Algorithmics I

Section 2 – Strings and text algorithms

Dr. Gethin Norman

School of Computing Science
University of Glasgow

gethin.norman@glasgow.ac.uk

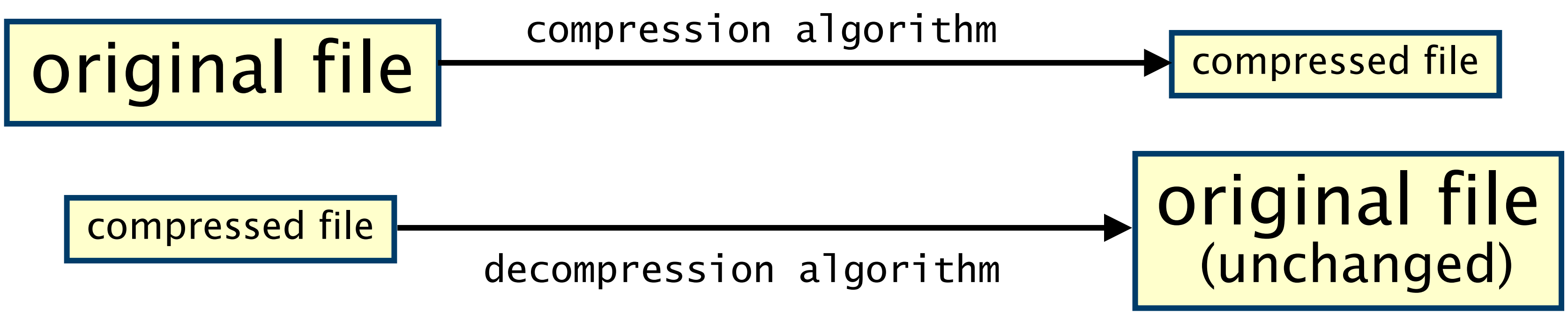
Text compression

A special case of data compression

– saves disk space and transmission time

Text compression must be lossless

– i.e. the original must be recoverable without error



Some other forms of compression can afford to be lossy

– e.g. for pictures, sound, etc. (not considered here)

Text compression

Examples of text compression

- `compress`, `gzip` in Unix, `ZIP` utilities for Windows, …
- two main approaches statistical and dictionary

Compression ratio: **`x/y`**

- `x` is the size of compressed file and `y` is the size of original file
- e.g. measured in B, KB, MB, …
- compressing a `10MB` file to `2MB` would yield a compression ratio of `2/10=0.2`

Percentage space saved: **`(1 - "compression ratio")` × 100%**

- space saved expressed as a percentage of the original file size
- compressing a `10MB` file to `2MB` yields a percentage space savings of `80%`

Space savings in the range **40% - 60%** are typical

- obviously the higher the saving the better the compression

Text compression – Huffman encoding

The classical statistical method

- now mostly superseded in practice by more effective dictionary methods
- fixed (ASCII) code replaced by variable length code for each character
- every character is represented by a unique codeword (bit string)
- frequently occurring characters are represented by shorter codewords

The code has the prefix property

- no codeword is a prefix of another (gives unambiguous decompression)

Based on a Huffman tree (a proper binary tree)

- each character is represented by a leaf node
- codeword for a character is given by the path from the root to the appropriate leaf (`left`=0 and `right`=1)
- the prefix property follows from this

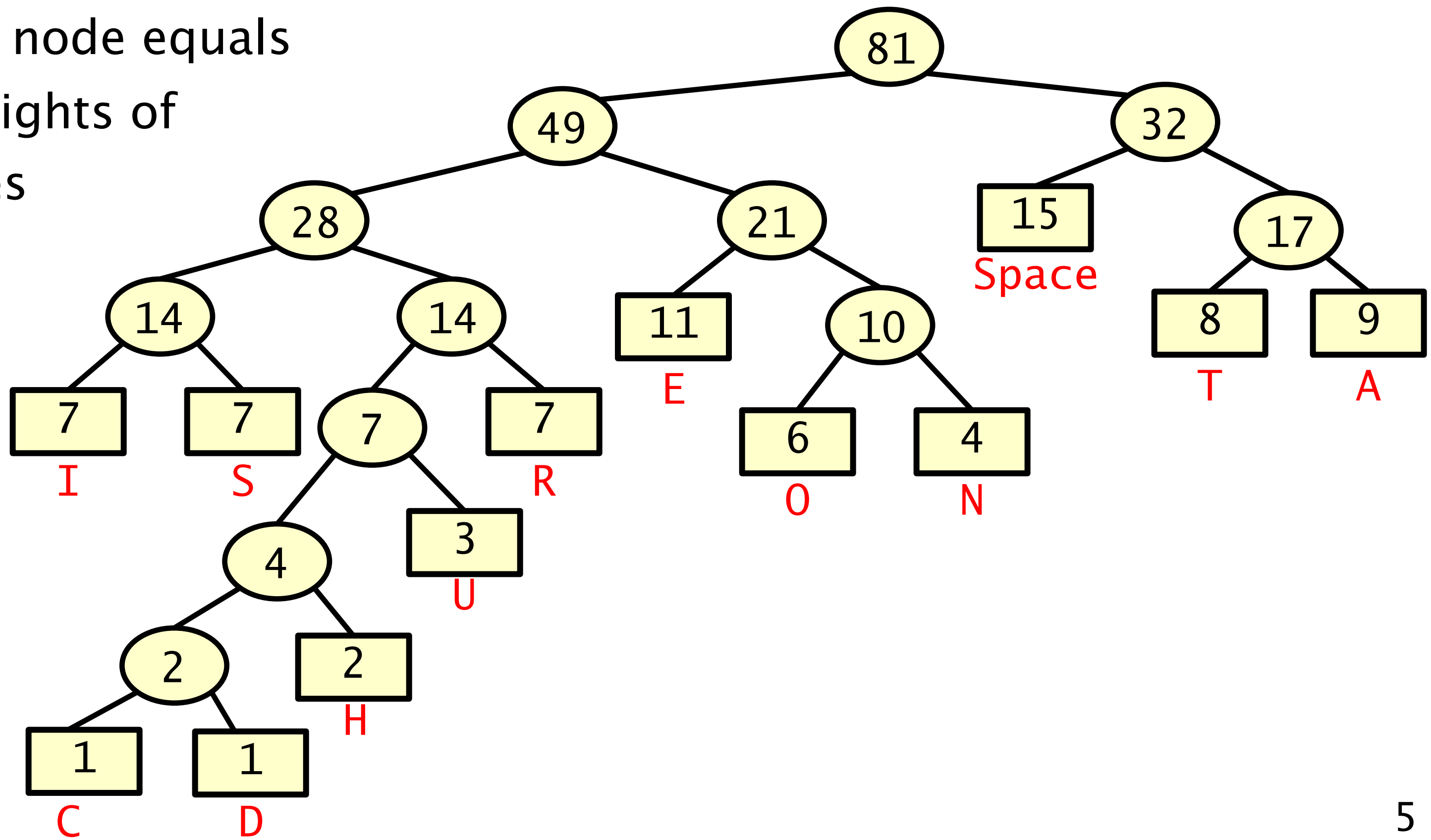
Huffman tree construction – Example

Character frequencies:

Space	E	A	T	I	S	R	O	N	U	H	C	D
15	11	9	8	7	7	7	6	4	3	2	1	1

Next, while there is more than one parentless node

- add new parent to nodes of smallest weight
- weight of new node equals sum of the weights of the child nodes



Huffman tree construction – Pseudocode

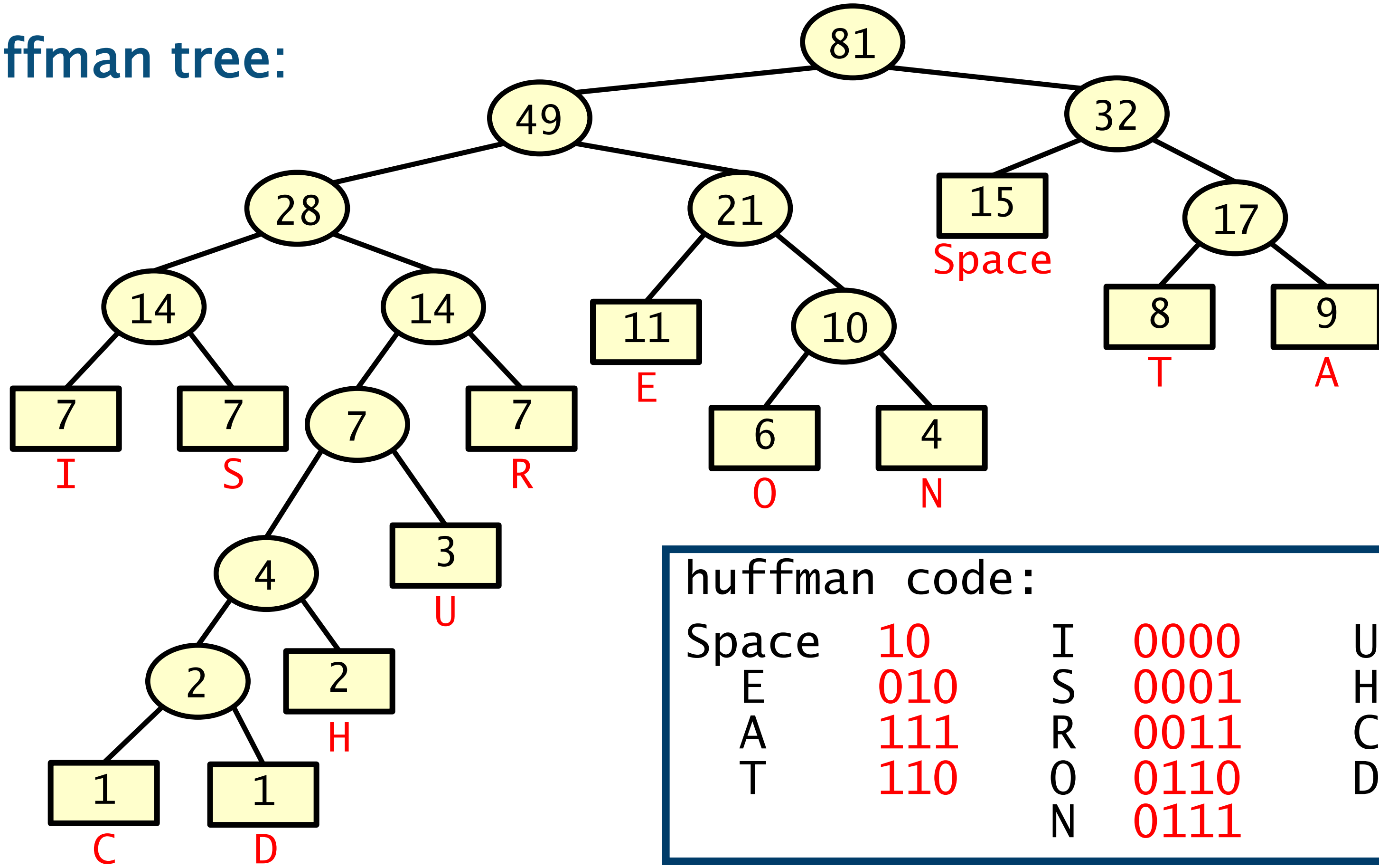
```
// set up the leaf nodes
for (each distinct character c occurring in the text){
  make a new parentless node n;
  int f = frequency count for c;
  n.setWeight(f); // weight equals the frequency
  n.setCharacter(c); // set character value
  // leaf so no children
  n.setLeftChild(null);
  n.setRightChild(null);
}
// construct the branch nodes and links
while (no. of parentless nodes > 1){
  make a new parentless node z; // new node
  x, y = 2 parentless nodes of minimum weight; // its children
  z.setLeftChild(x); // set x to be the left child of new node
  z.setRightChild(y); // set y to be the right child of new node
  int w = x.getWeight()+y.getWeight(); // calculate weight of node
  z.setWeight(w); // set the weight of the new node
}
// the final node z is root of Huffman tree
```

Huffman code – Example

Character frequencies:

Space	E	A	T	I	S	R	O	N	U	H	C	D
15	11	9	8	7	7	7	6	4	3	2	1	1

Huffman tree:



huffman code:

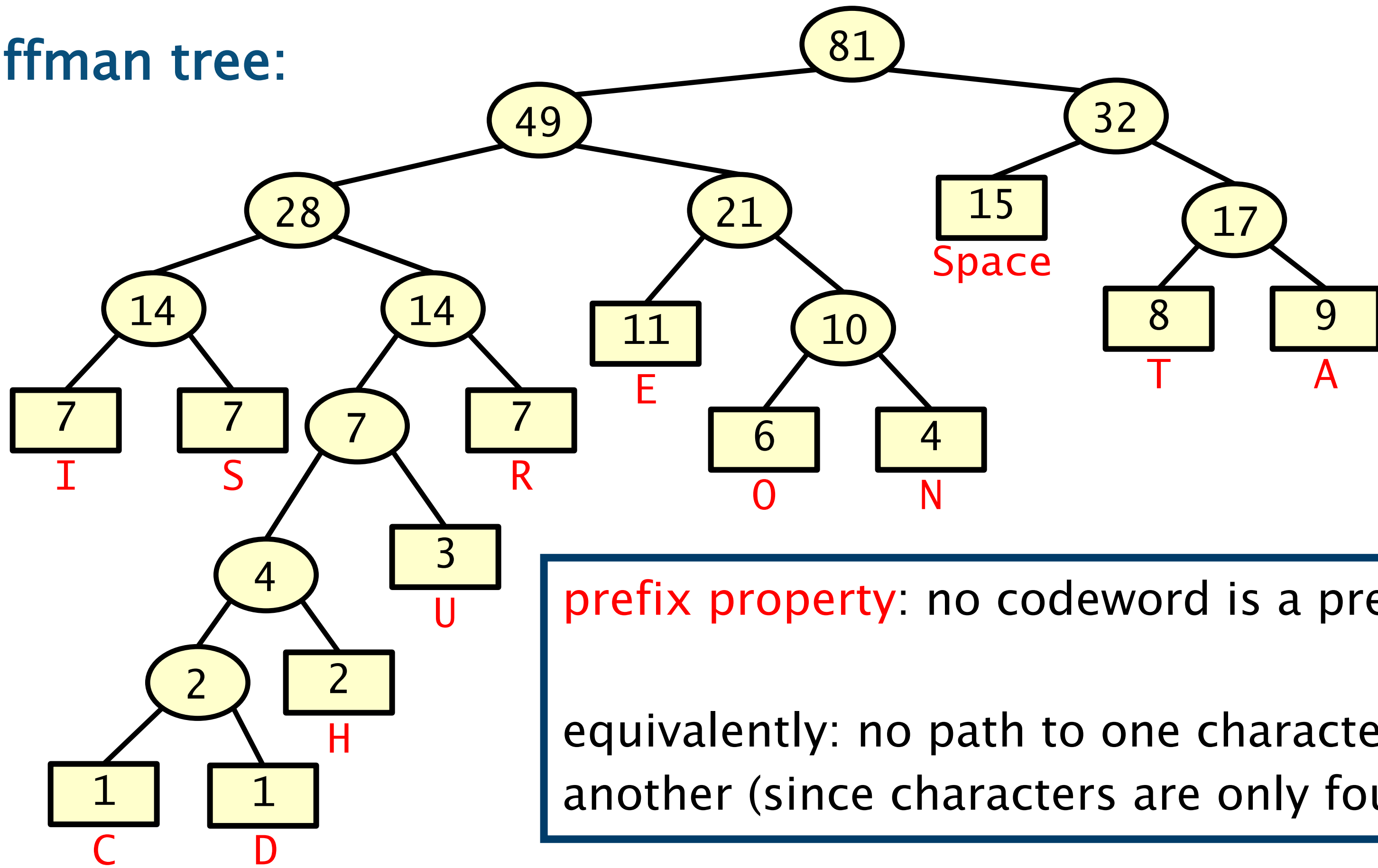
Space	10	I	0000	U	00101
E	010	S	0001	H	001001
A	111	R	0011	C	0010000
T	110	O	0110	D	0010001
		N	0111		

Huffman code – Example

Character frequencies:

Space	E	A	T	I	S	R	O	N	U	H	C	D
15	11	9	8	7	7	7	6	4	3	2	1	1

Huffman tree:



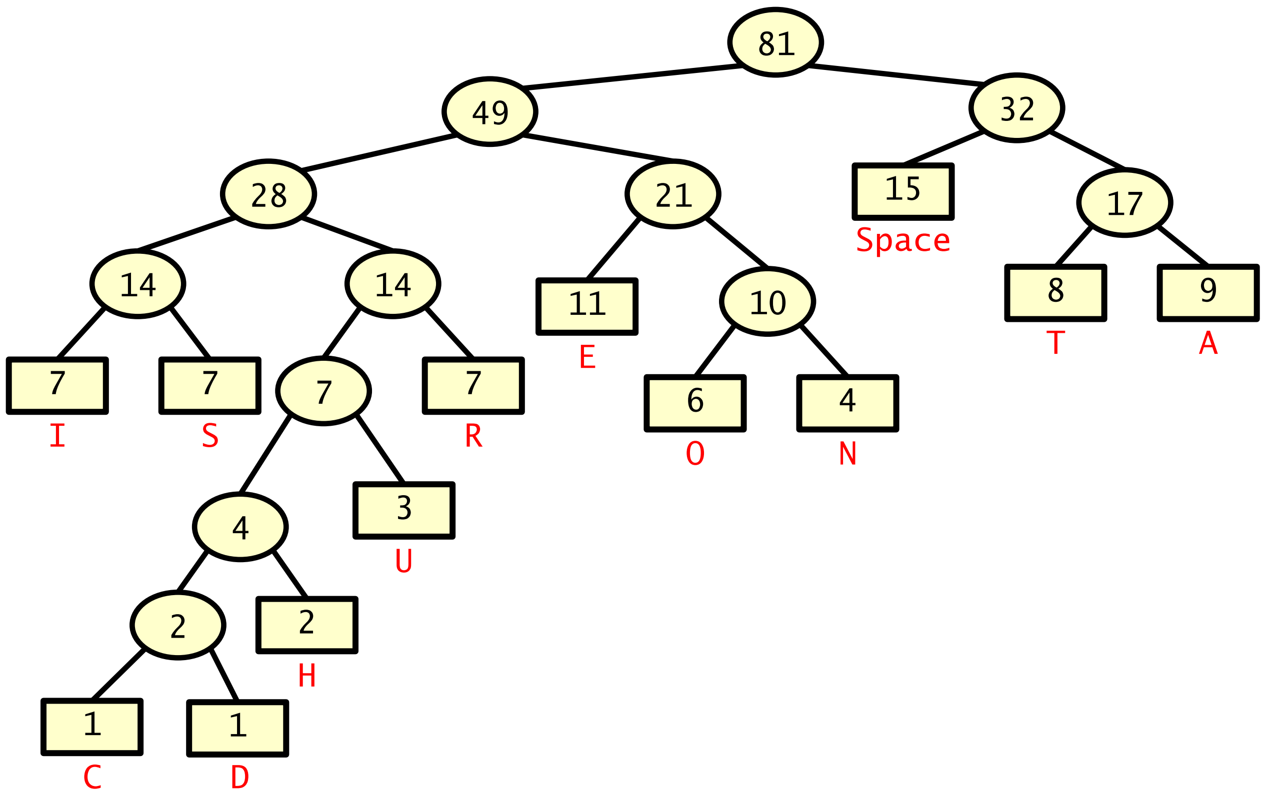
prefix property: no codeword is a prefix of another

equivalently: no path to one character is a prefix of another (since characters are only found at leaves)

Huffman encoding – Optimality

Weighted path length (WPL) of a tree **T**

- Σ (weight)$\times$(distance from root) where sum is over all leaf nodes
- for the example tree: WPL equals: $7\times4 + 7\times4 + 1\times7 + 1\times7 + 2\times6 + 3\times5 + 7\times4 + 11\times3 + 6\times4 + 4\times4 + 15\times2 + 8\times3 + 9\times3 = 279$



Huffman encoding – Optimality

Weighted path length (WPL) of a tree T

- `Σ (weight)×(distance from root)` where sum is over all leaf nodes
- for the example tree: WPL equals: `7×4 + 7×4 + 1×7 + 1×7 + 2×6 + 3×5 + 7×4 + 11×3 + 6×4 + 4×4 + 15×2 + 8×3 + 9×3 = 279`

Huffman tree has minimum WPL over all binary trees with the given leaf weights

- Huffman tree need not be unique (e.g. nodes>2 with min weight)
- however all Huffman trees for a given set of frequencies have same `WPL`
- so what?
- weighted path length (WPL) is the number of bits in compressed file
  - `bits = sum over chars (frequency of char × code length of char)`
- so a Huffman tree minimises this number
- hence Huffman coding is optimal, for all possible codes built in this way

Huffman encoding – Algorithmic requirements

Building the Huffman tree

- if the text length equals `n` and there are `m` distinct chars in text
- `O(n)` time to find the frequencies
- `O(mlog m)` time to construct the code, for example using a (min-) heap to store the parentless nodes and their weights
  - initially build a heap where nodes correspond to the `m` characters labelled by their frequencies, therefore takes `O(m)` time to build the heap
  - one iteration takes `O(log m)` time:
    - find and remove (`O(log m)`) two minimum weights
    - then insert (`O(log m)`) new weight (sum of minimum weights found)
  - and there are `m-1` iterations before the heap is empty
    - each iteration decreases the size of the heap by `1`
- so `O(n + mlog m)` overall
- in fact, `m` is essentially a constant, so it is really `O(n)`

Huffman encoding – Algorithmic requirements

Compression & decompression are both **O(n)** time

- assuming m is constant

Compression uses a code table (an array of codes, indexed by char)

- O(mlog m) to build the table:
  - m characters so m paths of length O(log m)
- O(n) to compress: n characters in the text so n lookups in the array O(1)
- so O(nlog m) + O(n) overall

Decompression uses the tree directly (repeatedly trace paths in tree)

- O(nlog m) as n characters so n paths of length O(log m)

Huffman encoding – Algorithmic requirements

Problem: some representation of the Huffman tree must be stored with the compressed file

- otherwise decompression would be impossible

Alternatives

- use a fixed set of frequencies based on typical values for text
  - but this will usually reduce the compression ratio
- use adaptive Huffman coding: the (same) tree is built and adapted by the compressor and by the decompressor as characters are encoded/decoded
  - this slows down compression and decompression (but not by much if done in a clever way)

LZW compression

A popular dictionary-based method

- the basis of `compress` and `gzip` in Unix also used in gif and tiff formats
- due to Lempel, Ziv and Welch
- algorithm was under patented to Unisys (but patent now expired)

The dictionary is a collection of strings

- each with a codeword that represents it
- the codeword is a bit pattern
- but it can be interpreted as a non-negative integer

Whenever a codeword is outputted during compression, what is written to the compressed file is the bit pattern

- using a number of bits determined by the current codeword length
- so at any point all bit patterns are the same length

LZW compression

The dictionary is build dynamically during compression

- and also during decompression

Initially dictionary contains all possible strings of length 1

Throughout the dictionary is closed under prefixes

- i.e. if the string s is represented in the dictionary, so is every prefix of s

It follows that a trie is an ideal representation of the dictionary

- every node in the trie represents a 'word' in the dictionary
- a trie is effective and efficient for other reasons too

LZW compression

Key question: how many bits are in a codeword?

- in the most used version of the algorithm, this value changes as the compression (or decompression) algorithm proceeds

At any given time during compression (or decompression)

- there is a current codeword length k
- so there are exactly 2^k distinct codewords available
  - i.e. all possible bit-strings of length k
- this limits the size of the dictionary
- however the codeword length can be incremented when necessary
- thereby doubling the number of available codewords
- initial value of k should be large enough to encode all strings of length 1

LZW compression – Pseudo code

```
set current text position i to 0;
initialise codeword length k (say to 8);
initialise the dictionary d;

while (the text t is not exhausted) {

  identify the longest string s, starting at position i of text t
  that is represented in the dictionary d;
  // there is such string, as all strings of length 1 are in d

  output codeword for the string s; // using k bits

  // move to the next position in the text
  i += s.length(); // move forward by the length of string just encoded
  c = character at position i in t; // character in next position

  add string s+c to dictionary d, paired with next available codeword;
  // may have to increment the codeword length k to make this possible
}
```

LZW compression – Variants

Constant codeword length: fix the codeword length for all time

- the dictionary has fixed capacity: when full, just stop adding to it

Dynamic codeword length (the version described here)

- start with shortest reasonable codeword length, say, 8 for normal text
- whenever dictionary becomes full
  - add 1 to current codeword length (doubles the number of codewords)
  - does not affect the sequence of codewords already output
- may specify a maximum codeword length, as increasing the size indefinitely may become counter-productive

LRU version: when dictionary full and codeword length maximal

- current string replaces Least Recently Used string in dictionary

LZW compression – Example

Text = G A C G A T A C G A T A C G
File size = 14 bytes, or 28 bits if 2 bits/char

Compressed file: 10 000 001 100 011 0101 0111 1001
file size = 26 bits

step	position in string	longest string in dictionary	b	add to dictionary	code
1	1	G	10	GA	4
2	2	A	000	AC	5
3	3	C	001	CG	6
4	4	GA	100	GAT	7
5	6	T	011	TA	8
6	7	AC	0101	ACG	9
7	9	GAT	0111	GATA	10
8	12	ACG	1001	-	-

LZW decompression

Decompression algorithm builds same dictionary as compression algorithm

- but one step out of phase

LZW decompression – Pseudo code

```
initialise codeword length k;
initialise the dictionary;

read the first codeword x from the compressed file f; // i.e. read k bits
String s = d.lookUp(x); // look up codeword in dictionary
output s; // output decompressed string

while (f is not exhausted){

  String oldS = s.clone(); // copy last string decompressed

  if (d is full) k++; // dictionary full so increase the code word length

  get next codeword x from f;  // i.e. read k bits
  s = d.lookUp(x); // look up codeword in dictionary
  output s; // output decompressed string

  String newS = oldS + s.charAt(0); // string to add to dictionary
  add string newS to dictionary d paired with next available codeword;
}
```

LZW decompression – Example

Compressed file: 10000001100011010101111001
file size = 26 bits

Uncompressed Text = G A C G A T A C G A T A C G

step	position in file	old string	code from dictionary	string	add to dictionary	code
0	1	-	10	G	-	-
1	3	G	000	A	GA	4
2	6	A	001	C	AC	5
3	9	C	100	GA	CG	6
4	12	GA	011	T	GAT	7
5	15	T	0101	AC	TA	8
6	19	AC	0111	GAT	ACG	9
7	23	GAT	1001	ACG	GATA	10

LZW decompression – Special case

It is possible to encounter a codeword that is not (yet) in the dictionary

- because decompression is ‘out of phase’ with compression
- but in that case it is possible to deduce what string it must represent
- consider: A A B A B A B A A
  and work through compression and decompression for this text

The solution:

```
if (lookUp fails)  s = oldS + oldS.charAt(0);
```

Example of this special case is available on moodle

LZW decompression

Appropriate data structure for decompression is a simple table

Complexity of compression and decompression both **O(n)**

- for a text of length n (if suitably implemented)
- algorithms essentially involves just one pass through the text

Strings – Notation

For a string $s=s_0s_1\ldots s_{m-1}$

- `m` is the length of the string
- `s[i]` is the `(i+1)`th element of the string, i.e. s_i
- `s[i..j]` is the substring from the `i`th to `j`th position, i.e. $s_is_{i+1}\ldots s_j$

Prefixes and suffixes

- `j`th prefix is the first `j` characters of `s` denoted `s[0..j-1]`
  - i.e. `s[0..j-1]` = $s_0s_1\ldots s_{j-1}$
  - `s[0..0-1]=s[0..-1]` (the `0`th prefix) is the empty string
- `j`th suffix is the last `j` characters of `s` denoted `s[m-j..m-1]`
  - i.e. `s[m-j..m-1]` = $s_{m-j}s_{m-j+1}\ldots s_{m-1}$
  - `s[m..m-1]` (the `0`th suffix) is the empty string

String comparison

Fundamental question: how similar, or how different, are 2 strings?

- applications include:
  - biology (DNA and protein sequences)
  - file comparison (diff in Unix, and other similar file utilities)
  - spelling correction, speech recognition,…

A more precise formulation:

given strings $s=s_0s_2…s_{m-1}$ and $t=t_0t_2…t_{n-1}$ of lengths m and n,

what is the smallest number of basic operations needed to transform s to t?

'Basic' operations for transforming strings:

- **insert** a single character
- **delete** of a single character
- **substitute** one character by another

String comparison – String distance

The distance between s and t is defined to be the smallest number of basic operations needed to transform s to t

- for example consider the strings s and t

s:	a	b	a	d	c	d	b	
t:	a	c	b	a	c	a	c	b

- we can show an alignment between s and t that illustrates how 4 steps would suffice to transform s into t
- hence the distance between s and t is less than or equal to 4

insert 'c' delete 'd' substitute 'a' for 'd' insert 'c'

s:	a	-	b	a	d	c	d	-	b
t:	a	c	b	a	-	c	a	c	b

String comparison – String distance

The distance between s and t is defined to be the smallest number of basic operations needed to transform s into t

– for example for the strings

```
s:  a  b  a  d  c  d  b
t:  a  c  b  a  c  a  c  b
```

the distance between s and t is less than or equal to 4

```
s:  a  -  b  a  d  c  d  -  b
t:  a  c  b  a  -  c  a  c  b
```

But could it be done in 3 steps?

– the answer is no, proof later based on our algorithm to find the distance for any two strings, so above alignment is an optimal alignment

String comparison – String distance

More complex models are possible

- e.g. we can allocate a cost to each basic operation
- our methods adapt easily but we will stick to the unit-cost model

String comparison algorithms use dynamic programming

- the problem is solved by building up solutions to sub-problems of ever increasing size
- often called the tabular method (it builds up a table of relevant values)
- eventually, one of the values in the table gives the required answer

The dynamic programming technique has applications to many different problems

String distance – Dynamic programming

Recall the i^{th} prefix of string s is the first i characters of s

- let d(i,j) be the distance between i^{th} prefix of s and the j^{th} prefix of t
- distance between s and t is then d(m,n)
  (since s and t of lengths m and n)

The basis of dynamic programming method is a recurrence relation

- more precisely we define the distance d(i,j) between i^{th} prefix of s and the j^{th} prefix of t in terms of the distance between shorter prefixes
  - i.e. in terms of the distances d(i-1,j-1), d(i,j-1) and d(i-1,j)

- in the base cases we set d(i,0)=i and d(0,j)=j for all i≤n and j≤m
- since the distance from/to an empty string to/from a string of length k is equal to k (we require k insertions/deletions)

String distance – Dynamic programming

In an optimal alignment of the **i**th prefix of **s** with the **j**th prefix of **t** the last position of the alignment must either be of the form:

$\boxed{\begin{matrix}*\\ *\end{matrix}}$ if `s[i-1] = t[j-1]` and $\boxed{\begin{matrix}-\\ *\end{matrix}}$, $\boxed{\begin{matrix}*\\ -\end{matrix}}$ or $\boxed{\begin{matrix}*\\ \$\end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, no operations are required and the distance is given by that between the **i-1**th and **j-1**th prefixes of **s** and **t**

$$
d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ & \text{otherwise} \end{cases}
$$

String distance – Dynamic programming

In an optimal alignment of the i^{th} prefix of **s** with the j^{th} prefix of **t** the last position of the alignment must either be of the form:

$\begin{bmatrix} * \\ * \end{bmatrix}$ if s[i-1] = t[j-1] and $\boxed{\begin{matrix} - \\ * \end{matrix}}$, $\begin{bmatrix} * \\ - \end{bmatrix}$ or $\begin{bmatrix} * \\ \$ \end{bmatrix}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, **insert** element into **s** and distance given by **1** (for the insertion) plus distance between i^{th} prefix of **s** and $i\text{-}1^{th}$ prefix of **t**

$$
d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\ d(i,j-1) \quad \} & \text{otherwise} \end{cases}
$$

String distance – Dynamic programming

In an optimal alignment of the i^th prefix of s with the j^th prefix of t the last position of the alignment must either be of the form:

$\boxed{\begin{matrix} * \\ * \end{matrix}}$ if s[i-1] = t[j-1] and $\boxed{\begin{matrix} - \\ * \end{matrix}}$, $\boxed{\begin{matrix} * \\ - \end{matrix}}$ or $\boxed{\begin{matrix} * \\ \$ \end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, delete an element from s and distance given by 1 plus distance between i-1^th prefix of s and i^th prefix of t

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j), \,\} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

In an optimal alignment of the **i**th prefix of **s** with the **j**th prefix of **t** the last position of the alignment must either be of the form:

$\boxed{\begin{matrix} * \\ * \end{matrix}}$ if s[i-1] = t[j-1] and $\boxed{\begin{matrix} - \\ * \end{matrix}}$, $\boxed{\begin{matrix} * \\ - \end{matrix}}$ or $\boxed{\begin{matrix} * \\ \$ \end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

In this case, **substitute** an element in **s** and distance given by **1** plus distance between **i-1**th prefix of **s** and **i-1**th prefix of **t**

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j),\ d(i-1,j-1)\,\} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

In an optimal alignment of the i^{th} prefix of s with the j^{th} prefix of t the last position of the alignment must either be of the form:

$\boxed{\begin{matrix} * \\ * \end{matrix}}$ if s[i-1] = t[j-1] and $\boxed{\begin{matrix} - \\ * \end{matrix}}$, $\boxed{\begin{matrix} * \\ - \end{matrix}}$ or $\boxed{\begin{matrix} * \\ \$ \end{matrix}}$ otherwise

where – is a gap, while * and $ are arbitrary but different characters

We take the minimum when s[i-1]≠t[j-1] as we want the optimal (minimal) distance

$$d(i,j) = \begin{cases} d(i-1,j-1) & \text{if } s[i-1]=t[j-1] \\ 1 + \min\{\, d(i,j-1),\ d(i-1,j),\ d(i-1,j-1) \,\} & \text{otherwise} \end{cases}$$

String distance – Dynamic programming

The complete recurrence relation is given by:

$$
\texttt{d(i,j)} = \begin{cases} \texttt{d(i-1,j-1)} & \text{if } \texttt{s[i-1]=t[j-1]} \\ \texttt{1+min\{ d(i,j-1),d(i-1,j),d(i-1,j-1)\}} & \text{otherwise} \end{cases}
$$

subject to **d(i,0)=i** and **d(0,j)=j** for all **i≤n-1** and **j≤m-1**

String distance – Dynamic programming

The dynamic programming algorithm for string distance comes immediately from the formula

- fill in the entries of an $m \times n$ table row by row, and column by column

Time and space complexity both O(mn)

- a consequence of the size of the table
- can easily reduce the space complexity to O(m+n)
- just keep the most recent entry in each column of the table

But what about obtaining an optimal alignment?

- can use a ‘traceback’ in the table (see example below)
- less obvious how this can be done using only O(m+n) space
- but in fact it turns out that it's still possible (Hirschberg's algorithm)

String distance – Example

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

The entries are calculated one by one by application of the formula

- the final table: d(7,8)=4 so the string distance is 4

String distance – Dynamic programming

The traceback phase used to construct an optimal alignment

- trace a path in the table from bottom right to top left
- draw an arrow from an entry to the entry that led to its value

Interpretation

- vertical steps as `deletions`
- horizontal steps as `insertions`
- diagonal steps as `matches` or `substitutions`
  - a `match` if the distance does not change and a `substitution` otherwise

The traceback is not necessarily unique

- since there can be more than one optimal alignment

String distance – Example (traceback)

s\t		0	1	2	3	4	5	6	7	8
			a	c	b	a	c	a	c	b
0		0	1	2	3	4	5	6	7	8
1	a	1	0	1	2	3	4	5	6	7
2	b	2	1	1	1	2	3	4	5	6
3	a	3	2	2	2	1	2	3	4	5
4	d	4	3	3	3	2	2	3	4	5
5	c	5	4	3	4	3	2	3	3	4
6	d	6	5	4	4	4	3	3	4	4
7	b	7	6	5	4	5	4	4	4	4

Corresponding alignment:

```
s: a  -  b  a  d  -  c  d  b
t: a  c  b  a  c  a  c  -  b
step: d ← h ← d ← d ← d ← h ← d ← v ← d
```

(d=diagonal, v = vertical, h = horizontal)

String/pattern search

Searching a (long) text for a (short) string/pattern

- many applications including
  - information retrieval
  - text editing
  - computational biology

Many variants, such as exact or approximate matches

- first occurrence or all occurrences
- one text and many strings/patterns
- many texts and one string/pattern

We describe three different solutions to the basic problem:

- given a text `t` (of length `n`) and a string/pattern `s` (of length `m`)
- find the position of the first occurrence (if it exists) of `s` in `t`
- usually `n` is large and `m` is small

String search – Brute force algorithm

Given a text `t` (of length `n`) and a string/pattern `s` (of length `m`) find the position of the first occurrence (if any) of `s` in `t`

The naive brute force algorithm

- also known as exhaustive search (as we simply test all possible positions)
- set the current starting position in the text to be zero
- compare text and string characters left-to-right until the entire string is matched or a character mismatches
- in the case of a mismatch

  advance the starting position in the text by 1 and repeat
- continue until a match is found or the text is exhausted

Algorithms expressed with char arrays rather than strings in Java

String search – Brute force algorithm

```
/** return smallest k such that s occurs in t starting at position k */
public int bruteForce (char[] s, char[] t){
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int sp = 0;  // starting position in text t
   int i = 0;   // curr position in text
   int j = 0;   // curr position in string/pattern s
   while (sp <= n-m && j < m) { // not reached end of text/string
      if (t[i] == s[j]){ // chars match
         i++;     // move on in text
         j++;     // move on in string/pattern
      } else {    // a mismatch
         j = 0;   // start again in string
         sp++;    // advance starting position
         i = sp;  // back up in text to new starting position
       }
   }
   if (j == m) return sp; // occurrence found (reached end of string)
   else return -1; // no occurrence (reached end of text)
}
```

String search – Brute force algorithm

Worst case is no better than O(mn)

- e.g. search for $\underbrace{\texttt{s = aa … ab}}_{\text{length } m}$ in $\underbrace{\texttt{t = aa ... aaaa … ab}}_{\text{length } n}$
- m character comparisons needed at each n−(m+1) positions in the text before the text/pattern is found

Typically, the number of comparisons from each point will be small

- often just 1 comparison needed to show a mismatch
- so we can expect O(n) on average

Challenges: can we find a solution that is…

1. linear, i.e. O(m+n) in the worst case?
2. (much) faster than brute force on average?

String search – KMP algorithm

The Knuth–Morris–Pratt (KMP) algorithm

- addresses first challenge: linear (`O(m+n)`) in the worst case

It is an on-line algorithm

- i.e., it removes the need to back-up in the text
- involves pre-processing the string to build a border table
- border table: an array `b` with entry `b[j]` for each position `j` of the string

If we get a mismatch at position `j` in the string/pattern

- we remain on the current text character (do not back-up)
- the border table tells us which string character should next be compared with the current text character

String search – KMP algorithm

A **substring** of string **s** is a sequence of consecutive characters of **s**

– if `s` has length `n`, then `s[i..j]` is a substring for `i` and `j` with $0 \le i \le j \le n-1$

A **prefix** of **s** is a substring that begins at position **0**

– i.e. `s[0..j]` for any `j` with $0 \le j \le n-1$

A **suffix** of **s** is a substring that ends at position **n-1**

– i.e. `s[i..n-1]` for any `i` with $0 \le i \le n-1$

A **border** of a string **s** is a substring that is both a prefix and a suffix and cannot be the string itself

– e.g. `s = a c a c g a t a c a c`

– `a c` and `a c a c` are borders and `a c a c` is the longest border

Many strings have no border

– we then say that the empty string **ε** (of length 0) is the longest border

String search – Border table

KMP algorithm requires the **border table** of the string pattern

– a border of a string s is a substring that is both a prefix and a suffix and cannot be the string itself

Border table b: array which has the same size as the string

– `b[j]` = the length of the longest border of `s[0..j-1]`

= `max { k | s[0..k-1] = s[j-k..j-1] ∧ k<j }`

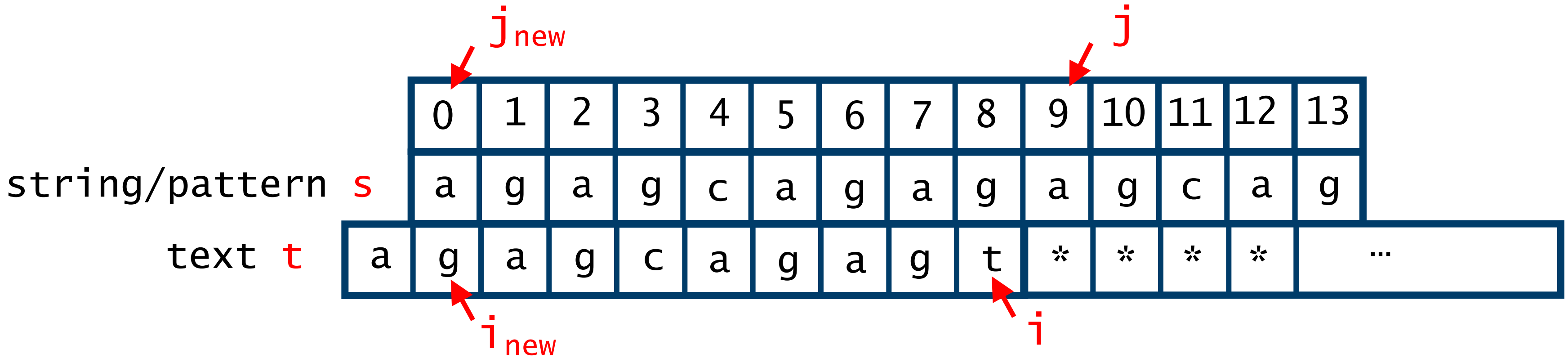
Example

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

– no common prefix/suffix of ababac so set to 0

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the brute force algorithm, after the mis-match:

- s has to be ‘moved along’ one position relative to t
- then we start again at position 0 in s and jump back j-1 positions in t

String search – Brute force versus KMP

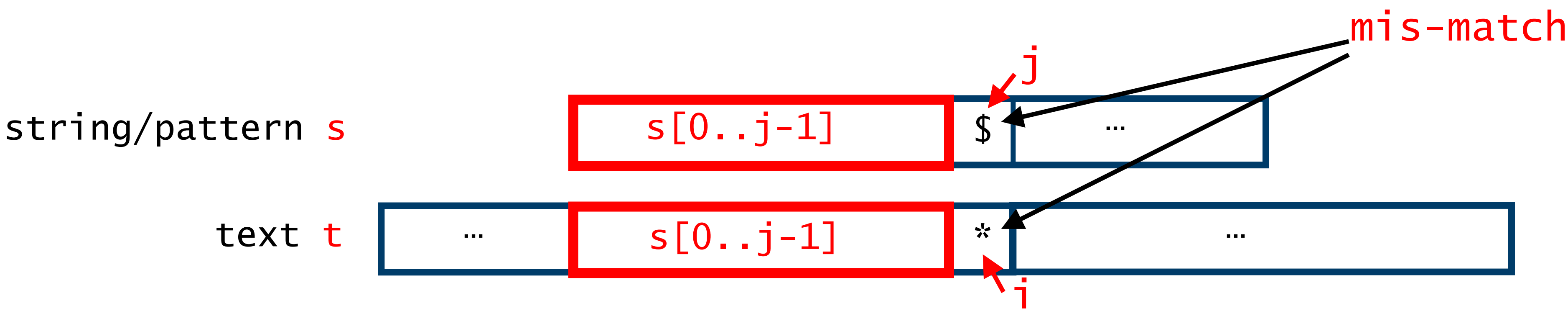
Example – Mismatch between **s** and **t** at position **9** in **s**

														j				
					0	1	2	3	4	5	6	7	8	9	10	11	12	13
string/pattern s					a	g	a	g	c	a	g	a	g	a	g	c	a	g
text t	a	g	a	g	c	a	g	a	g	t	*	*	*	*	…			
										i								

Applying the KMP algorithm, after the mis-match:

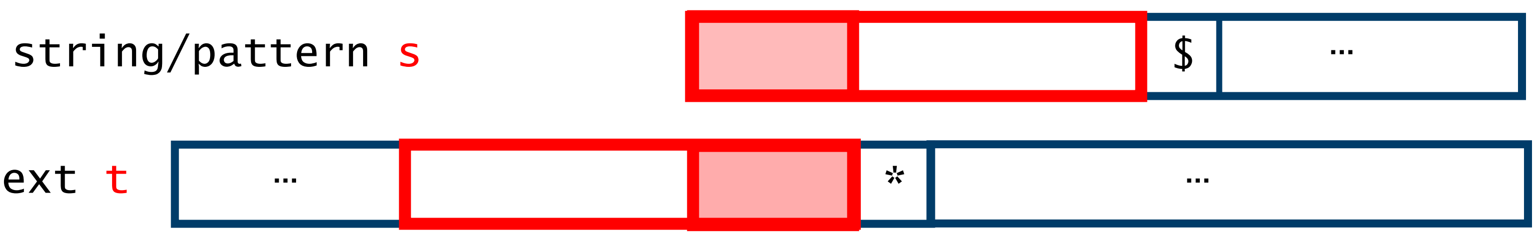
– s has to be ‘moved along’ until the characters to the left of i again match

String search – Brute force versus KMP



Need to move s along until the characters to the left of i match therefore need start of s[0..j-1] to match end of s[0..j-1]

- therefore use longest border of s[0..j-1]
- i.e. longest substring that is both a prefix and a suffix of s[0..j-1]



String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**

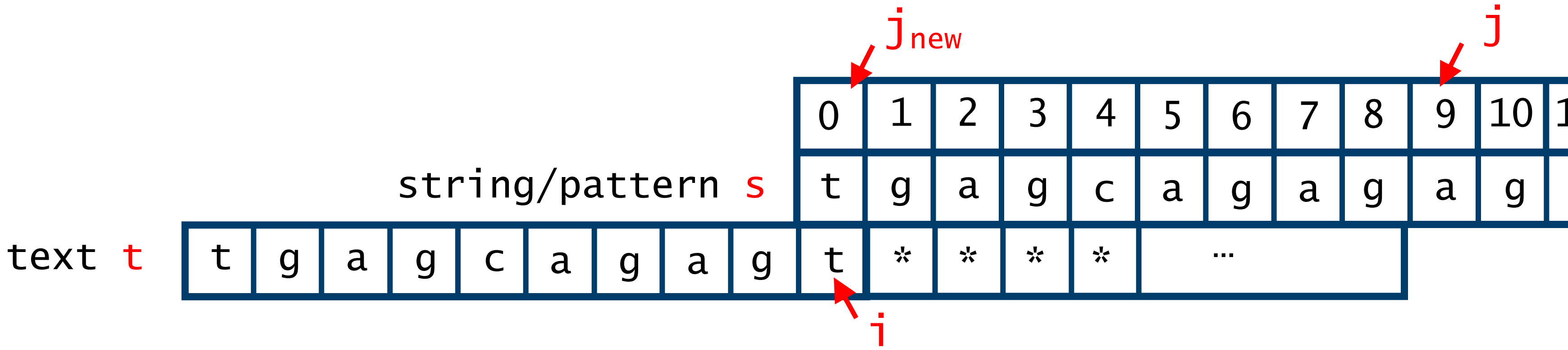
										j_{new}					j				
						0	1	2	3	4	5	6	7	8	9	10	11	12	13
string/pattern s						**a**	**g**	**a**	**g**	c	a	g	a	g	a	g	c	a	g
text t	a	g	a	g	c	**a**	**g**	**a**	**g**	*	*	*	*	*	…				
										i									

Applying the KMP algorithm, after the mis-match:

- s has to be 'moved along' until the characters to the left of i again match
- this determines the new value of j, the value of i is unchanged
- length of the longest border of s[0..j-1] is 4 in this case
  - i.e. longest substring that is both a prefix and a suffix of s[0..j-1]
- so the new value of j is 4

String search – Brute force versus KMP

Example – Mismatch between **s** and **t** at position **9** in **s**



Applying the KMP algorithm, after the mis-match:

– s has to be 'moved along' until the characters to the left of i again match

If we cannot move s along to get a match, then we need to

– reset j (i.e. return to the start of the string) and i remains unchanged

String search – Brute force versus KMP

Example – Mismatch between s and t at position 0 in s

		j														
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	
string/pattern s		t	g	a	g	c	a	g	a	g	a	g	c	a	g	
text t	a	g	a	g	c	a	g	a	g	t	*	*	*	*	…	
	i	i_{new}														

Applying the KMP algorithm, after the mis-match:

– s has to be 'moved along' until the characters to the left of i again match

If we cannot move s along to get a match, then we need to

– reset j (i.e. return to the start of the string) and i remains unchanged

– unless j is already 0 and in this case increment i

KMP search – Implementation

```
/** return smallest k such that s occurs from position k in t or -1 if no k exists */
public int kmp(char[] t, char[] s) {
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int i = 0; // current position in text
   int j = 0; // current position in string s
   int [] b = new int[m]; // create border table
   setUp(b); // set up the border table
   while (i <= n) { // not reached end of text
      if (t[i] == s[j]){ // if positions match
         i++; // move on in text
         j++; // move on in string
         if (j = m) return i - j; // reached end of string so a match
      } else { // mismatch adjust current position in string using the border table
          if (b[j] > 0) // there is a common prefix/suffix
              j = b[j]; // change position in string (position in text unchanged)
          else { // no common prefix/suffix
              if (j = 0) i++; // move forward one position in text if not advanced
              else j = 0; // else start from beginning of the string
          }
      }
   }
   return -1; // no occurrence
}
```

KMP – Example

string/pattern s

a	b	a	b	a	c	a

text t

b	a	c	b	a	b	**a**	**b**	**a**	**b**	**a**	**c**	**a**	a	b

String/pattern has been found

position in string j=6

string/pattern s	a	b	a	b	a	c	a
j	0	1	2	3	4	5	6
b[j]	0	0	0	1	2	3	0

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
          if (j=0) i++;
          else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of **i** and **k** (where **k=i-j**) during the iterations

- clearly i≤n and since j is never negative we also have k≤n
- in each iteration either i or k is incremented and neither is decremented

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of `i` and `k` (where `k=i-j`) during the iterations

- clearly `i≤n` and since `j` is never negative we also have `k≤n`
- in each iteration either `i` or `k` is incremented and neither is decremented
  - `i++ > i` and `(i++)-(j++) = i-j`

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
          if (j=0) i++;
          else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of i and k (where k=i-j) during the iterations

- clearly $i \leq n$ and since j is never negative we also have $k \leq n$
- in each iteration either i or k is incremented and neither is decremented
  - i = i and i-b[j] > i-j
    - since b[j]<j as b[j] longest border in a string of length j

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of i and k (where k=i-j) during the iterations

- clearly $i \le n$ and since j is never negative we also have $k \le n$
- in each iteration either i or k is incremented and neither is decremented
  - `i++ > i` and `(i++)-j > i-j`

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of `i` and `k` (where `k=i-j`) during the iterations

- clearly `i≤n` and since `j` is never negative we also have `k≤n`
- in each iteration either `i` or `k` is incremented and neither is decremented
  - `i = i` and `i-0 > i-j`
    - since `j>0` must hold for the else case to be taken

KMP search – Analysis

```
while (i<n)
   if (t[i] == s[j]){
       i++; j++;
   }
   else {
       if (b[j]>0) j = b[j];
       else {
           if (j=0) i++;
           else j = 0;
       }
    }
```

For the complexity we need to know the number of loop iterations

Consider values of `i` and `k` (where `k=i-j`) during the iterations

- clearly `i≤n` and since `j` is never negative we also have `k≤n`
- in each iteration either `i` or `k` is incremented and neither is decremented
- so the number of iterations of the loop is at most `2n`

Hence KMP is `O(n)` in the worst case

KMP search – Analysis

KMP search is $O(n)$ in the worst case

Creating the border table

– naïve method requires $O(j^2)$ steps to evaluate b[j] giving $O(m^3)$ overall

– a more efficient method is possible that requires just $O(m)$ steps in total involves a subtle application of the KMP algorithm (details are omitted)

Overall complexity of KMP search

– KMP can be implemented to run in $O(m+n)$ time

– $O(m)$ for setting up the border table

– $O(n)$ for conducting the search

Have addressed challenge 1

– KMP algorithm is linear (i.e. $O(m+n)$)

Boyer-Moore Algorithm

Challenge 1: can we find a solution that is linear in the worst case?

Yes: KMP

Challenge 2: can we find a solution that is (much) faster than brute force on average?

Boyer-Moore: almost always faster than brute force or KMP

- variants are used in many applications
- typically, many text characters are skipped without even being checked
- the string/pattern is scanned right-to-left
- text character involved in a mismatch is used to decide next comparison

Boyer–Moore Algorithm – Example

Search for ‘**pill**’ in ‘**the caterpillar**’

```
the caterpillar
pill
   ^
```

Search for string from right to left

- start by comparing m^{th} element of text with last character of string

  m is the length of the string, i.e. equals 4

Boyer–Moore Algorithm – Example

Search for '**pill**' in '**the caterpillar**'

```
the caterpillar
        pill
        ^
```

Search for string from right to left

- continue search from the last position in the string
- 'p' matches and we have found the string in the text

Boyer-Moore Algorithm – Simplified version

The string is scanned right-to-left

- text character involved in a mismatch is used to decide next comparison
- involves pre-processing the string to record the position of the last occurrence of each character c in the alphabet
- therefore the alphabet must be fixed in advance of the search

Last occurrence position of character c in the string s

- equals `max{k | s[k]=c }` if such a k exists and `-1` otherwise

Want to store last occurrence position of c in an array element p[c]

- but in Java we can not index an array by characters
- instead can use the static method `Character.getNumericValue(c)`
- to compute an appropriate array index

Simplified version (often called the Boyer-Moore-Horspool algorithm)

Boyer–Moore Algorithm – Simplified version

In our pseudocode we assume an array **p[c]** indexed by characters

- the characters range over the underlying alphabet of the text
- p[c] records the position in the string of the last occurrence of char c
- if the character c is absent from the string s, then let p[c]=-1

Assume ASCII character set (**128** characters)

- for Unicode (more than 107,000 characters), p would be a large array

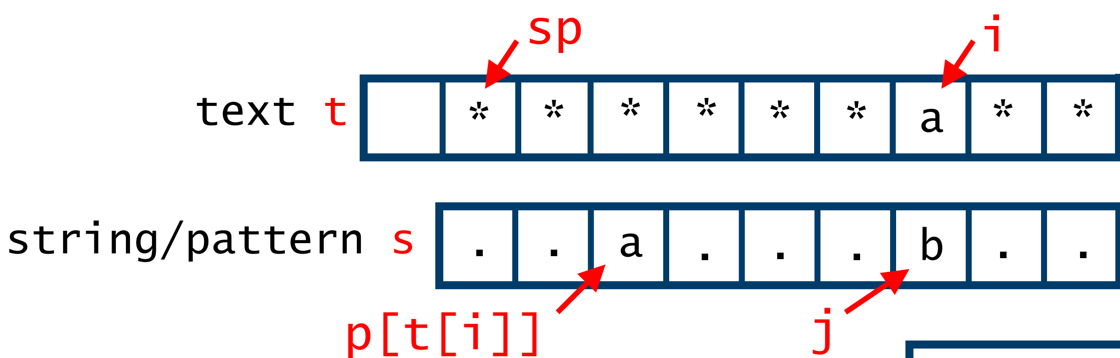
On finding a mismatch there is a jump step in the algorithm

- if the mismatch is between s[j] and t[i]
- 'slide s along' so that position p[t[i]] of s aligns with t[i]
  - i.e. align last position in s of character t[i] with position i of t
- if this moves s in the 'wrong direction', instead move s one position right
- if t[i] does not appear in string, 'slide string' passed t[i]
  - i.e. align position -1 of s with position i of t

Boyer–Moore Algorithm – Jump step – Case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**



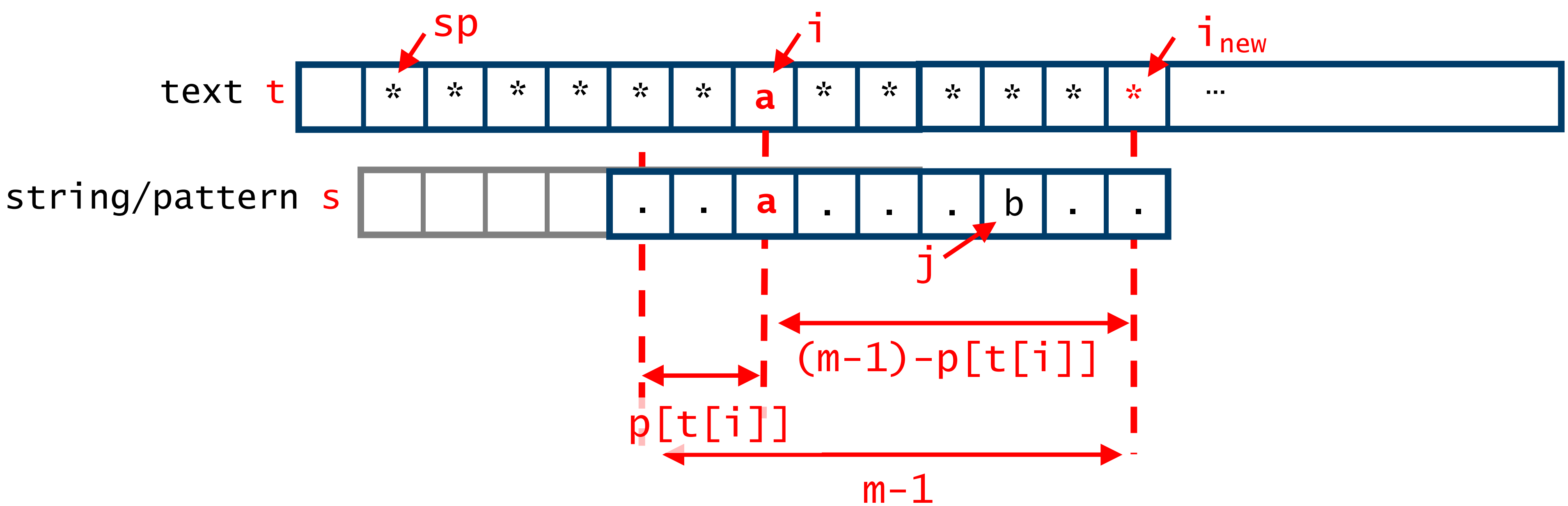
reminder: p[t[i]] records the position in s the last occurrence of character t[i]

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step – Case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**

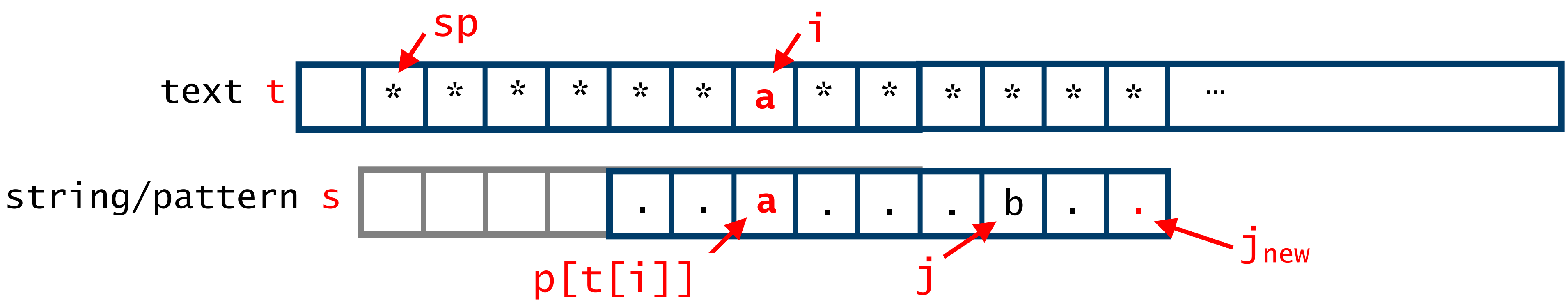


- `i` records the current position in the text we are checking
- new value of `i` equals `i+(m-1)-p[t[i]]`

Boyer-Moore Algorithm – Jump step – Case 1

Assume a mismatch between position s[j] and position t[i]

Case 1: the last position of character t[i] in s is before position j

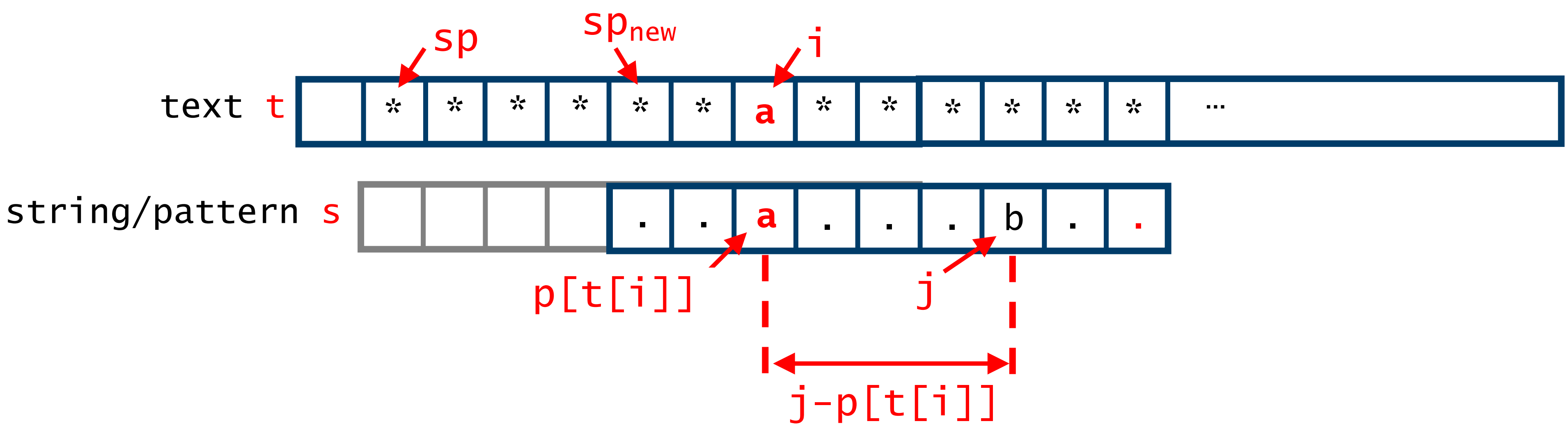


- j records the current position in the string we are checking
- new value of j equals m-1 (start again from the end of the string/pattern)

Boyer–Moore Algorithm – Jump step – Case 1

Assume a mismatch between position **s[j]** and position **t[i]**

Case 1: the last position of character **t[i]** in **s** is before position **j**

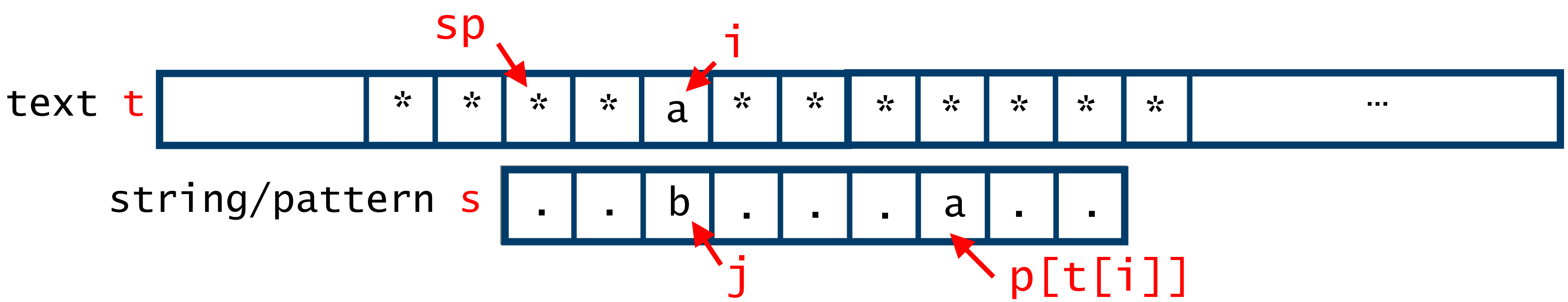


- sp records the current starting position of string in the text
- new value of sp equals sp+j-p[t[i]] as this is the amount the pattern/ string has been moved forward

Boyer–Moore Algorithm – Jump step – Case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**



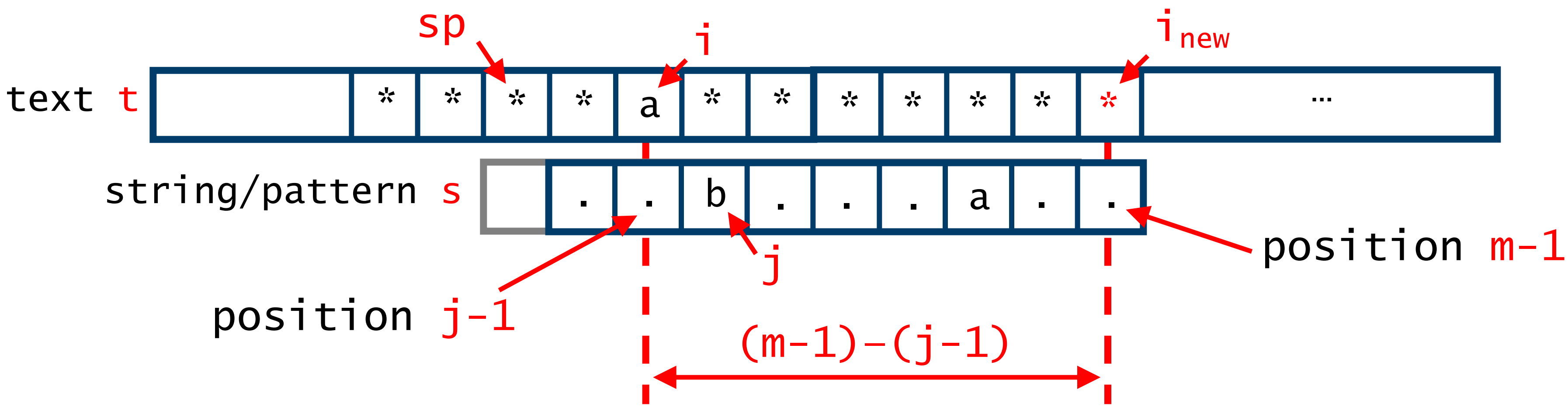
move string along by one place and start again from the end of the string

- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step – Case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**

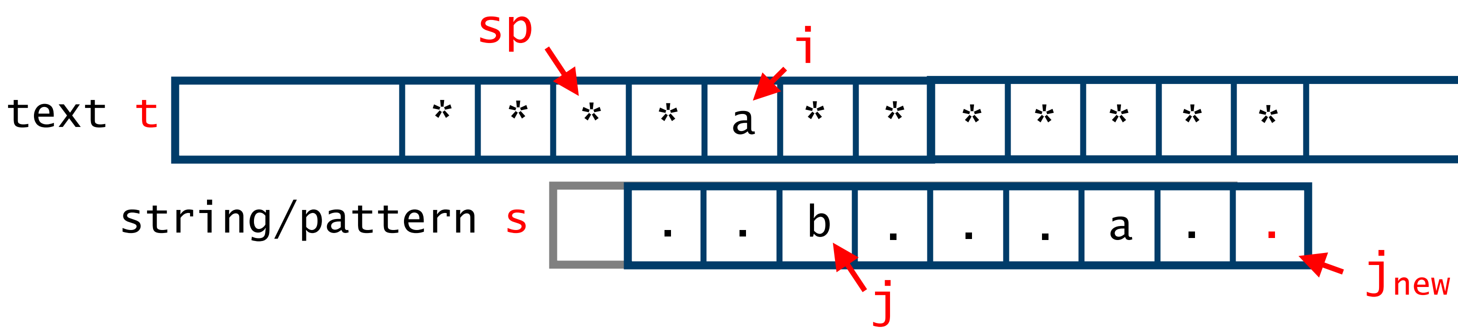


- i records the current position in the text we are checking
- new value of i equals i+(m-1)-(j-1) = i+(m-j)

Boyer–Moore Algorithm – Jump step – Case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**

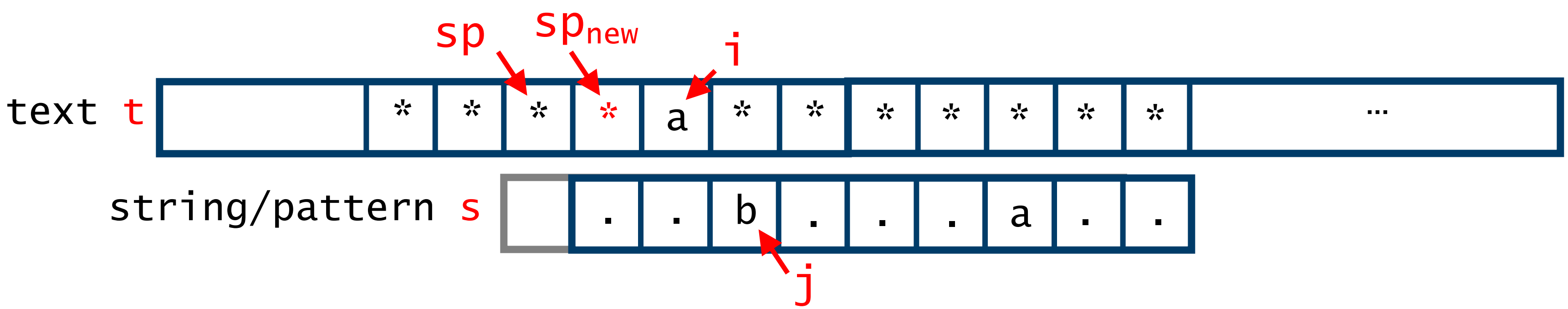


- j records the current position in the string we are checking
- new value of j equals m-1

Boyer–Moore Algorithm – Jump step – Case 2

Assume a mismatch between position **s[j]** and position **t[i]**

Case 2: last position of character **t[i]** in **s** is at least at position **j**

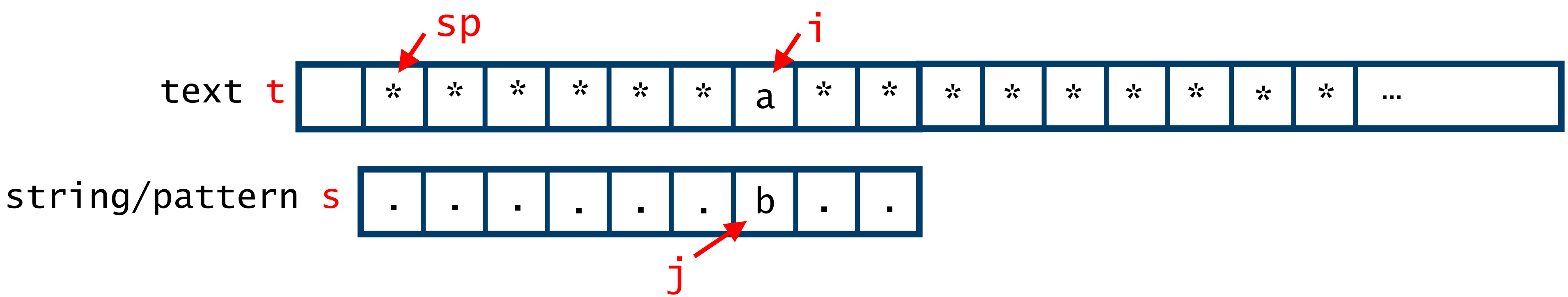


- sp records the current starting position of string in the text
- new value of sp equals sp+1

Boyer–Moore Algorithm – Jump step – Case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)

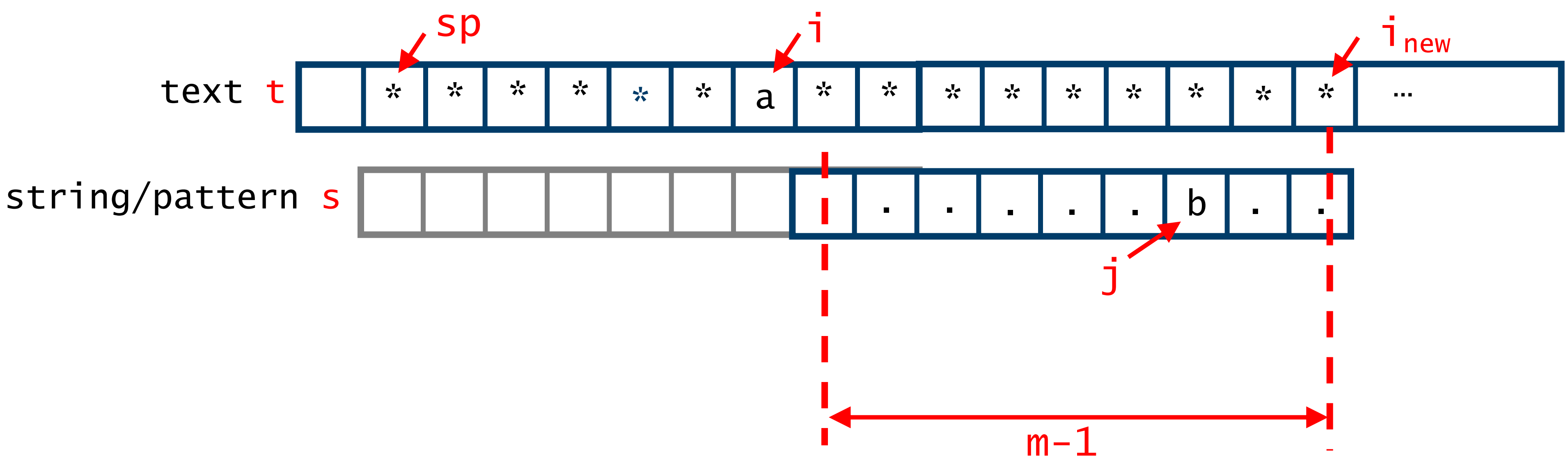


- i records the current position in the text we are checking
- j records the current position in the string we are checking
- sp records the current starting position of string in the text

Boyer–Moore Algorithm – Jump step – Case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)

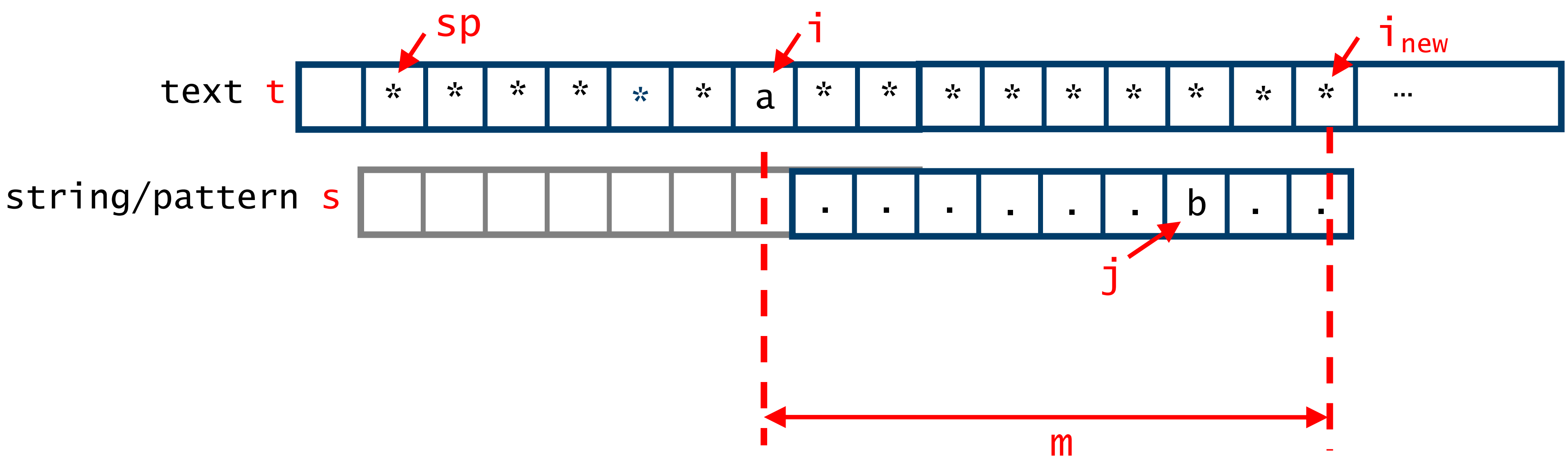


– i records the current position in the text we are checking
– new value of i equals i+m

Boyer–Moore Algorithm – Jump step – Case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



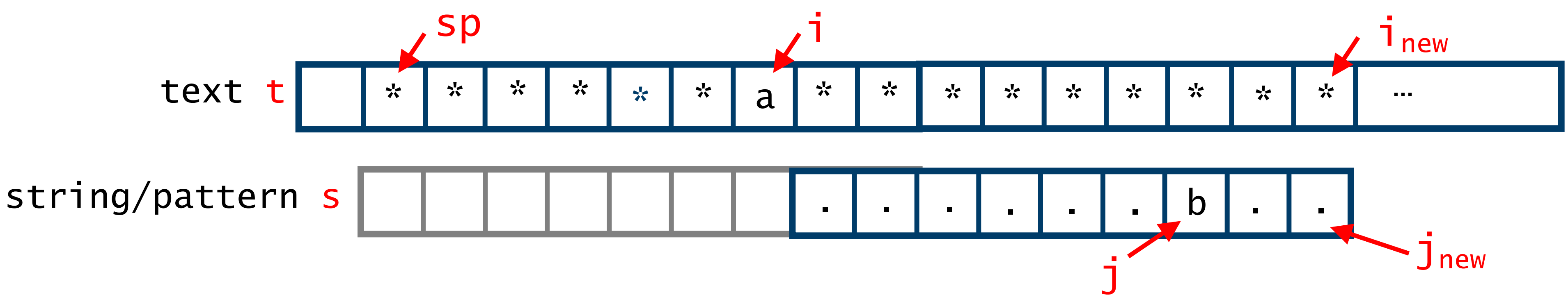
– i records the current position in the text we are checking

– new value of i equals i+m

Boyer–Moore Algorithm – Jump step – Case 3

Assume a mismatch between position `s[j]` and position `t[i]`

Case 3: **character `t[i]` does not appear in `s`** (i.e. we have `p[j]=-1`)



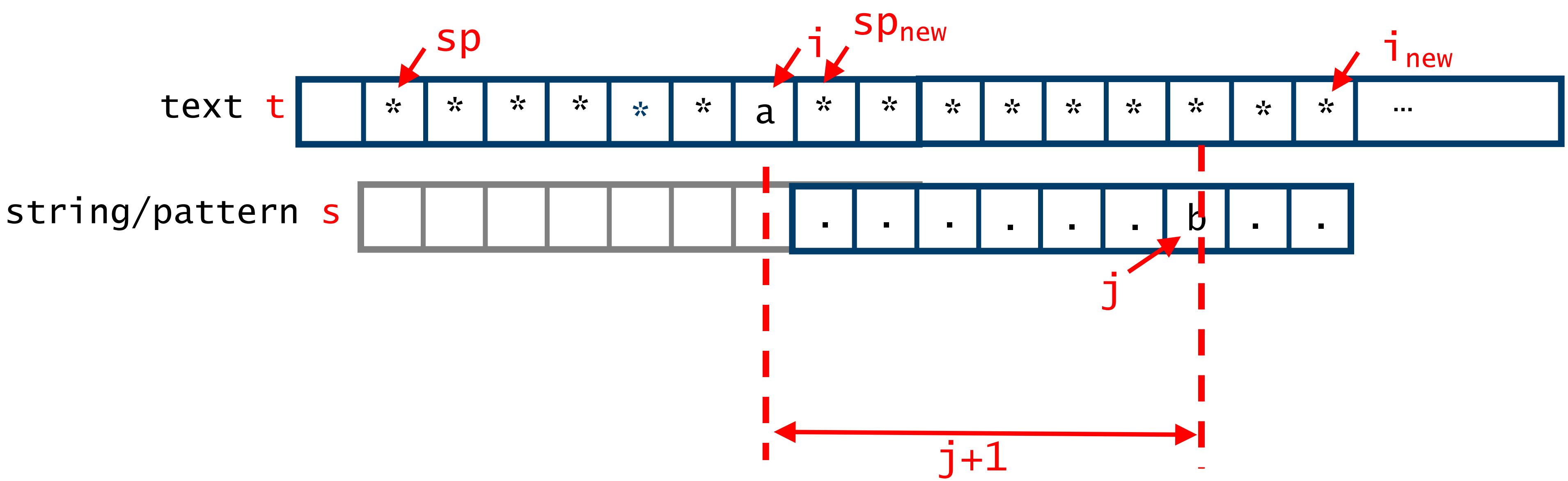
– j records the current position in the string we are checking

– new value of j equals m-1 (start again from the end of the string/pattern)

Boyer–Moore Algorithm – Jump step – Case 3

Assume a mismatch between position **s[j]** and position **t[i]**

Case 3: character **t[i]** does not appear in **s** (i.e. we have **p[j]=-1**)



- sp records the current starting position of string in the text
- new value of sp equals sp+(j+1) as this is the amount the pattern/string has been moved forward

Boyer–Moore Algorithm – All cases

Case 1: **p[t[i]]<j** and **p[t[i]]≥0**

- new value of i equals i+m-1-p[t[i]]
- new value of j equals m-1

Case 2: **p[t[i]]>j**

- new value of i equals i+m-j
- new value of j equals m-1
- new value of sp equals sp+1

Case 3: **p[t[i]]=-1**

- new value of i equals i+m
- new value of j equals m-1
- new value of sp equals sp+j+1

Note p[t[i]] cannot equal j as p[t[i]] last position of character t[i] in s and mismatch between t[i] and s[j]

Boyer–Moore Algorithm – All cases

We find that we can express these updates as follows:

- new value of `i` equals `i + m - min(1+p[t[i]],j)`
- new value of `j` equals `m-1`
- new value of `sp` equals `sp + max(j-p[t[i]],1)`

You do not need to learn these updates, just how the algorithm works

- this is sufficient for running it on an example (as you saw)
- and for working out what the updates are if needed (again as you saw)

Boyer–Moore Algorithm – Implementation

```
/** return smallest k such that s occurs at k in t or -1 if no k exists */
public int bm(char[] t, char[] s) {
   int m = s.length; // length of string/pattern
   int n = t.length; // length of text
   int sp = 0; // current starting position of string in text
   int i = m-1; // current position in text
   int j = m-1; // current position in string/pattern
   // declare a suitable array p
   setUp(s, p);   // set up the last occurrence array
   while (sp <= n-m && j >= 0) {
      if (t[i] == s[j]){ // current characters match
         i--; // move back in text
         j--; // move back in string
      } else { // current characters do not match
         sp += max(1, j - p[t[i]]);
         i += m - min(j, 1 + p[t[i]]);
         j = m-1;   // return to end of string
      }
   }
   if (j < 0) return sp; else return -1; // occurrence found yes/no
}
```

Boyer–Moore Algorithm – Complexity

Worst case is no better than O(mn)

– e.g. search for $\underbrace{\texttt{s = ba ... aa}}_{\text{length } m}$ in $\underbrace{\texttt{t = aa ... aaaa ... aa}}_{\text{length } n}$

– m character comparisons needed at each n-(m+1) positions in the text before the text/pattern is found

There is an extended version which is linear, i.e. O(m+n)

– this as the good suffix rule (or magic)