Estructuras de Datos y Algoritmos – IIC2133

I1

13 sept. 2021

Hora de inicio: 14.00
Entrega: hasta las 23.59

0) Responde esta pregunta de manera manuscrita.

a) Nombre completo y número de alumno: _____

b) Me comprometo a **no preguntar ni responder dudas** de esta prueba, ya sea directa o indirectamente, a nadie que no sea parte del equipo docente del curso. **Firma**: _____

Responde sólo 3 de las siguientes 4 preguntas

1)

a) Te presentamos el siguiente algoritmo de ordenación, que bautizamos *QuicoSort*:

QuicoSort($A[i, f]$):
Tomar $n = f - i + 1$.
Si $n \leq 1$:
No hacer nada.
Si $n = 2$:
Si $A[1] > A[2]$, hacerles swap.
Si $n \geq 3$:
Tomar $a = i + \lfloor n/3 \rfloor$ y $b = f - \lfloor n/3 \rfloor$.
QuicoSort($A[i, b]$)
QuicoSort($A[a, f]$)
QuicoSort($A[i, b]$)

Demuestra que ***QuicoSort*** es correcto.

b) Te presentamos el siguiente algoritmo de ordenación, que se llama *SqrtSort*.

SqrtSort($A[1,n]$): (Puedes asumir que n es un cuadrado perfecto)

1. Separar A en $\sqrt{n}$ segmentos $S_1, S_2, \ldots, S_{\sqrt{n}}$ de igual tamaño.
2. Ordenar cada uno de los segmentos $S_1, S_2, \ldots, S_{\sqrt{n}}$ usando Quick Sort.
3. Crear una nueva lista B vacía.
4. Mientras queden elementos en alguno de los segmentos S_i:
   a. Elegir el S_i cuyo primer elemento x sea el menor de todos los que quedan.
   b. Eliminar x de S_i y agregarlo al final de B.

Calcula la complejidad de ***SqrtSort*** en el peor caso.

2) Imagina que tienes una estructura de datos hipotética que se llama **MagicList**. Esta estructura almacena valores en secuencia y, si tiene n elementos, permite las siguientes dos operaciones en tiempo **constante**:

1. Acceder al elemento en la posición i, para cualquier i entre 1 y n.
2. Insertar un elemento en la posición i, para cualquier i entre 1 y n+1. Si i = 1, entonces lo agrega al comienzo; si i = n+1, lo agrega al final; en otro caso, lo agrega entre los elementos en las posiciones i–1 e i. El largo de la lista nueva es n+1.

Ahora, tú vas a implementar una versión de *InsertionSort* que aprovecha el poder de **MagicList**. Recuerda que *InsertionSort* funciona de la siguiente manera, cuando el input es una secuencia A:

1. Definir una secuencia B, que inicialmente está vacía.
2. Tomar el primer dato x de A y quitarlo.
3. Insertar x en B de tal manera que quede ordenado.
4. Si quedan datos en A, volver a 2.

Al final de estos pasos, B va a contener los datos de A en orden.

Explica detalladamente cómo **MagicList** te permite implementar una versión de *InsertionSort* que toma tiempo O(n log n)

3) Luego de realizar la Tarea 0 acerca de DCCwarts, un amigo tuyo te dijo que tenía un problema. La lista ligada de vagones y el arreglo de los asientos de cada vagón se desordenaron misteriosamente (el desorden es respecto al *id* de los vagones y al *id* de los pasajeros). Todo esto ocurrió mientras tú te comías un delicioso pan con huevo en el patio de ingeniería. Para poder resolver esta interrogante y otras dudas, tu amigo le pidió que lo ayudaras con lo siguiente:

a) Tu debes ordenar con una estrategia que asegure tiempo O(n log n) en el peor caso, aunque para los arreglos de asientos puede usar cualquier algoritmo. Para esto, puedes escribir un pseudocódigo o explicar detalla-damente cómo lo harías (pensado en el lenguaje C).

b) Una vez hecho esto, tu amigo no sabe cuál es la complejidad de este algoritmo. Calcúlala y muestra cómo llegaste a esa conclusión.

4) Considera el algoritmo QuickSort estudiado en clases para ordenar un arreglo A. Calcula la complejidad de la ejecución del algoritmo en cada uno de los siguientes casos:

a) Si el arreglo *A* viene ordenado (de menor a mayor) y siempre, en cada llamada a *Partition*, se elige como pivote el último elemento (el de más a la derecha) del subarreglo correspondiente.

b) Si el arreglo *A* viene ordenado (de menor a mayor) y siempre, en cada llamada a *Partition*, se elige como pivote el elemento del medio del subarreglo correspondiente.

Considera ahora una nueva versión del algoritmo QuickSort, llamada *QuickerSort*, para ordenar un arreglo A en que todos los elementos son distintos:

```
QuickerSort(A[i, f]):
    if i ≤ f:
        (p, q) ← PartitioninThree(A[i, f])
        QuickerSort(A[i, p–1])
        QuickerSort(A[p+1, q–1])
        QuickerSort(A[q+1, f])
```

PartitioninThree elige dos pivotes, $A[i]$ y $A[f]$, y reorganiza el subarreglo $A[i, f]$ de modo que todos los elementos menores que el menor pivote queden en el extremo izquierdo del subarreglo, todos los elementos mayores que el mayor pivote queden en el extremo derecho del subarreglo, y todos los otros elementos queden en el subarreglo entre ambos pivotes; p y q son las posiciones finales de los pivotes al terminar la ejecución de *PartitioninThree*, tal que $p < q$ y $A[p] < A[q]$.

Para este algoritmo *QuickerSort*, responde:

c) Da un caso que sea lo más lento posible y argumenta por qué es así.

d) Da un caso que sea lo más rápido posible y argumenta por qué es así.