Data Structure and Algorithm

Homework #6

Due: 5pm, Friday, June 14, 2013

TA email: dsa1@csie.ntu.edu.tw

=== Homework submission instructions ===

- For Problem 1, submit your source codes, a Makefile to compile the source, and a brief documentation to the SVN server (katrina.csie.ntu.edu.tw). You should create a new folder "`hw6`" and put these three files in it.
- The filenames of the Makefile, and the documentation file should be "`Makefile`" and "`report.txt`", respectively; you can use any filenames for the source codes, but the filename of the executable file which is generated after executing "`make`" command should be "`main.exe`". You will get some penalties in your grade if your submission does not follow the naming rule. The documentation file should be in plain text format (`.txt` file). In the documentation file you should explain how your code works, **the reference**, and anything you would like to convey to the TAs.
- For Problem 2 through 5, submit the answers via the SVN server (electronic copy) or to the TA at the beginning of class on the due date (hard copy). Please hand in your homework with A4 paper.
- Except the programming assignment, each student may only choose to submit the homework in only one way; either all in hard copies or all via SVN. If you submit your homework partially in one way and partially in the other way, you might only get the score of the part submitted as hard copies or the part submitted via SVN (the part that the grading TA chooses).
- If you choose to submit the answers of the writing problems through SVN, please combine the answers of all writing problems into only ONE file in the pdf format, with the file name in the format of "`hw6_[student ID].pdf`" (e.g. "`hw6_b01902888.pdf`"); otherwise, you might only get the score of one of the files (the one that the grading TA chooses).
- Discussions with others are encouraged. However, you should write down your solutions by your own words. In addition, for each problem you have to specify the references (the Internet URL you consulted with or the people you discussed with) on the first page of your solution to that problem, but on `report.txt` for the programming assignment.
- **NO LATE SUBMISSION IS ALLOWED** for the homework submission in hard copies - no score will be given for the part that is submitted after the deadline. For submissions via SVN (including the programming assignment and electronic copies of the writing problems), up to one day of delay is allowed; however, the score of the part that is submitted after the deadline will get some penalties according to the following rule (the time will be in seconds):

$$\text{LATE SCORE} = \text{ORIGINAL SCORE} \times (1 - DelayTime/86400)$$

Problem 1. Suffix Tree (25%)

In computer science, a suffix tree is a data structure that presents the suffixes of a given string in a way that allows for a particularly fast implementation of many important string operations.

The following presents a related and common string matching problem:

Given a long string $S[1..n]$, you are asked to find the longest common prefix of 2 suffixes of S.

Let $p_k(S) = S[1..k]$ and $s_k(S) = S[k..n]$. Then $p_k(S) \sqsubset S$ and $s_k(S) \sqsupset S$. The longest common prefix of two strings A and B is $p_k(A) \mid k = \text{argmax}_{0 \le i \le \min(|A|,|B|)} p_i(A) = p_i(B)$.

For example, given the string "eabcaba", "eabc" is one of its prefixes, and "aba" is one of its suffixes. When you are given a query with $a = 2$, $b = 5$ and $T =$ "eabcaba", you should find the longest common prefix of $s_2(T)$ ("abcaba") and $s_5(T)$ ("aba"). Both "a" (satisfying $p_1(s_2(T)) = p_1(s_5(T))$) and "ab" (satisfying $p_2(s_2(T)) = p_2(s_5(T))$) are the common prefix of $s_2(T)$ and $s_5(T)$, but "ab" is the longest and is the solution.

One day, you found that there is a bug in the open source suffix tree library. To contribute to the open source community, you would like to resolve this problem and submit the fix to the repository.

Input Format

The test data will be given in the following format.

The 1st line contains an integer N, followed by the description of N sub-problems.

For each sub-problem description, the 1st line contains an integer $n = |S|$, the length of string S.

The 2nd line contains a string S. All characters in S are lowercase letters.

The 3rd line contains an integer q, the number of queries in this sub-problem.

In the next q lines, the i-th line contains 2 integers a_i and b_i which are separated by a space character. This indicates that the i-th query is to find the longest common prefix of $s_{a_i}(S)$ and $s_{b_i}(S)$.

Output Format

For each query in each sub-problem, your program should output an integer indicating *the length* of the longest common prefix in one line.

Grading

- 3 points for the report, including the reference.

  Remember to put down the reference(s) of *Problem 1* in `report.txt`.

- 2 points for Makefile, including whether the compilation of your source codes is successful.

- 2 points for each set of test data; 20 points in total.

  The execution time limit for each set of test data is 20 seconds.

The following are conditions that you can assume for the test data.

- For 1 set of test data, $n \leq 500$, $q \leq 250000$.
- For 1 set of test data, $n \leq 2000$, $q \leq 2000$.
- For 1 set of test data, $q \leq 500$.
- For 2 sets of test data, $q \leq 2000$.
- For 2 sets of test data, S is generated randomly.
- For all test data, $N \leq 10$, $n \leq 200000$, $1 \leq a, b \leq n$, $q \leq 400000$.

Sample Input

```
2
3
aba
3
1 1
1 2
1 3
7
eabcaba
2
2 7
2 5
```

Sample Output

```
3
0
1
1
2
```

Hints and Technical Reminders

- The suffix tree could be built in $O(n)$, and by using the Range Minimum Query (RMQ) data structure, each query could be processed in $O(1)$.

  However, we do NOT expect you to use this approach to solve the problem. The main reason for mentioning the suffix tree is to motivate you to read about this powerful data structure by yourself. Instead, utilize what we have covered in the lecture to solve this problem.

- Hint: do a little pre-processing in $O(n)$ time and then you will be able to test whether $S[l..r]$ and $S[l'..r']$ are the same in $O(1)$ time.

  Come up with a "hash function", i.e., a way to use a number to represent a string. If you do not have any idea, solve the *Problem 2* first.

- You may find that you cannot distinguish the cases of that two strings are the same and that it is just a hash collision (two different strings have the same hash function output value). But if your hash function include a step that modulo the original output with a sufficiently large number, the collision probability is much lower.

  For more information, you may search for "birthday problem" or "birthday paradox" on the Internet.

- If you already know string A and string B have a common prefix of length k, then any prefix of either string of length $i \leq k$ is also a common prefix of these two strings. In addition, if you know a prefix of either string of length k is not a common prefix, then any prefix of either string of length $i \geq k$ is also not a common prefix.

  Shorter mathematical formulations: $p_k(A) = p_k(B)$ implies $p_i(A) = p_i(B)$ for any $i \leq k$ and $p_k(A) \neq p_k(B)$ implies $p_j(A) \neq p_j(B)$ for any $j \geq k$.

  Try to derive a $O(\lg n)$ algorithm to find the longest common prefix, utilizing the algorithm to determine whether two sub-strings are the same or not in $O(1)$ time (from the second hint).

- The length of readable official solution contains about 90 lines of C code.

- Each test data can be solved in 7 seconds by the official solution without any compiler optimization on CSIE 217 workstations.

Problem 2. Rabin-Karp Tricks (20%)

2.1. (5%) Given a string T with only digits (characters '0' - '9'). Suppose we use the Rabin-Karp hash value with moduler 113 and 10 as the base (the number of different characters in T is 10, obviously). The hash values for the two prefixes of T are $T[1..12] = 100$ and $T[1..6] = 50$. Please calculate the Rabin-Karp hash value of $T[7..12]$.

2.2. (5%) Given a string T consisting of K different kinds of characters. Suppose we have the Rabin-Karp hash value with moduler p and K as the base. The hash values for the two prefixes of T are $T[1..m] = M$, and $T[1..n] = N$ $(n < m)$. Please calculate the Rabin-Karp hash value of $T[n+1..m]$.

2.3. (10%) Give a string that will collide with your school ID number (ex. "b01902999") using the Rabin-Karp hash with moduler 91 and 36 as the base.

Hint: Why is 36 the base? Because there are 36 different characters in the string: ($0 \to 0, 1 \to 1, \ldots, a \to 10, b \to 11, \ldots, z \to 35$).

Problem 3. Hash Function Designing (15%)

There are many applications of hashing. You have learned a few different hashing functions for strings and numbers in the lecture. In this problem, you are asked to design a few special hash functions.

Your score is based on the following criterion:

- Correctness, which is the most basic requirement.
- The ability to avoid collisions. For example, if we use summation of characters of a string, which is a word or a sentence written in English, as the hash function, then two strings with same characters set always have same hash value even if the permutation is different (e.g., algorithm and logarithm). But if we shift each character before adding them together, English strings have different hash value in most cases. Thus the second hash function is better.
- Time complexity. Extremely bad performance is not allowed.

3.1. (5%) Given two bitmaps S, P. A bitmap is a 2-dimensional integer array with each number representing the color a pixel in the 2-dimensional space). Both the width and the height of P is less than the size of S. We want to check if P occurs in S by testing each possible position. Similar to string matching, we want to solve this problem in $O(|S|)$ time, where $|S|$ = the width of $S \times$ the height of S. Thus your hash function needs to be calculated with amortized time complexity $O(1)$ (i.e., calculating the hash function values for all positions in $O(|S|)$).

Here is an example:

S

0	0	8	6	0
0	2	4	2	1
2	6	6	3	1
1	8	4	5	1
0	1	1	1	0

P (hash value = 135)

6	3	1
4	5	1

hash value at each position of S

—	—	—	—	—
—	—	321	816	621
—	—	201	882	172
—	—	437	371	135
—	—	451	966	903

3.2. (5%) Given two sets of 2-dimensional coordinates, we are wondering that if these two sets of coordinates are entirely the same after we scale and/or shift one set of coordinates. For example, $\{(5,3),(4,4),(1,2)\}$ is the same as $\{(6,6),(8,4),(0,2)\}$. Note that the order of the coordinates in a set is not important. Design a hash function which can be applied to check that if two sets of coordinates are the same by comparing their hash values. The input of your hash function is a set of b 2-dimensional coordinates $P_1, P_2, \ldots P_n$ and all the coordinates are integers.

3.3. (5%) Given a rooted binary tree as the input of the hash function. Each tree node is associated with an integer. Design a hash function that can be used to check if two trees are equal. Note that two trees are considered equivalent if we can transform one to the other with a sequence of operations that swap the two children of a node. The operation can still be performed even when one of the children is empty; in this case, the only child node is switched to different side (i.e., left child $\rightarrow$ right child and right child $\rightarrow$ left child).

Problem 4. Multi-Pattern String Matching (20%)

In the lecture, we have learned efficiency KMP algorithm which can be used to search for pattern string P in string T in $O(|T|)$ time and pre-processing requires additional $O(|P|)$ time. However, in some real-world cases, there could be more than one pattern strings that need to be searched for. For example, profanity filtering is an useful and common application of the string matching algorithm but there are a large number of profanities to be filtered. In this problem, we will study how to efficiently search for K pattern strings $P_1, P_2, \ldots, P_K$ in string T.

4.1. (5%) Design a naïve brute-force algorithm to search for all occurrences of the given K pattern strings in string T in $O(|T| \sum_{i=1}^{K} |P_i|)$ time. The algorithm should be written in pseudo code or C code. Briefly show the correctness of your code and show that it indeed runs in $O(|T| \sum_{i=1}^{K} |P_i|)$.

4.2. (5%) Apply the KMP algorithm to accelerate your algorithm to $O(K|T|)$ and $O(\sum_{i=1}^{K} |P_i|)$ for pre-processing. Briefly argue the correctness of your code and show that it indeed runs in the specified time complexities.

4.3. (5%) Do you remember the dictionary tree in Problem 4 of HW2? No? Then it is time to refresh the memory. In the original single-pattern string matching problem, we treat R as a dictionary tree contain only one pattern string P. The prefix function of each position can be represented as a *back-edge* in the dictionary tree R from that character to the character at the location indicated by the value of the prefix function. The matching process of KMP can then be treated as a traversal on R. For example, Fig. 1 represents the dictionary tree R of a pattern string "*abbabb*". Each *back-edge* represented by a red dashed line shows the prefix function value of a character in the pattern string. The matching process is a traversal on this tree: if we match a character successfully, we will go through the only tree edge to the next character; otherwise, we go through the *back edge* to a preceding character. Given the pattern string $P = abababba$ to be searched for and the string $T = abababbaa$ to be searched, draw the structure of the dictionary tree R corresponding to P. Perform the matching process with a tree traversal on R and draw the traversal.

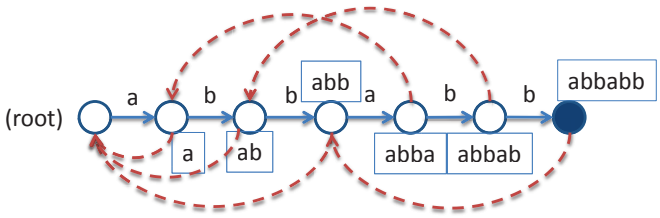

Figure 1: An example of mapping the KMP prefix function to a dictionary tree

4.4. (5%) Consider the multi-pattern string matching problem now. Your are given K patterns $P_1, P_2, \ldots, P_K$ to be searched for in T. None of these K pattern strings is a substring of any of the other pattern strings. For example, string *"a"* is a substring of string *"caa"*, so *"a"* and *"caa"* cannot both be the pattern strings.

The extension of the algorithm in Problem 3.3 can be used to match multiple patterns (with the stated assumption) in an efficient way. Figure 2 shows an example of using the dictionary tree for multi-pattern string matching. Two patterns are given "*abcd*" and "*bcabc*" in the example in Fig. 2. The definition of the prefix function and the *back-edge* are the same as that in the single-pattern string matching problem.. For each character, the *back-edge* links to the node representing the longest suffix of itself. For example, node "*bcab*" links to the node "*ab*" because the node "*cab*" does not exist.

With the stated assumption, it can be shown that none of the *back-edge* leads to a node representing a pattern string. The matching process can be treated as a tree traversal, the same as in Problem 3.3.

In this problem, you are asked to design an algorithm to build the dictionary tree from the given K pattern strings. In addition, design an algorithm to use the derived dictionary tree to perform multi-pattern string matching. Your algorithm should run in $O(|T|)$ time for matching and in $O(\sum_{i=1}^{K} |P_i|)$ time for pre-processing. The algorithms should be written in pseudo code or C code. Briefly show the correctness of your code and show that it indeed runs in the specified time complexities.

Hint #1: The BFS algorithm can help you to build the *back-edges* of the dictionary tree.

Hint #2: The target of a node's *back-edge* is relevant to the target of its parent node's *back-edge*.

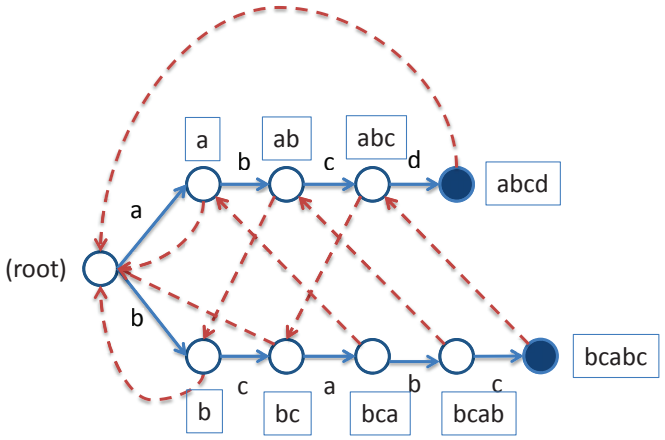

Figure 2: An example of using the dictionary tree for multi-pattern string matching.

Problem 5. Magic Spell (20%)

Albert's family are very religious, so Albert is asked to say a special 'spell' every morning. This spell - said by his mother - can help Albert to become healthier and luckier. Albert hates this tedious spell, which just repeats some letters over and over. So he decides to use a recorder to record the first few letters (i.e., the prefix) of the spell. Whenever he is asked to say this spell again, he will play this tape for several times.

For example, if the spell is 'pppoippppoippp', he can record 'pppoippppoi' on the tape. Then he can play the tape for one time and stop at the third letter when playing for the second time. There is also a better way to do this. He can record 'pppoip' on the tape, and will only need to say 6 letters overall! As you've seen, Albert really really hates this spell, so he wonders that how many different strings s can be recorded on the tape to create the spell, and which one of those has the minimal length.

Oh, there is one thing left. If the spell is 'behappy', Albert will not choose the string s that is longer than the spell, such as 'behappysucks', even though it still works.

5.1. (4%) Let string $P[1..n]$ be the spell. $P_t = P[1..t]$ is the prefix of P of length t. Fill the following blanks and briefly explain your answer: for a t, Albert can choose P_t as s if and only if $P[i] = P[i + \_]$ for $i = 1, ..., \_$.

5.2. (4%) Let $\pi[1 \dots n]$ be the prefix function of P as defined in the KMP algorithm. Due to the definition of $\pi[]$, show that $s = P_t$ always works when $t = n - \pi[n]$.

5.3. (4%) Due to the definition of $\pi[]$, show that the minimum possible length of s is $n - \pi[n]$.

5.4. (4%) Can you figure out an algorithm that can find all possible s from part of the KMP algorithm? You need to explain why the strings you found work, and why the others don't.

Good enough? Well, this method works so well that Albert becomes more evil. He doesn't want to stay and wait to stop the recorder. Fortunately, the recorder supports a function that one can set how many times the tape will be played. After the recorder starts to play, Albert would leave the recorder there and start playing games. Since the recorder always plays the tape in its full length, if the spell is 'abaaba', to record 'aba' is a good idea, but to record 'abaab' is not. Once again, Albert wonders what the minimal length of string s that he can use to cheat would be.

5.5. (4%) For a t, in which condition can Albert choose P_t as s? (Modify your answer in Problem 5.1)

5.6. (Bonus 5%) From previous problems, we know that, if $s = P_t$ where $t = n - \pi[n]$ is a feasible solution, it is indeed the minimal length solution. Prove that if it is not a feasible solution, then $s = P$ is the only solution.