Repaso C1

Profesores: Pablo Barceló, Gonzalo Navarro
Auxiliares: Matilde Rivas, Bernardo Subercaseaux

P1 - Corrección.-

Nos enfrentamos claramente a un problema de reducción, su primera tarea es identificar eso, y cuál es el problema al que conviene reducir este.
El único problema de base del cuál dominamos una cota en el modelo de comparaciones es el de ordenar N elementos, que requiere $\Omega(N \log N)$ comparaciones. Además, ya ha sido revisada en auxiliar la reducción desde el problema de detectar duplicados. ¿Podemos construir rápidamente un input a partir de una lista, tal que su árbol cobertor mínimo nos de información sobre la presencia de elementos duplicados en la lista? Notemos que construir un input implica en este caso construir N puntos (x_i, y_i). En caso de tener duplicados en la lista, nos interesa asignarles el mismo punto en el espacio, ya que así necesariamente tendremos una arista de largo 0 en el árbol cobertor mínimo. (Debemos recordar que todo árbol cobertor mínimo contiene la arista de peso mínimo del grafo, es evidente si recordamos el algoritmo de Kruskal).

Una respuesta correcta es por lo tanto: Supongamos que podemos resolver el problema del árbol cobertor mínimo euclideano haciendo $o(N \log N)$ comparaciones. Luego consideremos una lista desordenada $L = l_1, l_2, ..., l_N$. Si construimos N puntos de la forma $(h(l_i), 0)$ (h es una función de hash, notemos que la lista no tiene por qué ser necesariamente de enteros o números) notar que esto no cuesta ninguna comparación), y luego utilizamos nuestro algoritmo para computar el árbol cobertor mínimo. Este poseerá una arista de peso 0 sí, y solo sí, habían dos elementos l_i y l_j tales que $l_i = l_j, i \neq j$. Por lo tanto sin hacer más comparaciones, resolvimos el problema de existencia de duplicados. Sabemos que ordenar se reduce a detectar duplicados (recordatorio: requiere al menos comparar los elementos que serían adyacentes bajo un ordenamiento (de lo contrario el adversario elige si hay duplicados o no) y a partir de eso se puede construir un grafo de comparaciones cuyo orden topológico es el orden de los elementos), y que ordenar requiere $\Omega(N \log N)$ comparaciones. Contradicción.

P2 - Corrección.-

El tipo de datos (es decir, el hecho de que sean números reales) es irrelevante en este momento. Debemos darnos cuenta que dado el contexto de la pregunta, la única opción razonable es modificar ligeramente alguna estructura que ya conocemos (nunca se nos va a pedir inventar una estructura de datos inteligente y novedosa en el tiempo de un control). ¿Qué estructura conocemos con inserción y borrado en $O(\log_B N)$ I/Os en memoria secundaria? **B-Trees**.

La inserción y borrado es directamente parte de la materia que debiesen dominar. Está en el

apunte y por lo tanto no será especificada aquí.

Nos enfocaremos en la query `range(x, y)`. Notemos inmediatamente que la respuesta puede ser $O(N)$, por lo tanto cualquier algoritmo que llegue hasta todas las hojas de la respuesta tiene ese cómo cota inferior. Por lo tanto, necesitamos que nuestro algoritmo se detenga antes, ojalá, de hecho, que se detenga lo antes posible. Necesariamente el algoritmo realizará una exploración desde arriba hacia abajo en el árbol, pero necesitamos detenernos tan pronto cómo podamos. ¿En qué caso debemos frenar la exploración? Para aligerar notación, definamos para un nodo d del árbol los valores d_a y d_b cómo la clave más pequeña y la clave más grande que almacena el nodo d. Notemos que d_a y d_b pueden ser construidas al insertar claves sin lecturas adicionales; basta con que al insertar una clave, mientras bajamos por el árbol buscando su nodo correspondiente, actualicemos los valores d_a y d_b de cada nodo por el que pasamos.

1. En el caso en que el rango $[x, y]$ esté completamente fuera del nodo en que se encuentra la exploración (llamaremos d a ese nodo). En ese caso no tiene sentido seguir hacia abajo y retornamos 0.

2. En el caso en que el rango de llaves que abarca el nodo d está completamente contenido en $[x, y]$, en cuyo caso el nodo retona el tamaño de su subárbol (esto se puede computar sin costo adicional sumando 1 al hacer el recorrido en inserción).

Esto lleva al siguiente algoritmo:

```
1   function range(x, y, d) {
2   if(y < d.a or x > d.b) { // intervalos  disjuntos
3       return 0;
4   }
5   if(x <= d.a and d.b <= y) { // completamente contenido
6       return d.size; // tamaño del subárbol
7   }
8
9   answer = 0;
10  for(node s: d.children) {
11      answer += range(x, y, s);
12  }
13  return answer;
14 }
```

Para analizar su complejidad, podemos considerar solamente la cantidad de nodos en que se ejecutan llamados recursivos. Ya que podemos asumir que por cada nodo guardamos junto a sus los punteros a sus hijos, los valores a y b de cada uno de esos hijos, y por lo tanto los nodos que no hacen llamdos recursivos no hacen ningún acceso al disco.

Para contar cuántos nodos hay en total que ejecutan llamados recursivos, podemos contarlos por nivel y luego multiplicar por la cantidad de niveles del árbol ($O(\log_B N)$ niveles). Vamos a mostrar que en cada nivel hay a lo más 2 nodos que ejecutan llamados recursivos. Supongamos que se ejecutan $K > 2$ llamados recursivos en un nivel. Eso significa que se ejecutan llamados recursivos en nodos $l_1, l_2, ...l_K$. Eso contiene una contradicción, ya que si l_1 tiene intersección no vacía con $[x, y]$ y l_K también, l_2 cubre un rango completamente contenido en $[x, y]$ y por lo tanto no ejecuta

llamado recursivo. Tenemos entonces una solución en $O(\log_B N)$.

P3 - Corrección.- Sabemos que $O(\log_B N)$ es óptimo para buscar un solo elemento x en un B-Tree, esto es contenido del curso. La idea es que podemos buscar un elemento x en nuestro B-Tree aumentado haciendo `range(x, x)` y por tanto si pudiésemos hacer `range(x,y)` de forma más eficiente, podríamos hacer el `search` normal de forma más eficiente, cosa que no puede hacerse. Notemos que en general `search(x)` retorna también un puntero a los datos de la clave x, y no solo si está presente o no. Sin embargo, si entendemos bien la prueba de optimalidad de B-Tree, notamos que la prueba muestra también directamente que $O(\log_B N)$ es cota inferior para reportar si un elemento existe o no.