Space & Time Trade off

Ashesi University
Brekuso – E/R Ghana
CS456 – Algorithm Analysis & Design
March /2024

What is Space-time trade off? …1/2

- ***Space – time*** trade off is a case where an algorithm or program trades increased space usage with decreased time or vice versa.
- ***Space*** = data storage (ram, hdd, ssd,etc.)
- ***Time*** = time consumed during the computation of a computer task

What is Space-time trade off? ...2/2

- A ***tradeoff*** is a situation where one thing increases, and another thing decreases. i.e
  - Either in less time and more space, or
  - little space and more time
- ***Space –Time*** tradeoff is therefore a way of balancing the amount of time and space required by a computer during the execution of an algorithm.

Space Time Trade offs

Two varieties of algorithms that trades space for time:

Input Enhancement:

Preprocess the input (or its part) to store some infomation to be used later in solving the problem

- counting sorts
- string searching algorithms

Pre-Structuring:

Preprocess the input to make subsequent accessing of its elements easier

- hashing

Short video on Comparison Counting Sort Algorithm

- https://www.youtube.com/watch?v=31aWTP4TGJE

Two type of Counting Sort algorithms

1. Comparison Counting Sort Algorithm

2. Distribution Counting Sort Algorithm

Comparison Counting – Counting Sorts

Let us consider the sorting of the numbers 62, 31, 84, 96, 19, and 47 using comparison-based counting sort

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
		0	1	2	3	4	5

Example of sorting by comparison counting

Comparison Counting Sort Algorithm

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

Time Complexity of Comparison Counting Sort

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i) = \frac{n(n-1)}{2}.$$

Therefore, the time Complexity for the counting sort algorithm is $\Theta(n^2)$

Space complexity for Comparison Counting Sort = O(n), because we have to create an extra array of size n called count[0..n-1]

2. Distribution Counting Sort

- https://www.youtube.com/watch?v=0B33As8jPgo

2. Example of Counting Sort – Distribution Counting

EXAMPLE Consider sorting the array

13	11	12	13	12	12

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

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

$S[0..5]$

0	1	2	3	4	5
			12		
		12			
					13
	12				
11					
				13	
11	12	12	12	13	13

Example of sorting by distribution counting. The distribution values being decremented are shown in bold.

Array values must not be overwritten in the process of sorting

Distribution Counting Sort Algorithm – Counting Sort

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

//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 $(l \le u)$
//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
```

Time Complexity of Distribution Counting Sort

- the initialization of the count array + the loop which performs a prefix sum on the count array takes O(k) time. k = range of the numbers to be sorted
- And other two loops for initialization of the output array takes O(n) time.
- Therefore, the total time complexity for the algorithm is : **O(k)+ O(n)+ O(k)+ O(n) = O(n+k).**
- Space complexity is O(n+k)

String Matching

String Matching Problems

- Considered as being of two forms:
  - Exact string matching;
  - Approximate string matching
- **text** - We have a long string of n characters of some alphabets (Σ)
- **pattern** - string of m characters from the same alphabet

Exact Matching

Given a long string **text** T of length n, and another string **pattern** P of length m, $m \leq n$, the exact string matching problem is to find the positions of all occurrences of P in T.

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

Review: The Brute Force Exact Matching Algorithm

Example

```
T =  x a b x y a b x y a b x z

P =  a b x y a b x z
       a b x y a b x z
         a b x y a b x z
           a b x y a b x z
             a b x y a b x z
               a b x y a b x z
```

Review: Brute Force String Matching Pseudocode

ALGORITHM *BruteForceStringMatch*$(T[0..n-1], P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//        an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//          matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

Time Complexity is **O(nm)**

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

- **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

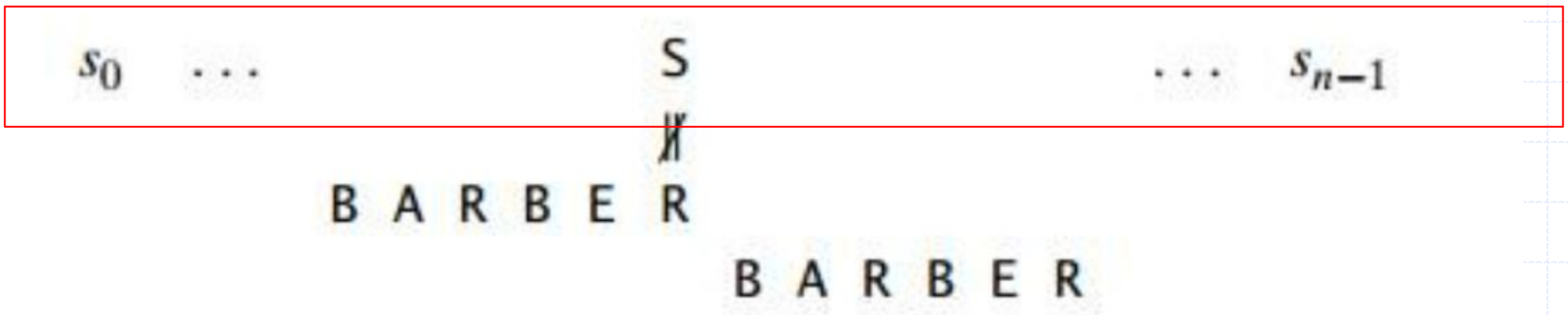
A simplified version of Boyer-Moore algorithm:

- 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* <u>aligned with the last</u>** character in the pattern according to the shift table's entry for *c*

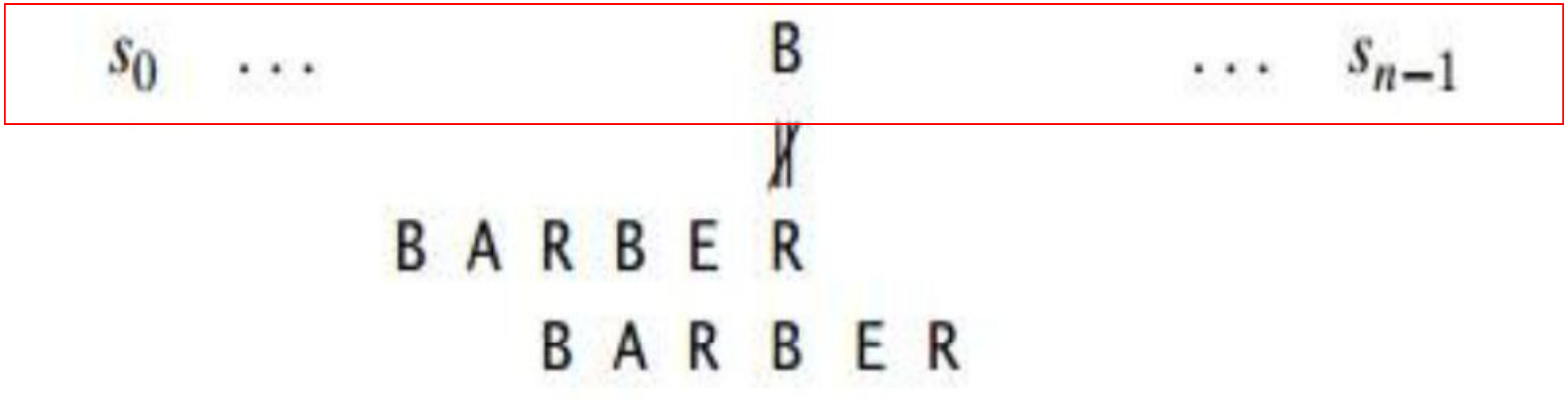
Cases during Shifting

- CASE I :



- If letter in the text ‘S’ does not occur anywhere in the pattern, shift the entire pattern past ‘S’.

Cases during Shifting
Case II :



Case 2 If there are occurrences of character ***c*** in the pattern but it is not the last one there—e.g., ***c*** is letter **B** in our example—the shift should align the rightmost occurrence of ***c*** in the pattern with the ***c*** in the text:

Cases during Shifting - Case III

```
s0  ...        M  E  R              ...  sn-1
               ≠  ‖  ‖
      L  E  A  D  E  R
                        L  E  A  D  E  R
```

If ***c*** happens to be the last character in the pattern but there are no ***c***’s among its other ***m*** − 1 characters—e.g., ***c*** is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern’s length ***m***:

Cases during Shifting -Case IV

s_0 ... A R ... s_{n-1}

$\neq$ $\|$

R E O R D E R

R E O R D E R

if ***c*** happens to be the last character in the pattern and there are other ***c***'s among its first ***m*** − 1 characters— e.g., ***c*** is letter R in our example— the situation is similar to that of Case 2 and the rightmost occurrence of ***c*** among the first ***m*** − 1 characters in the pattern should be aligned with the text's ***c***:

Calculating the Shift Table (the input enhancement)

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

Start by initializing the table entries to the **size** of the pattern. e.g. for a pattern **BARBER,** the size is **6**
So all entries will be initialized to **6** (including , even those that are not present in the pattern now) then we use the above t(c) to update the letters that are present in the pattern.

How to fill the shift table

- *Table[k] = pattern length -1-index*
- 0<=k< (m-1)
- For pattern BARBER the shift table will be

B	**6-0-1 = ~~5~~ 2**
A	**6-1-1 =4**
R	**6-2-1=3**
B	**6-3-1=2**
E	**6-4-1=1**
R	**No value**

Shift Table for pattern BARBER

0	1	2	3	4	5	
B	**A**	**R**	**B**	**E**	**R**	*****
2	**4**	**3**	**2**	**1**	**-**	6

* Means any other character

For repetitive letters we store ONLY the least no
We do not calculate/assign value to the last letter

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
1   B A R B E R             4 B A R B E R
         2  B A R B E R       5   B A R B E R
              3 B A R B E R          6  B A R B E R
```

B	**6-0-1 = ~~5~~ 2**
A	**6-1-1 =4**
R	**6-2-1=3**
B	**6-3-1=2**
E	**6-4-1=1**
R	**No value**

Always compare from RIGHT to LEFT

n means order of shifting

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

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

//Fills the shift table used by Horspool's 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 ($t(c)$)

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*

EndAlg

Algorithm *HorspoolMatching*

```
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]]                // shift appropriately
 return −1
EndAlg
```

Time Complexity of *Horspool Algorithm*

- The worst case time complexity is O(mn), where m = the length of the pattern and n = the length of the text
- But for a random text, the runtime is O(n), which is linear. Horspool alogirthm runs much faster than the brute force algorithm even when they happen to be in the same time complexity.

Short video on Horspool’s Algorithm

1. https://www.youtube.com/watch?v=0-FZyh46zwA&t=19s

- Watch the next video (No. 2, a bit longer) at home

2. https://youtu.be/XTw9_SElm68

In-class discussion 1/4

1. Apply Horspool's algorithm to search for the pattern BAOBAB in the text

BESS_KNEW_ABOUT_THE_BAOBABS_IN_TOWN

2. Consider the problem of searching for genes in DNA sequences using Horspool's algorithm. A DNA sequence is represented by a text on the alphabet {A, C, G, T}, and the gene or gene segment is the pattern. Construct the shift table for the following gene segment of your chromosome 10:

In-class discussion – 2/4

TCCTATTCTT

Apply Horspool's algorithm to locate the above pattern in the following DNA sequence:

TTATAGATCTCGTATTCTTTTATAGATCTCCTATTCTT

3. How many character comparisons will be made by Horspool's algorithm in searching for each of the following patterns in the binary text of 100010010 zeros?

a. 0001 **b.** 1001 **c.** 01010

In-class exercises … 3/4

- 1.Consider the following numbers. Use the counting sort algorithm to sort these numbers. Show all steps.
-

[20, 30, 60, 40, 30, 20, 10, 40, 30, 60, 60, 40, 40]

In-class discussion – …4/4

- Assuming that the set of possible list values is {a, *b*, *c*, *d*}, sort the following list in alphabetical order by the distribution counting algorithm:
  b, c, *d*, *c*, *b*, *a*, *a*, *b*.