Strings

Román Castellarin

¿Qué son?

Cadenas de caracteres.

En C: `char s[100];` ← **desaconsejado**
En C++: `string s;`

A diferencia de los arrays numéricos, los strings toman valores de un conjunto **reducido** de caracteres (alfabeto), generalmente las letras minúsculas del alfabeto inglés. Esto suele ser útil.

Strings

Sea S el string "PERRITO".

La longitud se nota por |S| = 7

"PERRITO"	"PERRITO"	"PERRITO"
el prefijo	el sufijo	el substring
S[0..2]	S[5..6]	S[2..5]
S[0..3)	S[5..7)	S[2..6)

String Matching

String Matching

Dados dos strings S, T, hallar las apariciones de S en T.

A las coincidencias las llamamos *matches*, y cuando no coinciden, *mismatches*.

"ALA" aparece en "ALALADALA" en los índices: 0, 2, y 6

Solución trivial: O(|S|·|T|)
Esta solución no reutiliza ningún tipo de información aprendida sobre S o T.

String Matching

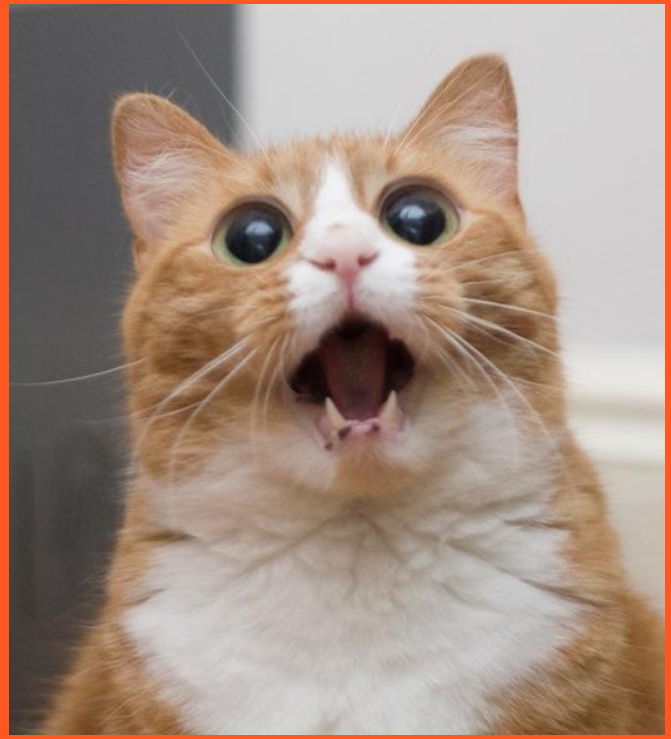
En particular, si estamos buscando "ABRACADABRALES" en un texto y venimos matcheando 11 caracteres desde la posición i ...

"...ABRACADABRA..."
[i] [i+7]

..y de repente no matchea; en lugar de intentar matchear desde i+1 podemos intentar matchear desde i+7 sabiendo que ya tenemos matcheado "ABRA".

Si de nuevo no matchea, intentamos desde i+10, sabiendo que tenemos matcheado "A"

Bordes



gato luego de aprender bordes

Borde

Un borde de un string es un substring propio que es a la vez prefijo y sufijo.

ABRACABRA → borde de largo 1
ABRACABRA → borde de largo 4

Los bordes se pueden solapar, ej:

ABRABRA → tiene un borde de largo 4
AAAAAA → tiene un borde de largo 5

Borde de un borde es un borde

¿Cómo calculamos todos los bordes de un string?

Observación: Si B_1 y B_2 son bordes de S, y $|B_1| < |B_2|$, B_1 es borde de B_2.

Ej: en "ABRACABRA", "A" es borde de "ABRA"

Visto que todos los bordes son prefijos de S, es útil calcular la longitud del máximo borde para cada prefijo de S.

Borde

Dicho esto, rellenaremos una tabla B del mismo tamaño de S tal que B[i] = longitud del máximo borde en S[0..i]

Veamos que podemos calcular B en tiempo lineal.

Observación: Si B[i] = k > 0, entonces B[i-1] = k-1. Si removemos el último carácter de un borde del prefijo i, obtenemos un borde del prefijo i-1.

ABRACADABRA → ABRACADABR

B[10] = 4 B[9] = 3

Implementación

```
b.resize(N, 0);
for(int i = 1, j = 0; i < N; ++i){
     while( j > 0 and s[j] != s[i] )
          j = b[j-1];
     if( s[j] == s[i] )
          ++j;
     b[i] = j;
 }
```

String Matching

¿Cómo hallar todas las ocurrencias de S en T en tiempo O(T) ?

Concatenar S + '$' + T, y calcular su tabla de bordes B.
($ es un carácter que no aparece ni en S ni en T)

Todos los i tales que B[i] = |S| son donde termina una aparición de S en S+T.
Es decir, hay una aparición de S en T en el índice i - 2 |S|

Problema ejemplo: compresión

Dado un string S, se quiere hallar un string t (si existe) de longitud mínima tal que S se componga de concatenaciones de t.

Es decir, S = t + t + .. + t

Problemas

- UVA # 455 "Periodic Strings"
- SPOJ - File Recover Testing
- SPOJ - Extend to Palindrome
- UVA # 11022 "String Factoring"
- UVA # 11452 "Dancing the Cheeky-Cheeky"
- UVA 12604 - Caesar Cipher
- UVA 12467 - Secret Word
-
- UVA 11019 - Matrix Matcher
- SPOJ - Pattern Find
- Codeforces - Anthem of Berland
- Codeforces - MUH and Cube Walls

Rolling Hashing

lechuza intentando aprender rolling hashing

Hashing

Una función de hash mapea una entrada de *tamaño arbitrario* a una salida de *tamaño acotado*, generalmente [0..M) para algún entero M.

Los strings pueden tener tamaño arbitrario, así que no es inusual encontrarse con funciones que *hashean* strings.

ejemplo:

```
int f(const string &s){
    return s.size() % 3;      // no usar en la vida real
}
```

String Hashing

¿Para qué? Para manipular strings rápidamente.

Manipular enteros es más simple y eficiente que manejar strings.
(O(1) vs O(N))

Podemos ver rápido si dos strings son diferentes dados sus hashes:
$f(s) \neq f(w) \Rightarrow s \neq w$
Cuando trabajamos con strings, nos gustaría también que lo contrario sea probable:
$f(s) = f(w) \Rightarrow s = w$ *probablemente*

String Hashing

Un par de strings s, w tales que $s \neq w$ pero $f(s) = f(w)$ se llama colisión.

Si f devuelve valores en [0..M) uniformemente distribuidos, la probabilidad de colisión es 1/M. *Aceptaremos los riesgos*

Tip: escoger M grande y primo, ej: $10^9 + 7$, $10^9 + 9$, etc...

Si se escogen n funciones de hash con valores en $[0..M_i)$, la probabilidad de colisión en todas es $1 / (M_1 \cdot M_2 \cdot \ldots \cdot M_n)$

String Hashing

¿Cómo convertir un string S en un número?

¡Interpretándolo como un polinomio!

Elegimos una base *p*, y los caracteres de S serán los coeficientes.

Luego, evaluamos el polinomio en módulo M.

$$f(S) = \sum_{i=0}^{|S|-1} S[i] \cdot p^i \mod M$$

por ejemplo:

f("casa") = 'c' + 'a'·p + 's'·p^2 + 'a'·p^3 mod M

String Hashing

Será de gran ayuda guardar en en un array H, los hashes para cada sufijo de S, es decir:

H[i] = h(S[i..])

para S = "casa" tenemos:

$H[0] = 'c' + 'a' \cdot p + 's' \cdot p^2 + 'a' \cdot p^3 \mod M$

$H[1] = 'a' + 's' \cdot p + 'a' \cdot p^2 \mod M$

$H[2] = 's' + 'a' \cdot p \mod M$

$H[3] = 'a' \mod M$

String Hashing

Podemos calcular H a partir de S en O(N):

```
H[i] = (S[i] + H[i+1] * p) % M;
```

Notemos que ahora podemos calcular el hash de cualquier substring de S en O(1):

$h(S[i..j]) = (H[i] - H[j] \cdot p^{j-i}) \bmod M$

Podemos precalcular $p^n \bmod M$ para todo n en O(N) usando la misma idea

String Hashing

Ejemplo.

$$H[0] = \text{'p'} + \text{'e'}\cdot p + \text{'r'}\cdot p^2 + \text{'r'}\cdot p^3 + \text{'i'}\cdot p^4 + \text{'t'}\cdot p^5 + \text{'o'}\cdot p^6 \mod M$$

$$H[1] = \text{'e'} + \text{'r'}\cdot p + \text{'r'}\cdot p^2 + \text{'i'}\cdot p^3 + \text{'t'}\cdot p^4 + \text{'o'}\cdot p^5 \mod M$$

$$H[5] = \text{'t'} + \text{'o'}\cdot p \mod M$$

$$f(\text{"erri"}) = f(S[1..5)) = H[1] - H[5] \cdot p^4 \mod M$$

Implementación

```
struct stringhash{
    stringhash(const string &s, ll BASE){
        H.resize(s.size() + 1, 0);
        P.resize(s.size() + 1, 1);
        forn(i, s.size()) P[i+1] = (P[i] * BASE) % MOD;
        rforn(i, s.size())
            H[i] = (s[i] - 'A' + 1 + H[i+1] * BASE) % MOD;
    }
    int hash(int i, int j){
        return ((H[i] - (ll) H[j] * P[j-i]) % MOD + MOD) % MOD;
    }
    vector<int> H, P;
};
```

Uso

```
stringhash s1("PERRITO", 31);
stringhash s2("ERRI", 31);

cout << s1.hash(1, 5) << endl;    // == hash(ERRI)
cout << s2.hash(0, 4) << endl;    // == hash(ERRI)
```

Aplicaciones

- String Matching de $S_1 \dots S_k$ en T en $O(S_1 + S_2 + \dots + S_k + T)$, con S_i de la misma longitud.

- Comparar substrings (lexicográficamente) en O(log N).
  - Rotación lexicográficamente mínima en O(N log N)

- Verificar si un substring es palíndromo en O(1)
- Contar substrings únicas de una string en $O(N^2 \log N^2)$

Problemas

TODO

Tries

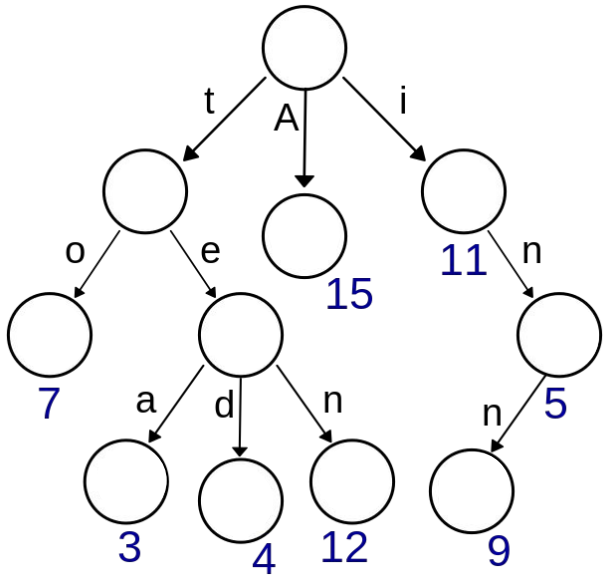
ardilla en shock luego de aprender tries

Tries

Un Trie es un árbol con raíz donde cada arco está etiquetado con un carácter tal que cada nodo representa al string que se obtiene de mirar el camino desde la raíz a él:

Ocasionalmente, podemos guardar datos asociados a cada string.

“A” → 15,
“to” → 7,
“tea” → 3,
“ted” → 4,
“ten” → 12,
“i” → 11,
“in” → 5,
“inn” → 9



Operaciones

- **insertar**(clave, valor) : asocia el valor a la clave
- **buscar**(clave) : devuelve el valor asociado a la clave
  (o indica que no se encuentra)
- **eliminar**(clave) : elimina la entrada

Vamos a centrarnos en las claves e ignoraremos los valores.

En cada nodo guardaremos la lista de arcos y un contador de cuántas claves finalizan en dicho nodo.

Implementación

```
struct trie{
    int final = 0;
    map<char, trie> arc;

    void insert(const string &s, int i = 0){
        if( s[i] ) arc[s[i]].insert(s, i+1);
        else       final++;
    }
    trie* find(const string &s, int i = 0){
        if( s[i] )
            return arc.count(s[i]) ? arc[s[i]].find(s, i+1) : nullptr;
        return final ? this : nullptr;
    }
};
```

Mejoras

Podemos guardar información extra en los nodos, por ejemplo:

- Contador de apariciones (cuántos descendientes)
- Máximo/Mejor string dicho prefijo
- Puntero al padre
- Puntero al siguiente vértice final lexicográficamente hablando:

```
trie* findnext(trie *s=nullptr){
    for(auto i = arc.rbegin(); i != arc.rend(); i++)
        s = i->second.findnext(s);
    sig = s;
    return final ? this : sig;
}
```

suffix tries

Suffix Trie

Insertar un string en un trie es O(N).

Insertar todos sus prefijos es O(N) también, es cuestión de marcar los vértices como finales a medida que se los recorre.

Por lo tanto, es posible armar un Trie de todos los substrings para un string dado simplemente insertando todos los sufijos en el trie en $O(N^2)$.

Aplicaciones

- Cantidad de apariciones de un substring S en O(S)
  (+ construir el trie)

- Substring más largo que se repite al menos K veces en $O(N^2)$
  (+ construir el trie)

- Substring más largo que aparezca en K de N palabras $S_1..S_N$ en tiempo cuadrático $O(S_1^2 + .. + S_N^2)$

- Stack/Queue persistente

Problemas

TODO