INSTITUTO POLITECNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO (ESCOM)

ANÁLISIS DE ALGORITMOS

NOMBRE DEL ALUMNO:

- SANTOS MÉNDEZ ULISES JESÚS

PRÁCTICA 02:

- ANÁLISIS DE BÚSQUEDA

FECHA DE ENTREGA:

- 15/04/2022

GRUPO:

- 3CM14

Planteamiento del problema

Con base en el archivo de entrada de la practica 01 que tiene 10,000,000 de números diferentes. Realizar la búsqueda de elementos bajo 5 métodos, realizar el análisis temporal a priori (análisis de casos) y a posteriori (empírico) de las complejidades.

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

Finalmente realizar la: Adaptación de los cinco métodos de búsqueda empleando hilos de manera que se busque mejorar los tiempos de búsqueda de cada método.

Entorno de pruebas

Realice esta práctica en un sistema operativo Ubuntu instalado en una partición de memoria con 500Gb de disco duro y 8Gb de memoria RAM.

En Ubuntu 3.12, copile y ejecute con la terminal de Linux, con un entorno controlado.

1 Análisis teórico de casos

Búsqueda Lineal

```
int search(int* numeros, int x, int n)
{
    int i;
    for (i = 0; i < x; i++)
        if (numeros[i] == n)
            return i;
    return -1;
}
```

Mejor caso:

Que el número a buscar sea el primero del arreglo, por lo tanto:

$$f(n) = 2$$

Caso medio:

$$f(n) = \frac{1}{2}(2) + \frac{1}{2}(n) = \frac{n}{2}$$

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = n$$

Búsqueda binaria

```
struct abb* nuevoNodo(int dato){
    struct abb* nuevo = (struct abb*)malloc(sizeof(struct abb));
    nuevo->dato = dato;
    nuevo->derecha = nuevo->izquierda = NULL;
    return nuevo;
}
```

```
struct abb* insertar(struct abb* nodo, int dato){

    struct abb *aux = nodo; //variable auxiliar para ir recorriendo el abb
    int i=1; //variable pde control para el while

    if(aux == NULL)
        return nuevoNodo(dato); //Creamos el nodo

    while(i){
```

```
        if(dato < aux->dato)
            if(aux->izquierda == NULL){
                aux->izquierda = nuevoNodo(dato);
                i = 0; //para terminar el while
            } else aux = aux->izquierda;
        else if (dato > aux->dato){
            if(aux->derecha == NULL){
                    aux->derecha = nuevoNodo(dato);
                    i = 0; //para terminar el while
            } else aux = aux->derecha;
        }
    }
    return nodo;
}
```

```
void buscarABB(struct abb* root, int dato){
    while(root != NULL){

        if(root->dato < dato)
            root = root->derecha;
        else if(root->dato > dato)
            root = root->izquierda;
            else{
                break;
            }
    }

}
```

```
void inorder(struct abb* root){
    if(root != NULL){
        inorder(root->izquierda);
        printf("%d \n", root->dato);
        inorder(root->derecha);
    }
}
```

Mejor caso:

Que el número a buscar sea el primero del arreglo, por lo tanto:

$$f(n) = 3n$$

Caso medio:

$$f(n) = \frac{1}{3}3n + \frac{1}{3}4n + \frac{1}{3}3n = \frac{n}{3}$$

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = 4n$$

Búsqueda Binaria

```
int binarySearch(int* numeros, int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (numeros[m] == x)
            return m;

        // If x greater, ignore left half
        if (numeros[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```

Mejor caso:

Que el número a buscar sea el primero del arreglo, por lo tanto:

$$f(n) = 2$$

Caso medio:

$$f(n) = \frac{1}{3}(2) + \frac{1}{3}(3n) + \frac{1}{3}(2n) = \frac{2}{3} + \frac{5n}{3}$$

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = 3n$$

Búsqueda Exponencial

```
int exponentialSearch(int* numeros, int x, int n)
{
    // If x is present at first location itself
    if (numeros[0] == n)
        return 0;

    // Find range for binary search by
    // repeated doubling
    int i = 1;
    while (i < x && numeros[i] <= n)
        i = i*2;

    //  Call binary search for the found range.
    return binarySearch(numeros, i/2, min(i, x-1), n);
}
```

```
int binarySearch(int* numeros, int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (numeros[m] == x)
            return m;

        // If x greater, ignore left half
        if (numeros[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```

Mejor caso:

Que el número a buscar sea el primero del arreglo, por lo tanto:

$$f(n) = 3(2) = 6$$

Caso medio:

$$f(n) = \frac{1}{2}1 + \frac{1}{2}(n)\big((2) + (3n) + (2n)\big) = \frac{1}{2} + n + 2n^2$$

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = 1 + n(3n) = 1 + 3n^2$$

Búsqueda Fibonacci

```
int fibMonaccianSearch(int* numeros, int x, int n)
{
    /* Initialize fibonacci numbers */
    int fibMMm2 = 0; // (m-2)'th Fibonacci No.
    int fibMMm1 = 1; // (m-1)'th Fibonacci No.
    int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

    /* fibM is going to store the smallest Fibonacci
       Number greater than or equal to n */
    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    // Marks the eliminated range from front
    int offset = -1;

    /* while there are elements to be inspected. Note that
       we compare arr[fibMm2] with x. When fibM becomes 1,
       fibMm2 becomes 0 */
    while (fibM > 1) {
        // Check if fibMm2 is a valid location
        int i = min(offset + fibMMm2, n - 1);

        /* If x is greater than the value at index fibMm2,
           cut the subarray array from offset to i */
        if (numeros[i] < x) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        }
```

```
        /* If x is greater than the value at index fibMm2,
           cut the subarray after i+1  */
        else if (numeros[i] > x) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        /* element found. return index */
        else
            return i;
    }

    /* comparing the last element with x */
    if (fibMMm1 && numeros[offset + 1] == x)
        return offset + 1;

    /*element not found. return -1 */
    return -1;
}
int min(int x, int y)
{
    return (x <= y) ? x : y;
}
```

Mejor caso:

Que el número a buscar sea el primero del arreglo, por lo tanto:

$$f(n) = 2n + 2$$

Caso medio:

$$f(n) = \frac{1}{3}(4n+2) + \frac{1}{3}(2n+2) + \frac{1}{3}(3n+2) = \frac{4n}{3} + \frac{2}{3} + \frac{2n}{3} + \frac{2}{3} + n + \frac{2}{3} = 3n + 2$$

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = 3n + 2$$

Zona de códigos

Búsqueda Lineal

```
int search(int* numeros, int x, int n)
{
    int i;
    for (i = 0; i < x; i++)
        if (numeros[i] == n)
            return i;
    return -1;
}
```

Búsqueda binaria árbol

```
struct abb* nuevoNodo(int dato){
    struct abb* nuevo = (struct abb*)malloc(sizeof(struct abb));
    nuevo->dato = dato;
    nuevo->derecha = nuevo->izquierda = NULL;
    return nuevo;
}
struct abb* insertar(struct abb* nodo, int dato){

    struct abb *aux = nodo; //variable auxiliar para ir recorriendo el abb
    int i=1; //variable pde control para el while

    if(aux == NULL)
        return nuevoNodo(dato); //Creamos el nodo

    while(i){
        if(dato < aux->dato)
            if(aux->izquierda == NULL){
                aux->izquierda = nuevoNodo(dato);
                i = 0; //para terminar el while
            } else aux = aux->izquierda;
        else if (dato > aux->dato){
            if(aux->derecha == NULL){
                    aux->derecha = nuevoNodo(dato);
                    i = 0; //para terminar el while
            } else aux = aux->derecha;
        }
    }
    return nodo;
}
void buscarABB(struct abb* root, int dato){
```

```
    while(root != NULL){

        if(root->dato < dato)
            root = root->derecha;
        else if(root->dato > dato)
            root = root->izquierda;
            else{
                break;
            }
    }

}
void inorder(struct abb* root){
    if(root != NULL){
        inorder(root->izquierda);
        printf("%d \n", root->dato);
        inorder(root->derecha);
    }
}
```

Búsqueda binaria

```
int binarySearch(int* numeros, int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (numeros[m] == x)
            return m;

        // If x greater, ignore left half
        if (numeros[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```

Búsqueda exponencial

```
int exponentialSearch(int* numeros, int x, int n)
{
    // If x is present at first location itself
    if (numeros[0] == n)
        return 0;

    // Find range for binary search by
    // repeated doubling
    int i = 1;
    while (i < x && numeros[i] <= n)
        i = i*2;

    //  Call binary search for the found range.
    return binarySearch(numeros, i/2, min(i, x-1), n);
}
```

```
int binarySearch(int* numeros, int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (numeros[m] == x)
            return m;

        // If x greater, ignore left half
        if (numeros[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```

Búsqueda Fibonacci

```
int fibMonaccianSearch(int* numeros, int x, int n)
{
    /* Initialize fibonacci numbers */
    int fibMMm2 = 0; // (m-2)'th Fibonacci No.
    int fibMMm1 = 1; // (m-1)'th Fibonacci No.
    int fibM = fibMMm2 + fibMMm1; // m'th Fibonacci

    /* fibM is going to store the smallest Fibonacci
       Number greater than or equal to n */
    while (fibM < n) {
        fibMMm2 = fibMMm1;
        fibMMm1 = fibM;
        fibM = fibMMm2 + fibMMm1;
    }

    // Marks the eliminated range from front
    int offset = -1;

    /* while there are elements to be inspected. Note that
       we compare arr[fibMm2] with x. When fibM becomes 1,
       fibMm2 becomes 0 */
    while (fibM > 1) {
        // Check if fibMm2 is a valid location
        int i = min(offset + fibMMm2, n - 1);

        /* If x is greater than the value at index fibMm2,
           cut the subarray array from offset to i */
        if (numeros[i] < x) {
            fibM = fibMMm1;
            fibMMm1 = fibMMm2;
            fibMMm2 = fibM - fibMMm1;
            offset = i;
        }

        /* If x is greater than the value at index fibMm2,
           cut the subarray after i+1  */
        else if (numeros[i] > x) {
            fibM = fibMMm2;
            fibMMm1 = fibMMm1 - fibMMm2;
            fibMMm2 = fibM - fibMMm1;
        }

        /* element found. return index */
        else
```

```
            return i;
    }

    /* comparing the last element with x */
    if (fibMMm1 && numeros[offset + 1] == x)
        return offset + 1;

    /*element not found. return -1 */
    return -1;
}
int min(int x, int y)
{
    return (x <= y) ? x : y;
}
```

Registro de tiempo

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	1,000,000	1.77E-03 s
Búsqueda binaria		1.97E-05 s
Búsqueda en ABB		1.82E+01 s
Búsqueda Exponencial		1.64E-05 s
Búsqueda Fibonacci		1.76E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	2,000,000	1.88E-03 s
Búsqueda binaria		2.16E-05 s
Búsqueda en ABB		1.93E+01 s
Búsqueda Exponencial		1.32E-05 s
Búsqueda Fibonacci		2.20E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	3,000,000	1.22E-02 s
Búsqueda binaria		1.83E-05 s
Búsqueda en ABB		2.03E+01 s
Búsqueda Exponencial		2.22E-05 s
Búsqueda Fibonacci		1.67E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	4,000,000	1.35E-02 s
Búsqueda binaria		3.86E-05 s
Búsqueda en ABB		2.14E+01 s
Búsqueda Exponencial		1.64E-04 s
Búsqueda Fibonacci		2.07E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	5,000,000	3.26E-02 s
Búsqueda binaria		2.46E-05 s
Búsqueda en ABB		3.45E+01 s
Búsqueda Exponencial		2.17E-05 s
Búsqueda Fibonacci		2.35E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	6,000,000	3.71E-02 s
Búsqueda binaria		1.89E-05 s
Búsqueda en ABB		3.50E+01 s
Búsqueda Exponencial		1.89E-05 s
Búsqueda Fibonacci		2.09E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	7,000,000	3.11E-02 s
Búsqueda binaria		2.16E-05 s
Búsqueda en ABB		3.45E+01 s
Búsqueda Exponencial		5.23E-04 s
Búsqueda Fibonacci		1.83E-05 s

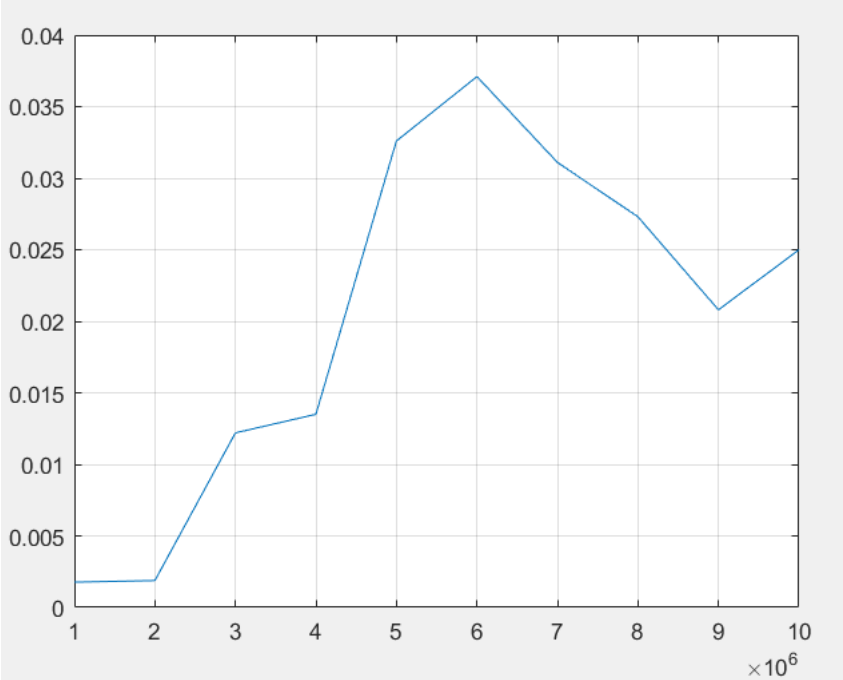
Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	8,000,000	2.73E-02 s
Búsqueda binaria		2.16E-05 s
Búsqueda en ABB		3.12E+01 s
Búsqueda Exponencial		1.63E-05 s
Búsqueda Fibonacci		2.16E-05 s

Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	9,000,000	2.08E-02 s
Búsqueda binaria		2.62E-05 s
Búsqueda en ABB		2.95E+01 s
Búsqueda Exponencial		2.42E-05 s
Búsqueda Fibonacci		2.00E-05 s

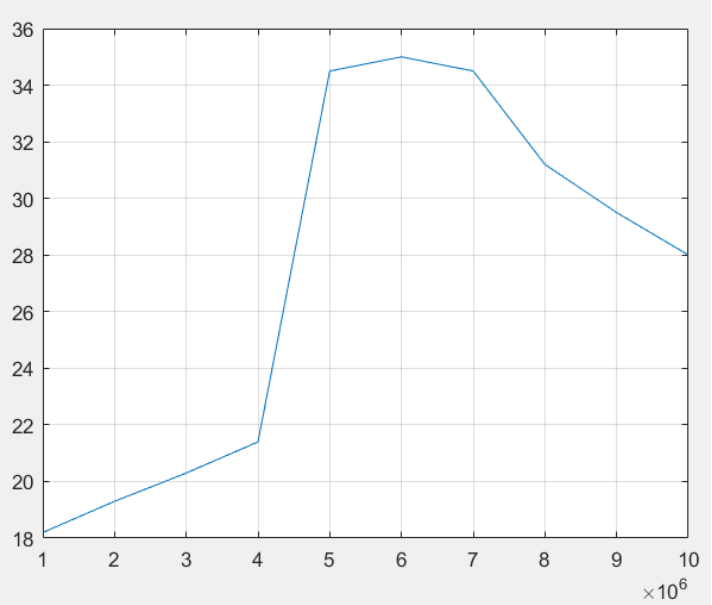
Algoritmo de Búsqueda		
Algoritmo	**Tamaño de N**	**Tiempo promedio de todas las búsquedas**
Búsqueda lineal	10,000,000	2.50E-02 s
Búsqueda binaria		2.46E-05 s
Búsqueda en ABB		2.80E+01 s
Búsqueda Exponencial		2.11E-05 s
Búsqueda Fibonacci		2.65E-05 s

Gráficas de comportamiento temporal

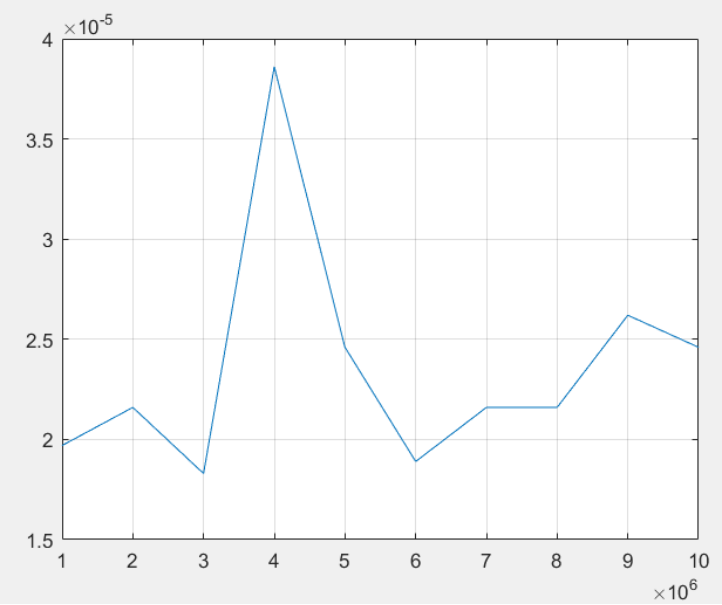
Búsqueda Lineal



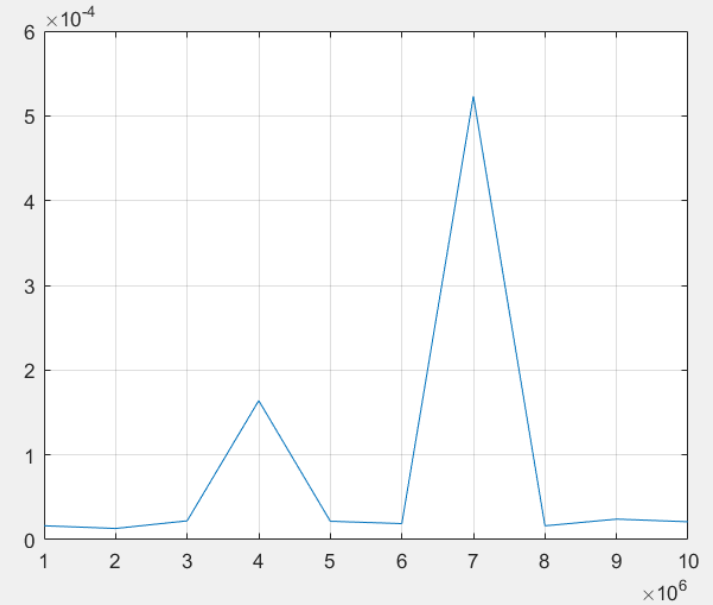
Búsqueda Binaria árbol



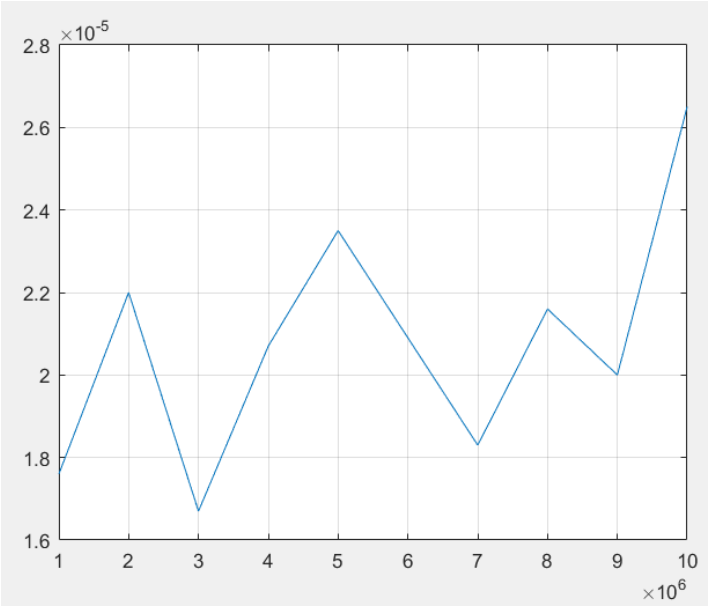
Búsqueda binaria



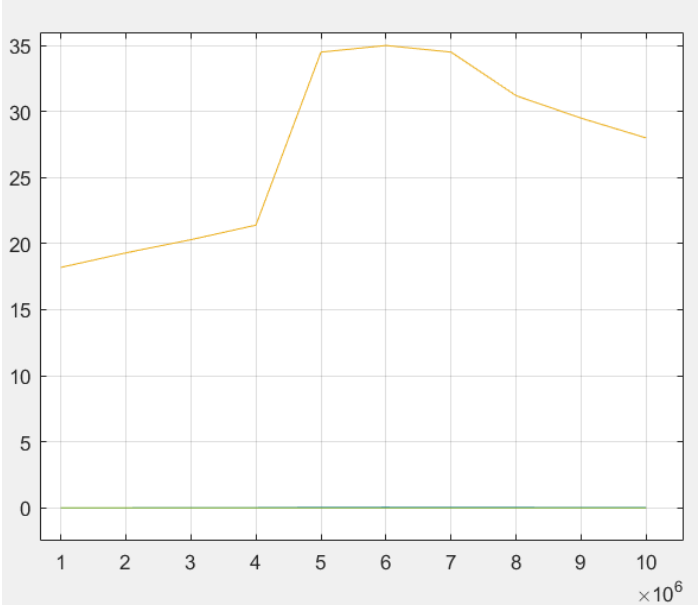
Búsqueda exponencial



Búsqueda Fibonacci

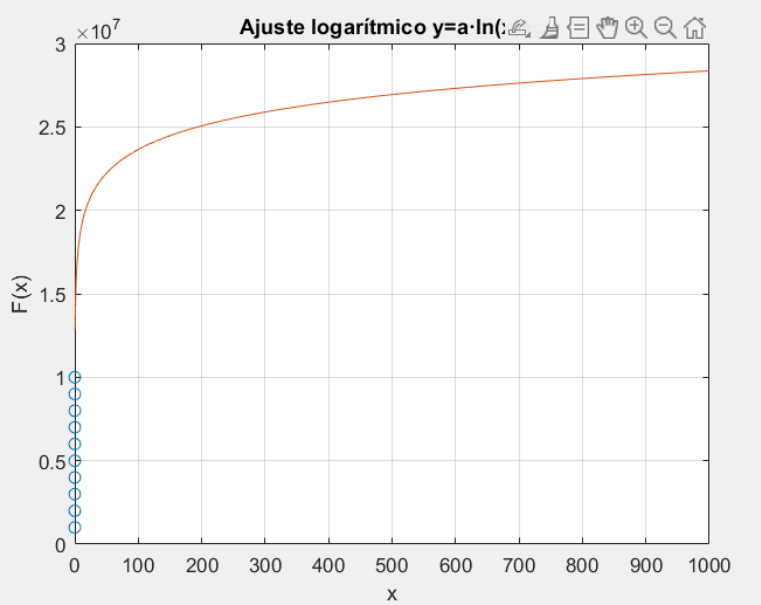


Gráfica comparativa

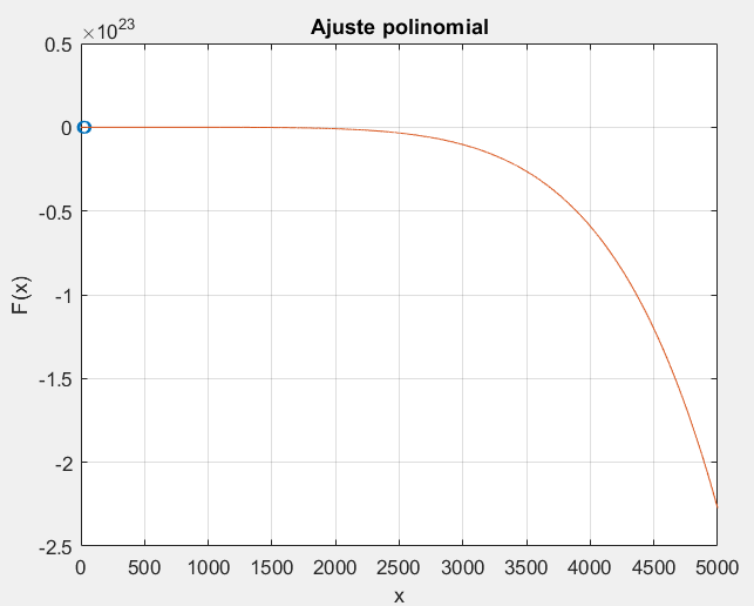


Aproximación del comportamiento temporal

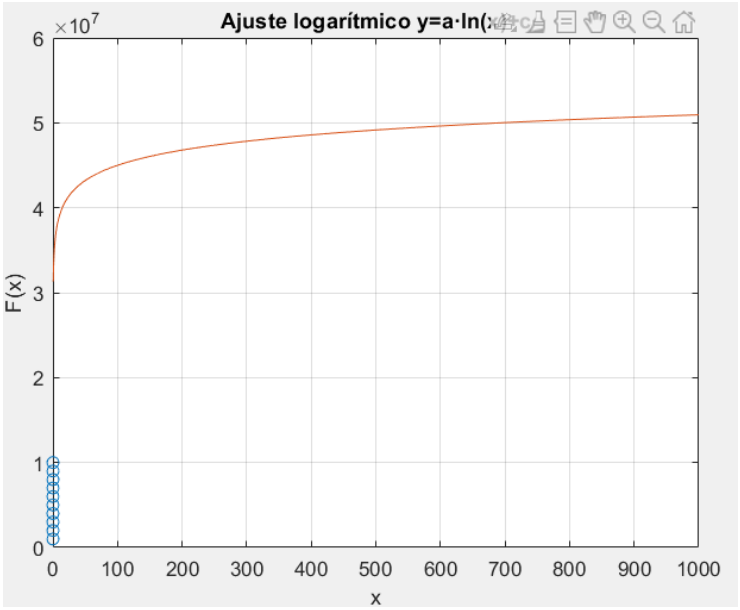
Búsqueda Lineal



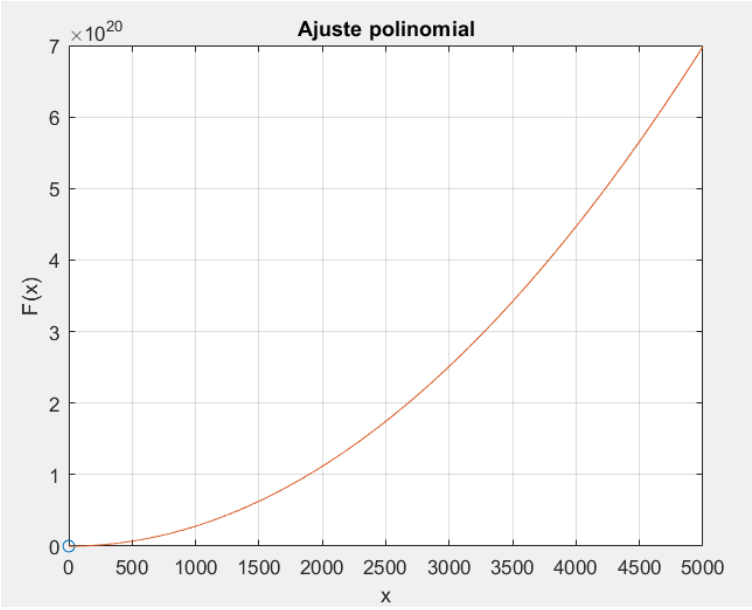
Búsqueda binaria árbol



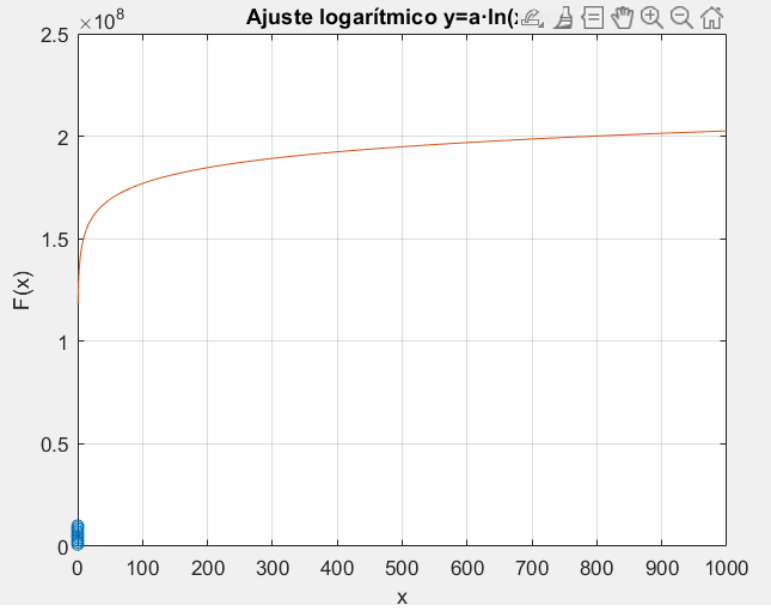
Búsqueda binaria



Búsqueda exponencial



Búsqueda Fibonacci



Constante multiplicativa

Búsqueda Lineal

Teniendo que para 1000000 tarda un promedio de 1.77e-03 s podemos decir que para un número tarda 1.77e-09 por lo tanto podemos decir que tarda 1.77e-09 en realizar una operación básica.

Caso medio:

$$f(n) = \frac{1}{2}(2) + \frac{1}{2}(n) = (1.77e-09)\frac{n}{2}$$

Búsqueda binaria árbol

Teniendo que para 1000000 tarda un promedio de 1.82e+01 s podemos decir que para un número tarda 1.82e-05 por lo tanto podemos decir que tarda 1.82e-05 en realizar una operación básica.

Caso medio:

$$f(n) = \frac{1}{3}3n + \frac{1}{3}4n + \frac{1}{3}3n = (1.82e-05)\frac{n}{3}$$

Búsqueda binaria

Teniendo que para 1000000 tarda un promedio de 1.97e-05 s podemos decir que para un número tarda 1.97e-11 por lo tanto podemos decir que tarda 1.97e-11 en realizar una operación básica.

Caso medio:

$$f(n) = \frac{1}{3}(2) + \frac{1}{3}(3n) + \frac{1}{3}(2n) = (1.97e-11)\frac{2}{3} + \frac{5n}{3}$$

Búsqueda exponencial

Teniendo que para 1000000 tarda un promedio de 1.64e-05 s podemos decir que para un número tarda 1.64e-11 por lo tanto podemos decir que tarda 1.64e-11 en realizar una operación básica.

Caso medio:

$$f(n) = \frac{1}{2}1 + \frac{1}{2}(n)\big((2) + (3n) + (2n)\big) = (1.64e-11)\frac{1}{2} + n + 2n^2$$

Búsqueda Fibonacci

Teniendo que para 1000000 tarda un promedio de 1.76e-05 s podemos decir que para un número tarda 1.76e-11 por lo tanto podemos decir que tarda 1.76e-11 en realizar una operación básica.

Caso medio:

$$f(n) = \frac{1}{3}(4n+2) + \frac{1}{3}(2n+2) + \frac{1}{3}(3n+2) = \frac{4n}{3} + \frac{2}{3} + \frac{2n}{3} + \frac{2}{3} + n + \frac{2}{3} = (1.76e-11)3n + 2$$

Tiempo de búsqueda de peor caso

Búsqueda Lineal

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = (1.77e - 09)n$$

Para 50000000

88.5e-3

Para 100000000

177e-3

Para 500000000

885e-3

Para 1000000000

1.77

Para 5000000000

8.85

Búsqueda binaria árbol

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = (1.82e - 05)4n$$

Para 50000000

3640

Para 100000000

7280

Para 500000000

36400

Para 1000000000

72.8e+3

Para 5000000000

364e+3

Búsqueda binaria

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = (1.97e - 11)3n$$

Para 50000000

2.955e-3

Para 100000000

5.91e-3

Para 500000000

29.55e-3

Para 1000000000

59.1e-3

Para 5000000000

295.5e-3

Búsqueda exponencial

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = 1 + n(3n) = (1.64e - 11)1 + 3(n^2)$$

Para 50000000

7.73e-3

Para 100000000

8.03e-3

Para 500000000

77.3e-3

Para 1000000000

80.3e-3

Para 5000000000

773e-3

Búsqueda Fibonacci

Peor caso:

Que no se encuentre en el arreglo, por lo tanto:

$$f(n) = (1.76e - 11)3n + 2$$

Para 50000000

2.64e-3

Para 100000000

52.80e-3

Para 500000000

26.4e-3

Para 1000000000

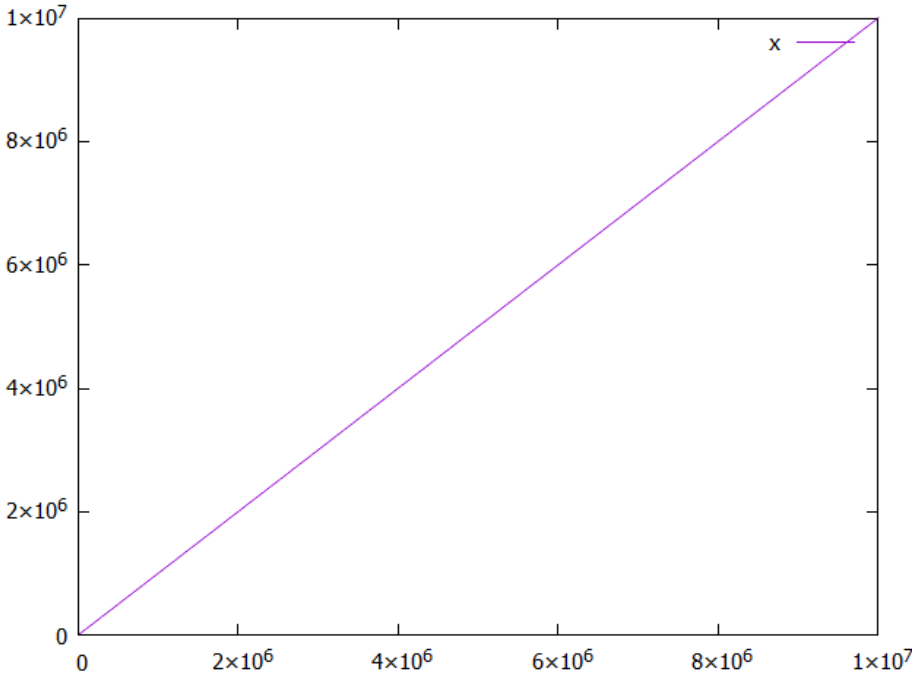
528e-3

Para 5000000000

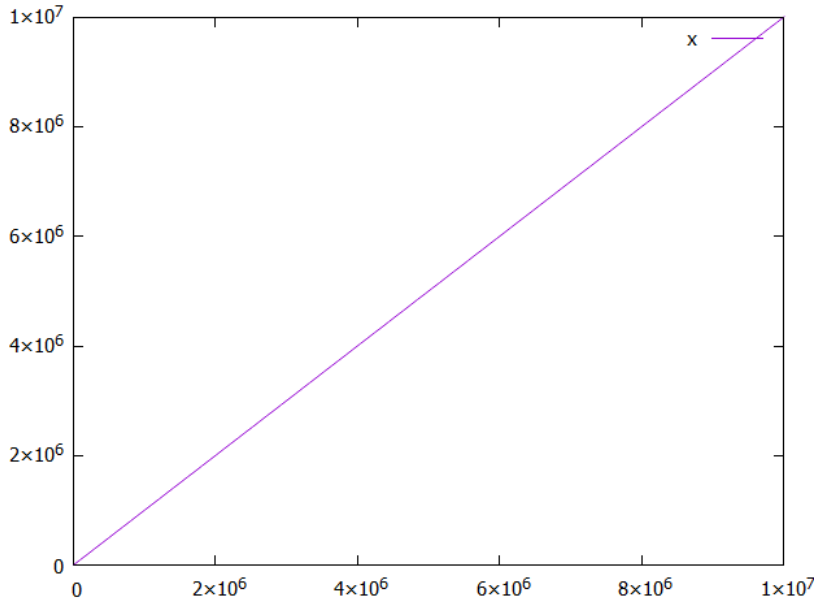
264e-3

Cotas de complejidad

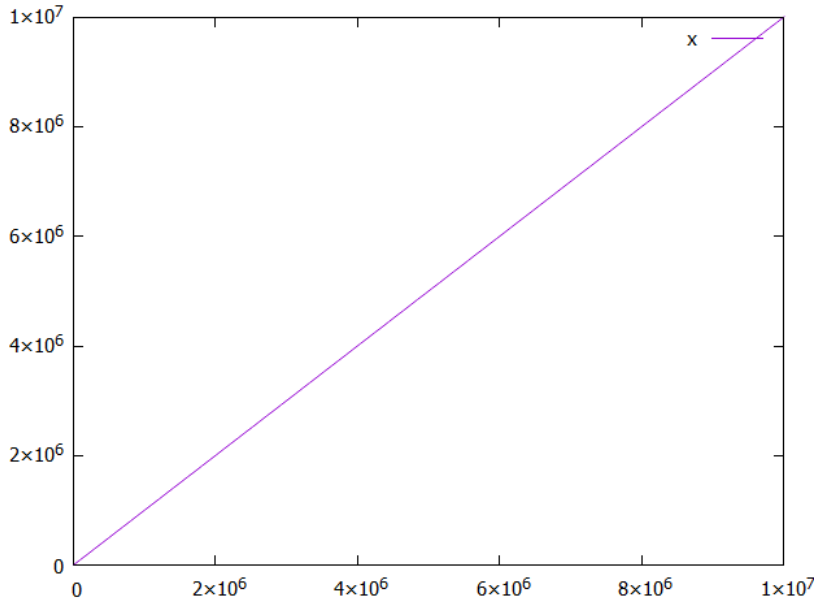
Búsqueda Lineal



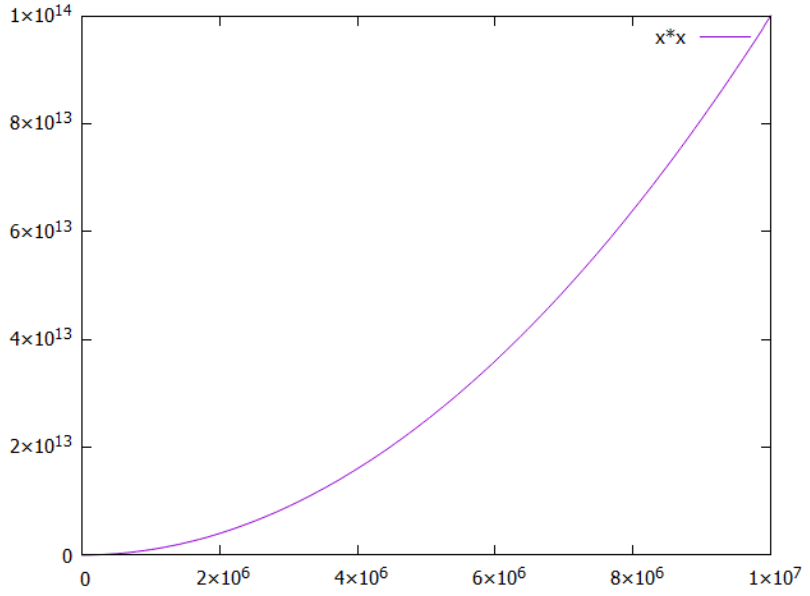
Búsqueda binaria árbol



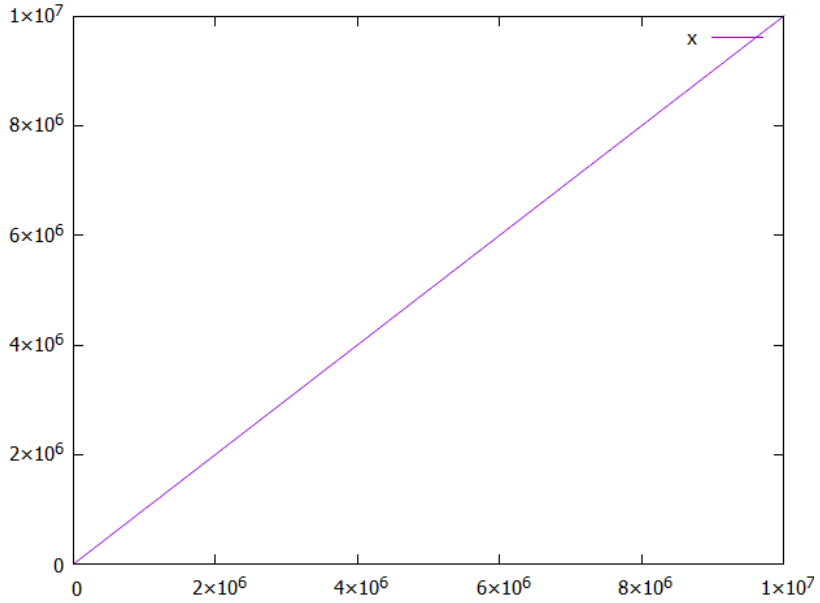
Búsqueda binaria



Búsqueda exponencial



Búsqueda Fibonacci



Cuestionario

1. ¿Cuál de los 5 algoritmos es más fácil implementar?

   R= El algoritmo de búsqueda lineal

2. ¿Cuál de los 5 algoritmos es más difícil implementar?

   R= El de búsqueda de árbol binario

3. ¿Cuál de los 5 algoritmos fue más difícil de modelar con su variante con hilos?

   R= El de búsqueda de árbol binario

4. ¿Cuál de los 5 algoritmos en su variante con hilos resulto ser más rápido? ¿Por qué?
5. ¿Cuál de los 5 algoritmos en su variante con hilos no represento alguna ventaja? ¿Por qué?
6. ¿Cuál algoritmo tiene una menor complejidad temporal?

El de búsqueda lineal

7. ¿Cuál algoritmo tiene una mayor complejidad temporal?

El ABB

8. ¿El comportamiento experimental de los algoritmos era el esperado? ¿Por qué?

En algunos casos si, porque es lo que se calculó o lo que se estimaba, pero en uno no estuve muy de acuerdo con lo obtenido, pero al ver el código se entendió mejor.

9. ¿Sus resultados experimentales difieren mucho de los análisis teóricos que realizo? ¿A qué se debe?

En algunos casos si y en otros no, esto se puede deber a que yo contaba cosas de más o de menos y eso hacia que el resultado difiriera demasiado en algunas ocasiones.

10. ¿En la versión con hilos, usar c hilos dividió el tiempo en c? ¿lo hizo c veces más rápido?

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

12. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?

Si, esta práctica se realizó en una máquina virtual por lo que se sugirió el darle una mayor capacidad de procesamiento, así como de tratar de tener los menos posibles procesos en ejecución.

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

Si, esto se debe a que los algoritmos son un proceso matemático implementado, por lo que si se podría predecir cual sería el mejor.

14. ¿Qué tan difícil fue realizar el análisis teórico de cada algoritmo?

En algunos no fue tan complicado, pero en los que usaban más de una función si lo fue.

15. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

Que observen y cuenten a detalle las operaciones de un algoritmo, ya que si te falta alguna podría afectar gradualmente a tu calculo.

Bibliografía

[1]"Linear Search - GeeksforGeeks", *GeeksforGeeks*, 2015. [Online]. Available: https://www.geeksforgeeks.org/linear-search/. [Accessed: 08- Apr- 2022].

[2]"Binary Search Tree | Set 1 (Search and Insertion) - GeeksforGeeks", *GeeksforGeeks*, 2015. [Online]. Available: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/. [Accessed: 08- Apr- 2022].