Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2133 — Estructuras de Datos y Algoritmos
2021 - 2

Pauta Interrogación 1

Pregunta 1

a)

Para demostrar que es correcto se utilizara el método de inducción:

Caso Base:
Cuando el largo del input es 1, el algoritmo no hace nada antes de retornar, lo que es correcto ya que un array de largo uno ya esta ordenado. Cuando el largo es igual a dos, ordena ambos elementos y retorna el array.

Hipótesis:
El algoritmo es correcto para inputs de cualquier largo que estén en el rango entre 1 y n-1. Consideremos un inputArray de largo n, donde $3 \leq n$. Por la hipótesis tenemos que las lineas después de la condición del largo de input $3 \leq n$ ordenan correctamente dos tercios del inputArray. Necesitamos demostrar que el array entero queda ordenado.

Detalle:
Generalmente se ignora que cada dos tercios que es ordenado, se ordena en verdad $\lceil \frac{2n}{3} \rceil$ elementos. El hecho de que se ordene la parte superior de los dos tercios de n y no el piso es significante, pues puede haber una superposición, pero ignoraremos eso en esta demostración.

Tesis:
Sea B un subarray del input original de largo n, que contiene los primeros dos tercios de este:
Después de la primera recursion la primera mitad de elementos no pueden pertenecer nunca al ultimo tercio del array, ya que sabemos que los ultimos elementos de B son mayores que estos.

Entonces la segunda recursion, que contendrá algunos de los últimos elementos de B y el ultimo tercio del array original serán ordenados correctamente.

Finalmente la ultima llamada a recursion vueleve a ordenar B (que posee sus elementos originales ordenados o nuevos elementos del ultimo tercio del array original).
Por lo anterior el algoritmo ordena de forma correcta.

(0.1 pts) Por utilizar el método de inducción.
(0.2 pts) Por mencionar o tomaar en cuenta la importancia de overlap.
(0.3 pts) Por demostrar que el algotimo cumple su funcion.
(0.3 pts) Por demostrar que QuicoSort termine (Queda a criterio del ayudante).
(0.1 pts) Por concluir que QuicoSort es correcto.

b)

Veamos la complejidad de este algoritmo paso por paso:
1. Dividir la lista completa en $\sqrt{n}$ segmentos tiene una complejidad de $O(\sqrt{n})$
2. Ordenar $\sqrt{n}$ listas de largo $\sqrt{n}$ usando quicksort, como sabemos que quicksort en el peor caso tiene complejidad $O(n^2)$ con n largo de arrglo, la complejidad de este paso es:

$$O((\sqrt{n})^2 * \sqrt{n}) = O(n\sqrt{n})$$

3.Crear una lista vacia se toma $O(1)$
4.En este paso, comparar el primer elemento de $\sqrt{n}$ listas obteniendo el menor de ellos tiene una complejidad de $\sqrt{n}$, y esta comparación como maximo se puede realizar $n-1$ veces. Luego la inserción tiene una complejidad de $O(1)$ repitiendo n veces.
En resumen, la complejidad total es:

$$O(\sqrt{n}) + O(n\sqrt{n}) + O(n\sqrt{n}) + O(n) = O(n\sqrt{n})$$

Puntajes:
(0.5) Por calcular correctamente la complejidad de QuickSort
(0.3) Por calcular correctamente la complejidad del paso 4
(0.2) Por llegar al resultado final correcto (de forma correcta)

Pregunta 2

De clases sabemos que, en un arreglo, buscar un elemento se puede hacer en complejidad $\mathcal{O}(\log(n))$ utilizando búsqueda binaria, mientras que la inserción de un elemento en el arreglo tiene complejidad $\mathcal{O}(n)$. En el caso de una lista ligada, la búsqueda de un elemento tiene complejidad $\mathcal{O}(n)$, mientras que la inserción de un elemento toma complejidad $\mathcal{O}(1)$. En ambos casos, realizar n inserciones y una búsqueda por cada inserción resulta en tiempo $\mathcal{O}(n^2)$.
Nuestra nueva estructura de datos *MagicList* puede acceder a un elemento en particular en tiempo $\mathcal{O}(1)$ al igual que un arreglo, y puede insertar en tiempo $\mathcal{O}(1)$ al igual que la lista ligada, trayendo lo mejor de los dos mundos. Luego, podemos aprovechar esto para el algoritmo InsertionSort para mejorar la búsqueda del lugar donde se debe insertar el nuevo elemento, utilizando búsqueda binaria.
Sin embargo, la búsqueda binaria nos entrega el índice del elemento que estamos buscando, por lo que necesitamos una modificación del algoritmo que nos entregue el índice donde el nuevo elemento se debe insertar en el arreglo, utilizando la misma idea para mantener la complejidad $\mathcal{O}(\log(n))$. Para esto, proponemos el siguiente algoritmo:

```
not_bin_search(A, x, i, f): // A un arreglo ordenado
    if i == f:
        if x <= A[i]:
            return i
        else:
            return i + 1
    else:
        m = ceiling((i + f) / 2)
        if A[m - 1] <= x <= A[m]:
            return m
        else if A[m] < x:
            return not_bin_search(A, x, m, f)
        else if x < A[m - 1]:
            return not_bin_search(A, x, i, m-1)
```

Como este algoritmo utiliza la misma idea y procedimiento que búsqueda binaria, también tiene complejidad $\mathcal{O}(\log(n))$. Ahora, si tenemos un elemento, que queremos buscar dónde hay que insertarlo en el arreglo para mantener el orden y luego insertarlo, la operación toma $\mathcal{O}(\log(n) + 1)$ que es $\mathcal{O}(\log(n))$. Luego, realizar esa operación para los n elementos de un arreglo A, toma tiempo $\mathcal{O}(n \log(n))$.

Puntaje:

- [0.7 pto] Por relacionarlo por búsqueda binaria
- [0.3 pto] Mencionar alteración a búsqueda binaria
- [1 pto] Explicar o demostrar que efectivamente es $O(n \log(n))$.

Pregunta 3

1. A) $\longrightarrow$ Utilizar MergeSort() para ordenar la Linked List $\mathcal{O}(n \log n)$. para ordena los asientos podemos utilizar :

   a) InsertionSort() $\mathcal{O}(M^2)$
   b) MergeSort() $\mathcal{O}(M \log M)$
   c) QuickSort() $\mathcal{O}(M^2)$
   d) SelectionSort() $\mathcal{O}(M^2)$

   Pseudo-codigo implementación :
   Identificar las sublistas a mezclar y la condición de terminación del algoritmo:

   sea P = puntero al elemento vigente de la lista (inicialmente, al primer elemento) a partir de P, recorrer la lista mientras los elementos no decrezcan en valor.
   sea Q = puntero al último elemento "no decreciente"
   sea R = puntero al siguiente elemento en la lista

   if R = null
   2terminar, la lista está ordenada
   else
   2a partir de R, recorrer la lista mientras los elementos no decrezcan en valor.
   2sea S = puntero al último elemento no decreciente
   2elemento vigente = S.next
   2**Mezclar** la sublista de P a Q con la sublista de R a S (*)
   2volver a la primera instrucción

   Hacer la mezcla (el algoritmo merge visto en clase no sirve, porque mezcla arreglos y no listas)
   (*) Mezclar la sublista de P a Q con la sublista de R a S:

   T = T3 = new(Node)
   T1 = P, T2 = R
   if T1 < T2
   2T3 = T1, T1 = T1.next
   else
   2T3 = T2, T2 = T2.next
   if T1 = Q.next
   2agregar a a T3 la lista de T2 a S
   else
   2if T2 = S.next
   22agregar a T3 la lista de T1 a Q
   2else
   22repetir el primer if
   P = T

   (0.25 pts) Reconocer la utilización de MergeSort() para Listas Ligadas
   (0.25 pts) Especificar el algoritmo de ordenación para los asientos.
   (0.5 pts) Explicación de la implementación

   Consideración:
   Si el alumno no entrega pseudo-código pero explica detalladamente como hubiera implementado su algoritmo en C, asignar puntaje completo.

2. B) $\longrightarrow$ Dado n vagones y m asientos dentro de cada vagón y ordenando la Lista Ligada utilizando MergeSort() el cual tiene complejidad $\mathcal{O}(n \log n)$ y considerando cualquiera de estos algoritmos para los asientos se tienen las siguiente complejidad:

   - InserionSort(): Debido a que la complejidad de ordenar un arreglo de asientos es $\mathcal{O}(M^2)$ y al haber n vagones, la complejidad total es de $\mathcal{O}(n \cdot M^2 + n \log n) \approx \mathcal{O}(n \cdot M^2)$.
   - SelectionSort(): Debido a que la complejidad de ordenar un arreglo de asientos es $\mathcal{O}(M^2)$ y al haber n vagones, la complejidad total es de $\mathcal{O}(n \cdot M^2 + n \log n) \approx \mathcal{O}(n \cdot M^2)$.
   - QuickSort(): Debido a que la complejidad de ordenar un arreglo de asientos es $\mathcal{O}(M^2)$ y al haber n vagones, la complejidad total es de $\mathcal{O}(n \cdot M^2 + n \log n) \approx \mathcal{O}(n \cdot M^2)$.
   - MergeSort(): Debido a que la complejidad de ordenar un arreglo de asientos es $\mathcal{O}(M \log M)$ y al haber n vagones, la complejidad total es de $\mathcal{O}(n \cdot M \log M + n \log n) \approx \mathcal{O}(n \cdot M \log M)$.

   (0.25 pts) Tener bien la complejidad de algoritmo para los asientos, varia según implementación.
   (0.5 pts) Llegar a la complejidad final conjunta, MergeSort() + Asientos
   (0.25) Realizar la diferenciación en complejidad en función de N vagones y M asientos

Pregunta 4

a)

Este seria el peor caso, ya que en cada instancia tendra que comparar con los $n-k$ elementos a su izquierda, asi causando $n+(n-1)+\cdots+1$. concluyendo $\mathcal{O}(n^2)$
(0.25 pts) Por calcular correctamente la complejidad.
(0.25 pts) Por llegar al resultado correcto.

b)

El numero de llamadas seria el minimo, debido a que solo se realizaran $log(n)$ sub llamadas (ademas los pivotes ya estan ordenados). Asi presentando el mejor caso de $\mathcal{O}(nlog(n))$
(0.25 pts) Por calcular correctamente la complejidad.
(0.25 pts) Por llegar al resultado correcto.

c)

El peor caso es cuando se eligen 2 pivotes que sean consecutivos en el array y posicionados relativamente al extremo izquierdo, ya que de esa forma, se alcanza la complejidad de $O(n^2)$ al tener que realizar el maximo numero de comparaciones

(0.25 pts) Por elegir un caso válido.
(0.25 pts) Por argumentar correctamente la elección de este.

d)

El mejor caso es cuando el array esta ordenado y ademas los pivotes elegidos en cada etapa son exactamente $\frac{1}{3}$ y $\frac{2}{3}$ del array. Ya que en este caso, las operaciones en cada llamada recursiva son minimas. Y por ende se alcanza una complejidad de $O(nlog(n))$

(0.25 pts) Por elegir un caso válido.
(0.25 pts) Por argumentar correctamente la elección de este.