DSE 2256 DESIGN & ANALYSIS OF ALGORITHMS

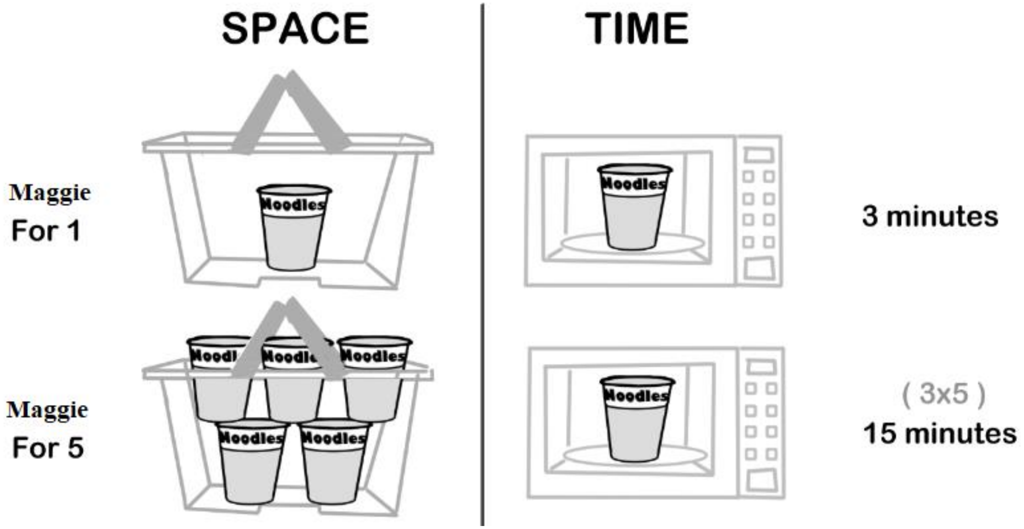
Lecture 27, 28, 29, 30

Space and Time Trade-Offs

Input Enhancement:

Sorting by Counting

String Matching: Horspool's, Boyer-Moore Algorithms



Instructors:

Dr. Savitha G,
Assistant Professor, DSCA, MIT, Manipal

Dr. Abhilash K. Pai,
Assistant Professor, DSCA, MIT, Manipal

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
- 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 ≤ u)
//Output: Array S[0..n − 1] of A's elements sorted in nondecreasing order
for j ← 0 to u − l do D[j] ← 0                           //initialize frequencies
for i ← 0 to n − 1 do D[A[i] − l] ← D[A[i] − l] + 1 //compute frequencies
for j ← 1 to u − l do D[j] ← D[j − 1] + D[j]         //reuse for distribution
for i ← n − 1 downto 0 do
    j ← A[i] − l
    S[D[j] − 1] ← A[i]
    D[j] ← 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

$$
\begin{array}{ccccccccccc}
s_0 & \dots & & & & & & c & \dots & s_{n-1} \\
 & & \mathrm{B} & \mathrm{A} & \mathrm{R} & \mathrm{B} & \mathrm{E} & \mathrm{R} & &
\end{array}
$$

How far to shift?

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

Case 1:

$s_0 \quad \ldots \quad \text{S} \quad \ldots \quad s_{n-1}$

$\neq$

B A R B E R

B A R B E R

Case 2:

$s_0 \quad \ldots \quad \text{B} \quad \ldots \quad s_{n-1}$

$\neq$

B A R B E R

B A R B E R

Case 3:

$s_0 \quad \ldots \quad \text{M E R} \quad \ldots \quad s_{n-1}$

$\neq \; \| \; \|$

L E A D E R

L E A D E R

Case 4:

$s_0 \quad \ldots \quad \text{A R} \quad \ldots \quad s_{n-1}$

$\neq \; \|$

R E O R D E R

R E O R D E R

Shift table

Shift sizes can be precomputed by the formula

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

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 $\quad ShiftTable(P[0..m-1])$

//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 ← m − 1                      //position of the pattern's right end
while i ≤ n − 1 do
    k ← 0                      //number of matched characters
    while k ≤ m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k + 1
    if k = m
        return i − m + 1
    else  i ← i + Table[T[i]]
return −1
```

<u>Worst case Complexity</u> : **O(mn)**

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

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

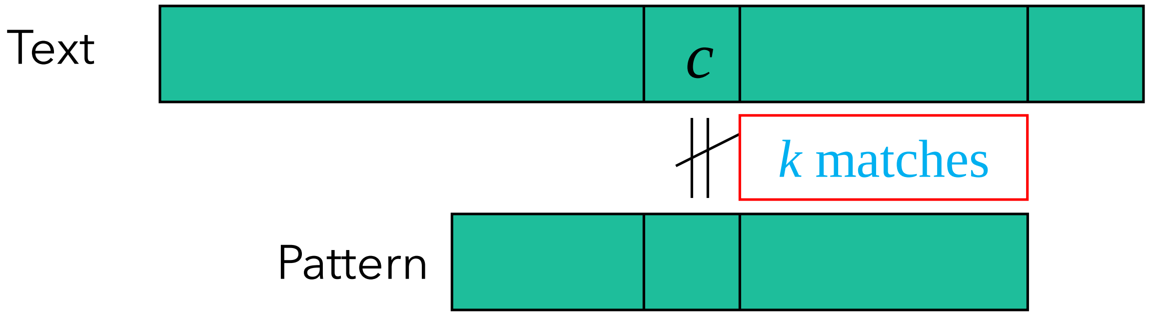
  - **Bad-symbol shift** (d_1) : indicates how much to shift based on text's character causing a mismatch (computed using the shift table in Horspool's Algorithm)

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

```
s0   . . .         S  E  R              . . .   s(n-1)
                   ≠  ‖  ‖
          B  A  R  B  E  R
                      B  A  R  B  E  R
```

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

```
s0   . . .         A  E  R              . . .   s(n-1)
                   ≠  ‖  ‖
             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	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	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 $d_1 = \max\{t_1(c) - k, 1\}$.

Boye-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

Bad Symbol Table

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 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

k	**pattern**	d_2
1	$\text{BAO}\overline{\text{B}}\text{A}\underline{\text{B}}$	2
2	$\overline{\text{B}}\text{AOB}\underline{\text{AB}}$	5
3	$\overline{\text{B}}\text{AO}\underline{\text{BAB}}$	5
4	$\overline{\text{B}}\text{A}\underline{\text{OBAB}}$	5
5	$\overline{\text{B}}\underline{\text{AOBAB}}$	5

Thank you!

Any queries?