MODEL ANSWER TO INFORMAL COURSEWORK 1

Question 1 (a)

General plan. To compute the time function $t(n)$ for method `divisions`, we proceed in two stages:

(1) identify the primitive operations of the method `divisions`;
(2) compute, for a given method argument n, the number of times each of the operations identified at stage (1) is executed.

When at stage (2), we will see that all operations identified at stage (1) will fall into two categories: (a) operations executed a fixed number of times regardless of n, and (b) operations whose number of executions will depend on n – the larger n, the more times an operation of this category is executed.

Having accomplished (1) and (2), we will be able to find out what $t(n)$ is. Indeed, according to common sense, to compute the time consumed by a method, we just have, for each primitive operation of the method, to multiply the time taken up by the operation by the number of times it is executed, and then sum up all these numbers. Thus, if our operations, as identified at stage (1) above, will be $o_1, \ldots, o_m$, the time needed to execute operation o_i is t_i, and the number of times o_i is executed is n_i (n_i is calculated at stage (2)), then the formula for $t(n)$ will be:

$$t(n) = t_1 \times n_1 + t_1 \times n_1 + \cdots + t_m \times n_m \tag{1}$$

Now, supposing that stages (1) and (2) have already been completed, we know our $o_1, \ldots, o_m$ (stage (1)), we know $n_1, \ldots, n_m$ (stage (2)), but how do we know $t_1, \ldots, t_m$? Since the time a given operation takes up depends on the hardware and software we use, there can be no absolute answer to this question. So the time consumed by every primitive operation is taken to be some arbitrary constant c. What this c is, does not really matter, so we may just as well assume that $c = 1$. Thus, under this assumption, $t_1 = t_2 \cdots = t_m = 1$. With this assumption at hand, it is easy to see that equation (1) becomes:

$$t(n) = n_1 + n_2 + \cdots + n_m \tag{2}$$

Now we turn to stage (1).

Stage 1. At this stage, we identify the primitive operations of method `divisions`.

o_1: declaration of an integer variable
o_2: assignment to an integer variable
o_3: comparison of two integers
o_4: division by 2
o_5: incrementing an integer variable
o_6: return of a method - usually not counted, anyway only happens once.

Note, that in some cases, several operations are disguised as a single Java statement. For example, operations o_2 and o_4 are carried out by a single Java statement `k++;`. Having identified the primitive operations of method `divisions`, we can now turn to stage (2).

```
1 int divisions(int n){      Operations:
2    int k = 1;              o1, o2
3    while(n > 1){           o3
4       n = n/2;             o2, o4
5       k++;                 o2, o5
6    }
7    return k;               o6
8 }
```

Stage 2. At this stage, for each of the operations o_1 through o_6 above, we have to compute the number of times the operation is executed.

Operations o_1 and o_6 are each executed only once; thus, $n_1 = n_6 = 1$. The number of executions of each of these operations does not depend on the value of n.

For all the other operations, the number of times they are carried out does depend on n, since they are executed inside the `while` loop, and the number of iterations of the loop depends on n (as n occurs in the condition of the loop). Each of operations o_3–o_5 is executed once inside each iteration of the loop. $o2$ is executed once before the loop and twice inside each iteration of the loop. Thus, all we have to do is compute the number of iterations of the loop. It will obviously depend on n, but how?

For simplicity, let's first assume that $n = 2^x$. (This way, division by 2 will always produce a number divisible by 2 without remainder.) Then, by definition of $\log_2$, $x = \log_2 n$. At each iteration of the loop, n is divided by 2, which means setting the value of n to 2^{x-1}. This is done until the condition $n > 1$ fails. In our case, it means that $n = 1$ (since we always get, by dividing by 2, a whole number divisible by

2, we will sooner or later reach 1), that is $n = 2^0$. It is clear that, to reach this stage, we have to go through the loop exactly $x - 0 = x$ times. Since, as we have noted earlier, $x = \log_2 n$, the loop iterates $\log_2 n$ times.

Now, let's remove our simplifying assumption, thus allowing n to be any natural number. Then, there exists a natural number n' such that $n' \leq n$, $n' = 2^x$ for some x, and no other number with these two properties lies between n and n' (in other words, n' is the largest number that is smaller than or equal to n and that is some power of 2.) It is not difficult to see that in such a case the loop iterates exactly $\log_2 n'$ times. It is also easy to see that $\log_2 n' = \lfloor \log_2 n \rfloor$. (We remind that $\lfloor x \rfloor$ is, by definition, the largest whole number less than or equal to x.) Thus, our loop iterates $\lfloor \log_2 n \rfloor$ times. Since this case generalises the one considered in the foregoing paragraph, this is our answer to the question of how many times the `while` loop iterates: $\lfloor \log_2 n \rfloor$ times.

Since, as we have established above, the number of iterations of the loop is the number of times each of operations o_3, o_4, o_5 are executed, we now know that each of those operations is executed $\lfloor \log_2 n \rfloor$ times. The number of times o_2 is executed is 1 (before the loop) plus $2 \times \lfloor \log_2 n \rfloor$ (twice in each iteration).

Final computation. Now that we know the number of times each of our primitive operations is executed, we can finally compute the function $t(n)$ according to the equation (2) above:

$$\begin{aligned} t(n) &= n_1 + n_2 + n_3 + n_4 + n_5 + n_6 = \\ &\quad 1 + (1 + 2 \times \lfloor \log_2 n \rfloor) + \lfloor \log_2 n \rfloor + \\ &\quad \lfloor \log_2 n \rfloor + \lfloor \log_2 n \rfloor + 1 = \\ &\quad 3 + 5 \times \lfloor \log_2 n \rfloor \end{aligned}$$

Thus, and this is our final answer, $t(n) = 3 + 5 \times \lfloor \log_2 n \rfloor$.

Question 1 (b)

By definition, $t(n)$ is in $O(log_2 n)$ if there exist positive numbers c and n_0 such that $t(n) \leq c \times log_2 n$ for all $n > n_0$.

Therefore, to prove that $t(n)$, that is $3 + 5 \times \lfloor \log_2 n \rfloor$, is in $O(log_2 n)$, we have to show that there exist positive numbers c and n_0 such that $3 + 5 \times \lfloor \log_2 n \rfloor \leq c \times log_2 n$ for all $n > n_0$.

We start by observing that, by the definition of the floor function, $\lfloor \log_2 n \rfloor \leq \log_2 n$, and hence $5 \times \lfloor \log_2 n \rfloor \leq 5 \times \log_2 n$.

Furthermore, let's note that if $n \geq 2$ then $1 \leq \log_2 n$, and hence $3 \leq 3 \times \log_2 n$.

From these two observations we may infer that if $n \geq 2$ then $3 + 5 \times \lfloor \log_2 n \rfloor \leq 3 \times \log_2 n + 5 \times \log_2 n$; that is $3 + 5 \times \lfloor \log_2 n \rfloor \leq 8 \times \log_2 n$ for all $n \geq 2$.

Thus, we have proven that $3 + 5 \times \lfloor \log_2 n \rfloor \leq c \times log_2 n$ for all $n > n_0$, where $c = 8$ and $n_0 = 2$.

Question 1 (c)

To compute the space function $s(n)$ for method `divisions`, we proceed, like in the answer to question 1 (a), in two stages:

(1) identify those (and only those) statements that allocate new chunks of memory to the method;
(2) for each of the statements allocating new memory, compute how many units of memory are allocated

Having accomplished (1) and (2), we will be able to find out what $s(n)$ is by adding up extra memory allocated by each statement.

Which statements allocate memory? In Java, new memory is allocated when

(1) new variables are declared - allocating spece to hold the value of the variable of the basic type, or a reference to a variable of the object type
(2) when keyword `new` is used as in `int[] arr = new int[10]` which allocates memory for a new data structure. We assume that the number of units of memory for an integer array of size k is k times the unit for holding an integer.

Statements allocating new memory in `divisions`. The only statement allocating new memory in `divisions` is `int k = 1` because this declares a new variable `k`. It allocates one unit of memory.

Final computation. We can now compute $s(n)$:

$$s(n) \quad = \quad 1.$$

Question 1 (d)

By definition, $s(n)$ is in $O(1)$ if there exist positive numbers c and n_0 such that $t(n) \leq c \times 1$ for all $n > n_0$.

Therefore, to prove that $s(n)$, that is 1, is in $O(1)$, we have to show that there exist positive numbers c and n_0 such that $1 \leq c \times 1$ for all

$n > n_0$. It is obvious that this inequality holds with $c = 1$ and any choice of n_0 (so we can set n_0 to 0).

Question 2 (a)

No. For two reasons. (1) It will make not exactly n^2 steps, but at most $c \times n^2$ steps, for some positive number c. (2) This will only happen as the size of input grows "large enough". Nothing is promised for input of very small size.

Question 2 (b)

No. With input of very small size, an algorithm with complexity $O(n)$ may run faster than an algorithm with complexity $O(log_2 n)$. The latter algorithm will beat the former as the size of input grows "sufficiently large".

Question 2 (c)

Yes. Since $c_1 \times log_2 n = 2 \times c_2 \times log_4 n$, which means that $c_1 \times log_2 n$ is in $O(c_2 \times log_4 n)$ (with $c = 2$ and $n_0 = 0$); and $c_2 \times log_4 n = \frac{1}{2} \times c_1 \times log_2 n$, which means that $c_2 \times log_4 n$ is in $O(c_1 \times log_3 n)$ (with $c = \frac{1}{2}$ and $n_0 = 0$). Since the algorithms are in each other's O, they are of the same complexity.