Hashing sobre strings

November 23, 2019

1 Hashing

Un **metodo genérico** para detectar la identidad entre objetos, y que se puede usar en el contexto de cadenas de caracteres, es el **hashing**: en lugar de usar un algoritmo más o menos complejo para comparar los sub-elementos del objeto (los caracteres), mapeamos los objetos a $\mathbb{N}$ (por una **función de hash** h) y comparamos los valores obtenidos.

De esa manera, suponiendo que la función de hash h esté evaluada sobre dos cadenas s_1 y s_2, el costo de la cmparación baja de $O(\min(n_1, n_2))$ en $O(1)$.

Para la conversión a $\mathbb{N}$, se usa una **función de hash** $h()$.

Esta función tiene que cumplir ciertos criterios:

- ser **determinística** y en particular tal que si dos cadenas de caracteres son iguales, entonces la evaluacion de esas cadenas por la funcion de hash da el mismo resultado.
- ser calculable de manera **eficiente**.
- dado el conjunto de todas las cadenas de caracteres posibles (las de longitud L son 26^L), **repartir los valores de hash de la manera mas uniforme posible** entre, por ejemplo, los int's.

En el otro sentido, podrá ser que $h(s) = h(t)$ para $s \neq t$, porque en general **no podremos asegurar la inyectividad de la función de hash** (nuestro espacio de imagen es mucho más chico que el espacio de los valores de hash).

Cuando $h(s) = h(t)$ para $s \neq t$ hablamos de **colisión**.

Comparar dos cadenas en base a su valor de hash en principio no implicará poder decidir con certeza que son iguales o diferentes pero:

- si disenamos bien nuestra funcion de hash y terminamos mapeando nuestras cadenas de manera uniforme en el intervalo $[0, m-1]$, tendremos una **probabilidad de colisión** de

$$\frac{1}{m}$$

- si el valor de hash es distinto entre dos cadenas entonces **podemos descartar igualdad**; si el valor de hash es igual, podemos **verificar de manera más exhaustiva** (y costosa) la igualdad, e.g. caracter por caracter.

1.1 Funciones de hash para cadenas de caracteres

Para **cadenas de carácteres**, una función de hash extremadamente usada es la **polinomial**:

$$\begin{aligned} h(s) &= s[0] + s[1]p + s[2]p^2 + ... + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i]p^i \mod m, \end{aligned}$$

donde p, m son parametros de la función.

Supone una **conversión de cada caracter a entero** (tipicamente, restando 'a').

Tipicamente, p **es del orden de magnitud del rango de caracteres** (e.g. para evitar colisiones al usar caracteres iguales modulo el p).

Valores de p muy usados: $31, 33, 53$.

- Se usan muchos números **primos**, pero no hay prueba de que la función polinomial es mejor por eso.
- Para números compositos como 33, se tiene buenos resultados.
- Hay elecciones claramente malas: 1 por ejemplo.

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4045622

```
The table below summarizes the performance of the various hash functions described above, for th

1) All of the words and phrases with entries in Merriam-Webster's 2nd Int'l Unabridged Dictionar

2) All of the strings in /bin/*, /usr/bin/*, /usr/lib/*, /usr/ucb/* and /usr/openwin/bin/*  (66,

3) A list of URLs gathered by a web-crawler that ran for several hours last night (28,372 string

The performance metric shown in the table is the "average chain size" over all elements in the h

                    Webster's   Code Strings    URLs
                    ---------       ------------    ----
P(37)    [Java]      1.2508     1.2481      1.2454
P(65599) [Aho et al] 1.2490     1.2510      1.2450
P(31)    [K+R]       1.2500     1.2488      1.2425
P(33)    [Torek]     1.2500     1.2500      1.2453
```

Por otra parte m tiene que ser un **número grande** (probabilidad de colisión en $\frac{1}{m}$).

```cpp
long long hash(const string& s) {
    const int p = 31;
    const int m = 1000000007;
    long long hashed = 0;
    long long pi     = 1;
    for (int i=0;i<s.size();i++) {
        hashed = (hashed + (s[i] - 'a' + 1) * pi) % m;
        pi     = (pi * p) % m;
    }
    return hashed;
}
```

Mejoras?

Estamos evaluando un polinomio: podemos ser más eficiente y no evaluar todas las potencias de p **pasando por el método de Horner**.

```
long long hash(const string& s) {
    const int p = 31;
    const int m = 1000000007;
    long long hashed = 0;
    for (int i=s.size()-1;i>=0;i--) {
        hashed = (hashed*p + (s[i] - 'a' + 1)) % m;
    }
    return hashed;
}
```

Complejidad en $O(n)$ donde n es el tamano de la cadena.

1.1.1 Otros ejemplos de funciones de hash

- Java es un lenguaje donde la noción de función de hash es central.
- Consultar:

  http://www.cse.yorku.ca/~oz/hash.html

```
// djb2
// this algorithm (k=33) was first reported by dan bernstein many // years ago in comp.lang.c. a
unsigned long hash(unsigned char *str) {
    unsigned long hash = 5381;
    int c;
    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
    return hash;
}
```

Observar el uso de los shifts! Interesante computacionalmente usar un número cerca de una potencia de 2.

```
/*
  This algorithm was created for sdbm (a public-domain reimplementation of ndbm) database librar
  */
static unsigned long hash(unsigned char *str) {
    unsigned long hash = 0;
    int c;
    while (c = *str++)
      hash = c + (hash << 6) + (hash << 16) - hash;
    return hash;
}

/* This hash function appeared in K&R (1st ed) but at least the reader was warned: "This is not
    unsigned long hash(unsigned char *str) {
```

```
        unsigned int hash = 0;
        int c;

        while (c = *str++)
            hash += c;

        return hash;
    }
```

1.2 Aplicación: cadenas duplicadas en un arreglo de cadenas

Dado un arreglo de N cadenas de caracteres, identificar todas las cadenas duplicadas y agruparlas en clusters de cadenas iguales.

Un algoritmo ingenuo sería: ordenar los strings en base al orden lexicografico y recorrer el arreglo ordenado para detectar los intervalos con cadenas de mismo valor.

Si la dimensión de las cadenas es $O(n)$, entonces, el costo total sería de

$$O(nN \log N).$$

Pasando por función de hash, reducimos este costo a $O(N \log N)$.

```
vector<vector<int>> clusterIdenticalStrings(const vector<string> &arr) {
    vector<pair<long long, int>> hashValues(arr.size());
    for (int i = 0; i < arr.size(); i++)
        hashValues[i] = {hash(arr[i]), i};

    // Sort the hash values
    sort(hashValues.begin(), hashValues.end());

    vector<vector<int>> clusters;
    for (int i = 0; i < arr.size(); i++) {
        // We create a new group
        if (i == 0 || hashValues[i].first != hashValues[i-1].first)
            clusters.push_back(vector<int>());
        // We put the string index in the current group
        clusters.back().push_back(hashValues[i].second);
    }
    return clusters;
}
```

1.3 Aplicación: Buscar una cadena en otra

Suponemos que s_2 se busca dentro de s_1 ($n_1 > n_2$). Podríamos usar el siguiente algoritmo:

1. Calcular $h(s_2)$. Costo en $O(n_2)$.
2. Calcular $h()$ en todas las sub-cadenas de S de longitud n_2, y comparar cada valor de hash con $h(s_2)$. Costo en $O(n_1 n_2)$.
3. Si una sub-cadena tiene un hash igual a $h(s_2)$, hacer una comparación caracter por caracter. Costo en $O(cn_2)$ donde c es el número de occurencias.

El segundo paso es el más pesado e intuitivamente, **debe de poder mejorar!**. Muchos calculos parecen estar presentes en los diferentes valores de $h()$ sobre las diferentes sub-cadenas.

Tenemos:

$$h(s_1[k, k+n_2-1]) \quad = \quad \sum_{i=0}^{n_2-1} s_1[k+i]p^i \mod m,$$

entonces

$$\begin{aligned} h(s_1[k-1, k+n_2-2]) &= \sum_{i=0}^{n_2-1} s_1[k-1+i]p^i \mod m, \\ &= s_1[k-1] + \sum_{i=1}^{n_2-1} s_1[k-1+i]p^i \mod m \\ &= s_1[k-1] + p(\sum_{i=1}^{n_2-1} s_1[k-1+i]p^{i-1}) \mod m \\ &= s_1[k-1] + p(\sum_{j=0}^{n_2-2} s_1[k+j]p^j) \mod m \\ &= s_1[k-1] + p(h(s_1[k, k+n_2-1]) - s_1[k+n_2-1]p^{n_2-1}) \mod m \end{aligned}$$

Podemos entonces **calcular cada término en $O(1)$ a partir del de su vecino**.

Técnica llamada **rolling hash**: usa nada más dos caracteres en cada evaluación (inicio y final de la sub-cadena).

Hace que el costo total del paso 2 en el algoritmo arriba se reduce a $O(n_2 + n_1)$.

1. Calcular $h(s_2)$. Costo en $O(n_2)$.
2. Calcular $h()$ en la primera sub-cadena de S de longitud n_2. Costo en $O(n_2)$.
3. Calcular todas las sub-cadenas siguientes de S de longitud n_2 con el rolling hash, y comparar cada valor de hash con $h(s_2)$. Costo en $O(n_1)$.
4. Si una sub-cadena tiene un hash igual a $h(s_2)$, hacer una comparación caracter por caracter. Costo en $O(cn_2)$ donde c es el número de occurencias.

- Eso supone que la función de hash da $O(1)$ colisiones (caso **ideal**)
- Sino tendremos mas verificaciones que hacer en el paso 4 (falsos positivos).
- En práctica, el número esperado de colisiones es $O(\frac{n_1}{m})$, por lo que el costo total para este caso se modificaría en

$$O((c + \frac{n_1}{m})n_2).$$

1.4 Hash para sub-cadenas

Nos damos una cadena s, dos índices i, j, y queremos ahora encontrar el **hash de la sub-cadena** $s[i, j]$.

Tenemos:

$$h(s[i, j]) = \sum_{k=i}^{j} s[k]p^{k-i} \mod m.$$

Entonces al multiplicar por p^i:

$$\begin{aligned} h(s[i, j])p^i \mod m &= \sum_{k=i}^{j} s[k]p^{k-i} \mod m \\ &= h(s[0, j]) - h(s[0, i-1]) \mod m. \end{aligned}$$

Entonces si **pre-calculamos los valores de hash de todos los prefijos de una cadena** s (en $O(n^2)$ en implementación ingenua, pero mejorable en $O(n)$ al usar el mismo truco de arriba), podemos luego aceder **a los valores de hash de cualquiera sub-cadena** (en realidad, un multiple de ella).

Qué hacer de la división por p^i?

En realidad, no es un problema porque tipicamente, los valores de hash sólo se comparan entre sí. Entonces:

- Si comparamos una sub-cadena $s[i, j]$ contra una cadena test t, basta comparar

$$h(t)p^i \mod m \text{ y } h(s[0, j]) - h(s[0, i-1]) \mod m$$

- Si comparamos dos sub-cadenas $s[i, j]$ y $t[k, l]$, con $i < k$, basta comparar

$$(h(s[0, j]) - h(s[0, i-1]))p^{k-i} \mod m \text{ y } (h(t[0, l]) - h(t[0, k-1])) \mod m.$$

2 Aplicación: Algoritmo de Rabin-Karp

Es el problema de matching: Dadas dos cadenas, s, donde se va a buscar, y $pattern$, determinar todas las occurencias de $pattern$ en s.

Paso 1:

- Calcular el valor hash **para el patrón pattern**.
- Calcular los valores hash para **todos los prefijos del texto s**:

$$s[0, i].$$

Paso 2:

- Usar los prefijos anteriores para evaluar el multiple de hash de todas las sub-cadenas de tamano el tamano del patrón.
- Usar esas valores de hash para comparacion con el valor de hash de patron; en caso de igualdad comparar caracter por caracter o agregar directamente la sub-cadena a la lista.

Complejidad en $O(|pattern| + |s|)$: * O(|s|) para el precalculo de los hash de los prefijos. * O(|pattern|) para el hash del patrón. * O(|s|) para las comparaciones de hash.

Una vez preprocedada s

$$O(|pattern| + |s|).$$

```
vector<int> rabinKarpMatch(const string& s, const string& pattern) {
    const int p = 31;
    const int m = 1000000007;

    // Precompute modular powers
    vector<long long> pi(s.size());
    pi[0] = 1;
    for (int i = 1; i < s.size(); i++)
```

```
        pi[i] = (pi[i-1] * p) % m;

    // Precompute the hash value for all the prefixes of s
    // Does not depend on the pattern
    vector<long long> hashed_prefixes(s.size() + 1, 0);
    for (int i = 0; i < s.size(); i++)
        hashed_prefixes[i+1] = (hashed_prefixes[i] + (s[i] - 'a' + 1) * pi[i]) % m;

    // Compute the hash value for pattern
    long long hashed_pattern = 0;
    for (int i = 0; i < pattern.size(); i++)
        hashed_pattern = (hashed_pattern + (pattern[i] - 'a' + 1) * pi[i]) % m;

    // Compare sub-string hashes
    vector<int> found;
    for (int i = 0; i + pattern.size() - 1 < s.size(); i++) {
        long long hashed = (hashed_prefixes[i+pattern.size()] + m - hashed_prefixes[i]) % m;
        if (hashed == hashed_pattern * pi[i] % m)
            found.push_back(i);
    }
    return found;
}
```

2.1 Aplicación: sub-cadenas distintas en una cadena

Dada una cadena de caracteres de tamano n, determinar el numero de sub-cadenas **distinta** en esta cadena.

Con hashing: construir **un set de todos los valores de hash** (o más bien de un multiplicador de ellas).

- Recorrer las sub-cadenas por longitud y punto de inicio en la cadena.
- Para cada una evaluar el valor de hash.
- Multiplicarle por una potencia de p para asegurarse de que todos los valores obtenidos corresponden a un mismo multiplicador del valor de hash.
- Contar el número de elementos en el set.

```
vector<int> uniqueSubStrings(const string& s) {
    const int p = 31;
    const int m = 1000000007;

    // Precompute modular powers
    vector<long long> pi(s.size());
    pi[0] = 1;
    for (int i = 1; i < s.size(); i++)
        pi[i] = (pi[i-1] * p) % m;

    // Precompute the hash value for all the prefixes of s
    // Does not depend on the pattern
```

```
    vector<long long> hashed_prefixes(s.size() + 1, 0);
    for (int i = 0; i < s.size(); i++)
        hashed_prefixes[i+1] = (hashed_prefixes[i] + (s[i] - 'a' + 1) * pi[i]) % m;

    int distinct = 0;

    // Cycle over substring lengths
    for (int l = 1; l <= n; l++) {
        set<long long> allHashed;
        // Cycle over starting positions
        for (int i = 0; i <= n - l; i++) {
            long long hashed = (hashed_prefixes[i + l] + m - hashed_prefixes[i]) % m;
            // To compare hashed values, set them all to power p^{n-1}
            // As the difference above is the hash multiplies by p^i
            // we multiply by p^{n-1-i}
            hashed = (hashed * pi[n-i-1]) % m;
            allHashed.insert(hashed);
        }
        distinct += allHashed.size();
    }
    return distinct;
}
```

Complejidad?

- $O(n^2 \log n)$

Con el arbol de sufijos?

2.2 Colisiones

Para cada problema, evaluar la probabilidad de tener al menos una colisión.

Con $m = 10^9$, probabilidad de tener colisión con dos cadenas de $\pi = 10^{-9}$.

Al considerar mucho más cadenas (K) en un problema de tipo matching, esa probabilidad puede subir mucho!

$$1 - (1 - \pi)^K \approx K\pi$$

Si la probabilidad de colisión se hace muy alta: una solución barata es **calcular dos funciones de hash** (con diferentes valores de p).

2.3 Ejemplos de problemas

Codeforces Beta Round 7D

String s of length n is called k-palindrome, if it is a palindrome itself, and its prefix and suffix of length are (k - 1)-palindromes. By definition, any string (even empty) is 0-palindrome.

Let's call the palindrome degree of string s such a maximum number k, for which s is k-palindrome. For example, "abaaba" has degree equals to 3.

You are given a string. Your task is to find the sum of the palindrome degrees of all its prefixes.

Input: The first line of the input data contains a non-empty string, consisting of Latin letters and digits. The length of the string does not exceed 5·106. The string is case-sensitive.

Output: Output the only number — the sum of the polindrome degrees of all the string's prefixes.

Dificil de hacer la verificación caracter por caracter, prefijo por prefijo ($O(n^2)$).

La definición del grado es **recursiva**: el grado palindromico del prefijo $[0, i]$ se deduce facilmente del de $[0, i/2]$: * Si $[0, i]$ es un palindromo entonces su grado es 1 plus el grado de $[0, i/2]$ * Sino **es cero**.

Lo último que nos falta es evaluar **rapidamente cuando un prefijo es palindrom**. Una opción rápida con hashing:

- evaluar el hash del prefijo con su función de hash polinomial preferida.
- evaluar al mismo tiempo el hash de la cadena reversa (**aplicando el método de Horner**).

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
const int p = 31;
const int m = 1000000007;
const int sizemax = 5000001;

int main() {
  // Precompute modular powers
  vector<long long> pi(sizemax);
  pi[1] = 1;
  for (int i = 2; i < sizemax; i++)
    pi[i] = (pi[i-1] * p) % m;

  string s;
  cin >> s;
  s.insert(0,1,'0'); // To make characters start at 1
  long long sum            = 0;
  long long hashed         = 0;
  long long hashed_reverse = 0;
  vector<long long> pdegrees(sizemax,0);

  for(int i=1;i<s.size();i++) {
      // Prefix hash evaluated from left to right
      hashed        =hashed*p+s[i];
      // Prefix hash evaluated from right to left
      hashed_reverse=s[i]*pi[i]+hashed_reverse;
      // Is this a palindrome?
      if (hashed==hashed_reverse)
        pdegrees[i]=pdegrees[i/2]+1;
      sum += pdegrees[i];
  }
  cout<<sum<<endl;
}
```

Codeforces 752D

Santa Claus likes palindromes very much. There was his birthday recently. k of his friends came to him to congratulate him, and each of them presented to him a string si having the same length n. We denote the beauty of the i-th string by ai. It can happen that ai is negative — that means that Santa doesn't find this string beautiful at all.

Santa Claus is crazy about palindromes. He is thinking about the following question: what is the maximum possible total beauty of a palindrome which can be obtained by concatenating some (possibly all) of the strings he has? Each present can be used at most once. Note that all strings have the same length n.

Recall that a palindrome is a string that doesn't change after one reverses it.

Since the empty string is a palindrome too, the answer can't be negative. Even if all ai's are negative, Santa can obtain the empty string.

Input: The first line contains two positive integers k and n divided by space and denoting the number of Santa friends and the length of every string they've presented, respectively ($1 \leq k, n \leq 100\,000$; $n \cdot k \leq 100\,000$).

k lines follow. The i-th of them contains the string si and its beauty ai ($-10\,000 \leq ai \leq 10\,000$). The string consists of n lowercase English letters, and its beauty is integer. Some of strings may coincide. Also, equal strings can have different beauties.

Output: In the only line print the required maximum possible beauty.

Observar que, porque todas las cadenas **tienen el mismo tamano**, la estructura del palindromo final puede ser:

- con un número **par** de cadenas y tal que las sub-cadenas de tamano *n* incluidas sean simétricas (en posición y en contenido).
- con un número **impar** de cadenas, y con la misma propriedad anterior, excepto por el elemento central, que **tiene que ser palindromo**.
- Mientras se pueda, agregar **pares de cadenas simétricas** (usar el **hash** para detectar eso!), con los scores máximos y mientras la suma de las dos cadenas sea positiva (sino no vale la pena).
- Entre las cadenas que quedan, eventualmente podemos agregar un **elemento-palindromo como elemento central** (mientras su score sea positivo).
- Un caso que puede hacer corregir los pares entradas previamente: si se ha puesto pares de 2 instancias de palindromo con un valor negativo en uno de los elementos, entonces, **buscamos entre esos pares el con el valor positivo mayor y le ponemos como elemento central**.

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <algorithm>

using namespace std;
const int p = 31;
const int m = 1000000007;
```

```
const int sizemax = 100001;

vector<long long> pi(sizemax);
map<long long,vector<int>> scores;
map<long long,long long> reverses;
string s;

std::pair<long long,long long> hashString(const string& s) {
    const int p = 31;
    const int m = 1000000007;
    long long hashed         = 0;
    long long hashed_reverse = 0;

    for (int i=1;i<s.size();i++) {
      // Hash evaluated from right to left
      hashed_reverse  = hashed_reverse*p+s[i];
      // Hash evaluated from left to right
      hashed          = s[i]*pi[i]+hashed;
    }
    return std::make_pair(hashed,hashed_reverse);
}

int main() {
  // Precompute modular powers
  pi[1] = 1;
  for (int i = 2; i < sizemax; i++)
    pi[i] = (pi[i-1] * p) % m;
  int k,n,score,totalScore=0;
  cin >> k >> n;

  // Get the strings and build a map indexed by the hash value
  vector<string> allStrings(k);
  for (int i = 0; i < k; i++) {
        cin >> s >> score;
        s.insert(0,1,'0');
        pair<long long, long long> p = hashString(s);
        scores[p.first].push_back(score);
        reverses[p.first]=p.second;
    }
    // Sort all the scores vectors
    auto it = scores.begin();
    for (auto it = scores.begin();it != scores.end();it++) {
        sort(it->second.begin(), it->second.end());
    }

  // Try to select candidate pairs
  int minNegative = 0;
  for (auto it = scores.begin();it != scores.end();it++) {
```

```
    // String is a palindrome? Compare hash and reverse hash
    if (it->first != reverses[it->first]) {
      // Search for the symmetric
            auto sym = scores.find(reverses[it->first]);
            if (sym != scores.end()) {
                while (it->second.size() && sym->second.size()) {
                    if (it->second.back() + sym->second.back() > 0) {
                        totalScore += it->second.back() + sym->second.back();
                        it ->second.pop_back();
            sym->second.pop_back();
                    }
                    else break;
                }
            }
        }
    // String is a palindrome
    else {
            while (it->second.size() >= 2) {
                if (it->second.back() + it->second[it->second.size() - 2] >= 0) {//
                    totalScore += it->second.back() + it->second[it->second.size() - 2];
                    it->second.pop_back();
          // Case where we may consider the first element as the central one
          // (that would mean not including the second one)
                    if (it->second.back() < 0)
            minNegative = min(minNegative, it->second.back());
                    it->second.pop_back();
                }
                else break;
            }
        }
    }
  // Determine the max. single palindrome
    int maxSinglePalindrome = 0;
  for (auto it = scores.begin();it != scores.end();it++) {
    // String is a palindrome? Compare hash and reverse hash
    if (it->first == reverses[it->first]) {
            if (it->second.size() && it->second.back() > 0) {
                maxSinglePalindrome = max(maxSinglePalindrome, it->second.back());
            }
        }
    }
    maxSinglePalindrome = max(maxSinglePalindrome, -minNegative);
    totalScore += maxSinglePalindrome;
    cout << totalScore << endl;
}
```

UVA 11855

The word "the" is the most common three-letter word. It even shows up inside other words,

such as “other” and “mathematics”. Sometimes it hides, split between two words, such as “not here”. Have you ever wondered what the most common words of lengths other than three are? Your task is the following. You will be given a text. In this text, find the most common word of length one. If there are multiple such words, any one will do. Then count how many times this most common word appears in the text.

If it appears more than once, output how many times it appears. Then repeat the process with words of length 2, 3, and so on, until you reach such a length that there is no longer any repeated word of that length in the text.

Input The input consists of a sequence of lines. The last line of input is empty and should not be processed. Each line of input other than the last contains at least one but no more than one thousand uppercase letters and spaces. The spaces are irrelevant and should be ignored.

Output For each line of input, output a sequence of lines, giving the number of repetitions of words of length 1, 2, 3, and so on. When you reach a length such that there are no repeated words of that length, output one blank line, do not output anything further for that input line, and move on to the next line of input. Note: Remember that the last line of the sample input and of the sample output must be blank.

Observar que la secuencia es necesariamente decreciente.

Cuando encontramos un tamano l para el cual no tenemos duplicados, **abandonamos**.

Usar hashes sobre **todas las sub-cadenas a partir de los prefijos** (estilo Rabin-Karp).

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <algorithm>
#include <cctype>

using namespace std;

int main() {
    const int p    = 31;
    const int m    = 1000000007;
    const int sMax = 1001;
    // Precompute modular powers
    vector<long long> pi(sMax);
    pi[0] = 1;
    for (int i = 1; i < sMax; i++)
        pi[i] = (pi[i-1] * p) % m;

    string s;
    while (1) {
      getline(cin,s);
      if (!s.size())
        break;
      s.erase(remove_if(s.begin(), s.end(), [](unsigned char c){ return std::isspace(c); }), s.e
      s.insert(0,1,'0');

      // Precompute the hash value for all the prefixes of s
```

```
      vector<long long> hashed_prefixes(s.size() + 1, 0);
      for (int i = 0; i < s.size(); i++)
        hashed_prefixes[i+1] = (hashed_prefixes[i] + (s[i] - 'a' + 1) * pi[i]) % m;

      // Cycle over substring lengths and get the maximal lengths of repeated
      vector<int> mostCommon;
      for (int l = 1; l < s.size(); l++) {
        map<long long,int> allHashed;
        int maxFrequency = 0;
        // Cycle over starting positions
        for (int i = 0; i <= s.size() - l; i++) {
            long long hashed = (hashed_prefixes[i + l] + m - hashed_prefixes[i]) % m;
            // To compare hashed values, set them all to power p^{n-1}
            // As the difference above is the hash multiplies by p^i
            // we multiply by p^{n-1-i}
            hashed = (hashed * pi[s.size()-i-1]) % m;
            allHashed[hashed]++;
            if (allHashed[hashed]>maxFrequency)
              maxFrequency = allHashed[hashed];
        }
        if (maxFrequency>1) {
          cout << maxFrequency << endl;
        } else {
          cout << endl;
          break;
        }

    }
  }
  return 0;
}
```

Codeforces Beta Round 271D

You've got string s, consisting of small English letters. Some of the English letters are good, the rest are bad.

A substring s[l...r] ($1 \le l \le r \le |s|$) of string s = s1s2...s|s| (where |s| is the length of string s) is string slsl + 1...sr.

The substring s[l...r] is good, if among the letters $s_l, s_{l+1}, ..., s_r$ there are at most k bad ones (look at the sample's explanation to understand it more clear).

Your task is to find the number of distinct good substrings of the given string s. Two substrings s[x...y] and s[p...q] are considered distinct if their content is different, i.e. $s[x...y] \neq s[p...q]$.

Input: The first line of the input is the non-empty string s, consisting of small English letters, the string's length is at most 1500 characters.

The second line of the input is the string of characters "0" and "1", the length is exactly 26 characters. If the i-th character of this string equals "1", then the i-th English letter is good, otherwise it's bad. That is, the first character of this string corresponds to letter "a", the second one corresponds to letter "b" and so on.

The third line of the input consists a single integer k ($0 \leq k \leq |s|$) — the maximum acceptable number of bad characters in a good substring.

Output: Print a single integer — the number of distinct good substrings of string s.

- Extension relativamente simple del **problema de sub-cadenas únicas** con el hash calculado a la manera de Rabin-Karp.
- La parte adicional es la de determinar si es good o no. Para eso, en un ciclo similar, contar el número de letras 'bad' de cada sub-cadena (con memoización).

```
// Determine the goodness first
// Cycle over starting positions
for (int i = 1; i <= s.size(); i++) {
  sum[i][0]=badness[s[i]-'a'];
  // Cycle over substring lengths
  for (int l = 1; i+l < s.size(); l++) {
    sum[i][l]=sum[i][l-1]+badness[s[i+l]-'a'];
  }
}

int distinct = 0;
// Cycle over substring lengths
for (int l = 1; l <=s.size(); l++) {
  set<long long> allHashed;
  // Cycle over starting positions
  for (int i = 0; i+l<=s.size(); i++) if (sum[i][l]<=k) {
      long long hashed = (hashed_prefixes[i + l] + m - hashed_prefixes[i]) % m;
      // To compare hashed values, set them all to power p^{n-1}
      // As the difference above is the hash multiplies by p^i
      // we multiply by p^{n-1-i}
      hashed = (hashed * pi[s.size()-i]) % m;
      allHashed.insert(hashed);
  }
  distinct += allHashed.size();
}
```

UVA 12012

E.T. Inc. employs Maryanna as alien signal researcher. To identify possible alien signals and background noise, she develops a method to evaluate the signals she has already received. The signal sent by E.T is more likely regularly alternative. Received signals can be presented by a string of small latin letters 'a' to 'z' whose length is N. For each X between 1 and N inclusive, she wants you to find out the maximum length of the substring which can be written as a concatenation of X same strings. For clarification, a substring is a consecutive part of the original string.

Input: The first line contains T, the number of test cases ($T \leq 200$). Most of the test cases are relatively small. T lines follow, each contains a string of only small latin letters 'a' - 'z', whose length N is less than 1000, without any leading or trailing whitespaces.

Output: For each test case, output a single line, which should begin with the case number counting from 1, followed by N integers. The X-th (1-based) of them should be the maximum length of the substring which can be written as a concatenation of X same strings. If that substring doesn't exist, output 0 instead. See the sample for more format details.

- Cada sub-cadena que tiene la propriedad mencionada es **formada por X sub-cadenas idénticas**.
- Una solución: detectar para cada sub-cadena si tiene vecinas idénticas (usar hash para eso).
- Contar esas cadenas idénticas y actualizar un arreglo indizado por el número de repeticiones.

```
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <algorithm>
#include <cctype>
#include <cstring>

using namespace std;

int main() {
    const int p    = 31;
    const int m    = 1000000007;
    const int sMax = 1001;
    // Precompute modular powers
    vector<long long> pi(sMax);
    pi[0] = 1;
    for (int i = 1; i < sMax; i++)
        pi[i] = (pi[i-1] * p) % m;
    int T;
    string s;
    cin >> T;
    for (int k=1;k<=T;k++) {
      cin >> s;
      if (!s.size())
        break;
      s.erase(remove_if(s.begin(), s.end(), [](unsigned char c){ return std::isspace(c); }), s.e
      s.insert(0,1,'0');

      // Precompute the hash value for all the prefixes of s
      vector<long long> hashed_prefixes(s.size() + 1, 0);
      for (int i = 0; i < s.size(); i++)
        hashed_prefixes[i+1] = (hashed_prefixes[i] + (s[i] - 'a' + 1) * pi[i]) % m;

      long long allHashed[sMax][sMax];
      // Cycle over starting positions
      for (int i = 0; i <= s.size(); i++) {
        // Cycle over lengths
        for (int l = 1; l+i <= s.size(); l++) {
            long long hashed = (hashed_prefixes[l+i] + m - hashed_prefixes[i]) % m;
            // To compare hashed values, set them all to power p^{n-1}
            // As the difference above is the hash multiplies by p^i
```

```
            // we multiply by p^{n-1-i}
            hashed = (hashed * pi[s.size()-i-1]) % m;
            allHashed[l][i]=hashed;
        }
    }

    // Detect if for a given length we have repetitions
    int tab[sMax];
    memset(&tab[0],0,sizeof(tab));
    for (int l = s.size()/2; l>=1; l--) {
      for (int i = 0; i+l <= s.size(); i++) {
        int nr = 0;
        while (i+nr*l <= s.size() && allHashed[l][i]==allHashed[l][i+nr*l]) {
          nr++;
          if (l*nr>tab[nr])
            tab[nr]=l*nr;
        }
      }
    }
    cout << "Case #" << k << ": "<< s.size()-1;
    for (int i = 2; i < s.size(); i++)
     cout << " " << tab[i];
    cout << endl;
  }
  return 0;
}
```