58093 String Processing Algorithms (Autumn 2016)
Course Exam, 19 December 2016. 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) String quicksort and string mergesort.

   Solution. Both are algorithms for sorting strings. Both are adaptations of standard (non-string) sorting algorithms. Both work on an ordered alphabet and have time complexity $\mathcal{O}(n \log n + L)$ for sorting n strings with total lcp length of L, which is optimal. Both are divide-and-conquer algorithms: string quicksort partitions by value and concatenates sorted parts, while string mergesort partitions arbitrarily and merges sorted parts. Both use lcp information. String quicksort compares strings using only one symbol at a given stage, while string mergesort compares strings using the lcp-comparison technique. Mergesort returns the LCP array in addition to the sorted set. String quicksort needs only $\mathcal{O}(\log n)$ extra space, while string mergesort needs $\mathcal{O}(n)$ extra space.

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

Element	points
string sorting	1
adaptations of standard sorting algorithms	$\frac{1}{2}$
ordered alphabet / based on character comparisons	$\frac{1}{2}$
time complexity	1
divide-and-conquer / recursion	$\frac{1}{2}$
difference in divide-and-conquer	$\frac{1}{2}$
use lcp information	$\frac{1}{2}$
symbolwise comparison vs. lcp-comparison	$\frac{1}{2}$
mergesort returns LCP array	$\frac{1}{2}$
extra space	$\frac{1}{2}$

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

   (b) Shift–And algorithm and BNDM algorithm.

   Solution. Both are exact string matching algorithms. Both are based on bit-parallel simulation of a nondeterministic automaton. Shift–Or automaton maintains information about which pattern prefixes match a suffix of the scanned string and accepts when the string ends with the full pattern, while the BNDM automaton maintains information about which factors of the reverse pattern match the scanned string and accepts suffixes of the reverse pattern. Shift–Or makes one, long left-to-right scan, while BNDM makes many right-to-left scans and shifts to the right between scans. Both require an integer alphabet. When pattern length m is at most the machine word size w, the BNDM search time complexity is $\mathcal{O}(mn)$ in the worst case, $\mathcal{O}(n(\log_\sigma m)/m)$ in the average case (which is optimal), and $\mathcal{O}(n/m)$ in the best case, while the Shift-Or search time complexity is always $\mathcal{O}(n)$ (which is optimal in the worst case). Both need $\mathcal{O}(m + \sigma)$ time for pattern preprocessing. Both algorithms slow down when $m > w$.

Scoring.

Element	points
exact string matching	1
bitparallel	1
non-deterministic automaton	$\frac{1}{2}$
pattern prefixes vs. reverse pattern factors/suffixes	$\frac{1}{2}$
forwards vs. backwards scan	$\frac{1}{2}$
continuous scan vs. shifting	$\frac{1}{2}$
integer alphabet	$\frac{1}{2}$
worst case search time	$\frac{1}{2}$
average case search time	$\frac{1}{2}$
optimality	$\frac{1}{2}$
best case search time	$\frac{1}{2}$
pattern preprocessing time	$\frac{1}{2}$
pattern length and word size	$\frac{1}{2}$

(c) Aho–Corasick automaton and suffix tree.

Solution. Both can solve the multiple exact string matching problem in linear time for a constant alphabet, but the solutions are very different: AC algorithm preprocess the patterns, while suffix tree solution preprocesses the text. AC algorithm is particularly designed for solving multiple exact string matching, but suffix tree can be used for solving many other problems, too.

Both are based on the trie structure. AC automaton is a trie for the set of patterns, while the suffix tree is the compact trie for the set of suffixes of the text. Both tries can be augmented with special links defined similarly, failure links in AC automaton and suffix links in suffix tree. Suffix tree can be used as an AC automaton for the set of suffixes.

Scoring.

Element	points
exact string matching	$\frac{1}{2}$
multiple exact string matching	$\frac{1}{2}$
linear time construction	$\frac{1}{2}$
linear time multiple exact string matching	$\frac{1}{2}$
pattern vs. text preprocessing	$\frac{1}{2}$
dedicated vs. general	$\frac{1}{2}$
trie	1
compact vs. non-compact	$\frac{1}{2}$
patterns vs. suffixes	$\frac{1}{2}$
links are similar	$\frac{1}{2}$
suffix tree as AC automaton	$\frac{1}{2}$

2. [*13 points*] Consider a variant of edit distance that does not allow insertions. That is, let $ed'(A, B)$ be the smallest number of substitutions and deletions needed to change A into B. If no such edit operations exist, $ed'(A, B) = \infty$.

   (a) Give either a dynamic programming recurrence or a pseudocode for a dynamic programming algorithm for computing the edit distance $ed'(A, B)$.

   Solution. The recurrence for $ed'(A[1..m], B[1..n])$ is:

$$
\begin{aligned}
d'_{00} &= 0, \\
d'_{i0} &= i, \ 1 \le i \le m, \\
d'_{0j} &= \infty, \ 1 \le j \le n, \ \text{and} \\
d'_{ij} &= \min \begin{cases} d'_{i-1,j-1} + \delta(A[i], B[j]) \\ d'_{i-1,j} + 1 \end{cases} \quad 1 \le i \le m, 1 \le j \le n,
\end{aligned}
$$

   where

$$
\delta(A[i], B[j]) = \begin{cases} 1 & \text{if } A[i] \ne B[j] \\ 0 & \text{if } A[i] = B[j] \end{cases}
$$

   Then $ed'(A, B) = d'_{m,n}$.

   (b) Modify your recurrence or algorithm for approximate string matching: Given T, P and k, find all ending positions of substrings S of T such that $ed'(P, S) \le k$.

   Solution. The recurrence is:

$$
\begin{aligned}
g'_{00} &= 0, \\
g'_{i0} &= i, \ 1 \le i \le m, \\
g'_{0j} &= 0, \ 1 \le j \le n, \ \text{and} \\
g'_{ij} &= \min \begin{cases} g'_{i-1,j-1} + \delta(P[i], T[j]) \\ g'_{i-1,j} + 1 \end{cases} \quad 1 \le i \le m, 1 \le j \le n.
\end{aligned}
$$

   Then there is an occurrence of $P[1..m]$ ending at position j in $T[1..n]$ iff $g'_{m,j} \le k$.

   Make sure that your solution handles correctly all special and boundary cases.

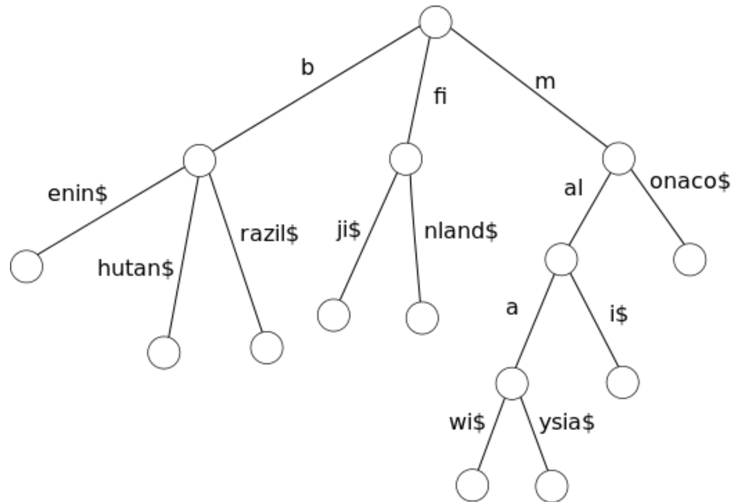
   Scoring.

   - 5 points for omitting the insertions when taking the minimum
   - 3+3 points (3 for (a) and 3 for (b)) for correct first column and row
   - 1+1 points for correct (interpretation of the) output

3. [*4+4+4 points*]
For the string set {`benin`, `bhutan`, `brazil`, `fiji`, `finland`, `malawi`, `malaysia`, `mali`, `monaco`}, give

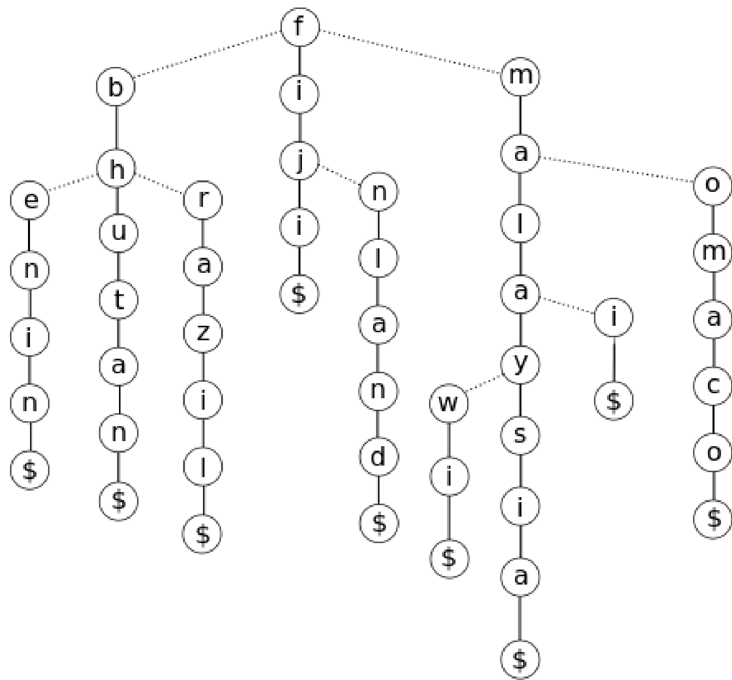
(a) the compact trie

Solution.



(b) the balanced ternary trie

Solution.



(c) the LLCP and RLCP arrays for efficient binary searching in the sorted array

Solution. There is no single correct answer, because the LLCP and RLCP arrays depend on the implementation details of the binary search. For example, different rounding of fractional values when computing a new midpoint can change the course of the binary search and therefore the LLCP and RLCP arrays. Also, it's worth noting that a straightforward binary search has to precompute the initial values for llcp and rlcp before running. This can be avoided by adding a dummy element to the start and end of the array and defining their LCP with everything to be zero, resulting in a debatably cleaner implementation.

Below is one possible solution without the dummy elements. The left and right ends for whenever a given element is the midpoint of the search are tabulated along with the corresponding LLCP and RCLP values.

Index	String	left	right	LLCP	RLCP
0	benin	0	1	5	1
1	bhutan	0	2	1	1
2	brazil	0	4	1	0
3	fiji	2	4	0	2
4	finland	0	8	0	0
5	malawi	4	6	0	4
6	malaysia	4	8	0	1
7	mali	6	8	3	1
8	monaco	7	8	1	1

Scoring.

- 4 points for each correct data structure
- $\frac{1}{2}$ points subtracted for minor errors
- 1–2 points subtracted for more systematic errors

4. [*9+4 points*]

(a) Describe a linear time algorithm that given two strings, S and T, and an integer k finds if there exists a string that occurs exactly k times in S and exactly k times in T.

Solution. Build the suffix tree of the concatenation $S\$T£$, i.e., the generalized suffix tree of S and T. Mark each leaf according to whether it starts in S-part or T-part. For each internal node, count the number of S-type leaves and the number of T-type leaves in subtree. This can be done in linear time by depth-first traversal of the tree. If there exists a node, where both numbers are k, the algorithm answers "yes". Otherwise the algorithm answers "no".

Scoring.

- 3 points for using a generalized suffix tree
- 3 points for appropriate marking of the nodes
- 3 points for correctly determining the output from the markings

(b) Show that your algorithm can be implemented to run in deterministic worst case linear time in the integer alphabet model. If you are using a trie-based data structure, your argument should include a description of how to implement the child-operation.

Solution. The child operation can be implemented with a linked list, which is sufficient for linear time traversal of the tree. The suffix tree can be constructed by first constructing the suffix array and the LCP array and then the suffix tree from them, all of which can be done in linear time.

Scoring.

- 2 points for an approriate child operation implementation
- 2 points for linear time construction