58093 String Processing Algorithms (Autumn 2012)
Course Exam, 13 December 2012. Solutions.

1. [*4+4+4 points*] Each of the following pairs of concepts are somehow connected. Describe the main connecting factors or commonalities as well as the main separating factors or differences.

   (a) (Knuth–)Morris–Pratt algorithm and Aho–Corasick algorithm.

   Solution. Both are exact string matching algorithms, KMP for a single pattern and AC for multiple patterns. Both are based on an automaton that uses failure transitions, but only AC constructs the automaton explicitly. Both are linear time algorithms, but KMP for an ordered alphabet and AC for an integer alphabet or a constant size alphabet.

   Scoring. Points were awarded for mentioning key elements according to the following table.

Element	points
exact string matching	1
single vs. multiple patterns	1
automaton with failure transitions	1
implicit vs. explicit automaton	$\frac{1}{2}$
linear time	1
ordered vs. integer/constant alphabet	$\frac{1}{2}$

   Factual errors may subtract points. If the sum exceeds four points, only four points is awarded.

   (b) String quicksort and MSD radix sort.

   Solution. Both are string sorting algorithm that work by partitioning the strings and then recursing on the partitions. Both partition by a single symbol following the known common prefix. String quicksort has three and MSD radix sort σ partitions in each step. String quicksort works on ordered alphabet while MSD radix sort requires integer alphabet. Both have time complexity of the form $\mathcal{O}(L + n \log x)$, with $x = n$ for string quicksort and $x = \sigma$ for MSD radix sort. MSD radix sort calls string quicksort to handle small partitions.

Element	points
string sorting	1
partition and recurse	$\frac{1}{2}$
three vs./ σ partitions	$\frac{1}{2}$
partitioning by one symbol	$\frac{1}{2}$
skip known prefix	$\frac{1}{2}$
ordered vs. integer alphabet	$\frac{1}{2}$
time complexities	$\frac{1}{2}+\frac{1}{2}$
MSD RS calls SQS	$\frac{1}{2}$

   (c) Compact trie and suffix tree.

   Solution. The suffix tree is the compact trie that stores the set of all suffixes of a text instead of an arbitrary set of strings. Suffix links are a special feature of the suffix tree that are not supported in arbitrary compact tries. The suffix tree can be constructed in linear time in the number of suffixes while a compact trie construction requires $\mathcal{O}(n + L(\mathcal{R}))$ time for a string set $\mathcal{R}$. Suffix trees have many applications that are not relevant to arbitrary compact tries.

Scoring. Points were awarded according to the following table.

Element	points
ST is a CT	1
suffixes vs. any set	1
suffix links	1
construction	1
applications	$\frac{1}{2}$

A few lines for each part is sufficient.

2. [*7+7 points*] Let $T[0..n)$ be a string and let $lcp(T_i, T_j)$ denote the length of the longest common prefix between the suffixes of T starting at positions i and j. The *longest previous factor* array $LPF[1..n)$ is defined by

$$LPF[i] = \max_{j \in [0..i)} lcp(T_i, T_j) \ .$$

(a) Show that for all $i \in [1..n-1)$, $LPF[i+1] \geq LPF[i] - 1$.
Hint: If $S[0..p)$ is a prefix of T_i then $S[1..p)$ is a prefix of T_{i+1}.

Solution. Let $i \in [1, n-1)$ and assume $LPF[i] = l$. Consider two cases.

- $l = 0$. Then $LPF[i] - 1 = -1 < 0 \leq LPF[i+1]$, where the last inequality is the consequence of *lcp* function being non-negative.
- $l > 0$. The definition of l implies that there exists $k < i$ such that $T[i..i+l) = T[k..k+l)$. In particular $T[i+1..i+l) = T[k+1..k+l)$ hence $lcp(T_{i+1}, T_{k+1}) \geq |T[i+1..i+l)| = l-1$. Notice that $k+1 < i+1$ so from the LPF array definition we have

$$LPF[i+1] = \max_{j \in [0..i+1)} lcp(T_{i+1}, T_j) \geq lcp(T_{i+1}, T_{k+1}) \geq l - 1 = LPF[i] - 1.$$

Scoring. Full points were awarded for complete justification using the definition of LPF array. Partial points were given for an incomplete proof. One point was subtracted if the case $LPF[i] = 0$ was not handled.

(b) Suppose we are given an array $Prev[1..n)$ of integers in $[0..n)$ satisfying for all i

$$Prev[i] < i$$
$$lcp(T_i, T_{Prev[i]}) = LPF[i]$$

Describe an algorithm for computing the LPF array from the $Prev$ array in linear time. *Hint:* Use the result of (a)-part.

Solution. The pseudo-code of the algorithm is given below.

Input: string $T[0..n)$ and array $Prev[1..n)$
Output: array $LPF[0..n)$

```
(1)  LPF[0] ← l ← 0
(2)  for i ← 1 to n − 1 do
(3)      while i + l < n and T[i + l] = T[Prev[i] + l] do
(4)          l ← l + 1
(5)      LPF[i] ← l
(6)      l ← max(l − 1, 0)
```

The correctness of the algorithm follows from (a). To prove it has linear time complexity we observe that each iteration of the while loop increases l. Its maximal value is $n-1$ and during the whole algorithm it is decreased by 1 at most $n-1$ times (line 6). The line 4 is therefore executed at most $2(n-1)$ times.

Scoring. Points were awarded according to the following table.

Element	points
using the computed LPF values to avoid character comparisons	3
handling the case $LPF[i+1] > LPF[i]-1$	3
handling the case $LPF[i]=0$	1

Up to two points can be subtracted for mistakes in the pseudo-code / description.

3. [*6+6 points*]

(a) Compute the edit distance between strings `tukholma` and `stockholm` using the dynamic programming algorithm described on the course.

Solution.

d		s	t	o	c	k	h	o	l	m	
	0	→ 1	2	3	4	5	6	7	8	9
t		↘									
	1	1	1	→ 2	3	4	5	6	7	8
u			↘	↘							
	2	2	2	2	→ 3	4	5	6	7	8
k					↘						
	3	3	3	3	3	3	4	5	6	7
h						↘					
	4	4	4	4	4	4	3	4	5	6
o							↘				
	5	5	5	4	5	5	4	3	4	5
l								↘			
	6	6	6	5	5	6	5	4	3	4
m									↘		
	7	7	7	6	6	6	6	5	4	3
a										↓	
	8	8	8	7	7	7	7	6	5	4

The edit distance is 4.

Scoring. Points were awarded according to the following table.

Element	points
using the correct formula	4
boundary cases handled correctly	1
correct value of edit distance	1

(b) Give *all* optimal alignments between `tukholma` and `stockholm`, i.e., alignments with the same cost as the edit distance.

Solution.

```
INSINNNNND    INISNNNNND
-tu-kholma    -t-ukholma
stockholm-    stockholm-
```

Scoring. Points were awarded according to the following table.

Element	points
alignments consistent with the matrix produced in (a)	3
alignments were optimal	3

One point was subtracted if some extra alignments were given or were missing.

4. [*12 points*] Let T be a string of length n over an alphabet Σ of constant size. Describe an algorithm that finds the *shortest* string over the alphabet Σ that does *not* occur in T. The time complexity should be $\mathcal{O}(n)$.

   Solution. The basic idea is to perform a breadth-first traversal of the suffix tree of T and find the first locus that does not have children for all symbols. The locus may not have an explicit node. More precisely, the traversal ends when encountering an explicit node v satisfying one of the following conditions:

   - $\mathit{child}(v, c) = \perp$ for some symbol c. In this case, the answer is $S_v c$, where S_v is the string represented by v.
   - $\mathit{depth}(v) > \mathit{depth}(u) + 1$, where u is the parent of v. In this case, there is an implicit node on the edge (u, v). The answer is $S_u cd$, where c is the first symbol in the label of the edge (u, v) (i.e., $\mathit{child}(u, c) = v$), and d is any symbol different from the second character of the edge label.

   Scoring. Three points is subtracted if implicit nodes are ignored. One or two points were given for some incorrect solutions with correct elements.