GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CANTIDAD DE OCURRENCIA DE UN PATRÓN SUBSECUENCIA DE ELEMENTOS

1. Introducción

Durante la lectura de problemas nos podemos encontrar algunos que su solución se reduce a encontrar dentro de una cadena la cantidad de ocurrencias de un determinado patrón como una subsecuencia de caracteres de no necesariamente consecutivos.

Digamos que tenemos como cadena en la que debemos buscar *subsequence* y como patrón a buscar la secuencia *sue*. Si buscamos el patrón *sue* dentro de la cadena como subsecuencia de caracteres no consecutivas encotramos que la respuesta es 7.

subs**e**quence

subsequ**e**nce

subsequenc**e**

subseq**ue**nce

subseq**ue**nc**e**

sub**s**eq**ue**nce

sub**s**eq**u**enc**e**

2. Conocimientos previos

2.1. Programación dinámica

En informática, la programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas. Una subestructura óptima significa que se pueden usar soluciones óptimas de subproblemas para encontrar la solución óptima del problema en su conjunto. Se pueden resolver problemas con subestructuras óptimas siguiendo estos tres pasos:

1. Dividir el problema en subproblemas más pequeños.
2. Resolver estos problemas de manera óptima usando este proceso de tres pasos recursivamente.
3. Usar estas soluciones óptimas para construir una solución óptima al problema original.

3. Desarrollo

Una idea para resolver este problema es utilizar la recursividad. Si comparamos los últimos carácteres de la cadena donde vamos a buscar(a partir de ahora vamos a nombrarla X) y del patrón que queremos encontrar (a partir de ahora la nombraremos Y) siendo m y n las longitudes de las cadenas X y Y respectivamente podemos encontrar dos posibilidades.

- Si el último cáracter de la cadena es el mismo que el último cáracter del patrón, recursamos para analizar ahora para las subcadenas X[0 ... m-1] y Y[0 ... n-1]. Como queremos hallar todas las posibles soluciones debemos considerar el caso que el último cáracter de la cadena no sea igual al último cáracter del patrón, en ese caso recursamos para analizar las subcadenas X[0 ... m-1] y Y[0 ... n].
- Si el último cáracter de la cadena no es el mismo que el último cáracter del patrón entonces recursamos para analizar las subcadenas X[0 ... m-1] y Y[0 ... n].

El único problema de esta solución es que su complejidad temporal es exponencial pero por el lado bueno su complejidad en cuanto a uso de memoria es *O(1)*.

La idea es utilizar a Programación Dinámica para solucionar este problema. El problema tiene una subestructura óptima. Por encima de la solución también exhibe subproblemas que se superponen . Si dibujamos el árbol del recursion de la solución, podemos ver que los mismos subproblemas son calculados una y otra vez.

Sabemos que los problemas que tienen subestructura óptima y los subproblemas implicados pueden ser solucionados usando programación dinámica, en cuál los subproblemas solucionados y memorizados en vez de calculados una y otra vez. La versión Memorización sigue el acercamiento de arriba a abajo, desde que primero quebrantamos el problema en los subproblemas y luego calculamos y almacenamos valores. También podemos solucionar este problema en la manera de abajo hacia arriba. En el acercamiento de abajo hacia arriba, solucionamos subproblema más pequeño primero, entonces solucionan mayores subproblemas de ellos.

4. Implementación

4.1. C++

```
long long count(string a, string b){
   int m= a.size();
   int n= b.size();
   long long look[m+1][n+1];
   for(int i=0;i<=n;i++) look[0][i]=0;

   for(int i=0;i<=m;i++) look[i][0]=1;

   for(int i=1;i<=m;i++){
      for(int j=1;j<=n;j++){
        if(a[i-1]==b[j-1]) look[i][j]=look[i-1][j-1]+ look[i-1][j];
        else look[i][j]=look[i-1][j];
      }
   }
   return look[m][n];
}
```

4.2. Java

```java
public long countSubsequence(String _text, String _pattern){
   int m=_text.length();
   int n=_pattern.length();
   long [][] dp =new long [m+10][n+10];

   for(int i=0;i<=n;++i)dp[0][i]=0;

   for(int i=0;i<=m;++i)dp[i][0]=1;

   for(int i=1;i<=m;i++){
      for(int j=1;j<=n;j++){
         if(_text.charAt(i-1) == _pattern.charAt(j-1))
            dp[i][j]=dp[i-1][j-1]+dp[i-1][j];
         else
            dp[i][j]=dp[i - 1][j];
      }
   }
   return dp[m][n];
}
```

5. Complejidad

Esta solución es que su complejidad temporal es *O(nm)* y su complejidad en cuanto a uso de memoria es *O(nm)*.

6. Aplicaciones

Esta claro la aplicación de este algoritmo. Aunque en la implementación se utilizo una cadena de caracteres se puede utilizar perfectamente una colección de otros tipos de datos.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se resuelven utilizando este algoritmo:

- DMOJ - Nerdson y Alejandra.