Facultad de Ciencias, UNAM
Análisis de Algoritmos
Tarea 1

Rubí Rojas Tania Michelle

09 de octubre de 2020

1. ¿Cuántas comparaciones son necesarias y suficientes para ordenar cualquier lista de cinco elementos? Justifique su respuesta.

   Solución: Un árbol de decisión es un árbol binario completo que representa las comparaciones realizadas en todas las ejecuciones posibles sobre entradas de tamaño n. Cada nodo interno se representa de la forma $i : j$ con $1 \leq i < j \leq n$ y cada hoja con una permutación $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$ sobre $\{1, 2, ..., n\}$.

   La ejecución del algoritmo de ordenamiento sobre una entrada dada corresponde a un camino en el árbol desde la raíz hasta la hoja. Dicho camino denota las comparaciones realizadas y su órden de realización para la entrada. Es decir, un nodo $i : j$ en el camino denota la comparación $a_i \leq a_j$. Si el camino continúa sobre el hijo izquierdo, la comparación es cierta y si continúa sobre el hijo derecho entonces la comparación es falsa. Si la entrada alcanza la hoja $\langle \pi(1), \pi(2), ..., \pi(n) \rangle$, entonces la permutación de entrada ordenada es $\langle a_{v\pi(1)}, a_{\pi(2)}, ..., a_{\pi}(n) \rangle$.

   Un ejemplo de árbol de decisión sería el siguiente:

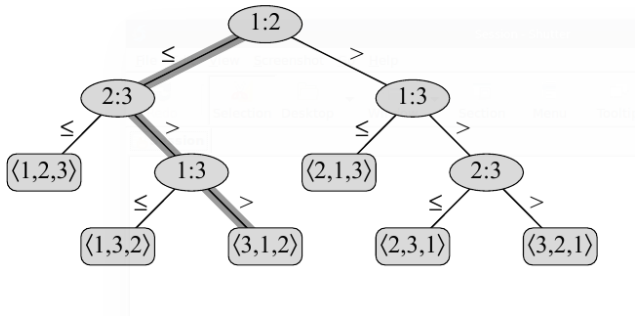

Figura 1: Árbol de decisión correspondiente a tres elementos (Cormen, pag. 192)

La entrada corresponde a $\langle a_1, a_2, a_3 \rangle = \langle 6, 8, 5, \rangle$. Lo primero que hace es la comparación $a_1 \leq a_2$, que es cierta. Luego se compara $a_2 \leq a_3$, que es falsa. Finalmente compara $a_1 \leq a_3$, que también es falsa. La entrada ordenada es $\langle 5, 6, 8 \rangle$. Notemos que hay $3! = 6$ posibles permutaciones en la entrada de elementos, por lo que el árbol de decisión tiene 6 hojas.

Ahora bien, de manera más general, digamos que para n elementos tenemos un árbol de decisión T de altura h y con l hojas alcanzables. Como el árbol ordena las $n!$ distintas permutaciones, entonces el árbol contiene una hoja por cada una de las $n!$ permutaciones, por lo que $n! \leq l$. Como un árbol

binario de altura h tiene a lo más 2^h hojas, entonces $n! \leq l \leq 2^h$, de donde

$$
\begin{aligned}
2^h &\geq n! && \text{por transitividad} \\
\log_2(2^h) &\geq \log_2(n!) && \text{tomando los logaritmos} \\
h &\geq \log_2(n!) && \text{propiedad de logaritmo} \\
&= \log_2(n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \cdots \cdot 3 \cdot 2 \cdot 1) && \text{definición de factorial} \\
&= \log_2(n) + \log_2(n-1) + \log_2(n-2) + \log_2(n-3) + && \text{propiedad de logaritmo} \\
&\quad \cdots + \log_2(3) + \log_2(2) + \log_2(1) && \\
&= \int_1^n \log_2(x)\,\mathrm{d}x. && \text{es el área bajo la curva} \\
&= x\log_2(x) - x\Big|_1^n && \text{resolvemos la integral}
\end{aligned}
$$

Es decir, para poder ordenar un arreglo de n elementos son `necesarias` y `suficientes`

$$h = x\log_2(x) - x\Big|_1^n$$

comparaciones. ¿Por qué? Esto se debe a que el árbol de decisión ya contempla las $n!$ permutaciones de nuestro arreglo y éstas se encuentran en las hojas del árbol, así que para acceder a ellas necesitamos recorrer (en un peor caso, si se quiere ver así) toda la altura del árbol. Son `suficientes` porque si agregamos un nivel al árbol, realmente no estamos obteniéndo información nueva (sino repetida) pues el árbol ya se encargó de contemplar todas las comparaciones que necesitamos. Ahora bien, son `necesarias` porque si eliminamos un nivel al árbol entonces estamos eliminando comparaciones que son realmente de utilidad, por lo que como mínimo debemos tener ese número de comparaciones.

En particular, si $n = 5$ entonces tendríamos que son `necesarias` y `suficientes`

$$h = 5\log_2(5) - 5\Big|_1^5 = 7$$

comparaciones.

2. Dados dos arreglos ordenados A y B de longitud n y m, respectivamente. Diseña un algoritmo de tiempo $O(n+m)$ que obtenga un arreglo C que contenga los elementos en común entre A y B, C no debe tener elementos repetidos.

   Solución: El algoritmo propuesto para resolver este problema es el siguiente

Algorithm 1 Obtener los elementos en común entre los arreglos A y B

```
encontrarInterseccion(A, B):
1:  i ← 0
2:  j ← 0
3:  C = []
4:  while (i < n and j < m) do
5:      if A[i] == B[j] then
6:          if C.length > 0 and C[C.length − 1] == A[i] then
7:              i ← i + 1
8:              j ← j + 1
9:          else
10:             C.append[A[i]]
11:             i ← i + 1
12:             j ← j + 1
13:         end if
14:     else
15:         if A[i] < B[j] then
16:             i ← i + 1
17:         else
18:             j ← j + 1
19:         end if
20:     end if
21: end while
```

Primero, explicaremos porqué el algoritmo funciona. En las líneas $(1-2)$ estamos definiéndo dos variables i, j; las cuales nos ayudarán a recorrer los arreglos A y B, respectivamente. En la línea 3 simplemente creamos a nuestro arreglo C. A partir de la línea 4 empieza lo interesante: como sabemos que los arreglos A y B están ordenados, eso significa que tenemos que checar tres casos en específico:

a) (líneas $(5-13)$). Si el $i-$ésimo elemento de A es igual al $j-$ésimo elemento de B eso significa que tienen un elemento en común y, en teoría, se debe agregar al arreglo C. Pero antes de agregarlo, comprobamos que no esté repetido en C: para verificar esto simplemente hay que ver si el elemento $A[i]$ (o $B[j]$) es igual al último elemento que fue agregado a C (como A y B están ordenados, si es que tienen elementos repetidos, entonces éstos están juntos). Si el elemento ya se encuentra en C entonces simplemente incrementamos nuestros contadores en *uno*; en caso contrario, agregamos al elemento a C e incrementamos los contadores en *uno*.

b) (líneas $(15-16)$). Si el $i-$ésmo elemento de A es menor que el $j-$ésimo elemento de B, eso quiere decir que debemos movernos un índice a la derecha en el arreglo A; ya que como están ordenados ambos arreglos, eso quiere decir que, si tienen elementos en común, entonces éste (o éstos) es mayor que el $i-$ésimo elemento de A.

c) (línea 18). Aquí cae el caso en que $B[j] < A[i]$, y análogamente al caso anterior, tenemos que movernos un índice a la derecha en el arreglo B; ya que como están ordenados ambos arreglos,

eso quiere decir que, si tiene elementos en común, entonces éste (o éstos) es mayor que el $j-$ésimo elemento de B.

Todo esto se realizará hasta que la condición de nuestro *while* (línea 14) se cumpla: como n puede ser diferente de m, lo que hay que tener en cuenta es que si terminamos de recorrer un arreglo, entonces debemos terminar el proceso. Esto se debe al hecho de que los arreglos están ordenados. Si terminamos de recorrer un arreglo, entonces ya no habrá más elementos en común, es decir, podríamos recorrer un arreglo sin terminar de recorrer al otro; pero en el peor caso, debemos recorrer completamente ambos arreglos, por este motivo la complejidad de nuestro algoritmo es $O(n+m)$. En particular, porque sólo estamos recorriéndo los arreglos y las demás operaciones que realizamos se hacen en tiempo constante.

3. Consider the following sorting algorithm:

```
STUPIDSORT(A[0..n-1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      STUPIDSORT(A[0..m-1])
      STUPIDSORT(A[n-m..n-1])
      STUPIDSORT(A[0..m-1])
```

a) Prove that STUPIDSORT actually sorts its input.

Demostración. Inducción fuerte sobre la longitud n del arreglo.

- **Caso base**
  $n = 2$. Aquí tenemos dos posibles casitos:
  1) $A[0] < A[1]$. No se contempla como tal en el algoritmo, pero seguramente es porque esto significa que el arreglo ya está ordenado y no hay nada más que hacer.
  2) $A[0] > A[1]$. En este caso, como el elemento menor se encuentra al final del arreglo, entonces intercambiamos las posiciones del primer y último elemento con ayuda del `swap`; y así obtenemos que el algoritmo ordena este caso, y por lo tanto se cumple.
- **Hipótesis de inducción**
  Supongamos que el algoritmo STUPIDSORT ordena todos los arreglos de longitud k que sean menores a n.
- **Paso inductivo**
  Primero, mostraremos que $\lceil \frac{2n}{3} \rceil < n$ para $n > 2$.

  Demostración. Inducción sobre n.

  - **Caso base**
    $n = 3$. Este caso se cumple ya que

    $$\left\lceil \frac{2(3)}{3} \right\rceil < 3$$
    $$\lceil 2 \rceil < 3$$
    $$2 < 3$$

  - **Hipótesis de inducción**
    Supongamos que se cumple para $n > 2$, es decir, que se cumple $\lceil \frac{2n}{3} \rceil < n$.

- **Paso inductivo**
  Queremos mostrar que se cumple para $n+1$, es decir, que se cumple $\lceil \frac{2(n+1)}{3} \rceil < n+1$. Entonces

$$\begin{aligned}
\left\lceil \frac{2(n+1)}{3} \right\rceil &= \left\lceil \frac{2n+2}{3} \right\rceil && \text{aritmética} \\
&= \left\lceil \frac{2n}{3} + \frac{2}{3} \right\rceil && \text{propiedad de fracciones} \\
&< \left\lceil n + \frac{2}{3} \right\rceil && \text{por H.I} \\
&< \lceil n+1 \rceil && \text{ya que } \frac{2}{3} < 1 \\
&< n+1
\end{aligned}$$

  Por lo tanto, $\lceil \frac{2n}{3} \rceil < n, n > 2$.

□

Ahora bien, como $m = \lceil \frac{2n}{3} \rceil < n, n > 2$, entonces

- `STUPIDSORT(A[0...m-1])`
  Como $m - 1 < n$, por hipótesis de inducción podemos concluir que el algoritmo efectivamente ordena este arreglo.
- `STUPIDSORT(A[n-m...n-1])`
  Como $n - m < n$ y $n - 1 < n$, por hipótesis de inducción podemos concluir que el algoritmo efectivamente ordena este arreglo.

Por lo tanto, el algoritmo StupidSort realmente ordena su entrada. □

b) Would the algorithm still sort correctly if we replaced $m = \lceil \frac{2n}{3} \rceil$. Justify your answer.
SOLUCIÓN:

c) Show that the number of swaps executed by STUPIDSORT is at most $\binom{n}{2}$.

Demostración. □

4. Supongamos que tenemos que ordenar una lista L de n enteros cuyos valores están entre 1 y m. Pruebe que si m es $O(n)$ entonces los elementos de L pueden ser ordenados en tiempo lineal. ¿Qué pasa si m es de $O(n^2)$? ¿Se puede realizar en tiempo lineal? ¿Por qué?

Demostración. Supongamos que m es $O(n)$. Para probar nuestra primera interrogante, nos auxiliaremos del algoritmo de ordenamiento *Counting Sort*. Éste supone que cada uno de los n elementos de entrada es un natural en un rango de a lo más k. El objetivo de este algoritmo es determinar, para cada elemento de entrada x, el número de elementos que son menores a x. Esto es usado para colocar a x en la posición adecuada de salida del arreglo. En el algoritmo de *Counting Sort* suponemos que la entrada es el arreglo $A_{[1...n]}$ y que su longitud es n. Además, necesitamos de dos arreglos extra: $B_{[1...n]}$ que tiene los elementos ordenados, y $C_{[0...k]}$ que sirve como almacenamiento temporal. Hay que tener en cuenta que este algoritmo nunca hace comparaciones, sino que usa los valores de los elementos para indicar las posiciones en el arreglo.

Las líneas $(2-3)$ inicializan al arreglo C (creado en la línea 1) con todas sus entradas en 0. Este proceso nos toma $\Theta(k)$. En las líneas $(4-5)$ estamos contando las veces que aparece cada uno de los elementos de A en este mismo arreglo. Como tenemos que recorrerlo todo, entonces este proceso nos toma $\Theta(n)$. Las líneas $(7-8)$ determinan, para cada $i = 0, 1, ..., k$, cuántos elementos de A son menor o iguales a i, al estar realizando una suma *corrida* entre los elementos de C. Nuevamente,

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Figura 2: Algoritmo Counting Sort (Cormen, pag. 195)

como estamos recorriendo todo el arreglo C, este proceso nos toma $\Theta(k)$. Finalmente, las líneas $(10-12)$ colocan a cada elemento $A[j]$ en la correcta posición que le corresponde para que esté ordenad, y esta colocación se hará en el arreglo B. Como recorremos a todo nuestro arreglo A, eso quiere decir que este proceso nos toma $\Theta(n)$. Por lo tanto, la complejidad del algoritmo *Counting Sort* es $\Theta(k+n)$. Así, por lo anterior, si $k = O(n)$, esto quiere decir que $T(n) = n$, y por lo tanto el algoritmo será lineal.

Cuando m es $O(n^2)$, también es posible ordenarlo en $O(n)$. Para ello, utilizaremos una variación de *Radix Sort*, el cual también hemos discutido en clase. El siguiente paso forma parte del algoritmo original: *para cada elemento i lo que haremos es ordenar el arreglo de entrada tomando en consideración el número de menos significado del más pequeño al más grande.* Para usar este algoritmo debe haber d digitos en los enteros de entrada. Usualmente, *Radix Sort* toma $O(d \cdot (n+k))$, donde k es la base para representar números (por ejemplo, la base decimal). Como $m = O(n^2)$, entonces ese es su mayor valor posible, por lo que $d = O(\log_k n)$. Así, la complejidad sería $O(n+k) \cdot O(\log_k n)$. El truco aquí está en cambiar la base k. Si reemplazamos a k con n entonces el valor de $O(\log_k n)$ se convierte en $O(1)$, y la complejidad se convierte en $O(n)$. □

5. Describe an algorithm that, given n integers in the range 0 to k, preprocesses its input and then answers any query about how many of the n integers fall into a range $[a...b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

   SOLUCIÓN: La solución propuesta es utilizar las líneas $(1-9)$ del algoritmo *Counting Sort* para preprocesar los n enteros en el rango 0 a k.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

Figura 3: Algoritmo Counting Sort (Cormen, pag. 195)

Por el análisis visto en el inciso anterior, sabemos que esto se hace en $\Theta(k+n)$.

Ahora tenemos un arreglo $C_{[0...k]}$ en donde cada $C_{[i]}$ contiene el número de enteros que son menores o iguales a i, por lo que es fácil ver que el número de enteros en el rango $[a...b]$ es $C_[b] - C_[a-1]$, donde podemos interpretar a $C_{[-1]}$ como 0. Esto último toma $O(1)$ porque lo único que hacemos es acceder a dos elementos en el arreglo C (sabemos que esto es constante) y posteriormente restar.

6. Sea A un arreglo de n elementos, tal que cada elemento se encuentra a lo más a k posiciones de su posición ordenada. Diseñe un algoritmo que ordene A en $O(n \log k)$.

   Solución: El algoritmo propuesto es el siguiente

Algorithm 2 Ordenar un arreglo cuyos elementos se encuentran a lo más k posiciones de su posición correcta.
ordenaArreglo(A, k):

```
1: T ← creaMinHeap(A, k + 1)
2: T ← T.actualizaMinimo
3: i ← 0
4: j ← k + 1
5: while i ≤ A.length − 1 do
6:     A[i] ← T.getRaiz
7:     i ← i + 1
8:     T ← T.eliminaMinimo()
9:     if j < A.length − 1 then
10:        j ← j + 1
11:        T ← T.agregaElemento(A[j])
12:    end if
13: end while
14: return A
```

Primero, explicaremos por qué el algoritmo funciona. Como los elementos del arreglo A se encuentran a lo más k posiciones de su posición correcta, entonces el elemento más pequeño se debería encontrar en el subarreglo de longitud $k+1$, ya que en el peor caso, éste se encuentra justo a k posiciones de su posición correcta; y al tomar el subarreglo con los elementos $k+1$ entonces nos aseguramos de tomar al elemento más pequeño (y en caso de que esté antes de la posición k entonces de todas formas lo estamos agarrando). Tomando este subarreglo de longitud $k+1$ vamos a crear un árbol, en particular un `MinHeap`, por lo que éste tendrá $k+1$ elementos. La línea (1) justo esto es lo que hará: `creaMinHeap` creará un minHeap (por lo general se usa un arreglo) de tamaño $k+1$ y se agregarán los primeros $k+1$ elementos de A en el árbol. Una vez que ya tengamos el `minHeap`, lo que haremos es subir al elemento más pequeño a la raíz del árbol (como se mencionó al inicio, ya tenemos garantizado que el elemento más pequeño de A se encuentra dentro del `minHeap`). Esto se realiza durante la línea (2), y las líneas $(3-4)$ simplemente inicializan contadores que nos ayudarán a saber cuándo terminar ciertas partes del algoritmo.

Ahora bien, comenzamos con la parte más interesante: como ya tenemos al elemento más chiquito en la raíz del árbol, entonces cambiámos al elemento que se encuentra en la i-ésima posición del arreglo original por el elemento que se encuentra en la raíz del `minHeap` y aumentamos nuestro contador. Luego, eliminamos al elemento más pequeño (y esto en el algoritmo de los `minHeap` implica que además de eliminar al elemento, actualiza al árbol para mantener en todos sus nodos la definición de `minHeap`, por lo que en particular tendremos al nuevo elemento más chico en la raiz del árbol). El contador j nos ayudará a ir agregando a los elementos del arreglo A que nos faltaron por incluir en el `minHeap`, es decir, los elementos que estaban después de la posición $k+1$. Entonces, si aún nos falta agregar elementos del arreglo A, aumentamos el contador j y agregamos al `minHeap` el elemento que

se encuentra en la entrada j-ésima del arreglo original (y al igual que cuando eliminamos, se debe actualizar el árbol para mantener la definición de `minHeap` en todos sus nodos). De igual manera, como cada elemento está a lo más k posiciones de su posición correcta, al ir agregando de uno en uno los elementos que nos faltan del arreglo original, entonces seguimos garantizando que el más pequeño se encuentra dentro del `minHeap`. Todo esto se realiza en las líneas $(6-12)$. Así, vamos a ir realizando este proceso hasta que agreguemos todos los elementos de A en el `minHeap` y los hallamos eliminado a todos (lo que implicaría que ya todos fueron acomodados en el arreglo). Así, en la línea (14) sólo regresamos al arreglo A ya que todas sus entradas fueron actualizadas durante el algoritmo.

Finalmente, como hemos visto en clase, crear un `minHeap` nos toma $O(k)$, donde k es el número de elementos en el árbol (en particular, $k+1$). Actualizar al mínimo nos toma la altura del árbol (en el peor caso) y como nuestra altura será $\log k$ (por el número de elementos) entonces la línea (2) nos toma $O(\log k)$. Obtener al elemento que se encuentra en la raíz nos toma tiempo constante. Además, eliminar al elemento mínimo y agregar un elemento al árbol nos toma igual $O(\log k)$ ya que ambas operaciones implican que se tiene que actualizar el árbol; pero ambas operaciones no las realizamos en la misma cantidad. Como sólo agregamos los elementos que nos faltaron por añadir al `minHeap`, entonces la operación `agregaElemento` se realiza $n-(k+1)$ veces, por lo que en total nos toma $O((n-(k+1))\log k)$; mientras que la operación `eliminaMinimo` la estamos realizando por cada elemento de A, lo que implica que nos tomará en total $O(n \log k)$.

Así pues, el algoritmo en total nos toma

$$O(k) + O(\log k) + O(n \log k) + O((n-(k+1))\log k) = O(n \log k)$$

7. An abs-sorted array is an array of numbers in which $|A[i] \leq A[j|$ whenever $i < j$. For example, the array $A = [-49, 75, 103, -147, 164, -197, -238, 314, 348, -422]$, though not sorted in the standard sence, is abs-sorted. Design and algorithm that takes an abs-sorted array A and a number k, and returns a pair of indices of elements in A that sum up to k. For example, if $k = 167$ your algorithm should output $(3, 7)$. Output $(-1, -1)$ if the is no such pair.

   Solución: El algoritmo propuesto es el siguiente

Algorithm 3 Obtener la pareja de índices del arreglo A que son iguales a k
encuentraPares(A, k):

```
1:  B ← sort(A)
2:  i ← 0
3:  x ← −1
4:  y ← −1
5:  while i ≤ B.length − 1 do
6:      s = busquedaBinaria(B, k − B[i])
7:      if s then
8:          x ← A.getIndice(A[i])
9:          y ← A.getIndice(A[s])
10:         return (x, y)
11:     end if
12: end while
13: return (x, y)
```

donde

Algorithm 4 Busca un elemento e en el arreglo A usando búsqueda binaria.
busquedaBinaria(A, e):

```
1:  L ← 0
2:  R ← A.length − 1
3:  while L ≤ R do
4:      mid = ⌊(L+R)/2⌋
5:      if A[mid] < e then
6:          L ← mid + 1
7:      else
8:          if A[mid] > e then
9:              R ← mid − 1
10:         else
11:             return mid
12:         end if
13:     end if
14: end while
15: return
```

Primero, veremos por qué funciona el algoritmo. Sabemos que nuestro arreglo A es un `abs-sorted`, por lo que va a estar ordenado, pero respecto a los valores absolutos de sus elementos. Para buscar la pareja de índice (i, j) cuya suma de sus elementos sea igual a k se podría pensar inicialmente en comparar todos los elementos con todos hasta encontrar los elementos que buscamos, pero esto sería bastante ineficiente pues sería de complejidad $O(n^2)$. La propuesta en esta ocasión es ordenar nuestro arreglo A y después utilizar búsqueda binaria para hallar a la pareja.

Notemos que cualquier suma es de la forma $a + b = c$, en particular, la suma que buscamos se ve de la forma $A[i] + A[j] = k$. Así que, haciendo un simple despeje, obtenemos

$$k - A[i] = A[j]$$

Entonces, una vez que tengamos nuestro arreglo A ordenado, si aplicamos a cada elemento $A[i]$ búsqueda binaria para buscar a $k - A[i]$ podemos obtener los índices que estamos buscando. En la línea (1) estamos ordenando nuestro arreglo A (en este caso, `sort` es el algoritmo de ordenamiento `heapSort`, el cual tiene complejidad en tiempo $O(n \log n)$ y espacio adicional $O(1)$). En las líneas $(2-4)$ simplemente definimos nuestro contador i y los índices x y y (éstos tienen valor de -1 ya que si el `while` termina, entonces no encontramos nuestra pareja y regresamos los índices pertinentes para representar esto). En las líneas $(6-10)$ tenemos lo interesante: definimos a s como el resultado de aplicar búsqueda binaria (escribí el algoritmo porque había que hacer una modificación al último `return`, pues si no encontramos al elemento deseado, no regresamos nada). Como se explicó antes, si hallamos el valor $k - B[i]$ entonces hemos encontrado al otro sumando que buscamos (ya tenemos el primero, que es $B[i]$). Con la condición de la línea (7) verificamos que s exista, es decir, que búsqueda binaria le haya regresado un valor. En caso de que esto ocurra, ya tenemos los dos números que sumados dan como resultado a k, pero los índices corresponden al arreglo ordenado, por lo que hay que buscar los índices que corresponden a esos elementos en A; y para esto están las líneas $(8-9)$. Una vez que hallamos los índices del arreglo original, entonces los regresamos en forma de tupla.

Si al aplicar búsqueda binaria a cada elemento $A[i]$ no hemos encontrado en B al elemento $k - B[i]$ entonces la tupla no existe.

Finalmente, mencionaremos la complejidad del algoritmo. Como mencionamos que `sort` será para nosotros `heapSort` entonces esto nos toma $O(n \log n)$ en tiempo. Sabemos que búsqueda binaria toma $O(\log n)$, pero como se aplica a cada uno de los elementos del arreglo, entonces estar buscando

a $k - B[i]$, en el peor caso, nos tomará $O(n \log n)$. Además, obtener el índice de los elementos i y s nos tomará, en el peor caso, $O(n)$ pues tendríamos que recorrer todo el arreglo. Por lo tanto, la complejidad del algoritmo será $O(n \log n) + O(n \log n) + O(n) + O(n) = O(n \log n)$.

8. **The Hogwarts Sorting Hat**

   Every year, upon their arrival at Hogwarts School of Witchcraft and Wizardry, new students are sorted into one of four houses (Gryffindor, Hufflepuff, Ravenclaw, or Slytherin) by the Hogwarts Sorting Hat. The student puts the Hat on their head, and the Hat tells the student which house they will join. This year, a failed experiment by Fred and George Weasley filled almost all of Hogwarts with sticky brown goo, mere moments before the annual Sorting. As a result, the Sorting had to take place in the basement hallways, where there was so little room to move that the students had to stand in a long line. After everyone learned what house they were in, the students tried to group together by house, but there was too little room in the hallway for more than one student to move at a time. Fortunately, the Sorting Hat took Algorithms many years ago, so it knew how to group the students as quickly as possible. What method did the Sorting Hat use? More formally, you are given an array of n items, where each item has one of four possible values, possibly with a pointer to some additional data. Design and analyze an algorithm that rearranges the items into four clusters in $O(n)$ time using only $O(1)$ extra space.

   Solución:

9. Pruebe que el segundo elemento más chico de una lista de n elementos distintos puede encontrarse con $n + \lceil \log n \rceil - 2$ comparaciones.

   Demostración. Sea T el árbol binario, en particular un *min-Heap*, que contiene a los n elementos de nuestra lista en sus hojas. Por lo discutido en clase, sabemos que para encontrar al elemento más pequeño necesitamos realizar $n - 1$ comparaciones, ya que al ir comparando los elementos desde las hojas, vamos formando nuestros nodos internos (éstos serán los elementos más pequeños de las comparaciones que se vayan haciendo), los cuales siempre son $n - 1$.

   Ahora bien, para encontrar al segundo elemento más pequeño debemos tener en cuenta una observación importante: como siempre vamos *subiendo* a los elementos más pequeños, eso quiere decir que el segundo elemento más pequeño ya fue comparado con la raíz del árbol T, así que sólo queda ubicar a todos los elementos que *perdieron* contra la raíz de T, actualizar el árbol y compararlos entre sí. Al ubicar estos elementos, obtendremos que hay uno de ellos (a lo más) en cada nivel del árbol, y como la altura del árbol es $\log_2 n$ entonces hacer esta última pasada al árbol nos tomará $\lceil \log_2 n \rceil - 1$ comparaciones (menos 1 porque ya no vamos a tomar en cuenta al elemento más pequeño). Por lo tanto, encontrar al segundo elemento más pequeño de una lista de n elementos distintos nos toma

   $$(n - 1) + (\lceil \log_2 n \rceil - 1) = n + \lceil \log_2 \rceil - 2$$

   comparaciones.

   □

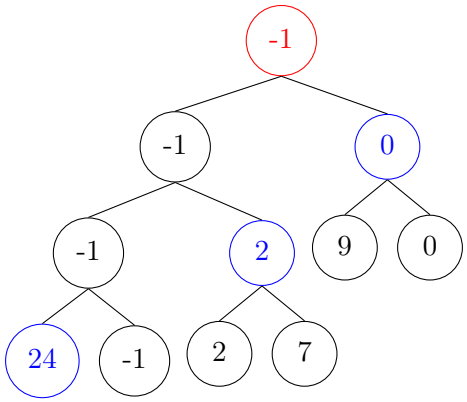

Figura 4: Ejemplo de la explicación con la lista $[24, -1, 2, 7, 9, 0]$. Los elementos en azul son aquellos que *perdieron* contra el elemento más pequeño.