GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
HW4

Yusuf Arslan
200104004112

1. System Requirement

First of all the operating system should have Java virtual machine (JVM), Java development kit (JDK) and ‘at least’ Java runtime environment (JRE) – 11 in a linux distribution.

The application is designed to display result of some problems that solved by recursive approach. -Problems are defined in the scope of Homework4-

In the 1st problem, programmer should set a large string and another string to search in the large string and enter which number of occurrence that he/she wants to search.

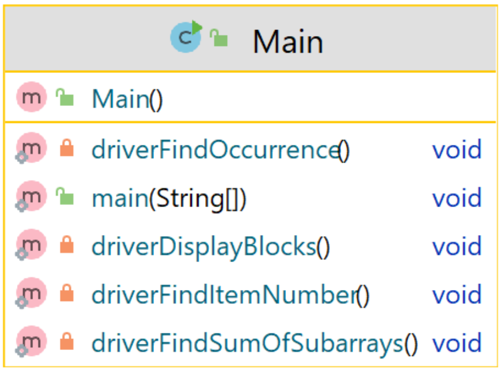
In the 2nd problem, programmer should set an sorted integer array and the range. The function will return number of items in that range in the array.

In the 3rd problem, programmer sets an unsorted array and a target value. Program will display contiguous subarrays which sum of items of this subarrays correspond to the target value.

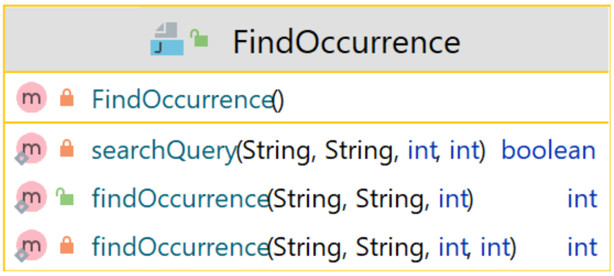
In the 5th problem programmer sets length of the blocks. Program will display sub-blocks in order.

The program can be compiled and run with the makefile to view the pre-made tests.

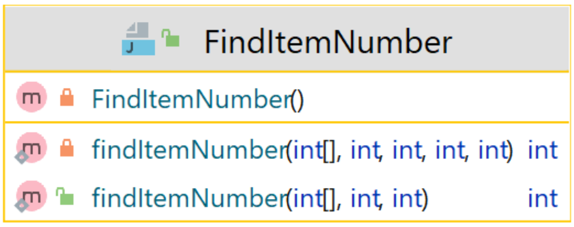
2. Class Diagrams



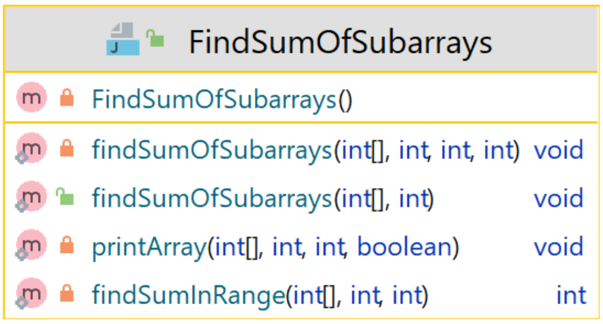
```
Main
Main()
driverFindOccurrence()        void
main(String[])                void
driverDisplayBlocks()         void
driverFindItemNumber()        void
driverFindSumOfSubarrays()    void
```



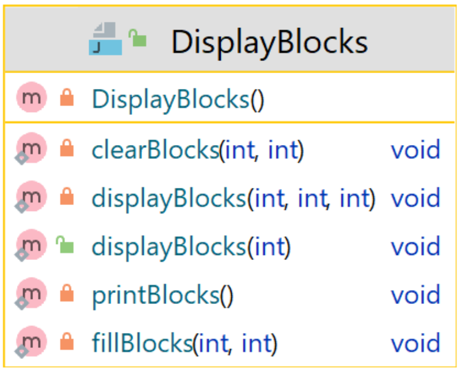
```
FindOccurrence
FindOccurrence()
searchQuery(String, String, int, int)  boolean
findOccurrence(String, String, int)        int
findOccurrence(String, String, int, int)   int
```



```
FindItemNumber
FindItemNumber()
findItemNumber(int[], int, int, int, int)  int
findItemNumber(int[], int, int)            int
```



```
FindSumOfSubarrays
FindSumOfSubarrays()
findSumOfSubarrays(int[], int, int, int)  void
findSumOfSubarrays(int[], int)            void
printArray(int[], int, int, boolean)      void
findSumInRange(int[], int, int)            int
```



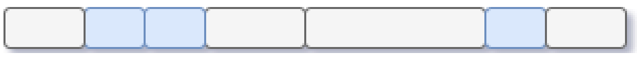
```
DisplayBlocks
DisplayBlocks()
clearBlocks(int, int)           void
displayBlocks(int, int, int)    void
displayBlocks(int)              void
printBlocks()                   void
fillBlocks(int, int)            void
```

3. Problem Solutions Approach

- Problem1: Find the index of the 'n'th occurrence of a substring in a large string.

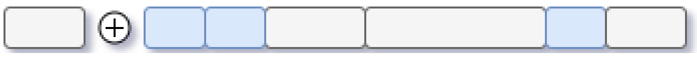
  We can say that a string is the sum of first string and the following strings. (thinking recursively)

  Assume it is our string that we are searching in it:

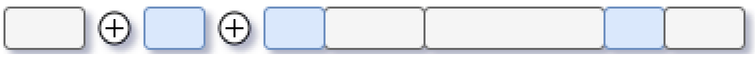


  Every piece of that string is the candidate that we are looking its repeat.

  We know that, it is same as:



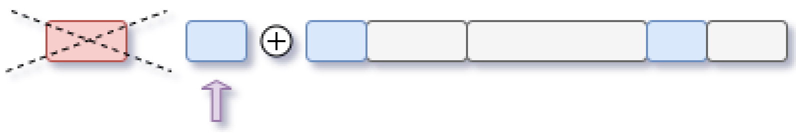
  And also:



  And so on…

  Assume we are looking the occurrences of the blue piece:

  Since string at the head of the string not equal to that blue piece, we can think the remaining string as a separate string.



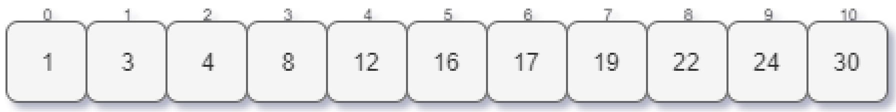
  Now, we encounter with our first occurrence of the string.
  If we are looking for the first occurrence, we can return the index here.
  Otherwise, we will discard this piece as well, in order to find the other occurence requested from us.

- Problem2: Find number of items in the range in a sorted array.

In an ordered array, numbers greater and equal to the lower limit of the range and less and equal to the upper limit of the range will give us the total number in that range.

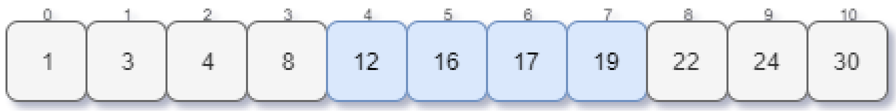
Let's search by splitting the array in half until we find the first number in the given range. After we find that number, keep searching until we find the smallest number inside the border in the array to the left of that number, and until we find the largest number inside the border in the array to the right of that number. Do same operations by constantly narrowing the range.

Let’s assume that we have such a sorted array:



And the range 9 to 19 (inclusive).

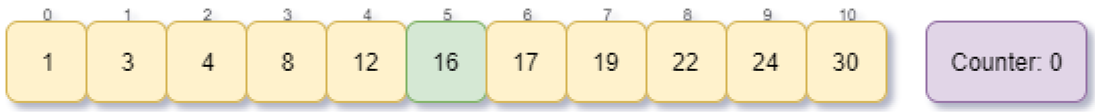
So the items in the range should be as follow.



Let’s look how to find that numbers.

Step1:

Middle = (0 + 10) / 2 = 5
Go to the 5th index.

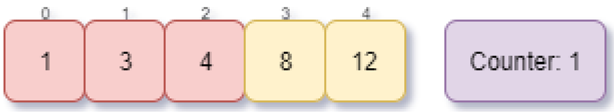


16 in the range, increase counter and check both right and left side of the 5th index.

Step2:

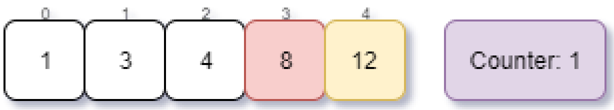
Let's take a closer look at the left side first.

Middle = (0 + 4) / 2 = 2
Go to the 2nd index.



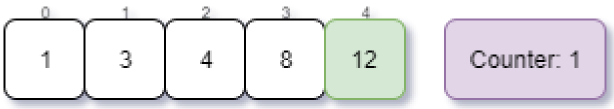
4 is smaller then the lower bound, therefor, we don't need to check the left side of 4 anymore, but still there can be elements in the rigth side.

Middle = floor((3 + 4) / 2) = 3
Go to the 3rd index.



8 is smaller then the lowerbound, therefor, we don't need to check the left side of 8 (in this case, we already knew there is no element on the left side of 8), but still there can be elements in the rigth side.

Middle = (4 + 4) / 2 = 4
Go to the 4th index.

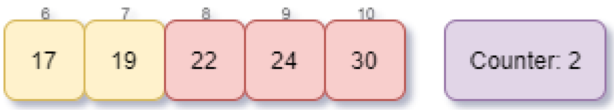


12 is in the range, increase the counter. Finally, there is no element left in the left-side of the array.

Step3:

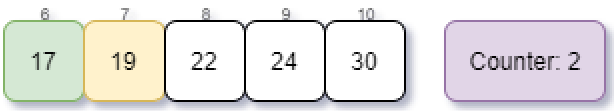
Let's take a closer look at the right side now.

Middle = (6 + 10) / 2 = 8
Go to the 8th index.



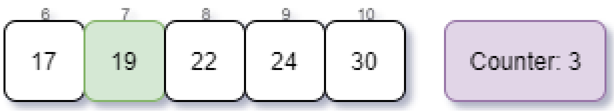
22 is larger then the upperbound, therefor, we don't need to chek the right side of 22 anymore, but there can be element on the left-side.

Middle = floor((6 + 7) / 2) = 6
Go to the 6th index.



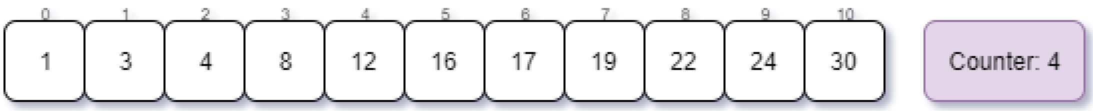
17 is in the range, increase the counter and check the remaining element on the right side of 16.

Middle = (7+7) / 2 = 7
Go to the 7th index.



19 is in the range, increase the counter.

There is no element left in the right-side of the array, too.



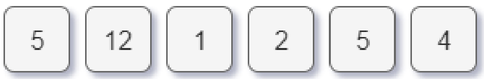
Finally, we can return the counter.

- Problem3: Find contiguous subarrays of a given array which the sum of their items are equal to the given a target value.

We know that, a subarray without the head item is also a subarray, and also, these subarrays without the last items also subarrays. (thinking recursively)

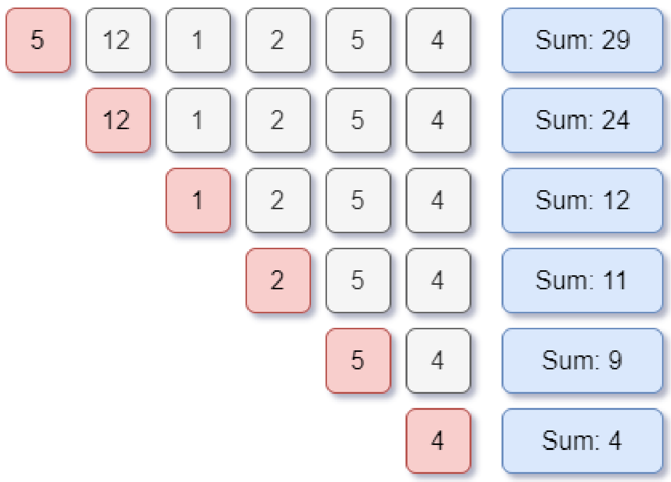
Until no elements are left in the subarray, the first element is discarded one by one and new subarrays are created.
When there is no element to be discarded, the same operations for the new subarray which are created by removing the last element will be continued until there is no element to be removed.

Let’s assume that we have such an array:

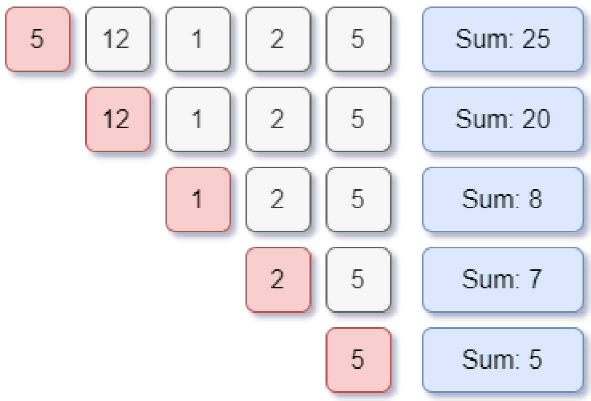


And the target value is 12.

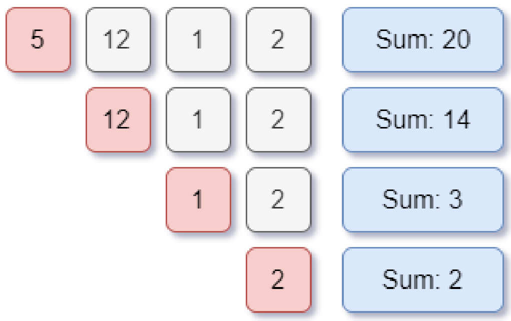
Step1: Create subarrays by discarding the first element in every step.

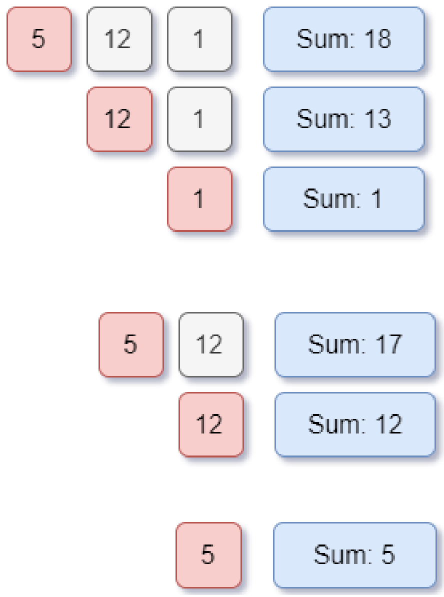


Step2: There is no element to discard, so remove the last element and do the same operation for the new array.



Step3: Do the same operation until there is no element to remove.





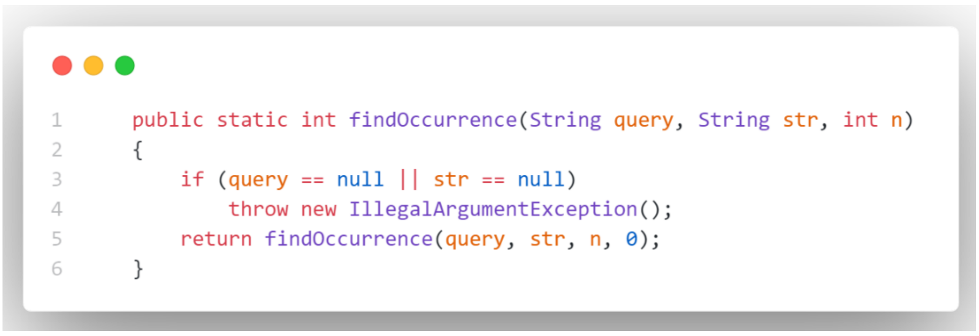
Finally, we have all the contiguous subarrays.
Those whose elements are equal to 12 are as follows:
[1, 2, 5, 4]
[12]

- Problem4: It is already written in the homework, it will analys later.

4. Code Analysis

- Problem1

The user calls wrapper `findOccurence()` method with parameters `query` (string we are searching its occurrences), `str` (string in which we search the query.) and `n` (number of occurrence).

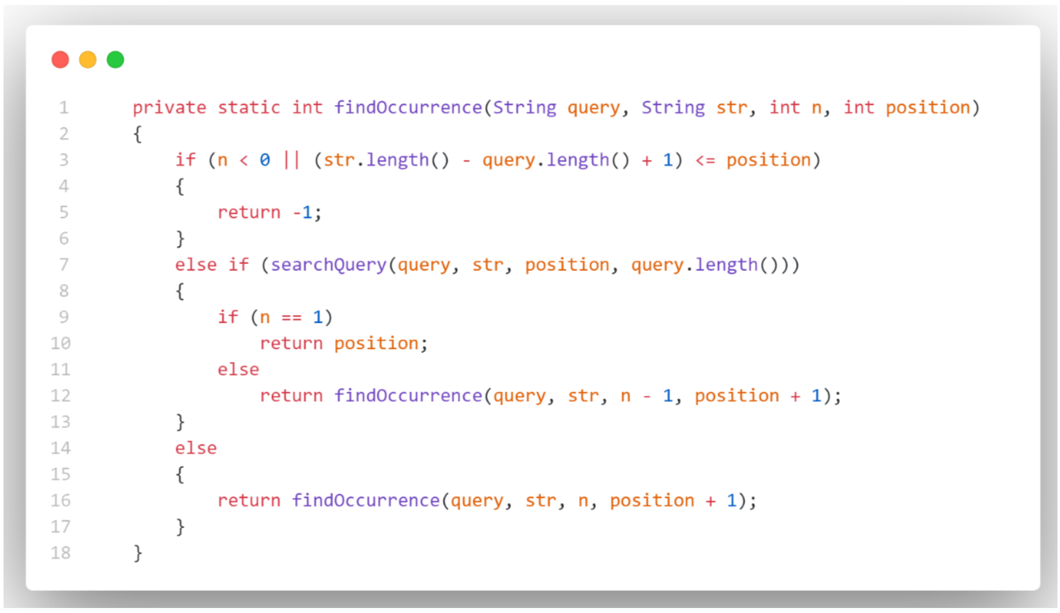


```java
public static int findOccurrence(String query, String str, int n)
{
    if (query == null || str == null)
        throw new IllegalArgumentException();
    return findOccurrence(query, str, n, 0);
}
```

In this method we call the recursive `findOccurence()` method with additional `position` parameter.

The time complexity [T(n, m)] of this method will be result of recursive `findOccurence()` method [T_R(n, m)].

Time complexity of recursive `findOccurence()` [T_R(n, m)]:



```
1       private static int findOccurrence(String query, String str, int n, int position)
2       {
3           if (n < 0 || (str.length() - query.length() + 1) <= position)
4           {
5               return -1;
6           }
7           else if (searchQuery(query, str, position, query.length()))
8           {
9               if (n == 1)
10                  return position;
11              else
12                  return findOccurrence(query, str, n - 1, position + 1);
13          }
14          else
15          {
16              return findOccurrence(query, str, n, position + 1);
17          }
18      }
```

Function starts searching with 0th position of `str` and it recursively calls itself by increasing position until reach the end, and in every recursive calling, another recursive method `searchQuery()` is being invoked.

Base case:

Line3/5 – Check if `n` is negative or `position` reach the end of the `str`. If so return -1. It is *Θ(1)*

Recursive case1:

Line7/13 – Call recursive `searchQuery()` method. It will check the characters in `str` between current `position` + `length` of query string.
[Assume time complexity of `searchQuerry()`: T_s(m)]

If the occurrence is found, then return the position *-Θ(1)-*

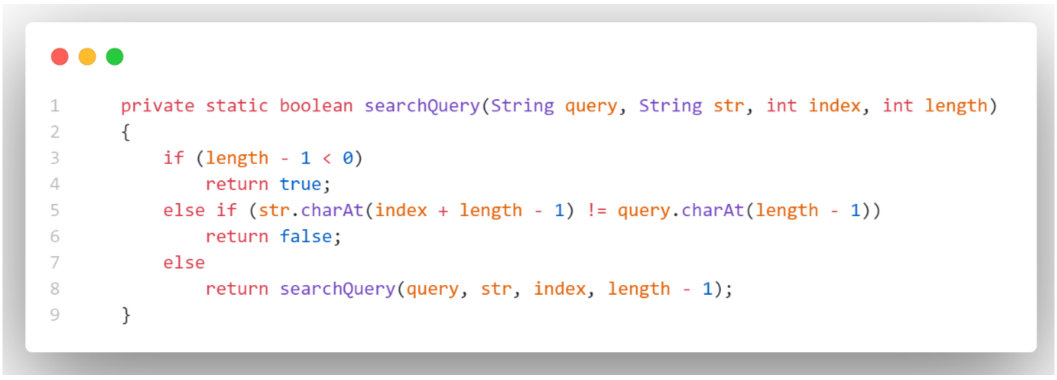
Otherwise,
make an recursive call and indicate (by decreasing n) it is not the occurrence we are searching, and increase the position.

So its time complexity will be: $T_R(n-1)$

Recursive case2:
Line14/17 – If the condition on above not executed then increase the `position` and keep searching the occurrence by recursive calls. So, its time complexity will be $T_R(n-1)$

$T_R(1) = \Theta(1)$
$T_R(n, m) = T_R(n-1) + T_S(m) + \Theta(1)$

<u>Let's find the time complexity of `searchQuerry()`</u> $[T_s(m)]$:


```java
private static boolean searchQuery(String query, String str, int index, int length)
{
    if (length - 1 < 0)
        return true;
    else if (str.charAt(index + length - 1) != query.charAt(length - 1))
        return false;
    else
        return searchQuery(query, str, index, length - 1);
}
```

Start from the last character of the query and compare it with corresponding character in str. [comparision is: -$\Theta(1)$-]
If characters are not equal, return false [returning: -$\Theta(1)$-]
If they are equal recursively call itself until the no character left. [$T_S(m-1)$]

Relations:
$T_S(1) = \Theta(1)$ (base case)
$T_S(m) = T_S(m-1) + \Theta(1)$

Let's solve this and apply it to the relation we found above.

$T_S(m) = T_S(m-1) + 1$
$= T_S(m-2) + 1 + 1$
$= T_S(m-3) + 1 + 1 + 1$

$= T_S(m-4) + 1 + 1 + 1 + 1$

.

.

.

$= T_S(1) + (m-1) \times 1$

Apply $T_S(1) = \Theta(1)$ to the result.

$T_S(m) = \Theta(1) + \Theta(m-1) = \Theta(m)$

Apply $T_S(m) = \Theta(m)$ to the relation on above:

$T_R(1) = \Theta(1)$ (base case)
$T_R(n) = T_R(n-1) + T_S(m) + \Theta(1)$

$T_R(n) = T_R(n-1) + \Theta(m)$
$= T_R(n-2) + \Theta(m) + \Theta(m)$
$= T_R(n-3) + \Theta(m) + \Theta(m) + \Theta(m)$
$= T_R(n-4) + \Theta(m) + \Theta(m) + \Theta(m) + \Theta(m)$

.

.

.

$= T_R(1) + (n-1) \times \Theta(m)$

Apply $T_R(1) = \Theta(1)$ to the result.

$T_R(n, m) = \Theta(1) + \Theta(n-1) \times \Theta(m) = \Theta(n \times m)$ in the worst case.

[n is `length` of str, m is length of `query`]

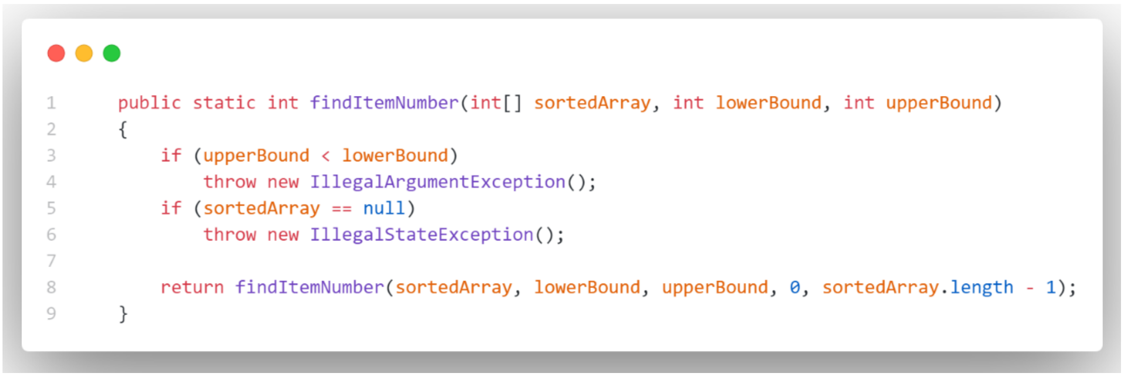
In the best case, 1st occurrence is requested and that is at the beginning of the string. -$\Theta(m)$-

Finally,

$T(n, m) = O(n \times m)$
$T(n, m) = \Omega(m)$

- Problem2

User calls `findItemNumber()` method with parameters `sortedArray`, `lowerBound` and `upperBound`.



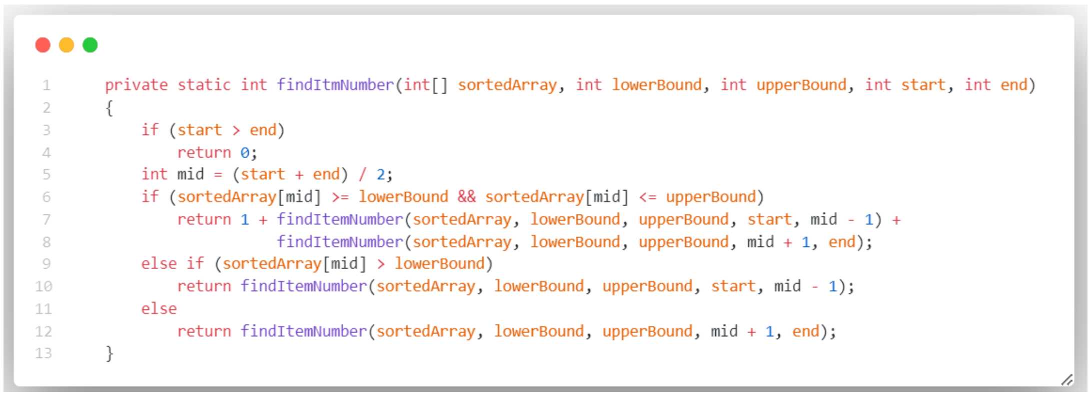
```java
public static int findItemNumber(int[] sortedArray, int lowerBound, int upperBound)
{
    if (upperBound < lowerBound)
        throw new IllegalArgumentException();
    if (sortedArray == null)
        throw new IllegalStateException();

    return findItemNumber(sortedArray, lowerBound, upperBound, 0, sortedArray.length - 1);
}
```

In this method we call the recursive `findItemNumber()` method with additional `start`(0) and `end`(length - 1) indexes as parameter.

The time complexity [T(n)] of this method will be result of recursive `findItemNumber()` method [T_R(n)].

Time complexity of recursive `findItemNumber()` [T_R(n)]:



```java
private static int findItmNumber(int[] sortedArray, int lowerBound, int upperBound, int start, int end)
{
    if (start > end)
        return 0;
    int mid = (start + end) / 2;
    if (sortedArray[mid] >= lowerBound && sortedArray[mid] <= upperBound)
        return 1 + findItemNumber(sortedArray, lowerBound, upperBound, start, mid - 1) +
                   findItemNumber(sortedArray, lowerBound, upperBound, mid + 1, end);
    else if (sortedArray[mid] > lowerBound)
        return findItemNumber(sortedArray, lowerBound, upperBound, start, mid - 1);
    else
        return findItemNumber(sortedArray, lowerBound, upperBound, mid + 1, end);
}
```

Function starts with finding middle index, if number in that index is in the range, then check both side of array.
If this number is greater then the `lowerBound`, you don't need to check left side of array anymore.
If this number smaller then the `upperBound`, you don't need to check right side of array anymore.

Base case:
If there is no element to compare return 0.
$T_R(1) = \Theta(1)$

In the worst case, function will return both left and right side.
So, the relation will be:
$T_W(n) = T_W(n/2) + T_W(n/2) + \Theta(1)$

Let's solve the relation

$T_W(1) = 1$
$T_W(n) = 2 \times T_W(n/2) + 1$

$$\begin{aligned} T_W(n) &= 2 \times T_W(n/2) + 1 \\ &= 4 \times T_W(n/4) + 1 + 1 = 2^2 \times T_W(n/2^2) + 2 \times 1 \\ &= 8 \times T_W(n/8) + 1 + 1 + 1 = 2^3 \times T_W(n/2^3) + 3 \times 1 \\ &= \underline{n} \times T_W(n/n) + 1 + 1 + .. + 1 = \underline{2^k} \times T_W(1) + k \times 1 \\ \\ &= 2^k \times T_R(1) + k \times 1 \end{aligned}$$

Apply the base case to the result.

$T_W(n) = 2^k \times 1 + k \times 1$

We found that, $n = 2^k$

Let's take the logarithm of both sides.

$\log_2 n = \log_2 2^k = k$

Apply result of k to the formula:

$T_W(n) = 2^k \times 1 + k \times 1$
$T_W(n) = 2^{\log(n)} + \log(n) = n + \log(n)$

$T_W(n) = \Theta(n)$

The worst case of recursive `findItemNumber()` is *Θ(n).*

Let’s look at the best case now.
If the range is greater or less than the numbers in the array, the method will recursively be called in one direction only (left or right side of array).

In this case, relation will be such as:
$T_B(n) = T_B(n/2) + \Theta(1)$
$T_B(1) = \Theta(0)$

<u>Let’s solve the relation</u>

$T_B(1) = 0$
$T_B(n) = T_B(n/2) + 1$

$T_B(n) = T_B(n/2) + 1$
$= T_B(n/4) + 1 + 1 = T_B(n/2^2) + 2 \times 1$
$= T_B(n/8) + 1 + 1 + 1 = T_B(n/2^3) + 3 \times 1$
$= T_B(n/n) + 1 + 1 + .. + 1 = T_B(n/2^k) + k \times 1$
$= T_B(1) + k \times 1$

So, $n / 2^k = 1$
$n = 2^k$
$\log_2 n = k$

$T_B(n) = T_B(1) + k \times 1$
$= 0 + \log(n)$

$T_B(n) = \Theta(\log(n))$

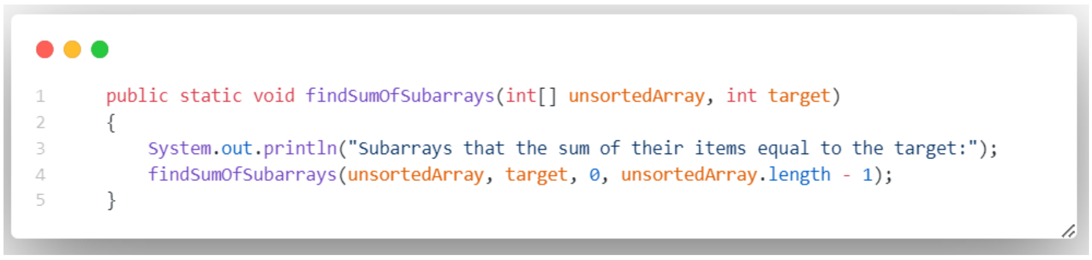
The best case of recursive `findItemNumber()` is *Θ(log(n))*.

Finally,

$T(n) = O(n)$
$T(n) = \Omega(\log(n))$

- <u>Problem3</u>

User calls `findSumOfSubarrays()` method with parameters `unsortedArray` and `target`.

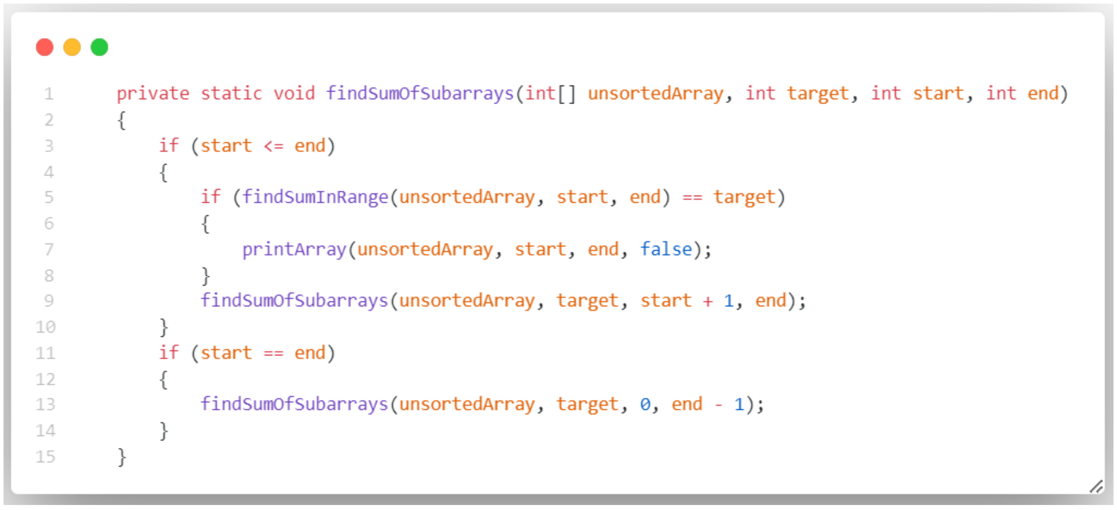


```java
public static void findSumOfSubarrays(int[] unsortedArray, int target)
{
    System.out.println("Subarrays that the sum of their items equal to the target:");
    findSumOfSubarrays(unsortedArray, target, 0, unsortedArray.length - 1);
}
```

In this wrapper method, I call the recursive `findSumOfSubarrays()` method with additional `start` and `end` parameters which are 0th index and last index of the array.

So, the time complexity of the `findSumOfSubarrays()` will be the same as this recursive method.

<u>Time complexity of recursive `findSumOfSubarrays()`</u> [$T_R(n)$]:



```java
1     private static void findSumOfSubarrays(int[] unsortedArray, int target, int start, int end)
2     {
3         if (start <= end)
4         {
5             if (findSumInRange(unsortedArray, start, end) == target)
6             {
7                 printArray(unsortedArray, start, end, false);
8             }
9             findSumOfSubarrays(unsortedArray, target, start + 1, end);
10        }
11        if (start == end)
12        {
13            findSumOfSubarrays(unsortedArray, target, 0, end - 1);
14        }
15    }
```

In every recursive call, first of all, the start position is increases. But as seen in 13th line, when start position become equal to the end position, start position being setted to 0 again and end decrease by one.

So, in a 4 size arrary, number of subarrays will take this shape:

Before the 1st recursive call in 13th line:
4 subarray,

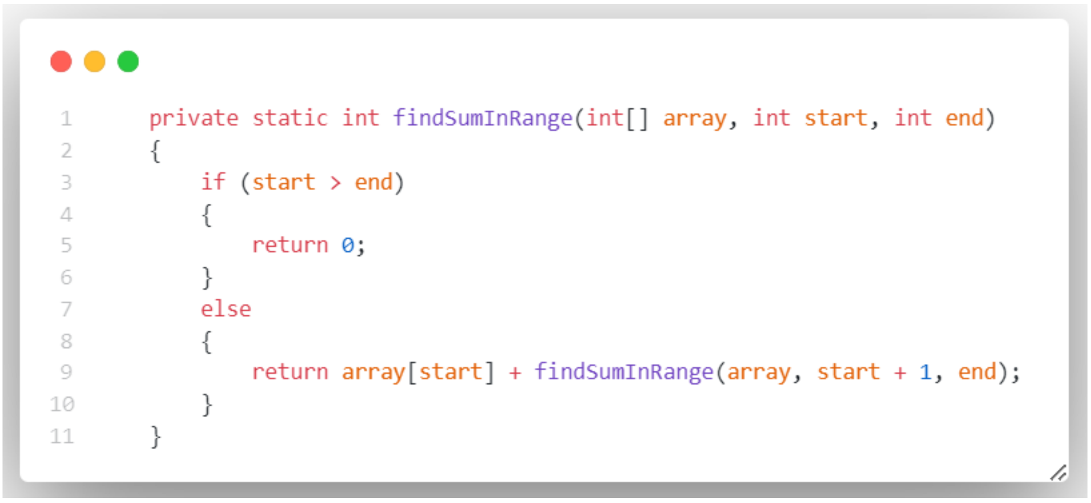
Before the 2nd recursive call in 13th line:
3 subarray,

Before the 3rd recursive call in 13th line:
2 subarray,

Before the 4th recursive call in 13th line:
1 subarray,

number of subarrays will be: n * (n+1) / 2

<u>Let's look at the helper methods before calculate results:</u>

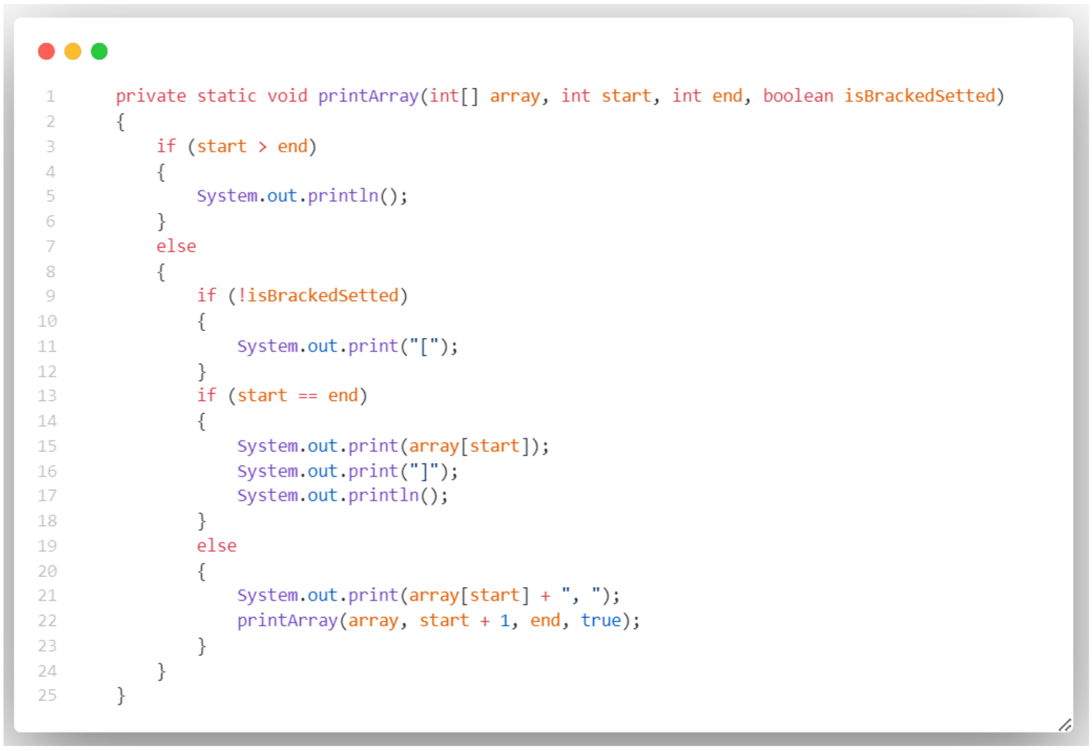


```java
private static int findSumInRange(int[] array, int start, int end)
{
    if (start > end)
    {
        return 0;
    }
    else
    {
        return array[start] + findSumInRange(array, start + 1, end);
    }
}
```

This method gives the sum of the remaining items between start and end position.
m = start – end (number of items that will be calculated)

$T_H(0) = 1$
$T_H(m) = T_H(m - 1) + 1$

We've solved this relation before, it is:
$T_H(m) = \Theta(m)$



```
1      private static void printArray(int[] array, int start, int end, boolean isBrackedSetted)
2      {
3          if (start > end)
4          {
5              System.out.println();
6          }
7          else
8          {
9              if (!isBrackedSetted)
10             {
11                 System.out.print("[");
12             }
13             if (start == end)
14             {
15                 System.out.print(array[start]);
16                 System.out.print("]");
17                 System.out.println();
18             }
19             else
20             {
21                 System.out.print(array[start] + ", ");
22                 printArray(array, start + 1, end, true);
23             }
24         }
25     }
```

This method prints the array in pretty format with the items between start and end position.
m = start – end (number of items that will be calculated)

$T_H(0) = 1$
$T_H(m) = T_H(m - 1) + 1$

We've solved this relation before, it is:
$T_H(m) = \Theta(m)$

So, both helpers methods are $\Theta(m)$ (the m is changes depend on the length of array).

If we go back to our main recursive method, the number of recursive calls were: n * (n+1) / 2, end the helpers method are $\Theta(m)$.

So the $T_R(n) = n * (n+1) / 2 * m$

We can considere m as depend on the length of array. Time complexity T_R will be:

$T_R(n) = \Theta(n^3)$ in the worst case and also best case (Line 5 will be executed in every recursive call).

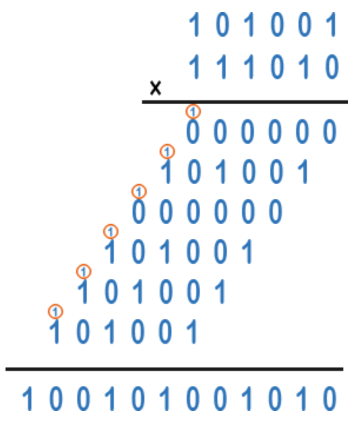
Finally, time complexity T will be:

T(n) = Θ(n^3)

- <u>Problem4</u>

Method `foo()` allows the product of two long binary numbers (which is $\Theta(n^2)$) to be multiplied in a shorter time using the divide and conquere technique.

Let's look at first how it is $\Theta(n^2)$ and how to solve it faster:



the number of digits of the first number: n
the number of digits of the second number: n

As seen in the image above, it is necessary to perform n * n multiplication operations to find the product of the multiplication of two binary numbers.

Let's assume we have 2 binary number as:

x: 101001, digit number: 6
y: 111010, digit number: 6

We can consider x and y as:

x: 101 | 001 $\rightarrow$ 101 (a) * $2^{6/2}$ + 001 (b)
y: 111 | 010 $\rightarrow$ 111 (c) * $2^{6/2}$ + 010 (d)

We can write multiplication like this:

$$\begin{aligned} x \times y &= (a\times 2^{n/2}+b) \times (c\times 2^{n/2}+d) \\ &= (a\times 2^{n/2}) \times (c\times 2^{n/2}) + \underline{ad}\times 2^{n/2} + \underline{bc}\times 2^{n/2} + \underline{bd} \\ &= ac \times 2^{2(n/2)} + (ad + bc) \times 2^{n/2} + bd \end{aligned}$$

So we can call our method for the $\underline{a*c}$, $\underline{ad + bc}$, $\underline{b*d}$.

```
sub0 = foo(a, c)
sub1 = foo(a + b, c + d)
sub2 = foo(b, d)
```

$$\begin{aligned} &(a + b)(c + d) - ac - bd \\ &= (ac + ad + bc + bd) - ac - bd \\ &= (ad + bc) \end{aligned}$$

So, $\underline{ad + bc}$ = `sub1 - sub0 - sub2`

We can apply the sub results to the result of multiplication, which is:

$$x \times y = sub2\ \ 2^{2(n/2)} + (sub1 - sub0 - sub2) \times 2^{n/2} + sub2$$

<u>Time complexity from basic operations:</u>

We have 3 recursive call and in every iteration, the number will be divide into 2 number. When 1 digit remains, multiply them and return -Θ(1)-. Finally apply all multiplication results to the formula and return the result (basic operation) -Θ(n)-.

$T(1) = 1$
$T(n) = 3T(n/2) + \Theta(n)$

$$\begin{aligned} T(n) &= 3T(n/2) + n \\ &= 3^2T(n/2^2) + n + n/2 \\ &= 3^3T(n/2^3) + n + n/2 + n/4 \\ &= 3^k(n/n) + n + n/2 + n/4 + \dots + n/2^{k-1} \\ &= 3^k (n/2^k) + (n + n/2 + n/4 + \dots + n/2^{k-1}) \end{aligned}$$

As last term, $n/2^k = 1$, therefor $n = 2^k$
$\log_2 n = k$

Apply $\log_2 n$ to the result.

$= 3^{\log_2 n} (n/n) + (n + n/2 + \dots + n/2^{\log_2 n - 1})$
$= n^{\log_2 3} + (n + n/2 + n/4 + \dots + 2)$
$= n^{1.6} + ((n + n/2 + n/4 + \dots + 2 + 1) - 1)$
$[\ (n + n/2 + n/4 + \dots + 2 + 1) = 2n - 1\ ]$
$= n^{1.6} + ((2n-1) - 1)$
$= n^{1.6} + 2n-2$

Therefor time complextiy T(n) will be,
$T(n) = \Theta(n^{1.6})$

Thanks to the `foo()` method, time complexity has been reduced from $\Theta(n^2)$ to $\Theta(n^{1.6})$.

5. Inductions

- <u>Induction of Problem2 for the worst case -*Θ(n)*-:</u>

$T(n) = 2T(n/2) + \Theta(1)$
$T(1) = 1$

<u>Base case:</u> $T(1) = 1$, $n = 1$
$1 = 1$ is true.

<u>Induction Hypothesis:</u> Assume for all values of array is in the given range.
So, $T(n) = n$ is true.

<u>Induction Step:</u> We want to show $T(n+1) = n + 1$

$T(n+1) = 2T((n + 1)/2) + 1 = n + 1$ (by induction hypothesis)
$= 2T(n/2 + 1/2) + 1 = n + 1$
$= 2T(n/2) + 2T(1/2) + 1 = n + 1$
$= \underline{2T(n/2) + 1} + 1 = n + 1$
$= \underline{n} + 1 = n + 1$ *(true)*

<u>Conclusion:</u>
Thus we conclude that $T_W(n) = n$

- Induction of Problem2 for the best case -*Θ(log(n))*-:

  $T(n) = T(n/2) + \Theta(1)$
  $T(1) = 0$

  Base case: $T(1) = \log_2(1) = 0$, $n = 1$
  $\log_2(1) = 0$ is true.

  Induction Hypothesis: There is no number in the range in the array. The array will be split in half and only one side will be looked at until the elements are gone.

  So, $T(n) = \log_2(n)$ is true.

  Induction Step: We want to show $T(n*2) = \log_2(n*2)$

  $T(n*2) = T((n*2) / 2) + 1 = \log(n*2)$ (by induction hypothesis)
  $= T(n) + 1 = \log_2(n*2)$
  $= \log_2(n) + 1 = \log_2(n) + \log_2(2)$
  $= \log_2(n) + 1 = \log_2(n) + 1$ *(true)*

  Conclusion:
  Thus we conclude that $T_B(n) = \log(n)$

- Induction of Problem3 for the worst case -*Θ(n*m)*-:

  Assume m is constant (because the length of the query is quite small compared to the large string, then we can ignore it).

  $T(n) = T(n-1) + 1$
  $T(1) = \Theta(1)$

  Base case: $T(1) = 1$, $n = 1$
  $1 = 1$ is true.

  Induction Hypothesis: Assume the requested occurrence is not in the string, or it is at the tail of the string.

  So, $T(n) = n$ is true.

Induction Step: We want to show T(n+1) = n + 1

T(n+1) = T(n) + 1 = n + 1 (by induction hypothesis)
= n + 1 = n + 1 (true)

Conclusion:
Thus we conclude that $T_W(n) = n$

6. Test Cases

- Test cases for 1st problem:

P1- First test:

Test initializes with following parameters:

`query` = “aa”
`str` = “zzaaabbccaadd”
`occurrence` = 2

step1:

`query` = “aa”
`str` = “zzaaabbccaadd”
`occurrence` = 2
`position` = 0

Compare indexes from end to start and check if all indexes are equals.

zzaaabbccaadd
aa (they are not same)

Return false and increase the `position`.

step2:

`query` = “aa”
`str` = “zzaaabbccaadd”
`occurrence` = 2
`position` = 1

zz$\underline{a}$aabbccaadd
a$\underline{a}$ (they are same)

Compare the previous elements.

z$\underline{z}$aaabbccaadd
$\underline{a}$a (they are not same)

Return false and increase the `position`.

step3:

`query` = “aa”
`str` = “zzaaabbccaadd”
`occurrence` = 2
`position` = 2

zza$\underline{a}$abbccaadd
a$\underline{a}$ (they are same)

Compare the previous elements.

zz$\underline{a}$aabbccaadd
$\underline{a}$a (they are same)

Return true, decrease `occurrence` number and increase the `position`.

step4:

`query` = “aa”
`str` = “zzaaabbccaadd”
`occurrence` = 1
`position` = 3

zzaa$\underline{a}$bbccaadd
a$\underline{a}$ (they are same)

Compare the previous elements.

zzaaabbccaadd
aa (they are same)

Return true.

`occurrence` number is 1, so, return the `position` which is 3.

P1- Second test:

Test initializes with following parameters:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 4

step1:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 4
`position` = 0

compare indexes from end to start and check if all indexes are equals.

aaabbc
aa (they are same)

Return false and increase the `position`.

Compare the previous elements.

aaabbc
aa (they are same)

Return true increase `position` and decrease `occurrence` number.

step2:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 3
`position` = 1

aaabbc
aa (they are same)

Compare the previous elements.

aaabbc
aa (they are same)

Return true increase `position` and decrease `occurrence` number.

step3:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 2
`position` = 2

aaabbc
aa (they are not same)

Return false and increase the `position`.

step4:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 2
`position` = 3

aaabbc
aa (they are not same)

Return false and increase the `position`.

step5:

`query` = “aa”
`str` = “aaabbc”
`occurrence` = 2
`position` = 4

aaabb<u>c</u>
a<u>a</u> (they are not same)

Return false and increase the `position`.

So, finally there is no element to. We could not find the requested occurrence, therefor return -1.

- Test cases for 2nd problem

P2- First test:

Test initializes with following parameters:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 0
`upperBound` = 5

Step1:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 0
`upperBound` = 5
`start` = 0
`end` = 6
`counter` = 0 (total returning values)

mid = (0 + 6) / 2 = 3
sortedArray[3] = 9 (greater then the `upperBound`)

No need to look further to the right, only check the left side!

<u>Step2:</u>

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 0
`upperBound` = 5
`start` = 0
`end` = 2
`counter` = 0 (total returning values)

mid = (0 + 2) / 2 = 1
sortedArray[1] = 4 (in the range)

Increase `counter` and check both left and right side!

<u>Step3 (Check the left side):</u>

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 0
`upperBound` = 5
`start` = 0
`end` = 0
`counter` = 1 (total returning values)

mid = (0 + 0) / 2 = 0
sortedArray[0] = 1 (in the range)
Increase `counter`, it is last element, no more check!

<u>Step3 (Check the right side):</u>

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 2
`upperBound` = 2
`start` = 0
`end` = 0
`counter` = 2 (total returning values)

mid = (2 + 2) / 2 = 2
sortedArray[2] = 7 (greater then the `upperBound`)
There is no element to check, return the `counter` which is 2.

P2- Second test:

Test initializes with following parameters:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 17
`upperBound` = 22

Step1:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 17
`upperBound` = 22
`start` = 0
`end` = 6
`counter` = 0 (total returning values)

mid = (0 + 6) / 2 = 3
sortedArray[3] = 9 (smaller then the `lowerBound`)

No need to look further to the left, only check the right side!

Step2:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 17
`upperBound` = 22
`start` = 4
`end` = 6
`counter` = 0 (total returning values)

mid = (4 + 6) / 2 = 5
sortedArray[5] = 15 (smaller then the `lowerBound`)

No need to look further to the left, only check the right side!

Step3:

`sortedArray` = {1, 4, 7, 9, 12, 15, 16}
`lowerBound` = 17

`upperBound` = 22
`start` = 6
`end` = 6
`counter` = 0 (total returning values)

mid = (6 + 6) / 2 = 6
sortedArray[6] = 16 (smaller then the `lowerBound`)

There is no element to check, return the `counter` which is 0.

- Test case of 3rd problem

Test initializes with following parameters:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6

Step1:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 0
`end` = 4

sum = 2 + 3 + 1 + 5 + 6 = 17

It is not equal to the target. Increase `start` position.

Step2:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 1
`end` = 4

sum = 3 + 1 + 5 + 6 = 15

It is not equal to the target. Increase `start` position.

<u>Step3:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 2
`end` = 4

sum = 1 + 5 + 6 = 13

It is not equal to the target. Increase `start` position.

<u>Step4:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 3
`end` = 4

sum = 5 + 6 = 11

It is not equal to the target. Increase `start` position.

<u>Step5:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 4
`end` = 4

sum = 6

It is equal to the target. *Print it to the screen*!
We are end of the list, so decrease the `end` and set `start` to the 0.

<u>Step6:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 0
`end` = 3

sum = 2 + 3 + 1 + 5 = 11

It is not equal to the target. Increase `start` position.

<u>Step7:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 1
`end` = 3

sum = 3 + 1 + 5 = 9

It is not equal to the target. Increase `start` position.

<u>Step8:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 2
`end` = 3

sum = 1 + 5 = 6

It is equal to the target. *Print it to the screen*!
Increase the `start` position.

<u>Step9:</u>

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 3
`end` = 3

sum = 5

It is not equal to the target.

We are end of the list, so decrease the `end` and set `start` to the 0.

Step10:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 0
`end` = 2

sum = 2 + 3 + 1 = 6

It is equal to the target. *Print it to the screen*! Increase the `start`.

Step11:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 1
`end` = 2

sum = 3 + 1 = 4

It is not equal to the target. Increase the `start` position.

Step12:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 2
`end` = 2

sum = 1

It is not equal to the target.
We are end of the list, so decrease the `end` and set `start` to the 0.

Step13:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 0
`end` = 1

sum = 2 + 3 = 5

It is not equal to the target. Increase the `start` position.

Step14:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 1
`end` = 1

sum = 3

It is not equal to the target.
We are end of the list, so decrease the `end` and set `start` to the 0.

Step15:

`unsortedArray` = {2, 3, 1, 5, 6}
`target` = 6
`start` = 0
`end` = 0

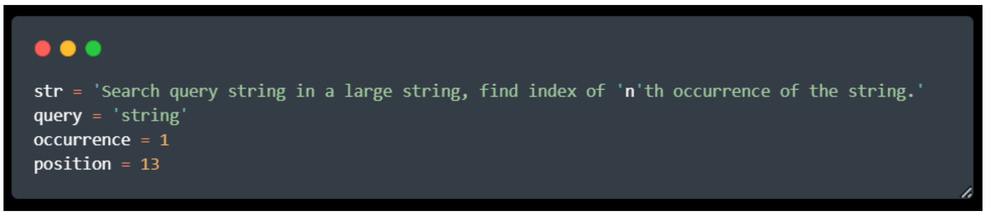
sum = 2

It is not equal to the target.

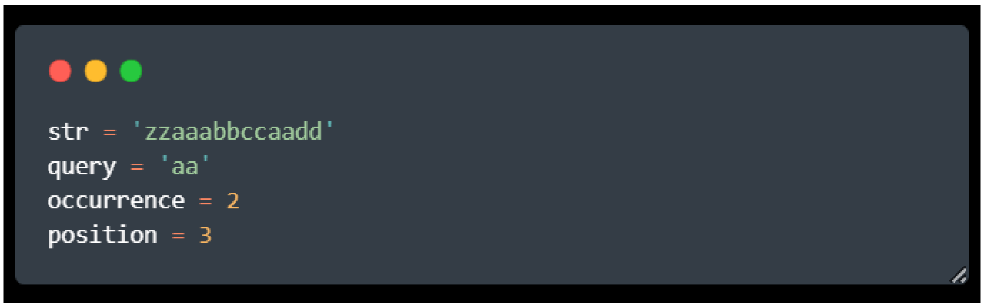
We checked all the subarrays. We have printed those whose results are equal to target, and now we can return from the function.

7. Running Command and Results on Terminal

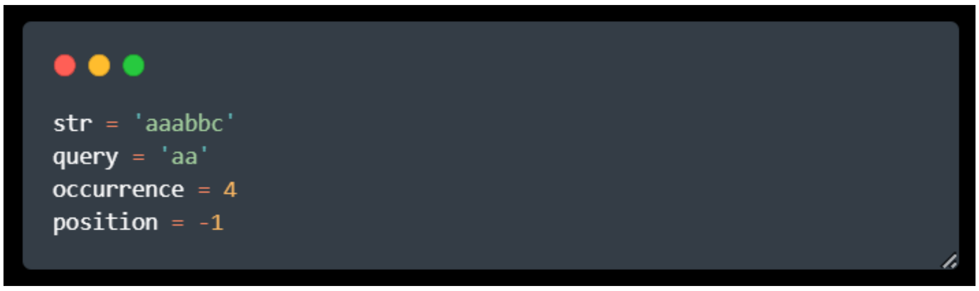
- Running command result for 1st problem



```
str = 'Search query string in a large string, find index of 'n'th occurrence of the string.'
query = 'string'
occurrence = 1
position = 13
```

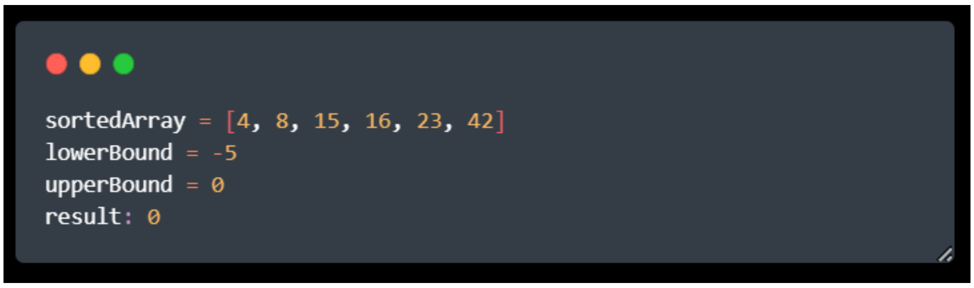


```
str = 'zzaaabbccaadd'
query = 'aa'
occurrence = 2
position = 3
```

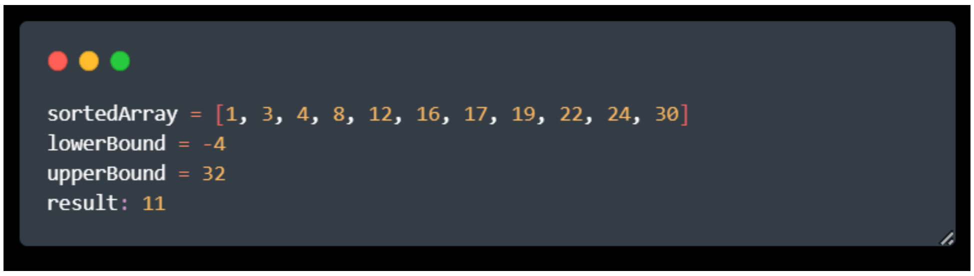


```
str = 'aaabbc'
query = 'aa'
occurrence = 4
position = -1
```

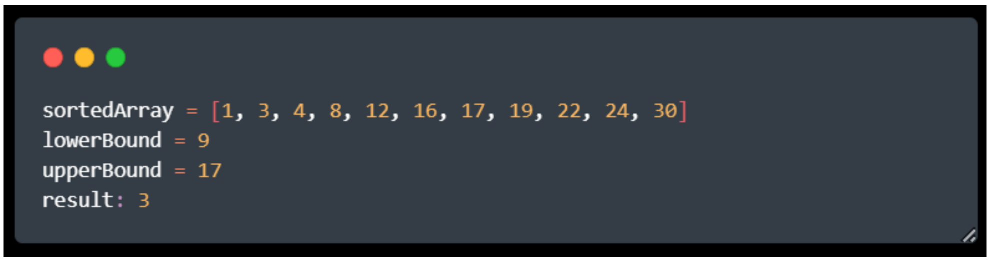
- Running command result for 2nd problem



```
sortedArray = [4, 8, 15, 16, 23, 42]
lowerBound = -5
upperBound = 0
result: 0
```

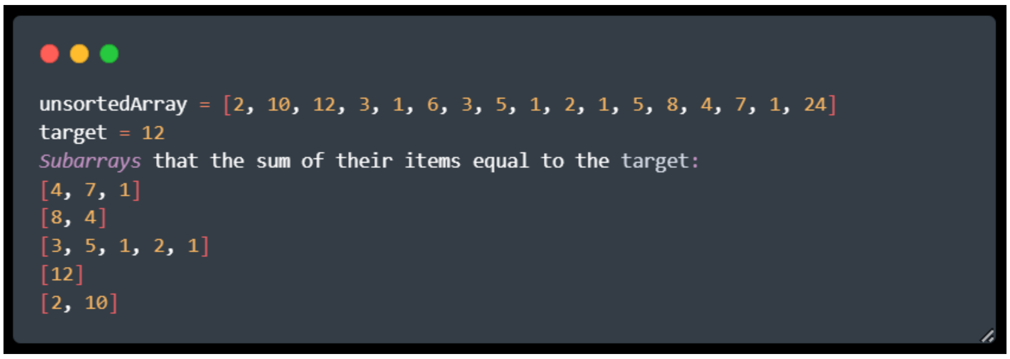


```
sortedArray = [1, 3, 4, 8, 12, 16, 17, 19, 22, 24, 30]
lowerBound = -4
upperBound = 32
result: 11
```

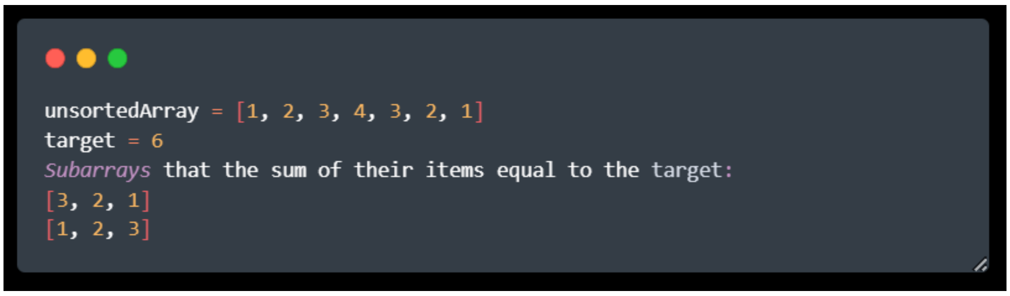


```
sortedArray = [1, 3, 4, 8, 12, 16, 17, 19, 22, 24, 30]
lowerBound = 9
upperBound = 17
result: 3
```

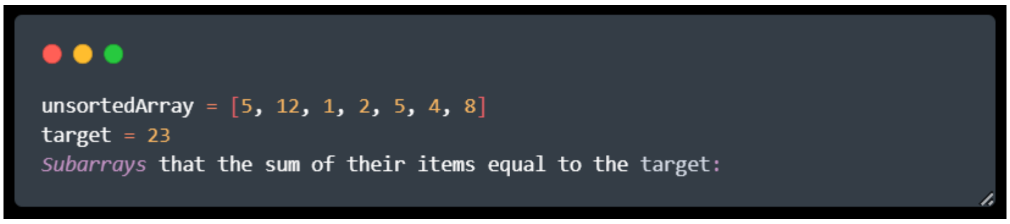
- Running command result for 3rd problem



```
unsortedArray = [2, 10, 12, 3, 1, 6, 3, 5, 1, 2, 1, 5, 8, 4, 7, 1, 24]
target = 12
Subarrays that the sum of their items equal to the target:
[4, 7, 1]
[8, 4]
[3, 5, 1, 2, 1]
[12]
[2, 10]
```

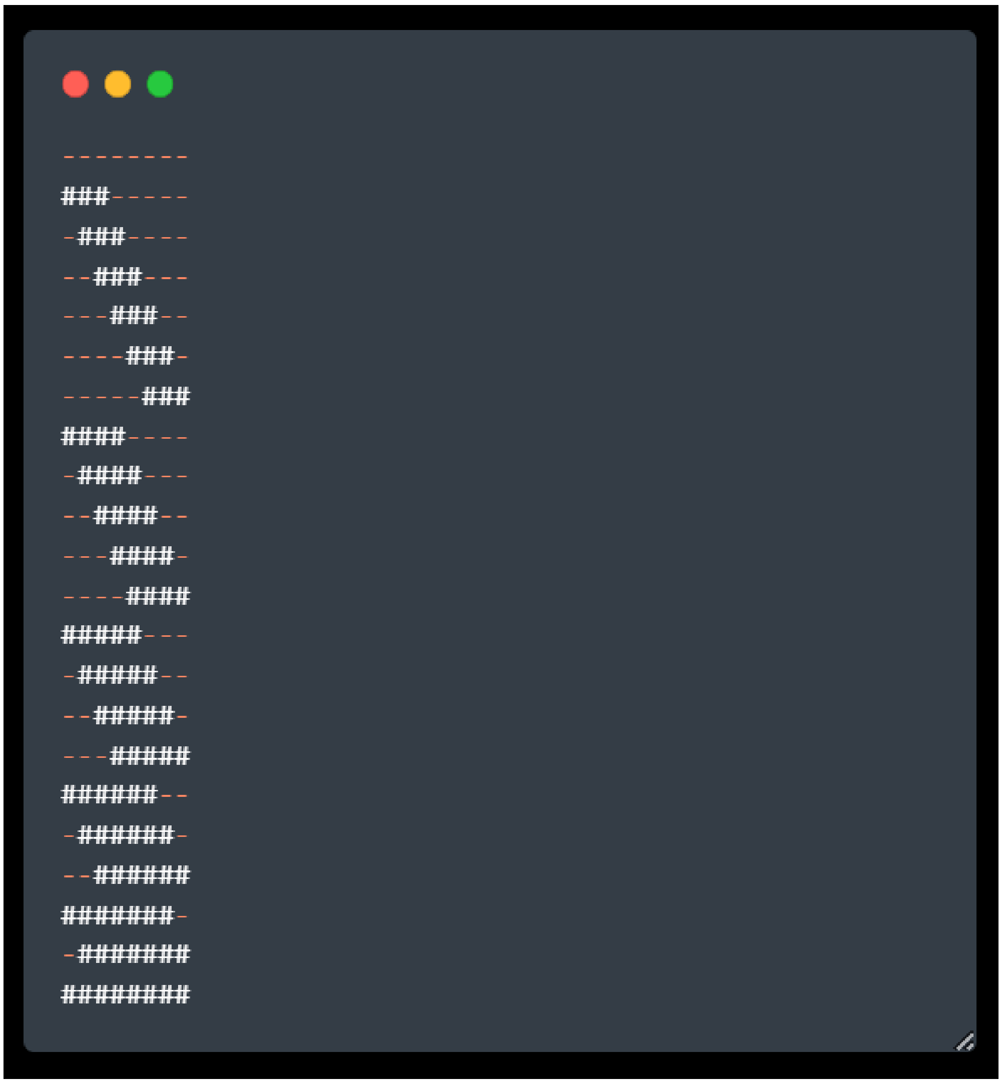


```
unsortedArray = [1, 2, 3, 4, 3, 2, 1]
target = 6
Subarrays that the sum of their items equal to the target:
[3, 2, 1]
[1, 2, 3]
```



```
unsortedArray = [5, 12, 1, 2, 5, 4, 8]
target = 23
Subarrays that the sum of their items equal to the target:
```

- Running command result for 5th problem (missing)



```
--------
###-----
-###----
--###---
---###--
----###-
-----###
####----
-####---
--####--
---####-
----####
#####---
-#####--
--#####-
---#####
######--
-######-
--######
#######-
-#######
########
```