Pattern Matching Algorithms 89-775-01 FINAL EXAM

MOED A

Instructor: *Prof. Amihood Amir*

Length of Exam: 2 hours

Time: 12:00, February 16th, 2010

BOOKS AND NOTEBOOKS ARE ALLOWED!!!

1. The *Point-Set Matching Problem* is defined as follows.
   INPUT: Text $T = t_0, ..., t_n$, and pattern $P = p_0, ..., p_m$,
   $p_i, t_j \in \{0, 1\},\ i = 0, ..., m,\ j = 0, ..., n.$
   OUTPUT: All locations i in T where $t_{i+j} = p_j$ or $t_{i+j} = 1$ and $p_j = 0, \quad j = 0, ..., m.$

   The text and pattern represent points on the line, where every 1 is a point. The idea is to find the text locations where every pattern point aligns with a text point. (We do not care if there are text points that do not align with pattern points.) text element.
   Example: If $T = 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1$ and $P = 1, 0, 0$ then there is a match in location 1 of the text, since the first location of the pattern matches the text, and there are no more 1's in the pattern. Similarly, there is a match in text locations $2, 5, 7$ and 8.

   a) Write an algorithm that solves the *Point-Set Matching Problem* (explain the algorithm's idea, the tools you used, and give a high level description).
   b) What is the time complexity of your algorithm? Justify.
   (The more efficient the algorithm the higher the points given.)

   Answer:

   Note that the only illegal possibility is when a "0" in the text is juxtaposed with a "1" in the pattern. We conclude that $\overline{T} \times P^R[i] \neq 0$ iff P does not point-set-match T at location i. ($\overline{T}$ is the complement of T, i.e. 0 becomes a 1 and 1 becomes a 0, P^R is the reversal of P, $\times$ is polynomial multiplication.) This is true because the convolution will match all the 0's in the text with their counterparts, counting an error when the pattern has a 1 and not counting an error where either the pattern has a 0 or the text has a 1.

   Time: $O(n \log m)$ using FFT.

   Error Codes:

   (1) Confused explanation but general idea correct. (-5)

   (2) Using KMP or any algorithm utilizing transitivity. (-18)

   (3) Presenting an $O(nm)$ algorithm correctly. (-13)

   (4) Error in calculation. (-5)

2. In Berkman's LCA algorithm, there is a step where one seeks the index of leaf i on the min suffix of the left son of the $LCA(i, j)$. Show how this can be done in constant time. Explain your idea, give a high level description and justify that the time is indeed constant.

Answer:

Let k be the "name" of the left son of $LCA(i, j)$. By "name" we mean the 0/1 encoding of the left/right path from the root to the node.

k is a prefix of i. The desired location in the min-suffix table of k is the part of i without the prefix k. Thus if we shift k the appropriate number of bits the the left and the XOR with i we will get the correct address. This is preformed by the operation below:

$$i \text{ XOR } shift - left(k, \lfloor \log i \rfloor - \lfloor \log k \rfloor)$$

Where $shift - left(a, b)$ shifts a to the left b positions.

Since we assume that register word operations are constant time, this allows us to perform the above table lookup at a RAM in constant time.

Error codes:

The poor performance in this question was especially disappointing since it was assigned as a voluntary exercise during the class.

(1) Correct explanation about tree but no understanding of what i is. (-16)

(2) Correct explanation about tree but question not answered. (-20)

(3) Knows that a XOR operation is necessary but does not correctly specify with what. (-7)

(4) Knows that a shift is necessary but does not correctly specify with what. (-7)

(5) Vague explanations of Berkman's algorithm, question not addressed. (-23).

3. The *Permutation Indexing Problem* is defined as follows.
INPUT: Text $T = t_0, ..., t_n$ over the Natural numbers.
PREPROCESS: The text, to enable solutions for queries of the form:
QUERY: All locations i in T where there is a parameterized match of input pattern $P = p_0, ..., p_m$, where P is a permutation of $\{1, ..., m + 1\}$.

a) Write an algorithm that solves the *Permutation Indexing Problem* (explain the algorithm's idea, the tools you used, and give a high level description).
b) What is the time complexity of your preprocessing? Query processing? Justify. (The more efficient the algorithm the higher the points given.)

Answer:

It is clear that what the problem requests is an indexing problem to the query: "Given a permutation of length m, find all text locations where there are m different consecutive symbols".

The idea is, therefore, to preprocess the text counting for every location what is the longest run of different symbols starting at that location. For the query, we will need the inverse of that table, i.e. for every length, the indices where a substring of non-repeating symbols of that length starts. If we also want to keep the space linear, we

will look the maximal maximal length substring of non-repeating symbols starting at every text location.

For simplicity's sake, let's assume that our alphabet is $\{1, ..., n\}$. (If not, in time $O(n \log m)$ one can sort and convert to such an alphabet.)

Preprocessing: Construct a table A of length n for the different symbols $1, ..., n$. Initialize A to 0. Use a left pointer ℓp and a right pointer rp for a "sliding window" over T of substrings non-repeating symbols. ℓp and rp are initialized to 1.

As long as here are elements in the text and $A[rp] = 0$ (new symbol), set $A[rp]$ to rp, extend the length of the run and move rp one to the right. If $A[rp]$ was k, then the run ends at $rp-1$, the new run starts at $k+1$, all the symbols from ℓp to k are reset to 0 in A, all locations in the text from ℓp to k get their maximum substring of non-repeating symbols ending at $rp - 1$, rp is moved one to the right, and ℓp is set to $k + 1$.

When the text is completed, bucket sort by the maximum lengths of the substrings at every index, so that we get, for every length, a list of all indices where that length is maximal.

Time: Clearly, $O(n)$.

Query Processing: For a given permutation of length m, print all lists of lengths greater than or equal to m.

Time: Assuming that all inputs are permutations of $\{1, ..., m\}$ the time is $O(tocc)$. If one needs to check whether the input is a permutation of $\{1, ..., m\}$ then the time is $O(m + tocc)$.

Error Codes:

(1) Gave a serial algorithm. (-24)

(2) Query time $O(n + tocc)$. (-15)

(3) Incorrect time analysis. (-5)

(4) No distinction made between finite, $\{1, ..., n\}$ and infinite alphabets. (-3)

(5) $O(n)$ time claim when in reality it was $O(n \log n)$. (-5)

(6) Bad analysis of query time. (-5)

(7) Preprocessing in time $O(n^2 \log n)$. (-10)

(8) Query time $O(m!)$. (-15)

GOOD LUCK