Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2021 - 2

Pauta Interrogación 2

Pregunta 1

Solución con Segment Tree

Se tiene un segment tree tal que la hoja a_i es exactamente cuantas queries de la forma $\texttt{add}(i)$ se han hecho, y en los otros nodos se guarda la suma de los nodos hijos, naturalmente cada query $\texttt{add}(i)$ corresponde en sumar 1 en la posición i, este tipo de update fue visto en clases con complejidad $O(\log n)$. Para responder $\texttt{query}(a, b, k)$, usaremos la query auxiliar $\texttt{sum}(a, b)$, que es la suma del rango $[a, b]$, i.e. la cantidad de elementos agregados hasta ahora que están en ese rango. Ahora hay dos formas de responder $\texttt{query}(a, b, k)$:

1. Se puede usar binary search para encontrar el menor c tal que $\texttt{sum}(a, c) \geq k$, esto tiene una complejidad $O(\log^2(n))$ ya que en cada iteración del binary search se llama a $\texttt{sum}(a, b)$ que tiene una complejidad $O(\log n)$, y binary search hace $O(\log n)$ iteraciones.

2. Se puede usar la estructura misma del segment tree usando material visto en ayudantía, sea d la respuesta de $\texttt{sum}(1, a-1)$ si $a \neq 0$ y cero en otro caso, redefinimos $k := k + d$, y ahora queremos hacer el siguiente procedimiento recursivo sobre los nodos del segment tree comenzando en la raíz. Si el presente nodo es una hoja, se retorna el índice, si no, sea s la suma del hijo izquierdo, si $k > s$, entonces procedemos recursivamente sobre el hijo derecho con $k := k - s$, si no procedemos recursivamente sobre el hijo izquierdo. Notamos que el proceso anterior toma $O(\log n)$ para la primera query y $O(\log n)$ para el proceso recursivo (recorremos a lo más la altura del árbol nodos en total). Por lo que esto tiene complejidad $O(\log n)$.

3. Se pueden buscar los nodos que contienen al rango (a, b). La cantidad de estos nodos es $O(\log n)$. Luego, se revisa si el valor de cada uno de estos nodos supera el k. Se toma primero el primer nodo, y si su valor v es menor a k, entonces se redefine $k := k - v$ y se pasa a revisar el siguiente nodo que construye el rango (a, b). Si al revisar alguno de los nodos, se tiene $v \leq k$, sabemos que el k-ésimo mínimo valor se encuentra en el rango (i, j) de ese nodo. Aquí entramos a un proceso recursivo para revisar los valores de ese rango: el caso base, si $i = j$, el nodo es una hoja, y el valor de ese nodo es el k-ésimo menor que buscamos. Si $i \neq j$, el nodo no es hoja, entonces se pasa a revisar sus dos hijos. Si el valor del hijo izquierdo es menor o igual a k, se entra al proceso recursivo en ese nodo. En caso contrario, se entra al proceso recursivo del hijo derecho. Vemos que el proceso recursivo se realiza en $O(1)$, y a lo más se puede llamar una cantidad de veces equivalente a la altura del árbol, lo que es $O(\log n)$. Luego la complejidad final queda $O(\log n)$.

Puntaje

Propuesta:

- 0,5 por descripción del segment tree.

- 0,2 por query `add` correcta y 0,3 por la justificación de la complejidad.

    1. 0,5 por usar binary search y una query auxiliar `sum`, y 0,5 por justificar la complejidad correctamente.
    2. 0,2 por reducir el problema a encontrar el 'k-ésimo' elemento del segment tree, 0,2 por dar una justificación correcta de que la complejidad de la reducción, 0,2 por la explicación de como encontrar el k-ésimo elemento del segment tree, 0,2 por la justificación de la complejidad y 0,2 por la justificación de la complejidad de ambas partes juntas.
    3. $0,2$ por buscar nodos que responden a rango (a, b), $0,3$ por explicar proceso para buscar el k-ésimo elemento del rango, $0,5$ por justificar complejidad

Solución con Árbol Binario de Búsqueda

Sea T su árbol binario de búsqueda favorito balanceado tal que su profundidad sea $O(\log n)$ con n la cantidad de nodos en el árbol y que un nodo que no es hoja siempre tiene un hijo izquierdo.
Cada nodo se va a guardar la siguiente información: el valor asignado al nodo (x), la cantidad de veces que se ha guardado el valor asignado al nodo ($\#x$), y la suma de todos los $\#x$ del subárbol que tiene al presente nodo como raíz.
Al instanciar el árbol, se hace de tal forma que exista un nodo para cada valor entero en $[1, n]$, cada nodo inicialmente con $\#x = 0$. Ahora, dado ese árbol se ve que la query `add`(x) corresponde a encontrar el nodo que tiene el valor x, aumentar en 1 el atributo $\#x$ y por cada nodo en el camino al nodo con valor x se aumenta s en 1 (incluyendo el con nodo con valor x).
Con lo anterior para responder la query `query`(a, b, k) se necesita calcular la cantidad de elementos menores estrictos a a, y esto se logra llamando un proceso recursivo con 4 casos distintos. Estando en un nodo v:

1. Si v es hoja, retorna $v_{\#x}$
2. Sea u el hijo izquierdo de v. Si $v_x = a - 1$, entonces se retorna $u_s + v_{\#x}$
3. Si $v_x > a - 1$, se retorna el valor llamar el proceso recursivo en u.
4. Si $v_x < a - 1$, entonces se retorna el valor del proceso recursivo en el hijo derecho de v, más $u_s + v_{\#x}$.

Si empezamos llamando el proceso recursivo en la raíz del árbol, obtendremos la cantidad de elementos menores estrictos a a. Este proceso pasa por a lo más la profundidad del árbol, por lo que toma $O(\log n)$.
Ahora sea d el resultado del proceso anterior, se redefine $k := k + d$ y se hace un proceso recursivo similar al de la segunda solución con segment tree.

Puntaje

Propuesta:

- 0.5 por descripción del árbol
- 0.3 por query `add` y 0.2 por justificación de su complejidad.
- 0.2 por reducción a k-ésimo elemento, 0,2 por justificación de la complejidad de la reducción, 0,2 por explicación correcta de encontrar k-ésimo elemento, 0,2 por justificación de complejidad de encontrar k-ésimo elemento y 0,2 por justificación de complejidad final.

d

Pregunta 2

El problema consiste en diseñar una solucion con backtracking para el problema dado. Es por eso que se requieren ciertas condiciones como minimo

- Una descripcion de la funcion *is_solvable*, que revise si es nescesario realizar backtrack
- Una descripcion de la funcion que ejecuta el backtrack de una forma tradicional.

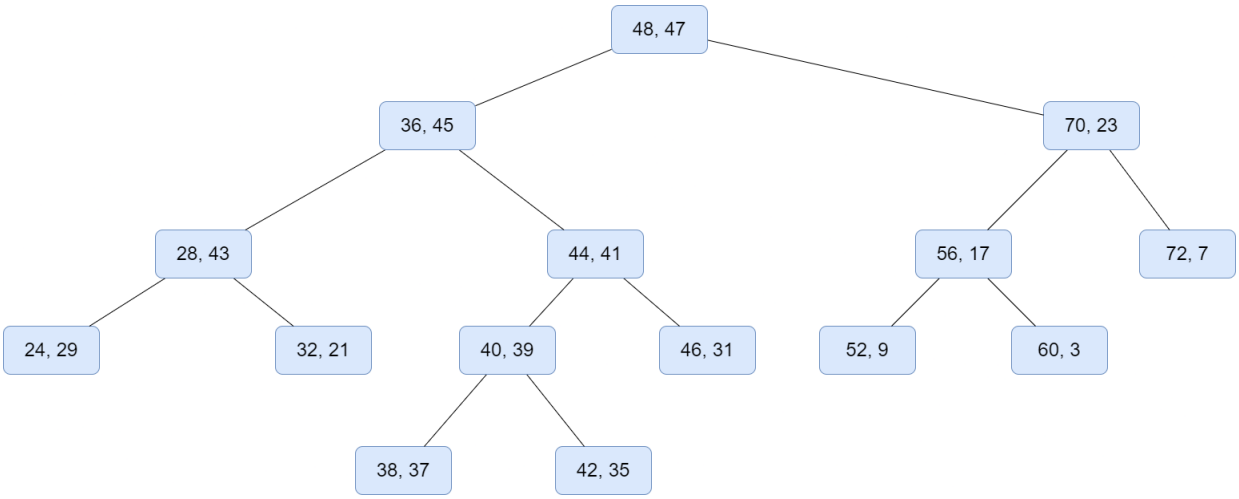
El dominio del problema son los valores de la lista `D`, y se van realizando las asignaciones en la recta mediante backtracking, por lo tanto siempre que se asigne una nueva variable se ha de revisar si las nuevas diferencias existen dentro de la lista de distancias. Es importante agregar ademas, que cada vez que la distancia de un par de puntos coincide con algun valor de la lista, este valor ha de ser extraido para no ser utilizado nuevamente.
La distribucion de codigo queda de la siguiente forma

- 0.1 Pts por calcular o mencionar el largo maximo de X
- 0.2 Pts Por mencionar la condicion de termino (Ultimo elemento de X es el ultimo elemento de D, esto debido a que se entregan los tiempos en orden creciente)
- 0.1 Pts Mencionar que el dominio de X son los elementos de D
- 0.2 Pts Por implementar *correctamente* la funcion *is_solvable*
- 1 Pts utilizar correctamente una solucion con backtracking (en caso de utilizar backtracking de forma incorrecta asignar 0.5)
- 0.2 Pts por mencionar que no se deben repetir valores de D
- 0.2 Pts por mencionar que no deben repetirse los valores de Tiempos

Pregunta 3

A)

Solución :



Puntaje :

- 0.75 puntos por llegar al resultado correcto del árbol
- 0.25 puntos por mostrar como llego a este
- descuento proporcional a la cantidad de errores en el árbol, mas de 2 errores no da puntaje.

B)

Solución :

Las estructuras de datos esperadas a utilizar son una tabla de hash y un min heap
Se construye una tabla de hash con tuplas de la forma (Nº Alumno, votos totales) con 0 votos iniciales, a medida que leemos un voto de un alumno le agregamos al valor en la casilla correspondiente $O(1)$
Para obtener a los k votantes mas votados utilizaremos un min - heap de tamaño k, inicialmente vació, a medida que se votan a los alumnos el heap se va rellenando hasta ser de tamaño k, como sabemos en la raíz se encuentra el numero, dentro de los k mas votados, menos votado. por lo que una vez lleno simplemente se compara el valor de su tupla con la tupla del alumno recién votado $O(1)$, ahí vemos si es necesaria realizar una operación o dejamos el heap como esta.

- 0.25 puntos por decidir utilizar una tabla de hash y un min-heap
- 0.35 puntos por explicar como utilizar la tabla de hash
- 0.4 puntos por explicar como utilizar el min-heap
- descuento proporcinal en caso de no utilizar alguno de estos algoritmos, si el detalle de la implementacion es bueno dar puntaje parcial.

Pregunta 4

a) 0 ptos.

b) 0 ptos.

c) 0.5 ptos. (0.3 ptos si se describió uno de los dos casos posibles)

d) 0.8 ptos.

e) 0.7 ptos.