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:

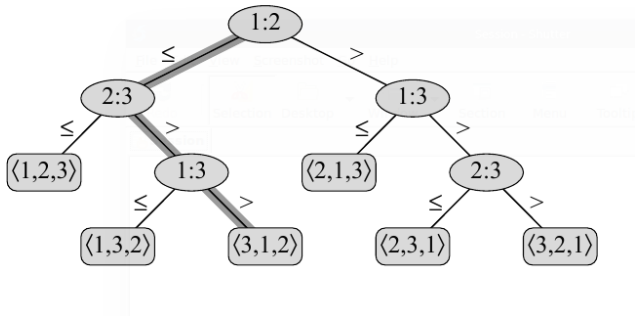

Figura 1: Árbol de decisión correspondiente a tres elementos (Cormen, pag. 192)

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

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

Es decir, en el peor de los casos tenemos que hacer $x\log_2(x) - x\Big|_1^n$ comparaciones. Por lo tanto, para ordenar una lista de 5 elementos son *necesarias* 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

```
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 $O(n+m)$.

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.

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

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

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 entero 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 $A_{[1\ldots n]}$ 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.

```
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)

Ahora, realizamos el análisis de complejidad: el ciclo en las líneas $(2-3)$ toman $O(k)$, el ciclo en las líneas $(4-5)$ toman $O(n)$, el ciclo de las líneas $(7-8)$ toman k y el ciclo en las líneas $(10-12)$ toman $O(n)$. Así, $O(k+n)$ es el tiempo total que toma este algoritmo. Entonces, si $k = O(n)$ eso quiere decir que $T(n) = (n)$; y por lo tanto el algoritmo será lineal. □

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 $O(n \log k)$.

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.

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.

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 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. 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.

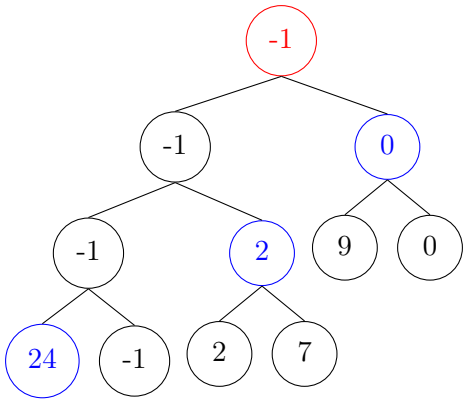

Figura 4: Ejemplo de la explicación con la lista $[24, -1, 2, 7, 9, 0]$. Los elementos en azul son aquellos que *perdieron* contra el elemento más pequeño.

□