# DSC 190 DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 1

**String Matching** 

#### **Strings**

An alphabet is a set of possible characters.

$$\Sigma = \{G, A, T, C\}$$

A **string** is a sequence of characters from the alphabet.

"GATTACATACGAT"

# **Example: Bitstrings**

```
Σ = {0, 1}
"0110010110"
```

# **Example: Text (Latin Alphabet)**

```
Σ = {a,...,z,<space>}
"this is a string"
```

## **Comparing Strings**

Suppose s and t are two strings of equal length, m.

► Checking for equality takes worst-case time Θ(m) time.

```
def strings_equal(s, t):
    if len(s) != len(t):
        return False
    for i in range(len(s)):
        if s[i] != t[i]:
        return False
    return True
```

# **String Matching**

(Substring Search)

- Given: a string, s, and a pattern string p
- **Determine**: all locations of p in s
- Example:

```
s = "GATTACATACG" p = "TAC"
```

#### **Naïve Algorithm**

Idea: "slide" pattern p across s, check for equality at each location.

#### **Time Complexity**

```
def naive_string_match(s, p):
    match_locations = []
    for i in range(len(s) - len(p) + 1):
        if s[i:i+len(p)] == p:
            match_locations.append(i)
    return match_locations
```

# **Naïve Algorithm**

- ► Worst case:  $\Theta((|s| |p| + 1) \cdot |p|)$  time<sup>1</sup>
- Can we do better?

<sup>&</sup>lt;sup>1</sup>The + 1 is actually important, since if |p| = |s| this should be  $\Theta(1)$ 

#### Yes!

- There are numerous ways to do better.
- We'll look at one: Rabin-Karp.
- ▶ Under some assumptions, takes  $\Theta(|s| + |p|)$  expected time.
- Not always the fastest, but easy to implement, and generalizes to other problems.

# DSC 190 DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 2

**Rabin-Karp** 

#### Idea

- The naïve algorithm performs  $\Theta(|s|)$  comparisons of strings of length |p|.
- ▶ String comparison is slow: O(|p|) time.
- ▶ Integer comparison is fast:  $\Theta(1)$  time<sup>2</sup>.
- ▶ Idea: **hash** strings into integers, compare them.

<sup>&</sup>lt;sup>2</sup>As long as the integers are "not too big"

#### **Recall: Hash Functions**

- A **hash function** takes in an object and returns a (small) number.
- ► **Important**: Given the same object, returns same number.

It may be possible for two different objects to hash to same number. This is a **collision**.

# **String Hashing**

A string hash function takes a string, returns a number.

Given same string, returns same number.

```
»> string_hash("testing")
32
»> string_hash("something else")
7
»> string_hash("testing")
32
```

#### Idea

▶ Instead of performing O(|p|) string comparison for each i:

```
s[i:i + len(p)] == p
```

Hash, and perform Θ(1) *integer* comparison:

```
string_hash(s[i:i + len(p)]) == string_hash(p)
```

In case of collision, need to perform full string comparison in order to ensure this isn't a false match.

# **Example**

```
s = "ABBABAABBABA"
p = "BAA"
```

Х	<pre>string_hash(x)</pre>
AAA	2
AAB	5
ABA	3
BAA	1
ABB	4
BAB	1
BBA	3
BBB	2

#### **Pseudocode**

```
def string_match_with_hashing(s, p):
    match_locations = []
    for i in range(len(s) - len(p) + 1):
        if string_hash(s[i:i+len(p)]) == string_hash(p):
            # make sure this isn't a spurious match due to collision
        if s[i:i+len(p)] == p:
            match_locations.append(i)
    return match_locations
```

### **Time Complexity**

- ightharpoonup Comparing (small) integers takes  $\Theta(1)$  time.
- ▶ But hashing a string x usually takes Ω(|x|).
- In this case, |x| = |p|, so overall:

$$\Omega((|s| + |p| + 1) \cdot |p|)$$

No better than naïve!

#### **Idea: Rolling Hashes**

- We hash many strings.
- But the strings we are hashing change only a little bit.

Example: s = "ozymandias", p = "mandi".

#### **Rabin-Karp**

- We'll design a special hash function.
- Instead of computing hash "from scratch", it will "update" old hash in Θ(1) time.

```
>>> old_hash = rolling_hash("ozymandias", start=0, stop=5)
>>> new_hash = rolling_hash("ozymandias", start=1, stop=6, update=old_has
```

```
def rabin karp(s, p):
    hashed window = string hash(s, o, len(p))
    hashed_pattern = string_hash(p, o, len(p))
    match locations = []
    if s[o:len(p)] == p:
        match locations.append(0)
    for i in range(1, len(s) - len(p) + 1):
        # update the hash
        hashed window = update string hash(s, i, i + len(p), hashed window)
        if hashed window == hashed pattern:
            # make sure this isn't a false match due to collision
            if s[i:i + len(p)] == p:
                match locations.append(i)
```

return match\_locations

### **Time Complexity**

- $\triangleright$   $\Theta(|p|)$  time to hash pattern.
- $\triangleright$   $\Theta(1)$  to update window hash, done  $\Theta(|s| |p| + 1)$  times.
- ▶ When there is a collision,  $\Theta(|p|)$  time to check.

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{c \cdot |p|}_{\text{check collisions}})$$

#### **Worst Case**

- In worst case, every position results in a collision.
- That is, there are Θ(|s|) collisions:

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{|s| \cdot |p|}_{\text{check collisions}}) \longrightarrow \Theta(|s| \cdot |p|)$$

- Example: s = "aaaaaaaaa", p = "aaa"
- This is just as bad as naïve!

#### **More Realistic Time Complexity**

- Only a few valid matches and a few spurious matches.
- Number of collisions depends on hash function.
- Our hash function will reasonably have  $\Theta(|s|/|p|)$  collisions.

$$\Theta(\underbrace{|p|}_{\text{hash pattern}} + \underbrace{|s| - |p| + 1}_{\text{update windows}} + \underbrace{c \cdot |p|}_{\text{check collisions}}) \rightarrow \Theta(|s|)$$

# DSC 190 DATA STRUCTURES & ALGORITHMS

Lecture 14 | Part 3

**Rolling Hashes** 

#### The Problem

We need to hash:

```
s[0:0 + len(p)]
s[1:1 + len(p)]
s[2:2 + len(p)]
...
```

- $\triangleright$  A standard hash function takes  $\Theta(|p|)$  time per call.
- But these strings overlap.
- Goal: Design hash function that takes Θ(1) time to "update" the hash.

### **Strings as Numbers**

Our hash function should take a string, return a number.

Should be unlikely that two different strings have same hash.

Idea: treat each character as a digit in a base- $|\Sigma|$  expansion.

### **Digression: Decimal Number System**

► In the standard decimal (base-10) number system, each digit ranges from 0-9, represents a power of 10.

Example:

$$1532_{10} = (2 \times 10^{0}) + (3 \times 10^{1}) + (5 \times 10^{2}) + (1 \times 10^{3})$$

#### **Digression: Binary Number System**

- Computers use binary (base-2). Each digit ranges from 0-1, represents a power of 2.
- Example:

$$10110_2 = (0 \times 2^0) + (1 \times 2^1) + (1 \times 2^2) + (0 \times 2^3) + (1 \times 2^4)$$
$$= 22_{10}$$

#### **Digression: Base-256**

► We can use whatever base is convenient. For instance, base-128, in which each digit ranges from 0-127, represents a power of 128.

$$12,97,199_{128} = (101 \times 128^0) + (97 \times 128^1) + (12 \times 128^2)$$
  
=  $209125_{10}$ 

# What does this have to do with strings?

- ► We can interpret a character in alphabet Σ as a digit value in base |Σ|.
- For example, suppose  $\Sigma = \{a, b\}$ .
- ► Interpret a as 0, b as 1.
- ► Interpret string "babba" as binary string 101102.
- ► In decimal: 10110<sub>2</sub> = 22<sub>10</sub>

#### Main Idea

We have mapped the string "babba" to an integer: 22. In fact, this is the *only* string over  $\Sigma$  that maps to 22. Interpreting a string of a and b as a binary number hashes the string!

#### **General Strings**

- What about general strings, like "I am a string."?
- Choose some encoding of characters to numbers.
- Popular (if outdated) encoding: ASCII.
- Maps Latin characters, more, to 0-127. So  $|\Sigma| = 128$ .

#### **ASCII TABLE**

0 0 0 0 0 MULLI MERION   48 30 11000 60 0 96 60 110000 140 1		l Hexadecimal					Hexadecimal					Hexadecimal			Char
2 2 10 2 FINE OFFICE OF															
3								110001	61						
4 4 100 4 PROFITMENSISSON SET 101 5 PROFITMENSISSON SET 101 101 101 101 101 101 101 101 101 10															
5 5 101 5   FROMERIC   S3 35 11010 65   100 65   110010 145   6   1   1   1   1   1   1   1   1   1															
6 6 110 6 PACHOMERICAL SA 36 11010 66 0 1002 66 110010 146 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7															
7 7 111 7 PREMIUM STATE															
8 8 0 1000 10   MACASHACE   55 38 111000 70 8 104 68 1101000 150   10   10   10   10   10															
9 9 1001 11   PORNOCONTA TAN   57 39 111001 71 9 105 69 1101001 151   1															g
100   A   1010   12															h
11   0	9			11	[HORIZONTAL TAB]					9	105	69	1101001	151	1
12 C 1100 14   FORM FIETY   60 3C 111100 74   100 6C 1101100 154   111 10 10 1   111										1					j.
13										;					
14   E			1100		[FORM FEED]			111100	74	<	108		1101100	154	1
15				15	(CARRIAGE RETURN)			111101	75		109		1101101	155	m
16 10 10000 20   DAMAINE CAMPIL   64 40 1000000 100   90   112 70   111000 100   10   111		E	1110	16	[SHIFT OUT]	62	3E	111110	76		110	6E	1101110	156	n
19	15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	0
18		10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	0	112	70	1110000	160	p
18   12   10011   22	17	11	10001	21	(DEVICE CONTROL 1)	65	41	1000001	101	A	113	71	1110001	161	a
20 14 10100 24 [PRINT CONTROL 0] 68 44 1000100 104 0 116 74 111010 104 12 12 15 15 10101 25 0 PRINT ACROMATING 69 45 100010 105 0 116 75 1110101 15 0 V	18	12	10010	22	(DEVICE CONTROL 2)	66	42	1000010	102	В	114	72	1110010	162	
20 14 10100 24 [PRIVE CONTROL 4] 68 44 1000100 104 0 116 74 1110100 164 1 12 15 15 10101 25 PRIVE CONTROL 4] 69 44 1000100 105 0 116 75 1111010 164 1 12 15 15 15 15 15 15 15 15 15 15 15 15 15	19	13	10011	23	IDEVICE CONTROL 31	67	43	1000011	103	C	115	73	1110011	163	
21 15 10010 25 PRESENTE ACRONALDED 0 45 1000101105 0 117 75 1110101165 0 4 1 100101106 0 4 117 75 1110101165 0 4 1 100101106 0 4 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	20	14	10100	24	(DEVICE CONTROL 4)	68	44	1000100	104	D	116	74			t
22 16 1011 26   PRICE PR	21	15	10101	25	INEGATIVE ACKNOWLEDGE!	69	45	1000101	105	E	117	75			ü
23 17 1011 27 [PRO F TRANS SLOCK] 71 47 100011107 0 119 77 111011167 W 23 18 10100 101 [PRO F TRANS SLOCK] 72 48 1001001 101 H 120 77 8 1111001 107 X 24 18 10 11000 12 [PRO F TRANS SLOCK] 73 48 1001001 101 H 120 77 8 1111001 107 X 25 11						70				ř.					
24 18 1100 30															
25 10 1100 31 [ROO O'MENON] 73 49 1001001111 1 121 79 111100177 2 26 11 11100 32 [ROO O'MENON] 74 44 1001010112 1 122 79 111100177 2 27 10 11100 34 [ROO O'MENON] 75 4C 1001010112 1 122 70 111100177 2 28 11 C 11100 34 [ROO O'MENON] 76 4C 1001100112 1 122 4 7C 1111100174 2 30 11 E 11110 35 [ROO O'MENON] 77 4D 1001010118 1 122 4 7C 1111100174 2 31 12 100001 31 [ROO O'MENON] 78 4E 100110118 M 125 70 1111101175 1 32 2 10 10000 41 [ROO O'MENON] 78 4E 100110118 M 125 70 1111101175 1 33 2 12 100001 41 [ROO O'MENON] 78 4E 100110118 M 125 70 1111101175 1 34 2 2 10000 41 [ROO O'MENON] 78 4E 100110118 M 125 70 1111101175 1 35 2 5 10000 41 [ROO O'MENON] 78 4E 100110118 M 125 70 1111101175 1 36 24 10010 41 ROO O'MENON] 88 51 101000112 1 S 36 24 10010 41 ROO O'MENON] 88 51 10100112 U S 37 25 100101 45 8 8 55 10100112 U S 38 27 100111 47 ROO O'MENON] 88 51 10100112 U S 39 27 100111 47 ROO O'MENON] 88 51 10100112 U S 40 28 101000 52 ROO O'MENON] 88 51 10100112 U S 41 28 101000 52 ROO O'MENON] 88 51 101101113 U S 42 28 101000 52 ROO O'MENON] 88 51 101101113 U S 43 28 10100113 ROO O'MENON] 88 51 101101113 U S 44 27 101001 52 ROO O'MENON] 88 51 101101113 U S 44 27 101001 54 ROO O'MENON] 88 51 101101113 U S 44 27 101011 54 ROO O'MENON] 88 51 101101113 U S 44 27 101011 54 ROO O'MENON] 88 51 101101113 U S 44 27 101011 54 ROO O'MENON] 88 51 101101113 U S 44 27 101011 54 ROO O'MENON] 88 51 101101113 U S 44 27 101011 54 ROO O'MENON] 88 51 101101113 U S 45 27 101111 U S 46 28 101110 54 ROO O'MENON] 88 51 101101113 U S 46 27 101101 54 ROO O'MENON] 88 51 101101113 U S 47 101101 54 ROO O'MENON] 88 51 1011010113 U S 48 101101 51 U S 51 11111 U S 51 11111 U S 51 111111 U S 51 11111 U S 51 11111 U S 51 11111 U S 51 11111 U S 51 111111 U S 51 11111 U															
26 1A 11010 32   SUBSTITUTE   74   4A 1001001112   1222 7A 1111010 177   27   27   27   27   27   27   27															
27   10															
28 1C 1100 34   PILE SPRINGFORM   76 4C 1001100 113															
29 10 1110 35 [GROOF SPANATON] 77 40 100110115 M 125 70 111110175 3 30 1E 11110 36 [GROOF SPANATON] 78 4E 1001110116 M 125 70 111110175 3 31 1F 11113 37 [DRY SPANATON] 78 4E 100111110 0 127 7F 111110176 - 31 21 10001 42 * 8 8 51 100010122 0 34 22 100001 42 * 82 52 10000122 0 35 23 10001 43 * 8 8 3 34 10001123 5 37 25 10001 45 * 8 8 3 34 10001123 5 38 26 10010 66 6 8 65 56 100101012 0 39 27 10011 47 * 87 8 57 100111117 W 41 28 10001 47 * 87 8 57 100111117 W 42 2A 10100 52 * 90 5A 10100113															7
10   16															
31															
32															
33 22 10000141 1 81 81 51 100001121 0 34 22 10000142 9 82 23 1010001122 8 36 24 100100 44 8 84 54 1010100124 7 37 25 1000114 7 8 85 54 1010100124 7 38 27 100111 47 8 85 55 1010011125 W 39 27 100111 47 8 87 57 101011112 W 40 28 101000 50 6 88 58 101000 110 X 41 29 101001 51 1 88 59 101100 110 X 42 42 43 20 101001 51 1 88 59 101100 110 X 43 28 101000 51 1 88 59 101100 110 X 44 2 2 5 101001 51 1 88 59 101100 110 X 44 2 6 7 101001 51 1 88 59 101100 110 X 44 4 6 7 101001 51 1 88 59 101100 113 X 44 6 7 101001 51 1 89 59 101100 113 X 46 6 7 101001 51 1 89 59 101100 113 X											447	**	1111111	277	(DEL)
34 22 100010 42 * 82 52 1010010 122 8 3 3 3 3 4 3 1010011 123 5 3 3 3 3 2 3 1 100011 32 8 3 3 3 3 1010011 32 5 3 3 3 3 3 3 4 5 101001 34 8 8 8 3 5 3 1010011 32 5 3 3 3 3 3 4 5 101001 34 8 8 8 5 5 101001 32 6 7 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1					(SPACE)										
35 23 100011 43 # 83 53 1010011123 5 84 54 101100124 T 378 25 100110 45 8 85 55 1010101 125 U 878 25 100110 45 7 8 85 55 1010110 125 U 88 25 100110 45 7 8 85 55 101010 125 U 89 28 101000 50 (							52								
36 24 100100 44 5 8 84 54 101100 124 T  37 25 100101 45 5 8 85 54 101010 124 T  38 27 10011 47 8 8 55 101010 125 U  39 27 10011 47 8 87 57 101010 125 U  44 2 28 10100 152 1 88 58 101100 110 X  44 2 28 10100 152 1 88 58 101100 110 X  44 2 28 10100 152 1 88 58 101100 113 X  46 2															
377 25 100101 45 % 88 55 1010101 125 U  38 26 101010 47 6 6 86 56 1010101 125 V  39 27 100111 47 6 87 57 101011 117 W  41 28 101001 51 1 88 59 1010101 131 W  42 2A 101010 52 9 90 5A 1010101 131 V  43 28 101001 53 9 90 5A 1011010 132 C  44 28 20 10101 55 9 90 5A 1011010 133 (  45 20 10101 55 9 90 5A 1011010 135 (  46 26 10101 55 9 93 50 101101 135 (					1										
38															
39 27 100111 47 67 57 1001111 127 W 40 28 101000 50 1 88 58 1011000 10 X 41 29 101000 50 1 88 58 1011000 10 X 42 20 10100 52 9 90 54 101100 132 Z 43 28 10100 54 9 95 56 1011100 133 ( 44 20 10100 54 9 90 55  1011100 134 ( 45 25 101100 54 9 92 55  1011100 134 ( 46 26 101110 156 9 94 55  1011100 136 (															
40 28 100000 50 ( 88 58 1011000 130 X 11 20 101000 131 ) 89 59 1011000 131 Y 12 2					ex .										
41 29 101001 51 1 89 59 1011001 131 Y  42 2A 101010 52 * 90 5A 1011001 312 Z  43 28 101011 33 * 91 58 101101 133 [  44 2C 101100 54 , 92 5C 1011100 114 \  45 2D 10110 155 - 93 5D 101110 115 ]  46 2E 101110 156 . 94 5E 1011110 136 6															
42 2A 10100 52 * 90 5A 1011010 132 Z 43 2R 101100 133 * 91 15 R 1011010 133 [ 44 2 E 101100 134 , 92 5 101100 134 [ 46 2E 101100 14 , 92 5 1011100 134 [ 47 2					§										
43 28 10101 33 4 91 58 101101 133 1 44 2C 101100 34 92 5C 1011100 134 1 45 2D 10110 35 - 93 5D 101110 135 1 46 2E 101110 136 - 94 5E 1011110 136 6					1										
44 2C 101100 54 , 92 5C 1011100 134 \ 45 2D 10110 55 - 93 5D 101110 135   46 2E 101110 56 . 94 5E 101110 136 ^					*					4					
45 2D 101101 55 - 93 5D 1011101 135 1 46 2E 101110 56 . 94 5E 1011110 136 ^					*					į.					
46 2E 101110 56 . 94 5E 1011110 136 ^					A Company of the Comp					J.					
47 2F 101111 57 / 95 5F 1011111 137					4					^					
	47	2F	101111	57	1	95	5F	1011111	137	-	l)				

# **In Python**

```
>> ord('a')
97
>> ord('Z')
90
>> ord('!')
33
```

#### **ASCII as Base-128**

- ► Each character represents a number in range 0-127.
- ► A string is a number represented in base-128.

#### Example:

Hello <sub>128</sub> = (111 × 128 <sup>0</sup> )	_	character	ASCII code
+ (108 × 128 <sup>1</sup> )	-	Н	72
+ (108 × 128 <sup>2</sup> )		e	101
+ (101 × 128 <sup>3</sup> )		l	108
+ (72 × 128 <sup>4</sup> )		О	111
= 19540948591 <sub>10</sub>			

```
def base_128_hash(s, start, stop):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    p = 0
    total = 0
    while stop > start:
        total += ord(s[stop-1]) * 128**p
        p += 1
```

stop -= 1 return total

# **Rolling Hashes**

- We can hash a string x by interpreting it as a number in a different base number system.
- ▶ But hashing takes time  $\Theta(|x|)$ .
- With rolling hashes, it will take time  $\Theta(1)$  to "update".

character	ASCII code
Н	72
e	101
l	108
О	111

Hash of "Hel" in "Hello" ► Hash of "ell" in "Hello"

# "Updating" a Rolling Hash

- Start with old hash, subtract character to be removed.
- "Shift" by multiplying by 128.
- Add new character.
- Takes Θ(1) time.

```
def update_base_128_hash(s, start, stop, old):
    # assumes ASCII encoding, base 128
    length = stop - start
    removed_char = ord(s[start - 1]) * 128**(length - 1)
    added_char = ord(s[stop - 1])
    return (old - removed_char) * 128 + added_char
```

```
»> base_128_hash("Hello", 0, 3)
1192684
```

1668716

>>> base 128 hash("Hello", 1, 4)

»> base\_128\_hash("Hello", 1, 4
1668716

»> update\_base\_128\_hash("Hello", 1, 4, 1192684)

#### **Note**

- In this hashing strategy, there are no collisions!
- ► Two different string have two different hashes.
- But as we'll see... it isn't practical.

# **Rabin-Karp**

```
def rabin karp(s, p):
    hashed_window = base_128_hash(s, o, len(p), q)
    hashed pattern = base 128 hash(p, o, len(p), g)
   match locations = []
    if s[o:len(p)] == p:
        match_locations.append(0)
   for i in range(1. len(s) - len(p) + 1):
        # update the hash
        hashed window = update base 128 hash(s. i. i + len(p). hashed window)
        # hashes are unique; no collisions
        if hashed window == hashed pattern:
            match locations.append(i)
    return match locations
```

## **Example**

```
s = "this is a test",
p = "is"
```

hashed\_pattern = 13555

i	s[]	hashed_window
0	"th"	14952
1	"hi"	13417
2	"is"	13555
3	"s "	14752
4	" i"	4201
5	"is"	13555
6	"s "	14752
7	" a"	4193
8	"a "	12448
9	" t"	4212
10	"te"	14949
11	"es"	13043
12	"st"	14836

### **Large Numbers**

Hashing because integer comparison takes Θ(1) time.

- Only true if integers are small enough.
- Our integers can get very large.

### **Example**

```
»> p = "University of California"
»> base_128_hash(p, o, len(p))
250986132488946228262668052010265908722774302242017
```

### **Large Integers**

- In some languages, large integers will overflow.
- Python has arbitrary size integers.
- But comparison no longer takes Θ(1)

### **Solution**

▶ Use modular arithmetic.

```
Example: (4 + 7) % 3 = 11 % 3 = 2
```

Results in much smaller numbers.

#### Idea

- ightharpoonup Choose a random prime number > |m|.
- ▶ Do all arithmetic modulo this number.

```
def base_128_hash(s, start, stop, q):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    p = 0
    total = 0
    while stop > start:
        total = (total + ord(s[stop-1]) * 128**p) % q
        p += 1
        stop -= 1
    return total
def update base 128 hash(s, start, stop, old, g):
    # assumes ASCII encoding, base 128
    length = stop - start
```

removed\_char = ord(s[start - 1]) \* 128\*\*(length - 1)

return ((old - removed char) \* 128 + added char) % q

added char = ord(s[stop - 1])

#### **Note**

Now there can be collisions!

Even if window hash matches pattern hash, need to verify that strings are indeed the same.

```
def rabin karp(s. p. q):
    hashed window = base 128 hash(s, o, len(p), g)
    hashed_pattern = base_128_hash(p, o, len(p), q)
   match locations = []
   if s[o:len(p)] == p:
        match locations.append(0)
   for i in range(1, len(s) - len(p) + 1):
        # update the hash
        hashed window = update base 128 hash(s, i, i + len(p), hashed window, q)
        if hashed window == hashed pattern:
            # make sure this isn't a false match due to collision
            if s[i:i + len(p)] == p:
                match locations.append(i)
```

return match\_locations

# **Time Complexity**

- If q is prime and > |p|, the chance of two different strings colliding is small.
- From before: if the number of matches is small, Rabin-Karp will take  $\Theta(|s| + |p|)$  expected time.
- ► Since  $|p| \le |s|$ , this is  $\Theta(s)$ .
- ▶ Worst-case time:  $\Theta(|s| \cdot |p|)$ .