Informatics II

Introduction, Sorting, Recursion
— SL01 —

Dr. Anton Dignös

adignoes@ifi.uzh.ch

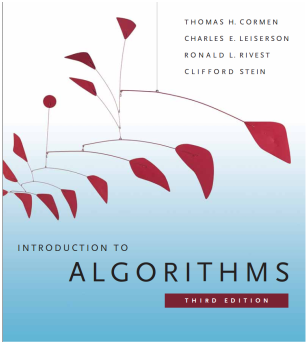
Acknowledgement

I am indebted to Prof. Dr. Michael Böhlen for providing me his slides.

TABLE OF CONTENTS — SL01

LITERATURE

- T. Cormen, C. Leiserson, R. Rivest and C. Stein (CLRS), Introduction to Algorithms, Third Edition, MIT Press, 2009.
- Available in IfI library (two books on display; 6 books for borrowing)
- Available as ebook from inside the UZH network:



`http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=343613&site=ehost-live`

Goals of this course

The main things we will learn in this course:

- To **think algorithmically** and get the spirit of how algorithms are designed.
- To get to know a **toolbox** of classical algorithms.
- To learn a number of algorithm **design techniques** (such as divide-and-conquer).
- To reason in a precise way about the **efficiency** and the **correctness** of algorithms.

SYLLABUS

1. **Introduction, basic sorting, recursion** (chap 1 in CLRS)
2. **Complexity and correctness** (chap 2, 3 in CLRS)
3. **Divide and conquer, recurrences** (chap 4 in CLRS)
4. **Heap sort, quick sort** (chap 6, 7 in CLRS)
5. **Pointers, lists, sets, abstract data types** (chap 10 in CLRS)
6. **Trees, red-black trees** (chap 12, 13 in CLRS)
7. **Hash tables** (chap 11 in CLRS)
8. **Dynamic programming** (chap 15 in CLRS)
9. **Graph algorithms** (chap 22, 23, 24 in CLRS)

Administration/1

Home page

- http://www.ifi.uzh.ch/dbtg
- https://lms.uzh.ch/url/RepositoryEntry/16974184451
- Check OLAT frequently

Course book

- Introduction to Algorithms, 3rd edition, Cormen et al.

Assistants

- Muhammad Saad, saad@ifi.uzh.ch
- Qing Chen, qing@ifi.uzh.ch

TAs:

- Andrianos Michail, andrianos.michail@uzh.ch
- Xiaozhe Yao, xiaozhe.yao@uzh.ch
- Zifan Jiang, zifan.jiang@uzh.ch

Administration/2

Lectures

- Tuesdays 14:00 - 15:45
- Fridays 12:15 - 13:45
- First lecture: Tuesday 23.02.2021
- Last lecture: Friday 25.5.2021
- Exceptions are the following dates:
  - FR 02.04.2021 12:15 - 13:45 no lecture (Spring break)
  - TU 06.04.2021 14:00 - 15:45 no lecture (Spring break)
  - FR 09.04.2021 12:15 - 13:45 no lecture (Spring break)

Administration/3

Exercises:

- There are 12 weekly exercises
- An exercise is published one week before it is being discussed in the exercise classes.
- There is no hand in of exercises.
- Attending exercises is not mandatory.
- It is strongly recommended that you solve the exercise before the exercise class.

Administration/3

Exercise classes:

- Monday 09:00 - 11:45
- Wednesday 09:00 - 11:45
- Wednesday 14:00 - 16:45
- Thursday 09:00 - 11:45
- Thursday 14:00 - 16:45

- Exceptions:
  - Lab 5, 01.04.2021, terminates at 16:00
  - Spring break, 05.04.2021 - 09.04.202, no labs
  - Lab 3, 12.05.2021, terminates at 16:00
  - Lab 4 and 5, 13.05.2021, not held
  - Lab 1, 24.05.2021, not held

ADMINISTRATION/4

Tutors:

- Adam Klebus, `adam.klebus@uzh.ch`
- Christoph Vogel, `christoph.vogel@uzh.ch`
- Hsuan-Pin Wu, `hsuan-pin.wu@uzh.ch`
- Jérome Hadorn, `jerome.hadorn.uzh.ch`
- Qasim Warraich, `qasim.warraich@uzh.ch`

ADMINISTRATION/5

Exams: The course assessment consists of two midterms and a final exam.

Exam dates (check VVZ and official WWF web pages):

- Midterm 1 (MT1, 45min): Monday, 29.03.2021, 12:15 - 13:00
- Midterm 2 (MT2, 45min): Monday, 03.05.2021, 12:15 - 13:00
- Final exam (FE): Friday 02.06.2021, 14:00 - 15:30

Your final grade is calculated as follows:

```
MAX(MT1,FE)*0.2 + MAX(MT2,FE)*0.2 + FE*0.6
```

Administration/6

Office hours

- Q/A sessions with me and assistants
- Bi-weekly on Mondays 15:00 - 16:00 starting from 01.03.2021
- Exception:
  - MO 24.05.2021 (Whit Monday) will be postponed to MO 31.05.2021

General remarks/1

- Note that there will be parts in the midterms and the final exam where you have to write working C code.
- The midterms and exam are **open book exams**.

General remarks/2

- **Hands-on exercises** are an important part of this course: an abstract understanding of the concepts is not good enough
- You must be able to **apply** your knowledge to **new** examples. Use exercises to practice this during the semester.
- Often it is most effective to first solve algorithms on paper and later key them in on the computer.
- A very important thing is to be **simple** and **precise**.

General remarks/3

- During lectures:
  - Interaction is welcome; ask questions.
  - Speed up/slow down the progress.
  - Additional explanations if desired.
- During the lectures we solve many examples. Please participate and take notes. Trying, failing and improving is good. Not trying is bad.
- This script is designed as a **working script**. Solutions to selected exercises are included at the end.

PREREQUISITES

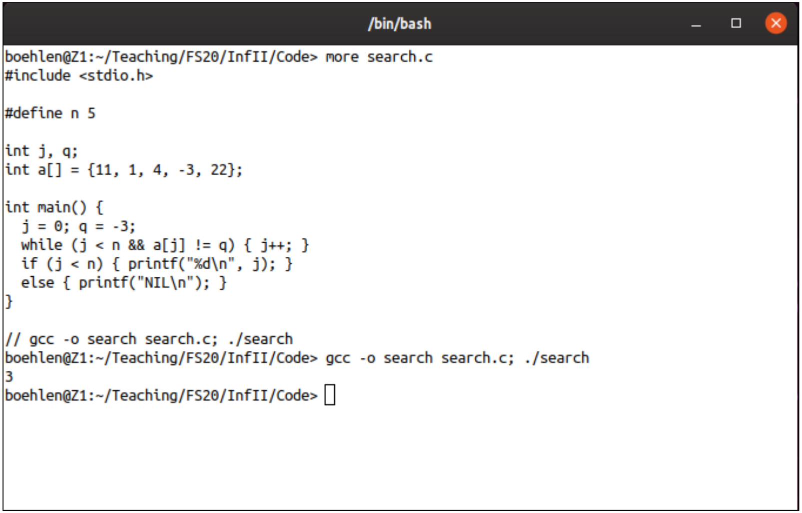
Introduction to programming (aka Informatik I)

- Data types, operations
- Conditional statements
- Loops
- Procedures and functions

Ability to edit, compile, and execute a program

- editor and terminal
- Xcode, Eclipse or equivalent

Shell


/bin/bash

```
boehlen@Z1:~/Teaching/FS20/InfII/Code> more search.c
#include <stdio.h>

#define n 5

int j, q;
int a[] = {11, 1, 4, -3, 22};

int main() {
  j = 0; q = -3;
  while (j < n && a[j] != q) { j++; }
  if (j < n) { printf("%d\n", j); }
  else { printf("NIL\n"); }
}

// gcc -o search search.c; ./search
boehlen@Z1:~/Teaching/FS20/InfII/Code> gcc -o search search.c; ./search
3
boehlen@Z1:~/Teaching/FS20/InfII/Code> 
```

Additional Acknowledgments

- The slides are based on the textbook Introduction to Algorithms from T. Cormen, C. Leiserson, R. Rivest and C. Stein.
- A very early version of these slides was developed by Simonas Saltenis from Aalborg University, Denmark.
- The course is based on a course that Michael Böhlen taught at the Free University of Bolzano, Italy.
- Kurt Ranaltar created the initial Latex version of these slides.

TABLE OF CONTENTS — SL01

What is an algorithm?

- Algorithms are about solving problems
  - Enrollment at UZH
  - Booking a module
  - Graduate from IfI@UZH
  - Get me from home to work
  - Earn lots of money
  - Simulate a jet engine
  - Human genome project: algorithms to store and analyze human DNA
  - Electronic commerce: algorithms for public-key cryptography and digital signatures
  - Internet/WWW: algorithms to determine good routes for data and to quickly find data
- Algorithms = procedures, recipes, process descriptions

History

- Etymology: from Persian mathematician al-Khwarizmi
- 400-300 B.C.
  - First algorithm: Euclidean algorithm, greatest common divisor
- 19th century
  - Charles Babbage (first mechanical computer)
  - Ada Lovelace (first computer program)
- 20th century
  - Alan Turing (Turing machine, cryptography)
  - Alonzo Church (decidability)
  - John von Neumann (Von Neumann architecture)

TERMINOLOGY

- **Data structure**
    - Organization of data to solve problem at hand
    - Solutions depend on the choice of data structure
- **Algorithm**
    - Finite sequence of unambiguous instructions
    - Step-by-step outline of computational procedure
    - Independent from choice of programming language
- **Program**
    - Implementation of given computational procedure
    - Programming language, software engineering issues

Overall picture/1

- Specification
  - Precisely specify problem
- Design
  - Specify structure/blocks of solution
  - Develop algorithms (often pseudocode procedures)
  - Goals: correctness & efficiency
- Development
  - Implement algorithm in some language
  - Goals: robustness, reusability, adaptability
- Testing
  - Verify that implementation meets specification

OVERALL PICTURE/2

What this course is **not** about

- Software architecture
- Computer architecture
- Programming languages
- Software engineering

Other related topics that we touch upon

- Computability and complexity
- Efficiency vs NP-completeness

Algorithms/1

An **algorithm** is

- Any well-defined computational procedure that
  - takes some value, or set of values, as input
  - produces some value, or set of values, as output

  i.e., an unambiguous sequence of computational steps that transform the input into the output
- A tool for solving a well-specified computational problem
  - Problem statement specifies the desired input/output relation
  - Algorithm describes a procedure for achieving such a relation

ALGORITHMS/2

Algorithmic problem

Input/output relation

$$\boxed{\text{input specification}} \rightsquigarrow \boxed{\cdots} \rightsquigarrow \boxed{\text{output specification}}$$

Specification of output as function of specification of input

Algorithmic solution

Input/output relation

$$\boxed{\text{input instance}} \rightsquigarrow \boxed{\text{algorithm}} \rightsquigarrow \boxed{\text{output instance}}$$

Procedure transforming input instance into output instance

How to develop an algorithm

1. Understanding the problem
   - Precisely define the problem.
   - Precisely specify the input and output.
   - Construct the simplest possible representative example for the problem.
   - Consider all cases.
2. Come up with a simple plan to solve the problem at hand.
   - The plan is language independent.
   - The precise problem specification influences the plan.
3. Turn the plan into an implementation
   - The problem representation (data structure) influences the implementation.

Example of an algorithm/1

- Searching problem
  - Input: a sequence (array) of n numbers $A = [a_1, a_2, \ldots, a_n]$ and a value v
  - Output: an index i such that $v = a_i (= A[i])$ or the special value `NIL` if v does not appear in A
- Instance of searching (example)
  - Input: $A = [31, 59, 26, 41, 58]$, $v = 59$
  - Output: $i = 2$

Example of an algorithm/2

Example: linear search

- Input:
  sequence of numbers $A = [a_1, a_2, \ldots, a_n]$ and value v
- Output:
  index i such that $v == A[i]$ or NIL if v does not appear in A
- Algorithms

```
Algo: LinSearch1(A,v)
p = NIL;
for $i$ = $1$ to $n$ do
    if $A[i]==v$ then p = i;
return p;
```

```
Algo: LinSearch2(A,v)
i = 1;
while $i \leq n \wedge A[i] \neq v$ do i++;
if $i \leq n$ then return i;
else return NIL;
```

Example of an algorithm/3

Comparing algorithms

- Assume that $A = [31, 41, 59, 26, 41, 58]$ and $v = 41$ Then:
    - `LinSearch1`(A, v) = 5
    - `LinSearch2`(A, v) = 2
    - `LinSearch1`$(A, v) \neq$ `LinSearch2`(A, v)
- Ambiguous problem statement: first or last index?
- There is always more than one solution to a given problem.

Example of an algorithm/4

Fundamental properties

- Efficiency of an algorithm
  - `LinSearch2` is more efficient in general
  - `LinSearch1` always scans the entire array
- Correctness of an algorithm

```
Algo: LinSearch3(A,v)
i = 1;
while $i \leq n \vee A[i] \neq v$ do
  if $i > n$ then break else i = i+1;
if $i \leq n$ then return i else return NIL;
```

  - `LinSearch3` is incorrect; `LinSearch3` crashes for all inputs.

EXAMPLE OF AN ALGORITHM/5

- What about the fourth solution for searching?
- Run through the array and return the index of the value in the array.

```
Algo: LinSearch4(A,v)
for i = 1 to n do
    if A[i]==v then return i;
return NIL;
```

- OK?

EXAMPLE OF AN ALGORITHM/6

Metaphor: shopping behavior when buying a beer.

- `LinSearch1`: scan products until you get to the exit; if during the process you find a beer put it into the basket (and remove the rest).
- `LinSearch2`: scan products; stop as soon as a beer is found and go to the exit.
- `LinSearch4`: scan products; stop as soon as a beer is found and exit through next door.

Pseudo code

Use of pseudocode

- Similar in many respects to C, Pascal, Java, Python (assignments, control structures, arrays, …)
- Provides means to convey the essence of an algorithm in a concise fashion
- No fixed format for pseudocode. OK as long as it is unambiguous and breaks down the problem to the relevant basic steps.
- Pseudocode works out the important parts and abstracts the unimportant parts.

EXERCISE SL01-1/1

A prime number is a natural number greater than 1 and divisible only by 1 and itself. Write a program that determines all prime numbers between 0 and n.

Hint: Implement the sieve of Eratosthenes. Use an array of length $n + 1$ that indicates whether the number corresponding to the index is prime. Start with an array where all numbers greater than 1 are marked as primes and then proceed as follows: first eliminate the multiples of 2, then the multiples of 3, then the multiples of 4, etc.

EXERCISE SL01-1/2

SEARCHING, C SOLUTION

```
#include <stdio.h>

#define n 5

int i, v;
int a[] = { 11, 1, 4, -3, 22 };

int main() {
  i = 0;  v = -2;
  while (i < n && a[i] != v) { i++; }
  if (i < n) { printf("%d\n", i); }
  else { printf("NIL\n"); }
}
// gcc -o search search.c; ./search
```

How to approach C, etc.

- Do not study it (it is close to Java or Python and we only use a small subset).
- Whenever you meet a new construct learn and use it.
- Here:
  - `#include <stdio.h>` includes IO library with printf function
  - `#define n 5`
    defines n as a constant with value 5
  - `printf("%d\n",i)`
    prints an integer argument (`%d` is replaced by the value of i) followed by a new line (`\n`).
  - Arrays have a fixed size and start with index 0.
  - `//` is the start of a one line comment
  - `gcc -o search search.c; ./search` command to compile and run the program from the command line

SEARCHING, JAVA SOLUTION

```
import java.io.*;

class search {
  static int n = 5;
  static int i, v;
  static int a[] = { 11, 1, 4, -3, 22 };

  public static void main(String args[]) {
    i = 0; v = 22;
    while (i < n && a[i] != v) { i++; }
    if (i < n) { System.out.println(i); }
    else { System.out.println("NIL"); }
  }
}
// javac search.java; java search
```

SEARCHING, PYTHON SOLUTION

```
a = [ 11, 1, 4, -3, 22 ]
v = -3

j = -1
for i in range (len(a)):
    if a[i] == v:
        j = i
print(j)

# python search.py
```

Data structures

Informal definition

- A data structure is a way to store and organize data
- As such it facilitates access and modification of data
- Elementary data structures: lists, stacks, queues, trees

Things to consider

- No single data structure works well for all purposes
- Choice of appropriate data structure is a **crucial** issue
- Important to know strengths and limitations of a variety of data structures.

Table of contents — SL01

SORTING/1

Why sorting?

- Most fundamental problem in the study of algorithms
- Algorithms often employ sorting as a key subroutine
- Wide range of sorting algorithms, rich set of techniques

Some observations

- Sorting of arrays: does provide random access (each element can be directly accessed)
- Efficient management of space: in-place sorting
- Efficiency is generally measured in terms of
    - number of comparisons and/or
    - number of exchange operations

SORTING/2

Simple methods

- Bubble sort
- Selection sort
- Insertion sort
- Roughly n^2 comparisons

Fast methods

- Merge sort
- Heap sort
- Quick sort
- Roughly $n \log n$ comparisons

SORTING/3

Problem specification

- Sorting problem
  - Input: a sequence of n numbers $A = [a_1, a_2, \dots, a_n]$
  - Output: a permutation (reordering) $[b_1, b_2, \dots, b_n]$ of the input sequence such that $b_1 \leq b_2 \leq \cdots \leq b_n$
- Instance of sorting
  - Input sequence: $[31, 41, 59, 26, 41, 58]$
  - Output sequence: $[26, 31, 41, 41, 58, 59]$

Bubble sort / 1

Rough idea

- Scan sequence and swap unsorted adjacent elements
- Repeat the procedure until sequence is actually sorted

The algorithm

```
Algo: BubbleSort(A)
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;
```

s	o	r	t	i	n	g
o	r	s	i	n	g	t
o	r	i	n	g	s	t
o	i	n	g	r	s	t
i	n	g	o	r	s	t
i	g	n	o	r	s	t
g	i	n	o	r	s	t

BUBBLE SORT/2

Basic properties

- Number of comparisons:

$$C = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2-n}{2}$$

- The number of comparisons is independent of the original ordering and the values.

Bubble sort/3

Number of exchanges:

$$Mmin = 0$$

$$Mmax = \sum_{i=2}^{n} 3(i-1) = \frac{3n(n-1)}{2} = \frac{3n^2 - 3n}{2}$$

EXERCISE SL01-2

Modify the bubble sort algorithm such that, at each iteration of the outer loop, the smallest element of the corresponding subarray is moved to the left. Implement this variant of bubble sort and show how it works on a concrete example.

Selection sort / 1

Rough idea

- Select smallest element and swap it with 1st element
- Repeat the procedure on remaining unsorted sequence

The algorithm

```
Algo: SelectionSort(A)
for $i$ = 1 to $n$-1 do
    k = i;
    for $j$ = $i$+1 to $n$ do
        if $A[j] < A[k]$ then k = j;
    exchange A[i] and A[k];
```

s	o	r	t	i	n	g
g	o	r	t	i	n	s
g	i	r	t	o	n	s
g	i	n	t	o	r	s
g	i	n	o	t	r	s
g	i	n	o	r	t	s
g	i	n	o	r	s	t

Selection sort/2

Basic properties

- Number of comparisons:

$$C = \sum_{i=1}^{n-1} i = \frac{n^2 - n}{2}$$

- The number of comparisons is independent of the original ordering.

Selection sort/3

Basic properties

- Number of exchanges:

$$M = \sum_{i=1}^{n-1} 3 = 3(n-1) = 3n - 3$$

- Only the movement of elements is counted. Updating an index is not counted.

EXERCISE SL01-3

Modify selection sort in such a way that, at each iteration of the outer loop, it selects the largest element and swaps it with the last element of the corresponding subarray. Implement this variant of selection sort and show how it works on a concrete example.

Insertion sort / 1

Rough idea

- Take first element and consider it as (sorted) sequence
- Continue taking elements and inserting into right place

The algorithm

```
Algo: InsertionSort(A)
for $i$ = 2 to $n$ do
    j = i-1;
    t = A[i];
    while $j \geq 1 \wedge t < A[j]$ do
        A[j+1] = A[j];
        j = j-1;
    A[j+1] = t;
```

s	o	r	t	i	n	g
o	s	r	t	i	n	g
o	r	s	t	i	n	g
o	r	s	t	i	n	g
i	o	r	s	t	n	g
i	n	o	r	s	t	g
g	i	n	o	r	s	t

Insertion sort/2

Basic properties

- Number of comparisons:

$$Cmin = \sum_{i=2}^{n} 1 = n - 1$$

$$Cmax = \sum_{i=2}^{n} (i - 1) = \frac{n^2 - n}{2}$$

Insertion sort/3

Basic properties

- Number of exchanges:

$$Mmin = \sum_{i=2}^{n} 2 = 2(n-1) = 2n-2$$

$$Mmax = \sum_{i=2}^{n} (i+1) = \frac{n^2+3n-4}{2}$$

TABLE OF CONTENTS — SL01

Recursion/1

Recursive object

- It contains itself as part of it
- It is defined in terms of itself

Recursive procedure

- Procedure that calls itself
- Multiple & mutual recursive calls
- Termination condition to stop recursion

Example: simple recursion

$$\text{Factorial} \quad fact(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot fact(n-1) & \text{if } n > 0 \end{cases}$$

RECURSION/2

Evaluate the following three implementations for computing the factorial of an integer n.

Algo: fact1(n)

```
if n==0 then return 1 else return n * fact1(n-1);
```

Algo: fact2(n)

```
if n==0 then return 1;
return fact2(n-1) * n;
```

Algo: fact3(n)

```
return n * fact3(n-1);
if n==0 then return 1;
```

RECURSION/3

Tracing the call `fact1(3)`:

`fact1`(3)	6
3 * `fact1`(2)	6
2 * `fact1`(1)	2
1 * `fact1`(0)	1
`fact1`(0)	1

Recursion/4

Example: multiple recursion

$$\text{Fibonacci} \quad \mathit{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ fib(n-1) + fib(n-2) & \text{if } n > 1 \end{cases}$$

Example: mutual recursion

$$\text{Odd number} \quad odd(n) = \begin{cases} false & \text{if } n = 0 \\ even(n-1) & \text{if } n > 0 \end{cases}$$

$$\text{Even number} \quad even(n) = \begin{cases} true & \text{if } n = 0 \\ odd(n-1) & \text{if } n > 0 \end{cases}$$

EXERCISE SL01-4

Design recursive algorithms for odd and $even$.

RECURSION/5

Tracing the call `fib(4)`:

fib(4)	3
fib(3) + fib(2)	3
fib(2) + fib(1)	2
fib(1) + fib(0)	1
fib(1)	1
fib(0)	0
fib(1)	1
fib(1) + fib(0)	1
fib(1)	1
fib(0)	0

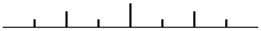
RECURSION/6

Is recursion necessary?

- Practice: recursion is elegant and in some cases the best solution by far
- Theory: one could resort to iteration and explicitly maintain a recursion stack
- In the above examples recursion is not necessary: there exist simple iterative solutions
- Recursion is more expensive than corresponding iterative solution: bookkeeping is necessary

MARKING A RULER/1

Problem description

- Print the marks of a ruler
- Example: 

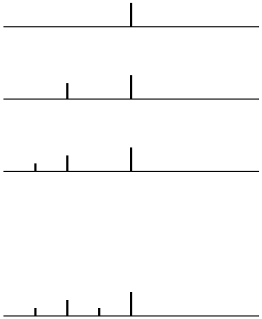
Recursive algorithm

```
Algo: Ruler(l,r,h)
m = (r+l)/2;
if $h>0$ then
    Mark(m,h);
    Ruler(l,m,h-1);
    Ruler(m,r,h-1);
```

Marking a ruler/2

Printing of marks:

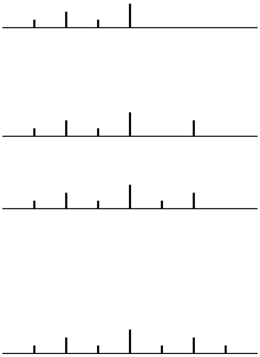
```
Ruler (0, 8, 3)
  Mark (4, 3)
  Ruler (0, 4, 2)
    Mark (2, 2)
    Ruler (0, 2, 1)
      Mark (1, 1)
      Ruler (0, 1, 0)
      Ruler (1, 2, 0)
    Ruler (2, 4, 1)
      Mark (3, 1)
      Ruler (2, 3, 0)
      Ruler (3, 4, 0)
```



MARKING A RULER/3

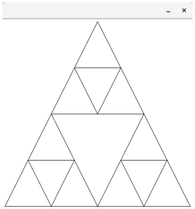
Printing of marks:

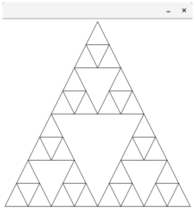
```
[ ··· ]

  Ruler (4, 8, 2)
    Mark (6, 2)
    Ruler (4, 6, 1)
      Mark (5, 1)
      Ruler (4, 5, 0)
      Ruler (5, 6, 0)
    Ruler (6, 8, 1)
      Mark (7, 1)
      Ruler (6, 7, 0)
      Ruler (7, 8, 0)
```

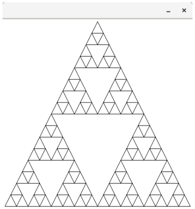


EXERCISE SL01-5/1

Consider the Sierpinski triangles of depths 2, 3, and 4 in the following pictures:





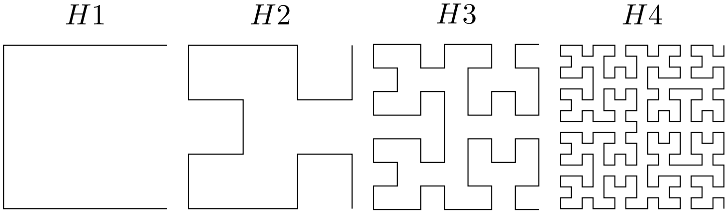


- Determine the Sierpinski triangles of depths $d = 0$ and $d = 1$.
- Design a recursive algorithm that draws a Sierpinski triangle of depth d.
- Show how to call your algorithm.

EXERCISE SL01-5/2

HILBERT CURVE/1

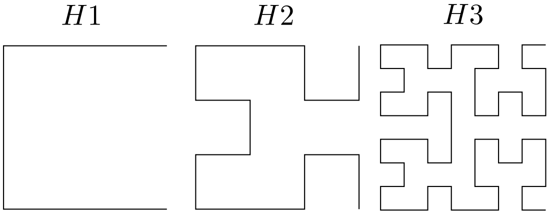
Consider the following patterns:



- Hi is the Hilbert curve of order i (named after the inventor David Hilbert).
- Goal: write a program to produce Hi.

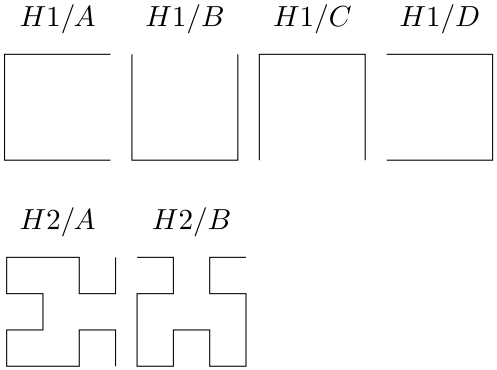
HILBERT CURVE/2

- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.



Hilbert curve/3

- Observation: Hi+1 is composed of four components of Hi.
- Solution: Determine the recursion scheme to generate Hi.



HILBERT CURVE/4

The first part of the code:

```
#include <SDL2/SDL.h>

SDL_Window *win;
SDL_Renderer *ren;

float x, y, u;

void l(float dx, float dy) {
  SDL_RenderDrawLine(ren, x, y, x+dx, y+dy);
  x=x+dx; y=y+dy;
  SDL_Delay(5);
  SDL_RenderPresent(ren);
}

...
```

HILBERT CURVE/5

The last part of the code:

```
...

int main (int argc, char** argv) {
  int d;
  sscanf(argv[1], "%d", &d);

  SDL_CreateWindowAndRenderer(400, 400, 0, &win, &ren);
  SDL_SetRenderDrawColor(ren, 255, 255, 255, 0);
  x=395; y=5; u=390/(pow(2,d)-1);
  A(d);

  SDL_Event e;
  do { SDL_PollEvent(&e); } while (e.type != SDL_QUIT);
}

// gcc hilbert.c -o hilbert -lm -lSDL2; ./hilbert 2
```

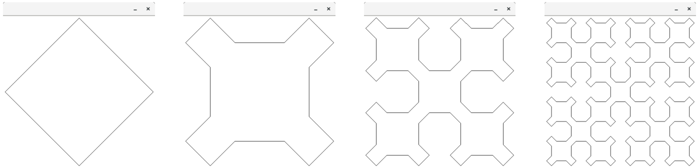
HILBERT CURVE/6

The middle part of the code:

```
...

void A(int);  void B(int);  void C(int);  void D(int);

void A(int i) {
  if (i>0) {
    B(i-1); l(-u,0);
    A(i-1); l(0,u);
    A(i-1); l(u,0);
    C(i-1);
  }
}


void B(int i) { ...

...
```

EXERCISE SL01-6/1

Consider the Sierpinski curve:



- The above pictures illustrate the Sierpinski curve of order 0, 1, 2 and 3.
- Identify the recursive pattern and sketch the code to draw the Sierpinski curve of order i.

EXERCISE SL01-6/2

SUMMARY

- Algorithmic problem:
  - An algorithm transforms input data into output data
  - Precisely specify **input** and **output**; consider special cases; work out representative examples
  - A first step is to come up with a simple plan.
  - Split or revise complex plans until they get simple.
- Sorting algorithms
  - A classical and representative algorithmic problem.
  - Initialization and conditions in loops are important.
  - Make sure you get details of loops **correct by design**. Avoid trial and error.
- Recursive algorithms
  - Recursion is an important algorithmic technique.
  - Practice the design of recursive algorithms.
  - Consider **special cases**: termination, the first couple of recursive calls.