Universidad Nacional Autónoma de México

Facultad de Ciencias

Análisis de Algoritmos | 7083

Tarea 2 : | Algoritmos de ordenamiento

Sosa Romo Juan Mario | 320051926

09/09/24

1. **Cada año, cuando los estudiantes llegan a Hogwarts, el sombrero seleccionador clasifica a los nuevos estudiantes en una de las cuatro casas Gryffindor, Hufflepuff, Ravenclaw o Slytherin. Un profesor pone el sombrero en la cabeza del estudiante y después de tiempo constante el sombrero le dice a qué casa se unirá. Este año, los gemelos Weasley llenaron casi todo Hogwarts con una pegajosa sustancia marrón, momentos antes de la selección anual. Como resultado, la selección tuvo que llevarse a cabo en los pasillos del sótano, donde había tan poco espacio para moverse que los estudiantes tenían que hacer una larga fila. Después de que todos supieran en qué casa estaban, los estudiantes trataron de agruparse por casa, pero había muy poco espacio en el pasillo para que más de un estudiante se moviera a la vez. Afortunadamente, el sombrero seleccionador tomó algoritmos hace muchos años, por lo que sabía cómo agrupar a los estudiantes en tiempo lineal a pesar del poco espacio ¿Qué algoritmo usó el sombrero seleccionador?.**

Tengo algunas ideas para esta solución, pero comencemos con la más sencilla.

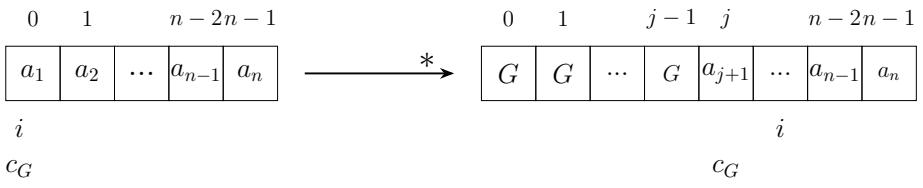
Simple

La idea mas sencilla que tengo es ir agrupando hasta el principio de la fila a los estudiantes de la misma casa.

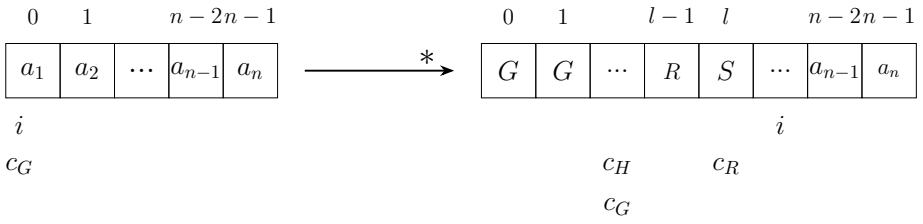
Es decir, tomamos nuestra fila $A = [a_1, a_2, \ldots, a_n]$ y aparte tenemos 2 indices, los podemos llamar c y i, donde c es el ultimo estudiante de la casa actual $+1$ y i es el estudiante que estamos evaluando; ademas, voy a denotar a c_k como el indice con el ultimo alumno de la casa k ; en este caso vamos a acomodar de la siguiente manera *G-H-R-S*, donde G es el grupo de Gryffindor, H es el grupo de Hufflepuff, R es el grupo de Ravenclaw y S es el grupo de Slytherin.

De esta manera, comenzamos con nuestra primer iteración, donde $c_G = 0$ y $i = 0$, checamos estudiante por estudiante, si el estudiante en la posición i es de la casa k en este caso Gryffindor y $i \neq c_k$, entonces lo intercambiamos con el estudiante en la posición c_G y aumentamos c_G e i en 1, si no, simplemente aumentamos i en 1. Esta iteración acaba cuando $i = n$.

Esto se ve algo asi:



Despues de n todos los estudiantes de la casa Gryffindor estan al principio de la fila, ahora hacemos lo mismo con las otras casas; aprovechando que c_G tiene el valor del indice del ultimo estudiante de la casa Gryffindor $+1$, entonces nuestras siguientes iteraciones solo pasaran por $n - c_G$ estudiantes, y asi sucesivamente, para Ravenclaw se podria ver algo asi:



Nota importante en este algoritmo es que una vez que un estudiante es acomodado en su lugar, no se vuelve a mover, y ademas una vez acomodemos las primeras 3 casas en la fila, la ultima casa se acomoda sola.

De manera general el algoritmo hace lo siguiente:

- Comenzamos con $c = 0$ y $i = 0$.
- Iteramos moviendo a i hasta que $i = n$, siempre que encontremos a alguien de la casa que estamos buscando lo intercambiamos a la posición c y aumentamos c e i en 1.
- Repetimos para las primeras 3 casas.

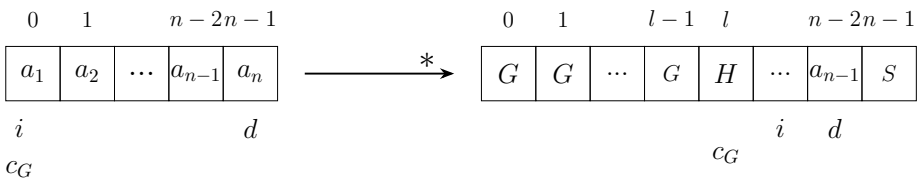
Esto claramente esta en n, pero vamos a ver que hace, en la primera casa movemos i por n elementos, en la segunda casa movemos i por $n - c_G$ elementos, en la tercera casa movemos i por $n - c_H$ elementos, podemos acotar este algoritmo con $3n$, por lo que es lineal.

Ahora solo voy a agregar la idea que me gusta un poquito mas pero es basicamente lo mismo.

Mejorada

La idea de este segundo algoritmo es la misma que el anterior, pero agregando un indcie, le podemos decir d y nos va a servir para ordenar al fondo mientras vamos ordenando.

Igual que en el anterior, vamos a ir moviendo a i por la fila, pero ahora, si encontramos a alguien de Slytherin, (porque acomodamos G-H-R-S) lo intercambiamos con el estudiante en la posición d si este mismo no es de Slytherin, si lo es, decrementamos a d y checamos otra vez hasta poder poner al Slytherin de la posicion i en la posición d y reducimos en uno a d para que este apunte a uno antes del primer Slytherin que hemos visto, se puede ver algo asi:



Entonces la idea es que vamos acomodando al principio a los Gryffindor y al final a los Slytherin, en el medio nos va a quedar Ravenclaw y Hufflepuff, pero solo tendremos que acomodar a los Hufflepuff en la segunda iteración para que queden todos.

Notemos que ahora no es verdad que i va a recorrer n elementos si no que acabamos cuando $i = d$, y en la segunda iteracion cuando solo queremos ordenar al frente a los Hufflepuff, i va a recorrer $n - c_G - c_S$ elementos.

En este algoritmo i no necesariamente recorre n pero con d si, sin embargo, en la segunda iteracion ya acomodamos a Gryffindor y Slytherin por lo que en el peor de los casos (que no hubiera alumnos de estas 2 casas) tendremos que recorrer n elementos, para acomodar a Hufflepuff, por lo que este algoritmo esta acotado por $2n$, por lo que es lineal.

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

   (a) **Encuentre un algoritmo que toma a lo mas $O(n \log\log n)$ tiempo para ordenar S.**

   Vamos a resolver esto con un árbol ordenado, veamoslo.

   árbol Balanceado Ordenado

   Ahora, sabemos que crear un árbol balanceado ordenado toma $O(n\ log\ n)$ tiempo, ademas $n\ log\ n \notin O(n \log\log n)$, por lo tanto habra que modificar un poco.

   La idea es cambiar los nodos del árbol por tuplas (x, y), donde x es el valor del nodo y y es la cantidad de veces que hemos visto el valor x. Viendolo asi, como tenemos que S solo tiene $O(\log n)$ valores distintos, entonces el árbol tendra $O(\log n)$ nodos.

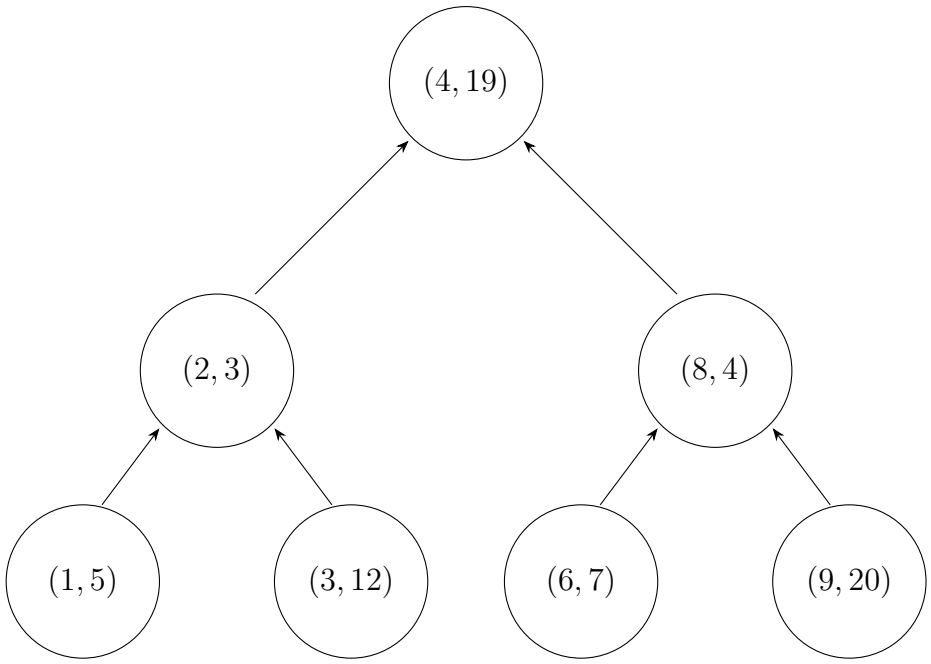
   Ademas, sabemos que la operación de insertar un nodo en un árbol balanceado ordenado toma $O(\log n)$ tiempo (con n nodos), por lo tanto, insertar un nodo en este árbol toma $O(\log\log n)$ tiempo.

   Como nota importante es que por cada valor en verdad tendremos que hacer 2 operaciones, una para buscar si el valor ya esta, si no esta insertar y si ya esta solo aumentar el contador, aun asi, la operación de buscar en estos árboles tiene la misma complejidad que insertar.

   Ya con esto en mente, como tenemos que ya sea insertar o al menos buscar y cambiar el contador por n elementos y sabemos cuanto tardaran estas operaciones, entonces el tiempo total que tomara ordenar S sera $O(n \log\log n)$.

   Ahora si quisieramos tener la lista ordenada otra vez, solo tendriamos que recorrer el árbol en orden y por cada nodo imprimir el valor x y veces, esto tarda $O(\log n)$ tiempo. (en ABB recuperar es $O(n)$ con n nodos)

   Al final quedaria algo asi:



Vemos por ejemplo que aqui nuestro árbol tiene 7 nodos con 7 valores distintos ordenados, pero si sumamos todos los valores de los contadores nos damos cuenta que $n = 70$, entonces es claro que podemos ordenar los 70 elementos usando pocas operaciones porque tiene muchos valores repetidos.

(b) **¿Por qué no viola la cota inferior de $O(n \log n)$ para el problema de ordenación.**

Explicación

Primero que nada, la cota se cumple siempre y cuando no sepamos nada de información adicional de los elementos a ordenar, pero ya hemos visto que si tenemos información adicional podemos hacerlo en menos tiempo, cuando vimos cosas como counting sort.

En este caso, lo que pasa es que no estamos realmente ordenando los n elementos, sino que estamos ordenando unicamente los valores distintos de los n elementos, o ordenando $O(\log n)$ elementos; aunque si tenemos que pasar por los n elementos para hacerlo y necesitamos verificar si ya estan o insertarlos por tanto el tiempo total es $O(n \log \log n)$.

3. **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* elemeneto en la unión de *A* y *B*. Por ejemplo, si $k = 1$, tu algoritmo debe regresar el 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.**

La idea es usar busqueda binaria para encontrar el k-ésimo elemento en la unión de A y B, con algunas consideraciones.

Busqueda Binaria Modificado

Primero que nada hay que notar que no podemos simplemente hacer BB con el completo,

esto porque si hacemos esto, vamos a tener que hacer $O(n)$ comparaciones, y $O(n) \notin \Theta(log\, n)$.

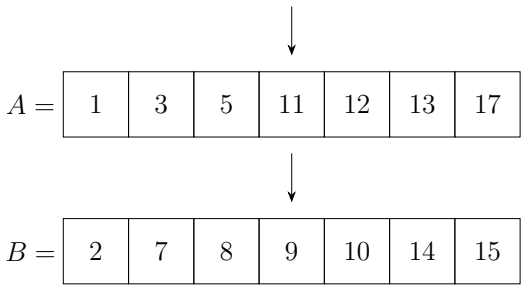
La idea es ir reduciendo el espacio de busqueda por fracciones de n para poder hacer de la complejidad de tiempo $\Theta(log\, n)$.

De manera general vamos a hacer lo siguiente:

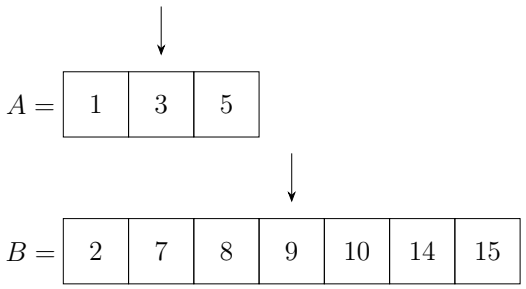
- Encontrar los puntos medios enteros de A y B llamemoslos m_A y m_B; en este algoritmo se me olvido que habia diferencia entre el indice y el elemento entonces cuando me refiero a un numero dentro del algoritmo es el elemento numero i del arreglo, no el indice.
- Si $m_A + m_B < k$, entonces sabemos que el k-esimo no esta aqui y de hecho es mayor, por lo que vamos a querer descartar a los menores:
  - Si $A[m_A] > B[m_B]$, entonces podemos descartar a los mas chicos osea $B[1, \ldots, m_B]$; ademas, como quitamos m_B elementos, tenemos que restar m_B a k.
  - Si $A[m_A] \leq B[m_B]$, entonces podemos descartar a los mas chicos osea $A[1, \ldots, m_A]$; ademas, como quitamos m_A elementos, tenemos que restar m_A a k.
- Si $m_A + m_B \geq k$, significa que el k-esimo elemento esta antes o en el punto medio, por lo que vamos a querer descartar a los mayores:
  - Si $A[m_A] > B[m_B]$, entonces podemos descartar a los mas grandes osea $A[m_A, \ldots, n]$; en este caso no restamos pues k se encuentra antes de los que quitamos.
  - Si $A[m_A] \leq B[m_B]$, entonces podemos descartar a los mas grandes osea $B[m_B, \ldots, n]$; igualmente no restamos.
- Repetimos el paso 1 a 3 hasta que hayamos descartado un arreglo de los 2, en cuyo caso el k^*-esimo elemento es el que queda.

Este algoritmo va descartando fracciones lineales de alguno de los 2 arreglos, es parecido a hacer una busqueda binaria en ambos arreglos pero no de manera simultanea, sino mas bien independiente, de ahi podemos justificar que este algoritmo tendra complejidad de tiempo $\Theta(log\, n)$.

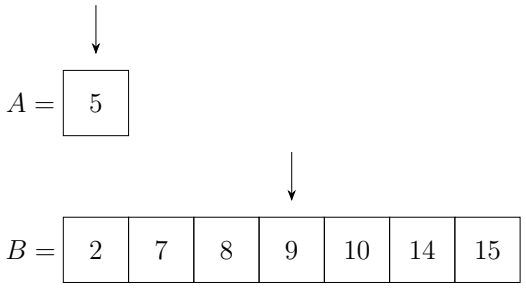
Pero veamos un ejemplo, buscaremos el 8-esimo:



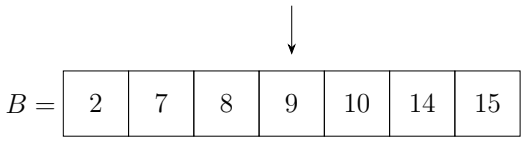
Aqui podemos ver que $m_A = 4$ y $m_B = 4$, entonces $m_A + m_B = 8 = k$, por lo que compmaramos elementos, como $A[m_A] = 11 > B[m_B] = 9$ entonces descartamos a los mas grandes de A, osea $A[11, 12, 13, 17]$, y no restamos a k.



Ahora tenemos que $m_A = 2$ y $m_B = 4$, entonces $m_A + m_B = 6 < 8$ por lo que comparamos elementos y como $A[m_A] = 3 < 9 = B[m_B]$ entonces descartamos a los mas chicos de A, osea $A[1, 3]$ y restamos a k, entonces $k = 8 - 2 = 6$.



Ahora tenemos que $m_A = 1$ y $m_B = 4$, entonces $m_A + m_B = 5 < 6$ por lo que comparamos elementos y como $A[m_A] = 5 < 9 = B[m_B]$ entonces descartamos a los mas chicos de A, osea $A[5]$ y restamos a k, entonces $k = 6 - 1 = 5$.



Como A ya no tiene elementos entonces el 5-esimo elemento es $B[5] = 10$. (como mencione se me fue la onda entre elementos e indices pero se entiende el algoritmo)

Este algoritmo va reduciendo el espacio de busqueda con fracciones lineales de alguno de los 2 arreglos, por lo que podemos justificar que este algoritmo tiene complejidad de tiempo $\Theta(log\, n)$ y de hecho se parece bastante a una busqueda binaria independiente en ambos arreglos.

4. **Sea A un arreglo de n númreos 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.**

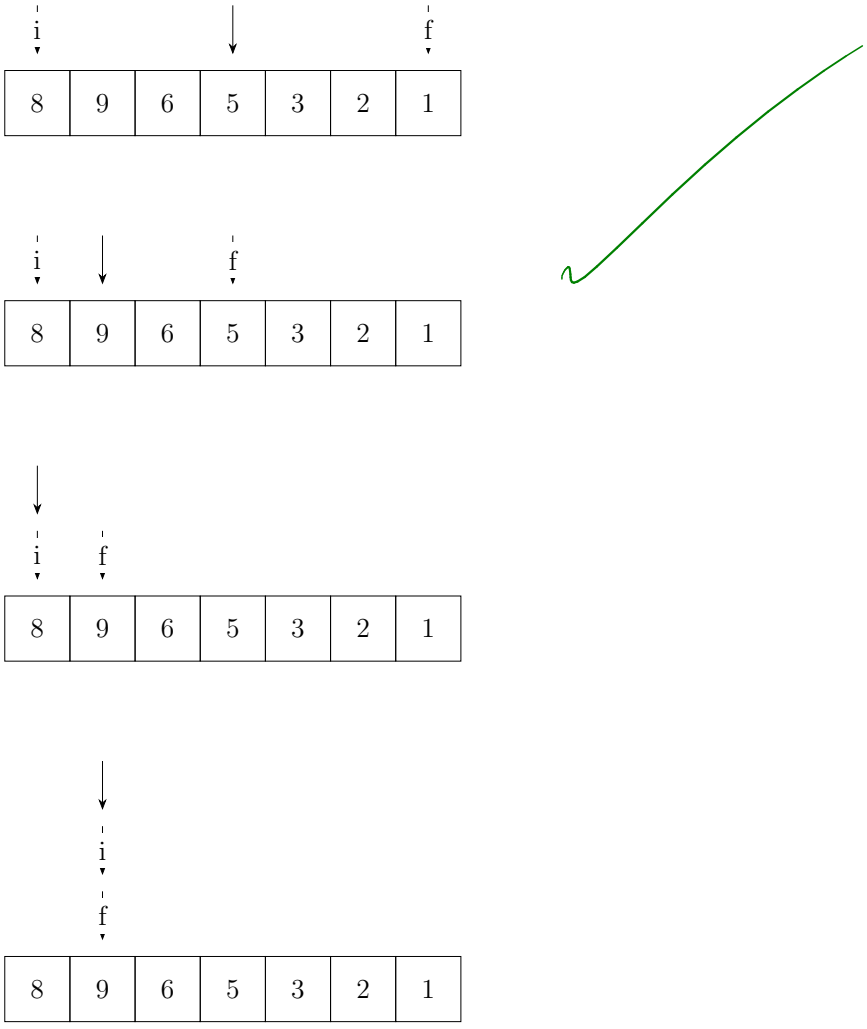
Para encontrar k, podemos hacer uso de la técnica de *Divide y Vencerás*, vamos a intentar que salga en tiempo logarítmico.

BB Modificada

Igualmente este es un problema clasico de BB, asi que vamos a modificarlo un poco:

- Comenzamos con 3 apuntadores uno al final y uno al inicio, y definimos $m = \frac{inicio+fin}{2}$.
- Checamos si $A[m] < A[m+1]$, si es asi, es porque estamos en la parte creciente, por lo que vamos a querer descartar a los menores, entonces movemos el inicio a $m+1$.
- Si no se cumple lo anterior, entonces estamos en la parte decreciente, por lo que vamos a querer descartar a los mayores, entonces movemos el fin a m.
- Repetimos el proceso hasta que $inicio \geq fin$, en ese caso, el valor de inicio es el valor de k.

El algoritmo basicamente usa la misma idea que la busqueda binaria, entonces su complejidad es $O(\log n)$. (siempre descartamos una fraccion lineal de los elementos)
Veamos como funciona:



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

La idea es usar el algoritmo anterior y en este paso solo queremos encontrar el valor de n. Para ello, vamos a usar lo siguiente:

Busqueda Exponencial Controlada:

La idea es parecida a una búsqueda binaria, pero en reversa, es decir, primero empezamos por un elemento (asumo que al menos tiene 3 porque si no, no tiene sentido) y vamos a ir creciendo el tamaño posible de nuestro n al doble por cada iteración, hasta que encontremos un valor que sea mayor a n, en ese momento, ya sabemos que nuestro n está entre el valor anterior y el actual, digamos que el anterior era 2^k y el actual es 2^{k+1}, entonces, hacemos una búsqueda binaria entre 2^k y 2^{k+1} para encontrar el valor de n.

Estoy asumiendo que se pueden usar cosas de programación, como ver si el posible valor de n es mayor que n, esto seria similar a intentar acceder al indice del arreglo y que regrese exception o null.

La complejidad de este algoritmo es de $O(\log m)$, ya que es una busqueda binaria donde m es $2^{k+1} - 2k$ (busca en este posible rango el valor de n), la busqueda exponencial tambien es del $O(\log n)$.

Después de encontrar el valor de n, se puede usar el algoritmo anterior para encontrar el valor de k y ya no me da tiempo de hacer figuritas bonitas.

5. **Rotar un arreglo significa hacer un corrimiento a la derecha de todos los elementos, excepto el último que pasará a la primera posición del arreglo. Por ejemplo** $A = [1, 3, 4, 5, 7, 10, 14, 15, 16, 19, 20, 25]$ **y después de una rotación** $A = [25, 1, 3, 4, 5, 7, 10, 14, 15, 16, 19, 20]$**. Dado un arreglo ordenado** $\boldsymbol{A}$ **de n números enteros distintos que ha sido rotado un número desconocido de veces. Diseña un algoritmo de** $O(log\, n)$ **tiempo que encuentre el** ***k-ésimo*** **elemento del arreglo. Por ejemplo,** $A = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]$ **y** $k = 5$ **la respuesta es 7.**

Este algoritmo igualmente va a ocupar BB modificada asi que aqui vamos otra vez: Busqueda Binaria Modificada:

Aqui vamos a tener 2 partes:

- Buscar el punto de rotación.
- Buscar el k-ésimo elemento.

La primera parte es clara, queremos buscar el elemento tal que $A[i] > A[i+1]$, en ese momento, ya sabemos que el punto de rotación está en $i+1$.

Teniendo el punto de rotación, podemos hacer una busqueda binaria en el rango $[0, i]$ y $[i+1, n-1]$ para encontrar el k-ésimo elemento.

Asi que buscamos el punto de rotación como ya dijimos con BB, asi que tomamos los indices $l = 0$ y $r = n - 1$ y vamos a hacer lo siguiente:

- Mientras $l < r$:
- Definimos $m = \frac{l+r}{2}$. (parte entera)
- Si $A[m] > A[r]$, entonces $l = m + 1$.

- Si no, $r = m$.
- Cuando $l \geq r$, el punto de rotación es l.

Una vez tenemos este punto de rotación tenemos 2 subarreglos ordenados, (tambien lo puedes pensar como que los pegas y tienes un arreglo ordenado) asi que podemos hacer busqueda binaria, tenemos el subarreglo desde el inicio del arreglo hasta el punto de rotación y el subarreglo desde el punto de rotación hasta el final del arreglo.

Entonces, si k es mayor que $n - i$ con i el punto de rotación, entonces k está en el primer subarreglo, si no, está en el segundo subarreglo.

En el ejemplo de arriba $A = [15, 16, 19, 20, 25, 1, 3, 4, 5, 7, 10, 14]$ y $k = 5$, el punto de rotación esta en 5, entonces $n - i = 12 - 5 = 7$, entonces k está en el segundo subarreglo (es decir el arreglo desde el punto de rotación hasta el final del arreglo) o lo que es igual el subarreglo $A[i, n]$. Esto tiene sentido pues si k fuera mas grande que este rango no cabria dentro de este subarreglo.

Ahora solo hacemos BB con el subarreglo en el que este k y listo. Es claro que esto esta en $O(log\, n)$, pues literalmente hacemos una busqueda binaria, conseguimos subarreglos ordenados y hacemos busqueda binaria sobre uno de ellos. /

0/1

6. **Eres un joven cientifico quien acaba de recibir un nuevo trabajo en un gran equipo de 100 personas (tu eres la persona 101). Un amigo tuyo en quien confias te dijo que tienes mas colegas honestos que metirosos; eso es todo lo que te dijo. Un mentiroso es una persona que puede decir una verdad o una mentira, mientras que una persona honesta es alguien que siempre dice la verdad. Claro, a ti te gustaría saber quienes son los mentirosos y quienes son honestos, asi que decides empezar una investigación. Haces una serie de preguntas a tus colegas, Ya que no quieres aparecer sospechoso, decides solo hacer preguntas del tipo "¿Y una persona honesta?" y claro, quieres hacer las menos preguntas posibles. ¿Puedes distinguir a todos tus colegas honestos?. ¿Cúal es la mínima cantidad de preguntas que tienes que preguntar en el peor caso? Puedes asumir que tus colegas se conocen muy bien entre ellos, lo suficiente para decir si otra persona es un mentiroso o no. (Hint: Agrupa a las personas en pares (X, Y) y pregunta a X si Y es mentiroso y después a Y preguntale si X es mentiroso o a los 2 si el otro es honest; Analiza las 4 posibles respuestas. Una vez encuentres a una persona honesta, puedes usarlo para encontrar a todos los otros. Como desafio, ¿puedes resolver el problema con menos de 280 preguntas?.)**

Generaliza la estrategía de arriba para mostrar que dadas n personas tal que menos de la mitad son metirosos, puedes separar a los mentirosos de los honestos en $\Theta(n)$ preguntas.

Advierto que la solución a este problema es muy larga y tediosa, ademas de que el algoritmo que pense no es para nada intuitivo y es muy dificil de explicar, pero hare mi mejor esfuerzo.

Notas importantes:

- Una vez que encontramos a una persona honesta, podemos usarla para encontrar a todas las demas.
- Hay al menos 51 personas honestas. (a lo mas 49 mentirosos)
- Vamos a denotar a las personas honestas con 1 y a los mentirosos con 0.

- Tenemos la operación (X, Y) que nos dice si X piensa que Y es honesto.
- Ademas tenemos la operación $[X, Y] \Rightarrow x, y$ que hace la operacion (X, Y) y (Y, X), y nos regresa como el primer numero lo que piensa X de Y y como segundo numero lo que piensa Y de X.

Ahora, como vamos a estar usando las operaciones de arriba, vamos a ver los casos con una tablas primero:

X	Y	(X, Y)	(Y, X)	$[X, Y]$
1	1	1	1	1,1
1	0	0	1/0	$0, 1/0$
0	1	1/0	0	$1/0, 0$
0	0	1/0	1/0	$1/0, 1/0$

Ahora con esta tabla podemos ver algunas cosas interesantes; si tenemos algun 0 tras aplicar la operación $[X, Y]$ entonces sabemos que al menos uno de los dos es mentiroso, y podemos descartar esa pareja. Por otro lado si tenemos dos 1, no sabemos nada.

O eso pareciera, pero no es verdad, ya que si tenemos dos 1, entonces sabemos que son del mismo tipo; ahora si, vamos a ver el algoritmo.

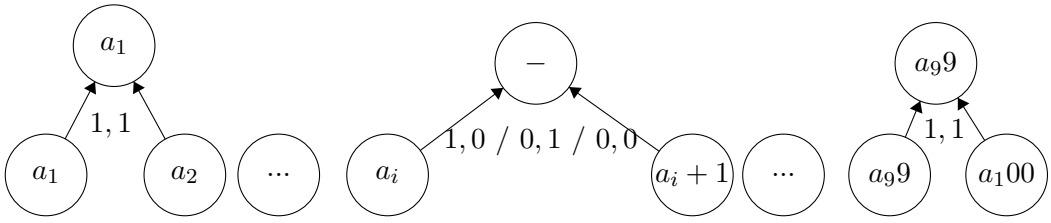
Algoritmo:

Como ya mencione, la idea principal del algoritmo es intentar encontrar a una persona honesta con la menor cantidad de preguntas posibles, y una vez que la encontramos, podemos usarla para encontrar a todas las demas.

Ademas tenemos que podemos eliminar grupos de personas que contengan a un mentiroso (vamos a tener mucho cuidado haciendo esto por lo que se va a ver a continuación). Y en el proceso, vamos a obtener grupos cada vez mas grandes de personas del mismo tipo (si es que no las eliminamos).

Voy a hacer uso de diagramas para explicarlo pero vamos a ir creando un arbolito, comenzamos con una base de 100 personas, algo asi:

Ahora, justo como describimos, les vamos a aplicar la operación $[a_k, a_{k+1}]$, por parejas de izquierda a dercha y vamos a ir eliminando a las parejas que contengan a un mentiroso, aquellas que regresen dos unos podemos pasar al izquierdo a un nuevo nodo; quizas algo asi:



¿Qué pasa si tienes solo mentirosos emparejados y solo honestos emparejados excepto una pareja? Pasa que solo descartas una pareja. Si repites el proceso es probable que solo descartes 1 pareja a la vez.
Esto pasa al no tener cuidado en emparejar y como consecuencia la complejidad se vuelve $O(n^2)$. Es este el caso :(

Probablemente se eliminen un monton de parejas y aun en las que queden sube nada mas un miembro al nodo de arriba; sin embargo, vamos a ver el peor caso.

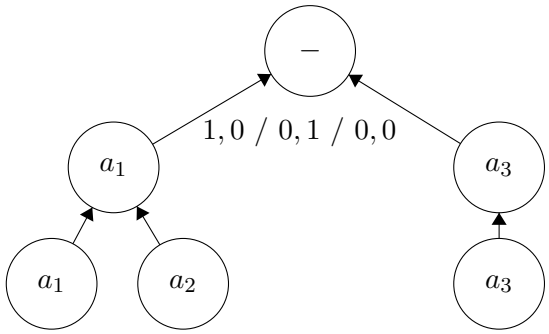
En este caso todos los mentirosos estan al principio del arbol y despues todos los honestos en orden, ademas, los mentirosos saben mentir bien y siempre dicen que el otro es honesto; de esta manera los primeros 24 pares seran mentirosos que fingen ser honestos un par mixto, y el resto honestos, parece que solo eliminamos una pareja y nos quedan aun 49 parejas o nodos.

Ahora vamos a repetir lo mismo pero con los nodos de arriba pero vamos a hacerlo de derecha a izquierda, esto es importante porque una pareja no se va a comparar con nadie (ya que tenemos un numero impar de nodos).

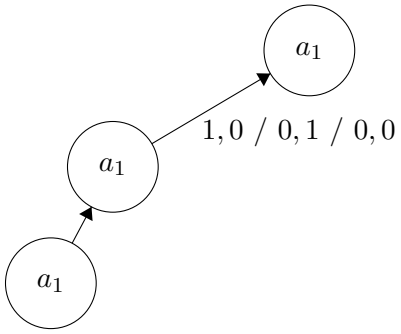
Ahora, en el peor de los casos, todos los mentirosos estan acomodados de tal manera que es el honesto el que sobra y no se compara con nadie, por lo que nos quedan 24 parejas de las cuales 12 son honestas y 12 mentirosas, y 1 honesto que no se compara con nadie.

De nuevo vamos a cambiar el orden, y vamos a hacerlo de izquierda a derecha, (esto ya garantiza que el honesto que no se comparo con nadie se compare con alguien), la idea es seguir el algoritmo hasta llegar a un nodo raiz que tenga un solo nodo, y ahi ya sabemos quien es honesto.

Pero vamos a ver algunos casos que van a suceder, al eliminar, ya no es tan sencillo como quitar todos los que tengan un mentiroso, ya que si tenemos un arbol mas grande que otro podemos perder mas honestos que mentirosos y no garantizamos encontrar el honesto, vamos a ilusrarlo:



En este caso, digamos que a_1, a_2 son honestos y a_3 es mentiroso, pero si eliminamos la "pareja" a_1, a_3 entonces perdemos a 2 honestos y solo 1 mentiroso; entonces para eliminar en vez de esto quitamos la pareja a_2, a_3 y subimos a a_1 al nodo de arriba.

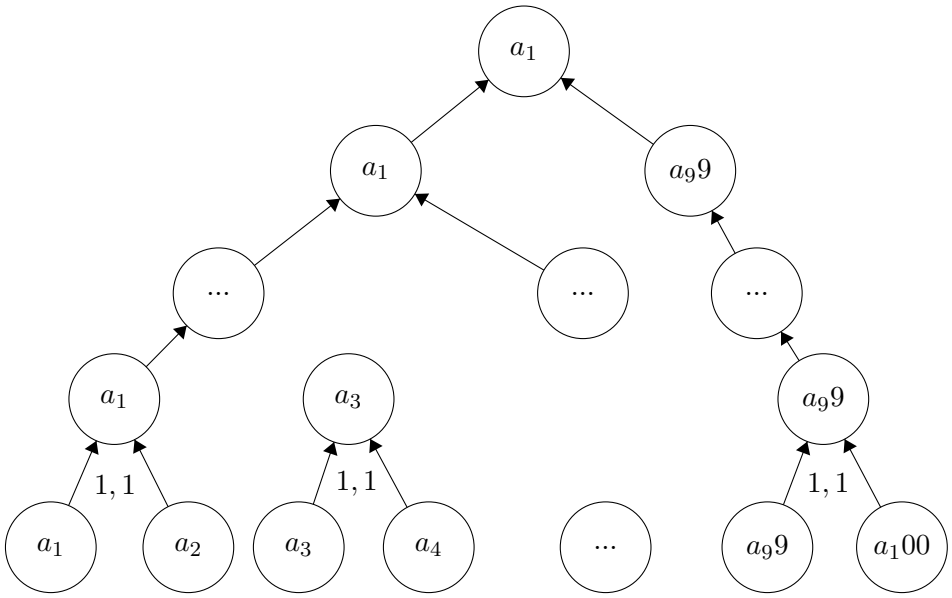


De manera mas general si tenemos 2 nodos que con la operacion $[X, Y]$ nos regresan al menos un cero pero difieren en tamaño, entonces eliminamos una subrama del mas grande y el arbol mas pequeño. (Creo que aqui se ve mejor porque necesitamos ir cambiando de orden porque si nada mas hacemos de izquierda a derecha o de derecha a izquierda, entonces vamos a tener un arbol muy delgadito y muy alto que al final va a ser molesto de tratar, si camiamos el sentido garantizamos tamaño de este a lo largo del proceso).

Otra cosa importante de esto es que si los arboles tienen el mismo tamaño, entonces si podemos eliminar a ambos pues estamos quitando la misma cantidad de honestos que de mentirosos.

Pareceria que solo eliminamos unas cuantas parejas pero en realidad estamos reduciendo el espacio de busqueda maso menos a la mitad en cada paso (o de manera logaritmica) (checa que si son del mismo tipo solo suibmos a uno de los 2 y este sera el "representante" de su grupo); tras 100 preguntas nos deshicimos de al menos un par pero ademas vamos a preguntarle solo a 49 de los que quedan, y en el segundo paso aun sin quitar a nadie vamos a preguntarle a 24, y asi sucesivamente.

Al final el arbol podria verse algo asi:



Al final sabemos que la raiz sera un honesto pues solo suben uno de cada par del mismo tipo, y tras la primera ronda de preguntas, hay 50 honestos y 48 mentirosos (en el peor caso) aparte solo elimianmos la misma cantidad de honestos que mentirosos con lo que no podemos haber eliminado a todos los honestos sin eliminar a todos los falsos.

Otra cosa importante es que el mismo proceso nos va a dejar el nodo raiz que es honesto y los que esten por debajo tambien lo seran, pues son de su mismo tipo. (en el peor de los casos el raiz no tiene a nadie abajo)

Ahora si vamos a contar las preguntas, como el algoritmo es sumamente complicado, solo voy a asumir que no eliminamos a nadie y hacemos un arbol completo pero esto es una cota superior, en realidad el algoritmo es mucho mas eficiente.

En la primera ronda de preguntas hacemos 100 preguntas, en la segunda 50, en la tercera

25, en la cuarta 12, en la quinta 6, en la sexta 3, en la septima 2 (el grupo que te sobro del 25) y en la octava 1. En total 199 preguntas.

Al final ya con este honesto (asumiendo que no tiene a nadie abajo para hacerlo peor) hara falta preguntarle sobre las personas restantes, en el peor caso 98 preguntas, en total 297 preguntas.

Una manera de hacer aun mas optimo en el caso esperado es no eliminar los arboles si no que solo mantenerlos por ahi y asi tenemos grupos de honestos y mentirosos, para usar con nuestro honesto god pero ya mucho.

Y ya sueltenme porfavor xd. La vida de un estudiante a veces puede ser difícil xD.

Para generalizar el algoritmo, vamos a inetntar ponerlo en puntos:

- Comienza tomando a n personas y ponlas hasta abajo de tu arbol.
- Aplica la operación $[X, Y]$ a todas las parejas de izquierda a derecha (o de derecha a izquierda cambiando de orden cada vez) y elimina a las parejas que contengan a un mentiroso aplicando que si son arboles del mismo tamaño, entonces eliminamos a ambos, si no, eliminamos una subrama del mas grande y todo el arbol mas pequeño garantizando quitar la misma cantidad de honestos que de mentirosos.
- Sube al izquierdo de los que regresen dos unos al nodo de arriba.
- Regresa al paso 2 hasta que obtengas un solo nodo.
- El nodo raiz sera honesto y los que esten por debajo tambien lo seran, usalo para encontrar a los demas.

Vemos que durante el arbol haremos $n + \frac{n}{2} + \frac{n}{4} + \cdots + 1$ preguntas, algo asi como $2n - 1$ preguntas, (serie geometrica, la ultima no la preguntamos pues ya es el nodo unico) y al final hacemos $n - 2$ preguntas (la ultima es redundante y uno es con el que pregunatmos), en total $3n - 3$ preguntas.

Obviamente $3n - 3 \in \Theta(n)$, pues si elegimos $c_1 = 1$ y $c_2 = 3$ entonces $c_1 n \leq 3n - 3 \leq c_2 n$ para todo $n \geq 2$ y hemos terminado.

Gracias diosito santo por terminar este problema, y gracias a ti por leerlo, espero que te haya gustado.

7. **Sea $A[1, \ldots, n]$ un arreglo de números reales. Diseña un algoritmo para realizar cualquier secuencia de las siguientes operaciones:**

- **Add(i, j)**: suma el valor y al *i-ésimo* número.
- **PartialSum(i)**, regresa la suma de los primeros i números, es decir $PartialSum(i) = A[1] + \cdots + A[i]$

No hay inserciones ni eliminaciones; el único cambio es a los valores de los números. Cada operación debe tomar $O(log\, n)$ pasos. puedes utilizar un arreglo adicional de tamaño n como espacio de trabajo.

Para este algoritmo vamos a usar una EDD bastante famosa llamada *Binary Indexed Tree* o BIT. Esta estructura de datos nos permite hacer operaciones de suma y consulta de suma acumulada en $O(log\, n)$, lo cual es perfecto para este problema; la desventaja de esto es que utilizaremos n espacio adicional.

Binary Indexed Tree:

Sabemos que se pueden representar un arbol binario con un arreglo, y la implementación de BIT especifica nos la vamos a saltar, lo vamos a pensar unicamente en su forma de arbol. (es interesante que esta EDD en se suele implentar con n+1 en espacio, pero bueno como no nos vamos a meter con como se implementa podemos hacerlo con n)

Lo primero es que los nodos de este arbol nos guardan la suma de los elementos de un rango; especificamente el nodo i guarda la suma de los elementos $A[i - 2^r + 1], \ldots, A[i]$, donde r es la posición del bit menos significativo de i.

No importa tanto pero lo que importa es que esto permite que tenga una suma parcial pero no toda la suma para evitar que actualizar un nodo sea $O(n)$, y que la suma de un rango sea $O(log\, n)$.

Add(i,y)

La idea aqui es que no solo actualicemos el nodo i sino que actualicemos todos los nodos que contienen a i en su rango. Para esto vamos a sumar y a $A[i]$ y a todos los nodos que contienen a i en su rango, que, por la definición de los nodos, son los nodos que tienen un bit menos significativo en i y esto garantiza que se cumpla en $O(log\, n)$.

Esto puede parecer una perdida pues usualmente actualizar un valor en un arreglo nos cuesta $O(1)$ y aqui estamos "gastando" $O(log\, n)$ pero es escencial si queremos lograr consulta en el mismo tiempo; la verdad es que los especificos son confusos para nosotros asi que recomiendo checar la documentación de BIT para entenderlo mejor.

PartialSum(i)

Aqui es donde hace sentido usar el BIT, pues la suma de un rango es $O(log\, n)$, y la idea es que si queremos la suma de los primeros i elementos, entonces vamos a sumar los nodos que contienen a i en su rango, en si en implementación se hace $i- = i\&(-i)$ que es un rango mas grande que incluye al rango actual; por ejemplo, si queremos la suma de los primeros 6 elementos comenzamos en el indice 6, luego checamos su representación binaria que es 110, y le restamos el bit menos significativo que es 10, y nos queda 100 que es 4, entonces sumamos el nodo 4 y checamos su representación binaria que es 100, le restamos el bit menos significativo que es 100, y nos queda 0, como es el nodo 0 terminamos.

Para la demostración de que esto es $O(log\, n)$ se puede hacer por inducción estructurada, pero me voy a recargar en la popularidad de la EDD para no hacerlo. (y que apenas les entiendo xd)

8. **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, (moviendonos siempre en la dirección de las manecillas del reloj) comenzamos a remover los ocupantes de las sillas como sigue: Primero eliminiamos 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 generarla *(n,m)-permutación de Josephus*.**

Para este algoritmo vamos a usar solamente una lista:

Algoritmo usando una lista circular enlazada:

Como mencione, vamos a unicamente usar una lista, $L = [1, 2, 3, \ldots, n]$; adicionalmente, vamos a tener un indice p que va a ser la posicion de inicio donde contamos, este empieza en 1.

El algoritmo elimina de manera secuencial en cada iteracion:

- Comenzando desde la persona actual, la que vamos a eliminar es la que esta en la posicion $p + m - 1$.
- Vamos a simular una lista circular usando el modulo n; entonces la forma de calcular la posicion de la persona a eliminar es: $p = (p + m - 1)\%n_i$ con n_i es el tamaño de la lista en la iteracion i. (usar modulo nos garantiza no recorrer la lista una cantidad insana de veces y estar en una posicion valida de la lista).
- Eliminamos a la persona de la lista, registramos su resultado y actualizamos el tamaño de la lista ademas de la posicion de inicio.

Este proceso anterior se repite hasta que la liste se vacie completamente. Al final, la lista de resultados va a ser la permutacion de Josephus que buscamos.
Este algoritmo es lineal en el sentido que en cada iteracion eliminamos una persona de la lista (una lista ordenada y ligada se puede eliminar basicamente en constante). Por lo tanto, el algoritmo es lineal en el tamaño de la lista.

(b) **Supongamos que m no es constante. Describa un algoritmo de complejidad $O(n \log n)$ para encontrar la *(n,m)-permutación de Josephus*. Hint: Construya un árbol de búsqueda de rangos.**

Para este algoritmo vamos a usar un árbol de búsqueda de rangos. Este árbol es un árbol de segmentos que nos permite hacer consultas y actualizaciones en un rango de manera eficiente.

Algoritmo usando un árbol de búsqueda de rangos:

Comenzamos creando el arbol de búsqueda de rangos con n hojas, inicialmente todos estan activos (podemos representarlo como 1 e inactivo como 0, de esta forma los internos solo contaran los activos). Este arbol nos permite hacer consultas eficientes del tipo "Dado un rango $[l, r]$, ¿cuantos elementos hay en el rango?". Tiene hasta sus hojas los elementos individuales y los internos tienen una suma de los elementos en sus intervalos.

Cada eliminacion y update toma $O(log\, n)$, pero solo crearlo toma $O(n \log n)$; vamos a encontrar el m-ésimo elemento y actualizando en $O(log\, n)$ respectivamente, entonces el algoritmo es $O(n \log n)$.

de manera general el algoritmo se ve de la siguiente manera:

- Creamos el arbol de búsqueda de rangos con n hojas y vamos creando internos sumando pares hasta llegar a una raiz.
- Vamos a hacer una consulta en el arbol para encontrar el indice de la persona a eliminar, p, usando la operacion que vimos en el inciso anterior (nos garantiza que este dentro de la cantidad de personas activas); ahora checamos si este es mayor a lo que tiene el subarbol izquierdo de la raiz, si lo es, entonces vamos a buscar en el subarbol derecho, si no, vamos a buscar en el subarbol izquierdo.

- Repetimos esta busqueda hasta llegar a una hoja, que va a ser la persona a eliminar.
- Actualizamos el arbol, eliminando a la persona y actualizando los valores de los internos.
- Repetimos hasta que la raiz tenga valor 0.

Vemos que las consultas a lo mas bajaran la altura del arbol osea $O(log\, n)$, y las actualizaciones tambien, por lo tanto el algoritmo es $O(n\, log\, n)$, pues elimina una persona en cada iteracion y ademas crea el arbol de busqueda de rangos.

Perdon por no incluir tanto detalle y formalidad en estos ultimos ejercicios pero la tarea me ha consumido mucho tiempo.