Algoritmos y Estructuras de Datos

Guía Práctica 8
Sorting

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicio 1. Comparar la complejidad de los algoritmos de ordenamiento dados en la teórica para el caso en que el arreglo a ordenar se encuentre perfectamente ordenado de manera *inversa* a la deseada.

Ejercicio 2. Imagine secuencias de naturales de la forma s = Concatenar (s', s''), donde s' es una secuencia ordenada de naturales y s'' es una secuencia de naturales elegidos al azar. ¿Qué método utilizaría para ordenar s? Justificar. (Entiéndase que s' se encuentra ordenada de la manera deseada.)

Ejercicio 3. Escribir un algoritmo que encuentre los k elementos más chicos de un arreglo de dimensión n, donde $k \leq n$. ¿Cuál es su complejidad temporal? ¿A partir de qué valor de k es ventajoso ordenar el arreglo entero primero?

Ejercicio 4. Se tiene un conjunto de n secuencias $\{s_1, s_2, \ldots, s_n\}$ en donde cada s_i $(1 \leq i \leq n)$ es una secuencia ordenada de naturales. ¿Qué método utilizaría para obtener un arreglo que contenga todos los elementos de la unión de los s_i ordenados. Describirlo. Justificar.

Ejercicio 5. Se tiene un arreglo de n números naturales que se quiere ordenar por frecuencia, y en caso de igual frecuencia, por su valor. Por ejemplo, a partir del arreglo [1, 3, 1, 7, 2, 7, 1, 7, 3] se quiere obtener [1, 1, 1, 7, 7, 7, 3, 3, 2]. Describa un algoritmo que realice el ordenamiento descrito, utilizando las estructuras de datos intermedias que considere necesarias. Calcule el orden de complejidad temporal del algoritmo propuesto.

Ejercicio 6. Sea $A[1 \ldots n]$ un arreglo que contiene n números naturales. Diremos que un rango de posiciones $[i \ldots j]$, con $1 \leq i \leq j \leq n$, contiene una escalera en A si valen las siguientes dos propiedades:

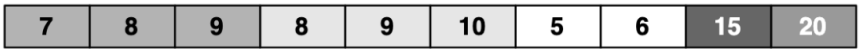
1. $(\forall\, k : \text{nat})\ i \leq k < j \implies A[k+1] = A[k] + 1$ (esto es, los elementos no sólo están ordenados en forma creciente, sino que además el siguiente vale exactamente uno más que el anterior).
2. Si $1 < i$ entonces $A[i] \neq A[i-1] + 1$ y si $j < n$ entonces $A[j+1] \neq A[j] + 1$ (la propiedad es *maximal*, es decir que el rango no puede extenderse sin que deje de ser una escalera según el punto anterior).

Se puede verificar fácilmente que cualquier arreglo puede ser descompuesto de manera única como una secuencia de escaleras. Se pide escribir un algoritmo para reposicionar las escaleras del arreglo original, de modo que las mismas se presenten en orden decreciente de longitud y, para las de la misma longitud, se presenten ordenadas en forma creciente por el primer valor de la escalera.

El resultado debe ser del mismo tipo de datos que el arreglo original. Calcule la complejidad temporal de la solución propuesta, y justifique dicho cálculo.

Por ejemplo, el siguiente arreglo

debería ser transformado a



Ayuda: se aconseja comenzar el ejercicio con una clara descripción en castellano de la estrategia que propone para resolver el problema.

Ejercicio 7. Suponga que su objetivo es ordenar arreglos de naturales (sobre los que no se conoce nada en especial), y que cuenta con una implementación de árboles AVL. ¿Se le ocurre alguna forma de aprovecharla? Conciba un algoritmo que llamaremos *AVL Sort*. ¿Cuál es el mejor orden de complejidad que puede lograr?

Ayuda: debería hallar un algoritmo que requiera tiempo $O(n \log d)$ en peor caso, donde n es la cantidad de elementos a ordenar y d es la cantidad de elementos distintos. Para inspirarse, piense en *Heap Sort* (no en los detalles puntuales, sino en la idea general de este último).

Justifique por qué su algoritmo cumple con lo pedido.

Ejercicio 8. Se tienen dos arreglos de números naturales, $A[1..n]$ y $B[1..m]$. Nada en especial se sabe de B, pero A tiene n' secuencias de números repetidos continuos (por ejemplo $A = [333331111888887777145555], n' = 7$). Se sabe además que n' es mucho más chico que n. Se desea obtener un arreglo C de tamaño $n+m$ que contenga los elementos de A y B, ordenados.

1. Escriba un algoritmo para obtener C que tenga complejidad temporal $O(n + (n' + m)\log(n' + m))$ en el peor caso. Justifique la complejidad de su algoritmo.
2. Suponiendo que todos los elementos de B se encuentran en A, escriba un algoritmo para obtener C que tenga complejidad temporal $O(n + n'(\log(n') + m))$ en el peor caso y que utilice solamente arreglos como estructuras auxiliares. Justifique la complejidad de su algoritmo.

Ejercicio 9. Considere la siguiente estructura para guardar las notas de un alumno de un curso:

alumno es tupla⟨*nombre*: string, *género*: GEN, *puntaje*: Nota⟩ donde GEN es $enum\{masc, fem\}$ y Nota es un nat no mayor que 10.

Se necesita ordenar un arreglo(alumno) de forma tal que todas las mujeres aparezcan al inicio de la tabla según un orden creciente de notas y todos los varones aparezcan al final de la tabla también ordenados de manera creciente respecto de su puntaje, como muestra en el siguiente ejemplo:

Entrada

Ana	*F*	10
Juan	*M*	6
Rita	*F*	6
Paula	*F*	7
Jose	*M*	7
Pedro	*M*	8

Salida

Rita	*F*	6
Paula	*F*	7
Ana	*F*	10
Juan	*M*	6
Jose	*M*	7
Pedro	*M*	8

1. Proponer un algoritmo de ordenamiento *ordenaPlanilla*(**inout** *p*: *planilla*) para resolver el problema descripto anteriormente y cuya complejidad temporal sea $O(n)$ en el peor caso, donde n es la cantidad de elementos del arreglo. Justificar.
2. Modificar la solución del inciso anterior para funcionar en el caso que GEN sea un tipo enumerado con más elementos (donde la cantidad de los mismos sigue estando acotada y el órden final está dado por el órden de los valores en el enum. Puedo hacer ***for*** *g* ***in*** *GEN*).
3. ¿La cota $O(n)$ contradice el "lower bound" sobre la complejidad temporal en el peor caso de los algoritmos de ordenamiento? (El Teorema de "lower bound" establece que todo algoritmo general de ordenamiento tiene complejidad temporal $\Omega(n \log n)$.) Explique su respuesta.

Ejercicio 10. Suponer que se tiene un hipotético algoritmo de "casi" ordenamiento *casiSort* que dado un arreglo A de n elementos, ordena $n/2$ elementos arbitrarios colocándolos en la mitad izquierda del arreglo $A[1 \ldots n/2]$. Los elementos no ordenados de A se colocan en la mitad derecha $A[n/2 + 1 \ldots n]$.

1. Describir un algoritmo de ordenamiento para arreglos de n elementos (con n potencia de 2) utilizando el algoritmo *casiSort*.
2. Obtener la complejidad temporal en el peor caso del algoritmo dado en el punto anterior, suponiendo que *casiSort* tiene complejidad temporal $\Theta(n)$.
3. Cree que es posible diseñar un algoritmo para *casiSort* que realice $O(n)$ comparaciones en el peor caso? Justifique su respuesta.

Ejercicio 11. Sea $A[1 \ldots n]$ un arreglo de números naturales *en rango* (cada elemento está en el rango de 1 a k, siendo k alguna constante). Diseñe un algoritmo que ordene esta clase de arreglos en tiempo $O(n)$. Demuestre que la cota temporal es correcta.

Ejercicio 12. Se desea ordenar los datos generados por un sensor industrial que monitorea la presencia de una sustancia en un proceso químico. Cada una de estas mediciones es un número entero positivo. Dada la naturaleza del proceso se sabe que, dada una secuencia de n mediciones, a lo sumo $\lfloor \sqrt{n} \rfloor$ valores están fuera del rango $[20, 40]$.

Proponer un algoritmo $O(n)$ que permita ordenar ascendentemente una secuencia de mediciones y justificar la complejidad del algoritmo propuesto.

Ejercicio 13. Se tiene un arreglo $A[1 \dots n]$ de T, donde T son tuplas $\langle c_1 : nat \times c_2 : string[\ell] \rangle$ y los $string[\ell]$ son *strings* de longitud máxima ℓ. Si bien la comparación de dos *nat* toma $O(1)$, la comparación de dos $string[\ell]$ toma $O(\ell)$. Se desea ordenar A en base a la segunda componente y luego la primera.

1. Escriba un algoritmo que tenga complejidad temporal $O(n\ell + n\log(n))$ en el peor caso. Justifique la complejidad de su algoritmo.
2. Suponiendo que los naturales de la primer componente están acotados, adapte su algoritmo para que tenga complejidad temporal $O(n\ell)$ en el peor caso. Justifique la complejidad de su algoritmo.

Ejercicio 14. Se tiene un arreglo A de n números naturales y un entero k. Se desea obtener un arreglo B ordenado de $n \times k$ naturales que contenga los elementos de A multiplicados por cada entero entre 1 y k, es decir, para cada $1 \leq i \leq n$ y $1 \leq j \leq k$ se debe incluir en la salida el elemento $j \times A[i]$. Notar que podría haber repeticiones en la entrada y en la salida.

a) Implementar la función

```
proc ordenarMúltiplos(in A: array<nat>, in k: nat): array<nat>
```

que resuelve el problema planteado. La función debe ser de tiempo $O(nk \log n)$, dónde $n =$ `A.length` (el tamaño del arreglo).

b) Calcular y justificar la complejidad del algoritmo propuesto.

Ejercicio 15. Dado un conjunto de naturales, diremos que un agujero es un natural x tal que el conjunto no contiene a x y sí contiene algún elemento menor que x y algún elemento mayor que x.

Diseñar un algoritmo que, dado un arreglo de n naturales, diga si existe algún agujero en el conjunto de los naturales que aparecen en el arreglo. Notar que el arreglo de entrada podría contener elementos repetidos, pero en la vista de conjunto, no es relevante la cantidad de repeticiones.

a) Implementar la función

```
proc tieneAgujero?(in A: array<nat>): bool
```

que resuelve el problema planteado. La función debe ser de tiempo lineal en la cantidad de elementos de la entrada, es decir, $O(n)$, dónde $n =$ `A.length` (el tamaño del arreglo).

b) Calcular y justificar la complejidad del algoritmo propuesto.

Ejercicio 16. Se tiene un arreglo A de n números naturales. Sea $m := \text{máx}\{A[i] : 1 \leq i \leq n\}$ el máximo del arreglo. Se desea dar un algoritmo que ordene el arreglo en $O(n \log m)$, utilizando únicamente arreglos y variables ordinarias (i.e., sin utilizar listas enlazadas, árboles u otras estructuras con punteros).

a) Implementar la función

```
proc raroSort(in A: array<nat>): bool
```

que resuelve el problema planteado. La función debe ser de tiempo $O(n \log m)$, dónde $n =$`A.length` (el tamaño del arreglo) y $m = \text{máx}\{A[i] : 1 \leq i \leq n\}$.

b) Calcular y justificar la complejidad del algoritmo propuesto.

c) Hay al menos 3 formas de resolver este ejercicio, pensar las que se puedan, discutir luego con los compañeros las que encontraron ellos y si hace falta, consultar.

Ejercicio 17. Se tiene un arreglo de enteros no repetidos $A[1..n]$, tal que se sabe que para todo i hay a lo sumo i elementos mas chicos que $A[i]$ en todo el arreglo. Dar un algoritmo que ordene el arreglo en $O(n)$.

Ejercicio 18. *Arreglos de enteros.*

1. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n en tiempo $O(n)$.
2. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^2 en tiempo $O(n)$. *Pista: Usar varias llamadas al ítem anterior.*
3. Dar un algoritmo que ordene un arreglo de n enteros positivos menores que n^k para una constante arbitraria pero fija k.
4. ¿Qué complejidad se obtiene si se generaliza la idea para arreglos de enteros no acotados?