	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: TISTA GARCÍA EDGAR

Asignatura: ESTRUCTURA DE DATOS Y ALGORITMOS I

Grupo: 1

No de Práctica(s): Guía práctica de estudio 11: Estrategias para la construcción de algoritmos.

Integrante(s): MURRIETA VILLEGAS ALFONSO / VALDESPINO MENDIETA JOAQUÍN

Semestre: 2018 - 2

Fecha de entrega: 7 DE MAYO DE 2018

Observaciones:

CALIFICACIÓN: _____

ESTRATEGIAS PARA LA CONSTRUCCIÓN DE ALGORITMOS.

INTRODUCCIÓN

En el mundo de la programación existen ciertas características que hacen a éste metódico, para poder resolver cualquier problema siempre se debe plantear un algoritmo el cual se encargará de dar ya sea a primera instancia una solución cualquiera, o la mejor solución, posteriormente este se lleva a través de un lenguaje de programación para poder representarlo en lo que conocemos como programa.

Pero antes, se le conoce como **algoritmo** a un conjunto finito de instrucciones libre de ambigüedades que sirven para realizar una tarea o acción específica. Para hacer un algoritmo debe tener algunas características particulares como son:

1) Debe ser exacta la descripción de las actividades a realizar.
2) Debe excluir cualquier ambigüedad
3) Debe ser invariante en el tiempo como en lugar.

Algunas herramientas para elaborarlos son los diagramas de flujo ó los pseudocódigos, que pretenden facilitar la codificación posterior (en algún lenguaje de programación), mientras que el diagrama hace uso de elementos gráficos y notación estandarizada, el pseudocódigo hace una aproximación a la codificación en un lenguaje de programación.

Cuando se diseñan o realizan programas computacionales, se requiere un análisis o un proceso para elaborarlo, por ello debe de haber métodos para construir algoritmos de una manera más sencilla o eficiente en ciertos casos, a estos son denominados como estrategias para la construcción o diseño de algoritmos, en esta práctica se revisaran las estrategias como: fuerza bruta, ávidos, top-down y bottom-up.

Como se mencionó anteriormente, el fin de un algoritmo es resolver un problema, sin embargo, muchas veces pueden existir distintas formas de resolver un problema, es ahí donde se menciona el concepto de "Problemas de optimización", muchas el punto principal no es solamente maximizar o minimizar una variable, realmente dentro del mundo de la programación no sólo se buscar dar una solución a algo sino dar la "mejor" solución, para ello se toman factores como son un costo total menor o aumentar al máximo el beneficio total de algún sistema.

OBJETIVOS DE LA PRÁCTICA

- El objetivo de esta guía es aplicar los algoritmos básicos para la solución de problemas

DESARROLLO

Actividad 1. Análisis del manual de prácticas

Para poder hacer un mejor análisis de toda la parte teórica del manual de prácticas, se decidió separar en distintos sub-apartados para un mejor análisis.

1.1 Fuerza Bruta

Cuando hablamos de fuerza bruta nos referimos a verificar todas las posibles soluciones a un problema o al menos hasta encontrar aquellas que lo resuelvan (Podría decirse en cierta forma a prueba y error).

Su forma de aproximación a la solución del problema es de la más simple posible. Entre sus ventajas y desventajas se encuentran:

Ventajas	Desventajas
Simplicidad de algoritmos	Carece de eficiencia en muchos casos
Es aplicable a una gran variedad de problemas	Funciona para instancias cortas

Resumiendo, fuerza bruta es hacer una búsqueda exhaustivamente de todas las posibilidades que nos lleven a la solución.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Fuerza bruta tiene el objetivo de a través de probar todas las posibilidades nos lleve a una solución del problema, en este problema se da una contraseña la cual se debe averiguar, para ello en el programa en Python a través de la función "***buscador()***" se dará la posible solución.

Dentro de esta función lo primero que podemos ver es una simple condición if encargada de condicionar y limitar los ciclos for de búsqueda, esto debido a que si tuviera más caracteres la contraseña no podría llegarse a una solución. Posteriormente y dentro de la condición if, notamos dos ciclos for anidados encargados de probar cada una de las contraseñas (cadenas de caracteres) que se le ha pasado al programa a través de un archivo de texto plano (txt).

```
if 3<= len(con) <= 4:
    for i in range(3,5):
        for comb in product(caracteres, repeat = i):
            #Se utiliza join() para concatenar los caracteres regresado por la función product().
            #Como join necesita una cadena inicial para hacer la concatenación, se pone una cadena vacía
            #al principio
            prueba = "".join(comb)
            #Escribiendo al archivo cada combinación generada
            archivo.write( prueba + "\n"  )
```

Imagen 1: Captura de pantalla de las líneas de código encargadas de la búsqueda de la contraseña

Como es obvio, en caso de que no se pueda encontrar una contraseña se tienen las líneas del bloque else que indican precisamente la inexistencia de la contraseña.

```
In [5]: runfile('C:/Users/Joaquin/Desktop/uni/estructuras/p11/ejemplo1.py', wdir='C:/Users/Joaquin/
Desktop/uni/estructuras/p11')
Tu contraseña es H0l4
Tiempos de ejecución 26.246041
```

Imagen 2: Salida del programa

NOTA: Ya como parte final y en caso que se encuentre la contraseña, se tiene un apartado para la impresión del tiempo empleado de la búsqueda de ésta.

```
print("Tiempos de ejecución {}".format(round(time()-t0, 6)))
```

Imagen 3: Salida del programa (Línea encargada del apartado del tiempo)

Lo que realmente podemos notar con este procedimiento-algoritmo es la forma forzada en que se trata de obtener algo en concreto., tal vez en este caso no se utilizó mucho tiempo para encontrar la contraseña, sin embargo, el agregar más caracteres o incluso comparar entre un grupo más grande de contraseñas obvio aumentaría la cantidad de tiempo de búsqueda.

ESTRATEGIA USADA Y PROBABLE MÉTODO(S) ALTERNO

Este algoritmo pertenece o es del tipo Fuerza Bruta debido a que realmente se está comparando todos los casos (Posibles contraseñas) con respecto a la contraseña correcta, recordemos que en fuerza bruta se parte de un punto en el que se deben probar todas las posibles soluciones.

Realmente el conocer una contraseña es algo realmente complicado de saber y más si esta tiene muchos caracteres, probablemente no se pueda emplear otro algoritmo que sea mejor que Fuerza Bruta para encontrar la contraseña, tal vez esto se pueda resolver con conocer más acerca de encriptación.

1.2 Ávidos - Greedy

Esta estrategia se diferencia de fuerza bruta porque va tomando una serie de decisiones en un orden específico, una vez que se ha ejecutado esa decisión, ya no se vuelve a considerar. Algo destacable es que pese siempre se considera la mejor decisión dentro del momento en que se encuentra el algoritmo lamentablemente esto no garantiza que la solución final-general se la óptima.

Ventajas	Desventajas
Plantea pocas dificultades al diseñar y comprobar su funcionamiento.	No se pueden revertir las decisiones tomadas
Siempre tendremos una solución	No siempre da la solución óptima

Los pasos que se realizan para llevar a cabo un algoritmo mediante Greedy son:

1) Enunciar el problema de tal forma que se creen sub-problemas.
2) Mostrar que siempre hay una solución al problema mediante Greedy
3) Demostrar que la solución al sub-problema puede ayudar a la solución general del problema
4) No necesariamente tenemos la solución óptima.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

A continuación, se muestra un problema que se puede resolver a través de Greedy, es el caso 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.

Para llevar esto a código lo primero que se utilizó en el programa fue un ciclo while que a través de la condición de que la cantidad sea mayor a 0 se pueda entrar al bloque de código encargado de comparar las monedas (Es el bloque dentro de la condición if).
Una vez que la cantidad era todavía posible de partir utilizando alguna denominación de las monedas se entra al apartado encargado de determinar la cantidad (Obvio en números enteros) de monedas que asumieran la mayor cantidad posible.

```
def cambio(cantidad, denominaciones):
    resultado = []
    while (cantidad > 0):
        if (cantidad >= denominaciones[0]):

            num = cantidad // denominaciones[0]
            cantidad = cantidad - (num * denominaciones[0])
            resultado.append([denominaciones[0], num])
        denominaciones = denominaciones[1:]  #Se va consumiendo la lista de denominaciones
    return resultado
```

Imagen 4: Captura de pantalla de las líneas de código encargadas de la distribución de monedas con respecto al dinero.

Sin embargo, la desventaja de 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 forma óptima, además de que no precisamente podemos llegar a la solución óptima.

```
In [11]: runfile('C:/Users/Joaquin/Desktop/uni/estructuras/p11/ejemplo2.py', wdir='C:/Users/Joaquin/
Desktop/uni/estructuras/p11')
[[500, 2]]
[[500, 1]]
[[50, 6]]
[[5, 40]]
[[50, 1], [20, 2], [5, 1], [1, 3]]
ejemplo de solucion no optima
[[5, 19], [1, 3]]
```

Imagen 5: Salida del programa

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.

EXTRA: Dentro de Python las divisiones se dan a través del siguiente operador " / ", sin embargo, cuando empleamos dos veces este " // "podemos notar en cierta forma una división truca pues solamente obtenemos el resultado entero de la operación.

ESTRATEGIA USADA Y PROBABLE MÉTODO(S) ALTERNO

Este algoritmo pertenece al tipo Greedy debido a que conforme se va dando otra cantidad monetaria ésta se va asociando a una cantidad de monedas con respecto al total monetario, además como bien sabemos Greedy se caracteriza por tomar la mejor decisión en el momento en que se encuentra, esto lo podemos ver en cómo va escogiendo las monedas con respecto a la cantidad de dinero restante.

Como versión alternativa podría emplearse ***Fuerza Bruta*** la cual a través de la comparación de todas las posibles respuestas (Todas las combinaciones de cambio) podríamos llegar a la mejor respuesta (La menor cantidad de monedas), sin embargo, esto tomaría muchísimo tiempo con respecto a Greedy debido a que estaríamos buscando y comparando cada una de las combinaciones.

1.3 Bottom-up (programación dinámica)

Tanto ***Top Down*** como ***Bottom Up*** son técnicas de diseño de algoritmos que especifican el tipo de enfoque con el que se resuelve o resolverá un problema.

En el caso de ***Bottom Up*** el objetivo es resolver un problema a partir de sub-problemas, una vez resueltos estos sub-problemas se trata de recolectar o conjuntar las soluciones para poder llegar a una solución final.

NOTA: Es importante denotar que se utiliza principalmente cuando ya se elaboró el diseño de la solución, sin embargo, al partir de sub-problemas puede llegar a ser complicado debido a que no se tiene un panorama general del problema.

EXTRA: 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.
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Lo primero que podemos ver es que el programa implementa un algoritmo iterativo a través de ciclos for, cabe destacar que realmente partimos de la idea o del onocimiento de algunos números de la sucesión de Fibonacci (Para ser exacto los primeros 3), posteriormente estos datos son almacenados en la función "soluciones_parciales", como se dijo anteriormente este método trata de no repetir u obtener datos dos veces por eso mismo al ser guardados estos datos, inmediatamente se impieza a iterar en los siguientes elementos de la sucesión.

```
def fibonacci_bottom_up(numero):
    f_parciales = [0, 1, 1]  #Esta es la lista que mantiene las soluciones previamente calculadas
    while len(f_parciales) < numero:
        f_parciales.append(f_parciales[-1] + f_parciales[-2])
        print(f_parciales)
    return f_parciales[numero-1]
```

```
fibonacci_bottom_up(5)
```

```
[0, 1, 1, 2]
[0, 1, 1, 2, 3]

3
```

Imagen 6: Captura de pantalla de las líneas de código encargadas de la obtención de los elementos de la sucesión de Fibonacci, en la parte de abajo el número siguiente a los números dados al inicio del programa (Números base)

Ya como resultado final, lo único que arroja el programa son los demás elementos de la sucesión.

```
In [6]: runfile('C:/Users/Joaquin/Desktop/uni/estructuras/p11/ejemplo3.py', wdir='C:/Users/Joaquin/
Desktop/uni/estructuras/p11')
[0, 1, 1]
0
[0, 1, 1]
1
[0, 1, 1]
1
[0, 1, 1, 2]
2
[0, 1, 1, 2, 3]
3
[0, 1, 1, 2, 3, 5]
5
```

Imagen 7: Salida del programa, se puede apreciar cómo se van agregando cada uno de los números de Fibonacci dentro de la lista de elementos.

NOTA: Se modificó el programa para que no imprimiera cada vez que se actualizara la lista (se omitió el print que estaba dentro del ciclo while).

ESTRATEGIA USADA Y PROBABLE MÉTODO(S) ALTERNO

Este algoritmo pertenece a Bottum Up debido a que se parte de algunos elementos de la sucesión de Fibonacci (Partimos de casos particulares y damos un resultado general).

Podemos llevar a cabo este algoritmo a través de Top Down donde a través de una generalización del problema (O sea encontrar una versión iterativa general, la cual se encargaría de encontrar elemento por elemento) podemos ir partiéndolo en sub-problemas (Puede verse en el siguiente apartado)

1.4 Top-down

Consiste en establecer una serie de niveles de mayor a menor complejidad, además de relacionarlos. La utilización de esta técnica nos ofrece:

1) Simplificar el problema en sub-problemas
2) Los sub-problemas pueden ser independientes entre si
3) EL programa o resultado final está estructurado en módulos los cuales resultan fáciles de interpretar y dar mantenimiento.

La principal diferencia de Top Down con respecto a bottom-up es que aquí se empiezan a hacer los cálculos de n hacia abajo (Como se dijo anteriormente). Además, se aplica una técnica llamada ***memoización*** la cual consiste en guardar los resultados previamente calculados, de tal manera que no se tengan que repetir operaciones y de esta forma optimizar los calculos y memoria.

Para aplicar la estrategia top-down, se utiliza un diccionario (memoria) el cual va a almacenar valores previamente calculados.

ANÁLISIS DEL PROBLEMA-PROGRAMA DE LA PRÁCTICA

Para aplicar la estrategia top-down, se utiliza un diccionario (Como se vio en la práctica anterior, el uso de listas o diccionarios es una de las ventajas e implementaciones que ofrece Python con respecto a otros lenguajes) el cual va a almacenar valores previamente calculados. Una vez realizados los cálculos de algunos elementos de la sucesión de Fibonacci, estos se irán almacenando en un diccionario.

Lo primero que podemos observar es que al igual en que en Bottom Up se utiliza la iteración para ir calculando elemento por elemento, la diferencia es que al no partir de ningún elemento lamentablemente el algoritmo se vuelve ineficiente debido a que se repiten operaciones para calcular el elemento siguiente de la sucesión de Fibonacci, pues pese se conocen los números anteriores no se utilizan para el cálculo del siguiente elemento.

NOTA: Recordemos que en la sucesión de Fibonacci el elemento siguiente es la suma de los 2 elementos anteriores.

```
def fibonacci_top_down(numero):
    if numero in memoria:      #Si el número ya se encuentra calculado, se regresa el valor ya ya no se hacen más cálcul
        return memoria[numero]
    f = fibonacci_iterativo_v2(numero-1) + fibonacci_iterativo_v2(numero-2)
    memoria[numero] = f
    return memoria[numero]
```

Imagen 8: Captura de pantalla de las líneas de código encargadas de la obtención de los elementos de la sucesión de Fibonacci, dentro del if vemos el retorno de algún elemento de la sucesión, mientras que la siguiente parte se encarga de determinar el número siguiente de la sucesión a través de la suma de los 2 anteriores, posteriormente el número calculado se asigna a "memoria[numero]" y por último es retornada y guardada en un diccionario.

Además, la ventaja de este algoritmo, es que una vez que ya se calcularon elementos, se guardan en una memoria, (En el diccionario) en dado caso de que se necesite un valor que ya ha sido calculado, sólo regresa y ya no se realizan los cálculos.

EXTRA: Para crear archivos desde un programa de Python se puede utilizar la biblioteca "pickle", sin embargo, estos archivos están en binario lo cual limita su legibilidad.

ESTRATEGIA USADA Y PROBABLE MÉTODO(S) ALTERNO

El algoritmo del programa es del tipo Top Down debido a que partimos del caso general de la sucesión de Fibonacci para poder encontrar elemento a elemento además de que la ventaja que ofrece es que no se repiten casos u operaciones.

Como se vio en el apartado anterior, este algoritmo también se puede llevar realizar a través de Buttom Up, debido a que podemos partir de algunos elementos particulares para posteriormente obtener elemento a elemento, realmente estos 2 algoritmos pueden llegar a ser relativamente similares debido a que ambos son iterativos.

Podríamos en dado caso, a través de fuerza bruta obtener elemento a elemento partiendo de la regla de correspondencia de la sucesión de Fibonacci donde sabemos que el elemento es la suma de los 2 elementos anteriores, sin embargo, y como opinión personal, este sería probablemente el peor algoritmo para encontrar los elementos de la sucesión debido a que tendríamos que ir comparando si los elementos cumplen con la regla de correspondencia.

Actividad 2. Ejercicios de laboratorio

En el ejercicio del laboratorio, se tenía como problema seleccionar el número máximo de actividades que pudieran ser realizadas por una persona, a partir de la idea de que solamente pueda trabajar una sola actividad a la vez.

ANÁLISIS DEL PROBLEMA-PROGRAMA

El programa realiza una selección de actividades, considerando que se debe realizar la mayor cantidad de actividades en un tiempo determinado, como se dijo anteriormente solamente se debe realizar una sola actividad a la vez.

Para ello lo que se utiliza es un ciclo for donde iremos iterando una condición if encargada de ir comparando si es posible o no la asignación de una actividad.

ESTRATEGIA USADA

En el algoritmo se empleó Greedy debido a varias características del algoritmo:

1) Se va tomando una serie de decisiones en un orden específico, esto lo podemos ver dentro del ciclo for y la condición if donde comparamos las listas o arreglos "s" y "f"

2) Una vez que se ha ejecutado esa decisión, ya no se vuelven a considerar otros horarios y esto lo notamos en cómo va descartando elementos (Dicho de otra forma, no podemos regresar a la decisión anterior).

3) Como es notable, se inicia con una actividad, esta decisión no puede regresar porque alteraría todo el código, debido a que todo se basa en esta primera decisión.

4) Por último, la solución no es la óptima debido a que hay otras alternativas mejores, y aquí nuevamente afirmamos que es Greedy debido a que pese se tomaron las mejores decisiones de selección en el momento dado, esto no garantizó el que el resultado fuera el mejor.

NOTA: El mejor resultado es aquel que considera el total de horas empleadas, es decir, el que se realiza en el menor tiempo.

A continuación, se muestra la salida del programa donde se muestran las actividades dadas a través del uso del algoritmo de tipo Greedy.

```
In [1]: runfile('C:/Users/pon_c/Downloads/Documents/
Programación/Python/EDA-1/3_practice/ejercicio.py',
wdir='C:/Users/pon_c/Downloads/Documents/Programación/
Python/EDA-1/3_practice')
range(0, 9)
actividades seleccionadas son:
A1
A5
A7
A8
```

Imagen 9: Captura de pantalla de la salida de la actividad 2

CONCLUSIONES

ALFONSO MURRIETA VILLEGAS

Durante esta práctica se conocieron algunas estrategias importantes de cómo crear o diseñar algoritmos, como se dijo a lo largo de esta práctica, pese muchas veces el propósito general es resolver un problema, constantemente se trata de llegar a una solución óptima o al menos no la menos ineficiente.

A través de la interacción de estrategias como son Fuerza Bruta, Greedy, Top Down y Bottom Up se nos mostró ejemplos donde pudimos apreciar tanto las ventajas como desventajas que cada una de éstas estrategias.

Por otro lado, en el último ejercicio se puede apreciar el uso de Greedy además de cómo pese parece ser un código simple con un algoritmo realmente fácil de comprender e interpretar, lamentablemente el resultado no es el óptimo.

La implementación de estrategias para construir algoritmos tiene el propósito de plantearnos un método especifico que sobretodo nos facilite y no haga considerar las ventajas y desventajas al momento de crear un algoritmo. Realmente para esto se requiere tanto abstracción como una gran capacidad de lógica sobretodo porque el plantear ya sea de forma general o particular un problema necesita de saber hacerlo sin ambigüedades y de considerar que debe ser lo más legible posible.

JOAQUÍN VALDESPINO MENDIETA

En conclusión se han cumplido los objetivos debido a que se han realizado las actividades, sin embargo, cabe recalcar que las estrategias vistas en esta práctica requieren una lógica y un análisis de los problemas más profundo, ya que se debe de pensar en la forma y en la implementación de ese código en dicha estrategia, aunque a veces estas no otorgan la solución óptima, pueden reducir la complejidad y el tiempo de realización del código, pero como se dice se necesita un análisis para implementarlo de manera correcta y esa es la complejidad de este tema, por otra parte cada estrategia tiene sus ventajas y desventajas, su complejidad en ciertos aspectos.

REFERENCIAS

Solano Jorge A. (2017). *Manual de Prácticas del laboratorio de Estructuras de Datos y algoritmos 1*. UNAM, Facultad de Ingeniería.

Bradley N. Miller y David L. Ranum, Franklin, Beedle & Associates. (2011). Problem Solving with Algorithms and Data Structures using Python. Segunda Edición.

Mark J. Guzdial, Barbara Ericson. (2015). *Introducción a la computación y programación con Python*. 3ra Edición. Madrid España.

Marta, (2013)., Dieño de programas. Pseudocódigo y diagramas , recuperado de: http://informatica.iesvalledeljerteplasencia.es/wordpress/diseno-de-programas-pseudocodigo-y-diagramas/