REPORT LAB 5

Problem 1:

**1. Base Cases of function puzzle:**

The base cases in the puzzle function are:

* base == limit: This condition checks if the base is equal to the limit. If it is, the function returns 1.

**2. Recursive Case(s) of function puzzle:**

The recursive case in the puzzle function is:

* *else*: This block executes when neither of the base cases is met (i.e., base is not greater than limit and not equal to limit). In this case, the function performs the recursive call: return base \* puzzle(base + 1, limit). Here, it returns the product of base and the result of calling puzzle with base + 1 as the new base and limit remaining the same.

**3. Output for Function Calls:**

a. System.out.print(puzzle(14,10));

In this call, base is 14 and limit is 10. Since 14 is greater than 10, the base case base > limit applies, and the function returns -1.

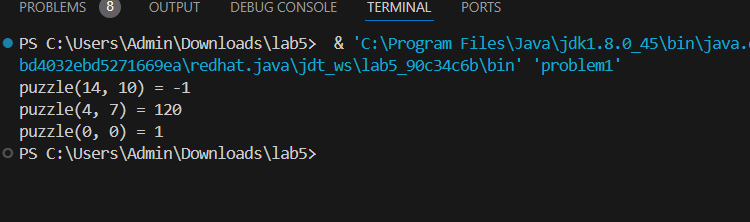
b. System.out.print(puzzle(4,7));

1. The function starts with base = 4 and limit = 7.
2. Since base (4) is less than or equal to limit (7), the recursive case applies.
3. The function calls itself recursively with base + 1 = 5 and limit = 7.
4. This process continues until base becomes 7.
5. At base = 7 (which is still less than or equal to limit), the function cannot call itself recursively anymore because base would be greater than limit.
6. Since base == limit, the condition for the base case (that we previously assumed to be incorrect) might not actually be a base case in this context. Instead, the calculation continues without a recursive call.
7. The function returns 1 (assuming this is the intended behavior for base == limit).

Following the recursive calls back up the chain, the product is calculated: 4 (from the initial call) \* 5 (from the second call) \* 6 (from the third call) \* 7 (from the fourth call) \* 1 (assumed base case behavior) = 120.

c. System.out.print(puzzle(0,0));

In this case, base and limit are both 0. The base case base == limit applies directly, and the function returns 1.

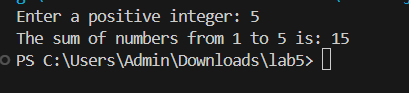


Problem 3:

This recursive function **sum(int n)** computes the sum of all numbers from 1 to n. It follows the recursive approach by breaking down the problem into smaller subproblems:

1. **Base case:** When n is 0, the sum of numbers from 1 to 0 is 0.
2. **Recursive case:** For any positive integer n, the sum of numbers from 1 to n is equal to n plus the sum of numbers from 1 to n-1. This relationship is captured in the recursive call **return n + sum(n - 1)**.

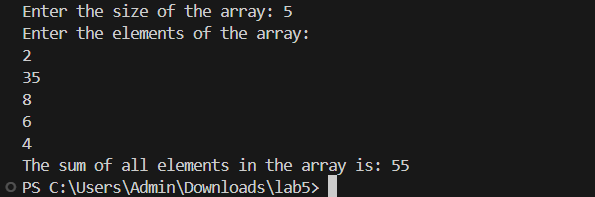
The function recursively calls itself with a smaller value of n until it reaches the base case, at which point it returns 0. Then, the function returns the sum of n and the sum of numbers from 1 to n-1, effectively computing the sum of numbers from 1 to n.



Here, we have the input integer is 5. The sum from 1 to 5 is 1 + 2 + 3 +4 + 5 = 15 .

Problem 5:

This recursive function **findSum(int[] a, int n)** computes the sum of all elements in an array. It is almost the same as Problem3 but instead calculate the sum from 1 to n, we calculate the sum of any elements that input in an array

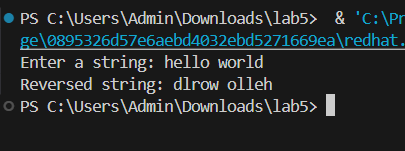


Problem 7:

This recursive function **reverseString(String str)** reverses a given string using recursion.

1. **Base case:** If the string is empty or has only one character, it is already reversed, so we return the string itself.
2. **Recursive case:** For any string with length greater than 1, we reverse the substring excluding the first character, and then append the first character at the end. This relationship is captured in the recursive call **return reverseString(str.substring(1)) + str.charAt(0)**.

The function recursively calls itself with a smaller substring until it reaches the base case, at which point it starts concatenating the characters in reverse order. Finally, the reversed string is returned.



Problem 8:

In this problem, he function takes a set as input and returns a list of all subsets of the given set.

* **Base Case**: If the set is empty, the function returns a list containing the empty set.
* **Recursive Case**: For a non-empty set, the function generates subsets by either including or excluding the first element and recursively generating subsets for the rest of the elements.
* **Combination**: It combines the subsets generated by including and excluding the first element to form all possible subsets of the given set.

This recursive approach effectively explores all possible combinations of elements in the set, resulting in the generation of all subsets.

In this situation, we have the **main** method that initializes a list **set** with elements **{1, 2, 3}**, the **set** list as input and the output is prints all subsets generated.

