**Universidad Nacional Autónoma de México**

Facultad de Ingeniería

**Laboratorios de computación**

**salas A y B**

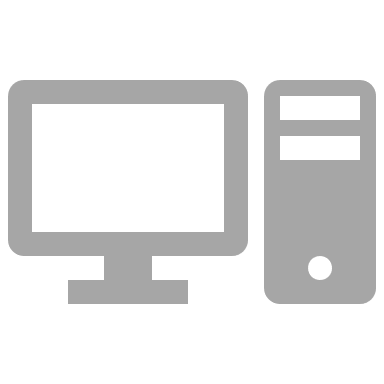
**PROFESOR:** M.I. Marco Antonio Martínez Quintana

**ASIGNATURA:** Estructura de Datos y Algoritmos I

**GRUPO:** 17

**NO DE PRÁCTICA:** 11

**NOMBRE:** Reyes Mendoza Miriam Guadalupe

**SEMESTRE:** 2020-2

**FECHA DE ENTREGA:** 21/04/2020

**OBSERVACIONES:**

**CALIFICACIÓN:**

**ESTRATEGIAS PARA LA**

**CONSTRUCCIÓN DE ALGORITMOS**

**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**

**FUERZA BRUTA**

En criptografía, se denomina *ataque de fuerza bruta* a la forma de encontrar una clave probando todas las combinaciones posibles de claves hasta hallar aquella que permite recuperar el texto plano. Es evidente que no se trata de una técnica muy refinada, pero desde luego sí efectiva, más aún cuando un ordenador puede probar un enorme número de claves en un tiempo muy reducido. Cuanto más pequeño sea el espacio de claves, es decir, el número de claves totales qué pueden usarse en un criptosistema, menor será el tiempo requerido para romper el sistema.

El ejemplo muestra la implementación de un buscador de contraseñas de entre 3 y 4 caracteres; se va a usar la biblioteca ***string***, de manera que se van a importar los caracteres y dígitos. También se usa la biblioteca ***itertools***, la cual tiene la función ***product()*** la cual se utiliza para *realizar las combinaciones* en cadenas de 3 y 4 caracteres.

Las diferentes combinaciones generadas por el algoritmo se van a guardar en un archivo. Para *guardar datos* en un archivo se utiliza la función ***open()***, que es para tener una *referencia del archivo que se quiere abrir*. Con la referencia creada se utiliza la función ***write()***, que *recibe la cadena* que se va a escribir en el archivo. Finalmente, una vez que se termina la escritura hacia el archivo, éste se *cierra* con la función ***close()***.

Una captura de pantalla de una red social

Descripción generada automáticamente

**ALGORITMOS ÁVIDOS (GREEDY)**

Los algoritmos ávidos son algoritmos en los que tomamos decisiones locales para llegar a una cantidad óptima global. En este sentido, son parecidos a los de programación dinámica. La diferencia es que, para resolver un problema con programación dinámica, tomamos en cuenta todas las soluciones para los subproblemas anteriores y con esto encontramos la solución óptima; mientras que para resolver un problema mediante un algoritmo ávido, sólo tomamos la solución óptima del subproblema anterior y con esto calculamos la óptima actual (por lo que no necesitamos guardar datos).

Esto es, empezamos por el caso más sencillo y en cada paso tomamos la decisión que resulte más eficiente en ese momento, con lo que esperamos llegar a una solución óptima global, sin reconsiderar opciones.

Los algoritmos ávidos encuentran las soluciones mucho más rápido que por programación dinámica, y muchas veces que por cualquier otro método. El problema es que, aunque el enfoque es generalmente fácil de implementar, en la mayoría de las ocasiones falla por no tomar en cuenta todas las combinaciones. Es por esto que no es conveniente utilizarlo a menos de que estemos convencidos de que funciona (preferentemente con una demostración).

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.

Captura de pantalla de un celular

Descripción generada automáticamente

**BOTTOM-UP (PROGRAMACIÓN DINÁMICA)**

La función *Bottom-Up* es función una manera de evitar la recursividad, ahorrando el costo de memoria en el que incurre la recursividad cuando se construye la pila. En pocas palabras, un algoritmo *Bottom-Up* "comienza desde el principio", mientras que un algoritmo recursivo a menudo "comienza desde el final y trabaja hacia atrás”.

Ir de abajo hacia arriba es una estrategia común para problemas de programación dinámica, que son problemas en los que la solución se compone de soluciones al mismo problema con entradas más pequeñas (como con la multiplicación de los números *1...n*, arriba). La otra estrategia común para los problemas de programación dinámica es la memorización.

Como ejemplo, se va calcula el número **n** de la sucesión de Fibonacci. Una vez que se conoce como calcular la sucesión de Fibonacci, se aplica la estrategia bottom-up. Partiendo del hecho de que ya tenemos las soluciones para:

*f(0) = 0* , *f(1) = 1* y *f(2)=1*

Estas soluciones previas son almacenadas en la tabla de soluciones f\_parciales.

*f\_parciales = [0, 1, 1]*

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

*[0, 1, 1] Datos iniciales*

*[0, 1, 1, 2] Primera iteración, se calcula n-1 = 1, y n - 2 = 1*

*[0, 1, 1, 2, 3] Segunda iteración, se calcula n-1 = 2, y n - 2 = 1*

Una captura de pantalla de una red social

Descripción generada automáticamente

Captura de pantalla de un celular

Descripción generada automáticamente

**TOP-DOWN**

También conocida como de arriba-abajo y consiste en establecer una serie de niveles de mayor a menor complejidad (arriba-abajo) que den solución al problema. Consiste en efectuar una relación entre las etapas de la estructuración de forma que una etapa jerárquica y su inmediato inferior se relacionen mediante entradas y salidas de información. Este diseño consiste en una serie de descomposiciones sucesivas del problema inicial, que recibe el refinamiento progresivo del repertorio de instrucciones que van a formar parte del programa.

La utilización de la técnica de diseño Top-Down tiene los siguientes objetivos básicos:

* Simplificación del problema y de los subprogramas de cada descomposición.

* Las diferentes partes del problema pueden ser programadas de modo independiente e incluso por diferentes personas.
* El programa final queda estructurado en forma de bloque o módulos lo que hace más sencilla su lectura y mantenimiento.

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.

La ventaja, es que una vez que ya se calcularon, se guardan en una memoria, que en este caso es un diccionario; en dado caso de que se necesite un valor que ya ha sido calculado, sólo regresa y ya no se realizan los cálculos.

Captura de pantalla de un celular

Descripción generada automáticamente

El problema con esta versión es que se siguen haciendo cálculos de más, ya que la función fibonacci\_iterativo\_v2() no tiene acceso a la variable memoria, lo que implica que tenemos que hacer modificaciones a la implementación.

Ahora, se requiere que *los valores ya calculados sean guardados en un archivo*, de tal manera que se puedan utilizar en otro instante de tiempo. Para esto se va a hacer uso de la biblioteca ***pickle***. Los archivos que se generan con pickle están en binario, por lo que no se puede leer a simple vista la información que contienen, como se haría desde un archivo de texto plano.

Captura de pantalla de un celular

Descripción generada automáticamente

**INCREMENTAL**

Python está diseñado para ser coherente y legible. Un error común de un programador principiante en lenguajes con ++ y -- operadores es mezclar las diferencias (tanto en precedencia como en valor devuelto) entre los operadores pre y post incremento/decremento. Los operadores simples de incremento y decremento no son tan necesarios como en otros idiomas.

**INSERTION SORT**

Insertion Sort es muy simple e intuitivo de implementar, que es una de las razones por las que generalmente se enseña en una etapa temprana de la programación. Es un algoritmo estable, in-place, que funciona muy bien para arreglos de discos casi ordenados o pequeños.

***In-Place:*** *Requiere un espacio adicional pequeño y constante (sin importar el tamaño de entrada de la colección), pero vuelve a escribir las ubicaciones de memoria originales de los elementos de la colección.*

Otra cosa para tener en cuenta, es ordenar de inserción no necesita conocer toda la matriz de antemano antes de ordenar. El algoritmo puede recibir un elemento a la vez. Lo cual es genial si queremos agregar más elementos para ordenar - el algoritmo sólo inserta ese elemento en su lugar adecuado sin "rehacer" todo el tipo.

La ordenación de inserción se utiliza con bastante frecuencia en la práctica, debido a lo eficiente que es para los conjuntos de datos pequeños (10 elementos).

Captura de pantalla de un celular

Descripción generada automáticamenteCaptura de pantalla de un celular

Descripción generada automáticamente

**DIVIDE Y VENCERÁS**

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.

**QUICK SORT**

Quicksort consiste en dividir la lista original en dos listas más pequeñas. Para ello, se elige un elemento cualquiera (aunque en general se suele utilizar el que se encuentra en medio de la lista) que nos servirá como pivote. Luego se recorre toda la lista, con el objeto de colocar los elementos más pequeños que el pivote a la izquierda de este, y los mayores a la derecha.

Las implementaciones más eficientes realizan esta tarea a la vez, recorriendo simultáneamente la lista en ambas direcciones e intercambiando entre sí cada par de elementos “*descolocados*” que se encuentran a su paso. Culminada esta etapa tenemos un grupo de elementos menores que el pivote, el pivote, y otro grupo compuesto por números mayores a él. En este punto es donde juega un papel importante la recursividad: *si cada uno de esos grupos vuelve a dividirse en dos y se aplica lo explicado anteriormente de forma recursiva, tenemos resuelto el problema*. Lo mejor de todo es que el tamaño de las sublistas y por ende el tiempo que se necesita para procesarlas- es cada vez menor.

En cada nivel el tamaño de las sublistas la mitad del anterior, lo que permite ordenar una lista de 1024 elementos en 10 “*pasadas*”, y una de más de un millón en solo 20. Y lo mejor de todo es que no requiere de una “*segunda caja vacía*” sobre la que ordenar los elementos, con lo que el consumo de recursos (memoria) es menor.

Captura de pantalla de un celular

Descripción generada automáticamente Captura de pantalla de un celular

Descripción generada automáticamente

**MEDICIÓN Y GRÁFICAS DE LOS TIEMPOS DE EJECUCIÓN**

**Tip:** Las funciones en Python pueden ser guardadas en archivos individuales (insertionSort.py) o varias en un sólo archivo (quickSort.py). En el siguiente ejemplo, se agrego \_time al nombre de la función en los archivos.

**Tip:** En dado caso de que se quiera llamar más funciones que estén en un mismo archivo se pueden escribir los nombres de las funciones separados por nombres: \*from file\_name import función1, función2, función3\*

Captura de pantalla de un celular

Descripción generada automáticamente

Captura de pantalla de un celular

Descripción generada automáticamenteUna captura de pantalla de una red social

Descripción generada automáticamente

**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.

Captura de pantalla de un celular

Descripción generada automáticamente

Una captura de pantalla de una red social

Descripción generada automáticamente

**CONCLUSIONES**

Con esta práctica pude observar que existen diferentes funciones que nos van a permitir que los programas o los códigos sean muchísimo más eficientes e igualmente muchas estrategias que nos permiten realizarlo de una manera más cómoda y práctica. Y esto pudimos utilizar diferentes diseños o estrategias de algoritmos e ir viendo los pasos que implicar cada uno de ellos si es que los vamos a utilizar.

Yo creo que además de los ejercicios que vienen las prácticas para aprender necesitas ponerlos en práctica y estar buscando nuevas formas de hacer un mejor programa con herramientas que sean útiles prácticas y eficientes para el problema que queramos solucionar. Esto es lo que estamos estado viendo durante todo el curso formas de optimizar tiempo de ejecución, código y memoria, esto es realmente útil durante la práctica. A fin de cuentas, creo que como programador siempre debes estar buscando estrategias que para ti sean útiles y te permitan un buen manejo de toda la información, pero también que sean buenas y vayan mejorando los códigos cada vez más, con su dominio que se va aprendiendo con el paso del tiempo y sobre todo con la práctica.

**BIBLIOGRAFÍA**

* Arboledas, D. (2017). *Criptografía sin secretos con Python* (1ª edición). Madrid, España: RA-MA.
* Cuevas, A. (2019). *Programar con Python 3* (1ª edición). Madrid, España: RA-MA.
* Cake Labs, Inc. (s.f.). *Bottom-Up Algorithms and Dynamic Programming | Interview Cake*. Recuperado 21 de abril de 2020, de https://www.interviewcake.com/concept/java/bottom-up
* Hernández, P. P. G. (s.f.). *Algoritmos ávidos (glotones, voraces)*. Recuperado 21 de abril de 2020, de http://www.pier.guillen.com.mx/algorithms/11-otros/11.6-algoritmos\_avidos.htm
* Palazzesi, A. (2010, septiembre 7). *Quicksort: Algoritmo de ordenamiento rápido*. Recuperado 21 de abril de 2020, de https://www.neoteo.com/quicksort-algoritmo-de-ordenamiento-rapido/