# Problema C: subcadenas en la secuencia mira-y-di

Óscar Martín Sánchez
Unidad de Organización y Sistemas
Caja Madrid
oscarm@epersonas.net

Manuel Carro Liñares
Facultad de Informática
Universidad Politécnica de Madrid
mcarro@fi.upm.es

Conway y Guy dieron esta descripción de la secuencia que ellos llamaron *Look-and-Say* y que aquí llamaremos *mira-y-di* (traducido de *The Book of Numbers*, página 209):

> El primer término consiste en un "uno", así que el segundo es "uno uno." Esto consta de dos "unos," así que el tercero es "dos uno." Esto a su vez está formado por un "dos" y un "uno" y por tanto da "uno dos uno uno," y así sucesivamente.

Por tanto, la secuencia es `1`, `11`, `21`, `1211`, `111221`, `312211`,…

Nosotros vamos a considerar un tipo de secuencia más general en la que el primer término, la *semilla*, es una cadena cualquiera formada sólo por los dígitos `1`, `2` y `3`, y los otros términos se obtienen como se explica arriba. Queremos estudiar qué subcadenas aparecen más frecuentemente en los términos de esas secuencias.

El programa que se pide tiene que explorar un número dado de términos de la secuencia que empieza con una semilla también dada, encontrar la subcadena más larga que aparece al menos el número de veces que se requiere y escribirla. Si hay varias subcadenas candidatas de igual longitud, la primera (en orden lexicográfico) será la seleccionada. Supongamos, por ejemplo, que la semilla es `32`, que queremos considerar seis términos de la secuencia, y que la cantidad mínima de apariciones requeridas es cuatro. Nuestra secuencia en este caso se muestra abajo a la izquierda, y la lista de subcadenas que aparecen cuatro veces o más está a la derecha:

```
32
1312
11131112
31133112
1321232112
111312111213122112
```

| subcadena | apariciones | subcadena | apariciones |
|---|---|---|---|
| 1 | 27 | 131 | 4 |
| 11 | 12 | 2 | 13 |
| 111 | 4 | 21 | 5 |
| 112 | 5 | 3 | 10 |
| 12 | 9 | 31 | 6 |
| 13 | 6 | | |

Las subcadenas de longitud mayor que tres no aparecen la cantidad de veces necesaria. La cadena `111` aparece cuatro veces y es lexicográficamente anterior a todas las otras de la misma longitud, así que ésa es la respuesta. Si se hubieran pedido cinco apariciones en vez de cuatro, la respuesta al problema habría sido `112`.

Es fácil demostrar que sólo los dígitos `1`, `2` y `3` aparecen en la secuencia mira-y-di original, y garantizamos que todas nuestras secuencias cumplirán también esta propiedad. Dos propiedades más pueden tomarse como seguras: el término más largo que se tenga que estudiar tendrá como máximo 500 dígitos y siempre habrá una o más subcadenas que aparezcan la cantidad de veces requerida.

## Descripción de la entrada

La primera línea de la entrada contiene un número que indica cuántos problemas van a continuación. Cada problema ocupa una línea, con tres números separados por blancos. El primero es la semilla, es decir, una cadena de dígitos que se tomará como primer término de la secuencia. El segundo es el número de términos de la secuencia que se deben considerar. El tercero es la cantidad mínima de apariciones que se requieren.

## Descripción de la salida

Una línea para cada problema, con la solución al problema correspondiente.

## Ejemplo de entrada

```
3
32 6 4
11 3 1
123123123123 1 2
```

## Salida para el ejemplo de entrada

```
111
1211
123123123
```

# Solución al problema C:
# subcadenas en la secuencia mira-y-nombra

Recordemos rápidamente el enunciado. La secuencia mira-y-nombra se construye tomando como primer término (semilla) una cadena cualquiera de dígitos y, luego, leyendo un término para obtener el siguiente. Por ejemplo, tomando `32` como semilla obtenemos: `32`, `1312`, `11131112`, `31133112`, `1321232112`, etc. Cada problema viene dado por tres números en la entrada: el primero es la cadena que hay que tomar como semilla; el segundo, la cantidad de términos que hay que estudiar; el tercero, una cantidad de apariciones. El programa debe encontrar la subcadena más larga que aparece en los términos de la secuencia la cantidad de veces que indica el tercer número (o más).

Un teorema debido a Conway afirma que la longitud de los términos de cualquier secuencia mira-y-nombra aumenta de forma exponencial. Para este problema no parece que haya más remedio que generar y contar una por una todas las subcadenas que aparecen, sin desechar ninguna: el conocimiento del método de generación de la secuencia no proporciona ninguna ventaja en este sentido. Así que la complejidad del programa, tanto en tiempo como en espacio, será irremediablemente exponencial. Sin embargo, aún debemos esforzarnos para que el comportamiento del programa sea aceptable para datos no muy grandes. El límite de 500 dígitos que se da en el enunciado para la longitud de los términos de la secuencia corresponde a no más de unos 20 términos para semillas pequeñas. Es suficientemente bajo como para permitir algunas licencias en la codificación, pero un simple algoritmo de fuerza bruta quizá no sirva. Una buena elección de la estructura de datos será crucial. Veamos nuestra propuesta.

Recordemos que nuestras cadenas están compuestas únicamente por los caracteres `1`, `2` y `3`. Si una cadena aparece en un término también lo hacen todas sus subcadenas, lo que sugiere naturalmente la idea de usar una estructura arbórea, en la que cada prefijo común se almacene una sola vez. Eso traerá consigo un ahorro de memoria. Más concretamente, proponemos almacenar todos las cadenas que nos interesan en un árbol ternario, en el que cada arco esté etiquetado con un `1`, un `2` o un `3`. Cada nodo representa la cadena que resulta recorriendo el árbol desde la raíz hasta ese nodo. Y en cada nodo guardamos un contador que indica la cantidad de veces que ha aparecido la cadena correspondiente. Una estructura intrínsecamente ordenada como ésta facilita y hace más rápida la búsqueda de una cadena para aumentar su contador y la búsqueda final de la solución. A esta estructura de datos (o una pequeña variación de ella) se la denomina con el vocablo inglés *trie*.

Para analizar cada término de la secuencia, lo recorremos de izquierda a derecha, a la vez que descendemos por el trie, incrementando los contadores, desde la raíz, por los arcos adecuados según los dígitos que vamos encontrando. Si los arcos en cuestión no existen, los creamos sobre la marcha. Pero un solo recorrido como el descrito no es suficiente: sólo sirve para almacenar el término entero y contabilizar todos sus prefijos. Para considerar todas las subcadenas tenemos que hacer nuevos recorridos empezando cada vez por un carácter distinto de los que forman el término y volviendo a empezar desde la raíz del trie. Y cada vez que, de este modo, pasamos por un nodo, aumentamos su contador en una unidad. Por conveniencia (véase más abajo) aumentaremos también el contador de la raíz, que viene a representar la cadena vacía. Los detalles exactos están con el código del programa. Algunos gráficos ayudarán a entenderlo. La figura 1 muestra la secuencia de estados por los que pasa la construcción del trie durante la exploración del término `32`; recuadrada, sobre cada flecha, está la subcadena que contamos en cada paso.

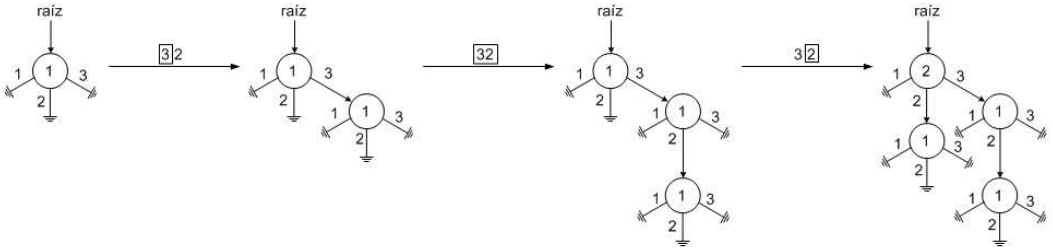


**Figura 1**

Al añadir ahora, tras el término `32`, su sucesor, el `1312`, primero colgaremos de la raíz la secuencia de arcos `1312`, después `312`, después `12` y después `2` (que ya existía), para llegar al resultado de la figura 2, en la que no mostramos los punteros nulos.

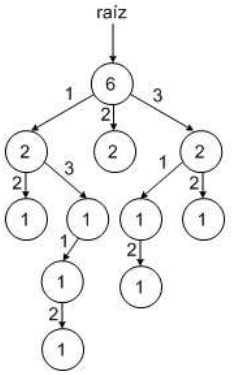


**Figura 2**

| Cantidad de términos | Cantidad de nodos | Cantidad de caracteres | Tiempo (mseg) |
|---|---|---|---|
| 2 | 3 | 3 | 0 |
| 5 | 20 | 60 | 0 |
| 8 | 78 | 355 | 0 |
| 11 | 517 | 5.119 | 0 |
| 14 | 3.261 | 67.557 | 0 |
| 17 | 17.538 | 721.744 | 0 |
| 20 | 92.449 | 8.088.396 | 5 |
| 23 | 489.842 | 94.892.811 | 40 |
| 26 | 2.512.392 | 1.088.369.957 | 185 |

**Figura 3**

En el trie de la figura 2 tenemos contadas las cadenas `1`, `12`, `13`, `131`, `1312`, `2`, `3`, `31`, `312` y `32`, que suponen un total de 21 caracteres, con sólo 11 nodos. La ventaja resulta más apreciable según aumenta el número de términos. La tabla de la figura 3 se ha obtenido estudiando secuencias mira-y-nombra con el término `1` como semilla, y con el número de términos calculados que se indica en la primera columna. Se muestra la cantidad de nodos que nuestro método almacena y la cantidad de caracteres totales (de todas las subcadenas de los términos) correspondientes en un método de fuerza bruta. La cantidad de nodos y la de caracteres no son comparables directamente, porque un nodo ocupa bastante más espacio de memoria, pero el crecimiento de la tercera columna es mucho más rápido y compensa rápidamente esa desventaja (se puede comprobar asumiendo 16 octetos por nodo y un octeto por carácter). La última columna muestra el tiempo aproximado de ejecución en un ordenador modesto (Pentium III a 930 MHz). Ojo, que el límite de 500 dígitos debe ser ampliado para calcular los términos del número 22 en adelante.

Vamos con los detalles del programa. Según el enunciado, podemos asumir que la longitud máxima de cualquier término será 500. Añadimos una posición para el carácter de fin de cadena:

```
#define maxLongTerm (500+1)
```

Las cabeceras que serán necesarias:

```
#include <stdio.h>  // scanf y printf
#include <string.h>  // strcpy y strncpy
```

El tipo de datos que usaremos, según lo explicado arriba:

```
typedef struct Nodo
{
   int contador;
   struct Nodo *rama[3];   // un subárbol para cada dígito
} Nodo;
```

Hay que tener un poco de cuidado con los índices del array *rama*, porque van de 0 a 2, mientras los dígitos van de `1` a `3`; así, el índice *i* se referirá al dígito `'1'+i`.

La estructura global del programa es ésta:

```
int main()
{
   int nrProbl;   // número de problemas en la entrada (también se toma de la entrada)
   char semilla[maxLongTerm];   // primer término (entrada)
   int nrTerm;   // número de términos (entrada)
```

```
   int nrAparic;   // número de apariciones necesarias (entrada)

   char term[maxLongTerm];   // término analizado en cada momento
   Nodo *raiz;   // puntero a la raíz del trie

   scanf ("%d", &nrProbl);
   while (nrProbl > 0)   // para cada problema
   {
      nrProbl --;
      scanf ("%s %d %d", semilla, &nrTerm, &nrAparic);
      raiz = creaNodo ();
      strcpy (term, semilla);
      nrTerm --;
      recorreAcumulando (term, raiz);
      while (nrTerm > 0)   // para cada término
      {
         sigTerm (term);
         nrTerm --;
         recorreAcumulando (term, raiz);
      }
      buscaYEscribeRdo (raiz, nrAparic);
      libera (raiz);
   }
   return 0;
}
```

Ciertas comprobaciones que serían necesarias en un programa *serio* van a ser omitidas en este programa. Igual que aquí arriba hemos supuesto que la entrada contiene lo que esperamos de ella, algo más abajo vamos a suponer que siempre obtenemos la memoria que solicitamos y que los términos de la secuencia que calculamos no sobrepasan los 500 caracteres de longitud. En un concurso como éste esas comprobaciones no son necesarias y nuestro código resultará así algo menos complicado.

Para empezar por lo más fácil, veamos cómo reservar memoria para un nodo nuevo y, antes aún, cómo liberar, de forma recursiva, toda la memoria usada en el trie.

```
void libera (Nodo *pNodo)
{
   if (pNodo)
   {
      libera (pNodo->rama[0]);
      libera (pNodo->rama[1]);
      libera (pNodo->rama[2]);
      free (pNodo);
   }
}

Nodo *creaNodo ()
{
   Nodo *pNodo;
   pNodo = (Nodo *) malloc (sizeof(Nodo));
   pNodo->contador = 0;
   pNodo->rama[0] = NULL;
   pNodo->rama[1] = NULL;
   pNodo->rama[2] = NULL;
   return pNodo;
}
```

Como sabemos, calcular un término significa describir el anterior. La siguiente función machaca el argumento que recibe con su descripción.

```
void sigTerm (char *term)
{
   char descr[maxLongTerm];   // término que vamos a calcular
   int dig = term[0];   // dígito que estamos contando
```

```
   int cont = 0;   // contador
   char *pTerm = term;   // puntero para recorrer term
   char *pDescr = descr;   // puntero para recorrer descr

   while (*pTerm)
   {
      while (*pTerm == dig)
      {
         cont ++;
         pTerm ++;
      }
      *pDescr = '0'+cont;
      *(pDescr+1) = dig;
      pDescr += 2;
      dig = *pTerm;
      cont = 0;
   }
   *pDescr = 0;   // para terminar la cadena
   strcpy (term, descr);
}
```

Vamos con la penúltima parte: el análisis de cada término y la inserción de cadenas y subcadenas en el trie. El proceso es tal como se describe al principio de este artículo. El mismo argumento *sufijo*, que en la llamada tiene el contenido del término entero, va luego apuntando a cada carácter del término: considerado como cadena de caracteres, *sufijo* va guardando los distintos sufijos del término original. La variable *aux* recorre cada uno de esos sufijos a la vez que recorremos el trie y acumulamos en los contadores de cada nodo intermedio y del nodo final. De ese modo estoy contando cada prefijo de cada sufijo, es decir, cada subcadena. Recuérdese que en la llamada a esta función, el primer parámetro contiene el término que hay que analizar y el segundo apunta a la raíz del trie.

```
void recorreAcumulando (char *sufijo, Nodo *laRaiz)
{
   while (*sufijo)
   {
      acumula (sufijo, laRaiz);
      sufijo ++;
   }
}

void acumula (char *cadena, Nodo *pNodo)
{
   int indice;
   pNodo->contador ++;   // ¡ojo!, véase el comentario abajo
   while (*cadena)
   {
      indice = *cadena - '1';
      if (pNodo->rama[indice] == NULL)
         pNodo->rama[indice] = creaNodo();
      pNodo = pNodo->rama[indice];
      pNodo->contador ++;
      cadena ++;
   }
}
```

La instrucción marcada merece una explicación. Lo que estamos haciendo es acumular también en el contador de la raíz, es decir, en el contador de cadenas vacías, lo cual puede parecer raro. La razón es que así conseguimos que *todo* el trie cumpla una interesante propiedad: el contador de cualquier nodo es mayor o igual que el de cualquiera de sus hijos (y que la suma de ellos, en realidad). O lo que es lo mismo, una cadena aparece al menos tantas veces como sus extensiones, lo que parece bastante natural. Esta propiedad será usada en la función *buscaYEscribeRdo* para acortar la búsqueda de la solución.

Encaremos, finalmente, la búsqueda del resultado, que nos va a proporcionar materia para alguna discusión, digamos, estilística. Un procedimiento iterativo tiene sus ventajas, pero también presenta un problema: durante el recorrido del trie necesitaremos subir de los nodos hijos a sus padres, y no hay forma

directa de hacer tal cosa, porque los nodos no tienen punteros "hacia arriba". Tendríamos que guardar una tabla de punteros a todos los nodos del camino que estamos visitando. Por el contrario, un procedimiento recursivo, como el *busca* que a continuación damos en seudocódigo, resulta muy elegante.

```
string busca (subarbol)
{
   if (subarbol == NULL || subarbol->contador < nrAparic)
      return fracaso;
   else
      return construirRdo (busca(subarbol->rama[0]),
                           busca(subarbol->rama[1]),
                           busca(subarbol->rama[2]));
}
```

La condición *subarbol->contador < nrAparic* sirve como caso base, porque el contador de un nodo siempre es mayor o igual que los de cualquiera de sus descendientes, según quedó explicado. La función *construirRdo* toma el mejor de los tres resultados obtenidos de sus subárboles y le pone delante un `'1'`, un `'2'` o un `'3'` según corresponda. Si las tres llamadas recursivas han fracasado, devolverá la cadena vacía (que no es lo mismo que un fracaso, porque el contador de *este* nodo sí cumple las condiciones requeridas).

Aquí, sin embargo, aparece un problema recurrente: estamos malgastando el espacio de memoria. Cada vez que se ejecuta el cuerpo de la función *busca* (excepto en los casos base) tenemos que usar tres nuevas cadenas de caracteres para guardar los resultados parciales y trabajar con ellos. Como puede haber hasta 500 copias de la función ejecutándose a la vez, estamos usando 1500 cadenas de caracteres de tamaño 500 cada una de ellas. Un lujo que no queremos permitirnos (aunque quizá podamos: ya se ha dicho que los límites no están tan ajustados).

Ese seudocódigo se puede manipular para que use una sola variable de tipo cadena, la misma en las tres llamadas, que puede ser el mismo argumento que se ha recibido. Quedaría algo como esto que sigue, donde el parámetro *cadena* contendrá en cada momento la cadena representada por el nodo que estamos visitando y el parámetro *rdo* irá conteniendo el mejor candidato a solución encontrado hasta el momento.

```
busca (subarbol, cadena, rdo)
{
   if (subarbol == NULL || subarbol->contador < nrAparic)
      fracaso;
   else
   {
      if (strlen(cadena) > strlen(rdo))
         strcpy (rdo, cadena);
      for (i=0; i<=2; i++)
         busca (subarbol->rama[i], agrega(cadena,'1'+i), rdo);
   }
}
```

(La función *agrega* añade el carácter al final de la cadena y devuelve la propia cadena resultante.)

Aquí es donde aparecen las cuestiones estilísticas. El tercer parámetro, *rdo*, es un puntero constante que apunta a una dirección de memoria en la que hay almacenada una cadena de caracteres que todas las copias de la función pueden modificar. Otro tanto pasa con el segundo parámetro, *cadena*. Es decir, que eso son en realidad dos variables globales, y el hecho de que estén camufladas las hace menos justificables y menos manejables. Además, la recursión se vuelve así impropia: sirve, eso sí, para mantener implícitamente la tabla de punteros a nodos de la que hablábamos arriba. Pero a cambio obtenemos un código menos claro y poco elegante.

Las dos opciones anteriores pueden valer. Nos decidimos, sin embargo, por un procedimiento iterativo, que de paso nos permite practicar el saludable ejercicio de emular recursión mediante iteración y pilas. Usamos el array *pila* para guardar la pila de punteros a nodos; la variable *cadena* contiene en cada momento la cadena representada por el nodo que estamos visitando; la variable *prof* guarda la profundidad del nodo en el trie (la raíz tiene profundidad cero), que coincide con la longitud de la cadena que ese nodo representa; la variable *mejor* contiene el mejor candidato a solución encontrado hasta el

momento; y la variable *procedencia* indica desde qué nodo hemos llegado al actual, para poder actuar en consecuencia: '1', '2' o '3' indican que venimos de abajo, de la rama indicada, y un '0' indica que hemos llegado desde arriba, desde el nodo padre. El contenido de la variable *cadena* nos servirá también para saber por qué camino hemos llegado al nodo actual. Las siguientes dos funciones y definiciones de tipos nos sirven para manejar la pila; según la terminología tradicional, nos referimos con *push* al procedimiento que apila un elemento y con *pop* al que lo saca.

```
typedef Nodo *ElemPila;
typedef ElemPila *Pila;

void push (Pila *p, Nodo *n)
{
  **p = n;
  (*p) ++;
}

Nodo *pop (Pila *p)
{
  (*p) --;
  return **p;
}

void buscaYEscribeRdo (Nodo *pNodo, int umbral)
{
   Pila pila = (Pila) malloc (maxLongTerm*sizeof(ElemPila));
   char cadena[maxLongTerm];
   char mejor[maxLongTerm];
   int longMejor = 0;
   int prof = 0;
   char procedencia = '0';

   while (prof >= 0)   // saldremos cuando intentemos ir al padre de la raíz
   {
      if ((pNodo==NULL) || (pNodo->contador<umbral))   // fracaso
      {
         prof --;
         pNodo = pop (&pila);
         procedencia = cadena[prof];
      }
      else if (procedencia == '3')   // tenemos que subir en el trie
      {
         if (prof > longMejor)   // sólo comprobamos esto justo antes de subir
         {
            strncpy (mejor, cadena, prof);
            mejor[prof] = 0;
            longMejor = prof;
         }
         prof --;
         pNodo = pop (&pila);
         procedencia = cadena[prof];
      }
      else   // tenemos que bajar en el trie
      {
         push (&pila, pNodo);
         cadena[prof] = procedencia+1;
         prof ++;
         pNodo = pNodo->rama[procedencia-'0'];
         procedencia = '0';
      }
   }
   printf ("%s\n", mejor);
}
```

Sin duda, el programa que hemos presentado admite aún algunos ajustes que mejoren sus prestaciones. Posibilidades de ahorro de memoria aparecen al observar que la gran mayoría de los nodos de nuestros

árboles tienen un único hijo: estudiando 20 términos de la secuencia con semilla `1` aparecen 92.449 nodos, de los cuales 1.069 son hojas, 90.388 tienen un solo hijo, 916 tienen dos hijos y 76 tienen tres. Además, aproximadamente la mitad de los dígitos que aparecen en cualquier secuencia mira-y-nombra son `1`. En vista de esto, puede ser ventajoso usar la idea de los tries denominados *Patricia*, que permiten etiquetar cada arco del árbol con una secuencia de dígitos en vez de uno sólo. Si nos importa más el tiempo de ejecución, podemos agrupar las llamadas a `malloc` de forma que se reserve espacio de una sola vez para, digamos, 1000 nodos, lo que ahorrará mucho trabajo al sistema. Pero las florituras de este tipo no suelen ser parte de un programa de concurso.