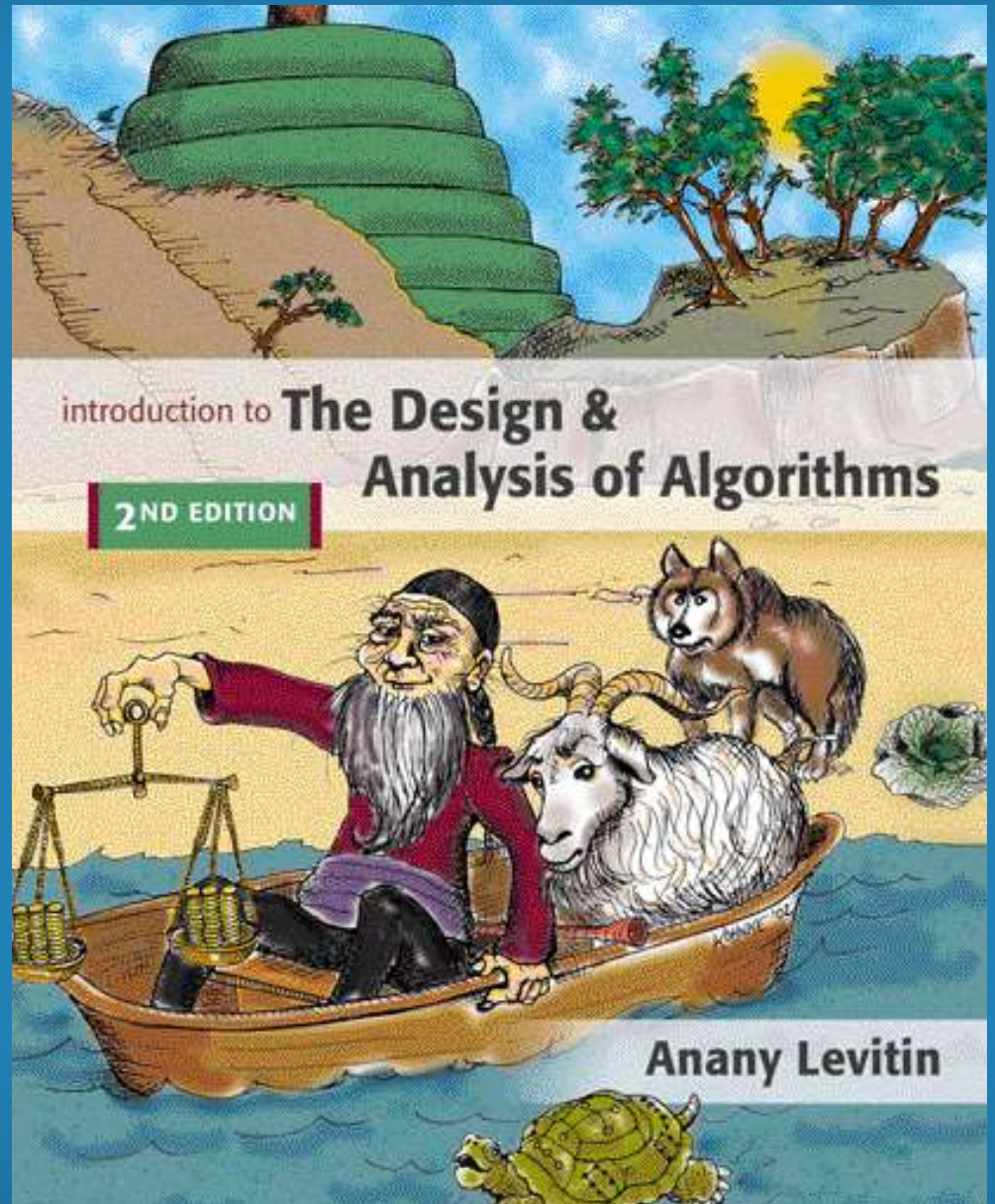
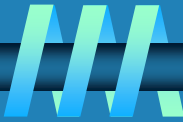


Chapter 7

Space and Time Tradeoffs



Space-for-time tradeoffs

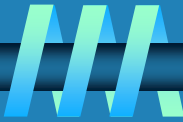


Two varieties of space-for-time algorithms:

- input enhancement — preprocess the input (or its part) to store some info to be used later in solving the problem
 - Sorting by counting
 - string searching algorithms

- prestructuring — preprocess the input to make accessing its elements easier
 - hashing
 - indexing schemes (e.g., B-trees)

Sorting by Counting



❧ **Algorithm ComparisonCounting($A[0..n-1]$)**

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

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

for $i \leftarrow 0$ to $n-1$ do $\text{Count}[i] \leftarrow 0$

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

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

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

$\text{Count}[j] \leftarrow \text{Count}[j] + 1$

else $\text{Count}[i] \leftarrow \text{Count}[i] + 1$

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

return S

Array $A[0..5]$

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

Initially

<i>Count</i> []	0	0	0	0	0	0
-----------------	---	---	---	---	---	---

After pass $i = 0$

<i>Count</i> []	3	0	1	1	0	0
-----------------	---	---	---	---	---	---

After pass $i = 1$

<i>Count</i> []		1	2	2	0	1
-----------------	--	---	---	---	---	---

After pass $i = 2$

<i>Count</i> []			4	3	0	1
-----------------	--	--	---	---	---	---

After pass $i = 3$

<i>Count</i> []				5	0	1
-----------------	--	--	--	---	---	---

After pass $i = 4$

<i>Count</i> []					0	2
-----------------	--	--	--	--	---	---

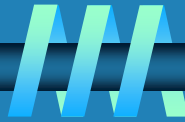
Final state

<i>Count</i> []	3	1	4	5	0	2
-----------------	---	---	---	---	---	---

Array $S[0..5]$

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

Continued.....



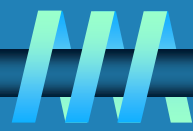
Ω $C(n) = \sum_{i=0}^{n-2} 1$

Ω Example

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

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

Distribution Counting



Algorithm Distribution Counting($A[0..n-1], l, u$)

//Input: An array $A[0..n-1]$ of integers between l and u

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

for $j \leftarrow 0$ to $u-l$ do $D[j] \leftarrow 0$ //initialise freq

for $i \leftarrow 0$ to $n-1$ do $D[A[i]-l] \leftarrow D[A[i]-l]+1$ //compute freq

for $j \leftarrow 1$ to $u-l$ do $D[j] \leftarrow D[j-1]+D[j]$ //reuse for distribution

for $i \leftarrow n-1$ down to 0 do

$j \leftarrow A[i]-l$

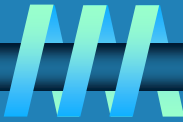
$S[D[j]-1] \leftarrow A[i]$

$D[j] \leftarrow D[j]-1$

return S

Note that the distribution values indicate the proper positions for the last occurrences of their elements in the final sorted array. If we index array positions from 0 to $n-1$, the distribution values must be reduced by 1 to get corresponding element positions.

Example



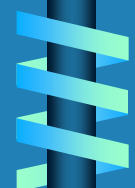
13	11	12	13	12	12
----	----	----	----	----	----

D[0..2]

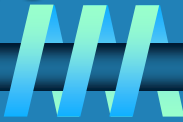
S[0..5]

1	4	6
1	3	6
1	2	6
1	2	5
1	1	5
0	1	5

			12		
		12			
					13
	12				
11					
				13	



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

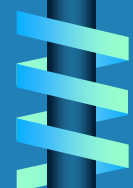
Brute force algorithm

Step 1 Align pattern at beginning of text

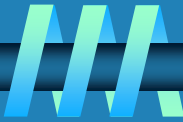
Step 2 Moving from left to right, compare each character of pattern to the corresponding character in text until either all characters are found to match (successful search) or a mismatch is detected

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

Time complexity (worst-case): $O(mn)$



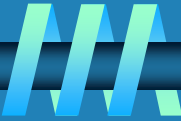
String searching by preprocessing



Several string searching algorithms are based on the input enhancement idea of preprocessing the **pattern**

- ❧ **Knuth-Morris-Pratt (KMP) algorithm preprocesses pattern left to right to get useful information for later searching**
 $O(m+n)$ time in the worst case
- ❧ **Boyer-Moore algorithm preprocesses pattern right to left and store information into two tables**
 $O(m+n)$ time in the worst case
- ❧ **Horspool's algorithm simplifies the Boyer-Moore algorithm by using just one table**

Horspool's Algorithm

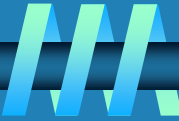


A simplified version of Boyer-Moore algorithm:

- 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 compared (mismatched) character in the pattern according to the shift table's entry for c



How far to shift?



Look at first (rightmost) character in text that was compared:

❧ The character is not in the pattern

.....**C**..... (C not in pattern)

BAOBAB

❧ The character is in the pattern (but not the rightmost)

.....**O**..... (O occurs once in pattern)

BAOBAB

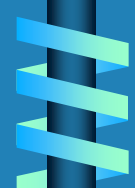
.....**A**..... (A occurs twice in pattern)

BAOBAB

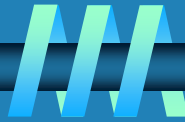
❧ The rightmost characters do match

.....**B**.....

BAOBAB



Shift table



Shift sizes can be precomputed by the formula

$$t(c) = \begin{cases} \text{distance from } c\text{'s rightmost occurrence in pattern} \\ \text{among its first } m-1 \text{ characters to its right end} \\ \text{pattern's length } m, \text{ otherwise} \end{cases}$$

by scanning pattern before search begins and stored in a table called *shift table*. After the shift, the right end of pattern is $t(c)$ positions to the right of the last compared character in text.

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

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6

Example of Horspool's algorithm

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	—
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

BARD LOVED BANANAS

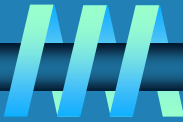
BAOBAB

BAOBAB

BAOBAB

BAOBAB (unsuccessful search)

Boyer-Moore algorithm

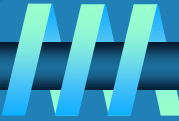


Based on the same two ideas:

- 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 (taking advantage of the periodic structure of the pattern)

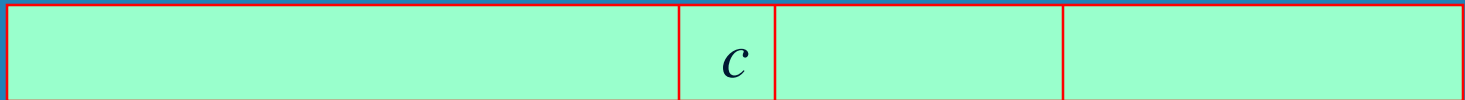


Bad-symbol shift in Boyer-Moore algorithm



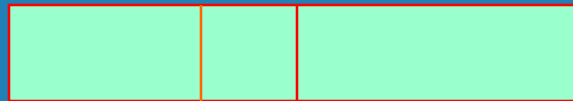
- ❧ If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's
- ❧ If the rightmost character of the pattern does match, BM compares preceding characters right to left until either all pattern's characters match or a mismatch on text's character c is encountered after $k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

text



k matches

pattern



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

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 (the last letter of) the matched suffix of size k and (the last letter of) its rightmost occurrence in the pattern that is not preceded by the same character preceding the suffix

Example: CABABA $d_2(1) = 4$

— — — — —

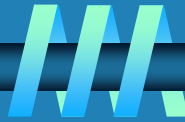
.....c^{zyx}.....^k
a^{zyx}.....b^{zyx}
a^{zyx}...b^{zyx}^{d₂(k)}

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

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

— — — — —
 = — — — — —

Boyer-Moore Algorithm



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

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

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

$d_2(k)$ is good-suffix shift

Example: Find pattern **AT_THAT** in

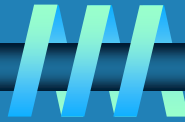
WHICH_FINALY_HALTS. _ **AT_THAT**
 AT_THAT AT_THAT THAT AT_THAT THAT

$$d_1 = 7 - 1 = 6 \quad d_1 = 4 - 2 = 2$$

t A H T _ ?
 1 2 3 4 7

d_2 1 2 3 4 5 6
 3 5 5 5 5 5

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

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	—
1	2	6	6	6	6	6	6	6	6	6	6	6	6	3	6	6	6	6	6	6	6	6	6	6	6	6

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

$$d_1 = t(K) = 6$$

B A O B A B

$$d_1 = t(_) - 2 = 4$$

$$\underline{d_2(2) = 5}$$

B A O B A B

$$\underline{d_1 = t(_) - 1 = 5}$$

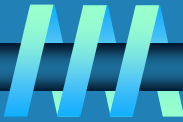
$$d_2(1) = 2$$

B A O B A B (success)

<i>k</i>	pattern	<i>d</i> ₂
1	BAO B AB	2
2	B AOBAB	5
3	B AOBAB	5
4	B AOBAB	5
5	B AOBAB	5

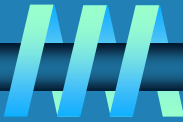
Worst-case time complexity: $O(n+m)$.

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:

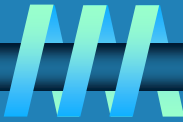
$h(K) = K \bmod m$ where m is some integer

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

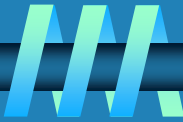
Collisions



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

- ❧ Good hash functions result in fewer collisions but some collisions should be expected
- ❧ 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)

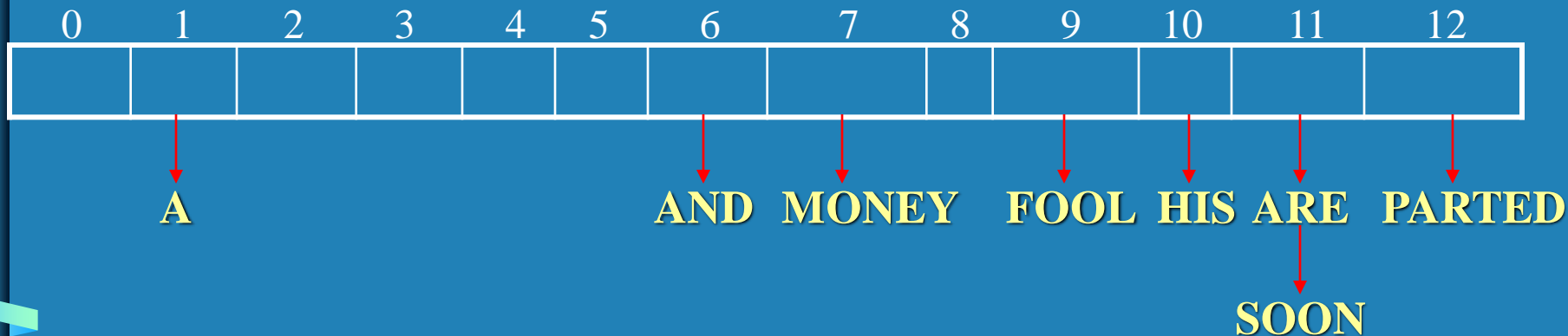


Keys are stored in linked lists outside a hash table whose elements serve as the lists' headers.

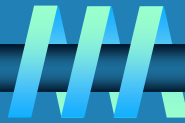
Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED

$h(K)$ = 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



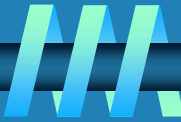
Search for KID $(11+9+4) \bmod 13$



In general, the efficiency of searching depends on the lengths of the linked lists, which, in turn, depend on the dictionary and table sizes, as well as the quality of the hash function. If the hash function distributes n keys among m cells of the hash table about evenly, each list will be about n/m keys long. The ratio $\alpha = n/m$, called the *load factor* of the hash table, plays a crucial role in the efficiency of hashing. In particular, the average number of pointers (chain links) inspected in successful searches, S , and unsuccessful searches, U , turn out to be

$$S \approx 1 + \frac{\alpha}{2} \text{ and } U = \alpha, \quad (7.4)$$

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*.

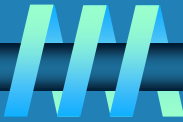
⌚ For ideal hash functions, the average numbers of probes in successful, S , and unsuccessful searches, U :

$$S \approx 1 + \alpha/2, \quad U = \alpha$$

⌚ Load α is typically kept small (ideally, about 1)

⌚ Open hashing still works if $n > m$

Closed hashing (Open addressing)

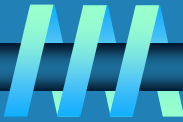


Keys are stored inside a hash table.

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:

$$S \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \text{ and } U \approx \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

- ❧ As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:

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