Question 1:
It is asked to find query string ith times in the given bigger string. If it is found ith time return first index of ith occur ence else returns -1

To solve this problem recursive algorithm starts from the first index of the bigger string and continues while compari n indexes of bigger string and query string. If letters in current index are different then passes the first index of the bigger string and continues the next. If they are the same until the end of the query string it increases occurenc e number. If occurence number reaches i before the end of the function then returns first index of ith occurence otherwise returns -1. This operation continues until reaching the end of the bigger string.

Induction Method:
In base case index value of the big string is bigger than bigger string length. Each call (or m call) of the function incr eases the index one so every calls makes progress to the base case.

Question 1 Algorithm Analysis:

$T(n,m) = T(n-1,m) + m$ $T(1) = 1$
$T(n,m) = T(n-2,m) + m + m$
$T(n,m) = T(n-3,m) + m + m +m$
.
.
.
$T(n) = n*T(n-k,m) + k*m$ $n-k = 1$, $T(n-k,m) = 1$ so the algorithm complexity is -> $O(n*m)$
$T(n,m) = T(n-1,m) + m$ -> according to master theorem algorithm comlexity is $O(n*m)$

Best case occurs if it stops before reaching the end of the big string because it can find enough query string to return i'th occurence. i can be 1 and
query string may be in the beginning of the big string. Or just maybe query string is bigger than big string and algorit hm stops directly.

Worst case occurs when the algorithm needs to look m letters (that is the length of the query string) in every letter of big string(starts from a letter in big string and continues m times because all of the letters were the same except the last letter. Then algorithm starts from next letter in big string and this continues).

Best case: $\Theta(1)$
Worst case: $\Theta(n*m)$

Question 2:
It is asked to find number of integers between given top and bottom value in given sorted array

To solve this problem binary search approach is used in recursive function. Firstly the function looks the middle of t he array, if the number is smaller than interval it looks middle of the upper half, if the number is bigger than interval it looks middle of the lower half, if it is in interval it looks both. This operations continues until the function gets 1 length array. For every number occurs in interval it returns 1 otherwise returns 0. In the and function returns number number of integers between given top and bottom value.

Induction Method:
In base case there is one length array. Each call of the function reduces array size to the half so every call makes pro gress to the base case. Each base case has return value and sum of them gives number of values in given interval

Question 2 Algorithm Analysis:
$T(n) = 2*T(n/2) + 1$ $T(1) = 1$

T(n) = 2*(2*T(n/4) + 1) + 1
.
.
.
T(n) = 2^k*T(n/2^k) + 2^(k-1) + 2^(k-2) ... + 1   n/2^k = 1, n = 2^k, T(n/2^k) = 1 so the algorithm complexity is -> O(n)

Best case occurs if there is no value in asked interval. To find this the function needs to be called logn times so it rea ches a state where middle element is only element.

Worst case occurs when every number is in the giver sorted array

Best case: Θ(logn)
Worst case: Θ(n)

Question 3:
It is asked to find contiguous arrays in given unsorted array that sum of their numbers is equal to given integer value.

To solve this problem the function starts from the first number of the unsorted array. If the number is equal to given i nteger value inserts this number's index into the double dimensional ArrayList as a subarray, if it is bigger than given integer value it passes to the next number, if it is smaller then again passes the next number but keeps the index of previous number and sums its value with the new number and compares their values again. With this approach it starts from every index of unsorted array and sums subsequent numbers until reaches the given integer v alue or passes the value. When it reaches given value it takes index values where it started and where it is now. Adds these indexes to the double dimensional ArrayList. To show the subarrays this ArrayList is used.

Induction Method:
In base case index reaches end of the unsorted array so it's like there is no array left. Each call (probably more than o ne call to reach sum value) of the function increases the index one so every calls makes progress to the base case.

Question 3 Algorithm Analysis:
T(n) = T(n-1) + n   T(0) = 1
T(n) = T(n-2) + n + n
T(n) = T(n-3) + n + n +n
.
.
.
T(n) = n*T(n-k) + k*n   n-k = 0, n = k, T(n-k) = 1   so the algorithm complexity is -> O(n^2)
T(n) = T(n-1) + n -> according to master theorem algorithm comlexity is O(n^2).

Worst case occurs when sum of the numbers of the subarrays can't reach the searched value. So every time the algori thm needs to start from index where it left(firstly index 0 than +1 every time) and continue until the end of the serached array.

Best case occurs when every number in the searched array is bigger then the serached value. So the algorithm looks every number once and reaches the end of the array.

Best case: Θ(n)
Worst case: Θ(n^2)

Question 4:

The function multpiles 2 binary numbers by using recursive algorithm.

Question 4 Algorithm Analysis:
T(n) = 3*T(n/2) + 1   T(1) = 1
T(n) = 3*(3*T(n/4) + 1) + 1
.
.
.
T(n) = 3^k*T(n/2^k) + 3^(k-1) + 3^(k-2) ... + 1   n/2^k = 1, k = log2_3, T(n/2^k) = 1   so the algorithm complexity is -> O(n^log2_3)
T(n) = 3*T(n/2) + 1 -> according to master theorem algorithm comlexity is Θ(n^log2_3) n represents the digit number of the given big integer.

The function always runs according to number of digits of bigger integer so the algorithm comlextiy is Θ(n^log2_3).