Diseño y análisis de algoritmos. Tarea II

Sebastián Valencia Calderón
201111578

17 de junio de 2015

1. **Subarreglo de suma máxima**. Se quiere construir un programa que reciba un arreglo de números enteros y encuentre la suma más grande de un subarreglo de acuerdo con el esquema indicado.

   Para desarrollar el problema, se propone una notación alternativa a la provista por el enunciado. La notación propuesta, ofrece claridad, frescura y legibilidad. Sin embargo, ambas propuestas son equivalentes.

$$sumax(i) = \max_{0 \leq j < i} b[0..j]$$

$$sumaxf(i) = \max_{0 \leq j < i} b[j..i-1]$$

$$P : 0 \leq i \leq n \wedge p = sumax(i) \wedge q = sumaxf(i)$$

   a) Explique qué técnica pudo haberse utilizado para proponer el invariante P.

   La técnica utilizada, se basa en reducir el espacio de búsqueda de la respuesta a través de ir progresivamente apretando el cerco de búsqueda, es decir, por un lado, se tiene la suma máxima de los subarreglos contenidos en $b[0..j], 0 \leq j < i$; por otra parte, se tiene los subarreglos cuyo último índice es $i - 1$, es decir, se tiene acotado el espacio de búsqueda y lo que debe hacer el bucle es reducir éste espacio. La solución propuesta, cumple con las especificaciones pedidas, sin embargo, usa una notación completamente distinta a la del enunciado. A continuación, se introduce la notación, se incluye su definición y la equivalencia con la del enunciado.
   Sea $s_{j,k}$ la suma del arreglo entre los índices $j, k, 0 \leq j < n, 0 \leq k < n$, de la misma forma, sea $Q_k = \text{máx}\left\{0, \text{máx}_{1 \leq j \leq k}\left\{s_{j,k}\right\}\right\}$, luego para una arreglo a de tamaño mayor o igual a tres, $Q_3 = \text{máx}\left\{0, \text{máx}_{1 \leq j \leq 3}\left\{s_{j,3}\right\}\right\} = \text{máx}\left\{0, \text{máx}(s_{0,3}, s_{1,3}, s_{2,3}, s_{3,3})\right\}$. El último resultado es igual a:

$$Q_2 = \text{máx}\left\{0, \text{máx}\left\{\sum_{i=0}^{2} a_i, \sum_{i=1}^{2} a_i, \sum_{i=2}^{2} a_i\right\}\right\}$$

$$Q_3 = \text{máx}\left\{0, \text{máx}\left\{\sum_{i=0}^{3} a_i, \sum_{i=1}^{3} a_i, \sum_{i=2}^{3} a_i, \sum_{i=3}^{3} a_i\right\}\right\}$$

El cero, descarta las sumas con números negativos. Por lo que la solución no sirve para números negativos, de todas maneras, el enunciado advierte que son números enteros, y la especificación que son naturales. Si son naturales, la respuesta es trivial: todo el arreglo hace la suma máxima. Aquí, se asume que debe haber al menos un positivo en el arreglo por la ambiguedad del enunciado. Si $Q_k > 0$, entonces, esto es la suma del máximo subarreglo cuyo último índice es k. Por lo tanto, por la definición misma del problema y de Q_k, lo que se desea hallar, es:

$$\max_{0 \leq k < n} \{Q_k\}$$

Ahora, es necesario calcular una regla de computación eficiente para Q_t, para ésto, se sigue la siguiente deducción:

$$k \geq 1 \wedge Q_k \geq 0 \Rightarrow Q_k = a_{[k:k]} \oplus Q_k = Q_{k-1} + a_k \oplus Q_k = 0$$

Ésta última, sirve para crear una tabla de memoización para calcular Q_k, pues $Q_k = \text{máx}_{0 \leq k < n} \{0, Q_{k-1} + a_k\}$. Las equivalencias con la notación original son: $q = Q_k, 0 \leq k < n$, y $p = \text{máx}_{0 \leq k < n} \{Q_k\}, 0 \leq k < n$.

b) Desarrolle un algoritmo que satisfaga la especificación indicada (operación básica: suma).

Para el desarrollo del algoritmo, siguiendo la especificación dada, se sigue la notación introducida en el literal anterior. A continuación se muestra el algoritmo con las anotaciones de la lógica de Floyd-Hoare debidas.

MAX-SUBARRAY(b : **array**$[0 \,..\, n-1]$ **of** $\mathbb{Z}$) : $\mathbb{N}$

1 Q : **array**$[-1 \,..\, n-1]$ **of** $\mathbb{Z}$
2 $Q_{-1} \leftarrow 0$
3 **for** $k \leftarrow 0$ **to** $n-1$ **do**
4 $\quad Q_k \leftarrow \text{máx}_{0 \leq k < n} \{0, Q_{k-1} + b_k\}$
5 $p, q \leftarrow 0, 0$
6 $\{Inv : 0 \leq i \leq n \wedge p = \text{máx}_{0 \leq i < n} \{Q_i\} \wedge q = Q_i\}$
7 **for** $i \leftarrow 0$ **to** $n-1$ **do**
8 $\quad q \leftarrow Q_i$
9 $\quad p \leftarrow \text{máx}(p, q)$
10 $\{p = \text{máx}_{0 \leq k < n} \{Q_k\}\}$
11 **return** p

c) Estime las complejidades temporal y espacial de su solución. Explique sus respuestas.

La complejidad temporal, se estima calculando las frecuancias de ejecución de cada operación y multiplicando éstas frecuancias con el costo de cada operación, luego se suman éstos resultados. Los costos de indexado, asignación, suma, y evaluación del máximo son $\Theta(1)$. De la linea 1 a la linea 2, se tiene un costo de 1, en la linea 3, a 4, se realiza n veces una operación de costo $\Theta(1)$, en la linea cinco, se realizan dos operaciones de costo $\Theta(1)$, de la linea 7 a la nueve, se realiza n veces dos operaciones de costo $\Theta(1)$ cada una. El máximo cuesta $\Theta(1)$, por que se presume que es una comparación únicamente. La complejidad temporal es:

$$T_t(n) = \Theta(1) + \left(\sum_{k=0}^{n-1} \Theta(1)\right) + \Theta(1) + \left(\sum_{k=0}^{n-1} \Theta(1)\right)$$

$$T_t(n) \sim 1 + \left(\sum_{k=0}^{n-1} 1\right) + 1 + \left(\sum_{k=0}^{n-1} 1\right)$$

$$T_t(n) \sim 1 + [n - 1 + 1] + 1 + [n - 1 + 1] \sim \Theta(n)$$

Para la complejidad espacial, predomina el espacio ocupado por la tabla de memoización, éste tamaño es $n + 1 \sim \Theta(n)$, luego, $T_s(n) \sim \Theta(n)$.

2. **Orden lexicográfico**. Sea A un alfabeto y A^*, el conjunto de palabras construidas con letras de A. Sobre A se supone entendido (y disponible en el lenguaje de programación) un orden $. < .$, determinado por la secuencia en que se nombran sus elementos. Por ejemplo, si A es ASCII, A^* son palabras con caracteres ASCII en minúsculas, y se entiende –por ejemplo– que $a < b < c \ldots < z$.
Considere el problema de, dadas dos palabras $a, b \in A^*$, decidir si $a <_{lex} b$, donde $. <_{lex} .$ es el orden lexicográfico correspondiente a $. < .$.
Para palabras en A, use la siguiente notación:

- ε, la palabra vacía
- $|x|$, longitud de x
- x_i, i-ésima letra de x $(1 \leq i \leq |x|)$
- $esvac(x)$, x es una palabra vacía
- $equals(x, y)$, la palabra x es igual a la palabra y
- $x_{p,q}$, la subpalabra de x con las letras desde la posición p hasta la posición q. $(1 \leq p \leq q \leq |x|)$

a) Especifique el problema (Contexto, Pre-, Poscondición).

El problema consiste, en determinar si el primer parámetro de una función es menor lexicográficamente al segundo parámetro, ambos, del tipo A^*, donde $*$, es la estrella de Kleene. Por lo tanto, el contexto está dado por los argumentos de la función.

Aparte, se introducen las variables y su respectivo tipo que serán usados para el desarrollo del programa.

$$\textbf{Ctx} : a, b \in A^*$$

$$\textbf{Var} flag \in \mathbb{Z}^2 \cup \{-1\}, ans \in \mathbb{B}, i \in \mathbb{N}$$

$$\textbf{Pre} : True,\ \textbf{Post} : ans \leftarrow (flag = -1)\ ?\ True : False \wedge ans \leftarrow a <_{lex} b$$

b) Proponga un invariante P y una cota Cot para contribuir a desarrollar el programa solución con un ciclo.

Para el desarrollo del programa, se propone que se maneje un caso base, éste es el de la palabra vacia, si $esvac(a)$, entonces es menor que cualquier b por definición. Ahora, se decide iterar mientras i sea menor al mínimo de las longitudes de ambas cadenas ($0 \leq i < \text{mín}(|a|, |b|) - 1$), y además $a_i = b_i$, luego, a través de éste ciclo se sabe hasta que índice j las longitudes son iguales. Ahora, dependiendo del valor del siguiente índice ($j + 1$), se sabe el orden de las palabras. Por lo tanto:

$$\textbf{Inv} : a_{0,i} = b_{0,i} \wedge k, i \in [0, \text{máx}(|a|, |b|)]$$

$$\textbf{Cot} : \text{mín}(|a|, |b|) - 1$$

c) Escriba código que satisfaga lo anotado en a y b.

```
MIN-LEX(a, b : A*) : B
1  flag, i ← 0, 0
2  if ESVAC(a) then flag ← −1
3  {a_{0,i} = b_{0,i}}
4  while i < mín(|a|, |b|) − 1 ∧ a_i = b_i do
5      i ← i + 1
6  {flag = 0 ∨ flag = −1}
7  return flag = −i ∨ a_i < b_i ▷ Lazy evaluation
```

d) Calcule la complejidad temporal de su solución (operación básica: comparación de letras). Explique su respuesta.

Se sigue la estrategia del primer punto, es decir ponderar frecuencias por costos, el costo de indexación, el de comparación de caractéres y asignación básica es $\Theta(1)$. La linea 1, tiene un costo de $\Theta(1)$, pues realiza tres operaciones de éste costo. Asimismo, las lineas 2, y 5 tienen operaciones de costo $\Theta(1)$. Por otra parte, la linea 4, se ejecuta por lo menos $\text{mín}(|a|, |b|) - 1$ veces, luego, se tiene:

$$T_t(|a|, |b|) = \Theta(1) + \sum_{i=0}^{\text{mín}(|a|,|b|)-1} \Theta(1) \sim 1 + \sum_{i=0}^{\text{mín}(|a|,|b|)-1} 1$$

$$T_t(|a|, |b|) = O(\text{mín}(|a|, |b|))$$

e) Estime la complejidad espacial de su solución.

No se hacen consideraciones de espacio distintas a las tres variables de tipo básico usadas, por lo tanto, $T_s(|a|, |b|) = \Theta(1)$

3. **Búsqueda lexicográfica**. Con la notación del númeral 2, suponga que se tienen un arreglo pal$[0 \,..\, n-1] : A*$, ordenado lexicográficamente en orden ascendente y una palabra $x \in A*$, tal que $x \in pal$. Desarrolle un algoritmo para encontrar un $i, 0 \leq i \leq n$, tal que $equals(x, pal[i])$.

   Puede suponer conocida una función $menlex(x, y)$ que decide, para palabras $x, y \in A*$, si $x <_{lex} y$.

   a) Especifique el problema (Contexto, Pre-, Poscondición).

   El problema es el de buscar la posición de la palabra en un arreglo ordenado lexicográficamente de palabras. La estrategia elegida para el desarrollo del programa es dividir, conquistar y combinar. Ya que el arreglo está ordenado ascendentemente, se puede comparar el elemento en la mitad del arreglo con el elemento que se busca, si son iguales, el índice es el correspondiente a la mitad del arreglo. Si el elemento buscado es menor al elemento de la mitad, se busca ahora en la primera mitad del arreglo, de otra forma, se busca en la mitad con elementos mayores al elemento de la mitad. Estos pasos, sugieren una aproximación recursiva al problema.
   A continuación se enuncian el contexto, y la precondición y la postcondición.

$$\textbf{Ctx} : pal : \textbf{array}[0 \,..\, n-1] \textbf{ of } A^*, x \in A^*$$

$$\textbf{Var} : ans \in [0 \,..\, n-1] = \mathbb{Z}_n,\ mid,\ high,\ low \in \mathbb{Z}_n,\ found \in \mathbb{B} \wedge$$
$$\exists i :\ 0 \leq i < n : pal[i] = x$$
$$\textbf{Pre} : \forall i, j : 0 \leq i < j \leq n \Rightarrow pal[i] \leq pal[j]$$
$$\textbf{Post} : mid = ans \wedge pal[ans] = x$$

   b) Proponga un invariante P y una cota t que sirva para resolver el problema.

$$\textbf{Inv} : 0 \leq mid < n \wedge mid = \frac{low + high}{2} \wedge 0 \leq low < high < n, ans \in \{-1, mid\}$$
$$\textbf{Cot} : high - low$$

   El cuerpo del método (ya que se sabe que $\exists i, 0 \leq i < n : pal[i] = x$) lleva a que $low = high + 1$, ya que en cada avance del ciclo o se suma uno a la cota mínima o se resta uno a la máxima, cada vez cortando en la mitad el espacio de búsqueda, por lo que una cota más apropiada podría ser, $\log_2(high - low)$.

c) Escriba código que satisfaga lo anotado en 3a y 3b.

Se incluyen las implementaciones recursiva e iterativa de la solución. Es necesario considerar que el código no corre sobre una máquina real, luego, pasar un arreglo como parámetro, no requiere una copia del mismo, es decir, se pasa por referencia. Asimismo, se omiten consideraciones de $overflow$ para el cálculo de la posición de la mitad. Ambas versiones, son equivalentes, el código recursivo es una aproximación natural al problema, mientras el código iterativo es la deducción natural del invariante y la cota. Resulta fácil pasar de la versión recursiva a la iterativa por que el avance recursivo, es determinado por una instrucción condicional.

Para el desarrollo del algoritmo, es necesario tener el algoritmo desarrollado en el literal anterior (MIN-LEX(a, b)), y además, saber la complejidad asociada la función *equals*, a continuación, se incluye una implementación de ésta función para una máquina secuencial:

```
EQUALS(a, b : A*; i ∈ ℤ) : 𝔹
1  if |a| ≤ i ∧ |b| ≤ i then
2      return True
3  else if |a| ≤ i ∨ |b| ≤ i then
4      return False
5  else
6      if a_i = b_i then
7          return EQUALS(a, b, i + 1)
8      else return False
```

Para el cálculo de las complejidades, se asume que la complejidad de obtener la longitd de una palabra es: $\Theta(1)$. La complejidad temporal de EQUALS es:

$$T_t(|a|, |b|) = \Theta(1) + T_t(|a| - 1, |b| - 1) \sim \Theta(\text{mín}\,|a|, |b|)$$

La función $equals(a,\ b)$ del enunciado es: EQUALS$(a,\ b,\ 0)$.

Ésto se debe a que el caso base se obtiene cuando i sea mayor a alguna de las longitudes, luego, se llama la recursión hasta que i sea mayor a la menor de las longitudes.

```
SEARCH-LEX-ITER(pal : array[0 .. n − 1] of A*; x : A*) : ℕ
1  low, high, ans, mid, found ← 0, n − 1, −1, 0, False
2  while low ≤ high ∧ ¬found do
3      mid ← (high + low)/2
4      if EQUALS(pal[mid], x, 0) then
5          ans ← mid; found ← True
6      else if MIN-LEX(x, pal[mid]) then high ← mid − 1
7      else low ← mid + 1
8  return ans
```

La versión recursiva es:

```
SEARCH-LEX-REC(pal : array[0 .. n − 1] of A*; low, high : ℕ; x : A*) : ℕ
1  mid ← (high + low)/(2)
2  if EQUALS(pal[mid], x, 0) then
3      return mid
4  else if MIN-LEX(x, pal[mid]) then
5      return SEARCH-LEX-REC(pal, low, mid − 1, x)
6  else
7      return SEARCH-LEX-REC(pal, mid + 1, high, x)
8  return ans
```

La segunda versión debe llamarse como SEARCH-LEX-REC($pal,\ 0,\ n-1,\ x$). Esta última versión, se utiliza para el análisis del algoritmo.

d) Calcule la complejidad temporal de su solución (operación básica: comparación de letras). Explique su respuesta.

La complejidad de EQUALS y de MIN-LEX, son ambas $\Theta(\text{mín}\,|a|\,,|b|)$. Para hallar la complejidad temporal de la solución ($T_t(n,|x|)$), es necesario considerar la complejidad de los procedimientos de los cuales la solución depende. La linea 1, tiene una complejidad de $\Theta(1)$, la linea 2, depende de la llamada a EQUALS, la cual tiene una compleidad de $\Theta(\text{mín}\,|a|\,,|b|)$, la linea 4, depende de la computación de MIN-LEX, la cual tiene una complejidad de $\Theta(\text{mín}\,|a|\,,|b|)$, las lineas 5, 7 cada una, tienen una compleidad de $T_t(n/2,|x|)$. La ecuación de recurrencia para $n \neq 1$ es:

$$T_t(n,|x|) = \Theta(\text{mín}\,|a|\,,|b|) + \Theta(1) + \Theta(\text{mín}\,|a|\,,|b|) + T_t(n/2,|x|)$$

$$T_t(n,|x|) = \begin{cases} 1 & \text{if } n = 1 \\ 2\Theta(\text{mín}\,|a|\,,|b|) + \Theta(1) + T_t(n/2,|x|) & \text{if } n \equiv 1. \end{cases}$$

En ésta última ecuación, los valores de a y b, son en éste contexto de: $pal[mid]$ y x respectivamente. Para el análisis, es necesario analizar la distribución de las longitudes de las letras en un alfabeto A, una aproximación sencilla es:

$$A = \{\text{"}a\text{"},\ \text{"}b\text{"},\ \text{"}c\text{"}\ \} \Rightarrow A^* = \{\varepsilon,\ \text{"}a\text{"},\ \text{"}b\text{"},\ \text{"}c\text{"},\ \text{"}ab\text{"},\ \text{"}aa\text{"},\ \text{"}ac\text{"}, ...,\ \text{"}aaa\text{"}, ...\}$$

Sea $L(A,n)$ el número de letras en A con longitud n. Es fácil ver que:

$$L(A,0) = 1,\ L(A,1) = 3,\ L(A,2) = 9, ..., L(A,k) = 3^k$$

Ésta función crece rápido con respecto a k, si $k = 10$, $L(A,k) = 59049$, se deduce que en promedio $|x| < |pal[mid]|$, por lo tanto, la ecuación de recurrencia ($n \neq 1$) se puede reescribir como:

$$T_t(n, |x|) = 2\,(O(|x|)) + 1 + T_t(n/2, |x|) \sim T_t(n/2, |x|) + 2\,|x| + 1$$
$$T_t(n, |x|) = T_t(n/2, |x|) + 2\,|x| + 1$$
$$T_t(n, |x|) = T_t(n/4, |x|) + 2\,|x| + 1 + 2\,|x| + 1$$
$$T_t(n, |x|) = T_t(n/8, |x|) + 2\,|x| + 1 + 2\,|x| + 1 + 2\,|x| + 1$$
$$T_t(n, |x|) = T_t(n/2^k, |x|) + k(2\,|x| + 1)$$

El último paso, se comprueba con inducción.
En particular $T_t(n/2^k, |x|) = T_t(1, |x|) \iff 2^k = n \Rightarrow k = \log_2(n)$

$$T_t(n, |x|) = T_t(1, |x|) + (2\,|x| + 1)\log_2(n)$$
$$T_t(n, |x|) = 1 + (2\,|x| + 1)\log_2(n)$$
$$T_t(n, |x|) \sim 2\,|x|\log_2(n) = O(2\,|x|\log_2(n))$$

En particular, si $|x| \ll n \Rightarrow T_t(n, |x|) = O(\log_2(n))$. Si la complejidad de las funciones involucradas co se deben tener en cuenta, ésta es la complejidad.
En general, $T_t(n, |x|) = O(|x|\log_2(n))$

e) Estime la complejidad espacial de su solución.

Teniendo en cuenta las consideraciones de paso por referencia, ambas soluciones son $T_s(n, |x|) = \Theta(1)$.

4. **Embaldosamientos**. Se desea pavimentar un camino rectangular de dimensiones $1 \times N$ con losas de dimensiones $1 \times k$, para $k = 1, 2, \ldots, M$. Se quiere determinar de cuántas maneras puede llevarse a cabo la pavimentación. Diseñe un algoritmo de programación dinámica que resuelva el problema.

   a) Construya su solución de acuerdo con la "receta para programación dinámica" que se vio en clase (lenguaje, recurrencia, ...) para resolver el problema.

   Dado que las dimensiones del camino son $1 \times N$, y las de cada losa son $1 \times k,\ 0 < k \leq M$, se desea saber cuántas maneras hay de cubrir N unidades lineales con segmentos de k unidades lineales para $0 < k \leq M$.

   - **Lenguaje**. Sea $C(N, M)$, el número de maneras de pavimentar un camino de dimensiones $1 \times N$, usando losas de hasta $1 \times M$ dimensiones. Las losas, pertenecen a un multi-conjunto de la forma: $L = \{L_1, L_2, \ldots, L_M\}$, donde cada $L_i,\ 0 < i \leq M$. $C(N, M)$, puede parametrizarse de la siguiente forma:

     $$C(N,\ M) = \begin{cases} |A| : A \cap \{S_m\} = \emptyset \\ |A| : c(A,\ S_m) = 1 \end{cases}$$

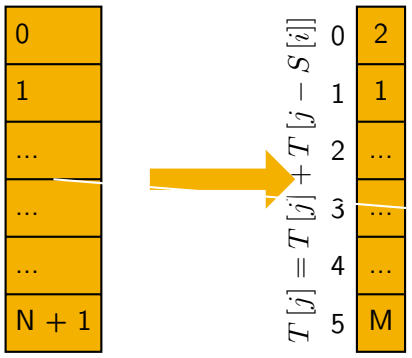
     Es decir, $C(N,\ M)$, depende de los multiconjuntos que no contengan a S_m, y a de los que lo exactamente una vez. Es necesario decir que $N, M \in \mathbb{N}$

- **Recurrencia**. Es claro ver que las ramas de la definición de $C(N, M)$ son cada una $C(N,\ m-1)$ y $C(N-S_m, m)$ respectivamente. Por lo tanto:

$$C(0,\ m) = 1$$

$$C(N,\ M) = C(N, m-1) + C(N-S_m, m)$$

- **Diagrama de necesidades**.



- **Invariante**.

$$\textbf{Inv} : 1 \leq S_k \leq M,\ 0 \leq k < M\ \wedge\ T_{j-1} = C(N,\ j-1)\ \wedge\ T_j = C(N,\ j)$$

$$0 < j < N+2$$

- **Estructura de datos**. La estructura de datos, son dos arreglos, uno con la dimension más grande de las losas para cada k, y otro con $C(N,\ j)$.

b) Estime complejidades temporal (operación básica: asignación) y espacial de su solución para 4a.

Como es necesario tener dos arreglos, la complejidad espacial es:

$$T_s(N,) = \Theta(\text{máx}(N,\ M))$$

Como se requiere llenar la tabla de las losas, y recorrerla para cada iteración sobre la tabla principal, la complejidad espacial es:

$$T_t(N,\ M) = \Theta(NM)$$

5. **3-Nim**. Suponga 3 montones con p_0, q_0, r_0 fichas, respectivamente, $q_0 \neq r_0$, $r_0 \neq p_0$, y dos jugadores, A y B. Los jugadores alternan turnos para quitar, de uno cualquiera de los montones, cualquier número de fichas. A es el primero que juega. Gana quien retira la última ficha.

a) Modele con un grafo el desarrollo del juego.
Para el diseño del modelo, se llevo a cabo una aproximación a través de una máquina de estados, los estados, modelan la interacción del juego a través de estados

que representan a cada jugador, y estados que representan si el jugador gana o no, finalmente, un estado final que revela el ganador. El desarrollo del juago con ésta máquina de estados, está dado por el siguiente algoritmo:

```
NIM(P_0, Q_0, R_0)
 1  a_1, a_2, a_3, b_1, b_2, b_3 ← 0, 0, 0, 0, 0, 0
 2  winner ← A
 3  A:
 4      GETS(a_1, a_2, a_3)
 5      if P_0 + Q_0 + R_0 = 0 then
 6          winner ← A
 7          goto winner
 8      else goto B
 9  B:
10      GETS(b_1, b_2, b_3)
11      if P_0 + Q_0 + R_0 = 0 then
12          winner ← B
13          goto winner
14      else goto A
15  WINNER:
16      PRINT winner
```

El procedimiento, GETS$(x,\ y,\ z)$, cambia el estado de las variables $P_0,\ Q_0,\ R_0$ por $P_0 - x,\ Q_0 - y,\ R_0 - y$. En éste modelo, el perdedor es quien no sea impreso al finalizar el juego.

El diseño del modelo anterior, da lugar al siguiente grafo:

$$G(V,\ E)$$

$$V = \{\mathbb{Z}_{P_0+1} \times \mathbb{Z}_{Q_0+1} \times \mathbb{Z}_{R_0+1}\} \times A \times B$$

$$E = \{((P_0,\ Q_0,\ R_0),\ A)(A,\ (\mathbb{Z}_{P_0-a_1}, \mathbb{Z}_{Q_0-a_2}, \mathbb{Z}_{R_0-a_3})), ((\mathbb{Z}_{P_0-a_1}, \mathbb{Z}_{Q_0-a_2}, \mathbb{Z}_{R_0-a_3}),\ B), ...$$
$$(B,\ (\mathbb{Z}_{P_0-b_1}, \mathbb{Z}_{Q_0-b_2}, \mathbb{Z}_{R_0-b_3})), ((\mathbb{Z}_{P_0-b_1}, \mathbb{Z}_{Q_0-b_2}, \mathbb{Z}_{R_0-b_3}),\ A)\}$$

En el anterior grafo, los nodos modelan a cada jugador y la jugada del jugador, a partir de una tupla con el estado actual del programa, un nodo sin entradas, es decir sin flechas hacia el, es $(P_0,\ Q_0,\ R_0)$, de él, se va a A, quien es el primer jugador, de allí, se va al nodo que representa el esta actual de cada montón dependiendo la jugada de A.

b) Explique en su modelo cómo se reconoce que un jugador pierde.

Si partiendo del nodo de cada jugador, se llega al nodo $(0,\ 0,\ 0)$, el perdedor será el nodo contrario del cual se partió.