Gran Premio de México 2020 - Repechaje

13 de Febrero de 2021

Libro de soluciones

Este documento contiene las soluciones esperadas para los 14 problemas usados durante la competencia.

Las siguientes personas apoyaron desarrollando el set de problemas ya sea creando o mejorando enunciados, soluciones, casos de prueba, verificadores de entradas y salidas:

Lina Rosales

Saraí Ramírez

Tania Martínez Villagómez

Eddy Ramírez

Esteban Artavia

Orlando Isay Mendoza

Abraham Macias

Roberto Solís

Moises Osorio

Moroni Silverio

Juan Pablo Marín

Atsa's Checkers Board

Observemos que la clave para resolver el problema se encuentra en la primera columna. Si la primera columna contiene al menos dos fichas consecutivas del mismo color, el resto del tablero se encontrará definido. De lo contrario, si no hay dos fichas consecutivas del mismo color, significa que la siguiente columna tampoco deberá contener fichas consecutivas que compartan color.

Veamos que para el primer caso existen 2^N–2 cadenas binarias que tienen al menos dos elementos consecutivos iguales, mientras que, para el segundo caso, cada columna tiene dos opciones (las dos cadenas binarias sin elementos consecutivos iguales), dándonos un total de 2^M posibles acomodos. De esta manera se concluye que existen $2^N + 2^M$–2 maneras en que se pueden acomodar las piezas de un tablero de $N * M$ para lograr el cometido del problema.

Baking Lucky Cakes

El problema pide particionar los puntos en la mayor cantidad de triángulos no degenerados posible, donde cada punto debe pertenecer a lo más a un triángulo.

Si no existieran tres o más puntos colineales, entonces la respuesta sería simplemente $floor(n/3)$, ya que los puntos se podrían dividir en tripletas, posiblemente dejando uno o dos puntos sin grupo.

No podemos formar un triángulo con tres puntos colineales, ya que estaría degenerado. Sea M la cantidad de puntos que contiene la línea con más puntos del conjunto, entonces:

- Si $floor(M/2) > n–M$, entonces la respuesta es $n–M$. Para formar la mayor cantidad de triángulos en este caso, hay que formarlos tomando dos puntos de la línea con más puntos y un punto del resto.
- Si $floor(M/2) <= n–M$, entonces la respuesta es $floor(n/3)$. Para lograr la respuesta en este caso, podemos realizar $floor(n/3)$ pasos, y en cada paso tomamos tres puntos, tal que dos puntos pertenecen a la línea con más puntos y un punto pertenece a la segunda línea con más puntos. Con la condición dada, siempre es posible elegir dichos puntos. Después de cada paso, se seguirá cumpliendo la condición $floor(M'/2) <= n'–M'$ hasta que n' sea menor a 3.

Para saber cuál es la línea con más puntos, podemos iterar por cada pareja de puntos, y aumentar un contador en un map que tenga como llave las rectas $y = mx + c$.

CLETS Patrols

Las probabilidades de moverse entre puestos se pueden representar como la matriz

$$P = \begin{pmatrix} P_{1,1} & P_{2,1} & ... & P_{n,1} \\ P_{1,2} & P_{2,2} & ... & P_{n,2} \\ . & . & ... & . \\ . & . & ... & . \\ . & . & ... & . \\ P_{1,n} & P_{2,n} & ... & P_{n,n} \end{pmatrix}$$

, donde $P_{i,j}$ es la probabilidad de moverse del puesto i al puesto j (nótese que la orientación es transpuesta a la usual matriz de adyacencia)

Podemos también construir el vector

$$A = \begin{pmatrix} A_1 \\ A_2 \\ . \\ . \\ . \\ A_1 \end{pmatrix}$$

Siendo A_i la probabilidad de que un guarda se encuentre en el puesto i.
Considerando ahora $B = PA$, podemos ver que $B_j = \sum_{i=1}^{n} P_{i,j} \cdot A_i$, donde cada entrada de esta suma es el producto de la probabilidad que el guarda se encuentre en el puesto i, con la probabilidad de que, al estar en el puesto i, un guarda se mueva al puesto j. Es fácil ver que la suma de estos representa la probabilidad de que el guarda termine en el puesto j después de tomar un paso, y que B, entonces, es la nueva distribución de probabilidad de la posición del guarda, si la anterior era A.
Con esto tenemos una solucion al problema: cada vez que multiplicamos P por un vector de probabilidades, avanzamos un paso, por lo que tenemos que calcular

$$\underbrace{P(P(P...P(}_{m} \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix})...))$$

con $O(n^3 m)$, pero podemos mejorar el tiempo gracias a la asociatividad del producto de matrices:

$$P^m \begin{pmatrix} 1 \\ 0 \\ ... \\ 0 \end{pmatrix}$$

se puede calcular en $O(n^3 log(m))$, al calcular P^m con el algoritmo logarítmico comúnmente utilizado en la exponienciación binaria de números, de nuevo, gracias a la asociatividad del producto de matrices.

Determine the Winner Marshaland

Sea $p > 2$ un número primo. Si el equipo ganador está formado por p miembros y por lo tanto encontró p pelotas, y tomando en cuenta el bono que daría el gobernador, entonces, el equipo habrá obtenido $2^p + k$ marsh-coins.

El pequeño teorema de Fermat dice que si p es un número primo, entonces para cualquier entero a:

$$a^p \equiv a \ (mod \ p)$$

Particularmente estamos interesados en el caso donde $a = 2$. Y entonces:

$$2^p \equiv 2 \ (mod \ p)$$

De esta expresión se puede observar que $p|2^p - 2$.

Usando esto junto con nuestra expresión $2^p + k$, tenemos que:

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

Entonces para saber si p divide a $2^p + k$, solo debemos verificar si p divide a $k + 2$, es decir $p|k + 2$.

Entonces la solución para un valor K hay que imprimir todos los números primos que dividen a $K + 2$. Para encontrar estos números primos se puede utilirzar una criba de Eratosthenes.

Enterprise Recognition Program

Algo importante a notar es que dado que cada empleado tendrá un único gerente a quien reportar, excepto Jaime, y que la unidad de reporte de Jaime será toda la compañía, entonces la estructura jerárquica de la organización es un árbol.

Suponga que para un empleado u se conocen las respuestas a cualquiera de las dos preguntas que se pueden hacer para todos los empleados de los que u es el gerente directo. Entonces, las respuestas a las consultas para el empleado u serán:

$$puntos\_up(u) = puntos\_up(u) + \sum_{v \in hijos(u)} puntos\_up(v)$$

$$unidad(u) = puntos(u) + puntos\_up(u) + \sum_{v \in hijos(u)} unidad(v)$$

Donde $puntos(u)$ es la suma de puntos que ha recibido el empleado u directamente, $puntos\_up(u)$ es la suma de puntos que se pasan por toda la cadena jerárquica que contiene a u, y $unidad(u)$ es la suma de puntos que tiene la unidad organizacional representada por el empleado u, entonces, podemos observar que si conocemos estos valores la respuesta a la consulta de puntos que tiene un empleado e en específico será $puntos(e) + puntos\_up(e)$, y la respuesta a la consulta de los puntos que tiene toda la organización del empleado e será $unidad(e)$, pudiendo dar respuesta a cada consulta en $O(1)$.

Entonces, al principio se inicializan todos los $puntos(u)$ y $puntos\_up(u)$ en 0, para cada uno de los registros de puntos se aumenta la cantidad de *puntos* o *puntos_up* del empleado e que lo recibió, según el tipo de registro, cada una de estas actualizaciones toma $O(1)$. Los *puntos_up* de ese empleado se pasarán por la cadena en el próximo paso, si durante la lectura de los registros los *puntos_up* se pasan por el camino hasta la raíz la complejidad del algoritmo aumentaría en un orden de complejidad y daría TLE.

Después de la lectura de todos los registros, tendremos la solución para todas las hojas del árbol (los empleados que no son gerentes de ningún otro empleado), y entonces se pueden realizar las actualizaciones de *puntos_up* y *unidad* como se describió más arriba para todos los nodos siguiendo el orden topológico del árbol desde las hojas hasta la raíz, como cada nodo se procesa una sola vez, y cada nodo es hijo solo de un nodo, entonces el orden de complejidad de aplicar todas las actualizaciones será O(N).

Fitness Baker

Podemos observar que hay en total $5 * N^2$ celdas en la casa de Baker, además cada una de las celdas de la casa se podrá emparejar con $5 * N^2 - 1$ celdas (todas las demás excepto esa celda). En base a esto, podemos saber que habrá en total $(5 * N^2)(5 * N^2 - 1) = 25N^4 - 5N^2$ caminos diferentes que se deben calcular y sumar. Cada uno de los caminos tendrá una distancia que va entre 1 y $4 * N - 2$.

Si utilizamos una solución que haga fuerza bruta para calcular las sumas entre todos los pares de puntos, observaremos que los primeros valores son:

N	f(N)
1	16
2	600
3	3680
4	19904
5	61000
6	152136
7	329280

Si graficamos estos puntos podemos observar que la gráfica forma un polinomio, y dado que sabemos que la cantidad de celdas está en el orden $O(N^4)$ y la máxima distancia en el orden $O(N)$, podemos comprobar que el polinomio estará en el orden $O(N^5)$, y por lo tanto, el polinomio será de grado 5.

Utilizando interpolación de Newton podemos encontrar el polinomio que pasa por los puntos que ya conocemos de la función y al aplicar el módulo $10^9 + 7$ a los términos del polinomio obtendremos que $f(N) = 666666691N^5 + 333333332N^3 \ mod \ 1000000007$.

Goombas Colliding

Como los Goombas estarán rebotando mucho entre ellos, llevar un registro de la posición de cada uno después de cada rebote será difícil y tardado. Pero, si visualizamos el problema con una gráfica que muestre la posición y tiempo de los goombas podremos observar facilmente una propiedad. Tomemos el siguiente caso de prueba como ejemplo:

- La plataforma tiene una longitud de 11 bloques.
- $\{(1,1),(3,1),(6,0),(9,0)\}$ es el conjuntode los Goombas (cada par representa la posición y dirección inicial de cada uno).

Si grficamos entonces la posición y el tiempo en los ejes horizontal y vertical respectivamente, obtendremos la siguiente gráfica:

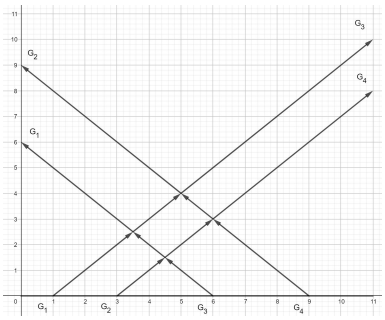

Figura 1: Time-Position graph for the case given

De la gráfica, podemos observar que el goomba G_3 será el último en caer de la plataforma en $t = 10$, después de haber rebotado con tres veces, primero con G_2, después con G_4, y al final con G_2.

Compare esta gráfica con la que se obtiene del mismo caso pero ignorando los rebotes:

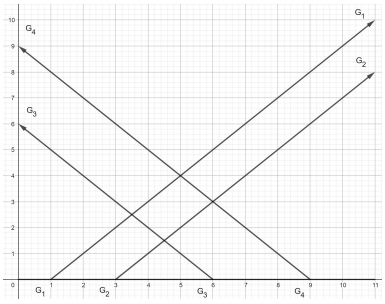

Figura 2: Time-Position graph for the case given (ignoring bounces)

Ambas gráficas son virtualmente la misma, los caminos son los mismos y la única posible diferencia es que las posiciones finales de los goombas podrían ser diferentes entre ellas.

Como en el problema solo nos interesa saber cuándo caerá de la plataforma el último Goomba, no cuál Goomba será, se puede resolver el problema ignorando los rebotes. Solo debemos identificar cuál de los caminos tomará más tiempo.

Cada Goomba define un camino, y hay dos casos posibles para el camino definido por el Goomba en la posición x. Si el goomba está apuntando a la izquierda entonces el camino tomará x segundos, de otro modo tomará $L - x$ segundos.

Para obtener la respuesta entonces, solo hay que iterar sobre todos los caminos y guardar el máximo. Complejidad $O(G)$.

Hamsters Training

Primero hay que notar que para cada subconjunto de N cartas solo hay una configuración válida, la que cumple con $e_{i-1} \leq e_i$ (donde e_i son los elementos de la línea). Entonces, el problema se puede resolver contando el número de diferentes subconjuntos de tamaño N de las N^2 cartas. Note que por las condiciones las N cartas en la línea podrían tener números repetidos.

Cualquier línea de hamsters correcta se puede visualizar también de la siguiente manera:

$$M_1, M_2, M_3, ..., M_N$$

Donde M_i representa la cantidad de cartas con el número i que se acomodarán en la línea. M_i puede tener un valor entero entre $[0, N]$.

Nuestra tarea entonces se puede centrar en encontrar todas las tuplas que cumplen:

$$M_1 + M_2 + \cdots + M_N = N$$

Pensemos en esta expresión usando combinatoria: Debemos de dividir N unidades en N subconjuntos – que pueden incluso ser vacios –, y esto es algo que se puede resolver usando la estrategia de "barras y estrellas". Dado que necesitamos N subconjuntos, será necesario introducir $N - 1$ barras a las N estrellas (recuerde que es posible $M_i = 0$, lo que significa que ninguna carta con el número i fue elegida por los hamsters). Entonces, Dejamos de concentrarnos en las tupas de N elementos y trabajamos con los conjuntos de $N + (N - 1) = 2N - 1$ elementos.

Finalmente, contamos las maneras en las que se pueden poner las $N - 1$ barras en el conjunto de $2N - 1$ elementos, lo cual se puede calcular con el siguiente binomio:

$$\binom{2N-1}{N-1}$$

Se debe tener cuidado en la implementación y utilizar apropiadamente los inversos multiplicativos para obtener la equivalencia modulo $10^9 + 7$.

Integers Rectangle Challenge

El problema se reduce a encontrar 3 subrectángulos que no se traslapen, cuya suma total sea la mayor posible.

La primer observación es que siempre es posible realizar dos cortes consecutivos a la cuadrícula original, de tal manera que cada uno de los 3 mejores subrectángulos quede en un área por sí solo. Una vez hecho esto, se puede observar que cada subrectángulo es el mejor del área donde se encuentra.

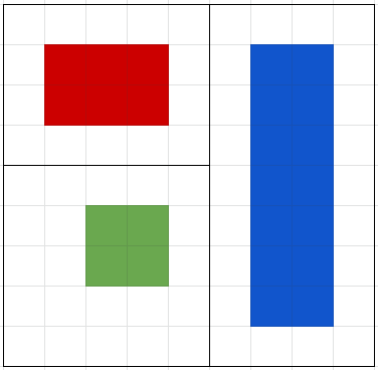

Figura 1: Separación de subrectángulos

Dado lo anterior, solo falta calcular rápidamente el subrectángulo con la suma más alta que se encuentre en cualquier área posible. Ésto se puede lograr con programación dinámica, observando que el mejor subrectágulo de un rectángulo dado es el mayor de:

- El subrectángulo que no incluye la primer columna.
- El subrectángulo que no incluye la última columna.
- El subrectángulo que no incluye la primer fila.
- El subrectángulo que no incluye la última fila.
- La suma total del rectángulo mismo.

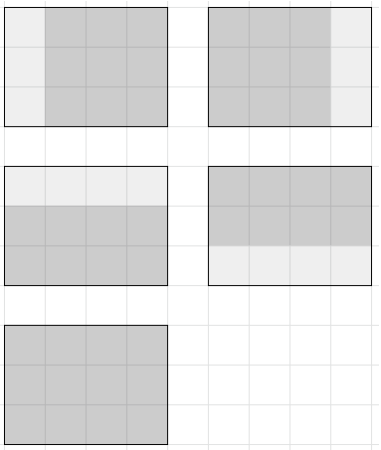

Figura 2: Subrectángulos inmediatos de un rectángulo

Juntando todo lo anterior, se reduce el problema a precalcular la suma del mejor subrectángulo de cada posible rectángulo en tiempo y espacio $O(n^2m^2)$, y luego intentar cada combinación de cortes de área en tiempo $O(max(n,m)^2)$.

Just Turn the Wheels!

Una observación a realizar es que independientemente de la cantidad de lados que tengan los polígonos de las dos llantas, cada 12 "turns" siempre estarán en la condición necesaria debido a que habrán girado ambas exactamente $12 * 30 = 360$ grados, y entonces estarán en la misma posición que al iniciar habiendo recorrido C de distancia. Usando esta información podriamos decir que siempre después de $\lceil \frac{12*S}{C} \rceil$ "turns" las dos ruedas estarán paralelas al piso y entonces se cumpliría la condición, sin embargo, esta no es necesariamente la mínima cantidad de vueltas.

Tomemos como ejemplo un caso donde las dos ruedas tienen 3 lados, podemos observar que en este caso, cada rueda después de $\frac{12}{3} = 4$ "turns" tendrá un lado paralelo al piso, y habrá recorrido una distancia de $\frac{4*C}{12}$, en este caso, la cantidad mínima de "turns" t será el número mínimo de veces que hay que dar 4 "turns" para recorrer al menos S distancia, esto es, el valor t más pequeño que cumpla:

$$t * \frac{4 * C}{12} \geq S$$

Despejando t tendremos que :

$$t = \lceil \frac{12 * S}{4 * C} \rceil$$

Y entonces habrá que dar $4 * t$ "turns".

Si las ruedas tienen polígonos con distintos números de lados podemos aplicar la misma estrategia que en el ejemplo anterior, primero debemos encontrar un valor m que es el mínimo número de "turns" que deben darse para que las dos ruedas estén con un lado paralelo al piso, esto lo podemos obtener con el máximo común divisor, y entonces $m = \frac{12}{MCD(MCD(F,B),12)}$. Después obtenemos el valor de t :

$$t = \lceil \frac{12 * S}{m * C} \rceil$$

Y por último, se calcula la cantidad de "turns" como $turns = m * t$

Kitchen Waste

Los límites dados permiten que se realice la simulación de como se sirve la sopa durante la noche. Se inicializa un contador y al seguir la simulación cada que se deja de utilizar una olla se suma al contador la cantidad de sopa que queda en la olla, si se terminan los panes se suma el volumen de sopa que tiene la olla actual y el volumen total del resto de las ollas.

Una observación más que se puede hacer es que la cantidad de sopa que se deshechara es la diferencia entre el total de sopa que se tiene (la suma de las capacidades de las ollas) y la cantidad de sopa que se debe servir (la suma de las capacidades de cada pan). Esta observación se puede comprobar de la siguiente manera, consideremos el valor O_i la capacidad de la i-ésima olla, B_j la capacidad del j-ésimo pan, y U_i la cantidad de sopa que se sirvió de la olla i antes de ser deshechada, entonces, el volúmen que se desperdició será:

$$\sum_{i=1}^{M} O_i - U_i = \sum_{i=1}^{M} O_i - \sum_{i=1}^{M} U_i$$

.

Como se garantiza que todos los panes serán servidos, entonces, la capacidad que se utilizó entre todas las ollas es igual a la capacidad de los panes que se sirvieron, por lo que:

$$\sum_{i=1}^{M} U_i = \sum_{j=1}^{N} B_j$$

Al sustituir este valor en la fórmula anterior tenemos que:

$$\sum_{i=1}^{M} O_i - U_i = \sum_{i=1}^{M} O_i - \sum_{i=1}^{M} U_i = \sum_{i=1}^{M} O_i - \sum_{j=1}^{N} B_j$$

Lets Count Factors

La solución es construir una criba modificada con los números del 2 al 10^7, que es el valor máximo que puede tener cada valor A y B, que contenga para cada número X la cantidad de primos diferentes que se requiere multiplicar para obtener X.

Una vez que se tiene esta criba, solo basta con encontrar el número de factores primos diferentes de A y el número de factores primos diferentes de B, sumarlos y restar la cantidad de factores primos que tengan en común A y B, lo cual puede hacerse obteniendo $m = MCD(A, B)$ y encontrando el número de factores primos de m.

La complejidad de calcular la criba es aproximadamente $O(M * loglogM)$, donde M es el máximo número para el que construiremos la criba, y para cada una de las N preguntas que se haga, el cálculo del MCD es lineal en el número de bits del mayor de los números A y B, $O(log(max(A, B))) = 24$ en el peor de los casos, dando entonces una complejidad para contestar las preguntas de $24 * N$ una vez que se tiene la criba

Magic Spells

Consideremos que la cadena está indexada en 0 en lugar de estar indexada en 1. De esta forma, la condición de las tuplas se convierte en que debe de existir una tupla mágica (s_i, s_{i+1}, p), tal que i es un múltiplo de p. También sustituyamos cada carácter con su posición en el alfabeto (a con 0, b con 1, ..., t con 19).

Ahora, construyamos un grafo con $20 * N$ nodos, donde cada nodo está denotado por una pareja (i, c), para todo i en el rango $[0, (N-1)]$ y toda c en el rango $[0, 19]$. Si tenemos una tupla mágica (c_i, d_i, p_i), coloquemos una arista dirigida desde (j, c_i) hasta $(j+1, d_i)$ para todas las j válidas que son múltiplos de p_i. Ahora, el problema se convierte en contar la cantidad de caminos entre $(0, x_j)$ y $(M–1, y_j)$, donde dos caminos son diferentes si las secuencias de nodos que siguen son distintas.

Para una N no muy grande, podemos construir el grafo y resolver el problema con una DP, ya que es un DAG. Veamos cómo resolverlo para N hasta 10^{18}.

Definamos la matriz $M_{i,j}$ de $20x20$, donde la posición $[x, y]$ de esta matriz (denotada por $M_{i,j}[x, y]$) es igual a la cantidad de caminos desde el nodo (i, x) hasta el nodo (j, y). Veamos que se cumple que $M_{i,j} = M_{i,k} * M_{k,j}$, para $i < k < j$:

La posición $[x, y]$ del producto $M_{i,k} * M_{k,j}$ se define como la suma de $M_{i,k}[x, z] * M_{k,j}[z, y]$ para toda $0 \leq z \leq 19$. El producto $M_{i,k}[x, z] * M_{k,j}[z, y]$ es igual a la cantidad de caminos desde (i, x) hasta (j, y), pasando por (k, z), es decir, los caminos de la forma $(i, x), \ldots, (k, z), \ldots, (j, y)$. Si sumamos estos productos para todas las z, entonces $(M_{i,k} * M_{k,j})[x, y]$ es la cantidad de caminos desde (i, x) hasta (j, y). Esta idea es parecida al problema de encontrar caminos de tamaño k en un grafo, con la diferencia de que las aristas dependen del "nivel i" en el que están.

Consideremos la secuencia de matrices $(M_{0,1}, M_{1,2}, M_{2,3}, \ldots, M_{n-2,n-1})$, en donde colocaremos un 1 en $M_{j,j+1}[ci, di]$ si existe una tupla mágica (c_i, d_i, p_i) tal que j es múltiplo de p_i, y un 0 en otro caso. Usando la propiedad anteriormente mencionada, tenemos que $M_{0,n-1} = M_{0,1} * M_{1,2} * M_{2,3} * \cdots * M_{n-2,n-1}$.ç

Notemos que las aristas se "repiten" cada $MCM(p_1, p_2, \ldots, p_m)$ niveles, es decir, se cumple que $M_{j,j+1} = M_{j+MCM(p1,p2,\ldots,pm),j+MCM(p1,p2,\ldots,pm)+1}$. Como p_j puede tomar valores entre 1 y 10, entonces el MCM es a lo más 2520, con lo que podemos asegurar que $M_{j,j+1} = M_{j+2520,j+2521}$. Utilizando lo anteriormente mencionado y la propiedad asociativa de la multiplicación de matrices, tenemos que:

$$M_{0,n-1} = M_{0,1} * ... * M_{n-2,n-1} = (M_{0,1} * ... * M_{2519,2520})^{(n-1)/2520} * M_{0,1} * ... * M_{(n-2)\%2520,(n-1)\%2520}$$

Podemos utilizar exponenciación binaria para obtener la exponenciación de las matrices. Para ahorrar tiempo, p odemos precalcular el valor de $M_{0,2520} = M_{0,1} * \cdots * M_{2519,2520}$, sus potencias de la forma 2^k para la exponenciación binaria, y los productos de prefijos $M_{0,1} * \cdots * M_{j,j+1}$, para todas los j menores a 2520. Complejidad: $O(2520 * 20^3 + 2520 * m + q * log(n) * 20^3)$

Newest Jaime's Delivery

Digamos que Jaime se encuentra en la bodega u, ha entregado n paquetes y le queda c combustible, definiremos entonces la función: $f(u, n, c)$ que nos dice si es posible o no regresar al origen entregando todos los paquetes sin quedarnos sin combustible, el si es o no posible hacerlo dependerá de las siguientes situaciones:

- Si Jaime se encuentra en la bodega u, puede moverse a alguna de las bodegas v siempre y cuando Jaime conozca el costo ($costo(u, v)$) de combustible necesario para ir de la bodega u a la bodega v, y el vehículo tenga al menos esa cantidad de combustible. Entonces f(u, n, c) es posible si alguno de los siguientes es posible: $f(v, n, c - costo(u, v))$, para cada v que se conoce el costo entre u y v.
- Si la bodega u, era un punto de entrega, se puede registrar esa entrega como realizada utilizando máscaras de bits sobre el valor de n y entonces sabemos que es $f(u, n, c)$ es posible si es posible alguno de los : $f(v, n|posicion(u), c - costo(u, v))$, para cada v que se conoce el costo entre u y v
- Si u es un punto de recarga de combustible, se recarga el combustible antes de seguir moviendose, y entonces $f(u, n, c)$ será posible si es posible alguno de los $f(v, n, min(c + p_u, F) - costo(u, v))$ si u no es punto de entrega, o $f(v, n|posicion(u), min(c + p_u, F) - costo(u, v))$ si u es un punto de entrega, para cada v que se conoce el costo entre u y v.
- Se puede observar entonces que si durante el recorrido se llega a $f(1, N', c)$ donde $N' = 2^K - 1$ (la máscara cuando todos los paquetes se han entregado), entonces no es necesario seguir con la función debido a que alcanzar este caso demuestra que es posible entregar.

En la implementación de esta función es importante mantener una memorización de los estados que se han alcanzado para reducir el tiempo de procesamiento por calcular varias veces la respuesta a alguna de las opciones.

Entonces, lo que el problema pide es el valor de c más pequeño tal qué $f(1, 0, c)$ es posible. Para responder esto, después de haber implementado la función $f(u, n, c)$ se puede aplicar búsqueda binaria sobre el valor de c, buscando el valor más pequeño de c, la búsqueda binaria funciona porque si es posible $f(1, 0, c)$ también será posible $f(1, 0, c + d)$ con $d > 0$, debido a que el camino que hace posible $f(1, 0, c)$ también es un camino posible para $f(1, 0, c + d)$.

ICPC Masters Mexico 2020 - 2021

Enero 30 de 2021

Libro de soluciones

Este documento contiene las soluciones esperadas para los 8 nuevos problemas usados durante la competencia, así como las referencias de los problemas no originales tomados de competencias anteriores.

Problem A

Adventure on the space

Este problema apareció como el problema A en la segunda fecha del Gran Premio de México 2017.

Problem B

Buggy Text Processor

Este era probablemente el problema más simple de la competencia. Lo único que se debe hacer es llevar un contador inicializado en 0 y procesar caracter por caracter aumentando el contador cada vez que se encuentre una letra mayúscula o minúscula.

Problem C

Containers

Este problema apareció como el problema C en la tercera fecha del Gran Premio de México 2016.

Problem D

DIOS primes

Este problema apareció como el problema D en la segunda fecha del Gran Premio de México 2017.

Problem E

Ellipses

Debido a que los límites en los tamaños de cada elipse son pequeños, se puede hacer un recorrido en puntos con coordenadas enteras en una region que garantice que cualquier elipse que se de en la entrada no tendrá puntos fuera de ese rectánculo, varias soluciones usaron un cuadrado delimitado entre $(-1000, -1000)$ y $(1000, 1000)$, . Después de observar esto, la parte complicada del problema es revisar dada una coordenada si esta se encuentra dentro o no de la elipse, por la descripción del problema sabemos que un punto p está en el perímitro de la elipse si se cumple que: $dist(p, F_1) + dist(p, F_2) = D$, donde $dist$ es una función que mide la distancia entre dos puntos: $dist(P_1, P_2) = \sqrt{(P_{2_x} - P_{1_x})^2 + (P_{2_y} + P_{1_y})^2}$.

Podemos observar entonces que si $dist(p, F_1) + dist(p, F_2) > D$ entonces el punto está fuera de la elipse, y si $dist(p, F_1) + dist(p, F_2) < D$ entonces el punto está dentro de la elipse.

Problem F

Finding the Train

Este problema apareció como el problema F en la primera fecha del Gran Premio de México 2018.

Problem G

Game

Este problema apareció como el problema E en la segunda fecha del Gran Premio de México 2019.

Problem H

HTML From Shortcuts

La descripción del problema muestra directamente la tarea que se tiene que realizar: reemplazar los caracteres '_' y '*' en tags de apertura y cierre de HTML para italic y bold, respectivamente. En general, lo único que podría dificultar la generación del texto HTML a partir de la entrada es decidir al leer uno de los caracteres especiales si se debe imprimir el tag de apertura o el de cierre; el hecho de que no habrá tags anidados facilita esta decisión. Se puede utilizar una variable booleana que nos permita conocer si se ha impreso un tag de apertura para italic, y otra variable para bold, así al leer un '_' podemos decidir: si la variable que revisa si se ha impreso un tag de apertura de italic es verdadera entonces se imprime un tag de cierre y la variable se asigna a falso, en caso contrario se imprime un tag de apertura y la variable se asigna a verdadero. El mismo proceso se puede hacer con la variable que revisa si se ha impreso el tag de apertura para bold. Para cualquier otro caracter en la entrada se imprime el caracter leído.

Problem I
Interesting DNA Sequences

Este es un problema clásico de programación dinámica. Se debe llevar un registro de cuántas ocurrencias de cada uno de los genotipos se ha utilizado y cuál fué el último genotipo que se puso en la cadena, así, podemos saber cuál es el genotipo que no podemos poner a continuación. Usando esta observación se puede realizar una relación de recurrencia en la que consideramos cuál fue el último genotipo agregado para contar el total:

$$cont(A,C,G,T,last) = \begin{array}{llllll} cont(A-1,C,G,T,0) & si & A>0 & y & last \neq 0 \\ +cont(A,C-1,G,T,1) & si & C>0 & y & last \neq 1 \\ +cont(A,C,G-1,T,2) & si & G>0 & y & last \neq 2 \\ +cont(A,C,G,T-1,3) & si & T>0 & y & last \neq 3 \end{array}$$

Problem J

John's New Product

A pesar de que parece que hay muchas variables para resolver este problema, los casos de ejemplo daban algunas pistas. Los límites en la entrada garantizan que siempre habrá alguna manera de asignar los productos candidatos de modo que se cumplan las restricciones establecidas John.

La entrada da una evidencia clara de que estamos trabajando con un grafo donde las ciudades son los vértices y las carreteras que las unen son aristas. El asignar los productos candidatos en el grafo de modo que ciudades vecinas no tengan el mismo producto candidato es una aplicación de bi coloración del grafo. En general, nos interesa bi colorear cada una de las componentes conexas que el grafo tiene, esto se puede lograr con un recorrido al grafo de la entrada casi de manera directa y es un problema clásico: Para cada componente en el grafo, se toma un nodo arbitrario como inicial y se le asigna un producto, de ahí se recorre el grafo asignando el producto diferente a cada uno de los vecinos, y se repite para cada uno de ellos hasta que se hayan coloreado todos los vértices. Por las características que tiene la bicoloración del grafo podemos observar que solo existen dos posibles asignaciones de productos para cada componente conexa, la que ya se realizó y otra donde todas las ciudades a las que se les asignó el producto A se les asigna el B y viceversa. Debido a esto, supongamos que una componente conexa i tiene una diferencia D_i de testers entre el producto A y el producto B con la asignación de productos que se realizó, entonces, la otra posible asignación tendrá una diferencia de $-D_i$.

Una vez que se tiene todo el preprocesamiento definido en el parrafo anterior, el problema se reduce a encontrar para cada componente i cual asignación se usará, la que aporta D_i o la que aporta $-D_i$, de modo que el valor absoluto de la suma sea el menor posible, este paso se puede resolver haciendo una pequeña modificación al clásico problema de la mochila.

Problem K

K Scores

La consulta de obtener el K-ésimo elemento de un subarreglo después de ordenarlo es una tarea que se puede resolver utilizando una estructura de datos como el Wavelet tree.

Para responder el segundo valor de cada consulta se puede mantener la suma prefijo de cada valor de la entrada, llamemos a este preprocesamiento P, donde P_i contiene la suma de los valores del primer paquete al i-ésimo paquete de la entrada, y agregar una función al wavelet tree (que llamaremos $sumK$) está función que debería regresar la suma de todos los valores menores o iguales a un elemento dado en el rango definido por la consulta, esta función es solo una modificación a la estructura de cada nodo para almacenar las sumas y una leve modificación a la función que nos regresa la respuesta a la primer consulta. Así al tener los valores de P calculados y la función $sumK$ implementadas, el segundo valor que debemos regresar para cada consulta será: $P_R - P_{L-1} - sumK(L, R, query(L, R, K))$

Problem L
Lost Card Game

Un intento inicial para resolver el problema es el de simular el juego, al principio puede haber un conjunto de cartas que podría ser la primera en eliminarse, al eliminar una de estas tendremos una pila con $N-1$ cartas y podemos realizar el mismo procedimiento que se hizo para eliminar la primera carta de las N que hay, para encontrar la segunda con $N-1$. Entonces buscamos de todas las cartas que podrían ser la primera en eliminarse cual nos lleva a un mayor número de cartas eliminadas después de simular el juego. Este backtracking nos dará la respuesta correcta pero no correrá en el tiempo necesario.

Una solución más eficiente es similar al backtracking definido antes pero en lugar de enfocarnos en la primer carta que se eliminará nos enfocamos en dos casos:

1. Suponiendo que todas las cartas se pueden eliminar de una pila, entonces hay una carta C que es la última en eliminarse. En este caso, todas las cartas que se encuentran arriba de C (excepto la primera) y todas las que se encuentran abajo de C (excepto la última) se eliminarán de la pila antes de eliminar a C. Podemos observar que estos dos conjuntos de cartas (el de las que están arriba de C y el de las que están abajo de C), no se afectarán entre ellos: en caso de que una carta abajo de C requiera una carta arriba de C para ser eliminada, entonces C no sería la última carta a eliminarse en la pila. Como estos dos conjuntos son independientes, entonces podemos resolver el mismo problema para las pilas que se encuentran entre la primer carta de la pila hasta la carta C, y de otra pila definida entre la carta C y la última de la pila.

2. En caso de que no se pudieran eliminar todas las cartas de una pila, entonces hay otra carta C, que no será eliminada, podemos observar similar al caso anterior que al ser C una carta que no se eliminará, las cartas abajo de esta y arriba de esta no se afectarán entre ellas, entonces, podemos resolver tambien para las dos pilas que van entre la primer carta de la pila y la última.

Entonces, para resolver el problema podemos utilizar memorización, iniciando con la pila dada en la entrada, se prueba con cada carta en la pila como si fuera la carta C de cada uno de los dos casos, si se pueden eliminar todas las cartas y esta al final, o si esa carta no se eliminara. La memorización se usa para guardar el resultado entre los pares de pilas de modo que no se calcule el mismo valor más de una vez.

Problem M
Managing Race Results

Supongamos que encontramos un caballo del que sabemos que no debe terminar después que otro y digamos que este es el que terminará primero la carrera, podemos asignar el resto de caballos de manera recursiva de la misma manera ignorando el caballo que ya fue posicionado. Al hacer esto con cada posible caballo que pueda terminar en primer lugar entonces habremos encontrado la respuesta, sin embargo esta es demasiado lenta.

Una optimización a esta solución es observar que el número de formas de acomodar cualquier subconjunto será siempre la misma, sin importar el orden de los caballos que ya se han acomodado previamente, así se podría aplicar memorización sobre los subconjuntos para evitar recalcular ordenes que ya se han calculado previamente, esto es $O(2^N)$, y tampoco se ejecutará en tiempo.

Una última observación a realizar es que incluso cuando hay un máximo de 30 caballos, ninguna de las componentes del grafo de dependencias que se puede crear con las evidencias que se tienen tendrá más de 16 caballos. Así el algoritmo anterior se puede aplicar por separado a cada una de las componentes y al final calcular el total de ordenes posibles haciendo la unión de las formas entre las componentes. Si hay dos componentes que tienen H_1 y H_2 elementos respectivamente, y cada una de ellas tiene F_1 y F_2 formas distintas de ordenar a los caballos, entonces su unión tendrá $F_1 * F_2 * \binom{H_1+H_2}{H_1}$ formas diferentes de acomodar a los caballos.

Hay que tener cuidado en las operaciones para mantener el módulo de manera adecuada.

Gran Premio de México 2020 - Tercera fecha

05 de Diciembre de 2020

Libro de soluciones

Este documento contiene las soluciones esperadas para los 14 problemas usados durante la competencia.

Acing the contest

Como los concursantes deben escoger el orden en el que trabajan, se puede usar una máscara de bits para representar a los concursantes que ya han resuelto problemas.

De la descripción del problema cada concursante tiene tres opciones

- Hacer el problema j
- No hacer el problema j
- Dejar de resolver problemas

Sea $f(i, j, e)$ el máximo score que se puede obtener con el concursante actual que está resolviendo problemas, en el problema j, con energia e i en este caso es la máscara de bits.

Si el concursante hace la tarea j entonces el score será $v_j + f(i, j + 1, e - a_j)$ Si no la hace el score será $f(i, j + 1, e)$

Si deja de trabajar para calcular el score habrá que buscar con cada competidor k que no ha trabajado y el score será el máximo que se pueda obtener de $f(i|(1 << k), j, e_k)$ donde k es el índice de cada concursante que no ha trabajado aún.

La complejidad entonces será $O(N * (2^N) * P * E)$, donde E es el máximo valor de energía.

Beautiful necklaces

La idea está basada en el problema de encontrar el subarreglo más largo que contiene el mismo elemento, en un arreglo de enteros usando un Segment Tree. Ese problema se puede resolver guardando los siguientes valores en los nodos del Segment Tree:

- Máximo subarreglo con el mismo elemento que hemos encontrado en el rango del nodo hasta el momento
- El prefijo más largo con el mismo elemento en el rango del nodo
- El sufijo más largo con el mismo elemento en el rango del nodo

Cuando combinamos dos nodos, el hijo izquierdo y el hijo derecho, hay que quedarnos con el máximo subarreglo entre ambos nodos, pero también hay que considerar que el sufijo del nodo izquierdo se puede combinar con el prefijo del derecho si tienen el mismo elemento. El mayor prefijo del padre será el del hijo izquierdo y el mayor sufijo será el del hijo derecho, pero hay que considerar que, por ejemplo, si el hijo izquierdo tiene los mismos elementos en todo su rango y ese elemento es igual al prefijo del hijo derecho, entonces todo el nodo izquierdo se puede combinar con el prefijo del nodo derecho para convertirse en el mayor prefijo del padre; algo parecido sucede con el sufijo.

Para resolver el problema propuesto, podemos considerar los collares como arreglos de colores, sin embargo, hay que considerar dos cosas:

1. Como los collares son circulares, entonces los arreglos también deben serlo. Esto se puede manejar usando el truco de duplicar el arreglo con una copia de sí mismo. Sólo hay que tener cuidado cuando el collar tiene el mismo color en todas sus cuentas.
2. Las operaciones que puede hacer Alice se pueden ver como insertar o remover un elemento de un arreglo, por lo que en vez de un Segment Tree, necesitamos usar un Treap, ya que nos permite soportar estas operaciones. La mezcla de los nodos en el Treap será similar a como sería en el Segment Tree.

Una última consideración para la implementación es que los nodos del Treap deben guardar quién es su padre, ya que la entrada del problema no indica en qué collar Alice está haciendo la operación, sólo nos da los IDs de las cuentas, por lo que es necesario encontrar la raíz del Treap donde está el nodo que representa esos ID con tal de aplicar las operaciones.

Para mantener la belleza máxima de entre todos los collares que tiene Alice, se puede usar un map o un multiset.

Counting triangles

Para formar un triángulo se deben elegir dos lineas de las $N + 2$ que van de arriba hacia abajo en la imagen y una de las $K + 1$ líneas que son paralelas a la base. Entonces la solución es el producto de cuántas formas se pueden tomar dos de un conjunto de $N + 2$ por $K + 1$. Esto se calcula con la siguiente fórmula. $\frac{(N+2)*(N+1)*(K+1)}{2}$

Hay que tener cuidado para calcular el residuo. Una manera de calcularlo a pesar de tener la división es considerar que siempre alguno de $N + 2$ y de $N + 1$ será par, entonces primero se divide y se realizan todos los productos calculando el módulo entre cada uno de ellos.

Debugging the network

La solución es directa a la descripción del problema. Hay que dividir la cadena en números y letras, e interpretar los números de manera cuidadosa. Una vez hecho eso hay que repetir cada letra la cantidad de veces que especifica el número previo a el.

End of the year bonus

El bono $f(i)$ de la i-ésima persona se obtiene de la siguiente manera.

Si $p_i > p_{i-1}$ (tuvo mejor desempeño que la persona a su izquierda), entonces: $f(i) = B + f(i-1)$

Si $p_i > p_{i+1}$ (tuvo mejor desempeño que la persona a su derecha), entonces: $f(i) = B + f(i+1)$

Si $p_i > p_{i-1}$ y $p_i > p_{i+1}$ (tuvo mejor desempeño que las dos personas), entonces $f(i) = max(B + f(i-1), B + f(i+1))$

Solo hay que tener cuidado con la implementación para considerar que la primer persona está sentada a la derecha de la última.

Fit them all

Una vista rápida a los límites establecidos en el problema muestra que la cantidad máxima de posible de cajas a posicionar no será muy grande. Esto permite idear una estrategia de solución en la que se pruebe colocar primero la caja más grande posible dentro de la plataforma, digamos de dimensiones KxKxK y en cada posible posición que se pueda colocar esta caja colocar la siguiente caja menor en tamaño, y así sucesivamente hasta poner la caja más pequeña de dimension 1x1x1. Si no hay una manera de posicionar las K cajas, entonces se intenta lo mismo comenzando con la caja que mide $(K-1)$x$(K-1)$x$(K-1)$, y así hasta encontrar una dimension de caja inicial en la que se puedan colocar todas las cajas, la respuesta sería esa dimensión. Sin embargo probar todas las posibles posiciones en cada tamaño posible de caja lleva la solución a Tiempo Límite.

Se puede observar que siempre que exista una solución donde se posicionen K cajas, existe tambien una solución en la que estas K cajas comparten al menos un lado horizontal y un lado vertical, aplicando esta consideración se puede seguir con la misma estrategia descrita previamente, pero reduciendo considerablemente la cantidad de posiciones que se prueban al intentar colocar una nueva caja posible. Entonces en la estragia se colocará siempre la caja más grande que queremos probar en la solución en una posición tal que al menos un lado horizontal y un lado vertical de esta caja compartan un lado horizontal y un lado vertical de la plataforma, y despues se intenta colocar de una por una en orden de descendente de tamaño las demás cajas, colocándolas siempre en posiciones donde al menos uno de sus lados horizontales y uno de sus lados verticales se comparta con alguna caja ya posicionada o con la plataforma.

Gold Fever

Sea $f(i)$, la maxima cantidad de dinero que se puede obtener si estamos en la casilla i $f(i) = g_i + max = f(i+j)$ con j en el rango $[a_i, b_i]$.

Esta solución es de orden de complejidad $O(N^2)$, y daría tiempo límite, sin embargo se puede optimizar usando un segment tree, la DP es la misma, pero lo haremos de la siguiente manera.

Iniciando en la última posición, y recorriendo hacia atrás suponiendo que tenemos la solucion de $f(j)$ para $j > i$ y que hemos guardado previamente en el segment tree las soluciones para los $f(j)$. Queremos calcular $f(i)$ con una consulta al segment tree obtenemos el valor de $max(f(i+j))$, con j en el rango $[a_i, b_i]$. Después actualizamos el valor del segment tree en i, porque ya obtuvimos $f(i)$. La complejidad de esta implementación es $O(NlogN)$, y correrá en el tiempo límite dado.

How to Work Less to Pass a Programming Course in Planet E-13

Dado que el problema pide encontrar todas las posibles combinaciones posible, es necesario hacer una búsqueda exhaustiva, las combinaciones pueden probarse con una mascara de bits que varie entre 1 y $2^n - 1$ donde n es el número de calificaciones, para cada combinación obtener la calificación que se obtendría y en caso de cumplir (ser a menos la calificación deseada) se agregan a un vector llevando el rastro de la cantidad mínima de problemas que permite obtener al menos la calificación deseada.

El vector se ordena de acuerdo a la mascara de bits encontrada para que quede en el orden deseado y se imprime el número mínimo de problemas a resolver, la cantidad de combinaciones que permiten obtener al menos la calificación deseada con el número mínimo de problemas y se recorre el vector imprimiendo las combinaciones.

Is this the best deal?

Dado que siempre habrá 3 cantidades, llamémoslas t_1, t_2, t_3, y suponiendo que tenemos una función 'descuento(valor)' que regresa el valor que se habría que pagar por hacer una compra con un total de *valor*, entonces solo hay 5 posibles maneras de hacer las compras:
$s_1 = discount(t_1) + discount(t_2) + discount(t_3)$
$s_2 = t_1 + discount(t_2 + t_3)$
$s_3 = discount(t_1 + t_2) + t_3$
$s_4 = discount(t_1 + t_3) + t_2$
$s_5 = discount(t_1 + t_2 + t_3)$
La respuesta es el valor más pequeño entre s_1, s_2, s_3, s_4, y s_5.

Jaime's greedy delivery

A simple vista Se visualiza un floyd warshal para obtener el tiempo minimo entre todos los pares de ciudadss puntos, seria razonable, pero dado que las distancias/tiempos entre cualesquiera par de puntos entre los que hay alguna conexion es siempre la misma, y que por la cantidad de aristas el grafo puede ser disperso, basta con hacer N busquedas en amplitud para sacar las distancias.

Ya que se tienen las distancias minimas como preproceso se puede calcular el tiempo entre puntos que Jaime debe seguir en la ruta, esto es una dp:

Sea $f(i,t)$ la maxima cantidad de dinero que Jaime puede hacer si se encuentra en el punto i, y le queda t tiempo los puntos de entrega se van a expandir en dos, i, e $i+1$.

Si i es par, entonces está en un lugar de entrega normal, si i es impar, está en una entrega especial (estas le dan dinero).

Cuando i es par, puede rechazar el dinero y pasarse a la siguiente entrega es decir: $f(i+2, t - tiempo[punto_i][punto_i+2])$ o puede hacer el pedido especial $v_i + f(i+1, t - tiempo[punto_i][punto_i+1])$

Cuando i es impar debe pasarse al siguiente punto, sin mas (ya ha recibido el dinero) $f(i+1, t - tiempo[punto_i][punto_i+1])$

Resolviendo esta función en $f(0,T)$ se tendrá la máxima cantidad de dinero que Jaime puede obtener.

K contestants

Para formar el equipo se deben tomar i alumnos del grupo A, y $k-c-i$ Alumnos del grupo B Entonces se pueden tomar $\binom{A}{K-c}$ del grupo A y $\binom{B}{K-c-i}$ del grupo B, para así tomar exactamente c alumnos del grupo C como solicitan los profesores.

Supongamos que se puede formar el equipo con 0 alumnos de cualquiera de los grupos A y B, entonces se podrían tomar los $K-c$ alumnos de la siguientes formas: $\sum i = 0^{K-c}\binom{B}{K-c-i} = \binom{A+B}{K-c}$

Para eliminar los casos donde $i = 0$ o $i = K-c$ tenemos $\binom{A+B}{K-c} - \binom{A}{K-c} - \binom{B}{K-c}$ Para obtener la cantidad total de formas habrá que multiplicar esto por la cantidad de maneras de tomar los c alumnos del grupo C y entonces la solución sería: $(\binom{A+B}{K-c} - \binom{A}{K-c} - \binom{A}{K-c}) * \binom{C}{c}$

Hay que tener cuidado con el manejo de los residuos considerando los valores que se están restando, y contemplar los casos en donde no hay manera de formar algún equipo.

Let's count words

Los límites de este problema son suficientes para encontrar la cantidad de palabras usadas para generar las demás probando si cada par de palabras son iguales después de rotarlas una cantidad de veces. Digamos sea W_1 y W_2 dos palabras del mismo tamaño L, si después en alguna de las posibles L rotaciones de W_2 en alguna $W_1 = W_2$ entonces las dos palabras son generadas por una misma palabra. Entonces para resolver podemos inicializar un contador en 0, y para cada una de las N palabras probar, si ninguna de las palabras anteriores a ella son generadas por una misma palabra aumentar el contador en 1. Al finalizar con las N palabras, la respuesta está en el contador.

Mathematics society problem

Por la descripción que se da podemos observar que la cantidad total de dígitos a eliminar de N, digamos D es igual a la suma de la cantidad de veces que se debe eliminar cada uno de los 9 dígitos. El número resultante independientemente del orden en el que se eliminen lo dígitos tendrá siempre un tamaño de $M = |N| - D$. Entonces, buscamos el número de mayor valor de entre los números que se pueden generar con exactamente M dígitos después de eliminar los dígitos con las restricciones dadas. Podemos notar entonces que como todos los posibles números a probar tienen la misma cantidad de dígitos, entonces, el número más grande comenzará siempre con el dígito más grande que se pueda quedar después de eliminar los D dígitos a N. De este modo podemos considerar una solución en la que en lugar de decidir qué dígitos de N se deben eliminar, determinar qué dígitos de N se deben quedar. Dado el dígito d en la posición i, este se debe quedar para generar el número más grande siempre y cuando se cumplan dos condiciones:

- Hay al menos una manera de eliminar todos los dígitos antes de la posición i que no han sido seleccionados.
- El dígito debe aparecer al menos D_d veces después de la posición i, de otro modo, no se podrían eliminar todas las ocurrencias del dígito d en N.

Network connection

Supongamos que conocemos un valor de frecuencia f que se quiere utilizar y queremos determinar si con ese valor de frecuencia se pueden posicionar las antenas de algún modo tal que no se exceda el presupuesto B. Una de las maneras de determinar esto es intentar colocar las antenas en posiciones de modo que se minimice el presupuesto utilizado con usando la frecuencia f, así si el presupuesto mínimo necesario usando la frecuencia f es mayor a B entonces no hay manera de acomodar la red de modo que se conecte toda la carretera con la frecuencia f y el presupuesto dado. Este subproblema se puede resolver con programación dinámica. Sea $DP[i][j]$ el menor costo de acomodar las primeras i antenas de modo que haya una conexion entre Nlogonia y la i-ésima antene estando esta en el metro j de la carretera. Podemos observar que el valor de $DP[i][j]$ está siempre determinado por las soluciones dadas para haber conectado Nlogonia hasta la antena $i-1$ en las diferentes posibles posiciones que esta pueda estar, y que este valor será $DP[i][j] = min(DP[i-1][j-k] + abs(j-p_i))$ para k que está en el rango $[0, f]$. Para finalizar, hay que considerar que $DP[N][0...D]$ contiene los valores de conectar Nlogonia con las N antenas, entonces podemos observar que el presupuesto mínimo necesario para conectar las dos ciudades será $min(DP[N][D-k])$ con k en el rango $[0, f]$.

Supongamos que hemos probado una frecuencia f y esta frecuencia puede ser usada para conectar toda la carretera, entonces se puede observar que la misma configuración de antenas puede conectar la carretera con cualquier frecuencia $f_1 \geq f$. Podemos entonces comprobar que la función de poder o no utilizar una frecuencia f es una función monótona cresciente y por lo tanto podemos usar búsqueda binaria para encontrar el valor f más pequeño que tiene una solución usando la programación dinámica descrita para cada valor posible utilizado de f. El orden de complejidad de la solución es entonces $O(N * D * logD)$

Maratona de Programação da SBC 2020

This problem set is used in simultaneous contests:
Maratona de Programação da SBC 2020
Segunda Fecha Gran Premio de México 2020
Primera Fecha Gran Premio de Centroamérica 2020
Torneo Argentino de Programación 2020

November 14th, 2020

Editorial

This editorial was prepared by the following volunteers:

- Agustín Santiago Gutiérrez
- Fernando Fonseca Andrade Oliveira
- Naum Azeredo Fernandes Barreira
- Teodoro Freund

Problem A
Sticker Album

As in most expected value problems, it's useful to name the expected value of the number of packets one must buy to fill an album that has x empty slots left: let this value be E_x. We have $E_0 = 0$, as it is not necessary to buy any packet to fill a full album.

As the number of cards per packet is uniformly random, the probability of a packet containing any number of cards between A and B is the same, $\frac{1}{B-A+1}$. To make the next expressions simpler, let's denote $L = B - A + 1$.

If the album needs x more cards, after buying a packet, with probability $\frac{1}{L}$ the album will need $x - A$ cards, with the same probability it's going to need $x - A - 1$, and so on. Therefore we can write E_x as a function of other values of E:

$$E_x = 1 + \frac{1}{L}E_{x-A} + \frac{1}{L}E_{x-A-1} + \cdots + \frac{1}{L}E_{x-B}$$

Using the equation above to calculate $E_1, E_2, \cdots, E_N$ allows us to solve the problem in $O(N(B - A))$, which is too slow. To make it faster, we can use a sliding window of sums of $B-A+1$ consecutive values of E to get the value of the right side sum quickly. Alternatively, we can also calculate prefix sums of E to get the sum in constant time, but we need to be careful with double precision issues when using this approach. Both approaches make the final solution work in $O(N)$.

A detail is that A can be zero, and in this case in the equation above E_x would depend on itself. This is not a big problem, because we can still solve the equation for E_x:

$$E_x = 1 + \frac{1}{L}E_x + \frac{1}{L}E_{x-1} + \cdots + \frac{1}{L}E_{x-B}$$

$$E_x = \frac{L}{L-1}\left(1 + \frac{1}{L}E_{x-1} + \cdots + \frac{1}{L}E_{x-B}\right)$$

Problem B

Battleship

For this problem, it is enough to place each ship in the board, one by one, and check if each ship can be placed according to the conditions in the statement.

At the start, we can represent an empty board by a 10x10 two-dimensional array filled with zeros, indicating there are no occupied squares. Whenever a new ship is placed, we can use a for or while loop to iterate over all squares that the ship occupies, and mark all of those squares in the matrix with a 1, indicating that those squares are now full. If at any point we try to place a ship in a square that was already full, or a square that is outside the 10x10 board, we report that the configuration is not good. Otherwise, the configuration is good.

Problem C
Concatenating Teams

The key observation is to concentrate, within university A, on the triplets (x, x', S) such that $x = x'S$ and both x and x' are team names from university A.

Similarly, we can consider the triplets (z, z', S) such that $z' = Sz$ and both z and z' are team names from university B.

A certain name X is not peculiar, exactly when there exists such a triplet (x, x', S) for university A and such (z, z', S) for university B, both having the same S and such that X is one of the 4 different teams involved in this pair of triplets.

One might think at first that there can be too many triplets to explicitly generate them all, but this is not so: For each x for example, there is at most one triplet (x, x', S) for each prefix of x, and so the total number of triplets is at most the total number of characters in the input strings.

Hashing seems to be the easiest way to efficiently generate and group the triplets: Iterate all prefixes of all the words in the first set. If a prefix of a word is found to be some other word from the set (which can be efficiently checked by putting all hashes in a set/map), the remaining suffix (whose hash can also be efficiently computed) is the S involved in the triplet. The same can be done for set B, and then all triplets can be grouped by S and iterated. Once grouped, for each S such that at least one triplet with S exist for each university, then **all** of the x, x', z, z' involved in any such triplet can be marked as nonpeculiar. Those names remaining unmarked after all of this are precisely the peculiar names.

Using a simple trie, all the triplets can be generated in the same time, but while x and x' can be stored explicitly, S will be computed as a pair of starting and ending indices i, j. To then group the triplets having identical S from this representation is not quite simple.

There is that detail that, while there are $O(input)$ different S, and they can be identified and grouped by equality using hashing, the total sum of their lengths can be very big, so explicitly creating a trie or similar with all of them can be too large.

This solution runs in a linear number of map operations.

By creating a suffix array / suffix tree of the words and working with it carefully, a similar fully deterministic efficient solution can be written, although it is quite harder and longer than just using hashing.

Problem D
Divisibility Dance

Since the only movements allowed are rotations of an entire circle, the final configuration must also be some rotation of the initial pairing. To solve this problem, we need to answer two questions: which of the rotations satisfy the sum condition, and in how many ways can we reach each of those valid final configurations.

For the first question, it's useful to notice that, if the pairwise sum is constant when summing corresponding values of arrays A and B, then if going from position i to position $i+1$ the element in A increases by 3, the element of B must decrease by 3. In other words, the difference array of array B (the array D where $D_i = B_{i+1} - B_i$) is equal to the difference array of A with all elements negated.

Let's take the difference array of A, D_A, and the negative of the difference array of B, $-D_B$. We want to know for what rotations of D_A it becomes equal to $-D_B$. This is a classic string problem: we can concatenate two copies of D_A next to each other such that the N elements beginning at position i in the concatenated array correspond to D_A rotated i times, and then any string matching algorithm can be used to determine in which positions $-D_B$ matches the concatenated array.

Finally, we must determine the number of ways to reach each valid rotation. Note that from one configuration we can reach all rotations of that configuration except for the configuration itself, so by symmetry all rotations that are not the initial one are equivalent and can be reached in the same number of ways.

Let $f(x, 0)$ the number of ways to rotate x times and finish in the starting configuration, and $f(x, 1)$ the number of ways to rotate x times and reach some other configuration (as the number of ways is the same for all other configurations, it doesn't matter which one). We can then write:

$$f(x, 0) = (N - 1)f(x - 1, 1)$$
$$f(x, 1) = f(x - 1, 0) + (N - 2)f(x - 1, 1)$$

In other words, the initial configuration can be reached by any of the $N - 1$ other ones, and each of the other ones can be reached by the initial configuration and the other $N - 2$ configurations that are not themselves.

This is a linear recurrence, so it can be solved by matrix exponentiation in $O(\log K)$.

Problem E
Party Company

For each party i, first use binary lifting to find what is the earliest supervisor S_i of the owner O_i whose age is still in the range $[L_i, R_i]$. The owner of the party is always in the correct age range, so by the property that all supervisors are at least as old as their subordinates, we know that all employees in the chain between O_i and S_i are also in the correct age range, so S_i will be invited to the party i.

Since all partygoers are connected, we can consider any of them as the owner of the party and the list of invited people will not change. Therefore, for every party i, consider that S_i now is the owner of that party. This makes the upper limit of age not relevant anymore, since we know S_i's manager is too old and none of the subordinates of S_i is too old.

Also, by the same reasoning we used to argue that all employees between S_i and O_i are in the age range, we can show that any employee that is a subordinate of S_i and is inside the age range will be invited, since all employees in the chain between them and S_i will also be in the age range.

Therefore, the condition to be invited to a party can be rephrased as "all subordinates of S_i with age at least L_i are invited to the party i", which is much simpler than the original conditions.

We can now finish the problem in several ways: one of them is to do a DFS on the tree and add parties owned by the current employee in some structure that allows to query, in logarithmic time, how many parties have a lower age limit that is lower than the age of the current employee (a binary indexed tree is a good choice, for example). When the DFS leaves node v, we remove all parties owned by v from this structure.

Problem F

Fastminton

It is enough to keep the current number of points and games for both players and simulate all of the rules exactly as described in the statement. See the official solutions for more details on how to implement all of the checks.

Problem G
Game Show!

Ricardo's decision to continue or stop depends on only the value of the future reward that Ricardo can get: if he knows that he can get a positive value of sbecs by continuing to play, he should continue; otherwise, if he is going to lose sbecs, he should stop.

This suggests a dynamic programming approach to solve this problem, in which dp_i is the best value Ricardo can get if the last i boxes are still left in the game. Before each play, Ricardo can choose to stop, which gets him no sbecs, or continue, which gets him the value of the current box and brings him to the situation in which $i - 1$ boxes are left:

$$dp_i = \max(0, dp_{i-1} + value_i)$$

Our final answer is the best value Ricardo can get if all N boxes are left, plus his initial balance of 100 sbecs, giving a final answer of $100 + dp_N$.

Problem H

SBC's Hangar

First observe that the number of combinations that have a final weight in the interval $[A, B]$ is the number of combinations that have weight at most B, minus the number of combinations that have weight at most $A - 1$. Therefore, it suffices to find an algorithm to calculate how many combinations have weight at most W, for some W.

We can think of some backtracking solution, in which we would process the boxes in order, decide if the current box is going to be included in the final combination or not, then for each of the two possibilities recurse for the next boxes. This is too slow, as it has an exponential complexity of $O(2^N)$.

However, the boxes follow a very particular property, that for any two boxes, the larger box weighs always at least twice as much as the smaller box. This implies that the weight of each box is greater than the combined weight of all lighter boxes.

We can prove this by induction: the property holds for the two lightest boxes, since the smaller box weighs at most half of the larger box. Now assume the property is true for the k-th smallest box. For the $k + 1$-th smallest box, the weight of all boxes that weigh less is the weight of the k-th box, plus the weight of all boxes lighter than the k-th box. As we know the combined weight of all boxes lighter than the k-th box is lighter than the k-th box, the combined weight of all boxes lighter than the $k + 1$-th box is less than two times the weight of box k, and therefore less than the weight of box $k + 1$.

This property can be used to cut several possibilities of box selection in the backtracking approach. Order the boxes from heaviest to lightest, and process the boxes in this order. Then:

- If the box is heavier than the maximum allowed remaining weight, it can never be in any configuration, so we can simply ignore this box and move on.
- If the box is lighter than the maximum allowed remaining weight, we now have a choice: we can include this box in the final group or not. However, note that if we don't include this box, then *any* combination of the remaining boxes is valid, because their combined weight is less than the weight of this box and this box is lighter than the maximum. So if we choose to include this box we continue processing the following boxes, and if we choose not to include this box, we can add $\binom{n}{k}$ to the answer, where n is the number of remaining boxes and k is the number of boxes that still need to be included.

We now never recurse twice for any box, and in fact the algorithm above can be implemented with a single loop. The complexity is $O(N)$.

Problem I
Interactivity

The size of the minimal set of queries is the number of leaves of the tree. This can be proven by a recursive approach (hard to understand proof):

A subtree is *fully determined*, meaning you can know all the values in this subtree with all queries, if:

1) Every children of the root of this subtree is *fully determined*. So you can just sum the value of the children to determine the value of the root.

2) The root value is determined by a query, and every children of the root of this subtree is *fully determined*, except one, which can become *fully determined* if you knew the value of its root. So you can subtract the value of the current root from the sum of values of the children that are determined, thus calculating the value of the child's root that is not *fully determined*.

It's easy to see that, in the 2nd case, you can't have more than one subtree that isn't *fully determined*, since it would only be possible to calculate the sum of its root's values, which would lead to multiple possible trees. Also it's not optimal to have both root and children subtrees *fully determined*, because the root value can be calculated in case you have the value of children's roots.

Then you can use this definition going up from leaves until the root of the tree for the proof:

1) Leaves that were queried are *fully determined*, and leaves that were not queried are not.

2) The internal node that is a parent of a leaf that is not *fully determined* (and all other children being *fully determined*, otherwise it can't be uniquely determined), has to be determined by a query to make its subtree *fully determined* or it is a subtree that is not *fully determined* but it can become in case you can know the value of its root (which is part of the case 2 of the definition).

This means that for each leaf that is not queried, another node on the path from it to the root must be queries to be able to *fully determine* the whole tree, and this query on the internal node uniquely compensates a single leaf not being queried. So the size of the minimal set of queries must be the number of leaves of the tree.

Using the definition we can formulate a dynamic programming solution, similar to Minimum Vertex Cover, to calculate the amount of different minimal sets we can have. Let $dp(u, k)$ be the total number of different minimal sets of the subtree of the node u that needs k extra queries to become *fully determined*. k can only be 0 or 1, since either the subtree is already *fully determined* or we need a single extra query to compensate one leaf still not compensated (if we have more queries not being compensated than this subtree can't become *fully determined*, as described).

Leaf case: $dp(u, 0) = 1$, $dp(u, 1) = 1$
Internal node case:

Let $S1 = \prod dp(v, 0)$, where v are children nodes of u, which means the sum of all children's arrangements in case every single one is *fully determined*.

Let $S2 = \sum S1 \times dp(v, 0)^{-1} \times dp(v, 1)$, which means the sum of all possible arrangements where only a single children's subtree needs an extra query to become *fully determined*.

$dp(u, 0) = S1 + S2$
$dp(u, 1) = S2$

Explanation:

If $k = 0$, the subtree is *fully determined*, so the definition is directly applied: either all children's subtrees are *fully determined* (which is $S1$), or a single child's subtree needs an extra query and we have to do this query at u, since $k = 0$, (which is $S2$).

Else ($k = 1$) one child's subtree must be needing an extra query (extra queries must come from children since it means leaves that don't have queries compensated) (which is $S2$).

The tricky case is to calculate $S2$ in modulo arithmetic. There can be a problem in case $S1$ is multiple of the modulo (in case $10^9 + 7$), so $S1$ would be zero, and $S2$ would also be zero, even when the only term multiple of the modulo is the swapped term.

To avoid this you can preprocess the prefixes and suffixes of $S1$:
$pre_i = \prod_0^{i-1} dp(v, 0)$
$suf_i = \prod_{i+1}^{c} dp(v, 0)$
and change the $S2$ expression to $S2 = \sum_{i=0}^{c} pre_{i-1} \times dp(v, 1) \times suf_{i+1}$.

Problem J
Collecting Data

This problem has many details, so it is very useful to eliminate some edge cases by treating them separately. First, if all values are the same, then all pairs are the same point and we have exactly one configuration. We can check if horizontal or vertical lines are possible by verifying if at least half of the values are the same, and then we can ignore horizontal and vertical lines for the remainder of the solution. Extra care must be taken with the case in which two values appear in the input with frequency of $\frac{N}{2}$ each, which is another case better handled separately.

After these edge cases are handled, we can now make two simplifying assumptions: for every configuration of points, there is exactly one line that goes through all points in this configuration, and for every value of x there is exactly one value of y that is in this line, and vice-versa. This is very useful because it helps avoid double-counting: we can now count for every possible line how many configurations form that line, and we know that a configuration can only be counted once as it only forms a single line.

Our strategy therefore is to form a list of candidate lines, and then for each line count in how many ways we can pair coordinates so that the line goes through all points (possibly zero, if it is impossible to make such a pairing).

The most straightforward strategy to choose candidate lines is to iterate through all ordered tuples of 4 values (v_a, v_b, v_c, v_d) chosen from the input, and take the line that goes through the points (v_a, v_b) and (v_c, v_d) as a candidate line. This will reach the right answer, but is too slow as there would potentially be $O(N^4)$ lines generated by this process.

To speed this up, we need to use the fact that line doesn't go through only two of the points, but through all of them. One idea is to filter only lines that were generated many times in the above process, which can work but is tricky to get right. A more direct approach is to note that since the line goes through all points, it must also go through the point that uses the smallest of the coordinates v_1. Similarly, it must also go through a point that uses the smallest coordinate v_2 that is greater than v_1. In this manner, it suffices to iterate only through tuples that contain both v_1 and v_2 in any position. This guarantees that the number of tuples that will be analyzed is $O(N^2)$.

Some notes on the correctness of the above approach: note that the point that goes through v_1 might be the same that goes through v_2, but this is not a problem since in this case we can take the remaining two values (v_3, v_4) to be any other point in the line. It is very important that we choose $v_1 \neq v_2$, as we need two distinct points to determine a line. Had we chosen $v_1 = v_2$, we could hit a case in which both of the points that use v_1 and v_2 are in the same location, which would make determining the line impossible. This is not a problem for the case $v_1 \neq v_2$, as if the points (v_1, v_a) and (v_b, v_2) are in the same location, this implies the point (v_1, v_2) is also in the line, so we will add this line as a candidate line later.

It remains to determine in how many ways each line can be formed. Here our second observation comes into play: since we are ignoring horizontal and vertical lines, for each x there is exactly one corresponding y in the line, and for each y there is exactly one corresponding x in the line. This means that for every value it has potentially two other values it could be paired to (depending on which coordinate it is being used as).

If we make a graph in which each node is a coordinate value and an edge means that two coordinates can be paired to form a point, each node will have degree at most 2. A graph in which every node has degree at most 2 is a union of paths and cycles. Paths can be paired greedily because the end of the path has only one possibility, so it is possible to repeatedly pair the end of the path until a contradiction is reached or the path is fully paired.

Cycles would be trickier, but for this particular graph we can show that all cycles are actually very small. Indeed, if we write the line as $y(x) = ax + b$, then a potential cycle of size 3 would imply $y(y(y(x))) = x$, which is a linear equation and therefore has only one solution for x, which must be

the fixed point of the line (the point of the line such that $y(x) = x$). We still have to be careful with edge cases when solving this linear equation, as it might involve a division by zero for two values of a, which have to be analyzed separately. If $a = 1$ then all points are fixed points if $b = 0$, or there are no fixed points if $b \neq 0$. If $a = -1$, then $y(y(x)) = x$ for all x.

Therefore we can only have paths and fixed points for most lines, unless $a = -1$ in which case there may also be cycles of length 2. For fixed points, it's enough to check if the value appears in the input an even or odd number of times; an odd number is impossible to pair and an even number can be paired in exactly one way.

The cycles of length 2 are an interesting case, because they are the only configuration that can be paired in more than one way: if both A and B appear in the input an equal number of times, then we can choose how many points of the form (A, B) and how many points of the form (B, A) will be formed. (Note that we treated cases that are a single 2-cycle separately earlier, so it is not a problem to select all points to be in the same location here).

There are $O(N^2)$ candidate lines, and we can test each candidate line in $O(N)$, so the overall complexity is $O(N^3)$.

Problem K
Between Us

Describe the final partition as a set of N variables $x_1, x_2, \ldots, x_N$, such that for every i, $x_i = 0$ if student i is in the first group and $x_i = 1$ if student i is in the second group.

Let the friends of student i be students $A, B, C, \ldots$. Analyzing the condition "the number of friends of student i that are in the same group as i is an odd number", we have two cases:

- Student i has an even number of friends in total

In this case, note that it does not matter for the condition in which group student i is in: either both groups have an odd number of friends of i, or both groups have an even number of friends of i. To have a valid partition, we want the first condition to hold (both groups have an odd number of friends of i), which is equivalent to saying that an odd number of the variables $x_A, x_B, x_C, \ldots$ is equal to 1. This can be written mathematically as $x_A \oplus x_B \oplus x_C \oplus \cdots = 1$, where $\oplus$ denotes the XOR operation (exclusive or).

- Student i has an odd number of friends in total

In this case, there is exactly one group with an odd number of friends of student i, so the position of student i is uniquely determined by the position of all of their friends. We can write the desired position for student i as $x_i = x_A \oplus x_B \oplus x_C \oplus \cdots$, or after rearranging the terms, $x_i \oplus x_A \oplus x_B \oplus x_C \oplus \cdots = 0$.

Therefore, for both cases, the condition can be expressed as an equation relating some of our variables. The exclusive or operation is addition under modulo 2, so those are also linear equations, and we can solve the system of linear equations using Gaussian elimination in $O(N^3)$. There is a valid partition if and only if the system has at least one solution.

Problem L
Lavaspar

The brute-force idea would be:

Create an auxiliary matrix A to count the number of words that covers each position.

Create an auxiliary matrix B to count, for the current word being processed, which positions have a matching for any anagram of it (this matrix has only zeroes and ones).

Then, for each word in the collection ($O(N)$), reset matrix B to zeroes ($O(L \times C)$), generate all anagrams of the word and ($O(P!)$), for each anagram, iterate in the original matrix for each initial position ($O(L \times C)$) and try to match to right, down and diagonal, settings 1 on every position that has a match ($O(P)$). After all anagrams, sum the matrix B into A, and go to next word ($O(L \times C)$).

This would lead to a $O(N \times (L \times C + (P! \times L \times C \times P))) = O(N \times P! \times P \times L \times C)$ complexity, which is totally above the time limit.

To improve it, we can't iterate on each anagram. We count the number of appearances of each letter in the current word $O(P)$ and we count the number of appearances of each letter in the matrix for each interval of length P ($O(L \times C \times P)$), going to right, left or diagonal, and for each interval we can compare if the amount of each letter is the same or not ($O(26)$). In case the interval matches, we have an anagram and we can update the auxiliary matrix B with ones ($O(P)$).

This would lead to a complexity of $O(26 \times N \times P^2 \times L \times C)$. Which may be enough to pass, but we can optimize some details. We just need to use one of the optimizations below, but we can use all of them together:

1) To update the auxiliary matrix B we can use the concept of sum in interval with prefix sums: sum 1 at the start and -1 one positions after the end. We have to split horizontal, vertical and diagonal tests to use this, though, but it would remove P from the complexity.

2) We could also improve the check. Instead of checking each one of the 26 letters for each interval, we can create a *letters_matching* variable that counts the amount of letters that have the same value between the interval being checked and the word, and if it's 26 we know that the interval matches some anagram of the word. To calculate it, when we iterate over the desired interval, we add each letter to the amount of that letter the interval has, then we can check if this value is equal to the value in our word values (then we sum 1 to *letters_matching*) or if it was equal before and it's not equal anymore (then we subtract 1). After going through every letter of the interval, *letters_matching* stores how many letters, from 'a' to 'z', have the same number of appearances between the interval and the current word. This removes the 26 from the complexity.

3) Finally we can also use a sliding window to not have to go through every interval of length P starting in every position of the matrix. For each line, start with an empty interval before the first letter of the line. Expand the interval by adding the letters one by one of this line until we have an interval of length P. Then we check if this interval is an anagram. Then add next letter and remove the first letter, which makes the current interval have length P. Then we check again if they match and repeat this adding/removing until we don't have more letters to add. Then do the same for columns and diagonals. This would remove P from the complexity.

Using all three optimizations we can go down to $O(N \times L \times C)$ complexity.

Problem M
Machine Gun

The angle that a machine gun covers, casts on the $x = 0$ vertical axis a certain interval. If we cast analogous angles from each initial enemy towards the left, each will cast its own interval over this line.

It can be verified that a machine gun kills an enemy precisely when the machine gun interval intersects the enemy interval.

Thus, queries can be answered using an interval tree `https://en.wikipedia.org/wiki/Interval_tree`, in $O((N + Q + m) \lg N)$ time, where m is the total number of killed enemies.

Alternatively, coordinate compression + persistent segment tree + sweep line + binary search can be used: after changing the plane coordinates so that machine guns span 90 degrees angles aligned with the coordinate axes, each query asks for the set of given points that are above and to the left of a certain point. Using sum-queries over rectangles and binary searching to quickly find where the points are, the full set of interesting points can be found. This solution has query complexity $\lg^2 N$ instead of $\lg N$, but it should probably be allowed to pass. The technique is more standard and well known, but the implementation is probably trickier than that of an interval tree. Note that this solution is in fact more general: it can handle queries of this kind with points anywhere at all in the plane, while the interval tree solution makes use of the special structure that these queries have (all points are above the $y = x$ line, and all queries corners are below that diagonal).

The offline version of the problem would be quite easier: when all queries are known in advance, they can be added as special points and then a single sweep line performed, keeping all seen enemies in a c++ set so that when each query is found, the answer can be read from the set using a simple lower bound operation.

However, this same idea can be made to work efficiently for the problem by implementing and using a persistent balanced binary search tree, giving a third solution to the problem.

Problem N
Number Multiplication

This problem allows two solutions:

The first one may annoy you, since you don't need to read the whole input to solve it.

Just take a fast factorization algorithm (let's say Pollard's rho), factorize every composite, put every prime factor in a set and print them in order.

The complexity of this solution is something like $O(N * \sqrt[4]{max(composites)} * log(max(composites)))$.

Before moving forward, try to think the other solution. As a clue, the expected complexity is $O(E + \sqrt{(max(composites))})$.

———————≀———————

The idea is to realize that we can do a simple $O(\sqrt{N})$ factorization algorithm going through all primes smaller than $\sqrt{N}$ just once.

We can keep an index `so_far`, the largest number we have tested and repeat the following:

1. If `so_far` is greater than $\sqrt{max(composites)}$ then take all composites values different than 1, and print them in order, they're all primes. There can be repeated values, so be careful. Exit
2. Take any composite node connected to the smallest prime node not yet discovered (remember they're ordered)
3. Starting in `so_far` check what's the next number that divides that composite (it'll be a prime). Check if `so_far` at any point becomes larger than $\sqrt{max(composites)}$, if it does, go back to 1.
4. Once found, print it (it's part of the answer), mark that node as discovered and divide every composite node connected to that prime node as many times as possible (the edge actually has that value). Increase `so_far` and repeat.

Problem O
Venusian Shuttle

Each position in the shuttle is going to receive sunlight in some parts of the path and not receive any sunlight in other parts. For now, we will not consider this detail and assume that a position receives sunlight during the whole trip, and find the position that receives the least amount of sunlight.

Each line in the shuttle path can be described by the parameters L, the length, and α, the orientation of the line. Someone sitting in a position x of the shuttle receives, during this line, $L\cos(x+\alpha)$ units of sunlight. The total sunlight is therefore given by a sum of several cosines.

It is now very useful to know that a sum of several cosines with same frequency is still a cosine multiplied by some constant, that is, we can write

$$A\cos(x+\alpha) + B\cos(x+\beta) + \cdots = C\cos(x+\gamma)$$

for some appropriate value of C and γ. There are several ways to reach this conclusion, including geometrical approaches and using complex numbers. A direct algebraic way is to use the expression for cosine of a sum:

$$\cos(A+B) = \cos A\cos B - \sin A\sin B$$

This expression lets us rewrite $A\cos(x+\alpha)$ as $B\cos x + C\sin x$, which makes adding two cosines with different orientations easy as we can now separately add the coefficients for $\cos x$ and $\sin x$. We can also use the same expression in reverse to convert our final expression $B\cos x + C\sin x$ back into a single cosine, and from there finding the minimum value of the expression is simple.

Remembering that not all positions receive sunlight in all parts of the path, we need to include in our sum only the cosines from parts of the path in which a position x receives sunlight. We can note that a line of the path provides sunlight to an interval of angles $[-\frac{\pi}{2}-\alpha, \frac{\pi}{2}-\alpha]$. We can sort the endpoints of those intervals and use a line sweep on them to consider for every interval of x values, only the cosines that are relevant to those values of x.

Finally, we can note that as each sum only includes positive terms, the minimum for each interval $[x_1, x_2]$ must be in x_1 or x_2, so it suffices to compute the sum for both extremes of all intervals hit by the line sweep.

Gran Premio de México 2020 - Primera fecha

31 de Octubre del 2020

Libro de soluciones

Este documento contiene las soluciones esperadas para los 14 problemas usados durante la competencia.

Advanced Recommendation System

Las operaciones de este problema son claras y muy directas de implementar. Se puede observar que si conocieramos los valores $P_{i,j}$ tal que $P_{i,j}$ es la preferencia que se tiene del producto j sobre el producto i, se podría responder el producto con mayor preferencia para un valor i encontrando el valor k tal que $P_{i,k}$ es máximo. De la misma manera obtener el par con la máxima preferencia entre todos los pares posibles existentes se podría obtener evaluando todo los valores $P_{i,j}$ posibles y encontrar de esta manera el máximo.

Una manera de implementar es almacenar P como una matriz, así el valor $P[i][j]$ representa la preferencia de j sobre i, y para cada línea en la entrada de la forma $R\ i\ j$, aumentaremos $P[i][j]$ en 1. Almacenando las preferencias de esta manera podemos actualizar y obtener la preferencia de un producto j sobre el producto i en $O(1)$, sin embargo para las consultas de tipo Q la complejidad sería $O(N)$ y la complejidad para las consultas tipo B sería de $O(N^2)$, lo cual excedería el tiempo límite de ejecución.

Una alternativa más eficiente es utilizar una estructura de datos que nos permita realizar las consultas con una complejidad computacional menor. Seguiremos utilizando la matriz P para almacenar la preferencia que hay para el producto j sobre i, pero, adicional a eso mantendremos un *set* para cada uno de los productos i que almacene la preferencia para cada producto j que se ha visto. De esta manera, se puede obtener el producto con mayor preferencia para i consultando el máximo valor almacenado en el *set*, lo cual toma $O(logN)$. Para responder la consulta de tipo B mantendremos otro *set* donde se almacene el par i,j, y el $P[i][j]$, así consultas el máximo entre todos se puede obtener consultando el máximo almacenado en el set en $O(log(N^2))$

En la implementación hay que tener cuidado al actualizar todas las referencias que se están almacenando al recibir las consultas que aumentan la preferencia de un producto sobre otro, por ejemplo, sea S_i el *set* que almacena los productos que tienen preferencia sobre i, al recibir una consulta del tipo $R\ i\ j$, es necesarioi quitar el elemento $(j, P[i][j])$ de S_i, agregar el elemento $(j, P[i][j]+1)$ y actualizar $P[i][j] = P[i][j] + 1$, de una manera similar se debe actualizar el set para responder la consulta que pide el par con mayor preferencia.

Bus Line

Por las características que se mencionan en el problema, podemos identificar que para un usuario del camión que bajará en la parada b_i y se encuentra en la posición i, al bajar por la puerta trasera del camión tendrá que cambiar posición con todos los usuarios que se encuentren en una posición j tal que $0 \leq j < i$ y $b_j > b_i$, entonces, sea $T(i)$ esta cantidad para cada i, entonces para calcular la cantidad mínima d e intercambios que se deben realizar cuando solo está permitido bajar por la puerta trasera debemos calcular $A = \sum_{i=0}^{N-1} T(i)$.

De la misma manera podemos calcular $F(i)$ como la cantidad de intercambios que debe dar la persona en la posición i de la fila para salir por la puerta frontal, estas serán la cantidad de posiciones j tal que $i < j < N$ y $b_j > b_i$. Para calcular entonces la cantidad mínima de intercambios que se deben realizar cuando las dos puertas están permitidas para bajar del camión debemos considerar por cuál puerta cada usuario del camión tendría que hacer menos intercambios y entonces debemos calcular $B = \sum_{i=0}^{N-1} min(T(i), F(i))$.

Calcular los valores de $T(i)$, y $F(i)$ revisando para cada valor i todos los posibles j hace que la implementación sea de complejidad $O(N^2)$, para reducir la complejidad se debe utilizar una estructura de datos que permita calcular estos valores de una manera eficiente. El problema para calcular $T(i)$ y $F(i)$ se puede convertir en un problema de conteo de frecuencias, en donde $T(i) = i - t(i)$ siendo $t(i)$ la cantidad de usuarios que subieron al camion antes que i y ya han bajado o bajan en la misma estación que i (nótese que este valor siempre será el mismo que para la definición de T(i) que se había definido previamente). El problema de contar cuantos elementos $b_j <= b_i$ con $j < i$ $(t(i))$ es un problema que se resuelve de manera eficiente usando un BIT (Binary Indexed Tree) o Fenwick Tree. Entonces hay que calcular los valores A y B usando esta estructura para cumplir con el límite de tiempo establecido.

Continuous Replacement Algorithm

Una incosistencia en el enunciado del problema y la entrada hacía que este problema pareciera más difícil de lo que era en realidad. Una vez se clarifico que aunque Bob y Alice buscan la cadena más pequeña que se puede obtener, en cada iteración podría sustituir una cadena de mayor longitud por una de mayor comenzarón a llegar varios envíos aceptados.

Se puede notar que por la manera en la que se realizan las sustituciones se genera un grafo que tiene como vertices las palabras y una arista entre dos palabras si estas se pueden sustituir con la información que se da en la entrada. Dado esto se puede observar tambien que todas las palabras que se encuentran en una misma componente conexa del grafo se pueden sustituir entre si, entonces, la cadena más pequeña en tamaño y lexicográficamente menor que se puede obtener partiendo del mensaje original es la que se obtiene al sustituir cada palabra con la palabra de tamaño más pequeño y lexicográficamente menor en la componente conexa que se encuentra. Esto se puede hacer utilizando una el algoritmo de union find para encontrar de manera eficiente la palabra por la que se debe sustituir cada palabra en el mensaje original.

Detailed Sorting Machine

Hay que observar que nunca será necesario mover la misma caja más de una vez, además considerando que la caja que tiene el valor más pequeño siempre estará ordenada, entonces, nuestra solución siempre será menor al número de cajas.

Tomando la segunda observación que hicimos en el parrafo anterior, si la caja con el segundo valor más pequeño se encuentra en una posicón j tal que $j > i$ donde i es la posición en la que se encuentra la caja con el valor más pequeño, entonces, todas las cajas en las posiciones $k = i + 1, .., j - 1$ deberán moverse para ordenar las cajas. Siguiendo con la misma idea podemos observar que la cantidad de movimientos a hacer es igual a la cantidad de cajas que hay entre las cajas que ya se encontrarían ordenadas si no tuvieran cajas entre ellas. Análogamente sería el total de cajas $N - s$ donde s son las cajas que ya se encontrarían ordenadas. Esto es mínimo porque se garantiza que solo se mueve 1 vez cada caja que deba ser movida, para lograr el orden primero se debería mover la caja con el $(s + 1)$-ésimo valor, después la $(s + 2)$-ésimo, hasta haber movido la caja con el valor más grande.

Una manera de implementar esto es ordenar las cajas de menor a mayor e identificar cuales cajas ya estarían ordenadas si no tuvieran cajas intermedias:

```
S=sort(cajas);
si=0
for i = 0; i < N; i++:
    si S[si] == cajas[i]
        si++
print(N-si)
```

Enthusiastic Mathematics Challenge

Los límites sugieren probar cada una de las permutaciones posibles de la cadena dibujada por Bob. Para cada permutación posible se puede romper en los 3 números tomando dos indices i y j donde $0 \leq i < j \leq |S|$, de modo que el primer número está dado por las posiciones en el rango $[0, i]$, el segundo en el rango $[i+1, j]$ y el tercero en el rango $[j+1, |S|)$. De esta manera se prueban todos los posibles conjuntos de 3 números que se pudieran encontrar en cualquier orden en la cadena dibujada por Bob. Entonces, para cada permutación se evaluan en cada par de indices posibles que los tres números formados sean primos, y que su producto sea menor al que ya se había encontrado, al finalizar el proceso se regresa el producto mas pequeño que se encontró o "Bob lies"en caso que no se hayan encontrado 3 números primos en ninguna permutación.

Para eficientar el cálculo se puede observar que para poder romper la cadena en 3 números, ningun número excederá el valor 999999, entonces se puede evaluar si un número es primo auxiliado de una criba que calcule todos los primos menores a 10^6.

Feeding The Judges

Después de que se ha procesado la entrada y se almacenan los datos, es necesario definir un modelo matemática que ayude a resolver el problema, sea a_i la cantidad de platos que pidieron los jueces para el platillo i , y x_i el número de platos que se compran para ese platillo al realizar una orden, donde $1 \leq i \leq k$ y k es la cantidad de platillos que el restaurant ofrece. entonces:

$$a_i \leq x_i \leq b_i$$
$$x_1 + x_2 + \cdots + x_k = N \tag{1}$$

Si introducimos un nuevo conjunto de variables $y_i = x_i - a_i$. Entonces: $x_i = y_i + a_i$ y la ecuación (1) se puede reescribir de la siguiente forma:

$$(y_1 + a_1) + (y_2 + a_2) + \cdots + (y_k + a_k) = N$$
$$y_1 + y_2 + \cdots + y_k = n - A \tag{2}$$

Donde $A = a_1 + a_2 + \cdots + a_k$; y además, cada y_i cumple que $0 \leq y_i \leq b_i - a_i$.

Con estas definiciones se puede observar que la cantidad de soluciones enteras que existen para la ecuación (2), es la cantidad de órdenes diferentes que se pueden hacer para satisfacer la orden de comida de los jueces, el cual cuando no hay restricciones para y_i se puede resolver con separadores con la siguiente fórmula:

$$S = \binom{n - A + (k-1)}{k-1}$$

Para incluir las restricciones de los valores de y_i, usaremos el principio de inclusión y exclusión, sea P_i la propiedad que $y_i \geq b_i - a_i + 1$. Entonces, el número de soluciones para (2), donde se satisface P_1 ($y_1 \geq b_1 - a_1 + 1$), usando separadores, son:

$$\binom{n - A - (b_1 - a_1 + 1) + (k-1)}{k-1}$$

Si llamamos S_1 al número de soluciones para (2) que satisfacen al menos una de las propiedades en P, considerando que existen k propiedades, entonces:

$$S_1 = \sum_{i=1}^{k} \binom{n - A - (b_i - a_i + 1) + (k-1)}{k-1}$$

Bajo la misma lógica, sea S_2 el número de soluciones para (2) que satisfacen al menos dos propiedades de P,

$$S_2 = \sum_{w \in \binom{|k|}{2}} \binom{n - A - \sum_{i \in w}(b_i - a_i + 1) + (k-1)}{k-1}$$

Y en general:

$$S_j = \sum_{w \in \binom{|k|}{j}} \binom{n - A - \sum_{i \in w}(b_i - a_i + 1) + (k-1)}{k-1}$$

Donde $1 \leq j \leq k$.

Aplicando entonces el principio de inclusión y exclusión, la cantidad de soluciones de (2) que no tienen ninguna propiedad en P (esto es, todas las soluciones donde $y_i \leq b_i - a_i$) es:

$$Answer = S - S_1 + S_2 - \cdots + (-1)^k S_k$$

Para poder calcular los valores de S_j es necesario calcular considerando todos los posibles subconjuntos de las propiedades en P, esto se puede hacer con máscaras de bits en $O(2^k)$. Para cada uno de estos subconjuntos es necesario calcular $\sum_{i \in w}(b_i - a_i + 1)$, lo que tiene complejidad $O(k)$ en el peor caso. Por lo que la complejidad total del algoritmo será $O(k * 2^k)$

Golf Score

La solución a este problema es muy directa, de hecho, era considerado el problema más sencillo del conjunto de problemas de la competencia.

Para calcular la cantidad de tiros que el golfista realizó se debe calcular la cantidad de tiros que realizó en cada hoyo. Sea P_i la cantidad de tiros para tener "par"en el hoyo i, entonces, la cantidad de tiros en el hoyo (S_i) según el término utilizado será:

Término	Tiros (S_i)
Hole in one	1
Condor	$P_i - 4$
Albatross	$P_i - 3$
Eagle	$P_i - 2$
Birdie	$P_i - 1$
Par	P_i
Bogey	$P_i + 1$
Double bogey	$P_i + 2$
Triple bogey	$P_i + 3$

Entonces, la solución requiere calcular los valores S_i dados P_i y el término usado en golf para el score que tuvo el golfista, y sumar los 18 valores que se obtuvieron.

Halloween

Este problema es un problema clásico de programación dinámica. Sea M(i) una función que calcula la máxima cantidad de dulces que Baker puede obtener al haber visitado las primeras i casas. Podemos observar que cuándo Baker visita la casa i, la cantidad de dulces que puede obtener depende de si llega a la casa después de haber visitado la casa $i-1$ o después de haber visitado la casa $i-2$. En el primer caso, Baker no recibiría ningún dulce de la casa i, entonces tendría $M(i-1)$ dulces. Para el segundo caso Baker puede recibir los $c(i)$ dulces que le entregaría la casa i dado que no visitó a su vecino, en este caso Baker podría tener $M(i-2)+c(i)$ dulces.

Con estas observaciones entonces podemos definir la función $M(i)$ con la siguiente relación de recurrencia.

$$f(x) = \begin{cases} c(i) & si \quad i = 0 \\ max(c(i-1), c(i)) & si \quad i = 1 \\ max(M(i-1), M(i-2)+c(i)) & si \quad i \geq 2 \end{cases}$$

La cantidad máxima de dulces que Baker puede obtener será M(N-1), donde N es el número de casas en el vecindario

Is It Secure Enough?

La solución a este problema es directa, básicamente hay que realizar la validación que se pide en la descripción. Para cada una de las cadenas que se dan en la entrada, hay que evaluar cada una de las reglas que se mencionan en la descripción e imprimir la salida en el formato establecido dependiendo de cuántas reglas se cumplieron. Durante la competencia la regla de que no debería haber dígitos consecutivos juntos causo confusión, pero una vez aclarado el número de envíos correctos aumento considerablemente.

King's Dilemma

No es difícil observar que el problema describe un grafo no dirigido donde los vértices son las ciudades y las aristas son los caminos que unen a las ciudades.

Podemos observar que si la única restricción impuesta por el rey fuera que las ciudades estén conectadas después de reparar los caminos, entonces, la solución sería encontrar un árbol que recubra el grafo (árbol de expansión) y después seleccionar un conjunto arbitrario de caminos hasta haber elegido las K autopistas para reparar.

Agregar la restricción de necesidad que tiene un camino con respecto a otro no cambia tanto la solución a lo descrito en el parrafo anterior, si se ordenan los caminos por necesidad, esto es, primero los caminos de la ciudad 1, despues los de la ciudad 2, ..., etc. Al tener los caminos ordenados de esta manera buscamos un árbol de expansión usando el algoritmo de Kruskal, este árbol de expansión garantiza que todas las ciudades estén conectadas, posterior a este se seleccionan los $K - (N - 1)$ caminos con mayor necesidad que no son parte del árbol de expansión. Esto funciona porque si suponemos que hay un camino c que debería estar en la solución y no lo seleccionamos, entonces significa que $u_c < u_k$ o que $u_c = u_k$ y $v_c < v_k$ para algún camino que está en la solución, y entonces, nuestro algoritmo hubiera considerado c antes k, por lo que ese camino c no existe (a menos que se implemente mal el ordenamiento).

Hay que considerar los casos en los que no es posible seleccionar los caminos para satisfacer las restricciones del rey: cuando se piden más caminos para arreglar que los que hay en el reino, cuando no se puede encontrar un árbol de expansión en el grafo que modela el reino (el grafo de la entrada no es conexo).

Let's run!

Se busca el valor de t que maximiza la $(At^2 + Bt + C) - (Dt + E) = At^2 + (B - D)t + (C - E)$, esto sucede cuando $2At + B - D = 0$ y entonces $t = (D - B)/(2A)$. Sustituyendo las variables podemos obtener los valores de v_1 y v_2, y recordando que estos valores deben ser enteros entonces $v_1 = \lfloor v_1 \rfloor$ y $v_2 = \lfloor v_2 \rfloor$.

Entonces estamos interesados en encontrar el valor entero positivo del minuto m más pequeño tal que : $m * v_1 \equiv m * v_2 + 10(mod 188888881)$, lo cual es lo mismo que $m(v_1 - v_2 + 188888881) \equiv 10(mod 188888881)$. Multiplicando ambos lados de esta última equivalencia por el inverso multiplicativo modular de $(v_1 - v_2 + 188888881)$ tenemos que:

$$m \equiv 10 * (v_1 - v_2 + 188888881)^{-1} (mod 188888881)$$

Entonces, el problema se reduce a calcular el valor de $(v_1 - v_2 + 188888881)^{-1}$ y con este obtener el valor de m como se describe en la ecuación anterior. Se puede obtener el valor del inverso multiplicativo modular ya sea por el algoritmo extendido de euclides o con el pequeño teorema de Fermat.

Se puede observar además que claramente siempre se puede obtener la foto.

MEOW

Para facilitar la descripción digamos que $M = u_m - l_m + 1$, $E = u_e - l_e + 1$, $O = u_o - l_o + 1$, y $W = u_w - l_w + 1$, representan la cantidad de diferentes valores que puede tomar cada una de las letras en el lenguaje de los gatos.

Se puede observar que la cantidad de palabras que tendrá el lenguaje de los gatos se puede calcular como $L = M * E * O * W$, y se podría pensar que el lenguaje de Baker tendría $B = L^2$ palabras diferentes. Sin embargo, el hecho de que los valores para l_m, l_e, l_o, l_w puede ser igual a 0 hace que sea posible que en las L^2 palabras haya algunas repetidas. Por ejemplo, supongamos que $l_m > 0$ y $l_e = 0$, $l_o = 0$, y $l_w = 0$, el calcular el lenguaje donde $B = L^2$ se estarán repitiendo las palabras donde no hay ninguna letra entre las m de la primer palabra y las m de la segunda, para este caso podemos observar que se están contando $(M-1)^2 * E * O * W$ palabras de más.

La solución se puede calcular entonces como $L^2 - R$ donde R es el total de palabras que se están contando más de una vez en L^2.

New Pump System

Este problema fue utilizado en este set tal como apareció en la segunda fecha del Gran Premio de México 2017.

Dado que se garantiza que la manera de acomodar las tuberías será única, entonces hay que buscar por una manera de acomodar las tuberías de modo que se conecten con las restricciones que se describen. Esto se hace haciendo un backtracking en el que se agrega una tubería siempre que se pueda poner y conecte alguna de las tuberías que no están conectadas. Cuando se pone una tubería y se han conectado todas, no hay más tuberías por poner y se encontró un resultado. Algo importante es tener cuidado de no repetir escenarios que ya se hayan probado para poder acortar un poco el tiempo de ejecución. Durante la implementación se debe tener cuidado de probar en cada lugar que se pueda poner una tubería, con las diferentes opciones que se pueden poner, en general no es difícil idear una estrategia para probar las opciones posibles de tuberías para encontrar la solución, pero si es bastante truculento y propenso a errores.