FPT UNIVERSITY
DREAM OF INNOVATION

8. Text Processing

Objectives

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

FPT UNIVERSITY
DREAM OF INNOVATION

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

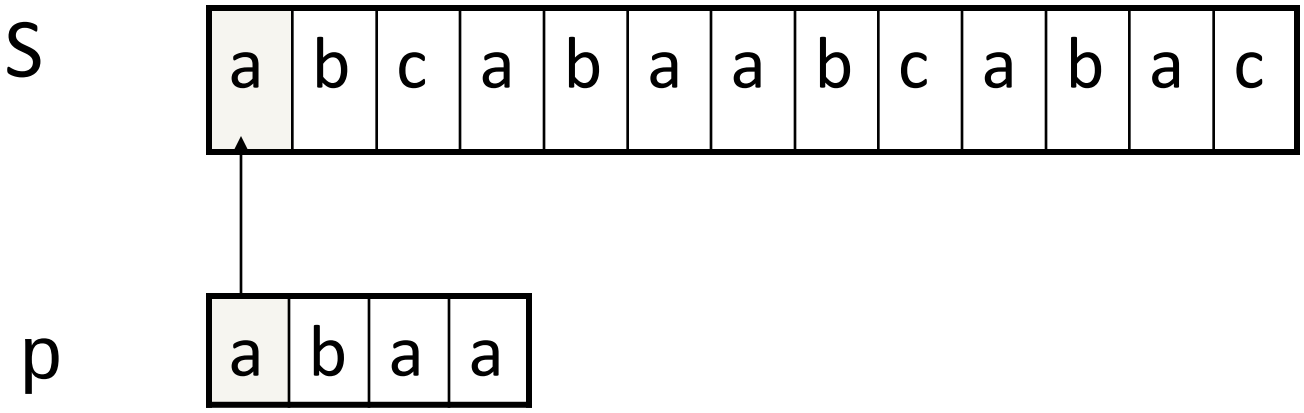
a	b	c	a	b	a	a	b	c	a	b	a	c

Pattern p

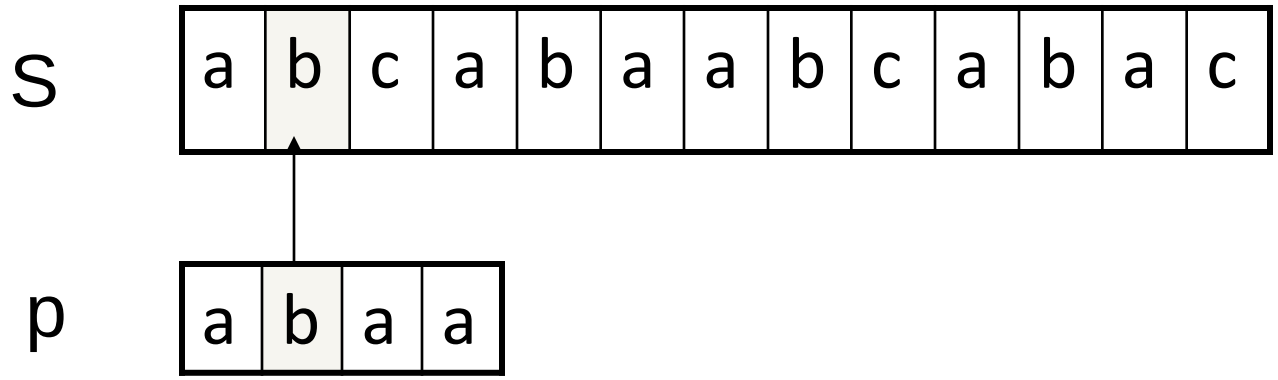
a	b	a	a

Brute-Force algorithm demo - 2

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

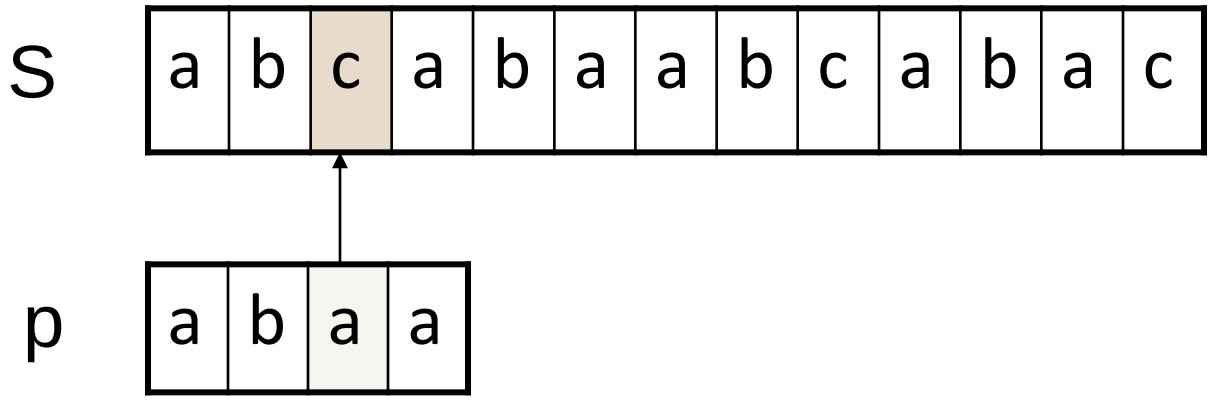


Step 2: compare p[2] with S[2]



Brute-Force algorithm demo - 3

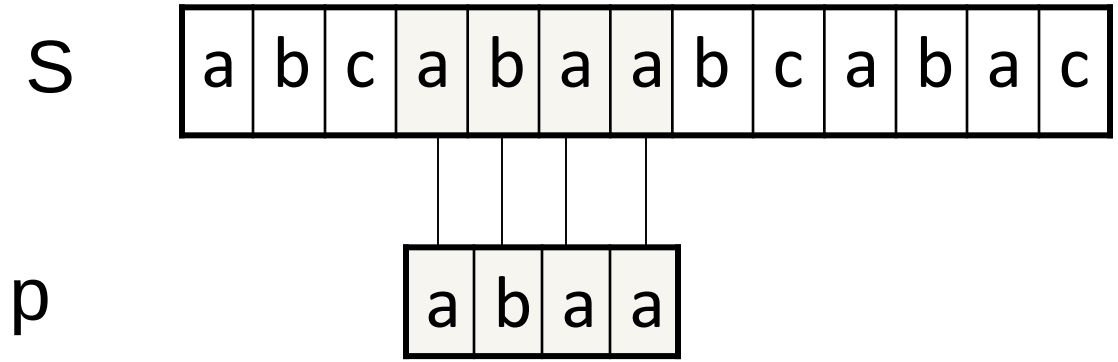
Step 3: compare p[3] with S[3]



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 (KMP) Algorithm - 1

- 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

The Knuth-Morris-Pratt Algorithm - 2

To illustrate the ideas of the algorithm, we consider the following example:

Index i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Text a	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**
Pattern p	1	0	1	0	1	1							
		1	0	1	0	1	1						
			1	0	1	0	1	1					

Thus the size of the text a n = 13 and the pattern's one is m = 6.
At a high level, the KMP algorithm is similar to the naive algorithm: it considers shifts in order from 0 to n-m, and determines if the pattern matches at that shift. The difference is that the KMP algorithm uses information gleaned from partial matches of the pattern and text to skip over shifts that are guaranteed not to result in a match.

The Knuth-Morris-Pratt Algorithm - 3

Index i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Text a	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**
Pattern p	1	0	1	0	1	1							
		1	0	1	0	1	1						
			1	0	1	0	1	1					

We call the starting position for a matching is r. Thus, for the first matching r = 0. We starting to compare a[i] with p[j], j = 0, 1, 2, ..., m-1; i = r+j. The result is:

a[0] = p[0], a[1] = p[1], a[2] = p[2], a[3] = p[3], a[4] ≠ p[4] (1)

Thus r = 0 is a wrong "starting position". The first wrong position is j = k = 4. By Brute-Force (BF) algorithm, we set r' = r+1 = 1 and starting new matching by comparing: (a[1],p[0]), (a[2],p[1]), (a[3],p[2]), (a[4], p[3]),...

However from (1) we can replace the above comparing with the following:

(p[1],p[0]), (p[2],p[1]), (p[3],p[2]), (a[4],p[3]),...

In the general case:

(p[1],p[0]), (p[2],p[1]), ..., (p[k-1],p[k-2]), (a[k],p[k-1]),...

The bold comparisions based on the pattern only and can be pre-done. These can be done by sliding a copy pp of the pattern to right 1 position. We can see that in the given example this comparision is wrong.

The Knuth-Morris-Pratt Algorithm - 4

Index i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Text a	**1**	**0**	**1**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**1**	**1**	**0**
Pattern p	1	0	1	0	1	1							
Slide 1		1	0	1	0	1	1						
Slide 2 (ok)			1	0	1	0	1	1					

Now we slide the pattern pp to right 2 positions and we can see the bold comparisions are matched. We can start to compare a[4] with p[2] only. **(p[2],p[0]), (p[3],p[1]),** (a[4],p[2]),...

For a given pattern p, we can precalculate the so called Knuth/Morris/Pratt ***failure function*** next(j) and stores them in an array T[j], where j = 0, 1, ..., m-1.

We define next(0) = -1, next(1) = 0. For j, 2 ≤ j ≤ m-1 the next(j) takes a value in an interval [0, j-1].

This value can be calculated by sliding a copied pp to right by some positions and comparing the values of pp with the p's values.

The Knuth-Morris-Pratt Algorithm - 5

In summary, a "step" of the KMP algorithm makes progress in one of two ways. Before the step, suppose that p[0,..,k-1] is already matched with a[r,..., r+k-1].
If p[k] = a[r+k], the length of the match is extended, unless k = m-1, in which case we have found a complete match of the pattern in the text.
If p[k] $\neq$ a[r + k], the pattern slides to the right to the index h = next[k]. (Thus we start to compare p[h] with a[r+k]).
In either case, progress is made. The algorithm repeats such steps of progress until the end of the text is reached.

Running Time
Each time through the loop, either we increase i or we slide the pattern right. Both of these events can occur at most n times, and so the repeat loop is executed at most 2n times. The cost of each iteration of the repeat loop is O(1). Therefore, the running time is O(n), assuming that the values (q) are already computed.

The KMP Algorithm examples

Example 1:

i	0	1	2	3	4	5	6
p[i]	A	B	C	D	A	B	D
T[i]	-1	0	0	0	0	1	2

Example 2:

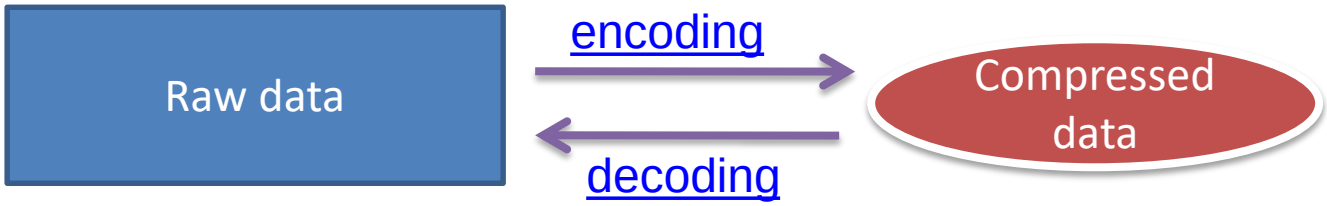
i	0	1	2	3	4	5	6	7	8
p[i]	A	B	A	C	A	B	A	B	C
T[i]	-1	0	0	1	0	1	2	3	2

Example 3:

i	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
p[i]	P	A	R	T	I	C	I	P	A	T	E		I	N		P	A	R
T[i]	-1	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	1	2

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`?

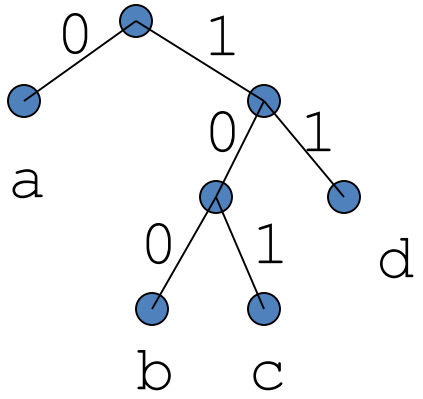
A **uniquely decodable code** is a variable length code in which bit strings can always be uniquely decomposed into its codewords.

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

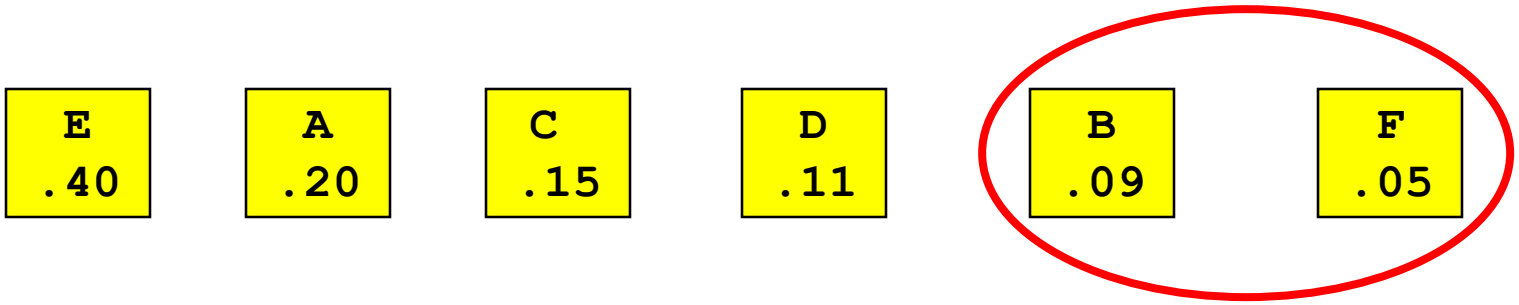
Character (or symbol) frequencies

- **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

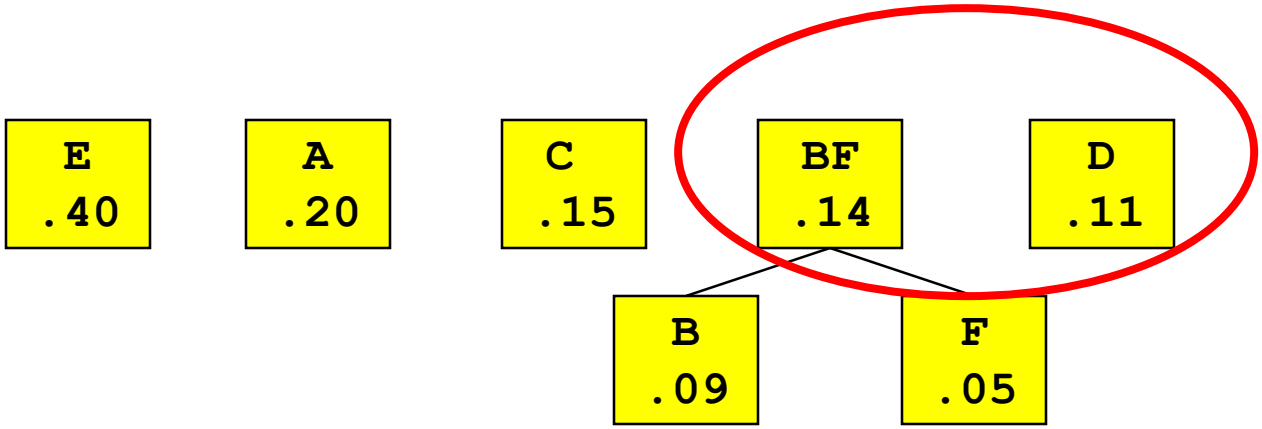
Huffman Coding example - 2

- 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.



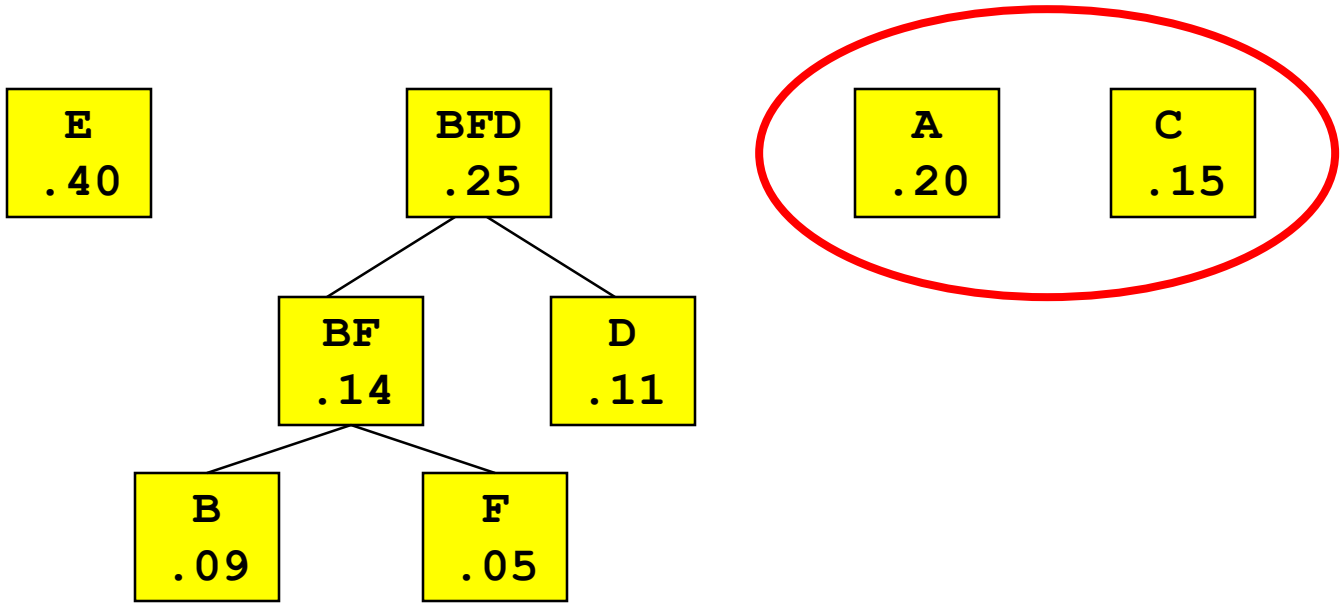
Huffman Coding example - 3

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



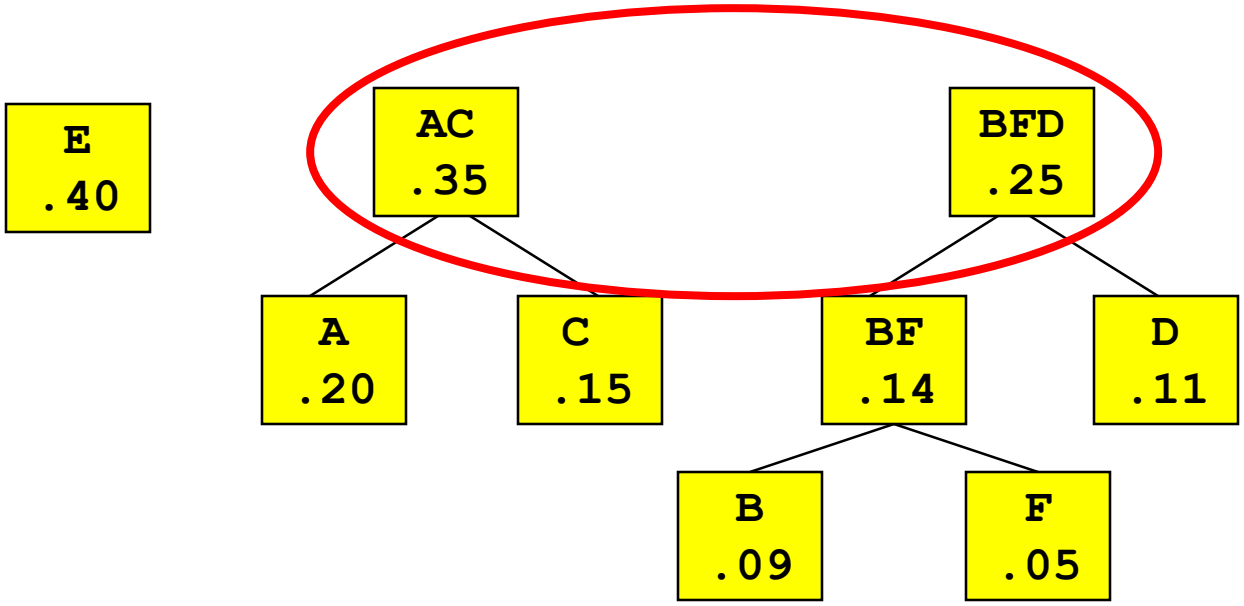
Huffman Coding example - 4

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

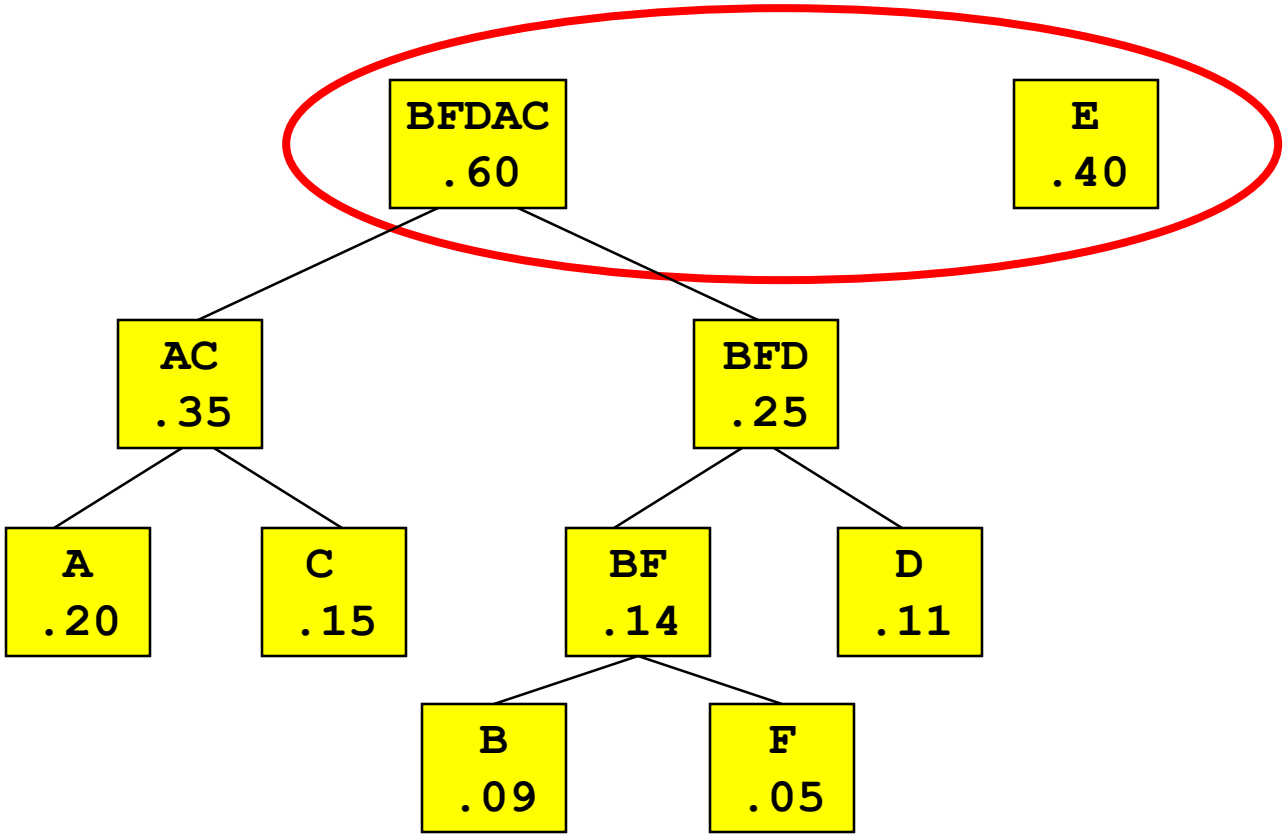


Huffman Coding example - 5

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



Huffman Coding example - 6

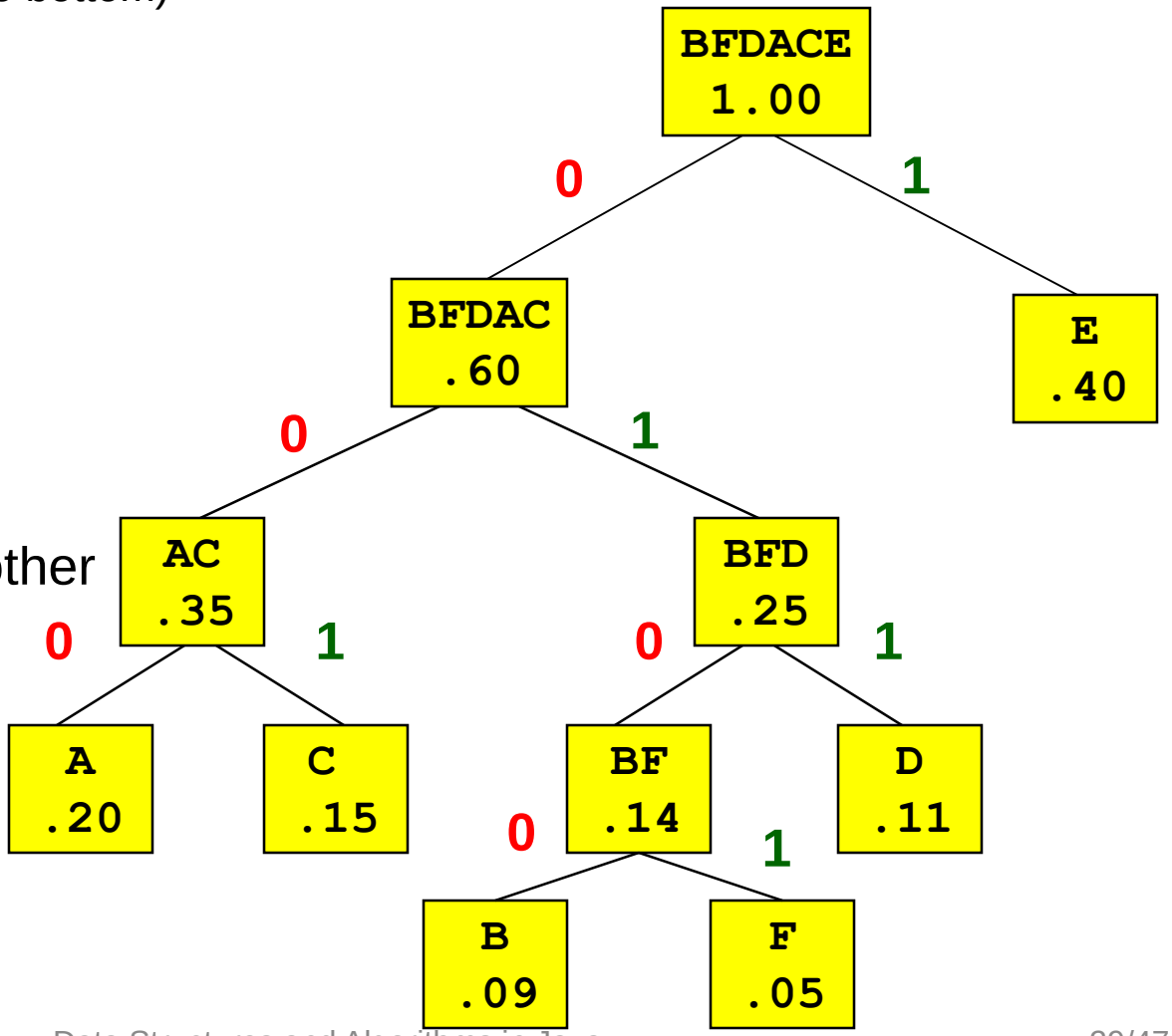


Huffman Coding example - 7

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 Encoding Algorithm

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.

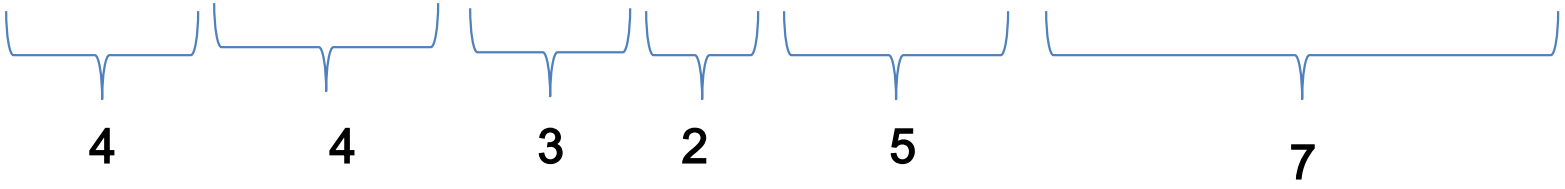
Contents of the dictionary at the beginning of encoding: **(1)A (2)B (3)C**

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)**

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
- Knuth-Morris-Pratt Algorithm
- Data Compression
- Condition for Data Compression
- Huffman Coding Algorithm
- LZW Algorithm
- Run-length Encoding

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

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

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

•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.

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