Resuelto por:
Christopher Gómez

Universidad Simón Bolívar
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Tarea 6: Árboles

1. Considere un arreglo `A[1..N]`, representando una permutación de los números de 1 a N .

   Se desea que ejecute N acciones de la forma `multiswap(a, b)`. Esta acción consiste en:

   (a) Intercambiar el valor en la posición `a` con el valor en la posición `b`.

   (b) Invocar `multiswap(a+1, b+1)`.

   (c) El proceso termina cuando `b` se sale del rango del arreglo o `a` alcanza el primer valor de `b` utilizado.

   A continuación se presenta una implementación en pseudo–Python para `multiswap(a,b)`:

```
def multiswap(A, a, b):
    i, j = a, b
    while i < b and j ⩽ N:
        swap(A, i, j)
        i += 1
        j += 1
```

   Si A inicia como la permutación identidad (números del 1 al N , de menor a mayor) y se ejecutan N operaciones `multiswap(ai, bi)` (donde los valores para ai y bi vienen dados en una lista de tuplas), se desea que imprima el arreglo resultado.

   Diseñe un algoritmo que pueda ejecutar esta acción en tiempo <u>promedio</u> $O(N\log(N))$, usando memoria adicional $O(N)$.

   Pista:

   Una estructura de datos que permita *dividir* o *reunir* subarreglos eficientemente con una *alta probabilidad*, puede llevar al camino del bien.

   Analizando el problema, notamos que podemos expresar el resultado de `multiswap(a, b)` según dos casos como:

   - Si $b - a \leqslant N + 1 - b$ (la parada se da porque `a` alcanza el primer valor de `b`):

     `multiswap(a, b)` $= A\big[1..a\big) + A\big[b..2b - a\big) + A\big[a..b\big) + A\big[2b - a..N\big]$. Es decir, se intercambia $A\big[a..b\big)$ con $A\big[b..2b - a\big)$ y se deja el resto igual.

   - En caso contrario (la parada se da porque `b` se sale del rango del arreglo):

     `multiswap(a, b)` $= A\big[1..a\big) + A\big[b..N\big] + A(a + N - b..b) + A\big[a..a + N - b\big]$. Esta vez se intercambia $A\big[a..a + N - b\big]$ con $A\big[b..N\big]$ y se deja el resto igual.

   Donde el operador $+$ representa la concatenación de arreglos.

   De esta forma, el problema consiste en hallar una forma de intercambiar dos subarreglos de rangos disjuntos en tiempo promedio $O(\log(N))$, con lo cual lograríamos el objetivo de hacer N operaciones en tiempo promedio $O(N\log(N))$.

Así, la idea consiste en construir un treap implícito en base a la permutación identidad (que en el caso promedio se encontrará aproximadamente balanceado) y nos permitirá hacer las operaciones de `split` y `merge` en tiempo promedio $O(\log(N))$. Un pseudocódigo para el programa que resuelve el problema es el siguiente:

```
def multi_multiswap(A: Lista[Entero], swaps: Lista[Tupla[Entero, Entero]]):
    t = construir_treap(A)
    for swap in swaps:
        t = multiswap(t, swap.a, swap.b)

    imprimir_inorden(t)

def multiswap(t: Treap, a: Entero, b: Entero) → Treap:
    if t is None:
        return

    n = t.tamaño()

    if b - a ⩽ n - b + 1:
        sub1, r1 = t.dividir(a - 1)
        sub2, r2 = r1.dividir(b - a)
        sub3, sub4 = r2.dividir(b - a)

        return t.mezclar(sub1, sub3, sub2, sub4)

    sub1, r1 = t.dividir(a - 1)
    sub2, r2 = r1.dividir(n - b + 1)
    sub3, sub4 = r2.dividir(2 * b - n - a - 1)


    return t.mezclar(sub1, sub3, sub2, sub4)
```

La construcción de un treap implícito toma memoria adicional $O(N)$, ya que por cada nodo se almacenan una cantidad constante de campos, y tiempo $O(N\log(N))$ en el caso promedio. Luego, cada operación `multiswap` hace una cantidad constante de operaciones que en promedio toman tiempo $O(\log(N))$, por lo que el tiempo total de ejecución es $O(N\log(N))$ en el caso promedio.

Una implementación de este algoritmo en Lua se puede encontrar aquí.

2. Sea $A = (N, C)$ un árbol (notemos que $|C| = |N| - 1$) y un predicado $p : C \to \{\mathit{true}, \mathit{false}\}$. Queremos responder consultas que pueden tener una de dos formas:

   - $\mathit{forall}(x, y)$, para $x, y \in N$, que diga si evaluar p para *todas* las conexiones entre entre los nodos x e y resulta en true.
   - $\mathit{exists}(x, y)$, para $x, y \in N$, que diga si evaluar p para *alguna* de las conexiones entre entre los nodos x e y resulta en true.

   Diseñe un algoritmo que pueda responder Q consultas de cualquiera de estas formas en tiempo $O(|N| + Q \log|N|)$, usando memoria adicional $O(|N|)$.

   Pista:

   Realice un precondicionamiento adecuado en $O(|N|)$, que le permita responder cada consulta en $O(\log|N|)$.

   Podemos resolver el problema aplicando la descomposición Heavy–Light, la idea es la siguiente:

   - Se obtiene una descomposición Heavy–Light del árbol A.
   - Se construye un árbol de segmentos para las aristas c de cada cadena pesada, tomando como valores de las aristas el valor de $p(c)$.
   - Cada árbol de segmentos contendrá un campo para responder consultas de tipo forall y otro para consultas de tipo exists, en los que se acumulan los valores de los hijos con un $\wedge$ y un $\vee$ respectivamente.
   - Por cada consulta se realiza el procedimiento usual: se halla el ancestro común más bajo de x y y, se descompone el camino entre x y el ancestro común más bajo en cadenas pesadas y se responde la consulta con los valores acumulados en los árboles de segmentos correspondientes.

   Veamos que para el precondicionamiento se obtiene una descomposición Heavy–Light en tiempo y memoria $O(|N|)$, y otro con el mismo requerimiento asintótico de tiempo y memoria para hallar el ancestro común más bajo de cualesquiera dos nodos. Luego, por cada consulta se halla el ancestro común más bajo en tiempo $O(\log|N|)$ y se responde la consulta en tiempo $O(\log|N|)$, por lo que el tiempo total de ejecución para responder Q consultas es $O(|N| + Q \log|N|)$.

   Una implementación parcial de este algoritmo en Swift se puede encontrar aquí.

3. Considere un arreglo `A[1..N]`, representando una permutación de los números de 1 a N.

   Se desea que responda Q consultas de la forma `seleccion(i, j, k)`. Esta consulta pide calcular el k–ésimo elemento del subarreglo `A[i..j]`, si ese subarreglo estuviera ordenado.

   Tomemos, por ejemplo, `A = [2,6,3,1,8,4,7,9,5]`:

   - Al hacer `consulta(2,5,3)`, se refiere al subarreglo comprendido entre las posiciones 2 y 5; es decir: `[6,3,1,4]`. Si ordenáramos este sub–arreglo, el resultado sería `[1,3,4,6]` y el tercero (3–ésimo elemento) sería 4.
   - Al hacer `consulta(3,7,1)`, se refiere al subarreglo comprendido entre las posiciones 3 y 7; es decir: `[3,1,8,4,7]`. Si ordenáramos este sub–arreglo, el resultado sería `[1,3,4,7,8]` y el primero (1–ésimo elemento) sería 1.
   - Al hacer `consulta(1,9,5)`, se refiere al subarreglo comprendido entre las posiciones 1 y 9; es decir: `[2,6,3,1,8,4,7,9,5]`. Si ordenáramos este sub–arreglo, el resultado sería `[1,2,3,4,5,6,7,8,9]` y el quinto (5–ésimo elemento) sería 5.

   Se desea que diseñe un algoritmo que pueda responder todas las consultas usando tiempo $O((N+Q)\log N)$ y memoria $O(N \log N)$.

   Pistas:

   - Consideremos un arreglo de ocurrencias, donde la i–ésima posición representa la ocurrencia del valor `i` (1 si está y 0 si no está). Consideremos el mismo subarreglo del primer ejemplo: `[6,3,1,4]`. Su arreglo de ocurrencias sería `[1, 0, 1, 1, 0, 1, 0, 0, 0]`.
   - En el arreglo de ocurrencias anterior, ¿cuántas veces aparecen los números del 2 al 5? O, en general, ¿cuántas veces apareces los números del i al j? ¿Hay alguna estructura que permita responder este tipo de consultas *eficientemente*?
   - En algún momento hablamos sobre arreglos cumulativos para resolver consultas estilo `suma(i, j)`. Una idea en particular, que usamos ahí, podría ser de utilidad.
   - Cuando sientan que el problema se vuelve muy difícil, sean *persistentes*.
   - El tiempo y el espacio son uno.

   501 Not Implemented.