# Proyecto Integrador

# ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO

**Alumnos:**

- Kevin Gastaldello - kevin.gastaldello21@gmail.com
- Franco Ghilardi - franco.ghilardi1@gmail.com

**Docentes**

- **Titular:**
- **Tutor:**

**Cátedra:** Programación

**Carrera**: Tecnicatura Universitaria en Programación

**Fecha de entrega**: 9 de Junio de 2025

UTN

**Índice**

# 1. Introducción

El presente trabajo aborda el estudio de algoritmos de ordenamiento y búsqueda, implementados en el lenguaje de programación Python. La elección de este tema surge de su importancia central en el desarrollo de software, dado que ordenar y buscar datos son operaciones fundamentales en la mayoría de las aplicaciones informáticas, desde estructuras simples hasta sistemas complejos de gestión de datos.

Estos algoritmos no solo permiten mejorar la eficiencia en el procesamiento y recuperación de información, sino que también representan una excelente oportunidad para ejercitar el pensamiento lógico y fortalecer la capacidad de análisis y resolución de problemas, competencias clave en la formación de cualquier programador.

El objetivo principal de este trabajo es comprender el funcionamiento de distintos algoritmos de ordenamiento y búsqueda, identificar sus ventajas y desventajas, y poner en práctica su implementación en Python. A través de este análisis comparativo, se busca no solo adquirir un dominio técnico sobre su uso, sino también desarrollar el criterio necesario para seleccionar el algoritmo más adecuado según el contexto y los requisitos específicos de cada problema.

# 2. Marco Teórico

## 2.1. Algoritmos

Un **algoritmo** es un conjunto finito de instrucciones o pasos que permiten resolver un problema específico o realizar una tarea. Según Thomas H. Cormen et al. en *"Introduction to Algorithms"*, un algoritmo debe ser claro, finito, definido y efectivo. En programación, los algoritmos permiten automatizar tareas, procesar información y tomar decisiones lógicas.

## 2.2. Algoritmos de Ordenamiento

Los **algoritmos de ordenamiento** se utilizan para organizar una colección de datos según un criterio determinado (por ejemplo, de menor a mayor). Son esenciales para mejorar la eficiencia de búsquedas, facilitar análisis de datos y optimizar el rendimiento de programas.

### 2.2.1. Ordenamiento por burbuja

Es un algoritmo de ordenamiento simple y fácil de implementar. Funciona comparando cada elemento de la lista con el siguiente elemento y luego intercambiando los elementos si están en el orden incorrecto.

Ejemplo en Python – Bubble Sort:

```
def bubble_sort(lista):
    n = len(lista) #calcula cuántos elementos tiene la lista y los guarda
    for i in range(n): #se itera n veces
        for j in range(0, n - i - 1): #se recorren elementos que aún están desordenados reduciendo el rango cada vez
            if lista[j] > lista[j + 1]: #si el elemento actual es mayor que el siguiente, se intercambian
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
```

### 2.2.2. Ordenamiento por selección

Es otro algoritmo de ordenamiento simple que funciona encontrando el elemento más pequeño de la lista y luego intercambiándolo con el primer elemento. Este proceso se repite hasta que todos los elementos de la lista están ordenados.

Ejemplo en Python – Selection Sort:

```
def seleccion_sort(lista):
    n = len(lista) #se obtiene la cantidad de elementos
    for i in range(n): #se recorre la lista desde el inicio
        min_idx = i #guarda el índice del elemento más chico encontrado hasta el momento
        for j in range(i + 1, n): #se busca el elemento más chico desde i hasta el final
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i] #se intercambia el elemento más chico con el que está en la posición actual i
```

### 2.2.3. Ordenamiento por inserción

Es un algoritmo de ordenamiento que funciona insertando cada elemento de la lista en su posición correcta en la lista ordenada.

Ejemplo en Python – Insertion Sort:

```
def insercion_sort(lista):
    for i in range(1, len(lista)): #empieza desde el segundo elemento
        clave = lista[i] #guarda la “clave” que quiere insertar en la parte ordenada
        j = i - 1
        while j >= 0 and lista[j] > clave: #mientras la parte ordenada (izquierda) tenga valores más grandes que la clave, los va corriendo hacia la derecha
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = clave #inserta la clave en su posición correcta
```

### 2.2.4. Ordenamiento rápido

Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes y luego ordenando cada parte de forma recursiva.

Ejemplo en Python – Quick Sort:

```
def quick_sort(lista):
    if len(lista) <= 1:
        return lista #si la liste tiene 0 o 1 elementos, la devuelve tal cual porque ya está ordenada
    else:
        pivote = lista[0] #toma el primer elemento como pivote
        menores = [x for x in lista[1:] if x <= pivote] #lista con elementos menores o iguales al pivote
        mayores = [x for x in lista[1:] if x > pivote] #lista con elementos mayores al pivote
        return quick_sort(menores) + [pivote] + quick_sort(mayores) #mediante recursión, ordena las sublistas y las combina con el pivote en el medio
```

### 2.2.5. Ordenamiento por mezcla

Es un algoritmo de ordenamiento eficiente que funciona dividiendo la lista en dos partes, ordenando cada parte y luego fusionando las dos partes ordenadas.

Ejemplo en Python – Merge Sort:

```
def merge_sort(lista): #esta es la parte “principal”
    if len(lista) <= 1:
        return lista #si la lista tiene 0 o 1 elementos, ya esta ordenada
    medio = len(lista) // 2 #divide la lista en dos mitades y las ordena recursivamente
    izquierda = merge_sort(lista[:medio])
    derecha = merge_sort(lista[medio:])
    return merge(izquierda, derecha) #une ambas mitades ya ordenadas
def merge(izquierda, derecha): #esta es la parte “auxiliar”
    resultado = [] #inicializa lista vacía para el resultado
    i = j = 0 #inicializa dos punteros
    while i < len(izquierda) and j < len(derecha): #compara elementos de ambas listas y va agregando el menor al resultado
        if izquierda[i] < derecha[j]:
            resultado.append(izquierda[i])
            i += 1
        else:
            resultado.append(derecha[j])
            j += 1
    resultado.extend(izquierda[i:]) #si queda algún elemento en alguna lista, lo agrega al final
    resultado.extend(derecha[j:])
    return resultado
```

## 2.3. Algoritmos de Búsqueda

Los **algoritmos de búsqueda** permiten encontrar uno o varios elementos dentro de una estructura de datos. Son fundamentales para acceder a información de manera rápida y eficiente.

| Tipo | Ejemplos | Requisitos |
|---|---|---|
| **Secuencial** | Búsqueda lineal | No requiere orden |
| **Binaria** | Búsqueda binaria | Requiere lista ordenada |
| **Hash / Indexada** | Diccionarios, tablas hash | Estructura asociativa |
| **Interpolación** | Búsqueda por interpolación | Requiere lista ordenada y datos distribuidos uniformemente |

### 2.3.1. Búsqueda lineal

Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero puede ser lento para conjuntos de datos grandes.

Ejemplo en Python – Búsqueda lineal:

```
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

### 2.3.2. Búsqueda binaria

Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.

Ejemplo en Python – Búsqueda binaria:

```
def busqueda_binaria(lista, objetivo):
    izquierda, derecha = 0, len(lista) - 1 #define extremos de búsqueda: de primer a último índice
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2 #mientras el rango sea válido, calcula el punto medio
        if lista[medio] == objetivo:
            return medio #si el elemento del medio es el objetivo, devuelve su posición
        elif lista[medio] < objetivo:
            izquierda = medio + 1 #si el objetivo está a la derecha, descarta la mitad izquierda
        else:
            derecha = medio - 1 #si el objetivo está a la izquierda, descarta la derecha
    return -1 #si no lo encuentra devuelve este valor
```

### 2.3.3. Búsqueda de hash

Es un algoritmo de búsqueda que utiliza una función hash para asignar cada elemento a una ubicación única en una tabla hash. Esto permite acceder a los elementos en tiempo constante, lo que lo hace muy eficiente para conjuntos de datos grandes.

Ejemplo en Python – Búsqueda de hash:

```
tabla_hash = {
    "clave1": "valor1",
    "clave2": "valor2",
    "clave3": "valor3"
} #se crea tabla con claves y valores
def busqueda_hash(clave):
    return tabla_hash.get(clave, "No encontrado") #si la clave existe, devuelve su valor. Sino, devuelve “No encontrado”
# Ejemplo de uso
```

```
clave_buscada = "clave2"

resultado = busqueda_hash(clave_buscada)

print(f"Resultado para '{clave_buscada}': {resultado}")
```

### 2.3.4. Búsqueda de interpolación

Es un algoritmo de búsqueda que mejora la búsqueda binaria al estimar la posición del elemento deseado en función de su valor. Puede ser más eficiente que la búsqueda binaria para conjuntos de datos grandes con una distribución uniforme de valores.

Ejemplo en Python – Búsqueda de interpolación:

```
def busqueda_interpolacion(arr, x):
    bajo = 0 #define los rangos de búsqueda
    alto = len(arr) - 1
     while bajo <= alto and x >= arr[bajo] and x <= arr[alto]: #mientras el valor este dentro del rango
        if bajo == alto: #caso especial donde solo queda un elemento, verifica si es el buscado
            if arr[bajo] == x:
                return bajo
            return -1
            pos = bajo + ((alto - bajo) * (x - arr[bajo])) // (arr[alto] - arr[bajo]) #estima la posición donde podría estar x
        if arr[pos] == x: #compara el valor en la posición estimada con x y ajusta los límites
            return pos
        elif arr[pos] < x:
            bajo = pos + 1
        else:
            alto = pos - 1
             return -1 #si no se encuentra, devuelve este valor
# Ejemplo de uso
lista = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
valor = 70

indice = busqueda_interpolacion(lista, valor)

if indice != -1:

    print(f"Elemento {valor} encontrado en la posición {indice}")

else:

    print("Elemento no encontrado")
```

## 2.4. Estructuras de datos utilizadas

### 2.4.1. Listas (list)

Son estructuras de datos dinámicas y mutables que permiten almacenar colecciones ordenadas de elementos. Son ampliamente utilizadas porque permiten acceder, modificar, insertar y eliminar elementos fácilmente. Las listas son ideales para algoritmos de ordenamiento y búsqueda, ya que permiten operaciones de índice, que facilitan la implementación de métodos como la búsqueda binaria o el ordenamiento por inserción. Además, la función integrada sort() y el método sorted() aprovechan algoritmos eficientes para ordenar listas.

### 2.4.2. Diccionarios (dict)

Son estructuras de datos que almacenan pares clave-valor, donde cada clave es única. Implementan tablas hash internamente, lo que permite acceder y buscar valores en tiempo promedio constante O(1). Esto los hace muy eficientes para búsquedas rápidas cuando se conoce la clave. Son usados comúnmente para búsquedas de hash o indexadas, donde el acceso directo a la información es crucial para el rendimiento

### 2.4.3. Tuplas (tuple)

Similares a las listas pero inmutables, es decir, no pueden modificarse después de su creación. Esto las hace útiles para almacenar datos que no deben cambiar, garantizando la integridad de la información. Aunque no son tan comunes para operaciones de búsqueda o ordenamiento que requieren modificaciones, las tuplas pueden ser utilizadas para representar datos ordenados y pasarlos como claves en diccionarios o para asegurar que ciertos conjuntos de datos permanezcan constantes.

# 3. Caso Práctico

El trabajo se centra en el desarrollo de un **gestor de contactos** en Python que permite al usuario ingresar o cargar contactos desde un archivo externo, visualizarlos, ordenarlos y realizar búsquedas eficientes por nombre o número telefónico. El objetivo principal es simular una pequeña aplicación de agenda telefónica que permita comparar el rendimiento entre distintos algoritmos de búsqueda y aplicar principios básicos de programación estructurada.

**El programa permite al usuario**:

- Cargar contactos manualmente o desde un archivo JSON precargado.
- Ordenar los contactos alfabéticamente por nombre.
- Buscar contactos por nombre utilizando búsqueda lineal y binaria.
- Buscar contactos por número telefónico.
- Visualizar los resultados de búsqueda y comparar los tiempos de ejecución.

**Explicación de decisiones de diseño:**

- **Entrada de datos validada:** Se implementó una función específica para asegurar que el usuario solo ingrese opciones válidas ("s" o "n"), evitando errores lógicos y mejorando la experiencia de uso.
- **Separación en funciones:** Se modularizó el código para mejorar su legibilidad, reutilización y mantenimiento. Cada función cumple una única responsabilidad (por ejemplo: búsqueda, ordenamiento, carga de datos, etc.).
- **Ordenamiento con Bubble Sort:** Se eligió el algoritmo *Bubble Sort* por su simplicidad y facilidad de implementación en listas pequeñas, que es el caso típico en una agenda de contactos. Aunque no es el más eficiente en términos de complejidad, es adecuado para propósitos didácticos y suficiente para la cantidad de datos esperada.
- **Comparación de algoritmos de búsqueda:** Se incluyeron tanto la búsqueda **lineal** como la **binaria** para resaltar la diferencia de rendimiento entre ambas. Dado que la búsqueda binaria requiere una lista previamente ordenada, se aplica el ordenamiento antes de realizar dicha búsqueda. El objetivo es evidenciar cómo las técnicas de búsqueda adecuadas pueden optimizar el rendimiento en aplicaciones reales.

**Validación del funcionamiento:**

Para validar el correcto funcionamiento del programa, se realizaron pruebas ingresando distintos tipos de datos:

- Se verificó que el ordenamiento alfabético mostrara los contactos en el orden esperado.
- Se probaron búsquedas exitosas y fallidas tanto por nombre como por teléfono, observando que las respuestas fueran precisas y claras para el usuario.
- Se midió el tiempo de ejecución de las búsquedas, permitiendo comprobar empíricamente que la búsqueda binaria, en listas ordenadas, presenta mejor rendimiento que la búsqueda lineal.
- Se comprobó la persistencia y reutilización de contactos precargados desde un archivo JSON, lo que valida el manejo correcto de archivos externos y estructuras JSON en Python.

# 4. Metodología Utilizada

Para el desarrollo de este proyecto se siguió una metodología estructurada en varias etapas, orientada al aprendizaje progresivo y la correcta aplicación de los contenidos trabajados en la materia. A continuación, se detallan los pasos y recursos utilizados:

### 4.1. Investigación previa

Antes de comenzar con el desarrollo, se realizó una investigación exploratoria sobre:

- Algoritmos de búsqueda y ordenamiento (búsqueda lineal, búsqueda binaria, Bubble Sort).
- Manejo de archivos JSON en Python para la carga de datos externos.
- Librerías para mejorar la experiencia de usuario en consola, como colorama para resaltar la salida de texto con colores.

Las fuentes utilizadas incluyeron la documentación oficial de Python, sitios educativos como W3Schools y Programiz, así como foros de discusión como Stack Overflow para resolver dudas puntuales sobre validaciones y tiempos de ejecución.

### 4.2. Etapas de diseño y prueba del código

El trabajo se estructuró en las siguientes etapas:

- **Diseño inicial:** Se definieron los requerimientos del programa (cargar contactos, mostrarlos, ordenarlos y buscar por nombre o teléfono).
- **Desarrollo incremental:** Se comenzó con funciones básicas (carga de datos, visualización), y se fueron incorporando progresivamente funciones de ordenamiento, búsqueda y comparación de rendimiento.
- **Refactorización modular:** Una vez funcional el código, se organizaron las funciones en bloques separados para facilitar su lectura y mantenimiento. También se incorporó validación de entradas para mejorar la robustez del sistema.
- **Pruebas manuales:** Se realizaron diversas pruebas ingresando distintos tipos de contactos y nombres para asegurar que el programa respondiera correctamente ante distintos escenarios (búsquedas exitosas y fallidas, listas vacías, nombres repetidos, etc.).
- **Medición de rendimiento:** Se integró la biblioteca time para medir y comparar los tiempos de ejecución de los algoritmos de búsqueda.

### 4.3. Herramientas y recursos utilizados

- **Lenguaje de programación:** Python 3.11
- **IDE:** Visual Studio Code
- **Librerías externas:**
    - colorama: para mejorar la experiencia visual en consola mediante colores.
    - json: para el manejo de archivos JSON con contactos precargados.
    - time: para calcular tiempos de ejecución de algoritmos.
- **Control de versiones:** Se utilizaron versiones locales del código en cada equipo, compartiendo los avances y realizando revisiones conjuntas para validar que todas las funcionalidades funcionaran correctamente.

### 4.4. Trabajo colaborativo

El desarrollo fue realizado por dos personas, quienes se repartieron las tareas de la siguiente forma:

- Uno de los integrantes se enfocó en la lógica de búsqueda, ordenamiento y estructura del programa.

- El otro se encargó del manejo de archivos, validaciones, presentación en consola y pruebas finales.

# 5. Resultados Obtenidos

El desarrollo del gestor de contactos permitió aplicar de manera práctica los conceptos fundamentales de programación, manejo de estructuras de datos y comparación de algoritmos. A lo largo del trabajo se alcanzaron los siguientes resultados:

### 5.1. Funcionalidades implementadas correctamente

Se logró cumplir con todos los objetivos planteados en el diseño inicial del proyecto. Entre los aspectos que funcionaron de manera correcta se destacan:

- La carga de contactos tanto manual como desde archivo JSON externo.
- El ordenamiento alfabético utilizando el algoritmo *Bubble Sort*.
- La búsqueda de contactos por nombre mediante dos enfoques distintos: búsqueda lineal y búsqueda binaria.
- La búsqueda por número telefónico.
- La medición del tiempo de ejecución de ambos algoritmos de búsqueda para evaluar el rendimiento relativo.
- La visualización clara y colorida en consola.
- La validación de entradas, evitando errores comunes y mejorando la experiencia del usuario.

### 5.2. Casos de prueba realizados

Para validar el funcionamiento del sistema, se realizaron los siguientes casos de prueba:

- Carga de múltiples contactos con nombres y teléfonos variados, incluyendo mayúsculas, minúsculas y caracteres acentuados.
- Búsquedas exitosas por nombre exacto, y búsquedas fallidas para verificar el manejo de errores.
- Comparación de los tiempos de ejecución entre búsqueda lineal y binaria con listas de distintos tamaños.
- Ingreso de caracteres inválidos para validar que el sistema solicitara nuevamente la opción correspondiente (por ejemplo, al elegir "s" o "n" al inicio).

### 5.3. Errores detectados y corregidos

Durante el desarrollo se presentaron algunas dificultades menores:

- Al principio, la búsqueda binaria no devolvía resultados esperados. Esto se debió a que no se ordenaba correctamente la lista antes de aplicar el algoritmo. Se soluciona invocando la función de ordenamiento previamente.
- La validación de la opción inicial (cargar contactos manuales o precargados) permitía cualquier texto. Se implementó una validación en bucle para asegurar que solo se acepten las opciones "s" o "n".
- Algunos contactos no se encontraban debido a diferencias de mayúsculas/minúsculas. Se resolvió usando .lower() para normalizar los datos al comparar.

### 5.4. Evaluación del rendimiento

Una de las partes más interesantes del proyecto fue comparar empíricamente el rendimiento entre la búsqueda lineal y la búsqueda binaria:

- En listas pequeñas, los tiempos de ambos algoritmos fueron similares, aunque la binaria fue levemente más rápida.
- A medida que aumentó la cantidad de contactos, se notó que la búsqueda binaria era consistentemente más eficiente, confirmando la teoría aprendida en clase.
- Se calculó la relación de rendimiento (lineal/binaria) para cada búsqueda, mostrando de forma cuantitativa la mejora obtenida con algoritmos más eficientes.

### 5.5. Repositorio del proyecto

El proyecto se encuentra subido en un repositorio público de GitHub, disponible para su revisión y ejecución: https://github.com/kvin21ok/Integrador_ProgI

# 6. Conclusiones

La realización de este proyecto permitió al grupo afianzar conocimientos fundamentales de programación, como estructuras de datos, validación de entradas, algoritmos de búsqueda y ordenamiento, así como el trabajo con archivos externos y la modularización del código.

### 6.1. Aprendizajes obtenidos

Durante el desarrollo del gestor de contactos, se aprendió a:

- Implementar y comparar distintos algoritmos de búsqueda y ordenamiento.
- Utilizar estructuras condicionales y ciclos para validar la interacción del usuario.
- Leer y manipular archivos en formato JSON.
- Medir el rendimiento de fragmentos de código para analizar su eficiencia.
- Modularizar el código dividiendo la lógica en funciones reutilizables.
- Mejorar la experiencia de uso en la consola utilizando librerías como colorama.

### 6.2. Utilidad del tema trabajado

Los conceptos aplicados en este trabajo tienen un alto valor tanto para proyectos educativos como para desarrollos reales. La capacidad de manejar y buscar datos de manera eficiente es fundamental en prácticamente cualquier aplicación, desde agendas personales hasta bases de datos más complejas. Además, la evaluación de rendimiento permite tomar decisiones informadas sobre qué soluciones son más adecuadas para cada contexto.

### 6.3. Posibles mejoras o extensiones futuras

Aunque el proyecto cumple con todos los objetivos planteados, se identificaron varias ideas para mejorarlo en futuras versiones:

- Reemplazar el algoritmo de ordenamiento por uno más eficiente como QuickSort o MergeSort.
- Permitir editar o eliminar contactos.
- Guardar los cambios en el archivo JSON para mantener los datos entre ejecuciones.

### 6.4. Dificultades y soluciones

Durante el desarrollo surgieron algunas dificultades, especialmente en la lógica de búsqueda binaria y en la validación de opciones ingresadas por el usuario. Estas situaciones fueron superadas mediante pruebas iterativas, consultas a documentación oficial y colaboración entre los integrantes del grupo. El trabajo en equipo fue clave para resolver problemas de lógica y mejorar el diseño del código.

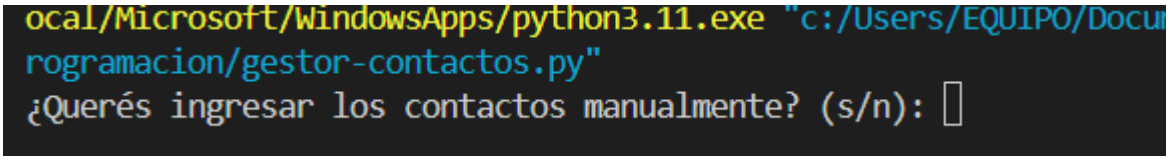
# 7. Bibliografía


- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3.ª ed.). MIT Press.
- Python Software Foundation. (n.d.). *5.1. More on lists*. En *The Python tutorial*. https://docs.python.org/3/tutorial/datastructures.html#more-on-lists
- Python Software Foundation. (n.d.). *5.5. Dictionaries*. En *The Python tutorial*. https://docs.python.org/3/tutorial/datastructures.html#dictionaries
- Python Software Foundation. (n.d.). *5.3. Tuples and sequences*. En *The Python tutorial*. https://docs.python.org/3/tutorial/datastructures.html#tuples-and-sequences
- GeeksforGeeks. (n.d.). *GeeksforGeeks: A computer science portal for geeks*. https://www.geeksforgeeks.org
- Real Python. (n.d.). *Real Python*. https://realpython.com
- Apuntes de la Cátedra. (2025). Universidad Tecnológica Nacional

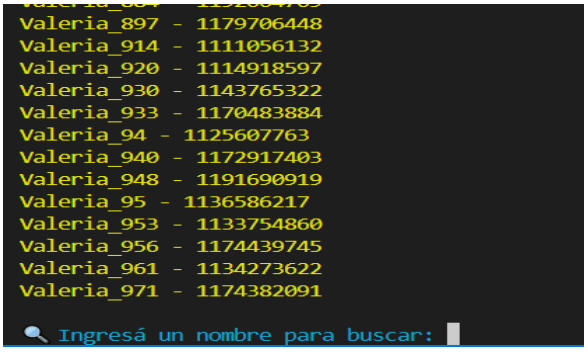

# 8. Anexos

Capturas del programa funcionando:

Inicio del programa




Se elige el modo de carga automática de contactos desordenados, el programa los ordena por nombre mediante Bubble Sort y solicita un nombre para buscar



Se realizan ambas formas de búsqueda y se comparan tiempos, mostrando la relación en términos de tiempo entre ambos algoritmos

```
🔍 Ingresá un nombre para buscar: Valeria_212

📊 Resultados de búsqueda por nombre:
🔎 Lineal: {'nombre': 'Valeria_212', 'telefono': '1177601111'} (0.00040809996426 segundos)
🔎 Binaria: {'nombre': 'Valeria_212', 'telefono': '1177601111'} (0.00001119996887 segundos)

📈 Relación de rendimiento (lineal/binaria): 36.44 a 1
```

Si se busca dentro de los primeros valores de la lista, la búsqueda lineal le gana en tiempo a la binaria

```
🔍 Ingresá un nombre para buscar: Andrés_114

📊 Resultados de búsqueda por nombre:
🔎 Lineal: {'nombre': 'Andrés_114', 'telefono': '1170153477'} (0.00001449999399 segundos)
🔎 Binaria: {'nombre': 'Andrés_114', 'telefono': '1170153477'} (0.00003970006946 segundos)

📈 Relación de rendimiento (lineal/binaria): 0.37 a 1
```

```
🔍 Ingresá un nombre para buscar: Andrés_128

📊 Resultados de búsqueda por nombre:
🔎 Lineal: {'nombre': 'Andrés_128', 'telefono': '1135641511'} (0.00004629988689 segundos)
🔎 Binaria: {'nombre': 'Andrés_128', 'telefono': '1135641511'} (0.00004730001092 segundos)

📈 Relación de rendimiento (lineal/binaria): 0.98 a 1
```

Tendencia que se revierte cuando se solicitan contactos que estén más alejados del inicio de la lista

```
🔍 Ingresá un nombre para buscar: Andrés_8

📊 Resultados de búsqueda por nombre:
🔎 Lineal: {'nombre': 'Andrés_8', 'telefono': '1182488444'} (0.00004700000864 segundos)
🔎 Binaria: {'nombre': 'Andrés_8', 'telefono': '1182488444'} (0.00002739997581 segundos)

📈 Relación de rendimiento (lineal/binaria): 1.72 a 1
```

```
🔍 Ingresá un nombre para buscar: Andrés_804

📊 Resultados de búsqueda por nombre:
🔎 Lineal: {'nombre': 'Andrés_804', 'telefono': '1162384415'} (0.00005000003148 segundos)
🔎 Binaria: {'nombre': 'Andrés_804', 'telefono': '1162384415'} (0.00002039992251 segundos)

📈 Relación de rendimiento (lineal/binaria): 2.45 a 1
```

- Enlace al video explicativo: https://www.youtube.com/watch?v=sdaAh6PVmhw&ab_channel=KevinGastaldello