

# Hashing: Substring Search

Michael Levin

Department of Computer Science and Engineering  
University of California, San Diego

**Data Structures Fundamentals**  
**Algorithms and Data Structures**

# Outline

- 1 Find Substring in Text
- 2 Rabin-Karp's Algorithm
- 3 Recurrence Equation for Substring Hashes
- 4 Improving Running Time

## Searching for Substring

Given a text  $T$  (website, book, Amazon product page) and a string  $P$  (word, phrase, sentence), find all occurrences of  $P$  in  $T$ .

# Searching for Substring

Given a text  $T$  (website, book, Amazon product page) and a string  $P$  (word, phrase, sentence), find all occurrences of  $P$  in  $T$ .

## Examples

- Specific term in Wikipedia article

## Searching for Substring

Given a text  $T$  (website, book, Amazon product page) and a string  $P$  (word, phrase, sentence), find all occurrences of  $P$  in  $T$ .

## Examples

- Specific term in Wikipedia article
- Gene in a genome

# Searching for Substring

Given a text  $T$  (website, book, Amazon product page) and a string  $P$  (word, phrase, sentence), find all occurrences of  $P$  in  $T$ .

## Examples

- Specific term in Wikipedia article
- Gene in a genome
- Detect files infected by virus — code patterns

# Substring Notation

## Definition

Denote by  $S[i..j]$  the substring of string  $S$  starting in position  $i$  and ending in position  $j$ .

## Examples

If  $S = \text{"hashing"}$ , then

$S[0..3] = \text{"hash"}$ ,

$S[4..6] = \text{"ing"}$ ,

$S[2..5] = \text{"shin"}$ .

## Find Substring in String

**Input:** Strings  $T$  and  $P$ .

**Output:** All such positions  $i$  in  $T$ ,  
 $0 \leq i \leq |T| - |P|$  that  
 $T[i..i + |P| - 1] = P$ .

# Naive Algorithm

For each position  $i$  from 0 to  $|T| - |P|$ , check whether  $T[i..i + |P| - 1] = P$  or not.

If yes, append  $i$  to the result.

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## AreEqual( $S_1, S_2$ )

```
if  $|S_1| \neq |S_2|$ :  
    return False  
for  $i$  from 0 to  $|S_1| - 1$ :  
    if  $S_1[i] \neq S_2[i]$ :  
        return False  
return True
```

## FindSubstringNaive( $T, P$ )

```
positions  $\leftarrow$  empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## FindSubstringNaive( $T, P$ )

```
positions ← empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## FindSubstringNaive( $T, P$ )

```
positions ← empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## FindSubstringNaive( $T, P$ )

```
positions ← empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## FindSubstringNaive( $T, P$ )

```
positions ← empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## FindSubstringNaive( $T, P$ )

```
positions ← empty list
for  $i$  from 0 to  $|T| - |P|$ :
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

# Running Time

## Lemma

Running time of `FindPatternNaive( $T, P$ )` is  $O(|T||P|)$ .

# Running Time

## Lemma

Running time of `FindPatternNaive( $T, P$ )` is  $O(|T||P|)$ .

## Proof

- Each `AreEqual` call is  $O(|P|)$

# Running Time

## Lemma

Running time of `FindPatternNaive( $T, P$ )` is  $O(|T||P|)$ .

## Proof

- Each `AreEqual` call is  $O(|P|)$
- $|T| - |P| + 1$  calls of `AreEqual` total to  $O((|T| - |P| + 1)|P|) = O(|T||P|)$   $\square$

## Bad Example

$T = \text{"aaa}\dots\text{aa"}$  (very long)

$P = \text{"aaa}\dots\text{ab"}$  (much shorter than  $T$ )

## Bad Example

$T = \text{"aaa}\dots\text{aa"}$  (very long)

$P = \text{"aaa}\dots\text{ab"}$  (much shorter than  $T$ )

For each position  $i$  in  $T$  from 0 to  $|T| - |P|$ , the call to `AreEqual` has to make all  $|P|$  comparisons, because the difference is always in the last character.

## Bad Example

$T = \text{"aaa}\dots\text{aa"}$  (very long)

$P = \text{"aaa}\dots\text{ab"}$  (much shorter than  $T$ )

For each position  $i$  in  $T$  from 0 to  $|T| - |P|$ , the call to `AreEqual` has to make all  $|P|$  comparisons, because the difference is always in the last character.

Thus, in this case the naive algorithm runs in time  $\Theta(|T||P|)$ .

# Outline

- 1 Find Substring in Text
- 2 Rabin-Karp's Algorithm**
- 3 Recurrence Equation for Substring Hashes
- 4 Improving Running Time

# Rabin-Karp's Algorithm

- Compare  $P$  with all substrings  $S$  of  $T$  of length  $|P|$

# Rabin-Karp's Algorithm

- Compare  $P$  with all substrings  $S$  of  $T$  of length  $|P|$
- Idea: use hashing to make the comparisons faster

# Comparing Hashes

- If  $h(P) \neq h(S)$ , then definitely  $P \neq S$

# Comparing Hashes

- If  $h(P) \neq h(S)$ , then definitely  $P \neq S$
- If  $h(P) = h(S)$ , call `AreEqual(P, S)` to check whether  $P = S$  or not

# Comparing Hashes

- If  $h(P) \neq h(S)$ , then definitely  $P \neq S$
- If  $h(P) = h(S)$ , call `AreEqual(P, S)` to check whether  $P = S$  or not
- Use polynomial hash family  $\mathcal{P}_p$  with prime  $p$

# Comparing Hashes

- If  $h(P) \neq h(S)$ , then definitely  $P \neq S$
- If  $h(P) = h(S)$ , call `AreEqual(P, S)` to check whether  $P = S$  or not
- Use polynomial hash family  $\mathcal{P}_p$  with prime  $p$
- If  $P \neq S$ , the probability  $\Pr[h(P) = h(S)]$  of collision is at most  $\frac{|P|}{p}$  for polynomial hashing — can be made small by choosing very large prime  $p$

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )
positions  $\leftarrow$  empty list
pHash  $\leftarrow$  PolyHash( $P, p, x$ )
for  $i$  from 0 to  $|T| - |P|$ :
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )
    if pHash  $\neq$  tHash:
        continue
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    tHash  $\leftarrow$  PolyHash( $T[i..i + |P| - 1], p, x$ )  
    if pHash  $\neq$  tHash:  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

# False Alarms

“False alarm” is the event when  $P$  is compared with a substring  $S$  of  $T$ , but  $P \neq S$ .

# False Alarms

“False alarm” is the event when  $P$  is compared with a substring  $S$  of  $T$ , but  $P \neq S$ .

The probability of “false alarm” is at most  $\frac{|P|}{p}$

# False Alarms

“False alarm” is the event when  $P$  is compared with a substring  $S$  of  $T$ , but  $P \neq S$ .

The probability of “false alarm” is at most  $\frac{|P|}{p}$

On average, the total number of “false alarms” will be  $\frac{(|T|-|P|+1)|P|}{p}$ , which can be made small by selecting  $p \gg |T||P|$ .

# Running Time without AreEqual

- $h(P)$  is computed in  $O(|P|)$

# Running Time without AreEqual

- $h(P)$  is computed in  $O(|P|)$
- $h(T[i..i + |P| - 1])$  is computed in  $O(|P|)$ ,  $|T| - |P| + 1$  times

# Running Time without AreEqual

- $h(P)$  is computed in  $O(|P|)$
- $h(T[i..i + |P| - 1])$  is computed in  $O(|P|)$ ,  $|T| - |P| + 1$  times
- $O(|P|) + O((|T| - |P| + 1)|P|) = O(|T||P|)$

# AreEqual Running Time

- AreEqual is computed in  $O(|P|)$

# AreEqual Running Time

- AreEqual is computed in  $O(|P|)$
- AreEqual is called only when  $h(P) = h(T[i..i + |P| - 1])$ , meaning that either an occurrence of  $P$  is found or a “false alarm” happened

# AreEqual Running Time

- AreEqual is computed in  $O(|P|)$
- AreEqual is called only when  $h(P) = h(T[i..i + |P| - 1])$ , meaning that either an occurrence of  $P$  is found or a “false alarm” happened
- By selecting  $p \gg |T||P|$  we make the number of “false alarms” negligible

# Total Running Time

- If  $P$  is found  $q$  times in  $T$ , then total time spent in `AreEqual` is on average  $O\left(\left(q + \frac{(|T|-|P|+1)|P|}{p}\right)|P|\right) = O(q|P|)$  for  $p \gg |T||P|$

# Total Running Time

- If  $P$  is found  $q$  times in  $T$ , then total time spent in `AreEqual` is on average  $O\left(\left(q + \frac{(|T|-|P|+1)|P|}{p}\right)|P|\right) = O(q|P|)$  for  $p \gg |T||P|$
- Total running time is on average  $O(|T||P|) + O(q|P|) = O(|T||P|)$  as  $q \leq |T|$

# Analysis

- $O(|T||P|)$  is the same as running time of the Naive algorithm, but it can be improved!

# Analysis

- $O(|T||P|)$  is the same as running time of the Naive algorithm, but it can be improved!
- The second summand  $O(q|P|)$  is unavoidable as we need to check each of the  $q$  occurrences of  $|P|$  in  $|T|$

# Analysis

- $O(|T||P|)$  is the same as running time of the Naive algorithm, but it can be improved!
- The second summand  $O(q|P|)$  is unavoidable as we need to check each of the  $q$  occurrences of  $|P|$  in  $|T|$
- The first summand  $O(|T||P|)$  is so big because we compute hash of each substring of  $|T|$  separately

# Analysis

- $O(|T||P|)$  is the same as running time of the Naive algorithm, but it can be improved!
- The second summand  $O(q|P|)$  is unavoidable as we need to check each of the  $q$  occurrences of  $|P|$  in  $|T|$
- The first summand  $O(|T||P|)$  is so big because we compute hash of each substring of  $|T|$  separately
- This can be optimized — see next video

# Outline

- 1 Find Substring in Text
- 2 Rabin-Karp's Algorithm
- 3 Recurrence Equation for Substring Hashes**
- 4 Improving Running Time

# Idea

Polynomial hash:

$$h(S) = \sum_{i=0}^{|S|-1} S[i]x^i \text{ mod } p$$

# Idea

Polynomial hash:

$$h(S) = \sum_{i=0}^{|S|-1} S[i]x^i \text{ mod } p$$

Idea: polynomial hashes of two consecutive substrings of  $T$  are very similar

# Idea

Polynomial hash:

$$h(S) = \sum_{i=0}^{|S|-1} S[i]x^i \text{ mod } p$$

Idea: polynomial hashes of two consecutive substrings of  $T$  are very similar

For each  $i$ , denote  $h(T[i..i + |P| - 1])$  by  $H[i]$

# Consecutive substrings

$$T = \text{b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

## Consecutive substrings

$$T = \text{b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$
$$h(\text{"ach"}) =$$

## Consecutive substrings

$$\begin{aligned} T &= \text{b e a c h} \\ \text{encode}(T) &= \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3 \\ h(\text{"ach"}) &= 1 \quad x \quad x^2 \end{aligned}$$

## Consecutive substrings

$$\begin{aligned} T &= \text{b e a c h} \\ \text{encode}(T) &= \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3 \\ h(\text{"ach"}) &= 0 \quad 2x \quad 7x^2 \end{aligned}$$

## Consecutive substrings

$$\begin{aligned} T &= \text{b e a c h} \\ \text{encode}(T) &= \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3 \\ h(\text{"ach"}) &= 0 + 2x + 7x^2 \end{aligned}$$

## Consecutive substrings

$$T = \text{ b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$h(\text{"eac"}) =$$

## Consecutive substrings

$$T = \text{ b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$h(\text{"eac"}) = 1 \quad x \quad x^2$$

## Consecutive substrings

$$T = \text{b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$h(\text{"eac"}) = 4 + 0x + 2x^2$$

## Consecutive substrings

$$T = \text{b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

# Consecutive substrings

$$T = \text{ b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

↓ · x   ↓ · x

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

## Consecutive substrings

$T =$     b    e    a    c    h

encode( $T$ ) = 

1	4	0	2	7
---	---	---	---	---

 $|P| = 3$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

↓ · x    ↓ · x

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

$$H[2] = h(\text{"ach"}) = 0 + 2x + 7x^2$$

## Consecutive substrings

$$T = \text{b e a c h}$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

↓ · x   ↓ · x

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

$$H[2] = h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$H[1] = h(\text{"eac"}) = 4 + 0x + 2x^2 =$$

## Consecutive substrings

$T =$     b    e    a    c    h

encode( $T$ ) = 

1	4	0	2	7
---	---	---	---	---

 $|P| = 3$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$\downarrow \cdot x \quad \downarrow \cdot x$

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

$$H[2] = h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$\begin{aligned} H[1] &= h(\text{"eac"}) = 4 + 0x + 2x^2 = \\ &= 4 + x(0 + 2x) = \end{aligned}$$

## Consecutive substrings

$T =$     b    e    a    c    h

encode( $T$ ) = 

1	4	0	2	7
---	---	---	---	---

 $|P| = 3$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$\downarrow \cdot x \quad \downarrow \cdot x$

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

$$H[2] = h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$H[1] = h(\text{"eac"}) = 4 + 0x + 2x^2 =$$

$$= 4 + x(0 + 2x) =$$

$$= 4 + x(0 + 2x + 7x^2) - 7x^3 =$$

## Consecutive substrings

$$T = \quad b \quad e \quad a \quad c \quad h$$
$$\text{encode}(T) = \begin{array}{|c|c|c|c|c|} \hline 1 & 4 & 0 & 2 & 7 \\ \hline \end{array} \quad |P| = 3$$

$$h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$\downarrow \cdot x \quad \downarrow \cdot x$$

$$h(\text{"eac"}) = 4 + 0 + 2x^2$$

$$H[2] = h(\text{"ach"}) = 0 + 2x + 7x^2$$

$$H[1] = h(\text{"eac"}) = 4 + 0x + 2x^2 =$$

$$= 4 + x(0 + 2x) =$$

$$= 4 + x(0 + 2x + 7x^2) - 7x^3 =$$

$$= xH[2] + 4 - 7x^3$$

# Recurrence Equation for $H[i]$

$$H[i + 1] = \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \text{ mod } p$$

# Recurrence Equation for $H[i]$

$$H[i+1] = \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \bmod p$$

$$H[i] = \sum_{j=i}^{i+|P|-1} T[j]x^{j-i} \bmod p =$$

# Recurrence Equation for $H[i]$

$$H[i+1] = \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \text{ mod } p$$

$$H[i] = \sum_{j=i}^{i+|P|-1} T[j]x^{j-i} \text{ mod } p =$$

$$= \sum_{j=i+1}^{i+|P|} T[j]x^{j-i} + T[i] - T[i+|P|]x^{|P|} \text{ mod } p =$$

# Recurrence Equation for $H[i]$

$$H[i+1] = \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \text{ mod } p$$

$$H[i] = \sum_{j=i}^{i+|P|-1} T[j]x^{j-i} \text{ mod } p =$$

$$= \sum_{j=i+1}^{i+|P|} T[j]x^{j-i} + T[i] - T[i+|P|]x^{|P|} \text{ mod } p =$$

$$= x \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} + (T[i] - T[i+|P|]x^{|P|}) \text{ mod } p$$

# Recurrence Equation for $H[i]$

$$H[i+1] = \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} \text{ mod } p$$

$$H[i] = \sum_{j=i}^{i+|P|-1} T[j]x^{j-i} \text{ mod } p =$$

$$= \sum_{j=i+1}^{i+|P|} T[j]x^{j-i} + T[i] - T[i+|P|]x^{|P|} \text{ mod } p =$$

$$= x \sum_{j=i+1}^{i+|P|} T[j]x^{j-i-1} + (T[i] - T[i+|P|]x^{|P|}) \text{ mod } p$$

$$H[i] = xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \text{ mod } p$$

# Using Recurrence Equation

$$H[i] = xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \bmod p$$

# Using Recurrence Equation

$$H[i] = xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \bmod p$$

- $x^{|P|}$  can be computed once and saved

# Using Recurrence Equation

$$H[i] = xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \bmod p$$

- $x^{|P|}$  can be computed once and saved
- Using this recurrence equation,  $H[i]$  can be computed in  $O(1)$  given  $H[i+1]$  and  $x^{|P|}$

# Using Recurrence Equation

$$H[i] = xH[i+1] + (T[i] - T[i+|P|]x^{|P|}) \bmod p$$

- $x^{|P|}$  can be computed once and saved
- Using this recurrence equation,  $H[i]$  can be computed in  $O(1)$  given  $H[i+1]$  and  $x^{|P|}$
- See next video to learn how this improves the running time of Rabin-Karp

# Outline

- 1 Find Substring in Text
- 2 Rabin-Karp's Algorithm
- 3 Recurrence Equation for Substring Hashes
- 4 Improving Running Time

# Use Precomputation

- Use the recurrence equation to precompute all hashes of substrings of  $|T|$  of length equal to  $|P|$
- Then proceed same way as the original Rabin-Karp algorithm implementation

## PrecomputeHashes( $T, |P|, p, x$ )

$H \leftarrow$  array of length  $|T| - |P| + 1$

$S \leftarrow T[|T| - |P| .. |T| - 1]$

$H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$

$y \leftarrow 1$

for  $i$  from 1 to  $|P|$ :

$y \leftarrow (y \cdot x) \bmod p$

for  $i$  from  $|T| - |P| - 1$  down to 0:

$H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$

return  $H$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

## PrecomputeHashes( $T, |P|, p, x$ )

$H \leftarrow$  array of length  $|T| - |P| + 1$

$S \leftarrow T[|T| - |P| .. |T| - 1]$

$H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$

$y \leftarrow 1$

for  $i$  from 1 to  $|P|$ :

$y \leftarrow (y \cdot x) \bmod p$

for  $i$  from  $|T| - |P| - 1$  down to 0:

$H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$

return  $H$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P| + |P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P| + |P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$O(|P| + |P| + |T| - |P|)$

## PrecomputeHashes( $T, |P|, p, x$ )

```
 $H \leftarrow$  array of length  $|T| - |P| + 1$   
 $S \leftarrow T[|T| - |P|..|T| - 1]$   
 $H[|T| - |P|] \leftarrow \text{PolyHash}(S, p, x)$   
 $y \leftarrow 1$   
for  $i$  from 1 to  $|P|$ :  
     $y \leftarrow (y \cdot x) \bmod p$   
for  $i$  from  $|T| - |P| - 1$  down to 0:  
     $H[i] \leftarrow (xH[i + 1] + T[i] - yT[i + |P|]) \bmod p$   
return  $H$ 
```

$$O(|P| + |P| + |T| - |P|) = O(|T| + |P|)$$

# Precomputing $H$

- PolyHash is called once —  $O(|P|)$
- $x^{|P|}$  is computed in  $O(|P|)$
- All values of  $H$  are computed in  $O(|T| - |P|)$
- Total precomputation time  $O(|T| + |P|)$

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )
positions  $\leftarrow$  empty list
pHash  $\leftarrow$  PolyHash( $P, p, x$ )
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )
for  $i$  from 0 to  $|T| - |P|$ :
    if pHash  $\neq H[i]$ :
        continue
    if AreEqual( $T[i..i + |P| - 1], P$ ):
        positions.Append( $i$ )
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

## RabinKarp( $T, P$ )

```
 $p \leftarrow$  big prime,  $x \leftarrow$  random( $1, p - 1$ )  
positions  $\leftarrow$  empty list  
pHash  $\leftarrow$  PolyHash( $P, p, x$ )  
 $H \leftarrow$  PrecomputeHashes( $T, |P|, p, x$ )  
for  $i$  from 0 to  $|T| - |P|$ :  
    if pHash  $\neq$   $H[i]$ :  
        continue  
    if AreEqual( $T[i..i + |P| - 1], P$ ):  
        positions.Append( $i$ )  
return positions
```

# Improved Running Time

- $h(P)$  is computed in  $O(|P|)$

# Improved Running Time

- $h(P)$  is computed in  $O(|P|)$
- PrecomputeHashes in  $O(|T| + |P|)$

# Improved Running Time

- $h(P)$  is computed in  $O(|P|)$
- PrecomputeHashes in  $O(|T| + |P|)$
- Total time spent in AreEqual is  $O(q|P|)$  on average (for large enough prime  $p$ ), where  $q$  is the number of occurrences of  $P$  in  $T$

# Improved Running Time

- $h(P)$  is computed in  $O(|P|)$
- PrecomputeHashes in  $O(|T| + |P|)$
- Total time spent in AreEqual is  $O(q|P|)$  on average (for large enough prime  $p$ ), where  $q$  is the number of occurrences of  $P$  in  $T$
- Total running time on average  $O(|T| + (q + 1)|P|)$

# Improved Running Time

- $h(P)$  is computed in  $O(|P|)$
- PrecomputeHashes in  $O(|T| + |P|)$
- Total time spent in AreEqual is  $O(q|P|)$  on average (for large enough prime  $p$ ), where  $q$  is the number of occurrences of  $P$  in  $T$
- Total running time on average  $O(|T| + (q + 1)|P|)$
- Usually  $q$  is small, so this is much less than  $O(|T||P|)$

# Conclusion

- Hash tables are useful for storing Sets and Maps

# Conclusion

- Hash tables are useful for storing Sets and Maps
- Possible to search and modify hash tables in  $O(1)$  on average!

# Conclusion

- Hash tables are useful for storing Sets and Maps
- Possible to search and modify hash tables in  $O(1)$  on average!
- Must use good hash families and randomization

# Conclusion

- Hash tables are useful for storing Sets and Maps
- Possible to search and modify hash tables in  $O(1)$  on average!
- Must use good hash families and randomization
- Hashes are also useful while working with strings and texts

# Conclusion

- Hash tables are useful for storing Sets and Maps
- Possible to search and modify hash tables in  $O(1)$  on average!
- Must use good hash families and randomization
- Hashes are also useful while working with strings and texts
- There are many more applications, including blockchain — see next video!