Algorithms in Bioinformatics

Exact pattern matching

Recap

- HW up, due next Tuesday
- No midterm - longer reading assignment instead
- Tomorrow: no office hours 2-3pm (sorry!)

Instead: tomorrow 11:30-noon
Friday: 11-12
↳ but please email to set up!

Today: Exact matching & repeat finding

- 50% of human genome is repeats
- However, repeats are also important!

(Go read Section 9.1 - associated with disease, evolution, etc.)

In particular: long, maximal repeats.

Different from motifs:

pattern is known

First tool: Hashing

Hash tables, dictionaries, or associative arrays, are built into most languages these days.

Side note:

Don't ever implement these yourself!

Hashing's goal:

- Book says duplicate removal.

This is one goal — but not the biggest one from a CS or BCB perspective!

Hashing: Fast data storage

Given key/value pairs, want to be able to retrieve value ~~quickly~~ $O(1)$ given the key.
(As well as store/update)

Examples:

- Course # & schedule info
- URL and html page
- Flight # & arrival info
- Color and BMP
- Directors & movies
- ℓ-mers & repetition locations in a sequence

⋮

Dictionary

A data structure which supports:

3 supported fcns

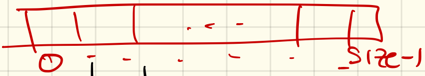
- insert (key, data)
- find (key)
- remove (key)

O(1)

Note: An array is a kind of dictionary!

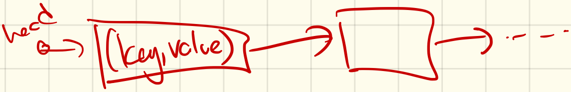
key: index/position

data: stored value



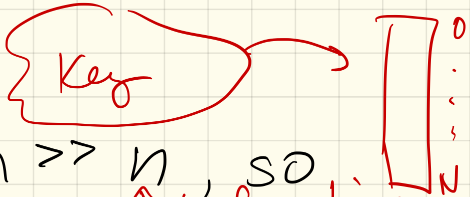
Other implementations:

Linked List:



Vectors

Hashing



Assume $\underset{\text{possible keys}}{m} >> \underset{\text{\# of entries}}{n}$, so array takes too much space.

Goal: $O(n)$ space
fast lookup/insert/remove

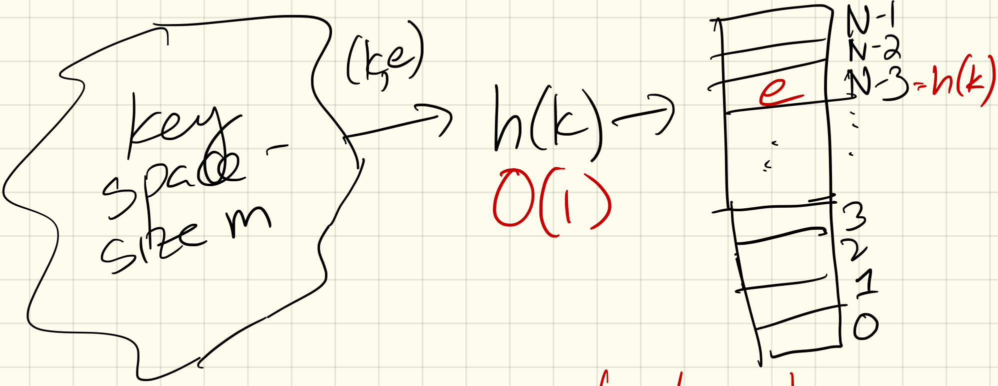
A hash function h maps each key to an integer in range $[0..N-1]$

Goal: N is bigger than n, but much smaller than m.

Then: Given (k,e), store it in $A[h(k)]$ (in an array).

Picture:

$$m \gg n \gg N$$



n actual values stored in array

Good hash functions:

- are fast $O(1)$!
- avoid collisions

↳ if $k \neq k'$ want $h(k) \neq h(k')$ with high probability

So, how to do this?

① Make the key a #

② Compress # to $[0, \dots, N-1]$

③ Handle collisions

① & ②: often combined,
& saw some of it
in data structure

We'll recap a bit...

<u>First idea</u>

For something like ASCII, can break into pieces & treat as bits:

$$\begin{tikzcd} E \arrow[d] & r \arrow[d] & i \arrow[d] & n \arrow[d] \\ 69 + & 114 + & 105 + & 110 = \# \end{tikzcd}$$ ☺

Then what?

<u>Problem</u>: this can backfire w/ words:

$$h(\text{temp01}) = h(\text{temp10}) = h(\text{pm0te1})$$

Want to <u>avoid</u> collisions.

So...

Polynomial Hash Codes

Split date to 32-bit pieces.

$$x = (x_0, \dots, x_{k-1})$$

Pick $a \neq 1$.

Let $p(x) =$

$$\underset{\uparrow t}{x_0}\, a^{k-1} + \underset{\uparrow e}{x_1}\, a^{k-2} + \cdots + \underset{\uparrow m}{x_{k-2}}\, a + \underset{\uparrow}{x_{k-1}}$$

Ex: Erin (or $\underline{69}$, $\underline{105}$, $\underline{114}$, $\underline{110}$)

and $a = 37$:

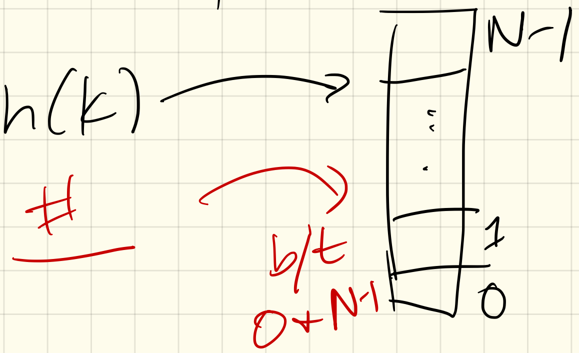
$$p(x) = 69 \cdot 37^3 + 105 \cdot 37^2 + 114 \cdot 37 + 110$$

Why?

- relatively fast
- avoids collisions!

(more tricks like this)

Next: Compress:



Idea: Take $h(k) \bmod N$ % in C

✓ Python

Recall: 3 mod 10 = 3

60 mod 10 = 0

14 mod 10 = 4

Example: $h(k) = k \bmod \underset{N}{\underline{11}}$

A:

0	1	2	3	4	5	6	7	8	9	10
	(12, E)			(37, I)	(16, N)					(21, R)

Insert: key

$(12, E)$: $h(12) = 12 \bmod 11 = 1$

$(21, R)$: $h(21) = 10$

$(37, I)$ $h(37) = 4$

$(16, N)$ $h(16) = 5$

$(26, C)$ $h(26) = 4$ X

$(5, H)$

Comment: Works best if #s are relatively prime.

Why? go take number theory

Another way: M.A.D

Instead of $h(k) \bmod N$,

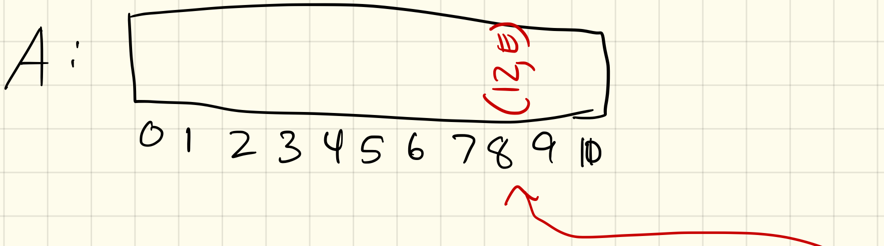
do $\underline{h(k)} = |\underline{a}k+\underline{b}| \bmod N$

where a + b are:

- relatively prime
- less than N

Why? go take NT

Example: $h(k) = 3k+5 \bmod 11$



Insert:

$(12, E)$ $\overset{h(12) =}{=}$ $3 \cdot 12 + 5 \bmod 11 = 8$

$(21, R)$

$(37, I)$

$(16, N)$

$(26, C)$

$(5, H)$

(collisions may still happen)

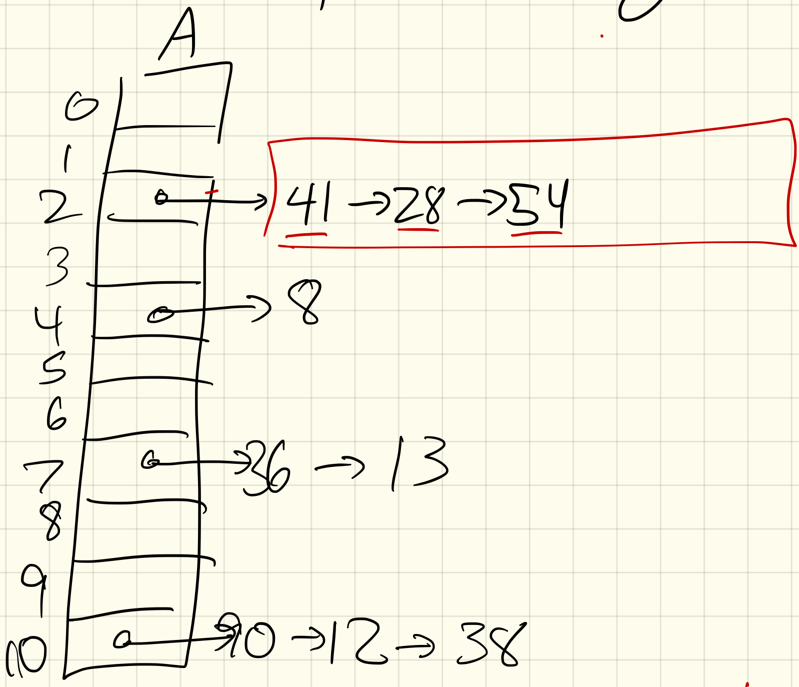
Why bother?

MUCH better in practice

Step 3: Handle Collisions

(Hint: What data structures can store more than 1 thing??)

Ex: Simple Chaining:



Run times: Worst case, bad hash function.
↳ insert/lookup list time

Other techniques:

- linear probing
- quadratic probing
- re-hashing

Takeaway:

Handle collisions
On most data, all of
these work well in
practice.
(No theoretical guarantee)

Load Factors

Whatever method you use, usually starts to do badly if n gets close to N:

Want $\frac{n}{N} < .5$

more than half full

Re hashing:

When more than half full, most implementations double the array size + choose a new hash function

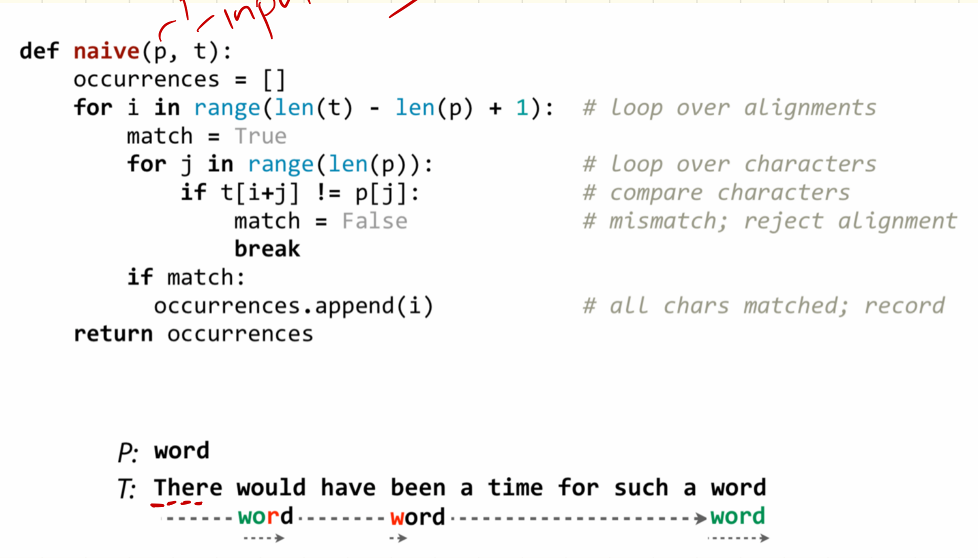
(Hence, don't write these yourself!)

Back to pattern matching

Naive Pattern matching:

pattern

input string (longer)



```
def naive(p, t):
    occurrences = []
    for i in range(len(t) - len(p) + 1):  # loop over alignments
        match = True
        for j in range(len(p)):           # loop over characters
            if t[i+j] != p[j]:            # compare characters
                match = False             # mismatch; reject alignment
                break
        if match:
          occurrences.append(i)           # all chars matched; record
    return occurrences
```

P: word

T: There would have been a time for such a word

Runtime:

$|p| = n$

$|T| = m$

$n(m-n+1) = O(mn)$

[Boyer-Moore]

How to improve?

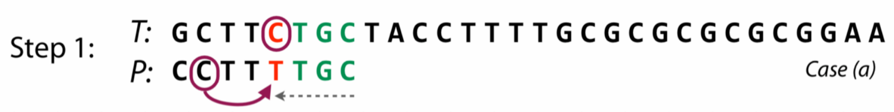
1) Skip pointless alignments:

("Bad character rule")

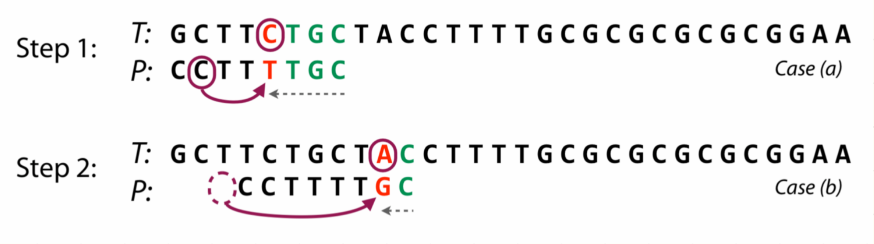
Align P at start of T:

a) Look at position of the last occurance of a mismatching character

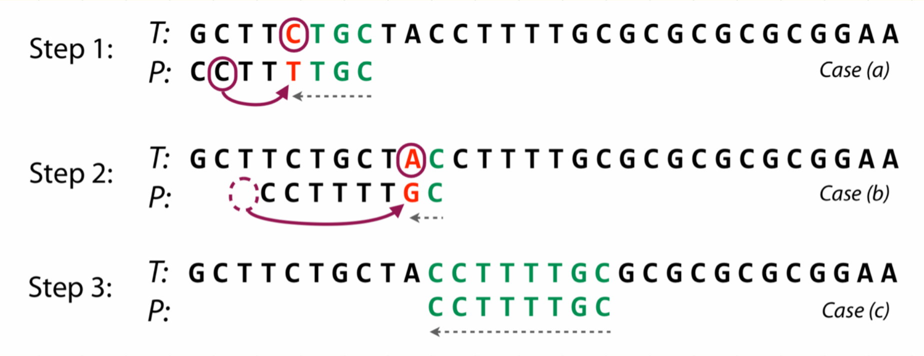
If this character exists in pattern, realign to last (prior) occurance



b) If that character isn't in pattern, just go past entirely


Step 1:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C
Case (a)
Step 2:
T: G C T T C T G C T A C C T T T T G C G C G C G C G G A A
P: C C T T T T G C
Case (b)

(c) Of course, if you don't find this 'character', you've hit a match!



Step 1: T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (a)

Step 2: T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (b)

Step 3: T: G C T T C T G C T A C C T T T T G C G C G C G C G C G G A A
P: C C T T T T G C
Case (c)

Run-time of this:

Still O(mn), since could get all the same character:

AAA A

AA AA - - - - A

② Good suffix rule

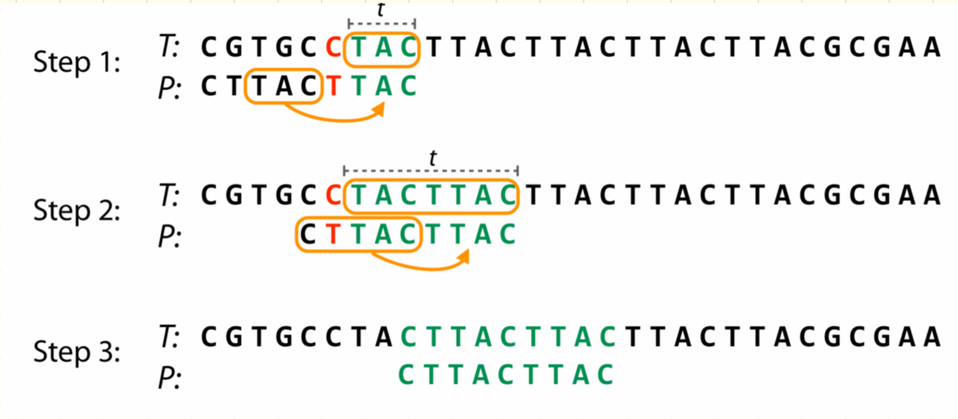
Let t = substring matched by the inner loop

Look at suffixes:

Skip until either

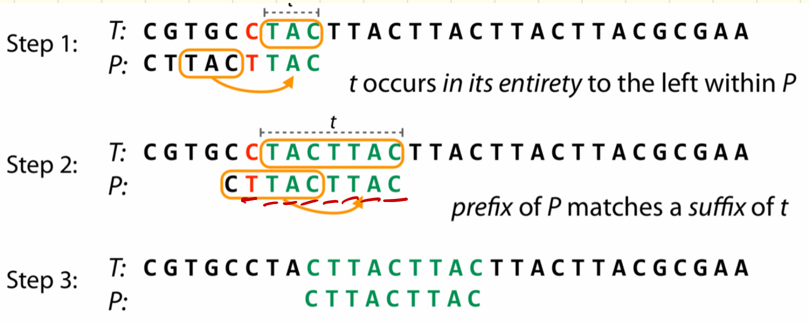
Ⓐ no mismatches between P + t

Ⓑ P moves past t



t
Step 1:
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
t
Step 2:
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
Step 3:
T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC

Note: Can break @ down to cases:



Step 1: T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
t occurs *in its entirety* to the left within P

Step 2: T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC
prefix of P matches a *suffix* of t

Step 3: T: CGTGCCTACTTACTTACTTACTTACGCGAA
P: CTTACTTAC

Algorithm:

Tradeoff b/t these rules

just suffix rule: O(mn)