Nombre: Sergio Guzmán Camacho **Matrícula:** 2-17-1817

Algoritmos Paralelos: Proyecto final

Simular una Carrera:

Crear una aplicación con los diferentes algoritmos de sorteos seleccionados en esta plantilla, la aplicación consiste en ordenar un solo arreglo con todos los algoritmos de búsquedas, ejecutarlos todos al mismo tiempo y calcular el tiempo de que algoritmo termino primero el proceso. Presentarlo en clase y realizarlo en grupos de 1 – 3 personas.

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.

2. **Objetivos**
   a. **Objetivo General**

      Interactuar con los distintos tipos de algoritmos de búsquedas que se han explicado en el transcurso del ciclo y observar su funcionamiento y 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**

      - Programar algoritmos de búsqueda capaces de ordenar arreglos desordenados.
      - Cronometrar el tiempo de ejecución de los algoritmos desarrollados.
      - 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

Un algoritmo 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.

Los algoritmos paralelos son importantes porque 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, logrando así la ejecución de varios flujos de instrucciones dentro del procesador.

4. Etapas de los Algoritmos paralelos

a. 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.

b. 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.

c. Agrupamiento

Las tareas y las 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.

d. Asignación

Cada tarea es asignada 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.

5. Técnicas Algorítmicas Paralelas

- ✓ **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.
- ✓ **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.

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 sub-dividir 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:

- ✓ Esqueletos algorítmicos
- ✓ Componentes
- ✓ Objetos distribuidos
- ✓ Invocación de Método de manera remota
- ✓ Workflows
- ✓ Máquina de Acceso paralelo Aleatorio
- ✓ Procesamiento de flujo
- ✓ Bulk synchronous parallelism

7. Algoritmos de Búsquedas y Ordenamiento (Adjuntar PSeudocódigo y código de cada uno)

a. Búsqueda Secuencial

Es el algoritmo de búsqueda más simple, menos eficiente y que menos precondiciones requiere: no requiere conocimientos sobre el conjunto de búsqueda ni acceso aleatorio. Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto.

Supondremos que los datos están almacenados en un array y se asumirá acceso secuencial.

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:

```
#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");
}
```

La complejidad del algoritmo medida en número de iteraciones en el mejor caso será 1, y se corresponderá con aquella situación en la cual el elemento a buscar está en la primera posición del array. El peor caso la complejidad será TAM y sucederá cuando el elemento buscar esté en la última posición del array. El promedio será ((TAM+1) / 2). El orden de complejidad es lineal (O *(TAM)*). Cada iteración necesita una suma, dos comparaciones y un AND lógico.

- ✓ **Búsqueda con centinela**

  Si tuviésemos la seguridad de que el elemento buscado está en el conjunto, nos evitaría controlar si se supera el límite superior. Para tener esa certeza, se almacena un elemento adicional (centinela), que coincidirá con el elemento buscado y que se situará en la última posición del array de datos. De esta forma se asegura que encontraremos el elemento buscado.

```
#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");
}
```

  Ahora sólo se realiza una suma y una única comparación (se ahorra una comparación y un AND). El algoritmo es más eficiente.

b. Búsqueda Binaria

Es un método muy eficiente, pero 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.

```
#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();
}
```

En el caso más favorable (el dato es el elemento mitad) se realiza 1 iteración. En el caso más desfavorable, el número de iteraciones es el menor entero K que verifica 2K >= TAM.

c. Algoritmo de Ordenamiento de la 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. 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.

```
#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);
}
```

d. Quick Sort

El método 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.

```
#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;
 }
```

e. Método de Inserción

El primer elemento del array (CB [0]) se considerado ordenado (la lista inicial consta de un elemento). A continuación, se inserta el segundo elemento (CB [1]) en la posición correcta (delante o detrás de CB [0]) dependiendo de que sea menor o mayor que CB [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.

Para colocar el dato en su lugar, se debe encontrar la posición que le corresponde en la parte ordenada y hacerle un hueco de forma que se pueda insertar. Para encontrar la posición se puede hacer una búsqueda secuencial desde el principio del conjunto hasta encontrar un elemento mayor que el dado. Para hacer el hueco hay que desplazar los elementos pertinentes una posición a la derecha.

```
#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);
}
```

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.
  - ➢ **Seudocódigo**

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.

- **Seudocódigo**

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 ini 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 (ini más fin) dividido entre 2
    h. Mientras ini 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. Ini es igual a mitad más 1
        iii. Declarar mitad igual a (ini 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 esta 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.
    - ➢ **Seudocódigo**

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.

- ✓ **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.
  - ➢ **Seudocódigo**
    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 mas 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**

```
100     go func() {
101
102         var a [100]int
103         var dato int
104         defer MedirTiempo(time.Now(), "Busqueda Secuencial")
105
106         LlenarArrayOrdenado(&a)
107
108         dato = 58 //Dato a buscar
109
110         i := 0
111
112         for (a[i] != dato) && (i < 100) {
113
114             i++
115
116             if a[i] == dato {
117
118                 fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, i)
119
120             } else {
121
122                 //fmt.Printf("\nEl Elemento no esta en el array")
123
124             }
125         }
126     }()
```

- **Búsqueda Binaria**

```go
go func() {

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

    ini := 0
    fin := 100 - 1
    var mitad, dato int

    dato = 58 //Dato a Buscar

    mitad = (ini + fin) / 2

    for (ini <= fin) && (a[mitad] != dato) {

        if dato < a[mitad] {

            fin = mitad - 1

        } else {

            ini = mitad + 1

        }

        mitad = (ini + fin) / 2
    }

    if dato == a[mitad] {

        fmt.Printf("\n\nEl dato %d, fue encontrado en la posicion: %d del arreglo!!\n", dato, mitad)

    } else {

        //fmt.Printf("\nEl Elemento no esta en el array")

    }
}()
```

- **Ordenamiento de Burbuja**

```
168     go func() {
169
170         var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}
171         var n int = len(a)
172         defer MedirTiempo(time.Now(), "Ordenamiento de Burbuja")
173         var e, i, auxiliar int
174
175         for e = 0; e < n; e++ {
176
177             for i = 0; i < n-1-e; i++ {
178
179                 if a[i] > a[i+1] {
180
181                     auxiliar = a[i+1]
182                     a[i+1] = a[i]
183                     a[i] = auxiliar
184
185                 }
186             }
187         }
188
189         fmt.Println(a)
190     }()
```

- **QuickSort**

```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
}
```

```go
    go func() {

        var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}
        var n int = len(a)
        defer MedirTiempo(time.Now(), "QuickSort")

        b := quicksort(a, 0, n-1)
        fmt.Println(b)

    }()
```

- **Método de inserción**

```
203     go func() {
204
205         var a = []int{15, 3, 8, 6, 18, 1, 20, 10, 37, 150}
206         var n int = len(a)
207         defer MedirTiempo(time.Now(), "Insercion")
208         var auxiliar int
209
210         for i := 1; i < n; i++ {
211
212             auxiliar = a[i]
213
214             for j := i - 1; j >= 0 && a[j] > auxiliar; j-- {
215
216                 a[j+1] = a[j]
217                 a[j] = auxiliar
218             }
219         }
220
221         fmt.Println(a)
222     }()
```

c. Link de Github y Ejecutable de la aplicación

https://github.com/DnaCyphers/Algoritmos_Paralelos.git

d. Resultados (Tiempo en terminar los ordenamientos y búsqueda de cada algoritmo)

- ✓ **Algoritmos de Búsqueda**
  - **Búsqueda Secuencial:** 518.1 µs (El más rápido)
  - **Búsqueda Binaria:** 1.0405 ms

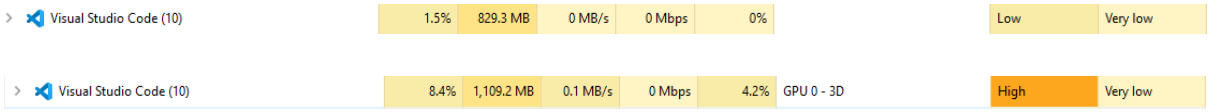
```
PS C:\Users\sergi\OneDrive\Universidad\Clases Virtuales\5. Ciclo 2-2021\5. Algoritmos Paralelos\3. Tercer Parcial> go run .\AlgoritmoFinal.go

Arreglo Ordenado ------------------------
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75
 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100]


El dato 58, fue encontrado en la posicion: 57 del arreglo!!
Algoritmo: Busqueda Binaria | Tiempo de ejecucion: 518.1µs


El dato 58, fue encontrado en la posicion: 57 del arreglo!!
Algoritmo: Busqueda Secuencial | Tiempo de ejecucion: 1.0405ms
```

- ✓ **Algoritmos de Ordenamiento**
  - **Ordenamiento de Burbuja:** 1.0939 ms
  - **QuickSort:** 521.4 µs
  - **Inserción:** 52.8 µs (El más rápido)

```
PS C:\Users\sergi\OneDrive\Universidad\Clases Virtuales\5. Ciclo 2-2021\5. Algoritmos Paralelos\3. Tercer Parcial> go run .\AlgoritmoFinal.go

Arreglo Desordenado ------------------------
[15 3 8 6 18 1 20 10 37 150]

[1 3 6 8 10 15 18 20 37 150]
Algoritmo: Insercion | Tiempo de ejecucion: 52.8µs

[1 3 6 8 10 15 18 20 37 150]
Algoritmo: QuickSort | Tiempo de ejecucion: 521.4µs

[1 3 6 8 10 15 18 20 37 150]
Algoritmo: Ordenamiento de Burbuja | Tiempo de ejecucion: 1.0939ms
```

e. ¿Qué tanta memoria se consumió este proceso?

Entre 829.3 MB y 1,109.2 MB de consumo en la memoria.



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

- ✓ **Algoritmos de Búsqueda**
  - **Búsqueda Secuencial**

    Con un tiempo de 518.1 µs, el algoritmo de Búsqueda Secuencial fue el más rápido en realizar la búsqueda del dato dentro del array. Este algoritmo fue el más rápido en esta ocasión debido a que el algoritmo de Búsqueda Binaria tuvo que realizar varias iteraciones hasta poder encontrar el dato deseado, ya que este dato no se encontraba en el medio del arreglo, por lo que tuvo que realizar varias veces las iteraciones. En cambio, el algoritmo de Búsqueda Secuencial, al ser tan simple y recorrer el arreglo hasta encontrar el dato comparando cada elemento, pues fue una tarea mas sencilla que el de la Búsqueda Binaria y por ese motivo tuvo par de microsegundos de ventaja con respecto al otro.

- ✓ **Algoritmos de Ordenamiento**
  - **Método de Inserción**

    Con un tiempo de 52.8 µs, el algoritmo de Método de inserción fue más rápido que sus competidores. El algoritmo QuickSort (que personalmente pensaba iba a ser el más rápido) le sigue en la fila con un tiempo de 521.4 µs (muy a la par hay que reconocer). En este caso el algoritmo de método de inserción fue más rápido, ya que, al no depender si su “pivote” o “auxiliar” quede en el centro o no, da igual si se le coloca en un extremo o en el centro del arreglo. Es decir, como el pivote fue un extremo del arreglo, era el peor caso de optimización para el algoritmo quicksort, ya que debía realizar mas comparaciones y era menos eficiente, en cambio, el método de inserción solo compara 2 valores.

10. Conclusión

En esta prueba, resultaron ganadores los algoritmos mas “simples” por decirlo de esa forma, ya que, los pivotes o auxiliares de los demás algoritmos quedaron en la posición menos eficiente del array (los extremos), por lo que los algoritmos debían trabajar más y realizar más iteraciones. Pudimos ver en esta prueba como funcionan los distintos algoritmos de búsqueda y ordenamiento y como funciona el paralelismo en la programación para optimizar procesos que nos encontramos día a día en nuestras jornadas laborales.

11. Bibliografías

- ✓ 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
- ✓ 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