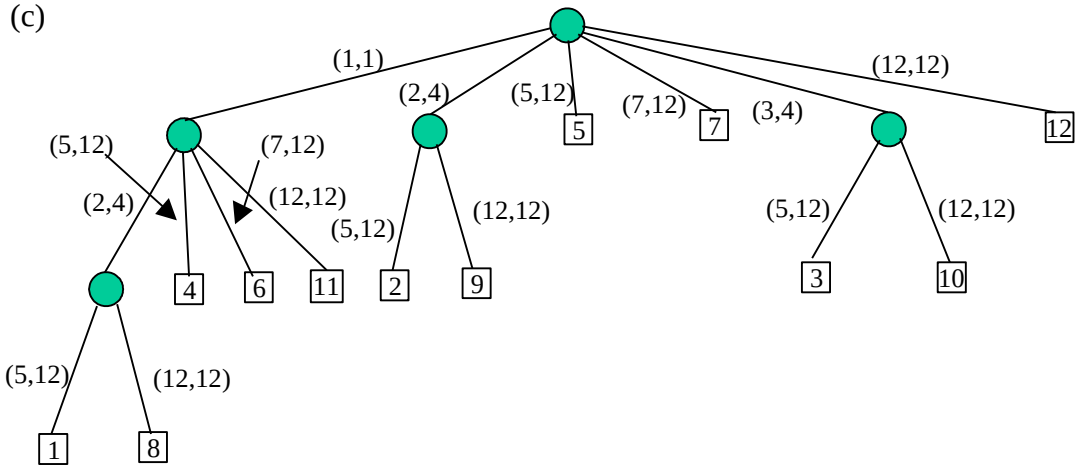
Algorithmics II (H)

Tutorial Exercises on String and Text Algorithms

Outline Solutions

1. Suppose that the tree has m branch nodes and a total of x edges. Each node, except for the root, has one incoming edge, so $x = m + n - 1$. However, every branch node has at least two children, so $x \geq 2m$. It follows that $m + n - 1 \geq 2m$, and therefore that $m \leq n - 1$.

2. (c)



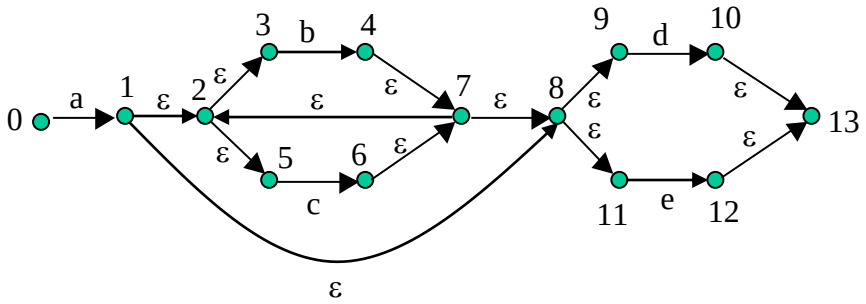
3. The sum of the lengths of the edge labels is the same as in the corresponding suffix trie. Consider the suffix trie for a string of length n in which no symbol is repeated (which is a valid case). The sum of the lengths of the edge labels is $1 + 2 + 3 + \ldots + n = n(n+1)/2$. Hence the result. (In fact the quadratic bound will apply in almost all cases, but it is not so easy to prove it analytically except in special cases such as the one given here.)

   As a side-remark, it is worth noting that there are examples showing that suffix tree construction can take $O(n^2)$ time using the edge-splitting algorithm even with short edge labels. This holds even for an alphabet of size 1, for example if we consider a string of length n comprising character `a` only.

4. Build the suffix tree T for the string S appended with a unique termination symbol \$. The non-repeated substrings of S correspond to leaf nodes w whose incoming edge label is of length >1 (i.e., it is not merely '\$'). Note that such a w must exist, for if v is a branch node with two leaf children, both of whose incoming edge labels are of length 1, then these labels are both '\$' and therefore equal, a contradiction. Let k be the suffix number of such a node w and let d be the string depth of w's parent v. The corresponding non-repeated substring is then the substring $S(k \,..\, k+d)$ of S. Using a traversal of T, we can choose w satisfying the above property such that w's parent v has minimum string depth, in order to obtain a shortest unique substring of S in $O(n)$ time.

5. Break the circular string at some arbitrary position to obtain a string *S*, and build the suffix tree for *S′*=*SS*\$ (where *S′* is the string *S* concatenated with itself and the symbol \$, where \$ is a symbol lexicographically greater than any character in *S*). Follow a path from the root to a leaf by taking, at each step, the branch whose label starts with the lexicographically smallest character. Since \$ is lexicographically greater than any character in *S*, we are guaranteed to reach a suffix of *S′* beginning in the first copy of *S*. The suffix number in that leaf indicates the position of *S* where the circular string should be broken. Since suffix tree construction can be completed in linear time, this is a linear-time solution.

6. * is the closure of the 'or' of every symbol in the alphabet.

7. 

8. Following the notation of the lecture notes:

X = {0}

J = 1:	Y = {0, 1}	X = {0, 1, 2, 3, 5, 8, 9, 11}
J = 2:	Y = {0, 4}	X = {0, 2, 3, 4, 5, 7, 8, 9, 11}
J = 3:	Y = {0, 4}	X = {0, 2, 3, 4, 5, 7, 8, 9, 11}
J = 4:	Y = {0, 1}	X = {0, 1, 2, 3, 5, 8, 9, 11}
J = 5:	Y = {0, 6}	X = {0, 2, 3, 5, 6, 7, 8, 9, 11}
J = 6:	Y = {0, 4}	X = {0, 2, 3, 4, 5, 7, 8, 9, 11}
J = 7:	Y = {0, 1}	X = {0, 1, 2, 3, 5, 8, 9, 11}
J = 8:	Y = {0, 4}	X = {0, 2, 3, 4, 5, 7, 8, 9, 11}
J = 9:	Y = {0, 10}	X = {0, 10, 13}

The Halt state belongs to X and therefore the algorithm terminates, having found a substring that matches the given regular expression.

9. Possibilities include

- a list of integers, possibly ordered
- an array of Boolean

The key operations are '`Extend_by_ε`' and '`Extend_by_Char`'. It may be difficult to obtain O(*n*) time for '`Extend_by_Char`' and O(*n*+*m*) time for '`Extend_by_ε`' using the former representation; every time a further state is generated from the digraph, its membership in the list has to be checked. On the other hand, the latter representation allows these bounds to be achieved easily.

10.

		d	***c***	b	***a***	***b***	***c***	a	c	d	a
	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1	1
b	0	0	0	1	1	2	2	2	2	2	2
b	0	0	0	1	1	2	2	2	2	2	2
a	0	0	0	1	2	2	2	3	3	3	3
d	0	1	1	1	2	2	2	3	3	4	4
c	0	1	2	2	2	2	3	3	4	4	4
a	0	1	2	2	3	3	3	4	4	4	5
d	0	1	2	2	3	3	3	4	4	5	5
b	0	1	2	3	3	4	4	4	4	5	5
c	0	1	2	3	3	4	5	5	5	5	5

11. Form an alignment of the two strings in which the characters of a longest common subsequence match. For example:

```
a  b  b  a  d  c     a  d  b        c
            |  |     |     |        |
            d  c  b  a     b  c  a  c  d  a
```

The alignment is a common supersequence of the two strings, and its length is clearly $m + n - \ell$ (since ℓ characters are 'saved' from the total of $m + n$). No common supersequence can be shorter, as this would mean 'saving' more than ℓ characters, and therefore would imply an even longer common subsequence.

12. In this question and the next, we assume a Boolean function, based on memoisation, that determines whether a given array element has been evaluated.

```
/** Recursive function to return the ith Fibonacci number;
  * uses memoisation in global array f */
public int fib(int i){
  if (!evaluated(i))
    if ( i <= 1 )
      f[i] = 1;
    else
      f[i] = fib(i-1) + fib(i-2);
  return f[i];
}
```

13. The recurrence relation is as follows:

$$d_{i,j} = \begin{cases} d_{i-1,j-1} & \text{if } X(i) = Y(j) \\ 1 + \min(d_{i,j-1}, d_{i-1,j}, d_{i-1,j-1}) & \text{otherwise} \end{cases}$$

subject to

$d_{i,0} = i$ for all i
$d_{0,j} = j$ for all j

```
/** Recursive function to return the edit distance between
  * x(1..i) and y(1..j); uses memoisation in global array d */
public int dist(int i, j){
  if (!evaluated(i, j))
    if ( i==0 )
      d[i][j] = j;
    else if ( j==0 )
      d[i][j] = i;
    else if (x.charAt(i)== y.charAt(j))
      d[i][j] = dist(i-1,j-1);
    else
      d[i][j] = 1 + Math.min(dist(i-1,j),
                             dist(i,j-1),
                             dist(i-1,j-1));
  return d[i][j];
}
```