|  |  |  |
| --- | --- | --- |
|  | **Carátula para entrega de prácticas** | |
| Facultad de Ingeniería | | Laboratorio de docencia |

Laboratorios de computación

salas A y B



|  |  |
| --- | --- |
| Profesor(a): | M.I. Marco Antonio Martínez Quintana. |
| Asignatura: | Estructura de Datos y Algoritmos I |
| Grupo: | 17 |
| No de Práctica(s): | 11 |
| Integrante(s): | Pineda González Rodrigo |
| No. de Equipo de cómputo empleado: | 26 |
| No. de Lista o Brigada: | 30 |
| Semestre: | 2020-II |
| Fecha de entrega: | 26/04/2020 |
| Observaciones: |  |

CALIFICACIÓN: \_\_\_\_\_\_\_\_\_

# Objetivo:

El objetivo de esta guía es implementar, al menos, dos enfoques de diseño (estrategias) de algoritmos y analizar las implicaciones de cada uno de ellos.

# Introducción:

Un algoritmo es un método para resolver un problema mediante una serie de pasos (acciones, operaciones, instrucciones, órdenes, entre otras.) definidos, precisos y finitos. Esto implica que no tengan divagaciones, sean claros y tengan un fin. Por otro lado, un programa es la solución a un problema inicial. El proceso de programación es el siguiente: Dado un determinado problema el programador debe idear una solución y expresarla usando un algoritmo, luego de esto, debe codificarse en un determinado lenguaje de programación y por último ejecutar el programa en el computador el cual refleja una solución al problema inicial, realizando pruebas de escritorio para de ser necesario, corregir los errores. En Python, es posible implementar una gran cantidad de algoritmos, entre ellos:

## Fuerza bruta:

El objetivo es hacer una búsqueda exhaustiva de todas las posibilidades que lleven a la solución del problema. Un ejemplo de esto es encontrar una contraseña haciendo una combinación exhaustiva de caracteres alfanuméricos generando cadenas de cierta longitud. Sin embargo, si la solución es larga podría tomar un tiempo excesivo.

## Algoritmos ávidos (greedy):

Va tomando una serie de decisiones en un orden específico, una vez que se ha ejecutado esa decisión, ya no se vuelve a considerar. En comparación con fuerza bruta, ésta puede ser más rápida; aunque una desventaja es que la solución que se obtiene no siempre es la más óptima.

## Bottom-up (programación dinámica):

El objetivo de esta estrategia es resolver un problema a partir de subproblemas que ya han sido resueltos. La solución final se forma a partir de la combinación de una o más soluciones que se guardan en una tabla, ésta previene que se vuelvan a calcular las soluciones.

## Top-down:

A diferencia de bottom-up, aquí se empiezan a hacer los cálculos de n hacia abajo. Además, se aplica una técnica llamada memorización la cual consiste en guardar los resultados previamente calculados, de tal manera que no se tengan que repetir operaciones.

## Incremental:

Es una estrategia que consiste en implementar y probar que sea correcto de manera paulatina, ya que en cada iteración se va agregando información hasta completar la tarea.

## Divide y vencerás:

Es una estrategia que consiste en dividir el problema en subproblemas hasta que son suficientemente simples que se pueden resolver directamente. Después las soluciones son combinadas para generar la solución general del problema.

## Modelo RAM:

Cuando se realiza un análisis de complejidad utilizando el modelo RAM, se debe contabilizar las veces que se ejecuta una función o un ciclo, en lugar de medir el tiempo de ejecución.

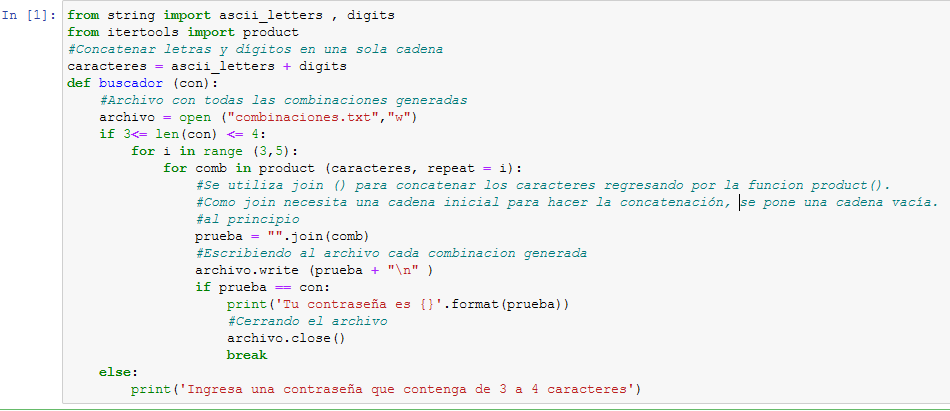
# Repositorio de la práctica:

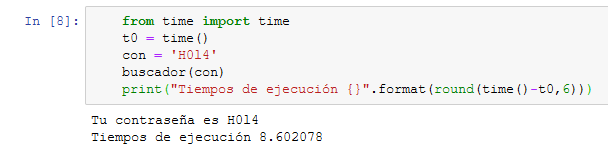
<https://github.com/RodrigoPG-W/Python11-EDA1>

# Desarrollo:

## Fuerza bruta:

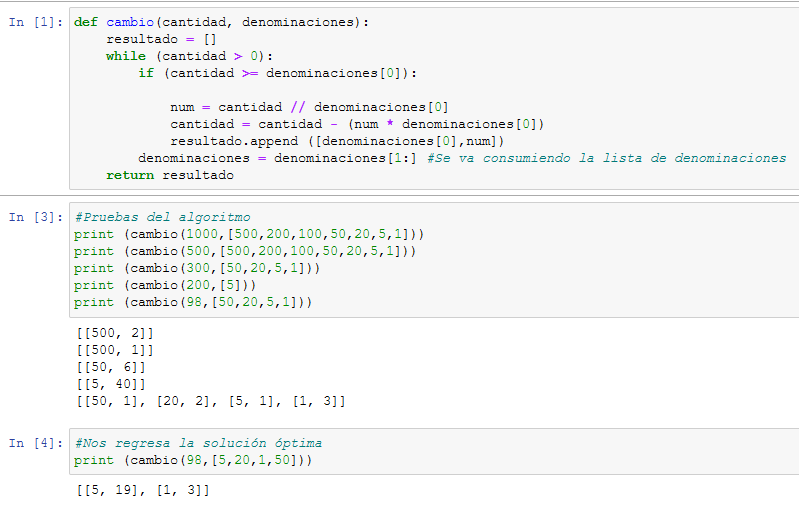
Implementación de un buscador de contraseñas de entre 3 y 4 caracteres.





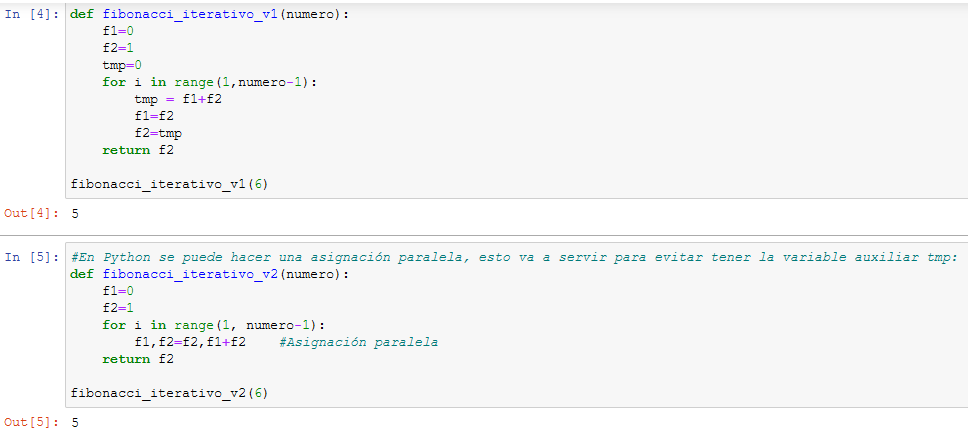
## Algoritmos ávidos (greedy):

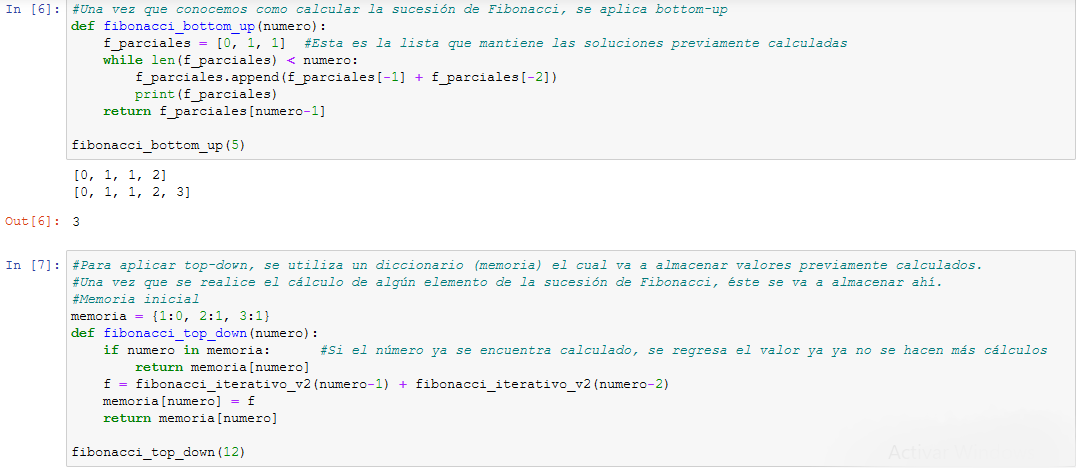
Implementación del problema de cambio de monedas. El problema consiste en regresar el cambio de monedas, de cierta denominación, usando el menor número de éstas. Este problema se resuelve escogiendo sucesivamente las monedas de mayor valor hasta que ya no se pueda seguir usándolas y cuando esto pasa, se utiliza la siguiente de mayor valor. La desventaja en esta solución es que, si no se da la denominación de monedas en orden de mayor a menor, se resuelve el problema, pero no de una manera óptima.



## Bottom-up (programación dinámica) y Top-down:

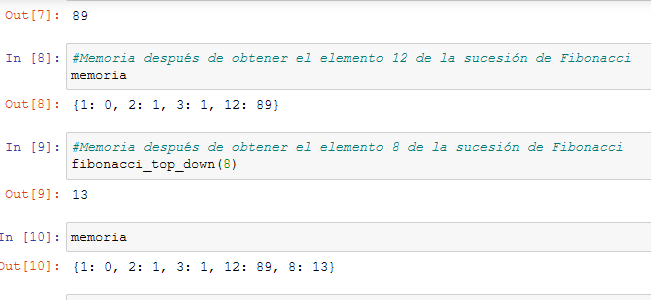
Implementación del cálculo el número n de la sucesión de Fibonacci. La sucesión de Fibonacci es una sucesión infinita de números enteros cuyos primeros dos elementos son 0 y 1, los siguientes números son calculados por la suma de los dos anteriores:

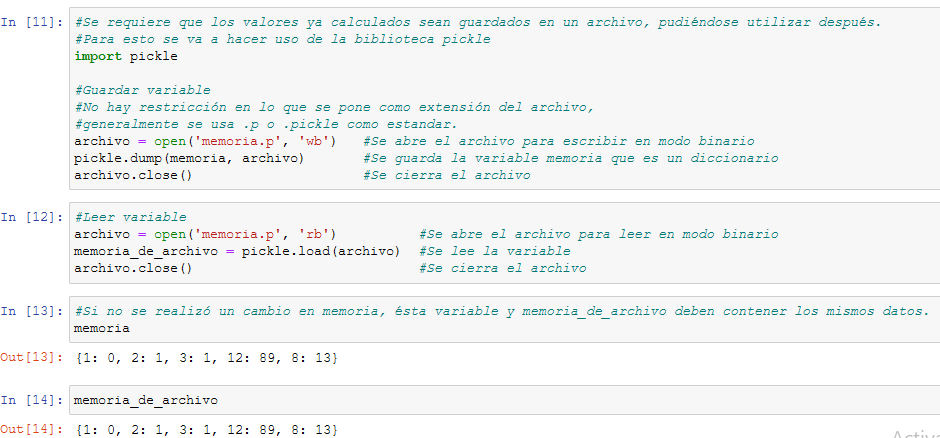




Como se observa en el resultado anterior, no se hace el cálculo de los primeros números, si no que se toman las soluciones ya existentes. La solución se encuentra calculando los resultados desde los primeros números (casos base), hasta llegar a n, de abajo hacia arriba.

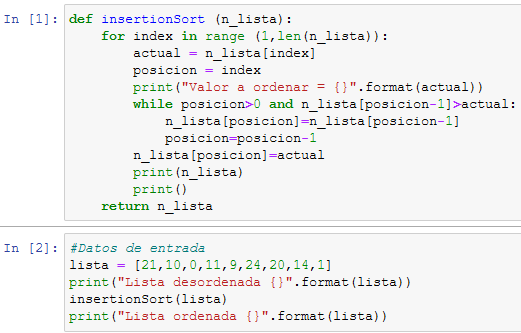
Para aplicar la estrategia top-down, se utiliza un diccionario (memoria) el cual va a almacenar valores previamente calculados. Una vez que se realice el cálculo de algún elemento de la sucesión de Fibonacci, éste se va a almacenar ahí.

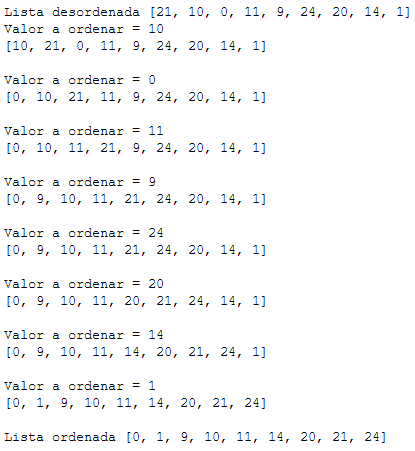




## Incremental:

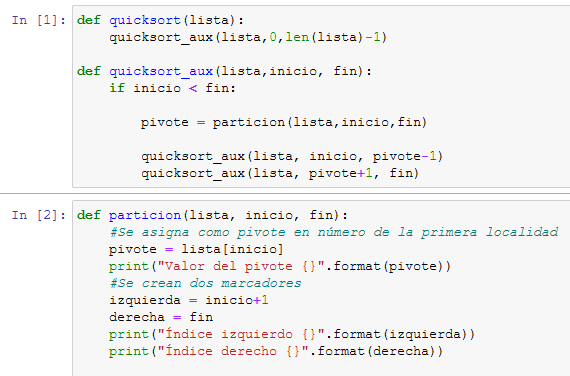
Se implementará “Insertion sort”, que ordena los elementos manteniendo una sublista de números ordenados empezando por las primeras localidades de la lista. Al principio se considera que el elemento en la primera posición de la lista está ordenado. Después cada uno de los elementos de la lista se compara con la sublista ordenada para encontrar la posición adecuada.

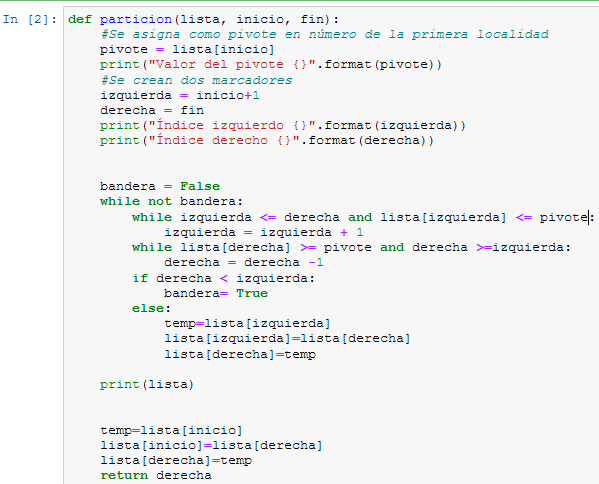




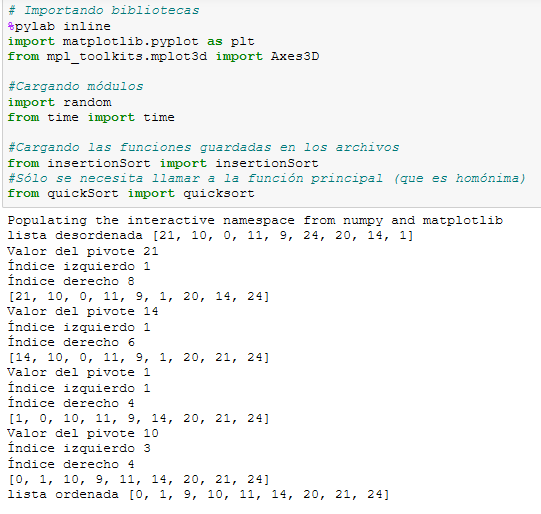
## Divide y vencerás:

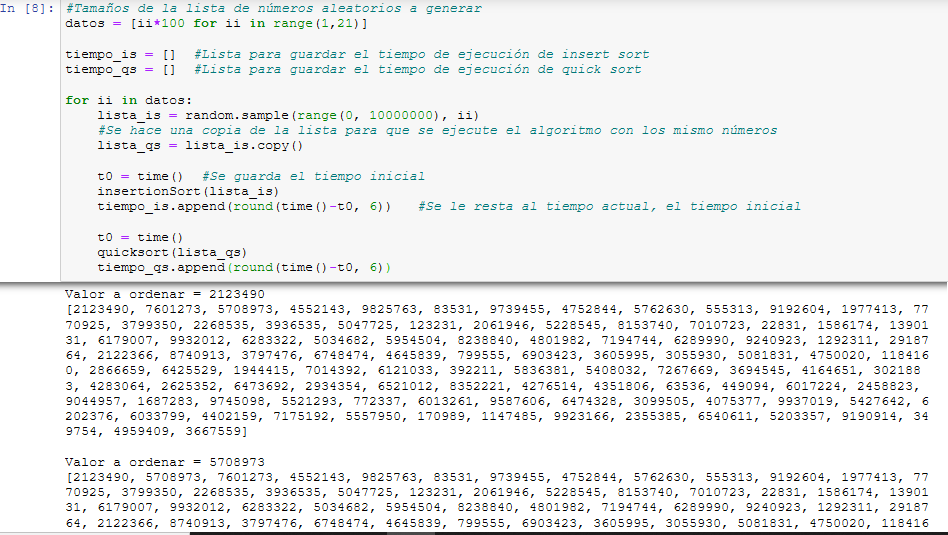
Se implementará “Quick sort”, que divide en dos el arreglo que va a ser ordenado y se llama recursivamente para ordenar las divisiones. La parte más importante en Quicksort es la partición de los datos. Lo primero que se necesita es escoger un valor de pivote el cual está encargado de ayudar con la partición de los datos. El objetivo de dividir los datos es mover los que se encuentran en una posición incorrecta con respecto al pivote.

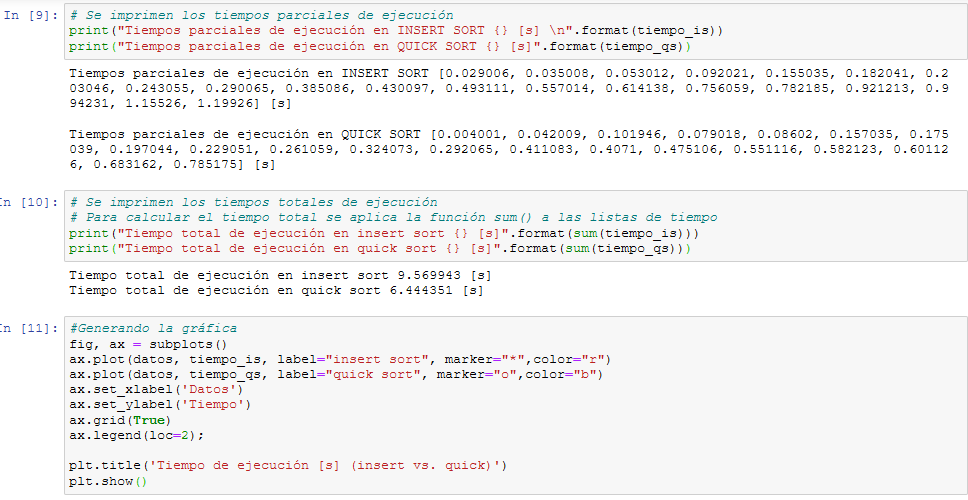


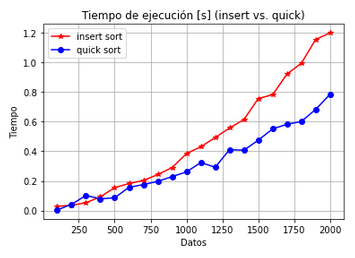


## Medición y gráficas de los tiempos de ejecución:

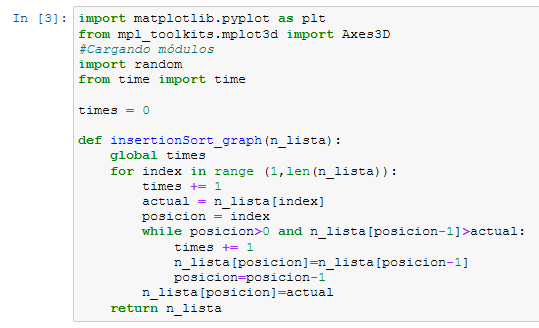


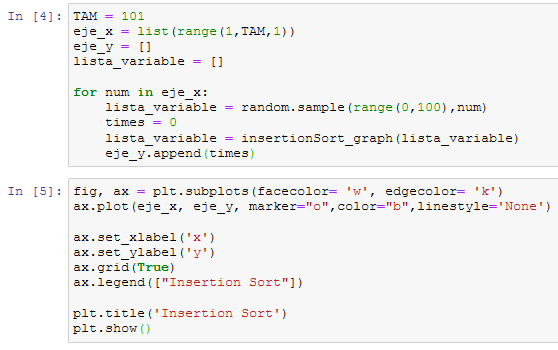


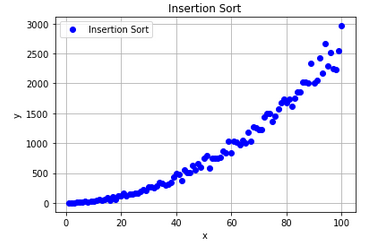




## Modelo RAM:







# Conclusión:

La importancia de la construcción de algoritmos es algo que ningún programador debe olvidar, ya que éstos brindan la base en la resolución de problemas. El uso del lenguaje Python se ve bastante favorecido en el momento de la implementación, ya que cuenta con el apoyo de muchas personas que han desarrollado bastantes librerías a lo largo de los años. Debido a la sencillez en cuanto la especificación de tipos, acciones y sintaxis, lo hace un lenguaje más fácil para trabajar y cometer menos errores sintácticos, facilitando así la fácil adaptación de un algoritmo a la codificación. Por otro lado, los códigos en la práctica siguen siendo algo confusos en algunas partes, sin embargo, nada que no se pueda resolver analizando a detalle lo necesario para hacer compilar el programa.

Referencias:

* Edgar Tovar (2012), “*Algoritmo y Diagrama de Flujo*”, (consultado el 24/04/2020). Recuperado de: <https://www.monografias.com/trabajos94/algoritmo-y-diagrama-flujo/algoritmo-y-diagrama-flujo.shtml#Comentarios>
* Design and analysis of algorithms; Prabhakar Gupta y Manish Varshney; PHI Learning, 2012, segunda edición.
* Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein; The MIT Press; 2009, tercera edicion.
* Problem Solving with Algorithms and Data Structures using Python; Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates; 2011, segunda edition.
* [https://docs.python.org/3/library/itertools.html#](https://docs.python.org/3/library/itertools.html)
* [https://docs.python.org/3/library/itertools.html#itertools.product](https://docs.python.org/3/library/itertools.html#itertools.producth)
* <https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files>
* <https://docs.python.org/3.5/library/pickle.html>