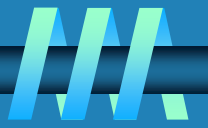
Space-for-time tradeoffs

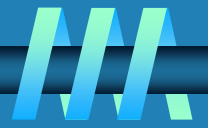


Two varieties of space-for-time algorithms:

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

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

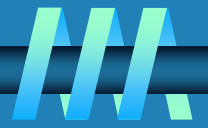
Sorting by Counting



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

Sorting by Counting



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

FIGURE 7.1 Example of sorting by comparison counting.

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

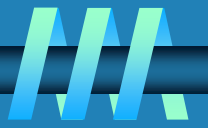
Brute force algorithm

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

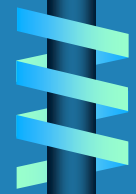
Step 3 While a mismatch is detected and the text is not yet exhausted, realign pattern one position to the right and repeat Step 2

String searching by preprocessing



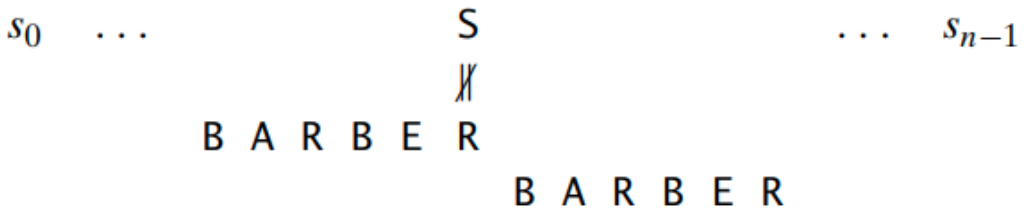
Several string searching algorithms are based on the input enhancement idea of preprocessing the pattern

- **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**

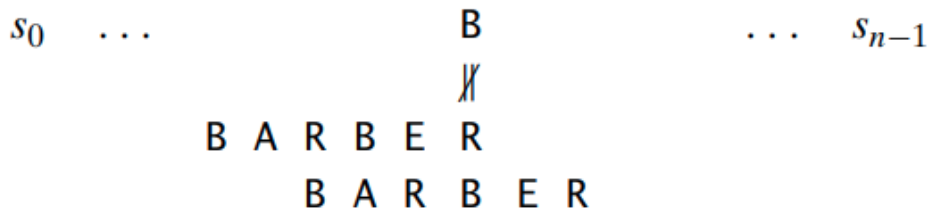


How far to shift?

Case 1 If there are no c's in the pattern—e.g., c is letter S in our example—we can safely shift the pattern by its entire length (if we shift less, some character of the pattern would be aligned against the text's character c that is known not to be in the pattern):

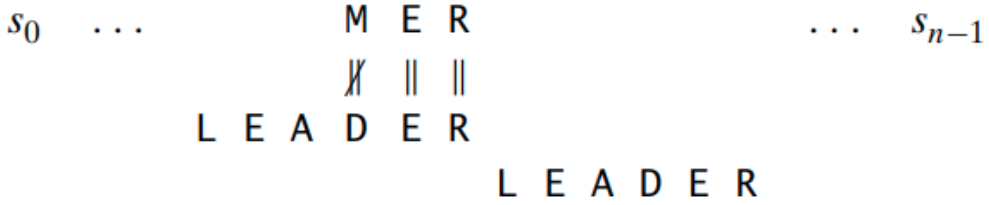


Case 2 If there are occurrences of character c in the pattern but it is not the last one there—e.g., c is letter B in our example—the shift should align the rightmost occurrence of c in the pattern with the c in the text:

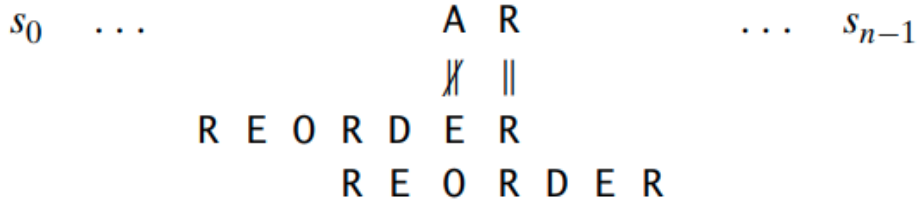


How far to shift?

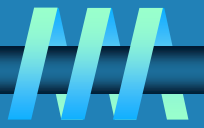
Case 3 If c happens to be the last character in the pattern but there are no c's among its other $m - 1$ characters—e.g., c is letter R in our example—the situation is similar to that of Case 1 and the pattern should be shifted by the entire pattern's length m:



Case 4 Finally, 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:



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 character in the pattern according to the shift table's entry for *c***

Shift table

- Shift sizes can be precomputed by the formula

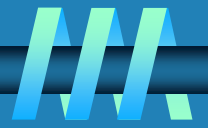
$$t(c) = \begin{cases} \text{the pattern's length } m, \\ \text{if } c \text{ is not among the first } m-1 \text{ characters of the pattern;} \\ \\ \text{the distance from the rightmost } c \text{ among the first } m-1 \text{ characters} \\ \text{of the pattern to its last character, otherwise.} \end{cases}$$

by scanning pattern before search begins and stored in a table called *shift table*

- 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

Shift table



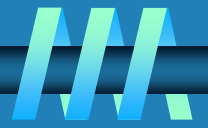
ALGORITHM *ShiftTable*$(P[0..m-1])$

//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)
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*

search pattern : **B A O B A B**

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

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 of Horspool's 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

```
//Implements Horspool's algorithm for string matchi
//Input: Pattern P[0..m − 1] and text T[0..n − 1]
//Output: The index of the left end of the first matchi
//        or −1 if there are no matches
ShiftTable(P[0..m − 1])        //generate Table of shif
i ← m − 1                      //position of the pattern
while i ≤ n − 1 do
    k ← 0                      //number of matched ch
    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
```

i=5 i=11 i=13

text: BARD LOVED BANANAS

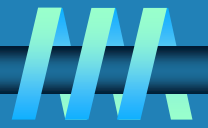
pattern: BAOBAB

BAOBAB

BAOBAB

BAOBAB (unsuccessful search)

Example of Horspool's alg. application



EXAMPLE As an example of a complete application of Horspool's algorithm, consider searching for the pattern BARBER in a text that comprises English letters and spaces (denoted by underscores). The shift table, as we mentioned, is filled as follows:

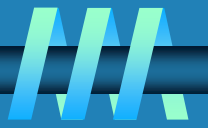
character c	A	B	C	D	E	F	. . .	R	. . .	Z	_
shift $t(c)$	4	2	6	6	1	6	6	3	6	6	6

The actual search in a particular text proceeds as follows:

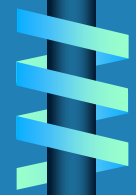
```
J I M _ S A W _ M E _ I N _ A _ B A R B E R S H O P
B A R B E R               B A R B E R
        B A R B E R             B A R B E R
          B A R B E R                   B A R B E R
```

■

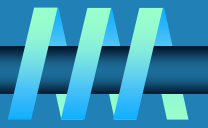
Hashing



- 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*,

$$h: K \rightarrow \text{location (cell) in the hash table}$$

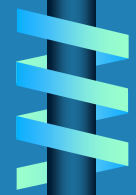
Example: student records, key = SSN. Hash function:

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

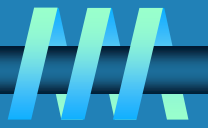
If $m = 1000$, where is record with SSN= 314159265 stored?

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 (*birthday paradox*)

- 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

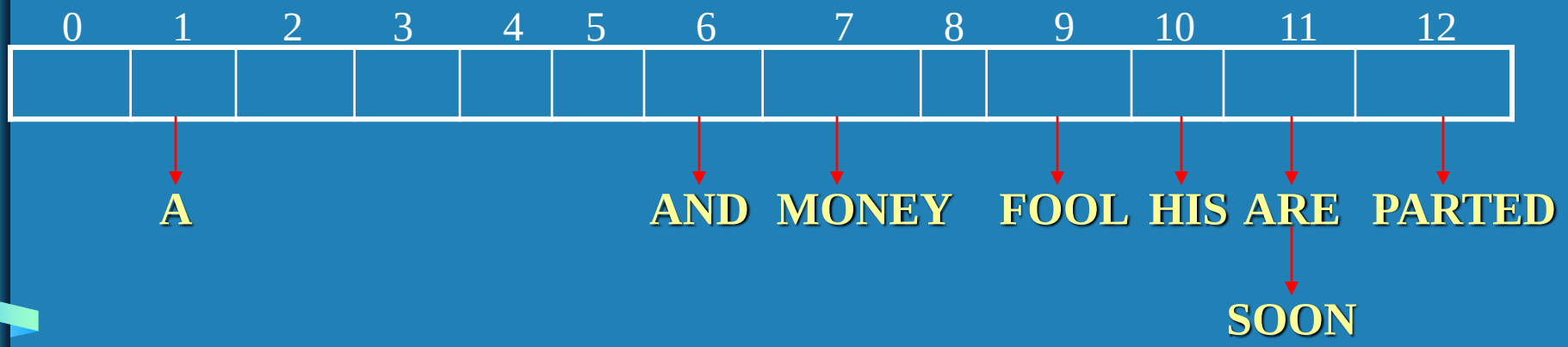
Open hashing (Separate chaining)

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

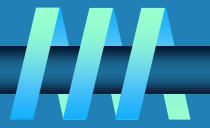
***h*(*K*) = sum of *K* 's letters' positions in the alphabet MOD 13**

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
h(*K*)	1	9	6	10	7	11	11	12



Search for KID

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*.

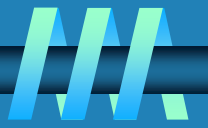
- Average number of probes in successful, S, and unsuccessful searches, U:

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

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

- Open hashing still works if $n > m$

Closed hashing (Open addressing)



Keys are stored <u>inside</u> 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:

  $S = (1/2)\,(1 + 1/(1-\alpha))$ and $U = (1/2)\,(1 + 1/(1-\alpha)^2)$
- 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