# Strings

Mariano Crosetti

Rosario, Argentina
Universidad Nacional de Rosario

Strings Nivel Avanzado

# Contenidos I

1. Repaso Hashing
   - Hashing de Rabin Karp
2. Repaso Función Z y KMP
   - Función Z
   - Bordes
3. Suffix Array + Longest Common Prefix
   - Suffix Array
   - Longest Common Prefix
   - Relación con Suffix Tree
   - Problemas con SA + LCP

# Contenidos

# Hashing de Rabin Karp

- Tener una función $hash : String \rightarrow Integer$ inyectiva nos permite usar para comparar por igualdad: $hash(s) = hash(s') \implies s = s'$
- Sea $S = s_0 s_1 s_2 .. s_{n-1}$ podemos pensarla como un polinomio:

$$hash(s) = \sum_{i=0}^{n-1} s_i . X^i$$

  - Podemos pensar a las letras $s_i$ como números (0, 1, 2 ..) en el orden del alfabaeto $\sum$.
  - Ejemplo: $BAC \rightarrow 2X^2 + 1X + 3$
  - Si $X > |\sum|$ cada string tendrá un número único. Luego la función será inyectiva y será una ideal función de hash.

# Propiedades Hashing de Rabin Karp

- Si agregamos un caracter a la derecha:
  $hash(S + c) = hash(S) + c.X^{|S|}$
- Si agregamos un caracter a la izquierda:
  $hash(c + S) = c + hash(S).X$
- Si cambiamos $s_i$ por c:
  $hash(S') = hash(S) - s_i.X^i + c.X^i$
- Si concatenamos dos string:
  $hash(S + S') = hash(S) + hash(S').X^{|S|}$
- Las operaciones anteriores son $O(1)$ teniendo precalculadas las potencias de $X$.
- Problema: números enormes: overflow.
  - Usar módulo P para algún primo P.
  - Tomar X un primo $> |\sum|$.
  - Para el módulo usar 3 primos **al azar** entre $10^9$ y $2{,}10^9$.
    La chance de colisión es despreciable.

# Comparando substring con Rabin Karp

Sea $S$ una cadena definiremos:

- Precomputamos en $O(N)$:
  $T[i] = hash(S[0,i))$.
  - Esto lo podemos hacer agregando caracteres a derecha en $O(1)$.
- Definimos nuestro hash de substrings:
  $hash'(S[i,j]) = (T[j] - T[i]).X^{|S|-i}$
  $T[i] = s_0 + s_1.X + ... + s_{i-1}.X^{i-1}$
  $T[j] = s_0 + s_1.X + ... + s_{i-1}.X^{i-1} + s_i.X^i + s_{i+1}.X^{i+1} + .. + s_{j-1}.X^{j-1}$
  $hash'(S[i,j]) = s_i.X^{|S|} + s_{i+1}.X^{|S|+1} + .. + s_{j-1}.X^{|S|+j-1-i}$
- $hash'$ nos da un hash de una substring en $O(1)$ (no depende de $i$).
  - Luego $S[i,j] = S[a,b] \Leftrightarrow hash'(S[i,j]) = hash'(S[a,b])$.
  - Podemos comparar dos substring en $O(1)$.
- Debemos precomputar $T$ y las potencias de $X$. ($O(N)$).

# Problemas con Hashing

- LCP en $O(log\ N)$ (arreglo $Z$ en $O(N\ log\ N)$).
- Dada una cadena $T$ calcular la cantidad de cadenas distintas de tamaño $K$.
- Búsqueda en $O(1)$ de varios $S_1$, $S_2$, ... , $S_n$ de una misma longitud dentro de $T$.
- Algoritmo de Manacher en $O(N\ log\ N)$.
- Máximo substring común entre varios string en $O(N\ log\ N)$.

# Contenidos

# Función Z

Dada una cadena $S$

- $lcp(i,j)$ es la cantidad de coincidencias, si empezamos a comparar en $i$ y en $j$.
- $lcp(i,j)$ es *longest common preffix* de los sufijos que empiezan en $i$ y $j$.
- $z[i] = lcp(i,0)$

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| S[i] | b | a | m | b | a | m | b | a | b |
| Z[i] | 0 | 0 | 0 | 5 | 0 | 0 | 2 | 0 | 1 |

# Función Z

Dada una cadena $S$

- $lcp(i, j)$ es la cantidad de coincidencias, si empezamos a comparar en $i$ y en $j$.
- $lcp(i, j)$ es *longest common preffix* de los sufijos que empiezan en $i$ y $j$.
- $z[i] = lcp(i, 0)$

| S[i] | b | a | m | b | a | m | b | a | b |
|---|---|---|---|---|---|---|---|---|---|
| Z[i] | 0 | 0 | 0 | 5 | 0 | 0 | 2 | 0 | 1 |

- se puede calcular en $O(N\ log\ N)$ con hashing.

# Función Z

Dada una cadena $S$

- $lcp(i, j)$ es la cantidad de coincidencias, si empezamos a comparar en $i$ y en $j$.
- $lcp(i, j)$ es *longest common preffix* de los sufijos que empiezan en $i$ y $j$.
- $z[i] = lcp(i, 0)$

| S[i] | b | a | m | b | a | m | b | a | b |
|---|---|---|---|---|---|---|---|---|---|
| Z[i] | 0 | 0 | 0 | 5 | 0 | 0 | 2 | 0 | 1 |

- se puede calcular en $O(N\ log\ N)$ con hashing.
- se puede calcular en $O(N)$.

# Función Z - código

```
char s[ N ];
// z[i] = i==0 ? 0 : max k tq s[0,k) match with s[i,i+k)
int z[ N ];
int n = strlen(s);
forn(i, n) z[i]=0;
for (int i = 1, l = 0, r = 0; i < n; ++i) {
    if (i <= r) z[i] = min (r - i + 1, z[i - l]);
    while (i + z[i] < n && s[z[i]] == s[i + z[i]])  ++z[i];
    if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
}
```

# Problemas con Función Z

Dada una cadena $S$

- Dado $P$ y $T$ encontrar todas las apariciones de $P$ en $T$.
- Verificar si una cadena $S_1$ es rotacion de otra $S_2$.

# Contenidos

# Bordes

Dada una cadena $S$, se le llama un **borde** de $S$, a un prefijo que es al mismo tiempo sufijo.

- Con Z function: el sufijo $[i, n)$ es borde sí y sólo sí $z[i] = n - i$.
- **Tabla KMP**: para cada prefijo $[0, i)$ guardamos el borde máximo.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| S[i] | b | a | m | b | a | m | b | a | b | |
| KMP[i] | -1 | 0 | 0 | 0 | 1 | 2 | 3 | 4 | 5 | 1 |

  - Se puede generar con arreglo z de $S\ S^T$.

```
string P; //cadena a buscar(what)
int b[MAXLEN]; //back table b[i] maximo borde de [0..i)
void kmppre(){
    int i =0, j=-1; b[0]=-1;
    while(i<sz(P)){
        while(j>=0 && P[i] != P[j]) j=b[j];
        i++, j++, b[i] = j;
    }
}
```

# Bordes aplicación

- Teniendo la Tabla KMP del patrón $P$ podemos buscarlo en el texto $T$ en $O(|T|)$.

```
void kmp(){
	int i=0, j=0;
	while(i<sz(T)){
		while(j>=0 && T[i]!=P[j]) j=b[j];
		i++, j++;
		if(j==sz(P)) printf("P is found at index %d in T\n", i
			-j), j=b[j];
	}
}
```

# Contenidos

# Suffix Array

El **Suffix Array** es una estructura que nos presenta los sufijos de una string ordenados.

- $SA[i]$ es una permutación de los índices $[0, n)$.
- $SA[i]$ representa el sufijo $[SA[i], n)$.
- Se encuentran ordenados lexicográficamente.

# Suffix Array

| i | Suffix |
|---|---|
| 0 | GATAGACA$ |
| 1 | ATAGACA$ |
| 2 | TAGACA$ |
| 3 | AGACA$ |
| 4 | GACA$ |
| 5 | ACA$ |
| 6 | CA$ |
| 7 | a$ |
| 8 | $ |

| i | SA[i] | Suffix |
|---|---|---|
| 0 | 8 | $ |
| 1 | 7 | A$ |
| 2 | 5 | ACA$ |
| 3 | 3 | AGACA$ |
| 4 | 1 | ATAGACA$ |
| 5 | 6 | CA$ |
| 6 | 4 | GACA$ |
| 7 | 0 | GATAGACA$ |
| 8 | 2 | TAGACA$ |

# Computando Suffix Array

- Ordenar todos los prefijos tardaría $O(N^2\ log\ N)$
- Calcularemos $R[i][k]$ como "el valor" del substring $S[i, min(n, i + 2^k))$.
  Tal que
  $R[i][k] \leq R[j][k] \Leftrightarrow S[i, min(n, i + 2^k)) \leq S[j, min(n, j + 2^k))$
- $R[i][0] = S[i]$
- Para computar $k + 1$ ordenamos $SA[i] = i$ por $SF[i] = (R[i][k], R[i + 2^k][k])$.
  Luego definimos $r = R[SA[0]][k + 1] = 0$
  Luego iremos recorriendo el $SA$ ordenado.
  $R[SA[i]][k + 1] = SF[SA[i]]! = SF[SA[i - 1]]?r + + : r$
- Complejidad $O(N\ log^2\ N)$

# Computando Suffix Array

```
pair<int, int> sf[MAXN];
bool comp(int lhs, int rhs) {return sf[lhs] < sf[rhs];}
int sa[MAXN], r[MAXN];
forn(i, n) r[i] = a[i];
for(int m = 1; m < n; m <<= 1) {
  forn(i, n) sa[i]=i, sf[i] =
                    make_pair(r[i], i + m < n? r[i + m] : -1);
    stable_sort(sa, sa+n, comp);
    r[sa[0]] = 0;
    forr(i, 1, n) r[sa[i]]= sf[sa[i]] != sf[sa[i - 1]] ?
                                   i : r[sa[i-1]];
}
```

# Contenidos

# Longest Common Prefix

Definimos $LCP[i]$ como la longitud del prefijo común más largo entre $SA[i]$ y $SA[i-1]$ ($LCP[0] = 0$ por definición).

| i | SA[i] | LCP[i] | Suffix |
|---|---|---|---|
| 0 | 8 | 0 | $ |
| 1 | 7 | 0 | A$ |
| 2 | 5 | 1 | **A**CA$ |
| 3 | 3 | 1 | **A**GACA$ |
| 4 | 1 | 1 | **A**TAGACA$ |
| 5 | 6 | 0 | CA$ |
| 6 | 4 | 0 | GACA$ |
| 7 | 0 | 2 | **GA**TAGACA$ |
| 8 | 2 | 0 | TAGACA$ |

# Construyendo LCP

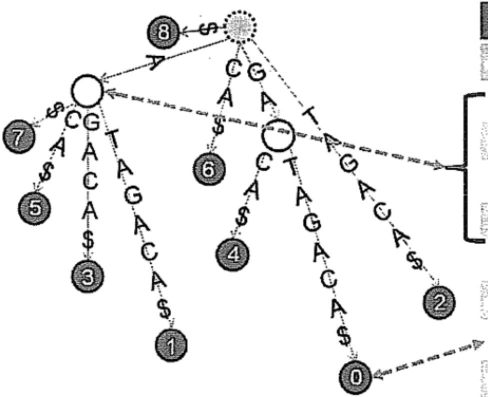
Para resolver LCP la idea es que si por ejemplo el LCP de *hola* y *hongo* es 2, entonces para calcular el LCP de *chola* y *chongo* es 3, y sólo hace falta mirar un caracter.

```
int LCP[N], phi[N], PLCP[N];
phi[sa[0]]=-1;
forr(i, 1, n) phi[sa[i]]=sa[i-1];
int L=0;
forn(i, n){
   if(phi[i]==-1) {PLCP[i]=0; continue;}
   // Modificar aca para tener varios terminales distintos.
   while(s[i+L]==s[phi[i]+L]) L++;
  PLCP[i]=L;
  L=max(L-1, 0);
}
forn(i, n) LCP[i]=PLCP[sa[i]];
```

# Contenidos

# Relación con Suffix Tree

- Rango en Suffix Array $\leftrightarrow$ Vértice interno Suffix Tree.
- Índice de Suffix Array $\leftrightarrow$ Vértice terminal Suffix Tree.



| i | SA[i] | Suffix |
|---|---|---|
| 0 | 8 | $ |
| 1 | 7 | A$ |
| 2 | 5 | ACA$ |
| 3 | 3 | AGACA$ |
| 4 | 1 | ATAGACA$ |
| 5 | 6 | CA$ |
| 6 | 4 | GACA$ |
| 7 | 0 | GATAGACA$ |
| 8 | 2 | TAGACA$ |

Figure 6.8: Suffix Tree and Suffix Array of T = 'GATAGACA$'

# Relación con Suffix Tree (cont.)

- $LCP[i] \leftrightarrow$ Distancia a la raíz del LCA de los vértices respectivos a SA[i] y SA[i-1] en el Suffix Tree.
- $min_{i<k\leq j}LCA[k] \leftrightarrow$ Distancia a la raíz del LCA de los vértices respectivos a SA[i] y SA[j] en el Suffix Tree.

Muchos problemas son fáciles de pensar dados el Suffix Tree. Suffix Tree se puede calcular en $O(N)$ tiempo y memoria.
Pero Suffix Array es una estructura mucho más fácil de codear y manipular en competencia.
Muchos problemas pueden salir pensándolo en el Suffix Tree y usando las equivalencias anteriores idear una solución para Suffix Array.

# Contenidos

# Problemas con SA + LCP

- Buscar $P$ en $T$ en $O(|P| \; log \; |T|)$.
- Longest Repeated Substring.
- Longest Repeated Substring. Idea: concatenar una string consigo misma y ver SA.
- Longest Common Substring. Idea: concatenar las dos string y ver LCP.

## Nota de implementación

- Cuando implementamos Suffix Array solemos usar un terminador como '$' o '#', un caracter más livianos que todo el alfabeto.
- Si concatenamos $N$ cadenas $S_1$, $S_2$, ... $S_N$ solemos separarlas por terminadores ($S_1\$S_2\$...\$S_N\$$).
  Generalmente queremos que los terminadores sean distintos (para que el *LCP* sólo no se extienda entre cadenas). Para eso podemos modificar el código de *LCP* y definir cuándo se comparan caracteres que $ son distintos.

# Problemas con SA + LCP

- Buscar $P$ en $T$ en $O(|P| \ log \ |T|)$. Idea: binary search en SA.
- Longest Repeated Substring.
- Longest Repeated Substring.
- Longest Common Substring. Idea: concatenar las dos string y ver LCP.

## Nota de implementación

- Cuando implementamos Suffix Array solemos usar un terminador como '\$' o '#', un caracter más livianos que todo el alfabeto.
- Si concatenamos $N$ cadenas $S_1$, $S_2$, ... $S_N$ solemos separarlas por terminadores ($S_1\$S_2\$...\$S_N\$$).
  Generalmente queremos que los terminadores sean distintos (para que el *LCP* sólo no se extienda entre cadenas). Para eso podemos modificar el código de *LCP* y definir cuándo se comparan caracteres que \$ son distintos.

# Problemas con SA + LCP

- Buscar $P$ en $T$ en $O(|P| \, log \, |T|)$. Idea: binary search en SA.
- Longest Repeated Substring. Idea: ver LCP.
- Longest Repeated Substring.

- Longest Common Substring.

### Nota de implementación

- Cuando implementamos Suffix Array solemos usar un terminador como '\$' o '#', un caracter más livianos que todo el alfabeto.
- Si concatenamos $N$ cadenas $S_1$, $S_2$, ... $S_N$ solemos separarlas por terminadores ($S_1\$S_2\$...\$S_N\$$).
  Generalmente queremos que los terminadores sean distintos (para que el *LCP* sólo no se extienda entre cadenas). Para eso podemos modificar el código de *LCP* y definir cuándo se comparan caracteres que \$ son distintos.

# Problemas con SA + LCP

- Buscar $P$ en $T$ en $O(|P| \log |T|)$. Idea: binary search en SA.
- Longest Repeated Substring. Idea: ver LCP.
- Longest Repeated Substring. Idea: concatenar una string consigo misma y ver SA.
- Longest Common Substring.

## Nota de implementación

- Cuando implementamos Suffix Array solemos usar un terminador como '\$' o '#', un caracter más livianos que todo el alfabeto.
- Si concatenamos $N$ cadenas $S_1$, $S_2$, ... $S_N$ solemos separarlas por terminadores ($S_1\$S_2\$...\$S_N\$$).
  Generalmente queremos que los terminadores sean distintos (para que el *LCP* sólo no se extienda entre cadenas). Para eso podemos modificar el código de *LCP* y definir cuándo se comparan caracteres que \$ son distintos.

# Problemas con SA + LCP

- Buscar $P$ en $T$ en $O(|P| \; log \; |T|)$. Idea: binary search en SA.
- Longest Repeated Substring. Idea: ver LCP.
- Longest Repeated Substring. Idea: concatenar una string consigo misma y ver SA.
- Longest Common Substring. Idea: concatenar las dos string y ver LCP.

## Nota de implementación

- Cuando implementamos Suffix Array solemos usar un terminador como '\$' o '#', un caracter más livianos que todo el alfabeto.
- Si concatenamos $N$ cadenas $S_1$, $S_2$, ... $S_N$ solemos separarlas por terminadores ($S_1\$S_2\$...\$S_N\$$).
  Generalmente queremos que los terminadores sean distintos (para que el *LCP* sólo no se extienda entre cadenas). Para eso podemos modificar el código de *LCP* y definir cuándo se comparan caracteres que \$ son distintos.