	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorio de docencia

Laboratorios de computación salas A y B

Profesor: Edgar Tista Garcia

Asignatura: Estructuras de Datos y Algoritmos I

Grupo: 2

No. de práctica(s): 10 - 12

Integrante(s): Álvarez López Carlos Manuel

No. de lista o brigada: 2

Semestre: 2023-2

Fecha de entrega: Martes 6 del 2023

Observaciones:

CALIFICACIÓN: _____

OBJETIVO: Implementar, al menos, dos enfoques de diseño (estrategias) de algoritmos y analizar diferencias de cada uno de ellos

OBJETIVO DE CLASE: Identificar algunos ejemplos de algoritmos y asociarlos con las estrategias más adecuadas

Actividad 1.- Ejercicios de la guía

1.- Explica en tus palabras en qué consiste la estrategia fuerza bruta

La estrategia de fuerza bruta consiste en tratar con todas las combinaciones de elementos posible hasta llegar a una solución satisfactoria, en este sentido el principal problema de esta estrategia es el tiempo de ejecución, ya que se están buscando todas las posibles combinaciones que nos lleven a una solución recorriendo cada combinación posible.

2.- Ejecuta al menos 5 veces el programa de ejemplo e indica el tiempo promedio de ejecución. Explica el incremento del tiempo si el tamaño de la contraseña (agrega alguna captura de pantalla de la ejecución)

```
Tu contraseña es H0l4
Tiempos de ejecución 3.398203
Ecuador33:Downloads edaI02alu02$ /usr/local/bin/python3 /Users/edaI02alu02/Downloads/FuerzaBruta.py
Tu contraseña es H0l4
Tiempos de ejecución 3.361648
Ecuador33:Downloads edaI02alu02$ /usr/local/bin/python3 /Users/edaI02alu02/Downloads/FuerzaBruta.py
Tu contraseña es H0l4
Tiempos de ejecución 3.371532
Ecuador33:Downloads edaI02alu02$ /usr/local/bin/python3 /Users/edaI02alu02/Downloads/FuerzaBruta.py
Tu contraseña es H0l4
Tiempos de ejecución 3.318867
Ecuador33:Downloads edaI02alu02$ /usr/local/bin/python3 /Users/edaI02alu02/Downloads/FuerzaBruta.py
Tu contraseña es H0l4
Tiempos de ejecución 3.647331
```

El promedio de ejecución del programa fue de 3.4 segundos.

```
Tu contraseña es H0l45
Tiempos de ejecución 225.814704
```

Si aumentamos un carácter a la contraseña y modificamos el código para que permita buscar contraseñas de 5 caracteres, el tiempo de ejecución aumenta sustancialmente, yendo a los 225 segundos y fracción, más de 60 veces el tiempo de ejecución cuando la contraseña era de 4 caracteres, y tiene algo de sentido ya que el número de combinaciones posibles se multiplicó por el número de posibles caracteres que puede tomar el último carácter añadido a la contraseña, que es muy cercano a 60.

3.- Investiga otros dos problemas computacionales que se resuelvan por fuerza bruta

Búsqueda de subcadenas
Si se tienen dos cadenas, y se desea saber si una cadena esta contenida dentro de la otra y si está en alguna posición, se puede aplicar un algoritmo de fuerza bruta, deslizando la cadena a buscar de izquierda a derecha a lo largo de la cadena en la que se está buscando, hasta encontrar una coincidencia.

Selection sort
Es método de ordenamiento usa fuerza bruta, buscando en un conjunto de datos el dato de menor valor e intercambiandolo por el que está en primera posición, después sigue con el segundo menor valor y lo intercambia por aquel en segunda posición y así respectivamente con el resto de elementos.

4. Explica qué hace el programa de ejemplo de algoritmos tipo “greedy”.

En el programa se crea una función para calcular qué cantidad de cada moneda de cada denominación se necesitan para tener cierta cantidad. Reuniendo el resultado en una lista que guarda listas de dos elementos, en las cuales se tienen 2 elementos, la denominación de la moneda y la cantidad de ellas.

El funcionamiento de la función consiste en usar la división entera entre la cantidad a calcular y la primera denominación dada, para después restar a la cantidad el resultado de la división entera por la denominación y añadiendo a la lista que se retornara la denominación y la cantidad de esa moneda que se requiere para la combinación, siguiendo así con la siguiente denominación hasta que la cantidad sea 0.

4.1 ¿Qué hacen las líneas de código?

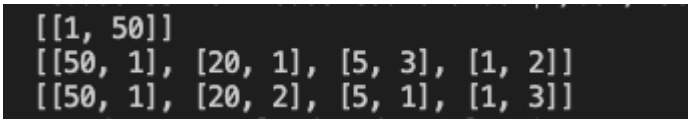
```
if cantidad >= denominaciones [0]:
...
denominaciones = denominaciones[1:]
```

La primera comprueba si la cantidad es mayor a la denominación en el primer elemento de la lista de denominación, ya que esta debe de ser mayor para que la cantidad para que se le pueda restar aunque sea una, de lo contrario se debería pasar a la siguiente denominación.

La segunda línea mostrada lo que hace es recorrer los elementos de la lista a su localidad anterior, eliminando el primer elemento. Esta línea se ejecuta al final del ciclo, cuando ya se ha restado y guardado la cantidad de elementos de una denominación a la cantidad, por lo que se debe pasar a la siguiente que se consigue

con esta línea y por eso siempre se accede al primer elemento de la lista de denominaciones.

5.- Ejecuta el programa con otras denominaciones de monedas y anota tus observaciones



Si en la lista de denominaciones no se colocan las denominaciones en orden decreciente, el programa no calcula la solución donde se usen menos monedas, y si es 1, directamente dirá que se necesitan la cantidad introducida de monedas de 1.

Del mismo modo, si no es posible conseguir la cantidad buscada con las denominaciones indicadas, el programa fallará.

6.- Investiga en qué consiste la estrategia “incremental” (amplía la explicación que viene en la guía)

La estrategia de incremental se podría interpretar como que va añadiendo valores de entrada con el tiempo. En este sentido tomando como ejemplo el algoritmo de insertion sort, primero se toma como que el primer elemento ya está ordenado, y después se agrega el siguiente elemento para solo ordenar ese en el subarreglo ya ordenado; repitiendo este proceso añadiendo de uno en uno los elementos del arreglo hasta tener todo el arreglo ordenado.

Se podría interpretar como que esta estrategia va resolviendo la tarea en pequeñas sub porciones, una vez resuelve una introduce otra porción del problema a la implementación de la solución, comprobando a cada paso que la subsolucion sea correcta.

7.- ¿Por qué el ejemplo de ordenamiento “insertionSort” que se proporciona, pertenece a esta estrategia?

Este algoritmo pertenece ya que sigue con la estrategia de resolver una fracción del problema, y después agregar más fracciones del mismo a la solución. En este caso cada fracción del problema es un elemento del arreglo a ordenar, aunado a que con cada iteración se agrega un nuevo elemento para ordenar y así mismo se asegura que el arreglo está ordenado a cada iteración.

Actividad 2.- Calendarización de actividades

El problema de calendarización de actividades es un problema de optimización relativo a la selección de actividades “no conflictivas” a realizar dentro de un marco de tiempo determinado. Dado un conjunto de actividades cada una marcada por un tiempo de inicio y un tiempo de finalización. El problema es seleccionar el número máximo de actividades que pueden ser realizadas por una persona, suponiendo que sólo puede trabajar en una sola actividad a la vez

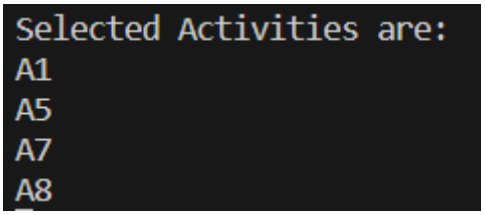
a) Analiza e interpreta la implementación proporcionada, explica a que se refieren los arreglos que se muestran y la variable “n”

> Los arreglos s y f son respectivamente las horas de comienzo(start) y finalización(finish) de las actividades, con estas ordenadas coincidiendo su índice con el número de actividad menos 1. La variable “n” por otro lado representa el número de actividades; este se obtiene al dividir el tamaño en memoria del arreglo entre el tamaño de uno de sus elementos, un método usual para obtener el tamaño de un arreglo en lenguaje c.

b) Indica a cuál o a cuáles de las estrategias de construcción de algoritmos pertenece este problema

La principal estrategia para la construcción de este algoritmo considero que es la avida. Esto debido a que el algoritmo lo que hace es introducir la primera actividad cuya hora de comienzo sea mayor o igual a la hora de finalización de la última actividad registrada, para esto da por hecho que estamos disponibles para la primera actividad y con respecto al horario de esta va comprobando las horas de las actividades para ir agregando cada una. Este funcionamiento trata de maximizar las actividades registradas tomando la primera que esté disponible. Aun así debido a su naturaleza de analizar cuál es la mejor decisión localmente, puede causar ineficiencias como horas entre actividades, o que no se acuda a la mayor cantidad de actividades por que primero el algoritmo incluyó una que abarca el tiempo que podrían abarcar otro conjunto mayor de actividades.

c) Realiza la implementación ahora en PYTHON. (formato libre)



Dado a que se trató de transportar la función lo más parecido al lenguaje de Python, el funcionamiento de la misma es similar a su contraparte en C.

Se define la función activities(), que recibe como parámetros las tuplas con las horas de comienzo y finalización de las actividades respectivamente, además del número de actividades.

La función imprimirá un mensaje para indicar que lo siguiente en imprimirse serán las actividades seleccionadas por la función. Se iguala la variable i a 0, y se imprime “A1” dado que la primera actividad siempre se podrá asistir por que no solapa a ninguna. Enseguida se entra a un ciclo for que itera j en una lista de 0 hasta n, con cada iteración se evalúa “if s[j] >= f[i]” pudiendo interpretarse como si la hora de comienzo de la actividad con índice “j” es mayor o igual a la hora de finalización de la actividad con índice “i”, la última actividad seleccionada. En caso de que la expresión sea verdadera, entonces se indica imprimiendo en pantalla que la actividad j+1, también es seleccionada y se iguala i a j, para indicar que es la última actividad seleccionada. Se repite este proceso con todas las actividades resultando en un conjunto de actividades que no se sobreponen.

Actividad 3.- Analizando un algoritmo

3.1.- Codifica y ejecuta el siguiente 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)
def principal():
    lista=[3, 10, 32, 100, 4, 76, 45, 32, 17, 12, 1]
    print("los valores son: ",MinMax(lista))
principal()
```

a) Indica qué hace el programa y la salida que muestra (si es necesario agrega o quita valores en la lista para ver el funcionamiento

El objetivo del programa es imprimir el valor máximo y mínimo dentro de una lista. Para esto hace uso de la función MinMax(), la cual es una función recursiva. La función recibe como argumento la lista, y devuelve tanto el valor máximo como mínimo. Los casos base de la función es cuando el arreglo recibido es de longitud 1, y devuelve que el único elemento del arreglo es el máximo y mínimo; también tiene el caso base cuando sólo hay 2 elementos en el arreglo, para lo cual compara ambos elementos dado que si el de índice 0 es mayor que el de índice 1, se devuelve el primero como máximo y el segundo como mínimo y viceversa.

En caso de que la longitud del arreglo sea mayor a dos, se determina la mitad del arreglo dividiendo su longitud entre 2 y asignando ese valor a la variable “mid”. Seguido de esto se llama recursivamente a MinMax(), tomando ahora desde despues del indice mid hasta el final del arreglo, y de nuevo pero desde el principio del arreglo hasta el índice mid. Enseguida compara los resultados de las llamadas del lado izquierdo y derecho para determinar el máximo y mínimo entre ambas mitades y finalmente devolver el menor y el mayor de entre ambos y por lo tanto del arreglo en general.

b) Indica qué hace la instrucción L[:int(mid)]

> Establece el rango del arreglo desde un índice después del entero de la variable mid, hasta el final del arreglo.

c) ¿Por qué el programa pertenece a la categoría divide y vencerás?-¿Podría pertenecer a otra categoría? ¿Por qué?

> Este programa pertenece a su categoría debido a que la función divide el arreglo en subarreglos más pequeños hasta tener un arreglo que pueda resolver directamente como son los de longitud de uno o dos elementos.
>
> No considero que el algoritmo pueda pertenecer directamente a otra categoría directamente ya que su principal funcionamiento es fundamentalmente el de divide y vencerás, y aunque pueda presentar algunas características de otras estrategias, como la toma de decisiones basada en condiciones como en greedy en el sentido de hacer una operación u otra como con los casos bases; o como la división en subproblemas y el uso de soluciones parciales de la programación dinámica. Aun así considero que hay factores faltantes para considerar al algoritmo de estas categorías, como el solapamiento de subproblemas por parte de la programación dinámica o que el algoritmo no selecciona la opción óptima a cada paso como con un enfoque greedy.

4.- Algoritmo de ordenamiento por merge-sort

A continuación, se muestra una implementación del algoritmo de ordenamiento por “merge sort”.

a) Codifica y ejecuta el programa (si es necesario modifica algunas líneas para entender su funcionamiento

> Este programa consiste en el algoritmo de ordenamiento merge sort, en este se tiene a la función recursiva merge_sort() la cual recibe el arreglo a ordenar y como caso base tiene que si el arreglo es de longitud de 1, la función retorna el arreglo directamente. Fuera del caso base se crean dos subarreglos, el izquierdo y derecho tomando como separación el elemento medio del arreglo original, seguido de esto se llama recursivamente a la función merge_sort con cada subarreglo creado, de esta manera se continúa dividiendo cada subarreglo hasta tener arreglos de un solo elemento. Seguido de esto se empiezan a combinar cada subarreglos con la función merge() que la cual recibe dos arreglos y devuelve un arreglo con los elementos de ambos ordenados, para esto se ocupan dos variables para recorrer los subarreglos y un ciclo while que itera hasta que ambas variables hayan recorrido sus respectivos arreglos completamente; dentro del ciclo se evalúa cual de los elementos de los arreglos con su respectivo índice es menor, y este se agrega al arreglo a retornar; enseguida el índice del subarreglo del cual se agregó el elemento se aumenta y se repite la iteración hasta distribuir todos los elementos en el arreglo solución. Esta función realiza un retroceso recursivo con todos los subarreglos creados por las llamadas iterativas de merge_sort() hasta retornar el arreglo ordenado.

b) Indica a qué estrategia(s) de construcción de algoritmo pertenece este algoritmo y por qué

> La estrategia utilizada para este algoritmo es divide y vencerás, una vez se analiza que el objetivo de las llamadas recursivas es llegar al caso base cuando se tienen arreglos de un solo elemento que por lo mismo ya están ordenados, se hace claro el enfoque del algoritmo es divide y vencerás, dividiendo el arreglo a ordenar un subarreglos más sencillos de ordenar hasta llegar a uno que directamente ya esta ordenado.

Conclusiones

Ya habiendo concluido satisfactoriamente con todos los ejercicios de la práctica, puedo afirmar que los objetivos de la misma se cumplieron, esto gracias a que durante los ejercicios se analizaron varios enfoques de diseño y se comprendieron las razones de su aplicación así como las ventajas y desventajas de cada uno en

distintos entornos. A su vez se desarrolló la capacidad de análisis para identificar cual de este enfoque podría ayudar para llegar a una solución óptima.

A lo largo de la realización de los ejercicios y el analisis del codigo proporcionado, se deja en claro varias señales que nos dan a entender cuándo y por qué un enfoque puede ser usado efectivamente para resolver un problema, así como la posibilidad de descomponer un problema en varios más simples pero del mismo tipo, o la posibilidad de crear progresivamente una solución analizando cada paso como un caso aislado.

Los distintos enfoques son de gran ayuda para orientarnos hacia una óptima solución a un problema, siendo un recurso a tener en cuenta al afrontar un problema y dar con una solución satisfactoria y eficiente según el caso que se presente.