String Searches

Thomas Schwarz, SJ

Problem

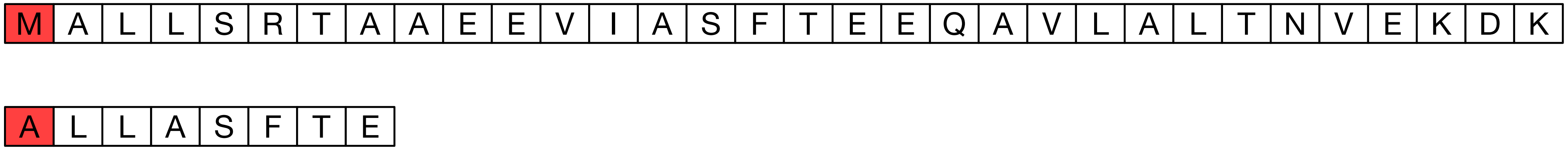
- We are given a long string (*text*)
  - such as a book or a genome
- We are given a short string (*pattern*)
- We want to find where the shorter string is located in the longer one

Naïve Algorithm

- We slide the pattern successively through the text
- We compare the letters in the pattern with the text
- If two letters differ, go to the next location
- If we reach the end, we have found a match

Naïve Algorithm

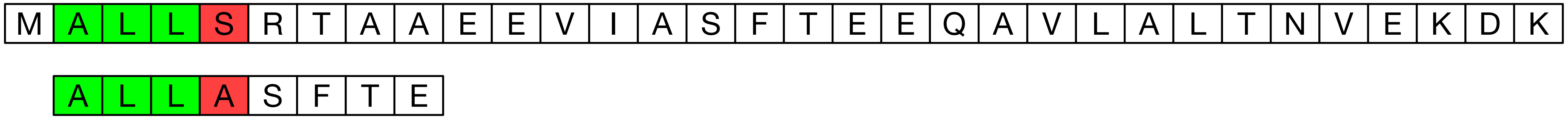
- Example



M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K

A	L	L	A	S	F	T	E

- No match on first character: move pattern by one



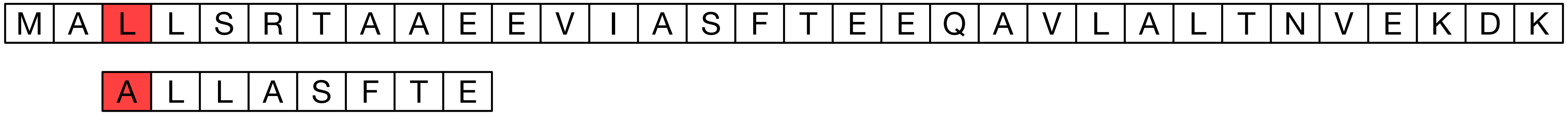
M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K

A	L	L	A	S	F	T	E

- After sliding, three letters coincide, but then we have a mismatch: move pattern by one

Naïve Algorithm

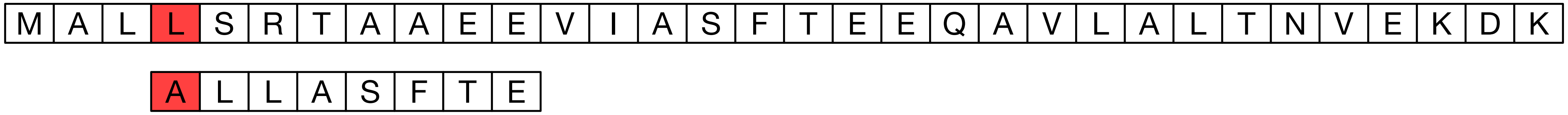
- Example:



M A L L S R T A A E E V I A S F T E E Q A V L A L T N V E K D K

A L L A S F T E

- No match on first, slide



M A L L S R T A A E E V I A S F T E E Q A V L A L T N V E K D K

A L L A S F T E

- No match on first, slide
- ...

Naïve Algorithm

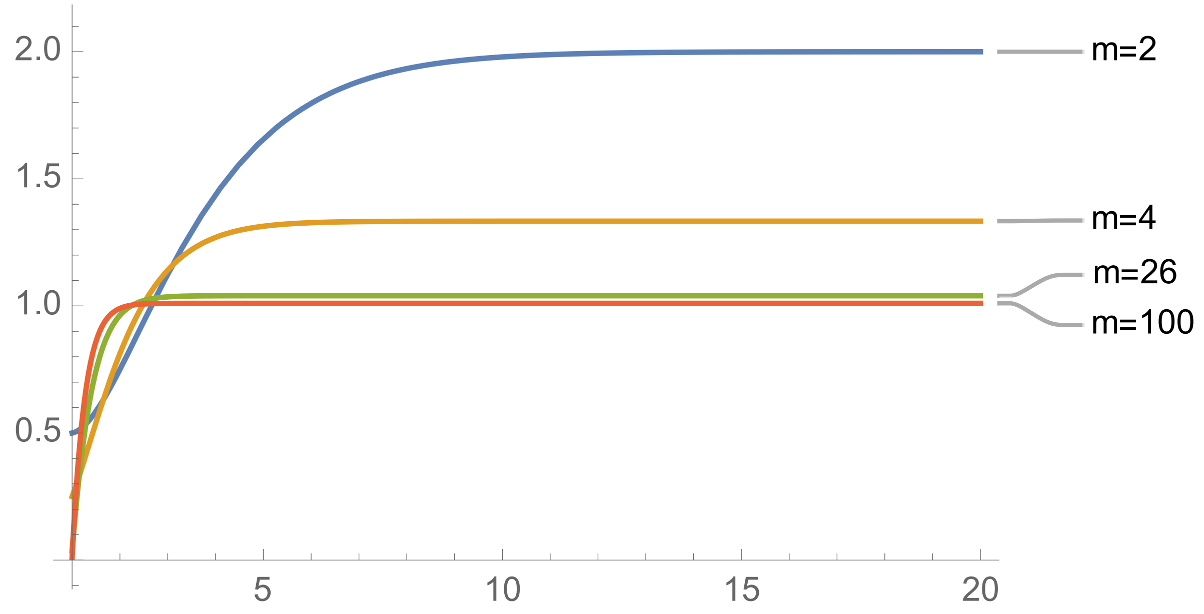
- What are the costs:
    - At best, we compare each letter of the text with a letter in the pattern
    - n: length of pattern
    - m: length of string
- Best time: n
- Worst time: nm

Naïve Algorithm

- Average time:
  - Depends on how likely matches between letters are
  - If we assume there are c characters and all are equally likely and the appearance of a character is independent of its neighbors and ... :
  - Probability of a character matching is $1/c$
  - Expected number of characters compared is
  - $\frac{c-1}{c} \cdot 1 + \frac{c-1}{c^2} \cdot 2 + \ldots + \frac{c-1}{c^{m-1}} \cdot (m-1) + \frac{1}{c^m} \cdot m$

Naïve Algorithm

- Average time:
- $\frac{c-1}{c} \cdot 1 + \frac{c-1}{c^2} \cdot 2 + \ldots + \frac{c-1}{c^{m-1}} \cdot (m-1) + \frac{1}{c^m} \cdot m$
- Converges quickly to $\frac{c}{c-1}$

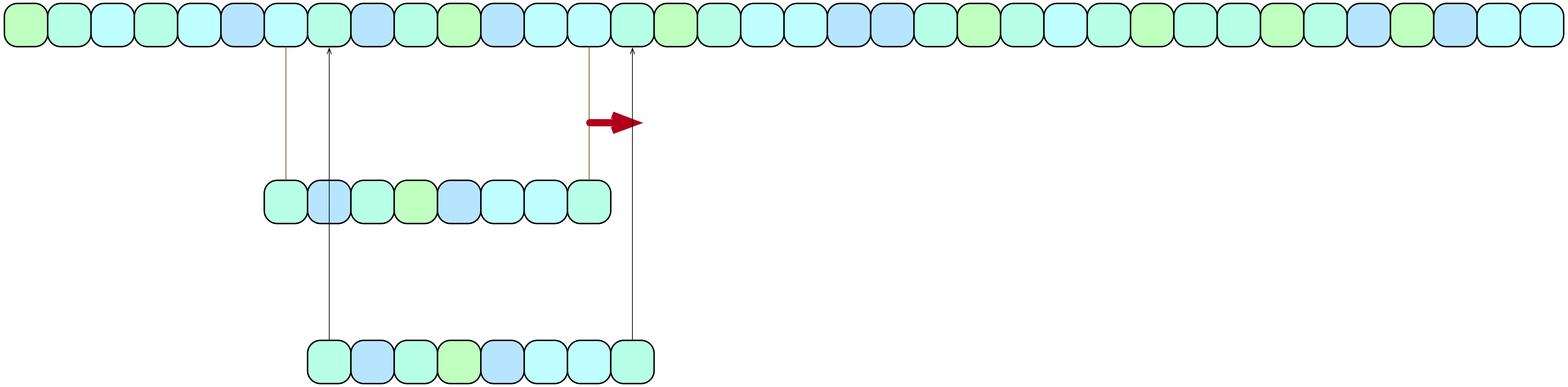


Naïve Algorithm

- Thus:
    - Average number of comparisons is close to 1

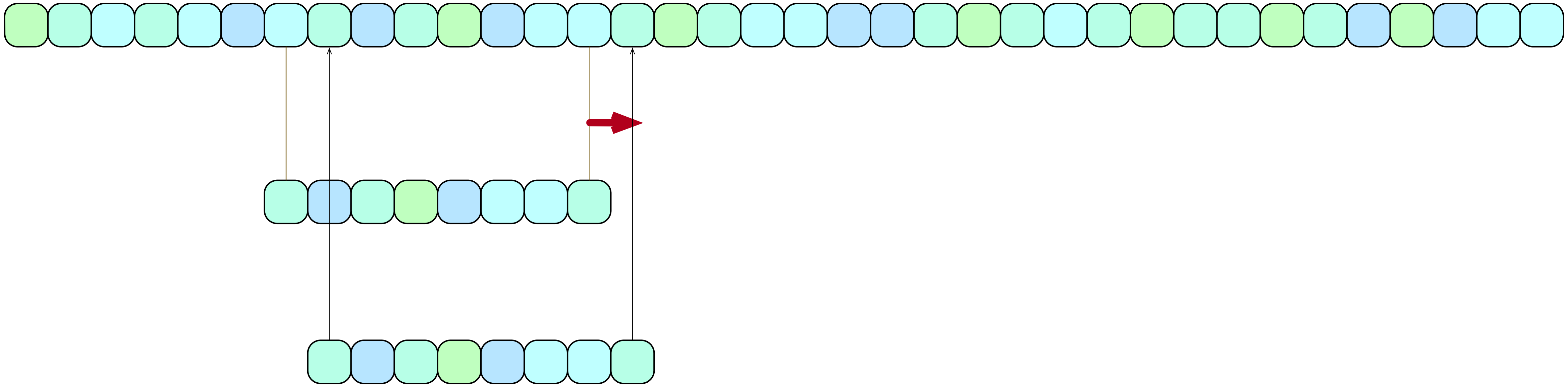
Karp Rabin

- Idea:
  - Use a hash function to compare a sub-string with the pattern
  - Hash function needs to be calculated from a sliding window:



Karp Rabin

- Idea:
  - The hash for a window needs to be calculated from:
    - the previous hash
    - the element leaving the sliding window
    - the element entering the sliding window



Karp Rabin

- Example for Rabin Hashes:
  - Assign values $v(l)$ to each letter l in the alphabet
    - Finite-field elements
    - Integers
  - Use

$$\rho(a_i, a_{i+1}, a_{i+2}, \ldots, a_{i+n-1}) = \sum_{\nu=0}^{n-1} \alpha^{n-\nu} v(a_{i+\nu})$$

Karp Rabin

$$\rho(a_i, a_{i+1}, a_{i+2}, \ldots, a_{i+n-1}) = \sum_{\nu=0}^{n-1} \alpha^{n-\nu} v(a_{i+\nu})$$

```
def rabin(word):
    suma = 0
    for i, letter in enumerate(word):
        suma += (g**(len(word)-i-1)*ord(letter)) % p
    return suma % p

def rabin2(word):
    return sum( (g**(len(word)-i-1)*ord(word[i])) % p
                for i in range(len(word)))%p
```

Karp Rabin

- Then calculate the effect of a shift by one to the right

$$\rho(a_{i+1}, a_{i+1}, \ldots, a_{i+n})$$

$$= \alpha^n a_{i+1} + \alpha^{n-1} a_{i+2} + \alpha^{n-2} a_{i+3} + \ldots + \alpha a_{i+n-1} + a_{i+n}$$

$$= \alpha \left(\alpha^n a_i + \alpha^{n-1} a_{i+1} + \alpha^{n-2} a_{i+2} + \ldots + \alpha a_{i+n-2} + a_{i+n-1} \right)$$

$$- \alpha^{n+1} a_i + a_{i+n}$$

$$= -\alpha^{n+1} a_i + \alpha \rho(a_i, a_{i+1}, \ldots a_{i+n-1}) + a_n$$

Karp Rabin

- Thus:
  - We can calculate the Rabinesque hash from the previous hash, the entering element, and the leaving element
  - This is the *shift*

Karp Rabin

- The algorithm begins by calculating the ρ of the pattern and of the first len(pattern) letters in the text
- Using the shift, we compare the ρ of a portion of the text with the ρ of the pattern. If they are the same, then we have a possible occurrence.
- We still need to verify.

Karp Rabin

- Implementation:
  - Pick a prime p just below a power of 2 (and much larger than the values of the letters in the alphabet)
  - Find a good value for α
    - Best choice is a generator g
      - The powers of g make up all the values between 1 and $p-1$

Karp Rabin

- Implementation:
  - This is not the best way, but we need more Algebra

```
def test_generator(gen, prime):
    return len({ (gen**i)%prime for i in
                   range(prime-1) }) == prime-1
```

Karp Rabin

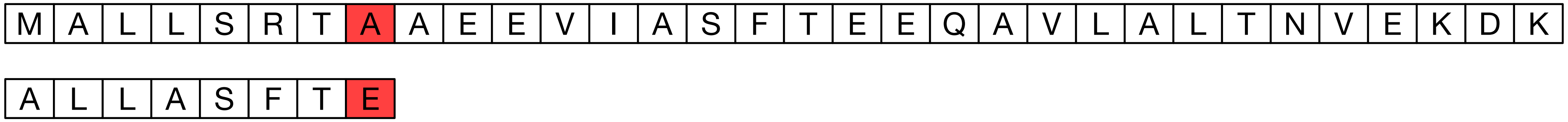
- Complexity:
  - $m =$ len(pattern), $n =$ len(text)
  - Still looks at every possible position $n - m + 1$
  - Replace m comparisons with:
    - One comparison, two additions, one multiplication with a constant (can be done with a table lookup)
  - Improvement from $\Theta(nm)$ to $\Theta(n + m)$ to find possible locations
    - But if the hash is bad, most possible locations still need to be verified

Boyer Moore Algorithm

- How can we do better?
  - Need to be able to slide the pattern further
  - But for this we need to foresee the text
  - That is why it is better to compare the pattern and the text from the right

Boyer Moore Algorithm

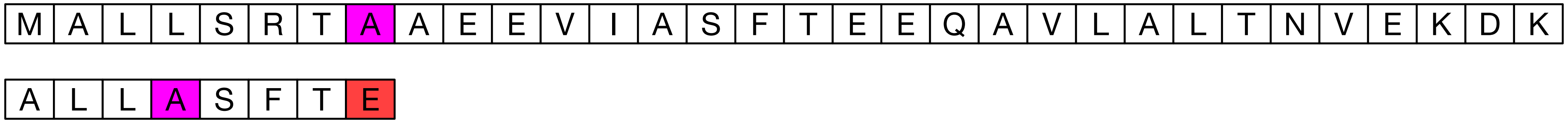
- Example:



M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K

A	L	L	A	S	F	T	E	

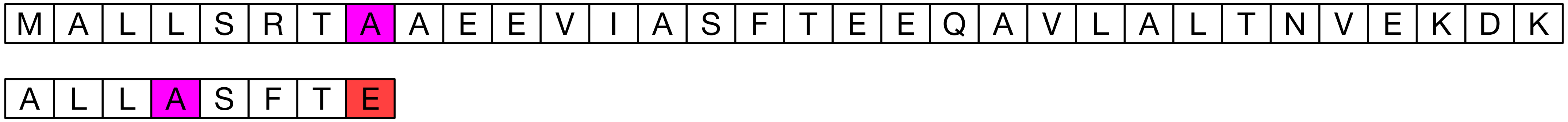
- Compare pattern from the right
- The 'A' in the text can at best be matched by the rightmost 'A' in the pattern



M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K

A	L	L	A	S	F	T	E

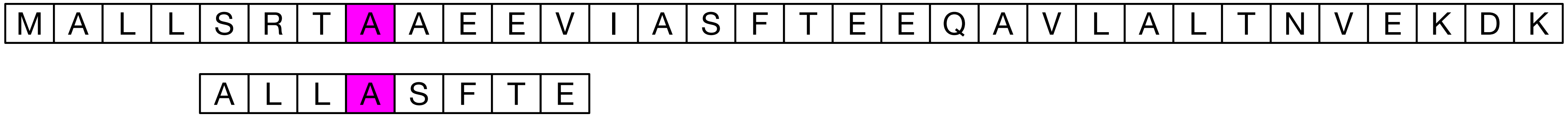
Boyer Moore Algorithm



M A L L S R T A A E E V I A S F T E E Q A V L A L T N V E K D K

A L L A S F T E

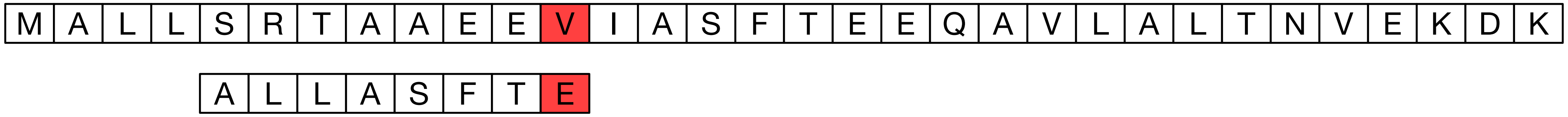
- So we slide four to the right



M A L L S R T A A E E V I A S F T E E Q A V L A L T N V E K D K

A L L A S F T E

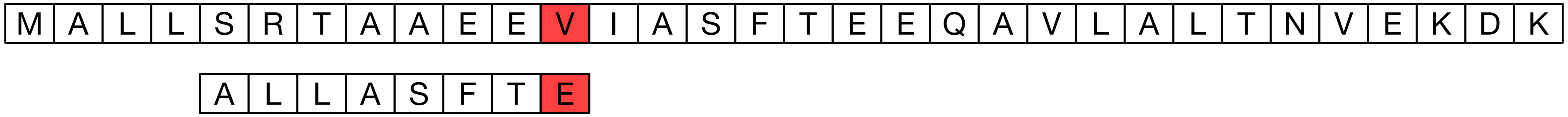
- And then compare at the new location



M A L L S R T A A E E V I A S F T E E Q A V L A L T N V E K D K

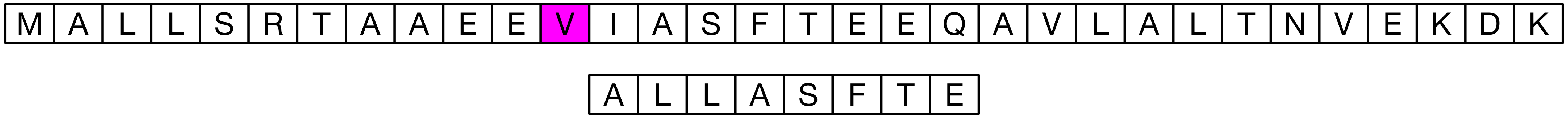
A L L A S F T E

Boyer Moore Algorithm



M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K
				A	L	L	A	S	F	T	E																				

- The 'V' does not appear in the string at all
- So we can slide by the length of the pattern



M	A	L	L	S	R	T	A	A	E	E	V	I	A	S	F	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K
												A	L	L	A	S	F	T	E												

Boyer Moore Algorithm

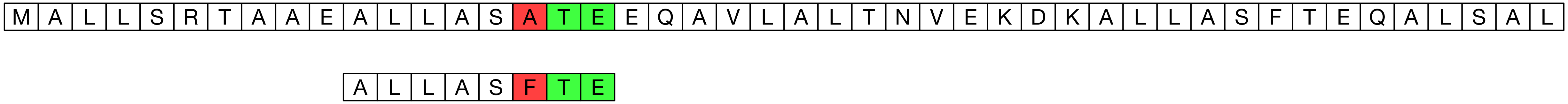
- To implement the "bad character" at the end, we need to process the string
  - Shift is the smallest distance of the bad character to the end in the pattern or the length of the pattern

A	L	L	A	S	F	T	E

A: 4
E: 0
F: 2
L: 5
S: 3
T: 1

Boyer Moore Algorithm

- We can use this also if we find a bad character after j successful comparisons



M A L L S R T A A E A L L A S A T E E Q A V L A L T N V E K D K A L L A S F T E Q A L S A L

A L L A S F T E

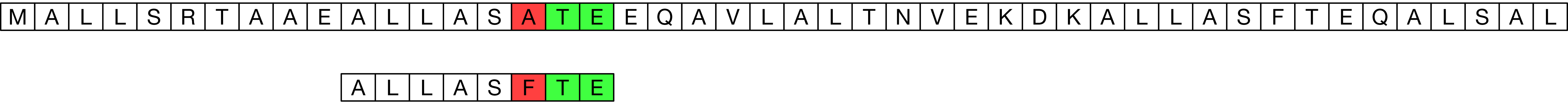
- We can shift by 5-3
- In general: table[char]-j

A L L A S F T E

A: 4
E: 0
F: 2
L: 5
S: 3
T: 1

Boyer Moore Algorithm

- This is not the only knowledge that we can use
  - Assume we have already matched part of the pattern, but now have a disagreement
  - This means that we know a part of the text



M	A	L	L	S	R	T	A	A	E	A	L	L	A	S	A	T	E	E	Q	A	V	L	A	L	T	N	V	E	K	D	K	A	L	L	A	S	F	T	E	Q	A	L	S	A	L

A	L	L	A	S	F	T	E

- Where can 'ATE' be matched in the pattern?
- Answer: not at all

Boyer Moore Algorithm

- We **preprocess** the pattern
  - For each letter we find the minimum distance to the end
  - For each suffix, we find the minimum distance of another copy of the suffix to the end
  - Or: If the alphabet is small:
    - Where can the suffix preceded by a single letter be found

Boyer Moore Algorithm

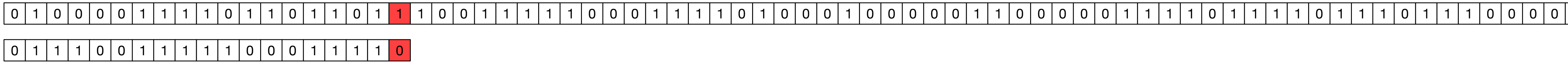
- Example:
  - pattern: 011001001
    - match "1": where can "11" be found: distance 6
    - match "01" where can "101" be found:
      - **01**1001001: shift 7
    - match "001": where is "0001":
      - **01**1001001: shift 7
    - match "01001": where is "101001" :
      - **01**1001001: shift 7

Boyer Moore Algorithm

- Both rules give usually different safe shift amounts
  - Always use the larger one

Boyer Moore Algorithm

- Example



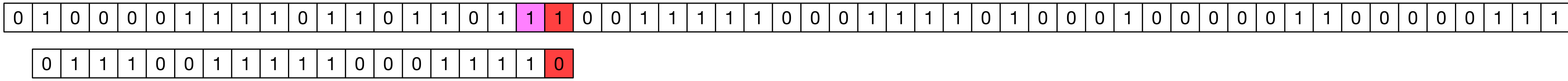
0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1 0 0 0 0

0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0

- Bad letter: shift 1
- Good suffix "": shift 1

Boyer Moore Algorithm

- Example:



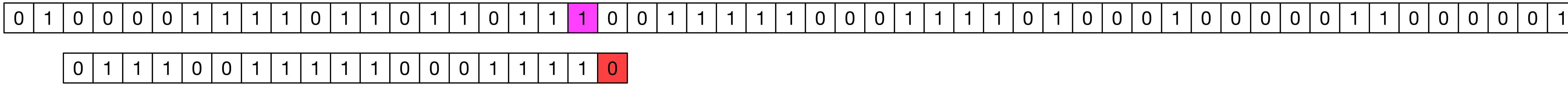
0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1

0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0

  - Bad character: shift 1

Boyer Moore Algorithm

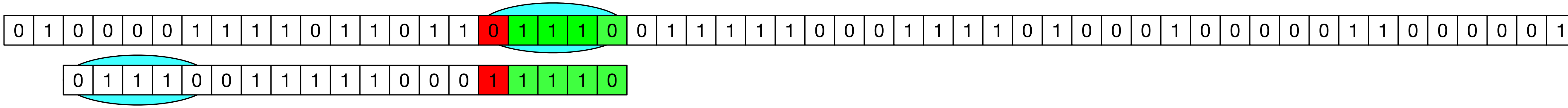
- Example



0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1

0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0

- Compare:



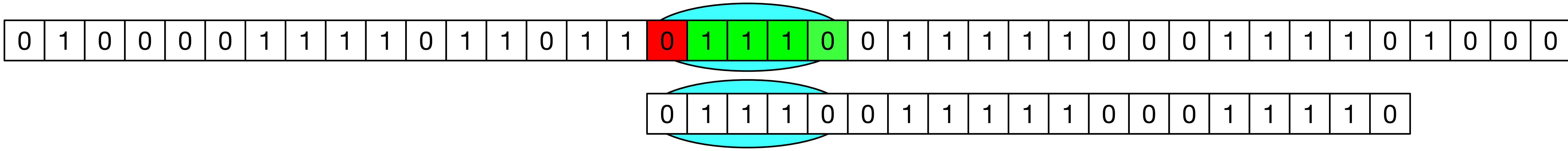
0 1 0 0 0 0 1 1 1 1 0 1 1 0 1 1 0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 0 1

0 1 1 1 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0

  - Bad character rule: shift by one
  - Good suffix rule: shift by 14

Boyer Moore Algorithm

- Example



0	1	0	0	0	0	1	1	1	1	0	1	1	0	1	1	0	1	1	1	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0	1	0	0	0
																0	1	1	1	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0				

  - After shift, we find a match
  - Then we shift by one

Boyer Moore Algorithm

- Your turn:
  - Preprocess "AGGTAA"
  - Bad character table
  - Bad suffix table

```
CA*****
 AGGTAA
```

```
A: 0
C: 6
G: 3
T: 2
```

```
CA: 5
GA: 5
TA: 1
AAA: 5
CAA: 5
GAA: 5
ATAA: 5
CTAA: 5
TTAA: 5
```

Boyer Moore

- Analysis is very difficult
    - Worst case:
        - Pattern and text consists of a single letter
            - $\sim n$ comparisons
    - Best case:
        - Pattern and text have completely different letters
        - $\lfloor \frac{n}{m} \rfloor$ comparisons

Boyer Moore

- Analysis is very difficult
  - Speed-up usually substantial
  - Called a "sub-linear" algorithm

Variants:

- Only the bad character rule:
  - Boyer-Moore-Horspool:
    - Only bad character rule
  - Apostolico-Giancarlo
    - Uses the pattern preprocessing in order to not compare letters that are known to be good
- Instead of a single bad character:
  - Use pairs of characters

Evaluation

- Algorithm comparison depends on the model
- Experimental evaluation:
  - Define and find "typical scenarios"
  - Use statistics to compare results