Auxiliar #6 - Algoritmos sobre strings

16 de noviembre de 2020 - Bernardo Subercaseaux

Problema 1. (★★★) En esta pregunta veremos como ordenar strings. Los strings son secuencias de caractéres de un alfabeto constante de tamaño σ, y se desean ordenar alfabéticamente. Teniendo strings $s_1, \ldots, s_n$, tales que su largo total es $\Sigma_i |s_i| = N$, diseñe algoritmos para ordenarlos en tiempo $O(N)$ en los siguientes casos:

- todos los strings tienen el mismo largo.
- los strings tienen largo variable.

Solución 1 . En el caso en que los strings tienen el mismo largo, tienen largo exactamente N/n. Si hacemos un `RadixSort` que ordena caracter a caracter los strings, la complejidad total será $O\left(\frac{N}{n}(n+\sigma)\right) = O(N)$. En el caso de largo variable no podemos usar `RadixSort` directamente, ya que por ejemplo $s_1 =$ `ca` quedaría antes de $s_2 =$ `b`, lo que es incorrecto. Una solución ingenua es rellenar con un caracter especial los strings más cortos, para que todos tengan el mismo tamaño. Sin embargo, eso puede ser $O(N^2)$ si es que originalmente había un string de largo $N/2$ y $N/2$ strings de largo 1.

Para resolverlo en tiempo $O(N)$ podemos usar una variante recursiva de `RadixSort`, en que ordenaremos desde el caracter más significativo hasta el menos significativo. Antes de describirla en detalle, resolvamos un caso borde. Podemos agregar una caracter especial (como \$) al final de cada string, que sea menor lexicográficamente que el resto, para que palabras como `hola$` aparezcan antes que `holanda$` al tener menor tamaño. Describamos el algoritmo. Partimos inicialmente haciendo `BucketSort` con respecto al primer caracter. Luego de eso tendremos todos los strings ordenados con respecto a su primer caracter. Por cada grupo de strings que comienza con el mismo caracter, llamaremos recursivamente al algoritmo, pero ahora considerando desde el segundo caracter en adelante. Si un grupo de strings que comienza con el mismo caracter tiene tamaño 1, entonces obviamente no es necesario hacer nada con él. A la hora de procesar un grupo de strings que comparten los caracteres anteriores en cuanto a su d-ésimo caracter, aquellos que tengan \$ en ese caracter son simplemente saltados, ya que ya se encuentran ordenados y no podremos ordenarlos por caracteres posteriores. Para el análisis de tiempo en vez de contar cuantos `BucketSort` hacemos, cobremosle a un caracter cada vez que un `BucketSort` lo considera el ordenar. Cada caracter del input se procesará en un solo `BucketSort`, ya que el d-ésimo caracter de un string s_i solo entrará en un `BucketSort` cuando se esté ordenando con respecto al d-ésimo caracter a todos quienes comparten los primeros $d-1$ caracteres de s_i. Así, por cada caracter del input se está pagando una constante (la del único `BucketSort` que pasa por él), lo que da un costo total de $O(N)$.

Problema 2. (★★★) Considere nuevamente que tienes strings $s_1, \ldots, s_n$, cuyo largo total es $\sum_i |s_i| = N$. Se desea procesar consultas en que dado un nuevo string p, se desea saber si este coincide con alguno de los s_i. Diseñe un algoritmo con complejidad $O(p)$ por consulta, tras un preprocesamiento en que puede usar espacio extra $O(1)$.

Solución 2 . No podemos utilizar una tabla de hash, ya que eso requiere guardar un hash por cada string s_i, lo que no sería constante. En lugar de eso, utilizaremos una estructura de

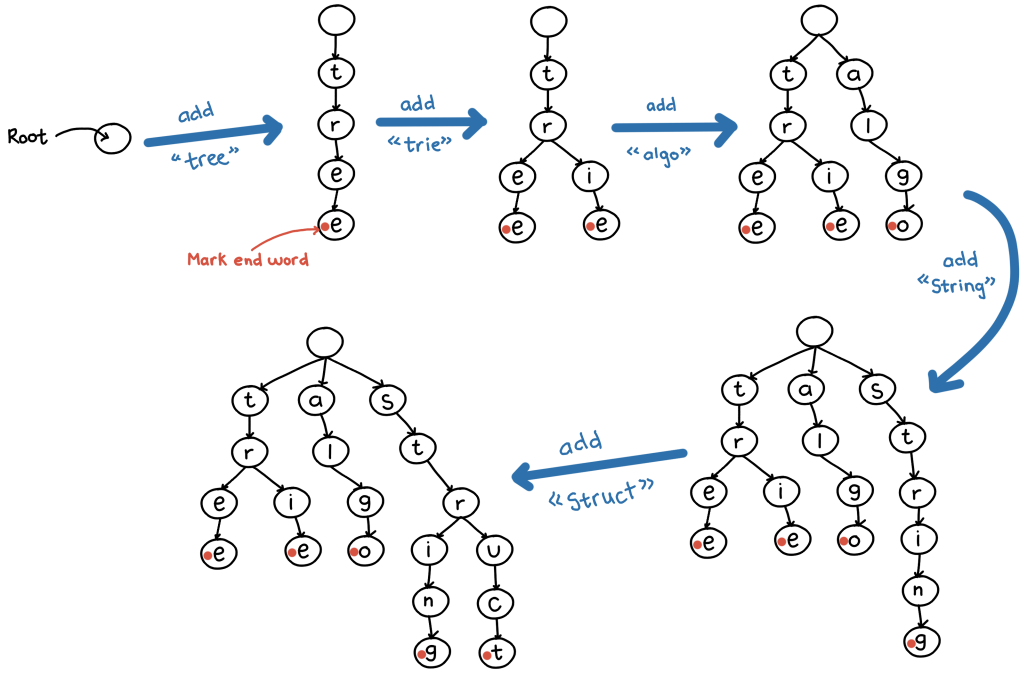

Figura 1: Ejemplo de un trie al que se le añaden las palabras `[tree, trie, algo, String, Struct]` en ese orden.

árbol llamada *trie* (o *árbol de prefijos*). En esta árbol cada nodo está etiquetado con un caracter, y tiene a lo sumo σ hijos. Como se muestra en la Figura 1, para insertar strings en el árbol se recorre partiendo desde la raíz, y por cada caracter $c \in [1, \sigma]$ del string se sigue por el c-ésimo hijo (creándolo en el momento si este no existe). Al insertar un nodo que corresponde al último caracter de una palabra, ese nodo se marca como *final*, para luego poder recordar que ahí termina una palabra.

Cada caracter del input se guarda una vez en el trie (como nodo), así que el espacio usado por los caracteres en el trie es el mismo que usaban todos los strings, y por tanto luego de construir el trie podemos borrar el arreglo inicial de strings. Además, cada nodo del trie simplemente requiere guardar σ punteros. Luego, para buscar un string $p = c_1, \ldots, c_k$. Comenzamos en la raíz, y en cada paso $i = 1..k$, vamos tomando el hijo correspondiente al caracter c_i. Si tal hijo no existe, entonces el string p no está en el trie. Si llegamos al nodo correspondiente a c_k y tiene una marca de nodo *final*, entonces concluimos que p si estaba en el trie. Cada caracter de p es un paso, que simplemente requiere chequear si existe el hijo deseado de un nodo, y si es así avanzar a él, lo que toma tiempo constante. Así, el costo de cada consulta es $O(|p|)$.

Problema 3. (★★) Modifique la estructura de la pregunta anterior para soportar búsquedas aproximadas de la siguiente forma. Dado un entero fijo k, que representa el grado de aproximación a soportar, se desea consultar con un string p si alguno de los strings $s_1, \ldots, s_n$ difiere en a lo más k caracteres con p. ¿Qué complejidad obtiene?

Solución 3 . Si al comparar dos caracteres estos no coinciden, diremos que eso es una *falla*. Modificaremos el algoritmo de búsqueda para que soporte a lo más k fallas. Para eso podemos un parámetro adicional, siendo `search(p, r, f)` donde p es el string a buscar, r es el nodo del trie desde el cuál se busca, y f son las fallas que nos podemos permitir. Inicialmente r es la raíz

del trie, y $f = k$. Luego, por cada hijo h de r, se llama recursivamente `search`$(p[1:], h, f')$ con $f' = f - 1$ si es que $h \neq p[0]$ (hay falla) o $f' = f$ si es que no. Si una llamada tiene $f = 0$ entonces ya no se toleran fallas.

Para estudiar la complejidad, notemos que por cada nodo del árbol a lo más se hace un llamado recursivo, por lo que el costo total es $O(N)$. Por otra parte, por cada caracter de p se pueden hacer a lo más σ llamados recursivos, así que tampoco hay más de $\sigma^{|p|}$ llamados. Así el costo es $O(N + \sigma^{|p|})$.