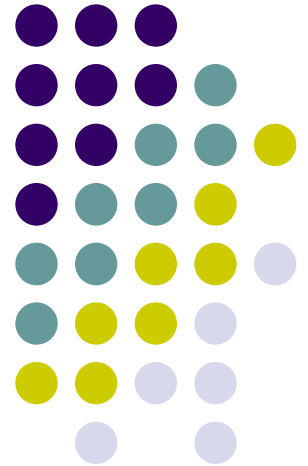


Algorithms

Chapter 7.1, 7.2, 7.3





ROAD MAP

- **Space-for-time tradeoffs**
 - Sorting by Counting
 - Input Enhancement in String Matching
 - Hashing



Space-for-time tradeoffs

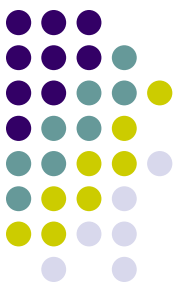
Two varieties of space-for-time algorithms:

- input enhancement — preprocess the input (or its part) to store some info to be used later in solving the problem
 - counting sorts
 - string searching algorithms
- prestructuring — preprocess the input to make accessing its elements easier
 - hashing
 - indexing schemes (e.g., B-trees)



ROAD MAP

- **Space-for-time tradeoffs**
 - **Sorting by Counting**
 - Input Enhancement in String Matching
 - Hashing



Sorting by Counting

ALGORITHM *ComparisonCountingSort*($A[0..n - 1]$)

//Sorts an array by comparison counting

//Input: An array $A[0..n - 1]$ of orderable elements

//Output: Array $S[0..n - 1]$ of A 's elements sorted in nondecreasing order

for $i \leftarrow 0$ **to** $n - 1$ **do** $Count[i] \leftarrow 0$

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

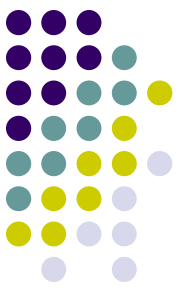
if $A[i] < A[j]$

$Count[j] \leftarrow Count[j] + 1$

else $Count[i] \leftarrow Count[i] + 1$

for $i \leftarrow 0$ **to** $n - 1$ **do** $S[Count[i]] \leftarrow A[i]$

return S



Sorting by Counting

Array $A[0..5]$

62	31	84	96	19	47
----	----	----	----	----	----

Initially

Count []

0	0	0	0	0	0
---	---	---	---	---	---

After pass $i = 0$

Count []

3	0	1	1	0	0
---	---	---	---	---	---

After pass $i = 1$

Count []

	1	2	2	0	1
--	---	---	---	---	---

After pass $i = 2$

Count []

		4	3	0	1
--	--	---	---	---	---

After pass $i = 3$

Count []

			5	0	1
--	--	--	---	---	---

After pass $i = 4$

Count []

				0	2
--	--	--	--	---	---

Final state

Count []

3	1	4	5	0	2
---	---	---	---	---	---

Array $S[0..5]$

19	31	47	62	84	96
----	----	----	----	----	----

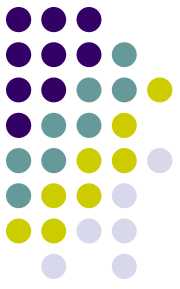
FIGURE 7.1 Example of sorting by comparison counting.



ROAD MAP

- **Space-for-time tradeoffs**
 - Sorting by Counting
 - **Input Enhancement in String Matching**
 - Hashing

Review: String searching by brute force



pattern: a string of m characters to search for

text: a (long) string of n characters to search in

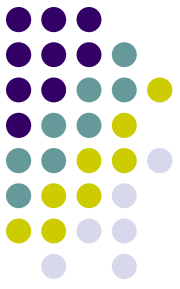
Brute force algorithm

Step 1 Align pattern at beginning of text

Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String searching by preprocessing



Several string searching algorithms are based on the input

enhancement idea of preprocessing the pattern

- Boyer -Moore algorithm preprocesses pattern right to left and store information into two tables
- Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table



Horspool's Algorithm

A simplified version of Boyer-Moore algorithm:

- preprocesses pattern to generate a shift table that determines how much to shift the pattern when a mismatch occurs
- always makes a shift based on the text's character c aligned with the last character in the pattern according to the shift table's entry for c

Horspool's Algorithm



$s_0 \quad \dots \quad S \quad \dots \quad s_{n-1}$
 $\quad \quad \quad \diagup$
 $\quad \quad \quad B \ A \ R \ B \ E \ R$
 $\quad \quad \quad \quad \quad \quad B \ A \ R \ B \ E \ R$

Case 1

$s_0 \quad \dots \quad B \quad \dots \quad s_{n-1}$
 $\quad \quad \quad \diagup$
 $\quad \quad \quad B \ A \ R \ B \ E \ R$
 $\quad \quad \quad \quad \quad \quad B \ A \ R \ B \ E \ R$

Case 2

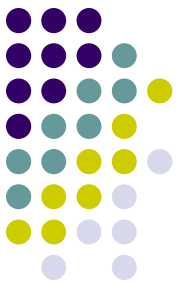
$s_0 \quad \dots \quad M \ E \ R \quad \dots \quad s_{n-1}$
 $\quad \quad \quad \diagup \quad \parallel \quad \parallel$
 $\quad \quad \quad L \ E \ A \ D \ E \ R$
 $\quad \quad \quad \quad \quad \quad L \ E \ A \ D \ E \ R$

Case 3

$s_0 \quad \dots \quad A \ R \quad \dots \quad s_{n-1}$
 $\quad \quad \quad \diagup \quad \parallel$
 $\quad \quad \quad R \ E \ O \ R \ D \ E \ R$
 $\quad \quad \quad \quad \quad \quad R \ E \ O \ R \ D \ E \ R$

Case 4

Shift table



- Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m - 1 \text{ characters of the pattern;} \\ \text{the distance from the rightmost } c \text{ among the first } m - 1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases} \quad (7.1)$$

by scanning pattern before search begins and stored in a table called *shift table*

- Shift table is indexed by text and pattern alphabet
Eg, for BARBER:

character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

Example of Horspool's alg. application



character c	A	B	C	D	E	F	...	R	...	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R B A R B E R
 B A R B E R B A R B E R
 B A R B E R B A R B E R



Boyer-Moore algorithm

Based on same two ideas:

- comparing pattern characters to text from right to left
- precomputing shift sizes in two tables
 - *bad-symbol table* indicates how much to shift based on text's character causing a mismatch
 - *good-suffix table* indicates how much to shift based on matched part (suffix) of the pattern

Bad-symbol shift in Boyer-Moore algorithm



- If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k > 0$ matches

s_0	...	c	s_{i-k+1}	...	s_i	...	s_{n-1}	text
		\nparallel	\parallel		\parallel			
p_0	...	p_{m-k-1}	p_{m-k}	...	p_{m-1}			pattern

bad-symbol shift $d_1 = \max\{t_1(c) - k, 1\}$

Good-suffix shift in Boyer-Moore algorithm



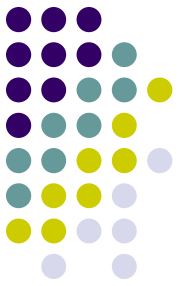
- Good-suffix shift d_2 is applied after $0 < k < m$ last characters were matched
- $d_2(k)$ = the distance between matched suffix of size k and its rightmost occurrence in the pattern that is not preceded by the same character as the suffix

Example: CABABA $d_2(1) = 4$

- If there is no such occurrence, match the longest part of the k -character suffix with corresponding prefix;
if there are no such suffix-prefix matches, $d_2(k) = m$

Example: WOWWOW $d_2(2) = 5$, $d_2(3) = 3$, $d_2(4) = 3$, $d_2(5) = 3$

Good-suffix shift in the Boyer-Moore alg. (cont.)

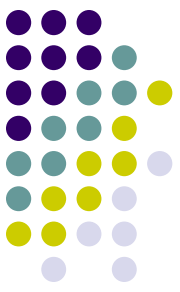


After matching successfully $0 < k < m$ characters, the algorithm shifts the pattern right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$ is bad-symbol shift

$d_2(k)$ is good-suffix shift



Boyer-Moore Algorithm (cont.)

Step 1 Fill in the bad-symbol shift table

Step 2 Fill in the good-suffix shift table

Step 3 Align the pattern against the beginning of the text

Step 4 Repeat until a matching substring is found or text ends:

Compare the corresponding characters right to left.

If no characters match, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and shift the pattern to the right by $t_1(c)$.

If $0 < k < m$ characters are matched, retrieve entry $t_1(c)$ from the bad-symbol table for the text's character c causing the mismatch and entry $d_2(k)$ from the good-suffix table and shift the pattern to the right by

$$d = \max \{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Example of Boyer-Moore alg. application



c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

B E S S _ K N E W _ A B O U T _ B A O B A B
 B A O B A B

$$d_1 = t_1(K) - 0 = 6$$

B A O B A B

$$d_1 = t_1(_) - 2 = 4 \quad \text{B A O B A B}$$

$$d_2 = 5$$

$$d_1 = t_1(_) - 1 = 5$$

$$d = \max\{4, 5\} = 5$$

$$d_2 = 2$$

$$d = \max\{5, 2\} = 5$$

B A O B A B

k	pattern	d_2
1	BAO <u>B</u> A <u>B</u>	2
2	<u>B</u> A <u>O</u> B <u>A</u> B	5
3	<u>B</u> A <u>O</u> B <u>A</u> B	5
4	<u>B</u> A <u>O</u> B <u>A</u> B	5
5	<u>B</u> A <u>O</u> B <u>A</u> B	5



ROAD MAP

- **Space-for-time tradeoffs**
 - Sorting by Counting
 - Input Enhancement in String Matching
 - **Hashing**



Hashing

- A very efficient method for implementing a *dictionary*, i.e., a set with the operations:
 - find
 - insert
 - delete
- Based on representation-change and space-for-time tradeoff ideas
- Important applications:
 - symbol tables
 - databases (*extendible hashing*)

Hash tables and hash functions



The idea of *hashing* is to map keys of a given file of size n into a table of size m , called the *hash table*, by using a predefined function, called the *hash function*,

$h: K \rightarrow \text{location (cell) in the hash table}$

Example: student records, key = SSN. Hash function:

$h(K) = K \bmod m$ where m is some integer (typically, prime)

If $m = 1000$, where is record with SSN= 314159265 stored?

Generally, a hash function should:

- be easy to compute
- distribute keys about evenly throughout the hash table

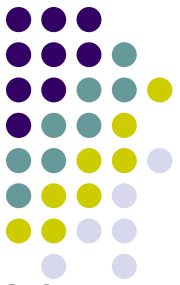


Collisions

If $h(K_1) = h(K_2)$, there is a *collision*

- Good hash functions result in fewer collisions but some collisions should be expected (*birthday paradox*)
- Two principal hashing schemes handle collisions differently:
 - *Open hashing*
 - each cell is a header of linked list of all keys hashed to it
 - *Closed hashing*
 - one key per cell
 - in case of collision, finds another cell by
 - *linear probing*: use next free bucket
 - *double hashing*: use second hash function to compute increment

Open hashing (Separate chaining)

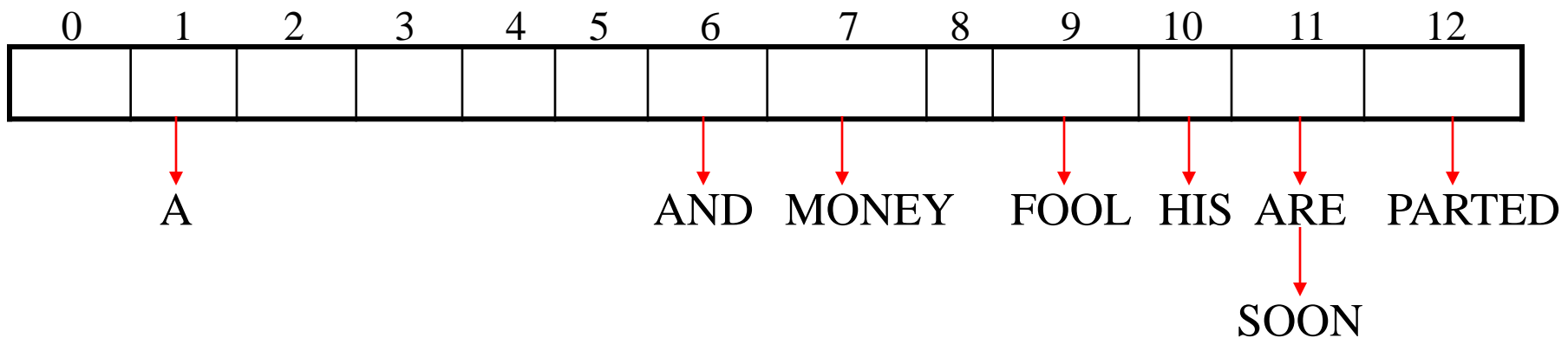


Keys are stored in linked lists outside a hash table whose elements serve as the lists' headers.

Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

$h(K)$ = sum of K 's letters' positions in the alphabet MOD 13

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12



Search for KID

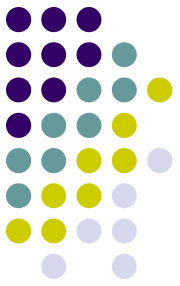


Open hashing (cont.)

- If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$. This ratio is called *load factor*.
- Average number of probes in successful, S , and unsuccessful searches, U :
$$S \approx 1 + \alpha/2, \quad U = \alpha$$
- Load α is typically kept small (ideally, about 1)
- Open hashing still works if $n > m$

Closed hashing (Open addressing)

Keys are stored inside a hash table.



Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
$h(K)$	1	9	6	10	7	11	11	12

	0	1	2	3	4	5	6	7	8	9	10	11	12
	A												
	A								FOOL				
	A					AND			FOOL				
	A					AND			FOOL	HIS			
	A					AND	MONEY		FOOL	HIS			
	A					AND	MONEY		FOOL	HIS	ARE		
	A					AND	MONEY		FOOL	HIS	ARE	SOON	
→ PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON	



Closed hashing (cont.)

- Does not work if $n > m$
- Avoids pointers
- Deletions are *not* straightforward
- Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ (hash table density) and collision resolution strategy. For linear probing:
$$S = (\frac{1}{2}) (1 + 1/(1 - \alpha))$$
 and
$$U = (\frac{1}{2}) (1 + 1/(1 - \alpha)^2)$$
- As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:

α	$\frac{1}{2}(1 + \frac{1}{1-\alpha})$	$\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5