DSA3 Lab session 7

Administrivia first

- Kyle is once again (mentally) hanging by a thread today, gomenasai
- If there’s anything you wanted covered that we don’t go over today, tell me and we’ll look at it next week
- Should we just make labs movie time?

What is hashing good for?

- Fast indexing and data lookup
- Password security
  - Store a hash instead of the actual password
- Data integrity
  - Sending a hash (checksum) along with data
  - Recipient checks received data hash to check for corruption
- Many other uses…

We can, in principle, hash anything

- Step 1: Take anything, turn it into a really really big integer.
- Step 2: Use some hash function to map the integer down to hash table size m
- m is often a prime number to distribute keys evenly, exactly why is beyond the scope of this course
- Trust me bro

A hands-on example

- List: [27, 18, 29, 28, 39, 13, 16, 42]
- Hash function: h(k) = k % 11
- First question: how big is the hash table?

A hands-on example

- List: [27, 18, 29, 28, 39, 13, 16, 38]

- Hash function: h(k) = k % 11

- Second question: What does the hash table look like with chaining?

A hands-on example

- List: [27, 18, 29, 28, 39, 13, 16, 38]

- Hash function: h(k) = k % 11

0	1	2	3	4	5	6	7	8	9	10
		13			27	28	18			
					16	39	29			
					38					

A hands-on example

- List: [27, 18, 29, 28, 39, 13, 16, 38]
- Hash function: h(k) = k % 11
- Third question: What does it look like with linear probing?

A hands-on example

- List: [27, 18, 29, 28, 39, 13, 16, 38]

- Hash function: h(k) = k % 11

0	1	2	3	4	5	6	7	8	9	10
38		13			27	28	18	29	39	16

Hashing is totally awesome right?

- A basic array is O(1) insertion and O(n) lookup
- We can improve that with sorting
- Hashes are O(1)
- Can we think of any disadvantages that might lead us away from hashing in specific technical environments?

Let’s break this down in detail, it will help for your lab

```
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

0xffff is a hexadecimal

do you remember the binary conversion from TT?

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

0xffff is a hexadecimal

do you remember the binary conversion from TT?

What does it mean in terms of hashing?

```
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

<< and >> are bit shift operators, suppose x is 101101

x << 5 is 10110100000

x >> 5 is 1

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

<< and >> are bit shift operators, suppose x is 101101

x << 5 is 10110100000

x >> 5 is 1

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

Suppose x is 101101 and mask is 0xf

x << 2 is 10110100

x << 2 & mask is 0100

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

h << 5 and h >> 11 are not arbitrary numbers

The practical effect is cyclic shift

In the first loop, at the red arrow, what is the value of h?

```python
def cyclic_shift(s):
    mask = 0xffff
    h = 0
    for ch in s:
        h = (h << 5 & mask) | (h >> 11)
        h ^= ord(ch)
    return h
```

ord(ch) returns the Unicode code-point of ch

^= is XOR

What is the value of h after this line?

```python
def cyclic_shift(s):
  mask = 0xffff
  h = 0
  for ch in s:
    h = (h << 5 & mask) | (h >> 11)
    h ^= ord(ch)
  return h
```

2nd cycle, we just cyclic shift h and ^= the next ch
and again
and again

String matching

- The best way to understand the algorithms:
- just sit down and process the slides step-by-step
- We won’t cover FSA much today; we’ll deal with it in detail later

Brute force

- Easy to implement
- Actually viable for short search patterns
  - No preprocessing overhead means it can beat more elaborate algorithms in certain situations

- Suffers tremendously from adversarial conditions (consider the AAAAAAAAAAAAAAAAAC situation)
- Potentially lots of repeated work

Boyer-Moore

- O(n), O(nm) worst case
- In practice, mismatches often happen early when comparing right-to-left
- Can skip large portions of text

- Preprocessing required
- Suffers when the alphabet is tiny or the pattern has many repeating characters (the right-left mismatch advantage disappears)

FSA

- O(n) matching performance
- Great for matching the same pattern repeatedly
- Great for multi-pattern searching

- High precomputation overhead to build the transition table before actually matching

KMP

- O(n + m)
- Prefix table tracks partial matches
- Guaranteed linear worst case

- Can be slower in typical use cases than Boyer-Moore

Rabin-Karp

- O(n) typical
- Easier to implement
- Good for multi-pattern search (just compare against multiple hash codes)

- O(nm) worst case if lots of hash collisions