1. Tenemos una especificación formal axiomática para listas (**L**) de números naturales (**N**) con las siguientes operaciones, usando la sintaxis de Maude:

Sintaxis	**Descripción**
`listaVacia : -> L .`	Constante que representa la lista vacía.
`insertar : N L -> L .`	Devuelve la lista resultante de añadir al principio del segundo parámetro el natural representado por el primer parámetro.
`esMenorIgual : N N -> Bool .`	Comparación de naturales. Devuelve *true* si el primer parámetro es menor o igual que el segundo.

Estudiar y determinar el significado de la operación `misterio` cuya especificación se describe a continuación. Esta operación se apoya en otras dos operaciones, `auxiliar1` y `auxiliar2`, cuya especificación también se facilita.

Sintaxis:

```
op misterio : L -> L .
op auxiliar1 : L -> Bool .
op auxiliar2 : L -> L .
```

Axiomas:

```
var n, m : N .
var l : L .
eq misterio (l) = if auxiliar1 (l) then
                        l
                  else
                        misterio ( auxiliar2 (l))
                  fi .

eq auxiliar1 (listaVacia) = true .
eq auxiliar1 (insertar (n, listaVacia)) = true .
eq auxiliar1 (insertar (n, insertar (m, l))) =
       esMenorIgual (n, m) and auxiliar1 (insertar (m, l)) .

eq auxiliar2 (listaVacia) = listaVacia .
eq auxiliar2 (insertar (n, listaVacia)) = insertar (n, listaVacia) .
eq auxiliar2 (insertar (n, insertar (m, l))) =
                if esMenorIgual (n, m) then
                      insertar (n, auxiliar2 (insertar (m, l)))
                else
                      insertar (m, auxiliar2 (insertar (n, l)))
                fi .
```

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

Respuesta 1.

De forma resumida, podemos decir que: **misterio(l)** ordena los elementos de la lista **l** de menor a mayor; **auxiliar1(l)** devuelve true si los elementos de **l** están ordenados de menor a mayor; y **auxiliar2(l)** recorre todos los elementos de la lista **l**, permutando los pares sucesivos si el primero es mayor que el segundo (sería equivalente a un paso del algoritmo de ordenación de la burbuja). Veámoslo con más detalle:

- **auxiliar1(l)**: siempre devuelve true si la lista l tiene 0 o 1 elementos. Si tiene dos elementos (a, b), devuelve true si $a \leq b$. Si tiene tres elementos (a, b, c), comprueba si $a \leq b$ y luego recursivamente también comprueba si $b \leq c$. Si tiene cuatro elementos (a, b, c, d), comprueba si $a \leq b$, luego si $b \leq$

c, y luego si c ≤ d. Por lo tanto, al final solo devolverá true cuando todos los elementos de la lista estén ordenados de menor a mayor.

- **auxiliar2(l)**: devuelve la misma lista l, si solo tiene 0 o 1 elementos. Si tiene dos elementos (a, b), devuelve (a, b) si a ≤ b, y devuelve (b, a) si a > b; es decir, permuta los elementos si no están en orden de menor a mayor. Si tiene tres elementos, (a, b, c) primero comprueba el par (a, b), realizando una permutación si no están en orden; y después recursivamente compara el mayor de a y b, con c, también haciendo una permutación si no están en orden. Por lo tanto, en general realizará una pasada completa por todos los pares consecutivos de elementos de la lista (i, j), realizando una permutación si no se cumple que i ≤ j. Este resultado no garantiza que la lista esté ordenada. Pero sí que garantiza que el elemento mayor se coloca al final de la lista. Esto corresponde a una pasada del método de ordenación de la burbuja. Si se realizan dos pasadas, sabemos que los dos mayores estarán al final de la lista y así sucesivamente.
- **misterio(l)**: en primer lugar, comprueba si la lista está ordenada de menor a mayor; en ese caso devuelve la lista l sin modificar. En caso contrario, realiza una pasada de ordenación (auxiliar2) y además vuelve a llamarse recursivamente (misterio(auxiliar2(l))). En consecuencia, después de la pasada de ordenación volverá a comprobar si la lista resultante está ordenada, y si no lo está volverá a aplicar otra nueva pasada de ordenación. Así que se volverán a aplicar sucesivas pasadas de ordenación mientras la lista no esté ordenada. Como sabemos que en cada pasada se colocará un nuevo elemento de los mayores al final de la lista, se puede garantizar que como máximo en n-1 pasadas la lista estará ordenada, siendo n el número total de elementos. Por lo tanto, **misterio** realiza el algoritmo de ordenación por el método de la burbuja, ordenando los números de menor a mayor.

2. Tenemos que representar un tipo de datos *polinomio escaso*, que serán polinomios de grado potencialmente muy alto (hasta x^{10000}), de la forma:

$$P(x) = \sum_{i=0}^{10000} a_i \cdot x^i$$

El término "escaso" significa que la mayor parte de los términos valen 0. Por ejemplo, un polinomio escaso podría ser: $p1(x) = 4{,}5 \cdot x^{37} + 3{,}71 \cdot x^{198} + 0{,}37 \cdot x^{2500} - 1{,}62 \cdot x^{3030}$. O, por ejemplo: $p2(x) = 3{,}25 \cdot x^{92} - 9 \cdot x^{5241} + 8{,}08 \cdot x^{632}$. En concreto, no tendrán más de 50 coeficientes distintos de cero. Las operaciones que se espera realizar con estos polinomios son:

Sintaxis	**Descripción**
op polinomioNulo () : Polinomio	Operación constante que devuelve el polinomio p(x) = 0.
op añadir_coef (p: Polinomio, i: entero, a: real) : Polinomio	Esta operación devuelve el resultado de añadir al polinomio p un nuevo término $a \cdot x^i$.
op eval (p: Polinomio, x: real) : real	Devuelve el resultado de evaluar el polinomio p en x, es decir, devuelve el valor de p(x).

Describir de forma completa, detallada y justificada cómo se podrían utilizar las tablas de dispersión para representar este tipo de datos. Esta descripción debe contener:

- La definición del tipo de datos "polinomio escaso" utilizando la representación propuesta con tablas de dispersión.
- Las decisiones de diseño de las tablas de dispersión usadas: tipo de tablas, función de dispersión, tamaño de tabla, estrategia de reestructuración, etc.
- Mostrar gráficamente algún ejemplo de estas tablas (p.ej. usando los dos ejemplos de arriba).
- La implementación en pseudocódigo de las tres operaciones, explicando su funcionamiento.
- El estudio de la eficiencia computacional, en tiempo de ejecución, de las operaciones.

Respuesta 2.

En pocas palabras, cada **Polinomio** es un diccionario donde las claves son enteros entre 0 y 10000 (el grado del término) y el valor asociado son números reales (el coeficiente de cada término). Por ejemplo, el polinomio p1 almacenaría los pares (clave→ valor): {(37 → 4,5), (198 → 3,71), (2500 → 0,37), (3030 → -1,62)}. Las tablas de dispersión son una estructura de datos ideal para esta aplicación. La operación **polinomioNulo** simplemente debería devolver una tabla de dispersión vacía. La operación **añadir_coef (p, i, a)** es una inserción en la tabla **p** de la clave **i** y el valor **a**; con la particularidad de que si ya está la clave **i**, debe sumar a su valor existente la **a**. La operación **eval(p, x)** debe recorrer toda la tabla, y para cada par (clave **i**, valor **a**) que encuentre, debe sumar al resultado $\mathbf{a \cdot x^i}$. Vamos a ver las soluciones con dispersión abierta y cerrada, aunque en el ejercicio solo se requería una de ellas.

- **Definición del tipo de datos**

Lo primero es definir los tipos de datos que se usarán. En primer lugar, definimos el tipo **Término** (llamado también monomio), que es simplemente una pareja formada por un coeficiente y un grado:

```
tipo Término = registro
     grado : entero (0...10000)
     coeficiente : real
finregistro
```

En principio, podemos tener una buena solución tanto con dispersión abierta como con cerrada. Con dispersión abierta, la tabla sería un array de listas de Término. Suponiendo que está definido un tipo Lista<T> al estilo del tipo list<T> de C++, la definición del polinomio con dispersión abierta sería:

```
tipo PolinomioDA = array [0...B-1] de Lista<Término>
```

Con dispersión cerrada, la tabla sería directamente un array de Términos:

```
tipo PolinomioDC = array [0...B-1] de Término
```

- **Decisiones de diseño: tipo de tablas**

Aunque podrían servir tanto las tablas de dispersión abiertas como las cerradas, la eficiencia en tiempo de ejecución y uso de memoria es distinta. En la abierta, el tiempo de la consulta e inserción es un O(1+n/B), siendo n el número de términos y B el tamaño de la tabla; con cerrada es O(1/(1-n/B)). Si la B es la misma, la dispersión abierta será más rápida. Por ello, la cerrada necesita un B más grande para conseguir el mismo tiempo que la abierta, pero entonces necesitará más memoria. Por ejemplo, con B = 100 y n = 50, las secuencias de búsqueda de la dispersión cerrada serían de longitud promedio 1/(1-50/100) = 2; para esa misma longitud, en la abierta necesitaríamos solo B = 50 cubetas, puesto que 1+50/50 = 2. Además, cuando la tabla está vacía o tiene pocos elementos, la dispersión cerrada necesita más memoria que la abierta, puesto que la abierta almacenaría B punteros y la cerrada B enteros y reales. A medida que la tabla se llene, la abierta puede llegar a necesitar más memoria. Considerando todos los factores, en términos globales la dispersión abierta podría ser más adecuada en la relación tiempo/memoria.

- **Decisiones de diseño: tamaño de la tabla**

En cuanto al tamaño de la tabla, sabemos que el número de términos almacenados, n, será como máximo 50. Por lo tanto, con dispersión abierta lo más adecuado es que B sea 50. No obstante, podría ser interesante que B sea un número primo, por ejemplo 47 o 53, para mejorar la "aleatoriedad" de la función de dispersión. Con dispersión cerrada, sabemos que a medida que la tabla se llena, el número de colisiones crece rápidamente, por lo que interesa tener mucho más de 50 cubetas (y nunca menos de 50). Por ejemplo, para que la longitud promedio de las secuencias de búsqueda no sobrepase de 3 ($1/(1-n/B) < 3$), necesitamos que B = 75; también aquí podría ser interesante que B sea primo, por ejemplo 79, tanto para la función de dispersión como para la de redispersión.

- **Decisiones de diseño: función de dispersión**

En cuanto a la función de dispersión, debemos recordar que se aplica **solo** sobre la clave, por lo que tendría la forma h(grado). Puesto que no tenemos indicios sobre la forma que tendrán los términos, nos podría servir una función simple como: h(grado) = grado módulo B. También podríamos aplicar otras técnicas más avanzadas, como el método de la multiplicación o el centro del cuadrado, por ejemplo: $h'(grado) = \lfloor grado \cdot \pi \rfloor$ módulo B; o bien: $h''(grado) = \lfloor grado^2/1414 \rfloor$ módulo B. Los cálculos realizados por estas funciones h' y h" aumentan la aleatoriedad del valor obtenido, pero no podemos asegurar que provoquen menos sinónimos que la simple función h.

- **Decisiones de diseño: función de redispersión**

Con dispersión cerrada, deberíamos definir también una función de redispersión. Podría ser, por ejemplo, una redispersión doble: $h_i(grado) = (h(grado)+i \cdot C(grado))$ módulo B. Si B = 79 en la dispersión cerrada, al ser un número primo, la función C(grado) puede devolver valores entre 1 y B-1, pues todos ellos son coprimos con 79. Por ejemplo: $C(grado) = (\lfloor 7 \cdot grado/\pi \rfloor$ módulo (B-1))+1. En cualquier caso, la función de redispersión solo puede depender del grado, no del coeficiente.

- **Decisiones de diseño: reestructuración de la tabla**

En cuanto a la estrategia de reestructuración de la tabla, como tenemos garantizado que n nunca será mayor que 50, no es necesario usar reestructuración. No obstante, sí que podría ser interesante definir una estrategia donde el B inicial sea pequeño (por ejemplo B = 5), y duplicar el número de cubetas cuando la tabla se llene mucho. Esto mejorará el uso de memoria, a costa de necesitar más tiempo de ejecución en cada reestructuración. Por lo tanto, es adecuada si se espera que el número de términos sea muy pequeño.

- **Implementación de las operaciones: polinomioNulo**

La implementación de **polinomioNulo** es trivial, porque simplemente debe crear una tabla nueva (vacía), ya sea en la abierta o en la cerrada. En el caso de la dispersión abierta, todas las listas se inicializarían a vacía. En la dispersión cerrada, todos los grados se deberían inicializar a -1. Existe un buen motivo para no inicializarlos a 0, puesto que el grado 0 es un valor válido (sería la constante de un polinomio, por ejemplo, p(x) = 7 + …).

- **Implementación de las operaciones: añadir_coef**

La implementación de **añadir_coef** podría ser de la siguiente forma, suponiendo que la hacemos con dispersión abierta:

```
op añadir_coef (p: PolinomioDA, i: entero, a: real) : PolinomioDA
    pos:= h(i)
    para cada términoAct en p[pos] hacer      // Iterador de listas
        si términoAct.grado == i entonces
            términoAct.coeficiente:= términoAct.coeficiente + a
            devolver p
        finsi
    finpara
    p[pos].insertar_final(Término(i, a))
    devolver p
```

Con dispersión cerrada, suponiendo que no se usa reestructuración de la tabla, sino que el B inicial es suficientemente grande para que la tabla nunca se llene, la implementación sería:

```
op añadir_coef (p: PolinomioDC, i: entero, a: real) : PolinomioDA
    pos:= h(i)
    nredisp:= 1
    mientras p[pos].grado ≠ i Y p[pos].grado ≠ -1 hacer
        pos:= h_nredisp(i)
        nredisp:= nredisp + 1
    finmientras
    si p[pos].grado == i entonces
        p[pos].coeficiente:= p[pos].coeficiente + a
    sino
        p[pos].grado:= i
        p[pos].coeficiente:= a
    finsi
```

- **Implementación de las operaciones: eval**

Con dispersión abierta, la función eval sería de la siguiente forma, usando los iteradores de listas:

```
op eval (p: PolinomioDA, x: real) : real
    valor:= 0
    para i:= 0 … B-1 hacer
        para cada términoAct en p[i] hacer
            valor:= valor + términoAct.coeficiente·pow(x, términoAct.grado)
        finpara
    finpara
    devolver valor
```

Y con dispersión cerrada:

```
op eval (p: PolinomioDC, x: real) : real
    valor:= 0
    para i:= 0 … B-1 hacer
        si p[i].grado ≠ -1 entonces
            valor:= valor + p[i].coeficiente·pow(x, p[i].grado)
        finsi
    finpara
    devolver valor
```

- **Estudio de la eficiencia**

En ambas representaciones, la función **polinomioNulo** simplemente debe crear una tabla de tamaño B e inicializar sus términos, por lo que su tiempo de ejecución sería un O(B).

En cuanto a la operación **añadir_coef**, el tiempo es equivalente a una consulta/inserción en una tabla de dispersión. Por lo tanto, con dispersión abierta tenemos O(1+n/B) y con dispersión cerrada O(1/(1-n/B)).

Y para la función **eval**, en ambos casos debe recorrer toda la tabla, pero con la diferencia de que en la abierta debe recorrer todas las cubetas y en cada cubeta todos sus elementos (que serán n en total). Por lo tanto, con dispersión cerrada sería O(B) y con abierta O(n+B).

3. En una aplicación tenemos almacenado un conjunto de palabras utilizando una estructura de árboles. Escribir una operación que liste de forma eficiente todas las palabras que se encuentren alfabéticamente entre dos palabras dadas **pmin** y **pmax** (ambas inclusive). Por ejemplo, podemos consultar las palabras que están entre "caballo" y "cabello"; o entre "fuerza" y "mano".
   Para hacer este ejercicio, puedes elegir y utilizar cualquiera de las estructuras de árboles vistas en clase. Si se hace con árboles trie, no hay que acceder a los nodos, sino utilizar las operaciones genéricas sobre nodos trie: Consulta (n: NodoTrie, c: carácter): NodoTrie, y el iterador para cada carácter c hijo del nodo n hacer.
   Desarrollar la solución, escribiendo el pseudocódigo del algoritmo, explicando la solución, mostrando algún ejemplo y haciendo una estimación del tiempo de ejecución del algoritmo.

Respuesta 3.
Vamos a ver cómo sería la resolución de este ejercicio con árboles AVL y con árboles trie.

Con árboles AVL
La solución con árboles AVL es sencilla, porque estos árboles están ordenados de forma alfabética. Una solución muy simple, aunque ineficiente, sería recorrer todo el árbol, listando únicamente las palabras que estén en el rango requerido. Para ello usaremos el recorrido en in-orden del árbol, que es un recorrido en orden alfabético. Suponemos que el NodoAVL tiene dos punteros izq y der a sus hijos, y la clave palabra:

```
op ListarEntre (a: Puntero[NodoAVL]; pmin, pmax: cadena)
    si a ≠ NULO entonces
        ListarEntre(a→izq, pmin, pmax)
        si pmin ≤ a→palabra Y a→palabra ≤ pmax entonces Escribir(a→palabra)
        ListarEntre(a→der, pmin, pmax)
    finsi
```

Aunque esta operación calcula el resultado correcto, es ineficiente porque recorre todo el árbol. Podríamos eliminar ramas de árbol por las cuales podemos saber que no existirán palabras válidas. Por ejemplo, si la palabra del nodo actual (a→palabra) es menor o igual que pmin, entonces podemos podar el hijo izquierdo (a→izq), puesto que todas sus palabras son a su vez menores que el nodo actual. Por otro lado, si a→palabra es mayor o igual que pmax, podemos eliminar el hijo derecho. Esto podemos añadirlo fácilmente al algoritmo anterior:

```
op ListarEntre (a: Puntero[NodoAVL]; pmin, pmax: cadena)
    si a ≠ NULO entonces
        si a→palabra > pmin entonces ListarEntre(a→izq, pmin, pmax)
        si pmin ≤ a→palabra Y a→palabra ≤ pmax entonces Escribir(a→palabra)
        si a→palabra < pmax entonces ListarEntre(a→der, pmin, pmax)
    finsi
```

El tiempo de esta operación en el peor caso sería un recorrido completo del árbol, que con los AVL tarda un O(n), siendo n el número de palabras almacenadas en el árbol.

Con árboles trie
Igual que con los AVL, podemos hacer una primera versión sencilla, aunque ineficiente, basada en un recorrido completo del árbol, como el ListarTodas que vimos en clase:

```
op ListarEntre (a: Puntero[NodoTrie]; palabra: cadena; pmin, pmax: cadena; palabra)
    para cada carácter car hijo del nodo n hacer
        si car == $ entonces
            si pmin ≤ palabra Y palabra ≤ pmax entonces Escribir(palabra)
        sino ListarTodas(Consulta(n, car), palabra+car, pmin, pmax)
    finpara
```

La llamada inicial sería con: ListarEntre(raíz, “”, pmin, pmax).

Para hacer que esta operación sea más eficiente, podemos evitar las llamadas recursivas cuando la palabra que se va formando se sale del rango entre pmin y pmax. Esto podría tener la siguiente forma:

```
op ListarEntre (a: Puntero[NodoTrie]; palabra: cadena; pmin, pmax: cadena; palabra)
    para cada carácter car hijo del nodo n hacer
        si car == $ entonces
            si pmin ≤ palabra Y palabra ≤ pmax entonces Escribir(palabra)
        sino si palabra+car ≤ pmax Y pmin ≤ palabra+car+char(255) entonces
            ListarTodas(Consulta(n, car), palabra+car, pmin, pmax)
        finsi
    finpara
```

La comparación pmin ≤ palabra+car+char(255) merece una aclaración. ¿Por qué no ponemos, simplemente, pmin ≤ palabra+car? Supongamos, por ejemplo, que estamos buscando palabras mayores que pmin= “fuerza”. En un momento dado, podemos estar en una rama del árbol trie donde tenemos la palabra actual “fu”. Si usáramos la comparación pmin ≤ palabra+car, como “fu” < “fuerza”, no se cumple la comparación simple y se eliminaría esa rama del árbol. Esto es un error porque por esa rama “fu” sí puede haber palabras mayores que “fuerza”. Por ello, concatenamos el carácter 255, que es un valor ASCII mayor que los valores de las letras normales. Para hacernos una idea, podemos pensarlo como “concatenar una z” (la letra mayor); cuando la concatenamos a “fu”, tenemos “fuz”, así que ahora sí se cumple que “fuerza” < “fuz”. Y seguiríamos recorriendo la rama “fu”. En lugar de concatenar una “z”, concatenamos 255, que es mayor que cualquier letra válida.

En cuanto al tiempo de ejecución, en el peor caso se requeriría un recorrido de todo el árbol trie. Sabemos que un árbol trie con p prefijos tiene 2p+1 nodos con listas, y p nodos con arrays. En la representación con listas, el tiempo en cada nodo es constante, luego el tiempo total sería de un orden O(2p+1) = O(p). En la representación de los nodos con arrays, cada nodo es un array de a posiciones, siendo a el tamaño del alfabeto. Por lo tanto, el recorrido completo de todos los nodos en el peor caso sería O(a·p).

4. Este año la gripe se está extendiendo a una velocidad alarmante. El contagio se produce cuando dos personas, una enferma y otra sana, se encuentran; a partir del siguiente día, la persona que estaba sana pasa a estar enferma y puede contagiar a otros. Suponer que en cierta población tenemos **n** personas. Tenemos una matriz **M** de booleanos de tamaño **n**×**n** que indica en cada posición **M**[**i**, **j**] si las personas **i** y **j** se encuentran diariamente; esta matriz es simétrica. Por otro lado, el array **E** de booleanos de tamaño **n**, indica en cada **E**[**i**] si la persona **i** está inicialmente enferma el día 1 o no.
   Escribir un algoritmo completo y eficiente que calcule tres cosas: (1) el número de personas que se acaban contagiando; (2) el día en el cual ya no se producen más contagios; y (3) las personas que no se contagiarán (porque no tienen ninguna relación directa o indirecta con otros contagiados). Escribir el algoritmo en pseudocódigo y explicar la solución.

Respuesta 4.

Podemos modelar este problema como un problema de grafos, donde las personas son los nodos del grafo y las aristas son los pares de personas que se encuentran, siendo M la matriz de adyacencia. Sería un grafo no dirigido y sin pesos. De esta forma, el problema se puede plantear como un problema de caminos mínimos entre las personas contagiadas y las no contagiadas, siendo todas las aristas de coste 1. También se puede resolver como un problema de recorrido en grafos, siendo el contagio una búsqueda primero en anchura; en el primer nivel estarían las personas contagiadas inicialmente; sus hijos serían las que se contagian el día 1; los hijos de estos serían los que se contagian el día 2, y así sucesivamente.

La solución por búsqueda primero en anchura es más sencilla y computacionalmente más eficiente. Así que lo vamos a resolver de este modo. Podemos usar el array de marcas de visitado para almacenar el número de día en el que se contagia cada persona. Lo llamaremos díaContagio; lo inicializamos a -1, y para las personas contagiadas inicialmente a 0. El algoritmo, adaptado de la BPA, sería el siguiente:

```
op Contagio (M: array [1...n, 1…n] de booleano, E: array [1…n] de booleano,
             var numContagiadas, var díaMáximo: entero, var díaContagio: array [1…n] de entero)
var  C: Cola[entero]
     x, y: entero

     díaMáximo:= 0
     numContagiadas:= 0
     para i:= 1…n hacer
          si E[i] entonces
               marca[i]:= 0
               InsertaCola (i, C)
               numContagiadas:= numContagiadas + 1
          sino
               marca[i]:= -1
          finsi
     finpara
     mientras NOT EsVacíaCola (C) hacer
          x:= FrenteCola (C)
          SuprimirCola (C)
          para y:= 1…n hacer
               si M[x, y] Y marca[y] == -1 entonces
                    marca[y]:= marca[x] + 1
                    InsertaCola (y, C)
                    numContagiadas:= numContagiadas + 1
                    díaMáximo:= marca[y]
               finsi
          finpara
     finmientras
```

Los parámetros de entrada del algoritmo son la matriz de adyacencia, M, y el array de las personas contagiadas inicialmente, E. Los valores de salida del algoritmo son el número de personas que se contagiarán al final (numContagiadas), el día en el que se producirá el último contagio (díaMáximo) y el array que indica para cada persona el día en el que se contagiará (díaContagio). Obviamente, las personas que no se contagiarán serán las personas i con díaContagio[i] = -1.

Como se puede ver que el algoritmo es una adaptación de la BPA usando matrices de adyacencia, sabemos que su tiempo de ejecución en el peor caso es un $O(n^2)$. En el mejor caso, si no hubiera ningún contagio, sería un O(n).