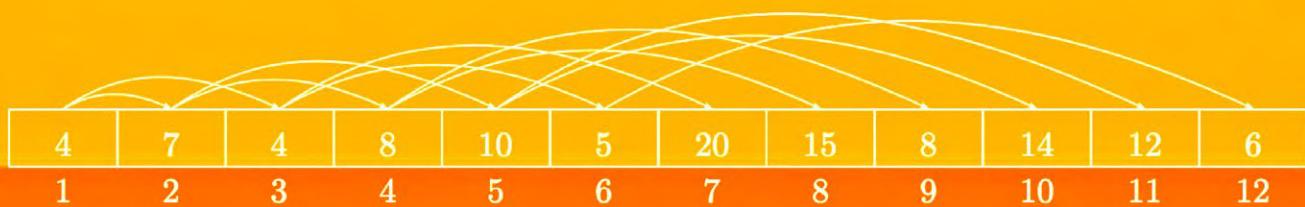
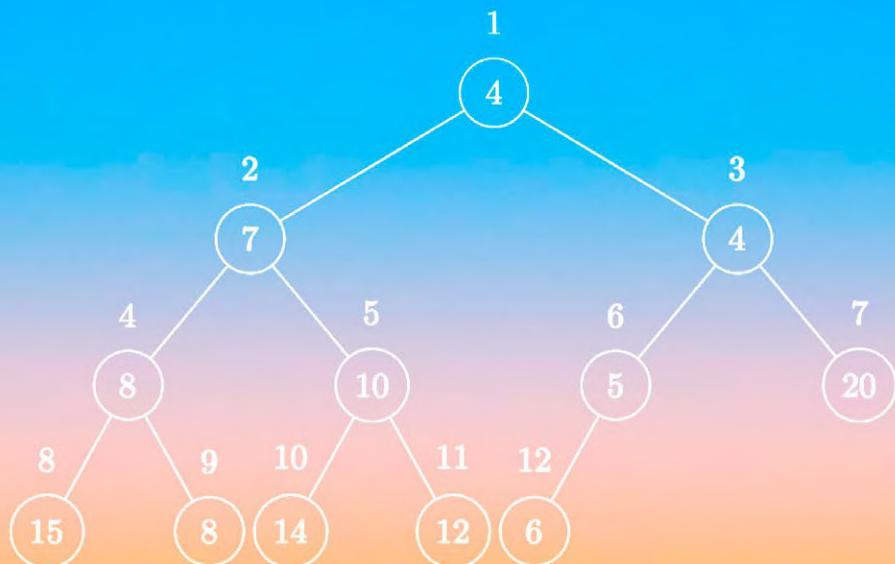


# ESTRUCTURAS DE DATOS

## y sus aplicaciones en Programación Competitiva

### TOMO I

Hugo Humberto Morales Peña



**Hugo Humberto Morales Peña** (Medellín, Antioquia, 1975)

Ingeniero de Sistemas por la Universidad del Valle (2002) y Magíster en Enseñanza de las Matemáticas, en la línea de Matemáticas Computacionales, por la Universidad Tecnológica de Pereira (2012). Se desempeña como Profesor Asociado en el programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira, donde desde 2011 es tutor del Semillero de Investigación In Silico, responsable del proceso de maratones de programación de la institución. Desde 2013 es Coordinador Académico y miembro fundador de la Red de Programación Competitiva (RPC). Ha recibido el Master Coaching Award – ICPC World Finals Astana 2024 por su labor en el entrenamiento de equipos clasificados a los mundiales de programación de 2013 (San Petersburgo, Rusia), 2014 (Ekaterimburgo, Rusia), 2018 (Pekín, China), 2021 (Daca, Bangladesh) y 2024 (Astana, Kazajistán).

[huhumor@utp.edu.co](mailto:huhumor@utp.edu.co)

# **ESTRUCTURAS DE DATOS**

y sus aplicaciones  
en Programación Competitiva

**TOMO I**

**Hugo Humberto Morales Peña**



**Facultad de Ingenierías  
Colección Textos Académicos  
2025**

Morales Peña, Hugo Humberto  
Estructuras de datos y sus aplicaciones en programación competitiva - Tomo I / Hugo Humberto Morales Peña. -- Pereira : Editorial Universidad Tecnológica de Pereira, 2025.  
288 páginas. (Colección Textos académicos).  
e-ISBN: 978-628-501-048-4  
1. Estructura de datos 2. Programación competitiva 3. Competencias de programación 4. Métodos de estudio  
CDD.005.73

## **ESTRUCTURAS DE DATOS y sus aplicaciones en Programación Competitiva -TOMO I**

© Hugo Humberto Morales Peña

© Universidad Tecnológica de Pereira

e-ISBN: 978-628-501-048-4

### **Universidad Tecnológica de Pereira**

Vicerrectoría de Investigaciones, Innovación y Extensión

Editorial Universidad Tecnológica de Pereira

Pereira, Colombia

### **Coordinador editorial:**

Luis Miguel Vargas Valencia

luismvargas@utp.edu.co

Teléfono (606) 313 7381

Edificio 9, Biblioteca Central Jorge Roa Martínez

Cra. 27 No. 10-02 Los Álamos, Pereira, Colombia

[www.utp.edu.co](http://www.utp.edu.co)

### **Diagramación y producción:**

Tomás Flórez Calle

Universidad Tecnológica de Pereira

Pereira, Risaralda, Colombia.

*Es mejor hecho que perfecto...  
para algo han de servir las futuras  
ediciones 2, 3 y 4 de este libro.*

*Dedico este libro a  
María Gladys, mi esposa,  
quien es mi compañera de vida,  
mi consejera, mi polo a tierra,  
y a mis hijas Angie Daniela  
y Yesica Alejandra,  
quienes han sido la fuente  
de inagotable energía para  
las largas jornadas  
dedicadas a este trabajo.*



# Prefacio

No todo en la pandemia del COVID-19 fue malo, también fue la oportunidad de reencontrarnos con nuestras propias familias, de rehacernos en nuestra vida laboral, de convertirnos en docentes virtuales de una semana a la otra, sin tener la más mínima experiencia en la era digital.

Este libro, que hoy comparto con todos ustedes, es hijo de los casi dos años de cuarentena generados por la pandemia. Durante cuatro semestres de clases virtuales, escribí y pulí mis apuntes (construidos inicialmente a mano alzada a lo largo de 10 o más años) del curso de Estructuras de Datos, con calidad de libro; calidad que se logra en la edición de documentos al trabajar con el lenguaje de programación L<sup>A</sup>T<sub>E</sub>X.

Ahora, una pregunta que debe rondar por la mente de los lectores de este libro es: ¿para qué un libro de *estructuras de datos en lenguaje C* en pleno año 2026? La razón es muy sencilla: no hay bibliotecas donde ya estén hechas las estructuras de datos que se van a trabajar en el libro, en este lenguaje de programación se tiene que construir desde cero cada una de ellas. En ese proceso se va de la mano de la complejidad computacional en tiempo de ejecución y en espacio de almacenamiento.

El objetivo de este libro no es capacitar usuarios finales en el uso de unas estructuras de datos, sino que el lector aprenda a construir las suyas sin perder de vista la eficiencia computacional.

Este libro no es simplemente un libro de estructuras de datos al estilo de los que se escribieron en la década de los 90 (por citar algunos, [AHU98], [Bec95], [JZ98], [LAT97] y [Vil96]); Tam poco es simplemente un libro con las temáticas del curso. Es un libro que trabaja en conjunto con el juez en línea *HackerRank*, donde los lectores pueden enviar a evaluar las soluciones que construyan para cada unos de sus retos de programación, obteniendo alguno de los veredictos: aceptado, respuesta incorrecta, tiempo límite excedido o error en tiempo de ejecución. De esta forma, el lector recibe retroalimentación (en unos pocos segundos de juzgamiento) sobre la correctitud y eficiencia computacional de sus soluciones.

Como recomendación para los lectores de este libro: una vez finalizado el curso, es el momento de comenzar a trabajar con **C++** y hacer uso de las bibliotecas en las que se encuentran implementadas las estructuras de datos; ahora si ya no será simplemente un usuario final, ahora si se entenderá el verdadero poder de lo que se a utilizar.

Hugo Humberto Morales Peña

Profesor Asociado

Programa de Ingeniería de Sistemas y Computación

Universidad Tecnológica de Pereira



# Prólogo

Cuando, como profesores, impartimos una clase, a menudo explicamos respuestas a preguntas que muchos estudiantes aún no se han formulado, o que surgen al enfrentarse a retos de programación donde esas respuestas son necesarias para construir una solución.

En particular, temas como el de *estructuras de datos* suelen ser cursos donde esta tendencia se acentúa. La densidad de la materia nos lleva a trabajar con casos pequeños para probar algoritmos, con el objetivo de que los estudiantes “entiendan” lo que queremos mostrarles.

Sin embargo, como dijo Einstein: “Hay que explicar las cosas tan sencillo como se pueda, pero no más sencillo de lo que son”. Este libro es, en mi opinión, un esfuerzo por presentar las primeras estructuras de datos tal como son, sin sobresimplificaciones. Cada capítulo está acompañado de una serie de ejercicios no triviales que exigen comprender los conceptos planteados. Además, estos ejercicios funcionan como una excelente introducción al fascinante mundo de la programación competitiva. En este ámbito, cada problema demanda un entendimiento profundo para su resolución, y los torneos (o maratones, como se conocen en Colombia) de programación evidencian las falencias que puede tener un competidor.

Asimismo, este libro busca transmitir la diversión de programar y aprender, a través de retos que invitan a pensar y a demostrar las habilidades y competencias esenciales que todo programador debe dominar.

Espero que el lector encuentre este libro tan agradable y ameno como lo fue para mí.

Eddy Ramírez Jiménez  
Profesor  
Escuela de Ingeniería en Computación  
Tecnológico de Costa Rica  
Regional Contest Director (RCD) Centroamérica  
San José, Costa Rica



# Agradecimientos

Un agradecimiento y reconocimiento especial para los escritores de retos de programación que han permitido que estos sean utilizados en este libro. Todos los retos que aparecen en este libro han sido utilizados en maratones de programación originales, que se han realizado en la **Red de Programación Competitiva (RPC)** en las que han participado competidores de toda América Latina.

Lista de autores de retos de programación de la comunidad académica del programa de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira (ya graduados):

- Jhon Jimenez (reto: 2.5.2 Tobby y los tanques II)
- Alejandro Moreno Agudelo (reto: 2.5.4 Teorema del número poligonal de Fermat)
- Yeferson Gaitán Gómez (retos: 2.5.6 Tobby y las galletas y 3.4.4 Duerme, Tobby)
- Sebastián Gómez González (reto: 3.4.5 Hecho en Colombia)
- Gabriel Gutiérrez Tamayo (retos: 1.5.9 Triángulos internos, 1.5.10 ¿Cuántos triángulos equiláteros?, 3.4.6 Juego de pares e impares, 4.9.3 Josefo y los números no-coprimos, 4.9.7 Domino a ciegas y encolado y 4.9.8 De nuevo ... sumar todos (versión 2025))

Lista de autores de retos de programación externos a la Universidad Tecnológica de Pereira:

- Milton Jesús Vera Contreras, profesor de la Universidad Francisco de Paula Santander, Cúcuta, Colombia (reto: 3.4.2 Imposible)
- Eddy Ramírez Jiménez, profesor del Tecnológico de Costa Rica - Campus Alajuela, Alajuela, Costa Rica (reto: 3.4.3 Asignación de citas para la vacuna del COVID-19)
- Yonny Mondelo Hernández, UnDosTres México, México (retos: 3.4.7 Máximo de eventos en una sala, 3.4.8 Máximo de eventos en múltiples salas y 4.7.5 Consultas sobre la pila)

Un agradecimiento y reconocimiento especial para Diego Alejandro Rangel Mazo, estudiante de Ingeniería de Sistemas y Computación de la Universidad Tecnológica de Pereira, quien ha sido mi monitor académico durante el último año de la escritura de este libro, él fue quien construyó en L<sup>A</sup>T<sub>E</sub>X más del 90 % de los gráficos que se encuentran en este documento.

Un agradecimiento y reconocimiento especial para los profesores: Juan Andrés García (UTP), Juan Manuel Velásquez (UTP) y Eddy Ramírez Jiménez (TEC - Alajuela) por las valiosas sugerencias, recomendaciones y/o correcciones que realizaron sobre las primeras versiones de este documento. Gracias a ellos este libro es aún mejor de lo que originalmente era.



# Contenido

<b>Capítulo 1</b>	
<b>Repaso de herramientas matemáticas</b>	<b>. . . . . 19</b>
1.1. Series, sumatorias e inducción matemática	. . . . . 21
1.1.1. Teorema (sumatoria de Gauss)	. . . . . 21
1.1.2. ¿Particularidad o generalidad?	. . . . . 25
1.1.3. Serie geométrica	. . . . . 29
1.1.4. Suma de términos de una serie geométrica	. . . . . 29
1.1.5. Serie geométrica que converge en el infinito	. . . . . 31
1.1.6. Fórmulas de sumatorias útiles	. . . . . 31
1.1.7. Preguntas tipo Saber Pro	. . . . . 31
1.2. Tallas de los tipos de datos enteros	. . . . . 36
1.2.1. Unsigned int	. . . . . 37
1.2.2. Signed int	. . . . . 37
1.2.3. Unsigned long long int	. . . . . 37
1.2.4. Signed long long int	. . . . . 37
1.3. Retos de programación resueltos	. . . . . 39
1.3.1. Números triangulares	. . . . . 39
1.3.2. La ladrona de libros	. . . . . 45
1.3.3. Felipe y la secuencia	. . . . . 48
1.3.4. La función de Dangie	. . . . . 51
1.4. Relaciones de recurrencia	. . . . . 55
1.4.1. Método de iteración	. . . . . 56
1.5. Retos de programación propuestos	. . . . . 63
1.5.1. La sucesión de Humberto Moralov	. . . . . 64
1.5.2. ¿Cuántos números triangulares?	. . . . . 65
1.5.3. Dora la exploradora I	. . . . . 67
1.5.4. Dora la exploradora II	. . . . . 69

1.5.5. Humbertov y el espiral triangular I . . . . .	71
1.5.6. Humbertov y el espiral triangular II . . . . .	73
1.5.7. Desviación estándar . . . . .	75
1.5.8. La función de Johann . . . . .	77
1.5.9. Triángulos internos . . . . .	79
1.5.10. ¿Cuántos triángulos equiláteros? . . . . .	81
<b>Capítulo 2</b>	
<b>Complejidad computacional . . . . .</b>	<b>83</b>
2.1. Notaciones computacionales ( $O$ , $\Omega$ , $\Theta$ ) . . . . .	85
2.1.1. Notación $O$ . . . . .	86
2.1.2. Notación $\Omega$ . . . . .	88
2.1.3. Notación $\Theta$ . . . . .	89
2.2. Algoritmo de búsqueda binaria (Binary Search) . . . . .	91
2.2.1. Complejidad en tiempo de ejecución del algoritmo Binary Search . . . . .	93
2.2.2. Algoritmo iterativo de búsqueda binaria . . . . .	99
2.2.3. Implementación del algoritmo de búsqueda binaria . . . . .	94
2.2.4. Solución alternativa del reto: La ladrona de libros . . . . .	96
2.2.5. Búsqueda binaria de un elemento que está múltiples veces en el arreglo . . . . .	98
2.2.6. Reto de programación: Subsecuencia atractiva . . . . .	100
2.3. Algoritmo de ordenamiento por mezclas (Merge Sort) . . . . .	105
2.3.1. Función Merge . . . . .	105
2.3.2. Función Merge Sort . . . . .	105
2.3.3. Complejidad en espacio de almacenamiento del algoritmo de ordenamiento por mezclas . . . . .	110
2.3.4. Complejidad en tiempo de ejecución del algoritmo de ordenamiento por mezclas . . . . .	110
2.3.5. Implementación del algoritmo de ordenamiento por mezclas . . . . .	112
2.3.6. Reto de programación: ¿Cuántas inversiones? . . . . .	113
2.4. Retos de programación resueltos . . . . .	118
2.4.1. ¿Cuántos subconjuntos? . . . . .	118
2.4.2. Colección dinámica . . . . .	125
2.5. Retos de programación propuestos . . . . .	131
2.5.1. Centro numérico . . . . .	132
2.5.2. Tobby y los tanques II . . . . .	133
2.5.3. Taza de café . . . . .	135
2.5.4. Teorema del número poligonal de Fermat . . . . .	137
2.5.5. Frecuencias de subconjuntos . . . . .	139
2.5.6. Tobby y las galletas . . . . .	141

2.5.7. Diccionario .....	143
2.5.8. Generar, ordenar y buscar .....	145

**Capítulo 3**  
**Colas de prioridad .....** **149**

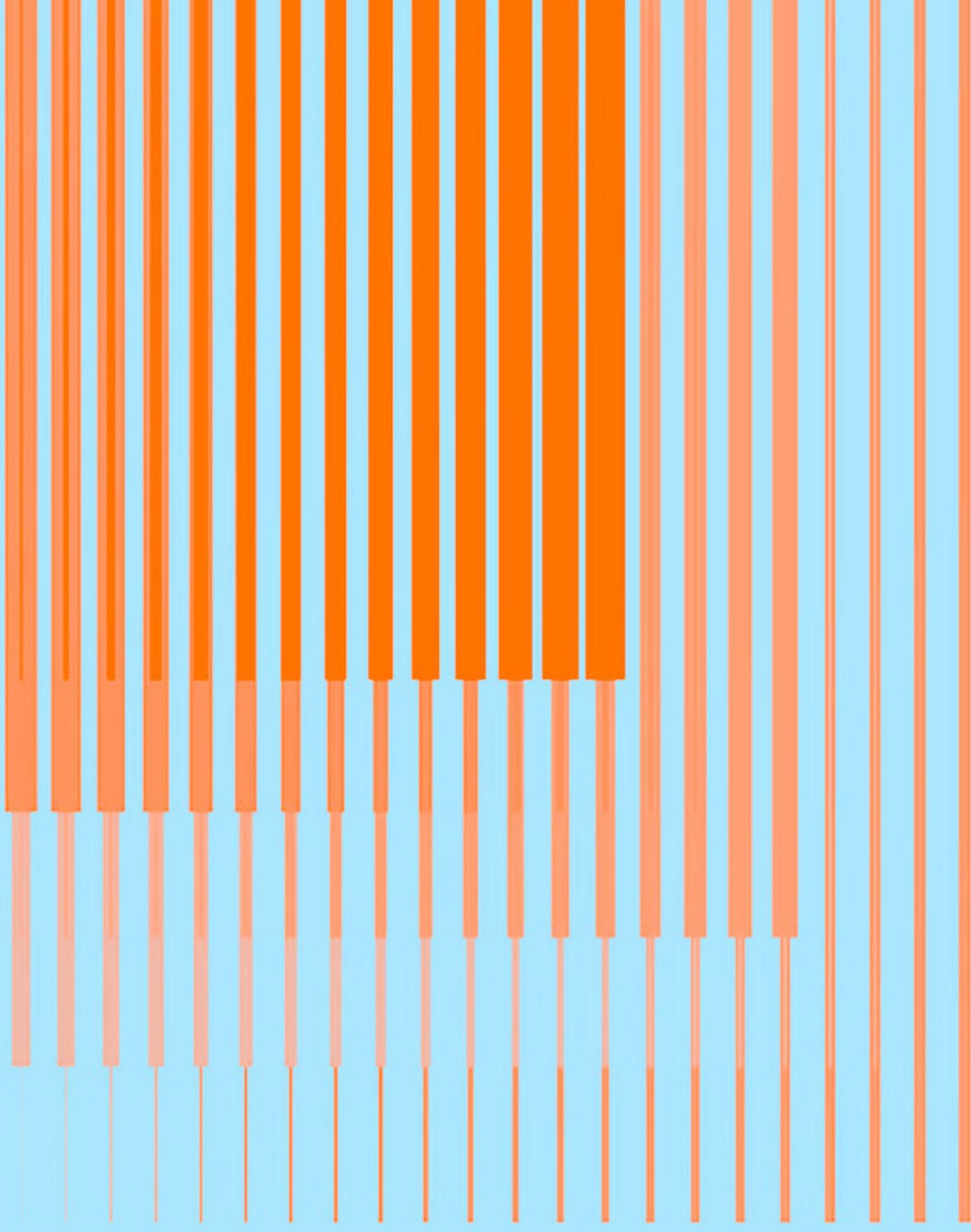
3.1. Estructura de datos montón .....	151
3.1.1. Las propiedades de forma y orden en un montón. ....	152
3.1.2. Función para garantizar la propiedad de orden de montón. ....	153
3.2. Colas de prioridad .....	156
3.2.1. Implementación de la cola de prioridad mínima. ....	157
3.3. Retos de programación resueltos .....	160
3.3.1. Sumar todos (versión 2025). ....	160
3.3.2. Cardinalidad de conjuntos .....	165
3.3.3. Sala de urgencias .....	171
3.4. Retos de programación propuestos .....	177
3.4.1. Guanex y las colas de prioridad .....	178
3.4.2. Imposible .....	180
3.4.3. Asignación de citas para la vacuna del COVID-19 .....	181
3.4.4. Duerme, Tobby. ....	183
3.4.5. Hecho en Colombia. ....	185
3.4.6. Juego de pares e impares .....	187
3.4.7. Máximo de eventos en una sala. ....	189
3.4.8. Máximo de eventos en múltiples salas .....	190

**Capítulo 4**  
**Listas, pilas y colas .....** **193**

4.1. Función malloc() .....	195
4.2. Función free().....	197
4.3. Concepto de lista .....	199
4.3.1. Reto de programación: Sumar todos (versión 2025 - talla pequeña) .....	207
4.4. Listas circulares simplemente enlazadas .....	213
4.4.1. Inserción de un elemento en una lista circular simplemente enlazada .....	213
4.4.2. Borrado del primer nodo en una lista circular simplemente enlazada .....	216
4.4.3. Impresión desde el primero hasta el último de los elementos de una lista circular simplemente enlazada .....	216
4.4.4. Programa completo para el mantenimiento de una lista circular simplemente enlazada .....	217
4.4.5. Reto de programación: Problema de Josefo .....	219
4.5. Listas circulares doblemente enlazadas .....	222

4.5.1. Inserción de un elemento en una lista circular doblemente enlazada .....	223
4.5.2. Borrado del primer nodo en una lista circular doblemente enlazada.....	227
4.5.3. Impresión desde el primero hasta el último de los elementos de una lista circular doblemente enlazada.....	229
4.5.4. Impresión desde el último hasta el primero de los elementos de una lista circular doblemente enlazada .....	229
4.5.5. Programa completo para el mantenimiento de una lista circular doblemente enlazada....	230
4.5.6. Borrar un elemento en una lista circular doblemente enlazada.....	232
4.5.7. Programa completo para el mantenimiento de una lista circular doblemente enlazada en la cual se inserta o se borra un elemento específico .....	233
4.5.8. Reto de programación: Rifa de Josefo .....	237
4.6. Pilas y colas .....	241
4.7. Pilas.....	241
4.7.1. Función Push.....	241
4.7.2. Función Pop .....	242
4.7.3. Función StackEmpty .....	242
4.7.4. Programa completo para el manejo de pilas .....	242
4.7.5. Reto de programación: Consultas sobre la pila.....	244
4.8. Colas.....	249
4.8.1. Función Enqueue .....	249
4.8.2. Función Dequeue .....	249
4.8.3. Función QueueEmpty .....	250
4.8.4. Programa completo para el manejo de colas .....	250
4.8.5. Reto de programación: Consultas sobre la cola .....	252
4.9. Retos de programación propuestos .....	257
4.9.1. Borrar, borrar, borrar .....	258
4.9.2. Consultas sobre la lista (talla pequeña) .....	260
4.9.3. Josefo y los números no-coprimos .....	262
4.9.4. Rifa de Josefo II .....	264
4.9.5. Juego de mesa .....	266
4.9.6. Selección de personal.....	268
4.9.7. Domino a ciegas y encolado .....	271
4.9.8. Sumar todos II (versión 2025).....	273
<b>A. Cómo evitar el error 404 de HackerRank .....</b>	<b>277</b>
<b>Bibliografía .....</b>	<b>287</b>





## **Capítulo 1**

# **Repaso de herramientas matemáticas**



En este capítulo, como su nombre lo indica, se hace un repaso de las herramientas matemáticas que se necesitan para poder calcular la complejidad computacional (costo computacional en tiempo de ejecución y en espacio de almacenamiento) de un algoritmo. Se abordan temas como sumatorias, series geométricas, suma de términos de una serie geométrica, series geométricas que convergen en el infinito, relaciones de recurrencia e inducción matemática. Además, a partir del resultado de la suma de términos de una serie geométrica, se calculan los máximos valores que se pueden almacenar en el computador para los diferentes tipos de datos enteros.

El capítulo continúa con la solución de cuatro retos de programación que se apoyan en conceptos matemáticos, donde se transita por soluciones ingenuas (muy costosas en tiempo de ejecución, pero que generan resultados correctos), y por soluciones eficientes (en tiempo de ejecución) que se generan a partir de la solución analítica (solución matemática) del problema.

Finalizando el capítulo, se plantean 10 retos de programación para que los lectores no solamente programen las soluciones sino para que también las validen en el juez en línea *HackerRank*<sup>1</sup>.

## 1.1. Series, sumatorias e inducción matemática

### 1.1.1. Teorema (sumatoria de Gauss)

$$\sum_{i=1}^n i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}, \text{ para } n \geq 1, n \in \mathbb{Z}^+$$

**Demostración:**

Sea  $S = 1 + 2 + 3 + \cdots + n$ , como el orden de los sumandos no altera el resultado entonces  $S = n + (n - 1) + (n - 2) + \cdots + 1$

$$\begin{array}{rccccccccccccc} S & = & 1 & + & 2 & + & 3 & + & \cdots & + & n \\ S & = & n & + & (n-1) & + & (n-2) & + & \cdots & + & 1 \\ \hline 2S & = & \underbrace{(n+1) + (n+1) + (n+1) + \cdots + (n+1)}_{n \text{ veces}} \end{array}$$

---

<sup>1</sup><https://www.hackerrank.com/data-structure-utp>

$$2S = n(n + 1)$$

$$S = \frac{n(n + 1)}{2}$$

Como  $S$  es igual a  $1 + 2 + 3 + \dots + n$  entonces  $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$ , para  $n \in \mathbb{Z}^+$ .

### Ejemplo 1:

Probar o refutar utilizando la técnica de demostración por inducción matemática que:

$$\sum_{i=1}^n i = 1 + 2 + 3 + \dots + n = \frac{n(n + 1)}{2}, \text{ para } n \geq 1, n \in \mathbb{Z}^+$$

#### Paso base $n = 1$ :

$$\underbrace{\sum_{i=1}^1 i}_{\text{resultado a partir de la sumatoria de términos}} = \underbrace{\frac{1(1 + 1)}{2}}_{\text{resultado a partir de la fórmula}} = 1$$

Como se obtiene el mismo resultado en la sumatoria de términos y en la fórmula que es la solución de la sumatoria, entonces la solución de la sumatoria es correcta en el paso base, y la demostración continúa en el paso inductivo.

#### Paso inductivo $n = k, k \in \mathbb{Z}^+$ :

$$\sum_{i=1}^k i = 1 + 2 + 3 + \dots + k = \frac{k(k + 1)}{2}, \text{ se asume que la solución de la sumatoria de términos}$$

es correcta para un valor de  $n = k$ .

#### Paso pos-inductivo $n = k + 1$ :

$$\sum_{i=1}^{k+1} i = \underbrace{1 + 2 + 3 + \dots + k}_{\text{Se reemplaza por su equivalente}} + (k + 1) = \frac{(k + 1)(k + 2)}{2}$$

en el paso inductivo

$$\frac{k(k + 1)}{2} + (k + 1) = \frac{(k + 1)(k + 2)}{2}$$

$$(k + 1) \left[ \frac{k}{2} + 1 \right] = \frac{(k + 1)(k + 2)}{2}$$

$$(k + 1) \left[ \frac{k + 2}{2} \right] = \frac{(k + 1)(k + 2)}{2}$$

$$\frac{(k + 1)(k + 2)}{2} = \frac{(k + 1)(k + 2)}{2}$$

Se pudo comprobar que se cumple la igualdad. Como se cumplen los tres pasos de técnica de demostración por inducción matemática, entonces queda demostrada la validez de la solución de la sumatoria de términos.

**Ejemplo 2:**

Obtener una fórmula que sea la solución de la siguiente suma de términos:

$$2 + 4 + 6 + 8 + \cdots + 2 \cdot n, \text{ para } n \geq 1, n \in \mathbb{Z}^+.$$

$$\begin{aligned} 2 + 4 + 6 + 8 + \cdots + 2 \cdot n &= 2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + 2 \cdot 4 + \cdots + 2 \cdot n \\ &= 2 \cdot (1 + 2 + 3 + 4 + \cdots + n) \\ &= 2 \cdot \left( \sum_{i=1}^n i \right) \\ &= 2 \cdot \left( \frac{n(n+1)}{2} \right) \\ &= n(n+1) \end{aligned}$$

Por lo tanto,  $\sum_{i=1}^n 2 \cdot i = 2 + 4 + 6 + 8 + \cdots + 2 \cdot n = n(n+1)$

**Gráficamente:**

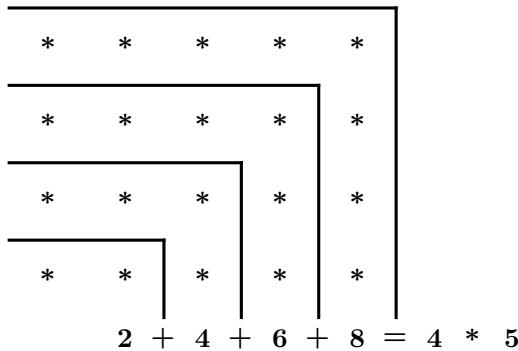


Figura 1.1. Números Rectangulares.

**Ejemplo 3:**

Obtener una fórmula que sea la solución de la siguiente suma de términos:

$$1 + 3 + 5 + 7 + \cdots + 2n - 1, \text{ para } n \geq 1, n \in \mathbb{Z}^+.$$

$$\begin{aligned} 1 + 3 + 5 + \cdots + 2 \cdot n - 1 &= 2 \cdot 1 - 1 + 2 \cdot 2 - 1 + 2 \cdot 3 - 1 + \cdots + 2 \cdot n - 1 \\ &= (2 \cdot 1 + 2 \cdot 2 + 2 \cdot 3 + \cdots + 2 \cdot n) - (\underbrace{1 + 1 + 1 + \cdots + 1}_{n \text{ veces}}) \\ &= 2 \cdot (1 + 2 + 3 + 4 + \cdots + n) - n \\ &= 2 \cdot \left( \sum_{i=1}^n i \right) - n \end{aligned}$$

$$\begin{aligned}
 &= 2 \cdot \left( \frac{n(n+1)}{2} \right) - n \\
 &= n(n+1) - n \\
 &= n^2 + n - n \\
 &= n^2
 \end{aligned}$$

Por lo tanto,  $\sum_{i=1}^n (2 \cdot i - 1) = 1 + 3 + 5 + \dots + 2 \cdot n - 1 = n^2$ .

A continuación, se presenta una forma alternativa de obtener la solución de la suma de términos.

Sea  $S = 1 + 3 + 5 + \dots + 2n - 1$ , como el orden de los sumandos no altera el resultado, entonces  $S = (2n - 1) + (2n - 3) + (2n - 5) + \dots + 1$ .

$$\begin{aligned}
 S &= 1 + 3 + 5 + \dots + 2n - 1 \\
 S &= (2n - 1) + (2n - 3) + (2n - 5) + \dots + 1 \\
 \hline
 2S &= \underbrace{(2n) + (2n) + (2n) + \dots + (2n)}_{n \text{ veces}} \\
 2S &= n(2n) \\
 S &= n^2
 \end{aligned}$$

Como  $S$  es igual a  $1 + 3 + 5 + \dots + 2n - 1$ , entonces  $1 + 3 + 5 + \dots + 2n - 1 = n^2$ , para  $n \in \mathbb{Z}^+$ .

**Gráficamente:**

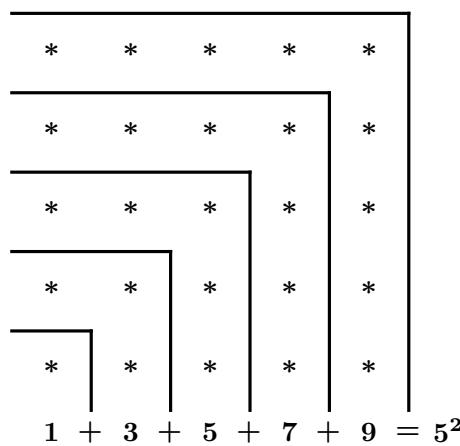


Figura 1.2. Números Cuadrados.

**Ejemplo 4:**

Probar o refutar utilizando la técnica de demostración por inducción matemática que:

$$\sum_{i=1}^n (2 \cdot i - 1) = 1 + 3 + 5 + 7 + \dots + (2 \cdot n - 1) = n^2, \text{ para } n \geq 1, n \in \mathbb{Z}^+$$

**Paso base  $n = 1$ :**

$$\underbrace{\sum_{i=1}^1 (2 \cdot i - 1)}_{\text{Resultado a partir de la sumatoria de términos}} = 2 \cdot 1 - 1 = 1 \quad \underbrace{1^2 = 1}_{\text{Resultado a partir de la fórmula}}$$

Como se obtiene el mismo resultado en la sumatoria de términos y en la fórmula que es la solución de la sumatoria, entonces la solución de la sumatoria es correcta en el paso base, y la demostración continúa en el paso inductivo.

**Paso inductivo  $n = k, k \in \mathbb{Z}^+$ :**

$\sum_{i=1}^k (2 \cdot i - 1) = 1 + 3 + 5 + \dots + (2 \cdot k - 1) = k^2$ , se asume que la solución de la sumatoria de términos es correcta para un valor de  $n = k$ .

**Paso pos-inductivo  $n = k + 1$ :**

$$\sum_{i=1}^{k+1} (2 \cdot i - 1) = \underbrace{1 + 3 + 5 + \dots + (2 \cdot k - 1)}_{\text{Se reemplaza por su equivalente}} + 2 \cdot (k + 1) - 1 = (k + 1)^2$$

en el paso inductivo

$$\begin{aligned} k^2 &+ 2 \cdot (k + 1) - 1 = (k + 1)^2 \\ k^2 &+ 2 \cdot k + 2 - 1 = (k + 1)^2 \\ k^2 + 2 \cdot k + 1 &= (k + 1)^2 \\ (k + 1)^2 &= (k + 1)^2 \end{aligned}$$

Como se cumplen los tres pasos de técnica de demostración por inducción matemática, entonces queda demostrada la validez de la solución de la sumatoria de términos.

**1.1.2. ¿Particularidad o generalidad?**

Considerar la Figura 1.3 en la que se tiene una fila con los primeros cinco números enteros positivos y el resultado de la suma de ellos utilizando la fórmula de la sumatoria de Gauss:

$$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \frac{5 \cdot (5+1)}{2}$$

Figura 1.3. Primera fila.

En la Figura 1.4 se ha generado la segunda fila, obteniendo cada valor al multiplicar los valores de la primera fila por 2, la suma de los elementos de la segunda fila es:  $2 \cdot [1 + 2 + 3 + 4 + 5] = 2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$ .

$$\begin{array}{cccccc} 2 & 4 & 6 & 8 & 10 & 2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 1 & 2 & 3 & 4 & 5 & 1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \end{array}$$

Figura 1.4. Primeras dos filas.

En la Figura 1.5 se ha generado la tercera fila, obteniendo cada valor al multiplicar los valores de la primera fila por 3, la suma de los elementos de la tercera fila es:  $3 \cdot [1 + 2 + 3 + 4 + 5] = 3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$ .

$$\begin{array}{cccccc} 3 & 6 & 9 & 12 & 15 & 3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 2 & 4 & 6 & 8 & 10 & 2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 1 & 2 & 3 & 4 & 5 & 1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \end{array}$$

Figura 1.5. Primeras tres filas.

En la Figura 1.6 se ha generado la cuarta fila, obteniendo cada valor al multiplicar los valores de la primera fila por 4, la suma de los elementos de la cuarta fila es:  $4 \cdot [1 + 2 + 3 + 4 + 5] = 4 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$ .

$$\begin{array}{cccccc} 4 & 8 & 12 & 16 & 20 & 4 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 3 & 6 & 9 & 12 & 15 & 3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 2 & 4 & 6 & 8 & 10 & 2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\ 1 & 2 & 3 & 4 & 5 & 1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \end{array}$$

Figura 1.6. Primeras cuatro filas.

En la Figura 1.7 se ha generado la quinta fila, obteniendo cada valor al multiplicar los valores de la primera fila por 5, la suma de los elementos de la quinta fila es:  $5 \cdot [1 + 2 + 3 + 4 + 5] = 5 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$ .

<b>5</b>	<b>10</b>	<b>15</b>	<b>20</b>	<b>25</b>	$5 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$
<b>4</b>	<b>8</b>	<b>12</b>	<b>16</b>	<b>20</b>	$4 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$
<b>3</b>	<b>6</b>	<b>9</b>	<b>12</b>	<b>15</b>	$3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$
<b>2</b>	<b>4</b>	<b>6</b>	<b>8</b>	<b>10</b>	$2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$
<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	$1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]$

Figura 1.7. Primeras cinco filas.

En la Figura 1.8 se identifican dos regiones diferentes cuya suma de sus elementos tiene el mismo valor. La región de la izquierda es una matriz  $5 \times 5$  y la región de la derecha es una matriz  $5 \times 1$

<b>5</b>	<b>10</b>	<b>15</b>	<b>20</b>	<b>25</b>	<b><math>5 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]</math></b>
<b>4</b>	<b>8</b>	<b>12</b>	<b>16</b>	<b>20</b>	<b><math>4 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]</math></b>
<b>3</b>	<b>6</b>	<b>9</b>	<b>12</b>	<b>15</b>	<b><math>3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]</math></b>
<b>2</b>	<b>4</b>	<b>6</b>	<b>8</b>	<b>10</b>	<b><math>2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]</math></b>
<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b><math>1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right]</math></b>

Figura 1.8. Dos regiones diferentes que suman el mismo valor.

En la Figura 1.9 se plantea una forma diferente para sumar los elementos de la matriz  $5 \times 5$ , creando 5 regiones en forma de escuadra a partir de la esquina inferior izquierda, la primera escuadra tiene el elemento 1 que se puede representar como un  $1^3$ , la segunda escuadra tiene los elementos 2, 4 y 2, los cuales suman un 8 que se puede representar como un  $2^3$ , la tercera escuadra tiene los elementos 3, 6, 9, 6 y 3, los cuales suman un 27 que se puede representar como un  $3^3$ , la cuarta escuadra tiene los elementos 4, 8, 12, 16, 12, 8 y 4, los cuales suman un 64 que se puede representar como un  $4^3$ , por último, la quinta escuadra tiene los elementos 5, 10, 15, 20, 25, 20, 15, 10 y 5, los cuales suman un 125 que se puede representar como un  $5^3$ .

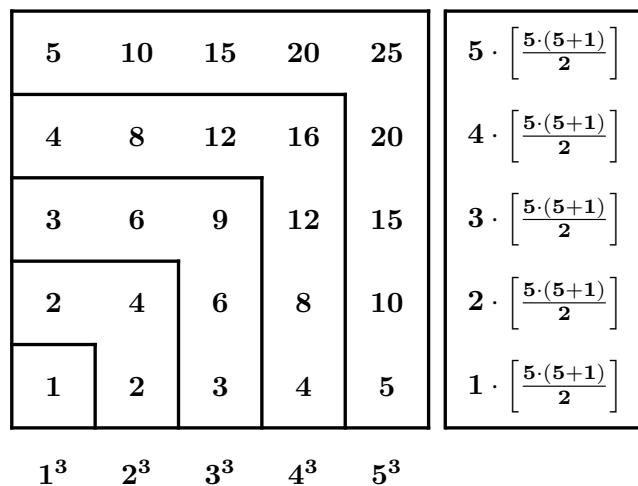


Figura 1.9. Diferentes regiones en forma de escuadra en la matriz  $5 \times 5$ .

La suma de las 5 regiones de escuadra generadas a partir de la matriz  $5 \times 5$  es igual a la suma de los elementos de la matriz  $5 \times 1$ , por lo tanto:

$$\begin{aligned}
 1^3 + 2^3 + 3^3 + 4^3 + 5^3 &= 1 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] + 2 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] + 3 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] + 4 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] + 5 \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\
 &= [1 + 2 + 3 + 4 + 5] \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\
 &= \left[ \frac{5 \cdot (5+1)}{2} \right] \cdot \left[ \frac{5 \cdot (5+1)}{2} \right] \\
 &= \frac{5^2 \cdot (5+1)^2}{4}
 \end{aligned}$$

Con la deducción anterior, se puede garantizar, de forma particular, que se cumple hasta un  $n = 5$ , ¿también se cumple de forma genérica para cualquier valor de  $n$  en los números enteros positivos?, es decir:

$$\sum_{i=1}^n i^3 = 1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n^2 \cdot (n+1)^2}{4}, \text{ para } n \in \mathbb{Z}^+$$

La duda se puede despejar apoyados en la técnica de demostración por inducción matemática, para lo cual tenemos:

**Paso base  $n = 1$ :**

$$\underbrace{\sum_{i=1}^1 i^3}_{\text{Resultado a partir de la sumatoria de términos}} = 1^3 = 1 = \underbrace{\frac{1^2 \cdot (1+1)^2}{4}}_{\text{Resultado a partir de la fórmula}} = \frac{1 \cdot (4)}{4} = 1$$

Resultado a partir de la sumatoria de términos

Resultado a partir de la fórmula

Como se obtiene el mismo resultado en la sumatoria de términos y en la fórmula que es la solución de la sumatoria, entonces la solución de la sumatoria es correcta en el paso base, y la demostración continúa en el paso inductivo.

**Paso inductivo  $n = k$ ,  $k \in \mathbb{Z}^+$ :**

$\sum_{i=1}^k i^3 = 1^3 + 2^3 + 3^3 + \dots + k^3 = \frac{k^2 \cdot (k+1)^2}{4}$ , se asume que la solución de la sumatoria de términos es correcta para un valor de  $n = k$ .

**Paso pos-inductivo  $n = k + 1$ :**

$$\sum_{i=1}^{k+1} i^3 = \underbrace{1^3 + 2^3 + 3^3 + \dots + k^3}_{\text{Se reemplaza por su equivalente}} + (k+1)^3 = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

en el paso inductivo

$$\frac{k^2 \cdot (k+1)^2}{4} + \frac{4 \cdot (k+1)^3}{4} = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

$$\frac{(k+1)^2}{4} \cdot [k^2 + 4 \cdot (k+1)] = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

$$\frac{(k+1)^2}{4} \cdot [k^2 + 4k + 4] = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

$$\frac{(k+1)^2}{4} \cdot (k+2)^2 = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

$$\frac{(k+1)^2 \cdot (k+2)^2}{4} = \frac{(k+1)^2 \cdot (k+2)^2}{4}$$

Como se cumplen los tres pasos de la demostración por inducción matemática, entonces queda demostrada la validez de la solución de la sumatoria de términos, independientemente del valor de  $n$  en los números enteros positivos.

**1.1.3. Serie geométrica**

Una serie o progresión geométrica es una sucesión de la forma:

$$a, a \cdot r, a \cdot r^2, a \cdot r^3, \dots, a \cdot r^n$$

Donde el término inicial  $a$  y la razón constante  $r$  son números reales con  $r \neq 0$  y  $n$  que toma valores en los números naturales.

**1.1.4. Suma de términos de una serie geométrica**

**Teorema:**

Si  $a$  y  $r$  son números reales, con  $r \neq 0$  y  $n$  es un número natural, entonces:

$$\sum_{i=0}^n a \cdot r^i = \begin{cases} a \cdot (n+1) & \text{si } r = 1 \\ \frac{a \cdot r^{n+1} - a}{r - 1} & \text{si } r \neq 1 \end{cases}$$

### Demostración:

Se debe demostrar cada uno de los dos casos del teorema de forma independiente. Si los dos casos se cumplen entonces queda demostrada la validez del teorema.

- Caso donde  $r = 1$

$$\sum_{i=0}^n a \cdot 1^i = \sum_{i=0}^n a \cdot 1 = \sum_{i=0}^n a = \underbrace{a + a + a + a + \cdots + a}_{n+1 \text{ veces}} = a \cdot (n + 1).$$

Queda demostrado el caso.

- Caso donde  $r \neq 1$

Sea  $S = a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \cdots + a \cdot r^n$ , al multiplicar a ambos lados de la igualdad por  $-r$  se sigue conservando la igualdad, donde se obtiene:  $-S \cdot r = -a \cdot r - a \cdot r^2 - a \cdot r^3 - a \cdot r^4 - \cdots - a \cdot r^{n+1}$ ; al sumar ambas igualdades se tiene:

$$\begin{array}{rcl} S = a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \cdots + a \cdot r^n \\ -S \cdot r = -a \cdot r - a \cdot r^2 - a \cdot r^3 - a \cdot r^4 - \cdots - a \cdot r^n - a \cdot r^{n+1} \\ \hline S - S \cdot r = a & & - a \cdot r^{n+1} \end{array}$$

$$\begin{aligned} S \cdot (1 - r) &= a - a \cdot r^{n+1} \\ S &= \frac{a - a \cdot r^{n+1}}{1 - r} \\ S &= \frac{-1}{-1} \cdot \frac{a - a \cdot r^{n+1}}{1 - r} \\ S &= \frac{-a + a \cdot r^{n+1}}{-1 + r} \\ S &= \frac{a \cdot r^{n+1} - a}{r - 1} \end{aligned}$$

Como  $S$  es igual a  $a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \cdots + a \cdot r^n$  entonces  $a + a \cdot r + a \cdot r^2 + a \cdot r^3 + \cdots + a \cdot r^n = \frac{a \cdot r^{n+1} - a}{r - 1}$ , para  $n \in \mathbb{N}$ .

Queda demostrado el caso.

Como se cumplen todos los casos del teorema, entonces queda demostrada la validez del teorema de la suma de términos de la serie o progresión geométrica.

### 1.1.5. Serie geométrica que converge en el infinito

La suma de términos de una serie geométrica converge en el infinito, siempre y cuando el valor absoluto de la razón es estrictamente menor a uno ( $|r| < 1$ ), para lo cual tenemos:

$$\lim_{\substack{|r| < 1, \\ n \rightarrow +\infty}} \left( \sum_{i=0}^n a \cdot r^i \right) = \lim_{\substack{|r| < 1, \\ n \rightarrow +\infty}} \left( \frac{a - a \cdot r^{n+1}}{1 - r} \right) = \lim_{\substack{|r| < 1, \\ n \rightarrow +\infty}} \left( \frac{a}{1 - r} - \frac{\cancel{a \cdot r^{n+1}}}{\cancel{1 - r}} \right) = \frac{a}{1 - r}$$

### 1.1.6. Fórmulas de sumatorias útiles

Las siguientes, son algunas de las sumatorias más importantes (o más utilizadas) en matemáticas computacionales junto con su solución. Esta información será de gran importancia para su uso en la solución de relaciones de recurrencia, para el trabajo con notaciones computacionales y para la construcción de soluciones eficientes en tiempo de ejecución.

- $\sum_{i=1}^n i = \frac{n(n+1)}{2}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .
- $\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .
- $\sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .
- $\sum_{i=1}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .
- $\sum_{i=0}^n a \cdot r^i = \frac{ar^{n+1} - a}{r - 1}$ , para  $r \neq 0, r \neq 1, a, r \in \mathbb{R}, n \geq 0, n \in \mathbb{N}$ .
- Si  $|r| < 1$  y  $r \neq 0$ , entonces  $\sum_{i=0}^{\infty} a \cdot r^i = \frac{a}{1 - r}$ , para  $i \geq 0, i \in \mathbb{N}$ .
- $\sum_{i=1}^n i \cdot 2^i = (n-1)2^{n+1} + 2$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .
- $\sum_{i=1}^n \frac{i}{2^i} = 2 - \frac{n+2}{2^n}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .

### 1.1.7. Preguntas tipo Saber Pro

En Colombia, todos los estudiantes de pregrado (independiente del programa académico) son evaluados en el último año de carrera con la Prueba Saber Pro [ICF24], la cual es realizada por el ICFES (Instituto Colombiano del Fomento a la Educación Superior). Dicha prueba es uno de los requisitos para poder obtener el grado de la carrera que se está estudiando.

Es de vital importancia que los estudiantes, desde los primeros semestres en la Universidad, se familiaricen con el tipo de preguntas que se encontrarán en la Prueba Saber Pro, para obtener resultados de calidad.

Se trabajarán tres preguntas, corresponden a la temática de series geométricas que convergen en el infinito.

La metodología de trabajo es la siguiente: el estudiante debe contestar cada una de las preguntas tipo Saber Pro en un tiempo que oscile entre dos y tres minutos, luego se presentarán una o dos heurísticas que sirven como estrategia para seleccionar la respuesta correcta y, por último, se presentará la solución analítica.

### Pregunta 1:

En la Figura 1.10 se tiene un cuadrado de lado 1, el cual se divide en cuatro partes iguales y se sombrean dos; luego, se toma uno de los cuadrados de lado 0.5, se divide en cuatro partes iguales y se sombrean dos. El proceso sigue así sucesivamente.

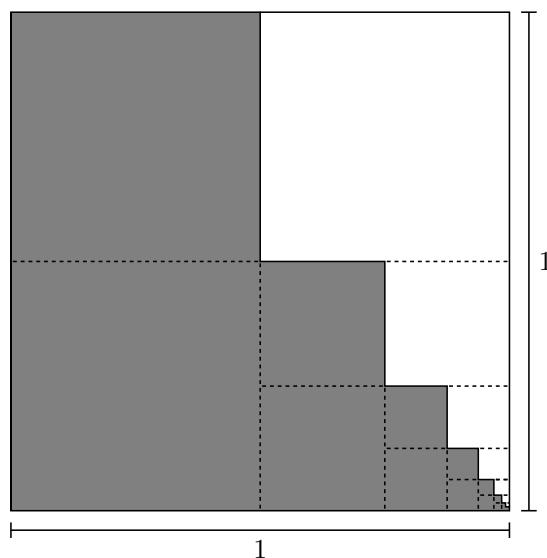


Figura 1.10. Cuadrado de lado 1 con la región sombreada según el procedimiento.

El área de la región sombreada está dada por:

- A.)  $3/4$
- B.)  $3/5$
- C.)  $2/3$
- D.)  $5/7$

### Solución con heurísticas:

Utilizando una heurística de buena lectura de gráfico, en la Figura 1.11 se logra evidenciar que se genera una sucesión de cuadrados de color gris oscuro en la parte inferior, una sucesión de cuadrados de color gris claro en el centro y una sucesión de cuadrados de color blanco en la parte derecha. Las tres sucesiones de cuadrados cubren toda el área del cuadrado. Además, dos de las tres sucesiones de cuadrados están sombreadas, por lo tanto, la respuesta a la pregunta es  $2/3$ .

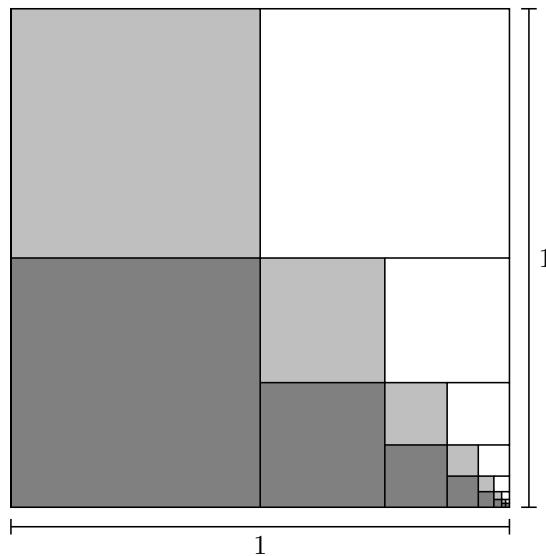


Figura 1.11. Heurística: sucesiones de cuadrados.

**Solución analítica:**

$$\underbrace{\frac{1}{2} + \frac{1}{2} \cdot \left(\frac{1}{4}\right) + \frac{1}{2} \cdot \left(\frac{1}{4}\right)^2 + \frac{1}{2} \cdot \left(\frac{1}{4}\right)^3 + \dots}_{\text{Suma de términos de una serie geométrica que converge en el infinito con } a=\frac{1}{2} \text{ y } r=\frac{1}{4}} = \frac{\frac{1}{2}}{1 - \frac{1}{4}} = \frac{\frac{1}{2}}{\frac{3}{4}} = \frac{2}{3}$$

**Pregunta 2:**

En la Figura 1.10 se observa que se tiene un cuadrado de lado 1, el cual se divide en cuatro partes iguales y se sombrean dos; luego, se toma uno de los cuadrados de lado 0.5, se divide en cuatro partes iguales y se sombrean dos. El proceso sigue así sucesivamente.

El perímetro de la región sombreada esta dado por:

- A.) 3.25
- B.) 3.50
- C.) 3.75
- D.) 4.00

**Solución con heurísticas - suma de los anchos de los escalones:**

Utilizando una heurística de buena lectura de gráfico, en la Figura 1.12 se puede evidenciar que se pueden proyectar sobre la base cada uno de los anchos de los escalones y que, independientemente de que el proceso se repita infinitamente, la suma total de los anchos de los escalones será uno. Lo mismo sucede si proyectamos los altos de los escalones al costado izquierdo de la figura, la suma total de los altos de los escalones también será uno, por lo tanto, el perímetro de la figura será 4.

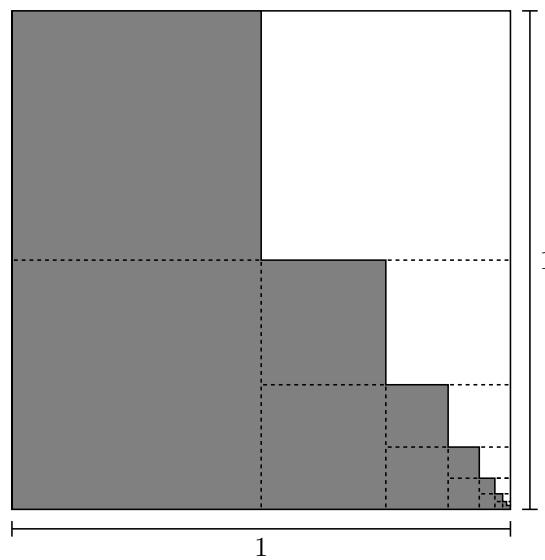


Figura 1.12. Heurística: proyectar anchos y altos de los escalones a la base.

#### Solución con heurísticas - recorrido de una hormiga por el perímetro de la figura:

Utilizando una heurística de buena lectura de gráfico, en la Figura 1.13 se pueden tener algunos puntos de control de una hormiga que camina por el perímetro de la figura. Los puntos de control sirven para descartar rápidamente tres de las cuatro opciones de respuesta y proclamar como respuesta correcta la opción que no se ha podido falsear. En el último punto de control, la hormiga ya lleva un recorrido de 3,75 y aún queda un pedazo de perímetro por recorrer; por lo tanto, es imposible que la respuesta correcta sea alguna de la opciones A, B, o C (con perímetros de 3,25, 3,50 y 3,75 respectivamente), quedando con la opción “D.) 4.00” como la única opción de respuesta correcta. Recordar que en una prueba Saber (ya sea 11, TyT o Pro) solamente se está pidiendo que el evaluado seleccione una respuesta, no se está pidiendo que se haga una demostración formal del porqué se escoge o no una de las opciones.

#### Solución analítica:

Se hará uso de la series geométricas que convergen en el infinito a partir de la suma de los anchos de los escalones de la Figura 1.12, donde tenemos:

$$\underbrace{\frac{1}{2} + \frac{1}{2} \cdot \left(\frac{1}{2}\right) + \frac{1}{2} \cdot \left(\frac{1}{2}\right)^2 + \frac{1}{2} \cdot \left(\frac{1}{2}\right)^3 + \dots}_{\text{Suma de términos de una serie geométrica que converge en el infinito con } a=\frac{1}{2} \text{ y } r=\frac{1}{2}} = \frac{\frac{1}{2}}{1 - \frac{1}{2}} = \frac{\frac{1}{2}}{\frac{1}{2}} = \frac{1}{1} = 1$$

De forma simétrica, se aplica el mismo análisis para obtener la suma de los altos de los escalones, obteniendo también un 1, con lo cual, se pudo apreciar que el perímetro de la figura es 4 (1 de la parte de abajo, 1 por la parte izquierda, 1 por los anchos de los escalones y 1 por los altos de los escalones).

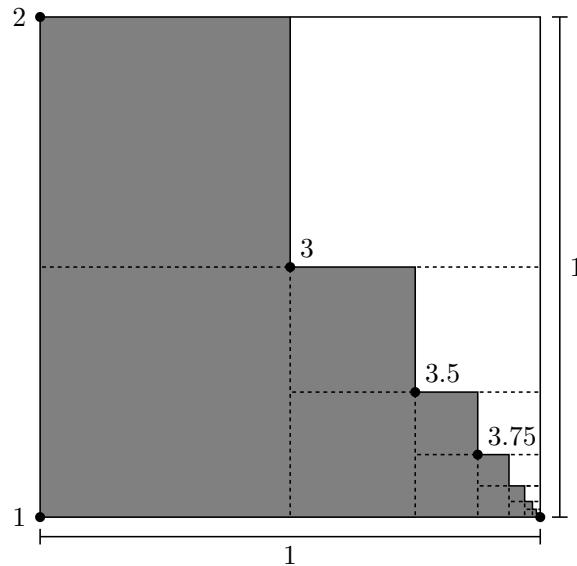


Figura 1.13. Heurística: recorrido de una hormiga por el perímetro.

**Pregunta 3:**

Desde una altura de un metro se deja caer una pelota que rebota medio metro, luego rebota un cuarto de metro, luego rebota un octavo de metro y así sucesivamente hasta que se detiene.

¿Cuál es la suma total en metros de la trayectoria que ha recorrido la pelota hasta que se detiene?

- A.) 2.00 m
- B.) 2.50 m
- C.) 2.75 m
- D.) 3.00 m

**Solución con heurísticas - recorrido de una hormiga por la trayectoria:**

Utilizando una heurística de buena lectura de gráfico, en la Figura 1.14 se tienen algunos puntos de control de una hormiga que camina por la trayectoria generada por la pelota hasta que se detiene. Los puntos de control sirven para descartar rápidamente tres de las cuatro opciones de respuesta y proclamar como respuesta correcta la opción que no se ha podido falsear. En el último punto de control, la hormiga ya lleva un recorrido de 2,75 m y aun queda trayectoria de la pelota por recorrer, por lo tanto es imposible que la respuesta correcta sea alguna de la opciones A, B, o C (con trayectorias de 2,00 m, 2,50 m y 2,75 m respectivamente), quedando con la opción D.) 3.00 m como la única opción de respuesta correcta.

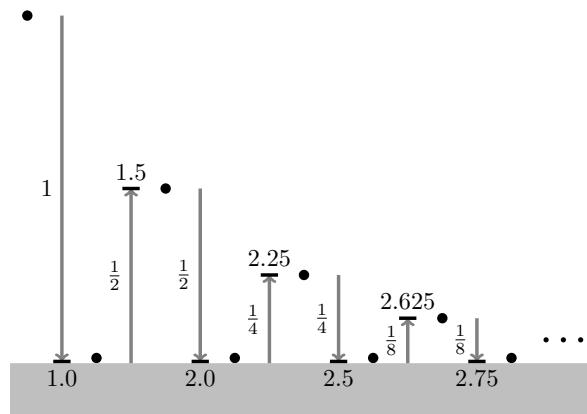


Figura 1.14. Heurística: recorrido de una hormiga por la trayectoria.

### Solución analítica:

Se hará uso de las series geométricas que convergen en el infinito, de la suma términos que se van generando a partir de la trayectoria que se genera con cada rebote de la pelota contra el piso. El primer término que se suma es un uno, porque la pelota parte desde el reposo a una altura de un metro, luego se suma dos veces el  $\frac{1}{2}$  porque la pelota rebota medio metro (en la trayectoria la pelota sube medio metro y baja de nuevo medio metro), y así sucesivamente para cada rebote que es la mitad de la altura alcanzada en el rebote anterior, donde tenemos:

$$1 + 2 \cdot \frac{1}{2} + 2 \cdot \frac{1}{2} \cdot \left(\frac{1}{2}\right) + 2 \cdot \frac{1}{2} \cdot \left(\frac{1}{2}\right)^2 + \dots = 1 + 2 \cdot \underbrace{\left[ \frac{1}{2} + \frac{1}{2} \cdot \left(\frac{1}{2}\right) + \frac{1}{2} \cdot \left(\frac{1}{2}\right)^2 + \dots \right]}_{\text{Suma de términos de una serie geométrica que converge en el infinito con } a=\frac{1}{2} \text{ y } r=\frac{1}{2}}$$

$$\begin{aligned} &= 1 + 2 \cdot \left[ \frac{\frac{1}{2}}{1 - \frac{1}{2}} \right] \\ &= 1 + 2 \cdot \left[ \frac{\frac{1}{2}}{\frac{1}{2}} \right] \\ &= 1 + 2 \cdot \left[ \frac{2}{2} \right] \\ &= 1 + 2 \cdot [1] \\ &= 1 + 2 \\ &= 3 \end{aligned}$$

## 1.2. Tallas de los tipos de datos enteros

Se usará la temática de series geométricas para determinar las tallas de los diferentes tipos de datos para los números enteros en el computador, en el caso específico de Lenguaje C/C++.

### 1.2.1. Unsigned int

El número más grande que se puede representar en un entero sin signo (*unsigned int*) que utiliza una representación (o una arquitectura) de 32 bits, es el siguiente:

$$\begin{aligned}
 (111111111111111111111111111111111111)_2 &= 2^{31} + 2^{30} + \dots + 2^0 \\
 &= 2^0 + 2^1 + 2^2 + \dots + 2^{31} \\
 &= 2^{32} - 1 \\
 &= 4,294,967,295 \\
 &\approx 4 \cdot 10^9
 \end{aligned}$$

### 1.2.2. Signed int

El número más grande que se puede representar en un entero con signo (*signed int*, o *int*) que utiliza una representación (o una arquitectura) de 32 bits, es el siguiente:

$$\begin{aligned}
 (111111111111111111111111111111)_2 &= 2^{30} + 2^{29} + \dots + 2^1 + 2^0 \\
 &= 2^0 + 2^1 + 2^2 + \dots + 2^{30} \\
 &= 2^{31} - 1 \\
 &= 2,147,483,647 \\
 &\approx 2 \cdot 10^9
 \end{aligned}$$

### 1.2.3. Unsigned long long int

El número más grande que se puede representar en un entero largo largo sin signo (*unsigned long long int*) que utiliza una representación (o una arquitectura) de 64 bits, es el siguiente:

#### 1.2.4. Signed long long int

El número más grande que se puede representar en un entero largo largo con signo (*signed long long int*, o *long long int*) que utiliza una representación (o una arquitectura) de 64 bits, es el siguiente:

### Ejemplo 5:

El siguiente programa en Lenguaje C, sirve para validar los máximos valores que se pueden almacenar en los diferentes tipos de datos de las variables enteras.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXUI32 4294967295
#define MAXI32 2147483647
#define MAXUI64 18446744073709551615
#define MAXI64 9223372036854775807

int main()
{
    unsigned int ui32bits = MAXUI32;
    int i32bits = MAXI32;
    unsigned long long int ui64bits = MAXUI64;
    long long int i64bits = MAXI64;

    printf("Maximo entero sin signo (32 bits): %u\n", ui32bits);
    printf("Maximo entero sin signo mas uno (32 bits): %u\n", ui32bits + 1)
        ;

    printf("Maximo entero con signo (32 bits): %d\n", i32bits);
    printf("Maximo entero con signo mas uno (32 bits): %d\n", i32bits + 1);

    printf("Maximo entero sin signo (64 bits): %llu\n", ui64bits);
    printf("Maximo entero sin signo mas uno (64 bits): %llu\n", ui64bits +
        1);

    printf("Maximo entero con signo (64 bits): %lld\n", i64bits);
    printf("Maximo entero con signo mas uno (64 bits): %lld\n", i64bits +
        1);

    return 0;
}
```

La salida del programa anterior es la siguiente:

```
Maximo entero sin signo (32 bits): 4294967295
Maximo entero sin signo mas uno (32 bits): 0
Maximo entero con signo (32 bits): 2147483647
Maximo entero con signo mas uno (32 bits): -2147483648
Maximo entero sin signo (64 bits): 18446744073709551615
Maximo entero sin signo mas uno (64 bits): 0
Maximo entero con signo (64 bits): 9223372036854775807
Maximo entero con signo mas uno (64 bits): -9223372036854775808
```

Figura 1.15. Salida del programa de máximos valores de los tipos de variables enteras.

### 1.3. Retos de programación resueltos

En esta sección se trabajarán cuatro retos de programación que se apoyan en el repaso de herramientas matemáticas, estos son:

- Números triangulares
- La ladrona de libros
- Felipe y la secuencia
- La función de Dangie

#### 1.3.1. Números triangulares

**Nombre original:** Triangular Numbers<sup>2</sup>.

**Fuente:** UTP Open 2013.

**Fecha:** 4 de Mayo de 2013.

**Autor:** Hugo Humberto Morales Peña.

Los *números triangulares* son todos aquellos números enteros positivos que representan una cantidad de asteriscos, que pueden conformar un triángulo compacto con la misma cantidad de asteriscos en cada uno de sus tres lados.

Los primeros cinco números triangulares son presentados en la Figura 1.16.

1	3	6	10	15
*	*	* * *	* * * * * *	* * * * * * * * * *

Figura 1.16. Primeros cinco números triangulares.

El trabajo a realizar en este reto de programación es determinar si un número entero  $n$  es triangular o no.

#### Formato de entrada:

La entrada puede contener varios casos de prueba. Cada caso de prueba se presenta en una línea independiente y contiene un entero  $n$  ( $1 \leq n \leq 16 \cdot 10^{18}$ ). La entrada finaliza con un caso de prueba en el que  $n$  tiene el valor de 0, caso que no debe ser procesado.

#### Formato de salida:

Por cada caso de prueba de la entrada, se debe imprimir un YES o un NO dependiendo si el número de la entrada es o no triangular. Cada caso valido de entrada debe generar una línea de salida.

<sup>2</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/triangular-numbers-2-3>

**Ejemplo de entrada:**

```
1
15
16
101
15999999994386249876
0
```

**Ejemplo de salida:**

```
YES
YES
NO
NO
YES
```

**Primera aproximación a la solución**

En un reto de programación, lo fundamental es entender el problema y plantear soluciones que generen resultados correctos, sin perder de vista el tipo de datos que se debe utilizar dependiendo de las tallas de las variables; en este caso, la variable  $n$  ( $1 \leq n \leq 16 \cdot 10^{18}$ ) tiene que ser obligatoriamente del tipo `unsigned long long int`.

Una estrategia que genera resultados válidos es ir generando uno a uno los números triangulares y compararlos con el valor de  $n$ , el proceso termina en el momento en el que el número triangular sea mayor o igual al valor de  $n$ , imprimiendo el resultado que corresponde según sea el caso, un YES o un NO.

En el siguiente programa de Lenguaje C se implementa la estrategia anterior

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    unsigned long long int n, i, triangular;

    while(scanf("%llu", &n) && (n > 0))
    {
        triangular = 0;
        for(i = 1; triangular < n; i++)
            triangular += i;

        if(triangular == n)
            printf("YES\n");
        else
            printf("NO\n");
    }

    return 0;
}
```

El programa anterior, es una solución que genera resultados correctos, pero es muy costosa en tiempo de ejecución y genera en el juez en línea un veredicto de tiempo límite excedido (en inglés el veredicto es **Time Limit Exceeded**) .

El siguiente fragmento de código en la solución es el que permite calcular el valor del número triangular más grande que es menor o igual a  $n$ , pero con un costo computacional muy alto

```

||   triangular = 0;
||   for(i = 1; triangular < n; i++)
||       triangular += i;

```

A continuación, se va a realizar el mismo trabajo, pero, de forma eficiente, haciendo uso de las matemáticas.

### Enfoque matemático del reto

Matemáticamente se puede calcular el valor del número triangular más grande que sea menor o igual a  $n$  apoyados en la solución de la sumatoria de Gauss, con la cual se plantea y resuelve un polinomio de grado 2, como se muestra a continuación:

$$\begin{aligned} \frac{k(k+1)}{2} &= n \\ k(k+1) &= 2n \\ k^2 + k &= 2n \\ k^2 + k - 2n &= 0 \end{aligned}$$

Es importante recordar que la fórmula de la cuadrática para obtener las raíces de un polinomio de grado 2 es:

$$k = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En la ecuación cuadrática que estamos trabajando se tiene que  $a = 1$ ,  $b = 1$  y  $c = -2n$ , al reemplazar en la fórmula se tiene:

$$\begin{aligned} k &= \frac{-1 \pm \sqrt{1^2 - 4 \cdot 1 \cdot (-2n)}}{2 \cdot 1} \\ k &= \frac{-1 \pm \sqrt{1 + 8n}}{2} \end{aligned}$$

Como los valores que debe tomar  $k$  tienen que ser positivos, entonces se descarta la solución negativa, por lo tanto:

$$k = \frac{-1 + \sqrt{1 + 8n}}{2}$$

Como  $k$  representa el último valor sumado en la sumatoria de Gauss, y esta trabaja sobre los números enteros positivos, entonces  $k$  debe tomar un valor entero, por lo tanto:

$$k = \left\lfloor \frac{-1 + \sqrt{1 + 8n}}{2} \right\rfloor$$

Ahora, es simplemente entrar a validar si  $\frac{k(k+1)}{2}$  es igual a  $n$  y concluir si es o no un número triangular.

En el siguiente programa, en Lenguaje C, se implementa la estrategia anterior:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    unsigned long long int n, triangular, k, kplus1;

    while(scanf("%llu", &n) && (n > 0))
    {
        k = (-1 + sqrt(1 + 8 * n)) / 2;
        kplus1 = k + 1;
        triangular = (k * kplus1) / 2;

        if(triangular == n)
            printf("YES\n");
        else
            printf("NO\n");
    }

    return 0;
}
```

Esta solución, en algunos casos, genera resultados incorrectos (en inglés el veredicto del juez en línea es `Wrong Answer`). La solución analítica es correcta, pero el programa genera en algunos casos resultados incorrectos. ¿En dónde está (o están) el (o los) errores en el programa?

El primer error se encuentra en la siguiente línea de código:

```
||     k = (-1 + sqrt(1 + 8 * n)) / 2;
```

El tipo de datos de la variable  $n$  (`unsigned long long int`) se desborda al ser multiplicado por 8, el máximo valor que toma como entrada en el reto la variable  $n$  es  $16 \cdot 10^{18}$ , como  $8 \cdot 16 \cdot 10^{18} = 128 \cdot 10^{18}$ , donde el máximo valor para el tipo de datos de la variable es del orden de  $18 \cdot 10^{18}$ .

En el siguiente programa, en Lenguaje C, se soluciona el desbordamiento de la variable  $n$  al hacerle un `casting` y llevarla al tipo de datos `double`, el cual permite almacenar números mucho más grandes.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    unsigned long long int n, triangular, k, kplus1;

    while(scanf("%llu", &n) && (n > 0))
    {
```

```

    k = (-1 + sqrt(1 + 8 * (double)n)) / 2;
    kplus1 = k + 1;
    triangular = (k * kplus1) / 2;

    if(triangular == n)
        printf("YES\n");
    else
        printf("NO\n");
}

return 0;
}

```

Esta solución sigue generando en algunos casos resultados incorrectos. Ahora, ¿en dónde está (o están) el (o los) errores en el programa?

El segundo error, se encuentra de nuevo en la misma línea de código:

```

||     k = (-1 + sqrt(1 + 8 * (double)n)) / 2;

```

En el que se debe hacer de nuevo un **casting** para transformar al tipo `unsigned long long int` en el resultado de la función `sqrt` el cual es del tipo `double`.

```

||     k = (-1 + (unsigned long long int)sqrt(1 + 8 * (double)n)) / 2;

```

En el siguiente programa, en Lenguaje C, se corrige la línea anterior, pero el programa sigue generando respuestas incorrectas para algunos valores de entrada.

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    unsigned long long int n, triangular, k, kplus1;

    while(scanf("%llu", &n) && (n > 0))
    {
        k = (-1 + (unsigned long long int)sqrt(1 + 8 * (double)n)) / 2;
        kplus1 = k + 1;
        triangular = (k * kplus1) / 2;

        if(triangular == n)
            printf("YES\n");
        else
            printf("NO\n");
    }

    return 0;
}

```

El tercer y último error, se encuentra en las líneas de código:

```

||     kplus1 = k + 1;
||     triangular = (k * kplus1) / 2;

```

Donde en el resultado intermedio de la multiplicación de  $k$  por  $k+1$  se desborda la capacidad del tipo de dato `unsigned long long int`. El último caso de prueba en los ejemplos del reto es el número 15999999994386249876, para el cual se indica que el número si es triangular.

$$\frac{5656854248 \cdot (5656854248 + 1)}{2} = 15999999994386249876$$

$k \cdot (k + 1) = 5656854248 \cdot (5656854248 + 1) = 3199999998772499752$ , con lo cual se genera el desbordamiento en el tipo de datos.

Se garantiza que  $k$  o  $k + 1$  es un número par, entonces se puede evitar el desbordamiento del tipo de datos de la siguiente forma:

Si  $k$  es par entonces:  $\frac{k \cdot (k+1)}{2} = \left(\frac{k}{2}\right) \cdot (k + 1)$ .

Si  $k + 1$  es par entonces:  $\frac{k \cdot (k+1)}{2} = k \cdot \left(\frac{k+1}{2}\right)$ .

A continuación, se encuentra la solución correcta para el reto de programación, sin problemas de desbordamientos en los tipos de datos.

### Solución en Lenguaje C del reto

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    unsigned long long int n, triangular, k, kplus1;

    while(scanf("%llu", &n) && (n > 0))
    {
        k = (-1 + (unsigned long long int)sqrt(1 + 8 * (double)n)) / 2;
        kplus1 = k + 1;

        if(k % 2 == 0)
            k /= 2;

        if(kplus1 % 2 == 0)
            kplus1 /= 2;

        triangular = k * kplus1;

        if(triangular == n)
            printf("YES\n");
        else
            printf("NO\n");
    }

    return 0;
}
```

La salida del programa anterior, corriendo los ejemplos de prueba del reto, es la siguiente:

```

1
YES
15
YES
16
NO
101
NO
1599999994386249876
YES
0

```

Figura 1.17. Salida del programa para el reto: Números triangulares.

### 1.3.2. La ladrona de libros

**Nombre original:** The Book Thief<sup>3</sup>.

**Fuente:** UTP Open 2015.

**Fecha:** 11 de Abril de 2015.

**Autor:** Hugo Humberto Morales Peña.

El 18 de Febrero de 2014, la Red Matemática propuso el siguiente reto matemático en su cuenta en Twitter (@redmatematicant): “Anita mientras leía: *La ladrona de libros* de Markus Zusak, sumaba los número de sus páginas empezando en la página 1. Obtuvo una suma igual a 9.000 y cuando terminó, se dio cuenta que le faltó por sumar el número de una página. ¿Cuál fue el número de esa página y cuantas páginas tiene el libro?”

Utilizando como punto de partida este reto, el problema que usted debe resolver es: dada como entrada un número entero positivo  $s$  ( $1 \leq s \leq 10^8$ ) que representa la suma realizada por Anita, encontrar el número de la página que hizo falta por sumar y el total de páginas que tiene el libro.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 10^6$ ), denotando el número total de casos. Cada caso de prueba es presentado en una sola línea, y contiene un número entero positivo  $s$ .

#### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir dos enteros positivos que representan el número de la página olvidada y el total de páginas del libro. Cada caso valido de la entrada debe generar una sola línea en la salida.

---

<sup>3</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/the-book-thief-1>

Ejemplo de entrada:

```
9
1
2
3
4
5
6
9000
499977
49999775
```

Ejemplo de salida:

```
2 2
1 2
3 3
2 3
1 3
4 4
45 134
523 1000
5225 10000
```

Solución ingenua en Lenguaje C del reto

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    int totalCases, idCase, s, page, gauss;

    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d", &s);
        gauss = 0;
        for(page = 1; gauss <= s; page++)
            gauss += page; /* gauss = gauss + page */

        printf("%d %d\n", gauss - s, page - 1);
    }

    return 0;
}
```

Esta solución arroja resultados correctos, pero es muy costosa en tiempo de ejecución y genera en el juez en línea un veredicto de tiempo límite excedido. Por ese motivo, el código anterior representa una *solución ingenua* del reto de programación.

## Enfoque matemático del reto

En la solución analítica del reto de *La ladrona de libros* se utiliza la solución de la sumatoria de Gauss, con la cual se plantea y resuelve una ecuación de grado 2 para determinar el número triangular más grande que es menor o igual a  $s$  (suma que obtuvo Anita).

$$\begin{aligned}\frac{p(p+1)}{2} &= s \\ p(p+1) &= 2s \\ p^2 + p &= 2s \\ p^2 + p - 2s &= 0\end{aligned}$$

Recordar que la fórmula de la cuadrática para obtener las raíces de un polinomio de grado 2 es:

$$p = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

En la ecuación cuadrática se tiene que  $a = 1$ ,  $b = 1$  y  $c = -2s$ , al reemplazar en la fórmula se tiene:

$$\begin{aligned}p &= \frac{-1 \pm \sqrt{1^2 - 4 \cdot 1 \cdot (-2s)}}{2 \cdot 1} \\ p &= \frac{-1 \pm \sqrt{1 + 8s}}{2}\end{aligned}$$

Como los valores que debe tomar  $p$  (la página) tienen que ser positivos, entonces se descarta la solución negativa, por lo tanto:

$$p = \frac{-1 + \sqrt{1 + 8s}}{2}$$

Como  $p$  representa el último valor sumado en la sumatoria de Gauss y esta trabaja sobre los números enteros positivos, entonces  $p$  debe tomar un valor entero, por lo tanto:

$$p = \left\lfloor \frac{-1 + \sqrt{1 + 8s}}{2} \right\rfloor$$

Ahora es simplemente determinar el total de páginas del libro que sería  $p + 1$  (el valor de la página obtenida más uno) y la página que dejó de sumar Anita, que sería  $\frac{(p+1) \cdot (p+2)}{2} - s$  (la suma correcta del total de páginas del libro menos la suma obtenida por ella).

## Solución en Lenguaje C del reto

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    unsigned long long int s, page, gauss;
    int totalCases, idCase;
```

```
    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%llu", &s);
        page = (-1 + (unsigned long long int)sqrt(1 + 8 * s)) / 2;
        gauss = ((page + 1) * (page + 2)) / 2;

        printf("%llu %llu\n", gauss - s, page + 1);
    }

    return 0;
}
```

La salida del programa anterior, al ejecutar los ejemplos del reto, es la siguiente:

```
9
1
2 2
2
1 2
3
3 3
4
2 3
5
1 3
6
4 4
9000
45 134
499977
523 1000
49999775
5225 10000
```

Figura 1.18. Salida del programa para el reto: La ladrona de libros.

### 1.3.3. Felipe y la secuencia

**Nombre original:** Felipe and the Sequence<sup>4</sup>.

**Fuente:** UTP Open 2017.

**Fecha:** 1ro de Abril de 2017.

**Autor:** Hugo Humberto Morales Peña.

En Febrero 19 de 2017, la Red Matemática propuso el siguiente reto matemático en su cuenta en Twitter (@redmatematicant): “Felipe, ¿cuántos términos de la siguiente secuencia de números se deben sumar para que el resultado sea igual a 200?”

$$\frac{1}{\sqrt{1} + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \frac{1}{\sqrt{4} + \sqrt{5}} + \dots = 200$$

Utilizando este reto como nuestro punto de partida, el problema que usted debe resolver ahora es: dado un número entero positivo  $S$  ( $1 \leq S \leq 10^9$ ) que representa el resultado obtenido por

<sup>4</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/felipe-and-the-sequence>

la suma de términos en la secuencia, encontrar el número  $n$  que representa el total del número de términos en la secuencia que se deben sumar.

$$\frac{1}{\sqrt{1} + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \frac{1}{\sqrt{4} + \sqrt{5}} + \cdots + \frac{1}{\sqrt{n} + \sqrt{n+1}} = S$$

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 5 \cdot 10^5$ ), el número de casos de prueba, seguido por  $t$  líneas, cada línea conteniendo un número entero positivo  $S$  ( $1 \leq S \leq 10^9$ ).

#### Formato de salida:

Para cada caso de prueba, su programa debe imprimir un número entero positivo  $n$  que representa el número total de términos en la secuencia que se suman.

#### Ejemplo de entrada:

```
1
200
```

#### Ejemplo de salida:

```
40400
```

#### Solución ingenua en Lenguaje C del reto

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    unsigned long long int n, s;
    int totalCases, idTestCase;
    double sum;

    scanf("%d", &totalCases);

    for(idTestCase = 1; idTestCase <= totalCases; idTestCase++)
    {
        scanf("%llu", &s);
        sum = 0.0;

        for(n = 1; sum <= (double)s; n++)
            sum += 1.0 / (sqrt(n) + sqrt(n + 1));

        printf("%llu\n", n - 1);
    }

    return 0;
}
```

La solución anterior es muy costosa en tiempo de ejecución, por el fragmento de código del ciclo **for**:

```
||   for(n = 1; sum <= (double)s; n++)
||       sum += 1.0 / (sqrt(n) + sqrt(n + 1));
```

Además, no siempre se generan resultados correctos por los problemas que se presentan de redondeo, al trabajar en el computador con números de punto flotante.

### Enfoque matemático del reto

La solución computacional que se va a plantear para el reto, hace uso de la racionalización de la fracción que tiene raíces cuadradas en el denominador, así como se muestra a continuación:

$$\begin{aligned} \frac{1}{\sqrt{n} + \sqrt{n+1}} &= \frac{1}{\sqrt{n} + \sqrt{n+1}} \cdot \left( \frac{\sqrt{n} - \sqrt{n+1}}{\sqrt{n} - \sqrt{n+1}} \right) \\ &= \frac{\sqrt{n} - \sqrt{n+1}}{n - \cancel{\sqrt{n} \cdot \sqrt{n+1}} + \cancel{\sqrt{n} \cdot \sqrt{n+1}} - (n+1)} \\ &= \frac{\sqrt{n} - \sqrt{n+1}}{\cancel{n} - \cancel{n} - 1} \\ &= \frac{\sqrt{n} - \sqrt{n+1}}{-1} \\ &= -\sqrt{n} + \sqrt{n+1} \end{aligned}$$

Donde tenemos que:

$$\begin{aligned} \frac{1}{\sqrt{1} + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \frac{1}{\sqrt{4} + \sqrt{5}} + \cdots + \frac{1}{\sqrt{n} + \sqrt{n+1}} &= \\ (-\sqrt{1} + \sqrt{2}) + (-\sqrt{2} + \sqrt{3}) + (-\sqrt{3} + \sqrt{4}) + (-\sqrt{4} + \sqrt{5}) + \cdots + (-\sqrt{n} + \sqrt{n+1}) &= \\ -\sqrt{1} + \cancel{\sqrt{2}} - \cancel{\sqrt{2}} + \cancel{\sqrt{3}} - \cancel{\sqrt{3}} + \cancel{\sqrt{4}} - \cancel{\sqrt{4}} + \cancel{\sqrt{5}} + \cdots - \cancel{\sqrt{n}} + \sqrt{n+1} &= \\ -\sqrt{1} + \sqrt{n+1} &= -1 + \sqrt{n+1} = \sqrt{n+1} - 1 \end{aligned}$$

Ahora tenemos que despejar  $n$  de la siguiente expresión:

$$\begin{aligned} \sqrt{n+1} - 1 &= S \\ \sqrt{n+1} &= S + 1 \\ (\sqrt{n+1})^2 &= (S+1)^2 \\ n+1 &= (S+1)^2 \\ n &= (S+1)^2 - 1 \end{aligned}$$

## Solución en Lenguaje C del reto

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    unsigned long long int n, s;
    int totalCases, idCase;

    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%llu", &s);
        n = (s + 1) * (s + 1) - 1;
        printf("%llu\n", n);
    }

    return 0;
}
```

La salida del programa anterior, ejecutando el ejemplo del reto, es la siguiente:

```
1
200
40400
```

Figura 1.19. Salida del programa para el reto: Felipe y la secuencia.

### 1.3.4. La función de Dangie

**Nombre original:** Dangie's Function<sup>5</sup>.

**Fuente:** Maratón de Programación UFPS 2021.

**Fecha:** 11 de Diciembre de 2021.

**Autor:** Hugo Humberto Morales Peña.

Normalmente, en un curso de programación de primer año en un programa académico de ingenierías o de ciencias de la computación, se le enseña a los estudiantes a construir funciones que hacen uso de los ciclos de repetición anidados como la siguiente:

```
unsigned long long DangiesFunction(int n)
{
    int i, j, k;
    unsigned long long result = 0;

    for(i = 1; i <= n - 1; i++)
    {
        for(j = i + 1; j <= n; j++)

```

<sup>5</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/dangies-function>

```
    {
        for(k = 1; k <= j; k++)
        {
            result += 1;
        }
    }

    return result;
}
```

La complejidad temporal del algoritmo anterior es de orden  $O(n^3)$ , lo que implica que con el valor de  $n = 10^6$ , el número total de operaciones realizadas por el algoritmo, para poder dar el resultado, requeriría  $10^{18}$  pasos. Como un competidor de las diferentes fases de la ICPC (International Collegiate Programming Contest, en español, Concurso Internacional Universitario de Programación), usted sabe que esa solución obtendrá el veredicto de tiempo límite excedido en la competencia. Por esta razón, se le pide proponer una solución que tenga una complejidad temporal tan cercana a  $O(1)$  como sea posible, independientemente del tamaño de  $n$ . Para ello, requerirá de toda su experiencia como un competidor de la ICPC.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 10^6$ ), el cual representa el total de casos de prueba. Luego son presentadas  $t$  líneas, cada una de ellas conteniendo un número entero positivo  $n$  ( $1 \leq n \leq 10^6$ ), para el cual se debe calcular el resultado de la función de Dangie.

#### Formato de salida:

La salida debe contener  $t$  líneas, cada una de ellas conteniendo un entero positivo largo como resultado de la función de Dangie.

#### Ejemplo de entrada:

```
7
1
5
25
52
833
5234
1000000
```

#### Ejemplo de salida:

```
0
40
5200
46852
192669568
47794715890
3333333333330000000
```

## Enfoque matemático del reto

En este reto de programación, lo que se pide implícitamente es plantear una fórmula en términos de  $n$  para calcular el resultado de la siguiente función:

```
unsigned long long DangiesFunction(int n)
{
    int i, j, k;
    unsigned long long result = 0;

    for(i = 1; i <= n - 1; i++)
    {
        for(j = i + 1; j <= n; j++)
        {
            for(k = 1; k <= j; k++)
            {
                result += 1;
            }
        }
    }

    return result;
}
```

Para lo cual, se plantea una solución equivalente por medio de sumatorias que permite obtener el mismo resultado de la función `Dangie`, en la que se utiliza el triple ciclo `for`. Para luego pasar de sumatorias triples a sumatorias dobles y, por último, pasar a sumatorias simples, como se realiza a continuación:

$$\begin{aligned}
 \text{Dangie}(n) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \sum_{k=1}^j 1 \\
 &= \sum_{j=2}^n \sum_{k=1}^j 1 + \sum_{j=3}^n \sum_{k=1}^j 1 + \dots + \sum_{j=n}^n \sum_{k=1}^j 1 \\
 &= \left( \sum_{k=1}^2 1 + \sum_{k=1}^3 1 + \dots + \sum_{k=1}^n 1 \right) + \left( \sum_{k=1}^3 1 + \sum_{k=1}^4 1 + \dots + \sum_{k=1}^n 1 \right) + \dots + \left( \sum_{k=1}^{n-1} 1 + \sum_{k=1}^n 1 \right) + \left( \sum_{k=1}^n 1 \right) \\
 &= (2 + 3 + \dots + n) + (3 + 4 + \dots + n) + \dots + ((n - 1) + n) + (n) \\
 &= \left( \sum_{i=2}^n i \right) + \left( \sum_{i=3}^n i \right) + \dots + \left( \sum_{i=n-1}^n i \right) + \left( \sum_{i=n}^n i \right) \\
 &= \left( \sum_{i=1}^n i - \sum_{i=1}^1 i \right) + \left( \sum_{i=1}^n i - \sum_{i=1}^2 i \right) + \dots + \left( \sum_{i=1}^n i - \sum_{i=1}^{n-2} i \right) + \left( \sum_{i=1}^n i - \sum_{i=1}^{n-1} i \right) \\
 &= (n - 1) \cdot \left( \sum_{i=1}^n i \right) - \left( \sum_{i=1}^1 i + \sum_{i=1}^2 i + \dots + \sum_{i=1}^{n-2} i + \sum_{i=1}^{n-1} i \right) \\
 &= (n - 1) \cdot \left( \frac{n \cdot (n + 1)}{2} \right) - \left( \frac{1 \cdot 2}{2} + \frac{2 \cdot 3}{2} + \frac{3 \cdot 4}{2} + \dots + \frac{(n - 1) \cdot (n)}{2} \right)
 \end{aligned}$$

$$\begin{aligned}
 &= \frac{(n-1) \cdot n \cdot (n+1)}{2} - \frac{1}{2} \cdot \left( \sum_{i=1}^{n-1} i \cdot (i+1) \right) \\
 &= \frac{(n-1) \cdot n \cdot (n+1)}{2} - \frac{1}{2} \cdot \left( \sum_{i=1}^{n-1} (i^2 + i) \right) \\
 &= \frac{(n-1) \cdot n \cdot (n+1)}{2} - \frac{1}{2} \cdot \left( \sum_{i=1}^{n-1} i^2 + \sum_{i=1}^{n-1} i \right) \\
 &= \frac{(n-1) \cdot n \cdot (n+1)}{2} - \frac{1}{2} \cdot \left( \frac{(n-1) \cdot n \cdot (2 \cdot (n-1) + 1)}{6} + \frac{(n-1) \cdot n}{2} \right) \\
 &= \frac{(n-1) \cdot n \cdot (n+1)}{2} - \frac{(n-1) \cdot n \cdot (2 \cdot n - 2 + 1)}{12} - \frac{(n-1) \cdot n}{4} \\
 &= \frac{6 \cdot (n-1) \cdot n \cdot (n+1)}{6 \cdot 2} - \frac{(n-1) \cdot n \cdot (2 \cdot n - 1)}{12} - \frac{3 \cdot (n-1) \cdot (n)}{3 \cdot 4} \\
 &= \frac{(n-1) \cdot n}{12} \cdot (6 \cdot (n+1) - (2n-1) - 3) \\
 &= \frac{(n-1) \cdot n}{12} \cdot (6n + 6 - 2n + 1 - 3) \\
 &= \frac{(n-1) \cdot n}{12} \cdot (4n + 4) \\
 &= \frac{(n-1) \cdot n \cdot \cancel{4} \cdot (n+1)}{\cancel{4} \cdot 3} \\
 &= \frac{(n-1) \cdot n \cdot (n+1)}{3}.
 \end{aligned}$$

$$\text{Dangie}(n) = \frac{(n-1) \cdot n \cdot (n+1)}{3}, \text{ para } n \geq 1, n \in \mathbb{Z}^+.$$

Implementar en un lenguaje de programación la fórmula en términos de  $n$  genera una solución en tiempo computacional constante, es decir, una solución que produce resultados en  $O(1)$ , donde independientemente del valor que tome la variable  $n$ , siempre se tienen que realizar de forma constante cinco operaciones (una resta, una suma, dos multiplicaciones y una división).

### Solución en Lenguaje C del reto

```

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

unsigned long long DangiesFunctionEfficient(unsigned long long n)
{

```

```

        unsigned long long result = 0;
        result = ((n - 1) * n * (n + 1)) / 3;
        return result;
    }

int main()
{
    int totalCases, idCase;
    unsigned long long n;
    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%llu", &n);
        printf("%llu\n", DangiesFunctionEfficient(n));
    }

    return 0;
}

```

La salida del programa anterior, ejecutando los ejemplos del reto, es la siguiente:

```

7
1
0
5
40
25
5200
52
46852
833
192669568
5234
47794715890
1000000
33333333333000000

```

Figura 1.20. Salida del programa para el reto: La función de Dangie.

## 1.4. Relaciones de recurrencia

A menudo, es posible encontrar relaciones entre los elementos de una sucesión. Estas relaciones se llaman relaciones de recurrencia.

Una relación de recurrencia para una sucesión  $a_0, a_1, a_2, \dots, a_n$  es una ecuación que relaciona  $a_n$  con alguno (o algunos) de sus antecesores  $a_0, a_1, a_2, \dots, a_{n-1}$  y la suma o multiplicación de alguna cantidad.

### Ejemplo 6:

En clases anteriores, cuando se trabajó el tema de “Sucesiones y Sumatorias”, se pidió que se generara una fórmula para calcular el  $n$ -ésimo término de la sucesión que tiene los primeros 10 términos siguientes: 3, 9, 15, 21, 27, 33, 39, 45, 51 y 57. La fórmula que se obtuvo en ese ejemplo fue:

$$S_1 = 3$$

$$S_n = S_{n-1} + 6, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Esta fórmula es una relación de recurrencia, con la cual se indica que el término ubicado en la posición  $n$  de la sucesión se obtiene al relacionar el término que se encuentra en la sucesión en la posición  $n - 1$  con la suma del número 6.

¿Cuántos llamados recursivos son necesarios en la fórmula anterior para calcular el elemento que se encuentra en la posición un millón de la sucesión?

Son necesarios un millón de llamados recursivos, y en cada llamado recursivo, a excepción del caso base, es realizada una suma. La fórmula recursiva o relación de recurrencia anterior es correcta para calcular el  $n$ -ésimo término de la sucesión, pero tiene un costo computacional muy alto. Por este motivo es fundamental determinar fórmulas sin recursividad, para generar el  $n$ -ésimo término de la sucesión. La solución de relaciones de recurrencia permite generar a partir de la relación de recurrencia una fórmula sin recursividad, que permite generar el mismo valor  $n$ -ésimo de la sucesión.

La fórmula  $S_n = 3 + 6 * (n - 1)$ , para  $n \in \mathbb{Z}^+$ , es la solución de la relación de recurrencia, y para calcular el elemento ubicado en la posición  $n$  de la sucesión, sólo necesita realizar tres operaciones (una resta, una multiplicación y una suma), independientemente del valor de  $n$ .

#### 1.4.1. Método de iteración

El *método de iteración* sirve para resolver relaciones de recurrencia de primer orden no homogéneas. El método consiste en comenzar en el caso base de la relación de recurrencia y utilizarlo para definir sin recursividad el caso que sigue después del caso base, y, así sucesivamente, se itera tantas veces como sea necesario, hasta lograr determinar cuál es la sumatoria o sumatorias “ocultas” que sirven para solucionar la relación de recurrencia sin utilizar recursividad.

Después de tener la sumatoria o sumatorias para la relación de recurrencia evaluada en un valor  $n$ , estas (o esta) son resueltas; de esta forma se obtiene la solución de la relación de recurrencia.

Por último, el método de demostración por inducción matemática, puede ser utilizado para ratificar o refutar que la solución de la relación de recurrencia es correcta.

#### Ejemplo 7:

Resolver la siguiente relación de recurrencia:

$$T(1) = 1$$

$$T(n) = T(n - 1) + n, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Utilizando el método de iteración, se tiene:

$$T(1) = 1$$

$$T(2) = T(1) + 2 = 1 + 2, \text{ es importante no obtener el resultado de } 1 + 2, \text{ sino, dejar}$$

indicada la suma de términos para no desaparecer la sumatoria.

$$T(3) = T(2) + 3 = 1 + 2 + 3$$

$$T(4) = T(3) + 4 = 1 + 2 + 3 + 4$$

$$T(5) = T(4) + 5 = 1 + 2 + 3 + 4 + 5$$

Se itera tantas veces como se considere necesario hasta que se identifique cual es la sumatoria oculta.

En estos momentos debe ser evidente que:

$$T(n) = 1 + 2 + 3 + 4 + \dots + n$$

De esta forma, se detecta que la sumatoria oculta que resuelve la relación de recurrencia sin recursividad es  $\sum_{i=1}^n i = 1 + 2 + \dots + n$ .

La cual tiene solución  $\frac{n(n+1)}{2}$ , por lo tanto,  $T(n) = \sum_{i=1}^n i = \frac{n(n+1)}{2}$ . De esta forma, la solución de la relación de recurrencia es  $T(n) = \frac{n(n+1)}{2}$  para  $n \geq 1$ .

Ahora, se va a utilizar el método de demostración por inducción matemática para ratificar que la solución obtenida de la relación de recurrencia es correcta.

#### Caso base $n = 1$ :

$n$  toma el valor de 1 porque este es el valor con el cuál termina la recursividad en la relación de recurrencia, o dicho en otras palabras, para  $n = 1$ , está definido el caso base de la relación de recurrencia.

$$\underbrace{T(1) = 1}_{\text{Caso base de la R.R.}} = \underbrace{\frac{1(2)}{2} = 1}_{\text{Solución de la relación de recurrencia evaluada en 1}}$$

Como se cumple la igualdad entre el caso base de la relación de recurrencia y la solución de la relación de recurrencia evaluada en  $n = 1$ , entonces la demostración continua en el caso inductivo.

#### Caso inductivo $n = k$ , $k \in \mathbb{Z}^+$ :

Se asume como cierta la solución de la relación de recurrencia evaluada en  $k$ , donde  $k \in \mathbb{Z}^+$  para  $k > 1$

$$T(k) = \frac{k(k+1)}{2}$$

#### Caso pos-inductivo $n = k + 1$ :

En este caso, se recuerda el paso recursivo de la relación de recurrencia, donde  $T(n) = T(n - 1) + n$ , reemplazando  $n$  por  $k + 1$  se tiene:

$$T(k + 1) = T(k + 1 - 1) + k + 1$$

$$T(k + 1) = T(k) + (k + 1)$$

Donde  $T(k+1)$  y  $T(k)$  se reemplazan por su equivalente en el caso inductivo evaluado en  $k+1$  y  $k$ , respectivamente:

$$\begin{aligned}\frac{(k+1)(k+1+1)}{2} &= \frac{k(k+1)}{2} + (k+1) \\ \frac{(k+1)(k+2)}{2} &= (k+1)\left[\frac{k}{2} + 1\right] \\ &= (k+1)\left[\frac{k}{2} + \frac{2}{2}\right] \\ &= (k+1)\left[\frac{k+2}{2}\right] \\ &= \frac{(k+1)(k+2)}{2}\end{aligned}$$

Como efectivamente se llegó a la igualdad, entonces se ratifica que la solución de la relación de recurrencia obtenida por el método de iteración es correcta.

### Ejemplo 8:

Resolver la siguiente relación de recurrencia:

$$T(1) = 1$$

$$T(n) = 2 \cdot T(n-1) + 1, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Por el método de iteración se tiene:

$$T(1) = 1$$

$$T(2) = 2 \cdot T(1) + 1 = 2 \cdot (1) + 1 = 2^1 + 2^0$$

$$T(3) = 2 \cdot T(2) + 1 = 2 \cdot (2^1 + 2^0) + 1 = 2^2 + 2^1 + 2^0$$

$$T(4) = 2 \cdot T(3) + 1 = 2 \cdot (2^2 + 2^1 + 2^0) + 1 = 2^3 + 2^2 + 2^1 + 2^0$$

⋮

$$T(n) = 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0$$

$$T(n) = 2^0 + 2^1 + \dots + 2^{n-1}$$

Como  $T(n) = \sum_{i=0}^{n-1} 2^i$ , entonces la solución de la relación de recurrencia es la solución de la suma de términos de la serie geométrica con primer término  $a = 2^0 = 1$ , razón constante  $r = 2$  y potencia más grande igual a  $n - 1$ ; tomando la solución de dicha sumatoria de la sección 1.1.6 Fórmulas de sumatorias útiles, se tiene que:

$$T(n) = \sum_{i=0}^{n-1} 2^i = \frac{a \cdot r^{(\text{potencia más grande})+1} - a}{r - 1} = \frac{1 \cdot 2^{(n-1)+1} - 1}{2 - 1} = 2^n - 1$$

Por lo tanto, la solución de la relación de recurrencia es  $T(n) = 2^n - 1$ , para  $n \geq 1$ .

Si se quiere ratificar o refutar la solución obtenida para la relación de recurrencia, entonces se puede utilizar el método de demostración por inducción matemática.

### Ejemplo 9:

Resolver la siguiente relación de recurrencia:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Primero, se debe hacer el cambio de variable en la relación de recurrencia:

$$T(2^0) = 1$$

$$T(2^m) = T\left(\frac{2^m}{2}\right) + 1, \text{ para } m \geq 1, m \in \mathbb{Z}^+$$

$$T(2^m) = T\left(\frac{2^m}{2^1}\right) + 1, \text{ para } m \geq 1, m \in \mathbb{Z}^+$$

$$T(2^m) = T(2^m \cdot 2^{-1}) + 1, \text{ para } m \geq 1, m \in \mathbb{Z}^+$$

$$T(2^m) = T(2^{m-1}) + 1, \text{ para } m \geq 1, m \in \mathbb{Z}^+$$

Ahora la relación de recurrencia en términos del cambio de variable es:

$$T(2^0) = 1$$

$$T(2^m) = T(2^{m-1}) + 1, \text{ para } m \geq 1, m \in \mathbb{Z}^+$$

Después de realizar el paso anterior, ya se puede aplicar el método de iteración con respecto al cambio de variable en la relación de recurrencia:

$$T(2^0) = 1$$

$$T(2^1) = T(2^0) + 1 = 1 + 1$$

$$T(2^2) = T(2^1) + 1 = 1 + 1 + 1$$

$$T(2^3) = T(2^2) + 1 = 1 + 1 + 1 + 1$$

$$T(2^4) = T(2^3) + 1 = 1 + 1 + 1 + 1 + 1$$

⋮

$$T(2^m) = m + 1$$

Por lo tanto, la solución de la relación de recurrencia en términos del cambio de variable es:

$$T(2^m) = m + 1, \text{ para } m \geq 0, m \in \mathbb{N}.$$

Por último, se debe llevar la solución de la relación de recurrencia a la variable original.

Es importante recordar que  $n = 2^m$ . Si a esta igualdad se le aplica la misma operación en ambos lados ... sigue siendo una igualdad,  $\log_2(n) = \log_2(2^m) = m$ .

Por lo tanto, la solución de la relación de recurrencia en términos de la variable original es:

$$T(n) = \log_2(n) + 1, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}$$

### Ejemplo 10:

Resolver la siguiente relación de recurrencia:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + n, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Primero, se debe hacer el cambio de variable en la relación de recurrencia:

$$T(2^0) = 1 = 2^0$$

$$T(2^m) = T(2^{m-1}) + 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Ahora, se puede aplicar el método de iteración con respecto al cambio de variable en la relación de recurrencia:

$$T(2^0) = 2^0$$

$$T(2^1) = T(2^0) + 2^1 = 2^0 + 2^1$$

$$T(2^2) = T(2^1) + 2^2 = 2^0 + 2^1 + 2^2$$

$$T(2^3) = T(2^2) + 2^3 = 2^0 + 2^1 + 2^2 + 2^3$$

⋮

$$T(2^m) = \underbrace{2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^m}_{\text{Suma de términos de una serie geométrica}}$$

con primer término  $a = 2^0 = 1$ , razón contante  $r = 2$   
y potencia más grande igual a  $m$ .

$$\begin{aligned} &= \frac{1 \cdot 2^{m+1} - 1}{2 - 1} \\ &= 2^{m+1} - 1 \end{aligned}$$

Por lo tanto, la solución de la relación de recurrencia en términos del cambio de variable es:

$$T(2^m) = 2^{m+1} - 1, \text{ para } m \geq 0, m \in \mathbb{N}.$$

Por último, se debe llevar la solución de la relación de recurrencia a la variable original.

Es importante recordar que  $n = 2^m$ . La solución de la relación de recurrencia en términos de la variable original es:

$$\begin{aligned}
 T(2^m) &= 2^{m+1} - 1 \\
 &= 2 \cdot 2^m - 1 \\
 T(n) &= 2 \cdot n - 1, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}
 \end{aligned}$$

**Ejemplo 11:**

Resolver la siguiente relación de recurrencia:

$$\begin{aligned}
 T(1) &= 1 \\
 T(n) &= 2 \cdot T\left(\frac{n}{2}\right) + 1, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+
 \end{aligned}$$

Primero, se debe hacer el cambio de variable en la relación de recurrencia:

$$\begin{aligned}
 T(2^0) &= 1 = 2^0 \\
 T(2^m) &= 2 \cdot T(2^{m-1}) + 1, m \geq 1, m \in \mathbb{Z}^+
 \end{aligned}$$

Ahora, se puede aplicar el método de iteración con respecto al cambio de variable en la relación de recurrencia:

$$\begin{aligned}
 T(2^0) &= 1 \\
 T(2^1) &= 2 \cdot T(2^0) + 1 = 2 \cdot [1] + 1 = 2^1 + 2^0 \\
 T(2^2) &= 2 \cdot T(2^1) + 1 = 2 \cdot [2^1 + 2^0] + 1 = 2^2 + 2^1 + 2^0 \\
 T(2^3) &= 2 \cdot T(2^2) + 1 = 2 \cdot [2^2 + 2^1 + 2^0] + 1 = 2^3 + 2^2 + 2^1 + 2^0 \\
 T(2^4) &= 2 \cdot T(2^3) + 1 = 2 \cdot [2^3 + 2^2 + 2^1 + 2^0] + 1 = 2^4 + 2^3 + 2^2 + 2^1 + 2^0 \\
 &\vdots \\
 T(2^m) &= 2^m + \dots + 2^3 + 2^2 + 2^1 + 2^0
 \end{aligned}$$

$$T(2^m) = \underbrace{2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^m}$$

Suma de términos de una serie geométrica

con primer término  $a = 2^0 = 1$ , razón contante  $r = 2$   
y potencia más grande igual a  $m$ .

$$\begin{aligned}
 &= \frac{1 \cdot 2^{m+1} - 1}{2 - 1} \\
 &= 2^{m+1} - 1
 \end{aligned}$$

Por lo tanto, la solución de la relación de recurrencia en términos del cambio de variable es:

$$T(2^m) = 2^{m+1} - 1, \text{ para } m \geq 0, m \in \mathbb{N}.$$

Por último, se debe llevar la solución de la relación de recurrencia a la variable original.

Es importante recordar que  $n = 2^m$ . La solución de la relación de recurrencia en términos de la variable original es:

$$T(2^m) = 2^{m+1} - 1$$

$$= 2 \cdot 2^m - 1$$

$$T(n) = 2 \cdot n - 1, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}$$

### Ejemplo 12:

Resolver la siguiente relación de recurrencia:

$$T(1) = 1$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Primero, se debe hacer el cambio de variable en la relación de recurrencia:

$$T(2^0) = 1 = 2^0$$

$$T(2^m) = 2 \cdot T(2^{m-1}) + 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Ahora, se puede aplicar el método de iteración con respecto al cambio de variable en la relación de recurrencia:

$$T(2^0) = 2^0$$

$$T(2^1) = 2 \cdot T(2^0) + 2^1 = 2 \cdot [2^0] + 2^1 = 2^1 + 2^1$$

$$T(2^2) = 2 \cdot T(2^1) + 2^2 = 2 \cdot [2^1 + 2^1] + 2^2 = 2^2 + 2^2 + 2^2$$

$$T(2^3) = 2 \cdot T(2^2) + 2^3 = 2 \cdot [2^2 + 2^2 + 2^2] + 2^3 = 2^3 + 2^3 + 2^3 + 2^3$$

⋮

$$T(2^m) = \underbrace{2^m + 2^m + 2^m + \dots + 2^m}_{m+1 \text{ veces el } 2^m}$$

$$T(2^m) = (m+1) \cdot 2^m$$

Por lo tanto, la solución de la relación de recurrencia en términos del cambio de variable es:

$$T(2^m) = 2^m \cdot (m+1), \text{ para } m \geq 0, m \in \mathbb{N}.$$

Por último, se debe llevar la solución de la relación de recurrencia a la variable original.

Es importante recordar que  $n = 2^m$ . Si a esta igualdad se le aplica la misma operación en ambos lados, sigue siendo una igualdad!,  $\log_2(n) = \log_2(2^m) = m$ .

Por lo tanto, la solución de la relación de recurrencia en términos de la variable original es:

$$T(n) = n \cdot (\log_2(n) + 1),$$

$T(n) = n \cdot \log_2(n) + n$ , para  $n = 2^m$ ,  $m \geq 0$ ,  $m \in \mathbb{N}$

## 1.5. Retos de programación propuestos

En esta sección, se propone una lista de retos de programación que se pueden resolver con el uso de las herramientas matemáticas trabajadas en este capítulo:

- La sucesión de Humbertov Moralov<sup>6</sup>
- ¿Cuántos números triangulares?
- Dora la exploradora I<sup>7</sup>
- Dora la exploradora II
- Humbertov y el espiral triangular I<sup>8</sup>
- Humbertov y el espiral triangular II
- Desviación estándar
- La función de Johann
- Triángulos internos
- ¿Cuántos triángulos equiláteros?

---

<sup>6</sup>Este reto de programación tomó como inspiración el reto “6.6.7 Sucesión autodescriptiva” del capítulo 6 del libro *Desafíos de Programación* [SR08]

<sup>7</sup>Las versiones I y II de este reto fueron inspiradas en el ejemplo 4 y en los ejercicios 9 y 10 de la sección “1.2 Conjuntos efectivamente enumerables” del libro *Elementos de Lógica y Calculabilidad* [Cai89] y en el reto “12.6.1 Una hormiga en un tablero de ajedrez” del capítulo 12 del libro *Desafíos de Programación* [SR08]

<sup>8</sup>Las versiones I y II de este reto fueron inspiradas en el reto “12.6.4 La abeja Maya” del capítulo 12 del libro *Desafíos de Programación* [SR08]

### 1.5.1. La sucesión de Humbertov Moralov

**Nombre original:** Humbertov Moralov's Sequence<sup>9</sup>.

**Fuente:** UTP Open 2013.

**Fecha:** 4 de Mayo de 2013.

**Autor:** Hugo Humberto Morales Peña.

La sucesión de Humbertov Moralov (HM) se apoya en la sucesión de los números enteros impares positivos  $I(n)$ :

$n$	1	2	3	4	5	6	7	8	9	...
$I(n)$	1	3	5	7	9	11	13	15	17	...

En la sucesión de Humbertov Moralov el valor  $n$  está  $I(n)$  veces, de esta forma se tienen los primeros 16 términos en dicha sucesión:

$n$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
$I(n)$	1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	...
$HM(n)$	1	2	2	2	3	3	3	3	3	4	4	4	4	4	4	4	...

En este problema se debe escribir un programa que calcule el valor que está en la posición  $n$  en la sucesión de Humbertov Moralov (es decir, que calcule el  $HM(n)$ ).

#### Formato de entrada:

La entrada puede contener varios casos de prueba. Cada caso de prueba se presenta en una línea independiente y contiene un entero  $n$  ( $1 \leq n \leq 2 \cdot 10^9$ ). La entrada finaliza con un caso de prueba en el que  $n$  tiene el valor de 0, caso que no debe ser procesado.

#### Formato de salida:

Por cada caso de prueba de la entrada, se debe imprimir el valor de  $HM(n)$  en una línea independiente.

#### Ejemplo de entrada:

```
100
9999
123456
1000000000
0
```

#### Ejemplo de salida:

```
10
100
352
31623
```

<sup>9</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/humbertov-moralovs-sequence>

### 1.5.2. ¿Cuántos números triangulares?

**Nombre original:** How Many Triangular Numbers?<sup>10</sup>.

**Fuente:** UCO<sup>11</sup> - Code Fortress 2.

**Fecha:** 2 de Agosto de 2014.

**Autor:** Hugo Humberto Morales Peña.

Los *números triangulares* son todos aquellos números enteros positivos que representan una cantidad de asteriscos que pueden conformar un triángulo compacto con la misma cantidad de asteriscos en cada uno de sus tres lados.

Los primeros cinco números triangulares son presentados en la Figura 1.21.

1	3	6	10	15
*	* *	* * *	* * * *	* * * * *

Figura 1.21. Primeros cinco números triangulares.

Dados dos números enteros positivos largos  $a$  y  $b$ , calcular cuántos números triangulares existen en el intervalo cerrado  $[a, b]$ .

#### Formato de entrada:

La entrada puede contener varios casos de prueba. Cada caso de prueba se presenta en una línea independiente y contiene dos números enteros positivos largos  $a$  y  $b$  ( $1 \leq a \leq b \leq 2 \cdot 10^{12}$ ). La entrada finaliza con un caso de prueba en el que tanto  $a$  y  $b$  tiene el valor de 0, caso que no debe ser procesado.

#### Formato de salida:

Por cada caso de prueba, imprimir una única línea con la cantidad de números triangulares  $t_i$  que cumplan que  $a \leq t_i \leq b$ .

#### Ejemplo de entrada:

```
1 15
2 15
1 14
2 14
5 5
6 6
0 0
```

<sup>10</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/how-many-triangular-numbers>

<sup>11</sup>Universidad Católica de Oriente, Rionegro, Antioquia, Colombia.

**Ejemplo de salida:**

5  
4  
4  
3  
0  
1

### 1.5.3. Dora la exploradora I

**Nombre original:** Dora the Explorer I<sup>12</sup>.

**Fuente:** UTP Open 2012.

**Fecha:** 28 de Abril de 2012.

**Autor:** Hugo Humberto Morales Peña.

Un día, una hormiga llamada Dora la Exploradora llegó a un tablero triangular de  $d$  diagonales. Como quería explorar todas las casillas del tablero, comenzó a caminar por el mismo desde la diagonal que tiene una sola casilla.

Dora la Exploradora comenzó en la casilla (1, 1). En primer lugar, dio un paso adelante quedando en la casilla (1, 2), luego, descendió por la diagonal quedando en la casilla (2, 1), después, dio un paso a la derecha quedando en la casilla (3, 1), posteriormente, ascendió por la diagonal recorriendo las casillas (2, 2) y (1, 3). Cada vez añadía una nueva diagonal al recorrido, en ascenso o descenso, dependiendo de la primera casilla a la que llegara de la diagonal.

Por ejemplo, en 15 pasos hizo el recorrido de la Figura 1.22 en el tablero triangular, donde el número de cada casilla indica el orden en que la visitó.

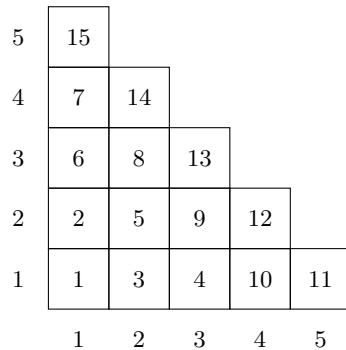


Figura 1.22. Primeros 15 paso de Dora la Exploradora en el tablero triangular.

En el décimo paso, la colocó en la casilla (4, 1), mientras que en el paso número 15 lo hizo en la casilla (1, 5).

La tarea consiste en determinar la ubicación de Dora la Exploradora en el tablero triangular para un paso dado, asumiendo que el tablero triangular es lo suficientemente grande como para admitir cualquier paso.

#### Formato de entrada:

Cada línea de la entrada consta de un número entero positivo  $n$ , el cual indica el número del paso, donde ( $1 \leq n \leq 2 \cdot 10^{18}$ ). La entrada finaliza con una línea que contiene un 0, para el cual el programa no debe hacer nada, simplemente finalizar.

<sup>12</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/dora-the-explorer-1-1>

**Formato de salida:**

Por cada caso de entrada, imprimir una línea con dos números enteros positivos  $x$  y que indiquen los valores de la columna y de la fila, respectivamente. Entre ellos debe haber un único espacio en blanco.

**Ejemplo de entrada:**

```
1
6
10
11
15
16
16
17
0
```

**Ejemplo de salida:**

```
1 1
1 3
4 1
5 1
1 5
1 6
2 5
```

### 1.5.4. Dora la exploradora II

**Nombre original:** Dora the Explorer II<sup>13</sup>.

**Fuente:** Maratón de Programación UFPS<sup>14</sup> 2019.

**Fecha:** 8 de Junio de 2019.

**Autor:** Hugo Humberto Morales Peña.

Un día, una hormiga llamada Dora la Exploradora llegó a un tablero triangular de  $d$  diagonales. Como quería explorar todas las casillas del tablero, comenzó a caminar por el mismo desde la diagonal que tiene una sola casilla.

Dora la Exploradora comenzó en la casilla (1, 1). En primer lugar, dio un paso adelante quedando en la casilla (1, 2), luego, descendió por la diagonal quedando en la casilla (2, 1), después, dio un paso a la derecha quedando en la casilla (3, 1), posteriormente, ascendió por la diagonal recorriendo las casillas (2, 2) y (1, 3). Cada vez añadía una nueva diagonal al recorrido, en ascenso o en descenso, dependiendo de la primera casilla a la que llegara de la diagonal.

Por ejemplo, en 15 pasos hizo el recorrido de la Figura 1.23 en el tablero triangular, donde el número de cada casilla indica el orden en que la visitó.

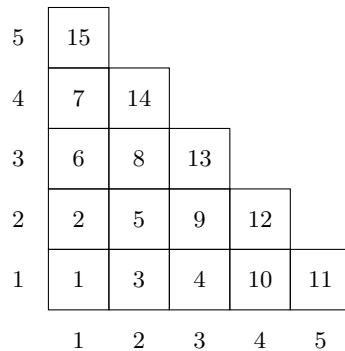


Figura 1.23. Primeros 15 paso de Dora la Exploradora en el tablero triangular.

En el décimo paso, la colocó en la casilla (4, 1), mientras que el paso número 15 lo hizo en la casilla (1, 5).

Ahora, la tarea consiste en determinar la cantidad de pasos que debe dar Dora la Exploradora en su recorrido para alcanzar la casilla  $(x, y)$  del tablero triangular. Se debe asumir que se tiene que dar un paso para alcanzar la casilla (1, 1), el cual es el punto de partida en el tablero triangular. Adicionalmente, se debe considerar que el tablero triangular es lo suficientemente grande como para admitir las coordenadas de cualquier casilla.

<sup>13</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/dora-the-explorer-ii>

<sup>14</sup>Universidad Francisco de Paula Santander, Cúcuta, Norte de Santander, Colombia.

**Formato de entrada:**

La entrada del problema consiste de múltiples casos de prueba. Cada caso consiste de una línea que contiene un par de números enteros positivos  $x$  y, que indican las coordenadas de la casilla con el valor de columna y fila respectivamente. Entre ellos debe haber un único espacio en blanco. Los valores de la columna y de la fila están en el rango  $1 \leq x, y \leq 2 \cdot 10^9$ . La entrada finaliza con una línea que contiene 0 0, para la cual el programa no debe hacer nada, simplemente finalizar.

**Formato de salida:**

Por cada caso de entrada, imprimir una linea con el numero entero positivo  $n$  que indique la cantidad de pasos que debe realizar Dora la Exploradora en su recorrido para alcanzar la casilla.

**Ejemplo de entrada:**

```
1 1
1 3
4 1
5 1
1 5
1 6
2 5
0 0
```

**Ejemplo de salida:**

```
1
6
10
11
15
16
17
```

### 1.5.5. Humbertov y el espiral triangular I

**Nombre original:** Humbertov and the Triangular Spiral<sup>15</sup>.

**Fuente:** UTP Open 2017.

**Fecha:** 1ro de Abril de 2017.

**Autor:** Hugo Humberto Morales Peña.

En días pasados, el profesor Humbertov Moralov estuvo enfermo, tenía fiebre y, cuando se acostó a dormir, comenzó a tener un sueño delirante en el que se repetía y se repetía el dibujo de un espiral triangular que comenzaba en el origen del plano cartesiano (coordenada  $(0, 0)$ ), y que generaba una nueva espiral triangular cada vez más grande al unir punto de coordenadas enteras sobre el plano cartesiano. Para mayor claridad, el espiral triangular se presenta en la Figura 1.24.

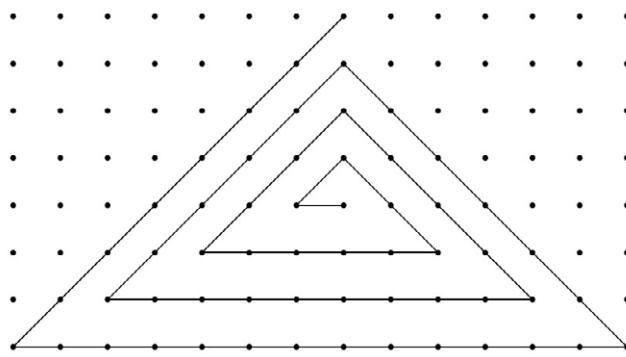


Figura 1.24. Espiral triangular.

Fue tan perturbador el sueño, y se repitió tantas veces, que cuando el profesor Moralov se despertó, se acordaba perfectamente del espiral triangular y por eso hizo el gráfico dispuesto en la Figura 1.24.

En el sueño, el profesor Moralov estaba perturbado e intrigado, se cuestionaba si todas las coordenadas enteras se podían alcanzar en algún momento en el espiral triangular, y si esto fuera cierto, entonces, ¿en cuál coordenada en el plano cartesiano estaría el punto que se alcanza en el movimiento  $n$  cuando se comienza a dibujar desde el origen de coordenadas el espiral triangular?. La primera duda fue inmediatamente resuelta cuando el profesor hizo el gráfico; todos los puntos (coordenadas enteras) del plano cartesiano se pueden alcanzar con alguno de los espirales triangulares. Ahora, el profesor Moralov necesita de tu ayuda para que le indiques la coordenada del plano cartesiano del  $n$ -ésimo punto que se alcanza al dibujar el espiral triangular.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 5 \cdot 10^5$ ), que representa el número de casos de prueba, seguido por  $t$  líneas, cada una conteniendo un número entero positivo  $n$  ( $1 \leq n \leq 10^{12}$ ).

---

<sup>15</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/triangular-spiral>

**Formato de salida:**

Para cada caso de prueba, se debe imprimir una sola línea que contenga dos números enteros, separados por un espacio, denotando las coordenadas  $x$   $y$  en el plano cartesiano del punto  $n$  en el espiral triangular.

**Ejemplo de entrada:**

```
15
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

**Ejemplo de salida:**

```
0 0
-1 0
0 1
1 0
2 -1
1 -1
0 -1
-1 -1
-2 -1
-3 -1
-2 0
-1 1
0 2
1 1
2 0
```

### 1.5.6. Humbertov y el espiral triangular II

**Nombre original:** Dimitrov and the Triangular Spiral<sup>16</sup>.

**Fuente:** ICPC Bolivia 2019.

**Fecha:** 28 de Septiembre de 2019.

**Autor:** Hugo Humberto Morales Peña.

En días pasados, el profesor Humbertov Moralov estuvo enfermo, tenía fiebre y, cuando se acostó a dormir, comenzó a tener un sueño delirante en el que se repetía y se repetía el dibujo de un espiral triangular que comenzaba en el origen del plano cartesiano (coordenada  $(0, 0)$ ), y que generaba una nueva espiral triangular cada vez más grande al unir punto de coordenadas enteras sobre el plano cartesiano. Para mayor claridad, el espiral triangular se presenta en la Figura 1.25.

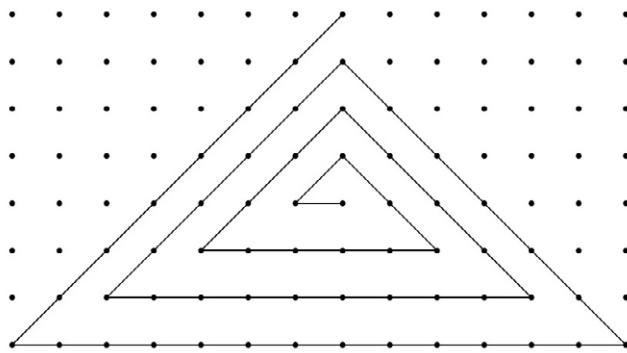


Figura 1.25. Espiral triangular.

Fue tan perturbador el sueño, y se repitió tantas veces, que cuando el profesor Moralov se despertó, se acordaba perfectamente del espiral triangular, y por eso hizo el gráfico dispuesto en la Figura 1.25.

En el sueño, el profesor Moralov estaba perturbado e intrigado, se cuestionaba si todas las coordenadas enteras se podían alcanzar en algún momento en el espiral triangular, y si esto fuera cierto, entonces, ¿cuál sería el valor del movimiento  $n$  para poder alcanzar una coordenada  $(x, y)$  en el plano cartesiano sobre el dibujo del espiral triangular?. La primera duda fue inmediatamente resuelta cuando el profesor hizo el gráfico; todos los puntos (coordenadas enteras) del plano cartesiano se pueden alcanzar en alguno de los espirales triangulares. Ahora, el profesor Moralov necesita de tu ayuda para que le indiques el valor del movimiento  $n$  que se necesita para poder alcanzar un punto específico en el plano cartesiano al dibujar el espiral triangular.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 5 \cdot 10^5$ ), que representa el número de casos de prueba, seguido por  $t$  líneas, cada una conteniendo dos números enteros

<sup>16</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/humbertov-and-the-triangular-spiral-ii>

separados por un espacio, denotando las coordenadas  $x$  y  $y$  en el sistema de coordenadas cartesianas ( $-10^8 \leq x, y \leq 10^8$ ).

**Formato de salida:**

Para cada caso de prueba, se debe imprimir una sola línea que contenga un número entero, denotando el movimiento  $n$  en el que se alcanza en el espiral triangular el punto de coordenadas ( $x, y$ ) en el plano cartesiano.

**Ejemplo de entrada:**

```
15
0 0
-1 0
0 1
1 0
2 -1
1 -1
0 -1
-1 -1
-2 -1
-3 -1
-2 0
-1 1
0 2
1 1
2 0
```

**Ejemplo de salida:**

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

### 1.5.7. Desviación estándar

**Nombre original:** Standard Deviation<sup>17</sup>.

**Fuente:** UTP Open 2016.

**Fecha:** 9 de Abril de 2016.

**Autor:** Hugo Humberto Morales Peña.

En matemáticas, la *desviación estándar* de un conjunto de  $n$  números enteros es definida como:

$$S = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n - 1}}$$

Donde  $\bar{x}$  es el promedio del conjunto de  $n$  números enteros para los cuales la desviación estándar es calculada. El promedio es calculado como:

$$\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^n x_i$$

La tarea es calcular, de una forma eficiente, la desviación estándar de los primeros  $n$  números enteros positivos impares.

#### Formato de entrada:

Hay múltiples casos de prueba en la entrada. Cada caso de prueba consta de una sola línea que contiene un solo número entero positivo  $n$  ( $2 \leq n \leq 10^6$ ), el cual indica la posición en el conjunto de los números enteros positivos impares (comenzando desde uno) hasta la cual se debe calcular la desviación estándar. La entrada finaliza con un valor 0, para el cual no se debe generar ninguna respuesta.

#### Formato de salida:

Para cada caso de prueba, se debe imprimir una sola línea que contenga un número en punto flotante que represente la desviación estándar de los primeros  $n$  números enteros positivos impares. Se permite un valor absoluto en el error de precisión de la respuesta de máximo  $10^{-6}$ .

#### Ejemplo de entrada:

```
10
100
1000
10000
100000
1000000
0
```

---

<sup>17</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/standard-deviation>

**Ejemplo de salida:**

6.055301  
58.022984  
577.638872  
5773.791360  
57735.315593  
577350.557865

### 1.5.8. La función de Johann

**Nombre original:** Johann's Function<sup>18</sup>.

**Fuente:** 4<sup>a</sup> Fecha de la ICPC Centroamérica 2020.

**Fecha:** 30 de Enero de 2021.

**Autor:** Hugo Humberto Morales Peña.

Normalmente, en un curso de programación de primer año en un programa académico de ingenierías o de ciencias de la computación, se le enseña a los estudiantes a construir funciones que hacen uso de los ciclos de repetición anidados, como la siguiente:

```
unsigned long long JohannsFunction(int n)
{
    int i, j;
    unsigned long long result = 0;

    for(i = 1; i <= n; i++)
    {
        for(j = 1; j <= i; j++)
        {
            result += j;
        }
    }

    return result;
}
```

La complejidad temporal del algoritmo pertenece a  $O(n^2)$ , lo que implica que con el valor de  $n = 10^6$ , el número total de operaciones realizadas por el algoritmo, para poder dar el resultado, requeriría  $10^{12}$  pasos. Como un competidor de las diferentes fases de la ICPC (International Collegiate Programming Contest, o su traducción al lenguaje español, Competición Internacional Universitaria de Programación), usted sabe que esa solución obtendrá el veredicto de tiempo límite excedido en la competencia. Por esta razón, se le pide proponer una solución que tenga una complejidad temporal tan cercana a  $O(1)$  como sea posible, independientemente del tamaño de  $n$ . Para ello requerirá de toda su experiencia como un competidor de la ICPC.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 10^6$ ), el cual representa el total de casos de prueba. Luego son presentadas  $t$  líneas, cada una de ellas conteniendo un número entero positivo  $n$  ( $1 \leq n \leq 10^6$ ) para el cual se debe calcular el resultado de la función de Johann.

#### Formato de salida:

La salida debe contener  $t$  líneas, cada una de ellas conteniendo un entero positivo largo como resultado de la función de Johann.

---

<sup>18</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/johanns-function>

**Ejemplo de entrada:**

```
7
1
10
100
1000
10000
100000
1000000
```

**Ejemplo de salida:**

```
1
220
171700
167167000
166716670000
166671666700000
166667166667000000
```

### 1.5.9. Triángulos internos

**Nombre Original:** Internal Triangles<sup>19</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Gabriel Gutiérrez Tamayo.

Dado el número de lados de un polígono regular convexo, determine cuántos triángulos distintos pueden formarse seleccionando tres de sus vértices (ver Figura 1.26).

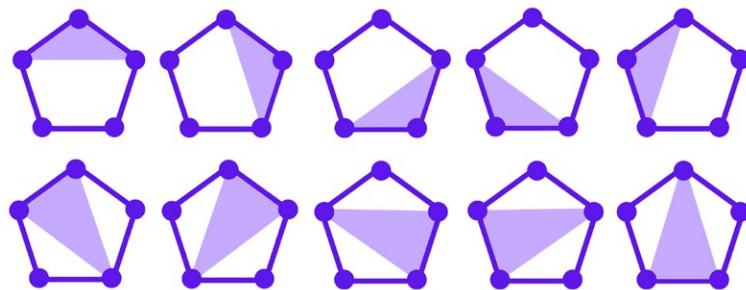


Figura 1.26. Todos los posibles triángulos del polígono convexo regular de 5 lados.

Un polígono es *convexo* si todos sus ángulos internos son menores de 180 grados.

Un polígono es *regular* si todos sus lados y ángulos internos son iguales.

Dos triángulos se consideran distintos si al menos uno de los vértices de un triángulo no pertenece al otro.

#### Formato de Entrada:

La primera línea contiene un entero  $q$  ( $1 \leq q \leq 10^5$ ), que indica el número de consultas. Cada una de las siguientes  $q$  líneas contiene un entero  $n$  ( $3 \leq n \leq 10^{18}$ ).

#### Formato de Salida:

Para cada consulta, imprimir una línea conteniendo la respuesta correspondiente. Dado que la respuesta a cada consulta puede ser muy grande, imprimirla módulo  $10^9 + 7$ .

#### Ejemplo de Entrada:

```
4
3
4
5
7
```

<sup>19</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/internal-triangles>

**Ejemplo de Salida:**

1  
4  
10  
35

### 1.5.10. ¿Cuántos triángulos equiláteros?

**Nombre Original:** Kcuadrado<sup>20</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Gabriel Gutiérrez Tamayo.

Dado un triángulo equilátero de lado  $k$ , calcular el número máximo de triángulos equiláteros de lado 1, que se pueden colocar dentro de él, sin superponerse (ver Figura 1.27).

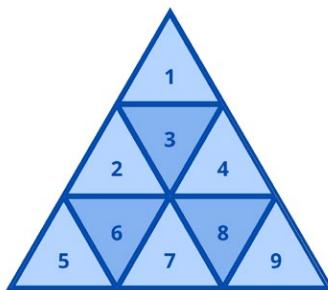


Figura 1.27. Solución óptima para un triángulo equilátero con lados de longitud 3.

Un *triángulo equilátero* es un triángulo en el que todos sus lados tienen la misma longitud.

#### Formato de Entrada:

La primera línea contiene un entero  $q$  ( $1 \leq q \leq 10^6$ ), que indica el número de consultas. Cada una de las siguientes  $q$  líneas contiene un entero  $k$  ( $1 \leq k \leq 10^{18}$ ).

#### Formato de Salida:

Para cada consulta, imprima una línea conteniendo la respuesta correspondiente. Dado que la respuesta a cada consulta puede ser muy grande, imprimirla módulo  $10^{18} + 3$ .

#### Ejemplo de Entrada:

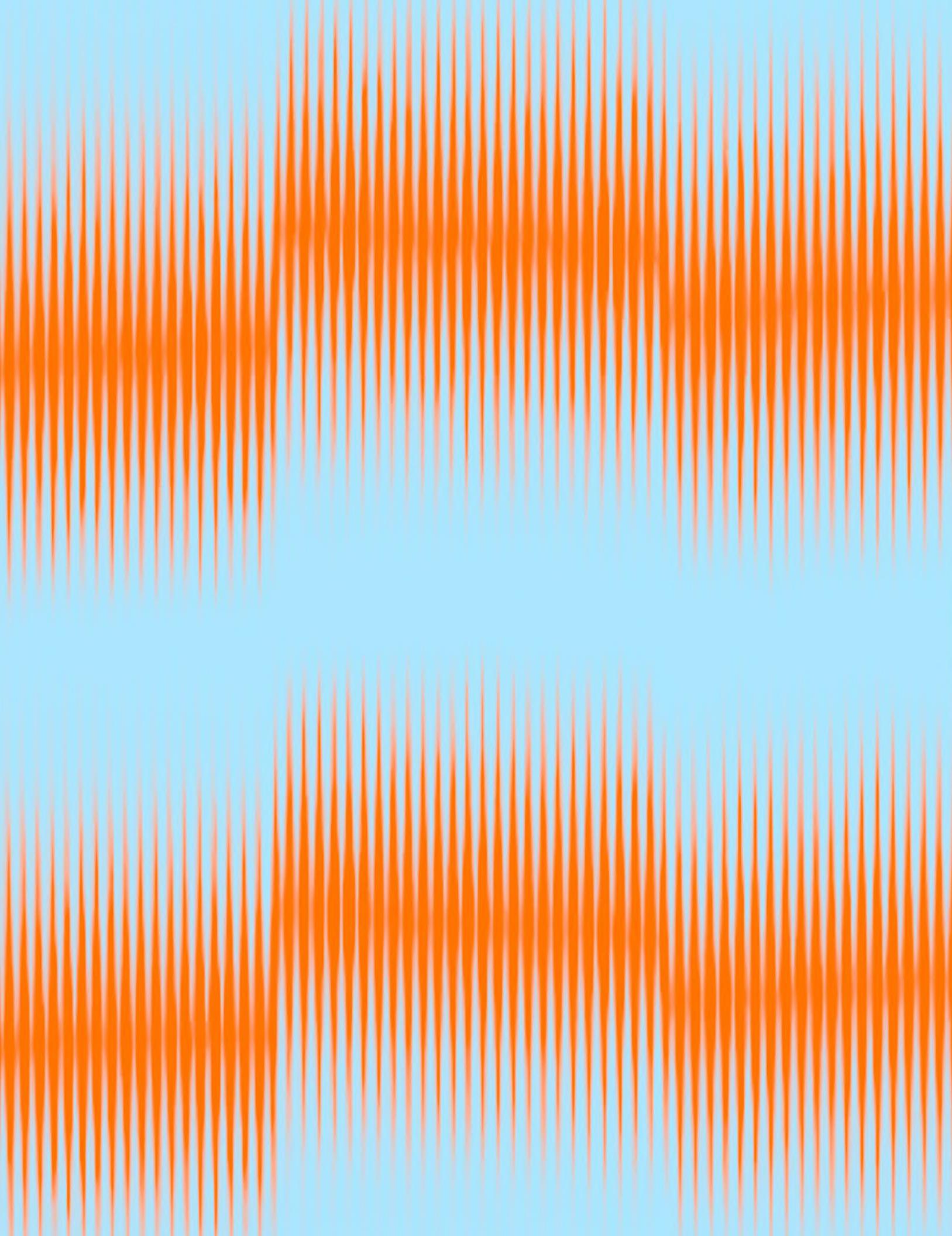
```
3
1
2
3
```

#### Ejemplo de Salida:

```
1
4
9
```

---

<sup>20</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/k-squared>



## **Capítulo 2**

# **Complejidad computacional**



En este capítulo se tratan las notaciones computacionales  $O$ ,  $\Omega$  y  $\Theta$ , las cuales son utilizadas para determinar las funciones de costo del peor de los casos, del mejor de los casos y del caso promedio de un algoritmo.

Así mismo, se presentan los algoritmos de búsqueda binaria y de ordenamiento por mezclas que pertenecen a la estrategia de divide y vencerás, donde se hacen sus respectivos análisis de la complejidad computacional al trabajar en conjunto los temas de relaciones de recurrencia y de notaciones computacionales.

El capítulo continúa con el análisis, diseño y solución (con eficiencia computacional en tiempo de ejecución y almacenamiento en espacio) de un par de retos de programación, al utilizar en conjunto los algoritmos de ordenamiento y búsqueda binaria.

Para finalizar, se proponen 8 retos de programación para que se planteen sus respectivas soluciones utilizando los algoritmos de ordenamiento y/o búsqueda binaria. Dichas soluciones se pueden validar en el juez en línea HackerRank<sup>1</sup>.

## 2.1. Notaciones computacionales ( $O$ , $\Omega$ , $\Theta$ )

En esta sección se toman como punto de partida las secciones “2.2 Crecimiento de funciones” y “2.3 Complejidad de algoritmos” del libro *Matemáticas Discretas y sus Aplicaciones* [Ros04].

El tiempo requerido para solucionar un problema no depende solamente del número de operaciones que realiza, el tiempo de ejecución también depende del software y hardware usado para ejecutar el programa en el que se implementa el algoritmo. Sin embargo, cuando se cambia el software o hardware, se debe aproximar el tiempo requerido para solucionar un problema de tamaño  $n$ , multiplicando el tiempo previo por una constante.

El trabajo con notaciones computacionales (ya sea  $O$ ,  $\Omega$  o  $\Theta$ ) permite estimar el crecimiento de la función de costo de un algoritmo, dicho en otras palabras, permite calcular la complejidad computacional de un algoritmo sin necesidad de preocuparse por el software o hardware que se utilice para implementarlo.

---

<sup>1</sup><https://www.hackerrank.com/data-structure-utp>

Se asumirá que las diferentes operaciones (como las de suma, multiplicación, división, resta, asignación, comparación, etc.) usadas en un algoritmo toman el mismo tiempo, lo cual simplifica el análisis de la complejidad del algoritmo considerablemente.

### 2.1.1. Notación O

La *notación O* es usada para estimar el número máximo de operaciones que un algoritmo realiza a partir de sus datos de entrada. Esta notación permite calcular la cota superior para una función  $f(n)$ , para valores de  $n \geq k$ .  $f(n)$  es la función de costo del algoritmo, función que representa la cantidad de operaciones que realiza el algoritmo para los datos con los que trabaja.

#### Definición:

Sean  $f$  y  $g$  funciones desde el conjunto de los números reales al conjunto de los números reales. Se dice que  $f(n)$  es  $O(g(n))$  si hay constantes positivas  $c$  y  $k$  tales que:

$$|f(n)| \leq c \cdot |g(n)|, \text{ donde } n \geq k.$$

$k$  es un número natural a partir del cual se comienza a cumplir el orden  $O(g(n))$  y  $c$  es una constante que pertenece a los números reales positivos.

**Nota 1:** Las constantes  $c$  y  $k$  son llamadas testigos para que la función  $f(n)$  sea  $O(g(n))$ .

**Nota 2:** No hay un único par de testigos  $c$  y  $k$  para la función  $f(n)$ , hay infinitos testigos y donde lo único que importa es encontrar al menos un par de estos.

#### Ejemplo 1:

¿Cuál es la notación  $O$  de  $f(n) = 1 + 2 + 3 + \dots + n$ , para  $n \geq 1$ ,  $n \in \mathbb{Z}^+$ ?

Cada uno de los términos que se están sumando es menor o igual a  $n$ , por este motivo es valida la siguiente forma de aplicar la notación  $O$ :

$$f(n) = 1 + 2 + 3 + \dots + n \leq \underbrace{n + n + n + \dots + n}_{n \text{ veces}} = n \cdot n = n^2 = \underbrace{1}_c \cdot \underbrace{n^2}_{g(n)}$$

Con lo anterior, se tiene que  $f(n) = 1 + 2 + 3 + \dots + n$  es  $O(n^2)$ , donde los testigos son  $c = 1$  y  $k = 1$ .

La forma ideal de aplicar la notación  $O$  es a partir de la solución de la suma de términos, donde tenemos:

$$f(n) = 1 + 2 + 3 + \dots + n = \frac{n \cdot (n + 1)}{2} = \frac{n^2}{2} + \frac{n}{2} \leq \frac{n^2}{2} + \frac{n^2}{2} = \underbrace{\frac{1}{2}}_c \cdot \underbrace{n^2}_{g(n)}$$

De esta forma,  $f(n) = 1 + 2 + \dots + n$  es  $O(n^2)$ , donde los testigos son  $c = 1$  y  $k = 1$ .

En este ejemplo se obtiene la misma cota superior para la función  $f(n)$  independientemente de la variante que se puede aplicar para obtener ésta, ya sea a partir del reemplazo de uno a uno de los términos que se están sumando por el más grande, o a partir de la fórmula que es la solución de la sumatoria de términos. Pero esto no siempre será así, en el Ejemplo 2 las notaciones  $O$  que se obtienen no son las mismas.

### Ejemplo 2:

¿Cuál es la notación  $O$  para  $f(n) = 2^0 + 2^1 + 2^2 + \dots + 2^n$ , para  $n \geq 0$ ,  $n \in \mathbb{N}$ ?

El término  $2^n$  es el más grande de los que se están sumando. Al aplicar la notación  $O$  término a término en la función  $f(n)$  con respecto a éste valor, tenemos:

$$\begin{aligned} \underbrace{2^0 + 2^1 + 2^2 + \dots + 2^n}_{f(n)} &\leq \underbrace{2^n + 2^n + 2^n + \dots + 2^n}_{(n+1) \text{ veces}} = (n+1) \cdot 2^n = n \cdot 2^n + 2^n \\ &\leq n \cdot 2^n + n \cdot 2^n = \underbrace{2}_c \cdot \underbrace{n \cdot 2^n}_{g(n)} \end{aligned}$$

Con el análisis anterior se tiene que  $f(n) = 2^0 + 2^1 + \dots + 2^n$  es  $O(n \cdot 2^n)$  con  $c = 2$  y  $k = 1$  como testigos. Es importante observar que en este caso el testigo  $k = 0$  no sirve porque no se cumpliría la definición de la notación  $O$ , obteniéndose  $1 \leq 0$ , lo cual es falso.

A partir de la solución de la suma de términos, tenemos:

$$f(n) = 2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1 \leq 2^{n+1} = \underbrace{2}_c \cdot \underbrace{2^n}_{g(n)}$$

Con este análisis, la función  $f(n) = 2^0 + 2^1 + \dots + 2^n$  es  $O(2^n)$ , con testigos  $c = 2$  y  $k = 0$ .

Las diferentes variantes para aplicar la notación  $O$  no siempre dan el mismo resultado; previamente se obtuvo que la función  $f(n)$  es acotada superiormente por  $n \cdot 2^n$  y por  $2^n$ , donde  $2^n$  acota mucho más cercanamente a la función  $f(n)$  versus  $n \cdot 2^n$ . Desde el punto de vista computacional, siempre se va a preferir la función que acote más cercanamente a la función  $f(n)$ . Las mejores cotas superiores se obtienen al aplicar la notación  $O$  sobre la fórmula que es la solución de la sumatoria de términos, cuando esto sea posible.

### Ejemplo 3:

En el Ejemplo 9 de la sección 1.4 se trabajó la relación de recurrencia:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

La cual representan la función de costo del algoritmo de búsqueda binaria (dicho algoritmo se trabajará en la sección 2.2).

El total de operaciones que realiza el algoritmo de búsqueda binaria sobre un arreglo ordenado de  $n$  elementos es:  $T(n) = (\log_2 n) + 1$ , resultado obtenido como solución de la relación de recurrencia. Ahora, ¿cuál es la notación  $O$  del algoritmo de búsqueda binaria?

$$\begin{aligned} T(n) &= (\log_2 n) + 1 \leq \log_2 n + \log_2 n \\ &= 2 \cdot \log_2 n \\ &= 2 \cdot \frac{\log n}{\log 2} \\ &= \underbrace{\frac{2}{\log 2}}_c \cdot \underbrace{\log n}_{g(n)} \end{aligned}$$

Con el análisis anterior, se tiene que el peor de los casos del algoritmo de búsqueda binaria es  $O(\log n)$  con  $c = \frac{2}{\log 2}$  y  $k = 2$  como testigos.

### 2.1.2. Notación $\Omega$

La *notación Omega* ( $\Omega$ ) es usada para estimar el número mínimo de operaciones que un algoritmo realiza a partir de sus datos de entrada. Esta notación permite calcular la cota inferior para una función  $f(n)$ , para valores de  $n \geq k$ .  $f(n)$  es la función de costo del algoritmo, función que representa la cantidad de operaciones que realiza el algoritmo para los datos con los que trabaja.

#### Definición

Sean  $f$  y  $g$  funciones desde el conjunto de los números reales al conjunto de los números reales, se dice que  $f(n)$  es  $\Omega(g(n))$  si hay constantes positivas  $c$  y  $k$  tales que:

$$|f(n)| \geq c \cdot |g(n)|, \text{ donde } n \geq k.$$

$k$  es un número natural a partir del cual se comienza a cumplir el orden  $\Omega(g(n))$  y  $c$  es una constante que pertenece a los números reales positivos.

#### Ejemplo 4:

¿Cuál es la mejor notación  $\Omega$  para  $f(n) = 1 + 2 + 3 + \dots + n$ , para  $n \geq 1$ ,  $n \in \mathbb{Z}^+$ ?

Al trabajar directamente sobre los términos que se están sumando, tenemos:

$$1 + 2 + 3 + 4 + \dots + n \geq n = \underbrace{1}_c \cdot \underbrace{n}_{g(n)}$$

De esta forma, se puede garantizar que la función  $f(n) = 1 + 2 + 3 + \dots + n$  es  $\Omega(n)$  con testigos  $c = 1$  y  $k = 1$ ; pero esta no es la mejor cota inferior.

Al trabajar sobre la solución de la sumatoria de términos, tenemos:

$$f(n) = 1 + 2 + 3 + \dots + n = \frac{n \cdot (n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \geq \frac{n^2}{2} = \underbrace{\frac{1}{2}}_c \cdot \underbrace{n^2}_{g(n)}$$

La función  $f(n) = 1 + 2 + 3 + \dots + n$  es  $\Omega(n^2)$ , donde se tienen los testigos  $c = \frac{1}{2}$  y  $k = 1$ .

La mejor notación  $\Omega$  para la función  $f(n) = 1 + 2 + 3 + \dots + n$  es  $n^2$  porque es la cota inferior más cercana. Desde el punto de vista computacional, siempre se le va a dar prioridad a la cota más cercana a la función.

### Ejemplo 5:

¿Cuál es la notación  $\Omega$  para  $f(n) = 2^0 + 2^1 + 2^2 + \dots + 2^n$ , para  $n \geq 0$ ,  $n \in \mathbb{N}$ ?

Como se puede garantizar que la mejor notación, ya sea  $O$  u  $\Omega$ , se obtiene al trabajar sobre el resultado de la solución de la sumatoria de términos, entonces se presenta a continuación únicamente el análisis sobre dicha solución:

$$f(n) = 2^0 + 2^1 + 2^2 + \dots + 2^n = 2^{n+1} - 1 \geq \frac{2^{n+1}}{2} = \underbrace{1}_c \cdot \underbrace{2^n}_{g(n)}$$

La función  $f(n) = 2^0 + 2^1 + 2^2 + \dots + 2^n$  es  $\Omega(2^n)$ , con  $c = 1$  y  $k = 0$  como testigos.

#### 2.1.3. Notación $\Theta$

La notación *Theta* ( $\Theta$ ) permite definir tanto una cota superior como inferior para la función  $f(n)$ .

##### Definición:

Sean  $f$  y  $g$  funciones desde el conjunto de los números reales al conjunto de los números reales, se dice que  $f(n)$  es  $\Theta(g(n))$  si  $f(n)$  es al mismo tiempo  $O(g(n))$  y  $\Omega(g(n))$ . Dicho en otras palabras,  $f(n)$  es  $\Theta(g(n))$  si la misma función  $g(n)$  acota superior e inferiormente a la función  $f(n)$ .

Cuando  $f(n)$  es  $\Theta(g(n))$ , se dice que  $f(n)$  es de orden  $g(n)$ .

### Ejemplo 6:

¿Cuál es la notación  $\Theta$  de  $f(n) = 1 + 2 + 3 + \dots + n$ , para  $n \geq 1$ ,  $n \in \mathbb{Z}^+$ ?

En ejemplos anteriores ya se hizo el análisis para la función  $f(n)$  con respecto a las notaciones  $O$  y  $\Omega$ , donde se obtuvo la misma función  $g(n) = n^2$ , por este motivo se garantiza que la suma de los primeros  $n$  números enteros positivos es  $\Theta(n^2)$ , o, dicho en otras palabras, es de orden  $n^2$ .

### Ejemplo 7:

¿Cuál es la notación  $\Theta$  para la función  $f(n) = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots + \frac{n}{2^n}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ ?

Se tiene que  $f(n) = \sum_{i=1}^n \frac{i}{2^i} = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots + \frac{n}{2^n} = 2 - \frac{n+2}{2^n}$ , para  $n \geq 1, n \in \mathbb{Z}^+$ .

#### Análisis de la notación O:

$$f(n) = 2 - \frac{n+2}{2^n} \leq 2 = \underbrace{2}_c \cdot \underbrace{\frac{1}{g(n)}}_{g(n)}$$

De esta forma,  $f(n) = \sum_{i=1}^n \frac{i}{2^i}$  es  $O(1)$ , con testigos  $c = 2$  y  $k = 1$ .

#### Análisis de la notación $\Omega$ :

$$f(n) = 2 - \frac{n+2}{2^n} \geq \frac{1}{2} = \underbrace{\frac{1}{2}}_c \cdot \underbrace{\frac{1}{g(n)}}_{g(n)}$$

De esta forma,  $f(n) = \sum_{i=1}^n \frac{i}{2^i}$  es  $\Omega(1)$ , con testigos  $c = \frac{1}{2}$  y  $k = 1$ .

Se obtuvo la misma función  $g(n) = 1$ , tanto en la notación  $O$  como en la notación  $\Omega$ , por este motivo  $f(n) = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots + \frac{n}{2^n}$  es  $\Theta(1)$ .

### Ejemplo 8:

En el Ejemplo 12 de la sección 1.4 se trabajó la relación de recurrencia:

$$T(1) = 1$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

Que representa la función de costo del algoritmo de ordenamiento por mezclas (Merge Sort), el cual se trabajará en la sección 2.3. La solución que se obtuvo para la relación de recurrencia fue  $T(n) = (n \cdot \log_2 n) + n$ . ¿Cuál es la notación  $\Theta$  del algoritmo Merge Sort?

#### Análisis de la notación O:

$$T(n) = (n \cdot \log_2 n) + n \leq n \cdot \log_2 n + n \cdot \log_2 n$$

$$\begin{aligned} &= 2 \cdot n \cdot \log_2 n \\ &= 2 \cdot n \cdot \frac{\log n}{\log 2} \\ &= \frac{2}{\log 2} \cdot n \cdot \log n \\ &= \underbrace{\frac{2}{\log 2}}_c \cdot \underbrace{n \cdot \log n}_{g(n)} \end{aligned}$$

De esta forma,  $T(n) = (n \cdot \log_2 n) + n$  es  $O(n \log n)$ , con testigos  $c = \frac{2}{\log 2}$  y  $k = 2$ .

**Análisis de la notación  $\Omega$ :**

$$\begin{aligned} T(n) &= (n \cdot \log_2 n) + n \geq n \cdot \log_2 n \\ &= n \cdot \frac{\log n}{\log 2} \\ &= \frac{1}{\log 2} \cdot n \cdot \log n \\ &= \underbrace{\frac{1}{\log 2}}_c \cdot \underbrace{n \cdot \log n}_{g(n)} \end{aligned}$$

De esta forma,  $T(n) = (n \cdot \log_2 n) + n$  es  $\Omega(n \log n)$ , con testigos  $c = \frac{1}{\log 2}$  y  $k = 2$ .

Se obtuvo la misma función  $g(n) = n \log n$ , tanto en la notación  $O$  como en la notación  $\Omega$ , por este motivo el algoritmo Merge Sort es  $\Theta(n \log n)$ .

## 2.2. Algoritmo de búsqueda binaria (Binary Search)

En esta función se tienen como parámetros a  $A[ ]$  que es un arreglo de elementos,  $i$  y  $j$  son índices que indican las posiciones del primer y del último elemento en el rango de valores del arreglo sobre el cual se realiza la búsqueda, y  $k$  es el elemento a buscar en el arreglo. Se exige como *precondición* para la búsqueda binaria que los elementos del arreglo  $A$  tienen que estar ordenados de forma ascendente.

```

1: function BINARYSEARCH( $A[ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:   if  $i > j$  then
3:     return  $(-1) * i - 1$ 
4:   else
5:      $m = \lfloor (i + j)/2 \rfloor$ 
6:     if  $k == A[m]$  then
7:       return  $m$ 
8:     else
9:       if  $k > A[m]$  then
10:        return BINARYSEARCH( $A[ ]$ ,  $m + 1$ ,  $j$ ,  $k$ )
11:      else
12:        return BINARYSEARCH( $A[ ]$ ,  $i$ ,  $m - 1$ ,  $k$ )
13:      end if
14:    end if
15:  end if
16: end function

```

Se tiene como *postcondición*<sup>2</sup> para la búsqueda binaria, que el valor devuelto por la función sea un entero correspondiente al índice del elemento que coincide con el valor buscado. Si el valor buscado no se encuentra, entonces el valor devuelto es:  $-1 * (\text{punto de inserción}) - 1$ . El valor del punto de inserción es el índice del elemento del arreglo donde debería encontrarse el valor buscado. La expresión: “ $-1 * (\text{punto de inserción}) - 1$ ” garantiza que el índice devuelto será mayor o igual que cero solo si el valor buscado es encontrado.

La versión de la búsqueda binaria trabajada en este libro es mucho más interesante (es mucho más poderosa) que las versiones de las búsquedas binarias presentadas en los libros: [BV02], [Cor13], [DPV06], [Fag+14], [Hil14], [Kar20], [KT06], [Laa24], [Man20], [MM21], [MR13], [Ski20], [SO15], [SW11], [Ueh19] y [Wen20], para cuando el elemento buscado no se encuentra en el arreglo  $A[ ]$ , porque no solamente devuelve un  $-1$ , o un `false` o un `not found`, sino que devuelve un número entero negativo a partir del cual se puede recuperar el punto de inserción, con el cual se indica la posición en la que “debería” estar el elemento buscado en el arreglo. Esta fortaleza será utilizada en la mayoría de los retos de programación, que en su solución utilizan la búsqueda binaria en este capítulo.

### Ejemplo 9:

Realizar el paso a paso de la búsqueda binaria cuando se tiene el arreglo de elementos:

$A[i]$	2	4	7	9	10	11	13	14	17	30
$i$	1	2	3	4	5	6	7	8	9	10

y se manda a buscar el número 12.

#### Paso a paso del algoritmo BINARYSEARCH

```

BINARYSEARCH( $A, 1, 10, 12$ )
     $i \quad j \quad k \quad m$ 
    1   10  12  5
BINARYSEARCH( $A, 6, 10, 12$ )
     $i \quad j \quad k \quad m$ 
    6   10  12  8
BINARYSEARCH( $A, 6, 7, 12$ )
     $i \quad j \quad k \quad m$ 
    6   7   12  6
BINARYSEARCH( $A, 7, 7, 12$ )
     $i \quad j \quad k \quad m$ 
    7   7   12  7
BINARYSEARCH( $A, 7, 6, 12$ )
     $i \quad j \quad k$ 
    7   6   12
return  $-8$ 

```

---

<sup>2</sup>Se ajustó el pseudocódigo del algoritmo de búsqueda binaria para que devolviera lo mismo que el método `Arrays.binarySearch()` en el lenguaje de programación Java [Ceb03], esto es similar pero diferente al método `OrderedRecordArray.find()` en el lenguaje de programación Python [CBL23].

La función `BinarySearch` devolvió un  $-8$ ; como es un número negativo, entonces el número  $12$  no se encuentra presente en el arreglo. Es importante recordar que  $-8 = -1 \cdot puntoInsercion - 1$ , donde  $puntoInsercion = 7$ , con lo cual, se indica que el elemento  $12$  debería estar ubicado en la posición  $7$  del arreglo, pero dicho elemento no se encuentra allí.

### 2.2.1. Complejidad en tiempo de ejecución del algoritmo Binary Search

Sea  $T(n)$  la función que cuenta el total de operaciones que realiza el algoritmo `BINARYSEARCH`, para determinar si el número  $k$  se encuentra presente en el arreglo  $A$ , el cual tiene  $n$  elementos.

$$T(1) = O(1)$$

$$T(n) = \underbrace{O(1)}_{\text{Costo de las líneas 2-7}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de las líneas 8-14}}, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

$$T(n) = T\left(\frac{n}{2}\right) + O(1), \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Para poder resolver la relación de recurrencia, primero se deben limpiar todas las notaciones computacionales y considerar que  $n$  toma valores que son potencias de  $2$ , por lo tanto tenemos:

$$T(1) = 1$$

$$T(n) = T\left(\frac{n}{2}\right) + 1, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

En el Ejemplo 9 de la sección 1.4 se resolvió esta relación de recurrencia, donde se obtuvo que en términos de la variable original la solución de la relación de recurrencia es:

$$T(n) = \log_2(n) + 1, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}$$

Ahora, toca aplicar sobre la solución de la relación de recurrencia la notación computacional  $O$  la cual fue borrada de la relación de recurrencia, en el Ejemplo 3 de la sección 2.1 se realizó dicho trabajo, con lo cual obtenemos que  $T(n) = O(\log(n))$ , o dicho en palabras, tenemos que en el peor de los casos la búsqueda binaria realiza  $\log(n)$  operaciones sobre el arreglo  $A$ , para determinar si el elemento  $k$  se encuentra presente en el. Donde  $n$  es la cantidad de elementos que se encuentran almacenados en el arreglo  $A$ .

### 2.2.2. Algoritmo iterativo de búsqueda binaria

El algoritmo que se presentó en la sección 2.2 para la búsqueda binaria es una *recursividad de cola*, debido a que la última instrucción que se ejecuta en el algoritmo es un llamado recursivo. Todo algoritmo que tenga una recursividad de cola puede transformarse a una versión iterativa donde la recursividad se simula por medio de un ciclo de repetición. Las versiones recursiva e iterativa del algoritmo siguen teniendo la misma complejidad teórica, pero el factor constante en la versión iterativa de la búsqueda binaria es más pequeño al no tener que hacer uso de la memoria que se reserva para la pila de los trabajos pendientes del sistema operativo. Por este motivo, en el momento de implementar los algoritmos siempre se va a preferir la versión iterativa versus la versión recursiva.

El siguiente algoritmo es la variante iterativa de la búsqueda binaria, donde se siguen teniendo los mismos parámetros de entrada, la misma precondición y la misma postcondición.

```

1: function BINARYSEARCH( $A[ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:    $r = -1$ 
3:   while  $i \leq j$  do
4:      $m = \lfloor (i + j)/2 \rfloor$ 
5:     if  $k == A[m]$  then
6:        $r = m$ 
7:       break
8:     else
9:       if  $k > A[m]$  then
10:         $i = m + 1$ 
11:      else
12:         $j = m - 1$ 
13:      end if
14:    end if
15:   end while
16:   if  $r == -1$  then
17:      $r = (-1) * i - 1$ 
18:   end if
19:   return  $r$ 
20: end function
```

### 2.2.3. Implementación del algoritmo de búsqueda binaria

Se implementará la versión iterativa del algoritmo de búsqueda binaria.

```

|| #include <stdio.h>
|| #include <stdlib.h>
|| #include <math.h>

||||***** ****
/* El valor devuelto por la función es un entero correspondiente al */
/* índice del elemento que coincide con el valor buscado. Si el valor */
/* buscado no se encuentra, entonces el valor devuelto es: */
/* - (punto de inserción) - 1. El valor del punto de inserción es el */
/* índice del elemento del arreglo donde debería encontrarse el valor */
/* buscado. La expresión: "- (punto de inserción) - 1" garantiza que el */
/* índice devuelto será mayor o igual que cero solo si el valor buscado */
/* es encontrado. */
||||***** ****

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        m = (i + j)>>1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
```

```

        break;
    }
    else
    {
        if(k > A[m])
            i = m + 1;
        else
            j = m - 1;
    }
}
if(result == -1)
    result = (-1) * i - 1;

return result;
}

int main()
{
    int A[100], index, n, queries, idQuery, k, position;

    scanf("%d", &n);
    for(index = 1; index <= n; index++)
        scanf("%d", &A[index]);

    scanf("%d", &queries);
    for(idQuery = 1; idQuery <= queries; idQuery++)
    {
        scanf("%d", &k);
        position = BinarySearch(A, 1, n, k);
        if(position >= 0)
            printf("The element %d is in the array, position: %d\n", k,
                   position);
        else
            printf("The element %d is not in the array, insertion point: %d
                   \n", k, -1 * position - 1);
    }

    return 0;
}

```

```

E:\HUGO\EstructuraDeDatos\ 10
2 4 7 9 10 11 13 14 17 30
5
12
The element 12 is not in the array, insertion point: 7
13
The element 13 is in the array, position: 7
50
The element 50 is not in the array, insertion point: 11
1
The element 1 is not in the array, insertion point: 1
10
The element 10 is in the array, position: 5

```

Figura 2.1. Salida del programa de la búsqueda binaria.

#### 2.2.4. Solución alternativa del reto: La ladrona de libros

Para este reto de programación ya se planteó una solución analítica (solución matemática) en la sección 1.3.2. Ahora, se planteará una solución que se apoya en el algoritmo de búsqueda binaria sobre un arreglo de números triangulares. Son únicamente 14,142 números triangulares que se necesitan generar y almacenar en el arreglo, porque el número triangular de la posición 14,142 es 100,005,153, el cual es el número triangular más pequeño que es mayor a la máxima talla de  $10^8$  que puede obtener como resultado Anita al ir sumando el número de cada una de las páginas que va leyendo del libro.

Sobre el arreglo de números triangulares se manda a buscar el valor de  $s$  (suma obtenida por Anita). Si dicho valor se encuentra en el arreglo, entonces la búsqueda binaria devuelve un valor positivo indicando la posición del número triangular, lo que indica que la página olvidada por Anita fue la última, por lo tanto, el total de páginas del libro y la página olvidada es el valor de la búsqueda binaria más uno. Si la búsqueda binaria devuelve un valor negativo, entonces se debe recuperar el valor del punto de inserción, el cual representa el total de páginas del libro. La página olvidada es el número triangular ubicado en el punto de inserción menos la suma obtenida por Anita.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXT 14142

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;

    while(i <= j)
    {
        m = (i + j)>>1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int main()
{
    int totalCases, idCase, s, pages, forgottenPage, i, index;
    int triangularNumbers[MAXT + 1];
```

```

triangularNumbers[0] = 0;
for(i = 1; i <= MAXT; i++)
    triangularNumbers[i] = triangularNumbers[i - 1] + i;

scanf("%d", &totalCases);
for(idCase = 1; idCase <= totalCases; idCase++)
{
    scanf("%d", &s);
    index = BinarySearch(triangularNumbers, 1, MAXT, s);

    if(index > 0)
    {
        pages = index + 1;
        forgottenPage = index + 1;
    }
    else
    {
        pages = -1 * (index + 1);
        forgottenPage = triangularNumbers[pages] - s;
    }
    printf("%d %d\n", forgottenPage, pages);
}
return 0;
}

```

El mayor costo computacional de la solución, se presenta en el fragmento de código que contiene el ciclo en el cual se hace el llamado a la búsqueda binaria:

```

for(idCase = 1; idCase <= totalCases; idCase++)
{
    scanf("%d", &s);
    index = BinarySearch(triangularNumbers, 1, MAXT, s);
    -
    -
    -
}

```

La búsqueda binaria tiene una complejidad de  $O(\log(\text{MAXT}))$  y se ejecuta  $t$  veces en el ciclo de repetición, es decir, se ejecuta una sola vez por cada caso de prueba, por lo tanto, la complejidad en tiempo de ejecución del peor de los casos es  $O(t \cdot \log(\text{MAXT}))$ .

Tomando el máximo valor para la variable  $t = 10^6$ , la cantidad de operaciones en el peor de los casos es del orden de  $(10^6) \cdot \log(14142)$ , donde  $(10^6) \cdot \log(14142) \leq (10^6) \cdot \log(10^5) = (10^6) \cdot 5 = 5 \cdot 10^6 \leq 10^7$ .

La solución del reto utilizando la búsqueda binaria es eficiente desde el punto de vista computacional porque en el peor de los casos una cantidad de operaciones del orden de  $10^7$  corre en menos de un segundo en los jueces en línea de maratones de programación, además, es un orden de  $10^7$  operaciones para dar respuesta a  $10^6$  casos de prueba.

### 2.2.5. Búsqueda binaria de un elemento que está múltiples veces en el arreglo

El algoritmo de búsqueda binaria trabaja de forma correcta sobre un arreglo de elementos ordenado de forma ascendente que tenga elementos repetidos. Si el elemento  $k$  que se manda a buscar se encuentra múltiples veces en el arreglo, el algoritmo termina retornando alguna de las posiciones del arreglo sobre la cual se encuentra el elemento almacenado. Sin embargo, no se puede garantizar que la posición devuelta por el algoritmo sea la primera o la última ocurrencia del elemento en el arreglo; por ello, es necesario trabajar las variantes del algoritmo de búsqueda binaria, para determinar la posición de la primera o de la última ocurrencia del elemento  $k$ .

El siguiente algoritmo, es la variante de la búsqueda binaria para determinar la posición de la primera ocurrencia de un elemento  $k$ , donde se sigue teniendo los mismos parámetros de entrada y la misma precondición del algoritmo de búsqueda binaria.

```
1: function BINARYSEARCHFIRSTOCCURRENCE( $A[ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:    $r = \text{BINARYSEARCH}(A[ ], i, j, k)$ 
3:   if  $r \geq 0$  then
4:      $r2 = \text{BINARYSEARCH}(A[ ], i, r - 1, k)$ 
5:     while  $r2 \geq 0$  do
6:        $r = r2$ 
7:        $r2 = \text{BINARYSEARCH}(A[ ], i, r - 1, k)$ 
8:     end while
9:   end if
10:  return  $r$ 
11: end function
```

Se tiene como *postcondición* del algoritmo que el valor devuelto es un número entero correspondiente al índice de la primera ocurrencia del elemento que coincide con el valor buscado en el arreglo. Si el valor buscado no se encuentra en el arreglo, entonces el valor devuelto es el resultado del llamado al algoritmo de búsqueda binaria de la línea 2, el cual es:  $-1 * (\text{punto de inserción}) - 1$ . De nuevo, el valor del punto de inserción es el índice en el arreglo donde debería encontrarse el valor buscado.

El siguiente algoritmo, es la variante de la búsqueda binaria para determinar la posición de la última ocurrencia de un elemento  $k$ , donde se sigue teniendo los mismos parámetros de entrada y la misma precondición del algoritmo de búsqueda binaria.

```
1: function BINARYSEARCHLASTOCCURRENCE( $A[ ]$ ,  $i$ ,  $j$ ,  $k$ )
2:    $r = \text{BINARYSEARCH}(A[ ], i, j, k)$ 
3:   if  $r \geq 0$  then
4:      $r2 = \text{BINARYSEARCH}(A[ ], r + 1, j, k)$ 
5:     while  $r2 \geq 0$  do
6:        $r = r2$ 
7:        $r2 = \text{BINARYSEARCH}(A[ ], r + 1, j, k)$ 
8:     end while
9:   end if
```

```

10:     return r
11: end function

```

Se tiene como *postcondición* del algoritmo que el valor devuelto es un número entero correspondiente al índice de la última ocurrencia del elemento que coincide con el valor buscado en el arreglo. Si el valor buscado no se encuentra en el arreglo, entonces el valor devuelto es el resultado del llamado al algoritmo de búsqueda binaria de la línea 2, el cual es:  $-1 * (\text{punto de inserción}) - 1$ . De nuevo, el valor del punto de inserción es el índice en el arreglo donde debería encontrarse el valor buscado.

En un análisis sencillo de la complejidad en tiempo de ejecución de las variantes de la búsqueda binaria (para encontrar la primera o la última ocurrencia), se tiene lo siguiente: determinar que un elemento no está en el arreglo  $A$  que contiene  $n$  elementos, es un  $O(\log(n))$ . Ahora, considerar que el arreglo  $A$  contiene múltiples veces el elemento buscado, entonces el ciclo **while** de las líneas 5–8 se ejecutará en el peor de los casos  $\log(n)$  veces con un costo del llamado del algoritmo BINARYSEARCH en el peor de los casos de  $O(\log(n))$ , donde se tiene  $(\log(n)) \cdot O(\log(n)) = O((\log(n)) \cdot (\log(n))) = O((\log(n))^2)$ .

En el análisis anterior, se sobre-estima el costo real de encontrar la primera o la última ocurrencia de un elemento  $k$  en un arreglo  $A$  que tiene  $n$  elementos ordenados de forma ascendente. Sin embargo, un  $O((\log(n))^2)$  es una complejidad en tiempo de ejecución muy pequeña, por ejemplo si  $n = 10^6$ , entonces se necesitan al rededor de  $(\log(10^6))^2 = (6)^2 = 36$  operaciones.

Se pueden generar variantes de la búsqueda binaria para encontrar la primera o la última ocurrencia de un elemento en un arreglo de  $n$  elementos con una complejidad en el peor de los casos de  $O(\log(n))$ , sin embargo, se le dio prioridad a las variantes ya trabajadas en este capítulo, a partir de la estrategia de buscar la siguiente ocurrencia del elemento, más allá de la última ocurrencia encontrada por la búsqueda binaria, y así sucesivamente hasta que la búsqueda falle.

El código fuente en Lenguaje C de las dos variantes de la búsqueda binaria es el siguiente:

```

int BinarySearchFirstOccurrence(int A[], int i, int j, int k)
{
    int result, result2;
    result = BinarySearch(A, i, j, k);

    if(result >= 0)
    {
        result2 = BinarySearch(A, i, result - 1, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, i, result - 1, k);
        }
    }

    return result;
}

int BinarySearchLastOccurrence(int A[], int i, int j, int k)
{

```

```
    int result, result2;
    result = BinarySearch(A, i, j, k);

    if(result >= 0)
    {
        result2 = BinarySearch(A, result + 1, j, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, result + 1, j, k);
        }
    }

    return result;
}
```

## 2.2.6. Reto de programación: Subsecuencia atractiva

**Nombre original:** Attractive Subsequence<sup>3</sup>.

**Fuente:** Maratón de Programación UFPS 2018.

**Fecha:** 9 de Junio de 2018.

**Autor:** Hugo Humberto Morales Peña.

Usted recibe una secuencia  $S$  de números enteros no negativos. Su tarea es calcular el total de subsecuencias atractivas. Una *subsecuencia atractiva* es una subsecuencia de elementos consecutivos en  $S$  tal que la suma de los elementos en esta es igual al valor  $K$ . Por ejemplo, considerar la secuencia  $S = \langle 0, 0, 25, 0, 0, 25 \rangle$  y el valor  $K = 25$ , hay 12 subsecuencias atractivas, estas son representadas con los pares ordenados  $(ind_1, ind_2)$ ,  $ind_1$  es la posición de el primer elemento y  $ind_2$  es la posición de el último elemento en la secuencia original  $S$ . En esta representación las subsecuencias atractivas son:  $(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 3), (3, 4), (3, 5), (4, 6), (5, 6)$  y  $(6, 6)$ .

### Formato de entrada:

La entrada contiene un único caso de prueba que consiste de tres líneas. La primera línea contiene dos números enteros positivos  $N$  ( $1 \leq N \leq 10^5$ ) y  $Q$  ( $1 \leq Q \leq 10^2$ ), el número de elementos en la secuencia  $S$  y el número de consultas respectivamente. La segunda línea contiene  $N$  números enteros no negativos  $S_i$  ( $0 \leq S_i \leq 10^3$ ). La tercera línea contiene  $Q$  números enteros positivos  $K_j$  ( $1 \leq K_j \leq 10^7$ ), para las consultas de las subsecuencias atractivas.

### Formato de salida:

Se debe imprimir  $Q$  números enteros en una sola línea, separados por un único espacio en blanco, uno por consulta, con el total de subsecuencias atractivas.

### Ejemplo de entrada 1:

```
6 2
0 0 25 0 0 25
25 50
```

<sup>3</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/attractive-subsequence>

**Ejemplo de salida 1:**

12 3

**Ejemplo de entrada 2:**

7	3					
1	6	5	2	3	4	7
7	5	11				

**Ejemplo de salida 2:**

4 2 2

**Solución del reto**

Para este reto de programación es muy importante tener en cuenta que **no** se pueden mover los elementos de la secuencia (no se pueden ordenar). Para utilizar las variantes de la búsqueda binaria para encontrar las posiciones de la primera y de la última ocurrencia, es necesario generar un arreglo de acumulados (que en el idioma inglés se conoce con la palabra *integral vector*) a partir de la secuencia; de esta forma, se cumple la precondición que se exige para el funcionamiento correcto del algoritmo de la búsqueda binaria, donde se lanzan las búsquedas sobre un arreglo que está ordenado de forma ascendente. Este arreglo puede contener valores repetidos porque en la secuencia de elementos se permite el número 0.

Para la secuencia  $S = \langle 0, 0, 25, 0, 0, 25 \rangle$ , se tiene lo siguiente:

$Acum(n)$	0	0	0	25	25	25	50
$S(n)$		0	0	25	0	0	25
$n$	0	1	2	3	4	5	6

Ahora, para contar el total de subsecuencias atractivas que tienen una suma de 25, se hace lo siguiente:

- Al valor de la posición 0 en el arreglo de acumulados se le suma 25 ( $Acum[0] + 25 = 0 + 25 = 25$ ). Para este valor (25) se averiguan las posiciones de la última y de la primera ocurrencia, para determinar la cantidad de valores en el rango ( $5 - 3 + 1 = 3$ ); el valor 3 representa las subsecuencias (1, 3), (1, 4) y (1, 5).
- Al valor de la posición 1 en el arreglo de acumulados se le suma 25 ( $Acum[1] + 25 = 0 + 25 = 25$ ). Para este valor (25) se averiguan las posiciones de la última y de la primera ocurrencia, para determinar la cantidad de valores en el rango ( $5 - 3 + 1 = 3$ ); el valor 3 representa las subsecuencias (2, 3), (2, 4) y (2, 5).
- Al valor de la posición 2 en el arreglo de acumulados se le suma 25 ( $Acum[2] + 25 = 0 + 25 = 25$ ). Para este valor (25) se averiguan las posiciones de la última y de la primera ocurrencia, para determinar la cantidad de valores en el rango ( $5 - 3 + 1 = 3$ ); el valor 3 representa las subsecuencias (3, 3), (3, 4) y (3, 5).
- Al valor de la posición 3 en el arreglo de acumulados se le suma 25 ( $Acum[3] + 25 = 25 + 25 = 50$ ). Para este valor (50) se averiguan las posiciones de la última y de la

primera ocurrencia, para determinar la cantidad de valores en el rango ( $6 - 6 + 1 = 1$ ); el valor 1 representa la subsecuencia (4, 6).

- Al valor de la posición 4 en el arreglo de acumulados se le suma 25 ( $Acum[4] + 25 = 25 + 25 = 50$ ). Para este valor (50) se averiguan las posiciones de la última y de la primera ocurrencia, para determinar la cantidad de valores en el rango ( $6 - 6 + 1 = 1$ ); el valor 1 representa la subsecuencia (5, 6).
- Al valor de la posición 5 en el arreglo de acumulados se le suma 25 ( $Acum[5] + 25 = 25 + 25 = 50$ ). Para este valor (50) se averiguan las posiciones de la última y de la primera ocurrencia, para determinar la cantidad de valores en el rango ( $6 - 6 + 1 = 1$ ); el valor 1 representa la subsecuencia (6, 6).

De esta forma, se han contado las 12 subsecuencias atractivas que hay en la secuencia.

La estrategia explicada previamente se implementa a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define MAXN 100000

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;

    while(i <= j)
    {
        m = (i + j)>>1; /* m = (i + j)/2; */
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int BinarySearchFirstOccurrence(int A[], int i, int j, int k)
{
    int result, result2;
    result = BinarySearch(A, i, j, k);

    if(result >= 0)
    {
```

```

        result2 = BinarySearch(A, i, result - 1, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, i, result - 1, k);
        }

    }

    return result;
}

int BinarySearchLastOccurrence(int A[], int i, int j, int k)
{
    int result, result2;
    result = BinarySearch(A, i, j, k);

    if(result >= 0)
    {
        result2 = BinarySearch(A, result + 1, j, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, result + 1, j, k);
        }
    }

    return result;
}

int main()
{
    int i, n, s, element, beginPosition, finalPosition;
    int q, idQuery, index1, index2;
    int integralVector[MAXN + 1];
    long long int result;

    scanf("%d %d", &n, &q);
    integralVector[0] = 0;

    for(i=1; i<=n; i++)
    {
        scanf("%d", &element);
        integralVector[i] = integralVector[i - 1] + element;
    }

    for(idQuery=1; idQuery<=q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i=0; i<=n; i++)
        {
            index1 = BinarySearchFirstOccurrence(integralVector,
                                              i + 1, n, integralVector[i] + s);
            if(index1 > 0)
            {
                beginPosition = index1;
                index2 = BinarySearchLastOccurrence(integralVector,
                                              index1, n, integralVector[i] + s);
            }
        }
    }
}

```

```
        finalPosition = index2;
        result += (finalPosition - beginPosition + 1);
    }
}
if(idQuery == 1)
    printf("%lld", result);
else
    printf(" %lld", result);
}
printf("\n");

return 0;
}
```

Salida del programa anterior ejecutando los ejemplos del reto:

```
6 2
0 0 25 0 0 25
25 50
12 3

7 3
1 6 5 2 3 4 7
7 5 11
4 2 2
```

Figura 2.2. Salida del programa para el reto: Subsecuencia atractiva.

El mayor costo computacional se presenta en el fragmento de código donde se evalúa una a una cada consulta:

```
for(idQuery=1; idQuery<=q; idQuery++)
{
    result = 0;
    scanf("%d", &s);
    for(i=0; i<=n; i++)
    {
        index1 = BinarySearchFirstOccurrence(integralVector,
                                              i + 1, n, integralVector[i] + s);
        if(index1 > 0)
        {
            beginPosition = index1;
            index2 = BinarySearchLastOccurrence(integralVector,
                                              index1, n, integralVector[i] + s);
            finalPosition = index2;
            result += (finalPosition - beginPosition + 1);
        }
    }
    ...
    ...
    ...
}
```

Para cada una de las  $q$  consultas se recorren las  $n$  posiciones del arreglo de acumulados lanzando las variantes de la búsqueda binaria para la primera y la última ocurrencia; esto tiene una complejidad en tiempo de ejecución de  $O(q \cdot n \cdot (\log(n))^2)$ .

Tomando los máximos valores para las variables  $q$  y  $n$ , la cantidad de operaciones en el peor de los casos es del orden de  $q \cdot n \cdot (\log(n))^2 = 10^2 \cdot 10^5 \cdot (\log(10^5))^2 = 10^7 \cdot (5)^2 = 10^7 \cdot 25 = 25 \cdot 10^7 = (2,5 \cdot 10) \cdot 10^7 = 2,5 \cdot (10 \cdot 10^7) = 2,5 \cdot 10^8$

## 2.3. Algoritmo de ordenamiento por mezclas (Merge Sort)

El pseudocódigo del algoritmo de ordenamiento por mezclas fue tomado de [Cor+01].

### 2.3.1. Función Merge

En esta subrutina se tienen como parámetros a  $A[ ]$ , que es un arreglo de elementos,  $p$ ,  $q$  y  $r$ , que son índices que numeran los elementos del arreglo, tal que  $p \leq q < r$ .

```

1: function MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
2:    $n_1 = q - p + 1$ 
3:    $n_2 = r - q$ 
4:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
5:   for  $i = 1$  to  $n_1$  do
6:      $L[i] = A[p + i - 1]$ 
7:   end for
8:   for  $j = 1$  to  $n_2$  do
9:      $R[j] = A[q + j]$ 
10:  end for
11:   $L[n_1 + 1] = \infty$ 
12:   $R[n_2 + 1] = \infty$ 
13:   $i = 1$ 
14:   $j = 1$ 
15:  for  $k = p$  to  $r$  do
16:    if  $L[i] \leq R[j]$  then
17:       $A[k] = L[i]$ 
18:       $i = i + 1$ 
19:    else
20:       $A[k] = R[j]$ 
21:       $j = j + 1$ 
22:    end if
23:  end for
24: end function

```

### 2.3.2. Función Merge Sort

Esta es la rutina que llama a MERGE.  $A[ ]$  es un arreglo de elementos,  $p$  y  $r$  son dos índices que denotan el inicio y el fin del arreglo, tal que  $p \leq r$ .

```

1: function MERGESORT( $A[ ]$ ,  $p$ ,  $r$ )
2:   if  $p < r$  then
3:      $q = \lfloor (p+r)/2 \rfloor$ 
4:     MERGESORT( $A[ ]$ ,  $p$ ,  $q$ )
5:     MERGESORT( $A[ ]$ ,  $q+1$ ,  $r$ )
6:     MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
7:   end if
8: end function

```

### Ejemplo 10:

Realizar el paso a paso del algoritmo MERGESORT sobre el arreglo de elementos:

$A[i]$	3	1	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

### Paso a paso del algoritmo MERGESORT

MERGESORT( $A$ , 1, 8)

$p$	$r$	$q$
1	8	4

MERGESORT( $A$ , 1, 4)

$p$	$r$	$q$
1	4	2

MERGESORT( $A$ , 1, 2)

$p$	$r$	$q$
1	2	1

MERGESORT( $A$ , 1, 1)

$p$	$r$
1	1

MERGESORT( $A$ , 2, 2)

$p$	$r$
2	2

MERGE( $A$ , 1, 1, 2)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	1	2	1	1	1	1	1
					2	2	2
					1	1	3
					2	2	

$A[i]$	3	1	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	3	$\infty$	$R[j]$	1	$\infty$
$i$	1	2	$j$	1	2

$A[i]$	1	3	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A, 3, 4$ )

$p$	$r$	$q$
3	4	3

MERGESORT( $A, 3, 3$ )

$p$	$r$
3	3

MERGESORT( $A, 4, 4$ )

$p$	$r$
4	4

MERGE( $A, 3, 3, 4$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
3	3	4	1	1	1	1	3
					2	2	4
					1	1	5
					2	2	

$A[i]$	1	3	4	2	5	1	3	2
$i$	1	2	3	4	5	6	7	8
$L[i]$	4	$\infty$			$R[j]$	2	$\infty$	
$i$	1	2			$j$	1	2	

$A[i]$	1	3	2	4	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGE( $A, 1, 2, 4$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	2	4	2	2	1	1	1
					2	2	2
					3	3	3
					1	1	4
					2	2	5
					3	3	

$A[i]$	1	3	2	4	5	1	3	2
$i$	1	2	3	4	5	6	7	8
$L[i]$	1	3	$\infty$		$R[j]$	2	4	$\infty$
$i$	1	2	3		$j$	1	2	3

$A[i]$	1	2	3	4	5	1	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A, 5, 8$ )

$p$	$r$	$q$
5	8	6

MERGESORT( $A$ , 5, 6)

$p$	$r$	$q$
5	6	5

MERGESORT( $A$ , 5, 5)

$p$	$r$
5	5

MERGESORT( $A$ , 6, 6)

$p$	$r$
6	6

MERGE( $A$ , 5, 5, 6)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
5	5	6	1	1	1	1	5
					2	2	6
					1	1	7
					2	2	

$A[i]$	1	2	3	4	5	1	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	5	$\infty$		$R[j]$	1	$\infty$	
$i$	1	2		$j$	1	2	

$A[i]$	1	2	3	4	1	5	3	2
$i$	1	2	3	4	5	6	7	8

MERGESORT( $A$ , 7, 8)

$p$	$r$	$q$
7	8	7

MERGESORT( $A$ , 7, 7)

$p$	$r$
7	7

MERGESORT( $A$ , 8, 8)

$p$	$r$
8	8

MERGE( $A$ , 7, 7, 8)

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
7	7	8	1	1	1	1	7
					2	2	8
					1	1	9
					2	2	

$A[i]$	1	2	3	4	1	5	3	2
$i$	1	2	3	4	5	6	7	8

$L[i]$	3	$\infty$		$R[j]$	2	$\infty$	
$i$	1	2		$j$	1	2	

$A[i]$	1	2	3	4	1	5	2	3
$i$	1	2	3	4	5	6	7	8

MERGE( $A, 5, 6, 8$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
5	6	8	2	2	1	1	5
					2	2	6
					3	3	7
					1	1	8
					2	2	9
					3	3	
$A[i]$		1	2	3	4	1	5
$i$		1	2	3	4	5	6
$L[i]$		1	5	$\infty$			
$i$		1	2	3			
$A[i]$		1	2	3	4	1	2
$i$		1	2	3	4	5	6
$R[j]$		2	3	$\infty$			
$j$		1	2				
$A[i]$		1	2	3	4	1	2
$i$		1	2	3	4	5	6

MERGE( $A, 1, 4, 8$ )

$p$	$q$	$r$	$n_1$	$n_2$	$i$	$j$	$k$
1	4	8	4	4	1	1	1
					2	2	2
					3	3	3
					4	4	4
					5	5	5
					1	1	6
					2	2	7
					3	3	8
					4	4	9
					5	5	
$A[i]$		1	2	3	4	1	2
$i$		1	2	3	4	5	6
$L[i]$		1	2	3	4	$\infty$	
$i$		1	2	3	4	5	
$R[j]$		1	2	3	5	$\infty$	
$j$		1	2	3	4	5	
$A[i]$		1	1	2	2	3	4
$i$		1	2	3	4	5	6

*Propiedad de estabilidad:* un algoritmo de ordenamiento tiene la propiedad de estabilidad, cuando después de ordenar los elementos del arreglo se sigue conservando el orden relativo que originalmente había entre los elementos repetidos.

El algoritmo MERGESORT tiene la propiedad de estabilidad. En el paso a paso anterior, se evidencia al analizar el resultado generado por el llamado a la sub-rutina MERGE( $A, 1, 4, 8$ ), donde si el elemento activo a mezclar es el mismo entre los arreglos  $L$  y  $R$ , entonces primero se toma (primero se ubica) el elemento del arreglo  $L$ .

### 2.3.3. Complejidad en espacio de almacenamiento del algoritmo de ordenamiento por mezclas

La Figura 2.3 representa el consumo de memoria realizado por el algoritmo de ordenamiento por mezclas, al realizar las copias del sub-arreglo de los elementos de la izquierda y del sub-arreglo de los elementos de la derecha entre los diferentes llamados recursivos del algoritmo.

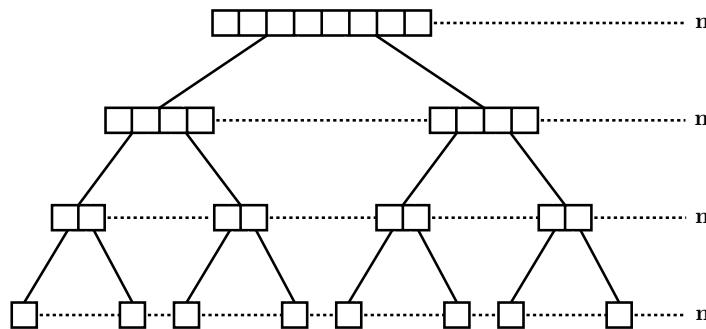


Figura 2.3. Árbol del consumo de la memoria generado por el algoritmo de ordenamiento por mezclas.

La complejidad en consumo de espacio de almacenamiento en memoria es  $n$  por la altura del árbol, donde  $n$  es la cantidad de elementos del arreglo  $A$ . Por cada nivel del árbol se necesita una copia completa de los  $n$  elementos del arreglo  $A$ .

Para poder determinar la altura del árbol, entonces se considerará que  $n$  es una potencia del número 2 y que se divide sucesivamente por 2 hasta llegar a 1, por lo tanto:

$$\frac{\left(\frac{n}{2}\right)}{2} = 1, \quad \frac{n}{2^{\text{altura}}} = 1, \quad n = 2^{\text{altura}}, \quad \log_2(n) = \log_2(2^{\text{altura}}) = \text{altura}, \quad \text{altura} = \log_2(n).$$

La complejidad en consumo de espacio de almacenamiento en memoria es:  $n \cdot \log_2(n) = \Theta(n \log n)$ .

### 2.3.4. Complejidad en tiempo de ejecución del algoritmo de ordenamiento por mezclas

```

1: function MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
2:    $n_1 = q - p + 1$ 
3:    $n_2 = r - q$ 
4:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
5:   for  $i = 1$  to  $n_1$  do
6:      $L[i] = A[p + i - 1]$ 
7:   end for
8:   for  $j = 1$  to  $n_2$  do
9:      $R[j] = A[q + j]$ 
10:  end for
11:   $L[n_1 + 1] = \infty$ 
```

```

12:    $R[n_2 + 1] = \infty$ 
13:    $i = 1$ 
14:    $j = 1$ 
15:   for  $k = p$  to  $r$  do
16:     if  $L[i] \leq R[j]$  then
17:        $A[k] = L[i]$ 
18:        $i = i + 1$ 
19:     else
20:        $A[k] = R[j]$ 
21:        $j = j + 1$ 
22:     end if
23:   end for
24: end function

```

Considerar que en el rango de  $p$  a  $r$  en el arreglo  $A$  hay  $n$  elementos. El ciclo de repetición **for** de las líneas 5-7 se ejecuta  $\frac{n}{2}$  veces. El ciclo de repetición **for** de las líneas 8-10 se ejecuta  $\frac{n}{2}$  veces. El ciclo de repetición **for** de las líneas 15-23 se ejecuta  $n$  veces. Por lo tanto, el costo total de la función MERGE es:  $\frac{n}{2} + \frac{n}{2} + n = 2n = \Theta(n)$ .

```

1: function MERGESORT( $A[ ]$ ,  $p$ ,  $r$ )
2:   if  $p < r$  then
3:      $q = \lfloor (p+r)/2 \rfloor$ 
4:     MERGESORT( $A[ ]$ ,  $p$ ,  $q$ )
5:     MERGESORT( $A[ ]$ ,  $q+1$ ,  $r$ )
6:     MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
7:   end if
8: end function

```

De nuevo, considerar que en el rango de  $p$  a  $r$  en el arreglo  $A$  hay  $n$  elementos.

Sea  $T(n)$  la función que cuenta el total de operaciones que realiza el algoritmo MERGESORT, para ordenar un arreglo  $A$  que tiene  $n$  elementos.

$$T(1) = \Theta(1)$$

$$T(n) = \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de la línea 4}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{Costo de la línea 5}} + \underbrace{\Theta(n)}_{\text{Costo de la línea 6}}, \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + \Theta(n), \text{ para } n \geq 2, n \in \mathbb{Z}^+$$

Para resolver la relación de recurrencia, se deben limpiar todas las notaciones computacionales y considerar que  $n$  toma valores que son potencias de 2, por lo tanto tenemos:

$$T(1) = 1$$

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n, \text{ para } n = 2^m, m \geq 1, m \in \mathbb{Z}^+$$

En el Ejemplo 12 de la sección 1.4 se resolvió esta relación de recurrencia, donde se obtuvo que en términos de la variable original la solución de la relación de recurrencia es:

$$T(n) = n \cdot \log_2(n) + n, \text{ para } n = 2^m, m \geq 0, m \in \mathbb{N}.$$

Ahora, toca aplicar sobre la solución de la relación de recurrencia la notación computacional  $\Theta$  la cual fue borrada de la relación de recurrencia. En el Ejemplo 8 de la sección 2.1 se realizó dicho trabajo, con lo cual obtenemos que  $T(n) = \Theta(n \log n)$ , o dicho en palabras, tenemos que en el mejor y el peor de los casos el algoritmo de ordenamiento por mezclas realiza operaciones por el orden de  $n \log n$  sobre el arreglo  $A$  para ordenarlo, donde  $n$  es la cantidad de elementos que se encuentran almacenados en el arreglo  $A$ .

### 2.3.5. Implementación del algoritmo de ordenamiento por mezclas

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, L[n1 + 2], R[n2 + 2];

    for(i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];

    for(j = 1; j <= n2; j++)
        R[j] = A[q + j];

    L[n1 + 1] = myInfinite;
    R[n2 + 1] = myInfinite;
    i = 1;
    j = 1;

    for(k = p; k <= r; k++)
    {
        if(L[i] <= R[j])
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
    }
}

void MergeSort(int A[], int p, int r)
{
    int q;

    if(p < r)
    {
```

```

        q = (p + r) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }

int main()
{
    int A[20], i, n;

    while(scanf(" %d", &n) != EOF)
    {
        for(i = 1; i <= n; i++)
            scanf(" %d", &A[i]);

        for(i = 1; i <= n; i++)
            printf("%d ", A[i]);
        printf("\n");

        MergeSort(A, 1, n);

        for(i = 1; i <= n; i++)
            printf("%d ", A[i]);
        printf("\n");
    }

    return 0;
}

```

```

E:\HUGO\EstructuraDeDatos\ 8
3 1 4 2 5 1 3 2
3 1 4 2 5 1 3 2
1 1 2 2 3 3 4 5
3 1 4 2 5 1 3 2

```

Figura 2.4. Salida del programa que realiza el ordenamiento por mezclas.

### 2.3.6. Reto de programación: ¿Cuántas inversiones?

**Nombre original:** How Many Inversions?<sup>4</sup>

**Fuente:** Maratón de Programación UFPS 2017.

**Fecha:** 10 de Junio de 2017.

**Autor:** Hugo Humberto Morales Peña.

<sup>4</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/how-many-inversions>

<sup>5</sup>[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=878&page=show\\_problem&problem=5135](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=878&page=show_problem&problem=5135)

<sup>6</sup>Este reto de programación se generó a partir del problema “2-4 Inversions” del libro *Introduction to Algorithms* [Cor+01]

<sup>7</sup>Los hermanos Halim en su libro [HHE18] sugieren la solución de este reto de programación en la sección “2.2. Linear DS with built-in Libraries”

Humbertov Moralov en su época de estudiante de Ingeniería de Sistemas en la Universidad Sin Lomas, fue evaluado en su primer examen de análisis de algoritmos (en el primer semestre del año 1997) con la siguiente temática y preguntas:

**Inversiones:**

Sea  $A[1 \dots n]$  un arreglo de números enteros de tamaño  $n$  con todos sus elementos distintos. Si  $i < j$  y  $A[i] > A[j]$ , entonces el par ordenado  $(i, j)$  es llamado una inversión<sup>8</sup> del arreglo  $A$ .

Dada la definición anterior acerca de una inversión, Humbertov Moralov debe responder las siguientes preguntas:

- 1.) Listar todas las inversiones del arreglo  $\langle 3, 2, 8, 1, 6 \rangle$ .
- 2.) ¿Cuál arreglo de tamaño  $n$  con todos los números del conjunto  $\{1, 2, 3, \dots, n\}$  tiene la mayor cantidad de inversiones? En concordancia con la respuesta a la pregunta anterior, ¿cuántas inversiones tiene?
- 3.) Escribir un algoritmo que determine el número de inversiones en cualquier permutación de  $n$  elementos en  $\theta(n \log n)$  en el peor de los casos en tiempo de ejecución.

Humbertov Moralov logró contestar sin mayor problemas las preguntas 1 y 2, pero con el punto 3 no fue capaz de hacerlo en el tiempo de duración del examen. Días después, logró plantear la siguiente solución:

```
1: inv = 0
2: function MERGE( $A[ ]$ ,  $p$ ,  $q$ ,  $r$ )
3:    $n_1 = q - p + 1$ 
4:    $n_2 = r - q$ 
5:   create arrays  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$ 
6:   for  $i = 1$  to  $n_1$  do
7:      $L[i] = A[p + i - 1]$ 
8:   end for
9:   for  $j = 1$  to  $n_2$  do
10:     $R[j] = A[q + j]$ 
11:   end for
12:    $L[n_1 + 1] = \infty$ 
13:    $R[n_2 + 1] = \infty$ 
14:    $i = 1$ 
15:    $j = 1$ 
16:   for  $k = p$  to  $r$  do
17:     if  $L[i] \leq R[j]$  then
18:        $A[k] = L[i]$ 
19:        $i = i + 1$ 
20:     else
21:        $A[k] = R[j]$ 
22:        $j = j + 1$ 
```

<sup>8</sup>El concepto de inversión se trabaja a profundidad en los libros [Knu98] y [Rou17]

```

23:         inv = inv + n1 - i + 1
24:     end if
25: end for
26: end function

```

```

27: function MERGESORT(A[ ], p, r)
28:     if p < r then
29:         q = ⌊(p + r)/2⌋
30:         MERGESORT(A[ ], p, q)
31:         MERGESORT(A[ ], q + 1, r)
32:         MERGE(A[ ], p, q, r)
33:     end if
34: end function

```

¿Este código resolvería el problema? ¿fue suficiente con agregar las líneas 1 y 23 para resolver el problema?

Por favor, ayudar a Humberto Moralov a validar dicha solución. Para lo cual, usted debe implementar la solución propuesta en alguno de los lenguajes de programación aceptados por el ICPC y verificar si se generan los resultados esperados.

#### Formato de entrada:

La entrada comienza con un número entero positivo  $t$  ( $1 \leq t \leq 10$ ), el cual representa el total de casos de prueba. Cada uno de los casos de prueba tiene la siguiente estructura:

La primera línea contiene un número entero positivo  $n$  ( $1 \leq n \leq 10^6$ ), el cual representa la cantidad de elementos que tiene el arreglo  $A$ .

La segunda línea contiene  $n$  números enteros positivos separados por un espacio en blanco los cuales constituyen el arreglo  $A$ . Estos valores están en el intervalo cerrado  $[1, 10^8]$ .

#### Formato de salida:

Para cada caso de prueba, su programa debe imprimir un número entero no negativo que represente la cantidad total de inversiones que se encuentran presentes en el arreglo  $A$ . Cada caso de prueba debe generar una sola línea en la salida.

#### Ejemplo de entrada:

```

3
5
3 2 8 1 6
5
5 4 3 2 1
1
10

```

### Ejemplo de salida:

```
5
10
0
```

### Solución del reto

El objetivo de este reto de programación, es simplemente mostrar una forma eficiente de calcular las inversiones que se encuentran presentes entre los elementos de un arreglo. Los retos de programación con la temática de inversiones son muy comunes en programación competitiva y se debe trabajar en las primeras etapas de formación de sus participantes.

La solución planteada en el reto es correcta para contar el número de inversiones que se encuentran presentes entre los elementos del arreglo. Es una implementación directa del pseudocódigo donde la variable `inv` tiene que ser definida como un entero de 64 bits, porque en el peor de los casos la entrada del reto es un arreglo de un millón de elementos ordenado de forma descendente, donde el total de inversiones es el resultado de  $1 + 2 + 3 + 4 + \dots + 999,999 = \frac{999,999 \times 1,000,000}{2} = 499,999,500,000$ , resultado que desborda una variable del tipo entero de 32 bits.

La implementación de la solución del reto es la siguiente:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXT 1000000

unsigned long long int inv = 0;

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, L[n1 + 2], R[n2 + 2];

    for(i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];

    for(j = 1; j <= n2; j++)
        R[j] = A[q + j];

    L[n1 + 1] = myInfinite;
    R[n2 + 1] = myInfinite;
    i = 1;
    j = 1;

    for(k = p; k <= r; k++)
    {
        if(L[i] <= R[j])
        {
            A[k] = L[i];
            i++;
        }
    }
}
```

```

        else
        {
            A[k] = R[j];
            j++;
            inv += n1 - i + 1;
            /* inv = inv + n1 - i + 1; */
        }
    }

void MergeSort(int A[], int p, int r)
{
    int q;

    if(p < r)
    {
        q = (p + r) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int main()
{
    int A[MAXT + 1], t, i, n, idCase;

    scanf("%d", &t);

    for(idCase = 1; idCase <= t; idCase++)
    {
        scanf("%d", &n);
        inv = 0;
        for(i = 1; i <= n; i++)
            scanf("%d", &A[i]);

        MergeSort(A, 1, n);
        printf("%llu\n", inv);
    }

    return 0;
}

```

```

E:\HUGO\EstructuraDeDatos\ 3
5
3 2 8 1 6
5
5
5 4 3 2 1
10
1
10
0

```

Figura 2.5. Salida del programa para el reto: ¿Cuántas inversiones?

Para cada uno de los  $t$  casos de prueba se hace el llamado del algoritmo de ordenamiento por mezclas, esto tiene una complejidad en tiempo de ejecución de  $\Theta(t \cdot n \cdot (\log(n)))$ .

Tomando los máximos valores para las variables  $t$  y  $n$ , la cantidad de operaciones es del orden de  $t \cdot n \cdot (\log(n)) = 10 \cdot 10^6 \cdot (\log(10^6)) = 10^7 \cdot (6) = 6 \cdot 10^7 \leq 10^8$ .

## 2.4. Retos de programación resueltos

En esta sección se trabajarán dos retos de programación que se apoyan en ordenamiento y búsqueda:

- ¿Cuántos subconjuntos?
- Colección dinámica

### 2.4.1. ¿Cuántos subconjuntos?

**Nombre original:** How Many Sub Sets<sup>9</sup>.

**Fuente:** ICPC Bolivia 2019.

**Fecha:** 28 de Septiembre de 2019.

**Autor:** Hugo Humberto Morales Peña.

Se tiene un conjunto  $A$  de números enteros positivos con cardinalidad  $n$  ( $|A| = n$ , es importante recordar que la cardinalidad de un conjunto es el total de elementos distintos que este tiene). Se quiere averiguar cuántos subconjuntos hay de tamaño dos que tengan una suma de sus elementos menor o igual a  $S$ .

Por ejemplo, si se tiene el siguiente conjunto  $A = \{6, 5, 1, 4, 2, 3\}$  y  $S = 7$ , entonces hay un total de 9 subconjuntos de tamaño dos cuya suma de sus elementos es menor o igual a 7, dichos subconjuntos son:  $\{6, 1\}, \{5, 1\}, \{5, 2\}, \{1, 4\}, \{1, 2\}, \{1, 3\}, \{4, 2\}, \{4, 3\}$  y  $\{2, 3\}$ .

Su misión, si decide aceptarla, es el de contar el total de subconjuntos de tamaño dos que tienen una suma de sus elementos menor o igual a  $S$ .

#### Formato de entrada:

La entrada del problema consiste de un único caso de prueba. El caso de prueba comienza con una línea que contiene dos números enteros positivos  $n$  ( $1 \leq n \leq 5 \cdot 10^5$ ) y  $q$  ( $1 \leq q \leq 10$ ), los cuales representan respectivamente la cardinalidad del conjunto  $A$  y el total de consultas que se van a realizar sobre el conjunto  $A$ . La siguiente línea contiene exactamente  $n$  números enteros positivos (separados por un espacio en blanco)  $A_1, A_2, A_3, \dots, A_n$  ( $1 \leq A_i \leq 10^8$ , para  $1 \leq i \leq n$ ); obviamente se garantiza que los  $n$  elementos del conjunto  $A$  son diferentes. La última línea contiene exactamente  $q$  números enteros positivos (separados por un espacio en blanco)  $S_1, S_2, S_3, \dots, S_q$  ( $1 \leq S_j \leq 2 \cdot 10^8$ , para  $1 \leq j \leq q$ ), para las consultas.

<sup>9</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/how-many-sub-sets>

**Formato de salida:**

Su programa debe imprimir  $q$  líneas (una línea de respuesta por consulta), cada una de ellas conteniendo un único valor que representa el resultado total de subconjuntos de tamaño dos que la suma de sus elementos es menor o igual a  $S$ .

**Ejemplo de entrada:**

```
6 3
6 5 1 4 2 3
7 8 12
```

**Ejemplo de salida:**

```
9
11
15
```

**Solución “ingenua” del reto**

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXN 500000

int main()
{
    int n, q, A[MAXN + 1], s, result;
    int idElement, idQuery, i, j;

    scanf("%d %d", &n, &q);

    for(idElement = 1; idElement <= n; idElement++)
        scanf("%d", &A[idElement]);

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            for(j = i + 1; j <= n; j++)
            {
                if((A[i] + A[j]) <= s)
                    result++;
            }
        }
        printf("%d\n", result);
    }
    return 0;
}
```

Esta solución genera resultados correctos para el reto, sin embargo, la complejidad computacional es muy alta.

```

    for(i = 1; i < n; i++)
    {
        for(j = i + 1; j <= n; j++)
        {
            if((A[i] + A[j]) <= s)
                result++;
        }
    }
}

```

El fragmento de código que contiene los ciclos anidados genera la siguiente cantidad de operaciones:

- Cuando  $i = 1$  el ciclo anidado se ejecuta 499,999 veces.
- Cuando  $i = 2$  el ciclo anidado se ejecuta 499,998 veces.
- Cuando  $i = 3$  el ciclo anidado se ejecuta 499,997 veces.
- $\vdots$
- Cuando  $i = 499,999$  el ciclo anidado se ejecuta una (1) vez.

Por lo tanto, el doble ciclo **for** ejecuta un total de operaciones de:

$$\begin{aligned}
 499,999 + 499,998 + 499,997 + 499,996 + \dots + 1 &= 1 + 2 + 3 + 4 + \dots + 499,999 \\
 &= \frac{499,999 \cdot (500000)}{2} \\
 &= 124,999,750,000 \\
 &\approx 10^{11}
 \end{aligned}$$

Un orden de  $10^{11}$  operaciones para dar respuesta a cada consulta es inaceptable. No tiene sentido poner a “correr” dicha solución por no menos de dos horas para que pueda generar todas las salidas para el archivo de entrada de los casos de prueba. Estamos en la obligación de seguir trabajando en una solución que en la complejidad del tiempo de ejecución sea mejor.

### Solución “ingenua mejorada” del reto

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXN 500000

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, L[n1 + 2], R[n2 + 2];

    for(i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];

    for(j = 1; j <= n2; j++)
        R[j] = A[q + j];
}

```

```

L[n1 + 1] = myInfinite;
R[n2 + 1] = myInfinite;
i = 1;
j = 1;

for(k = p; k <= r; k++)
{
    if(L[i] <= R[j])
    {
        A[k] = L[i];
        i++;
    }
    else
    {
        A[k] = R[j];
        j++;
    }
}
}

void MergeSort(int A[], int p, int r)
{
    int q;
    if(p < r)
    {
        q = (p + r) >> 1; /* q = (p + r) / 2 */
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int main()
{
    int n, q, A[MAXN + 1], s, result;
    int idElement, idQuery, i, j;
    scanf("%d %d", &n, &q);
    for(idElement = 1; idElement <= n; idElement++)
        scanf("%d", &A[idElement]);

    MergeSort(A, 1, n);
    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            for(j = i + 1; j <= n; j++)
            {
                if((A[i] + A[j]) <= s)
                    result++;
                else
                    break;
            }
        }
        printf("%d\n", result);
    }
    return 0;
}

```

Esta solución genera resultados correctos para el reto, sin embargo, la complejidad computacional sigue siendo muy alta, debido a que el fragmento de código que contiene los ciclos anidados:

```

for(i = 1; i < n; i++)
{
    for(j = i + 1; j <= n; j++)
    {
        if((A[i] + A[j]) <= s)
            result++;
        else
            break;
    }
}

```

Se sigue realizando un orden de  $10^{11}$  operaciones para dar respuesta a cada caso de prueba, independientemente de tener los elementos en el arreglo ordenados de forma ascendente, esto se presenta cuando  $S = 2 \cdot 10^8$ , donde se garantiza que cualquier subconjunto de tamaño dos en el conjunto  $A$  tendrá una suma de sus elementos menor que dicha cantidad, y donde el total de subconjuntos de tamaño dos es del orden de  $10^{11}$ . De nuevo, estamos en la obligación de plantear una solución que en la complejidad en tiempo de ejecución realice una cantidad de operaciones del orden de  $10^7$  o  $10^8$ .

### Solución eficiente en tiempo de ejecución para el reto

```

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXN 500000

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, L[n1 + 2], R[n2 + 2];

    for(i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];

    for(j = 1; j <= n2; j++)
        R[j] = A[q + j];

    L[n1 + 1] = myInfinite;
    R[n2 + 1] = myInfinite;
    i = 1;
    j = 1;

    for(k = p; k <= r; k++)
    {
        if(L[i] <= R[j])
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
    }
}

```

```

        }
    }

void MergeSort(int A[], int p, int r)
{
    int q;

    if(p < r)
    {
        q = (p + r) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;

    while(i <= j)
    {
        /* m = (i + j)/2; */
        m = (i + j)>>1;
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }

    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int main()
{
    int n, q, A[MAXN + 1];
    int s, index, element;
    int idElement, idQuery, i;
    unsigned long long int result;

    scanf(" %d %d", &n, &q);

    for(idElement = 1; idElement <= n; idElement++)
        scanf(" %d", &A[idElement]);

    MergeSort(A, 1, n);
}

```

```

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            element = s - A[i];
            if(element > A[i])
            {
                index = BinarySearch(A, i + 1, n, element);
                if(index < 0)
                    index = (-1 * index) - 2;

                result += (index - i);
            }
            else
                break;
        }
        printf("%llu\n", result);
    }
    return 0;
}

```

Esta solución, además de generar resultados correctos para el reto, también cuenta con una complejidad en tiempo de ejecución aceptable. El mayor costo computacional se presenta en el fragmento de código que contiene el ciclo en el cual se hace el llamado a la búsqueda binaria:

```

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        result = 0;
        scanf("%d", &s);
        for(i = 1; i < n; i++)
        {
            element = s - A[i];
            if(element > A[i])
            {
                index = BinarySearch(A, i + 1, n, element);
                if(index < 0)
                    index = (-1 * index) - 2;

                result += (index - i);
            }
            else
                break;
        }
        printf("%llu\n", result);
    }

```

La búsqueda binaria tiene una complejidad de  $O(\log n)$  y se ejecuta  $n$  veces en el ciclo de repetición. Además, esto se ejecuta para cada una de las  $q$  consultas, por lo tanto, la complejidad en tiempo de ejecución es  $O(q \cdot n \cdot \log n)$ .

Tomando los máximos valores para las variables  $q$  y  $n$ , la cantidad de operaciones es del orden de  $10 \cdot (5 \cdot 10^5) \log (5 \cdot 10^5)$ , donde  $10 \cdot (5 \cdot 10^5) \log (5 \cdot 10^5) \leq 5 \cdot (10 \cdot 10^5) \log (10^6) = 5 \cdot 10^6 \cdot 6 = (6 \cdot 5) \cdot 10^6 = 30 \cdot 10^6 = 3 \cdot 10 \cdot 10^6 = 3 \cdot 10^7 \leq 10^8$ .

### 2.4.2. Colección dinámica

**Nombre original:** Dynamic Collection<sup>10</sup>.

**Fuente:** Gran Premio de México 2023 - Primera Fecha.

**Fecha:** 13 de Mayo de 2023.

**Autor:** Hugo Humberto Morales Peña.

Se cuenta con una colección de elementos, cada uno de los cuales es un número entero positivo (se permiten elementos repetidos en la colección). Sobre la colección se permiten dos tipos de operaciones: la operación 1, para actualizar la colección y la operación 2, para consultar la colección. El formato de estas operaciones es el siguiente:

- Operación 1: “1  $k$ ”
- Operación 2: “2  $a$   $b$ ”

Para la operación 1,  $k$  es un número entero positivo sobre el cual se realiza una de las siguientes tres acciones:

- Si el elemento  $k$  se encuentra en la colección, entonces no se hace nada, la colección permanece igual.
- Si el elemento  $k$  es más grande que cualquiera de los elementos en la colección, entonces se adiciona a la colección.
- Si el elemento  $k$  no se encuentra en la colección, entonces se reemplaza por  $k$  la primera ocurrencia del elemento más pequeño de la colección que sea mayor que  $k$ .

Para la operación 2,  $a$  y  $b$  son dos números enteros positivos ( $a \leq b$ ), sobre los cuales se debe consultar en la colección cuantos elementos están en el rango delimitado por  $[a, b]$ .

Por ejemplo, sea la colección que inicialmente tiene diez elementos:

$$C = [7, 1, 7, 1, 3, 9, 7, 9, 10, 4]$$

Después de la operación “2 2 8”, el resultado generado es 5.

Después de la operación “1 8”, la colección queda como:

$$C = [7, 1, 7, 1, 3, 8, 7, 9, 10, 4]$$

Ahora, al volver a lanzar la consulta “2 2 8”, el resultado generado es 6.

Después de la operación “1 20”, la colección queda con once elementos:

$$C = [7, 1, 7, 1, 3, 8, 7, 9, 10, 4, 20]$$

---

<sup>10</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/dynamic-collection>

### Formato de entrada:

La entrada consiste en un único caso de prueba. La primera línea contiene dos números enteros positivos  $n$   $q$  ( $1 \leq n, q \leq 10^6$ ), los cuales representan respectivamente la cantidad de elementos de la colección y el total de operaciones a realizarse sobre la colección. La segunda línea contiene exactamente  $n$  números enteros positivos, separados por un espacio en blanco  $c_1, c_2, \dots, c_n$  ( $1 \leq c_i \leq 10^9, 1 \leq i \leq n$ ). Las siguientes  $q$  líneas del caso de prueba son las operaciones, las cuales pueden ser del tipo 1 o del tipo 2, con el siguiente formato: “1  $k$ ” o “2  $a$   $b$ ” ( $1 \leq k, a, b \leq 10^9, a \leq b$ ).

### Formato de salida:

Para cada operación del tipo 2 (operación de consulta) en la entrada del caso de prueba, se debe imprimir una sola línea que contenga un número entero no negativo  $r$ , el cual representa el total de elementos en la colección que están en el rango  $[a, b]$ .

### Ejemplo de entrada:

```
10 11
7 1 7 1 3 9 7 9 10 4
2 2 8
1 8
2 2 8
2 1 20
1 20
2 1 20
2 7 12
1 5
2 7 12
1 12
2 7 12
```

### Ejemplo de salida:

```
5
6
10
11
6
5
6
```

### Solución del reto

Este reto de programación se puede resolver con estructuras de datos avanzadas, especializadas en el trabajo de segmentos o rangos como el SEGMENTTREE (las cuales están fuera del alcance de este libro). Lo único que realmente se necesita para resolver este reto, es el uso de un algoritmo de ordenamiento con una complejidad en el peor de los casos de  $O(n \cdot \log(n))$  (como lo es el algoritmo de ordenamiento por mezclas) y de las variantes del algoritmo de la búsqueda

binaria, para determinar las posiciones de la primera y de la última ocurrencia de un elemento.

La estrategia a utilizar para resolver el reto es la siguiente:

- Se leen y almacenan los  $n$  elementos de la colección en un arreglo; luego, se hace el llamado al algoritmo de ordenamiento por mezclas para que la información quede ordenada de forma ascendente.
- Luego, se leen una a una las  $q$  consultas:
  - Para la operación 1 de actualización, únicamente se necesita consultar la primera ocurrencia del elemento  $k$ ; si el elemento  $k$  se encuentra en el arreglo entonces no se hace nada, de lo contrario, se obtiene el punto de inserción y en dicha posición del arreglo  $C$  se almacena el valor  $k$ . Si el punto de inserción es más grande que la cantidad de elementos que están almacenado en el colección, entonces el tamaño de la colección se incrementa en uno. Se garantiza que el arreglo sigue estando ordenado de forma ascendente después de la actualización porque se cumple el invariante de la búsqueda binaria para el punto de inserción, donde  $C[puntoInsercion - 1] < k < C[puntoInsercion]$ .
  - Para la operación 2 de consulta de la cantidad de elementos en la colección en el rango  $[a, b]$ , se consultan la primera ocurrencia del elemento  $a$  y la última ocurrencia del elemento  $b$  con las variantes de la búsqueda binaria. Según sea el caso, si el elemento consultado no se encuentra en el arreglo  $C$ , entonces se ajusta el índice de comienzo o cierre del rango con respecto a los valores de los puntos de inserción, obteniendo el tamaño del rango al realizar la diferencia entre los índices de cierre y apertura del mismo.

La estrategia explicada previamente se implementa a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myInfinite 2147483647
#define MAXN 1000000
#define MAXQ 1000000

void myMerge(int A[], int p, int q, int r)
{
    int n1 = q - p + 1, n2 = r - q;
    int i, j, k, L[n1 + 2], R[n2 + 2];

    for(i = 1; i <= n1; i++)
        L[i] = A[p + i - 1];

    for(j = 1; j <= n2; j++)
        R[j] = A[q + j];

    L[n1 + 1] = myInfinite;
    R[n2 + 1] = myInfinite;
    i = 1;
    j = 1;
```

```
    for(k = p; k <= r; k++)
    {
        if(L[i] <= R[j])
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
    }

void MergeSort(int A[], int p, int r)
{
    int q;

    if(p < r)
    {
        q = (p + r) >> 1;
        MergeSort(A, p, q);
        MergeSort(A, q + 1, r);
        myMerge(A, p, q, r);
    }
}

int BinarySearch(int A[], int i, int j, int k)
{
    int m, result = -1;
    while(i <= j)
    {
        /* m = (i + j)/2; */
        m = (i + j)>>1;
        if(A[m] == k)
        {
            result = m;
            break;
        }
        else
        {
            if(k > A[m])
                i = m + 1;
            else
                j = m - 1;
        }
    }
    if(result == -1)
        result = (-1) * i - 1;

    return result;
}

int BinarySearchFirstOccurrence(int A[], int i, int j, int k)
{
    int result, result2;
    result = BinarySearch(A, i, j, k);
```

```

    if(result >= 0)
    {
        result2 = BinarySearch(A, i, result - 1, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, i, result - 1, k);
        }
    }

    return result;
}

int BinarySearchLastOccurrence(int A[], int i, int j, int k)
{
    int result, result2;
    result = BinarySearch(A, i, j, k);

    if(result >= 0)
    {
        result2 = BinarySearch(A, result + 1, j, k);
        while(result2 >= 0)
        {
            result = result2;
            result2 = BinarySearch(A, result + 1, j, k);
        }
    }

    return result;
}

int main()
{
    int n, q, idElement, idQuery, operation, k, a, b;
    int indexLeft, indexRight, sizeCollection;
    int insertionPoint;
    int C[MAXN + MAXQ + 1];

    scanf("%d %d", &n, &q);
    sizeCollection = n;

    for(idElement = 1; idElement <= n; idElement++)
        scanf("%d", &C[idElement]);

    MergeSort(C, 1, sizeCollection);

    for(idQuery = 1; idQuery <= q; idQuery++)
    {
        scanf("%d", &operation);
        if(operation == 1)
        {
            scanf("%d", &k);
            indexLeft = BinarySearchFirstOccurrence(C, 1, sizeCollection, k
                );
            if(indexLeft < 0)
            {
                insertionPoint = -1 * (indexLeft + 1);
                if(insertionPoint <= sizeCollection)
                    C[insertionPoint] = k;
            }
        }
    }
}

```

```
        else
    {
        sizeCollection++;
        C[sizeCollection] = k;
    }
}
else
{
    if(operation == 2)
    {
        scanf("%d %d", &a, &b);
        indexLeft = BinarySearchFirstOccurrence(C, 1,
                                                sizeCollection, a);
        indexRight = BinarySearchLastOccurrence(C, 1,
                                                sizeCollection, b);
        if(indexLeft < 0)
            indexLeft = -1 * (indexLeft + 1);
        if(indexRight < 0)
            indexRight = -1 * (indexRight + 2);
        printf("%d\n", indexRight - indexLeft + 1);
    }
}
return 0;
}
```

Salida del programa anterior ejecutando los ejemplos del reto:

```
10 11
7 1 7 1 3 9 7 9 10 4
2 2 8
5
1 8
2 2 8
6
2 1 20
10
1 20
2 1 20
11
2 7 12
6
1 5
2 7 12
5
1 12
2 7 12
6
```

Figura 2.6. Salida del programa para el reto: Colección dinámica.

El costo computacional de la solución anterior se encuentra en el llamado al algoritmo de ordenamiento por mezclas, para ordenar los  $n$  elementos que originalmente se encuentran en la colección más el fragmento de código en el cual se atienden una a una las  $q$  operaciones a realizar sobre la colección. Esto tiene una complejidad en tiempo de ejecución de  $O(n \cdot (\log(n))) + O(q \cdot (\log(n + q)))$ .

Tomando los máximos valores para las variables  $n$  y  $q$ , la cantidad de operaciones es del orden de:

$$\begin{aligned}
 n \cdot (\log(n)) + q \cdot (\log(n + q)) &= 10^6 \cdot (\log(10^6)) + 10^6 \cdot (\log(10^6 + 10^6)) \\
 &= 10^6 \cdot (6) + 10^6 \cdot (\log(2 \cdot 10^6)) \\
 &\leq 10^6 \cdot (6) + 10^6 \cdot (\log(10^7)) \\
 &= 10^6 \cdot (6) + 10^6 \cdot (7) \\
 &= 10^6 \cdot (6 + 7) \\
 &= 10^6 \cdot (13) \\
 &= 13 \cdot 10^6 \\
 &= (1,3 \cdot 10) \cdot 10^6 \\
 &= 1,3 \cdot (10 \cdot 10^6) \\
 &= 1,3 \cdot 10^7 \\
 &\leq 10^8.
 \end{aligned}$$

## 2.5. Retos de programación propuestos

En esta sección, se propone una lista de retos de programación que se pueden resolver con el uso de los algoritmos de ordenamiento y/o búsqueda binaria:

- Centro numérico
- Tobby y los tanques II
- Taza de café<sup>11</sup> <sup>12</sup>
- Teorema del número poligonal de Fermat
- Frecuencias de subconjuntos
- Tobby y las galletas
- Diccionario
- Generar, ordenar y buscar

---

<sup>11</sup>Como se indica en [MS07], no es necesario que los valores de  $A[i]$  estén almacenados en un arreglo. Solamente se necesita la capacidad de poder calcular  $A[i]$  dado  $i$ . Por ejemplo, si se tiene una función  $f$  monótonamente creciente y argumentos  $i$  y  $j$  con  $f(i) < x < f(j)$ , se puede usar la búsqueda binaria para encontrar  $m$  con  $f(m) \leq x < f(m + 1)$ . En este contexto, la búsqueda binaria es a menudo referenciada como el Método de Bisección.

<sup>12</sup>Se puede revisar el Método de Bisección en [CC21].

### 2.5.1. Centro numérico

**Nombre original:** Numeric Center<sup>13</sup> <sup>14</sup>.

**Fuente:** UTP Open 2015.

**Fecha:** 11 de Abril de 2015.

**Autor:** Hugo Humberto Morales Peña.

Un centro numérico es un número que separa una lista de números enteros positivos consecutivos (comenzando en 1) en dos grupos de números enteros positivos consecutivos, cuyas sumas son iguales. El primer centro numérico es el 6, el cual separa la lista de números enteros positivos consecutivos {1, 2, 3, 4, 5, 6, 7, 8} en los grupos: {1, 2, 3, 4, 5} y {7, 8}, cuyas sumas son ambas iguales a 15. El segundo centro numérico es el 35, el cual separa la lista de números enteros positivos consecutivos {1, 2, 3, 4, ..., 49} en los grupos: {1, 2, 3, 4, ..., 34} y {36, 37, 38, 39, ..., 49}, cuyas sumas son ambas iguales a 595.

Escribir un programa que calcule el total de centros numéricos que hay entre 1 y  $n$ .

#### Formato de entrada:

La entrada puede contener varios casos de prueba. Cada caso está compuesto de una sola línea, que contiene un número entero positivo  $n$  ( $1 \leq n \leq 10^6$ ) que representa el número hasta el cual se deben calcular los centros numéricos. La entrada finaliza con un caso de prueba que contiene un valor de  $n$  igual a cero, el cual no debe ser procesado.

#### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea el número que representa el total de centros numéricos que se encuentran en la lista desde 1 hasta  $n$ .

#### Ejemplo de entrada:

```
7
8
48
49
50
0
```

#### Ejemplo de salida:

```
0
1
1
2
2
```

---

<sup>13</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/numeric-center-i>

<sup>14</sup>Este reto de programación se generó a partir del “punto 6 de los ejercicios propuestos en el Capítulo 6” del libro *Java 2: Curso de Programación* [Ceb03]

### 2.5.2. Tobby y los tanques II

**Nombre original:** Tobby and Tanks (II)<sup>15</sup>.

**Fuente:** UTP Open 2014.

**Fecha:** 26 de Abril de 2014.

**Autor:** Jhon Jimenez.

Tobby es un perro curioso que siempre está pensando en cosas nuevas y en ideas revolucionarias. Ahora, él se imagina  $N$  tanques ubicados en una fila escalonada (ver Figura 2.7); cada tanque tiene una capacidad de  $W$  litros de agua. Tobby se pregunta, si se tienen  $K$  litros de agua, ¿cuál es la posición del tanque más lejano?, tal que si se chorrea (se vacía) toda el agua sobre este, el agua alcance a llegar al primer tanque, al menos un litro. No importa si el primer tanque se rebosa por completo y se riega agua.

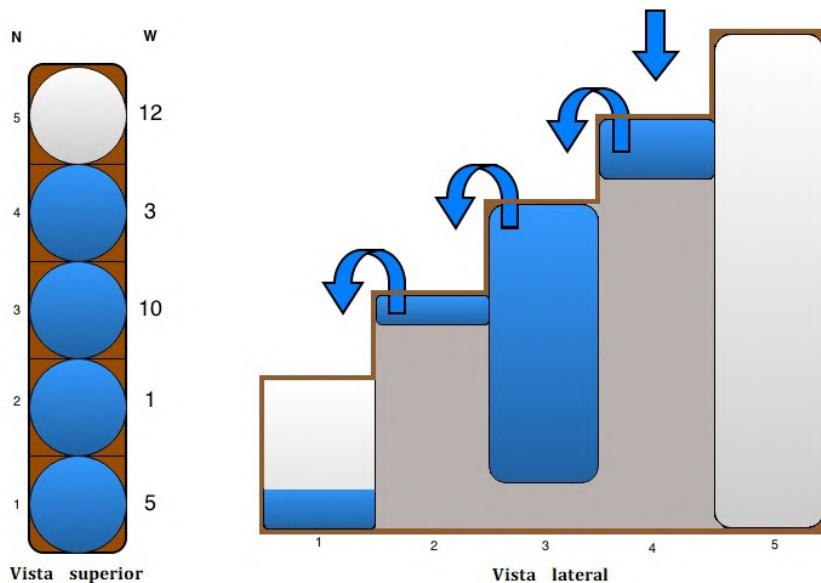


Figura 2.7. Vistas superior y lateral de una fila escalonada de tanques.

Tobby tiene muchas responsabilidades familiares y no tiene tiempo para resolver este interrogante, por ese motivo, necesita de su ayuda, para lo cual a usted le dan  $n$  tanques con capacidad  $w_i$ , y  $q$  consultas, cada consulta contiene un número entero  $k$ , el cual representa la cantidad de agua.

#### Formato de entrada:

La entrada contiene múltiples casos de prueba. Para cada caso de prueba, la primera línea contiene dos números enteros positivos  $n$  ( $1 \leq n \leq 10^5$ ) y  $q$  ( $1 \leq q \leq 10^4$ ), el número de tanques y el número de consultas respectivamente, la siguiente línea contiene  $n$  números enteros positivos  $w_i$  ( $1 \leq w_i \leq 10^4$ ,  $1 \leq i \leq n$ ), que representan las capacidades en litros de los  $n$  tanques. La siguiente línea contiene  $q$  números enteros positivos  $k_j$  ( $1 \leq k_j \leq 10^9$ ,  $1 \leq j \leq q$ ), que representan cada una de las  $q$  consultas con la cantidad de agua en litros.

<sup>15</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/tobby-and-tanks-ii>

Leer casos de prueba en la entrada hasta que se alcance el fin de archivo (EOF - End Of File).

**Formato de salida:**

Por cada caso de prueba, imprimir una sola línea que contenga  $q$  números enteros positivos, que representan las posiciones de los tanques desde los cuales se debe poner a derramar el agua para cada una de las  $q$  consultas.

**Ejemplo de entrada:**

```
5 2
5 1 10 3 12
16 1
```

**Ejemplo de salida:**

```
4 1
```

### 2.5.3. Taza de café

**Nombre Original:** Café<sup>16</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Hugo Humberto Morales Peña.

El profesor Humbertov Moralov tiene la fortuna de trabajar en una de las mejores Universidades del Eje Cafetero en Colombia. Obviamente, su bebida favorita para comenzar el día es una buena taza de café, sin embargo, tiene una obsesión con respecto a mezclar el café con agua, donde debe quedar un 50 % de café y un 50 % de agua. Si no se respeta esto, entonces el profesor considerará que el café esta muy aguado o muy espeso.

La taza de café del profesor Moralov (ver Figura 2.8) tiene la forma de un cono truncado volteado (ver Figura 2.9), con radio de la base  $r$  (o radio del fondo de la taza), radio de la parte superior  $R$  (o radio de la boca de la taza) y altura  $h$ .



Figura 2.8. Taza de café del profesor Humbertov Moralov.

El grupo de fans del profesor Moralov (los cuales se hacen llamar los Moralovers) se ha dado cuenta de la problemática que vive día a día su profesor con la taza de café y necesitan de tu ayuda para que por favor le indiques hasta donde debe de llenar de café su taza para que la combinación con el agua sea perfecta.

Los Moralovers te facilitan la siguiente información, que podría llegar a ser de utilidad:

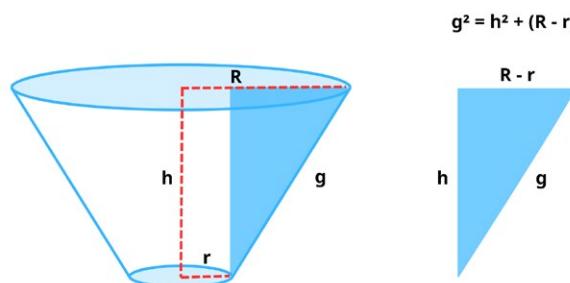


Figura 2.9. Cono truncado.

<sup>16</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/cup-of-coffee>

La fórmula para calcular el volumen del cono truncado es:

$$V = \frac{1}{3} \cdot \pi \cdot h \cdot (R^2 + r^2 + R \cdot r)$$

**Formato de Entrada:**

La entrada comienza con un número entero  $t$  ( $1 \leq t \leq 5 \cdot 10^4$ ) el cual representa el total de casos de prueba. Cada uno de los casos de prueba consta de tres números reales  $r$   $R$   $h$  ( $1,5 \leq r \leq 4,0$ ,  $2,5 \leq R \leq 6,0$ ,  $7,0 \leq h \leq 10,0$ ,  $r \leq R$ ), los cuales representan respectivamente el radio del fondo, el radio de la boca y la altura de la taza en centímetros.

**Formato de Salida:**

Por cada caso de prueba, imprimir en una sola línea un número real que representa la altura hasta la cual el profesor Moralov debe de llenar su taza de café para obtener la combinación perfecta. La altura (en centímetros) será considerada correcta si el error absoluto o relativo no excede al  $10^{-6}$ .

**Ejemplo de Entrada:**

```
4
2.5 3.8 8.8
2.0 4.0 10.0
1.5 2.5 9.0
3.0 3.0 8.8
```

**Ejemplo de Salida:**

```
5.271806416
6.509636245
5.561206129
4.400000000
```

### 2.5.4. Teorema del número poligonal de Fermat

**Nombre original:** Triangular Test II<sup>17</sup>.

**Fuente:** UTP Open 2016

**Fecha:** 9 de Abril de 2016.

**Autor(es):** Alejandro Moreno Agudelo, Hugo Humberto Morales Peña.

En 1638, Fermat propuso que cada número entero positivo es la suma de como máximo tres números triangulares, cuatro números cuadrados, cinco números pentagonales, y  $n$  números  $n$ -poligonales. Esto es lo que se conoce como el Teorema del número poligonal de Fermat<sup>18</sup>.

Apoyados en el Teorema del número poligonal de Fermat, en este reto se debe calcular la cantidad mínima de números triangulares que se deben sumar para obtener un número entero positivo  $n$ .

#### Formato de entrada:

La entrada del problema consiste en varios casos de prueba. Cada caso de prueba es presentado en una sola línea que contiene un número entero positivo  $n$  ( $1 \leq n \leq 3 \cdot 10^6$ ). El final de los casos de prueba es dado por el fin de archivo (End Of File - EOF).

#### Formato de salida:

Para cada caso de prueba, su programa debe imprimir un número entero positivo que denota la mínima cantidad de números triangulares, cuya suma es igual a  $n$ . Cada caso de prueba debe generar una sola línea en la salida.

#### Ejemplo de entrada:

```

1
2
3
4
5
6
7
8
9
10
11
49

```

---

<sup>17</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/triangular-test-ii>

<sup>18</sup>En [GM98] se hace una buena recopilación histórica de los números poligonales.

**Ejemplo de salida:**

```
1
2
1
2
3
1
2
3
2
1
2
2
```

### 2.5.5. Frecuencias de subconjuntos

**Nombre original:** Subset Frequencies<sup>19</sup>.

**Fuente:** Coding Cup ITSUR 2018.

**Fecha:** 18 de Abril de 2018.

**Autor:** Hugo Humberto Morales Peña.

Se tienen dos conjuntos  $A$  y  $S$  de números enteros positivos, el conjunto  $A$  tiene cardinalidad  $n$  ( $|A| = n$ ) y el conjunto  $S$  tiene cardinalidad  $k$  ( $|S| = k$ ); es importante recordar que la cardinalidad de un conjunto es el total de elementos distintos que este contiene.

La tarea a realizar consiste en lo siguiente: para cada uno de los  $s_i$  elementos del conjunto  $S$  se quiere averiguar cuantos subconjuntos de tamaño dos en el conjunto  $A$  tienen una suma de sus elementos igual a el. Para hacer esto aun más interesante, la salida debe cumplir el ordenamiento que se pide en la sección “Formato de salida”.

#### Formato de entrada:

La entrada del problema consiste de un único caso de prueba. La primera línea del caso de prueba contiene dos números enteros positivos  $n$  y  $k$  ( $3 \leq n \leq 10^6$ ,  $1 \leq k \leq 50$ ), que representan respectivamente las cardinalidades de los conjuntos  $A$  y  $S$ . Luego, la segunda línea del caso de prueba contiene los  $n$  números enteros positivos  $a_i$  del conjunto  $A$  ( $1 \leq a_i \leq 10^8$ ,  $1 \leq i \leq n$ ); obviamente se garantiza que los  $n$  elementos del conjunto  $A$  son diferentes. La tercera línea del caso de prueba contiene los  $k$  números enteros positivos  $s_j$  del conjunto  $S$  ( $1 \leq s_j \leq 2 \times 10^8$ ,  $1 \leq j \leq k$ ), se garantiza que los  $k$  elementos del conjunto  $S$  son diferentes.

#### Formato de salida:

Su programa debe imprimir  $k$  líneas, cada una de ellas conteniendo dos números enteros positivos  $s_i$   $f_i$  ( $1 \leq i \leq k$ ) los cuales representan el valor de la suma de los dos elementos del subconjunto de tamaño dos y la frecuencia de ocurrencia de dichos subconjuntos sobre el conjunto  $A$ .

**Nota:** las  $k$  líneas de la salida del programa deben estar ordenadas de forma descendente con respecto a el valor  $f_i$ , en el caso de líneas que tenga el mismo valor de  $f_i$ , entonces se debe ordenar de forma ascendente con respecto al valor del  $s_i$ .

#### Ejemplo de entrada:

```
6 3
6 5 1 4 2 3
7 8 6
```

#### Ejemplo de salida:

```
7 3
6 2
8 2
```

---

<sup>19</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/subset-frequencies>

**Explicación:**

Para el caso de prueba del ejemplo se tienen los conjuntos  $A = \{6, 5, 1, 4, 2, 3\}$  y  $S = \{7, 8, 6\}$ , para los cuales se tienen tres subconjuntos de tamaño dos cuya suma es igual a  $s_1 = 7$ , estos son  $\{1, 6\}$ ,  $\{2, 5\}$  y  $\{3, 4\}$ . Se tienen dos subconjuntos de tamaño dos cuya suma es igual a  $s_2 = 8$ , estos son  $\{2, 6\}$  y  $\{3, 5\}$ . Por último, se tienen dos subconjuntos de tamaño dos cuya suma es igual a  $s_3 = 6$ , estos son  $\{1, 5\}$  y  $\{2, 4\}$ .

### 2.5.6. Tobby y las galletas

**Nombre original:** Tobby y las galletas<sup>20</sup>.

**Fuente:** Maratón Interna de Programación UTP 2019.

**Fecha:** 28 de Noviembre de 2019.

**Autor:** Yeferson Gaitán Gómez.

Tobby tiene mucha hambre, así que fue a la nevera por algo de comer y sacó una caja de galletas de muchos tipos (la caja puede ser vista como un arreglo de  $n$  elementos), Tobby sabe que si come muchas galletas su mamá se dará cuenta, así que decidió comer solo las galletas que son de un tipo especial, aquellas que con respecto a su peso son menores o iguales a  $x$  gramos. Tobby tiene muchas consultas, para cada consulta se tiene que responder cuántas galletas se comerá él.

#### Formato de entrada:

La entrada contiene múltiples casos de prueba. Cada caso de prueba contiene cuatro líneas, donde la primera línea contiene un número entero positivo  $n$  ( $1 \leq n \leq 10^5$ ), que representa la cantidad de unidades (es decir, la cantidad de galletas) que contiene la caja de galletas. La segunda línea contiene  $n$  números enteros positivos separados por un espacio en blanco, los cuales representan los pesos de las  $n$  galletas de la caja; cada uno de los pesos de las galletas pertenecen al intervalo cerrado  $[1, 10^9]$ . La tercera línea contiene un número entero positivo  $q$  ( $1 \leq q \leq 10^5$ ), que representa el total de consultas que tiene que contestar Tobby. La cuarta y última línea del caso de prueba contiene  $q$  números enteros positivos separados por un espacio en blanco, los cuales representan los pesos consultados, cada uno de los pesos pertenecen al intervalo cerrado  $[1, 10^9]$ . Se deben leer casos de prueba hasta alcanzar el fin de archivo (EOF - End Of File).

#### Formato de salida:

Para cada caso de prueba de la entrada, se debe imprimir una línea conteniendo  $q$  números enteros positivos separados por un espacio en blanco, los cuales representan el total de galletas que se puede comer Tobby en cada una de las consultas evaluadas.

#### Ejemplo de entrada:

```
10
4 2 4 5 3 1 5 3 3 4
5
4 2 5 3 1
10
21 4 32 1 16 9 30 12 6 10
4
20 10 15 26
```

---

<sup>20</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/toby-y-las-galletas-1>

**Ejemplo de salida:**

8 2 10 5 1  
7 5 6 8

### 2.5.7. Diccionario

**Nombre original:** Dictionary<sup>21</sup>.

**Fuente:** X Maratón UFPS + RPC y I Maratón Interuniversitaria

**Fecha:** 25 de Noviembre de 2023.

**Autor:** Hugo Humberto Morales Peña.

Se cuenta con un diccionario de palabras (donde obviamente no hay palabras repetidas). Sobre el diccionario, se quiere consultar la cantidad de palabras que en el ordenamiento lexicográfico se encuentran en el rango de dos palabras.

Por ejemplo, si se tiene un diccionario con el siguiente conjunto de palabras {camila, angel, margot, bambi, sabrina, mabel, celia}, y se pregunta por la cantidad de palabras que están en el ordenamiento lexicográfico en el rango de las palabras “candy” y “patrick”, la respuesta sería 3, porque las palabras que están en el rango en el ordenamiento lexicográfico son “celia”, “mabel” y “margot”.

#### Formato de entrada:

La entrada consiste de un único caso de prueba. La primera línea contiene dos números enteros positivos  $n$   $q$  ( $1 \leq n \leq 10^6$ ,  $1 \leq q \leq 5 \cdot 10^5$ ), que representan, respectivamente, la cantidad de palabras en el diccionario y el total de consultas a realizarse sobre el diccionario. Luego, se presentan  $n$  líneas, cada una conteniendo una palabra de máximo 10 letras minúsculas en el alfabeto del idioma inglés. Por último, en el caso de prueba, son presentadas  $q$  líneas, cada una de ellas presentando dos palabras distintas  $w_1$  y  $w_2$  (separadas por un espacio en blanco), donde cada una es de máximo longitud 10 y donde se utilizan únicamente letras minúsculas del alfabeto del idioma inglés. Se garantiza que la palabra  $w_1$  esté primero que la palabra  $w_2$  en el orden lexicográfico.

#### Formato de salida:

La salida del problema debe contener  $q$  líneas, cada una de ellas conteniendo un número entero no negativo que representa la cantidad de palabras que están en el rango  $[w_1, w_2]$  en el ordenamiento lexicográfico de las palabras del diccionario.

#### Ejemplo de entrada:

```
7 9
camila
angel
margot
bambi
sabrina
mabel
celia
candy patrick
angel sabrina
```

---

<sup>21</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/dictionary-22>

candy conner  
angel vivian  
celia penny  
daisy margot  
daisy india  
ada lara  
lara penny

**Ejemplo de salida:**

3  
7  
1  
7  
3  
2  
0  
4  
2

### 2.5.8. Generar, ordenar y buscar

**Nombre original:** Generate, Sort and Search<sup>22</sup>.

**Fuente:** ICPC Bolivia 2016.

**Fecha:** 3 de Septiembre de 2016.

**Autor:** Hugo Humberto Morales Peña.

Sea la siguiente función recursiva:

$$\begin{aligned}f(1) &= x \\f(n) &= (a \cdot f(n-1) + c) \bmod m, \text{ para } n \geq 2, n \in \mathbb{Z}^+\end{aligned}$$

Nota: recordar que la operación *mod* calcula el residuo de la división entera.

Con la función recursiva anterior se deben generar una sucesión que contenga  $n$  términos, los cuales son:  $f(1), f(2), f(3), f(4), \dots, f(n)$ . Después, se deben ordenar de forma ascendente (con respecto a su valor) los  $n$  términos de la sucesión, para después poder contestar preguntas con respecto al término se encuentra en la sucesión ordenada en la posición  $i$ .

#### Formato de entrada:

Hay múltiples casos de prueba en la entrada. La primera línea de cada caso de prueba contiene seis números enteros positivos  $a, c, m, x, q, n$  separados por espacios ( $2 \leq a < m, 0 \leq c < m, 3 \leq m \leq 10^3, 0 \leq x < m, 1 \leq q \leq 10^4, 1 \leq n \leq 10^8$ ). Por último, en el caso de prueba, son presentadas  $q$  líneas, cada una de ellas conteniendo un número entero de la posición en la secuencia ordenada cuyo valor quiere ser conocido.

#### Formato de salida:

Para cada una de las  $q$  consultas se debe imprimir una sola línea que contenga el número entero que se encuentra en la posición  $i$ , la cual está siendo consultada en el ordenamiento de la sucesión que contiene los  $n$  términos.

#### Ejemplo de entrada:

```
7 4 9 3 5 10
2
10
3
9
4
```

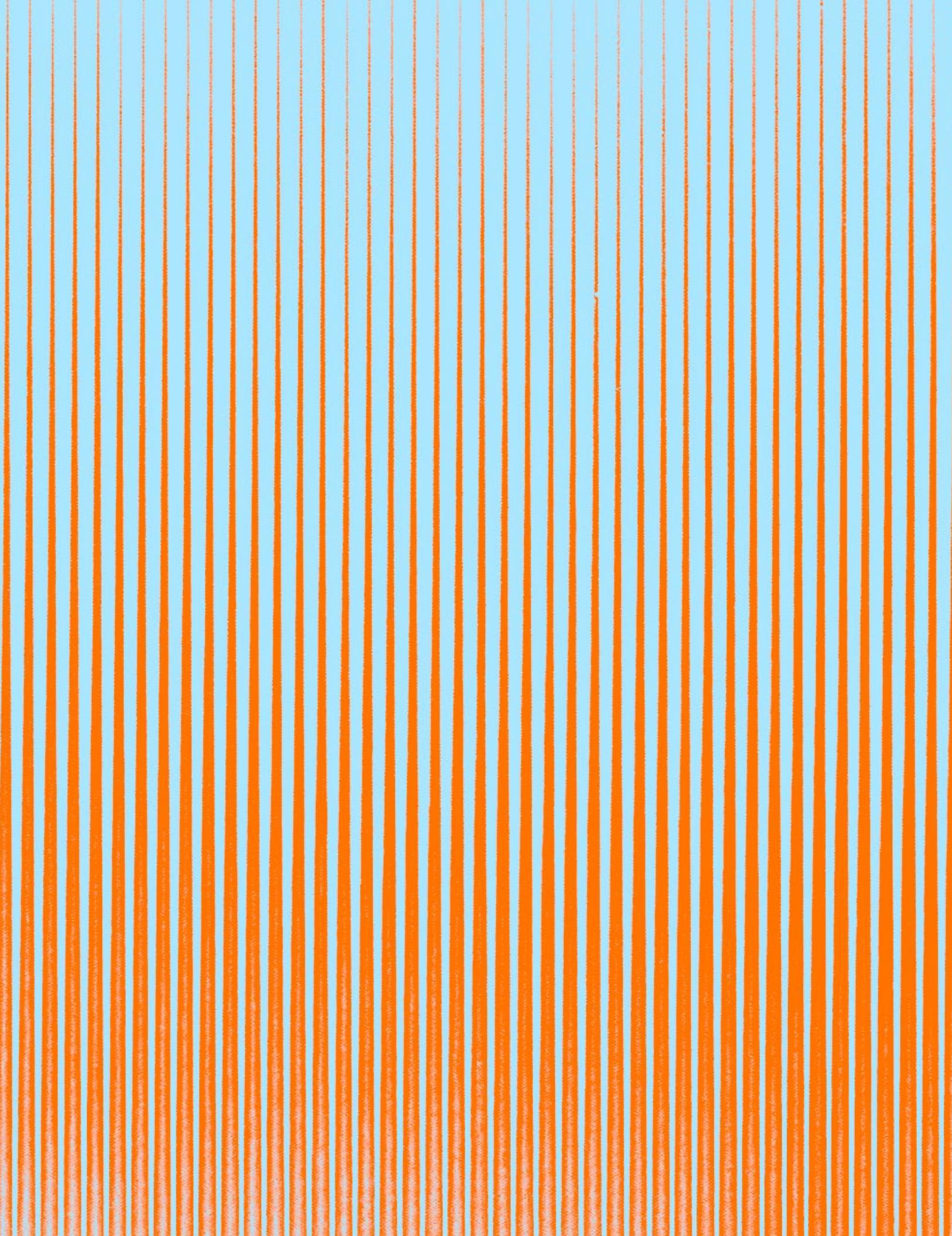
---

<sup>22</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/generar-ordenar-y-buscar>

**Ejemplo de salida:**

1  
8  
2  
7  
3





## **Capítulo 3**

# **Colas de prioridad**



En este capítulo se toma como punto de partida los pseudocódigos y teoría de las estructuras de datos montón máximo y colas de prioridad máxima presentados en [Cor+01], para luego ajustarlos a sus versiones de montón mínimo y colas de prioridad mínima.

Este capítulo comienza con el tema de la estructura de datos montón, con sus propiedades de forma y orden. Gracias a la propiedad de forma de un montón, se establecen las funciones para determinar los indices (posiciones en un arreglo) del padre, del hijo por izquierda y del hijo por derecha del elemento ubicado en la posición  $i$ . Se presenta también la función que garantiza la propiedad de orden en los elementos de un montón.

Luego se presentan las funciones que trabajan sobre la estructura de datos montón y permiten el manejo de las colas de prioridad.

El capítulo continúa con el análisis, diseño e implementación (con eficiencia computacional en tiempo de ejecución y almacenamiento en espacio) de tres retos de programación utilizando colas de prioridad mínima.

Finalizando el capítulo, se plantean 8 retos de programación para que los lectores no solamente programen las soluciones utilizando colas de prioridad, sino para que también las validen en el juez en línea HackerRank<sup>1</sup>.

### 3.1. Estructura de datos montón

El algoritmo de Ordenamiento por Montones (Heap Sort) creado por Williams en 1964 [Wil64] es un algoritmo excelente, aunque el Quicksort es mejor en la práctica. Sin embargo, la estructura de datos montón (heap) tiene muchos usos. El uso más importante que se le da a la estructura de datos montón es la implementación de forma eficiente de colas de prioridad (priority queue). A continuación, se presentan las propiedades de forma y orden de los montones y la función que garantiza la propiedad de orden de montón mínimo.

---

<sup>1</sup><https://www.hackerrank.com/data-structure-utp>

### 3.1.1. Las propiedades de forma y orden en un montón

La estructura de datos montón es un arreglo de “objetos” muy parecido a un árbol binario completo. Cada nodo del árbol corresponde a un elemento del arreglo que almacena el valor en el nodo. La propiedad de forma de un montón indica que el árbol está completamente lleno en todos los niveles excepto, posiblemente, en el nivel más bajo, el cual es llenado de izquierda a derecha hasta algún punto. La Figura 3.1 representa la situación anteriormente planteada.

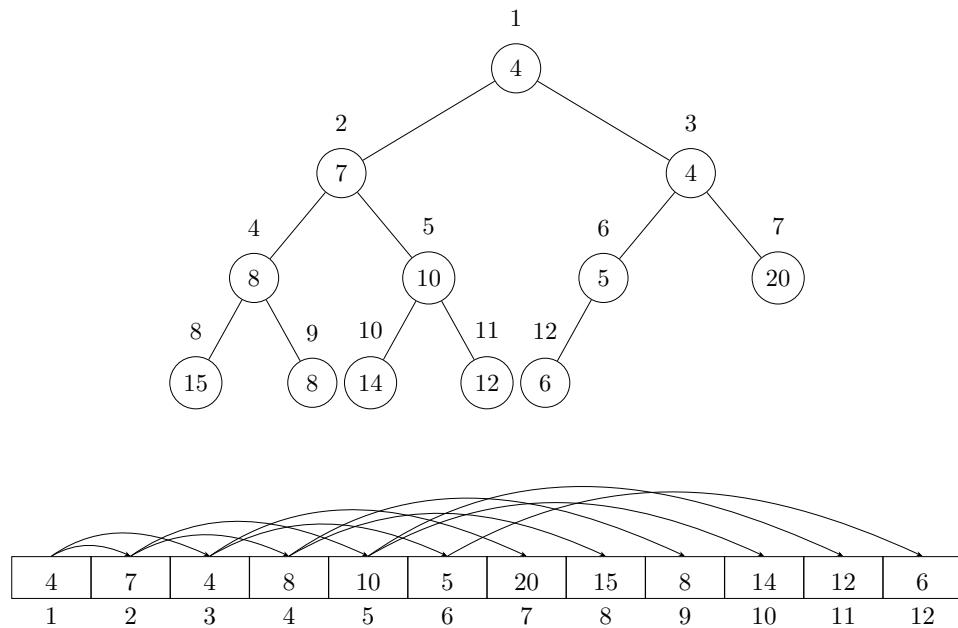


Figura 3.1. Representaciones de un montón mínimo (árbol y arreglo).

Sobre el arreglo  $Q$  que representa un montón tenemos dos valores, dos medidas:  $length[Q]$ , el cual es el tamaño del arreglo  $Q$ , y  $heapSize$ , el cual es el número de elementos del montón almacenados en las primeras posiciones del arreglo  $Q$ , donde  $heapSize \leq length[Q]$ .

Hay dos clases de montones binarios: el *montón máximo* y el *montón mínimo*. En ambos casos, los valores de los nodos satisfacen una propiedad de orden de un montón, en el caso de un *montón máximo*, la propiedad de orden del montón máximo es que para todo nodo  $i$  diferente de la raíz se cumple que  $Q[\text{PARENT}(i)] \geq Q[i]$ , esto es que el valor de cada nodo como máximo es el valor de su padre, de esta forma, el elemento más grande de un montón máximo está almacenado en la raíz; en el caso de un *montón mínimo*, la propiedad de orden del montón mínimo es que para todo nodo  $i$  diferente de la raíz se cumple que  $Q[\text{PARENT}(i)] \leq Q[i]$ , esto es que el valor de cada nodo como mínimo es el valor de su padre, de esta forma, el elemento más pequeño de un montón mínimo está almacenado en la raíz.

El contenido de la raíz del árbol está almacenado en  $Q[1]$ , y dado el índice de un nodo, el índice de su padre  $\text{PARENT}(i)$ , el índice del hijo por la izquierda  $\text{LEFT}(i)$  y el índice del hijo por la derecha  $\text{RIGHT}(i)$  pueden ser calculados simplemente con las siguientes funciones:

```

1: function PARENT( $i$ )
2:   return  $\lfloor \frac{i}{2} \rfloor$ 
3: end function

```

```

1: function LEFT( $i$ )
2:   return  $2 * i$ 
3: end function

```

```

1: function RIGHT( $i$ )
2:   return  $2 * i + 1$ 
3: end function

```

Al mirar un montón como un árbol, se define la *altura* de un nodo en el montón como la cantidad de aristas que hay desde el nodo a alguna de las hojas más lejanas que se alcanzan desde él en una ruta simple descendente, de esta forma, la altura del montón es la altura del nodo raíz del árbol. La altura de un montón es  $\Theta(\log n)$ .

### 3.1.2. Función para garantizar la propiedad de orden de montón

Los parámetros de entrada de la función MINHEAPIFY son un arreglo  $Q$  y un subíndice  $i$  sobre el arreglo. La función MINHEAPIFY asume que tanto el subárbol izquierdo como el subárbol derecho, a partir de la posición  $i$ , son montones mínimos, pero que  $Q[i]$  puede ser más grande que alguno de sus dos hijos, con lo que se violaría la propiedad de montón mínimo. La función MINHEAPIFY le permite al valor almacenado en  $Q[i]$  “sumergirse” en el montón mínimo hasta lograr que el subárbol con raíz  $i$  sea un montón mínimo.

```

1: function MINHEAPIFY( $Q[ ]$ ,  $i$ )
2:    $l = \text{LEFT}(i)$ 
3:    $r = \text{RIGHT}(i)$ 
4:   if  $l \leq \text{heapSize}$  and  $Q[l] < Q[i]$  then
5:      $least = l$ 
6:   else
7:      $least = i$ 
8:   end if
9:   if  $r \leq \text{heapSize}$  and  $Q[r] < Q[least]$  then
10:     $least = r$ 
11:   end if
12:   if  $least \neq i$  then
13:     exchange  $Q[i]$  with  $Q[least]$ 
14:     MINHEAPIFY( $Q[ ]$ ,  $least$ )
15:   end if
16: end function

```

Si  $Q[i]$  es menor o igual que la información almacenada en la raíz de los subárboles izquierdo y derecho, entonces el árbol con raíz el nodo  $i$  es un montón mínimo y la función termina. De lo contrario, la raíz de alguno de los subárboles tiene información menor que la que se encuentra en  $Q[i]$  y es intercambiada con ésta, con lo cual se garantiza que el nodo  $i$  y sus hijos cumplen

la propiedad de montón mínimo, sin embargo, el subárbol hijo con el cual se intercambio la información de  $Q[i]$  ahora puede no cumplir la propiedad de montón mínimo, por lo tanto, se debe llamar de forma recursiva a la función MINHEAPIFY sobre el subárbol hijo con el cual se hizo el intercambio.

La complejidad en el peor de los casos de la función MINHEAPIFY es  $O(h)$ , donde  $h$  es la altura del montón, como la altura del montón es  $\Theta(\log n)$ , entonces MINHEAPIFY es  $O(\log n)$ .

### Ejemplo 1:

Realizar el paso a paso del algoritmo MINHEAPIFY( $Q[ ]$ , 1), cuando el arreglo  $Q$  tiene los elementos:

$Q[i]$	24	7	4	8	10	5	20	15	8	14	12	6
$i$	1	2	3	4	5	6	7	8	9	10	11	12

Y tiene su equivalencia gráfica por medio de árbol binario como en la Figura 3.2.

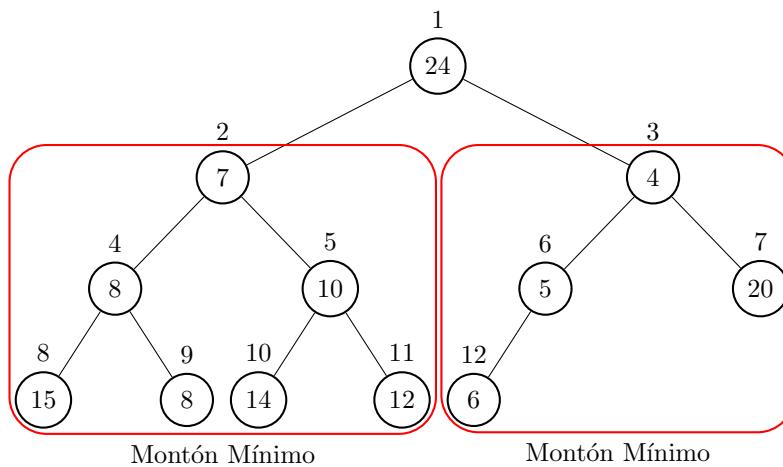


Figura 3.2. Representación en árbol binario del arreglo  $Q$ .

### Paso a paso del algoritmo MINHEAPIFY

MINHEAPIFY( $Q[ ]$ , 1)

$i$	$l$	$r$	$heapSize$	$least$
1	2	3	12	2
				3

$Q[i]$	24	7	4	8	10	5	20	15	8	14	12	6
$i$	1	2	3	4	5	6	7	8	9	10	11	12
$Q[i]$	4	7	24	8	10	5	20	15	8	14	12	6
$i$	1	2	3	4	5	6	7	8	9	10	11	12

MINHEAPIFY( $Q[ ]$ , 3)

<i>i</i>	<i>l</i>	<i>r</i>	<i>heapSize</i>	<i>least</i>
3	6	7	12	6
$Q[i]$	4	7	<b>24</b>	8
$i$	1	2	<b>3</b>	4
$Q[i]$	4	7	<b>5</b>	8
$i$	1	2	<b>3</b>	4

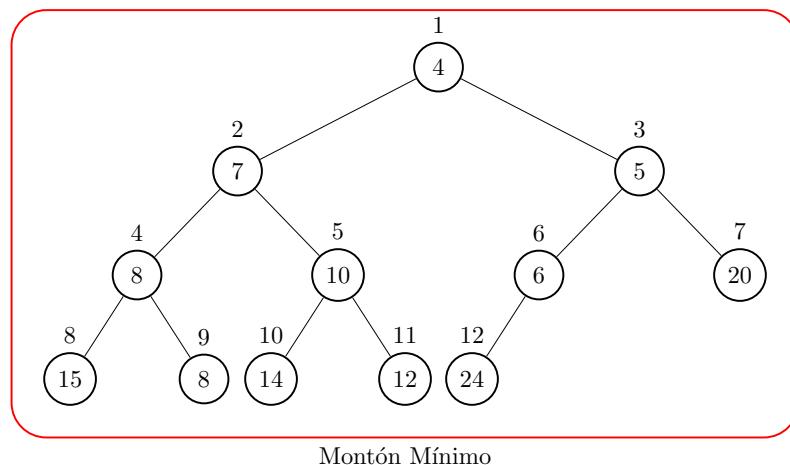
MINHEAPIFY( $Q[ ]$ , 6)

<i>i</i>	<i>l</i>	<i>r</i>	<i>heapSize</i>	<i>least</i>
6	12	13	12	12
$Q[i]$	4	7	5	8
$i$	1	2	3	4
$Q[i]$	4	7	5	8
$i$	1	2	3	4

MINHEAPIFY( $Q[ ]$ , 12)

<i>i</i>	<i>l</i>	<i>r</i>	<i>heapSize</i>	<i>least</i>
12	24	25	12	12
$Q[i]$	4	7	5	8
$i$	1	2	3	4

Al finalizar el paso a paso del algoritmo MINHEAPIFY obtenemos un montón mínimo desde la posición 1, el cual tiene su equivalencia gráfica por medio de árbol binario como en la Figura 3.3.

Figura 3.3. Representación en árbol binario del montón almacenado en el arreglo  $Q$ .

### 3.2. Colas de prioridad

Como en los montones, las colas de prioridad son de dos tipos: colas de prioridad máxima y colas de prioridad mínima. En este curso trabajaremos las colas de prioridad mínima.

En una cola de prioridad mínima el elemento más pequeño es el que se atiende primero.

Una cola de prioridad mínima soporta las siguientes operaciones:

- MINPQ\_INSERT( $Q, key$ ): Inserta el elemento  $key$  en la cola de prioridad mínima  $Q$ .
- MINPQ\_MINIMUM( $Q$ ): Retorna el elemento más pequeño de la cola de prioridad mínima  $Q$ .
- MINPQ\_EXTRACT( $Q$ ): Remueve y retorna el elemento más pequeño de la cola de prioridad mínima  $Q$ .
- MINPQ\_DECREASEKEY( $Q, i, key$ ): Disminuye el valor del elemento ubicado en la posición  $i$  de la cola de prioridad  $Q$  al nuevo valor  $key$ . Se asume que el valor de  $key$  como máximo es el valor del elemento ubicado en la posición  $i$  de  $Q$ .

Las funciones que soportan las operaciones del manejo de colas de prioridad mínima son:

```
1: function MINPQ_MINIMUM( $Q[ ]$ )
2:   return  $Q[1]$ 
3: end function

1: function MINPQ_EXTRACT( $Q[ ]$ )
2:    $min = -\infty$ 
3:   if heapSize < 1 then
4:     error "heap underflow"
5:   else
6:      $min = Q[1]$ 
7:      $Q[1] = Q[heapSize]$ 
8:     heapSize = heapSize - 1
9:     MINHEAPIFY( $Q[ ]$ , 1)
10:   end if
11:   return min
12: end function

1: function MINPQ_DECREASEKEY( $Q[ ], i, key$ )
2:   if key >  $Q[i]$  then
3:     error "new key is higher than current key"
4:   else
5:      $Q[i] = key$ 
6:     while  $i > 1$  and  $Q[\text{PARENT}(i)] > Q[i]$  do
7:       exchange  $Q[i]$  with  $Q[\text{PARENT}(i)]$ 
8:        $i = \text{PARENT}(i)$ 
9:     end while
```

```

10:    end if
11: end function

1: function MINPQ_INSERT( $Q[ ]$ ,  $key$ )
2:      $heapSize = heapSize + 1$ 
3:      $Q[heapSize] = \infty$ 
4:     MINPQ_DECREASEKEY( $Q[ ]$ ,  $heapSize$ ,  $key$ )
5: end function

```

La función MINPQ\_MINIMUM tiene una complejidad en tiempo de ejecución de  $O(1)$ . Las funciones MINPQ\_EXTRACT, MINPQ\_DECREASEKEY y MINPQ\_INSERT tienen complejidad en tiempo de ejecución en el peor de los casos de  $O(\log n)$ .

### 3.2.1. Implementación de la cola de prioridad mínima

En la implementación de colas de prioridad mínima no se considera que el tamaño del montón (`heapSize`) sea definido como una variable global. Es una mala práctica de programación el uso de variables globales; por este motivo, la variable del tamaño del montón será enviada por referencia (el cual es un manejo parecido al uso de punteros) a las funciones que modifican dicho valor, para que los cambios sean reflejados en la función principal del programa.

```

||#include <stdio.h>
||#include <stdlib.h>
||#include <math.h>
||#define myNegativeInfinite -2147483647
||#define myPositiveInfinite 2147483647
||#define MAXT 1000

int Parent(int i)
{
    return i >> 1; /* return i / 2; */
}

int Left(int i)
{
    return i << 1; /* return 2 * i; */
}

int Right(int i)
{
    return (i << 1) + 1; /* return 2 * i + 1; */
}

void MinHeapify(int Q[], int i, int heapSize)
{
    int l, r, least, temp;
    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l] < Q[i]))
        least = l;
    else
        least = i;
}

```

```
    if((r <= heapSize) && (Q[r] < Q[least]))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

int MinPQ_Minimum(int Q[])
{
    return Q[1];
}

int MinPQ_Extract(int Q[], int *heapSize)
{
    int min = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }

    return min;
}

void MinPQ_DecreaseKey(int Q[], int i, int key)
{
    int temp;

    if(key > Q[i])
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = key;
        while((i > 1) && (Q[Parent(i)] > Q[i]))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
    }
}

void MinPQ_Insert(int Q[], int key, int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize] = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, key);
}
```

```

int main()
{
    int operation, element, Q[MAXT + 2], heapSize = 0;

    while (scanf("%d", &operation) != EOF)
    {
        /* Insert into Priority Queue */
        if (operation == 1)
        {
            scanf("%d", &element);
            MinPQ_Insert(Q, element, &heapSize);
        }
        else
        {
            /* Extract min element of Priority Queue */
            if (operation == 2)
            {
                element = MinPQ_Extract(Q, &heapSize);
                if (element == myNegativeInfinite)
                    printf("The Min Priority Queue is empty.\n");
                else
                    printf("%d\n", element);
            }
            else
                printf("Bad use.\n 1. Insert into PQ \n 2. Extract of the
                      PQ.\n");
        }
    }

    return 0;
}

```

```

E:\HUGO\EstructuraDeDatos\ 1 5
1 4
1 3
1 2
1 1
2
1
2
2
1 3
2
3
2
3
1 6
2
4
2
5
1 9
2
6
2
9

```

Figura 3.4. Salida del programa del manejo de la cola de prioridad mínima.

### 3.3. Retos de programación resueltos

En esta sección, se trabajarán tres retos de programación que se apoyan en el tema de colas de prioridad:

- Sumar todos (versión 2025)
- Cardinalidad de conjuntos
- Sala de urgencias

#### 3.3.1. Sumar todos (versión 2025)

**Nombre original:** Add All (version 2025)<sup>2</sup>.

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor:** Hugo Humberto Morales Peña.

El profesor Humberto Moralov, al organizar los papeles de su oficina, se encontró con la siguiente solución, que planteó aproximadamente 10 años atrás para el reto de programación: **UVa - 10954 - Add All**<sup>3</sup>.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXT 5000

int ExtractMin(int A[], int *n)
{
    int temp, i, min;

    for(i = *n - 1; i >= 1; i--)
    {
        if(A[*n] > A[i])
        {
            temp = A[i];
            A[i] = A[*n];
            A[*n] = temp;
        }
    }
    min = A[*n];
    *n = *n - 1;

    return min;
}

void Insert(int A[], int *n, int element)
{
    *n = *n + 1;
    A[*n] = element;
}
```

---

<sup>2</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/add-all-2025>

<sup>3</sup>[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=21&page=show\\_problem&problem=1895](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=21&page=show_problem&problem=1895)

```

int AddAll(int A[], int n)
{
    int result = 0, value1, value2, value3;

    while(n >= 2)
    {
        value1 = ExtractMin(A, &n);
        value2 = ExtractMin(A, &n);
        value3 = value1 + value2;
        result = result + value3;
        Insert(A, &n, value3);
    }

    return result;
}

int main()
{
    int n, A[MAXT + 1], index;

    while(scanf("%d", &n) && (n > 0))
    {
        for(index = 1; index <= n; index++)
            scanf("%d", &A[index]);
        printf("%d\n", AddAll(A, n));
    }

    return 0;
}

```

En la solución anterior, se trabaja con un arreglo  $A[ ]$  en el que se almacenan, inicialmente  $n$  elementos. En la función `AddAll` en el ciclo de repetición `while`, se extraen (y eliminan) del arreglo los dos valores más pequeños, posteriormente se suman y el resultado se inserta (se almacena) en el arreglo; ese valor también se utiliza para actualizar la suma total en la variable `result`, donde se almacena el resultado de sumar todos los pares de valores más pequeños que hay en el arreglo, mientras este contenga 2 o más valores. El costo computacional de la función `AddAll` es  $O(n^2)$ , donde  $n$  es la cantidad de elementos almacenados en el arreglo.

El ciclo `while` se ejecuta  $n - 1$  veces, porque por cada iteración el arreglo tienen un número menos almacenado. El costo de cada iteración es un  $O(n)$ , puesto que se tienen que recorrer los  $n$  elementos almacenados en el arreglo  $A[ ]$  para determinar el valor mínimo, es decir,  $(n - 1) \cdot O(n) = O(n^2)$ .

Un  $O(n^2)$  es una solución muy costosa en tiempo de ejecución; el único motivo en la actualidad por el cual el juez en línea UVa da un veredicto de `accepted`, es porque es un reto de programación muy viejo (se estrenó en el año 2005) y en ese entonces una talla de  $n = 5,000$  era suficiente para un tiempo de 1 o 2 segundos, con el poder de computo de los servidores de esa época.

Actualmente (año 2026), una solución del orden de  $10^8$  operaciones corre en 1 o 2 segundos en los jueces en línea, por este motivo, se le pide a usted, como estudiante de un programa académico de ciencias de la computación, plantear una solución que tenga un tiempo de ejecución de  $O(n \cdot \log n)$  por caso de prueba, donde se tiene que programar eficientemente la función `AddAll` para la nueva talla de la variable  $n$  que ahora es  $10^6$ .

**Formato de entrada:**

La entrada comienza con un entero positivo  $t$  ( $1 \leq t \leq 10$ ), denotando el número de casos de prueba. Cada caso de prueba está conformado por dos líneas. La primera línea, consta de un número entero positivo  $n$  ( $2 \leq n \leq 10^6$ ), que representa la cantidad de elementos para almacenar en el arreglo  $A[ ]$ . La segunda línea, consta de  $n$  números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a  $10^6$ ).

Se garantiza que la suma de todos los valores de  $n$  en los casos de prueba es a lo sumo  $10^6$ .

**Formato de salida:**

Para cada caso de prueba imprimir una sola línea con el resultado generado por la función `AddAll`.

**Ejemplo de entrada:**

```
5
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
5
3 2 5 1 4
5
1 1 1 1 1
```

**Ejemplo de salida:**

```
9
19
33
33
12
```

**Solución del reto**

Se utilizará la estructura de datos de cola de prioridad mínima para implementar de forma eficiente la solución planteada originalmente en el reto de programación.

En el siguiente programa de Lenguaje C se implementa la solución:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myNegativeInfinite -9223372036854775807
#define myPositiveInfinite 9223372036854775807
#define MAXT 1000000
```

```

int Parent(int i)
{
    return i >> 1;
}

int Left(int i)
{
    return i << 1;
}

int Right(int i)
{
    return (i << 1) + 1;
}

void MinHeapify(long long int Q[], int i, int heapSize)
{
    int l, r, least;
    long long int temp;
    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l] < Q[i]))
        least = l;
    else
        least = i;

    if((r <= heapSize) && (Q[r] < Q[least]))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

long long int MinPQ_Minimum(long long int Q[])
{
    return Q[1];
}

long long int MinPQ_Extract(long long int Q[], int *heapSize)
{
    long long int min = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }
    return min;
}

```

```
void MinPQ_DecreaseKey(long long int Q[], int i, long long int key)
{
    long long int temp;

    if(key > Q[i])
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = key;
        while((i > 1) && (Q[Parent(i)] > Q[i]))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
    }
}

void MinPQ_Insert(long long int Q[], long long int key, int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize] = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, key);
}

long long int AddAll(long long int Q[], int *heapSize)
{
    long long int result = 0, value1, value2, value3;

    while(*heapSize >= 2)
    {
        value1 = MinPQ_Extract(Q, &(*heapSize));
        value2 = MinPQ_Extract(Q, &(*heapSize));
        value3 = value1 + value2;
        result = result + value3;
        MinPQ_Insert(Q, value3, &(*heapSize));
    }

    return result;
}

int main()
{
    long long int element, Q[MAXT + 1];
    int n, index, heapSize = 0;

    scanf("%d", &n);

    for(index = 1; index <= n; index++)
    {
        scanf("%lld", &element);
        MinPQ_Insert(Q, element, &heapSize);
    }

    printf("%lld\n", AddAll(Q, &heapSize));
    return 0;
}
```

```

5
3
1 2 3
9
4
1 2 3 4
19
5
5 4 3 2 1
33
5
3 2 5 1 4
33
5
1 1 1 1 1
12

```

Figura 3.5. Salida del programa para el reto: Sumar todos (versión 2025).

En la solución anterior, insertar los  $n$  elementos en la cola de prioridad mínima tiene un costo computacional de  $O(n \cdot \log n)$ ; esta complejidad computacional es exactamente la misma de la función `AddAll`, donde su costo computacional se determina por el fragmento de código:

```

while (*heapSize >= 2)
{
    value1 = MinPQ_Extract(Q, &(*heapSize));
    value2 = MinPQ_Extract(Q, &(*heapSize));
    value3 = value1 + value2;
    result = result + value3;
    MinPQ_Insert(Q, value3, &(*heapSize));
}

```

El ciclo `while` se va a repetir  $n - 1$  veces, por cada iteración la cola de prioridad disminuye su tamaño en 1, porque se extraen dos elementos y se inserta uno. El costo computacional del llamado a las funciones `MinPQ_Extract` y `MinPQ_Insert` es  $O(\log n)$ , por lo tanto, el fragmento del código tiene un costo computacional en el peor de los casos de  $(n - 1) \cdot (\log n + \log n + \log n) = (n - 1) \cdot (3 \cdot \log n) \leq n \cdot (3 \cdot \log n) = 3 \cdot (n \cdot \log n) = O(n \cdot \log n)$ .

En el reto de programación se especifica que hay  $t$  casos de prueba, por lo tanto, la complejidad en el peor de los casos en tiempo de ejecución de la solución es  $t \cdot O(n \cdot \log n) = O(t \cdot n \cdot \log n)$ .

Tomando el máximo valor para la variable  $n$  de  $10^6$ , el valor de la variable  $t$  tiene que tomar un valor de 1, por lo tanto, la cantidad de operaciones en el peor de los casos es del orden de  $t \cdot n \cdot \log(n) = 1 \cdot 10^6 \cdot \log(10^6) = 10^6 \cdot 6 = 6 \cdot 10^6 \leq 10^7$ .

### 3.3.2. Cardinalidad de conjuntos

**Nombre original:** Cardinality of Sets<sup>4</sup>.

**Fuente:** 4<sup>a</sup> Fecha de la ICPC Centroamérica 2020.

**Fecha:** 30 de Enero de 2021.

**Autor:** Hugo Humberto Morales Peña.

<sup>4</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/cardinality-of-sets>

Se tienen dos conjuntos de números enteros positivos  $A$  y  $B$  con  $|A| = m$  y  $|B| = n$  (es importante recordar que la cardinalidad de un conjunto es la cantidad de elementos diferentes que contiene, la cardinalidad se indica con las barras que encierran al conjunto).

Se quiere determinar las cardinalidades de los conjuntos que se obtienen como resultado de  $A - B$ ,  $A \cap B$ ,  $B - A$  y  $A \cup B$ .

**Formato de entrada:**

La entrada contiene un único caso de prueba que consiste de tres líneas.

La primera línea, contiene dos números enteros positivos  $m$   $n$  ( $1 \leq m, n \leq 10^6$ ), los cuales representan las cardinalidades de los conjuntos  $A$  y  $B$  respectivamente.

La segunda línea, contiene  $m$  números enteros positivos diferentes (separados por un espacio en blanco) los cuales conforman el conjunto  $A$ , dichos valores están en el intervalo cerrado  $[1, 10^9]$ .

La tercera línea, contiene  $n$  números enteros positivos diferentes (separados por un espacio en blanco) los cuales conforman el conjunto  $B$ , dichos valores están en el intervalo cerrado  $[1, 10^9]$ .

**Formato de salida:**

Para el caso de prueba, la salida debe contener una sola línea con cuatro números enteros no negativos, separados por un solo espacio los cuales representan respectivamente  $|A - B|$   $|A \cap B|$   $|B - A|$   $|A \cup B|$ .

**Ejemplo de entrada 1:**

```
5 6
7 1 4 3 9
8 3 2 10 7 1
```

**Ejemplo de salida 1:**

```
2 3 3 8
```

**Ejemplo de entrada 2:**

```
5 5
1 3 5 7 9
10 8 6 4 2
```

**Ejemplo de salida 2:**

```
5 0 5 10
```

**Ejemplo de entrada 3:**

```
8 6
10 1 5 3 6 4 7 2
1 2 3 4 5 6
```

**Ejemplo de salida 3:**

```
2 6 0 8
```

**Solución del reto**

La solución natural para este reto de programación es almacenar y ordenar los elementos del conjunto  $A$  en un arreglo  $A$ , luego se leen uno a uno los elementos del conjunto  $B$  y se mandan a buscar en el arreglo  $A$ , de esta forma se determina la cardinalidad de la intersección de los conjuntos  $A$  y  $B$ .

La solución por colas de prioridad no es convencional, es una forma alternativa que tiene exactamente la misma complejidad teórica en tiempo de ejecución que la solución por ordenamiento y búsqueda binaria.

La estrategia es la siguiente, los elementos del conjunto  $A$  se almacenan en una cola de prioridad mínima  $PQ_A$  y los elementos del conjunto  $B$  se almacenan en una cola de prioridad mínima  $PQ_B$ . Se repite el siguiente procedimiento hasta que alguna de las dos colas de prioridad queda vacía; posteriormente se consultan los elementos más pequeños en las dos colas de prioridad, si estos elementos tienen el mismo valor, entonces pertenecen a la intersección de los conjuntos y por lo tanto son extraídos de las colas de prioridad, de lo contrario, solamente se extrae el elemento más pequeño de la cola de prioridad a la cual pertenece, y vuelve y se repite el procedimiento.

En el siguiente programa de Lenguaje C se implementa la estrategia anterior:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define myNegativeInfinite -2147483647
#define myPositiveInfinite 2147483647
#define MAXT 1000000

int Parent(int i)
{
    return i >> 1;
}

int Left(int i)
{
    return i << 1;
}

int Right(int i)
{
    return (i << 1) + 1;
}
```

```
void MinHeapify(int Q[], int i, int heapSize)
{
    int l, r, least, temp;
    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l] < Q[i]))
        least = l;
    else
        least = i;

    if((r <= heapSize) && (Q[r] < Q[least]))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

int MinPQ_Minimum(int Q[])
{
    return Q[1];
}

int MinPQ_Extract(int Q[], int *heapSize)
{
    int min = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }

    return min;
}

void MinPQ_DecreaseKey(int Q[], int i, int key)
{
    int temp;

    if(key > Q[i])
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = key;
        while((i > 1) && (Q[Parent(i)] > Q[i]))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
    }
}
```

```

        Q[Parent(i)] = temp;
        i = Parent(i);
    }
}

void MinPQ_Insert(int Q[], int key, int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize] = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, key);
}

int main()
{
    int m, n, element, idElement, heapSizeQA = 0, heapSizeQB = 0;
    int intersection, QA[MAXT + 1], QB[MAXT + 1];

    scanf("%d %d", &m, &n);

    for(idElement = 1; idElement <= m; idElement++)
    {
        scanf("%d", &element);
        MinPQ_Insert(QA, element, &heapSizeQA);
    }

    for(idElement = 1; idElement <= n; idElement++)
    {
        scanf("%d", &element);
        MinPQ_Insert(QB, element, &heapSizeQB);
    }

    intersection = 0;

    while((heapSizeQA >= 1) && (heapSizeQB >= 1))
    {
        if(MinPQ_Minimum(QA) < MinPQ_Minimum(QB))
            element = MinPQ_Extract(QA, &heapSizeQA);
        else
        {
            if(MinPQ_Minimum(QA) > MinPQ_Minimum(QB))
                element = MinPQ_Extract(QB, &heapSizeQB);
            else if(MinPQ_Minimum(QA) == MinPQ_Minimum(QB))
            {
                intersection++;
                element = MinPQ_Extract(QA, &heapSizeQA);
                element = MinPQ_Extract(QB, &heapSizeQB);
            }
        }
    }

    printf("%d %d %d\n", m - intersection, intersection,
           n - intersection, m + n - intersection);

    return 0;
}

```

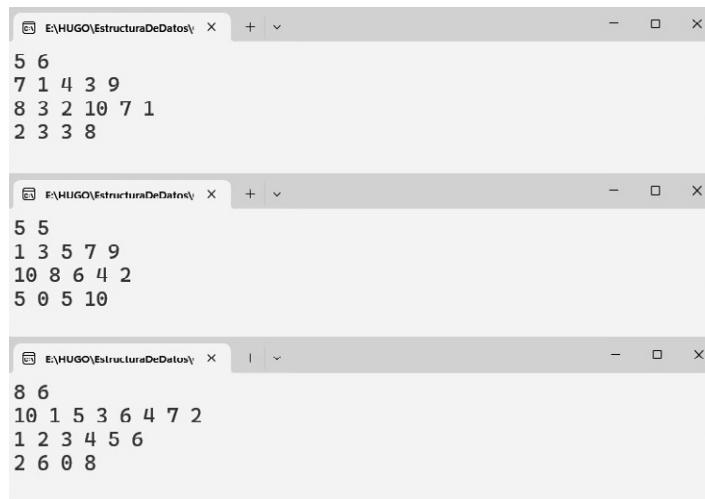


Figura 3.6. Salida del programa para el reto: Cardinalidad de conjuntos.

El siguiente fragmento de código es el que determina la complejidad en el peor de los casos de la solución:

```

while((heapSizeQA >= 1) && (heapSizeQB >= 1))
{
    if(MinPQ_Minimum(QA) < MinPQ_Minimum(QB))
        element = MinPQ_Extract(QA, &heapSizeQA);
    else
    {
        if(MinPQ_Minimum(QA) > MinPQ_Minimum(QB))
            element = MinPQ_Extract(QB, &heapSizeQB);
        else if(MinPQ_Minimum(QA) == MinPQ_Minimum(QB))
        {
            intersection++;
            element = MinPQ_Extract(QA, &heapSizeQA);
            element = MinPQ_Extract(QB, &heapSizeQB);
        }
    }
}

```

La peor entrada que se puede tener para el reto es un par de conjuntos disyuntos, como los siguientes:

- $A = \{1, 3, 5, 7, \dots, 1999999\}$
- $B = \{2, 4, 6, 8, \dots, 2000000\}$

La intersección de un par de conjuntos disyuntos es 0. Para llegar a este resultado, el ciclo `while` se tiene que ejecutar  $m + n$  veces alternando el llamado a la función de extraer el elemento más pequeño de cada una de las colas de prioridad, lo cual tiene un costo computacional en el peor de los casos de  $O((m + n) \cdot \log(\max(m, n)))$ .

Tomando los máximos valores para las variables  $m$  y  $n$ , la cantidad de operaciones en el peor de los casos es del orden de  $(m + n) \cdot \log(\max(m, n)) = (10^6 + 10^6) \cdot \log(\max(10^6, 10^6)) = (2 \cdot 10^6) \cdot \log(10^6) = (2 \cdot 10^6) \cdot 6 = 2 \cdot 6 \cdot 10^6 = 12 \cdot 10^6 = 1,2 \cdot 10^7 = 1,2 \cdot 10^7 \leq 10^8$ .

### 3.3.3. Sala de urgencias

**Nombre original:** Just an Emergency<sup>5</sup>.

**Fuente:** Maratón de Programación UFPS 2020.

**Fecha:** 12 de Diciembre de 2020.

**Autor:** Hugo Humberto Morales Peña.

En días pasados el profesor Humbertov Moralov tuvo un pequeño accidente en su pie izquierdo que lo hizo ir al hospital; durante las casi cuatro horas de espera el profesor tuvo tiempo para analizar y entender como funciona el servicio de urgencias colombiano. Constantemente están llegando al servicio de urgencias pacientes que solicitan atención médica, quienes inicialmente son valorados por un médico en el procedimiento de triage en el que se determina la prioridad de la urgencia, las posibles clasificaciones del triage son 1, 2, 3, 4 y 5; siendo 1 la calificación con la cual se indica que la atención tiene que ser inmediata (la vida del paciente está en alto riesgo) y 5 la calificación con la cual se indica que el paciente necesita atención médica pero que no tiene comprometido ningún órgano vital y que, por lo tanto, puede esperar por cinco o seis horas en la sala de urgencias. Después de la valoración del triage, los pacientes pasan a la sala de espera en la cual son llamados por orden de prioridad con respecto a la valoración del triage.

El profesor Moralov requiere de su ayuda y le pide una solución computacional para poder determinar el orden en el cual son atendidos los pacientes en el servicio de urgencias.

**Nota:** cuando hay varios pacientes con la misma prioridad en el triage se tiene que atender al que llegó primero (con respecto a la hora de llegada); si esto no se respeta, se generará una “asombra” por parte de los pacientes y el servicio de urgencias será destruido. De usted depende que el servicio de urgencias siga prestando sus servicios a la comunidad.

#### Formato de entrada:

La entrada del problema consiste de varios casos de prueba. Cada caso de prueba comienza con una línea que contiene un número entero positivo  $n$  ( $3 \leq n \leq 2 \cdot 10^5$ ) que representa el total de “transacciones” realizadas en el servicio de urgencias. Luego se presentan  $n$  líneas, cada una de ellas comenzando con un par de números enteros positivos  $t$   $h$  ( $t \in \{1, 2\}$ ,  $1 \leq h \leq 8 \cdot 10^5$ ) que representan respectivamente el tipo de transacción (1 para indicar que es una solicitud del servicio de urgencias de un paciente y 2 para indicar la atención de la urgencia por parte de un doctor) y la hora (en segundos). Para las transacciones del tipo 1, la línea se complementa con  $p$   $s$  ( $p \in \{1, 2, 3, 4, 5\}$ ,  $s$  es una cadena sin espacios que solo incluye letras mayúsculas del alfabeto inglés con una longitud máxima de 20) que representan respectivamente la prioridad del triage y el nombre del paciente. El final de los casos de prueba es dado por el fin de archivo (End Of File - EOF).

#### Formato de salida:

Para cada caso de prueba, su programa debe imprimir tantas líneas como transacciones del tipo 2, cada una de estas líneas debe contener tres números enteros positivos y una cadena  $a$   $b$   $c$   $s$  ( $1 \leq a, b, c \leq 8 \cdot 10^5$ ,  $a < b$ ) que representan la información de la atención al paciente

---

<sup>5</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/sala-de-urgencias>

con respecto a la hora de llegada, la hora en que es atendido, el tiempo total de espera y el nombre.

**Ejemplo de entrada:**

```
10
1 1 5 CARLOS
1 3 4 MANUEL
1 4 4 ANDRES
1 6 1 GABRIEL
2 9
1 13 2 YEFRI
2 20
1 24 1 STEVEN
2 30
2 50
```

**Ejemplo de salida:**

```
6 9 3 GABRIEL
13 20 7 YEFRI
24 30 6 STEVEN
3 50 47 MANUEL
```

**Solución del reto**

Para la solución de este reto es necesario extender el manejo de las colas de prioridad mínima sobre un arreglo de estructuras, donde una de las componentes es el campo `key`, campo de la llave sobre la cual se realiza el manejo de la prioridad entre los elementos almacenados en las diferentes posiciones del arreglo; el otro campo de la estructura es `name` donde se almacena el nombre del paciente. La definición de la estructura de las transacciones es la siguiente:

```
|| struct transaction
{|
    int key;
    char name[22];
};|
```

Para las transacciones del tipo 1, la prioridad se obtiene al multiplicar el triage por un millón y sumarle la hora de llegada:

$$key = triage * 1000000 + hora;$$

De esta forma nunca se tendrán dos llaves con el mismo valor, porque en las 6 cifras menos significativas se almacena la hora (cuyo máximo valor es 800,000) y en la cifra de unidades de millón se almacena el valor del triage. Los valores mínimo y máximo que se puede almacenar (con respecto a las tallas del reto) son 1,000,001 y 6,800,000, respectivamente.

Para las transacciones del tipo 2, la hora de llegada del paciente se recupera a partir de la llave, donde se deben recuperar las 6 cifras menos significativas, esto se logra al aplicar la

operación módulo de un millón sobre el valor de la llave.

En el siguiente programa de Lenguaje C se implementa la estrategia anterior:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#define myNegativeInfinite -2147483647
#define myPositiveInfinite 2147483647
#define MAXV 200000

struct transaction
{
    int key;
    char name[22];
};

int Parent(int i)
{
    return i >> 1;
}

int Left(int i)
{
    return i << 1;
}

int Right(int i)
{
    return (i << 1) + 1;
}

void MinHeapify(struct transaction Q[], int i, int heapSize)
{
    int l, r, least;
    struct transaction temp;

    l = Left(i);
    r = Right(i);

    if((l <= heapSize) && (Q[l].key < Q[i].key))
        least = l;
    else
        least = i;

    if((r <= heapSize) && (Q[r].key < Q[least].key))
        least = r;

    if(least != i)
    {
        temp = Q[i];
        Q[i] = Q[least];
        Q[least] = temp;
        MinHeapify(Q, least, heapSize);
    }
}

```

```
struct transaction MinPQ_Minimum(struct transaction Q[])
{
    return Q[1];
}

struct transaction MinPQ_Extract(struct transaction Q[], int *heapSize)
{
    struct transaction min;
    min.key = myNegativeInfinite;

    if(*heapSize < 1)
        printf("Heap underflow.\n");
    else
    {
        min = Q[1];
        Q[1] = Q[*heapSize];
        (*heapSize)--;
        MinHeapify(Q, 1, *heapSize);
    }

    return min;
}

void MinPQ_DecreaseKey(struct transaction Q[], int i,
                      struct transaction service)
{
    struct transaction temp;
    if(service.key > Q[i].key)
        printf("New key is higher than current key.\n");
    else
    {
        Q[i] = service;
        while((i > 1) && (Q[Parent(i)].key > Q[i].key))
        {
            temp = Q[i];
            Q[i] = Q[Parent(i)];
            Q[Parent(i)] = temp;
            i = Parent(i);
        }
    }
}

void MinPQ_Insert(struct transaction Q[], struct transaction service,
                  int *heapSize)
{
    (*heapSize)++;
    Q[*heapSize].key = myPositiveInfinite;
    MinPQ_DecreaseKey(Q, *heapSize, service);
}

int main()
{
    int n, t, h, p, idTransaction;
    int heapSize, arrivalTime, waitTime;
    char name[22];
    struct transaction Q[MAXV + 1], service;

    while(scanf("%d", &n) != EOF)
    {
```

```

heapSize = 0;
for(idTransaction = 1; idTransaction <= n; idTransaction++)
{
    scanf("%d %d", &t, &h);

    /* request */
    if(t == 1)
    {
        scanf("%d %s", &p, name);
        service.key = p * 1000000 + h;
        strcpy(service.name, name);
        MinPQ_Insert(Q, service, &heapSize);
    }
    else
    {
        /* attention */
        if(t == 2)
        {
            if(heapSize >= 1)
            {
                service = MinPQ_Extract(Q, &heapSize);
                arrivalTime = service.key % 1000000;
                waitTime = h - arrivalTime;
                printf("%d %d %d %s\n", arrivalTime, h,
                       waitTime, service.name);
            }
        }
    }
}

return 0;
}

```

Salida del programa anterior ejecutando los ejemplos del reto:

```

10
1 1 5 CARLOS
1 3 4 MANUEL
1 4 4 ANDRES
1 6 1 GABRIEL
2 9
6 9 3 GABRIEL
1 13 2 YEFRI
2 20
13 20 7 YEFRI
1 24 1 STEVEN
2 30
24 30 6 STEVEN
2 50
3 50 47 MANUEL

```

Figura 3.7. Salida del programa para el reto: Sala de urgencias.

El siguiente fragmento de código es el que determina la complejidad en el peor de los casos de la solución:

```

while (scanf("%d", &n) != EOF)
{
    heapSize = 0;
    for (idTransaction = 1; idTransaction <= n; idTransaction++)
    {
        scanf("%d %d", &t, &h);

        /* request */
        if (t == 1)
        {
            scanf("%d %s", &p, name);
            service.key = p * 1000000 + h;
            strcpy(service.name, name);
            MinPQ_Insert(Q, service, &heapSize);
        }
        else
        {
            /* attention */
            if (t == 2)
            {
                if (heapSize >= 1)
                {
                    service = MinPQ_Extract(Q, &heapSize);
                    arrivalTime = service.key % 1000000;
                    waitTime = h - arrivalTime;
                    printf("%d %d %d %s\n", arrivalTime, h,
                           waitTime, service.name);
                }
            }
        }
    }
}

```

Para cada uno de los casos de prueba se insertan o se extraen  $n$  elementos de la cola de prioridad, esto tiene una complejidad de  $O(n \log n)$ .

En el reto de programación se indica que existen múltiples casos de prueba, pero no se indican cuantos son, hay que leer y leer casos de prueba hasta que llegue el fin de archivo (EOF). Por este motivo, es importante considerar que la variable  $t$  representa el total de casos de prueba, por lo tanto, la complejidad en tiempo de ejecución de la solución es  $O(t \cdot n \log n)$ .

Tomando el máximo valor para la variable  $n$ , la cantidad de operaciones en el peor de los casos es del orden de  $t \cdot n \cdot \log(n) = t \cdot 2 \cdot 10^5 \cdot \log(2 \cdot 10^5) \leq t \cdot 2 \cdot 10^5 \cdot \log(10^6) = t \cdot 2 \cdot 10^5 \cdot 6 = t \cdot 2 \cdot 6 \cdot 10^5 = t \cdot 12 \cdot 10^5 = t \cdot 1,2 \cdot 10 \cdot 10^5 = t \cdot 1,2 \cdot 10^6 = 1,2 \cdot t \cdot 10^6$ .

Sin ningún problema, el total de casos de prueba podría llegar a tener un valor de 50 y aun así tener una solución aceptable en tiempo de ejecución, con un orden de  $1,2 \cdot 50 \cdot 10^6 = 60 \cdot 10^6 = 6 \cdot 10 \cdot 10^6 = 6 \cdot 10^7 \leq 10^8$  operaciones, la cual correría en los jueces en línea en 1 o 2 segundos.

### 3.4. Retos de programación propuestos

En esta sección, se propone una lista de retos de programación que se pueden resolver con el uso de las colas de prioridad:

- Guanex y las colas de prioridad
- Impossible
- Asignación de citas para la vacuna del COVID-19
- Duerme, Tobby
- Hecho en Colombia
- Juego de pares e impares
- Máximo de eventos en una sala
- Máximo de eventos en múltiples salas

### 3.4.1. Guanex y las colas de prioridad

**Nombre original:** Guanex y las Colas de Prioridad<sup>6</sup> <sup>7</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Hugo Humberto Morales Peña.

Santiago Guaneme (Guanex) es un joven de 14 años de Dosquebradas (Risaralda), pionero en el Eje Cafetero en las competencias de programación del IOI (International Olympiad in Informatic, en español, Olimpiada Internacional en Informática). Este joven, que aún está en el colegio, fue quien obtuvo el mejor desempeño de la selección Colombia en el mundial de programación del IOI en Egipto (IOI 2024 - Egypt).

Guanex se encuentra entrenando fuertemente para el IOI 2025, que se realizará en Sucre (Bolivia) del 27 de Julio al 3 de Agosto. Por este motivo, está estudiando nuevas temáticas, una de ellas es las de colas de prioridad; te solicita que le ayudes a resolver el siguiente reto de programación:

Inicialmente, se tiene una cola de prioridad y se dan algunas consultas sobre ella. Las consultas son operaciones básicas que se pueden realizar sobre la estructura de datos cola de prioridad mínima, tales como **MinPQ\_Insert** (inserta un elemento en la cola de prioridad mínima), **MinPQ\_Extract** (extrae la primera ocurrencia del elemento mínimo en la cola de prioridad, siempre y cuando la cola de prioridad tenga elementos). También, es necesario conocer en cualquier momento el valor del elemento mínimo **MinPQ\_Minimum** de los números enteros que se encuentran en la cola de prioridad. La tarea a realizar es procesar las consultas dadas.

#### Formato de entrada:

La entrada contiene un único caso de prueba. La primera línea, contiene un número entero positivo  $Q$  ( $1 \leq Q \leq 2 \cdot 10^6$ ), denotando el número de consultas a procesar, seguidas por exactamente  $Q$  líneas, basadas en el siguiente formato:

- 1  $V$ : Operación **MinPQ\_Insert**  $V$  ( $0 \leq V \leq 10^6$ ). Inserta el número entero  $V$  en la cola de prioridad.
- 2: Operación **MinPQ\_Extract**. Si la cola de prioridad tiene elementos, entonces extrae la primera ocurrencia del elemento mínimo; si la cola de prioridad está vacía, entonces no se hace nada.
- 3: Imprimir, en una línea, el valor del elemento mínimo de la cola de prioridad. Si la cola de prioridad está vacía, imprimir en una línea el mensaje: **Empty!**

#### Formato de salida:

Para cada consulta del tipo 3, imprimir en una sola línea, el valor del elemento mínimo de la cola de prioridad. Si la cola de prioridad está vacía, imprimir en una línea el mensaje: **Empty!**

---

<sup>6</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/queries-priority-queue>

<sup>7</sup><https://codeforces.com/profile/Guanexxx>

**Ejemplo de entrada:**

```
12  
3  
2  
1 10  
1 5  
3  
2  
3  
1 5  
1 3  
3  
1 15  
3
```

**Ejemplo de salida:**

```
Empty!  
5  
10  
3  
3
```

### 3.4.2. Impossible

**Nombre original:** Impossible<sup>8</sup>.

**Fuente:** Maratón de Programación UFPS 2022.

**Fecha:** 3 de Septiembre de 2022.

**Autor:** Milton Jesús Vera Contreras

Dos matemáticos y un ingeniero auditán las transacciones en cajeros automáticos de un banco. Cada transacción está identificada con un número ID único y consecutivo. Ellos tienen la lista de transacciones desordenadas, pero falta el ID de una de las identificaciones. Los matemáticos dicen que no es posible encontrar el número de identificación que falta. El ingeniero dice que hay casos que se pueden solucionar. ¿Puedes ayudarlos?

#### Formato de entrada:

La entrada contiene múltiples casos de prueba. Cada caso de prueba comienza con un número  $n$  ( $10^3 \leq n \leq 10^6$ ), que corresponde al número de transacciones. Luego sigue una lista de  $n - 1$  números de identificación  $x_i$  ( $10^5 \leq x_i \leq 10^9$ ). Los números de identificación son consecutivos y desordenados, no hay números duplicados y falta un número. Los números están separados por un espacio en blanco.

#### Formato de salida:

Para cada caso de prueba, se debe imprimir en una sola línea el número de identificación de la transacción  $x_i$  que falta o **IMPOSSIBLE** si los matemáticos tienen razón y “no es posible encontrar el número de identificación que falta”.

#### Ejemplo de entrada 1:

```
10
123465 123460 123459 123457 123458 123463 123461 123464 123456
```

#### Ejemplo de salida 1:

```
123462
```

#### Ejemplo de entrada 2:

```
11
789065 789060 789059 789057 789058 789062 789063 789061 789064 789056
```

#### Ejemplo de salida 2:

```
IMPOSSIBLE
```

**Nota:** dada la cantidad de datos, los ejemplos presentados en el enunciado utilizan un valor de  $n$  pequeño.

---

<sup>8</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/impossible-1>

### 3.4.3. Asignación de citas para la vacuna del COVID-19

**Nombre original:** Health HPF<sup>9</sup> <sup>10</sup>.

**Fuente:** Primera Fecha Gran Premio de Centro América 2021.

**Fecha:** 28 de Agosto de 2021.

**Autor:** Eddy Ramírez Jiménez.

Debido a una pandemia actual, los centros de vacunación tienen que atender a una población enorme, por lo que han catalogado a las personas para poder atender primero a las que tienen mayor riesgo por su edad o porque tienen enfermedades de base. Sin embargo, para algunas personas se ha fallado en identificar su verdadera prioridad, se han equivocado en cuándo deberían de vacunarse, o simplemente, han atendido tarde a la cita de vacunación.

Sin embargo, la directriz del Ministerio de Salud es clara, cualquier persona que se haya inscrito para la vacunación, debe ser llamada acorde con su prioridad. Además, como un dato no menor, nuestro país cuenta con un contrato de exclusividad con la Compañía Johnson & Johnson que fabrica la vacuna Janssen, que es de una sola dosis.

Si dos personas tuvieran la misma prioridad, entonces la primera en haber sido ingresada al sistema iría primero.

Para ello usted debe hacer un programa que responda de la manera más eficiente posible sobre quién es el siguiente en ser llamado a vacunar.

#### Formato de entrada:

La entrada del problema consiste en un solo caso de prueba.

El caso de prueba contiene como máximo  $10^6$  líneas. Hay dos tipos de líneas, la de “ingreso”, de la información de una persona al sistema y la de “llamado”, para citar a una persona a que se acerque a un centro de vacunación para recibir la vacuna.

Una línea de ingreso contiene una cadena sin espacios (de máximo 20 letras en minúscula [a-z] del alfabeto en inglés), que representa el nombre de una persona y un número entero positivo  $p$  ( $1 \leq p \leq 1000$ ) que señala la prioridad (entre más alto el valor, más alta es la prioridad).

Una línea de llamado contiene únicamente una cadena de longitud uno con la letra mayúscula V (de vacuna).

El final del caso de prueba es dado por el fin de archivo (End Of File - EOF).

#### Formato de salida:

Para cada caso de prueba, su programa debe imprimir tantas líneas como líneas de llamado hay en la entrada, cada una de estas líneas debe contener una cadena con el nombre de la

---

<sup>9</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/asignacion-de-citas-para-la-vacuna-del-covid-19>

<sup>10</sup>Explicación de como solucionar el reto generada por su propio autor en: <https://www.youtube.com/watch?v=f1UffBwg-jM>

persona que debe ser vacunada. Si no hay personas registradas en el sistema en el momento del llamado, entonces se debe imprimir una línea en blanco.

**Ejemplo de entrada 1:**

```
laura 1
oscar 100
emilio 20
V
V
andres 30
V
```

**Ejemplo de salida 1:**

```
oscar
emilio
andres
```

**Ejemplo de entrada 2:**

```
hugo 100
humberto 90
V
V
V
rafael 80
V
```

**Ejemplo de salida 2:**

```
hugo
humberto

rafael
```

### 3.4.4. Duerme, Tobby

**Nombre original:** Rockabye Tobby<sup>11</sup>.

**Fuente:** UTP Open 2017.

**Fecha:** 1ro de Abril de 2017.

**Autor:** Yeferson Gaitán Gómez.

“Duerme bebé, no llores”.

Tobby es muy bueno atrapando la pelota; le encanta tanto ese juego que un día decidió salir a jugar, aunque estaba lloviendo. Jugó durante mucho tiempo y, además de atrapar la pelota muchas veces, también se resfrió. Por eso, ahora su madre perruna lo cuidará cantando una hermosa canción de cuna (“duerme bebé, no llores”) y dándole los medicamentos en los momentos en que deben tomarse.

En la receta médica enviada por el doctor, se especifica el nombre de los medicamentos y la frecuencia con la que deben tomarse. El doctor le dijo que, si toma los medicamentos de manera constante, se recuperará después de tomar  $k$  medicamentos. A Tobby no le gusta estar enfermo (de hecho, a nadie le gusta), por lo que promete a su madre ser constante con los medicamentos. Por eso, ahora quiere saber cuáles son los primeros  $k$  medicamentos que debe tomar para sentirse mejor. ¿Puedes ayudarlo?

#### Formato de entrada:

La primera línea de entrada contiene un entero  $T$  ( $1 \leq T \leq 5$ ), el numero de casos de prueba.

Para cada caso de prueba, la prescripción médica de Tobby está escrita de la siguiente manera:

Cada caso de prueba comienza con una línea que contiene dos enteros  $n$  ( $1 \leq n \leq 10^5$ ) y  $k$  ( $1 \leq k \leq 10^4$ ). El numero de medicamentos enviados por el doctor y el número mínimo de medicamentos que Tobby debe tomar para sentirse mejor. El caso de prueba continua con  $n$  líneas, cada una de la forma *nombre frecuencia* ( $1 \leq |\text{nombre}| \leq 15$ ,  $1 \leq \text{frecuencia} \leq 2 \cdot 10^5$ ), indicando el nombre del medicamento y que tan seguido debe tomarse.

Los medicamentos están listados de acuerdo con su grado de prioridad, es decir, el primero será el fármaco mas importante y el ultimo será el menos importante.

#### Formato de salida:

Para cada caso de prueba, la salida debe contener  $k$  líneas, cada una de la forma  $t\ m$  indicando que, en el momento  $t$  Tobby debe tomar el medicamento  $m$ .

Si hay dos o mas medicamentos que deben ser tomados al mismo tiempo  $t$ , deben ser listados de acuerdo a su prioridad.

---

<sup>11</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/rockabye-toby>

**Ejemplo de entrada:**

```
1
2 5
Acetaminofen 20
Loratadina 30
```

**Ejemplo de salida:**

```
20 Acetaminofen
30 Loratadina
40 Acetaminofen
60 Acetaminofen
60 Loratadina
```

### 3.4.5. Hecho en Colombia

**Nombre original:** Made in Colombia<sup>12</sup>.

**Fuente:** UTP Open 2013.

**Fecha:** 4 de Mayo de 2013.

**Autor:** Sebastián Gómez González.

Los libros estadísticos de todas partes del mundo especifican cómo calcular la media y la mediana de cualquier conjunto de datos. La media o promedio de un conjunto de datos  $X = \{X_0, X_1, \dots, X_{n-1}\}$  se define como:

$$\mu = \sum_{i=0}^{n-1} \frac{X_i}{n}$$

La mediana del mismo conjunto de datos, es el dato en la posición  $\lfloor \frac{n}{2} \rfloor$  del arreglo  $X$  ordenado de forma ascendente. Veamos el ejemplo de la Figura 3.8.

$I$	Datos	Ordenado	Mediana
1	{3}	{3}	3
2	{3, 2}	{2, 3}	2
3	{3, 2, 1}	{1, 2, 3}	2
4	{3, 2, 1, 8}	{1, 2, 3, 8}	2
5	{3, 2, 1, 8, 7}	{1, 2, 3, 7, 8}	3

Figura 3.8. Ejemplo cálculo del valor de la mediana.

En el ejemplo de la Figura 3.8, la mediana con  $I = 5$  es el elemento en la posición  $\lfloor \frac{5}{2} \rfloor = 2$  del arreglo ordenado, es decir, el número 3. En Colombia, para diferenciarnos del resto de países, se incluye en los libros de estadística cómo calcular la media de las medianas. Si  $M(X)$  calcula la mediana de los datos del conjunto  $X$ , la media de las medianas  $\mu_m$  de un conjunto de datos  $X = \{X_0, X_1, \dots, X_{n-1}\}$  se calcula así:

$$\mu_m = \frac{1}{n} \sum_{i=0}^{n-1} M(\{X_0, X_1, \dots, X_i\})$$

Para explicar mejor como se hace el cálculo, supongamos que el conjunto de datos  $X$  es  $\{3, 2, 1, 8, 7\}$ . Como la entrada tiene 5 valores, se deben calcular 5 medianas. La tabla anterior muestra el ejemplo del cálculo de las 5 medianas sobre las que se debe calcular el promedio, que representan las medianas después de insertar cada uno de los 5 valores. Para calcular la media de las medianas, simplemente se toma el promedio de las medianas calculadas:

$$\mu_m = \frac{3 + 2 + 2 + 2 + 3}{5} = 2,4$$

En INSILICO (INvestigaciones Sobre Informática y LIBros de COlombia), lo han contratado a usted para que implemente un programa que pueda calcular la media de las medianas de un conjunto de datos.

<sup>12</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/made-in-colombia>

**Formato de entrada:**

La entrada consiste de varios casos de prueba. La primera línea de cada caso de prueba contiene un entero  $N$  que indica la cantidad de elementos del conjunto de entrada  $X$ , luego vienen  $N$  enteros que indican cada uno de los elementos desde  $X_0$  hasta  $X_{n-1}$ . El último caso de prueba va seguido por un caso con  $N = 0$ , el cual no debe ser procesado.

- $1 \leq N \leq 10^6$
- $0 \leq X_i \leq 10^9 \quad \forall i \in [0, N - 1]$

**Formato de salida:**

Por cada caso de prueba de la entrada, se debe imprimir en una sola línea el valor de la media de medianas redondeada a 2 cifras decimales.

**Ejemplo de entrada:**

```
5
3 2 1 8 7
0
```

**Ejemplo de salida:**

```
2.40
```

### 3.4.6. Juego de pares e impares

**Nombre original:** Juego<sup>13</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor(es):** Hugo Humberto Morales Peña, Gabriel Gutiérrez Tamayo.

Emma y Otto son un par de niños muy talentosos que aprovechan su tiempo libre para jugar y practicar matemáticas al mismo tiempo. En estos momentos ellos se encuentran jugando “pares e impares”, juego creado por ellos en el último mes.

El juego de pares e impares consiste en lo siguiente:

- En secreto, Emma escribe en una hoja de papel una lista con  $n$  números enteros no negativos.
- En secreto, Otto escribe en una hoja de papel una lista con  $n$  números enteros no negativos.
- Después, los jugadores conocen las dos listas de números que se van a enfrentar y así será hasta el final del juego.
- Por cada turno:
  - Emma saca de su lista el número más grande y lo juega.
  - Otto saca de su lista el número más pequeño y lo juega.
  - Se suman los dos números, si el resultado es un número impar, entonces este es adicionado a la lista de Otto, de lo contrario es dividido por 2 y el nuevo valor es adicionado a la lista de Emma.
- El juego termina cuando al final de un turno una de las dos listas queda vacía.
- El juego es perdido por quien queda con la lista vacía.

Emma y Otto necesitan de tu ayuda para simular el juego e indicar el turno en el que termina y el ganador del mismo, esto con el fin de validar los resultados a los cuales están llegando con el juego; ojo, ellos no solamente están jugando, también están practicando las funciones matemáticas de: mínimo, máximo, suma, división y residuo.

#### Formato de entrada:

La entrada comienza con un número entero  $t$  ( $1 \leq t \leq 10$ ), que representa el total de casos de prueba. Cada uno de los casos de prueba consta de tres líneas. La primera línea, contiene un número entero  $n$  ( $1 \leq n \leq 5 \cdot 10^5$ ), que representa el tamaño de las listas. La segunda línea, contiene  $n$  números enteros no negativos en el rango de  $[0, 10^6]$  separados por espacio en blanco, representando la lista de números de Emma. La tercera línea, contiene  $n$  números enteros no negativos en el rango de  $[0, 10^6]$  separados por espacio en blanco, representando la lista de números de Otto.

**Nota:** se garantiza que la suma de los valores de  $n$  en los  $t$  casos de prueba es a lo sumo un  $5 \cdot 10^5$ .

---

<sup>13</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/game-2025>

**Formato de salida:**

Por cada caso de prueba, imprimir en una sola línea un número entero que representa la cantidad de turnos del juego hasta que termina y el nombre del jugador ganador, ya sea **Emma** u **Otto**.

**Nota:** en este juego nunca hay empates, siempre hay un ganador.

**Ejemplo de entrada:**

```
5
5
1 1 1 1 1
2 2 2 2 2
5
5 4 3 2 1
5 1 3 1 5
4
2 4 4 2
6 2 8 4
5
13 17 2 0 4
5 1 11 3 18
5
5 1 11 3 18
13 17 2 0 4
```

**Ejemplo de salida:**

```
5 Otto
9 Emma
5 Emma
8 Otto
8 Emma
```

### 3.4.7. MÁXIMO DE EVENTOS EN UNA SALA

**Nombre original:** Maximum Events in a Single Room<sup>14</sup>.

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor:** Yonny Mondelo Hernández.

Dada una lista de eventos, cada uno representado por sus horas de comienzo y finalización como una tupla de dos números enteros (tiempo\_comienzo, tiempo\_finalización), determinar el máximo número de eventos que pueden ser programados en una sala sin que se traslapen (no pueden haber dos eventos que compartan la sala al mismo tiempo). Los eventos no pueden ser partidos o modificados.

#### Formato de entrada:

La entrada contiene un único caso de prueba. La primera línea del caso de prueba contiene un número entero  $N$  ( $1 \leq N \leq 10^6$ ), representando el número de eventos que pueden ser programados en la sala, luego son presentadas  $N$  líneas, cada una de ellas conteniendo dos números enteros separados por un único espacio (tiempo\_comienzo, tiempo\_finalización;  $\text{tiempo\_comienzo} \leq \text{tiempo\_finalización}$ ), representando los eventos. La lista de eventos no es presentada en algún orden específico.

#### Formato de salida:

Imprimir, en una sola línea, el número entero que representa la máxima cantidad de eventos que pueden ser programados en la sala.

#### Ejemplo de entrada:

```
5
1 3
2 4
5 7
6 8
9 10
```

#### Ejemplo de salida:

```
3
```

#### Explicación:

Para el ejemplo, los eventos programados deben ser (1, 3) – (5, 7) – (9, 10).

---

<sup>14</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/events-single-room>

### 3.4.8. Máximo de eventos en múltiples salas

**Nombre original:** Maximum Events in Multiple Room<sup>15</sup>.

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor:** Yonny Mondelo Hernández.

Dada una lista de eventos, cada uno representado por sus horas de comienzo y finalización como una tupla de dos números enteros (tiempo\_comienzo, tiempo\_finalización), determinar el máximo número de eventos que pueden ser programados en múltiples salas sin que se traslapen (no pueden haber dos eventos que compartan la misma sala en el mismo tiempo). Los eventos no pueden ser partidos o modificados. Minimizar el número de salas que son utilizadas.

#### Formato de entrada:

La entrada contiene un único caso de prueba. La primera línea del caso de prueba contiene dos números enteros  $N$  ( $1 \leq N \leq 10^6$ ) y  $M$  ( $1 \leq M \leq 10^3$ ), representando, respectivamente, el número de eventos que pueden ser programados y el máximo número de salas que pueden ser utilizadas; luego, son presentadas  $N$  líneas, cada una de ellas conteniendo dos números enteros separados por un único espacio (tiempo\_comienzo, tiempo\_finalización;  $\text{tiempo\_comienzo} \leq \text{tiempo\_finalización}$ ), representando los eventos. La lista de eventos no es presentada en algún orden específico.

#### Formato de salida:

Imprimir, en una sola línea, dos números enteros que representen respectivamente, la máxima cantidad de eventos que pueden ser programados y la mínima cantidad de salas que se necesitan.

#### Ejemplo de entrada:

```
5 5
1 3
2 4
5 7
6 8
9 10
```

#### Ejemplo de salida:

```
5 2
```

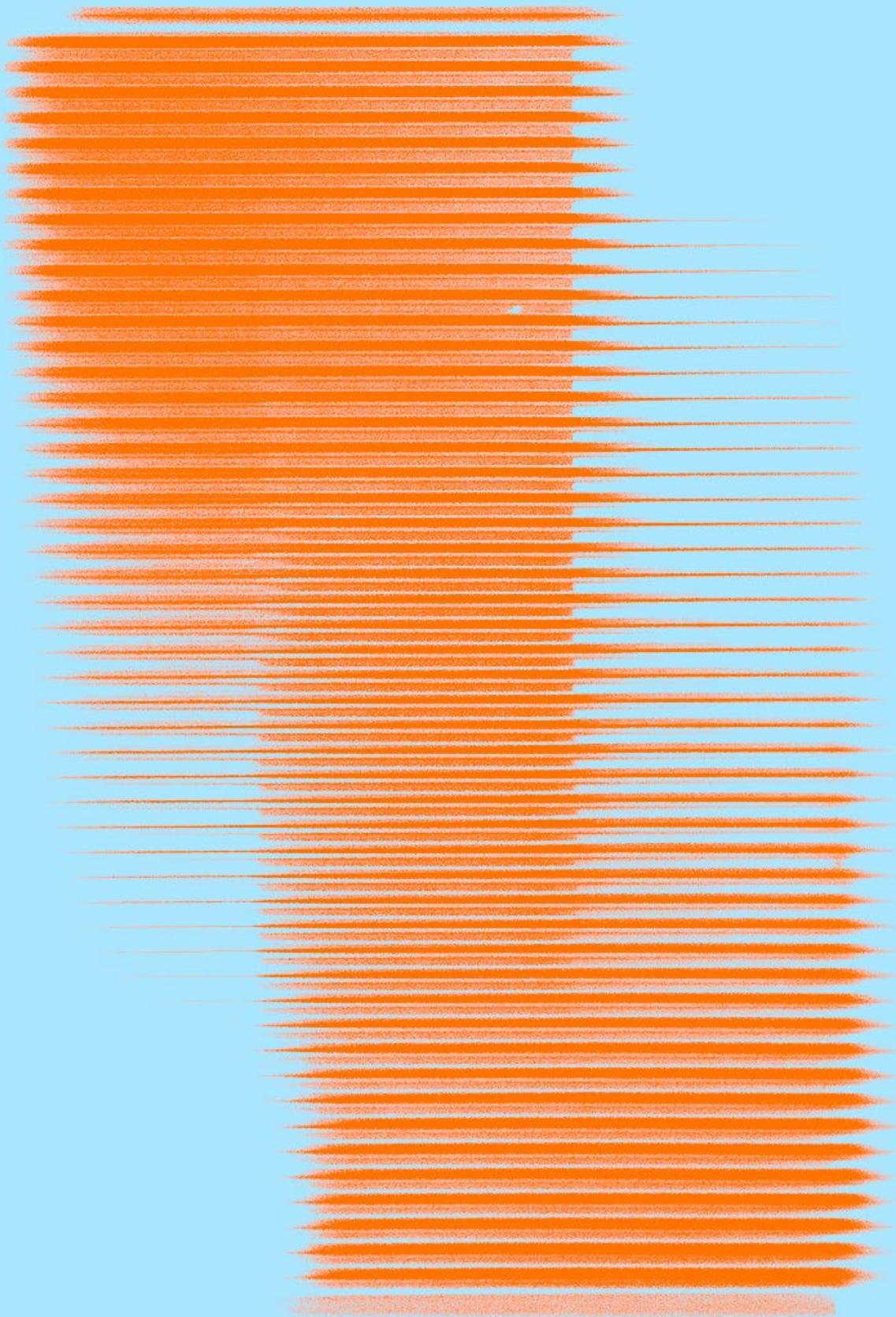
#### Explicación:

Para el ejemplo, los eventos programados deben ser  $[(1, 3) - (5, 7) - (9, 10)]$ ,  $[(2, 4) - (6, 8)]$ .

---

<sup>15</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/events-multiple-room>





## **Capítulo 4**

# **Listas, pilas y colas**



En este capítulo, se abordan inicialmente los temas de listas, pilas y colas siguiendo la línea de trabajo marcada por el Ing. César Becerra Santamaría, en su libro *Estructuras de Datos en C* [Bec95]; la calidad pedagógica del libro aún sigue teniendo vigencia a pesar de los 30 años o más que ya lleva de su publicación.

Este capítulo comienza con el uso de las instrucciones para pedir y liberar memoria de forma dinámica para su posterior uso en las estructuras que su tamaño es dinámico con el tiempo, tales como listas, pilas y colas.

El capítulo continúa con el tema de listas simplemente enlazadas, luego, la variante de las listas simplemente enlazadas donde la información es insertada de tal forma que ésta quede ordenada de forma ascendente, para su posterior uso en la simulación de colas de prioridad (donde su complejidad en tiempo de ejecución es muy alto, es un  $O(n^2)$ , donde  $n$  es la cantidad de elementos en la estructura). Luego, se presentan las listas circulares simplemente enlazadas y su posterior uso en el problema de Josefo. Las listas doblemente enlazadas, listas circulares doblemente enlazadas y su uso para resolver variantes del problema de Josefo. Después del trabajo realizado con las diferentes variantes de las listas, se evidencia que las pilas y colas son simplemente casos específicos de estas.

El capítulo termina con 8 retos de programación propuestos, para que no solamente se planteen sus respectivas soluciones con el uso de las temáticas trabajadas, sino para que también se validen en el juez en línea HackerRank<sup>1</sup>.

#### 4.1. Función malloc()

La función `malloc()` se utiliza para asignar a un apuntador una localización de memoria. La asignación de memoria es dinámica.

**Ejemplo 1:**

```
|| char *p;  
|| int n = 10;  
|| p = (char *) malloc(n);
```

---

<sup>1</sup><https://www.hackerrank.com/data-structure-utp>

En la variable *p* está almacenada una dirección que apunta a un bloque de 10 bytes.

El programa completo en Lenguaje C sería:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    char *p;
    int n = 10;
    p = (char *) malloc(n);
    strcpy(p, "Sencillo");
    printf("%s\n", p);
    return 0;
}
```



Figura 4.1. Salida del Ejemplo 1.

### Ejemplo 2:

Almacenar el número entero 821 en una localización de memoria asignada por la función `malloc()`.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    int *p;
    p = (int *) malloc(sizeof(int));
    printf("Direccion de memoria apuntada por el puntero p: %d\n", p);
    *p = 821;
    printf("Direccion de memoria apuntada por el puntero p: %d\n", p);
    printf("Contenido del bloque apuntado por p: %d\n", *p);
    return 0;
}
```

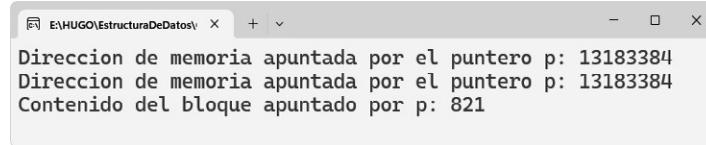


Figura 4.2. Salida del Ejemplo 2.

### Ejemplo 3:

En el siguiente código se trabaja con una estructura propia:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

typedef struct R
{
    int a;
    float b;
    double c;
    char *p;
} estructura;

int main()
{
    int n = 10;
    estructura *q;
    q = (estructura *) malloc(sizeof(estructura));
    q->a = 8;
    q->b = 10.31;
    q->c = 10.32;
    q->p = (char *) malloc(n);
    strcpy(q->p, "Cadena");
    printf("%d %.2f %.2lf %s\n", q->a, q->b, q->c, q->p);
    return 0;
}
```



Figura 4.3. Salida del Ejemplo 3.

## 4.2. Función free()

La función `free()` permite liberar bloques de memoria dinámica que han sido asignados por la función `malloc()`.

### Ejemplo 4:

¿En el siguiente programa de Lenguaje C seda un buen manejo de la memoria?

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
```

```
    char *p, *q;
    int n = 12;
    p = (char *) malloc(n);
    strcpy(p, "Pascal");
    q = (char *) malloc(n);
    strcpy(q, "Lenguaje C");
    printf("%s, %s\n", p, q);
    q = p;
    printf("%s, %s\n", p, q);

    return 0;
}
```



Figura 4.4. Salida del Ejemplo 4.

No se hace un buen uso de la memoria; el bloque de memoria que originalmente estaba siendo apuntado por *q* ahora queda como un bloque de memoria “zombi”, el cual no se puede volver a utilizar hasta que se termine la ejecución del programa. Lo que se debe hacer es liberar previamente el bloque de memoria apuntada por *q* con la instrucción `free()`, así como se hace en el siguiente programa:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

int main()
{
    char *p, *q;
    int n = 12;
    p = (char *) malloc(n);
    strcpy(p, "Pascal");
    q = (char *) malloc(n);
    strcpy(q, "Lenguaje C");
    printf("%s, %s\n", p, q);
    /* la función free libera memoria dinámica
       asignada con la función malloc(); */
    free(q);
    q = p;
    printf("%s, %s\n", p, q);

    return 0;
}
```

### Ejemplo 5:

¿Cuál es el resultado que se genera por la ejecución del siguiente programa, en Lenguaje C?

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

void SumarAyB(int a, int b, int c)
{
    c = a + b;
}

void SumarAyBprima(int a, int b, int *c)
{
    *c = a + b;
}

int main()
{
    int a = 10, b = 15, c = 0;

    SumarAyB(a, b, c);
    printf("El resultado de %d + %d es: %d\n", a, b, c);

    SumarAyBprima(a, b, &c);
    printf("El resultado de %d + %d es: %d\n", a, b, c);

    return 0;
}

```

El resultado que se genera por la ejecución del programa es el siguiente:

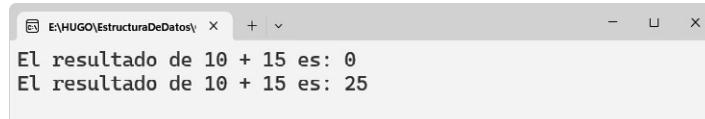


Figura 4.5. Resultado generado por el programa del Ejemplo 5.

La justificación del resultado generado se encuentra a continuación:

En la función `void SumarAyB(int a, int b, int c)` los tres parámetros se reciben por valor, por lo tanto, el valor que se le asigna a la variable `c` solo está en el alcance de la función y no está reflejado por fuera de ella, en consecuencia, en la función `main()` su valor sigue siendo de 0.

En la función `void SumarAyBprima(int a, int b, int *c)` las variables `a` y `b` se reciben por valor, mientras que la variable `c` se recibe por referencia, por lo tanto, el valor que se le asigna a la variable `c` se ve reflejado en la función `main()`, donde su valor ahora es 25.

### 4.3. Concepto de lista

Una lista es un conjunto de variables encadenadas en sí a través de un apuntador. Cada variable se denomina nodo. Un nodo es una variable compuesta por dos partes; la información (key) y un apuntador a la próxima variable.

### Ejemplo 6:

En el siguiente programa se crea una lista que contiene los números 1, 2 y 3:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

int main()
{
    struct node *head, *q;
    head = (struct node *) malloc(sizeof(struct node));
    head->key = 1;
    q = (struct node *) malloc(sizeof(struct node));
    q->key = 2;
    head->next = q;
    q->next = (struct node *) malloc(sizeof(struct node));
    q->next->key = 3;
    q->next->next = NULL;
    printf("%d -> %d -> %d\n",
           head->key, head->next->key, head->next->next->key);

    return 0;
}
```



Figura 4.6. Salida del programa en el cual se trabaja el Ejemplo 6.

### Ejemplo 7:

En el siguiente programa se crea una lista que contiene los números del 1 al 100, después se recorre la lista desde el primero hasta el último nodo imprimiendo la información, por último, se recorre la lista borrando uno a uno cada nodo.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};
```

```

int main()
{
    struct node *head, *newNode, *currentNode;
    int n = 100;
    head = NULL;

    while(n >= 1)
    {
        newNode = (struct node *) malloc(sizeof(struct node));
        newNode->key = n;
        newNode->next = head;
        head = newNode;
        n--;
    }

    currentNode = head;

    while(currentNode != NULL)
    {
        printf("%d -> ", currentNode->key);
        currentNode = currentNode->next;
    }
    printf("NULL\n");

    while(head != NULL)
    {
        currentNode = head;
        head = head->next;
        free(currentNode);
    }

    return 0;
}

```

```

1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21
-> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 ->
31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40
-> 41 -> 42 -> 43 -> 44 -> 45 -> 46 -> 47 -> 48 -> 49 ->
50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56 -> 57 -> 58 -> 59
-> 60 -> 61 -> 62 -> 63 -> 64 -> 65 -> 66 -> 67 -> 68 ->
69 -> 70 -> 71 -> 72 -> 73 -> 74 -> 75 -> 76 -> 77 -> 78
-> 79 -> 80 -> 81 -> 82 -> 83 -> 84 -> 85 -> 86 -> 87 ->
88 -> 89 -> 90 -> 91 -> 92 -> 93 -> 94 -> 95 -> 96 -> 97
-> 98 -> 99 -> 100 -> NULL

```

Figura 4.7. Salida del programa en el cual se trabaja el Ejemplo 7.

**Ejemplo 8:**

Hacer un programa en Lenguaje C que realice el mismo trabajo que el realizado por el Ejemplo 7, pero ahora trabajando con funciones, para lo cual el programa debe tener:

- 1.) Una función para crear una lista ordenada de forma ascendente con los primeros  $n$  números enteros positivos. La función debe recibir como parámetro el número entero positivo  $n$  y debe devolver un puntero a la cabeza de la lista.

- 2.) Una función para imprimir el contenido de la lista, para lo cual la función debe recibir un puntero a la cabeza de la lista.
- 3.) Una función para borrar todos los elementos de la lista, para lo cual la función debe recibir un puntero a la cabeza de la lista y debe retornar el puntero de la cabeza de la lista apuntando a una lista vacía (apuntando a NULL).

El siguiente programa cumple con los requerimientos pedidos:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *MakeLinkedList(int n)
{
    struct node *head, *newNode;
    head = NULL;

    while(n >= 1)
    {
        newNode = (struct node *) malloc(sizeof(struct node));
        newNode->key = n;
        newNode->next = head;
        head = newNode;
        n--;
    }

    return head;
}

void PrintLinkedList(struct node *head)
{
    struct node *currentNode;
    currentNode = head;

    while(currentNode != NULL)
    {
        printf("%d -> ", currentNode->key);
        currentNode = currentNode->next;
    }
    printf("NULL\n");
}

struct node *DeleteLinkedList(struct node *head)
{
    struct node *currentNode;

    while(head != NULL)
    {
        currentNode = head;
        head = head->next;
```

```

        free(currentNode);
    }

    return head;
}

int main()
{
    struct node *head;
    int n;

    while (scanf("%d", &n) != EOF)
    {
        head = MakeLinkedList(n);
        PrintLinkedList(head);
        head = DeleteLinkedList(head);
        PrintLinkedList(head);
    }

    return 0;
}

```

```

100
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9 -> 10 -> 11 ->
12 -> 13 -> 14 -> 15 -> 16 -> 17 -> 18 -> 19 -> 20 -> 21
-> 22 -> 23 -> 24 -> 25 -> 26 -> 27 -> 28 -> 29 -> 30 ->
31 -> 32 -> 33 -> 34 -> 35 -> 36 -> 37 -> 38 -> 39 -> 40
-> 41 -> 42 -> 43 -> 44 -> 45 -> 46 -> 47 -> 48 -> 49 ->
50 -> 51 -> 52 -> 53 -> 54 -> 55 -> 56 -> 57 -> 58 -> 59
-> 60 -> 61 -> 62 -> 63 -> 64 -> 65 -> 66 -> 67 -> 68 ->
69 -> 70 -> 71 -> 72 -> 73 -> 74 -> 75 -> 76 -> 77 -> 78
-> 79 -> 80 -> 81 -> 82 -> 83 -> 84 -> 85 -> 86 -> 87 ->
88 -> 89 -> 90 -> 91 -> 92 -> 93 -> 94 -> 95 -> 96 -> 97
-> 98 -> 99 -> 100 -> NULL
NULL

```

Figura 4.8. Salida por pantalla del programa de listas que hace uso de funciones.

### Ejemplo 9:

Hacer un programa en Lenguaje C que permita trabajar con listas ordenadas de forma ascendente, para lo cual el programa debe tener:

- 1.) Una función que permita insertar un elemento en una lista que está ordenada de forma ascendente; la lista debe seguir conteniendo sus elementos en orden ascendente después de la inserción del elemento. El prototipo de la función tiene que ser el siguiente: recibir como parámetros el puntero a la cabeza de la lista, el número entero a insertar y devolver el puntero a la cabeza de la lista.
- 2.) Una función para imprimir el contenido de la lista, para lo cual la función tiene que recibir un puntero a la cabeza de la lista.
- 3.) Una función que permita borrar la primera ocurrencia de un elemento en una lista que está ordenada de forma ascendente; la lista seguirá conteniendo sus elementos en orden

ascendente después del borrado del elemento. El prototipo de la función tiene que ser el siguiente: recibir como parámetros el puntero a la cabeza de la lista, el número entero a borrar y devolver el puntero a la cabeza de la lista.

La solución en Lenguaje C es la siguiente:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *InsertElementInAscendantLinkedList(struct node *head,
                                                int element)
{
    struct node *newNode, *previous, *current;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(head == NULL)
    {
        newNode->next = head;
        head = newNode;
    }
    else
    {
        if(element <= head->key)
        {
            newNode->next = head;
            head = newNode;
        }
        else
        {
            previous = head;
            current = head->next;

            while(current != NULL)
            {
                if(element > current->key)
                {
                    previous = current;
                    current = current->next;
                }
                else
                    break;
            }

            newNode->next = current;
            previous->next = newNode;
        }
    }
}

return head;
}
```

```

void PrintAscendantLinkedList(struct node *head)
{
    struct node *current;
    current = head;

    while(current != NULL)
    {
        printf("%d -> ", current->key);
        current = current->next;
    }

    printf("NULL\n");
}

struct node *DeleteElementOfAscendantLinkedList(struct node *head,
                                                int element)
{
    struct node *previous, *current;

    if(head == NULL)
        printf("The ascendant linked list is empty.\n");
    else
    {
        if(element < head->key)
            printf("The %d is not in the ascendant linked list.\n",
                   element);
        else
        {
            if(element == head->key)
            {
                current = head;
                head = head->next;
                free(current);
            }
            else
            {
                previous = head;
                current = head->next;

                while(current != NULL)
                {
                    if(element > current->key)
                    {
                        previous = current;
                        current = current->next;
                    }
                    else
                        break;
                }

                if(current == NULL)
                    printf("The %d is not in the ascendant linked list.\n",
                           element);
                else
                {
                    if(current->key != element)
                        printf("The %d is not in the ascendant linked list.
                               \n", element);
                }
            }
        }
    }
}

```

```
        else
        {
            previous->next = current->next;
            free(current);
        }
    }
}

return head;
}

int main()
{
    int operation, element;
    struct node *head;
    head = NULL;

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            head = InsertElementInAscendantLinkedList(head, element);
            PrintAscendantLinkedList(head);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                head = DeleteElementOfAscendantLinkedList(head, element);
                PrintAscendantLinkedList(head);
            }
            else
                printf("Bad use. \n 1. Insert \n 2. Delete\n");
        }
    }
    return 0;
}
```

```

E:\HUGO\EstructuraDeDatos\ > +
1 10
10 -> NULL
1 5
5 -> 10 -> NULL
1 20
5 -> 10 -> 20 -> NULL
1 12
5 -> 10 -> 12 -> 20 -> NULL
2 5
10 -> 12 -> 20 -> NULL
2 12
10 -> 20 -> NULL
2 20
10 -> NULL
1 25
10 -> 25 -> NULL
2 25
10 -> 25 -> NULL
2 30
The 20 is not in the descendant linked list.
10 -> 25 -> NULL
2 30
The 30 is not in the descendant linked list.
10 -> 25 -> NULL

```

Figura 4.9. Salida por pantalla del programa que manipula una lista ordenada de forma ascendente.

#### 4.3.1. Reto de programación: Sumar todos (versión 2025 - talla pequeña)

**Nombre original:** Add All (version 2025) - Small<sup>2</sup>.

**Fuente:** Curso de Estructura de Datos. Universidad Tecnológica de Pereira.

**Fecha:** 1ro de Enero de 2025.

**Autor:** Hugo Humberto Morales Peña.

El profesor Humberto Moralov, al organizar los papeles de su oficina, se encontró con la siguiente solución que planteó aproximadamente 10 años atrás para el reto de programación:  
**UVa - 10954 - Add All.**

```

#include <stdio.h>
#define MAXT 5000

int ExtractMin(int A[], int *n)
{
    int temp, i, min;
    for(i = *n - 1; i >= 1; i--)
    {
        if(A[*n] > A[i])
        {
            temp = A[i];
            A[i] = A[*n];
            A[*n] = temp;
        }
    }
    min = A[*n];
    *n = *n - 1;
}

```

<sup>2</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/add-all-version-2025-small>

```

        return min;
    }

void Insert(int A[], int *n, int element)
{
    *n = *n + 1;
    A[*n] = element;
}

int AddAll(int A[], int n)
{
    int result = 0, value1, value2, value3;
    while(n >= 2)
    {
        value1 = ExtractMin(A, &n);
        value2 = ExtractMin(A, &n);
        value3 = value1 + value2;
        result = result + value3;
        Insert(A, &n, value3);
    }
    return result;
}

int main()
{
    int n, A[MAXT + 1], index;
    while(scanf("%d", &n) && (n > 0))
    {
        for(index = 1; index <= n; index++)
            scanf("%d", &A[index]);
        printf("%d\n", AddAll(A, n));
    }
    return 0;
}

```

En la solución anterior se trabaja con un arreglo  $A[ ]$  en el que se almacenan inicialmente  $n$  elementos. En la función `AddAll` en el ciclo de repetición `while` se extraen (y eliminan) del arreglo los dos valores más pequeños, posteriormente se suman y el resultado se inserta (se almacena) en el arreglo; ese valor también se utiliza para actualizar la suma total en la variable `result`, donde se almacena el resultado de sumar todos los pares de valores más pequeños que hay en el arreglo mientras este contenga 2 o más valores. El costo computacional de la función `AddAll` es  $O(n^2)$ , donde  $n$  es la cantidad de elementos almacenados en el arreglo.

El ciclo `while` se ejecuta  $n - 1$  veces, porque, por cada iteración, el arreglo tienen un número menos almacenado. El costo de cada iteración es un  $O(n)$ , porque se tienen que recorrer los  $n$  elementos almacenados en el arreglo  $A[ ]$  para determinar el valor mínimo, es decir,  $(n - 1) \cdot O(n) = O(n^2)$ .

Un  $O(n^2)$  es una solución muy costosa en tiempo de ejecución; el único motivo por el cual el juez en línea UVa da un veredicto de `accepted`, es porque es un reto de programación muy viejo (año 2005) y en ese entonces una talla de  $n = 5,000$  era suficiente para un tiempo de 1 o 2 segundos, con el poder de computo de los servidores de esa época.

Actualmente (año 2026), una solución del orden de  $10^8$  operaciones corre en 1 o 2 segundos en los jueces en línea, por este motivo el profesor Moralov le pide a usted, como estudiante de su

curso de Estructuras de Datos, construir una solución que utilice *listas enlazadas ordenadas de forma ascendente* para este reto de programación; esta no es la estructura de datos más eficiente para resolver este problema, pero con tallas pequeñas para  $n$  funciona en un tiempo aceptable.

#### **Formato de entrada:**

La entrada comienza con un entero positivo  $t$  ( $1 \leq t \leq 10$ ), denotando el número de casos de prueba. Cada caso de prueba está conformado por dos líneas. La primera línea, consta de un número entero positivo  $n$  ( $2 \leq n \leq 10^4$ ), que representa la cantidad de elementos para almacenar en el arreglo  $A[ ]$ . La segunda línea, consta de  $n$  números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a  $10^6$ ).

Se garantiza que la suma de todos los valores de  $n$  en los casos de prueba es a lo sumo  $10^4$ .

#### **Formato de salida:**

Para cada caso de prueba, imprimir una sola línea con el resultado generado por la función `AddAll`.

#### **Ejemplo de entrada:**

```
5
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
5
3 2 5 1 4
5
1 1 1 1 1
```

#### **Ejemplo de salida:**

```
9
19
33
33
12
```

#### **Solución del reto**

Para el reto de programación original (con una talla de  $n = 10^6$ ), ya se planteó una solución eficiente en la sección 3.3.1 que hace uso de la estructura de datos de colas de prioridad. Ahora, se planteará una solución utilizando la estructura de datos de listas enlazadas ordenadas de forma ascendente; solución que desde el punto de vista de complejidad computacional en tiempo de ejecución es muy ineficiente. En el peor de los casos, la inserción de un elemento tiene que recorrer uno a uno los  $n$  elementos de la lista ordenada, por lo tanto, dicha operación

tienen un costo computacional de  $O(n)$ . Lo más costoso en tiempo de ejecución en la siguiente solución, es insertar los  $n$  elementos en la lista ordenada, donde se tiene un costo computacional de  $n \cdot O(n) = O(n^2)$ .

En esta variante del reto de programación,  $n$  toma como máximo un valor de  $10^4$ ; para este valor el orden de la cantidad de operaciones es  $n^2 = (10^4)^2 = 10^8$ . Máximo de operaciones que no se supera, debido a la restricción en la que se indica que al sumar todos los valores de  $n$  en los  $t$  casos de prueba el resultado no excede el  $10^4$ , por lo tanto, el orden del total de operaciones para correr todos los casos de prueba es a lo sumo un  $10^8$ .

La solución en Lenguaje C utilizando *listas enlazadas ordenadas de forma ascendente* es:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

struct node
{
    long long int key;
    struct node *next;
};

struct node *InsertElementInAscendantLinkedList(struct node *head,
                                                long long int element)
{
    struct node *newNode, *previous, *current;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(head == NULL)
    {
        newNode->next = head;
        head = newNode;
    }
    else
    {
        if(element <= head->key)
        {
            newNode->next = head;
            head = newNode;
        }
        else
        {
            previous = head;
            current = head->next;
            while(current != NULL)
            {
                if(element > current->key)
                {
                    previous = current;
                    current = current->next;
                }
                else
                    break;
            }
        }
    }
}
```

```

        newNode->next = current;
        previous->next = newNode;
    }
}

return head;
}

struct node *DeleteElementOfAscendantLinkedList(struct node *head,
                                              long long int element)
{
    struct node *previous, *current;

    if(head == NULL)
        printf("The ascendant linked list is empty.\n");
    else
    {
        if(element < head->key)
            printf("The %lld is not in the ascendant linked list.\n",
                   element);
        else
        {
            if(element == head->key)
            {
                current = head;
                head = head->next;
                free(current);
            }
            else
            {
                previous = head;
                current = head->next;

                while(current != NULL)
                {
                    if(element > current->key)
                    {
                        previous = current;
                        current = current->next;
                    }
                    else
                        break;
                }

                if(current == NULL)
                    printf("The %lld is not in the ascendant linked list.
                           \n", element);
                else
                {
                    if(current->key != element)
                        printf("The %lld is not in the ascendant linked
                               list.\n", element);
                    else
                    {
                        previous->next = current->next;
                        free(current);
                    }
                }
            }
        }
    }
}

```

```
        }
    }

    return head;
}

long long int AddAll(struct node *head)
{
    long long result = 0, value1, value2, value3;

    while(head->next != NULL)
    {
        value1 = head->key;
        head = DeleteElementOfAscendantLinkedList(head, value1);
        value2 = head->key;
        head = DeleteElementOfAscendantLinkedList(head, value2);
        value3 = value1 + value2;
        result = result + value3;
        head = InsertElementInAscendantLinkedList(head, value3);
    }

    free(head);
    return result;
}

int main()
{
    int n, i, totalCases, idCase;
    long long int key;
    struct node *head;

    scanf("%d", &totalCases);

    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        head = NULL;
        scanf("%d", &n);

        for(i=1; i<=n; i++)
        {
            scanf("%lld", &key);
            head = InsertElementInAscendantLinkedList(head, key);
        }

        printf("%lld\n", AddAll(head));
    }

    return 0;
}
```

Salida del programa anterior ejecutando los ejemplos del reto:

```

5
3
1 2 3
9
4
1 2 3 4
19
5
5 4 3 2 1
33
5
3 2 5 1 4
33
5
1 1 1 1 1
12

```

Figura 4.10. Salida del programa para el reto: Sumar todos (versión 2025 - talla pequeña).

## 4.4. Listas circulares simplemente enlazadas

Una lista circular simplemente enlazada es aquella donde el campo `next` del último elemento en la lista (`tail`) apunta al primer elemento en la lista, como se ilustra en la Figura 4.11.

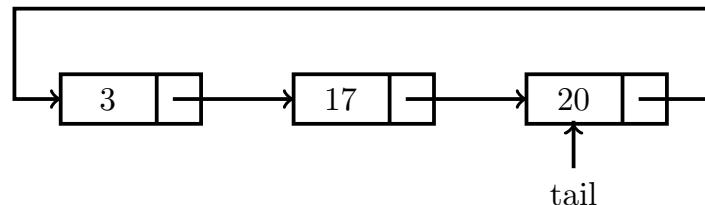


Figura 4.11. Ejemplo de lista circular simplemente enlazada.

### 4.4.1. Inserción de un elemento en una lista circular simplemente enlazada

Por el momento se considerará que, después de insertar un elemento en la lista circular, este quedará en la última posición de la lista; adicionalmente, el puntero a la lista será `tail` y apuntará al último nodo en la lista, esto resulta ser muy estratégico: el siguiente del último nodo es el primer nodo en la lista.

Inicialmente tenemos un lista circular vacía, como se ilustra en la Figura 4.12.



Figura 4.12. Inicialmente la lista circular está vacía **tail** apunta a **NULL**.

Si se quiere insertar el número 3 en la lista circular, entonces esta debería de quedar como se ilustra en la Figura 4.13.

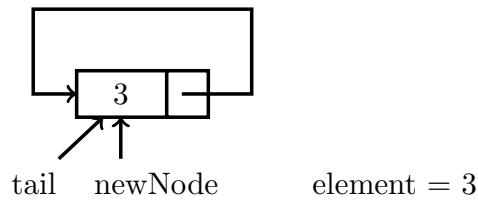


Figura 4.13. Lista circular que contiene un solo nodo con el valor 3.

El código en Lenguaje C que considera este caso es:

```
struct node *InsertElementInCircularLinkedList(struct node *tail,
                                              int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        /* falta el código para el caso en el cual la lista circular
           contiene al menos un nodo. */
    }

    return tail;
}
```

Ahora, considerar el caso en el cual se quiere insertar el número 17 en la lista circular que contiene un solo nodo con el valor de 3; el paso a paso de lo que se debería de hacer se ilustra en las Figuras 4.14, 4.15 y 4.16.

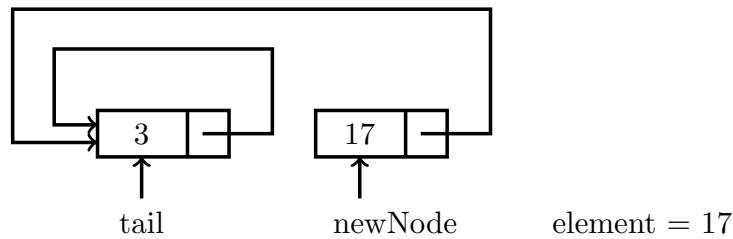


Figura 4.14. Instrucción que enlaza el campo **next** del **newNode** al nodo apuntado por **tail→next**.

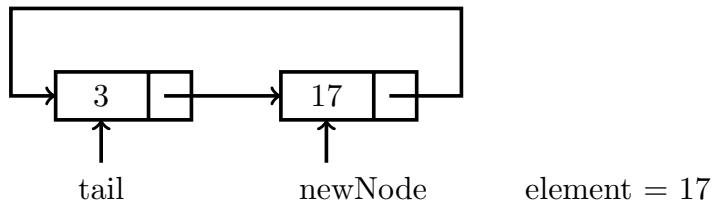


Figura 4.15. Instrucción que enlaza el campo **next** del **tail** al nodo apuntado por el **newNode**.

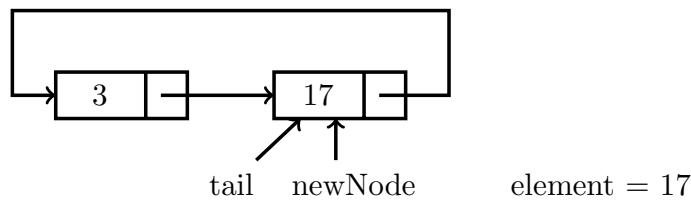


Figura 4.16. Instrucción que actualiza el puntero **tail** al **newNode**, al nuevo último.

El código en Lenguaje C de la función completa es:

```

struct node *InsertElementInCircularLinkedList(struct node *tail,
                                              int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

```

#### 4.4.2. Borrado del primer nodo en una lista circular simplemente enlazada

El código en Lenguaje C para borrar el primer nodo de la lista es el siguiente:

```
struct node *DeleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }

    return tail;
}
```

#### 4.4.3. Impresión desde el primero hasta el último de los elementos de una lista circular simplemente enlazada

El código en Lenguaje C para imprimir los elementos de una lista circular simplemente enlazada es el siguiente:

```
void PrintCircularLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;

        while(currentNode != tail)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}
```

#### 4.4.4. Programa completo para el mantenimiento de una lista circular simplemente enlazada

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *InsertElementInCircularLinkedList(struct node *tail,
                                              int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *DeleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }

    return tail;
}

```

```
void PrintCircularLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;

        while(currentNode != tail)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

    tail = NULL;

    while(scanf(" %d", &operation) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            scanf(" %d", &element);
            tail = InsertElementInCircularLinkedList(tail, element);
            PrintCircularLinkedList(tail);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                tail = DeleteFirstNodeInCircularLinkedList(tail);
                PrintCircularLinkedList(tail);
            }
            else
                printf("Bad use. \n 1. Insert\n 2. Delete\n");
        }
    }

    return 0;
}
```

```

E:\HUGO\EstructuraDeDatos\ x + v
1 10
10 ...
1 20
10 -> 20 ...
1 30
10 -> 20 -> 30 ...
2
20 -> 30 ...
1 5
20 -> 30 -> 5 ...
2
30 -> 5 ...

```

Figura 4.17. Salida del programa para el mantenimiento de listas circulares simplemente enlazadas.

#### 4.4.5. Reto de programación: Problema de Josefo

**Nombre original:** El Problema de Josefo<sup>3</sup> <sup>4</sup>.

**Fuente:** Curso de Estructura de Datos. Universidad Tecnológica de Pereira.

**Fecha:** 1ro de Enero de 2024.

**Autor:** Hugo Humberto Morales Peña.

El Problema de Josefo (Josephus) hace referencia a Flavio Josefo, un historiador judío que vivió en el siglo I. Según lo que cuenta Josefo, un grupo de 40 soldados (entre los cuales se encontraba él) estaban atrapados en una cueva, rodeados de romanos. Decidieron hacer un pacto de muerte antes que ser capturados por sus enemigos, donde echarían a suertes quién mataría a quién; el último que quedara debería suicidarse.

Formalmente el problema de Josefo es: hay  $n$  personas paradas en un círculo (mirando al interior de éste) esperando a ser ejecutadas. Una de las personas se toma como punto de partida (posición uno) y a partir de ella se enumeran a todos los demás de forma consecutiva siguiendo el sentido de las manecillas del reloj, desde 1 hasta  $n$ . Posteriormente, se comienza a contar desde la persona de la posición 1 evitando a  $k - 1$  personas y la persona número  $k$  es ejecutada. Ahora, se comienza a contar nuevamente desde la persona que originalmente estaba en la posición  $k + 1$ , evitando a  $k - 1$  personas y la persona número  $k$  es ejecutada. La eliminación continúa siguiendo siempre el sentido de las manecillas del reloj alrededor del círculo, el cual se hace cada vez más pequeño, hasta que sólo queda la última persona, la cual decidirá si vive o muere, si se entrega a los romanos o si cumple el pacto de muerte y se suicida, obviamente conocemos la decisión que tomó Josefo, gracias a la cual se conoce esta historia.

Josefo necesita de tu ayuda para poder determinar la posición en el círculo en la cual debe pararse, para que sea el último soldado que quede con vida y pueda entregarse a los romanos.

<sup>3</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/josephuss-problem>

<sup>4</sup>Este reto de programación se generó a partir de la sección “1.17 El famoso problema de Josephus” del libro *Estructuras de Datos en C* [Bec95]

**Formato de entrada:**

La entrada contiene varios casos de prueba. Cada caso está compuesto de una sola línea, que contienen dos números enteros positivos  $n$  ( $1 \leq n \leq 10^4$ ) y  $k$  ( $1 \leq k \leq n$ ) que representan respectivamente el número de soldados atrapados en la cueva y el valor del movimiento para asesinar soldados en el círculo. La entrada finaliza con un caso de prueba que contiene dos ceros, el cual no debe ser procesado.

Se garantiza que la suma de todos los valores de  $n$  en los casos de prueba es a lo sumo  $10^4$ .

**Formato de salida:**

Para cada caso de prueba de la entrada, su programa debe imprimir, en una sola línea, el número entero que representa posición en el círculo en la cual se debe parar Josefo para ser el último soldado que quede con vida.

**Ejemplo de entrada:**

```
1 1
10 1
10 5
10 10
5 5
5 4
0 0
```

**Ejemplo de salida:**

```
1
10
3
8
2
1
```

**Solución del reto**

Se plantea una solución utilizando listas circulares simplemente enlazadas. Para cada caso de prueba, se insertan los elementos del 1 al  $n$  en la lista circular. Ahora, una observación clave: la función que elimina el primer nodo de la lista circular en esencia elimina el nodo siguiente del nodo apuntado por `tail`, entonces es simplemente mover el puntero `tail` al nodo que precede al nodo que se debe borrar y hacer uso de dicha función para borrar  $n - 1$  elementos de la lista circular. El costo computacional en tiempo de ejecución de la solución es eliminar  $n - 1$  nodos de la lista circular, realizando  $k - 1$  movimientos entre los nodos para garantizar que el nodo a eliminar es el que se encuentra en el movimiento  $k$ , es decir,  $(n - 1) \cdot (k - 1) = n \cdot k - n - k + 1 = O(n \cdot k) = O(k \cdot n)$ ; como  $k$  está acotado superiormente por  $n$ , entonces la solución en el peor de los casos es un  $O(n^2)$  por caso de prueba.

La solución en Lenguaje C utilizando listas circulares simplemente enlazadas es:

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
};

struct node *InsertElementInCircularLinkedList(struct node *tail,
                                              int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *DeleteFirstNodeInCircularLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
    }

    return tail;
}

```

```
int main()
{
    struct node *tail;
    int n, k, i, j;
    tail = NULL;

    while(scanf("%d %d", &n, &k) && (n > 0) && (k > 0))
    {
        for(i = 1; i <= n; i++)
            tail = InsertElementInCircularLinkedList(tail, i);

        for(i = 1; i < n; i++)
        {
            for(j = 1; j < k; j++)
                tail = tail->next;

            tail = DeleteFirstNodeInCircularLinkedList(tail);
        }
        printf("%d\n", tail->key);
        tail = DeleteFirstNodeInCircularLinkedList(tail);
    }

    return 0;
}
```

```
1 1
1
10 1
10
10
10 5
3
10 10
8
5 5
2
5 4
1
0 0
```

Figura 4.18. Salida del programa para el reto: Problema de Josefo.

## 4.5. Listas circulares doblemente enlazadas

A la estructura de datos `node` que se ha utilizado en las listas simplemente enlazadas hay que agregarle un puntero del tipo `struct node` al nodo previo, para que de esta forma un nodo tenga un puntero al nodo siguiente y un puntero al nodo previo (nodo anterior). El código en Lenguaje C de la estructura sería:

```
struct node
{
    int key;
    struct node *prev;
    struct node *next;
};
```

Una lista circular doblemente enlazada es aquella donde el campo `next` del último nodo en la lista (`tail`) apunta al primer nodo y el campo `prev` del primer nodo apunta al último nodo.

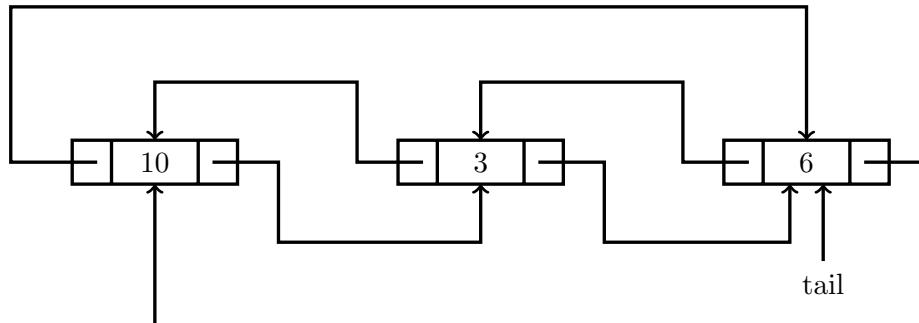


Figura 4.19. Ejemplo de lista circular doblemente enlazada.

#### 4.5.1. Inserción de un elemento en una lista circular doblemente enlazada

Por el momento, se considerará que después de insertar un elemento en la lista circular doblemente enlazada este quedará en la última posición de la lista; adicionalmente, el puntero a la lista será `tail` y el cual apuntará al último elemento en la lista, esto resulta ser muy estratégico. El siguiente del último nodo es el primer nodo en la lista y el nodo previo del primer nodo es el último nodo.

Inicialmente, tenemos una lista circular doblemente enlazada vacía, como se ilustra en la Figura 4.20.



Figura 4.20. Inicialmente la lista circular doblemente enlazada está vacía `tail` apunta a `NULL`.

Si se quiere insertar el número 3 en la lista circular doblemente enlazada, entonces ésta debería de quedar como se ilustra en la Figura 4.21.

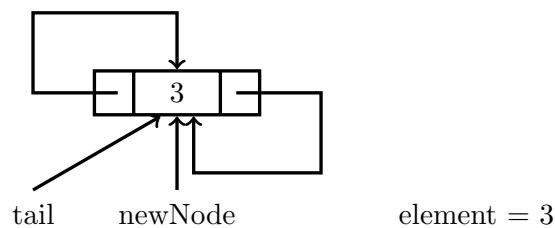


Figura 4.21. Lista circular doblemente enlazada que contiene un solo nodo con el valor 3.

El código en Lenguaje C que considera éste caso es:

```
struct node *InsertElementInCircularDoublyLinkedList(struct node *tail,
                                                       int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        /* falta el código para el caso en el cual la lista circular
           doblemente enlazada contiene al menos un nodo. */
    }

    return tail;
}
```

Ahora, considerar el caso en el cual se quiere insertar el número 6 en la lista circular doblemente enlazada que contiene un solo nodo con el valor de 3, el paso a paso de lo que se debería de hacer se ilustra en las Figuras 4.22, 4.23, 4.24, 4.25, 4.26 y 4.27.

Primero, se debe pedir de forma dinámica memoria para un nodo nuevo el cual es apuntado por `newNode`; luego, se debe asignar en el campo `key` el valor de 6.

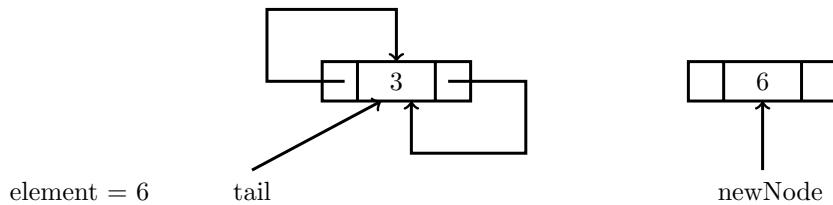


Figura 4.22. Se genera un nuevo nodo el cual es apuntado por `newNode`, en el campo `key` es almacenado el número 6.

Segundo, el campo **next** del nodo apuntado por **newNode** se pone a apuntar al nodo apuntado por **tail→next**

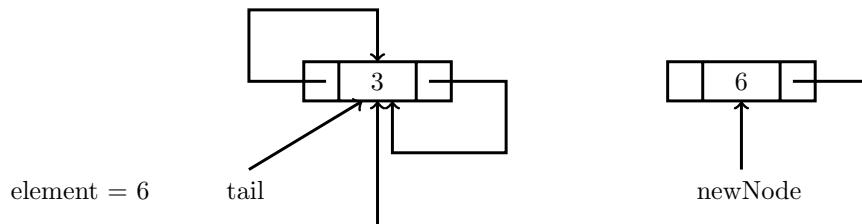


Figura 4.23. Se enlaza el campo **next** del **newNode** al nodo apuntado por **tail→next**.

Tercero, el campo **prev** del nodo apuntado por **newNode→next** se pone a apuntar al nodo apuntado por **newNode**

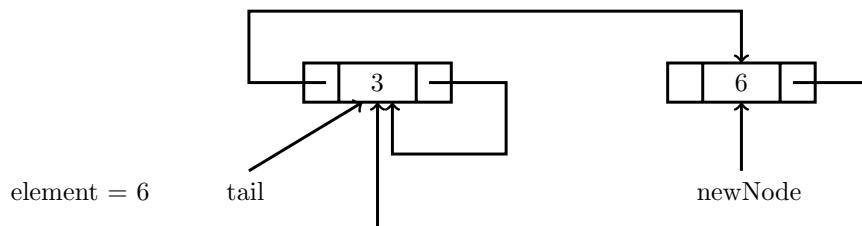


Figura 4.24. Se enlaza el campo **prev** del primer nodo de la lista al nodo apuntado por el **newNode**.

Cuarto, el campo **prev** del nodo apuntado por **newNode** se pone a apuntar al nodo apuntado por **tail**.

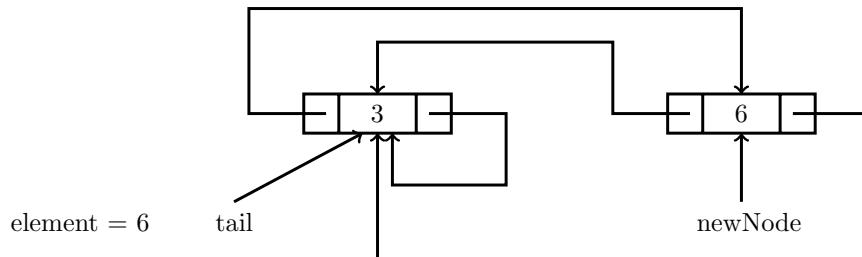


Figura 4.25. Se enlaza el campo **prev** del **newNode** al nodo apuntado por **tail**.

Quinto, el campo **next** del nodo apuntado por **tail** se pone a apuntar al nodo apuntado por **newNode**.

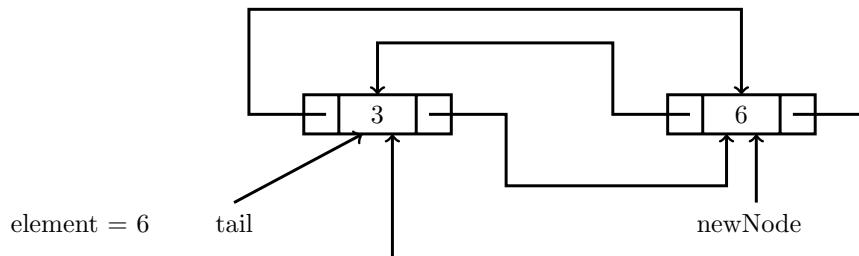


Figura 4.26. Se enlaza el campo **next** de **tail** al nodo apuntado por **newNode**.

Sexto, por último, el puntero **tail** se pone a apuntar al nodo apuntado por **newNode**, dicho nodo ahora es el último en la lista circular doblemente enlazada.

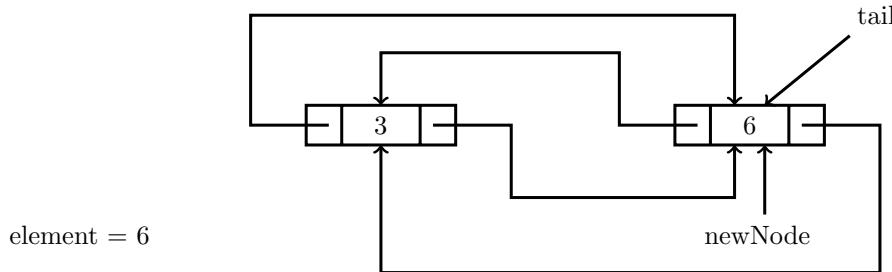


Figura 4.27. Se actualiza el puntero **tail** al **newNode**, al nuevo último.

El código en Lenguaje C de la función completa es:

```

struct node *InsertElementInCircularDoublyLinkedList(struct node **tail,
                                                       int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

```

#### 4.5.2. Borrado del primer nodo en una lista circular doblemente enlazada

El paso a paso para borrar el primer nodo de una lista circular doblemente enlazada se ilustra en las Figuras 4.28, 4.29, 4.30, 4.31 y 4.32.

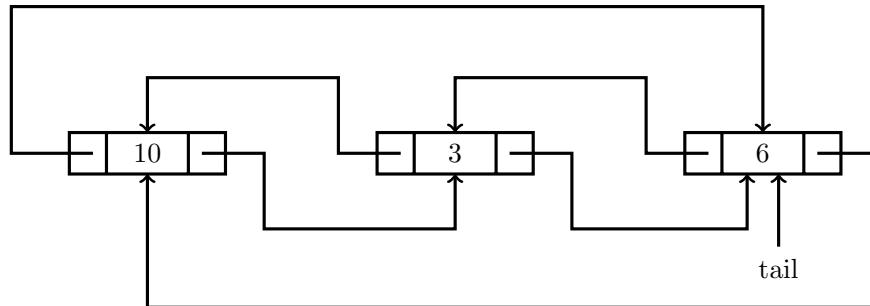


Figura 4.28. Lista circular doblemente enlazada de la cual se borrará el primer nodo.

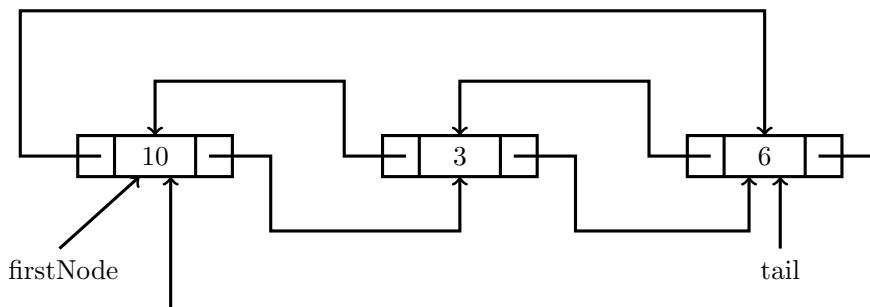


Figura 4.29. Se enlaza el puntero **firstNode** al nodo apuntado por **tail→next**.

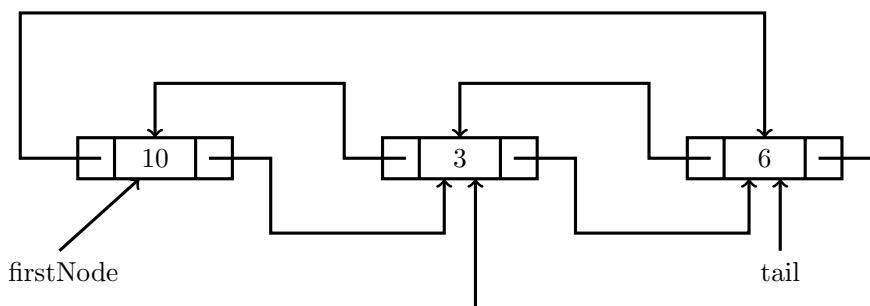


Figura 4.30. Se enlaza el campo **next** del último nodo al nodo apuntado por **firstNode→next**.

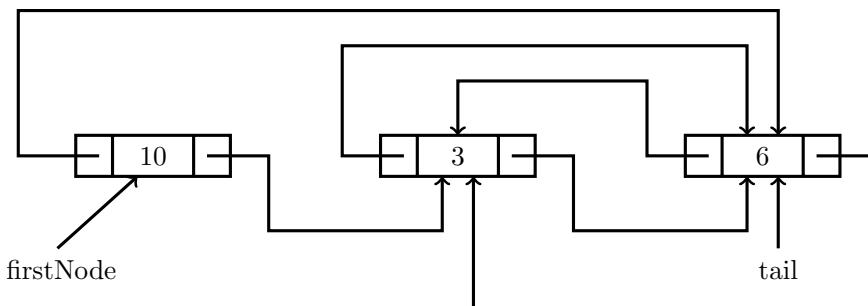


Figura 4.31. Se enlaza el campo `prev` del nodo apuntado por `tail->next` al nodo apuntado por `tail`.

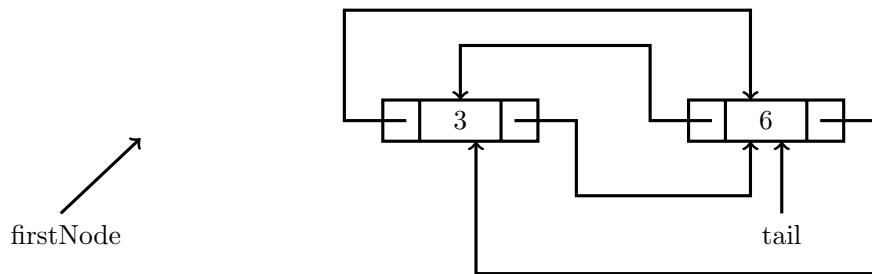


Figura 4.32. Se libera la memoria del nodo apuntado por `firstNode`.

El código en Lenguaje C para borrar el primer nodo de la lista circular doblemente enlazada es el siguiente:

```
struct node *DeleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            tail->next->prev = tail;
            free(firstNode);
        }
    }
    return tail;
}
```

#### 4.5.3. Impresión desde el primero hasta el último de los elementos de una lista circular doblemente enlazada

El código en Lenguaje C para imprimir los elementos del primero al último de una lista circular doblemente enlazada es el siguiente:

```
void PrintFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;

        while(currentNode != tail)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->next;
        }

        printf("%d ... \n", tail->key);
    }
}
```

#### 4.5.4. Impresión desde el último hasta el primero de los elementos de una lista circular doblemente enlazada

El código en Lenguaje C para imprimir los elementos del último al primero de una lista circular doblemente enlazada es el siguiente:

```
void PrintFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail;

        while(currentNode != tail->next)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->prev;
        }

        printf("%d ... \n", tail->next->key);
    }
}
```

#### 4.5.5. Programa completo para el mantenimiento de una lista circular doblemente enlazada

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>

struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *InsertElementInCircularDoublyLinkedList(struct node *tail,
                                                       int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *DeleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        if(tail == tail->next)
        {
            free(tail);
            tail = NULL;
        }
        else
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            tail->next->prev = tail;
        }
    }
}
```

```

        free(firstNode);
    }

}

return tail;
}

void PrintFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;

        while(currentNode != tail)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("%d ... \n", tail->key);
    }
}

void PrintFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail;

        while(currentNode != tail->next)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->prev;
        }
        printf("%d ... \n", tail->next->key);
    }
}

int main()
{
    struct node *tail = NULL;
    int operation, element;

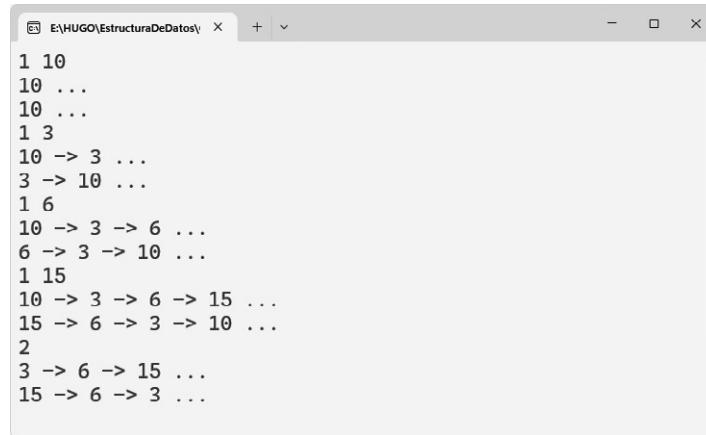
    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            scanf("%d", &element);
            tail = InsertElementInCircularDoublyLinkedList(tail, element);
            PrintFromFirstToLastCircularDoublyLinkedList(tail);
            PrintFromLastToFirstCircularDoublyLinkedList(tail);
        }
    }
}

```

```
    else
    {
        if(operation == 2) /* Delete */
        {
            tail = DeleteFirstNodeInCircularDoublyLinkedList(tail);
            PrintFromFirstToLastCircularDoublyLinkedList(tail);
            PrintFromLastToFirstCircularDoublyLinkedList(tail);
        }
        else
            printf("Bad use. \n 1. Insert\n 2. Delete\n");
    }

}

return 0;
}
```



```
E:\HUGO\EstructuraDeDatos\ > + - x
1 10
10 ...
10 ...
10 ...
1 3
10 -> 3 ...
3 -> 10 ...
1 6
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
1 15
10 -> 3 -> 6 -> 15 ...
15 -> 6 -> 3 -> 10 ...
2
3 -> 6 -> 15 ...
15 -> 6 -> 3 ...
```

Figura 4.33. Salida del programa para el mantenimiento de listas circulares doblemente enlazadas.

#### 4.5.6. Borrar un elemento en una lista circular doblemente enlazada

El código en Lenguaje C para borrar la primera ocurrencia de un elemento en una lista circular doblemente enlazada es el siguiente:

```
struct node *DeleteElementOfCircularDoublyLinkedList(struct node *tail,
                                                       int element)

{
    struct node *currentNode;
    int flag = FALSE;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        currentNode = tail->next;

        while((currentNode != tail) && (flag == FALSE))
        {
            if(currentNode->key == element)
```

```

    {
        flag = TRUE;
        break;
    }
    else
        currentNode = currentNode->next;
}

if(flag == FALSE)
{
    if(tail->key == element)
    {
        flag = TRUE;
        currentNode = tail;
    }
}

if(tail == tail->next)
{
    if(flag == TRUE)
    {
        free(tail);
        tail = NULL;
    }
    else
        printf("The %d is not in the circular doubly linked list.\n"
               " ", element);
}
else
{
    if(flag == FALSE)
        printf("The %d is not in the circular doubly linked list.\n"
               " ", element);
    else
    {
        currentNode->next->prev = currentNode->prev;
        currentNode->prev->next = currentNode->next;
        if(currentNode == tail)
            tail = tail->prev;
        free(currentNode);
    }
}
}

return tail;
}

```

#### 4.5.7. Programa completo para el mantenimiento de una lista circular doblemente enlazada en la cual se inserta o se borra un elemento específico

```

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

```

```
struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *InsertElementInCircularDoublyLinkedList(struct node *tail,
                                                    int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *DeleteElementOfCircularDoublyLinkedList(struct node *tail,
                                                    int element)
{
    struct node *currentNode;
    int flag = FALSE;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
        currentNode = tail->next;

        while((currentNode != tail) && (flag == FALSE))
        {
            if(currentNode->key == element)
            {
                flag = TRUE;
                break;
            }
            else
                currentNode = currentNode->next;
        }
    }
}
```

```

    if(flag == FALSE)
    {
        if(tail->key == element)
        {
            flag = TRUE;
            currentNode = tail;
        }
    }

    if(tail == tail->next)
    {
        if(flag == TRUE)
        {
            free(tail);
            tail = NULL;
        }
        else
            printf("The %d is not in the circular doubly linked list.\n",
                   " , element);
    }
    else
    {
        if(flag == FALSE)
            printf("The %d is not in the circular doubly linked list.\n",
                   " , element);
        else
        {
            currentNode->next->prev = currentNode->prev;
            currentNode->prev->next = currentNode->next;
            if(currentNode == tail)
                tail = tail->prev;
            free(currentNode);
        }
    }
}

return tail;
}

void PrintFromFirstToLastCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;

        while(currentNode != tail)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->next;
        }

        printf("%d ... \n", tail->key);
    }
}

```

```
void PrintFromLastToFirstCircularDoublyLinkedList(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail;

        while(currentNode != tail->next)
        {
            printf("%d -> ", currentNode->key);
            currentNode = currentNode->prev;
        }
        printf("%d ... \n", tail->next->key);
    }
}

int main()
{
    struct node *tail;
    int operation, element;

    tail = NULL;

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            tail = InsertElementInCircularDoublyLinkedList(tail, element);
            PrintFromFirstToLastCircularDoublyLinkedList(tail);
            PrintFromLastToFirstCircularDoublyLinkedList(tail);
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                tail = DeleteElementOfCircularDoublyLinkedList(tail,
                    element);
                PrintFromFirstToLastCircularDoublyLinkedList(tail);
                PrintFromLastToFirstCircularDoublyLinkedList(tail);
            }
            else
                printf("Bad use. \n 1. Insert\n 2. Delete\n");
        }
    }

    return 0;
}
```

```

1 10
10 ...
10 ...
1 3
10 -> 3 ...
3 -> 10 ...
1 6
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
1 15
10 -> 3 -> 6 -> 15 ...
15 -> 6 -> 3 -> 10 ...
2 15
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...
2 15
The 15 is not in the circular doubly linked list.
10 -> 3 -> 6 ...
6 -> 3 -> 10 ...

```

Figura 4.34. Salida del programa para el mantenimiento de listas circulares doblemente enlazadas donde se inserta y se borra un elemento específico.

#### 4.5.8. Reto de programación: Rifa de Josefo

**Nombre original:** Josephus Lottery<sup>5</sup>.

**Fuente:** UTP Open 2015.

**Fecha:** 11 de Abril de 2015.

**Autor:** Hugo Humberto Morales Peña.

Queremos hacer una rifa entre los estudiantes de este grupo de Estructura de Datos y Pepito Pérez (estudiante de este curso) sugiere que utilicemos el problema de Josefo para determinar quien es el ganador, sin embargo, ustedes y yo sabemos que se puede conocer de antemano la posición ganadora si se conocen los valores de  $n$  (el total de estudiantes en la rifa) y  $k$  (cantidad de movimientos, para ir sacando estudiantes por ronda).

El premio de la rifa es interesante: el ganador quedará exonerado del examen final, por este motivo, se propone la siguiente variante al problema de Josefo: se toma la lista de clase de los estudiantes de la materia, en la cual se encuentran numerados los estudiantes del 1 al  $n$ , se organizan estos números en círculo y se comienza a contar desde el número 1 hasta llegar al valor  $k$  (considerar que esto es en el mismo sentido de las manecillas del reloj), el estudiante con el numero  $k$  en la lista se retira del círculo y se comienza a contar desde el siguiente estudiante (en este caso sería el que está ubicado en la posición  $k + 1$ ), pero ahora se realizan los  $k$  movimientos en el círculo en el sentido contrario al de las manecillas del reloj, se retira el estudiante donde terminó el conteo y se comienza de nuevo el conteo en el estudiante que seguiría en el sentido del conteo, es decir, en el que estaría ubicado en la posición  $k + 1$ . El proceso continua así sucesivamente alternando el sentido horario y anti-horario hasta encontrar el ganador de la rifa.

<sup>5</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/josephus-lottery>

### Formato de entrada:

La entrada contiene varios casos de prueba. Cada caso está compuesto de una sola línea, que contienen dos números enteros positivos  $n$  ( $1 \leq n \leq 10^4$ ) y  $k$  ( $1 \leq k \leq n$ ) que representan respectivamente el número de estudiantes en la rifa y el valor del movimiento para descartar estudiantes del círculo. La entrada finaliza con un caso de prueba que contiene dos ceros, el cual no debe ser procesado.

Se garantiza que la suma de todos los valores de  $n$  en los casos de prueba es a lo sumo  $10^4$ .

### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea el número que representa en la lista de estudiantes del curso al estudiante ganador.

### Ejemplo de entrada:

```
1 1
10 1
10 5
10 10
5 5
5 4
0 0
```

### Ejemplo de salida:

```
1
6
2
5
4
2
```

### Solución del reto

Se plantea una solución utilizando listas circulares doblemente enlazadas, en las que se simula los movimientos en el sentido de las manecillas del reloj al moverse de nodo a nodo con el puntero `next` (siguiente). De forma similar, se simula los movimientos en contra de la manecillas del reloj (movimiento anti-horario) al moverse de nodo a nodo con el puntero `prev` (previo).

Para cada caso de prueba se insertan los elementos del 1 al  $n$  en la lista circular. De nuevo tenemos la observación clave que ya habíamos trabajado en el reto de programación del problema de Josefo, la función que elimina el primer nodo de la lista circular en esencia elimina el nodo siguiente del nodo apuntado por `tail` (en el caso específico de este reto, el siguiente nodo apuntado por `actualNode`), entonces es simplemente alternar los movimientos en sentido horario y anti-horario borrando el nodo que sigue del nodo apuntado por el `actualNode`.

El costo computacional en tiempo de ejecución de la solución es eliminar  $n - 1$  nodos de la lista circular doblemente enlazada, realizando  $k - 1$  movimientos entre los nodos para garantizar

que el nodo a eliminar es el que se encuentra en el movimiento  $k$ , alternando entre el sentido horario y anti-horario, es decir,  $(n - 1) \cdot (k - 1) = n \cdot k - n - k + 1 = O(n \cdot k) = O(k \cdot n)$ , como  $k$  está acotado superiormente por  $n$  entonces la solución en el peor de los casos es un  $O(n^2)$  por caso de prueba.

La solución en Lenguaje C utilizando listas circulares doblemente enlazadas es:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define CLOCKWISE 1
#define COUNTERCLOCKWISE 0

struct node
{
    int key;
    struct node *next;
    struct node *prev;
};

struct node *InsertElementInCircularDoublyLinkedList(struct node *tail,
                                                       int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        newNode->prev = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        newNode->next->prev = newNode;
        newNode->prev = tail;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

struct node *DeleteFirstNodeInCircularDoublyLinkedList(struct node *tail)
{
    struct node *firstNode;

    if(tail == NULL)
        printf("The circular doubly linked list is empty.\n");
    else
    {
```

```
    if(tail == tail->next)
    {
        free(tail);
        tail = NULL;
    }
    else
    {
        firstNode = tail->next;
        tail->next = firstNode->next;
        tail->next->prev = tail;
        free(firstNode);
    }
}

return tail;
}

int main()
{
    struct node *currentNode = NULL;
    int n, k, index, direction, move, element;

    while(scanf("%d %d", &n, &k) && (n > 0) && (k > 0))
    {
        direction = CLOCKWISE;
        for(index = 1; index <= n; index++)
            currentNode = InsertElementInCircularDoublyLinkedList(
                currentNode, index);

        while(currentNode != currentNode->next)
        {
            if(direction == CLOCKWISE)
            {
                for(move = 1; move < k; move++)
                    currentNode = currentNode->next;

                currentNode = DeleteFirstNodeInCircularDoublyLinkedList(
                    currentNode);
                direction = COUNTERCLOCKWISE;
            }
            else
            {
                for(move = 1; move < k; move++)
                    currentNode = currentNode->prev;

                currentNode = DeleteFirstNodeInCircularDoublyLinkedList(
                    currentNode);
                currentNode = currentNode->prev;
                direction = CLOCKWISE;
            }
        }

        printf("%d\n", currentNode->key);
        currentNode = DeleteFirstNodeInCircularDoublyLinkedList(currentNode
            );
    }

    return 0;
}
```

Salida del programa anterior ejecutando los ejemplos del reto:

```

1 1
1
10 1
6
10 5
2
10 10
5
5 5
4
5 4
2
0 0

```

Figura 4.35. Salida del programa para el reto: Rifa de Josefo.

## 4.6. Pilas y colas

Las pilas (stacks) y las colas (queues) son conjuntos dinámicos en los cuales los elementos son removidos del conjunto por la operación `delete`, que es pre-establecida para cada uno de ellos.

En la pila, el elemento borrado es el último elemento insertado. En las pilas se implementa una política LIFO (Last-In First-Out).

En la cola, el elemento borrado es siempre el que lleva más tiempo en el conjunto. En las colas se implementa una política FIFO (First-In First-Out).

## 4.7. Pilas

La operación `insert` en pilas se llama `push`, y la operación `delete` se llama `pop`.

Para la implementación de pilas se utilizará listas simplemente enlazadas donde la inserción (`push`) y borrado (`pop`) de elementos se realizará por el nodo apuntado por el puntero `top`.

### 4.7.1. Función Push

```

struct node *Push(struct node *top, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->next = top;
    top = newNode;

    return top;
}

```

#### 4.7.2. Función Pop

```
int Pop(struct node **top)
{
    struct node *currentNode;
    int element;

    currentNode = *top;
    element = currentNode->key;
    *top = currentNode->next;
    free(currentNode);

    return element;
}
```

#### 4.7.3. Función StackEmpty

```
int StackEmpty(struct node *top)
{
    return top == NULL ? TRUE : FALSE;
}
```

#### 4.7.4. Programa completo para el manejo de pilas

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
};

struct node *Push(struct node *top, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->next = top;
    top = newNode;

    return top;
}

int Pop(struct node **top)
{
    struct node *currentNode;
    int element;

    currentNode = *top;
    element = currentNode->key;
```

```

        *top = currentNode->next;
        free(currentNode);

    return element;
}

void PrintStack(struct node *top)
{
    struct node *currentNode;

    if(top == NULL)
        printf("NULL\n");
    else
    {
        currentNode = top;
        while(currentNode != NULL)
        {
            printf("%d ", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("\n");
    }
}

int StackEmpty(struct node *top)
{
    return top == NULL ? TRUE : FALSE;
}

int main()
{
    struct node *top = NULL;
    int operation, element;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Push */
        {
            scanf("%d", &element);
            top = Push(top, element);
            PrintStack(top);
        }
        else
        {
            if(operation == 2) /* Pop */
            {
                if(!StackEmpty(top))
                {
                    element = Pop(&top);
                    printf("Element: %d\n", element);
                    PrintStack(top);
                }
                else
                {
                    printf("The stack is empty.\n");
                }
            }
            else
                printf("Bad use. \n 1. Push\n 2. Pop\n");
        }
    }
}

```

```
    }
}

return 0;
```

```
1 4
4
1 3
3 4
1 2
2 3 4
1 1
1 2 3 4
2
Element: 1
2 3 4
2
Element: 2
3 4
2
Element: 3
4
2
Element: 4
NULL
```

Figura 4.36. Salida del programa para el manejo de pilas.

#### 4.7.5. Reto de programación: Consultas sobre la pila

**Nombre original:** Queries on the Stack<sup>6</sup>.

**Fuente:** ICPC Centroamérica 2020.

**Fecha:** 30 de Junio de 2021.

**Autor:** Yonny Mondelo Hernández.

Inicialmente, se tiene una pila vacía y se dan algunas consultas. Las consultas son operaciones básicas que se pueden realizar sobre la estructura de datos Pila, tales como Push (insertar un elemento en el tope de la pila), Pop (borrar el elemento del tope de la pila), y encontrar el elemento del tope de la pila. También, es necesario conocer en cualquier momento el valor absoluto de la diferencia entre el máximo y el mínimo valor que se encuentran actualmente en la pila. La tarea a realizar es procesar las consultas dadas.

##### Formato de entrada:

La primera línea contiene un número entero positivo  $T$  ( $1 \leq T \leq 100$ ) denotando el número de casos. Cada caso comienza con una línea que contiene un número entero positivo  $Q$  ( $1 \leq Q \leq 10^4$ ), denotando el número de consultas a procesar, seguidas por exactamente  $Q$  líneas basadas en el siguiente formato:

- 1  $V$ : Operación Push  $V$  ( $0 \leq V \leq 10^4$ ), inserta  $V$  en el tope de la pila.

<sup>6</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/queries-on-the-stack>

- 2: Operación Pop, borra el elemento del tope de la pila. Si la pila está vacía entonces no se hace nada.
- 3: Imprimir, en una línea, el valor absoluto de la diferencia entre el máximo y el mínimo valor que se encuentran en la pila. Si la pila está vacía, imprimir en una línea el mensaje: **Empty!**

**Formato de salida:**

Para cada consulta del tipo 3 calcular e imprimir, en una sola línea, el valor absoluto de la diferencia entre el máximo y mínimo valor que se encuentran en la pila.

**Ejemplo de entrada:**

```
2
12
2
1 10
1 15
3
2
2
3
1 18
3
1 10
1 1
3
3
3
1 999
3
```

**Ejemplo de salida:**

```
5
Empty!
0
17
Empty!
0
```

**Solución del reto**

Se plantea una solución utilizando la estructura de datos pila. Las funciones Push y Pop (insertar y borrar) en la pila tienen un costo computacional en tiempo de ejecución de  $O(1)$ . También, se puede consultar el valor absoluto de la diferencia entre los valores máximo y mínimo de la pila en un tiempo de ejecución de  $O(1)$ , donde simplemente es hacer la diferencia entre el valor máximo (`maxValue`) y el valor mínimo valor (`minValue`) del nodo que se encuentra en el tope de la pila. Obviamente, dichos campos fueron adicionados previamente a la estructura nodo

(node). En el momento de insertar un nodo en el tope de la pila (operación Push), los valores de máximo y mínimo únicamente se determinan al comparar los valores de máximo y mínimo del nodo que se encuentra en el tope de la pila con respecto al valor del elemento a insertar.

El costo computacional en tiempo de ejecución por cada caso de prueba se determina de la siguiente forma, se realizan  $Q$  operaciones sobre la pila, cada una de ellas con un costo de  $O(1)$ , por lo tanto,  $Q \cdot O(1) = O(Q)$ . Correr todos los  $T$  casos de prueba tiene un costo de  $T \cdot O(Q) = O(T \cdot Q)$ , como  $T$  esta acotado superiormente por  $100 = 10^2$  y  $Q$  está acotado superiormente por  $10^4$ , entonces la solución planteada en el peor de los casos, realiza una cantidad de operaciones del orden de  $T \cdot Q = 10^2 \cdot 10^4 = 10^6$ .

La siguiente es la solución en Lenguaje C utilizando la estructura de datos pila:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    int maxValue;
    int minValue;
    struct node *next;
};

struct node *Push(struct node *top, int element, int maxValue,
                  int minValue)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;
    newNode->maxValue = maxValue;
    newNode->minValue = minValue;
    newNode->next = top;
    top = newNode;

    return top;
}

int Pop(struct node **top)
{
    struct node *currentNode;
    int element;

    currentNode = *top;
    element = currentNode->key;
    *top = currentNode->next;
    free(currentNode);

    return element;
}
```

```

int StackEmpty(struct node *top)
{
    return top == NULL ? TRUE : FALSE;
}

int Maximum(int value1, int value2)
{
    int maximum = value1;
    if(maximum < value2)
        maximum = value2;

    return maximum;
}

int Minimum(int value1, int value2)
{
    int minimum = value1;
    if(minimum > value2)
        minimum = value2;

    return minimum;
}

struct node *DeleteStack(struct node *top)
{
    struct node *currentNode;
    currentNode = top;

    while(currentNode != NULL)
    {
        top = currentNode->next;
        free(currentNode);
        currentNode = top;
    }

    return top;
}

int main()
{
    struct node *top = NULL;
    int operation, element, totalCases, idCase;
    int totalQueries, idQuery, maxValue, minValue;

    scanf("%d", &totalCases);
    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        scanf("%d", &totalQueries);
        for(idQuery = 1; idQuery <= totalQueries; idQuery++)
        {
            scanf("%d", &operation);
            if(operation == 1) /* Push */
            {
                scanf("%d", &element);
                if(StackEmpty(top))
                {
                    maxValue = element;
                    minValue = element;
                }
            }
        }
    }
}

```

```
        else
    {
        maxValue = Maximum(element, top->maxValue);
        minValue = Minimum(element, top->minValue);
    }

    top = Push(top, element, maxValue, minValue);
}
else
{
    if(operation == 2) /* Pop */
    {
        if(!StackEmpty(top))
            element = Pop(&top);
    }
    else
    {
        if(operation == 3)
        {
            if(StackEmpty(top))
                printf("Empty!\n");
            else
                printf("%d\n", top->maxValue - top->minValue);
        }
    }
}
top = DeleteStack(top);
}

return 0;
}
```

```
2
12
2
1 10
1 15
3
5
2
2
3
Empty!
1 18
3
0
1 10
1 1
3
17
3
3
Empty!
1 999
3
0
```

Figura 4.37. Salida del programa para el reto: Consultas sobre la pila.

## 4.8. Colas

La operación `insert` en colas se llama `enqueue`, y la operación `delete` se llama `dequeue`.

Para la implementación de colas se utilizará listas simplemente enlazadas circulares, donde la inserción (`enqueue`) y borrado (`dequeue`) de elementos se realizará por el nodo apuntado por el puntero `tail`.

### 4.8.1. Función Enqueue

```
struct node *Enqueue(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}
```

### 4.8.2. Función Dequeue

```
int Dequeue(struct node **tail)
{
    struct node *firstNode;
    int element;

    if(*tail == (*tail)->next)
    {
        element = (*tail)->key;
        free(*tail);
        *tail = NULL;
    }
    else
    {
        firstNode = (*tail)->next;
        element = firstNode->key;
        (*tail)->next = firstNode->next;
        free(firstNode);
    }

    return element;
}
```

#### 4.8.3. Función QueueEmpty

```
int QueueEmpty(struct node *tail)
{
    return tail == NULL ? TRUE : FALSE;
}
```

#### 4.8.4. Programa completo para el manejo de colas

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
};

struct node *Enqueue(struct node *tail, int element)
{
    struct node *newNode;

    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

int Dequeue(struct node **tail)
{
    struct node *firstNode;
    int element;

    if(*tail == (*tail)->next)
    {
        element = (*tail)->key;
        free(*tail);
        *tail = NULL;
    }
    else
    {
        firstNode = (*tail)->next;
```

```

        element = firstNode->key;
        (*tail)->next = firstNode->next;
        free(firstNode);
    }

    return element;
}

int QueueEmpty(struct node *tail)
{
    return tail == NULL ? TRUE : FALSE;
}

void PrintQueue(struct node *tail)
{
    struct node *currentNode;

    if(tail == NULL)
        printf("NULL\n");
    else
    {
        currentNode = tail->next;
        while(currentNode != tail)
        {
            printf("%d ", currentNode->key);
            currentNode = currentNode->next;
        }
        printf("%d\n", tail->key);
    }
}

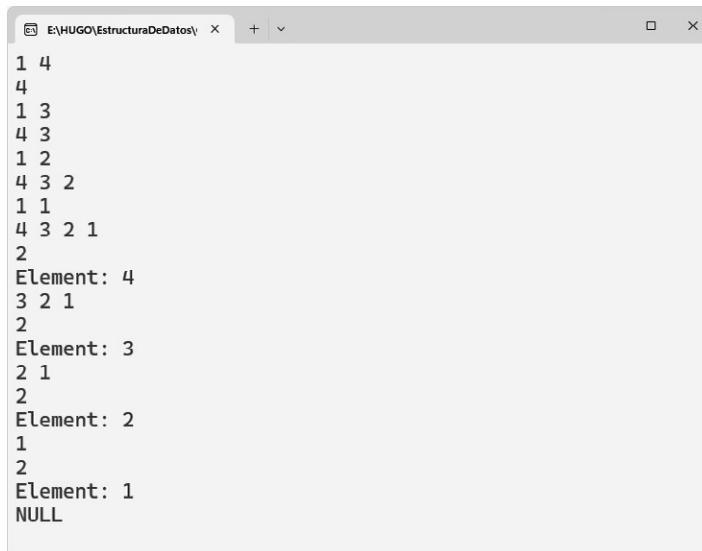
int main()
{
    struct node *tail = NULL;
    int operation, element;

    while(scanf("%d", &operation) != EOF)
    {
        if(operation == 1) /* Enqueue */
        {
            scanf("%d", &element);
            tail = Enqueue(tail, element);
            PrintQueue(tail);
        }
        else
        {
            if(operation == 2) /* Dequeue */
            {
                if(!QueueEmpty(tail))
                {
                    element = Dequeue(&tail);
                    printf("Element: %d\n", element);
                    PrintQueue(tail);
                }
                else
                {
                    printf("The queue is empty.\n");
                }
            }
        }
    }
}

```

```
        else
            printf("Bad use. \n 1. Enqueue\n 2. Dequeue\n");
    }

    return 0;
}
```



```
1 4
4
1 3
4 3
1 2
4 3 2
1 1
4 3 2 1
2
Element: 4
3 2 1
2
Element: 3
2 1
2
Element: 2
1
2
Element: 1
NULL
```

Figura 4.38. Salida del programa para el manejo de colas.

#### 4.8.5. Reto de programación: Consultas sobre la cola

**Nombre original:** Felipe y la Estructura de Datos Cola<sup>7</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Hugo Humberto Morales Peña.

Felipe inicialmente tiene una cola vacía y se le dan algunas consultas sobre ella. Las consultas son operaciones básicas que se pueden realizar sobre la estructura de datos cola, tales como **Enqueue** (insertar un elemento al final de la cola), **Dequeue** (borrar el elemento que se encuentra al frente de la cola). También, es necesario conocer en cualquier momento el valor de la suma de todos los elementos que se encuentran actualmente en la cola. La tarea a realizar es procesar las consultas dadas.

##### Formato de entrada:

La primera línea contiene un número entero positivo  $T$  ( $1 \leq T \leq 100$ ) denotando el número de casos. Cada caso comienza con una línea que contiene un número entero positivo  $Q$  ( $1 \leq Q \leq 10^4$ ), denotando el número de consultas a procesar, seguidas por exactamente  $Q$  líneas basadas en el siguiente formato:

<sup>7</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/queries-on-the-queue>

- 1  $V$ : Operación **Enqueue**  $V$  ( $0 \leq V \leq 10^4$ ), inserta  $V$  al final de la cola.
- 2: Operación **Dequeue**, borra el elemento que se encuentra al frente de la cola. Si la cola está vacía entonces no se hace nada.
- 3: Imprimir, en una línea, el valor de la suma de todos los elementos que se encuentran actualmente en la cola. Si la cola está vacía, imprimir en una línea el mensaje: **Empty!**

### **Formato de salida:**

Para cada consulta del tipo 3 calcular e imprimir en una sola línea el valor de la suma de todos los elementos que se encuentran actualmente en la cola. Si la cola está vacía, imprimir en una línea el mensaje: **Empty!**

### **Ejemplo de entrada:**

```
2
12
2
1 10
1 15
3
2
2
3
1 18
3
1 10
1 1
3
3
3
1 999
3
```

### **Ejemplo de salida:**

```
25
Empty!
18
29
Empty!
999
```

### **Solución del reto**

Se plantea una solución utilizando la estructura de datos cola. Las funciones **Enqueue** y **Dequeue** (insertar y borrar) en la cola tienen un costo computacional en tiempo de ejecución de  $O(1)$ . También se puede actualizar el resultado de la suma de los elementos que están en la cola en un tiempo de ejecución de  $O(1)$ , adicionando al resultado de la suma el elemento que se

inserta en la cola o restando al resultado de la suma el elemento que es borrado de la cola.

El costo computacional en tiempo de ejecución por cada caso de prueba se determina de la siguiente forma, se realizan  $Q$  operaciones sobre la cola, cada una de ellas con un costo de  $O(1)$ , por lo tanto,  $Q \cdot O(1) = O(Q)$ . Correr todos los  $T$  casos de prueba tiene un costo de  $T \cdot O(Q) = O(T \cdot Q)$ , como  $T$  esta acotado superiormente por  $100 = 10^2$  y  $Q$  está acotado superiormente por  $10^4$ , entonces, la solución planteada en el peor de los casos realiza una cantidad de operaciones del orden de  $T \cdot Q = 10^2 \cdot 10^4 = 10^6$ .

La siguiente es la solución en Lenguaje C utilizando la estructura de datos cola:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

struct node
{
    int key;
    struct node *next;
};

struct node *Enqueue(struct node *tail, int element)
{
    struct node *newNode;
    newNode = (struct node *) malloc(sizeof(struct node));
    newNode->key = element;

    if(tail == NULL)
    {
        newNode->next = newNode;
        tail = newNode;
    }
    else
    {
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }

    return tail;
}

int Dequeue(struct node **tail)
{
    struct node *firstNode;
    int element;

    if(*tail == (*tail)->next)
    {
        element = (*tail)->key;
        free(*tail);
        *tail = NULL;
    }
}
```

```

    else
    {
        firstNode = (*tail)->next;
        element = firstNode->key;
        (*tail)->next = firstNode->next;
        free(firstNode);
    }
    return element;
}

int QueueEmpty(struct node *tail)
{
    return tail == NULL ? TRUE : FALSE;
}

struct node *DeleteQueue(struct node *tail)
{
    struct node *firstNode;

    if(tail != NULL)
    {
        while(tail != tail->next)
        {
            firstNode = tail->next;
            tail->next = firstNode->next;
            free(firstNode);
        }
        free(tail);
        tail = NULL;
    }

    return tail;
}

int main()
{
    struct node *tail;
    int operation, element, totalCases, idCase;
    int totalQueries, idQuery;
    long long int sum;

    tail = NULL;

    scanf("%d", &totalCases);
    for(idCase = 1; idCase <= totalCases; idCase++)
    {
        sum = 0;
        scanf("%d", &totalQueries);
        for(idQuery = 1; idQuery <= totalQueries; idQuery++)
        {
            scanf("%d", &operation);
            if(operation == 1) /* Enqueue */
            {
                scanf("%d", &element);
                sum += element;
                tail = Enqueue(tail, element);
            }
            else
            {

```

```
    if(operation == 2) /* Dequeue */
    {
        if(!QueueEmpty(tail))
        {
            element = Dequeue(&tail);
            sum -= element;
        }
    }
    else
    {
        if(operation == 3)
        {
            if(QueueEmpty(tail))
                printf("Empty!\n");
            else
                printf("%lld\n", sum);
        }
    }
}
tail = DeleteQueue(tail);
}

return 0;
}
```

Salida del programa anterior ejecutando los ejemplos del reto:

```
2
12
2
1 10
1 15
3
25
2
2
3
Empty!
1 18
3
18
1 10
1 1
3
29
3
3
Empty!
1 999
3
999
```

Figura 4.39. Salida del programa para el reto: Consultas sobre la cola.

## 4.9. Retos de programación propuestos

En esta sección, se propone una lista de retos de programación que se pueden resolver con el uso de listas, pilas o colas:

- Borrar, borrar, borrar...
- Consultas sobre la lista (talla pequeña)
- Josefo y los números no-coprimos
- Rifa de Josefo II
- Juego de mesa
- Selección de personal
- Domino a ciegas y encolado
- Sumar todos II (versión 2025)

#### 4.9.1. Borrar, borrar, borrar...

**Nombre original:** Delete, delete, delete...<sup>8</sup>

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor:** Hugo Humberto Morales Peña.

El profesor Humbertov Moralov (ver Figura 4.40) ha tenido tiempo de sobra en sus últimas vacaciones; lo ha invertido en escribir retos de programación para evaluar a sus nuevos estudiantes del curso de programación.



Figura 4.40. Imagen del profesor Humbertov Moralov generada con ChatGPT en modo “Studio Ghibli”.

El profesor Moralov ha dejado un mensaje oculto en un texto (entiéndase, en una cadena de caracteres). Para poder descubrir el mensaje oculto, se tienen que eliminar del texto todas las ocurrencias de una lista de caracteres. Además, se sabe de antemano que el mensaje oculto no comienza ni finaliza con uno o más espacios en blanco y que no contiene el mismo símbolo de forma consecutiva.

##### Formato de entrada:

La entrada contiene un único caso de prueba conformado por dos líneas.

La primera línea, contiene un texto cuya longitud es de mínimo 10 y de máximo  $10^6$ , que puede contener las letras en minúscula del alfabeto inglés, los dígitos del 0 al 9 y el símbolo de espacio en blanco (ASCII 32).

La segunda línea, contiene una cadena de caracteres (cadena de símbolos) que no pertenecen al mensaje encerrada entre el corchete que abre (“[”) y el corchete que cierra (“]”), cuya longitud es de mínimo 1 y de máximo 36, la cual puede contener letras en minúscula del alfabeto inglés, dígitos del 0 al 9, o, inclusive, el símbolo del espacio en blanco (ASCII 32).

<sup>8</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/delete-delete-delete>

**Formato de salida:**

Imprimir, en una sola línea, la cadena de caracteres que representa el mensaje oculto. La línea debe finalizar con el símbolo de cambio de línea (“\n”).

**Nota:** el mensaje oculto más corto es de una sola letra o un solo dígito. Además, el mensaje oculto no contiene espacios en blanco ni al comienzo ni al final.

**Ejemplo de entrada 1:**

```
king lokif bfbbdddafbbtggffaf kong bsbtbgrgugfcgtfugrfegg bigbig kjong kiko  
[bfgijklmno]
```

**Ejemplo de salida 1:**

```
data structure
```

**Ejemplo de entrada 2:**

```
auub attb appb aoob appb aeeb annb a22b a00b a22b a55b  
[ abc]
```

**Ejemplo de salida 2:**

```
utpopen2025
```

#### 4.9.2. Consultas sobre la lista (talla pequeña)<sup>9</sup>

**Nombre original:** Consultas sobre la lista (talla pequeña)<sup>9</sup>.

**Fuente:** Curso de Estructura de Datos. Universidad Tecnológica de Pereira.

**Fecha:** 1ro de Enero de 2024.

**Autor:** Hugo Humberto Morales Peña.

Inicialmente, se tiene una lista vacía y se dan algunas consultas. Las consultas son operaciones básicas que se pueden realizar sobre la estructura de datos lista, tales como **Insert** (insertar un elemento en la lista), **Delete** (borrar la primera ocurrencia de un elemento en la lista, siempre y cuando este se encuentre en ella). También, es necesario conocer en cualquier momento el valor de la mediana de los números enteros que se encuentran en la lista. La tarea a realizar es procesar las consultas dadas.

En estadísticas, la *mediana* es el número que se encuentra en la posición de la mitad en una lista de números cuando estos se ordenan de forma ascendente, dejando la misma cantidad de números a un lado y al otro. En el caso de una lista con longitud par, entonces la mediana es el promedio de los dos valores de la mitad.

##### Formato de entrada:

La entrada contiene un único caso de prueba. La primera línea, contiene un número entero positivo  $Q$  ( $1 \leq Q \leq 10^4$ ), denotando el número de consultas a procesar, seguidas por exactamente  $Q$  líneas basadas en el siguiente formato:

- 1  $V$ : Operación **Insert**  $V$  ( $0 \leq V \leq 10^4$ ), inserta el número entero  $V$  en la lista.
- 2  $V$ : Operación **Delete**  $V$  ( $0 \leq V \leq 10^4$ ), si el número entero  $V$  está en la lista entonces borra la primera ocurrencia de este. Si la lista está vacía entonces no se hace nada.
- 3: Imprimir, en una línea, el valor de la parte entera de la mediana de los números enteros que se encuentran en la lista. Si la lista está vacía, imprimir en una línea el mensaje: **Empty!**

##### Formato de salida:

Para cada consulta del tipo 3 calcular e imprimir en una sola línea el valor de la parte entera de la mediana de los números enteros que se encuentran actualmente en la lista.

---

<sup>9</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/queries-on-the-list>

**Ejemplo de entrada:**

```
12  
3  
2 5  
1 10  
1 10  
3  
2 10  
3  
1 25  
1 20  
3  
1 25  
3
```

**Ejemplo de salida:**

```
Empty!  
10  
10  
20  
22
```

#### 4.9.3. Josefo y los números no-coprimos

**Nombre original:** Humbertov y el Josephus de los No-Coprimos<sup>10</sup>.

**Fuente:** Maratón Interna de Programación UTP 2025.

**Fecha:** 5 de Abril de 2025.

**Autor:** Gabriel Gutiérrez Tamayo.

El profesor Humbertov Moralov dicta el curso de Estructuras de Datos y, como están próximos al examen de listas enlazadas, propone exonerar a uno de los estudiantes a través de un juego en el que se podría aplicar este tema. El proceso del juego es el siguiente:

- En una bolsa hay  $n$  pedazos de papel, cada uno con un número entre 1 y  $n$ , donde cada número aparece exactamente una vez.
- Cada uno de los  $n$  estudiantes saca uno de los pedazos de papel de la bolsa; el número que tenga escrito, identificará al estudiante en el juego.
- El estudiante con el número 1 es excluido del juego.
- Los estudiantes restantes se paran formando un círculo, todos mirando hacia el centro, de modo que, para cada estudiante  $i$  ( $2 \leq i \leq n - 1$ ), tenga a mano izquierda al estudiante  $i + 1$ , y el estudiante  $n$  tenga a mano izquierda al estudiante 2.
- Se realizan varias rondas hasta que solo quede un estudiante. En cada ronda, un estudiante escribe su número sobre una hoja y la rota a mano izquierda. Cada vez que un estudiante recibe la hoja, observa el número allí, calcula el MCD (Máximo Común Divisor) entre su número y el de la hoja, y si el resultado es mayor que 1, se retira del juego; de lo contrario, sigue rotando la hoja a mano izquierda. El estudiante que inicia en la primera ronda es el número 2, y en las rondas posteriores inicia el estudiante a mano izquierda del estudiante eliminado.

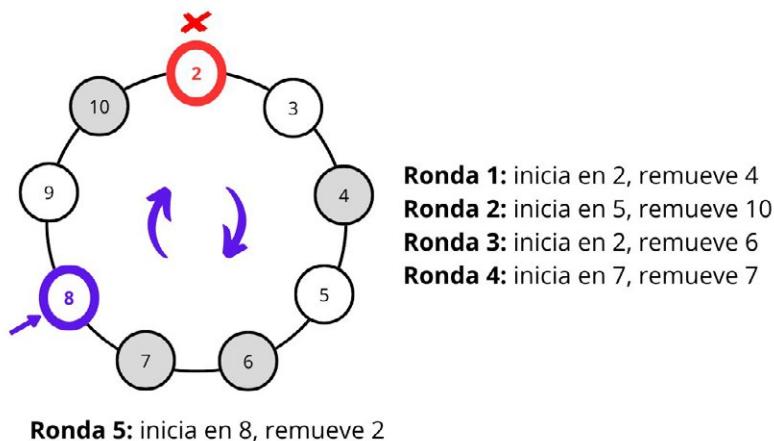


Figura 4.41. Primer caso de prueba, ronda 5.

Dado el número de estudiantes, su tarea es determinar quién gana el juego.

<sup>10</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/josephus-no-coprimos>

**Formato de entrada:**

La primera línea contiene un entero  $t$  ( $1 \leq t \leq 100$ ), indicando el número de casos de prueba. Cada caso de prueba consiste en una línea con un entero  $n$  ( $3 \leq n \leq 10^4$ ), indicando el número de estudiantes.

Se garantiza que la suma de  $n$  en todos los casos de prueba es a lo sumo  $10^4$ .

**Formato de salida:**

Para cada caso de prueba, imprima una línea con un entero, indicando el número del estudiante ganador del juego.

**Ejemplo de entrada:**

```
4
3
7
8
10
```

**Ejemplo de salida:**

```
3
3
3
8
```

#### 4.9.4. Rifa de Josefo II

**Nombre original:** Josephus Lottery II<sup>11</sup>.

**Fuente:** Maratón de Programación UFPS 2016.

**Fecha:** 4 de Junio de 2016.

**Autor:** Hugo Humberto Morales Peña.

El profesor Humbertov Moralov quiere hacer una rifa entre los 100 estudiantes de su grupo de Estructura de Datos y Pepito Pérez (estudiante de este grupo) sugiere que se utilice el problema de Josefo para determinar quien es el ganador, sin embargo, ustedes y yo sabemos que se puede saber de antemano la posición ganadora si se conoce el valor de  $n$  (el total de estudiantes en la rifa) y el valor de  $k$  (cantidad de movimientos para ir sacando estudiantes del círculo).

El premio de la rifa es interesante: el ganador quedará exonerado del examen final, por este motivo, el profesor Humbertov propone la siguiente variante al problema de Josefo: se toma la lista de clase de los estudiantes de la materia, en la cual se encuentran numerados los estudiantes del 1 al 100, se organizan estos números en círculo y se comienza a contar desde el número 1 hasta llegar al valor  $k$  (considerar que esto es en el mismo sentido de las manecillas del reloj), el estudiante con el numero  $k$  en la lista se retira del círculo y se comienza a contar desde el siguiente estudiante (en este caso sería el que está ubicado en la posición  $k + 1$ ), pero ahora se realizan los  $k$  movimientos en el círculo en el sentido contrario al de las manecillas del reloj, se retira el estudiante donde terminó el conteo y se comienza de nuevo el conteo en el estudiante que seguiría en el sentido del conteo, es decir en el que estaría ubicado en la posición  $k + 1$ . El proceso continua así sucesivamente alternando el sentido horario y anti-horario hasta encontrar el ganador de la rifa.

El profesor Humbertov quiere garantizar que el estudiante que gane la rifa sea Pepito Pérez. Por este motivo, se debe garantizar que la posición en la lista en la que se encuentra Pepito Pérez debe ser la ganadora, para esto se debe calcular el valor más pequeño de  $k$  que genere dicha posición.

##### Formato de entrada:

La entrada contiene varios casos de prueba. Cada caso está compuesto de una sola línea, que contiene un número entero positivo  $m$  ( $1 \leq m \leq 100$ ) y representa la posición en la lista del estudiante Pepito Pérez (quien debe ser el ganador de la rifa). La entrada finaliza con un caso de prueba que contiene un cero, el cual no debe ser procesado.

##### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea el número que representa el valor de  $k$  con el cual se garantiza que gane el estudiante Pepito Pérez. Se garantiza que  $1 \leq k \leq 10^3$ .

---

<sup>11</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/josephus-lottery-ii>

**Ejemplo de entrada:**

1  
15  
30  
88  
0

**Ejemplo de salida:**

115  
36  
364  
750

#### 4.9.5. Juego de mesa

**Nombre original:** Table Game<sup>12</sup>.

**Fuente:** UTP Open 2018.

**Fecha:** 12 de Mayo de 2018.

**Autor:** Hugo Humberto Morales Peña.

Tenemos el siguiente juego: en una mesa redonda se encuentran  $N$  personas. Inicialmente, todos los  $N$  jugadores comienzan con la misma cantidad  $D$  de dinero. Uno de los jugadores es seleccionado como el punto de partida (posición uno), el jugador que sigue después de él en el sentido de las manecillas del reloj es el jugador dos, y así sucesivamente hasta llegar al último jugador (jugador de la posición  $N$ ). Tener en cuenta que el jugador que sigue después del último jugador es el primer jugador. En cada turno del juego es lanzada una moneda, un dado rojo y un dado azul. La moneda indica la dirección del movimiento, “cara” para el sentido horario y “sello” para el sentido anti-horario, el valor del dado rojo ( $R$ ) indica el total de movimientos y el dado azul ( $A$ ) indica el total de dinero que le es dado al jugador de la posición alcanzada por parte del “banco”. Ahora, si el dinero acumulado por el jugador alcanzado es par, entonces los jugadores que están sentados inmediatamente a su derecha e izquierda son eliminados del juego y pierden todo el dinero que tienen; el dinero se va al “banco” o “pozo común” de la mesa, en el caso contrario en el cual el dinero acumulado por el jugador alcanzado sea impar, entonces dicho jugador es “salvado” del juego, lo cual consiste en ser sacado del juego conservando el dinero que tiene acumulado hasta ese momento, es decir, el jugador es eliminado del juego pero sin perder su dinero. El jugador que queda con el turno es el que sigue inmediatamente después, en el sentido del movimiento con el cual se salvó el jugador.

El juego finaliza cuando queda un solo jugador en la mesa, el cual también es salvado.

Usted debe calcular las posiciones (en orden ascendente) de los jugadores salvados y la cantidad de dinero que ellos tienen en el momento de abandonar el juego.

##### Formato de entrada:

La entrada contiene varios casos de prueba. Cada caso comienza con una línea que tiene dos valores  $N$  ( $2 \leq N \leq 10^4$ ) y  $D$  ( $1 \leq D \leq 10^3$ ), que representan el total de jugadores y dinero inicial con el que comienzan cada uno ellos. Despues, son presentadas  $N - 1$  líneas, que representan los  $N - 1$  turnos que como máximo se pueden realizar; cada línea contiene tres valores,  $M$  ( $M \in \{C, S\}$ ),  $R$  y  $A$  ( $1 \leq R, A \leq 10^3$ ), que representan respectivamente, el valor de la moneda (cara ('C') o sello ('S')), el valor del dado rojo y el valor del dado azul.

Se garantiza que la suma de todos los valores de  $N$  en los casos de prueba es a lo sumo  $10^4$ .

La entrada finaliza con el símbolo de fin de archivo (EOF - End Of File).

##### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir tantas líneas como posiciones de jugadores salvados en el juego. Cada línea debe contener dos valores, la posición del

<sup>12</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/table-game-3>

jugador y la cantidad de dinero con el cual fue retirado del juego.

**Ejemplo de entrada:**

5 3  
C 1 2  
S 2 3  
C 3 2  
C 2 5  
10 2  
S 5 3  
C 3 2  
C 4 2  
S 5 4  
S 6 5  
S 5 3  
C 3 2  
S 6 5  
C 4 2

**Ejemplo de salida:**

2 5  
3 5  
5 6  
1 2  
4 13  
6 5  
9 7

#### 4.9.6. Selección de personal

**Nombre original:** Hiring Candidates Game<sup>13</sup>.

**Fuente:** Gran Premio de México 2024 - Segunda Fecha.

**Fecha:** 8 de Junio de 2024.

**Autor:** Hugo Humberto Morales Peña.

Ha habido un proceso de selección de personal en su empresa. Según los criterios de evaluación, hay un empate entre  $n$  candidatos. Para la fase final del proceso de selección, el coordinador de recursos humanos ha diseñado un juego para posiblemente filtrar a algunos de ellos, ya que quieren reducir los costos de personal.

El juego es el siguiente:

- 1.) Los  $n$  candidatos forman un círculo, todos ellos mirando al centro de este. Entonces, a algún candidato le es asignada la posición 1, al que se encuentra a su izquierda le es asignada la posición 2 y así sucesivamente hasta llegar a la posición  $n$ . Gracias a que es un círculo, el candidato de la posición 1 se encuentra a la izquierda del candidato de la posición  $n$ .
- 2.) Luego, dos supervisores  $s_1$  y  $s_2$  se ubican por fuera del círculo, detrás de los candidatos de las posiciones 1 y  $n$  respectivamente.
- 3.) Si el número de candidatos que quedan en el círculo es a lo sumo 2, entonces finaliza el juego contratando estos candidatos.
- 4.) De otro modo, el supervisor  $s_1$  se mueve contando  $r$  candidatos (incluyendo su propia posición) en el sentido de las manecillas del reloj y se detiene, luego el supervisor  $s_2$  se mueve contando  $c$  candidatos (incluyendo su propia posición) en el sentido contrario al de las manecillas del reloj y se detiene. Entonces, hay dos posibilidades:
  - a) Ambos supervisores  $s_1$  y  $s_2$  alcanzan (llegan) al mismo candidato. En este caso, se retira al candidato del círculo y es contratado.
  - b)  $s_1$  y  $s_2$  alcanzan candidatos diferentes. En este caso, se retiran ambos candidatos del círculo y ninguno de ellos es contratado.
- 5.) Antes de volver al paso 3, los supervisores  $s_1$  y  $s_2$  son ubicados (según el movimiento que llevan) en el siguiente candidato (de los que aún siguen en el círculo) del que acaba de ser retirado.

Dado el escenario del juego, calcular los candidatos que deben ser contratados de acuerdo al juego.

#### Formato de entrada:

La entrada contiene múltiples casos de prueba. Cada caso de prueba está compuesto de una sola línea, que contienen tres números enteros positivos  $n$  ( $1 \leq n \leq 10^4$ ),  $r$  y  $c$  ( $1 \leq r, c \leq 10^5$ ) que representan respectivamente el número de personas, los movimientos del supervisor uno y los movimientos del supervisor dos para escoger personas.

---

<sup>13</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/hiring-candidates-game>

Se garantiza que la suma de todos los valores de  $n$  en los casos de prueba es a lo sumo  $10^4$ .

La entrada finaliza con el símbolo de fin de archivo (EOF - End Of File).

### Formato de salida:

Para cada caso de prueba de la entrada, su programa debe imprimir en una sola línea ordenados de forma ascendente las posiciones en el círculo que representan a las personas seleccionadas (personas que fueron escogidas en el proceso de selección para trabajar con la empresa).

### Ejemplo de entrada:

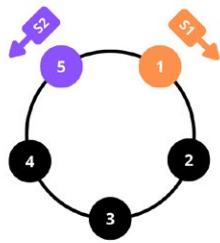
```
5 3 3
4 4 3
6 5 2
```

### Ejemplo de salida:

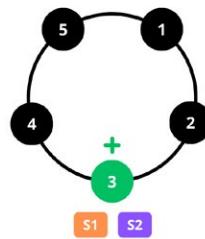
```
2 3 4
1 3
1 2 5 6
```

### Explicación del primer caso de prueba:

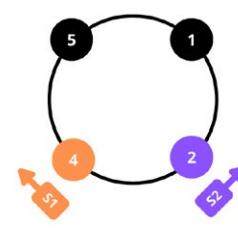
El siguiente es el paso a paso del proceso de selección hasta que este termina:



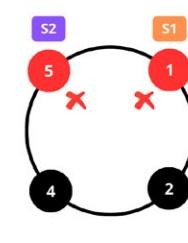
**Iteración 1:** Posición inicial



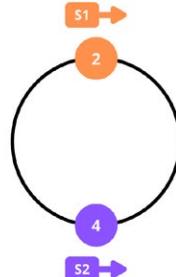
**Iteración 1:** Resultado



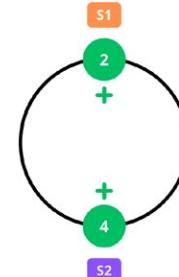
**Iteración 2:** Posición inicial



**Iteración 2:** Resultado



**Iteración 3:** Posición inicial



**Iteración 3:** Resultado

Figura 4.42. Paso a paso primer caso de prueba.

El candidato con el identificador 3 es seleccionado en la primera iteración del proceso, luego los candidatos con los identificadores 1 y 5 son removidos y rechazados en la segunda iteración del proceso. El proceso de selección termina en la tercera iteración porque únicamente quedan dos candidatos (con identificadores 2 y 4), los cuales son seleccionados. Al final, los candidatos con los identificadores 2, 3 y 4 son seleccionados en el proceso.

#### 4.9.7. Domino a ciegas y encolado

**Nombre original:** Blind Queue Dominoes<sup>14</sup>.

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor:** Gabriel Gutiérrez Tamayo.

El dominó es un juego de mesa jugado por turnos que consta de 28 fichas diferentes, donde cada ficha está identificada por dos enteros entre 0 y 6. En esta versión del juego participan dos jugadores: sus fichas permanecerán boca abajo; solo podrán ver una ficha en cada turno. Antes de iniciar una partida, cada jugador selecciona 7 fichas y las organiza en una fila de izquierda a derecha. En cada turno, un jugador voltear su ficha más a la izquierda, para poder ver su valor y, si alguno de sus dos valores coincide con alguno de los dos extremos de la secuencia de juego, puede juntarla al respectivo extremo por el lado de coincidencia; de lo contrario, coloca su ficha nuevamente al inicio, volteada boca abajo, pero esta vez de derecha a izquierda. Despues de que un jugador mueve su ficha a uno de los extremos o decide pasar, sigue el turno del otro jugador.

Una partida finaliza cuando un jugador se queda sin fichas o si desde el último turno en el que se colocó una ficha, todas las fichas restantes fueron volteadas al menos una vez. Cuando la partida finaliza, gana el jugador que se quedó sin fichas. Si ambos jugadores quedaron con fichas, gana quien tenga el menor puntaje.

El *puntaje* de un jugador al final de una partida es la suma de los puntos de las fichas que no usó.

Cada jugador lleva la cuenta de cuántas diferentes fichas ha visto un número específico, contando dos veces para fichas dobles: a esto se le llama el contador del número. Cada número entero del 0 al 6 tiene un contador. Ambos jugadores son muy cuidadosos de no mostrar su ficha al otro jugador al momento de voltearla, por lo cual, cada jugador lleva la cuenta con base en las fichas que él ha volteado o que ya han sido colocadas.

Alice y Bob quieren jugar unas partidas, y ellos siempre usan la siguiente estrategia cada vez que es su turno:

- Si ambos extremos son iguales y alguno de los valores de su ficha coincide con los extremos, junta la ficha a cualquiera de los dos extremos.
- Si los extremos son diferentes y la ficha coincide con uno solo, junta la ficha a ese extremo.
- Si los extremos son diferentes y la ficha coincide con ambos, la ficha se junta al extremo con el contador más bajo. En caso de empate, se junta al extremo de menor valor.

Dadas las fichas que tiene cada jugador, determine quién gana la partida. Alice siempre tiene el primer turno y siempre colocará su primera ficha en ese turno.

---

<sup>14</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/blind-queue-dominoes>

**Formato de entrada:**

La primera línea, contiene un entero  $t$  ( $1 \leq t \leq 10^4$ ), indicando el número de casos de prueba. Cada caso de prueba consta de dos líneas, cada una conteniendo 7 pares de enteros  $a_i, b_i$  ( $0 \leq a_i, b_i \leq 6$ ), describiendo en la primera las fichas de Alice y, en la segunda línea, las fichas de Bob. Se garantiza que, para cada caso de prueba, las 14 fichas son distintas entre sí.

**Formato de salida:**

Para cada caso de prueba, imprima una línea con el formato  $W T A B$ , donde  $W$  es una cadena indicando quién ganó la partida, “Alice”, “Bob” o “Draw”; en caso de empate,  $T$  es el número de turnos que duró la partida, y  $A$  y  $B$  son los puntajes finales de Alice y Bob, respectivamente.

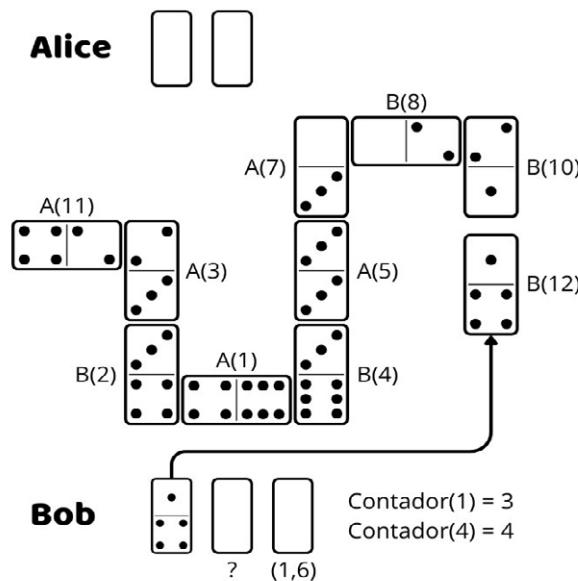


Figura 4.43. Primer caso de prueba, turno 12.

**Ejemplo de entrada:**

```

3
4 6 2 3 3 3 0 3 5 3 4 2 6 2
4 3 3 6 1 6 0 2 2 1 1 4 1 1
0 3 1 5 5 4 1 6 4 2 5 0 1 3
3 2 0 1 4 3 4 4 5 3 1 4 2 0
3 5 5 6 3 3 4 1 6 2 2 2 4 4
3 2 6 3 3 4 1 5 6 1 4 2 6 0

```

**Ejemplo de salida:**

```

Bob 16 16 9
Alice 26 6 16
Draw 14 12 12

```

#### 4.9.8. Sumar todos II (versión 2025)

**Nombre original:** Furthermore ... Add All (version 2025)<sup>15</sup>.

**Fuente:** UTP Open 2025.

**Fecha:** 17 de Mayo de 2025.

**Autor(es):** Hugo Humberto Morales Peña, Gabriel Gutiérrez Tamayo.

El profesor Humbertov Moralov, al organizar los papeles de su oficina, se encontró con la siguiente solución que planteó aproximadamente 10 años atrás para el reto de programación:  
**UVa - 10954 - Add All**<sup>16</sup>.

```
#include <stdio.h>
#define MAXT 5000

int ExtractMin(int A[], int *n)
{
    int temp, i, min;
    for(i = *n - 1; i >= 1; i--)
    {
        if(A[*n] > A[i])
        {
            temp = A[i];
            A[i] = A[*n];
            A[*n] = temp;
        }
    }
    min = A[*n];
    *n = *n - 1;
    return min;
}

void Insert(int A[], int *n, int element)
{
    *n = *n + 1;
    A[*n] = element;
}

int AddAll(int A[], int n)
{
    int result = 0, value1, value2, value3;
    while(n >= 2)
    {
        value1 = ExtractMin(A, &n);
        value2 = ExtractMin(A, &n);
        value3 = value1 + value2;
        result = result + value3;
        Insert(A, &n, value3);
    }
    return result;
}
```

---

<sup>15</sup><https://www.hackerrank.com/contests/data-structure-utp/challenges/furthermore-add-all-version-2025>

<sup>16</sup>[https://onlinejudge.org/index.php?option=com\\_onlinejudge&Itemid=8&category=21&page=show\\_problem&problem=1895](https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=21&page=show_problem&problem=1895)

```
int main()
{
    int n, A[MAXT + 1], index;
    while (scanf("%d", &n) && (n > 0))
    {
        for (index = 1; index <= n; index++)
            scanf("%d", &A[index]);
        printf("%d\n", AddAll(A, n));
    }
    return 0;
}
```

En la solución anterior se trabaja con un arreglo  $A[ ]$ , en cual se almacenan inicialmente  $n$  elementos. En la función `AddAll` en el ciclo de repetición `while`, se extraen (y eliminan) del arreglo los dos valores más pequeños, se suman, y dicho valor se inserta (se almacena) en el arreglo; ese valor también se utiliza para actualizar la suma total en la variable `result`, donde se almacena el resultado de sumar todos los pares de valores más pequeños que hay en el arreglo mientras este contenga 2 o más valores. El costo computacional de la función `AddAll` es  $O(n^2)$ , donde  $n$  es la cantidad de elementos almacenados en el arreglo.

El ciclo `while` se ejecuta  $n - 1$  veces, porque por cada iteración el arreglo tienen un número menos almacenado. El costo de cada iteración es un  $O(n)$ , porque se tienen que recorrer los  $n$  elementos almacenados en el arreglo  $A[ ]$  para determinar el valor mínimo, es decir,  $(n - 1) \cdot O(n) = O(n^2)$ .

Un  $O(n^2)$  es una solución muy costosa en tiempo de ejecución y el único motivo en estos momentos por el cual el juez en línea UVa da un veredicto de `accepted` es porque es un reto de programación muy viejo (año 2005) y en ese entonces una talla de  $n = 5,000$  era suficiente para un tiempo de 1 o 2 segundos, con el poder de computo de los servidores de esa época.

Actualmente (año 2026) una solución del orden de  $10^7$  operaciones corre en menos de un segundo en los jueces en línea, por este motivo, para volver aun más interesante el reto, se le pide a usted como estudiante de un programa académico de ciencias de la computación que planteé una solución que en el peor de los casos corra en un tiempo de  $O(n)$  por caso de prueba, para lo cual se tiene que programar de forma eficientemente la función `AddAll` para la nueva talla de la variable  $n$  que ahora es  $10^7$ .

#### Formato de entrada:

La entrada comienza con un entero positivo  $t$  ( $1 \leq t \leq 5$ ), denotando el número de casos de prueba. Cada caso de prueba está conformado por dos líneas. La primera línea, consta de un número entero positivo  $n$  ( $2 \leq n \leq 10^7$ ), que representa la cantidad de elementos para almacenar en el arreglo  $A[ ]$ . La segunda línea, consta de  $n$  números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a  $10^5$ ).

Se garantiza que la suma de todos los valores de  $n$  en los  $t$  casos de prueba es a lo sumo  $2 \cdot 10^7$ .

#### Formato de salida:

Para cada caso de prueba imprimir una sola línea con el resultado generado por la función `AddAll`.

**Ejemplo de entrada:**

```
5
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
5
3 2 5 1 4
5
1 1 1 1 1
```

**Ejemplo de salida:**

```
9
19
33
33
12
```



## Apéndice A

# Cómo evitar el error 404 de HackerRank

Possiblemente, más del 90 % de los lectores de este libro se encontrarán con el error 404 de HackerRank y creerán que los retos de programación del libro no se pueden juzgar allí.

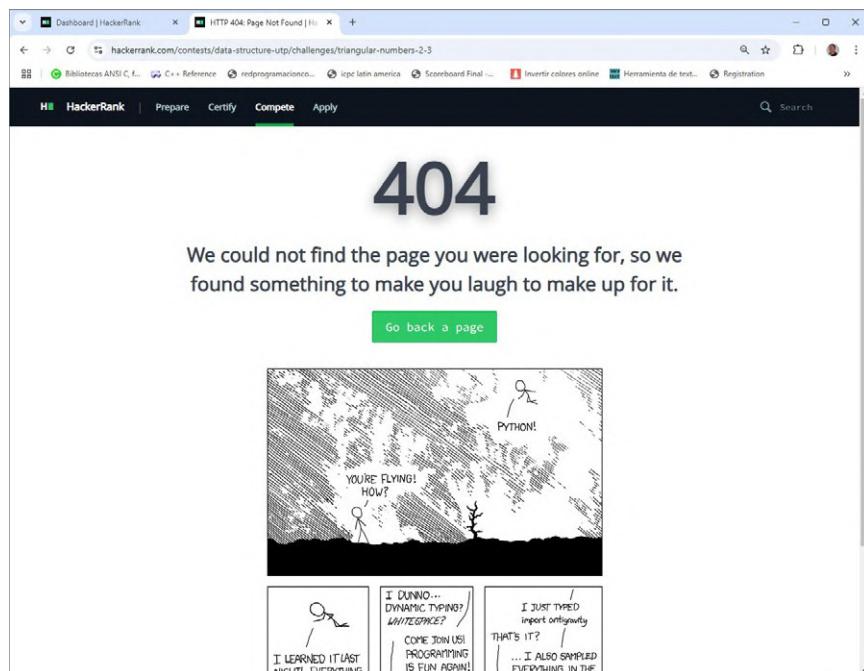


Figura A.1. Página del error 404 de HackerRank.

Para poder juzgar cualquiera de los retos de programación que se encuentran en el libro, el lector debe crear una cuenta de usuario “desarrollador” en el juez en línea HackerRank y registrarse en la competencia donde están almacenados todos los retos del libro de Estructuras de datos. Para lo cual, debe realizar el paso a paso del procedimiento que se detalla a continuación. Dicho procedimiento se realiza una sola vez.

**Paso 1:** Ingresar al juez en línea HackerRank (<https://www.hackerrank.com/>).

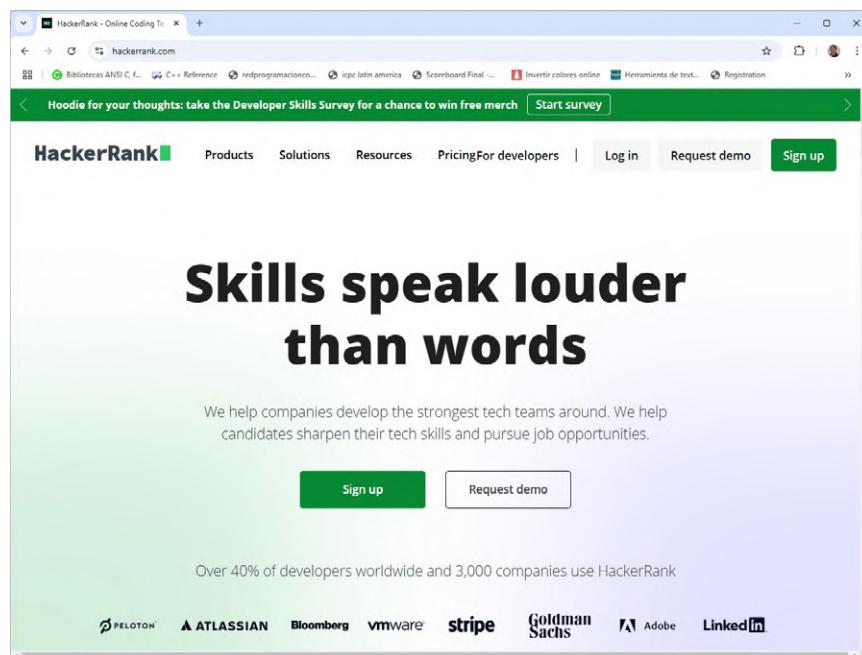


Figura A.2. Página de inicio del juez en línea HackerRank.

**Paso 2:** Al hacer clic en el botón “Sign up” debe quedar en la siguiente ventana.

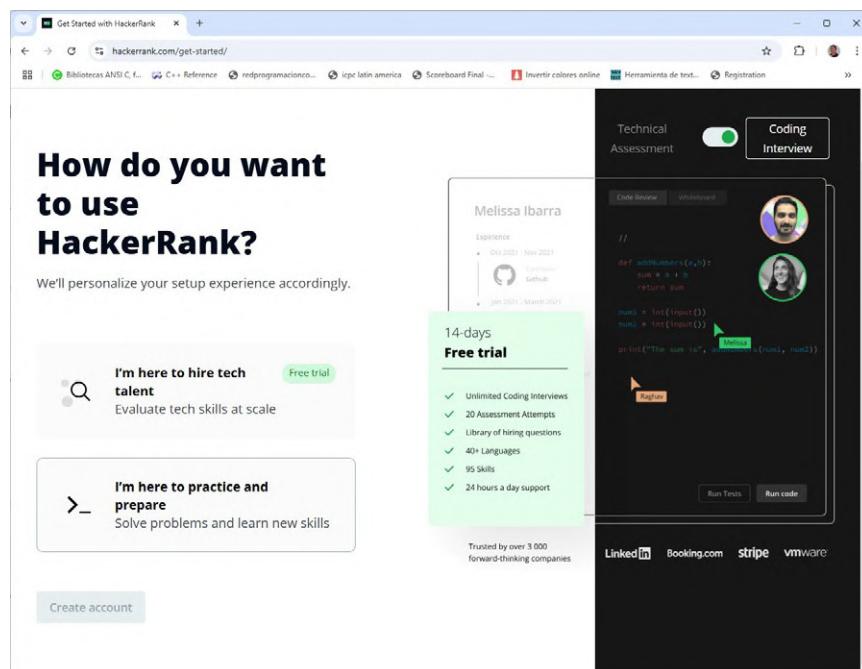


Figura A.3. Página de ¿cómo quiere usar HackerRank?

**Paso 3:** Al hacer clic en el botón de la opción “I’m here to practice and prepare” en la ventana se activa el botón de “Create account”.

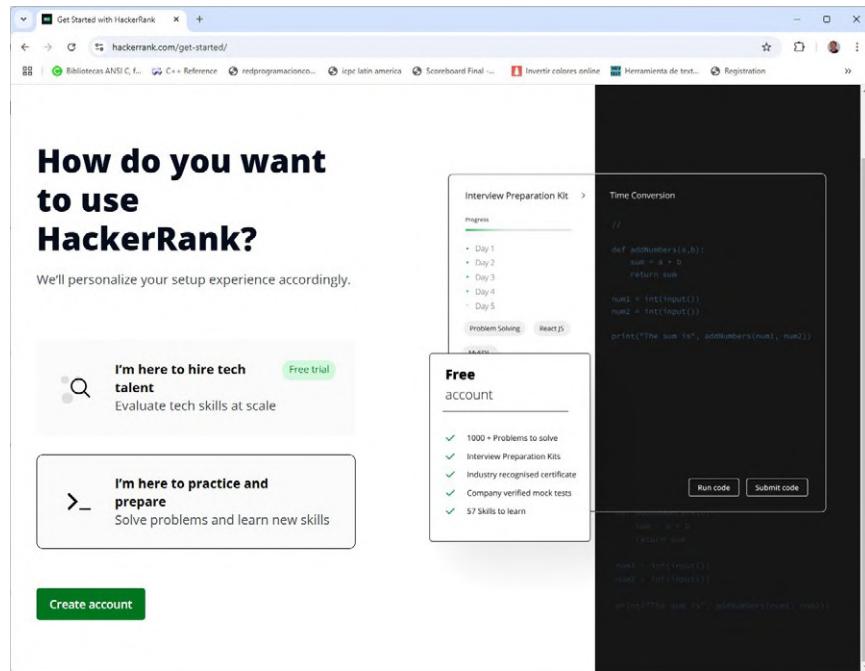


Figura A.4. Página de ¿cómo quiere usar HackerRank?

**Paso 4:** Al hacer clic en el botón “Create account” se activa la ventana para diligenciar la información de la cuenta a crear.

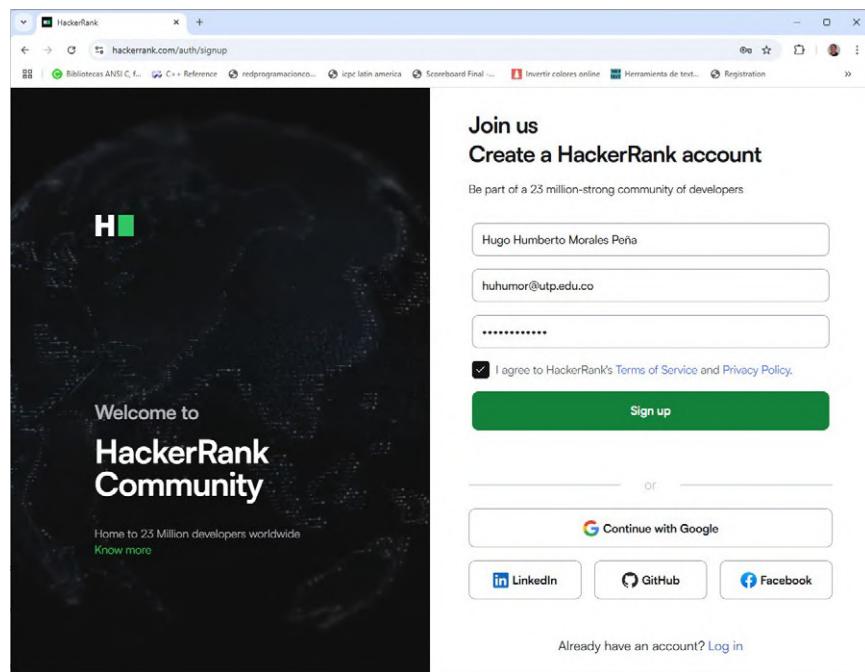


Figura A.5. Página para diligenciar la información de la cuenta.

**Paso 5:** Al diligenciar la información y hacer clic en el botón “Sign up” se crea la cuenta y se activa la ventana de bienvenida.

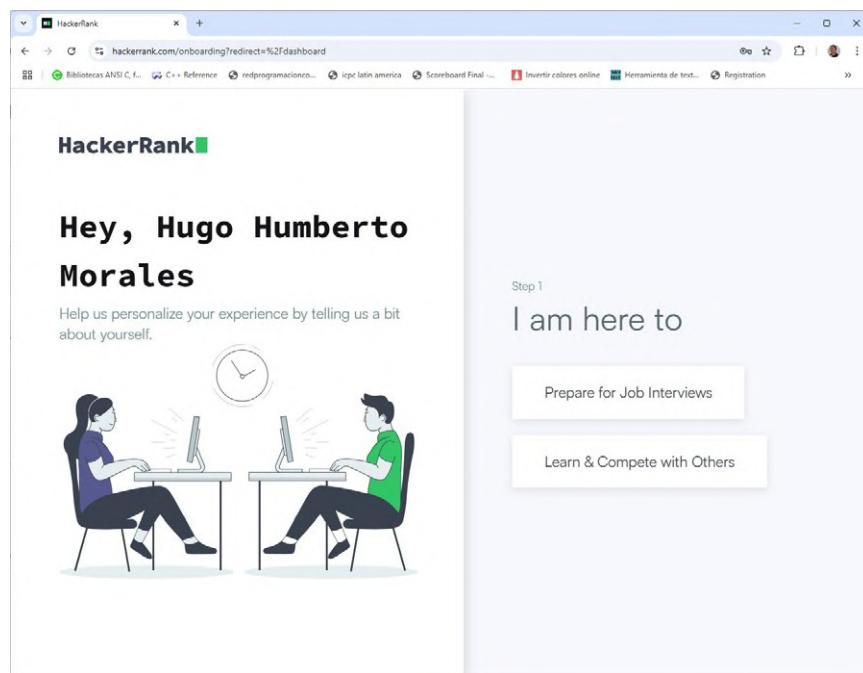


Figura A.6. Página de bienvenida.

**Paso 6:** Al hacer clic en el botón de la opción “Learn & Compete with others” se activa la ventana donde se pregunta el tipo del usuario.

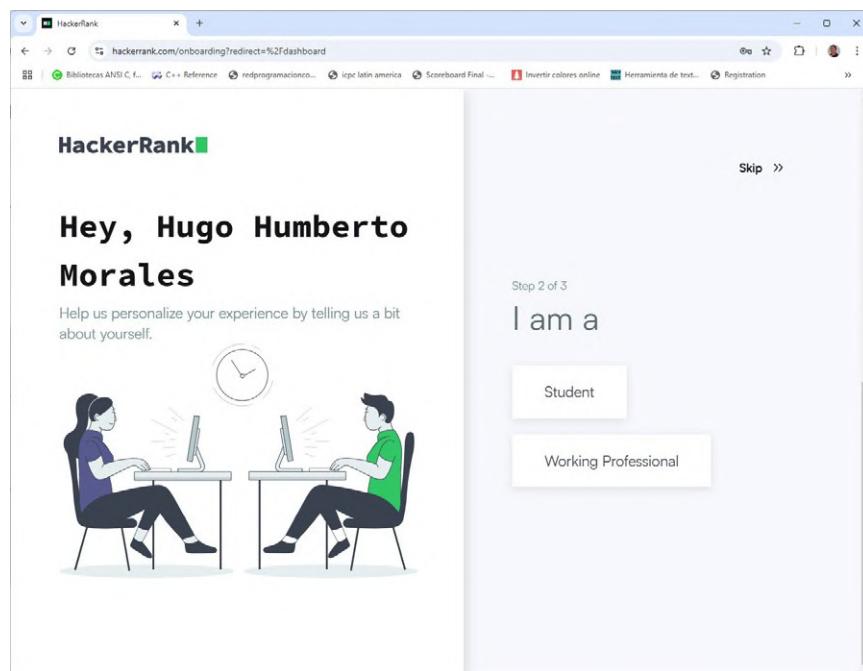


Figura A.7. Página de tipo de usuario.

**Paso 7:** Al hacer clic en el botón de la opción “Student” se activa la ventana con la pregunta de la proyección del año de graduación.

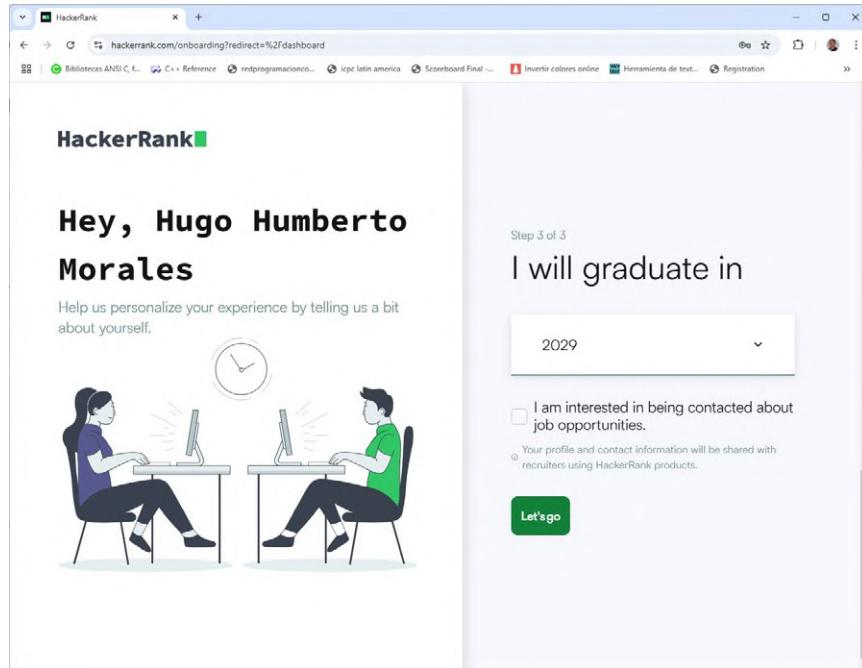


Figura A.8. Página de la proyección del año de graduación.

**Paso 8:** Al hacer clic en el botón “Let’s go” se activa la ventana de la confirmación de la cuenta por correo electrónico.

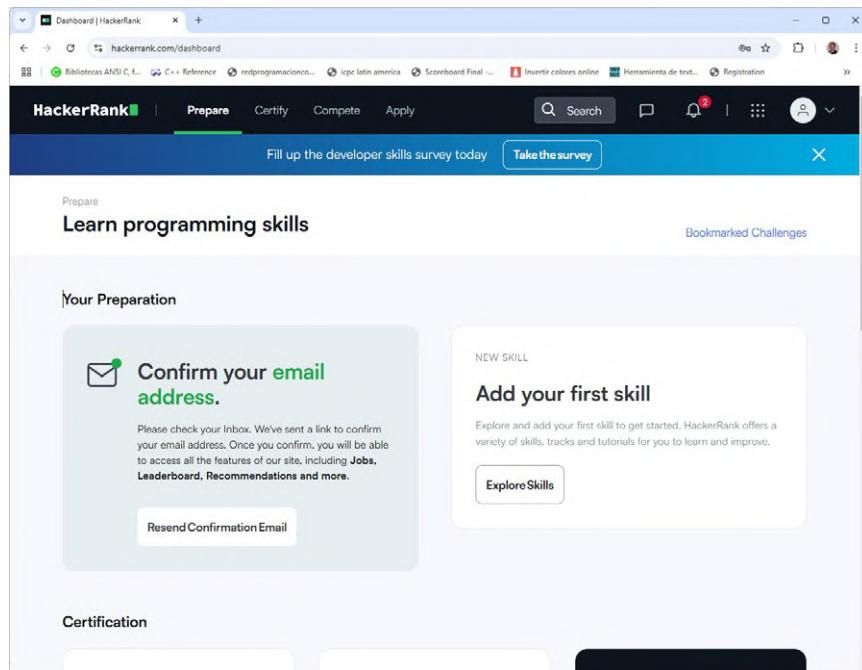


Figura A.9. Página de confirmación de la cuenta por correo electrónico.

**Paso 9:** Al confirmar la creación de la cuenta por el correo, se activa la siguiente ventana.

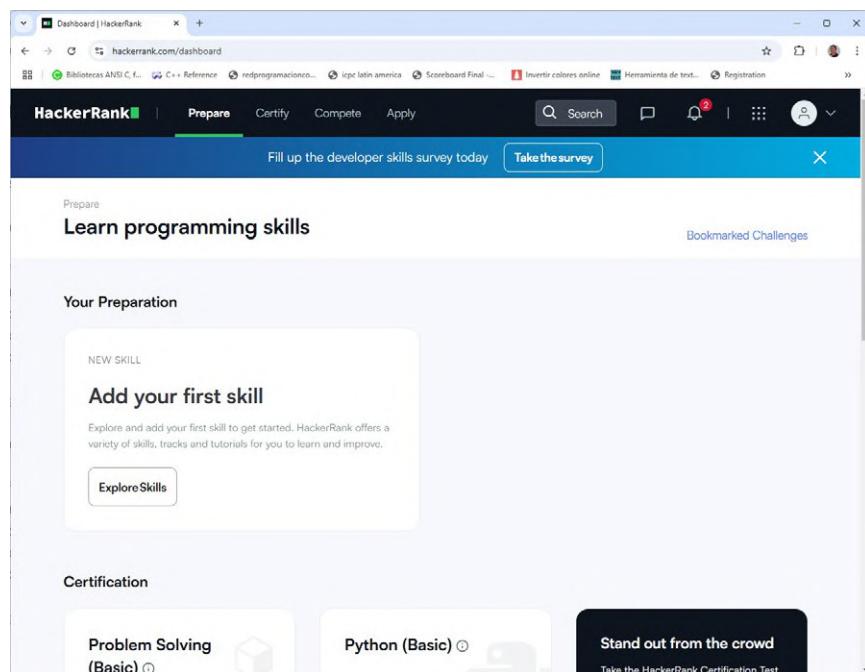


Figura A.10. Página con la cuenta de HackerRank confirmada.

**Paso 10:** Cerrar por primera vez la sesión en HackerRank al hacer clic en la opción “Logout” en el menú desplegable.

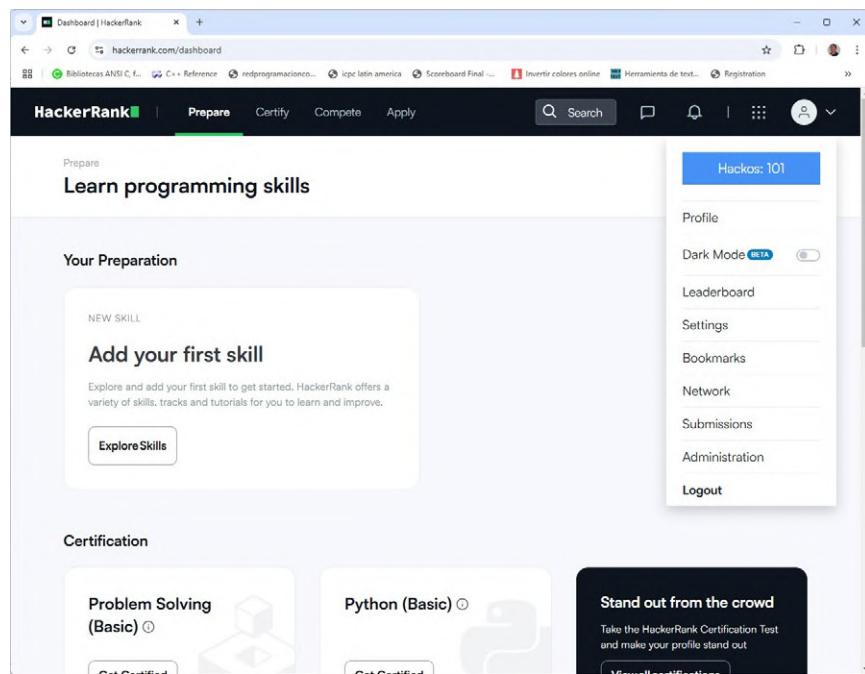


Figura A.11. Página con la opción de cerrar sesión.

**Paso 11:** Volver de nuevo a la pagina de inicio en HackerRank (<https://www.hackerrank.com/>) y hacer clic en el botón “Log in”.

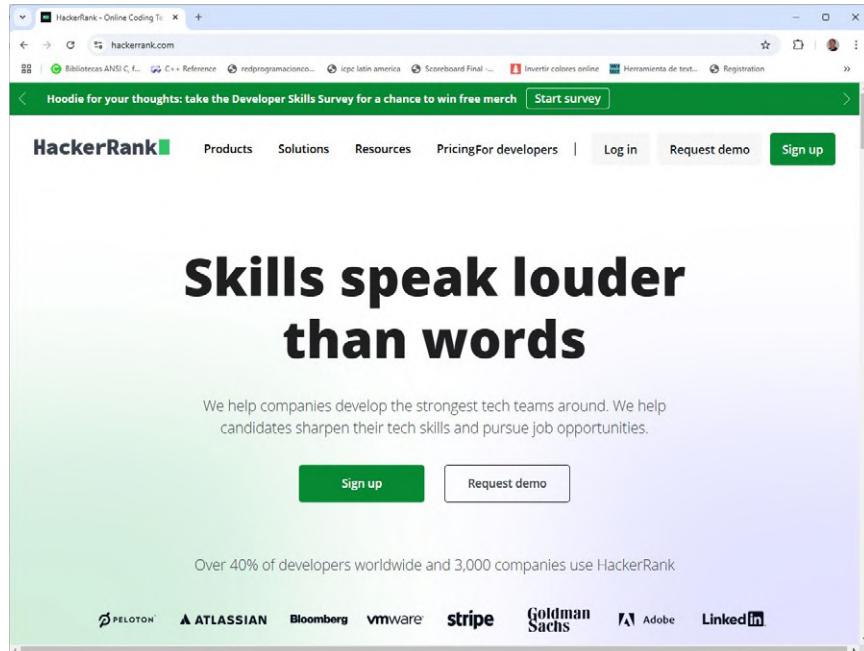


Figura A.12. Página de inicio del juez en línea HackerRank.

**Paso 12:** Al hacer clic en el botón “Log in” se activa la ventana de ingreso en la cual se debe escoger la opción “For Developers” (para desarrolladores).

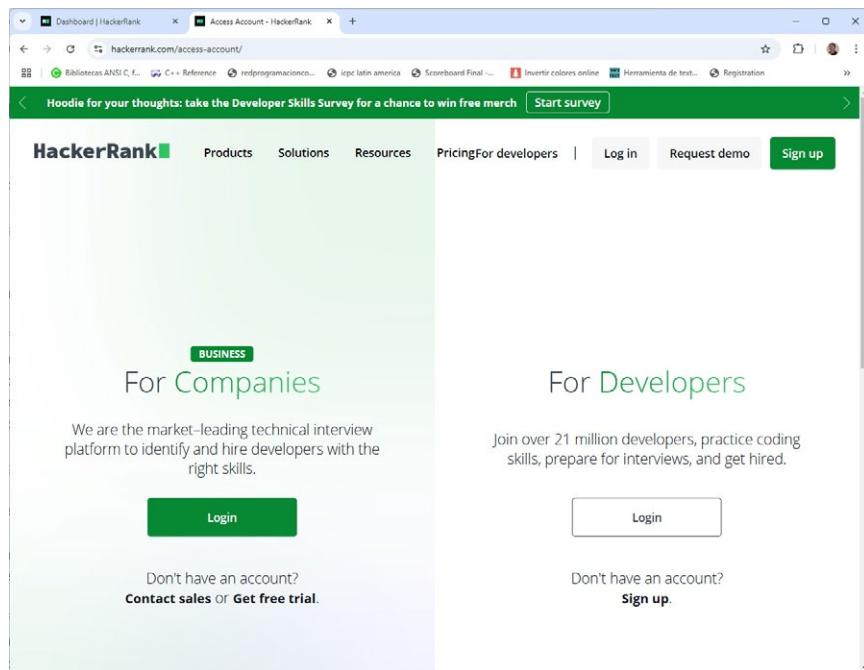


Figura A.13. Página de ingreso con el perfil de desarrollador.

**Paso 13:** Al hacer clic en el botón “Login” del perfil de “For Developers” se activa la ventana de ingreso donde se debe diligenciar el usuario y la contraseña de la cuenta, luego hacer clic en el botón “Log in”.

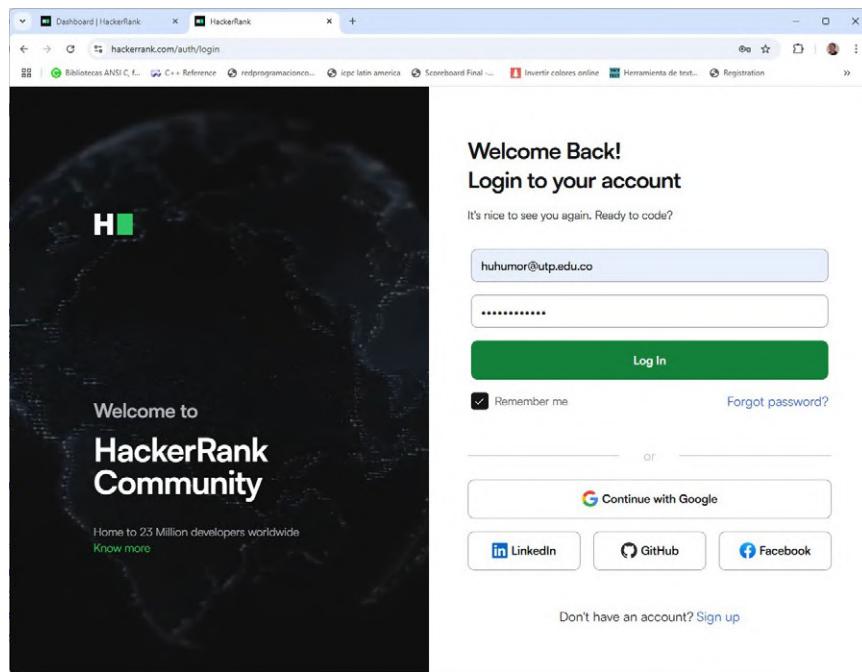


Figura A.14. Página de ingreso con usuario y contraseña.

**Paso 14:** De nuevo se esta en la página “Dashboard” (panel) de la cuenta.

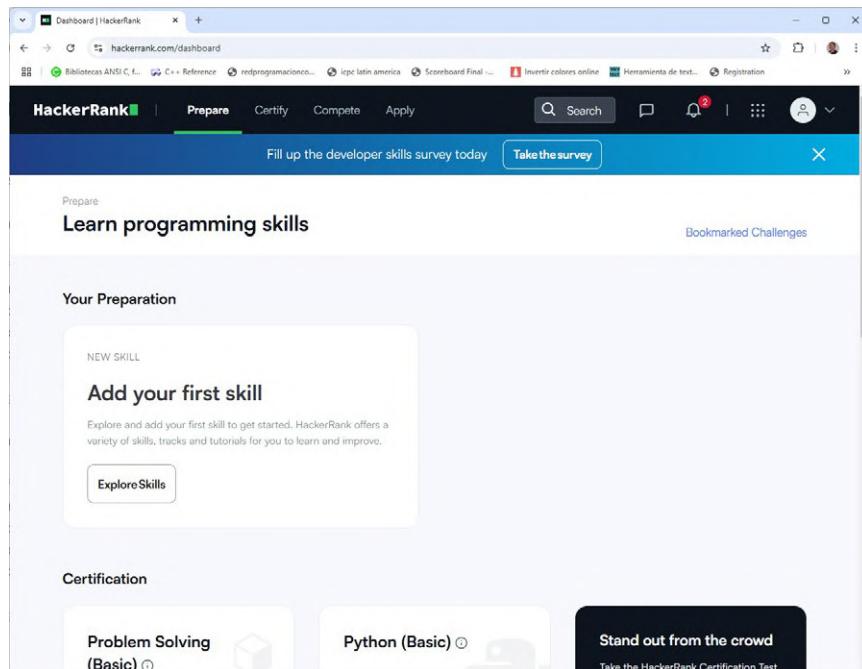


Figura A.15. Página del panel de la cuenta.

**Paso 15:** Ahora la cuenta del usuario se debe registrar en la competencia “Data Structure - UTP”, para lo cual se debe hacer clic en la dirección <https://www.hackerrank.com/data-structure-utp>.

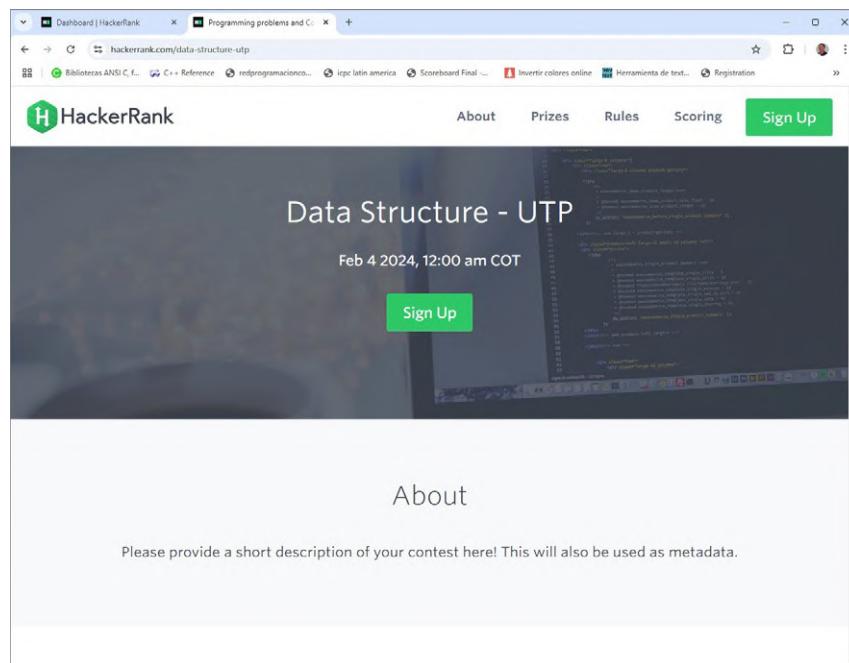


Figura A.16. Página de registro del curso de Estructuras de Datos.

**Paso 16:** Al hacer clic sobre el botón “Sign Up” se activa una ventana donde se puede navegar sobre el listado de todos los retos de programación que se encuentran en el libro.

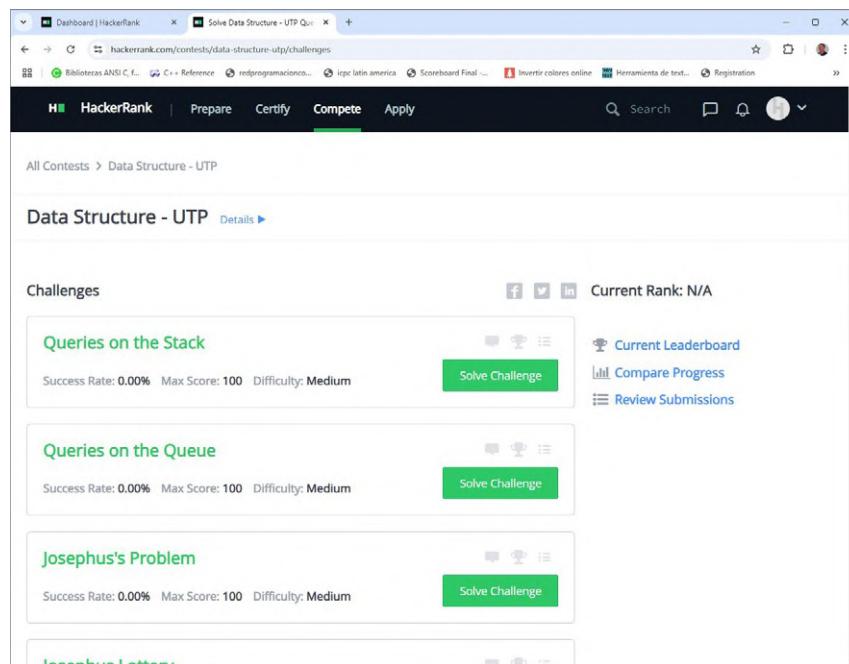


Figura A.17. Página con el listado de los retos de programación del libro.



# Bibliografía

- [AHU98] Alfred V. Aho, John E. Hopcroft y Jeffrey D. Ullman. *Estructuras de Datos y Algoritmos*. Addison-Wesley, 1998.
- [Bec95] César Becerra Santamaría. *Estructuras de Datos en C (quinta edición)*. Por Computador Ltda, 1995.
- [BV02] S. Baase y A. Van Gelder. *Algoritmos Computacionales, Introducción al Análisis y Diseño (tercera edición)*. Pearson Educación, 2002.
- [Cai89] Xavier Caicedo. *Elementos de Lógica y Calculabilidad*. Una Empresa Docente, 1989.
- [CBL23] John Canning, Alan Broder y Robert Lafore. *Data Structures & Algorithms in Python*. Pearson Education, 2023.
- [CC21] Steven C. Chapra y Raymond P. Canale. *Numerical Methods for Engineers (eighth edition)*. McGraw-Hill Education, 2021.
- [Ceb03] Fco. Javier Ceballos Sierra. *Java 2: Curso de Programación (segunda edición)*. Alfaomega/Ra-Ma, 2003.
- [Cor+01] T. H. Cormen, C. E. Leiserson, R. L. Rivest y C. Stein. *Introduction to Algorithms (second edition)*. The MIT Press, 2001.
- [Cor13] Thomas H. Cormen. *Algorithms Unlocked*. The MIT Press, 2013.
- [DPV06] S. Dasgupta, C. H. Papadimitriou y U. V. Vazirani. *Algorithms*. 2006.
- [Fag+14] José Fager, W. Libardo Pantoja Y., Marisol Villacrés, Luz Andrea Páez M., Daniel Ochoa y Ernesto Cuadros V. *Estructuras de Datos (primera edición)*. Iniciativa Latinoamericana de Libros de Texto Abiertos (LATIn), 2014.
- [GM98] J. A. García Cruz y A. Martinón. “Números Poligonales”. En: *Educación Matemática* 10.3 (1998), págs. 109-127. URL: <https://doi.org/10.24844/EM1003.07>.
- [HHE18] Steven Halim, Felix Halim y Suhendry Effendy. *Competitive Programming 4. The Lower Bound of Programming Contests in the 2020s. Book 1 (4th edition)*. Lulu Press, 2018.
- [Hil14] Paul N. Hilfinger. *Data Structures (Into Java), seventh edition*. University of California, 2014.
- [ICF24] Instituto Colombiano para el Fomento de la Educación Superior ICFES. *Guía de Orientación del examen Saber Pro, aplicación 2024-02, Módulo de Competencias Genéricas*. 2024. URL: <https://goo.su/dL44b9> (visitado 02-08-2024).
- [JZ98] Luis Joyanes A. e Ignacio Zahonero M. *Estructura de Datos: Algoritmos, Abstracción y Objetos*. McGraw-Hill Interamericana, 1998.

- [Kar20] Elshad Karimov. *Data Structures and Algorithms in Swift: Implement Stacks, Queues, Dictionaries, and Lists in Your Apps*. Apress Media, 2020.
- [Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3, Sorting and Search (Second Edition)*. Addison-Wesley, 1998.
- [KT06] Jon Kleinberg y Éva Tardos. *Algorithm Design (first edition)*. Addison-Wesley, 2006.
- [Laa24] Antti Laaksonen. *Guide to Competitive Programming: Learning and Improving Algorithms Through Contests (third edition)*. Springer, 2024.
- [LAT97] Y. Langsam, M. J. Augenstein y A. M. Tenenbaum. *Estructuras de Datos con C y C++ (segunda edición)*. Prentice-Hall Hispanoamericana, 1997.
- [Man20] Luciano Manelli. *Introducing Algorithms in C: A Step by Step Guide to Algorithms in C*. Apress Media, 2020.
- [MM21] Dheeraj Malhotra y Neha Malhotra. *Data Structures and Program Design Using Python*. Mercury Learning e Information, 2021.
- [MR13] B. Miller y D. Ranum. *Problem Solving with Algorithms and Data Structures (Release 3.0)*. 2013.
- [MS07] Kurt Mehlhorn y Peter Sanders. *Algorithms and Data Structures. The Basic Toolbox*. Springer, 2007.
- [Ros04] K. H. Rosen. *Matemática Discreta y sus Aplicaciones (quinta edición)*. Mc Graw Hill Interamericana de España, 2004.
- [Rou17] Tim Roughgarden. *Algorithms Illuminated. Part 1: The Basics (first edition)*. Sound-likeyourself Publishing, 2017.
- [Ski20] Steven S. Skiena. *The Algorithm Design Manual (third edition)*. Springer, 2020.
- [SO15] Guillermo Roberto Solarte y Carlos Alberto Ocampo. *Guía didáctica de Estructuras de Datos*. Editorial UTP, 2015.
- [SR08] Steven S. Skiena y Miguel A. Revilla. *Desafíos de Programación. El manual de entrenamiento para concursos de programación*. Edición Colombiana, 2008.
- [SW11] R. Sedgewick y K. Wayne. *Algorithms (fourth edition)*. Addison-Wesley Professional, 2011.
- [Ueh19] Ryuhei Uehara. *First Course in Algorithms Through Puzzles*. Springer Nature Singapore Pte Ltd, 2019.
- [Vil96] Jorge A. Villalobos S. *Diseño y Manejo de Estructuras de Datos en C*. McGraw-Hill Interamericana, 1996.
- [Wen20] Jay Wengrow. *A Common-Sense Guide to Data Structures and Algorithms (second edition)*. The Pragmatic Programmers, 2020.
- [Wil64] J. W. J. Williams. “Algorithm 232 (HEAPSORT)”. En: *Communications of the ACM* 7 (1964), pp. 347-348.

La Editorial de la Universidad Tecnológica de Pereira tiene como política la divulgación del saber científico, técnico y humanístico para fomentar la cultura escrita a través de libros y revistas científicas especializadas.

Las colecciones de este proyecto son: Trabajos de Investigación, Ensayos, Textos Académicos y Tesis Laureadas.

Este libro pertenece a la colección **Textos Académicos**.

El **ICPC (International Collegiate Programming Contest - Concurso Internacional de Programación Universitaria)**, reúne los concursos de programación universitaria en todos sus niveles: nacional, regional/continental y mundial. A los participantes de estos concursos son presentados un conjunto de 12 retos de programación (en promedio) y con los cuales se ponen a prueba sus conocimientos y habilidades en Ciencias de la Computación, en donde la clave son las **Estructuras de Datos** y los algoritmos que trabajan sobre ellas.

La Universidad Tecnológica de Pereira ha clasificado y participado en 4 mundiales de programación del ICPC y ha liderado junto con otras universidades colombianas la **Red de Programación Competitiva (RPC)** para toda Latino América. Este libro surge gracias al trabajo continuo (durante más de 10 años) de muchos colaboradores en la escritura y validación de retos de programación para las competencias de la RPC.

Este libro fue diseñado para estudiantes de primer o segundo año de los programas académicos de Tecnología o Ingeniería de Sistemas que ya hayan trabajado previamente un curso de programación en Lenguaje C.

Este es el material ideal para todos aquellos estudiantes de pregrado que quieran comenzar en el mundo de la programación competitiva, sacando el mayor provecho de las estructuras de datos trabajadas en el libro y de su aplicación práctica en la resolución de retos de programación.