Strings

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp Argentina 2018

Contenidos

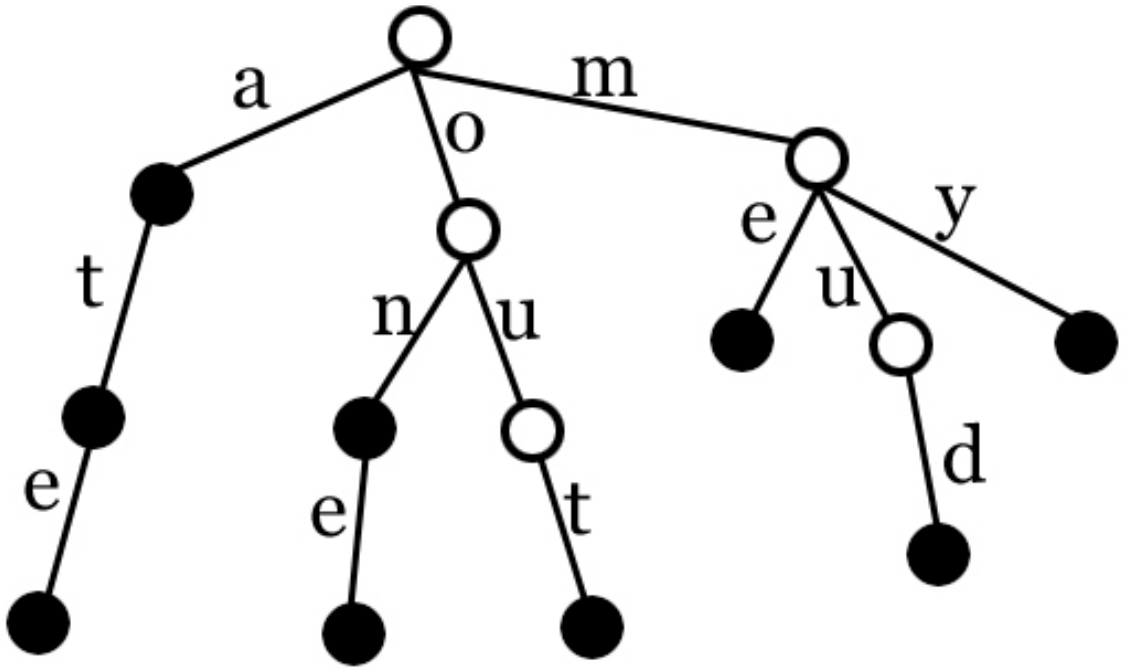
“ If a string has one end, then it has another end.”

Miksch’s Law

Contenidos

Idea

- Guardar un **conjunto de cadenas**, en un **árbol**.
- Por cada **hijo** de un nodo, hay una posible **elección** del **siguiente carácter**.
- Cada **camino** desde la raíz hasta un nodo, corresponde a una cadena posible.
- Los strings a priori **no están explícitamente almacenados** en ningún nodo.



a,at,ate
on,one,out
me,mud,my

Qué almacenar en los nodos

- Toda la información útil del nodo (string particular)
- Toda la información útil del subárbol (strings con ese prefijo)
- Ejemplos típicos
  - Booleano (aparece o no la palabra)
  - Contador (tamaño del subárbol, o sea cantidad con ese prefijo)
  - Máxima cadena con ese prefijo (con algún peso, longitud, etc)
- Es común usar programación dinámica o técnicas de árboles sobre un trie
- Es común querer tener los **hijos ordenados** para recorrer lexicográficamente

Implementación fácil recomendada

```
struct Trie // Ejemplo: solo guarda
{           // si esta o no la cadena
    bool esta;
    map<char,Trie> hijos;
    Trie () { esta = false; }
};
```

Para insertar `hola`:

```
Trie t;
t.hijos['h'] // Se crean
 .hijos['o'] // automaticamente
 .hijos['l'] // nuevos nodos
 .hijos['a'].esta = true;
```

Implementación (con insertar recursivo)

```
struct Trie // Ejemplo: solo guarda
{           // si esta o no la cadena
    bool esta;
    map<char,Trie> hijos;
    Trie () { esta = false; }
    void insertar(const string &s, int pos) {
        if (pos < int(s.size()))
            hijos[s[pos]].insertar(s,pos+1);
        else
            esta = true;
    }
    void insertar(const string &s) {
        insertar(s, 0);
    }
};
```

Implementación (con buscar recursivo)

```
struct Trie // Ejemplo: solo guarda
{           // si esta o no la cadena
    bool esta;
    map<char,Trie> hijos;
    Trie () { esta = false; }
    bool buscar(const string &s, int pos) {
        if (pos < int(s.size()))
            return hijos.find(s[pos]) != hijos.end() &&
                   hijos[s[pos]].buscar(s,pos+1);
        else
            return esta;
    }
    bool buscar(const string &s) {
        buscar(s, 0);
    }
};
```

Implementación (con insertar iterativo)

```
struct Trie // Ejemplo: solo guarda
{           // si esta o no la cadena
    bool esta;
    map<char,Trie> hijos;
    Trie () { esta = false; }
    void insertar(const string &s) {
        Trie *t = this;
        for (char c : s)
            t = &t->hijos[c];
        t->esta = true;
    }
};
```

Implementación (con buscar iterativo)

```
struct Trie // Ejemplo: solo guarda
{           // si esta o no la cadena
    bool esta;
    map<char,Trie> hijos;
    Trie () { esta = false; }
    bool buscar(const string &s) {
        Trie *t = this;
        for (char c : s)
            if (t->hijos.find(c) != t->hijos.end())
                t = &t->hijos[c];
            else
                return false;
        return t->esta;
    }
};
```

Contenidos

Uso como diccionario

Si se opera con una palabra de longitud L:

- Los Trie permiten implementar conjuntos y diccionarios de Strings con tiempo lineal en L
- Un `map<string, int>` en cambio tarda $L \cdot \lg n$ por acceso, que depende de n

Problema del estribillo

- Dado el texto de una canción, determinar su estribillo.
- Definición: el estribillo es la máxima subcadena del texto que aparece **al menos dos veces, sin solaparse**.
- Complejidad de soluciones directas: $O(N^4)$, $O(N^3 \lg N)$, $O(N^3)$
- ¿Con un simple Trie?

Problema del estribillo

- Dado el texto de una canción, determinar su estribillo.
- Definición: el estribillo es la máxima subcadena del texto que aparece **al menos dos veces, sin solaparse**.
- Complejidad de soluciones directas: $O(N^4)$, $O(N^3 \lg N)$, $O(N^3)$
- ¿Con un simple Trie? $O(N^2)$: Insertamos los substring en un Trie.
- Cuando hay **muchos prefijos compartidos**, los Tries permiten ahorrar tiempo

Queries por prefijos

- Máximo peso empezando con tal prefijo (autocompletar)
- Sumar longitudes de las palabras que empiezan con tal prefijo

Segment Tree

- Observación: Un Segment Tree es un Trie, donde las cadenas son la representación en binario de los índices.
  - 0 genera números más chicos y va a la mitad izquierda
  - 1 genera números más grandes y va a la mitad derecha
- Por eso las técnicas de árboles aplican por igual a Trie y a Segment Tree:
  - Lazy Creation
  - Persistencia

Problema de los equipos de Rugby

- `http://www.dc.uba.ar/events/icpc/download/problems/taip2012-problems.pdf`
- Problema D: Diseño de camisetas
  - Dos equipos, de *N* jugadores cada uno
  - Cada jugador tiene su propio apellido
  - Hay que hacer *N* camisetas que sirvan para cualquiera de los dos
  - Cada camiseta debe tener un **prefijo** del nombre de quien la usa
  - Escribir la máxima cantidad de letras en total en las camisetas

Problema de los equipos de Rugby

- `http://www.dc.uba.ar/events/icpc/download/problems/taip2012-problems.pdf`
- Problema D: Diseño de camisetas
  - Dos equipos, de *N* jugadores cada uno
  - Cada jugador tiene su propio apellido
  - Hay que hacer *N* camisetas que sirvan para cualquiera de los dos
  - Cada camiseta debe tener un **prefijo** del nombre de quien la usa
  - Escribir la máxima cantidad de letras en total en las camisetas
- Idea: Mirando todo en el Trie de los nombres, los hijos inducen problemas separados.

Problema de los equipos de Rugby

- `http://www.dc.uba.ar/events/icpc/download/problems/taip2012-problems.pdf`
- Problema D: Diseño de camisetas
  - Dos equipos, de N jugadores cada uno
  - Cada jugador tiene su propio apellido
  - Hay que hacer N camisetas que sirvan para cualquiera de los dos
  - Cada camiseta debe tener un **prefijo** del nombre de quien la usa
  - Escribir la máxima cantidad de letras en total en las camisetas
- Idea: Mirando todo en el Trie de los nombres, los hijos inducen problemas separados.
- Código: Meter ambos equipos en un Trie, haciendo `A++` y `B++` en cada caso, por todos los nodos visitados.
- La respuesta final es sumar $min(A, B)$ sobre todos los nodos (menos la raíz).

Problema de los equipos de Rugby (código)

```
int N,ret;
struct Trie {
    int v[2];
    map<char, Trie> h;

    Trie() : v{0,0} {}
    void solve() {
        for(auto &it : h)
            it.second.solve();
        ret += min(v[0], v[1]);
    }
};
int main() {
    while (scanf("%d", &N) && (N != -1)) {
        Trie root;
        forn(k,2)
        forn(i,N) {
            static char name[128];
            assert(scanf("%s", name));
            Trie *t = &root;
            for(char *p = name; *p; p++) {
                t = &t->h[*p];
                t->v[k]++;
            }
        }
        ret = 0;
        root.solve();
        printf("%d\n", ret);
    }
    return 0;
}
```

Contenidos

Idea

- Dadas dos cadenas, compararlas es “ir buscando coincidencias” hasta la primera diferencia.
- Vamos a “ir buscando coincidencias” **contra el comienzo del string**, para **cada posición del string**.
- Vamos a anotar para cada posición del arreglo, cuántas coincidencias contra el comienzo hay **empezando desde allí**.
- Ejemplo: `banabanba` genera el arreglo `9 0 0 0 3 0 0 2 0`

Idea (cont)

- Definición: $lcp(i, j)$ es la cantidad de coincidencias, si empezamos a comparar en i y en j.
- Es decir: $lcp(i, j)$ es el *longest common prefix* entre los sufijos que empiezan en las posiciones i y j de la cadena.
- Observación: $z[i] = lcp(i, 0)$, por la definición del arreglo z.
- Tiempo de cómputo trivial: $O(n^2)$

Algoritmo eficiente

- Se puede computar el arreglo z en $O(n)$
- Idea clave: solo se paga caro, si hay muchas coincidencias. Podemos “usar esa información” para abaratar futuras preguntas.
- El algoritmo mantiene todo el tiempo un j, tal que “ya se han observado” los caracteres hasta j.
- Además mantiene un i lo más chico posible, de manera tal que la subcadena del rango $[i, j)$ es prefijo de la cadena.

Ejemplo: bambambab

```
bambambab
9........
```

- Inicializamos con la longitud, pues siempre $lcp(0, 0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera `m`, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Ejemplo: bambambab

bambambab
900.....

- Inicializamos con la longitud, pues siempre $lcp(0, 0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera `m`, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Ejemplo: bambambab

```
bambambab
9005.....
```

- Inicializamos con la longitud, pues siempre $lcp(0, 0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera `m`, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Ejemplo: bambambab

bambambab

900500**5..**

- Inicializamos con la longitud, pues siempre $lcp(0,0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera `m`, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Ejemplo: bambambab

bambambab
9005002..

- Inicializamos con la longitud, pues siempre $lcp(0, 0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera m, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Ejemplo: bambambab

bambambab
900500201

- Inicializamos con la longitud, pues siempre $lcp(0, 0) = n$
- Además tenemos $i = 1$ y $j = 1$.
- A continuación, vamos explorando los nuevos caracteres, pero no hay coincidencias.
- Quedamos en $i = j = 3$.
- Exploramos y queda $i = 3, j = 8$. $[i, j)$ en azul **ya sabemos que coincide**.
- Para todos los de la zona azul, podemos saber lo que pasa dentro de la zona azul en $O(1)$, copiando valores anteriores ya calculados.
- ¡Pero ojo, el 5 se sale de la zona azul! Hay que verificar:
  - En este caso queda en 2.
  - Si la última fuera `m`, seguiríamos expandiendo la frontera azul.
- Como en cada paso llenamos uno más, o bien avanzamos lo azul, es $O(n)$

Código

```
int i = 1, j = 1;
z[0] = n;
for (int pos=1;pos<n;pos++) {
    if (j < pos)
        i = j = pos;
    int k = min(z[pos-i], j-pos);
    while (pos+k < n && s[pos+k] == s[k])
        k++;
    z[pos] = k;
    if (pos+k > j) {
        j = pos+k;
        i = pos;
    }
}
```

Contenidos

El problema String matching

- Dada una cadena A y otra B, dar todas las apariciones de A dentro de B.
- Ya vimos cómo “ir comparando” **contra el comienzo** del string.
- Si armo el arreglo z de la cadena AB,en aquellos valores de la parte B que sean al menos $|A|$ hay apariciones.

Problemas de String Matching

- Verificar si una cadena es rotación de otra
- Plagio musical: Buscar un patrón “trasladado” (1 10 3 5 = 3 12 5 7)
- Armar notas a partir del diario: $O(N^2)$.
  `https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1521`

Contenidos

El problema de bordes

- Dada una cadena S, se le llama un **borde** de S, a un prefijo que es al mismo tiempo sufijo.
- Es fácil identificar bordes con el arreglo z: Es borde si y solo si $z[i] = n - i$
- Tabla de KMP: Borde máximo de cada prefijo. Se puede generar fácilmente a partir del arreglo z.
- Relación entre bordes y períodos: P es período del string, si y solo si $n - P$ es borde.
- Sirve para detectar potencias: S es una potencia, si y solo si tiene un período que divide a $|S|$

Ejemplo

- File recovery: Dado un string S y un tamaño n, cuántas veces puede aparecer S como máximo en una cadena de tamaño n.
- Mayor sufijo palindrómico
  `https://www.spoj.com/problems/EPALIN/`
- Tarea: `https://www.spoj.com/problems/PERIOD/`

Contenidos

Idea

- Un número en la vida cotidiana lo escribimos en “texto”. Así que es un String.
- ¿Cómo es este mapeo de un String a un número?

- Si hacemos la cuenta para un $X > |\Sigma|$, cada string tendrá un número único.

Idea

- Un número en la vida cotidiana lo escribimos en “texto”. Así que es un String.
- ¿Cómo es este mapeo de un String a un número?
- Hay una base ($X = 10$), y “325” representa $3X^2 + 2X + 5$, un **polinomio**.
- Podemos pensar cualquier string como un polinomio
  - Se pueden pensar las letras como números $0, 1, 2, \cdots$ en el orden del alfabeto.
  - Las letras de la cadena serán los coeficientes.
  - Ejemplo: $BAC \to 2X^2 + 1X + 3$
- Si hacemos la cuenta para un $X > |\Sigma|$, cada string tendrá un número único.

Hash de Rabin-Karp

- Al número anterior se lo llama el hash del string.
- ¿Qué pasa con el hash al agregar un caracter a derecha?

Hash de Rabin-Karp

- Al número anterior se lo llama el hash del string.
- ¿Qué pasa con el hash al agregar un caracter a derecha?
- $H \to H \cdot X + c$
- ¿Y a la izquierda?

Hash de Rabin-Karp

- Al número anterior se lo llama el hash del string.
- ¿Qué pasa con el hash al agregar un caracter a derecha?
- $H \to H \cdot X + c$
- ¿Y a la izquierda?
- $H \to H + c \cdot X^L$

Similarmente, se puede restar o despejar con cuidado para borrar un caracter de la punta.
$S = c_{left} S' \to S' c_{right}$
$H \to H \cdot X + c_{right} - c_{left} \cdot X^L$
Conclusión: Al moverse, se actualiza en $O(1)$

Idea de Rabin-Karp

- Para hacer string-matching y similares, trabajar con los numeritos. Como son $O(1)$ cuentas, es todo eficiente.

Idea de Rabin-Karp

- Para hacer string-matching y similares, trabajar con los numeritos. Como son $O(1)$ cuentas, es todo eficiente.
- Problema: números enormes: overflow o complejidad mala, como usar string directo.
- Rabin y Karp lo sabían: Proponen hacer todo módulo P, para algún primo P.
- Sugerencia: Usar 3 primos distintoss **al azar**, entre 10^9 y $2 \cdot 10^9$. La chance de colisión (falso positivo) es despreciable.
- Cuidado con las colisiones: Con un solo primo de ese tamaño, hay colisión seguro en muchos casos. Y con 2 está medio al borde.

Cálculo $O(1)$

- Precomputamos $T[i] = hash(S[0, i))$.
- El hash de $S[i, j)$ es $T[j] - T[i] \cdot X^{j-i}$ (precomputar los X^k)

Contenidos

Ejemplos

- LCP general en $\lg n$ (en particular, arreglo Z en $n \lg n$)
- Comparacion general de dos sufijos
- Sufijo / rotación lexicográficamente mínima (comparando)
- "Algoritmo de Manacher" en $n \lg n$
- Máximo substring común entre varios strings, en $n \lg^2 n$ (o $n \lg n$ con tabla hash)
- Problema del estribillo $O(n lg^2 n)$
- Búsqueda rápida de varios $S_1, S_2, \cdots, S_n$ de una misma longitud, dentro de T
- Búsqueda rápida de patrón rectangular en una grilla

Contenidos

Definiciones

- Definición: arreglo con todos los sufijos (identificados por índice) **en orden lexicográfico**
- Se computa fácil con hashing (permite comparar dos cualquiera) + sort. Total: $n \lg^2 n$.
- Estructura extremadamente útil: es como tener "un trie con todos los sufijos" pero en forma "camuflada", con un costo operativo de $\lg n$ por cada paso.
- La Burrows-Wheeler Transform es lo mismo que el suffix array pero con las rotaciones (se puede computar fácilmente cualquiera de los dos teniendo el otro)

Contenidos

Ejemplos

- Armar notas a partir del diario: $O(N \lg N)$.
  `https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1521`
- Responder queries de cantidad de apariciones de substring arbitrario en $O(s \lg N)$ por query