CS 251, Spring 2009
Voicu Popescu
Assignment 10
Due Monday April 20, 11:59PM

Pattern matching

1. Implement the Boyer-Moore and the KMP pattern matching algorithms
2. Define 5 alphabets of 2, 4, 8, 16, and 32 characters. Generate a random text of ten million characters for each alphabet. Generate random patterns of 4, 8, 16, 32, and 64 characters for each of alphabets used before. Search for the patterns in the texts with the two pattern matching algorithms and report the number of comparisons and the search result.
3. Extra credit 5%. Implement Huffman's encoding algorithm. Test it on a random ten million character text with characters from an alphabet with 32 characters. What is the length in bits of the Huffman encoding of the text?
4. Additional instructions:
    a. Implement the **Boyer-Moore (BM) algorithm** in a program file called "**bm.cpp**". Your program should read the text and the pattern from two different files. The names of files are given as arguments to the program (first argument is the file that includes the text and the second argument is the file that includes the pattern). The files have the data in one single line. Sample input files are provided. An example execution is as follows:

        ./bm bmsampletext bmsamplepattern

        Your program should search for the pattern in the text using Boyer-Moore algorithm and print the output to the **standard output** (screen) as follows (**DO NOT** write output to a file):
        i. Print the last function for the alphabet to the first line. Leave a space character between the values. Print only the values other than -1. Print the values in the order that their corresponding characters are alphabetically ordered (i.e. function value for A, then B, then C, etc.). You can assume the alphabet includes only ASCII characters.
        ii. Print the search result to the second line. If the pattern is found print the index where pattern starts in the text. Otherwise, print -1.
        iii. Print the number of comparisons to the third line.

        The sample output is provided in the file "bmsampleoutput". Please make sure that your program can EXACTLY reproduce that output given the input files. You may want to use the `diff' command to test this.

        ./bm bmsampletext bmsamplepattern > myoutput
        diff myoutput bmsampleoutput

    b. Implement the **Knuth-Morris-Pratt** (**KMP) algorithm** in a program file called "**kmp.cpp**". Your program should read the text and the pattern from two different files. The names of files are given as arguments to the program (first argument is the file that includes the text and the second argument is the file that includes the pattern). The files have the data in one single line. Sample input files are provided. An example execution is as follows:

        ./kmp kmpsampletext kmpsamplepattern

        Your program should search for the pattern in the text using KMP algorithm and print the output to the **standard output** (screen) as follows (**DO NOT** write output to a file):
        i. Print the failure function for the pattern to the first line. Leave a space character between the values. Print the values in the order that their corresponding characters are the characters in the pattern (i.e. function value for first character of pattern, then second character of the pattern, then third, etc.).

ii. Print the search result to the second line. If the pattern is found print the index where pattern starts in the text. Otherwise, print -1.
iii. Print the number of comparisons to the third line.

The sample output is provided in the file "kmpsampleoutput". Please make sure that your program can EXACTLY reproduce that output given the input files. You may want to use the `diff' command to test this.

```
./kmp kmpsampletext kmpsamplepattern > myoutput
diff myoutput kmpsampleoutput
```

c. **Part 2**: Report your results as a **table** in a pdf document named as "**results.pdf**". A sample table may be as in the last page of this document.

d. **Extra credit:** Implement **Huffman's encoding algorithm** in a program file called "**huffman.cpp**". Your program should read the text from a file and write the encoded text to an output file. The names of files are given as arguments to the program (first argument is the file that includes the text to be encoded and the second argument is the name of the output file where you will write the encoded text). The files have the data in one single line. An example execution is as follows:

```
./huffman text encodedtext
```

Your program should encode the text using Huffman's encoding algorithm and write the output to an output file with the given name.

Answer the question in Part 3 in a pdf document named as "**huffman.pdf**". Include any explanations, clarifications, your design and implementation choices into this pdf file in order to get a full credit.

e. Be sure to test your programs thoroughly.
f. Please use the Makefile provided (you may modify it only for extra credit part).
g. Do not partition your code into files we haven't asked for.
IMPORTANT: Any files other than bm.cpp, kmp.cpp and huffman.cpp (optional) WILL NOT BE GRADED!

5. Turn in instructions
   a. Your codes MUST compile on lore.cs.purdue.edu for it to be graded.
   b. All of your documents (bm.cpp, kmp.cpp, results.pdf, huffman.cpp (OPTIONAL) and huffman.pdf (OPTIONAL)) MUST be in the directory A10.
   In the directory above A10, run the command on lore.cs.purdue.edu

```
turnin -c cs251 -p A10 A10
```

optionally run the following command to verify what files were submitted

```
turnin -c cs251 -p A10 –v
```

Warning: turnin will be automatically disabled at 04/20/2009 at 11:59 PM EST and you will be unable to turnin your assignment after this time.

Warning: turning in multiple times is ok but only the last one will be graded as each turnin permanently overwrites the previous one.

Warning: failure to turnin exactly according to the instructions given above will result in your A10 receiving a grade of 0.

A sample table for Part 2:

			BM		KMP	
alphabet size	**text size**	**pattern size**	**# comparisons**	**search result**	**# comparisons**	**search result**
2	10 million	4				
		8				
		16				
		32				
		64				
4	10 million	4				
		8				
		16				
		32				
		64				
8	10 million	4				
		8				
		16				
		32				
		64				
16	10 million	4				
		8				
		16				
		32				
		64				
32	10 million	4				
		8				
		16				
		32				
		64				