Instructions for MA1

This document contains the material for the first module (M1) in the course Computer Programming II. It is about algorithm construction with recursion and algorithm analysis.

The exercises for the module can be found in the yellow frames below. There are **12 mandatory exercises** in this module. Each group of exercises follows a related theory and a set of examples. Read carefully instructions below.

The following files are associated with the module:

1. `MA1.py` : The document where exercise solutions should be written when you present them, before being uploaded to the STUDIUM.
2. For each coding exercise there is a test file named `MA1_test_`*name*`.py` where *name* is the name of the function to be tested (for example `MA1_test_largest.py` and `MA1_test_multiply.py`).
3. For the tests, the framework unittest is used. You do not need to familiarize yourself with the entire framework, it is enough to look at how the given tests are written.
4. There is tests.sh file to run all tests from terminal.

Follow the instruction below when preparing your solutions for presenting:

1. The tasks must be solved in the order they come and with the tools (classes, functions, methods, ...) which are introduced before the exercise.
2. Unless otherwise stated, you may not use other packages than those already included in the downloaded files.
3. The functions must be named and return exactly what is stated in the task.
4. In the `main` function, you can write code that tests your implementations. If you're writing top-level code (outside of all functions), you should remove it as well as any debug printouts before uploading.
5. Tasks are to be answered with text written as comments or printed text strings at the end of the code document. The answers can be kept short.
6. The solutions must first be presented orally to an assistant or a teacher and then uploaded to STUDIUM.
7. At the oral presentation, you must start with identifying yourself by showing an ID. When the presentation is done, you should fill in the first lines of the document (name, email, reviewer, date).

   Then upload make a zip file containing `MA1.py` and the test functions that you have written and upload the zip file into STUDIUM.

   If you refer to graphs which *could* be interesting in some tasks they should be in pdf, jpg or png format and also included in the zip file.

   We do not accept other formats (word, Jupyter notebook, ...) and no submissions via email.

Important note:
You may collaborate with other students, but you must write and be able to explain your own code. You may not copy code, neither from other students nor from the Internet except from the places explicitly pointed out in this lesson. Changing variable names and similar modifications does not count as writing your own code. Since the lesson tasks (MA's) are included as part of the examination, we are obliged to report failures to follow these rules.

1 Algorithm construction with recursion

Recursion is a powerful tool for finding efficient algorithms. However, it is also a way of constructing hopelessly inefficient algorithms. It is therefore important to be able to analyze the algorithms to understand whether it is good or bad.

Solving a problem recursively means expressing the solution in terms of the same type of problem, which must be simpler in some sense. You divide the problem into one or more sub-problems of the same type, solve these in the same way and combine the solutions of the sub-problems into a solution of the original problem.

We will demonstrate how this is done through a number of examples. In many of the examples there exist simple iterative algorithms or built-in functions / methods which can solve the problems but our focus here is to *practice recursive thinking*!

Example 1: Computing factorials

Suppose we want to write a function that calculates $n! = n(n-1)(n-2)\cdot\ldots\cdot 2\cdot 1$ for a natural number $n \geq 1$. We also define $0! = 1$.

A Python function for this calculation:

```
1 def fac(n):
2     p=1
3     for x in range(1,n+1):
4         p=p*x
5     return p
```

This algorithm, which fits well with the given definition, is said to be *iterative* because it uses an *iteration* (a `for` statement).

The factorial definition can also be formulated *recursively*:

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

The definition of $n!$ is recursive because it uses $(n-1)!$, i.e. the same problem which is to be solved except one step closer to the predefined base case $0! = 1$.

This can be used in programming by writing a *recursive* function:

```
1  def fac(n):
2      if n==0:
3          return 1
4      else:
5          return n*fac(n-1)
```

It is recursive because it contains a function call to itself, but with a different argument (`n-1` instead of `n`) unless the function is called with the argument 0 — a so-called *base case*.

To calculate, for example, `fac(3)`, we must first calculate `fac(2)`. To calculate `fac(2)`, we must calculate `fac(1)` which, in turn, requires `fac(0)` to be calculated. However, the value of `fac(0)` can be returned immediately.
This can be illustrated in the figure:

```
fac(3):
|   3*fac(2)
|       |   2*fac(1)
|       |       |  1*fac(0)
|       |       |     1
|       |       1
|       2
6
```

When the program enters `fac(0)`, there are three unfinished calls to `fac`, each with its own argument. Since `fac(0)` does not make a new call, it can return its value to the waiting `fac(1)`.

The first multiplication performed is the one on row 4 and the last is the one on row 2.

Note that there must always be at least one *base case* which stops the call chain.

Example 2: Calculation of x^n – first version

We shall calculate x^n, n integer ≥ 0, with repeated multiplications.

Iterative definition: $x^n = x \cdot x \cdot x \cdot x \cdot \ldots \cdot x$

Iterative function:

```
1  def power(x,n):
2      p=1
3      for i in range(1,n+1):
4          p=p*x
5      return p
```

Recursive definition:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{if } n > 0 \end{cases}$$

Recursive function:

```
1  def power(x, n):
2      if n == 0:
3          return 1
4      else:
5          return x*power(x, n-1)
```

Include negative n:

```
def power(x, n):
    if n == 0:
        return 1
    elif n > 0:
        return x*power(x, n-1)
    else:
        return 1./power(x, -n)
```

In the chapter on algorithm analysis, we will present a significantly more efficient way of doing the calculation.

Exercise 1: The function `multiply`

Write a recursive function (i.e. without iteration) `multiply(m, n)` where `m` and `n` are two non-negative integers. The function should return the product `m * n` and the calculation should be done *without* using multiplication. The product must therefore be calculated with *additions*.

If you wanted to minimize the number of recursive calls, how would you modify the code?

Exercise 2: The harmonic sum.

Write a recursive function `harmonic(n)` which calculates the harmonic sum $h(n)$ defined as

$$h(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$$

where `n` is a positive integer.

Exercise 3: Binary representation of an integer.

Write a recursive function `get_binary (x)` that returns the binary representation of the integer `x` as a string.

Example:

`get_binary(0)` should return '0'
`get_binary(1)` should return '1'
`get_binary(5)` should return '101'
`get_binary(15)` should return '1111'
`get_binary(-18)` should return '-10010'

Hint: Use *two* base cases $x == 0$ and $x == 1$ respectively!

2 Efficient recurrent algorithms

In the examples above, the recursive methods are neither simpler nor more efficient than the iterative ones. However, we will later see that recursive reasoning is a powerful way to solve problems and find efficient algorithms.

Expressing oneself recursively is often natural in mathematics. The differential rules ("the derivative of a sum is the sum of the derivatives") are examples of this. Nevertheless, recursion is considered difficult at the beginning of learning programming.

Questions to be answered when constructing a recursive algorithm:

1. How can I divide the problem into smaller problems of the same sort?
2. How do I combine the solutions to the sub-problems into a solution for the original problem?
3. Which recursion-terminating cases are appropriate? Does the function always reach a base case independent of input?

There is a strong resemblance to an induction proof in mathematics. In such proofs we assume that a theorem holds for a certain value n and, if so, we show that it also must hold for $n - 1$.

Here, we assume that we can solve the problem for a smaller problem and then use that solution to solve it for a larger problem.

Example 3: Reverse a string

Suppose we want to write a function that prints a string where the characters come in reverse order.

How can we define the problem in terms of smaller "reverse-problems"?

> Let n be the number of characters in the string.
>
> Suppose that we can solve the problem for the $n - 1$ first characters in the string.
>
> To solve the problem for a string of length 1 is trivial.

This reasoning gives us the following code:

```
1  def reverse_string(s):
2    if len(s)<=1:
3        print(s, end='')
4    else:
5        print(s[-1])      # Print the last character
6        reverse_string(s[:-1])  # All but the last character in reverse order
```

Example 4: Stack tiles

Given a pile f of n tiles. The tiles are all of different sizes and they are stacked in order of size with the largest at the bottom.

The problem is to move the whole pile to another location (t) following the rules:

1. Only one tile may be moved at a time.
2. A larger tile must never be placed on a smaller one.

To be able to do this we need another place (h) that can be used as an intermediate pile.

We can now do the operation as follows:

1. Move the $n-1$ top tiles on f to h using t as an intermediate pile.
2. Move the remaining one on f to t.
3. Move the remaining $n-1$ tiles on h to t using f as an intermediate pile.

The problem is thus solved recursively by solving two problems of size $n-1$.

We can use either $n = 1$ or $n = 0$ as base case. (Using $n = 0$ gives simpler code in the exercise below!)

Exercise 4: Alternative `reverse_string`.

Write a **recursive** function `reverse_string(s)` that *returns a string* (i.e. does not print it) where the characters come in reverse order from the characters in `s`.

Requirements: Use the assumption that we can solve the problem for the $n-1$ *last* characters in the string. Do not modify input string.

Note: Returning the result instead of printing it is of course a much better design — after all, it's the caller who has to decide what to do with the result!

We do not need to split the problem at the first or last character — we can split anywhere, e.g. in the middle:

```
1 def reverse_string(x):
2     if len(x) <= 1:
3         return x
4     else:
5         mid = len(x)//2
6         return reverse_string(x[mid:]) +
          ↪  reverse_string(x[:mid])
```

On line 6, we see that the function calls itself twice. The function is then said to be *tree recursive*. In this case, it does not matter which division we choose, but in the discussion of algorithm analysis we will see that the choice of sub-problems can be crucial for the efficiency of the algorithm.

Exercise 5: The function `largest`.

Write a recursive function `largest(a)` which returns the largest value of the elements in the list `a`. You can assume that the list is not empty and that it only contains comparable elements (e.g. all numbers or all strings).

Requirements: Do not modify input list. You must not use the `max` method in this task. The complexity of your method should be less than exponential.

Exercise 6: The function `count`.

Write a recursive function `count(x, s)` which counts and returns how many times `x` is at the highest level in the list `s`.

Example: The call `count(4, [1, 4, 2, ['a', [[4], 3, 4]]])` should return 1. Then modify the code to count instances at *all* levels. The call above should then return 3.

Requirements: Make sure that lists are not destroyed. Searching of non-existing elements should return 0.

Hint: The expression `type(x) == list` returns `True` if x is a list, otherwise `False`.

The following points should be checked:

1. That searching in an empty list returns 0.
2. That the first and last elements in the list are also checked.
3. That it is possible to search for lists.
4. That multi-level sublists are searched.
5. That searching for non-existent elements returns 0.
6. That the searched list is not destroyed.

Exercise 7: Moving tiles.

Write a recursive function `def bricklek(from, to, help, n)` which returns a list of instructions on how to move the tiles. The parameters `from`, `to` and `help` are strings that identify the different piles and `n` is the number of tiles.

The call `bricklek('f', 't', 'h', 2)` should return the list

`['f->h', 'f->t', 'h->t']`

which should be read as

"*move from `f` to `h`, move from `f` to `t`, move from `h` to `t`*".

Each element in the list is thus a string that indicates from which pile and to which pile tiles should be moved. Since it is always the top tile that is to be moved, no other information is needed.

It is important that the strings look exactly as in the example and do not contain any blank spaces!

Hint: The function does not need more than 5 lines, including the `def` line!

Recursion depth

In some of the above and future tasks and examples you may have problems with the so-called *recursion depth*, e.g. `largest`. This means you run out of memory which is reserved for handling function calls. In general, using recursion over very long lists can be problematic. We point out again our aim has been to practice recursive thinking!

3 Algorithm analysis

In the algorithm analysis we study how the run time for an algorithm depends on the problem size. Traditionally, this is called the algorithm's *complexity*, which is is an expression of the efficiency of the algorithm. It has nothing to do with how difficult the algorithm is to understand or implement.

The results are of the type "the time for this algorithm grows proportionally to the square of the problem size".

Instead of using time, we can talk about how many times a certain operation is performed.

Example 5: Computing x^n revisited

In the chapter on recursion, we presented two different ways to calculate x^n.

In the first one we used an iteration:

```
1  def power(x,n):
2      p=1
3      for i in range(n):
4          p=p*x
5      return p
```

This function obviously performs n multiplications. Assuming that all multiplications take the same amount of time regardless of the value of x gives that we can expect the time for this function to grow linearly by n. We then say that *the complexity of the function* is $\mathcal{O}(n)$ (pronounced " Big Oh ")) or $\Theta(n)$ (pronounced " Theta "). We will return to the exact definitions of these terms later.

The other one used a recursive function:

```
1  def fac(n):
2      if n==0:
3          return 1
4      else:
5          return n*fac(n-1)
```

This function will make n function calls and n multiplications so this too has the complexity $\Theta(n)$.

Another possible definition is:

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ (x^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ even} \\ x \cdot (x^{n/2})^2 & \text{if } n > 0 \text{ and } n \text{ odd} \end{cases} \tag{1}$$

This definition is also recursive, but instead of using the solution of a problem of size $n-1$ it uses a problem of size $n/2$, i.e. a much smaller problem.

```
def power(x, n):
    if (n == 0):
        return 1.0
    else:
        p = power(x, n//2)  # Integer division
        if (n % 2 == 0):    # n even
            return  p*p
        else:               # n odd
            return  x*p*p

```

The call `power (x, 1000)` will generate the following sequence of calls:

```
power(x,1000) -> power(x,500) -> power(x,250) -> power(x,125) ->
power(x,62)   -> power(x,31)  -> power(x,15)  -> power(x,7)   ->
power(x,3)    -> power(x,1)   -> power(x,0)
```

Here, the result will be calculated through 10 function calls and and 15 multiplications (10 squares and 5 extra multiplications when the argument is odd). This is a significant improvement over the two first implementations

Since the problem is halved each recursion step, there will generally be approximately $\log_2(n)$ calls (in fact $\lfloor \log_2(n) \rfloor + 1$ but we make it a little easier for us) and the same number of squares plus at most as many extra multiplications to handle odd arguments. The time should therefore grow in proportion to $\log(n)$. It is then said that the algorithm has the complexity $\Theta(\log(n))$

Asymptotic notation

The Ordo and Theta expressions are used to describe the *characteristic* time dependence without calculating an exact mathematical expression.

A (mathematical) function $t(n)$ is $\mathcal{O}(f(n))$ if there exist two constants c and n_0 such that:

$$|t(n)| \leq c \cdot |f(n)| \qquad \forall n > n_0 \tag{2}$$

If we apply this definition to the examples with the number of multiplications above, $|t(n)|$ denotes the number of multiplications that need to be performed. If we use one of the first two definitions, $c = 1$ and $f(n) = n$. With definition (1), $f(n) = \log_2(n)$ and $c = 2$. In all cases, $n_0 = 0$.

In these examples, $\mathcal{O}$ is used to express an upper limit on how many times a central operation (in this case multiplication) is used. The notation can also be used to give an upper limit on the run time or how much memory is taken up by the algorithm. The run time for an algorithm normally grows linearly with the number of operations but the value of c also depends on factors such as which programming language the algorithm is implemented in and which computer the algorithm is run on.

Since $\mathcal{O}$ denotes an *upper* limit, an algorithm that is $\mathcal{O}(\log(n))$ is also $\mathcal{O}(n)$, $\mathcal{O}(n^2)$, $\mathcal{O}(2^n)$ and so on.

We use Ω to specify a lower limit: A function $t(n)$ is $\Omega(f(n))$ if there are two constants c and n_0 such that:

$$|t(n)| \geq c \cdot |f(n)| \qquad \forall n > n_0 \tag{3}$$

If the function $t(n)$ is both $\mathcal{O}(f(n))$ and $\Omega(f(n))$ it is said to be $\Theta(f(n))$. (Of course we have to use different constants c for the upper and lower limits.)

Note that people often say $\mathcal{O}$ when they really mean Θ!

Note that when we use logarithms in $\mathcal{O}$-, Θ- and Ω- expressions we do not have to specify the base because

$$\log_a(x) = \frac{1}{\log_b(a)} \cdot \log_b(x)$$

and $\frac{1}{\log_b(a)}$ can be baked into the constant c.

Example 6: Indexing in Python lists

Suppose you have a list and want to access an element in a given position (first, nineteenth, last, ...). This is an example of when the time required does not grow with the problem size. It will take the same number of operations to retrieve the item no matter how long the list is. Thus, indexing in lists is $\Theta(1)$ (or $\mathcal{O}(1)$ which is the same thing).

Note that other programming languages may have other ways of implementing lists which may make indexing a $\Theta(n)$-operation.

Example 7: Searching in an unsorted list

When it comes to searching, we usually differ between a *successful* and an *unsuccessful* search.

If the value we are looking for is not in the list, the search will fail. However, we have to check all values to discover that so the number of operations grows linearly with the problem size (i.e. the length of the list). A failed search in an unsorted list is a $\Theta(n)$-operation.

If the searched value is in the list, the search will be successful. Here it is relevant to distinguish between best, worst and average case. In *worst case* we will find the value at the very end of the list and the work is thus proportional to the length of the list.

Assuming that all values are equally likely to be searched, we need on *average* check half of the elements. Thus, the work is also proportional to the length of the list however with a smaller constant than in the worst case.

In the *best case* we find the value in the first place. Then a constant number of operations is needed — it does not matter how long the list is.

A successful search in an unsorted list is thus:

- $\Theta(1)$ best case,
- $\Theta(n)$ worst,
- $\Theta(n)$ average case.

Usually, we are interested in worst and average case.

Note that the codes on lines 7 and 10 in the adjacent examples both have the complexity $\Theta(n)$ (where n is the length of the list) but the one in row 10 is both faster and more elegant!

The fact that an operation is built into the language does not change the complexity, even though it often significantly affects the time of the operation.

```
1  def search(x, lst)
2      for i = range(len(lst))
3          if x == lst[i]:
4              return True
5      return False
6
7  if search(x, mylist):
8      do_something()
9
10 if x in mylist:
11     do_something()
12
```

Example 8: Searching in a sorted list

If the list to be searched is sorted, the amount of work can be considerably decreased.

In this example, we call the list `a` and the sought value `v`. We assume that the elements in `a` are sorted in ascending order, i.e. so that `a[i+1]` $\geq$ `a[i]` for all indexes `i`.

A search then goes like this:

1. If the list is empty, the search has failed. `v` cannot possibly exist in an empty list.
2. If `a[n//2] == v` we are done.
3. Otherwise, repeat the procedure, with a list half as long:
   - If `a[n//2]<v`, search `a[n//2+1 : n]` (the part to the right of the middle element).
   - If `a[n//2]>v`, search `a[0 : n//2]` (the part to the left of the middle element).

In this way, we halve the problem size in each step. Compare how you yourself would search a list of ordered elements, e.g. a telephone catalogue, a class attendance list or the index of a book!

In the case of a failed search, we have to continue until the list is empty. If the length of `a` would be doubled, the number of operations needed increases by 1. In other words, the amount of work increases logarithmically: the algorithm is $\Theta(\log(n))$.

In the case of a successful search, we perform the same procedure but will on average find `v` half way down the recursive call chain of the worst case scenario. The amount of work thus technically increases as $0.5 \cdot \log(n)$ but 0.5 is only a constant and can be absorbed into c. The algorithm is thus $\Theta(\log(n))$ on average.

The question is then how many times the the length n can be halved:

$$\frac{n}{2^k} = 1$$

which gives $k = \lfloor \log_2(n) \rfloor$

Example 9: Hash tables, dictionaries

By sorting the items in order of size, we saw that it was possible to reduce the complexity of searching from $\Theta(n)$ to $\Theta(\log n)$ which is a dramatic improvement for large values of n. However, we can do even better using so-called *hash tables*. Python uses this method to implement dictionaries. In such, the time of the search operation is independent of the size of the table making it a $\mathcal{O}(1)$ operation.

We will return to hash tables in a later section which deals with data structures.

Example 10: Selection sort

The adjacent code implements a simple sorting algorithm called *selection sort*.
(Note the nice *Pythonian* way to exchange contents between two variables on line 7!)

```
1  def selection_sort(a):
2      for i in range(len(a)-1):
3          k = i
4          for j in range(i+1, len(a)):
5              if a[j] < a[k]:
6                  k = j
7          a[i], a[k] = a[k], a[i]
8
```

If we denote the length of `a` by n, we see that the first `for` statement executes $n-1$ times with the indices $i = 0, 1, 2, \ldots, n-2$. The inner loop executes $n-1$ times the first time, $n-2$ times the second time, and so on. The comparison on row 5 will thus be performed

$$(n-1) + (n-2) + (n-3) + \cdots + 2 + 1 = \frac{n(n-1)}{2}$$

which is $\Theta(n^2)$ (Exercise: Prove this!).

We consider the time for the code on lines 5 - 6 to be constant, which makes this a typical $\Theta(n^2)$ -metod.

Example 11: Analysis of the tile game

Here we solve a problem of size n using two problems of size $n-1$ plus one tile movement. If we let $t(n)$ denote for the number of tile movements, we get

$$t(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 + 2 \cdot t(n-1) & \text{if } n > 1 \end{cases}$$

This difference equation can be solved by "telescoping", i.e. we apply the equation over and over again:

$$\begin{aligned} t(n) &= 1 + 2 \cdot t(n-1) = 1 + 2 \cdot [1 + 2 \cdot t(n-2)] \\ &= 1 + 2 + 4 \cdot t(n-2) = 1 + 2 + 4 \cdot [1 + 2 \cdot t(n-3)] \ldots \\ &= 1 + 2 + 4 + \ldots + 2^k \cdot t(n-k) \\ &= \sum_{i=0}^{n-1} 2^i = 2^n - 1 \end{aligned}$$

Example 12: The Fibonacci numbers

The Fibonacci numbers are a sequence of numbers starting with 0 and 1 and then each number is the sum of the two immediately preceding numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21 ...

This function computes the n:th number (we number them starting with 0):

```
1 def fib(n):
2     if n==0:
3         return 0
4     elif n==1:
5         return 1
6     else:
7         return fib(n-1) + fib(n-2)
```

This algorithm is a little harder to analyze.

Instead of looking at *time*, we examine how *many additions* the call `fib(n)` will generate. If n is 0 or 1, no additions are made. If n is greater than 1 then we will make 1 addition *plus* the number of additions made by the calls `fib(n-1)` and `fib(n-2)`.

If we let $t(n)$ denote the total number of additions made by the call `fib(n)` we thus get:

$$t(n) = \begin{cases} 0 & \text{if } n \leq 1, \\ 1 + t(n-1) + t(n-2) & \text{if } n > 1. \end{cases} \qquad (4)$$

Note the similarity with the Fibonacci number!

This is a linear difference equation. We will give the solution below but we start by making a simpler estimate.

Since $t(n)$ must be a growing function, we can estimate an upper bound:

$$t(n) = 1 + t(n-1) + t(n-2) < 1 + 2 \cdot t(n-1)$$

We solved this as an *equation* when analyzing the tile game above and found that the solution was $2^n - 1$. Since this is a *upper limit*, we can conclude that the number of additions (and thus also the time) is $\mathcal{O}(2^n)$.

We could find a lower bound in a similar way:

$$t(n) = 1 + t(n-1) + t(n-2) > 1 + 2 \cdot t(n-2) \tag{5}$$

which will give us that $t(n)$ is $\Omega((\sqrt{2})^n)$. Thus the number of additions is growing faster than $(1.4142)^n$.

The difference equation (4) can be solved exactly and gives $t(n) \approx 1.618^n$ We give the proof here but it is not included in the course.

Proof. The characteristic equation of the homogeneous equation is

$$r^2 - r - 1 = 0,$$

which has the solution

$$r_{1,2} = \frac{1 \pm \sqrt{5}}{2},$$

The homogeneous equation has the solution

$$F(n) = ar_1^n + br_2^n,$$

where a and b are determined by the initial conditions. Since

$$t(n) = -1$$

is a particular solution, the general solution can be written

$$t(n) = ar_1^n + br_2^n - 1.$$

Using the initial conditions we get

$$\begin{aligned} a =& (1 - r_2)/(r_1 - r_2) \\ b =& -(1 - r_1)/(r_1 - r_2) \end{aligned}$$

Since $r_1 \approx 1.618$ and $r_2 \approx 0.618 < 1$ we see that

$$t(n) \approx 1.618^n$$

for large n.

□

The program is *tree recursive*, i.e. each call results in two new calls. This can potentially result in unreasonable execution times.

The problem with the code is that we will solve the same problem many times (for space reasons we write f instead of `fib`):

$$\begin{aligned} f(5) &= f(4) + f(3) \\ &= (f(3) + f(2)) + (f(2) + f(1)) \\ &= ((f(2) + f(1)) + (f(1) + f(0)) + ((f(1) + f(0)) + 1 \\ &= f(1) + f(0) + 1 + 1 + 0 + 1 + 0 + 1 \\ &= 1 + 0 + 1 + 1 + 0 + 1 + 0 + 1 = 5 \end{aligned}$$

We see that f(4) is called 1 time, f(3) is called 2 times, f(2) is called 3 times, f 1) is called 5 times and f(0) 3 times.

The usual way to calculate Fibonacci numbers is instead to start from the bottom and successively calculate f(0), f(1), ... f(n) which is then done with $\Theta(n)$ additions.

Here, however, we will take the opportunity to introduce another technique called *memoization* (note the spelling – it is **not** *memorization)* or, sometimes, *dynamic programming*. The problem with the recursive solution is that we solve identical problems many times. Memoization means that when we have solved a certain sub-problem, we save this solution. When we enter the function we first check if the solution is stored and, if so, use that one instead of doing the recursive calls.

For the Fibonacci number, the problem is unambiguously characterized by the parameter n. We can then use a dictionary to save the solutions:

```
1 memory = {0:0, 1:1}
2
3 def fib_mem(n):
4     if n not in memory:
5         memory[n] = fib_mem(n-1) + fib_mem(n-2)
6     return memory[n]
```

The function `fib_mem` above uses a *global* variable `memory` to keep track of calculated values. It would be nicer if this dictionary could be hidden inside the function. In general, you should avoid having global variables unless they mean something actually global to the whole program. The variable `memory` only has meaning for the function `fib_mem` so we really would like to hide it. However, just moving the statement `memory = {0: 0, 1: 1 }` into the function does not work since then each instance of the function will create its own `memory`. Already calculated values will be stored in *different* dictionaries (though all with the name `memory`).

The following code solves the problem:

```
def fib(n):
    memory = {0:0, 1:1}

    def fib_mem2(n):
        if n not in memory:
            memory[n] = fib_mem2(n-1) + fib_mem2(n-2)
        return memory[n]

    return fib_mem2(n)
```

We have gone back to the name `fib` for the function to be used. Inside it we create both the dictionary `memory` and the help function `fib_mem`. Both of these are local in `fib` but all instances of `fib_mem` uses the same `memory`.

This significant improvement of the original `fib` can thus be made completely "invisible" to the calling code.

More about variables *scope* can be found in Corey Schafer's YouTube-lesson.

Memoization technique for computing Fibonacci numbers is for example described in Socraticas YouTube-lesson.

An alternative way to solve the problem is to send `memory` as a parameter:

```
def fib(n, memory = None):
    if memory is None:
        memory = {0: 0, 1: 1}
    if n not in memory:
        memory[n] = fib(n-1, memory) + fib(n-2, memory)
    return memory[n]
```

If you call without entering `memory`, then the dictionary will be created and then sent with the recursive calls.

The technique of writing functions locally in functions we will use several times in the course.

As we will see in the section on sorting below, tree recursive algorithms do not necessarily provide unreasonable execution times. In fact, many classic, efficient algorithms (such as sorting algorithms and fast Fourier transform) are tree recursive.

Exercise 8: Time for the tile game.
Suppose you have a stack of 50 tiles and it takes 1 second to move one tile. How long will it take you to complete the entire transfer by the rules of the tile game? Reply with appropriate units!

Exercise 9: Run time of the given Fibonacci function.

a) Verify by test runs that the time for the given Fibonacci algorithm grows as $\Theta(1.618^n)$

b) Find out how long the calls `fib(50)` and `fib(100)` take (would take) on your computer. Reply with appropriate units! (Seconds are not a suitable unit if it takes several hours. Hours are not a suitable unit if it takes several days or years.)

Note: You should not optimize the code but use it as it is given!

Tip 1: To measure time in a Python program, you can use the function `perf_counter` in the module `time`.

Example:

```
import time
tstart = time.perf_counter()
   # code to be timed
tstop = time.perf_counter()
print(f"Measured time: {tstop-tstart} seconds")
```

Tip 2: You will not be able to run `fib(100)` but you must estimate the time using some computations!

Exercise 10: `fib_mem(100)`

What is the Fibonacci number 100 and how long did it take to calculate it using the `fib_mem` method?

4 Sorting

Here we present and analyze two recursive sorting algorithms.

Insertion sort

The common, simple insertion sort can be expressed recursively as follows:

Induction assumption: We can sort $n-1$ element.

Base case: We can sort one element.

Recursion step: Insert the n:th element among the already sorted $n-1$ elements so that the sorting is maintained.

```
def insertion_sort(l, n):
    if n > 1:
        insertion_sort( l, n-1 )     # sort the n-1 first
        x = l[n-1]
        i = n-2              # move elements larger than x to the right
        while i>=0 and l[i]>x :
            l[i+1] = l[i]
            i=i-1
        l[i+1] = x           # put in x
```

See a visualisation on YouTube!

Analysis

The time to sort n numbers with this algorithm depends on how the numbers are permuted.

In the *best case*, the elements are already sorted. This means that the `while` - loop is never executed and there will be a total of $n-1$ calls which take a constant time, i.e. $\Theta(n)$.

In the *worst case*, the numbers are sorted in reverse order. Each call will then also contain an iteration which is made $n-1$ times, i.e. in total

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)n}{2}$$

times i.e. $\Theta(n^2)$

On *average*, each element "travels" half way instead of the whole way as in the worst case. The sum then becomes $(n-1)n/4$ which also is $\Theta(n^2)$.

Merge sort

The next algorithm is called *merge sort* (eller samsortering på svenska).This algorithm is, as the analysis and exercise below will show, significantly more efficient than insertion sort.

Induction Assumption: We can sort $n/2$ elements.

Recursion step: Divide the quantity into two parts with $n/2$ elements each, sort these separately and then merge the two sorted the parts.

Base cases: A set with 0 or 1 elements is sorted.

The adjacent code is intended to illustrate *the idea* in this sorting method.

It *is not* a good implementation for large amounts of data.

One problem is that the `merge` help method recurs over the length of the list which causes a workspace (" stack ") to run out of memory for large amounts of data.

```
1  def merge_sort(l):
2      if len(l) <= 1:
3          return l.copy()
4      else:
5          n = len(l)//2
6          l1 = merge_sort(l[:n])
7          l2 = merge_sort(l[n:])
8          return merge(l1, l2)
9
10 def merge(l1, l2):
11     if len(l1) == 0:
12         return l2
13     elif len(l2) == 0:
14         return l1
15     elif l1[0] <= l2[0]:
16         return [l1[0]] + merge(l1[1:], l2)
17     else:
18         return [l2[0]] + merge(l1, l2[1:])
19
20
21
```

See a visualisation on YouTube!

The work of joining the two sorted parts (the help function `merge` above) is proportional to the number of elements. (The expression " the time is $\Theta(f(n))$ " can, loosely expressed, be interpreted as time being proportional to $f(n)$ i.e. $t(n)=c\cdot f(n)$.)

Let $t(n)$ denote the time to sort n elements. Then

$$t(n) = \begin{cases} c & \text{if } n = 0, \\ 2 \cdot t(n/2) + dn & \text{if } n > 0. \end{cases}$$

If n is an even power of 2, i.e. $n = 2^k$ then

$$\begin{aligned} t(n) =& 2t(n/2) + dn = 2(2t(n/4) + dn/2) + dn \\ =& 4t(n/4) + dn + dn \ldots \\ =& 2^k t(n/2^k) + dnk \\ =& nt(1) + dn \log n \end{aligned}$$

Thus, the time is $\Theta(n \log n)$.

Exercise 11: Comparison of sorting methods.
Assume that insertion sort and merge sort take the same amount of time for 1000 random numbers – say 1 second. How long does it take for each algorithm to sort 10^6 and 10^9 random numbers respectively? Answer with appropriate units!

As we have seen in a couple of different examples above, the time required for algorithms can quickly become unreasonably large if you choose the wrong algorithm (or if there is no efficient algorithm available). We want algorithms for which the time required grows slowly. The time to solve a small problem is often negligible; it is when the problem size grows that it starts to become important to have an efficient algorithm!

However, as the following exercise shows one should not be blinded by the complexity. In practice, for example, an algorithm that is $\Theta(n \cdot \log(n))$ can be as efficient as an algorithm that is $\Theta(n)$, if the constant (c) is much larger for the latter.

Exercise 12: Theoretical comparison of $\Theta(n)$ and $\Theta(n \cdot \log n)$.
Suppose you can choose between two algorithms, A and B, to solve a problem. We let n denote the number of elements in the data structure on which the algorithms operate. You know that algorithm A solves a problem of size n in n seconds. The time required for algorithm B is $c \cdot n \cdot \log(n)$ seconds, where c is a constant. You run a test of algorithm B on your computer and find that it takes 1 second to solve a problem when $n = 10$. How big must n be for algorithm A to take less time than algorithm B?

Conclusions

Exponential growth: Hopeless to solve big problems!

How to know if a recursive solution is good or not?

If the solution to a problem is based on solutions to two or more problems that are almost as big as the original problem, we will get an exponential growth. The naïve recursive computation of Fibonacci numbers as well as the tile game are examples of such algorithms.

If the sub-problems are significantly smaller than the original problem (e.g. half as large) it can be an efficient algorithm. The squaring algorithm to calculate x^n as well as the merge sort algorithm are examples of that.

The sorting examples showed that the algorithm with *two* subproblems of size $n/2$ was significantly better than the one based of *one* subproblem of size $n - 1$.

Another observation is that a $\log n$ dependency is *much* better than a n dependency but that $\log n$ is not much worse than constant.