Problema 1

a)

- Bucle 1 : $i = n/2$; $\boxed{i \leq n}$

  ⇒ Iteraciones: $n - n/2 + 1 = n/2 + 1$ → constantes y coef. se simplifican

  ⇒ $O(n)$

- Bucle 2: $J = 1$; $\boxed{J + n/2 \leq n}$

  ⇒ $J \leq n/2$

  ⇒ Iteraciones: $n/2$

  ⇒ $O(n)$

- Bucle 3: $K = 1$; $\boxed{K \leq n}$; $K = K * 2$

  Iteración 1: $K = 1 = 2^0$

  Iteración 2: $K = 2 = 2^1$

  Iteración 3: $K = 4 = 2^2$

  ⋮

  Iteración m: $K = 2^{m-1}$

  ⇒ $2^{m-1} \leq n$

  $m \leq \log_2 n + 1$

  ⇒ Iteraciones: $\log_2 n$

  ⇒ $O(\log n)$

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

$= O(n^2 \log n)$

Escaneado con CamScanner

Problema 2

a)

- Condición: if $n \leq 1$ return
  - → Nos interesa el comportamiento cuando n crece
  - ⇒ no afecta
- Bucle 1: $i = 1$; $i \leq n$
  - → Iteraciones: n
  - → $O(n)$
- Bucle 2: $j = 1$; $j \leq n$
  - → Hay un print y seguido un break
  - ⇒ Se detiene después de la primera iteración
  - ⇒ No importa el valor de n, solo hay 1 iteración
  - → $O(1)$

Complejidad total: $O(n) \cdot O(1) = O(n)$

Escaneado con CamScanner

Problema 3

- Bucle 1: $i = 1$; $\boxed{i \leq n/3}$

  ⇒ Iteraciones: $n/3$

  ⇒ $O(n)$

- Bucle 2: $j = 1$; $\boxed{j \leq n}$; $j += 4$

  ⇒ $J = 1, 5, 9, 13, \ldots$

  $J_k = 1 + 4(k-1)$

  ⇒ $1 + 4(k-1) \leq n$

  $k \leq (n-1)/4 + 1$

  $k \leq (n+3)/n$

  ⇒ Iteraciones: $(n+3)/4$

  ⇒ $O(n)$

Complejidad total: $O(n) \cdot O(n) = \underline{O(n^2)}$

Escaneado con CamScanner

Problema 4

- Linear Search

```
def linear_search(arr,x):
    n = len(arr)
    for i in range(0,n):
        if (arr[i] == x):
            return i
    return -1
```

- Mejor Caso:

Elemento está en la primera posición del array

⇒ Solo se hace 1 comparación

⇒ No depende de n

→ $O(1)$

- Peor Caso

Elemento está en la última posición o no está

⇒ Hace n comparaciones

→ Recorre todo el array

→ $O(n)$

- Caso Promedio

Elemento puede estar en cualquier posición con igual prob.

→ Se hacen aprox. $n/2$ comparaciones

Escaneado con CamScanner

→ O(n)

- Binary search

```
def binary_search(arr, x):
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = low + (high - low) // 2
        if arr[mid] == x:
            return mid
        elif arr[mid] < x:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

- Mejor caso:

  Elemento está exactamente en el medio

  → 1 comparación

  → O(1)

- Peor caso:

  Elemento está en una de las extremos o no está

  ⇒ Con cada iteración el espacio de búsqueda se reduce a la mitad

  ⇒ El número máx. de iteraciones es el número de veces que n puede dividirse en 2

Escaneado con CamScanner

→ Iteraciones: $n/x^2 = 1$

$x = \log_2 n$

→ $O(\log n)$

- Caso Promedio:

Elemento está en cualquier posición con igual prob.

→ Se hacen aprox. $\log_2 n - 1$ comparaciones

→ $O(\log n)$

- Quick Sort

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)
```

→ Creo que se ve mejor con otra versión

- Mejor Caso y Caso Promedio:

Proceso de partición divide el array uniformemente

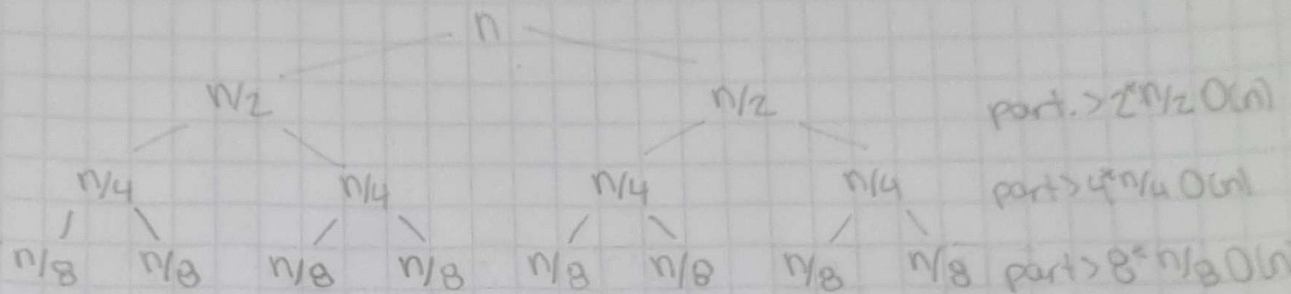
⇒ Altura del árbol: $n/2^h = 1 \Rightarrow h = \log_2 n$

⇒ Trabajo total: $n \cdot \log_2 n$

→ $O(n \log n)$

Escaneado con CamScanner

Ilustración mejor caso:



- Peor caso

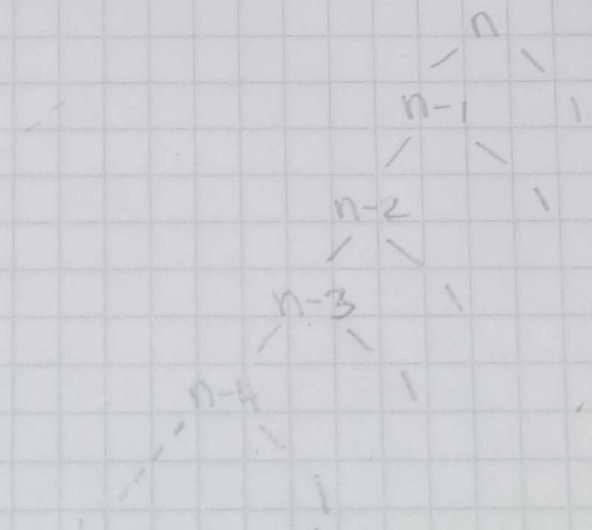
Partición está muy desbalanceada

⇒ Se hacen n llamadas recursivas

⇒ Cada llamada recursiva está hecha de un sub-array de tamaño n-1

⇒ $O(n^2)$

Ilustración peor caso



Escaneado con CamScanner

UVG | UNIVERSIDAD DEL VALLE DE GUATEMALA

Problema 5

a) Verdadera.

Sabemos que, $\exists c_1, c_2 \ni c_1(g(n)) \leq f(n)$ y $f(n) \leq c_2 g(n)$. Además, $\exists c_3, c_4 \ni c_3 h(n) \leq g(n) \leq c_4 h(n)$.

$\Rightarrow c_1 c_3 h(n) \leq f(n) \leq c_2 c_4 h(n)$

$\Rightarrow f(n) = \Theta(h(n))$, y por simetría $h(n) = \Theta(f(n))$. ☑

b) Verdadero

Sabemos que, $\exists c_1 \ni f(n) \leq c_1 g(n)$, y $\exists c_2 \ni g(n) \leq c_2 h(n)$

$\Rightarrow f(n) \leq c_1 c_2 h(n) = ch(n)$, $c = c_1 c_2$

$\Rightarrow \frac{1}{c} f(n) \leq h(n)$

$\therefore h(n) = \Omega(f(n))$. ☑

Escaneado con CamScanner

c) Falso.

En el código se tiene una tupla de $O(n)$ con un doble buncle $\Rightarrow O(n^2)$. Sin embargo al final se crea una subtupla, entonces,

$$\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} O(j-1) = O\left(\sum_{i=0}^{n-1} \frac{(n-i-1)(n-i)}{2}\right)$$

$$= O(n^3/6)$$

$\therefore O(n^3)$.

Descubre el Universo UVG

Escaneado con CamScanner