Trabajo final TTPS

Teoría de juegos

Ariel Dapia Graziani 16056/2

Mayo 2021

Índice

Introducción

A lo largo del presente trabajo de investigación se explorará el concepto de teoría de juegos aplicado a los problemas de concursos de programación. Al ser un tema tan amplio, no es el objetivo de este trabajo el describir al detalle dicha teoría, sino más bien, dar un pantallazo general como punto de partida para posteriormente seguir investigando sobre él.

El trabajo se estructura en su comienzo con una definición sobre el tema escogido, detallando qué clases de problemas pueden aparecer en los concursos de programación, con sus respectivos ejemplos.

Se prosigue dando un pequeño contexto sobre la historia de la teoría de juegos, y sobre algunas de sus aplicaciones en la actualidad.

Luego se analizan distintas formas de encarar los problemas que refieran al tema, explicando distintas soluciones que se pueden llevar a cabo para sus resoluciones. En esta sección se define un problema especial en la teoría de juegos, el problema de Nim.

Posteriormente se detallan los procesos para solucionar 3 problemas que se han escogido sobre el tema. Cada problema tiene asociado su enunciado, su proceso de resolución, los problemas encontrados en el desarrollo, y su consecuente resolución junto al resultado recibido del juez. Los problemas que se han seleccionado para este trabajo se han escogido dado que están categorizados como “Must try” en el libro Competitive Programming 3: The New Lower Bound of Programming Contests (2013) de Halim S. y Halim F..

Se finaliza el informe con el conjunto de recursos que se utilizaron para llevarlo a cabo.

En el siguiente repositorio se pueden encontrar todos los archivos que fueron utilizados para realizar este informe.

Definición

La teoría de juegos es un modelo matemático de situaciones estratégicas (no necesariamente juegos como en el significado común de "juegos") en el que el éxito de un jugador al hacer elecciones depende de las elecciones de los demás.

Sus investigadores estudian las estrategias óptimas así como el comportamiento previsto y observado de individuos en juegos. Tipos de interacción aparentemente distintos pueden, en realidad, presentar estructuras de incentivos similares y, por lo tanto, se puede representar mil veces conjuntamente un mismo juego.

Entonces esta teoría trata de analizar, mediante un novedoso marco de referencia matemáticamente, la competencia que se produce entre dos o más sistemas racionales (o parte de un sistema) antagonista, los que buscan maximizar sus ganancias y minimizar sus pérdidas (es decir, buscan alcanzar o “jugar” la estrategia óptima). A través de esta técnica se puede estudiar el comportamiento de partes en conflicto, sean ellas individuos, oligopolios o naciones.

Muchos problemas de programación relacionados con la teoría de juegos se clasifican como juegos de suma cero (Zero-Sum Game), una forma matemática de decir que si un jugador gana, el otro jugador pierde. Por ejemplo, un juego de Ta-Te-Ti (por ejemplo, UVa 10111), Ajedrez, varios juegos de números / enteros (por ejemplo, UVa 847, 10368, 10578, 10891, 11489) y otros (UVa 10165, 10404, 11311) con dos jugadores jugando alternativamente (normalmente de forma perfecta) y donde solo puede haber un ganador. La pregunta común que se hace en los problemas de concursos de programación relacionados con la teoría de juegos es si el jugador inicial de un juego competitivo de dos jugadores tiene una jugada ganadora, asumiendo que ambos jugadores están haciendo jugadas perfectas. Es decir, cada jugador elige siempre la opción más óptima disponible para él.

Historia y aplicaciones

La primera discusión conocida de la teoría de juegos aparece en una carta escrita por James Waldegrave en 1713. En esta carta, Waldegrave proporciona una solución mínima de estrategia mixta a una versión para dos personas del juego de cartas le Her.

Sin embargo no se publicó un análisis teórico de teoría de juegos en general hasta la publicación de Recherches sur les príncipes mathématiques de la théorie des richesses, de Antoine Augustin Cournot en 1838.

Aunque el análisis de Cournot es más general que el de Waldegrave, la teoría de juegos realmente no existió como campo de estudio aparte hasta que John von Neumann publicó una serie de artículos en 1928. Estos resultados fueron ampliados más tarde en su libro de 1944, Theory of Games and Economic Behavior, escrito junto con Oskar Morgenstern. Este trabajo contiene un método para encontrar soluciones óptimas para juegos de suma cero de dos personas. Durante este período, el trabajo sobre teoría de juegos se centró, sobre todo, en teoría de juegos cooperativos. Este tipo de teoría de juegos analiza las estrategias óptimas para grupos de individuos, asumiendo que pueden establecer acuerdos entre sí acerca de las estrategias más apropiadas.

La teoría de juegos tiene aplicaciones en numerosas áreas, entre las cuales se destacan las ciencias económicas, la biología evolutiva, la psicología, las ciencias políticas, la investigación operativa, la informática y la estrategia militar.

Economía y negocios

Los economistas han usado la teoría de juegos para analizar un amplio abanico de problemas económicos, incluyendo subastas, duopolios, oligopolios, la formación de redes sociales, y sistemas de votaciones.

Biología

En biología, la teoría de juegos se emplea para entender muchos problemas diferentes. Se usó por primera vez para explicar la evolución (y estabilidad) de las proporciones de sexos 1:1 (mismo número de machos que de hembras).

Informática y lógica

La teoría de juegos ha empezado a desempeñar un papel importante en la lógica y la informática. Muchas teorías lógicas se asientan en la semántica de juegos. Además, los investigadores de informática han usado juegos para modelar programas que interactúan entre sí. Podemos aplicar la teoría de juegos en lo que respecta a las redes y comunicaciones, agentes inteligentes, machine learning, diseño de redes de energía, entre otros.

Ciencias políticas

La investigación en ciencias políticas también ha usado resultados de la teoría de juegos. Una explicación de la teoría de la paz democrática es que el debate público y abierto en la democracia envía información clara y fiable acerca de las intenciones de los gobiernos hacia otros estados.

Filosofía

La teoría de juegos ha demostrado tener muchos usos en filosofía. A partir de dos trabajos de W.V.O. Quine publicados en 1960 y 1967, David Lewis (1969) usó la teoría de juegos para desarrollar el concepto filosófico de convención.

En el campo militar

La teoría de juegos se utiliza como una definición de pensamiento estratégico como arte de vencer al adversario sabiendo que éste está tratando de hacer lo mismo con uno (supone un nivel de conflicto).

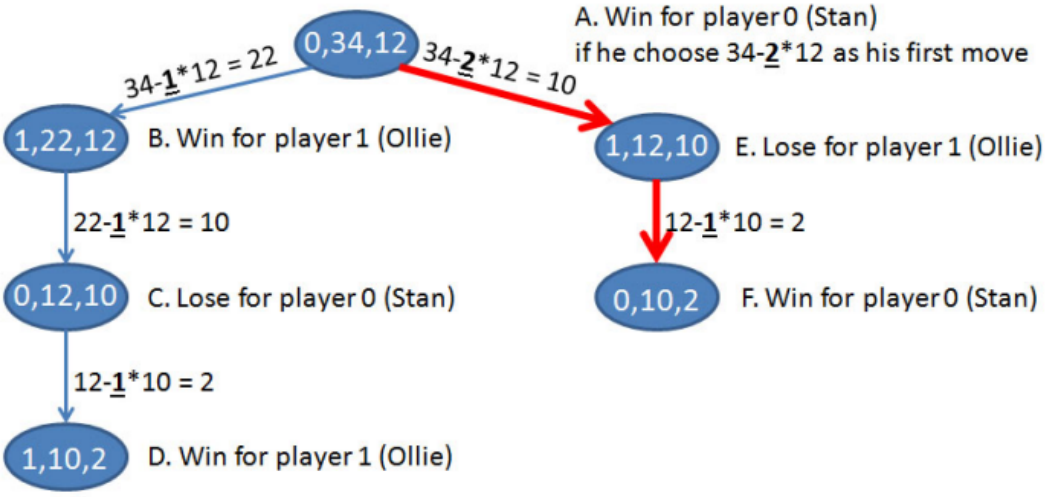
Análisis de resoluciones

Árbol de decisión

Una solución es escribir un código recursivo para explorar el árbol de decisiones del juego (también conocido como el Árbol de juego). Si no existe superposición de subproblemas, lo adecuado es utilizar backtracking recursivo de forma pura. De lo contrario, se necesita hacer uso de programación dinámica. Cada vértice describe el jugador actual y el estado actual del juego. Cada vértice está conectado a todos los demás vértices legalmente accesibles desde ese vértice de acuerdo con las reglas del juego. El vértice raíz describe el jugador inicial y el estado inicial del juego. Si el estado del juego en un vértice de la hoja es un estado ganador, es una victoria para el jugador actual (y una pérdida para el otro jugador). En un vértice interno, el jugador actual elige un vértice que garantiza una victoria con el mayor margen (o si no es posible una victoria, elige un vértice con la menor pérdida). A esto se le llama la estrategia Minimax.

Ejemplo

En UVa 10368 - Euclid's Game, hay dos jugadores: Stan (jugador 0) y Ollie (jugador 1). El estado del juego es una terna de números enteros (id, a, b). El id del jugador actual puede restar cualquier múltiplo positivo del menor de los dos números (b), del mayor de los dos números (a), siempre que el número resultante no sea negativo. Siempre mantenemos que a ≥ b. Stan y Ollie juegan alternativamente, hasta que un jugador es capaz de restar un múltiplo del número menor al número mayor para llegar a 0 y, por lo tanto, gana. El primer jugador es Stan. El árbol de decisiones para un juego con id de estado inicial = 0, a = 34 y b = 12 se muestra a continuación:



Veamos en detalle lo que sucede en el ejemplo graficado. En la raíz tenemos la terna (0, 34, 12). En este punto, el jugador 0 (Stan) tiene dos opciones: restar a - b = 34 - 12 = 22 y moverse al vértice (1, 22, 12) (la rama izquierda) o restar a - 2 × b = 34 - 2 × 12 = 10 y moverse al vértice (1, 12, 10) (la rama derecha). Sigamos ambas opciones de forma recursiva.

Comenzando con la rama izquierda, en el vértice (1, 22, 12) (B), el jugador actual 1 (Ollie) no tiene más remedio que restar a − b = 22 − 12 = 10. Ahora nos posicionamos en el vértice (0, 12, 10) (C). Nuevamente, Stan solo tiene una opción que es restar a - b = 12 - 10 = 2. En consecuencia, nos posicionamos en el vértice de la hoja (1, 10, 2) (D). Ollie puede definitivamente ganar realizando a - 5 × b = 10 - 5 × 2 = 0, esto implica que el vértice (0, 12, 10) es un estado perdedor para Stan, y en consecuencia, el vértice (1, 22, 12) es un estado ganador para Ollie.

Explorando la rama derecha, en el vértice (1, 12, 10) (E), el jugador 1 (Ollie) no tiene más remedio que restar a - b = 12 - 10 = 2. Ahora estamos en la hoja vértice (0, 10, 2) (F). Stan tiene varias opciones, pero Stan definitivamente puede ganar realizando a - 5 × b = 10 - 5 × 2 = 0, implicando que el vértice (1, 12, 10) es un estado perdedor para Ollie.

Por lo tanto, para que el jugador 0 (Stan) gane este juego, Stan debe elegir a − 2 × b = 34−2 × 12 primero, ya que este es un movimiento ganador para Stan (A).

En cuanto a la implementación, el primer ID entero en la terna se puede eliminar ya que sabemos que las alturas pares siempre son turnos de Stan, y las impares siempre son turnos de Ollie. Este id se usa en la imagen para simplificar la explicación.

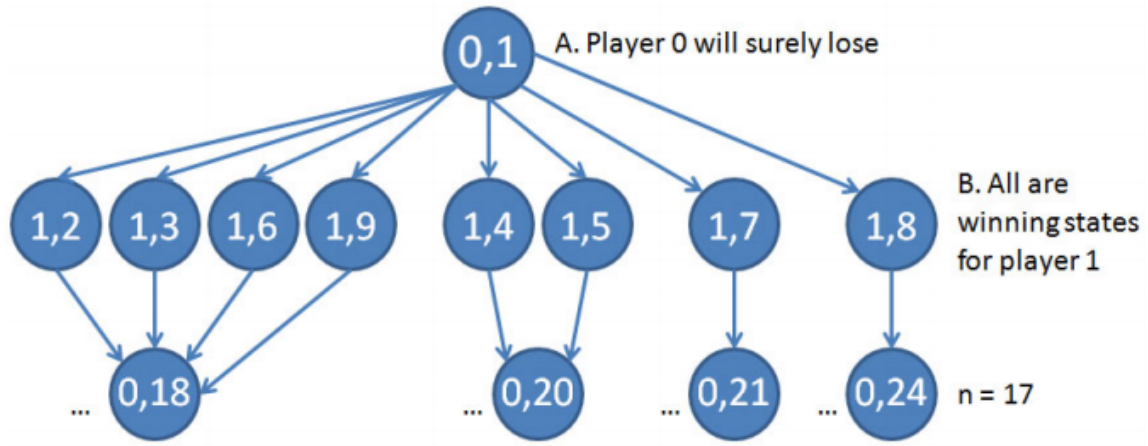
Optimización de la solución mediante herramientas matemáticas

No todos los problemas de la teoría de juegos se pueden resolver explorando todo el árbol de decisiones del juego, especialmente si el tamaño del árbol es grande. Si el problema tiene que ver con números, es posible que necesitemos involucrar algunos conocimientos matemáticos para acelerar el cálculo.

Ejemplo

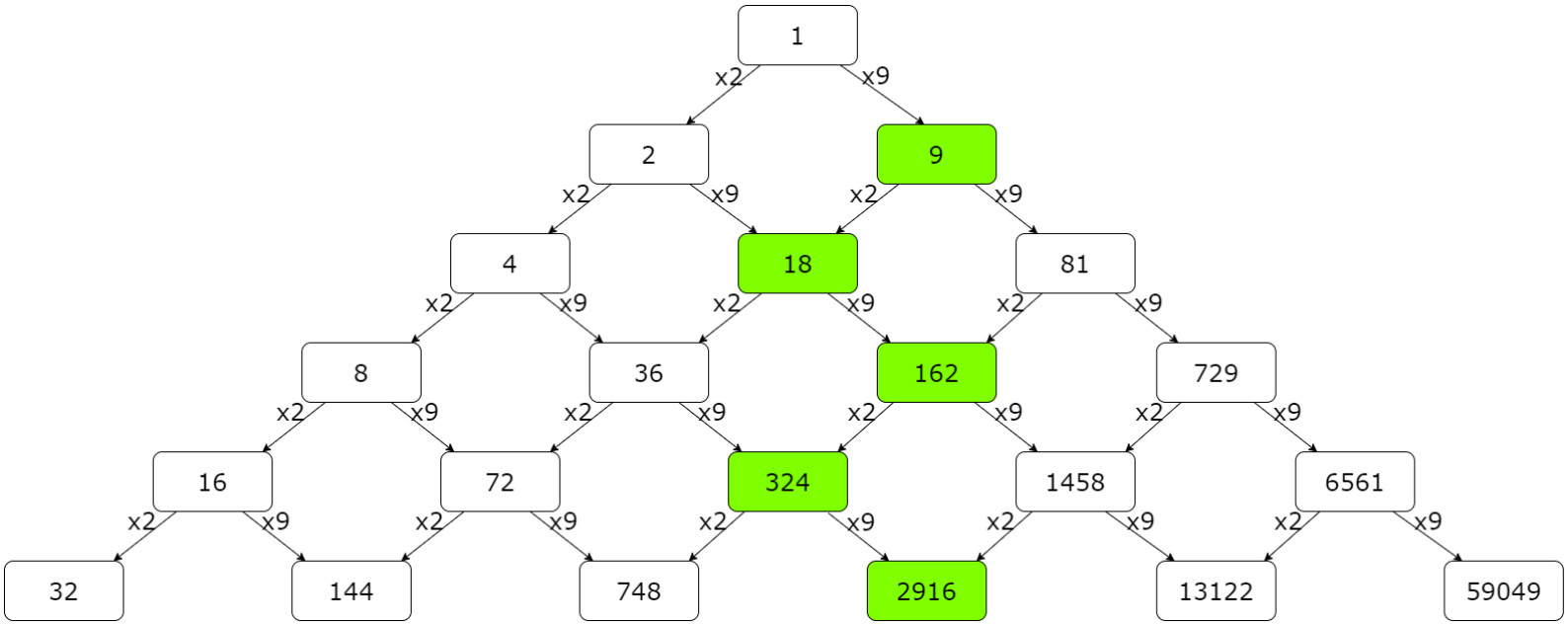
Por ejemplo, en UVa 847 - Un juego de multiplicación, hay dos jugadores: Stan (jugador 0) y Ollie (jugador 1) nuevamente. El estado del juego es un entero p. El jugador actual puede multiplicar p con cualquier número entre 2 y 9. Stan y Ollie también juegan alternativamente, hasta que un jugador sea capaz de multiplicar p con un número entre 2 y 9 de modo que p ≥ n (n es el número objetivo), por lo tanto, gana. El primer jugador es Stan con p = 1.

La siguiente figura muestra una instancia del juego con n = 17. Inicialmente, el jugador 0 tiene hasta 8 opciones (para multiplicar p = 1 por [2..9]). Sin embargo, todos estos 8 estados son estados ganadores del jugador 1, ya que el jugador 1 siempre puede multiplicar el p actual por [2..9] para hacer p ≥ 17 (B). Por lo tanto, el jugador 0 seguramente perderá (A).



Como 1 < n < 4294967295, el árbol de decisión resultante en el caso de prueba más grande puede ser extremadamente grande. Esto se debe a que cada vértice en este árbol de decisión tiene un factor de ramificación de 8 (ya que hay 8 números posibles para elegir entre 2 y 9). No es factible explorar realmente el árbol de decisiones.

Resulta que la estrategia óptima para que Stan gane es multiplicar siempre p por 9 (el más grande posible) mientras que Ollie siempre multiplicará p por 2 (el más pequeño posible). Estos conocimientos de optimización se pueden obtener observando el patrón encontrado en la salida de instancias más pequeñas de este problema:



Solución

```
from sys import stdin, stdout


def stan_wins(n):
    stan_turn = True
    p = 1
    while (p < n):
        if stan_turn:
            p *= 9
        else:
            p *= 2
        stan_turn = not stan_turn
    return not stan_turn


def main():
    numbers = [int(x) for x in stdin.readlines()]
    for number in numbers:
        if stan_wins(number) == True:
            stdout.write("Stan wins.\n")
        else:
            stdout.write("Ollie wins.\n")


if __name__ == '__main__':
    main()
```

#31041868 | arieldg1997's solution for [UVA-847]

Status	Time	Length	Lang	Submitted	Shared	RemoteRunId
Accepted	10ms	498	PYTH3 3.5.1	2021-05-15 22:56:59	☐	26404518

Optimización del juego de Euclides

Otro problema que se puede optimizar mediante artificios matemáticos es el ya mencionado “Juego de Euclides”. Para ello se hace uso del teorema del cociente y del residuo:

Dado cualquier entero A y un entero positivo B, existen dos enteros únicos Q y R tales que:

$$A = B * Q + R \text{ donde } 0 \leq R < B$$

Este algoritmo optimizado, determina que gana el juego quien consiga una instancia del problema *(m,n)* tal que *(n mod m)* sea 0 ó *(n div m)* sea estrictamente mayor a 1.

Prueba:

- El par inicial (m,n), donde $0<m\leq n$ puede ser reescrito como *(m, q.m + r)* donde $q\geq 0$ y $0\leq r\leq m-1$ (se utiliza el teorema del cociente y del residuo con A = *n* y B = *m*)
- Se pueden dar dos casos:
    - r = 0: el juego termina, ya que el jugador puede restar inmediatamente *q.m* de *n* para que el próximo número sea 0 y ganar.
    - *r > 0 y q > 1:* entonces el próximo par es *(m,(q-k).m+r),* donde $1 \leq k \leq q$. Si se elige un $k < q$, entonces $(q-k).m+r > m$ y el menor número en el próximo par será *m.* Entonces está garantizado que el próximo resto de dividir *(q-k).m+r* por *m* es $r > 0$. En consecuencia, está garantizado que el otro jugador no ganará en la próxima jugada.

Solución

```
from sys import stdin, stdout


def stan_wins(n, m):
    stan_turn = True
    while(n != 0 and m != 0):
        # Aseguro que n sea mayor o igual a m.
        if (m > n):
            n, m = m, n
        if (n//m > 1 or n % m == 0):
            break
        else:
            n %= m
            stan_turn = not stan_turn
    return stan_turn
```

```
def main():
    numbers = [int(x) for x in stdin.readline().strip().split()]
    while numbers[0] != 0 and numbers[1] != 0:
        if stan_wins(numbers[0], numbers[1]) == True:
            stdout.write("Stan wins\n")
        else:
            stdout.write("Ollie wins\n")
        numbers = [int(x) for x in stdin.readline().strip().split()]


if __name__ == '__main__':
    main()
```

#31084062 | arieldg1997's solution for [UVA-10368]

Status	Time	Length	Lang	Submitted	Shared	RemoteRunId
Accepted	10ms	678	PYTH3 3.5.1	2021-05-18 23:48:29	☐	26413367

Juego de Nim

El juego de Nim es un problema especial que suele aparecer en los concursos de programación. En el juego Nim, dos jugadores se turnan para eliminar objetos de distintos montones (o pilas). En cada turno, un jugador debe eliminar al menos un objeto y puede eliminar cualquier número de objetos siempre que todos provengan del mismo montón. El estado inicial del juego es el número de objetos n_i en cada uno de los k montones: $\{n_1, n_2, ..., n_k\}$. Hay una buena solución para este juego. Para que gane el primer jugador (inicial), el valor de $n_1 \wedge n_2 \wedge ... \wedge n_k$ debe ser distinto de cero donde $\wedge$ es el operador de bit xor.

En el siguiente enlace se puede jugar al juego de Nim en el navegador.

Otros ejercicios

UVa 10111 - Find the Winning Move

En este ejercicio se presenta un juego de Ta-Te-Ti de tamaño 4x4, donde dos jugadores “x” y “o”, deben jugar alternativamente con “x” comenzando siempre. Gana el jugador que consiga alinear cuatro de sus piezas en la misma fila, columna o diagonal. Si el tablero está lleno y ningún jugador ganó, entonces se considera un empate. Asumiendo que es el turno de “x” de jugar, se dice que “x” tiene una jugada ganadora si “x” puede hacer tal movimiento que no importa que movimientos realizará “o” en el resto del juego, “x” puede ganar. Esto no implica que necesariamente “x” gane en exactamente la siguiente jugada, aunque es una posibilidad. Esto significa que “x” tiene una estrategia ganadora que garantiza una victoria eventual sin importar que hace “o”. El objetivo es escribir un programa que, dado un juego parcialmente completo donde le toca mover a “x”, determine si “x” tiene una jugada ganadora. Se asume que cada jugador ha realizado al menos 2 movimientos, y que todavía ninguno ha ganado el juego, y además el tablero no está lleno.

El input contiene uno o más casos de prueba, seguidos de una línea que comienza con un “$” que señala el final del input. Cada caso de prueba comienza con una línea que contiene un “?” y se sigue por cuatro líneas que representan el tablero; el formato es exactamente como se muestra en el ejemplo. Los caracteres usados en una descripción de tablero son el punto (que representa un espacio vacío), “x” minúscula y “o” minúscula.

Para cada caso de prueba, se debe generar una línea que contenga la posición (fila, columna) de la primera victoria forzada para “x”, o "#####" si no hay una victoria forzada.

Para este problema, la primera victoria forzada se determina por la posición del tablero, no por el número de movimientos requeridos para ganar. Se debe buscar una jugada ganadora examinando las posiciones (0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), ..., (3, 2), (3, 3), en ese orden, e imprimir la primera posición que genere una jugada ganadora. Observando el segundo caso de prueba a continuación, se observa que “x” podría ganar inmediatamente si juega en (0, 3) o (2, 0), pero si juega en (0, 1) seguirá asegurando la victoria (aunque la retrasa innecesariamente), y la posición (0, 1) es lo primera en evaluarse.

Caso de ejemplo:

Input de ejemplo

```
?
....
.xo.
.ox.
```

....
?
o...
.ox.
.xxx
xooo
$

Output de ejemplo

#####
(0,1)

En primera instancia, se pensó resolver este ejercicio mediante un árbol de decisión. Al notar que existía superposición de subproblemas se hizo uso de programación dinámica. Es interesante ver cómo, con un diccionario donde las claves son codificaciones de instancias de juego a string, se pueden almacenar resultados parciales para llevar a cabo la resolución total. El valor que guarda cada clave del diccionario es un número entero (10, -10 o 0) dependiendo si siempre termina ganando “x” (10), o siempre termina ganando “o” (-10), o siempre hay empate (0).

Se hizo uso de la estrategia minimax donde “x” cumple el rol de maximizador, y “o” de minimizador. En consecuencia, “x” busca al menos una jugada que le de la victoria asegurada, mientras que “o” busca que en ninguna de sus jugadas “x” pueda resultar ganador. Siempre pensando que ambos toman constantemente decisiones perfectas, o al menos, las más óptimas.

Solución

```
from sys import stdin, stdout

mem = {}

# La siguiente funcion comprueba si dado un tablero de juego existen
# movimientos por realizarse
def is_moves_left(gameboard):
    for i in range(4):
        for j in range(4):
            if (gameboard[i][j] == '.'):
                return True
    return False
```

```
# La siguiente funcion evalua si dado un tablero existe un ganador
def evaluate(b):
    for i in range(4):
        if (b[i][0] == b[i][1] == b[i][2] == b[i][3]):
            if (b[i][0] == 'x'):
                return 10
            elif (b[i][0] == 'o'):
                return -10

        if (b[0][i] == b[1][i] == b[2][i] == b[3][i]):
            if (b[0][i] == 'x'):
                return 10
            elif (b[0][i] == 'o'):
                return -10

    if (b[0][0] == b[1][1] == b[2][2] == b[3][3]):
        if (b[0][0] == 'x'):
            return 10
        elif (b[0][0] == 'o'):
            return -10

    if (b[0][3] == b[1][2] == b[2][1] == b[3][0]):
        if (b[0][3] == 'x'):
            return 10
        elif (b[0][3] == 'o'):
            return -10

    return 0

# La siguiente funcion utiliza recursion junto a la estrategia
# minimiax para explorar el arbol de juego.
def minimax(board, isMax):
    # Se codifica la instancia de juego a string
    str_board = str(board)
    # Se analiza si se tiene almacenado el resultado del juego.
    if str_board in mem.keys():
        return mem[str_board]
    # Se analiza si hay ganador
```

```
    score = evaluate(board)

    if (score == 10):
        mem[str_board] = score
        return score

    if (score == -10):
        mem[str_board] = score
        return score

    # Se analiza si se termino el juego por tablero lleno.
    if (is_moves_left(board) == False):
        mem[str_board] = score
        return 0

    # Se divide la ejecucion del programa en 2 dependiendo de si
juega “x” u “o”.
    if (isMax):
        for i in range(4):
            for j in range(4):
                if (board[i][j] == '.'):
                    board[i][j] = 'x'
                    branch_res = minimax(board, not isMax)
                    board[i][j] = '.'
                    # Si existe alguna jugada que asegure la victoria
sera la que se escoja, por lo tanto se retorna con exito.
                    if branch_res == 10:
                        mem[str_board] = branch_res
                        return branch_res
        ret = -10
    else:
        for i in range(4):
            for j in range(4):
                if (board[i][j] == '.'):
                    board[i][j] = 'o'
                    branch_res = minimax(board, not isMax)
                    board[i][j] = '.'
                    # Si existe alguna jugada que asegure que x no
```

```
termine victorioso, sera la que se escoja, por lo tanto se retorna
con un fracaso para x.
                    if branch_res < 10:
                        mem[str_board] = branch_res
                        return branch_res
        ret = 10
    mem[str_board] = ret
    return ret


# La siguiente funcion se encarga de comenzar la recursion y retornar
la respuesta al problema con el formato adecuado.
def findBestMove(board):
    for i in range(4):
        for j in range(4):
            if (board[i][j] == '.'):
                board[i][j] = 'x'
                moveVal = minimax(board, False)
                board[i][j] = '.'
                if (moveVal > 0):
                    return "("+str(i)+","+str(j)+")\n"

    return "#####\n"


def main():
    while (stdin.readline().strip() != '$'):
        gameboard = [[], [], [], []]
        for i in range(4):
            row = stdin.readline().strip()
            for j in row:
                gameboard[i].append(j)
        stdout.write(findBestMove(gameboard))


if __name__ == '__main__':
    main()
```

#31114105 | arieldg1997's solution for [UVA-10111]

Status	Time	Length	Lang	Submitted	Shared	RemoteRunId
Accepted	320ms	2817	PYTH3 3.5.1	2021-05-20 17:11:42	☐	26419062

Problemas durante la resolución:

En el desarrollo de la solución, se encontró un obstáculo al intentar retornar de la función recursiva minimax, habiendo modificado el tablero recibido por parámetro sin deshacer dicha modificación. O sea:

```
    if (isMax):
        for i in range(4):
            for j in range(4):
                if (board[i][j] == '.'):
                    board[i][j] = 'x'
                    branch_res = minimax(board, not isMax)
                    if branch_res == 10:
                        mem[str_board] = branch_res
                        return branch_res
                    board[i][j] = '.'
        ret = -10
    else:
        for i in range(4):
            for j in range(4):
                if (board[i][j] == '.'):
                    board[i][j] = 'o'
                    branch_res = minimax(board, not isMax)
                    if branch_res < 10:
                        mem[str_board] = branch_res
                        return branch_res
```

```
board[i][j] = '.'
```

En python, los tipos simples de datos (int, float, string, bool, etc) se pasan por valor, y los tipos compuestos de datos (listas, diccionarios, objetos, etc) se pasan por referencia. Por ende, modificar, dentro de la función, la lista que se recibe como parámetro, refleja los cambios en la misma lista que se envió como parámetro en la invocación de dicha función. Lo correcto en este caso, es previamente a retornar de la función, deshacer las modificaciones realizadas en el tablero de juego.

UVa 11311 - Exclusively Edible

En este ejercicio se presenta una gran torta, compuesta de tortas, representada por una matriz de *mxn*, donde cada uno de sus elementos es una torta distinta al resto. Hansel y Gretel quieren dividirse este pastel, el problema es que existe una torta en una posición dada que ambos desean evitar. Así que idearon un plan para decidir quién se queda con la torta fea: primero Hansel corta una porción del pastel a lo largo de las líneas, luego Gretel hace lo mismo y siguen alternando hasta que solo queda el trozo de pastel malo y uno de ellos se ve obligado a tomarlo.

Se debe determinar, dada una torta y la posición de la torta a evitar, quién se quedará con la torta fea si ambos eligen de forma perfecta sus porciones.

La primera línea del input contiene un número *t (1 ≤ t ≤ 100)*, el número de casos de prueba. Luego siguen *t* líneas, cada una de las cuales contiene *m n r c* (separadas por espacios) donde *m* y *n (2 ≤ m, n ≤ 48)* son el ancho y la longitud del gran pastel y *(r, c)* es la posición de la torta a evitar *(0 ≤ r ≤ m - 1, 0 ≤ c ≤ n - 1)*.

Para cada caso de prueba, se imprime el nombre de la persona que obtiene la torta fea, asumiendo que Hansel hace el primer corte y que Hansel y Gretel siempre cortan el pastel en una ubicación óptima (tratando de no obtener el pedazo de pastel feo). Teniendo en cuenta que "cortar" se refiere a un corte en línea recta (a lo largo de una línea de cuadrícula) que separa el pastel en dos partes.

Caso de ejemplo:

Input de ejemplo

```
2
2 3 0 2
11 11 5 5
```

Output de ejemplo

Gretel
Hansel

La implementación de este ejercicio es bastante sencilla si utilizamos lo aprendido sobre el juego de Nim. Cada jugador, tiene a lo sumo 4 posibles elecciones:

- Realizar un corte horizontal por arriba de la torta fea.
- Realizar un corte horizontal por debajo de la torta fea.
- Realizar un corte vertical por izquierda de la torta fea.
- Realizar un corte vertical por derecha de la torta fea.

Podemos reducirlo a un juego de Nim con 4 montones, donde la cantidad de elementos que cada montón contiene equivalga a la cantidad de cortes que Hansel y Gretel pueden realizar horizontalmente por arriba, horizontalmente por debajo, verticalmente por izquierda, y verticalmente por derecha, de la torta a evitar.

Python brinda el operador ^ que realiza la operación XOR entre la representación binaria de números enteros.

Solución

```
from sys import stdin, stdout


def solve(n, m, r, c):
    left = c
    below = n-r-1
    right = m-c-1
    above = r
    nim_number = left ^ below ^ right ^ above
    return nim_number == 0


def main():
    t = int(stdin.readline().strip())
    for _ in range(t):
        n, m, r, c = [int(x) for x in
stdin.readline().strip().split()]
        if solve(n, m, r, c):
```

```
            stdout.write("Hansel\n")
        else:
            stdout.write("Gretel\n")


if __name__ == '__main__':
    main()
```

#31114194 | arieldg1997's solution for [UVA-11311]

Status	Length	Lang	Submitted	Shared	RemoteRunId
Accepted	496	PYTH3 3.5.1	2021-05-20 17:40:05	☐	26419123

Se recomienda investigar sobre el Teorema de Sprague-Grundy.

UVa 11389 - Integer Game

En este ejercicio se presenta un juego en el que dos jugadores, S y T, realizan movimiento alternativamente. S siempre comienza. En este juego, se comienza con un número entero N. En cada movimiento, un jugador quita un dígito del número entero, y pasa el número resultante al otro jugador. El juego continúa de esta forma hasta que un jugador no encuentra un dígito a remover, de esta forma se declara perdedor.

No solo eso, también se aplica una restricción adicional. Un jugador puede eliminar un dígito en particular si la suma de dígitos del número resultante es un múltiplo de 3 o si no quedan dígitos. El objetivo del ejercicio es determinar quién gana cuando ambos jugadores juegan de forma perfecta.

La primera línea de entrada es un número entero T (T < 60) que determina el número de casos de prueba. Cada caso es una línea que contiene un número entero positivo N. N tiene como máximo 1000 dígitos y no contiene ceros.

Para cada caso, se debe imprimir el número de caso empezando por 1. Si S gana, entonces se imprime “S”; de lo contrario, se imprime “T”.

Caso de ejemplo:

Input de ejemplo

```
3
4
33
771
```

Output de ejemplo

```
Case 1: S
Case 2: T
Case 3: T
```

En este problema se ideó una solución que involucra un truco matemático. El truco consiste en contar las cantidades de cifras del número original tales que sus restos de dividir por 3 sean 0, 1 y 2.

Al comenzar el juego tenemos se recibe un número N con 3 posibilidades:

1. El resto de dividir por 3 es 0.
2. El resto de dividir por 3 es 1.
3. El resto de dividir por 3 es 2.

Para cada caso inicial, se tienen las siguientes alternativas:

1. Solo se pueden remover cifras tales que su resto de dividirlas por 3 sea 0 (si es que existe alguna, caso contrario pierde S y gana T). Y esto se repite constantemente hasta que no queden cifras con resto 0. Por lo tanto, si la cantidad de cifras con resto 0 es impar, ganará S; caso contrario, ganará T.
2. Se debe remover inicialmente una cifra tal que su resto al dividirse por 3 sea 1 (si es que existe alguna, caso contrario pierde S y gana T). Y luego, se removerá en intentos sucesivos cifras tales que su resto sean 0. Si hay un número par de cifras con resto 0, entonces ganará S; caso contrario ganará T.
3. Se debe remover inicialmente una cifra tal que su resto al dividirse por 3 sea 2 (si es que existe alguna, caso contrario pierde S y gana T). Y luego, se removerá en intentos sucesivos cifras tales que su resto sean 0. Si hay un número par de cifras con resto 0, entonces ganará S; caso contrario ganará T.

La función solve(n) se encarga de analizar a n, de acuerdo a los criterios definidos anteriormente.

Solución

```
from sys import stdin, stdout


def solve(n):
    # Se define una lista que almacena en la posicion i la cantidad
de cifras tales que su resto al dividir por 3 se i.
    modules = [0, 0, 0]
    # Se recorre la cadena de entrada y se cuentan sus cifras
dependiendo del valor de sus restos.
    str_n = str(n)
    for char in str_n:
        modules[int(char) % 3] += 1
    if modules[n % 3] != 0:
        modules[n % 3] -= 1
        return modules[0] % 2 == 0
    else:
        return False


def main():
    t = int(stdin.readline().strip())
    for t in range(t):
        n = int(stdin.readline().strip())
        if solve(n):
            stdout.write("Case "+str(t+1)+": S\n")
        else:
            stdout.write("Case "+str(t+1)+": T\n")


if __name__ == '__main__':
    main()
```

#31114673 | arieldg1997's solution for [UVA-11489]

Status	Length	Lang	Submitted	Shared	RemoteRunId
Accepted	564	PYTH3 3.5.1	2021-05-20 21:55:32	☐	26419353

Recursos utilizados

Adrián Paenza - Grandes temas de la matemática: Capítulo 6: Teoría de los juegos

Competitive Programming 3: The New Lower Bound of Programming Contests (2013) de Halim S. y Halim F..

Khan Academy - El teorema del cociente y del residuo

Wikipedia - Teoría de Juegos

Geeks for Geeks - Minimax algorithm