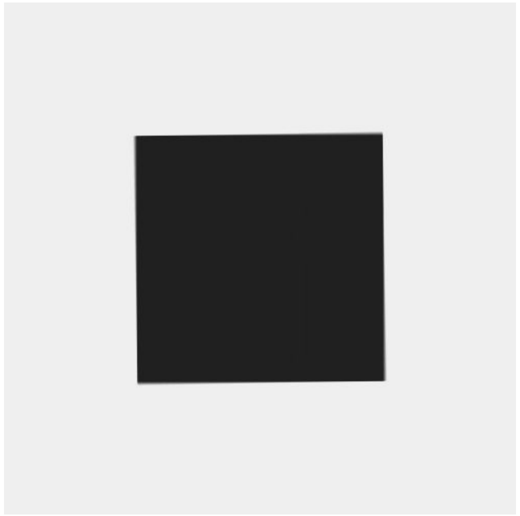
Recursion

Lab 13: https://maryash.github.io/135/labs/lab_13.html

- Sadab Hafiz

Introduction to Recursion

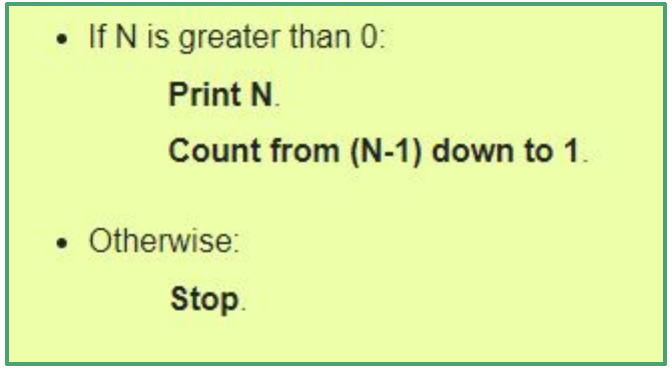
Recursion is a programming technique where the solution to a problem depends on solutions to smaller instances of the same problem.



Programming languages support recursion by allowing functions to call themselves within their code. Any function that calls itself within the implementation would be considered a recursive function.

Recursion Example Pseudocode

Pseudocode to count from an integer N to 1 using recursion:



```
• If N is greater than 0:
        Print N.
        Count from (N-1) down to 1.

• Otherwise:
        Stop.
```

A recursive program would not stop if it did not have a base case. When n == 0, there is nothing left to count, so the function does not need to call itself any more. In other words, countdown(0) is the simplest possible subproblem, which does not need to do any recursive calls to itself.

Calling this function with input 10 produces the following output:

```
10   9   8   7   6   5   4   3   2   1
```

Recursion Example C++ Code

```cpp
// to count from n down to 1

void countdown(int n) {

    if (n > 0) {

        cout << n << endl;     // print n

        countdown(n-1);        // make recursive call, counting from (n-1) down to 1

    }

    else {

        cout << "Done!";       // base case

    }

}
```

Recursion replaces Loops

All loops can be replaced with recursive functions.

It is possible to write code without using any loops, implementing all iterations as recursive function calls, like in the example `countdown` function.

To practice that, in this lab, **you will not be allowed to use loops**. Instead of loops, write functions and call them recursively for all the tasks.

Fun fact: Look up recursion on Google. You'll get it eventually!

Task A

void printRange(int left, int right);
Prints all numbers in range left ≤ x ≤ right, separated by spaces without using loops, global or static variables. Usage example:

```
int main() {
    printRange(-2, 10);              // output: -2 -1 0 1 2 3 4 5 6 7 8 9 10
}
```

Base Case: When left > right, the range is empty and the program should not print any numbers.

Task B

int sumRange(int left, int right);

Computes the sum of all numbers in range left ≤ x ≤ right without using loops, global or static variables. Function usage:

```
int main() {
     int x = sumRange(1, 3);
     cout << x << endl;                         // prints 6
     int y = sumRange(-2, 10);
     cout << y << endl;                         // prints 52
}
```

Function implementation

int sumRange(int left, int right);
The range is the same as Task A. Therefore the base case would be similar. This function is slightly different since we have to return an int.

Base Case: The sum can be computed only when left is less than or equal to right. Therefore, we should return 0 when left is greater than right. Returning 0 wouldn't affect our sum.

Let's pretend our range is between 2 and 6 (left = 2 and right = 6)
sumRange(2, 6) = 2 + 3 + 4 + 5 + 6
sumRange(2, 6) = 2 + (3 + 4 + 5 + 6)
sumRange(2, 6) = 2 + sumRange(3, 6)
sumRange(left, right) = left + sumRange(left+1, right)

Task C

```
int sumArray(int *arr, int size);
```

Receives an array (as a pointer to its first element) and the size of the array, and should return the sum of its elements.

The function itself should not do any new dynamic memory allocations.

The lab instruction suggests one approach using a helper function. You can also do it without a helper function.

Accessing the items in the pointer array

We know that parameter arr is a pointer to the first item in the array. We can access all the other items of the array like this:

arr[index]	arr[index] (dereferenced)	pointer to arr[index]
arr[0]	*arr	arr
arr[1]	*(arr + 1)	arr + 1
arr[2]	*(arr + 2)	arr + 2
...	...	...
arr[size-1]	*(arr + size - 1)	arr + size - 1

Note: parameter size is one more than the index of the last item which is why last line of the table contains size-1 for the last item.

Function implementation

int sumArray(int *arr, int size);
arr is pointer to first item. Thus, left = *arr.
arr + size - 1 is pointer to last item. Thus, right = *(arr + size-1).

Base Case: The sum can be computed only when the left pointer is less than or equal to the right pointer. Therefore, we should return 0 when arr > (arr + size -1). Also, the size is decreasing on each recursive call and it will eventually be 0. That can also be used as the base case.

Function implementation continued...

Let’s say we have an array with items [1, 2, 3, 4, 5] (size = 5),
sumArray(*[1, 2, 3, 4, 5], 5) = 1 + 2 + 3 + 4 + 5
sumArray(*[1, 2, 3, 4, 5], 5) = 1 + (2 + 3 + 4 + 5)
sumArray(*[1, 2, 3, 4, 5], 5) = 1 + sumArray(*[2, 3, 4, 5] , 4)
sumArray(arr, size) = *arr + sumArray(arr + 1, size - 1)

This is similar to Task B sumRange( ) function where we had:
sumRange(left, right) = left + sumRange(left+1, right)

If the base case is not met, return *arr + sumArray(arr + 1, size - 1)

Task D

bool isAlphanumeric(string s);

Returns true if all characters in the string are letters and digits, otherwise returns false. Implement it without using loops.

```
int main () {
     cout << isAlphanumeric("ABCD") << endl;                    // true (1)
     cout << isAlphanumeric("Abcd1234xyz") << endl;             // true (1)
     cout << isAlphanumeric("KLMN 8-7-6") << endl;              // false (0)
}
```

Pseudocode for Task D

The logic is similar to the sumRange() function from Task B:

```
// everything in green is pseudocode

bool isAlphanumeric(string s){

     if(the string is empty){

          return true;

     }

     return (first character is alphanumeric) && isAlphanumeric(rest of the string s);

}
```

Use the <string> function substr(pos, len) to get the rest of the string. You can also use the <cctype> function isalnum() to check if the first character is alphanumeric.