Announcement

Homework #5 has been posted in both Blackboard and course website.

Due at: 10:05 am, Thursday, April 7

Chapter 7: Space-Time Tradeoffs

For many problems some extra space really pays off:

Prestructuring

- hashing

Preprocessing (Input enhancement)

- auxiliary tables (shift tables for pattern matching)

Dynamic programming

String Matching

***pattern*: a string of *m* characters to search for**

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

Brute force algorithm:

1. Align pattern at beginning of text
2. moving from left to right, compare each character of pattern to the corresponding character in text until
   - all characters are found to match (successful search); or
   - a mismatch is detected
3. while pattern is not found and the text is not yet exhausted, realign pattern one position to the right and repeat step 2.

What is the complexity of the brute-force string matching?

Worst case: $m(n-m+1) \in \boldsymbol{O(nm)}$

String Searching - History

1970: Cook shows (using finite-state machines) that problem can be solved in time proportional to *n*+*m*

1976 Knuth-Morris-Pratt find algorithm based on Cook's idea; when a mismatch occurs, the word itself has sufficient information to determine where the next match could begin,

At about the same time Boyer and Moore find an algorithm that examines only a fraction of the text in most cases (by comparing characters in pattern and text from right to left, instead of left to right)

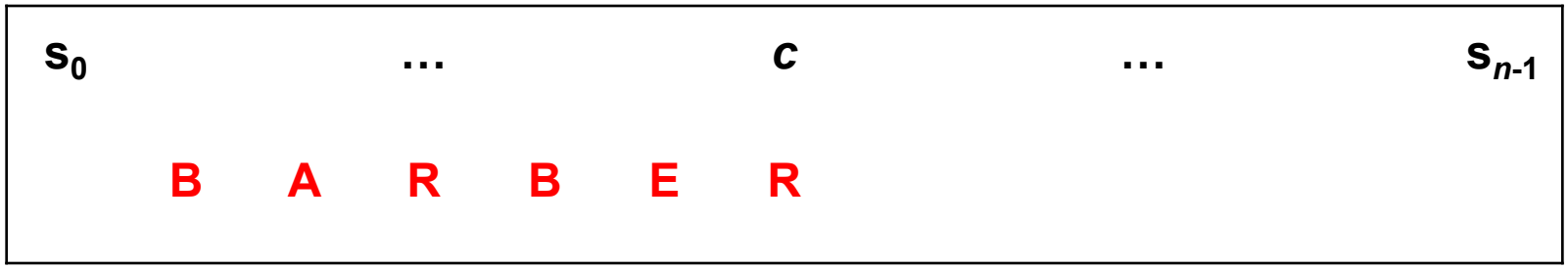
Horspool's Algorithm

A simplified version of Boyer-Moore algorithm that retains key insights:

- compare pattern characters to text from right to left
- given a pattern, create a shift table that determines how much to shift the pattern when a mismatch occurs (*input enhancement*)

Consider the Problem

Search pattern BARBER in some text



Compare the pattern in the current text position from the right to the left

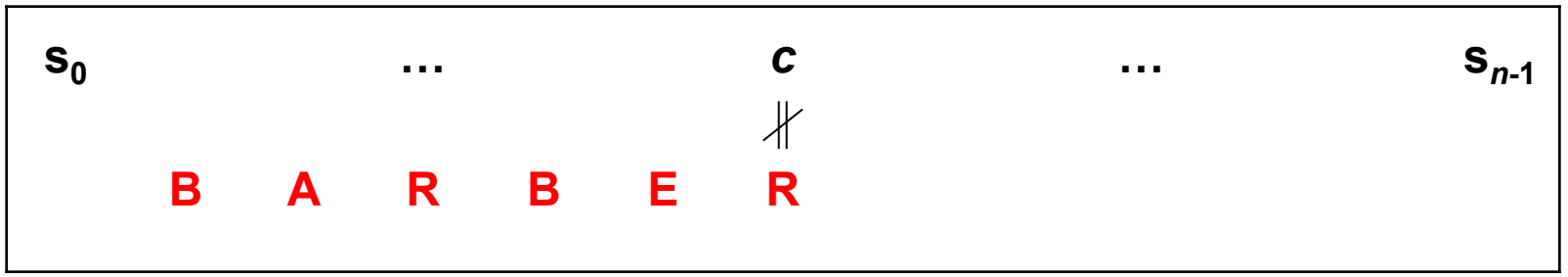
If the whole match is found, done.

Otherwise, decide the shift distance of the pattern (move it to the right)

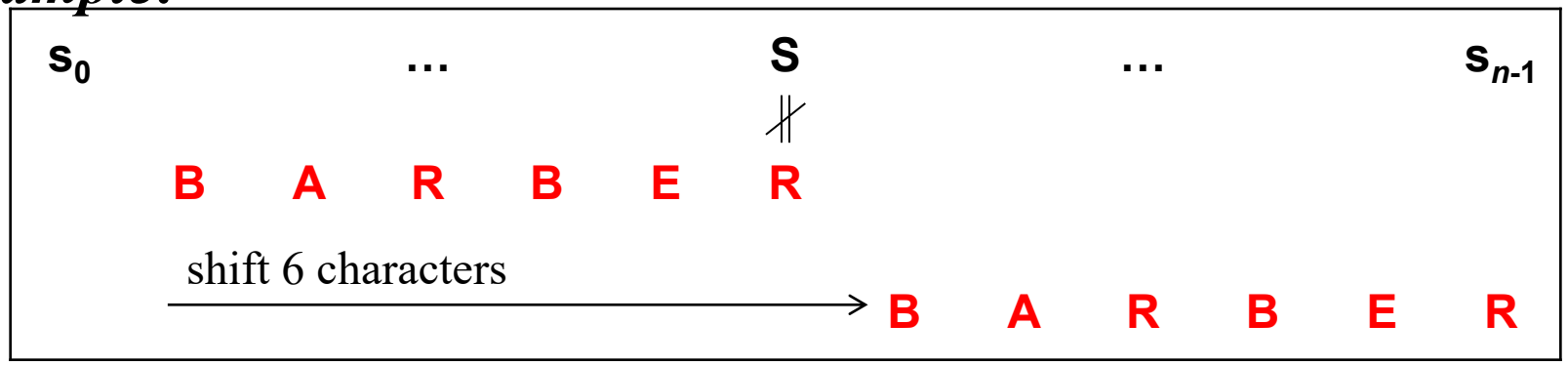
There are four cases!

Shift Distance -- Case 1:

There is no '*c*' in the pattern. Shift by the *m* – the length of the pattern

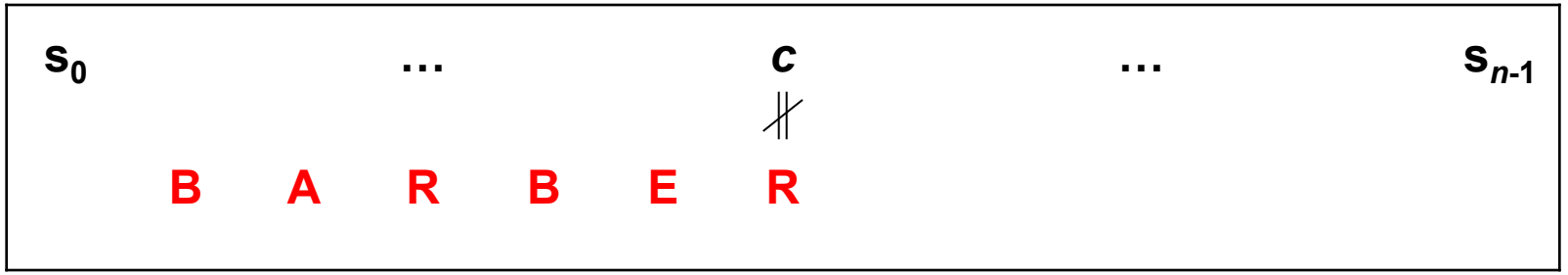


Example:

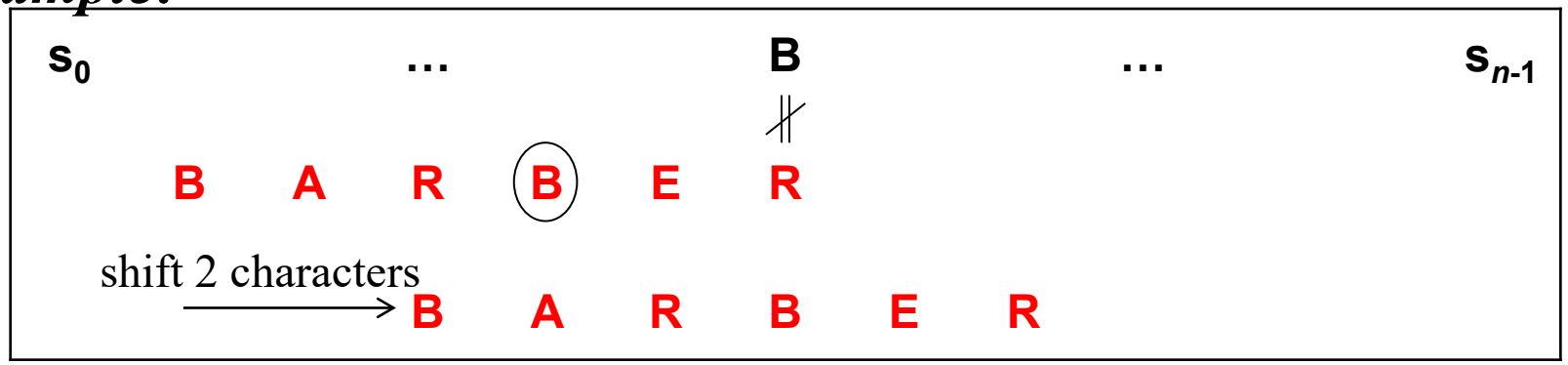


Shift Distance -- Case 2:

There are occurrence of '*c*' in the pattern, but it is not the last one. Shift should align the rightmost occurrence of the '*c*' in the pattern

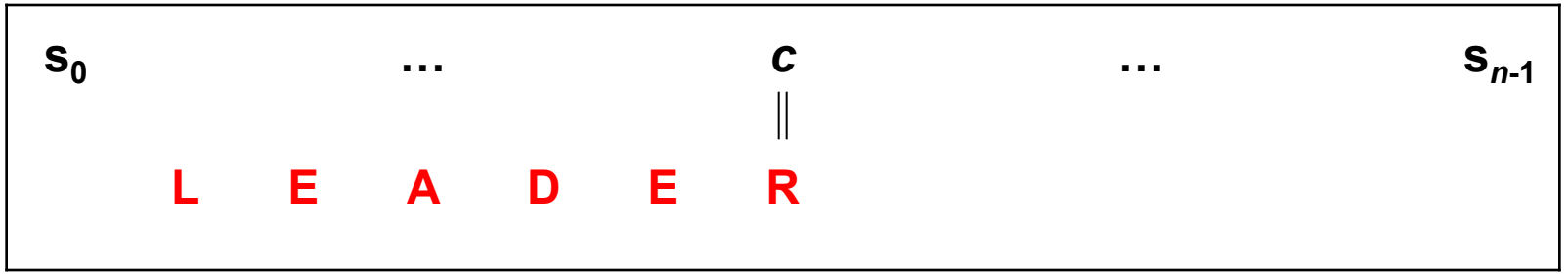


Example:

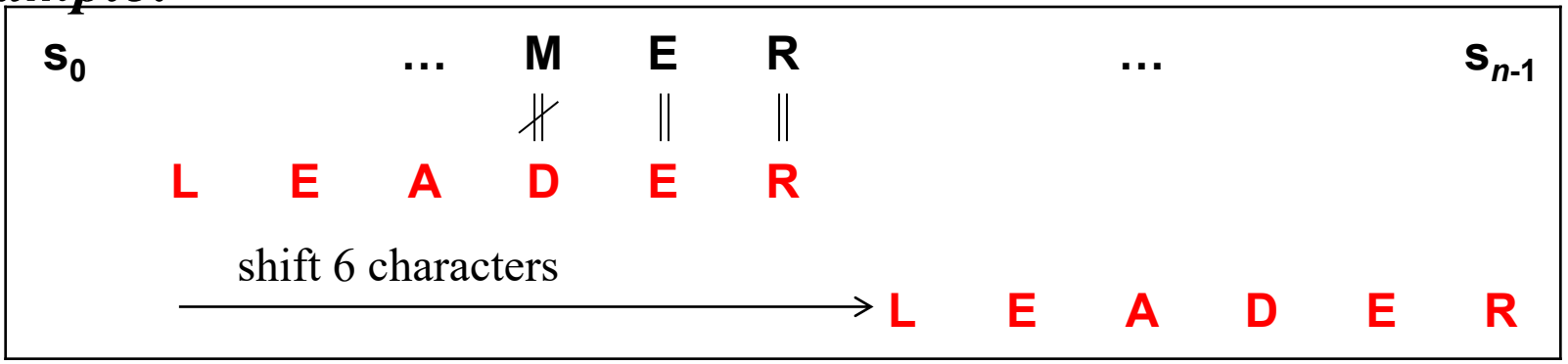


Shift Distance -- Case 3:

'*c*' matches the last character in the pattern, but no '*c*' among the other *m*-1 characters. Follow Case 1 and shift by *m*

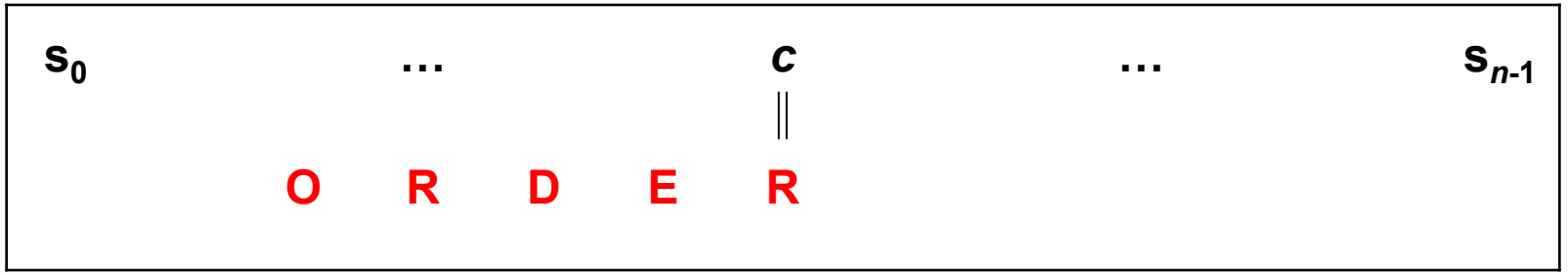


Example:

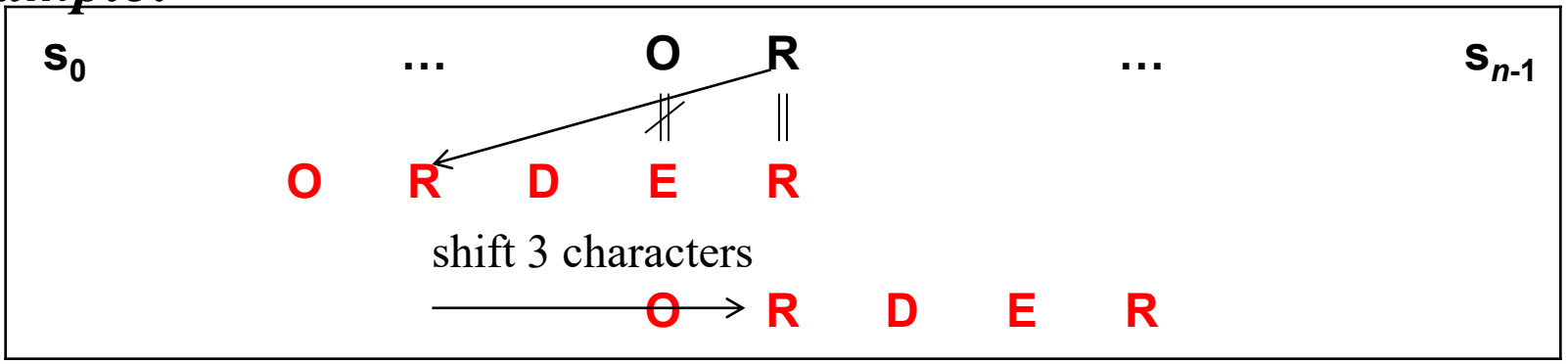


Shift Distance -- Case 4:

'*c*' matches the last character in the pattern, and there are other '*c*'s among the other *m*-1 characters. Follow Case 2.



Example:



We can precompute the shift distance for every possible character ‘c’ (given a pattern)

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

Case 1&3

Case 2&4

Shift Table for the pattern “BARBER”

c	A	B	C	D	E	F	…	R	…	Z	_
t(c)	4	2	6	6	1	6	6	3	6	6	6

Create a Shift Table

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)
initialize all the elements of *Table* with m
for $j \leftarrow 0$ **to** $m-2$ **do** $Table[P[j]] \leftarrow m-1-j$
return *Table*

Shift Table for the pattern "BARBER"

c	A	B	C	D	E	F	…	R	…	Z	_
t(*c*)	4	2	6	6	1	6	6	3	6	6	6

Horspool's Algorithm

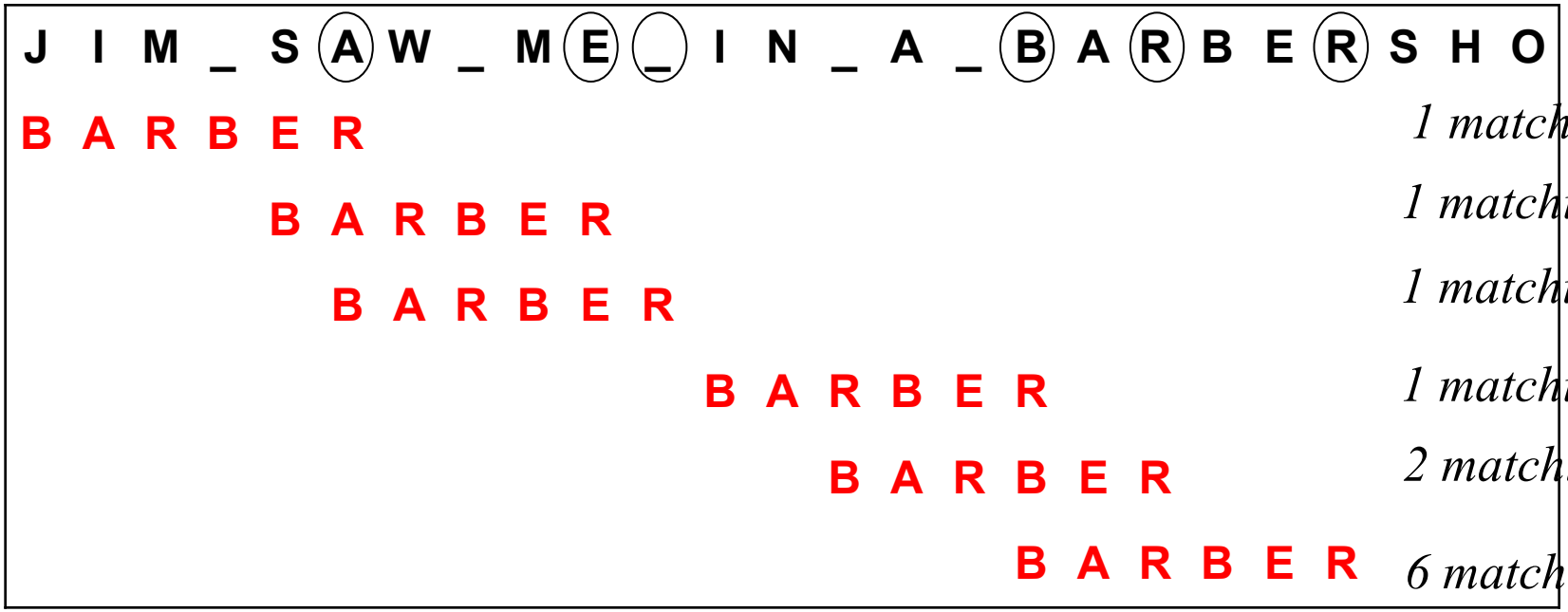
ALGORITHM *HorspoolMatching*$(P[0..m-1], T[0..n-1])$

```
//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
```

Example

Example: find the pattern BARBER from the following text

c	A	B	C	D	E	F	…	R	…	Z	_
t(*c*)	4	2	6	6	1	6	6	3	6	6	6

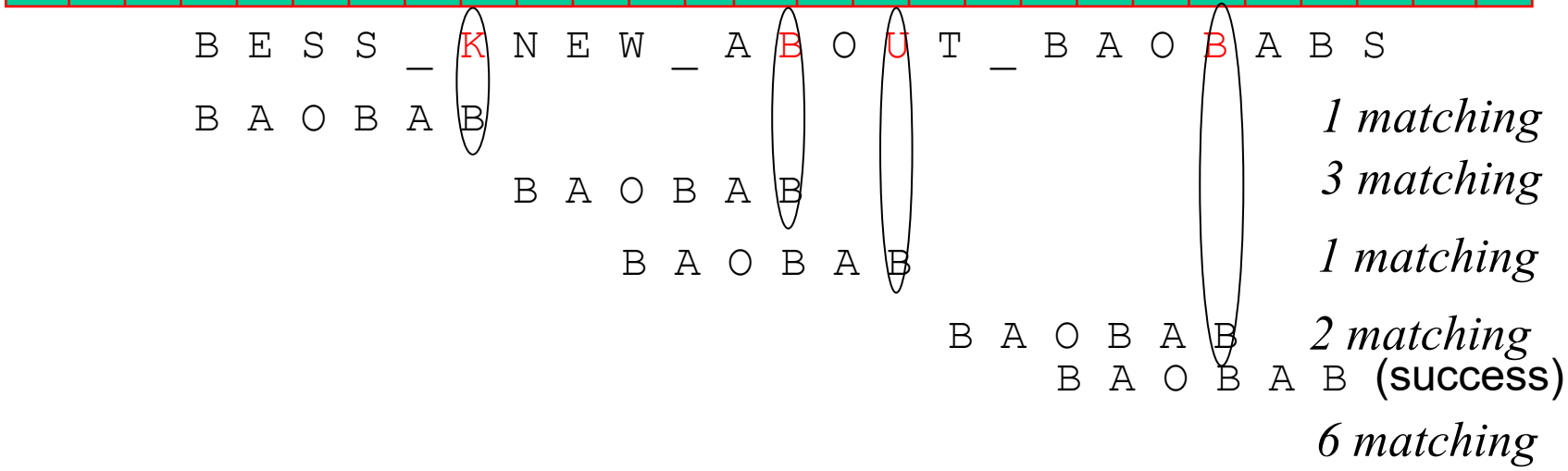


Total: 12 matching operations

Another Example: Pattern = B A O B A B

Shift table

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	_
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



Total: 13 matching operations

Algorithm Efficiency

The worst-case complexity is $\Theta(nm)$

In average, it is $\Theta(n)$

It is usually much faster than the brute-force algorithm

A simple exercise: Create the shift table of 26 letters and space for the pattern BAOBABCD

Boyer-Moore algorithm

Based on two ideas:

- compare 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

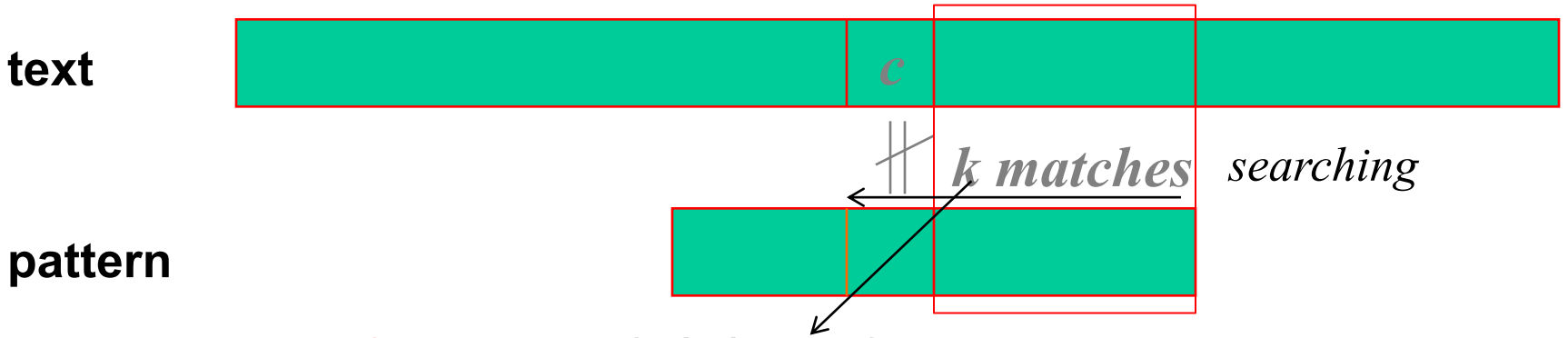
The worst-case efficiency of Boyer-Moore algorithm is linear.

Bad-symbol Shift in Boyer-Moore Algorithm

Build a bad-symbol shift table as in the Horspool's algorithm.

If the rightmost character of the pattern doesn't match, BM algorithm acts as Horspool's (Case 1 and 2)

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

Shift in the bad-symbol shift table

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(k=1) = 4

Case 1: if there is no such occurrence – unknown prefix, match the longest part of the k-character suffix with corresponding prefix;

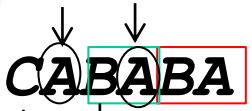
Case 2: if there are no such suffix-prefix matches, $d_2(k) = m$

Good-suffix shift in Boyer-Moore algorithm

Example: CABABA

$d_2(1) = 4$, 

Choose the one with different prefix

$d_2(2) = 6$, Case 2, 

Same prefix

$d_2(3) = 2$, 

Choose the one with different prefix

$d_2(4) = 6$, Case 2, 

Cannot find BABA

$d_2(5) = 6$, Case 2, 

Cannot find ABABA

Good-suffix shift in Boyer-Moore algorithm

Example: BAOBAB

$d_2(1) = 2$, BAOBAB

$d_2(2) = 5$, Case 1,

$d_2(3) = 5$, Case 1,

$d_2(4) = 5$, Case 1,

$d_2(5) = 5$, Case 1,

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 Construct the bad-symbol shift table (the one as Horspool's)

Step 2 Construct 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 character match, follow the Case 1&2 as the Horspool's algorithm

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 Algorithm

Bad-symbol shift table

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	_
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

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

1 matching

$d_1 = t_1(\text{K}) = 6$ B A O B A B

3 matching

good-symbol shift table

k	pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

$k = 2$, $d_1 = t_1(\_) - k = 4$,

$d_2(2) = 5$, d = 5

B A O B A B

2 matching

$k = 1$, $d_1 = t_1(\_) - k = 5$

$d_2(1) = 2$, d=5

B A O B A B (success)

6 matching

Total: 12 matching operations

Compare BM with Horpool's

Pattern: ABBB

Bad-symbol shift table

A	B	C	…	Z	_
3	1	4	4	4	4

Using Horpool's

Text	C	B	B	B	B	B	A	B	B	B
4 comparisons	**A**	**B**	**B**	**B**						
4 comparisons		A	B	B	B					
4 comparisons			A	B	B	B				
1 comparison				A	B	B	B			
4 comparisons							A	B	B	B

A total of 17 comparisons

Compare BM with Horpool's

Pattern: ABBB

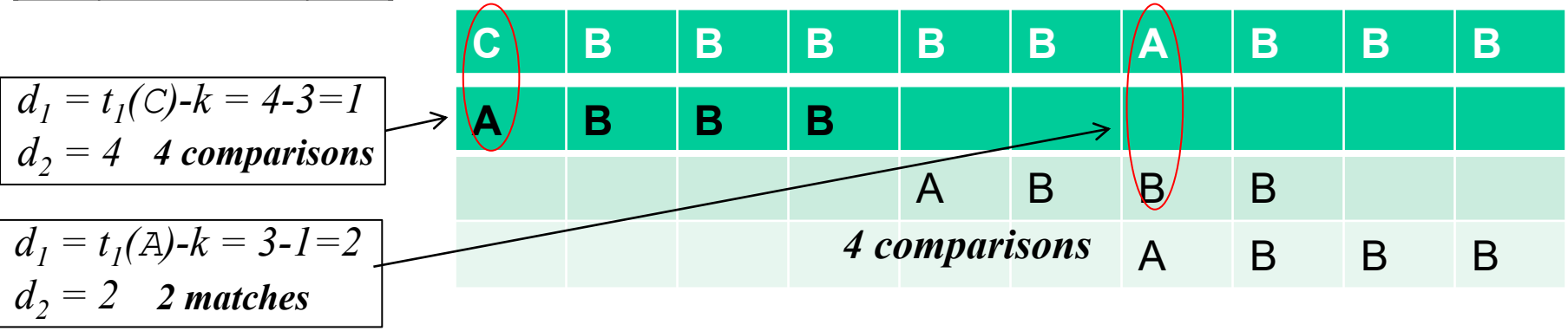
Bad-symbol shift table

A	B	C	…	Z	_
3	1	4	4	4	4

good-symbol shift table

k	pattern	d_2
1	ABB**B**	2
2	AB**BB**	4
3	A**BBB**	4

The worst-case efficiency of Boyer-Moore algorithm is linear!



A total of 10 comparisons

Space and Time Tradeoffs: Hashing

A very efficient method for implementing a *dictionary, i.e.,* a set with the operations:

- insert
- search
- delete

Each entry has many fields, at least one of which is for identification - unique

Applications:

- databases
- symbol tables

Hash tables and hash functions

Hash table: **an array with indices that correspond to buckets**

Hash function: **determines the bucket for each record**

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

$h(k) = k \bmod m$

(k is a key and m is the number of buckets)

- if m = 1000, where is record with SSN= 123-45-6789 stored?

Desirable hash functions:

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

Collisions

If *h*(*k1*) = *h*(*k*2) then there is a *collision.*

Good hash functions result in fewer collisions.

Collisions can never be completely eliminated.

Two types handle collisions differently:

- *Open hashing*
  - **bucket points to linked list of all keys hashing to it.**
- *Closed hashing*
  - **in case of collision, find another bucket for one of the keys (need** *Collision resolution strategy*)
    - *linear probing:* **use next bucket**
    - *double hashing:* **use second hash function to compute increment**

Example of Open Hashing

Store student record into 10 bucket using hashing function

$h(SSN) = SSN \bmod 10$

xxx-xx-3333

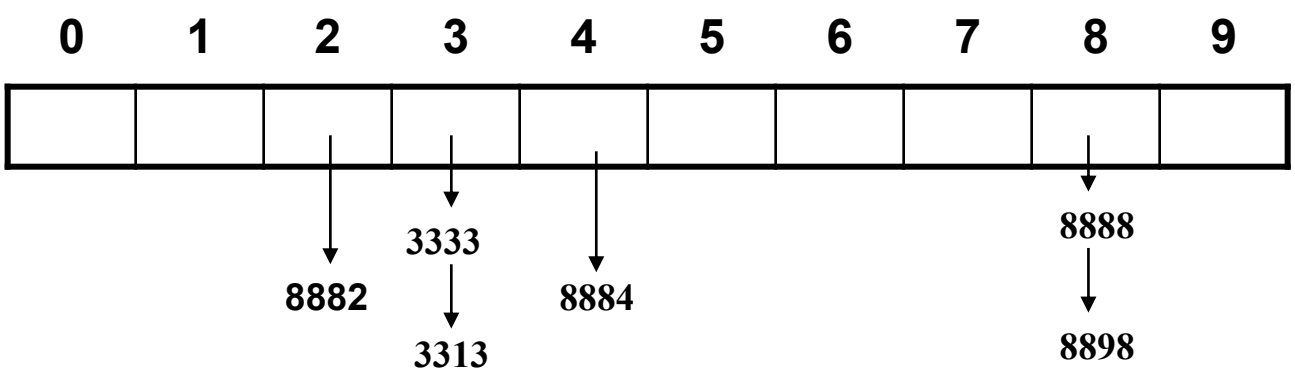
xxx-xx-8888

xxx-xx-3313

xxx-xx-8882

xxx-xx-8884

xxx-xx-8898



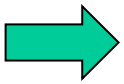
Efficiency of searching a key depends on the length of the linked list

Open hashing

If hash function distributes keys uniformly, average length of linked list will be $\alpha = n/m$ *(load factor)*

Average number of successful search $\approx 1+\alpha/2$

Average number of unsuccessful search $=\alpha$



Carefully select m

Open hashing still works if $n>m$.

Insertion: append to the end $\Theta(1)$

Deletion: search the key and deleted $\Theta(\alpha)$

Closed Hashing (Linear Probing)

At the most one key per bucket and does not work if $n > m$.

h(*SSN*)=*SSN* mod 10

When an collision occurs, use the next available bucket (wrapped to the beginning when reaching the end) to store the new key.

xxx-xx-3333

xxx-xx-8888

xxx-xx-8883

xxx-xx-8882

xxx-xx-8884

xxx-xx-8898

xxx-xx-8828

0	1	2	3	4	5	6	7	8	9
8828		8882	3333	8883	8884			8888	8898

Closed Hashing (Linear Probing)

Search for a given key

Keep searching until either find a match or find an empty bucket.

***h*(*SSN*)=*SSN* mod 10**

xxx-xx-3333

xxx-xx-8888

xxx-xx-8883

xxx-xx-8882

xxx-xx-8884

xxx-xx-8898

xxx-xx-8828

0	*1*	2	3	4	5	6	7	8	9
8828		8882	3333	8883	8884			8888	8898

Closed Hashing (Linear Probing)

Deletions are *not* straightforward.

- lazy deletion: mark the previous occupied bucket with a special symbol

Number of probes (matching operations) to insert/find/delete a key depends on load factor α = *n*/*m* (hash table density)

- successful search: (½) (1+ 1/(1- α))
- unsuccessful search: (½) (1+ 1/(1- α)²)

As the table gets filled (α approaches 1), number of probes 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