**ESCUELA POLITÉCNICA NACIONAL**

ESCUELA DE FORMACIÓN DE TECNÓLOGOS

**ALGORITMOS Y ESTRUCTURAS DE DATOS (TSDS)**

|  |  |
| --- | --- |
|  | |
| ASIGNATURA: | ALGORITMOS Y ESTRUCTURAS DE DATOS |
| PROFESOR: | Ing. Lorena Chulde MSc. |
| PERÍODO ACADÉMICO: | 2023-B |
|  | |

**TALLER - TAREA**

**Grupal**

|  |
| --- |
|  |
| **TÍTULO:**  **DICCIONARIOS**  **Nombres de los estudiantes:**  Guerra Lovato Josué Eduard  Soria Ansa Richard Mauricio  Pérez Orosco Carlos David |
|  |
|  |
|  |
|  |
|  |

**2023-B**

**PROPÓSITO DE LA TAREA**

Aplicar diccionario mediante funciones para implementar un CRUD

**OBJETIVO GENERAL**

Desarrollar habilidades de programación en Python mediante la práctica de implementar algoritmos de ordenamiento como Quicksort y algoritmos de selección, así como también explorar y comprender el uso de estructuras de datos como diccionarios y conjuntos para resolver problemas de manipulación y gestión de datos.

**Parte I**

Replicar los ejercicios del taller, capturar las pantallas y pegarlas en cada ejercicio

**GUIA DEL TALLER:**

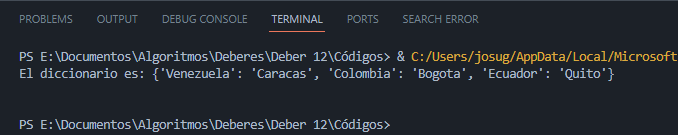
**EJERCICIOS CON DICCIONARIOS**

#Declarar el diccionario con clave y valor

diccionario = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito"}

print(diccionario)

Ejecución:

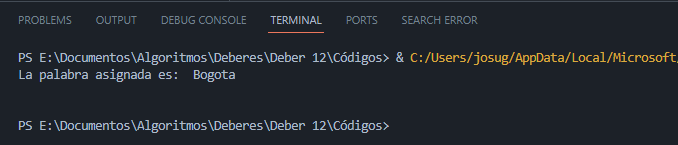


#Acceder a un valor

diccionario = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito"}

print(diccionario["Colombia"])

Ejecución:

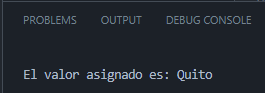


#Acceder a un valor

valor = diccionario["Ecuador"]

print(valor)

Ejecución:

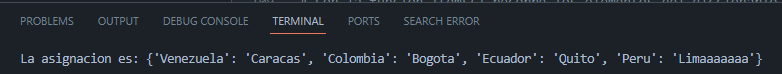


#Agregar un elemento

diccionario["Peru"] = "Limaaaaaaa"

print(diccionario)

Ejecución:

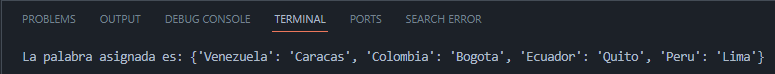


#Modificar un elemento

diccionario["Peru"] ="Lima"

print(diccionario)

Ejecución:



#Eliminar un elemento con del

del diccionario["Venezuela"]

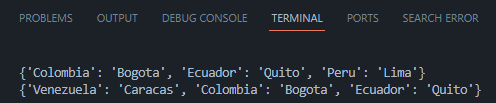
print(diccionario)

diccionario2 = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito", "Peru":"Lima"}

del(diccionario2["Peru"])

print(diccionario2)

Ejecución:



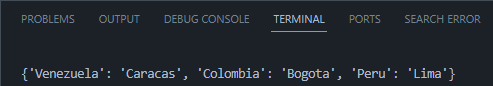
#Eliminar un elemento con la funcion pop

diccionario = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito", "Peru":"Lima"}

diccionario.pop("Ecuador")

print(diccionario)

Ejecución:



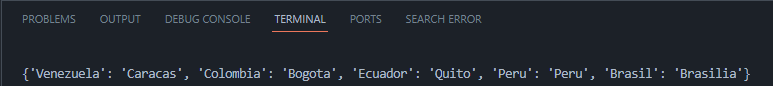
#con tuplas

tupla = ["Venezuela", "Colombia", "Ecuador", "Peru", "Brasil"]

dicPaises = {tupla[0]:"Caracas", tupla[1]:"Bogota", tupla[2]:"Quito", tupla[3]:"Peru", tupla[4]:"Brasilia" }

print(dicPaises)

Ejecución:

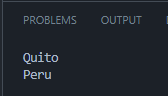


#Acceder un elemento en concreto

print(dicPaises[tupla[2]])     # por el indice de la tupla

print(dicPaises["Peru"])       # por el valor de la tupla

Ejecución:



#Crear diccionarios con diferentes tipo de datos

jugadores = {"nombre":"Juan","apellido":"Perez","edad":20,"partidosJugados":5,"partidosGanados":3}

print(jugadores)

Ejecución:

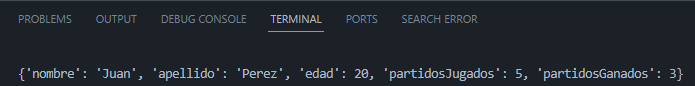


#con valores de tipo tupla

jugador = {"nombre":"Juan","apellido":"Perez","edad":25,"partidosJugados":5,"partidosGanados":3, "detalleAnios":[2020, 2021, 2022, 2023, 2024]}

print("Los partidos se jugaron en los años:  ", jugador["detalleAnios"])

Ejecución:

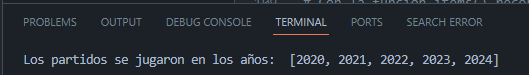


#un diccionario puede contener otro diccionario

jugador = {"nombre":"Juan","apellido":"Perez","edad":25,"partidosJugados":5,"partidosGanados":3, "detalleAnios":{"temporadas":[2020, 2021, 2022, 2023, 2024]}}

print("Los partidos se jugaron en los años:  ", jugador["detalleAnios"])

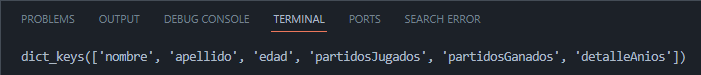
Ejecución:



#Consultar las claves del diccionario

print(jugador.keys())

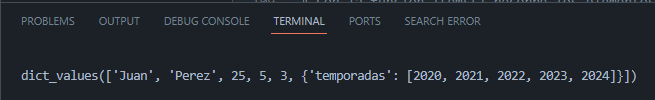
Ejecución:



#Consultar los valores del diccionario

print(jugador.values())

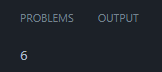
Ejecución:



#Consultar la longitud del diccionario

print(len(jugador))

Ejecución:



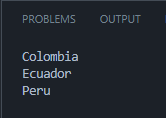
#Recorrer el diccionario con for e imprimir claves

diccionario\_paises = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito", "Peru":"Lima"}

for pais in diccionario\_paises:

    print(pais)

Ejecución:



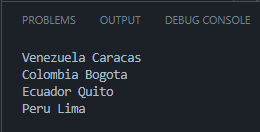
#Recorrer el diccionario con for e imprimir clave y valor

diccionario\_paises = {"Venezuela": "Caracas", "Colombia":"Bogota", "Ecuador":"Quito", "Peru":"Lima"}

for clave, valor in diccionario\_paises.items():

    print(clave, valor)

Ejecución:



**CRUD CON DICCIONARIOS**

#Dicionario que permite gestionar notas

notas = {}

def agregarNota(notas, titulo, contendo):

    notas[titulo] = contenido

def verNotas(notas):

    if not notas:

        print("No existe ninguna nota ")

    else:

        for titulo, contenido in notas.items():

            print(f"     {titulo}    ")

            print(contenido)

def editarNota(notas, titulo, nuevoContenido):

    if titulo in notas:

        notas[titulo] = nuevoContenido

        print("La nota se actualizo exitosamente")

    else:

        print("La nota consulatda no se encuentra en el registro")

def eliminarNota(notas, titulo):

    if titulo in notas:

        del notas[titulo]

        print("La nota se elimino exitosamente")

    else:

        print("La nota que desea eliminar no se encuentra en el registro")

#MENU

while True:

    print("--------- BIENVENIDOS AL REGISTRO DE NOTAS ---------")

    print("1. Agregar nota")

    print("2. Ver nota")

    print("3. Editar nota")

    print("4. Eliminar nota")

    print("5. Salir \n")

    opcion = int(input("Ingrese la opcion a realizar: "))

    if opcion == 1:

        titulo = input("Ingrese el titulo de la nota: ")

        contenido = input("Ingrese el contenido de la nota: ")

        agregarNota(notas, titulo, contenido)

        print("La nota se agrego exitosamente")

    elif opcion == 2:

        verNotas(notas)

    elif opcion == 3:

        titulo = input("Ingrese el titulo de la nota que desea editar: ")

        nuevoContenido = input("Ingrese el nuevo contenido de la nota: ")

        editarNota(notas, titulo, nuevoContenido)

    elif opcion == 4:

        titulo = input("Ingrese el titulo de la nota que desea editar: ")

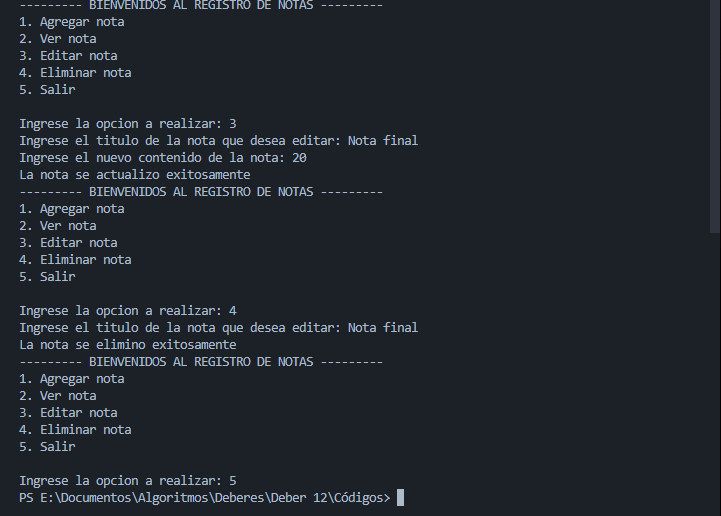
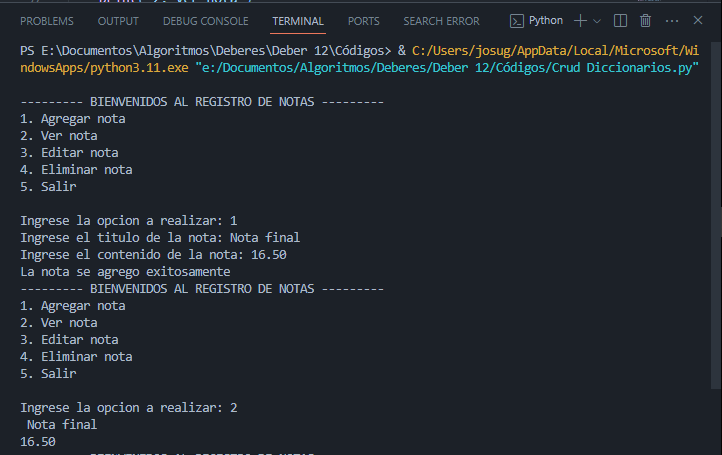
        eliminarNota(notas, titulo)

    elif opcion == 5:

        break

    else:

        print("La opcion ingresada no es valida")

****Ejecución del programa:

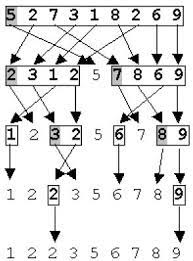
**Algoritmo de ordenamiento Quicksort**

En qué consiste el Algoritmo de ordenamiento Quicksort:

El método Quick Sort es actualmente el más eficiente y veloz de los métodos de ordenación interna. Es también conocido con el nombre del método rápido y de ordenamiento por partición.

Para implementar este algoritmo se usan listas auxiliares vamos a aplicar la estrategia divide y vencerás sobre la lista para ir dividiendo la lista en partes más pequeñas y así tener que llevar a cabo menos comparaciones. Para esto se elige un elemento de la lista, a partir de este elemento se va a dividir la lista en los mayores y los menores.

El elemento se llama pivote y va el primer elemento de la lista. Se recorre el resto de los elementos para compararlos con el pivote si son mayores se los ubica a la derecha si son menores se los ubica a la izquierda.



**Ejemplo:**

Tenemos la siguiente lista con valores desordenados:

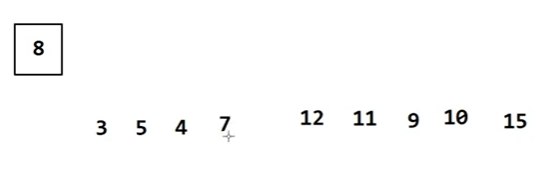
valores = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]

Elemento pivote el 8, tomamos el primer elemento:

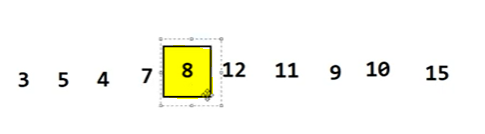


Ahora tomamos los elementos y los comparamos con el pivote (8):

* El 12 como es mayor lo ponemos a la derecha.
* El 3 es menor va a la izquierda
* El 11 es mayor va a la derecha
* El 5 es menor va a la izquierda
* El 9 es mayor va a la derecha
* El 4 es menor va a la izquierda
* El 15 es mayor va a la derecha
* El 7 es menor va a la izquierda



Finalmente, el pivote (8) lo ubicamos en el centro



Queda hacerle recursivo puesto que las instrucciones se han ejecutado una vez obteniendo el resultado de la figura anterior.

Al hacerle recursiva se ejecuta la misma operación con las dos partes de la lista hasta llegar al caso base.

**Caso Base:**

Que la lista tenga un solo elemento con lo cual ya estará colocado o con una lista que no tenga elementos.

**Ejemplo:**

Crear una función auxiliar dividir para particionar la lista; toma como parámetro la lista y que va a dividir la lista en dos partes a partir de un pivote que se tome en este caso es el primer elemento de la lista.

lista = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]

pivote = lista[0]

Vamos a necesitar dos listas auxiliares donde guardar los elementos menores y los elementos mayores. Declaramos dos listas vacías una para menores y otra para mayores.

lista = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]

def dividir(lista):

    pivote = lista[0]

    menores = []

    mayores = []

Recorremos el resto de los elementos de la lista con for, desde el 1 hasta la longitud de la lista, puesto que el primero ya no se lo recorre porque es el pivote.

Comparamos si el elemento con el que vamos iterando es menor que el pivote entonces a la lista menor se le añade el elemento caso contrario se le añade a la lista mayores o si es igual también se lo incluye en mayores. Retorna la lista de elementos menores, mayores y el pivote.

lista = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]

def dividir(lista):

    pivote = lista[0]

    menores = []

    mayores = []

for i in range(1, len(lista)):

    if lista[i] < pivote:

        menores.append(lista[i])

    else:

        mayores.append(lista[i])

return menores, pivote, mayores

Mediante esta función hemos divido la lista en dos partes a partir del pivote

de tal forma que los menores ya no hay que compararlos con los mayores estamos aplicando la técnica divide y vencerás.

Hemos dividido la lista en dos partes de tal forma que los menores ya no hay que compararlos con los mayores, ni los mayores con los menores, por lo tanto reducimos en una parte considerable las comparaciones.

A la lista menores volvemos a llevar a realizar la misma operación, la volvemos a dividir en dos partes volviéndose a reducir de nuevo las comparaciones. Una vez que hacemos esto se trata de aplicar la función de forma recursiva a las listas auxiliares.

Definimos la función “quicksort” que va a tomar la lista y si la longitud es menor que dos, es decir si tiene solo un elemento o 0 elementos llegamos al caso base y retorna la lista, caso contrario llamamos a la función “dividir” para dividir la lista en dos listas más pequeñas y un pivote, así que menores pivote y mayores va a ser igual a dividir(lista). Esta función devuelve la concatenación de estos tres elementos, por lo que ya se tiene la lista ordenada de tres elementos.

def quicksort(lista):

    if len(lista) < 2:

        return lista

    menores, pivote, mayores = dividir(lista)

    return menores + [pivote] + mayores

**Salida:**

lista = [8, 12, 3]



Pero si la lista tiene más elementos como la del ejemplo.

lista = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]



Lo que hace es una pasada por eso se queda en el estado de la figura, el pivote 8 se coloca al centro y los menores a la izquierda y lo mayores a la derecha.

Para que recorra hasta el final de los elementos, debemos llamar a la función Quicksort en la lista menores y la lista mayores.

def quicksort(lista):

    if len(lista) < 2:

        return lista   #devulve los elementos ordenados porque solo tiene 1 o ninguno

    menores, pivote, mayores = dividir(lista)

    return quicksort(menores) + [pivote] + quicksort(mayores)

Ahora si se ordenan todos los elementos:



**Código completo:**

lista = [8, 12, 3, 11, 5, 9, 10, 4, 15, 7]

#lista = [8, 12,3]

def dividir(lista):

    pivote = lista[0]

    menores = []

    mayores = []

    for i in range(1, len(lista)):

        if lista[i] < pivote:

            menores.append(lista[i])

        else:

            mayores.append(lista[i])

    return menores, pivote, mayores

def quicksort(lista):

    if len(lista) < 2:

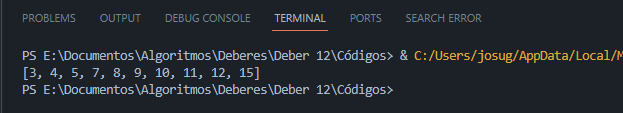
        return lista   #devulve los elementos ordenados porque solo tiene 1 o ninguno

    menores, pivote, mayores = dividir(lista)

    return quicksort(menores) + [pivote] + quicksort(mayores)

print(quicksort(lista))

**Ejecución:**

****

**Parte II**

**TAREA:**

Consultar sobre conjuntos

**CONJUNTOS EN PYTHON**

Es una colección desordenada de elementos únicos. Los conjuntos son útiles cuando se necesita almacenar una colección de elementos sin duplicados sin importar el orden en que se almacenan los elementos.

Los conjuntos en Python se definen utilizando llaves {} y los elementos se separan por comas. Por ejemplo:

1. mi\_conjunto = {1, 2, 3, 4, 5}

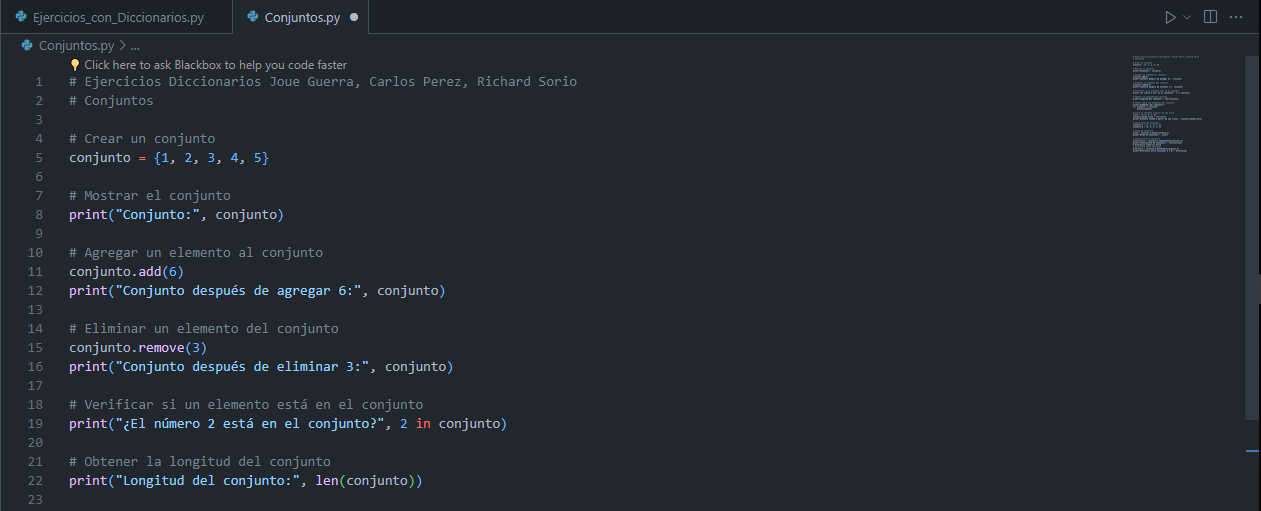
También se pueden crear conjuntos vacíos utilizando la función set():

1. conjunto\_vacio = set()

Los conjuntos tienen varias características útiles, como la capacidad de

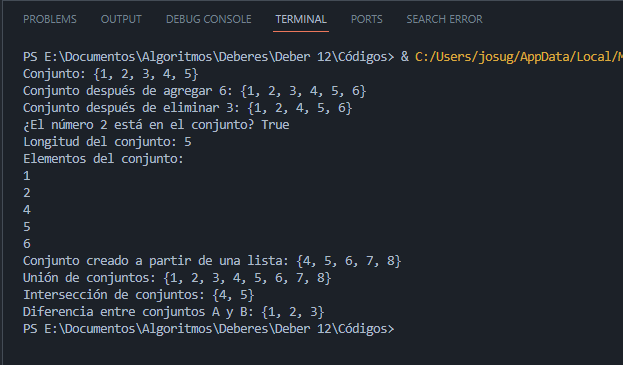
realizar operaciones de conjuntos como unión, intersección, diferencia y diferencia simétrica. También son eficientes en términos de búsqueda de elementos, ya que utilizan tablas hash internamente para almacenar los elementos únicos.

Es importante tener en cuenta que los conjuntos no conservan el orden de inserción de los elementos, por lo que no se puede acceder a los elementos mediante índices como en una lista. Además, los conjuntos no admiten elementos mutables como listas o diccionarios, pero sí admiten elementos inmutables como cadenas, números y tuplas [1].

EJEMPLO:

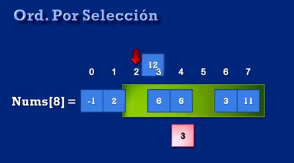


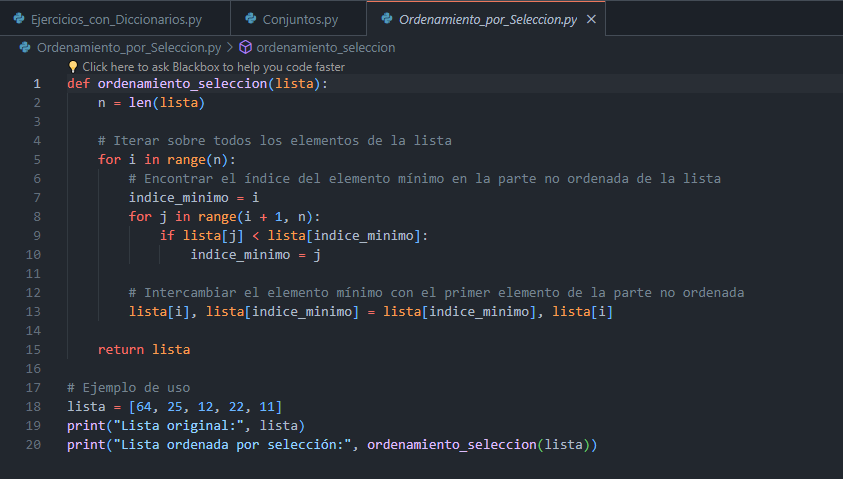
Ejecución del programa de ejemplo:



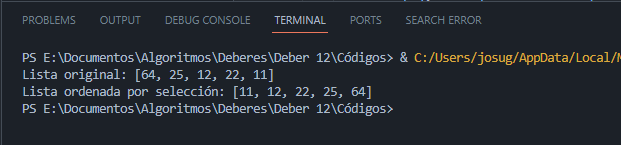
Algoritmos de ordenamiento por selección

**ALGORITMO DE ORDENAMIENTO POR SELECCIÓN**

Es un algoritmo simple e intuitivo que consiste en seleccionar repetidamente el elemento más pequeño (o más grande, dependiendo del orden deseado) de una lista no ordenada y colocarlo al principio (o al final) de la lista ordenada. Este proceso se repite hasta que toda la lista esté ordenada [2].

#Ejemplo:

Ejecución del programa de ejemplo:



**ENTREGABLES:**

* Una vez culminada tu tarea, capturar las pantallas de la ejecución de cada ejercicio con tus datos y súbela en el apartado del aula virtual “S15-Taller-Tarea
* Subir los ejercicios al git o al drive y entrega la url de los archivos .py o, a su vez, entregue el archivo.
* Recordar que el nombre del archivo deberá ser: S15\_Taller-Tarea\_Algoritmos\_2023B\_NApellido(de todos los integrantes)

RECURSOS NECESARIOS

* Acceso a Internet.
* Imaginación.
* VSC

RECOMENDACIONES:

Se recomienda dedicar tiempo regularmente a practicar la implementación de algoritmos de ordenamiento, como Quicksort y algoritmos de selección, en diferentes contextos y con diversas variaciones de datos. Además, es fundamental dedicar tiempo a comprender en profundidad el funcionamiento de estas estructuras de datos y algoritmos, así como a explorar casos de uso reales donde se puedan aplicar de manera efectiva. Se sugiere también aprovechar recursos en línea, como tutoriales, cursos y comunidades de programadores, para obtener orientación y apoyo adicional durante el proceso de aprendizaje.

CONCLUSIONES:

El desarrollo de habilidades de programación en Python a través de la práctica de implementar algoritmos de ordenamiento y explorar estructuras de datos como diccionarios y conjuntos es fundamental para mejorar la capacidad de resolver problemas de manipulación y gestión de datos de manera eficiente. Al dedicar tiempo y esfuerzo a comprender en profundidad estos conceptos y practicar su implementación en diversos escenarios, se adquiere una base sólida para abordar problemas más complejos en el futuro y para desarrollar soluciones efectivas en el ámbito de la programación.

ENLACES:

Enlace GitHub:

<https://github.com/JosueGuerra2023B/Estructuras-Datos2023B/tree/master/Deberes/Deber%2012>

# Bibliografía

|  |  |
| --- | --- |
| [1] | A. H. Gonzales, «Recursos Python,» 29 06 2020. [En línea]. Available: https://recursospython.com/guias-y-manuales/conjuntos-sets/. [Último acceso: 2024 03 01]. |
| [2] | parzibyte, «Parzibyte's blog,» 23 05 2021. [En línea]. Available: https://parzibyte.me/blog/2021/05/23/python-ordenamiento-seleccion/. [Último acceso: 03 01 2024]. |
| [3] | H. E. HERRERA Monterroso, septiembre 2012. [En línea]. Available: http://goo.gl/JcqRY. |
| [4] | M. ATEHORT, marzo 2002. [En línea]. Available: http://goo.gl/HbbfD. |