UNIVERSIDAD TECNOLÓGICA DE SANTIAGO, UTESA

SISTEMA CORPORATIVO
FACULTAD DE INGENIERIA Y ARQUITECTURA
CARRERA DE INGENIERIA EN SISTEMAS COMPUTACIONALES

ASIGNATURA:

Algoritmos Paralelos
INF-025-001

PROYECTO FINAL:

Algoritmos de Búsqueda y Ordenamiento en Paralelo

PRESENTADO A:

Iván Mendoza

PRESENTADO POR:

Liván Herrera (2-16-0686)
Carlos Acosta (2-16-1340)
Esmirna Estrella (1-17-1556)

Santiago de los Caballeros
República Dominicana
Abril, 2022

Introducción

En la actualidad, la mayoría de los algoritmos son secuenciales en el sentido de que especifican solo una secuencia de pasos, donde cada paso consta de una operación. La ventaja de estos algoritmos es que son muy adecuados para el procesamiento en las computadoras actuales. Deben tenerse en cuenta las mejoras computacionales en el procesamiento paralelo. Usamos el paralelismo para resolver problemas de manera más eficiente, pero resolver estos problemas requiere implementar un algoritmo que pueda realizar múltiples operaciones en un solo paso.

1. **Descripción del Proyecto**

   Se debe de crear una aplicación donde se ejecuten los diferentes algoritmos de búsqueda que hemos estado investigando en el transcurso del ciclo para ordenar un arreglo único. Se deben ejecutar todos los algoritmos seleccionados al mismo tiempo y se debe calcular el tiempo que tomó cada algoritmo para resolver el problema y apuntar el que menor tiempo duró en realizar la tarea.

   Este proyecto busca obtener soluciones óptimas para el procesamiento de múltiples operaciones utilizando los algoritmos de búsquedas. El propósito de este es que el usuario pueda ejecutar todos los algoritmos de búsquedas al mismo tiempo como también calcular el tiempo de ejecución de cada uno. Surge con la finalidad de sustituir los algoritmos que se ejecutan en una secuencia de pasos a un algoritmo que resuelva todas las operaciones en un solo paso y así obtener una solución más rápida y eficiente.

2. **Objetivos**

   a. **Objetivo General**

      Interactuar con los distintos tipos de algoritmos de búsquedas que se han explicado en el transcurso del cuatrimestre, observar su funcionamiento y su tiempo de reacción. A la vez que se intenta desarrollar maneras de ordenar arreglos desordenados de manera óptima y en el menor tiempo posible.

   b. **Objetivos Específicos**

      - Analizar y comprender cada uno de los algoritmos de búsquedas para la optimización de los problemas.
      - Programar algoritmos de búsqueda capaces de ordenar arreglos desordenados.
      - Cronometrar el tiempo de ejecución de los algoritmos desarrollados.
      - Realizar pruebas con los diferentes modelos de búsquedas que ayuden el trabajo en paralelo.
      - Identificar el algoritmo más rápido a la hora de realizar el ordenamiento de arreglos.
      - Explicar el por qué el algoritmo más rápido fue el más rápido.

3. **Definición de Algoritmos Paralelos**

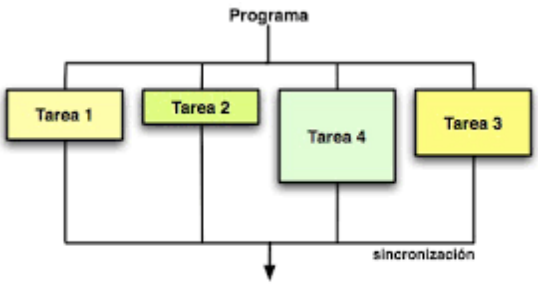


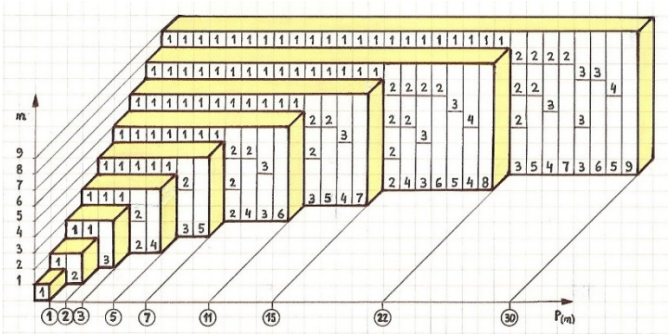
Figura 1. Ejecución de un programa paralelo

Se refiere a un algoritmo que puede ser ejecutado por distintas unidades de procesamiento en un mismo instante de tiempo, para al final unir todas las partes y obtener el resultado correcto. Dicho en otras palabras, son algoritmos que tienen la capacidad de poder realizar múltiples instrucciones de manera simultánea (al mismo tiempo) en uno o más dispositivos

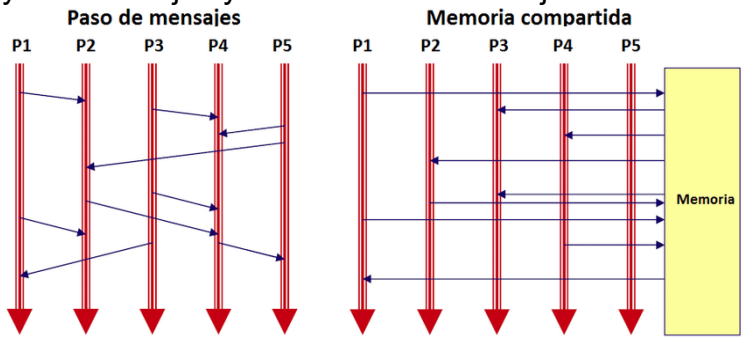
de procesamiento, y combinar dichas las salidas individuales para producir un resultado final.
Su importancia radica en que es más rápido tratar grandes tareas de computación mediante la paralelización que mediante técnicas secuenciales. Esta es la forma en que se trabaja en el desarrollo de los procesadores modernos, ya que es más difícil incrementar la capacidad de procesamiento con un único procesador que aumentar su capacidad de cómputo mediante la inclusión de unidades en paralelo.

4. Etapas de los Algoritmos paralelos

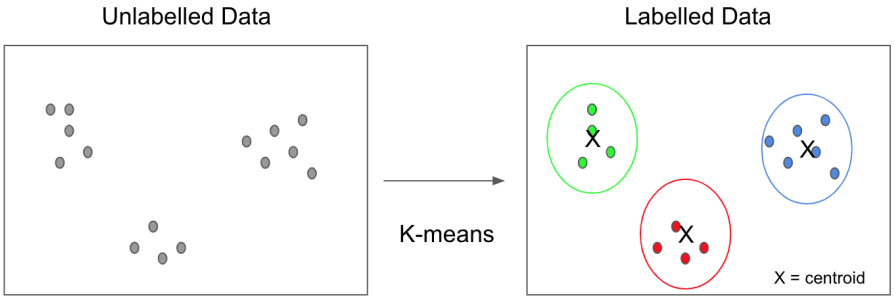
- **Partición**
  Los cálculos se descomponen en pequeñas tareas. Usualmente es independiente de la arquitectura o del modelo de programación. Un buen particionamiento divide tanto los cálculos asociados con el problema como los datos sobre los cuales opera.



- **Comunicación**
  Los algoritmos paralelos también necesitan optimizar la comunicación entre diferentes unidades de procesamiento. Esto se consigue mediante la aplicación de dos paradigmas de programación y diseño de procesadores distintos: memoria compartida o paso de mensajes.
  - ✓ La técnica memoria compartida necesita del uso de cerrojos en los datos para impedir que se modifique simultáneamente por dos procesadores, por lo que se produce un coste extra en ciclos de CPU desperdiciados y ciclos de bus.
  - ✓ La técnica paso de mensajes usa canales y mensajes, pero esta comunicación añade un coste al bus, memoria adicional para las colas y los mensajes y latencia en el mensaje.



- **Agrupamiento**
  Las tareas y estructuras de comunicación definidas en las dos primeras etapas del diseño son evaluadas con respecto a los requerimientos de ejecución y costos de implementación. Si es necesario, las tareas son combinadas en tareas más grandes para mejorar la ejecución o para reducir los costos de comunicación y sincronización.



- **Asignación**
  Cada tarea se asigna a un procesador de tal modo que intente satisfacer las metas de competencia al maximizar la utilización del procesador y minimizar los costos de comunicación.

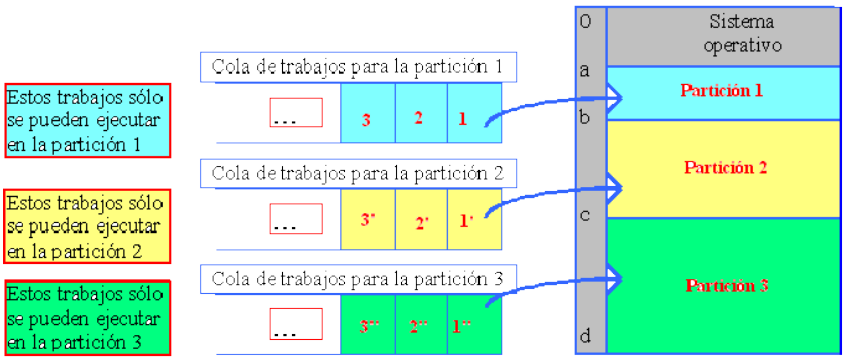


Figura 6.3. Multiprogramación con particiones estáticas, con traducción y carga absolutas

5. Técnicas Algorítmicas Paralelas

En el diseño de algoritmos paralelos existen técnicas generales que se pueden utilizar:

→ *Técnica List Ranking:* Consiste en indicar la posición de cada elemento de una lista. Por ejemplo, asignarle el numero 1 al primer elemento de una lista.

→ *Técnica de Euler:* Se utiliza para resolver ecuaciones diferenciales ordinarias (EDO) dado un valor inicial. Este método sirve para construir métodos más complejos. El método de Euler es un método de primer orden, lo que significa que el error local es proporcional al cuadrado del tamaño del paso, y el error global es proporcional al tamaño del paso.

→ *Algoritmo divide y vencerás:* Divide el problema a ser resuelto en subproblemas que son más fáciles de resolver que el problema original, resuelve los subproblemas, y se funden las soluciones a los subproblemas para construir una solución al problema original. Esta técnica mejora la modularidad del programa. Este juega un papel aún más importante en el diseño de algoritmos paralelos. Debido a los subproblemas que han sido creados en el primer paso son típicamente independientes, se pueden resolver en paralelo.

→ *Contracción de Arboles:* Este mantiene la estructura de un árbol demasiado grande y dedica sus recursos a asignar una predicción distinta a la frecuencia promedio del nodo.

→ *Aleatorización:* En algoritmos paralelos son utilizados para garantizar que los procesadores puedan tomen decisiones locales que, se suman a buenas decisiones globales. Aquí consideramos tres usos de la aleatoriedad.

→ *Técnica de Puntero Paralelo:* Esta técnica se puede aplicar a árboles o listas. En cada r a listas o árboles. En cada paso en que el puntero salta, cada nodo en paralelo reemplaza su puntero con el de su sucesor (o padre).

Muchas otras técnicas han demostrado ser útiles en el diseño de algoritmos paralelos. Encontrar pequeños separadores gráficos es útil para dividir datos entre procesadores y reducir la comunicación. Las funciones hash son útiles para el equilibrio de carga y el equilibrio de carga, y para asignar direcciones de memoria.

6. Modelos de Algoritmos Paralelos

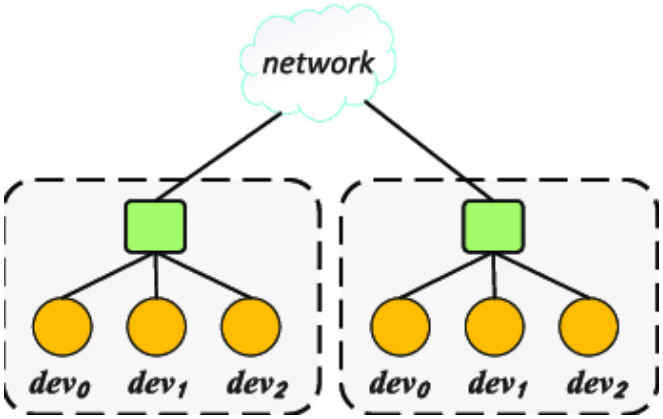
Un modelo de programación paralela es un modelo para escribir programas paralelos los cuales pueden ser compilados y ejecutados. El valor de un modelo de programación puede ser juzgado por su generalidad (Si las soluciones ofrecidas son óptimas a comparación de diferentes arquitecturas o soluciones existentes), y su rendimiento (Eficiencia, precisión o velocidad de la ejecución).

Las características de los modelos de programación paralela se pueden subdividir ampliamente, pero se puede generalizar en 2 rasgos fundamentales: la interacción de procesos y los problemas de descomposición.

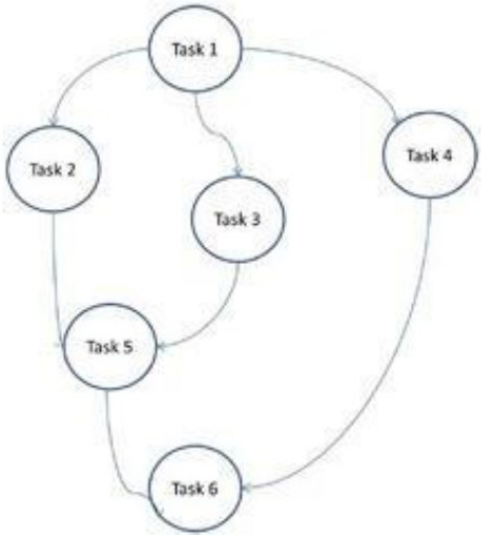
- ✓ Interacción de proceso: La interacción de proceso se refiere a los mecanismos por los cuales procesos paralelos son capaces de comunicarse entre sí. Las formas más comunes de interacción son la memoria y el paso de mensajes compartidos, pero también puede ser implícita.
- ✓ Descomposición de problema: Un programa paralelo está compuesto de procesos que están ejecutándose simultáneamente. La descomposición del problema se refiere a la forma en que se formula estos procesos.

Algunos de los ejemplos de modelos de programación paralela son:

→ Modelo de Data-Parallel: Muchos datos son tratados de una igual o similar. Se usa el paralelismo de asignación a diferentes partes del array y a diferentes procesadores.

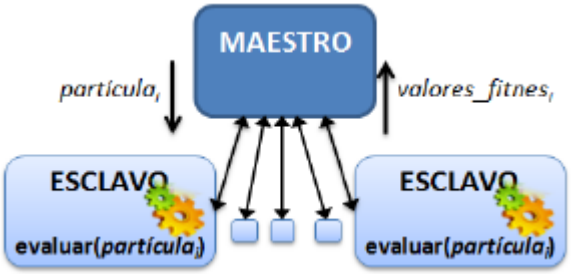


→ Modelo de Gráfico de Tareas: el paralelismo se expresa mediante un gráfico de tareas. Un gráfico de tareas puede o no ser trivial. En este modelo, la correlación entre tareas se utiliza para promover la localidad o para minimizar los costos de interacción.

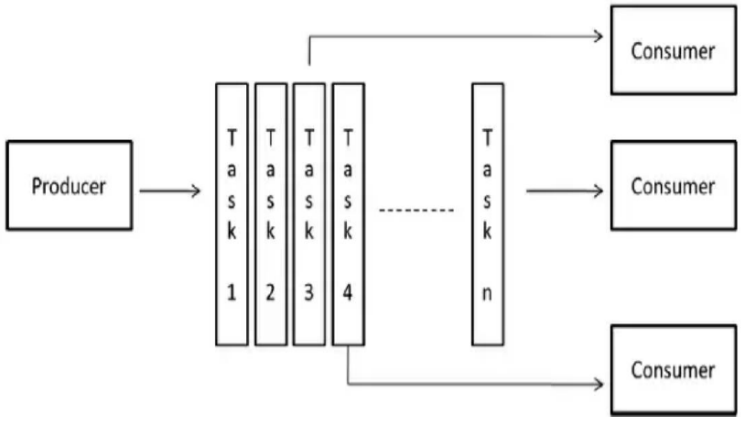


→ Modelo de Grupo de Trabajo: En el modelo de grupo de trabajo, las tareas se asignan dinámicamente a los procesos de equilibrio de carga. Por lo tanto, cualquier proceso puede potencialmente realizar cualquier tarea. Este modelo se utiliza cuando la cantidad de data asociados con la tarea es comparativamente menor que el cálculo asociado con las tareas.

→ Modelo Maestro Esclavo: En el modelo maestro-esclavo, uno o más procesos maestros generan una tarea y la asignan a procesos esclavos. Se debe tener cuidado de que el maestro no se convierta en una congestión. Esto puede suceder si las tareas son demasiado pequeños o los trabajadores son relativamente rápidos.

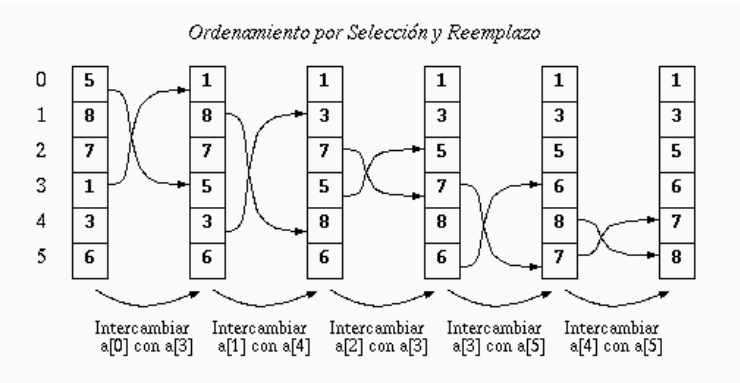


→ Modelo de Canalización: También se lo conoce como modelo productor-consumidor. Aquí, un conjunto de data se pasa a través de una serie de procesos, cada uno de los cuales realiza una tarea. Aquí, la llegada de nuevos data genera la ejecución de una nueva tarea por un proceso en la cola. Los procesos pueden formar una cola en forma de matrices lineales o multidimensionales, árboles o gráficos generales con o sin ciclos.



→ Modelos Híbridos: Un modelo de algoritmo hibrido es necesario cuando se pueden necesitar varios modelos para resolver un problema. Un modelo hibrido puede estar compuesto por varios modelos aplicados jerárquicamente o por varios modelos aplicados secuencialmente a diferentes fases de un algoritmo paralelo.

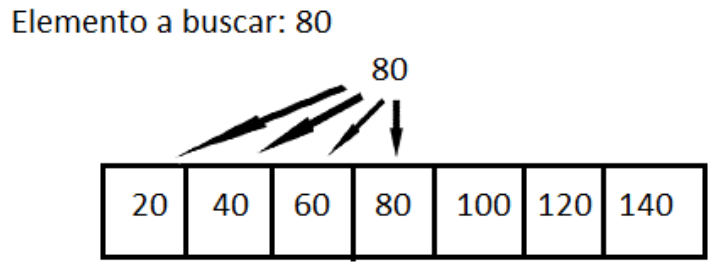
7. Algoritmos de Búsquedas y Ordenamiento (Adjuntar PSeudocódigo y código de cada uno en su lenguaje de programación)



Búsqueda y ordenamiento son los elementos fundamentales para la organización de los datos. Estas operaciones toman como referencia la clave de los datos. Cabe destacar que las búsquedas nos permiten encontrar un elemento dentro de una lista o arrays, mientras que el ordenamiento nos permite a identificar la ubicación donde se encuentran dichos datos almacenados.

En pocas palabras, los algoritmos de búsquedas son las instrucciones que nos permiten ubicar los elementos dentro de una estructura de datos.

a. Búsqueda Secuencial



El método consiste en tomar algún dato clave que identifica al elemento que se está buscando y hace un recorrido por todo el arreglo, para luego comparar el dato de referencia con el dato que tiene cada posición. Es más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio.

Este método se utiliza cuando el contenido de Vector no se puede encontrar o no se puede ordenar. Implica buscar un elemento comparando secuencialmente el elemento (de ahí el nombre) con cada elemento de la matriz o conjunto de datos hasta que se encuentra, o hasta que se alcanza el final de la matriz. La existencia de un elemento se puede determinar desde el momento en que se coloca, pero no podemos garantizar que no exista hasta que analicemos todos los elementos de la disposición.

Se pueden considerar dos variantes del método: con y sin centinela.

- → Búsqueda sin centinela: el algoritmo simplemente recorre el array comparando cada elemento con el dato que se está buscando.
- → Búsqueda con centinela: se almacena un elemento adicional (centinela), que coincidirá con el elemento buscado y que se situará en la última posición del array de datos

b. Búsqueda Binaria

Este método se conoce por ser más eficiente que la búsqueda secuencial, pero este sólo puede ser aplicado sobre vectores o listas de datos ordenados. Aquí no se hace un recorrido de principio a fin, lo que se hace es delimitar progresivamente el espacio de búsqueda hasta que llegue al elemento buscado.

El de búsqueda secuencia es más fácil de implementar, pero tarda mucho tiempo para realizar la búsqueda mientras que el binario es más eficiente, pero requiere de un ordenamiento.

Tiene varios prerrequisitos:

→ El conjunto de búsqueda está ordenado.
→ Se dispone de acceso aleatorio.

Este algoritmo compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior, respectivamente, al elemento central, y así sucesivamente.

c. Algoritmo de Ordenamiento de la Burbuja

Vector original

45	17	23	67	21

Iteración 1:

45	17	23	67	21	Se genera cambio
17	45	23	67	21	Se genera cambio
17	23	45	67	21	No hay cambio
17	23	45	67	21	Se genera cambio
17	23	45	21	67	Fin primera iteración

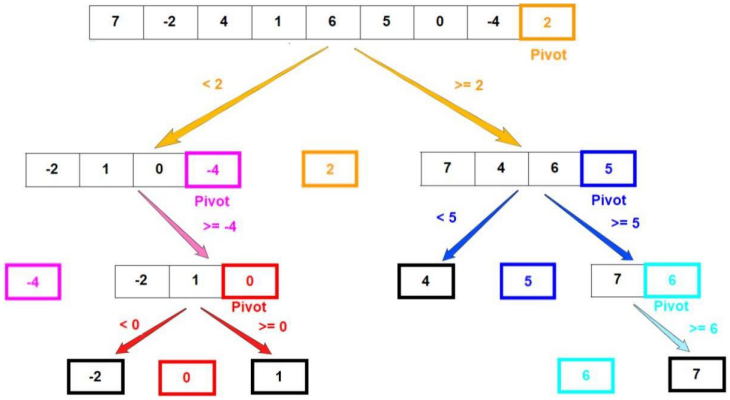
Este es un algoritmo que nos permite ordenar vectores o matrices. Se basa en recorrer el array un cierto número de veces, comparando pares de valores que ocupan posiciones adyacentes (0-1,1-2, ...). Si ambos datos no están ordenados, se intercambian. Esta operación se repite n-1 veces, siendo n el tamaño del conjunto de datos de entrada. Al final de la última pasada el elemento mayor estará en la última posición; en la segunda, el segundo elemento llegará a la penúltima, y así sucesivamente.

Su nombre se debe a que el elemento cuyo valor es mayor sube a la posición final del array, al igual que las burbujas de aire en un depósito suben a la parte superior. Para ello debe realizar un recorrido paso a paso desde su posición inicial hasta la posición final del array.

Advertencia:

→ Es ineficiente comparado con los otros algoritmos.
→ No se debería utilizar para la ordenación de datos.

d. Quick Sort

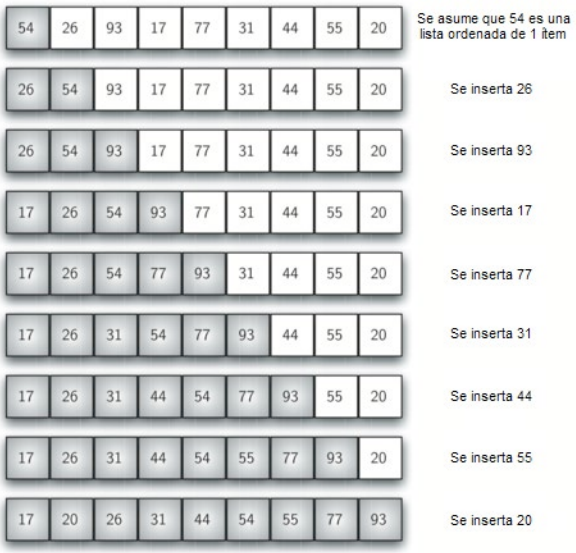


Es un algoritmo ordenado muy eficiente y es un poco más complejo que el de ordenamiento de burbuja y el método de inserción. Una de las desventajas de este algoritmo es que su codificación es compleja y debemos hacer uso de la recursividad.

Se basa en la estrategia típica de "divide y vencerás". El array a ordenar se divide en dos partes: una contendrá todos los valores menores o iguales a un cierto valor (que se suele denominar pivote) y la otra con los valores mayores que dicho valor. El primer paso es dividir el array original en dos subarrays y un valor que sirve de separación, esto es, el pivote. Así, el array se dividirá en tres partes:

→ La parte izquierda, que contendrá valores inferiores o iguales al pivote.
→ El pivote.
→ La parte derecha, que contiene valores superiores o iguales al pivote.

e. Método de Inserción



Es un algoritmo de ordenación de matrices. Este funciona recorriendo cada elemento del array y para cada uno de esos elementos se comprueba si esta ordenado o no. Para saber si el elemento se encuentra ordenado tiene que cumplirse que un elemento a sea menor o igual que el elemento siguiente.

Lo que hacemos luego es comprobar que elemento que tiene a su izquierda es mayor que este propio elemento y si se da ese caso entonces significa que está desordenado así que tenemos que hacer un intercambio esto se repite hasta que se encuentre que está ordenado o bien hasta que se llega a la primera posición.

PSEUDOCÓDIGOS (EJEMPLOS EN C & PROYECTO EN GO)

Búsqueda secuencial (sin centinela & con centinela)

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
 printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM];
 int i, dato;

 srand((unsigned int)time(NULL));
 for(i = 0; i < TAM; i++)
    CB[i] = (int)(rand() % 100);
 imprimeCB(CB);
 dato = (int)(rand() % 100);
 printf("Dato a buscar %d\n",dato);

 i=0;
 while ((CB[i]!=dato) && (i<TAM))
    i++;

 if (CB[i]==dato) printf("Posicion %d\n",i);
 else printf("Elemento no esta en el array");
}
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
    printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM+1];
 int i, dato;

 srand((unsigned int)time(NULL));
 for(i = 0; i < TAM; i++)
    CB[i] = (int)(rand() % 100);
 imprimeCB(CB);
 dato = (int)(rand() % 100);
 CB[i] = dato;
 printf("Dato a buscar %d\n",dato);

 i=0;
 while (CB[i]!=dato)
    i++;

 if (CB[i]==dato) printf("Posicion %d\n",i);
 else printf("Elemento no esta en el array");
}
```

```go
go func() {

    var a [100]int
    var dato int
    defer MedirTiempo(time.Now(), "Busqueda Secuencial")

    ArregloOrdenado(&a)
    dato = 60 //Dato a buscar
    i := 0

    for (a[i] != dato) && (i < 100) {
        i++
        if a[i] == dato {
            fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, i)

        }
    }
}()
```

Búsqueda Binaria

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
    printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM];
 int ini=0,fin=TAM-1,mitad,dato,i;

 srand((unsigned int)time(NULL));
 for(i = 0; i < TAM; i++)
    CB[i] = (int)(rand() % 100);
 imprimeCB(CB);
 dato = (int)(rand() % 100);
 CB[i] = dato;
 printf("Dato a buscar %d\n",dato);

 mitad=(ini+fin)/2;
 while ((ini<=fin)&&(CB[mitad]!=dato))
 {
  if (dato < CB[mitad])
  fin=mitad-1;
  else ini=mitad+1;
  mitad=(ini+fin)/2;
 }

 if (dato==CB[mitad]) printf("Posicion %d\n", mitad);
 else printf("Elemento no esta en el array");
getch();
}
```

```go
go func() {

    var a [100]int
    ArregloOrdenado(&a)
    defer MedirTiempo(time.Now(), "Búsqueda Binaria")

    inicio := 0
    fin := len(a) - 1
    var mitad, dato int

    dato = 45 //Dato a buscar
    mitad = (inicio + fin) / 2
    for (inicio <= fin) && (a[mitad] != dato) {
        if dato < a[mitad] {
            fin = mitad - 1
        } else {
            inicio = mitad + 1

        }
        mitad = (inicio + fin) / 2
    }

    if dato == a[mitad] {
        fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, mitad)

    }
}()
```

Algoritmo de Ordenamiento de Burbuja

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
    printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM];
 int e, i, auxiliar;

 srand((unsigned int)time(NULL));

 for(e = 0; e < TAM; e++)
    CB[e] = (int)(rand() % 100);

 printf( "Antes de ordenar\--------------------\n");
 imprimeCB(CB);

 for(e = 0; e < TAM; e++)
    for(i = 0; i < TAM-1-e; i++)
        if(CB[i] > CB[i+1]) {
            auxiliar = CB[i+1];
            CB[i+1] = CB[i];
            CB[i] = auxiliar;
 }

 printf( "\nDespués de ordenar\n------------------\n");
 imprimeCB(CB);
}
```

```go
go func() {

    var a = []int{13, 25, 3, 1, 110, 102, 58, 91, 150, 33, 8, 69, 137, 54, 76}
    var n int = len(a)
    defer MedirTiempo(time.Now(), "Ordenamiento de Burbuja")
    var e, i, auxiliar int

    for e = 0; e < n; e++ {
        for i = 0; i < n-1-e; i++ {
            if a[i] > a[i+1] {
                auxiliar = a[i+1]
                a[i+1] = a[i]
                a[i] = auxiliar
            }
        }
    }

    fmt.Println(a)
}()
```

Quick Sort

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void quickSort( int[], int, int);
int partition( int[], int, int);

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
    printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM];
 int e;
 srand((unsigned int)time(NULL));

 for(e = 0; e < TAM; e++)
    CB[e] = (int)(rand() % 100);

 printf( "Antes de ordenar\n----------------------\n");
 imprimeCB(CB);

 quickSort( CB, 0, TAM-1);

 printf( "\nDespués de ordenar\n--------------------\n");
 imprimeCB(CB);
}

void quickSort( int CB[], int izquierda, int derecha){
 int indice_pivote;
 if( izquierda < derecha ) {
    indice_pivote = partition( CB, izquierda, derecha);
    quickSort( CB, izquierda, indice_pivote-1);
    quickSort( CB, indice_pivote+1, derecha);
 }
}

int partition( int CB[], int izquierda, int derecha) {
 int pivote, i, j, tmp;
 pivote = CB[izquierda];
 i = izquierda; j = derecha;

 while( 1){
    while( CB[i] <= pivote && i <= derecha ) ++i;
    while( CB[j] > pivote ) --j;
    if( i >= j ) break;
    tmp = CB[i]; CB[i] = CB[j]; CB[j] = tmp;
 }
 tmp = CB[izquierda]; CB[izquierda] = CB[j]; CB[j] = tmp;
 return j;
}
```

```go
func QuickSort(a []int, izq int, der int) []int {

    pivote := a[izq]
    i := izq
    j := der
    var aux int

    for i < j {
        for a[i] <= pivote && i < j {
            i++
        }
        for a[j] > pivote {
            j--
        }
        if i < j {
            aux = a[i]
            a[i] = a[j]
            a[j] = aux
        }
    }

    a[izq] = a[j]
    a[j] = pivote

    if izq < j-1 {
        QuickSort(a, izq, j-1)
    }
    if j+1 < der {
        QuickSort(a, j+1, der)
    }

    return a
}
```

Método de Inserción

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define TAM 100

void imprimeCB(int *CB) {
 int i;
 for(i = 0; i < TAM-1; i++) {
    printf( "%d, ", CB[i]);
 }
 printf( "%d\n", CB[i]);
}

int main() {
 int CB[TAM];
 int e,i,k,temp;

 srand((unsigned int)time(NULL));

 for(e = 0; e < TAM; e++)
    CB[e] = (int)(rand() % 100);

 printf( "Antes de ordenar\n----------------------------\n");
 imprimeCB(CB);

 for (e=1;e<TAM;e++){
    temp=CB[e];
    i=0;
    while (CB[i]<=temp)
        i++;
    if (i<e)
        {
        for (k=e;k>i;k--)
            CB[k]=CB[k-1];
            CB[i]=temp;
        }
    }

 printf( "\nDespués de ordenar\n-----------------------\n");
 imprimeCB(CB);
}
```

```go
go func() {

    var a = []int{13, 25, 3, 1, 110, 102, 58, 91, 150, 33, 8, 69, 137, 54, 76}
    var n int = len(a)
    defer MedirTiempo(time.Now(), "Insercion")
    var auxiliar int

    for i := 1; i < n; i++ {
        auxiliar = a[i]
        for j := i - 1; j >= 0 && a[j] > auxiliar; j-- {
            a[j+1] = a[j]
            a[j] = auxiliar
        }
    }
    fmt.Println(a)
}()
```

8. Programa desarrollado

a. Explicación de su funcionamiento

- ✓ **Búsqueda Secuencial:** El algoritmo simplemente recorre el array comparando cada elemento con el dato que se está buscando.

Pseudocódigo

```
1. Función imprimir (entero a)
      a. Entero i
      b. Para (i igual a 0; i menor que 100 – 1; sumar i
         + 1)
           i. Imprimir (a[i])
2. Función LlenarArray (entero a)
      a. Entero i
      b. Entero c igual a 207
      c. Para (i igual a 0; i menor que 100; sumar i +
         1)
           i. A[i] igual a c
          ii. Sumar c + 7
3. Función main
      a. Entero a[TAM (100)]
      b. Entero i, dato
      c. Llamar función "LlenarArray" con valores a
      d. Llamar función "Imprime" con valores a
      e. Declarar dato igual a 816
      f. Imprimir "Dato a buscar", dato
      g. Declarar i igual a 0
      h. Mientras a[i] sea diferente de dato y i sea
         menor que 100, entonces
           i. Sumar i + 1;
          ii. Si a[i] es igual a dato
                 1. Imprimir "Posición", i
         iii. Sino
                 1. Imprimir "El elemento no está en el
                    array"
```

→ **Búsqueda Binaria:** Este algoritmo compara el dato buscado con el elemento central. Según sea menor o mayor se prosigue la búsqueda con el subconjunto anterior o posterior, respectivamente, al elemento central, y así sucesivamente.

Pseudocódigo

```
1) Función imprimir (entero a)
      a. Entero i
      b. Para (i igual a 0; i menor que 100 – 1; sumar i
         + 1)
           i. Imprimir (a[i])
2) Función LlenarArray (entero a)
      a. Entero i
      b. Entero c igual a 207
      c. Para (i igual a 0; i menor que 100; sumar i +
         1)
           i. A[i] igual a c
          ii. Sumar c + 7
3) Función main
      a. Entero a[100]
      b. Llamar función "LlenarArray" con valores a
      c. Llamar función "imprime" con valores a
      d. Entero inicio igual a 0, fin igual a 100 -1,
         mitad, dato, i
      e. Declarar dato igual a 816
      f. Imprimir "Dato a buscar", dato
      g. Declarar mitad igual a (inicio más fin) dividido
         entre 2
      h. Mientras inicio sea menor o igual que fin y
         a[mitad] sea diferente de dato, entonces
           i. Si (dato es menor que a[mitad])
                1. Fin es igual a mitad menos 1
          ii. Sino
                1. Inicio es igual a mitad más 1
         iii. Declarar mitad igual a (inicio más fin)
              dividido entre 2
```

```
        i. Si (dato es igual a a[mitad]), entonces
            i. Imprimir "Posicion", mitad
        j. Si no, entonces
            i. Imprimir "El elemento no está en el array"
```

→ **Ordenamiento de Burbuja:** Se basa en recorrer el array un cierto número de veces, comparando pares de valores que ocupan posiciones adyacentes (0-1,1-2, ...). Si ambos datos no están ordenados, se intercambian. Esta operación se repite n-1 veces, siendo n el tamaño del conjunto de datos de entrada.

Pseudocódigo

```
1) Función imprimir (entero a)
        a. Entero i
        b. Para (i igual a 0; i menor que 100 – 1; sumar i
           + 1)
            i. Imprimir (a[i])
2) Función LlenarArray (entero a)
        a. Entero i
        b. Entero c igual a 1
        c. Para (i igual a 0; i menor que 100; sumar i +
           1)
            i. A[i] igual a (entero) ((c por i más 1024)
               por ciento 500)
           ii. Sumar c + 1
3) Función main
        a. Entero a[100]
        b. Llamar "LlenarArray" con valores a
        c. Entero e, i, auxiliar
        d. Imprimir "Antes de ordenar-------------"
        e. Llamar función "Imprime" con valores a
        f. Para (e igual a 0; e menor que 100; sumar e +
           1)
            i. Para (i igual a 0; i menor que 100 menos
               1 menos e; sumar i más 1)
```

```
                1. Si (a[i] es mayor que a[i más 1]),
                   entonces
                      a. Auxiliar es igual a a[i más 1]
                      b. A[i más 1] es igual a a[i]
                      c. A[i] es igual a auxiliar
        g. Imprimir "después de ordenar--------"
        h. Llamar función "imprime" con valores a
```

→ **QuickSort:** El array a ordenar se divide en dos partes: una contendrá todos los valores menores o iguales a un cierto valor (que se suele denominar pivote) y la otra con los valores mayores que dicho valor.

Pseudocódigo

```
Función QuickSort(a []int, izq int, der int) []int {
i<-izq
j<-der
Para i menor que j {
     Para a[i] menor o igual que pivote && i menor que
     j {
          i++ }
     Para a[j] mayor que pivote {
               j--
          }
          Si i es menor que j {
               aux igual a a[i]
               a[i] igual a a[j]
               a[j] igual a aux
          }
     }
a[izq] igual a a[j]
     a[j] igual a pivote
```

```
Si izq menor que j-1 {
      QuickSort(a, izq, j-1)
}
Si j+1 menor que der {
      QuickSort(a, j+1, der)
}
return a }
```

→ **Método de inserción:** El primer elemento del array (a[0]) se considerado ordenado (la lista inicial consta de un elemento). A continuación se inserta el segundo elemento (a[1]) en la posición correcta (delante o detrás de a[0]) dependiendo de que sea menor o mayor que a[0]. Repetimos esta operación sucesivamente de tal modo que se va colocando cada elemento en la posición correcta. El proceso se repetirá TAM-1 veces.

Pseudocódigo

```
1) Función imprimir (entero a)
      a. Entero i
      b. Para (i igual a 0; i menor que 100 – 1; sumar i
         + 1)
           i. Imprimir (a[i])
2) Función LlenarArray (entero a)
      a. Entero i
      b. Entero c igual a 1
      c. Para (i igual a 0; i menor que 100; sumar i +
         1)
           i. A[i] igual a (entero) ((c por i más 1024)
              por ciento 500)
          ii. Sumar c + 1
3) Función main
      a. Entero a[100]
      b. Entero e, i, k, temp
      c. Llamar "LlenarArray" con valores a
```

d. Imprimir “Antes de ordenar----------”
e. Llamar función “Imprime” con valores a
f. Para (e igual a 0; e menor que 100; sumar e + 1)
    i. Temp igual a a[e]
    ii. I igual a 0
    iii. Mientras a[i] sea menor o igual que temp, entonces
        1. Sumar i más 1
        2. Si (i es menor que e), entonces
            a. Para (k igual a e; k menor que i; k menos 1)
                i. A[k] igual a a[k menos 1]
                ii. A[i] igual a temp
g. Imprimir “después de ordenar--------”
h. Llamar función “imprime” con valores a

b. Fotos de la aplicación

Búsqueda Secuencial

```
go func() {

    var a [100]int
    var dato int
    defer MedirTiempo(time.Now(), "Busqueda Secuencial")

    ArregloOrdenado(&a)
    dato = 60 //Dato a buscar
    i := 0

    for (a[i] != dato) && (i < 100) {
        i++
        if a[i] == dato {
            fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, i)

        }
    }
}()
```

Búsqueda Binaria

```
go func() {

    var a [100]int
    ArregloOrdenado(&a)
    defer MedirTiempo(time.Now(), "Búsqueda Binaria")

    inicio := 0
    fin := len(a) - 1
    var mitad, dato int

    dato = 45 //Dato a buscar
    mitad = (inicio + fin) / 2
    for (inicio <= fin) && (a[mitad] != dato) {
        if dato < a[mitad] {
            fin = mitad - 1
        } else {
            inicio = mitad + 1

        }
        mitad = (inicio + fin) / 2
    }

    if dato == a[mitad] {
        fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, mitad)

    }
}()
```

Algoritmo de Ordenamiento de Burbuja

```
go func() {

    var a = []int{13, 25, 3, 1, 110, 102, 58, 91, 150, 33, 8, 69, 137, 54, 76}
    var n int = len(a)
    defer MedirTiempo(time.Now(), "Ordenamiento de Burbuja")
    var e, i, auxiliar int

    for e = 0; e < n; e++ {
        for i = 0; i < n-1-e; i++ {
            if a[i] > a[i+1] {
                auxiliar = a[i+1]
                a[i+1] = a[i]
                a[i] = auxiliar
            }
        }
    }

    fmt.Println(a)
}()
```

Quick Sort

```go
func QuickSort(a []int, izq int, der int) []int {

    pivote := a[izq]
    i := izq
    j := der
    var aux int

    for i < j {
        for a[i] <= pivote && i < j {
            i++
        }
        for a[j] > pivote {
            j--
        }
        if i < j {
            aux = a[i]
            a[i] = a[j]
            a[j] = aux
        }
    }

    a[izq] = a[j]
    a[j] = pivote

    if izq < j-1 {
        QuickSort(a, izq, j-1)
    }
    if j+1 < der {
        QuickSort(a, j+1, der)
    }

    return a
}
```

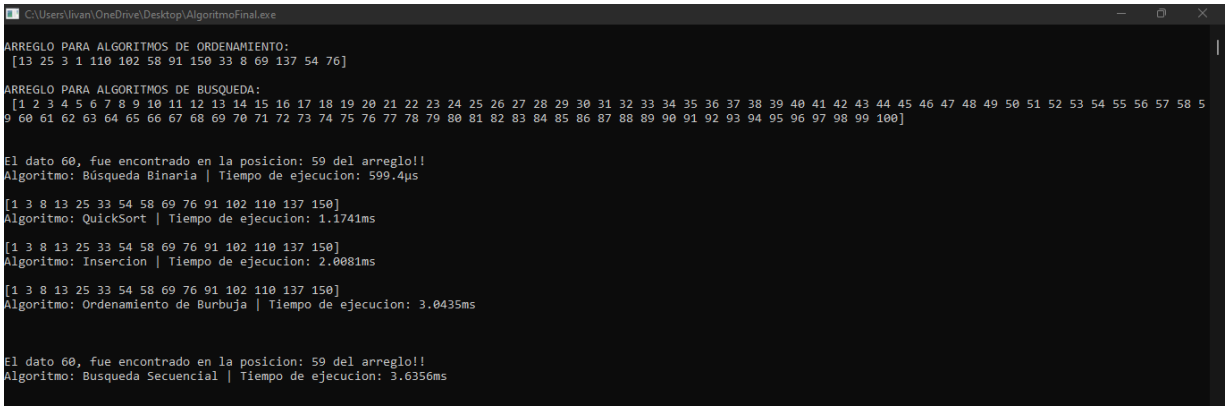
Inserción

```go
go func() {

    var a = []int{13, 25, 3, 1, 110, 102, 58, 91, 150, 33, 8, 69, 137, 54, 76}
    var n int = len(a)
    defer MedirTiempo(time.Now(), "Insercion")
    var auxiliar int

    for i := 1; i < n; i++ {
        auxiliar = a[i]
        for j := i - 1; j >= 0 && a[j] > auxiliar; j-- {
            a[j+1] = a[j]
            a[j] = auxiliar
        }
    }
    fmt.Println(a)
}()
```

c. **Link de Github y Ejecutable de la aplicación**

https://github.com/livanh1/AlgoritmosParalelosFinal.git

d. **Resultados (Tiempo en terminar los ordenamientos y búsqueda de cada algoritmo) [Milisegundos (ms) → Microsegundos (µs)]**

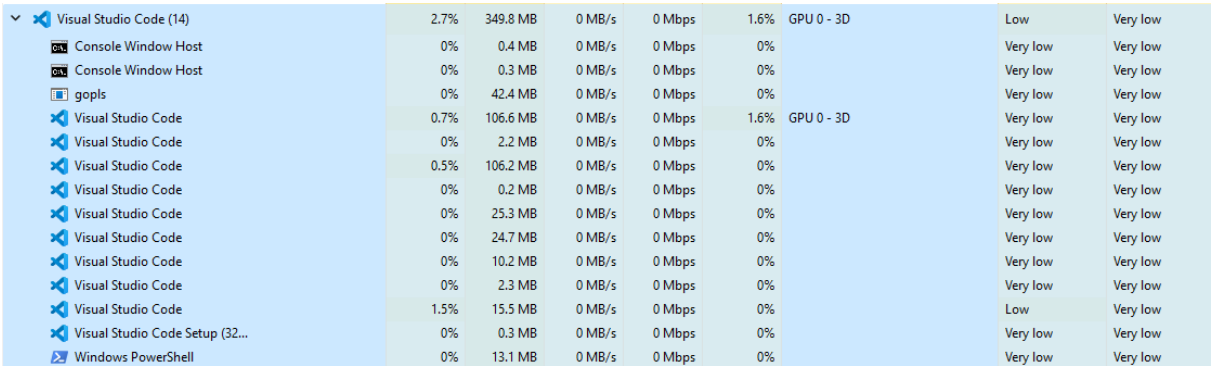


- ✓ **Algoritmos de Búsqueda**
  - **Búsqueda Secuencial:** 3.6356 ms
  - **Búsqueda Binaria:** 599.4 µs (El más rápido)

- ✓ **Algoritmos de Ordenamiento**
  - **Ordenamiento de Burbuja:** 3.0435 ms
  - **QuickSort:** 1.1741 ms (El más rápido)
  - **Inserción:** 2.0081 ms

e. **¿Qué tanta memoria se consumió este proceso?**

Entre 322 MB y 400 MB de consumo en la memoria.



Visual Studio Code (14)	2.7%	349.8 MB	0 MB/s	0 Mbps	1.6%	GPU 0 - 3D	Low	Very low
Console Window Host	0%	0.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Console Window Host	0%	0.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
gopls	0%	42.4 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0.7%	106.6 MB	0 MB/s	0 Mbps	1.6%	GPU 0 - 3D	Very low	Very low
Visual Studio Code	0%	2.2 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0.5%	106.2 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0%	0.2 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0%	25.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0%	24.7 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0%	10.2 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	0%	2.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Visual Studio Code	1.5%	15.5 MB	0 MB/s	0 Mbps	0%		Low	Very low
Visual Studio Code Setup (32...	0%	0.3 MB	0 MB/s	0 Mbps	0%		Very low	Very low
Windows PowerShell	0%	13.1 MB	0 MB/s	0 Mbps	0%		Very low	Very low

9. **¿Cuál fue el algoritmo que realizo la búsqueda y el ordenamiento más rápido? Explique.**

- ✓ **Algoritmos de Búsqueda**
  - **Búsqueda Binaria**

    Con un tiempo de 599.4 µs, el algoritmo de Búsqueda Binaria fue el más rápido en completar la tarea, esto debido a que el dato que se iba a buscar (60) se encontraba cerca de la mitad del arreglo, hizo que este algoritmo sacara ventaja. Como este algoritmo funciona tomando como pivote el centro o la mitad del arreglo, al descartar la primera mitad ya que el dato a buscar (60) era mayor que el pivote solo tenía que buscar en la sección de la derecha del arreglo, y como el dato se encontraba cerca, pues le fue fácil al algoritmo terminar la tarea de manera rápida y eficiente.

- ✓ **Algoritmos de Ordenamiento**
  - **Algoritmo de QuickSort**

    Con un tiempo de 1.1741 ms, el algoritmo de Quicksort fue el más rápido en realizar el ordenamiento del arreglo. Lo que favoreció en este caso al algoritmo de quicksort, es que el pivote (en este caso el primer elemento del arreglo), resulta ser el centro (o casi el centro) del arreglo. Por lo que tuvo que realizar muchas menos comparaciones que si el pivote hubiera sido un extremo del arreglo luego de ordenarlo.

Conclusión

En esta prueba de algoritmos paralelos, dando uso de algoritmos tanto de ordenamiento como de búsqueda, nos dimos cuenta una buena selección de pivotes para los algoritmos de ordenamientos y búsqueda (los que lo utilicen), pueden hacer la diferencia a la hora de optimizar estos algoritmos para sacarle mayor provecho a los mismos. A diferencia que, si no tomamos en cuenta esta selección de pivotes, se da el caso de que algoritmos como quicksort o búsqueda binaria, van a resultar menos eficiente que los algoritmos tradicionales como ordenamiento de burbuja o búsqueda secuencial.

Bibliografía

https://www.questionpro.com/es/software-de-analisis-de-texto-y-contenido.htmlhttps://www.questionpro.com/es/software-de-analisis-de-texto-y-contenido.html

http://lwh.free.fr/pages/algo/tri/tri_es.htm

https://www.ecured.cu/Algoritmo_de_búsqueda

https://es.wikipedia.org/wiki/Algoritmo_paralelo

https://es.wikipedia.org/wiki/Modelo_de_programaci%C3%B3n_paralela

https://www.hebergementwebs.com/tutorial-sobre-el-algoritmo-paralelo/algoritmo-paralelo-guia-rapida

https://www.cs.cinvestav.mx/tesisgraduados/2003/resumenMarcoOrtega.html

http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S2227-18992016000100006