Universität Zürich^UZH

Institut für Informatik

Informatics II

Tutorial Session 3

Wednesday, 9th of March 2022

Discussion of Exercise 2,
Recursion

14.00 – 15.45

BIN 0.B.06

Agenda

- Intro
- Review of Exercise 2
- Preview on Exercise 3
- Summary and Wrap-Up

Universität Zürich^UZH

Institut für Informatik

Recursion

- Explaining Recursion
- Recursion Trees
- Recursion and Memory
- Recursion vs. Iteration
- Different Types of Recursion
- Example Problems and Applications

Recursion: Code Example I

Consider the following recursive function:

```
int recFun(int n) {
      if (n < 10) {
            return n;
      } else {
            int a = n / 10;
            int b = n % 10;
            return recFun(a + b);
      }
}
```

What is the result of the call `recFun(648)`?

A: **8**
B: **9**
C: **54**
D: **72**
E: **648**

Result is B: 9.

The recursion tree looks as follows:

$$\begin{tikzcd} \text{recFun(648)} \arrow[d, dash] \\ \text{recFun(72)} \arrow[d, dash] \\ \text{recFun(9)} \end{tikzcd}$$

Recursion: Code Example II

Consider the following code snippet. What will be the output?

```
#include <stdio.h>

void whatDoesItDo(int n) {
    if (n > 0) {
        int a = 1;
        int b = 2;
        int c = a + b;
        whatDoesItDo(c);
    }
    else {
        printf("world!\n");
    }
}
```

```
int main() {
    printf("Hello ");
    int x = 42;
    whatDoesItDo(x);
    return 0;
}
```

→ **Segmentation fault: stack overflow**. The base case of the recursive function can never be reached. Note that the «steering variable» n of the recursion will never be changed and remain 3 if it is bigger than zero in the first call of the function. Therefore the memory will be filled with return addresses until has been used completely – at which time the program will crash.

Recursion: Comprehension Question

What is the difference between an infinite loop and an infinite recursion?

Exercise 2, Task 1a

How many recursive calls will be executed for rec_fun1(3)?

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

Solution: 5

Exercise 2, Task 1b

What will be the return value of the call rec_fun2(3, 0)?

```
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;
    }
}
```

Solution: 54

Exercise 2, Task 1c

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);
}
```

Solution: 53423122132435

Exercise 2, Task 1d

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).

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

Solution: x <= y

Exercise 2, Task 1e

Is the following statement true or false?

«A recursive function always has to have exactly one base case.»

Recursion: What You Should Already Know

- Function / method which calls itself (directly or indirectly)
- Recursive case and base case
- Can be rewritten as iteration (and vice-versa)

- Main idea: solve a smaller problem which is an exact copy of the bigger problem but smaller

What Algorithm Is This?

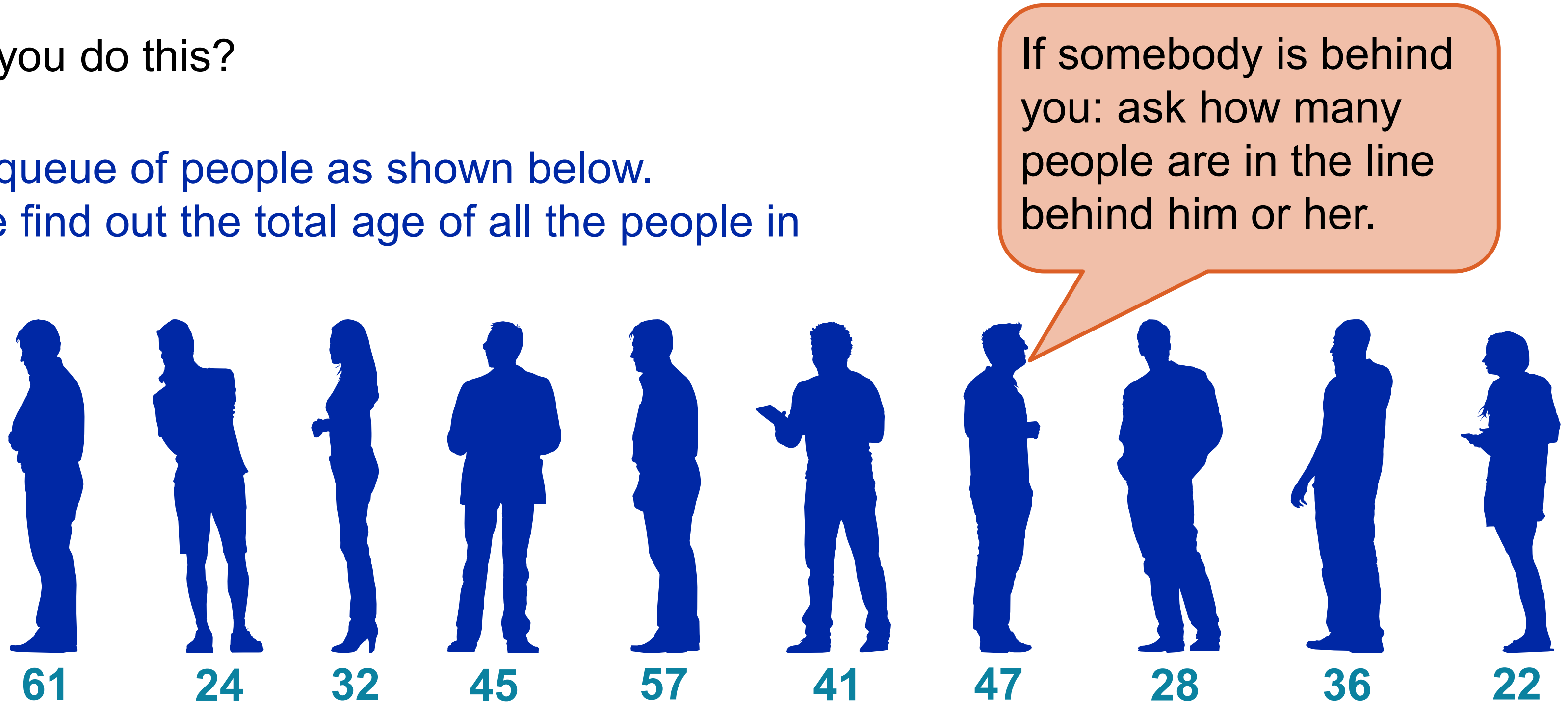
```
Algorithm: SomeAlgo(A[1..n])
for i = n to 2 do
    for j = 2 to i do
        if A[j] < A[j−1] then
            t = A[j];
            A[j] = A[j−1];
            A[j−1] = t;
```

Explaining Recursion

Assume you want to explain recursion to your 12 year old cousin.

How would you do this?

Consider a queue of people as shown below.
How can we find out the total age of all the people in the queue?


If somebody is behind you: ask how many people are in the line behind him or her.
61
24
32
45
57
41
47
28
36
22

Explaining Recursion: Iterative vs. Recursive Solution

iterative approach:
go from person to person and add together their ages

Algorithm: GetTotalAgeIter(peopleQueue)

```
totalAge = 0
for k = 1 to length(peopleQueue) do
    totalAge = totalAge + age of kth person
```

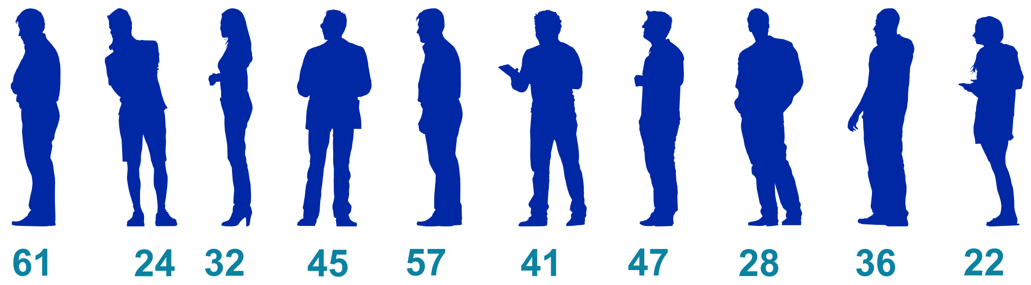
recursive approach:
ask person behind you to find out using the same strategy

Algorithm: GetTotalAgeRec(peopleQueue)

```
if no person standing behind you then
    tell person before you your own age
else
    ask person behind you for age
    tell answer to person before you
```

Explaining Recursion

Use a C array to represent ages of people in queue; use C language to implement the solution:



$\downarrow$

arr =

61	24	32	45	57	41	47	28	36	22
0	1	2	3	4	5	6	7	8	9

$$\text{sum}(A[1..k]) = \begin{cases} A[k] & \text{if k is the last position} \\ \text{sum}(A[1..k-1]) + A[k] & \text{otherwise} \end{cases}$$

Exercise Recursion: Sum of an Integer Array

Implement a function in C which recursively calculates the sum of the elements of an integer array.

```c
int sumrec(int arr[], int n, int i) {
    if (i == n - 1) {
        return arr[i];
    } else {
        return arr[i] + sumrec(arr, n, i + 1);
    }
}
```

Exercise Recursion: Average of an Integer Array

Calculate the average of the elements in an array of integers recursively.

Exercise Recursion: Average of an Integer Array

By definition, the average is the sum of all elements divided by the number of the elements. Conversely, the sum can be expressed as the average multiplied with the number of elements:

$$\text{avg}(A[1..n]) = \frac{\text{sum}(A[1..n])}{n} \quad \Rightarrow \quad \text{sum}(A[1..n]) = \text{avg}(A[1..n]) \cdot n$$

Further, we can replace the sum of an array by adding the very last element to the sum of the all the previous elements of the array. The latter can then be expressed in terms of its average as shown above:

$$\text{sum}(A[1..n]) = \text{sum}(A[1..n-1]) + A[n] = \big(\text{avg}(A[1..n-1]) \cdot (n-1)\big) + A[n]$$

Hence, we can write the following recursive formulation for the average:

$$\text{avg}(A[1..n]) = \begin{cases} A[1] & \text{if } n = 1 \\ \dfrac{\big(\text{avg}(A[1..n-1]) \cdot (n-1)\big) + A[n]}{n} & \text{otherwise} \end{cases}$$

Exercise Recursion: Average of an Integer Array

```
double average(double arr[], int n) {
      if (n == 1) {
            return arr[1];
      } else {
            return (average(arr, n-1) * (n-1) + arr[n-1]) / n;
      }
}
```

base case

recursive call

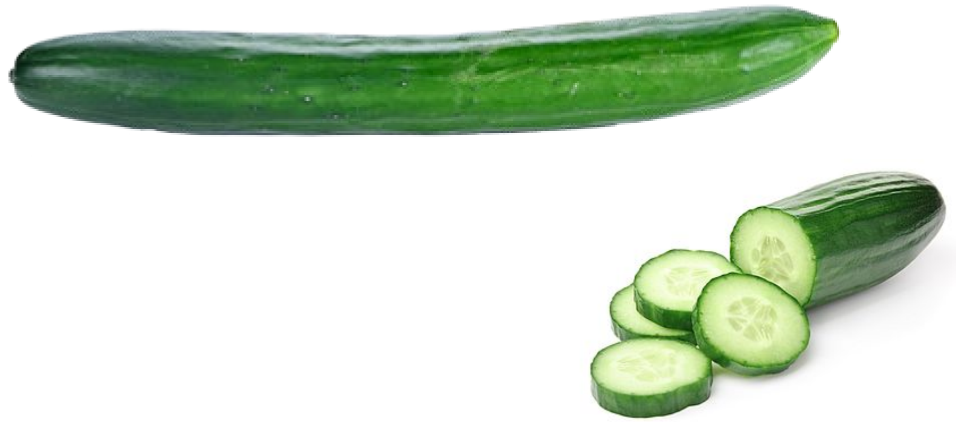
Additional Example: Comparing Iteration and Recursion

```
int iterativeFirstUpper(char str[]) {
    int pos = 0;
    while (str[pos] != '\0') {
        if (str[pos] >= 'A' && str[pos] <= 'Z') {
            return pos;
        }
        pos++;
    }
    return -1;
}
```

```
int recursiveFirstUpper(char str[], int pos) {
    if (str[pos] == '\0') {
        return -1;
    }
    if (str[pos] >= 'A' && str[pos] <= 'Z') {
        return pos;
    }
    return recursiveFirstUpper(str, pos + 1);
}
```

Explaining Recursion: How To Make Cucumber Salad

- How do cut a cucumber to make a salad iteratively and recursively?
- How do you eat a plate of soup recursively?



Cucumber Salad and Soup: Pseudocode Algorithms

Algorithm: cut(cucumber)

```
if less than 1 cm of the cucumber is left then
    stop, you're done
else
    remove a 1 cm slice from the cucumber
    cut(remaining cucumber)
```

Algorithm: eat(soup)

```
if nothing of the soup is left then
    stop, you're done
else
    eat as much of the soup as fits on spoon
    eat(remaining soup)
```

Recursion: «Recursive Leap of Faith»

How can you transport 1000 elephants to the moon?

→ If all elephants already are on the moon, we're done. Else, ship one elephant to the moon first and then use the some approach on the remaining 999.

This kind of logic applied in recursive solutions oftentimes can seem a bit «magical». One just has a solution for a very small (sometimes even trivial) problem and then just calls the same approach on a reduced copy of the complete problem which then will «somehow» solve the problem.

This part of constructing a recursive solution is therefore sometimes called the «Recursive Leap of Faith».

Recursion: Comprehension Question

Can there be a meaningful recursive function which has return type `void` and no arguments, e.g. a function with signature `void myRecFun(void)`? Explain your answer.

Exercise 2, Task 2: Second Smallest Element

```
int get_second(int arr[], int upper_limit, int smallest, int second_smallest) {
    /* Base case */
    if (upper_limit == 0) {
        if (arr[0] < smallest) {
            return smallest;
        }
        else if (arr[0] < second_smallest && arr[0] != smallest) {
            return arr[0];
        }
        else {
            return second_smallest;
        }
    }
```

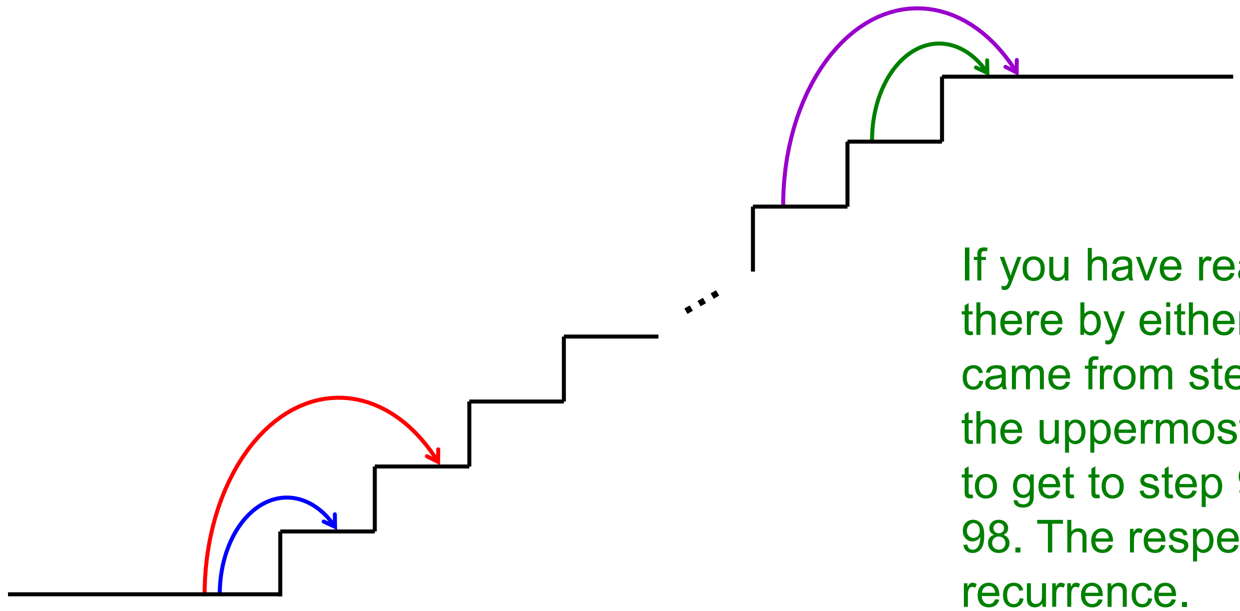
Exercise 2, Task 2: Second Smallest Element

```
        /* Recursive case */
        if (smallest < 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) {
                second_smallest = arr[upper_limit];
            }
        }
        return get_second(arr, upper_limit - 1, smallest, second_smallest);
    }
```

Optimal substructure example 1: number of ways to climb up stairs

Consider a staircase with N = 100 steps. Assume, a person can climb up one or two stairs at a time. After each step, he/she decides whether to take one or two stairs in the next step.

How many different ways are there to climb up the N = 100 stairs?

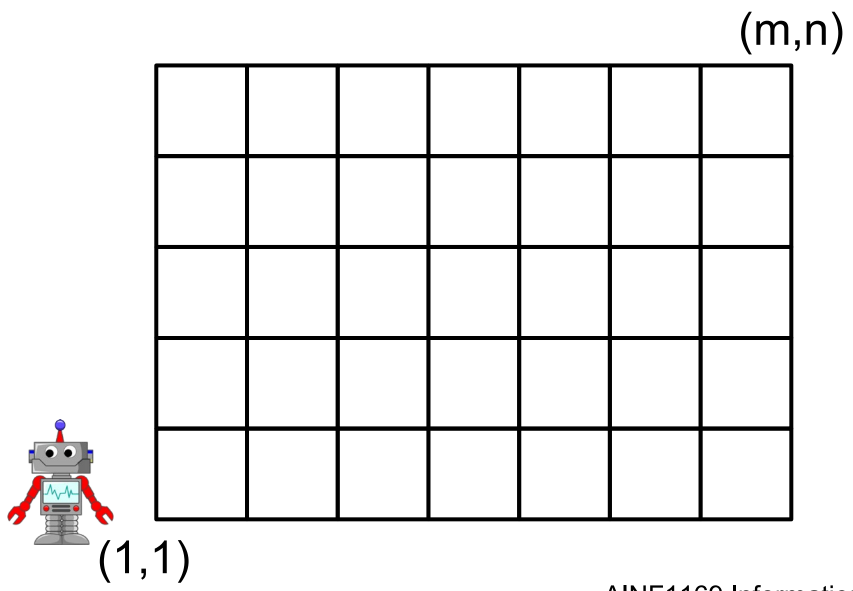


If you have reached the uppermost step (step 100), you got there by either taking one stair or two stairs, so you either came from step 98 or step 99. The number of ways to get to the uppermost step is therefore equal to the number of ways to get to step 99 the plus the number of ways to get to step 98. The respective recurrence is equivalent to the Fibonacci recurrence.

Optimal substructure example 2: robot in Manhattan

Consider a robot which is moving on a grid of size m × n. The robot can move only horizontally or vertically, thus alongside the grid axes. At the beginning, the robot is positioned at left lower edge of the grid which shall be denoted as the point (1,1).

How many different ways can the robot get from his initial position to the position (m,n) in the upper right edge? (The robot may not move in circles.)



Again, consider the robot having reached its target destination (m,n). The last move it made was either a vertical move from position (m-1,n) or a horizontal move from position (m,n-1). Therefore, the number of ways to get to the position (m,n) is the sum of the number of ways to get to position (m-1,n) and the number of ways to get to position (m,n-1).

Exercise 2, Task 3: Blinking Light

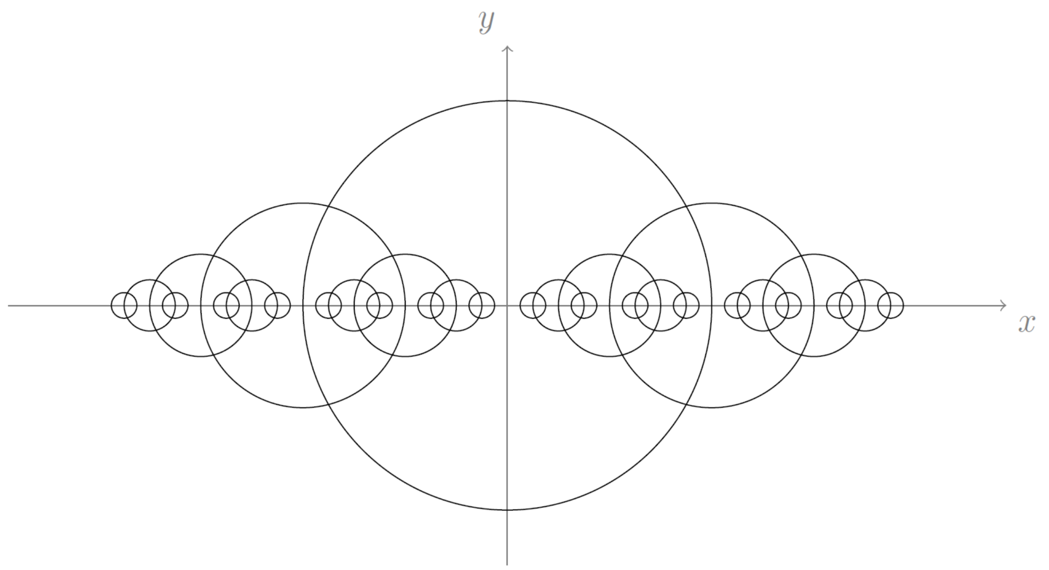
```
int num_patterns(int num_blinks) {
        if (num_blinks == 1) {
                return 1;
        }
        if (num_blinks == 2) {
                return 2;
        }
        return num_patterns(num_blinks - 1) + num_patterns(num_blinks - 2);
}
```

Exercise 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 r0 is drawn with its centre at the position (x0, y0) = (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 rmin = 10 should be drawn. See Figure 1 for an example produced for r0 = 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.



Exercise 2, Task 4: Fractal Circles

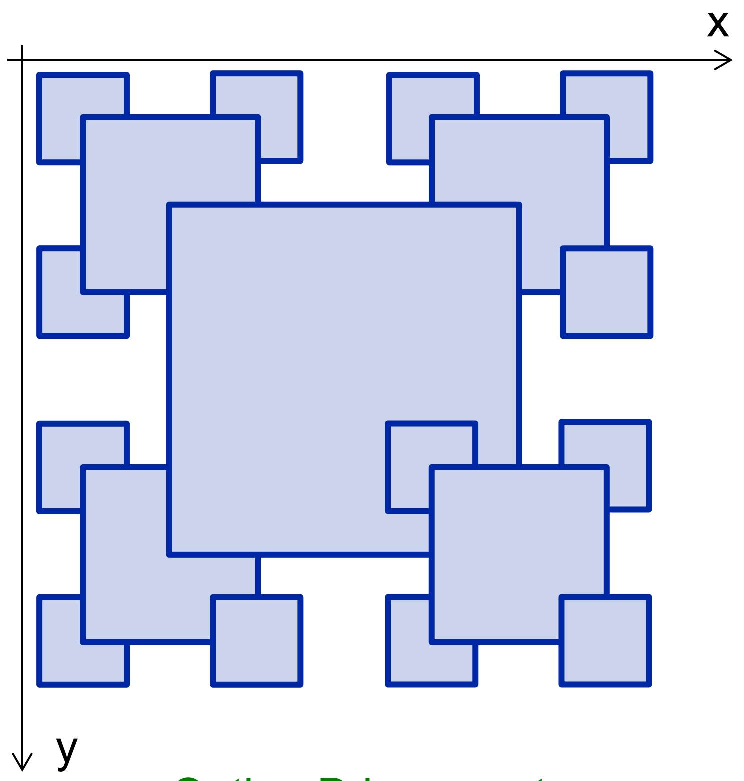
```
**Algorithm:** fractal_circle(xpos, ypos, radius)

$draw\_circle(xpos, ypos, radius)$
**if** $radius/2 \geq 10$ **then**
    $fractal\_circle(xpos + radius, ypos, radius/2)$
    $fractal\_circle(xpos - radius, ypos, radius/2)$
```

Additional Exercise: T-Square Fractal

At which position has the call of the drawing function (i.e. `drawsquare(x,y,length);`) to be inserted in order to produce the adjacent image?

```
void TSquareFractal(double x, double y, double length, int iter) {
    if (iter > 0) {
        /* Option A */
        TSquareFractal(x-length/4,y-length/4,length/2,iter-1);
        /* Option B */
        TSquareFractal(x-length/4,y+(3.0/4.0)*length,length/2,iter-1);
        /* Option C */
        TSquareFractal(x+(3.0/4.0)*length,y-length/4,length/2,iter-1);
        /* Option D */
        TSquareFractal(x+(3.0/4.0)*length,y+(3.0/4.0)*length,length/2,iter-1);
        /* Option E */
    }
    else {
        return;
    }
}
```



Option D is correct.

Universität
Zürich^UZH

Institut für Informatik

Preview on Exercise 3

Wrap-Up

- Summary
- Feedback
- Outlook
- Questions

Wrap-Up

– Summary

Outlook on Next Thursday’s Lab Session

Next tutorial: Thursday, 18.03.2021, 09.00 h, on Zoom

Topics:

- Review of Exercise 3
- Preview to Exercise 4
- Asymptotic Complexity
- Running Time Analysis
- Recurrences (Repeated Substitution, Recursion Tree, Master Method)
- …
- … (your wishes)

Questions?

Thank you for your attention.