Lecture 18

Algorithms & Data Structures

Goldsmiths Computing

March 11, 2019

Outline

Outline

Lecture

1. Counting sort
   - better than $\Theta(N \log N)$
2. Topological sorting
   - linearization of DAGs
   - Kahn's algorithm
   - depth-first linearization
3. Quicksort and quickselect
   - median in $\Theta(N)$ worst-case
   - quicksort in $\Theta(N \log N)$ worst case

Lab

Big Integer implementation

VLE activities

More recursive algorithms quiz

Statistics so far:

- A attempts: average mark B
- C students: average mark D
  - E under 4.00, F over 6.99, G at 10.00

Quiz closes at 16:00 on Friday 15th March

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

VLE activities (cont'd)

Numbers quiz

VLE activities (cont'd)

Numbers quiz

VLE activities (cont'd)

Numbers quiz

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

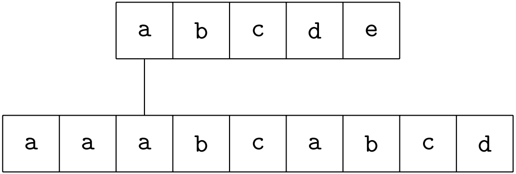
- all pattern characters different:

a	b	c	d	e

a	a	a	b	c	a	b	c	d

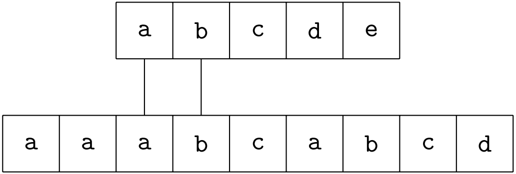
Diagram

- all pattern characters different:



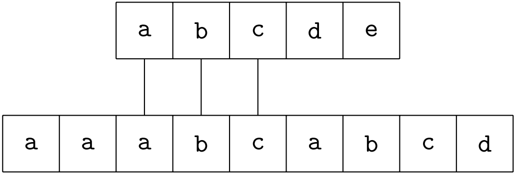
Diagram

- all pattern characters different:



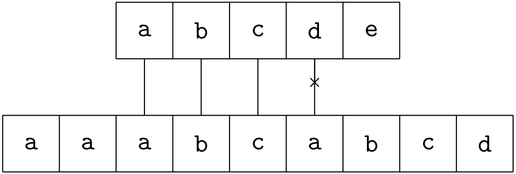
Diagram

- 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

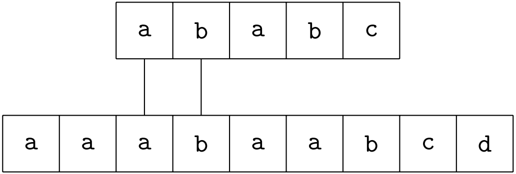
- pattern contains similar suffixes:

a	b	a	b	c

a	a	a	b	a	a	b	c	d

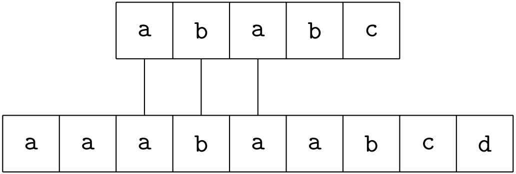
Diagram

- pattern contains similar suffixes:



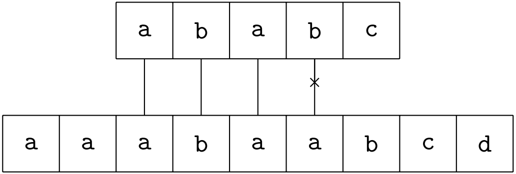
Diagram

- pattern contains similar suffixes:



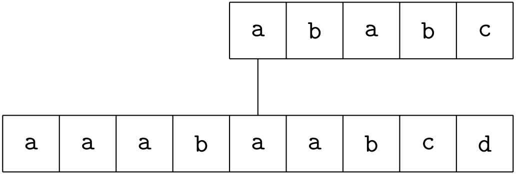
Diagram

- pattern contains similar suffixes:



Diagram

- pattern contains similar suffixes:



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 $1 \leq q < m$ **do**
    while $k > 0 \land P[k] \neq P[q]$ **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

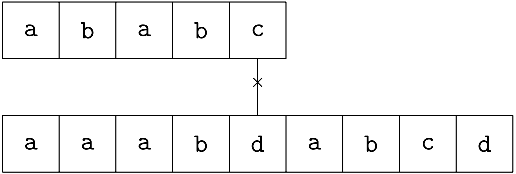
- bad character not in pattern:

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

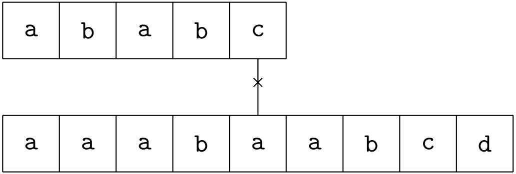
- bad character in pattern:

a	b	a	b	c

a	a	a	b	a	a	b	c	d

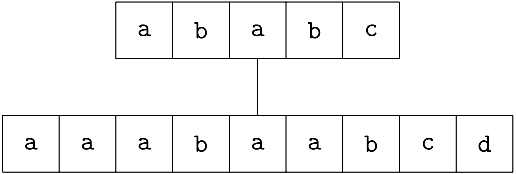
Diagram

- 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

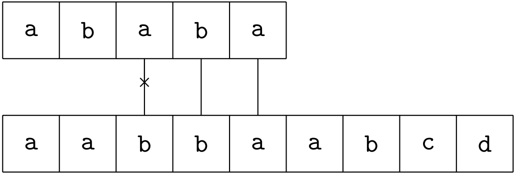
a	b	a	b	a

Diagram

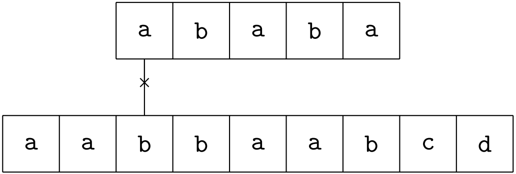
a	b	a	b	a

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

γ, λ each $\Theta(m)$

- λ 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 $\Theta(n/m)$

Outline

Motivation

A data structure

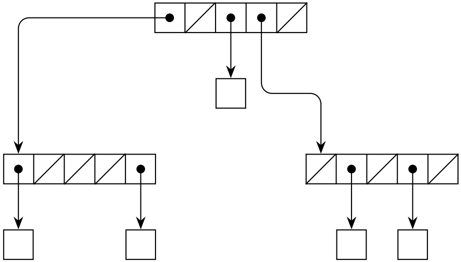
- to hold a set of strings
- to answer efficiently string prefix match
  - (including set membership)

Definition

A trie is a tree structure where each internal node has children labelled by characters from an alphabet. The trie represents the set of strings formed by concatenating labels of traversals from the root to a leaf.

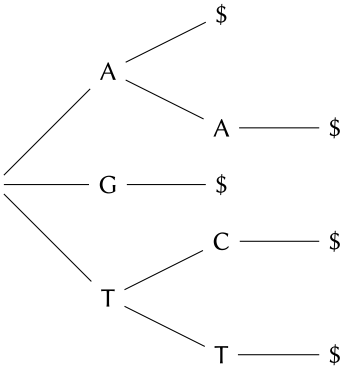
Implementation

- alphabet: `A`, `C`, `G`, `T`
- set of strings: `A`, `AA`, `G`, `TC`, `TT`



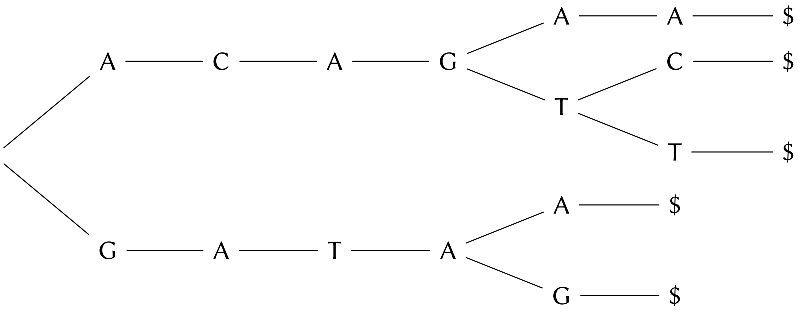
Implementation

- alphabet: `A`, `C`, `G`, `T`
- set of strings: `A`, `AA`, `G`, `TC`, `TT`



Implementation

- alphabet: `A`, `C`, `G`, `T`
- set of strings: `GATAA`, `ACAGAA`, `GATAG`, `ACAGTC`, `ACAGTT`



Algorithm

```
function PREFIX(T,P)
    if EMPTY?(P) then
        return true
    else if LEAF(T) then
        return false
    else if NULL?(T[P[0]]) then
        return false
    else
        return PREFIX(T[P[0]],P[1...])
    end if
end function
```

Algorithm

```
function MEMBER(T,P)
    if EMPTY?(P) then
        if INTERNAL(T) then
            return T[$]
        else
            return true
        end if
    else if LEAF(T) then
        return false
    else if NULL?(T[P[0]]) then
        return false
    else
        return PREFIX(T[P[0]],P[1...])
    end if
end function
```

Implementation

Size of nodes:

small alphabets fixed size internal nodes

large alphabets most branches non-existent: use variable-sized data structure

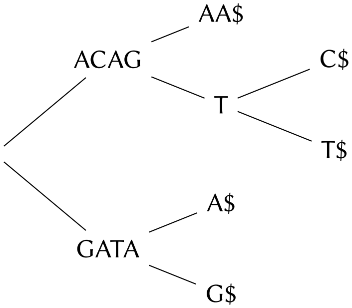
Single-branch internal nodes:

compressed trie collapse the internal nodes and concatenate the labels.

Implementation

Compressed trie:

- alphabet: `A`, `C`, `G`, `T`
- set of strings: `GATAA`, `ACAGAA`, `GATAG`, `ACAGTC`, `ACAGTT`



Suffix trees

- tries allow efficient – $\Theta(m)$ – match
  - at the *beginning* of the text,
  - for *multiple* texts;
- string match performs match
  - at the beginning of all suffixes of a text;
- … so solve string matching by inserting all *suffixes* of a text into a tree, then using prefix match of the pattern.

But:

- construction of suffix tree in $\Theta(n)$ time is tricky;
- worthwhile if doing multiple string matches (with arbitrary patterns) on the same text.

Work

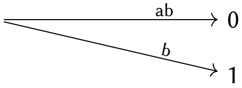
1. Reading
   - Drozdek, sections 7.2–7.4
   - Mark Nelson, *Fast String Searching with Suffix Trees*, Dr Dobb's Journal (August 1996)
     - and references therein

Outline

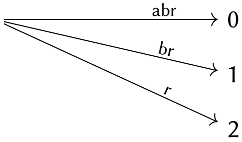
Example: a

$$\xrightarrow{\quad\quad\quad\quad\quad\quad\text{a}\quad} 0$$

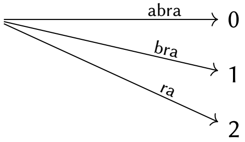
Example: ab



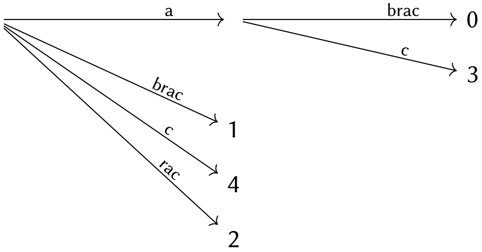
Example: abr



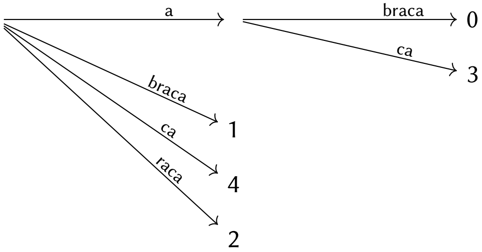
Example: abra



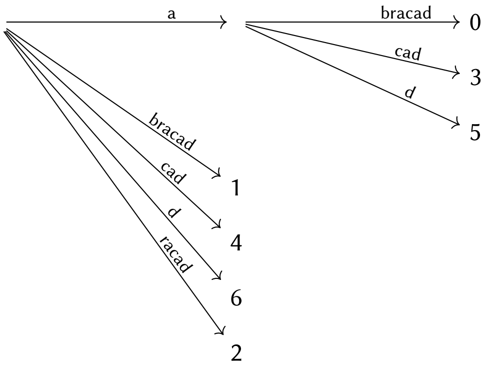
Example: abrac



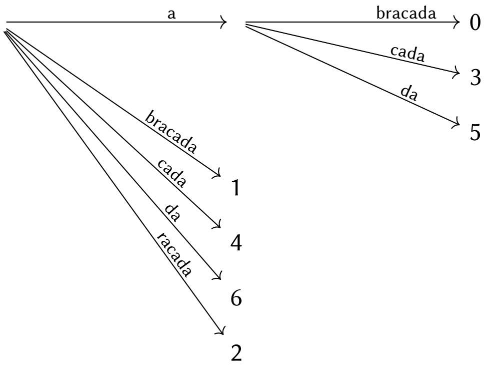
Example: abraca



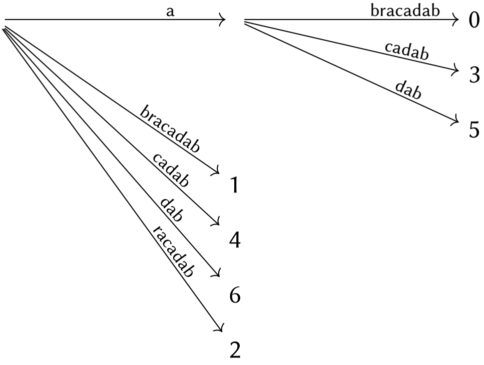
Example: abracad



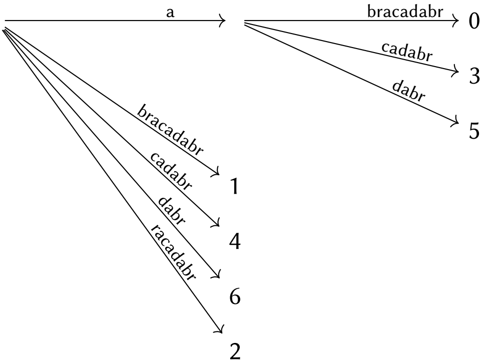
Example: abracada



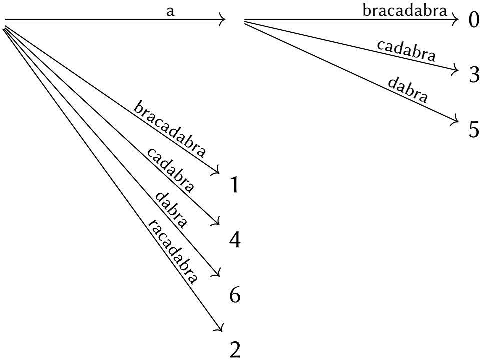
Example: abracadab



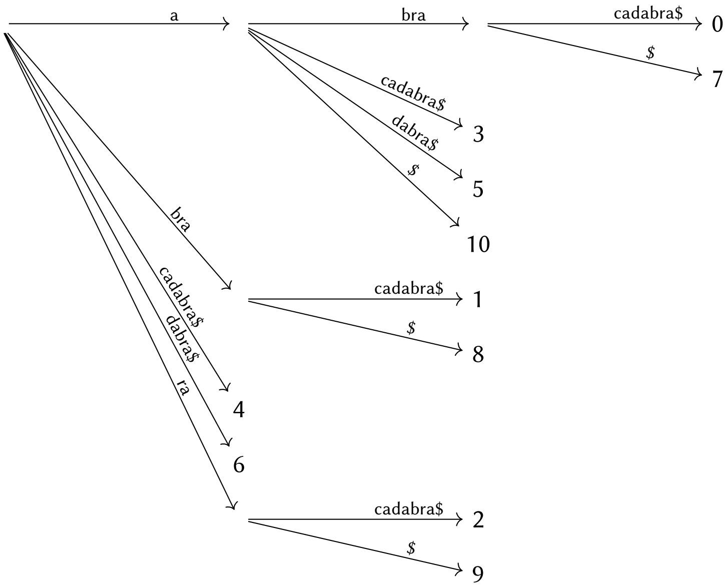
Example: abracadabr



Example: abracadabra



Example: abracadabra$



Work

1. Reading
   - Drozdek, section 13.1.8