

Algoritmos y Estructuras de Datos 2

Práctica 5: Ordenamiento

1er cuatrimestre 2022

Índice

1. Ejercicio 1: Comparacion de algoritmos de ordenamiento	2
2. Ejercicio 2: Ordenar concatenacion	2
3. Ejercicio 3: K elementos mas chicos	2
4. Ejercicio 4: Conjunto de secuencias	2
5. Ejercicio 5: frecuencia	3
6. Ejercicio 6: Escalera	4
7. Ejercicio 7: AVL Sort	4
8. Ejercicio 8: Dos arreglos	5
9. Ejercicio 9: Planilla	6
10.Ejercicio 10: casiSort	7
11.Ejercicio 11: Counting Sort	7
12.Ejercicio 12: Mediciones	7
13.Ejercicio 13: Tuplas	8
14.Ejercicio 14: Multiplos	8
15.Ejercicio 15: Agujero en conjunto	10
16.Ejercicio 16	10
17.Ejercicio 17: Arreglo de enteros no repetidos	10
18.Ejercicio 18	11

1. Ejercicio 1: Comparacion de algoritmos de ordenamiento

nombreFuncion(in r: recibe) \rightarrow out d: devuelve

1: codigo

Complejidad: $O(n^2)$

2. Ejercicio 2: Ordenar concatenacion

Utilizaria el metodo InsertionSort ya que en su invariante requiere que el arreglo este ordenado desde 0 hasta i, siendo $i = \text{Longitud}(s')$.

3. Ejercicio 3: K elementos mas chicos

kMasChicos(in A: arreglo(nat), in k: nat) \rightarrow out res: arreglo(nat)

Pre $\equiv \{k \leq \text{Longitud}(A)\}$

```
1: res  $\leftarrow$  crearArreglo(k)
2: max  $\leftarrow$  maximo(A)
3: n  $\leftarrow$  Longitud(A)
4: for i  $\leftarrow$  1 to k do
5:   res[i]  $\leftarrow$  max
6: end for
7: for i  $\leftarrow$  1 to k do
8:   for j  $\leftarrow$  1 to n do
9:     if res[i] > A[j]  $\wedge_L$  (i = 0  $\vee_L$  A[j] > res[i-1]) then
10:      res[i]  $\leftarrow$  A[j]
11:     end if
12:   end for
13: end for
```

Complejidad: $O(kn)$

Si $k > \log(n)$, conviene ordenarlo primero usando mergeSort el cual tiene complejidad $O(n \log(n))$.

4. Ejercicio 4: Conjunto de secuencias

unirOrdenados(in A: arreglo(arreglo(nat))) \rightarrow out res: arreglo(nat)

Pre $\equiv \{(\forall i : nat)(1 \leq i \leq \text{Longitud}(A) \Rightarrow_L \text{estaOrdenado}(A[i]))\}$

```
1: if Longitud(A) = 1 then
2:   res  $\leftarrow$  A[1]
3: else
4:   izq  $\leftarrow$  Vector::Vacia();
5:   der  $\leftarrow$  Vector::Vacia();
6:   for i  $\leftarrow$  1 to Longitud(A) do
7:     if i  $\leq$  Longitud(A)/2 then
8:       AgregarAtras(izq, A[i])
9:     else
10:      AgregarAtras(der, A[i])
11:    end if
12:   end for
13:   res  $\leftarrow$  merge(unirOrdenados(izq), unirOrdenados(der))
14: end if
```

Complejidad: $O(m * \text{long}(n))$

Sea n la cantidad de secuencias en A, en cada llamada recursiva divido al arreglo en dos, hasta llegar a $n = 1$, entonces $n/2^i = 1$ lo que es igual a $i = \log(n)$, es decir el costo de cada llamada recursiva va a ser $O(\log(n))$.

Luego el peor caso del merge es de $O(m)$, siendo m la cantidad total de elementos en el arreglo. Luego como hacemos el merge de m elementos recursivamente el costo total es $O(m \log(n))$

merge(in izq: vector(nat), in der: vector(nat)) \rightarrow **out** res: vector(nat)

```

1: res  $\leftarrow$  Vector::Vacia()
2: while !EsVacio?(izq)  $\vee_L$  !EsVacio?(der) do
3:   if EsVacio?(izq) then
4:     while !EsVacio?(der) do
5:       AgregarAtras(res, der[0])
6:       Eliminar(der, 0)
7:     end while
8:   else
9:     if EsVacio?(der) then
10:      while !EsVacio?(izq) do
11:        AgregarAtras(res, izq[0])
12:        Eliminar(izq, 0)
13:      end while
14:    else
15:      if izq[0]  $\leq$  der[0] then
16:        AgregarAtras(res, izq[0])
17:        Eliminar(izq, 0)
18:      else
19:        AgregarAtras(res, der[0])
20:        Eliminar(der, 0)
21:      end if
22:    end if
23:  end if
24: end while

```

Complejidad: $O(n)$, siendo $n = \text{Longitud}(\text{izq}) + \text{Longitud}(\text{der})$

5. Ejercicio 5: frecuencia

ordenarPorFrecuencia(in/out A: arreglo(nat))

```

1: frecuencia  $\leftarrow$  DiccLineal::Vacio()  $\triangleright O(1)$ 
2: mergeSort(A)  $\triangleright O(n \log(n))$ 
3: n  $\leftarrow$  Longitud(A)  $\triangleright O(1)$ 
4: for i  $\leftarrow$  1 to n do  $\triangleright O(n^2)$ 
5:   if Definido?(frecuencia, A[i]) then  $\triangleright O(n)$ 
6:     aux  $\leftarrow$  Significado(frecuencia, A[i])  $\triangleright O(n)$ 
7:     Definir(frecuencia, A[i], aux + 1)  $\triangleright O(n)$ 
8:   else
9:     Definir(frecuencia, A[i], 1)  $\triangleright O(n)$ 
10:  end if
11: end for
12: res  $\leftarrow$  Arreglo::Vacia()  $\triangleright O(1)$ 
13: while #Claves(frecuencia) > 0 do  $\triangleright O(n^2)$ 
14:   c  $\leftarrow$  obtenerMax(frecuencia)
15:   for i  $\leftarrow$  1 to Significado(frecuencia, b) do
16:     AgregarAtras(res, c)
17:   end for
18:   Borrar(frecuencia, c)
19: end while
20: A  $\leftarrow$  res

```

Complejidad: $O(n^2)$

obtenerMax(in D: diccLineal(nat, nat)) → **out** res: nat

```
1: max ← 0
2: res ← 0
3: it ← CrearIt(D)
4: while HaySiguiente(it) do
5:   if SiguienteSignificado(it) > max then
6:     max ← SiguienteSignificado(it)
7:     res ← SiguienteClave(it)
8:   end if
9:   Avanzar(it)
10: end while
```

▷ $O(1)$
▷ $O(1)$
▷ $O(1)$
▷ $O(n)$

Complejidad: $O(n)$

6. Ejercicio 6: Escalera

ordenarEscalera(in/out r: recibe)

```
1: B ← obtenerEscaleras(A)
2: mergeSort(B, A)
3: mergeSortPorLong(B)
4: C ← Copiar(A)
5: k ← 1
6: for i ← 1 to Longitud(B) do
7:   for j ← B[i][0] to B[i][1] do
8:     A[k] ← C[j]
9:     k ← k + 1
10:  end for
11: end for
```

▷ $O(n \log(n))$, clave de ordenamiento: $B[i][0]$ en el arreglo A.

▷ $O(n \log(n))$, clave de ordenamiento: $B[i][1] - B[i][0] + 1$, es decir la longitud.

Complejidad: $O(n \log(n))$

obtenerEscaleras(in A: arreglo(nat)) → **out** res: arreglo(\langle nat, nat \rangle)

```
1: res ← Vector::Vacía()
2: i ← 1
3: while i ≤ n do
4:   j ← i
5:   while j < n ∧  $A[j] + 1 = A[j+1]$  do
6:     j ← j + 1
7:   end while
8:   AgregarAtras(res,  $\langle i, j \rangle$ )
9:   i ← j + 1
10: end while
```

Complejidad: $O(n^2)$

7. Ejercicio 7: AVL Sort

Como aconseja el ejercicio uso arboles AVL, ya que todas las operaciones sobre este serán $O(\log d)$, para implementarlo lo pense como un arbol de tuplas, siendo el 1er elemento de la tupla el valor que agregamos y el 2do la cantidad de veces que lo agregamos. Si se agrega por primera vez se le pone un 1 en el 2do elemento, luego si lo agregas de nuevo se le suma 1 y así... Primero recorro el arreglo y agrego cada elemento al arbol, luego de haber recorrido todo el arreglo saco el minimo valor y la cantidad de veces que aparece del arbol y lo inserto en el arreglo, una vez termine con ese minimo lo borro y busco el siguiente minimo, y así hasta que el arbol me quede vacio.

AVLSort(in/out A: arreglo)

```
1: arbol ← Vacío::AVL( $\langle$  nat, nat  $\rangle$ )
2: for i ← 1 to tam(A) do
3:   Insertar(arbol, A[i])
4: end for
5: A ← AVLAArreglo(arbol, A)
```

Complejidad: $O(n \log(d))$

AVLAArreglo(in AB: abAVL($\langle\text{nat}, \text{nat}\rangle$)) \rightarrow **out** res: arreglo

```
1: res  $\leftarrow$  Vacia::Arreglo
2: while !Vacía?(arbol) do  $\triangleright O(n)$ 
3:   min  $\leftarrow$  minimo(arbol)  $\triangleright O(\log d)$ 
4:   cant  $\leftarrow \pi_2(min)$   $\triangleright O(1)$ 
5:   val  $\leftarrow \pi_1(min)$   $\triangleright O(1)$ 
6:   for j  $\leftarrow$  1 to cant do  $\triangleright O(m)$ , siendo m la cantidad de veces que aparece el elemento, si sumo todos los m me da n
7:     AgregarAtras(res, val)  $\triangleright O(1)$ 
8:   end for
9:   borrarMinimo(arbol)  $\triangleright O(\log d)$ 
10: end while
```

Complejidad: $O(n^2)$

8. Ejercicio 8: Dos arreglos

DosArreglos(in A: arreglo, in B: arreglo) \rightarrow **out** res: arreglo

```
1: res  $\leftarrow$  Arreglo::CrearArreglo(tam(A)+tam(B))
2: C  $\leftarrow$  arregloTuplas(A)  $\triangleright O(n)$ 
3: D  $\leftarrow$  arregloTuplas(B)  $\triangleright O(m)$ 
4: E  $\leftarrow$  mergeSort(C, B)  $\triangleright O((n' + m)\log(n' + m))$ , Clave de ordenamiento de ambas: 1er elemento de la tupla.
5: k  $\leftarrow$  1
6: for i  $\leftarrow$  1 to tam(E) do  $\triangleright O(n + m)$ 
7:   for j  $\leftarrow$  1 to  $\pi_2(E[i])$  do
8:     res[k]  $\leftarrow \pi_1(E[i])$ 
9:     k  $\leftarrow$  k + 1
10:   end for
11: end for
```

Complejidad: $O(n + (n' + m)\log(n' + m))$

arregloTuplas(in A: arreglo(nat)) \rightarrow **out** res: arreglo($\langle\text{nat}, \text{nat}\rangle$)

```
1: C  $\leftarrow$  Vector::Vacía()
2: val  $\leftarrow$  A[1]
3: cant  $\leftarrow$  1
4: for i  $\leftarrow$  2 to tam(A) do  $\triangleright O(n)$ 
5:   if A[i] == val then  $\triangleright O(1)$ 
6:     cant  $\leftarrow$  cant + 1  $\triangleright O(1)$ 
7:   else
8:     AgregarAtras(C,  $\langle\text{val}, \text{cant}\rangle$ )  $\triangleright O(1)$ 
9:     val  $\leftarrow$  A[i]  $\triangleright O(1)$ 
10:    cant  $\leftarrow$  1  $\triangleright O(1)$ 
11:   end if
12: end for
13: res  $\leftarrow$  C
```

Complejidad: $O(n)$

9. Ejercicio 9: Planilla

9.a.

ordenaPlanilla(in/out p: planilla)

```
1: alumnos ← Vector::Vacía()                                ▷  $O(1)$ 
2: n ← Longitud(p)                                           ▷  $O(1)$ 
3: for i ← 1 to 10 do
4:   for j ← 1 to n do                                       ▷  $O(n)$ 
5:     if p[j].puntaje = i then
6:       AgregarAtras(alumnos, p[j])
7:     end if
8:   end for
9: end for
10: t ← 0
11: for alum in alumnos do                                    ▷  $O(n)$ 
12:   if alum.genero = FEM then
13:     p[t] = alum
14:     t ← t + 1
15:   end if
16: end for
17: for alum in alumnos do                                    ▷  $O(n)$ 
18:   if alum.genero = MASC then
19:     p[t] = alum
20:     t ← t + 1
21:   end if
22: end for
```

Complejidad: $O(n)$

9.b.

ordenaPlanilla(in/out p: planilla)

```
1: alumnos ← Vector::Vacía()                                ▷  $O(1)$ 
2: n ← Longitud(p)                                           ▷  $O(1)$ 
3: for i ← 1 to 10 do
4:   for j ← 1 to n do                                       ▷  $O(n)$ 
5:     if p[j].puntaje = i then
6:       AgregarAtras(alumnos, p[j])
7:     end if
8:   end for
9: end for
10: t ← 0
11: for g in GEN do
12:   for alum in alumnos do                                    ▷  $O(n)$ 
13:     if alum.genero = g then
14:       p[t] = alum
15:       t ← t + 1
16:     end if
17:   end for
18: end for
```

Complejidad: $O(n)$

9.c.

El lower bound solo se aplica a algoritmos de ordenamiento basados en comparaciones. Y en nuestro algoritmo nunca comparamos 1 a 1 los elementos de nuestro arreglo.

10. Ejercicio 10: casiSort

10.a.

ordenarConCasiSort(in/out A: arreglo(nat))

```
1: if tam(A) == 2 then
2:   if A[1] < A[2] then
3:     Swap(A[1], A[2])
4:   end if
5: else
6:   casiSort(A)
7:   mitadIzq ← A[1...tam(A)/2]
8:   mitadDer ← ordenarConCasiSort([tam(A)/2+1...tam(A)])
9:   A ← merge(mitadIzq, mitadDer)
10: end if
```

▷ $O(n)$

▷ $O(n)$

▷ $O(n)$

▷ $O(n)$

Complejidad: $O(n)$

10.b.

La complejidad es $O(n)$

10.c.

No, no es posible sin tener mas informacion, si tuviera una cota podria usar BucketSort, CountingSort o RadixSort. Pero al no ser asi el caso la minima complejidad que puedo tener es $O(n\log(n))$, usando MergeSort, HeapSort, entre otros.

11. Ejercicio 11: Counting Sort

CountingSort(in/out A: arreglo)

```
1: m ← 0
2: for i ← 1 to tam(A) do
3:   m ← max(m, A[i])
4: end for
5: B ← Arreglo::CrearArreglo(m)
6: for i ← 1 to tam(A) do
7:   B[A[i]] ← B[A[i]] + 1
8: end for
9: i ← 0
10: for j ← 1 to m do
11:   for t ← 1 to B[j] do
12:     A[i] ← j
13:     i ← i + 1
14:   end for
15: end for
```

Complejidad: $O(n)$

12. Ejercicio 12: Mediciones

Se que la mayoria de los valores estan entre 20 y 40, de ahi tengo una cota, luego \sqrt{n} elementos estan fuera del rango, pero tambien tengo otra cota, siendo 19 el maximo, luego tengo los elementos mayores a 40, eso lo puedo ordenar con un mergeSort

Mediciones(in/out A: arreglo(nat))

```
1: IZQ ← Vector::Vacía()                                ▷ O(1)
2: MED ← Vector::Vacía()                                ▷ O(1)
3: DER ← Vector::Vacía()                                ▷ O(1)
4: for i ← 1 to tam(A) do                                ▷ O(n)
5:   if A[i] < 20 then
6:     AgregarAtras(IZQ, A[i])
7:   else if A[i] > 40 then
8:     AgregarAtras(DER, A[i])
9:   else
10:    AgregarAtras(MED, A[i])
11:  end if
12: end for
13: CountingSort(IZQ)                                    ▷ O(n), con cota 19.
14: CountingSort(MED)                                    ▷ O(n), con cota minima 20 y maxima 40
15: MergeSort(DER)                                       ▷ O( $\sqrt{n} * \log(\sqrt{n}) < O((\sqrt{n})^2)$ ) = O(n)
16: Merge(IZQ, MED, DER)                                ▷ O(n)
```

Complejidad: $O(n)$

13. Ejercicio 13: Tuplas

13.a.

tuplas(in/out A: arreglo($\langle c1: nat, c2: string[l] \rangle$))

```
1: RadixSort(A)                                          ▷ O(nl), Clave de ordenamiento: la longitud del string
2: MergeSort(A)                                         ▷ O(nlog(n)), Clave de ordenamiento: 1er elemento de cada tupla
```

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

El RadixSort tiene complejidad $O(l(n + k))$, ya que, en mi caso, usa BucketSort, el cual tiene una Complejidad $O(n + k)$, siendo n el largo del arreglo y k la cantidad maxima posible de caracteres siendo esta = 257, por lo que esta acotada. Luego Repito recursivamente el BucketSort l veces, siendo asi la complejidad $O(nl)$
Luego para cumplir con la complejidad uso MergeSort, la cual tiene complejidad $O(nlog(n))$
Sumando todo te queda, $O(nl + nlog(n))$

13.b.

tuplas(in/out A: arreglo($\langle c1: nat, c2: string[l] \rangle$))

```
1: RadixSort(A)                                          ▷ O(nl), Clave de ordenamiento: la longitud del string
2: CountingSort(A)                                       ▷ O(n), Por el 1er componente ya que esta acotado
```

Complejidad: $O(nl)$

Para ordenar por el 2do componente uso de nuevo el RadixSort ya que no me cambio nada respecto a eso, luego para el 1er componente al estar acotado puedo usar CountingSort, lo cual me da $O(n)$

14. Ejercicio 14: Multiplos

Sea $k = 3$:

```
A = [2, 5, 1]
A_{1} = [2, 4, 6]
A_{2} = [5, 10, 15]
A_{3} = [1, 2, 3]
```

Usando lo solucion de honi:

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

```
1: mat ← Arreglo(Arreglo(nat))::CrearArreglo(tam(A))
2: for i ← 1 to tam(A) do
3:   mat[i] ← ArregloDeMultiplo(A[i], k)
4: end for
5: UnirOrdenados(mat)
```

▷ O(n*k)
▷ O(k)
▷ O(nklog(n))

Complejidad: $O(nk\log(n))$

ArregloDeMultiplo(in e: nat, in k: nat) → out res: arreglo(nat)

```
1: res ← Arreglo::CrearArreglo(k)
2: for i ← 1 to k do
3:   res[i] ← e * i
4: end for
```

Complejidad: $O(k)$

Otra opcion usando heap:

La idea seria tener un puntero por fila, asi cada puntero apunta al 1er elemento de cada fila de la matriz, y los elementos a los cuales estos punteros apuntan los voy insertando en un heap, luego al sacar el minimo e insertarlo en un arreglo despues avanzas la posicion de la fila del cual pertenece y asi hasta no poder seguir mas con los punteros, no se me ocurrio como hacerlo asi que lo hice a lo bruto.

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

```
1: mat ← Arreglo(Arreglo(nat))::CrearArreglo(tam(A))
2: for i ← 1 to tam(A) do
3:   mat[i] ← ArregloDeMultiplo(A[i], k)
4: end for
5: res ← MatrizConHeap(mat)
```

▷ O(n*k)
▷ O(k)
▷ O(nklog(n))

Complejidad: $O(nk\log(n))$

1era Opcion:

MatrizConHeap(in M: arreglo(arreglo(nat))) → out res: arreglo(nat)

```
1: res ← CrearArreglo(tam(M)*tam(M[1]))
2: C ← CrearArreglo(tam(M))
3: H ← Heap::Vacio()
4: while tam(M) > 0 do
5:   for i ← 1 to tam(M) do
6:     Insertar(H, ⟨M[i][1], i⟩)
7:   end for
8:   min ← Minimo(H)
9:   BorrarMinimo(H)
10:  AgregarAtras(C, min)
11:  Eliminar(M[π2(min)], 1)
12:  if tam(M[π2(min)]) == 0 then
13:    Eliminar(M, π2(min))
14:  end if
15:  H ← Heap::Vacio()
16: end while
17: res ← C
```

▷ O(n*k)
▷ O(n)
▷ O(1)
▷ O(nk*log(n))
▷ O(n*log(n))
▷ O(1)
▷ O(log(n))
▷ O(1)
▷ O(k)
▷ O(n)
▷ O(1)

Complejidad: $O(n^2)$

2da Opcion:

MatrizConHeap(in M: arreglo(arreglo(nat))) → **out** res: arreglo(nat)

```

1: res ← CrearArreglo(tam(M)*tam(M[1]))                                ▷ O(n*k)
2: C ← Vector::Vacio()                                                  ▷ O(n)
3: H ← HeapMin::Vacio()                                                  ▷ O(1)
4: k ← tam(M[1])                                                         ▷ O(1)
5: B ← Arreglo::CrearArreglo(tam(M))                                    ▷ O(n)
6: for j ← 1 to tam(B) do                                                ▷ O(n)
7:   B[j] ← 1
8: end for
9: for i ← 1 to tam(M) do                                                ▷ O(n)
10:   Insertar(H, ⟨M[i][1], i⟩)
11: end for
12: while tam(M)*k > Longitud(C) do                                       ▷ O(nk*log(n))
13:   min ← Minimo(H)                                                    ▷ O(1)
14:   BorrarMinimo(H)                                                    ▷ O(log(n))
15:   if B[π2(min)] != tam(M[1]) then
16:     Insertar(H, M[π2(min)][B[π2(min)]+1])                            ▷ O(log(n))
17:     B[π2(min)] ← B[π2(min)] + 1
18:   end if
19:   AgregarAtras(C, [π1(min)])                                         ▷ O(1)
20: end while
21: res ← C

```

Complejidad: $O(n^2)$

15. Ejercicio 15: Agujero en conjunto

Pendiente

16. Ejercicio 16

Pendiente

17. Ejercicio 17: Arreglo de enteros no repetidos

Cumplen:

```

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

```

elementosMasChicosQuePos(in r: recibe) → **out** d: devuelve

```

1: i ← tam(A)
2: while i > 1 do
3:   if A[i] > A[i-1] then
4:     Swap(A[i], A[i-1])
5:   end if
6:   i ← i - 1
7: end while

```

Complejidad: $O(n^2)$

18. Ejercicio 18

18.a.

ordenarEnN(in A: arreglo(nat))

1: countingSort(A)

Complejidad: $O(n)$

18.b.

ordenarEnN²(in A: arreglo(nat))

1: $n \leftarrow \text{Longitud}(A)$

2: $\text{baseN} \leftarrow \text{CrearArreglo}(n)$

3: **for** $i \leftarrow 1$ **to** n **do**

4: $\text{baseN}[i] \leftarrow \langle A[i] \text{ div } n, A[i] \text{ mod } n \rangle$

5: **end for**

6: $\text{ordenarEnN}(\text{baseN})$

▷ $O(n)$, ordenar por la 2da tupla

7: $\text{ordenarEnN}(\text{baseN})$

▷ $O(n)$, ordenar por la 1era tupla

8: **for** $i \leftarrow 1$ **to** n **do**

9: $A[i] \leftarrow \text{BaseN}[i][0] * n + \text{BaseN}[i][1]$

10: **end for**

Complejidad: $O(n)$
