Ejercicio 1:

Demuestre que $6n^3 \neq O(n^2)$.

$F(n) = c.n^2$, $c \geq 1$

$O(F(n)) = O(n^2)$

$F'(n) = 6n^3$, $c'=6$

$O(F'(n)) = O(n^3)$, ya que se eliminan todas las constantes y solo queda la variable n de mayor exponente.

Por lo tanto, $O(F(n)) = O(n^2) \neq O(n^3) = O(F(n))$

Ejercicio 2:

¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n)?

Para el mejor caso, el primer pivote sería el elemento del medio por lo que la primer partición el arreglo quedaría equilibrado. Y, siguiendo la partición en la función recursiva hasta tener solo un elemento, lo ideal sería que cada vez que tengo un pivote nuevo y realizo otra partición, me vaya quedando la lista equilibrada.

Ejercicio 3:

¿Cuál es el tiempo de ejecución de la estrategia Quicksort(A), Insertion-Sort(A) y Merge-Sort(A) cuando todos los elementos del array A tienen el mismo valor?

Insertion-Sort(A): $\Omega(n)$ (mejor de los casos)

Merge-Sort(A): $O(n \log n)$

Quicksort(A): $O(n^2)$ (peor de los casos)

Ejercicio 4:

Implementar un algoritmo que ordene una lista de elementos donde siempre el elemento del medio de la lista contiene antes que él en la lista la mitad de los elementos menores que él. Explique la estrategia de ordenación utilizada.

Ejemplo de lista de entrada

7	3	2	8	5	4	1	6	10	9

ejercicio4.py > Node

```
30  medioPos = math.trunc(((length(L))/2)-1) #(posicion del elemento del medio)
31  cant = medioPos  #(cantidad de elementos antes del medio)
32  elemM = cant/2
33  elemm = cant/2   #(cantidad de elementos menores y mayores antes del medio)
34
35  medio = L.head
36  for i in range(0,medioPos):
37  |  medio = medio.nextNode
38
39  current = L.head
40  #variables m y M me cuenta la cant de elementos menores y mayores que medio antes de el en la lista
41  m = 0
42  M = 0
43  for i in range(0,medioPos):
44  |  if current.value < medio.value:
45  |  |  m += 1
46  |  elif current.value > medio.value:
47  |  |  M += 1
48  |  current = current.nextNode
49
50  #Si m>elemm intercambio un valor menor de la izq por otro mayor de la der (de medio), caso contrario si m < 
51
```

```
52  def menorXmayor(L,medioPos):
53  |  #busco menor
54  |  menor = L.head
55  |  for i in range (0,medioPos):
56  |  |  if menor.value < medio.value:
57  |  |  |  menorPos = i
58  |  |  |  break
59  |  |  else:
60  |  |  |  menor = menor.nextNode
61  |  #busco mayor
62  |  mayor = medio.nextNode
63  |  for i in range (0,medioPos):
64  |  |  if mayor.value > medio.value:
65  |  |  |  mayorPos = i + medioPos
66  |  |  |  break
67  |  |  else:
68  |  |  |  mayor = mayor.nextNode
69  |  update(L,mayor.value,menorPos)
70  |  update(L,menor.value,mayorPos)
71  |  return L
```

```
73  def mayorXmenor(L,medioPos):
74      
75      #busco mayor
76      mayor = L.head
77      for i in range (0,medioPos):
78          if mayor.value > medio.value:
79              mayorPos = i
80              break
81          else:
82              mayor = mayor.nextNode
83      
84      #busco menor
85      menor = medio.nextNode
86      for i in range (0,medioPos):
87          if menor.value < medio.value:
88              menorPos = i + medioPos +1
89              break
90          else:
91              menor = menor.nextNode
92      
93      auxMayorValue = mayor.value
94      update(L,menor.value,mayorPos)
95      update(L,auxMayorValue,menorPos)
96      return L
```

```
98  if m > elemm:
99      menorXmayor(L,medioPos)
100 else:
101     mayorXmenor(L,medioPos)
102 print('Lista final:')
103 printLista(L)
```

Ejercicio 5:

Implementar un algoritmo Contiene-Suma(A,n) que recibe una lista de enteros A y un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el costo computacional.

```
def ContieneSuma(L,n):
    element = L.head
    current= L.head
    ContieneSumaR(L, n, element, current.nextNode)

def ContieneSumaR(L,n,element,current):
    for i in range(length(L)-1):
        if current != element and (element.value + current.value) == 7:
            return print ('True')
            break
        else:
            current = current.nextNode
    if current == element:
        return print('False')
    else:
        ContieneSumaR(L,n,element.nextNode,L.head)

ContieneSuma(L,n)
```

Como uso una función recursiva (que la llamo n-1 veces en el peor caso) y dentro de ella hay un bucle for, la complejidad es de $O(n^2)$

Ejercicio 6:

Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o RadixSort, brindando un ejemplo que explique su funcionamiento en un caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

HeapSort:

Una forma sencilla de implementar una cola de prioridad es utilizando listas. Sin embargo, insertar en una lista es O(n) y ordenar una lista es O(n log n), pero podemos hacerlo mejor: la forma clásica de implementar una cola de prioridad es usar una estructura de datos llamada binary heap . Esta estructura, nos permitirá poner en cola y eliminar elementos en un peor caso de O(log n)

Cuando diagramamos el heap se parece mucho a un árbol, pero cuando lo implementamos usamos solo una lista como representación interna. El binary heap tiene dos variaciones comunes: el min heap, en el que la clave más pequeña siempre está al frente, y el max heap, en el que el valor de clave más grande siempre está al frente

Ejemplo:

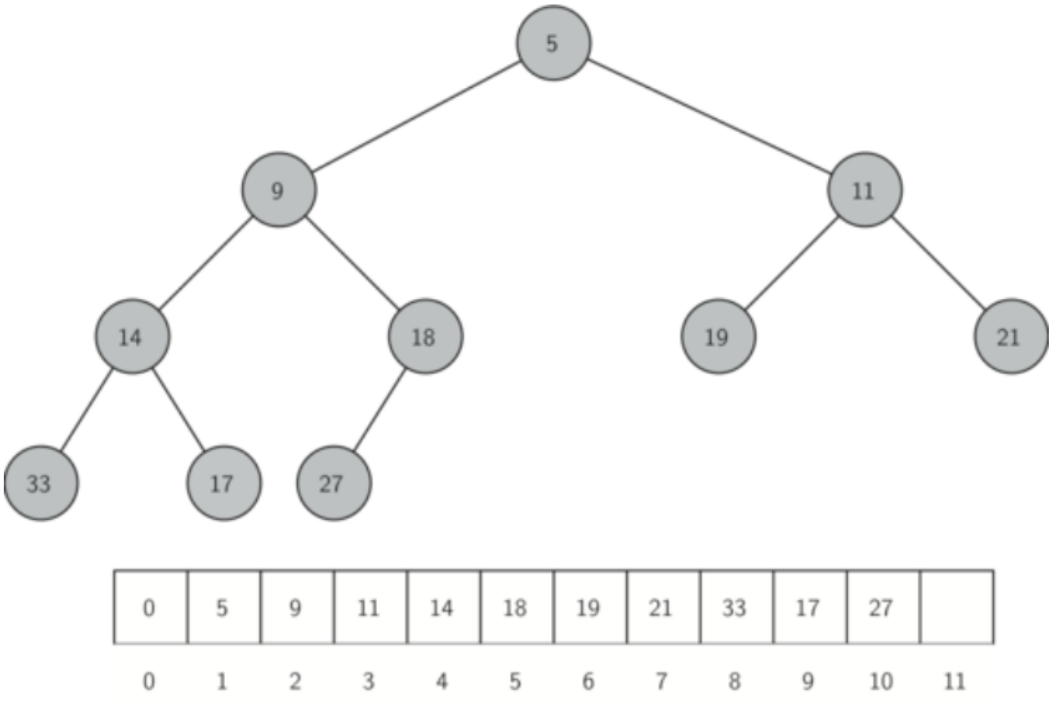
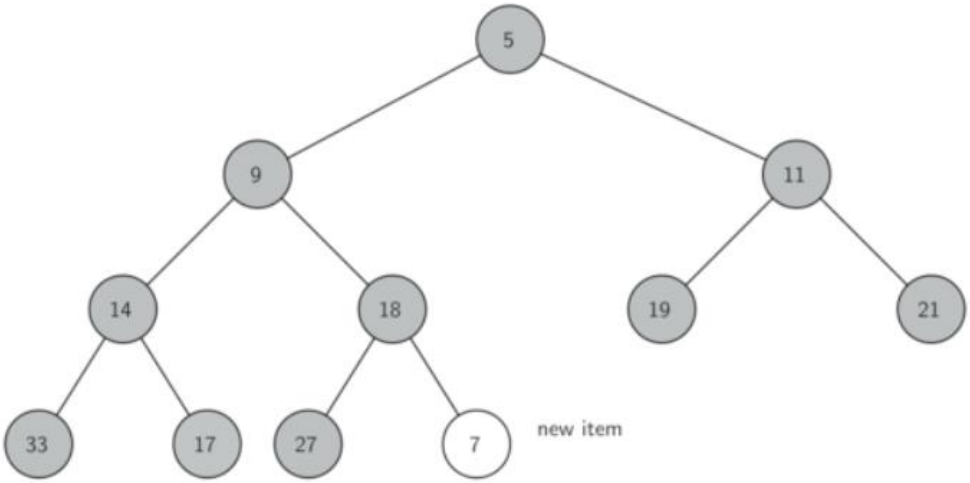
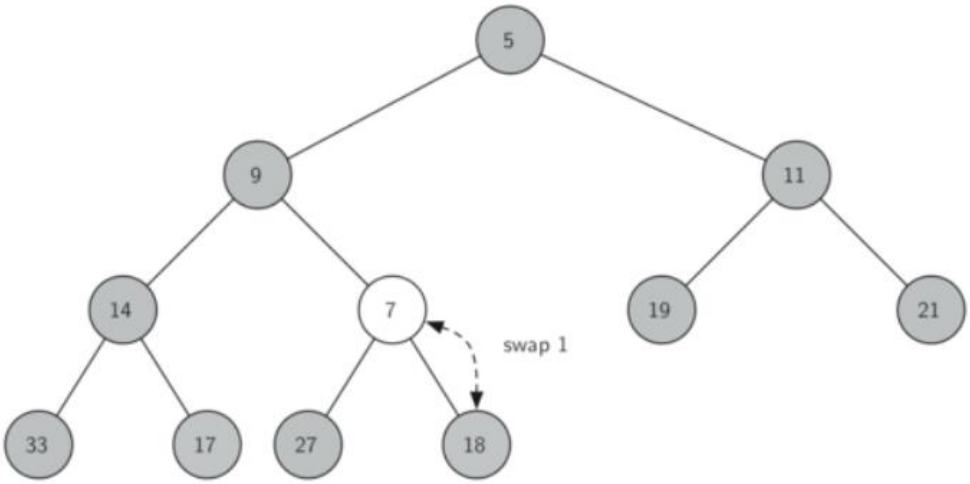


Figura 2: Un árbol binario completo, junto con su representación de lista

Un binary heap tiene un cero como primer elemento y este cero no se usa, pero está ahí para que la división entera simple se pueda usar en métodos posteriores.

En insert, la forma más fácil y eficiente de agregar un elemento a una lista es simplemente agregar el elemento al final de la lista. Por lo que es necesario utilizar un método que nos permita recuperar la propiedad de la estructura del montón al comparar el elemento recién agregado con su padre. Si el elemento recién agregado es menor que su elemento principal, podemos intercambiar el elemento con su elemento principal.





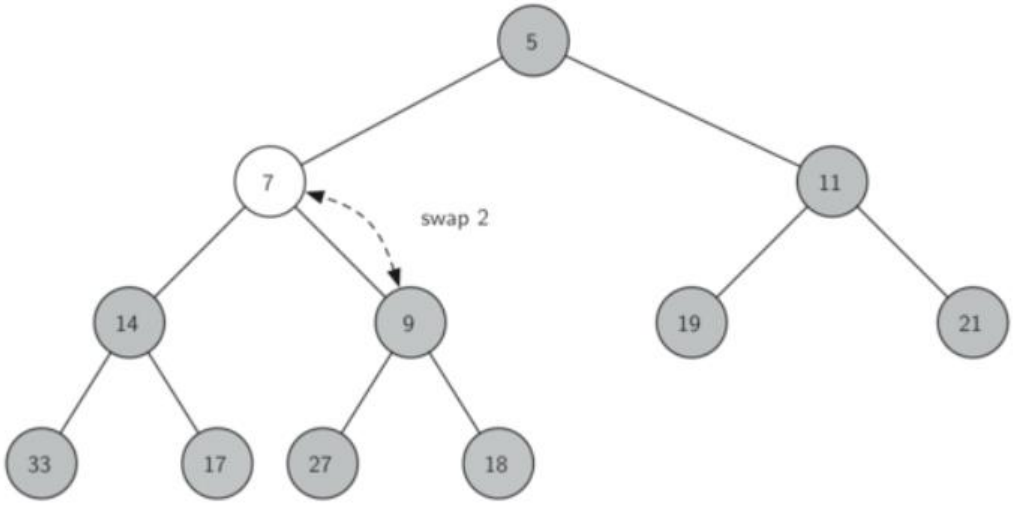
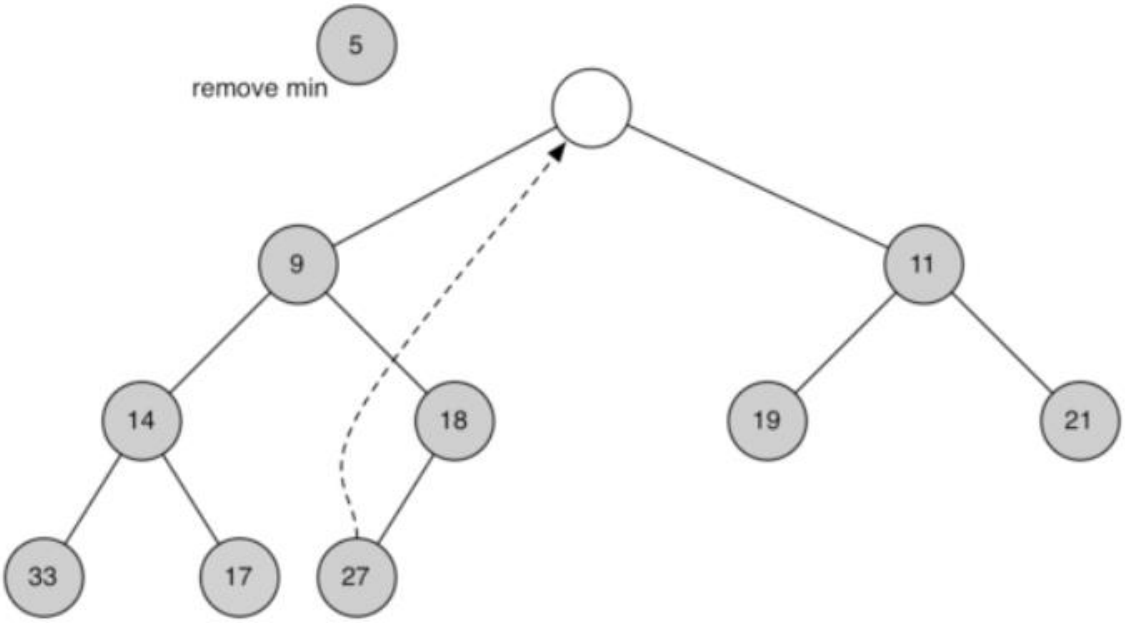
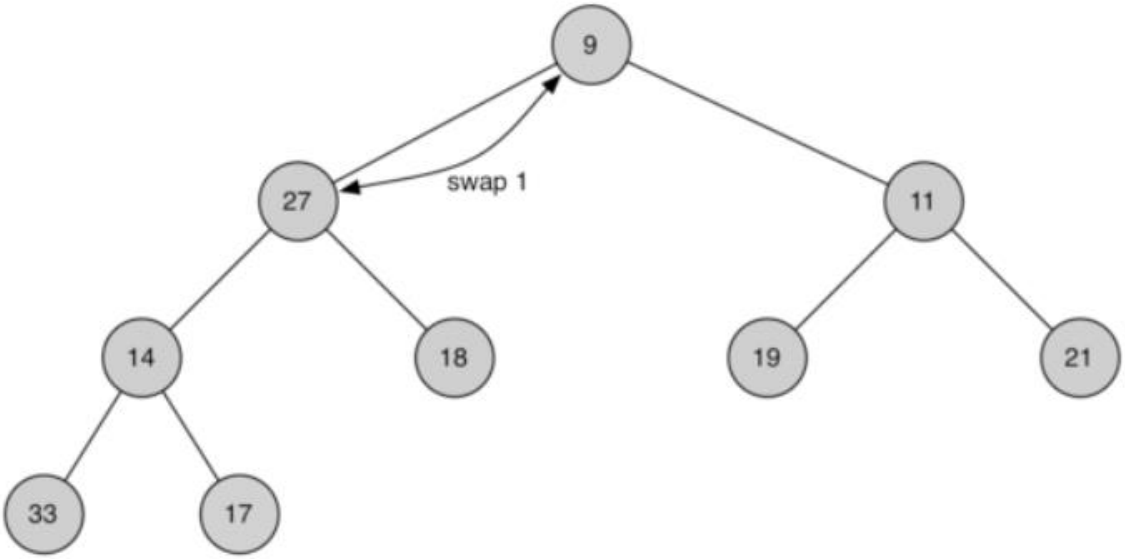


Figura 2: filtrar el nuevo nodo hasta su posición adecuada

Luego podemos realizar el min heap. Dado que la propiedad del montón requiere que la raíz del árbol sea el elemento más pequeño del árbol, es fácil encontrar el elemento mínimo. La parte difícil del min heap es restaurar el cumplimiento total con la estructura del montón y las propiedades del orden del montón después de que se eliminó la raíz. Podemos restaurar nuestro montón en dos pasos. Primero, restauraremos el elemento raíz tomando el último elemento de la lista y moviéndolo a la posición raíz. Mover el último elemento mantiene nuestra propiedad de estructura de heap. Sin embargo, probablemente hemos destruido la propiedad de orden de montón de nuestro montón binario. En segundo lugar, restauraremos la propiedad de orden del montón empujando el nuevo nodo raíz hacia abajo en el árbol hasta su posición adecuada:


5
remove min
9
11
14
18
19
21
33
17
27


9
swap 1
27
11
14
18
19
21
33
17

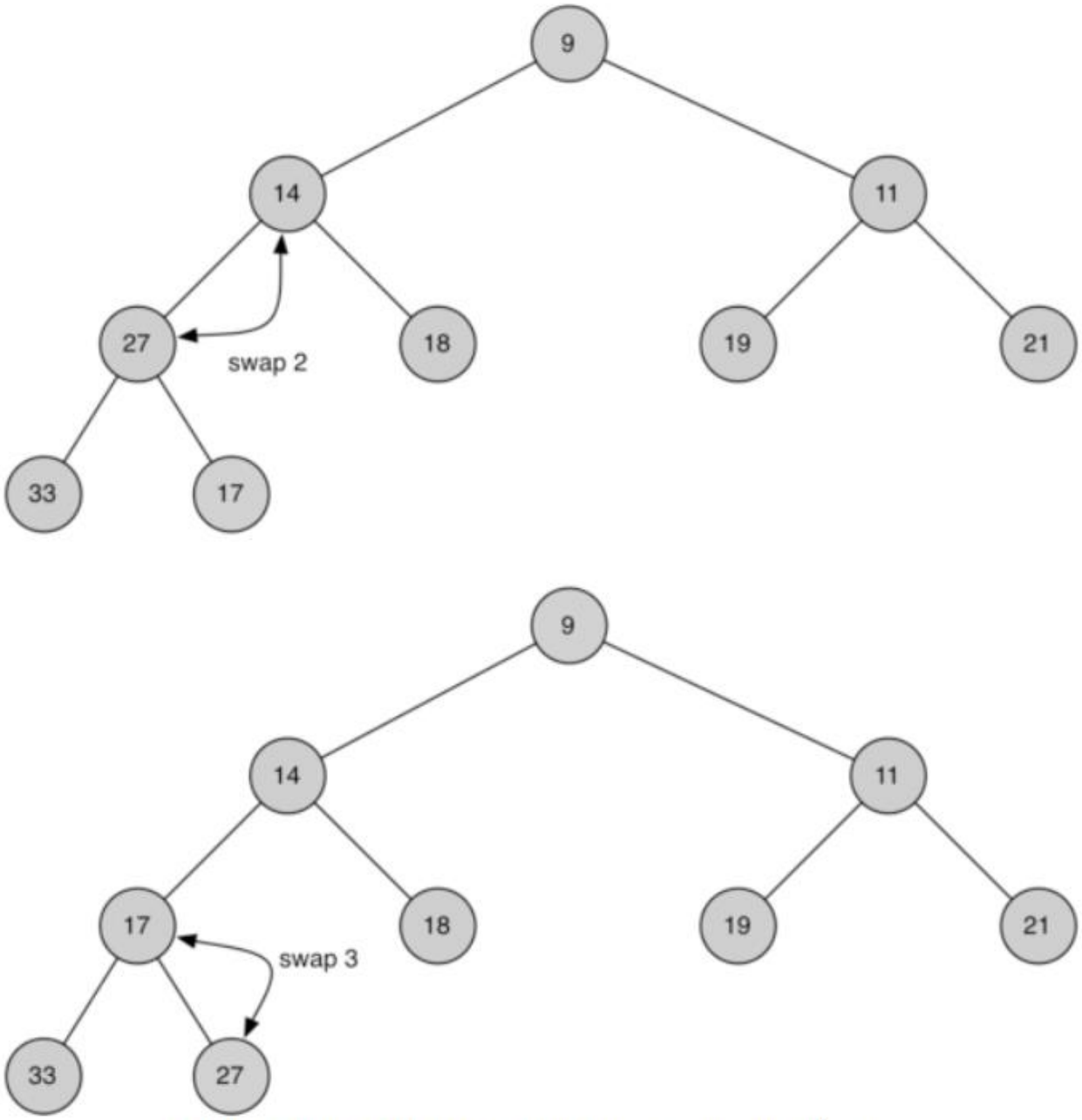


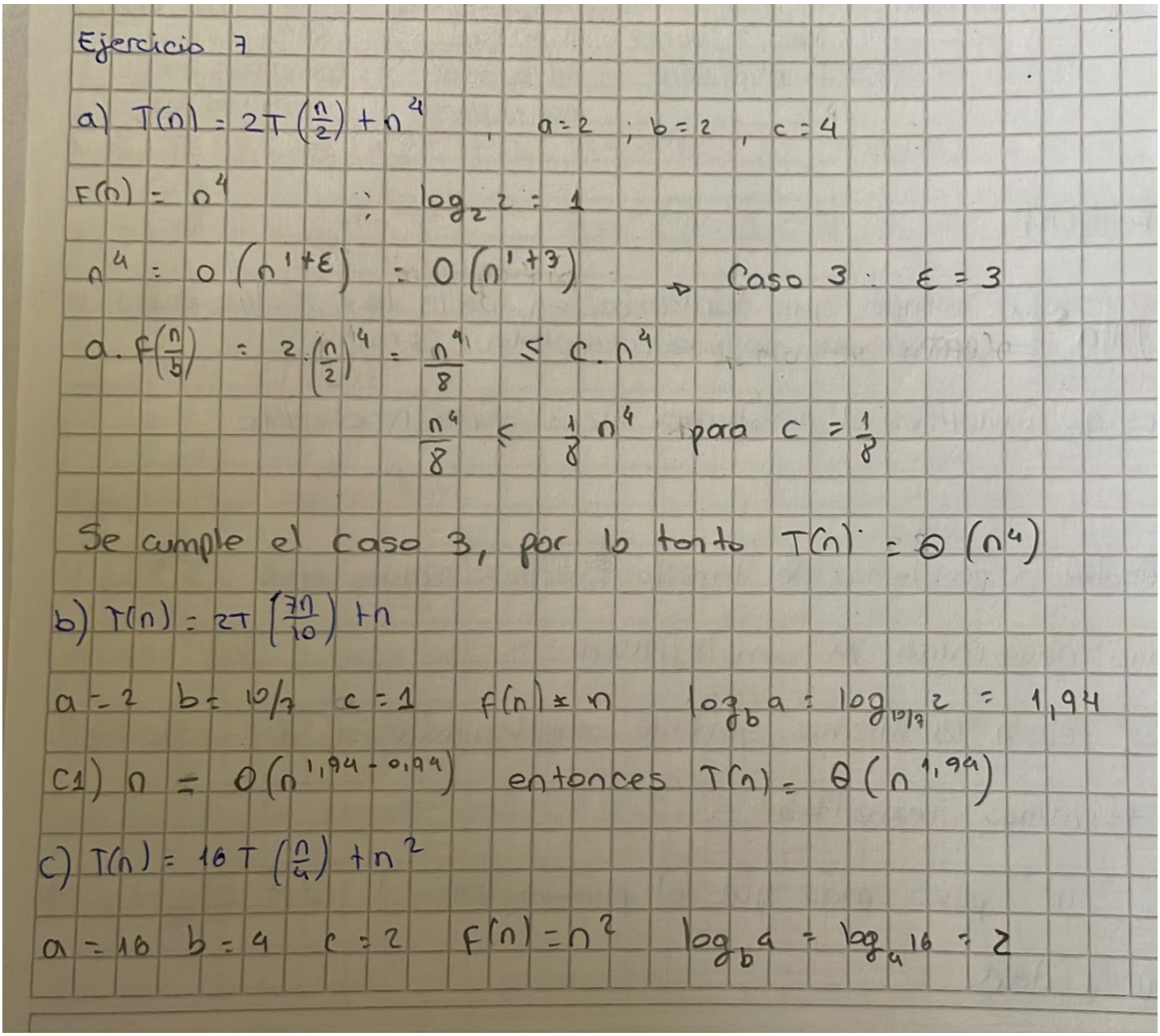
Figura 3: Filtrando el Nodo Raíz hacia abajo en el Árbol

Ejercicio 7:

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a T(n/b) + f(n) y otros 3 con el método maestro simplificado: T(n) = a T(n/b) + n^c

a. **T(n) = 2T(n/2) + n^4**

b. **T(n) = 2T(7n/10) + n**
c. **T(n) = 16T(n/4) + n^2**
d. **T(n) = 7T(n/3) + n^2**
e. **T(n) = 7T(n/2) + n^2**
f. **T(n) = 2T(n/4) + √n**



Ejercicio 7

a) $T(n) = 2T\left(\frac{n}{2}\right) + n^4$, $a=2$; $b=2$, $c=4$

$F(n) = n^4$; $\log_2 2 = 1$

$n^4 = \Omega(n^{1+\varepsilon}) = \Omega(n^{1+3})$ → Caso 3: $\varepsilon = 3$

$a \cdot f\left(\frac{n}{b}\right) = 2 \cdot \left(\frac{n}{2}\right)^4 = \frac{n^4}{8} \leq c \cdot n^4$

$\frac{n^4}{8} \leq \frac{1}{8} n^4$ para $c = \frac{1}{8}$

Se cumple el caso 3, por lo tanto $T(n) = \Theta(n^4)$

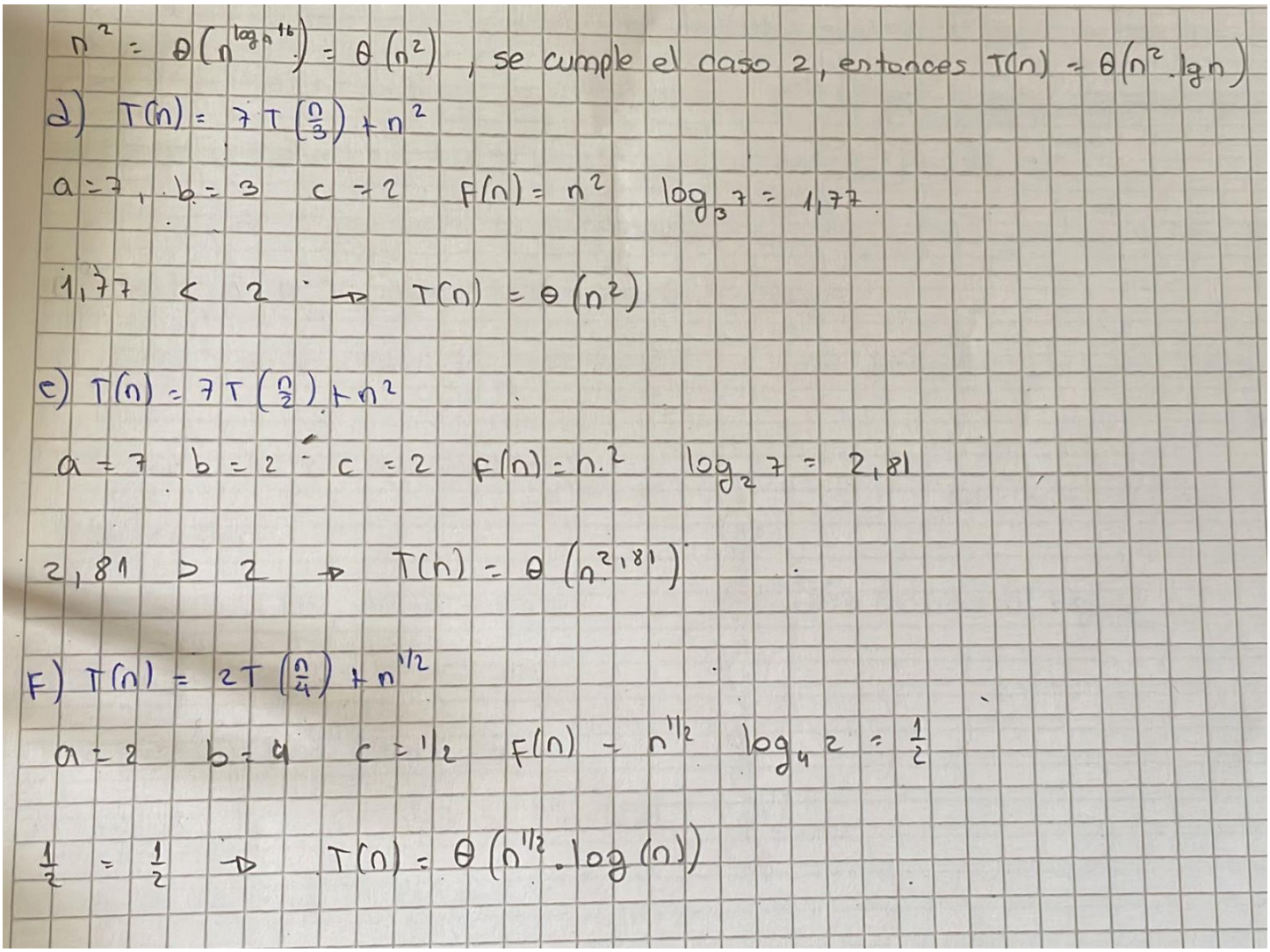
b) $T(n) = 2T\left(\frac{7n}{10}\right) + n$

$a=2$ $b=10/7$ $c=1$ $f(n)=n$ $\log_b a = \log_{10/7} 2 = 1,94$

C1) $n = O(n^{1,94-0,94})$ entonces $T(n) = \Theta(n^{1,94})$

c) $T(n) = 16T\left(\frac{n}{4}\right) + n^2$

$a=16$ $b=4$ $c=2$ $F(n)=n^2$ $\log_b a = \log_4 16 = 2$



A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca mas alla de algo1.py y linkedlist.py**~~
3. **Hacer una análisis por cada algoritmo implementado del caso mejor, el caso peor y una perspectiva del caso promedio.**