# Lecture 9

## Algorithms & Data Structures

Goldsmiths Computing

December 3, 2018

# Outline

# Outline

# Lecture

- Hash tables
  - collision resolution
  - deletion: tombstones vs backward-shift
  - Robin Hood hashing
- Characters
  - symbols, graphemes, grapheme clusters
  - code points
- Strings
  - ordered collections of code points
  - new operation: matching

# Lab

- Be a data structure implementor
    1. Hash tables!
        - basics: insert, find
        - loadFactor
        - delete, extend/rehash

# VLE activities

## Recursive algorithms quiz

Statistics so far:

- 135 attempts: average mark 5.57
- 74 students: average mark 5.67
  - 26 under 4.00, 32 over 6.99, 15 at 10.00

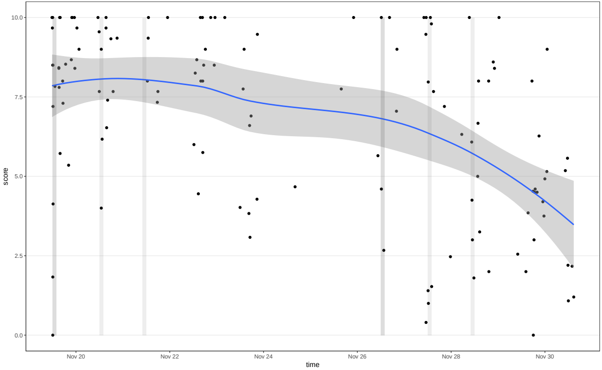
Quiz closes at 16:00 on Friday 7th December

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

# VLE activities (cont'd)

## Recurrence relations quiz

- 575 attempts: average mark 4.82
- 130 students: average mark 6.78
    - 24 under 4.00, 75 above 6.99, 24 at 10



# VLE activities (cont'd)

## List visualiser

- 127 submissions
- assessment phase:
  - assess according to questions (use my provided driver program!)
  - differences of interpretation
  - assume good intent
  - read submitted code
  - give written feedback!

Assessment phase closes at 16:00 on Friday 7th December

- no extensions
- grade is
  - 0 (for incomplete assessments)
  - 30 + score

# VLE activities (cont'd)

## First term questionnaire

non-anonymous survey (for my benefit):

- what's gone well;
- what you've enjoyed;
- what has most helped you learn

https://learn.gold.ac.uk/mod/feedback/view.php?id=613718

# Outline

# Motivation

- generalisation of search operation (sequences, not just single elements)
- applications include text editors, classifiers, information retrieval systems
- extensions used in
  - spelling checkers
  - DNA sequence matching
  - protein structure representations

# Definition

String matching returns the smallest index at which the *pattern*, P, is found exactly in the *text*, T, or false if the pattern is not present in the text at all.

C++ `std::string::find()`

Java `java.lang.String.indexOf()`

# String matching algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        if T[s...s+m] = P[0...m] then
            return s
        end if
    end for
    return false
end function
```

# Naïve algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        found ← true
        for 0 ≤ j < m do
            if T[s+j] ≠ P[j] then
                found ← false; break
            end if
        end for
        if found then
            return s
        end if
    end for
    return false
end function
```

# Diagram

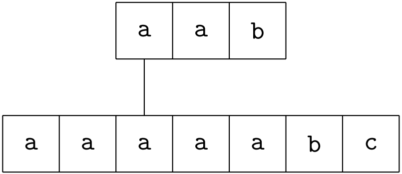
| | | |
|---|---|---|
| a | a | b |

# Diagram

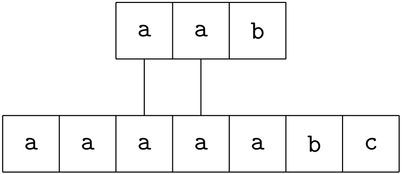
| | | |
|---|---|---|
| a | a | b |

| | | | | | | |
|---|---|---|---|---|---|---|
| a | a | a | a | a | b | c |

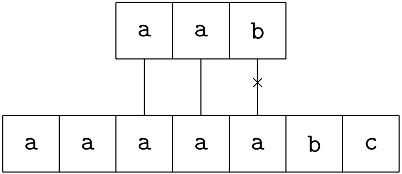
# Diagram



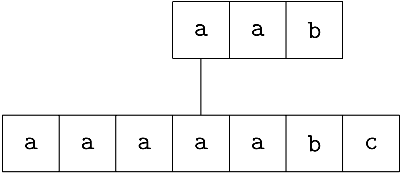
# Diagram



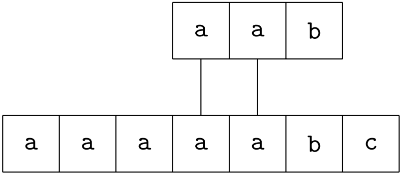
# Diagram



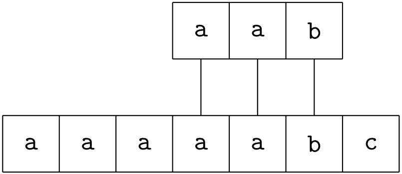
# Diagram



# Diagram



# Diagram



# Complexity analysis

## space

- no particular requirements for additional storage

$$\Longrightarrow \Theta(1)$$

## time

- outer loop happens $n - m + 1$ times (worst case)
- inner loop $m$ times (worst case)

$$\Longrightarrow \Theta((n+1)m - m^2) \sim \Theta(nm)$$

For particular sizes of pattern:

small $m \sim c \Longrightarrow \Theta(n)$

large $m \sim n \Longrightarrow \Theta(n)$

intermediate $m \sim \frac{n}{2} \Longrightarrow \Theta(n^2)$

# Work

1. Reading
   - CLRS, section 32.1
   - Drozdek, section 13.1.1 “Straightforward Algorithms”
2. Questions from CLRS

   **Exercises** 32.1-1, 32.1-2
3. Lab work
   - (week of 3rd December) implement naïve string match for strings of characters. Use `OpCounter` (remember that?) to count how many character comparisons happen in the worst case. Construct a table and verify the theoretical results in this lecture.

# Outline

# Motivation

- naïve string matching takes time in $\Theta(mn)$
- lots of wasted work

# Naïve algorithm

```
function MATCH(T,P)
    m ← LENGTH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        found ← true
        for 0 ≤ j < m do
            if T[s+j] ≠ P[j] then
                found ← false; break
            end if
        end for
        if found then
            return s
        end if
    end for
    return false
end function
```

# Less work in the inner loop

- avoid $\Theta(m)$ comparisons where possible
- constant-time test:
  - hash value comparison

# Rabin-Karp algorithm

```
function RKMATCH(T,P)
    m ← LENGTH(P); hm ← HASH(P)
    for 0 ≤ s ≤ LENGTH(T) - m do
        if HASH(T[s...s+m]) = hm then
            found ← true
            for 0 ≤ j < m do
                if T[s+j] ≠ P[j] then
                    found ← false; break
                end if
            end for
            if found then
                return s
            end if
        end if
    end for
    return false
end function
```

# Hash function

Normally:

- HASH(T[s...s+m]) takes time in $\Theta(m)$
- no saved work in general

# Rolling hash

Clever choice of hash function makes a difference!

- ROLLING-HASH(h,T[s-1],T[s+m])

Examples of suitable hash functions

modular add $\sum_i x_i \bmod k$

exclusive or $\oplus_i x_i$

modular polynomial $\sum_i x_i p^i \bmod k$

# Modular add

$$\sum_i x_i \bmod k$$

- 21-bit characters: $k$ might be $2^{24}$ or $2^{32}$
  - (resist temptation to use 8-bit characters and $k$ of $2^8$)

**function** ROLLING-HASH(prev,remove,add)
    **return** (prev - remove + add) mod k
**end function**

- extremely limited bit mixing
- high chance of hash collisions in typical texts
  - *e.g.* HASH(ab) = HASH(ba)

# Exclusive or

$$\oplus_i x_i$$

- no parameters
  - (still need to resist temptation to use 8-bit characters)

**function** ROLLING-HASH(prev,remove,add)
    **return** prev $\oplus$ remove $\oplus$ add
**end function**

- no bit mixing at all
- high chance of hash collisions in typical texts
  - *e.g.* HASH(`oboe`) = HASH(`bell`)

# Modular polynomial

$$\sum_i x_i p^i \bmod k$$

- typically choose a small(ish) prime $p$
- use machine word (*e.g.* $2^{32}$) for $k$

**function** ROLLING-HASH(prev,remove,add)
  **return** $\left((\text{prev} - \text{remove} \times p^{\mathbf{m-1}}) \times p + \text{add}\right) \bmod k$
**end function**

- good mixing (*e.g.* for prime $p = 101$, character bits 0-7 affect hash bits 0-13)
- hash collisions in typical texts rarer

# Complexity analysis

## space

no need for extra space that scales with any parameter

$$\Longrightarrow \Theta(1)$$

## time

- for good rolling hash:
  - new hash computation from old hash in $\Theta(1)$ time
  - hash collisions rare (still need to do at least two $\Theta(m)$ hash computations)

$$\Longrightarrow \Theta(n) + \Theta(m) \text{ (average case)}$$

- even for the best hash function...
  - ...suitably adversarial input will collide a lot

$$\Longrightarrow \Theta(nm) \text{ (worst case)}$$

# Work

1. Reading
   - CLRS, section 32.2
2. Questions from CLRS
   - Exercise 32.2-2
3. Lab work
   - (week of 3rd December) implement Rabin-Karp string match for strings of characters. Use `OpCounter` to count how many character comparisons happen in the best and worst case. Construct a table and verify the theoretical results in this lecture.

# Outline

# Motivation

- deterministically $\Theta(m + n)$ string matching

# Definition

Knuth-Morris–Pratt matching uses information about the pattern P to avoid redundant work when doing string matching.

# Example

Consider MATCH(abcde, text)

- all characters in P different
- mismatch in index position $k$
  - matches in all previous positions [0,k)
  - can safely advance next start position to $k$.

# Diagram

- all pattern characters different:

| | | | | |
|---|---|---|---|---|
| a | b | c | d | e |

# Diagram

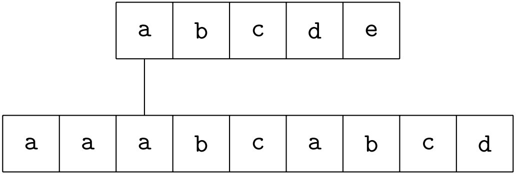
- all pattern characters different:

| | | | | |
|---|---|---|---|---|
| a | b | c | d | e |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | c | a | b | c | d |

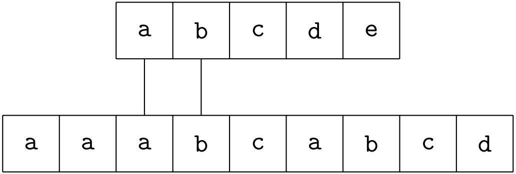
# Diagram

- all pattern characters different:



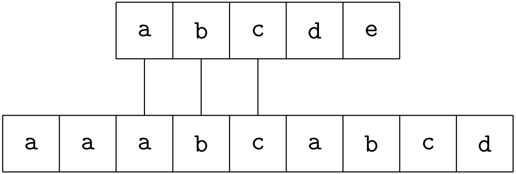
# Diagram

- all pattern characters different:



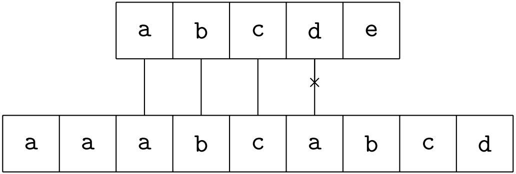
# Diagram

- all pattern characters different:



# Diagram

- all pattern characters different:



# Diagram

- all pattern characters different:

| | | | | |
|---|---|---|---|---|
| a | b | c | d | e |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | c | a | b | c | d |

# Diagram

- pattern contains similar suffixes:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

# Diagram

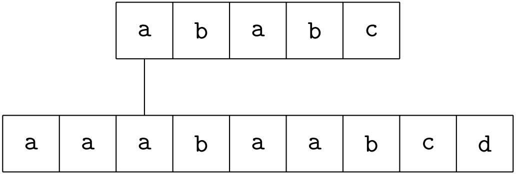
- pattern contains similar suffixes:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | a | a | b | c | d |

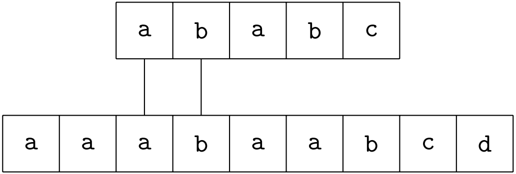
# Diagram

- pattern contains similar suffixes:



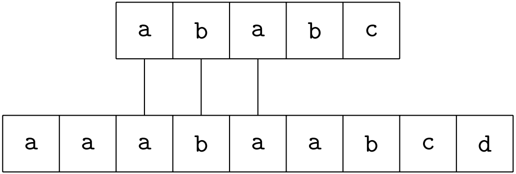
# Diagram

- pattern contains similar suffixes:



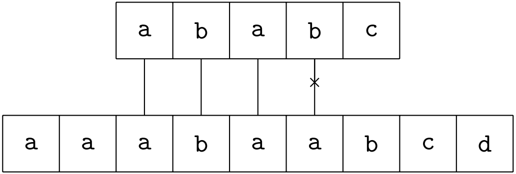
# Diagram

- pattern contains similar suffixes:



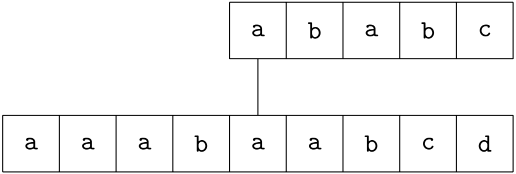
# Diagram

- pattern contains similar suffixes:



# Diagram

- pattern contains similar suffixes:



# Prefix table

Also called “prefix function” or “failure function”

- encode for each index $k$ the length of the longest prefix of the pattern P which is a suffix of the subsequence of the pattern P[0..k]

| | | | | |
|---|---|---|---|---|
| a | b | c | d | e |
| 0 | 0 | 0 | 0 | 0 |

# Prefix table

Also called "prefix function" or "failure function"

- encode for each index *k* the length of the longest prefix of the pattern P which is a suffix of the subsequence of the pattern P[0..k]

| | | | | |
|---|---|---|---|---|
| a | b | c | d | e |
| 0 | 0 | 0 | 0 | 0 |

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |
| 0 | 0 | 1 | 2 | 0 |

# Knuth-Morris-Pratt algorithm

```
function KMPMATCH(T,P)
    n ← LENGTH(T); m ← LENGTH(P)
    π ← COMPUTEPREFIX(P)
    q ← 0
    for 0 ≤ i < n do
        while q > 0 ∧ P[q] ≠ T[i] do
            q ← π[q-1]
        end while
        if P[q] = T[i] then
            q ← q + 1
        end if
        if q = m then
            return i - m + 1
        end if
    end for
    return false
end function
```

# Knuth-Morris-Pratt algorithm: compute prefix

```
function COMPUTEPREFIX(P)
    m ← LENGTH(P)
    π ← new Array(m); π[0] ← 0
    k ← 0
    for 1 ≤ q < m do
        while k > 0 ∧ P[k] ≠ P[q] do
            k ← π[k-1]
        end while
        if P[k] = P[q] then
            k ← k + 1
        end if
        π[q] ← k
    end for
    return π
end function
```

# Work

1. Reading
   - CLRS, section 32.4
   - Drozdek, section 13.1.2 “The Knuth-Morris-Pratt Algorithm”
     - NB: `next` table in Drozdek is very slightly different from result of COMPUTEPREFIX
2. Lab work
   - (week of 3rd December) implement Knuth-Morris-Pratt string match. Use `OpCounter` to count how many character comparisons happen in the best and worst cases, and verify the theoretical results in this lecture.

# Outline

# Motivation

- deterministically $\Theta(m + n)$ string matching
- can achieve $\Theta(n/m)$ for matching phase in the best case

# The bad character heuristic

- previously: use the *fact* that a mismatch has occurred to save work;
- now: use the specific character in the *text* that doesn't match (the "bad character") to save work.
  - check characters backwards from the end of the pattern for maximum effect

# Diagram

- bad character not in pattern:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

# Diagram

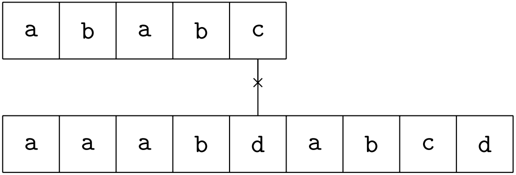
- bad character not in pattern:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | d | a | b | c | d |

# Diagram

- bad character not in pattern:



# Diagram

- bad character not in pattern:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | d | a | b | c | d |

# Diagram

- bad character in pattern:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

# Diagram

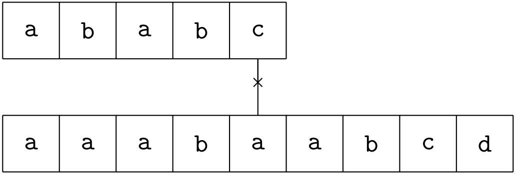
- bad character in pattern:

| | | | | |
|---|---|---|---|---|
| a | b | a | b | c |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | a | b | a | a | b | c | d |

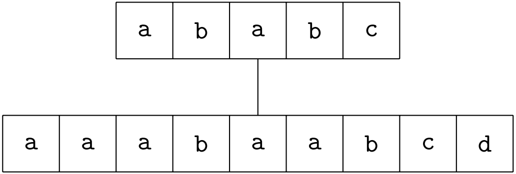
# Diagram

- bad character in pattern:



# Diagram

- bad character in pattern:



# Boyer-Moore-Horspool

```
function BMHMatch(T,P)
    n ← length(T); m ← length(P)
    λ ← computeBadCharacter(P)
    s ← 0
    while s ≤ n - m do
        j ← m - 1
        while j ≥ 0 ∧ P[j] = T[s+j] do
            j ← j - 1
        end while
        if j = -1 then
            return s
        else
            s ← s + max(1,j-λ[T[s+j]])
        end if
    end while
    return false
end function
```

# Boyer-Moore-Horspool: compute bad character

```
function computeBadCharacter(P)
    m ← length(P)
    λ ← new Table(-1)
    for 0 ≤ j < m do
        λ[P[j]] ← j
    end for
    return λ
end function
```

# Work

1. Reading
   - Drozdek, section 13.1.3 “The Boyer-Moore Algorithm”

# The good suffix heuristic

- bad character heuristic can recommend zero (or negative) shift
- not using information about any partial match
- good suffix: use knowledge that the suffix of the pattern matched must match any shifted pattern
  - find rightmost instance of good suffix...
  - ... not at the end of the pattern ...
  - (... preceded by a different character)

# Diagram

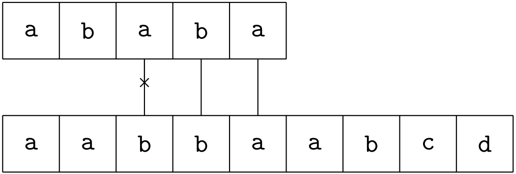
| | | | | |
|---|---|---|---|---|
| a | b | a | b | a |

# Diagram

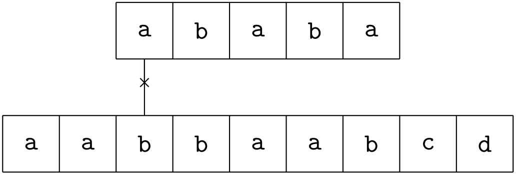
| | | | | |
|---|---|---|---|---|
| a | b | a | b | a |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | b | b | a | a | b | c | d |

# Diagram



# Diagram



# Diagram

| | | | | |
|---|---|---|---|---|
| a | b | a | b | a |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | a | b | b | a | a | b | c | d |

# Boyer-Moore

```
function BMMATCH(T,P)
    n ← LENGTH(T); m ← LENGTH(P)
    λ ← COMPUTEBADCHARACTER(P)
    γ ← COMPUTEGOODSUFFIX(P)
    s ← 0
    while s ≤ n - m do
        j ← m - 1
        while j ≥ 0 ∧ P[j] = T[s+j] do
            j ← j - 1
        end while
        if j = -1 then
            return s
        else
            s ← s + max(γ[j],j-λ[T[s+j]])
        end if
    end while
    return false
end function
```

# Boyer-Moore: compute good suffix

```
function computeGoodSuffix(P)
    m ← length(P); π ← computePrefix(P)
    P' ← reverse(P); π' ← computePrefix(P')
    γ ← new Array(m)
    for 0 ≤ j < m do
        γ[j] ← m - π[m-1]
    end for
    for 0 ≤ l < m do
        j ← m - π'[l] - 1
        if γ[j] > l + 1 - π'[l] then
            γ[j] ← l + 1 - π'[l]
        end if
    end for
    return γ
end function
```

# Galil Rule

If pattern is shifted to start at a text position after positions already checked:

- no need to recheck known-good matches

# Complexity Analysis

## space

$\gamma$, $\lambda$ each $\Theta(m)$

- $\lambda$ is $\Theta(\Sigma)$ if implemented using an array

## time

Boyer-Moore-Horspool and Boyer-Moore

- preprocessing: $\Theta(m)$
- match:
  - worst case $\Theta(mn)$
  - best case $\Theta(n/m)$

With Galil Rule:

- worst case $\Theta(m + n)$
- best case $\Theta(n/m)$