# Hashing y prefix function

Carlos Lagos

carlos.lagos@progcomp.cl

carlos.lagosc@usm.cl

# String Matching

Dado un string $T$ y un patrón $S$, tu tarea es contar la cantidad de posiciones donde el patrón aparece en el string.

Ejemplo:

```
S = car
T = blacarcarbla
```

Aparece en dos posiciones distintas.

# KMP - Prefix function

> $T[i:j]$ representa el substring de $T$ que parte en $i$ y termina en $j$.

Se define el arreglo $\pi$ de un string $S$ de la siguiente forma:

$\pi[i] =$ Longitud del mayor prefijo propio (es decir, un prefijo que no es todo el string) que también es sufijo de $S[0:i]$

**Ejemplo:**

| $i$ | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** |
|---|---|---|---|---|---|---|---|---|---|---|
| $s[i]$ | b | a | b | a | b | a | c | b | b | a |
| $\pi[i]$ | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | 1 | 0 |

## String Matching con KMP

Podemos calcular el arreglo $\pi$ del string $S\#T$, donde $\#$ es un carácter que no aparece ni en $S$ ni en $T$.

$$P = S + \# + T$$

Luego se calcula el arreglo $\pi$ del string $P$. Si en alguna posición se cumple que $\pi[i] = |S|$, entonces $S$ aparece como substring de $T$.

## Ejemplo

Supongamos que tenemos los siguientes strings:

```
S = car
T = blacarcarbla
```

| $i$ | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $s[i]$ | c | a | r | # | b | l | a | c | a | r | c | a | r | b | l | a |
| $\pi[i]$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 3 | 1 | 2 | 3 | 0 | 0 | 0 |

# KMP - Prefix function

- El arreglo $\pi$ se puede construir en tiempo $O(n)$ de forma amortizada, donde n es el largo del string o vector s.
- Si ya tenemos calculado el valor $\pi[i-1]$, hay tres posibles casos para $\pi[i]$:
  - El carácter s[i] coincide con el carácter siguiente al sufijo actual ($s[\pi[i-1]]$), entonces podemos extender ese sufijo y asignamos $\pi[i] = \pi[i-1] + 1$.
  - Si no coincide, buscamos el siguiente mejor sufijo posible repitiendo el proceso con $\pi[\pi[i-1]-1]$, y así sucesivamente hasta encontrar un sufijo que pueda extenderse, o llegar a cero.
  - Si no se puede extender ningún sufijo, entonces $\pi[i] = 0$.
- Esta lógica hace que en la construcción del arreglo siempre intentemos extender primero el sufijo más largo posible. Si no es válido, usamos los valores ya calculados para retroceder eficientemente y probar con sufijos más cortos.
- Implementación: https://miniurl.cl/qhbybm

Una función de hash es una función que toma un objeto y lo transforma en un número entero, que actúa como una especie de identificador del objeto.

La idea es que si hasheamos dos strings $P$ y $Q$, y sus *hashes* son iguales, entonces (con alta probabilidad) los strings también lo son.

# Hashing

La función de *hash* comúnmente usada consiste en interpretar el string como un número en una base determinada (mayor al tamaño del alfabeto) y convertir ese número a decimal.

Asignamos a cada letra un valor:
`a = 1`, `b = 2`, `c = 3`, ..., y el vacío = 0.

Entonces el alfabeto tiene al menos $26 + 1 = 27$ símbolos, por lo que elegimos una base $B \geq 27$.

Si lo convertimos a decimal, obtenemos:

$$H(S) = h \cdot B^3 + o \cdot B^2 + l \cdot B^1 + a \cdot B^0$$

- **¿Qué sucede si el string es muy grande?**
  - Redefinimos la función de *hash* para que opere **módulo** $m$.
    Esto evita que los valores crezcan demasiado y permite trabajar con enteros manejables.
- **¿Podría ocurrir una colisión?**
  - Sí, pero se puede reducir su probabilidad.
    Usar una **base** $B$ (como $37$) y un **módulo** $m$ (como $10^9 + 7$) que sean primos ayuda a distribuir mejor los hashes.
    La probabilidad de colisión en una comparación es aproximadamente: $\frac{1}{m}$
- **¿Qué pasa si hago muchas comparaciones?**
  - Podemos usar un segundo par de valores $(B_2, m_2)$ distintos.
    Comparamos ambos hashes.
    La probabilidad de colisión conjunta es: $\frac{1}{m_1 \cdot m_2}$.

## Hash de un substring

Para calcular el hash de cualquier substring, primero precalculamos los hashes de todos los prefijos del string:

$$\begin{aligned}
\text{pre}[0] &= a_0 \cdot B^0 = a_0 \\
\text{pre}[1] &= \text{pre}[0] \cdot B + a_1 \\
\text{pre}[2] &= \text{pre}[1] \cdot B + a_2 \\
&\vdots \\
\text{pre}[n-1] &= \text{pre}[n-2] \cdot B + a_{n-1}
\end{aligned}$$

## Hash de un substring

Una vez precalculado el arreglo `pre`, el hash del substring $S[l:r]$ se puede obtener en tiempo constante con:

$$H(S[l:r]) = \text{pre}[r] - \text{pre}[l-1] \cdot B^{r-l+1}$$

> Si $l = 0$, entonces $H(S[0:r]) = \text{pre}[r]$.
> También se puede definir $\text{pre}[0] = 0$ y que $\text{pre}[i]$ guarde el hash de $S[0:i-1]$.

Con lo anterior, podemos replicar la idea del algoritmo básico, pero ahora verificar si el patrón comienza en una posición específica del texto se vuelve mucho más eficiente, logrando una complejidad total de $O(n+m)$.

Implementación: https://miniurl.cl/4m61p5

# Finding Periods

Un período de un string es un prefijo que puede usarse para generar todo el string repitiendo ese prefijo. La última repetición puede ser parcial. Por ejemplo, los períodos de $abcabca$ son $abc$, $abcabc$ y $abcabca$.

Tu tarea es encontrar todas las longitudes de los períodos de un string.

Link problema: https://cses.fi/problemset/task/1733
Link solución: https://cses.fi/paste/c09eb6e324fa17dad30221/

# Palíndromo en un rango

Se entrega un string $S$ y se realizan $Q$ consultas del tipo: **¿Es palíndromo el substring $S[l : r]$?**

# Palíndromo en un rango con actualizaciones

Se entrega un string $S$ y se realizan $Q$ consultas.

Además de consultar si un substring $S[l : r]$ es un palíndromo,
ahora también se permiten **actualizaciones** que modifican un carácter en una posición específica.

Link problema: https://cses.fi/problemset/task/2420
Link solución: https://cses.fi/paste/35d950097c35a92c708e95/

# MUH and Cube Walls

Los osos polares Menshykov y Uslada del zoológico de San Petersburgo, y el elefante Horace del zoológico de Kiev, consiguieron muchos cubos de madera de algún lugar. Comenzaron a construir torres apilando los cubos uno sobre otro. Definieron que múltiples torres alineadas forman una **muralla**. Una muralla puede estar compuesta por torres de diferentes alturas.

Horace fue el primero en terminar su muralla. La llamó un **elefante**. Su muralla consiste en **w** torres. Los osos también terminaron su muralla, pero no le pusieron nombre. La muralla de ellos consiste en **n** torres. Horace miró la muralla de los osos y se preguntó: ¿en cuántas partes de la muralla puede "ver un elefante"?

Él puede "ver un elefante" en un segmento de **w** torres contiguas si las alturas de esas torres coinciden, como secuencia, con las alturas de las torres en la muralla de Horace.

Para poder ver la mayor cantidad de elefantes posible, Horace puede **subir o bajar** su muralla. Incluso puede bajarla por debajo del nivel del suelo (ver las imágenes de los ejemplos para más claridad).

**Tu tarea** es contar cuántos segmentos de la muralla de los osos permiten a Horace "ver un elefante".