Data Structures & Algorithms 2

Topic 9 – String Searching Algorithms

String Searching

- String searching finds the location of a specific text pattern within a larger body of text (e.g., a sentence, a paragraph, a book)
- As with most algorithms, the main considerations for string searching are speed and efficiency

String Searching

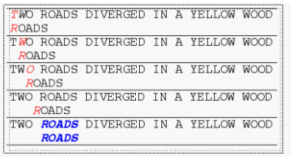
- Efficient string searching is of vital importance in many areas

Word processors
Virus scanners
Digital libraries
Web search engines
Bioinformatics

- We will examine four algorithms that match an exact string of text within a larger document
  - Naïve Search
  - Rabin-Karp
  - Knuth-Morris-Pratt
  - Boyer-Moore

Naïve String Search

- This is the kind of algorithm that you would come up with intuitively
- The Naïve Search or *Brute Force* algorithm compares the pattern to the text, one character at a time – not very clever!



Naïve String Search

Pseudo-Code

```
while (! entire pattern found OR end of text){
  if (text letter == pattern letter) {
      compare next letter of pattern to next letter of text
  }else {
      move pattern down text by one letter
  }
}
```

Efficiency of naïve search

Given a pattern P characters in length, and a text N characters in length.....

- Best case scenario: always mismatch on first character
  - Total number of comparisons: N
  - Best case time complexity: O(N)

Efficiency of naïve search

Given a pattern P characters in length, and a text N characters in length.....

- Worst case scenario: compares each character in pattern to each substring of text
  - Total number of comparisons: P (N-P+1) or O(NP)
  - More likely to occur using a small alphabet

Actual performance will be somewhere in between

Not fast enough

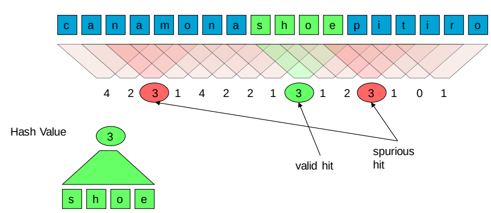
- Most major databases are growing at a very fast rate as disk space increases
- The GenBank database contains more than 200 terabytes of DNA sequences
- Google runs over one million servers in data centres around the world, and processes over one billion search requests and 24 petabytes of user-generated data each day
- We need a more efficient algorithm, especially if we are searching for long strings or multiple terms

Rabin-Karp String Search

- Frequently used for searching many strings at the same time because its complexity is not affected by the number of search strings

1. Involves the computation of hash values for every successive substring of the text to be searched
2. If the hash values of the search string and the text are unequal, the algorithm will calculate the hash value for next character sequence
3. If the hash values are equal, the algorithm will do a *Naïve search comparison* between the pattern and the character sequence

- A simple example of a hash value could be simply adding successive power of 26 and modulo-ing 5 by so "shoe" would have a hash value of 19 + 8*(26^1) + 15*(26^2) + 5*(26^3) % 5 = 98248 % 5 = 3
- A good hash function will ensure that there is little overlap between hash values – fewer false positives
- Lets assume we're searching for "shoe" which has a hash value of 3



Increasing Efficiency

- We don't want to have to re-compute a hash value based on P characters for every character in the text we're searching through
- This would yield O(NP) since the bigger the string we're searching for, the more calculation involved in computing the hash value
- For the most efficient O(N) computation of hash values, rolling hash values are used
- This means that as the hash window moves along you can figure out the new hash value just by taking into account the characters entering and leaving the "hash window"
- You don't need to consider the characters in between
- The length of the string has no impact on the difficulty of calculating rolling hash

Hash Function

- Different letters are raised to the power of a base number
- The letters are raised to different powers according to their position in the substring
- The total value is then modulo-ed to bring it into the range of the HashArray
- Using a rolling hash avoids having to redo the whole calculation every time

Example

"proverb" in ASCII is:
112
114
111
118
101
114
98

so the hash value is:
$112 \times 256^6 +$
$114 \times 256^5 +$
$111 \times 256^4 +$
$118 \times 256^3 +$
$101 \times 256^2 +$
$114 \times 256 +$
98
% 100,003 =

which gives:
43,552 +
49,505 +
68,046 +
52,100 +
18,939 +
29,184 +
98
% 100,003 =
61,418

Base = 256
Modulo = 100,003

Rolling Hash

- We hash the characters from 1 -7 in the text to be searched
- We don't get a match so now we move the hash window along: we want to get the hash values for characters 2 – 8
- However, if we know the hash value for characters 1 – 7 and we know the letters leaving and entering the hash window (characters 1 and 8) then the calculation is simple:
    - Subtract the letter leaving the window (e.g. *p*) - 112×256^6
    - Now multiply the remaining value by the base (256) so that you're effectively increasing all the powers of 256 for the remaining letters by 1 (e.g. 114×256^5 becomes 114×256^6 etc.)
    - Finally add in the value of the new letter entering the window

Example

Say the new letter is a space (ASCII values is 32)

$112 \times 256^6 +$ ← Subtract this
$114 \times 256^5 +$
$111 \times 256^4 +$
$118 \times 256^3 +$
$101 \times 256^2 +$ ← Multiply what's left by 256
$114 \times 256 +$
98
% 100,003 ← Now add in 32

Base = 256
Modulo = 100,003

which gives:

61,418
\- 43552
\+ 32
= 17898

Rabin-Karp Efficiency

- If a sufficiently large prime number is used for the *hash function*, the hashed values of two different patterns will usually be distinct so there are no false positives
- If this is the case, searching takes O(N) time, where N is the number of characters in the larger body of text
- It is always possible to construct a scenario with a worst case complexity of O(NP) where every hash value triggers a false positive and must be checked
- This, however, is likely to happen only if the prime number used for hashing is too small to produce unique hash values

Multiple Strings

- The really great thing about Rabin Karp is that it is the only algorithm that lets you search for as many strings as you want without affecting runtime
- With other string searching algorithms the complexity would be O(NK) where K is the number of strings being searched for
- In other algorithms each character must be checked against characters in all K strings
- Rabin-Karp uses hash tables and since hash tables have O(1) lookup time the number of strings being searched for has little effect – complexity is O(N + K)

Multiple Strings

- All of the K strings you're looking for are hashed to give their hash values
- These are then bundled into a hash table according to their hash value
- We then go through the text computing hash values and looking up the hash table to see if the character sequence matches a string we're looking for

Example

- We're searching for "frog", "bird", "goat" and "fish"
- We pass these strings into the hash function and the following values emerge
  - "frog" → 3
  - "bird" → 6
  - "goat" → 1
  - "fish" → 2
- We then insert these strings into a hash table according to their hash values

0	
1	"goat"
2	"fish"
3	"frog"
4	
5	
6	"bird"

Example

- Now we start searching through our text

T	h	e		f	r	o	g		j	u	m	p	e	d

- We hash each consecutive four letter sequence yielding a hash table index to look up

T	h	e		→ 0 which is empty
h	e		f	→ 4 which is empty
e		f	r	→ 1 which contains a string but naïve comparison reveals they don't match despite having the same hash value
	f	r	o	→ 5 which is empty
f	r	o	g	→ 3 which contains a string and which does match

Rabin-Karp against plagiarism

- The size of the hash table and the number of strings in it has no effect on the complexity of the algorithm
- Because we're using a rolling hash, the length of the strings don't matter either: Rabin-Karp is often used to detect plagiarism (e.g. Turnitin)
- If a class of students hand up essays and you want to check if they've copied any material off Google then you're going to need to search for multiple strings at the same time
- You could obtain sample sentences on the essay topic from Google and stick a selection of substrings (fix some size, e.g. 10) in a hash table
- Rabin Karp runs fine and will tell you if any of those substrings appear in any of the essays in O(N + K) time where N is the total length of all the essays and K is the number of substrings you're checking for

Knuth–Morris–Pratt Algorithm

- However, Rabin Karp cannot *guarantee* that the search will take place in linear time - in a worse case scenario, where every step triggers a false hash match, it runs in O(NP + K) time (the false matches have to be checked using brute force)
- Knuth-Morris-Pratt (KMP) guarantees a worse case running time of O(N + P)
- For this reason Rabin-Karp is only used when you're searching for multiple strings
- The idea behind KMP is simply that you remember characters you've seen before so you don't have to look at them again

Knuth–Morris–Pratt Algorithm

- When a mismatch occurs, the algorithm remembers the letters it has checked, avoiding re-examination of previously matched characters
  - A partial match table indicates how much of the last comparison can be reused if it fails
  - Looks out for a pattern that matches the start of the one you're looking for while you're in the middle of checking a potential hit
  - That way it remembers how far back you need to go after a failed comparison and saves you rechecking the same characters again

Complexity of KMP

- The algorithm has two parts
  1) partial-match-table building algorithm
  2) comparison algorithm
- Efficiency of comparison is O(N) where N is the length of text we're searching
- Efficiency of the table-building algorithm is O(P) where P is the length of the pattern
- Therefore, the complexity of the overall algorithm is O(N + P)
- KMP guarantees that the search will take linear time both in best and worst case scenarios

Partial Match Table

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1	0	1	2	3

- The partial match table is constructed before the algorithm begins
- We pre-search the **pattern**
- For each position we produce a number which indicates how far we can move up the pattern if there is a mismatch
- For example, the number 2 associated with the last R means "if you mismatch while checking this character then move the pattern up so that slot 2 is at this position"
- The first character A is assigned -1 because it is a special case (if you mismatch on the first character, just move the pattern up another space)

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1										

-1 always goes in the first slot, telling you move the pattern up a space

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0									

Until we see a repeat of the first character keep filling in zeroes

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0								

Until we see a repeat of the first character keep filling in zeroes

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0							

Now ask the question – how many characters of the beginning of the pattern have we just gone past?

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1						

We've just gone past an A. So that's one.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0					

After the A we saw a C. That's not the start of the pattern. So we're back down to scratch.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1				

We've just gone past an A again. So we've matched one letter from the start of the pattern.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1	0			

But the D isn't what we're looking for. So back to 0 we go.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1	0	1		

We're after going past an A again. That's one in the bag.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1	0	1	2	

Now we've gone past a B. A and B are the two starting letters of the pattern. So we're up to 2.

Partial Match Table

- Compute the Partial Match table for the pattern ABRACADABRA

A	B	R	A	C	A	D	A	B	R	A
-1	0	0	0	1	0	1	0	1	2	3

Now we've gone past ABR. That's the first 3 letters. So we're up to 3.

Example

- Show the KMP algorithm would find the pattern ABCDABD in the text ABC ABCDAB ABCDABCDABD. How many comparisons are required?
- First compute the partial match table, then show the comparison table

Partial Match Table

A	B	C	D	A	B	D
-1	0	0	0	0	1	2

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D   Pattern

Text

- A, B and C check out fine
- There is a mismatch on D
- Brute force would now start the whole process again at the second character in the text
- KMP lets us slide the pattern up as much as possible given the matches we have already identified
- The partial match table has a 0 for this character in the pattern, so move the pattern up until slot 0 is at this point

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- That doesn't match either
- The partial match table always has a -1 for the first character
- Slide up the pattern so that slot "-1" is at this mismatch point – in other words, move the pattern up one space

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- A, B, C, D, A and B check out fine
- There is a mismatch on the D
- KMP lets us slide the pattern up as much as possible given the matches we have already identified
- The partial match table has the number 2 for the last character in the pattern
- Move up the pattern so that its slot 2 is at this point

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- The next letter C is a mismatch
- The partial match table gives us a 0
- Slide the pattern up so that its first character (the one at slot 0) is at this point

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- Another mismatch
- The partial mismatch table gives us a -1
- Whenever there is a mismatch on the first character of the pattern, move the pattern up another space

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- A, B, C, D, A and B are matched
- There is a mismatch on the final character of the pattern
- The partial match table tells us to move up the pattern so that slot 2 goes here

Example

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

- We eventually get a complete match
- The performance of the KMP algorithm is O(N + P) in the worst case scenario
- The O(N) part comes from the comparisons
- The O(P) part comes from the time needed to initially construct a partial match table which tells you which character in the pattern to check next following a mismatch

Comparison Table

A B C   A B C D A B   A B C D A B C D A B D

A B C D A B D

A B C D A B D

A B C D A B D

A B C D A B D

A B C D A B D

A B C D A B D

A B C D A B D

- There are 26 comparisons required in total

Boyer-Moore Algorithm

- Boyer-Moore string searching algorithm works backwards
- The longer the pattern you are looking for, the quicker the algorithm runs
- Unlike KMP which must look at every character in the text, Boyer-Moore attempts to ignore as many characters as it can to boost performance
- Boyer-Moore has worst-case running time of O(N + P) only if the pattern does not appear in the text. If the pattern does appear it is O(NM) in the worst case

Boyer-Moore Algorithm

- Boyer-Moore is handy when the string we're searching for is very long – maybe a sentence with 150 characters
- The idea is that we immediately jump to character 150 in the text and see if this matches the last character in the pattern
- Say character 150 is an "x" and there is no x in the pattern we are searching for
- We now know that the first 150 characters cannot be part of our pattern so we can ignore them completely – we don't even need to check the characters!

Boyer-Moore Algorithm

- Boyer Moore essentially works backwards, starting with the last character in the pattern and working back towards the first
- Every time there is a mismatch is knows how far ahead it is allowed to jump
- The longer the pattern, the quicker the algorithm can run because it will be able to make bigger jumps

Boyer-Moore Algorithm

- Two heuristics
  1. Looking-glass: when comparing `P` against a substring of `T`, start the comparison at the *end* of `P`, not the start
  2. Character-jump: if a comparison fails at `T[j] = c`, then
     - If c does not occur in `P`, shift `P` completely past `T[j]`
     - Otherwise, shift `P` until the last occurrence of `c` in `P` is aligned with `T[j]`
- The two complement each other
  - Looking-glass tries to find a distant mismatch which character-jump tries to exploit

Example

- Let `Text = ADCDABEAB` and `Pattern = ABE`

A D C D A B E A B
↑
A B E

- Check `ADC` against `ABE`
- By looking-glass we find the mismatch `C` first
- `C` does not occur in the pattern, so shift the pattern past the mismatch

Example

A D C D A B E A B
↑
A B E

- Check `DAB` against `ABE`
- Mismatch at `B`
- `B` does occur in `ABE` so shift the pattern to align the `B` in the text with the first `B` in the pattern

Example

A D C D A B E A B
↑ ↑ ↑
A B E

- Check `E` against `E`
- Check `B` against `B`
- Check `A` against `A`
- Succeed

Boyer-Moore Algorithm

- The algorithm computes two "jump tables" containing information allowing it to calculate how far it can jump after a mismatch
- The first table states how many positions in front of the point of mismatch the pattern can be shifted (e.g. if mismatch on X after first comparison then jump ahead 150 since X not in pattern)
- Sometimes we can jump even further than the character-based estimation
- The second table takes into account letters that have been already checked before a mismatch occurs and states how many positions in front of the point of mismatch the pattern can be shifted based on the already partially matched pattern
- The algorithm jumps the greater of the amounts in the two jump tables

Example First Table

- ***Example****: For the string ANPANMAN, the first table would be as shown (for clarity, entries are shown in the order they would be added to the table):*
- The amount of shift calculated by the first table is sometimes called the "bad character shift"
- If the algorithm checks a character in the text and it's a P then the end of the pattern can be jumped another 5 places from the point of mismatch because the first P in the pattern is 5 places from the back

Character	Shift
N	0
A	1
M	2
P	5
All others	8

Bad Character Shift

- The pattern is A N P A N M A N
- First step is to create a table with a shift for each character in it
- The shift for all other characters is always the length of the pattern (i.e. if you mismatch on a character that's not in the pattern, shift the pattern all the way past this point)

Character	Shift
N	
A	
M	
P	
All others	8

Bad Character Shift

- The pattern is A N P A N M A N
- How many characters from the back does the first N appear?
- That's 0

Character	Shift
N	0
A	
M	
P	
All others	8

Bad Character Shift

- The pattern is A N P A N M A N
- How many characters from the back does the first A appear?
- That's 1

Character	Shift
N	0
A	1
M	
P	
All others	8

Bad Character Shift

- The pattern is A N P A N M A N
- How many characters from the back does the first M appear?
- That's 2

Character	Shift
N	0
A	1
M	2
P	
All others	8

Bad Character Shift

- The pattern is A N P A N M A N
- How many characters from the back does the first P appear?
- That's 5

Character	Shift
N	0
A	1
M	2
P	5
All others	8

Example Second Table

- ***Example**: For the string ANPANMAN, the second table would be as shown. It tells you how much you can jump based on failures at different points in the checking process*
- The amount of shift calculated by the first table is sometimes called the "good suffix shift"
- If the pattern mismatches on the second check (after matching N, mismatching on A) then the pattern can be moved up 8 spaces because there is no substring anywhere in the pattern which involves an N preceded by (NOT an A) – it's as good as mismatching on an X in the text

Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	3
~~N~~MAN	6
~~A~~NMAN	6
~~P~~ANMAN	6
~~N~~PANMAN	6
ANPANMAN	6

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the back character then we know the last character is "not an N"
- We want to move the pattern up as much as possible based on this information
- "Not an N" could be an A and A is the next character
- We can only move the pattern up one space then

Sub-Pattern	Shift
~~N~~	1
~~A~~N	
~~M~~AN	
~~N~~MAN	
~~A~~NMAN	
~~P~~ANMAN	
~~N~~PANMAN	
~~A~~NPANMAN	

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the second character from the back then we know the last character was an N and the second last character was not A
- Move the pattern up as much as possible until you get a match for ~~A~~N
- There are no matches – we can move the pattern all the way past

Sub-Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	
~~N~~MAN	
~~A~~NMAN	
~~P~~ANMAN	
~~N~~PANMAN	
~~A~~NPANMAN	

~~A~~ N
A N P A N M A N

~~A~~ N
8 spaces
A N P A N M A N

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the third character from the back then we know the last character was an N, the second last was an A and the third last was "not an M"
- Move the pattern up as much as possible until you get a match for ~~M~~AN
- We can shift the pattern up 3 spaces

Sub-Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	3
~~N~~MAN	
~~A~~NMAN	
~~P~~ANMAN	
~~N~~PANMAN	
~~A~~NPANMAN	

~~M~~ A N
A N P A N M A N

~~M~~ A N
3 spaces
A N P A N M A N

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the fourth character from the back then the subpattern to match is ~~N~~MAN
- Move the pattern up as much as possible until you get a match for ~~N~~MAN
- We can shift the pattern up 6 spaces

Sub-Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	3
~~N~~MAN	6
~~A~~NMAN	6
~~P~~ANMAN	
~~N~~PANMAN	
~~A~~NPANMAN	

~~N~~ M A N
A N P A N M A N

~~N~~ M A N
6 spaces
A N P A N M A N

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the fifth character from the back then the subpattern to match is ~~A~~NMAN
- Move the pattern up as much as possible until you get a match for ~~A~~NMAN
- We can shift the pattern up 6 spaces

Sub-Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	3
~~N~~MAN	6
~~A~~NMAN	6
~~P~~ANMAN	
~~N~~PANMAN	
~~A~~NPANMAN	

~~A~~ N M A N
A N P A N M A N

~~A~~ N M A N
6 spaces
A N P A N M A N

Good Suffix Shift

- The pattern is A N P A N M A N
- If we mismatch on the sixth, seventh or eighth character from the back it's all the same
- We can always shift the pattern up 6 spaces before getting a match with the partial pattern

Sub-Pattern	Shift
~~N~~	1
~~A~~N	8
~~M~~AN	3
~~N~~MAN	6
~~A~~NMAN	6
~~P~~ANMAN	6
~~N~~PANMAN	6
~~A~~NPANMAN	6

~~A~~ N P A N M A N
A N P A N M A N

~~A~~ N P A N M A N
6 spaces
A N P A N M A N

Example

A N C D A N M A N P A N M A N — Text

A N P A N M A N — Pattern

- With the two tables computed let's go through the comparisons
- Check the last character
- Mismatch because it's not an N, it's an A
- Bad character shift for A is 1 (first table)
- Good suffix shift for a first check is also 1 (second table)
- Shift the pattern up 1 from the point of mismatch

Example

A N C D A N M A N P A N M A N

A N P A N M A N

- N, A, M, N and A match
- The P in the pattern is mismatched with a D in the text
- The bad character shift for a D is 8 places from the point of mismatch (i.e. the end of the pattern ends up 8 spaces beyond where the red X is, which is a net shift of just 3 positions)
- The good suffix shift for (Not P) ANMAN is 6
- The good suffix shift allows the bigger jump so take it

Example

A N C D A N M A N P A N M A N

A N P A N M A N

- Match N, A, M, N, A, P, N and A
- Success

Summary of comparisons

- There are a total of 15 comparisons needed

```
A N C D A N M A N P A N M A N
        ✗
A N P A N M A N
      ✗ ✓ ✓ ✓ ✓ ✓
  A N P A N M A N
              ✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓
              A N P A N M A N
```

Boyer Moore Efficiency

- The best case performance of the algorithm is O(N/P) (jumps the whole distance every single time)
- Boyer-Moore has worst-case running time of O(N + P) only if the pattern does not appear in the text. If the pattern does appear it is O(N*P) in the worst case (never jumps ever, this can happen with small alphabets)
- Therefore, Boyer-Moore is handy when P is large and you don't expect too many hits (i.e. you're looking for a long complex pattern which not even appear)

Quick Comparison

- Let `P = "abacab"` and `T = "abacaabaccabacabaabb"`
  - Brute Force takes 29 steps
  - Boyer-Moore takes 19 steps
  - Knuth-Morris-Pratt also takes 19 steps
- If we change one letter so `T = "abacaabacdabacabaabb"`
  - Brute Force still takes 29 steps
  - Boyer-Moore takes 16 steps
  - Knuth-Morris-Pratt still takes 19 steps
- Boyer-Moore improves mainly because `"d"` is not in the pattern
- The "best" search depends critically on the structure of the text and pattern, which won't be known *a priori*

Conclusion

- Searching for patterns in text is quite expensive, but can be improved using heuristics - approaches that aren't *guaranteed* to improve performance, but *typically do* in common cases
- The fastest heuristic depends on the exact details of both text and pattern, so is hard to determine in general
- Either BM or KMP typically do OK, with KMP often being better for patterns with lots of internal repetitions
- KMP gives us a guaranteed worst case search time, BM is riskier but can enhance performance substantially when big jumps are frequent
- Rabin-Karp is rarely used for single searches because of its poor worse case scenario