Informatics II
Exercise 2

Spring Semester 2022
Week 3
Recursion

Task 1. Short Questions

Solve all of the following subtasks without executing any given code.

(a) Consider the C function below. How many recursive calls will be executed for `rec_fun1(3)`?

```c
int rec_fun1(int n) {
    if (n == 13) {
        return 12;
    }
    else {
        return 11 * rec_fun1(n + 2);
    }
}
```

Answer: 5

(b) Consider the following function in C
What will be the return value of the call `rec_fun2(3, 0)`?

```c
int rec_fun2(int x, int y) {
    if (x <= 0) {
        y = y + 5;
        return y;
    }
    else {
        int t1 = rec_fun2(x - 1, y + 2);
        int t2 = rec_fun2(x - 2, y + 3);
        return t1 + t2;
    }
}
```

Answer: 54

(c) Consider the following two C functions:
What will be the output on the console for the call `rec_fun3a(5)`?

```c
void rec_fun3a(int n) {
    if (n == 0) {
        return;
    }
    printf("%d", n);
    rec_fun3b(n - 2);
    printf("%d", n);
}

void rec_fun3b(int n) {
    if (n == 0) {
        return;
    }
    printf("%d", n);
    rec_fun3a(n + 1);
    printf("%d", n);
}
```

Answer: 53423122132435

(d) Consider the following function in C:
Formally describe the set of input values `x` and `y` for which an infinite recursion will occur (i.e. for which the base case is never reached).

```c
int rec_fun4(int x, int y) {
    if (x > y) {
        return x * y;
    }
    else {
        return rec_fun4(x - 1, y);
    }
}
```

Answer: an endless recursion occurs iff $x \leq y$ or: $\forall(x, y) \in \{(x, y) \mid x \leq y\}$

(e) Is the following statement true or false?
«A recursive function always has to have exactly one base case.»

Answer: ☐ True ☒ False

Task 2. Second Smallest Element
Write a program in C which *recursively* finds the value of the second smallest element in an arbitrary array $A[0..n-1]$ of $n > 1$ mutually distinct, strictly positive integers.
A code skeleton with the code for reading in an array from the user is provided as `task2_skeleton.c`.

C code solution:

```c
#include <stdio.h>
const int MAX_LENGTH = 1000;

/**
 * Returns the position of the second smallest element in an array
 * @param arr array in which the second smallest element shall be found
 * @param upper_limit array index until which the array is to be considered
 * @param smallest smallest element found so far
 * @param second_smallest second smallest element found so far
 * @pre elements of array are strictly positive and mutually distinct &&
        array has at least two elements && upper_limit >= 0
 */
int get_second(int arr[], int upper_limit, int smallest, int second_smallest) {
    /* Base case: if only the first element of the array is still to be considered,
    there are three possible cases:
       - 1a: the first element can be the smallest, then return the smallest until here
       - 1b: the first element is the second smallest in the array, then return this element
       - 1c: the first element is neither the smallest nor the second smallest, then return
             the second smallest until here
     */
    if (upper_limit == 0) {
        if (arr[0] < smallest) {
            // case 1a
            return smallest;
        }
        else if (arr[0] < second_smallest && arr[0] != smallest) {
            // case 1b
            // note: the second part of the condition can apply for an array of length 2
            return arr[0];
        }
        else {
            // case 1c
            return second_smallest;
        }
    }

    /* Recursive case: if more than the first element of the array is to be considered,
    check how the element at the upper bound influences minima determined until now and
    then recursively apply the function to the remaining array
    */
    if (smallest < 1) {
        /* handle the case where the recursive case has never been executed before and
        therefore both variables smallest and second_smallest are below 1 */
        if (arr[upper_limit] < arr[upper_limit - 1]) {
            smallest = arr[upper_limit];
            second_smallest = arr[upper_limit - 1];
        }
        else {
            smallest = arr[upper_limit - 1];
            second_smallest = arr[upper_limit];
        }
    }
    else {
        if (arr[upper_limit] < smallest) {
            second_smallest = smallest;
            smallest = arr[upper_limit];
        }
        else if (arr[upper_limit] < second_smallest && arr[upper_limit] != smallest) {
            /* note: the second part of the condition can apply if the second but last
            element is the smallest in the array */
            second_smallest = arr[upper_limit];
        }
    }
```

```c
    printf("get_second(arr, %d, %d, %d)\n", upper_limit, smallest, second_smallest);
    return get_second(arr, upper_limit - 1, smallest, second_smallest);
}

int main() {
    printf("Values of array separated by spaces (non-number to stop): ");
    int arr[MAX_LENGTH];
    int pos = 0;
    while (scanf("%d", &arr[pos]) == 1) {
        pos++;
    }

    int result = get_second(arr, pos - 1, -1, -1);
    printf("Second smallest element in array: %d\n", result);
    return 0;
}
```

Pseudo code solution:

Algorithm: get_second($arr[0..n-1], upper\_limit, smallest, second\_smallest$)

if $upper\_limit == 0$ **then**
  if $arr[0] < smallest$ **then**
    return $smallest$
  else if $arr[0] < second\_smallest \wedge arr[0] \neq smallest$ **then**
    return $arr[0]$
  else
    return $second\_smallest$
if $smallest < 1$ **then**
  if $arr[upper\_limit] < arr[upper\_limit - 1]$ **then**
    $smallest = arr[upper\_limit]$
    $second\_smallest = arr[upper\_limit - 1]$
  else
    $smallest = arr[upper\_limit - 1]$
    $second\_smallest = arr[upper\_limit]$
else
  if $arr[upper\_limit] < smallest$ **then**
    $second\_smallest = smallest$
    $smallest = arr[upper\_limit]$
  else if $arr[upper\_limit] < second\_smallest \wedge arr[upper\_limit] \neq smallest$ **then**
    $second\_smallest = arr[upper\_limit]$
return get_second($arr[0..n-1], upper\_limit - 1, smallest, second\_smallest$)

Task 3. Blinking Light

Consider a LED which is emitting different blinking patterns. In each second, the LED can either exhibit exactly two short blinks (which will be denoted as **--** in the following examples) or exactly one long blink (denoted as **—**). Hence a blinking pattern will consist of a certain number n of blinks (each either short or long). For example, a blinking pattern consisting of $n = 3$ blinks (regardless whether short or long) can have one of the following 3 configurations:
(-- —), (— --), (— — —)
A blinking pattern consisting of $n = 4$ blinks can be constructed in 5 different ways:
(-- --), (-- — —), (— -- —), (— — --), (— — — —)
Write a program in C which calculates the number of different blinking patterns which consist of exactly n blinks.
A code skeleton with the code for reading in an integer value is provided as `task3_skeleton.c`.

C code solution:

```c
#include <stdio.h>

/**
 * Returns the number of ways how an LED can blink according to certain rules
 * @param num_blinks the total number of blinks by the LED
 * @pre num_blinks > 0
 */
int num_patterns(int num_blinks) {
    if (num_blinks == 1) {
        return 1;
    }
    if (num_blinks == 2) {
        return 2;
    }
    return num_patterns(num_blinks - 1) + num_patterns(num_blinks - 2);
}

int main() {
    int input;
    printf("Enter the number of blinks: ");
    scanf("%d", &input);
    int result = num_patterns(input);
    printf("Number of blink patterns: %d\n", result);
    return 0;
}
```

Pseudo code solution:

Algorithm: num_patterns(*num_blinks*)

if *num_blinks == 1* **then**
  return 1
if *num_blinks == 2* **then**
  return 2
return num_patterns(*num_blinks* − 1) + num_patterns(*num_blinks* − 2)

Task 4. Fractal Circles

Devise a pseudo code algorithm which will produce in a Cartesian coordinate system a picture according to the following rules:
Initially, a circle with radius r_0 is drawn with its centre at the position $(x_0, y_0) = (0, 0)$. At each point of intersection of a circle with the x-axis, another circle is drawn which has half the radius of the circle intersecting the x-axis. No circle with a radius smaller than $r_{min} = 10$ should be drawn. See Figure 1 for an example produced for $r_0 = 256$.
Assume that a subroutine *draw_circle*(*pos_x*, *pos_y*, *radius*) does already exist and will draw a circle around a centre position at coordinate (*pos_x*, *pos_y*) with the radius given as an argument.

Pseudo code solution:

Algorithm: fractal_circle(xpos, ypos, radius)

if *radius* ≥ 10 **then**
  draw_circle(*xpos*, *ypos*, *radius*)
  fractal_circle(*xpos* + *radius*, *ypos*, *radius*/2)
  fractal_circle(*xpos* − *radius*, *ypos*, *radius*/2)

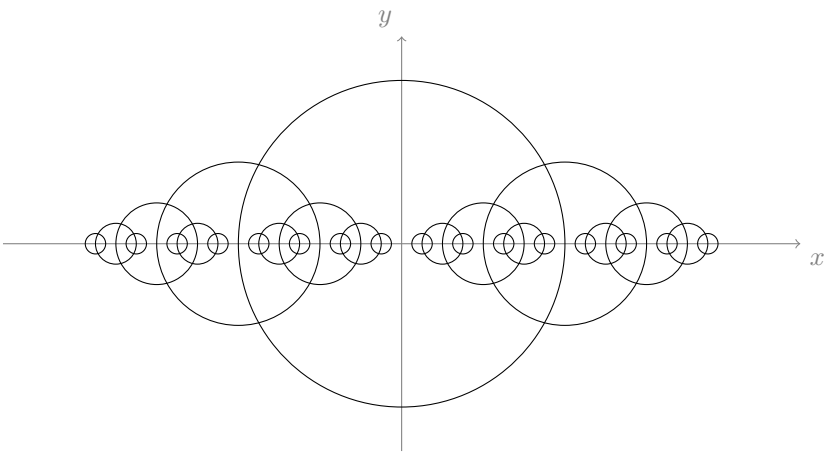

Figure 1: Figure produced according to the given rules for $r_0 = 256$.