Taller Teórico 1
Presentado por: John Alejandro Pastor Sandoval

Haga una copia de este documento, debe nombrarlo como su ldap (usuario institucional) y subirlo en Entrega.Fecha de entrega: máximo el 21 jul 2025

Punto 1: Complejidades asintóticas y recursiones

(a) [0.5] Organice las siguientes funciones por el orden ascendente de crecimiento asintótico. Es decir encuentre un orden $g1, g2, g3, g4, g5, g6, g7, g8$ de las funciones tal que cada función tenga un crecimiento asintótico (big O) mayor o igual que las anteriores.

$$f_1(n) = n^{\pi} \qquad f_2(n) = \pi^n \qquad f_3(n) = \binom{n}{5} \qquad f_4(n) = \sqrt{2^{\sqrt{n}}}$$

$$f_5(n) = \overline{\binom{n}{n-4}} \qquad f_6(n) = 2^{\log^{\pi} n} \qquad f_7(n) = n^{5(\log n)^{\bar{2}}} \qquad f_8(n) = n^4 \binom{n}{4}$$

R/ Crecimiento asintótico de cada función:

1. $f_1(n) = n^{\pi} \Rightarrow f_1(n) = O(n^{\pi})$
2. $f_2(n) = \pi^n \Rightarrow f_2(n) = O(\pi^n)$
3. $f_3(n) = (n\ C\ 5) \Rightarrow f_3(n) = O(n^5)$
4. $f_4(n) = \sqrt{2^{\sqrt{n}}} = 2^{\frac{1}{2}\sqrt{n}} \Rightarrow f_4(n) = exp(\Theta(\sqrt{n}))$
5. $f_5(n) = (n\ C\ n-4) \Rightarrow = (n\ C\ 4) = O(n^4) \Rightarrow f_5(n) = O(n^4)$
6. $f_6(n) = 2^{\log_4 n} = 2^{\frac{\log_2 n}{2}} = n^{\frac{1}{2}} \Rightarrow f_6(n) = O(n^{\frac{1}{2}})$
7. $f_7(n) = n^{5(\log n)^2} \Rightarrow f_7(n) = O(n^{5(\log n)^2})$
8. $f_8(n) = n^4\ (n\ C\ 4);\ (n\ C\ 4) = O(n^4);\ f_8(n) = n^4 \cdot n^4 = n^8 \Rightarrow f_6(n) = O(n^8)$

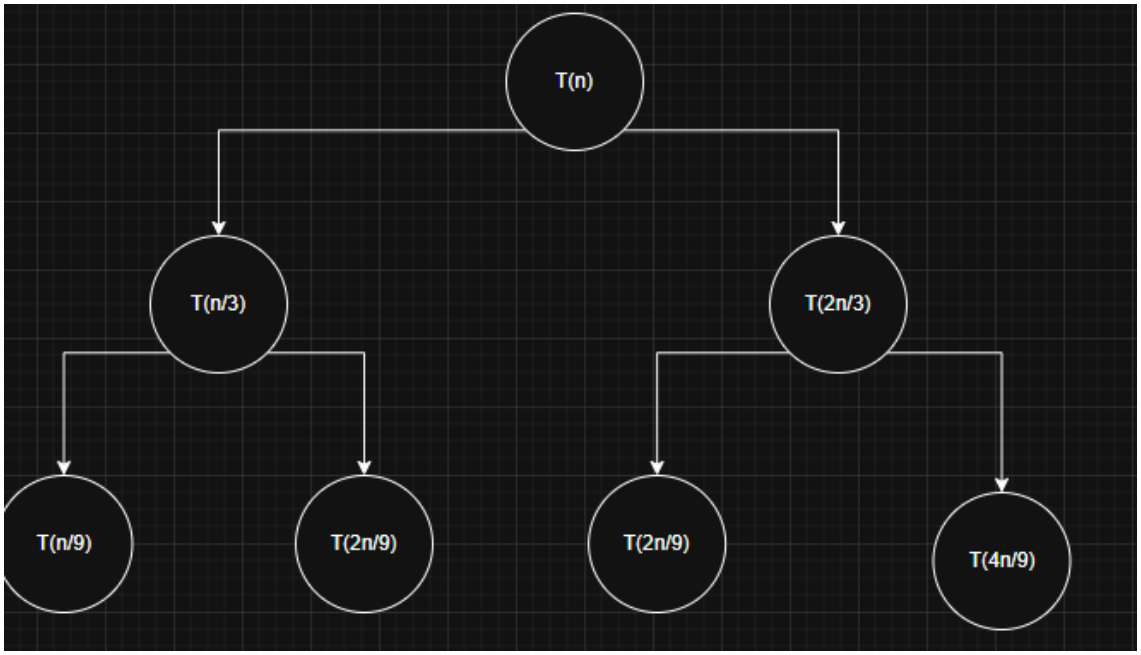
El orden ascendente de crecimiento asintótico es:

- $f_4 \Rightarrow f_6 \Rightarrow f_1 \Rightarrow f_5 \Rightarrow f_3 \Rightarrow f_7 \Rightarrow f_8 \Rightarrow f_2$

(b) [0.5] Encuentre una solución de la recurrencia : $T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + \Theta(n).$

Ayuda: Dibuje el árbol de recursión (la complejidad sería O(n) por nivel). Calcule cuántos niveles hay en total. Aproxime T(n) y use sustitución para resolver.

R/



Cada nodo representa el costo de procesar el problema en cierto tamaño. El análisis por niveles es:

- Nivel 1: nodo con costo $\Theta(n)$
- Nivel 2: el costo es $\Theta(\frac{n}{3})\Theta(\frac{2n}{3}) = \Theta(n)$
- Nivel 3: el costo es $\Theta(\frac{n}{9}) + \Theta(\frac{2n}{9}) + \Theta(\frac{2n}{9}) + \Theta(\frac{4n}{9}) = \Theta(n)$
- Así sigue sucesivamente.

Por lo tanto el costo es de $\Theta(n)$. El número de niveles L cumple:

$$(\frac{2}{3})^L n = \Theta(1) \Rightarrow L = log_{3/2} n = \Theta(log\ n)$$

Tenemos que hay $\Theta(log\ n)$ niveles y cada nivel tiene un costo de $\Theta(n)$, entonces el costo total es:

$$T(n) = \Theta(n\ log\ n)$$

Por sustitución, suponemos que $T(n) \leq cn\ log(n)$ para alguna constante $c > 0$ obtenemos entonces:

$$T(n) \leq c(\frac{n}{3})log(\frac{n}{3}) + c(\frac{2n}{3})log(\frac{2n}{3}) + kn$$

$$T(n) \leq cn[\frac{1}{3}(log\ n\ -\ log\ 3) + \frac{2}{3}(log\ n - log\ (\frac{3}{2}))] + kn$$

$$T(n) \leq cn\ log\ n - cn[\frac{1}{3} log\ 3 + \frac{2}{3} log\ (\frac{3}{2})] + kn$$

El término negativo es constante por n así que

$$c_1 n\ representa\ a: c_1 n[\frac{1}{3} log\ 3 + \frac{2}{3} log\ (\frac{3}{2})]\ y\ así\ para\ todo\ c_i$$

$$T(n) \leq cn\ log\ n - c_1 n + kn \leq cn\ log\ n$$

solo si c es lo suficientemente grande entonces confirmamos que :

$$T(n) = \Theta(n\ log\ n)$$

Punto 2. Verdadero o falso:

(a) [0.1] En el árbol de ejecución del mergesort, aproximadamente la misma cantidad de trabajo se realiza en cada nivel del árbol.

R/ **Verdadero**. Cada nivel del árbol de recursión de mergesort hace un trabajo de $\Theta(n)$, porque el total de comparaciones y fusiones por nivel sigue siendo proporcional a n.

(b) [0.1] Counting sort es un algoritmo de ordenamiento estable e "in-place".

R/ **Falso**. Usa espacio extra para ordenar los elementos del arreglo.

(c) [0.1] En un min-heap, el siguiente elemento más grande de cualquier otro elemento puede ser encontrado en tiempo O(log n).

R/ **Falso**. Solo el mínimo es accesible en $\Theta(1)$; encontrar el "siguiente mayor" puede llegar a $\Theta(n)$.

(d) [0.1] Binary insertion sorting (insertion sort que utiliza binary search para encontrar el siguiente elemento a insertar) requiere O(n log n) número total de operaciones.

R/ **Falso**. La búsqueda es $\Theta(log\ n)$ pero desplazar elementos sigue costando $\Theta(n)$, dando en total $\Theta(n^2)$.

(e) [0.1] Un algoritmo de tiempo polinomial generalmente se prefiere sobre un algoritmo de tiempo exponencial.

R/ **Verdadero**.Un algoritmo polinomial (n^k) se prefiere generalmente sobre uno exponencial (2^n).

(f) [0.1] Radix sort funciona correctamente cuando se usa cualquier algoritmo de ordenamiento para organizar cada dígito.

R/ **Falso**. Radix sort requiere un algoritmo estable para cada dígito; uno inestable desbarata el orden de los dígitos ya procesados.

(g) [0.2] Dado un arreglo A[1…n] de enteros, el tiempo de ejecución de Counting Sort es polinomial respecto al tamaño de entrada n.

R/ **Falso**.Counting sort corre en tiempo $\Theta(n + k)$, que es polinomial en el tamaño de la entrada (si k es acotado por un polinomio de n).

(h) [0.2] Dado un arreglo A[1…n] de enteros, el tiempo de ejecución de Heapsort es polinomial respecto al tamaño de entrada n.

R/ **Verdadero**. Heapsort es $\Theta(n\ log\ n)$, claramente polinomial en n.

Punto 3 Escenarios de ordenamiento.

Marque una x junto al que sería el mejor (es decir, el más eficiente) algoritmo de ordenamiento para cada escenario con el fin de reducir el tiempo de ejecución esperado.

(a) [0.4] Está manejando un catálogo de biblioteca. Sabe que los libros de su colección están casi en orden ascendente por título, con la excepción de un libro que está en el lugar equivocado. Quiere que el catálogo esté completamente ordenado en orden ascendente.

1. (X)Insertion Sort
2. ( )Merge Sort
3. ( )Radix Sort
4. ( )Heap Sort
5. ( )Counting Sort

Insertion sort es la mejor opción cuando tenemos una lista casi ordenada y necesitamos eficiencia en lugares concretos de desorden.

(b) [0.3] Estás trabajando en un dispositivo embebido (un cajero automático) que solo tiene 4 KB (4.096 bytes) de memoria libre, y desea ordenar las 2.000.000 de transacciones por el monto de dinero retirado (descartando el orden original de las transacciones).

1. ( )Insertion Sort
2. ( )Merge Sort
3. ( )Radix Sort
4. (X)Heap Sort
5. ( )Counting Sort

Heap Sort es la opción más realista entre las disponibles si se debe ordenar una gran cantidad de elementos en una cantidad poca de memoria.

(c) [0.3] Para determinar cuáles de sus amigos de una red social fueron los primeros en adoptar, quiere ordenarlos por sus identificadores de cuenta, que son enteros de 64 bits. (Recuerda que usted es muy popular, por lo que tiene muchos amigos en la red social).

1. ( )Insertion Sort
2. ( )Merge Sort
3. (X)Radix Sort
4. ( )Heap Sort
5. ( )Counting Sort

Usar Radix Sort es buena idea cuando toca ordenar grandes volúmenes de enteros de un tamaño fijo, como identificadores de cuentas de 64 bits.

Punto 4 Hotel

Has decidido irte al Caribe y comenzar una nueva vida. Sin embargo, las cosas no van muy bien, así que estás haciendo una consultoría para un hotel. Este hotel tiene N habitaciones de una cama, y los huéspedes entran y salen durante todo el día. Cuando un huésped se registra, pregunta por una habitación cuyo número está en el rango [l,h].

Quiere implementar una estructura de datos que soporte las siguientes operaciones de datos de la manera más eficiente posible.

1. INIT(N): Inicializar la estructura de datos para N habitaciones vacías numeradas 1,2,…,N, en tiempo polinomial.
2. COUNT(l,h): Devolver el número de habitaciones disponibles en [l,h], en tiempo O(logN).

3. CHECKIN(l,h): En tiempo O(logN), devolver la primera habitación vacía en [l,h] y marcarla como ocupada, o devolver NULL si todas las habitaciones en [l,h] están ocupadas.
4. CHECKOUT(x): Marcar la habitación x como no ocupada, en tiempo O(logN).

Se espera que para las respuestas describa las **ideas y pseudocódigo**. No se espera código de ningún lenguaje o pseudocódigo sin explicación de la idea

(a) [0.2] Describa la estructura de datos que utilizarás y cualquier invariante que sus algoritmos necesitan mantener. No proporciona algoritmos para las operaciones de su estructura de datos aquí; escríbalos en las partes (b)-(e) a continuación.

R/ Para este problema usaremos la estructura de datos árbol segmentado, principalmente escogí esta opción ya que investigando y leyendo el ejercicio la inicialización de las habitaciones será O(N) porque toca recorrer cada nodo,,la funciones COUNT y CHECKIN tienen que ser tiempo en O(Log N) normalmente los árboles funcionan en este tiempo, pero un árbol segmentado nos permite cumplir con estas funciones mejor ya que nos pasan unos valores que definen el rango de habitaciones a trabajar ([l,h]) al igual el checkout será una búsqueda recorriendo el árbol O(LogN). En esta estructura cada nodo almacena el número habitaciones en el rango de [l,r], el número de habitaciones que estén libres en el mismo rango y la primera habitación que se encuentre libre en ese rango.

Las invariantes para los nodos hoja (L = R):

- libre= 1 si la habitación está libre, 0 si está ocupada
- primera habitación libre = si la primera habitación del rango está libre, NULL si está ocupada

Para los nodos hijo seria:

- libre= left.free + right.free
- primera habitacion libre = min(izq.primera habitación libre , der.primera habitacion libre)

(b) [0.2] Proporcione un algoritmo que implemente INIT(N). El tiempo de ejecución debe ser polinomial en N.

R/

```
INIT(N):
  root = build_tree(1, N)
  return root

build_tree(l, h):
  nodo = new Node
  nodo.l = l
  nodo.h = h
```

```
  if l == h:
    nodo.total_disponibles = 1
    nodo.min_disponible = l
    nodo.izq  = NULL
    nodo.der = NULL
    return nodo

  mid = (l+h)/2
  nodo.izq  = build_tree(l,mid)
  nodo.der = build_tree(mid + 1, h)

  nodo.total_disponibles =
      nodo.izq.total_disponibles + nodo.der.total_disponibles

 if nodo.izq.min_disponible ≠ −l:
    nodo.min_disponible = nodo.izq.min_disponible
  else:
    nodo.min_disponible = nodo.der.min_disponible

  return nodo
```

(c) [0.2] Proporcione un algoritmo que implemente COUNT(l,h) en tiempo O(logN).

R/

```
COUNT(l, h):
    return count_range(root, l, h)

count_range(nodo, l, h):
    if nodo == NULL
       or h < nodo.l
       or l > nodo.h:
        return 0

  if l ≤ nodo.l
    and nodo.h ≤ h:
        return nodo.total_disponibles

    return count_range(nodo.izq,  l, h)+ count_range(nodo.right, l, h)
```

(d) [0.2] Proporcione un algoritmo que implemente CHECKIN(l,h) en tiempo O(logN).

R/

```
CHECKIN(l, h):
```

```
    habitación = primera_disponible(raíz, l, h)
  if habitación ≠ NULL:
        marcar_ocupada(raíz, habitación)
    return habitación


primera_disponible(nodo, l, h):
    if nodo == NULL or h < nodo.l or l > nodo.h:
        return NULL

  if l ≤ nodo.l and nodo.h ≤ h:
        return nodo.min_disponible

    resultado_izq = primera_disponible(nodo.izq, l, h)
  if resultado_izq ≠ NULL:
        return resultado_izq

    return first_available(nodo.der, l, h)

marcar_ocupada(nodo, hab):
    if nodo.l == nodo.h:
        nodo.total_disponibles = 0
        nodo.min_disponible    = -1
        return

  mid = ⌊(nodo.l + nodo.h) / 2⌋
  if hab ≤ mid:
        marcar_ocupada(nodo.izq, hab)
    else:
        marcar_ocupada(nodo.der, hab)

    nodo.total_disponibles = nodo.izq.total_disponibles
                           + nodo.der.total_disponibles

  if nodo.izq.min_disponible ≠ -1:
        nodo.min_disponible = nodo.izq.min_disponible
    else:
        nodo.min_disponible = nodo.der.min_disponible
```

(e) [0.2] Proporcione un algoritmo que implemente CHECKOUT(l,h) en tiempo O(logN).

R/

```
CHECKOUT(x):
    marcar_libre(raíz, x)

marcar_libre(nodo, hab):
    if nodo.l == nodo.h:
        nodo.total_disponibles = 1
        nodo.min_disponible    = hab
        return

  mid = ⌞(nodo.l + nodo.h) / 2⌟
  if hab ≤ mid:
        marcar_libre(nodo.l, hab)
    else:
        marcar_libre(nodo.r, hab)

    nodo.total_disponibles = nodo.izq.total_disponibles \
                           + nodo.der.total_disponibles

  if nodo.izq.min_disponible ≠ -1:
        nodo.min_disponible = nodo.izq.min_disponible
    else:
        nodo.min_disponible = nodo.der.min_disponible
```

Problema 5. ¿Zombis

En un intento por tomar la Tierra, los alienígenas del mal han contaminado cierto suministro de agua con un virus que transforma a los humanos en zombis que comen carne. Para rastrear a los alienígenas, la Unidad Nacional de Alerta y Laboratorios (UNAL) necesita determinar los epicentros del brote (cuáles suministros de agua han sido contaminados). Hay N ciudades potencialmente infectadas C={c1,c2,…,cN}, pero el gobierno tiene la certeza de que solo k ciudades tienen suministros de agua contaminados.

Desafortunadamente, la única prueba conocida para determinar la contaminación del suministro de agua de una ciudad es que un humano la beba y vea si se convierte en Zombie. Varios han ofrecido someterse a tal experimento, pero solo pueden probar suerte una vez. Cada voluntario está dispuesto a beber un solo vaso de agua que mezcla muestras de agua de cualquier subconjunto $C' \subseteq C$ de las n ciudades, que revela si al menos una ciudad en C' ha contaminado el agua.

Su objetivo es usar la menor cantidad posible de experimentos (voluntarios) para determinar, si para cada ciudad Ci su agua estaba contaminada, bajo el supuesto de que exactamente k ciudades tienen agua contaminada. Puede diseñar cada experimento basado en los resultados de los experimentos anteriores.

(a) [0.5] Observe que, como en un modelo de comparación, cualquier algoritmo se puede ver como un árbol de decisión donde un nodo corresponde a un experimento con dos resultados (contaminado o no) y dos hijos. Calcule un límite inferior de en el número de experimentos que deben realizarse para salvar el mundo**.**

R/ Para calcular el límite inferior de el número de experimentos que deben realizarse para salvar el mundo, primeramente el árbol de decisión que vamos a crear va a necesitar al menos $N\ C\ k$ hojas para una altura H definimos entonces:

$$2^H \geq (N\ C\ k) \Rightarrow T \geq\ log_2(N\ C\ k)$$

Si definimos la cota estándar como $(N\ C\ k) \geq \left(\frac{N}{K}\right)^k$ obtenemos entonces que:

$$T \geq\ k\ log_2 \frac{N}{k}$$

Es el valor mínimo de experimentos a realizar.

(b) [0.5] Salve el mundo diseñando un algoritmo que determine cuáles de las N ciudades tienen agua contaminada usando O(k log N) experimentos. Describa y analice su algoritmo.

R/ Este algoritmo realiza búsquedas binarias que máximo serán $log_2 ciudades \leq log_2 N$ experimentos esta búsqueda se realizará k veces (el número de ciudades que se supone tienen agua contaminada) de esta forma aseguramos que el costo máximo será de:

$\Theta(k\ log\ N)$ que se acerca mucho al límite inferior $k\ log_2 \frac{N}{k}$ definido anteriormente.

```
encontrar_contaminadas(N, k):
    ciudades = [1 a N]
    contaminadas = []

    para i de 1 a k:
        inicio = 0
        fin = tamaño(ciudades) - 1

        mientras inicio < fin:
     mid ← ⌊(inicio + fin) / 2⌋
            A   ← ciudades[inicio a mitad]
            si prueba(A) = TRUE:
                fin = mid
            else:
                inicio = mid + 1

        x = ciudades[inicio]
        contaminadas.agregar(x)
```

```
        eliminar ciudad x de ciudades
    devolver contaminadas
```

Problema 6. Calificaciones

Valide sus respuestas, basado en la correctitud, su aprendizaje y el valor de cada parte, calcule su calificación de cada punto (en el rango decimal [0,1]) y sumarlas para calcular la calificación total del taller. Si el profesor no está de acuerdo asignará una calificación de ‘Coevaluación’ y la calificación total del punto será el promedio

	Autoevaluación	**Coevaluación**	**Total**
Punto 1	**1**		
Punto 2	**1**		
Punto 3	**1**		
Punto 4	**0.7**		
Punto 5	**0.8**		
Calificación Total	**4.5**		