Universidad Nacional Autónoma de México
Facultad de Ciencias
Análisis de Algoritmos | 7083
Tarea 5 : | Tarea final
Sosa Romo Juan Mario | 320051926
13/11/2024

1. **Pruebe que el segundo elemento más chico de una lista de n elementos puede encontrarse con $n + \lceil \log_2 n \rceil - 2$ comparaciones.**

   Usando arbol de torneo

   Lo primero que nos va a resultar muy útil es conseguir el arbol de torneo de la lista, este arbol es un arbol binario balanceado donde cada nodo interno es el ganador de sus dos hijos, es decir, el nodo con el valor mas chico de sus dos hijos.

   Entonces, comenzamos metiendo cada valor de la lista en una hoja del arbol, luego, en cada nivel del arbol, vamos a comparar los nodos de izquierda a derecha y vamos a ir subiendo el nodo mas chico, al final, el nodo raíz será el elemento mas chico de la lista. Como en el primer nivel tenemos n nodos hacemos $n/2$ comparaciones, en el siguiente haremos $n/4$ comparaciones, y así sucesivamente, entonces la cantidad de comparaciones que hacemos es $n - 1$ (cuando $n/2^k = 1$ no hay comparación).

   Ahora tenemos una EDD con el mas chico en su raiz y como ya notamos en la tarea 1 (o 2 xd) es que para encontrar el segundo elemento mas chico basta con checar alguno que haya perdido directamente contra el mas chico, esto es porque si un nodo perdió contra el mas chico, entonces no puede ser el mas chico, y a su vez si es el segundo mas chico le gano a los que se comprararon con el.

   Sabemos que el mas chico subio de ser hoja a ser la raiz y nuestro árbol tiene altura $\lceil \log_2\ n \rceil + 1$, (la altura es approx $\log_2\ 2n = \log_2\ n + 1$ por ser balanceado con base n pero como debe ser un número entero entonces se usa la función techo), por lo que por los 2 puntos anteriores sabemos que el segundo mas chico esta en un conjunto de a lo mas $(\lceil \log_2\ n \rceil + 1) - 1$ nodos (sabemos que la raiz no es el segundo mas chico).

   Finalmente solo es buscar en este conjunto bajando y comparando o usando la logica de arriba, podemos usar otro árbol con los elementos que queremos comparar para encontrar el segundo mas chico, teniendo como base los $\lceil \log_2\ n \rceil$ nodos que sabemos que pueden ser el segundo mas chico, encontramos el segundo mas chico en a lo mas $\lceil \log_2\ n \rceil - 1$ comparaciones.

   Finalmente nuestro algoritmo tiene una complejidad de $(n - 1) + (\lceil \log_2\ n \rceil - 1) = n + \lceil \log_2 n \rceil - 2$ comparaciones.

2. **Give an $O(n\ log\ k)$-time algorithm which merges k sorted lists with a total of n elements into one sorted list. (Hint: use a heap to speed up the elementary $O(kn)$-time algorithm).**

   Voy a asumir que el problema quiere juntar ”k” listas ya ordenadas por separado en una lista ordenada y no se refiere a que las listas estan k-ordenadas y se quieren juntar en una sola lista ordenada.

Usando un monticulo minimo

La idea es utilizar un min heap para que, en cada paso siempre extraemos el elemento mas chico lo que evidentemente resultara en una lista ordenada de n elementos.

Lo primero es crear el heap, como las listas ya estan ordenadas y buscamos solo el mas chico, es suficiente con tener el primer elemento de cada lista en nuestro heap, esto se puede hacer en $O(k \ log \ k)$ tiempo (insertar k nodos con cada inserción tomando a lo mas log k pues el monticulo tiene k nodos).

No creo que sea necesario explicar como funciona un min heap porque eso es mas de EDD pero tiene como propiedad que ira acomodando los nodos de forma ascendente basandose en sus valores.

Ahora para la creación de la lista, tomamos el primer elemento del heap, lo agregamos a la lista y lo eliminamos del heap, si el nodo que agregamos a la lista tenía un nodo siguiente en su lista de origen, habrá que agregarlo al heap, podremos acabar cuando el heap este vacío.

Ahora, como ya mencionamos el heap en cualquier momento sera de tamaño a lo mas k (el elemento mas pequeño que estamos procesando de cada lista) por lo que la complejidad en espacio será de $O(k)$, ademas, insertar en este heap toma $O(log \ k)$ tiempo y vamos a insertar en total n nodos por lo que la complejidad en tiempo sera de $O(n \ log \ k)$ (notese que $n \geq k$ si una lista es vacia ni la consideramos).

3. **Consider an $n \times n$ board of checkerboard B of alternating black and white squares. Assume that n is even. We seek to cover this checkerboard with rectangular dominos of size 2×1.**

Esta no esta bien hecha no calificar xd

(a) **Show how to cover the board with $\frac{n \times n}{2}$ dominos.**

Dividir el tablero

El problema parece bastante sencillo como n es par, tomamos cada par de cuadrados en una fila y los cubrimos con un domino. De esta forma, una fila de $1 \times n$ cuadrados queda cubierta por $\frac{n}{2}$ dominos. Repetimos esto para las n filas y el tablero queda cubierto por $\frac{n \times n}{2}$ dominos.

(b) **Remove the upper left and lower right corners from B. Show that you cannot cover the remaining board with $\frac{n \times n}{2} - 1$ dominos.**

Demostracion

Una manera medio rapida de ver esto es con un ejemplo, en este caso si por ejemplo tenemos un tablero 2x2, y quitamos las esquinas, nos queda la esquina superior derecha y la esquina inferior izquierda, y ahi no cabe un domino.

Otro argumento para en general es ver la coloracion, si quitamos las esquinas, nos queda un tablero con $\frac{nxn}{2} - 2$ cuadrados negros (podria ser blanco tambien pero el punto es que son del mismo color ya que las esquinas superior izquierda e inferior derecha estan en la misma diagonal porque n es par) y $\frac{nxn}{2}$ cuadrados blancos y como los cuadrados tienen colores alternantes entonces cada domino cubre uno de cada tipo por lo que si no tienen la misma cantidad de colores no hay forma de cubrirlo.

(c) **Remove one arbitrary black square and one arbitrary white square from B. Show that the rest of the board can be covered with $\frac{n \times n}{2} - 1$ dominos.**

Esta no esta bien hecha no calificar xd

Demostracion por induccion

En este caso notamos que no se incumple lo de antes, es decir se mantienen la misma cantidad de cuadros de ambos colores por lo que por ahora podemos seguir.

Caso base: n=2
Para un tableror de 2x2 tenemos 2 de cada tipo, independientemente de cuales 2 quitemos nos queda o una fila o una columna de color alternante y se cubre con un domino o $\frac{2x2}{2} - 1$.

Hipotesis de induccion: n Para un tablero de $n \times n$ con n par, si quitamos un cuadro de cada color, el tablero restante se puede cubrir con $\frac{n \times n}{2} - 1$ dominos.

Paso inductivo: n+2 Tenemos un tablero de $(n+2) \times (n+2)$, quitamos un cuadro de cada color, ahora tomamos 2 filas y 2 columnas no afectadas por el cambio y las quitamos, nos queda un tablero de $n \times n$ y por hipotesis de induccion se puede cubrir con $\frac{n \times n}{2} - 1$ dominos, ademas estas 2 filas y 2 columnas se pueden cubrir con lo que nos falta pero bueno ya no me da tiempo xd.

4. **Suppose we are given the minimum spanning tree T of a given graph G (with n vertices and m edges) and a new edge $e = (u - v)$ of weight w that we will add to G. Give an efficient algorithm to find the minimum spanning tree of the graph $G+e$. Your algorithm should run in $O(n)$ time.**

5. **Usted tiene que ordenar una serie Σ_n de n números, tales que todos son 0 ó 1. La única operación que puede hacer es comparar dos números cualesquiera x y y, y cada que los compara recibe la respuesta $x < y$, $x = y$, or $x > y$.**

(a) **De un algoritmo que con a lo más $n - 1$ comparaciones ordena Σ_n. Pruebe que su algoritmo es óptimo.**

Tomando un representante

La idea de este algoritmo es simple, primero tomas uno al azar, por ejemplo el primero, y lo pones en una lista L. Comparas con el segundo el que sea menor lo pones al principio de L (este es 0 y el mayor es un 1), podemos seguir con el mayor que sabemos que es un 1, lo comparamos con el tercero y si es menor lo ponemos atras del que tenemos y si es mayor adelante (en este caso el 1 va al final de L). Así sucesivamente hasta que termines de comparar todos los elementos.

Como el algoritmo compara cada elemento con los que ya están en la lista, el número de comparaciones es a lo más $n - 1$ (el primero ocupa al segundo para compararse).

Notamos que como no contamos con información adicional, no podemos ignorar a ningún elemento; por lo que el algoritmo es óptimo. Un caso muy feo por ejemplo es si tenemos puros 1's, de cualquier forma que los compares solo tendras iguales y no puedes saber si es porque son grupitos de iguales o porque todos son iguales hasta que los compares todos o con alguno que sea de su mismo tipo (con un arbol por ejemplo).

(b) **Encuentre un algoritmo que utilizando en promedio $\frac{2n}{3}$ comparaciones ordena Σ_n (esto bajo la suposición que los elementos de Σ_n tienen la misma probabilidad de ser 0 ó 1). Demuestre que su algoritmo es óptimo.**

Algo vital para resaltar en este caso es que si tenemos información extra por lo que no significa que el algoritmo anterior no sea optimo.

Usando arboles

La idea de este algoritmo va a ser agrupar por tipos cuando sean iguales y cuando no, ya tenemos un orden sobre esos grupos que sean diferentes; es importante destacar que en el caso que fueran todos de un solo tipo la complejidad sería $n-1$ comparaciones.

Vamos a comenzar por dividir nuestra serie de elementos en hojas de un arbol, adicionalmente a el valor que tengan (que desconocemos) vamos a agregarle un valor a estos nodos que indiquen de que tamaño es su grupo; ahora comparamos por pares (si nos sobra uno podemos compararlo en el siguiente nivel). Tenemos 4 casos posibles:

i. $x = y$ Ambos son iguales (dos 1), en este caso creamos un nodo padre con el valor de uno de los 2 y el tamaño de su grupo es la suma del tamaño de ambos grupos (en este caso 2).
ii. $x < y$ En este caso ya sabemos quienes son 1's y quienes son 0's, por lo que los contamos dependiendo del tamaño de su grupo y estos ya no juegan (sabemos que hay un 1 y un 0).
iii. $x > y$ En este caso ya sabemos quienes son 1's y quienes son 0's, por lo que los contamos dependiendo del tamaño de su grupo y estos ya no juegan (sabemos que hay un 1 y un 0).
iv. $x = y$ Ambos son iguales (dos 0), este caso es analogo al caso 1.

Esto nos tomo $\mathbf{n/2}$ comparaciones, pero lo interesante aqui es que si todos los casos tienen la misma probabilidad (por la suposición del problema) entonces la mitad de los elementos ya estan ordenados y no los volvemos a comparar esto significa que en el siguiente nivel solo hay $(n/2)/2 = n/4$ nodos (normalmente serían $n/2$ pero la mitad no sobrevive).

Podemos repetir un procedimiento similar en el siguiente nivel, este tiene $n/4$ nodos por lo que comparando por parejas nos toma $(n/4)/2 = \mathbf{n/8}$ comparaciones. Seguimos haciendo esto y por ejemplo el siguiente nivel tendra $(n/8)/2 = n/16$ nodos y usara $(n/16)/2 = \mathbf{n/32}$ comparaciones.

Se sigue de este razonamiento que la cantidad de comparaciones se ve algo como esto:

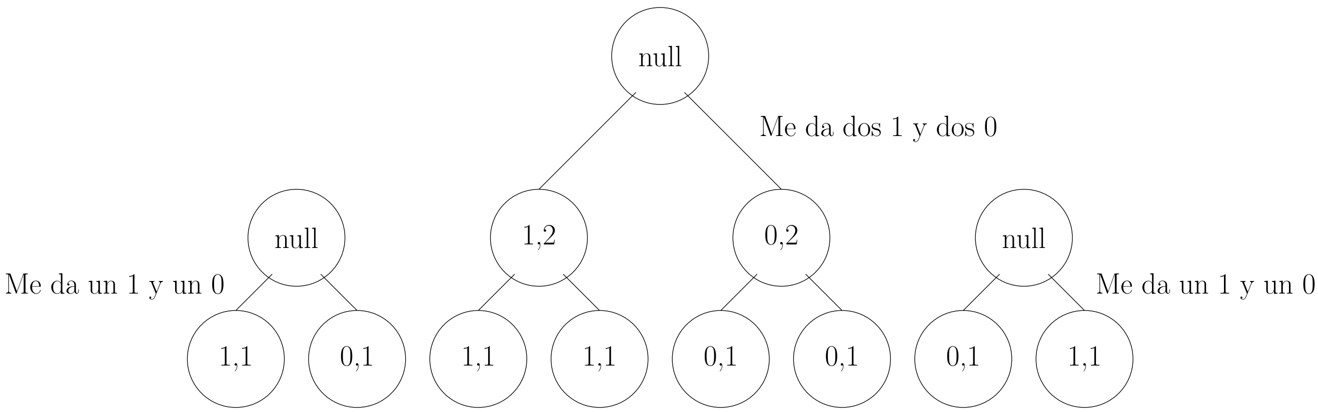
$$\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + 1$$

Esta es una progresión geométrica con razón de $\frac{1}{4}$ empezando por $\frac{n}{2}$, ademas en este caso cuando llegamos a la raiz mas bien no sabemos de que tipo es solo sabemos que todos sus elementos son del mismo tipo pero podrían ser 1's o 0's, por lo que ocupamos una comparación mas con alguno ya ordenado (si solo hay un tipo entonces ocupa $n-1$ comparaciones y esta trivialmente ordenado).

$$\frac{n/2}{1 - 1/4} = \frac{n/2}{3/4} = \frac{2n}{3}$$

Hagamos un ejemplo grafico chico para ver como se ve esto:

Final: [0,0,0,0,1,1,1,1]



valor, tamaño de grupo

En este caso tenemos 8 elementos, hacemos 4 comparaciones y ordenamos a 4; nos quedan 2 nodos vivos con lo que hacemos 1 comparación y tenemos 0 nodos vivos con lo que ya sabemos cuantos 1's y 0's hay.

Notemos que no hay un caso terrible, en cuanto no sean iguales podemos quitar a todos los elementos de ambos arboles y como ya mencionamos el peor caso seria si todos fueran iguales.

6. **Consider the numerical game "20 questions". In this game, player 1 thinks of a number in the range 1 to n. Player 2 has to figure out this number by asking the fewest number of true/false questions. Assume that nobody cheats.**

   (a) **What is an optimal strategy if n in known?**

   Asumimos que se pueden hacer mas de 20 pregutnas porque si no es posible que el jugador 1 escoja un numero demasiado grande donde lo siguiente no funcionara.

   Usando Busqueda Binaria

   Como ya sabemos si conocemos el rango de valores sobre los cuales tenemos que buscar podemos usar busqueda binaria. La estrategia es ir diviendo el rango en mitades y ver de que lado esta el numero. Empezando con el rango $[1, n]$ y preguntando si el numero es mayor o menor a $n/2$. Si es mayor entonces el rango se convierte en $[n/2, n]$ y si es menor el rango se convierte en $[1, n/2]$. Repitiendo este proceso hasta que el rango sea de tamaño 1, esto se logra en $\log_2(n)$ tiempo en el peor caso.

   (b) **What is a good strategy is n is not known?**

   Usando Busqueda Exponencial

   La idea de la busqueda exponencial es primero encontrar un rango donde el numero se encuentre y luego hacer busqueda binaria en ese rango. Primero necesitamos encontrar un numero k tal que $2^{k-1} < n \leq 2^k$. Luego hacemos busqueda binaria en el rango $[2^{k-1}, 2^k]$ para encontrar el número.

Para ser mas especificos, comenzamos con $i = 1$ y checamos si es o no mas grande que nuestro numero si no lo es entonces $i = 2i$ y repetimos el proceso hasta encontrar el que cumpla la desigualdad anterior o lleguemos a uno mas grande que n. Vemos que el rango se esta duplicando en cada paso lo que significa que estamos considerando exponencialmente mas valores en cada paso lo que nos garantiza encontrar el rango en $O(\log(n))$ tiempo.

Una vez que tengamos el rango, hacemos BB para encontrar el numero en $O(\log(n))$ tiempo.

7. **Sea T un árbol con raíz con n vértices. Cada nodo v tiene asociado un peso $w(v)$. Utilizando programación dinámica, encuentre un algoritmo de tiempo lineal para encontrar el conjunto independiente de T de peso máximo.**

Este problema se asemeja bastante al problema 3 de la tarea 3 asi que lo vamos a abordar de manera similar. (reutilice algunos de los diagramas que hice porque me quedaron bonitos)

Aclaraciones:

- Solo vamos a considerar pesos positivos.
- El arból puede o no ser binario.
- Solo vamos a considerar arrboles con mas de 2 nodos pues si no el problema es trivial. (aunque el algoritmo que vamos a presentar funciona para arboles para todo n)

La idea es utilizar el arbol que ya existe, agregarle informacion y recorerlo de manear eficiente.

Usando DP en el arból

Lo que vamos a intentar es considerar a un nodo y sus hijos para ver si nos conviene tomarlo o no. Comenzamos por los nodos mas faciles de analizar, las hojas. Si un nodo es hoja, entonces el conjunto independiente de peso maximo que lo contiene es el mismo nodo.

Vamos a utilizar un recorrido postorden paa recorrer el arbol. De manera que procesamos primero a los hijos de un nodo antes de procesar al nodo en si, este recorrido tiene complejidad $O(n)$.

Ahora, en cada nodo vamos a agregarle información, para ello vamos a usar 2 funciones que se pueden describir asi:

- **Incluir:** Si incluimos al nodo en el conjunto, entonces sus hijos no pueden ir. Esto se ve asi:

$$\text{Incluir}(nodo) = peso(nodo) + \sum_{h \in hijos} \text{excluir}(h)$$

- **Excluir:** Si excluimos al nodo del conjunto, entonces podemos o no incluir a sus hijos. Esto se ve asi:

$$\text{Excluir}(nodo) = \sum_{h \in hijos} \max\{\text{incluir}(h), \text{excluir}(h)\}$$

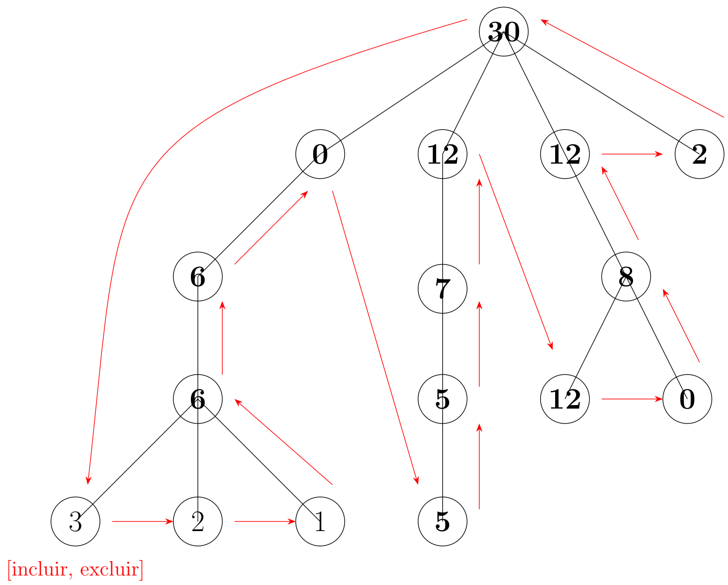
Ademas de estas funciones, vamos a agregar la base de la recurrencia, como ya dijimos cuando un nodo es hoja no tiene restricciones, por lo que podemos definir las funciones asi:

$$\text{incluir}(nodo) = peso(nodo)$$
$$\text{excluir}(nodo) = 0$$

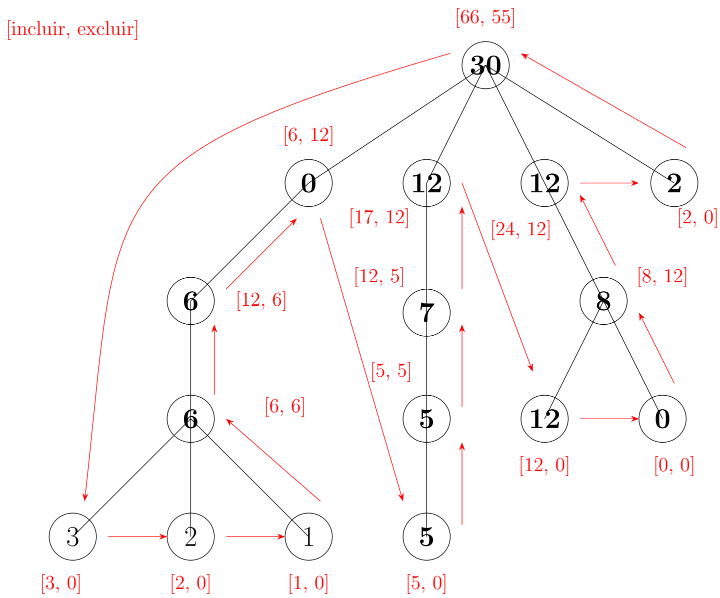
Adicionalmente, si queremos considerar negativos (aunque ya lo resuelve) podemos decir que si un valor es negativo simplemente no lo agregamos.

Entonces para cada nodo, empezando por las hojas agregamos esta información y recuperamos la información de sus hijos para poder calcular la información del nodo padre.

Entonces el arbol se va a ir llenando algo asi en el recorrido:

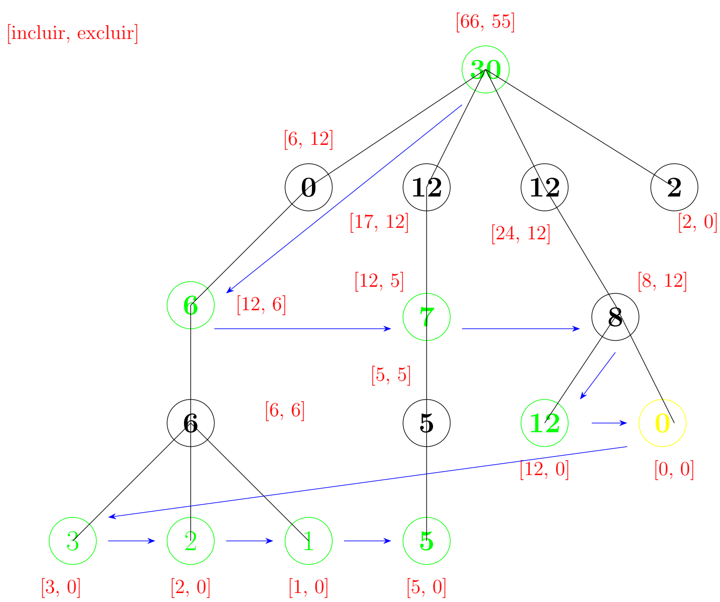


Seguimos el camino rojo y para cada nodo vamos calculando sus 2 valores, esto sucede en a lo mas O(n) pues cada nodo puede tener a lo mas n-1 hijos. (las funciones de incluir y excluir tienen que checar a todos sus hijos) pero en el caso de las hojas y en general cuando tienen una cantidad razonable de hijos (hijos "constantes" relativo a n) se hace en O(1).



De aqui ya podremos decir cual sera el peso maximo, pero el problema nos pide el conjunto independiente, asi que vamos a crearlo, para ello vamos a hacer un recorrido bfs para checar por nivel si un nodo es incluido o no, si es incluido no tenemos que cehcar a sus hijos, solo habria que checar a sus nietos, si un valor da igual podemos decidir si incluirlo o no.

Esto nos puede quedar algo asi:



El camino azul nos dice cuales tenemos que verificar pero hay que recalcar que todos los nodos entran a la fila en algun momento aunque sea para meter a sus hijos, por lo que la complejidad de este paso es O(n) (bfs usualmente es la suma de aristas y vertices pero en arboles tenemos n-1 aristas).

Los nodos que agregamos los pinte de verde pero podemos agregarlos a una lista y los que excluyo son o hijos de incluidos o nodos cuyo valor de excluir es mayor que el de incluir y no quedan coloreados (o incluidos).

En total entonces el algoritmo tiene complejidad de recorrer el arbol con un recorrido postorden, agregar informacion por cada nodo mas el recorrerlo con bfs, o lo que es lo mismo O(n)*O(1) + O(n) = O(n) y el espacio adicional es O(n) para guardar la lista de nodos que vamos a incluir. (notemos que en el caso de que un nodo tiene muchos hijos estos a su vez ya no pueden tener muchos hijos).

8. **Dado un arreglo A con n enteros positivos y negativos.**

Comenzamos por notar que la multiplicacion en enteros multiplica valores absolutos y el signo del resultado depende de si tienen signos distintos. Por lo tanto para encontrar el par de numeros que maximizan el producto, debemos encontrar el par de numeros con los valores absolutos mas grandes que tengan signos iguales (los 2 mas grandes o los 2 mas chicos).

(a) **Diseña un algoritmo de tiempo cuadrático que encuentre la pareja de números que maximizan el producto.**

Fuerza bruta

Una respuesta inocente sería recorrer todos los pares de números y calcular su producto. Esto es, toma el primer numero y recorre el resto del arreglo multiplicandolo uno por uno, si el producto es mayor que el maximo lo guardamos. Luego toma el segundo numero y repite el proceso, hacemos esto para todos los numeros del arreglo. La complejidad de este algoritmo es $O(n^2)$.

(b) **Diseña un algoritmo de tiempo $O(n \log n)$ que encuentre la pareja de números que maximizan el producto.**

Ordenación

Podemos, por el corolario del principio, tomar y ordenar nuestro arreglo usando algun algoritmo de ordenación $O(n \log n)$, como puede ser merge sort. Luego, el producto de los dos extremos del arreglo será el mayor posible, los 2 mas negativos o los 2 mas positivos.

(c) **¿Se puede mejorar el inciso anterior? Si es el caso, muestre el algoritmo, sino explique por qué.**

Si se puede mejorar, como ya notamos, para maximizar el producto es suficiente con considerar los dos extremos del arreglo ordenado. Lo que es lo mismo, encontrar los 2 maximos y 2 minimos, que ya mostramos se puede hacer en tiempo lineal en otra tarea.

Minimos y maximos

En otra tarea nos pedian minimizar la cantidad de comparaciones para encontrar el minimo y maximo de un arreglo. La solución a ese problema se hacia con heaps y se lograba en algo asi como $3n/2 + k$ o algo asi pero aqui no importa la cantidad de comparaciones si no el tiempo de ejecucion del algoritmo.

Entonces la idea es bastante mas sencilla, vamos a recorrer el arreglo y para cada elemento vamos a comprarlo con los 2 maximos y 2 minimos (en el peor caso) que ya tenemos guardados. Algo asi como ver si es mayor que el maximo maximo si lo es ahora guardamos ahi el nuevo maximo maximo y el antiguo maximo maximo pasa a ser el nuevo maximo. En otro caso comparamos con el segundo maximo y si lo es lo guardamos, si no lo descartamos. Hacemos lo mismo con los minimos. Esto toma algo asi como $4n$ comparaciones y por lo tanto es $O(n)$.

Finalmente el producto de los dos maximos o el producto de los 2 minimos sera el mayor posible solo hay que compararlos.

9. **Suppose that we are given a sequence of n values $x_1, x_2, \ldots, x_n$ and seek to quickly answer repeated queries of the form: given i and j, find the smallest value in $x_i, \ldots, x_j$.**

Como nota importante para este problema es que no nos sirve ordenar el arreglo puesto que esto cambia la posición de los elementos y no podemos acceder al rango especifico.

(a) **Design a data structure that uses $O(n^2)$ space and answers queries in $O(1)$ time.**

DP con matriz

Primero que nada vamos a crear una matriz usando $dp[i][j] = \min\{x_i, x_{i+1}, \ldots, x_j\}$, es obvio que esta matriz tiene un tamaño de $n \times n$ para considerar todos los rangos posibles. Si se llena mal puede subir mucho la complejidad aunque no afecta a lo que nos estan pidiendo pero, para llenarla eficientemente vamos a usar DP. Empezamos llenando la diagonal con los valores de x_i (el minimo de un solo elemento) y usamos la siguiente funcion de recurrencia para llenar la matriz:

$$dp[i][j] = \min\{dp[i][j-1], x_j\}; \quad i < j$$

Ahora, para responder a las queries simplemente devolvemos $dp[i][j]$ que es constante y no depende de n aunque el precomputo tarda $O(n^2)$.

Ejemplo:

Supongamos que tenemos el arreglo $x = [3, 1, 4, 1, 5]$ y queremos llenar la matriz dp:

$i \backslash j$	1	2	3	4	5
1	3	1	1	1	1
2		1	1	1	1
3			4	1	1
4				1	1
5					5

Notemos que la matriz solo se llena en la diagonal principal y por encima de ella, el resto de las celdas no se llenan puesto que no tiene sentido el rango x_3, x_2 por ejemplo.

(b) **Design a data structure that uses $O(n)$ space and answers queries in $O(log\ n)$ time.**

Segment Tree

La idea es usar un segment tree para guardar los minimos de los rangos. Comenzamos por dividir el arrreglo en subarrerglos, cada nodo en el segment tree representa el minimo de un intervalo en el arreglo original. (vamos a guardar 2 indices tambien para saber que intervalo cubre)

La raiz del arbol representa el rango completo $[1, n]$ y por tanto el minimo de todo el arreglo. Cada nodo tiene dos hijos que representan la mitad del rango del padre y tienen el minimo de ese rango.

El razonamiento del porque este arbol usa $O(n)$ espacio es que el rango se va partiendo a la mitad y por tanto el arbol tiene a lo mucho $2n$ nodos donde cada nodo solo guarda 3 numeros. (serie geometica con termino inicial n y razon 1/2).

Para construir el arbol, tomamos el arreglo y lo dividimos en 2 partes, si no es de tamaño 1, llamamos recursivamente a la funcion para cada parte y guardamos el minimo de cada parte en el nodo actual la altura de este arbol sera de $O(\log\ n)$ y construirlo toma O(n).

Para responder a las queries, si el rango del nodo actual esta contenido en el rango de la query, devolvemos el minimo de ese rango. Si el rango del nodo actual esta fuera del rango de la query, lo ingoramos. Si el rango del nodo actual intersecta con el rango de la query, llamamos recursivamente a los hijos y devolvemos el minimo de los minimos de los hijos, esto toma $O(\log\ n)$ pues en cada nivel procesamos a lo mas 2 nodos (un rango es continuo) y la altura del arbol es $O(\log\ n)$.

10. **El intervalo común más largo de dos sucesiones X y Y , es un conjunto de elementos consecutivos de X y Y más largo que aparece en ambas sucesiones (ojo, no estamos hablando de la subsucesión común más larga). Por ejemplo, el intervalo común más largo de las sucesiones *fotografía* y *tomografía* es *ografía*. Sean $n = |X|$ y $m = |Y|$. Encuentre un algoritmo que en $\Theta(nm)$ encuentre el intervalo común más largo de X y Y.**

DP con matriz

Vamos a crear una matriz dp de dimensiones $(n{+}1)\times(m{+}1)$, donde $dp[i][j]$ va a representar el tamaño del intervalo común más largo de $X[0..i{-}1]$ y $Y[0..j{-}1]$. Inicializamos la matriz con ceros. Luego, recorremos la matriz llenandola de la siguiente manera:

$$dp[i][j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0, \\ dp[i-1][j-1] + 1 & \text{si } X[i-1] = Y[j-1], \\ 0 & \text{si } X[i-1] \neq Y[j-1]. \end{cases}$$

Podemos ademas utilizar una variable para almacenar la posicion final del intervalo común más largo, y así poder reconstruirlo, ademas de otra para almacenar el tamaño del intervalo común más largo. La complejidad de este algoritmo es $\Theta(nm)$ ya que recorremos la matriz una sola vez. Funciona porque si encontramos que los caracteres de X y Y en la posición i y j son iguales, entonces el intervalo común más largo se extiende en uno, y si no son iguales, entonces el intervalo común más largo se rompe; la matriz garantiza que se busca en todas las posibles combinaciones de intervalos comunes.

Ejemplo:

		t	o	m	o	g	r	a	f	i	a
	0	0	0	0	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	1	0	0
o	0	0	1	0	1	0	0	0	0	0	0
t	0	1	0	0	0	0	0	0	0	0	0
o	0	0	2	0	1	0	0	0	0	0	0
g	0	0	0	0	0	2	0	0	0	0	0
r	0	0	0	0	0	0	3	0	0	0	0
a	0	0	0	0	0	0	0	4	0	0	1
f	0	0	0	0	0	0	0	0	5	0	0
i	0	0	0	0	0	0	0	0	0	6	0
a	0	0	0	0	0	0	0	1	0	0	7

Table 1: Matriz de programación dinámica para las secuencias "fotografía" y "tomografía".

Si tenemos la variable max que almacena el tamaño del intervalo común más largo, y la variable end que tiene la posición final del intervalo, entonces en este ejemplo $max = 7$ y $end = 9$ (si contamos desde 0 en la palabra fotografía), entonces el intervalo común más largo es "ografía" construyendo desde la posición $end - max + 1$ hasta end en la palabra fotografía (se usa el +1 porque las palabras empiezan por la posición 0 pero la subcadena la contamos por cantidad osea desde 1). Aunque tambien se puede construir la subcadena buscando el indice mas grande en la matriz dp y luego retrocediendo por la diagonal hasta encontrar un 0, y asi construir la subcadena.

11. **Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.**

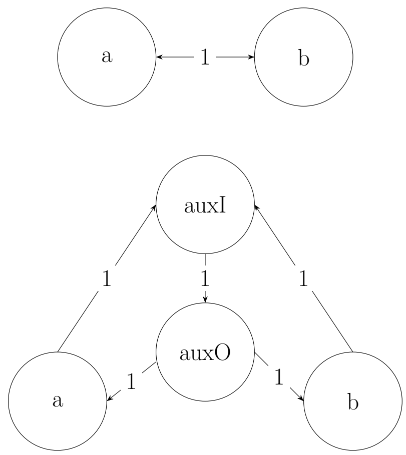
12. **Professor Adams has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining if both his children can go to the same school as a maximum-flow problem.**

Este problema se asemeja bastante al problema 6 de la tarea anterior asi que voy a usar la misma idea y corregir mis errores.

Modelando el problema como uno de flujos

Comenzamos por modelar las intersecciones de las calles como nodos, debido a que pueden pasar ambos hijos no tienen restricciones, además, la casa es nodo fuente y la escuela es nodo sumidero.

Ahora agregaremos las aceras como aristas, pero vamos a checar esto, porque nos gustaría que se pudieran usar en ambos sentidos pero solo por uno de los 2 hijos, es asi que vamos a modelar las aristas de la siguiente manera:



Es decir modelare la calle que conecta las esquinas a,b que inicialmente era bidireccional de la siguiente manera: agregamos 2 nodos, digamos calleI y calleO, a en vez de conectarse con b se conectara a calleI al igual que b, y calleI se conectara con calleO, y calleO con a y b, todas utilizaran una capacidad de 1 para que solo puedan ser usadas por un hijo 1 vez. Esto permite que exista una trayectoria de "a" a "b" y de "b" a "a" pero como solo existe una arista calleI - calleO entonces no se pueden usar ambos sentidos al mismo tiempo.

Es evidente que esto no genera nuevas trayectorias pues solo se esta dividiendo la arista en 2, no hay conexiones nuevas, por lo que el flujo máximo en este grafo es el mismo que en el original.

Ahora tenemos un grafo dirigido con capacidades en las aristas, los nodos fuentes y sumidero deben tener al menos 2 aristas salientes y entrantes respectivamente para que el flujo pueda pasar por ellos.

Para decidir si es posible que ambos hijos lleguen a la escuela sin violar restricciones, simplemente calculamos el flujo máximo en el grafo, si el flujo máximo es 2 entonces ambos hijos pueden llegar a la escuela sin problemas, si es 1 entonces solo uno de ellos puede llegar a la escuela, si es 0 entonces ninguno puede llegar a la escuela.

Para encontrar el flujo máximo podemos usar el algoritmo de Ford-Fulkerson, Edmonds-Karp o cualquier otro algoritmo de flujo máximo.

13. **Mientras caminas por la playa encuentras un cofre de tesoros. El cofre contiene n tesoros con pesos $w_1, \ldots, w_n$ y valores $v_1, \ldots, v_n$. Desafortunadamente sólo tienes una mochila que solo tiene capacidad de carga M. Afortunadamente los tesoros se pueden romper si es necesario. Por ejemplo, la tercera parte de un tesoro i tiene peso $\frac{w_i}{3}$ y valor $\frac{v_i}{3}$.**

(a) **Describe un algoritmo voraz de tiempo $\Theta(n\ log\ n)$ que resuelve este problema.**

Este se parece bastante al problema de la mochila que vimos con la profesora asi que voy a utilizar esa idea.

Algoritmo greedy

Como podemos llevarnos cuanto queramos de cada tesoro mientras no sobrepase la capacidad de nuestra mochila, nos interesa comenzar por calcular la densidad de cada tesoro, es decir, $\frac{v_i}{w_i}$ para cada i esto toma $\Theta(n)$.

Luego ordenamos los tesoros de acuerdo a su densidad, de forma descendiente usando merge sort, esto toma $\Theta(n\ log\ n)$, esto es escencial para utilizar el algoritmo greedy.

Ahora, mientras la mochila tenga capacidad y existan tesoros por tomar, tomamos la mayor cantidad de la fracción de un tesoro que podamos, es decir, si el peso del tesoro i es w_i y cabe en la mochila (es decir peso restante $> w_i$) tomamos el tesoro completo, si no, tomamos la fracción que cabe, esto toma $\Theta(n)$.

Finalmente, el algoritmo toma $\Theta(n\ log\ n)$, sabemos que es exactamente $\Theta\ (n\ log\ n)$ porque el tiempo de ordenar los tesoros usando merge sort es el que domina el tiempo total del algoritmo.

(b) **Prueba que tu algoritmo obtiene la solución correcta.**

Por contradicción

Supongamos que el algoritmo no obtiene la solución correcta. Sea S la solución y sea S' la solución que obtiene el algoritmo. Entonces $S > S'$, ahora sabemos que S' se puede escribir asi:

$$S' = \sum_{i=1}^{n} x_i v_i$$

Donde x_i es la fracción de i-ésimo tesoro que se lleva en la mochila. Ademas, sabemos que S se puede escribir asi:

$$S = \sum_{i=1}^{n} y_i v_i$$

Es decir debe existir uno o mas objetos que nuestro algoritmo no tomo ya sea parcial o totalmente y que si se hubieran tomado la solución hubiera sido mayor. Pero como tenemos que minimizar la cantidad de peso que llevamos a la mochila por valor entonces esto significa que la densidad del objeto que no se tomo es mayor a la densidad de los objetos que si se tomaron. Sin embargo, esto contradice la selección de objetos que hace nuestro algoritmo ya que siempre toma el objeto con mayor densidad. Por lo tanto, nuestro algoritmo obtiene la solución correcta.

Una manera mas intuitiva de verlo es que querermos maximizar la densidad dentro de la mochila, entonces ir metiendo los objetos mas densos que podamos es la mejor opción, otra manera de decirlo es que una solución optima a un subproblema de este problema tambien es optima para el problema original.

(c) **Mejora el tiempo de ejecución de tu algoritmo a $\Theta(n)$**

La verdad no estoy seguro de esta parte pero lo voy a intentar:

Usando K-select

La idea para esta variación de la solución empieza igual, tomamos y obtenemos la relación valor peso (densidad) para cada objeto. Pero en vez de ordenar completamente el arreglo usaremos selection k para considerar solo una parte del arreglo, una cosa importante en este algoritmo es que para mantener su complejidad en $O(n)$ voy a usar la mediana de medianas para seleccionar el pivote, esto me garantiza que el algoritmo no se vaya a $O(n^2)$.

Una vez tomando la mediana de medianas en $O(n)$ hacemos 3 particiones del arreglo, los elementos menores a la mediana, los iguales y los mayores; ahora calculamos la suma de los pesos de los elementos cuya densidad es mayor a la mediana, si esta suma es igual a la capacidad de la mochila entonces terminamos y nos podemos llevar solo a esos.

Si la suma es menor a la capacidad de la mochila, vamos calculando los pesos de los objetos cuya densidad es igual a la mediana y viendo si nos van cabiendo en la mochila hasta que se llene, si se llena acabamos.

Si aun no se llena, entonces calculamos la mediana de medianas para los menores y repetimos el proceso.

Otro caso que puede pasar es que la suma de los pesos de los objetos cuya densidad es mayor a la mediana sea mayor a la capacidad de la mochila, en este caso calculamos la mediana de medianas para los mayores y repetimos el proceso.

Si en algun paso estamos en una particion y solo cabe una fraccion de un objeto, entonces lo tomamos y terminamos.

Me parece que este algoritmo tiene una complejidad de $O(n)$, estamos haciendo $O(n)$ operaciones por paso y en cada paso reducimos el tamaño del arreglo a la mitad, la serie tiende a algo en $O(n)$.

14. **Encuentre la parentización óptima para multiplicar seis matrices de dimensiones 4×9, 9×4, 4×10, 10×2, 2×5, 5×6.**

Primero que nada voy a ponerle nombres a las matrices con las que estoy trabajando para que sea más fácil referirse a ellas. Así que las matrices serán:

$$A0 : 4 \times 9$$
$$A1 : 9 \times 4$$
$$A2 : 4 \times 10$$
$$A3 : 10 \times 2$$
$$A4 : 2 \times 5$$
$$A5 : 5 \times 6$$

Y el vector de dimensiones será $p = \{4, 9, 4, 10, 2, 5, 6\}$.

Usando DP

Voy a crear una matriz dp de tamaño 6×6 donde $dp[i][j]$ va a ser el costo mínimo de multiplicar las matrices $A_i \times A_{i+1} \times \ldots \times A_j$.

Además, voy a crear una matriz s de tamaño 6×6 donde $s[i][j]$ va a ser el índice de la matriz que se va a multiplicar en la última multiplicación de la cadena $A_i \times A_{i+1} \times \ldots \times A_j$.

Para llenar la matriz dp y s voy a usar la siguiente función:

$$\begin{cases} \mathrm{dp}[i][j] & = \min_{i \le k < j}\{\mathrm{dp}[i][k] + \mathrm{dp}[k+1][j] + p_i \cdot p_{k+1} \cdot p_{j+1}\} \quad \text{si } i < j \\ \mathrm{dp}[i][j] & = 0 \quad \text{si } i = j \end{cases}$$

Donde $p_{i-1} \cdot p_k \cdot p_j$ es el numero de operaciones que se hacen al multiplicar las matrices resultantes de esas subcadenas.

Y para llenar la matriz s voy a usar la siguiente función:

$$\mathrm{s}[i][j] = k$$

Empezamos por llenar la tabla dp por su diagonal, es decir, $dp[i][i] = 0$ para todo i esto es porque el costo mínimo de multiplicar una sola matriz es 0, despues seguimos la función es optimo ir llendo en diagonales empezando por la principal para arriba, podemos optimizar y solo llenar la mitad de la matriz ya que es simétrica.

Ahora si vamos a aplicarlo y llenar la tabla (cuidado con confundir indices y elementos):

Tabla de costos mínimos (dp)

i/j	0	1	2	3	4	5
0	0	144	304	224	264	332
1		0	360	152	242	320
2			0	80	120	188
3				0	100	180
4					0	60
5						0

Tabla de puntos de división (s)

i/j	0	1	2	3	4	5
0	0	0	1	0	3	3
1		0	1	1	3	3
2			0	2	3	3
3				0	3	3
4					0	4
5						0

No escribi todos los calculos para no hacerlo tan largo pero es un proceso tedioso y cada uno de los valores puede cambiar dependiendo del k en el que vamos.

Como vemos el costo minimo es 332 y el punto de división es 3, eso significa que creamos un grupo con las matrices A_0, A_1, A_2, A_3 y otro con las matrices A_4, A_5 nos queda (A_0, A_1, A_2, A_3) $(A_4,\ A_5)$ hasta ahora.

Checamos $s[0, 3] = 0$ asi que el primer grupo se divide en $(A_0\ (A_1, A_2, A_3))$ y el segundo grupo se queda igual.

Checamos $s[1, 3] = 1$ asi que el primer grupo se divide en $(A_0\ (A_1\ (A_2, A_3)))$ y el segundo grupo se queda igual, separar mas este grupo ya no tiene mucho efecto porque se queda igual.

Seguimos con el segundo grupo pero igual ya solo son 2 matrices entonces lo dejamos asi, al final nos queda la siguiente parentización óptima:

$$(A_0 \times (A_1 \times (A_2 \times A_3))) \times (A_4 \times A_5)$$

15. **Sea P_n una familia de n puntos en el plano, en posición general. Encuentre un algoritmo que encuentre el triángulo de mayor área cuyos vértices estén en P_n. Su algoritmo tiene que trabajar en menos de $O(n^3)$. Hint: Pruebe primero que los vértices de dicho triángulo están en el cierre convexo de P_n, esto reduce el problema al caso en que los elementos de P_n son los vértices de un polígono convexo.**

16. **Supongamos que usted es el dueño de la compañía Anuncios Espectaculares y que recibe un contrato para colocar anuncios espectaculares en la autopista México Querétaro . Los lugares donde puede colocar sus anuncios están localizados en los kilómetros $x_1, \ldots, x_n$ de dicha autopista. Si usted coloca un anuncio en el lugar localizado en el kilómetro x_i, recibirá una ganancia r_i. Por razones de seguridad, los anuncios no pueden estar muy cerca uno del otro, y la distancia entre dos anuncios consecutivos tiene que ser al menos 10 kilómetros. Observe que la distancia entre dos sitios cualquiera donde puede localizar sus anuncios no es necesariamente mayor que 10 kilómetros.**

    Encuentre un algoritmo que resuelva el problema de colocar anuncios de tal forma que las ganancias de su compañía se maximicen. Por ejemplo si:

    $$\{x_1, x_2, x_3, x_4\} = \{12, 14, 23, 28\}$$

    y

    $$\{r_1, r_2, r_3, r_4\} = \{5, 6, 5, 1\}$$

    la solución óptima sería colocar anuncios en los kilómetros 12 y 24 para una ganancia de 10.

17. **Consider the following data compression technique. We have a table of m text strings, each at most k in length. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains $(a, a, abab, b)$ and the data string is $bababbaababa$, the best way to encode it is $(b, abab, ba, abab, a)$ a total of five code words. Give an $O(nmk)$ algorithm to find the length of the best encoding. You may assume that every string has at least one encoding in terms of the table.**

    Un detalle muy importante en esta parte es que no nos piden regresar el código, sino la longitud del mejor código. Por lo tanto, no es necesario guardar los códigos, sino simplemente la longitud de los mismos.

    Usando DP

Comenzamos definiendo un arreglo dp de tamaño $n+1$ donde $dp[i]$ será el minimo numero de palabras necesarias para codificar los primeros i caracteres de D. Inicializamos $dp[0] = 0$ (0 palabras para codificar la cadena vacia).

Ahora inicia lo interesante, para cada posicion i de D y para cada palabra en la tabla digamos s vamos a checar si s es sufijo de D de 0 a i es importante notar que la longitud de las palabras es a lo mas k, si s es sufijo entonces sea $j = i - |s|$ y updateamos el arreglo de la siguiente manera:

$$dp[i] = min(dp[i], dp[j] + 1)$$

Lo que estamos intentando hacer aqui basicamente es codificar los primeros j caracteres de D y luego agregar la palabra s para codificar los primeros i caracteres de D para ver si nos da menos palabras que codificar los primeros i caracteres de D directamente. Finalmente la respuesta sera $dp[n]$.

Ademas de esto nuestra funcion de recurencia le agregamos valores por defecto para poder usar la funcion min, $dp[i] = \infty\ ; \forall i > 0$

Complejidad: La complejidad de este algoritmo es $O(nmk)$ ya que tenemos que recorrer cada posicion de D (n) y para cada posicion tenemos que recorrer cada palabra de la tabla (m) y para cada palabra tenemos que recorrer la palabra para ver si es sufijo de D (k).

Para nuestro ejemplo, corri un codigo en python y obtuve la siguiente los dp en cada iteracion:

0	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	1	1	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	1	1	2	∞	∞	∞	∞	∞	∞	∞	∞	∞
0	1	1	2	2	∞	∞	∞	∞	∞	∞	∞	∞
0	1	1	2	2	2	∞	∞	∞	∞	∞	∞	∞
0	1	1	2	2	2	3	∞	∞	∞	∞	∞	∞
0	1	1	2	2	2	3	3	∞	∞	∞	∞	∞
0	1	1	2	2	2	3	3	4	∞	∞	∞	∞
0	1	1	2	2	2	3	3	4	5	∞	∞	∞
0	1	1	2	2	2	3	3	4	5	5	∞	∞
0	1	1	2	2	2	3	3	4	5	5	4	∞
0	1	1	2	2	2	3	3	4	5	5	4	5

Donde la respuesta es 5, nos da 12 iteraciones porque la longitud de D es 12, en cada iteracion tuvo que checar toda la tabla de palabras y para cada palabra tuvo que recorrer la palabra para ver si era sufijo de D.

18. **La compañía *Monsters, Inc* preparó para el día de hoy una fila de $n \geq 2$ puertas para abrir y recolectar los gritos de los niños con el fin de abastecer de energía a la Monstropolis. Con el propósito de salir temprano los monstruos determinaron abrir todas las puertas de jalón y así terminar lo más rápido posible, por lo que asignaron a Sullivan a recorrer toda la fila y abrir todas las puertas. Al ver el desastre que esto causaría, Mike Wazowski recorrió la fila y cerró todas las puertas de manera alternada iniciando en la puerta 2. Sin embargo, ¡Quedaron $\frac{n}{2}$ puertas abiertas!. Por lo que el resto de los monstruos decidieron ayudar de la siguiente manera: el monstruo con el recorrido i-ésimo, cambiaría el estado de cada i-ésima puerta iniciando desde la puerta**

i. Después de hacer este proceso n veces, ¿Quedan puertas abiertas? ¿Cuántas y cuáles son si es el caso?

Este problema es mas de teoría de números aunque tambien se puede programar para ver casos específicos.

Usando teoria de números

Para empezar hice un ejemplo a mano para darme una idea de como se comportaba el problema. En este caso $n = 26$.

$$
\begin{array}{rl}
 & [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0] \\
2 & [0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1] \\
3 & [0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,1,1,1,0,0,0,1] \\
4 & [0,1,1,0,0,0,0,0,1,1,0,1,0,1,1,0,0,0,0,0,1,1,0,1,0,1] \\
5 & [0,1,1,0,1,0,0,0,1,0,0,1,0,1,0,0,0,0,0,1,1,1,0,1,1,1] \\
6 & [0,1,1,0,1,1,0,0,1,0,0,0,0,1,0,0,0,1,0,1,1,1,0,0,1,1] \\
7 & [0,1,1,0,1,1,1,0,1,0,0,0,0,0,0,0,0,1,0,1,0,1,0,0,1,1] \\
8 & [0,1,1,0,1,1,1,1,1,0,0,0,0,0,0,1,0,1,0,1,0,1,0,1,1,1] \\
9 & [0,1,1,0,1,1,1,1,0,0,0,0,0,0,0,1,0,0,0,1,0,1,0,1,1,1] \\
10 & [0,1,1,0,1,1,1,1,0,1,0,0,0,0,0,1,0,0,0,0,0,1,0,1,1,1] \\
11 & [0,1,1,0,1,1,1,1,0,1,1,0,0,0,0,1,0,0,0,0,0,0,0,1,1,1] \\
12 & [0,1,1,0,1,1,1,1,0,1,1,1,0,0,0,1,0,0,0,0,0,0,0,0,1,1] \\
13 & [0,1,1,0,1,1,1,1,0,1,1,1,1,0,0,1,0,0,0,0,0,0,0,0,1,0] \\
14 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,0,1,0,0,0,0,0,0,0,0,1,0] \\
15 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,0] \\
16 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,0] \\
17 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,0,0,0,0,0,0,0,1,0] \\
18 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,0,0,0,0,0,0,1,0] \\
19 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,0,0,0,0,0,1,0] \\
20 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,0,0,0,0,1,0] \\
21 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,0,0,0,1,0] \\
22 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,0,0,1,0] \\
23 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,0] \\
24 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0] \\
25 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0,0] \\
26 & [0,1,1,0,1,1,1,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,0,1]
\end{array}
$$

En este caso, las puertas que quedaron abiertas fueron las puertas 1, 4, 9, 16 y 25. Que casualmente son los cuadrados perfectos menores a 26. Esto sucede porque un número n tiene un número impar de divisores si y solo si es un cuadrado perfecto. Por ejemplo el 25 tiene como divisores a 1, 5 y 25.

Entonces las puertas que quedarán abiertas serán las puertas i^2 con $i \leq \sqrt{n}$. Por lo que el número de puertas abiertas será $\lfloor \sqrt{n} \rfloor$, esta función nos da el entero más cercano hacia abajo.