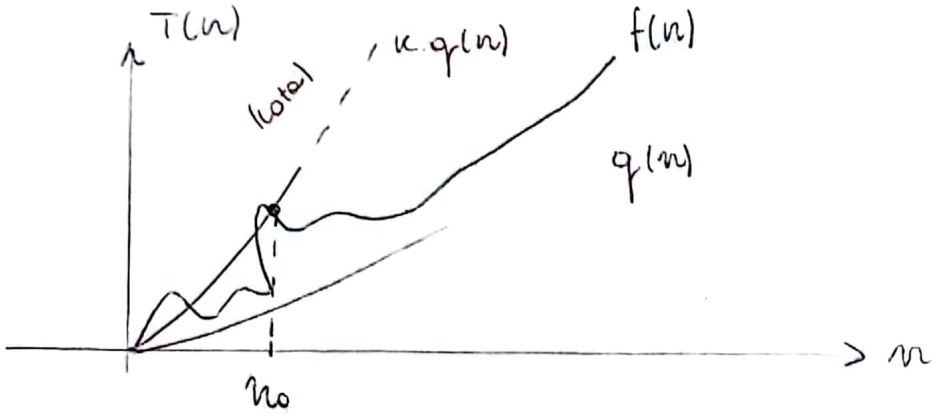
Big-Oh formal definition

Adriano Fabris - TP Algo 2

$f(n) = O(g(n))$ if $\exists k, n_0 \in \mathbb{R}^+ / f(n) \leq k \cdot g(n) \quad \forall n \geq n_0$

(1)



"A partir de n_0, $f(n)$ nunca sobrepasará a $k.g(n)$"

Si $6n^3 = O(n^2) \Rightarrow \exists k, n_0 \in \mathbb{R} / f(n) \leq k\, g(n) \quad \forall n \geq n_0$

con $\begin{cases} f(n) = 6n^3 \\ g(n) = n^2 \end{cases}$ $\quad 6n^3 \leq k \cdot n^2 \Rightarrow n^3 \leq k_0 n^2 \Rightarrow k_0 = \frac{k}{6} \geq n$

$k_0 = k/6$

Como $k = f(n)$, k no es cte, luego por absurdo $6n^3 \neq O(n^2)$. QED.

(2) Para quicksort no existe mejor caso "a priori", ya que el arreglo viene ordenado al azar. Sí un mejor caso cuando la elección del pivot sea la mediana del arreglo (y cada subarreglo sucesivo).

La mediana divide a un conjunto en 2 partes (1 tiene todo $\geq$; a ella otro tiene todo $\leq$

1, 2, (3), 4, 5 → Med = 3

1, 2, 3, 20, (50) 70, 80, 97, 1001020 → Med = 50

(3) QuickSort: $O(n^2)$, ya que la partición se achica en un sola unidad, hasta acabar.

InsertionSort: $O(n)$, como no encuentra menor, sólo avanza hasta llegar al final

MergeSort: $O(n \log n)$, independientemente de la condición inicial, MergeSort debe dividir y rearmar la lista final.

Ej 7

Orden ascendente en vel. crecimiento

a) $T(n) = 2T(n/2) + n^4$

$\log_2^2 = 1$
$k = 4$
} caso 3 con $p \leq 0 \Rightarrow \Theta(n^4)$ ⑥

b) $T(n) = 2T(7n/10) + n$

$\log_{\frac{10}{7}} 2 \cong 1,94$
$k = 1$
} caso 1 con $\Rightarrow \Theta(n^2)$ ③

c) $T(n) = 16T(n/4) + n^2$

$\log_4 16 = 2$
$k = 2$
} caso 2 con $p > -1 \Rightarrow \Theta(n^2 \log(n))$ ②

d) $T(n) = 7T(n/3) + n^2$

$\log_3 7 \cong 1,77$
$k = 2$
} caso 3 con $p \leq 0 \Rightarrow \Theta(n^2)$ ④

e) $T(n) = 7T(n/2) + n^2$

$\log_2^7 \cong 2,8$
$k = 2$
} caso 1 $\Rightarrow \Theta(n^3)$ ⑤

f) $T(n) = 2T(n/4) + \sqrt{n}$

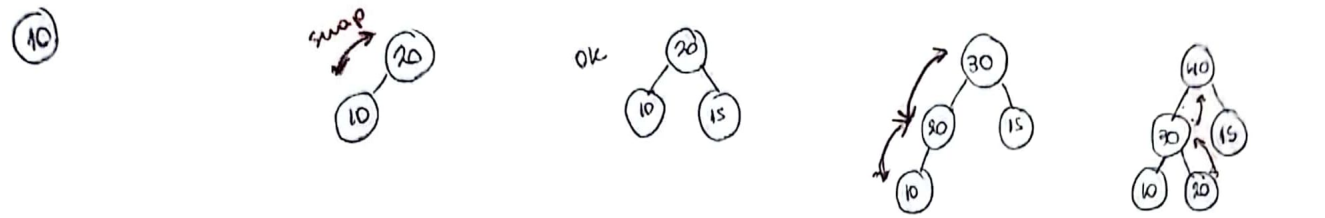
$\log_4 2 = 1/2$
$k = 1/2$
} caso 2 con $p > -1 \Rightarrow \Theta(\sqrt{n} \cdot \log n)$ ①

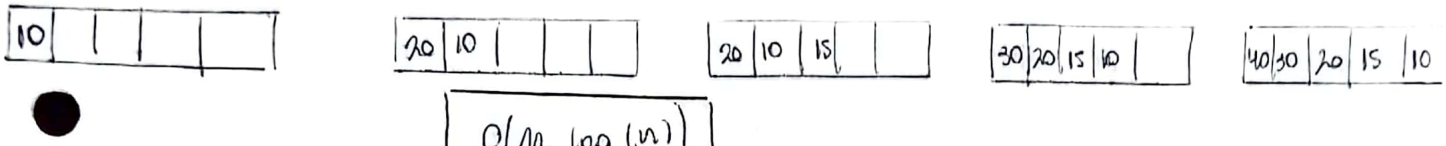
Ej 6

HeapSort: Dada una lista, primero se ① crea el binary heap (máx en este caso) y luego de eliminar todos sus nodos. Devolviendo así una lista ordenada ②.

CreateHeap(Array n) crea un BH (máx) a partir de un array. Ej. HeapSort( )

10	20	15	30	40



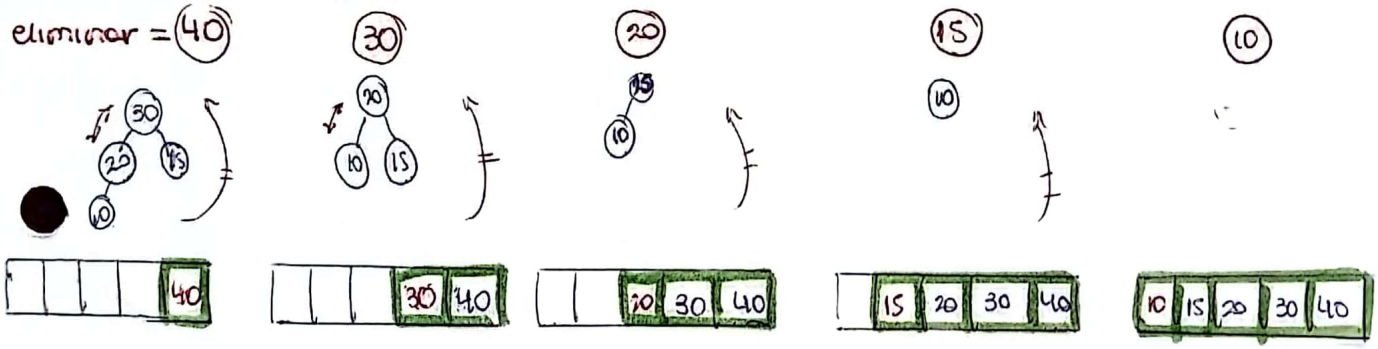


$O(n \cdot \log(n))$

long. lista a ingresar → ya que, a lo sumo se necesitan $h = \log(n)$ swap para reacomodar.

HeapSort(): elimina los elementos de BH, y por su forma de eliminar (y almacenar), termina ordenando en la lista auxiliar. El proceso trata de eliminar la raíz y

a eliminar = 40



: lista auxiliar que devuelve la lista ordenada, también cumple la función de reserva de datos (no los pierde).

- acomodar, y así sucesivamente hasta que no queden nodos.

se acomoda, pasando el último del árbol (ordenado en amplitud) hacia la raíz y de allí desciende (haciendo swap) hasta encontrar hijos menores; allí queda.

de nuevo, $O(n \log(n))$ ⇒ Caso prom: $2n \log(n) \Rightarrow O(n \log n)$

	Create Heap	Heap Sort	Total
Mejor caso:	$O(n)$	$O(n \log(n))$	$O(n \log(n))$
Peor caso:	$O(n \log(n))$	$O(n \log(n))$	$O(n \log(n))$

```python
def contiene_suma(A, n):  # (O(n*logn))

    A.sort()  # (O(n*logn)) Timsort algorithm
    lo = 0
    hi = len(A) - 1

    # Mientras que el puntero low sea menor al high, continuemos intentando
    while lo < hi:  # O(n) en el peor de los casos se recorre todo la lista
        # Si la suma de los dos punteros cumple el objetivo, devolvemos True
        if A[lo] + A[hi] == n:
            return True
        # Si no, si no alcanza con la suma, adelantemos low
        elif A[lo] + A[hi] < n:
            lo += 1
        # Si ni la suma cumple, y hi nos sobrepasa (la lista está ordenada), entonces
bajemos high
        else:
            hi -= 1

    # Si no hemos hallado habiendo recorrido toda la lista (lo >= hi), no existe tal caso
    return False
```