

Trabajo Práctico

Integrador

Tema elegido

Algoritmos de Búsqueda y Ordenamiento de Datos en Python

Alumnos:

Nicolás Olima < nicolima200@gmail.com />

Nicolás Pannunzio < nicolas.a.pannunzio@gmail.com />

Asignatura:

Programación 1

Profesor:

Julietta Trapé

9 de Junio 2025

Índice

1. Introducción	1
2. Marco Teórico	2
3. Caso Práctico	13
4. Metodología Utilizada	16
5. Resultados Obtenidos	19
6. Conclusiones	21
7. Bibliografía	23
8. Anexos	24

1. Introducción

El tema elegido para el desarrollo del presente trabajo integrador es Algoritmos de Búsquedas y Ordenamiento de datos en Python, entendemos que hoy en día el correcto uso, manipulación y transformación de esos Datos en Información es muy relevante para distintos sectores sean públicos o privados, puesto que de ellos dependen distintas métricas obtenidas a partir del análisis de Datos, e incluso las decisiones se basan en dichos resultados.

En programación podemos desarrollar un escenario ideal para trabajar con Datos, puesto que tenemos muchas herramientas que nos lo permiten. La forma de trabajo, de procesar y de entender esos Datos, va a depender de distintos factores, por ejemplo si nos encontramos con una lista ordenada o desordenada y el volumen de éstos; son cruciales para saber qué camino tomar mientras programamos.

El objetivo principal es entender las herramientas que se utilizan en estos algoritmos, saber cuándo se implementan una u otra, comprender la sintaxis del código y abordar un caso práctico, que se aproxime a una situación real. De este modo, se busca no solo adquirir conocimientos teóricos, sino también desarrollar competencias técnicas aplicables a desafíos concretos de programación.

2. Marco Teórico

“La búsqueda de un elemento en una secuencia es un algoritmo básico pero importante. El problema que intenta resolver puede plantearse de la siguiente manera: Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia, si se encuentra, de no encontrarse el valor en la secuencia señalarlo apropiadamente.” (Wachenchauzer et al., 2012, p. 107).

“Ordenar el contenido de una lista es un problema importante porque se plantea en numerosos campos de aplicación de la programación: la propia palabra <<ordenador>> lo pone de manifiesto. Ordenar es, quizá, el problema más estudiado y para el que existe mayor número de soluciones diferentes, cada una con sus ventajas e inconvenientes o especialmente adaptada para tratar casos particulares.” (Andrés Marzal, Isabel García, 2009, p. 205)

Algoritmos de búsqueda

Básicamente, el objetivo de los algoritmos de búsqueda es encontrar la ubicación de un elemento específico dentro de una lista de elementos. La elección del algoritmo de búsqueda dependerá de si nuestra lista está ordenada o desordenada. Si los elementos de nuestra lista están ordenados, podemos usar un algoritmo de búsqueda binaria, mientras que para una lista desordenada, este algoritmo no nos servirá. En este último caso, sería conveniente utilizar un algoritmo de búsqueda lineal. Además, existen métodos más avanzados como la búsqueda de interpolación para listas numéricas ordenadas con distribución uniforme, o la búsqueda de hash, ideal para acceder rápidamente a datos mediante claves únicas.

Algoritmos de búsqueda lineal

Los algoritmos de búsqueda lineal o secuencial, implican recorrer la lista de elementos uno por uno hasta encontrar un elemento específico. Estos algoritmos son muy fáciles de implementar, pero pueden ser muy ineficientes dependiendo del tamaño del conjunto de datos y de la ubicación del elemento a encontrar.

Para implementar este algoritmo, simplemente deberíamos definir una función que reciba como argumentos una lista y un elemento a encontrar. Dentro de esta función definimos una estructura *for* que recorra la lista y compare uno a uno los elementos de esta con el elemento que le dimos como argumento. Si lo encuentra nos devuelve el índice donde encontró el elemento, si no lo encuentra, nos devuelve -1:

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

Si bien este algoritmo es útil para listas desordenadas o pequeñas, el problema de este tipo de búsqueda, es que el tiempo que tarda en encontrar el elemento objetivo, es proporcional al tamaño de la lista, por lo que en conjuntos muy grandes, el tiempo para encontrar el objetivo sería igual de grande, lo que se traduce en una búsqueda ineficiente.

Algoritmos de búsqueda binaria

Los algoritmos de búsqueda binaria o búsqueda logarítmica son muy eficientes, pero sólo funcionan en listas ordenadas. Este método consiste en dividir la lista en dos mitades repetidas veces, comparando nuestro elemento con el elemento del medio de la lista en cada repetición y desechando la mitad en la que no puede estar nuestro elemento, reduciendo así el conjunto a una mitad y disminuyendo drásticamente la cantidad de comparaciones necesarias para determinar la existencia de nuestro elemento objetivo en ese conjunto.

```
def busqueda_binaria(lista, objetivo):  
  
    izquierda, derecha = 0, len(lista) - 1  
  
    while izquierda <= derecha:  
  
        medio = (izquierda + derecha) // 2
```

```

        if lista[medio] == objetivo:

            return medio

        elif lista[medio] < objetivo:

            izquierda = medio + 1

        else:

            derecha = medio - 1

    return -1

```

Tomando el código anterior, tomemos como ejemplo la *lista* = [2, 4, 6, 8, 9, 12] y *objetivo* = 4.

Al ejecutarse la función, *izquierda* vale 0 y *derecha* vale 5 (ya que *derecha* es igual a la longitud de la *lista* menos 1), luego el *while* se ejecuta ya que se cumple $0 \leq 5$. La variable *medio* será igual a la división entera de 5 (*izquierda+derecha*) entre 2, lo que nos da 2, este será el índice de nuestro elemento “pivote” (*lista*[2]=6). Luego el *if*: si el elemento central (*lista*[*medio*]) de nuestra lista es igual a nuestro elemento *objetivo*, habremos tenido suerte y encontrado nuestro *objetivo* en la primera comparación! Pero si es diferente, comprueba si el elemento medio es menor al *objetivo*, y de ser así, el elemento medio + 1 pasa a ser nuestro extremo izquierdo y habremos desechado la parte izquierda de la lista. De lo contrario, si el elemento medio es mayor a nuestro objetivo, el elemento medio -1 pasa a ser nuestro extremo derecho de la lista y habremos desechado la parte derecha de la misma. Este último caso sería el nuestro ya que el elemento medio (*lista*[2]) es igual a 6 y es mayor a nuestro *objetivo* (4). Por lo tanto, nuestro elemento *derecha* pasa a ser *medio*-1, es decir *lista*[1], que es el valor 4. Después de esto, entramos de nuevo al ciclo *while* ya que *izquierda* (vale 0) \leq *derecha* (vale 1). Aquí *medio* es igual a la división entera de 1 (*izquierda+derecha*) entre 2, lo que nos da 0, el elemento con índice 0 será nuestro pivote. El *if* comprueba si hay igualdad entre el elemento con *lista*[0] (valor 2) y nuestro *objetivo* (valor 4), seguimos con la subestructura *elif lista*[0] < 4, lo cual es verdadero, por lo tanto se ejecuta la sentencia *izquierda=medio+1*, lo que nos lleva a que nuestro extremo izquierdo será *lista*[0+1]=*lista*[1], se vuelve a iniciar el ciclo *while* y se verifica *izquierda* (vale 1) \leq *derecha* (vale 1), la condición se cumple, se ejecuta el *while*. La variable pivote *medio* será igual a 1 ((*izquierda+derecha*)/2), luego el *if* verifica si *lista*[1]==*objetivo*, lo que resulta verdadero, eso significa que hemos encontrado nuestro *objetivo*, por lo tanto, nos retorna *medio*, que es la posición de nuestro *objetivo* (4) en la lista.

Algoritmos de interpolación

La búsqueda por interpolación es un algoritmo para buscar en matrices ordenadas y uniformemente distribuidas. Mejora la búsqueda binaria al estimar la posición del objetivo a partir de los valores de los índices inferior y superior.

La búsqueda por interpolación puede superar a la búsqueda binaria para conjuntos de datos grandes y uniformemente distribuidos, con una complejidad temporal de $O(\log \log n)$ en el mejor de los casos.

Casos de uso:

- Búsqueda en conjuntos de datos grandes y uniformemente distribuidos.
- Aplicaciones donde la búsqueda binaria no es eficiente debido a la distribución uniforme.

Algoritmos Hash

Los algoritmos de Hash son una herramienta fundamental para almacenar y acceder a datos de forma altamente eficiente.

Estos mapas hash de Python, a menudo llamados diccionarios de Python, son estructuras de datos que almacenan pares clave-valor.

Las claves actúan como etiquetas para recuperar valores rápidamente sin necesidad de buscar en todo el conjunto de datos.

Los diccionarios de Python utilizan internamente funciones hash para asignar claves a ubicaciones específicas de memoria, lo que permite búsquedas rápidas. Esto los hace muy eficientes para tareas como contar, organizar o crear asignaciones entre conjuntos de datos. Los mapas hash de Python destacan porque admiten varios tipos de claves (cadenas, números o incluso tuplas), siempre que sean inmutables.

Algoritmos de ordenamiento

El **objetivo** de un algoritmo de **ordenamiento** consiste en **reorganizar**, repitiendo una secuencia de pasos, una **lista** de elementos o nodos. El ordenamiento puede ser numérico o alfabético, de forma ascendente, descendente o según otro criterio, con la finalidad de **operar** sobre la lista.

Los **algoritmos** de **ordenamiento** son una pieza vital de muchas otras aplicaciones, como herramientas de **búsqueda**, **análisis de datos** y **e-commerce**. Hay muchos algoritmos de ordenamiento, pero la mayoría de las aplicaciones usan ordenamientos de una **complejidad computacional** relativamente **baja**, por ejemplo *Quicksort* o *merge sort*.

Siguiendo con el factor **complejidad computacional**, este es un vector **clave** al evaluar un **algoritmo** de **ordenamiento**. Esta complejidad hace referencia a cuánto tiempo y memoria requiere un algoritmo particular para funcionar, y teniendo en cuenta que el tiempo y la memoria consumidos dependen de las especificaciones del problema a resolver (por ejemplo ordenar 100 elementos o 1 millón), los científicos de la computación utilizan la notación *big-O* para describir qué tan rápidamente crece el uso de recursos a medida que crece el tamaño del problema. En resumen, la notación *big-O* sirve para comparar la eficiencia de los algoritmos teniendo en cuenta el tiempo de ejecución y la memoria utilizada. (Stephen Eldridge, 2025)

En cuanto a la clasificación de los algoritmos de ordenamiento, podemos separarlos en “Algoritmos de comparación” y “Algoritmos no basados en comparación”.

Ordenamiento Burbuja

El ordenamiento **burbuja** hace múltiples pasadas a lo largo de una lista. Compara los ítems **adyacentes** e **intercambia** los que no están en orden. Cada pasada a lo largo de la lista ubica el siguiente valor más grande en su lugar apropiado. En esencia, cada ítem “burbujea” hasta el lugar al que pertenece. (Ver Figura 1 en el Anexo)

Ordenamiento por Selección

El ordenamiento por selección identifica el valor más alto durante cada pasada por la lista y lo coloca en su posición definitiva dentro del conjunto ordenado.

(Ver Figura 2 en el Anexo)

Ordenamiento por inserción

El algoritmo de ordenamiento por inserción mantiene una sublista ordenada en la parte inicial de la lista. A medida que recorre los elementos, cada nuevo ítem se inserta en la posición

correspondiente dentro de esa sublista, expandiéndola progresivamente en un elemento más. (Ver Figura 3 en el Anexo)

Ordenamiento por mezcla

El **ordenamiento por mezcla** se basa en dividir la lista recursivamente en mitades, hasta obtener sublistas de uno o ningún elemento. Luego, en el proceso inverso, estas sublistas se van fusionando de forma ordenada, reconstruyendo la lista original con todos sus elementos en orden. (Ver Figura 4 en el Anexo)

Ordenamiento rápido

En el ordenamiento rápido, se toma un elemento aleatorio de la lista como pivote y los elementos restantes se dividen en dos sublistas: los que son menores que el pivote y los que son mayores que el pivote. Este proceso se repite con cada sublista hasta que estén ordenadas. Por último, las sublistas ordenadas se combinan nuevamente con el pivote formando una nueva lista. Este proceso se repite hasta obtener la lista completa con los elementos ordenados.

Cabe mencionar que este método funciona bien para listas grandes, pero no es estable (lo que significa que puede cambiar el orden de elementos iguales) y puede ser lento si los pivotes se eligen mal (si el sesgo está muy a la derecha o muy a la izquierda), ya que la división entre elementos mayores y elementos menores que el pivote sería muy desigual. (Ver Figura 5 en el Anexo)

Ordenamiento por montículo

El ordenamiento por montículo se basa en una estructura de árbol llamada montículo o *heap*. Un **heap** es una estructura de datos en forma de **árbol binario completo** y, dependiendo de su estructura, si cada nodo es mayor o igual que sus hijos, se denomina **max-heap** y si cada nodo es menor o igual a sus hijos, se denomina **min-heap**.

El ordenamiento por montículo utiliza **max-heap** para ordenar de mayor a menor y **min-heap** para ordenar de menor a mayor.

El **ordenamiento por montículos** utiliza la propiedad de la **raíz** para ordenar la lista. Una vez que la lista cumpla las propiedades del **montículo**, quitamos la **raíz** y la colocamos al **final** de la lista. Con los datos que sobran, creamos otro **montículo** y repetimos hasta que todos los datos estén **ordenados**. (Ver Figura 6 en el Anexo)

Ordenamiento Shell

El **ordenamiento de Shell**, también conocido como **“ordenamiento por incrementos decrecientes”**, es una mejora del ordenamiento por **inserción**. Su estrategia consiste en dividir la lista en varias **sublistas**, que no están formadas por elementos contiguos, sino por elementos **separados** entre sí por un cierto **intervalo**, llamado **incremento o brecha (gap)**.

Cada una de estas sublistas se **ordena** utilizando el algoritmo de **inserción**. A medida que el **incremento** disminuye progresivamente (por ejemplo, de **3 a 1**), se van generando nuevas **sublistas** que permiten **acercar** cada elemento a su posición correcta de manera más eficiente que si usáramos inserción simple desde el inicio.

Por ejemplo, si se parte de una lista de 9 elementos y se utiliza un incremento de 3, se crean 3 sublistas con elementos separados por 3 posiciones. Una vez ordenadas estas sublistas, la lista resultante aún no está completamente ordenada, pero los elementos ya se encuentran mucho más cerca de su ubicación final, lo cual facilita la ordenación final cuando el incremento llega a 1. (Ver Figura 7.1 y Figura 7.2 en el Anexo)

Ordenamiento por conteo

El ordenamiento por conteo es un algoritmo que ordena los elementos de un array contando el número de ocurrencias de cada elemento único. El conteo se almacena en un array auxiliar y el ordenamiento se realiza mapeando el conteo como índice del array auxiliar.

Funcionamiento del conteo y clasificación

1. Descubrir el elemento máximo (sea max) de la matriz dada.
2. Inicializar una matriz de longitud max+1 con todos los elementos 0. Esta matriz se utiliza para almacenar el recuento de elementos en la matriz.
3. Almacenar el recuento de cada elemento en su índice correspondiente en count. la matriz.

Por ejemplo: si el recuento del elemento 3 es 2, entonces 2 se almacena en la tercera posición de contar. Matriz. Si el elemento "5" no está presente en la matriz, se almacena 0 en la quinta posición.

4. Almacenar la suma acumulada de los elementos del array de conteo. Esto facilita la colocación de los elementos en el índice correcto del array ordenado.

5. Encontrar el índice de cada elemento del array original en el array de conteo. Esto te dará el conteo acumulado. Coloca el elemento en el índice calculado, como se muestra en la figura a continuación.

6. Después de colocar cada elemento en su posición correcta, disminuya su conteo en uno.

(Ver Figura 9 en el Anexo)

Ordenamiento por raíz

El ordenamiento por radix o raíz es un algoritmo de ordenamiento potente y eficiente que gestiona grandes conjuntos de datos, especialmente cuando estos se componen de números o cadenas con múltiples dígitos o caracteres. A diferencia de los métodos tradicionales de ordenamiento basados en la comparación, el algoritmo de ordenamiento por radix procesa cada dígito o carácter individualmente.

Este enfoque lo hace especialmente eficaz para ordenar grandes conjuntos de enteros o cadenas. Comprender la ordenación por radix es esencial para quienes aprenden estructuras de datos y algoritmos, ya que introduce una forma única y eficiente de organizar los datos.

El ordenamiento por radix es un algoritmo que ordena los elementos agrupando primero los dígitos individuales con el mismo valor posicional. Luego, ordena los elementos según su orden creciente o decreciente.

(Ver figura 10 en el Anexo)

Ordenamiento por cubos

El ordenamiento por cubos es un algoritmo que divide los elementos de una matriz sin ordenar en varios grupos llamados cubos. Cada cubo se ordena mediante cualquiera de los algoritmos de ordenamiento adecuados o aplicando recursivamente el mismo algoritmo. Este algoritmo es especialmente útil para ordenar grandes conjuntos de datos con una distribución uniforme.

Su funcionamiento consiste en:

- Crear contenedores: primero, decidimos cuántos contenedores necesitamos y qué rango de números contendrá cada uno.
- Distribuir números: A continuación, revisamos la lista de números y colocamos cada número en su grupo apropiado en función de su valor.
- Ordenar cada grupo: Una vez que todos los números están en sus grupos, ordenamos cada grupo. Para este paso, podemos usar un método de ordenación simple, como la ordenación por inserción .
- Combinar cubos: finalmente, tomamos los números ordenados de cada cubo y los combinamos para formar la lista ordenada.

(Ver figura 11 en el Anexo)

Notación big-O

Teniendo en cuenta que en conjunto, la **búsqueda** y el **ordenamiento** se usan para mejorar la **velocidad** y **eficiencia** de los programas, especialmente cuando se trata de **conjuntos grandes** de datos, podemos medir esta velocidad y eficiencia con la notación **big-O**.

La notación big-O representa la complejidad del peor de los casos de un algoritmo y, para describir esto, utiliza términos algebraicos.

En palabras simples, big-O nos da una evaluación que permite clasificar los algoritmos según su tasa de crecimiento, es decir, cómo varía su rendimiento (tiempo o espacio) a medida que aumenta el tamaño de la entrada. Cuando varios algoritmos presentan la misma tasa de crecimiento, se agrupan bajo la misma notación de complejidad, comúnmente expresada en notación Big-O.

Tiempo constante $O(1)$

Se dice que un algoritmo tiene **tiempo constante** cuando su **tiempo de ejecución no varía con el tamaño del conjunto de datos**. Es decir, sin importar cuántos elementos haya en la entrada, la operación siempre tomará un tiempo similar.

Este tipo de algoritmos es **altamente escalable**, ya que su rendimiento se mantiene estable incluso con grandes volúmenes de datos. Acceder directamente a un elemento de una lista por índice, por ejemplo, es una operación típica de complejidad **$O(1)$** .

Tiempo logarítmico $O(\log n)$

Un algoritmo tiene **complejidad logarítmica** cuando su tiempo de ejecución **crece en proporción al logaritmo del tamaño de la entrada**. En otras palabras, aunque la cantidad de datos aumente significativamente, el número de pasos necesarios **crece muy lentamente**.

Este tipo de complejidad es común en algoritmos que **dividen la entrada en partes más pequeñas en cada paso**, como la **búsqueda binaria**, donde la lista se reduce a la mitad en cada iteración hasta encontrar el valor deseado.

Tiempo lineal $O(n)$

Un algoritmo presenta **complejidad lineal** cuando su **tiempo de ejecución crece en proporción directa al tamaño de la entrada**. Es decir, si la cantidad de datos se duplica, el tiempo que tarda el algoritmo también se duplica aproximadamente.

Este comportamiento es típico en algoritmos que **recorren todos los elementos de una lista o estructura**, como los bucles **for** que procesan cada ítem uno por uno. Cuantos más datos haya, mayor será el tiempo necesario para completarse.

Tiempo cuadrático $O(n^2)$

Los algoritmos con **tiempo cuadrático** tienen un crecimiento proporcional al **cuadrado del tamaño de la entrada**. Esto significa que si duplicamos la cantidad de datos, el tiempo de ejecución se cuadruplica. Es común en algoritmos que usan **bucles anidados**, donde cada

elemento debe compararse o procesarse frente a todos los demás. Este tipo de complejidad puede volverse muy costosa rápidamente, por lo que es importante controlar el tamaño de los datos que se procesan.

Tiempo exponencial $O(2^n)$

Un algoritmo tiene **complejidad exponencial** cuando su tiempo de ejecución **se duplica con cada elemento adicional** en la entrada. Este comportamiento suele aparecer en algoritmos de **fuerza bruta con recursión**, como el cálculo de subconjuntos o permutaciones sin optimización. Su rendimiento cae drásticamente con entradas medianas o grandes, por lo que se recomienda evitar su uso, salvo en problemas muy específicos y con conjuntos de datos pequeños.

Tiempo factorial $O(n!)$

Los algoritmos con **complejidad factorial** presentan un crecimiento extremadamente rápido: su tiempo de ejecución **aumenta de forma dramática** incluso con pocos elementos. Esto ocurre, por ejemplo, en algoritmos que generan **todas las permutaciones posibles** de una lista. Debido a su alto costo computacional, este tipo de algoritmos deben limitarse a situaciones controladas y de pequeña escala, ya que no son viables para datos grandes.

3. Caso Práctico

La elección de un caso práctico adaptable al tema en estudio, nos abre un abanico muy amplio de opciones debido a su importancia en la manipulación y organización con Datos. En este sentido, nos propusimos abordar un:

< Buscador de productos en un catálogo de e-commerce />

Consideramos que este tema a desarrollar es bastante probable de encontrarlo en un entorno laboral real, motivo por el cual despertó nuestro interés para ponerlo en práctica.

El escenario propuesto simula el funcionamiento básico de un catálogo de productos para un negocio de ventas minoristas que se encuentra en expansión y desea implementar un canal de ventas en línea.

Para ello, se busca desarrollar una funcionalidad que permita:

- Visualizar el listado de productos.
- Ordenar los productos por nombre o precio (de menor a mayor y viceversa).
- Buscar un producto específico por su nombre.

El catálogo estará representado mediante una estructura de datos (lista de diccionarios), donde cada producto está representado por un diccionario y tendrá los siguientes atributos (claves):

- nombre
- precio
- stock
- categoría

Decisiones de diseño del programa

- Para representar el catálogo de productos hardcodeamos (escribimos manualmente) un diccionario de datos, agregando algunas categorías de productos (Electrónica, Deportes, etc) y dentro de estas, algunos productos característicos (Laptop, Balón de fútbol, etc).
- Luego implementamos distintas librerías siguiendo el propósito inicial. Por ejemplo con faker, generamos datos falsos realistas para las marcas y modelos de nuestros productos; y con los módulos nativos de Python, random y time, logramos por un lado generar números aleatorios para los stocks y los precios, y por otro, medir los tiempos de ejecución.
- Una vez implementada la función para generar los nombres de los productos, pasamos a la siguiente instancia que era implementar los métodos de búsqueda y ordenamiento.
- Decidimos generar dos tipos de listas que se diferencian por su tamaño (una de 100 elementos y otra de 10.000) para poder comparar la velocidad de los métodos de búsqueda y ordenamiento
- Los métodos de ordenamiento aplicados fueron: ordenamiento rápido y por burbuja.
- Finalmente, los métodos que decidimos comparar fueron el algoritmo de búsqueda lineal y el algoritmo de búsqueda binaria en cada una de las listas, a fin de observar el comportamiento de cada una en las operaciones respectivas.

En base a la propuesta planteada, desarrollamos el código necesario para aplicar algunos de los métodos de Búsqueda y Ordenamiento de Datos con Python sobre listas de diccionarios. A través de múltiples pruebas, pudimos comprobar su funcionamiento y analizar su comportamiento en distintos escenarios. Observando:

- De modo intencional, una búsqueda binaria no funciona si la lista no está previamente ordenada, lo cual validó su dependencia estructural.
- El tamaño del diccionario de datos, influye de modo directo en la performance de la búsqueda que estemos utilizando. Por ejemplo, en listas pequeñas, en general una

búsqueda lineal es más eficiente, mientras que en listas grandes resulta más conveniente utilizar una búsqueda binaria.

- En tanto, los métodos de ordenamiento Bubble Sort, mostró buen desempeño en volúmenes pequeños, pero resultó ineficiente en listas grandes. En cambio, QuickSort mantuvo su rendimiento óptimo, incluso con grandes volúmenes de datos.

Todas estas observaciones se vieron reflejadas tanto en el comportamiento lógico de las funciones como en la medición objetiva de tiempos de ejecución, los cuales fueron capturados mediante *time.perf_counter()* y se incluyen como evidencia en el Anexo.

Estas validaciones nos permitieron confirmar la teoría en un entorno práctico, reforzando el aprendizaje de cómo, cuándo y por qué conviene aplicar un algoritmo sobre otro en función del contexto.

4. Metodología Utilizada

La metodología empleada para el desarrollo práctico del TP integrador fue puesta en marcha posteriormente a la investigación que hicimos y plasmamos en el marco teórico.

Con el caso práctico ya elaborado, investigamos cómo crear una lista lo suficientemente grande como para aplicar los métodos de búsqueda y ordenamiento. Fue entonces que, investigando, nos topamos con la librería *Faker*, que nos permitió alcanzar este objetivo con facilidad y sin inconvenientes. Esta librería nos permite generar infinidad de datos falsos, como nombres, direcciones, números de teléfono, patentes de vehículos, provincias, calles, etc... y, todo esto podemos restringirlo a ubicaciones geográficas (países o idiomas).

Luego, con los módulos nativos de Python, incorporamos `random` y `time`, ya que necesitábamos generar valores numéricos aleatorios y medir los tiempos de ejecución de cada algoritmo aplicado. En este proceso descubrimos que `time.time()` no brindaba la precisión deseada para nuestras comparaciones, por lo que tomamos la decisión de reemplazarla por `time.perf_counter()`, lo que nos permitió obtener **mediciones más exactas y confiables** para el análisis de rendimiento.

El siguiente paso fue crear dos listas vacías, una pequeña y otra grande, que serían de alguna manera los pilares para las posteriores pruebas.

La lista grande de 10000 productos, la creamos con la función para crear listas (generando diccionarios y agregándolos a una lista), mientras que para la lista chica, tomamos los primeros 100 elementos de la lista grande, ya que para hacer las pruebas de velocidad de los métodos de búsqueda, necesitamos que el elemento buscado (objetivo) esté en las dos listas. Por esta razón no era recomendable que la lista chica sea también aleatoria (condición de la que nos percatamos durante las pruebas).

Cada diccionario, representando un producto único, tiene las siguientes claves:

- **nombre:** Generado por la función creada para tal motivo, tomando uno de los productos del diccionario inicial hardcoded (el de las categorías y productos típicos) más una palabra aleatoria obtenida de *faker* y un número aleatorio. Ej: 'Auriculares Spend-6164'

- **precio:** Generamos aleatoriamente un valor de tipo float con `random.uniform()` entre los valores 5000.0 y 99999.9.
- **stock:** Generamos aleatoriamente un valor entero con `random.randint()` entre 0 y 1000.
- **categoría:** Tomada también del diccionario inicial hardcodeado. Ej: 'Hogar', 'Libros', etc.

Luego usamos estas listas para poner a prueba los algoritmos de ordenamiento y búsqueda mencionados *ut supra*.

Usamos diferentes variables para cada lista:

- `lista_grande`,
- `lista_chica`,
- `lista_chica_ordenada_rapido` (QuickSort),
- `lista_grande_ordenada_rapido`,
- `lista_chica_ordenada_burbuja` (BubbleSort)
- `lista_grande_ordenada_burbuja`

Creamos funciones básicas de utilidad que nos permitieran:

- Visualizar el listado de productos.
- Ordenar los productos por nombre o precio.
- Buscar un producto específico por nombre.

Utilizamos ciclos `for` y `while` en el desarrollo del código para generar las listas, y también incorporamos condicionales `if / elif / else` en el método de búsqueda binaria, entre otros.

Desde la primera línea de código escrita hasta el final, nos apoyamos en herramientas colaborativas que nos permitieron trabajar de forma remota y coordinada. Con **Discord** mantuvimos una conversación fluida durante todo el proceso y, con **Google Docs** y **Google Colab** pudimos escribir y editar código en tiempo real. Luego, para poder estructurar mejor el código, utilizamos **Visual Studio Code**. Estas herramientas fueron clave para acortar distancias físicas y facilitar el trabajo en equipo.

Durante la ejecución del programa y al final del mismo, mostramos los resultados de tiempo de ejecución de los algoritmos de búsqueda y ordenamiento a fin de poder visualizar y demostrar las diferencias de velocidad que presentan los estos métodos en diferentes escenarios.

5. Resultados Obtenidos

Cuando comenzamos a codear hubo ciertos aspectos que no tuvimos en cuenta, tal vez por ser la primera vez que empleamos estos métodos en estudio. Por ejemplo, tuvimos que adaptar las funciones que están en el marco teórico, las cuales están ideadas en general para listas de elementos, ya que nuestro trabajo utiliza listas de diccionarios y ese fue el primer desafío a sortear.

Por otra parte, utilizamos dos métodos de búsqueda: *Lineal* y *Binaria*, donde corroboramos su teoría. En este sentido podemos decir que su eficiencia para procesar listas de Datos depende del tamaño, lo cual se reflejó en los tiempos de ejecución respectivos y entendimos así porqué utilizar un método sobre otro según el caso.

Cuando llegó el momento de aplicar métodos de ordenamiento, elegimos implementar y comparar dos enfoques: QuickSort y Bubble Sort, utilizando listas de diccionarios con distintos tamaños (una de 1.000 elementos y otra de más de 10.000). Esta decisión nos permitió observar en la práctica cómo la eficiencia algorítmica influye directamente en el rendimiento del programa.

A través de las pruebas, confirmamos que Bubble Sort, aunque sencillo de implementar y útil en contextos educativos, se vuelve rápidamente ineficiente en volúmenes grandes de datos, presentando tiempos de ejecución significativamente altos (hasta 14 segundos a favor del QuickSort en la lista grande). En cambio, QuickSort (o algoritmos optimizados similares como el que utiliza la función `sorted()` de Python), mostró un comportamiento mucho más estable y rápido, incluso ante grandes cantidades de productos.

Esta experiencia nos permitió comprender la importancia de elegir el método de ordenamiento adecuado según el contexto, destacando cómo una decisión de diseño algorítmico puede impactar en la escalabilidad y usabilidad de una solución en entornos reales.

Finalmente, dejamos registro visual de las pruebas realizadas mediante capturas de pantalla de la terminal de Visual Studio Code, donde se pueden observar los tiempos de ejecución de cada método de ordenamiento y búsqueda aplicados. - Ver Anexo, Imágenes 11, 12 y 13-

Estas evidencias se incluyen en el anexo del informe, junto con el enlace al repositorio del proyecto en GitHub, donde se encuentra disponible todo el código fuente:

<https://github.com/nicolspannunzio/TPi-Programacion.I>

6. Conclusiones

A lo largo del desarrollo del trabajo integrador, pudimos afianzar conocimientos clave sobre los algoritmos de búsqueda y ordenamiento de datos en Python, así como también comprender la importancia de estos procesos dentro del mundo de la programación. Al trabajar con estructuras de datos realistas como listas de diccionarios, se adquirió una mirada más práctica sobre cómo aplicar estos algoritmos en situaciones concretas.

Uno de los principales aprendizajes fue reconocer que la elección del algoritmo adecuado no es un detalle menor, sino una decisión de diseño fundamental, que incide directamente en la eficiencia del código. Aprendimos a analizar el contexto y las características de los datos para decidir, por ejemplo, cuándo usar búsqueda lineal o binaria, o cuándo conviene utilizar ordenamiento rápido frente a burbuja.

Desde una perspectiva aplicada, estos conocimientos resultan de gran utilidad para proyectos futuros en los que se trabaje con grandes volúmenes de datos, ya sea en entornos educativos, sistemas comerciales, aplicaciones web o herramientas de análisis. Comprender cómo optimizar la forma en que se accede y se organiza la información es una habilidad transversal y muy valorada en el desarrollo de software.

Durante el proceso surgieron diversas dificultades, especialmente relacionadas con adaptar los algoritmos del marco teórico a un catálogo de productos basado en diccionarios. También tuvimos que investigar cómo generar datos de prueba realistas y cómo medir tiempos de ejecución para validar el rendimiento de cada método. En este sentido, una mejora importante fue reemplazar la función `time.time()` por `time.perf_counter()`, ya que detectamos que la primera arrojaba resultados poco precisos. Esta modificación nos permitió obtener mediciones más confiables y afinar nuestro análisis comparativo entre algoritmos.

Estas barreras fueron superadas con búsqueda activa de información, trabajo colaborativo constante y mucha práctica en la escritura y corrección de código.

Como posibles mejoras o extensiones futuras, se podría ampliar el sistema de búsqueda incorporando filtros combinados (por categoría y rango de precios), implementar una interfaz visual básica para simular la experiencia de un e-commerce real, o incluso integrar bases de datos para gestionar el catálogo dinámicamente.

En síntesis, el trabajo permitió consolidar conceptos fundamentales, ejercitar habilidades de codificación, y sobre todo, entender el valor de un diseño algorítmico eficiente y contextualizado. La experiencia resultó sumamente enriquecedora tanto desde lo técnico como desde lo colaborativo.

7. Bibliografía

Wachenchauzer, R., Manterola, M., Curia, M., Medrano, M., & Paez, N. (2012). *Algoritmos y Programación I: Aprendiendo a programar usando Python como herramienta* [Libro académico]. Licencia Creative Commons BY-SA 2.5 Argentina.

Marzal, A., & García, I. (2009). *Introducción a la programación con Python* [Libro académico]. Licencia CC BY-NC-SA 2.5 ES.

Faker. (s.f.). *Faker Documentation*. <https://faker.readthedocs.io/en/master/index.html>

El Programador Chapuzas. (2022, diciembre 22). *Algoritmos en Python: Búsqueda binaria*. <https://programacionpython80889555.wordpress.com/2021/12/22/algoritmos-en-python-busqueda-binaria/>

StrataScratch. (s.f.). *Easy Guide to Python Hashmaps*. <https://www.stratascratch.com/blog/easy-guide-to-python-hashmaps/>

Eldridge, S. (2025, mayo 16). *Sorting algorithm*. Britannica. <https://www.britannica.com/technology/sorting-algorithm>

Programiz. (s.f.). *Counting Sort – Data Structure and Algorithms*. <https://www.programiz.com/dsa/counting-sort>

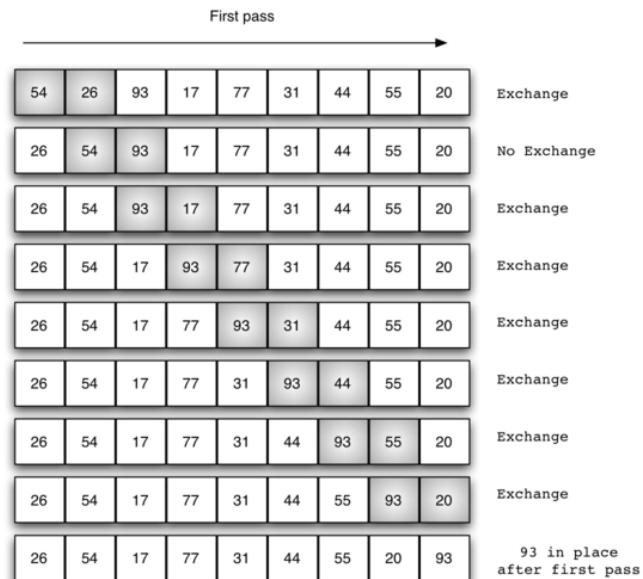
WsCubeTech. (s.f.). *Radix Sort – DSA*. <https://www.wscubetech.com/resources/dsa/radix-sort>

Fernández, O. (2025, enero 14). *Big-O para principiantes*. Aprender Big Data. <https://aprenderbigdata.com/big-o/>

8. Anexo

Figura 1

Comparaciones e intercambios en el algoritmo de **ordenamiento burbuja**.

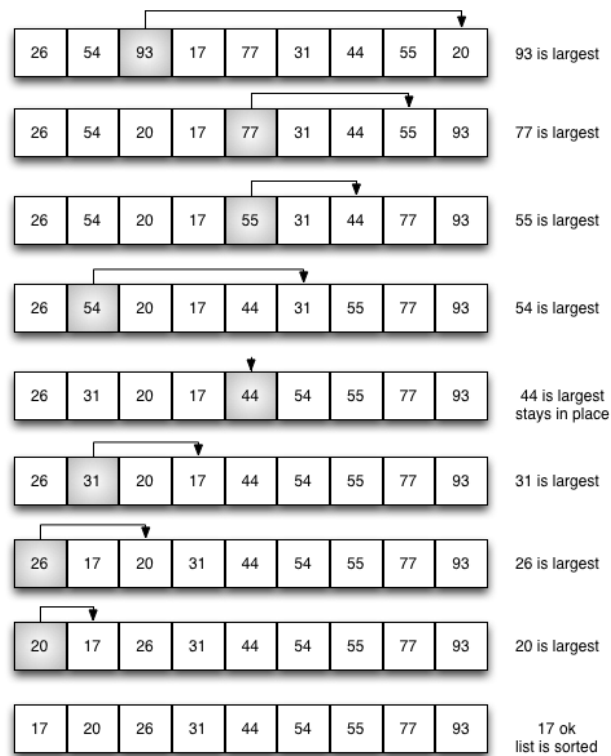


Nota. Tomado de “El ordenamiento burbuja”, por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoBurbuja.html>

Figura 2

Comparaciones e intercambios en el algoritmo de **ordenamiento por selección**.

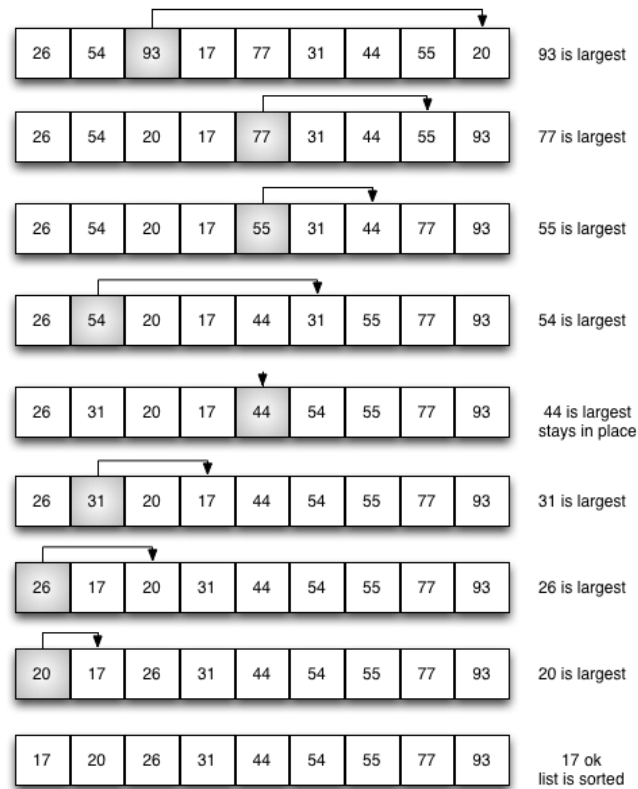


Nota. Tomado de “El ordenamiento por selección”, por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoPorSeleccion.html>

Figura 3

Visualización del algoritmo **Ordenamiento por Inserción**.

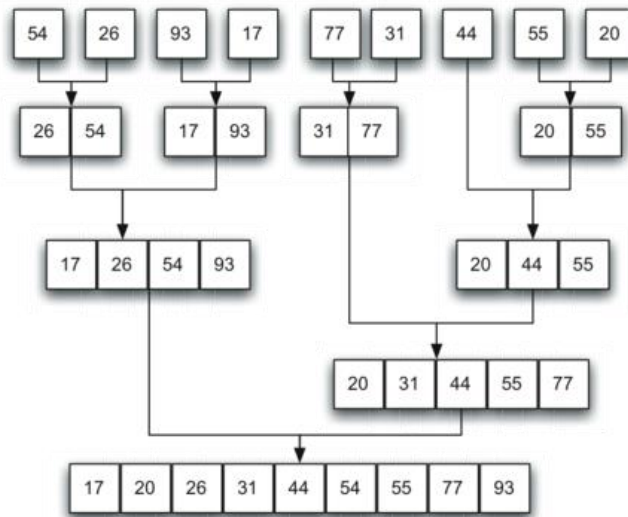


Nota. Tomado de "El ordenamiento por inserción", por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoPorInsercion.html>

Figura 4

Visualización de la “fusión” de las listas (ver de arriba hacia abajo).

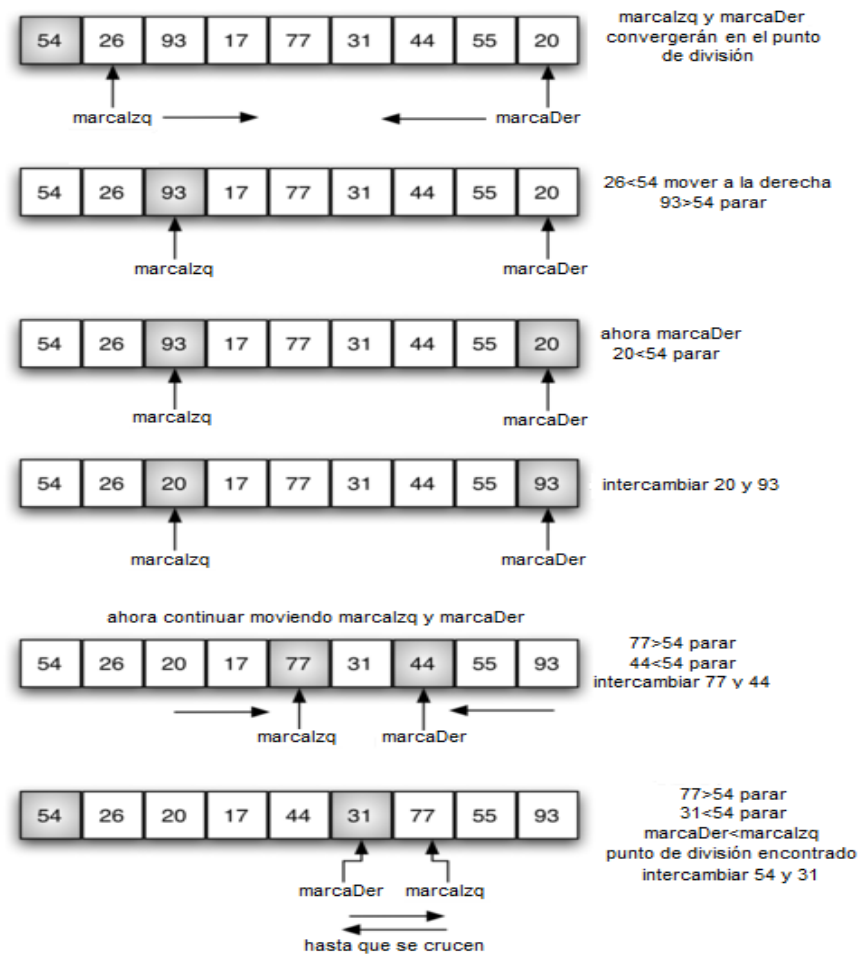


Nota. Tomado de “El ordenamiento mezcla”, por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoPorMezcla.html>

Figura 5

Reorganización de la lista alrededor del pivote



Nota. Tomado de "El ordenamiento rápido", por Runestone Academy, s.f.

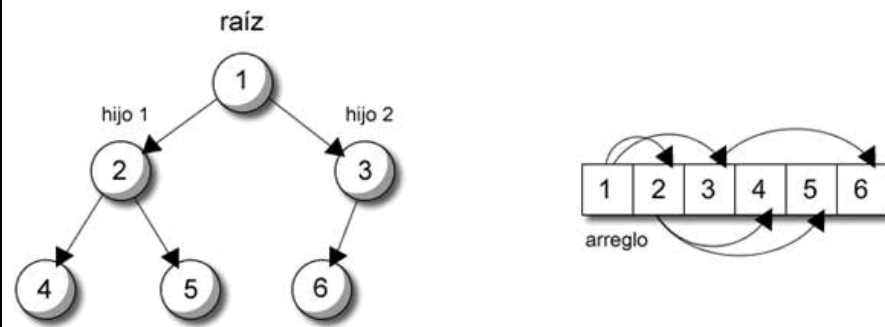
<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoRapido.html>

Figura 6

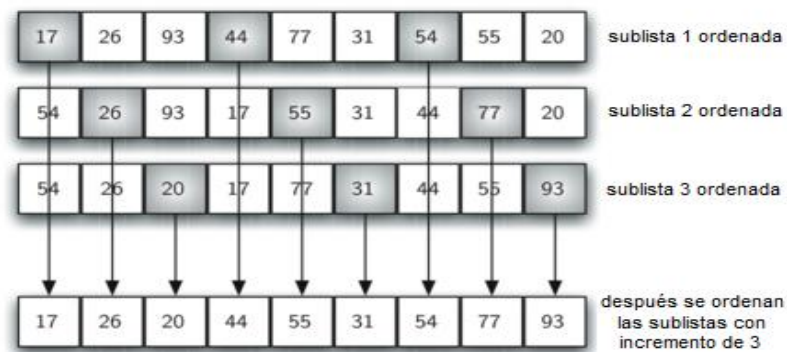
Dinámica del ordenamiento por montículos

Nota. Tomado de *Ordenación por montículos (Heapsort)*, por Pier Paolo Guillen, s.f.

<https://pier.guillen.com.mx/algorithms/03-ordenacion/03.4-heapsort.htm>



a)



b)

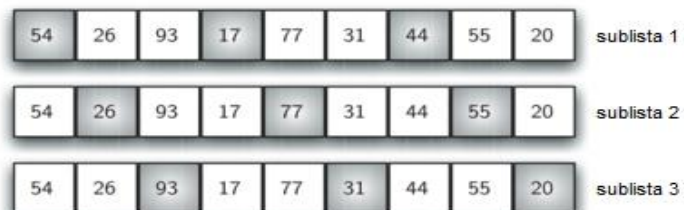


Figura 7.1

Visualización del ordenamiento Shell (**a**- Ordenamiento shell con incrementos de 3 y **b**- Ordenamiento de shell después de ordenar cada sublista)

Nota. Tomado de “El ordenamiento Shell”, por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoDeShell.html>

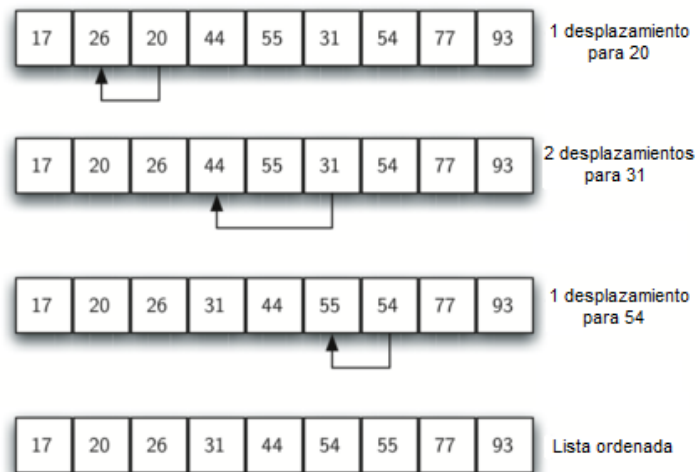


Figura 7.2

Visualización del ordenamiento Shell (inserción final con incremento de 1)

Nota. Tomado de “El ordenamiento Shell”, por Runestone Academy, s.f.

<https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoDeShell.html>

Figura 9 - Ordenamiento por conteo

<https://www.programiz.com/dsa/counting-sort>

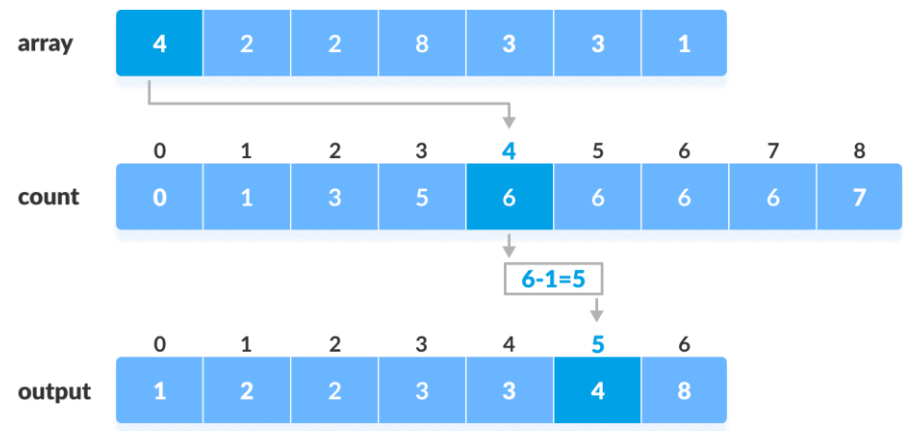


Figura 10 - Ordenamiento radix

<https://www.programiz.com/dsa/radix-sort>

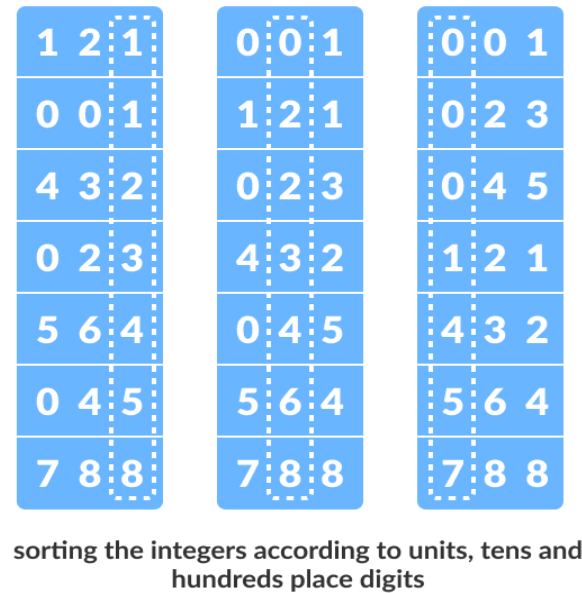
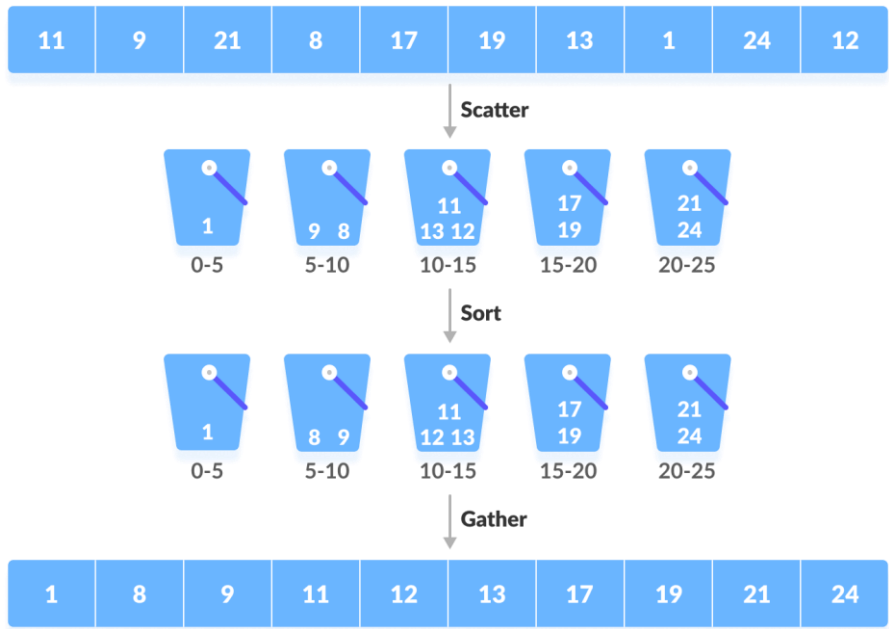
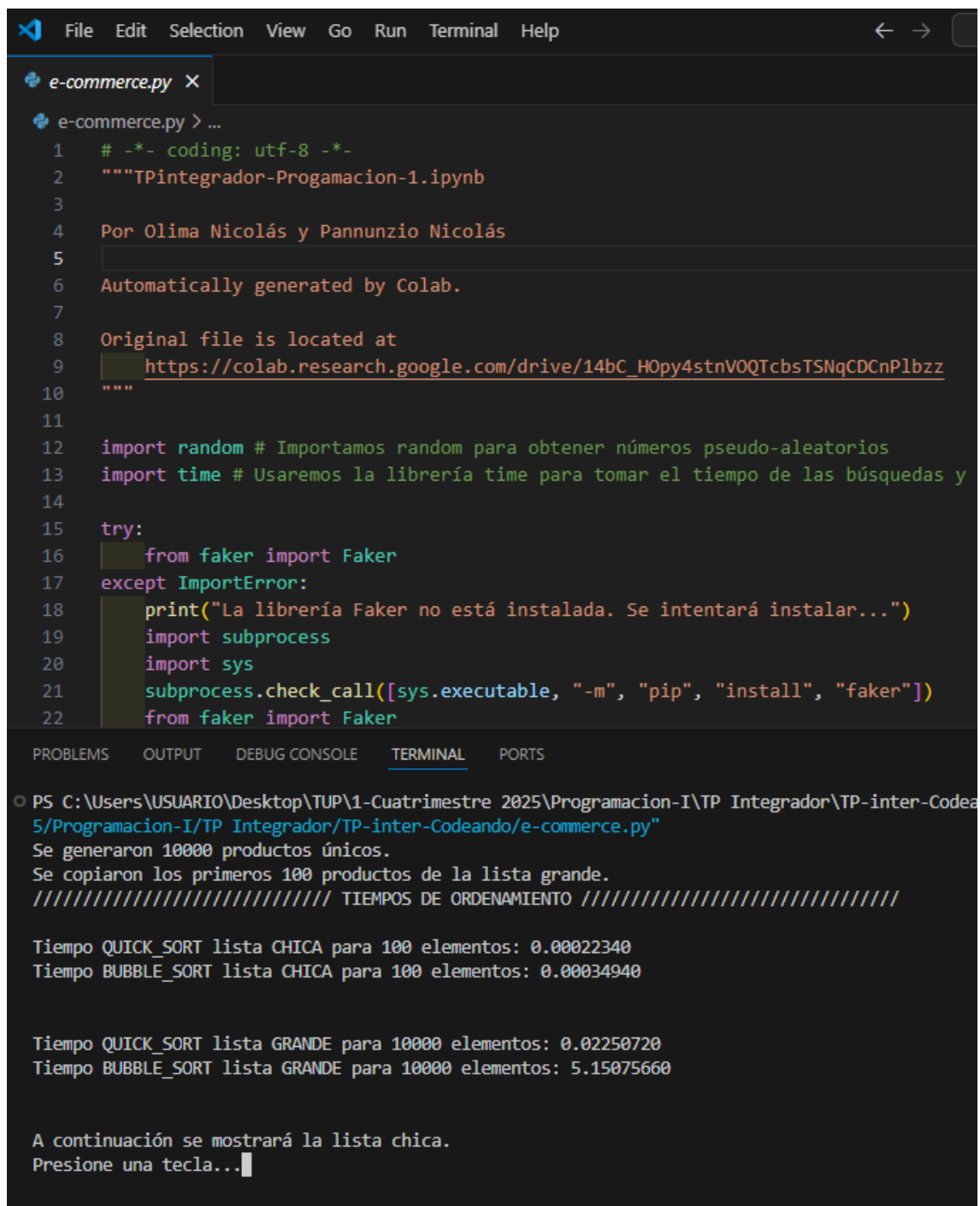


Figura 11 - Ordenamiento cubos o cubetas

<https://www.programiz.com/dsa/bucket-sort>





The image shows a Visual Studio Code window with a Python file named `e-commerce.py` open. The file contains a docstring and Python code for generating products and testing sorting algorithms. The terminal output shows the execution of the script, including the generation of 10,000 unique products and the timing of sorting algorithms (QUICK_SORT and BUBBLE_SORT) on both a small (100) and large (10,000) list.

```

1  # -*- coding: utf-8 -*-
2  """TPintegrador-Programacion-1.ipynb
3
4  Por Olima Nicolás y Pannunzio Nicolás
5
6  Automatically generated by Colab.
7
8  Original file is located at
9  https://colab.research.google.com/drive/14bC_H0py4stnVOQTcbsTSNqCDCnPlbzz
10 """
11
12 import random # Importamos random para obtener números pseudo-aleatorios
13 import time # Usaremos la librería time para tomar el tiempo de las búsquedas y
14
15 try:
16     from faker import Faker
17 except ImportError:
18     print("La librería Faker no está instalada. Se intentará instalar...")
19     import subprocess
20     import sys
21     subprocess.check_call([sys.executable, "-m", "pip", "install", "faker"])
22     from faker import Faker
  
```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

PS C:\Users\USUARIO\Desktop\TUP\1-Cuatrimestre 2025\Programacion-I\TP Integrador\TP-inter-Codea
5/Programacion-I/TP Integrador/TP-inter-Codeando/e-commerce.py"
Se generaron 10000 productos únicos.
Se copiaron los primeros 100 productos de la lista grande.
////////////////////////////////// TIEMPOS DE ORDENAMIENTO ////////////////////////////////////

Tiempo QUICK_SORT lista CHICA para 100 elementos: 0.00022340
Tiempo BUBBLE_SORT lista CHICA para 100 elementos: 0.00034940

Tiempo QUICK_SORT lista GRANDE para 10000 elementos: 0.02250720
Tiempo BUBBLE_SORT lista GRANDE para 10000 elementos: 5.15075660

A continuación se mostrará la lista chica.
Presione una tecla...
  
```

```

Nombre: Smartphone Labore-8831 || Precio: $ 23488.19 || Stock: 944 || Categoría: Electrónica
Nombre: Smartwatch Sunt-9286 || Precio: $ 38343.91 || Stock: 583 || Categoría: Electrónica
Nombre: Snacks proteicos Amet-7268 || Precio: $ 68830.71 || Stock: 194 || Categoría: Alimentación
Nombre: Sofá cama Distinctio-9474 || Precio: $ 13446.69 || Stock: 449 || Categoría: Muebles
Nombre: Sofá cama Mouth-7690 || Precio: $ 45909.71 || Stock: 331 || Categoría: Muebles
Nombre: Sofá cama Rich-2391 || Precio: $ 63429.02 || Stock: 309 || Categoría: Muebles
Nombre: Sofá cama Tempora-797 || Precio: $ 83822.0 || Stock: 534 || Categoría: Muebles
Nombre: Sofá cama Vel-9779 || Precio: $ 35314.33 || Stock: 754 || Categoría: Muebles
Nombre: Tablet Consequuntur-7093 || Precio: $ 28444.68 || Stock: 513 || Categoría: Electrónica
Nombre: Tablet Focus-5261 || Precio: $ 71426.86 || Stock: 257 || Categoría: Electrónica
Nombre: Tablet Inventore-622 || Precio: $ 57991.35 || Stock: 554 || Categoría: Electrónica
Nombre: Tablet Numquam-6823 || Precio: $ 97154.2 || Stock: 995 || Categoría: Electrónica
Nombre: Tablet Repellat-8016 || Precio: $ 70969.56 || Stock: 130 || Categoría: Electrónica
Nombre: Vestido Letter-3106 || Precio: $ 10991.13 || Stock: 141 || Categoría: Ropa
Nombre: Zapatillas Across-7323 || Precio: $ 73280.81 || Stock: 154 || Categoría: Ropa

```

Ingrese el nombre del producto: Café gourmet Social-68

Ingrese el nombre del producto: Café gourmet Social-68

Busqueda LINEAL lista chica

Producto encontrado en la posición 16

Nombre: Café gourmet Social-68 || Precio: \$53239.94 || Stock: 806 || Categoría: Alimentación

Busqueda LINEAL lista grande

Producto encontrado en la posición 1300

Nombre: Café gourmet Social-68 || Precio: \$53239.94 || Stock: 806 || Categoría: Alimentación

Busqueda binaria lista chica

Producto encontrado en la posición 16

Nombre: Café gourmet Social-68 || Precio: \$53239.94 || Stock: 806 || Categoría: Alimentación

Busqueda binaria lista grande

Producto encontrado en la posición 1300

Nombre: Café gourmet Social-68 || Precio: \$53239.94 || Stock: 806 || Categoría: Alimentación

////////// TIEMPOS DE BÚSQUEDA //////////

Tiempo BÚSQUEDA LINEAL lista CHICA: 0.00055440

Tiempo BÚSQUEDA BINARIA lista CHICA: 0.00036140

Tiempo BÚSQUEDA LINEAL lista GRANDE: 0.00097700

Tiempo BÚSQUEDA BINARIA lista GRANDE: 0.00028310

© PS C:\Users\USUARIO\Desktop\TUP\1-Cuatrimestre 2025\Programacion-I\TP Integrador\TP-inter-Codeando>