Question 1. What is the correct recurrence relation for computing the complexity of the code below?

```
function recursiveProblem(n) is:
//n is a positive number
    if n ≤ 1 then
        recursiveProblem ← 1
    else
        recursiveProblem ← 1 + recursiveProblem(n - 5)
    end-if
end-function
```

Select one or more:

- ☐ $T(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ T(\frac{n}{5})+1 \text{ otherwise} \end{cases}$
- ☐ $T(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ T(n-5) \text{ otherwise} \end{cases}$
- ☑ $T(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ T(n-5)+1 \text{ otherwise} \end{cases}$
- ☐ $T(n) = \begin{cases} 1 \text{ if } n \leq 1 \\ T(\frac{n}{5})+n \text{ otherwise} \end{cases}$

Question 2. Consider the code below.

From the previous question you know that the recursive formula for this function is:

$$T(n) = \{1, if\ n \leq 1$$
$$T(n-5) + 1\ otherwise$$

If you want to compute the complexity, what notation/assumption do you need to introduce for n?

Select one or more:

☒ n = 5k

☐ None of the other options

☐ $n = 5^k$

☐ $n = 2^k$

In the lecture we used $n=2^k$ for both examples, but the reason for this was the fact that we had n^2 in the recursive model.

Now that we have n - 5 in the recursive model, we need the notation $n = 5^k$.

The correct answer is:

$n=5^k$.

Question 3. For the complexity of a recursive function we will always use the Θ notation, because we cannot have a best case situation.

Select one:

☐ True

☒ False

False, actually our first example, the recursive binary search had a best case and we said that the complexity is $O(\log_2 n)$.

The correct answer is 'False'.

Question 4. If an array of integer numbers starts at the memory address 1038 and an integer number uses 4 bytes, what is the address of the element from position 100 (assume that the first element is at position 0)?

Select one or more:

☐ 1038

☐ 1436

☒ 1438

☐ We cannot determine.

We have discussed the following formula: address of element from position i = address of array + i * size of an element.

If we put our values in the equation we will have: address of element from position 100 = 1038 + 100 * 4 = 1438.

The correct answer is:

1438.

Question 5. Which of the following is a good resize strategy for a full Dynamic Array?

Select one or more:

☒ Multiply the current capacity by 2 (so newCapacity = 2 * capacity)

☐ Multiply the current capacity by 1 (so newCapacity = 1 * capacity)

☒ Multiply the current capacity by 1.5 (so newCapacity = 1.5 * capacity)

☐ Multiply the current capacity by 0.7 (so newCapacity = 0.7 * capacity)

☒ Multiply the current capacity by 1.3 (so newCapacity = 1.3 * capacity)

☐ Add 100 more empty positions to the capacity (so newCapacity = capacity + 100)

☐ Add 1 more empty position to the capacity (so newCapacity = capacity + 1)

The correct answers are:

Multiply the current capacity by 1.5 (so newCapacity = 1.5 * capacity),

Multiply the current capacity by 2 (so newCapacity = 2 * capacity),

Multiply the current capacity by 1.3 (so newCapacity = 1.3 * capacity)

Question 6. What is the complexity of the addToPosition operation from the dynamic array?

Select one or more:

☐ Θ(1)

☐ Θ(1) - amortized

☒ O(n)

☐ Θ(n)

☐ O(n) - amortized

The addToPosition operation can be divided into two parts: resize the array (if necessary) and move elements to make the position where we add the new element empty.

The resize part is Θ(1) amortized (it happens rarely), but moving the elements is an O(n) operation (can have best case, worst case, average case). Together, the complexity is going to be O(n).

Another way of thinking about whether we have amortized complexity or not, is to ask the question: Can I have worst case performance twice in a row?

We will have worst case complexity in one of the following two situations: we need a resize (and this cannot happen twice in a row) or we add to position 1 and we need to move every element (and this can happen any number of times in a row). So we do not have amortized complexity.

The correct answer is:

O(n).

Question 7. What is the complexity of removing the first occurrence of a given element from a dynamic array (assume that no resize happens, but the order of the elements has to be kept)?

Select one or more:

☐ Θ(1)

☐ Θ(1) - amortized

☐ O(n)

☒ Θ(n)

☐ O(n) - amortized

Since we want the first occurrence, we should start checking the elements from position 1 and go towards the end of the array, so when we find the element, what we have is the first occurrence. This part has an O(n) complexity (best case is when you find the element on position 1, worst case is when it is not in the array).

Once you found the element, removing it means to move all the elements after it to the left with one position. This part is O(n) as well (you might need to not move any element, or you might need to move all elements).

So it seems like complexity is O(n). However, if we look at the two parts together, we can see that overall, we never have a best case situation: what is the best case situation for the searching part (finding the element on the first position) is actually the worst case situation for the removing part. So our algorithm will always check all positions of the array: some of them during the searching and the rest during the element moving part of the removal.

The correct answer is:

Θ(n).

Question 8. In order to avoid having a Dynamic Array with too many empty slots, we can resize the array after deletion as well. Which of the following two strategies do you thing would be best for resizing?

Select one or more:

☒ Wait until the array is a quarted full (da.nrElem ≈ da.cap/4) and resize it to half of the capacity (da.cap <- da.cap /2)

☐ Both are equally good.

☐ Wait until the array is half full (da.nrElem ≈ da.cap/2) and resize it to half of the capacity (da.cap <- da.cap /2)

The main idea is to avoid having worst case twice in a row. And if you resize at half of the capacity when the array is half full, it will result in a full array, and then if you add an element, you need resize again. And if you remove it, you will resize again...so that is not a good strategy.

The correct answer is:

Wait until the array is a quarted full (da.nrElem ≈ da.cap/4) and resize it to half of the capacity (da.cap <- da.cap /2)