Editorial Técnica y Resolución de Problemas de la Categoría Universitaria — Copa Salvadoreña de Programación 2025 (C3)

1st Diego Francisco Arévalo Miranda
20245123@esen.edu.sv
2nd Roberto Carlos Morán Gómez
20245258@esen.edu.sv
3rd Christopher Alexander Marroquín Figueroa
20245332@esen.edu.sv
4th Roberto Josué Polanco Hernández
20245367@esen.edu.sv
5th Óscar José Pleités Lemus
20245452@esen.edu.sv

Escuela Superior de Economía y Negocios (ESEN)
Competitive Coding Club (C3)

***Abstract*—Este artículo presenta la editorial oficial de los problemas universitarios realizados para la Copa Salvadoreña de Programación 2025, organizada por el Competitive Coding Club (C3) en la Escuela Superior de Economía y Negocios (ESEN). El documento describe el contexto, la justificación, los principios de diseño de problemas y la resolución detallada de cada uno. Además, analiza la aplicación de estructuras de datos y algoritmos fundamentales en escenarios prácticos de programación competitiva. La publicación de este editorial busca fortalecer el aprendizaje, la autoevaluación y la difusión de la programación competitiva en El Salvador.**

***Index Terms*—Competitive Programming, Editorial de problemas, Soluciones Algoritmicas, Programación Competitiva.**

I. Introducción

La programación competitiva es una de las prácticas más efectivas para desarrollar pensamiento algorítmico, resolución de problemas bajo presión, trabajo colaborativo y habilidades técnicas avanzadas en estudiantes. Competencias internacionales como el International Collegiate Programming Contest (ICPC), combinan el rigor técnico con la motivación del reto académico, generando un impacto significativo en la formación de profesionales en áreas STEM.

En este contexto, nace la **Copa Salvadoreña de Programación**, organizada por el *Competitive Coding Club (C3)* en la Escuela Superior de Economía y Negocios (ESEN). Este evento tiene como propósito fomentar una cultura de excelencia algorítmica en El Salvador, ofreciendo a estudiantes universitarios y de bachillerato una plataforma para aplicar sus conocimientos en programación, algoritmos y estructuras de datos en escenarios desafiantes y contextualizados.

La edición 2025 de la Copa C3 se desarrolló en dos fases: una **fase virtual clasificatoria**, en la que participaron más de una docena de equipos de todo el país, y una **fase presencial final**, que reunió a los mejores exponentes para competir en problemas de mayor complejidad técnica. Ambas fases siguieron el modelo de competencia ICPC, donde equipos de tres estudiantes comparten una estación de trabajo y deben resolver de 6 a 8 problemas en un límite de 4 horas, con evaluaciones automatizadas y rankings dinámicos.

El presente artículo técnico tiene como objetivo documentar de forma rigurosa la construcción y evaluación de los problemas propuestos durante la competencia. A lo largo del documento se detallan las metodologías utilizadas para su diseño, validación y publicación, se presenta la editorial oficial con soluciones funcionales en Python, y se analizan los resultados obtenidos por los equipos en ambas fases. Además, se reflexiona sobre el impacto educativo del evento y su contribución al fortalecimiento del ecosistema de programación competitiva en El Salvador.

Esta investigación se enmarca dentro del curso de *Estructuras de Datos y Algoritmos*, integrando tanto el desarrollo técnico de problemas como su aplicación pedagógica en entornos reales. La Copa se presenta no solo como una competencia, sino como una herramienta formativa de alto valor para estudiantes, docentes y organizadores.

II. Descripción del Problema

La Copa Salvadoreña de Programación 2025 es una competencia de programación competitiva organizada por el Competitive Coding Club (C3). Su propósito es fomentar habilidades cruciales en resolución de prob-

lemas, lógica y ciencias de la computación, reuniendo a estudiantes talentosos de colegios y universidades para potenciar su aprendizaje integral y desarrollo en el ámbito tecnológico.

La Copa se inspira en el formato del International Collegiate Programming Contest (ICPC). Su objetivo es desarrollar y fortalecer las habilidades de resolución de problemas, lógica algorítmica, trabajo en equipo y toma de decisiones bajo presión.

La estructura de la Copa se inspira en el esquema clásico ICPC: los participantes se organizan en equipos de hasta tres estudiantes universitarios, comparten una estación de trabajo y resuelven problemas de programación con límite de tiempo y recursos.

Para la edición 2025, la competencia se dividió en dos fases:

- **Fase Virtual:** [5] Etapa preliminar que permitió a los equipos enfrentarse a problemas de dificultad intermedia, practicar la dinámica de trabajo en equipo y validar su nivel de preparación. Los mejores resultados de esta jornada, fueron seleccionados para competir en la siguiente fase.
- **Fase Presencial:** [6] Final presencial donde los mejores equipos resolvieron problemas de mayor dificultad, enfocados en temas más avanzados y con mayor exigencia.

Los principales objetivos de la Copa son:

- Fomentar la práctica de programación competitiva en El Salvador.
- Brindar a los estudiantes una experiencia cercana a estándares internacionales.
- Desarrollar pensamiento crítico, habilidades de depuración y colaboración efectiva.
- Generar una cultura de resolución de problemas basada en algoritmos y estructuras de datos.

A. Temario Cubierto

Los problemas propuestos abarcan temas esenciales de Estructuras de Datos y Algoritmos y tradicionales de la programación competitiva, incluyendo:

- Estructuras de Datos estándar
- Simulación
- Búsqueda completa
- Ordenamiento
- Estrategias Greedy
- Recursividad
- Geometría
- Introducción a Grafos
- Dos punteros
- Ad Hoc
- Sumas de prefijos
- Comparadores Customizados
- Navegación de grafos (BFS, DFS)
- Búsqueda Binaria
- Relleno por difusión (Flood Fill)
- Introducción a Arboles
- Matemática (Combinatorio y Teoría de números)
- Programación Dinámica
- Ordenamiento Topológico

B. Especificaciones de la competencia

La competencia siguió un formato similar al utilizado en concursos ICPC y adaptado a las capacidades logísticas de la plataforma OmegaUp. Los equipos estuvieron conformados por hasta tres estudiantes, resolviendo entre 6 y 8 problemas en un tiempo máximo de 4 horas en cada fase.

Cada problema era evaluado automáticamente con conjuntos de casos ocultos, permitiendo múltiples envíos, pero penalizando el tiempo por cada envío incorrecto. El ranking se ordenó por número de problemas resueltos y, en caso de empate, por tiempo total acumulado.

Los lenguajes permitidos fueron*C++, Python y Java*, priorizando la compatibilidad con los entornos típicos de competencias internacionales.

C. Fuentes y referencias educativas

Para el diseño de los problemas se tomaron como referencia estructuras, estilos y niveles de dificultad similares a los utilizados en competencias de alto nivel como el [1] International Collegiate Programming Contest (ICPC) y la plataforma [2] Codeforces, ambas ampliamente reconocidas por su enfoque riguroso y pedagógico en la resolución algorítmica de problemas.

Como base temática se utilizó la clasificación por niveles de la [3] USACO Guide (United States of America Computing Olympiad), que organiza los conocimientos en niveles: bronze, silver y gold, cubriendo progresivamente estructuras de datos, técnicas de simulación, grafos, programación dinámica, greedy, búsqueda binaria, entre otros.

Finalmente, para la gestión técnica del concurso se utilizó la plataforma [4] OmegaUp, una herramienta de evaluación automática ampliamente adoptada en competencias hispanoamericanas, que permite gestionar concursos, definir problemas, cargar casos de prueba, y visualizar rankings en tiempo real.

III. Metodología

El diseño editorial de los problemas de la Copa Salvadoreña de Programación 2025 sigue un proceso estructurado y reproducible, con el fin de asegurar la calidad técnica, la claridad narrativa y la validez de cada desafío.

A. Formato Estandarizado

La editorial de cada problema se presenta bajo un formato unificado que incluye:

- **Planteamiento del Problema:** Descripción general, contexto narrativo y especificación clara de entradas, salidas, ejemplos de prueba y restricciones.
- **Solución:** Explicación de la idea central, enfoque algorítmico, complejidad esperada y consideraciones clave.
- **Código de Referencia:** Implementación funcional en Python que resuelve el problema y sirve como solucion al problema en la plataforma OmegaUp.

B. Proceso de Desarrollo

El proceso metodológico contempló las siguientes etapas:

1) **Planteamiento de la Idea:** Cada problema se originó a partir del temario establecido, seleccionando y combinando algunas de estructuras de datos, algoritmos y estrategias.
2) **Diseño del Problema:** Con base en la idea principal, se definieron distintos escenarios, construyendo casos de prueba que representaran distintos niveles de dificultad (según los datos de entrada). También se plantearon 'corner cases', que son situaciones especificas para probar los algoritmos y soluciones de los participantes.
3) **Redacción del Enunciado:** Se generó la narrativa o contexto del problema para hacer la experiencia más atractiva y divertida para los participantes.
4) **Configuración Técnica:** Cada problema se subió a *OmegaUp* como problema privado. Luego, se organizó dentro de un concurso privado que solo el equipo de organización y testers podía ver.
5) **Pruebas y Validación:** Un equipo de testers resolvió todos los problemas, aportando soluciones, detectando ambigüedades en enunciados, verificando rangos de entrada, revisando límites de tiempo y explorando casos borde.
6) **Revisión Final y Publicación:** Tras la aprobación técnica, los problemas se integraron en concursos privados listos para la competencia. Los links fueron compartidos exclusivamente con los equipos participantes el día del evento.

Este flujo de trabajo asegura que cada problema cumpla estándares de calidad, sea resoluble dentro del tiempo establecido y aporte valor formativo a los participantes.

IV. Editorial de Problemas

A. FASE VIRTUAL

1) Problema 1: Encontrando a los Espías Deceptinant:

Enunciado:

¡Bienvenido a la **Copa Salvadoreña de Programación**! Aunque sea tu primer día en los **Autovengadores**, tu ayuda es **crucial**.

Nuestra red de inteligencia, **MIP7**, ha detectado la infiltración de espías **Deceptinant** entre nuestros rangos. Afortunadamente, uno de nuestros aliados, el agente alemán *Manuel Mütze*, logró enviarnos información valiosa a través de un canal seguro.

La información recibida consiste en un arreglo de N enteros. Cada entero representa la **confianza** de Manuel hacia un agente, así como su **influencia** en su sector.

- Un número **positivo** indica **confianza**.
- Un número **negativo** indica **desconfianza**.
- El **valor absoluto** del número representa la magnitud de la **influencia** del agente.

Tu misión es determinar el estado de seguridad de Q sectores dentro de la organización. Cada sector está representado por un rango de posiciones $[L, R]$ (inclusive) dentro del arreglo. Un sector se considera **comprometido** si la suma total de las influencias de sus miembros es **negativa**. En caso contrario, se considera **seguro**.

Entradas:

- La primera línea contiene un entero N, el número de agentes.
- La segunda línea contiene N enteros distintos $A_1, A_2, ..., A_n$, donde cada A_i representa la influencia del agente en la posicion i.
- La tercera línea contiene un entero Q, el número de sectores a evaluar. Las siguientes Q líneas contienen dos enteros L y R, que representan el rango del sector.

Salidas:

Imprime Q líneas, cada una con un solo número:

- 1, si la suma total de las influencias en el sector correspondiente es mayor o igual a cero.
- 0, si dicha suma es negativa.

Ejemplo de Entrada:

```
5
1 -2 3 -4 5
3
1 5
2 2
3 4
```

Ejemplo de Salida:

```
1
0
0
```

Restricciones:

- $1 \leq N \leq 2 \times 10^5$
- $-1000 \leq A_i \leq 1000$, para todo $1 \leq i \leq N$
- $1 \leq Q \leq 2 \times 10^3$
- $1 \leq L \leq R \leq N$

a) Solución: La solución esperada de este problema es una aplicación clásica de la técnica 'prefix sums' (prefijos de sumas). Para ello, no es necesaria ninguna observación clave, solamente es necesario realizar los siguientes pasos.

- **Almacenar los datos**, de tal forma que $suma[i] = suma[i-1] + A[i-1]$
- **Imprimir las respuestas**, 1 si $suma[r] - suma[l-1] >= 0$, 0 en caso contrario.

El objetivo de **prefix sums** [3] es almacenar la suma de los valores en un rango desde 0 hasta cualquier punto. Esto puede ser utilizado para encontrar la suma de los elementos en cualquier rango $[a, b]$ en una sola consulta, que por definición sería la suma hasta el punto b menos la suma hasta el punto $a-1$, permitiendo así que la complejidad de esta solución sea de $O(N+Q)$.

Fundamento teórico: Las *prefix sums* o sumas de prefijos permiten calcular la suma de elementos en cualquier subarreglo en tiempo constante luego de un preprocesamiento lineal [2]. La idea es crear un arreglo auxiliar donde cada posición almacena la suma de todos los elementos anteriores. Así, para calcular la suma de un rango `[L, R]`, se usa la fórmula: `prefix[R] - prefix[L - 1]`.

b) Código (Python):

```
def solve():
    n = int(input())
    suma = [0] * (2 * 100005)

    values = list(map(int, input().split()))

    for i in range(1, n + 1):
        suma[i] = suma[i - 1] + values[i - 1]

    q = int(input())
    for _ in range(q):
        l, r = map(int, input().
            split())

        print(1 if suma[r] -
            suma[l - 1] >= 0 else 0)
```

```
t = 1
#t = int(input())
for _ in range(t):
    solve()
```

2) Problema 2: El Adabraniumth Utilizado:

Enunciado:
Los **Deceptinant** continúan atacando a los **Autovengadores**, por lo que es crucial completar el proyecto más secreto de todos: el **Autovengajet**. Este prototipo fue diseñado por los ingenieros *Zinedine Kaká* y *Luís Nazario*, pero la instalación de su armadura está bajo la responsabilidad del **profe Pleités**.
Aunque nadie sabe cómo el **profe Pleités** consiguió un rol tan importante dentro de los Autovengadores, la teoría más aceptada es que **siempre aparece en el lugar indicado en el momento justo** (y pues es algo **gracioso**). Como es costumbre, el profe dejó todo para el **último momento** y realizó su trabajo de forma apresurada, lo que ha generado dudas sobre su eficiencia, particularmente en un área de tamaño $N \times M$.
En este espacio, separado en casillas, el profe instaló una serie de hojas **rectangulares** del metal más fuerte del planeta, **Adabraniumth**.
No se conoce la cantidad exacta de hojas utilizadas, pero sí se dispone del código de color X que se alcanza a ver en cada casilla del área. Debido a un proceso **químico-físico-atómico-nuclear** involucrado en su creación, **no existen dos hojas diferentes con el mismo color**.

Entradas:
La primera línea contiene dos enteros naturales N y M, que representan el número de filas y columnas del área a evaluar.
Las siguientes N líneas contienen M enteros naturales cada una. Cada entero X representa el código de color de Adabraniumth visible en la casilla correspondiente.

Salidas:
Imprime una sola línea con un entero: la cantidad mínima de hojas utilizadas por el **profe Pleités**.

Ejemplo de Entrada:

```
3 4
1 1 3 3
9 9 9 3
1 1 6 6
```

Ejemplo de Salida:

```
4
```

Restricciones:
$1 \leq N, M \leq 1000, \quad 0 \leq X \leq 1000$

a) Solución: Para resolver este problema, se esperaba que los concursantes llegaran a la conclusión de que bastaba con contar la cantidad de colores (representados por números) que aparecen en la matriz, a excepción del 0. Para ello, se recomienda utilizar una estructura de tipo set, aunque se podría obtener el mismo resultado mediante otros métodos, como un arreglo de tipo booleano. La complejidad de esta solución es de O(N).
Fundamento teórico: El uso de conjuntos (`set` en Python) permite almacenar elementos únicos, eliminando automáticamente duplicados [2]. Esto resulta útil para contar cuántos colores distintos aparecen en una matriz, especialmente si se desea ignorar un color específico (como el 0).

b) Código (Python):

```
import sys
input = sys.stdin.readline

def solve():
    n, m = map(int, input().split())
    aparece = set()

    for i in range(n):
        row = list(map(int, input()
            .split()))

        for x in row:
            if x != 0:
                aparece.add(x)

    print(len(aparece))

t = 1
#t = int(input())
for _ in range(t):
    solve()
```

3) Problema 3: ¿Quién Hizo Esta Diablura?:

Enunciado:
"¿Quién hizo esta diablura?" Esas fueron las **devastadoras** palabras con las que el **profe Pleités** reaccionó ante la **terrible imagen** que se le presentaba. Algún miembro espía de los **Deceptinant**, con un claro objetivo de desestabilizar a los **Autovengadores**, había creado una imagen con **Inteligencia Artificial** del profe con una camiseta del *FC Barcelona*, un claro acto contrario a la moral.
El profe está **seriamente preocupado** por su reputación ante el resto de la organización, puesto que no es posible que se **hable mal de un miembro tan responsable, valioso y gracioso** del equipo. Es por ello que ha

pedido encontrar cuántas personas **pueden haber visto la imagen**.
Hay N miembros en los Autovengadores, además, sabemos que hay M parejas de miembros que **se hablan constantemente entre sí**.
Sabemos que **una sola persona** realizó la **diablura original**, pero no sabemos quién fue o a cuántas personas le contó. ¿Cuál es el **máximo y mínimo número de personas** que pueden haberse enterado?

Importante: Esta diablura es un **suceso muy polémico**, por lo que, si el tema es conocido por uno de los miembros, **es seguro que el tema saldrá en cualquiera de sus conversaciones**.

Entradas:
La primera línea consiste en 2 enteros N y M, la cantidad de miembros y la cantidad de conversaciones realizadas. Las siguientes M líneas presentan 2 enteros, U y V, que representan una conversación entre los miembros U y V.

Salidas:
Debes imprimir 2 líneas. En la primera, la **máxima** cantidad de personas y en la segunda la **mínima** cantidad que puede haber visto la **diablura**.

Restricciones:

- $1 \leq N \leq 10^5$
- $1 \leq M \leq 2 \times N$
- $1 \leq U, V \leq N$

Ejemplo de Entrada:

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

Ejemplo de Salida:

```
6
6
```

a) Solución: Para la solución esperada, las personas se toman como nodos, y sus conversaciones, aristas, y de esta forma, se generan uno o más grafos con aristas bidireccionales. En este punto, la observación clave es que la persona que haya generado la diablura provocará que todas las personas de su grafo se enteren (puesto que las personas que se hablan entre sí lo hacen de forma constante), por lo que el problema se transforma en encontrar los grafos con la mayor y menor cantidad de nodos.

Para ello, se recomienda hacer uso de cualquiera de los 2 algoritmos de recorrido de grafos más comunes:

- **DFS**, la solución base.
- **BFS**, una alternativa.

Para evitar una complejidad cuadrática, se hace uso de un arreglo de vistos, que asegura que el algoritmo no busque el tamaño del mismo grafo 2 veces; además, esto también evita que se pase por nodos ya vistos en el recorrido. Esto es válido pues no importa en qué nodo se 'entre' al grafo, el tamaño de este será el mismo.
Esta solución cuenta con una complejidad de O(N).
Fundamento teórico: Este problema puede modelarse mediante un grafo no dirigido, donde los nodos representan personas y las aristas representan conversaciones entre ellas. El mensaje se propaga completamente dentro de cada componente conexo. Por tanto, para encontrar el número mínimo y máximo de personas que pudieron haber visto la imagen, se debe usar un recorrido en profundidad o en amplitud (DFS o BFS) dentro de cada grafo para encontrar todos los componentes que lo conforman.

b) Código (Python):

```
import sys
input = sys.stdin.readline

def dfs(nodo, grafo, visto):
    global act
    stack = [nodo]
    while stack:
        curr = stack.pop()
        if curr in visto:
            continue
        visto.add(curr)
        act += 1
        for neighbor in grafo[curr]:
            if neighbor not in visto:
                stack.append(neighbor)

def solve():
    global act, maxi, mini
    n, m = map(int, input().split())
    grafo = [[] for _
        in range(200005)]

    visto = set()
    maxi = 0
    mini = float('inf')

    for _ in range(m):
        u, v = map(int,
            input().split())
```

```
        grafo[u].append(v)
        grafo[v].append(u)

    for i in range(1, n + 1):
        if i not in visto:
            act = 0
            dfs(i, grafo, visto)
            maxi = max(maxi, act)
            mini = min(mini, act)

    print(maxi)
    print(mini)

t = 1
# t = int(input())
for _ in range(t):
    solve()
```

4) Problema 4: Los Códigos de los Autovengadores:

Enunciado:
Una nueva generación de candidatos ha llegado para unirse a los **Autovengadores**. ¡Es momento de evaluarlos! Ser un Autovengador no solo requiere una **condición física impecable**, sino también una **capacidad intelectual de élite**.
Este año, una de las pruebas presentadas a los aspirantes se basa en un escenario inspirado en **hechos reales**: la generación de **códigos de seguridad** para las bases de los Autovengadores.
El jefe de Relaciones Justicieras, *Marrochris*, ha liderado una investigación exhaustiva con el objetivo de encontrar los mejores caracteres para construir contraseñas seguras. Su conclusión es clara: es preferible utilizar exactamente N caracteres, concretamente, aquellos presentes en la cadena S.

Tu tarea consiste en determinar **cuántas cadenas distintas** pueden formarse utilizando **todos** los caracteres de la cadena S, considerando que algunos caracteres pueden **repetirse**.
Dado que el número de combinaciones puede ser muy grande, debes imprimir la respuesta **módulo** $10^9 + 7$.

Entradas:

- La primera línea contiene un número entero N, la cantidad de caracteres a utilizar.
- La segunda línea contiene una cadena S de longitud N, con los caracteres seleccionados por Marrochris.

Salidas:
Imprimirás un solo entero: la cantidad de cadenas diferentes que se pueden formar, módulo $10^9 + 7$.

Restricciones:

- $1 \leq N \leq 1000$
- La cadena S puede contener letras mayúsculas y minúsculas del alfabeto inglés, así como dígitos del 0 al 9.

Ejemplo de Entrada:

```
3
aab
```

Ejemplo de Salida:

```
3
```

Ejemplo de Entrada:

```
5
abcd9
```

Ejemplo de Salida:

```
120
```

a) Solución: La cantidad total de permutaciones posibles de una cadena de N caracteres es $N!$, pero si hay caracteres que se repiten, es necesario dividir entre el factorial de las repeticiones para evitar contar combinaciones idénticas múltiples veces [2]. En otros términos, la fórmula sería:

$$\text{Res} = \frac{N!}{f_1! \cdot f_2! \cdot \ldots \cdot f_k!}$$

donde f_i es la frecuencia de cada carácter. Sin embargo, simular todo este proceso resultaría en una complejidad demasiado alta, por lo que es necesario encontrar una forma más eficiente.
En lugar de calcular factoriales directamente, se construye un triángulo de Pascal para obtener los coeficientes multinomiales de bajo el módulo de $10^9 + 7$ [3]. Esta solución cuenta con una complejidad de $O(N^2)$

b) Código (Python):

```
import sys
from collections import Counter
input = sys.stdin.readline

MOD = 10**9 + 7
n = int(input())
p = input().strip()

pascal = [[0] * (n + 1) for _ in
    range(n + 1)]

for i in range(n + 1):
    pascal[i][0] = 1
```

```
for i in range(1, n + 1):
    for j in range(1, i + 1):
        pascal[i][j] = (pascal[i - 1]
            [j] + pascal[i - 1]
            [j - 1]) % MOD

hay = Counter(p)

ans = 1
for freq in hay.values():
    ans = (ans * pascal[n][n - freq])
        % MOD

    n -= freq

print(ans)
```

---

5) Problema 5: La Escuela Polanski para Estudiantes Programadores:

Aclaración adicional:
Este problema presentó 2 versiones, que se diferencian en los límites de las restricciones. Esto fue necesario para evitar beneficiar a los concursantes que utilizaran los lenguajes más rápidos (como C++), o evitar perjudicar a los que usaran lenguajes más lentos (como Python).

Enunciado:
Los **Autovengadores** no están solos en su lucha contra las fuerzas del mal. Por suerte, cuentan con el apoyo de otros equipos que, como ellos, luchan cada día por el **bien de la programación**. Uno de sus más grandes aliados es el grupo de los **C++Men**, un equipo de héroes liderado por **Robert Polanski**.
Desde pequeño, **Polanski** demostró habilidades excepcionales: resolvía ecuaciones trigonométricas diferenciales cuánticas de séptimo grado **mentalmente**, se comunicaba con **entidades temidas** incluso por los Autovengadores, y se convirtió en un **referente** en la enseñanza de nuevas generaciones. Sin embargo, tras una batalla contra el temible **Lord Guardiola**, su visión fue afectada por un hechizo que le provocó **Astigmiopía Cataráctica Biogenética**.
Lejos de rendirse, Polanski fundó la **Escuela Polanski para Estudiantes Programadores**, y con ella, a los legendarios **C++Men**.
Cuando estableció su escuela, Polanski impuso una regla clara: **durante los próximos N periodos de admisión, exactamente X alumnos deben ser admitidos en cada uno de esos periodos**.
Gracias a sus dones intelectuales, Polanski puede predecir con exactitud que en el periodo i habrá A_i postulantes, y que cada uno de ellos está dispuesto a permanecer en una lista de espera por hasta K_i periodos adicionales. Es decir, si un estudiante aplica en el periodo i, podrá ser admitido en cualquier periodo j tal que $i \leq j \leq i + K_i$.
Como Polanski está ocupado salvando el mundo, te ha pedido que lo ayudes para determinar el **valor máximo posible de** X, la cantidad de alumnos que pueden ser admitidos en cada periodo, bajo estas condiciones.

Entradas:

- La primera línea contiene un número entero N: el número de periodos de admisión.
- Las siguientes N líneas contienen dos enteros A_i y K_i, indicando respectivamente la cantidad de estudiantes que aplican en el periodo i, y cuántos periodos adicionales pueden esperar.

Salidas:
Imprime un solo número entero: el **valor máximo de** X, la cantidad de estudiantes que pueden ser admitidos en cada periodo cumpliendo las reglas dadas.

Restricciones:

- $1 \leq N \leq 10^4$
- $1 \leq A_i \leq 10^9$
- $0 \leq K_i \leq N$

Ejemplo de Entrada:

```
3
4 1
2 1
3 0
```

Ejemplo de Salida:

```
3
```

Ejemplo de Entrada:

```
4
2 0
2 0
2 0
10 0
```

Ejemplo de Salida:

```
2
```

a) Solución: Para resolver este problema, se esperaban las siguientes observaciones clave:

- **Búsqueda Binaria**: Si un valor X es válido, todos los valores menores a él serán válidos también. De la misma forma, si un valor X no es válido, todos los mayores a él tampoco lo serán [2].
- **Prioridad para admitir**: La mejor opción es siempre admitir a quienes estén más cerca de dejar de esperar. Esto se puede hacer de forma óptima mediante una estructura como una **cola de prioridad**.

- **Simulación optimizada**: Los límites del problema permiten una solución que utilice **Búsqueda Binaria** para los valores de X, simulando para todos los períodos si sería válido.

Aplicando las observaciones anteriores, se almacenan los datos en la forma período límite, cantidad de aplicantes. Mediante una **Búsqueda Binaria**, se llama una función encargada de simular el proceso de admisión de forma óptima.

En esta función, se utiliza una **cola de prioridad** [3] para admitir a todos los estudiantes posibles (para el X dado) que pueden ser admitidos en este período. Si no hay suficientes aplicantes, el valor de X es inválido, caso contrario, se almacenan los sobrantes (siguiendo el formato período, aplicante) y se continúa hasta que se hayan examinado los N períodos, en cuyo caso, el valor de X es válido.

La complejidad de esta solución es de $O(log(N)^2 * N)$.

b) Código (Python):

```
import heapq
import sys
input = sys.stdin.readline

def solve():
    n = int(input())
    values = []
    for i in range(n):
        a, k = map(int, input().
            split())

        values.append((k + i,
            a))

    def check(limit):
        pq = []
        period = 0
        for i in range(n):
            while pq and pq[0][0]
                < period:

                heapq.heappop(pq)

            period += 1

            heapq.heappush(pq,
                values[i])

            aux = limit

            while pq and aux -
                pq[0][1] >= 0:
                aux -= pq[0][1]
                heapq.heappop(pq)
            if not pq:
                if aux != 0:
                    return False
            else:
                top = heapq.
                    heappop(pq)

                new_cost = top[1]
                    - aux

                heapq.heappush(pq, (
                    top[0], new_cost))

        return True

    left, right = 0, int(1e9) + 5
    while left + 1 < right:
        mid = (left + right) // 2
        if check(mid):
            left = mid
        else:
            right = mid

    print(right if check(right) else
        left)

solve()
```

6) Problema 6: Buscando la Base Deceptinant 2:

Enunciado:

Los **Autovengadores** han resistido incontables ataques de los temibles **Deceptinant** durante esta temporada... Ha llegado el momento de **contraatacar**.

Nuestros mejores estrategas, *Carlo Mourinho* y *Jürgen Zidane*, han diseñado un plan maestro para ubicar la base enemiga. Sin embargo, llevarlo a cabo no será sencillo. Para ejecutarlo, necesitamos la ayuda de **Ic-Son**, la mítica mascota del *último hijo de Apopa* y fiel compañero de **Robert Polanski**, líder de los **C++Men**.

El legendario domador de firewalls, el **Viejo Sigma**, es nuestro mayor experto en los Deceptinant, principalmente porque, según se dice, ha estado presente en este mundo desde **el principio de los tiempos**. El Viejo Sigma está convencido de que la base enemiga está oculta en algún lugar dentro de una región de tamaño $N \times M$, donde N representa el número de filas (alto) y M el número de columnas (ancho) del terreno.

Utilizando datos enviados por nuestro servicio de **IP (Inteligencia Profesional)**, **Marrochris** ha determinado que la base Deceptinant debe ubicarse en un **terreno**

plano. Se define como terreno plano a toda área en la cual **todas las casillas tienen la misma altura**.
Además, según una teoría confirmada por el **profe Pleités** (quien sigue consternado por la diablura anterior, así que, sorprendentemente, está trabajando), debido al ego desmedido de los Deceptinant, utilizarán **todo el espacio posible** (si pueden tomar más terreno plano, lo harán).
Tu misión es encontrar **cuántas áreas del terreno califican como posibles ubicaciones para la base**, es decir, **cuántas subregiones del terreno son completamente planas**.

Entradas:

- La primera línea contiene dos enteros N y M, el número de filas y columnas del terreno.
- Las siguientes N líneas contienen M enteros X cada una, representando la altura del terreno en la posición correspondiente.

Salida:
Imprime un solo número entero: la **cantidad de subregiones completamente planas** que pueden ser utilizadas por los Deceptinant.

Restricciones:

- $1 \leq N, M \leq 1000$
- $1 \leq X \leq 10^6$

Ejemplo de Entrada:

```
3 3
1 1 2
3 3 2
1 3 3
```

Ejemplo de Salida:

```
4
```

Ejemplo de Entrada:

```
3 3
1 1 2
1 1 2
1 1 3
```

Ejemplo de Salida:

```
3
```

a) Solución: La solución esperada de este problema es una aplicación del algoritmo **Flood Fill** [2].
Para que este sea eficiente, se debe almacenar un arreglo de vistos, que evite que se visite la misma casilla múltiples veces, además de que solo hay que continuar por aquellas casillas que tengan la misma altura que aquella desde la que se partió originalmente. La respuesta será la cantidad de **componentes conexos** existentes en la matriz. [3]
La complejidad de esta solución es de O(N * M).

b) Código (Python):

```
import sys
sys.setrecursionlimit(1 << 25)
input = sys.stdin.readline

n, m = map(int, input().split())
matriz = [list(map(int,
    input().split())) for _ in
        range(n)]

visto = [[False] * m for _ in
    range(n)]

def flfl(f, c, alto):
    if f < 0 or c < 0 or f >= n or
        c >= m:

        return

    if matriz[f][c] != alto or
        visto[f][c]:

        return

    visto[f][c] = True
    flfl(f + 1, c, alto)
    flfl(f - 1, c, alto)
    flfl(f, c + 1, alto)
    flfl(f, c - 1, alto)

ans = 0
for i in range(n):
    for j in range(m):
        if not visto[i][j]:
            ans += 1
            flfl(i, j, matriz[i][j])

print(ans)
```

B. FASE PRESENCIAL

1) Problema 1: Fotografiando Autovengadores:

Enunciado:

¿Estás list@?

Finalmente hemos llegado a la temida base de los **Deceptinant**. Antes de ingresar, **Hermione**, la **fotógrafa mágica** de los Autovengadores, quiere capturar una serie de imágenes de los héroes que enfrentarán al temible **Lado Oscuro de la Programación** y el **Anti-Fútbol**.

Los **Autovengadores** están organizados en N grupos, cada uno con una cantidad de miembros X. Para mantener la espontaneidad en las fotos, Hermione ha decidido **no reordenar** a los grupos.

Hermione tomó Q fotografías. Cada fotografía cubre un rango de grupos, desde el grupo L hasta el grupo R (ambos inclusive). Gracias a su magia, **todos los miembros de esos grupos aparecen perfectamente en cada imagen**. Sin embargo, ahora necesitamos saber **cuántos Autovengadores aparecen** en cada una de las fotos.

Entradas:

- La primera línea contiene un entero N, el número de grupos.
- La segunda línea contiene N enteros $X_1, X_2, \ldots, X_N$, donde X_i representa la cantidad de Autovengadores en el grupo i.
- La tercera línea contiene un entero Q, el número de fotos tomadas.
- Las siguientes Q líneas contienen dos enteros L y R, que representan el rango de grupos fotografiados.

Salida:

Imprime Q líneas. Cada una debe contener un solo número: el total de Autovengadores que aparecen en la correspondiente foto.

Restricciones:

- $1 \leq N \leq 10^5$
- $1 \leq X_i \leq 10^3$
- $1 \leq Q \leq 10^5$
- $1 \leq L \leq R \leq N$
- Los índices de los grupos comienzan en 1.

Ejemplo de Entrada:

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

Ejemplo de Salida:

```
9
7
10
```

a) Solución: La solución esperada de este problema es una aplicación clásica de la técnica 'prefix sums' (prefijos de sumas) [3]. Para ello, no es necesario ninguna observación clave, solamente es necesario realizar los siguientes pasos.

- **Almacenar los datos**, de tal forma que $suma[i] = suma[i-1] + X[i-1]$
- **Imprimir las respuestas**, $suma[r] - suma[l-1]$

Esto permite responder las Q consultas en tiempo total $O(N+Q)$.

b) Código (Python):

```
def solve():
    n = int(input())
    suma = [0]

    arr = list(map(int, input()
        .split()))

    for i in range(n):
        suma.append(arr[i] + suma[i])

    q = int(input())
    for _ in range(q):
        l, r = map(int, input()
            .split())

        print(suma[r] - suma[l - 1])

t = 1
# t = int(input())
for _ in range(t):
    solve()
```

2) Problema 2: Autovengadores Vs Darthinho: Round I:

Enunciado:

Estamos a punto de llegar a la antesala del **Cuarto de Mordor**, la oficina del mismísimo **Señor Oscuro Issem Lenol**, líder de los **Deceptinant**, **señor del Lado Oscuro de la Programación**, exponente del **Anti-Fútbol** y **ganador de la mejor sonrisa 2024** según la revista *Flor Esenita.*

Sin embargo, antes de enfrentarnos a él, debemos enfrentar a **Darthinho**, también conocido como **la tristeza del Anti-Fútbol** (o simplemente **Dinho** para los compas). Es el atacante más **impredecible** del Lado

Oscuro, y uno de los enemigos más temidos de los **Autovengadores**. Incluso **Marrochris**, nuestro Jefe de Relaciones Justicieras, fracasó al intentar neutralizarlo. De no ser por la ayuda de su inseparable amigo **Sir Colocado II**, habría sido incapaz de escapar y, por lo tanto, **expulsado de las canchas**.
Por fortuna, las mentes combinadas de **Odlanor Onaitsirc** y **Robert Polanski** han logrado un gran avance: han descubierto un **patrón en los ataques de Darthinho**.
Cada uno de los N ataques posibles que puede realizar tiene asociado un número especial X. Este número determina **cuándo puede ocurrir el ataque**: un ataque solo puede suceder en los segundos que son **factores primos** de X, además del segundo 1.

Tu misión es predecir todos los **segundos diferentes** (desde el segundo 0 en adelante) en los que **podríamos ser atacados**, basándote en los números especiales de los ataques.

Entradas:

- La primera línea contiene un entero N, el número de ataques que puede realizar Darthinho.
- La segunda línea contiene N enteros $X_1, X_2, \ldots, X_n$, los números especiales asociados a cada ataque.

Salida:
Imprime en una sola línea todos los segundos diferentes en los que podríamos recibir un ataque, en orden ascendente, separados por espacios. No incluyas segundos repetidos.

Restricciones:

- $1 \leq N \leq 2 \cdot 10^4$
- $2 \leq X_i \leq 10^4$

Ejemplo de Entrada:

```
3
6 15 10
```

Ejemplo de Salida:

```
1 2 3 5
```

a) Solución: Las **restricciones** de este problema son muy bajas, por lo que es válido analizar cada entero en el rango de $[2, 10^4]$, para obtener sus factores primos. Para evitar repeticiones, se utiliza una estructura de tipo **set** [3]; además, se debe agregar el 1.
La complejidad de este algoritmo es $O(N \cdot \sqrt{M} + K \log K)$, que es válida. Donde M es el valor más grande de X y K representa la cantidad de primos distintos encontrados.

b) Código (Python):

```
n = int(input())
numbers = list(map(int, input().split()))

primes = {1}
for x in numbers:
    i = 2
    while i * i <= x:
        if x % i == 0:
            primes.add(i)
        while x % i == 0:
            x //= i
        i += 1
    if x != 1:
        primes.add(x)

print(*sorted(primes))
```

3) Problema 3: Entrando al Cuarto de Mordor:

Enunciado:

¡Cuidado! Es normal emocionarse al ver la entrada al **cuarto de Mordor**, la oficina del mismísimo **Issem Lenoil**, líder de los **Deceptinant**, señor del Lado Oscuro de la Programación, exponente del **Anti-Fútbol** y **mejor sonrisa 2024** según la revista Flor Esenita. Sin embargo, debes recordar que antes de entrar a un lugar así, es seguro que habrá una cantidad exorbitante de trampas, especialmente en este salón de N por M casillas (ancho por alto).
Al momento de llegar a la antesala, se **activan** R **robots**. Cada robot comienza a **moverse en una dirección fija**, determinada por un valor D_i:

- 0 representa **derecha**
- 1 representa **arriba**
- 2 representa **izquierda**
- 3 representa **abajo**

Además, gracias a la **Astigmiopía Catarática Biogenética Congénita** de **Sir Colocado II**, la *Vanguardia Autovengadora*, hemos sido capaces de detectar lo siguiente:

- Cada robot puede **detectar** a cualquier intruso **Autovengador si pasa por su casilla**.
- Al pasar por una casilla, cada robot **deja un robot miniatura**, el cual posee **todas las habilidades** de un robot normal, salvo que es **inmóvil**.
- Un robot de tamaño normal **deja de moverse** si llega al **límite** del salón, si la casilla que tiene delante **ya contiene un robot**, o si ya ha avanzado durante T_i turnos.

- Si **dos o más robots** intentan pasar por la **misma casilla al mismo tiempo**, ambos **continúan** su trayecto normalmente y cada uno deja un robot miniatura en esa casilla.
- Si **Sir Colocado II** pasa por una casilla al **mismo tiempo** que un robot está en ella, no es detectado.
- Todos realizan sus acciones de forma **simultánea** en cada turno.
- En cada turno, un movimiento de una casilla a otra incrementa el tiempo en una unidad. Es decir, si se parte desde una casilla con tiempo t, se llega a la siguiente con tiempo $t + 1$.

Para desactivar las trampas y permitir el avance del equipo, solo **Sir Colocado II** puede infiltrarse (ocupando una sola casilla a la vez). En cada turno puede elegir entre **moverse a una casilla adyacente** (solo en dirección **vertical** u **horizontal**) o quedarse **esperando** en la misma. **Sir Colocado II** parte desde la casilla $(1, 1)$ y debe llegar a la casilla (N, M) para **hackear** el sistema de seguridad y **abrir** la puerta al **cuarto de Mordor**.
Como no sabemos a qué nos enfrentaremos una vez desactivemos las trampas, queremos **optimizar nuestra estrategia**. Por ello, deseamos calcular **cuántos turnos puede esperar Sir Colocado II antes de iniciar su movimiento**, sin comprometer la posibilidad de alcanzar su objetivo **sin ser detectado**. En otras palabras, ¿cuál es la **mayor cantidad de turnos** que **Sir Colocado II** puede esperar en la casilla inicial antes de comenzar a moverse, y aun así **alcanzar** con **éxito** la casilla final?
Debido a los poderes del **Lado Oscuro de la Programación**, nadie puede moverse en diagonal, únicamente de forma **vertical** u **horizontal**.

Entradas:

- La primera línea contiene 3 enteros, N, M y R las dimensiones de la antesala y la cantidad de robots
- Las siguientes R líneas contienen 4 enteros cada una, X_i, Y_i, Di y T_i, la posición en el eje 'x', en el eje 'y', la dirección en la que se moverá y la cantidad de turnos que lo hará como máximo el Robot i

Salidas: Imprime:

- **-1** si es imposible para Sir Colocado II llegar al objetivo sin ser detectado
- **Infinito** si Sir Colocado II no tiene un límite de espera para empezar a moverse
- **O**, la cantidad máxima de turnos que Sir Colocado II puede esperar y llegar a su objetivo sin ser detectado

Ejemplo 1 de Entrada:

```
4 4 2
2 2 0 3
4 3 3 2
```

Ejemplo 1 de Salida:

```
Infinito
```

Ejemplo 2 de Entrada:

```
4 4 2
2 2 0 3
4 3 1 2
```

Ejemplo 2 de Salida:

```
-1
```

Ejemplo 3 de Entrada:

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

Ejemplo 3 de Salida:

```
2
```

Limitaciones:

- $N, M \leq 500$
- $R \leq 500$
- $1 \leq X \leq N$
- $1 \leq Y \leq M$
- $0 \leq D \leq 3$
- $T \leq 500$

a) Solución: Para resolver este problema, se realizan tres pasos clave:

- **Simulación** del movimiento de los robots.
- **Busqueda Binaria sobre la solución**, en este caso, sobre el tiempo que puede esperar **Sir Colocado II**
- **Recorrido por Anchura (BFS)** en las casillas del salon para determinar si un tiempo t es valido

El objetivo de la **Simulación** consiste en marcar y visualizar el salón como una matriz 'v' con dimensiones de $N * M$. Para cada $1 \leq i \leq N$, $1 \leq j \leq M$, la casilla v[i][j] guardará **el primer instante de tiempo** t **en que esa celda fue pisada por algún robot**. Si nunca fue pisada, ese valor se mantiene como *Infinito* (un valor muy alto).
Luego se realiza una **Busqueda Binaria** sobre el valor de **T** para encontrar el máximo valor que se puede esperar antes de empezar moverse en un tiempo $O(log_2 500)$.

Por último, para cada valor obtenido en la **Busqueda Binaria**, se realiza un recorrido **BFS** para realizar la técnica **FloodFill**. En esta, se busca marcar el último nodo, determinando si T es un tiempo válido. La complejidad de esta solución es de $O(max(N, M) * R + N * M * log_2 500)$ *

b) Código (Python):

```
from collections import deque

n, m, r = map(int, input().split())
coord = [(0,1), (1,0), (0,-1), (-1,0)]

class Robot:
    def __init__(self, x, y, d, t):
        self.x = x
        self.y = y
        self.d = d
        self.t = t

robots = []
for _ in range(r):
    x, y, d, t = map(int,
                 input().split())
    robots.append(Robot(x, y, d, t))

v = [[350005 for _ in range(n+1)]
             for _ in range(m+1)]

# Simulacion de robots
time = 0
while True:
    ban = True
    for rob in robots:
        if rob.t == 0:
            continue
        if v[rob.y][rob.x] < time:
            rob.t = 0
            continue
        v[rob.y][rob.x] = time
        rob.t -= 1
        rob.y += coord[rob.d][0]
        rob.x += coord[rob.d][1]
        if rob.y > m or rob.x > n or
                rob.y < 1 or rob.x < 1:
            rob.t = 0
        if rob.t != 0:
            ban = False
    if ban:
        break
    time += 1

def posible(mid):
    vis = [[0 for _ in range(n+1)]
           for _ in range(m+1)]
    if v[1][1] < mid:
        return False
    q = deque()
    q.append(((1, 1), mid))
    vis[1][1] = 1
    while q:
        (x, y), t = q.popleft()
        if x == n and y == m:
            return True
        for dx, dy in coord:
            nx = x + dx
            ny = y + dy
            if 1 <= nx <= n
            and 1 <= ny <= m
            and not vis[ny][nx]
            and v[ny][nx] >= t + 1:
                vis[ny][nx] = 1
                q.append(((nx, ny), t + 1))
    return False

# Busqueda binaria en la solucion
l = 0
ri = 500
while l < ri:
    mid = l + (ri - l + 1) // 2
    if posible(mid):
        l = mid
    else:
        ri = mid - 1

if l == 0:
    if not posible(0):
        print(-1)
    else:
      print(l)
elif l == 500:
    if posible(505):
        print("Infinito")
    else:
      print(l)
else:
  print(l)
```

4) Problema 4: Decodificando el Lenguaje de Og-Die:

Enunciado:
Hace exactamente $2^{32} - 1$ *años...*

Og-Die era un **legendario desarrollador, investigador y maestro del buen fútbol**. Sus principales aportes a la humanidad primigenia fueron el **descubrimiento de la programación** y del **buen fútbol**, gracias a los cuales la humanidad pudo descubrir su potencial oculto y avanzar

varios bits en el futuro.
Todo marchaba bien... hasta que llegaron los **Reapers**.
Seres **nacidos del rincón más oscuro de la programación**, los Reapers tenían un único objetivo que cumplían cada 2^{32} años: **atacar cualquier civilización que comprendiera la programación y destruir toda forma de buen fútbol**, dejando a cada civilización del universo en un estado crasheado de **anti-fútbol**. La **Liga de la Programación** fue incapaz de detenerlos.
Pero **Og-Die sobrevivió**.
Se encerró en una **máquina RAM ancestral**, artefacto críptico de **ingeniería binaria pura**. Sin embargo, al despertar en el año 1004 d.C., **la RAM había sufrido un overflow**... y parte de su memoria se perdió, incluido su nombre.
Solo quedaba un símbolo $\sum$ en la máquina. Por eso se le conoce hoy como el **señor Sigma**.
Aunque su memoria está corrupta, el señor Sigma sabe que dejó un mensaje oculto con **toda la información que podría destruir a los Reapers**. Tus compañeros lo han encontrado, y ahora debes decodificarlo.
La antigua civilización de Og-Die utilizaba un lenguaje distinto. Pero las palabras registradas están **ordenadas lexicográficamente de menor a mayor** según el **alfabeto de Og-Die**, diferente del nuestro. Tu tarea es reconstruir ese alfabeto.

Tu tarea:

- Si existe un único orden válido del alfabeto, imprímelo.
- Si existen múltiples órdenes posibles, imprime `MULTIPLE`.
- Si no existe ningún orden posible coherente, imprime `IMPOSIBLE`.

Entradas:

- La primera línea contiene un entero N, la cantidad de palabras que Og-Die recuerda.
- Las siguientes N líneas contienen una palabra P cada una. Ya están ordenadas en forma lexicográfica según el alfabeto de Og-Die.

Salida:
Imprime una sola línea: el orden del alfabeto si hay uno, o `MULTIPLE` o `IMPOSIBLE` según corresponda.

Restricciones:

- $N \leq 1000$
- Cada palabra tiene a lo sumo 1000 caracteres

Ejemplo de Entrada:

```
4
eab
eda
eee
badea
```

Ejemplo de Salida:

```
adeb
```

Ejemplo de Entrada:

```
3
abc
abe
ace
```

Ejemplo de Salida:

```
MULTIPLE
```

Ejemplo de Entrada:

```
4
xyz
xzy
zxy
zzx
```

Ejemplo de Salida:

```
IMPOSIBLE
```

a) Solución: La solución esperada de este problema requiere de las siguientes observaciones clave:

- **Relaciones importantes**: Para este problema, solamente se necesitan aquellas letras que ”afecten el orden lexicográfico”. Por ejemplo, en el primer caso de prueba, en ”eab” y ”eda”, importa la relación entre ’a’ y ’d’, puesto que es la primera letra diferente entre ambas palabras.
- **Construcción de grafo**: Las letras presentes en cada caso se pueden representar como nodos. Además, las relaciones importantes entre las letras se pueden representar como aristas dirigidas en un grafo.
- **Ordenamiento Topológico**: Se aplica el **Algoritmo de Kahn** [3]. Si hay más de un nodo con ”in degree” cero (cuenta con más de una arista dirigida hacia él) en algún paso, el orden es ambiguo (respuesta: `MULTIPLE`). Si no se visitan todos los nodos, hay un ciclo (respuesta: `IMPOSIBLE`). Si no ocurre nada de lo anterior, se imprime la secuencia de letras.

Complejidad:
Sea A la cantidad total de caracteres. La complejidad es: $O(A+U+V)$, donde U es el número de caracteres únicos y V el número de aristas entre ellos. En la práctica, como $U \leq 26$, es eficiente.

b) Código (Python):

```
from collections import deque,
    defaultdict

n = int(input())
words = [input().strip() for _
    in range(n)]

alphabet = set(c for word in
    words for c in word)

edges = set()

for i in range(1, n):
    w1, w2 = words[i - 1], words[i]
    j = 0
    while j < len(w1) and j < len(w2)
        and w1[j] == w2[j]:

        j += 1
    if j < len(w1) and j < len(w2):
        edges.add((w1[j], w2[j]))
    elif len(w1) > len(w2):
        print("IMPOSIBLE")
        exit()

indegree = {c: 0 for c in alphabet}
graph = defaultdict(list)
for u, v in edges:
    indegree[v] += 1
    graph[u].append(v)

queue = deque([c for c in alphabet if
    indegree[c] == 0])
order = []
multiple = False

while queue:
    if len(queue) > 1:
        multiple = True
    u = queue.popleft()
    order.append(u)
    for v in graph[u]:
        indegree[v] -= 1
        if indegree[v] == 0:
            queue.append(v)

if len(order) != len(alphabet):
    print("IMPOSIBLE")
elif multiple:
    print("MULTIPLE")
else:
    print(''.join(order))
```

5) Problema 5: Autovengadores: Programación del Infinito (Vol. 2):

Enunciado:

¡Ha llegado el momento!

Los **Autovengadores** se enfrentan al temido **Señor Oscuro Issem Lenoil** en el **duelo final**. Aunque contamos con aliados legendarios como **Vini-El**, **Kyde Bellinippé**, y el invencible **Odlanor Onaitsirc**, Issem sigue siendo un enemigo formidable. Su dominio de la **programación oscura** y el **Anti-Fútbol** no debe subestimarse.

Para protegerse, Issem invoca el **Protegothon del Infinito**, una función mágica que bloquea **cualquier daño**, incluso los ataques directos de Odlanor.

Pero no todo está perdido.

La mejor spyware de los Autovengadores, **Dorothea**, puede infiltrarse por detrás de Issem y desactivar el Protegothon usando sus **Exploits**.

Sin embargo, necesita tu ayuda para determinar **en cuánto tiempo podría llegar**.

La arena de combate ha sido **destrozada** por los ataques de Issem, por lo que Dorothea solo puede moverse entre algunas posiciones: si salta de la posición U a la V, tarda exactamente 1 segundo, sin importar U y V.

Tu tarea es simple: determinar el **mínimo número de segundos** en que Dorothea puede llegar de la posición 1 a la N, o -1 si es imposible.

Entradas:

- La primera línea contiene dos enteros N y M, la cantidad de posiciones y la cantidad de saltos posibles.
- Las siguientes M líneas contienen dos enteros U_i, V_i, indicando que se puede saltar entre U_i y V_i.

Salida:

Imprime un solo entero: el **mínimo número de segundos** para que Dorothea llegue desde la posición 1 hasta la N, o -1 si es imposible.

Restricciones:

- $1 \leq N \leq 10^4$
- $M \leq 10^5$
- $1 \leq U_i, V_i \leq N$

Ejemplo de Entrada:

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

Ejemplo de Salida:

```
2
```

Ejemplo de Entrada:

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

Ejemplo de Salida:

```
-1
```

a) Solución: Para este problema, se espera una aplicación clásica de **BFS (Breadth-First Search)** [2] sobre un grafo no dirigido. Cada posición es un nodo, y cada salto permitido es una arista. Como cada salto toma el mismo tiempo (1 segundo), BFS nos asegura la mínima cantidad de movimientos desde el nodo 1 al nodo N.
La complejidad de este algoritmo es de $O(N + M)$.

b) Código (Python):

```
from collections import deque

n, m = map(int, input().split())
graph = [[] for _ in range(n + 1)]
seen = [-1] * (n + 1)

for _ in range(m):
    u, v = map(int, input().split())
    graph[u].append(v)
    graph[v].append(u)

q = deque()
q.append(1)
seen[1] = 0

while q:
    u = q.popleft()
    for v in graph[u]:
        if seen[v] == -1:
            seen[v] = seen[u] + 1
            q.append(v)

print(seen[n])
```

6) Problema 6: Autovengadores: Fin de la Compilación (Vol. 2):

Enunciado:

La batalla final entre los **Autovengadores** y los **Deceptinant** sigue en marcha. Todos los Autovengadores se han unido en el mismo bando en contra de los Deceptinant, luchando para que **La Tierra no caiga en el Anti-Fútbol**. El ataque final inicia con Odlanor gritando **Autovengadores Autovenguémonos**.
Con el **Protegothon del Infinito** finalmente desactivado, el malvado **Issem Lenoil** es vulnerable. Todos los héroes están lanzando su ataque definitivo: **Marrochris**, **Robert Polanski**, el **Viejo Sigma**, **Sir Colocado II**, **Dorothea**, **Hermione**, **Bellinippé**, **Vini-El**, **Odlanor**, y muchos más se encuentran en la primera línea de combate.
(*El profe Pleités* asegura que está colaborando desde una distancia de 2^8 kilómetros para evitar que cualquier inclemencia se acerque a la batalla, aunque parece estar más dormido que otra cosa).
Volviendo a la acción, **Odlanor** se prepara con un solo uso de su energía, y de forma cuidadosa **elige uno de los N Autovengadores** para lanzar un ataque de poder P, con un costo de energía E.
Teniendo en cuenta que Odlanor puede dar un total de energía X, ¿**cuál es la mayor cantidad de daño** que podemos hacer a Issem?

Entradas:

- La primera línea contiene dos enteros N, la cantidad de Autovengadores disponibles, y X, la energía total que Odlanor puede distribuir.
- Las siguientes N líneas contienen dos enteros P_i y E_i, representando el poder y el costo de energía del ataque del Autovengador i.

Salida:
Imprime un solo número entero: la **mayor cantidad de daño** que se le puede hacer a Issem sin exceder el límite de energía.

Restricciones:

- $N \leq 100$
- $X \leq 10^4$
- $1 \leq P_i, E_i \leq 10^4$

Ejemplo de Entrada:

```
3 10
10 5
7 4
5 6
```

Ejemplo de Salida:

```
17
```

a) Solución: Se esperaba que los concursantes aplicaran el clásico de programación dinámica conocido como la **Knapsack** (mochila) [2]. Es necesario seleccionar un

subconjunto de elementos (Autovengadores) que maximice el poder total, sin exceder el límite de energía X. Para ello, se define una matriz $dp[i][j]$ que representa el máximo daño que se puede lograr usando (a lo sumo) los primeros i Autovengadores y j unidades de energía. La forma de guardar los datos es: $dp[i][j] = \max(dp[i-1][j],\ dp[i-1][j-E_i] + P_i)$, siempre que $j \geq E_i$.
La complejidad de esta solución es de $O(N * X)$

b) Código (Python):

```
n, m = map(int, input().split())
items = [tuple(map(int, input().
    split()))for _ in range(n)]

dp = [[0] * (m + 1) for _ in range
    (n + 1)]

for i in range(1, n + 1):
    p, e = items[i - 1]
    for j in range(m + 1):
        dp[i][j] = dp[i - 1][j]
        if j >= e:
            dp[i][j] = max(dp[i][j],
                dp[i - 1][j - e] + p)

print(dp[n][m])
```

V. Resultados y Análisis

A. Fase Virtual

1) Veredictos: La Figura 1 muestra la distribución de los veredictos generados por el juez automático de la plataforma OmegaUp durante la fase virtual. Se contabilizaron un total de 179 envíos en la competencia.

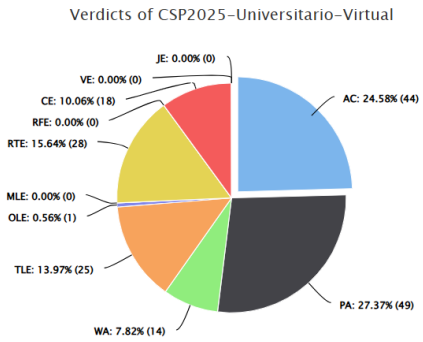

Fig. 1. Distribución de Veredictos - Fase Virtual

Se observa que:

- El 24.58% de los envíos fueron **Accepted (AC)**, lo que indica una tasa moderada de éxito en la solución de los problemas.
- El 27.37% recibió el veredicto **PA (Partially Accepted)**, lo cual refleja que muchos equipos lograron aproximarse a la solución o resolver subtareas.
- Los errores más comunes fueron **RTE (Run-Time Error)** con 15.64% y **TLE (Time Limit Exceeded)** con 13.97%, asociados a soluciones correctas pero ineficientes o con errores de ejecución.
- Un 10.06% correspondió a errores de compilación (**CE**), usualmente causados por descuidos de sintaxis.

2) Puntajes por equipos: La tabla I presenta el resumen completo de puntajes obtenidos por cada equipo. Cada problema tenía un valor máximo de 100 puntos, sumando un total de 700. Es importante aclarar que el problema 5 tenia una segunda versión adecuada a soluciones en Python.

Con estos datos, se interpreta lo siguiente:

- Seis equipos alcanzaron los 500 puntos, resolviendo correctamente al menos 5 de los 6 problemas.
- El equipo **RafaelDM** obtuvo la puntuación más alta con 594.23 puntos, seguido muy de cerca por **keyminds** y **Los_Epsilons**.
- Los problemas 1 a 4 fueron resueltos por una mayor parte de los participantes, lo que valida su correcto posicionamiento como punto de inicio.
- Prácticamente todos los equipos del top 10 resolvieron el problema 1 con éxito.

TABLE I
Resultados completos – Fase Virtual

Equipo	1	2	3	4	5	5(Py)	6	Total
RafaelDM	100.00	100.00	100.00	100.00		94.23	100.00	594.23
keyminds	100.00	100.00	100.00	100.00	94.23		77.55	571.78
Los_Epsilons	100.00	100.00	100.00	100.00		42.31	100.00	542.31
Fernando González*	100.00	100.00	100.00	100.00	3.85		100.00	503.85
ACRunners	100.00	100.00	100.00	100.00			100.00	500.00
Mario Martinez	100.00	100.00	100.00	100.00	0.00		100.00	500.00
ChristianRL	100.00	100.00	100.00	100.00	30.77			430.77
JR_2005	98.00	100.00	90.20	100.00		1.92	2.04	392.16
Liss M	100.00	2.00	78.43	100.00			97.96	378.39
JartigaRox	100.00	100.00		1.92			14.29	216.21
Bit_Masters_	98.00	100.00		15.38				213.38
Pamtenorio26	100.00	0.00		100.00				200.00
Armando	100.00	4.00		5.77		0.00		109.77
Alejandro Durón	100.00							100.00
Jason-Lopez	0.00			3.85				3.85
404								0.00
AlexAscencio								0.00
arliebolanoss	0.00							0.00
Cosmic_Ray_Believers								0.00
Carlos Castillo								0.00
Diego789Jr								0.00
Jade_Nic	0.00			0.00				0.00
Sofía Pocasangre								0.00
squesadab								0.00

- Los problemas 5 y 6 fueron resueltos parcialmente únicamente por los equipos más preparados. Esto nos muestra cómo estos problemas funcionaron como filtros técnicos de nivel "plata-avanzado" de USACO.
- Se observa una amplia dispersión: algunos equipos alcanzaron puntajes superiores a 400, mientras que otros apenas lograron sumar puntos.

3) Problemas resueltos: La Tabla II resume la cantidad de equipos que lograron resolver al 100% cada uno de los problemas propuestos.

TABLE II
Resumen de Problemas Resueltos – Fase Virtual

Problema	Equipos que lo resolvieron
1	12
2	10
3	7
4	10
5	0
5 (Python)	0
6	5

Los problemas 1, 2 y 4 fueron resueltos por la mayoría de equipos, lo cual indica que funcionaron bien como entrada accesible. En contraste, los problemas 5 y 6 evidencian una dificultad significativamente mayor, y sirvieron como filtro técnico para los equipos mejor preparados. En el caso del 5, ningún equipo pudo lograr la solución completa, lo que lo posiciona como el problema de mayor dificultad en la ronda.

B. Fase Presencial

Tras la clasificación virtual, los mejores equipos fueron convocados a la gran final presencial, celebrada en las instalaciones de la ESEN. Esta fase mantuvo el mismo formato establecido.

1) Veredictos: La Figura 2 presenta la distribución de veredictos generados por la plataforma durante esta etapa. Se registraron 152 envíos en total.

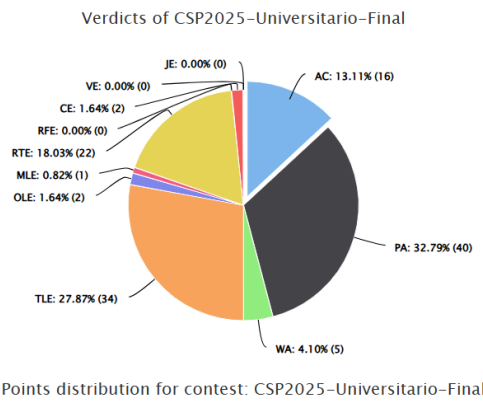

Fig. 2. Distribución de Veredictos - Fase Presencial

Se observa que:

- El porcentaje de envíos aceptados (**AC**) fue del 13.11%, menor al de la fase virtual, lo que refleja la mayor dificultad de los problemas.
- Se observó un ligero aumento en los veredictos **RTE (15.13%)** y **CE (11.18%)**, posiblemente debido al estrés de la competencia en tiempo real.
- Existe un gran aumento en los resultados **TLE (27.87%)**. Lo que representa que las soluciones requerian algoritmos de menor complejidad.
- La tasa de **PA** fue del 32.79%, lo que indica que muchos equipos lograron avances parciales en los problemas. Tambien representa un gran aumento en soluciones parciales desde la fase virtual.

2) Puntajes por equipos: La Tabla III muestra los puntajes completos obtenidos por los equipos finalistas. A diferencia de la fase virtual, la distribución es más dispersa y con resultados más bajos en general.

TABLE III
RESULTADOS COMPLETOS – FASE PRESENCIAL

Equipo	1	2	3	3(Py)	4	5	6	Total
xMakuno	100	31.37			5.66	100	100	337.03
thatsgonzalez	100	100	1.89			100	1.96	303.85
ACRunners	100	1.96				100	100	301.96
Los_Epsilons	100	100			15.38	7.84		223.22
keyminds	100	100	3.77		5.66	5.77	7.84	223.04
RafaelDM	100	1.96					11.76	113.72
Bit_Masters_	100	1.96					11.76	113.72
Pamtenorio26	100						0	100.00
Arman_Salgado	19.61	0			40.38	29.41		89.40
ChristianRL	21.57	1.96						23.53
Jade_Nic	3.92							3.92
JR_2005	1.96							1.96
Ale_duron13	0							0.00
arliebolanoss	0	0						0.00
Jason-Lopez								0.00

Análisis:

- Solo tres equipos superaron los 300 puntos: **xMakuno**, **thatsgonzalez** (Participante no oficial) y **ACRunners**.
- El promedio general fue considerablemente más bajo que en la fase virtual, lo que valida la mayor dificultad de los problemas y la presión del entorno presencial.
- Se evidencia que muchos equipos enfocaron su estrategia en resolver uno o dos problemas completamente y luego intentar sumar puntos parciales.
- Algunos equipos participaron sin sumar puntos, lo que refleja el nivel de exigencia de la competencia.

3) Problemas resueltos: La Tabla IV detalla cuántos equipos lograron resolver completamente cada problema (puntaje igual a 100).

TABLE IV
RESUMEN DE PROBLEMAS RESUELTOS – FASE PRESENCIAL

Problema	Equipos que lo resolvieron
1	8
2	3
3	0
3 (Python)	0
4	0
5	3
6	2

Análisis:

- El problema 1 fue resuelto por la mayoría de los equipos, lo que lo convierte en un excelente "problema de entrada".
- Solo dos problemas adicionales (2 y 5) superaron las 2 soluciones perfectas.
- Los problemas 3 y 4 no fueron resueltos por ningún equipo, destacándose como los más exigentes del set.
- El problema **G**, pese a ser uno de los últimos, logró 2 soluciones completas, lo que muestra que algunos equipos lo priorizaron de forma efectiva.

VI. CONCLUSIONES

La Copa Salvadoreña de Programación 2025 demostró la viabilidad y el impacto de organizar una competencia técnica de alcance nacional siguiendo estándares internacionales como ICPC. A través del desarrollo detallado de problemas, la implementación rigurosa en plataformas de evaluación automatizada como OmegaUp, y un equipo organizador comprometido, se logró ofrecer una experiencia formativa, inclusiva y desafiante para los estudiantes.

Los resultados evidencian un nivel técnico en ascenso por parte de los estudiantes universitarios salvadoreños, así como un creciente interés institucional en fomentar este tipo de iniciativas. La alta participación, la calidad de las soluciones propuestas y la competitividad mostrada reflejan un ecosistema en desarrollo que debe ser respaldado y potenciado.

De cara a futuras ediciones, se recomienda:

- Fortalecer espacios de preparación previa para los equipos.

- Publicar lineamientos formativos que guíen la práctica de problemas.
- Consolidar alianzas con más universidades e instituciones.
- Documentar y publicar editoriales como esta para preservar el aprendizaje colectivo.

Esta investigación y editorial busca servir como un recurso técnico, pedagógico y de inspiración tanto para los participantes como para quienes deseen incursionar en el mundo de la programación competitiva desde El Salvador.

REFERENCES

[1] ICPC Foundation, "International Collegiate Programming Contest." [Online]. Available: https://icpc.global

[2] Codeforces, "Competitive Programming Contests and Editorials." [Online]. Available: https://codeforces.com

[3] USACO Guide. [Online]. Available: https://usaco.guide

[4] OmegaUp, "Online Judge Platform for Latin America." [Online]. Available: https://omegaup.com

[5] OmegaUp, "CSP2025 – Universitario Virtual." [Online]. Available: https://omegaup.com/contest/CSP2025-Universitario-Virtual/

[6] OmegaUp, "CSP2025 – Universitario Final." [Online]. Available: https://omegaup.com/arena/CSP2025-Universitario-Final/