2013-4-27

1. Examination Time: 18:30-20:30
2. NOTE: Please carefully read the comments in the attached source code, where some requirements/tips are given.
3. Submission time: Please start to submit your source code by Tsinghua Web Learning (网络学堂) at least before 20:30. The submission site will be closed at 20:40. Late submissions are NOT accepted.
4. Submission requirement: Please follow the rules for our homework submission.

1. Code **Reuse** (Credits 40%)

We have emphasized (强调) code reuse many times in class, which is a basic programming concept that you are highly suggested to bear in mind. Code reuse is important because it helps you to improve your programming efficiency and your code's quality. Actually, the idea of reuse is also important in improving the program's efficiency, i.e., reduction of the program's runtime (程序运行时间). We will introduce a classic KMP string matching algorithm (字符串匹配) to show how the idea of reuse is so important.

The KMP algorithm was published by Donald Knuth (author of "The Art of Computer Programming"), Vaughan Pratt, and James H. Morris in 1977 (courtesy of Wikipedia). A string matching algorithm wants to find the starting index m in string (character array) A[] that matches the search word (pattern) P[]. A straightforward algorithm simply tries successive values of m until it finds a match or fails. Each trial (匹配检验) involves using a loop that checks A[m+i]==P[i] for each character P[i] in the search pattern. Assume (假设) the length of the searching string A[] is k and the length of the pattern P[] is n, then the worst case performance is $O(k \cdot n)$, i.e., k·n comparisons (k 乘 n 次比较运算) are needed to find all the matches. In contrast, the KMP algorithm runs in $O(k+n)$, which is much faster. The key idea of KMP is to

reuse previous match information for the next trial. To reuse the previous match information, Knuth et al. presented to preprocess the pattern P[] for computing a table to be used during the string searching in A[], which avoids the redundant comparisons (冗余比较) for each character in A[]. Details of the KMP algorithm are covered in Problem 4. Now the KMP algorithm is implemented and compiled into a library (KMP.a). However, the provided KMP algorithm only finds the first match in A[]. Please reuse the provided class "KMP" by "public composition", "private composition", "public inheritance" and "private inheritance" respectively (分别使用以上四种复用方法) to compute all the matches.

Requirements:

(1) The input for matching is 0-1 arrays instead of character strings (examples are given in "pattern.txt" and "array.txt").

(2) The provided input files ("pattern.txt" stores P[] and "array.txt" stores A[]) are subject to change, i.e., you CANNOT assume the pattern and array are fixed either in size or contents.

(3) All the four class reuse methods are required. Please check "main.cpp" for the expected coding task.

(4) You may add new files or add new codes, but you MUST NOT delete any code in the provided files.

2. Check the program's efficiency with reuse (Credits 30%)

Implement (实现) the straightforward algorithm described in Problem 1 (or any algorithm you come up with instead of KMP, that can find the correct matches)

using OOP design style. First randomly generate (生成) (TIPS: using function rand(). You may need to include the header file "cstdlib") a 0-1 pattern of length 10 and stores it into file "pattern.txt", and randomly generate a 0-1 array of length 10,000 to 10,000,000,000 (you may change the length such that the total runtime is suitable for comparison) and stores it into file "array.txt". Then load in (读入) the generated files for matching. Implement any one of the four class-reuse methods in Problem 1 ("public composition", "private composition", "public inheritance" and "private inheritance") (采用四种方法之一). Measure the runtime of the straightforward algorithm (t1) and the class-reuse method (t2), then compute and print the runtime improvement (i.e., (t1-t2)/t1).

Requirements:

(1) Please use OOP design to implement your method, i.e., better data abstraction, encapsulation, etc. The method MUST NOT be implemented with just a function.

(2) Use QueryPerformanceFrequency and QueryPerformanceCounter to measure the runtime.

(3) Do NOT measure the runtime of generating and loading "pattern.txt" and "array.txt".

(4) Only a single program is allowed, i.e., the different algorithms are compiled and linked into one program.

3. Better code reuse by polymorphism (Credits 30%)

Sometimes, it is possible that one only wants to find the first *i* matches in the array. A good design of the base class will make it very easy for the derived class to reuse and to terminate the searching process when the first *i* matches are

found. Check and examine carefully the attached code to see how to complete the files and functions to find the first *i* matches.

Requirements:

(1) The number of matches *i* is input by command-line arguments.

(2) In the base class "KMPBase", you may either use your own code implemented in Problem 2 or use the KMP library (by including "KMP.h" and linking "KMP.a") to find the matches. You need to add some member functions and member variables in class "KMPBase" for the matching process. And the provided makefile may need to be modified correspondingly.

(3) Check the comments in the source files for more requirements.

4. Bonus (Credits 10%) (加分題)

The key idea of the KMP algorithm is given in the attached document "KMPAlgorithm.pdf" (courtesy of Professor Holger H. Hoos at the Computer Science Department of the University of British Columbia (UBC) in Vancouver, BC, Canada.). Please implement the KMP algorithm according to the description using OOP design style, and compare the runtime of your implementation with the provided KMP library (KMP.a). Please see the requirements for runtime comparison in Problem 2.

Requirements:

(1) Please implement the algorithm by yourself even you have "materials" for this. Just the bonus points do not deserve giving up one's personality.