Question 1. [9 MARKS]

Consider the following Python code which simulates rolling a pair of dice and counting the number of rolls until we reach n pairs. We are interested in the number of times the `random.randint` method is called, which corresponds to the number of rolls. *Do not express the complexity in O notation but give exact expressions.*

```
from random import randint

# Attempt to roll n pairs. Use a maximum of 10n rolls.
def countRolls(n):
    count = 0
    tries = 0
    while count < n and tries < 10*n:
        die1 = randint(1,6)      # roll the first die
        die2 = randint(1,6)      # roll the second die
        if die1 == die2:
            count += 1
        tries += 2       # count these 2 rolls even if we don't get a pair
    return count, tries
```

Part (a) [1 MARK]

Perform a best-case analysis of `countRolls`.

SAMPLE SOLUTION: The best case occurs when the first n pairs of rolls are all pairs. This takes 2n calls to randint. You can't possibly do better because to get n pairs you must have at least 2n numbers rolled.

Part (b) [3 MARKS]

Perform a worst-case analysis of `countRolls`.

SAMPLE SOLUTION:

- Worst-case occurs when the n'th pair occurs last or when n pairs do not occur in the 5*n tries.
- For each of 5*n tries we do 2 rolls for a total of 10*n rolls.
- We can't do worse than this because the algorithm stops after 10*n rolls regardless of how many pairs have been found. It can only stop earlier than this, not run longer.

Part (c) [5 MARKS]

Perform an average-case analysis of `countRolls`. You do not need to simplify your expressions.

SAMPLE SOLUTION: There are at most 5*n attempts to roll a pair. We will count the attempts from 1 to $5n$. The probability of rolling a pair on each attempt is $\frac{1}{6}$. Each of these attempts is independent. The probability of rolling the n^{th} pair on the i^{th} roll is given by

$$p_{n,i} = \begin{cases} 0 & 1 \leq i < n \\ \frac{1}{6}^n & i = n \\ \frac{1}{6}^n \frac{5}{6}^{i-n} \binom{i-1}{n-1} & n < i \leq 5 * n \end{cases}$$

Then you still need the probability that we roll all 5*n attempts and don't get n pairs. We can calculate this as follows:

$$p_{\text{not found}} = 1 - \sum_{i=1}^{5*n} p[n^{th}\text{pair found on roll}i]$$

Using both those pieces we calculate the average number of rolls as:

$$10n * p_{\text{not found}} + \sum_{i=1}^{5n} (2i * p_{n,j})$$

Question 2. [14 MARKS]

You are designing an ADT that contains information about students. Each student has a name, an age, and a score. In addition to being able to insert and delete students, you must provide the following new operation in worst case complexity $\mathcal{O}(\log n)$ where n is the number of students. You may assume that the ages are unique.

- FINDBESTWITHINAGE(*agelimit*): returns the student with the highest score from all students whose age does not exceed *agelimit*.

You will accomplish this by augmenting an AVL tree.

Part (a) [1 MARK]

What will you use as the key for the tree?

SAMPLE SOLUTION: age — if we needed to look up students by name we would need an additional data-structure to map from name to age – another AVL tree or a hashtable would do.

Part (b) [2 MARKS]

What additional information will you store at each node?

SAMPLE SOLUTION: We would need name and score but also the maximum score of the nodes in the subtree rooted at this node. Call it `max_in_subtree`.

Part (c) [2 MARKS]

Draw a valid AVL tree for the following records showing any additional information.

Name	Age	Score
Don	8	172
Eshan	10	180
Ken	14	190
Kevin	9	165
Matt	11	185
Nick	6	167
Sam	7	169
Scott	12	187

SAMPLE SOLUTION: Lots of options that are valid. The tree must obey the AVL balance property.

Part (d) [4 MARKS]

Give the algorithm for FINDBESTWITHINAGE(*agelimit*) explaining briefly why it is $\mathcal{O}(\log n)$.

SAMPLE SOLUTION:

```
    def find_best(age_limit, current_root):

        if current_root == NIL:  #this age_limit is not in the tree
            return -infinity
        elif current_root.age == age_limit:
            return max(current_root.score, current_root.left.max_in_subtree)
        elif age_limit < current_root.age:  # going left
            return find_best(age_limit, current_root.left)
        elif age_limit > current_root.age:  # going right
            #root and all values in left are less than age_limit
            return max(current_root.score, current_root.left.max_in_subtree,
                 find_best(age_limit, current_root.right))

    FindBestWithinAge(age_limit):
        return find_best(age_limit, root)
```

Note: Create a NIL node for every left and right child where none exists.
NIL.max_in_subtree = -infinity

Part (e) [3 MARKS]

What else do you need to be concerned with when you augment this data-structure? Address this concern here.

SAMPLE SOLUTION: You need to keep the new information `max_in_subtree` correct on insertions and deletions (including any rotations) on the AVL tree.

Part (f) [2 MARKS]

The assumption that we have only one student in each category is ridiculous. What would have to change in the data structure or algorithms to accommodate non-unique ages? How would those changes affect the runtime of the operations?

SAMPLE SOLUTION: Now, when we find a node with the age limit, we can't assume that this is the only such node: there might be other equal nodes in the left or right subtree. The left subtree is already handled by our code. So, we additionally have to explore the right subtree by also including `find_best(age_limit, current_root.right)` in the `max` call. The runtime is unchanged.