University of Waterloo

version: 2018-03-27 11:12

CS240 Winter 2018
Sample Solutions Assignment 5

Note: you may assume all logarithms are base 2 logarithms: $\log = \log_2$.

Problem 1 [8+8+9 = 25 marks]

a)

	0	1	2	3	4	5	6	7	8
a	0	0	3	0	0	0	3	0	8
b	1	2	2	1	5	6	2	8	8
c	0	0	0	4	0	0	7	0	8

b)

	0	1	2	3	4	5	6	7	8
✓	1	2	3	4	5	6	7	8	8
×	0	0	1	0	0	1	2	0	(1)
✓-symbol	b	b	a	c	b	b	c	b	

c) **Idea:** We obtain $\delta(q, c)$ by simulating the KMP-automaton for the next symbol c: Follow a (potentially empty) chain of failure links starting at q until we can traverse an arc with label c (using a ✓-link). The state we reach that way is $\delta(q, c)$.

Code:

```
computeDelta() :
   F = failureArray(P)
   delta[:] = 0 // initalize all to 0
   delta[0,P[0]] = 1;
   for q = 1,...,m-1
      for c in Σ
         if c == P[q]
            delta[q,c] = q+1
         else
            delta[q,c] = delta[F[q-1],c]
   for c in Σ
      delta[m,c] = m
```

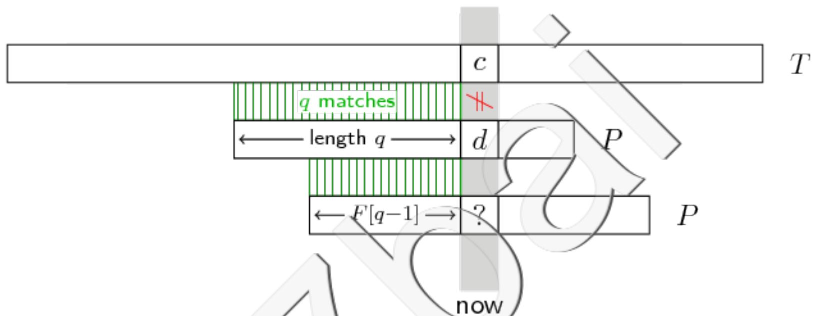
Correctness: We use the following observation on the transition function:

$$q \notin \{0, m\} \wedge P[q] \neq c \implies \delta(q, c) = \delta(F[q-1], c)$$

Proof: by induction on q.
For $q = 1$, we could only match the first character of P with the text. As the next

character does not extend this match by assumption ($P[q] = P[1] \neq c$), we can only end in states 0 or 1. Indeed, we end in state 1 iff $P[0] = c$. For $q = 1$, we always have $F[q-1] = F[0] = 0$, and $\delta(0, c) = 1$ if $P[0] = c$ and 0 otherwise. Hence the claim is true for $q = 1$.

Now assume the claim holds for all values $< q$. If in state q the next character c does not extend the current best match ($c \neq P[q]$), we have to find the (length q' of the) longest prefix $P[0..q'-1]$ that is strict suffix of $P[0..q-1] \cdot c$. By the definition of the failure array, $F[q-1]$ is the longest prefix of $P[0..q-1]$ that is a strict suffix of $P[0..q-1]$. This is the first candidate for our prefix of $P[0..q-1]c$ since any longer prefix for $P[0..q-1]c$ would imply a longer prefix for $P[0..q-1]$ in contradiction to the definition of $F[q-1]$. However, F does not guarantee us that we can extend that match to the next text character c. The situation is illustrated below:



Now there are two cases. If we can extend the match, i. e., $P[F[q-1]] = c$, the resulting state is $\delta(q, c) = F[q-1] + 1 = \delta(F[q-1], c)$ as claimed. The second equality holds because c is a match transition from there. If $P[F[q-1]] \neq c$, we have to consider the next shorter prefix. This situation is the same as our initial situation, but with q replaced by $F[q-1]$, so our best the next guess is $F[F[q-1]-1]$. By the inductive hypothesis we know that in this case $\delta(F[q-1], c) = \delta(F[F[q-1]-1], c)$ holds, so we again find $\delta(q, c) = \delta(F[q-1], c)$ as claimed.

Since we compute δ for increasing values of q and $F[q-1] < q$, `delta[F[q-1],c]` has already been computed when we use it in line 10.

Runtime: The nested loops take $O(m \cdot |\Sigma|)$ time, since each computation runs in constant time.

Problem 2 [20 marks]

The idea is that we can reduce this problem to pattern matching, by searching for pattern $P = w$ in string $T = xx$.

```
isCyclicShift(x,w)
   if (|x| != |w|) return false
   P = w
   T = x + x // + is string concatenation
   return KMP(T,P) != FAIL
```

To see the correctness, we prove both implications of

$$w \text{ is cyclic shift of } x \iff w \text{ occurs in } xx$$

$\Rightarrow$ If w is a cyclic shift of x, then it must be contained in xx, because the beginning of w matches the end of x, and the rest of w matches the beginning of x.

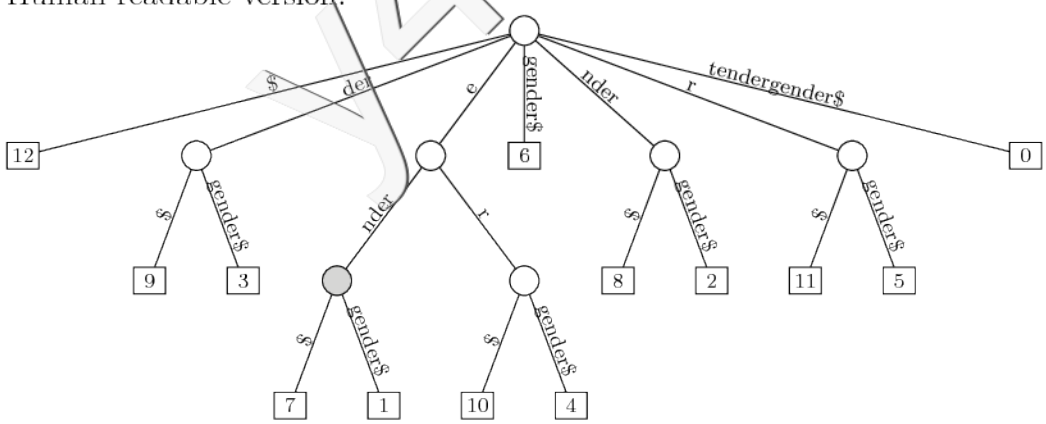
$\Leftarrow$ If xx contains w, then some suffix of x plus some prefix of x equals w. Since w and x have the same length, this implies that w is in fact a cyclic shift of x.

Hence any string matching algorithm would work correctly, but to be efficient, we use the Knuth-Morris-Pratt algorithm. Then the run time of this algorithm is $O(|T| + |P|)$, which is $O(n)$.

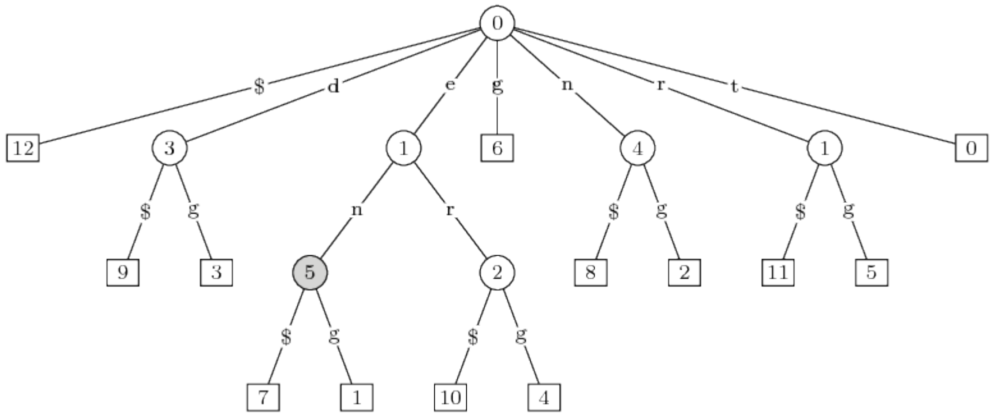
Alternatives: Since here $|P| = \Theta(|T|)$, we can also use suffix trees to achieve the same running time. (They are more complicated to build, but it is possible in linear time.)

Problem 3 [10+3+2+20 = 30 marks]

a) Human-readable version:



Compressed trie version:



Hint: A helpful intermediate step to build a suffix tree by hand is a *sorted list* of all suffixes. This allows to easily identify the edge labels without first building the trie of suffixes.

```
$
d e r $
d e r g e n d e r $
e n d e r $
e n d e r g e n d e r $
e r $
e r g e n d e r $
g e n d e r $
n d e r $
n d e r g e n d e r $
r $
r g e n d e r $
t e n d e r g e n d e r $
```

b) $R = \texttt{ender}$, $\ell = 5$, $(i, j) = (1, 7)$.

c) The search stops at an internal node, namely the one shaded in the pictures above.

d) **Idea:** If P occurs twice in T, then there are at least two suffixes of T that start with P. Hence there is an internal node, reached by traversing along this repeated pattern, at which these two suffixes first differ.

Code:

```
computeLongestRepeatedPattern(T)
   𝒯 = suffixTree(T)
   maxDepth = 0
   maxDepthNode = root of 𝒯
   for each internal node v in 𝒯
      // v.index is the index of the character compared at this node
      if v.index > maxDepth
         maxDepth = v.index
         maxDepthNode = v
   leaf = maxDepthNode.someLeafInSubtree() // as in slides
   i = leaf.startIndex
```

```
    l = maxDepthNode.index
    return T[i..i+l-1]
```

Correctness: In compressed tries, internal nodes store in `index` their "string depth", i. e., the length of the string leading to this internal node in a traversal starting at the root. Let v be an internal node of maximal string depth in $\mathcal{T}$, the suffix tree for T, and let R be the string that leads to it starting at the root. (This is the concatenation of the edge labels in the human-readable version of the suffix tree.) Since it is a compressed trie, there are at least to leaves reachable from v, so there are two distinct suffixes of T starting with R, so R is indeed a repeat.

Let conversely R' be any locally maximal repeat in T, i. e., if R' occurs at positions $i \neq j$ and has length ℓ, then $T_{i+\ell} \neq T_{j+\ell}$. (In other words, R' cannot be extended to the right.) Then traversing $\mathcal{T}$ with R' leads to an internal node, where the paths to leaves i and j fork. Any globally longest repeat must be locally maximal, as well, so any longest repeat corresponds to an internal node. But then our algorithm finds such a globally longest because it selects an internal node of maximal string depth.

Runtime: Constructing the suffix tree $\mathcal{T}$ take $O(n)$ time. For finding the deepest internal node, we traverse all internal nodes once. Since suffix trees are compressed tries, there are $O(n)$ internal nodes, so this step is again linear in $|T|$.

Problem 4 [10+10 = 20 marks]

a) Let $\Sigma = \{0, 1\}$. Assume, A is a compression method that always reduces its input size. That means, $A(\Sigma^n) \subseteq \Sigma^{\le n-1}$. But we have $|\Sigma^n| = 2^n$ whereas

$$|\Sigma^{\le n-1}| = \sum_{i=0}^{n-1} 2^i = 2^n - 1,$$

so A cannot be injective, a contradiction.

An alternative argument is by applying A iteratively. If A would always reduce the input size, after at most n steps, we have $A(A(\cdots(w))\cdots) = \Lambda$ the empty string, for any input $w \in \Sigma^n$. Since A has a unique inverse A^{-1}, its decoder, applying A^{-1} some $k \le n$ times to Λ must reproduce every $w \in \Sigma^n$, but obviously $(A^{-1})^k(\Lambda)$ can produce at most n different preimages (for $k = 1, \ldots n$) in Σ^n, whereas $|\Sigma^n| = 2^n > n$ for $n \ge 2$.

b) We note that

$$\left|\Sigma^{\le n}\right| = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1\ .$$

Consider $A(\Sigma^{\le n})$, i. e., the set of codewords assigned to all strings up to length n. These are $2^{n+1} - 1$ many strings, but there are only $|\Sigma^{\le n-1}| = 2^n - 1$ bit strings that are *strictly* shorter than n. That means $A(\Sigma^{\leq n})$ has to contain $2^{n+1}-1-(2^n-1)=2^n$ strings w of length at least n; for any of these holds $A(w) \geq |w|$, and they comprise more than half of $\Sigma^{\leq n}$.

Problem 5 [7+2+7+4 = 20 marks]

a)

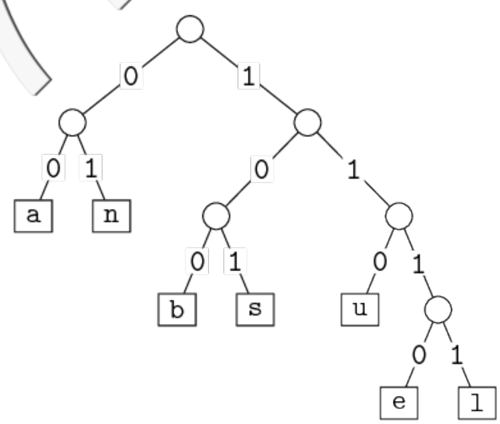
Symbol	Frequency	Codeword	Codeword Length
`a`	3	`10`	2
`b`	2	`110`	3
`e`	1	`1110`	4
`l`	1	`1111`	4
`n`	2	`00`	2
`s`	1	`010`	3
`u`	1	`011`	3

b) `ananasbubbles`

(I agree that "ananas bubbles" is not exactly a very meaningful phrase, but it compares reasonably favorably with "banana blues", doesn't it?)

c)

Symbol	codelen	Codeword
`a`	2	`00`
`b`	3	`100`
`e`	4	`1110`
`l`	4	`1111`
`n`	2	`01`
`s`	3	`101`
`u`	3	`110`



d) It is not possible to obtain the above code tree using Huffman's algorithm, since there is no valid execution that would group `b` and `s` directly. Since we have four letters of minimal frequency – `e`, `l`, `s` and `u` – any valid execution of Huffman's algorithm will start by grouping these four letters into two pairs. In the above tree, however, `b` and *s* instead form a pair, so it cannot be produced.