DSC 190

Data Structures & Algorithms

Lecture 14 | Part 1

String Matching

Today’s Problem

- Is a needle in a haystack?
- That is, where does the small string p occur in the large string s?

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

$\Sigma = \{0, 1\}$

"0110010110"

Example: Text (Latin Alphabet)

$\Sigma = \{a, \ldots, z, \text{<space>}\}$

”this is a string”

Comparing Strings

- Suppose `s` and `t` are two strings of equal length, m.
- Checking for equality takes worst-case time $\Theta(m)$ time.

```python
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.

```
s =     G   A   T   T   A   C   A   T   A   C
        |   |   |
p =     T   A   C
```

Naïve Algorithm

- Idea: “slide” pattern p across s, check for equality at each location.

```
s =    G  A  T  T  A  C  A  T  A  C
          |  |  |
p =       T  A  C
```

Naïve Algorithm

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

```
s =     G  A  T  T  A  C  A  T  A  C
              |  |  |
p =           T  A  C
```

Naïve Algorithm

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

```
s =     G  A  T  T  A  C  A  T  A  C
                 |  |  |
p =              T  A  C
```

match!

Naïve Algorithm

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

```
s =     G  A  T  T  A  C  A  T  A  C
                    |  |  |
p =                 T  A  C
```

Naïve Algorithm

- Idea: “slide” pattern p across s, check for equality at each location.

```
s  =     G  A  T  T  A  C  A  T  A  C

p  =                    T  A  C
```

Naïve Algorithm

- Idea: “slide” pattern p across s, check for equality at each location.

```
s  =    G  A  T  T  A  C  A  T  A  C
                          |  |  |
p  =                      T  A  C
```

Naïve Algorithm

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

```
s  =   G  A  T  T  A  C  A  T  A  C
                            |  |  |
p  =                        T  A  C
```

match!

Exercise

Exactly how many sliding windows are checked, as a formula involving $|s|$ and $|p|$?

```
s  =  G  A  T  T  A  C  A  T  A  C
               |  |  |
p  =           T  A  C
```

Naïve Algorithm

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

Time Complexity

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

Naïve Algorithm

- Worst case: $\Theta((|s| - |p| + 1) \times |p|)$ time[1]
- Can we do better?

[1] The $+ 1$ is actually important, since if $|p| = |s|$ this should be $\Theta(p)$

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 (potentially many) comparisons of strings of length $|p|$.
- String comparison is slow: $O(|p|)$ time.
- Integer comparison is fast: $\Theta(1)$ time[2].
- Idea: **hash** strings into integers, compare hashes.

[2] 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

- Slide a “window” across s, like before.
- But don’t compare window to p directly!
- Instead, compare hash of window to hash of p.
  - If **unequal**, then **no match**.
  - If **equal**, then **possible match**, perform full string comparison to verify.

Example

hash = 12

s = G A T T A C A T A C

p = T A C

hash = 11

Example

hash = 8

s = G $\boxed{\text{A T T}}$ A C A T A C

p = $\boxed{\text{T A C}}$

hash = 11

Example

hash = 11

s = G A T T A C A T A C

p = T A C

hash = 11

spurious (fake) match!

Example

hash = 11

s = G A T T A C A T A C

p = T A C

hash = 11

match!

Example

hash = 9

s = G A T T A C A T A C

p = T A C

hash = 11

Example

hash = 14

s = G A T T A $\boxed{\text{C A T}}$ A C

p = $\boxed{\text{T A C}}$

hash = 11

Example

hash = 7

s = G A T T A C $\boxed{\text{A T A}}$ C

p = $\boxed{\text{T A C}}$

hash = 11

Example

hash = 11

s = G A T T A C A $\boxed{\text{T A C}}$

p = $\boxed{\text{T A C}}$

hash = 11

match!

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
```

Exercise

What is the time complexity of this approach, assuming that there is at most one match?

```
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 of Hashing

- Often, we assume hashing takes constant time with respect to the input size.
- But now, we'll be more careful.
- Hashing a string p takes $\Theta(|p|)$ time.

Time Complexity

- Comparing (small) integers takes $\Theta(1)$ time.
- But hashing a string p takes $\Theta(|p|)$.
- In this case, overall:

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

- **No better than naïve!**

Idea: Rolling Hashes

- We have hashed each window from scratch.
- But the strings we are hashing change only a little bit.
  - Example: `s = "ozymandias"`, `p = "mandi"`.
- What if, instead of computing hash from scratch, we could "update" the old hash?

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

Rabin-Karp

- This is the idea behing the **Rabin-Karp** string matching algorithm.
- We'll design a special rolling hash function.
- Instead of computing hash "from scratch", it will "update" old hash in $\Theta(1)$ time.

```python
def rabin_karp(s, p):
    hashed_window = string_hash(s, 0, len(p))
    hashed_pattern = string_hash(p, 0, len(p))
    match_locations = []

    if s[0: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

- $\Theta(|p|)$ time to hash pattern.
- $\Theta(1)$ to update window hash, done $\Theta(|s| - |p| + 1)$ times.
- For each collision, $\Theta(|p|)$ time to check.
- If there are k collisions:

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

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

Exercise

What is the worst case situation for Rabin-Karp?

Worst Case

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

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

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

More Realistic Time Complexity

- Typicall, there are 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.

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

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)]`
  - ...
- A standard hash function takes $\Theta(|p|)$ time per call.
- But these strings overlap. Maybe we can save work?
- Goal: Design hash function that takes $\Theta(1)$ 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:

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

Digression: Binary Number System

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

- Example:

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

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.

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

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 10110_2.
- In decimal: $10110_2 = 22_{10}$

Main Idea

We have mapped the string `”babba”` to an integer: 22. In fact, this is the *only* string over Σ 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

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]
1	1	1	1	[START OF HEADING]
2	2	10	2	[START OF TEXT]
3	3	11	3	[END OF TEXT]
4	4	100	4	[END OF TRANSMISSION]
5	5	101	5	[ENQUIRY]
6	6	110	6	[ACKNOWLEDGE]
7	7	111	7	[BELL]
8	8	1000	10	[BACKSPACE]
9	9	1001	11	[HORIZONTAL TAB]
10	A	1010	12	[LINE FEED]
11	B	1011	13	[VERTICAL TAB]
12	C	1100	14	[FORM FEED]
13	D	1101	15	[CARRIAGE RETURN]
14	E	1110	16	[SHIFT OUT]
15	F	1111	17	[SHIFT IN]
16	10	10000	20	[DATA LINK ESCAPE]
17	11	10001	21	[DEVICE CONTROL 1]
18	12	10010	22	[DEVICE CONTROL 2]
19	13	10011	23	[DEVICE CONTROL 3]
20	14	10100	24	[DEVICE CONTROL 4]
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]
22	16	10110	26	[SYNCHRONOUS IDLE]
23	17	10111	27	[ENG OF TRANS. BLOCK]
24	18	11000	30	[CANCEL]
25	19	11001	31	[END OF MEDIUM]
26	1A	11010	32	[SUBSTITUTE]
27	1B	11011	33	[ESCAPE]
28	1C	11100	34	[FILE SEPARATOR]
29	1D	11101	35	[GROUP SEPARATOR]
30	1E	11110	36	[RECORD SEPARATOR]
31	1F	11111	37	[UNIT SEPARATOR]
32	20	100000	40	[SPACE]
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/

Decimal	Hexadecimal	Binary	Octal	Char
48	30	110000	60	0
49	31	110001	61	1
50	32	110010	62	2
51	33	110011	63	3
52	34	110100	64	4
53	35	110101	65	5
54	36	110110	66	6
55	37	110111	67	7
56	38	111000	70	8
57	39	111001	71	9
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	A
66	42	1000010	102	B
67	43	1000011	103	C
68	44	1000100	104	D
69	45	1000101	105	E
70	46	1000110	106	F
71	47	1000111	107	G
72	48	1001000	110	H
73	49	1001001	111	I
74	4A	1001010	112	J
75	4B	1001011	113	K
76	4C	1001100	114	L
77	4D	1001101	115	M
78	4E	1001110	116	N
79	4F	1001111	117	O
80	50	1010000	120	P
81	51	1010001	121	Q
82	52	1010010	122	R
83	53	1010011	123	S
84	54	1010100	124	T
85	55	1010101	125	U
86	56	1010110	126	V
87	57	1010111	127	W
88	58	1011000	130	X
89	59	1011001	131	Y
90	5A	1011010	132	Z
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_

Decimal	Hexadecimal	Binary	Octal	Char
96	60	1100000	140	`
97	61	1100001	141	a
98	62	1100010	142	b
99	63	1100011	143	c
100	64	1100100	144	d
101	65	1100101	145	e
102	66	1100110	146	f
103	67	1100111	147	g
104	68	1101000	150	h
105	69	1101001	151	i
106	6A	1101010	152	j
107	6B	1101011	153	k
108	6C	1101100	154	l
109	6D	1101101	155	m
110	6E	1101110	156	n
111	6F	1101111	157	o
112	70	1110000	160	p
113	71	1110001	161	q
114	72	1110010	162	r
115	73	1110011	163	s
116	74	1110100	164	t
117	75	1110101	165	u
118	76	1110110	166	v
119	77	1110111	167	w
120	78	1111000	170	x
121	79	1111001	171	y
122	7A	1111010	172	z
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	[DEL]

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:

$$
\begin{aligned}
&\texttt{Hello}_{128} \\
&= (72 \times 128^4) \\
&\quad + (101 \times 128^3) \\
&\quad + (108 \times 128^2) \\
&\quad + (108 \times 128^1) \\
&\quad + (111 \times 128^0) \\
&= 19540948591_{10}
\end{aligned}
$$

character	ASCII code
H	72
e	101
l	108
o	111

```python
def base_128_hash(s, start, stop):
    """Hash s[start:stop] by interpreting as ASCII base 128"""
    # this is only pseudo-code!
    p = 0
    total = 0
    while stop > start:
        total += ord(s[stop-1]) * 128**p # <--  : O
        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”.

Example

character	ASCII code
H	72
e	101
l	108
o	111

- Hash of “Hel” in “**Hel**lo”
- Hash of “ell” in “H**ell**o”

“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
»> base_128_hash("Hello", 1, 4)
1668716
»> update_base_128_hash("Hello", 1, 4, 1192684)
1668716
```

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

```python
def rabin_karp(s, p):
    hashed_window = base_128_hash(s, 0, len(p), q)
    hashed_pattern = base_128_hash(p, 0, len(p), q)
    match_locations = []

    if s[0: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

- We're hashing because integer comparison takes $\Theta(1)$ time.
- But this is only true if integers are small enough.
- Our integers can get **very large**.

$$128^{|p|-1}$$

Example

```
»> p = ”University of California”
»> base_128_hash(p, 0, len(p))
2509861324889462282626680520102659087227743022242017
```

Large Integers

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

Solution

- Use **modular arithmetic**.
- Example:
  `(4 + 7) % 3 = 11 % 3 = 2`
- Results in much smaller numbers.

Idea

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

Fact

$$
\begin{aligned}
&(c_1 \times b^2 + c_2 \times b + c_3) \mod q \\
&\qquad = \Big(\big[(c_1 \mod q)b + c_2\big] \mod q\Big) b + c_3 \mod q
\end{aligned}
$$

- Example: $c_1 = 7, c_2 = 3, c_3 = 5, b = 2, q = 11$
- This allows us to keep numbers small.

```python
import math

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

```python
def update_base_128_hash(s, start, stop, old, q):
    length = stop - start

    # assumes ASCII encoding, base 128
    # remove the old value, effectively subtracting
    # ord(s[start]) * 128**(length-1) from old, but
    # mod q so that we don't overflow
    new = old - ord(s[start-1]) * pow(128, length - 1, q)

    # "shift" up
    new *= 128
    new %= q

    # add the new character
    new += ord(s[stop - 1])
    new %= q

    return new

print(base_128_hash("DSC190", 0, 6, 499))
```

Example

```
»> base_128_hash("testing", start=0, stop=4, q=117)
103
»> base_128_hash("testing", start=1, stop=5, q=117)
84
»> update_base_128_hash("testing", start=1, stop=5, old=103, q=117)
84
```

Note

- Now there can be collisions!
- Even if window hash matches pattern hash, need to verify that strings are indeed the same.

```python
def rabin_karp(s, p, q):
    hashed_window = base_128_hash(s, 0, len(p), q)
    hashed_pattern = base_128_hash(p, 0, len(p), q)
    match_locations = []

    if s[0: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: $\Theta(|s| \cdot |p|)$.