Algorithmics I - Tutorial Sheet 3

Strings and text algorithms

---

1. Explain how the Huffman-tree-building algorithm can be extended so that it also determines the weighted path length of the tree that it constructs.

   Recall: the weighted path length (`WPL`) of a tree T is the sum of

   $$\text{(weight)} \times \text{(distance from root)}$$

   over all leaf nodes of the tree.

   Solution: The Huffman-tree building algorithm can be extended to determine the weighted path length of the tree that it constructs as follows:

   1. initialise an integer variable `w` to zero;
   2. whenever a new internal node of the tree is created, add the weight of the node to `w`;
   3. on termination return `w`.

   The fact that on termination `w` equals the `WPL` of the Huffman tree can be proved as follows. First, by construction of the Huffman tree, the weight of each internal node equals the sum of the weights of the leaf nodes it has as descendants. Therefore, by construction of `w`, the weight of each leaf node i gets added in to `w` the number of times that is equal to the number of ancestors of i, in other words, the distance of i from the root. This is true for all leaf nodes, so `w` is the sum over all leaf nodes, of weight multiplied by the distance from root, i.e. the weighted path length.

2. Trace the `LZW` algorithm by hand on the text below. Generate a representation of the compressed file (with code words represented by integers), and a representation of the dictionary trie at the end of the compression. (Assume the dictionary initially contain the 128 characters of the basic `ASCII` character set, represented by the code words $0, 1, \ldots, 127$.)

   `peter piper picked a peck of pickled pepper`

   For information: the coding of relevant `ASCII` characters are:

   "`space`" : 32 `a` : 97 `c` : 99 `d` : 100 `e` : 101 `f` : 102 `i` : 105
   `k` : 107 `l` : 108 `o` : 111 `p` : 112 `r` : 114 `t` : 116

   and the next available code word is 128.

   Solution: Compressed version (with `@` representing 'blank', and each 8-bit codeword interpreted as an integer):

p	e	t	e	r	@	p	i	pe	r@	pi	c	k	e	d	@
112	101	116	101	114	32	112	105	128	132	134	99	107	101	100	32

a	@p	e	ck	@	o	f	@p	i	ck	l	ed	@pe	p	per
97	133	101	139	32	111	102	133	105	139	108	141	145	112	136

Dictionary: 0 - 127 is standard `ASCII`, while the new elements are given by:

`pe`:128	`et`:129	`pi`:134	`ip`:135	`ke`:140	`ed`:141	`ec`:146	`ck@`:147	`ic`:152	`ckl`:152
`te`:130	`er`:131	`per`:136	`r@p`:137	`d@`:142	`@a`:143	`@o`:148	`of`:149	`le`:154	`ed@`:155
`r@`:132	`@p`:133	`pic`:138	`ck`:139	`a@`:144	`@pe`:145	`f@`:150	`@pi`:151	`@pep`:156	`pp`:157

3. (2008-09 degree exam question) Suppose that the `LZW` compression algorithm is applied to the text:

   `abababababab...`

   of length 100 (i.e. the text that contains an alternating sequence of a's and b's and 100 characters in total).

   Determine the resulting compression ratio and saved space (as a percentage of the original file size). Assume that each character in the source file occupies 8 bits, that the initial dictionary size is 128, and that the initial codeword size is 8.

   Hint: apply the `LZW` algorithm until a pattern in the amount of text used up in each step emerges. This pattern can then be used to compute the number of steps required to reach the 100th (and last) character. From the number of steps you can then find the number of code words required to encode the text.

Solution: The compression algorithm proceeds as follows on the given string:

steps	output codeword represents string	string entered into dictionary
1	`a`	`ab`
2	`b`	`ba`
3	`ab`	`aba`
4	`aba`	`abab`
5	`ba`	`bab`
6	`bab`	`baba`
7	`abab`	`ababa`
8	`ababa`	`ababab`
9	`baba`	`babab`
10	`babab`	`bababa`

Analysing the pattern, it can be seen that the number of characters of the string consumed by each successive output codeword is

`1, 1, 2, 3, 2, 3, 4, 5, 4, 5, 6, 7, 6, 7, 8, 9, 8, 9, 10, 11,...`

To find the number of code words required, we need to know the number of terms required for the sum of this sequence to equal 100. Performing this summation, we find the first 19 terms sum to 100, and hence 19 steps (and 19 codewords) are required to encode the string. Each of these codewords is of length 8 bits (the initial code word size), since there was a total of 128 codewords free (the initial dictionary size was 128) and we have only used 19. Therefore, the input size is 800 and the output size $8 \cdot 19 = 152$, giving a compression ratio $\frac{152}{800}$ and the saved space equals:

$$\left(1 - \frac{152}{800}\right) \cdot 100\% = 81\%$$

4. Find the distance between the strings s and t shown below, and derive an optimum alignment.

$$s = \texttt{agcgatc} \text{ and } t = \texttt{ctacgaccg}$$

Solution: The distance is 5 (see the tutorial for more details). One possible alignment, with 3 insertions, 1 deletion, and 1 substitution, is as shown:

```
-  -  a  g  c  g  a  t  c  -
c  t  a  -  c  g  a  c  c  g
```

5. (2018-19 degree exam question) A string u is a subsequence of a string s if u can be obtained from s by deleting zero or more characters. A string u is a common subsequence of s and t if it is a subsequence of both s and t. (The length of an `LCS` is often used as a measure of similarity of two strings.)

   Design a dynamic programming algorithm to determine the length of the longest common subsequence (`LCS`) of two strings s and t.

   Hint: base your algorithm on evaluating $l(i, j)$, the length of the `LCS` of the ith prefix of s and the jth prefix of t.

Solution: A dynamic programming algorithm is based on the recurrence

$$l(i,j) = \begin{cases} l(i{-}1, j{-}1) + 1 & \text{if } s[i] = t[j] \\ \max\{\, l(i{-}1, j), l(i, j{-}1) \,\} & \text{otherwise} \end{cases}$$

subject to $l(i, 0) = l(0, j) = 0$.

To understand the algorithm considering the cases in turn we have

- If $s[i] = t[j]$, then an `LCS` of $s[0..i]$ and $t[0..j]$ will be 1 longer than an `LCS` of $s[0..i{-}1]$ and $t[0..j{-}1]$
- If $s[i] \neq t[j]$, then obviously an `LCS` cannot end with a character that is both $s[i]$ and $t[j]$, hence one of these characters will have to be deleted. Since we are looking for the longest common subsequence of $s[0..i]$ and $t[0..j]$, we take the maximum of those of $s[0..i{-}1]$ and $t[0..j]$ and $s[0..i]$ and $t[0..j{-}1]$.

6. Find a longest common subsequence of the strings s and t in Question 6 above.

Solution: An `LCS` has length 5, e.g., one such subsequence is `a c g a c`.

7. Construct the border table B for the `KMP` algorithm for the string:

`agcagacagcacg`

Solution: The border table is given by:

j	0	1	2	3	4	5	6	7	8	9	10	11	12
$B(j)$	0	0	0	0	1	2	1	0	1	2	3	4	0

8. Indicate precisely which character comparisons would be made if the Boyer-Moore algorithm were used to locate the first occurrence of the string $s =$ `agcga` in the text $t =$ `agcgcctgatagcgacagt`.

Solution: The following outlines the character comparisons performed by the Boyer-Moore algorithm.

```
agcgcctgatagcgacagt
agcga
```

1 comparison (`c` appears in string so move along so `c`'s line up)

```
agcgcctgatagcgacagt
  agcga
```

1 comparisions (`t` does not appears in string so move along by the length of the string)

```
agcgcctgatagcgacagt
       agcga
```

1 comparison (`g` appears in string so move along so `g`'s line up)

```
agcgcctgatagcgacagt
        agcga
```

1 comparison (`c` appears in string so move along so `c`'s line up)

```
agcgcctgatagcgacagt
          agcga
```

5 comparisons (string has been found) 9 comparisons in total.