1 Introduction and overview

1.1 Algorithms and programs

- An **algorithm** is an exact description of how some (computational) task is to be carried out.
- A **program** is the presentation (implementation) of an algorithm in some programming language executable by a computer.
1.2 Example: selection sort

Consider the common task of arranging \( n \) integers contained in array \( A[1 \ldots n] \) in increasing order. A high-level “pseudocode” presentation of the simple selection sort algorithm for this task is as follows:

**Algorithm 1:** The selection sort algorithm

1. **function** `SELECTSORT (A[1 \ldots n])`
2. **Input:** Integer array \( A[1 \ldots n] \).
3. **Output:** Array \( A \), with same items in increasing order.
4. Set \( i \leftarrow 1 \).
5. Find smallest of items \( A[i], \ldots, A[n] \). Let this have index \( i_{\text{min}} \).
6. Exchange elements \( A[i] \) and \( A[i_{\text{min}}] \).
7. Set \( i \leftarrow i + 1; \) if \( i < n \), go back to line 3.
8. When \( i = n \), the items in \( A \) are in increasing order.
The algorithm can be implemented in, e.g., the Python programming language (V.3) as follows:

```python
def select_sort(A):
    n = len(A)
    for i in range(n-1):
        imin = i
        for j in range(i+1, n):
                imin = j
    return A
```
Let us embed this Python function in a complete executable module, with some timing features, and test it:

```python
import random
import time

def select_sort(A):
    [...] 

rand_max = 2147483647  # Same value as in C.
nmax = 10000000

while True:
    try:
        n = int(input("Number of items = "))
    except EOFError:
        break
    if n < 1:
        print("Number must be positive. Try again.")
        continue
```
if n > nmax:
    print("Number too large. Try again.")
    continue

print("Generating %d random numbers in the range 0..%d." % (n, rand_max))
A = [random.randint(0,rand_max) for i in range(n)]

print("Beginning selection sort of %d items." % (n))
start = time.time()

select_sort(A)

stop = time.time()
print("Done. Time = %.4f seconds." % (stop - start))

print("\n")
> python3 selectsort.py
Number of items = 2000
Generating 2000 random numbers in the range 0..2147483647.
Beginning selection sort of 2000 items.
Done. Time = 0.2238 seconds.
Number of items = 4000
Generating 4000 random numbers in the range 0..2147483647.
Beginning selection sort of 4000 items.
Done. Time = 0.8275 seconds.
Number of items = 6000
Generating 6000 random numbers in the range 0..2147483647.
Beginning selection sort of 6000 items.
Done. Time = 1.8435 seconds.
...
Number of items = 20000
Generating 20000 random numbers in the range 0..2147483647.
Beginning selection sort of 20000 items.
Done. Time = 19.7716 seconds.
Figure: Runtime plot of selection sort (x: elements, y: seconds).

The runtime of the program appears to grow as the square of the number $n$ of items to be sorted. Ignoring the constants, we say that the runtime is of order $n^2$, or briefly $O(n^2)$. 
A priori analysis of selection sort (1/2)

The quadratic runtime of selection sort could in fact have been predicted directly from the structure of the algorithm.

Consider the following, more explicit pseudocode presentation:

Algorithm 2: The selection sort algorithm

1. function SELECTSORT (A[1...n])
2. for $i \leftarrow 1$ to $n - 1$ do
3.     $i_{\text{min}} \leftarrow i$
4.     for $j \leftarrow i + 1$ to $n$ do
5.         if $A[j] < A[i_{\text{min}}]$ then $i_{\text{min}} \leftarrow j$
6.     end
7.     Exchange elements $A[i]$ and $A[i_{\text{min}}]$.  
8. end
A priori analysis of selection sort (2/2)

Assuming each “elementary operation” of the algorithm requires 1 time unit, then for each value of index $i$ ($i = 1 \ldots n - 1$) one obtains the computation times:

- Row 3: 1 time unit.
- Rows 4–6: $(n - i)$ time units.
- Row 7: 1 time unit.

The total computation time on an $n$-item input array can then be obtained by summing over all values of $i$:

\[
T(n) = \sum_{i=1}^{n-1} ((n - i) + 2) = \frac{1}{2} n^2 + \frac{3}{2} n - 2
\]

\[
= \mathcal{O}(n^2).
\]
1.3 Merge sort (1/2)

An alternative sorting method is the merge sort algorithm based on repeated partitioning of the input array. (In the following, assume for simplicity that \( n \) is of the form \( 2^k \) for some \( k = 0, 1, 2, \ldots \).)

**Algorithm 3:** The merge sort algorithm

1. **function** `MERGE_SORT(A[1 . . . n])`
2. If \( n = 1 \), there is nothing to do and the algorithm terminates.
3. If \( n \geq 2 \), partition \( A \) into two equal-sized halves: \( A' = A[1 \ldots \frac{n}{2}] \) and \( A'' = A[\frac{n}{2} + 1 \ldots n] \). Use the present method recursively to sort each of these.
4. When the half-arrays are sorted, *merge* them back together by picking items from each in increasing order.
Merge sort (2/2)

E.g. on input sequence $A = [3 \ 1 \ 4 \ 2]$ the algorithm proceeds as follows:

Figure: Schematic operation of merge sort.
A Python implementation of merge sort:

def merge_sort(A):
    n = len(A)
    if n == 1:
        return A
    else:
        h = n//2
        return merge(merge_sort(A[0:h]),
                     merge_sort(A[h:n]))
def merge(A, B):
    C = []
    m, n = len(A), len(B)
    i, j = 0, 0
    while (i < m) and (j < n):
        if A[i] <= B[j]:
            C.append(A[i])
            i = i+1
        else:
            C.append(B[j])
            j = j+1
    if i < m:
        C.extend(A[i:m])
    if j < n:
        C.extend(B[j:n])
    return C
Runtime of merge sort

![Runtime plot of merge sort](image)

**Figure:** Runtime plot of merge sort (x: elements, y: seconds).

Thus merge sort is *much* more efficient than selection sort! (Note the difference in scale in the plots.) Why is this?

Also, the runtime of the program now seems to grow approximately linearly in the number $n$ of items to be sorted. Is this so?
A priori analysis of merge sort (1/2)

Let $T(n)$ denote the time required by merge sort on an $n$-element input array $A[1 \ldots n]$. Assume for simplicity that $n$ is a power of 2, and each “elementary operation” requires 1 time unit.

It is easy to see that partitioning $A[1 \ldots n]$ and merging the two sorted halves back together requires $O(n)$ time units. Assume for simplicity that this time is exactly $n$.

Then the total computation time of merge sort is described by the following recurrence equation:

\[
T(1) = 1,
\]
\[
T(n) = 2T(n/2) + n, \quad \text{when } n = 2^k, k \geq 1.
\]
It can be shown (or verified using e.g. induction) that the solution to this recurrence is:

\[ T(n) = n \log_2 n + n. \]

Thus, the runtime of merge sort is not strictly linear, but of the order \( \Theta(n \log_2 n) \). As the experiments show, however, already for moderately large \( n \) this is significantly better than the runtime \( \Theta(n^2) \) of selection sort.
1.4 Fibonacci numbers

Let us consider another example where the choice of a proper algorithm leads to an even more dramatic improvement in efficiency.

The sequence of Fibonacci numbers \( F_0, F_1, \ldots \) is defined by the well-known recursion formula:

\[
F_n = \begin{cases} 
0, & \text{if } n = 0, \\
1, & \text{if } n = 1, \\
F_{n-1} + F_{n-2}, & \text{if } n \geq 2.
\end{cases}
\]

This sequence arises in a natural way in countless applications. The numbers grow at an exponential rate: \( F_n \approx 2^{0.694n} \).

(Exercise.)
It is tempting to use the definition of Fibonacci numbers directly as the basis of a simple recursive method for computing them:

**Algorithm 4: First algorithm for Fibonacci numbers**

```java
1 function FIB1 (n)
2   if n = 0 then return 0
3
4   if n = 1 then return 1
5
6   else return FIB1(n − 1) + FIB1(n − 2)
```

This is, however, not a good idea, because the computation of $F_{IB1}(n)$ requires time proportional to the value of the $F_n$ itself. (Verify this!)
The reason for the inefficiency is that the algorithm recomputes most of the values several (in fact, exponentially many) times:

Figure: Computation tree for Fibonacci numbers.
The recomputations can, however, be easily avoided by computing the values iteratively “bottom-up” and tabulating them:

**Algorithm 5: Improved algorithm for Fibonacci numbers**

```plaintext
function FIB2 (n)
if n = 0 then return 0
else
    Introduce auxiliary array F[0 ... n]
    F[0] ← 0; F[1] ← 1
    for i ← 2 to n do
        F[i] ← F[i − 1] + F[i − 2]
    end
    return F[n]
end
```

The computation time is now just $O(n)$ — a huge improvement!
Lecture 2: Algorithms and their analysis

▶ Introduction to the a priori analysis of algorithms and the big-O notation
▶ Examples: merge sort revisited, Fibonacci numbers, etc.
Lecture 3: Divide-and-conquer I

- *Divide-and-conquer* is one of the basic strategies in algorithm design. It operates by
  1. *breaking* the original problem into subproblems of the same type;
  2. *recursively solving* the subproblems; and
  3. *combining* the solutions of the subproblems to a solution of the original.

- E.g. merge sort

- We will look at further examples of divide-and-conquer algorithms and learn how to analyse their running time by solving recurrence relations.
Lecture 4: Divide-and-conquer II

- We continue our exploration of divide-and-conquer algorithms.
- In particular, we encounter the evaluation/interpolation paradigm for polynomial multiplication via the fast Fourier transform.
Lecture 5: Graph algorithms I

- Graphs are a fundamental and highly useful abstraction of many natural phenomena.
- A systematic search over all vertices in a graph is a fundamental task, and also a surprisingly powerful algorithmic primitive.
- In this first lecture on graph algorithms, we consider exploring a graph in the depth-first order.
- This deceptively simple idea leads to some quite subtle and surprising algorithms, such as a linear-time method for finding the strongly connected components of a graph.
In this second lecture on graph algorithms, we consider exploring a graph in the breadth-first order, and use this to find shortest paths between vertices, leading to the ubiquitous Dijkstra’s algorithm.
Many natural planning problems can be expressed as the task of maximising the flow of some quantity (water, traffic, electricity) on a capacity-constrained network.

Perhaps surprisingly, network flows are also among the most powerful general algorithmic primitives, and have a beautiful intrinsic structure leading to effective and unexpected algorithms.
Lecture 8: Greedy algorithms

- *Greedy algorithms* are one of the most fundamental algorithm design paradigms.
- The basic idea is to construct the solution to a problem instance as a sequence of locally optimal choices of components.
- In some cases this approach provably leads to globally optimal solutions, in other cases such “greedy” local choices may lead to an impasse later on, and force the algorithm to conclude with a suboptimal final result.
- Applications considered: minimum spanning trees in graphs, Huffman coding, set cover approximation
In *dynamic programming*, a problem is solved by identifying a collection of subproblems and tackling them one by one, smallest first, using the answers to smaller problems to solve larger ones, until all subproblems are solved.

We introduce this powerful design principle through several example problems:

- Longest increasing subsequences
- Edit distance
- Knapsack
- Chain matrix multiplication
- Shortest paths
- Travelling salesman problem
- Independent sets in trees
Lecture 10: NP-completeness

- So far in the course we have considered computational tasks for which we have been able to devise efficient algorithms using a variety of algorithm design techniques.
- However, there are computational problems for which all known algorithmic techniques scale exponentially in the worst case.
- In this lecture we look at examples of such problems and their relationships and develop an understanding of perhaps the most important class of such problems, i.e., that of NP-complete problems.
Lecture 11: Approximation algorithms

▶ Sometimes when exact solutions to a difficult (e.g. NP-complete) problem cannot be determined efficiently, it is still possible to produce good approximations, i.e. solutions which are guaranteed to be no more than $k$ or $(1 + \varepsilon)$ times worse than the optimum.

▶ The area is rich with interesting practical methodologies and deep theoretical questions.

▶ Examples considered: vertex cover, clustering, travelling salesman problem, knapsack
In terms of speed and simplicity it is in many cases advantageous to employ *randomness* in an algorithm; that is, certain decisions made by the algorithm are based not only on the given input but also *on outcomes of random coin tosses*.

A randomised algorithm is allowed to fail (e.g. report an incorrect answer), but only *with small probability*.

We look at examples of randomised algorithms and study their analysis.
Lecture 13: Algorithms with numbers I

Basic algorithms for computing modular arithmetic will be presented and their computational complexity will be determined. They will be applied to primality testing, which will provide the first examples of randomised algorithms discussed in this course.
Lecture 14: Algorithms with numbers II

The basic concepts of cryptography will be introduced and the algorithms for setting up an instance of the public key cryptosystem RSA will be presented. The lecture ends up with an algorithm for universal hashing which has applications in efficient storage systems.
Lecture 15: The fast Fourier transform

▶ The fast Fourier transform (FFT) illustrates how fundamental mathematics (in this case polynomial algebra and complex analysis), combined with efficient algorithm design techniques (divide-and-conquer) can lead to revolutionary improvements in performance.

▶ The FFT is also one of the all-time most influential algorithms in science and technology.
Lecture 16: Fundamental data structures

- Efficient implementation of algorithms often requires nontrivial ways of arranging the data the algorithm processes.
- The study of computationally efficient data structure is a broad subarea of algorithmics.
- Certain widely applicable fundamental structures have emerged over the years: e.g. trees, stacks and queues are already familiar from earlier courses.
- Here we present two further clever and surprisingly efficient data structures, which can also be used in the implementation of some algorithms earlier in the course:
  - Heaps
  - Union-find trees
Lecture 17: Epilog & exam information

- Summary of the course, Q & A
- Exam requirements
- Exam priming: discussion of problems from previous course exams