Algoritmos y Estructuras de Datos I – 2º GII
Examen. 29 de mayo de 2025

1. Disponemos de una especificación en Maude para árboles binarios (**AB**) de números naturales (**N**) con las siguientes operaciones:

Sintaxis	**Descripción**
`arbolVacio : -> AB .`	Constante que representa un árbol binario vacío
`arbol : N AB AB -> AB .`	Devuelve el árbol resultado de colocar el natural representado por el primer parámetro como raíz de un subárbol que tiene como hijo izquierdo el segundo parámetro y como hijo derecho el tercer parámetro
`cero : -> N .`	Constante que representa el natural cero
`sucesor : N -> N .`	Devuelve el natural siguiente al parámetro
`suma : N N -> N .`	Devuelve la suma de los dos parámetros naturales

Estudia y determina el significado de la operación `misterio` cuya especificación se describe a continuación. Esta operación se apoya en otra operación, llamada `auxiliar`, cuya especificación también se facilita:

Sintaxis:

```
op misterio : AB N -> N .
op auxiliar : AB N N -> N .
```

Semántica:

```
var v, n, m : N .
var a1, a2 : AB .

*** Función misterio

eq misterio (arbolVacio, n) = cero .
eq misterio (arbol (v, a1, a2), n) = auxiliar (arbol (v, a1, a2), n, cero) .

*** Función auxiliar

eq auxiliar (arbolVacio, n, m) = cero .
eq auxiliar (arbol (v, a1, a2), n, n) = v .
eq auxiliar (arbol (v, a1, a2), n, m) =
      suma (auxiliar(a1, n, sucesor (m)), auxiliar(a2, n, sucesor (m))) .
```

Hay que describir de forma razonada qué es lo que hacen estas operaciones y mostrar algún ejemplo de su funcionamiento.

Respuesta 1.

En breves palabras, la función `misterio(a, n)` calcula la suma de los nodos del árbol `a` que tienen profundidad `n`, siendo la raíz del árbol profundidad 0, los hijos profundidad 1, los nietos profundidad 2, y así sucesivamente. Para conseguirlo, se utiliza una función `auxiliar(a, n, m)` que lleva un contador, `m`, de la profundidad actual, empezando desde 0. La función `auxiliar` va bajando a los hijos y aumentando el contador de uno en uno; al llegar al caso `n = m`, se devuelve el valor de la raíz. En realidad, la operación `misterio` se podría especificar solo con tres axiomas, sin necesidad de una operación `auxiliar`, de la siguiente forma (como ya sabemos lo que hace, la llamaremos `Contar`):

```
eq Contar(arbolVacio, n) = cero .
eq Contar(arbol (v, a1, a2), cero) = v .
eq Contar(arbol (v, a1, a2), sucesor(n)) = suma(Contar(a1, n), Contar(a2, n)) .
```

2. Definimos un nuevo tipo de tablas de dispersión, que llamaremos **tablas de dispersión trifásicas**. En este tipo de tablas, cada cubeta tiene siempre 3 huecos. De esta forma, si al insertar un nuevo elemento su cubeta está vacía, se puede insertar; si su cubeta ya tiene 1 elemento, también se puede insertar el nuevo; si tiene 2 elementos, también cabe en la tercera posición; y si ya tiene 3 elementos, no cabe y suponemos que se producirá un mensaje de error. Se pide lo siguiente:

   a) Define cómo sería el tipo de datos para almacenar tablas de dispersión trifásicas, suponiendo que las claves son de tipo *C*, los valores de tipo *V* y la tabla tiene *B* cubetas. Usando ese tipo de datos, programar las operaciones para insertar un par (*C*, *V*) en la tabla y para consultar una clave *C* en la tabla. Se supone que existe una función de dispersión: h(*C*) que devuelve un natural de 32 bits.
   b) Analiza cuánto sería el uso de memoria de una tabla de dispersión trifásica con *n* elementos y *B* cubetas. Compara el resultado con la dispersión cerrada y la dispersión abierta, usando algún ejemplo para mostrarlo.
   c) Discute sobre los tiempos de ejecución de los tres métodos: la dispersión cerrada, la abierta y la trifásica. No hace falta dar una fórmula explícita, solo describir cómo es su tiempo en relación con las otras.

Respuesta 2.

a) Las tablas de dispersión trifásicas se pueden ver como un término medio entre la dispersión abierta y la cerrada, en el sentido de que permiten almacenar varios elementos por cubeta (como las abiertas) pero solo un número limitado (como la cerrada). Podríamos definir cada cubeta o bien como una lista limitada a tamaño 3, o bien como un array de 3 posiciones. No obstante, el enunciado parece decantarse por la segunda opción, puesto que dice que "cada cubeta tiene siempre 3 huecos".

En primer lugar, debemos empezar definiendo los tipos de datos usados para las tablas de dispersión trifásicas. Como es un diccionario, vamos a almacenar pares de tipo (clave, valor):

```
tipo par = registro
               clave: C
               valor: V
           finregistro
```

Por lo tanto, una tabla de dispersión trifásica de tamaño *B* sería un array de *B* posiciones, cada una de las cuales es un array de 3 pares:

```
tipo TablaTrifásica = array [0...B-1, 0...2] de par
```

Vamos a ver ahora cómo serían las operaciones de inserción y de consulta. En la inserción, debemos tener en cuenta el caso de que ya exista la clave, en cuyo caso modificamos el valor asociado. Recordemos que la función *h* devuelve un valor grande, por lo que es necesario aplicar módulo *B*. Supondremos que el tipo *C* de las claves tiene definido un valor NULL.

```
operación Insertar (var T: TablaTrifásica; clave: C; valor: V)
    pos:= h(clave) mod B
    i:= 0
    mientras i<3 AND T[pos, i].clave!=NULL AND T[pos, i].clave!=clave hacer
        i:= i+1
    finmientras
    si i<3 entonces      // Esto incluye los casos de que se encuentre la clave o que se encuentre una posición vacía
        T[pos, i].clave:= clave
        T[pos, i].valor:= valor
    sino
        ERROR("Se ha llenado la cubeta ", pos)
    finsi
```

La operación de consulta seguirá la misma secuencia de búsqueda que la inserción. Es un error muy grave pensar que la consulta debe recorrer *todas* las cubetas, puesto que la clave solo puede estar en su posición h(clave). Por otro lado, en cada cubeta los elementos no están ordenados, aunque podríamos cambiar la implementación para que lo estuvieran. Supondremos que el tipo V tiene definido un valor NULL.

```
operación Consultar (T: TablaTrifásica; clave: C) : V
    pos:= h(clave) mod B
    i:= 0
    mientras i<3 AND T[pos, i].clave!=NULL AND T[pos, i].clave!=clave hacer
        i:= i+1
    finmientras
    si i<3 AND T[pos, i].clave==clave entonces
        devolver T[pos, i].valor
    sino
        devolver NULL
    finsi
```

b) Supongamos que cada puntero ocupa k_1 bytes, cada par (clave, valor) ocupa k_2 bytes, y la tabla tiene B cubetas y n elementos. Entonces, podemos deducir de forma sencilla el uso de memoria de la tabla de dispersión trifásica, además de la abierta y la cerrada.

Tabla de dispersión	**Uso de memoria**
Trifásica	$k_2 \cdot B \cdot 3$ bytes
Cerrada	$k_2 \cdot B$ bytes
Abierta	$k_1 \cdot B + (k_1 + k_2) \cdot n$ bytes

Podría parecer que la tabla de dispersión trifásica necesita mucha más memoria que la cerrada, puesto que ocupa 3 veces más memoria para el mismo valor de B. Sin embargo, la trifásica puede almacenar 3 veces más elementos. Por otro lado, la trifásica podría necesitar más o menos memoria que la abierta, según los valores de B y de n. Pero sabemos que la abierta solo almacena los pares que realmente existan, mientras que la trifásica tiene reservadas $3B$ posiciones, que pueden estar ocupadas o no. Así que, en general, es más probable que la trifásica requiera más memoria que la abierta.

c) Sobre los tiempos de ejecución, sabemos que la dispersión abierta tiene un tiempo promedio para la inserción y la consulta de $O(1+n/B)$; y la cerrada tiene un tiempo de $O(1/(1-n/B))$, con la limitación de que no puede almacenar más de B elementos.

En cuanto a la trifásica, si suponemos que los elementos se reparten uniformemente y las cubetas no sobrepasan tamaño 3, entonces el tiempo sería el mismo que la dispersión abierta, $O(1+n/B)$. Pero esta fórmula es engañosa, porque cuando n toma valor $3B$, debería tender a infinito (de forma semejante a lo que pasa con la cerrada), indicando que no caben más elementos.

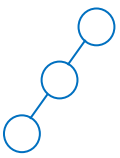
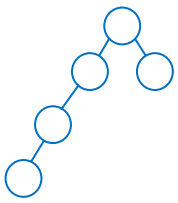
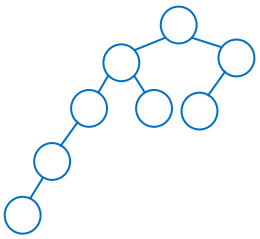
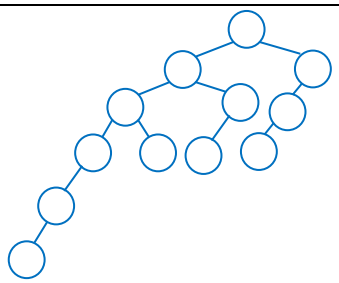
Es más, basta con que una sola cubeta tenga ya 3 elementos para que no quepan más y se produzca un error. Podríamos tener una tabla con un millón de cubetas vacías, pero con 3 colisiones en una cubeta, y se produciría un error. Esto es difícil expresarlo como una fórmula, ya que debería ser parte de un orden condicionado del estilo $O(1+n/B \mid$ condicionado a que $n<3B$ y los elementos se reparten uniformemente, porque en otro caso tiene a infinito). En la práctica, este problema hace que las tablas de dispersión trifásicas no tengan utilidad práctica real (a menos que cambiásemos la estrategia de lo que ocurre cuando se producen 3 colisiones en una cubeta).

Algoritmos y Estructuras de Datos I – 2º GII
Examen. 29 de mayo de 2025

3. Queremos introducir una variante de los árboles AVL, que llamaremos AVL2, donde se permite una diferencia de altura 2, en lugar de 1. Es decir, para cada nodo del árbol, la diferencia de alturas entre el subárbol izquierdo y el derecho puede ser como máximo 2. Se pide lo siguiente:
   a) Muestra cómo sería el peor caso del AVL2 de alturas h = 0, 1, 2, 3, 4 y 5. A partir de eso, indica cómo sería el caso general para el número de nodos en el peor caso de un árbol AVL2 de altura h, es decir, N(h).
   b) En los AVL, comprobamos en clase mediante el cálculo de alturas que si se produce un desbalanceo de tipo Izquierda-Izquierda en la raíz, A, se puede solucionar mediante una RSI(A). Comprueba mediante el cálculo de alturas si el desbalanceo Izquierda-Izquierda en un AVL2 también se soluciona mediante una RSI.
   c) Muestra cómo sería la inserción de los siguientes elementos en un AVL y en un AVL2, indicando los casos de desbalanceo y las rotaciones aplicadas. Sobre un árbol vacío, se insertan los elementos: 62, 32, 14, 87, 75, 68, 70. Compara los resultados de ambos tipos de árboles.

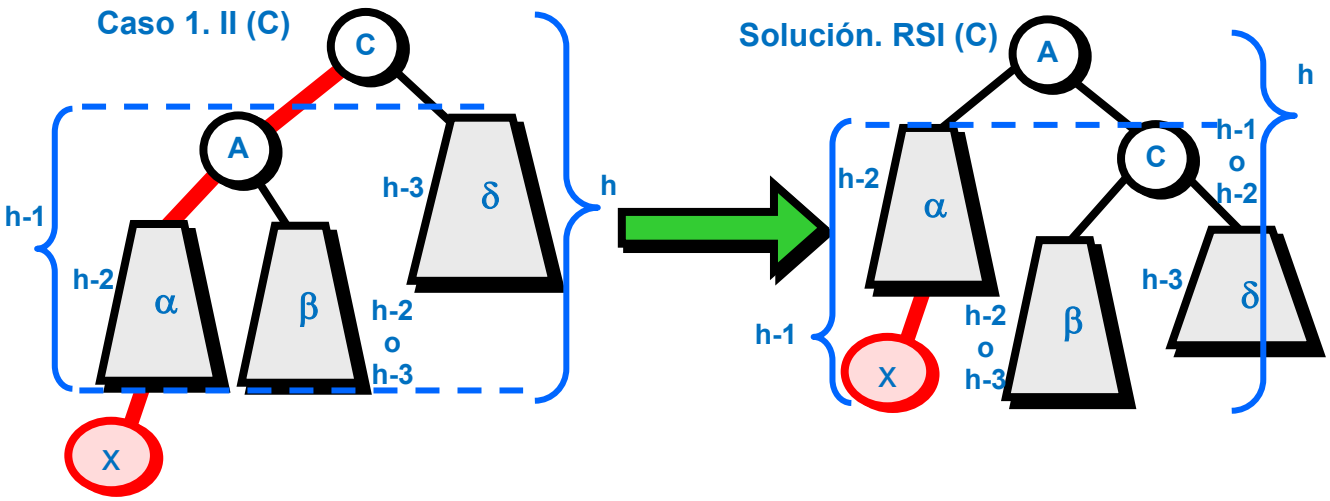
Respuesta 3.

a) Igual que vimos en clase con los árboles AVL, el peor caso de los árboles AVL2 sería el caso de un árbol que tenga el máximo desbalanceo permitido, para un árbol de cierta altura h. Esto es equivalente al árbol con el menor número de nodos posible para esa altura. Los peores casos de AVL2 serían del siguiente tipo. Se supone que un solo nodo es un árbol con altura 0.

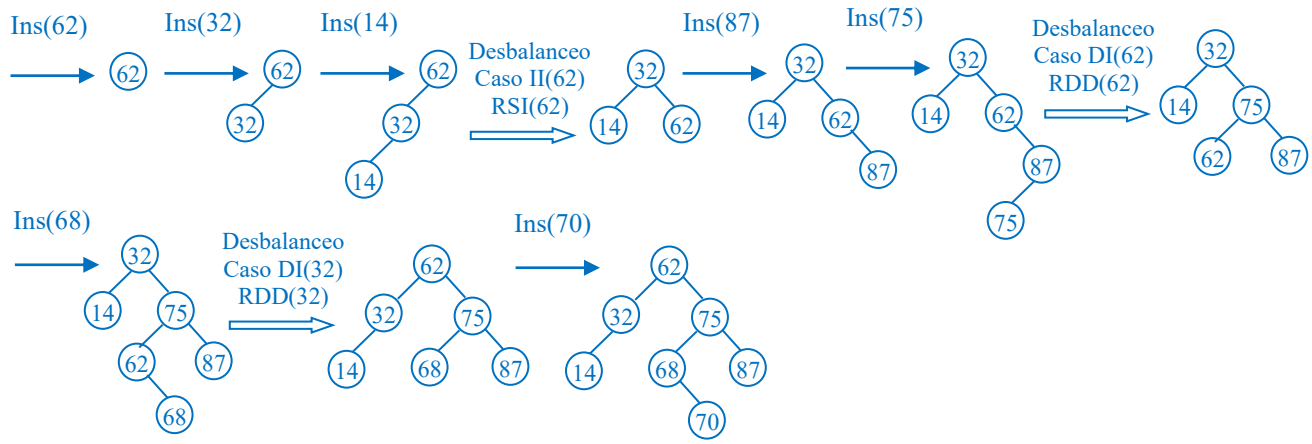
$h = 0$	$h = 1$	$h = 2$	$h = 3$	$h = 4$	$h = 5$
					

Como se puede ver, el caso general para altura h es un árbol cuyo subárbol izquierdo es el peor caso de altura h-1 y el derecho es el peor caso de altura h-3, o viceversa. Por ejemplo, el peor caso de h = 5 tiene a su izquierda el peor caso de h = 4, y a la derecha el peor caso de h = 2. Por lo tanto, el número total de nodos del árbol AVL2 en el peor caso para una altura h es: N(h) = N(h-1) + N(h-3) + 1.

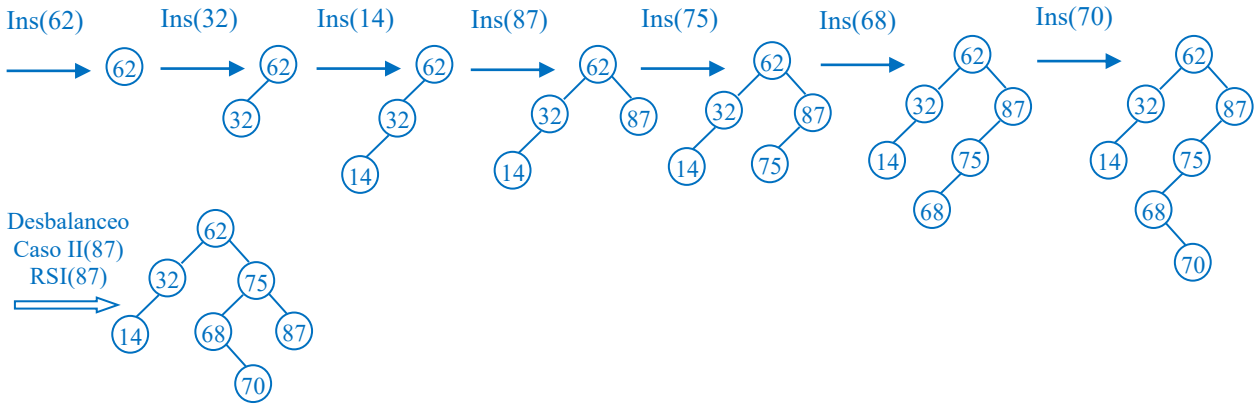
b) El **cálculo de alturas** para el caso de desbalanceo II es igual que como vimos en clase con los AVL. La idea es la siguiente. Tenemos un árbol AVL2 que está balanceado (abajo a la izquierda), aunque su rama izquierda tiene altura +2 que la derecha; entonces insertamos un nuevo elemento X, que hace que el árbol se desbalancee con una diferencia +3. Para solucionarlo, aplicamos una RSI(C) (el árbol de abajo a la derecha), siendo C la raíz. En el cálculo de alturas, primero calculamos las alturas que deben tener los subárboles genéricos α, β y δ. Y luego *aplicamos* esas alturas al árbol rotado, para comprobar que está balanceado. Lo único que cambia respecto al AVL es que δ debe tener altura h-3, y β puede ser h-2 o h-3.



c) Con AVL, la secuencia de inserciones sería la siguiente:



Con AVL2, la secuencia de inserciones sería la siguiente:



En este ejemplo concreto, el árbol AVL ha requerido 3 rotaciones, mientras que el AVL2 solo ha necesitado 1 rotación, obteniendo el mismo árbol resultante. En general, es de esperar que el AVL2 requiera menos rotaciones que el AVL, al ser menos restrictivo. No obstante, debemos recordar que las rotaciones tienen un tiempo de ejecución constante, por lo que el hecho de necesitar más rotaciones no necesariamente implica que sea más ineficiente. Además, aunque en este caso ambas estructuras hayan producido el mismo árbol, en general es de esperar que los árboles AVL2 estén algo más desbalanceados que los AVL.

4. En Ganímedes se ha descubierto un laberinto con n celdas. Lo tenemos modelado como un grafo no dirigido representado como una matriz simétrica de $n \times n$ booleanos $A[i,j]$ que representa la adyacencia entre pares de celdas. En la celda g se encuentra perdido Gervasio, el intrépido astronauta de Torre Pacheco, que puede moverse a una velocidad de 1 movimiento por día. En otra celda t hay un foco de contagio (la temible Tripichurla, cuya infección te convierte en un cani trianero) que se extiende en paralelo como una plaga en todas las direcciones posibles a una velocidad de 2 días por movimiento (es decir, si empezamos el día 0, se mueve solo los días pares). Gervasio tiene que sobrevivir D días dentro del laberinto sin que le pille la plaga (pues ese día será rescatado del laberinto, en cualquier celda que se encuentre). En cada día, Gervasio puede decidir moverse o no a cualquier celda adyacente. La plaga, como hemos visto, se extenderá siempre a todas las celdas adyacentes (a partir de su posición inicial) pero solo cada dos días.

   Nuestra misión es encontrar los movimientos que Gervasio debe hacer (o no) para garantizar que al cabo de D días, cuando llegue la misión de rescate, se encuentre en una casilla a la que aún no haya llegado la plaga, si eso es posible. Razona y escribe un algoritmo para resolver el problema de forma eficiente.

Respuesta 4.

Podemos descomponer este problema en dos partes: (a) calcular los movimientos que va a hacer Tripichurla; y (b) decidir los movimientos de Gervasio para no encontrarse con Tripichurla. La solución de (a) sigue claramente el recorrido de una búsqueda primero en anchura, tardando 2 días por arista. Sus resultados condicionan las posibles soluciones de (b).

Por lo tanto, primero debemos resolver el problema (a) con una BPA, calculando el día en que se infecta cada nodo del grafo, $T[i]$ = día en el que se infecta el nodo i del grafo, para i= 1…n. A continuación, debemos encontrar un camino en el grafo desde el nodo g hasta algún nodo s cuyo $T[s]$ sea mayor que D, y sin pasar por nodos infectados. Vamos a ver primero la BPA para resolver el problema (a). La entrada en la matriz de adyacencia, A, y el nodo inicial de Tripichurla, t; y la salida es el array T. Por simplicidad, no vamos a usar directamente la matriz de adyacencia, sino que usaremos el iterador genérico: `para cada nodo w adyacente a v`.

```
operación Trip (A: array [1…n, 1…n] de booleano; t: entero): array [1…n] de entero
     T : array [1…n] de entero = ∞     // Inicialización para todos los nodos
     C : cola de entero
     T[t]:= 0
     C.insertar(t)
     mientras not C.esVacía() hacer
          v:= C.cabeza()
          C.sacarCabeza()
          para cada nodo w adyacente a v hacer
               si T[w] == ∞ entonces   // Equivalente a que w no está visitado
                    T[w]:= T[v] + 2    // Para Tripichurla todas las aristas son de coste 2
                    C.insertar(w)
               finsi
          finpara
     finmientras
     devolver T
```

Si hay algún nodo al cual nunca llegue Tripichurla, su valor de T será ∞. Por otro lado, debemos calcular el camino de Gervasio desde g hasta algún nodo en que no sea alcanzado. Para esto podríamos usar, por ejemplo, el algoritmo de Dijkstra adaptado a que no pase por nodos infectados. No obstante, como todas las aristas tienen para Gervasio coste 1, podemos usar una BPA desde el nodo g, que también encontrará los caminos mínimos. Una vez que lleguemos a un nodo v con $T[v]$ mayor que D, se detendrá en ese nodo como su solución. La entrada del algoritmo de Gervasio es la matriz de adyacencia, A, el array de tiempos de paso de Tripichurla, T (obtenido en la anterior operación), el nodo inicial de Gervasio, g, y el tiempo límite, D. La salida es el nodo donde debe ir Gervasio, y además tenemos el array *Caminos* (pasado como parámetro por referencia) que indica los caminos óptimos de Gervasio desde g hasta todos los demás. En la operación, $G[v]$ indicará el tiempo mínimo en el cual Gervasio puede llegar a cada nodo v.

```
operación Gerv (A: array [1…n, 1…n] de booleano; T: array [1…n] de entero; g: entero;
                D: entero; var Caminos: array [1…n] de entero): entero
     G : array [1…n] de entero = ∞       // Inicialización del camino mínimo desde g hasta todos los nodos
     C : cola de entero
     G[g]:= 0
     C.insertar(g)
     mientras not C.esVacía() hacer
          v:= C.cabeza()
          C.sacarCabeza()
          si T[v] > D entonces           // Ya hemos llegado a un nodo solución
               devolver v
          finsi
          para cada nodo w adyacente a v hacer
               si G[w] == ∞ AND T[w]>G[v]+1 entonces // w no está visitado y no está infectado todavía
                    G[w]:= G[v] + 1     // Para Gervasio, todas las aristas son de coste 1
                    Camino[w]:= v       // El camino óptimo para w pasa por v
                    C.insertar(w)
               finsi
          finpara
     finmientras
     devolver -1          // Si llega aquí, no hay ningún camino para Gervasio
```

Finalmente, podemos combinar estas dos operaciones para obtener el algoritmo completo. Este algoritmo recibe como parámetros los datos del problema: la matriz de adyacencia, A, los nodos iniciales de Gervasio, g, y de Tripichurla, t, y el tiempo límite, D. Hemos supuesto que el algoritmo no devuelve ninguna salida, sino que la escribe por pantalla.

```
operación Ganímedes (A: array [1…n, 1…n] de booleano; g, t, D: entero)
     T : array [1…n] de entero
     Caminos : array [1…n] de entero
     T:= Trip (A, t)
     s:= Gerv (A, T, g, D, Caminos)
     si s == -1 entonces
         Escribir ("No hay solución para Gervasio")
     sino
         Escribir ("La solución es ", g → ... → Caminos[Caminos[s]] → Caminos[s] → s)
     finsi
```

Hemos expresado de forma simplificada el segundo `Escribir`, que en realidad sería un bucle `for`, empezando en s, moviéndose a $Caminos[s]$, $Caminos[Caminos[s]]$, y así hasta llegar a g. Después de llegar a s, ya no necesita moverse. El tiempo de ejecución de este algoritmo es el mismo que el de la BPA, es decir, $O(n^2)$ con matrices de adyacencia, o bien $O(n+a)$ si usáramos listas de adyacencia, puesto que simplemente se hacen dos llamadas a la BPA.

Nota: Por completitud, vamos a ver también de forma muy breve cómo se resolvería la segunda parte del problema si quisiéramos hacerla usando el algoritmo de Dijkstra en lugar de la BPA. Lo único que cambiaría respecto al algoritmo clásico sería evitar las aristas que conduzcan a un nodo que esté infectado. Esto es lo que arriba hemos hecho con la condición `T[w]>G[v]+1`. En el algoritmo de Dijkstra visto en clase, el paso de actualización de cada nodo v sería:

```
para cada nodo w adyacente a v hacer
     si (NOT S[w]) AND (G[v]+1<G[w]) AND (T[w]>G[v]+1) entonces
```

El uso del algoritmo de Dijkstra en este problema no resulta útil porque no mejora el tiempo de ejecución de la BPA. No obstante, si tuviéramos un problema donde las aristas tuvieran distintos costes, entonces sí que sería necesario aplicar el algoritmo de Dijkstra. El paso anterior cambiaría de la siguiente forma, siendo A la matriz de coste de las aristas:

```
para cada nodo w adyacente a v hacer
     si (NOT S[w]) AND (G[v]+A[v,w]<G[w]) AND (T[w]>G[v]+A[v,w]) entonces
```