

UTN - TUPaD 2025

# PROYECTO INTEGRADOR

## PROGRAMACIÓN I

**Algoritmos de búsqueda y ordenamiento**

**“Gestor de distancias en Python”**

Comisión: 5

Integrantes del grupo:

- Vergara, David ([david.vergara@tupad.utn.edu.ar](mailto:david.vergara@tupad.utn.edu.ar) )
- Vivas, Florencia ([florencia.vivas@tupad.utn.edu.ar](mailto:florencia.vivas@tupad.utn.edu.ar))

Profesora: Cinthia Rigoni

Tutor: Walter Pintos

Fecha de entrega: 09/06/2025

# INDICE

<b>Sección</b>	<b>Pag.</b>
<b>1. INTRODUCCIÓN.....</b>	<b>1</b>
<b>2. MARCO TEÓRICO.....</b>	<b>1</b>
<b>3. CASO PRÁCTICO.....</b>	<b>8</b>
<b>4. METODOLOGÍA UTILIZADA.....</b>	<b>13</b>
<b>5. RESULTADOS OBTENIDOS.....</b>	<b>14</b>
<b>6. CONCLUSIONES.....</b>	<b>16</b>
<b>7. BIBLIOGRAFÍA.....</b>	<b>17</b>
<b>8. ANEXOS.....</b>	<b>17</b>

# Proyecto Integrador

## Algoritmos de búsqueda y ordenamiento

### “Gestor de distancias en Python”

#### 1. INTRODUCCIÓN

El presente trabajo se centra en el estudio y aplicación de los algoritmos de búsqueda y ordenamiento, los cuales son fundamentales en el ámbito de la programación y la informática, ya que permiten organizar y acceder a la información de manera eficiente, facilitando su acceso y manipulación en diversas aplicaciones. Esto impacta directamente en el desarrollo de software y el rendimiento de los sistemas.

En programación, un buen método de ordenamiento puede reducir significativamente el tiempo de procesamiento, mientras que una estrategia de búsqueda adecuada permite el acceso a los datos de manera rápida y precisa.

La elección del tema para realizar el presente informe se debe a la comprensión de su importancia en la gestión de grandes volúmenes de datos. El estudio de estos algoritmos es esencial para cualquier programador que busque optimizar el rendimiento de sus programas.

El objetivo de este trabajo es analizar los principales algoritmos de búsqueda y ordenamiento, comparando su funcionamiento, eficiencia y aplicabilidad bajo un marco teórico y un caso práctico que permite visualizar las ventajas y desventajas de cada método brindando claridad para elegir la mejor opción según las necesidades específicas de cada problema.

#### 2. MARCO TEÓRICO

##### Búsqueda

Un algoritmo de búsqueda es un conjunto de instrucciones que están diseñadas para localizar un elemento con ciertas propiedades dentro de una estructura de datos. Es una tarea común

en muchas aplicaciones, como bases de datos, sistemas de archivos y algoritmos de inteligencia artificial.

Existen diferentes algoritmos de búsqueda, cada uno con sus propias ventajas y desventajas. Algunos son más comunes que otros por ejemplo los que serán desarrollados más adelante: Búsqueda lineal y Búsqueda binaria.

Los algoritmos de búsqueda son esenciales dentro de la programación ya que se utilizan en una amplia variedad de aplicaciones como por ejemplo en la búsqueda de palabras claves en un documento o la búsqueda de archivos con un nombre específico en un sistema de archivos.

Como se mencionó previamente, al comprender los diferentes algoritmos de búsqueda y cómo utilizarlos, se puede mejorar el rendimiento y la eficiencia de los programas.

### Métodos

1. **Búsqueda lineal:** Método simple que consiste en recorrer cada elemento de la lista hasta encontrar el buscado o llegar al final. Podría decirse que resulta intuitivo su funcionamiento.  
Es versátil ya puede aplicarse en listas ordenadas y desordenadas, pero a su vez resulta poco eficiente si es necesario aplicarla en listas largas ya que el tiempo de ejecución sería mayor al buscar un valor dentro de una lista larga.
2. **Búsqueda binaria:** Método eficiente que requiere que la lista esté ordenada (Puede utilizarse un método de ordenamiento que se describe en la siguiente sección). Funciona dividiendo la lista en mitades sucesivas lo cual disminuye significativamente el número de comparaciones necesarias.  
Primero, se toma la lista ordenada y se identifica el elemento central.  
Si el elemento buscado es igual al central, se da por finalizada la búsqueda.  
Si el elemento buscado es menor, se repite la búsqueda en la mitad izquierda.  
Si es mayor, se busca en la mitad derecha.  
Este proceso se repite hasta encontrar el elemento o determinar que no esta presente.  
Este método es más rápido de buscar en listas grandes.

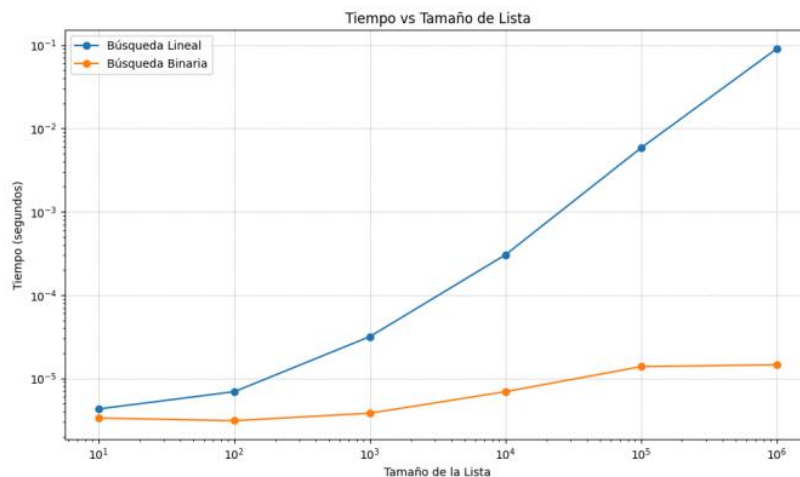


Imagen 1. Gráfico que ilustra el tiempo que tarda cada método en actuar en relación al tamaño de la lista.

El tamaño de la lista tiene un impacto significativo en el tiempo que tardan los algoritmos de búsqueda en encontrar el objetivo. Cuanto más grande sea la lista, más tiempo tardará el algoritmo en encontrar el elemento deseado. Esto se debe a que el algoritmo debe comprobar cada elemento de la lista hasta encontrar el que busca. Dicho esto, se procede a analizar la complejidad de los algoritmos.

La complejidad medida con  $O(n)$  es una forma de medir la eficiencia de un algoritmo. Representa el tiempo que tarda un algoritmo en ejecutarse en función del tamaño de la entrada.

La notación  $O(n)$  se utiliza para describir el peor caso de complejidad de tiempo de un algoritmo. Esto significa que el algoritmo nunca tardará más de  $O(n)$  tiempo en ejecutarse para cualquier entrada de tamaño  $n$ .

Los algoritmos de búsqueda lineal tienen un tiempo de ejecución de  $O(n)$ , lo que significa que el tiempo de búsqueda es directamente proporcional al tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará el doble de tiempo en encontrar el elemento deseado. Los algoritmos de búsqueda binaria tienen un tiempo de ejecución de  $O(\log n)$ , lo que significa que el tiempo de búsqueda aumenta logarítmicamente con el tamaño de la lista. Esto significa que, si la lista tiene el doble de elementos, el algoritmo tardará aproximadamente el mismo tiempo en encontrar el elemento deseado.

En general, el tamaño de la lista es un factor importante a tener en cuenta al elegir un algoritmo de búsqueda. Si la lista es pequeña, es probable que la búsqueda lineal sea más eficiente. Sin embargo, si la lista es grande, es probable que la búsqueda binaria sea más eficiente.

### Complejidad temporal de ambos métodos:

- **Búsqueda lineal:**

- ☐ Peor caso:  $O(n)$ , donde  $n$  es el número de elementos en la lista. Esto ocurre cuando el elemento buscado está al final de la lista o no está presente.
- ☐ Mejor caso:  $O(1)$ , cuando el elemento buscado es el primero de la lista.
- ☐ Caso promedio:  $O(n)$ , porque en promedio se recorren la mitad de los elementos.

- **Búsqueda binaria:**

- ☐ Peor caso:  $O(\log n)$ , donde  $n$  es el número de elementos en la lista. Esto ocurre cuando el elemento no está presente o está en una de las divisiones finales.
- ☐ Mejor caso:  $O(1)$ , cuando el elemento buscado está justo en el centro de la lista.
- ☐ Caso promedio:  $O(\log n)$ , porque el algoritmo divide la lista en mitades en cada iteración.

## Ordenamiento

El ordenamiento es un proceso fundamental en programación que consiste en organizar un conjunto de elementos siguiendo un criterio específico ya sea ascendente o descendente. Esta tarea es esencial para muchas aplicaciones, desde la búsqueda de información hasta la visualización de datos.

Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Los beneficios principales que se pueden mencionar son:

- **Búsqueda más eficiente:** Al estar ordenados los datos, es mucho más fácil buscar un elemento específico porque bajo estas condiciones puede utilizarse la búsqueda binaria, que como se mencionó anteriormente es un algoritmo de búsqueda más eficiente que la búsqueda lineal.

- Análisis de datos más fácil: Los datos ordenados pueden ser analizados fácilmente para identificar patrones y tendencias.
- Operaciones más rápidas: Muchas operaciones, como fusionar dos conjuntos de datos o eliminar elementos duplicados, se pueden realizar de manera rápida y eficiente en datos ordenados.

Existen muchos algoritmos de ordenamiento, cada uno presenta sus propias ventajas y desventajas. Algunos más comunes son:

- ★ Ordenamiento de burbuja: Es un algoritmo de ordenamiento simple y fácil de implementar. Consiste en ir comparando cada elemento de la lista con el siguiente y luego intercambiar los elementos si están en el orden incorrecto. Es eficiente para listas pequeñas, pero cuando el tamaño de datos crece, la eficiencia decae. Respecto a su complejidad temporal, el peor caso sería  $O(n^2)$  donde  $n$  es el número de elementos en la lista. Esto ocurre cuando la lista está en orden inverso. El mejor caso sería  $O(n)$  es decir cuando la lista ya está ordenada. Si hablamos del caso promedio debemos recalcar que es  $O(n^2)$ .
- ★ Ordenamiento por selección: Algoritmo de ordenamiento simple que funciona encontrando el elemento más pequeño de la lista no ordenada y colocándolo en la primera posición. Este proceso se repite hasta que todos los elementos de la lista estén ordenados. Es eficiente para listas pequeñas y casi ordenadas, ya que realiza menos comparaciones en ese caso. Si mencionamos su complejidad temporal, podemos decir que su peor caso es  $O(n^2)$ , ya que siempre realiza comparaciones para encontrar el mínimo. El mejor caso, también es  $O(n^2)$ , ya que incluso si la lista está ordenada igualmente tiene que recorrer toda la lista buscando el mínimo. Su caso promedio es  $O(n^2)$ . Podemos concluir que de los métodos para ordenar listas pequeñas que veremos este es el que demuestra un peor desempeño.
- ★ Ordenamiento por inserción: El algoritmo construye una parte ordenada de la lista uno por uno, insertando cada elemento en su posición correcta. Eficiente para listas pequeñas y casi ordenadas. Respecto a su complejidad temporal, su peor caso es  $O(n^2)$ , cuando la lista está en el orden inverso.  $O(n)$  es su mejor caso que se da cuando la lista ya está ordenada pero su caso promedio es  $O(n^2)$ .
- ★ Ordenamiento quicksort: Algoritmo eficiente que utiliza la técnica de “divide y vencerás”. Funciona seleccionando un elemento pivote y partiendo la lista en dos sublistas: una con

los elementos menores que el elemento pivote y otra con los elementos mayores. Luego va aplicando recursivamente la función en cada sublista.

Generalmente es más eficiente que inserción y selección para listas grandes.

Respecto a su complejidad temporal podemos decir que su peor caso es  $O(n^2)$ , cuando el pivote seleccionado es el menor o mayor elemento de cada sublista. El mejor caso es  $O(n \log n)$ , cuando el pivote divide la lista en dos partes aproximadamente iguales. Este último representa también su caso promedio

Algoritmo	Mejor Caso	Peor Caso	Caso Promedio	Eficiencia	Uso Recomendado
Burbuja	$O(n)$	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas o casi ordenadas
Selección	$O(n^2)$	$O(n^2)$	$O(n^2)$	Baja	Listas pequeñas
Inserción	$O(n)$	$O(n^2)$	$O(n^2)$	Media	Listas pequeñas o casi ordenadas
Quicksort	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	Alta	Listas grandes (evitar peor caso)

Imagen 2. Cuadro comparativo entre métodos de ordenamiento.

En resumen, los algoritmos de búsqueda y ordenamiento constituyen pilares fundamentales en el desarrollo de software, proporcionando soluciones eficientes, escalables y precisas para la gestión de información, contribuyendo así a la creación de programas más rápidos, organizados y confiables

### 3. CASO PRÁCTICO

#### Descripción:

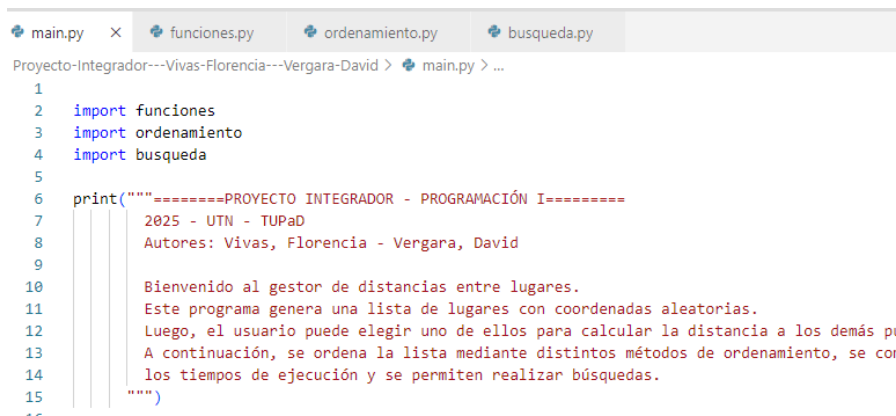
Para aplicar e implementar los conceptos expuestos en el marco teórico y, con el fin de analizar la eficiencia de cada algoritmo en tiempos de ejecución, se desarrolló un programa en Python que simula un gestor de distancias entre distintos puntos. Cada lugar, generado aleatoriamente, es representado por un nombre ("Lugar n"), y un par de coordenadas aleatorias "x" e "y" dentro de un plano cartesiano. Luego, el usuario elige uno de esos puntos como su ubicación y se calculan las distancias al resto de los puntos. Estas distancias se ordenan mediante distintos algoritmos clásicos (burbuja, quick sort, selección e inserción), permitiendo observar y comparar sus tiempos de ejecución. Finalmente, se permite realizar búsquedas dentro de listas ordenadas (búsqueda binaria y lineal), y sin ordenar (solamente lineal).

#### Funcionamiento general



El programa está dividido en módulos (main.py, funciones.py, ordenamiento.py y búsqueda.py), para organizar el código según su propósito y para una visualización clara e individual de los algoritmos de búsqueda y ordenamiento.

- main.py: controla el flujo principal del programa de manera secuencial, llama funciones y gestiona parte de la interacción con el usuario.
- funciones.py: contiene funciones adicionales, como la generación de lugares y el cálculo de distancias.
- ordenamiento.py: contiene solamente los algoritmos de ordenamiento (burbuja, selección, inserción y quicksort).
- busqueda.py: contiene solo las funciones de búsqueda lineal y binaria.



```

1
2 import funciones
3 import ordenamiento
4 import busqueda
5
6 print("""=====PROYECTO INTEGRADOR - PROGRAMACIÓN I=====
7         2025 - UTN - TUPaD
8         Autores: Vivas, Florencia - Vergara, David
9
10        Bienvenido al gestor de distancias entre lugares.
11        Este programa genera una lista de lugares con coordenadas aleatorias.
12        Luego, el usuario puede elegir uno de ellos para calcular la distancia a los demás pu
13        A continuación, se ordena la lista mediante distintos métodos de ordenamiento, se com
14        los tiempos de ejecución y se permiten realizar búsquedas.
15        """)

```

En todos los casos de ingreso de datos por el usuario, se agregaron métodos de validación de datos y mensajes de error.

### 3.1 Generación de lugares y coordenadas

Luego de solicitar al usuario la cantidad de lugares que desea generar, se genera esa cantidad de lugares, cada uno con coordenadas aleatorias dentro de un plano cartesiano de 1000000 x 1000000. Esto se realiza mediante una función que retorna una lista de diccionarios, cada diccionario representando un lugar con sus coordenadas.

```

#Genera una lista de lugares con coordenadas aleatorias
def generar_destinos(n):
    destinos = []
    for i in range(1, n+1):
        destino = {
            "Lugar" : f"Lugar {i}",
            "x" : random.randint(0, 1000000),
            "y" : random.randint(0, 1000000)
        }
        destinos.append(destino) #Se agrega cada lugar generado con sus coordenadas a la lista
    return destinos

```

### 3.2 Selección del lugar base (ubicación ficticia del usuario)

El usuario selecciona un lugar entre los generados como su ubicación actual. Este se toma como referencia para calcular la distancia a los demás lugares.

```
#Devuelve el elemento elegido por el usuario (Lugar y coordenadas)
def elegir_base(destinos, n):
    base = 0
    print(f"\nElija su ubicación (número entre 1 y {len(destinos)}) para calcular la distancia a los demas puntos: \n")
    while base <= 0 or base > n :
        try:
            base = int(input("\nLugar: "))
            if base <= 0 or base > n :
                print("Valor incorrecto.")
            else:
                print(f"Lugar elegido : {base}(x : {destinos[base -1]['x']}, y :{destinos[base -1]['y']})")
        except ValueError:
            print("Por favor, ingrese un número válido.")

    return destinos[base -1]
```

### 3.3 Cálculo de distancias

Se utiliza el teorema de Pitágoras para calcular la distancia entre el lugar base y todos los demás puntos. El resultado es una lista de tuplas (lugar, distancia).

```
#Cálculo de las distancias desde cada punto al elegido por el usuario (con teorema de pitágoras)
def calcular_distancia(destinos, base, n):
    distancias = []
    for i in destinos:
        x2 = i["x"]
        y2 = i["y"]
        x1 = base["x"]
        y1 = base["y"]
        distancia = round((math.sqrt((x2-x1)**2 + (y2-y1)**2)), 2)
        distancias.append((i["Lugar"], distancia)) #Va formando una lista de tuplas con el nombre del lugar y la distancia calculada
    if n < 20:
        input("Presione Enter para ver las distancias desde este lugar.")
        for i in distancias:
            print(f'{i[0]}, Distancia: {i[1]}') # Muestra el lugar y la distancia calculada desde el lugar elegido
    else : print("Se guardaron las distancias, pero la lista es demasiado grande para mostrarse")
    return distancias #Devuelve la lista de tuplas
```

### 3.4 Ordenamiento de distancias

Se implementaron cuatro fórmulas en el módulo ordenamiento.py, que corresponden a los cuatro métodos de ordenamiento mencionados (bubble sort, quick sort, ordenamiento por selección y por inserción). Cada uno ordena las distancias en orden ascendente y mide el tiempo de ejecución, permitiendo compararlos entre sí.

```
#Quick Sort (Ordenamiento rápido):
def ordenamiento_quick(distancias):
    tiempo_inicio = time.time() #Marca el tiempo de inicio
    def quick_sort(lista):
        if len(lista) <= 1: #Si la lista tiene un solo elemento o está vacía, ya está ordenada
            return lista
        else:
            pivot = lista[len(lista) // 2] #Elige el pivote como el elemento del medio
            menores = [x for x in lista if x[1] < pivot[1]] #Elementos menores que el pivote
            iguales = [x for x in lista if x[1] == pivot[1]] #Elementos iguales al pivote
            mayores = [x for x in lista if x[1] > pivot[1]] #Elementos mayores que el pivote
            return quick_sort(menores) + iguales + quick_sort(mayores) #Recursión para ordenar los sublistas
    resultado = quick_sort(distancias)
    tiempo_fin = time.time() #Marca el tiempo final
    tiempo_total = tiempo_fin - tiempo_inicio #Calcula el tiempo total de ejecucion.
    print(f"Tiempo del ordenamiento rápido: {tiempo_total}")
    return resultado #Devuelve la lista ordenada por quick sort
```

```
#Bubble Sort (Ordenamiento por burbuja):
def ordenamiento_bubble(distancias):
    tiempo_inicio = time.time() #Marca el tiempo de inicio
    n = len(distancias)
    for i in range(n): #Recorre la lista n veces
        for j in range(0, n - i - 1): #Compara cada elemento con el siguiente
            if distancias[j][1] > distancias[j + 1][1]: #Condicion de ordenamiento: Si el elemento actual es mayor que el siguiente, los intercambia.
                distancias[j], distancias[j + 1] = distancias[j + 1], distancias[j]
    tiempo_fin = time.time() #Marca el tiempo final
    tiempo_total = tiempo_fin - tiempo_inicio #Calcula el tiempo total de ejecucion
    print(f"Tiempo del ordenamiento por burbuja: {tiempo_total}")
    return distancias #Devuelve la lista ordenada por bubble sort

#Selection Sort (Ordenamiento por selección):
def ordenamiento_seleccion(distancias):
    tiempo_inicio = time.time() # Marca el tiempo de inicio
    n = len(distancias)
    for i in range(n):
        # Encuentra el índice del elemento mínimo en la parte no ordenada de la lista
        minimo = i
        for j in range(i + 1, n):
            if distancias[j][1] < distancias[minimo][1]:
                minimo = j
        # Intercambia el mínimo encontrado con el primer elemento
        distancias[i], distancias[minimo] = distancias[minimo], distancias[i]
    tiempo_fin = time.time() # Marca el tiempo de fin
    tiempo_total = tiempo_fin - tiempo_inicio # Calcula el tiempo total de ejecución
    print(f"Tiempo del ordenamiento por selección: {tiempo_total}")
    return distancias #Devuelve la lista ordenada por selección

#Insertion Sort (Ordenamiento por inserción):
def ordenamiento_insercion(distancias):
    tiempo_inicio = time.time() # Marca el tiempo de inicio
    for i in range(1, len(distancias)):
        clave = distancias[i]
        j = i - 1
        # Mueve los elementos de distancias[0..i-1], que son mayores que clave, a una posición adelante de su posición actual
        while j >= 0 and clave[1] < distancias[j][1]:
            distancias[j + 1] = distancias[j]
            j -= 1
        distancias[j + 1] = clave
    tiempo_fin = time.time() # Marca el tiempo de fin
    tiempo_total = tiempo_fin - tiempo_inicio
    print(f"Tiempo del ordenamiento por inserción: {tiempo_total}")
    return distancias #Devuelve la lista ordenada por inserción
```

### 3.5 Búsqueda por lugar (Lineal)

El usuario puede ingresar el número de un lugar (por ejemplo, "3" para "Lugar 3") y obtener su distancia al punto base. Esto se realiza mediante búsqueda lineal, por nombre, sobre la lista de distancias. Además, se muestra su ubicación en el ranking de distancias, siendo 1 el lugar más cercano.

```
def busqueda_lineal(distancias, lugar):
    tiempo_inicio = time.time()
    cont = 0
    for i in distancias:
        cont += 1
        if i[0].lower() == f"lugar {lugar}":
            tiempo_fin = time.time()
            tiempo_total = tiempo_fin - tiempo_inicio
            print(f"Distancia hasta {i[0]}: {i[1]}")
            print(f"Posición en el ranking de distancias: {cont}")
            print(f"Tiempo de búsqueda lineal: {tiempo_total}")
            return i
    tiempo_fin = time.time()
    tiempo_total = tiempo_fin - tiempo_inicio
    print(f"Tiempo de búsqueda lineal: {tiempo_total}")
    print(f"{lugar} no se encuentra en la lista.")
    return None
```

### 3.6 Búsqueda por distancia (lineal y binaria)

También es posible ingresar una distancia deseada, y el programa buscará el lugar más cercano a esa distancia. Para fines comparativos, en este caso se usa la misma lista ordenada como argumento de ambas funciones. se muestra el lugar más cercano, sus coordenadas, su distancia real, la diferencia con la distancia objetivo y el tiempo de búsqueda en ambos casos para efectuar un análisis comparativo.

```
#Búsqueda binaria de distancias. Devuelve el lugar mas cercano a la distancia elegida
def binaria_mas_cercano(lista_ordenada, objetivo, destinos):
    tiempo_inicio = time.time() #Inicia el temporizador
    #Índices para el inicio y fin de la búsqueda
    izquierda = 1 # Empezamos desde el segundo elemento, para exceptuar al lugar propio
    derecha = len(lista_ordenada) - 1
    mejor = lista_ordenada[1] #inicia con el primer elemento de la lista
    mejor_dif = abs(lista_ordenada[1][1] - objetivo) #diferencia inicial
    #Comienza la búsqueda binaria
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2 #Se calcula el punto medio
        actual = lista_ordenada[medio] #Elemento central
        dif = abs(actual[1] - objetivo) #Diferencia con el objetivo del usuario
        if dif < mejor_dif: #Si la diferencia es menor, se actualiza el valor
            mejor = actual
            mejor_dif = dif
        #Si el valor es menor al objetivo, buscamos en la mitad derecha. Sino en la izquierda
        if actual[1] < objetivo:
            izquierda = medio + 1
        elif actual[1] > objetivo:
            derecha = medio - 1
        else:
            # Coincidencia exacta: salimos del bucle
            mejor = actual
    # Fin del cronómetro
    tiempo_total = time.time() - tiempo_inicio
    # Mostrar resultados
    nombre, distancia = mejor
    print(f"\n\n==BUSQUEDA BINARIA==\nResultado más cercano encontrado:")
    print(f"Lugar: {nombre}")
    for destino in destinos:
        if destino['Lugar'] == nombre:
            print(f"Coordenadas: x = {destino['x']}, y = {destino['y']}")
    print(f"Distancia encontrada: {distancia:.2f}")
    print(f"Diferencia con la distancia buscada: {abs(distancia - objetivo):.2f}")
    print(f"Tiempo de búsqueda binaria (más cercano): {tiempo_total:.6f} segundos\n")
    return mejor

#Búsqueda lineal de distancias. Devuelve el lugar mas cercano a la distancia elegida
def lineal_mas_cercano(lista, base, objetivo, destinos):
    lugar_propio = base["Lugar"].strip().lower() # Obtenemos el nombre del lugar propio
    tiempo_inicio = time.time() # Inicia el cronómetro
    mejor = None
    mejor_dif = float('inf')
    # Recorre la lista excluyendo el lugar propio por nombre
    for lugar, distancia in lista:
        if lugar.strip().lower() == lugar_propio:
            continue
        dif = abs(distancia - objetivo)
        if dif < mejor_dif:
            mejor = (lugar, distancia)
            mejor_dif = dif
    tiempo_total = time.time() - tiempo_inicio # Fin del cronómetro
    if mejor:
        nombre, distancia = mejor
        print(f"\n\n==BUSQUEDA LINEAL==\nResultado más cercano encontrado:")
        print(f"Lugar: {nombre}")
        for destino in destinos:
            if destino['Lugar'] == nombre:
                print(f"Coordenadas: x = {destino['x']}, y = {destino['y']}")
        print(f"Distancia encontrada: {distancia:.2f}")
        print(f"Diferencia con la distancia buscada: {abs(distancia - objetivo):.2f}")
        print(f"Tiempo de búsqueda lineal (más cercano): {tiempo_total:.6f} segundos\n")
    else:
        print("\nNo se encontró ningún lugar distinto al propio.\n")
    return mejor
```

El programa finaliza con un mensaje de despedida y agradecimiento al usuario.

(Ver código completo en **Anexos**)

#### 4. METODOLOGÍA UTILIZADA

Los primeros pasos en el desarrollo del presente trabajo consistieron en realizar una investigación teórica sobre algoritmos de búsqueda y ordenamiento, enfocándose en su funcionamiento, casos de uso, eficiencia y complejidad. Esta etapa fue necesaria para comprender cada técnica y tomar decisiones informadas a la hora de implementarlas y compararlas.

También se pusieron en práctica la mayoría de los conceptos vistos durante la cursada, tales como estructuras de datos (listas, tuplas, diccionarios), estructuras de control (condicionales, bucles), definición y uso de funciones, modularización del código y trabajo colaborativo, incluyendo el presente concepto de algoritmos de búsqueda y ordenamiento.

Para el desarrollo del gestor de distancias se siguió un enfoque modular, comenzando por la identificación de los requisitos básicos del programa y la definición de las funciones principales aún sin desarrollar en esta instancia: generación de lugares con coordenadas, cálculo de distancias, ordenamiento de resultados y búsqueda de información. Se comenzó diseñando la estructura general del programa, dividiéndolo en módulos para mejorar la organización del código y facilitar futuras ampliaciones o modificaciones, así como también facilitar el trabajo colaborativo. Esta separación permitió una mejor organización y visualización del flujo principal del programa y de las funciones.

Luego de esta organización y definición de funciones, se desarrollaron secuencialmente las que iban resultando necesarias a medida que se avanzaba en la ejecución del programa. Se desarrollaron en primer lugar las funciones auxiliares para dar inicio a la creación y almacenamiento de los primeros datos. Input inicial, generación de lugares, cálculo de distancias). Posteriormente se programaron los cuatro algoritmos de ordenamiento clásico ya explicados, para ordenar las distancias de forma ascendente y midiendo en el proceso su eficiencia en términos de tiempo de ejecución. Con listas pequeñas, se decidió mostrar por pantalla el resultado del ordenamiento de las cuatro técnicas, para verificar que coincidan. Finalmente se desarrollaron las funciones de búsqueda lineal y binaria.

En el proceso, se realizaron pruebas constantes del funcionamiento del programa, corrigiendo errores, bugs, y mejorando gradualmente la lógica y modularidad del código.

Se tomaron capturas del código y del funcionamiento del programa, incluyendo las salidas por pantalla.

## 5. RESULTADOS OBTENIDOS

A partir de la ejecución del programa, se obtuvieron los siguientes resultados:

Generación de datos aleatorios y cálculo de distancias:

El sistema generó correctamente la cantidad de lugares elegida por el usuario, cada uno con un nombre único y coordenadas “x” e “y” dentro del plano simulado. Las coordenadas variaron de forma aleatoria, lo cual permitió obtener datos variados y realistas:

```
=====PROYECTO INTEGRADOR - PROGRAMACIÓN I=====
2025 - UTN - TUPaD
Autores: Vivas, Florencia - Vergara, David

Bienvenido al gestor de distancias entre lugares.
Este programa genera una lista de lugares con coordenadas aleatorias.
Luego, el usuario puede elegir uno de ellos para calcular la distancia a los demás puntos.
A continuación, se ordena la lista mediante distintos métodos de ordenamiento, se comparan
los tiempos de ejecución y se permiten realizar búsquedas.

Ingrese la cantidad de lugares a generar:
10
Lugar 1, Coordenadas: (687022, 729369)
Lugar 2, Coordenadas: (757984, 332390)
Lugar 3, Coordenadas: (265500, 489688)
Lugar 4, Coordenadas: (124679, 622586)
Lugar 5, Coordenadas: (181358, 436689)
Lugar 6, Coordenadas: (629804, 874006)
Lugar 7, Coordenadas: (907733, 468780)
Lugar 8, Coordenadas: (650035, 77929)
Lugar 9, Coordenadas: (89700, 181367)
Lugar 10, Coordenadas: (376123, 825853)
```

Las distancias se calcularon correctamente y se mostraron por pantalla en los casos de cantidades menores a 15 lugares generados:

Elija su ubicación (número entre 1 y 10) para calcular la distancia a los demás puntos:

```
Lugar: 5
Lugar elegido : 5(x : 181358, y :436689)
Presione Enter para ver las distancias desde este lugar.
Lugar 1, Distancia: 584258.22
Lugar 2, Distancia: 585982.79
Lugar 3, Distancia: 99442.3
Lugar 4, Distancia: 194345.58
Lugar 5, Distancia: 0.0
Lugar 6, Distancia: 626378.46
Lugar 7, Distancia: 727083.54
Lugar 8, Distancia: 590226.12
Lugar 9, Distancia: 271275.71
Lugar 10, Distancia: 435180.45
Presione Enter para continuar a la sección de ordenamiento.
```

Ordenamiento de distancias

En listas pequeñas, los cuatro métodos de ordenamiento funcionaron correctamente, evidenciándose escasas diferencias en los tiempos de ejecución. Sin embargo, a medida que se incrementaba la cantidad de datos, de evidenció la diferencia en la eficiencia de cada técnica:

- Quick sort fue el más eficiente en todos los casos.
- Bubble sort fue considerablemente más lento.
- Inserción y selección tuvieron un desempeño intermedio en las pruebas realizadas, volviéndose prácticamente inutilizables al superar los 30 mil datos.

A continuación se muestran los tiempos de ejecución en conjuntos de 10 y de 10000 elementos respectivamente:

```
=== Ordenamiento de distancias ===
Tiempo del ordenamiento por burbuja: 2.0742416381835938e-05
Tiempo del ordenamiento rápido: 4.649162292480469e-05
Tiempo del ordenamiento por selección: 1.6450881958007812e-05
Tiempo del ordenamiento por inserción: 1.2874603271484375e-05
Presione Enter para continuar.
```

```
=== Ordenamiento de distancias ===
Tiempo del ordenamiento por burbuja: 8.075176477432251
Tiempo del ordenamiento rápido: 0.04346013069152832
Tiempo del ordenamiento por selección: 3.962531328201294
Tiempo del ordenamiento por inserción: 4.080087661743164
Presione Enter para continuar a la sección de búsqueda.
```

## Búsqueda

La búsqueda lineal mostró tener un mejor desempeño que la binaria en listas pequeñas, incrementándose proporcionalmente el tiempo al aumentar la cantidad de datos a procesar. La búsqueda binaria superó significativamente la eficiencia de la búsqueda lineal en listas mayores. Se muestran los resultados con 10 y con 10000 elementos respectivamente:

```
==BUSQUEDA BINARIA==
Resultado más cercano encontrado:
Lugar: Lugar 3
Coordenadas: x = 265500, y = 489688
Distancia encontrada: 99442.30
Diferencia con la distancia buscada: 99437.30
Tiempo de búsqueda binaria (más cercano): 0.000015 segundos
```

```
==BUSQUEDA LINEAL==
Resultado más cercano encontrado:
Lugar: Lugar 3
Coordenadas: x = 265500, y = 489688
Distancia encontrada: 99442.30
Diferencia con la distancia buscada: 99437.30
Tiempo de búsqueda lineal (más cercano): 0.000014 segundos
```

```
==BUSQUEDA BINARIA==  
Resultado más cercano encontrado:  
Lugar: Lugar 321  
Coordenadas: x = 310411, y = 437492  
Distancia encontrada: 568457.39  
Diferencia con la distancia buscada: 88.61  
Tiempo de búsqueda binaria (más cercano): 0.000038 segundos
```

```
==BUSQUEDA LINEAL==  
Resultado más cercano encontrado:  
Lugar: Lugar 321  
Coordenadas: x = 310411, y = 437492  
Distancia encontrada: 568457.39  
Diferencia con la distancia buscada: 88.61  
Tiempo de búsqueda lineal (más cercano): 0.006167 segundos
```

## 6. CONCLUSIONES

A lo largo de este trabajo se pudo integrar y aplicar gran parte de los contenidos abordados en la cursada de Programación I. Desde estructuras básicas como listas, diccionarios y condicionales, hasta el uso de funciones, modularización del código, y la implementación de algoritmos clásicos de ordenamiento y búsqueda.

El desarrollo del gestor de distancias permitió no solo ejercitar habilidades de programación, sino también reflexionar sobre la eficiencia de distintas soluciones ante un mismo problema. La comparación de tiempos de ejecución entre algoritmos como burbuja, quicksort o búsqueda binaria ayudó a visualizar cómo la elección del algoritmo adecuado puede impactar significativamente en el rendimiento de una aplicación.

Además, la organización del código en módulos separados facilitó su mantenimiento y lectura, algo fundamental en proyectos reales. También se valoró el uso de herramientas externas para enriquecer la presentación del trabajo, como capturas de pantalla, informes y una representación clara en video.

En resumen, este proyecto no solo cumplió con los objetivos académicos propuestos, sino que también sirvió como una experiencia concreta de desarrollo y documentación de software, sentando una base sólida para futuros desafíos más complejos.



## 7. BIBLIOGRAFÍA

Catedra de programación I (2025). Tema: búsqueda y ordenamiento. Archivo PDF: Búsqueda y Ordenamiento en Programación. Aula Virtual, TUPaD, Universidad Tecnológica Nacional.

Catedra de programación I (2025). Notebook Búsqueda y ordenamiento. Aula Virtual, TUPaD, Universidad Tecnológica Nacional.  
<https://colab.research.google.com/drive/1KVqjJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing>

Catedra de programación I (2025). Video de youtube sobre búsqueda. Aula Virtual, TUPaD, Universidad Tecnológica Nacional.  
[https://www.youtube.com/watchv=gJIQTq80Ilg&ab\\_channel=Tecnicatura](https://www.youtube.com/watchv=gJIQTq80Ilg&ab_channel=Tecnicatura)

Catedra de programación I (2025). Video de youtube sobre ordenamiento. Aula Virtual, TUPaD, Universidad Tecnológica Nacional.  
[https://www.youtube.com/watchv=xntUhrhtLaw&ab\\_channel=Tecnicatura](https://www.youtube.com/watchv=xntUhrhtLaw&ab_channel=Tecnicatura)

## 8. ANEXOS

### Código completo del programa:

- **main.py**

```
import funciones
import ordenamiento
import busqueda

print("""=====PROYECTO INTEGRADOR - PROGRAMACIÓN I=====
      2025 - UTN - TUPaD
      Autores: Vivas, Florencia - Vergara, David

      Bienvenido al gestor de distancias entre lugares.
      Este programa genera una lista de lugares con coordenadas aleatorias.
      Luego, el usuario puede elegir uno de ellos para calcular la distancia
      a los demás puntos.
      A continuación, se ordena la lista mediante distintos métodos de
      ordenamiento, se comparan
      los tiempos de ejecución y se permiten realizar búsquedas.
      """)

n = funciones.inicio() # Solicita al usuario la cantidad de lugares a generar
```

```

destinos = funciones.generar_destinos(n)    #Genera lugares con coordenadas
"x" e "y" aleatorias

funciones.mostrar_lista(n, destinos)    # Si la lista es pequeña, muestra cada
lugar con sus coordenadas

base = funciones.elegir_base(destinos, n)  # Elección del lugar (base) para
calcular distancias a los demás puntos

distancias = funciones.calcular_distancia(destinos, base, n)  # Cálculo de
distancias desde el lugar elegido a los demás puntos

input("Presione Enter para continuar a la sección de ordenamiento.")

# =====Ordenamiento de la lista de distancias=====

print("\n=== Ordenamiento de distancias ===")

#Bubble Sort (Ordenamiento por burbuja):
distancias_bubble = ordenamiento.ordenamiento_bubble(distancias.copy())
#Ordenamiento por Burbuja
#Quick Sort (ordenamiento rapido):
distancias_quick = ordenamiento.ordenamiento_quick(distancias.copy())
#Selection Sort (Ordenamiento por seleccion):
distancias_seleccion =
ordenamiento.ordenamiento_seleccion(distancias.copy())  #Selection Sort
(Ordenamiento por selección)
#Insertion Sort(Ordenamiento por insercion):
distancias_insercion =
ordenamiento.ordenamiento_insercion(distancias.copy())  #Insertion Sort
(Ordenamiento por inserción)

funciones.mostrar_listas_ordenadas(n,distancias_bubble, distancias_quick,
distancias_seleccion, distancias_insercion)  #Muestra las listas ordenadas si
son chicas

input("Presione Enter para continuar a la sección de búsqueda.")

# =====Búsqueda en la lista de distancias=====
#Búsqueda de lugares en la lista. Se muestra la distancia del lugar
print("""\n=== Búsqueda de lugares ===
Ingrese el lugar que desea buscar (Se mostrará la distancia hasta ese lugar, y
la posición que ocupa en el ranking de distancias)\n
""")
lugar = input("Lugar: ")
busqueda.busqueda_lineal(distancias_insercion, lugar)

#Búsqueda lineal y binaria en lista ordenada

```

```

objetivo = funciones.obtener_objetivo()
busqueda.binaria_mas_cercano(distancias_insercion, objetivo, destinos)
busqueda.lineal_mas_cercano(distancias_insercion, base, objetivo, destinos)
input("Presione Enter para finalizar el programa.")
print("Gracias por utilizar el gestor de distancias de Florencia Vivas y David Vergara. ¡Hasta pronto!")

```

### funciones.py

```

import random
import math

def inicio():
    n = 0
    while n <= 0: # Manejo de errores de datos de entrada
        try:
            n = int(input("Ingrese la cantidad de lugares a generar: \n"))
            if n <= 0:
                print("Ingrese un número positivo.")
        except ValueError:
            print("Por favor, ingrese un número positivo válido.")
    return n

#Si la lista es chica, se muestra por pantalla
def mostrar_lista(n, destinos):
    if n < 20:
        for i in destinos:
            print(f"{i['Lugar']}, Coordenadas: ({i['x']}, {i['y']})")
    else:
        print("La lista fue guardada pero es demasiado extensa para mostrarla por pantalla")

#Genera una lista de lugares con coordenadas aleatorias
def generar_destinos(n):
    destinos = []
    for i in range(1, n+1):
        destino = {
            "Lugar" : f"Lugar {i}",
            "x" : random.randint(0, 1000000),
            "y" : random.randint(0, 1000000)
        }
        destinos.append(destino) #Se agrega cada lugar generado con sus coordenadas a la lista destinos
    return destinos

#Devuelve el elemento elegido por el usuario (Lugar y coordenadas)
def elegir_base(destinos, n):
    base = 0

```

```

    print(f"\nElija su ubicación (número entre 1 y {len(destinos)}) para
    calcular la distancia a los demas puntos: \n")
    while base <= 0 or base > n :
        try:
            base = int(input("\nLugar: "))
            if base <= 0 or base > n :
                print("Valor incorrecto.")
            else:
                print(f"Lugar elegido : {base}(x : {destinos[base -1]['x']}, y
                :{destinos[base -1]['y']})")
            except ValueError:
                print("Por favor, ingrese un número válido.")

    return destinos[base -1]

#Cálculo de las distancias desde cada punto al elegido por el usuario (con
teorema de pitágoras)
def calcular_distancia(destinos, base, n):
    distancias = []
    for i in destinos:
        x2 = i["x"]
        y2 = i["y"]
        x1 = base["x"]
        y1 = base["y"]
        distancia = round((math.sqrt((x2-x1)**2 + (y2-y1)**2)), 2)
        distancias.append((i["Lugar"], distancia)) #Va formando una lista de
tuplas con el nombre del lugar y la distancia calculada
    if n < 20:
        input("Presione Enter para ver las distancias desde este lugar.")
        for i in distancias:
            print(f"{i[0]}, Distancia: {i[1]}") # Muestra el lugar y la
distancia calculada desde el lugar elegido
        else : print("Se guardaron las distancias, pero la lista es demasiado
grande para mostrarse")
    return distancias #Devuelve la lista de tuplas

def mostrar_listas_ordenadas(n, distancias_bubble, distancias_quick,
distancias_seleccion, distancias_insercion):
    if n < 15:
        input("Presione Enter para continuar.")
        print("\nLista ordenada por Bubble sort:")
        for i in distancias_bubble:
            print(f"{i[0]}, Distancia: {i[1]}")
        input("Presione Enter para continuar.")
        print("\nLista ordenada por quick sort:")
        for i in distancias_quick:
            print(f"{i[0]}, Distancia: {i[1]}")
        input("Presione Enter para continuar.")
        print("\nLista ordenada por selección:")

```

```

    for i in distancias_seleccion:
        print(f"{i[0]}, Distancia: {i[1]}")
    input("Presione Enter para continuar.")
    print("\nLista ordenada por inserción:")
    for i in distancias_insercion:
        print(f"{i[0]}, Distancia: {i[1]}")
    return None

def obtener_objetivo():
    print("A continuación, puede buscar la distancia más cercana a un valor objetivo mediante búsqueda binaria y lineal.")
    # Se solicita al usuario la distancia deseada
    objetivo = -1
    while objetivo < 0:
        try:
            objetivo = float(input("Ingrese la distancia que desea buscar (se mostrará el lugar más cercano): "))
            if objetivo < 0:
                print("Por favor, ingrese un número positivo.")
        except ValueError:
            print("Entrada inválida. Por favor, ingrese un número válido.")
    return objetivo

```

#### - **busqueda.py**

```

import time

def busqueda_lineal(distancias, lugar):
    tiempo_inicio = time.time()
    cont = 0
    for i in distancias:
        cont += 1
        if i[0].lower() == f"lugar {lugar}":
            tiempo_fin = time.time()
            tiempo_total = tiempo_fin - tiempo_inicio
            print(f"Distancia hasta {i[0]}: {i[1]}")
            print(f"Posición en el ranking de distancias: {cont}")
            print(f"Tiempo de búsqueda lineal: {tiempo_total}")
            return i
    tiempo_fin = time.time()
    tiempo_total = tiempo_fin - tiempo_inicio
    print(f"Tiempo de búsqueda lineal: {tiempo_total}")
    print(f"{lugar} no se encuentra en la lista.")
    return None

#Búsqueda binaria de distancias. Devuelve el lugar mas cercano a la distancia elegida
def binaria_mas_cercano(lista_ordenada, objetivo, destinos):

```

```

    tiempo_inicio = time.time() #Inicia el temporizador
    #Índices para el inicio y fin de la búsqueda
    izquierda = 1 # Empezamos desde el segundo elemento, para exceptuar al
lugar propio
    derecha = len(lista_ordenada) - 1
    mejor = lista_ordenada[1] #inicia con el primer elemento de la lista
    mejor_dif = abs(lista_ordenada[1][1] - objetivo) #diferencia inicial
    #Comienza la búsqueda binaria
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2 #Se calcula el punto medio
        actual = lista_ordenada[medio] #Elemento central
        dif = abs(actual[1] - objetivo) #Diferencia con el objetivo del
usuario
        if dif < mejor_dif: #Si la diferencia es menor, se actualiza el valor
            mejor = actual
            mejor_dif = dif
        #Si el valor es menor al objetivo, buscamos en la mitad derecha. Sino
en la izquierda
        if actual[1] < objetivo:
            izquierda = medio + 1
        elif actual[1] > objetivo:
            derecha = medio - 1
        else:
            # Coincidencia exacta: salimos del bucle
            mejor = actual
    # Fin del cronómetro
    tiempo_total = time.time() - tiempo_inicio
    # Mostrar resultados
    nombre, distancia = mejor
    print(f"\n\n==BUSQUEDA BINARIA==\nResultado más cercano encontrado:")
    print(f"Lugar: {nombre}")
    for destino in destinos:
        if destino['Lugar'] == nombre:
            print(f"Coordenadas: x = {destino['x']}, y = {destino['y']}")
    print(f"Distancia encontrada: {distancia:.2f}")
    print(f"Diferencia con la distancia buscada: {abs(distancia -
objetivo):.2f}")
    print(f"Tiempo de búsqueda binaria (más cercano): {tiempo_total:.6f}
segundos\n")
    return mejor

    #Búsqueda lineal de distancias. Devuelve el lugar mas cercano a la
distancia elegida
def lineal_mas_cercano(lista, base, objetivo, destinos):
    lugar_propio = base["Lugar"].strip().lower() # Obtenemos el nombre del
lugar propio
    tiempo_inicio = time.time() # Inicia el cronómetro
    mejor = None
    mejor_dif = float('inf')

```

```

# Recorre la lista excluyendo el lugar propio por nombre
for lugar, distancia in lista:
    if lugar.strip().lower() == lugar_propio:
        continue
    dif = abs(distancia - objetivo)
    if dif < mejor_dif:
        mejor = (lugar, distancia)
        mejor_dif = dif
tiempo_total = time.time() - tiempo_inicio # Fin del cronómetro
if mejor:
    nombre, distancia = mejor
    print(f"\n\n==BUSQUEDA LINEAL==\nResultado más cercano encontrado:")
    print(f"Lugar: {nombre}")
    for destino in destinos:
        if destino['Lugar'] == nombre:
            print(f"Coordenadas: x = {destino['x']}, y = {destino['y']}")
            print(f"Distancia encontrada: {distancia:.2f}")
            print(f"Diferencia con la distancia buscada: {abs(distancia -
objetivo):.2f}")
            print(f"Tiempo de búsqueda lineal (más cercano): {tiempo_total:.6f}
segundos\n")
        else:
            print("\nNo se encontró ningún lugar distinto al propio.\n")
    return mejor

```

## - ordenamiento.py

```

import time

#Bubble Sort (Ordenamiento por burbuja):
def ordenamiento_bubble(distancias):
    tiempo_inicio = time.time() #Marca el tiempo de inicio
    n = len(distancias)
    for i in range(n): #Recorre la lista n veces
        for j in range(0, n - i - 1):#Compara cada elemento con el siguiente
            if distancias[j][1] > distancias[j + 1][1]:#Condicion de
ordenamiento: Si el elemento actual es mayor que el siguiente, los
intercambia.
                distancias[j], distancias[j + 1] = distancias[j + 1],
distancias[j]
        tiempo_fin = time.time() #Marca el tiempo final
        tiempo_total = tiempo_fin - tiempo_inicio #Calcula el tiempo total de
ejecucion
    print(f"Tiempo del ordenamiento por burbuja: {tiempo_total}")
    return distancias #Devuelve la lista ordenada por bubble sort

```

```

#Quick Sort (Ordenamiento rápido):
def ordenamiento_quick(distancias):
    tiempo_inicio = time.time() #Marca el tiempo de inicio
    def quick_sort(lista):
        if len(lista) <= 1: #Si la lista tiene un solo elemento o está vacía,
            ya está ordenada
            return lista
        else:
            pivot = lista[len(lista) // 2] #Elige el pivote como el elemento
            del medio
            menores = [x for x in lista if x[1] < pivot[1]] #Elementos menores
            que el pivote
            iguales = [x for x in lista if x[1] == pivot[1]] #Elementos
            iguales al pivote
            mayores = [x for x in lista if x[1] > pivot[1]] #Elementos mayores
            que el pivote
            return quick_sort(menores) + iguales + quick_sort(mayores)
    #Recursión para ordenar los sublistas
    resultado = quick_sort(distancias)
    tiempo_fin = time.time() #Marca el tiempo final
    tiempo_total = tiempo_fin - tiempo_inicio #Calcula el tiempo total de
    ejecucion.
    print(f"Tiempo del ordenamiento rápido: {tiempo_total}")
    return resultado #Devuelve la lista ordenada por quick sort

#Selection Sort (Ordenamiento por selección):
def ordenamiento_seleccion(distancias):
    tiempo_inicio = time.time() # Marca el tiempo de inicio
    n = len(distancias)
    for i in range(n):
        # Encuentra el índice del elemento mínimo en la parte no ordenada de
        la lista
        minimo = i
        for j in range(i + 1, n):
            if distancias[j][1] < distancias[minimo][1]:
                minimo = j
        # Intercambia el mínimo encontrado con el primer elemento
        distancias[i], distancias[minimo] = distancias[minimo], distancias[i]
    tiempo_fin = time.time() # Marca el tiempo de fin
    tiempo_total = tiempo_fin - tiempo_inicio # Calcula el tiempo total de
    ejecución
    print(f"Tiempo del ordenamiento por selección: {tiempo_total}")
    return distancias #Devuelve la lista ordenada por selección

#Insertion Sort (Ordenamiento por inserción):
def ordenamiento_insercion(distancias):
    tiempo_inicio = time.time() # Marca el tiempo de inicio

```



```
for i in range(1, len(distancias)):
    clave = distancias[i]
    j = i - 1
    # Mueve los elementos de distancias[0..i-1], que son mayores que
    clave, a una posición adelante de su posición actual
    while j >= 0 and clave[1] < distancias[j][1]:
        distancias[j + 1] = distancias[j]
        j -= 1
    distancias[j + 1] = clave
tiempo_fin = time.time()          # Marca el tiempo de fin
tiempo_total = tiempo_fin - tiempo_inicio
print(f"Tiempo del ordenamiento por inserción: {tiempo_total}")
return distancias                #Devuelve la lista ordenada por inserción
```