CS172 Computability & Complexity (Spring'09)

Instructor: Mihai Pătraşcu GSI: Omid Etesami

Problem Set 5 *Due: Thursday, March 5*

Homework policy: Please read carefully.

- For extensions to the due date, email Mihai with a *short* and sensible request.
- Collaboration is allowed, as long as *all* writing, of either code or proofs, is done in isolation (you may talk about ideas, but implement and write by yourself; go back to discuss if you discover new obstacles).
- On any problem, you will receive full credit if you write nothing else but "Please give me full credit." (Yes, we mean it. This is not a trick; no question will be asked.) For example, you may exercise this option for a problem that you have already solved in the past, which would be boring to solve again.
- Keeping the previous rule in mind, please don't waste your time (and ours) by copying solutions from the internet, colleagues, bibles, etc. Just ask for full credit.
- Regular (not [HARD]) problems are meant to help you understand interesting and useful material, and prepare for the exams. Abuse regarding the collaboration and "full credit" rules often correlates with low exam scores.
- Some problems are marked as [HARD]. It is possible to get top grades in the course even if you fail to solve many of them. They are meant as challenges to be attacked with good sportsmanship.

1. Implement the Knuth-Morris-Pratt algorithm in a programming language of your choice. (If different from C, Java, and Python, please notify Mihai in advance.)

Your code should read two lines from the standard input: the first is the needle, and the second is the haystack. It should produce a single line of output: the first position (counting from 1) where the needle appears in the haystack, or the word "NO" if it does not appear anywhere. Make sure your code passes all the tests found at: `http://inst.eecs.berkeley.edu/~cs172/sp09/hw5-p1.txt`

Email the code to Mihai as an attachment to an email with the subject `CS172-HW5-P1`. Grading will be done automatically so please follow I/O and email specifications carefully.

Clarification. There seems to be some confusion as to what "reading from standard input" means. Here is what your C code might look like:

```
#include <stdio.h>
int main()
{
   char needle[1000], haystack[1000];
   gets(needle);
   gets(haystack);
       /* ... do some computation here ... */
   if(found)
       printf("%d\n", position);
   else printf("NO\n");
   return(0);
}
```

Refresher. The main idea of the Knuth-Morris-Pratt algorithm was to construct an automaton like the following (in case we are searching for the string "ananab").

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[loop above, "\Sigma \setminus \{a\}"] \arrow[r, "a"] & q_1 \arrow[l, bend left, "\varepsilon"] \arrow[r, "n"] & q_2 \arrow[ll, bend left, "\varepsilon"] \arrow[r, "a"] & q_3 \arrow[ll, bend left, "\varepsilon"] \arrow[r, "n"] & q_4 \arrow[ll, bend left, "\varepsilon"] \arrow[r, "a"] & q_5 \arrow[ll, bend left, "\varepsilon"] \arrow[r, "b"] & q_6
\end{tikzcd}$$

The characters of the haystack are fed to the automaton one by one. An ε-transition is taken when the other (forward) transition does not match. The ε-transition does no "eat" the haystack character, so an arbitrary number of ε-transitions can be taken for each haystack character.

To construct the automaton in linear time, use ε-transition at node i to figure out the ε-transition at node $i+1$. You want to find the maximum prefix of the string that matches a suffix ending at position $i+1$. So go back on ε-transitions until the character at position $i+1$ matches the next character in the prefix.

2. Referring to the code you submitted in problem 1, argue formally that your algorithm for constructing the automaton runs in $O(m)$ time, where m is the number of characters in the pattern. (You may use phrases like "At the end of line 17, imagine adding a dollar to your bank account.")

3. Design an algorithm that goes through a stream of n numbers from $\{1, \ldots, u\}$, using only $O(\lg n + \lg u)$ bits of space. At the end of the stream, your algorithm should output a set S of at most 9 values, with the property that: for any x occuring at least $\frac{n}{10} + 1$ times in the stream, $x \in S$.

Note: If less than 9 elements appear $\frac{n}{10} + 1$ times, some of the 9 output values may be "junk." In other words, it is impossible to guarantee the converse: "the algorithm *only* outputs numbers that appear $\frac{n}{10} + 1$ times." We formally proved this impossibility in lecture, using the indexing problem.

Hint: If you have difficulties with this problem, a hint is posted at `http://inst.eecs.berkeley.edu/~cs172/sp09/hw5-p3.txt`

4. Remember the variant of the Boyer-Moore algorithm that we discussed in class:

```
• let $p = 1$
• while($p \le n - m + 1$):    // we will try to match the needle against positions $[p : p + m - 1]$ of the haystack.
    – let $k = m$      // we match from the last character, going back
    – while(Needle[$k$] = Haystack[$p + k$]):
    .          $k = k - 1$
    .          if($k = 0$): print "found needle at position $p$"; exit
    – // now we know that positions $[k + 1 : m]$ of the needle matched
    – $p = p + \delta[k]$
```

The crucial component of the algorithm is the array δ which tells us how many positions we can skip ahead without missing any matches. For instance, $\delta[M]$ should always be $+1$ (just increment POS if you didn't match anything, i.e. you have no information).

But if the needle is `xababx`, then $\delta[5]$ should be $+5$. Indeed, an `x` in the haystack may only match the first or last character of the needle. We look at $\delta[5]$ when we matched the last character (so we found an `x`). Thus the needle may slide by 5 positions without missing occurences.

Describe an algorithm to compute the array δ which runs in time $O(m^c)$ for some constant c. For instance, it is easy to achieve $O(m^3)$.

5. [Hard] Describe an algorithm to compute the array δ in $O(m)$ time.

6. [Hard] A firing squad is composed of n soldiers standing in line. Each soldier shouts a message per second, which is heard by his two immediate neighbors. Soldier n only hears soldier $n-1$. Soldier 1 hears soldier 2 and the General. The soldiers may only remember a constant amount of information (thus, they cannot know the value n). At some unknown time t_1, the General will shout "Fire." The goal of the soldiers is to all fire *simmultaneously* at some time $t_2 > t_1$.

Formally, you must design finite automata for the soldiers. Transitions are marked by pairs (ℓ, r), where ℓ is what the soldier hears from the left neighbor, and r is what he hears from the right neighbor. Each node is marked with a message that the soldier will shout after reaching the node. All soldiers will use the same finite automaton, except soldiers 1 (for whom ℓ is the general) and n (for whom r is missing). The size of the finite automaton must be an absolute constant, independent of n.

You may assume that n is a power of two.