Nombre: Daniel Andrés Vasquez Murillo
Código de estudiante: 8963154

Parte Teorica - Tarea 4

1.

Exercises

1. Describe recursive algorithms for the following generalizations of the SUBSETSUM problem:
   (a) Given an array $X[1..n]$ of positive integers and an integer T, compute the *number* of subsets of X whose elements sum to T.
   (b) Given two arrays $X[1..n]$ and $W[1..n]$ of positive integers and an integer T, where each $W[i]$ denotes the *weight* of the corresponding element $X[i]$, compute the *maximum weight* subset of X whose elements sum to T. If no subset of X sums to T, your algorithm should return $-\infty$.

A)

Especificación

Entrada: Un arreglo X [1..n) de enteros positivos y un entero positivo T >= 1

Salida: El número de subconjuntos de X que suman T

Ejemplo

Dado el siguiente ejemplo X = [2,4,1,3,5] para un T = 6, la cantidad de subconjuntos que sumados sus elementos suman 6 es de 3 y los subconjuntos respectivos son:

- [2,4]
- [2,1,3]
- [1,5]

Estrategia

La estrategia de reintento es tener dos caminos para construir las ramas del árbol de soluciones, las cuales son, recorrer de izquierda a derecha el arreglo X de números, del cual se toma y no se toma el valor en el que se esté en un determinado momento, donde cada solución es la suma de los elementos que se han tomado por uno de los dos caminos, para podar se utiliza una verificación que dirá si la suma solución es mayor que el numero T, esa rama ya no se sigue generando.

Código

```python
def subconjuntos(lista_numeros, indice, T, suma_solucion):
    respuesta = 0
    if suma_solucion == T:
        respuesta = 1
    elif suma_solucion < T and indice < len(lista_numeros):
        respuesta += subconjuntos(lista_numeros, indice + 1, T, suma_solucion)
        respuesta += subconjuntos(lista_numeros, indice + 1, T, suma_solucion + lista_numeros[indice])
    return respuesta
def main():
    X = [2,4,1,3,5]
    T = 6
    print(subconjuntos(X, 0, T, 0))
main()
```

```
PROBLEMS 20  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

ea_4_ADA/Codigos_teoricos.py
3
PS C:\Users\dhasc\Desktop\Tarea_4_ADA>
```

B)

Especificación

Entrada: Un arreglo X [1..n) y un arreglo W [1..n) de enteros positivos, donde W[i] representa el peso del elemento X[i]. Además de, un entero positivo T >= 1.

Salida: El subconjunto de X que tenga los elementos más pesados que suman T,

Ejemplo

Dado el siguiente ejemplo X = [2,4,1,3,5] y W = [15,2,10,20,5] para un T = 6, el subconjunto con los elementos más pesados que suman 6 es:

- [2,1,3]

Estrategia

La estrategia de reintento es tener dos caminos para construir las ramas del árbol de soluciones, las cuales son, recorrer de izquierda a derecha el arreglo X de números, del cual se toma y no se toma el valor en el que se esté en un determinado momento, donde cada solución es la suma de los elementos que se han tomado por uno de los dos caminos, para podar se utiliza una verificación que dirá si la suma solución es mayor que el numero T, esa rama ya no se sigue generando y para devolver el subconjunto más pesado lo que se hace es que una vez un subconjunto sume T se escoge o reemplaza ese subconjunto verificando en el arregle W cada elemento.

Código

```python
respuesta = float("-inf")
peso_mayor = 0
def subconjuntos_pesos(lista_numeros, pesos, indice, T, suma_solucion, sub_solucion, peso_solucion):
    global respuesta, peso_mayor
    if suma_solucion == T:
        if peso_solucion > peso_mayor:
            respuesta = sub_solucion
            peso_mayor = peso_solucion
    elif suma_solucion < T and indice < len(lista_numeros):
        subconjuntos_pesos(lista_numeros, pesos, indice + 1, T, suma_solucion, sub_solucion, peso_solucion)
        sub_solucion_plus = list(sub_solucion)
        sub_solucion_plus.append(lista_numeros[indice])
        subconjuntos_pesos(lista_numeros, pesos, indice + 1, T, suma_solucion + lista_numeros[indice], sub_solucion_plus, peso_solucion + pesos[indice])
def main():
    global respuesta
    X = [2,4,1,3,5]
    W = [15,2,10,20,5]
    T = 6
    subconjuntos_pesos(X, W, 0, T, 0, [], 0)
    print(respuesta)
main()
```

```
PROBLEMS 24  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\dhasc\Desktop\Tarea_4_ADA> & C:/Users/dhasc/AppData/Local/Programs/Python/Python311/python.exe c:/Users/dhasc/Desktop/Tarea_4_ADA/Codigos_teoricos.py
[2, 1, 3]
PS C:\Users\dhasc\Desktop\Tarea_4_ADA>
```

2.

6. This problem asks you to design backtracking algorithms to find the cost of an optimal binary search tree that satisfies additional balance constraints. Your input consists of a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches for $A[i]$. This is exactly the same cost function as described in Section 2.8. But now your task is to compute an optimal tree that satisfies some additional constraints.

   (a) ***AVL trees*** were the earliest self-balancing balanced binary search trees, first described in 1962 by Georgy Adelson-Velsky and Evgenii Landis. An AVL tree is a binary search tree where for every node v, the height of the left subtree of v and the height of the right subtree of v differ by at most one.

   Describe a recursive backtracking algorithm to construct an optimal AVL tree for a given set of search keys and frequencies.

   (b) ***Symmetric binary B-trees*** are another self-balancing binary trees, first described by Rudolf Bayer in 1972; these are better known by the name ***red-black trees***, after a somewhat simpler reformulation by Leo Guibas and Bob Sedgwick in 1978. A red-black tree is a binary search tree with the following additional constraints:

   - Every node is either red or black.
   - Every red node has a black parent.
   - Every root-to-leaf path contains the same number of black nodes.

   Describe a recursive backtracking algorithm to construct an optimal red-black tree for a given set of search keys and frequencies.

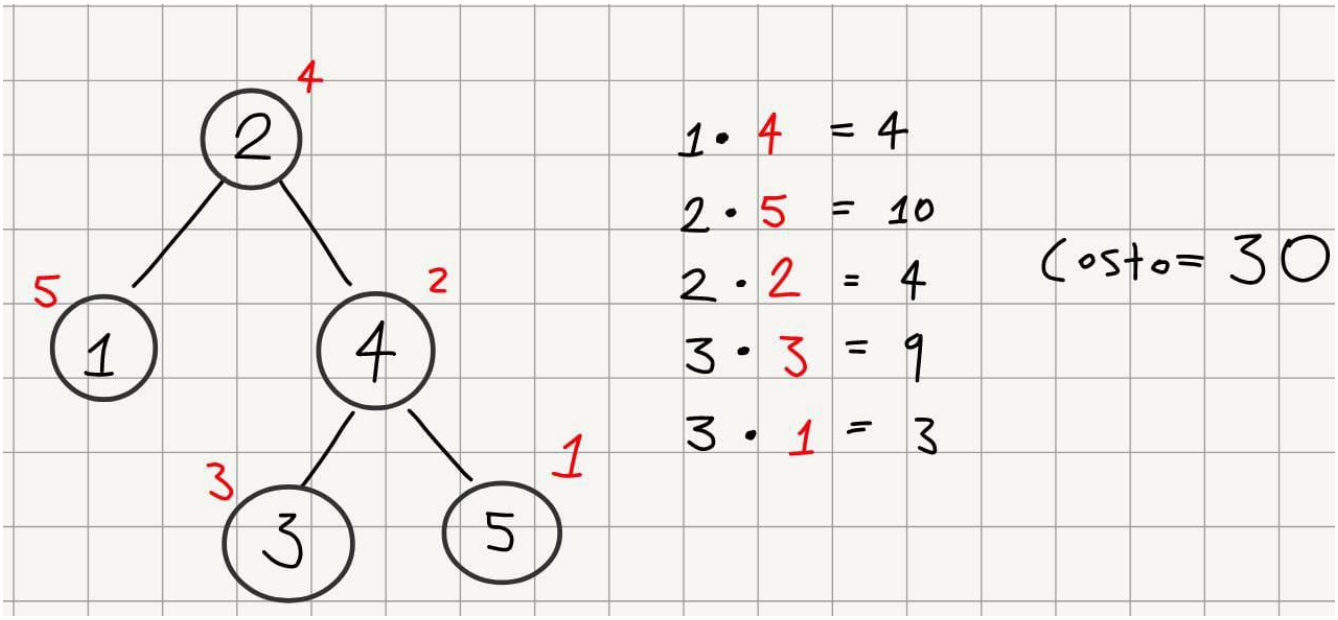
A)

Especificación:

Entrada: Un arreglo ordenado A[1..n) correspondiente al valor de los nodos, así como un arreglo F[1..n) correspondiente a la frecuencia de accesos en A, tal que n >= 1 y donde hay exactamente una frecuencia de accesos F[i] para cada nodo A[i].

Salida: Un óptimo árbol T de búsqueda binaria AVL de costo mínimo, así como un entero positivo K que representa el costo de las búsquedas en el árbol.

Ejemplo: Dado el siguiente ejemplo A = [1,2,3,4,5] y F = [5,4,3,2,1] el árbol de búsqueda AVL de costo mínimo es el siguiente:



Estrategia: La estrategia a seguir para poder hacer el mayor numero de podas posibles es primero que todo utilizar un algoritmo recursivo o una ecuación que dado un arreglo ordenado de menor a mayor (para este problema el arreglo es A el cual tiene el valor de los nodos) decida cuales son los nodos que pueden ser utilizados como nodos raíz para la construcción de un árbol óptimo T de búsqueda binaria AVL sin tener en cuenta los costos, Luego de eso se utiliza otra función recursiva para ahora si realizar la elaboración del árbol óptimo de búsqueda binaria AVL de costo mínimo **,** para lo que haremos uso en cada llamado recursivo a los nodos raíz posibles para la construcción de cada rama del árbol, con el ejemplo anterior entonces se tendría que dado un A = [1,2,3,4,5] y utilizando la función que encuentra los nodos raíz posibles para cada rama, se encontraría que para el primer nodo raíz los valores posibles son [2,3,4] ahora con eso en mente y teniendo constancia de en que nivel nos encontramos del árbol, como apenas se está construyendo entonces se asume que su nivel de profundidad es 1, entonces utilizando la lista de frecuencias para cada valor se tiene que las frecuencias para 2 = 4, para 3 = 3 y para 4 = 2, ahora para elegir el nodo raíz más optimo se deben multiplicar estas frecuencias con el respectivo nivel del árbol, por lo que tendríamos algo tal que así: 1x4 = 4, 1x3 = 3 y 1x2 = 2, luego de tener los cálculos realizados se procede a tomar como nodo raíz al que dio el valor con la multiplicación más alta, en este caso el nodo de valor 2 que tiene frecuencia de 4, posterior a eso se debe hacer el mismo procedimiento para cada rama del árbol resultante, para la izquierda se evaluaría que no nodo es

posible entre A = [1] y para la rama derecha evaluaría que nodo es posible para A= [3,4,5], para seguidamente realizar la misma operación y para ir construyendo el árbol en queda decisión

Seudocódigo:

```
arbol T

costo_minimo = 0

función arbol_avl (A, F, nivel):

global arbol T, costo_minimo

    si len(A) es igual a 1 entonces:

        agregar A[0] al arbol T

    si no entonces:

        nodos_raiz = encontrar_nodos_raiz(A)

        mejor_nodo_raiz = None

        frecuencia = 0

        for i en nodos_raiz:

            si F[i] * nivel > frecuencia entonces:

                mejor_nodo_raiz = i

                frecuencia = F[i] * nivel

        agregar A[mejor_nodo_raiz] al abol T

        costo_minimo += frecuencia

        arbol_avl (A[: mejor_nodo_raiz], F, nivel + 1) # rama izquierda

        arbol_avl (A[mejor_nodo_raiz +1:], F, nivel + 1) # rama derecha
```

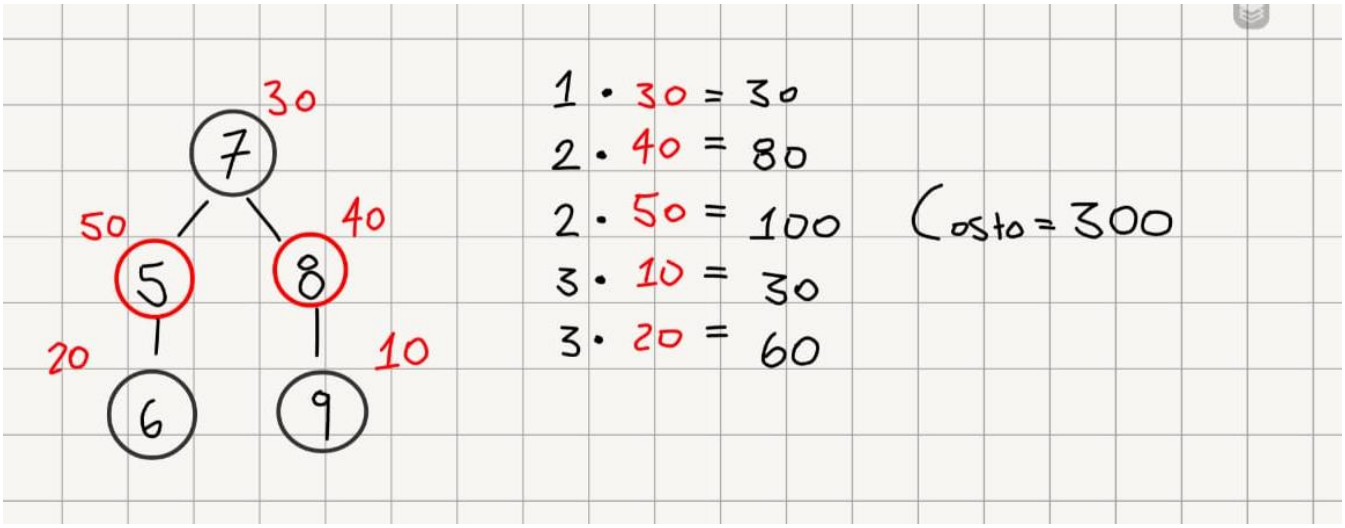
B)

Especificación:

Entrada: Un arreglo ordenado A[1..n) correspondiente al valor de los nodos, así como un arreglo F[1..n) correspondiente a la frecuencia de accesos en A, tal que n >= 1 y donde hay exactamente una frecuencia de accesos F[i] para cada nodo A[i].

Salida: Un óptimo árbol T de búsqueda binaria simétrico (árbol rojo - negro) de costo mínimo, así como un entero positivo K que representa el costo de las búsquedas en el árbol si es posible.

Ejemplo: Dado el siguiente ejemplo A = [5,6,7,8,9] y F = [50,20,30,40,10] el árbol de búsqueda simétrico de costo mínimo es el siguiente:



Estrategia: Similar al punto anterior también se debe tener un algoritmo recursivo o ecuación que permita encontrar los nodos raíz posibles para la construcción de un árbol óptimo de búsqueda binaria simétrico, para este árbol la búsqueda de nodos raíz potenciales es un poco más sencilla de encontrar usualmente están en el centro de los nodos disponibles cuando la cantidad de nodos disponibles es un numero impar mayor o igual a 3, en caso de que hayan únicamente 2 nodos disponibles esos dos son nodos raíz potenciales, luego dentro de la función recursiva que permite la construcción del árbol lo que se hace es básicamente lo mismo que en el punto anterior donde se multiplican las frecuencias de cada nodo raíz posible por el nivel del árbol y se escoge el que de el mayor resultado, con la particularidad de que al momento de elegir el nodo raíz se debe preguntar de que color es el nodo padre de la raíz, de modo que se cumplan las siguientes reglas:

Si no tiene nodo padre, el nodo raíz elegido será coloreado de negro

Si el nodo padre es de color negro, el nodo raíz elegido será coloreado de rojo

Si el nodo padre es de color rojo, el nodo raíz elegido será coloreado de negro.

Y así constantemente hasta terminar la construcción de todas las ramas del árbol.

Seudocódigo

```
arbol T

costo_minimo = 0

función arbol_simetrico (A, F, nivel, nodo_padre):

global arbol T, costo_minimo
        si len(A) es igual a 1 entonces:
                agregar A[0] al arbol T
        si no entonces:
                nodos_raiz = encontrar_nodos_raiz(A)
```

```
mejor_nodo_raiz = None
frecuencia = 0
for i en nodos_raiz:
        si F[i] * nivel > frecuencia entonces:
                mejor_nodo_raiz = i
                frecuencia = F[i] * nivel
nodo_elegido = A[mejor_nodo_raiz]
si el color del nodo_padre es None entonces:
        colorear nodo_elegido de negro
si el color del nodo_padre es negro entonces:
        colorear nodo_elegido de rojo
si no entonces:
        colorear nodo_elegido de negro
agregar nodo_elegido al abol T
costo_minimo += frecuencia
arbol_alv (A[: mejor_nodo_raiz], F, nivel + 1) # rama izquierda
arbol_alv (A[mejor_nodo_raiz +1:], F, nivel + 1) # rama derecha
```