Documento explicativo - Proyecto parcial 2: Matemáticas Discretas y sus Aplicaciones

1. Introducción

El objetivo de este proyecto es implementar un sistema que simule los clanes de espíritus en Setlândia, utilizando la estructura de datos de conjuntos disjuntos, y calcular la cantidad de formas distintas de organizar la energía total de un clan mediante la función de particiones.

La implementación se centra en procesar dos tipos de operaciones:

- `union(x, y)` : Une los clanes a los que pertenecen los espíritus x e y.
- `partitions(x)` : Consulta cuántas formas distintas hay de distribuir la energía del clan al que pertenece x.

2. Explicación del código

```
def buscar_raiz(padre, x):
    if padre[x] != x:
        padre[x] = buscar_raiz(padre, padre[x])
    return padre[x]
```

La función `buscar_raiz()` es la encargada de encontrar el representante (o raíz) del conjunto al que pertenece el elemento x. Esta función implementa la técnica de "compresión de caminos" para optimizar las búsquedas futuras. Cuando se llama a esta función, si el elemento x no es su propio padre (es decir, no es la raíz del conjunto), se actualiza su padre para que apunte directamente a la raíz, lo que hace que futuras búsquedas sean más eficientes. La compresión de caminos es una optimización crucial en la estructura de datos de conjuntos disjuntos porque reduce significativamente la complejidad de las operaciones, pasando de tener una complejidad casi lineal a una complejidad logarítmica amortizada.

El funcionamiento de la función consiste basciamente en verificar si un espíritu es su propio representante; si lo es, simplemente lo devuelve; en caso contrario, busca recursivamente el representante de su padre y actualiza el padre del espíritu actual para que apunte directamente a ese representante. Esta recursión va "aplanando" el árbol a medida que se realizan búsquedas, lo que mejora el rendimiento de las operaciones que le siguen sobre el mismo conjunto.

```
def union(padre, rango, size, x, y):
    root_x = buscar_raiz(padre, x)
    root_y = buscar_raiz(padre, y)
    if root_x == root_y:
        return
    if rango[root_x] < rango[root_y]:
        padre[root_x] = root_y
        size[root_y] += size[root_x]
    elif rango[root_x] > rango[root_y]:
        padre[root_y] = root_x
        size[root_x] += size[root_y]
    else:
        padre[root_y] = root_x
        size[root_x] += size[root_y]
        rango[root_x] += 1
```

La función `union()` es la encargada de unir los conjuntos a los que pertenecen los elementos x e y. Esta función implementa la técnica de "unión por rango" para mantener árboles balanceados, lo que también optimiza las operaciones. Primero se encuentran las raíces de los conjuntos de x e y utilizando la función `buscar_raiz()`. Si ya están en el mismo conjunto (tienen la misma raíz), no se hace nada y la función retorna. Si no están en el mismo conjunto, se procede a unirlos siguiendo la estrategia de unión por rango.

La unión por rango consiste en hacer que el árbol de menor rango se convierta en hijo del árbol de mayor rango. Esto garantiza que la altura del árbol resultante no aumente innecesariamente, lo que mantiene las operaciones de búsqueda eficientes. Si ambos árboles tienen el mismo rango, se elige arbitrariamente uno de ellos como padre (en este caso, root_x) y se incrementa su rango en 1. El incremento del rango solo ocurre cuando se unen dos árboles del mismo rango, lo que refleja el potencial aumento en la altura del árbol resultante.

Además de actualizar la estructura del árbol, la función también mantiene un seguimiento del tamaño de cada conjunto. Cuando dos conjuntos se unen, el tamaño del conjunto resultante es la suma de los tamaños de los conjuntos originales. Esto es crucial para el problema actual, ya que necesitamos conocer el tamaño del clan para determinar el número de particiones posibles.

```
def calcular_particiones():
    MOD = 1000000007
    tabla = [0] * 51
    tabla[0] = 1
    for i in range(1, 51):
        for j in range(i, 51):
```

```
            tabla[j] = (tabla[j] + tabla[j-i]) % MOD
    return tabla
```

La función `calcular_particiones()` implementa un algoritmo de programación dinámica para calcular la función de particiones para números hasta 50. Esta función calcula cuántas formas distintas hay de sumar enteros positivos para alcanzar un número dado, sin importar el orden de los sumandos.

El algoritmo utiliza una tabla de tamaño 51 (para manejar valores desde 0 hasta 50), inicializando `tabla[0] = 1`, lo que significa que hay una forma de representar el número 0 (no sumando nada). Luego, para cada número i del 1 al 50, se actualiza la tabla para todos los valores j desde i hasta 50, sumando `tabla[j]` (formas actuales de representar j) y `tabla[j-i]` (formas de representar j-i, a las cuales se les puede añadir i para obtener j).

Para cada número i, estamos considerando todas las formas de incluirlo en las particiones de números mayores o iguales a i. Por ejemplo, al considerar i=3, estamos agregando el número 3 a todas las particiones existentes para j-3, creando así nuevas particiones para j.

Para evitar desbordamientos, ya que los números pueden crecer rápidamente, se aplica el módulo 1,000,000,007 a cada resultado como se especifica en el problema. La tabla resultante contiene, para cada índice j, el número de particiones posibles del número j.

```
def main():
    particiones = calcular_particiones()
    lines = stdin.readlines()
    index = 0
    t = int(lines[index].strip())
    index += 1
    for _ in range(t):
        n, m = map(int, lines[index].strip().split())
        index += 1
        padre = [i for i in range(n + 1)]
        rango = [0] * (n + 1)
        size = [1] * (n + 1)
        for _ in range(m):
            operacion = lines[index].strip().split()
            index += 1
            if operacion[0] == "union":
                x, y = int(operacion[1]), int(operacion[2])
                union(padre, rango, size, x, y)
            elif operacion[0] == "partitions":
                x = int(operacion[1])
                root = buscar_raiz(padre, x)
```

```
            set_size = size[root]
            print(particiones[set_size])
```

La función `main()` es la encargada de manejar la entrada y llamar a las funciones correspondientes según las operaciones que se soliciten. La función comienza calculando todas las particiones posibles hasta n=50 utilizando la función `calcular_particiones()`. Este precálculo es importante porque evita tener que recalcular las particiones múltiples veces durante la ejecución del programa, lo que sería ineficiente.

Después del precáclulo, se leen todas las líneas de la entrada mediante `stdin.readlines()`. La primera línea contiene el número de casos de prueba t, y se procesa cada caso de prueba por separado. Para cada caso, se leen n (número de elementos) y m (número de operaciones), y se inicializan las estructuras necesarias para el Union-Find:

- `padre`: Inicializado para que cada elemento sea su propio padre (cada espíritu en su propio clan).
- `rango`: Inicializado en 0 para todos los elementos, ya que inicialmente todos los árboles tienen altura 0.
- `size`: Inicializado en 1 para todos los elementos, ya que inicialmente cada clan tiene un solo espíritu.

Luego, se procesan las m operaciones una por una. Si la operación es "union", se leen los índices x e y y se llama a la función `union()` para unir los clanes correspondientes. Si la operación es "partitions", se lee el índice x, se encuentra la raíz del clan al que pertenece x usando `buscar_raiz()`, se obtiene el tamaño de ese clan de la lista size, y se imprime el número de particiones para ese tamaño usando la tabla precalculada.

Es importante notar que el programa lee toda la entrada de una sola vez y luego procesa cada línea secuencialmente, lo que es una estrategia eficiente para manejar entradas grandes. Además, la estructura de datos de Union-Find con las optimizaciones de compresión de caminos y unión por rango garantiza que las operaciones sean eficientes incluso para conjuntos grandes.