The aim of the course *Data Structures and Algorithms* is to advance your programming skills and teach you techniques and ways of thinking that help you in implementing programs that are correct and efficient in all circumstances.

The course uses the Python language but the techniques taught on the course are applicable to other programming languages too. The course involves a lot of programming but some theoretical ideas and concepts are covered too.

What is an algorithm?

An *algorithm* is a method for solving some computational problem. An algorithm implemented in some programming language can be executed on a computer.

The *input* of an algorithm is the initial data provided to the algorithm. The *output* of an algorithm is the answer produced by the algorithm by the end of its execution. In Python an algorithm can be implemented as a function, and then typically the input is given as the function parameters and the output is the return value.

Let us consider an example, where the algorithm is given a list of numbers and the task is to count how many of the numbers are even. For example if the list is `[5, 4, 1, 7, 9, 6]`, the desired answer is `2`, because `4` and `6` are the even numbers.

This task can be solved with an algorithm that goes through the numbers on the list and maintains a variable that stores the count of even numbers seen so far. The algorithm can be implemented as a Python function `count_even`:

```
def count_even(numbers):
    result = 0
    for x in numbers:
        if x % 2 == 0:
            result += 1
    return result
```

The function can be tested with the following main program:

```
print(count_even([1, 2, 3])) # 1
print(count_even([2, 2, 2, 2, 2])) # 5
print(count_even([5, 4, 1, 7, 9, 6])) # 2
```

Here the function is tested with three different lists. For each test, the desired answer is given as a comment at the end of the line. When the program is executed, it prints out:

```
1
5
2
```

Thus the function produces the desired output, at least for these three lists, and it seems we have created a correctly working algorithm for the task.

What is a data structure?

A *data structure* is a way of storing data within a program. The basic data structure in Python is the list, but there are many other standard data structures too. The choice of data structures is an important part of designing an algorithm, because the data structures have a big effect on the efficiency of the algorithm.

On this course, we learn about many data structures and their uses in designing algorithms. We cover many standard Python data structures, and learn to implement data structures not provided by Python or other programming languages.

Implementing an algorithm

Any algorithm can be implemented with a few basic programming constructs. In Python, these basic constructs are:

- variables
- operators (`+`, `=` etc.)
- conditionals (`if`)
- loops (`for`, `while`)
- lists
- functions

- classes

In addition to these, programming languages have many other features that can help shorten the code, but do not affect the fundamental operating logic of the code. They can be used in implementing algorithms but are not necessary.

Let us return to the earlier example function `count_even` that was implemented with the basic constructs:

```
def count_even(numbers):
    result = 0
    for x in numbers:
        if x % 2 == 0:
            result += 1
    return result
```

This can be implemented more compactly with a special Python construct, the generator expression:

```
def count_even(numbers):
    return sum(x % 2 == 0 for x in numbers)
```

Here the `sum` function encloses a generator expression that computes the value of the expression `x % 2 == 0` for each element `x` of the list. The possible values are `True` and `False`, but when they are summed up, each `True` is counted as the number `1` and each `False` as the number `0`. Thus the result of the summation is the count of even numbers.

The latter function is much shorter but its fundamental operation is the same as the former one's. Both functions go through the numbers on the list and add up the times when an even number is encountered. The operating logic is essentially the same in both cases.

The advantage of the first function is that it is easier to explain to a person who is not familiar with Python special constructs. The function could be easily translated into other programming languages, for example JavaScript:

```
function countEven(numbers) {
    let result = 0;
    for (let x of numbers) {
        if (x % 2 == 0) result++;
```

```
    }
    return result;
}
```

The advantage of the second function is that is more concise and perhaps more in the style of Python language. Even though the basic constructs are sufficient, it can be interesting to learn more special constructs too.

Efficiency of algorithms

The same task can be solved by different algorithms, and there can be big differences in their efficiencies. Often the goal is to find an efficient algorithm that solves the task quickly.

Let us consider a task, where we are given a list of numbers, and the goal is to find the largest difference between any two numbers. For example, when the list is `[3, 2, 6, 5, 8, 5]`, the desired answer is `6`, because the largest difference is between the numbers `2` and `8`.

Three algorithms for solving the task are given below:

Algorithm 1

```
def max_diff(numbers):
    result = 0
    for x in numbers:
        for y in numbers:
            result = max(result, abs(x - y))
    return result
```

The first algorithm has two nested `for` loops that go through all ways of choosing two numbers from the list. The algorithm computes the difference using the `abs` (absolute value) function and remembers the largest difference it has encounter so far.

Algorithm 2

```
def max_diff(numbers):
    numbers = sorted(numbers)
    return numbers[-1] - numbers[0]
```

The idea of the second algorithm is that the biggest difference must be between the smallest number and the largest number on the list.

The algorithm first sorts the list using the `sorted` function. Then the smallest number is in the beginning (index `0`) and the largest is at the end (index `-1`) of the list.

Algorithm 3

```
def max_diff(numbers):
    return max(numbers) - min(numbers)
```

The third algorithm is based on finding the smallest and largest numbers too, but instead of sorting, it uses the functions `min` and `max`.

Measuring efficiency

The efficiency of an algorithm can be studied with a test program that runs the algorithm for a given input and measures the execution time. It is often a good idea to write the test program so that it generates a random input of a given size. Then it is easy to test the algorithm with inputs of different sizes.

Below is a program that tests the efficiency of the `max_diff` function:

```
import random
import time

def max_diff(numbers):
    ...

n = 1000
print("n:", n)
random.seed(1337)
numbers = [random.randint(1, 10**6) for _ in range(n)]

start_time = time.time()
result = max_diff(numbers)
end_time = time.time()

print("result:", result)
print("time:", round(end_time - start_time, 2), "s")
```

The value of the variable `n` is the length of the list used in the test. The function `random.seed` sets the seed number (here `1337`) of the random number generator so that it will always generate the same random numbers. This makes comparisons between test runs easier and more reliable. A list of `n` random numbers between $1 \ldots 10^6$ is created with the function `random.randint` and a generator expression.

The program measures the execution time using the `time.time` function. The function returns the time elapsed since the beginning of the year 1970 in seconds. The difference between the time before and the time after the execution of the algorithm tells how many seconds did the execution take. The time is rounded to two decimals with the function `round`.

The execution of the test program outputs something like this:

```
n: 1000
result: 999266
time: 0.09 s
```

This means that the input to the algorithm was a list of length `1000`, the output of the algorithm was `999266`, and the execution of the algorithm took `0.09` seconds.

The following table shows the execution times of the above three algorithms for inputs of different sizes on the test computer:

List length `n`	Algorithm 1	Algorithm 2	Algorithm 3
1000	0.17 s	0.00 s	0.00 s
10000	15.93 s	0.00 s	0.00 s
100000	–	0.01 s	0.00 s
1000000	–	0.27 s	0.02 s

The table reveals big differences in the efficiencies of the algorithms. Algorithm 1 is slow on large inputs, and the tests with the two largest inputs were aborted because the execution took too much time. Algorithms 2 and 3, on the other hand, are efficient on large inputs too. The largest input exposes a difference between Algorithms 2 and 3 too, although the difference is not as big as compared to Algorithm 1.

Analysis of algorithms

The efficiency of an algorithm can be estimated by counting how many steps the algorithm executes for an input of a given size. Often we can think of a step as corresponding to a line of code.

Let us consider the example algorithm that counts the number of even numbers on a list:

```
1  def count_even(numbers):
2      result = 0
3      for x in numbers:
4          if x % 2 == 0:
5              result += 1
6      return result
```

Let n denote the length of the list. Since the algorithm goes through all elements of the list, the number of steps depends on n.

- The lines 2 and 6 are executed once, because they are outside the loop.
- The lines 3 and 4 are executed n times, because they are executed once for each element of the list.
- The line 5 is executed at least 0 times and at most n times, depending on the count of even numbers on the list.

Thus the algorithm executes at least $2n + 2$ steps and at most $3n + 2$ steps. The exact number of steps depends on the contents of the list.

Time complexity

Often we do not need to determine the exact number of steps, but it is enough to know the *time complexity*, which gives the magnitude of the number of steps on a given input size.

A time complexity is usually shown in the form $O(\cdots)$, where the three dots are replaced by an arithmetic expression representing an upper bound on the number of steps. The expression involves a variable n that represents the size of the input. For example, if the input is a list, n is the length of the list, and if the input is a string, n is the length of the string.

The time complexity expression is typically a simplified form of the expression for

the exact number of steps, obtained by retaining only the fastest growing term of the expression and removing all constants. For example, the time complexity of the preceding algorithm is $O(n)$ because the exact number of steps is at most $3n + 2$.

Formally, the time complexity of an algorithm is $O(f(n))$ if we can choose two constants c and n_0 so that the algorithm executes at most $cf(n)$ steps when $n \geq n_0$. For example, the preceding algorithm has time complexity $O(n)$ because we can choose $c = 5$ and $n_0 = 1$. These are valid choices because $3n + 2 \leq 5n$ when $n \geq 1$.

Common time complexities include:

Time complexity	Name of complexity class
$O(1)$	Constant
$O(\log n)$	Logarithmic
$O(n)$	Linear
$O(n \log n)$	–
$O(n^2)$	Quadratic
$O(n^3)$	Cubic

Time complexity of loops

In practice, the time complexity is often determined by the loops in the code.

Constant time

If an algorithm has no loops and it executes the same steps independent of the input, its time complexity is $O(1)$.

For example, the time complexity of the following algorithms is $O(1)$:

```
def middle(numbers):
    n = len(numbers)
    return numbers[n // 2]
```

Single loop

If the algorithm contains a single loop that goes through all elements of the input, its time complexity is $O(n)$.

For example, the time complexity of the following algorithms is $O(n)$:

```
def calc_sum(numbers):
    result = 0
    for x in numbers:
        result += x
    return result
```

The time complexity is $O(n)$ because it the algorithm has a single loop that goes through the elements of the list.

Nested loops

If an algorithm contains a loop inside a loop, each of which goes through all elements of the input, its time complexity $O(n^2)$.

For example, the following algorithm has time complexity $O(n^2)$:

```
def has_sum(numbers, x):
    for a in numbers:
        for b in numbers:
            if a + b == x:
                return True
    return False
```

More generally, if an algorithm has k nested loops each of which goes through all elements of the input, its time complexity is $O(n^k)$.

Sequential code segments

If the algorithm consists of multiple code segments in sequence, the whole time complexity is the maximum of the segment time complexities.

For example, the time complexity of the following algorithm is $O(n)$:

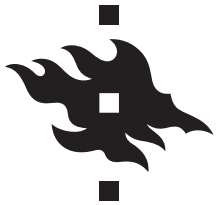
```
def count_min(numbers):
    # stage 1
```

```
    min_value = numbers[0]
    for x in numbers:
        if x < min_value:
            min_value = x

    # stage 2
    result = 0
    for x in numbers:
        if x == min_value:
            result += 1

    return result
```

In stage 1, the algorithm finds the minimum element on the list by going through all elements. The time complexity of stage 1 is $O(n)$.

In stage 2, the algorithm goes through the elements again to count the occurrences of the minimum element. The time complexity of stage 2 is $O(n)$.

Since each stage has time complexity $O(n)$, the time complexity of the whole algorithm is $O(n)$.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI