SEMINARIO DE PROGRAMACIÓN COMPETITIVA

ESTRUCTURA DE DATOS II

POR: LEONARDO SANTELLA

AGENDA

Introducción

Trie

- Idea general del algoritmo
- ¿Qué problema resuelve?
- Análisis de Complejidad en tiempo
- Observaciones
- Pseudocódigo del algoritmo
- Ejemplo
- Notas adicionales
- Ejercicios recomendados
- Preguntas/Dudas

Suffix Array

- Idea general del algoritmo
- Suffix Trie / Suffix Tree
- Qué problema resuelve?
- Análisis de Complejidad en tiempo
- Observaciones
- Pseudocódigo del algoritmo
- Ejemplo
- Notas adicionales
- Ejercicios recomendados
- Preguntas/Dudas

Referencias

INTRODUCCIÓN

- Acerca de mi
  - Licenciado en Ciencias de la computación
  - Finalista Mundial ACM-ICPC 2018 (Beijing)
  - SDE I en Amazon Madrid
  - Todo el contenido de esta clase está basada en mi opinión
  - Nada de lo que diga es en nombre, ni en representación de Amazon
  - No me considero un experto en Programación Competitiva
  - Me apasiona la computación
  - ID en la mayoría de websites de PC: cftryharder/lstd

INTRODUCCIÓN

- Esquema de las clases:
  - Idea general del algoritmo
  - ¿Qué problema resuelve?
  - Análisis de Complejidad en tiempo
  - Observaciones
  - Pseudocódigo(o código en C++) del algoritmo
  - Ejemplo
  - Notas adicionales
  - Ejercicios recomendados
  - Preguntas/Dudas

TRIE

TRIE

Idea general

- Problema introductorio:
  - Dado un **texto** representado por un string **T**, queremos saber si existe alguna palabra dentro del texto que tenga el **prefijo P**
- **Recordar**: Un prefijo es un substring de la forma **T[0..i] donde 0 <= i <= n-1**
- ¿Ideas?

TRIE

T = "programar programas en prolog es divertido"

- Consultas:
  - query(“divertido”) => **true**
  - query(“sat”) => **false**
  - query(“prol”) => **true**

TRIE

Técnica: búsqueda lineal sobre el texto entero

- Comparamos el prefijo de **tamaño M** con cada palabra del texto
- Si el texto tiene **N palabras**, la complejidad en tiempo de cada query es:
  - **O(N*M)**
- Next level: **M** queries **1 <= M <= 10^4**
- **Spoiler alert:** Trie permite realizar estas búsquedas en **O(M)** luego de insertar todas las palabras del texto en el Trie

TRIE

Definición:

Un trie es una estructura de datos de **tipo árbol** que permite la recuperación de información (de ahí su nombre del inglés reTRIEval). La información almacenada en un trie es un conjunto de claves, donde una **clave** es una secuencia de símbolos pertenecientes a un **alfabeto**. Las claves(y posiblemente los valores) son almacenadas en las hojas del árbol y los nodos internos son pasarelas para guiar la búsqueda.

Referencia: Wikipedia

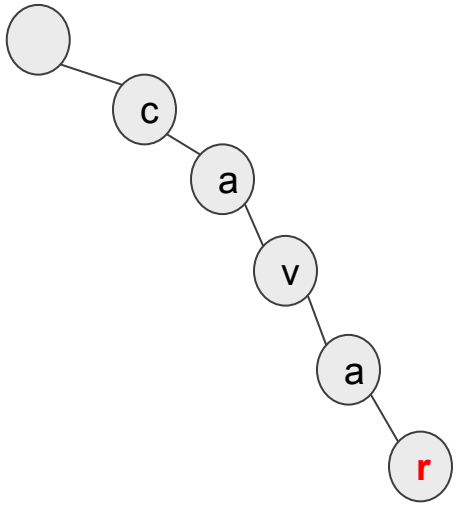
TRIE

Ejemplo:

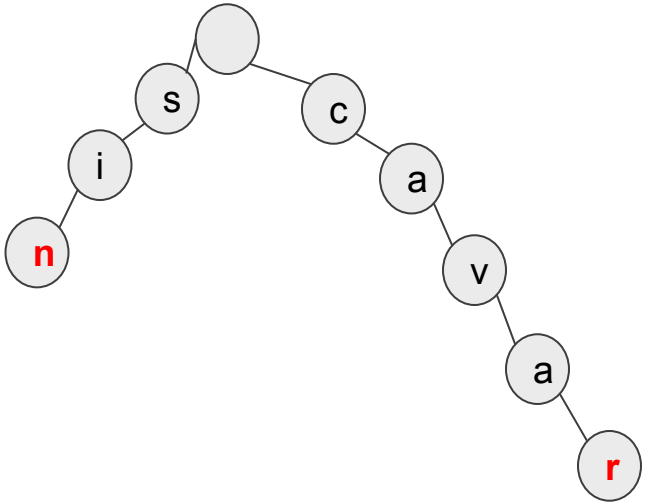
T = "cavar sin cava o caviar no es cavar"

TRIE

insert("cavar")

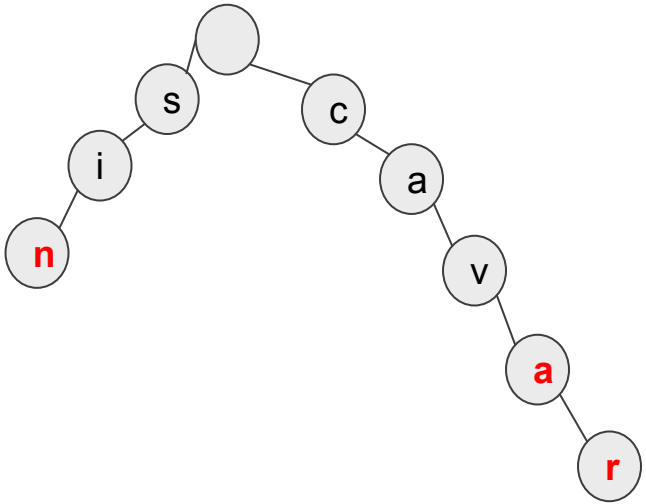


insert("sin")

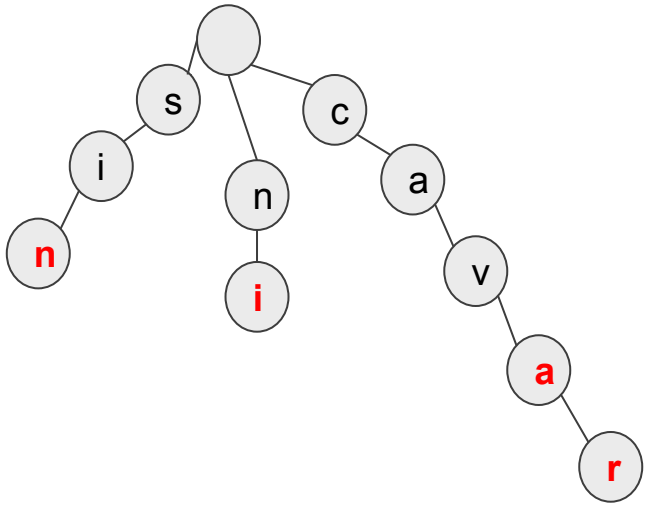


TRIE

insert("cava")

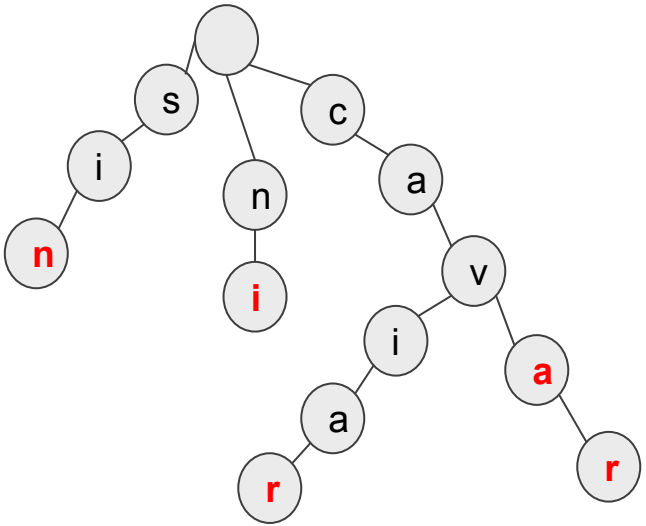


insert("ni")



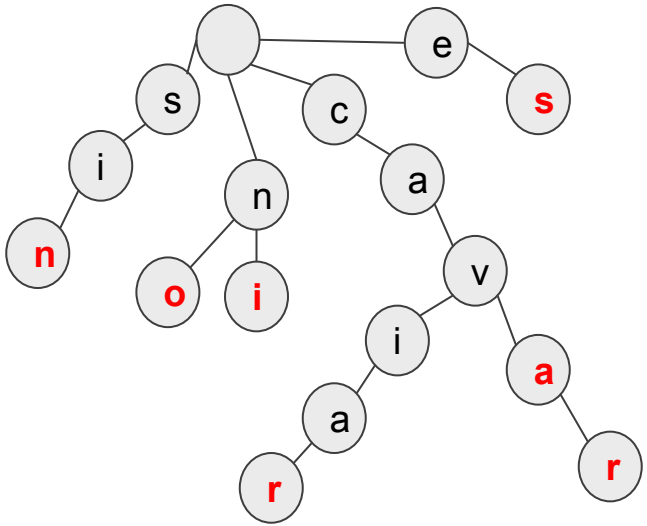
TRIE

insert("caviar")



TRIE

insert("no"),
insert("es"),
insert("cavar")



TRIE

- Luego de procesar todo el texto, responder una query solo tomaría **O(M)**
- Es posible responder queries de diferentes tipos en la misma complejidad:
  - Cuantas (o cuales) palabras empiezan por el **prefijo P**
  - Existe o no la palabra W en el texto

TRIE

C++:

```
const int ABCSZ = 30;
struct Node {...}
vector<Node> trie
insert(string word)
query(string prefix)
```

TRIE

```
struct Node {
    bool isWord;
    vector<int> abc;
     Node() {
        isWord = false;
        abc.assign(ABCSZ, -1);
    }
};
```

```
void insert(string &word) {
    int n = word.size();
    int cur = 0;
    for(int i = 0; i < n; i++) {
        char c = word[i];
        int next = trie[cur].abc[c-'a'];
        if (next == -1) {
            trie[cur].abc[c-'a'] = ++cnt;
            cur = cnt;
        } else {
            cur = trie[cur].abc[c-'a'];
        }
        if (i == n-1)
            trie[cur].isWord = true;
    }
}
```

TRIE

```
bool query(string &p) {
    int n = p.size();
    int cur = 0;
    for(int i = 0; i < n; i++) {
        int next = trie[cur].abc[p[i]-'a'];
        if (next == -1)
            return false;
        cur = next;
    }
    return true;
}
```

TRIE

Análisis de complejidad en tiempo:

- Insertar una palabra en el trie:
  - O(M) donde M es la longitud de la palabra a insertar
- Consulta basada en un prefijo.
  - Este tipo de consultas suelen ser O(P) donde P es la longitud del prefijo pero puede variar dependiendo del tipo de consulta.
- Eliminar una palabra del trie
  - O(M) donde M es la longitud de la palabra a eliminar
- Construcción del Trie
  - O(N*M) donde N son la cantidad de palabras a insertar y M es la longitud máxima de las palabras

TRIE

Problema de ejemplo: Replace K words

TRIE

- Como resolver este problema. ¿Ideas?
  - Comparar cada palabra del texto con las del diccionario en busca de la que tenga menor longitud y que comparta el mismo prefijo. TLE
  - Se puede hacer de manera más eficiente?
    - Si. Podríamos insertar todas las palabras del diccionario en un Trie y por cada palabra del texto tratar de encontrar la que tenga una palabra que comparta prefijo. La primera que se encuentra será la de menor longitud

TRIE

```
string findWordByPrefix(string &p) {
    int n = p.size();
    int cur = 0;
    string pre;
    for(int i = 0; i < n; i++) {
        if (trie[cur].isWord)
            return pre;
        int next = trie[cur].abc[p[i]-'a'];
        if (next == -1)
            break;
        pre += p[i];
        cur = next;
    }
    return p;
}
```

TRIE

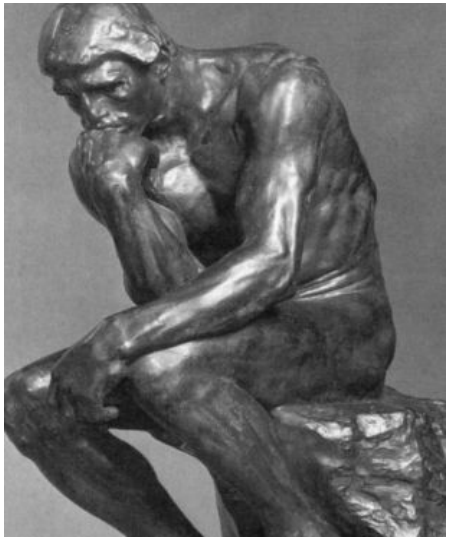
Notas adicionales:

- Esta estructura de datos fue introducidos en 1959 independientemente por Rene de la Briandais y Edward Fredkin
- Este algoritmo puede utilizarse no solo para strings sino para claves y valores en general
- Un trie es un caso especial de autómata finito determinista . Más Información aquí

TRIE

Problemas recomendados:

- https://leetcode.com/problemset/all/?topicSlugs=trie
- https://codeforces.com/blog/entry/75302



¿PREGUNTAS?

SUFFIX ARRAY

SUFFIX ARRAY

Idea general

- Problema introductorio:
  - Dado un **texto** representado por un string **T**, queremos saber si cierto **substring S** existe dentro del texto.
- ¿Ideas?

SUFFIX ARRAY

Algoritmo fuerza bruta

1. Comparamos todos los substrings del texto **T** con el **substring S**
2. Complejidad en tiempo: **O(|T| * |S|)** donde **|x|** significa: **longitud del arreglo x**
3. Este algoritmo podría servir si tenemos un **texto pequeño y pocas consultas**
4. **Pregunta**: cuál es la longitud (aproximada) que podrían tener T y S tal que este algoritmo de fuerza bruta sea viable si el Time Limit es 1s?

SUFFIX ARRAY

Una manera más eficiente: Suffix Trie/Suffix Tree

1. Este método se basa en el siguiente hecho:
   a. Todo **prefijo** de un **sufijo** es un **substring**
2. Consiste en agregar a un Trie todos los sufijos del string T
3. Verificar si un string S es un substring de T solo tomaría O(S)
4. El Suffix Tree es una mejora sobre el Suffix Trie
5. El Suffix Tree suele ser muy complicado de codificar en un entorno de competencias de programación

Ejemplo: https://visualgo.net/en/suffixtree

w = "GATAGACA", s = "TAG"

sufijos = ["GATAGACA","ATAGACA","TAGACA","AGACA","GACA","ACA","CA","A",""]

SUFFIX ARRAY

Suffix Trie/Suffix Tree

Estas estructuras de datos también resuelven los siguientes problemas:

1. Dado un **string T** determinar el **substring repetido más largo**
2. Dado **dos strings T1 y T2** determinar el **substring común más largo**

Ejemplo: https://visualgo.net/en/suffixtree

SUFFIX ARRAY

Los suffix arrays o arreglos de sufijos son arreglos que contienen el índice del carácter inicial de cada sufijo de un string

Existen diversas maneras de construir un suffix array

Los suffix array sirven para resolver problemas los mismos problemas que el Suffix Trie/Suffix Tree pero es más sencillo de codificar

SUFFIX ARRAY

Definición

Dado un **string s** de **longitud n**. El **i-ésimo sufijo** de s es el **substring s[i...n-1]**

- Un suffix array contiene enteros que representan el índice inicial de todos los sufijos ordenados lexicográficamente de un string dado. Ejemplo:

w = “ababba”

Sufijos: [“ababba”, “babba”, “abba”, “bba”, “ba”, “a”]

Suffix array: [5, 0, 2, 4, 1, 3]

SUFFIX ARRAY

Construcción del Suffix Array - Naive

1. La manera más ingenua de construir el SA es insertar todos los sufijos del string S en un arreglo de strings y ordenar el arreglo de strings
2. Este método tiene una complejidad en tiempo de O(|S|^2*log|S|) ya que la comparación entre dos strings se realiza caracter a caracter
3. **Pregunta**: cuál es la mayor longitud (aproximada) que puede tener el string S para que este método sea viable si el Time Limit es 1s?

SUFFIX ARRAY

Construcción del Suffix Array - Primera optimización

1. **Observación:** podemos ordenar por partes (iteraciones de potencias de dos) los sufijos y quitarnos las comparaciones caracter a caracter. Ejemplo: http://codeforces.com/edu/course/2/lesson/2/1
2. Este algoritmo requiere **log|T|** iteraciones donde en cada iteración se aplica un ordenamiento pero esta vez estaremos comparando solo las **clases de equivalencia** de cada par de **substrings de longitud igual a 2^k en la iteración k+1**
3. Complejidad resultante: **O(|T|log^2|T|)**

SUFFIX ARRAY

Construcción del Suffix Array - Segunda y tercera optimización

1. **Observación:** Los elementos a ordenar siempre serán menores que **|T|**.
2. **Segunda optimización**: utilizar **counting sort** para ordenar en **O(|T|)**
3. **Tercera optimización:** utilizar **radix sort** para ordenar en **O(|T|)**. Esta última suele ser más eficiente que la segunda optimización.
4. Complejidad resultante: **O(|T|log|T|)**

SUFFIX ARRAY

Implementación de la primera optimización en C++:

https://ideone.com/JKDP0d

SUFFIX ARRAY

Tarea: implementar la 2da y 3ra optimización

SUFFIX ARRAY

Ejemplo: Encontrar si un substring SS existe en un string S

1. Calculamos el SA del string S
2. Para cada substring SS hacemos binary search sobre el SA
3. Complejidad total: O(|SS|log|S|)

SUFFIX ARRAY

Problema clásico #1:

Encontrar la longitud del substring repetido más largo

1. Calculamos el SA del string S
2. Calculamos la tabla Longest Common Prefix (LCP) *
3. El valor maximal de la tabla LCP es el valor que buscamos
4. Complejidad total: O(|SS|log|S|)

(*): Para calcular la tabla LCP ver Step 3 del curso de Codeforces. Nota: en el libro CP explican otro método distinto para construir la tabla LCP. Ambas se basan en la misma observación.

SUFFIX ARRAY

Problema clásico #2:

Encontrar el prefijo comun entre 2 sufijos a y b

1. Calculamos el SA del string S
2. Calculamos la tabla Longest Common Prefix (LCP) *
3. Sobre el arreglo LCP debemos implementar una estructura de datos que nos permita calcular el RMQ(i, j). Para esto podemos implementar **Sparse Table** o **Segment Tree**
4. Complejidad total: **O(|SS|log|S|)**

(*): Para calcular la tabla LCP ver Step 3 del curso de Codeforces. Nota: en el libro CP explican otro método distinto para construir la tabla LCP. Ambas se basan en la misma observación.

SUFFIX ARRAY

Problema clásico #3:

Encontrar el substring común entre 2 strings s1 y s2

1. S = s1+"#"+s2
2. Calculamos el SA del string S
3. Calculamos la tabla Longest Common Prefix (LCP) *
4. Sobre el arreglo LCP debemos buscar el par de sufijos adyacentes con mayor valor y que pertenezcan a diferentes strings
5. Complejidad total: **O(|SS|log|S|)**

(*): Para calcular la tabla LCP ver Step 3 del curso de Codeforces. Nota: en el libro CP explican otro método distinto para construir la tabla LCP. Ambas se basan en la misma observación.

SUFFIX ARRAY

Problema clásico #4:

Encontrar el número de substrings diferentes en un string S

1. Calculamos el SA del string S
2. Calculamos la tabla Longest Common Prefix (LCP) *
3. Iteramos los sufijos en el orden dado por SA. Determinamos cuantos prefijos podemos contar sin tomar en cuenta los repetidos que será el LCP[i]
4. Complejidad total: **O(|S|log|S|)**

(*): Para calcular la tabla LCP ver Step 3 del curso de Codeforces. Nota: en el libro CP explican otro método distinto para construir la tabla LCP. Ambas se basan en la misma observación.

SUFFIX ARRAY

Notas Adicionales:

- Esta estructura de datos fue descubierta por Udi Manber y Gene Myers
- Los SA suelen ser aplicados en la bioinformática para estudiar patrones en el ADN y en cadenas de proteínas
- Se utilizan en algoritmos de compresión de datos. Por ejemplo, esta variante del algoritmo de compresión LZW
- Hay bastantes estudios recientes referenciados en wikipedia (los más recientes son del 2014 y 2016)

SUFFIX ARRAY

Ejercicios recomendados:

- https://codeforces.com/edu/course/2/lesson/2 Todos los ejercicios, de los steps
- https://leetcode.com/problemset/all/?topicSlugs=suffix-array
- https://cp-algorithms.com/string/suffix-array.html#toc-tgt-11

REFERENCIAS

- https://cp-algorithms.com/string/suffix-array.html
- https://codeforces.com/edu/course/2/lesson/2
- https://www.geeksforgeeks.org/persistent-segment-tree-set-1-introduction/
- https://www.youtube.com/watch?v=AXjmTQ8LEoI
- Wikipedia -Trie
- https://cpbook.net/details?cp=4 - Chapter 6

AGRADECIMIENTOS

¡GRACIAS!

HAPPY CODING A TODOS