1. (a)

Considering rooted binary tree with n nodes, we can construct such a tree by breaking into smaller subproblems. Starting by selecting a node to be the root of the tree. This root node can be any of the n nodes in the tree, so there are T(0) possibilities for the nodes on the left subtree and T(n-1) possibilities for the nodes on the right subtrees. This accounts for the term T(0)T(n-1). Continuing this process, choosing the last node to be the root (T(n-1)T(0)). Summing all these possibilities gives us the total number of binary tree configurations with n nodes.

Therefore, $T(n) = T(0)T(n-1) + T(1)T(n-2) + \cdots + T(n-1)T(0)$

(b)

```
function countTrees(n):
    # Initialize a 2D array dp
    dp = new Array(n+1, n+1)

    # Base case: There is one binary tree with zero nodes
    for i = 0 to n:
        dp[i][0] = 1

    # Build the dp table bottom-up
    for i = 1 to n:
        for j = 1 to i:
            dp[i][j] = 0
            for k = 0 to j-1:
                dp[i][j] += dp[i-k-1][k] * dp[k][j-1]

    # The result is stored in dp[n][n]
    return dp[n][n]
```

dp[i][j] represents the number of rooted binary tree configurations with 'i' nodes using the first 'j' elements of the Catalan numbers sequence.

(c) Let T(n) represent the number of such non-empty binary trees with n nodes. The recurrence relation is as follows:

$T(n) = T(n-2)\cdot T(0) + T(n-4)\cdot T(2) + \cdots T(0)\cdot T(n-2)$ there are T(0) possibilities for the nodes on the left subtree and T(n-2) possibilities for the nodes on the right subtrees.

2. (a) Let T(n) represents the minimal total penalty for hiking up to campground 'n'

Base case: $T(1) = (15 - A[1])^2$

Recursive case: for $n > 1$, we can consider two options:

① Daphne reaches the campground 'n' directly from the previous campground 'n-1'

② skips one or more campground in between to reach campground 'n'

$\therefore T(n) = \min\left((15 - (A[n] - A[i])^2 + T(i)\right)$

(b) Create a 1D dp array of size N+1, where N is the size N+1, where N is the size of S

Consider two pointers i and j, where i refers the strarting of the substring and j represents the ending of the substring. Run two nested loop, i=0 till N+1 and j=0 to j=i:

check if dp[j] > 0 and the dictionary of words contains contains the substring, then mark dp[i] = true

Return dp[N] > 0

```
Function wordBreak(s, wordDict):
    # Create a set of words for faster lookup
    word_set = Set(wordDict)

    # Initialize a dynamic programming table dp
    dp = Array of size (length of s + 1)
    dp[0] = true

    # Loop over the characters in the input string s
    for i from 1 to length of s:
        dp[i] = false

        # Loop over all possible substrings ending at position i
        for j from 0 to i:
            # Check if dp[j] is true (substring s[0:j] can be segmented)
            # and if s[j:i] is in the word set
            if dp[j] and s[j:i] in word_set:
                dp[i] = true
                break

    # The value in dp[length of s] represents whether the entire string can be segmented
    return dp[length of s]
```

3. (a) Let 'n' be the number of sculptors in the room.
Generating all possible combinations of sculptors that leave through exit 1 can be done using a recursive approach, and the number of such combinations is 2^n
(each sculptor can either be in exit 1 or exit 2, so there are 2 choices for each)

For each combination, we need to compute the sum of times for exit 1 and exit 2. Computing the sum of times for each exit takes O(n) time

We also need to find the combination with the smallest absolute difference.
The overall time complexity of the naive algorithm is $O(n \cdot 2^n)$

(b)

```
To solve this problem using dynamic programming, we can define a two-dimensional DP table dp,
where dp[i][j] represents whether it's possible to achieve a difference of j between the exit times
of the two groups when considering the first i sculptors. Here's the recurrence relation:

Let:

n be the number of sculptors.
w be an array of sculptor exit times.
C be the total time it takes for all sculptors to exit, i.e., C = sum(w[i] for i in range(n)).

dp[i][j] = true, if there exists a subset of the first i sculptors whose total exit time is j
dp[i][j] = dp[i-1][j] || dp[i-1][j - w[i]], otherwise

Function MinExitTimeDifference(w):
    n = length of w
    C = sum of all elements in w
    dp = 2D array of size (n+1) x (C+1)

    # Initialize the DP table
    for i from 0 to n:
        dp[i][0] = true  # It's always possible to achieve a difference of 0

    # Fill the DP table
    for i from 1 to n:
        for j from 1 to C:
            dp[i][j] = dp[i-1][j]  # Option 1: Exclude the i-th sculptor
            if j >= w[i-1]:
                dp[i][j] = dp[i][j] or dp[i-1][j - w[i-1]]  # Option 2: Include the i-th sculptor

    # Find the minimum difference
    minDiff = infinity
    for j from 0 to C//2:
        if dp[n][j]:
            minDiff = min(minDiff, C - 2 * j)

    return minDiff
```

4. (a)

When the drone can measure the oxygen density at its current position instantaneously (t_2 = 0),
the optimal search strategy involves minimizing the total travel time (time spent moving between points)
while ensuring that you find the critical point k as quickly as possible.

Here's the optimal search strategy:

Start at position 0 (initial position of the drone).
Begin moving to the right (increasing position) by a fixed step size.
At each step, measure the oxygen density at the current position.
Keep moving to the right until you find the first position k where ak > 1 > ak+1.
This is the critical point you are looking for.
Once you find k, stop the drone and report the value of k.

(b)

When t_1 = 0 (no time to jump between locations) and t_2 is not equal to 0 (time required for measurements),
the optimal search strategy is different. In this scenario,
you want to minimize the time spent measuring the oxygen density
while ensuring you find the critical point k as quickly as possible.
Here's the optimal search strategy:

Start at position 0 (initial position of the drone).

Measure the oxygen density at the current position.

If a_0 > 1 > a_1 (the condition is already met at the starting position), stop the search and report k = 0.

Otherwise, move to the right (increasing position) by one step.

Measure the oxygen density again at the new position.

Compare the new measurement to the previous one:
a. If the oxygen density has increased, continue moving to the right and measuring.
b. If the oxygen density has decreased or remained the same, you have found the critical point k.
Stop the search and report the position where you found a_k > 1 > a_(k+1).