Facultad de Ciencias, UNAM
Análisis de Algoritmos
Tarea 2

Rubí Rojas Tania Michelle

10 de noviembre de 2020

1. Sea $M[1 \dots n][1 \dots n]$ una matrix de $n \times n$, en el que cada renglón y cada columna están ordenados en orden creciente. Suponga que no hay dos elementos iguales.

   a) Diseña un algoritmo que encuentre la posición de un valor k en M o que determine si que no está. ¿Cuántas comparaciones usa tu algoritmo en el peor caso?

   Solución: Como los elementos de las columnas y las filas están ordenados de forma creciente, entonces podemos buscar el valor de k buscando en submatrices desde uno de los extremos, en particular, desde el extremo superior derecho de la matriz. Usando la información que nos proporciona la matriz, podemos encontrar un camino que nos lleve al valor indicado. Esto se puede hacer con el siguiente algoritmo:

   1) Definimos `fila` $= 1$ y `columna` $= n$.
   2) Mientras `fila` sea menor o igual a n y `columna` sea mayor que 0, hacemos:
      - Si $M[fila][columna] =$ k, entonces regresamos la tupla (i, j).Terminamos.
      - Si $M[fila][columna] < k$, entonces incrementamos en una unidad el valor de `fila`.
      - En otro caso, entonces disminuimos en una unidad el valor de `columna`.
   3) Si llegamos a este punto, es que no hemos encontrado el valor del elemento que nos piden, por lo que simplemente regresamos un -1. Terminamos.

   Este algoritmo funciona porque en cada submatriz de tamaño $fila \times columna$ garantizamos que está el elemento que buscamos. Como iniciamos a comparar desde el elemento que está en la esquina superior derecha de la matriz, entonces si hay un elemento mayor que él, éste debería de estar en alguna de las filas que están abajo de él (ya que las columnas y las filas están ordenadas). Pero si es menor que él, entonces debería de estar en alguna de las columnas anteriores. De acuerdo a esto, vamos actualizando nuestra posición (lo que implica que iremos reduciéndo el tamaño de la submatriz donde debería de estar el elemento que buscamos) y aplicamos el mismo razonamiento para los elementos en cada una de las nuevas posiciones hasta encontrar al elemento deseado.

   Ahora bien, en el peor caso, recorremos toda una fila y toda una columna de la matriz para encontrar el valor que buscamos. Cada uno de estos recorridos nos toma tiempo lineal, por lo que la complejidad total del algoritmo será de $O(n)+O(n) = O(n)$. Usando este mismo razonamiento, como la matriz es cuadrada, entonces a lo más hacemos $2n - 1$ comparaciones para buscar al elemento en el peor caso.

   b) Describe y analiza un algoritmo para resolver el siguiente problema en tiempo lineal. Dados 4 indices i, j, i', j' como entrada, calcule el número de elementos de M que son más pequeños que $M[i][j]$ y más grandes que $M[i'][j']$.

   Solución: Primero, veamos cómo podemos calcular el número de elementos de M que son más pequeños que $M[i][j]$. Los elementos sobre los cuales tenemos certeza de que son menores que $M[i][j]$ están en el subarreglo de tamaño $i \times j$ (los que están por encima de $M[i][j]$ y a la izquierda de éste). A este número le restamos una unidad para no considerar al propio elemento, por lo que tendremos $(i \times j) - 1$ elementos que forzosamente son menores a $M[i][j]$ (esto se debe a que los elementos de

las filas y las columnas están ordenados de forma creciente). Ahora bien, puede que existan más elementos que sean menores que $M[i][j]$ y que se encuentren a su derecha y por encima de la fila i. Para poder determinar el número de elementos menores que $M[i][j]$ en este subarreglo, entonces comparamos a $M[i][j]$ con cada $M[i-1][j+k]$, donde $k \in \{1, 2, \ldots, (n-j)\}$. Entonces, mientras $j+k$ sea menor o igual a n, hacemos

1) Si $M[i-1][j+k] < M[i][j]$, eso significa que todos los elementos anteriores a $M[i-1][j+k]$ dentro de esa columna deberían de ser menores que $M[i][j]$, por transitividad. Así, tenemos $i-1$ elementos adicionales que son menores que $M[i][j]$. Aumentamos en una unidad el valor de k, para así movernos a la siguiente columna.
2) En otro caso, tenemos que subir una fila más arriba para verificar si los elementos anteriores a $M[i-1][j+k]$ dentro de esa columna son menores a $M[i][j]$.
   - La suma de los elementos menores a $M[i][j]$ se realizará de manera análoga al paso anterior, considerando los $i-m$ elementos que fueron menores por transitividad, donde m fue el número de filas hacia arriba que tuvimos que subir para que se cumpliera $M[i-m][j+k] < M[i][j]$.
   - Si encontramos a un elemento $A[i-m][j+k] < M[i][j]$ entonces sumamos $i-m$ elementos a nuestra cuenta y nos movemos a la columna de la derecha, para realizar el procedimiento del paso 1, pero desde la posición actual donde nos encontramos y con los valores $i-m$, $j+k$ correspondientes.
   - Si llegamos al primer elemento de la columna, y tenemos que este elemento tampoco es menor que $M[i][j]$, eso significa que ya no hay elementos más a la derecha que puedan ser menores que $M[i][j]$, así que terminamos de buscar y nos quedamos con la cuenta de los elementos que llevamos.
3) Una vez que terminamos el ciclo del punto anterior, significa que ya no podemos recorrer más la matriz (o que ya no debemos, por el último punto del paso 2). Así que terminamos y regresamos la cuenta de los números menores a $M[i][j]$ que llevamos.

Ahora, para poder encontrar la cantidad de elementos que son mayores que $M[i'][j']$, usaremos un algoritmo análogo al anterior. Los elementos que forzosamente son mayores a $M[i'][j']$ son aquellos que están por debajo de $M[i'][j']$ y a la derecha de éste. Así, hay $((n-i')+1) \times ((n-j')+1)$ elementos dentro de este subarreglo. Nuevamente, disminuimos una unidad para no contar al propio elemento. Entonces, hay $((((n-i)+1) \times ((n-j)+1))) - 1$ elementos que son forzosamente mayores que $M[i'][j']$. Luego, puede que existan más elementos que sean mayores que $M[i'][j']$ a su izquierda y por debajo de la fila i. Para poder determinar el número de elementos que son mayores que $M[i'][j']$ en este subarreglo, entonces comparamos a $M[i'][j']$ con cada $M[i'+1][j'-k]$, donde $k \in \{1, 2, \ldots, (n-j')\}$. Entonces, mientras $j'-k$ sea mayor o igual a 1, hacemos:

1) Si $M[i'+1][j'-k] > M[i'][j']$, eso significa que todos los elementos siguientes a $M[i'+1][j'-k]$ dentro de esa columna deberían de ser mayores que $M[i'][j']$, por transitividad. Así, tenemos $i'+1$ elementos adicionales que son mayores que $M[i'][j']$. Disminuimos en una unidad el valor de k, para así movernos a la columna anterior (una a la izquierda).
2) En otro caso, tenemos que bajar una fila para verificar si los elementos siguientes a $M[i'+1][j'-k]$ dentro de esa columna son mayores a $M[i'][']j]$.
   - La suma de los elementos mayores a $M[i'][j']$ se realizará de manera análoga al paso anterior, considerando los $i+m$ elementos que fueron mayores por transitividad, donde m es el número de filas hacia abajo que tuvimos que recorrer para que se cumpliera $M[i'+m][j'-k] < M[i'][j']$.
   - Si encontramos a un elemento $A[i'+m][j'-k] < M[i'][j']$ entonces sumamos $i+m$ elementos a nuestra cuenta y nos movemos a la columna de la izquierda para realizar el procedimiento del paso 1, pero desde la posición actual donde nos encontramos y con los valores $i+m$, $j-k$ correspondientes.
   - Si llegamos al último elemento de la columna, y tenemos que este elemento tampoco es mayor que $M[i'][j']$, eso significa que ya no hay elementos más a la izquierda que puedan ser mayores

que $M[i'][j']$, así que terminamos de buscar y nos quedamos con la cuenta de los elementos que llevamos.

3) Una vez que terminamos el ciclo del punto anterior, significa que ya no podemos recorrer más la matriz (o que ya no debemos, por el último punto del paso 2). Así que terminamos y regresamos la cuenta de los números menores a $M[i'][j']$ que llevamos.

Este algoritmo funciona porque contemplamos todos las *zonas* o subarreglos cuyos elementos podrían ser menores (mayores) que $M[i][j]$.

Ahora bien, para encontrar la cantidad de elementos que son menores a $M[i][j]$ sólo recorremos linealmente a la matriz, y en el peor de los casos tenemos que recorrer toda una columna y toda una fila; por lo que la complejidad para obtener esta cantidad de elementos será $O(n)$. De manera análoga, para encontrar la cantidad de elementos que son mayores a $M[i'][j']$, el algoritmo toma tiempo lineal. Así, la complejidad total del algoritmo nos tomará tiempo $O(n)$.

2. **Permutaciones de Josephus**: Supongamos que n personas están sentadas alrededor de una mesa circular con n sillas, y que tenemos un entero positivo $m \leq n$. Comenzando con la persona con etiqueta 1, (moviéndonos siempre el la dirección de las manecillas del reloj) comenzamos a remover los ocupantes de las sillas como sigue: Primero eliminamos la persona con etiqueta m. Recursivamente, eliminamos al m-ésimo elemento de los elementos restantes. Este proceso continua hasta que las n personas han sido eliminadas. El orden en que las personas han sido eliminadas, se le conoce como la (n, m)-permutación de Josephus. Por ejemplo si $n = 7$ y $m = 3$, la $(7, 3)$-permutación de Josephus es: $\{3, 6, 2, 7, 5, 1, 4\}$.

   a) Supongamos que m es constante. De un algoritmo lineal para generar la (n, m)-permutación de Josephus.

   Solución: Usaremos una lista circular para este caso. Las listas circulares tienen las mismas propiedades que las listas ligadas, sólo se añade que la cola y la cabeza están conectadas.

   Como tenemos n personas alrededor de la mesa, entonces creamos una lista circular `listaC` de longitud n, cuyos elementos son los enteros $1, 2, 3, \ldots n$. Pero también necesitamos una lista vacía `listaP` para ir agregando la (n, m)-permutación de Josephus, así que también la creamos. Ahora lo que tenemos que hacer es encontrar al $m-$ésimo elemento de `listaC` desde nuestra posición actual (esto lo hacemos moviéndonos $m - 1$ luagres desde donde estamos, e iniciamos en la cabeza), lo agregamos a `listaP`, lo eliminamos de `listaC` y repetimos el proceso hasta que eliminemos a todos los elementos de la lista. Esto se puede ver con el siguiente algoritmo:

   1) Nos posicionamos en la cabeza de `listaC`.
   2) Si la longitud de `listaC` es igual a 1, entonces simplemente agregamos el elemento en la cabeza de `listaC` a `listaP`.
   3) Avanzamos $m - 1$ posiciones a la derecha de la lista para encontrar al $m-$ésimo elemento de la lista (esto porque comenzamos a contar a partir del siguiente elemento). Lo agregamos a `listaP`, actualizamos nuestra posición actual en `listaC` a la posición siguiente de donde estamos y eliminamos al elemento anterior (que es el $m-$ésimo). Repetimos este proceso (comenzando por el paso 2) desde nuestra posición actual (que es la posición siguiente del elemento que eliminamos) hasta que la longitud de `listaC` sea igual a 1 (pues ya sólo agregamos a `listaP` al elemento que nos hace falta).
   4) Regresamos `listaP`.

   Este algoritmo funciona porque en cada iteración nos aseguramos de encontrar al $m-$ésimo elemento e ir actualizando nuestra posición correctamente. Como la lista es circular, entonces podemos ir *dando vueltas* en la lista y sólo nos vamos moviéndo $m - 1$ luagres para ir encontrando la posición deseada. Además, nos aseguramos de eliminar al $m-$ésimo elemento e ir actualizando nuestras referencias adecuadamente.

   Finalmente, crear a `listaC` nos toma tiempo lineal (ya que le agregamos n elementos) y crear a `listaP` nos toma tiempo constante (pues es vacía). Como $m \in O(1)$, entonces movernos $m - 1$ posiciones a la derecha nos tomará también tiempo constante. Pero esto lo hacemos $n - 1$ veces, por lo que esto

también nos cuesta tiempo lineal en total. Agregar un elemento a `listaP` nos toma tiempo constante (pues lo estamos agregando al final). Y eliminar al elemento también nos toma tiempo constante (pues siempre eliminamos al elemento anterior, y para hacer esto sólo actualizamos las referencias de los nodos; por lo que nunca recorremos la lista para poder eliminarlo), pero esto lo hacemos también $n-1$ veces, por lo que en total nos toma tiempo lineal. Así, como realizamos puras operaciones lineales a lo largo de este algoritmo, entonces la complejidad total de éste es de $O(n)$.

Un ejemplo para ilustrar el algoritmo sería el siguiente: si $n = 5$ y $m = 4$, entonces

Figura 1: `listaC.cabeza.elemento = 1;` $m - 1 = 3$ `y listaP = []`

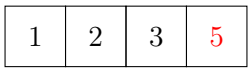

Figura 2: `listaP = [4], posicionActual.elemento = 5`

Figura 3: `listaP = [4, 3], posicionActual.elemento = 5`

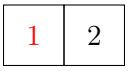

Figura 4: `listaP = [4, 3, 5], posicionActual.elemento = 1`

Figura 5: `listaP = [4, 3, 5, 2], posicionActual.elemento = 1`

Figura 6: `listaC.length = 1 => listaP = [4, 3, 5, 2, 1] Terminamos`

b) Supongamos que m no es constante. Describa un algoritmo con complejidad $O(n \log n)$ para encontrar la (n, m)-permutación de Josephus.

SOLUCIÓN: Para este caso, utilizaremos un árbol binario balanceado cuyas hojas tendrán como elemento a los enteros del conjunto {1, 2, ..., n}, respectivamente; por lo que tendremos n hojas. Además, se debe cumplir que los nodos padres de las hojas tengan como elemento el número de hojas que tienen. El resto de los nodos internos tendrán como elemento el número de hojas que tienen sus dos subárboles (cada nodo obtiene esto sumando los elementos que tienen sus nodos hijos, pues desde la propiedad anterior estamos garantizando que los padres de las hojas nos proporcionan esta información). La construcción de este árbol la haremos de abajo hacia arriba, y será como sigue:

- Construimos n hojas cuyos elementos son $1, 2, \ldots, n$; respectivamente. Además, los marcamos como no visitados.
- Para crear los padres de las hojas:
  - Calculamos `altura` $= \lceil \log(n) \rceil + 1$.
  - Tomamos el primer par de hojas (de izquierda a derecha) y construimos al nodo padre. Éste tendrá como elemento al número 2, ya que tiene dos hojas.
  - Después, tomamos el siguiente par de hojas y realizamos el paso anterior.
  - Si al momento de querer tomar un par de nodos tenemos que sólo podemos tomar uno, es decir, sólo queda o sólo hay una hoja en el nivel, entonces construimos al nodo padre y su elemento será el número 1, pues sólo tiene una hoja. Pasamos al siguiente punto del algoritmo.
  - Si al momento de querer tomar un par de nodos tenemos que ya no hay nodos (es decir, el número de hojas era par), entonces pasamos al siguiente punto del algoritmo.
- Una vez que terminamos de construir los nodos padre de las hojas, disminuimos en uno la variable `altura`.
- Para crear los nodos internos faltantes:
  Mientras `altura` sea mayor que 1, hacemos lo siguiente para los últimos nodos padres que hayamos creado (la primera iteración es sobre los padres de las hojas, después será sobre los padres de los padres de las hojas y así sucesivamente);
  - Tomamos los primeros dos nodos y construimos el nodo padre. El elemento del padre tendrá la suma de los elementos de sus hijos, por lo que
    `padre.elemento = padre.izquierdo.elemento + padre.derecho.elemento`.
  - Después tomamos el siguiente par de nodos y realizamos el paso anterior.
  - Si al momento de querer tomar un par de nodos sólo podemos tomar uno, es decir, sólo queda un nodo en el nivel, entonces construimos un nodo padre, y tendrá como hijos a este último nodo y al último nodo padre que construimos. El elemento de este nuevo nodo padre será la suma de los elementos de sus hijos. Disminuimos en una unidad a la variable `altura` y volvemos a repetir el proceso, pero ahora sobre los nuevos nodos padres que hemos creado.
  - Si al momento de querer tomar un par de nodos tenemos que ya no hay nodos (es decir, el número de nodos era par), entonces disminuimos en una unidad a la variable `altura` y volvemos a repetir el proceso, pero ahora sobre los nuevos nodos padres que hemos creado.
- Cuando llegamos a que `altura` es menor o igual a 1, significa que ya construimos la raíz del árbol. Terminamos.

Ahora necesitamos una forma de recorrer el árbol de tal forma que podamos encontrar cualquiera de las m posiciones que necesitemos buscar. Para buscar nuestra hoja, nos apoyaremos de la información que contiene cada uno de nuestros nodos internos. Cada nodo conoce el número de elementos de n que contiene cada uno de sus subárboles, así que usando esto podemos determinar, estando desde la raíz del árbol, hacia cuál de los subárboles debemos movernos. Este proceso lo hacemos recursivamente (desde la raíz de los subárboles) hasta que encontramos la hoja que necesitamos.

Esto se puede ver con el siguiente algoritmo:

1) Nos posicionamos en la raíz del árbol y hacemos `nodo = raiz`.
2) Mientras el hijo izquierdo del `nodo` no sea una hoja, hacemos:
   - Si m es menor o igual al elemento del hijo izquierdo del `nodo`, entonces nos movemos al subárbol izquierdo de éste, es decir, `nodo = nodo.hijoIzquierdo`.
   - En otro caso, actualizamos el valor de i como $m = m-$ `nodo.hijoIzquierdo.elemento`, y nos movemos al subárbol derecho; es decir, `nodo = nodo.hijoDerecho`.
3) Para este punto, los hijos del `nodo` serán ambos hojas (si es que tiene dos). Así que tenemos dos opciones:
   - Si m es igual a 1 pero el hijo izquierdo ya ha sido visitado, entonces nos movemos a la hoja derecha; es decir, `nodo = nodo.hijoDerecho`.

- Si m es igual a 1, entonces nos movemos a la hoja izquierda; es decir, `nodo = nodo.hijoIzquierdo`. Marcamos como visitada a la hoja. Terminamos.
- En otro caso, nos movemos a la hoja derecha; es decir, `nodo = nodo.hijoDerecho`. Marcamos como visitada a la hoja. Terminamos.

Una vez que tenemos una forma de buscar en el árbol, ya podemos comenzar a buscar la $(n - m)$ permutación de Josephus. Iniciamos en la primer posición, que sería el número 1. Le sumamos $(m-1)$ (para poder obtener la $m-$ ésima posición) y lo buscamos en el árbol. Una vez que lo encontramos, guardamos el valor de su elemento en una lista y lo eliminamos. Actualizamos nuestras referencias (la próxima posición inicial es la posición siguiente del elemento que eliminamos) y repetimos este proceso hasta que hayamos eliminado a todos los elementos en el árbol.

El algoritmo entero sería como:

1) Construimos el árbol con las n hojas y creamos una lista vacía `listaP`.
2) Definimos `posicion = 1`, un contador $i = 1$ y `len = n`.
3) Mientras i sea menor o igual que `len`:
   - Calculamos `posicion = posicion + (m-1)`
   - Si n es menor que `posicion` entonces actualizamos el valor de la posición como `posicion = ((posicion-1)% n) + 1`
   - Buscamos a `posicion` en el árbol. Una vez que estemos en la hoja correspondiente, guardamos el valor de la hoja en `listaP` y *eliminamos* al elemento: siguiendo el mismo camino que recorrió nuestro algoritmo de búsqueda, vamos a disminuir en una unidad el valor de todos los nodos visitados durante la búsqueda.
   - Disminuimos en una unidad a n e incrementamos en una unidad al contador i.
4) Regresamos `listaP`. Terminamos.

El algoritmo funciona por cómo está construido el árbol y porque gracias a las operaciones que realizamos, podemos obtener correctamente al $m-$ ésimo elemento y mantener actualizadas nuestras referencias. El árbol está construido de tal forma que los nodos internos nos proporcionen información valiosa para hacer la búsqueda de elementos hojas en el árbol (cada nodo sabe cuántas hojas contienen sus dos subárboles). La búsqueda de elementos en el árbol funciona correctamente gracias a que en cada iteración garantizamos que el subárbol en el que nos posicionamos contiene al $m-$ésimo elemento (como los nodos contienen el número de hojas que tiene cada uno de sus subárboles, entonces así podemos decidir a cuál subárbol movernos, y cada vez que nos movemos al subárbol derecho, actualizamos el valor de la posición que buscamos). Al momento de eliminar, basta con que borremos todas las referencias que indican que en esa hoja hay un elemento. Así, al momento de buscar nuevamente otro elemento en el árbol, gracias a cómo lo hace, podemos ignorar los subárboles donde no hay elementos y además como actualizamos parte del árbol (el recorrido de la hoja hacia la raíz), entonces siempre mantenemos las referencias actualizadas para poder saber cuántas hojas tiene cada subárbol. Luego, para calcular la $m-$ésima posición, damos $m - 1$ saltos para poder encontrar la siguiente posición a eliminar. En dado caso de que la suma que realicemos esté fuera del rango n, tenemos que calcular `((posicion-1)% n) + 1` para poder regresar nuestro valor de nuevo al rango n y que además sea el correcto. Así que por esto podemos garantizar que las referencias están siempre actualizadas y que siempre vamos a poder calcular la $m-$ésima posición correctamente.

Ahora bien, notemos que construir el árbol nos toma $O(n)$ ya que lo estamos construyendo de abajo hacia arriba, y progresivamente vamos formando los $n - 1$ nodos internos. Crear la lista vacía nos toma tiempo constante. Buscar al $m-$ésimo elemento nos toma $O(\log n)$ ya que en el peor de los casos recorremos la altura del árbol (pues los elementos que buscamos son hojas), pero esto lo hacemos n veces, por lo que nos toma $O(n \log n)$ esta operación en total. La operación eliminar no elimina nodos del árbol, lo único que hace es borrar las referencias de él en el árbol al disminuir en una unidad todos los elementos en los nodos visitados al momento de buscar la hoja (hacemos como que no existe). Entonces, tenemos que recorrer en el peor de los casos la altura del árbol, por lo que nos toma tiempo $O(\log n)$. Pero, esta operación la realizamos n veces, así que en total nos toma tiempo $O(n \log n)$. El

resto de las operaciones que realizamos son constantes, pues sólo hacemos asignaciones. Por lo tanto, la complejidad total del algoritmo es de $O(n \log n)$.

Un ejemplo para ilustrar la construcción del árbol y la búsqueda de un elemento sería el siguiente: si $n = 5$ (tenemos 5 hojas) y buscamos al elemento 3, entonces el árbol se vería como

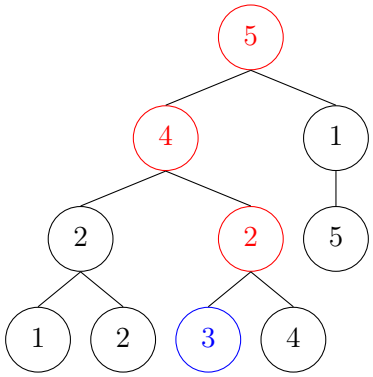

Figura 7: En color rojo está el camino a seguir para encontrar al elemento 3

3. Queremos ordenar una lista S de n enteros que contiene muchos elementos duplicados. Supongamos que los elementos de S sólo toman $O(\log n)$ valores distintos.

   - Encuentre un algoritmo que toma a lo más $O(n \log n)$ tiempo para ordenar S.
   - ¿Por qué esto no viola la cota inferior de $O(n \log n)$ para el problema de ordenación?

4. Dado un arreglo A de n números, queremos contestar la pregunta ¿Hay algún elemento de A que aparezca al menos $\frac{n}{3}$ veces? Encuentre un algoritmo lineal para resolver este problema.

   Solución: Primero, necesitamos encontrar elementos que sean candidatos

5. Sea $A[1, \ldots, n]$ un arreglo de números reales. Diseña un algoritmo que realice cualquier secuencia de las siguientes operaciones:

   - $Add(i, y)$, suma el valor y al i-ésimo número.

     Solución: Sea n la longitud del arreglo A. Para realizar esta operación, lo que haremos será construir un árbol binario balanceado donde las hojas de éste sean los elementos de A (sin cambiar su órden de aparición). Los nodos internos los vamos a construir con la siguiente condición: el nodo padre de las hojas tendrá como elemento el número de elementos que contiene de A. Los demás nodos internos tendrán como elemento el número de elementos del arreglo A que contienen sus dos subárboles. Una vez que tengamos el árbol construido, entonces debemos recorrerlo de tal forma que lleguemos al $i-$ésimo elemento de A y luego sumarle el valor de y.

     Esto lo podemos lograr realizando el siguiente algoritmo:

     a) Nos posicionamos en la raíz del árbol y hacemos `nodo = raiz`.

     b) Mientras el hijo izquierdo del `nodo` no sea una hoja, hacemos:

     - Si i es menor o igual al elemento del hijo izquierdo del `nodo`, entonces nos movemos al subárbol izquierdo de éste, es decir, `nodo = nodo.hijoIzquierdo`.
     - En otro caso, actualizamos el valor de i como $i = i-$ `nodo.hijoIzquierdo.elemento` y nos movemos al subárbol derecho; es decir, `nodo = nodo.hijoDerecho`.

     c) Para este punto, los hijos del `nodo` serán ambos hojas (si es que tiene dos). Así que tenemos dos opciones:

     - Si i es igual a 1, entonces nos movemos a la hoja izquierda; es decir, `nodo = nodo.hijoIzquierdo`.
     - En otro caso, nos movemos a la hoja derecha; es decir, `nodo = nodo.hijoDerecho`.

d) Al elemento de `nodo` le sumamos el valor y. Terminamos.

Este algoritmo funciona porque en cada paso siempre estamos en el subárbol donde se encuentra el $i-$ésimo elemento. Primero nos posicionamos en la raíz del árbol, y la variable `nodo` nos ayuda a mantener la referencia sobre el nodo en el que estamos posicionados en este momento. Luego hacemos un recorrido entre los nodos internos hasta que llegamos a uno donde su hijo izquierdo es una hoja (sólo verificamos al hijo izquierdo, por que un nodo interno puede tener uno o dos hijos, y si sólo tiene uno, entonces será el izquierdo). Durante este recorrido tenemos dos casos: nos movemos al subárbol izquierdo o al subárbol derecho. Para tomar esta decisión, debemos comparar al elemento del hijo izquierdo del nodo donde estamos posicionados con nuestro índice i. Como los nodos internos *saben* cuántos elementos de A tienen sus subárboles, entonces si i es menor o igual a `nodo.hijoIzquierdo.elemento` eso quiere decir que el elemento i se encuentra dentro del rango de elementos que contiene el subárbol izquierdo de `nodo`. En caso contrario, debe estar en el subárbol derecho de `nodo`. Pero, si nos movemos al subárbol derecho debemos de actualizar el valor de i, así que le restamos el número de elementos de A donde sabemos que i no va a estar (esto para que tenga el valor adecuado respecto al nuevo subárbol con el que vamos a tratar). Una vez que encontramos al nodo interno cuyos hijos (si es que tiene dos) son hojas, entonces tenemos dos opciones: elegir la hoja izquierda o la derecha. Esta decisión la tomaremos dependiéndo del valor que tenga i en ese momento. Si i es igual a 1 entonces el elemento que buscamos está en la hoja izquierda. En caso contrario, el elemento que buscamos está en la hoja derecha. Por los cambios que sufre i en el recorrido de nodos internos, entonces los valores finales que puede tomar i son 1 o 2; así que dependiéndo de esto, decidimos la hoja que le corresponde. Finalmente, como encontramos la hoja que contiene al $i-$ésimo elemento de A, simplemente le sumamos el valor de y y terminamos.

La construcción del árbol nos toma $O(n)$ en tiempo y en espacio (por lo visto en clase). Como los elementos del arreglo A se encuentran en las hojas, entonces en el peor de los casos tenemos que recorrer la altura del árbol para encontrar al elemento que deseamos. Y como sumar el elemento y con el elemento encontrado nos toma tiempo constante, entonces la complejidad total del algoritmo de búsqueda es de $O(\log n)$.

Un ejemplo para ilustrar el algoritmo sería el siguiente: si $A = [7, 14, 2, 8, 9, 11, 0, -1, 1]$ y `Add(7,6)`, entonces

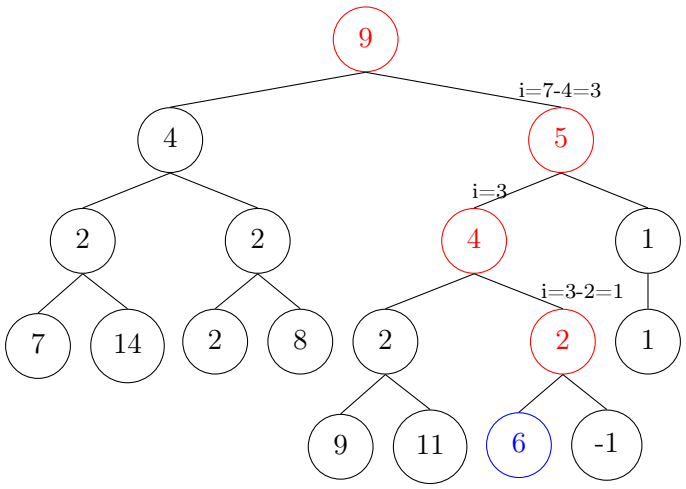

Figura 8: Como $i = 7$, entonces actualizamos la posición con $0 + 6 = 6$

- $Partial - sum(i)$, regresa la suma de los primeros i números, es decir,
  $Partial - sum(i) = A[1] + \ldots + A[i]$
  Solución: Usaremos básicamente el árbol del inciso anterior, sólo agregaremos el hecho de que cada nodo tiene una llave, la cual es la suma del valor de los elementos de sus hijos (para las hojas, el valor de su elemento es su propia llave). Ahora bien, para encontrar la suma de los $i-$ésimos elementos de

A entonces hay que recorrer el árbol hasta encontrar el $i-$ésimo elemento e ir sumando el valor de las llaves de los elementos anteriores en un contador, para que así podamos regresar la suma deseada.

Esto lo podemos lograr realizando el siguiente algoritmo:

a) Nos posicionamos en la raíz del árbol y hacemos `nodo = raiz`. Además, creamos un contador `cc=0`.

b) Mientras el hijo izquierdo del `nodo` no sea una hoja, hacemos:

- Si i es menor o igual al elemento del hijo izquierdo del `nodo`, entonces nos movemos al subárbol izquierdo de éste, es decir, `nodo = nodo.hijoIzquierdo`.
- En otro caso, actualizamos el valor de i como $i = i-$ `nodo.hijoIzquierdo.elemento`, actualizamos el valor de `cc` como `cc = cc + nodo.hijoIzquierdo.llave` y nos movemos al subárbol derecho; es decir, `nodo = nodo.hijoDerecho`.

c) Para este punto, los hijos del `nodo` serán ambos hojas (si es que tiene dos). Así que tenemos dos opciones:

- Si i es igual a 1, entonces nos movemos a la hoja izquierda; es decir, `nodo = nodo.hijoIzquierdo`.
- En otro caso, nos movemos a la hoja derecha; es decir, `nodo = nodo.hijoDerecho`.

d) Hacemos `cc = cc + nodo.llave`. Terminamos.

Este algoritmo funciona porque en cada paso siempre estamos en el subárbol donde se encuentra el $i-$ésimo elemento y porque gracias a la llave siempre podemos obtener la suma de los elementos anteriores al $i-$ésimo. Por el inciso anterior, sabemos que este algoritmo efectivamente encuentra al $i-$ésimo elemento, lo único que agregamos es que cada vez que nos movemos al subárbol derecho estamos sumando el valor de la llave del nodo hermano al contador `cc` que llevará la suma de los elementos anteriores a i. Como los cambios a la derecha que estamos contando son aquellos que están en el recorrido que hacemos de la raíz al $i-$ésimo elemento, entonces así nos aseguramos de no contar de más ni de menos. Por lo que, cuando llegamos al $i-$ésimo elemento, ya tenemos en el contador `cc` la suma de los $(i-1)$ elementos de A, así que sólo sumamos la llave del $i-$ésimo elemento con el valor actual del contador y así obtenemos la suma de los primeros i-ésimos elementos de A.

Finalmente, como el contador se actualiza de forma constante y éste va realizando su proceso durante el algoritmo de búsqueda del $i-$ésimo elemento, entonces la complejidad total del algoritmo será de $O(\log n)$, pues en el peor caso, tenemos que recorrer toda la altura del árbol para encontrar el $i-$ésimo elemento de A.

Un ejemplo para ilustrar el algoritmo sería el siguiente: si $A = [5, 6, -2, -1, 8, 7]$ e $i = 5$, entonces

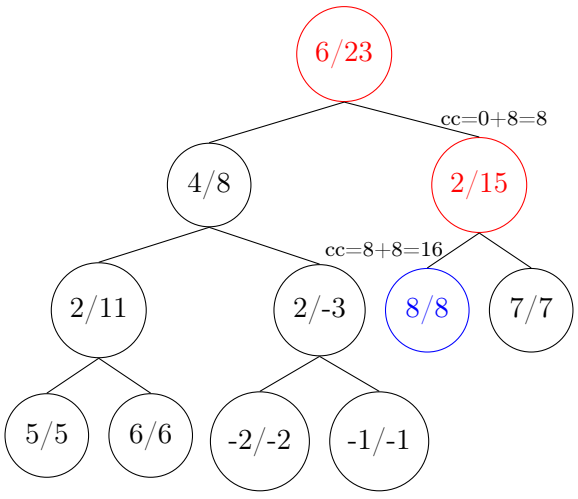

Figura 9: Visualmente, en cada nodo, los valores a la izquierda son los elementos de los nodos y los valores a la derecha son las llaves. Al final del algoritmo, `cc = 16`

6. Sea A un arreglo de n números enteros distintos. Suponga que A tiene la siguiente propiedad: existe un indice $1 \leq k \leq n$ tal que $A[1], \ldots, A[k]$ es una secuencia incremental y $A[k+1], \ldots, A[n]$ es una secuencia decremental.

   a) Diseña y analiza un algoritmo eficiente para encontrar k.

   Solución: Tenemos que nuestro arreglo está dividido en dos *subarreglos*, el primero ordenado en forma creciente y el segundo está ordenado en forma decreciente. Supongamos que n es la longitud del arreglo A y que B y C son los subarreglos contenidos en el arreglo A. Por ejemplo, supongamos que A es el siguiente arreglo:

-5	-2	0	2	3	8	6	4	1

   Entonces el arreglo B corresponde a los valores que están de color azul, y el arreglo C corresponde a los valores de color rojo. En este ejemplo, $k = 5$ y $A[k] = 8$.

   Para encontrar el valor de k, lo que debemos hacer es buscar al elemento $A[i]$ tal que

$$A[i-1] < A[i] > A[i+1] \tag{1}$$

   es decir, al elemento en el arreglo cuyos elementos adyacentes son menores que él; y por lo tanto, es el elemento que está al final del subarreglo B. Esto se debe a que los subarreglos B y C están ordenados de forma creciente y decreciente, respectivamente.

   Ahora bien, utilizaremos el método *divide y vencerás* para resolver este problema: Realizamos la operación $\lceil \frac{n}{2} \rceil$ para encontrar el índice del elemento que se encuentra a la mitad del arreglo. Si el elemento $A[\lceil \frac{n}{2} \rceil]$ cumple las condiciones de la expresión 1, entonces hemos encontrado al elemento en el índice k. Terminamos. En caso contrario, lo que hacemos es verificar si el elemento $A[\lceil \frac{n}{2} \rceil]$ pertenece al subarreglo B o al subarreglo C. Si el elemento $A[\lceil \frac{n}{2} \rceil]$ pertenece al subarreglo B, entonces cumple la propiedad

$$A[i-1] < A[i] \tag{2}$$

   es decir, el elemento anterior a $A[\lceil \frac{n}{2} \rceil]$ debería ser menor que él. Pero si el elemento $A[\lceil \frac{n}{2} \rceil]$ pertenece al subarreglo C, entonces debe cumplir la propiedad

$$A[i] > A[i+1] \tag{3}$$

   es decir, el elemento siguiente a $A[\lceil \frac{n}{2} \rceil]$ debería ser menor que él. Luego, si $A[\lceil \frac{n}{2} \rceil]$ pertence al subarreglo B, entonces realizamos este procedimiento recursivamente sobre la mitad derecha del arreglo A (ya que todavía no alcanzamos el final de este subarreglo), pero si $A[\lceil \frac{n}{2} \rceil]$ pertenece al subarreglo C, entonces realizamos este procedimiento recursivamente sobre la mitad izquierda del arreglo A (ya que nos pasamos del final del subarreglo B).

   A esto se debe añadir un caso especial: Si el arreglo es de longitud $n = 1$, entonces simplemente regresamos el índice del primer (y único) elemento de A.

   Durante la ejecución de este algoritmo realizamos operaciones constantes (las comparaciones) y trabajamos con todo el arreglo A sólo para encontrar la mitad del arreglo original, después vamos trabajando con subarreglos de longitud $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots, 1$; por lo que la complejidad de nuestro algoritmo es de $O(\log n)$. Es decir, realizamos una variación de *búsqueda binaria*. Y funciona porque cada subarreglo sobre el cual vamos trabajando eventualmente nos llevará al elemento que está en el índice k, que es el que estamos buscando.

   b) Si no conoces el valor de n, cómo resuelves el problema.

   Solución: Como no sabemos cuál es la longitud del arreglo A, entonces necesitamos ver una forma de cómo irnos moviéndo a través del arreglo. Para solucionar esto, los índices que vamos a ir revisando son aquellos que sean potencia de 2, es decir, los índices 2^i tales que $i \in \{0, 1, 2, 3 \ldots\}$. Ahora bien , para poder encontrar el valor de k, primero obtendremos un rango $[a, b]$ de elementos donde poder

aplicar el algoritmo del inciso anterior. Para esto, necesitaremos dos contadores a y b, los cuales nos indicarán la posición de inicio y final del intervalo.

El algoritmo que seguiremos para encontrar el intervalo $[a, b]$ será el siguiente: inicializamos nuestros contadores a, b e i en 0. Luego, realizamos la operación

$$2^i = 2^0 = 1$$

e incrementamos nuestro contador i en una unidad. Además, como b nos indicará la posición final del intervalo, entonces

$$b = b + 2^i = 0 + 1 = 1$$

Así, nuestro primer candidato de intervalo será $[a, b] = [0, 1]$. Luego, verificamos que el elemento en la posición $A[2^i]$ cumpla la propiedad 1. Si lo hace, entonces hemos encontrado al elemento cuyo índice es k. En caso contrario, debemos verificar si el elemento $A[2^i]$ pertenece al subarreglo B o al subarreglo C. Esto lo hacemos aplicándole las propiedades 2 y 3, respectivamente. Ahora bien,

- Si el elemento $A[2^i]$ pertenece al subarreglo B, entonces el valor de a será el valor que tenga b en este momento, ya que queremos ir moviéndo el intervalo de tal forma que podamos asegurar que el elemento en el índice k se encuentra dentro de éste. Luego, realizamos la operación 2^i con el valor que tenga i en este momento, lo aumentamos en una unidad y volvemos a actualizar nuestro contador b como

  $$b = b + 2^i$$

  Esto lo hacemos porque aún nos falta camino por recorrer para llegar al final del subarreglo B. Así, nuestro nuevo intervalo candidato es $[a, b]$. Luego, volvemos a la parte de verificación para saber si el elemento $A[b]$ es el que estamos buscando (o para saber qué hacer en casao de que no).
- Si el elemento $A[2^i]$ pertenece al subarreglo C, entonces ya hemos encontrado el intervalo que necesitamos. Esto se debe a que hemos estado moviéndo el intervalo hasta que nos encontramos con un elemento del subarreglo C, y aquí se detiene la búsqueda del intervalo porque ya no es necesario buscar más allá, ya que nuestro contador a estará posicionado en un elemento del subarreglo B (que no es el que buscamos) y b estará posicionado en un elemento del subarreglo C. Entonces, así podemos garantizar que el elemento en el índice k se encuentra en el intervalo $[a, b]$, donde a y b serán los valores que éstos contadores tengan al momento de encontrar al elemento en el subarreglo C, y éstos serán de la forma

  $$a = 2^0 + 2^1 + \ldots \quad \text{y} \quad b = 2^0 + 2^1 + 2^2 + \ldots$$

  Es decir, a tendrá un valor 2^i menor que b. Una vez que tenemos este intervalo, podemos mandar a llamar el algoritmo descrito en el inciso anterior y así obtener el valor k que estamos buscando.

Como nota especial, si al momento de querer acceder al elemento $A[2^i]$ no lo logramos (no existe), entonces sólo hay que volver a iniciar desde cero desde la posición 2^{i-1}; es decir, tendríamos $a = 2^{i-1}$ como el inicio del arreglo y empezamos a buscar las potencias de 2 desde $i = 0$. Así, podemos continuar con el algoritmo descrito arriba.

Este algoritmo funciona porque las potencias de dos nos ayudan a encontrar el intervalo que necesitamos (el inicio es un elemento de B y el final es un elemento de C), y el resto lo termina el algoritmo del inciso anterior; por lo que así obtenemos correctamente nuestro valor k.

Para ilustrar un poco el algoritmo, mostraremos cómo funcionaría con el mismo arreglo A del inciso anterior:

7. You are a young scientist who just got a new job in a large team of 100 people (you the 101- st). A friend of yours who you believe told you that you have more honest colleagues than liars, and that that's all what he can tell you, where a liar is a person who can either lie or tell the truth, while an honest person is one who always tells the truth. Of course, you'd like to know exactly your honest colleagues and the liars, so that you decide to start an investigation, consisting of a series of questions you are going to ask your

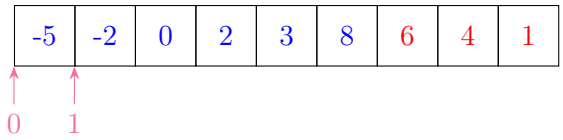

Figura 10: $A[2^0] = -2$ pertenece a B, actualizamos $a = 1$, $b = b + 2^1 = 3$, $i = 2$

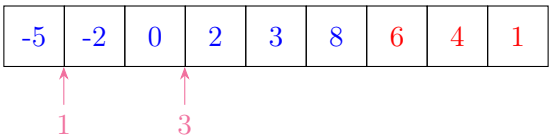

Figura 11: $A[2^i] = 2$ pertenece a B, actualizamos $a = 3$, $b = b + 2^2 = 3 + 4 = 7$, $i = 3$

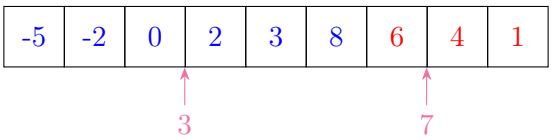

Figura 12: $A[2^i] = 4$ pertenece a C, ya encontramos el intervalo y aplicamos algoritmo del inciso a)

colleagues. Since you don't wish to look suspicious, you decide to ask only questions of the form "Is Mary an honest person?" and of course, to ask as few questions as possible. Can you sort out all your honest colleagues? What's the minimum number of questions you'd ask in the worst case? You can assume that your colleagues know each other well enough to say if another person is a liar or not. (Hint: Group people in pairs (X,Y) and ask X the question "Is Y honest?" and Y the question "Is X honest?". Analyze all the four possible answers. Once you find an honest person, you can easily find all the others. Challenge: can you solve this enigma asking less than 280 questions in total?)

Generalize the strategy above and show that given n people such that less than half are liars, you can sort them out in honest persons and liars by asking $\theta(n)$ questions.

8. Suponga que tenemos dos arreglos ordenados $A[1 \ldots n]$ y $B[1 \ldots n]$ y un entero k. Describe un algoritmo para encontrar el k-ésimo elemento en la unión de A y B. Por ejemplo, si $k = 1$, tu algoritmo debe regresar al elemento más pequeño de $A \cup B$; si $k = n$, tu algoritmo debe regresar la mediana de $A \cup B$. Puedes suponer que los arreglos no contienen duplicados. Tu algoritmo debe tener complejidad de tiempo $\Theta(\log n)$. Hint: Primero resuelve el caso especial $k = n$.

   SOLUCIÓN: Sabemos que ambos arreglos A y B son del mismo tamaño n.

9. Considera que un río fluye de norte a sur con caudal constante. Suponga que hay n ciudades en ambos lados del río, es decir n ciudades a la izquierda del río y n ciudades a la derecha. Suponga también que dichas ciudades fueron numeradas de 1 a n, pero se desconoce el orden. Construye el mayor número de puentes entre ciudades con el mismo número, tal que dos puentes no se intersecten.

   SOLUCIÓN: Supongamos que las ciudades a la izquierda del río son $a_1, a_2, \ldots, a_n$ y las ciudades a la derecha del río son $b_1, b_2, \ldots, b_n$. Como las ciudades fueron etiquetadas con un número del 1 al n, entonces definimos una función `etiqueta` que nos regresa el número que le fue asignado a cada una de las ciudades. Luego, escogemos las ciudades de un lado del río para poder Sin pérdida de generalidad, digamos que escogemos las ciudades del lado izquierdo del río para etiquetarlas y ordenarlas. Así, las ciudades de ese lado del río

quedarían como:

$$c_1 = 1 \to a_1$$
$$c_2 = 2 \to a_2$$
$$\dots$$
$$c_n = n \to a_n$$

y

$$c_1 < c_2 < \ldots < c_n$$

Ahora bien, mientras ordenamos las ciudades del lado izquierdo del río, también vamos a ir moviéndo las ciudades que estaban asociadas