Soluciones de Ejercicios Seleccionados de la Guía 4

Algoritmos y Estructuras de Datos II, DC, UBA.

Primer cuatrimestre 2020

Índice

1. Sorting

1.1. Ejercicio 1

Prestemos atención al enunciado:

1. Ordenar los datos por frecuencia
2. Luego por valor

Acá lo ideal es utilizar un algoritmo para ordenar por valor y luego por frecuencia utilizando un algoritmo estable. Porque digo esto:

- Observo que con frecuencia = 1 hay que hacer un algoritmo de sorting tradicional... o sea, no podemos evitar este peor caso
- Una vez ordenado, se puede utilizar simplemente un bucket sort ya que la frecuencia de valor va de entre 0 y n

Dicho lo anterior propongo este algoritmo:

1. Ordenar los elemengos con algún algoritmo de sorting en $O(n \log n)$
2. Utilizar bucket sort de la siguiente forma:
   a) Crear n buckets
   b) Mientras que se repita el elemento en el array ir contando, cuando terminás de contar cuantos elementos hay de cada número los insertás en su correspondiente bucket (o directamente poner un nodo que diga hay x veces tal número)
   c) Para finalizar recorrer todos los buckets e imprimir el array ordenado como piden

Complejidad total: $O(n \log n + n) = O(n \log n)$

1.2. Ejercicio 2

Como siempre, hay que prestar atención en los datos *especiales* que contamos acerca del arreglo:

- Hay un rango, y con esto empezamos a pensar en un algoritmo de sorting de complejidad acorde a dicho rango, por ejemplo counting sort
- Hay a lo sumo $\lfloor \sqrt{n} \rfloor$ elementos fuera del rango

Entonces veamos, ordenar los elementos en rango es $O(n + k)$ con counting sort, pero este k sería constante así que listo, nos sacamos todos los elementos en ese rango de encima con la complejidad pedida. Acerca de los otros elementos no tenemos ningún dato... es decir, no podemos aprovechar ninguno de los algoritmos que pueden ordernar en menor complejidad que $O(n \log n)$.
Por lo tanto, evaluemos si utilizar un algoritmo general nos sirve.

Utilizar heap sort para a lo sumo $\lfloor \sqrt{n} \rfloor$ cuesta $O(\sqrt{n} \log \sqrt{n})$ y como $log\sqrt{n} <= \sqrt{n}$ para $n \in N$ con $n > 0$ (que es lo que nos sirve), podemos ver que $O(\sqrt{n} \log \sqrt{n}) \in O(n)$.

Por lo tanto el algoritmo nos quedaría:

1. En una pasada hacer counting sort sobre los elementos comprendidos en el rango, para los fuera de ese rango los separamos (en un único grupo o en grupos superiores o menores al rango es lo mismo en cuanto a complejidad teórica) en $O(n)$
2. Los elementos separados se ordenan con heap sort (u otro algoritmo que cueste lo mismo) en $O(n)$
3. Devolver al final todos los grupos (inferiores al rango, dentro del rango y superiores al rango) en $O(n)$

Complejidad total: $O(n)$

1.3. Ejercicio 3

Nuevamente prestemos atención a que tenemos que ordenar y con cuales datos contamos:

- Tenemos que ordenar tuplas de nat, string donde los strings son de longitud máxima l
- La comparación de nat toma $O(1)$ pero la de strings $O(l)$...
- Tenemos que ordenar primero por el 2do componente (el string) y en caso de empate el primer componente (el nat)

Para el punto a) nos piden complejidad temporal $O(nl + n \log n)$, entonces:

- Cuando nos piden ordenar por un componente y luego por el otro, lo mejor es ordenar en orden inverso al que nos piden (ante la duda ver como funciona Radix Sort)
- Entonces primero ordenemos por el primer componente, como no tenemos datos de los números utilicemos cualquier algoritmo que cueste $O(n \log n)$ por ejemplo Heap Sort, ahora vayamos por el ordenamiento por string
- Si intentara utilizar un algoritmo convencional con el costo de comparación $O(l)$ me quedaría $O(ln \log nl)$ lo cual excede lo que nos piden... entonces recurramos a una estructura conocida: un diccionario implementado con un trie
- Si utilizamos esto, insertar cada elemento en el trie es $O(l)$, insertar todos es $O(nl)$ como nos pide el ejercicio
- No olvidemos que este algoritmo tiene que ser estable para mantener el ordenamiento previo, por lo que este algoritmo se tendría que comportar como un bucket sort teniendo una lista enlazada como significado del diccTrie
- Una vez distribuidos todos los elementos, con un iterador, iterar las a lo sumo n claves del diccionario en $O(n)$ y copiar todos los elementos al array resultante en $O(nl)$

Costo final: $O(nl + n \log n)$

Para el punto b) nos dicen que los naturales de la primer componente están acotados y nos piden una complejidad temporal de $O(nl)$ en el peor caso.

En este caso vamos a utilizar el mismo algoritmo anterior, pero en el punto donde utilizamos un algoritmo genérico para ordenar en base a los números utilicemos uno que aproveche este nuevo dato, por ejemplo, bucket sort, donde hay una cantidad de buckets acotados por la constante que dice el enunciado que existe.

Dicho esto, la complejidad total sería: $O(nl)$

1.4. Ejercicio 4

Tenemos un arreglo de enteros sin repetidos $A = [A_1 \dots A_n]$, el cual cumple la propiedad en donde para cada A_i la cantidad de elementos del arreglo (estrictamente) menores es a lo sumo i.
Pensemos primero en el arreglo ordenado resultante $B = [B_1 \dots B_n]$. Dado un A_i nos podemos preguntar en que posición j puede quedar en el arreglo B, en particular: ¿podría pasar que el $j > i + 1$? si esto fuese así entonces significaría que hay por lo menos $j - 1 > i$ elementos menores que A_i pues B está ordenado. Esto nos permite concluir que los elementos de A no pueden quedar "muy adelante" en la versión ordenada.
Sabiendo que un elemento A_i queda a lo sumo en la posición $i + 1$, un algoritmo que se viene a la mente para ordenar es *insertion sort* ya que en este los elementos se van "moviendo hacia atras". El problema es que este algoritmo en peor caso es $O(n^2)$, esto pasa porque para cada índice i en el peor caso hay que moverlo $i - 1$ veces para atras. Teniendo en cuenta la propiedad que remarcamos antes ¿podría darse este caso para el arreglo A?
Por lo que mencionamos antes sobre que un elemento no puede ser movido "muy adelante" derivamos que si en alguna iteración un elemento fue cambiado de lugar, entonces en ninguna otra iteración esto volverá a pasar. Si esto pasase, la posición final del elemento sería por lo menos su posición original más dos, y como ya dijimos esto es imposible.
Entonces, para un arreglo con esta propiedad, la cantidad de cambios que realiza *insertion sort* en peor caso es exactamente un cambio por elemento, lo cual resulta en una complejidad de $O(n)$

1.5. Ejercicio 5

Este es un ejercicio pensado para profundizar acerca de Radix Sort (o bien, Bucket Sort por dígitos...). Vayamos viendo punto por punto:

- En el punto a) sabemos que hay un arreglo de n enteros positivos menores que n, para ordenarlos podemos utilizar un bucket sort o un counting sort, hasta acá nada nuevo. La complejidad sería de $O(n)$
- La pregunta b) está orientada a profundizar Radix Sort. La idea es la siguiente:
  - Un número que esté acotado por n^2 se puede representar en base n con 2 cifras. Es decir, cada cifra del número tiene n posibilidades.
  - Piensen lo siguiente, en base 10 podemos escribir con 1 cifra 10 elementos, con 2 cifras 10^2 elementos. Entonces en base n podemos escribir n^2 con 2 cifras.
  - Dicho esto, un algoritmo que cambie de base estos elementos permitiría utilizar un radix sort en $O(n)$.
  - Para más info y algoritmo completo: https://www.geeksforgeeks.org/sort-n-numbers-range-0-n2-1-linear-time/
- En este caso se utiliza el algoritmo anterior, básicamente haríamos k bucket sorts, por lo que el costo sería $O(k * n)$
- Si los números no están acotados no hay ninguna relación entre n y la cantidad de dígitos que se necesitan ni nada, por lo que habría que utilizar un algoritmo genérico. Para más info hay una buena respuesta que encontré acá: https://cs.stackexchange.com/questions/5030/why-is-radix-sort-on#:~:text=In%20general%2C%20the%20complexity%20

2. Divide and Conquer

2.1. Ejercicio 1

Vemos que la definición misma de "ser más a la izquierda" nos está describiendo las dos componentes necesarias para resolver con *divide & conquer*:

- Subproblemas: como cada mitad del arreglo debe ser "ser más a la izquierda", tenemos dos subproblemas: el de la mitad izquierda del arreglo y el de la mitad derecha.
- Conquer: Si se cumple que tanto la mitad izquierda como la derecha cumplen la propiedad, la manera de determinar si el arreglo en total es "más a la izquierda" es computando las sumas y comparandolas.

Sabiendo que poseemos la propiedad de que el problema lo podemos dividir en subproblemas y luego tenemos como unirlos, derivemos un algoritmo.
Pensemos que el problema es sobre un arreglo de tamaño n, entonces cada vez que tomamos un subproblema es de tamaño $n/2$ lo cual nos asegura que es un problema más chico. Luego el caso base (como n es potencia de 2) es $n = 1$, el cual cumple la propiedad de manera trivial.
Para evitar pasar arreglos enteros por copia en cada llamado recursivo, vamos a implementarlo pasandole como parametros a la función los indices de inicio y final a chequear la propiedad. De esta manera la pregunta que nos hacemos es si el arreglo es más a la izquierda entre $(0, n)$. Luego el algoritmo que nos queda sería:

Algorithm 1 Divide & Conquer: más a la izquierda

```
1:  procedure esDeIzquierda?(A, desde, hasta)
2:      if desde == hasta then
3:          return True                                                    ▷ O(1)
4:      else
5:          mitad ← (desde + hasta)/2
6:          izqEsIzquierdista ← esDeIzquierda?(A, desde, mitad)            ▷ O(T(n/2))
7:          derEsIzquierdista ← esDeIzquierda?(A, mitad, hasta)            ▷ O(T(n/2))
8:          izquierdaSumaMas ←  (suma(A, desde, mitad) ≥ suma(A, mitad, hasta))   ▷ O(n)
9:          return IzqEsIzquierdista and DerEsIzquierdista and izquierdaSumaMas   ▷ O(1)
10:     end if
11: end procedure
```

Luego podemos escribir la función de complejidad de manera recursiva:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(n) & \text{Caso contrario} \end{cases}$$

Que por el caso 2 del teorema maestro sabemos que es $\Theta(nlog(n))$.
Hay una observación por hacer para el algoritmo propuesto: en cada paso estamos calculando la suma de ambas partes, pero de alguna forma ese calculo ya "fue hecho" en cada paso recursivo. Por ejemplo cuando se hace el llamado para la mitad izquierda efectivamente se computa toda la suma de esa mitad (de manera implicita) al calcular sus dos sumas parciales. Esto nos muestra que podríamos de alguna manera "pasar" esta información del caso recursivo al caso que lo llamó.
Una manera de hacer esto es que el algoritmo devuelva dos cosas (en una tupla por ejemplo): $\langle esDeIzquierda, sumaDelArreglo \rangle$. Es fácil adaptar el código anterior para que efectivamente devuelva esto y aprovecharlo:

Algorithm 2 Divide & Conquer: más a la izquierda

```
1:  procedure esDeIzquierda?(A, desde, hasta)
2:      if desde == hasta then
3:          return ⟨True, A[desde]⟩                                          ▷ O(1)
4:      else
5:          mitad ← (desde + hasta)/2
6:          izqEsIzquierdista, sumaIzq ← esDeIzquierda?(A, desde, mitad)     ▷ O(T(n/2)
7:          derEsIzquierdista, sumaDer ← esDeIzquierda?(A, mitad, hasta)     ▷ O(T(n/2)
8:          izquierdaSumaMas ←     (sumaIzq ≥ sumaDer)                       ▷ O(1)
9:          esDeIzquierda ← IzqEsIzquierdista and DerEsIzquierdista and izquierdaSumaMas
10:         return ⟨esDeIzquierda, sumaIzq + sumaDer⟩                        ▷ O(1)
11:     end if
12: end procedure
```

Aclaración sintáctica: al poner $a, b \leftarrow F(x)$ nos referimos a descomponer el resultado de la tupla que devuelve $F(X)$ en sus dos valores pero en variables diferentes.

De esta manera bajamos la complejidad a:

$$T(n) = \begin{cases} O(1) & n = 1 \\ 2T(n/2) + O(1) & \text{Caso contrario} \end{cases}$$

Que por caso 1 del teorema maestro es $\Theta(n)$

2.2. Ejercicio 2

Tenemos que computar la suma:

$$A^1 + A^2 + \dots A^n = \sum_i^n A^i$$

Y nos piden que lo realizemos utilizando D&C. En general para esta técnica algoritmica pensamos de manera recursiva y por lo tanto hay dos partes a determinar:

- Caso base: cuando $n = 1$, en este caso es simplemente devolver la matriz A.
- Paso recursivo: partimos el problema en subproblemas más chicos y los unimos.

En general para estos algoritmos buscamos "partir por la mitad", pero para una sumatoria de potencia de matrices: ¿qué sería partir por la mitad?
Lo primero es recordar que tenemos algunas propiedades sobre la multiplicación de matrices y sus potencias:

- $A(B + C) = AB + AC$
- $A^i \times A^j = A^{i+j}$

Para partir a la mitad, pensemos en partir la suma en los primeros $n/2$ terminos y el resto:

$$\sum_{i=1}^{n} A^i = \sum_{i=1}^{n/2} A^i + \sum_{i=n/2}^{n}$$

Notemos que, como n es potencia de 2, entonces el primer sumando es exactamente un subproblema de la mitad de terminos. Además, aplicando la segunda propiedad mencionada:

$$\sum_{i=1}^{n/2} A^i + \sum_{i=n/2}^{n} = \sum_{i=1}^{n/2} A^i + A^{n/2} \sum_{i=1}^{n/2} A^i$$

Si llamamos "sumPots" a la función que queremos computar para un n potencia de 2 y una matriz A entonces:

$$\text{sumPots}(n, A) = \begin{cases} A & n = 1 \\ sumPots(\frac{n}{2}, A) + A^{\frac{n}{2}} * sumPots(\frac{n}{2}, A) & \text{sino} \end{cases}$$

Y por lo tanto obtuvimos una expresión recursiva del problema. Con esta escritura queda explicito que la parte de dividir es calcular el subproblema para $n/2$ y la parte del conquer se da al combinar este subproblema con la suma y la multiplicación por $A^{n/2}$. Es muy fácil a partir de esta expresión recursiva derivar el algoritmo:

Algorithm 3 Divide & Conquer: suma de potencias

```
1: procedure SUMAPOTENCIASDC(A, n)
2:     if n == 1 then
3:         return A                                              ▷ O(1)
4:     else
5:         D ← sumaPotenciasDC(A, n/2)                           ▷ O(T(n/2))
6:         return D + potencia(n/2) × D                          ▷ O(1)
7:     end if
8: end procedure
```

Como en cada paso recursivo se divide n por 2, entonces la cantidad de llamados recursivos resultan ser $log_2(n)$ y como en cada paso se hace un único llamado al método potencia, la cantidad de veces que se realiza el llamado es $O(log(n))$.
Usando además la asumpción $A \in \mathcal{Z}^{4\times4}$ entonces tanto la multiplicación, suma y potencia de matrices es constante. Si escribimos la función de tiempo en peor caso de manera recursiva:

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(n/2) + O(1) & \text{Caso contrario} \end{cases}$$

Que por caso 2 del teorema maestro podemos concluir que $T(n) \in \Theta(log(n))$

2.3. Ejercicio 3

Tanto como las listas, los arboles binarios tienen una estructura inherentemente recursiva. Queremos saber la máxima distancia entre dos nodos del arbol, lo cual es equivalente al tamaño del camino más largo entre dos nodos. Este camino tiene tres opciones:

1. El camino se encuentra completamente contenido en el subarbol izquierdo.
2. El camino se encuentra completamente contenido en el subarbol derecho.
3. El camino pasa por el nodo raiz del arbol.

Si desglosamos lo que significa cada uno de estos casos llegamos a la conclusión que tanto el primero como el segundo es el mismo problema pero sobre cada subarbol hijo. Por su parte, el tercer caso corresponde a la parte de *conquer* del algoritmo y por lo tanto es lo ultimo que necesitamos determinar para poder terminar de definirlo.
Por definición la altura de un arbol es la máxima distancia de la raiz a una hoja. Podemos entonces concluir que el camino más largo que incluya a la raiz del arbol será: desde alguna hoja del arbol izquierdo que maximize su altura, pasando por la raiz, yendo hasta una hoja que maximize la altura del arbol derecho.
El enunciado nos pide que no hagamos recorridos innecesarios sobre el arbol, lo cual se parece a lo que marcamos en la resolución del ejercicio 1. El objetivo es darnos cuenta que no hace falta calcular la altura del arbol izquierdo y del derecho, porque estos valores son implicitamente calculados en cada llamado recursivo a los subproblemas. Entonces, como hicimos en el ejercicio 1, haremos que nuestro algoritmo devuelva una tupa: $\langle maxDistancia, altura \rangle$ Donde *maxDistancia* es la solución al problema y *altura* es la altura del arbol. Además, para evitar copias, haremos que el algoritmo se llame sobre un puntero a Nodo de arbol binario.
Asumiendo que la cantidad de nodos totales en el arbol son n y la cantidad de nodos en su subarbol izquierdo es I (y por lo tanto la cantidad de nodos en el derecho es $D = n - I$)

Algorithm 4 Divide & Conquer: maxima distancia en AB

```
1: procedure MAXDISTANCIA(nodo)
2:     if nodo == nil then
3:         return ⟨0, 0⟩                                                          ▷ O(1)
4:     else
5:         maxIzq, alturaIzq ← maxDistancia(nodo.izq)                             ▷ O(T(I))
6:         maxDer, alturaDer ← maxDistancia(nodo.der)                             ▷ O(T(D))
7:         return ⟨max(maxIzq, maxDer, alturaIzq + 1 + alturaDer), max(alturaIzq, alturaDer) + 1⟩   ▷ O(1)
8:     end if
9: end procedure
```

En la ultima linea del algoritmo queda claro que:

- El camino más largo es el más largo entre las tres opciones que mencionamos al principio
- la altura del arbol es el máximo entre la altura del subarbol izquierdo y el derecho, más uno porque se suma la raiz.

Luego la complejidad del algoritmo es:

$$T(n) = \begin{cases} O(1) & n = 1 \\ T(I) + T(D) + O(1) & \text{Caso contrario} \end{cases}$$

Ahora tenemos un problema, porque la función del peor caso no cumple la estructura necesaria para aplicar el teorema maestro. Esto pasa porque no sabemos si el arbol está balanceado: si esto fuese así entonces I, D $\frac{n}{2}$ y concluiriamos que el algoritmo es $O(n)$.
De todas formas por como escribimos el algoritmo podemos tener la intuición que cada nodo se recorre una única vez y eso podría indicarnos que el algoritmo tendrá una complejidad de $O(n)$. Dicho esto, procedemos a probarlo por inducción.
Queremos ver que $T(n) \in O(n)$ $\forall n \in \mathcal{N}$

Caso base: $n = 1$: como $n = 1 \Rightarrow T(n) = O(1) = O(n)$
Paso inductivo: asumiendo que vale para todo $n_0 < n$ que $T(n_0) \in O(n_0)$ veamos que $T(n) \in O(n)$.

Sabemos que $T(n) = T(I) + T(D) + O(1)$. Como tanto $I, D < n$ entonces para cada uno vale la hipótesis inductiva:

- $T(I) = a_i I + b_i$
- $T(D) = a_d D + b_d$

Sin perdida de generalidad asumamos que $a_i \geq a_d$. y por lo tanto:

$$T(n) = (a_i I + b_i) + (a_d D + b_d) + k = (a_i I + a_d D) + (b_i + b_d + k) \leq a_i(I + D) + (b_i + b_d + k) = a_i n + (b_i + b_d + k) \in O(n)$$

Concluimos entonces que el algoritmo tiene complejidad $T(n) \in O(n)$

2.4. Ejercicio 4

Arraquemos por lo fácil: En la parte de Dividir vamos a separar al arreglo en dos, y en la parte del Conquer vamos a aplicar recursión a izquierda y a derecha. Quedaría hacer el Combine. La pregunta es, ¿con qué información contamos luego de aplicar recursión? Bueno, tenemos cuántas parejas desordenadas hay en la parte izquierda, y cuantas parejas desordenadas hay en la parte derecha. Lo que nos falta contar son las parejas desordenadas formadas por números entre distintas partes, o sea, formadas por un número de la izquierda con un número de la derecha pero más chico.

A priori esto es dificil de hacer sin un algoritmo cuadrático, lo que nos va a arruinar la complejidad que nos piden. Ahí entra en juego la pista del enunciado, que es partir de Merge Sort. O sea, la idea va a ser que nuestro algoritmo no solo devuelva la cantidad de parejas desordenadas, sino que también ordene el arreglo. La gracia es que en el Combine vamos a poder asumir que la parte izquierda y la parte derecha van a estar ordenadas. A continuación una solución que aprovecha este hecho:

ParejasDesordenadas(**in** v : `vec`, **in** *desde* : `nat`, **in** *hasta* : `nat`) $\rightarrow$ *res* : nat

```
1: ParejasDesordenadasRec(Copiar(v), 0, Longitud(v))
```

ParejasDesordenadasRec(**in/out** v : `vec`, **in** *desde* : `nat`, **in** *hasta* : `nat`) $\rightarrow$ *res* : nat

```
1: if hasta − desde <= 1 then                                              ▷ O(1)
2:     res ← 0                                                             ▷ O(1)
3: else
       // Divide
4:     medio ← (desde + hasta)/2                                           ▷ O(1)
       // Conquer
5:     resIzq ← ParejasDesordenadas(v, desde, medio)                       ▷ T(n/2)
6:     resDer ← ParejasDesordenadas(v, medio, hasta)                       ▷ T(n/2)
       // Combine
7:     resMed ← 0                                                          ▷ O(1)
8:     i ← desde // índice de la izquierda                                 ▷ O(1)
9:     j ← medio // índice de la derecha                                   ▷ O(1)
       // Para cada nat en la izquierda, contamos cuantos en la derecha son más chicos
10:    while i < medio do                                                  ▷ O(n)*
11:        while j < hasta ∧ v[i] > v[j] do
12:            j++
13:        end while
           // Como la parte derecha está ordenada, y como avanzamos j mientras v[i] fuera
           // mayor, todos los números en el rango [medio, j) son más chicos que v[i]
14:        resMed ← resMed + (j − medio)
           // Como v[i] ≤ v[i+1], entonces v[i+1] también es mayor que los números en [medio, j),
           // y podemos usar el mismo j para la siguiente iteración
15:        i++
16:    end while
17:    Merge(v)                                                            ▷ O(n)
18:    res ← resIzq + resDer + resMed                                      ▷ O(1)
19: end if
```

* Observar que en el ciclo externo nunca se resetea el valor de j, con lo cual el ciclo interno va a iterar en total a lo sumo $hasta - medio$ veces, realizando operaciones $O(1)$ cada vez. Por ende, el ciclo interno aporta solo $O(n)$ a la complejidad total del ciclo externo. Además, en el ciclo externo realizamos algunas operaciones $O(1)$, lo cual como el ciclo itera $medio - desde$ veces, es $O(n)$. Entonces, podemos concluir que el ciclo externo es $O(n)$.

La complejidad entonces nos queda $T(n) = T(n/2) + O(n)$, que es igual a la de merge sort, $O(n \log n)$, que cumple lo requerido por el enunciado.