Sorting and Searching Strings, Suffixes and Substrings

Juha Kärkkäinen

University of Helsinki, Finland

July 2004

Summer School on Experimental Algorithmics

Rungsted Kyst, Denmark

Contents

- Sorting and Searching (simple objects)
  - binary search
  - quicksort
- Strings (of simple objects)
  - string binary search
  - multikey quicksort
- Suffixes
  - difference cover samples
- Substrings
  - longest common prefix array

Binary Search

BinarySearch(A, q) // $A =$ sorted array of objects
// $q =$ query object
// returns true if q is in A

if $|A| = 0$ return false

let $A_l \cdot a_m \cdot A_r = A$ // a_m is the middle object in A

if $q = a_m$ then return true
else if $q < a_m$ then return **BinarySearch**(q, A_l)
else if $q > a_m$ then return **BinarySearch**(q, A_r)

- Complexity: $\mathcal{O}(\log n)$ $n = |A|$

(Ternary) Quicksort

Quicksort(A) // $A =$ array of objects

if $|A| \leq 1$ return

choose pivot $p \in A$

$A_< := \{a \in A \mid a < p\}$
$A_= := \{a \in A \mid a = p\}$
$A_> := \{a \in A \mid a > p\}$

Quicksort$(A_<)$
Quicksort$(A_>)$

$A := A_< \cdot A_= \cdot A_>$

- Complexity: $\mathcal{O}(n \log n)$

String Sorting and Searching

- lexicographic string comparisons can be slow
- n strings with a common prefix of length ℓ
  - Binary search: $\mathcal{O}(\ell \log n)$ time
  - Quicksort: $\mathcal{O}(\ell n \log n)$ time

String Sorting and Searching

- lexicographic string comparisons can be slow
- n strings with a common prefix of length ℓ
  - Binary search: $\mathcal{O}(\ell \log n)$ time
  - Quicksort: $\mathcal{O}(\ell n \log n)$ time
- n random strings over a constant alphabet
  - random comparison: $\mathcal{O}(1)$ time
  - Binary search: $\mathcal{O}(\log^2 n)$ time $(= 2 + 3 + \cdots + \log n)$
  - Quicksort: $\mathcal{O}(n \log^2 n)$ time $(= 2n + 3n + \cdots + n \log n)$

String Sorting and Searching

- lexicographic string comparisons can be slow
- n strings with a common prefix of length ℓ
  - Binary search: $\mathcal{O}(\ell \log n)$ time
  - Quicksort: $\mathcal{O}(\ell n \log n)$ time
- n random strings over a constant alphabet
  - random comparison: $\mathcal{O}(1)$ time
  - Binary search: $\mathcal{O}(\log^2 n)$ time $(= 2 + 3 + \cdots + \log n)$
  - Quicksort: $\mathcal{O}(n \log^2 n)$ time $(= 2n + 3n + \cdots + n \log n)$
- Not that bad in practice
- But there are better algorithms

Efficient String Sorting and Searching

1. compare characters not strings (radix based)
   - trie
   - radix sort
2. compare characters not strings (comparison based)
   - ternary search tree [Bentley & Sedgewick '97]
   - multikey quicksort [Bentley & Sedgewick '97]
3. LCP-aware string comparison
   - string binary search
   - almost any search tree can be modified for strings [Grossi & Italiano '99]

Strings

- string is an array of characters
  $s = s[0,m) = s[0]s[1]s[2]\cdots s[m-1]$
- ℓ-prefix: $s[0,\ell) = s[0]s[1]\cdots s[\ell-1]$
- longest common prefix (LCP):
  $lcp(s,t) = \text{argmax}_\ell\ s[0,\ell) = t[0,\ell)$
- distinguishing prefix:
  $dp_A(q) = 1 + \max_{a \in A\setminus\{q\}} lcp(q,a)$
  - lower bound for searching q in A
- total distinguishing prefix:
  $DP(A) = \sum_{a\in A} dp_A(a)$
  - lower bound for sorting A

```
comment
compile

alignment
all
allocate
alphabet
alternate
alternative
```


Multikey quicksort *[Bentley & Sedgewick '97]*

MULTIKEY-QUICKSORT(A, ℓ) // $A =$ array of strings with
// common ℓ-prefix

if $|A| \le 1$ then return
choose pivot $p \in A$
$A_< := \{s \in A \mid s[\ell] < p[\ell]\}$ // compare characters
$A_= := \{s \in A \mid s[\ell] = p[\ell]\}$
$A_> := \{s \in A \mid s[\ell] > p[\ell]\}$
MULTIKEY-QUICKSORT($A_<$, ℓ)
MULTIKEY-QUICKSORT($A_=$, $\ell+1$)
MULTIKEY-QUICKSORT($A_>$, ℓ)
$A := A_< \cdot A_= \cdot A_>$

- Complexity: $\mathcal{O}(n\log n + DP(A))$
- $n\log n$ $<$- or $>$-comparisons, $DP(A)$ $=$-comparisons

String Binary Search *[Manber & Myers '89]*

- compares strings not characters
- LCP-aware string comparison
  - skip known common prefix
  - compute LCP as side result in comparison
- the array is augmented with precomputed LCPs
- Complexity: $\mathcal{O}(\log n + dp_A(q))$

String Binary Search *[Manber & Myers '89]*

BINARYSEARCH(A, q, $lcp(q,a_l)$, $lcp(q,a_r)$)
// a_l, a_r bound A from left and right
if $|A| = 0$ return false
let $A_l \cdot a_m \cdot A_r = A$ // a_m is the middle object in A
// $lcp(a_m,a_l)$ and $lcp(a_m,a_r)$ stored with a_m
if $lcp(a_m,a_l) < lcp(q,a_l)$ or $lcp(a_m,a_r) > lcp(q,a_r)$ then
BINARYSEARCH(q, A_l, $lcp(q,a_\ell)$, $lcp(a_m,a_l)$)
else if $lcp(a_m,a_l) > lcp(q,a_l)$ or $lcp(a_m,a_r) < lcp(q,a_r)$ then
BINARYSEARCH(q, A_r, $lcp(q,a_l)$, $lcp(a_m,a_r)$)
else ($lcp(a_m,a_l) = lcp(q,a_l)$ and $lcp(a_m,a_r) = lcp(q,a_r)$)
$\ell := \max(lcp(q,a_\ell), lcp(q,a_r))$
$\ell := \ell + lcp(q[\ell\ldots), a_m[\ell\ldots))$ // now $\ell = lcp(q,a_m)$
if $|q| = |a_m| = \ell$ then return true
if $q[\ell] < a_m[\ell]$ then BINARYSEARCH(q, A_l, $lcp(q,a_l)$, ℓ)
if $q[\ell] > a_m[\ell]$ then BINARYSEARCH(q, A_r, ℓ, $lcp(q,a_r)$)

Sorting and Searching Suffixes

- text $T = T[0, N)$
- suffix $S_i = T[i, N)$
- q and elements of A are suffixes of T
- $T = \mathtt{aa...a}$ or $T = s \cdot s$
  - String binary search: $\mathcal{O}(\log n + N)$
  - Multikey quicksort: $\mathcal{O}(n \log n + nN)$
- linear search time and quadratic sort time
- there are bad cases in practice
  - long periodic or repeated substrings
  - genomic data
  - duplicates in file or document collections

Suffix Array

- suffix array is the sorted array of all suffixes of T
- many algorithms with $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$ time
- sorting an arbitrary set of suffixes:
  - sort all suffixes and discard unwanted ones
- searching suffixes:
  - compare ranks
- space consuming when $n \ll N$

Burrows–Wheeler Transform (BWT)

- a permutation of the text
- important in text compression and indexing
- suffix array is space bottleneck in computing BWT
text	$\longrightarrow$	suffix array	$\longrightarrow$	BWT
N bytes		$4N$ bytes		N bytes
$2N$ bits		$4N$ bytes		$2N$ bits

Burrows–Wheeler Transform (BWT)

- a permutation of the text
- important in text compression and indexing
- suffix array is space bottleneck in computing BWT
text	$\longrightarrow$	suffix array	$\longrightarrow$	BWT
N bytes		$4N$ bytes		N bytes
$2N$ bits		$4N$ bytes		$2N$ bits
- BWT by blockwise suffix array construction:
  - text $\longrightarrow$ suffix array block $\longrightarrow$ BWT block
- needs
  - suffix searching to compute block sizes
  - suffix sorting to sort blocks

Difference Cover Samples

- developed for suffix array construction
  - [Burkhardt & Kärkkäinen '03] [Kärkkäinen & Sanders '03]
- precomputed data structure $DC(T)$
- parameter $v \in [3, N^{2/3})$
- size $\mathcal{O}(N/\sqrt{v})$
- supports constant time comparison of two suffixes with a common v-prefix

Sorting and Searching using $DC(T)$

- searching
  - string search using only v-prefixes
  - if result is larger than one, finish search using $DC(T)$
  - $\mathcal{O}(\log n + v)$ time using string binary search
- sorting
  - string sort using only v-prefixes
  - remaining unsorted groups are sorted using $DC(T)$
  - $\mathcal{O}(n(\log n + v))$ time using multikey quicksort
- given $DC(T)$ for $v = \log n$
  - optimal (comparison-based) sorting and searching
  - space $\mathcal{O}(N/\sqrt{\log n})$

Sample Ranks

$T =$ `whowhowhowho`

- choose a sample of suffixes

```
howhowhowho
    owhowho
     whowho
       owho
```

- sort the sample

```
1  howhowhowho
2  owho
3  owhowho
4  whowho
```

- augment text with sample ranks

```
whowhowhowho
 1    34  2
```


Comparing Suffixes with Sample Ranks

- compare $(v-1)$-prefixes

```
howhowhowho
howhowho
```

- if same, compare aligned ranks within first v positions

```
howhowhowho
1    34  2

howhowho
 34  2
```

- Sample Condition: for all $i, j \in [0, N-v]$, there is $\ell \in [0, v)$ such that $i + \ell$ and $j + \ell$ are sample positions

Difference Covers

Definition $D \subseteq [0, v)$ is a difference cover D modulo v if

$$\{(i - j) \bmod v \mid i, j \in D\} = [0, v)$$

Example
$D = \{2, 4, 5\}$ is a difference cover modulo 7

	$i - j$
0	$2 - 2$
1	$5 - 4$
2	$4 - 2$
4	$2 - 5$
5	$2 - 4$
6	$4 - 5$

Lemma For all $i, j \in \mathbb{Z}$, there exists $\ell \in [0, v)$ such that $(i + \ell) \bmod v$ and $(j + \ell) \bmod v$ are in D.

Proof There exists $i', j' \in D$ such that $i' - j' \equiv i - j \pmod{v}$. Let $\ell = (i' - i) \bmod v$. Then $i + \ell \equiv i' \in D \pmod{v}$ and $j + \ell \equiv i' - (i - j) \equiv j' \in D \pmod{v}$.

Difference Cover Sample

Definition A difference cover sample is a set

$$C = \{i \in [0, n) \mid i \bmod v \in D\}\ ,$$

where D is a difference cover modulo v.

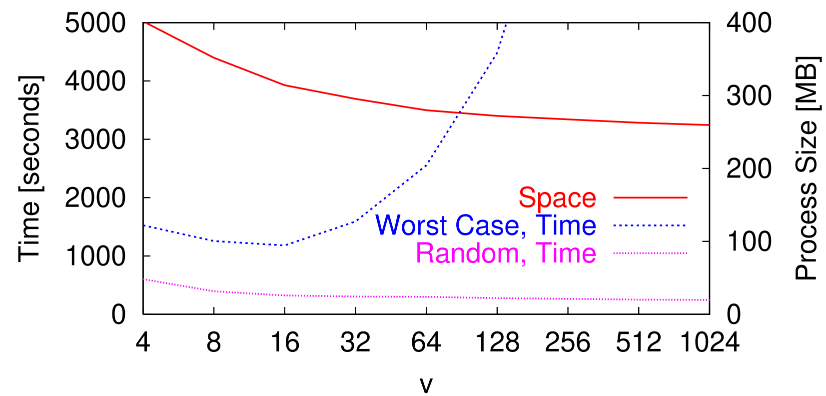
Example $D = \{2, 4, 5\}$ is a difference cover modulo 7.
$C =$ | 0 1 **2** 3 **4** **5** 6 | 7 8 **9** 10 **11** **12** 13 | 14 15 **16** 17 **18** **19** 20 | ...

Lemma C Satisfies the Sample Condition:
For all $i, j \in [0, N - v]$, there exists $\ell \in [0, v)$ such that $i + \ell$ and $j + \ell$ are in C.

Difference Cover Samples

- [Colbourn & Ling '00]: difference cover of size $\mathcal{O}(\sqrt{v})$ is easy to compute for all $v \Longrightarrow$ sample size $\mathcal{O}(N/\sqrt{v})$
- aligned samples can be found in constant time after $\mathcal{O}(v)$ time and space preprocessing
- construction
  - using suffix array in $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$ time
  - $\mathcal{O}(N/\sqrt{v})$ space and $\mathcal{O}(N\sqrt{v} + (N/\sqrt{v}) \log N)$ time

Experiment: Suffix Array Construction



text length = 50M
alphabet size = 26

Worst Case = (1K random)^{50K}

Sorting and Searching Substrings

- compare substring as its extension to suffix
- problem if one substring is a prefix of another

  `[com]pare...`
  `[commit]tee...`
- problem can be corrected if the LCP of the extensions is known
- difference cover samples can be augmented with LCP information to find LCPs in constant time
- same asymptotic complexities as in the case of suffixes
- is it practical?

Conclusion

- simple object $\longrightarrow$ string $\longrightarrow$ suffix $\longrightarrow$ substring
  - increasing difficulty of sorting and searching
  - incremental algorithms and data structures
- many challenges for algorithm engineering
  - LCP-aware string comparison
  - string binary search
  - cache-efficient comparison-based string sorting
- lots of real world data available !