Carátula para entrega de prácticas	
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor:	Edgar Tista García
Asignatura:	Estructuras de Datos y Algoritmos I
Grupo:	1
No. de Práctica(s):	11
Integrante(s):	Ugalde Velasco Armando
No. de Equipo de cómputo empleado:	
No. de Lista:	38
Semestre:	2020-2
Fecha de entrega:	13 de mayo de 2020
Observaciones:	

CALIFICACIÓN: _____

OBJETIVO

Implementar, al menos, dos enfoques de diseño (estrategias) de algoritmos y analizar las implicaciones de cada uno de ellos. Aplicar los algoritmos básicos para la solución de problemas.

ACTIVIDAD 1: EJERCICIOS DE LA GUÍA

I) DESCRIBE DETALLADAMENTE LA EJECUCIÓN DE CADA UNO DE LOS PROGRAMAS EN LA CATEGORÍA DONDE SE ENCUENTRAN, PARA CADA UNO DE LOS PROBLEMAS, INDICA A GRANDES RASGOS SI PUDIERA PERTENECER A CUALQUIER OTRA CATEGORÍA VISTA EN CLASE Y POR QUÉ.

Buscador (fuerza bruta)

La función **buscador** tiene un parámetro: **con**, el cual se trata de la contraseña a buscar. Dentro de ella se ejecuta un ciclo **for**, cuyo contador **i** toma los valores 3 y 4 para así generar todas las combinaciones posibles de caracteres de estas longitudes. Dentro de éste, se encuentra un ciclo anidado cuya función es comparar la contraseña brindada con cada combinación de caracteres de longitud **i (3 o 4)**. En cada iteración, además, se almacena cada combinación en un archivo de texto. En caso de encontrar una coincidencia, se imprime la contraseña correspondiente y se cierra el archivo.

Como podemos percatarnos, la estrategia que utiliza este algoritmo es **fuerza bruta**, ya que se intenta encontrar una solución con todas las combinaciones de caracteres posibles.

Cambio (greedy)

Este algoritmo obtiene la combinación de monedas mínima de “cambio” para una cantidad, dadas las denominaciones de las monedas a utilizar. Para que el algoritmo proporcione una solución óptima y correcta en todos los casos, las denominaciones se deben proporcionar en orden descendente (de mayor a menor), de lo contrario, el algoritmo puede no proporcionar una solución óptima, o bien, no proporcionar una solución.

En la función, se crea una lista vacía para almacenar las futuras combinaciones de monedas. Después, se ejecuta un ciclo **while**, el cual finalizará su ejecución cuando la cantidad dada no sea mayor a cero, es decir, mientras no se haya brindado una combinación de monedas cuyo valor sea igual al deseado. Dentro del ciclo se encuentra una condición:

si la primera moneda (denominación) se encuentra dentro del rango de la cantidad objetivo, entonces es posible utilizarla. En este caso, se procede a determinar la cantidad de monedas de la denominación correspondiente que es posible añadir a las combinaciones. Después, se actualiza la nueva cantidad objetivo, restando las monedas utilizadas, y finalmente se añade la denominación y el número de monedas utilizadas al arreglo **resultado.**

Finalmente, cuando se haya cumplido con la cantidad objetivo, se retorna el arreglo resultado, que, como ya se mencionó, contiene las combinaciones de monedas utilizadas.

Podría considerarse que la estrategia que utiliza este algoritmo es similar a la fuerza bruta, ya que se intenta encontrar una solución mediante la obtención de diferentes combinaciones. Sin embargo, sabemos que en realidad es una estrategia **greedy**, ya que en cada paso se trata de obtener la mayor "ganancia" posible.

Fibonacci iterativo (Bottom-up)

Este programa calcula el término **n** de la serie de Fibonacci. A diferencia de la común definición recursiva de ésta, en este programa se implementa de forma iterativa. Se inicia con los casos base, es decir, 0 y 1, los cuales se almacenan en las variables correspondientes (f1 y f2). Posteriormente, se ejecuta un ciclo **for** el número de veces restantes para obtener el término deseado, es decir, se ejecuta **n-2** veces. Dentro de éste, se realizan las asignaciones correspondientes para calcular el siguiente término. Esta estrategia se considera **bottom-up** debido a que el término deseado se calcula iniciando desde los casos base.

Después, se muestra otra implementación del mismo algoritmo. La diferencia es que en éste se almacenan sucesivamente los resultados obtenidos en cada iteración en un arreglo. Esta estrategia también es **bottom-up**, ya que, al igual que en el programa anterior, se parte de los casos base. Además, también se utiliza **programación dinámica**, ya que se almacenan los resultados para no tener que volver a realizar los cálculos correspondientes.

Fibonacci recursivo (Top-down)

Al igual que en el programa anterior, en éste también se calcula el término **n** de la serie de Fibonacci. Se implementa la forma común recursiva de éste, sin embargo, se utiliza una técnica llamada **memorización,** la cual consiste en almacenar las soluciones ya obtenidas de los términos, para así evitar realizar llamadas recursivas innecesarias y reali-

zar cálculos que ya se han hecho. En este caso, los términos se almacenan en un diccionario y en cada llamada recursiva se comprueba si el término a obtener ya se encuentra en éste, reduciendo el número de llamadas recursivas y mejorando la complejidad del algoritmo en gran medida.

La estrategia utilizada es **top-down** debido a que se realizan llamadas recursivas iniciando desde los términos mayores, es decir, el algoritmo inicia desde la parte superior del árbol de llamadas recursivas.

Insertion sort (Incremental)

En este programa se implementa el algoritmo de ordenamiento **insertion sort**, el cual consiste en iterar en cada elemento del arreglo, cambiándolo de posición si el elemento anterior a éste es mayor. El elemento se "recorre a la izquierda" hasta que llegue a una posición donde el elemento que se encuentre a la izquierda de éste sea menor o igual a él. Es decir, se coloca en una posición donde se encuentre "ordenado". Cuando finaliza la ejecución del algoritmo, todos los elementos en el arreglo cumplen la condición de que el elemento anterior a ellos es menor o igual al elemento en cuestión, por lo tanto, el arreglo se encuentra ordenado.

La estrategia utilizada es **incremental** debido a que se resuelve un problema independiente del otro en cada iteración, es decir, se ordena un número a la vez.

Quick sort (Divide y vencerás)

En este programa se implementa el algoritmo de ordenamiento **quick sort**. En éste, se divide en dos el arreglo a ordenar, y se realizan dos llamadas recursivas para ordenar las divisiones. La parte fundamental en este algoritmo es la partición de los datos, ya que se debe elegir un pivote en cada división del arreglo, es decir, cada que se ejecuta la función. Lo anterior se realiza con el objetivo de cambiar la posición de los datos que se encuentran en una posición incorrecta con respecto al pivote. Finalmente, cuando el algoritmo finaliza su ejecución, el arreglo original se encuentra completamente ordenado.

La estrategia utilizada es **divide y vencerás**, ya que el problema se divide en subproblemas más pequeños **(la partición del arreglo),** los cuales son más fáciles de resolver que el problema original, "combinando" las soluciones de cada subproblema en cada llamada a la función.

ACTIVIDAD 2: CALENDARIZACIÓN DE ACTIVIDADES

I) INDICA A CUÁL DE LAS ESTRATEGIAS DE CONSTRUCCIÓN DE ALGORITMOS PERTENECE EL PROBLEMA Y REALIZA LA IMPLEMENTACIÓN EN PYTHON.

A continuación, se muestra la implementación en Python del programa:

```
def activities(s, f, n):
    print("Selected activities are:")
    i = 0
    print("A1")

    for j in range(1,n):

        if s[j] >= f[i]:
            print("A" + str(j+1))
            i = j


s = [1, 2, 3, 2, 4, 5, 6, 8, 7]
f = [4, 5, 6, 8, 6, 7, 7, 12, 9]

n = len(s)
activities(s, f, n)
```

Código fuente

```
Selected activities are:
A1
A5
A7
A8
```

Salida del programa

El algoritmo funciona de la siguiente forma:

1. Se selecciona la primera actividad en el arreglo s, donde se encuentran las horas de inicio.

2. Se recorre el arreglo **s**, comparando cada elemento de éste (horas de inicio) con la hora de término de la última actividad seleccionada. Si la hora de inicio es mayor o igual a la hora de término de la última actividad seleccionada, entonces es posible

realizar la actividad con la hora de inicio analizada, por lo que se imprime el número de la actividad y se actualiza la hora de término de la última actividad.

Debido a que se recorre el arreglo considerando cada hora de inicio por separado, en mi opinión, la estrategia que se utiliza es **incremental**, ya que, como se menciona en la guía de estudio, en cada "paso" se agrega información al resultado final, en caso de ser posible. Además, el problema original no se ataca mediante subproblemas. No considero que la estrategia utilizada se considere fuerza bruta, ya que no se intentan todas las soluciones posibles. Es posible que también se considere un algoritmo ávido, ya que en cada iteración se agregan las actividades que sean posibles, conforme se presenten las actividades en el arreglo. Sin embargo, el número de actividades escogidas al finalizar la ejecución del algoritmo no es el máximo posible, por lo que, respecto a este criterio, no se podría considerar ávido.

ACTIVIDAD 3: MINMAX

CODIFICA Y EJECUTA EL SIGUIENTE PROGRAMA, INDICA LA SALIDA, LO QUE HACE EL PROGRAMA Y PORQUÉ PERTENECE A LA CATEGORÍA DIVIDE Y VENCERÁS.

A continuación, se muestra la implementación del programa:

```
def minMax(L):
    if len(L) == 1:
        return (L[0], L[0])

    elif len(L) == 2:
        if L[0] <= L[1]:
            return (L[0], L[1])
        else:
            return (L[1], L[0])

    else:
        mid = len(L) / 2
        (minL, maxL) = minMax(L[:int(mid)])
        (minR, maxR) = minMax(L[int(mid):])
        if minL <= minR:
            min = minL
        else:
            min = minR

        if maxL >= maxR:
            max = maxL
        else:
            max = maxR

        return (min, max)


lista = [3, 10, 32, 100, 4, 76, 45, 32, 17, 12, 1]
print("Los valores son: " + str(minMax(lista)))
```

Código fuente

Salida del programa

Este algoritmo calcula el valor máximo y el valor mínimo presentes en una lista de números. Su funcionamiento es el siguiente:

Casos base:

Lista de longitud uno: se retorna una tupla con el valor del único elemento, duplicado.

Lista de longitud dos: se retorna una tupla con el elemento menor en la posición 0, y el elemento mayor en la posición 1.

Dados los casos base del algoritmo, los pasos a seguir en el caso general son los siguientes:

1. Se divide la lista en dos partes.
2. Se ejecuta la función recursivamente para cada mitad, obteniendo así el mínimo y máximo de cada parte.
3. Se comparan los mínimos de las dos mitades y se determina cuál es el mínimo del arreglo completo. Se realiza el procedimiento con los máximos.
4. Se retorna una tupla con el mínimo y el máximo, respectivamente.

Dado el análisis del algoritmo, es fácil determinar que la estrategia que se utiliza en él es **divide y vencerás**, ya que se divide el problema en subproblemas más pequeños, es decir, el arreglo se divide a la mitad: se realizan dos llamadas recursivas con cada una. Estos subproblemas son más fáciles de resolver, ya que su tamaño es menor al del tamaño original. Lo anterior es una propiedad clara de los algoritmos de este tipo.

ACTIVIDAD 4: MERGE SORT

CODIFICA Y EJECUTA EL SIGUIENTE PROGRAMA, INDICA LA SALIDA, LO QUE HACE EL PROGRAMA Y PORQUÉ PERTENECE A LA CATEGORÍA DIVIDE Y VENCERÁS.

A continuación, se muestra la implementación del programa:

```python
def merge(left, right):
    result = []
    left_idx, right_idx = 0, 0
    while left_idx < len(left) and right_idx < len(right):
        if left[left_idx] <= right[right_idx]:
            result.append(left[left_idx])
            left_idx += 1
        else:
            result.append(right[right_idx])
            right_idx += 1

    if left:
        result.extend(left[left_idx:])
    if right:
        result.extend(right[right_idx:])

    return result

def merge_sort(m):
    if len(m) <= 1:
        return m

    middle = len(m) // 2
    left = m[:middle]
    right = m[middle:]
    left = merge_sort(left)
    right = merge_sort(right)
    return list(merge(left, right))

lista1 = [4, 12, 87, 1, 32, 54, 36, 78, 90, 7]
print(merge_sort(lista1))
```

Código fuente

```
[1, 4, 7, 12, 32, 36, 54, 78, 87, 90]
```

Salida del programa

En este programa se implementa el algoritmo de ordenamiento **merge sort**. Su funcionamiento es el siguiente:

Caso base:

Lista de longitud uno: Debido a que la lista únicamente tiene un elemento, ya se encuentra ordenada.

Dado el caso base del algoritmo, los pasos a seguir en el caso general son los siguientes:

1. Se divide la lista en dos partes.
2. Se ejecuta la función recursivamente para cada mitad, ordenándolas.
3. Se “unen” ambas mitades, utilizando la subrutina **merge**.

La subrutina **merge** es el punto más importante en el algoritmo. Su función es combinar las dos listas (o mitades) ya ordenadas, iterando sobre ambas y determinando los elementos a colocar en el arreglo final mediante comparaciones.

Al igual que en la actividad anterior, es fácil determinar que la estrategia que se utiliza en éste es **divide y vencerás**, ya que se divide el problema en subproblemas más pequeños, es decir, el arreglo se divide a la mitad: se realizan dos llamadas recursivas con cada una. Estos subproblemas son más fáciles de resolver, ya que su tamaño es menor al del tamaño original. Finalmente, se ejecuta la subrutina **merge**, la cual produce el resultado deseado. Podemos observar claramente el patrón de **divide** *(llamadas recursivas para ordenar las mitades del arreglo)* y **conquista** *(merge: combinar ambas soluciones).*

CONCLUSIONES

Gracias a la realización de la práctica logré comprender de forma un poco más profunda las diferentes estrategias de diseño de algoritmos, además de sus implicaciones y cómo implementarlas en diferentes casos. Los algoritmos propuestos en la guía de laboratorio me fueron útiles para analizar las características propias de cada estrategia, además de observar un ejemplo real de implementación en cada caso. Lo anterior, sin duda alguna, facilitó la realización de las actividades restantes, ya que logré identificar algunos patrones o características similares a las encontradas en los ejercicios propuestos.

En la actividad número 2 me fue un poco complicado determinar la estrategia utilizada en el algoritmo, ya que, en mi opinión, se distinguen algunos rasgos (mencionados en el desarrollo) que no son propios de una sola estrategia. Lo anterior es un claro ejemplo de que es posible que los algoritmos tengan características de más de un paradigma de diseño de algoritmos. En cambio, en las actividades 3 y 4 los rasgos de la estrategia **divide y vencerás** fueron muy fáciles de identificar: el problema original se dividía en subproblemas, que facilitaban su resolución.

Los ejercicios me parecieron aptos para obtener un poco de práctica en el análisis de algoritmos y la identificación de sus estrategias de diseño. Sin embargo, pienso que no es posible comprender e identificar profundamente las estrategias si no se analiza una cantidad mayor de ejemplos concretos para cada caso.

Es necesario identificar las distintas estrategias o paradigmas de diseño de algoritmos, para facilitar su análisis, o bien, para diseñarlos de forma eficiente y correcta.