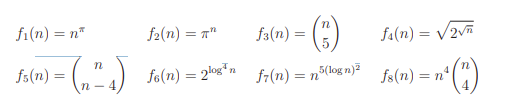
**Taller Teórico 1**

**Presentado por:** Jaime Darley Angulo Tenorio

**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 de las funciones tal que cada funcion tenga un crecimiento asintótico ( big O ) mayor o igual que las anteriores.



R/

**g1 = f₄(n) = √(2√n):** crece muy lento (raíz doble).

**g2 = f₅(n) = n / (n−4):** se comporta como constante.

**g3 = f₃(n) = C(n,5):** es un polinomio de grado 5.

**g4 = f₆(n) = 2^(log⁴n):** sube más que cualquier polinomio fijo.

**g5 = f₂(n) = πⁿ:** exponencial base π.

**g6 = f₈(n) = n⁴·C(n,4):** es aproximadamente n⁸ (polinomio).

**g7 = f₇(n) = n^{5·(log n)²}:** más que polinomial, pero menos que nⁿ.

**g8 = f₁(n) = nⁿ:** crece más rápido que todas.

**(b) [0.5]** Encuentre una solución de la recurrencia : 

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/

**Recurrencia:** T(n) = T(n/3) + T(2n/3) + Θ(n)

* En cada nivel del árbol de recursión se hacen dos llamadas (una con n/3 y otra con 2n/3).
* El trabajo por nivel es Θ(n), porque la suma de los tamaños sigue siendo proporcional a n.
* La rama más larga es la de T(2n/3), así que la altura del árbol es ≈ log₍₃⁄₂₎(n) = O(log n).
* Como hay O(log n) niveles y cada uno cuesta O(n), el costo total es:

**Resultado final:** T(n) = Θ(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.**

En el árbol de mergesort, cada nivel combina n elementos en total, por lo tanto el trabajo por nivel es Θ(n).

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

R/ **Falso.**

Counting Sort **es estable**, pero **no es in-place** porque usa estructuras auxiliares (arreglos extras).

**(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.**

En un min-heap solo puedes acceder al **mínimo** en O(1). Para encontrar el siguiente mayor necesitas recorrer el heap → O(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/  **Verdadero.**

Binary Insertion Sort usa búsqueda binaria (O(log n)) para cada inserción, pero mover los elementos sigue siendo O(n), resultando en O(n log n) operaciones.

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

R/ **Verdadero.**

Los algoritmos polinomiales son más eficientes y generalmente preferibles a los exponenciales, que son mucho más costosos.

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

R/ **Falso.**

Radix Sort **solo funciona correctamente** si el algoritmo usado para ordenar los dígitos es **estable** (como Counting Sort), no cualquiera.

**(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/ **Verdadero.**

Counting Sort tiene tiempo O(n + k), que es polinomial si k (rango de valores) es razonable con respecto a 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 tiene un tiempo de ejecución de **O(n log n)** en el peor caso, lo cual es una complejidad polinomial respecto al tamaño de entrada nnn.

**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 muy eficiente para datos **casi ordenados**: su mejor caso es **O(n)**, mucho mejor que otros algoritmos en ese escenario.

**(b) [0.3]** Estás trabajando en un dispositivo embebido (un cajero automático) que solo tiene 4KB (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 **in-place** (no necesita memoria extra) y tiene buen rendimiento en general (**O(n log n)**). Otras como Merge Sort o Counting Sort necesitan mucha memoria adicional, lo cual no es viable aquí.

**(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

Los identificadores son enteros → ideal para **Radix Sort**.

Radix Sort tiene tiempo **O(n · k)**, donde *k* es el número de dígitos (en este caso, 64 bits)

A diferencia de Counting Sort, **no necesita tanta memoria** como para cubrir todos los posibles valores de 64 bits.

Además, **es más rápido que comparativos** como Merge o Heap cuando *k* es pequeño y los datos son enteros.

**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/

**Se usa un Segment Tree (árbol de segmentos)** donde cada nodo representa un rango de habitaciones.  
 Cada nodo guarda:

* **count**: número de habitaciones **libres** en ese intervalo.
* **min\_free**: índice de la **primera habitación libre** en ese intervalo (o NULL si no hay).

**Invariante:**

* El count de cada nodo = suma de los count de sus hijos.
* El min\_free de cada nodo = el menor índice libre entre sus hijos, o NULL si ninguno está libre.

Esta estructura permite responder en **O(log N)** para todas las operaciones.

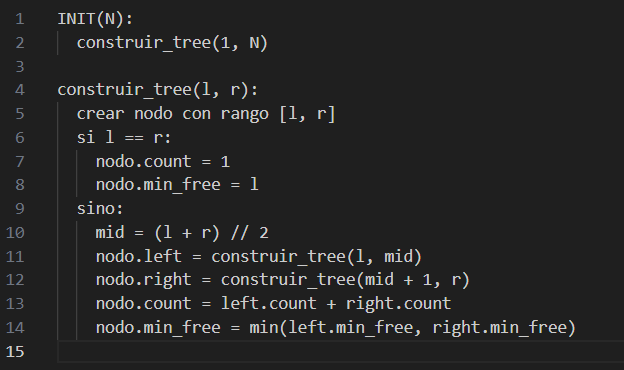
**(b) [0.2]** Proporcione un algoritmo que implemente INIT(N). El tiempo de ejecución debe ser polinomial en N.

R/

Construir el Segment Tree sobre el rango [1, N].  
 Inicialmente, todas las habitaciones están libres, así que:

* count = tamaño del nodo
* min\_free = índice más bajo del rango del nodo

Algoritmo en pseudocodigo



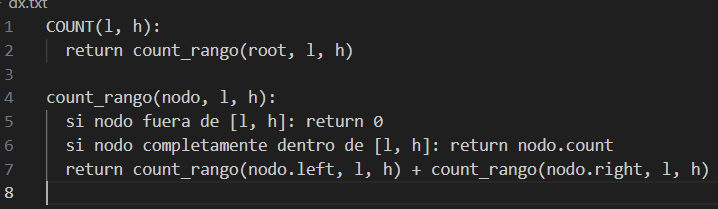
**(c) [0.2]** Proporcione un algoritmo que implemente COUNT(l,h) en tiempo O(logN).

R/

Recorrer el Segment Tree con búsqueda de rango.

Si el nodo está completamente dentro de [l, h], devolvemos su count.

**Algoritmo:**



**Tiempo: O(log N)**

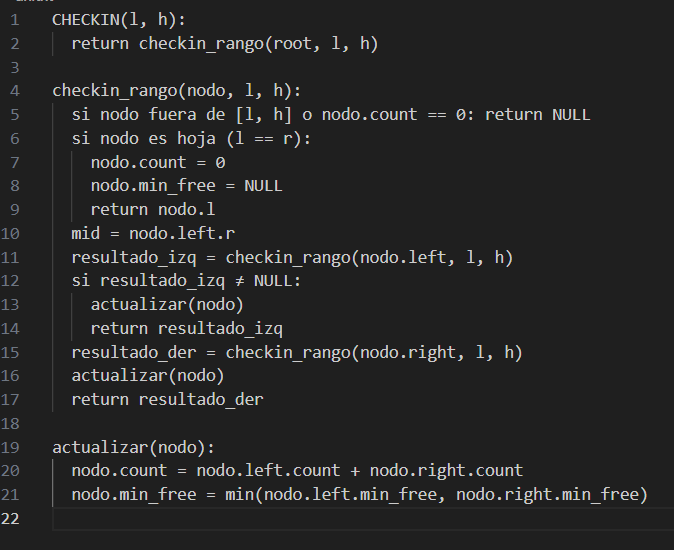
**(d) [0.2]** Proporcione un algoritmo que implemente CHECKIN(l,h) en tiempo O(logN).

R/

Recorremos el Segment Tree buscando la primera habitación libre en [l, h] usando min\_free.

Si encontramos una habitación libre, actualizamos los nodos para marcarla como ocupada.

**Algoritmo**

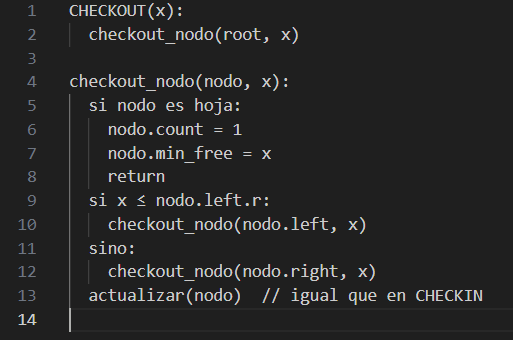
**Tiempo: O(log N)**

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

R/

Recorremos el árbol hasta el nodo hoja correspondiente a la habitación x y la marcamos como libre (count = 1, min\_free = x), y actualizamos los ancestros.

**Algoritmo**



**Tiempo: O(log N)**

**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′⊆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/**

Hay que distinguir entre todas las formas de escoger las ciudades contaminadas de un grupo de N.

El número de formas es “N choose k” (es decir, N combinatorio k).

Cada experimento aporta sólo 1 bit de información (sí o no), así que necesitas al menos

log₂( N choose k )experimentaciones. Usando que log₂( N choose k ) es de orden k·log N, obtenemos:

**Número mínimo de experimentos ≥ Ω(k · log N).**

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

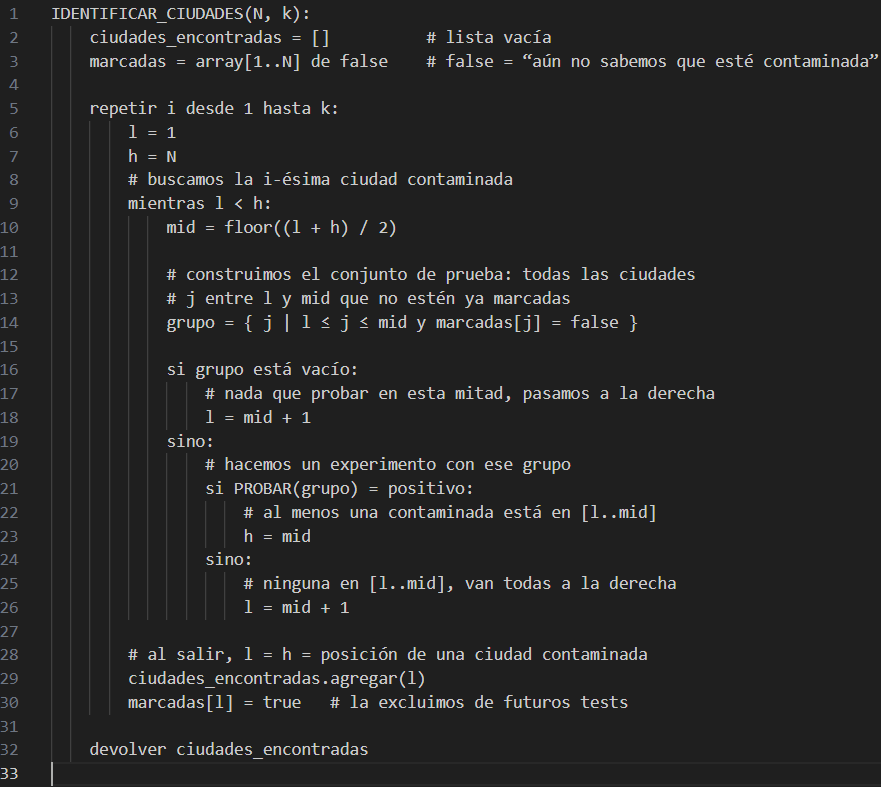
**R/ Algoritmo en O(k log N) experimentos**

La idea es localizar cada una de las k ciudades contaminadas por separado, usando búsqueda binaria sobre el rango 1…N y cada experimento prueba solo las ciudades que aún no hemos identificado como contaminadas.

**Por qué es O(k log N):**

* Para cada una de las ciudades contaminadas hacemos una búsqueda binaria en el rango 1…N, que usa a lo sumo ~log₂N experimentos.
* Marcar la ciudad hallada impide que vuelva a contarse en tests posteriores.
* Total de experimentos ≤ k·⌈log₂N⌉ = O(k log N).

Pseudocódigo



**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** | **4.5** |  |  |
| **Punto 2** | **4.7** |  |  |
| **Punto 3** | **4.4** |  |  |
| **Punto 4** | **4.6** |  |  |
| **Punto 5** | **5** |  |  |
| **Calificación Total** | **4.6** |  |  |