DSC 190

DATA STRUCTURES & ALGORITHMS

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

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

"0110010110"

Example: Text (Latin Alphabet)

$\Sigma = \{a, \ldots, z, \langle space \rangle\}$

”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.

```
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
```

s = "hello"
t = "hella"

String Matching

(Substring Search)

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

3, 7

s = "GATTACATACG"   p = "TAC"

0 1 2 3 4 5 6 7 8 9 10

Naïve Algorithm

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

```
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
```

s = "GATTACATACG"   p = "TAC"

TAC

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
```

$\Theta(|s| - |p| + 1)\, O(|p|)$

$O((|s|-|p|+1)|p|)$

Naïve Algorithm

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

[1] 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

Rabin-Karp

Idea

$42 == 47$

- 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[2].
- Idea: **hash** strings into integers, compare them.

[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

- Instead of performing $O(|p|)$ string comparison for each `i`:

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

- Hash, and perform $\Theta(1)$ *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" → 1

0	4
1	3
2	1 → spurious match
3	3
4	1 → good match
5	5
6	4
7	3
8	1 → spurious
9	3

x	string_hash(x)
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

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

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

- No better than naïve!

Idea: Rolling Hashes

ozyma
zyman
ymand

- 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 $\Theta(1)$ time.

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

```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
```

$\Theta(|p|)$

s = _ _ _ _ _

p = _ _ _

Time Complexity

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

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

Worst Case

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

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

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

P

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 $\overset{O}{\cancel{\Theta}}(|s|/|p|)$ collisions.

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

DSC 190

DATA STRUCTURES & ALGORITHMS

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.
- 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:

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

$$= 2 + 30 + 500 + 1000$$

$$= 1532$$

Digression: Binary Number System

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

- Example:

$$\begin{aligned} 10110_2 &= \overset{0}{(0 \times 2^0)} \overset{+}{+} \overset{2}{(1 \times 2^1)} \overset{+}{+} \overset{4}{(1 \times 2^2)} \overset{+}{+} \overset{0}{(0 \times 2^3)} \overset{+}{+} \overset{16}{(1 \times 2^4)} \\ &= 22_{10} \end{aligned}$$

Digression: Base-~~256~~ 128

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

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

$$= 209125_{10}$$

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

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	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*	48	30	110000	60	**0**	96	60	1100000	140	`
1	1	1	1	*[START OF HEADING]*	49	31	110001	61	**1**	97	61	1100001	141	**a**
2	2	10	2	*[START OF TEXT]*	50	32	110010	62	**2**	98	62	1100010	142	**b**
3	3	11	3	*[END OF TEXT]*	51	33	110011	63	**3**	99	63	1100011	143	**c**
4	4	100	4	*[END OF TRANSMISSION]*	52	34	110100	64	**4**	100	64	1100100	144	**d**
5	5	101	5	*[ENQUIRY]*	53	35	110101	65	**5**	101	65	1100101	145	**e**
6	6	110	6	*[ACKNOWLEDGE]*	54	36	110110	66	**6**	102	66	1100110	146	**f**
7	7	111	7	*[BELL]*	55	37	110111	67	**7**	103	67	1100111	147	**g**
8	8	1000	10	*[BACKSPACE]*	56	38	111000	70	**8**	104	68	1101000	150	**h**
9	9	1001	11	*[HORIZONTAL TAB]*	57	39	111001	71	**9**	105	69	1101001	151	**i**
10	A	1010	12	*[LINE FEED]*	58	3A	111010	72	**:**	106	6A	1101010	152	**j**
11	B	1011	13	*[VERTICAL TAB]*	59	3B	111011	73	**;**	107	6B	1101011	153	**k**
12	C	1100	14	*[FORM FEED]*	60	3C	111100	74	**<**	108	6C	1101100	154	**l**
13	D	1101	15	*[CARRIAGE RETURN]*	61	3D	111101	75	**=**	109	6D	1101101	155	**m**
14	E	1110	16	*[SHIFT OUT]*	62	3E	111110	76	**>**	110	6E	1101110	156	**n**
15	F	1111	17	*[SHIFT IN]*	63	3F	111111	77	**?**	111	6F	1101111	157	**o**
16	10	10000	20	*[DATA LINK ESCAPE]*	64	40	1000000	100	**@**	112	70	1110000	160	**p**
17	11	10001	21	*[DEVICE CONTROL 1]*	65	41	1000001	101	**A**	113	71	1110001	161	**q**
18	12	10010	22	*[DEVICE CONTROL 2]*	66	42	1000010	102	**B**	114	72	1110010	162	**r**
19	13	10011	23	*[DEVICE CONTROL 3]*	67	43	1000011	103	**C**	115	73	1110011	163	**s**
20	14	10100	24	*[DEVICE CONTROL 4]*	68	44	1000100	104	**D**	116	74	1110100	164	**t**
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*	69	45	1000101	105	**E**	117	75	1110101	165	**u**
22	16	10110	26	*[SYNCHRONOUS IDLE]*	70	46	1000110	106	**F**	118	76	1110110	166	**v**
23	17	10111	27	*[ENG OF TRANS. BLOCK]*	71	47	1000111	107	**G**	119	77	1110111	167	**w**
24	18	11000	30	*[CANCEL]*	72	48	1001000	110	**H**	120	78	1111000	170	**x**
25	19	11001	31	*[END OF MEDIUM]*	73	49	1001001	111	**I**	121	79	1111001	171	**y**
26	1A	11010	32	*[SUBSTITUTE]*	74	4A	1001010	112	**J**	122	7A	1111010	172	**z**
27	1B	11011	33	*[ESCAPE]*	75	4B	1001011	113	**K**	123	7B	1111011	173	**{**
28	1C	11100	34	*[FILE SEPARATOR]*	76	4C	1001100	114	**L**	124	7C	1111100	174	**|**
29	1D	11101	35	*[GROUP SEPARATOR]*	77	4D	1001101	115	**M**	125	7D	1111101	175	**}**
30	1E	11110	36	*[RECORD SEPARATOR]*	78	4E	1001110	116	**N**	126	7E	1111110	176	**~**
31	1F	11111	37	*[UNIT SEPARATOR]*	79	4F	1001111	117	**O**	127	7F	1111111	177	*[DEL]*
32	20	100000	40	*[SPACE]*	80	50	1010000	120	**P**					
33	21	100001	41	**!**	81	51	1010001	121	**Q**					
34	22	100010	42	**"**	82	52	1010010	122	**R**					
35	23	100011	43	**#**	83	53	1010011	123	**S**					
36	24	100100	44	**$**	84	54	1010100	124	**T**					
37	25	100101	45	**%**	85	55	1010101	125	**U**					
38	26	100110	46	**&**	86	56	1010110	126	**V**					
39	27	100111	47	**'**	87	57	1010111	127	**W**					
40	28	101000	50	**(**	88	58	1011000	130	**X**					
41	29	101001	51	**)**	89	59	1011001	131	**Y**					
42	2A	101010	52	*****	90	5A	1011010	132	**Z**					
43	2B	101011	53	**+**	91	5B	1011011	133	**[**					
44	2C	101100	54	**,**	92	5C	1011100	134	****					
45	2D	101101	55	**-**	93	5D	1011101	135	**]**					
46	2E	101110	56	**.**	94	5E	1011110	136	**^**					
47	2F	101111	57	**/**	95	5F	1011111	137	**_**					

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:

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

character	ASCII code
H	72
e	101
l	108
o	111

$s =$ Hello

start stop

```
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
```

$p=1$

$$111 \times 128^0 + 108 \times 128^1 + 108 \times 128^2 + \vdots$$

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".

$$\text{new} = \left(\text{old} - \text{ord}('H') \times 128^2\right) 128 + \text{ord}('l')$$

Example

character	ASCII code
H	72
e	101
l	108
o	111

- Hash of “Hel” in “**Hel**lo”

$$\begin{array}{ll} l & 108 \times 128^0 \\ & + \\ e & 101 \times 128^1 \\ & + \\ H & 72 \times 128^2 \\ \hline & \text{old} \end{array}$$

- Hash of “ell” in “H**ell**o”

$$\begin{array}{ll} 108 \times 128^0 & l \\ + & \\ 108 \times 128^1 & l \\ + & \\ 101 \times 128^2 & e \end{array}$$

“Updating” a Rolling Hash

- Start with old hash, subtract character to be removed.
- “Shift” by multiplying by 128.
- Add new character.
- Takes $\Theta(1)$ time.

```python
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
```

$\Theta(|p|)$

$\Theta(|p|)$

$\Theta(1)$

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

- Hashing because integer comparison takes $\Theta(1)$ time.
- 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))
250986132488946228262668052010265908722774302242017
```

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.

% q

Idea

q

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

```python
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, q):
    # 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) % q
```

Note

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

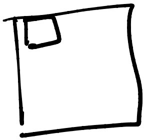
```python
def rabin_karp(s, p, q):
    hashed_window = base_128_hash(s, 0, len(p), q)
    hashed_pattern = base_128_hash(p, 0, len(p), q)
    match_locations = []

    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

$|s|/|p|$

- 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: $\Theta(|s| \cdot |p|)$.

s = aaaaaa

p = aaa