School of Computing and Information Systems
COMP20007 Design of Algorithms
Semester 1, 2020
End-of-Semester Test
Solutions

Question 1 [5 Marks]

(a) **Solution:** $f(n) \in \Omega(g(n))$

Justification: $f(n) = 4n^8$ and $g(n) = 4n^5 + 10n$, and $n^8 \succ n^5$.

(b) **Solution:** $f(n) \in \Theta(g(n))$

Justification: $f(n) = \log_2(3) + \log_2(n) \in \Theta(\log n)$ and $g(n) = \log_2(n^3) = 3\log_2(n) \in \Theta(\log n)$.

(c) **Solution:** $f(n) \in \Omega(g(n))$

Justification: $f(n) = 3^{2n} = 9^n$ and $g(n) = 3^{n+1} = 3 \times 3^n$, and exponentials with a larger base have a higher growth rate.

(d) **Solution:**

$$\begin{aligned}
T(n) &= T(n-1) + 4n \\
&= T(n-2) + 4(n-1) + 4n \\
&= T(n-3) + 4(n-2) + 4(n-1) + 4n \\
&\vdots \\
&= T(n-k) + 4(n-(k-1)) + 4(n-(k-2)) + \cdots + 4(n-1) + 4n \\
&\vdots \quad \text{let } k = n \\
&= T(0) + 4(1) + 4(2) + \cdots 4n \\
&= 20007 + 4 \times \frac{n(n+1)}{2} \\
&= 20007 + 2n(n+1) \\
&\in \Theta(n^2)
\end{aligned}$$

Question 2 [6 Marks]

(a) **Solution:** The second smallest element must be in the second layer, it's the left or right child of the root.

Justification: It can't be in the first layer because the root must be the smallest element in a min-heap, not the second smallest. All of a node's ancestors must be strictly smaller than it (in a min-heap with distinct values) and so the 2nd smallest element cannot be in layer 3 or lower as it would have 2 or more ancestors, and thus at least 2 values smaller than it, contradicting the fact that it is the 2nd smallest element.

(b) **Solution:** The time complexity is $\Theta(1)$ or $O(1)$ to find the third smallest element in a min-heap.

Justification: The element must be found in the second or third layer and thus there are only 6 possible positions for the element (a constant number of positions). The reason it must be in the 2nd or 3rd layer is similar to (a), if it were in the 4th layer or lower there would be at least 3 elements smaller than it. It can be in the second layer, in which case the other node on that layer is the 2nd smallest element.

To justify why we can find the element in constant time we can consider first finding the 2nd smallest element (by taking the minimum of the left and right children of the root) and then just running a for loop over the 6 possible indices for the 3rd smallest, finding the smallest which is not equal to the 2nd smallest. The size of the heap doesn't affect how many operations happen in these first 3 layers so it must be $\Theta(1)$.

(c) **Solution:** It is not possible for AVL trees.

Justification: Since an AVL tree is not necessarily complete (*i.e.,* all layers filled except the last which must be filled from the left to the right) giving a (level-order) ordering of the nodes doesn't uniquely determine the structure of the tree, as it does in a heap. ✱

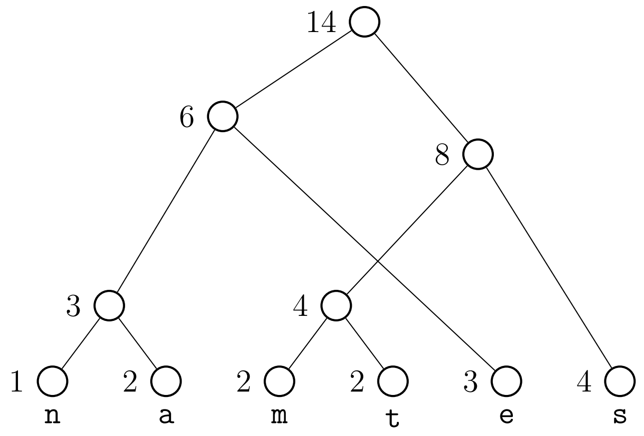
Alternate Solution: It is possible for AVL trees however the array will have to have many elements storing a value indicating a "gap" in the tree.

Question 3 [4 Marks]

The character frequencies, in order, are:

n	a	m	t	e	s
1	2	2	2	3	4

The code tree is:



(a) So with each left branch being a 0 and each right branch being a 1 we get:

`n` :	000
`a` :	001
`m` :	100
`t` :	101
`e` :	01
`s` :	11

(b) Using these codes to encode `state` we get:

00 101 001 101 01

?

with a **length of 13 bits**.

Question 4 [5 marks]

(a) We have $n = 9$ here. 30 is 75% between 0 and 40 so we choose $m = 75\% \times (n - 1) = 6$.

We compare 30 with $A[6] = 20$ and see $k > A[6]$ so we focus on the array $A' = [30, 40]$. Here 30 is 0% between 30 and 40 so we take $m = 0\% \times (n - 1) = 0$ and compare $A'[0] = 30$ with 30 and output that we've found 30 at index 7.

(b)
```
function AssumptionSearch($k, A[0 \ldots (n-1)]$)
    $\ell \leftarrow A[0]$
    $h \leftarrow A[n-1]$
    if $k < \ell$ or $k > h$ then
        return NotFound

    $m \leftarrow ((k - \ell)//(h - \ell)) * (n - 1)$
    if $A[m] == k$ then
        return $m$
    else if $A[m] > k$ then
        return $m + 1 +$ AssumptionSearch($k, A[(m+1) \ldots (n-1)]$)
    else
        return AssumptionSearch($k, A[0 \ldots (m-1)]$)
```

Or, alternatively an iterative approach,

```
function AssumptionSearch($k, A[0..n-1]$)
    $lo \leftarrow 0$
    $hi \leftarrow n - 1$
    while $lo < hi$ do
        if $A[lo] > k$ or $A[hi] < k$ then
            break
        if $lo == hi$ then
            if $A[lo] == k$ then
                return $lo$
            else
                break
        $m \leftarrow lo + (k - A[lo]) * (hi - lo)//(A[hi] - A[lo])$
        if $A[m] == k$ then
            return $m$
        if $A[m] > k$ then
            $hi \leftarrow m - 1$
        else
            $lo \leftarrow m + 1$
    return NotFound
```

(c) The worst-case time complexity for AssumptionSearch is $\Theta(n)$. This occurs when the size of the input array only decreases by 1 with each recursive call.

A concrete example of when this might happen is input which is not linear but growing exponentially, for instance,

$$A = [1, 10, 100, 1000, 10000, 10^5, \ldots, 10^9] \quad k = 10^8$$

At each point the index m will be $\approx 10\%$ of $n-1$ which—after rounding down—will become 0, so we compare k with $A[0]$ and then decide we must search the right component of the array.

We then make a recursive call on $A[1 \ldots (n-1)]$, which results in $m = 0$ again and a recursive call of size $n - 2$, *etc.*, resulting in a $O(n)$ runtime complexity.

END OF TEST