Tarea 6

Saul Ivan Rivas Vega

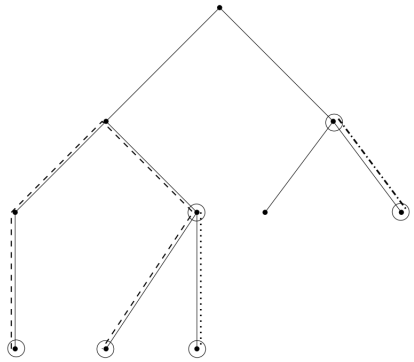
Diseño y análisis de algoritmos

Equipo Completo:
Yadira Fleitas Toranzo
Diego de Jesús Isla Lopez
Saul Ivan Rivas Vega

27 de octubre de 2019

1. Ejercicio 1.

Sea T un árbol binario con raíz y S un conjunto con un número par de vértices de T. Diseña un algoritmo de tipo $divide - y - vencerás$ de tiempo $O(n)$ que empareje los nodos en S de forma tal que en T existan caminos disjuntos en aristas entre cada par.
La figura siguiente muestra un ejemplo del problema en el S tiene los seis vértices encerrados en círculos y los emparejamientos se denotan con tres tipos distintos de lineas punteadas.



1.1. Algoritmo

Data: Árbol T, donde los nodos tienen adicionalmente un valor True o False si es que pertenecen al conjunto S.
Result: Conjunto R con las parejas.

```
/* Inicializar el conjunto R.                                    */
1 R := {};
/* Llamar la función obtenerParejas' con la raíz del árbol y el conjunto
   respuesta.                                                    */
2 obtenerParejas'(raíz, R);
3 return R;
```

Algorithm 1: Algoritmo obtenerParejas.

Data: Nodo u y el conjunto actual de parejas R.
Result: Un nodo que no fue emparejado en el subárbol de u, o un valor vacío si no existe dicho nodo.

```
/* Inicializar los valores de los nodos sin emparejar encontrados en el subárbol
   del hijo izquierdo y del hijo derecho, originalmente vacíos                  */
1  nodo_libre_izq := ∅;
2  nodo_libre_der := ∅;
/* Realizamos llamadas recursivas en el subárbol del hijo izquierdo y derecho si
   es que existen                                                               */
3  if u tiene hijo izquierdo then
4  |  nodo_libre_izq:=obtenerParejas'(hijo izquierdo de u, R)
5  end
6  if u tiene hijo derecho then
7  |  nodo_libre_der:=obtenerParejas'(hijo derecho de u, R)
8  end
/* Si no hay nodos por emparejar en los subárboles checamos si u tiene que ser
   emparejado, si es el caso lo devolvemos como nodo por emparejar en su
   subárbol, en caso contrario devolvemos un valor vacío                        */
9  if nodo_libre_izq y nodo_libre_der son ∅ then
10 |  if u esta en S then
11 |  |  return u;
12 |  else
13 |  |  return ∅;
14 |  end
15 end
/* En este punto hay al menos un nodo por emparejar en alguno de los subárboles
   izquierdo o derecho.                                                         */
/* Checamos si u tiene que emparejarse.                                         */
16 if u esta en S then
   |  /* Checamos si hubo un nodo libre en el subárbol izquierdo y si es el caso,
   |     emparejamos u con ese nodo y devolvemos el valor de nodo_libre_der que puede
   |     ser vacío o un nodo libre.                                            */
17 |  if nodo_libre_izq no es igual a ∅ then
18 |  |  agregamos la pareja (u, nodo_libre_izq) a R;
19 |  |  return nodo_libre_der;
20 |  else
   |  |  /* En caso contrario, emparejamos u con el nodo libre en el subárbol
   |  |     derecho y devolvemos un valor vacío.                               */
21 |  |  agregamos la pareja (u, nodo_libre_der) a R;
22 |  |  return ∅;
23 |  end
24 end
/* Checamos si existen ambos nodos de los subárboles para emparejarlos o en su
   defecto retornar el nodo que no este vacío.                                  */
25 if no son vacíos nodo_libre_izq y nodo_libre_der then
26 |  agregamos la pareja (nodo_libre_izq, nodo_libre_der) a R;
27 |  return ∅
28 end
29 return el nodo que no este vacío entre nodo_libre_izq y nodo_libre_der;
```

Algorithm 2: Algoritmo obtenerParejas'.

1.2. Demostración complejidad $O(n)$

El algoritmo 1 realmente no nos dice mucho, solo inicializa el conjunto respuesta y lo retorna al terminar $obtenerParejas'$ el cual es el segundo algoritmo. Por esta razón realizaremos el análisis en el segundo algoritmo. Cuando el algoritmo es invocado en un nodo hoja, no se realizan mas llamadas recursivas.
Esto es, el algoritmo seguirá bajando en en árbol hasta llegar a una hoja, pues antes de realizar una llamada recursiva revisa si tiene un hijo izquierdo o derecho, como se observa en las líneas 3 y 6, como una hoja no tiene hijos pues se detendrá ahí.

Observación 1 El algoritmo es invocado en un nodo una única vez. Es decir, el algoritmo se llama sobre los hijos del nodo pero en ningún momento se realiza una llamada al padre o a un nodo que no este directamente conectado al nodo actual u. Así pues cuando el algoritmo pasa por el nodo u, no volveremos a procesar a u.

Observación 2 Las operaciones realizadas al ser invocado en una hoja, o al regresar de las llamadas recursivas, son de tiempo constante. Es decir que a partir de la línea 9 hasta el final de las operaciones, solo se realizan comparaciones y asignaciones que podemos considerar de tiempo constante.
Por la observación 1 podemos decir que como existen n nodos, se realizarán n veces las operaciones descritas en la Observación 2 y al ser de tiempo constante decimos que realizamos n veces operaciones de tiempo constante.
Finalmente decimos que el algoritmo es de complejidad $O(n)$.

1.3. Explicación

Se pide solo el diseño del algoritmo con la complejidad $O(n)$ en el ejercicio pero se incluye una explicación de porque es correcto.

Correctitud. El algoritmo termina pues por la Observación 1, no se realizarán mas llamadas recursivas al llegar a las hojas, y en estas solo se toma en cuenta el if en la línea 9 si es que devolvemos o no al nodo para emparejar. Siendo el número de llamadas n y en cada llamada se realiza un número constante de operaciones.

Observación 3 En cada invocación se deja a lo más un nodo sin emparejar en el subárbol con raíz u. Esto lo podemos demostrar por inducción.
Invariante: Se deja a lo más un nodo sin emparejar en el subárbol con raíz u al terminar una invocación.
Caso base: Se considera un subárbol de un solo nodo, es decir una hoja. En este caso se devuelve el valor del único nodo si es que tiene que ser emparejado. Se cumple la invariante

pues solo podemos devolver el valor del nodo o un valor vacío.
Hipótesis inductiva: En cada invocación intermedia podemos emparejar los nodos libres que existan en el subárbol, de tal forma que dejemos a lo más uno libre.
Suponer verdadero hasta algún nodo intermedio en el árbol T.
Dicha invocación en el nodo u tiene los siguientes casos:

1. **u no tiene que emparejarse**

   a) La invocación del hijo izquierdo si devolvió un valor no vacío y también la invocación en el hijo derecho. En este caso se emparejan dichos nodos y no quedan nodos libres.

   b) Alguna de las dos invocaciones en los hijos devolvió un valor no vacío y otro sí. En este caso queda libre el nodo que no fue vacío de las invocaciones.

   c) Ninguna de las dos invocaciones en los hijos devolvió un valor no vacío. En este caso no quedan nodos libres.

2. **u si tiene que emparejarse**

   a) La invocación del hijo izquierdo si devolvió un valor no vacío y también la invocación en el hijo derecho. En este caso se emparejan el nodo u con el nodo libre de la invocación en el hijo izquierdo, dejando libre el nodo de la invocación en el hijo derecho.

   b) Alguna de las dos invocaciones en los hijos devolvió un valor no vacío y otro sí. En este caso se empareja la el nodo u con el nodo libre de las invocaciones y no quedan nodos libres.

   c) Ninguna de las dos invocaciones en los hijos devolvió un valor no vacío. En este caso queda libre el nodo u.

Finalmente como en ninguno de los casos se termina con dos o mas nodos libres podemos decir que en cada invocación se deja a lo más un nodo sin emparejar en el subárbol con raíz u.
Probemos ahora que las parejas resultantes no comparten aristas en el camino que las conecta.
Por contradicción.
Entonces significa que hay por lo menos una arista por el cual pasan 2 caminos entre parejas.
Tomemos el subárbol T_j el cuál tiene como raíz al nodo inferior en dicha arista. Tal como se muestra en la figura siguiente.

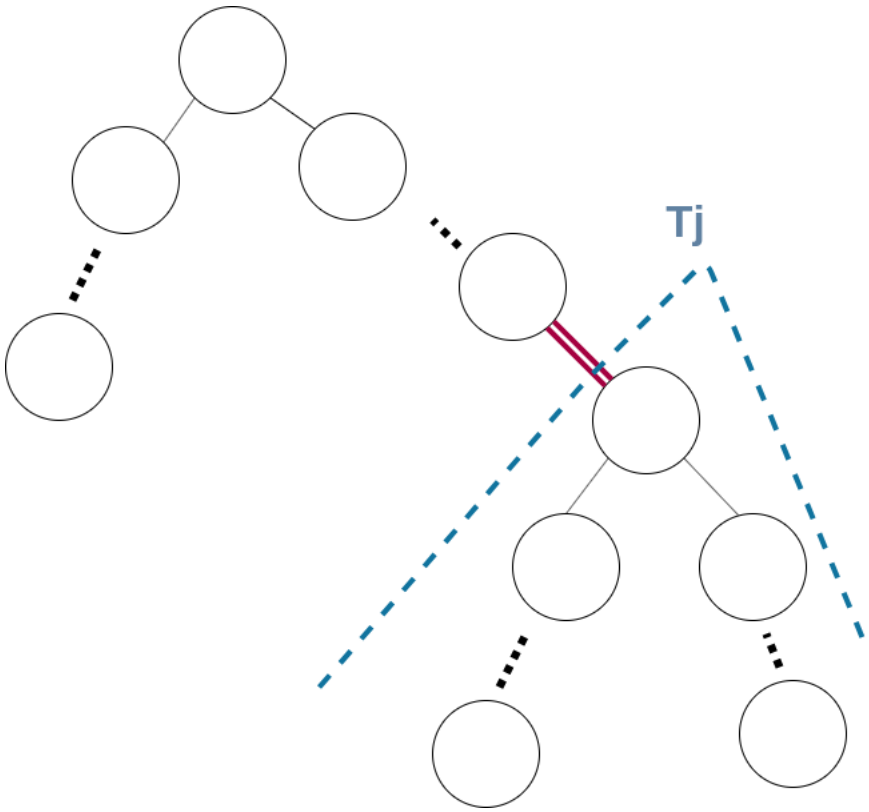


Figura 1: Subárbol T_j (línea punteada), arista con doble camino (en rojo).

Eso quiere decir que en el subárbol T_j existen por lo menos 2 nodos cuyas respectivas parejas se encuentran fuera de T_j. Si tomamos la invocación del algoritmo en la raíz de T_j, para que cumpla que las parejas de esos 2 nodos no están en T_j, al terminar dicha invocación debió dejar esos 2 nodos libres. Sin embargo esto es una contradicción ya que por la Observación 3 al terminar una invocación se deja a lo mas 1 nodo libre en el subárbol.
Finalmente como no puede existir ni una sola arista por el cual pasan dos caminos entre parejas decimos que las parejas resultantes no comparten aristas en el camino que las conecta.

2. Ejercicio 2.

Considera las siguientes recurrencias para las que $T(1) = c$:

a) $T(n) = T(n/2) + O(1)$. Prueba que $T(n) = \Theta(logn)$.

b) $T(n) = T(n/2) + O(n)$. Prueba que $T(n) = O(n)$.

c) $T(n) = T(n/d) + T(n(d-1)/d) + O(n)$, donde d es un entero positivo. Prueba que $T(n) = O(nlogn)$.

2.1. $T(n) = T(n/2) + O(1)$. Prueba que $T(n) = \Theta(logn)$.

Tomemos la siguiente representación de la recurrencia.

$$\begin{aligned} T(n) &= T(n/2) + O(1) \\ T(n) - T(n/2) &= O(1) \\ T(n) - T(n/2) &\in O(1) \end{aligned} \tag{1}$$

Eso quiere decir que existe una constante $C > 0$ y un n_0 tal que para toda $n \geq n_0$ se cumple que:

$$T(n) - T(n/2) \leq C \tag{2}$$

3. Ejercicio 3.

Se tienen dos bases de datos, BD_1 y BD_2, cada una con n valores numéricos distintos, en total $2n$ valores distintos. La única manera de accesar los datos es mediante preguntas, en las cuales se le da un valor $i \in \{1, ..., n\}$ a una de las dos bases de datos, BD_j, y esta responde con el i-ésimo valor más pequeño que contiene BD_j. Diseña un algoritmo que dado un entero $k \in \{1, ..., 2n\}$ de entrada, encuentre el k-ésimo valor más pequeño en las bases de datos, usando $O(logn)$ preguntas. Demuestra la correctez y complejidad de tu algoritmo.

3.1. Algoritmo

Data: Número k.
Result: El k-ésimo elemento de entre las dos bases de datos.
/* Inicializar los índices inicio y final para la busqueda binaria. */
1 *inicio* := 0;
2 *fin* := k;
/* Mientras el inicio sea menor o igual al final. */
3 **while** *inicio* $\leq$ *fin* **do**
  /* Obtenemos el valor en la mitad en el rango de los índices. */
4 mitad:=(inicio + fin) / 2;
  /* El valor 'mitad' funciona para representar cual valor se encuentra en esa posición en la primera base de datos y los restantes en la segunda base de datos. Es decir el k-ésimo elemento es aquel que es mayor a (mitad + (k-mitad)) - 1, elementos. */
5 tomados_bd1:=min(n, mitad);
6 valor_bd1:=PREGUNTAR_BD1(tomados_bd1);
7 tomados_bd2:=min(n, k -tomados_bd1);
8 valor_bd2:=PREGUNTAR_BD2(tomados_bd2);
  /* Ahora si la cantidad de elementos es igual a k (es decir que es mayor a la cantidad necesaria de elementos). */
9 **if** *tomados_bd1 + tomados_bd2 es igual a k* **then**
    /* Veamos si es que el elemento que le sigue en la base de datos 2 es menor al valor de la izquierdo, lo que causaría que sea mayor que mas de k elementos. */
10 valor_check_bd2:=PREGUNTAR_BD1(tomados_bd2 + 1);
11 **if** *valor_check_bd2* > *valor_bd1* **then**
      /* Si en efecto no fue menor entonces ya tenemos k elementos donde el mas grande es el mayor de entre el valor que obtuvimos de las bases de datos. */
12 return max(valor_bd1, valor_bd2);
13 **else**
      /* En este caso tendremos mas de k elementos. Por lo tanto debemos buscar un índice más pequeño */
14 fin:= mitad-1;
15 **end**
16 **else**
    /* En este caso tendremos menos de k elementos. Por lo tanto debemos buscar un índice más grande */
17 inicio:= mitad+1;
18 **end**
19 **end**

Algorithm 3: Algoritmo k-ésimo elemento.

3.2.

Correctitud. El algoritmo termina pues en los condicionales de las líneas 9 y 11 si se cumplen ambas se devuelve el valor encontrado, siendo este el mayor de entre los consultados en la base de datos 1 y 2. Y en caso de que al menos una no se cumpla el espacio de

búsqueda es reducido por la mitad, terminando en $logn$ pasos.
El algoritmo devolverá el k-ésimo valor más pequeño en las bases de datos.
Por inducción.
Invariante: El k-ésimo elemento es mayor a x elementos en la base de datos 1 y a $(k-x-1)$ elementos en la base de datos 2, tal que $inicio \leq x \leq fin$.
Caso base: Se considera la primera pregunta en el algoritmo, el rango es $inicio = 0$ y $fin = k$, y naturalmente el valor x que buscamos se encuentra en dicho rango pues la menor cantidad de elementos al que puede ser mayor es 0 y la máxima los k elementos, es decir los $2n$ elementos.
Hipótesis inductiva: Si el valor x que búscamos no es igual al valor 'mitad' actual, se encuentra ya sea en el rango $(inicio, mitad - 1)$ o en el rango $(mitad + 1, fin)$. Suponer verdadero hasta la i-ésima pregunta.
Tomemos al valor 'mitad' y tomemos el valor de preguntárselo a la primera base de datos. Cualquier valor en la base de datos 1 con un índice mayor a 'mitad' es mayor al valor que esta en 'mitad', por definición. Veamos el valor en exactamente la cantidad de elementos restantes, es decir a $(k - mitad)$ en la base de datos 2. Ahora cualquier valor en la base de datos 2 con un índice mayor a $(k - mitad)$ es mayor al valor que esta en $(k - mitad)$, por definición. Sin embargo no necesariamente son mayores al valor en $mitad$ en la base de datos 1.
Si tomamos el caso donde si es mayor entonces el mayor de los revisados en las bases de datos es la respuesta, puesto que si tenemos k elementos entre $mitad$ y $k - mitad$, y ademas no hay elementos menores a los comprendidos en ellos, el k-ésimo elemento es el mayor de ambos.
En caso contrario el valor significa que si tomamos a $mitad$ en la base de datos 1 tendremos por lo menos $mitad+k-mitad+1$ elementos, lo cual es mayor a k, por lo tanto si reducimos el valor que estamos considerando como $mitad$ reduciremos la cantidad de elementos a los que es mayor. Como se observa en la línea 14.
Análogamente si terminamos con menos elementos si aumentamos $mitad$ aumentaremos la cantidad de elementos a los que es mayor. Como se observa en la línea 17.
Entonces como el rango se adapta dependiendo si se tienen mas o menos elementos de los necesarios, decimos que x se encuentra en $(inicio, fin)$.
Finalmente como el rango se reducirá, de no encontrarlo todavía llegará al punto donde el rango comprende solo a un elemento siendo este el valor x buscado.

Complejidad en tiempo (cantidad de preguntas). Se realizan 3 preguntas en cada iteración del while, y el while reduce por la mitad el rango de búsqueda. El rango originalmente es de tamaño k, por lo tanto se realizan $logk$ iteraciones. Le sigue que son $3logk$ preguntas, y como $k = 2n$, es como si tuviéramos $3(log2+logn)$. Es decir $3(1+logn) = 3+logn$. Esta cantidad de preguntas es de orden $O(logn)$. Sin embargo para un análisis temporal completo tendríamos que multiplicar esa cantidad de preguntas por la función de complejidad de cada pregunta.

4. Ejercicio 4.

En clase revisamos el algoritmo de los códigos de Huffman. En el primer intento de prueba, quisieron probar que el árbol T que generaba el código encontrado satisfacía:

1. T es completo.
2. Si a y b son hojas del árbol tales que profu(a) < profu(b) entonces $f_a \geq f_b$ (f_x es la frecuencia con que aparece x en el texto).
3. Los dos símbolos de frecuencia menor son hermanos en T.

Demuestra que esa estrategia no funciona para probar lo que queríamos. Es decir, ofrece un alfabeto S con frecuencias $f : S \to [0, 1]$ y encuentra un código φ que no sea óptimo y cuyo árbol generado satisfaga las tres propiedades anteriores. Demuestra por qué el código que propones no es óptimo.
Nota: La prueba vista en clase de hecho muestra que la propiedad arriba mencionada se cumple no solo para el árbol T con n hojas, sino para el árbol T' que junta a y b en un solo símbolo.

4.1.

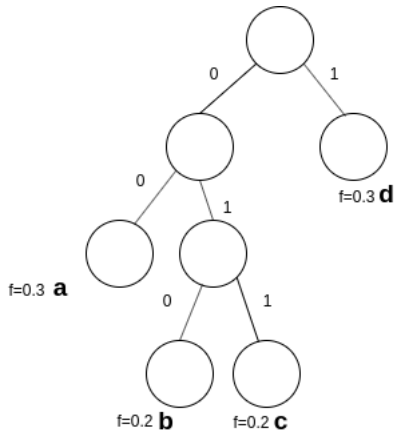


Figura 2: Codificación no óptima que cumple con las propiedades.

Ahora con esta codificación obtengamos las sumas de la longitud por la frecuencia:

$$Suma_a = 0{,}3 \times longitud(00) = 0{,}3 \times 2 = 0{,}6 \tag{3}$$

$$Suma_b = 0{,}2 \times longitud(010) = 0{,}2 \times 3 = 0{,}6 \tag{4}$$

$$Suma_c = 0{,}2 \times longitud(011) = 0{,}2 \times 3 = 0{,}6 \tag{5}$$

$$Suma_d = 0{,}3 \times longitud(1) = 0{,}3 \times 1 = 0{,}3 \tag{6}$$

Si sumamos todo para obtener $AVG(T)$ tenemos:

$$AVG(T) = 0{,}6 + 0{,}6 + 0{,}6 + 0{,}3 = 2{,}1 \tag{7}$$

4.2.

Ahora tomemos una codificación óptima: Ahora con esta codificación obtengamos las

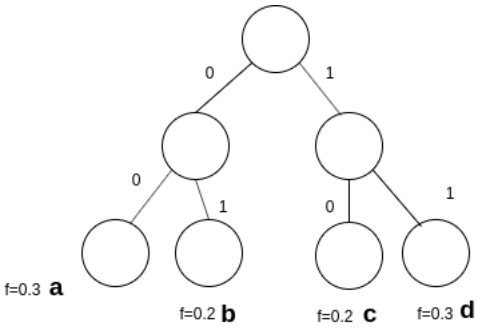


Figura 3: Codificación óptima que cumple con las propiedades.

sumas de la longitud por la frecuencia:

$$Suma_a = 0{,}3 \times longitud(00) = 0{,}3 \times 2 = 0{,}6 \tag{8}$$

$$Suma_b = 0{,}2 \times longitud(01) = 0{,}2 \times 2 = 0{,}4 \tag{9}$$

$$Suma_c = 0{,}2 \times longitud(10) = 0{,}2 \times 2 = 0{,}4 \tag{10}$$

$$Suma_d = 0{,}3 \times longitud(11) = 0{,}3 \times 2 = 0{,}6 \tag{11}$$

Si sumamos todo para obtener $AVG(T)$ tenemos:

$$AVG(T) = 0{,}6 + 0{,}4 + 0{,}4 + 0{,}6 = 2{,}0 \tag{12}$$

Finalmente como en efecto existe una codificación con menor AVG que la primera codificación entonces no es óptima aún cuando cumple con las 3 propiedades.