Strings

Conceptos Fundamentales para Programación Competitiva

Federico Bersano

Universidad Nacional de Rosario
Don Gato

Training Camp 2025

¡Gracias sponsors!

Organizador

Diamond Plus

¡Gracias sponsors!

Platino

FOLDER IT

INTERNATIONAL
SOFTWARE COMPANY

Gold

NeuralSoft

Oro

JS JERÁRQUICOS

¡Gracias sponsors!

Aliado

Definiciones

- **string**: arreglo de caracteres.
  - Ejemplo: `"banana"`
- **substring**: cadena no vacía consecutiva de caracteres de un string.
  - Ejemplos: `"bana"`, `"ban"`, `"ana"`, `"anana"`, `"nan"`
  - Un string de largo n tiene $n \times (n+1)/2$ substrings posibles.
- **prefijo**: substring que empieza al principio del string.
  - Ejemplos: `"b"`, `"ba"`, `"ban"`, `"bana"`, `"banan"`, `"banana"`
- **sufijo**: substring que termina al final del string.
  - Ejemplos: `"banana"`, `"anana"`, `"nana"`, `"ana"`, `"na"`, `"a"`
- **alfabeto**: conjunto de caracteres que pueden aparecer en un string.
  - Lo denotamos con Σ.
  - Suele ser el conjunto de 26 letras minúsculas del alfabeto inglés.

Interfaz en C++

- Declarar un string s → `string s = "hola mundo";`
- Caracter en el índice idx → `s[idx];`
- Largo del string → `s.size();`
- Concatenar un caracter c → `s += c;`
- Concatenar un string t → `s += t;`
- Substring desde el índice idx de largo len → `s.substr(idx, len);`

Motivación

Se tiene un diccionario de N palabras. Para cada una de otras M palabras se quiere saber si pertenece o no al diccionario.

La solución más sencilla sería iterar para cada palabra a buscar por todo el diccionario. La complejidad queda $O(N \times M \times L)$, donde L es el largo de los strings.

Podemos hacer mejor.

Trie

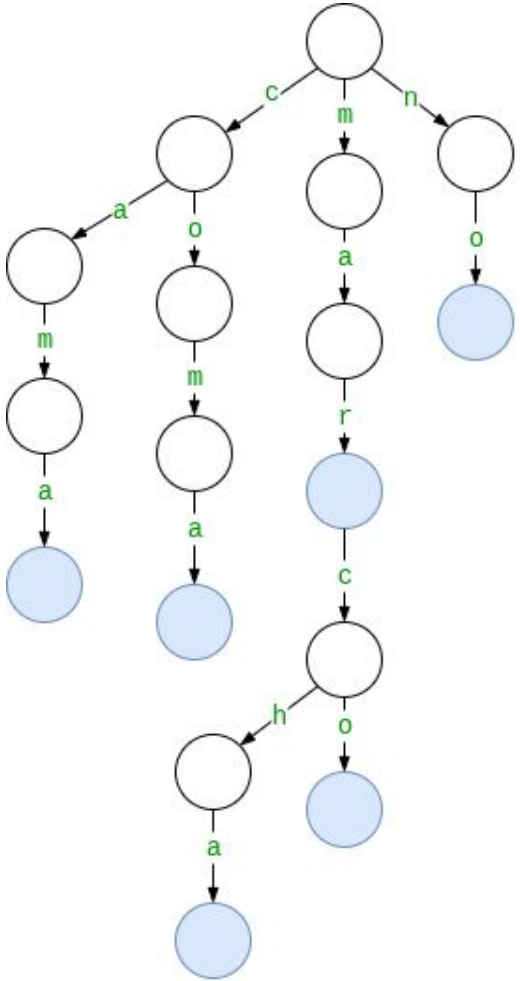
Un **trie** es un árbol que representa un conjunto de strings:

- el nodo raíz representa el string vacío.
- cada arista posee un caracter: recorrer la arista representa escribirlo.
- los nodos donde termina un string son nodos terminales.
- a cada nodo le corresponde un string generado por las aristas en el camino desde la raíz, estos strings son prefijos del diccionario.

Trie - Ejemplo

Ejemplo de trie para el diccionario `"cama"`, `"coma"`, `"mar"`, `"marco"`, `"marcha"`, `"no"`. Los nodos azules son nodos terminales.

Para ver si una palabra pertenece o no al trie, se recorre el árbol desde la raíz siguiendo las aristas con las letras correspondientes. Si se puede recorrer toda la palabra y al final se llega a un nodo terminal, entonces sí pertenece.



Trie - Implementación

La complejidad de insertar o buscar un string de largo L es $O(L \times \log(|\Sigma|))$.

Como en general Σ suele ser chico, el factor $\log$ es casi una constante. De ahora en más lo ignoramos.

```
struct trie {
	map<char, trie> hijos;
	bool term = false;

	void insertar(string const& s, int pos = 0) {
		char c = s[pos];
		if (pos < int(s.size())) hijos[c].insertar(s, pos+1);
		else term = true;
	}

	bool buscar(string const& s, int pos = 0) {
		char c = s[pos];
		if (pos < int(s.size())) return hijos.count(c) && hijos[c].buscar(s, pos+1);
		else return term;
	}
};
```

Trie - Usos

Volviendo al problema motivador, ahora podemos construir un trie de todo el diccionario en $O(N \times L)$.

Luego, para cada una de las M palabras a buscar, la búsqueda es $O(L)$.

El costo total es $O(N \times L) + O(M \times L) = O((N+M) \times L)$.

Trie - Usos

Los tries también suelen ser útiles en problemas de prefijos. Permiten calcular:

- número de strings con cierto prefijo.
- máxima cadena con cierto prefijo.
- etc.

Para resolver cosas así es común guardar un poco más de información en cada nodo, como la cantidad de nodos o nodos terminales en su subárbol.

Problema - Phone List

Dado un conjunto de números de teléfono, determinar si el mismo es consistente, es decir, ninguno de ellos es prefijo de otro.

- Input:
  - `911`
  - `91125426`
- Output:
  - `"NO"`

Problema - Phone List

Solución con trie:

- Armar un trie con todos los números de teléfono.
- Si existe un nodo terminal que no es hoja, ese nodo representa un número que a su vez es prefijo de otro.
- En caso contrario, todos los nodos terminales son hojas, por lo que es consistente.

Hashing

Cuando escribimos un número en la vida cotidiana, lo escribimos como un string.

Este mapeo entre string y número se puede pensar como un polinomio. Por ejemplo, el número 365 se representa $3\times B^2 + 6\times B + 5$, con base $B = 10$.

Lo mismo se puede pensar para strings en general: si pensamos las letras como números ($a = 1$, $b = 2$, etc.), los caracteres serán los coeficientes del polinomio. Si usamos una base B mayor a $|\Sigma|$, cada string tendrá asociado un número único.

Este número asociado a un string es su **hash**.

Hashing

Sea un string s y sea H su hash. Si agregamos un caracter c al final de s, su hash pasa a ser $H \times B + c$. Con esta idea podemos calcular el hash de todos los prefijos de un string de largo n en $O(n)$.

Si tengo el hash de $s[0, i)$ y el hash de $s[0, j)$, con una cuentita puedo también calcular el hash de $s[i, j)$. Es decir, con los hashes de todos los prefijos obtengo el hash de cualquier substring en $O(1)$.

Hashing

La idea de usar los hashes es que permitirían ver si dos strings son iguales simplemente viendo un número en vez de iterar letra por letra cada vez.

Así como lo vimos, los números pueden ser enormes, por lo que la comparación seguiría siendo lenta o nos puede dar overflow.

Para resolverlo, calculamos todos los valores módulo un M primo.

Pero ahora los hashes no son únicos, puede haber colisiones. Para reducir la probabilidad de esto, usamos 1 módulo primo de orden 10^{18} o 2 módulos primos diferentes de orden 10^{9}. Si se hace eso, la probabilidad de que $H_s = H_t$ con $s \neq t$ es muy chica (en la práctica no pasa).

Hashing - Implementación

Nuestra implementación usa 2 módulos primos diferentes de orden 10^9 y permite hallar en $O(1)$ el hash de cualquier substring de un string dado, con $O(n)$ precomputo. Para la mayoría de problemas no hace falta saber cómo está implementado.

Problema - String Matching

Dado un texto t y un patrón p, encontrar el número de ocurrencias de p en t.

- Input:
  - $t =$ `"abbaabccaab"`
  - $p =$ `"aab"`
- Output:
  - `2`

Problema - String Matching

Solución bruta:

- Para cada posición de t, me fijo caracter a caracter si matchea con p y cuento cuantos matches hay.
- La complejidad de cada comparación es $O(|p|)$ en el peor caso.
- La complejidad total $O(|t| \times |p|)$. Para este problema es muy lenta, dá **TLE**.

Problema - String Matching

Solución con hashing:

- Calculo el hash de todos los substrings de t y el hash de p.
- Itero por cada substring de t de largo $|p|$. Para cada uno, comparo su hash con el de p.
- La respuesta es el número de veces que la igualdad se cumplió.
- La complejidad es $O(|t|+|p|)$, ya que cada comparación es $O(1)$.

Hashing - Comparación Lexicográfica

Además de comparar strings por igualdad eficientemente, hashing permite también comparar strings lexicográficamente.

Si tengo dos strings s de largo n y t de largo m, la comparación lexicográfica normal tarda $O(\min(n, m))$: de a 1 busco el primer caracter donde difieren y lo comparo.

Con hashing, puedo hacer búsqueda binaria para encontrar el máximo prefijo donde coinciden en $O(\log(\min(n, m)))$, y luego comparar en $O(1)$ en el primer caracter donde difieren.

Problema - Finding Periods

Dado un string s de largo $n \leq 10^6$, hallar todos sus períodos. Un período de un string es un prefijo del mismo que puede ser usado para generar todo el string, con la última repetición posiblemente parcial.

- Input:
  - `"abcabcabcab"`
- Output:
  - `"abc"`
  - `"abcabc"`
  - `"abcabcabc"`
  - `"abcabcabcab"`

Problema - Finding Periods

Solución posible con hashing:

- Precalculo el hash de todos los substrings en $O(n)$.
- Para cada largo k entre 1 y n, chequeo que el hash de todos los substrings que arrancan en un índice múltiplo de k son iguales entre sí.
  - para $k = 1$ tengo que chequear n hashes.
  - para $k = 2$ tengo que chequear $n/2$ hashes.
  - …
  - para $k = q$ tengo que chequear n/q hashes.
  - …
- La complejidad de estos es $O(n + n/2 + ... + n/n) = O(n \times \log(n))$.
- Para $n \sim 10^6$ esta complejidad es lenta.

Problema - Finding Periods

Solución más eficiente con hashing:

- Precalculo el hash de todos los substrings en $O(n)$.
- Para cada largo k entre 1 y n, chequeo que el hash del prefijo de largo $n\text{-}k$ es igual al hash del sufijo de largo $n\text{-}k$. Si son iguales, los primeros k caracteres forman un periodo de s.
- La complejidad total queda $O(n)$.

Función Z

Dado un string s, definimos el arreglo z tal que $z[i]$ es la longitud del prefijo más largo de s que a su vez empieza en el índice i.

Este arreglo se puede calcular en $O(n)$ con la función Z.

Por ejemplo, para el string `"abaacabaad"`, su arreglo z asociado sería:

a	b	a	a	c	a	b	a	a	d
10	0	1	1	0	4	0	1	1	0

Problema - Finding Periods

Solución con función Z:

- Calculo el arreglo Z de todo el string.
- Para cada índice k, si $z[k] = n\text{-}k$ (es decir, el sufijo que empieza en k coincide con el inicio del string), entonces el prefijo de largo k es un período.
- La complejidad es $O(n)$ (precálculo del arreglo Z + chequeo de cada índice).

Problema - Password

Dado un string s de largo $n \leq 10^6$, hallar el substring más largo que es prefijo de s, sufijo de s, y que también aparece en algún lugar dentro de s, o decir que tal substring no existe.

- Input:
  - `"fixprefixsuffix"`
- Output:
  - `"fix"`

Problema - Password

Solución con función Z:

- Calculo el array Z de s.
- Itero de izquierda a derecha:
  - Estando en el índice i, si $i+z[i] = n$, entonces el prefijo de largo $z[i]$ es también sufijo.
  - A su vez, si existe un $0 < j < i$ tal que $z[j] \geq z[i]$, entonces ese prefijo ya apareció antes en el string.
  - Si se cumplen ambas condiciones, encontré la respuesta. Sino, continuo al siguiente índice.
- Si para ningún índice se cumplen ambas condiciones, no hay solución.
- La complejidad es $O(n)$.

Resumen

- Trie suele servir para problemas donde se trabaja mucho con prefijos o si se tienen muchos strings en los que buscar.
- Hashing suele servir en problemas donde queremos comparar muchos strings o substrings por igualdad. Es muy flexible.
- Función Z suele servir en problemas donde hay algún patrón repetitivo que queremos detectar o si importa comparar prefijos con sufijos.
- Es muy común que un mismo problema salga con más de una de estas técnicas que vimos hoy.

Extras

- KMP: otra forma de string matching.
- Manacher: encontrar todos los palíndromos de un string.
- Aho-Corasick: extensión de trie para hallar todas las ocurrencias de un conjunto de palabras en un texto.
- Suffix Array: trabajar con todos los sufijos de un string ordenados lexicográficamente.
- Suffix Automaton: hace de todo.

Recursos

- Clases Training Camps 2018, 2022 y 2023.
- Tutorial de string hashing en Codeforces.
- Tutorial de función Z en CP-Algorithms.
- Tutorial de string searching en USACO.

Problemas

- Trie
  - Search Suggestions System
  - A Lot of Games
  - Word Combinations
  - Matching Names
- Hashing
  - Finding Borders
  - Check Transcription
  - Rummaging
- Función Z
  - Prefixes and Suffixes
  - Lazy Printing

Consultas

Cualquier duda pueden preguntarme durante esta semana o en cualquier momento a través de:

- Email: bersanofede@gmail.com
- Telegram: @Federico_Bersano
- Codeforces: Tainel