Estructuras para cadenas de caracteres

November 15, 2019

1 Arboles de sufijos

Estructura de **arbol** para guardar todos los **sufijos posibles** dentro de un conjunto de cadenas de caracteres.

Sirve de **estructura posible para implementar un diccionario**.

Idea:

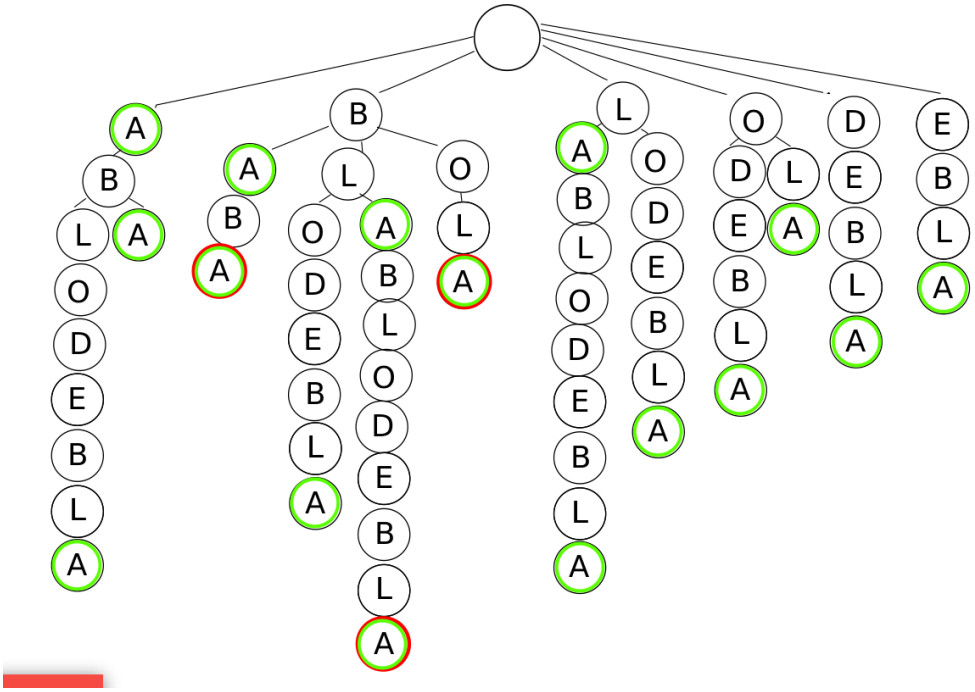
- Raiz "vacía"
- Cada ramificacion procede de una **expansión posible de un prefijo de sufijo** (hasta 26 posibles).
- Las aristas corresponden a **agregar letras**, los nodos a **partes (prefijos) de sufijos** (por el camino a la raiz).
- Los **nodos terminales** son necesariamente sufijos, pero nodos internos también lo pueden ser.
- Los sufijos que comparten un prefijo comparten parte del camino a la raiz.
- Dos flags: final de sufijo/final de palabra.

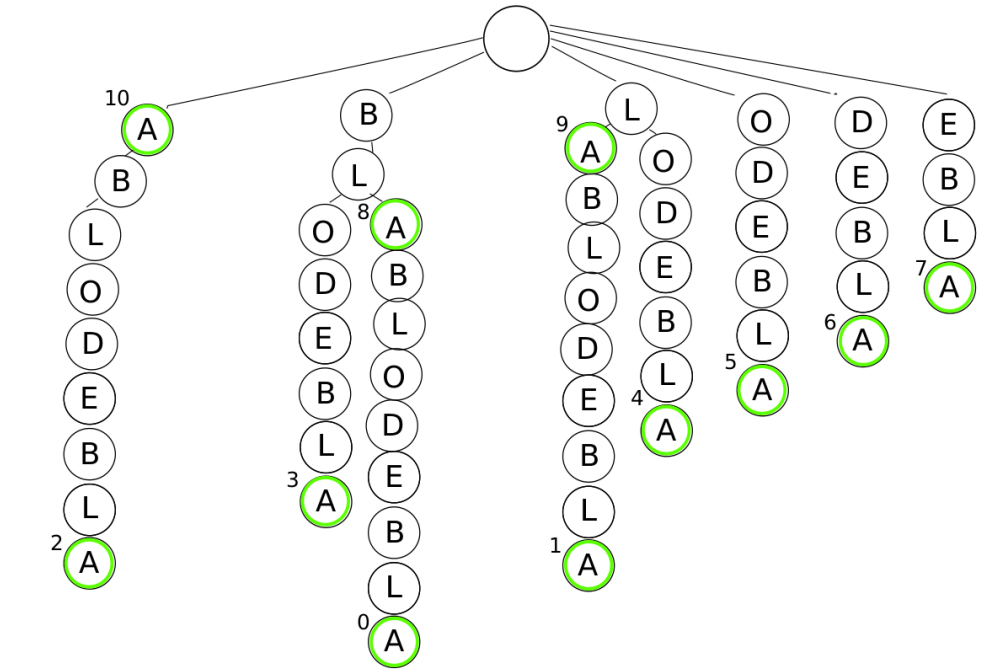
Ejemplo:
BLABLODEBLA
BOLA
BABA
Uso muy eficiente como **diccionario de palabras**, una vez que está construido:

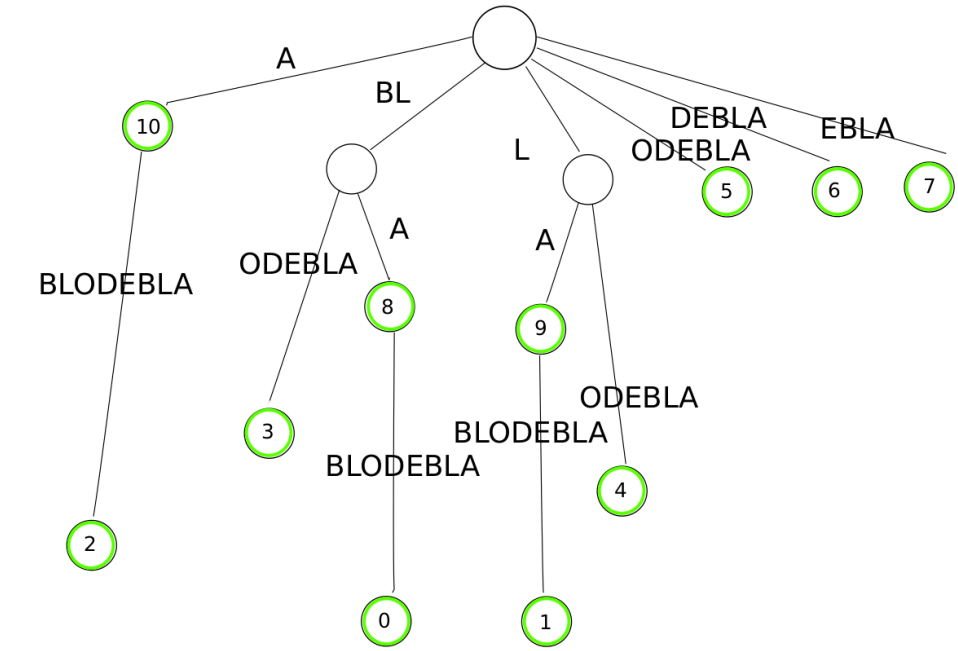
- Bajar desde el nodo raiz e ir comparando con la cadena buscada.
- Parar si la siguiente letra no está o si al llegar a la última letra no está el flag de palabra prendido.

$O(m)$ en el tamaño de la cadena buscada (patrón).
A priori $O(n^2)$ de preprocesamiento pero se puede mejorar en hasta $O(n)$.

- Uso posible de **diccionario de sufijos**.
- Ahora, esa misma estructura puede estar usada eficientemente para buscar patrones **en una sola (grande!) cadena (en lugar de un conjunto de palabras)**, de manera opuesta al KMP: preprocesamos la cadena s y podemos hacer muchas queries de busqueda de patrones.
- Guardamos ahora sólo el flag de los sufijos y el índex del inicio del sufijo en la cadena.


A B L A O D E B L A
A B A
B L O D E B L A O L A
L A B L O D E B L A O D E B L A
O D L E A B L A
D E B L A
E B L A


10 A B L O D E B L 2 A
B L O D E B L 3 A 8 A B L O D E B L 0 A
9 A B L O D E B L 1 A L O D E B L 4 A
O D E B L 5 A
D E B L 6 A
E B L 7 A



Ejemplo:
BLABLODEBLA
Pero la estructura es grande (redundancia): manera de reducirla?
Estructura de suffix tree: todos los nodos con 1 hijo o menos que no son sufijos se colapsan.

- Estructura **muy eficiente para procesar muchas solicitudes de matchings** etc... en una sola (grande) cadena.
- Estructura compacta: $2n$ nodos a lo más (n nodos terminales/sufijos, a lo más, y máximo $n-1$ nodos internos de bifurcación, y la raiz).

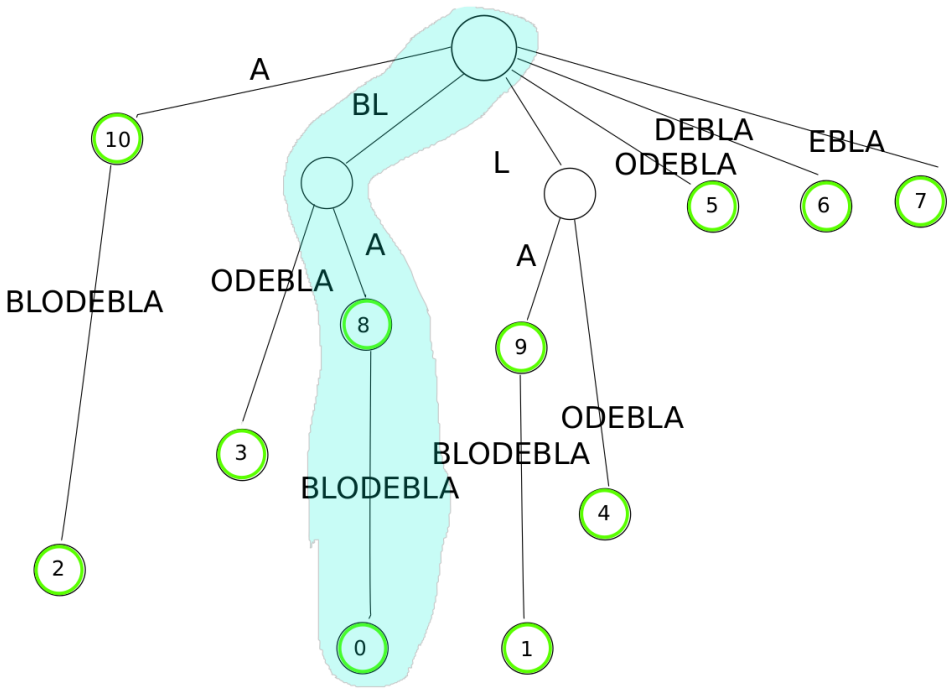
Como hacer para que todos los sufijos estén en nodos terminales?
Agregar un caracter dummy único al final de la cadena!

1.1 Aplicaciones de los suffix trees

1.1.1 Matching de strings

- Un matching de un patrón con una sub-cadena de una cadena se puede ver como un una **coincidencia del patrón con algún de los prefijos del arbol de sufijos**.
- Idea: tomar los caracteres del patrón para bajar el arbol.
- Si no se encuentran todos los caracteres, la sub-cadena no está.
- Si se encuentran, contar el número de nodos-sufijos ubicados a partir del nodo terminando la búsqueda.

Ejemplo: buscando “BLA”.
Complejidad:



- $O(m)$ para la búsqueda del patrón
- $O(n_{occ})$ para la determinación de todas las occurencias (exploración del sub-arbol enraizado en el nodo en que se acabó la búsqueda; no más de $2n_{occ}$ nodos).
- Guardando el número de sufijos abajo de cada nodo, $O(1)$.

No depende del tamaño de la cadena en la cual se busca (mucho mejor que cualquier otro algoritmo de búsqueda).

Ahora, el punto es que se necesita tener la estructura de suffix tree construida.

1.1.2 Mayor sub-cadena repetida en un string

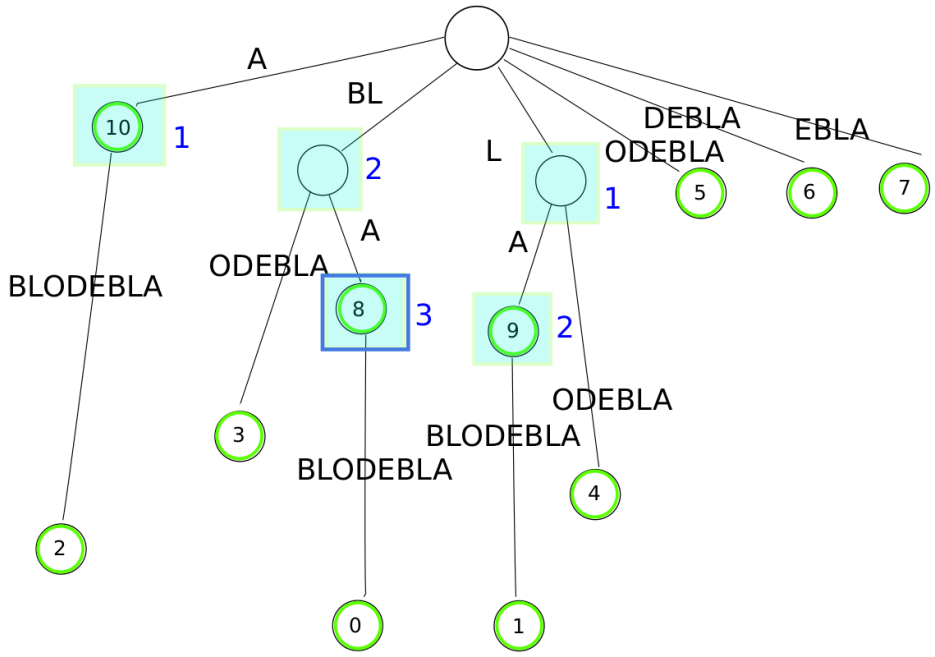
- Mayor sub-cadena repetida?
- Sub-cadena con mayor frecuencia de apariciones (de longitud $\geq x$)?

Para este tipo de pregunta: examinar los nodos internos!

- Cualquier nodo interno es un patrón repetido dentro de la cadena examinada.
- Su longitud es la longitud acumulada en el camino a la raiz (profundidad).
- La frecuencia de aparición es el **número de sufijos en el sub-arbol enraizado en este nodo interno**.
- Recorrido del arbol para ubicar los nodos deseados.

1.1.3 Mayor sub-cadena común en dos strings

- Dos (o más!) cadenas, esa vez.



- Queremos buscar **sub-cadenas comunes a las dos strings**.

Como usar el suffix tree para eso?

- Idea: construir el suffix tree para **la unión de todos los sufijos de las cadenas**.
- Marcar los nodos-sufijos con los **ids de las cadenas de las cuales son sufijos** (pueden ser sufijos de varias).

Ejemplo:

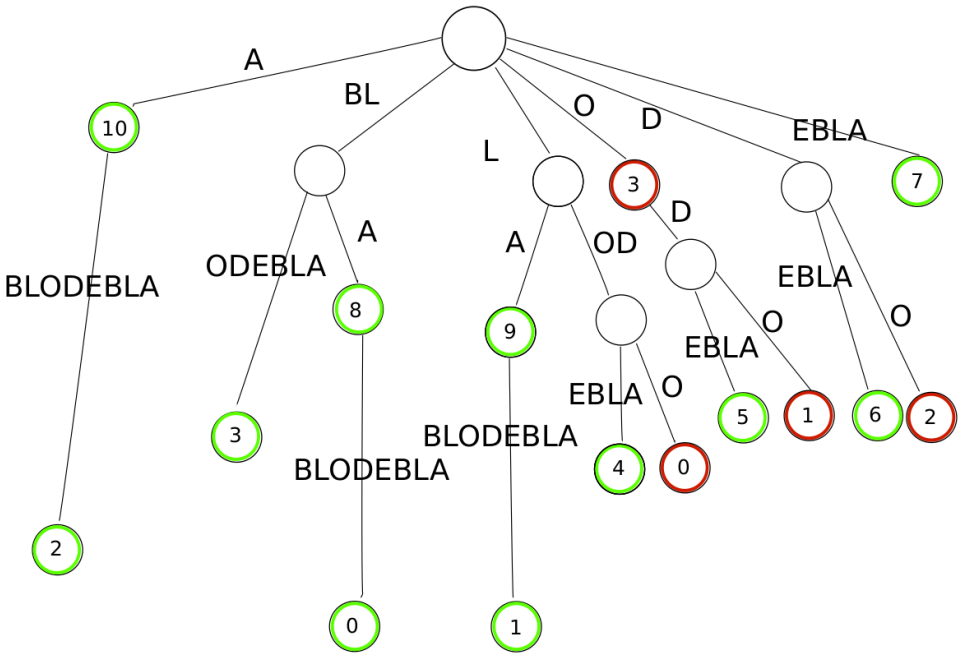
BLABLODEBLA

LODO

Donde estan las sub-cadenas comunes?

- Cualquier nodo interno en el sub-arbol del cual se incluyen sufijos de todas las sub-cadenas **es una sub-cadena común**.
- Basta recorrer los nodos internos de este arbol para determinar el de profundidad mayor.
- En el ejemplo:

O
OD
L
LOD
D



... todo pinta bien. Pero la mala noticia es que la **implementación en práctica de los suffix trees (en particular en entorno estresante) no es tan fácil**, a pesar de que su construcción es rapida ($O(n)$).

Algoritmo de Ukkonnen con una propuesta de implementación aquí.

La buena noticia: hay una estructura que tiene **propriedades idénticas** que se puede implementar relativamente fácil.

Es el suffix array (arreglo de suffijos).

2 Arreglos de suffijos

El **suffix array** es un arreglo de enteros, permutación de los posibles sufijos, representados por el **índex de su primer elemento en la cadena original**, con los **sufijos ordenados**.

Ejemplo: (BLABLODEBLA)

```
BLABLODEBLA 0
LABLODEBLA 1
ABLODEBLA 2
BLODEBLA 3
LODEBLA 4
ODEBLA 5
DEBLA 6
EBLA 7
BLA 8
LA 9
A 10
```

Al **ordenarlas**:

```
A 10
ABLODEBLA 2
BLA 8
BLABLODEBLA 0
BLODEBLA 3
DEBLA 6
EBLA 7
LA 9
LABLODEBLA 1
LODEBLA 4
ODEBLA 5
```

El suffix array de esta cadena es entonces:

$$(10, 2, 8, 0, 3, 6, 7, 9, 1, 4, 5)$$

- Observar que el **recorrido pre-orden del suffix tree** (construido adecuadamente) da exactamente este orden.
- Los elementos del arreglo corresponden a los **nodos-sufijos** (terminales o internos).
- Las **bifuraciones del arbol** corresponden a rangos de **indices contiguos** en el arreglo.

2.1 Construcción del suffix array

```
#include <iostream>
#include <algorithm>
using namespace std;
#define SIZEMAX 1000
string s;
int suffarray[SIZEMAX];

inline bool suffixcomp(int i,int j) {
    // Get the corresponding suffixes and compare them
    // with the lexicographical order
    return (s.substr(i)<s.substr(j));
}

int main() {
    cin >> s;
    // Write the distinct ids of all suffixes as their initial index
    for (int i=0;i<s.size();i++)
        suffarray[i]=i;
    // Here, all the indices-suffixes are sorted by comparing their
    // corresponding suffixes
    sort(suffarray,suffarray+s.size(),suffixcomp);
    for (int i=0;i<s.size();i++)
        cout << s.substr(i) << endl;
```

```
    cout << endl;
    for (int i=0;i<s.size();i++)
        cout << s.substr(suffarray[i]) << endl;
    return 0;
}
```

Complejidad de este algoritmo de construcción?

- Cada comparación de sufijos es **lineal**.
- $O(n \log n)$ elementos de comparaciones.

Total $O(n^2 log n)$. Ok para cadenas hasta tamaño de hasta 10^3.
Hay un mejor algoritmo en $O(n \log n)$ para poder procesar en particular largas cadenas.
Idea: como al final se trata de **caracteres (enteros chicos)**, se puede aprovechar para usar el **Radix Sort** y ordenar las cadenas por pares de caracteres.

2.1.1 Counting sort

- Cuando el **rango** de los datos es chico.
- La position de un dato en el arreglo ordentado **se deduce directamente del histograma acumulado**.
- Requiere poder construir el histograma.
- **Tiempo linear** en el número de elementos y en el rango M:

$$O(n + M).$$

```
void countSort() {
    memset(&freqs[0],0,sizeof(freqs));
    // Determine the frequencies of possible values
    for (int i=0;i<a.size();i++)
            freqs[a[i]]++;
    // Accumulate all the frequencies
    int sfs=0;
    for (int j = 0; j < freqs.size(); j++) {
        int freq = freqs[j]; freqs[j] = sfs; sfs += freq;
    }
    // Deduce the location of the corresponding data in the sorted array
    for (int i=0;i<a.size();i++)
            tmp[freqs[a[i]]++] = a[i];
    // Copy back the sorted data
    for (int i=0;i<a.size();i++)
        a[i] = tmp[i];
}
```

2.1.2 Radix sort

Por construcción counting sort es **estable**: dos datos con el mismo valor se encuentran en el mismo lugar antes y después de este ordenamiento.

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	0	GATAGACA$	71 (G)	65 (A)
1	1	ATAGACA$	65 (A)	84 (T)
2	2	TAGACA$	84 (T)	65 (A)
3	3	AGACA$	65 (A)	71 (G)
4	4	GACA$	71 (G)	65 (A)
5	5	ACA$	65 (A)	67 (C)
6	6	CA$	67 (C)	65 (A)
7	7	A$	65 (A)	36 ($)
8	8	$	36 ($)	00 (-)

Initial ranks RA[i] = ASCII value of T[i]
$ = 36, A = 65, C = 67, G = 71, T = 84

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+1]
0	8	$	36 ($)	00 (-)
1	7	A$	65 (A)	36 ($)
2	5	ACA$	65 (A)	67 (C)
3	3	AGACA$	65 (A)	71 (G)
4	1	ATAGACA$	65 (A)	84 (T)
5	6	CA$	67 (C)	65 (A)
6	0	GATAGACA$	71 (G)	65 (A)
7	4	GACA$	71 (G)	65 (A)
8	2	TAGACA$	84 (T)	65 (A)

If SA[i] + k >= n (beyond the length of string T),
we give a default rank 0 with label -

Imagina ahora que tenemos números grandes: la idea de radix sort es aplicar iterativamente el counting sort de dígito en dígito, **empezando por el dígito menos significativo**.

Por la propriedad de **estabilidad**, si números comparten el mismo dígito de peso más significativo, **guardarán el orden del counting sort realizado antes sobre el dígito anterior**.

Ejemplo:

```
123 456 21 457 981 112 356 423
21 981 112 123 423 456 356 457
112 21 123 423 456 356 457 981
21 112 123 35 423 456 457 981
```

Complejidad (base b):

$$O(\log_b(M)(n+b))$$

2.1.3 Construcción mejorada de los suffix array

Primer paso:

- atribuye inicialmente a cada caracter de la cadena un **index** (rankarray); para caracteres fuera de rango, usa el index 0.

```
for (int i=0;i<s.size();i++) {
    rankarray[i] = s[i]-'/';
}
```

- ordena los sufijos en función del par (rankarray[primer caracter],rankarray[second caracter]), por un **radix sort**.
- después de eso, están casi todos ordenados (excepto los sufijos que coinciden en los 2 primeros caracteres).

[from S. Halim's book]

Una **observación importante**: después de los 2 primeros caracteres, los **dos siguientes son... los dos primeros caracteres de otro sufijo!** Entonces les podemos comparar facilmente entre sí (ya lo hemos hecho!).

Segundo paso:

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	0	GATAGACA$	6 (GA)	7 (TA)
7	4	GACA$	6 (GA)	5 (CA)
8	2	TAGACA$	7 (TA)	6 (GA)

$- (first item) is given rank 0, then for i = 1 to n-1, compare rank pair of this row with previous row

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+2]
0	8	$	0 ($-)	0 (--)
1	7	A$	1 (A$)	0 (--)
2	5	ACA$	2 (AC)	1 (A$)
3	3	AGACA$	3 (AG)	2 (AC)
4	1	ATAGACA$	4 (AT)	3 (AG)
5	6	CA$	5 (CA)	0 ($-)
6	4	GACA$	6 (GA)	5 (CA)
7	0	GATAGACA$	6 (GA)	7 (TA)
8	2	TAGACA$	7 (TA)	6 (GA)

If SA[i] + k >= n (beyond the length of string T), we give a default rank 0 with label -

- genera otro índex para cada caracter, en función del **orden en la clasificacion anterior**: si cada sufijo es estrictamente superior al anterior, se le da un ranking r++, sino se le da el mismo que el anterior.
- se hace un **ordenamiento lexicografico con el par (rankingarray[suffarray[i]],rankingarray[suffarray[i]+2])**: la llave es que el rankeo usado para ordenar incluye el trabajo anterior (las subcadenadas en i+2).

[from S. Halim's book]
Se repite el proceso: entre i,i+4; luego i,i+8. . .

- Hasta que todos los RA[s] sean distintos.
- Cada par de counting sort (radix sort) toma $O(n)$. Se repite todo $O(\log n)$ veces entonces en total $O(n \log n)$. Con un sort normal $O(n \log^2 n)$.
- Permite manejar cadenas de tamaño hasta 10^5 !

[from S. Halim's book]

```
#include <iostream>
#include <algorithm>
#include <string.h>

using namespace std;
#define SIZEMAX 100000

int freqs[SIZEMAX];
int suffarray[SIZEMAX];
int rankarray[SIZEMAX];
int tmprankarray[SIZEMAX];
int tmpsuffarray[SIZEMAX];
string s;

// This function takes all
void radixSort(int k) {
    int m=max(100,(int)s.size());
    memset(&freqs[0],0,sizeof(freqs));
    for (int i=0;i<s.size();i++)
```

i	SA[i]	Suffix	RA[SA[i]]	RA[SA[i]+4]
0	8	$	0 ($---)	0 (----)
1	7	A$	1 (A$--)	0 (----)
2	5	ACA$	2 (ACA$)	0 (----)
3	3	AGACA$	3 (AGAC)	1 (A$--)
4	1	ATAGACA$	4 (ATAG)	2 (ACA$)
5	6	CA$	5 (CA$-)	0 (----)
6	4	GACA$	6 (GACA)	0 ($---)
7	0	GATAGACA$	7 (GATA)	6 (GACA)
8	2	TAGACA$	8 (TAGA)	5 (CA$-)

Now all suffixes have different ranking
We are done

```
        if (i+k<s.size())
            freqs[rankarray[i+k]]++;
        else
            freqs[0]++;
    // Accumulate all the frequencies
    int sfs=0;
    for (int i = 0; i < m; i++) {
        int freq = freqs[i]; freqs[i] = sfs; sfs += freq;
    }
    // Deduce the location of the data in the sorted array
    for (int i=0;i<s.size();i++)
        if (suffarray[i]+k<s.size())
            tmpsuffarray[freqs[rankarray[suffarray[i]+k]]++] = suffarray[i];
        else
            tmpsuffarray[freqs[0]++] = suffarray[i];
    // Copy back the sorted data
    for (int i=0;i<s.size();i++)
        suffarray[i] = tmpsuffarray[i];
}

void buildSuffArray() {
    for (int i=0;i<s.size();i++) {
        rankarray[i] = s[i]-'/';
    }
    // Init the suffix array
    for (int i=0;i<s.size();i++)
```

```
        suffarray[i]=i;
    // Does this log n times
    for (int k=1;k<s.size();k<<=1) {
        // Sort according to k-1th element
        radixSort(k);
        // Sort according to first element (this is a stable sort!)
        radixSort(0);
        // Re-generate the ranking based on the sorting
        // The smallest element gets the rank 0
        int r=0;
        tmprankarray[suffarray[0]] = 0;
        for (int i=1;i<s.size();i++)
            if (rankarray[suffarray[i]] != rankarray[suffarray[i-1]] ||
                rankarray[suffarray[i]+k] != rankarray[suffarray[i-1]+k])
                tmprankarray[suffarray[i]] = ++r;
            else
                tmprankarray[suffarray[i]] = r;
        for (int i=0;i<s.size();i++) {
             // Update the rank array
             rankarray[i] = tmprankarray[i];
        }
        cout << endl;
    }
}

int main() {
    cin >> s;
    cout << s.size() << endl;
    buildSuffArray();
    for (int i=0;i<s.size();i++)
        cout << s.substr(i) << endl;
    cout << endl;
    for (int i=0;i<s.size();i++)
        cout << s.substr(suffarray[i]) << endl;
    return 0;
}
```

2.2 Uso de los suffix arrays: busqueda de sub-cadenas

Con una cadena s (tamaño n) y su suffix array construido, podemos implementar eficientemente la búsqueda de una cadena patrón pattern (tamaño m)

- Buscar dentro del arreglo de sufijos el primer índex de sufijo tal que este sufijo no se compare (por orden lexicografico) como "inferior" al patrón (eventualmente 0). Da una **cuota inferior**.
- Buscar dentro del arreglo de sufijos el primer índex de sufijo tal que este sufijo se compare (por orden lexicografico) como "superior" al patrón (eventualmente el tamaño del sufijo). Da una **cuota superior**.

Por construcción, **todos los sufijos superiores o igual a la cuota inferior y inferiores a la cuota superior contendrán el patrón buscado**.

Complejidad?

- Dos búsquedas binarias en el arreglo de sufijos: $O(\log n)$.
- Para todas las comparaciones que hacer en la búsqueda binaria: hasta m comparaciones de caracteres.

En total: $O(m \log n)$. Aceptable en muchos casos donde la **cadena que se busca es chica, mientras la en la cual se busca es mucho más grande**.

```
inline bool suffCompare1(int ind,const string &pattern) {
    return (s.substr(ind).compare(0,pattern.size(),pattern)<0);
}
inline bool suffCompare2(const string &pattern,int ind) {
    return (s.substr(ind).compare(0,pattern.size(),pattern)>0);
}
pair<int,int> match(const string &pattern) {
    int *low = lower_bound (suffarray, suffarray+s.size(), pattern, suffCompare1);
    int *up = upper_bound (suffarray, suffarray+s.size(), pattern, suffCompare2);
    return make_pair((int)(low-suffarray),(int)(up-suffarray));
}
int main() {
    cin >> s;
    buildSuffArray();
    for (int i=0;i<s.size();i++)
        cout << s.substr(i) << endl;
    cout << endl;
    for (int i=0;i<s.size();i++)
        cout << s.substr(suffarray[i]) << endl;
    string pattern;
    cin >> pattern;
    cout << endl;

    pair<int,int> p = match(pattern);
    cout << p.first << " " << p.second << endl;
    for (int i=p.first;i<p.second;i++)
        cout << s.substr(suffarray[i]) << endl;
    return 0;
}
```

2.3 Uso de los suffix arrays: Mayor prefijo comun

- Supongamos que tenemos el arreglo de sufijos de una cadena s; queremos calcular la **tabla de los mayores prefijos en común entre sufijos consecutivos en el arreglo de sufijos** (nos va a servir por ejemplo el mayor sub-cadena repetida dentro de la cadena).
- Es el **LCP** en inglés.
- Por definición, lcp[0]=0.
- Se puede construir por un algoritmo ingenuo comparando los sufijos consecutivos.

```
for (int j=1;j<n;j++) {
    int l=0;
    while (suffarray[j-1]+l<n && s[suffarray[j]+l]==s[suffarray[j-1]+l]) l++;
    lcp[j]=l;
    if (l>lmax) lmax=l;
}
```

Algoritmo **no muy eficiente**: en el peor caso, podría evaluar $n(n+1)/2$ veces los tests de igualdades para incrementar l.

- Realmente, sale más fácil calcular el LCP con los sufijos **no ordenados** por el orden lexicografico pero por su orden original, el de la cadena!
- Observar que parece haber alguna estructura interesante en el caso de los sufijos en "orden original": para la cadena BLABLABLO, tenemos:

```
ABLABLO 0
ABLO 3
BLABLABLO 0
BLABLO 5
BLO 2
LABLABLO 0
LABLO 4
LO 1
O 0
```

mientras

```
BLABLABLO 0
ABLABLO 0
BLABLO 5
LABLO 4
ABLO 3
BLO 2
LO 1
O 0
```

Definimos entonces a:
`plcp[suffarray[j]] = lcp[j]`
La idea va a ser:

- construir plcp directamente
- usarle para deducir lcp.

Dos resultados importantes que usar:

- el número total de incrementos/decrementos es O(n) (no probado aquí)
- `plcp[i] >= plcp[i − 1] − 1`. Por qué?

Si tengo una sub-cadena s_i cuya cadena precedente en el orden lexicografico s_{ϕ_i} incluye k letras en comun en su prefijo, entonces, la sub-cadena s_{i+1} (tal que $s_i = Cs_{i+1}$) **comparte, en el peor caso, los $k-1$ primeras letras de la reducción (por su primer caracter) de s_{ϕ_i}.**

En un caso normal, podrá haber cadenas más cercanas aun de ella, compartiendo mas caracteres.

```
GATAGACA 2
ATAGACA 1
TAGACA 0
AGACA 1
GACA 0
ACA 1
CA 0
A -1
```

Pensando en el algoritmo ingenuo: podemos hacer **algo similar**, pero apoyandonos en el hecho de que **si s_i tiene `plcp[i] = k`, con `k>0` entonces no necesitamos comparar los `k-1` primeros caracteres de s_{i+1}, sino podemos empezar a partir del k-esimo**.

Ganamos mucho cuando hay partes en común!

2.3.1 Implementación

1. Calcular la tabla de correspondencias de indices entre los del orden original y sus precedentes en el lexicografico ($\phi[i]$).

```
phi[suffarray[0]] = -1; // The smallest one has no antecedent
for (int i = 1; i < s.size(); i++) // Orden lexicografixo
    phi[suffarray[i]] = suffarray[i-1];
```

2. Considerar $i = 0$: dado $\phi[0]$, se deducen los dos sufijos s_0 y $s_{\phi[0]}$. Se empieza por contar con cuantos caracteres consecutivos iguales estas dos sub-cadenas empiezan.

3. Para $i > 0$, aplico el truco descrito anteriormente: si en el paso anterior, encontre k caracteres en comunes, puedo dar por hecho que los $k-1$ primeros caracteres de s_i estarán en común con $s_{\phi[i]}$.

```
phi[suffarray[0]] = -1;
for (int i = 1; i < s.size(); i++) // Orden lexicografixo
    phi[suffarray[i]] = suffarray[i-1];

int k=0;
for (int i=0;i<n;i++) {
    if (phi[i] == -1) { plcp[i] = 0; continue; }
    while (s[i+k]==s[phi[i]+k]) k++; // Seguimos con k!
    plcp[i] = k;
    k = max(k-1, 0); // Aqui el truco!
}
// Ahora, rellenamos lcp
for (int i = 0;i<n;i++)
    lcp[i] = plcp[suffarray[i]];
```

Complejidad resultante **lineal**: $O(n)$!

2.4 Uso de los suffix arrays: Mayor sub-cadena repetida

Una vez que la tabla lcp está calculada, el problema es trivial!

Memorizar el **k máximo** en el calculo de lcp.

En el ejemplo anterior:

```
GATAGACA 2
ATAGACA 1
TAGACA 0
AGACA 1
GACA 0
ACA 1
CA 0
A 0
```

La mayor sub-cadena repetida es de **tamaño 2**, corresponde al index 6 de suffarray (orden lexicografixo). El sufijo anterior comparte este prefijo ("GA"), mientras que el siguiente no.

```
if (kmax<=0) {
    cout << "No repetitions found!" << endl;
    continue;
}
int count = 1;
for (int i = 0; i < s.size(); i++) {
    if (lcp[i]==kmax) {
        count++;
    } else if (count>=2) {
        break;
    }
}
```

UVA 11512

The Institute of Bioinformatics and Medicine (IBM) of your country has been studying the DNA sequences of several organisms, including the human one. Before analyzing the DNA of an organism, the investigators must extract the DNA from the cells of the organism and decode it with a process called “sequencing”.

A technique used to decode a DNA sequence is the “shotgun sequencing”. This technique is a method applied to decode long DNA strands by cutting randomly many copies of the same strand to generate smaller fragments, which are sequenced reading the DNA bases (A, C, G and T) with a special machine, and re-assembled together using a special algorithm to build the entire sequence. Normally, a DNA strand has many segments that repeat two or more times over the sequence (these segments are called “repetitions”). The repetitions are not completely identified by the shotgun method because the re-assembling process is not able to differentiate two identical fragments that are substrings of two distinct repetitions. The scientists of the institute decoded successfully the DNA sequences of numerous bacterias from the same family, with other method of sequencing (much more expensive than the shotgun process) that avoids the problem of repetitions. The biologists wonder if it was a waste of money the application of the other method because they believe there is not any large repeated fragment in the DNA of the bacterias of the

family studied. The biologists contacted you to write a program that, given a DNA strand, finds the largest substring that is repeated two or more times in the sequence.

Input: The first line of the input contains an integer T specifying the number of test cases ($1 \leq T \leq 100$). Each test case consists of a single line of text that represents a DNA sequence S of length n ($1 \leq n \leq 1000$). You can suppose that each sequence S only contains the letters 'A', 'C', 'G' and 'T'.

Output: For each sequence in the input, print a single line specifying the largest substring of S that appears two or more times repeated in S, followed by a space, and the number of ocurrences of the substring in S. If there are two or more substrings of maximal length that are repeated, you must choose the least according to the lexicographic order. If there is no repetition in S, print 'No repetitions found!'

```
#include <iostream>
#include <algorithm>
#include <string.h>
#include <math.h>
using namespace std;
#define SIZEMAX 1100

int freqs[SIZEMAX];
int suffarray[SIZEMAX];
int rankarray[SIZEMAX];
int tmprankarray[SIZEMAX];
int tmpsuffarray[SIZEMAX];
int phi[SIZEMAX];
int lcp[SIZEMAX];
int plcp[SIZEMAX];
string s;
int n;
// This function takes all
void radixSort(int k) {
    int m=max(300,n);
    memset(&freqs[0],0,sizeof(freqs));
    for (int i=0;i<n;i++)
        if (i+k<n)
            freqs[rankarray[i+k]]++;
        else
            freqs[0]++;
    // Accumulate all the frequencies
    int sfs=0;
    for (int i = 0; i < m; i++) {
        int freq = freqs[i]; freqs[i] = sfs; sfs += freq;
    }
    // Deduce the location of the data in the sorted array
    for (int i=0;i<n;i++)
        if (suffarray[i]+k<n)
            tmpsuffarray[freqs[rankarray[suffarray[i]+k]]++] = suffarray[i];
        else
```

```
            tmpsuffarray[freqs[0]++] = suffarray[i];
    // Copy back the sorted data
    for (int i=0;i<n;i++)
        suffarray[i] = tmpsuffarray[i];
}

void buildSuffArray() {
    for (int i=0;i<n;i++) {
        rankarray[i] = s[i];
    }
    // Init the suffix array
    for (int i=0;i<n;i++)
        suffarray[i]=i;
    // Does this log n times
    for (unsigned int k=1;k<n;k<<=1) {
        // Sort according to k-1th element
        radixSort(k);
        // Sort according to first element (this is a stable sort!)
        radixSort(0);
        // Re-generate the ranking based on the sorting
        // The smallest element gets the rank 0
        int r=0;
        tmprankarray[suffarray[0]] = r;
        for (int i=1;i<n;i++)
            if (rankarray[suffarray[i]] != rankarray[suffarray[i-1]] ||
                (suffarray[i]+k<n && rankarray[suffarray[i]+k] !=rankarray[sufarray[i-1]+k])) {
                    tmprankarray[suffarray[i]] = ++r;
                }
            else
                tmprankarray[suffarray[i]] = r;
        for (int i=0;i<n;i++) {
            // update the rank array RA
            rankarray[i] = tmprankarray[i];
        }
    }
}

void buildLCP() {
    phi[suffarray[0]] = -1;
    for (int i = 1;i<n;i++)
        phi[suffarray[i]] = suffarray[i-1];
    int l=0;
    for (int i=0;i<n;i++) {
        if (phi[i] == -1) { plcp[i] = 0; continue; }
        while (s[i+l]==s[phi[i]+l]) l++;
        plcp[i] = l;
        l = max(l-1, 0);
    }
```

```
    for (int i = 0; i < n; i++)
        lcp[i] = plcp[suffarray[i]];
}

pair<int,int> lrs() {
    int ind = 0, lcpmax = -1;
    for (int i = 1; i < n; i++)
    if (lcp[i] > lcpmax) {
        lcpmax = lcp[i];
        ind = i;
    }
    return make_pair(lcpmax, ind);
}

int main() {
    ios::sync_with_stdio(false);
    int nCases;
    cin >> nCases;
    getline(cin,s);
    for (int i=0;i<nCases;i++) {
        getline(cin,s);
        s.push_back('$');
        n = s.size();
        buildSuffArray();
        buildLCP();
        pair<int,int> ans = lrs();
        if (ans.first<=0) {
            cout << "No repetitions found!" << endl;
            continue;
        }
        int count = 1;
        for (int i = 0; i < n; i++) {
            // O(n), start fro
            if (lcp[i]== ans.first) {
                count++;
            } else if (count>=2) {
                break;
            }
        }
        cout << s.substr(suffarray[ans.second],ans.first) << " " << count << endl;
    }
    return 0;
}
```

2.4.1 Uso de los suffix arrays: Mayor sub-cadena común a dos cadenas

Acordarse de **como se proponía resolver este problema con un suffix tree**.

Cómo podríamos hacer algo similar aqui?
Idea: formar una **cadena compuesta**,

$$s_1 + \text{“.”} + s_2$$

y construir el arreglo de suffijos correspondiente.
Luego:

- marcar todos los sufijos por su "cadena proprietaria" (comparando por ejemplo suffarray[i] con s1.size())
- calcular el **lcp de la cadena conjunta**.
- detectar los casos de prefijos comunes entre sufijos consecutivos de proprietario distinto y lcp[i]>0.
- guardar de esa manera el prefijo común mayor.

Ejemplo: STEVEN vs. SEVEN
Formamos la cadena STEVEN.SEVEN

```
.SEVEN        * 2
EN            0 2
EN.SEVEN      2 1
EVEN          1 2
EVEN.SEVEN    4 1
N             0 2
N.SEVEN       1 1
SEVEN         0 2
STEVEN.SEVEN  1 1
TEVEN.SEVEN   0 1
VEN           0 1
VEN.SEVEN     3 2
```

```
int owner(int i) { return (i < s.size()-s2.size()-1) ? 1 : 2; }

pair<int,int> lcs() {
    int lcpind = 0, lcpmax = -1;
    for (int i = 1; i < s.size(); i++)
        if (owner(suffarray[i]) != owner(suffarray[i-1]) && lcp[i] > lcpmax) {
            lcpmax = lcp[i];
            lcpind = i;
        }
    return make_pair(lcpmax, lcpind);
}
```

UVA 719

Once upon a time there was a famous actress. As you may expect, she played mostly Antique Comedies most of all. All the people loved her. But she was not interested in the crowds. Her big hobby were beads of any kind. Many bead makers were working for her and they manufactured new necklaces and bracelets every day. One day she called her main Inspector of Bead Makers (IBM) and told him she wanted a very long and special necklace.

The necklace should be made of glass beads of different sizes connected to each other but without any thread running through the beads, so that means the beads can be disconnected at any point. The actress chose the succession of beads she wants to have and the IBM promised to make the necklace. But then he realized a problem. The joint between two neighbouring beads is not very robust so it is possible that the necklace will get torn by its own weight. The situation becomes even worse when the necklace is disjoined. Moreover, the point of disconnection is very important. If there are small beads at the beginning, the possibility of tearing is much higher than if there were large beads. IBM wants to test the robustness of a necklace so he needs a program that will be able to determine the worst possible point of disjoining the beads.

The description of the necklace is a string A = a1 a2 ... am specifying sizes of the particular beads, where the last character am is considered to precede character a1 in circular fashion. The disjoint point i is said to be worse than the disjoint point j if and only if the string a i a i+1 . . . a n a 1 . . . a i−1 is lexicografically smaller than the string a j a j+1 . . . a n a 1 . . . a j−1 . String a 1 a 2 . . . a n is lexicografically smaller than the string b 1 b 2 . . . b n if and only if there exists an integer i, $i \leq n$, so that aj=bj, for each j, $1 \leq j < i$ and ai < bi.

Input: The input consists of N cases. The first line of the input contains only positive integer N. Then follow the cases. Each case consists of exactly one line containing necklace description. Maximal length of each description is 10000 characters. Each bead is represented by a lower-case character of the english alphabet (a-z), where $a < b \cdots < z$.

Output: For each case, print exactly one line containing only one integer - number of the bead which is the first at the worst possible disjoining, i.e. such i, that the string A[i] is lexicographically smallest among all the n possible disjoinings of a necklace. If there are more than one solution, print the one with the lowest i.