Hashing

Miguel Raggi

Algoritmos
Escuela Nacional de Estudios Superiores
UNAM

14 de mayo de 2019

Índice:

Índice:

Motivación

Problema

Tenemos dos arreglos de números A y B. Ambos son gigantes (tamaño n). ¿Cómo encuentro la intersección $A \cap B$?

Motivación

Problema

Tenemos dos arreglos de números A y B. Ambos son gigantes (tamaño n). ¿Cómo encuentro la intersección $A \cap B$?

Veremos 3 soluciones para este problema.

Motivación

Problema

Tenemos dos arreglos de números A y B. Ambos son gigantes (tamaño n). ¿Cómo encuentro la intersección $A \cap B$?

Veremos 3 soluciones para este problema. Una de $O(n^2)$, otra $O(n \log(n))$ y otra $O(n)$.

Soluciónes

Sol tonta:

- Para cada $a \in A$, revisamos los elementos de B y vemos si está ahí.

Sol buena:

- Ordena B, y así puedo hacer búsqueda binaria para cada elemento de A

Sol mejor:

- Usa los números de B para construir una "tabla de hash", donde pueda revisar muy rápidamente si un número pertenece a B o no.

Índice:

Introducción a Hashing

- Queremos una estructura de datos para hacer un “diccionario”.

Introducción a Hashing

- Queremos una estructura de datos para hacer un “diccionario”.
- Es decir, queremos poder saber si una llave está en el “diccionario” lo más rápidamente posible, y ver sus datos satélite.

Introducción a Hashing

- Queremos una estructura de datos para hacer un “diccionario”.
- Es decir, queremos poder saber si una llave está en el “diccionario” lo más rápidamente posible, y ver sus datos satélite.
- También queremos poder insertar y quitar datos del diccionario, dadas sus llaves.

Introducción a Hashing

- Queremos una estructura de datos para hacer un "diccionario".
- Es decir, queremos poder saber si una llave está en el "diccionario" lo más rápidamente posible, y ver sus datos satélite.
- También queremos poder insertar y quitar datos del diccionario, dadas sus llaves.
- La idea básica es que podemos "convertir" el objeto que queremos guardar a un numerito de alguna manera, y luego utilizar un arreglo para guardarlo.

Introducción a Hashing

- Queremos una estructura de datos para hacer un "diccionario".
- Es decir, queremos poder saber si una llave está en el "diccionario" lo más rápidamente posible, y ver sus datos satélite.
- También queremos poder insertar y quitar datos del diccionario, dadas sus llaves.
- La idea básica es que podemos "convertir" el objeto que queremos guardar a un numerito de alguna manera, y luego utilizar un arreglo para guardarlo.
- En la práctica corre muy bien este método, pero los métodos que veremos tienen peor caso muy malo (pero caso promedio muy muy rápido).

Números Naturales vs Cadenas vs Otros Objetos

- Podemos suponer sin pérdida de generalidad que las llaves son números naturales.

Números Naturales vs Cadenas vs Otros Objetos

- Podemos suponer sin pérdida de generalidad que las llaves son números naturales.
- No siempre lo son (de hecho, muchas veces no lo son), a veces las llaves son nombres (cadenas) u otros objetos.

Números Naturales vs Cadenas vs Otros Objetos

- Podemos suponer sin pérdida de generalidad que las llaves son números naturales.
- No siempre lo son (de hecho, muchas veces no lo son), a veces las llaves son nombres (cadenas) u otros objetos.
- Pero cualquier objeto lo podemos pensar como un número natural (aunque quizás enorme):

Números Naturales vs Cadenas vs Otros Objetos

- Podemos suponer sin pérdida de generalidad que las llaves son números naturales.
- No siempre lo son (de hecho, muchas veces no lo son), a veces las llaves son nombres (cadenas) u otros objetos.
- Pero cualquier objeto lo podemos pensar como un número natural (aunque quizás enorme):
- Todo en la computadora son 0's y 1's, así que lo podemos pensar con un número enorme escrito en binario.

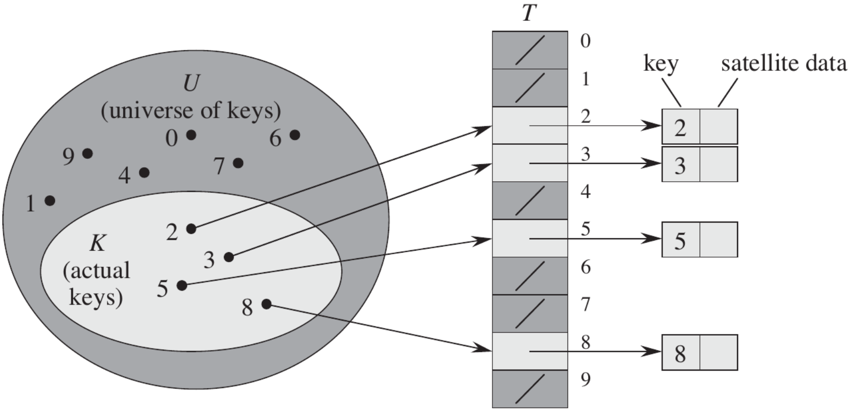
Números Naturales vs Cadenas vs Otros Objetos

- Podemos suponer sin pérdida de generalidad que las llaves son números naturales.
- No siempre lo son (de hecho, muchas veces no lo son), a veces las llaves son nombres (cadenas) u otros objetos.
- Pero cualquier objeto lo podemos pensar como un número natural (aunque quizás enorme):
- Todo en la computadora son 0's y 1's, así que lo podemos pensar con un número enorme escrito en binario.
- El universo U es el conjunto de posibles llaves, y es un conjunto de naturales: $\{0, 1, 2, 3, ..., m\}$.

Índice:

Acceso Directo

El hashing "trivial" es en el que haces un arreglo T que tenga tantos espacios como el universo:



Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.

Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.
- Sin embargo, en muchas situaciones utilizaría demasiado espacio.

Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.
- Sin embargo, en muchas situaciones utilizaría demasiado espacio.
- Por ejemplo, si tenemos cadenas de 10 caracteres o menos, ¿de qué tamaño es el universo? (y por lo tanto, ¿de qué tamaño tendría que ser el arreglo?)

Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.
- Sin embargo, en muchas situaciones utilizaría demasiado espacio.
- Por ejemplo, si tenemos cadenas de 10 caracteres o menos, ¿de qué tamaño es el universo? (y por lo tanto, ¿de qué tamaño tendría que ser el arreglo?)
- Pues cada letra tiene 256 posibilidades (un byte), así que tendrían que ser de tamaño $256^{10} = 1208925819614629174706176$, algo totalmente imposible.

Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.
- Sin embargo, en muchas situaciones utilizaría demasiado espacio.
- Por ejemplo, si tenemos cadenas de 10 caracteres o menos, ¿de qué tamaño es el universo? (y por lo tanto, ¿de qué tamaño tendría que ser el arreglo?)
- Pues cada letra tiene 256 posibilidades (un byte), así que tendrían que ser de tamaño $256^{10} = 1208925819614629174706176$, algo totalmente imposible.
- Si sabemos que el número de posibilidades es muy limitado, vale la pena, porque es muy rápido, pero si no, es imposible.

Acceso Directo: Problemas

- Esta manera de trabajar hace que sea muy rápido el diccionario: Todas las operaciones son triviales.
- Sin embargo, en muchas situaciones utilizaría demasiado espacio.
- Por ejemplo, si tenemos cadenas de 10 caracteres o menos, ¿de qué tamaño es el universo? (y por lo tanto, ¿de qué tamaño tendría que ser el arreglo?)
- Pues cada letra tiene 256 posibilidades (un byte), así que tendrían que ser de tamaño $256^{10} = 1208925819614629174706176$, algo totalmente imposible.
- Si sabemos que el número de posibilidades es muy limitado, vale la pena, porque es muy rápido, pero si no, es imposible.
- Por ejemplo, cuando vimos árbol generador de peso mínimo, para los nodos explorados utilizamos exactamente eso, porque sólo necesitamos un arreglo cuyo universo es el número de nodos.

Índice:

Tablas de Hashing

- Entonces utilizamos una función de hashing:

$$h : U \to \{0, 1, ..., n - 1\},$$

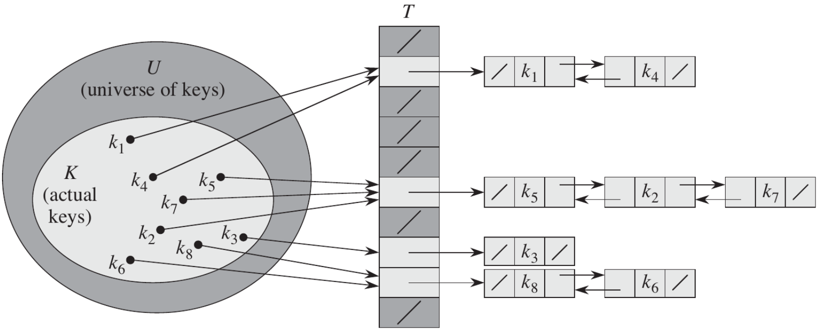
donde n es un número relativamente pequeño.

Tablas de Hashing

- Entonces utilizamos una función de hashing:

$$h : U \to \{0, 1, ..., n-1\},$$

  donde n es un número relativamente pequeño.
- Entonces guardamos así:



Tablas de Hashing

- En el arreglo T ponemos, no los objetos, sino alguna otra estructura de datos, como listas doblemente ligadas o apuntadores a arreglos o algo así.

Tablas de Hashing

- En el arreglo T ponemos, no los objetos, sino alguna otra estructura de datos, como listas doblemente ligadas o apuntadores a arreglos o algo así.
- En general queremos que las listas sean pequeñas.

Tablas de Hashing

- En el arreglo T ponemos, no los objetos, sino alguna otra estructura de datos, como listas doblemente ligadas o apuntadores a arreglos o algo así.
- En general queremos que las listas sean pequeñas.
- Cuando dos objetos que queremos guardar tienen la misma h (es decir, deben guardarse al mismo lugar en T) decimos que ocurrió una colisión.

Tablas de Hashing

- En el arreglo T ponemos, no los objetos, sino alguna otra estructura de datos, como listas doblemente ligadas o apuntadores a arreglos o algo así.
- En general queremos que las listas sean pequeñas.
- Cuando dos objetos que queremos guardar tienen la misma h (es decir, deben guardarse al mismo lugar en T) decimos que ocurrió una colisión.
- Para minimizar colisiones, queremos que todos los números entre 0 y n tengan la misma probabilidad de ocurrir.

Tablas de Hashing

- En el arreglo T ponemos, no los objetos, sino alguna otra estructura de datos, como listas doblemente ligadas o apuntadores a arreglos o algo así.
- En general queremos que las listas sean pequeñas.
- Cuando dos objetos que queremos guardar tienen la misma h (es decir, deben guardarse al mismo lugar en T) decimos que ocurrió una colisión.
- Para minimizar colisiones, queremos que todos los números entre 0 y n tengan la misma probabilidad de ocurrir.
- Como a priori no tenemos mucha información acerca de las llaves, debemos tomar una función de hashing h que tenga mucha "variedad", y que mande cosas parecidas (en términos de bits) a cosas diferentes.

Hashing: Módulos

- Una manera muy razonable de mapear es el módulo:

$$k \mapsto k \,\% m$$

para alguna m.

Hashing: Módulos

- Una manera muy razonable de mapear es el módulo:

$$k \mapsto k\,\%m$$

  para alguna m.
- El chiste es que los datos que tenemos no tengan mucho que ver con m. Por ejemplo, m no debe ser una potencia de 2, porque entonces simplemente estamos cortando un número de bits.

Hashing: Módulos

- Una manera muy razonable de mapear es el módulo:

$$k \mapsto k \,\%m$$

  para alguna m.
- El chiste es que los datos que tenemos no tengan mucho que ver con m. Por ejemplo, m no debe ser una potencia de 2, porque entonces simplemente estamos cortando un número de bits.
- Usualmente debemos tomar m como un primo que no esté cercano a una potencia de 2. Por ejemplo, $m = 701$, o algo así.

Hashing: Módulos

- Una manera muy razonable de mapear es el módulo:

$$k \mapsto k \,\% m$$

para alguna m.

- El chiste es que los datos que tenemos no tengan mucho que ver con m. Por ejemplo, m no debe ser una potencia de 2, porque entonces simplemente estamos cortando un número de bits.
- Usualmente debemos tomar m como un primo que no esté cercano a una potencia de 2. Por ejemplo, $m = 701$, o algo así.
- Un primo grande que no parezca tener nada que ver con el problema es una buena idea.

Índice:

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.
- Por ejemplo, podríamos hacer hashing y luego cuando haya muchas otras colisiones en alguna casilla, poner ahí una tabla de hashing también.

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.
- Por ejemplo, podríamos hacer hashing y luego cuando haya muchas otras colisiones en alguna casilla, poner ahí una tabla de hashing también.
- Eso es conocido como doble Hashing.

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.
- Por ejemplo, podríamos hacer hashing y luego cuando haya muchas otras colisiones en alguna casilla, poner ahí una tabla de hashing también.
- Eso es conocido como doble Hashing.
- Otra manera es tomar dos funciones de hashing y combinarlas.

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.
- Por ejemplo, podríamos hacer hashing y luego cuando haya muchas otras colisiones en alguna casilla, poner ahí una tabla de hashing también.
- Eso es conocido como doble Hashing.
- Otra manera es tomar dos funciones de hashing y combinarlas.
- O tomar una función de hashing que parezca "aleatoria" totalmente, pero que no lo sea.

Hashing Universal, Doble Hashing, Perfecto, etc.

- Hay varias otras maneras de hacer hashing para minimizar colisiones y hacer que todo sea muy rápido.
- Por ejemplo, podríamos hacer hashing y luego cuando haya muchas otras colisiones en alguna casilla, poner ahí una tabla de hashing también.
- Eso es conocido como doble Hashing.
- Otra manera es tomar dos funciones de hashing y combinarlas.
- O tomar una función de hashing que parezca “aleatoria” totalmente, pero que no lo sea.
- Etc.

Índice:

Para hacer Hashing en la Práctica

- Usualmente ya los lenguajes de programación traen una función de hashing que sirve para cualquier cosa. Por ejemplo, strings, números, etc.

Para hacer Hashing en la Práctica

- Usualmente ya los lenguajes de programación traen una función de hashing que sirve para cualquier cosa. Por ejemplo, strings, números, etc.
- A veces también traen estructuras de datos que utilizan hashing internamente para hacer las cosas más eficientes.

Para hacer Hashing en la Práctica

- Usualmente ya los lenguajes de programación traen una función de hashing que sirve para cualquier cosa. Por ejemplo, strings, números, etc.
- A veces también traen estructuras de datos que utilizan hashing internamente para hacer las cosas más eficientes.
- Por ejemplo, los diccionarios de python utilizan esto:

```python
telefonos = {'Pedro': 3543851, 'Juan': 1234871}
telefonos['Ana'] = 1419222
print(telefonos['Pedro'])
```

El programa imprime el teléfono de Pedro.

Definir hashing en nuevas clases

Cuando tu defines una nueva clase, puedes definir exactamente cómo hashear tu clase así:

```
class Hola(objeto):
  def __init__(self, x):
    self.x = x
  def __hash__(self):
    return self.x
```

Definir hashing en nuevas clases

Cuando tu defines una nueva clase, puedes definir exactamente cómo hashear tu clase así:

```
class Hola(objeto):
  def __init__(self, x):
    self.x = x
  def __hash__(self):
    return self.x
```

Si tu clase no tiene como hashear, no puede ser utilizada como llave en un diccionario. Todos los “tipos básicos” (como números, strings, etc.) ya traen como hashearse.

Veamos en C++ cómo se hashea.

Índice:

Hashing en strings

Problema

Sea s una cadena de texto. Quiero encontrar una buena manera de hacerle hashing a la cadena.

Hashing en strings

Problema

Sea s una cadena de texto. Quiero encontrar una buena manera de hacerle hashing a la cadena.

- Una manera es simplemente verla como un número gigante y luego sacarle módulo p para algún primo grande p.

Hashing en strings

Problema

Sea s una cadena de texto. Quiero encontrar una buena manera de hacerle hashing a la cadena.

- Una manera es simplemente verla como un número gigante y luego sacarle módulo p para algún primo grande p.
- Pero una mejor es la que veremos a continuación.

Hashing en strings

Problema

Sea s una cadena de texto. Quiero encontrar una buena manera de hacerle hashing a la cadena.

- Una manera es simplemente verla como un número gigante y luego sacarle módulo p para algún primo grande p.
- Pero una mejor es la que veremos a continuación.
- Mejor en el sentido que tiende a producir menos colisiones, es más fácil de calcular, y más rápida.

Hashing en strings

- Supongamos que

$$s = \text{``}a_0 a_1 a_2 ... a_n\text{''}$$

Hashing en strings

- Supongamos que

$$s = \text{“}a_0 a_1 a_2 ... a_n\text{”}$$

- Primero, convertimos los a_i's a sus números ascii.

Hashing en strings

- Supongamos que
$$s = \text{``}a_0a_1a_2 ... a_n\text{''}$$
- Primero, convertimos los a_i's a sus números ascii.
- Después, consideramos un número x cualquiera (usualmente, $x = 31$), y un número p primo grandotote.

Hashing en strings

- Supongamos que

$$s = \text{“}a_0 a_1 a_2 ... a_n\text{”}$$

- Primero, convertimos los a_i's a sus números ascii.
- Después, consideramos un número x cualquiera (usualmente, $x = 31$), y un número p primo grandotote.
- Definimos el hash de s como sigue:

$$h(s) := a_0 + a_1 x + a_2 x^2 + ... + a_n x^n (\text{mod } p)$$

Hashing en strings

- Supongamos que

$$s = \text{``}a_0 a_1 a_2 ... a_n\text{''}$$

- Primero, convertimos los a_i's a sus números ascii.
- Después, consideramos un número x cualquiera (usualmente, $x = 31$), y un número p primo grandotote.
- Definimos el hash de s como sigue:

$$h(s) := a_0 + a_1 x + a_2 x^2 + ... + a_n x^n (\text{mod } p)$$

- Factorizamos así:

$$h(s) := a_0 + x(a_1 + x(a_2 + x(... + (a_{n-1} + x(a_n))...)(\text{mod } p)$$

Ejercicio

Crea una función que me regrese el hash de una cadena de texto, si le paso x, p, *y la cadena. Recuerda usar* **long** *y reducir módulo* p *en cada momento.*

Rabin-Karp

Problema

Dado una cadena de texto grande T y otra cadena P, queremos saber si $P \subset T$ (y en donde)

Rabin-Karp

Problema

Dado una cadena de texto grande T y otra cadena P, queremos saber si $P \subset T$ (y en donde)

Por ejemplo, si $T =$abaaba, y $P =$aab, entonces sí aparece P en T :abaaba.

Rabin-Karp

Problema

Dado una cadena de texto grande T y otra cadena P, queremos saber si $P \subset T$ (y en donde)

Por ejemplo, si $T =$abaaba, y $P =$aab, entonces sí aparece P en T :abaaba.

- La idea del algoritmo de Rabin-Karp es hacer hashing a cada pedazo de T de tamaño $|P|$.

Rabin-Karp

Problema

Dado una cadena de texto grande T y otra cadena P, queremos saber si $P \subset T$ (y en donde)

Por ejemplo, si $T =$abaaba, y $P =$aab, entonces sí aparece P en T :abaaba.

- La idea del algoritmo de Rabin-Karp es hacer hashing a cada pedazo de T de tamaño $|P|$.
- Es decir, a P, y a cada pedazo $t \subset T$ de tamaño $|P|$, le vamos a asociar un número entero.

Rabin-Karp

Problema

Dado una cadena de texto grande T y otra cadena P, queremos saber si $P \subset T$ (y en donde)

Por ejemplo, si $T =$abaaba, y $P =$aab, entonces sí aparece P en T :abaaba.

- La idea del algoritmo de Rabin-Karp es hacer hashing a cada pedazo de T de tamaño $|P|$.
- Es decir, a P, y a cada pedazo $t \subset T$ de tamaño $|P|$, le vamos a asociar un número entero.
- En caso de que a algún t les asociemos el mismo número entero que a P, ya la hicimos.

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.
- Después, conviertes cada caracter de la cadena a un número (usualmente su representación en ascii): a_0,a_1,...

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.
- Después, conviertes cada caracter de la cadena a un número (usualmente su representación en ascii): a_0,a_1,...
- Tomas el polinomio $a_0 + a_1x + a_2x^2 +$ ... (mod p)

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.
- Después, conviertes cada caracter de la cadena a un número (usualmente su representación en ascii): a_0,a_1,...
- Tomas el polinomio $a_0 + a_1 x + a_2 x^2 + \ldots \pmod{p}$
- Por ejemplo, si la cadena era "hola", entonces tomamos el polinomio

$$h + ox + lx^2 + ax^3 \pmod{p}$$

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.
- Después, conviertes cada caracter de la cadena a un número (usualmente su representación en ascii): a_0,a_1,...
- Tomas el polinomio $a_0 + a_1 x + a_2 x^2 + \ldots \pmod{p}$
- Por ejemplo, si la cadena era "hola", entonces tomamos el polinomio

$$h + ox + lx^2 + ax^3 \pmod{p}$$

donde $a = 97$, $b = 98$, etc. (o A=0,G=1,T=2,C=3)

Hashing para strings

Esta es la manera tradicional de hacer hashing a una cadena A:

- Escoges un número entero cualquiera x (usualmente $x = 31$) y un número primo grande p.
- Después, conviertes cada caracter de la cadena a un número (usualmente su representación en ascii): a_0,a_1,...
- Tomas el polinomio $a_0 + a_1x + a_2x^2 + ...$ (mod p)
- Por ejemplo, si la cadena era "hola", entonces tomamos el polinomio

  $$h + ox + lx^2 + ax^3 \pmod p$$

  donde $a = 97$, $b = 98$, etc. (o A=0,G=1,T=2,C=3)
- Dadas dos cadenas distintas, es altamente improbable que su hashing resulte en el mismo número.

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.
- Después, ¿cómo cambia el polinomio?

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.
- Después, ¿cómo cambia el polinomio?

$$a_0 + a_1 x + a_2 x^2 + \ldots a_m x^m \to a_1 + a_2 x + \ldots a_m x^{m-1} + a_{m+1} x^m$$

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.
- Después, ¿cómo cambia el polinomio?

$$a_0 + a_1 x + a_2 x^2 + ... a_m x^m \to a_1 + a_2 x + ... a_m x^{m-1} + a_{m+1} x^m$$

$$h(x) \ \to \ (h(x) - a_0)/x + a_{m+1} x^m$$

- Así que hay que quitar a_0, dividir entre x y sumar $a_{m+1} x^m$.

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.
- Después, ¿cómo cambia el polinomio?

$$a_0 + a_1 x + a_2 x^2 + ... a_m x^m \to a_1 + a_2 x + ... a_m x^{m-1} + a_{m+1} x^m$$

$$h(x) \ \to \ (h(x) - a_0)/x + a_{m+1} x^m$$

- Así que hay que quitar a_0, dividir entre x y sumar $a_{m+1} x^m$.
- Es decir, podemos sacar el hash de todos los sub-pedazos en tiempo $O(n)$, a diferencia de tiempo $O(nm)$, si lo hacemos por fuerza bruta.

Rabin-Karp

- Entonces, dado el texto T de tamaño n, queremos obtener el hash de TODOS los patrones de cierto tamaño m.
- Eso se puede hacer entendiendo cómo se hace hash:
- Primero, sacaremos a mano el hash del primer bloque de tamaño m.
- Después, ¿cómo cambia el polinomio?

$$a_0 + a_1 x + a_2 x^2 + ... a_m x^m \to a_1 + a_2 x + ... a_m x^{m-1} + a_{m+1} x^m$$

$$h(x) \; \to \; (h(x) - a_0)/x + a_{m+1} x^m$$

- Así que hay que quitar a_0, dividir entre x y sumar $a_{m+1} x^m$.
- Es decir, podemos sacar el hash de todos los sub-pedazos en tiempo $O(n)$, a diferencia de tiempo $O(nm)$, si lo hacemos por fuerza bruta.
- La ventaja que tiene esto es que una vez que "preprocesas" T para cierto m, ya es muy rápido saber si un substring de tamaño m está.

Al revés

- Hay una optimización que hace que de hecho sea más fácil de programar (para no sacar inverso modular!):

Al revés

- Hay una optimización que hace que de hecho sea más fácil de programar (para no sacar inverso modular!):
- Toma:

$$h(x) = a_0 x^m + a_1 x^{m-1} + ... + a_m$$

Al revés

- Hay una optimización que hace que de hecho sea más fácil de programar (para no sacar inverso modular!):
- Toma:

$$h(x) = a_0x^m + a_1x^{m-1} + ... + a_m$$

- Así, ¿cómo cambia el polinomio al avanzar?

Al revés

- Hay una optimización que hace que de hecho sea más fácil de programar (para no sacar inverso modular!):
- Toma:

$$h(x) = a_0 x^m + a_1 x^{m-1} + ... + a_m$$

- Así, ¿cómo cambia el polinomio al avanzar?

$$h(x) \;\rightarrow\; (h(x) - a_0 x^m)x + a_{m+1}$$