

DM536/DM574: INTRODUCTION TO PROGRAMMING

Exercise list (Autumn 2023)

1 Operators and expressions

1. For each of the following expressions, write the order in which it is evaluated.

- | | |
|----------------------|------------------------------|
| (a) $a - b - c - d$ | (f) $(a - (b - c)) - d$ |
| (b) $a - b + c - d$ | (g) $a \% (b \% c) * d * e$ |
| (c) $a + b / c / d$ | (h) $(a + b) * c + d * e$ |
| (d) $a / b * c * d$ | (i) $(a + b) * (c - d) \% e$ |
| (e) $a / b / c ** d$ | |

2. Suppose that i and j are two numerical variables and that b is a logical value. Remove unnecessary parentheses from each of the following expressions.

- (a) $((3 * i) + 4) / 2$
(b) $((3 * j) / (7 - i)) * (i + (-23 * j))$
(c) $((((i + j) + 3) + j) * (((i - 4) / j) + -323))$
(d) $(3 >= (j - 3)) == ((323 - (j * -7)) != (43))$
(e) $((3 >= 5) == ((\text{not } b) \text{ or } b))$
(f) $(b \text{ or } (\text{not } (b \text{ and } (3 == (i * 2)))))$
(g) $(\text{not } (\text{not } b) \text{ or } (b \text{ and } ((4 >= i+j) \text{ or } (\text{False}))))$

Could some of these expressions have been written in a simpler form?

3. For each of the following code snippets, find the value stored in each variable at the end of execution.

- | | |
|---|--|
| (a) <div style="border: 1px solid black; padding: 5px; display: inline-block;">$y = 4$
$y = y + y$</div> | (d) <div style="border: 1px solid black; padding: 5px; display: inline-block;">$b = 3.1$
$c = 0$
$c = c + 2$
$b = b * (c + 3)$</div> |
| (b) <div style="border: 1px solid black; padding: 5px; display: inline-block;">$x = 5$
$y = x$
$x = x + y$</div> | (e) <div style="border: 1px solid black; padding: 5px; display: inline-block;">$j = 2$
$i = 1$
$j = 3 + i * 2$
$i = j // 2 * i + 3$
$i = i + 1$</div> |
| (c) <div style="border: 1px solid black; padding: 5px; display: inline-block;">$y = 4$
$z = 3$
$x = y // z;$</div> | |

4. Suppose we need to work with the following data:

- an age;
- a weight;
- the number of a lottery ticket;
- a salary;
- a person's gender (male or female);
- a person's marital status (single, married, divorced, widowed);
- a distance between stars, measured in light-years;
- a distance on the Earth's surface, measured in meters.

Propose names and types for variables to store these data.

5. *Solving equations.* Write a simple program to solve second-degree equations. The program should start by asking the coefficients of the equation, and afterwards print the solutions on the screen, if there are any, or a warning, otherwise.

Recall that a second-degree equation has the general form $ax^2 + bx + c = 0$, where a , b and c are real numbers with $a \neq 0$. The solutions of this equation are $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, assuming that $b^2 - 4ac > 0$. If $b^2 - 4ac = 0$, then there is only one solution ($x = -\frac{b}{2a}$), and if $b^2 - 4ac < 0$ then the equation has no (real-valued) solutions.

2 Recursive programming with numbers

1. Write a function `sum_up_to(n: int) -> int` that returns the sum of the natural numbers up to `n`.
2. Write a function `sum_even(n: int) -> int` that returns the sum of all even numbers up to `n`.
3. Write a function `sum_between(m: int, n: int) -> int` that returns the sum of the numbers between `m` and `n`.
4. Write a function `factorial(n: int) -> int` that returns the factorial of `n`.
5. Write a function `double_factorial(n: int) -> int` that returns `n!!` ($n!! = 1 \times 3 \times 5 \times \dots \times n$, if `n` is odd, and $n!! = 2 \times 4 \times 6 \times \dots \times n$, if `n` is even).
6. Write a function `logarithm(n: int) -> int` that returns the integer base-2 logarithm of `n`.
7. Write a function `gcd(m: int, n: int) -> int` that computes the greatest common divisor of `m` and `n` using Euclides' algorithm:
$$\begin{cases} \text{gcd}(m, m) = m \\ \text{gcd}(m, n) = \text{gcd}(m, n - m) & m < n \\ \text{gcd}(m, n) = \text{gcd}(m - n, n) & m > n \end{cases}$$
8. Write a function `lcm(m: int, n: int) -> int` that returns the least common multiple of `m` and `n`.
9. Write a function `first_digit(n: int, k: int) -> int` that returns the first digit of the decimal representation of `n` in base `k`. If unspecified, `k` should take the value 10.
10. Write a function `print_multiples() -> None` that prints on the screen the multiples of 7 that are than 500 (in ascending order).

Generalize your solution to a function `print_multiples(k: int, n: int) -> None` that prints on the screen the multiples of `k` that are less than `n` (in ascending order).

Hint: use an auxiliary function.
11. Write a function `count_divisors(n: int) -> int` that returns the number of divisors of `n`.
12. A perfect number is a number that equals the sum of its divisors (excluding itself). For example, 6 is a perfect number: its divisors are $\{1, 2, 3, 6\}$, and $1 + 2 + 3 = 6$.

Write a function `is_perfect(n: int) -> bool` that checks whether `n` is a perfect number.
13. Write a function `count_perfect(n: int) -> int` that returns the number of perfect numbers smaller than `n`.
14. Write a function `is_prime(n: int) -> bool` that checks whether `n` is prime.
15. Write a function `count_primes(n: int) -> int` that returns the number of primes smaller than `n`.
16. Write a function `sum_beyond(k: int) -> int` that finds the least `n` such that the sum of the natural numbers up to `n` is at least `k`.
17. Write a function `is_palindrome(n: int) -> bool` that checks whether `n` is a palindrome.
18. Write a function `find_power(k: int) -> int` that returns the smallest number `n` such that 2^n starts with `k`. What do you have to assume about `k`?

3 Recursive programming with lists

1. Write a function `length(v: list) -> int` that returns the length of `v`.
2. Write a function `count(x: Any, v: list) -> int` that counts the number of times that `x` appears in `v`.
3. Write a function `member(x: Any, v: list) -> bool` that checks whether `x` appears in `v`.
4. Write a function `subset(v: list, w: list) -> bool` that checks whether all elements of `v` occur in `w`.

5. Write a function `set_equals(v: list, w: list) -> bool` that determines whether `v` and `w` represent the same set. Recall that a set does not have order and does not count duplicate elements.
6. Write a function `intersection(v: list, w: list) -> list` returning a list containing the elements that occur both in `v` occur in `w`.
7. Write a function `sum(v: list[int]) -> int` that sums all the values in `v`.
8. Write a function `max(v: list[int]) -> int` that returns the largest element in the nonempty list `v`.
9. Write a function `smaller_than(n: int, v: list[int]) -> int` that counts how many elements of `v` are strictly smaller than `n`.
10. Write a function `two_zeros(v: list[int]) -> bool` that checks whether `v` contains two consecutive zeros.
11. Write a function `even_after_7(v: list[int]) -> int` that computes the number of even elements in `v` occurring after the first 7.
12. Write a function `is_sorted(v: list[int]) -> bool` that checks whether the list `v` is sorted.
13. Write a function `squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from 1 to `n`.
14. Write a function `decreasing_squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from `n` to 1.
15. Write a function `divisors(n: int) -> list[int]` that returns a list containing the divisors of `n`.
16. Write a function `square_it(v: list[int]) -> list[int]` that returns a list containing the squares of all elements in `v`.
17. Write a function `reverse(v: list) -> list` that returns a list containing the elements of `v` in reverse order.
18. Write a function `compare(v: list[int], n: int) -> tuple[int,int,int]` that returns a tuple containing: as first element, the number of elements of `v` larger than `n`; as second element, the number of elements of `v` equal to `n`; and, as third element, the number of elements of `v` smaller than `n`.
19. Write a function `join(v: list, w: list) -> list` that returns a list containing the elements of `v` followed by the elements of `w` (in the original order).
20. Write a function `sorted_join(v: list[int], w: list[int]) -> list[int]` that takes two ordered lists `v` and `w` as input and returns an ordered list containing all elements from either `v` or `w`.
21. Write a function `shuffle(v: list, w: list) -> list` that takes two lists `v` and `w` and constructs a list by taking alternately one element from each of `v` and `w`.
22. Write a function `remove(x: Any, v: list) -> list` that returns a list containing all the elements of `v` that are different from `x`.

4 Recursive programming with strings

Some exercises require identifying letters, or converting characters from/to lowercase. The Python functions `chr(n: int) -> str` and `ord(s: str) -> int` convert between characters and their ASCII code. The lowercase alphabet uses ASCII codes 97 to 122, and the uppercase alphabet uses ASCII codes 65 to 90.

1. Write a function `count(c: str, s: str) -> int` that counts the number of occurrences of the character `c` in the string `s`.
2. Write a function `member(c: str, s: str) -> bool` that checks whether the character `c` appears in the string `s`.
3. Write a function `is_prefix(s1: str, s2: str) -> bool` that checks whether `s1` is a prefix of `s2`.
4. Write a function `is_suffix(s1: str, s2: str) -> bool` that checks whether `s1` is a suffix of `s2`.

5. Write a function `is_substring(s1: str, s2: str) -> bool` that checks whether `s1` is a substring of `s2`.
6. Write a function `contains(s1: str, s2: str) -> bool` that checks whether `s2` can be obtained from `s1` by deleting some characters.
7. Write a function `caesar_code(s: str, n: int) -> str` that increases the ASCII code of each character in `s` by `n`. What is the simplest way to implement the inverse function `decode`?
8. Write a function `to_uppercase(s: str) -> str` that converts the string `s` to uppercase (ignoring all non-alphabetic characters).
9. Write a function `to_lowercase(s: str) -> str` that converts the string `s` to lowercase (ignoring all non-alphabetic characters).
10. Write a function `toCamelCase(s: str) -> str` that converts a string of text into camel notation (i.e.: removes spaces and changes the first character after each space into uppercase, if it is a letter).
11. Write a function `equals_ignore_case(s1: str, s2: str)` that determines whether `s1` and `s2` are equal up to changes of case.
12. Write a function `first_position(c: str, s: str)` that returns the index of the first occurrence of the character `c` in `s`, or `-1` if `c` does not occur in `s`.
13. Write a function `last_position(c: str, s: str)` that returns the index of the last occurrence of the character `c` in `s`, or `-1` if `c` does not occur in `s`.
14. Write a function `positions(c: str, s: str) -> list[int]` that returns a list containing the indices of the occurrences of `c` in `s`.
15. Write a function `is_permutation(s1: str, s2: str) -> bool` that determines whether `s1` and `s2` contain exactly the same characters (counting repetitions).
16. Write a function `reverse(s: str) -> str` that reverses a string.
17. Write a function `reverse_words(s: str) -> str` that reverses the individual words inside a given string (preserving their order).
Hint: write an auxiliary function `split(s: str) -> list[str]` that splits a string at every occurrence of a particular character.
18. Write a function `remove_vowels(s: str) -> str` that takes a string as an argument and returns the result of removing all vowels in it.
19. Write a function `respace(s: str, n: int) -> str` that, given a string `s` and a positive integer `n`, returns the string obtained by first removing all spaces from `s` and afterwards adding a space after every `n` characters.
20. Write a function `encode_with_key(s: str, code: dict[str, str]) -> str` that encodes the string `s` character-by-character. The dictionary `code` maps each uppercase letter to its encoding; lowercase characters should be encoded accordingly, and all remaining characters left unchanged.
21. Write a function `histogram(s: str) -> dict[str, int]` that receives a string and returns a dictionary mapping each uppercase letter to the number of times it occurs (in either lower- or uppercase) in `s`. Non-alphabetic characters are not counted.
22. Write a function `replicate(s: str, v: list[int]) -> str` that receives a string and a list of the same length and returns the string containing `v[i]` copies of the character `s[i]`.

5 Functional programming

1. Write a function `sum(v: list[int]) -> int` that sums all the values in `v`.
2. Write a function `length(v: list) -> int` that returns the length of `v`.
3. Write a function `remove(x: Any, v: list) -> list` that returns a list containing all the elements of `v` that are different from `x`.

4. Write a function `count(x:Any, v: list) -> int` that counts the number of times that `x` appears in `v`.
5. Write a function `max(v: list[int]) -> int` that returns the largest element in the nonempty list `v`.
6. Write a function `square_it(v: list[int]) -> list[int]` that returns a list containing the squares of all elements in `v`.
7. Write a function `squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from 1 to `n`.
8. Write a function `decreasing_squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from `n` to 1.
9. Write a function `reverse(v: list) -> list` that returns a list containing the elements of `v` in reverse order.
10. Write a function `sum_up_to(n: int) -> int` that returns the sum of the natural numbers up to `n`.
11. Write a function `sum_between(m: int, n: int) -> int` that returns the sum of the numbers between `m` and `n`.
12. Write a function `sum_even(n: int) -> int` that returns the sum of all even numbers up to `n`.
13. Write a function `factorial(n: int) -> int` that returns the factorial of `n`.
14. Write a function `double_factorial(n: int) -> int` that returns `n!!` ($n!! = 1 \times 3 \times 5 \times \dots \times n$, if `n` is odd, and $n!! = 2 \times 4 \times 6 \times \dots \times n$, if `n` is even).
15. Write a function `member(x:Any, v: list) -> bool` that checks whether `x` appears in `v`.
16. Write a function `subset(v: list, w: list) -> bool` that checks whether all elements of `v` occur in `w`.
17. Write a function `intersection(v: list, w: list) -> list` returning a list containing the elements that occur both in `v` and in `w`.
18. Write a function `smaller_than(n: int, v: list[int]) -> int` that counts how many elements of `v` are strictly smaller than `n`.
19. Write a function `caesar_code(s: str, n: int) -> str` that increases the ASCII code of each character in `s` by `n`.
20. Write a function `to_uppercase(s: str) -> str` that converts the string `s` to uppercase (ignoring all non-alphabetic characters).
21. Write a function `to_lowercase(s: str) -> str` that converts the string `s` to lowercase (ignoring all non-alphabetic characters).
22. Write a function `count_divisors(n: int) -> int` that returns the number of divisors of `n`.
23. Write a function `is_perfect(n: int) -> bool` that checks whether `n` is a perfect number.
24. Write a function `count_perfect(n: int) -> int` that returns the number of perfect numbers smaller than `n`.
25. Write a function `is_prime(n: int) -> bool` that checks whether `n` is prime.
26. Write a function `count_primes(n: int) -> int` that returns the number of primes smaller than `n`.
27. Write a function `two_zeros(v: list[int]) -> bool` that checks whether `v` contains two consecutive zeros.
28. Write a function `is_sorted(v: list[int]) -> bool` that checks whether the list `v` is sorted.
29. Write a function `compare(v: list[int], n: int) -> tuple[int,int,int]` that returns a tuple containing: as first element, the number of elements of `v` larger than `n`; as second element, the number of elements of `v` equal to `n`; and, as third element, the number of elements of `v` smaller than `n`.
30. Write a function `positions(c: str, s: str) -> list[int]` that returns a list containing the indices of the occurrences of `c` in `s`.

31. Write a function `replicate(s: str, v: list[int]) -> str` that receives a string and a list of the same length and returns the string containing `v[i]` copies of the character `s[i]`.
32. Write a function `remove_vowels(s: str) -> str` that takes a string as an argument and returns the result of removing all vowels in it.
33. Write a function `encode_with_key(s: str, code: dict[str,str]) -> str` that encodes the string `s` character-by-character. The dictionary `code` maps each uppercase letter to its encoding; lowercase characters should be encoded accordingly, and all remaining characters left unchanged.
34. Write a function `gcd(m: int, n: int) -> int` that computes the greatest common divisor of `m` and `n` using Euclides' algorithm:
$$\begin{cases} \text{gcd}(m, m) = m \\ \text{gcd}(m, n) = \text{gcd}(m, n - m) & m < n \\ \text{gcd}(m, n) = \text{gcd}(m - n, n) & m > n \end{cases}$$
35. Write a function `first_digit(n: int, k: int) -> int` that returns the first digit of the decimal representation of `n` in base `k`. If unspecified, `k` should take the value 10.
36. Write a function `trim(s: str) -> str` that removes leading spaces in `s`.
37. Write a function `dimensions(m: list[list]) -> list[int]` that returns a list with the lengths of all elements of `m`.
38. Write a function `is_matrix(m: list[list]) -> bool` that checks whether `m` is a matrix.
39. Write a function `is_square_matrix(m: list[list]) -> bool` that determines whether `m` is a square matrix.
40. Write a function `zeros(m: int, n: int) -> list[list[int]]` that returns a matrix with `m` rows and `n` columns whose entries are all 0.
41. Write a function `identity(n: int) -> list[list[int]]` that returns a matrix with `n` rows and `n` columns whose entries are 1 in the diagonal and 0 elsewhere.
For example, `identity(3)` should return `[[1,0,0],[0,1,0],[0,0,1]]`.
42. Write a function `triangle(n: int) -> list[list[int]]` that returns a triangular array of 1s where the first row has one element and each row afterwards contains one more element than the previous one.

6 List comprehension

All these functions can be written as one-liners using list comprehension. (Re)implement them.

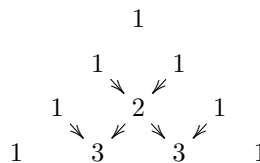
1. Write a function `squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from 1 to `n`.
2. Write a function `decreasing_squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from `n` to 1.
3. Write a function `divisors(n: int) -> list[int]` that returns a list of the divisors of `n`.
4. Write a function `square_it(v: list[int]) -> list[int]` that returns a list containing the squares of all elements in `v`.
5. Write a function `reverse(v: list) -> list` that returns a list containing the elements of `v` in reverse order.
6. Write a function `remove(x: Any, v: list) -> list` that returns a list containing all the elements of `v` that are different from `x`.
7. Write a function `positions(c: str, s: str) -> list[int]` that returns a list containing the indices of the occurrences of `c` in `s`.
8. Write a function `count(x: Any, v: list) -> int` that counts the number of times that `x` appears in `v`.

9. Write a function `member(x:Any, v: list) -> bool` that checks whether `x` appears in `v`.
10. Write a function `smaller_than(n: int, v: list[int]) -> int` that counts how many elements of `v` are strictly smaller than `n`.
11. Write a function `zeros(m: int, n: int) -> list[list[int]]` that returns a matrix with `m` rows and `n` columns whose entries are all 0.
12. Write a function `identity(n: int) -> list[list[int]]` that returns a matrix with `n` rows and `n` columns whose entries are 1 in the diagonal and 0 elsewhere.
13. Write a function `triangle(n: int) -> list[list[int]]` that returns a triangular array of 1s where the first row has one element and each row afterwards contains one more element than the previous one.
14. Write a function `multiplication_table(n: int) -> list[list[int]]` that returns a multiplication table up to `n`.
15. Write a function `transpose(m: list[list]) -> list[list]` that returns the matrix obtained from `m` by interchanging rows and columns.

7 Small projects I

The next exercises are slightly more complicated than the previous ones, and you should try to solve them using different programming styles.

1. *Finding zeros.* Given a function f on the natural numbers, a simple way to find the least n for which $f(n) = 0$ is by testing: we compute $f(0)$, $f(1)$, etc, until we find the right value.
Write a function `find_zero(f) -> int` that implements this algorithm. What happens if you call your function with argument `lambda n: n+1`?
2. *Solving equations.* A possible way to solve an equation $f(x) = 0$, if f is a continuous function, is the (*bisection method*). Given $a < b$ such that $f(a) \times f(b) < 0$, we first compute the midpoint $c = \frac{a+b}{2}$ and the value of $f(c)$; if $f(a) \times f(c) < 0$, we repeat the procedure with $b = c$; otherwise, we repeat it with $a = c$. This procedure terminates when the difference between a and b is smaller than a given value (the error), in which case the current value of a (or b , or c) is an approximate solution of $f(x) = 0$.
Implement this method as a function `bisection(f, a: float, b: float, eps: float) -> float`.
3. Write a functional implementation of the function `gcd(m: int, n: int) -> int` that computes the greatest common divisor of `m` and `n` using Euclides' algorithm.
Hint: use `fixpoint`.
4. Pascal's triangle is defined as follows: the first line contains simply the number 1; each line is computed from the previous by adding each pair of consecutive numbers, and including an additional 1 at the beginning and at the end.



Write a function `pascal(n: int) -> list[int]` that returns the `n`-th line of Pascal's triangle.

Challenge: be brave, and use `iterate`.

5. Write a function `differences(v: list[int]) -> list[list[int]]` that takes an array `v` and returns an array of arrays such that: its first line is `v`; and each other line contains the differences between consecutive elements of the previous lines. For example, for `v=[2,1,5,-2]` the expected result of `differences(v)` is `[[2,1,5,-2],[1,-4,7],[5,-11],[16]]`.
6. The sequence of Fibonacci numbers f_n is defined by $f(0) = f(1) = 1$ and $f(n+2) = f(n) + f(n+1)$. Write a function `int fibonacci(int n)` that returns the `n`-th Fibonacci number.
Note: the obvious recursive solution is easy, but it won't work.

7. Write a function `reverse_words(s: str) -> str` that reverses the individual words inside a given string (preserving their order).
8. Write a function `shift_words(s: str, n: int) -> str` that shifts each individual word inside the argument string by the given number of characters.
Note: use the function `split` defined previously.
9. Write a function `product(m1: list[list[int]], m2: list[list[int]]) -> list[list[int]]` that returns the matrix multiplication of `m1` and `m2`.

8 Imperative programming on numbers

1. Write a function `print_multiples() -> None` that prints on the screen the multiples of 7 that are than 500 (in ascending order).
Generalize your solution to a function `print_multiples(k: int, n: int) -> None` that prints on the screen the multiples of `k` that are less than `n` (in ascending order).
2. Write a function `sum_up_to(n: int) -> int` that returns the sum of the natural numbers up to `n`.
3. Write a function `sum_even(n: int) -> int` that returns the sum of all even numbers up to `n`.
4. Write a function `sum_between(m: int, n: int) -> int` that returns the sum of the numbers between `m` and `n`.
5. Write a function `sum_beyond(k: int) -> int` that finds the least `n` such that the sum of the natural numbers up to `n` is at least `k`.
6. Write a function `factorial(n: int) -> int` that returns the factorial of `n`.
7. Write a function `double_factorial(n: int) -> int` that returns `n!!`.
8. Write a function `int fibonacci(int n)` that returns the `n`-th Fibonacci number.
9. Write a function `logarithm(n: int, m: int) -> int` that returns the integer base-`m` logarithm of `n`. If unspecified, `m` should be 2.
10. Write a function `count_divisors(n: int) -> int` that returns the number of divisors of `n`.
11. Write a function `is_perfect(n: int) -> bool` that checks whether `n` is a perfect number.
12. Write a function `count_perfect(n: int) -> int` that returns the number of perfect numbers smaller than `n`.
13. Write a function `is_prime(n: int) -> bool` that checks whether `n` is prime.
14. Write a function `count_primes(n: int) -> int` that returns the number of primes smaller than `n`.
15. Write a function `nth_prime(n: int) -> int` that returns the `n`th prime number.
16. Write a function `gcd(m: int, n: int) -> int` that computes the greatest common divisor of `m` and `n` using Euclides' algorithm:
$$\begin{cases} \gcd(m, m) = m \\ \gcd(m, n) = \gcd(m, n - m) & m < n \\ \gcd(m, n) = \gcd(m - n, n) & m > n \end{cases}$$
17. Write a function `first_digit(n: int, k: int) -> int` that returns the first digit of the decimal representation of `n` in base `k`. If unspecified, `k` should take the value 10.
18. Write a function `is_palindrome(n: int) -> bool` that checks whether `n` is a palindrome.
19. Write a function `find_power(k: int) -> int` that returns the smallest number `n` such that 2^n starts with `k`.

9 Imperative programming on lists

1. Write a function `sum(v: list[int]) -> int` that sums all the values in `v`.
2. Write a function `count(x:Any, v: list) -> int` that counts the number of times that `x` appears in `v`.
3. Write a function `max(v: list[int]) -> int` that returns the largest element in the nonempty list `v`.
4. Write a function `smaller_than(n: int, v: list[int]) -> int` that counts how many elements of `v` are strictly smaller than `n`.
5. Write a function `squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from 1 to `n`.
6. Write a function `decreasing_squares(n: int) -> list[int]` that returns a list with the squares of all natural numbers from `n` to 1.
7. Write a function `divisors(n: int) -> list[int]` that returns a list containing the divisors of `n`.
8. Write a function `two_zeros(v: list[int]) -> bool` that checks whether `v` contains two consecutive zeros.
9. Write a function `is_sorted(v: list[int]) -> bool` that checks whether the list `v` is sorted.
10. Write a function `member(x:Any, v: list) -> bool` that checks whether `x` appears in `v`.
11. Write a function `subset(v: list, w: list) -> bool` that checks whether all elements of `v` occur in `w`.
12. Write a function `intersection(v: list, w: list) -> list` returning a list containing the elements that occur both in `v` occur in `w`.
13. Write a function `first_position_max(v: list[int]) -> int` that returns the index of the first occurrence of `v`'s maximum element, or `-1` if `v` is empty.
14. Write a function `last_position_max(v: list[int]) -> int` that returns the index of the last occurrence of `v`'s maximum element, or `-1` if `v` is empty.
15. Write a function `add_positions_max(v: list[int]) -> int` that returns the sum of the indices of the positions where the maximum of `v` appears.
16. Write a function `positions_max(v: list[int]) -> list[int]` that returns a list containing the indices of the positions where the maximum of `v` appears.
17. Write a function `square_it(v: list[int]) -> None` that replaces each element of `v` with its square.
18. Write a function `reverse(v: list) -> list` that returns a list containing the elements of `v` in reverse order.
19. Write a function `compare(v: list[int], n: int) -> tuple[int,int,int]` that returns a tuple containing: as first element, the number of elements of `v` larger than `n`; as second element, the number of elements of `v` equal to `n`; and, as third element, the number of elements of `v` smaller than `n`.
20. Write a function `even_after_first_7(v: list[int]) -> int` that computes the number of even elements in `v` occurring after the first 7.
21. Write a function `even_after_last_7(v: list[int]) -> int` that computes the number of even elements in `v` occurring after the last 7.
22. Write a function `sorted_join(v: list[int], w: list[int]) -> list[int]` that takes two ordered lists `v` and `w` as input and returns an ordered list containing all elements from either `v` or `w`.
23. Write a function `shuffle(v: list, w: list) -> list` that takes two lists `v` and `w` and constructs a list by taking alternately one element from each of `v` and `w`.
24. Write a function `largest_increasing_sequence(v: list) -> int` that returns the length of the largest increasing sequence of consecutive elements of `v`.

The next exercises use two-dimensional lists. Two-dimensional lists whose elements all have the same length are often called matrices.

25. Write a function `dimensions(m: list[list]) -> list[int]` that returns a list with the lengths of all elements of `m`.
26. Write a function `is_matrix(m: list[list]) -> bool` that checks whether `m` is a matrix.
27. Write a function `is_square_matrix(m: list[list]) -> bool` that determines whether `m` is a square matrix.
28. Write a function `zeros(m: int, n: int) -> list[list[int]]` that returns a matrix with `m` rows and `n` columns whose entries are all 0.
29. Write a function `identity(n: int) -> list[list[int]]` that returns a matrix with `n` rows and `n` columns whose entries are 1 in the diagonal and 0 elsewhere.
30. Write a function `triangle(n: int) -> list[list[int]]` that returns a triangular array of 1s where the first row has one element and each row afterwards contains one more element than the previous one.
31. Write a function `multiplication_table(n: int) -> list[list[int]]` that returns a multiplication table up to `n`.
32. Write a function `sum_all(m: list[list[int]]) -> int` that sums all the values in the list of lists `m`.
33. Write a function `max_all(m: list[list[int]]) -> int` that returns the largest element in the nonempty list of lists `m`.
34. Write a function `parity(m: list[list[int]]) -> None` that replaces each even element of `m` by 0 and each odd element of `m` by 1.
35. Write a function `trace(m: list[list[int]]) -> int` that returns the sum of all elements in the diagonal of `m` (the *trace* of `m`), if `m` is a square matrix.
36. Write a function `column(m: list[list], j: int) -> list` that returns the `j`-th column of the matrix `m`. Which expression gives us the `i`-th row of `m`?
37. If two matrices have the same number of rows and columns, we can add them entry-by-entry. Write a function `add(m1: list[list[int]], m2: list[list[int]]) -> list[list[int]]` that implements this operation.
38. Write a function `multiply(a: int, m: list[list[int]]) -> None` that multiplies all entries of `m` by `a`.
39. Write a function `del_row_and_col(m: list[list], i: int, j: int) -> list[list]` that returns the matrix obtained by removing the `i`-th row and the `j`-th column of `m`.
40. Write a function `differences(v: list[int]) -> list[list[int]]` that takes an array `v` and returns an array of arrays such that: its first line is `v`; and each other line contains the differences between consecutive elements of the previous lines. For example, for `v=[2,1,5,-2]` the expected result of `differences(v)` is `[[2,1,5,-2],[1,-4,7],[5,-11],[16]]`.
41. Write a function `transpose(m: list[list]) -> list[list]` that returns the matrix obtained from `m` by interchanging rows and columns.
42. Write a function `product(m1: list[list[int]], m2: list[list[int]]) -> list[list[int]]` that returns the matrix multiplication of `m1` and `m2`.

10 Imperative programming on strings

1. Write a function `count(c: str, s: str) -> int` that counts the number of occurrences of the character `c` in the string `s`.
2. Write a function `member(c: str, s: str) -> bool` that checks whether the character `c` appears in the string `s`.
3. Write a function `is_prefix(s1: str, s2: str) -> bool` that checks whether `s1` is a prefix of `s2`.

4. Write a function `is_suffix(s1: str, s2: str) -> bool` that checks whether `s1` is a suffix of `s2`.
5. Write a function `is_substring(s1: str, s2: str) -> bool` that checks whether `s1` is a substring of `s2`.
6. Write a function `contains(s1: str, s2: str) -> bool` that checks whether `s2` can be obtained from `s1` by deleting some characters.
7. Write a function `to_uppercase(s: str) -> str` that converts the string `s` to uppercase (ignoring all non-alphabetic characters).
8. Write a function `to_lowercase(s: str) -> str` that converts the string `s` to lowercase (ignoring all non-alphabetic characters).
9. Write a function `toCamelCase(s: str) -> str` that converts a string of text into camel notation (i.e.: removes spaces and changes the first character after each space into uppercase, if it is a letter).
10. Write a function `equals_ignore_case(s1: str, s2: str)` that determines whether `s1` and `s2` are equal up to changes of case.
11. Write a function `first_position(c: str, s: str)` that returns the index of the first occurrence of the character `c` in `s`, or `-1` if `c` does not occur in `s`.
12. Write a function `last_position(c: str, s: str)` that returns the index of the last occurrence of the character `c` in `s`, or `-1` if `c` does not occur in `s`.
13. Write a function `positions(c: str, s: str) -> list[int]` that returns a list containing the indices of the occurrences of `c` in `s`.
14. Write a function `is_permutation(s1: str, s2: str) -> bool` that determines whether `s1` and `s2` contain exactly the same characters (counting repetitions).
15. Write a function `reverse(s: str) -> str` that reverses a string.
16. Write a function `reverse_words(s: str) -> str` that reverses the individual words inside a given string (preserving their order).
17. Write a function `remove_vowels(s: str) -> str` that takes a string as an argument and returns the result of removing all vowels in it.
18. Write a function `respace(s: str, n: int) -> str` that, given a string `s` and a positive integer `n`, returns the string obtained by first removing all spaces from `s` and afterwards adding a space after every `n` characters.
19. Write a function `shift_words(s: str, n: int) -> str` that shifts each individual word inside the argument string by the given number of characters.
20. Write a function `caesar_code(s: str, n: int) -> str` that increases the ASCII code of each character in `s` by `n`.
21. Write a function `encode_with_key(s: str, code: dict[str,str]) -> str` that encodes the string `s` character-by-character. The dictionary `code` maps each uppercase letter to its encoding; lowercase characters should be encoded accordingly, and all remaining characters left unchanged.
22. Write a function `histogram(s: str) -> dict[str,int]` that receives a string and returns a dictionary mapping each uppercase letter to the number of times it occurs (in either lower- or uppercase) in `s`. Non-alphabetic characters are not counted.
23. Write a function `replicate(s: str, v: list[int]) -> str` that receives a string and a list of the same length and returns the string containing `v[i]` copies of the character `s[i]`.

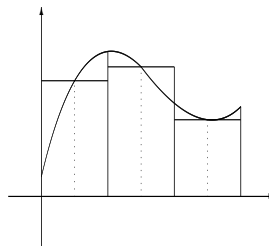
11 Small projects II

This exercises are all meant to be solved imperatively, possibly with some recursive/functional implementations of helper functions.

1. The *sieve of Eratosthenes* is one of the oldest algorithms to find all prime numbers up to a given n . First, one writes down a list containing all numbers from 1 to n , and crosses out the 1. Then, one repeatedly picks the next number k from the list that has not been crossed out, and crosses out all larger multiples of k . When the end of the list is reached, the numbers not crossed out are precisely the prime numbers smaller than or equal to n .

Implement this algorithm as a function `eratosthenes(n: int) -> list[int]`.

2. *Computing areas.* Suppose f is a continuous and positive function in an interval $[a, b]$. The area between the horizontal axis and the graph of f in the interval $[a, b]$ (also called the *integral* of f in $[a, b]$) can be computed as precisely as required by the following method: we divide the interval $[a, b]$ in n subintervals of equal width, and approximate the integral of f in each subinterval by the area of the rectangle whose height is given by the value of f value in the midpoint of the interval (see the figure below).



Implement this method as a function `integral(f, a: float, b: float, n: int) -> float` that returns the computed approximate value of the integral of f in the interval $[a, b]$.

3. Reimplement the bisection method presented earlier imperatively.
4. Reimplement the function `pascal(n: int) -> list[int]` that returns the n -th line of Pascal's triangle imperatively.

12 Implementing datatypes

1. *Time management.* Implement a module `timestamp` defining a datatype `TimeStamp` that allows working with points in time, characterized by hours, minutes and seconds. Your module should also provide the following functions:
 - a function `make_timestamp(hours: int, minutes: int, seconds: int) -> TimeStamp` returning a new instance of `TimeStamp` representing the given time, where all arguments are optional and default to 0 when absent;
 - a function `valid(hours: int, minutes: int, seconds: int) -> bool` that checks whether its given arguments are in the valid range;
 - a function `skip_second(t: TimeStamp) -> None`, a function `skip_minute(t: TimeStamp) -> None`, and a function `skip_hour(t: TimeStamp) -> None` that add one second, one minute and one hour, respectively, to the timestamp t (assume that 23:59:59 is followed by 0:00:00);
 - a function `skip(t1: TimeStamp, t2: TimeStamp) -> None` that adds the amount of time described in $t2$ to $t1$;
 - a function `equals(t1: TimeStamp, t2: TimeStamp) -> bool` that determines whether $t1$ and $t2$ represent the same timestamp;
 - a function `copy(t: TimeStamp) -> TimeStamp` that returns a copy of t ;
 - a function `to_string(t: TimeStamp) -> str` that returns a textual representation of t .
2. On top of the previous module, implement a module `date` defining a datatype `Date` whose instances include information about the year, month, day and timestamp. Your module should exploit `timestamp` as much as possible, and define the following functions:

- a function `make_date(year: int, month: int, day: int, time: Timestamp) -> Date` that creates an instance of `Date` corresponding to the given time on the given date; if the time is omitted, it should be set to midnight;
 - a function `valid(year: int, month: int, day: int) -> bool` that checks whether its arguments represent a valid date;
 - a function `skip_day(d: Date) -> None`, a function `skip_month(d: Date) -> None`, and a function `skip_year(d: Date) -> None` that skip the date `d` forward by one day, one month or one year, respectively;
 - a function `skip_time(d: Date, t: Timestamp) -> None` that skips `d` forward by the indicated amount of time;
 - a function `equals(d1: Date, d2: Date) -> bool` that determines whether `d1` and `d2` represent the same date;
 - a function `copy(d: Date) -> Date` that returns a copy of `d`;
 - a function `to_string(d: Date) -> str` that returns a textual representation of `d`.
3. A point on a flat surface (such as a computer screen) is defined by two coordinates, also called its horizontal and vertical components. Implement a module `point2d` defining a datatype `Point2D` whose instances represent two-dimensional points. Your module should also provide the following methods:
- a function `make_point(x: float, y: float) -> Point2D` that returns a new instance of `Point2D` with the given coordinates;
 - a function `move(p: Point2D, dx: float, dy: float) -> None` that moves `p` according to the vector `(dx,dy)`;
 - a function `distance_to_origin(p: Point2D) -> float` that returns `p`'s distance to the origin;
 - a function `distance(p1: Point2D, p2: Point2D) -> float` that returns the distance between `p1` and `p2`;
 - a function `equals(p1: Point2D, p2: Point2D) -> bool` that determines whether `p1` and `p2` represent the same point;
 - a function `copy(p: Point2D) -> Point2D` that returns a copy of `p`;
 - a function `to_string(p: Point2D) -> str` that returns a textual representation of `p`.

Consider the following client for your module.

```
p1 = make_point(0, 0)
p2 = p1

move(p1, 1, 1)
print(to_string(p2))

move(p2, 3, 3)
print(to_string(p2))
```

What output is produced by this program?

4. A polygon is a region on the plane limited by straight line segments (its sides). Implement a module `polygon` defining a datatype `Polygon` whose instances are polygons, represented as a sequence of points (its vertices) such that there is a line between each two consecutive points, as well as between the first and the last. Exploit module `point2d` as much as possible. Your module should also provide the following methods:
- a function `make_polygon(v: list[Point2D]) -> Polygon` that creates a polygon from the list of its vertices;
 - a function `perimeter(p: Polygon) -> float` returning the perimeter of `p`;
 - a function `nearest(p: Polygon) -> Point2D` that returns the vertex of `p` that is closest to the origin;
 - a function `longest_side(p: Polygon) -> float` returning the length of `p`'s longest side;

- a function `move(p: Polygon, dx: float, dy: float) -> None` that moves `p` according to the vector `(dx,dy)`;
 - a function `vertices_in_quadrant(p: Polygon, n: int) -> int` counting how many of `p`'s vertices lie on the `n`-th quadrant;
 - functions `is_triangle(p: Polygon) -> bool` and `is_rectangle(p: Polygon) -> bool` that determine whether `p` is a triangle or a rectangle, respectively;
 - a function `equals(p1: Polygon, p2: Polygon) -> bool` that determines whether `p1` and `p2` represent the same polygon (note that the polygon's vertices need not be given in the same order); ;
 - a function `copy(p: Polygon) -> Polygon` that returns a copy of `p`;
 - a function `to_string(p: Polygon) -> str` that returns a textual representation of `p`.
5. A discrete representation of a function f is a list of `Point2D` such that $f(x) = y$ holds for each point (x, y) of the list, and the elements of the list are sorted by their first coordinate.

Implement a module `graph` defining a datatype `Graph` whose instances are discrete representations of functions. Your module should also provide the following functions:

- a function `make_graph(p: list[Point2D]) -> Graph` that receives a list of the form described and returns the corresponding function representation;
 - a function `valid(graph: Point2D) -> bool` that checks whether its argument can be passed along to the previous function;
 - a function `max(f: Graph) -> float` that finds the maximum value of `f`, according to its discrete representation;
 - a function `increasing(f: Graph) -> bool` that checks whether `f` is increasing or not;
 - a function `changeRate(f: Graph) -> list[float]` that returns a list with one less element than the graph of `f` containing this function's average rate of change in each interval;
 - a function `equals(f1: Graph, f2: Graph) -> bool` that determines whether `f1` and `f2` are two equal discrete representations of the same function;
 - a function `copy(f: Graph) -> Graph` that returns a copy of `f`;
 - a function `to_string(f: Graph) -> str` that returns a textual representation of `f`.
6. *Shopping carts.* An online supermarket is building a backoffice system, consisting of several interacting datatypes.

Your task is to implement a module `shopping_cart` defining a datatype `ShoppingCart` to represent shopping carts containing the products selected by a client. The products in the shopping cart are themselves instances of a datatype `Product` that you will not develop, and which provides (among others) the following functions:

- `price(p: Product) -> float`: returns the price of product `p`;
- `equals(p1: Product, p2: Product) -> bool`: returns `True` if `p1` and `p2` represent the same product;
- `copy(p: Product) -> Product`: returns a new product that is equal to `p`.

Implement module `shopping_cart`. Besides defining the datatype `ShoppingCart`, your module should also provide the following methods:

- a function `make_shopping_cart() -> ShoppingCart` that returns an empty shopping cart;
- a function `add(p: Product, s: ShoppingCart) -> None` that adds product `p` to shopping cart `s`;
- a function `number_of_items(s: ShoppingCart) -> int` returning the number of products in `s`;
- a function `free_delivery(s: ShoppingCart) -> bool` indicating whether `s` is eligible for free delivery (only for shopping carts with more than 50 items);
- a function `total_price(s: ShoppingCart) -> float` returning the cost of `s`;
- functions `most_expensive(s: ShoppingCart) -> Product`, returning the most expensive product in `s`, and `highest_price(s: ShoppingCart) -> float`, returning its price;
- a function `how_many(p: Product, s: ShoppingCart) -> int` returning the number of items equal to `p` in `s`;
- a function `remove_most_expensive(s: ShoppingCart) -> None` removing all copies of the most expensive product in `s`.