CHAPTER 3

Functions and Algorithmic Problems

Lernziele

KdP1 Du bestimmst formale *Spezifikationen* von Algorithmen aufgrund von verbalen Beschreibungen, indem du *Signatur, Voraussetzung, Effekt und Ergebnis* angibst.

KdP2 Du implementierst gut strukturierte *Python-Programme* ausgehend von einer verbalen oder formalen Beschreibung unter adäquater Nutzung *imperativer Programmierkonzepte*.

KdP5 Du vergleichst die unterschiedlichen *Ausprägungen* der Programmierkonzepte- und paradigmen.

KdP6 Du wendest *Algorithmen* auf konkrete Eingaben an und wählst dazu *passende Darstellungen*.

Organisatorisches

- Videolänge: ∼ 154 Min.
- Tutorien: 6.+7. November
- Großplenum: 8. November, 8–10 Uhr im Hörsaal 1B
- Lernzentrum: 6.–10. November
- Abgabe der Übungsaufgaben: 10. November, 14 Uhr
- **Am 10. November, 8–10 Uhr im Hörsaal 1B beginnen wir mit der Methodenwoche. Die erste Exam Booklet Seite dürft ihr bis zum 17. November (14:00) abgeben. Wir werden in der Methodenwoche ein paar Tipps erarbeiten, wie ihr euer Exam Booklet optimieren könnt.**

Contents

Übungsaufgaben

Hinweise: Aufgabe 3 ist freiwillig und wird nicht bewertet. Von Aufgabe 1 und 2 wird eine Aufgabe ausführlich korrigiert und 10 Punkte vergeben, eine Aufgabe wird nur abgehakt und 1 Punkt vergeben. Welche der beiden Aufgaben das jeweils sein wird, wird nach dem Abgabedatum entschieden.
Plagiate werden beim 1. Mal mit 0 Punkten auf dem gesamten Zettel und beim 2. Mal mit dem Verlust der aktiven Teilnahme geahndet.
Ab sofort müssen alle Funktionen und Methoden, die ihr implementiert, nach dem 5-Schritte-Prinzip erstellt werden. Zudem muss sichtbar sein, dass die Tests auch durchgeführt wurden.
Bibliotheksfunktionen und -methoden, welche die gesamte Teilaufgabe fast unmittelbar lösen, sind verboten.

1. **Funktionen und Methoden** (10 Punkte)

   a) Benutze Binärsuche, um die Zahl 13 in der folgenden Liste von Zahlen zu finden. Zeige die einzelnen Schritte.

   `1 2 4 8 16 32 42 64 128 130 243 244 289`

   b) Betrachte die folgenden beiden Funktionen:

```python
def foo(xs, k):
    n = len(xs)
    for i in range(n):
        xs[i] = (xs[i] <= k)
    erg = False
    for b in xs:
        erg = b or erg
    return erg

def bar(xs, k):
    n = len(xs)
    erg = False
    for i in range(n):
        erg = (xs[i] <= k) or erg
    return erg
```

   Wende auf beide Funktionen jeweils das 5-Schritte-Prinzip an, d.h. bestimme Signatur, Spezifikation, Testfälle und sinnvolle Kommentare. (Der 4. Schritt, die Implementierung, existiert ja bereits.) Erläutere anhand dieses Beispiels den Begriff *Nebeneffekt* und warum dieser hier unerwünscht sein könnte. Implementiere die Funktion `bar(·)` anschließend rekursiv.

   c) Implementiere eine Python-Funktion namens `add(·)`, die zwei Listen mit Zahlen bekommt und eine Liste liefert, welche die komponentenweise Summe der

beiden Eingabelisten enthält und das Skalarprodukt der beiden Listen auf dem Bildschirm ausgibt.

d) Implementiere eine rekursive Funktion `countDigits(·)`, welche die Anzahl der Dezimalstellen einer ganzzahligen Eingabezahl liefert.

2. Methoden und Funktionen (10 Punkte)

a) Benutze Binärsuche, um die Zahl 16 in der folgenden Liste von Zahlen zu finden. Zeige die einzelnen Schritte.

```
289 244 243 130 128 64 42 32 16 8 4 2 1
```

b) Berechne ggT(2745, 1215) mit Hilfe des Euklidschen Algorithmus. Zeige die einzelnen Schritte.

c) Eine DNA-Sequenz wird in eine RNA-Sequenz umgewandelt, indem alle Vorkommen von `T` in `U` umgewandelt werden. Implementiere `dna2rna(·)` auf zwei verschiedene Weisen: Beim ersten Mal soll die Funktion die zugehörige RNA-Sequenz liefern und die Eingabesequenz soll unverändert bleiben. Beim zweiten Mal soll die Methode die Eingabesequenz verändern und keinen Ausgabewert liefern. Wie spiegelt sich dieses Verhalten in der Spezifikation wieder? Welche Vor- und Nachteile gibt es für diese beiden Vorgehen? Ihr müsst die DNA-Sequenzen als Listen von Buchstaben darstellen, z.B. `["A","T","C","A", "G"]`.

d) Implementiere eine rekursive Funktion `k_smallest(·)`, welche das k-kleinste Element einer Liste berechnet. Die Liste darf nicht sortiert werden. Hat deine Funktion einen Nebeneffekt? (Ist die Funktion nicht rekursiv, gibt es auch Punkte, aber nicht alle.)

e) Implementiere eine rekursive Funktion `quersumme(·)`, welche die Quersumme der Dezimaldarstellung einer ganzzahligen Eingabezahl berechnet.

3. Tankstellenproblem (ohne Punkte)

Wir befinden uns am Anfang einer seeeeehr langen Straße und möchten mit unserem Auto bis zum Ende der Straße fahren. Die Länge der Straße beträgt L km. Entlang dieser Straße befinden sich n Tankstellen. Für $1 \leq i \leq n$ bezeichnet l_i die Entfernung der i-ten Tankstelle zum Startpunkt (gemessen in km). Zusammengefasst gilt: $0 \leq l_1 \leq l_2 \leq \cdots \leq l_{n-1} \leq l_n \leq L$. Wir nehmen an, dass beim Startpunkt der Tank unseres Autos maximal gefüllt ist. Mit einer Tankfüllung können wir aber nicht mehr als M km fahren. Die Straße kann jedoch so lang sein, dass wir es nicht mit einer Tankfüllung bis zum Ende der Straße schaffen, sondern unter Umständen mehrere Male unterwegs tanken müssen. Als umweltbewusste Bürger·innen sind wir natürlich daran interessiert, so wenig wie möglich Tankstellen anzufahren, an denen wir tanken müssen.

a) Formuliere das Tankstellenproblem als algorithmisches Problem.

b) Entwickle einen Algorithmus zur optimalen Lösung des Tankstellenproblems und implementiere dieses in Python. Begründe warum euer Algorithmus eine optimale Lösung findet.

3.1 Fundamentals of Subroutines

Many imperative programs offer a way to define subroutines and functions. This allows for much more structured and sophisticated programs at a much larger scale. At a very basic level, a *subroutine* lets us collect a sequence of instructions into a unit that has a name. To execute this sequence of instructions in our program, we can *call* or *invoke* the subroutine by using its name. In Python, it looks like this:

```
def name():
    block
```

This defines a subroutine `name` that consists of the sequence of instructions `block`. To invoke this subroutine, we just write `name()` in our program, like we can see here:

```
def greet():
    print("Hello, nice to meet you.")
    print("How are you doing?")

x = 1
greet()
# Output on screen:
# Hello, nice to meet you.
# How are you doing?
```

A subroutine may receive *parameters*, variables that pass information to the subroutine when it is called and that can be used inside the subroutine. The parameters must be *declared* in the definition of the subroutine, and the subroutine call must conform exactly to this declaration. For example:

```
def greet(firstname, lastname):
    print("Hello " + firstname + " " + lastname + ", nice to meet you.")
    print("How are you doing?")

greet("Helmut", "Roth")
greet("Günter", "Alt")
# Output on screen:
# Hello Helmut Roth, nice to meet you.
# How are you doing?
# Hello Günter Alt, nice to meet you.
# How are you doing?
```

In this example, the `greet`-subroutine has two parameters that define the first and the last name of the person that should be greeted.

When using the subroutine, we must use exactly two parameters. For example, writing `greet("Max")` might result in an error.

In Python, it is possible to define default values for parameters. In this case, it is possible to omit the corresponding parameter in the function call. For example:

```python
def greet(firstname, lastname = "Wang"):
    print("Hello " + firstname + " " + lastname + ", nice to meet you.")
    print("How are you doing?")

greet("Helmut", "Roth")
greet("Günter")
# Output on screen:
# Hello Helmut Roth, nice to meet you.
# How are you doing?
# Hello Günter Wang, nice to meet you.
# How are you doing?
```

Now, the second function call is valid. The last name of `Günter` is taken to be `Wang`.

Selbsttest: Schaut euch das Video 023 an. Welche Subroutinen kennt ihr bereits, die ihr regelmäßig genutzt habt?

3.1.1 Functions

A subroutine may also have a return value. In this case, the subroutine is called a *function*. The value is returned using the `return`-statement. The `return`-statement also finishes the execution of the function, even if there are instructions that come after it. For example,

```python
def square(x):
  return x*x

def greet2(timeofday):
  print("Good " + timeofday + ".")
  name = input("What is your name? ")
  return name
```

A function may be used in an expression. When the expression is evaluated, the function is called and the return value is used as the value of the function in the expression. For example,

```python
z = square(5) + square(10)
name = "Person " + greet2("morning")
```

In Python, every subroutine is a function and may be used in an expression (whereas other programming languages distinguish between subroutines/procedures and functions). If there is no `return` statement, the function value is `None`. We can also use a return statement without a value. This is interpreted as returning `None`. Functions in Python that return `None` are called *methods*.

Attention: There is an essential difference between the method `print(·)` and the `return`-statement: While `print(·)` simply prints something on the screen it has return value `None`. On the other hand, the `return`-statement is used to pass the value to the invoking instance. This difference is highly relevant for the exact *specification* of functions. For example, the following function prints `42` on the screen but returns `0`.

```
def func():
    print(42)
    return 0

a = func() + 3
print(a)
# Output on the screen:
# 42
# 3
```

Selbsttest: Schaut euch das Video 024 an. Was ist der Unterschied zwischen `return 42` und `print(42)`? Gibt es Subroutinen in Python, die keinen Rückgabewert haben?

3.1.2 Scope

The *Scope* is the notion that certain identifiers in a program are valid only in a given part of the code, and that it cannot be used in another part of the program. We already discussed some details about this in Chapter 2.

A function in Python defines a *local scope*, and a variable name that is created inside the function is not visible outside the function. For example:

```
def f():
    x = 1

f()
print(x)
```

would result in an error, because the variable `x` is defined only in the local scope of the function `f`, and it is not a valid name outside the function.

Scopes can be nested. For example, the scope of the main program is called the *global scope*, and variable names that are in the global scope can also be used in functions that are defined inside the main program:

```
x = 10

def f():
    print(x)

f()
print(x)
```

It may happen that we define a variable in a local scope that has the same name as a variable in the outer scope. This defines two different variables with the same name. In the inner scope, we can access only the local variable, and once we leave the inner scope, the local variable disappears. We say that the variable in the inner scope *shadows* the variable in the outer scope:

```
x = 10

def f():
    x = 20
    print(x)

print(x)
f()
print(x)
```

In Python, we can used the `global` keyword to instruct Python not to create a new local variable but to use the variable from the global scope:

```
x = 10

def f():
    global x
    x = 20
    print(x)

print(x)
f()
print(x)
```

Selbsttest: Schaut euch das Video 025 an. Kann es zwei globale Variablen geben, die beide `x` heißen, aber unterschiedliche Objekte bezeichnen?

3.2 The 5 steps

Functions and methods are used to handle well-defined tasks, like for example computing the largest number in a list of numbers. It is used to improve the program structure. For this, it is important to specify the exact behaviour of a function and to provide comments for further details.

Hence, when implementing functions, we always have to use the following concepts that strengthen the readability of a function.

3.2.1 Signature

The *signature* provides the name of the function, the types of the input parameters, and the type of the return values.

Since Python is dynamically typed, we can always use parameters of different types, like in this example:

```python
def double(x):
    return 2 * x

print(double(4))      # outputs: 8
print(double([0,1]))  # outputs [0,1,0,1]
```

However, the is bad programming style and in languages with static type systems (like Scala, Java) not allowed. Therefore, we use comments in Python to state the signature of a function. The signature must be formulated before the function is implemented. Here are some examples:

```python
# double(int) : int

# maximum(List[int]) : int
```

The function `double(·)` shall be used to return the double of an integer value, which is itself again an integer value. Moreover, the function `maximum(·)` shall be used get a list of integers and returns an integer.

3.2.2 Specification

The *specification* is a contract between the developer and the user of the function. It is a precise description of the *preconditions* under which a function can be called and its *result* (what is the `return`-value of the function?) and *effect* (how does the function affect the global state?). The result and the effect is written in stative passive (deu.: Zustandspassiv). The specification must be formulated before the function is implemented. Here are some examples with signature and specification:

```python
# double(int) : int
# Precondition: None
# Effect: None
# Result: The double of the input number is returned.

# maximum(List[int]) : int
# Precondition: list is not empty
# Effect: The largest number of the list is printed on the screen.
# Result: The largest number of the list is returned.
def maximum(list):
    erg = list[0]
    for i in range(1, len(list)):
        erg = max(erg, list[i])
    print(erg)
    return erg
```

```python
# bar(float, float) : float
# Precondition: b is not Zero
# Effect: The input numbers are printed on the screen.
# Result: The quotient of a and b is returned.
def bar(a, b):
    print(a, b)
    return a / b
```

Note the difference between `print(·)` and `return` and how they affect the specification: The `print(·)`-function changes the state of the machine (output device) and belongs to the effect of the function. The value of the `return`-statment corresponds to the result of the function.

Attention: Whenever possible, a function should either have an effect or a result bot not both. The reason is as follows:
Functions sometimes have hidden effects in addition to just returning a value. These effects are called the *side effects* of the function evaluation. Sometimes, unintended side effects may lead to errors that are hard to find in an imperative program. The following implementation of the `minimum(·)`-function has the side effect that it changes the input list:

```python
# maximum(List[int]) : int
# Precondition: list is not empty
# Effect: list[i] contains the minimum of the elements at
#         indices 0 to i of the input list
# Result: The smallest number of the list is returned.
def minimum(list):
    m = list[0]
    for i in range(len(1,list)):
        m = min(m, list[i])
        list[i] = m
    return m
```

3.2.3 Tests

Tests are used to show the *existence of program errors*. They will never show the absence of program errors – this is only possible by using correctness proofs. In many cases the program is too complex to give a correctness proof. Therefore, it is elemental to think about test cases that cover all important and edge cases. The test cases have to be formulated after we finished the specification and before we implement the function. Here are some examples:

```python
# double(int) : int
# Precondition: None
# Effect: None
# Result: The double of the input number is returned.
```

```
''' Test cases:
double(4) == 8    # positive number
double(0) == 0    # number 0
double(-1)== -2   # negative number
'''

# maximum(List[int]) : int
# Precondition: list is not empty
# Effect: The largest number of the list is printed on the screen.
# Result: The largest number of the list is returned.
''' Test cases:
maximum([1,2,3,2,1]) == 3
maximum([1,2,3,4]) == 4
maximum([4,3,2,1]) == 4
maximum([1,1,1,1]) == 1
maximum([42]) == 42
'''
```

Observe that `maximum([])` was not a test case although it could be an important edge case. However, we omitted this case since we already excluded this case by our specification.

Selbsttest: Schaut euch das Video 026 an. Spezifiziere eine Methode, die genau dann `True` liefert, wenn ein Element in einer Liste enthalten ist und auf dem Bildschirm `"Ja"` oder `"Nein"` ausgibt.

3.2.4 Definition

Once the signature, the specification, and the test cases have been formulated, we can give the *definition* of the function, i.e., the actual implementation.

Once we have implemented the function we use the formulated test cases to find potential errors in our implementation.

3.2.5 Comments

Finally, we use natural language to write *comments* in our code – these are descriptions of the behaviour of certain parts and represent the used programming ideas. These comments are important when different people work on the same project – it is easier to understand what other people did.

Here are two full examples of how to build a function:

```
# summe(List[int]) : int
# Precondition: None
# Effect: None
# Result: The sum of the elements of the list is returned.
#         If the list is empty, 0 is returned
''' Test cases:
```

```python
summe([3,3,3,3]) == 12
summe([5]) == 5
summe([]) == 0
summe(range(1,101)) == 100*101//2
summe([9,-9]) == 0
summe([1,5,2,-4,9]) == 13
'''
def summe(list):
    # Zwischenergebnis ist anfangs 0
    erg = 0

    # Schleife, die schrittweise alle Elemente betrachtet
    # und immer das aktuelle Element auf erg draufaddiert:
    for x in list:
        erg += x

    return erg

# maximum(List[int]) : int
# Precondition: list is not empty
# Effect: The largest number of the list is printed on the screen.
# Result: The largest number of the list is returned.
''' Test cases:
maximum([1,2,3,2,1]) == 3
maximum([1,2,3,4]) == 4
maximum([4,3,2,1]) == 4
maximum([1,1,1,1]) == 1
maximum([42]) == 42
'''
def maximum(list):
    # starte mit erstem Element
    erg = list[0]

    # Schleife, die schrittweise alle Elemente betrachtet
    # und immer das aktuell größere speichert:
    for i in range(1, len(list)):
        erg = max(erg, list[i])

    return erg
```

Selbsttest: Schaut euch das Video 027 an. Durchlaufe die 5 Schritte für Funktionen, die das Minimum und das Produkt von Listenelementen berechnen.

3.3 Evaluation strategy

The parameters in the declaration of a function are called the *formal parameters*. The parameters that are used in an actual invocation of a function are called *actual parameters*. For example, in

```python
def func(a,b):
    return a + b

func(3 + 7//2, -42 + 7)
```

the function has formal parameters `a` and `b` and the actual parameters in the invocation are `3 + 7//2` and `-42 + 7`.

There are several kinds of how the formal and the actual parameters can be related to each other. We distinguish these kinds by their *evaluation order*.

3.3.1 Strict evaluation order

In the strict evaluation order, we start by evaluating the parameter of the function. Once this process is done, we invoke the function with the evaluated parameters. In the example above we would first evaluate `func(3 + 7//2, -42 + 7)` to `func(6, -35)` and then invoke the function with actual parameters `6` and `35`. The relation of these actual parameters and the formal parameters now depend on the *call by convention*. We distinguish at least two types.

Call by value. The simplest way in which the formal and the actual parameters relate is called call by value. We can imagine that when a function is called, there are assignment operations that assign the expressions that are passed as the actual parameters to the corresponding formal parameters. Therefore, it is also sometimes called *call by assignment*. The relation between the actual and the formal parameters is like after a regular assignment operation. Most modern programming languages (like Python, C, C++) use call by value.

Attention: Call by value does not mean that there is no relation between the formal and the actual parameter. For example, if we pass a mutable data object to a function in Python, changes to the data object that occur in the function are visible outside the function. This is consistent with the interpretation of the call by value convention as an assignment. As an example see the implementation of `minimum(·)` from the last Attention-Box.

Call by reference. If a parameter is passed by call by reference, the actual parameter may typically only be a variable name, and not an expression or a constant (literal). If we use call by reference, then the formal and the actual parameter become synonymous. If the formal parameter is changed in the function by an assignment statement, this is also visible outside the function.

For example, using call by reference, we could implement a swap function like this:

```python
def swap(a, b)
    temp = a
```

```
    a = b
    b = temp

x = 10
y = 20
swap(x, y)
# In Python, we would still have x == 10, y == 20,
# but if we used call by reference, we would now have
# x == 20, y == 10
```

Call by reference is supported for example in Visual Basic, C++, or Pascal.

Selbsttest: Schaut euch das Video 028 an. Implementiere eine Funktion `swap(xs, i,j)`, welche die Elemente in `xs` an den Positition `i` und `j` tauscht. Warum widerspricht das nicht dem Video? Zeichne auf, was im Speicher passiert.

3.3.2 Non-strict evaluation order

In the non-strict evaluation order a function may return a result before all of its arguments have been evaluated. We also distinguish at least two types.

Call by name. We do not evaluate the actual parameters. Instead, we may imagine that every occurrence of a formal parameter is replaced by the corresponding actual parameter, at a textual level. This is particularly relevant if the actual parameter is an expression. Then, this expression is evaluated every time when the formal parameter is used, possibly with different results.

```
def func(a):
    print("Hello")
    return a * a

def by_name(input):
    return input + input

z = by_name(func(3))    # z == 18
# Output:
# Since Python does not use call by name, the output is just:
# Hello
# If Python would use call by name it would evaluate as follows:
# by_name(func(3)) => return fun(3) + func(3) and therefore the output would be
# Hello
# Hello
```

Scala and the macros of the C preprocessor can use call by name.

Sometimes, actual parameters are never evaluated – which might save a lot of evaluation time or ignores infinite loops or undefined expressions. This behaviour is also used in boolean expressions in Python:

```
>>> True or 1/0
```

Although `1/0` is not defined (division by zero), the expression still evaluates to `True` because, whenever one boolean value is `True`, the whole boolean expression `True` as well. Hence, Python simply ignores `1/0`. However, if we would write

```
>>> 1/0 or True
```

the expression would cause an error, because `1/0` will be evaluated first.

Call by need. This call by convention is like call by name but whenever a parameter has been evaluated its value is stored for subsequent use. In the example above, the expression `by_name(func(3))` would cause only one `Hello`-output, since `func(3)` is evaluated the first time and its result is stored for the second time.

Call by need is sometimes called *Lazy evaluation* and is used in Haskell.

Selbsttest: Schaut euch das Video 029 an. Funktioniert die `swap(·)`-Methode mit Call-by-name?

3.4 Recursion

A function may call itself. This is called *recursion*. When this happens, each invocation of the function has its own local scope and its own variables.

```
def f(x):
    if x == 0:
        print(0)
    else:
        print(2*x)
    f(x-1)
        print(2*x)
```

The following function computes the factorial of n, i.e., $n! = 1 \cdot 2 \cdot \ldots \cdot n$.

```
def factorial(n)
    val = 1
    for i in range(1, n + 1):
        val = val * i
    return val
```

This approach is called *iterative*, since we use loops instead of recursion. The following function implementation also computes the factorial but uses a recursive approach.

```python
def factorial_rec(n)
    if n < 0:
        return 0  # Recursion anchor
    elif n == 0:
        return 1  # Recursion anchor
    else:
        return n * factorial_rec(n - 1) # Recursion step
```

For a recursive function, it is very important that there is at least one case in the function definition where the function does not call itself. This case must be reachable from every possible function call. These cases are called the *recursion anchor*. The other cases are called *recursion steps*.

Last, we can see two detailed examples using recursion:

```python
# summe(List[int]) : int
# Precondition: None
# Effect: None
# Result: The sum of the elements of the list is returned.
#         If the list is empty, 0 is returned
''' Test cases:
summe([3,3,3,3]) == 12
summe([5]) == 5
summe([]) == 0
summe(range(1,101)) == 100*101//2
summe([9,-9]) == 0
summe([1,5,2,-4,9]) == 13
'''
def summe(list):
    if len(list) == 0: # recursion anchor
        return 0
    else:
        return list[0] + summe(list[1:]) # recursion step

# maximum(List[int]) : int
# Precondition: list is not empty
# Effect: None
# Result: The largest number of the list is returned.
''' Test cases:
maximum([1,2,3,2,1]) == 3
maximum([1,2,3,4]) == 4
maximum([4,3,2,1]) == 4
maximum([1,1,1,1]) == 1
maximum([42]) == 42
'''
def maximum(list):
    if len(list) == 1: # recursion anchor
        return list[0]
    else:
        return max(list[0], maximum(list[1:])) # recursion step
```

Selbsttest: Schaut euch das Video 030 an. Implementiere eine rekursive Funktion, welche die Summe der ersten n Zweierpotenzen berechnet.

3.5 Algorithmic problems

An algorithmic problem is a problem that is suitable for processing with computers. It consists of a formal description of the input and a formal description of the corresponding output. Here you can see some examples:

Factorial:

Input: $n \in \mathbb{N}_0$

Output: $n!$

Product:

Input: $a, b \in \mathbb{R}$

Output: $a \cdot b$

GCD:

Input: $a, b \in \mathbb{N}_0$

Output: greatest common divisor $\gcd(a, b)$ of a and b, i.e., the largest number that is divisible by a and b (with $\gcd(0, 0) = 0$).

Single-pair-shortest-path problem:

Input: public transport of Berlin, start and target station

Output: shortest path between the start and the target station

In computer science we want to develop algorithms, that can provably solve algorithmic problems and which are provably efficient. This raises some questions: How do we argue about correctness and efficiency? What means "efficient"? Does each algorithmic problem can be solved with an algorithm? The answer for the last question is "no", but this will not be part of this lecture (see fundamentals of theoretical computer science).

3.5.1 Search problems

Search problems belong to the fundamental problems of computer science. In most cases we have some big data structures and would like to know if some given value is contained in this data structure. First, we focus on lists, but later we also consider more complex data structures like trees. For example, the campus management system has a huge list of student entries. Then given the student number we can find the position of the student in this list as well as the corresponding data.

We distinguish two types of search-problems:

Search problem:

Input: list `xs` of elements some type T, value `k`$\in T$

Output: position of the value `k` in the list `xs` (if existing)

Sorted search problem:

Input: list `xs` with ascending sorted elements of a totally ordered universe U, value `k`$\in U$

Output: position of the value `k` in the list `xs` (if existing)

The search problem is obviously more general than the sorted search problem. That means, an algorithm that computes a solution for the search problem will always compute a solution for the sorted search problem.

Selbsttest: Schaut euch das Video 031 an. Formuliere dein eigenes Algorithmisches Problem.

3.5.2 Linear search

The general search problem can easily be solved by going through the input list and check each position. This is the best approach we can do. The algorithm is called *linear search* and can be implemented using an iterative approach:

```python
# linear_search(List[T], T) : int     T is an arbitrary type
# Precondition: None
# Effect: None
# Result: The first position of k in xs is returned.
#         If xs does not contain k, the length of xs is returned.
''' Test cases
Your task: find some good test cases
'''
def linear_search(xs, k):
    n = len(xs)          # n is the length of xs
    for i in range(n):   # we check each position in the list
        if k == xs[i]:   # and stop by its first appearance
            return i
    return n             # k is not an element of xs
```

In the worst case the algorithm has to check each element, i.e., if the list has n elements the algorithm will check n positions. We say that the algorithm has running time $O(n)$, which is also called *linear running time*.[1]

Selbsttest: Schaut euch das Video 032 an. Implementiere eine lineare Suche, die gleichzeitig von links und rechts Richtung Mitte läuft und die Elemente überprüft.

[1] We will see the O-Notation in further examples. A formal definition will be given in higher semesters.

3.5.3 Binary search

Now consider the case that the input list is ordered. Of course the linear search algorithm would still find a solution. But using the fact, that the list is ordered we can develop a much better algorithm, which we call *binary search*. The idea for this algorithm is as follows: Compare the value `k` with the middle element `m` of the list. Then we have three different cases:

(i) If `k == m`, then we found the element and can output the position.

(ii) If `k < m`, then each element in the *right half* of the list must be *larger* than `k`. Hence, we can focus our search in the *left half* of the list.

(iii) If `m < k`, then each element in the *left half* of the list must be *smaller* than `k`. Hence, we can focus our search in the *right half* of the list.

We use a recursive approach for this algorithm:

```python
# binary_search(List[U], U) : int     U is an arbitrary type
# Precondition: xs is sorted in ascending order
# Effect: None
# Result: The first position of k in xs is returned.
#         If xs does not contain k, the length of xs is returned.
''' Test cases
Your task: find some good test cases
'''
def binary_search(xs, k):
    n = len(xs)

    # This function searches k in xs in the range left to right-1
    def bin_search(left, right):
        if left >= right:            # k is not in xs
            return n
        m_pos = (left + right) // 2  # index of the element in the middle
        m = xs[m_pos]                # element in the middle
        if k == m:           # k has been found
            return m_pos
        elif k < m:          # search in left half
            return bin_search(left, m_pos)
        else: # m < k        # search in right half
            return bin_search(m_pos + 1, right)

    # invocation of the help function
    return bin_search(0, n)
```

Finally, we can argue that binary search is much better than linear search by estimating the number of elements that are compared with `k`.

First, let n be the number of elements in the list `xs`. After each comparison we discard one half of the elements (either the left or the right half). That means in the beginning n elements are *relevant*, then $n/2$, then $n/4$, then $n/8$, and so on... In general, after t

steps there are $n/(2^t)$ elements relevant. The algorithm runs as long as there is at least one element relevant, after this the algorithm terminates. Thus, we can conclude:

$$\frac{n}{2^t} \geq 1 \Leftrightarrow n \geq 2^t \Leftrightarrow t \leq \log_2(n)$$

Hence, we have shown, that binary search applied to a list of n elements needs at most $\log_2(n)$ steps. We say that binary search has running time $O(\log n)$, which is also called *logarithmic time*. This is much better than linear time, since the linear function grows much faster than the logarithmic function.

Selbsttest: Schaut euch das Video 033 an. Denk dir eine eigene Liste aus und suche dort schrittweise ein Element.

3.5.4 Euclids algorithm for the GCD

In this section, we want to implement a recursive algorithm that computes the greatest common divisor of two natural numbers a and b. First, we start with a simple statement.

Lemma 1. *Let $a, b, c \in \mathbb{N}_0$ with $a \geq b$. Then we have:*

$$c \text{ divides } a \text{ and } c \text{ divides } b \Leftrightarrow c \text{ divides } b \text{ and } c \text{ divides } a - b$$

Proof. On the one hand, if c divides a and b, then there are $x, y \in \mathbb{N}_0$ such that $a = c \cdot x$ and $b = c \cdot y$. But then we can conclude $a - b = c \cdot (x - y)$ which means that c divides $a - b$.

On the other hand, if c divides b and $a - b$, then there are $x, y \in \mathbb{N}_0$ such that $b = c \cdot y$ and $a - b = c \cdot x$. But then we can conclude $a = (a - b) + b = c \cdot (x + y)$ which means that c divides a. □

This lemma works especially for $c = \gcd(a, b)$ and therefore, we know $\gcd(a, b) = \gcd(b, a - b)$. Moreover, we know that $\gcd(a, 0) = a$ for all $a \in \mathbb{N}_0$. Thus, we can present our first recursive approach:

```python
# gcd(int, int) : int
# Precondition: a>=0, b>=0
# Effect: None
# Result: the gcd of a and b is returned
''' Test cases
Your task: find some good test cases
'''
def gcd(a,b):
    if a < b:    # gcd is commutative
        return gcd(b,a)
    elif b == 0: # recursion anchor
        return a
    else:        # recursion step, a>=b, use the lemma
        return gcd(b, a-b)
```

This can still be optimized. The algorithm subtracts b from a until a number smaller than b has been reached. This can be shortened by using the modulo operation. The final result looks like this:

```python
# gcd(int, int) : int
# Precondition: a>=0, b>=0
# Effect: None
# Result: the gcd of a and b is returned
''' Test cases
Your task: find some good test cases
'''
def gcd(a,b):
    if a < b:     # gcd is commutative
        return gcd(b,a)
    elif b == 0: # recursion anchor
        return a
    else:         # recursion step, a>=b
        return gcd(b, a%b)
```

Note that this algorithm is quite fast. It uses a logarithmic number of steps, i.e., its running time is $O(\log(a \cdot b))$. We will not prove this fact.

Selbsttest: Schaut euch das Video 034 an. Berechne ggT(4711, 1234) mit dem Euklidschen Algorithmus.