EXERCISE 4: ITERATIVE AND RECURSIVE ALGORITHMS

Programming in Bioinformatics

Iterative and recursive algorithms

The change problem

Imagine you are in a bookstore and you're buying a book for 423 Czech Koruna (CZK). At the cashier, you pay with a 500 CZK banknote. The cashier needs to give you 77 CZK in change. If she returns your change using 50, 20, 5, and 2 CZK coins, you'll be satisfied because you'll have the minimum possible number of coins in your wallet that she could have given you. However, if she wanted to give you those 77 CZK in 77 one-Koruna coins, you might have doubts about her mental health.

How do you decide which coins and in what quantities to return to the customer to minimize the number of coins they receive? This is an example of a problem that can be algorithmized. However, before doing that, we need to analyze the problem and understand what we have as input, what we want as output, and what resources we have available.

Input:	the amount of money *M* to be returned
Output:	the minimum number of coins whose total value will be equal to *M*
Resource:	coins in denominations of 50 (*pd*), 20 (*dc*), 10 (*ds*), 5 (*p*), 2 (*d*), and 1 (*j*) CZK
Condition:	50 * *pd* + 20 * *dc* + 10 * *ds* + 5 * *p* + 2 * *d* + 1 * *j* = *M*
	pd, *dc*, *ds*, *p*, *d*, and *j* must have the smallest possible value

Outline of the algorithm in pseudocode:

```
ReturnCoins(M)
1 while M > 0
2   c ← the largest denomination coin but less than or equal to M
3   give the coin to the customer
4   M ← M - c
```

In a bit more detailed description:

```
ReturnCoins(M)
1  Give the customer the integer result of dividing M by 50 in 50 CZK coins.
2  Let remainder be the remaining amount due the customer.
3  Give the customer the integer result of dividing remainder by 20 in 20 CZK coins.
4  Let remainder be the remaining amount due the customer.
5  Give the customer the integer result of dividing remainder by 10 in 10 CZK coins.
6  Let remainder be the remaining amount due the customer.
7  Give the customer the integer result of dividing remainder by 5 in 5 CZK coins.
8  Let remainder be the remaining amount due the customer.
9  Give the customer the integer result of dividing remainder by 2 in 2 CZK coins.
10 Give the customer remainder after division in 1 CZK coins.
```

Task 1

- Write pseudocode according to the previous detailed description of the algorithm.
- Implement the algorithm in R.

Current solution of the change problem is not universal; we have only considered the case for coins in the Czech currency. What if we want to generalize the problem for use in any currency, or we have a limited number of coins of certain values? Let's first consider the general case for any currency:

Input: the amount of money M to be returned
an array of d denominations $c = (c_1, c_2, ..., c_d)$ in descending order $(c_1 > c_2 > ... > c_d)$
Output: integer values $i_1, i_2, ..., i_d$
Condition: $c_1 * i_1 + c_2 * i_2 + ... + c_d * i_d = M$, where $i_1 + i_2 + ... + i_d$ is as small as possible

Task 2

- Write a pseudocode for the change problem for any currency. Hint: use array indexing.
- Implement the pseudocode in R as a separate function, with the input being the amount to be returned and an array of denominations (see above).
- Find input values for which the algorithm will not work correctly, meaning the output will be incorrect.

Correct algorithm must never return incorrect output for any input variation! A correct algorithm handles all possibilities.

A proper solution to the change problem in any currency or coin values could be as follows:

```
ReturnCoins(M, c, d)
1  minimumCoinCount ← ∞
2  for each (i₁, i₂,…, i_d) from (0,…,0) to (M/c₁,…, M/c_d)
3    valueOfCoins ← ∑ᵈ_{k=1} i_k * c_k
4    if valueOfCoins = M
5      coinCount ← ∑ᵈ_{k=1} i_k
6      if coinCount < minimumCoinCount
7        minimumCoinCount ← coinCount
8        bestChange ← (i₁, i₂,…, i_d)
9  return bestChange
```

Iterative vs recursive algorithms

A problem can be solved either iteratively or recursively only if it keeps repeating the same sequence of tasks. A recursive algorithm calls itself within its own code, i.e., it divides a problem into subproblems until it reaches the smallest part of the problem that can be immediately solved. After solving it, the algorithm gradually solves the problem by combining the results of the previously solved subproblems. In contrast, an iterative algorithm solves individual recurring parts of the problem independently and sequentially (the use of loops like *while* and *for* provides the simplest basis for an iterative algorithm).

How to create a recursive algorithm: The most chocolate path

Imagine a mouse in a maze with little rooms, each containing a different amount of chocolate bars, as shown in the image below. The amount of chocolate bars is represented by numbers. The mouse can only move downward and diagonally to the right. Its goal is to eat as much chocolate as possible. What path will be the right one?

We want to solve the algorithm recursively, so we need to find the core of the problem and base case of this problem. The path begins at the top, and the mouse has two options for where to go, either straight down or diagonally to the right. We can imagine the maze as a matrix M where r marks rows and c marks columns. The initial position of the mouse is $M_{1,1}$, generally $M_{r,c}$. The mouse can change its position to $M_{r+1,c}$ or $M_{r+1,c+1}$.

The matrix can be of any size, and individual cells can have any value. The most trivial case to solve is when the matrix has only three elements (1 upper and 2 lower). The next (superior) problem is adding another row to the matrix. By solving the path for a 3-element matrix, we find a position from which we can solve the next row. We have thus found the core of the recursion, which is the solution to the 3-element matrix problem.

The mouse begins here at the top with 3 bars :

```
3
1 4
5 3 0
1 2 6 7
```

and can end up anywhere at the bottom.

Core of the recursion:

```
x
y z
```

We solve the core problem of the recursion as follows. There are only two possibilities for the mouse's movement:

1. From position $M_{1,1}$, it moves to position $M_{2,1}$, gaining 3 + 1 = 4 chocolate bars.
2. From position $M_{1,1}$, it moves to position $M_{2,2}$, gaining 3 + 4 = 7 chocolate bars.

Since the mouse is looking for the path with the maximum amount of chocolate, the solution to the core problem is the path where the mouse acquires a total of 7 pieces of chocolate. We can generalize the solution for this specific example: we are looking for the maximum value from $M_{r,c} + M_{r+1,c}$ and $M_{r,c} + M_{r+1,c+1}$.

Solution of the core problem in pseudocode:

```
Core(M, r, c)
1  bars ← M_{r, c}
2  down ← M_{r + 1, c}
3  diagonal ← M_{r + 1, c + 1}
4  return max(down, diagonal) + bars
```

The general solution of the core problem gives us a new starting position (i.e., the indexes *r* and *c*) for solving the next problem. At the same time, the original matrix is reduced as some elements cannot be considered further, given that we can only move downward and diagonally to the right. (Note that the reduction is purely conceptual; nothing is actually removed.)

Case for path to $M_{r+1,c}$

```
1
5 3 0
1 2 6 7
```

Case for path to $M_{r+1,c+1}$

```
4
3 0
2 6 7
```

We must solve all variants because we don't know which one will lead to the path with the maximum amount of chocolate bars. However, when you look closely, both variants start with the core problem. Likewise, all other variants that would result from any solution also start with the same core problem. This leads us to the exact same subproblem (except for values) as before. Once again, we use the function `Core(M, r, c)`, but the indexes *r* and *c* will not be equal to 1, as they were initially; they will be equal to the indexes derived from the previous function call. However, since we don't know what the previous solution was, we continue to delve deeper into the matrix. The actual problem solution starts from the end, moving backward from the lowest elements of the matrix up to position $M_{1,1}$, solving all possibilities through recursion and gradually selecting the best one.

We solve the algorithm recursively, meaning that we call the core of the recursion:

```
1  bars ← M_{r, c}
2  down ← Core(M, r + 1, c)
3  diagonal ← Core(M, r + 1, c + 1)
4  return max(down, diagonal) + bars
```

Before constructing the entire recursive algorithm, we need to clarify when the algorithm will stop. In this case, it will stop when *r* is equal to the number of rows in the matrix, which means the mouse reaches the cell in the last row, and we want to know how much chocolate is there. When writing the code for a recursive algorithm, always address the termination condition first.

Here is the entire recursive algorithm for finding the most chocolate path (currently, it only tells us how much chocolate the mouse will consume). We have replaced the `Core()` function with a recursive call to the final function.

```
Chocolate(M, r, c)
1 if r = number of rows in M
2    return M_r, c
3  else
4    bars ← M_r, c
5    down ← Chocolate(M, r + 1, c)
6    diagonal ← Chocolate(M, r + 1, c + 1)
7    return max(down, diagonal) + bars
```

Task 3

- Implement the recursive algorithm in R.
- Solve the same problem iteratively.

The towers of Hanoi

We have three pegs. On the left peg, there are discs stacked on top of each other, with smaller discs always placed on top of larger ones. The goal is to move all the discs to the right peg. A larger disc must never be placed on top of a smaller one. You can place any disc on an empty peg.

Input: an integer *n* (number of discs)
Output: the sequence of steps to solve the towers of Hanoi problem with *n* discs

The most basic problem of the towers of Hanoi is to move 1 disc from one peg to another. This is more or less the final step in solving the problem. Just like in the previous example with chocolates, the algorithm's code must first handle termination. In this case, when the number of discs is *n* but there is only one left, you simply move it from the current peg (`fromPeg`) to the desired peg (`toPeg`) and finish the process.

In the opposite case, the problem for the recursive algorithm is reduced by 1 (i.e., *n* - 1), and the discs are moved from the current peg (`fromPeg`) to another free peg (`unusedPeg`), not the one where you want to move all the others. Then you can move the largest remaining disc to the peg you originally intended (`toPeg`). As the final step, move the pile of *n* - 1 discs from the so-called free peg (`unusedPeg`) to the largest disc, which is already on the desired peg (`toPeg`). Now, we can translate this idea into a recursive algorithm:

```
HanoiTowers(n, fromPeg, toPeg)
1  if n = 1
2    output “Move disc from peg fromPeg to peg toPeg”
3    return
4 unusedPeg ← 6 - fromPeg - toPeg
5 HanoiTowers(n - 1, fromPeg, unusedPeg)
6 output “Move disc from peg fromPeg to peg toPeg”
7 HanoiTowers(n - 1, unusedPeg, toPeg)
8 return
```

Task 4

- Implement this algorithm in R and solve the problem for 5 discs.