FPT UNIVERSITY
DREAM OF INNOVATION

8. Text Processing

FPT UNIVERSITY
DREAM OF INNOVATION

Objectives

- Abundance of Digitized Text
- The problem of String Matching
- Brute-Force algorithm
- Knunth-Morris-Pratt Algorithm
- Data Compression
- Condition for Data Compression
- Huffman Coding Algorithm
- LZW Algorithm
- Run-length Encoding

Abundance of Digitized Text

Despite the wealth of multimedia information, text processing remains one of the dominant functions of computers. Computers are used to edit, store, and display documents, and to transport files over the Internet. Furthermore, digital systems are used to archive a wide range of textual information, and new data is being generated at a rapidly increasing pace. Common examples of digital collections that include textual information are:

- Snapshots of the World Wide Web, as Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content.
- All documents stored locally on a users computer
- Email archives
- Compilations of status updates on social networking sites such as Facebook
- Feeds from microblogging sites such as Twitter and Tumblr

These collections include written text from hundreds of international languages. Furthermore, there are large data sets (such as DNA) that can be viewed computationally as “strings” even though they are not language.
In this lesson, we explore some of the fundamental algorithms that can be used to efficiently analyze and process large textual data sets.

The problem of String Matching

Given a string S, the problem of string matching deals with finding whether a pattern p occurs in S and if p does occur then returning position in S where p occurs.

Brute-Force algorithm

One of the most obvious approach towards the string matching problem would be to compare the first element of the pattern to be searched p, with the first element of the string S in which to locate p. If the first element of p matches the first element of S, compare the second element of p with second element of S. If match found proceed likewise until entire p is found. If a mismatch is found at any position, shift p one position to the right and repeat comparison beginning from first element of p. This algorithm is called Brute-Force. Its' complexity (worst case) is $O(nm)$.

Brute-Force algorithm demo - 1

- Below is an illustration of how the previously described O(mn) approach works.
- String S

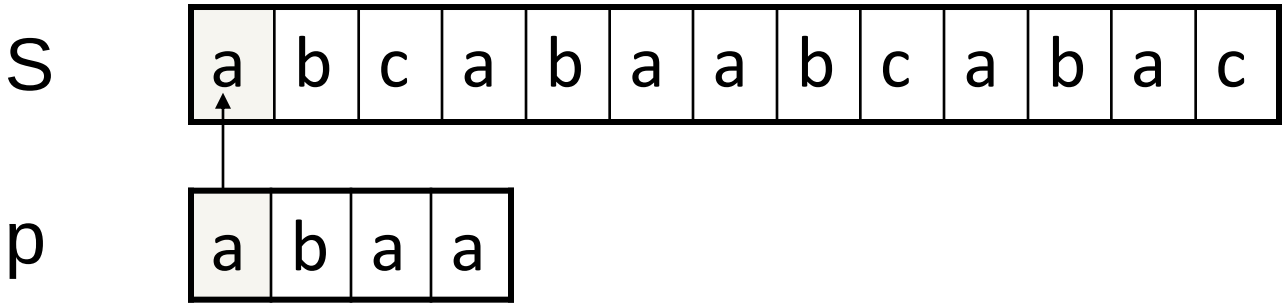
a	b	c	a	b	a	a	b	c	a	b	a	c

- Pattern p

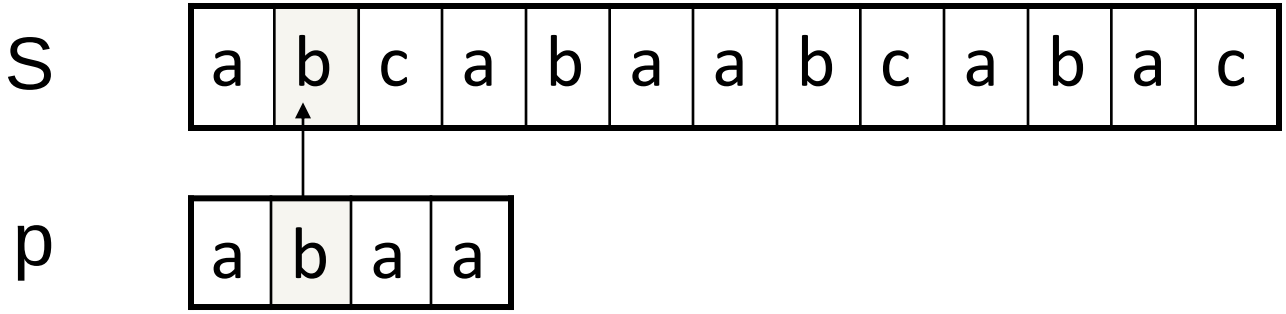
a	b	a	a

Brute-Force algorithm demo - 2

Step 1: compare p[1] with S[1]

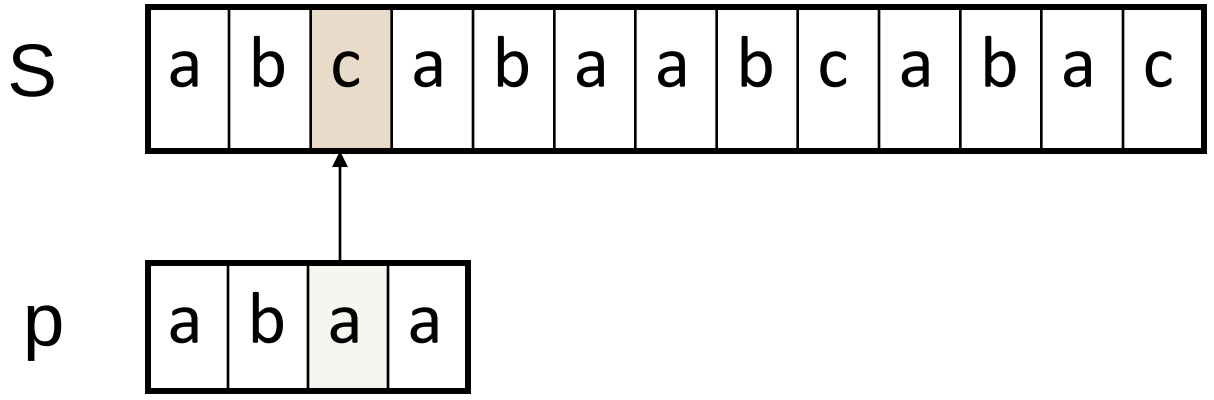


Step 2: compare p[2] with S[2]



Brute-Force algorithm demo - 3

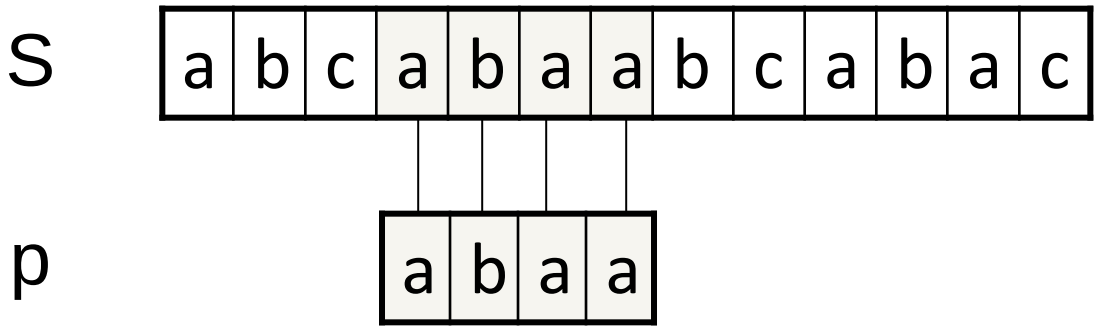
Step 3: compare p[3] with S[3]



Mismatch occurs here..

Since mismatch is detected, shift p one position to the left and perform steps analogous to those from step 1 to step 3. At position where mismatch is detected, shift p one position to the right and repeat matching procedure.

Brute-Force algorithm demo - 4



Finally, a match would be found after shifting p three times to the right side.
Drawbacks of this approach: if m is the length of pattern p and n the length of string S, the matching time is of the order O(mn). This is a certainly a very slow running algorithm.
What makes this approach so slow is the fact that elements of S with which comparisons had been performed earlier are involved again and again in comparisons in some future iterations. For example: when mismatch is detected for the first time in comparison of p[3] with S[3], pattern p would be moved one position to the right and matching procedure would resume from here. Here the first comparison that would take place would be between p[0]=a and S[1]=b. It should be noted here that S[1]=b had been previously involved in a comparison in step 2. this is a repetitive use of S[1] in another comparison.
It is these repetitive comparisons that lead to the runtime of O(mn).

The Knuth-Morris-Pratt Algorithm

- Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem.
- A matching time of O(n+m) is achieved by avoiding comparisons with elements of S that have previously been involved in comparison with some element of the pattern p to be matched. i.e., backtracking on the string S never occurs

Components of KMP algorithm

- **The prefix function, Π**

The prefix function,Π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern p. In other words, this enables avoiding backtracking on the string S.

- **The KMP Matcher**

With string S, pattern p and prefix function Π as inputs, finds the occurrence of p in S and returns the number of shifts of p after which occurrence is found.

The prefix function, Π

Following pseudocode computes the prefix fucnction, Π:

Compute-Prefix-Function (p)

```
1  m ← length[p]            //p pattern to be matched
2  Π[1] ← 0
3  k ← 0
4        for q ← 2 to m
5              do while k > 0 and p[k+1] != p[q]
6                    do k ← Π[k]
7                    If p[k+1] = p[q]
8                      then k ← k +1
9                    Π[q] ← k
10    return Π
```

Example: compute Π for the pattern p below:

p	a	b	a	b	a	c	a

Initially: m = length[p] = 7
Π[1] = 0
k = 0

Step 1: q = 2, k=0
Π[2] = 0

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0					

Step 2: q = 3, k = 0,
Π[3] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1				

Step 3: q = 4, k = 1
Π[4] = 2

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	A
Π	0	0	1	2			

Step 4: q = 5, k =2

Π[5] = 3

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3		

Step 5: q = 6, k = 3

Π[6] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	

Step 6: q = 7, k = 1

Π[7] = 1

q	1	2	3	4	5	6	7
p	a	b	a	b	a	c	a
Π	0	0	1	2	3	1	1

After iterating 6 times, the prefix function computation is complete: →

q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

The KMP Matcher

The KMP Matcher, with pattern p, string S and prefix function Π as input, finds a match of p in S.

Following pseudocode computes the matching component of KMP algorithm:

KMP-Matcher(S,p)

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0                                   //number of characters matched
5 for i ← 1 to n                          //scan S from left to right
6     do while  q > 0 and p[q+1] != S[i]
7           do  q ← Π[q]                  //next character does not match
8        if p[q+1] = S[i]
9          then q ← q + 1                 //next character matches
10       if q = m                         //is all of p matched?
11         then print "Pattern occurs with shift" i – m
12              q ← Π[ q]                 // look for the next match
```

Note: KMP finds every occurrence of a p in S. That is why KMP does not terminate in step 12, rather it searches remainder of S for any more occurrences of p.

Illustration: given a String S and pattern p as follows:

S

b	a	c	b	a	b	a	b	a	b	a	c	a	c	a

p

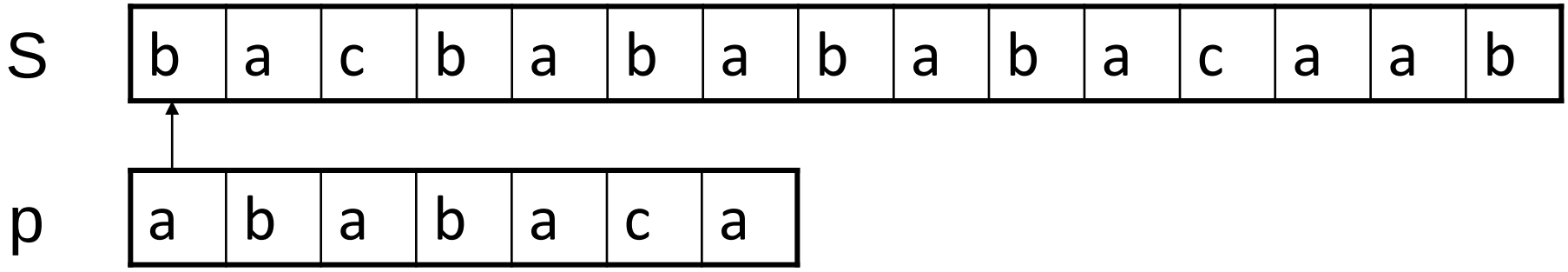
a	b	a	b	a	c	a

Let us execute the KMP algorithm to find whether p occurs in S.

For p the prefix function, Π was computed previously and is as follows:

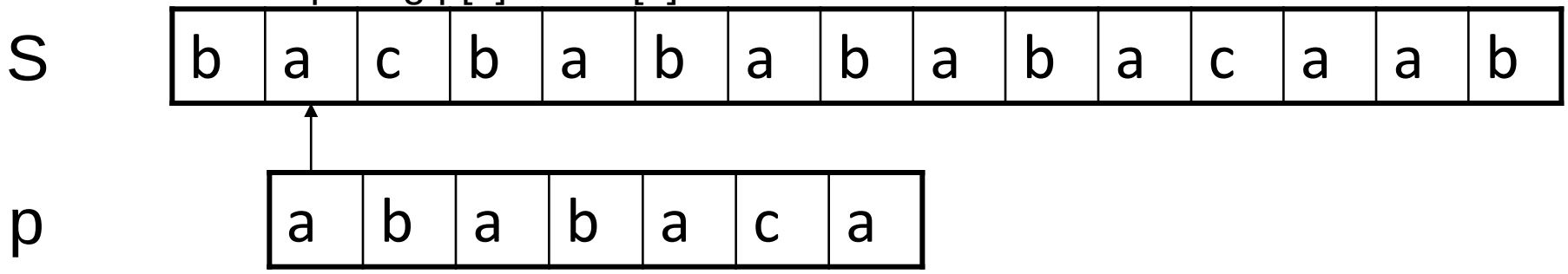
q	1	2	3	4	5	6	7
p	a	b	A	b	a	c	a
Π	0	0	1	2	3	1	1

Initially: n = size of S = 15;
m = size of p = 7
Step 1: i = 1, q = 0
comparing p[1] with S[1]



P[1] does not match with S[1]. p will be shifted one position to the right.

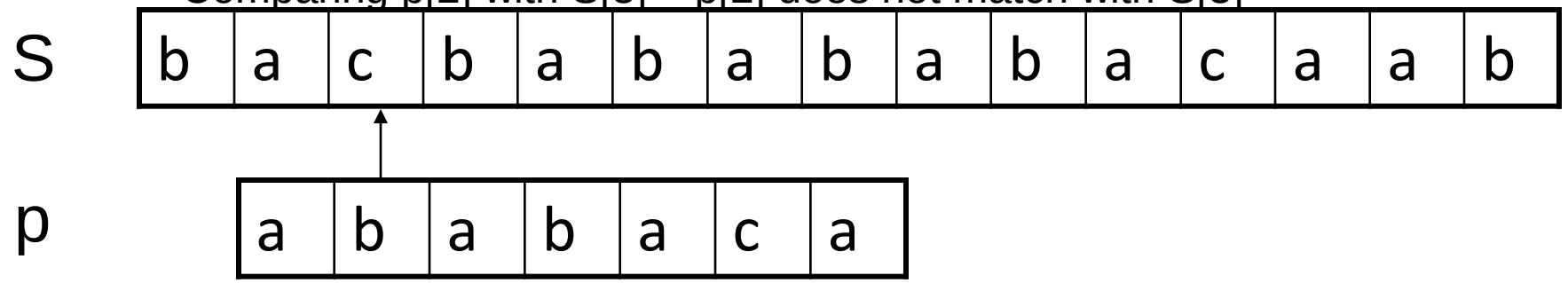
Step 2: i = 2, q = 0
comparing p[1] with S[2]



P[1] matches S[2]. Since there is a match, p is not shifted.

Step 3: i = 3, q = 1

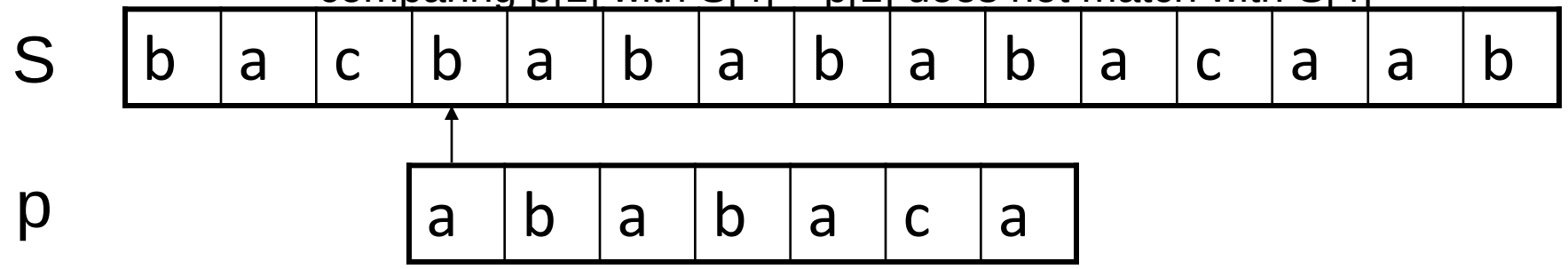
Comparing p[2] with S[3] p[2] does not match with S[3]



Backtracking on p, comparing p[1] and S[3]

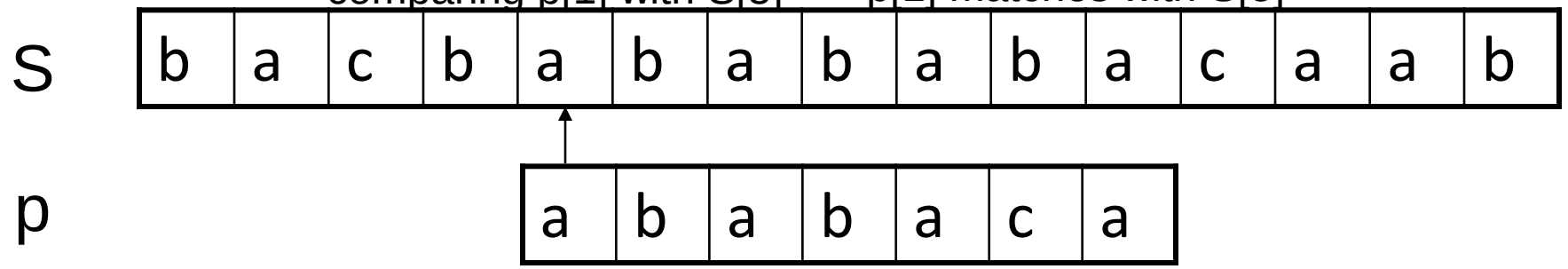
Step 4: i = 4, q = 0

comparing p[1] with S[4] p[1] does not match with S[4]

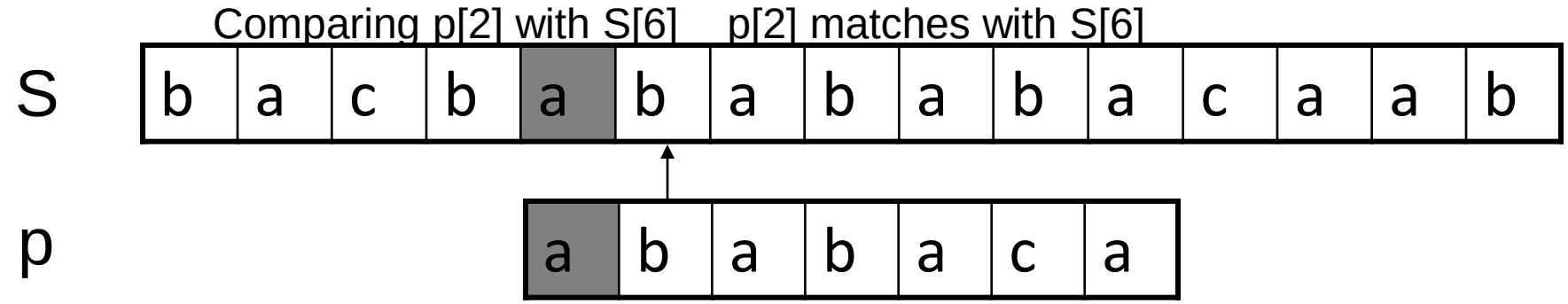


Step 5: i = 5, q = 0

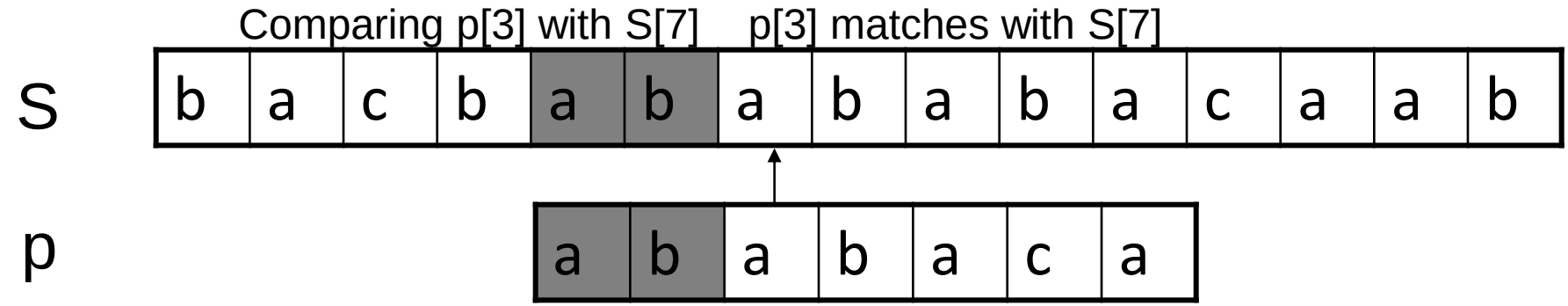
comparing p[1] with S[5] p[1] matches with S[5]



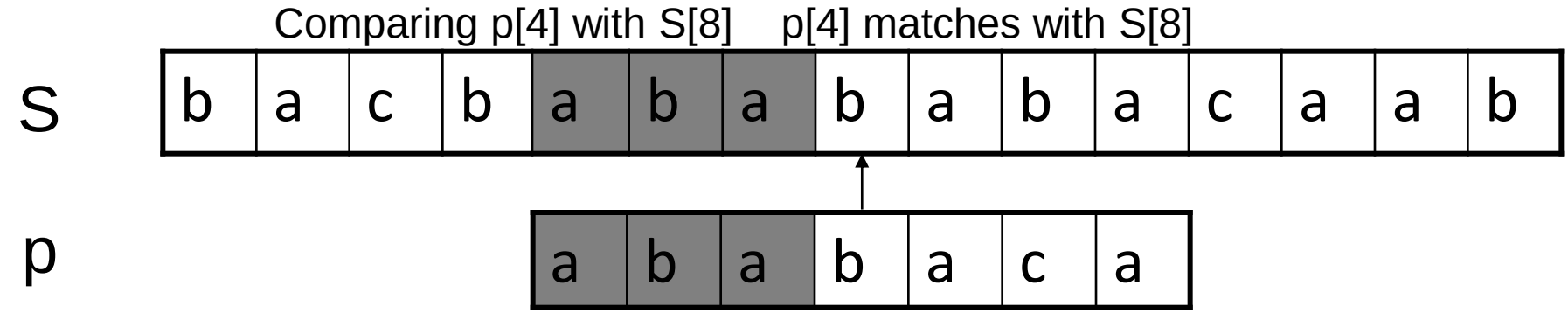
Step 6: i = 6, q = 1



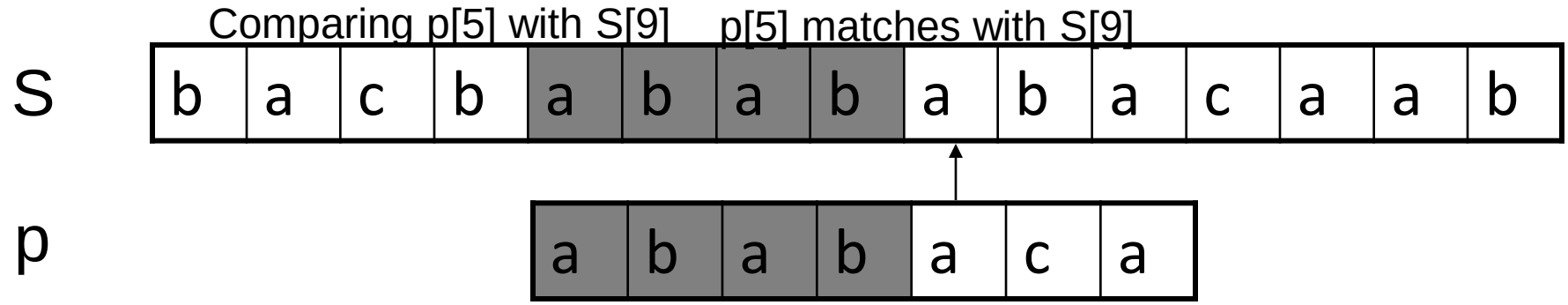
Step 7: i = 7, q = 2



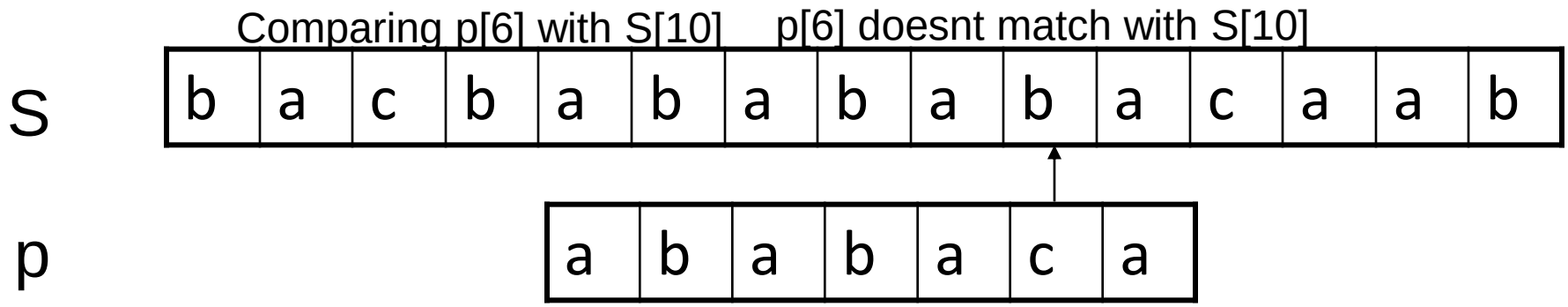
Step 8: i = 8, q = 3



Step 9: i = 9, q = 4

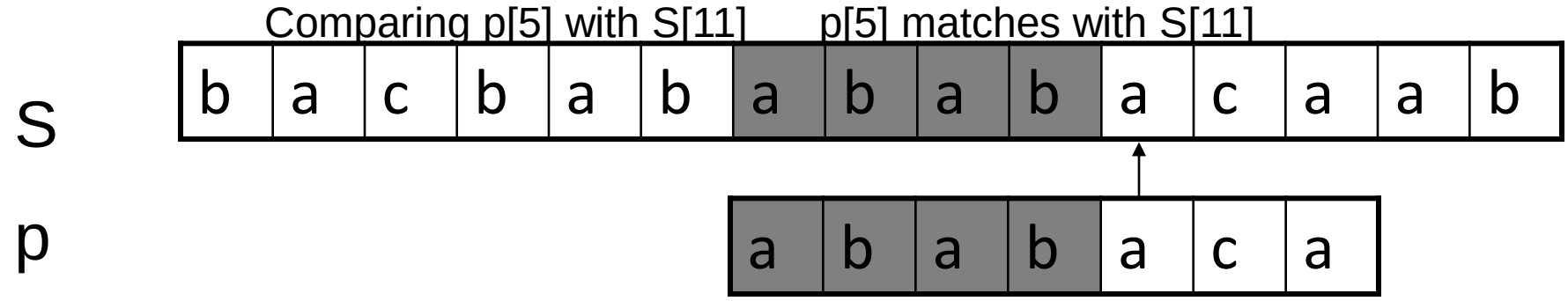


Step 10: i = 10, q = 5

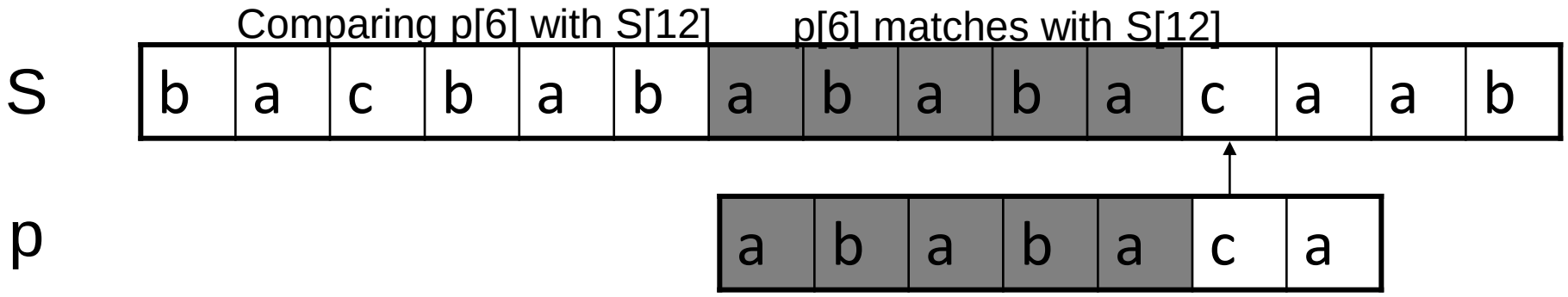


Backtracking on p, comparing p[4] with S[10] because after mismatch q = Π[5] = 3

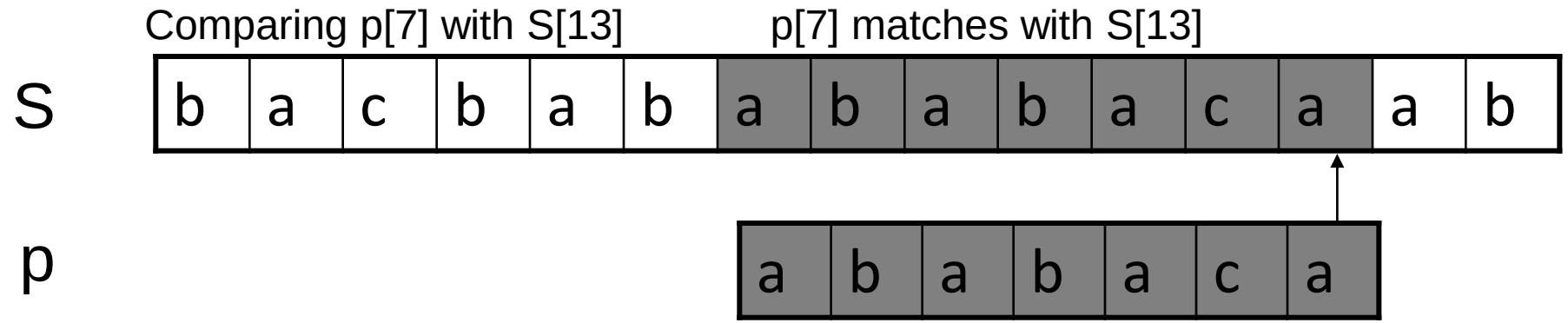
Step 11: i = 11, q = 4



Step 12: i = 12, q = 5



Step 13: i = 13, q = 6



Pattern p has been found to completely occur in string S. The total number of shifts that took place for the match to be found are: i – m = 13 – 7 = 6 shifts.

Running - time analysis

- Compute-Prefix-Function (Π)

```
1  m ← length[p]            //p pattern to be matched
2  Π[1] ← 0
3  k ← 0
4       for q ← 2 to m
5            do while k > 0 and p[k+1] != p[q]
6              do k ← Π[k]
7              If p[k+1] = p[q]
8                then k ← k +1
9              Π[q] ← k
10     return Π
```

In the above pseudocode for computing the prefix function, the for loop from step 4 to step 10 runs m times. Step 1 to step 3 take constant time. Hence the running time of compute prefix function is Θ(m).

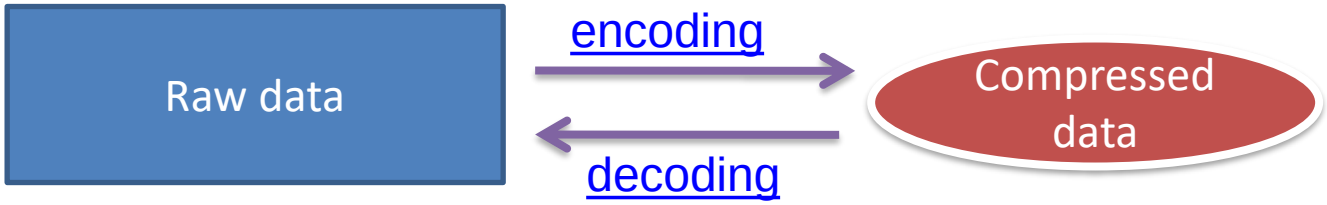
- KMP Matcher

```
1 n ← length[S]
2 m ← length[p]
3 Π ← Compute-Prefix-Function(p)
4 q ← 0
5 for i ← 1 to n
6   do while  q > 0 and p[q+1] != S[i]
7         do  q ← Π[q]
8     if p[q+1] = S[i]
9       then q ← q + 1
10    if q = m
11      then print "Pattern occurs with shift" i – m
12            q ← Π[ q]
```

The for loop beginning in step 5 runs n times, i.e., as long as the length of the string S. Since step 1 to step 4 take constant time, the running time is dominated by this for loop. Thus running time of matching function is Θ(n).

Data Compression - 1

- "In computer science and information theory, **data compression** or **source coding** is the process of encoding information using fewer bits (or other information-bearing units) than an unencoded representation would use through use of specific encoding schemes." (Wikipedia)
- Compression reduces the consumption of storage (disks) or bandwidth.
- However, it needs processing time to restore or view the compressed code.



Data Compression - 2

- Types of compression
  - *Lossy*: MP3, JPG
  - *Lossless*: ZIP, GZ
- Compression Algorithm:
  - Huffman Encoding
  - Lempel-Ziv
  - RLE: Run Length Encoding
- Performance of compression depends on file types.

Compress data by decoding symbols contained in it (lossless compression)

- The information content of the set *M,* called the **entropy** of the source X = $(x_1, x_2, \ldots, x_n)$, is defined by:

  $L_{ave} = H = P(x_1)L(x_1) + \cdots + P(x_n)L(x_n)$

  Where $L(x_i) = -\log_2 P(x_i)$, which is the minimum length of a codeword for symbol x_i (Claude E.Shannon, 1948). Shannons entropy represents an absolute limit on the best possible lossless compression of any communication.
- To compare the efficiency of different data compression methods when applied to the same data, the same measure is used; this measure is the **compression rate**

$$\frac{\text{length(input)} - \text{length (output)}}{\text{length(input)}}$$

Codeword:
sequence of bits of a code corresponding to a symbol.

Uniquely Decodable Codes

A **variable length code** assigns a bit string (codeword) of variable length to every message value

e.g. `a = 1, b = 01, c = 101, d = 011`

What if you get the sequence of bits `1011` ?

Is it `aba, ca,` or, `ad`?

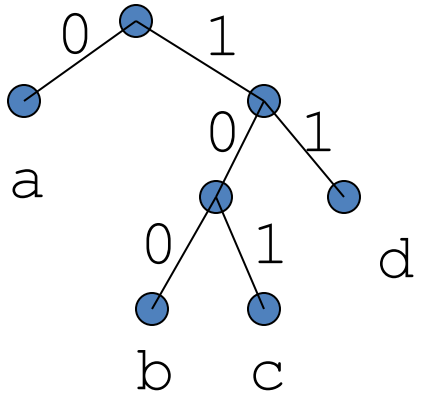
A **uniquely decodable code** is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

Prefix Codes

A **prefix code** is a variable length code in which no codeword is a prefix of another codeword

e.g a = 0, b = 110, c = 111, d = 10

Can be viewed as a binary tree with message values at the leaves and 0 or 1s on the edges.



Average Length

- For a code *C* with associated probabilities *p(c)* the **average length** is defined as

$$l_a(C) = \sum_{c \in C} p(c) l(c)$$

- We say that a prefix code *C* is **optimal** if for all prefix codes *C*, $l_a(C) \leq l_a(C)$
- The Huffman code is known to be provably optimal under certain well-defined conditions for data compression.

Huffman Coding algorithm

Main idea: Encode high probability symbols with fewer bits

1. Make a leaf node for each code symbol
   - Add the generation probability or the frequency of each symbol to the leaf node (arrange them from left to right in descending order by probability)
2. Take the two leaf nodes with the smallest probability and connect them into a new node
   - Add 1 or 0 to each of the two branches
   - The probability of the new node is the sum of the probabilities of the two connecting nodes
3. If there is only one node left, the code construction is completed. If not, go back to (2)

Huffman Coding example - 1

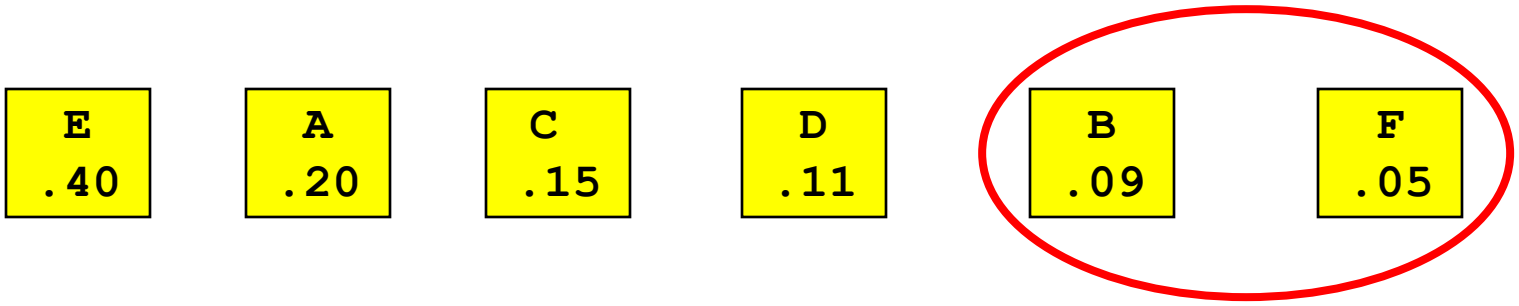
Character (or symbol) frequencies

- **A : 20% (.20)** *e.g., A occurs 20 times in a 100 character document, 1000 times in a 5000 character document, etc.*
- **B : 9% (.09)**
- **C : 15% (.15)**
- **D : 11% (.11)**
- **E : 40% (.40)**
- **F : 5% (.05)**

- Also works if you use **character counts**
- Must know frequency of every character in the document

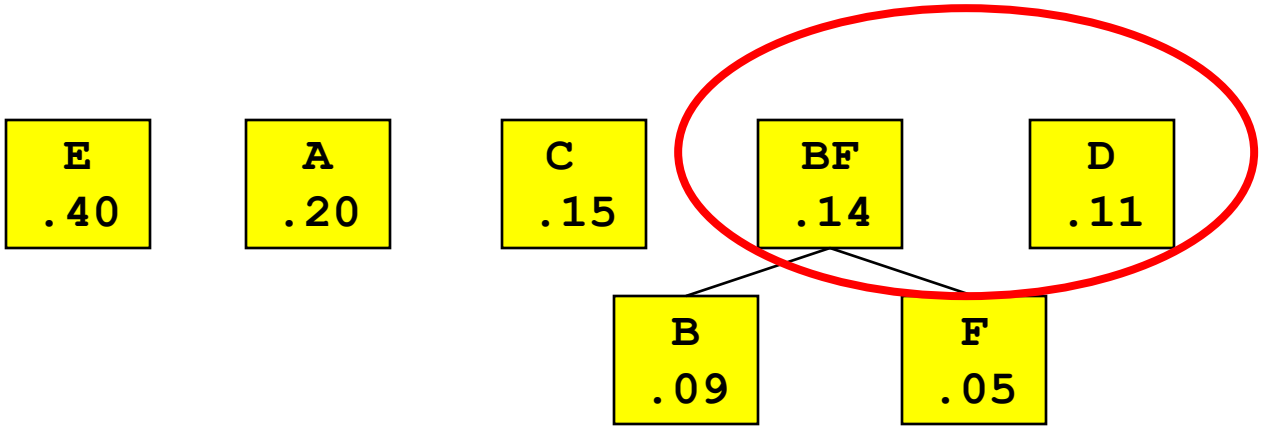
Huffman Coding example - 2

- Symbols and their associated frequencies.
- Now we combine the two least common symbols (those with the smallest frequencies) to make a new symbol string and corresponding frequency.



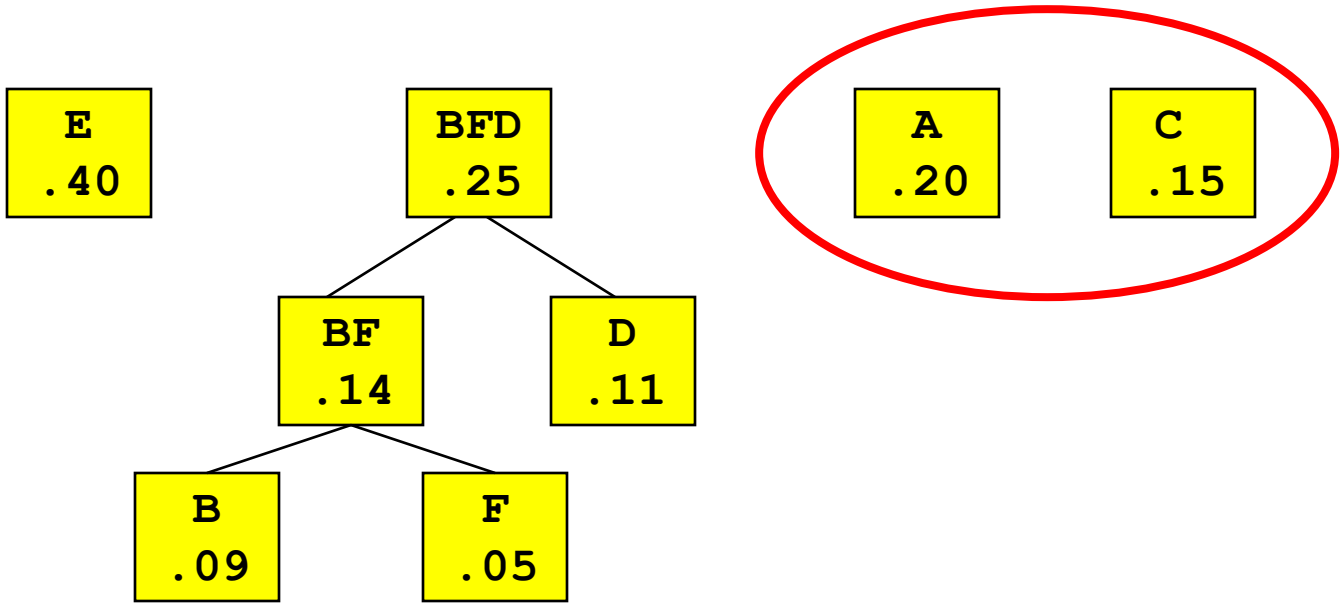
Huffman Coding example - 3

- Heres the result of combining symbols once.
- Now repeat until youve combined all the symbols into a single string.



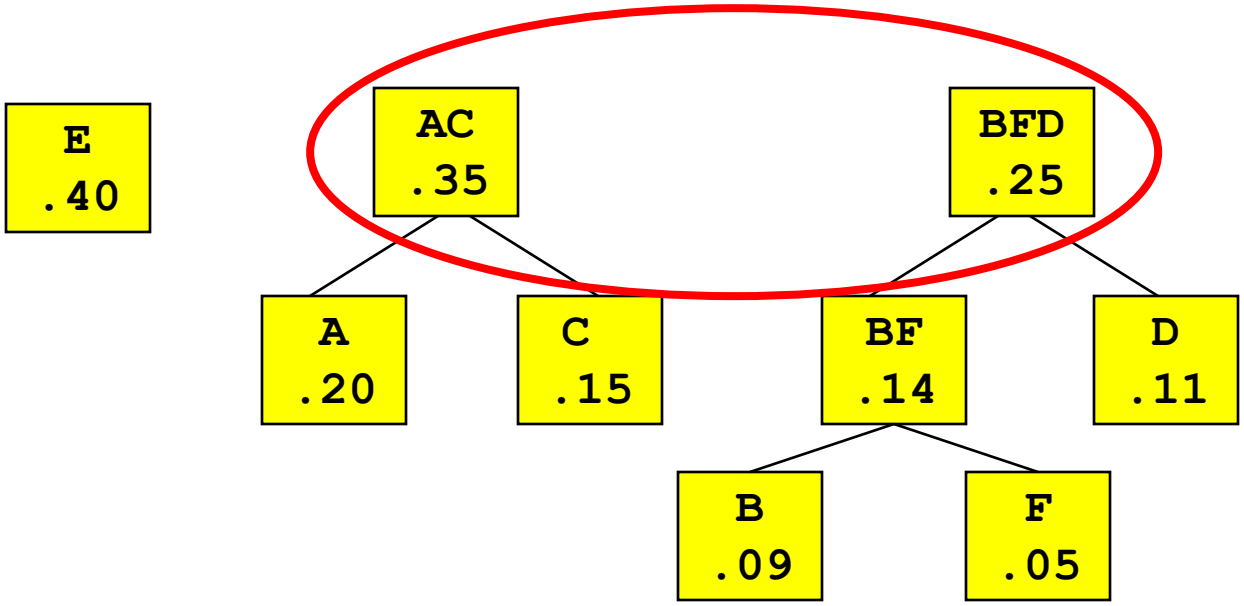
Huffman Coding example - 4

- Heres the result of combining symbols once.
- Now repeat until youve combined all the symbols into a single string.

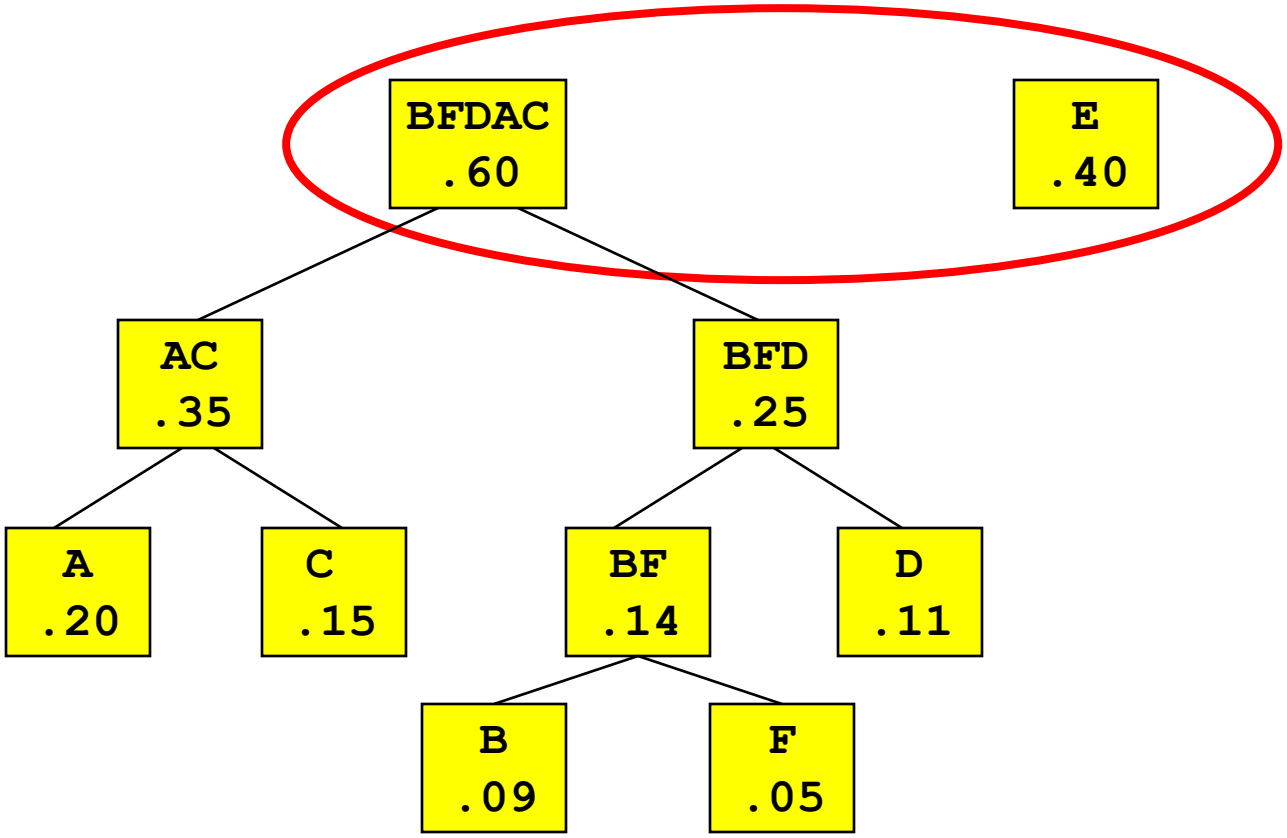


Huffman Coding example - 5

- Heres the result of combining symbols once.
- Now repeat until youve combined all the symbols into a single string.



Huffman Coding example - 6

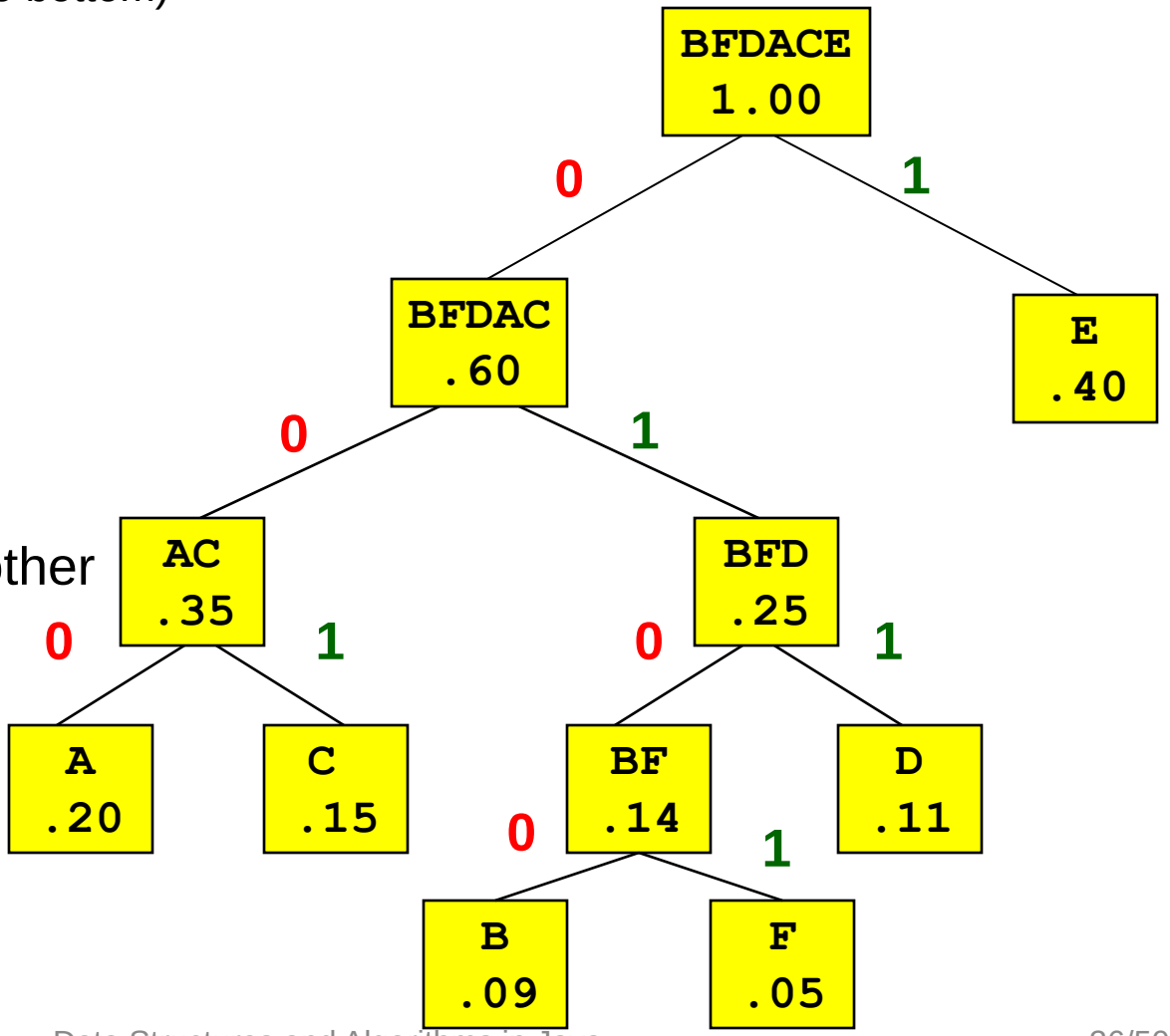


Huffman Coding example - 7

Codes: (reading from top to bottom)

A: 000
B: 0100
C: 001
D: 011
E: 1
F: 0101

Note
None are prefixes of another



Huffman Coding example - 8

Character	Code	Length	Probability
A	000	3	.20
B	0100	4	.09
C	001	3	.15
D	011	3	.11
E	1	1	.40
F	0101	4	.05

Average Code Length:

(3 x 0.20) + (4 x 0.09) + (3 x 0.15) + (3 x 0.11) + (1 x 0.40) + (4 x 0.05)

= 1.89 digits

Huffman Coding notes

- There is no unique Huffman code
  - Assigning 0 and 1 to the branches is arbitrary
  - If there are more nodes with the same probability, it doesnt matter how they are connected.
  - However, if the probability in each node is unique and the left nodes probability is always larger than the rights one, then the code is unique.
- Every Huffman code has the same average code length!

Some important statements in Huffman Encoding program

```
static final int R = 256;
// tabulate frequency counts
int[] freq = new int[R]; = new int[R];
for (int i = 0; i < input.length; i++) freq[input[i]]++;
// build Huffman tree
Node root = buildTree(freq);
// build code table
String [] st = new String[R]; buildCode(st, root, "");

// make a codewords table from symbols and their encodings
void buildCode(String[] st, Node x, String s)
  { if (!x.isLeaf())
      {buildCode(st, x.left, s + 0); buildCode(st, x.right, s + 1);  }
       else
         {st[x.ch] = s; }
  }
```

Lempel-Ziv Compression

- Encode sequences of symbols with location of sequence in a dictionary => dictionary coder
- Originated by Abraham Lempel and Jacob Ziv, improved by Tery Welch in 1984 (that is why it gets name LZW)
- This coding method is lossless coding
- Algorithms versions: LZ77, LZ78 and LZW

LZW Algorithm overview

The LZW algorithm stores strings in a "dictionary" with entries for 4,096 variable-length strings. The first 255 entries are used to contain the values for individual bytes, so the actual first string index is 256. As the string is compressed, the dictionary is built up to contain every possible string combination that can be obtained from the message, starting with two characters, then three characters, and so on.

```
LZWCompress()
   Enter all letters to the table;
   Initialize string s to the first letter from input;
   while any input left
      Read character c;
      if s+c is in the table
          s = s+c;
      else ouput codeword(s);
          Enter s+c to the table
          s = c
   Output codeword(s)
```

LZW Encoding Algorithm - 1

- A *Root* is a single-character string.
- Only code words are output. This means that the dictionary cannot be empty at the start: it has to contain all the individual characters (roots) that can occurr in the charstream;
- Since all possible one-character strings are already in the dictionary, each encoding step begins with a one-character prefix, so the first string searched for in the dictionary has two characters;
- The character with which the new prefix starts is the last character of the previous string. This is necessary to enable the decoding algorihtm to reconstruct the dictionary without the help of explicit characters in the codestream.

LZW Encoding Algorithm - 2

1. At the start, the dictionary contains all possible roots, and P is empty;
2. C := next character in the charstream;
3. Is the string P+C present in the dictionary?
    a. if it is, P := P+C (extend P with C);
    b. if not,
        i. output the code word which denotes P to the codestream;
        ii. add the string P+C to the dictionary;
        iii. P := C (P now contains only the character C);
    c. are there more characters in the charstream?
        - if yes, go back to step 2;
        - if not:
            i. output the code word which denotes P to the codestream;
            ii. END.

P is a current word, thus at the start, it is empty.

LZW Algorithm - Encoding process demo

Charstream to be encoded:

Pos	1	2	3	4	5	6	7	8	9
Char	A	B	B	A	B	A	B	A	C

- The column **Step** indicates the number of the encoding step. Each encoding step is completed when the step 3.b. in the encoding algorithm is executed.
- The column **Pos** indicates the current position in the input data.
- The column **Dictionary** shows the string that has been added to the dictionary and its index number in brackets.
- The column **Output** shows the code word output in the corresponding encoding step.

Contents of the dictionary at the beginning of encoding: **(1)A (2)B (3)C**

Step	Pos	Dictionary	Output
1.	1	(4) A B	(1)
2.	2	(5) B B	(2)
3.	3	(6) B A	(2)
4.	4	(7) A B A	(4)
5.	6	(8) A B A C	(7)
6.	--	--	(3)

The compressed output: **(1)(2)(2)(4)(7)(3)**

LZW Decoding Algorithm overview

Decoding: additional terms

- *Current code word*: the code word currently being processed. It's signified with cW, and the string it denotes with string.cW;
- *Previous code word*: the code word that precedes the current code word in the codestream. It's signified with pW, and the string it denotes with string.pW.

At the start of decoding, the dictionary looks the same as at the start of encoding -- **it contains all possible roots**.

Let's consider a point in the process of decoding, when the dictionary contains some longer strings. The algorithm remembers the previous code word (pW) and then reads the current code word (cW) from the codestream. It outputs the string.cW, and adds the string.pW extended with the first character of the string.cW to the dictionary.

LZW Decoding Algorithm

1. At the start the dictionary contains all possible roots;
2. cW := the first code word in the codestream (it denotes a root);
3. output the string.cW to the charstream;
4. pW := cW;
5. cW := next code word in the codestream;
6. Is the string.cW present in the dictionary?
   - if it is,
     i. output the string.cW to the charstream;
     ii. P := string.pW;
     iii. C := the first character of the string.cW;
     iv. add the string P+C to the dictionary;
   - if not,
     i. P := string.pW;
     ii. C := the first character of the string.pW;
     iii. output the string P+C to the charstream and add it to the dictionary (now it corresponds to the cW);
7. Are there more code words in the codestream?
   - if yes, go back to step 4;
   - if not, END.

LZW Algorithm - Decoding process demo

Lets analyze the step **4.** The previous code word (2) is stored in pW, and cW is (4). The string.cW is output ("A B"). The string.pW ("B") is extended with the first character of the string.cW ("A") and the result ("B A") is added to the dictionary with the index (6).

We come to the step **5.** The content of cW=(4) is copied to pW, and the new value for cW is read: (7). This entry in the dictionary is empty. Thus, the string.pW ("A B") is extended with its own first character ("A") and the result ("A B A") is stored in the dictionary with the index (7). Since cW is (7) as well, this string is also sent to the output.

Contents of the dictionary at the beginning of decoding: **(1)A (2)B (3)C**

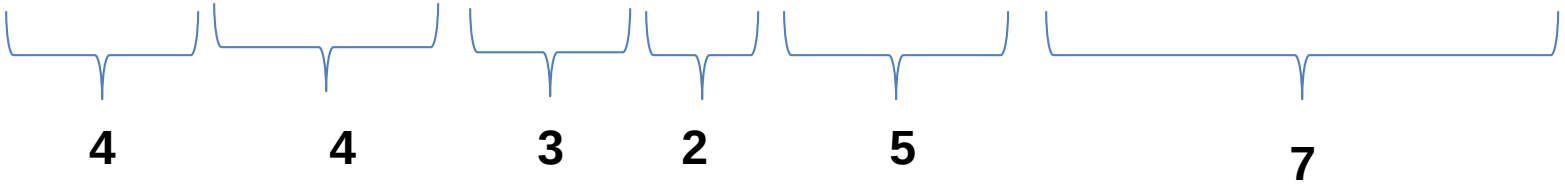
Table 2: The decoding process

Step	Code	Output	Dictionary
1.	(1)	A	--
2.	(2)	B	(4) A B
3.	(2)	B	(5) B B
4.	(4)	A B	(6) B A
5.	(7)	A B A	(7) A B A
6.	(3)	C	(8) A B A C

The decompressed output: ABBABABA

Run-Length Encoding

Raw : FFFFOOOOFFFOOFFFFFOOOOOOOO



Compressed: 4F4O3F2O5F7O

Compression Rate = (25-12)/25 = 52%

Summary

- Abundance of Digitized Text
- The problem of String Matching
- Brute-Force algorithm
- Knunth-Morris-Pratt Algorithm
- Data Compression
- Condition for Data Compression
- Huffman Coding Algorithm
- LZW Algorithm
- Run-length Encoding

FPT UNIVERSITY
DREAM OF INNOVATION

Reading at home

Text book: Data Structures and Algorithms in Java

- 13 Text Processing 573
- 13.1 Abundance of Digitized Text - 574
- 13.2 Pattern-Matching Algorithms - 576
- 13.2.1 Brute Force - 576
- 13.2.3 The Knuth-Morris-Pratt Algorithm - 582
- 13.4 Text Compression and the Greedy Method - 595
- 13.4.1 The Huffman Coding Algorithm - 596