Lab 12. Recursion

Introduction

Recursion in 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.

The main principle of recursive programming: Function **F** solves a complex problem **P** by breaking it into one or more slightly smaller problems: Q_1, Q_2 … Q_m, calling itself to solve them:

$F(Q_1)$, $F(Q_2)$, … $F(Q_m)$, then combining the results.

In the simplest such situation, the given task P is reduced to only one slightly smaller subtask Q:

How to count from N down to 1:

- If N is greater than 0:

  Print N.

  Count from (N-1) down to 1.

- Otherwise:

  Stop.

Will print:

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

The same in C++:

```
// 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
    }
}
```

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 itself.

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 above.

To practice that, in this lab, **you will not be allowed to use loops**. Instead `for`, `while`, and `do while` constructs, write functions and call them recursively.

Task A. Print all numbers in range

Write a program `recursion.cpp`, defining a function

```
void printRange(int left, int right);
```

that prints all numbers in range `left` ≤ x ≤ `right`, separated by spaces. (Don't use loops, global or static variables.)

A usage example:

```
int main() {
    printRange(-2, 10);
```

```
}
```

Will print:

```
-2 -1 0 1 2 3 4 5 6 7 8 9 10
```

When `left > right`, the **range is empty** and the program should not print any numbers.

Task B. Sum of numbers in range

In the same program `recursion.cpp`, add a function

```
int sumRange(int left, int right);
```

that computes the sum of all numbers in range `left` ≤ x ≤ `right`. (Don't use loops, global or static variables.)

A usage example:

```
int main() {
    int x = sumRange(1, 3);
    cout << This is << x << endl;   // 6

    int y = sumRange(-2, 10);
    cout << That is << y << endl;   // 52
}
```

What makes it different from the previous example, this function has to return the answer:

- **if the range is empty, the sum is zero**.
- **Otherwise** `sum(left, right) = left + sum(left + 1, right)`.

Task C. Sum of elements in array

In the same program, add a new function

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

which 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.**

There are several approaches to this task:

- One possible strategy is to define a helper function

  ```
  sumArrayInRange(int *arr, int left, int right);
  ```

  which adds up all elements of the passed array, but only for indexes in the interval `left` ≤ `i` ≤ `right`. It can be implemented very similarly to the function `sumRange`, but it should be adding the elements of the array instead of range indices.
  Then `sumArray(arr, size)` can be defined as `sumArrayInRange(arr, 0, size-1)`.

- Alternatively, can you maybe get away with just using the original function?

A usage example:

```
int main() {

    int size = 10;
    int *arr = new[size]; // allocate array dynamically
    arr[0] = 12;
    arr[1] = 17;
    arr[2] = -5;
    arr[3] = 3;
    arr[4] = 7;
    arr[5] = -15;
    arr[6] = 27;
    arr[7] = 5;
    arr[8] = 13;
    arr[9] = -21;
```

```
    int sum = sumArray(arr, size); // Add all elements
    cout << "Sum is " << sum << endl;  // Sum is 43

    int sum = sumArray(arr, 5); // Add up first five elements
    cout << "Sum is " << sum << endl;  // Sum is 34

    delete[] arr;          // deallocate it
}
```

Task D. Is string alphanumeric?

In the same program, add a new function

```
bool isAlphanumeric(string s);
```

which returns `true` if all characters in the string are letters and digits, otherwise returns `false`.

A usage example:

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

The logic is similar to the `sumRange` function:

- **if the sting is empty, return `true`**.
- **Otherwise,**
  - **check the first character**, and
  - **check that the rest of the string is alphanumeric**.

You may use the string function `substr(pos, len)`, which extracts a substring. It takes two parameters, the starting position and the length of the substring. For example:

```
string msg = "ABCDEFGH";
cout << msg.substr(2, 4);   // CDEF (start at char [2] and
                            //       take 4 characters)
```

Task E. Nested parentheses

Add a new function

```
bool nestedParens(string s);
```

which returns `true` if the string is a sequence of nested parentheses:

Strings `""`, `"()"`, `"(())"`, `"((()))"` … are all correct sequences and should return `true`. Any other symbols or mismatching parenthesis should make the function return `false`.

A usage example:

```
cout << nestedParens("((()))") << endl;      // true (1)
cout << nestedParens("()") << endl;          // true (1)
cout << nestedParens("") << endl;            // true (1)

cout << nestedParens("(((") << endl;         // false (0)
cout << nestedParens("(()") << endl;         // false (0)
cout << nestedParens(")(") << endl;          // false (0)
cout << nestedParens("a(b)c") << endl;       // false (0)
```

Task F. Fair division

Alice and **Bob** inherited a collection of paintings. However, they will receive it only if the collection can be **divided into two parts of exactly equal price**. (Otherwise, it will be donated to a local art museum.)

Is the collection divisible into two exactly equal halves? We have to find the answer.

The prices of the paintings are provided as an array of integers. For example:

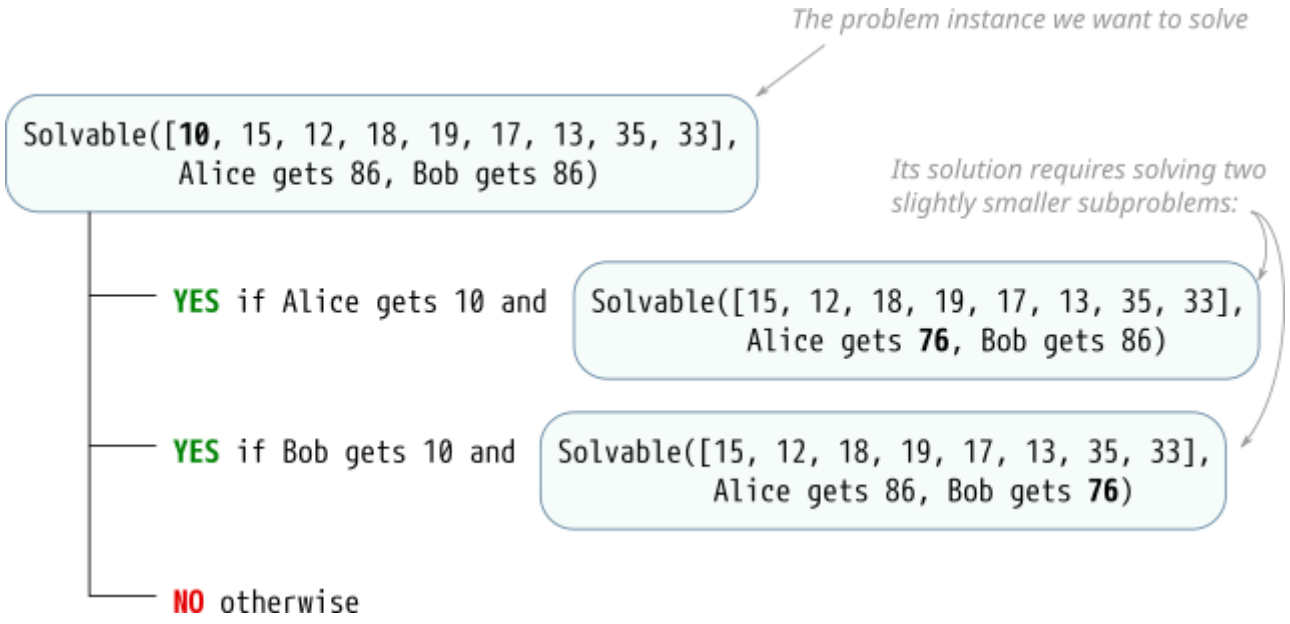
```
int prices [] = {10, 15, 12, 18, 19, 17, 13, 35, 33};
```

Here, the total sum is 172, so each person should receive the sum of 86. In this specific example, a solution exists, it is the following partition: (10 + 15 + 12 + 19 + 17 + 13) = (18 + 35 + 33) = 86.

How to solve the problem recursively?

Consider the example above. Is it possible to divide [10, 15, 12, 18, 19, 17, 13, 35, 33] into sums of 86 and 86?

Each item should go either to Alice or to Bob. Let's take the first item, **10**. Should we give it to Bob or Alice? In either case, there can be a solution. So, let's try both options:



If we can give **10** to Alice, and the rest can be divided so that she gets 76 and Bob gets 86, then a solution exists (and Alice gets 10).

Also, if we can give **10** to Bob and the rest can be divided so that he gets 76 and Alice gets 86, then the solution also exists (and Bob gets 10).

Otherwise, there is no solution.

Programming task

In the same program, write a function:

```
bool divisible(int *prices, int size);
```

which returns `true` if the collection is divisible, and `false` otherwise. The prices are provided in the array `prices`, and `size` is the number of items in the array.

Your function should not allocate new memory dynamically. Pass the same array data into recursive function calls. If you need more variables such as `left` and `right` boundary variables, or the amounts that should be given to Alice and Bob, make a helper function with any necessary extra variables.

(It is possible to make the program to actually print out the solution, who gets which item. For that, in each of the YES branches, once you know that a solution to the subproblem exists, print the current item and the name of the person who gets it.)

This is not a simple task, but if you can do it, this is great!

How to submit your programs

Each program should be submitted through Gradescope

Write separate programs for each part of the assignment.
Submit only the source code (.cpp) files, not the compiled executables.
Each program should start with a comment that contains your name and a short program description, for example:

```
/*
Author: your name
Course: CSCI-136
Instructor: their name
Assignment: title, e.g., Lab1A

Here, briefly, at least in one or a few sentences
```

```
describe what the program does.
*/
```