Beginselen van Progammeren
Exercise Session 7: Recursion

Ex 1: Number calculations

a. Factorial

Write a function `factorial(n)` that calculates the factorial ($n! = 1 \times 2 \times 3 \times \ldots \times n$) using recursion. Do not use iteration.

b. Power

Write a function `power(x, n)` that calculates x^n given x and n using only the $*$ operator and recursion. Consider n to be a positive integer. Think about the base case and make sure the function also works for $n = 0$. Do not use iteration.

c. Factors

Write a function `factors(n)` that, for a positive integer n, calculates the prime factorization. For example for $n = 12$, the result of the function should be [2, 2, 3]. For $n = 49$, the result should be [7, 7]. When $n = 1$, the result should be an empty list, and when n is a prime, the result should be a list containing only that prime.

```
factors(1)
# []
factors(3)
# [3]
factors(4)
# [2, 2]
factors(12)
# [2, 2, 3]
factors(27)
# [3, 3, 3]
factors(9699690)
# [2, 3, 5, 7, 11, 13, 17, 19]
```

Ex 2: List calculations

a. Maximum

Write a function `maximum(list)` that, given an unsorted list of integers, finds and returns the largest number included in that list. Do not use iteration or any of the built-in functions except `len()`.

b. Sum

Write a function `sum_list(list)` that, given a list of integers, calculates the sum of that list using recursion. Do not use iteration or any of the built-in functions except `len()`.

```
sum_list([])
# 0
sum_list([1, 2, 3])
# 6
```

Ex 3: String calculations

a. Reverse of a string (P11.5)

Write a function `reverse(string)` that returns the reverse of a string. Do not use iteration, use recursion. Keep in mind how a string responds to the `-1` index.

```
reverse("")
# ""
reverse("Hello World")
# "dlroW olleH"
reverse("tacocat")
# "tacocat"
```

b. Look for the presence of a substring (P11.6)

Write a function `substring(sub, string)` that looks for a substring in a string. Disregard capitalization. Do not use iteration or any built-in functions except `lower()` and `len()`.

```
substring("boards", "How many boards could the Mongols hoard if the Mongol hordes got bored")
# True
substring("Mongol hordes", "How many boards could the Mongols hoard if the Mongol hordes got bored")
# True
substring("Bored", "How many boards could the Mongols hoard if the Mongol hordes got bored")
# True
substring("Dinner plate", "How many boards could the Mongols hoard if the Mongol hordes got bored")
# False
```

c. Index of a substring (P11.7)

Adjust the previous function to a new function `index(sub, string)` that returns the position of the first occurrence of the substring, again disregarding capitalization. If the substring is not found, return `-1`.

```
index("can", "Can you can a can as a canner can can a can?")
# 0
index("a can", "Can you can a can as a canner can can a can?")
# 12
index("you", "Can you can a can as a canner can can a can?")
# 4
index("tin", "Can you can a can as a canner can can a can?")
# -1
```

For this exercise you want to use either a helper function or add a third argument to the `index` function with a default value.

Ex 4: Mystery code

Without running the code on your computer, figure out what the following function accomplishes:

```
def mystery(a, b):
    if b == 0:
        return 0
    elif b % 2 == 0:
        return mystery(a + a, b / 2)
    else:
        return mystery(a + a, b // 2) + a
```

What do `mystery(14, 3)`, `mystery(3, 14)` and `mystery(2, 21)` return?

Ex 5: Ruler

Implement a function `ruler(n)` that outputs a sort of ruler with the following pattern:

```
>>> ruler(2)
--
-
--
-
--
>>> ruler(3)
---
--
-
--
-
--
---
--
-
--
-
--
---
```

Ex 6: Pascal's triangle

A *Pascal's triangle* is a pattern of numbers such as:

```
    1
   1 1
  1 2 1
 1 3 3 1
1 4 6 4 1
```

where each number of a row is constructed by adding the number above and to the left with the number above and to the right, treating blank positions as 0. Write a function `pascal(c,r)` that calculates the number at a given position `c` (column), `r` (row) and returns the number at that spot. Start counting rows and columns at 0.

For example, `pascal(0, 0) = 1`, `pascal(0, 2) = 1` and `pascal(1, 3) = 3`.

Visualize the triangle as follows:

```
for i in range(0, 10):
    for j in range(0, i + 1):
        print(pascal(j, i), end = " ")
    print("")
```

This should produce:

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
```

Ex 7: Parenthesis balancing

Write a function `balance(string)` that checks whether all parentheses in a string are closed correctly. For example: `"This is (a very) nice (and not so (very) difficult) example (of the exercise)"` is a balanced string. So are `"()()"` and `"()((())())"`. Examples of unbalanced strings are `":-)"` and `"())("`. The last example shows that simply counting the opening and closing brackets is not sufficient. Use recursion to implement this function. The only built-in function you should use is `len()`.

Ex 8: Cashier

Write a function `cashier(amount, coins)` that calculates the number of ways to compose a certain amount out of a number of coins. For example, you can compose the amount 4 in 3 different ways if you have coins of 1 and 2: $1 + 1 + 1 + 1$, $1 + 1 + 2$ and $2 + 2$.

Use recursion in your function. Think about the base cases. (In how many ways can you pay if you have no coins? In how many ways can you pay an amount of 0?)

```
cashier(1, [1, 2])
# 1
cashier(4, [1])
# 1
cashier(4, [1, 2])
# 3
cashier(6, [1, 2])
# 4
cashier(6, [1, 2, 5]))
# 5
```

Ex 9: Paths

a. Path counting

Imagine you are standing in a grid at position (x, y). You want to reach the origin $(0, 0)$ and you are only allowed to move towards the origin (either decrease your x or y position by one). Write a recursive function `count_paths(x, y)` that calculates the number of possible paths from your position to the origin.

As shown in Figure 1, there are four distinct paths from position $(1, 3)$ to the origin.

```
count_paths(0, 0)
# 1
count_paths(4, 0)
# 1
count_paths(4, 1)
```

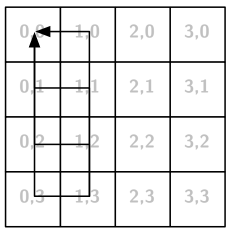

Figure 1: A 4×4 grid. There are four distinct paths from position $(1, 3)$ to the origin.

```
# 5
count_paths(8, 8)
# 12870
```

b. Path finding

Write a second function `find_paths(x, y)` that returns all paths between position (x, y) and the origin. Hint: you can easily represent a position in Python as a tuple: `(x,y)`.

```
find_paths(0, 0)
# [[(0, 0)]]
find_paths(4, 0)
# [[(4, 0), (3, 0), (2, 0), (1, 0), (0, 0)]]
find_paths(1, 1)
# [[(1, 1), (1, 0), (0, 0)], [(1, 1), (0, 1), (0, 0)]]
```

Ex 10: Sierpiński Triangle (extra)

A *Sierpiński triangle* is a fractal. An example is shown in Figure 2. Write a function `sierpinski(canvas, x, y, size, depth)` that draws a Sierpiński triangle of the requested `depth` on a canvas with the top of the triangle at position (x,y) and `size` as the width of the triangle.

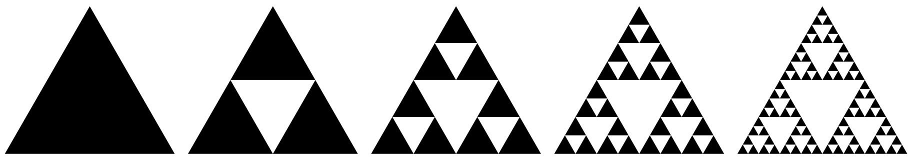

Figure 2: The evolution of a Sierpiński triangle. The first triangle is the Sierpiński triangle of depth 0, the second of depth 1, ... and so on.

One way of implementing this function is to start from a black triangle, to remove (paint white) the center triangle, and then to repeat this for the three smaller black triangles you have just created.