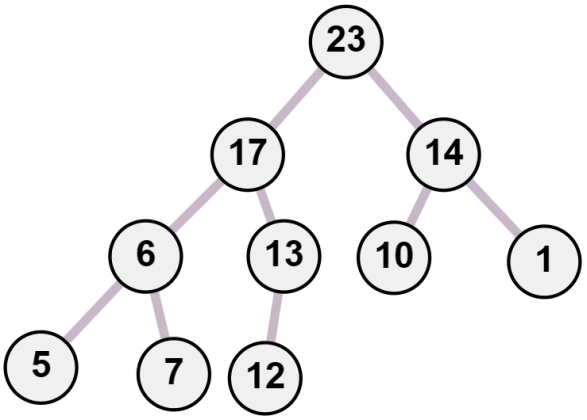
Solución Taller 2: Algoritmos de Ordenamiento y Estructuras de Datos

Alejandro Duarte Rojas
Cristhian Yamith Cely Oliveros
Mateo Gutierrez Melo
David Steven Rodriguez Guzman
Sebastián Escandón Flórez

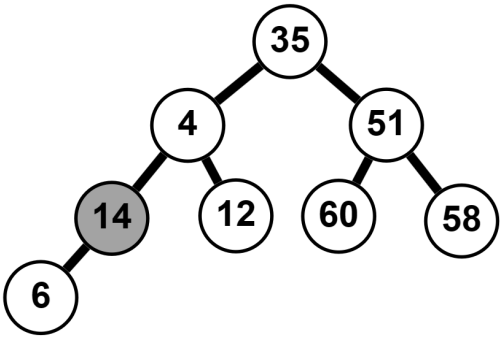
1) En la última posición de un max-heap, sí se representa como árbol binario estaría en el la última hoja de izquierda a derecha.

2) A = (23, 14, 6 13, 10, 1, 5, 7, 12)

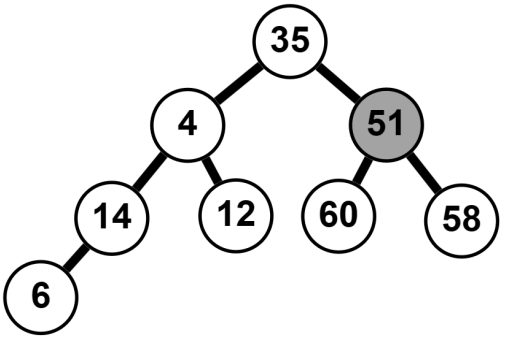


A **no** es un max-heap

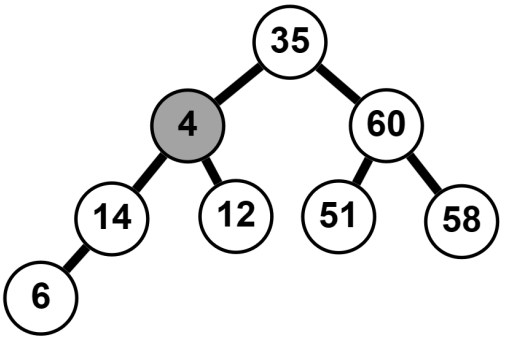
3) A = (35,4,51,14,12,60,58,6)
BUILD-MAX-HEAP(A)



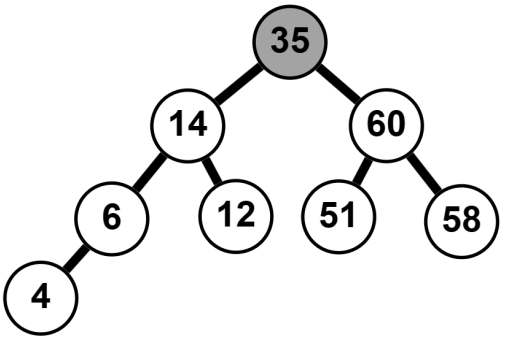
a)



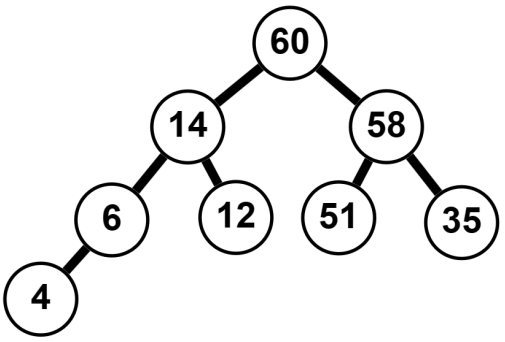
b)



c)



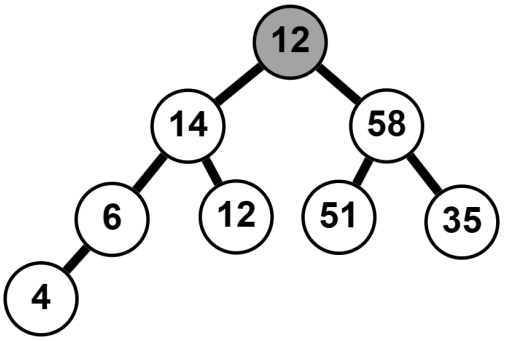
d)



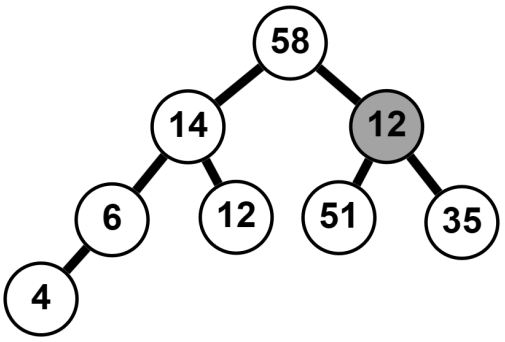
e)

A = (60,14,58,6,12,51,35,4)

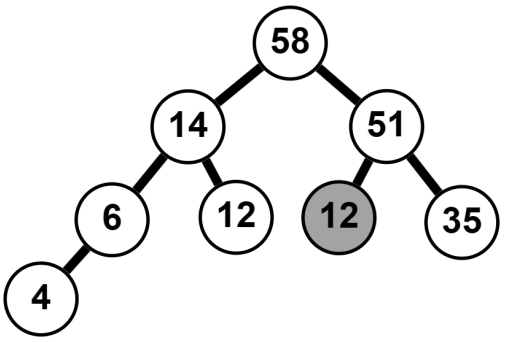
4) A = (12,14,58,6,12,51,35,4)
MAX-HEAPIFY(A,1)



a)



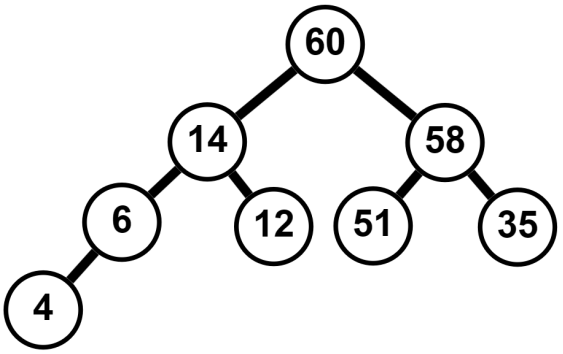
b)



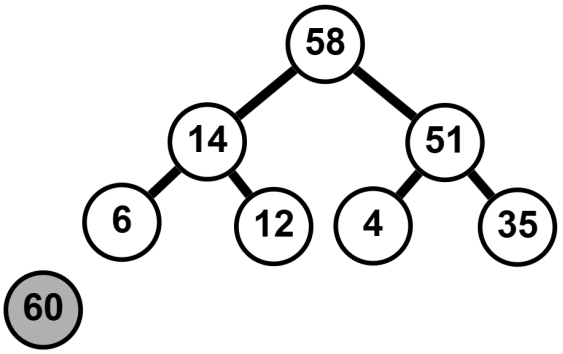
c)

A = (58,14,51,6,12,12,35,4)

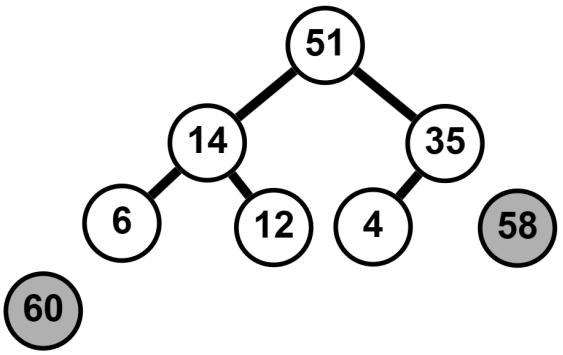
5) A = (35,4,51,14,12,60,58,6)
HEAPSORT(A)
BUILD-MAX-HEAP(A) = (60,14,58,6,12,51,35,4)



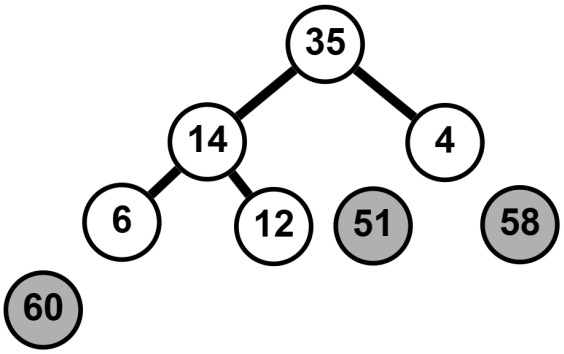
a)



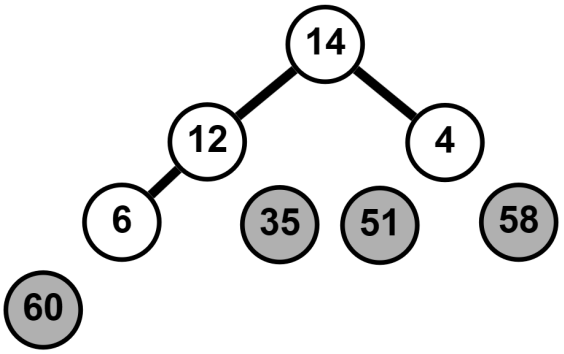
b)


51
14
35
6
12
4
58
60

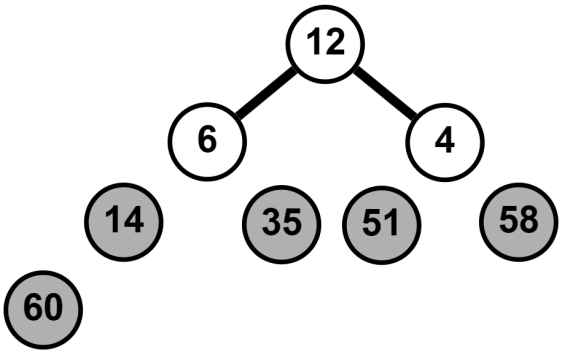
c)


35
14
4
6
12
51
58
60

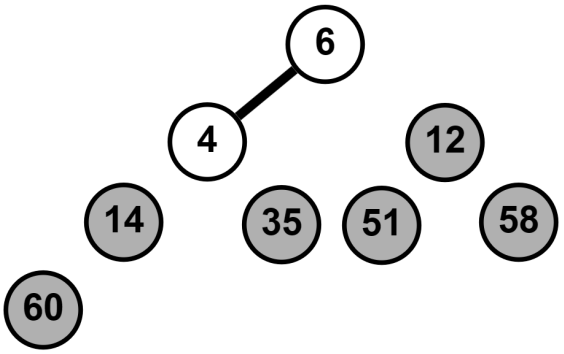
d)


14
12
4
6
35
51
58
60

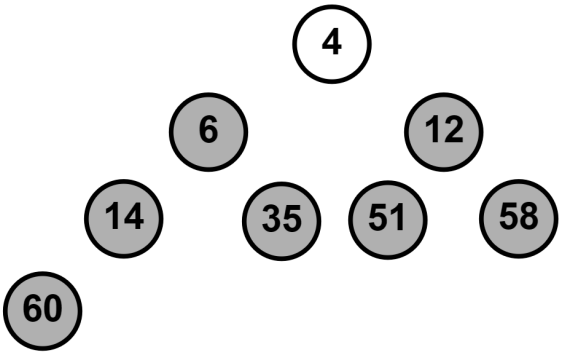
e)


12
6
4
14
35
51
58
60

f)



g)

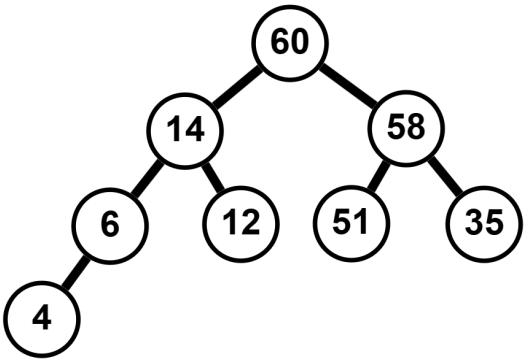


h)

A = (4,6,12,14,35,51,58,60)

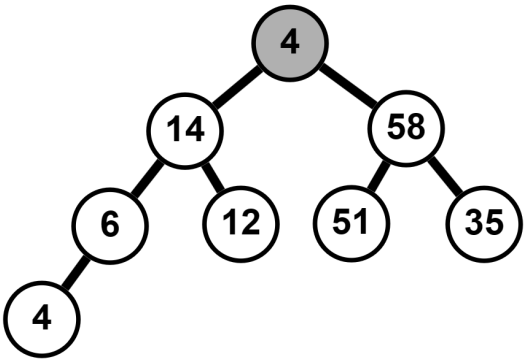
6) El tiempo de ejecución del HEAPSORT en un que ya está ordenado incrementalmente es de O(n lg n) ya que primero se le hace un BUIL-MAX-HEAP y luego hace todas las operaciones igualmente. Lo mismo pasa si está ordenado decrecientemente.

7) A = (60,14,58,6,12,51,35,4)
HEAP-EXTRACT-MAX(A)

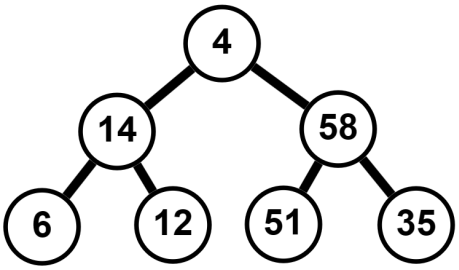


max = 60

a)

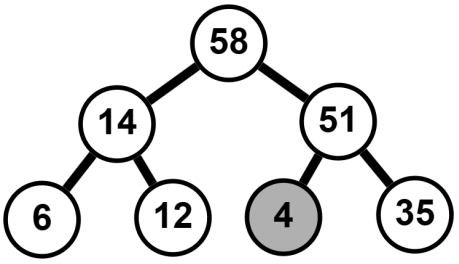


b)



c)

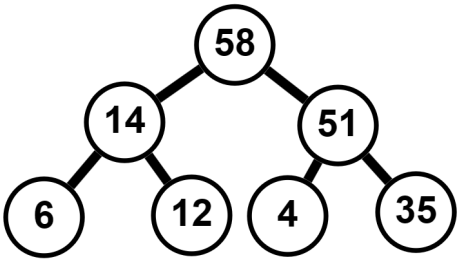
MAX-HEAPIFY(A,1)



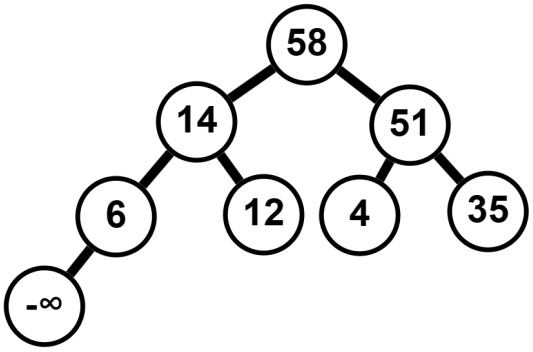
d)

max = 60
A = (58,14,51,6,12,4,35)

8) A = (58,14,51,6,12,4,35)
MAX-HEAP-INSERT(A,10)

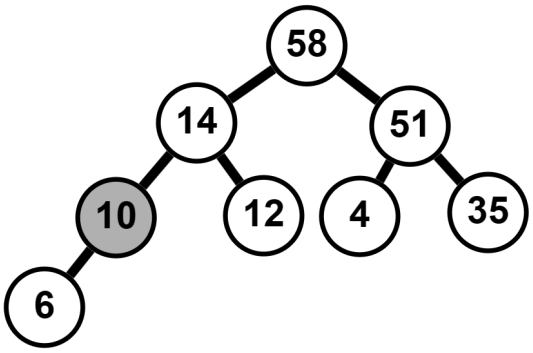


a)



b)

HEAP-INCREASE-KEY(A,8,10)



c)

A = (58,14,51,10,12,4,35,6)

9) HEAP-MINIMUN(A)

```
1       return A[1]
```

HEAP-EXTRACT-MIN(A)

```
1       if A.heap-size < 1
2               error “heap underflow”
3       min = A[1]
4       A[1] = A[A.heap-size]
```

```
5       A.heap-size = A.heapsize-1
6       MIN-HEAPIFY(A,1)
7       return min

HEAP-DECREASE-KEY(A, i, key)
1       if key > A[i]
2               error "new key is longer than current key"
3       A[i] =  key
4       while i > 1 and A[Parent(i)] > A[i]
5               exchange A[i] with A[Parent(i)]
6               i = Parent(i)

MIN-HEAP-INSERT
1       A.heap-size = A.heap-size + 1
2       A[A.heap-size] = +∞
3       HEAP-DECREASE-KEY(A, A.heap-size, key)
```

10)
```
HEAP-DELETE(A, i)
1   if A[i]  >  A[A.heap-size]
2     A[i]  =  A[A.heap-size]
3     MAX-HEAPIFY(A, i)
4   else
5      HEAP-INCREASE-KEY(A, i, A[A.heap-size])
6      A.heap-size = A.heap-size - 1
```

11) .

12) .

	1	2	3	4	5	6	7	8	9	10	11	12
B	0	1	5	4	3	2	2	1	5	2	2	3

	0	1	2	3	4	5
C	1	2	4	2	1	2

(a)

	0	1	2	3	4	5
C	1	3	7	9	10	12

(b)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	5	4	3	2	2	1	3	2	2	3

	0	1	2	3	4	5
C	1	3	7	8	10	12

©

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	5	4	3	2	2	1	3	2	2	3

	0	1	2	3	4	5
C	1	3	6	8	10	12

(d)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	5	4	3	2	2	1	3	2	2	3

	0	1	2	3	4	5
C	1	3	5	8	10	12

(e)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	5	4	3	2	2	1	3	2	2	5

	0	1	2	3	4	5
C	1	3	5	8	10	11

(f)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	4	3	2	2	1	3	2	2	5

	0	1	2	3	4	5
C	1	2	5	8	10	11

(g)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	4	2	2	2	1	3	2	2	5

	0	1	2	3	4	5
C	1	2	4	8	10	11

(h)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	2	2	2	2	1	3	2	2	5

	0	1	2	3	4	5
C	1	2	3	8	10	11

(i)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	2	2	2	2	3	3	2	2	5

	0	1	2	3	4	5
C	1	2	3	7	10	11

(j)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	2	2	2	2	3	3	4	2	5

	0	1	2	3	4	5
C	1	2	3	7	9	11

(k)

	1	2	3	4	5	6	7	8	9	10	11	12
R	0	1	1	2	2	2	2	3	3	4	5	5

	0	1	2	3	4	5
C	1	2	3	7	9	10

(l)

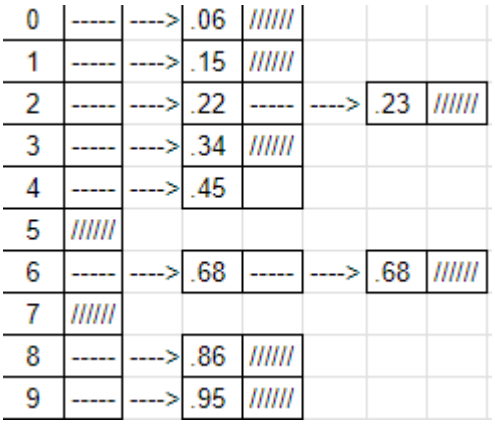
13) .

FER	FE**A**	F**A**A	**J**AA
LUZ	LU**B**	L**A**B	**F**AB
LIA	LI**B**	L**A**B	**F**AB
TAT	TA**C**	TA**C**	**F**AC
NIC	NI**L**	NA**L**	**G**AL
JOS	JO**L**	J**A**L	**L**AL
LAU	LA**L**	L**E**L	**L**EL
MAR	MA**R**	M**E**R	**T**ER
FAB	FA**R**	F**E**R	**M**ER
FEL	FE**R**	F**E**R	**M**ER
GAB	GA**R**	G**E**R	**M**ER
TAL	TA**S**	T**I**S	NIS
TER	TE**T**	T**I**T	**T**IT
MER	ME**U**	M**O**U	**T**OU
MEL	ME**Z**	M**U**Z	**T**UZ

14) .

1	.68
2	.23
3	.45
4	.95
5	.15
6	.34
7	.22
8	.86
9	.06
10	.68

(a)



(b)

15)

PUSH(S,4) | 4
PUSH(S,1) | 4 1
PUSH(S,3) | 4 1 3
POP(S) | 4 1
PUSH(S,8) | 4 1 8
POP(S) | 4 1

16) Implementación de un Stack utilizando dos colas.

a) Tenemos dos colas y marcamos una de ellas como activa.
b) Push pone en cola un elemento en la cola activa.
c) Pop debe poner en cola todos los elementos menos uno de la cola activa y ponerlos en inactiva
d) Luego, se invierten los papeles de las colas y se devuelve el último elemento que queda en la cola inactiva
e) La operación Push es Θ(1). La operación Pop es Θ(n).

17) ENQUEUE(Q, 4), ENQUEUE(Q, 1), ENQUEUE(Q, 3), DEQUEUE(Q), ENQUEUE(Q, 8), and DEQUEUE(Q) on an initially empty queue Q stored in array Q[1..6].

Índice	1	2	3	4	5	6
Q	4					

a)

Índice	1	2	3	4	5	6
Q	4	1				

b)

Índice	1	2	3	4	5	6

Q	4	1	3			

c)

Índice	1	2	3	4	5	6
Q	4	1	3			

d)

Índice	1	2	3	4	5	6
Q	4	1	3	8		

e)

Índice	1	2	3	4	5	6
Q	4	1	3	8		

f)

18) Implementación de una Cola utilizando dos stacks.

a) Sean los stacks A y B.
b) ENQUEUE agrega elementos a B.
c) DEQUEUE elimina elementos de A.
d) Si A esta vació, el contenido de B se transfiere a A eliminandolos de B y agregandolos a A.
e) Así aparecen en orden inverso y se eliminan en el original.
f) La operación DEQUEUE puede realizarse en Θ(n) sí solo si A esta vacío. Si se realizan varias veces ambas operaciones el tiempo total será lineal al número de elementos.

19) LIST-INSERT

```
1  def list_insert(L, x):
2      x.next = L.head
3      L.head = x
```

20) Operaciones INSERT, DELETE y SEARCH utilizando listas circulares de enlace simple.

a) INSERT: O(1)

```
def list_insert(L, x):
    x.next = L.nil.next
    L.nil.next = x
```

b) DELETE: O(n)

```
def list_delete(L, x):
    prev = L.nil
    while prev.next != x:
        if prev.next == L.nil:
            raise ValueError("Element does not exist")
        prev = prev.next
    prev.next = x.next
```

c) SEARCH: O(n)

```
def list_search(L, k):
    x = L.nil.next
    while x != L.nil and x.key != k:
        x = x.next
    return x
```

21)

a) Representamos la combinación de matrices key, prev y next mediante una matriz múltiple A. Cada objeto de A está en el L o en la lista F, pero no en ambos.

b) El procedimiento COMPACTIFY-LIST transpone el primer objeto en L con el primer objeto en A, los segundos objetos hasta agotar la lista L.

```
def compactify_list(L, F):
    transpose(A[L.head], A[1])
    if F.head == 1:
        F.head = L.head
    L.head = 1
    l = A[L.head].next
    i = 2
    while l != NIL:
        transpose(A[l], A[i])
        if F == i:
            F = l
        l = A[l].next
        i = i + 1

def transpose(a, b):
    swap(a.prev.next, b.prev.next)
    swap(a.prev, b.prev)
    swap(a.next.prev, b.next.prev)
    swap(a.next, b.next)

def swap(x, y):
    temp = x
    x = y
    y = temp
```

22) Procedimiento que imprime todas las keys de un árbol arbitrario enraizado con n nodos en tiempo O(n).

```
def print_lcrs_tree(T):
    x = T.root
    if x is not None:
        print(x.key)
        lc = x.left_child
        if lc is not None:
            print_lcrs_tree(lc)
            rs = lc.right_sibling
            while rs is not None:
                print_lcrs_tree(rs)
                rs = rs.right_sibling
```

23) Asignamos a cada elemento una posición en el vector y establecemos el bit correspondiente a esa posición en 1 para indicar que está ahí. Para insertar un elemento cambiamos el bit en la posición correspondiente a 1. Para eliminar un elemento, cambiamos el bit a 0. Y para verificar si un elemento está presente solo verificamos si el bit en la posición correspondiente es 1. Todas las operaciones toman un tiempo de O(1).

24) A = ⟨35, 4, 51, 14, 12, 60, 58, 6⟩
h(k) = k mod 9

T	k_i
/	
/	
/	
/	
/	
/	
/	
/	
/	

a)

T	k_i
/	
/	
/	
/	
/	
/	
/	
	35
/	

b)

T	k_i
/	
/	

T	k_i
/	
/	
/	
	4
/	

/	
/	
	35
/	

c)

	51
/	
	35
/	

f)

T	k_i
/	
/	
/	
	4
/	
	51
/	
	35
/	

d)

T	k_i	
/		
/		
	12	
/	4	
	14	
	51	60
/		
	35	
/		

g)

T	k_i
/	
/	
/	
	4
	14
	51
/	
	35
/	

e)

T	k_i	
/		
/		
	12	
	4	58
	14	
	51	60
/		
	35	
/		

h)

T	k_i
/	
/	
	12
	4
	14

T	k_i		
/			
/			
	12		
	4	58	
	14		

	51	60	6
/			
	35		

/			

i)

25) Ejercicio 11.3-4 (pag 269) pero computando las posiciones para los elementos del arreglo A mostrado arriba.

Tabla hash m tamaño 1000

Función hash h(k) = $\lfloor m(kA \bmod 1) \rfloor$

A = $\left(\sqrt{5} - 1\right)/2$

Arreglo A = {35,4,51,14,12,60,58,6}

Tabla Hash	Key
631	35
472	4
519	51
652	14
416	12
82	60
845	58
708	6

26) Ejercicio 11.4-1 (pag 277) pero para los elementos del arreglo A mostrado arriba.

Open Hashing, tabla hash m de longitud 11
Arreglo A = {35,4,51,14,12,60,58,6}

Función auxiliar $h'(k) = k$

Linear Probing:
Función hash de Linear Probing $h(k, i) = (h'(k) + i) \bmod m$

Tabla Hash con Linear Probing

Tabla Hash	Key
0	
1	12
2	35
3	14
4	4
5	60
6	58
7	51
8	6
9	
10	

Quadratic Probing:
Función hash de Quadratic Probing con $c1 = 1,\ c2 = 3$

$h(k,i) = (h'(k) + 1i + 3i^2)$

Tabla Hash con Quadratic Probing

Tabla Hash	Key
0	
1	12
2	35
3	14
4	4
5	60
6	58
7	51
8	
9	
10	6

Double hashing:
Función hash de Double Hashing, teniendo
$h_1(k) = k\ ,\ h_2(k) = 1 + (k\ mod(m - 1))$

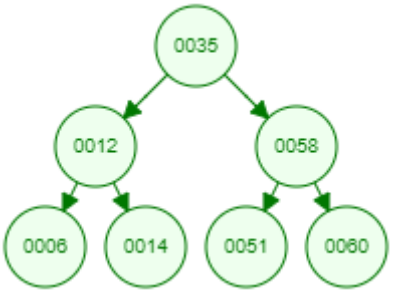
$h(k,i) = (h_1(k) + ih_2(k))\ mod\ m\ =$
$(k + i(1 + (k\ mod(m - 1))))\ mod\ m$

Tabla Hash con Double Hashing:

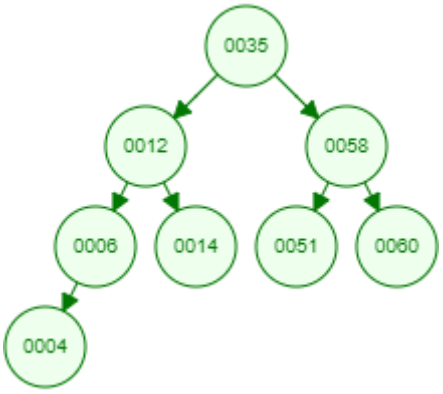
Tabla Hash	Key
0	
1	12
2	35
3	14
4	4
5	60
6	6
7	51
8	
9	
10	58

27) Ejercicio 12.1-1 (pag 289) pero para los elementos del arreglo A mostrado arriba. Dibujar árboles de búsqueda binaria con alturas 2,3,4,5 y 6
Arreglo A = {35,4,51,14,12,60,58,6}

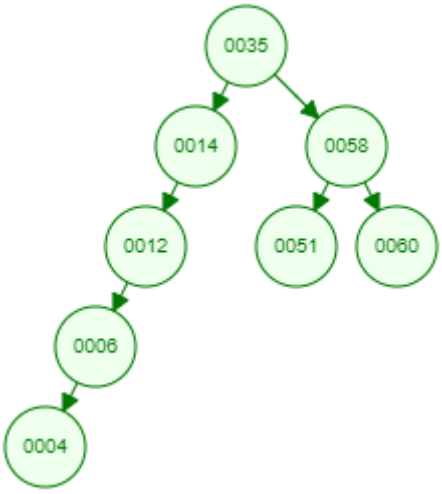
Altura 2: Para realizar un árbol binario de altura 2, uno de los requerimientos es que hayan 7 nodos, caso que no sucede, pues el arreglo A posee 8 elementos, haciendo que sea imposible construir un árbol binario de altura 2 con este, se construyó el siguiente eliminando el 4.



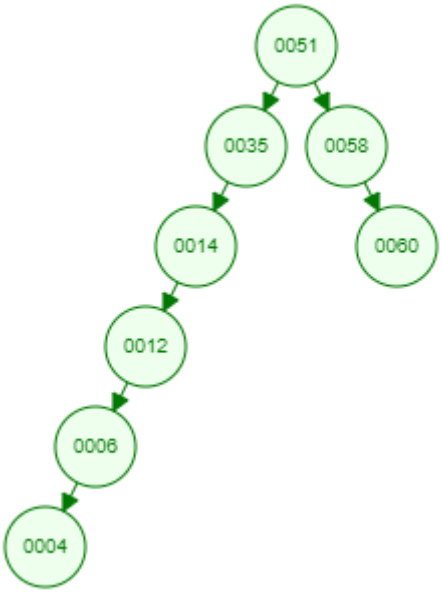
Altura 3:



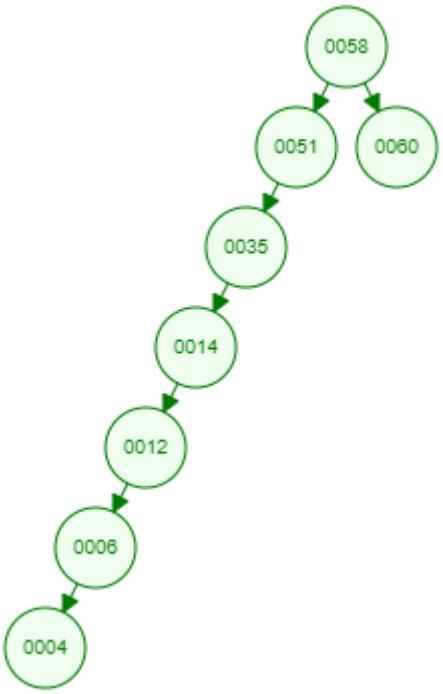
Altura 4:



Altura 5:



Altura 6:



28) Ejercicio 12.1-3 (pag 289).

```
function inorder(treel):
        stack = new stack()
        current = tree.root

        while stack is not empty or current != null:
                if current != null:
                        stack.push(current)
                        current = current.left_son
                else:
                        node = stack.pop()
                        print node.value
                        current = node.right_son
```

29) Ejercicio 12.1-1 (pag 289) pero para los elementos del arreglo A mostrado arriba.
Ya se realizó en el punto 27.

30) Ejercicio 12.2-2 (pag 293)

Versión recursiva de Tree-Minimum:

```
function treemin(node):
        if node == null:
                return null

        if node.left_son == null:
                return node.value

        return treemin(node.left_son)
```

Versión recursiva de Tree-Maximum:

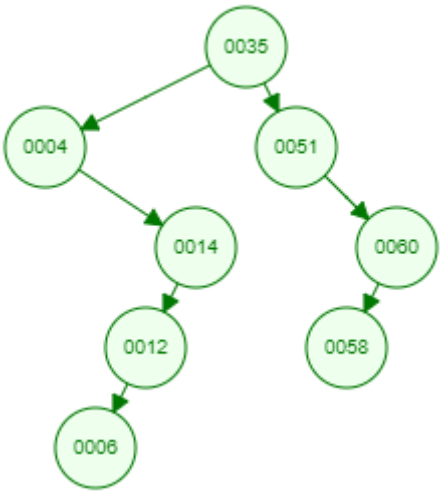
```
function treemax(node):
    if node == null:
        return null

    if node.right_son == null:
        return node.value

    return treemin(node.right_son)
```

31) Ejercicio 12.2-7 (pag 293)
Probar que invocando Tree-Minium y haciendo n-1 llamadas a Tree-successor, está llamando a un algoritmo con $\Theta(n)$ tiempo

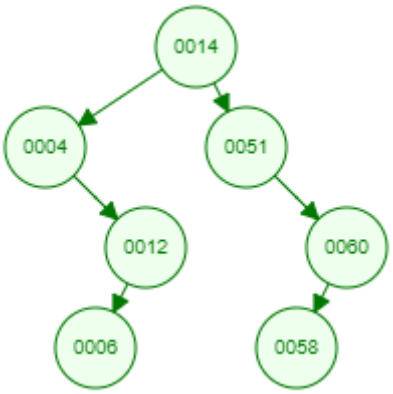
Se puede argumentar que este algoritmo tiene esta complejidad de tiempo, es que se recorre 2 veces cada arista al momento de usar cada llamado a Tree-successor, y como el número de aristas siempre es el número de vértices (Elementos, n), entonces tendríamos una complejidad 2(n-1), lo cuál nos deja con 2n-2, que tomando la parte de la variable, sí sería $\Theta(n)$

32) Dibuje el árbol binario que resulta de insertar los elementos del arreglo A, en orden ascendente de posición (en el arreglo), en un árbol binario inicialmente vacío. Luego elimine la raíz y dibuje el árbol resultante
Arreglo A = {35,4,51,14,12,60,58,6}

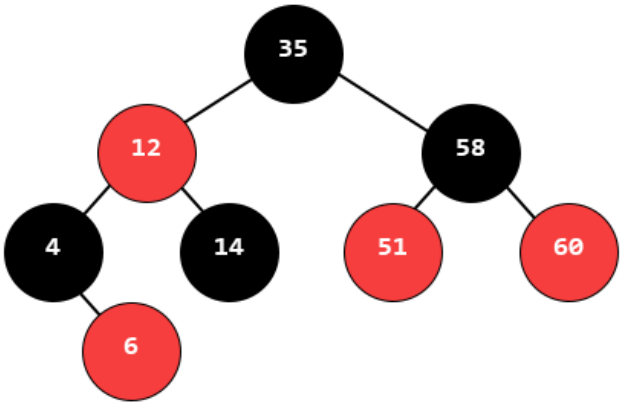
a)



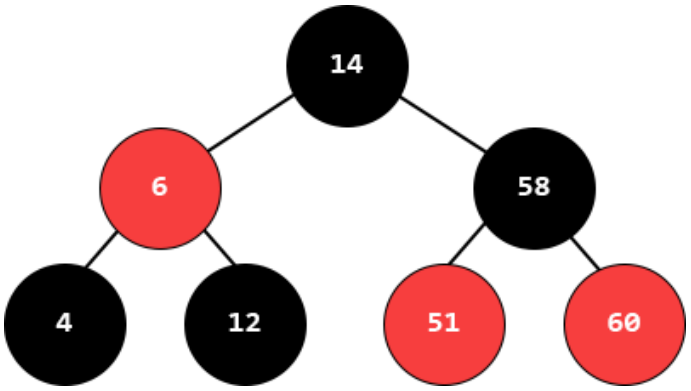
b) Como se tenía un árbol con hijo izquierdo e hijo derecho, para hallar la nueva raíz, se busca el nodo con mayor valor en el sub-árbol izquierdo, siendo este el 14, una vez se hace esto, la nueva raíz será 14, mientras que su hijo izquierdazo, el 12, será ahora el hijo derecho del 4, el 6 seguirá siendo hijo izquierdo del 12.



33) Ejercicio 13.3-2 (pag 322) insertando los elementos del arreglo A, en orden ascendente de posición (en el arreglo), en un árbol rojinegro inicialmente vacío. Muestre cómo queda el árbol al borrar la raíz.
Arreglo A = {35,4,51,14,12,60,58,6}



Así queda el árbol luego de realizar la remoción de la raíz:



PUNTO B

Algoritmos de ordenamiento en tiempo lineal. Extienda la comparación del Taller 1 para incluir ahora counting-sort para tamaños de las entradas a partir de n = 5000 hasta que los experimentos duren más de 5 minutos. Esto ya que k = 5000. Específicamente,

a) **Implementar en Python el algoritmo counting-sort**

```python
def counting_sort(arr):
    # Encuentra el valor máximo en el arreglo
    max_value = max(arr)

    # Crear un arreglo de conteo con el tamaño adecuado
    count = [0] * (max_value + 1)

    # Contar la frecuencia de cada elemento
    for num in arr:
        count[num] += 1

    # Reconstruir el arreglo ordenado
    sorted_arr = []
    for i in range(len(count)):
        while count[i] > 0:
            sorted_arr.append(i)
            count[i] -= 1

    return sorted_arr
```

b) Considerar diferentes valores de n, partiendo de 5000 e ir incrementándolo. Para cada valor de n, hacer 100 experimentos y reportar la mediana de los tiempos de ejecución de estos (para dicho n). Se deben tomar valores de n hasta que el tiempo de ejecución duren más de 5 minutos. Espec´ıficamente, cada experimento consiste en los siguientes pasos:

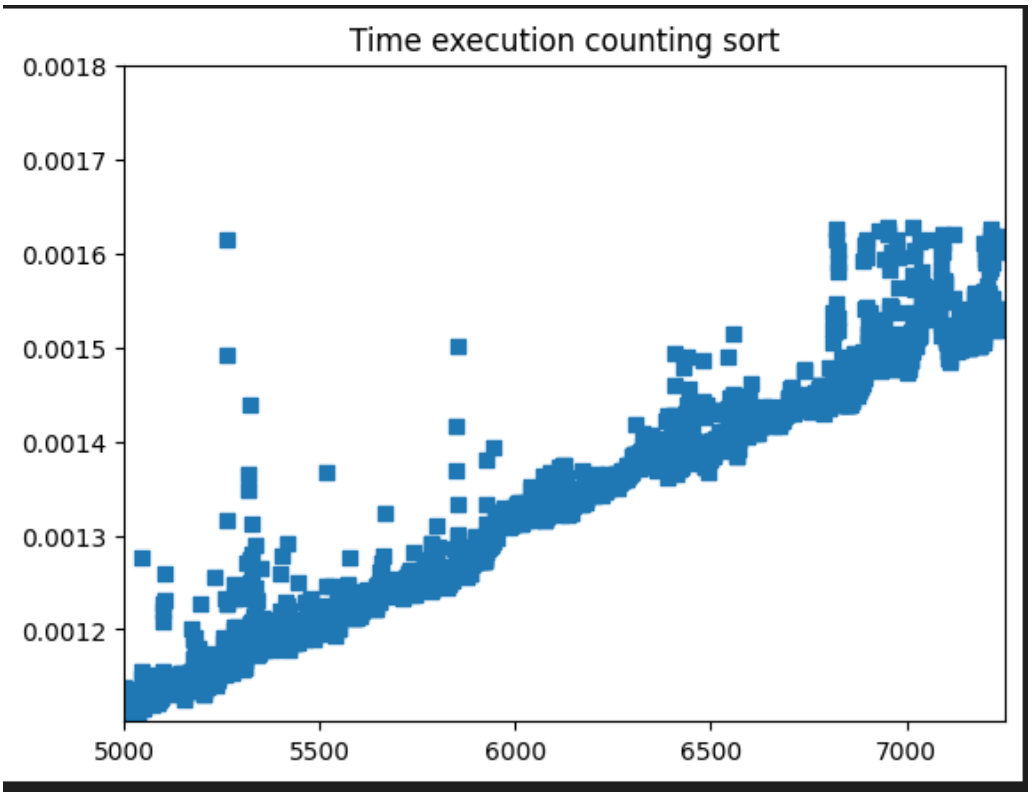
1) Generar aleatoriamente un arreglo de enteros de valores de 1 a 5000.
2) Calcular el tiempo de ejecución tomado por counting-sort para ordenar ese arreglo en especifico.

Creación del algoritmo para los tiempos de ejecución

```python
time_inicial = time.time()  # Tiempo inicio

time_final = time.time() - time_inicial
n = 5000

# DataFrame para guardar la media de tiempo de ejecucion para cada n experimento
times_sort = []
while time_final < 300:#300

    # 100 experimentos para cada valor de n

    temp_time = []


    for i in range(100):

        tiempo_experimentol_inicial = time.time()

        unsorted_array = np.random.randint(1,5000,n)
        sorted_array = counting_sort(unsorted_array)

        # Agregar tiempo de ejecucion para cada subexperimento de n
        temp_time.append(time.time() - tiempo_experimentol_inicial)

    # calcular promedio de cada subexperimento
    promedio = sum(temp_time)/len(temp_time)
    times_sort.append(promedio)

    time_final = time.time() - time_inicial
    n += 1

# Creacion dataframe


```

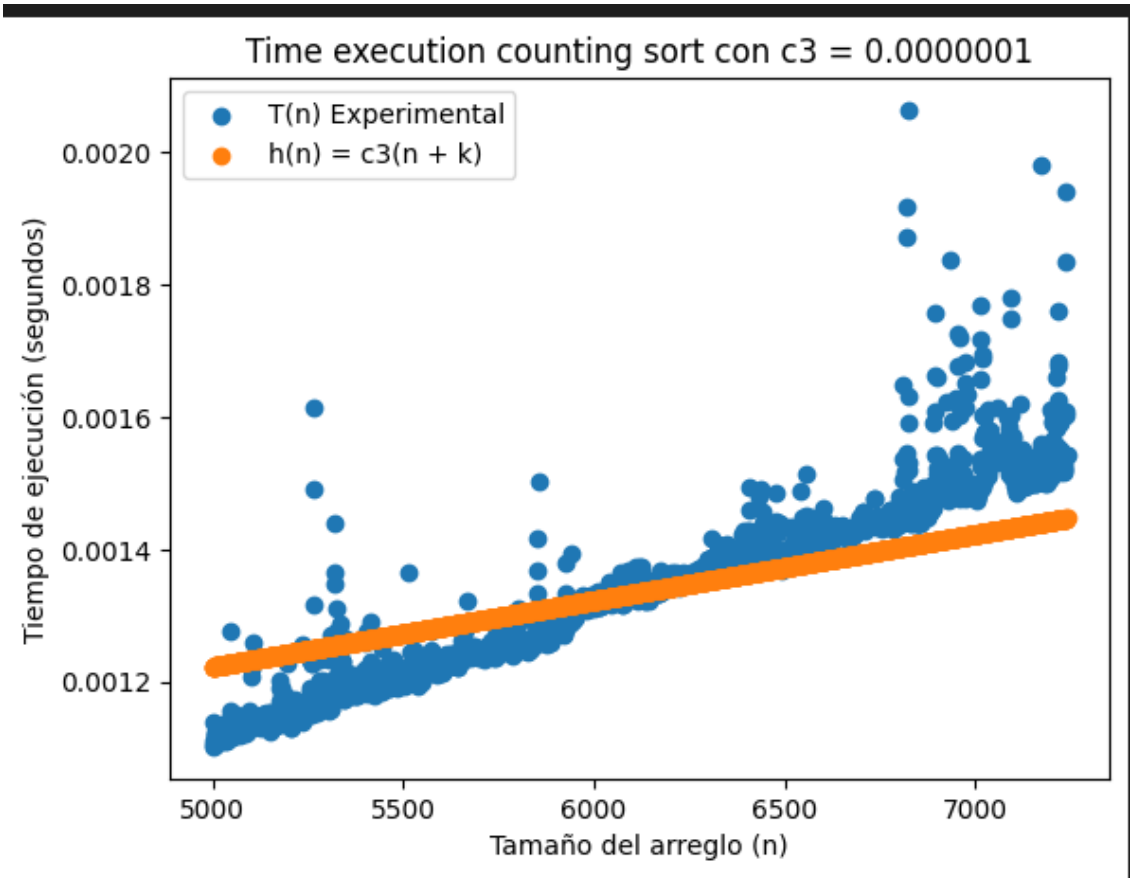
5m 0.1s

Gráfica de tiempo de ejecución counting sort



d) Encuentre las constantes c3 y n0 que demuestran que el tiempo de la ejecución experimental de counting-sort es T (n) = O(n + k). Sobre la gráfica de los resultados experimentales también graficar la función h(n) = c3(n + k). Debe especificar cuál es c3 y n0. Las gráficas deben ser claras. Se debe poder distinguir claramente las líneas y se debe especificar cuál es cada una.

Para resolver este punto se gráfico los tiempos de ejecución y se fue probando manualmente hasta encontrar que el valor de C3 es de 0.0000001 y el valor de k se tomó como la mayor iteración a la que se llegó para posteriormente graficarlos en una misma gráfica. A continuación se muestra la gráfica y el código correspondiente

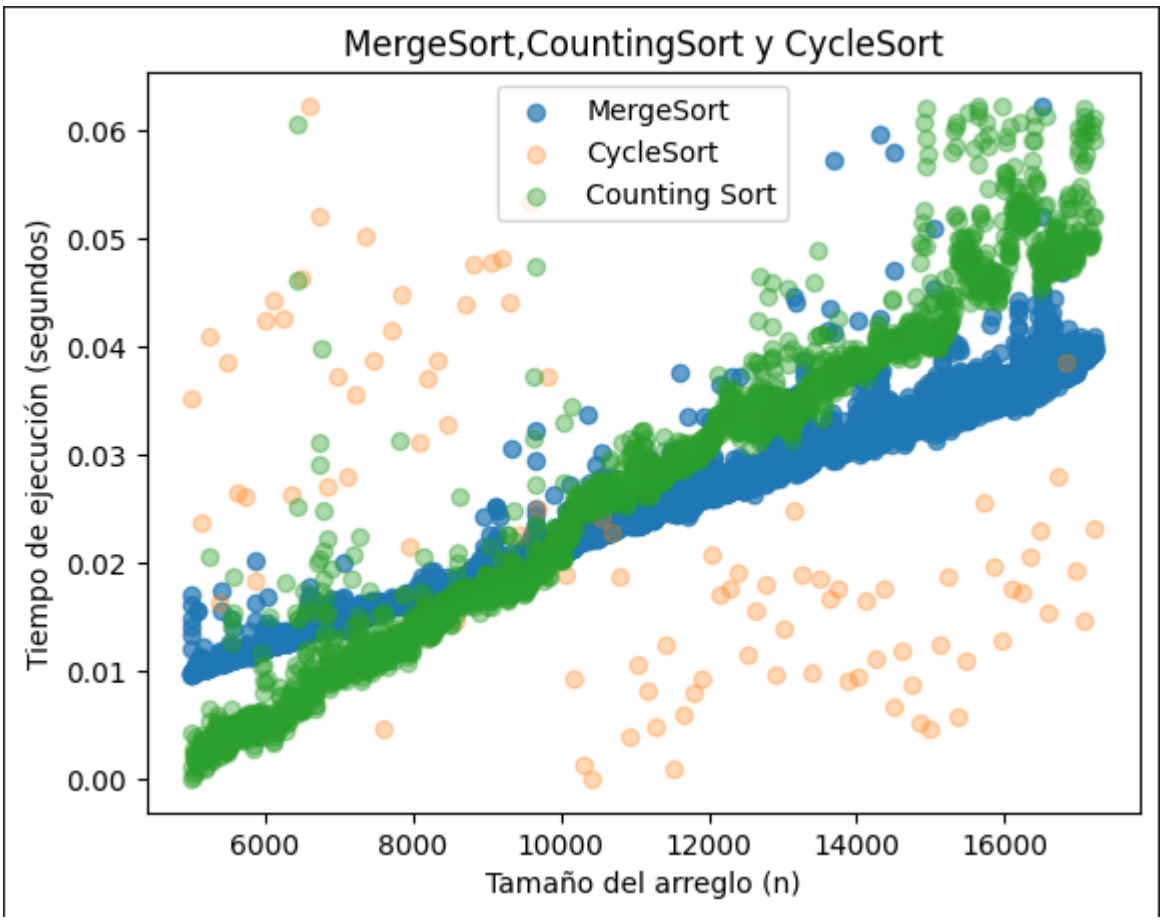



```
tamaños = np.array(list(range(5000,n)))

c3 = 0.0000001 # Valor arbitrario para c3
k = tamaños[-1] # Tomamos el máximo tamaño de arreglo como el rango 'k'
h_n = c3 * (tamaños + k)

plt.scatter(tamaños, times_sort, label='T(n) Experimental')
plt.scatter(tamaños, h_n, label='h(n) = c3(n + k)')
plt.xlabel('Tamaño del arreglo (n)')
plt.title("Time execution counting sort con c3 = 0.0000001")
plt.ylabel('Tiempo de ejecución (segundos)')
plt.legend()
plt.show()
```

✓ 0.1s

anteriormente. En una misma gráfica mostrar el tiempo experimental de los algoritmos merge-sort, cycle-sort y counting-sort para los diferentes valores de n trabajados.

Para hacer este punto realizamos los algoritmos y los corrimos cada uno por 5 minutos, nos dimos cuenta que el algoritmo cyclesort es bastante lento a comparación del merge sort o counting sort y es por ello que se escalaron los datos para manejar un mismo rango de datos tanto para el tamaño de los datos como para los tiempos de ejecución y así poder obtener la siguiente gráfica en donde vemos que el mergesort es similar a su complejidad O(nlogn) como el counting sort que es O(n) y el cycle sort se parece un poco a $O(n^2)$.



Código para escalar datos

```
# Escalar datos
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler(feature_range=(0,df_merge_sort['Time Execution'].max()))

scaler2 = MinMaxScaler(feature_range=(5000,n_merge))


scaled_column = pd.DataFrame(scaler.fit_transform(df_cycle_sort[['Time Execution']]), columns=['Time Execution'])
scaled_column_2 = pd.DataFrame(scaler2.fit_transform(df_cycle_sort[['Size']]), columns=['Size'])
scaled_column_3 = pd.DataFrame(scaler2.fit_transform(df_counting_sort[['Size']]), columns=['Size'])
scaled_column_4 = pd.DataFrame(scaler.fit_transform(df_counting_sort[['Time Execution']]), columns=['Time Execution'])

df_cycle_sort['Time Execution'] = scaled_column['Time Execution']
df_counting_sort['Time Execution'] = scaled_column_4['Time Execution']
df_cycle_sort['Size'] = scaled_column_2['Size']
df_counting_sort['Size'] = scaled_column_3['Size']

plt.scatter(df_merge_sort['Size'], df_merge_sort['Time Execution'], label='MergeSort',alpha = 0.7)
plt.scatter(df_cycle_sort['Size'], df_cycle_sort['Time Execution'], label='CycleSort',alpha = 0.3)
plt.scatter(df_counting_sort['Size'], df_counting_sort['Time Execution'], label='Counting Sort',alpha = 0.4)
plt.xlabel('Tamaño del arreglo (n)')
plt.title("MergeSort,CountingSort y CycleSort")
plt.ylabel('Tiempo de ejecución (segundos)')
plt.legend()
plt.show()
```

Código de ejecución merge sort()

```
time_inicial = time.time()  # Tiempo inicio

time_final = time.time() - time_inicial
n_merge = 5000

# DataFrame para guardar la media de tiempo de ejecucion para cada n experimento
tine_sort_merge = []
while time_final < 300:#300

    tiempo_experimentol_inicial = time.time()

    unsorted_array = np.random.randint(1,5000,n_merge)
    sorted_array = merge_sort(unsorted_array)

    # Agregar tiempo de ejecucion para cada subexperimento de n
    # calcular promedio de cada subexperimento

    tine_sort_merge.append(time.time() - tiempo_experimentol_inicial)

    time_final = time.time() - time_inicial
    n_merge += 1

```

Código de ejecución del cycle sort()

```python
time_inicial = time.time()  # Tiempo inicio

time_final = time.time() - time_inicial
n_cycle = 5000

# DataFrame para guardar la media de tiempo de ejecucion para cada n experimento
time_sort_cycle = []
while time_final < 300:#300

    tiempo_experimentol_inicial = time.time()

    unsorted_array = np.random.randint(1,5000,n_cycle)
    sorted_array = cycle_sort(unsorted_array)

    # Agregar tiempo de ejecucion para cada subexperimento de n
    # calcular promedio de cada subexperimento

    time_sort_cycle.append(time.time() - tiempo_experimentol_inicial)

    time_final = time.time() - time_inicial
    n_cycle += 1

```

Finalmente, se invita a revisar el notebook para ver los códigos anteriormente descritos.