# Algorithms and Data Structures - Lesson 3 -

Michael Schwarzkopf
https://www.uni-weimar.de/de/medien/professuren/medieninformatik/grafische-datenverarbeitung

Bauhaus-University Weimar

May 30, 2018

# Overview

...of things you should definetely know about if you want a very good grade

- Hashing
- String Searching
  - Naive approach
  - Knuth-Morris-Pratt
  - Boyer-Moore
  - Karp-Rabin
- Algorithms on Graphs
- Devide and Conquer
- ...

# Hashing

Example: Imagine an array of Circle Objects.

```
class Circle{

  int positionX;
  int positionY;
  int radius
}

...

  Circles[0].radius = 3;
  Circles[1].radius = 4;
  Circles[2].radius = 1;
...
  Circles[46].radius = 8;
...
  Circles[99].radius = 5;
  Circles[100].radius = 10;
...
```

- Access trough indices 1 to 100
- How do I find the Circle with radius 8? (e.g. to get its position)

# Hashing

- Sort + Binary search: $O(n \log n) + O(\log n)$

- Iterating through the Array: $O(n)$

- Hashing: $O(1)$ Time
  $O(n)$ Space

# Hashing

- We need a Function H like: H( value ) = Index
  Its the **Hash Funktion.** For example:

  H( 8 ) = 20

- Now we look at our **Hash Table**, where the Data is stored:

| Hash Value | Data |
|---|---|
| 0 | empty |
| 1 | empty |
| 2 | Circle[ 28 ] |
| … | … |
| 20 | Circle[ 46 ] |
| ... | ... |

# Hash Function

- Simplest example: Digit sum („Quersumme“)

$$H(34) = 3 + 4 = 7$$

- Two Observations:
  - There is no $H^{-1}$ cause how do we get 34 from 7 again?
  - $H(34) = H(16) = H(241) = \ldots$

    → So called „**collision**“
- Limited table size: use modulo operation.

# Hashing: Collision

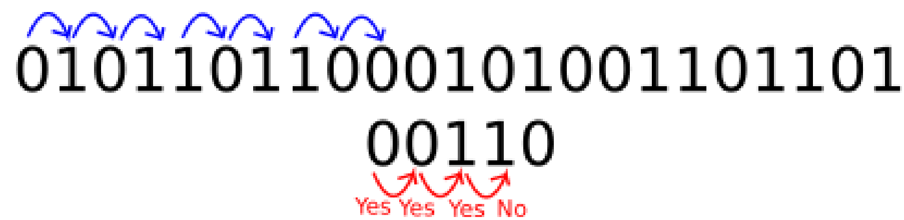
- Collision: $H( a ) = H ( b )$ ;  $a \neq b$
- Solutions:
  - <u>Open Adressing</u>: put a at  H(a)  and b at  H( b)+1  if this cell is empty. (If not, look further)
  - <u>Bucketing</u>: Each cell can hold more than one Element so Collisions are no problem.
  - <u>Double Hashing</u>: Take a different Hash Function H‘ and store b at H‘( H ( b ) )
- Worst case of Hashing: every Object collides → O(n)

# String Searching

- Simplest idea is least efficient again:

```
Input: Text, Pattern

for i =  0 ... |Text| - |Pattern|:
    for j =  0 ... |Pattern|
        if Pattern[j] != Text[i+j]
         break
        Output: "Found Match at " i
```




- Convention: |Text| = n, |Pattern| = m
- i times we check for j letters in Pattern:
  → O(n * m) in worst case.
- Let's talk about faster ones now!

# Knuth – Morris – Pratt

- First Step: Preprocessing on the Pattern:

  Check for each symbol, if it is (part of) a Prefix

| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- First letter is always 0
- b & c are no prefixes
- a is, so it gets 1
- ab is too, so a gets 1 and b gets 2 as it‘s the second letter in the prefix.

# Knuth – Morris – Pratt

- Second Step: Compare Pattern with String, if there is a missmatch at j, look at the Table at Position j – 1.

| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Take symbol at f(j - 1) and align it with the place you just checked
- You never compare a letter in the Text more than twice!

# Knuth – Morris – Pratt

aabcacbbabcaabcabcbacba

abcaab

| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Missmatch at Pos. j = 1, so look for f( j – 1 ) = f( 0 ) = 0 and align this Position the Letter you just checked.

# Knuth – Morris – Pratt

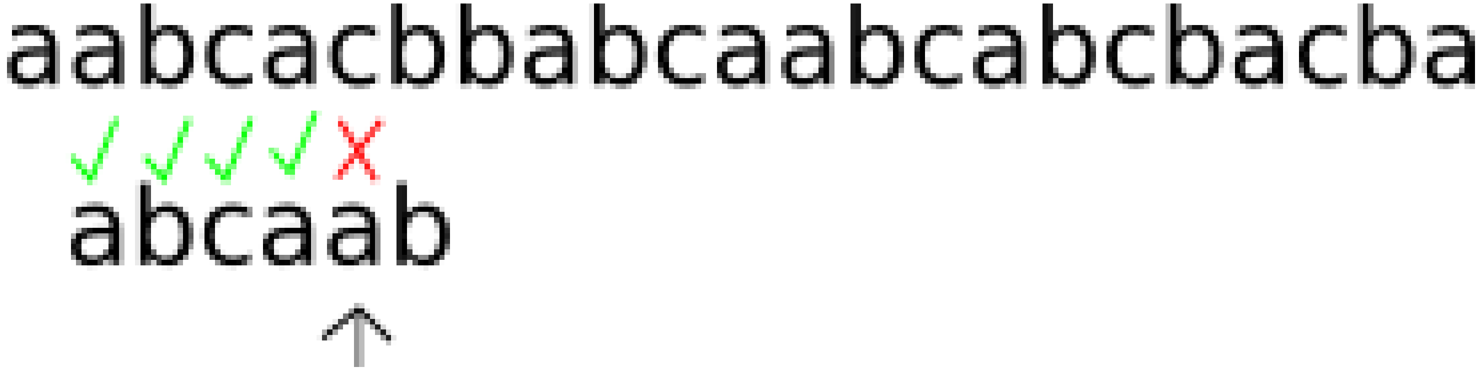
aabcacbbabcaabcabcbacba

abcaab

| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Compare again at j = 0 and continue to the right until you find a Missmatch

# Knuth – Morris – Pratt




| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Missmatch at j = 4. f( 3 ) = 1, so move the letter at j = 1 which is ‚b‘ above the coursor.

# Knuth – Morris – Pratt

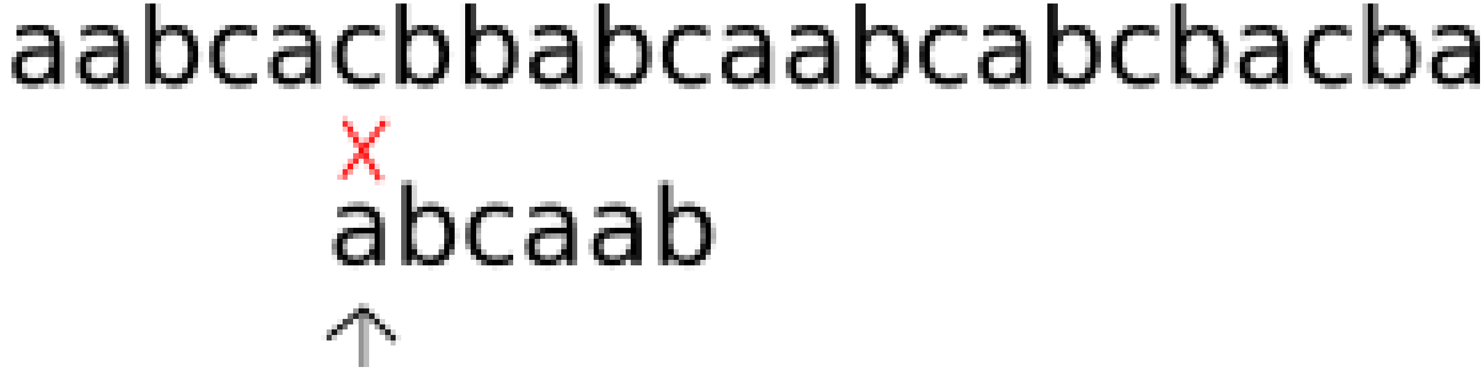
aabcacbbabcaabcabcbacba

abcaab

| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Because of magic of our prefix – table, we do not have to move the cursor left to know that the letters at the left hand side match.

# Knuth – Morris – Pratt




| j | 0 1 2 3 4 5 |
|---|---|
| symbol | a b c a a b |
| f( j ) | 0 0 0 1 1 2 |

- Missmatch at first letter, move 1 to the right…

# Knuth – Morris – Pratt

aabcacbbabcaabcabcbacba

abcaab

nope.

aabcacbbabcaabcabcbacba

abcaab

nope…

aabcacbbabcaabcabcbacba

abcaab

got it!

# KMP - Complexity

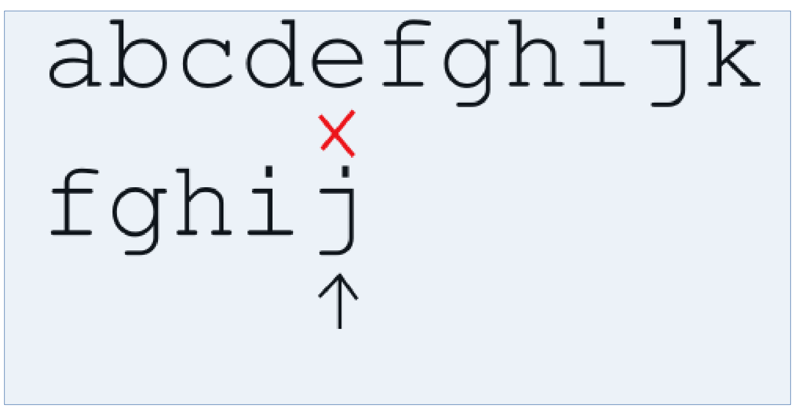
- Table: Check every letter in pattern: O(m)
- Searching: Each Index not more than twice
  $\rightarrow$ 2n steps $\rightarrow$ O(n)

  <u>$\rightarrow$ O(n + m)</u>

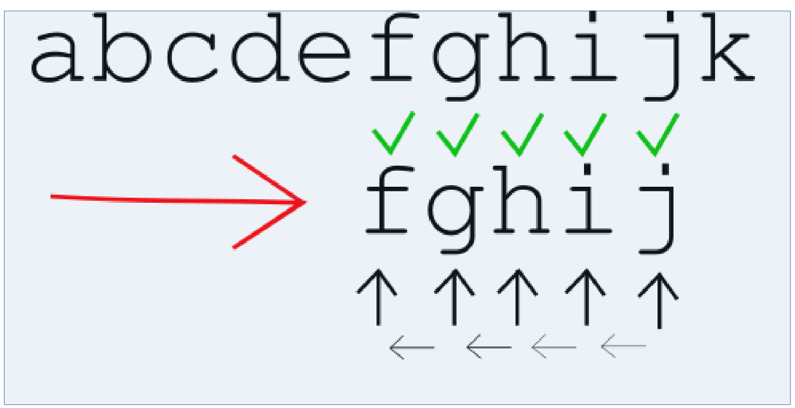
- Easy
- Works best with repeating patterns, as you can shift a lot to the right then.

# Boyer - Moore

- Align pattern with text as always, BUT
- Check Pattern from right to left!
  → In case of missmatch you can shift the pattern quite a lot.




nope.




yope! :)

- Only Question: <u>How far shall we shift?</u>

# Boyer – Moore: Strategy 1

- „Bad – Character – Rule“

  If missmatch happens: Check if missmatched Letter in Text is existing again in Pattern
  → align the first one left from the missmatch:

```
abdcabdcbacabdccdcd
cbaca
   cbaca
```

- Missmatch at „d“: No „d“ in pattern: → shift whole thing beyond missmatch.

# Boyer – Moore: Strategy 1

- „Bad – Character – Rule“

  If missmatch happens: Check if missmatched Letter in <u>Text</u> is existing again in <u>Pattern</u>
  → align the first one left from the missmatch:

```
abdcabdcbacabdccdcd
      cbaca
       cbaca
```

- Missmatch at „c“
  → align with „c“

# Boyer – Moore: Strategy 1

- „Bad – Character – Rule“

  If missmatch happens: Check if missmatched Letter in <u>Text</u> is existing again in <u>Pattern</u>
  → align the first one left from the missmatch:

```
abdcabdcbacabdccdcd
      cbaca
         cbaca
```

- Missmatch at „b“
  → align with „b“

# Boyer – Moore: Strategy 1

- „Bad – Character – Rule“

  If missmatch happens: Check if missmatched Letter in Text is existing again in Pattern
  → align the first one left from the missmatch:

```
abdcabdcbacabdccdcd
       ✓✓✓✓✓
       cbaca
```

- Got it!

# Boyer – Moore: Strategy 2

- „Good – Suffix – Rule“ (Suffix: Ending of a Word)

  If missmatch happens: Check if a suffix of matched letters is substring of pattern.
  If it is: align it!

```
abdcabacbadbadbacd

adbadba

    adbadba
```

- „ba“ matches.
  „ba“ is also part of the Pattern
  → align!

# Boyer – Moore: Strategy 2

- „Good – Suffix – Rule“ (Suffix: Ending of a Word)

  If missmatch happens: Check if a suffix of matched letters is substring of pattern.
  If it is: align it!

```
abdcabacbadbadbacd
      adbadba
         adbadba
```

- Same thing again

# Boyer – Moore: Strategy 2

- „Good – Suffix – Rule“ (Suffix: Ending of a Word)

  If missmatch happens: Check if a suffix of matched letters is substring of pattern.
  If it is: align it!

```
abdcabacbadbadbacd
        x√√√√√
        adbadba
           adbadba
```

- „badba“ isn‘t again in Pattern though, but it‘s suffix „adba“ is.

# Boyer – Moore: Strategy 2

- „Good – Suffix – Rule“ (Suffix: Ending of a Word)

  If missmatch happens: Check if a suffix of matched letters is substring of pattern.
  If it is: align it!

```
abdcabacbadbadbacd
        ✓✓✓✓✓✓✓
        adbadba
```

- There it is!

# Boyer – Moore

- Depending on strategy, you shift more or less.
  In each step: Choose strategy which shifts more to the right!
- Works best in big alphabets;
  Imagine this Algorithm in Binary String
  → barely shifts more than 2 positions.
- One more good example:
  https://youtu.be/4Xyhb72LCX4 - ADS1: Boyer-Moore basics
  https://youtu.be/Wj606N0IAsw - ADS1: Boyer-Moore: putting it all together

# BM - Complexity

- Figuring out suffixes or reappearing letters works best with preprocessing as in KMP.
  $\rightarrow O(m)$
- In <u>worst case</u> the leftmost character in Pattern missmatches and we have to check every letter in Text: $\rightarrow O(n)$
- Worst case: $O(n+m)$
- <u>Average & best:</u> we skip m Letters each step in a Text of length n:
  $\rightarrow O(n/m)$

# Karp – Rabin (Assignment)

- Idea:

Pattern: c b a

A Hash function: H

H(Pattern) = H (c b a) = 13

Text: a b c c b a a b c

H(a b c) = 8
H(b c c) = 2
H(c c b) = 19
H(c b a) = 13
H(b a a) = 21
H(a a b) = 5
H(a b c) = 8

- Hash the Pattern
- Hash every substring of the text which has the length of the pattern
- Compare Hashvalues
  here: Search for the 13
- If match: check for every letter

# Karp – Rabin: Hashfunction

- First: represent symbols as Integers. (→ ascii)
- Text: ( a[0], a[1], … ,a[n-1] )
- Pattern: ( b[0], b[1], … ,b[m-1] )
- d: number of different symbols we use (Base)
- p: quite big prime number

- Let $k[i] = (a[\,i\,] * d^{\,m-1}) + (a[\,i+1\,] * d^{\,m-2}) + \dots$
  $\dots + (a[\,i+m-2\,] * d^{1}) + (\,a[\,i+m-1\,]\,)$
  → Maps a Substring on a number k

*Notation is a bit different from the Lecture

# Karp – Rabin: Hashfunction

- Second: do H( k[i] ) = k[i] mod p

  → Until here we needed about m steps

- To calculate next Hash however, just:
  H( k[i +1] ) = ( (k[i] - a[i])*d + a[i+m] ) mod p

  → Constant time for every Substring :)

- Don't panic: only <u>looks</u> cryptic

# Karp – Rabin: Hashfunction

- **Example:**
  102321312 → Base is 4. ( {0,1,2,3} => 4 digits)
- Calculate k for Substring „213“:

  $k = 2 * 4^2 + 1 * 4^1 + 3 * 4^0$
  $k = 32 \quad + 4 \quad + 3$
  $\underline{k = 39}$
- Let p = 37 (Usually bigger, just for example)
- So H( k ) = 39 mod 37 = 2

# Karp – Rabin: Hashfunction

- **Example:**
  10232$\underline{131}$2 → Base is 4. ( {0,1,2,3} => 4 digits)
- Calculate k for Substring „131“:
- $k[i+1] = (39 - 2)*4 + 1$

  $k[i] \quad a[i] \quad d \quad a[i+m]$
- $k[i+1] = 149$
- $H(149) = 149 \bmod 37 = 1$

# Karp – Rabin: Hashfunction

- If you like to try yourself:

$\underline{102}321312 \rightarrow 18$
$1\underline{023}21312 \rightarrow 11$
$10\underline{232}1312 \rightarrow 9$
$102\underline{321}312 \rightarrow 20$
$1023\underline{213}12 \rightarrow 2$
$10232\underline{131}2 \rightarrow 1$
$102321\underline{312} \rightarrow 17$

- Let‘s search for 322:
  k = 58
  H(k) = 21
  21 is not in the List.

- O(n) Runtime for excluding

# Karp – Rabin: Hashfunction

- If you like to try yourself:

$\underline{102}321312 \rightarrow 18$

$1\underline{023}21312 \rightarrow 11$

$10\underline{232}1312 \rightarrow 9$

$102\underline{321}312 \rightarrow 20$

$1023\underline{213}12 \rightarrow 2$

$10232\underline{131}2 \rightarrow 1$

$102321\underline{312} \rightarrow 17$

- Let‘s search for 232:
  k = 46
  H(k) = 9
  9 is in the List!
- Compare:
  232 matches 232
- O(n + m) Runtime for finding

# KR: Complexity

- Hashing substrings: O(n)
- Hasing Pattern: O(m)
- Searching: O(n)

  → Average/ Best Case: <u>O(n+m)</u>
- Very unlikely: Most hashes cause collision:
  → Worst Case: <u>O(m*n)</u>
  (Gets even less likely if you choose a bigger prime number p)

# In Case of Exam

- Hashing
  - How does it work?
  - What is it for?
  - Problems: Collision (What‘s this?)
    → Open Adressing, Bucketing, Double Hashing
- String Searching:
  For each of the 4 algorithms shown:
  - Explain in detail
  - Discuss complexity

# Assignment

- Don‘t forget the conditions! (next Slide)
- Karp – Rabin might be not the most efficient string search algorithm, but it gives a good example about hashing and string searching!

Write a program, which takes a text and a pattern as <u>string</u>, uses Karp – Rabin to figure out if, where and how often the pattern exists in the text.

Good luck!

# Assignment Conditions

- Code comes from nowhere else than your brain!
- .java / .c / .cpp → <u>NO</u> .docx .pdf etc!!
- Good comments make the difference between „alright“ and „very good“!
- Put Matriculation number as comment above
- Deadline: 12 June 2018, 23:59
- Mail: michael.schwarzkopf@uni-weimar.de