

DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

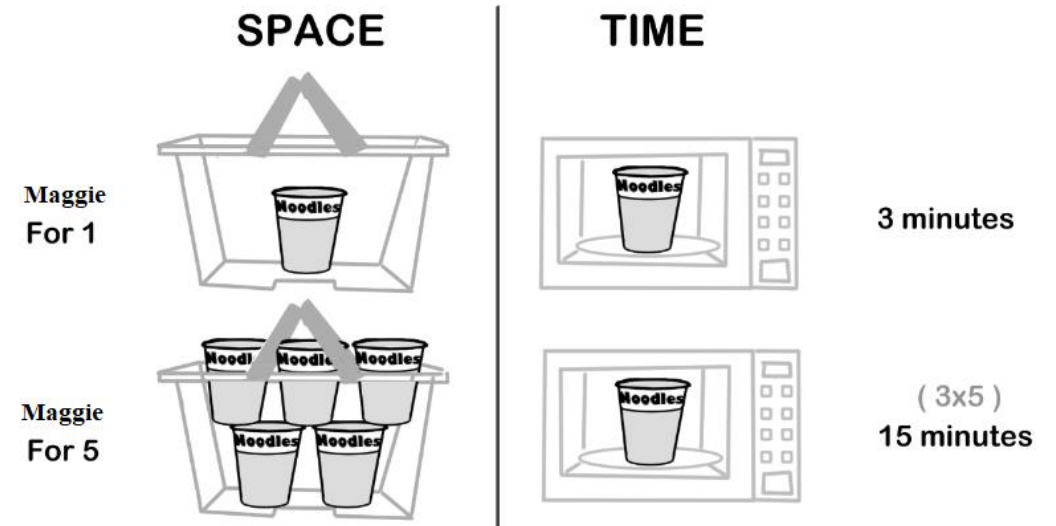
Lecture 27, 28, 29, 30

Space and Time Trade-Offs

Input Enhancement:

Sorting by Counting

String Matching: Horspool's, Boyer-Moore Algorithms



Space-for-time tradeoffs

Two varieties of space-for-time algorithms:

Input enhancement – preprocess the input (or its part) to store some **additional info** to be used later in solving the problem.

- Sorting by counting----- comparison counting sort, distribution counting sort
- String searching algorithms

Prestructuring – preprocess the input to make **accessing its elements easier**.

- Hashing
- Indexing schemes (e.g., B-trees)

Sorting by Counting : Comparison Counting Sort

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

Example

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
----	----	----	----	----	----

Time complexity for the above algorithm ?? $O(n^2)$

Example of sorting by distribution counting

Consider the sorting array

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

The frequency and distribution arrays

Array values	11	12	13
Frequencies	1	3	2
Distribution values	1	4	6

Sorting by Counting : Distribution Counting Sort

ALGORITHM *DistributionCountingSort*($A[0..n-1]$, l , u)

//Sorts an array of integers from a limited range by distribution counting

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

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

for $j \leftarrow 0$ to $u - l$ do $D[j] \leftarrow 0$ //initialize frequencies

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

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

for $i \leftarrow n - 1$ downto 0 do

$j \leftarrow A[i] - l$

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

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

return S

	$D[0..2]$			$S[0..5]$					
$A[5] = 12$	1	4	6				12		
$A[4] = 12$	1	3	6			12			
$A[3] = 13$	1	2	6						13
$A[2] = 12$	1	2	5		12				
$A[1] = 11$	1	1	5	11					
$A[0] = 13$	0	1	5					13	

Time complexity for the above algorithm ?? $O(n)$

Review: String searching/matching 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

Worst case time complexity of the above brute force approach ?? $O(mn)$

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

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

How far to shift?

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

Case 1:

s_0	...	S	...	s_{n-1}					
		X							
	B	A	R	B	E	R			
				B	A	R	B	E	R

Case 3:

s_0	...	M	E	R	...	s_{n-1}				
		X								
	L	E	A	D	E	R				
					L	E	A	D	E	R

Case 2:

s_0	...	B	...	s_{n-1}					
		X							
	B	A	R	B	E	R			
				B	A	R	B	E	R

Case 4:

s_0	...	A	R	...	s_{n-1}					
		X								
	R	E	O	R	D	E	R			
				R	E	O	R	D	E	R

Shift table

Shift sizes can be precomputed by the formula

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

Shift table for the pattern:

BARBER

A	B	E	R
4	2	1	3

- Scan pattern before search begins and store 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 : Pseudocode

ALGORITHM *ShiftTable*($P[0..m - 1]$)

//Fills the shift table used by Horspool's and Boyer-Moore algorithms

//Input: Pattern $P[0..m - 1]$ and an alphabet of possible characters

//Output: $Table[0..size - 1]$ indexed by the alphabet's characters and

// filled with shift sizes computed by formula (7.1)

for $i \leftarrow 0$ **to** $size - 1$ **do** $Table[i] \leftarrow m$

for $j \leftarrow 0$ **to** $m - 2$ **do** $Table[P[j]] \leftarrow m - 1 - j$

return $Table$

Example of Horspool's algorithm

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

The actual search in a particular text proceeds as follows:

```
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
```

Horspool's algorithm : Pseudocode

ALGORITHM *HorspoolMatching*($P[0..m - 1]$, $T[0..n - 1]$)
//Implements Horspool's algorithm for string matching
//Input: Pattern $P[0..m - 1]$ and text $T[0..n - 1]$
//Output: The index of the left end of the first matching substring
// or -1 if there are no matches
ShiftTable($P[0..m - 1]$) //generate *Table* of shifts
 $i \leftarrow m - 1$ //position of the pattern's right end
while $i \leq n - 1$ **do**
 $k \leftarrow 0$ //number of matched characters
 while $k \leq m - 1$ **and** $P[m - 1 - k] = T[i - k]$ **do**
 $k \leftarrow k + 1$
 if $k = m$
 return $i - m + 1$
 else $i \leftarrow i + \text{Table}[T[i]]$
return -1

Worst case Complexity: **$O(mn)$**

But, require lesser number of shifts/steps/time on an average than the brute force solution.

Average case= $O(n)$

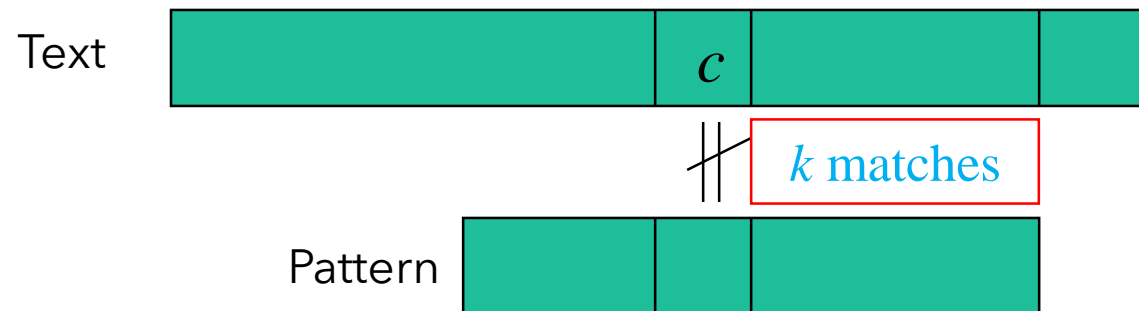
Boyer-Moore algorithm

Based on the same two ideas:

- Comparing pattern characters to text from right to left
- Precomputing shift sizes. But, in two ways
 - **Bad-symbol shift** (d_1) : indicates how much to shift based on text's character causing a mismatch (computed using the shift table in Horspool's Algorithm)
 - **Good-suffix shift** (d_2) : 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

- 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



Bad-symbol shift : Example

Example 1: $d_1(S) = t_1(S) - 2 = 6 - 2 = 4$

s_0	...			S	E	R		...	s_{n-1}	
				X						
		B	A	R	B	E	R			
					B	A	R	B	E	R

BARBER
4 3 2 1

Example 2: $d_1(A) = t_1(A) - 2 = 4 - 2 = 2$

s_0	...			A	E	R		...	s_{n-1}
				X					
		B	A	R	B	E	R		
				B	A	R	B	E	R

If $(t_1(c) - k) < 0$, then shift by ONE step.

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

Good-suffix shift

Example

k	pattern	d_2
1	ABC <u>B</u> AB	2
2	<u>AB</u> CBAB	4
3	<u>ABC</u> BAB	4
4	<u>ABCB</u> AB	4
5	<u>ABCBAB</u>	4

Final shift length (d) is calculated as:

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

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

Boyer-Moore Algorithm: Example

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

Pattern: B A O B A B
 3 2 1

Bad Symbol Table

<i>c</i>	A	B	C	D	...	0	...	Z	_
<i>t</i> ₁ (<i>c</i>)	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 S
B A O B A B

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

B A O B A B

$d_1 = t_1(_) - 2 = 4$ 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

Good Suffix Table

<i>k</i>	pattern	<i>d</i> ₂
1	BAO <u>B</u> A <u>B</u>	2
2	<u>B</u> AOBA <u>B</u>	5
3	<u>B</u> AOBA <u>B</u>	5
4	<u>B</u> AOBA <u>B</u>	5
5	<u>B</u> AOBA <u>B</u>	5

The Boyer-Moore algorithm

- Step 1** For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.
- Step 2** Using the pattern, construct the good-suffix shift table as described earlier.
- Step 3** Align the pattern against the beginning of the text.
- Step 4** Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c 's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the

|number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

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

Thank you!

Any queries?