DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Problemas, Algoritmos y Programación

Tp Nº 1

Septiembre 2016 PAP

Número de grupo: 8192896574593398

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Problema 1: La tienda de Apu **3**
1.1. Introducción: . . . 3
1.2. Pseudocódigo: . . . 3
1.3. Complejidad: . . . 4
1.4. Correctitud: . . . 5

2. Problema 2: El cumpleaños de Lisa **6**
2.1. Introducción: . . . 6
2.2. Pseudocódigo: . . . 6
2.3. Correctitud: . . . 7
2.4. Complejidad: . . . 9

3. Problema 3: Experimentos nucleares **10**
3.1. Introducción: . . . 10
3.2. Pseudocódigo: . . . 10
3.3. Correctitud: . . . 10
3.4. Complejidad: . . . 10

4. Problema 4: El error de Smithers **11**
4.1. Introducción: . . . 11
4.2. Pseudocódigo: . . . 11
4.3. Correctitud: . . . 11
4.4. Complejidad: . . . 12

1. Problema 1: La tienda de Apu

1.1. Introducción:

Homero quiere ir a la tienda de Apu a comprar todas las donas que pueda con el presupuesto que maneja (un solo billete). Apu tiene en venta packs de donas de cantidades variables, los cuales no permite fraccionar, el pack se compra completo o no se compra. Por suerte, cada pack cuesta tanto como la cantidad de donas que tenga. Homero quiere saber cuantas donas puede comprar sin irse de su presupuesto.

Figura 1: Un hombre con hambre e ilusiones.

De manera más formal, dado un billete de valor P y N packs de cantidades $D_1, D_2, \ldots, D_N$, parámetros del problema.

Sea $A \subseteq \{D_1, D_2, \ldots, D_N\}$ tal que maximiza:

$$\sum_{D \in A} D$$

teniendo en cuenta que

$$\sum_{D \in A} D \leq P$$

Se busca $x \in \mathbb{N}_0$, $x = \sum_{D \in A} D$.

1.2. Pseudocódigo:

Algoritmo 1: sumasConjuntosPartesDe

```
   Datos: vec ∈ ℕ^N
 1 res ← nuevoArreglo(2^|vec|);
 2 para mask ← 0 a |res| hacer
 3 |   suma_subconjunto ← 0
 4 |   mientras i ← 0 a |vec| hacer
 5 |   |   si (1 << i & mask) > 0 entonces
 6 |   |   |   suma_subconjunto ← suma_subconjunto + vec[i]
 7 |   |   fin
 8 |   fin
 9 |   res[mask] = suma_subconjunto
10 fin
11 ordenar(res)
12 retornar res
```

Donde:

- Se asume se tiene una función *nuevoArreglo(n)* la cual devuelve un arreglo de longitud n en θ(n).
- *mask* es un entero utilizado como una máscara de bits para representar en cada iteración lo elementos que están incluídos en el subconjunto, y así generar el conjunto de partes.

Dado un *multiconjunto* **vec** (un conjunto que admite elementos repetidos) implementado con un arreglo, se devuelve un arreglo que contiene exactamente todos los elementos del conjunto $\left\{\sum_{x\in P} x \mid P \in partesDe(\textbf{vec})\right\}$ ordenados de forma ascendente.

Algoritmo 2: maximaCantidadDeDonas

```
   Datos: packs ∈ ℕ^N, P ∈ ℕ
 1 res ← 0
 2 mitad_packs_1 ← sumasConjuntosPartesDe( packs[0, n/2) )
 3 mitad_packs_2 ← sumasConjuntosPartesDe( packs[n/2, n) )
 4 j ← |mitad_packs_2| − 1
 5 para i ← 0 a |mitad_packs_1| hacer
 6 |   mientras mitad_packs_1[i] + mitad_packs_2[j] > P ∧ 0 ≤ j hacer
 7 |   |   j ← j - 1
 8 |   fin
 9 |   si mitad_packs_1[i] + mitad_packs_2[j] ≤ P ∧ res < mitad_packs_1[i] + mitad_packs_2[j]
   |    entonces
10 |   |   res ← mitad_packs_1[i] + mitad_packs_2[j]
11 |   fin
12 fin
13 retornar res
```

Donde

- $packs = [D_1, D_2, \ldots, D_N]$ los packs disponibles en la tienda de Apu. N y *packs* parámetros del problema.
- Se asume que *vec*[0,r) devuelve un subarreglo de *vec* de tamaño r y con los elementos desde la posición 0 hasta r - *1* inclusive.

Se divide el arreglo de N packs en dos arreglos de tamaño $\frac{N}{2}$ y se llama a la función *sumasConjuntosPartesDe* con cada mitad. Esta devuelve dos arreglos ordenados de tamaño $2^{\frac{N}{2}}$ que se recorren uno de manera ascendente y otro descendente buscando a cada paso un sumando en cada uno que no sumen más que P.

Por último se retorna el máximo de las sumas obtenidas.

1.3. Complejidad:

Se puede ver que en la función `sumasConjuntosPartesDe`, dado un arreglo de tamaño N parámetro de la funcíon:

- Se crea otro de tamaño 2^N lo cual tiene una complejidad temporal $\theta(2^N)$.
- Se itera 2^N veces sobre un bloque de operaciones que realiza una operación con un costo temporal O(1) y otro ciclo, el cual itera N veces un bloque de operaciones con un costo temporal O(1).
- Se ordena un arreglo de tamaño 2^N, con un costo $\mathrm{O}(2^N \cdot log(2^N)) = \mathrm{O}(2^N \cdot N)$.

Por lo tanto se deduce que `sumasConjuntosPartesDe` tiene una complejidad temporal $O(2^N \cdot N)$.

En la función `maximaCantidadDeDonas`, dado un arreglo de tamaño N parámetro de la misma:

- Se hacen dos llamados a la función `sumasConjuntosPartesDe` con una entrada de tamaño $\frac{N}{2}$, cada una con una complejidad temporal de $O(2^{\frac{N}{2}} \cdot \frac{N}{2})$.
- Se itera $2^{\frac{N}{2}}$ veces un bloque de operaciones en las cuales se encuentra un conjunto de operaciones con un costo temporal $O(1)$ y un ciclo. Este último ciclo interno itera a lo sumo $2^{\frac{N}{2}}$ veces entre todas las iteraciones del ciclo principal, por lo que todo el ciclo tienen una complejidad temporal $O(2^{\frac{N}{2}})$.

Se concluye que la complejidad temporal de `maximaCantidadDeDonas` es $O(2^{\frac{N}{2}} \cdot \frac{N}{2} \cdot 2 + 2^{\frac{N}{2}}) = O(2^{\frac{N}{2}} \cdot N)$. La cual al ser el algoritmo principal, es la complejidad de la solución.

1.4. Correctitud:

El algoritmo recorre dos arreglos ordenados cuyo primer elemento es 0 (que corresponde a no tomar ningún *pack* de donas para cada conjunto), uno de manera creciente el otro decreciente.

Sea $B = sumasConjuntosPartesDe(\{D_1, D_2, \ldots, D_{\frac{N}{2}-1}\})$.

$C = sumasConjuntosPartesDe(\{D_{\frac{N}{2}}, D_{\frac{N}{2}+1}, \ldots, D_N\})$.

Cada índice dentro de B y C representa la sumatoria de todos los elementos correspondiente a un elemento en partes del conjunto que se le pasa por parámetros a *sumasConjuntosPartesDe*.

Si $B_0 + C_N > P$, entonces no es solución. Ademas, $(\forall 0 \leq i' < |B|)\ B_{i'} + C_j > P$, entonces C_N no podrá generar futuras soluciones. Por lo tanto no se contemplará C_N, y en el próximo paso se trabajará con $C[0, |C| - 1)$.

Si $B_0 + C_N \leq P$, entonces puede ser una posible solución por lo que se guarda en *res* el $MAX(res, B_0 + C_N)$. B_0 no generará una mejor solución que esta (pues si moves el otro indice, dejando este fijo, se obtiene un valor $P' \leq B_0 + C_N \leq P$), por lo que no se lo considerará para futuras decisiones. En el paso siguiente, solo se usará $B[1, |B|)$.

En cada iteración se puede asumir que *res* persiste el máximo valor obtenido utilizando todos los elementos que ya se descartaron. Análogamente se repiten los mismos dos casos anteriores siempre, pero sobre los subarreglos resultantes de acortarlos a cada paso.

2. Problema 2: El cumpleaños de Lisa

2.1. Introducción:

En este ejercicio buscamos resolver el siguiente problema: dado un conjunto de n amigas y una matriz simétrica $D \in \mathbb{R}^{nxn}$ donde D_{ij} representa la diversión que aportan las amigas i y j al estar en la misma fiesta, queremos encontrar la máxima diversión posible que podemos lograr organizando al conjunto de amigas en fiestas, de manera que todas participen en exactamente una fiesta. Debemos resolver el problema mediante un algoritmo que tenga complejidad temporal $O(3^n)$ en el peor caso. Para ello, construimos un algoritmo cuyos tres conceptos fundamentales son: máscaras de bits para codificar subconjuntos de amigas, programación dinámica para evitar repetir cálculos ya realizados, y recursión para obtener soluciones parciales que nos permitan calcular el resultado final.

2.2. Pseudocódigo:

Algoritmo 3: maximaDiversion

```
   Datos: n ∈ ℕ, D ∈ ℝ^{nxn}
 1 diversionFiestas[2^n] ← -1
 2 diversionMaxima[2^n] ← -1
 3 diversionFiestas[0] ← 0
 4 diversionMaxima[0] ← 0
 5 para (i ← 1 a 2^n − 1) hacer
 6 |   diversionFiestas[i] ← calcularDiversion(n, i, D)
 7 fin
 8 return maximaDiversionAux(n, 2^n-1, D, diversionFiestas, diversionMaxima)
```

Algoritmo 4: maximaDiversionAux

```
   Datos: n, mask ∈ ℕ, D ∈ ℝ^{nxn}, diversionFiestas, diversionAmigas ∈ ℝ^{2^n}
 1 si diversionMaxima[mask] != -1 entonces
 2 |   return diversionMaxima[mask]
 3 fin
 4 res ← -1
 5 para (i ← mask ; i != 0; i = mask & (i − 1)) hacer
 6 |   si diversionFiesta[i] ≥ 0 entonces
 7 |   |   res ← max(res, maximaDiversionAux(n, maskˆi, D, diversionFiestas,
   |   |     diversionMaxima) + diversionFiestas[i])
 8 |   fin
 9 fin
10 diversionMaxima[mask] ← res
11 return res
```

Algunas aclaraciones sobre el pseudocódigo:

- *mask* representa el subconjunto de amigas con el que estamos trabajando, codificado en una máscara de bits.
- $diversionFiestas[i]$ contiene la diversión asociada a la fiesta i.
- $diversionMaxima[i]$ contiene la máxima diversión que podemos alcanzar organizando las fiestas de alguna forma, para el conjunto de amigas representado por i.
- Cuando inicializamos los arreglos $diversionFiestas$ y $diversionMaxima$ en la función $maximaDiversion$, asumimos que la asignación de -1 es para todas las posiciones de los mismos.
- $calcularDiversion$ calcula la diversión de la fiesta dada, utilizando la matriz de relaciones entre las amigas. Dado que esta función es relativamente simple, omitimos su pseudocódigo. Sin em-

bargo será correctamente explicada en la sección de complejidad, y también puede ser consultada en el código que la implementa.
- En el pseudocódigo de *maximaDiversionAux* utilizamos el símbolo ^. Este representa a la operación XOR.

2.3. Correctitud:

En pocas palabras, la idea del algoritmo es ir calculando las diversiones de todos los posibles conjuntos de fiestas y a la vez ir verificando si esta es mayor que el máximo parcial que tenemos guardado. Cuando finalizamos, nos queda un valor final máximo, y esa es la solución. A continuación explicamos y justificamos detalladamente cada uno de los pasos que sigue el algoritmo.

El algoritmo que proponemos comienza calculando la diversión para cada una de las fiestas que se pueden construir. Cada fiesta es equivalente a un subconjunto del conjunto original de n amigas. Es decir, calculamos la diversión de un total de 2^n subconjuntos o fiestas. Hacemos esto de antemano, ya que sabemos que durante el desarrollo del algoritmo se requerirán todos y cada uno de estos valores, ya veremos más adelante por qué. Por lo tanto, lo único que hacemos en la función *maximaDiversion* es armar las estructuras y obtener información necesaria para poder invocar a *maximaDiversionAux* que es la encargada de calcular la solución buscada.

Es importante destacar el uso de las máscaras de bits como método de codificación de conjuntos. Este recurso nos permite iterar fácilmente a través de todos los subconjuntos del conjunto principal de amigas. Por ejemplo, si tenemos 5 amigas, el número $31 = (11111)_2$ representa el conjunto de las 5 amigas, $30 = (11110)_2$ el conjunto que contiene a todas las amigas menos la última, y así sucesivamente. Por lo tanto, todos los subconjuntos de un conjunto de n amigas estarán representados por algún $x \in [0, 2^n) \cap \mathbb{Z}$.

Entonces, lo primero que hacemos en *maximaDiversionAux* es chequear si la solución que buscamos ya fue previamente calculada. Obviamente en la primer llamada a la función no vamos a contar con dicha solución, pero como se trata de una función recursiva, más adelante sí encontraremos soluciones que nos sirvan, y evitaremos recalcularlas.

Si la solución no fue hallada en la estructura, procedemos a calcularla. Para ello inicializamos *max* en -1 (donde guardamos el máximo) e iteramos sobre todos los subconjuntos del conjunto definido por la máscara *mask* pasada como parámetro de la función. Y con esto, nos referimos precisamente al ciclo "for (i ← mask ; i != 0; $i = mask$ & $(i-1)$)"que en cada iteración redefine i como $mask$ & $(i-1)$. Redefinimos i de esta manera porque siempre nos interesa iterar sobre los subconjuntos de *mask*. Podríamos haber hecho "for (i ← mask ; i != 0; i=i-1)" y en cada iteración chequear si i es un subconjunto de *mask*, pero la primera opción es más eficiente. La idea de lo que hacemos con este ciclo es la siguiente: para cada subconjunto, asumimos que este es una fiesta del conjunto de fiestas para el cual estamos calculando la diversión y luego calculamos recursivamente la máxima diversión obtenible con el conjunto de amigas definido por las amigas incluidas en *mask* que no están incluidas en i. Es decir, el conjunto de amigas definido por $mask \triangle i$ (diferencia simétrica) o $mask$ - i (es equivalente pues $i \subseteq mask$). Luego tomamos como nuevo *max* el máximo entre *max* y $recursion + diversion(i)$. Veamos un ejemplo para que se entienda mejor:

Tenemos el conjunto de amigas {A, B, C, D} (n = 4) max = -1 (valor inicial)

$$\mathbf{D} = \begin{bmatrix} - & A & B & C & D \\ A & 0 & 2 & -1 & 0 \\ B & 2 & 0 & 0 & -1 \\ C & -1 & 0 & 0 & 2 \\ D & 0 & -1 & 2 & 0 \end{bmatrix}$$

(abreviamos *maximaDiversionAux* como *mDAux*)

- Primer iteración de ciclo:
  i = {A, B, C, D}

  $diversion(i) = 2$

  $mDAux(\{A, B, C, D\} \triangle \{A, B, C, D\}) = mDAux(\{\ \}) = 0$

  max = maximo(max, $mDAux(\{\ \})$ + $diversion(\{A, B, C, D\})$) = maximo(-1, 0 + 2) = 2
- Segunda iteración de ciclo:
  i = {A, B, C}

  $diversion(i) = 1$

  $mDAux(\{A, B, C\} \triangle \{A, B, C, D\}) = mDAux(\{D\}) = 0$

  max = maximo(max, $mDAux(\{D\})$ + $diversion(\{A, B, C\})$) = maximo(2, 0 + 1) = 2
- Tercera iteración de ciclo:
  i = {A, B, D}

  $diversion(i) = 1$

  $mDAux(\{A, B, D\} \triangle \{A, B, C, D\}) = mDAux(\{C\}) = 0$

  max = maximo(max, $mDAux(\{C\})$ + $diversion(\{A, B, C\})$) = maximo(2, 0 + 1) = 2
- Cuarta iteración de ciclo:
  i = {A, B}

  $diversion(i) = 2$

  $mDAux(\{A, B\} \triangle \{A, B, C, D\}) = mDAux(\{C, D\}) = 2$

  max = maximo(max, $mDAux(\{C, D\})$ + $diversion(\{A, B\})$) = maximo(2, 2 + 2) = 4
- Y podemos seguir hasta terminar. Si bien no entramos en las recursiones, estas funcionan exactamente igual.

Podemos observar en el ejemplo cómo el algoritmo va llamandose recursivamente para resolver casos cada vez mas pequeños que sirven para el cálculo de la solución final. Es importante destacar el uso de programación dinámica. Si bien no se ve expresamente en el ejemplo, podemos dar cuenta de que vamos a estar accediendo a valores ya calculados. Por ejemplo, cuando lleguemos a la iteración 8 (i = {A}) y tengamos que calcular $mDAux(\{B, C, D\})$, en alguna iteración de este llamado necesitaremos el resultado de $mDAux(\{C, D\})$, y no hará falta calcularlo pues ya lo guardamos en la iteración 4 del ejemplo.

Otro punto a tener en cuenta es que en cada iteración chequeamos si la diversión de i es menor que 0. Si es así, ni nos gastamos en llamar a la recursión y pasamos a la siguiente iteración. Hacemos

esto, porque sabemos que una fiesta cuya diversión sea negativa no va formar parte del conjunto de fiestas solución, ya que siempre existe al menos una opción mejor, que es ubicar a esas amigas todas separadas en fiestas individuales, sumando 0.

Por último, queda claro lo que dijimos al inicio de la explicación: utilizamos todas y cada una de las diversiones calculadas inicialmente. Esto se debe a que en la primer llamada a $maximaDiversionAux$, $mask$ es 2^n - 1, que es el conjunto completo de amigas, por ende iteraremos sobre todos sus subconjuntos y requerimos de las diversiones de todos ellos.

2.4. Complejidad:

En esta sección analizaremos la complejidad temporal de todos los pasos que sigue el algoritmo propuesto, para así determinar la complejidad total del mismo.

En primer lugar veamos por separado la función $calcularDiversion$. Como su nombre indica, se encarga de calcular la diversión para el conjunto de amigas dado como parámetro, codificado en una máscara de bits. La función es simple; primero decodifica la máscara de bits y almacena el conjunto de amigas en un vector, y luego en base a dicho conjunto calcula la diversión. Para decodificar la máscara utiliza un ciclo que en cada iteración chequea para una amiga del conjunto original si forma parte del conjunto en cuestión y la agrega al vector en caso de que así sea. Por ende esta parte toma $O(n)$ ya que son n amigas en total. Por otro lado, para calcular la diversión, por cada amiga del vector obtenido, suma la diversión que aporta su amistad con cada una de las amigas que se encuentran a su derecha dentro del vector. Por lo tanto, si tenemos un conjunto de m amigas, para la amiga de la posición 0 se harán m-1 sumas, para la de la posición 1 se harán m-2 sumas, y así sucesivamente. Por lo tanto esta parte toma $\sum_{i=0}^{m-1} i = \frac{(m-1)m}{2}$ pasos, lo cual tiene orden $O(m^2) \subseteq O(n^2)$ (esto último porque $m \leq n$). Finalmente, la complejidad del peor caso de $calcularDiversion$ es $O(n^2)$.

Ahora analicemos el algoritmo principal. Lo primero que hacemos es inicializar 2 arreglos de 2^n elementos cada uno, y uno de ellos lo llenamos completo con los valores obtenidos a partir de $calcularDiversion$. Por lo tanto esta porción tiene complejidad:

$$\underbrace{O(2^n)}_{inicializacion\ arreglos} + \underbrace{O(n^2 * 2^n)}_{calculo\ diversiones} = O(n^2 * 2^n) = O(3^n)$$

(La última equivalencia el enunciado permite utilizarla sin necesidad de demostración)

Luego tenemos 2^n instancias, cada una de las cuales tiene un ciclo que itera sobre todos sus subconjuntos (esto en el peor de los casos, que es cuando todas las diversiones son no negativas). Entonces cada subconjunto de i elementos tiene que iterar a través de sus 2^i subconjuntos, y tenemos $\binom{n}{i}$ subconjuntos de i elementos. Luego, la cantidad de iteraciones para el total de los subconjuntos de i elementos es $\binom{n}{i}$*2^i. De aquí concluimos que el total de iteraciones es :

$$\sum_{i=0}^{n} \binom{n}{i} * 2^i = \underbrace{\sum_{i=0}^{n} \binom{n}{i} * 2^i * 1^{n-i}}_{Binomio\ de\ Newton} = (2+1)^n = 3^n$$

De esta manera, dividimos nuestro algoritmo en 2 partes, cada de las cuales tiene complejidad $O(3^n)$. Por lo tanto nos queda una complejidad final de $O(3^n)$, cumpliendo así con la complejidad solicitada por el enunciado.

3. Problema 3: Experimentos nucleares

3.1. Introducción:

En este ejercicio nos piden, dado un arreglo de enteros, hallar la suma máxima de una secuencia de números que sean consecutivos. Además, debe ser resuelto con una complejidad lineal.

3.2. Pseudocódigo:

Para resolver este problema, realizamos el siguiente algoritmo goloso:

Algoritmo 5: sumaMax

```
   Datos: int v[ ], int n
 1 maximo ← 0
 2 sumaActual ← 0
 3 para (i ← 0 a n) hacer
 4 |   sumaActual ← sumaActual + v[i]
 5 |   si sumaActual < 0 entonces
 6 |   |   sumaActual ← 0
 7 |   fin
 8 |   en otro caso
 9 |   |   maximo ← max(sumaActual,maximo)
10 |   fin
11 fin
12 return maximo
```

3.3. Correctitud:

Este algoritmo recorre el arreglo de manera lineal. A medida vamos avanzando va almacenando una suma parcial. Mientras esta suma sea positiva representará la suma de todos los elementos que ya fueron visitados durante el algoritmo. Cada vez que se agrega un elemento más a la suma, se compara con el máximo valor que alcanzo esta suma parcial hasta el momento. En caso de ser mayor, se actualiza este máximo.

Si esta suma parcial llegara a ser negativa en algún momento, se resetea la variable y pasa a valer 0, pues esta subsecuencia no será parte de la subsecuencia que tenga suma máxima. Antes que tener en cuenta una porción del arreglo que aporta una suma negativa, podemos dejarla de lado y no sumar nada negativo. Entonces, si caemos en esta situación, es como si el problema volviese a comenzar, y se continua iterando a lo largo del arreglo.

Finalmente, cuando termina de recorrerse, nos queda el resultado final guardado en la variable máximo. De esta manera, concluimos que el algoritmo propuesto resuelve el problema correctamente.

3.4. Complejidad:

Como se puede observar en el pseudocódigo, este algoritmo recorre un arreglo de tamaño N linealmente, una única vez. Sobre cada elemento del arreglo, el algoritmo realiza operaciones de comparación y/o asignación de costo $O(1)$, por lo tanto, el algoritmo tiene como complejidad final $O(N)$.

4. Problema 4: El error de Smithers

4.1. Introducción:

En este ejercicio se nos pide resolver el siguiente problema: dada una matriz M $\in \mathbb{Z}_{10007}^{3x3}$ y un arreglo de N matrices en $\mathbb{Z}_{10007}^{3x3}$, decidir si existe un subarreglo de longitud L cuyo producto sea igual a M. Se busca que el algoritmo diseñado tenga complejidad **O**(N log N)

4.2. Pseudocódigo:

Algoritmo 6: ProdMtxL

```
   Datos: Matriz M, vector<Matriz> listaMtx, int N, int L
 1 altura ← log2(L)+1
 2 vector<Matriz> vacio
 3 vector< vector<Matriz> > listaProductos(altura,vacio)
 4 listaProductos[0] ← listaMtx
 5 para (i ← 1 a altura) hacer
 6 |   int tamaño = listaProductos[i-1].size()-2^(i-1)
 7 |   Matriz vacia
 8 |   vector<Matriz> prod(tamaño,vacia)
 9 |   para (j ← 0 a tamaño) hacer
10 |   |   prod[j] ← listaProductos[i-1][j]*listaProductos[i-1][j+2^(i-1)]
11 |   fin
12 |   listaProductos[i] ← prod
13 fin
14 vector<Matriz> posiblesSubmatrices ← listaProductos[listaProductos.size()-1]
15 int cantMatricesUtilizadas ← 2^(altura-1)
16 mientras (cantMatricesUtilizadas<L) hacer
17 |   altura–
18 |   si (cantMatricesUtilizadas+2^(altura-1) ≤ L) entonces
19 |   |   para (i ← 0 a N − L + 1) hacer
20 |   |   |   posiblesSubmatrices[i] *= listaProductos[altura-1][i+cantMatricesUtilizadas]
21 |   |   fin
22 |   |   cantMatricesUtilizadas += 2^(altura-1)
23 |   fin
24 fin
25 para (i ← 0 a N − L + 1) hacer
26 |   si (posiblesSubmatrices[i] == M) entonces
27 |   |   return 1
28 |   fin
29 fin
30 return 0
```

4.3. Correctitud:

Una solución sencilla a este problema es calcular todos los subarreglos de longitud L, y para cada uno de estos, hacer el producto de las L matrices y verificar si el resultado es igual a M. Existen $N - L + 1$ subarreglos de longitud L y hacer el producto de cada uno tiene un costo **O**(L). Entonces la complejidad de esta solución seria **O**(L$(N - L + 1)$) = **O**(N^2), ya que L se puede acotar por N.
El problema con esta resolución, es que estamos realizando muchas veces un mismo producto de ma-

trices. Por ejemplo, $A_1...A_L$, $A_2...A_{L+1}$ y $A_3...A_{L+2}$, repiten el producto $A_3...A_L$.
Nuestro algoritmo hace las multiplicaciones de manera inteligente. Primero obtenemos todas las multiplicaciones de 2 matrices, luego de 4 matrices, de 8, de 16, y asi sucesivamente. Para calcular cada producto de 2^i matrices, solo utilizamos los productos entre las 2^{i-1} matrices calculados en el paso anterior.
Si L es potencia de 2, entonces es fácil ver que calculamos los productos de todos los subarreglos de longitud L.
Si $2^i < L < 2^{i+1}$, entonces L puede escribirse como la suma de potencias de 2 (por ejemplo, 13 = 1 + 4 + 8). Esto es fácil de ver, escribiendo las potencias de 2 en codificación binaria.
Sea $L = 2^i + 2^{j_0} + ... + 2^{j_k}$ con $j_0 \neq ... \neq j_k < i$. Entonces calculamos el producto de longitud L como el producto $P_i * P_{j_0} * ... * P_{j_k}$ siendo P_h el resultado de multiplicar 2^h matrices.
Entonces podemos calcular el producto de todos los subarreglos de longitud L. Luego verificamos si alguna es igual a M.

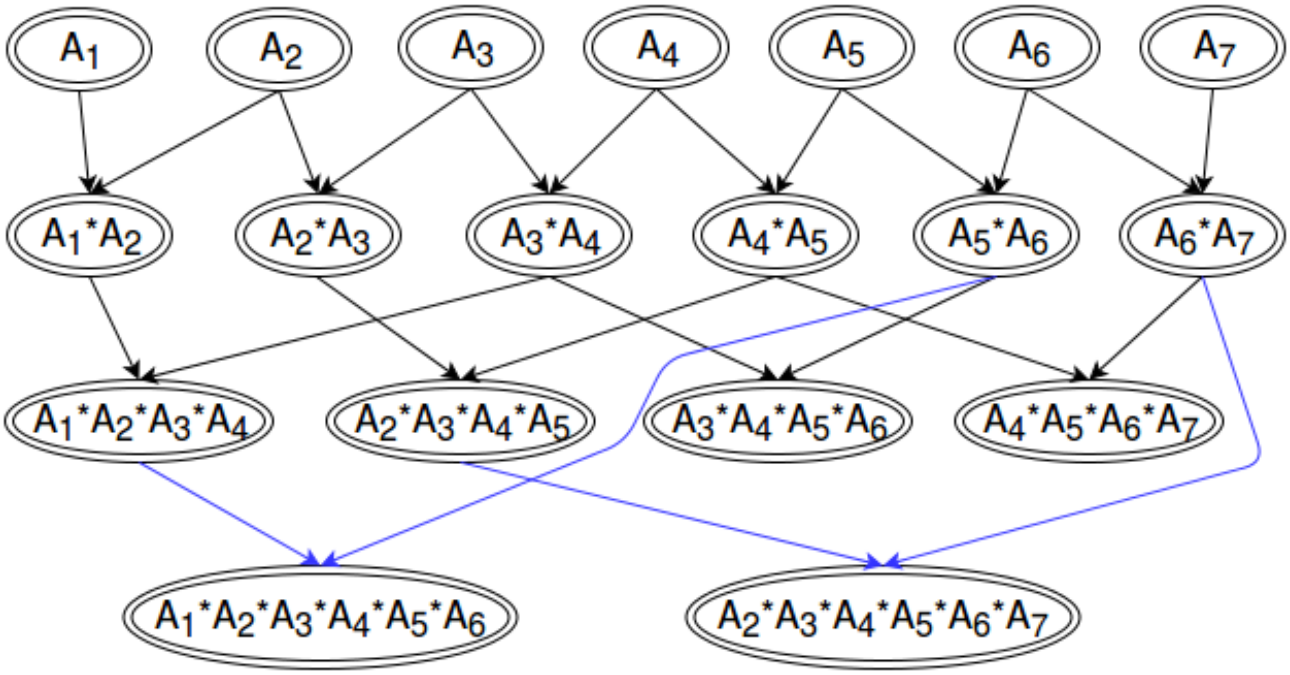

Figura 2: Ejemplo de nuestro algoritmo, con $L = 6$ y $N = 7$.

Ahora veamos que se cumple la complejidad pedida.

4.4. Complejidad:

Supongamos que L es potencia de 2, es decir, $L = 2^k$ para algun k.
Para calcular los productos de todos los subarreglos de tamaño $L = 2^k$ calculamos los productos de todos los subarreglos de tamaño 2^0, 2^1, ..., 2^{k-1}.
Calcular los productos de todos los subarreglos de tamaño 2^j, ya sabiendo los de 2^{j-1} es lineal (en función del tamaño del arreglo). Pues solo se necesitan dos elementos del producto de 2^{i-1} (y son accesibles en $\mathbf{O}(1)$).
Entonces la compejidad es $\mathbf{O}(N.k) = \mathbf{O}(N\ log_2(L)) = \mathbf{O}(N\ log_2(N)) = \mathbf{O}(N\ \log(N))$.
Si L no es potencia de 2, entonces $2^k < L < 2^{k+1}$ para algun k. Realizando lo mismo que antes obtenemos el los producto de 2^k matrices. En el peor caso, debemos multiplicar por los productos de 2^0, 2^1, ..., 2^{k-1} ya calculados (y accesibles en $\mathbf{O}(1)$). Estos productos se realizan para N - L + 1 subarreglos de longitud 2^k

Entonces obtenemos un costo de $\mathbf{O}(N \log(N) + N.k) = \mathbf{O}(N \log(N) + N\ log_2(L)) = \mathbf{O}(2.N \log(N)) = \mathbf{O}(N \log(N))$.
Esta complejidad se puede verificar con facilidad con el pseudocódigo. Para la primer parte se tiene un ciclo que itera desde 1 hasta *altura* (es decir, $log_2(L)$). Dentro de ese ciclo se encuentra otro que itera, en el peor caso, de 0 a N. Para la segunda parte, tenemos un ciclo que como mucho va a iterar *altura* veces. En cada iteración se recorre un vector de tamaño $N - L + 1$. Entonces la complejidad es $\mathbf{O}(N \log(N))$.