Algoritmos y Estructuras de Datos 2

Práctica 5: Ordenamiento

1er cuatrimestre 2022

Índice

1. Ejercicio 1

Pendiente.

2. Ejercicio 2: una parte ya ordenada

Nos conviene usar algún algoritmo de ordenamiento que tenga como invariante que en la iteración i-ésima, ya ordenamos la secuencia hasta la posición i-ésima. De esta manera podemos comenzar a ordenar s desde la posición $i = tam(s')$. Por ejemplo, podríamos usar InsertionSort.

3. Ejercicio 3: k más chicos

BuscarMinimos(**in** A: arreglo(nat), **in** k: nat) → **out** res: arreglo(nat)

```
1:  res ← CrearArreglo(k)                                              ▷ O(k)
2:  max ← BuscarMax(A)                                                 ▷ O(n)
3:  for i ← 1 to k do                                                  ▷ O(k)
4:      res[i] ← max
5:  end for
6:  for i ← 1 to k do                                                  ▷ O(k)
7:      for j ← 1 to n do                                              ▷ O(n)
8:          if A[j] < res[i] ∧ (i = 1 ∨L A[j] > res[i-1]) then
9:              res[i] ← A[j]
10:         end if
11:     end for
12: end for
```

Complejidad: $O(kn)$

Cuando $k > log(n)$ conviene primero ordenar el arreglo, por ejemplo con MergeSort que tiene complejidad $O(nlog(n))$, y luego simplemente copiar en res los primeros k elementos que tiene complejidad $O(k)$.

4. Ejercicio 4: n secuencias ordenadas

UnirOrdenados(**in** A: arreglo(arreglo(nat))) → **out** res: arreglo(nat)

```
1: if tam(A) = 1 then
2:     res ← A[1]
3: else
4:     res ← Merge(
5:       UnirOrdenados(PrimeraMitad(A)),
6:       UnirOrdenados(SegundaMitad(A))
7:     )
8: end if
```

Complejidad: $O(mlog(n))$ donde m es la cantidad total de elementos en todas las n secuencias.

Inicialmente hay n secuencias en A, y en cada paso recursivo dividimos A en 2 partes iguales y resolvemos esos 2 subproblemas de forma recursiva. El caso base es cuando $n = 1$. Si llamamos i al total de pasos recursivos, $n/2^i = 1 \iff i = log(n)$. Es decir, vamos a hacer en total $log(n)$ pasos recursivos.

El costo del Merge es $O(m)$ donde m es la cantidad total de elementos en todas las n secuencias. En cada paso recursivo siempre tenemos que iterar el total de m elementos para poder hacer el Merge entre las secuencias de ese nivel de la recursión (si bien hay cada vez menos secuencias que mergear, la cantidad total de elementos es constante).

Por lo tanto, la complejidad total está dada por $O(mlog(n))$, ya que vamos a hacer un Merge entre m elementos, $log(n)$ de veces.

5. Ejercicio 5: frecuencia

OrdenarPorFrecuencia(**in/out** A: arreglo(nat))

```
1: MergeSort(A)                                                          ▷ O(nlog(n))
2: B: arreglo(⟨ nat, nat ⟩) ← Compactar(A)                               ▷ O(n), porque A ya está ordenado
3: MergeSort(B)          ▷ O(nlog(n)), clave de ordenamiento: segunda componente (repeticiones), decreciente
4: A ← Expandir(B)                                                       ▷ O(n)
```

Complejidad: $O(nlog(n))$

Compactar(**in** A: arreglo(nat)) → **out** res: arreglo(⟨ nat, nat ⟩)

Pre ≡ {estáOrdenado(A)}

```
1: B: lista(⟨ nat, nat ⟩) ← CrearLista()                                 ▷ O(1)
2: n ← tam(A)
3: i ← 0
4: while i < n do                                                        ▷ O(n)
5:     c ← 1
6:     j ← i + 1
7:     while j < n ∧_L A[j] = A[i] do
8:         c ← c + 1
9:         j ← j + 1
10:    end while
11:    AgregarAtras(⟨ A[i], c ⟩)                                         ▷ O(1)
12:    i ← j
13: end while
14: res ← ListaToArreglo(B)                                              ▷ O(tam(B)) = O(n)
```

Complejidad: $O(n)$

Expandir(**in** A: arreglo(⟨ nat, nat ⟩)) → **out** res: arreglo(nat)

```
1: B: lista(nat) ← CrearLista()                                          ▷ O(1)
2: for i ← 0 to tam(A) do                                                ▷ O(tam(A))
3:     j ← A[i][1]
4:     while j > 0 do                          ▷ O(r) donde r son las repeticiones del elemento A[i][1]
5:         AgregarAtras(B, A[i][0])                                      ▷ O(1)
6:         j ← j - 1
7:     end while
8: end for
9: res ← ListaToArreglo(B)                                               ▷ O(n)
```

Complejidad: $O(n)$

Ejemplo de ejecución

```
A = [1, 3, 1, 7, 2, 7, 1, 7, 3]
A = [1, 1, 1, 2, 3, 3, 7, 7, 7]
B = [(1, 3), (2, 1), (3, 2), (7, 3)]
B = [(1, 3), (7, 3), (3, 2), (2, 1)]
A = [1, 1, 1, 7, 7, 7, 3, 3, 2]
```

6. Ejercicio 6: escaleras

Hacemos una primera iteración por el arreglo formando tuplas con las posiciones de inicio y fin de cada escalera.

El peor caso es que tengamos todas escaleras de un solo elemento. En este caso, el arreglo de tuplas tendrá exactamente el mismo tamaño que el arreglo original, y todas las tuplas tendrán la misma posición de inicio y fin.

Luego, ordenamos el arreglo de tuplas 2 veces utilizando algún algoritmo estable:

1. Ordenamos las tuplas según el primer elemento de la escalera. Para obtener el primer elemento de la escalera, indexamos el arreglo original con el elemento en la primer posición de la tupla (el inicio de la escalera).

2. Ordenamos las tuplas según el tamaño de la escalera. Para obtener el tamaño de la escalera, restamos el segundo elemento de la tupla con el primero y sumamos 1 porque los índices son inclusivos (posición de fin de la escalera - posición de inicio + 1).

Finalmente, para poder acomodar el arreglo según lo pedido, necesitamos primero copiar el arreglo original en otro arreglo copia para poder indexarlo en las posiciones calculadas (si intentamos indexar sobre el arreglo original mientras vamos cambiando los elementos, eventualmente vamos a obtener un elemento equivocado). Entonces iteramos por las tuplas, y por cada tupla, iteramos por las posiciones entre inicio y fin (inclusives) de la escalera para ir copiando los elementos en esas posiciones del arreglo copia hacia la posición actual en el arreglo original. Cada vez que copiamos un elemento incrementamos la posición actual en 1.

Al terminar de iterar por todas las tuplas, habremos copiado todos los elementos del arreglo copia en el original, pero en sus posiciones correctas.

Se asume que todos los arreglos se indexan desde 1.

OrdenarEscaleras(**in/out** A: arreglo(nat))

```
1: B: arreglo(⟨ nat, nat ⟩) ← ObtenerEscaleras(A)        ▷ O(n)
2: MergeSort(B, A)                                       ▷ O(nlog(n)), clave de ordenamiento: A[B[i][0]]
3: MergeSort(B)                                          ▷ O(nlog(n)), clave de ordenamiento: B[i][1] - B[i][0] + 1, decreciente
4: CopiarEscaleras(A, B)                                 ▷ O(n)
```

Complejidad: $O(nlog(n))$

ObtenerEscaleras(**in** A: arreglo(nat)) → **out** res: arreglo(⟨ nat, nat ⟩))

```
 1: B: list(⟨ nat, nat ⟩) ← CrearLista()                 ▷ O(1)
 2: n ← tam(A)
 3: i ← 1
 4: while i ≤ n do                                       ▷ O(n)
 5:     j ← i
 6:     while j < n ∧_L A[j + 1] = A[j] + 1 do           ▷ O(n − i)
 7:         j ← j + 1
 8:     end while
 9:     AgregarAtras(B, ⟨ i, j ⟩)                        ▷ O(1)
10:     i ← j + 1
11: end while
12: res ← ListaToArreglo(B)                              ▷ O(tam(B)) = O(n)
```

Complejidad: $O(n)$

Al incrementar i de esta forma, en efecto nos salteamos en el while exterior los elementos que ya recorrimos en el while interior.

CopiarEscaleras(**in/out** A: arreglo(nat), **in** B: arreglo(⟨ nat, nat ⟩))

```
1: C: arreglo(nat) ← Copiar(A)                           ▷ O(n)
2: k ← 1
3: for i ← 1 to tam(B) do                                ▷ O(n)
4:     for j ← B[i][0] to B[i][1] do
5:         A[k] ← C[j]
6:         k ← k + 1
7:     end for
8: end for
```

Complejidad: $O(n)$

La complejidad de los ciclos de CopiarEscaleras es $O(n)$ por lo siguiente:

Si todas las escaleras son de un solo elemento, entonces $tam(B) = tam(A)$, $B[i][0] = B[i][1] \forall i$, y por lo tanto el while interior hace una única iteración y tiene complejidad $O(1)$. El while exterior haría $tam(B) = tam(A) = n$ iteraciones y por lo tanto la complejidad de ambos ciclos sería $O(n) * O(1) = O(n)$.

En el extremo opuesto, si todo el arreglo original A ya era una escalera, en B tendríamos una única tupla $\langle 1, tam(A) \rangle$, el while exterior haría una única iteración pero el while interior haría $B[i][1] - B[i][0] + 1 = tam(A) - 1 + 1 = tam(A) = n$ iteraciones. En efecto, la complejidad de ambos ciclos sería $O(1) * O(n) = O(n)$.

Partiendo del caso donde son todas escaleras de un solo elemento, por cada elemento que en realidad es parte de una escalera, vamos a tener una tupla menos, y alguna otra tupla va a tener un rango incrementado en 1. En efecto, el while exterior haría una iteración menos pero el while interior haría una iteración más, y la complejidad final seguirá siendo $O(n)$.

Ejemplo de ejecución

```
A = [5, 6, 8, 9, 10, 7, 8, 9, 20, 15]
B = [(1, 2), (3, 5), (6, 8), (9, 9), (10, 10)]
B = [(1, 2), (6, 8), (3, 5), (10, 10), (9, 9)]
B = [(6, 8), (3, 5), (1, 2), (10, 10), (9, 9)]
A = [7, 8, 9, 8, 9, 10, 5, 6, 15, 20]
```

7. Ejercicio 7: AVL Sort

Se usa el operador [] para buscar/obtener las claves (elementos de A) que están en el diccionario. Al acceder a una clave se devuelve su significado (repeticiones) si está definida. Caso contrario, se define automáticamente inicializando el significado en el valor default de su tipo de datos (en este caso al ser Nat, inicializamos en 0).

Las operaciones sobre el AVL cuestan $O(log(d))$, con d = elementos únicos de A.

El iterador del diccionario recorre el AVL *inOrder* produciendo las claves ordenadas de menor a mayor en $O(d)$. La suma de todas las repeticiones es n.

AVLSort(**in/out** A: arreglo(nat))

```
1:  D: diccAVL(nat, nat) ← Vacio()                          ▷ O(1)
2:  for i ← 1 to tam(A) do                                  ▷ O(nlog(d))
3:      D[A[i]] ← D[A[i]] + 1                               ▷ O(log(d))
4:  end for
5:  i ← 1                                                   ▷ O(1)
6:  it ← CrearIt(D)                                         ▷ O(1)
7:  while HaySiguiente(it) do                               ▷ O(n)
8:      e ← SiguienteClave(it)                              ▷ O(1)
9:      r ← SiguienteSignificado(it)                        ▷ O(1)
10:     while r > 0 do                                      ▷ O(r)
11:         A[i] ← e                                        ▷ O(1)
12:         i ← i + 1                                       ▷ O(1)
13:         r ← r - 1                                       ▷ O(1)
14:     end while
15: end while
```

Complejidad: $O(nlog(d))$

8. Ejercicio 8: números repetidos continuos

Pendiente.

9. Ejercicio 9: alumnos

9.a.

OrdenarPlanilla(**in/out** P: arreglo(alumno))

```
1:  CountingSort(P)     ▷ O(n), clave de ordenamiento: alumno.puntaje, orden creciente
2:  CountingSort(P)     ▷ O(n), clave de ordenamiento: alumno.genero, orden decreciente
```

Complejidad: $O(n)$

9.b.

Igual que la solución anterior ya que si los géneros están acotados, podemos hacer CountingSort ordenando por alumno.genero en tiempo lineal $O(n)$ donde n es la cantidad de alumnos en la planilla.

9.c.

El lower bound $\Omega(nlog(n))$ solo aplica a algoritmos de ordenamiento basadas en comparaciones. No estamos contradiciendo ese lower bound porque el algoritmo de ordenamiento utilizado, CountingSort, no se basa en comparaciones.

10. Ejercicio 10: casiSort

Pendiente.

11. Ejercicio 11: en rango

Pendiente.

12. Ejercicio 12: sensor industrial

Pendiente.

13. Ejercicio 13: tuplas (nat, string)

13.a.

Ordenar(**in/out** A: arreglo($\langle$ c_1: nat $\times$ c_2: string[l] $\rangle$))

1: RadixSort(A) ▷ $O(nl)$, clave de ordenamiento: c_2
2: MergeSort(A) ▷ $O(nlog(n))$, clave de ordenamiento: c_1

Complejidad: $O(nl + log(n))$

Primero ordenamos el arreglo A por la segunda componente, el string de longitud máxima l, utilizando RadixSort que tiene complejidad $O(l(n + k))$, donde l es el largo máximo del string, n la cantidad de elementos en A, y k la cantidad máxima posible de caracteres para cada posición del string. Considerando que el string contiene únicamente caracteres ascii, $k = 256$ y es una constante. Además, utilizamos CountingSort como algoritmo estable de ordenamiento para cada posición i-ésima del string, y por lo tanto su complejidad resulta $O(n)$ (pues k es constante). Finalmente, todo el RadixSort tiene complejidad $O(nl)$ ya que debemos repetir el CountingSort por cada posición del string de largo máximo l (no podemos asumir que l es una constante).

Luego ordenamos el arreglo A por la primer componente de la tupla, el número natural, utilizando MergeSort. Este algoritmo tiene complejidad $O(nlog(n))$ y es estable (necesitamos que sea estable para mantener el orden previo realizado por RadixSort).

De esta forma la complejidad resultante del algoritmo es $O(nl + nlog(n))$ en el peor caso.

13.b.

Ordenar(**in/out** A: arreglo($\langle$ c_1: nat $\times$ c_2: string[l] $\rangle$))

1: RadixSort(A) ▷ Ordena por c_2: $O(nl)$
2: CountingSort(A) ▷ Ordena por c_1: $O(n)$

Al saber que la primer componente de las tuplas son naturales acotados por una constante, podemos utilizar CountingSort (que también es estable) para realizar el segundo ordenamiento en $O(n)$.

De esta forma la complejidad resultante del algoritmo es $O(nl + n) = O(nl)$ en el peor caso.

14. Ejercicio 14: múltiplos

OrdenarMultiplos(**in** A: arreglo(nat), **in** k: nat) → **out** res: arreglo(nat)

1: MergeSort(A) ▷ $O(nlog(n))$
2: Ak: arreglo(arreglo(nat)) ← CrearArreglo(k) ▷ $O(k)$
3: **for** i ← 1 **to** k **do** ▷ $O(k)$
4: Ak[i] ← MultiplicarTodos(A, i) ▷ $O(n)$
5: **end for**
6: res ← UnirOrdenados(Ak) ▷ $O(nklog(k)) = O(nklog(n))$

Complejidad: $O(nklog(n))$

MultiplicarTodos(**in** A: arreglo(nat), **in** k: nat) → **out** res: arreglo(nat)

1: n ← tam(A)
2: res: arreglo(nat) ← CrearArreglo(n) ▷ $O(n)$
3: **for** i ← 1 **to** n **do** ▷ $O(n)$
4: res[i] ← A[i] * k
5: **end for**

Complejidad: $O(n)$

Primero ordenamos el arreglo de entrada A. Luego creamos k arreglos en Ak, donde el arreglo en la posición i-ésima Ak[i] es el arreglo de entrada A con todos sus elementos multiplicados por i. A partir de este punto estamos en las mismas condiciones del ejercicio 4, tenemos un conjunto de arreglos ordenados, todos de tamaño n, y queremos unirlos de forma ordenada. Así que aprovechamos esa solución y simplemente usamos la función UnirOrdenados.

15. Ejercicio 15: tiene agujero

TieneAgujero(**in** A: arreglo(nat)) → **out** res: bool

```
1: n ← tam(A)                                                          ▷ O(1)
2: min ← BuscarMin(A)                                                  ▷ O(n)
3: max ← BuscarMax(A)                                                  ▷ O(n)
4: if max - min + 1 > n then
5:     res ← true
6: else
7:     C: arreglo(bool) ← CrearArreglo(n)          ▷ O(n), elementos inicializados en false
8:     r ← 0
9:     for i ← 1 to n do                                               ▷ O(n)
10:        j ← A[i] - min + 1
11:        if C[j] = true then
12:            r ← r + 1
13:        else
14:            C[j] ← true
15:        end if
16:    end for
17:    res ← max - min + 1 + r ≠ n
18: end if
```

Complejidad: $O(n)$

Que no haya agujeros en esencia es que el arreglo tenga todos números consecutivos.

Sabiendo el mínimo y el máximo del arreglo podemos calcular la cantidad de números consecutivos sin repetidos: $max - min + 1$. Sea r la cantidad de elementos repetidos (sin contar su aparición inicial), sabemos que hay agujeros si se cumple $max - min + 1 + r \neq n$.

Para contar los repetidos hacemos algo parecido a la parte de counting de un CountingSort, en donde recorremos una vez el arreglo de entrada y vamos contando cada vez que encontramos un número repetido.

16. Ejercicio 16: raroSort

RaroSort(**in/out** A: arreglo(nat))

```
1: m ← BuscarMax(A)                                                    ▷ O(n)
2: d ← log(m)                                          ▷ Cantidad máxima de dígitos en base 2
3: for i ← 0 to d do                                                   ▷ O(d) = O(log(m))
4:     CountingSort(A, m, i)          ▷ O(n + m), ordenamos por el i-ésimo dígito en base 2
5: end for
```

Complejidad: $O((n + m)log(m))$
Para revisar porque el enunciado pide $O(nlog(m))$. No se si vale decir que $O(n + m) = O(n)$ en este contexto.

17. Ejercicio 17: i elementos mas chicos

La clave es entender bien la precondición del arreglo que da el enunciado. Lo podemos escribir de esta forma:

Pre $\equiv$ noHayRepetidos(A) $\wedge$ $(\forall i : nat)(1 \leq i \leq n \Rightarrow_{\text{L}} \text{contarMenores}(A, A[i]) \leq i)$

Supongamos que tenemos un arreglo A ordenado de 5 elementos distintos. Sean $a_1, \ldots, a_5$ los elementos de A que están en las posiciones $1, \ldots, 5$. Es decir, $A \equiv a_1, a_2, a_3, a_4, a_5$.

Si quisiéramos desordenar el arreglo para que cumpla con la pre, veamos cuántas posiciones son válidas para cada elemento. Es decir, en cuántos lugares podemos meter cada a_i de forma tal que valga la pre. Notemos que como A está ordenada, se cumple que $a_1 < \ldots < a_5$, y por lo tanto sabemos qué devuelve contarMenores para cada elemento (no importa qué elementos son exactamente). A partir de esta información podemos determinar el rango de posiciones que serían válidas para cada a_i de forma tal que se cumpla la pre.

- contarMenores$(A, a_1) = 0 \leq [1, 2, 3, 4, 5]$
- contarMenores$(A, a_2) = 1 \leq [1, 2, 3, 4, 5]$
- contarMenores$(A, a_3) = 2 \leq [2, 3, 4, 5]$
- contarMenores$(A, a_4) = 3 \leq [3, 4, 5]$
- contarMenores$(A, a_5) = 4 \leq [4, 5]$

Comenzamos a desordenar los elementos de atrás hacia adelante. Observemos que a_5 tiene dos posiciones válidas. Fijada la posición de a_5, ahora a_4 solo tiene dos posiciones válidas, pues una la ocupamos con a_5. Por la misma razón, a_3 y a_2 también tienen solo dos posiciones válidas, y finalmente a_1 le queda una única posición libre para ubicarse y que se mantenga la pre.

Para ordenar, basta recorrer el arreglo de atrás hacia adelante e ir comparando cada elemento adyacente, y si están desordenados, ordenarlos con un swap. De esta forma vamos dejando los elementos más grandes ya ordenados y así reduciendo las posibles posiciones para los elementos más chicos.

Ejemplos de entradas válidas

```
A = [1, 2, 3, 4, 5]
A = [2, 1, 4, 3, 5]
A = [1, 5, 4, 7, 6]
A = [1, 3, 4, 2, 7]
A = [1, 2, 3, 5, 4]
```

OrdenarConSwaps(**in/out** A: arreglo(nat))

```
1: n ← tam(A)
2: for i ← n downto 2 do                                   ▷ O(n)
3:     if A[i] < A[i - 1] then
4:         Swap(A[i], A[i - 1])
5:     end if
6: end for
```

Complejidad: $O(n)$

18. Ejercicio 18: radix encubierto

18.a.

OrdenarHastaN(**in/out** A: arreglo(nat))

```
1: CountingSort(A)                                         ▷ O(n)
```

Complejidad: $O(n)$

18.b.

OrdenarHastaN^2(**in/out** A: arreglo(nat))

```
1: n ← tam(A)
2: B: arreglo(⟨ nat, nat ⟩) ← CrearArreglo(n)              ▷ O(n)
3: for i ← 1 to n do                                       ▷ O(n)
4:     B[i] ← ⟨ A[i] mod n, A[i] div n ⟩
5: end for
6: OrdenarHastaN(B)          ▷ O(n), clave de ordenamiento: primera componente de la tupla
7: OrdenarHastaN(B)          ▷ O(n), clave de ordenamiento: segunda componente de la tupla
8: for i ← 1 to n do                                       ▷ O(n)
9:     A[i] ← B[i][1] * n + B[i][0]
10: end for
```

Complejidad: $O(n)$

Ejemplo de ejecución

```
A = [40, 25, 29, 83, 12, 50, 67, 13, 99, 76]
B = [(0, 4), (5, 2), (9, 2), (3, 8), (2, 1), (0, 5), (7, 6), (3, 1), (9, 9), (6, 7)]
B = [(0, 4), (0, 5), (2, 1), (3, 8), (3, 1), (5, 2), (6, 7), (7, 6), (9, 2), (9, 9)]
B = [(2, 1), (3, 1), (5, 2), (9, 2), (0, 4), (0, 5), (7, 6), (6, 7), (3, 8), (9, 9)]
A = [12, 13, 25, 29, 40, 50, 67, 76, 83, 99]
```

18.c.

OrdenarHastaN^k(in/out A: arreglo(nat), **in** k: nat)

```
1: n ← tam(A)
2: B: arreglo(⟨ nat, ..., nat ⟩_k) ← CrearArreglo(n)        ▷ O(n), arreglo con tuplas de k elementos
3: for i ← 1 to n do                                          ▷ O(n)
4:     for j ← 0 to k-1 do                                    ▷ O(k)
5:         B[i][j] ← (A[i] div n^j) mod n
6:     end for
7: end for
8: for i ← 0 to k do                                          ▷ O(k)
9:     OrdenarHastaN(B)          ▷ O(n), clave de ordenamiento: i-ésima componente de la tupla
10: end for
11: for i ← 1 to n do                                         ▷ O(n)
12:     A[i] ← 0
13:     for j ← 0 to k-1 do                                   ▷ O(k)
14:         A[i] ← A[i] + B[i][j] * n^j
15:     end for
16: end for
```

Complejidad: $O(nk)$

Lo que estamos haciendo para ordenar es básicamente un RadixSort si interpretamos los números del arreglo de entrada en base n. En el arreglo B descomponemos el número original para obtener sus dígitos en base n. Como nos dicen que todos los números son positivos menores que n^k, la cantidad máxima de dígitos será $log_n(n^k) = k$. Luego ordenamos por el dígito menos significativo con un algoritmo estable como el CountingSort, y luego por el siguiente dígito, y así hasta el dígito k-ésimo, el más significativo.

18.d.

Si se generaliza la idea para arreglos no acotados de tamaño n, se podría buscar m el máximo del arreglo, luego buscar el mínimo k tal que $m \leq n^k$, y realizar el mismo algoritmo. Esto sólo conviene mientras $k \leq log(n)$, pues si $k > log(n)$ será más eficiente hacer un MergeSort sobre el arreglo original de entrada ya que tiene complejidad $O(nlog(n))$ sin importar cuál es el máximo del arreglo.