

**GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 4 Report**

**Student Name: Enes Abdülhalik
Student Number: 1901042803**

Question 1:

1- System Requirements:

- a. The system can find the index of a specified occurrence of a string in a larger string.
- b. The system needs the search target string, the container string, and the occurrence of the target string as input to function as expected.
- c. Returns the index as an integer if the target was found, or -1 if it wasn't.
- d. If occurrences less than 1, or the target or the container are given as null references, the system will throw `IllegalArgumentException`.

2- Problem-Solution Approach:

For this functionality, a recursive solution is chosen. The system has two base case:

- a- The given occurrence of the target is 1.

In the base case, only one occurrence is needed. So, the system will search for the target and return the index if found, and -1 if not.

In the meantime, the system searches for the target, and for each occurrence found, a recursive call is done with the number of occurrences reduced by 1, and searches the rest of the container after the index of the last occurrence found for more occurrences until reaching the target occurrence.

3- Test Cases:

Container: "A man was running in the corridor, when he slipped and fell while he was running in the corridor, because he was running in the corridor."

Cases:

- a. Target: "running in the corridor", occurrence: 2, expected output: 73.
- b. Target: "running in the corridor", occurrence: 4, expected output: -1.
- c. Target: "he", occurrence: 1, expected output: 22.
- d. Target: "woman", occurrence: 2, expected output: -1.

4- Running and Results:

Input is given to the function directly. The driver code is designed to be launched by a makefile in a Linux system.

- a. Target : running in the corridor
Occurrence: 2
Output: 73
- b. Target : running in the corridor
Occurrence: 4
Output: -1
- c. Target : he
Occurrence: 1
Output: 22
- d. Target : woman
Occurrence: 2
Output: -1

5- Time Complexity:

n = length of the container, m = length of the target string, c = occurrences

```

public static int searchStr(String target, String container, int occurrence){      T(n, m)
    if (occurrence < 1 || target == null || container == null)   $\Theta(1)$ 
        throw new IllegalArgumentException();
    return searchStr_help(target, container, 0, occurrence); C(n, m, c)
}

    expression: T(n, m, c) = C(n, m, c) , c > 1 (worst case)
                =  $\Theta(1)$  , c = 0 or n = 0 or m = 0 (best case)

int searchStr_help(String target, String container, int index, int occurrence)      C(n, m, c)
    if(occurrence == 1)       $\Theta(1)$ 
        return container.indexOf(target, index);      O(n*m)
    else{
        index = container.indexOf(target, index);      O(n*m)
        if(index < 0)      //the target does not exist in the container anymore
            return index;       $\Theta(1)$ 

        //An occurrence of the target was found, find the next occurrence
        else      C(n, m, c - 1)
            return searchStr_help(target, container, index + target.length(), occurrence - 1);
    }
}

```

Recurrence relation: $C(n, m, c) = O(n*m) + C(n, m, c - 1) = O(n*m*c)$, $c > 1$ (worst case)
 $= O(n*m)$, $c = 1$ (base case)

6- Proving the Solution:

By induction:

- Basis: $C(n, m, c) = O(n*m)$, $c = 1$
- Solution: $C(n, m, c) = O(n*m) + C(n-m, m, c-1)$
 $= O(n*m) + O(n*m) + C(n, m, c-2)$
 $= O(n*m) + O(n*m) + O(n*m) + C(n, m, c-3)$
.....
 $= O(n*m) + O(n*m) + O(n*m) + \dots + O(n*m)$
 $C(n, m, c) = O(n*m*c)$

Since: $C(n, m, c+1) = O(n*m) + C(n, m, c) = O(n*m*(c+1))$

$$O(n*m) + O(n*m*c) = O(n*m*c)$$

$$O(n*m*c) = O(n*m*c)$$

Then: $C(n, m, c)$ implies $C(n, m, c+1)$

- General expression:

$$T(n, m, c) = C(n, m, c) = O(n*m*c), \quad c > 1 \quad (\text{worst case})$$

$$= \Theta(1), \quad c = 0 \text{ or } n = 0 \text{ or } m = 0 \quad (\text{best case})$$

Question 2:

1- System Requirements:

- The system can find the number of integers inside the array that are within the given range of numbers
- The range is specified by entering the minimum and the maximum values of the range.
- If found, the maximum and minimum values are not included in the range.
- The system needs the array, the minimum and the maximum values as input to function as expected.
- Throws `IllegalArgumentException` when the minimum value is larger than the maximum value of the range.

2- Problem-Solution Approach:

The system searches for the indexes that define the given range on the array. So, the function `countInRange` calls the functions `getMaxIndex` for the end index of the range, and `getMinIndex` for the start index of the range in the array.

getMaxIndex and getMinIndex functions are used to get the indexes of the start and the end of the given range in the array respectively. The indexes returned include the elements that are inside the range. So, the result is basically end - start + 1.

For getMaxIndex and getMinIndex functions, a recursive solution is chosen. As base cases, the algorithm has two conditions:

- a- The given target was found in the array, return the index after it in the direction to the middle of the range.
- b- The search reached a point where the target was not found and it's less than the only item remaining in the search area. For getMaxIndex, return the index of the remaining element - 1. And for getMinIndex, return the index of the remaining element.

In the meantime, The system uses binary search techniques without recopying the array in order to make the search area smaller, which is done by storing the start and the end indexes of the search area.

3- Test Cases:

An array of length 10, stores even numbers from 0 to 18

Cases:

- a. Minimum: 2, maximum: 18, expected output: 7.
- b. Minimum: 2, maximum: 17, expected output: 7.
- c. Minimum: 3, maximum: 18, expected output: 7.
- d. Minimum: -4, maximum: 38, expected output: 10.
- e. Minimum: 5, maximum: 5, expected output: 0.
- f. Minimum: 10, maximum: 5, expected output: throws IllegalArgumentException.
- g. Minimum: 22, maximum: 34, expected output: 0.
- h. Minimum: -15, maximum: -2, expected output: 0.

4- Running and Results:

Input is given to the function directly. The driver code is designed to be launched by a makefile in a Linux system.

- a.

```
Minimum : 2
Maximum : 18
Output : 7
```
- b.

```
Minimum : 2
Maximum : 17
Output : 7
```

- c.

```
Minimum : 3
Maximum : 18
Output : 7
```
- d.

```
Minimum : -4
Maximum : 38
Output : 10
```
- e.

```
Minimum : 5
Maximum : 5
Output : 0
```
- f.

```
minimum value is larger than maximum value.
```
- g.

```
Minimum : 22
Maximum : 34
Output : 0
```
- h.

```
Minimum : -15
Maximum : -2
Output : 0
```

5- Time Complexity: n = size of the array

```
int countInRange(int[] array, int minValue, int maxValue){    T(n)
    if(maxValue < minValue)
        throw new IllegalArgumentException();

    int start = getMinIndex(array, minValue, 0, array.length - 1),    C(n)
        end = getMaxIndex(array, maxValue, 0, array.length - 1);    C(n)

    return end - start + 1;    O(1)
}
```

Expression: $T(n) = O(1) + C(n)$

```
int getMaxIndex(int[] array, int target, int start, int end){    C(n)
    if(array[(end + start)/2] == target)    O(1)
        return (end + start)/2 - 1;
    else if(end == start){    O(1)
        if(array[end] < target)
            return end;
        else
            return end - 1;
    }
    else if(array[(end + start)/2] > target)
        return getMaxIndex(array, target, start, (end + start)/2);    C(n/2)
```

```

else
    return getMaxIndex(array, target, (end + start + 1)/2, end); C(n/2)
}

```

Recurrence relation: $C(n) = \Theta(1) + C(n/2) = O(\log n)$, $n > 1$ (worst case)
 $= \Theta(1)$, $n = 1$ (base case)

```

int getMinIndex(int[] array, int target, int start, int end){      C(n)
    if(array[(end + start)/2] == target)       $\Theta(1)$ 
        return (end + start)/2 + 1;
    else if(end == start){       $\Theta(1)$ 
        if(array[start] <= target)
            return start + 1;
        else
            return start;
    }
    else if(array[(end + start)/2] > target)
        return getMinIndex(array, target, start, (end + start)/2);      C(n/2)
    else
        return getMinIndex(array, target, (end + start + 1)/2, end);      C(n/2)
}

```

Recurrence relation: $C(n) = \Theta(1) + C(n/2) = O(\log n)$, $n > 1$ (worst case)
 $= \Theta(1)$, $n = 1$ (base case)

6- Proving the Solution:

By induction:

- Basis: $C(n) = \Theta(1)$, $n = 1$
- Solution: $C(n) = \Theta(1) + C(n/2)$
 $= \Theta(1) + \Theta(1) + C(n/4)$
 $= \Theta(1) + \Theta(1) + \Theta(1) + C(n/4)$
.....
 $= \Theta(1) + \Theta(1) + \Theta(1) + \dots + \Theta(1)$
 $C(n) = O(\log n)$
Since: $C(2*n) = \Theta(1) + C(n) = O(\log(2*n))$
 $\Theta(1) + O(\log n) = O(\log n)$
 $O(\log n) = O(\log n)$
Then: $C(n)$ implies $C(2*n)$
- General expression:
 $T(n) = C(n) = O(\log n)$, $c > 1$ (worst case)
 $= \Theta(1)$, $c = 0$ or $n = 0$ or $m = 0$ (best case)

Question 3:

1- System Requirements:

- a. The system can find sub arrays of contiguous numbers that are in total equal to the given target sum regardless of their sign.
- b. The system needs the target sum, and the array containing the numbers as input to function as expected.
- c. Returns an ArrayList containing integer arrays. Each array stores two values, the first is the start index of a collection of contiguous numbers in the array, the second is the end index.
- d. For the returned indexes, the numbers at the start indexes are included inside the resulting sub array, but the numbers at the end indexes are not included.
- e. If the array was given as a null reference, the system will throw IllegalArgumentException.

2- Problem-Solution Approach:

The system uses a combination of three functions, `getContiguousSumList` is the wrapper function, and `getContiguousList` with `getContiguousSubArray` are helper functions.

For `getContiguousList` and `getContiguousSubArray`, recursive solutions were chosen. For `getContiguousList` function, the base case is:

- a- The last element in the array was reached, return an empty ArrayList.

And for the function `getContiguousSubArray`, the base cases are:

- a- The end of the array is reached, return -1.
- b- A number that matches the target sum is found, return its index.

The function `getContiguousList` collects the indexes of the number sequences that match the sum target by calling `getContiguousSubArray` for each number, and stores them in an ArrayList to be returned, starting from the head to the end of the array. Recursively, every call adds the ArrayList returned by the inner call to its result to be returned.

The function `getContiguousSubArray`, given a starting index, adds the numbers after the index one by one until the end of the array is reached, or the target sum was matched. Only when the target sum is matched, the function returns the index of the last number in the sequence, and returns -1 in the other cases.

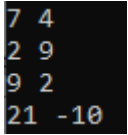
3- Test Cases:

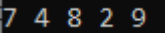
array = {3, 15, 2, 7, 4, 8, 2, 9, 3, 9, 2, 18, 21, -10, 6, 0, 4, 12, 5}

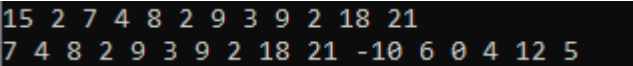
- Cases: a. Target sum: 1.
b. Target sum: 30.
c. Target sum: 100.

4- Running and Results:

Input is given to the function directly. The driver code is designed to be launched by a makefile in a Linux system. Each line represents a sub array.

a. 

b. 

c. 

5- Time Complexity:

n = length of the array

```
ArrayList<int[]> getContiguousSumList(int[] array, int sum){    T(n)
    if(array == null)                O(1)
        throw new IllegalArgumentException();
    return getContiguousList(array, sum, 0);    C(n)
}
```

Expression: $T(n) = O(1) + C(n)$

```
ArrayList<int[]> getContiguousList(int[] array, int sum, int index){    C(n)
    if(index == array.length)    O(1)
        return new ArrayList<int[]>();
```

```
    int index2 = getContiguousSubArray(array, sum, index);    D(n)
    ArrayList<int[]> result = new ArrayList<int[]>();    O(1)
    if(index2 != -1)
        result.add(new int[] {index, index2});    O(1)
```

```
    result.addAll(getContiguousList(array, sum, index + 1));    C(n - 1)
    return result;    O(1)
```

```
}    Recurrence relation:  $C(n) = O(1) + D(n) + C(n - 1) = O(n) + D(n^2)$  ,  $n > 0$  (worst case)
                        =  $O(1)$  ,  $n = 0$  (base case)
```

```

int getContiguousSubArray(int[] array, int sum, int index){      D(n)
    if(index == array.length)  Θ(1)
        return -1;
    else if(sum == array[index])  Θ(1)
        return index;
    else
        return getContiguousSubArray(array, sum - array[index], index + 1);  D(n - 1)
}

```

Recurrence relation: $D(n) = \Theta(1) + D(n - 1) = O(n)$, $n > 0$ (worst case)
 $= \Theta(1)$, $n = 0$ (base case)

6- Proving the Solution:

By induction:

D(n):

- Basis: $D(n) = \Theta(1)$, $n = 0$
- Solution: $D(n) = \Theta(1) + D(n - 1)$
 $= \Theta(1) + \Theta(1) + D(n - 2)$
.....
 $= \Theta(1) + \Theta(1) + \Theta(1) + \dots + \Theta(1)$
 $D(n) = O(n)$
Since: $D(n + 1) = \Theta(1) + D(n) = O(n + 1)$
 $\Theta(1) + O(n) = O(n)$
 $O(n) = O(n)$
Then: $D(n)$ implies $D(n + 1)$
- Expression: $D(n) = O(n)$, $n > 0$ (worst case)
 $= \Theta(1)$, $n = 0$ (best case)

C(n):

- Basis: $C(n) = \Theta(1)$, $n = 0$
- Solution: $C(n) = \Theta(1) + D(n) + C(n - 1)$
 $= O(n) + C(n - 1)$
 $= O(n) + O(n) + C(n - 2)$
.....
 $= O(n) + O(n) + O(n) + \dots + O(n)$
 $C(n) = O(n^2)$
Since: $C(n + 1) = O(n) + C(n) = O((n + 1)^2)$
 $O(n) + O(n^2) = O(n^2 + 2n + 1)$
 $O(n^2) = O(n^2)$
Then: $C(n)$ implies $C(n + 1)$
- Expression: $C(n) = O(n^2)$, $n > 0$ (worst case)
 $= \Theta(1)$, $n = 0$ (base case)

Finally: $T(n) = C(n) = O(n^2)$

Question 4:

1- System Requirements:

- a. The system returns the result of an integer multiplication of two integers regardless of their sign.
- b. The system needs the two given integers as input to function properly.

2- Problem-Solution Approach:

The system uses a recursive solution by processing the integers as halves of each integer's digits, and the middle point is determined by the number with the maximum number of digits. The base case is:

- a- When the given integers are smaller than 10, return their multiplication result.

In the meantime, when the base is not met, the function does three separate recursive calls, first for the first half of the digits of each integer, second for the sum of the two halves of each integer, and third for the second half. The results of each call are then processed in the following formula:

$$(\text{call3} * 10^{(\text{maxNumOfDigits})}) + ((\text{call2} - \text{call3} - \text{call1}) * 10^{(\text{maxNumOfDigits}/2)}) + (\text{call1})$$

3- Test Cases:

- a. Integer 1: 16, integer 2: 27.
- b. Integer 1: 118, integer 2: -31.
- c. Integer 1: 3, integer 2: 8.

4- Running and Results:

Input is given to the function directly. The driver code is designed to be launched by a makefile in a Linux system. Each output represents the returned result of the multiplication.

- a. 432
- b. -3658
- c. 24

5- Time Complexity:

n = Largest number of digits in the integers

```
public static int foo (int integer1, int integer2){          T(n)
    if (integer1 < 10 || integer2 < 10)                    Θ(1)
        return integer1 * integer2;

    //number_of_digit returns the number of digits in an integer
    int digits1 = (int) (Math.log10(integer1) + 1),        Θ(1)
        digits2 = (int) (Math.log10(integer2) + 1),        Θ(1)
        half;

    if(digits1 > digits2)                                    Θ(1)
        half = digits1/2;
    else                                                     Θ(1)
        half = digits2/2;

    // split_integer splits the integer into returns two integers
    // from the digit at position half. i.e.,
    // first integer = integer / 2^half
    // second integer = integer % 2^half
    int int1 = integer1 / (int) Math.pow(10, half),        Θ(1)
        int2 = integer1 % (int) Math.pow(10, half),        Θ(1)
        int3 = integer2 / (int) Math.pow(10, half),        Θ(1)
        int4 = integer2 % (int) Math.pow(10, half),        Θ(1)
        sub0 = foo (int2, int4),                            T(n/2)
        sub1 = foo ((int1 + int2), (int3 + int4)),          T(n/2)
        sub2 = foo (int1, int3);                            T(n/2)

    return (sub2*(int) Math.pow(10, 2*half))+((sub1-sub2-sub0)*(int) Math.pow(10, half))+(sub0);
}
```

Recurrence relation: $T(n) = \Theta(1) + 3 \cdot T(n/2) = O(3^{\log n})$, $n > 1$ (worst case)
 $= \Theta(1)$, $n = 1$ (base case)

6- Proving the Solution:

By induction:

- Basis: $T(n) = \Theta(1)$, $n = 1$
- Solution: $T(n) = \Theta(1) + 3 \cdot T(n/2)$
 $= \Theta(1) + 3 \cdot (\Theta(1) + T(n/4))$
.....
 $= \Theta(1) + 3 \cdot (\Theta(1) + 3 \cdot (\Theta(1) + 3 \cdot (\dots + 3 \cdot \Theta(1)))) \dots$
 $T(n) = O(3^{\log n})$

Since: $T(2n) = \Theta(1) + T(n) = O(3^{\log(2n)})$

$\Theta(1) + O(3^{\log n}) = O(3^{\log n})$

$O(3^{\log n}) = O(3^{\log n})$

Then: $T(n)$ implies $T(2n)$

- Expression: $T(n) = O(3^{\log n})$, $n > 1$ (worst case)
 $= \Theta(1)$, $n = 1$ (best case)

Question 5:

1- System Requirements:

- a. The system calculates the number of combination of blocks of length 3 to L an array of length L can contain.
- b. At least, one cell must be empty between two consecutive blocks.
- c. The array may contain multiple blocks of the same length.
- d. The system has a switch as a Boolean input value that allows printing each combination found to the terminal.

2- Problem-Solution Approach:

The system uses a combination of three functions, countConfigs is the wrapper function, and countArrayConfigs with countBlockConfigs are helper functions. The system generally uses a mathematical approach to find the number of configurations.

countArrayConfigs function calls countBlockConfigs, then recalls itself with a block bigger by one cell and returns the sum of the returned values of the two calls. The function keeps recalling itself until reaching a block that is equal to the array in length.

countBlockConfigs function counts the number of configurations that can be made with a block of a certain length, then recalls the function with a smaller array length to see if extra blocks can be placed in the array while the current block takes some space.

For the base cases, countArrayConfigs has the following:

- a- The length of the current block is equal to the length of the array, only one configuration is possible, return 1.
- b- The length of the current block is bigger than the length of the array, return 0.

And countBlockConfigs has:

- a- The length of the block is equal to the available length in the array, only one configuration is possible, return 1.
- b- The length of the block is bigger than the available length in the array, return 0.

3- Test Cases:

- Array Length: 7, print the configurations.
- Array Length: 4.
- Array Length: 2.
- Array Length: 10.

4- Running and Results:

Input is given to the function directly. The driver code is designed to be launched by a makefile in a Linux system. Each result represents the number of configurations returned.

- Printed configurations, 16
- 3
- 0
- 52

5- Time Complexity: n = length of the array, L = length of the given block
 c = counter

```
int countConfigs(int length, boolean printSwitch){      T(n)
    if(printSwitch)
        print the configurations;      Θ(n)

    return countArrayConfigs(length, 3, printSwitch);  C(n, 3)
}
```

Expression: $T(n) = O(n) + C(n, 3)$

```

int countArrayConfigs(int length, int blockLen, boolean printSwitch){    C(n, L)
    if(blockLen == length){                 $\Theta(1)$ 
        if(printSwitch)
            print the configurations;     $\Theta(n)$ 
        return 1;
    }
    else if(blockLen < length)                 $\Theta(1)$ 
        return  countBlockConfigs(length, blockLen, 0, new String(), printSwitch) +    B(n, L, 0)
                countArrayConfigs(length, blockLen + 1, printSwitch);                C(n, L + 1)
    else
        return 0;
}

```

Recurrence relation: $C(n, L) = \Theta(1) + B(n, L, 0) + C(n, L + 1)$ $L < n$
 $C(n, L) = O(n)$ $L = n$ (base case)

```

int countBlockConfigs
    (int length, int blockLen, int counter, String previous, bool printSwitch)    B(n, L, c)
{
    if(blockLen == length){                 $\Theta(1)$ 
        if(printSwitch)
            print the configurations;         $\Theta(n)$ 
        return 1;
    }
    else if(blockLen > length)                 $\Theta(1)$ 
        return 0;
    else {
        int total = 0;
        if(length >= blockLen*2 + 1){         $\Theta(1)$ 
            if(printSwitch)
                print the configurations;     $\Theta(n)$ 

            total = countBlockConfigs        B(n - L - 1, L, L)
                (length - blockLen - 1, blockLen, blockLen, previous, printSwitch);
            total = total*2 - 1;             $\Theta(1)$ 
        }

        if(counter > 0)
            total += countBlockConfigs
                (length - 1, blockLen, counter - 1, previous, printSwitch) - 1;    B(n - 1, L, c - 1)

        if(printSwitch)
            print the configurations;         $\Theta(n^2)$ 

        total += length - blockLen + 1;     $\Theta(1)$ 
        return total;                     $\Theta(1)$ 
    }
}

```

Reccurnece relation: $B(n, L, c) = O(n^2) + B(n - L - 1, L, L) + B(n - 1, L, c - 1)$ (worst case)
 $= O(n)$ $L = n$ (base case)