Universidad Nacional Autónoma de México
Facultad de Ciencias
Análisis de Algoritmos | 7083
Tarea 1 : | Algoritmos de ordenamiento
Sosa Romo Juan Mario hola | 320051926
24/08/24

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

Demostrando suficiencia:

Primero vamos a resaltar que para un arreglo de 5 elementos hay 5! diferentes maneras de arreglar nuestro arreglo, por tanto sabemos que tras las comparaciones que hagamos deben satisfacer todas estas posibles combinaciones.

Ahora, tomemos un árbol binario de k niveles donde cada nodo es una comparación diferente, sabemos que el árbol en el nivel k tendrá 2^k nodos; además sabemos que estas hojas son resultado de la serie de comparaciones que vienen de sus niveles superiores.

Es así que si tomamos nuestros 5 elementos y comenzamos a comparar usando un árbol de decisiones, sabemos que tenemos la siguiente desigualdad para satisfacer todas las ordenaciones:

$$\begin{aligned} 5! \leq 2^k = \\ log_2(5!) \leq log_2(2^k) = \\ log_2(120) \leq k\ log_2(2) \approx \\ 6.9068 \leq k \end{aligned}$$

Y como k es un numero natural pues es una cantidad de niveles sabemos que $7 \leq k$ es así que al menos necesitamos 7 comparaciones para satisfacer todas las posibles ordenaciones.

Demostrando necesidad:

Inicialmente en esta demostración intente usar Merge Sort pero es algo que no recomiendo pues este algoritmo en verdad puede tomar 8 comparaciones o 6 dependiendo del arreglo, por ejemplo el arreglo $[1, 5, 2, 3, 4]$ le toma al algoritmo 8 comparaciones.

Es entonces que en realidad vamos a utilizar un algoritmo hecho específicamente para n=5, lo cual es un código muy feo pero bueno aquí la idea:

- **Paso 1:** Comparar y organizar los primeros 2 pares (a,b y c,d) aquí usamos 2 comparaciones.
- **Paso 2:** Comparar los elementos mas grandes de ambos arreglos, de manera que el arreglo con el elemento mas grande de los 4 este después algo como (a,b,c,d) si 'd' es mas grande que 'b', esto toma 1 comparación.

- **Paso 3:** Este paso es un poquito mas truculento, tenemos el orden total de [a,b,d] con algo de información de 'c', ahora la idea es que queremos saber donde poner e pero tiene que usar a lo mas 2 comparaciones, entonces sabemos que empezar desde el principio es mala idea; vamos a tomar nuestro conjunto ordenado (a,b,d) y vamos a comparar a 'e' con 'b', si es mas grande entonces compararemos con 'd' y dependiendo de la respuesta lo ponemos al frente (a,b,d,e) o atrás (a,b,e,d) lo mismo si es mas chico que 'b', comparamos con 'a' y lo ponemos al frente (a,e,b,d) o atrás (e,a,b,d).

- **Paso 4:** Finalmente nos quedan 2 comparaciones para meter a 'c', afortunadamente, ya tenemos información relevante, sabemos por el paso uno que $c < d$, por tanto en el peor de los casos tenemos que comparar 2 veces como en el caso anterior, igualmente tenemos que comparar entonces con los 3 primeros elementos del arreglo, digamos ([a,e,b],d) comparamos con el de en medio de los tres digamos 'e', si es mas chico comparamos con 'a' y lo ponemos adelante (a,c,e,b,d) o atrás (c,a,e,b,d) o en el caso de ser mas grande que 'e' comparamos con el de adelante digamos 'b' y lo ponemos adelante (a,e,b,c,d) o atrás (a,e,c,b,d).

Una vez terminado este algoritmo tendremos el arreglo ordenado y en el peor de los casos usamos 7 comparaciones.

2. **Diseña un algoritmo eficiente que tome un arreglo A de n enteros positivos y obtenga el número de parejas de índices invertidos, donde una pareja de índices (i, j) son invertidos si $i < j$ y $A[i] > A[j]$. Por ejemplo si $A = [1, 4, 7, 2, 8]$ entonces los índices $(1, 3)$ son invertidos porque $1 < 3$ y $4 = A[i] > A[3] = 2$.**

Para este problema la primer idea seria utilizar fuerza bruta, tomando para cada elemento los elementos mas chicos a su derecha pero esto nos va a usar $O(n^2)$ en tiempo.

Algoritmo eficiente:

Entonces vamos a "buscar el árbol", el profesor enseno un método usando Arboles Binarios Indexados pero creo que es mas sencillo lo siguiente; entonces en base a lo que sabemos, vamos a ver si podemos utilizar un algoritmo de Merge Sort ligeramente alterada. La idea es la siguiente, empezar por separar el arreglo de n elementos en n arreglos de 1 elemento, ahora sabemos que estos subarreglos por definición no tendrán ninguna pareja invertida. Ahora vamos a juntar arreglos en parejas para obtener arreglos de tamaño 2, para esto puede o no tener un solo par alternado después de verificar esto solo ordenamos el arreglo.

Entonces llegamos al caso interesante, vamos a mezclar arreglos de tamaño 2 y si n es impar habrá un solo arreglo de tamaño 3, esto no afecta; comenzamos igual que Merge Sort, comparando mínimos, lo interesante aquí es que si encontramos la pareja invertida (i,j) con 'i' el índice que recorre el arreglo de la izquierda y 'j' el índice que recorre el arreglo de la derecha, entonces no solo contamos esta pareja si no también las parejas tipo (k,j) donde $k > i$ y $k < n_{izquierdo}$; por ejemplo con los arreglos (1,3,5,10) y (2,6,8,9) durante el proceso encontramos la pareja (3,2) entonces contamos esta mas las parejas (5,2) y (10,2) que sabemos van a ser pares invertidos (no necesitamos compararlo solo lo contamos) pues el arreglo izquierdo esta ordenado; además de esto solo ordenamos con Merge Sort y seguimos el proceso.

Una parte interesante es que cuando tenemos los casos [ ] con $[b_1, b_2, ...]$ o el caso $[a_1, a_2, a_3...]$ con [ ] a la hora de estar mezclando subarreglos por definición no habrá parejas invertidas y estas ya las contamos con el paso anterior, por lo que podemos pasar juntar el arreglo no vació con el final de nuestro arreglo resultante.

Al terminar este proceso sabemos que habremos contado pares invertidos y además habremos ordenado todo el arreglo en $O(n\ log(n))$ pues usamos Merge Sort y durante los pasos no hicimos realmente mas que hacer sumas tipo $n_{izquierdo} - i$ además del algoritmo normal, aunque hay que resaltar que si ocupa $O(n)$ en espacio y la constante es relativamente alta.

3. **Diseña un algoritmo que encuentre el segundo elemento mas pequeño y el segundo elemento mas grande entre un conjunto de n elementos usando a lo mas $\frac{3n}{2} + o(n)$ comparaciones.**

De nuevo vemos que usar fuerza bruta no es una alternativa viable pues tendríamos que ir elemento a elemento preguntándonos si es mas grande que el segundo mas grande, después si lo es entonces preguntar si es mas grande que el máximo y si la primera era no entonces preguntar si es mas pequeño que el segundo mínimo y finalmente si es el mínimo total; esto toma algo así como 3n comparaciones en el peor de los casos.

Usando Heaps:

La idea de este algoritmo es utilizar espacio y otras operaciones que no sean comparaciones para bajar la cantidad de estas, así que siguiendo este principio comenzamos por dividir el arreglo en parejas, esto toma 0 comparaciones; un punto importante es que las comparaciones que utilizamos en los heaps para obtener la raíz son $n - 1$,

Ahora con esta base vamos a crear 2 estructuras una en donde van a competir para ver quien es el mínimo y otra para ver quien es el máximo, pero notemos que en realidad solo vamos a realizar una comparación para llevar un elemento a cualquiera de estas en el primer turno, es así que comparamos por parejas cada pareja usando $n/2$ comparaciones y conseguimos los primeros "ganadores" y "perdedores", notemos que cada heap tiene $n/2$ elementos en su penúltimo nivel y si el arreglo original era impar nos sobra un elemento pero no lo comparamos aun con nadie.

Entonces ahora podemos razonar que la cantidad de comparaciones para terminar de subir los elementos en ambos arboles es $n/2 - 1$, la primera es viendo que tenemos un submax heap y submin heap, es decir dentro de nuestros heaps, por ejemplo, en nuestro max heap, sucede que en el punto anterior ya subimos el primer nivel y ahora tenemos un subárbol con $n/2$ elementos que pueden ser máximo (lo mismo pasa con el mínimo) y como establecí al principio de la solución un max heap o un min heap toma $n-1$ comparaciones para elegir su ganador, entonces cuando $n_{subarbol} = n/2 \rightarrow$ tardamos $2(n/2 - 1) = n - 2$ comparaciones en este paso.
Ahora si eligen no creer o no les hace sentido vamos a razonarlo de otra manera, con series; sabemos que cada uno de nuestros arboles tiene su nivel de hasta abajo lleno de los elementos iniciales y su siguiente nivel lleno de "ganadores" o en otro caso "perdedores" por el paso anterior, vamos a ver el caso de los "ganadores" y el de los "perdedores" sera análogo pero pasando el menor; comenzando con los "ganadores" comparamos una vez cada pareja de nuestro nivel y eso tarda $\frac{n}{2/2} = n/4$ comparaciones, el siguiente nivel

por lógica tardara $n/8$ comparaciones y dado un n muy grande podemos ver a donde convergerá esta suma usando series:

$$\frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \ldots = \sum_{k=0}^{\infty} \frac{n}{4} \left(\frac{1}{2}\right)^k$$

Esta es una serie geométrica "infinita" (no nos vamos a complicar la vida) y converge si $|r| < 1$, en este caso $|r| = |\frac{1}{2}|$ y entonces sabemos que la formula converge a:

$$S = \frac{\frac{n}{4}}{1 - \frac{1}{2}} = \frac{\frac{n}{4}}{\frac{1}{2}} = \frac{2\ n}{4} = \frac{n}{2}$$

Podrían preguntar porque es que aquí no restamos el -1 y es que la serie no para mientras que nosotros sabemos que podemos parar cuando $n/2^k > 1$ puesto que ya no podemos seguir comparando.

Entonces, ahora si convencidos espero, podemos concluir que tras el primer paso, construir ambos arboles nos toma algo de la forma $n - 2$ mas las $n/2$ comparaciones del primer paso llevamos $\frac{3n}{2} - 2$ comparaciones y solo tenemos 2 arboles.

Pero ojo porque hasta ahora solo consideramos arreglos pares, si el arreglo fuera de tamaño impar, el paso uno deja ese elemento solo, basta con comparar este elemento con el máximo del árbol de máximos y en el peor de los casos también con el mínimo del árbol de mínimos, así tomando 2 comparaciones y llevándonos a $\frac{3n}{2}$ comparaciones hasta ahora.

Notemos que, $n \notin o(n)$ pero $log(n) \in o(n)$; entonces vamos a usar este hecho y el que ya tengamos 2 arbolitos ya hechos para encontrar el 2ndo máximo y 2ndo mínimo; vamos a ver igualmente el caso con el máximo pero el mínimo es simétrico pero con el menor de. Sabemos que el segundo máximo tiene que ser alguno de los que perdió directamente contra el máximo (porque si no perdió contra otro elemento y este seria potencialmente el segundo) por tanto hay que buscar quienes fueron sus rivales, en el peor de los casos, el árbol era impar entonces podemos comenzar nuestra búsqueda por el ultimo elemento, si no es, entonces comparamos por nivel, sabemos que un árbol binario completo como lo es el max heap, tiene una altura de $log(n)$ y sabemos que como es binario y el máximo subió, tuvo que ganarle a un contrincante por nivel menos su propio nivel esto seria $log(n) - 1$ comparaciones pero recordando el caso del arreglo impar nos regresa esa comparación.

Finalmente recordando que son 2 arboles tenemos que sumarle 2 veces esta ultima cantidad de comparaciones para llegar a que en el peor de los casos, para encontrar el segundo mayor y el segundo menor necesitamos $\frac{3n}{2} + 2log(n)$ comparaciones.

4. **Imaginemos que tenemos un barril lleno de sake y n contenedores de madera y n contenedores de vidrio, todos de diferentes formas y tamaños. Además, para cada contenedor de madera hay un único contenedor de vidrio que le cabe la misma cantidad de sake y viceversa. Nuestro problema es encontrar para cada contenedor de madera su contenedor de vidrio que le cabe la misma cantidad de sake. Es importante mencionar que no se pueden comparar directamente dos contenedores de madera o dos de vidrio. Lo que si se puede hacer es lo siguiente: tomas un contenedor de madera y uno de vidrio y sumerges uno de los dos en el barril de sake hasta llenarlo; luego viertes**

el sake en el otro contenedor sin derramar el sake. Esto, te dirá si el contenedor de madera o el de vidrio pueden tener mas sake o si les cabe lo mismo. Suponga que tal comparación requiere una unidad de tiempo. No es necesario contar el tiempo que nos lleva sumergir un contenedor al barril.

(a) **Describe un algoritmo que use $\Theta(n^2)$ comparaciones para resolver el problema.**

La primer idea es tomar cada contenedor de madera, llenarlo de sake y probar pasarlo hacia cada uno de los de vidrio contenedor 1 hasta el n de vidrio en el peor de los casos (obviamente tenemos que rellenarlo cada que no sean del mismo tamaño y no lo menciona pero supongo que también vaciar los contenedores de vidrio), esto en el peor de los casos nos toma n unidades de tiempo, ahora nos quedan n-1 contenedores de ambos tipos y repetimos, esto también esta en el orden de n unidades de tiempo y vamos a repetir para cada contenedor de madera, entonces la complejidad es $n + (n-1) + (n-2) \cdots + 1 = \frac{n(n+1)}{2} \in O(n^2)$ además podemos demostrar que tiene una cota superior si multiplicamos por ejemplo por 2 entonces $c_2 = 2$ y $\frac{n^2+x}{2} \leq 2 * n^2$ y esto demuestra la cota superior $\forall n > 1$, ahora tambien si tomamos $c_1 = .1$ tenemos que $.1n^2 \leq \frac{n^2+x}{2}$ $\forall n > 1$ entonces demostrando que este algoritmo $\in \Theta(n^2)$.

(b) **Suponga por obvias razones, que solo nos interesa encontrar los contenedores que les cabe mas sake. Pruebe que esto puede hacerse con a lo mas 2n-2 comparaciones.**

Ahora, tomamos nuestro contenedor de madera, y comenzamos a rellenar y vertir en cada uno de los de vidrio, la idea es, buscar uno que le quepa mas que el que elegimos, en el peor de los casos comparamos nuestro botecito de madera con todos los y no fue hasta el ultimo que encontramos uno de vidrio que es mas grande (y no fue igual), afortunadamente ya solo quedan n-1 contenedores de madera así que comenzamos a comparar desde el segundo, en el peor de los casos, comparamos hasta llegar al penúltimo y este también es mas chico que nuestro bote de vidrio, es decir que ya sin comparar podemos saber que el ultimo contenedor de madera sera el que es de igual tamaño al que tenemos de vidrio y este par es el mas grande.

Para resumir el algoritmo, tomamos un contenedor de madera, el que sea, comparamos con cada uno de los de vidrio hasta llegar a uno mas grande (esto descarta los que comparaste antes) o acabemos con todos y solo a uno le cupo lo mismo (en cuyo caso acabaste), a la hora de intercambiar eliminamos los de madera que ya hayamos usado o comparado, de manera que seguimos con los que aun pueden ser mas grandes si encontramos uno mas grande otra vez alternamos y ahora usamos ese grande de madera para comparar con los de vidrio; notemos que recorremos el arreglo de los de vidrio hasta que no, tomamos el bote en el índice i+1 y recorremos los de madera desde el ultimo que usamos.

Como ya mencionamos esto en el peor de los casos recorre todos los de vidrio y después se pone a recorrer desde el segundo de madera hasta el penúltimo de madera (el primero lo usamos para comparar con los n de vidrio y el penúltimo no tiene caso comparar) así usando 2n-2 comparaciones para encontrar la pareja.

(c) **¿Puedes mejorar el numero de comparaciones esperadas del inciso a? Explica.**

Técnicamente podríamos usar algo como counting sort, fijándonos en vez del contenido que hay en los contenedores mas bien el contenido que queda en el barril de sake al llenar nuestro

barril pero vamos a obviar esa respuesta.

Supongo que como se trata de un problema de ordenación, la cota mínima es $O(nlog(n))$ pero en este caso sin información previa, no veo forma de hacer algo como merge sort o un árbol, en cualquier paso no podemos descartar parejas pues no podemos saber el orden de nuestros contenedores de madera con respecto a ellos mismos o los vidrio, por tanto para ordenarlos respecto a ellos mismos y al mismo tiempo respecto al otro grupo se necesitaran $O(n^2)$ comparaciones.

5. **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. ¿Que pasa si m es de $O(n^2)$? ¿Se puede realizar en tiempo lineal? ¿Por que?**

Vamos a comenzar probando lo primero, que ordenar n elementos con valores entre 1 y m con m $\in O(n) \in O(n)$, para ello usaremos **Counting Sort**.

Demostración:

Primero que nada hay que destacar que counting sort es un algoritmo que ordena pero no es basado en comparaciones, por tanto no esta acotado inferiormente por $O(n\ log(n))$; la idea de este algoritmo simplemente es contar las apariciones de cada uno de los elementos del arreglo original y escribirlos en un arreglo auxiliar de tamaño m, la cosa importante es que, al final, para ordenar el arreglo hace falta recorrer todo el arreglo de tamaño m, por tanto es importante que m no sea muy grande, veámoslo mas a detalle.

Comenzamos creando un arreglo auxiliar $B = [b_0, \ldots, b_{m-1}]$ (hacemos m-1 porque el arreglo inicia en el índice 0 pero el problema nos da números que empiezan en el 1, vamos a mapear estos elementos hacia abajo) entonces como ya dijimos va a recorrer el arreglo digamos $A = [a_0, \ldots, a_n]$ y por cada entrada $a_i = j$ tenemos que $B[j-1] += 1$ o dicho de otra forma vamos contando uno por uno en el arreglo auxiliar la cantidad de veces que sale cada elemento (importante destacar que $j - 1 \leq m - 1$ porque j sale de A y A solo tiene valores hasta m). Es claro que esto toma tiempo lineal pues recorremos el arreglo de n elementos y además sumamos 1, n veces en el arreglo B (insertar en arreglos se hace en tiempo constante); entonces podemos decir que hasta ahora crear el arreglo B y contar la cantidad de elementos nos ha tomado algo así como $n + n = 2n$ esto independientemente de la forma de m y n.

Pero aun no tenemos el arreglo ordenado, solo tenemos un arreglo $B = [b_0, \ldots, b_{m-1}]$, entonces para conseguir el arreglo ordenado vamos a sobrescribir A usando B, recorremos B y para cada $b_i = k$ ponemos k veces el elemento $i + 1$ en el arreglo A; esta vez solo recorrimos un arreglo B y reescribimos n elementos de regreso en el arreglo A, esto por tanto toma algo así como $m + n$ (reescribir elementos en A es constante).

Es así que nuestro algoritmo usando counting sort, toma algo del estilo $2n+m+n = 3n+m$ (solo si contamos cada mini operación que hagamos como sumar o reescribir en un arreglo pero podría simplificar a $O(n + m)$) entonces es claro que dependemos de la forma de m para determinar la complejidad, en este caso, como $m = O(n) = c * n$ entonces nuestra complejidad es de $(3n + c * n) \in O(n)$.

Ahora toca preguntarnos, ¿Que pasa si m es de $O(n^2)$? ¿Se puede realizar en tiempo lineal? ¿Por que?, afortunadamente podemos usar la mayor parte del desarrollo anterior:

¿Que pasa si $m = O(n^2)$?

Bueno, retomando nuestro procedimiento anterior, sabíamos que el algoritmo propuesto tenia una complejidad como $3n + m$ pero ahora tenemos que $m = O(n^2) = c * n^2$ por lo que tenemos que nuestro procedimiento tomara $3n + c * n^2 \notin O(n)$ por tanto al menos usando solo counting sort no va a salir.

Entonces en este caso vamos a usar **radix sort**; algo importante es que radix sort va a usar nuestro counting sort, pero como lo describimos no es "estable" es decir no ordena los elementos por como aparecieron, esto es crucial para radix sort y la única diferencia para arreglar esto es utilizar otro arreglo auxiliar para ordenar digamos un arreglo C de manera que no sobrescribimos A si no que escribimos en C el arreglo ordenado en el orden que aparece en A.

En cualquier caso, radix sort funciona utilizando como subrutina este counting sort estable;la idea es, empezando por los dígitos menos significativos de cada numero n aplicar counting sort para ordenarlos según ese primer dígito, después, es solo seguir por cada dígito hasta llegar al mas significativo; como sabemos que cada dígito que ordenemos va del 0 al 9 entonces tenemos un $k = O(n)$ y entonces cada uno de estos counting sort estable por dígito tendrá una complejidad aproximada de $n + k = n + c * n \in O(n)$ y digamos que tomamos la cantidad de dígitos del numero m y lo llamamos d (m tiene la máxima cantidad de dígitos), entonces este algoritmo tomara para ordenar algo del estilo $O(d(n + k))$, porque realizamos d veces el counting sort, (el demostrar que el algoritmo radix sort ordena se puede ver en el Cormen segunda edición paginas 172 y 173) como ya mencionamos esta complejidad se puede ver también como $O(d(n))$ porque los counting sort tienen un $k \in O(n)$.

Ahora para ver que es lineal es suficiente con ver que d no sea algo lineal; recordemos que d no es mas que la cantidad de dígitos que tiene m, entonces vamos a proceder usando una formula medio mágica:

$$Numero\ de\ digitos\ = \lfloor log\ _b(m) \rfloor + 1$$

con b la base del numero m (en este caso b es 10) y la función piso redondea el numero al entero abajo mas cercano. (la demostración de que funciona se ve o en calculo o teoría de números pero en este caso trust me bro). Sustituyendo nuestros valores tenemos una complejidad algo asi como de $O((\lfloor log\ _{10}(m) \rfloor + 1)(n))$ esto, quitando la función piso, viendo que el 1 no nos mueve mucho y viendo que la base del logaritmo es constante tenemos una complejidad mas como $O(n\ log(m))$, además, sabemos que $m = O(n^2) = c * n^2$ entonces nuestra complejidad es $O(n\ log(c * n^2)) = O(n * 2 * log(n)) = O(n\ log(n))$ de cualquier manera vemos que no nos da tiempo lineal y hasta ahora.

Pero vamos a probar cambio de base del logaritmo (cambio de base es una operación que se puede hacer en tiempo constante pues no depende del tamaño de n):

$$log_b(m) = \frac{log_n(m)}{log_n(b)}$$

Entonces podemos reescribir nuestra complejidad como:

$$O(n\ log\ _{10}(m)) \rightarrow O\left(n\ \frac{log_n(m)}{log_n(10)}\right)$$

$$= O\left(n\ \frac{log_n(n^2)}{log_n(10)}\right)$$

$$= O\left(n\ 2\frac{1}{log_n(10)}\right)$$

$$= O\left(2n\right) = O(n)$$

Esto nos lleva a que efectivamente podemos ordenar en tiempo lineal. (tomar en cuenta que el logaritmo de un numero constante es constante y por eso se puede eliminar en la notación asintótica, igualmente con el c que multiplicaba a n^2 en m).

6. **Se dice que un arreglo A[1,...,n] es k-ordenado si este puede ser dividido en k bloques cada uno de tamaño $\frac{n}{k}$ aproximadamente, tal que todos los elementos en cada bloque son mas grandes que el bloque anterior y mas pequeños que los elementos del bloque siguiente. Los elementos en cada bloque podrían no estar ordenados. Por ejemplo, el siguiente arreglo es 4-ordenado:**

$$1, 2, 4, 3 \,|\, 7, 6, 5 \,|\, 10, 11, 9, 12 \,|\, 15, 13, 16, 14$$

(a) **Describe un algoritmo que k-ordene un arreglo de tamaño n en tiempo $O(n\,logk)$.**

Para este problema vamos a utilizar una versión modificada de **Quick sort** pero además vamos a utilizar la versión que toma su pivote con la mediana en tiempo O(n) para asegurar la complejidad en el peor de los casos.

Quick sort

Lo primero que nos interesa es que dividamos el arreglo en k bloques, pero como estamos usando quick sort modificado, lo primero es encontrar la mediana en $O(n)$, una vez encontrada esta mediana tenemos un buen pivote, podemos comenzar a dividir, en este paso vamos a tener 2 fracciones de n, probablemente no estén cerca de ser $n/2$ pero si sabemos que serán algo como n/c_1 y n/c_2.

En el siguiente paso, de nuevo buscamos la mediana en cada fracción en tiempo $O(n)$ y obtenemos de nuevo una fracción de n (asegurado por buscar la mediana); esto lo vamos a repetir hasta que el árbol llegue a arreglos de tamaño n/k como lo estamos dividiendo en fracciones de n entonces la altura del árbol binario implícito va a ser logarítmica, ahora vamos a checar cual es la altura de dicho árbol, en cada paso van a ser fracciones diferentes pero como queremos hacernos el calculo mas fácil en promedio podemos decir que se van a dividir en una fracción $1/c$ con c constante (en verdad c cambia en cada paso pero su valor es poco relevante). Entonces la altura del árbol la

vamos a calcular de la siguiente manera, comenzamos con n elementos y luego vamos partiendo por nuestro c hasta que lleguemos a k grupos de tamaño n/k:

$$\begin{aligned} n, \frac{n}{c}, \frac{n}{c^2}, \ldots, \frac{n}{c^m} = \frac{n}{k} \rightarrow kn &= c^m * n \\ k &= c^m \\ log_c\ k &= log_c\ c^m \\ log_c\ k &= m \end{aligned}$$

Notemos que los valores de C en realidad no alteran mas que la base del logaritmo y además m representaba el nivel de profundidad del árbol, es decir tras m niveles del árbol que dividía en fracciones de n tendremos k grupos de tamaño aproximado n/k.

Entonces, sabemos que para cada nivel del árbol, buscaremos la mediana de las medianas que se puede hacer en $O(n)$ además de que vamos a recorrer todo el arreglo que también esta en $O(n)$; simplificando cada nivel hace una cantidad lineal de operaciones; después vimos que vamos a estar dividiendo (gracias a la mediana de las medianas) a n en una fracción de si misma por cada nivel y además dividiremos hasta que lleguemos a arreglos de tamaño n/k de manera que tendremos $log_C(\ k)$ niveles, multiplicando estas complejidades tenemos que $O(n(log_C(\ k))) = O(n\ log(k))$, aunque hay que aclarar que las constantes ocultas probablemente son muy grandes especialmente estar calculando mediana de medianas en todos los pasos.

(b) **Prueba que cualquier algoritmo basado en comparaciones k-ordena un arreglo en $\Omega(n\ logk)$ comparaciones en el peor caso.**

Primero que nada, con el anterior demostramos que existe un algoritmo que puede realizar la tarea en $O(nlog(k))$ podemos verlo como que demostramos necesidad. Ahora necesitamos saber si teóricamente hay un algoritmo que pueda hacer menos de estas operaciones.

Con arboles

Para este razonamiento vamos a usar algo similar al visto en el ejercicio 1, es decir, ver cuantas posibilidades hay en total, para esto vamos a utilizar el coeficiente combinatorio:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Osea que vamos a tomar n elementos y agarrar de k en k, esto nos da todas las posibles maneras en las que podríamos hacer esto, entonces de nuevo, un árbol de k niveles en donde cada nivel m tendrá 2^m nodos y donde cada hoja es resultado de la serie de

comparaciones anteriores, tenemos lo siguiente:

$$\frac{n!}{k!(n-k)!} \leq 2^m$$

$$log_2\left(\frac{n!}{k!(n-k)!}\right) \leq log_2 2^m$$

$$log_2\left(\frac{n!}{k!(n-k)!}\right) \leq m$$

$$log_2(n!) - log_2(k!(n-k)!) \leq m$$

$$O(nlog(n)) - (log_2(k!) + log_2((n-k)!)) \leq m$$

$$O(nlog(n)) - (O(klog(k)) + O((n-k)log(n-k)))) \leq m$$

De aquí probablemente me doy cuenta de quizás mi acercamiento no fue el mejor pero la idea es que el árbol de comparaciones tiene una altura mayor o igual que $O(nlog(k))$.

7. **Considera el siguiente algoritmo para ordenar:**

```
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) **Prueba que STUPIDSORT realmente arregla la entrada.**

Lo vamos a probar por inducción sobre el tamaño del arreglo:

Casos base

- $A = []$: No entra a ninguno de los 2 casos pero al final esta ordenado.
- $A = [a_1]$: Igualmente no entra a ninguno de los 2 casos pero al final esta ordenado.
- $A = [a_1, a_2]$**:** Ahora tenemos 2 casos:

  - **Caso 1:** $A[0] \leq A[1]$ entonces no entra a ninguno de los 2 casos pero al final ya esta arreglado.
  - **Caso 2:** $A[0] > A[1]$ entonces intercambiamos estos 2 elementos y ya esta ordenado.

Hipótesis de Inducción

Suponemos que para un arreglo $A = [a_0, \ldots, a_k]$ con $k < n$ el algoritmo de StupidSort visto ordena el arreglo.

Paso inductivo

Sea el arreglo $A = [a_0, \ldots, a_n]$ veamos si lo ordena; comenzamos y vemos que no entra al caso uno pues $n > 2$ entonces ahora $m = \lceil 2n/3 \rceil$ ahora hace llamadas recursivas con el algoritmo sobre $A = [a_0, \ldots, a_{m-1}]$ entonces esto por H.I nos ordena el arreglo desde a_0 hasta a_{m-1} pero puede ser que en el $n/3$ que no considero haya elementos no ordenados totalmente, así que hace otra llamada al algoritmo ahora ordenando básicamente desde el el primer tercio hasta el elemento, ahora esto cae en la H.I y esta parte esta parcialmente ordenada, pero ojo porque es probable que hayamos movido cosas que hagan que el primer tercio se desacomodo, entonces hacemos una tercera llamada en el primer tercio para reacomodar todo e igualmente cae en H.I. por tanto esta ordenado.

(b) **¿El algoritmo seguiría ordenando de manera correcta si reemplazamos $m = \lceil 2n/3 \rceil$ con $m = \lfloor 2n/3 \rfloor$? Justifica tu respuesta.**

No jala, propongo un contraejemplo:

$$A = [4, 2, 1, 3]$$

Algoritmo

Iniciamos con m=2, dividimos en subarreglo [4,2], primera llamada recursiva al arreglo 1: entramos en el caso base pues n=2 y $A[0] > A[1]$ entonces intercambiamos [2,4]. y salimos de la primera llamada, ahora vamos a considerar el arreglo 2 que toma [1,3] y no entra a ninguna pues ya esta arreglado, ahora finalmente regresa al arreglo [2,4] y en este caso no cambio nada respecto a la primera vuelta así que acabamos con el arreglo: [2,4,1,3] que claramente no esta ordenado; lo que sucede es que en el algoritmo original había intersecciones entre lo que ordenábamos de manera que lo que movíamos en el paso intermedio lo podíamos recuperar con el ultimo paso, en este caso vemos que el paso intermedio no comparte elementos con ninguno de los otros 2 pasos y por tanto no puede haber intercambio de elementos entre estos.

(c) **Demuestra que el numero de "swaps" hechos por STUPIDSORT es a lo mas $\binom{n}{2}$.**

Lo importante para demostrar esto y por falta de tiempo es que el algoritmo solo mete parejas de números una vez y es que notemos que si mete una pareja digamos (i,j) eso significa que $i < j$ o al revés, en cualquier caso los swapea una sola vez y después de esto no puede darse el caso de que vuelva a meter a la pareja (i,j) pues significaría que $j < i$ que no hay forma que suceda; entonces si consideramos $\binom{n}{2}$ es decir tomar de n números parejas de 2 en 2, ya estamos considerando todos los casos posibles de parejas lo cual sucede en el peor de los casos.

8. **Imagine que usted es un guardia de seguridad en un estadio de baseball con capacidad para n personas, y su trabajo consiste en dar una gorra a cada persona de la fila, con la condición que cada una la use de manera adecuada. Sin embargo, para algunas personas el ponérsela bien, consiste en dejar la visera hacia enfrente y para otras hacia atrás. Usted no comparte ninguno de los 2 gustos, pero todos los asistentes deben tener la visera del mismo lado al ingresar. Suponiendo que puede pedir a la persona de la fila en la posición i que se gire la gorra, dicha persona lo hará de inmediato, incluso puede pedir que las personas que están entre la posición i y la posición j con $i \leq j \leq n$ se giren la gorra, acatan la solicitud sin preguntar y cada una de las solicitudes requieren**

una unidad de tiempo.

(el problema no especifica si darles las gorras incrementa la complejidad o si ver a las personas ocupa tiempo)

(a) **Suponiendo que puede cambiar de posición a las personas ¿Se puede resolver el problema con a lo mas una orden? Explique.**

Si, asumiendo que el mover de posición no cuenta como una orden, seria poner a todos los que tengan de una forma de un lado y los que lo tengan de la otra del lado contrario, así podríamos dar la orden de tomar uno de los 2 lados y pedirles a todos hasta el ultimo de esa forma que se cambien la gorra hacia el lado opuesto y tendríamos a todos de manera uniforme, (no se si entendí bien esta pregunta xd).

(b) **Diseñe un algoritmo de tiempo lineal tal que todos los asistentes tengan la visera del mismo lado al ingresar suponiendo que puede cambiar de posición a las personas.**

No se si entendí bien la pregunta pero podríamos elegir que todos van a estar como el primero que veamos, así recorremos la lista de n elementos y si alguno esta de la forma contraria al primero sencillamente le pedimos que se la voltee, entonces sera un algoritmo lineal pues recorre la lista una vez y a lo mas hace n-1 ordenes; alternativamente podemos usar el algoritmo del punto anterior que ocuparia solo una orden y no se especifica si cambiar de posicion ocupa algo.

(c) **Suponiendo que no puede cambiar de posición a las personas. Diseñe un algoritmo de tiempo lineal tal que todos los asistentes tengan la visera del mismo lado al ingresar pero garantice el mínimo numero de cambios.**

En este caso primero vamos a contar la cantidad que lo tiene de cada forma, (de nuevo no se si contamos esto como algo pero sube la complejidad implementándolo) esto ocupa 0 ordenes o cambios, una vez contado esto solo hay que recorrer la lista y pedirles a los que tuvieron la forma menos común, que se giren la gorra.