Tarea 5

Juárez Torres Carlos Alberto

April 2, 2023

1 Problema Γ: Sea S un conjunto de n números reales y x un número real. Determinar si existen dos elementos en S cuya suma sea exactamente x.

1.1 Algoritmo para el problema *Gamma* sin suponer que S esté ordenado.

```
def existen_suma_exacta(S, x):
    n = len(S)
    for i in range(n-1):
        for j in range(i+1, n):
            if S[i] + S[j] == x:
                return True
    return False
```

Este algoritmo recorre todos los pares de elementos distintos de S y verifica si su suma es igual a x. Si encuentra un par cuya suma es exactamente x, devuelve True. Si termina de recorrer todos los pares y no encuentra ninguno cuya suma sea igual a x, devuelve False.

1.2 Complejidad del algoritmo propuesto:

Es $O(n^2)$, ya que tiene dos ciclos anidados que recorren todo el conjunto S. Por lo tanto, el tiempo de ejecución aumenta cuadráticamente con el tamaño de la entrada.

1.3 Algoritmo para el problema *Gamma* suponiendo que S está ordenado.

```
def existen_suma_exacta_ordenado(S, x):
    i = 0
    j = len(S)-1
    while i < j:
        s = S[i] + S[j]
        if s == x:
            return True
        elif s < x:
            i += 1
        else:
            j -= 1
    return False
```

Este algoritmo mantiene dos punteros, i y j, que inicialmente apuntan al primer y último elemento de S, respectivamente. En cada iteración, se calcula la suma de los elementos apuntados por los punteros. Si la suma es igual a x, el algoritmo devuelve True. Si la suma es menor que x, se incrementa el puntero i, para buscar una suma mayor. Si la suma es mayor que x, se decrementa el puntero j, para buscar una suma menor. Si los punteros se cruzan, el algoritmo devuelve False, indicando que no hay dos elementos en S cuya suma sea exactamente x.

La complejidad de este algoritmo es $O(n)$, ya que tiene un solo ciclo que recorre todo el conjunto S una única vez. Por lo tanto, el tiempo de ejecución aumenta linealmente con el tamaño de la entrada.

2 Juego de Adivinanza

2.1 Diseña y presenta una excelente estrategia para este juego.

Podemos usar busqueda binaria:

La idea básica es dividir repetidamente la lista en dos mitades y descartar una de ellas, dependiendo de si el elemento que se busca es mayor o menor que el elemento medio de la lista.

En el juego de adivinanza, la estrategia consiste en que el jugador B comience preguntando si el número pensado por el jugador A es mayor o menor que el número medio del rango (es decir, $\frac{n}{2}$). Si el jugador A responde que es menor, entonces el jugador B descarta la mitad superior del rango y repite la pregunta para la mitad inferior. Si el jugador A responde que es mayor, entonces el jugador B descarta la mitad inferior del rango y repite la pregunta para la mitad superior.

Este proceso se repite hasta que el jugador B adivina el número pensado por el jugador A. Debido a que en cada pregunta se descarta la mitad del rango, el número de preguntas necesarias para adivinar el número es logarítmico en el tamaño del rango. Por ejemplo, si el rango es de 1 a 100, se necesitarán como máximo 7 preguntas para adivinar el número.

Esta estrategia es óptima porque cada pregunta reduce a la mitad el tamaño del rango de búsqueda. De esta manera, el número de preguntas necesarias para adivinar el número crece de forma logarítmica con el tamaño del rango, lo que garantiza que la estrategia siempre encontrará el número en el menor número de preguntas posible.

2.2 Complejidad del Algoritmo

En cada paso del algoritmo, el tamaño del rango se reduce a la mitad, lo que significa que se necesitan $\log_2 n$ preguntas para adivinar el número. Por lo tanto, la complejidad del algoritmo de búsqueda binaria es $O(\log n)$.

La complejidad logarítmica del algoritmo de búsqueda binaria se debe a que cada pregunta elimina la mitad del rango de posibles números. Por lo tanto, a medida que el tamaño del rango aumenta, el número de preguntas necesarias para adivinar el número pensado por el jugador A aumenta lentamente en comparación con el tamaño del rango.

2.3 Valor de n desconocido

La estrategia consiste en hacer preguntas en las que se divida el rango en tres partes iguales en lugar de dos. En la primera pregunta, el jugador B pregunta si el número pensado por el jugador A es menor que $n/3$, mayor que $2n/3$, o está en el rango intermedio $[n/3, 2n/3]$. Dependiendo de la respuesta, el jugador B reduce el rango de búsqueda a uno de estos tres subrangos.

En la siguiente pregunta, el jugador B divide el subrango seleccionado en tres partes iguales y repite el proceso hasta que se encuentra el número pensado por el jugador A. En cada pregunta, el jugador B reduce el tamaño del rango de búsqueda en un factor de tres en lugar de dos, lo que hace que el número de preguntas necesarias crezca de forma logarítmica con el tamaño del rango.

Por ejemplo, si el rango es de 1 a 100, la primera pregunta del jugador B podría ser: ¿Es el número pensado por el jugador A menor que 33, mayor que 66, o está en el rango intermedio $[33, 66]$?. Dependiendo de la respuesta del jugador A, el jugador B reduciría el rango de búsqueda

a uno de estos tres subrangos y repetiría el proceso hasta encontrar el número pensado por el jugador A.

2.4 Complejidad del Algoritmo

Supongamos que el tamaño del rango es n. En cada paso del algoritmo, el tamaño del rango se reduce a una tercera parte en lugar de la mitad, lo que significa que se necesitan $\log_3 n$ preguntas para adivinar el número. Sin embargo, cada pregunta del jugador B implica tres posibles respuestas del jugador A en lugar de dos, por lo que cada pregunta tiene una información útil menor. Esto significa que se necesitan más preguntas en comparación con la búsqueda binaria estándar para llegar al número buscado.

3 Algoritmo de Búsqueda por Interpolación

3.1 Construye un ejemplo, con n datos, $n \geq 700$, para el cual BxI requiera de $\Omega(n)$ comparaciones para una secuencia de tamaño n

Podemos construir un ejemplo en el que la distribución de los datos no es uniforme, lo que lleva a un peor caso de complejidad de $\Omega(n)$ para la búsqueda por interpolación.

Consideremos una secuencia de n números ordenados en orden ascendente, donde los primeros $\frac{n}{2}$ números son todos iguales y tienen un valor a, y los segundos $\frac{n}{2}$ números son todos iguales y tienen un valor $b > a$.

Ahora, si buscamos el valor b en esta secuencia utilizando la búsqueda por interpolación, el algoritmo siempre elegirá el punto de interpolación en la mitad de la secuencia, ya que el valor en esa posición es igual a $a + \frac{(b-a)}{2} = \frac{a+b}{2}$. Como los primeros $\frac{n}{2}$ números son todos iguales a a, esto significa que la búsqueda por interpolación no hará ningún progreso en la mitad inferior de la secuencia, y requerirá comparar cada elemento en esa mitad antes de moverse a la mitad superior. Por lo tanto, la complejidad en este caso es de $\Omega(n)$.

3.2 Construye un ejemplo, con n datos, $n \geq 700$, para el cual BxI tenga una busqueda exitosa comparaciones para una secuencia de tamaño n

Consideremos una secuencia de n números en orden ascendente, donde los valores están distribuidos uniformemente en el rango $[1, 1000]$, es decir, el valor mínimo es 1 y el valor máximo es 1000. Además, supongamos que estamos buscando el valor x en la secuencia, donde x es un número aleatorio elegido en el rango $[1, 1000]$.

En este caso, la búsqueda por interpolación tendrá éxito con un número relativamente pequeño de comparaciones.

El algoritmo de búsqueda por interpolación funciona dividiendo la secuencia en partes y eligiendo un punto de interpolación que se acerque a x en función de la distribución de los datos. Si los datos están uniformemente distribuidos, la división de la secuencia y la elección del punto de interpolación se acercarán a x de manera uniforme, lo que lleva a una búsqueda exitosa con pocas comparaciones.

4 Se tiene un conjunto de $N = 2n + 1$ rocas, todas ellas de diferente tamaño forma y consistencia. Rocas que se ven del mismo tamaño pueden tener peso muy diferente.

Para encontrar la roca de peso mínimo, podemos usar un algoritmo de selección por comparaciones para encontrar el elemento más pequeño. Este algoritmo de selección toma $n - 1$ comparaciones para encontrar el elemento mínimo.

Para encontrar la roca de peso máximo, podemos dividir las rocas restantes en dos grupos de tamaño n. Luego, realizamos una búsqueda lineal en cada grupo para encontrar el elemento máximo en cada grupo. Esto requiere $2n - 2$ comparaciones en total. Finalmente, comparamos los dos elementos máximos encontrados para determinar cuál es el máximo global. Esto requiere una comparación adicional. En total, la búsqueda del máximo requiere $2n - 1$ comparaciones.

Por lo tanto, en total se requieren $n - 1 + 2n - 1 = 3n - 2$ comparaciones para encontrar la roca de peso mínimo y máximo. Como $3n - 2$ es menor que $3n$, podemos concluir que sí es posible encontrar las rocas de peso mínimo y máximo utilizando solo $3n$ comparaciones.

```
def min_max_rocas(rocas):
    n = len(rocas)
    # Encontrar la roca de peso minimo
    min_roca = rocas[0]
    for i in range(1, n):
        if rocas[i] < min_roca:
            min_roca = rocas[i]
    #Dividir las rocas restantes en dos grupos
    grupo_1 = rocas[1:n//2+1]
    grupo_12 = rocas[n//2+1:n]
    #Encontrar los elementos maximos en cada grupo
    max1 = grupo1[0]
    for i in range(1, n//2):
        if grupo_1[i] > max1:
            max1 = grupo_1[i]
    max2 = grupo_2[0]
    for i in range(1, n//2):
        if grupo_2[i] > max2:
            max2 = grupo_2[i]
    #Comparar los dos elementos maximos encontrados
    if max1 > max2:
        max_roca = max1
    else:
        max_roca = max2
    return min_roca, max_roca
```

5 Opcional: Problema Psi: Dados dos conjuntos ordenados de números enteros X_1, X_2 y un entero z se desea encontrar $x_1 \in X_1$ y $x_2 \in X_2$ tales que $z = x_1 + x_2$ y que se muestren tales elementos, en caso de existir.

5.1 Diseña una algoritmo que resuelva el Problema Psi.

Podemos resolver el Problema *Psi* utilizando la técnica de "Búsqueda de Dos Punteros" en X_1 y X_2. Los punteros se mueven hacia el centro de los conjuntos, comparando la suma de los elementos señalados en cada iteración con el valor objetivo z.

```
def busca_sum(X1, X2, z):
    n1, n2 = len(X1), len(X2)
    i, j = 0, n2-1  # inicializar punteros
    while i < n1 and j >= 0:
        # comparar la suma actual con z
        if X1[i] + X2[j] == z:
            return X1[i], X2[j]  # elementos encontrados
        elif X1[i] + X2[j] < z:
            i += 1  # incrementar puntero de X1
        else:
            j -= 1  # decrementar puntero de X2
    return None  # no se encontraron elementos
```

Este algoritmo toma como entrada los dos conjuntos ordenados X_1 y X_2, y el entero objetivo z, y devuelve una tupla con los elementos x_1 y x_2 si existen, o None si no se encuentran elementos que sumen z.

5.2 Indica específicamente cuántas operaciones realiza únicamente la búsqueda de los valores $x_1 \in X_1$ y de $x_2 \in X_2$.

La búsqueda de los valores $x_1 \in X_1$ y $x_2 \in X_2$ se realiza en los punteros i y j respectivamente, los cuales se mueven a través de los elementos de los conjuntos.

Cada iteración del ciclo while realiza una comparación y una actualización de punteros, por lo que se realizan en total $n_1 + n_2$ comparaciones en el peor caso. Además, las operaciones de incrementar o decrementar punteros son O(1), por lo que en total se realizan $3n$ operaciones de este tipo en el peor caso.

Por lo tanto, podemos afirmar que el algoritmo realiza $n_1 + n_2 + 3n$ operaciones en el peor caso para buscar los valores $x_1 \in X_1$ y $x_2 \in X_2$.