The theory of algorithms

- Interesting properties of algorithms in general
- For which sorts of problems do algorithms exist at all?
- Which algorithms are feasible in terms of their usage of physical resources?
- How to increase confidentiality in an algorithm’s correctness
Question:
Is there a job which a computer cannot do – a job for which no algorithm exists?
Question:
Is there a job which a computer cannot do – a job for which no algorithm exists?

Answer:
Yes there is. In fact, computers cannot do most things!
Question:
Is there a job which a computer cannot do – a job for which no algorithm exists?

Answer:
Yes there is. In fact, computers cannot do most things!

Historically, philosophers (Leibniz) and mathematicians (Hilbert) believed that the truth of philosophical arguments and mathematical theorems could be decided by computation.

In 1931, Gödel proved his famous incompleteness theorem, which refuted that belief.

That result was obtained before the first computers were built.
The Church-Turing thesis

- Before we can prove that no algorithm exists for a particular task, we must exactly know what we mean by an algorithm.
- We gave a definition earlier in this course, and Gödel, Church and Turing gave other definitions in the 1930s.
- Fact: All reasonable definitions of “algorithm” which are known so far are equivalent.
- Conjecture: Any reasonable definition of “algorithm” which anyone will ever make will turn out be equivalent to the definitions we know.
- Universality: For any programming language there exists an interpreter which, given the description of any program in the language, can simulate that program.
The halting problem

- The statement that no algorithm exists for solving a given problem is a very strong statement.
- It is much deeper than saying that we don’t know an algorithm for solving the problem.
- We say that we will never know such an algorithm!
- The most famous non-computable problem relevant for computer science is the halting problem.
- Its solution would be an algorithm which, given an arbitrary program $P$ and its input data $D$ can tell us whether or not $P$ would eventually halt when executed with input data $D$. 
Proof of non-computability of halting problem (1)

- Assume that there is an algorithm for solving the halting problem. Call that algorithm \textit{halttester}.
- Given $P$ and $D$, \textit{halttester}(P, D) prints “OK” if program $P$ would terminate when executed with input data $D$, and it prints “BAD” otherwise.

\begin{center}
\begin{tikzpicture}
\node (P) at (0,0) {$P$};
\node (D) at (2,0) {$D$};
\node (halt) at (1,-1) {Does $P(D)$ halt?};
\node (yes) at (0,-2) {Yes};
\node (no) at (2,-2) {No};
\node (ok) at (0,-3) {output "OK" and halt};
\node (bad) at (2,-3) {output "BAD" and halt};
\draw[->] (P) -- (halt); 
\draw[->] (D) -- (halt); 
\draw[->] (yes) -- (ok); 
\draw[->] (no) -- (bad); 
\end{tikzpicture}
\end{center}
Proof of non-computability of halting problem (2)

More limited problem: Does $P(P)$ halt?

Since a program is just a sequence of characters, it can be used as an input for some other program, or the same program.

Call that algorithm \textit{newhalttester}. 

Proof of non-computability of halting problem (3)

Specification: let $P$ be a sequence of characters.

| pre       | $P \neq \langle \rangle$ |
| post      | $\text{newhalttester}(P) = "OK" \iff "P(P) halts"$ and $\text{newhalttester}(P) = "BAD" \iff \text{not } "P(P) halts"$ |
| reads     | $P$ |
| changes   | - |
| mem       | - |

Does $P(P)$ halt?

Yes: output "OK" and halt

No: output "BAD" and halt
If we assume that an algorithm `halttester` exists, and that therefore `newhalttester` exists, we may construct the following algorithm `funny`, which has just one input `P`.

```plaintext
module funny(P : string)
    if newhalttester(P) = "BAD"
        then "halt"
        else "loop forever"
    endif
endmodule
```

Does `P(P)` halt?

- Yes
- No

`P`

`halt`

(loop forever)
Finally, we consider what happens during execution of $\textit{funny}(\textit{funny})$.

```
funny
Does funny(funny) halt?
  Yes
  loop forever
  No
  halt
```

This is clearly a contradiction, since it shows on the one hand that if it halts then it loops forever, but on the other hand that if it loops forever it halts. The contradiction can only be resolved by admitting that the algorithm $\textit{funny}$, and hence $\textit{halttester}$, cannot exist.
Summary of proof

1. Assume that we can write a program \textit{halttester}.
2. Use it to construct another program \textit{funny} (via an intermediate program \textit{newhalttester}).
3. Show that \textit{funny} has some impossible property (it can neither halt nor loop forever).
4. Conclude that the assumption in step 1 must be wrong.

Hence, there is no purely mechanical method which is guaranteed to solve the halting problem in every case, even though for concrete algorithms, we can (and must) show that they terminate. There is a difference between a routine algorithm and human creativity in the solution of a problem!
Summary of proof

1. Assume that we can write a program *halttester*.
2. Use it to construct another program *funny* (via an intermediate program *newhalttester*).
3. Show that *funny* has some impossible property (it can neither halt nor loop forever).
4. Conclude that the assumption in step 1 must be wrong.

- Hence, there is no purely mechanical method which is guaranteed to solve the halting problem in every case, even though for concrete algorithms, we can (and must) show that they terminate.
- There is a difference between a routine algorithm and human creativity in the solution of a problem!
More noncomputable problems

To show non-computability for other problems, we show that if the problem at hand were computable then so would the halting problem.

**Totality problem**: Will an arbitrary program $P$ halt on all inputs? We say that $P$ is *total* if $P(D)$ halts for all input data $D$.

If the totality problem were computable, we could define a module *funnypd*:

```plaintext
module funnypd(I : arbitrary_type)
{I is the input, but the module does not use it}
"simulate program P on input data D"
endmodule
```
Now, asking if *funnypd* is total is the same as asking whether $P(D)$ halts. So, an algorithm for the totality problem would give us the following algorithm for the halting problem:

```plaintext
module halttester(P : string, D : arbitrary_type)
  "construct the text of the program funnypd";
  if total(funnypd)
    then "output "OK" and halt"
  else "output "BAD" and halt"
endif
endmodule
```

However, we know that no algorithm for the halting problem exists. Hence, the assumption that there exists an algorithm for the totality problem is false. We have therefore proved that the totality problem is not computable.
Equivalence problem (1)

The problem to determine if two programs perform the same job (i.e. for the same input they produce the same output), is not computable.

Otherwise, given any $P$, would could construct a program that simulates $P$ and outputs “13” if $P$ eventually halts:

```
module funnyp(D : arbitrary_type)
{Output is “13” if $P(D)$ ever halts}
“simulate program $P$ on input data $D$”;
{The previous step may loop for ever, but if it doesn’t then . . .}
output(13)
endmodule
```
Now, consider a simple-minded program that always outputs “13”:

```
module simple(D : arbitrary_type)
{D is the input, but the module ignores it and outputs “13” regardless}
output(13)
endmodule
```

Now, asking whether `funnyp` is equivalent to `simple` is the same as asking whether `P` is total, because, if `P` halts on every input, then `funnyp` will output “13” on every input. However, if there is some input on which `P` does not halt, then `funnyp` will not halt and will not output “13” on that input.

**Exercise:** construct a module `total(P)` for the totality problem on the assumption that there exists an algorithm solving the equivalence problem.
Some noncomputable problems are less computable than others.

The halting problem is **partially computable**, because there exists an algorithm which outputs “Yes” if $P(D)$ halts, though it loops for ever if $P(D)$ does not halt.
The totality and equivalence problems are not partially computable.

Problems are partially computable if they have a proof system, i.e., a set of axioms together with some rules of inference.

A proof is just a sequence of lines, the early ones being axioms and the remainder following from previous ones by using the rules of inference.

In particular, if a program does halt on some input data, it is possible to write down a convincing proof of that fact.

In summary, problems can be computable, partially computable, or not even partially computable.
What have we learnt about computability?

- Not all problems are solvable by an algorithm.
- A famous non-computable problem is the **halting problem**.
- Proving that a problem is non-computable is done by **contradiction**.
- Other non-computable problems can be **reduced** to the halting problem or other problems known to be non-computable.
- There are different degrees of non-computability: some problems are **partially** computable, others not even that.
Computability vs. complexity

**Computability** leads to an understanding of which problems admit algorithmic solution, and which do not.

**Complexity**: Of those problems for which there do exist algorithms, it is of interest to know how much computer resources are required for their execution. Only those algorithms which use a feasible amount of resources are useful in practice.

**Complexity theory** deals with questions about computational resources.
Although the set of feasibly computable problems is only a small portion of all computable problems (which in turn are only a small portion of all problems) that set is so large that computer science has become an interesting, practical, and flourishing science.
Computer resources

**Time** elapsed period from start to finish of algorithm execution

**Memory** amount of storage required by algorithm

**Hardware** amount of physical mechanism required for execution of algorithm (e.g., number of processors for parallel algorithms)

The amount of any resource used by an algorithm may **vary** with the size of the input data.
Multiplication algorithms (1)

If the “junior school algorithm” is presented with two \( n \)-digit numbers, it adds together \( n \) rows, each containing \( n \) (or \( n + 1 \)) digits.

Each row can be computed in \( n \) steps, and there are \( n \) separate rows.

Thus all rows can be computed in \( n \times n \) (or \( n^2 \)) units of time.

Adding the rows also takes \( n^2 \) units of time.

Hence the execution time of the whole algorithm is proportional to \( n^2 \).
Multiplication algorithms (2)

A faster algorithm:

\[
\begin{array}{cc}
A & B \\
19 & 84 \\
\end{array}
\times
\begin{array}{cc}
C & D \\
67 & 13 \\
\end{array}
\]

\[
AC = 19 \times 67 = 1273
\]
\[
(A + B)(C + D) - AC - BD = (103 \times 80) - 1273 - 1092 = 5875
\]
\[
BD = 83 \times 13 = 1092
\]
\[
\text{This algorithm takes time proportional to } n^{1.59}
\]

The fastest known algorithm for a sequential machine has execution time proportional to \( n \log n \log \log n \) (\( \log n \) meaning the logarithm of \( n \) using the base 2).
Important points

- A problem can be solved by quite different algorithms which perhaps use different amounts of resources.
- It is interesting to find the best algorithm, which uses the least resources.
- Often, as one attempts to reduce some resource, some other resource must necessarily be increased (trade-off).

Moreover:

- The amount of resource used depends on the size of the input data.
  The more digits there are in the numbers to be multiplied, the longer it will take to perform the multiplication.
- In general, if there are $n$ characters of input data, we can express the amount of resource used as a function of $n$, such as $3n^2 + 5n$, or $2n \log n + n + 17$. 
Asymptotic behavior

- As $n$ grows larger, some term in the function which expresses the amount of resource used may begin to dominate the other terms.
- For example, if the execution time is $3n^2 + 5n$, then as $n$ grows larger, $3n^2$ grows very much larger than $5n$, and therefore $5n$ becomes increasingly less significant.
- The dominating term $3n^2$ (or even $n^2$) is called the asymptotic behavior of the algorithm.
- Usually, we only consider the asymptotic behavior of algorithms. That behavior ultimately governs whether or not a particular algorithm is feasible.
### Examples of execution times

Assumption: one step takes one microsecond.

<table>
<thead>
<tr>
<th>Size of input</th>
<th>$\log_2 n$</th>
<th>$n$</th>
<th>$n^2$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.000003 s</td>
<td>0.00001 s</td>
<td>0.0001 s</td>
<td>0.001 s</td>
</tr>
<tr>
<td>100</td>
<td>0.000007 s</td>
<td>0.0001 s</td>
<td>0.01 s</td>
<td>$10^{14}$ centuries</td>
</tr>
<tr>
<td>1000</td>
<td>0.00001 s</td>
<td>0.001 s</td>
<td>1 s</td>
<td>astronomical</td>
</tr>
<tr>
<td>10000</td>
<td>0.000013 s</td>
<td>0.01 s</td>
<td>1.7 min</td>
<td>astronomical</td>
</tr>
<tr>
<td>100000</td>
<td>0.000017 s</td>
<td>0.1 s</td>
<td>2.8 h</td>
<td>astronomical</td>
</tr>
</tbody>
</table>

- The $2^n$ algorithm is infeasible even for relatively small sizes of input data.
- The $n^2$ algorithm is much better, but will also use too much time for large input data.
- The complexity of an algorithm can hardly be less than linear, because an algorithm must at least look through the input data.
Algorithms whose asymptotic behavior is $2^n$, or more general $c^n$ for some constant $c$, are called exponential algorithms. Exponential algorithms are infeasible for all but the smallest input data sizes.

Algorithms whose asymptotic behavior is $n^c$ for some constant $c$ are called polynomial algorithms. Many polynomial algorithms are feasible for practical input data sizes, but, unfortunately, many are not.
Different kinds of complexity

For a given input size, there are many different inputs, and it is not unusual that an algorithm uses different amounts of resources on different input data, even when the different inputs have the same size (see for example the bubble sort algorithm).

**Worst-case complexity:** maximum amount of resource required by the algorithm for a given input size.

**Average-case complexity:** average amount of resource used over all the inputs of a given size.

For some algorithms, the worst-case complexity is decisive for their feasibility (e.g., control algorithms).

For others, an acceptable average-case complexity suffices (e.g., Quicksort, a sorting algorithm with worst-case complexity $n^2$ and average-case complexity $n \log n$).
Complexity of *problems*

- So far: complexity of *algorithms*.
- Complexity of a *problem*: complexity of the best algorithm that solves the problem.
- The complexity of many problems is unknown, because we only know the complexity of the best algorithm found *so far*.
- That complexity constitutes an *upper bound* on the complexity of the problem.
- **Lower bound**: least amount of resource needed by every algorithm solving the problem (often $n$).
- For integer multiplication, a lower bound is $n$ and an upper bound is $n \log n \log \log n$. Hence, the exact complexity of the problem lies between those two.
Divide and conquer

Common and elegant method of devising an efficient algorithm: divide the problem into smaller pieces, thus leaving only smaller problems to be solved.

Example: Mergesort

```
module sort(a : array of string, n : integer)
{Sorts given array a of n names into alphabetical order}
if n > 1
    then merge(sort(left(a)), sort(right(a)))
endif
endmodule
```

What is the time complexity of this algorithm? (I.e., how long does the algorithm take to sort n names?)
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
Recurrence relations

- Let \( T(n) \) be the time taken by the algorithm to sort \( n \) names.
- The algorithm begins by sorting the first half of the names. This will take \( T(n/2) \).
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$) (Exercise!).
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$) (Exercise!).
- Thus, the total time taken is $T(n) = 2T(n/2) + cn$
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$)(Exercise!).
- Thus, the total time taken is $T(n) = 2T(n/2) + cn$
- Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$) (Exercise!).
- Thus, the total time taken is $T(n) = 2T(n/2) + cn$
- Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test $(n > 1)$ need be carried out.
- Such equations are called recurrence relations.
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$)(Exercise!).
- Thus, the total time taken is $T(n) = 2T(n/2) + cn$
- Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.
- Such equations are called \emph{recurrence relations}.
- The above recurrence relation has the solution $T(n) = cn \log n + kn$. 

Let $T(n)$ be the time taken by the algorithm to sort $n$ names.

The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.

Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$)(Exercise!).

Thus, the total time taken is $T(n) = 2T(n/2) + cn$

Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.

Such equations are called \emph{recurrence relations}.

The above recurrence relation has the solution $T(n) = cn \log n + kn$. 

Let $T(n)$ be the time taken by the algorithm to sort $n$ names.

The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.

Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$)(Exercise!).

Thus, the total time taken is $T(n) = 2T(n/2) + cn$

Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.

Such equations are called \emph{recurrence relations}.

The above recurrence relation has the solution $T(n) = cn \log n + kn$. 

Let $T(n)$ be the time taken by the algorithm to sort $n$ names.

The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.

Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$)(Exercise!).

Thus, the total time taken is $T(n) = 2T(n/2) + cn$

Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.

Such equations are called \emph{recurrence relations}.

The above recurrence relation has the solution $T(n) = cn \log n + kn$.
Recurrence relations

- Let $T(n)$ be the time taken by the algorithm to sort $n$ names.
- The algorithm begins by sorting the first half of the names. This will take $T(n/2)$.
- Merging the two halves together can be carried out in time proportional to $n$ (say $cn$ for some constant $c$) (Exercise!).
- Thus, the total time taken is $T(n) = 2T(n/2) + cn$
- Also $T(1) = k$ for some constant $k$, since when $n = 1$ only the test ($n > 1$) need be carried out.
- Such equations are called recurrence relations.
- The above recurrence relation has the solution $T(n) = cn \log n + kn$.
- Hence, the asymptotic execution time of the merge sort algorithm is proportional to $n \log n$. 
More recurrence relations

- Consider the multiplication algorithm of slide 197.
- This algorithm uses a divide and conquer approach, since in order to multiply two numbers it splits each into halves \((A, B \text{ and } C, D)\), and then performs three multiplications of numbers only half the original length: \(AB, BC\) and \((A + B)(C + D)\).
Consider the multiplication algorithm of slide 197.

This algorithm uses a divide and conquer approach, since in order to multiply two numbers it splits each into halves \((A, B)\) and \((C, D)\), and then performs three multiplications of numbers only half the original length: \(AB\), \(BC\) and \((A + B)(C + D)\).

If \(T(N)\) is the time taken by the algorithm to multiply two \(n\) digit numbers, then \(T(n) = 3T(n/2) + cn\) because the time needed to add and subtract the pieces is only \(cn\).
More recurrence relations

- Consider the multiplication algorithm of slide 197.
- This algorithm uses a divide and conquer approach, since in order to multiply two numbers it splits each into halves $(A, B$ and $C, D)$, and then performs three multiplications of numbers only half the original length: $AB$, $BC$ and $(A + B)(C + D)$.
- If $T(N)$ is the time taken by the algorithm to multiply two $n$ digit numbers, then $T(n) = 3T(n/2) + cn$ because the time needed to add and subtract the pieces is only $cn$.
- This recurrence relation has the solution $T(n) = (2c + k)n^{\log_3 3} - 2cn$. 
More recurrence relations

- Consider the multiplication algorithm of slide 197.
- This algorithm uses a divide and conquer approach, since in order to multiply two numbers it splits each into halves \((A, B\) and \(C, D)\), and then performs three multiplications of numbers only half the original length: \(AB, BC\) and \((A + B)(C + D)\).
- If \(T(N)\) is the time taken by the algorithm to multiply two \(n\) digit numbers, then \(T(n) = 3T(n/2) + cn\) because the time needed to add and subtract the pieces is only \(cn\).
- This recurrence relation has the solution \(T(n) = (2c + k)n^{\log_3 3} - 2cn\).
- Thus the asymptotic behavior is proportional to \(n^{\log_3 3}\), which is about \(n^{1.59}\).
Advantages of divide and conquer algorithms

- Divide and conquer can sometimes be used to derive interesting, elegant and efficient algorithms.
- Recurrence relations express the resource usage of such algorithms, and the solution of the recurrence relation gives the resource usage in a clearer form.
- The algorithms themselves are best expressed using the recursion feature of a modern programming language.
Feasible vs. infeasible algorithms

- Aim of complexity theory: improve classification of algorithms, thereby improving our understanding of the difference between feasible and infeasible problems.

- Polynomial algorithms *tend* to be feasible for reasonable sizes of input data, even though many of them are infeasible, such as one which takes $5n^{1000}$ hours to solve a problem.

- Conversely, exponential algorithms tend to exceed the available resources even for small amounts of input data. Again, there are exceptions. An algorithm which uses $2^{0.0001n}$ seconds may be quite reasonable for values of $n$ up to 10000.
First approximation to distinguish between feasible and infeasible algorithms: say that algorithms executing in a polynomial amount of time are feasible, and all others are infeasible.

Attractive approximation because polynomial algorithms have a number of closure properties:

- Sequential composition of two feasible algorithms yields a feasible algorithm.
- If a step in a feasible algorithm is replaced by a call to a module representing a second feasible algorithm, the new algorithm is also feasible.
Moreover, the concept of a feasible algorithm should not depend on the details of the computer on which the algorithm will be executed.

For the polynomial algorithms, this is the case, because

- Not only can all reasonable sequential computers simulate each other, but the time losses associated with the simulations are not excessive.
- Any algorithm which executes in polynomial time on one computer can be run in polynomial time on any other computer.

Hence, a theory of feasible algorithms based on polynomial time is machine independent.
The sequential computation thesis says that all reasonable sequential computers which will ever be dreamed of have polynomially related execution times.

This is stronger than the Church-Turing thesis, because it claims not only that the computable problems are the same for all computers, but also that the feasibly computable problems are the same for all computers.

In summary, the best definition of to date is that the feasibly computable problems are those which have polynomial time (sequential) algorithms.

This definition matches practical experience quite well, and seems to be machine independent.
Infeasible problems (1)

- Following the previous discussion, it is of great interest to know which problems are computable in polynomial time.

- Many interesting and practical problems are in this category, including most of the problems discussed in the chapter “Design of Algorithms”.

- Some problems have proved not to be polynomial time computable. Such proofs are usually very difficult, as it is necessary to show that no algorithm that runs in polynomial time can solve the problem.

- An example of such a problem is a generalization of the game of chess to an \( n \times n \) board.
There are many problems for which the existence of a fast (i.e. polynomial time) algorithm remains an open question. No fast algorithms have been discovered for such problems, but neither has anyone yet been able to prove that no such algorithm exists.

Many of these problems have practical application, as well as presenting an interesting challenge to computer scientists.

An example of such a problem arises in connection with trucking companies.
Suppose a trucking company has a number \( N \) of crates of different weights.

The problem is to determine if \( T \) trucks, each of which can carry a load \( W \), are sufficient to transport the crates.

\[
\begin{align*}
\text{N = 6 crates} \\
\text{T = 2 trucks} \\
\text{maximum load } W = 14 \text{ tonnes}
\end{align*}
\]
Clearly, there are also many other applications in which the bin-packing problem arises.

Unfortunately, every known algorithm for this problem uses exponential time.

One might at first think of repeatedly loading the largest crate which still fits on to a truck until no more creates fit, and then proceeding to the next truck. The figure illustrates that this algorithm is not guaranteed to work.

![Diagram](image-url)
This is another problem which has puzzled operations research experts and computer scientists for years.

Given a roadmap of \( N \) cities, is it possible for a salesperson to complete a round trip within a given mileage allowance, visiting each city exactly once?

A related problem is to determine whether a round trip which visits each city exactly once is possible on the given map.

For both problems, no polynomial time algorithm has ever been found, nor has anyone been able to prove that no such algorithm exists.
This is also a problem of great practical interest for which no polynomial time algorithm is currently known.

In this problem, one is given a list of subjects and the students enrolled in them, as well as the number of time slots available.

The problem is to timetable the subjects so that no student has a clash.
Dealing with infeasible problems (1)

- What should computer scientists do when a computer solution to any of these problems is required?
- It is necessary to lower our sights in one of the following ways:
  - Rather than to seek an algorithm which provides an exact solution, it will sometimes be adequate to provide an approximate solution only.
  - For example, in the timetabling problem, a solution with only a few clashes may be better than no solution at all.
  - For the traveling salesperson problem, a fast algorithm that produces a reasonably short trip is better than using an infeasible amount of computer time to calculate the absolutely minimal mileage round trip.
Dealing with infeasible problems (2)

Other ways of lowering one’s sight:

- Try to produce an algorithm which executes quickly on the average input data, but which nevertheless exhibits an exponential behavior in the worst case. The hope is that the algorithm will terminate reasonably quickly on most of the data which it encounters in practice.
- Relax the condition that the algorithm must be correct. This may seem bizarre, but it may be useful in certain circumstances to try to produce a fast algorithm which is known to contain an error. Cute example: primality problem.
Given an $n$-digit number as input, the problem is to determine whether or not the number is prime.

An obvious algorithm is to divide the input number by every smaller number and check if the remainder is ever zero.

This is an exponential algorithm and becomes quite infeasible even for moderate values of $n$, such as 20 or 30 digits.

The algorithm that results from dividing only by prime numbers, and then only up to the square root of the input number is slightly faster but still infeasible.
Consider the following mathematical fact concerning any odd number \( x \). If \( x \) is prime, then every number \( y \) between 1 and \( x - 1 \) satisfies the following condition:

\[
\text{GCD}(x, y) = 1 \text{ and } y^{(x-1)/2} \mod x = J(y, x)
\]

where \( \text{GCD} \) is the greatest common divisor and \( J \) is called the Jacobi function. That function can be computed quickly.

It turns out that if \( x \) is not prime, then at least half the possible values of \( y \) will fail to satisfy the condition.

Thus the following algorithm can be used to check if \( x \) is prime.
If $x$ is prime then “PRIME” is output. If $x$ is not prime then “NOT PRIME” will probably be output, but there is a very small chance of the algorithm outputting “PRIME” erroneously.

```plaintext
module prime(x : integer)
repeat e times
    “randomly pick a number y between 1 and x-1”;
    if gcd(x,y) ≠ 1
        then {output “NOT PRIME” and halt}
            if \( y^{(x-1)/2} \mod x \neq J(y,x) \)
                then {output “NOT PRIME” and halt}
    endif
endif
endrepeat
{output “PRIME”}
endmodule
```
Analysis of $\text{prime}(x)$

- If $x$ is prime then, assuming the mathematical fact given above, it is easy to see that the algorithm outputs “PRIME”.
- $e$ is a constant such as 20 or 30.
- If $x$ is not prime, then each time around the loop there is at least a one in two chance that the randomly chosen $y$ either satisfies $GCD(x, y) \neq 1$ or $y^{(x-1)/2} \mod x \neq J(y, x)$.
- Therefore, after $e$ times around the loop, the chance of not finding an appropriate $y$ is less than one in $2^e$.
- If $e$ is 20, the chance is less than one in a million.
- If $e$ is 30, the chance is less than one in a billion.
- Therefore, if $x$ is not prime then the algorithm will probably output “NOT PRIME”, the chance of making an error being less than one in $2^e$. 

Probabilistic algorithms

- This type of algorithm – which may make an error, but very infrequently – is called a probabilistic algorithm.
- These algorithms are quite practical, because the constant $e$ can be adjusted to make the probability of an error as small as desired.
- If $e$ is around 40, then the algorithm is probably more reliable than the computer on which it is being executed.
For the problems described in the previous section, such as the bin-packing problem, the traveling salesperson problem, and the timetabling problem, no polynomial algorithms are currently known.

Given arbitrary values of the inputs, or an arbitrary instance of the problem, there seems to be no fast method of finding the solution.

However, these problems have a very interesting property in common:
Once a solution is found for a particular instance of a problem (e.g. a particular timetable which works is discovered), it is easy to verify that the solution is correct.

More precisely, for each of these problems there exists an algorithm which, given a particular instance of the problem and a proposed solution, can verify in polynomial time whether or not the solution is correct.
Examples of correct solutions

3 tonnes

3 tonnes

8 tonnes

4 tonnes

5 tonnes

5 tonnes

4 tonnes

5 tonnes

5 tonnes

8 tonnes
The set of problems which have a fast verification algorithm as described above is called NP (nondeterministic polynomial).

All feasible problems are in NP.

NP contains also many of the open problems which different disciplines have been trying to solve for many years.
NP-complete problems

- **NP-complete** problems are the *hardest* problems in NP.
- If an algorithm which runs in polynomial time were ever found for any one of these problems, then there would be a polynomial time algorithm for *every* problem in NP.
- The bin-packing, the traveling salesperson, Hamiltonian cycle, timetabling and many other problems are each NP-complete.
- There are literally thousands of well-known, naturally occurring and practically interesting problems which have been proved to be NP-complete.
- It follows from the definition of NP-completeness that all NP-complete problems are computationally equivalent.
Infeasibility of NP-complete problems

- It is widely believed that NP-complete problems are infeasible.

- One reason to believe this is that many different people have tried to find polynomial algorithms, but all have failed.

- Moreover, we experience the same dichotomy in real life between problems which are easy to solve and those which are hard to solve but whose solution appears obvious once it is found.

- Example: Given green leaves and a plastic sheet, how do you get water?
Old scouting trick for collecting water by condensation

- Hole in ground
- Container
- Plastic sheet
- Leaves
- Condensation about to drip
If one is having difficulty finding a polynomial time algorithm for a problem, then one can alternatively check if the problem is NP-complete.

In this case, one should seek a method of lowering one’s sights.

Unfortunately, there are problems for which no fast algorithm is known, but neither does the problem appear so difficult that it can be proved NP-complete. An example is the primality problem.
Feasible problems have a fast algorithm for finding their solutions.

Problems in NP may or may not have such an algorithm, but at least proposed solutions are easy to verify.

NP-complete problems are the hardest problems in NP, and it is widely believed that they do not have fast algorithms.
Many problems that are computable are nevertheless **infeasible**, because the algorithms solving the problems use too much resources.

Possible resources are time, memory and hardware.

The **complexity** of an algorithm is the amount of resources used.

The resources used are expressed in terms of a function on the size of the input of an algorithm.

The **asymptotic behavior** of an algorithm results from the dominating term of the complexity function.

**Polynomial** algorithms are often feasible, while **exponential** ones are clearly infeasible.
We distinguish between **worst-case** and **average-case** complexity of an algorithm.

The complexity of a problem is the complexity of the best algorithm solving the problem.

That algorithm may be unknown.

Hence, we consider **lower** and **upper bounds** for the complexity of a problem.
What have we learnt about complexity? (3)

- The **divide-and-conquer** principle often leads to efficient algorithms.
- The complexity of recursive algorithms is determined by solving **recurrence relations**.
- The **sequential computation thesis** supports the hypothesis that the class of feasible algorithms are the polynomial algorithms.
What have we learnt about complexity? (4)

- Many practically relevant problems are judged to be infeasible.
- Among them are the traveling salesman problem, the bin-packing problem, and the timetabling problem.
- In order to deal with infeasible problems, one can look for approximate solutions, an algorithm which executes quickly on the average input data, or a probabilistic algorithm.
What have we learnt about complexity? (5)

- The problems in **NP** are characterized by the fact that a proposed solution can be verified in polynomial time.
- **NP-complete** problems are the hardest ones in NP. If for one of them, a polynomial algorithm is found, then $P = NP$. 