Tarea 1

1) Usaremos un arbol de decisiones, como revisamos en clase sabemos que el numero de hojas es al menos el numero de posibles soluciones de ordenamiento, ya que tenemos 5 elementos entonces hay $5! = 120$ posibles soluciones, lo cual nos dice que si encontramos la profundidad del arbol binario de 120 hojas ya terminamos, para lo cual usamos que $2^k \geq$ # hojas (con k, la profundidad), entonces $2^k \geq 120$, lo cual nos da que la profundidad k es 7, por lo tanto necesitamos 7 comparaciones

Ahora veamos un algoritmo para ordenar la lista de 5 elementos en 7 comparaciones, para esto tendremos que la entrada sera la lista con los elementos a, b, c, d y e

primero vamos a comparar a y b (1 comparacion) y los acomodamos de menor a mayor, digamos que quedan a, b, luego hacemos esto mismo con c y d (2 comparaciones), obteniendo a, b y c, d despues compararemos los dos mayores de las parejas (3 comparaciones), aqui son b y d, si d es mayor nos quedamos con a, b, d pero si b es mayor nos quedamos con c, d, b , en el siguiente paso (usaremos a, b, d) vamos a buscar el lugar de e , para esto comparamos e con b (4 comparaciones) , si e es mayor comparamos con d , de otra forma con a (5 comparaciones) y lo ponemos en su lugar , para el ultimo paso vamos a buscar el lugar de c , ya que c es menor que d , entonces lo insertamos entre los elementos a, b y e , por lo tanto lo comparamos con el elemento de enmedio (6 comparaciones) , dependiendo si es mayor o menor lo comparamos con uno de los otros dos elementos (7 comparaciones) y ponemos a c en su lugar

de esta forma ordenamos 5 elementos con 7 comparaciones

2) si habra puertas abiertas y estas son

la puerta k-esima esta abierta si cumple esto

- si k es par y el numero de divisores menores o iguales a k es impar
- si k es impar y el numero de divisores menores o iguales a k es par

ahora veamos porque, primero veamos que pasa sin lo que hizo Mike Wazowski, tenemos que todas las puertas estan abiertas por lo tanto para que sigan abiertas deben ser modificadas un numero par de veces y como son modificadas en los ciclos que son divisores del numero de la puerta, de ahi sale que el numero de divisores menores o iguales a k es par, pero nos falta lo de Mike Wazowski, lo que hizo el fue alterar el estado inicial de las puertas pares (dejarlas cerradas) por lo tanto para que quedaran abiertas necesitamos que sean alteradas un numero impar de veces, de ahi sale la primera posible condicion ejemplo con 12 puertas

ciclo

puerta	0	1	2	3	4	5	6	7	8	9	10	11	12	M	final	
1	A	C											C	C	C	
2	A	C	A										A	C	C	
3	A	C		A									A	A	A	←
4	A	C	A		C								C	A	A	←
5	A	C				A							A	A	A	←
6	A	C	A	C			A						A	C	C	
7	A	C						A					A	A	A	←
8	A	C	A		C				A				A	C	C	
9	A	C		A						C			C	C	C	
10	A	C	A			C					A		A	C	C	
11	A	C										A	A	A	A	←
12	A	C	A	C	A		C						A	C	C	

3) necesitamos 7 carreras, veamos por qué, como las carreras son de 5 personas entonces dividimos en 5 grupos, de tal forma que en el grupo uno estén las personas 1-1, 1-2, .., 1-5 luego en el segundo grupo están 2-1, 2-2, ..., 2-5 y así los 5 grupos, ahora hacemos una carrera en cada grupo (5 carreras), para facilitar las cosas digamos que los 3 primeros lugares del grupo uno fueron 1-1, 1-2 y 1-3 (primero, segundo y tercero), en el grupo dos fueron 2-1, 2-2 y 2-3, lo mismo en los grupos cuatro, cinco y seis

ahora tomamos a los primeros lugares de las 5 carreras (1-1, 2-1, ..., 5-1) y hacemos una carrera (6 carreras), para seguir facilitando las cosas digamos que así fueron los resultados, primero fue 1-1, segundo 2-1, tercero 3-1, por lo tanto tenemos que 1-1 es el más rápido de todos, pero nos falta el segundo y tercero más rápido, para esto haremos otra carrera (7 carreras) y los competidores serán 2-1, 3-1, 1-2 (el segundo lugar del grupo 1), 2-2 (el segundo lugar del grupo 2) y 1-3 (el tercer lugar del grupo 1),

al final de esta carrera el primer lugar sera el segundo más rapido

y el segundo lugar sera el tercero más rapido

imagen

grupo 1	1-1	1-2	1-3	1-4	1-5
grupo 2	2-1	2-2	2-3	2-4	2-5
grupo 3	3-1	3-2	3-3	3-4	3-5
grupo 4	4-1	4-2	4-3	4-4	4-5
grupo 5	5-1	5-2	5-3	5-4	5-5

- 1ra carrera
- 2da carrera
- 3ra carrera
- 4ta carrera
- 5ta carrera
- 6ta carrera
- 7ma carrera

4) a) algoritmo de tiempo cuadratico,

si el arreglo tiene menos de 2 elementos error , en otro caso hacemos esto

(una version tipo java)

```
public String max (int arreglo[]) {

    int par1 = arreglo [0];
    int par2 = arreglo [1];
    int n = arreglo.length();

    for (int par3 = 0; par3 < n; par3++) {

        for (int par4 = par3+1; par4 < n; par4++) {

            if ( arreglo [par3] * arreglo [par4] >= par1 * par2) {

                par1 = arreglo [par3];
                par2 = arreglo [par4];
            }
        }
    }

    return "la pareja es " + par1 + " y " + par2;
}
```

veamos por qe es tiempo cuadratico , esto es debido a que

tenemos un for dentro de otro for (lo cuales recorren el arreglo),

el for exterior recorre una vez el arreglo por lo tanto es O(n), en el for interno tambien recorremos el arreglo $O(n)$, pero esto lo haremos una vez por cada elemento del arreglo, y las acciones dentro del for internos siempre seran las mismas por lo tanto eso es constante, y la n^2 sale de que por cada elemento del arreglo recorremos el arreglo $1 * n * n = n^2$, por eso es $O(n^2)$

el algoritmo funciona debido a que revisamos todas las posibles parejas y las comparamos para tener la mejor pareja posible

a) algoritmo tiempo $O(n \log n)$

si el arreglo tiene menos de 2 elementos error, en otro caso hacemos esto.

primero vamos a usar un algoritmo de ordenamiento, usamos merge sort para tener el arreglo de menor a mayor, luego hacemos esto, tomamos dos parejas, la primera con los dos valores mayores y la otra con los valores menores, luego hacemos el producto de cada pareja y comparamos los productos y regresamos los valores que generaron el producto mayor veamos la complejidad, la primera parte es ordenar el arreglo y sabemos que merge sort siempre toma $O(n \log n)$ por lo tanto esta parte toma $O(n \log n)$, ahora la parte de regresar la pareja es constante, ya que siempre son las mismas instrucciones, notamos que la parte de mayor complejidad es ordenar, por lo que concluimos que la complejidad es $O(n \log n)$

el algoritmo funciona debido a que tenemos acceso a la pareja de los valores más grandes y más pequeños, y esto nos sirve ya que $(-)(-)=(+)$ por lo tanto el producto de los menores puede ser mayor al producto de los mayores

b) el inciso anterior se puede mejorar, debido a que en el algoritmo pasado buscamos el valor mayor, el segundo mayor, el menor y el segundo menor, pero esto lo podemos buscar estos elementos solo comparando, ya que buscar el elemento mayor o menor nos toma $n-1$ comparaciones, por lo tanto haremos esto primero buscamos el elemento mayor comparando, el primer elemento con el segundo, y quedartonos con el mayor, así con todos los elementos, luego de la misma forma buscamos el elemento menor (nos quedamos con el menor en vez del mayor), luego volvemos a buscar el mayor pero sin el elemento mayor, que encontramos antes, en el arreglo, hacemos lo mismo para el segundo menor luego que tenemos los cuatro valores, juntamos los mayores y sacamos su producto y lo comparamos con el producto de los menores y regresamos a la pareja con producto mayor

la complejidad es esta, encontrar el mayor toma O(n), solo hacemos n-1 comparaciones, y encontrar los otros 3 valores también toman O(n), y cómo vimos antes decidir la pareja correcta es constante, por lo tanto la complejidad es O(n)

el algoritmo funciona debido a que es igual al anterior solo obtenemos de una manera distinta las parejas de mayores y menores

5) para este algoritmo usamos una estructura auxiliar, un minheap entonces el algoritmo es asi

creamos un minheap e insertamos los k+1 primeros elementos del arreglo (tomamos que el arreglo tiene como primer indice a [0), luego creamos una variable para indice $i=0$ y un arreglo nuevo de tamaño n, despues hacemos un for (for ($j = k+1$; $j<n$; $j++$)) dentro del cual sacamos un elemento del minheap y lo ponemos en la posicion i del arreglo nuevo, despues incrementamos en 1 a i y metemos el j-esimo elemento del arreglo original a la cola, cuando termine el for tenemos que hacer un while que corra mientras la cola no este vacia, lo que haremos en el while sera sacar un elemento del minheap, ponerlo en la posicion i del arreglo nuevo e incrementar i

al final regresamos el arreglo nuevo

veamos la complejidad, notemos que para agregar o sacar del minheap nos toma $O(\log k)$, ya que tenemos k elementos y en la clase vimos que "arreglar" el arbol binario del heap toma

$O(\log m)$ con m el numero de elementos, ahora veamos que mientras "recorremos" el arreglo original vamos metiendo y sacando del minheap, por lo tanto cada elemento se mete y saca una vez, es decir $2 * n * \log(k) = 2n\log(k)$, entonces la complejidad es $O(n \log k)$

veamos por que funciona, esto es debido a que como usamos un minheap siempre va a estar "ordenado", es decir el menor elemento sera el que saquemos, y al tener solo k elementos y que cada elemento esta a lo más a k posiciones de su lugar entonces cuando saquemos un elemento del minheap y lo pongamos en el arreglo nuevo, va a estar en su posicion correcta, luego metemos otro elemento (el siguiente del arreglo original) para que se cumpla lo de las k posiciones, hasta terminar con todos ordenados

6) para esto veamos como sirve encontrar el mínimo y maximo usando un metodo de torneo, para esto veremos el caso cuando n es par, dividimos en parejas, entonces tendremos $n/2$ parejas, luego hacemos $n/2$ comparaciones y obtenemos el min y max de cada pareja, ahora tomamos a los min y los juntamos para empezarlos a comparar y asi encontrar al minimo de todos, esto nos tomo $\frac{n}{2}-1$ comparaciones, ya comparamos al minimo hasta el momento con todos los min (que son $\frac{n}{2}$) solo que no lo comparamos consigo mismo (el -1), esto lo hacemos igual para encontrar al maximo, entonces nos quedan $\left(\frac{n}{2}\right)+\left(\frac{n}{2}-1\right)+\left(\frac{n}{2}-1\right)=\frac{3n}{2}-2$ comparaciones, notemos que $\frac{3n}{2}-2 = \left\lceil\frac{3n}{2}\right\rceil-2$ ya que n es par

ahora veamos que pasa si n es impar, aqui podemos hacer lo mismo que cuando n es par, pero hay un pequeño cambio y el cambio es que nos va a quedar un elemento, llamemoslo x, el cual no tiene pareja, entonces lo excluimos hasta el final, ya que tenemos el maximo y minimo de los elementos sin x, solo nos falta compararlos con x y esto nos toma 2 comparaciones, es decir si n es impar entonces tomara 2 comparaciones más que si hubiera $n-1$ elementos

ahora para facilitar las cosas supongamos que n es par, entonces $n+1$ es impar, por lo tanto para n elementos nos toma $\lceil \frac{3n}{2} \rceil - 2$ comparaciones (en el peor caso que es haciendo todas las comparaciones)

y $n+1$ toma $\frac{3n}{2} - 2 + 2$, ahora veamos que

$\frac{3(n+1)}{2} - 2 = \frac{3n}{2} + \frac{3}{2} - 2$ y que $\lceil \frac{3}{2} \rceil = 2$, entonces

$\lceil \frac{3(n+1)}{2} \rceil - 2 = \frac{3n}{2} - 2 + 2$, por lo tanto en el peor caso son necesarias $\lceil \frac{3n}{2} \rceil - 2$ comparaciones

7) el algoritmo es bucket sort, con algunos pequeños cambios entonces sera algo asi, primero creamos 4 cubetas vacias, y empezamos a recorrer a los n estudiantes y a cada uno lo enviamos a la cubeta que corresponga a la casa que le toco (la casa 1 a la cubeta 1 y asi), luego vamos a ordenar cada cubeta usando insertion sort, pero para comparar a cada estudiante usaremos a que casa pertenecen, y por ultimo paso sacamos todos los elementos de las cubetas en el orden de las cubetas y listo

De esta manera los estudiantes estaran ordenados, primero los de la casa 1, luego los de la casa 2 y asi hasta la casa 4

Ahora veamos la complejidad, en el primer paso solo recorremos a los estudiante entonces toma $O(n)$, luego toca ordenar cada cubeta, como usamos insertion sort sabemos que su complejidad en el caso general es $O(n^2)$, pero este no es un caso general, ya que todos los elementos de cada cubeta tienen el mismo valor (a que casa pertenecen), por lo tanto no tenemos que hacer los cambios de lugar y comparaciones (dependiendo de la implementacion), por lo tanto estamos en el mejor caso posible y la complejidad es O(n), por ultimo regresar de las cubetas nos toma O(n), ya que pasamos por cada estudiante, entonces la complejidad de todo es O(n)

y para la complejidad del espacio sabemos que bucket sort tiene O(n+k), donde k es el numero de cubetas, aqui usamos 4 por lo tanto no es mucho espacio

8) si podemos ordenar L si m es $O(n)$ en tiempo lineal

para esto usaremos counting sort, repasemos un poco sobre counting sort, se crea un arreglo de tamaño k (que en este caso sera m) y luego se recorre L mientras en el arreglo nuevo vamos incrementando las ocurrencias de cada elemento, luego en el arreglo nuevo vemos las apariciones de cada elemento y tendremos en que posicion va a estar cada elemento en la lista ordenada, creamos una lista nueva y recorremos el nuevo arreglo y agregamos cada elemento la misma cantidad de ocurrencias, agregando al final (para que sea lineal)

notemos que en este algoritmo solo recorremos la lista ($O(n)$) y el arreglo ($O(k)$), y ya que agregar en la lista al final y agregar en un arreglo toma $O(1)$, entonces la complejidad total es $O(n+k)$, pero como m es $O(n)$ y k es m en este caso tenemos que podemos ordenar la lista en $O(n)$

si m fuera $O(n^2)$ podemos ordenar tambien en tiempo lineal,

para esto usamos radix sort pero con un cambio

primero veamos como funciona radix sort rapidamente

primero encontramos el elemento más grande del arreglo, contamos la cantidad de digitos que tiene, seran las veces que repetiremos el siguente paso

ahora vamos a ordenar los elementos basandonos en el lugar de las unidades (usando sistema decimal), para esto usamos counting sort

luego vamos a repetir este paso dependiendo del numero de digitos del mayor elemento, solo que ahora no nos basamos en las unidades, si no en el digito del ciclo en el que estemos, por ejemplo en el ciclo 2 nos fijamos en las decenas, en el ciclo 3 nos fijamos en las centenas y así, de esta manera luego del ultimo ciclo tendremos todo ordenado

ahora ya sabemos cual es la complejidad de este algoritmo que es $O(d * (n + b))$ donde n es el numero de elementos, d es el numero de digitos y b es la base del sistema de numeracion que usamos ahora veamos el cambio que haremos a radix sort

como radix sort toma $O(d*(n+b))$ necesitamos volver esa d en una constante, para esto recordemos que la formula que da el numero de digitos de un numero es $\lfloor \log_a(b) \rfloor + 1$, donde a es la base y b es el numero, ahora como m es $O(n^2)$ entonces el numero maximo es n^2, por lo tanto $d = \lfloor \log_a(n^2) \rfloor + 1$, ahora necesitamos una a para que d sea una constante, digamos que $a=n$, es decir trabajaremos en base n, entonces $d = \lfloor \log_n(n^2) \rfloor + 1$ y esto nos da $3=d$, ya que $n>0$, por lo tanto ya lo logramos debido a que la complejidad $O(d*(n+b))$ se vuelve $O(3*(n+n)) = O(6n) \simeq O(n)$,

pero nos falta ver como trabajar en base n

para esto recordemos que para pasar un numero b a base a, tenemos que ir dividiendo por a y tomar residuos, por lo tanto la complejidad es $O(\log_a(b))$, ya que vamos partiendo

ahora si tenemos que nuestros elementos son de 1 a n^2, entonces la complejidad de pasar cada uno va a ser $O(\log_n(n^2))$, ya que n^2

es el mayor, pero $\log_n(n^2) = 2$ entonces la complejidad es constante

por lo tanto podemos cambiar a todos los elementos de base en $O(n)$

ahora para regresarlo a base 10 tenemos que ir multiplicando cada digito por n a la posicion de la base y luego sumarlo, entonces la complejidad de esto es $O(h)$ donde h es el numero de digitos para representar al numero en la base (que no es 10) y eso ya sabiamos que era d, y en este caso que la base es n y el maximo es n^2, tenemos que $d = 3$ entonces esto es constante y por lo tanto regresar todos a base 10 es $O(n)$

por lo tanto el algoritmo seria, primero pasar a todos a base n, luego hacer radix sort en base n y por ultimo regresar a todos a base 10

y como todos los pasos tienen complejidad $O(n)$, entonces la complejidad total es $O(n)$, y se cumple lo que queremos