Mini-Proyecto

Jhonier Mendez Bravo - 2372226

Juan Pablo Pazmiño Caicedo - 2325093

David Santiago Guerrero Delgado - 2324594

Cristian Daniel Guaza Mejia- 2372225

Facultad de Ingeniería

Escuela de Ingeniería De Sistemas y Computación

Análisis y Diseño de Algoritmos I

Jesús Alexander Aranda Bueno, Ph.D.

26 de Julio del 2025

TABLA DE CONTENIDO

Introducción

Decidimos usar python como lenguaje de programación, debido a la experiencia que tenemos los miembros del grupo usándolo.

En el código se usa programación orientada a objetos, donde tenemos las clases: encuesta, tema, pregunta y encuestado.

Este paradigma se usa para almacenar y poder acceder a los datos de forma sencilla y eficaz. Podríamos haber almacenado todo en listas de listas, pero, con dicho método sería más complicado acceder a los datos que queremos, lo cual es mucho más sencillo con los métodos get de los objetos.

Al acceder a los datos necesarios mediante los métodos get, estos no cambian su orden, por lo cual este no interfiere con los algoritmos de ordenamiento que vamos a usar, sino que simplemente facilita un poco el acceso a los datos.

Con respecto a las estructuras de datos a utilizar, se planea usar 2: Listas doblemente entrelazadas y árboles binarios de búsqueda

Listas entrelazadas

La definición de una lista entrelazada es que tiene un puntero, que apunta a la lista entrelazada anterior (None si es la primera lista), una llave y un puntero que apunta a la lista entrelazada siguiente (None si es la última lista), en código se vería de la siguiente manera:

```python
class LDE: # Clase LDE (Lista Doblemente Entrelazada)
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
```

Operaciones básicas

```python
def List_Insert(lista, llave):
    nuevo_nodo = LDE(llave)
    nuevo_nodo.next = lista
    if lista:
        lista.prev = nuevo_nodo
    return nuevo_nodo

def List_Insert_End(lista, data):
    nuevo = LDE(data)
    if lista is None:
        return nuevo
    actual = lista
    while actual.next:
        actual = actual.next
    actual.next = nuevo
    nuevo.prev = actual
    return lista

def List_Size(lista):
    if lista is None:
        return 0
    actual = lista
    size = 1
    while actual.next:
        size += 1
        actual = actual.next
    return size
```

```
def List_Divide(lista):
    lista_izq = None
    lista_der = None
    full_size = List_Size(lista)
    mitad = full_size // 2
    contador = 0
    while contador < full_size:
        if contador < mitad:
            lista_izq = List_Insert_End(lista_izq, lista.data)
        else:
            lista_der = List_Insert_End(lista_der, lista.data)
        lista = lista.next
        contador += 1
    return lista_izq, lista_der

def List_Median(lista):
    if lista is None:
        return 0
    if List_Size(lista)%2 == 0:
        lista = List_Insert(lista, 0)
    izq, der = List_Divide(lista)
    return der.data
```

List_Insert:
Esta función recibe una llave y una lista, la llave la vuelve una LDE y luego se conectan los punteros. Su complejidad es O(1).

List_Insert_End:
Esta función recibe igual una llave y una lista, se crea un LDE con la llave y luego con la lista recorremos hasta su último nodo, ajustamos los punteros next y prev para enlazar el LDE que creamos con la llave al final de la lista. Su complejidad es O(n).

List_Size:
Recorre toda la lista que le pasemos hasta que el puntero next esté vacío, en cada iteración con el while se le suma 1 al contador. Su complejidad es O(n).

List_Divide:
Calcula el tamaño total de la lista y dividimos el resultado en 2, luego recorremos la lista original desde el inicio hasta el final. La primera mitad elementos se insertan al final de lista_izq, los elementos restantes se insertan al final de lista_der (Lo de insertar al final se hace para mantener el orden de la lista original). Al final retornamos las dos listas resultantes (cabe recalcar que si la lista ingresada es par, las dos listas resultantes tendrán tamaño n/2, mientras que si la lista es impar, la lista izquierda tendrá tamaño

(n-1)/2 y la derecha (n+1)/2). Su complejidad es $O(n^2)$, porque hace n veces List_Insert_End, osea n * O(n).

List_Median:
Recibe una lista, contamos la cantidad de elementos con List_Size, si este es par, insertamos un 0 al inicio para hacerla impar (Esto porque hay dos medianas y queremos la menor), si ya era impar no hacemos nada. Luego divide la lista en dos partes con List_Divide y retorna el dato del primer nodo de la parte derecha (Esto porque la por la naturaleza de List_Divide, la mediana de una lista impar siempre se encontrará al inicio de la lista derecha). Si cuando tenemos dos medianas queremos la mayor cambiaría la función, pero este no es el caso aquí. Su complejidad es $O(n^2)$ porque usa List_Divide.

Algoritmos de ordenamiento

Se usaron dos algoritmos, insertion sort y merge sort

Merge Sort

El Merge Sort se escogió como una opción por su complejidad de O(n logn) en todos los casos, pero claro, esto para listas, por eso se quería probar si su complejidad iba a cambiar por el uso de un diferente estructura de datos, como lo es una lista entrelazada.

```python
def List_Merge(lista_izq, lista_der, metodo, orden):
    merged = None

    while lista_izq and lista_der:
        if orden == "ascendente":
            if metodo(lista_izq.data) <= metodo(lista_der.data):
                merged = List_Insert_End(merged, lista_izq.data)
                lista_izq = lista_izq.next
            else:
                merged = List_Insert_End(merged, lista_der.data)
                lista_der = lista_der.next
        if orden == "descendente":
            if metodo(lista_izq.data) >= metodo(lista_der.data):
                merged = List_Insert_End(merged, lista_izq.data)
                lista_izq = lista_izq.next
            else:
                merged = List_Insert_End(merged, lista_der.data)
                lista_der = lista_der.next

    while lista_izq:
        merged = List_Insert_End(merged, lista_izq.data)
        lista_izq = lista_izq.next
```

```
    while lista_der:
        merged = List_Insert_End(merged, lista_der.data)
        lista_der = lista_der.next

    return merged

def List_Merge_Sort(lista, metodo, orden):
    if List_Size(lista) <= 1:
        return lista

    izq, der = List_Divide(lista)
    lista_izq = List_Merge_Sort(izq, metodo, orden)
    lista_der = List_Merge_Sort(der, metodo, orden)

    return List_Merge(lista_izq, lisa_der, metodo, orden)
```

Explicación: List_Merge_Sort es un algoritmo de ordenamiento que aplica la técnica divide y vencerás sobre una lista doblemente enlazada. Primero divide la lista en dos mitades de forma recursiva hasta llegar a listas de un solo elemento (ya ordenadas). Luego, fusiona esas listas ordenadas comparando elemento por elemento según un criterio (método) y un orden (ascendente o descendente), hasta reconstruir la lista completamente ordenada.

Complejidad: El ordenamiento se logra recursivamente dividiendo la lista en mitades, ordenando cada mitad, y luego fusionándolas en una lista ordenada. Por tanto, su complejidad depende del número de divisiones (niveles de recursión) y del costo de la función List_Merge.

- **Mejor caso:** Supongamos que la lista ya está completamente ordenada según el orden solicitado (ascendente o descendente), pero a pesar de estar ordenada, merge sort no lo sabe, por lo que igualmente realiza las divisiones y fusiones. La única optimización es que, en List_Merge, la condición <= o >= se cumple siempre en la primera comparación, por lo que no hay mucho movimiento o reordenamiento extra.

  Aun así, el número de divisiones es $\log_2(n)$, y en cada nivel se recorren todos los elementos, por lo tanto la complejidad del mejor caso es O(n logn)

- **Peor caso:** El peor caso ocurre cuando la lista está totalmente desordenada según el criterio del metodo y orden. Aquí, List_Merge_Sort sigue realizando todas las divisiones ($\log_2(n)$ niveles) y, en cada nivel, la función List_Merge compara todos los elementos.

Sin embargo, lo crítico es que List_Merge llama muchas veces a List_Insert_End, y su complejidad tiene costo O(n) porque recorre toda la lista para insertar al final.

Por tanto, en cada nivel de fusión, el costo de juntar las mitades puede crecer linealmente, y hacerlo repetidamente en varios niveles implica que la complejidad del peor caso es $O(n^2 \log n)$ debido al uso repetido de List_Insert_End con costo O(n)

Conclusión: En el mejor caso el algoritmo tiene la misma complejidad que el Merge Sort normal, pero en el peor caso, el uso ineficiente de List_Insert_End en cada fusión genera un gran número de recorridos lineales, degradando el rendimiento.

Lo que nos muestra que la estructura de datos con la que trabajemos sí afecta el rendimiento de un algoritmo, la desventaja de tener que crear una función que recorra toda una lista entrelazada para insertar un elemento al final (complejidad O(n)) hace que la potencia del Merge Sort se reduzca mucho, con respecto a su versión de listas (en las que insertar un elemento al final es O(1)).

Insertion Sort

Voy a usar el insertion sort que vimos en el curso, aunque debe ser modificado totalmente para que funcione con listas doblemente enlazadas, además de usar los métodos de la clase "Encuestado". Escogí el insertion sort debido a que es fácil de implementar, y en este punto todavía no sabía muy bien cómo usar esta estructura de datos.

```python
def lde_insertion_sort(head):
    key=head.next
    while key:
        i=key.prev
        while i and key.data.getOpinion()>=i.data.getOpinion():
            key_0=copy.copy(key)
            if key.data.getOpinion()==i.data.getOpinion():
                if key.data.getExperticia()>=i.data.getExperticia():
                    key.data=i.data
                    i.data=key_0.data
                    if i.prev:
                        key=i
                    i=i.prev
                else:
                    i=i.prev
            else:
```

```
                key.data=i.data
                i.data=key_0.data
                if i.prev:
                    key=i
                i=i.prev
        key=key.next
    return head
```

Complejidad: Esta función consta principalmente de un ciclo while dentro de otro, por lo cual, para saber su complejidad se debe analizar el peor y mejor caso de este while doble:

- **Mejor caso:** Suponiendo que head es una lde de mínimo 2 elementos, el mejor caso se da si esta lista ya está ordenada descendentemente según su opinión, o experticia en caso de que hayan opiniones iguales, ya que en dicho caso la función nunca entraría al while interior, por lo cual se ahorraría varias operaciones. Ej: Opiniones [5,4,2,1]

  En este caso, ya que nos saltamos el while interior, key va avanzando de uno en uno, y comparándose con i, por lo cual si n es el tamaño de la lista, se deben realizar n comparaciones de la opinión de key e i, lo cual tiene costo de O(1), entonces, este caso tiene un costo de O(n).

- **Peor caso:** Este caso se da si se tiene una lista de encuestados donde todos tienen la misma opinión, pero experticias totalmente desordenadas, por ejemplo: Opiniones [7,7,7], con experticias [1,4,10].

  Si se tiene esta lista, la función entra al segundo while en cada interacción, por lo cual, key debe comparar su opinión con todos los nodos que están antes que el, debido a que todos estan mal ordenados, y también debe comparar su experticia para luego intercambiar valores entre key e i. Además, debido al primer while, el key debe recorrer toda la lista hasta llegar al final, osea cuando key = key.next da que key.next es None.

  Entonces, si n es el #de encuestados, se deben realizar n-1 + n-2 + n-3 +….+ 1 iteraciones del procedimiento descrito anteriormente. Esto se representa así:

$$\sum_{i=1}^{n-1} n - i = \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} - n = O(n^2)$$

Conclusión: El mejor caso tiene una complejidad casi idéntica al insertion sort de arreglos, mientras que el peor caso es ligeramente más demandante, debido a que se debe realizar una operación extra para comparar la experticia, pero esto no es suficiente para aumentar su cota superior de n cuadrado.

Explicación de las soluciones

1. Ordenar los encuestados descendentemente por su valor de opinión:

2. En cada tema ordenar las preguntas por su promedio del valor de opinión

Para darle solución a este punto, se

3. Ordenar los temas por el promedio de sus preguntas

```python
def Promedio_Pregunta(lista):
    actual = lista
    opinion = 0
    while actual:
        opinion += actual.data.getOpinion()
        actual = actual.next
    return round((opinion / List_Size(lista)), 2)

def Promedio_Tema(lista, M):
    pregunta_actual = lista
    promedio = 0
    while pregunta_actual:
        promedio += Promedio_Pregunta(pregunta_actual.data.getEncuestados())
        pregunta_actual = pregunta_actual.next
    return round(promedio / M, 2)

def Ordenar_Tema_Por_Promedio(lista, M):
    lista = List_Merge_Sort(lista, lambda e: Promedio_Tema(e.getPreguntas(),
M), "descendente")
    resultado = ""
    while lista:
        resultado = resultado + f"{[Promedio_Tema(lista.data.getPreguntas(),
M)]}" + " Tema " + f"{lista.data.getNombre()}:" + "\n"
        lista = lista.next
    print(resultado)
    return resultado
```

Promedio_Pregunta: Esta función recibe una lista doblemente enlazada de encuestados. Recorre la lista sumando las opiniones de cada uno de sus encuestados mediante getOpinion() y luego divide la suma total entre el tamaño de la lista (usando

List_Size). Redondea el resultado a 2 decimales y lo retorna. El tamaño de la lista de encuestados (n) de una pregunta que se le pase a esta función no será siempre la misma, así que para generalizar y dar una aproximación diremos que n = (Nmin + Nmax)/2. Con esto aclarado, diremos que la complejidad es O(n).

Promedio_Tema: Esta función recibe una lista de preguntas asociadas a un tema, y un entero M que representa cuántas preguntas tiene el tema. La función recorre cada nodo de la lista y llama a Promedio_Pregunta para obtener el promedio de opiniones de los encuestados de cada pregunta. Suma esos promedios y al final los divide entre M. Con respecto a su complejidad, si cada pregunta tiene n encuestados, entonces hace M llamadas a Promedio_Pregunta, por lo tanto la complejidad es O(M * n).

Ordenar_Tema_Por_Promedio: Esta función recibe una lista de temas y un entero M (número de preguntas por tema). Primero ordena la lista de temas usando List_Merge_Sort, comparando cada tema con el promedio de sus preguntas (calculado con Promedio_Tema). Luego recorre nuevamente la lista ya ordenada para imprimir el promedio y el nombre de cada tema en orden descendente. El List_Merge_Sort hace $O(K^2 \log K)$ comparaciones (K es la cantidad de temas) y en cada comparación invoca Promedio_Tema, que es O(M * n). La complejidad total es $O(K^2 \log K * M * n)$.

4. Ordenar a todos los encuestados según su experticia:

5. Pregunta con mayor promedio de opiniones:

Para solucionar este problema se creó una función llamada lde_mayor_promedio, la cual llama a una función auxiliar llamada lde_promedio en varias ocasiones.

-lde_promedio: Esta función debe recibir 2 parámetros, uno es la lista de encuestados, y el otro es un entero, el cual debe ser 1 o 2.

Esta función se encarga de calcular el promedio de la opinión o la experticia de la lista de encuestados, dependiendo si el parámetro method es 1 o 2 respectivamente, pero ambos casos hacen lo mismo, solo cambia el método.

Para calcular el promedio, se recorren todos los encuestados de la lista sumando su opinión o experticia, usando un ciclo while.

***Complejidad:** Si n es el número de encuestados, esta función tiene una complejidad de Θ(n), ya que siempre se debe recorrer toda la lista para hallar el promedio.

-lde_mayor_promedio: Esta función recibe como único parámetro una encuesta, a la cual le extrae los temas, y luego mediante un ciclo while extrae las preguntas de todos

los temas de la encuesta y las pone en una sola lista. Si n es el número de preguntas, este ciclo while tendrá complejidad Θ(n).

Ya que se tienen todas las preguntas en una lista, ahora se deben comparar entre sí usando 2 ciclos while, de forma parecida al insertion sort.

Este algoritmo tiene inspiración del insertion sort, aunque tiene la diferencia de que se debe llamar a lde_promedio para hacer las comparaciones, y que key no empieza siendo el segundo elemento de la lista sino el primero, e i va hacia delante en la lista en vez de hacia atrás.

***Complejidad:** Si n es el número de preguntas, debe realizar Θ(n) comparaciones de promedios en cada iteración, ya que no importa si el mayor promedio esta de primero o último, se deben comparar todas las preguntas para saber si no hay una mayor.

También se debe considerar que en cada iteración se debe llamar a lde_promedio, la cual tiene complejidad Θ(n), pero con n siendo el número de encuestados en vez de preguntas. Ya que cada pregunta tiene mínimo un encuestado, el número de encuestados es mayor o igual al de preguntas, por lo cual se tomará n como el número de encuestados incluso para lde_mayor_promedio.

***Mejor caso:** Se da cuando la lista está ordenada ascendentemente según el promedio de opiniones y no hay promedios iguales, ya que así no se entra al segundo while y se evita hacer más comparaciones.

***Peor caso:**

6. Pregunta con menor promedio de opiniones

7. Pregunta con mayor mediana de opiniones

```python
def Mayor_X_Pregunta(lista, K, M, dato):
    if lista is None:
        return 0
    temas = K
    tema_actual = lista
    preguntas_actuales = tema_actual.data.getPreguntas()
    mayor = preguntas_actuales

    while temas > 0:
        preguntas = M
        while preguntas > 0:
```

```
            if dato == "extremismo":
                if Extremismo_Pregunta(mayor.data.getEncuestados()) <
Extremismo_Pregunta(preguntas_actuales.data.getEncuestados()):
                    mayor = preguntas_actuales
            if dato == "mediana":
                if List_Median(List_Merge_Sort(mayor.data.getEncuestados(),
lambda e: e.getOpinion(), "ascendente")).getOpinion() <
List_Median(List_Merge_Sort(preguntas_actuales.data.getEncuestados(), lambda
e: e.getOpinion(), "ascendente")).getOpinion():
                    mayor = preguntas_actuales
            preguntas_actuales = preguntas_actuales.next
            preguntas -= 1

        tema_actual = tema_actual.next
        if tema_actual:
            preguntas_actuales = tema_actual.data.getPreguntas()
        temas -= 1

    if dato == "extremismo":
        resultado = "Pregunta con mayor extremismo: " +
f"{[Extremismo_Pregunta(mayor.data.getEncuestados())]}" + " Pregunta: " +
mayor.data.getNombre() + "\n"
        print(resultado)
        return resultado
    if dato == "mediana":
        resultado = "Pregunta con mayor mediana de opinión: " +
f"{[List_Median(List_Merge_Sort(mayor.data.getEncuestados(), lambda e:
e.getOpinion(), "ascendente")).getOpinion()]}" + " Pregunta: " +
mayor.data.getNombre() + "\n"
        print(resultado)
        return resultado
```

8. Pregunta con menor mediana opiniones

9. Pregunta con el mayor valor de moda de opiniones

10. Pregunta con el menor valor de moda de opiniones

11. Pregunta con mayor valor de extremismo

```
Python
def Extremismo_Pregunta(lista):
    actual = lista
```

```
    extremismo = 0
    while actual:
        if actual.data.getOpinion() == 0 or actual.data.getOpinion() == 10:
            extremismo += 1
        actual = actual.next
    return round((extremismo / List_Size(lista)), 2)
```

12. Pregunta con mayor consenso

Árboles de búsqueda binaria

La definición de un árbol binario de búsqueda (ABB) es que tiene un valor (val), un subárbol izquierdo (left) y un subárbol derecho (right). Si no tiene hijo izquierdo o derecho, los subárboles son None. En código se vería de la siguiente manera:

```python
class abb:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

Operaciones básicas

```python
def List_Insert(lista, llave):
    nuevo_nodo = LDE(llave)
    nuevo_nodo.next = lista
    if lista:
        lista.prev = nuevo_nodo
    return nuevo_nodo

def Arb_Size(arbol):
    if arbol is None:
        return 0
    return 1 + Arb_Size(arbol.left) + Arb_Size(arbol.right)
```

List_Insert: Esta función recibe una llave y una lista, la llave la vuelve una LDE y luego se conectan los punteros. Su complejidad es O(1).

List_Size: Recorre toda la lista que le pasemos hasta que el puntero next esté vacío, en cada iteración con el while se le suma 1 al contador. Su complejidad es O(n).

Algoritmos de ordenamiento

Explicación de las soluciones

1. Ordenar los encuestados descendentemente por su valor de opinión:

2. En cada tema ordenar las preguntas por su promedio del valor de opinión

3. Ordenar los temas por el promedio de sus preguntas

```python
def Suma_Raices(arbol, metodo):
    if arbol is None:
        return 0
    return metodo(arbol.val) + Suma_Raices(arbol.left, metodo) +
Suma_Raices(arbol.right, metodo)

def Promedio_Pregunta(arbol):
    if arbol is None:
        return 0
    total_encuestados = Arb_Size(arbol)
    suma_opiniones = Suma_Raices(arbol, lambda x: x.getOpinion())
    return round(suma_opiniones/total_encuestados, 2)

def Promedio_Preguntas(arbol):
    if arbol is None:
        return 0
    total_preguntas = Arb_Size(arbol)
    suma_preguntas = Suma_Raices(arbol, lambda x:
Promedio_Pregunta(x.getEncuestados()))
    return round(suma_preguntas/total_preguntas, 2)

def Temas_Print(nodo):
    if nodo is None:
        return ""
    resultado = ""
    resultado = resultado + Temas_Print(nodo.right)
    resultado = resultado + "[" +
str(Promedio_Preguntas(nodo.val.getPreguntas())) + "] " + "Tema " +
str(nodo.val.getNombre()) + ":\n"
    resultado = resultado + Temas_Print(nodo.left)
    return resultado
```

```
def Ordenar_Tema_Por_Promedio(arbol):
    actual = arbol
    arbol_temas = abb(actual.val)

    while actual.right:
        actual = actual.right
        arbol_temas = Arb_Insert(arbol_temas, actual.val, lambda e:
Promedio_Preguntas(e.getPreguntas()))

    resultado = Temas_Print(arbol_temas)
    print(resultado)
    return resultado
```

4. Ordenar a todos los encuestados según su experticia

5. Pregunta con mayor promedio de opiniones:

6. Pregunta con menor promedio de opiniones

7. Pregunta con mayor mediana de opiniones

```Python
def Mayor_X_Pregunta(arbol, K, M, dato):
    if arbol is None:
        return 0
    tema_actual = arbol
    temas = K
    preguntas_actuales = tema_actual.val.getPreguntas()
    mayor = preguntas_actuales

    while temas > 0:
        preguntas = M

        while preguntas > 0:
            if dato == "extremismo":
                posible_mayor =
Extremismo_Pregunta(preguntas_actuales.val.getEncuestados()) /
Arb_Size(preguntas_actuales.val.getEncuestados())
                if Extremismo_Pregunta(mayor.val.getEncuestados()) /
Arb_Size(mayor.val.getEncuestados()) < posible_mayor:
                    mayor = preguntas_actuales
            if dato == "mediana":
```

```
                posible_mayor =
Arb_Median(preguntas_actuales.val.getEncuestados(), lambda e:
e.getOpinion())
                if Arb_Median(mayor.val.getEncuestados(), lambda e:
e.getOpinion()).getOpinion() < posible_mayor.getOpinion():
                    mayor = preguntas_actuales

            preguntas_actuales = preguntas_actuales.right
            preguntas -= 1

        tema_actual = tema_actual.right
        if tema_actual is not None:
            preguntas_actuales = tema_actual.val.getPreguntas()
        temas -= 1

    if dato == "extremismo":
        extremismo = round(Extremismo_Pregunta(mayor.val.getEncuestados()) /
Arb_Size(mayor.val.getEncuestados()), 2)
        resultado = "Pregunta con mayor extremismo: [" + str(extremismo) +
"] Pregunta: " + mayor.val.getNombre() + "\n"
        print(resultado)
        return resultado
    if dato == "mediana":
        mediana = Arb_Median(mayor.val.getEncuestados(), lambda e:
e.getOpinion()).getOpinion()
        resultado = "Pregunta con mayor mediana de opinión: [" +
str(mediana) + "] Pregunta: " + mayor.val.getNombre() + "\n"
        print(resultado)
        return resultado
```

8. Pregunta con menor mediana opiniones

9. Pregunta con el mayor valor de moda de opiniones

10. Pregunta con el menor valor de moda de opiniones

11. Pregunta con mayor valor de extremismo

Python

```
def Extremismo_Pregunta(arbol):
    if arbol is None:
        return 0
    if arbol.val.getOpinion() == 0 or arbol.val.getOpinion() == 10:
```

```
        return 1 + Extremismo_Pregunta(arbol.left) +
Extremismo_Pregunta(arbol.right)
    else:
        return Extremismo_Pregunta(arbol.left) +
Extremismo_Pregunta(arbol.right)
```

12. Pregunta con mayor consenso

Análisis de resultados