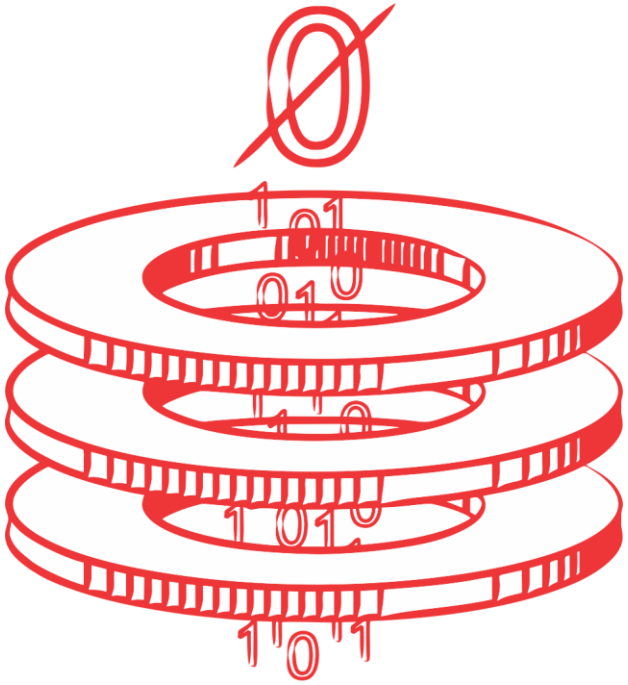
EDITORIAL CUARENTENA 2.0

Programación Competitiva UFPS
Semillero de Investigación en Linux y Desarrollo de Software Libre – SILUX



Ingeniería de Sistemas

"Educación y Tecnología con Compromiso Social"
Acreditado de Alta Calidad

http://ingsistemas.ufps.edu.co

ANTES DE COMENZAR

- Las soluciones de todos los ejercicios, así como los casos de prueba secretos, se pueden encontrar en: https://github.com/carlosfernandoufps/Solucionario-Cuarentena-2.0

A.

- Temática: AD-HOC
- Seguir leyendo strings y por cada uno iterarlo y contar el número de 't' y 'T'.

B.

- Temática: Matemáticas
- Para poder formar un triángulo se debe asegurar que la suma de dos lados no sea mayor que el tercero.
- El área de un triángulo se puede hallar con la fórmula de Herón. Si decimos que a, b, c son las longitudes de los lados del triángulo, su área se halla con la fórmula:

- $\sqrt{s(s-a)(s-b)(s-c)}$, *siendo s:*

- $s = \frac{a+b+c}{2}$

C.

- Témática: AD-HOC
- El dinero total de cada tesoro se consigue multiplicando la cantidad del tesoro i con su valor por unidad.
- Entonces solo es iterar de menor a mayor índice y cada vez que se encuentre un dinero total mayor al que se tiene, se guarda ese ID.

D.

- Temática: Matemáticas
- Fuerza Bruta dará Tiempo Límite Excedido.
- Sabemos que:
- P: Número de ejercicios que hace diariamente la persona.
- K: Número de ejercicios que lleva resueltos el tio.
- N: Número de ejercicios que lleva resueltos la persona.
- Definimos:
- D: Cantidad de días necesarios para que el tío tenga más ejercicios que la persona. Esto nos da una desigualdad:

$$K + \sum_{i=1}^{D} i > N + P * D$$

- Primero solucionamos la sumatoria y volvemos una ecuación la desigualdad

D.

$$K + \frac{D * (D + 1)}{2} = N + P * D$$

- Despejamos D:

$$2K + D^2 + D = 2N + 2P * D$$

$$D^2 + D - 2P * D = 2N - 2K$$

$$D^2 + D(1 - 2P) + 2K - 2N = 0$$

- Con esto nos queda una ecuación cuadrática donde a = 1, b = 1 + 2P, c = 2K – 2N.

$$D = \frac{2P - 1 \pm \sqrt{(1 - 2P)^2 - 4(1)(2K - 2N)}}{2}$$

D.

- Acomodamos la ecuación;

$$D = \frac{2P - 1 \pm \sqrt{1 - 4P + 4P^2 - 8K + 8N}}{2}$$

- Recordemos que las restricciones del programa nos dice que 0 < P < K < N. Con un poco de perspicacia y haciendo unos pequeños casitos a mano nos podemos fijar que la versión negativa de la ecuación nos dará respuestas negativas, así que sólo usaremos la versión positiva. Por lo tanto nuestra ecuación final nos queda:

$$D = \frac{2P - 1 + \sqrt{1 - 4P + 4P^2 - 8K + 8N}}{2}$$

D.

- Sin embargo, no olvidemos que la cantidad esperada es un número entero, y que la ecuación en realidad era una desigualdad, lo que haremos será convertirlo a entero usando la función techo.

- Recordemos que esta función redondea los números a su versión entera mayor. Ej: techo(5.127) = 6.

- Por otra parte, puede existir el improbable caso en que D nos dé un número exacto como 5.000000. En este caso la función techo devolverá 5 porque no hay nada que redondear hacia arriba. Pero recordemos que sí hay que redondear hacia arriba porque no queremos una igualdad, deseamos que la cantidad del tío sea mayor: Es por esto que un pequeño truco es sumarle un EPSILON muy pequeño, algo como 0.0000000001, de tal forma que ahora sí lo redondeará a 6 y no alterará a los demás resultados.

E.

- Temática: Búsqueda Completa Iterativa.
- Lo que haremos será revisar todos los posibles cuadrados que existen en la matriz y quedarnos con aquella que no tenga caracteres diferentes y será máxima.
- Para hacerlo primero necesitamos dos 'for' para recorrer cada casilla de la matriz.
- Por cada casilla de esta matriz fijaremos un entero K. Con este entero decimos que a partir de ahí visitaremos los cuadrados que estén a una distancia K hacia abajo y hacia la derecha siempre y cuando no se salga de la matriz.

E.

Estoy aquí.	K = 1	K = 2	K = 3
K = 1	K = 1	K = 2	K = 3
K = 2	K = 2	K = 2	K = 3
K = 3	K = 3	K = 3	K = 3

		Estoy aquí	K = 1
		K = 1	K = 1

E.

- Ahora que definimos el cuadrado, lo que haremos será recorrer cada casilla de este cuadrado y verificar que todos tengan el mismo caracter. Si esto se cumple entonces es un cuadrado válido y actualizaremos nuestra respuesta. Para esto necesitamos otros dos for.

- Existen formas más óptimas de hacer este ejercicio, pero como n, m <= 30 esto nos deja O(n^5) y como 30^5 < 10^8 sabemos que pasará en tiempo.

- En resumen dos for para recorrer cada casilla, un for para K, y dos for para recorrer las casillas marcadas por K.

F.

- Temática: Búsqueda Completa Recursiva (Backtracking).
- Abstrayendo el enunciado lo que nos pide es decir si existe un subconjunto de los N números tal que sumando y/o restando algunos se consigue el número Qi.
- Podemos entonces definir nuestra función recursiva de la siguiente manera:
- $$f(i, acum)\begin{cases} valor[acum] = true, & i == N \\ f(i+1, acum) \\ f(i+1, acum + arr[i]) \\ f(i+1, acum - arr[i]) \end{cases}$$
- Se explica la función a continuación.

F.

- Si el índice que estoy usando ha llegado al final (N) significa que ya terminé de recorrer todos los números y llevo un acumulado de acum. Por lo tanto vamos a decir que ese valor es alcanzable con los números dados. Esta información la guardamos en un arreglo de booleanos.

- Si el índice aun no es N es porque sigo recorriendo. Entonces en cada paso tengo tres opciones:
  - Borrar el número, es decir no tenerlo en cuenta así que mi acumulado seguirá igual.
  - Sumar el número, entonces mi acumulado será lo que llevo hasta el momento más el número. Este número lo guardé previamente en un array que llamé arr[].
  - Restar el número, entonces mi acumulado será lo que llevo hasta el momento menos el número.

F.

- Con esto nos garantizamos que la función se irá por todos los posibles caminos y hará todos los subconjuntos variando la suma y resta. Fijémonos que por cada paso hacemos tres llamados recursivos hasta llegar a N. Eso nos deja con una complejidad de O(3^N) que para N = 10 pasa sobrado en tiempo.

- ¿Por qué utilizamos un arreglo de booleanos y no lanzamos la recursión por cada consulta Qi? Bueno, consumiríamos más tiempo. Como son hasta 500 consultas estaríamos haciendo exactamente la misma recursión 500 veces, esto nos da peligro de TLE. Es más fácil lanzar la recursión una única vez y marcar los números que alcanza. Luego solo es iterar las consultas y verificar si están o no marcadas.

F.

- Hay que tener cuidado porque como también hay restas, no podré acceder a posiciones negativas dentro de un arreglo. Como son 10 números y por mínimo cada valor será de -100 el mínimo valor total será de -1000. Lo que podemos hacer es entonces sumarle un valor seguro a la consulta de tal forma que nunca quede negativo. Por ejemplo: valor[acum + 2000] = true. Con esto nunca actualizaremos valores negativos.

- Lo mismo hacemos para cuando consultemos si es posible o no. Ejemplo: if(valor[2000 + Qi] == true) entonces es posible, sino es imposible.

- Claramente en el main llamamos la función para f(0, 0). Porque empezamos desde el índice cero y llevamos un acumulado de cero.

G.

- Temática: Greedy.
- Debemos encontrar una forma inteligente de distribuir los tapetes de tal forma que minimicemos la cantidad de empaques que se usen. Hagamos algunas observaciones:

- Si nos piden K tapetes 6x6 sí o sí tengo que usar K empaques para estos.
- Si nos piden K tapetes 5x5 sí o sí tengo que usar K empaques. Sin embargo nos quedarían 11 casillas sobrantes que solo podemos tapar con tapetes 1x1. Entonces lo que haremos es que por cada tapete 5x5 que usemos gastaremos también 11 tapetes 1x1.
- Si nos piden K tapetes 4x4 sí o sí tengo que usar K empaques. Sin embargo nos quedan sobrando 20 casillas. En estas 20 casillas caben perfectamente 5 tapetes 2x2 o 20 tapetes 1x1. Obviamente es mejor gastar los tapetes 2x2 siempre que sea posible.

G.

- En un empaque nos caben perfectamente 4 tapetes de 3x3. Entonces, si nos piden K tapetes de 3x3 usaremos K/4 empaques y nos sobraran K%4 tapetes de 3x3.
  - Si nos sobran 3 tapetes de 3x3 los pondremos en un empaque y hay espacio para colocar 1 tapete de 2x2 y 5 de 1x1.
  - Si nos sobran 2 tapetes de 3x3 los pondremos en un empaque y habrá espacio para 3 tapetes de 2x2 y 6 de 1x1.
  - Si nos sobra 1 tapete de 3x3 los pondremos en un empaque y habrá espacio para 5 tapetes de 2x2 y 7 tapetes de 1x1.

- Si en este paso aún nos sobran tapetes de 2x2 vemos que en un empaque caben perfectamente 9. Así que si nos piden K tapetes usaremos K/9 empaques y nos sobrarán K%9 tapetes. Los que nos sobran los podemos acomodar en un tapete y el resto de casillas disponibles rellenarlo con tapetes de 1x1.

G.

- Si en este paso aún nos sobra tapetes de 1x1 usaremos K/36 empaques. Si la división no es exacta simplemente usamos un empaque más para el resto.

- Esta estrategia es la forma óptima ☺

- Hacer los dibujitos de la distribución puede hacer entender mejor el problema.

H.

- Temática: Búsqueda Binaria (o programación dinámica) + Greedy.
- Este ejercicio es engañosamente complicado y tiene varios casos bastante trickys.
- La primera parte es minimizar la máxima suma de los rangos después de partirlo en K partes. ¿Qué hacemos?
- Esta parte se puede resolver por dos caminos: Una es con búsqueda binaria y la otra con programación dinámica, aunque para mí se hace más fácil implementar la DP que la búsqueda binaria, aquí explicaremos con búsqueda binaria. Sin embargo, tener en cuenta que el código de mi solución para esta parte está con programación dinámica.

H.

- Trabajemos con un ejemplo para visualizar mejor la estrategia: El primer caso de ejemplo: {100 200 300 400 500 600 700 800 900} y 3 particiones.

- Si decimos que cada día se le asignará a lo sumo 100 minutos entonces necesitaríamos muchos más días, y algunos días tendrían más que este límite.

- Si decimos que cada día se le asignará a lo sumo 5000 minutos entonces en el primer día podría resolver todas las tareas y estaría desperdiciando los demás días.

- De esto deducimos que:
  - Si me sobraron días entonces trato de que cada día tenga un máximo acumulado de minutos menor
  - Si me hicieron falta días entonces trato de que cada día tenga un máximo acumulado de minutos mayor.

H.

- Ya con esto tenemos planteada la búsqueda binaria. Elegimos un rango entre 0 y la suma de los minutos entre todos los días hasta encontrar el mínimo máximo de minutos que se puede asignar a un día tal que se usen todos los días.

- Después de encontrar este rango, viene lo más difícil: ¿Cómo asignar cuántos minutos para cada día para satisfacer las condiciones que nos da el problema?

- Miremos algunos casos tricky junto a su mínimo máximo:
  - 9 8 1 7 6 2 3 4 5 para 3 particiones su óptimo es 17.
    - 9 8 / 1 7 6 / 2 3 4 5
  - 1000000 1 1 1 1 1 para 3 particiones su óptimo es 1000000
    - 1000000 / 1 / 1 1 1 1
  - Los otros casos de prueba que se muestran en la descripción del problema.

H.

- Una forma de hacer esta asignación es iterando desde el final hasta el inicio.
  - Si ya alcancé el máximo entonces coloco un slash y termino un día.
  - Si la cantidad de días que tengo por asignar es igual a la cantidad de tareas que me restan entonces sí o sí debo colocar un slash y acabar el día con una única tarea, de lo contrario me sobrarán particiones.

- Haciendo esto nos aseguramos que dejamos la mayor parte del trabajo para el final. Hacer casos de prueba duros es parte de ser un buen competidor y es esencial para ejercicios Greedy.

- La mejor forma de saber si una estrategia no funciona es encontrar esos casos que rompen el algoritmo.

GRACIAS

Carlos Calderón – Líder grupo de programación competitiva UFPS

carlosfernandocr@ufps.edu.co