos mucho más 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, más exactamente usar algo como el estándar para documentar en C++11/14.
- Podríamos haber hecho la organización del script Make.py mucho más documentada y separar en funciones el código.

9. Conclusiones

- **Alan:**

  Con esta práctica tuvimos la oportunidad de seguir estudiando el análisis de algoritmos, en este caso de búsqueda de un elemento en un arreglo ordenado. La diferencia con la práctica 1 es que ya tenemos más bases teóricas para decir cuál algoritmo sería el mejor sin programarlos.

  La búsqueda lineal es la más sencilla de todas, pues simplemente va buscando hasta encontrar el elemento deseado. Experimentalmente su gráfica casi se comportó como una recta, pues su complejidad teórica es $O(n)$. Al tratar de paralelizarla no notamos mucha mejora, e incluso empeoró el tiempo, pues el simple hecho de calcular los intervalos de cada hilo y lanzarlos al sistema operativo quita más tiempo que la búsqueda en sí.

  La búsqueda binaria fue la que mejor se desempeñó en la práctica, pues tomó tiempos de búsqueda en orden de los microsegundos por cada elemento, confirmando su complejidad en cualquier caso de $O(\log n)$. Es bastante eficiente y no requiere memoria adicional (en su versión iterativa). De igual forma que en la búsqueda lineal, paralelizarla no sirvió de mucho (aumentó ligeramente el tiempo), pues incluso tuvimos algunos resultados inconsistentes con los tiempos de búsqueda: la gráfica obtenida no resultó creciente.

  Finalmente la búsqueda en un BST fue la segunda en desempeñarse mejor, pues a diferencia de la binaria usa el doble de memoria para construir el árbol, aunque sigue con complejidad $O(\log n)$. Esta búsqueda, por su naturaleza, no fue posible paralelizarla de una forma adecuada como las anteriores. Sin embargo, cabe resaltar que en esencia es lo mismo que la búsqueda binaria, porque la división de intervalos en los que busca son los mismos que en una búsqueda binaria normal, asumiendo que el árbol esté balanceado, lo cual fue así pues se construyó a partir de un arreglo ordenado.

  De las aproximaciones por polinomio únicamente nos quedamos con la de la búsqueda lineal como buena aproximación, pues las complejidades de la búsqueda binaria y la del árbol son del orden logarítmico, es decir, cualquier polinomio va a ser eventualmente mucho más grande para valores de n grandes, incluso uno lineal.

- **Laura:** La diferencia entre un programa optimizado y uno que no muchas veces recae en el algoritmo de búsqueda implementado, esto nos hace ver la importancia de dichos algoritmos.

  Primeramente tenemos la búsqueda lineal que es tanto la más básica como la más intuitiva de los algoritmos.

  Después tenemos el árbol binario de búsqueda, este a diferencia de los otros necesita el uso de memoria para crear el árbol y después buscar el dato correspondiente.

  Y finalmente el ganador de la noche la búsqueda binaria, la cual me pareció personalmente interesante pues es el mismo método que se usan en los conversores ADC de aproximaciones sucesivas.

  Este tipo de conversor analógico digital tiene la curiosidad de usar una búsqueda, primero obtiene la entrada de voltaje la cual compara con un voltaje interno igual al bit más significativo del conjunto, esto implica que divide la búsqueda en 2 partes dependiendo

del resultado decide tener el bit encendido o apagado, y sigue la comparación con el siguiente bit más significativo, realiza estos pasos hasta llegar por aproximación al voltaje más cercano posible.

Con los hilos implementados pudimos ver una mejora en la búsqueda lineal ya que se divide el problema en varios procesos y permite buscar en varias partes a la vez, sin embargo esto no mejora demasiado su complejidad o tiempos.

Con los otros 2 algoritmos el intentar implémentalo con hilos no tiene sentido.

Para la búsqueda binaria, esta va dividiendo el problema y reduciéndolo con lo cual aseguramos que no se encuentra el las partes eliminadas, si deseamos usar hilos y ponemos a buscar en el lugar donde se descartó su lugar, simplemente haremos que el procesador sea más lento en su ejecución con la versión sin hilos.

Algo parecido pasa con el árbol. Si tenemos en cuenta la manera en que esta formado, estamos igual dividiendo el problema, con lo cual no tiene sentido colocarle hilos pues ralentizaría el sistema .

La mayoría de las computadoras como lo dije la vez pasada se la pasan ordenando y buscando, se volvió una operación básica de estas como lo es el sumar. El saber diferentes tipos de búsqueda te permite usarlos en problemas futuros y aunque tengamos el binario siempre de pueden usar los demás para algunos casos específicos.

De estos 3 algoritmos el único que me pareció sencillo fue el lineal, lo cual me hace recalque en que los algoritmos son tan variados como las personas que lo diseñan, así que es importante conocer algunos pues quizá algún día tengas la necesidad de usarlos en un programa.

- **Óscar:**

  Gracias a esta práctica 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 cómo 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 mejoró 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 sí que podría hacer que los tiempos cambiaran considerablemente, sobretodo como dije antes si no implementamos un árbol autobalanceable.

Finalmente creo que es importante escribir por qué 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 mucha 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 más grandes para que se pueda notar una diferencia.

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

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

Anexos

A. Código de fuente original

A.1. BinarySearch.c

```c
/*====================================================================
==================      BINARY SEARCH           =====================
====================================================================*/

/**
 * Is just binary search ...
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */

int BinarySearch(int Data[], int DataSize, int NumberToSearch) {    //=== BINARY SEARCH ======
    int Initial = 0,  Final = DataSize;                             //Variables that we need

    while (Initial <= Final) {                                      //While find make sense

        int Middle = Initial + ((Final - Initial) / 2);             //Find a new SearchPosition

        if (Data[Middle] == NumberToSearch)                         //If all ok!
            return Middle;                                          //If we find it!
        else if (Data[Middle] > NumberToSearch)                     //If we need to go to side
            Final = Middle - 1;                                     //Find the new final position
        else                                                        //If we need to go to side
            Initial = Middle + 1;                                   //Find the new initial position
    }

    return -1;
}

/*====================================================================
============     PARALELL BINARY SEARCH          ====================
====================================================================*/

/**
 * Is just binary search ... but I divide the search to N workers
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */

/*================================================
=======    PARALELL AUXILIAR FUNCTIONS     ==========
================================================*/

typedef struct BinarySearchDataStruct {                             //Parameters to the threads
    int Initial;                                                    //Initial index to found
    int Final;                                                      //Final index to found
    int *Data;                                                      //Pointer to teh data
    int NumberToSearch;                                             //What I'm searching
    int *FoundIt;                                                   //Flag
} BinarySearchData;

void* BinarySearchRange(void* Parameters) {                         //Thread Function
    BinarySearchData* Data = (BinarySearchData*) Parameters;        //Get the Parameters
```

```
59      int Initial = Data->Initial;                                    //Unwrap the data
60      int Final = Data->Final;                                        //Unwrap the data
61      int Middle;
62
63      while(Initial <= Final && *Data->FoundIt==-1){
64
65          Middle = Initial + ((Final - Initial) / 2);                 //Find a new SearchPosition
66
67          if (Data->Data[Middle] == Data->NumberToSearch){            //If we find it!
68              *Data->FoundIt = Middle;
69              break;
70          }
71
72          if (Data->Data[Middle] > Data->NumberToSearch)              //If we need to go to side
73              Final = Middle - 1;                                     //Find the new final position
74          else                                                        //If we need to go to side
75              Initial = Middle + 1;                                   //Find the new initial position
76      }
77
78      return NULL;                                                    //I found nothing :(
79  }
80
81
82  /*================================================
83  =======       PARALELL MAIN FUNCTIONS      ==========
84  ================================================*/
85  int ParalellBinarySearch
86      (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'BINARY' SEARCH ======
87
88      pthread_t* Workers =
89          (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));      //Now get the array worker
90
91      BinarySearchData* Parameters = (BinarySearchData*)
92          malloc(NumOfWorkers*sizeof(BinarySearchData));              //Now create the parameters data
93
94      int SizeOfSearch = DataSize / NumOfWorkers;                     //Get the size of the search
95
96      int FoundIt = -1;                                               //Found it flags
97      for (int i = 0; i < NumOfWorkers; ++i) {                        //For each worker:
98
99          Parameters[i].Initial = i * SizeOfSearch;                   //Get the initial index to search
100         Parameters[i].Final = (i == NumOfWorkers - 1)?
101             (i + 1) * SizeOfSearch - 1: DataSize - 1;               //Get the final index to search
102
103         Parameters[i].Data = Data;                                  //Put the data
104         Parameters[i].NumberToSearch = ToSearch;                    //Put the data
105         Parameters[i].NumberToSearch = ToSearch;                    //Put the data
106         Parameters[i].FoundIt = &FoundIt;                           //Put the data
107
108         pthread_create
109             (&Workers[i], NULL, BinarySearchRange, &Parameters[i]); //Get the thread working!
110     }
111
112     for (int i = 0; i < NumOfWorkers; ++i)                          //For each worker
113         pthread_join(Workers[i], NULL);                             //Now wait to the worker
114
115     return FoundIt;                                                 //Return the result
116
117 }
```

A.2. LinealSearch.c

```c
/*================================================================
==================    METADATA OF THE FILE      ====================
================================================================*/
/**
 * @author  Rosas Hernandez Oscar Andres
 * @author  Alan Enrique Ontiveros Salazar
 * @author  Laura Andrea Morales
 * @version 0.1
 * @team    CompilandoConocimiento
 * @date    2/04/2018
 */


/*================================================================
==================      LINEAL SEARCH           ====================
================================================================*/
/**
 * Is just lineal search ...
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */
int LinealSearch(int Data[], int DataSize, int NumberToSearch) {    //=== LINEAL SEARCH ======

    for (int i = 0; i < DataSize; ++i) {                            //For each item in Data
        if (Data[i] == NumberToSearch)                              //If find it
            return i;                                               //Go
    }

    return -1;                                                      //Not found

}


/*================================================================
============     PARALELL LINEAL SEARCH           ==================
================================================================*/

/**
 * Is just lineal search ... but I divide the search to N workers
 *
 * @param Data              A pointer to the array of int to sort
 * @param DataSize          The size of the Data array
 * @param NumberToSearch    The number to search
 * @return                  Nothing...I'm modifying the raw data
 */


/*================================================
=======    PARALELL AUXILIAR FUNCTIONS     ==========
================================================*/

typedef struct LinealSearchDataStruct {                             //Parameters to the threads
    int Initial;                                                    //Initial index to found
    int Final;                                                      //Final index to found
    int *Data;                                                      //Pointer to teh data
    int NumberToSearch;                                             //What I'm searching
    int *FoundIt;                                                   //Flag
} LinealSearchData;

void* LinealSearchRange(void* Parameters) {                         //Thread Function
    LinealSearchData* Data = (LinealSearchData*) Parameters;        //Get the Parameters
    int Initial = Data->Initial;                                    //Unwrap the data
    int Final = Data->Final;                                        //Unwrap the data

```

```
68      for (int i = Initial; i<=Final && *Data->FoundIt==-1; ++i){     //For each item in Data
69          if (Data->Data[i] == Data->NumberToSearch){                 //If find it
70              *Data->FoundIt = i;                                     //Now we have an index :)
71              break;
72          }
73      }
74
75      return NULL;                                                    //I found nothing :(
76  }
77
78
79  /*================================================
80  =======       PARALELL MAIN FUNCTIONS      ==========
81  ================================================*/
82  int ParalellLinealSearch
83      (int Data[], int DataSize, int ToSearch, int NumOfWorkers) {    //=== 'LINEAL' SEARCH ======
84
85      pthread_t* Workers =
86          (pthread_t*) malloc(NumOfWorkers * sizeof(pthread_t));      //Now get the array worker
87
88      LinealSearchData* Parameters = (LinealSearchData*)
89                      malloc(NumOfWorkers*sizeof(LinealSearchData)); //Now create the parameters data
90
91      int SizeOfSearch = DataSize / NumOfWorkers;                     //Get the size of the search
92
93      int FoundIt = -1;                                               //Found it flags
94      for (int i = 0; i < NumOfWorkers; ++i) {                        //For each worker:
95
96          Parameters[i].Initial = i * SizeOfSearch;                   //Get the initial index to search
97          Parameters[i].Final = (i == NumOfWorkers - 1)?
98              (i + 1) * SizeOfSearch - 1: DataSize - 1;               //Get the final index to search
99
100         Parameters[i].Data = Data;                                  //Put the data
101         Parameters[i].NumberToSearch = ToSearch;                    //Put the data
102         Parameters[i].NumberToSearch = ToSearch;                    //Put the data
103         Parameters[i].FoundIt = &FoundIt;                           //Put the data
104
105         pthread_create
106             (&Workers[i], NULL, LinealSearchRange, &Parameters[i]); //Get the thread working!
107     }
108
109     for (int i = 0; i < NumOfWorkers; ++i)                          //For each worker
110         pthread_join(Workers[i], NULL);                             //Now wait to the worker
111
112     return FoundIt;                                                 //Return the result
113
114 }
```

A.3. TreeAuxFunction.c

```c
/*================================================================
==================    METADATA OF THE FILE      ====================
================================================================*/
/**
 * @author  Rosas Hernandez Oscar Andres
 * @author  Alan Enrique Ontiveros Salazar
 * @author  Laura Andrea Morales
 * @version 0.1
 * @team    CompilandoConocimiento
 * @date    2/04/2018
 */

// ===================================
// ==== DECLARATION OF A BINARYTREE  ==
// ===================================
typedef struct Node {                                       //=== NODE ===
   int  NodeItem;                                           //Pointer to the real data
   struct Node *Left;                                       //Pointer to the left node
   struct Node *Right;                                      //Pointer to the left node
   int Index;                                               //Index of the original arary
} Node;                                                     //We call this struct a node


typedef Node BinaryTree;                                    //New name same functionality



/*================================================================
===========              DECLARATION            ====================
================================================================*/
Node* CreateBST(int Data[], int DataSize);
void FromSortedArray(Node** Tree, int Data[], int i, int j);
Node* CreateBSTFromSortedArray(int Data[], int DataSize);
void IterativeInsertBST(BinaryTree **Tree, int NewItem, int Index);
void IterativeCreateInOrder(Node **Tree, int *Data, int DataSize);


/*================================================================
===========        CREATE A BINARY TREE         ====================
================================================================*/
/**
 * Create a Pointer to a Binary Tree that have the data
 * that i give you. Note inline
 *
 * @param NewItem   The data that the node will save
 * @param Index     The index of the number in the array
 * @return          (Node*) A pointer to the new node
 */
extern inline Node* CreateBinaryTree(int NewItem, int Index) {      // ==== CREATE A NEW NODE ==
    Node *NewNode = (Node*) malloc(sizeof(Node));                   //Reserve memory for  node
    NewNode->NodeItem = NewItem;                                    //You will protect this
    NewNode->Left = NULL;                                           //And maybe you're a leaf
    NewNode->Right = NULL;                                          //And maybe you're a leaf
    NewNode->Index = Index;                                         //And maybe you're a leaf
    return NewNode;                                                 //Go, go NewNode :')
}



/*================================================================
===========        CREATE A BINARY SEARCH TREE        ===============
================================================================*/


/*=======================================================
```

```
68  ===========        NORMAL FUNCTION         ===============
69  ======================================================*/
70  /**
71   * Create a Pointer to a Seacrh Binary Tree
72   *
73   * @param NewItem   The data that the node will save
74   * @param Index     The index of the number in the array
75   * @return          (Node*) A pointer to the new node
76   */
77  Node* CreateBST(int Data[], int DataSize) {
78      Node* Tree = NULL;                                          //Start with empty tree
79
80      for (int i = 0; i < DataSize; i++)                          //For each element
81          IterativeInsertBST(&Tree, Data[i], i);                  //Insert in tree
82
83      return Tree;                                                //return the pointer to tree
84  }
85
86  /*======================================================
87  ===========        HELPER FUCTION          ===============
88  ======================================================*/
89  void FromSortedArray(Node** Tree, int Data[], int i, int j) {
90      if (i > j) return;                                          //Break
91
92      int Middle = i + ((j - i) >> 1);                            //Create new middle point
93
94      *Tree = CreateBinaryTree(Data[Middle], Middle);             //Create the data node
95
96      FromSortedArray(&((*Tree)->Left), Data, i, Middle - 1);     //Recursion
97      FromSortedArray(&((*Tree)->Right), Data, Middle + 1, j);    //Recursion
98  }
99
100
101 /*======================================================
102 =======      FROM A SORTED ARRAY FUNCTION         ========
103 ======================================================*/
104 /**
105  * Create a Pointer to a Search Binary Tree from a sorted array
106  *
107  * @param NewItem   The data that the node will save
108  * @param Index     The index of the number in the array
109  * @return          (Node*) A pointer to the new node
110  */
111 Node* CreateBSTFromSortedArray(int Data[], int DataSize) {
112     Node* Tree = NULL;                                          //Create a point
113     FromSortedArray(&Tree, Data, 0, DataSize - 1);              //Initial recursion
114     return Tree;                                                //Go
115 }
116
117
118 /*====================================================================
119 ========          INSERT IN A BINARY SEARCH TREE       ==============
120 ====================================================================*/
121 /**
122  * Create a Pointer to a Binary SEARCH Tree that have the data
123  * that i give you, note that this algorithm is an implementation
124  *  of a famous algorithm but this is not recursive
125  *
126  * @param Tree      A pointer to a pointer to a Tree struct
127  * @param Index     The index of the number in the array
128  * @param NewItem   The data to be inserted
129  * @return          Nothing...
130  */
131 void IterativeInsertBST(BinaryTree **Tree, int NewItem, int Index) {// ==== INSERT IN A TREE ==
132     Node **NewNode = Tree;                                      //Let start at root
133
134     while (*NewNode != NULL)                                    //While are not at a leaf
135         NewNode = (NewItem < (*NewNode)->NodeItem)?             //We have to move right
136             &((*NewNode)->Left): &((*NewNode)->Right);          //Move left or right
```

```
137
138     (*NewNode) = CreateBinaryTree(NewItem, Index);                  //Create a node at a leaf
139 }
140
141
142
143 /*================================================================
144 ========   IN ORDER TRANSVERSE OF A BINARY SEARCH TREE      =========
145 ================================================================*/
146 /**
147  * This will give you an array fill with the data of a tree
148  * if you transverse it inorder
149  *
150  * @param Tree      A pointer to a pointer to a Tree struct
151  * @param DataSize  The data size
152  * @param Data[]    This is a pointer to an ALREADY reserve Data
153                     size dimension
154  * @return          Nothing...
155  */
156 void IterativeCreateInOrder(Node **Tree, int *Data, int DataSize) { //= CREATE ARRAY FROM TREE ==
157     Node *ActualNode = *Tree;                                       //Now, use a temporal node
158
159     Node **Stack = calloc(DataSize, sizeof(Node*));                 //Create a stack like
160
161     int StackPointer = -1, i = 0;                                   //Aux variables
162
163     do {                                                            //Do the next:
164
165         while (ActualNode != NULL) {                                //While we are not a leaf
166             Stack[++StackPointer] = ActualNode;                     //Add to stack the node
167             ActualNode = ActualNode->Left;                          //Move to the fuck to left
168         }
169
170         if (StackPointer >= 0) {                                    //Ok, now while not empty
171             ActualNode = Stack[StackPointer--];                     //Start pushing the data
172             Data[i++] = ActualNode->NodeItem;                       //Now, add to the data array
173             ActualNode = ActualNode->Right;                         //And move to right
174         }
175     }
176     while (ActualNode != NULL || StackPointer >= 0);                //Now do this while we can
177
178     free(Stack);                                                    //Bye Stack :)
179 }
```

A.4. SearchInBST.c

```
1  /*====================================================================
2  ==================    METADATA OF THE FILE      =====================
3  ====================================================================*/
4  /**
5   * @author  Rosas Hernandez Oscar Andres
6   * @author  Alan Enrique Ontiveros Salazar
7   * @author  Laura Andrea Morales
8   * @version 0.1
9   * @team    CompilandoConocimiento
10  * @date    2/04/2018
11  */
12
13 #include "TreeAuxFunction.c"
14
15
16 /*====================================================================
17 ==================           BST SEARCH         =====================
18 ====================================================================*/
19 /**
20  * Just search like in a BTS, note that a node save the number and
21  * the index of the number in the original array
22  *
23  * @param Tree      A pointer to a node; the root of a tree
24  * @param ToSearch  The number to search
25  * @return          The index of the number in the Original Array
26  */
27 int SearchWithBST(Node* Tree, int ToSearch) {                    //=== BST SEACH ======
28     Node **NewNode = &Tree;                                      //Let start at root
29
30     while (*NewNode != NULL) {                                   //While are not at a leaf
31
32         if ((*NewNode)->NodeItem == ToSearch)                    //If we found it!
33             return (*NewNode)->Index;                            //return the index
34
35         NewNode = (ToSearch < (*NewNode)->NodeItem)?             //We have to move right
36             &((*NewNode)->Left): &((*NewNode)->Right);           //Move left or right
37     }
38
39     return -1;                                                   //Not found!
40 }
```

A.5. TestSearchAlgorithms.c

```c
/*================================================================
==================    METADATA OF THE FILE      ====================
================================================================*/
/**
 * @author  Rosas Hernandez Oscar Andres
 * @author  Alan Enrique Ontiveros Salazar
 * @author  Laura Andrea Morales
 * @version 0.1
 * @team    CompilandoConocimiento
 * @date    2/04/2018
 * @compile "gcc -std=c11 Time.c TestSearchAlgorithms.c -o TestSearchAlgorithms - pthread"
 * @run "./TestSearchAlgorithms DataSize NumAlgorithm NumberOfCases Cases OutputPlace < Input.txt"
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <unistd.h>

#include "Time.h"

#include "LinealSearch.c"
#include "BinarySearch.c"
#include "SearchInBST.c"

// ================================================
// ============     MAIN        =================
// ================================================
/**
 * This is the control program to check all the other
 * search algorithms.
 *
 * @param argc      Size of elements of argv
 * @param argv[0]   Name of the program
 * @param argv[1]   Size of the array to get
 * @param argv[2]   Algorithm number
 * @param argv[3]   Number of Cases
 * @param argv[4]   Number of Cases File Path
 * @param argv[5]   Output File Path
 * @return          (Int)A zero if all OK
 */

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

    // GET CONSOLE DATA
    int DataSize = atoi(argv[1]);                           //Get the DataSize
    int NumAlgorithm = atoi(argv[2]);                       //Get the NumAlgorithm
    int NumberOfCases = atoi(argv[3]);                      //Get the NumberOfCases

    FILE *CasesFile = fopen(argv[4], "r");                  //Cases
    FILE *FileName = fopen(argv[5], "w");                   //Open the file


    // GET CASES DATA
    int NumbersToSearch[NumberOfCases];                     //Get the space
    for (int i = 0; i < NumberOfCases; ++i)                 //Foreach number
        fscanf(CasesFile, "%i", &NumbersToSearch[i]);       //Read the case

    // GET THE ARRAY DATA
    int *Data =
            (int*) malloc(DataSize*sizeof(int));            //Reserve data

    for (int i = 0; i < DataSize; ++i)                      //For each number
        scanf("%i", &Data[i]);                              //Get the number
```

```
68
69      // CREATE THE TREE
70      Node* Tree = NULL;                                          //Create the Tree
71      if (NumAlgorithm == 4)                                      //Create the Tree
72          Tree = CreateBSTFromSortedArray(Data, DataSize);        //Create the BTS tree
73
74      // === NOW SEARCH THE DATA ========
75      double UserTimeStart, SysTimeStart, WallTimeStart;          //Start variables
76      double UserTimeEnd, SysTimeEnd, WallTimeEnd;                //End variables
77
78
79
80      // FOR EACH CASE
81      for (int i = 0; i < NumberOfCases; ++i) {                   //For each case
82
83          uswtime(&UserTimeStart, &SysTimeStart, &WallTimeStart); //START COUNTING
84
85          int Tmp, ToSearch = NumbersToSearch[i];                 //Find the search number
86
87          if (NumAlgorithm == 0)                                  //0 is LinealSearch
88              Tmp = LinealSearch(Data, DataSize, ToSearch);       //Lineal Search
89          if (NumAlgorithm == 1)                                  //1 is Paralell Lineal Search
90              Tmp = ParalellLinealSearch(Data, DataSize, ToSearch,4); //Paralell Lineal Search
91          if (NumAlgorithm == 2)                                  //2 is Binary Search
92              Tmp = BinarySearch(Data, DataSize, ToSearch);       //Binary Search
93          if (NumAlgorithm == 3)                                  //3 is Binary Search
94              Tmp = ParalellBinarySearch(Data, DataSize, ToSearch,4); //Search Paralell Binary Search
95          if (NumAlgorithm == 4)                                  //4 is BST Search
96              Tmp = SearchWithBST(Tree, ToSearch);                //Search with BST
97
98          uswtime(&UserTimeEnd, &SysTimeEnd, &WallTimeEnd);       //END THE COUNT
99
100         // === NOW SHOW THE DATA ========
101         double UserTime = UserTimeEnd - UserTimeStart;          //Get difference
102         double SysTime  = SysTimeEnd - SysTimeStart;            //Get difference
103         double RealTime = WallTimeEnd - WallTimeStart;          //Get difference
104
105         printf("%.10f %.10f %.10f\n", RealTime, UserTime, SysTime); //Print it!
106
107         if (Tmp == -1)                                          //If not find it
108             fprintf(FileName,"Do not find %i\n",NumbersToSearch[i]);//Print it!
109         else                                                    //If find it!
110             fprintf(FileName, "Find %i at %i\n",
111                               NumbersToSearch[i], Tmp);         //Print it!
112     }
113
114     fclose(FileName);                                           //Close the name
115
116     return 0;                                                   //Bye friends
117 }
```

A.6. Make.py

```python
import os
import subprocess
import numpy as np
import matplotlib.pyplot as plt

'''/*================================================================
==================    METADATA OF THE FILE      =====================
================================================================*/
/**
 * @author  Rosas Hernandez Oscar Andres
 * @author  Alan Enrique Ontiveros Salazar
 * @author  Laura Andrea Morales
 * @version 0.1
 * @team    CompilandoConocimiento
 * @date    2/04/2018
 * @run     "python3.6 Make.py"
 * @require numpy, matplotlib
 */
'''


'''=======================================================
==================    DATA    ============================
======================================================='''

DataSize = [
    100, 1000
]

Algorithms = [
    "LinealSearch",
    "ParalellLinealSearch",
    "BinarySearch",
    "ParalellBinarySearch",
    "SearchWithBST"
]

Cases = [
    322486, 14700764, 3128036, 6337399, 61396, 10393545,
    2147445644, 1295390003, 450057883, 187645041, 1980098116,
    152503, 5000, 1493283650, 214826, 1843349527, 1360839354,
    2109248666 , 214747085, 0
]



'''=======================================================
============      COMPILE THE PROGRAM     ================
=======================================================''' 

CasesFile = "Inputs/RealCases.txt"
CasesSize = 20

ProgramName = "TestSearchAlgorithms"
Input       = "Inputs/Input10MillionSorted.txt"
Flags       = "-std=c11 -pthread Time.c"

os.system("reset")
os.system(F"gcc {Flags} {ProgramName}.c -o {ProgramName}")



'''=======================================================
============      RUN THE PROGRAM         ================
=======================================================''' 
for NumAlgorithm in range(0, len(Algorithms)):
```

```
68
69      AlgorithmName = Algorithms[NumAlgorithm]
70
71      for n in DataSize:
72
73          OutputPlace = f"Outputs/Out-{AlgorithmName}-N={n}"
74
75          print(f"********** Running Algorithm {AlgorithmName} for n = {n} ****************")
76
77          OutputProgram = subprocess.check_output(
78              f'./{ProgramName} {n} {NumAlgorithm} {CasesSize} {CasesFile} {OutputPlace} < {Input}',
79              shell = True,
80              universal_newlines = True)
81
82          RealTimeAverage = 0
83          UserTimeAverage = 0
84          SysTimeAverage  = 0
85
86          print(f"********** Each Number ****************")
87          for Line in OutputProgram.splitlines(False):
88              Data = [float(i) for i in Line.split()]
89              RealTime = Data[0]
90              UserTime = Data[1]
91              SysTime = Data[2]
92
93              RealTimeAverage += RealTime
94              UserTimeAverage += UserTime
95              SysTimeAverage  += SysTime
96
97              CPUWall = (UserTime + SysTime) / RealTime;
98
99              print(f"Real:     {RealTime}s")
100             print(f"User:     {UserTime}s")
101             print(f"Sys:      {SysTime}s")
102             print(f"CPU/Wall: {CPUWall * 100}%")
103             print("")
104
105         print(f"********** Average Time ****************")
106         RealTimeAverage /= len(Cases)
107         UserTimeAverage /= len(Cases)
108         SysTimeAverage  /= len(Cases)
109         CPUWallAverage  = (UserTimeAverage + SysTimeAverage) / RealTimeAverage;
110
111         print(f"Real:     {RealTimeAverage}s")
112         print(f"User:     {UserTimeAverage}s")
113         print(f"Sys:      {SysTimeAverage}s")
114         print(f"CPU/Wall: {CPUWallAverage * 100}%")
115         print("")
```

B. Compilación y ejecución

- **El script completo:**

  `python3.6 Make.py` (requiere las bibliotecas `matplotlib` y `numpy`).

- **Únicamente el programa principal:**

  ```
  gcc -std=c11 Time.c TestSearchAlgorithms.c -o TestSearchAlgorithms -pthread
  ```

  ```
  ./TestSearchAlgorithms DataSize NumAlgorithm NumberOfCases Cases OutputPlace
  <Input.txt
  ```

Referencias

[1] S. Wikipedia and L. Books, *Sorting Algorithms: Sorting Algorithm, Merge Sort, Radix Sort, Insertion Sort, Heapsort, Selection Sort, Shell Sort, Bucket Sort*. General Books LLC, 2010. [Online]. Available: https://books.google.com.mx/books?id=VfU4bwAACAAJ