Algorithms and Data Structures (AED — Algoritmos e Estruturas de Dados) LEI, MIEC, 2017/2018 (40437)

— TP.01 —

Summary:

- What AED is all about
- Rules of the game
- List of planned lectures
- Recommended bibliography for the entire course
- The C programming language

Teachers:

- **TP1**, **P1**, and **P2**: Tomás Oliveira e Silva, `tos@ua.pt`, DETI 4.2.37
- **P4**: Óscar Pereira, `omp@ua.pt`, IT

When and where:

- **TP1** room Anf. V (DETI), Friday, 9h00m-11h00m
- **P1** room 4.2.07, Thursday, 9h00m-11h00m
- **P2** room 4.2.07, Thursday, 11h00m-13h00m
- **P4** room 4.2.03, Thursday, 16h00m-18h00m

Recommended bibliography for this lecture:

- **C in a Nutshell, A Desktop Quick Reference**, Peter Prinz and Tony Crawford, O'Reilly, 2006.
- **C, A Reference Manual**, Samuel P. Harbison III and Guy L. Steele Jr., fifth edition, Prentice Hall, 2002.
- **C Programming (A Comprehensive Look at the C Programming Language and Its Features)**, wiki book.
- **The C Programming Language**, Brian W. Kernighan and Dennis M. Ritchie, second edition, Prentice Hall, 1988.

This document was last updated on November 23, 2017.

How to navigate these lecture notes

Links to other parts of this document and to other documents are displayed in dark orange.

To avoid an excessive use of that color, in the summary of each lecture the links to the various parts of the lecture are located in the filled dark orange circles (●).

At the bottom of each page, on the right hand side, there are links that go

- to the first page of this document (AED link)
- to the first page of the current lecture (TP.N or P.N links)
- to the previous lecture (◀)
- to the next lecture (▶)

The list of planned lectures pages has links to all the other lectures. The first page of this document has a direct link to that page.

What AED is all about

Program = Algorithm + Data Structures

$$\textbf{Good Program} = \begin{cases} \textbf{Algorithm + Good Data Structures} \\ \quad \textbf{or} \\ \textbf{Good Algorithm + Data Structures} \end{cases}$$

Very good Program = Good Algorithm + Good Data Structures

Exceptional Program = State-of-the-art Algorithm + State-of-the-art Data Structures

A good algorithm/data structure has a “small” (i.e., as small as theoretically possible) computational complexity.

The computational complexity measures the time or memory resources (space) needed to run the program.

A state-of-the-art algorithm/data structure does the job better than any other algorithms/data structures available to solve the same problem.

It may be advantageous to trade time for space (or to trade space for time).

A good programmer knows many good algorithms and data structures (and knows where to look for more), and is capable of determining which ones are best for the job at hand.

An exceptional programmer is capable of devising new algorithms or data structures to solve a new problem.

Example: computing Fibonacci numbers

The Fibonacci numbers are defined by the recursive formula

$$F_n = F_{n-1} + F_{n-2}, \qquad n > 1,$$

with initial conditions $F_0 = 0$ and $F_1 = 1$. We present below some possible ways to compute F_n in C (without detection of bad inputs or of arithmetic overflow):

- **Recursive implementation:**

```
int F_v1(int n)
{
  return (n < 2) ? n : F_v1(n - 1) + F_v1(n - 2);
}
```

- **"Memoized" recursive implementation:**

```
int F_v2(int n)
{
  static int Fv[50] = { 0,1 };

  if(n > 1 && Fv[n] == 0)
    Fv[n] = F_v2(n - 1) + F_v2(n - 2);
  return Fv[n];
}
```

- **Non-recursive implementation:**

```
int F_v3(int n)
{
  int a,b,c;

  if(n < 2)
    return n;
  for(a = 0,b = 1;n > 1;n--)
  {
    c = a + b; // c = F(n-2) + F(n-1)
    a = b;     // a = F(n-1)
    b = c;     // b = F(n)
  }
  return b;
}
```

- **"Clever" implementation (Binet's formula):**

```
int F_v4(int n)
{
  const double c1 = 0.44721359549995793928;
  const double c2 = 0.48121182505960344750;
  return (int)round(c1 * exp((double)n * c2));
}
```

Note that $F_n = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$.

Rules of the game (part 1)

Grading in AED will abide by the following rules:

- Grading has two components: theoretical part, G_1, and computer lab part, G_2, with $0 \leqslant G_1, G_2 \leqslant 20$.
- A grade below 7.5 in either of the two parts means course failure (RNM).
- Presence in the theoretical lectures is not mandatory (but it is strongly recommended).
- Presence in the computer lab classes is mandatory for ordinary students. Failure to attend more than N computer lab classes means course failure (RPF), without possibility of attending the recourse epoch, N being equal to 3 for ordinary students and to ∞ for working students.
- The tentative final grade G is given by $G = \mathrm{round}\left(\frac{G_1+G_2}{2}\right)$. If $G \leqslant 16$, then G will be the final grade. Otherwise, the final grade will also take in consideration one oral presentation, to be done on the first two weeks of January 2018, of an extra individual work assigned to each eligible student. In that case the final grade will be $\geqslant 16$ and $\leqslant 20$.
- The G_1 grade can be obtained in one of two ways: either during the semester as the weighted average of three tests to be performed in the first hour of some theoretical lectures, or by a final exam. In the first case, $G_1 = 0.40t_1 + 0.35t_2 + 0.25t_3$, where t_1 is the **best** grade and t_3 is the **worst** grade. Each student has to chose by October ?? which of the two choices she/he desires. It is **very highly recommended** that the students take the **first** choice.
  For example, if the grades of the three tests are 14, 17, 12, then
  $$G_1 = 17 \times 0.40 + 14 \times 0.35 + 12 \times 0.25 = 14.7$$
  The final exam will have a duration of at least 3 hours.

Rules of the game (part 2)

- The G_2 grade is the weighted average of three reports of work done in the classroom (or outside classes for working students); $G_2 = 0.40p_1 + 0.35p_2 + 0.25p_3$, where p_1 is the **best** grade and p_3 is the **worst** grade. Each report will be graded based on

  1. clarity of exposition,
  2. quality of the results obtained,
  3. code quality,
  4. originality,
  5. amount of work done in the practical class, and
  6. amount of work done **before** the practical class.

  Plagiarism will be severely punished. Each report can be done by groups of at most 3 students. Grades may be different for the students of each group, according to

  1. how much each contributed to the work (stated in the report), and
  2. the teacher's perception of how much each student appeared to work.

  This example gives an idea of how a report should be structured.
- In the recourse and special epochs the G_2 grade is the grade of a report of the work done in an all day (8 hours) computer lab session.
- Students wishing to raise their grades must do so either in the recourse epoch or in the next school year.

List of planned lectures (part 1)

Class	Summary
15-09-2017 TP.01	What AED is all about. Rules of the game. The C programming language (overview, preprocessor directives, comments, data types, declaration, definition, and scope of variables).
22-09-2017 TP.02	The C programming language (assignments and expressions, statements, functions, and standard library functions). The C++ programming language (classes, templates, and exceptions).
29-09-2017 TP.03	Algorithms. Abstract data types. Computational complexity. The RAM model of computation. Formal and empirical algorithm analysis. Worst, best, and average cases. Asymptotic notation ($O()$, $o()$, $\Omega()$, and $\Theta()$). Examples.
06-10-2017 TP.04	Classes of problems. Polynomial-time algorithms and non polynomial-time algorithms. Analysis of the time/space needed by an algorithm. Examples (for non-recursive and for recursive algorithms).
13-10-2017 TP.05	**Test (TP.01 to TP.04).** Elementary data structures, part 1. Arrays. Linked lists (singly- and doubly-linked). Stacks.
20-10-2017 TP.06	Elementary data structures, part 2. Queues. Circular buffers. Heaps. Priority queues. Binary trees (unordered, ordered, and balanced). Tries. Hash tables.
27-10-2017 TP.07	Searching unordered data in i) an array, ii) a linked list, iii) a binary tree. How to improve search times. Searching ordered data in i) an array, ii) a binary tree. Sorting, part 1. Bubble sort and shaker sort. Insertion and Shell sort.
03-11-2017 TP.08	Sorting, part 2. Quicksort. Merge sort. Heap sort. Other sorting methods (count sort, rank sort, selection sort, bitonic sort). Algorithmic techniques: divide-and-conquer and dynamic programming. Examples.
10-11-2017 TP.09	**Test (TP.05 to TP.08, without topics about algorithmic techniques).** Finding all possibilities (exhaustive search), part 1. Depth-first search. Breadth-first search. Backtracking. Pruning.
17-11-2017 TP.10	Finding all possibilities (exhaustive search), part 2. Examples. Graphs, part 1. Data structures for graphs. Graph traversal (depth-first and breadth-first). Connected components.
24-11-2017 TP.11	Graphs, part 2. All paths and all cycles. Shortest paths. Minimal spanning tree. Topics in computational geometry, part 1. Point location. Quad-trees and oct-trees.
15-12-2017 TP.12	**Test (algorithmic techniques, TP.09 to TP.11).** Topics in computational geometry, part 2. Convex hull. Voronoi diagrams. Delaunay triangulations. Steiner trees.

List of planned lectures (part 2)

Class	Summary
14-09-2017 P.01	Study and simple modifications of some C programs.
21-09-2017 P.02	Study and simple modifications of some C and C++ programs.
28-09-2017 P.03	Study and simple modifications of some C++ programs.
12-10-2017 P.04	Computational complexity exercises (paper and pencil).
19-10-2017 P.05	To be defined.
26-10-2017 P.06	To be defined.
02-11-2017 P.07	To be defined.
09-11-2017 P.08	To be defined.
16-11-2017 P.09	To be defined.
23-11-2017 P.10	To be defined.
30-11-2017 P.11	To be defined.
07-12-2017 P.12	To be defined.
14-12-2017 P.13	To be defined.
31-12-2017	**Report (P.09 to P.13).**

All programming will be done in either `C` or `C++`.

Recommended bibliography for the entire course

Algorithms, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
Análise da Complexidade de Algoritmos, António Adrego da Rocha, FCA.
Analysis of Algorithms, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
C in a nutshell, a desktop quick reference, Peter Prinz and Tony Crawford, O'Reilly, 2006.
Estruturas de Dados e Algoritmos em C, António Adrego da Rocha, terceira edição, FCA.
Programming Pearls, Jon Bentley, second edition, Addison Wesley, 2000.
Thinking in C++. Volumes One and Two, Bruce Eckel and Chuck Allison, Prentice Hall, 2000 and 2003.

Books that each serious programmer should have (incomplete list)

Algorithm Design, Jon Kleinberg and Éva Tardos, Addison Wesley, 2006.
Algorithms, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
Computational Geometry. Algorithms and Applications, M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, second edition, Springer, 2000.
Concrete Mathematics, Ronald L. Graham, Donald E. Knuth, and Oren Patashnik, second edition, Addison Wesley, 1994.
Handbook of Data Structures and Applications, Dinesh P. Mehta and Sartaj Sahni (editors), Chapman and Hall/CRC, 2005.
Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT Press, 2009.
Numerical Recipes. The Art of Scientific Computing, William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, third edition, Cambridge University Press, 2007.
Object-Oriented Software Construction, Bertrand Meyer, second edition, Prentice-Hall, 1997.
The Algorithm Design Manual, Steven S. Skiena, second edition, Springer, 2008.
The Art of Computer Programming, Volume 1 (Fundamental Algorithms), Donald E. Knuth, third edition, Addison Wesley, 1997.
The Art of Computer Programming, Volume 2 (Seminumerical Algorithms), Donald E. Knuth, third edition, Addison Wesley, 1998.
The Art of Computer Programming, Volume 3 (Sorting and Searching), Donald E. Knuth, third edition, Addison Wesley, 1998.
The Art of Computer Programming, Volume 4A (Combinatorial Algorithms, Part 1), Donald E. Knuth, Addison Wesley, 2011.

The C programming language

Summary:

- C language overview
- Preprocessor directives
- Comments
- Data types
- Declaration, definition, and scope of variables
- Assignments and expressions
- Statements
- Functions
- Standard library functions
- Coding style

[Remark: due to space and time limitations, we will omit many details.]

Fact. According to an IEEE spectrum survey, in 2017 the top five most popular programming languages were:

- `Python` (100.0, 97.9 in 2016)
- `C` (100.0, 100.0 in 2016)
- `Java` (99.4, 98.1 in 2016)
- `C++` (96.9, 95.8 in 2016)
- `C#` (88.6, 86.3 in 2016)

An example:

```
 1 /*
 2 ** Hello world program
 3 */
 4
 5 #include <stdio.h>
 6
 7 int main(void)
 8 {
 9   puts("Hello world!");
10   return 0;
11 }
```

(The numbers on the left are not part of the code.) Lines 1 to 3 are a comment. Line 5 instructs the compiler to replace line 5 by the contents of the file named `stdio.h` (one of the files of the C compiler's standard libraries). Line 7 declares and defines a function named `main`, which is the entry point of the program and, in this case, takes no arguments and returns an integer. Lines 8 to 11 constitute the body of the function. Line 9 is a call to a function named `puts` (declared in `stdio.h`), belonging to the C standard library, that outputs its string argument to the terminal. Line 10 forces a return from the main function with a return value of 0 (since `main` is the entry point of the program, this actually specifies the error code of the entire program; 0 means all is well, non-zero means some error occurred).

C language overview

Each C source code file may contain

- preprocessor directives
  Preprocessor directives tell the compiler to manipulate the source code in certain ways. For example, the `#include` directive tells the compiler to replace the entire include directive line by the text of the file whose name appears after the include directive. There are also directives that allow us to define replacement text for a given word and to do conditional compilation of code.
- comments
  A comment is a chunk of text that is ignored by the compiler
- declaration of new data types
  New data types agglomerate one or more existing data type into a new type, and give it a name that we can use from that point on to refer to that new type.
- declaration of variables and of functions
  A declaration of a variable is a description of the type and memory storage attributes of the variable. A declaration of a function is a description of the arguments and return value (if any) and their type of the function. It **does not** reserve space in memory for a variable and **no code** is produced for a function. After a declaration, the variable or function can be used in our code, even if its definition (see next item) is elsewhere in the code (it can even be missing in our code if it resides in a code library).
- definition of variables and of functions
  When the compiler encounters a definition it reserves space for a variable and generates code for a function. It is possible to declare and define a variable (or a function) at the same time. Variable and function names have to be unique.

Preprocessor directives (part 1)

The C preprocessor can be used to modify on the fly the text that is going to be fed to the C compiler. Each preprocessor directive must be placed on a line that begins with the character #. The most important of them are:

- `#include <filename>`
  `#include "filename"`
  This directive instructs the preprocessor to replace the directive by the entire text of the file whose name follows the include directive. The first form looks for files only in compiler directories (standard library header files). The second form looks for user files (in the current directory).
- `#define NAME substitution_text`
  `#define NAME(arg1,arg2,...) substitution_text`
  This directive defines a preprocessor macro named `NAME`. On subsequent lines, each time the text `NAME` appears in the source code it is replaced by the substitution text. In the first form, the macro does not have any arguments. In the second form it can have one or more arguments. If the name of an argument appears in the substitution text it gets replaced by the text that was placed in the argument when the macro was invoked. For example, the code fragment

  ```
  #define C      (int)
  #define X(i)   x[i]
  #define Y(i,j) i * j
  C X(3) + Y(i,7)
  ```

  gets transformed into the code

  ```
  (int) x[3] + i * 7
  ```

Preprocessor directives (part 2)

- `#undef NAME`
  It is not possible to redefine a macro (with a different replacement text) without first removing its previous definition. This directive makes sure that a previous definition of the macro (if any) is removed.
- `#if EXPRESSION`
  `#elif EXPRESSION`
  `#else`
  `#endif`
  If the integer expression, which must use only constants known to the preprocessor (macros with replacement text that are integers), is non-zero, then the text in the lines following an `#if` directive and up to an `#elif`, an `#else` or an `#endif` directive gets fed to the compiler; `#elif` and `#else` directives are treated in the logical way. Symbols unknown to the preprocessor are replaced by zeros. For example, in the code fragment

```
#if N == 1
line1
#elif N == 2
line2
#else
line3
#endif
```

  `line1` is passed to the compiler only if the macro `N` is defined and evaluates to 1, `line2` is passed to the compiler only if the macro `N` is defined and evaluates to 2, and `line3` is passed to the compiler if the macro `N` is not defined (and so is replaced by 0) or if it does not evaluate to either 1 or 2.

Comments

Comments are annotations placed in the source code of a program. A good comment explains a non-obvious thing, such as how some piece of code works, or what trade-offs (between, say, execution time and memory usage) were made in that part of the code and why. Comments are also usually used to indicate who wrote a part of a program, and to record significant changes in the source code.

In C there are two kinds of comments: single line comments, which begin with // and end at the end of the line, such as in

```
d = (d + 1) | 1; // if d is even increment it by one, otherwise, increment it by 2
```

and comments that can span multiple lines, which begin with /* and end with **the first** */, such as in

```
/*
** in the following loop d takes the values 2, 3, 5, 7, 9, 11, 13, 15, 17, ...,
** up to (and including) the square root of n
**
** we would have liked for d to be the prime numbers 2, 3, 5, 7, 11, 13, 17, ...,
** but that is much more difficult to achieve
**
** FIX ME: there is arithmetic overflow if n is a prime number close to the largest representable
** signed integer; for 32-bit integers this can be fixed by exiting the loop as soon as d > 46340
*/
for(d = 2;d * d <= n;d = (d + 1) | 1)
```

Comments are removed by the preprocessor. The preprocessor joins a line terminated by \ with the next line, so single line comments may actually span more than one line if they are terminated in that way).

A simple and fast way to force the compiler to ignore a large continuous piece of code, even it is has comments, is to put it between #if 0 and #endif lines.

Data types (part 1, integer data types)

The C language has the following fundamental integer data types (in non-decreasing order of size): `char`, `short`, `int`, `long` and `long long`. Each of these types can be either `signed` (the default with the possible exception of the `char` type) or `unsigned`. In the following code fragment we present an example of the simultaneous declaration, definition, and initialization of variables for each one of these types.

```
                char  c0 = 'A';  // by default signed on most compilers
  signed        char  c1 = 'B';  // make sure the type is signed
unsigned        char  c2 = 'C';

                short s0 = 1763;  // the same as signed short
unsigned        short s1 = 1728;

                int   i0 = -1373762;  // the same as signed int
unsigned        int   i1 = 8382382U;  // the trailing U signals that the integer constant is unsigned

                long  l0  = 82781762873L;  // the same as signed long and signed long int
unsigned        long  l1 = 38273827322UL;  //   the int is optional, so we do usually do not put it

           long long  L0  = 8278176284398439847 3LL;  // the same as signed long long int
unsigned   long long  L1 = 3827382733493498332 2ULL;  //   the int is optional
```

Unfortunately, the designers of the C language did not specify the size (number of bytes) of most of these types. So, an `int` may have two bytes if the compiler is producing code for a very old processor, and four bytes if it is targeting a modern processor. The types `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, `uint32_t`, `int64_t`, and `uint64_t`, defined in the header file `stdint.h`, should be used whenever a specific size is desired.

Data types (part 2, floating point and pointer data types)

The C language has two fundamental floating point data types: `float` (single precision, 4 bytes) and `double` (double precision, 8 bytes). In the following code fragment we present an example of the simultaneous declaration, definition, and initialization of variables of each one of these types.

```
 float f =  1.23e3f;  // the same as 1230.0f (f denotes a 4-byte floating point constant)
double d = -1.23e6;   // the same as -1230000.0 (no f, so a 8-byte floating point constant)
```

The C language has only one more fundamental data type: a pointer. A pointer is an integer that represents the memory address where a variable of a given type is stored. One can, using the appropriate syntax (see below) read or write the contents of the memory location whose address is stored in the pointer. (Note that, to the processor, the pointer itself is an integer variable.) Given that it is possible to do arbitrary modifications to a pointer, it is possible in C to manipulate the contents of arbitrary memory locations (dangerous, but powerful!). Pointers are declared and used by putting a * before the pointer name. To get the address of a variable put an & before the variable name. For example, in the code fragment

```
float f = 1.0f,*pf = &f;
*pf = 2.0;
```

the single precision floating point variable `f` gets the value $\mathbf{1.0}$ at the end of the first line and the value of $\mathbf{2.0}$ at the end of the second. As long as `pf` is not modified, it is possible to change the contents of `f` through the pointer. Incrementing (decrementing) a pointer makes it point to the next (previous) adjacent variable in memory of the same type. If that memory location does not hold a variable of that type, changing the memory through the pointer leads to all kinds of problems. Assigning `NULL` to a pointer is the standard way in C to say that the pointer points to nothing.

We also have pointers to functions: they store the starting address of a function.

Data types (part 3, literal values)

Literal values are the numeric constants we put in our programs. Integer literals can be encoded in several ways:

- as a character, as in `'3'`, `'\''`, `'"'`,`'\n'`, or `'\t'`
- as a decimal integer, as in `123` or `-17`
- as an hexadecimal integer (base 16), as in `0x01F3`
- as an octal integer (base 8), as in `0173`. Constants beginning with with a 0 **are specified in octal**!
- as an binary integer (base 2), as in `0b10101` (gcc compiler).

With the exception of character encodings, append `U` at the end to mark an unsigned integer, append `L` to mark a `long` integer, and append `LL` to mark a `long long` integer. All of these can also be in lower case.

Floating points constants can have an optional sign (plus or minus), can have zero or more digits before an optional decimal point and zero or more digits after the decimal point (in all, at least one digit must be present). It can also have an optional exponent part, composed by the letter e (upper or lower case), followed by a decimal signed integer. For example, `-1.2`, `.2e-13`, `+12.`, `1e-8` are all valid floating point literals. By default, floating point literals are in double precision; append `f` to get a single precision literal, as in `1.23e4f`.

String literals, enclosed by double quotation marks, as in `"12\"9348"`, are of type `const char *`, i.e., they cannot be overwritten. Note, however, that in the first line of the following code

```
char str[] = "292348"; // a char array with 7 elements (why?); str[0] = 'T' is ok
char *pstr = "2983";   // a string literal; *pstr = '4' gives a runtime error
```

the string is used to initialize the array, and so it is not a string literal.

Data types (part 4, arrays)

It is possible to extend the fundamental data types in two ways: using arrays and creating new data types.

An array is just a contiguous group of variables of the same type. For example, the following code

```
double d[10],D[10][10];
```

declares an (unidimensional) array `d` of 10 double precision floating point numbers and declares a bi-dimensional array (a matrix) `D` of 10 by 10 (i.e., 100) double precision floating point numbers. Accessing the array elements is done using square brackets. Indices start at 0. Usually, **no run-time tests** are performed to verify if the index being used to access an array element has a valid value. Using an out-of-range value does not result in any compiler error but will usually lead to a hard to discover run-time error.

It is important to realize that C does not allow you to manipulate an entire array as a single entity. This is so because an array name is a pointer to its first element. For example, in the code

```
double d[10],*pd = d;
```

This implies that `i[d]` is the same as `d[i]`, but no one sane writes an array access in that twisted way.

it is possible to perform accesses to the array using either `d[i]`, which is the same as `*(d+i)`, or `pd[i]`, which is the same as `*(pd+i)`. Both are equivalent, as long as `pd` is not modified. On the other hand, in the code

```
double d[10],*pd = &d[9];
```

an access to `pd[i]` is equivalent to an access to `d[i+9]`, i.e., `pd[-9]` is the same as `d[0]` (assuming that `pd` has not been modified).

In C a string is an array of characters, terminated by a 0. For example, the code

```
char str[20] = "AB"; // same as char str[20] = { 'A','B',0 }; or char str[20] = { 65,66,0 };
```

defines a string that can hold up to 19 characters (space **must** be reserved for the 0 terminator), initialized with the two character string `AB`.

Data types (part 5, pointer arithmetic)

A pointer is the address of a memory location. In C, adding an integer to a pointer **does not**, in general, change the address by that amount: it changes the address by that amount **times** the size in bytes of the type that the pointer points to. Things are done in this way to make working with pointers easier to the programmer, in particular when dealing with arrays. For example, in the code

```
int a[100];
int *pa = &a[30]; // same as int *pa = a + 30;
int *pA = &a[-2]; // same as int *pA = a - 2;
```

the pointer `pa` points to the element with index 30 of the array `a`. The address of this element is the sum of the address of the beginning of the array (`a`, or, what is the same, `&a[0]`) with the number of bytes required to store 30 integers (120 bytes if each integer occupies 4 bytes). In C we only need to add 30 to the pointer to get the correct address; the multiplication by the size of the type pointed to is done automatically by the compiler.

There are two exceptions to the above rule of pointer arithmetic:

- Adding an integer to a pointer to a function is meaningless, because the size of a function (the number of bytes of its code) is not constant and is not known at compile time. So, if one attempts to do this the compiler generates an error.
- Adding a constant to a pointer to `void` (the `void` type will be formally introduced soon), adds that many bytes to the pointer (so `sizeof(void)` is 1 and not 0, as would be more natural). This is done in order to make life easier to the programmer, since manipulating pointers to `void` is sometimes useful.

[**Homework:** study carefully how pointer arithmetic in done in C.]

Data types (part 6, structures and unions)

A structure is a contiguous group of variables of the same or different types, each with its own name. It is declared using the keyword `struct`. In the following code,

```
struct dot
{
  double x;
  double y;
  int color;
  struct dot *next;
};
```

a new data type, named `struct dot`, is declared. It has 4 fields named `x`, `y`, `color`, and `next`, respectively of types `double`, `double`, `int`, and pointer to a dot structure. (One can also put arrays, and even other structures, inside a structure.) The following code fragment gives an example of how structure fields are accessed:

```
struct dot d;   // a dot structure (RESERVES SPACE FOR THE STRUCTURE)
struct dot *pd; // a pointer to a dot structure (DOES NOT RESERVE SPACE FOR THE STRUCTURE)
pd = &d;        // set the pointer to point to the structure (now it can be safely used)
d.x = 3;        // set the x field to 3
pd->color = 5;  // set the color field to 5
d.next = NULL;  // set the next field to NULL (a special address that points to nothing)
```

Unlike arrays, the name of the structure represents the entire structure (it is **not** a pointer to its position in memory; to get that use an &). The abbreviation `struct struct_name;` tells the compiler that a structure named `struct_name` will be fully specified later on.

Unions are like structures, except that all its fields are superimposed in memory. (Believe it or not that is sometimes useful.) Only one field can be in use at any given time.

Data types (part 7, enum and void)

It is possible to create a data type which has a discrete set of constant integer values, each one with its own name. Such a type is an enumerated type. For example, the code

```
enum color { black,red,green,blue = 4 };
```

declares an enumerated type called `enum color`, that can have 4 values: `black`, `red`, `green`, and `blue`, respectively with numerical values 0, 1, 2, and 4. (These numerical values are how the names are internally represented in the program; the program code should use the names.) The following code defines a variable of this type and initializes it with the value `red`:

```
enum color dot_color = red;
```

An enumerated data type is an integer data type. The compiler is free to choose the number of bytes needed to internally represent its values.

There is one final data type that can be used in programs: `void`. This data type cannot have a value. It can be used to tell the compiler that a function does not return anything, or that a function does not have argument, as exemplified in the following code:

```
void f(int x); // returns nothing, has an integer argument
int g(void);   // returns an integer, does not have arguments
void h(void);  // returns nothing, does not have arguments
```

A pointer to `void` is a popular way to declare a pointer to something which has a type that is not explicitly given. Of course it is not possible to dereference (access the memory) a pointer to void; one has to first cast it (see next page) to a pointer to a specific type.

Data types (part 8, casts)

It is possible to tell the compiler to explicitly convert one data type to another compatible type. Such conversions, called casts, are implicitly done by the compiler in arithmetic expressions. For example, if `i` if an `int` and `d` is a `double`, the result of the expression `i + d`, which is the sum of the integer variable `i` with the double precision floating point variable `d` is a double precision floating point number. Before doing the addition, `i` is converted (cast) to the double precision floating point number, and only then is the addition performed. To make the conversion explicit, use `(double)i + d`. [My personal opinion is that mixing different data types in an expression without explicit type conversions is a bad programming practice.]

In C, to explicitly convert `x` to the data type `T` one writes `(T)x`. In C it is only possible to convert from one numeric type to another (that the compiler knows about), because the language does not offer any mechanism to tell the compiler how to perform more complex conversions. For example, it is not possible to convert from an integer to a structure. You will need to write your own function, and call it explicitly, in order to do that.

It is possible to cast a pointer to a data type to a pointer to another data type. That is a very dangerous thing to do (you **must** know what you are doing!) unless one of the two is a pointer to `void`. Indeed, a pointer to `void` is the standard way to go when one desires to perform some action on a memory region without needing to known what it contains (for example, reading or writing it to a file):

```c
void write_data(void *ptr,int size);     // function to write size bytes starting at address ptr
double array[100];
write_data((void *)array,sizeof(array)); // write the 100 doubles
```

Since a pointer is represented by an integer it is also possible to convert a pointer to a sufficiently wide integer, and vice versa, but that is not recommended (and not needed in any sane program).

Data types (part 9, typedef)

In C it is also possible to give another name to an existing data type using the `typedef` keyword. For example, the following code fragment declares a data type named `u64` that is supposed to be a 64-bit unsigned integer:

```
#ifdef IS_A_32_BIT_CPU
typedef unsigned long long u64; // A 64-bit data type on a 32-bit CPU
#endif
#ifdef IS_A_64_BIT_CPU
typedef unsigned long u64;      // A 64-bit data type on a 64-bit CPU
#endif
```

The rest of our code can now use the type `u64`. Switching from a 32-bit to a 64-bit CPU requires only **two** very small changes in the code (and a recompilation), namely, undefining the symbol `IS_A_32_BIT_CPU` and defining the symbol `IS_A_64_BIT_CPU`. (This can be done without modifying the code by defining the appropriate symbol on the command line that invokes the compiler.)

A `typedef` can also be used in conjunction with the declaration of a `struct`:

```
typedef struct dot // "struct dot" is now known to exist
{
  double x,y; int color;
  struct dot *next; // the type dot is not yet known, so we have to use struct dot, which is known
}
dot; // the name of the new type is dot; it is the same as struct dot
```

Data types (part 10, sizeof)

The number of bytes used by any variable of type `T` is given by `sizeof(T)`. The parenthesis are optional (to avoid confusion due to the precedence of operators, please always use parenthesis). For example, in the code

```
int i = sizeof(dot); // same as i = sizeof dot;
```

the variable `i` will be initialized with the number of bytes required to store a `dot` in memory.

The number of bytes used by a specific variable named, say, `var`, is given by `sizeof(var)`; again, the parenthesis are optional. The `sizeof` operator does the right thing when the variable is an array (it returns the number of bytes needed to store the entire array), despite the fact that in other places the array name is actually a pointer to its first element.

The argument of the `sizeof` operator may also be an expression. For example, `sizeof(1 + 2)` is the number of bytes needed to store the result of the expression @1 + 2@. Here the parenthesis **must** be used; `sizeof(1 + 2)` is not the same as `sizeof 1 + 2` (why?).

The `sizeof` operator returns an integer with type `size_t`, which is usually a 32-bit data type in 32-bit processors and a 64-bit data type in 64-bit processors (it is usually the number of bytes needed to store a pointer variable). Thus, to avoid an implicit cast, the first example above should have been written as follows:

```
int i = (int)sizeof(dot);
```

[**Homework:** since `a[i]` is converted by the compiler into `*(a + i)` before generating code, what should the type of `i` be on i) 32-bit processors, ii) 64-bit processors?]

Declaration, definition, and scope of variables (part 1)

Variables can be defined either outside or inside a function body. Variables defined outside a function body, called global variables, can be used by more than one function. Variables defined inside a function body, called local variables, can only be used directly by that function; they can, however, be made accessible to other functions **called** by that function via pointers (it is a serious mistake to return the address of a local variable).

In the case of variables declared or defined **outside** a function body there are several cases to consider:

- Declaration of a variable that may, or may not, be defined in the same source code file, as in

  ```
  extern int global_var;
  ```

  The `extern` keyword tells the compiler that the variable may be defined elsewhere.
- Declaration of a variable that is defined in the same source code file, as in

  ```
  int global_var;
  ```

  Without initialization, this is actually a declaration (it will be transformed into a definition if a definition is not found elsewhere in the source code; in that case the memory location where the variable resides is initialized with all zeros).
- Definition of a variable that is defined in the same source code file, as in

  ```
  int global_var = 0;
  ```
- Declaration and definition of a global variable that is visible only to the functions of the same source code file, as in

  ```
  static int global_var;
  ```

  The same line of code in a different source code file gives rise to a **different** variable (it is considered very bad practice to use variables with the same name in this way).

Declarations remains in effect until the end of the source code file.

Declaration, definition, and scope of variables (part 2)

A code block (a compound statement) begins with `{` and ends with `}`. A function body has one outermost code block. Inside this outermost code block there may exist more, possibly nested, code blocks.

In the case of variables declared or defined **inside** a code block there are also several cases to consider:

- Declaration of a variable that may, or may not, be defined in the same source code file, as in

  ```
  extern int global_var;
  ```

- Definition of a variable that lives only inside the code block, as in

  ```
  int local_var_1;       // uninitialized
  int local_var_2 = 123; // initialized
  ```

  Without initialization, the variable initially has an unspecified value.
  The variable is created whenever the program enters the code block and is destroyed when it leaves the code block.

- Definition of a persistent variable, visible only on the code block, that retains its value even outside the code block, as in

  ```
  static int local_var_1;       // initialized
                                //   with zeros
  static int local_var_2 = 123; // initialized
  ```

  Each invocation of the function uses the **same** variable.

In all cases, the declaration/definition is forgotten at the end of the code block. It is considered bad practice to give the same name to a local and a global variable.

The C99 language standard allows declaration of variables anywhere inside a code block. In previous versions of the C language standard, variables could only be declared at the beginning of a code block.

Declaration, definition, and scope of variables (part 3)

Each data type may be modified by the use of so-called qualifiers. A qualifier may be used by the programmer in a declaration or definition to tell the compiler what operations or optimizations it is allowed to do when copying that variable to or from memory. Of the three qualifiers that the C language currently knows of, `const`, `volatile`, and `restrict`, only the `const` qualifier will be described here. [**Homework:** find you what the other two are for.] An object qualified as `const` is constant; the program cannot modify it. In the following code

```
const double pi = 3.14159265358979323846;
```

the value of `pi` can be used in an expression as if it were a `double` variable, but it cannot be changed.

For a pointer type, a qualifier to the right of the asterisk qualifies the pointer itself; a qualifier to the left of the asterisk qualifies the type of object it points to. For example, in the following code some things are allowed and some are not (read the comments).

```
int i;                      // an integer
const int c = 3;            // an integer constant
const int *p;               // a (variable) pointer to a constant integer
                            // p = &i; is allowed, but i cannot be changed via the pointer
                            // *p = 1; is not allowed
                            // p = &c; is allowed
int * const q = &i;         // constant pointer (it must always point to i)
                            // *q = 1; is allowed
int * const r = &c;         // not allowed
const int * const z = &c; // allowed
```

The `const` qualifier is particularly useful to qualify function arguments that are pointers (many functions receive a pointer to a memory region that will not be modified by the function, in which case a `const` will help the compiler produce better code).

Assignments and expressions (part 1, assignments)

Assignments take the form `LHS = RHS;`, where `LHS` is the so-called lvalue (left value, or, more accurately, location value) and where `RHS` is an expression. The lvalue represents the place where the value of the `RHS` expression will be stored. The following code presents examples of legal and illegal `LHS` lvalues:

```
int a;              a = 3;     // legal
                    a + 1 = 7; // illegal (what is the location of a+1?)
int *pa;            pa = &a;   // legal (the pointer itself has a location)
                    *pa = 7;   // legal
int A[10];          A[3] = a;  // legal
const int c = 3; c = 4;     // illegal (c has a location, but it is not writable)
```

When a pointer is used on the `LHS` to specify a write address, the `LHS` may also modify the pointer if the ++ or -- operators are used. For example, the following code

```
*++pa = 3; // same as *(++pa) = 3; increment the pointer, and use its new value as the write address
*pa-- = 3; // same as *(pa--) = 3; use the pointer as the write address, and then decrement the pointer
```

is legal, while the code

```
pa++ = &a; // nonsense (are we really trying to store &a in pa and then attempting to increment pa?)
```

is not.

An assignment is in itself an expression, with a value equal to that of the `RHS` (after conversion to the type of the `LHS`). So, it is possible to put several assignments in a chain, as in the following code:

```
int i,j,k;
i = j = k = 3; // same as i = (j = (k = 3));
```

Assignments and expressions (part 2, expressions)

As expression is a sequence of constants, variables, and function calls intertwined with operators that combine them. An expression may perform a mathematical calculation, in which case either we will be interested in recording its value by saving it in a variable (as assignment), or we may be interested to test its value with the purpose of deciding what the program should do next (a conditional jump). An expression may also be useful because of its side effects, as when a function that returns nothing (`void`) is called to do something. The type of an expression is the type of the value that is the result of the expression; it may be `void` if the expression has no value (as in a call to a function that does not return anything). The following are examples of expressions (`i` is an `int`, `d` is a `double` and `exit` is a function than has one `int` argument and that does not return anything):

```
i                   // int
i + '0'             // int
(i << 3) ^ (i & 7)  // int   (the parentheses force i<<3 and i&7 to be evaluated first)
(double)i * d       // double, same as i * d
exit(1)             // void
i && (d == 3.0)     // int
"abc"               // const char *
"abc"[i]            // char
i = 3               // int
d = i = 5           // double
i = d = 2.5         // int
```

The binary arithmetic operators perform an automatic type conversion whenever their two arguments are not of the same type: the argument having the type with smaller range of values is converted to the other. For example, a `char` is converted to an `int` (`char + int` becomes `int + int`), and an `int` is converted to a `double` (`int * double` becomes `double * double`).

Assignments and expressions (part 3a, operators)

C has the following operators, in decreasing order of priority:

1. **postfix operators, left to right associativity**

`[]`	array access
`()`	function call
`.`	structure field
`->`	structure field, from a pointer
`++`	post increment; use value, then increment
`--`	post decrement; use value, then decrement

2. **unary operators, right to left associativity**

`++`	pre increment; increment, then use value
`--`	pre decrement; decrement, then use value
`!`	logic negation (0 gives 1, non-zero gives 0)
`~`	bitwise negation
`+`	does nothing (`+1` is just `1`)
`-`	arithmetic negation
`*`	pointer dereference
`&`	address of

3. **cast operator, right to left associativity**

`(T)`	type conversion; T is a data type

4. **multiplicative operators, left to right associativity**

`*`	multiplication
`/`	division
`%`	remainder

5. **additive operators, left to right associativity**

`+`	addition
`-`	subtraction

6. **shift operators, left to right associativity**

`<<`	shift left
`>>`	shift right

Note: since these are usually used to perform multiplications and divisions by powers of 2 they should have been given a higher priority than addition and subtraction!

7. **relational operators, left to right associativity**

`<`	less than
`<=`	less than or equal to
`>`	larger than
`>=`	larger than or equal to

Note: 1 when true, 0 when false

8. **equality operators, left to right associativity**

`==`	equal to
`!=`	different from

Note: 1 when true, 0 when false

9. **bitwise and, left to right associativity**

`&`	bitwise and

Assignments and expressions (part 3b, operators)

10. **bitwise exclusive or, left to right associativity**

`^`	bitwise exclusive or

11. **bitwise or, left to right associativity**

`|`	bitwise or

12. **logical and, left to right associativity**

`&&`	logical and

    Note: if the argument on the left is zero, the result is 0 and the argument on the right **is not** evaluated; otherwise the result is 0 if the argument on the right is zero and is 1 if not.

13. **logical or, left to right associativity**

`||`	logical or

    Note: if the argument on the left is nonzero, the result is 1 and the argument on the right **is not** evaluated; otherwise the result is 1 if the argument on the right is nonzero and is 0 if not.

14. **conditional operator, right to left associativity**

`?:`	`a ? b : c`	evaluates to `b` if `a` is nonzero and evaluates to `c` if `a` is zero

15. **assignment operators, right to left associativity**

`=`	simple assignment
`+=`	compound assignment (add)
`-=`	compound assignment (subtract)
`*=`	compound assignment (multiply)
`/=`	compound assignment (divide)
`%=`	compound assignment (remainder)
`&=`	compound assignment (bitwise and)
`^=`	compound assignment (bitwise exclusive or)
`|=`	compound assignment (bitwise or)
`<<=`	compound assignment (left shift)
`>>=`	compound assignment (right shift)

    Note: the compound assignment `a op= b` where `op` is one of the operators above is equivalent to `a = a op (b)`.

16. **comma operator, right to left associativity**

`,`	discard an expression; start a new one

When in doubt about the priority of an operator, use parentheses! Right to left associativity means that things on the right have precedence over things on the left. For example, `a = b = 3;` means `a = (b = 3);`. Left to right associativity is just the opposite. Expressions with side-effects that affect the same variable, like `j = i++ + ++i;`, are ill-defined because different compilers may choose different orders of evaluation.

Statements (part 1, expression, compound, go to, and return statements)

Statements come in many guises:

- **expression statements**

As expression statement is an expression, possibly empty, followed by a semicolon. The following are valid expression statements:

```
;
i = 2;
i = 3, j = 4;
exit(1);
```

- **compound statements (block statements)**

A compound statement (what we called before a code block) starts with a {, is followed by zero of more declarations or definitions of variables and zero of more declarations of functions, is then followed by zero or more statements, and is terminated by a }. The following code is a valid compound statement:

```
{
  int i = 3 + x;       // x declared elsewhere
  {
    int j = i * i + y; // y declared elsewhere
    k += i + j;        // k declared elsewhere
  }
}
```

- **go to statements**

The go to statement causes an unconditional jump to another statement in the same function. It should be used with **care** and **parsimony**, if at all. The destination of the jump is specified by the name of a label, as in

```
goto x_marks_the_spot;
```

The label itself is a name followed by a colon, as in

```
x_marks_the_spot:
```

The break and continue statements, to be presented below, are disciplined (and disguised) go to statements.

- **return statements**

Return statements are used to end the execution of the current function. It has the form

```
return expression;
```

The expression must be missing if the function does not return anything (declared as returning a `void`). The value of the expression is returned to the caller of the function.

Statements (part 2, if statements)

- **if statements**

An if statement has two possible forms: either

```
if(expression)
  statement_t  // to be executed if the
               //   expression is non-zero
```

or

```
if(expression)
  statement_t  // to be executed if the
               //   expression is non-zero
else
  statement_f  // to be executed if the
               //   expression is zero
```

Care must be taken if several if statements are nested and the else part is present in some of them. For example, in the following code

```
if(i >= 0)
  if(i > 0)
    j = 1;
  else
    j = 0;
else
  j = -1;
```

the first else belongs to the second if and the second else belongs to the first if, as suggested by the indentation of the code. (**Advice:** always indent correctly your code.) This is so because the `statement_t` of the first if is `if(i > 0) j = 1; else j = 0;`. If in doubt use curly braces to transform a statement into a compound statement:

```
if(i >= 0)
{
  if(i > 0)
    j = 1;
  else
    j = 0;
}
else
  j = -1;
```

- **loop statements**

There are three kinds of loop statements: for, while, and do-while statements. There is one way to quickly get out of a loop: break statement. There is one way to quickly jump to the next loop iteration: continue statement.

Statements (part 3, for, while, do-while, break, and continue statements)

- **for statements**

A for statement has the following form:

```
  for(expression1;expression2;expression3)
    body_statement
```

It is equivalent to

```
  {
      expression1;
loop: if((expression2) == 0) goto end;
      body_statement
next: expression3;
      goto loop;
end:  ;
  }
```

- **while statements**

A while statement has the following form:

```
  while(expression)
    body_statement
```

It is equivalent to

```
  {
loop: if((expression) == 0) goto end;
      body_statement
next: goto loop;
end:  ;
  }
```

- **do-while statements**

A do-while statement has the following form:

```
  do
    body_statement
  while(expression);
```

It is equivalent to

```
  {
loop: body_statement
next: if((expression) != 0) goto loop;
end:  ;
  }
```

Each loop statement will get its own private label names. A break statement inside the body of a loop statement amounts to a `goto end;`, i.e., it forces an exit of the loop statement. A continue statement inside the body of a loop statement amounts to a `goto next;`, i.e., the remaining statements of the body of the loop are skipped.

Statements (part 4, switch statements)

- **labeled statements**

All statements can be labeled, i.e., they can start with a label. There are three forms of a labeled statement:

```
label_name:            statement
case const_expression: statement
default:               statement
```

The first form is used by go to statements. The other two are used by switch statements.

- **switch statements**

A switch statement has the form:

```
  switch(int_expression)
    body_statement
```

It is usual, but not necessary, for `body_statement` to be a compound statement. The switch statement works as follows. First, `int_expression` is evaluated. If its value matches the `const_expression` value of one of the case statements, the program jumps to that statement. If none of the cases match, and if there is a default label, the program jumps to the corresponding default statement. Otherwise, the program skips the entire switch statement. A break statement transfers execution to the end of the switch statement. The following code presents an example of a switch statement:

```
  switch(c)
  {
    case 't':
      k = 1;
      do_t();
      break;  // terminate the switch
    default:  // the default can be anywhere
      k = 2;
      break;  // terminate the switch
    case 'x':
      do_x(); // no break; do next statements
    case 'X':
      k = 3;
      break;  // terminate the switch
  }
```

The following code is valid but a bit weird (the switch can be replaced by an if statement):

```
  for(i = j = 0;i < 10;i++)
    switch(i % 3)
      case 1: j += i;
```

Functions (part 1, prototypes)

A function definition is composed of two parts, a function header, which specifies the function name, its return type, and its arguments and their types, and a function body, which must take the form of a compound statement. A function declaration (a so-called function prototype) is just the function header, followed by a semicolon, and possibly preceded by the keyword `extern`.

It is not possible to define a function inside another function. Just like variables, it is possible to declare functions at the beginning of a compound statement.

The modern form of the header of a function (there exists an older form, but nowadays no one uses it) has the following form

```
qualifier type function_name( parameter_declarations )
```

The `qualifier` may be absent. The `parameter_declarations` may either be `void`, if the function does not take any arguments, or be composed by one or more individually declared arguments (type and name), separated by comas. The following are examples of function prototypes (headers terminated by a semicolon):

```
int main(void);
int main(int argc,char **argv); // same as int main(int argc,char *argv[]);
extern double sqrt(double x);
static int F(int n);
inline static double sqrt_2(void);
```

(Actually, it is possible to omit the argument names in function prototypes, but we prefer to include them, as their name may shed some light about what they stand for.)

Functions (part 2, parameters)

The parameters of a function behave (in the function body) as if they were ordinary variables, i.e., as if they had been declared and initialized at the very beginning of the compound statement that constitutes the function body. They are passed to the function **by value**, i.e., a copy of each argument is made when the function is called. Thus, changes to the function arguments inside the function, which is allowed, are made on **the copy**. For example, the call `swap(x,y)` to the function

```
void swap(int x,int y)
{
  int tmp;

  tmp = x;
  x = y;
  y = tmp;
}
```

will exchange `x` with `y` only inside the `swap` function. To actually reflect the exchange outside of this function, in C one has to use pointers and pass to the (modified) function the addresses of what we want to exchange. In this case, the function call becomes `swap(&x,&y)` and the function becomes

```
void swap(int *x,int *y)
{
  int tmp;

  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Note that since the name of an array is a pointer to its first element, in the case of arrays what is passed to the function is a pointer. So, arrays are automatically passed **by reference**. No copy of the entire array is made. On the other hand, structures and unions are passed by value, so if you put an array inside a structure you can actually pass (in disguise) an array by value to a function.

Functions (part 3, qualifiers)

It is possible to declare that a function is `static`. This means that its name is only known to the current file being compiled. A function with the same name can be defined in another source code file. It is very bad practice to have static functions (or one non-static and the others static) with the same name in different source code files. If the source code of a program is distributed among several source code files, using a static function is a great method to hide it from the rest of the source code (for example, to make sure that it is not used inappropriately).

It is also possible to declare that a function is `inline`. This means that the compiler will try to replace all calls to the function by copies of its code. The program will be somewhat larger, as the function code may appears in several places, but it will also be faster, as there will be no function call overhead. If the function body is too large, the compiler may silently refuse to inline a function.

The function called `main` is the entry point of the program. It can be defined to either have no arguments, as in

```c
int main(void);
```

or it can be defined to have two arguments, as in

```c
int main(int argc,char **argv); // same as int main(int argc,char *argv[]);
```

In the second case, the first argument is the number of command line arguments with which the program was invoked and the second is an array of pointers to strings with the text of the arguments. For example:

```c
#include <stdio.h>
int main(int argc,char **argv)
{                               // this program prints its arguments, one per line
  for(int i = 0;i < argc;i++)  // definition of a variable anywhere (a la C++) is a c99 feature
    printf("%s\n",argv[i]);    // compile this using "cc -Wall -O2 -std=c99 main.c"
  return 0;
}
```

Functions (part 4, variable number of arguments)

It is possible for a function to have a variable (i.e., optional) number of arguments. It is the responsibility of the programmer to determine how many arguments were actually provided in every function call. There exists a standard mechanism (c.f. `stdarg.h`) to fetch the value of the next variable argument given its type. The programmer has to know the type of the argument. For example, this can be provided by a previous argument (as done, for example, in the `printf` function; see next slide).

A function has a variable number of arguments if its last argument (the optional part) is specified as `...`. It must have at least one standard argument. The following code illustrates how a variable number of arguments is specified and used (in this example all extra arguments all integers, with a special value to signal the end):

```c
#include <stdio.h>
#include <stdarg.h>

int sum(int terminator, ...)
{
  va_list a;                           // standard way to access the extra arguments
  int sum,n;

  va_start(a,terminator);              // the extra arguments start after the terminator argument
  for(sum = 0;;sum += n)
    if((n = va_arg(a,int)) == terminator)  // get the next int from the argument list
      break;
  va_end(a);
  return sum;
}

int main(void)
{
  printf("%d\n",sum(-1,3,4,7,3,-1));      // should print 17 (3+4+7+3)
  return 0;
}
```

Standard library functions

The C language comes equipped with a relatively large set of predefined functions, declared in so-called header files, and stored in library archives. Among them are functions to read and write data, declared in `stdio.h`, such as

```c
int printf(const char *format, ...);                                    // write formated data
int scanf(const char *format, ...);                                     // read formated data
FILE *fopen(const char *path, const char *mode);                        // open a file
int fclose(FILE *fp);                                                   // close a file
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);       // raw read
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream); // raw write
int fprintf(FILE *stream, const char *format, ...);                     // formated write
```

functions to allocate and free memory, and to terminate a program, declared in `stdlib.h`, such as

```c
void *malloc(size_t size);                                              // allocate memory
void free(void *ptr);                                                   // free memory
void *calloc(size_t nmemb, size_t size);                                // zero-allocate memory
void *realloc(void *ptr, size_t size);                                  // resize an allocation
void exit(int status);                                                  // terminate
```

and functions to compute transcendental mathematical function, declared in `math.h`, such as

```c
double sqrt(double x);
double sin(double x);
double cos(double x);
```

Use the help system of your computer (`man` command on GNU/Linux), to get a full description of what a given function does. This web page, which has links to documents describing in full the GNU implementation of the C standard library functions, is also quite useful.

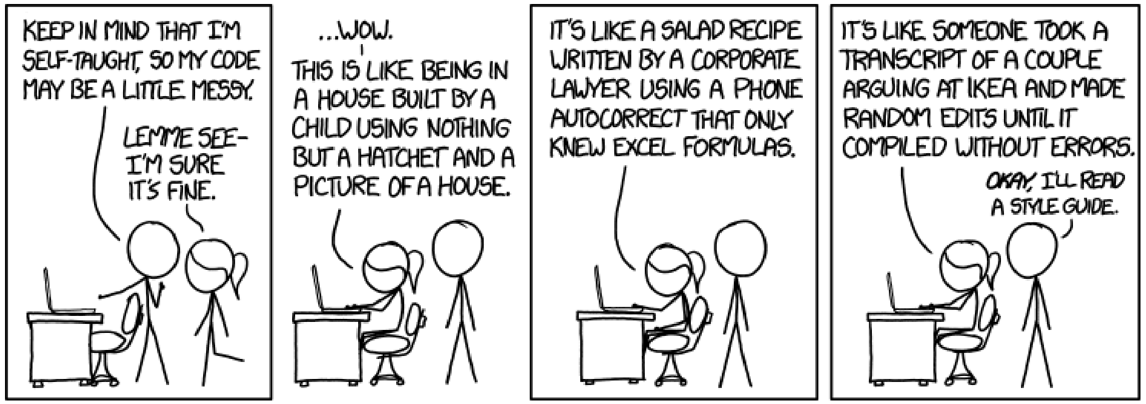
Coding style

Some advice:

- use a consistent coding style; in particular, always indent properly your code,
- do not put too much stuff in a single function,
- use reasonable function and variable names (are you a camelCase fan or a snake_case fan?),
- don't comment obvious code,
- explain, with a comment, each clever trick used in the program (see, for example, the explanation of what the expression `(d+1)|1` does in the `factor.c` program),
- try very hard not to write code what would be admired and envied in the international obfuscated C code contest, and
- try very hard not to be like the guy in the three xkcd cartoons on the next page.

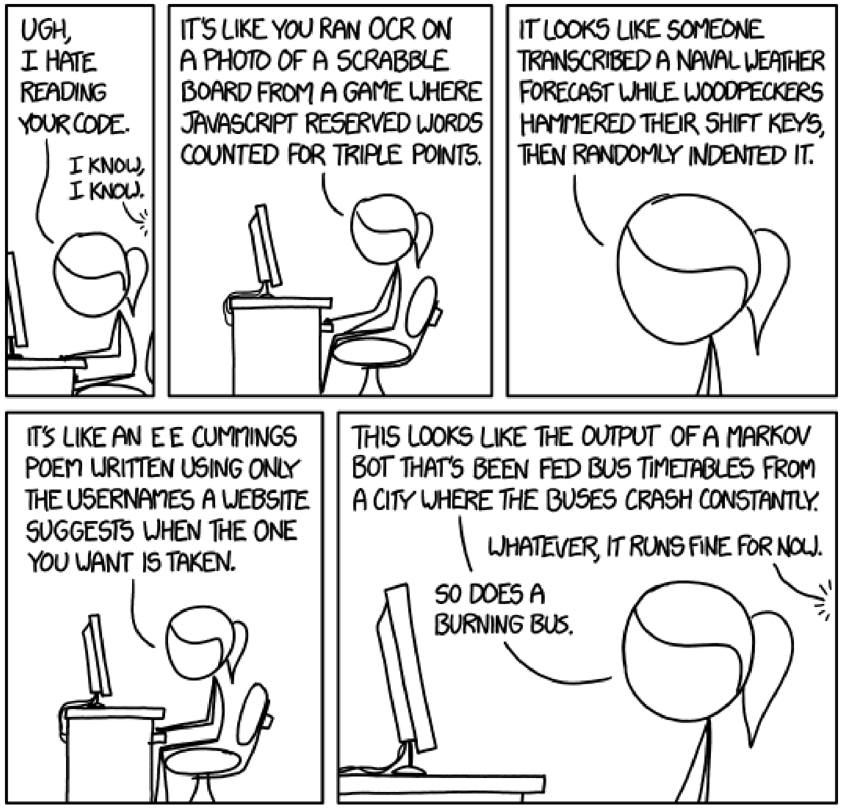
Coding style (xkcd cartoons)

Code Quality (spoiler)


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE– I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.

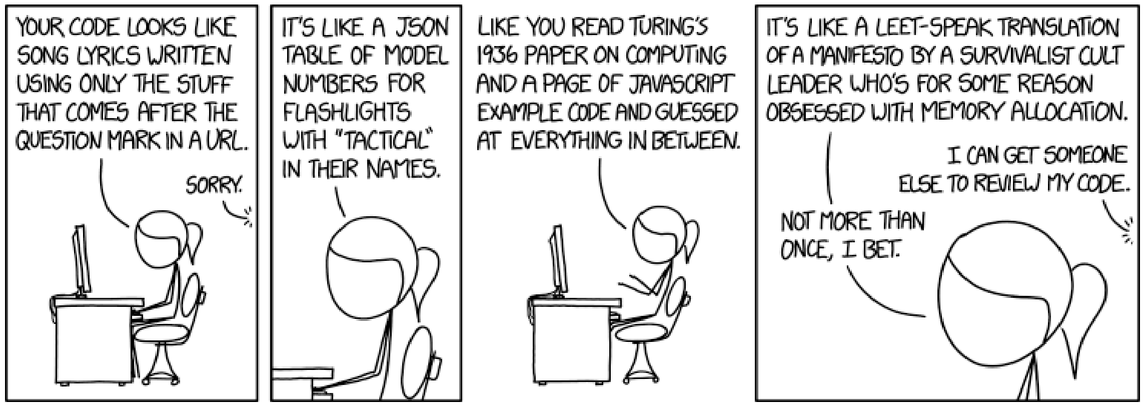
It's like you tried to define a formal grammar based on fragments of a raw database dump from the QuickBooks file of a company that's about to collapse in an accounting scandal.

Code Quality 2 (spoiler)


UGH, I HATE READING YOUR CODE.
I KNOW, I KNOW.
IT'S LIKE YOU RAN OCR ON A PHOTO OF A SCRABBLE BOARD FROM A GAME WHERE JAVASCRIPT RESERVED WORDS COUNTED FOR TRIPLE POINTS.
IT LOOKS LIKE SOMEONE TRANSCRIBED A NAVAL WEATHER FORECAST WHILE WOODPECKERS HAMMERED THEIR SHIFT KEYS, THEN RANDOMLY INDENTED IT.
IT'S LIKE AN E E CUMMINGS POEM WRITTEN USING ONLY THE USERNAMES A WEBSITE SUGGESTS WHEN THE ONE YOU WANT IS TAKEN.
THIS LOOKS LIKE THE OUTPUT OF A MARKOV BOT THAT'S BEEN FED BUS TIMETABLES FROM A CITY WHERE THE BUSES CRASH CONSTANTLY.
WHATEVER, IT RUNS FINE FOR NOW.
SO DOES A BURNING BUS.

I honestly didn't think you could even USE emoji in variable names. Or that there were so many different crying ones.

Code Quality 3 (spoiler)


YOUR CODE LOOKS LIKE SONG LYRICS WRITTEN USING ONLY THE STUFF THAT COMES AFTER THE QUESTION MARK IN A URL.
SORRY.
IT'S LIKE A JSON TABLE OF MODEL NUMBERS FOR FLASHLIGHTS WITH "TACTICAL" IN THEIR NAMES.
LIKE YOU READ TURING'S 1936 PAPER ON COMPUTING AND A PAGE OF JAVASCRIPT EXAMPLE CODE AND GUESSED AT EVERYTHING IN BETWEEN.
IT'S LIKE A LEET-SPEAK TRANSLATION OF A MANIFESTO BY A SURVIVALIST CULT LEADER WHO'S FOR SOME REASON OBSESSED WITH MEMORY ALLOCATION.
I CAN GET SOMEONE ELSE TO REVIEW MY CODE.
NOT MORE THAN ONCE, I BET.

It's like a half-solved cryptogram where the solution is a piece of FORTH code written by someone who doesn't know FORTH.

The C programming language (exercises)

— P.01 —

To compile C and C++ programs, Windows 10 users should install the Windows Subsystem for Linux. As an alternative, they can install instead an Integrated Development Environment (IDE), such as Visual Studio or Eclipse. GNU/Linux users need to install the C++ compiler.

Summary:

- How to compile and run a program (GNU/Linux)
- How to manage archives
- The “Hello World” program
- Program to print some numbers
- Program to print the size in bytes of the fundamental data types
- Computation of Fibonacci numbers
- A more elaborate example (integer factorization)
- Final example (rational approximation)

Study the provided C source code carefully. The syntax of the Java programming language was inspired by that of the C language.

How to compile and run a program (GNU/Linux)

A C program is composed by one or more `.c` source files and by zero or more `.h` files (included by the `.c` source files). To compile the program under GNU/Linux, the following command should be used:

```
cc -Wall -O2 source_files... -o executable_name -lm
```

Replace `source_files...` by the list of the `.c` source files, and replace `executable_name` by the name you desire to give to the executable file. If `-o executable_name` is omitted from the command line, the executable will get by default the name `a.out`. The option `-Wall` instructs the compiler (`cc`) to give you a warning whenever you use dubious C code (such as using an uninitialized variable). The option `-O2` instructs the compiler to optimize the program for speed. The linker option `-lm`, that must be placed at the end, instructs the compiler to link the program with the math library (so that functions like `sin()` and the like are properly taken care of). For example, if your program is composed by the files `source1.c`, `source2.c`, if they include the file `source.h`, and if you desire the executable to be named `prog`, the command should be

```
cc -Wall -O2 source1.c source2.c -o prog -lm
```

and you can run it on a terminal using the command

```
./prog
```

You can automate the process of compiling the program using a `makefile`. Put the text (beware of the tab, denoted below by an arrow)

```
prog:   source1.c source2.c source.h
———————→cc -Wall -O2 source1.c source2.c -o prog -lm
```

in a file named either `Makefile` or `makefile` (the second line begins with a tab). Running the command

```
make
```

will recompile your program if at least one of the source files has changed since the last compilation.

How to manage archives

All source code files for this class, together with the `makefile` needed to (re)compile all programs, is distributed in the compressed tar archive `P01.tgz`. On a GNU/Linux system, to extract the files it holds on a terminal, go to the directory (folder) where you want to extract the files (use the `cd` command to do this), and then use the command

```
tar xzvf P01.tgz
```

to extract the files. They should appear in a new directory named `P01`. To create an archive use the command

```
tar czvf name_or_your_archive.tgz list_of_files_and_directories_to_put_in_the_archive
```

The “Hello World” program

The `hello.c` file contains the C code

```c
/*
** Hello world program
*/

#include <stdio.h>

int main(void)
{
  puts("Hello world!");
  return 0;
}
```

Compile it and run it. Modify the code to print “Hello X”, where X is your name.

Program to print some numbers

The following program (`table.c`) prints the first ten positive integers, their squares, and their square roots.

```
#include <math.h>
#include <stdio.h>

void do_it(int N)
{
  int i;

  printf(" n n*n      sqrt(n)\n");
  printf("-- --- -----------------\n");
  for(i = 1;i <= N;i++)
    printf("%2d %3d %17.15f\n",i,i * i,sqrt((double)i));
}

int main(void)
{
  do_it(10);
  return 0;
}
```

Compile and run it. Modify the program to print the sine and cosine of the angles $0, 1, 2, \ldots, 90$ (in degrees) and to place the output in a file named `table.txt`.

Hints:

- Use the help system (on GNU/Linux use the command `man`) to study how the formatting string of the `printf` function is specified. Study also the functions `fopen`, `fclose` and `fprintf`.
- The argument of the `sin` and `cos` functions are in radians; the value of π is given by the symbol `M_PI`, which is defined in `math.h`. To convert from degrees to radians multiply the angle by `M_PI/180.0`.

Program to print the size in bytes of the fundamental data types

The file `sizes.h` contains the code

```
#ifndef SIZES_H
#define SIZES_H
void print_sizes(void);
#endif
```

The file `sizes.c` contains the code

```
#include <stdio.h>
#include "sizes.h"
void print_sizes(void)
{
  printf("sizeof(void *) ...... %d\n",(int)sizeof(void *)); // size of any pointer
  printf("sizeof(void) ........ %d\n",(int)sizeof(void));
  printf("sizeof(char) ........ %d\n",(int)sizeof(char));
  printf("sizeof(short) ....... %d\n",(int)sizeof(short));
  printf("sizeof(int) ......... %d\n",(int)sizeof(int));
  printf("sizeof(long) ........ %d\n",(int)sizeof(long));
  printf("sizeof(long long) ... %d\n",(int)sizeof(long long));
  printf("sizeof(float) ....... %d\n",(int)sizeof(float));
  printf("sizeof(double) ...... %d\n",(int)sizeof(double));
}
```

The file `main.c` contains the code

```
#include "sizes.h"
int main(void)
{
  print_sizes();
  return 0;
}
```

Study the way the program is split among the three files. Compile and run the program. On a 64-bit machine, add either -m32 or -m64 to the compilation flags and check if any of the sizes reported by the program changes.

Computation of Fibonacci numbers

The program in the file `fibonacci.c` computes Fibonacci numbers using the four different methods briefly described in this slide. Compare the code of this file with the one presented in the slide.

Compile and run the program. Make graphs of the execution times of each or the four functions reported by the program. Explain why the function `F_v1` is extremelly slow (hint: compare the execution time of that function with the value it returns). Estimate how long your computer will take to compute $\boldsymbol{F}_{60}$ using the `F_v1` function.

A more elaborate example (integer factorization)

The following program (`factor.c`) computes the factorization of an integer.

```c
#include <stdio.h>
#include <stdlib.h>

int factor(int n,int *prime_factors,int *multiplicity)
{
  int d,n_factors;

  n_factors = 0;
  for(d = 2;d * d <= n;d = (d + 1) | 1) // d = 2,3,5,7,9,...
    if(n % d == 0)
    {
      prime_factors[n_factors] = d; // d is a prime factor
      multiplicity[n_factors] = 0;
      do
      {
        n /= d;
        multiplicity[n_factors]++;
      }
      while(n % d == 0);
      n_factors++;
    }
  if(n > 1)
  { // the remaining divisor, if any, must be a prime number
    prime_factors[n_factors] = n;
    multiplicity[n_factors++] = 1;
  }
  return n_factors;
}
```

```c
int main(int argc,char **argv)
{
  int i,j,n,nf,f[16],m[16]; // 16 is more than enough...

  for(i = 1;i < argc;i++)
    if((n = atoi(argv[i])) > 1)
    {
      nf = factor(n,f,m);
      printf("%d = ",n);
      for(j = 0;j < nf;j++)
        if(m[j] == 1)
          printf("%s%d",(j == 0) ? "" : "*",f[j]);
        else
          printf("%s%d^%d",(j == 0) ? "" : "*",f[j],m[j]);
      printf("\n");
    }
  return 0;
}
```

Study the program. Compile and run it. Why is the program slow when n is a prime number larger than $\mathbf{46339^2}$, such as $\mathbf{2147483647}$? (Hint: arithmetic overflow in the signed integer data type.) Get rid of that programming bug.

Homework challenge: Modify the program so that it outputs all possible divisors of `n`. [Hint: there are `(1+m[0])*(1+m[1])*...*(1+m[nf-1])` divisors.]

Final example (rational approximation)

The program in the file `rational_approximation.c` computes the best rational approximation to a given real number using two different approaches. One is based on a slow but straightforward brute force search for the best solution, and the other (fast, but not the fastest possible!) is based on some interesting mathematical properties of best rational approximations.

Study the program. Pay attention to the data types that are defined and how preprocessor directives can enable or disable (at compile time) parts of the program. Try to understand how the (slow) brute force search works. Attempting to understand the mathematics behind the fast method lies outside the scope of AED (for the curious, it uses the so-called Stern-Brocot tree).

Compile the program and run it. Confirm that the two methods give the same result. Experiment with other real numbers. For example, if the line "`x = M_PI;`" is replaced by the line "`x = exp(1.0);`" or by the line "`x = M_E;`" the program computes best rational approximations to e (base of the natural logarithms).

Modify the program to count and print, if `DEBUG` is negative, the number of tests `e < best_e` that are performed by each of the two functions. Do this for several interesting real numbers, such as π, e, $\sqrt{2}$, and, say, $(1+\sqrt{5})/2$. What can you say about the growth of the number of tests as a function of `max_den` for each of the two functions?

Measure approximately the time it takes your computer to compute

```
best_rational_approximation_slow(M_PI,100000000u)
```

and to compute

```
best_rational_approximation_fast(M_PI,100000000u)
```

Try also other values of the second argument and make graphs of the execution time versus this second argument.

The C++ programming language — TP.02 —

Summary:

- Overview of the C++ programming language
- Some differences between C and C++
- Classes
- Templates
- Exceptions

[Remark: C++ is a very large and complex programming language (some say that it is far to much complex). For AED we will only need a relatively small subset of what it has to offer. The rest, although important, will not even be mentioned in these slides.]

Recommended bibliography for this lecture:

- **Thinking in C++. Volume One: Introduction to Standard C++**, Bruce Eckel, second edition, Prentice Hall, 2000.
- **Thinking in C++. Volume Two: Practical Programming**, Bruce Eckel and Chuck Allison, Prentice Hall, 2003.
- **Online reference documentation about C++**
- **C++ Annotations**, Frank B. Brokken, 2015.
- **C++ Primer**, Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo, fifth edition, Addison-Wesley, 2013.

Overview of the C++ programming language

The C++ programming language is a large and complex programming language. It is a superset of the C language (hence the name C++). Most programs written in C are also valid C++ programs. The C++ language has the following features, which are not present in the C language (this list is not exhaustive):

- It is possible to pass values by reference to a function, without making the use of pointers explicit.
- It is possible to define functions with the same name but with different lists of arguments (function overloading).
- The last arguments of a function may have default values (great if one wants to add a new parameter to a function without modifying the code already written that calls the function).
- One can create a data type and the functions that manipulate it (a `class`) in a much more elegant way. It is even possible to make the standard arithmetic operators, such as + and −, work with arguments of the new data type. It is possible to hide the internal details of the data type, so that we can change them without affecting the parts of the program that use the data type.
- One can create so-called name spaces to better control which symbols (such as variables and function names) are visible in each part of our code.
- It is possible to create templates of functions and classes (generic programming).
- It is possible to handle exceptions in a disciplined way (in C one would have to use `goto` statements or, in some cases, the ugly `setjump` and `longjump` functions)

It is customary to use the extension `.cpp` (meaning C-plus-plus) in the names of the source files of a C++ program (`.cc`, `.cxx`, `.c++`, and even `.C`, are also sometimes used).

Some differences between C and C++ (part 1)

C++ allows arguments to be passed by reference. This is accomplished by placing an & before the argument name. In the following code, we show on the left how this has to be done in C and on the right we show how this can be done in C++ (we may also use the left version, but the right one is more elegant, as there are no explicit pointer dereferences):

```
// C; called as follows: swap(&var1,&var2);
void swap(int *x,int *y)
{
  int tmp = *x;
  *x = *y:
  *y = tmp;
}
```

```
// C++; called as follows: swap(var1,var2);
void swap(int &x,int &y)
{
  int tmp = x;
  x = y:
  y = tmp;
}
```

C++ allows functions with the same name but different argument lists to coexist. For example, the following code is valid in C++:

```
int    square(int    x) { return x * x; }
double square(double x) { return x * x; }
```

The code `square(1)` will call the first function, because its argument is an `int`, and the code `square(1.0)` will call the second function, because its argument is a `double`. It is, however, illegal to have two functions with the same name and with the same argument list (same number of arguments and same types), but with different return types. For example, the function

```
double square(int x) { return double(x) * double(x); }  // double(x) does the same as (double)x
```

cannot coexist with the first function defined above.

Some differences between C and C++ (part 2)

C++ allows the specification of default values for the last arguments of a function. This is done by providing initializations (with the default values) in the argument list of the function. It is actually better to put the initializations in the function prototype, as in the following example:

```
int f(int x,int y = 2,int z = 3); // function prototype (usually placed in a header file)

int f(int x,int y,int z) // actual definition of the function
{
  return x + 2 * y + 3 * z;
}
```

In this case we not only can call the `f` function with three arguments as usual, but we can also call it with two (the third, `z`, will get the value 3, as specified in the function prototype), and with one (the second and third will get, respectively, the values 2 and 3). [**Warning:** this feature of the C++ language should be used with extreme care when the function is also overloaded.]

C++ allows the definition of variables in almost any place inside a compound statement. In (old) C, that is only allowed at the beginning of a compound statement. The following code is valid in C++ (it is also valid in modern C dialects):

```
int i;                      // i defined here
for(int j = 0;j < 10;j++)   // j defined here; it will cease to exist at the end of the for cycle
{
  i = 2 * j;
  int k = i + 2 * j;        // k defined here (definition after a statement is allowed)
  cout << k;                // print the value of k
}
```

Some differences between C and C++ (part 3)

In C++ it is possible to control the visibility of symbols by placing them is a name space. In the following code we define two name spaces and put in each of them a global variable and a function (same names but different name spaces; of course this is not recommended but sometimes it cannot be avoided):

```
namespace NEW
{
  static int t_bytes;
  int f(int x) { return 2 * x; }
}
namespace OLD
{
  static int t_bytes;
  int f(int x) { return 3 * x; }
}
```

To get a specific variable or function, place the name of the name space followed by `::` before the variable or function name. For example, `NEW::t_bytes` is the `t_bytes` of the `NEW` name space. It is also possible to say

```
using NEW::t_bytes;
```

and from that point on `t_bytes` will be synonymous with `NEW::t_bytes` (or course, for this to work no symbol with the name `t_bytes` can already exist in the current name space). It is also possible to import all symbols from a name space, thus making all of them available without the `name_space::` prefix. For example,

```
using namespace OLD;
```

will make `t_bytes` a synonym of `OLD::t_bytes` and `f` a synonym of `OLD::f`.

The `std` name space is reserved for standard library functions.

Some differences between C and C++ (part 4)

It is possible to call C functions from a C++ program (the calling conventions are the same, as are the fundamental data types), so that is a question of using the proper function names. This is actually a problem that has to be solved, because the function name that the compiler uses internally is not simply the name of the function: it has also to encode the types of its arguments (this has to be done because a function may be overloaded). The internal names are said to be "mangled." C functions do not have mangled names, so the compiler has to be told to use (or generate) unmangled function names. The following example shows how this is done:

```
extern "C" int f(int x);
extern "C"
{
  int g(int x);
  int h(int x);
}
```

Type casts, in C++, although they can be done just as in C, should be done in the form of a function call, as illustrated in the following code:

```
int i = (int)1.0;  // a C-style cast   (try not to use)
int j = int(1.0);  // a C++-style cast (use)
```

This allows the compiler to better check if the cast makes sense.

In C++, use `nullptr` instead of `NULL`. It serves the same purpose but its use is safer, because in C++ `NULL` is defined to be the constant **0** (and not a pointer to `void` with the value 0 as it is in C).

Some differences between C and C++ (part 5)

Memory can be allocated with the `new` operator and deallocated with the `delete` operator. When used to allocate an instance of a class `new` calls automatically its constructor. Likewise, `delete` calls its destructor. The argument of the `new` operator is a data type. Its return value is a pointer to that type. Note, however, that as in C when one specifies an array what one gets is a pointer to its first element (the constructor of the element type is called for each one of elements of the array). The argument of the `delete` operator should be a pointer received from the `new` operator; if it is not all hell can break loose. `delete` does not have a return value. For array types, the operator `delete[]` calls the destructor for each of the elements of the array (the `delete` operator calls the destructor only for the first element).

The following example shows how `new` and `delete` can be used.

```
int *p_i = new int;             // get memory to an integer
*p_i = 3;                       // give it the value 3
delete p_i;                     // free its memory
p_i = new int(10);              // get memory to another integer at initialize it with the value 10
double *p_d = new double[100];  // get memory for an array of 100 doubles
delete[] p_d;                   // free its memory
class abc;                      // class fully declared elsewhere
abc *pc = new abc;              // pointer to an instance of class abc
```

If there is not enough memory the `new` operator throws a `std::bad_alloc` exception.

Classes (part 1)

Roughly, a C++ class is the combination of a C structure with a set of functions that manipulate the structure. It is a great way to compartmentalize our code, safely hiding the details of how the structure and associated functions are actually implemented. A class is declared just like a structure, but with some extra ingredients:

- while the members of structures have to be data types, members of classes may also be functions.
- when an instance (an object) of a class is created, a constructor member function is called to initialize the data fields of the instance.
- when an instance of the class is destroyed, a destructor member function is called to do any necessary cleanup work.
- some of the members of the class, be they data fields or functions, can be made public, i.e., visible to the entire program, or they can be made private, i.e., visible only by the code that implements the class.
- some data fields may act like global variables, existing only one instance of them irrespective of the number of instances of the class that were created (in C one would have to use a separate global variable to get the same effect; in C++ it is an integral part of the class).

A class with name `CLASS_NAME` is declared as follows:

```
class CLASS_NAME
{
  private:  // private members part
    // put declarations (of functions) and definitions (of functions or data fields) here
  public:   // public members part
    // put declarations (of functions) and definitions (of functions or data fields) here
};
```

We may have several private and public parts. Class member functions may be defined outside the class declaration.

Classes (part 2)

The following example presents the code of a very simple class:

```
class dot
{
  private:
    static int n_dots;  // counts the number of dots created
    double d_x;         // x coordinate
    double d_y;         // y coordinate
  public:
    dot(double x,double y) { n_dots++; d_x = x; d_y = y; } // constructor
    ~dot(void) { n_dots--; }                               // destructor
    double x(void) { return d_x; }                         // definition
    double y(void);                                        // declaration (prototype)
    int number_of_dots(void) { return n_dots; }            // definition
};
double dot::y(void) { return d_y; }                        // definition
int main(void)
{
  dot d(0.0,0.0);     // create a dot; almost the same as "dot d = dot(0.0,0.0);"
  double x = d.x();   // get the x coordinate of d
  // more code
}
```

Note that the name of the constructor function is the name of the class and that the name of the destructor is the name of the class preceded by ~. Note also that inside member functions the names of the data members of the class can be used without reference to the class instance (see discussion of the `this` pointer in the next slide).

Classes (part 3)

When a member function of a class is called it receives an extra hidden argument named `this`. This argument is a pointer to the memory area that holds the data of the class instance that is being used. For example, the member function `dot::y` of the previous slide, shown on the left hand side of the following code, is transformed by the compiler into the code shown on the right hand side:

```
// C++ code
double dot::y(void)
{
  return d_y;
}
```

```
// possible C implementation
double dot_y_implementation(dot * const this)
{
  return this->d_y;
}
```

We can use the `this` pointer in our code (in non static member functions!) without declaring it.

The syntax used to access `struct` data fields is also used to access `class` data members and `class` member functions. For example, if `d_x` had been made `public` in the previous slide, we could have used it as follows:

```
dot d;
d.d_x = 3.0; // set the d_x field of d to 3
```

Since it was made `private` that is not allowed, and we need to provide member functions to set and get its value (if we want to make that data member available to the rest of the program). Calling a member function is done in the same way:

```
double y = d.y();
```

will call the public member function `dot::y` with the `this` pointer set to `&d`.

Templates

Templates are a way to write code in a generic way, without specifying beforehand the data types or other parameters that will be used in a data structure or function. The idea is to write code once, and to use it many times. One writes a template for the code, keeping some data types, and possibly other parameters, unspecified. For a function, this is done as in the following example:

```
template <typename T> T f(T x)
{
  return T(7) * x;  // multiply x by 7; 7 is cast to type T (it must be possible to do that)
}
```

Here, the function template has one generic type named `T` (it can have more), and describes a family of functions, named `f`, whose purpose it to multiply its argument by 7. (Of course this could also be done in a far simpler way, but our purpose here is the describe how a template works.) To use the template to define an actual function, do as follows:

```
int    i = f<int>(3);     // i = 7 * 3
double d = f<double>(5.0) // d = 7.0 * 5.0
```

Class templates are done similarly (here using two generic types):

```
template <typename T1,typename T2> class XYZ
{
  private:
    T1 a_member_variable_of_type_T1;
    T2 a_member_variable_of_type_T2;
    // ...
};
```

and used similarly: `XYZ<int,double> a_variable_of_class_XYZ_int_double;`.

Exceptions

In a program, one possible way to deal with an unexpected case (such as trying to compute the square root of a negative number, when that was thought not to be possible to happen) is just to terminate it. In mission critical applications that is not desirable. What one needs is a way to handle gracefully the unexpected condition (after all, it may have been the result of memory corruption due to a very rare cosmic ray, and not the fault of the program). C++ implements a mechanism (try–catch) that can do that. The idea is to surround the program area we want to protect with a "safety net," that catches these unexpected events. We put our normal code in a `try` block, and we put the recovery code in one or more `catch` blocks. Exceptions (the unexpected events) are signaled by "throwing" an exception, using a `throw` statement. The following example will make things clear:

```
double sqrt(double x)
{
  if(x < 0.0) throw 0;  // throw an integer exception with the value 0
  return sqrt(x);
}
int main(void)
{
  try
  {
    cout << sqrt(-1.0) << endl;
  }
  catch(int i)
  {
    cout << "integer exception number " << i << " caught" << endl;
    exit(1);
  }
}
```

The C++ programming language (exercises)
— P.02 —

Summary:

- How to compile a C++ program (linux)
- The "Hello World" program
- Program to print some numbers
- Program that uses function overloading
- Program that uses a class
- Program that uses a function template
- Program that uses a class template
- Program that uses an exception handler
- Homework

How to compile a C++ program (linux)

A C++ program is composed by one or more `.cpp` source files and by zero or more `.h` files (included by the `.cpp` source files). To compile the program under linux, the following command should be used:

```
c++ -Wall -O2 source_files... -o executable_name -lm
```

Replace `source_files...` by the list of the `.cpp` source files, and replace `executable_name` by the name you desire to give to the executable file. All that was said about compiling C programs also holds for compiling C++ programs (except that now the compiler program is called `c++` and not `cc`).

The "Hello World" program

Extract the file `hello.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Compare it with the `hello.c` given in the P.01 class (you can find it in the `P01.tgz` archive). Modify the program to print the numbers $1, 2, 3, \ldots, 10$.

Program to print some numbers

Extract the file `table.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Compare it with the C program given in the P.01 practical class. Modify the program to print in another column the cubic roots of the numbers of the second column. (Hint: the function `cbrt` computes a cubic root.)

Program that uses function overloading

Extract the file `overload.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Add two other `show` functions to it, to print i) a char, and ii) an array of 3 integers (fixed array size). For example,

```
show('a');
```

should print

```
char: a
```

and

```
int a[3] = { 2,7,-1 };
show(a);
```

should print

```
array: [2,7,-1]
```

Test your new program.

Program that uses a class

Extract the file `person.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Set `debug` to 1 and recompile and rerun the program. Is the output the same as before? Why? Change the program so that the class `person` also stores the phone number of a person.

Program that uses a function template

Extract the file `f_template.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Add another function to the program than computes the mean of the elements of the array. The return type of that new function should be `double`.

Program that uses a class template

Extract the file `c_template.cpp` from the archive `P02.tgz`. Study, compile, and run the program.

Program that uses an exception handler

Extract the file `exception.cpp` from the archive `P02.tgz`. Study, compile, and run the program. Modify the value of the `special_value` constant so that the exception of type `int` is triggered before the exception of type `double` has a change to be triggered.

Homework

Add exceptions to the `c_template` program. (**Hint:** create an enumerated type with values `stack_full` and `stack_empty` and use then as the exception values.)

Computational complexity — TP.03 —

Summary:

- Algorithms
- Abstract data types
- Computational complexity
- Algorithm analysis
- Asymptotic notation
- An example
- Useful formulas

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **The Algorithm Design Manual**, chapter 2, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, chapters 1 and 3, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Análise da Complexidade de Algoritmos**, chapter 1, António Adrego da Rocha, FCA.

Algorithms (part 1, definition)

An algorithm is a detailed description of a method to solve a given problem. To properly specify the problem it is necessary to specify its input, which encodes the information necessary to uniquely identify each instance of the problem, and its output, which encodes the solution to the problem. It is also necessary to specify the properties that the output must have in order for it to be a solution to the problem for the given input (in other words, we must also specify what the problem actually is). An algorithm is a finite sequence of instructions that explain, in painstaking detail, how to produce a valid output (a solution to the problem) given a valid input. These instructions can be given in a natural language, such as English, or in a form closer to how an algorithm may be actually implemented in a given programming language (pseudocode).

An algorithm:

- must be **correct**, i.e., it must produce a valid output for **all** possible valid inputs;
- must be **unambiguous**, i.e., its steps must be precisely defined;
- must be **finite**, i.e., it must produce a valid output in a finite number of steps for **all** possible valid inputs (we are not interested in "algorithms" that may take forever to produce a valid output);
- should be **efficient**, i.e., it should do the job as quickly as possible;
- can be **deterministic**, if the sequence of steps that produces the output depends only on the input, or it
- can be **randomized**, if the sequence of steps that produces the output depends on the input and on random choices.

Algorithms (part 2, example)

The following algorithm is a possible way to sort a sequence of numbers:

Algorithm: Generic exchange sort.

Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$, with $n > 0$.

Output: a permutation (reordering) $b_1, b_2, \ldots, b_n$ of the input sequence such that $b_1 \leqslant b_2 \leqslant \cdots \leqslant b_n$.

Steps:

1. [Copy input.] For $i = 1, 2, \ldots, n$, set b_i to a_i.
2. [Deal with a special case.] If $n = 1$ terminate the algorithm.
3. [Are we done?] Find a pair of indices (i, j) such that $i < j$ and $b_i > b_j$. Terminate the algorithm if such a pair does not exist.
4. [Exchange.] Exchange b_i with b_j. Return to step 3.

Is the algorithm correct? Yes. For $n > 1$ the algorithm can only terminate when the current b sequence is sorted in non-decreasing order.

Is the algorithm unambiguous? No, because step 3 does not specify how the pair of indices (i, j) is to be found. That is a subproblem that also has to be fully specified. However, no matter how this is done, the algorithm is correct.

Is the algorithm finite? Yes. Since the maximum possible number of exchanges is $n(n-1)/2$ — that is the number of different pairs (i, j) with $i < j$ — sooner or latter the search in step 3 to find an acceptable pair (i, j) will fail.

Is the algorithm efficient? That depends on how the pair of indices is found in step 3. (If that is done from scratch every time, it will be inefficient.)

Abstract data types

Most algorithms need to organize the data they work with in certain ways. So, all modern programming languages allow the programmer to define new data types that are not available natively in the language. More often than not each particular kind of data used by an algorithm can be stored in more than one way but is used in essentially the same way. In those cases the programmer should choose the more efficient storage organization for each kind of data. What constitutes the best storage organization can change during the development of the program. For example, at first it might be thought that the space required to store the data is more important that the time it takes to query or modify it. Later on it may turn out that it is the other way around. So, a good programmer will pay considerable attention to the operations (transformations, queries) that the algorithm needs to perform on its data, and will define data types not by the specific way they store their information but by what operations are allowed to be performed on the information that is supposed to be stored in each one of them. He/she will design abstract data types.

An abstract data type is a data type that exposes to the rest of the program the ways the data it stores can be queried or modified (the interface), without exposing to the rest of the program its internal workings (the implementation), be it how the data is actually stored or how the queries/modifications are actually performed. This will make the program more modular, because as long as the interface of an abstract data type is not changed, changes in its implementation will not affect the rest of the program.

A proper specification of the interface defines not only the names and arguments of the operations that can be performed on instances of the data type (that has to be placed in the source code) but also what their side effects are (that should be placed in the source code in the form of comments). Some programming languages provide facilities (assertions, and pre- and post-conditions) that help ensure that the interface of a data type is used correctly.

Computational complexity (part 1, the RAM model of computation)

To be able to compare the efficiency of different algorithms one needs a model of computation. A model of computation quantifies how much work is needed to perform an elementary task, such as performing an arithmetic operation or a memory access. The RAM model of computation is one of the simplest we can use. It is based on the notion of a Random Access Machine, abbreviated as RAM. Under this model of computation

- an elementary arithmetic operation, such as +, − and the like, a comparison, and an assignment, of numbers with a given fixed number of bits, uses one time step,
- the operation of calling a subroutine (just the call, not the actual work done by the subroutine), of evaluating a numerical transcendental function such as `sin(x)`, and of following a conditional branch, also uses one time step,
- loops, and subroutines, have to be broken down into their individual constituents,
- a memory access, be it a read or a write access, also uses one time step, and
- the memory space used to store a number with a given fixed number of bits uses one unit of space.

This is, of course, a very simplistic model of what happens on a true processor. For example, on a modern processor a division takes much more time than an addition. It is nonetheless a useful model, because in the worst case it deviates (above and below) from what happens on a modern processor by a constant factor.

The computational complexity of an algorithm gives the number of time steps, and the number of units of space, required by the algorithm to solve a given problem. It is a function of the size of the algorithm's input. The size of the input is usually either the number of its data items (say, the number of elements of an array), or one of the inputs itself (say, the exponent in an exponentiation algorithm).

Computational complexity (part 2, a simple example)

Consider the following very simple algorithm:

Algorithm: Mean.

Input: a sequence of n numbers $a_1, a_2, \ldots, a_n$, with $n > 0$.
Output: the arithmetic mean $\mu = \frac{1}{n}\sum_{i=1}^{n} a_i$
Steps:

1. [Initialization.] Set s to a_1.
2. [Sum.] For $i = 2, 3, \ldots, n$, add a_i to s.
3. [Return.] Terminate the algorithm, outputting s/n as the value of μ.

To determine the computational complexity of this algorithm we need to count the number of elementary operations it performs. Step 1 requires one time step to read a_1 (it is read directly into s, so no assignment is needed). For each value of i, step 2 requires five time steps: one to read a_i, another one to add it to s, and another three to increment i, to compare it with n, and to perform a conditional jump according to the result of the comparison. Step 3 requires two time steps, one to perform the division and another to terminate the algorithm (we consider it to be a subroutine return, so we also count it). Since there are $n - 1$ values in step 2, the total number of time steps used by the algorithm is $T(n) = 1 + 5(n - 1) + 2 = 5n - 2$.

Doing such a detailed analysis is usually not necessary. All that one is usually interested in is knowing how fast $T(n)$ grows when n grows. For large n, in this case $T(n)/n$ is almost constant (linear complexity). Knowing that is usually more important than knowing that $T(n) \approx 5n$; knowledge of the growth constant, 5 in this case, although useful, is in most cases an overkill. (After all, the true constant will depend on the processor where the algorithm will be run and on the optimizations of the code made by the compiler.)

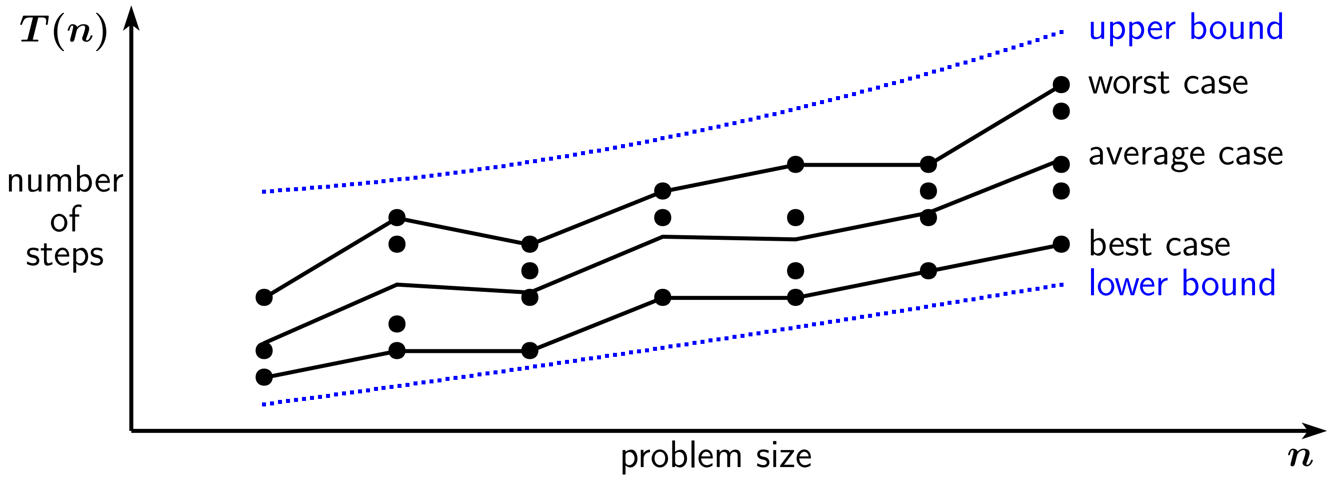
Algorithm analysis (part 1, formal and empirical analysis)

In the last slide we did a formal complexity analysis of a simple algorithm. We have taken the trouble to account for every elementary operation performed by the algorithm. That was easy to do because the algorithm was very easy. For more complex algorithms, specially for those with a flow of execution that depends on the input, that is hard to do, even if one takes a probabilistic approach to the problem. (In a probabilistic approach every conditional branch that depends on the input data has a specific probability of being taken. Assigning these probabilities usually requires a great deal of mathematical sophistication on the part of the person doing the formal complexity analysis.)

When a formal complexity analysis it too difficult to perform, or when we are just too lazy to do it (or when we do not know enough to do it), one can do an empirical complexity analysis. An empirical analysis is based on experimentation. One implements the algorithm using a suitable programing language, and one either adds code to count the number of operations done by the program (this is called instrumenting the program), or one just measures its execution time. In both cases, one has to select a good set of typical inputs of various sizes. The experimental values of $T(n)$ measured in this way can then be plotted as a function of n to see how they grow. With luck, it will be possible to find out a mathematical formula that fits the experimental observations reasonably well.

Algorithm analysis (part 2, worst, best, and average cases)

The worst case time of an algorithm is the function defined by the maximum number of time steps used by the algorithm to deal with any valid input of size n. The best case and average case times are defined in the same way. (Best, worst, and average spaces can also be defined.) As shown in the following figure, these functions may on occasion decrease when the problem size increases.



Best and worst case times are usually easier to determine than the average case time. Furthermore, exact best and worst times are usually more difficult to determine, and to use, than smooth bounds of these functions (in blue in the figure). Obviously, we need a lower bound for the best case and an upper bound for the worst case.

We are usually interested in knowing how fast the lower and upper bounds grow when the size of the problem increases. For example, $T_1(n) = 3n^2$ and $T_2(n) = 10n \log n$ grow at clearly distinct rates, while $T_3(n) = 4n^2$ and $T_4(n) = 3n^2 + 10n$ do not. Mathematicians have a way to express succinctly these differences: asymptotic notation. Using asymptotic notation we talk about the best, average, and worst time **complexity** of an algorithm.

Asymptotic notation (part 1, definitions)

Asymptotic notation allows us to hide unimportant details about how fast a function grows. For example, when n is a **very large number** it is overkill to know exactly that $T_1(n) = 2n^2 + 3000n + 5$ and that $T_2(n) = 10n^2 + 100n - 23$. For that kind of numbers all that matters is that $T_1(n)$ is approximately $2n^2$ and that $T_2(N)$ is approximately $10n^2$, so that $T_2(n)$ will be about 5 times larger than $T_1(n)$. In asymptotic notation the only detail that is kept about $T_1(n)$ and $T_2(n)$ is that they grow quadratically; even the constant factor that multiplies n^2 is hidden behind the mathematical notation.

Asymptotic notation comes in one of several forms (all functions and constants are assumed to be positive):

- [**Big Oh notation**] The Big Oh notation is useful to deal with **upper bounds**. The notation
  $$f(n) = O\bigl(g(n)\bigr)$$
  means that there exists an n_0 and a constant C such that, for all $n \geqslant n_0$, $f(n) \leqslant Cg(n)$.
- [**Big Omega notation**] The Big Omega notation is useful to deal with **lower bounds**. The notation
  $$f(n) = \Omega\bigl(g(n)\bigr)$$
  means that there exists an n_0 and a constant C such that, for all $n \geqslant n_0$, $f(n) \geqslant Cg(n)$.
- [**Big Theta notation**] The Big Theta notation is useful to deal with **upper and lower bounds** that have the **same form**. The notation
  $$f(n) = \Theta\bigl(g(n)\bigr)$$
  means that there exists an n_0 and two constants C_1 and C_2 such that for all $n \geqslant n_0$ one has $C_1 g(n) \leqslant f(n) \leqslant C_2 g(n)$.
- [**small oh notation**] The notation
  $$f(n) = o\bigl(g(n)\bigr)$$
  means that $f(n)$ grows slower than $g(n)$, i.e., that $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Asymptotic notation (part 2, properties)

The notation introduced in the previous slide can be used in an expression. For example, $f(n) = n^2 + O(n)$ means that $f(n)$ deviates from n^2 by a quantity that is $O(n)$, that is, whose absolute value is bounded by a multiple of n. (The definitions introduced before can be extended to general functions by using absolute values.) We may, for example, say that $O(2n^2 - \log n) = O(3n^2 + 100n - 23)$, because the absolute value of both functions can be bounded by a quadratic, i.e., both are $O(n^2)$.

What can we say about $O\big(f(n) + g(n)\big)$? There are three cases to consider:

- $f(n) = \Theta\big(g(n)\big)$, which implies that $g(n) = \Theta\big(f(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(f(n)\big)$.
- $f(n) = o\big(g(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(g(n)\big)$.
- $g(n) = o\big(f(n)\big)$. In this case $O\big(f(n) + g(n)\big) = O\big(f(n)\big)$.

In all cases the function that grows faster "wins" (remember, upper bounds).

How about $\Omega\big(f(n) + g(n)\big)$? Here the function that grows slower "wins" (remember, lower bounds).

How about $\Theta\big(f(n) + g(n)\big)$? If $f(n) = \Theta\big(g(n)\big)$ then $\Theta\big(f(n) + g(n)\big) = \Theta\big(f(n)\big)$. Otherwise, the lower and upper bounds of $f(n) + g(n)$ do not grow in the same way, and so the Big Theta notation cannot be used.

How about a multiplication by a (positive) constant c? Easy. The constant is discarded, as it is implicit in the notation: $O\big(cf(n)\big) = O\big(f(n)\big)$, $\Omega\big(cf(n)\big) = \Omega\big(f(n)\big)$, and $\Theta\big(cf(n)\big) = \Theta\big(f(n)\big)$.

How about the product of two functions? Easy. Products are retained: $O\big(f(n)g(n)\big) = O\big(f(n)\big)O\big(g(n)\big)$, $\Omega\big(f(n)g(n)\big) = \Omega\big(f(n)\big)\Omega\big(g(n)\big)$, and $\Theta\big(f(n)g(n)\big) = \Theta\big(f(n)\big)\Theta\big(g(n)\big)$.

Asymptotic notation (part 3, useful information)

To determine the truth or falsewood of each of the statements $f(n) = O\big(g(n)\big)$, $f(n) = \Omega\big(g(n)\big)$, $f(n) = \Theta\big(g(n)\big)$, $f(n) = o\big(g(n)\big)$, one usually only needs to find out how $\frac{f(n)}{g(n)}$ behaves when n grows to infinity. In pathological cases where $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not exist, one will need to compute $\liminf_{n\to\infty} \frac{f(n)}{g(n)}$ and $\limsup_{n\to\infty} \frac{f(n)}{g(n)}$. We will not encounter these cases in AED. We have:

$$\lim_{n\to\infty}\left|\frac{f(n)}{g(n)}\right| = \begin{cases} 0, & \Rightarrow \quad f(n) = o\big(g(n)\big) \quad f(n) = O\big(g(n)\big) \\ > 0, & \Rightarrow \quad f(n) = O\big(g(n)\big) \quad f(n) = \Theta\big(g(n)\big) \quad f(n) = \Omega\big(g(n)\big) \\ \infty, & \Rightarrow \quad f(n) = \Omega\big(g(n)\big) \end{cases}$$

Informally, $f(n) = O(g(n))$ when $f(n)$ does not grow faster than $g(n)$.

While mentally comparing two functions, one can discard all constants and all lower order terms. Note that n^a grows faster than $\log n$ for any $a > 0$, that n^a grows faster that n^b when $a > b \geqslant 0$, that a^n grows faster that b^n when $a > b \geqslant 1$, that a^n grows faster that n^b for any $a > 1$, and that $n! \approx \left(\frac{n}{e}\right)^n\sqrt{2\pi n}\big(1 + \frac{1}{12n}\big)$ (this is part of Stirling's asymptotic formula for $n!$) grows faster than any n^a or a^n. Note, however, that $n!$ grows slower than n^n. For example, in the comparison of

$$f(n) = 100n^3 + 10^{1000}n^{14/5} \qquad \text{with} \qquad g(n) = 10^{-1000}n! + 2^n$$

it is enough to compare

$$n^3 \qquad \text{with} \qquad n!$$

This is so because n^3 grows faster that $n^{14/5} = n^{2.8}$, and because $n!$ grows faster than any power. In this case $n!$ wins by a huge margin, and so $f(n) = o\big(g(n)\big)$. Of course, this also means that $f(n) = O\big(g(n)\big)$. In this particular case, when $n \leqslant 810$, $f(n)$ is actually larger than $g(n)$.

Asymptotic notation (part 4, examples)

The following examples may help understand better the properties of the Big Oh notation:

- $10n^2 + 30n + 13 = O(n^2)$, because for $n \geqslant 31$ we have $10n^2 + 30n + 13 \leqslant 11n^2$. We have chosen $C = 11$, thus forcing n_0 to be at least 31. Any value of C larger than 10 would also work, but the closer it gets to 10 the larger n_0 has to be.
- $10n^2 + 30n + 13 = O(n^3)$, because for $n \geqslant 13$ we have $10n^2 + 30n + 13 \leqslant n^3$. We have chosen $C = 1$. In this case any positive value of C would also work.
- $10n^2 + 30n + 13 \neq O(n)$, because no matter how C is chosen there exists an n for which $10n^2 + 30n + 13 > Cn$.

We also have [**Homework:** explain why]:

- $10n^2 + 30n + 13 = \Omega(n^2)$.
- $10n^2 + 30n + 13 \neq \Omega(n^3)$.
- $10n^2 + 30n + 13 = \Omega(n)$.

and

- $10n^2 + 30n + 13 = \Theta(n^2)$.
- $10n^2 + 30n + 13 \neq \Theta(n^3)$.
- $10n^2 + 30n + 13 \neq \Theta(n)$.

Asymptotic notation (part 5, dominance relations)

The following functions are ordered according to their growth rate: 1, $\log n$, $\sqrt{n}$, n, $n \log n$, n^2, n^3, 2^n, $n!$.

Consider a processor that can do 10^{10} arithmetic operations per second. (That lies within the capabilities of top end contemporary processors.) The following table presents the time it takes that processor to solve a problem requiring $\log n$, n, $n \log n$, n^2, n^3, 2^n, $n!$, and n^n arithmetic operations (s means seconds, h means hours, d means days, and y means years):

n	$\log n$	n	$n \log n$	n^2	n^3	2^n	$n!$	n^n
10	0.2ns	1.0ns	2.3ns	10ns	100ns	102ns	363μs	1s
20	0.3ns	2.0ns	6.0ns	40ns	800ns	105μs	7.7y	3.3×10^8y
30	0.3ns	3.0ns	10ns	90ns	2.7μs	107ms	8.4×10^{14}y	
40	0.4ns	4.0ns	15ns	160ns	6.4μs	110s		
50	0.4ns	5.0ns	20ns	250ns	12μs	1.3d		
60	0.4ns	6.0ns	25ns	360ns	22μs	3.7y		
10^2	0.5ns	10ns	46ns	1.0μs	100μs	4.0×10^{12}y		
10^3	0.7ns	100ns	691ns	100μs	100ms			
10^4	0.9ns	1.0μs	9.2μs	10ms	100s			
10^5	1.2ns	10μs	115μs	1.0s	1.2d			
10^6	1.4ns	100μs	1.4ms	100s	3.2y			
10^7	1.6ns	1.0ms	16ms	2.8h	3.2×10^3y			
10^8	1.8ns	10ms	184ms	11.6d				
10^9	2.1ns	100ms	2.1s	3.2y				

Asymptotic notation (part 6)

I also must confess a bit of bias against algorithms that are efficient only in an asymptotic sense, algorithms whose superior performance doesn't begin to "kick in" until the size of the problem exceeds the size of the universe. ... I can understand why the contemplation of ultimate limits has intellectual appeal and carries an academic cachet; but in The Art of Computer Programming I tend to give short shrift to any methods that I would never consider using myself in an actual program.

— Donald E. Knuth, The Art of Computer Programming, preface of volume 4A (2011)

An example

We will now determine the computational complexity of the following simple algorithm:

Algorithm: Linear search of unordered data.

Input: a sequence of n distinct numbers $a_1, a_2, \ldots, a_n$, with $n > 0$, and a number b.

Output: the smallest index i such that $a_i = b$, or 0 if no such index can be found.

Steps:

1. [Initialize index.] Set k to 1.
2. [Test.] If $a_k = b$ terminate the algorithm, with k as the output.
3. [Advance.] If $k < n$ then increase k and return to step 2. Otherwise terminate the algorithm with 0 as the output.

Let $f(n)$ denote the number of steps taken by the algorithm, and let $g(n)$ denote the number of average steps. The best case occurs when $b = a_1$. So, $f(n) = \Omega(1)$. The worst case occurs when b is different for all the a_i. In that case steps 2 and 3 will be done n times. Hence, $f(n) = O(n)$.

The average case depends on the probabilities p_i of the events $b = a_i$. Let $p_0 = 1 - \sum_{i=1}^{n} p_i$ be the probability of the remaining event, that b is different for all the a_i. When the algorithm terminates in step 2 for a certain value of i the total number of steps it performs is $1 + 2i + 3(i-1)$. That event has probability p_i. When the algorithm terminates in step 3, with probability p_0, the total number of steps it performs is $1 + 2n + 3(n-1) + 2$. We then have $g(n) = (5n)p_0 + \sum_{i=1}^{n}(5i-2)p_i$. When $p_i = \frac{1}{n}$ we have $p_0 = 0$ and $g(n) = \frac{1}{n}\sum_{i=1}^{n}(5i-2)$. This can be simplified (see next slide) to $g(n) = \frac{5n+1}{2}$, and so in this case $g(n) = O(n)$. In the general case $g(n)$ is as small as possible when $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_n$.

We could have counted only the number of times the body of the loop (steps 2 and 3) is performed. The result, $np_0 + \sum_{i=1}^{n} ip_i$, is similar to what we get from the more detailed analysis (but without the factor of 5).

Useful formulas

The following formulas are useful to analyze algorithms:

- $\displaystyle\sum_{k=1}^{n} 1 = n$
- $\displaystyle\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$
- $\displaystyle\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
- $\displaystyle\sum_{k=1}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2$
- $\displaystyle\sum_{k=1}^{n} \frac{1}{k} = \log n + \underbrace{\gamma}_{\substack{\text{Euler's constant}\\ \approx 0.577216}} + \frac{1}{2n} + O(n^{-2})$
- $\displaystyle\sum_{k=n}^{m} f(k) = \sum_{k=1}^{m} f(k) - \sum_{k=1}^{n-1} f(k)$

A summation of the form $\sum_{k=a}^{b-1} f(k)$, with a and b integers, can be approximated by an integral of the form $\int_a^b f(x)\,dx$. More precisely, we have (Euler-Maclaurin summation formula):

$$\sum_{k=a}^{b-1} f(k) = \int_a^b f(x)\,dx + \sum_{k=1}^{m} \frac{B_k}{k!} f^{(k-1)}(x)\bigg|_a^b + R_m,$$

where B_k are the Bernoulli numbers ($B_0 = 1$, $B_1 = -\frac{1}{2}$, $B_2 = \frac{1}{6}$, $B_3 = 0$, $B_4 = -\frac{1}{30}$, $\ldots$), are where

$$R_m = (-1)^{m+1} \int_a^b \frac{B_m(x - \lfloor x \rfloor)}{m!} f^{(m)}(x)\,dx$$

(here $\lfloor x \rfloor$ is the greatest integer less than or equal to x, so that $x - \lfloor x \rfloor$ is the fractional part of x, and $B_m(x)$ is the m-th order Bernoulli polynomial. [**Exercise**: confirm the first four formulas given above.]

Computational complexity (exercises)

— P.03 —

Summary:

- Paper and pencil exercises (with computer verification)
- Computational complexity of three algorithms

Paper and pencil exercises (part 1a, code)

Give a formula for the value returned by each of the following functions, and give their running time in Big Theta notation. Write a program to confirm your results (you can find these functions in the file `functions.c`).

-
  ```
  int f1(int n)
  {
    int i,r = 0;

    for(i = 1;i <= n;i++)
      r += 1;
    return r;
  }
  ```

-
  ```
  int f2(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= i;j++)
        r += 1;
    return r;
  }
  ```

-
  ```
  int f3(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= n;j++)
        r += 1;
    return r;
  }
  ```

-
  ```
  int f4(int n)
  {
    int i,r = 0;

    for(i = 1;i <= n;i++)
      r += i;
    return r;
  }
  ```

-
  ```
  int f5(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = i;j <= n;j++)
        r += 1;
    return r;
  }
  ```

-
  ```
  int f6(int n)
  {
    int i,j,r = 0;

    for(i = 1;i <= n;i++)
      for(j = 1;j <= i;j++)
        r += j;
    return r;
  }
  ```

Paper and pencil exercises (part 1b, solutions)

Let $r(n)$ be the value returned by the function and let $t(n)$ be the corresponding number of iterations of the body of the inner loop. Then,

- for the `f1` function,

$$t_1(n) = r_1(n) = \sum_{i=1}^{n} 1 = n.$$

In this case we have $t_1(n) = \Theta(n)$.

- for the `f2` function,

$$t_2(n) = r_2(n) = \sum_{i=1}^{n}\Big(\sum_{j=1}^{i} 1\Big) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

In this case we have $t_2(n) = \Theta(n^2)$.

- for the `f3` function,

$$t_3(n) = r_3(n) = \sum_{i=1}^{n}\Big(\sum_{j=1}^{n} 1\Big) = \sum_{i=1}^{n} n = n\sum_{i=1}^{n} 1 = n^2.$$

Since $t_3(n) = r_3(n)$ we have $t_3(n) = \Theta(n^2)$.

- for the `f4` function,

$$r_4(n) = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

and $t_4(n) = r_1(n)$.

- for the `f5` function,

$$t_5(n) = r_5(n) = \sum_{i=1}^{n}\Big(\sum_{j=i}^{n} 1\Big) = \sum_{i=1}^{n}(n-i+1)$$
$$= (n+1)\sum_{i=1}^{n} 1 - \sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

In this case we have $t_5(n) = \Theta(n^2)$.

- for the `f6` function,

$$r_6(n) = \sum_{i=1}^{n}\Big(\sum_{j=1}^{i} j\Big) = \sum_{i=1}^{n} \frac{i(i+1)}{2}$$
$$= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} + \frac{1}{2}\frac{n(n+1)}{2}$$
$$= \frac{n(n+1)}{12}(2n+1+3) = \frac{n(n+1)(n+2)}{6}$$

and $t_6(n) = r_2(n)$. Note that $r_6(n) = \Theta(n^3)$.

Paper and pencil exercises (part 2a, some problems)

Each one of the statements

- S1: $f(n) = O\big(g(n)\big)$,
- S2: $f(n) = \Omega\big(g(n)\big)$,
- S3: $f(n) = \Theta\big(g(n)\big)$,

can be either true of false for each of the following pairs of functions. Determine which ones are true and explain why.

$f(n)$	$g(n)$
$\log n^2$	$\log n + 2$
$\sqrt{n}$	$\log n^2$
$n\sqrt{n}$	n^2
$n \log n$	n
$n \log n$	$10n \log n + n$
$n \log n$	$n^2 + n \log n$

List the functions given below from lowest to highest order. Mark the functions with the same order.

20	$\log n$	$\log \log n$	1.000001^n	2^n	$0.1n \log n$
n	3^n	$\frac{n}{\log n}$	$n \log n + 100$	2^{n+10}	$n + \frac{100}{n}$
$n!$	$n + 10^9$	$n^2 + n\sqrt{n} \log n$	$\log^2 n$	n^2	$\log n + 10 \log \log n$

Paper and pencil exercises (part 2b, solutions)

$f(n)$	$g(n)$	comparison to perform	true statements
$\log n^2$	$\log n + 2$	$\log n$ compared to $\log n$, tie	S1, S2, and S3
$\sqrt{n}$	$\log n^2$	$n^{1/2}$ compared to $\log n$, $f(n)$ wins	S2
$n\sqrt{n}$	n^2	$n^{3/2}$ compared to n^2, $f(n)$ loses	S1
$n \log n$	n	$n \log n$ compared to n, $f(n)$ wins	S2
$n \log n$	$10n \log n + n$	$n \log n$ compared to $n \log n$, tie	S1, S2, and S3
$n \log n$	$n^2 + n \log n$	$n \log n$ compared to n^2, $f(n)$ loses	S1

rank	function(s)
1	20
2	$\log \log n$
3	$\log n$, $\log n + 10 \log \log n$, $\log^2 n$
4	$\frac{n}{\log n}$
5	n, $n + \frac{100}{n}$, $n + 10^9$
6	$0.1n \log n$, $n \log n + 100$
7	n^2, $n^2 + n\sqrt{n} \log n$
8	1.000001^n
9	2^n, 2^{n+10}
10	3^n
11	$n!$

Computational complexity of three algorithms

Given a sequence of n distinct integers $a_1, a_2, \ldots, a_n$, our task is to determine all pairs (a_i, a_j) such that $a_i + a_j$ is equal to a given v. The program `find_pairs.c` solves this problem in three different ways.
Study, compile, and run the program. What can we say about the time and space complexities of each of the three algorithms coded in the program? (In the space complexity do not take into account the space needed to store the algorithm's input.) Which of the three algorithms has a better computational complexity? Which one uses less extra space?

Auxiliary information:

- In the three algorithms (a_i, a_j) is considered to be the same as (a_j, a_i), so only one of the two is printed. It is also assumed that in a pair $i \neq j$.
- The second algorithm requires non-negative integers.
- The `qsort` function sorts an array using a user supplied comparison function. [Study how this is done!] It has an average case complexity of $O(n \log n)$ and a worst case complexity of $O(n^2)$. It can be replaced by another sorting routine that has a worst case complexity of $O(n \log n)$. So, in this problem assume that sorting can be done in $O(n \log n)$.
- The `calloc` function allocates a memory region with a size (number of bytes) that is the product of its two arguments, fills that region with zeros, and returns a pointer to its starting location.
- The `malloc` function allocates a memory region with a size that is given in its only argument and returns a pointer to its starting location. The initial contents of the memory are arbitrary.
- The `free` function deallocates a memory region previously allocated by either the `calloc` or `malloc` functions.

Computational complexity examples — TP.04 —

Summary:

- Classes of problems
- Examples

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **The Algorithm Design Manual**, chapter 2, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, chapters 1 and 3, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Análise da Complexidade de Algoritmos**, chapter 1, António Adrego da Rocha, FCA.

Classes of problems

In computer science problems are subdivided into classes:

- Problems that can be solved in polynomial time, i.e., for which there exits an $O(n^k)$ algorithm, are considered the tractable problems. They are said to belong to the class P (the P stands for **p**olynomial-time).
- There exist also some problems whose solution can be verified in polynomial time. Consider for example the problem of factoring an integer: finding its factors is believed to be a difficult problem, but verifying the factorization can be done using multiplications and primality tests, operations that are known to be in P. (Primality testing was only proved to be in P in 2002!) Problems of this type are said to belong to the class NP (NP stands for **n**ondeterministic **p**olynomial-time). It is not known if P=NP.
- There exists an important subset of NP problems that are equivalent to a certain "prototype" problem (the so-called boolean satisfiability problem). Finding an efficient solution to one such problem automatically provides an efficient solution to the rest of them. These are said to belong to the class NP-complete. Many meaningful problems are known to belong to this class.

Examples (part 1, $O(n)$ algorithms)

Examples of $O(n)$ algorithms:

- sum of the elements of a vector

```
double vector_sum(unsigned int n,double a[n])
{
  unsigned int i;
  double r;

  r = a[0];
  for(i = 1;i < n;i++)
    r += a[i];
  return r;
}
```

- inner product of two vectors

```
double vector_inner_product(unsigned int n,
                            double a[n],
                            double b[n])
{
  unsigned int i;
  double r;

  r = a[0] * b[0];
  for(i = 1;i < n;i++)
    r += a[i] * b[i];
  return r;
}
```

- sum of two vectors:

```
void vector_addition(unsigned int n,
                     double a[n],
                     double b[n],
                     double r[n])
{ // r = a + b
  unsigned int i;

  for(i = 0;i < n;i++)
    r[i] = a[i] + b[i];
}
```

- find the first index `i` for which `a[i]` is equal to `x`

```
unsigned int find_index(unsigned int n,int a[],int x)
{ // returns n when x is not found
  unsigned int i;

  for(i = 0;i < n && a[i] != x;i++)
    ;
  return i;
}
```

- recursive computation of $n!$

```
double factorial(unsigned int n)
{
  return (n < 2) ? 1.0 : n * factorial(n - 1);
}
```

Examples (part 2, $O(n^2)$ algorithms)

Examples of $O(n^2)$ algorithms:

- multiplication of multi-precision integers (one base 10 digit per array entry)

```
void multiplication(unsigned int n,
                    unsigned int a[n],
                    unsigned int b[n],
                    unsigned int r[2 * n])
{ // r = a * b
  unsigned int i,j;

  assert(sizeof(int) >= 4 && n <= 47721858);
  for(i = 0;i < 2 * n;i++)
    r[i] = 0;
  for(i = 0;i < n:i++)
    if(a[i] != 0)
      for(j = 0;j < n;j++)
        r[i + j] += a[i] * b[j];
  for(i = j = 0;i < 2 * n;i++)
  {
    j += r[i];
    r[i] = j % 10;
    j /= 10;
  }
  assert(j == 0);
}
```

[**Homework:** Why 47721858?]

- sum of two matrices

```
void matrix_addition(unsigned int n,
                     double A[n][n],
                     double B[n][n],
                     double R[n][n])
{ // R = A + B
  unsigned int i,j;

  for(i = 0;i < n;i++)
    for(j = 0;j < n;j++)
      R[i][j] = A[i][j] + B[i][j];
}
```

- insertion sort

```
void insertion_sort(unsigned int n,double a[n])
{
  unsigned int i,j;
  double d;

  for(i = 1;i < n;i++)
  {
    d = a[i];
    for(j = i;j > 0 && d < a[j - 1];j--)
      a[j] = a[j - 1];
    a[j] = d;
  }
}
```

[**Homework:** What are the best and worst cases?]

Examples (part 3, improved multiplication)

The multiplication of two integers, A and B, each with with $2n$ bits, can be done as follows.

- Split A into two halves, A_1 and A_0, each with n bits, so that $A = A_1 2^n + A_0$.
- Likewise, split B into two halves, B_1 and B_0, again each with n bits, so that $B = B_1 2^n + B_0$.
- In the standard multiplication method the product of A and B is computed with the formula $(A_1 B_1)2^{2n} + (A_1 B_0 + A_0 B_1)2^n + (A_0 B_0)$. This requires 4 multiplications of numbers with half the number of bits, so using this formula recursively gives rise to an $O(n^2)$ algorithm to compute the product.
- It is possible to eliminate one multiplication by doing more additions and subtractions, by taking advantage of the identity $A_1 B_0 + A_0 B_1 = (A_1 + A_0)(B_1 + B_0) - A_1 B_1 - A_0 B_0$. Its right hand side requires only one extra multiplication (the products $A_1 B_1$ and $A_0 B_0$ can be reused).
- Thus, if $T(n)$ is the time required to multiply two n-bits integers, we have $T(2n) \leqslant 3T(n) + \alpha n$, where the αn term captures the time required to do additions, subtractions, and house-keeping tasks. It follows (we omit a few details here), that $T(n) = O(n^{\log_2 3})$. Note that $\log_2 3 \approx 1.58496$ is considerably smaller than 2, so this simple method is a substantial improvement over the original method. [**Homework:** Compare n^2 with $2n^{\log_2 3}$ for $n = 10^k$, $k = 1, 2, 3, 4, 5, 6$.]
- Using more advanced methods it is possible to multiply two integers much faster than this. The best known methods do the job in $O(n \log n \log \log n)$ steps and use FFTs (Fast Fourier Transforms), or, what is similar but uses only integers, NTTs (Number Theoretical Transforms).

Examples (part 4, $O(n^3)$ algorithms)

Examples of $O(n^3)$ algorithms:

- multiplication of two matrices

```
void matrix_matrix_product(unsigned int n,
                           double A[n][n],
                           double B[n][n],
                           double R[n][n])
{ // R = A * B
  unsigned int i,j,k;

  for(i = 0;i < n;i++)
    for(j = 0;j < n;j++)
    {
      R[i][j] = 0.0;
      for(k = 0;k < n;k++)
        R[i][j] += A[i][k] * B[k][j];
    }
}
```

- matrix inversion (not much different from the next algorithm).

- determinant of a matrix

```
double matrix_determinant(unsigned int n,
                          double A[n][n])
{ // A is modified
  unsigned int i,j,k;
  double r,t;

  r = 1.0;
  for(i = 0;i < n;i++)
  {
    // find the biggest element (the pivot)
    j = i;
    for(k = i + 1;k < n;k++)
      if(fabs(A[k][i]) > fabs(A[j][i]))
        j = k;
    // exchange lines (if necessary)
    if(j != i)
      for(r = -r,k = i;k < n;k++)
      {
        t = A[i][k];
        A[i][k] = A[j][k];
        A[j][k] = t;
      }
    // Gauss-Jordan elimination
    for(r *= A[i][i],j = i + 1;j < n;j++)
      for(k = i + 1;k < n;k++)
        A[j][k] -= (A[j][i] / A[i][i]) * A[i][k];
  }
  return r;
}
```

Examples (part 5, improved matrix multiplication)

The matrix multiplication of a $(2n) \times (2n)$ matrix can be split as follows:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}$$

This yields 8 multiplications and 4 additions of $n \times n$ matrices. Doing this recursively gives rise to a $O(n^3)$ algorithm. Since the computational complexity of a matrix addition is smaller than that of a matrix multiplication, the number of additions is irrelevant in theory (but not in practice).

Strassen found a way to compute

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

that only requires 7 multiplications (and 18 additions). Winograd later reduced the number of additions to 15. The product is given by

$$\begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & W + V + (A_{11} + A_{12} - A_{21} - A_{22})B_{22} \\ W + U + A_{22}(B_{21} + B_{12} - B_{11} - B_{22}) & W + U + V \end{bmatrix}$$

where $U = (A_{21} - A_{11})(B_{12} - B_{22})$, $V = (A_{21} + A_{22})(B_{12} - B_{11})$, and $W = A_{11}B_{11} + (A_{21} + A_{22} - A_{11})(B_{11} + B_{22} - B_{12})$. This gives rise to a $O(n^{\log_2 7}) \approx O(n^{2.807})$ algorithm to multiply matrices. With considerable effort, it is possible to reduce the exponent to a number closer to 2. The current record is an exponent of only 2.376. Note, however, that all known methods with an exponent smaller that 2.7 are so complex, and with horrendous proportionality constants, that they are **useless** in practice!

Examples (part 6, exponential complexity)

The following algorithms have exponential complexity:

- computing Fibonacci numbers (in a dumb way)

```
double F(unsigned int n)
{
  return (n < 2) ? (double)n : F(n - 1) + F(n - 2);
}
```

Work for the practice class: what is the computational complexity of this function? How about the computational complexity of this “improved” way of computing Fibonacci numbers?

```
double Fi(unsigned int n)
{
  static double Ft[10] = { 0,1,1,2,3,5,8,13,21,34 };

  return (n < 10) ? Ft[n] : Fi(n - 1) + Fi(n - 2);
}
```

- print all possible sums of the elements of an array (generate all subsets)

```
void print_all_sums(unsigned int n,double a[n])
{ // 1 <= n <= 30
  unsigned int i,j,mask;
  double s;

  assert(n <= 30);
  mask = 0; // a bit set to 0 means that the
            // corresponding a[i] will not
            // contribute to the sum
  do
  {
    // do sum
    s = 0.0;
    for(i = j = 0;i < n;i++)
      if(((mask >> i) & 1) != 0)
      {
        s += a[i];
        printf("%sa[%u]",(j == 0) ? "" : "+",i);
        j = 1; // next time print a + sign
      }
    printf("%s = %.3f\n",(j == 0) ? "empty" : "",s);
    // next subset (discrete binary counter)
    mask++;
  }
  while(mask < (1 << n));
}
```

Examples (part 7, worst than exponential complexity)

When called with m equal to 0, the following recursive function prints all permutations of the integers a_0, a_1, $\ldots$, a_{n-1}, assumed to be distinct and to be stored in the array a[]:

```
void print_all_permutations(unsigned int n,unsigned int m,int a[])  // int a[] is a synonym of int *a
{
  unsigned int i;

  if(m < n - 1)
  { // not yet at the end
    for(i = m;i < n;i++)
    {
#define swap(i,j)  do { int t = a[i]; a[i] = a[j]; a[j] = t; } while(0)
      swap(i,m);                        // exchange a[i] with a[m]
      print_all_permutations(n,m + 1,a); // recurse
      swap(i,m);                        // undo the exchange of a[i] with a[m]
#undef swap
    }
  }
  else
  { // visit permutation
    for(i = 0;i < n;i++)
      printf("%s%d",(i == 0) ? "" : " ",a[i]);
    printf("\n");
  }
}
```

Note that although inside the function the contents of the a[] array are reordered, when it returns they fall back to their original values. The computational complexity of this function is $O(n \times n!)$, because the block that prints each permutation is visited $n!$ times.

Examples (part 8, algorithms with “small” computational complexity)

We conclude this tour of examples with two algorithms that have small computational complexity:

- computation of x^y using a mathematical formula (y is a real number)

```
double power_dd(double x,double y)
{
  return exp(y * log(x));
}
```

  This is an $O(1)$ algorithm. It works only when $x > 0$.

- computation of x^n using recursion (n is an integer)

```
double power_di(double x,int n)
{
  if(n < 0)  return power_di(1.0 / x,-n);
  if(n == 0) return 1.0;
  double t = power_di(x * x,n / 2); // the integer division discards a fractional part
  return (n % 2 == 0) ? t : x * t;  // take care of the discarded fractional part
}
```

  This is an $O(\log n)$ algorithm. With the exception of $x = 0$ and $n < 0$, it works for any x. (By convention $0^0 = 1$; 0 raised to a negative power is illegal.)

- computation of $a^b \bmod c$ (u32 is `unsigned int` and u64 is `unsigned long long`)

```
u32 power_mod(u32 a,u32 b,u32 c)
{
# define mult_mod(x,y,m)  (u32)(((u64)(x) * (u64)(y)) % (u64)(m))  // (x*y) mod m
  if(b == 0) return 1u;
  u32 t = power_mod(mult_mod(a,a,c),b / 2,c);
  return (b % 2 == 0) ? t : mult_mod(a,t,c);
# undef mult_mod
}
```

Computational complexity examples (exercises)

— P.04 —

Summary:

- Formal and empirical computational complexity of several algorithms
- The subset sum problem (**theme of the first written report**)

Formal and empirical computational complexity of several algorithms

Extract the file `examples.c` from the archive `P04.tgz`. This file contains the code of all functions described in the `TP.04` lecture. Redoing the formal analysis done in the TP lecture, determine the computational complexity of each one of the functions. Confirm your results by adding code to each of the functions to count (and print at the end) the number of times that the body of the innermost loop is executed.

Note that in C it is possible to write code like this (in C++ that is not possible):

```
for(n = 1;n <= 10;n++)
{
  double A[n][n]; // inside a function, non static arrays do NOT need to have a size defined at compile time!
  int i,j;

  for(i = 0;i < n;i++)
    for(j = 0;j < n;j++)
      A[i][j] = (double)rand() / (double)RAND_MAX; // one way to get uniformly distributed pseudo-random numbers
  //
  // put more stuff here, such as a call to an O(n^2) or an O(n^3) function
  //
}
```

For each interesting case (say, one for each computational complexity), make a log-log graph of the number of times the inner loop was executed versus the problem size.

The subset sum problem (part 1)

One of the earliest proposals for a public key cryptosystem was based on the subset sum problem. In this system anyone wishing to receive encrypted messages publishes a list of integers a_i, for $i = 1, 2, \ldots, n$. To send a message with bits b_i, for $i = 1, \ldots, n$, to this person the sender only has to send the number $M = \sum_{i=1}^{n} b_i a_i$, i.e., it sends the sum of a subset of the published numbers. Anyone wishing to decode the message has to find the particular subset that was used, a problem which in general is NP-hard. The person that published the list, however, can decode the message easily, using extra information she/he alone has (for details about how this is done, consult the wikipedia page about the Merkle-Hellman knapsack cryptosystem). The extra hidden structure imposed on the numbers a_i was ultimately what made this public key cryptosystem insecure.

The file `subset_sum_problems.h` contains sets $\{a_{ni}\}_{i=1}^{n}$ for $n = 10, 11, \ldots, 56$ and 100 subset sums for each n. For a given n the subset sums all use the same list. The main goal of the first practical work will be to solve as many problems as possible, as fast as possible. The code of the function `print_all_sums()` presented in this slide can be your starting point. With simple modifications of this function it is possible to solve easily the problems for small values of n. The largest values of n require some research into more advanced computational strategies. For this particular problem the branch-and-bound strategy yields moderate improvements over the dumb brute force strategy of `print_all_sums()` and the meet-in-the-middle strategy yields acceptable results for the largest n (at the expense of using a large amount of memory).

The subset sum problem (part 2)

The first written report (about the subset sum problem) must have

- a title page (front page) with the name of the course, the name of the report, date, the names of the students, and an estimate of the percentage each student contributed to the project (the sum of percentages should be 100%)
- a short introduction describing the problem; the source of material adapted from the internet must be cited
- a small description of the method or methods used to find the solutions
- a description of the solutions found; this should include a graph of the execution time of the program as a function of the problem size (there are 100 problems for each size, so it is also possible to compute means and standard deviations).
- comments or attempts at explanations of the execution times
- an appendix with all the code (use a small font)

For top grades your report should also include a description of the branch-and-bound and meet-in-the-middle strategies to solve this problem (or of any strategy invented by yourselves).

Elementary data structures (part 1)
— TP.05 —

Summary:

- Data containers
- Arrays (and circular buffers)
- Linked lists (singly- and doubly-linked)
- Stacks
- Queues
- Deques

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Data containers

In most programs it is necessary to store and manipulate information. This information may be stored and manipulated directly by the programmer, or it may be encapsulated in a so-called data container. A data container is a data structure that specifies how the information is organized and stored, together with a standardized interface to access and modify the information (thus, a class).

There are many types of data containers, differing on the services they provide and on the computational complexity of these services (the computational complexity depends on how the information is internally organized in the data container). The choice of data container should be done according to the answers to the following questions:

- Does the insertion and retrieval of information obey some rules? (Say, do we always remove the newest, oldest, largest, or smallest item of information?)
- Do we need efficient random access to the information?
- Do we need efficient sequential access to the information?
- Do we need to efficiently insert information in a random location?
- Do we need to efficiently delete information from a random location?
- Do we need to efficiently search for information?
- Is the information single valued, say, an integer, or is it multi-valued, say, a (key,value) pair?

Arrays (part 1, computational complexity)

An array is one data structure that can be used to implement a data container. Assuming that the items of information are to be stored consecutively in memory (in array elements) it follows that

- it is necessary to specify the size of the array before it is used; if later on it turns out that that size is too small, it will be necessary to resize the array (that is an $O(n)$ operation, but, if done rarely, its amortized cost is low)
- given a position (an index) random access to information is fast: $O(1)$
- sequential access is also fast
- inserting information at one end of the used part of the array (assuming it is not full) is fast: $O(1)$
- inserting information at an arbitrary location, opening space for it, is slow: $O(n)$
- replacing information at an arbitrary location is fast: $O(1)$
- deleting information at one end is fast: $O(1)$
- deleting information at an arbitrary location, closing the space it would otherwise leave behind, is slow: $O(n)$; without closing the space, it is fast: $O(1)$
- if the information is stored in the array in random order, then searching for information is slow: $O(n)$
- if the information is stored in the array in sorted order, then searching for information is fast: $O(\log n)$

Arrays (part 2, implementation of a circular buffer)

A circular buffer is an array in which index arithmetic is done modulo the size of the array (for an array with n elements, index n is the same as index 0). Besides supporting the usual array operations, by keeping track of the indices of the first and last data items stored in it, a circular buffer also supports $O(1)$ insertion and deletion of data at either end. Thus a circular buffer is a very efficient way of implementing a queue and a deque (see next slides).

Circular buffers are usually used in device drivers to implement efficiently a queue with a given maximum size, because it does not require any dynamic memory operations (allocation and deallocation of memory).

The following code illustrates one possible way to increment and decrement an index in a circular buffer (of floating point numbers):

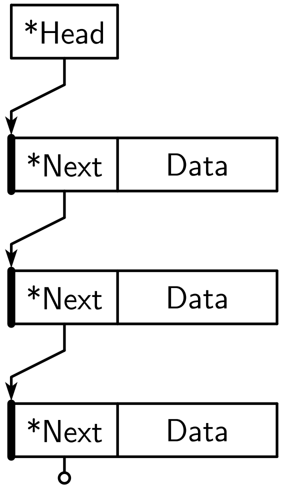
```
class circular_buffer
{
  private:
    int max_size; // the maximum size of the circular buffer
    double *data; // array allocated in the constructor
  public:
    circular_buffer(int max_size = 100)  { this->max_size = max_size; data = new double[max_size]; }
    ~circular_buffer(void)               { delete[] data;                                        }
  private:
    int increment_index(int i)  { return (i + 1 <  max_size) ? i + 1 : 0;            }
    int decrement_index(int i)  { return (i - 1 >= 0       ) ? i - 1 : max_size - 1; }
    //
    // data[i] accesses the number stored in position i
    // data[increment_index(i)], accesses the number stored in the position after position i
    // data[decrement_index(i)], accesses the number stored in the position before position i
    //
    // ... (rest of the code for the circular_buffer class)
    //
}
```

Linked lists (part 1, overview)

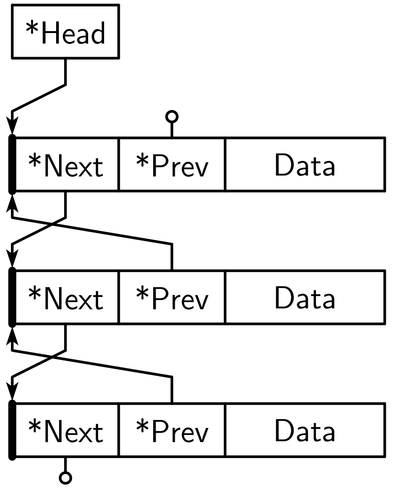
A linked list is a dynamic data structure, because it can grow as much as needed. In order to be able to do this, each node of information contains

- the information itself
- in a singly-linked list, a pointer to the next node of information
- in a doubly-linked list, a pointer to the next node and another to the previous node of information

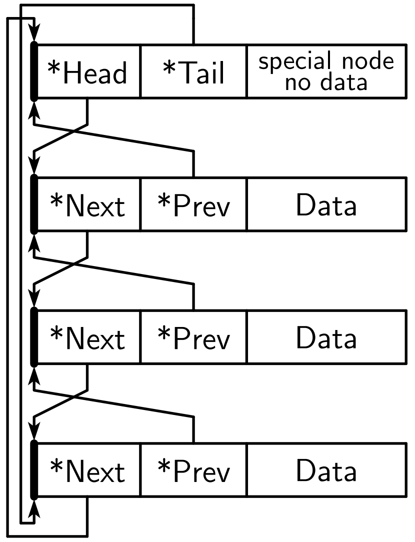
In a singly linked-list, the nodes of information are linked as follows (a small circle represents a `nullptr` pointer, and a ∗ before a field name means that it is a pointer):



In a doubly linked-list, the nodes of information are linked as follows (note that when inserting or removing information it may be necessary to deal with `nullptr` pointers):



Using an extra special node to hold the head, in the `Next` field, and the tail, in the `Prev` field, of a doubly linked-list makes its implementation far simpler (no `nullptr` pointers):



Linked lists (part 2, computational complexity)

Because of the way information is organized in a linked list,

- it is necessary to keep track of the first node (the head) of the list
- it is possible to keep track of the last node (the tail) of the list
- given a position (an index), random access to information is slow: $O(n)$
- forward sequential access (from head to tail) is fast; backward sequential access (from tail to head) is slow for singly-linked lists and fast for doubly-linked lists if the tail is known
- inserting information at the head of the list is fast: $O(1)$
- if the tail of the list is known, inserting information at the end is fast: $O(1)$
- if the tail of the list is not known, inserting information at the end is slow: $O(n)$
- inserting information after a node is fast: $O(1)$
- deleting information at the head of the list is fast: $O(1)$
- on a singly-linked list, deleting a node of information is slow: $O(n)$
- on a doubly-linked list, deleting a node of information is fast: $O(1)$
- searching for information is slow, even if the data is stored in sorted order: $O(n)$

Linked lists (part 3, operations)

The following operations are usually supported in a linked list implementation:

- creation of the linked list
- destruction of the linked list
- insertion of a new node of information (before the head of the list, after the tail of the list, or after a given node of information)
- deletion of a node of information (of the head of the list, of the tail of the list, or of a given node of information)
- given a node of information, determine its next node of information
- given a node of information, determine its previous node of information

Given the class

```
class list_node
{
  private:
    list_node *next;
    // ... (rest of the code for the list_node class)
}
```

the following code finds the tail of a singly-linked list:

```
list_node *tail = head;
if(tail != nullptr)
  while(tail->next != nullptr)
    tail = tail->next;
```

Stacks

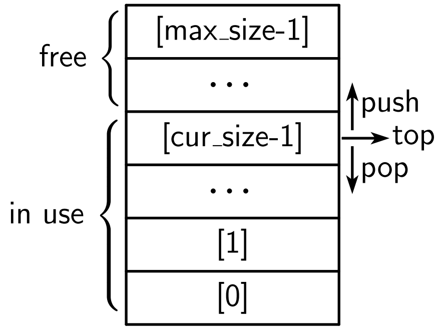
A stack, associated with a usage policy of First In Last Out (FILO), or, what is the same, Last In First Out (LIFO), is a data container that supports the following operations:

- creation of the stack
- destruction of the stack
- add a new element to the top of the stack, called **push**
- remove the element at the top of the stack, called **pop**
- take a look at the top element of the stack, called **top**
- determine the current size of the stack

It is possible to implement a stack using

- an array (in this case the stack has a maximum size, specified when the stack is created),
- a linked list (keeping the top of the stack at the head of the list), or
- a deque (see next slides).

The simplest implementation uses an array and two integers: `max_size`, which is the maximum size of the stack, and `cur_size`, which is the current size of the stack. In this case, the index of the top of the stack is `cur_size-1`, the stack is empty when `cur_size==0` and it is full when `cur_size==max_size`. The following figure illustrates how the information is organized when a stack is implemented using an array.



[**Homework:** implement a stack using a linked list.]

Queues

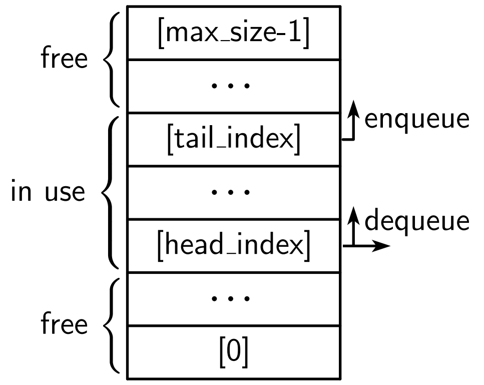
A queue, associated with a usage policy of First In First Out (FIFO), or, what is the same, Last In Last Out (LILO), is a data container that supports the following operations:

- creation of the queue
- destruction of the queue
- add a new element to the back (tail) of the queue, called **enqueue**
- remove the element at the front (head) of the queue, called **dequeue**
- determine the current size of the queue

It is possible to implement a queue using

- a circular buffer (in this case the queue has a maximum size, specified when the queue is created),
- a linked list (preferably one that keeps track of the tail, to make the **enqueue** operation efficient), or
- a deque (see next slides).

The simplest implementation uses a circular buffer and four integers: `max_size`, which is the maximum size of the queue, `cur_size`, which is the current size of the queue, `head_index`, which is the index of the head of the queue, and `tail_index`, which is the index of the tail of the queue.



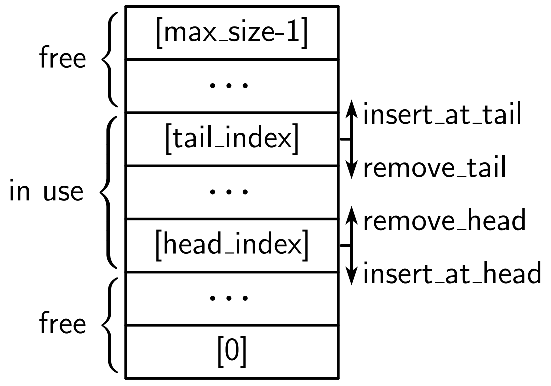
In an empty queue `cur_size==0` and the tail index has to be one more than the head index (modulo `max_size`).

Deques

A deque is a double-ended queue. It is similar to a queue, but it is possible to insert and remove elements at both ends of the queue. It supports the following operations:

- creation of the deque
- destruction of the deque
- insert a new element at the front (head) or at the back (tail) of the deque
- remove the element at the front (head) or at back (tail) the of the deque
- determine the current size of the deque

It can be implemented using either a circular buffer or a doubly-linked list (given that it may be necessary to move in either direction, using a singly-linked list to implement a deque is inefficient).



An insertion at either end forces the size of the deque to increase. That determines the direction of movement of the head or of the tail index.

Elementary data structures (part 1, exercises)

— P.05 —

Summary:

- Stacks
- Singly-linked lists
- Queues
- Deques
- Doubly-linked lists

Stacks

Extract the files aStack.h and aStack_demo.cpp from the archive P05.tgz. Study the generic implementation of a stack (file aStack.h). The purpose of the program aStack_demo.cpp is to verify if the parentheses of each of its text arguments are balanced. When called as follows (warning: copying and pasting may not work properly on the following line; if it does not work the accute accent is the culprit)

```
  ./aStack_demo 'abc' 'a(b)' 'a(b' 'a)b' 'a(b(c)(d((ef)g))h)i'
```

it should produce the output

```
abc
  good
a(b)
  '(' at position 1 and matching ')' at position 3
  good
a(b
  unmatched '(' at position 1
  bad
a)b
  unmatched ')' at position 1
  bad
a(b(c)(d((ef)g))h)i
  '(' at position 3 and matching ')' at position 5
  '(' at position 9 and matching ')' at position 12
  '(' at position 8 and matching ')' at position 14
  '(' at position 6 and matching ')' at position 15
  '(' at position 1 and matching ')' at position 17
  good
```

The code in aStack_demo.cpp is incomplete. Complete it using a stack.

Modify the aStack.h class so that the stack can grow as much as needed. (Hint: write a private member function that resizes the stack, and start with a stack with a maximum size of, say, 100.)

Singly-linked Lists

Extract the files `sList.h` and `sList_test.cpp` from the archive `P05.tgz`. The file `sList.h` implements a generic singly-linked list. Study it. Study also the file `sList_test.cpp`, that tests the correctness of the implementation in `sList.h`.

Queues

Extract the files `lQueue.h` and `lQueue_demo.cpp` for the archive `P05.tgz`. The file `lQueue.h` contains a skeleton of an implementation of a generic queue based on a singly-linked list. Complete the implementation and write code to test it.

Deque

Implement a generic deque (double-ended queue) using an array. On a deque, insertion and deletion can occur at both ends of the queue.

Doubly-linked lists

Work to be done at home: using the code in `sList.h` as starting point, implement a doubly-linked list. Hints:

- the `move()` member function can be improved, but that is not strictly necessary,
- the various `insert` and `remove` member functions have to be modified.
- the computational complexity of some of these functions may change!

Elementary data structures (part 2) — TP.06 —

Summary:

- Heaps
- Priority queues
- Binary trees
- Tries
- Hash tables

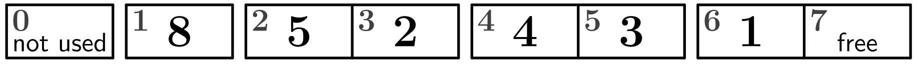
Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

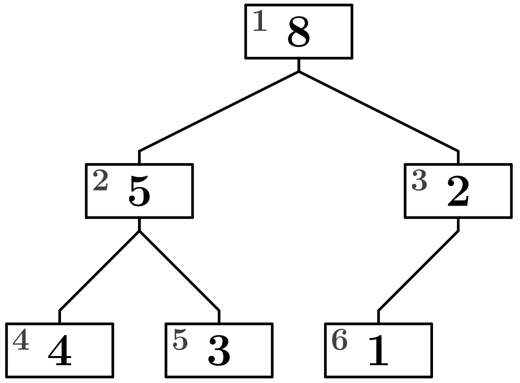
Heaps (part 1, properties)

A heap is an array with internal structure. The internal structure of the array takes the form of an order relation (the heap property) that has to be maintained between the data stored in certain index positions. To be more precise, in a **binary max-heap** with n data items, stored in indices 1, 2, $\ldots$, n, the information stored in index i, with $2 \leqslant i \leqslant n$, cannot be larger than the information stored in index $\lfloor i/2 \rfloor$; $\lfloor x \rfloor$ is the largest integer that is not larger than x (i.e., the `floor` function). For example, the data stored in the following array (left hand side) satisfies the binary max-heap property

index	value
0	-
1	8
2	5
3	2
4	4
5	3
6	1



Implicit binary tree organization



because $8 \geqslant 5$ (index 1 versus index 2), $8 \geqslant 2$ (index 1 versus index 3), $5 \geqslant 4$ (index 2 versus index 4), $5 \geqslant 3$ (index 2 versus index 5), and $2 \geqslant 1$ (index 3 versus index 6). The max-heap property can be easily checked using, for example, the following code

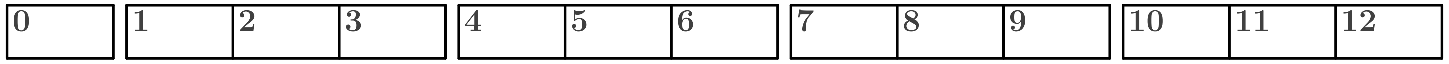
```
for(int i = 2;i <= n;i++)
    assert(heap[i / 2] >= heap[i]);
```

We may also have a binary min-heap, and even multi-way heaps. The starting index is usually either 1 or 0.

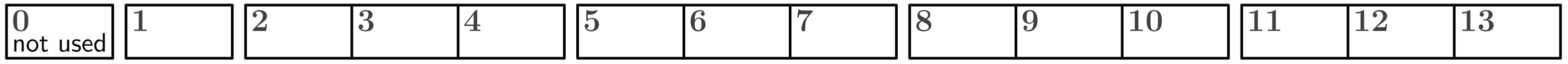
Heaps (part 2, m-way heaps)

For each index of a heap we define its parent index, and its children indices. In a max-heap the data stored in a given position cannot be larger that the data stored in the parent position, and it cannot be smaller that the data stored in each one of its children positions. This implicitly defines a tree organization for the data, as shown on the right hand side of the example given in the previous slide. The first index of the array used by the heap is the root index, as it corresponds to the root of the implicit tree.

In a m-way heap with a root index of 0, the parent of index $i > 0$ is $\lfloor \frac{i-1}{m} \rfloor$ and its children are $mi+1, \ldots, mi+m$. For $m = 3$ the array is implicitly subdivided in the following way:



In a m-way heap with a root index of 1, the parent of index $i > 1$ is $\lfloor \frac{i+m-2}{m} \rfloor$ and its children are $mi - m + 2, \ldots, mi + 1$. Note that for $m = 2$ the formulas are particularly simple. For $m = 3$ the array is implicitly subdivided in the following way:



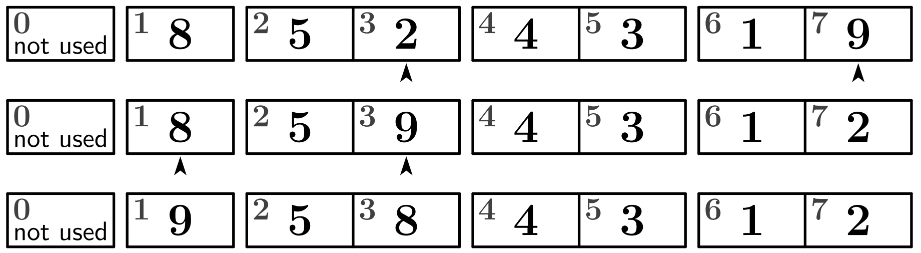
The path to the root of a node with index i is the sequence of indices i, $\mathbf{parent}(i)$, $\mathbf{parent}\big(\mathbf{parent}(i)\big)$, and so on, that stops at the root index. When $m = 2$ and the root index is 1, the path to the root of i is i, $\lfloor \frac{i}{2} \rfloor$, $\lfloor \frac{i}{2^2} \rfloor$, $\lfloor \frac{i}{2^3} \rfloor$, $\ldots$, $\lfloor \frac{i}{2^k} \rfloor$, where $k = \lfloor \log_2 i \rfloor$; $\log_2 x$ is the base-2 logarithm of x. In particular, for a binary heap of size n, the longest path to the root has length $1 + \lfloor \log_2 n \rfloor$, which is $O(\log n)$.

Heaps (part 3, insertion)

A max-heap supports at least the following operations (a min-heap is similar, with the word "largest" replaced by the word "smallest"):

- creation and destruction of the heap
- inspection of the largest data item (the root of a max-heap holds the largest data value)
- insertion of a data item
- removal of a data item

To insert a data item `v` on a heap with size `n` the first step is to place it at the end of the heap. The heap property is then enforced on the path to the root of the new node, by exchanging the data of a node with that of its parent whenever the heap property is not satisfied (there is no need to change data elsewhere). Thus, insertion is an $O(\log n)$ operation. The following example illustrate what happens when 9 is inserted on the heap $8, 5, 2, 4, 3, 1$:



9 is larger that 2 (its parent): exchange

9 is larger that 8 (its parent): exchange

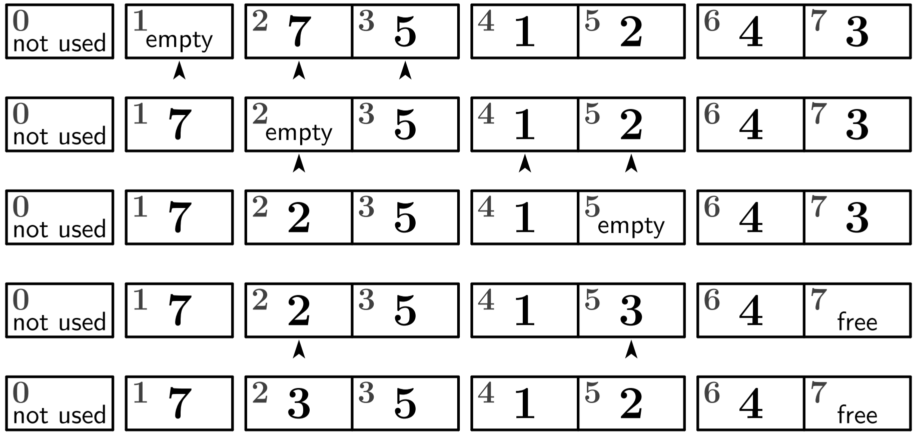
9 is at the root: terminate

In C or C++, all this can be done as follows ($m = 2$ and the root index is 1):

```
for(i = ++n;i > 1 && heap[i / 2] < v;i /= 2)
  heap[i] = heap[i / 2];
heap[i] = v;
```

Heaps (part 4, removal)

To remove a data item, we replace it by the largest of its children, and then we do the same to fill the empty slot vacated by each child that was moved towards the root, until that cannot be done any more. This procedure leaves an empty slot in the heap. If it is not the last slot, we move the last slot to the empty slot and then enforce the heap property on the path to the root starting at that slot. In the worst case about $2\log_2 n$ operations need to be done — $O(\log n)$ — to remove a data item. The following example illustrates what happens when the root (9) is removed from the heap $9, 7, 5, 1, 2, 4, 3$:



0 not used | 1 empty | 2 7 | 3 5 | 4 1 | 5 2 | 6 4 | 7 3
0 not used | 1 7 | 2 empty | 3 5 | 4 1 | 5 2 | 6 4 | 7 3
0 not used | 1 7 | 2 2 | 3 5 | 4 1 | 5 empty | 6 4 | 7 3
0 not used | 1 7 | 2 2 | 3 5 | 4 1 | 5 3 | 6 4 | 7 free
0 not used | 1 7 | 2 3 | 3 5 | 4 1 | 5 2 | 6 4 | 7 free

In C or C++, removal of the data at position pos can be done as follows ($m = 2$ and the root index is 1):

```
for(i = pos;2 * i <= n;heap[i] = heap[j],i = j)
  j = (2 * i + 1 <= n && heap[2 * i + 1] > heap[2 * i]) ? 2 * i + 1 : 2 * i; // select largest child
for(;i > 1 && heap[i / 2] < heap[n];i /= 2)
  heap[i] = heap[i / 2];
heap[i] = heap[n--];
```

[**Homework:** what happens in the second `for` loop when i is equal to n?]

Priority queues

A priority queue is a data container that supports the following operations:

- creation and destruction of the priority queue
- inspection of the largest data item (**peek**)
- insertion of a data item (**enqueue**)
- removal of the largest data item (**dequeue**)

It can be implemented easily using a max-heap. Conceptually, a priority queue is similar to an ordinary queue, its only difference being that instead of removing the **oldest** element it is the **largest** that gets removed. (In terms of implementation, a priority queue and an ordinary queue are quite different.)

Instead of inspecting and removing the largest data item, a priority queue can allow the inspection and removal of its smallest data item. This variant of the priority queue is, obviously, implemented using a min-heap.

Binary trees (part 1, overview)

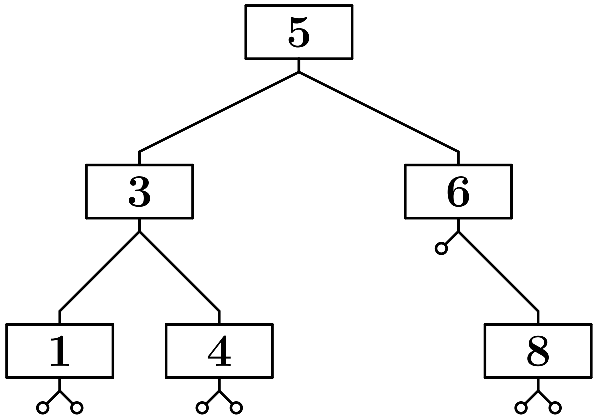
A binary tree is a dynamic data structure composed of nodes. Each node of information contains

- the information itself (the data item)
- a pointer to the node on the "left" side (the left branch); in an ordered binary tree the data items stored on this side are all of them smaller that the data item stored on the node
- a pointer to the node on the "right" side (the right branch); in an ordered binary tree the data items stored on this side are all of them larger that the data item stored on the node
- optionally, a pointer to the parent node

A null pointer (`NULL` in C and `nullptr` in C++) indicates the nonexistence of a node. The **root** node is the node where the tree begins. It is the only node without a parent node. A **leaf** node is a node whose left and right node pointers are null.

The **height** of a tree is the number of levels it has. Each time a left or a right pointer is followed the level increases by 1. It is usual to consider that the root is at level 0. We may also talk about the height of the left or right branches of a node: that it the height of the tree that has as root the left or right node of the node. It is not necessary for a tree to have its leaves all at the same level (but that is usually desired).

Visually, a tree is usually depicted upside-down, i.e., with its root on top:



Binary trees (part 2, node contents)

In these slides each node of the **ordered binary tree** will be implemented (in C) as follows:

```
typedef struct tree_node
{
  struct tree_node *left;   // pointer to the left branch (a sub-tree)
  struct tree_node *right;  // pointer to the right branch (a sub-tree)
  struct tree_node *parent; // optional
  int data;                 // the data item (we use an int here, but it can be anything)
}
tree_node;
```

Each node may also keep information about

- the height of the subtree having that node as root,
- the difference of the heights of its left and right branches (the balance)
- the level of the node, or
- other useful information that may be need by a particular program.

In the following slides we present C code snippets of functions that do useful things to a tree. **Study them carefully.** Most of them take the form of a recursive function, because trees are recursive structures: the left and right branches of a node are also trees!

In some of these functions we pass a pointer to the location in memory where the pointer to the root of the tree is stored (a pointer to a pointer). Things are done in this way because it may be necessary to change the root of the tree!

Binary trees (part 3, insertion)

The following non-recursive function creates a new node and inserts it in the tree at the appropriate location:

```
tree_node *new_tree_node(int data,tree_node *parent); // sets left and right to NULL

void insert_non_recursive(tree_node **link,int data)
{
  tree_node *parent = NULL;
  while(*link != NULL)
  {
    parent = *link;
    link = (data <= (*link)->data) ? &((*link)->left) : &((*link)->right); // select branch
  }
  *link = new_tree_node(data,parent);
}
```

This can also be done recursively as follows:

```
void insert_recursive(tree_node **link,tree_node *parent,int data)
{
  if(*link == NULL)
    *link = new_tree_node(data,parent);
  else if(data <= (*link)->data)
    insert_recursive(&((*link)->left),*link,data);
  else
    insert_recursive(&((*link)->right),*link,data);
}
```

These functions are used as follows:

```
tree_node *root = NULL;

insert_nonrecursive(&root,4);
insert_recursive(&root,NULL,7);
```

What is the height of an initially empty tree after insertion of $1, 2, \ldots, 31$? What about the heigth of an initially empty tree after insertion of $f(1), f(2), \ldots, f(31)$, where the function $f(n)$ reverses the order of the least significant 5 bits of its argument?

Binary trees (part 4, search in an ordered and in an unordered binary tree)

For an **ordered** binary tree, the following non-recursive and recursive functions return one node of the tree for which its data field is equal to a given value (or NULL if there is no such node):

```
tree_node *search_non_recursive(tree_node *link,int data)
{
  while(link != NULL && data != link->data)
    link = (data < link->data) ? link->left : link->right;
  return link;
}

tree_node *search_recursive(tree_node *link,int data)
{
  if(link == NULL || link->data == data)
    return link;
  if(data < link->data)
    return search_recursive(link->left,data);
  else
    return search_recursive(link->right,data);
}
```

These functions are used as follows:

```
tree_node *root,*n;
int data;

n = search_non_recursive(root,data);
n = search_recursive(root,data);
```

For an **unordered** binary tree, the following recursive function returns one node of the tree for which its data field is equal to a given value (or NULL if there is no such node):

```
tree_node *search_recursive(tree_node *link,int data)
{
  node *n;

  if(link == NULL || link->data == data)
    return link;
  // try the left branch
  if((n = search_recursive(link->left,data)) != NULL)
    return n;
  // not found in the left branch, try the right branch
  return search_recursive(link->right,data);
}
```

This function is used as follows:

```
tree_node *root,*n;
int data;

n = search_recursive(root,data);
```

Note that for an unordered binary tree it may be necessary to search the entire tree, which is an order $O(n)$ operation, while in an ordered binary tree it is necessary to examine at most $h+1$ nodes, where h is the maximum height of any node of the tree.

Binary trees (part 5, traversal)

The following non-recursive and recursive functions traverse the entire tree in different orders:

```c
void visit(tree_node *n)
{
  printf("%d\n",n->data);
}

void traverse_breadth_first(tree_node *link)
{
  queue *q = new_queue();

  enqueue(q,link);
  while(is_empty(q) == 0)
  {
    link = dequeue(q);
    if(link != NULL)
    {
      visit(link);
      enqueue(q,link->left);
      enqueue(q,link->right);
  }
  free_queue(q);
}
```

```c
void traverse_depth_first_recursive(tree_node *link)
{
  if(link != NULL)
  {
    visit(link);
    traverse_depth_first_recursive(link->left);
    traverse_depth_first_recursive(link->right);
  }
}

void traverse_in_order_recursive(tree_node *link)
{
  if(link != NULL)
  {
    traverse_in_order_recursive(link->left);
    visit(link);
    traverse_in_order_recursive(link->right);
  }
}
```

They are used as follows:

```c
tree_node *root;

traverse_breadth_first(root);
traverse_depth_first_recursive(root);
traverse_in_order_recursive(root);
```

[**Homework:** do the `traverse_depth_first` function in a non-recursive way.]

Binary trees (part 6a, misc)

Does the purpose of each of the following functions match its name?

```
// use count_nodes(root) to count the nodes of an entire tree

int count_nodes(tree_node *link)
{
  return (link == NULL) ? 0 : count_nodes(link->left) + 1 + count_nodes(link->right);
}


// use count_leaves(root) to count the leaves (terminal nodes) of an entire tree

int count_leaves(tree_node *link)
{
  if(link == NULL)
    return 0;
  if(link->left == NULL && link->right == NULL)
    return 1;
  return count_leaves(link->left) + count_leaves(link->right);
}


// use check_node(root,NULL,INT_MIN,INT_MAX) to check an entire ordered binary tree

void check_node(tree_node *link,tree_node *parent,int min_bound,int max_bound)
{
  if(link != NULL)
  {
    assert(min_bound <= link->data && link->data <= max_bound && link->parent == parent);
    check_node(link->left,link,min_bound,link->data);
    check_node(link->right,link,link->data,max_bound);
  }
}
```

Binary trees (part 6b, misc)

Does the purpose of each of the following functions match its name?

```
// use set_level_nodes(root,0) to set the level of each node of an entire tree

void set_level(tree_node *link,int level)
{
  if(link != NULL)
  {
    link->level = level;
    set_level(link->left,level + 1);
    set_level(link->right,level + 1);
  }
}


// use set_height(root) to set the height of each node of an entire tree

int set_height(tree_node *link)
{
  int left_height,right_height;

  if(link == NULL)
    return 0;
  left_height = set_height(link->left);
  right_height = set_height(link->right);
  link->height = (left_height >= right_height) ? 1 + left_height : 1 + right_height;
  return link->height;
}
```

Binary trees (part 7a, balancing)

A binary tree is said to be **perfect** if all its leaves are at the same level. A perfect binary tree with height h has $2^h - 1$ nodes: 2^{h-1} leaves and $2^{h-1} - 1$ internal nodes. Abusing the concept of perfect binary tree, we will say that a binary tree is also perfect if some of the leaves at the last level of the tree are missing (but all other levels are full). Such a tree has a height as small as possible. That is desirable because many operations that modify a binary tree do a number of elementary operations that is proportional to the height of the tree. A perfect binary tree with n nodes has a height equal to $\lceil \log_2(n+1) \rceil$, which is $O(\log n)$.

A binary tree is said to be **balanced** if for each of its nodes the height of its left branch does not differ by more than 1 from the height of its right branch. A balanced binary tree of height h has at least G_h nodes, where $G_0 = 0$, $G_1 = 1$ and, for $n > 1$, $G_n = G_{n-1} + 1 + G_{n-2}$, and it has at most $2^h - 1$ nodes. It can be verified that $G_n = F_{n+2} - 1$, where F_n is the n-th Fibonacci number. It follows that the height of a balanced binary tree is $\Theta(\log n)$. It is possible, using simple $O(\log n)$ operations, to keep a binary tree balanced after the insertion or removal of a node. Searching is also an $O(\log n)$ operation in a balanced binary tree.

Binary trees (part 7b, how to keep an ordered binary tree balanced)

Explain here how to balance a binary tree (to be done, maybe, in the 2017/2018 school year).

Tries

A trie is essentially an m-way tree (each node of the tree has m children.) An index is usually used to select the appropriate children. For example, storing and searching for a telephone number can be done very efficiently using a trie (here m will be 10). The most significant digit of the telephone number selects the child of the root of the tree that must be followed, the second most significant digit selects the child of that node, and so on. Only the nodes that are needed are actually allocated.

Hash tables (part 1, overview)

A hash table is a data container that supports (at least) the following operations:

- creation and destruction of the hash table
- insertion of a data item composed of two parts: the key and its corresponding value
- search for a data item with a given key
- removal of a data item given its key

In an array, information is accessed given its index. In a hash table, information is accessed given its key. In a properly dimensioned hash table, the insertion, removal and search operations have very good expected computational complexity: $O(1)$. This is better that the computational complexity of the same operations in a balanced binary tree, which is $O(\log n)$, where n is the number of data items stored in the data container.

A hash table is usually the data container used to implement an associative array, a symbol table, or a dictionary. (Other data containers may also be used for this, but the hash table is usually more efficient.) The key may not be an integer; it may be, for example, a string.

In the insert operation, if a data item with the same key already exists in the hash table, the insert operation should either fail or it should replace the corresponding value (and no new data item is created). The programmer has to decide which is best for a given application.

An hash table is usually implemented using an array. It may be an array of data items (keys and respective values), if **open addressing** (explained later) is used, or it may be an array of pointers to the heads of (doubly-)linked lists of data items, if **chaining** is used. Instead of a pointer to the head of a linked list, it is also possible to use a pointer to the root of a binary tree, or even a pointer to another hash table.

Hash tables (part 2a, hash functions)

Let s be the size of the array used to implement the hash table. An hash function h maps each possible key k to an integer i in the range $0, \ldots, s-1$. This integer will then be used to access the array. This conversion is necessary because the key itself may not be an integer, and even if it is an integer, its value may be too small or too large.

If the number of keys, n, is larger than the size of the array, s, then it is inevitable that two (or more) keys map, via the hash function, to the same index. Even when n is smaller than s, it is possible, if the hash function is not chosen with extreme care, for these so-called **collisions** to happen. Indeed, due to the birthday paradox, if the hash function spreads the indices in an uniform way, then there is at least a 50% chance of a collision when $n \geqslant (1+\sqrt{1+8s\log 2})/2$.

A good hash function should attempt to avoid too many collisions. There are many ways to attempt to do this. One of them it to treat the key, or rather, its memory representation, as a possibly very large integer, and to choose as hash function the remainder of the division of this large integer by the array size. When the key is a string this gives rise to code such as:

```
unsigned int hash_function(const char *str,unsigned int s)
{ // for 32-bit unsigned integers, s should be smaller that 16777216u
  unsigned int h;

  for(h = 0u;*str != '\0';str++)
    h = (256u * h + (0xFFu & (unsigned int)*str)) % s;
  return h;
}
```

It turns out that hash functions of this form are better (less collisions) when s is a prime number. (Furthermore, the order of 256 in the multiplicative group of remainders co-prime to s should be large; in particular, s should not be an even number.)

Hash tables (part 2b, more hash functions)

The hash function presented in the previous slide is reasonably good but it is slow, because it requires a remainder operation in each iteration of the for loop, and it works better when s is a prime number. Furthermore, for a 32-bit `unsigned int` data type, due to a possible integer overflow, it should not be used when `s` is larger than or equal to $2^{32-8} = 16777216$.

One way to solve these problems is to get rid of all but the last of the remainder operations, as done in the following variant of the hash function of the previous slide:

```c
unsigned int hash_function(const char *str,unsigned int s)
{
  unsigned int h;

  for(h = 0u;*str != '\0';str++)
    h = 157u * h + (0xFFu & (unsigned int)*str); // arithmetic overflow may occur here (just ignore it!)
  return h % s; // due to the unsigned int data type, it is guaranteed that 0 <= h % s < s
}
```

(The multiplication factor 157 was chosen is an almost arbitrary way.) Note that return values smaller that $2^{32} \bmod s$ (for a 32-bit data type) will be slightly more probable than those larger than or equal to $2^{32} \bmod s$ (and, of course, smaller that s). This defect does not cause any significant problem when s is many times smaller (say, 1000 times smaller) that the largest possible unsigned integer. Of course, this potential problem can be almost eliminated if 64-bit integers are used (`unsigned long long` data type).

Hash tables (part 2c, even more hash functions)

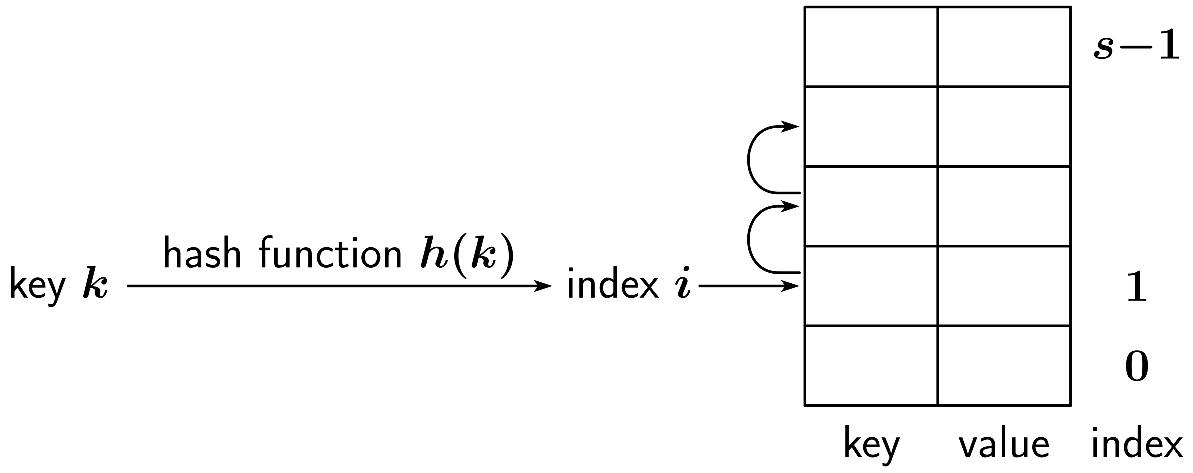
Other possible hash functions are based on so-called cyclic redundancy checksums (CRC), or on so-called message digest signatures (such as MD5), or on secure hash algorithms (such as SHA-1). The following hash function is based on a 32-bit cyclic redundancy checksum.

```
unsigned int hash_function(const char *str,unsigned int s)
{
  static unsigned int table[256];
  unsigned int crc,i,j;

  if(table[1] == 0u) // do we need to initialize the table[] array?
    for(i = 0u;i < 256u;i++)
      for(table[i] = i,j = 0u;j < 8u;j++)
        if(table[i] & 1u)
          table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
        else
          table[i] >>= 1;
  crc = 0xAED02016u; // initial value (chosen arbitrarily)
  while(*str != '\0')
    crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((unsigned int)*str++ << 24);
  return crc % s;
}
```

For the curious, the “magic” constant encodes the coefficients (bits) of a primitive polynomial in the finite field $GF(2^{32})$. In this case the polynomial is $x^{32} + x^{30} + x^{26} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^2 + 1$. [**Homework:** In the function given above replace the 32-bit CRC by a 64-bit CRC; use a “magic” constant of `0xAED0AED0AED0011Full`.]

Hash tables (part 3a, open addressing)

When a hash table uses open addressing the key and respective value are stored directly in the array:



This implies that $n \leqslant s$, and that it is necessary to resolve collisions by looking at other positions of the array when position i, with $i = h(k)$, does not contain the correct key. One possibility is to try $(i+1) \bmod s$, $(i+2) \bmod s$, and so on, until either the desired key is found or an empty array position is found. Instead of trying consecutive positions, it is also possible to try positions further apart, with jumps of j between positions: $(i+j) \bmod s$, $(i+2j) \bmod s$, and so on. For this to work is is necessary and sufficient that $\gcd(j,s) = 1$, which is ensured if $0 < j < s$ and if s is a prime number. Instead of using a fixed j, in **double hashing** each key uses its own j, computed by another hash function.

Open addressing has several major disadvantages:

- the hash table cannot have more that s keys
- when the hash table is nearly full and there are collisions the worst search time can be quite large
- it is difficult to remove keys from the hash table

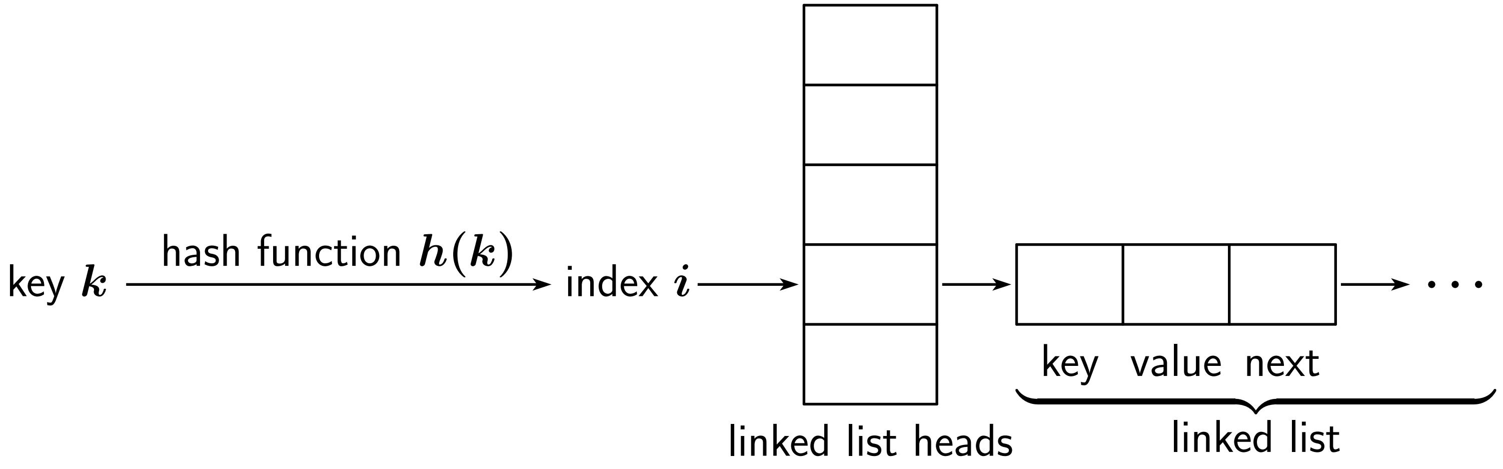
Hash tables (part 3b, open addressing)

The following C code exemplifies one way to perform a key search in a hash table that uses open addressing (compare to similar code that used separate chaining, presented in part 4b):

```
typedef struct hash_data
{
  char key[10]; // empty when key[0] = '\0'
  int value;
}
hash_data;

#define hash_size 1009u

hash_data hash_table[hash_size];

hash_data *find_data(const char *key)
{
  unsigned int idx;

  idx = hash_function(key,hash_size);
  while(hash_table[idx].key[0] != '\0' && strcmp(key,hash_table[idx].key) != 0)
    idx = (idx + 1u) % hash_size; // try the next array position
  return (hash_table[idx].key[0] == '\0') ? NULL : &hash_table[idx];
}
```

Hash tables (part 4a, separate chaining)

When the hash table uses chaining, sometimes also called separate chaining, the array stores pointers to the heads of linked lists. For each key the hash function specifies in which list we will operate (search, insert or remove). Of course, it is possible to replace the linked lists by a more sophisticated data structure, such as a binary search tree (or even another hash table!), that provides the same operations but with lower computational complexity.



When separate chaining is being used, the main purpose of the hash function is to distribute the keys as evenly an possible among the s positions of the array. A good spread implies a small number of collisions, and so a search operation will be fast. (If linked lists are being used, the worst search time is proportional to the length of the longest linked list.)

Because linked lists are dynamic data structures, a hash table that uses them can store more keys that the size of the array. The average search time will be $\max(1, n/s)$ if all keys are equally probable. So, the performance of a hash table implemented with separate chaining will degrade gracefully if its array is under-dimensioned.

Hash tables (part 4b, separate chaining)

The following C code exemplifies one way to perform a key search in a hash table that uses separate chaining (compare to similar code that used open addressing):

```
typedef struct hash_data
{
  struct hash_data *next;
  char key[10];
  int value;
}
hash_data;

#define hash_size 1009u

hash_data *hash_table[hash_size];

hash_data *find_data(const char *key)
{
  unsigned int idx;
  hash_data *hd;

  idx = hash_function(key,hash_size);
  hd = hash_table[idx];
  while(hd != NULL && strcmp(key,hd->key) != 0)
    hd = hd->next;
  return hd;
}
```

Hash tables (part 4c, separate chaining)

The following C code exemplifies how to allocate a new `hash_data` structure:

```
#include <stdio.h>
#include <stdlib.h>

hash_data *new_hash_data(void)
{
  hash_data *hd = (hash_data *)malloc(sizeof(hash_data));
  if(hd == NULL)
  {
    fprintf(stderr,"Out of memory\n");
    exit(1);
  }
  return hd;
}
```

The following C code exemplifies how to visit all nodes of a hash table:

```
void visit_all(void)
{
  unsigned int i;
  hash_data *hd;

  for(i = 0u;i < hash_size;i++)
    for(hd = hash_table[i];hd != NULL;hd = hd->next)
      visit(hd);
}
```

Hash tables (part 5, perfect hash functions)

A perfect hash function is an hash function that does not generate any collisions. Perfect hash functions are desired when the set of keys is fixed and known (for example, the reserved keywords of a programming language). In a perfect hash function it is usually desired that $s = n$ (to save space). As there are no collisions, the hash table is best implemented using open addressing.

Although perfect hash functions are rare (recall the birthday paradox), it is not difficult to construct a perfect hash function $h(k)$, that maps k to one of the integers $0, 1, \ldots, s-1$. Mathematically this can be expressed in a more compact way by $h : k \mapsto \{i\}_{i=0}^{s-1}$. The main idea is to use two distinct hash functions $h_1 : k \mapsto \{i\}_{i=0}^{s-1}$ and $h_2 : k \mapsto \{i\}_{i=0}^{t-1}$ and to use an auxiliary table T of size t. Indeed, when t is not too small ($t \approx s/2$ works well), and assuming that there are no key pairs that give rise to simultaneous collisions in the two hash functions, is it usually possible to construct the table in such a way that

$$h(k) = \big(h_1(k) + T[h_2(k)]\big) \bmod s$$

is a perfect hash function. Note that for a key k without a colision in $h_2(k)$ it is possible to assign to $h(k)$ any desired value.

Let K_i be the set of keys for which h_2 evaluates to i; mathematically we have $K_i = \{\ k\ :\ h_2(k) = i\ \}$. The table T can be constructed using a greedy approach by considering each K_i in turn, starting with the largest set and ending with the smallest set (the size of a set is its number of elements). For each i one tries $T[i] = 0$, $T[i] = 1$, and so on, until either s is reached (a failure!) or all the values of $h(k)$, for $k \in K_i$, do not collide with the values of $h(k)$ for the k belonging to the sets already dealt with. Sets with 1 or 0 elements, done at the end, do not pose any problem! In practice, this greedy approach works surprising well, if one is not too ambitious in the choice of t.

First written report work

— P.06 —

Summary:

- The subset sum problem.

Searching
— TP.07 —

Summary:

- Searching unordered data (in an array, in a linked list, or in a binary tree)
- How to improve the search time (data reordering)
- Searching ordered data (in an array — binary search — or in an ordered binary tree)

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Searching unordered data (part 1, array and linked list)

Searching for a certain piece of information when that information is not organized may require a complete pass over all data. That is an $O(n)$ operation. (If the information is stored in a hash table of appropriate size, searching for a given piece of information is, on average, a $O(1)$ operation. This is descriped in detail in the TP.06 slides.)

On an array, that can be done in as follows (the function returns the **first** index `i` for which `data[i] == value`, or -1 if none exists):

```
int find(T *data,int data_size,T value)
{
  int i;

  for(i = 0;i < data_size && data[i] != value;i++)
    ;
  return (i < data_size) ? i : -1;
}
```

On a linked list, it can be done in the following way (the function returns a pointer to the **first** node `n` for which `n->data == value`, or NULL if none exists):

```
node *find(node *head,T value)
{
  while(head != NULL && head->data != value)
    head = head->next;
  return head;
}
```

Searching unordered data (part 2,binary tree)

On an (unordered) binary tree, searching for a node n for which n->data == value can be done in the following way (note the use of recursion):

```
node *find(node *n,T value)
{
  node *nn;

  if(n == NULL || n->data == value)
    return n;
  if(n->left != NULL && (nn = find(n->left,value)) != NULL)
    return nn;
  if(n->right != NULL && (nn = find(n->right,value)) != NULL)
    return nn;
  return NULL;
}
```

If this function does not return NULL then it returns a pointer to **one** node that satisfies n->data == value (in an unordered tree, the concept of **first** is not well defined).

[**Homework:** It more than one tree node with the same value exist, which one is returned by this function?]

How to improve the search time

For unordered data, the search time can be improved if the number of queries for some data items is not uniformly distributed (check the example near the end of the TP.03 lecture).

One possible way to improve the average search time consists of reordering the information after each search, so that the data item that was found becomes closer to the beginning of the relevant data structure. For arrays its index becomes smaller, for linked lists its node becomes closer to the head of the list, and for binary trees its node becomes closer to the root of the tree.

The following function does this for an array:

```
int self_optimizing_find(T *data,int data_size,T value)
{
  int i;

  for(i = 0;i < data_size && data[i] != value;i++)
    ;
  if(i > 0 && i < data_size)
  {
    T tmp = data[i];
    data[i] = data[i - 1];
    data[--i] = tmp;
  }
  return (i < data_size) ? i : -1;
}
```

Other optimizations are possible. For example, if the same query has a tendency to be performed two or more times in a row, it will be advantageous to store (in a `static` variable) the result of the last query.

Searching ordered data (part 1a, array – binary search)

If the data is sorted in increasing (or decreasing) order, searching for a specific value can become a $O(\log n)$ operation. In the case where the information is stored in an array, the search can be performed using an algorithm known as binary search. For following C code presents one of its possible implementations.

```
int binary_search(T *data,int data_size,T value)
{
  int i_low = 0;
  int i_high = data_size - 1;
  int i_middle;

  while(i_low <= i_high)
  {
    i_middle = (i_low + i_high) / 2;
    if(value == data[i_middle])
      return i_middle; // this may not be the smallest possible index ...
    if(value > data[i_middle])
      i_low = i_middle + 1;
    else
      i_high = i_middle - 1;
  }
  return -1;
}
```

If the data is approximately uniformly distributed, it may be better to select the `i_middle` index by performing a linear interpolation between the points `(i_low,data[i_low])` and `(i_high,data[i_high])`; for an y coordinate of `value` the corresponding x coordinate should be (close to) `i_middle`.

Searching ordered data (part 1b, array – “improved” binary search)

The binary search function presented in the previous slide can be made to return the **first** index i for which data[i] == value (or −1 if none exists). One possible way of doing this is as follows:

```c
int binary_search(int *data,int data_size,int value)
{
  int i_low = -1;          // off by one
  int i_high = data_size; // off by one
  int i_middle;

  while(i_low + 1 != i_high)
  {
    //
    // loop invariants: data[i_high] >= value and data[i_low] < value
    // (by convention, data[-1] = -infinity and data[data_size] = +infinity)
    //
    i_middle = (i_low + i_high) / 2;
    if(data[i_middle] < value)
      i_low = i_middle;
    else
      i_high = i_middle;
  }
  return (i_high >= data_size || data[i_high] != value) ? -1 : i_high;
}
```

[**Homework:** Compare (empirically) the average number of loop iterations performed by the two binary search functions given in this and in the previous slide. Which one is better? Modify the function given in this slide so that it returns the **last** index in case of a match.]

Searching ordered data (part 2, ordered binary tree)

Searching for information in an ordered binary tree can be done in the following way (C code, compare with the unordered case given previously):

```
node *find_recursive(node *root,T value)
{
  if(root == NULL || root->data == value)
    return root;
  return find_recursive((value < root->data) ? root->left : root->right,value);
}
```

It will be a $O(\log n)$ operation if the binary tree is "balanced," but it can be an $O(n)$ operation in the worst possible case if the binary tree is completely unbalanced (with all its left links NULL or all its right links NULL).

A non-recursive implementation is also possible (in this case, it is simpler and faster):

```
node *find_non_recursive(node *root,T value)
{
  while(root != NULL && root->data != value)
    root = (value < root->data) ? root->left : root->right;
  return root;
}
```

First written report work

— P.07 —

Summary:

- The subset sum problem.

Sorting
— TP.08 —

Summary:

- Bubble sort and shaker sort
- Insertion sort and Shell sort
- Quicksort
- Mergesort
- Heapsort
- Other sorting routines (rank sort, selection sort)
- Computational complexity summary

Highly recommended bibliography for this lecture:

- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Other useful stuff (on the web):

- Visualization (animation) of how some algorithms work (see for example the animations in http://www.sorting-algorithms.com/.
- Description and useful information about many sorting algorithms.

Bubble sort and shaker sort (part 1a, bubble sort description)

The bubble sort algorithm, although very simple, is almost always worse that other sorting algorithms (such as the insertion sort algorithm to be explained later). It has, however, pedagogical interest. It compares adjacent array entries and exchanges them if they are nor ordered. It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Do pass.] For i equal to $0, 1, \ldots, n-2$ do step 2. When that is finished, go to step 3.
2. [Compare and exchange.] If `a[i]` is larger than `a[i+1]` exchange them.
3. [Terminate or do it again.] Terminate the algorithm if no exchange was done in the last pass (steps 1 and 2). Otherwise, go to step 1 and do it all again.

If the array is already sorted, only one pass is necessary (to confirm that it is already sorted). If the array is sorted in decreasing order, n passes are necessary (each pass moves the largest array element not yet in its place to its proper place). In general, the computational complexity of the best, average, and worst cases of the bubble sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively.

Bubble sort and shaker sort (part 1b, bubble sort code)

The following C code is one possible implementation of the bubble sort algorithm. (In the slides of this lecture, all sorting algorithms will have the same interface.)

```
void bubble_sort(T *data,int first,int one_after_last)
{ // sort data[first],...,data[one_after_last-1] in increasing order
  int i,i_low,i_high,i_last;

  i_low = first;
  i_high = one_after_last - 1;
  while(i_low < i_high)
  {
    for(i = i_last = i_low;i < i_high;i++)
      if(data[i] > data[i + 1])
      {
        T tmp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = tmp;
        i_last = i;
      }
    i_high = i_last;
  }
}
```

To sort an array named abc with 10 elements, just do bubble_sort(abc,0,10).

Bubble sort and shaker sort (part 2a, shaker sort description)

In the bubble sort algorithm a small entry near the end of the array may require many passes until it migrates to its final position. The so-called cocktail-shaker sort algorithm (it also only has pedagogical interest) attempts to ameliorate this problem by doing "up" and "down" passes. It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Do up pass.] For i equal to $0, 1, \ldots, n-2$ do step 2. When that is finished, go to step 3.
2. [Compare and exchange.] If `a[i]` is larger than `a[i+1]` exchange them.
3. [Terminate or do down pass.] Terminate the algorithm if no exchange was done in the last up pass (steps 1 and 2). Otherwise, go to step 4.
4. [Do down pass.] For i equal to $n-1, n-2, \ldots, 1$ do step 5. When that is finished, go to step 6.
5. [Compare and exchange.] If `a[i]` is smaller than `a[i-1]` exchange them.
6. [Terminate or do it again.] Terminate the algorithm if no exchange was done in the last down pass (steps 4 and 5). Otherwise, go to step 1 and do it all again.

Like bubble sort, the computational complexity of the best, average, and worst cases of the shaker sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively.

Bubble sort and shaker sort (part 2b, shaker sort code)

The following C code is one possible implementation of the shaker sort algorithm.

```c
void shaker_sort(T *data,int first,int one_after_last)
{
  int i,i_low,i_high,i_last;

  i_low = first;
  i_high = one_after_last - 1;
  while(i_low < i_high)
  {
    // up pass
    for(i = i_last = i_low;i < i_high;i++)
      if(data[i] > data[i + 1])
      {
        T tmp = data[i];
        data[i] = data[i + 1];
        data[i + 1] = tmp;
        i_last = i;
      }
    i_high = i_last;
    // down pass
    for(i = i_last = i_high;i > i_low;i--)
      if(data[i] < data[i - 1])
      {
        T tmp = data[i];
        data[i] = data[i - 1];
        data[i - 1] = tmp;
        i_last = i;
      }
    i_low = i_last;
  }
}
```

Insertion sort and Shell sort (part 1, insertion sort description and code)

Insertion sort, and its very simple improvement Shell sort, are very simple sorting algorithms that can be used in practice to sort small arrays (for larger arrays other sorting algorithms, with better asymptotic computational complexity, should be used). Insertion sort can be described as follows:

To sort a[0],...,a[n-1] in increasing order do:

1. [Do pass.] For i equal to $1, 2, \ldots, n-1$ do step 2. When that is finished, terminate the algorithm.
2. [Insert.] For j equal to $i, i-1, \ldots, 1$ and while a[j] is smaller than a[j-1] exchange a[j] with a[j-1].

Like the bubble and shaker sort algorithms, the computational complexity of the best, average, and worst cases of the insertion sort algorithm are $O(n)$, $O(n^2)$, and $O(n^2)$, respectively. **However**, the multiplicative constants hidden behind the asymptotic notation are better (smaller) for the insertion sort algorithm.

The following C code is one possible implementation of the insertion sort algorithm.

```c
void insertion_sort(T *data,int first,int one_after_last)
{
  int i,j;

  for(i = first + 1;i < one_after_last;i++)
  {
    T tmp = data[i];
    for(j = i;j > first && tmp < data[j - 1];j--)
      data[j] = data[j - 1];
    data[j] = tmp;
  }
}
```

Insertion sort and Shell sort (part 2, Shell sort description and code)

The Shell sort algorithm is nothing more that successive applications of the insertion sort algorithm to sub-arrays of the entire array. In a pass with stride h the entire array a[0],a[1],. . . ,a[n-1] is subdivided into the sub-arrays

a[0],a[h+0],a[2h+0],. . .   a[1],a[h+1],a[2h+1],. . .   · · ·   a[h-1],a[2h-1],a[3h-1],. . .

Any sequence of strides is possible, as long as the last one is a stride of 1 (i.e., the last pass is just a single insertion sort). The computational complexity of this algorithm depends on the sequence of strides that is used (the best sequences of strides is not known). There are sequences of strides that make the algorithm $o(n^2)$. For example, when $h(s) = 9 \cdot 2^s - 9 \cdot 2^{s/2} + 1$ when s is even and $h(s) = 8 \cdot 2^s - 6 \cdot 2^{(s+1)/2} + 1$ when it is odd, where s is the number of the pass (counting starts at the last one) gives rise to a $O(n^{4/3})$ algorithm.

The following C code is one possible implementation of the Shell sort algorithm.

```
void shell_sort(T *data,int first,int one_after_last)
{
  int i,j,h;

  for(h = 1;h < (one_after_last - first) / 3;h = 3 * h + 1)
    ;
  while(h >= 1)
  { // for each stride h, use insertion sort
    for(i = first + h;i < one_after_last;i++)
    {
      T tmp = data[i];
      for(j = i;j - h >= first && tmp < data[j - h];j -= h)
        data[j] = data[j - h];
      data[j] = tmp;
    }
    h /= 3;
  }
}
```

Quicksort (part 1, description)

In practice, a well implemented quicksort algorithm appears to be the fastest sorting algorithm. It is a recursive algorithm that can be described as follows:

To sort `a[lo]`,...,`a[hi-1]` in increasing order do:

1. [Terminal case.] If `hi` $-$ `lo` is smaller than a predetermined value (say, 20), use insertion sort to sort the array and terminate the algorithm.
2. [Select pivot.] Select one of the array elements to be the "pivot"
3. [Partition array.] Subdivide the array into three parts: the array elements that are smaller than the pivot (these are placed at the beginning), the pivot, or array elements equal to the pivot in certain implementations (these are placed at the middle), and the array elements that are larger than the pivot (these are placed at the end).
4. [Recurse.] Apply the same algorithm to the smaller than the pivot and to the larger than the pivot parts of the array. After doing this terminate the algorithm.

The partition step requires $O(n)$ work, where $n = \mathtt{hi} - \mathtt{lo}$ (see C code of the next slides). The computational complexity of the best, average, and worst cases of the quicksort algorithm are $O(n \log n)$, $O(n \log n)$, and $O(n^2)$, respectively. Without care, the worst case occurs when the array is already sorted!

Quicksort (part 2, code)

The following C code is one possible implementation of the quicksort algorithm.

```c
void quick_sort(T *data,int first,int one_after_last)
{
  int i,j,one_after_small,first_equal,n_smaller,n_larger,n_equal;
  T pivot,tmp;

  if(one_after_last - first < 20)
    insertion_sort(data,first,one_after_last);
  else
  {
    //
    // select pivot (median of three, the pivot's position will be one_after_last-1)
    //
#   define POS1  (first)
#   define POS2  (one_after_last - 1)
#   define POS3  ((first + one_after_last) / 2)
#   define TEST(pos1,pos2)  do if(data[pos1] > data[pos2])                          \
                               { tmp = data[pos1]; data[pos1] = data[pos2]; data[pos2] = tmp; } \
                               while(0)
    TEST(POS1,POS2);  // bitonic
    TEST(POS1,POS3);  // sort of
    TEST(POS2,POS3);  // 3 items
#   undef POS1
#   undef POS2
#   undef POS3
#   undef TEST
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
//
// 3-way partition. At the end of the while loop the items will be partitioned as follows:
// |first  "smaller part"|one_after_small  "larger part"|first_equal  "equal part"|one_after_last
//
one_after_small = first;
first_equal = one_after_last - 1;
pivot = data[first_equal];
i = first;
while(i < first_equal)
  if(data[i] < pivot)
  { // place data[i] in the "smaller than the pivot" part of the array
    tmp = data[i];
    data[i] = data[one_after_small]; // tricky! this does the right thing when
    data[one_after_small] = tmp;     //   i == one_after_small and when i > one_after_small
    i++;
    one_after_small++;
  }
  else if(data[i] == pivot)
  { // place data[i] in the "equal to the pivot" part of the array
    first_equal--;
    tmp = data[i];               // this is known to be the pivot, but we do it in this way
    data[i] = data[first_equal]; //   to make life easier to those that need to adapt this
    data[first_equal] = tmp;     //   code so that it deals with more complex data items
  }
  else
  { // data[i] becomes automatically part of the "larger than the pivot" part of the array
    i++;
  }
n_smaller = one_after_small - first;
n_larger = first_equal - one_after_small;
n_equal = one_after_last - first_equal;
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
    j = (n_equal < n_larger) ? n_equal : n_larger;
    for(i = 0;i < j;i++)
    { // move the "equal to the pivot" part of the array to the middle
      tmp = data[one_after_small + i];
      data[one_after_small + i] = data[one_after_last - 1 - i];
      data[one_after_last - 1 - i] = tmp;
    }
    //
    // recurse
    //
    quick_sort(data,first,first + n_smaller);
    quick_sort(data,first + n_smaller + n_equal,one_after_last);
  }
}
```

Mergesort (part 1, description)

Mergesort is a recursive algorithm that can be described as follows:

To sort `a[lo]`,...,`a[hi-1]` in increasing order do:

1. [Terminal case.] If `hi` − `lo` is smaller than a predetermined value (say, $\mathbf{20}$), use insertion sort to sort the array and terminate.
2. [Subdivide.] Subdivide the array into two nearly equal parts (the "first half" and the "second half".
3. [Recurse.] Apply the same algorithm the two two halves
3. [Merge.] merge the two already sorted halves. After doing this terminate the algorithm.

The merge step requires $O(n)$ work, where n = `hi` − `lo` (see C code in the next slide). The computational complexity of the best, average, and worst cases of the mergesort algorithm are all $O(n \log n)$.

Mergesort (part 2, code)

The following C code is one possible implementation of the mergesort algorithm.

```c
void merge_sort(T *data,int first,int one_after_last)
{
  int i,j,k,middle;
  T *buffer;

  if(one_after_last - first < 40) // do not allocate less than 40 bytes
    insertion_sort(data,first,one_after_last);
  else
  {
    middle = (first + one_after_last) / 2;
    merge_sort(data,first,middle);
    merge_sort(data,middle,one_after_last);
    buffer = (T *)malloc((size_t)(one_after_last - first) * sizeof(T)) - first; // no error check!
    i = first;  // first input (first half)
    j = middle; // second input (second half)
    k = first;  // merged output
    while(k < one_after_last)
      if(j == one_after_last || (i < middle && data[i] <= data[j]))
        buffer[k++] = data[i++];
      else
        buffer[k++] = data[j++];
    for(i = first;i < one_after_last;i++)
      data[i] = buffer[i];
    free(buffer + first);
  }
}
```

Heapsort (part 1, description)

The heapsort algorithm uses a max-heap (see the slides of the TP.06 lecture). It can be described as follows:

To sort `a[0]`,...,`a[n-1]` in increasing order do:

1. [Construct heap.] For $i = 0, 1, \ldots, n-1$ put `a[i]` in the max-heap.
2. [Sort.] For $i = n-1, n-2, \ldots, 0$ remove the largest element of the max-heap and store it in `a[i]`. After doing this terminate the algorithm.

Given the way this algorithm is structured the array and the heap can share the same memory area. The computational complexity of the best, average, and worst cases of the heap sort algorithm are, like for the mergesort algorithm, all $O(n \log n)$. heapsort, however, does not require extra space.

Heapsort (part 2, code)

The following C code is one possible implementation of the heapsort algorithm.

```
void heap_sort(T *data,int first,int one_after_last)
{
  int i,j,k,n;
  T tmp;

  data += first - 1;           // adjust pointer (data[first] becomes data[1])
  n = one_after_last - first; // number of items to sort
  //
  // phase 1. heap construction
  //
  for(i = n / 2;i >= 1;i--)
    for(j = i;2 * j <= n;j = k)
    {
      k = (2 * j + 1 <= n && data[2 * j + 1] > data[2 * j]) ? 2 * j + 1 : 2 * j;
      if(data[j] >= data[k])
        break;
      tmp = data[j];
      data[j] = data[k];
      data[k] = tmp;
    }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
  //
  // phase 2. sort (by successively removing the largest element)
  //
  while(n > 1)
  {
    tmp = data[1]; // largest
    data[1] = data[n];
    data[n--] = tmp;
    for(j = 1;2 * j <= n;j = k)
    {
      k = (2 * j + 1 <= n && data[2 * j + 1] > data[2 * j]) ? 2 * j + 1 : 2 * j;
      if(data[j] >= data[k])
        break;
      tmp = data[j];
      data[j] = data[k];
      data[k] = tmp;
    }
  }
}
```

Other sorting routines (part 1: rank sort)

The following C code is one possible implementation of the “rank” sort algorithm.

```c
void rank_sort(T *data,int first,int one_after_last)
{
  int i,j,*rank;
  T *buffer;

  rank = (int *)malloc((size_t)(one_after_last - first) * sizeof(int)) - first; // no error check!
  for(i = first;i < one_after_last;i++)
    rank[i] = first;
  for(i = first + 1;i < one_after_last;i++)
    for(j = first;j < i;j++)
      rank[(data[i] < data[j]) ? j : i]++;
  buffer = (T *)malloc((size_t)(one_after_last - first) * sizeof(T)) - first; // no error check!
  for(i = first;i < one_after_last;i++)
    buffer[i] = data[i];
  for(i = first;i < one_after_last;i++)
    data[rank[i]] = buffer[i];
  free(buffer + first);
  free(rank + first);
}
```

This algorithm has a fixed computational complexity of $O(n^2)$.

Other sorting routines (part 2: selection sort)

The following C code is one possible implementation of the selection sort algorithm.

```
void selection_sort(T *data,int first,int one_after_last)
{
  int i,j,k;

  for(i = one_after_last - 1;i > first;i--)
  {
    for(j = first,k = 1;k <= i;k++)
      if(data[k] > data[j])
        j = k;
    if(j < i)
    {
      T tmp = data[i];
      data[i] = data[j];
      data[j] = tmp;
    }
  }
}
```

This algorithm also has a fixed computational complexity of $O(n^2)$.

Computational complexity summary

The following table presents the computational complexity (of the number of comparisons and data movements) of the best, average, and worst cases of the sorting algorithms described in this lecture (n is the array size).

algorithm	best	average	worst	comments
bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	
shaker sort	$O(n)$	$O(n^2)$	$O(n^2)$	
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	
Shell sort	?	?	?	
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	
mergesort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	requires extra space
heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	
rank sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	requires extra space
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	

Elementary data structures

— P.08 —

Summary:

- Min-heap
- Priority queue
- Hash tables

Min-heap

Using the code presented in TP.06 as inspiration, implement a min-heap in C. Use the min-heap to sort an array of integers in decreasing order.

Priority queue

Using your min-heap, implement a priority queue (assume that lower values have higher priorities). Test it.

Hash tables

Using the code presented in the TP.06 lecture as inspiration, complete the implementation (in C) of a hash table (with separate chaining) capable of storing (key,value) pairs, in which the key is a string with at most 64 characters and in which the value is of type `T`. An incomplete implementation is stored in the file `hash_table.h` (`P08.tgz`). Use a hash table to count the number of times each word occurs in the text file `SherlockHolmes.txt`.[1] (Use as keys the words, and as values the number of times each one occurs.) Which word appears more times?

[1]Source: `http://sherlock-holm.es`.

Algorithmic techniques
— TP.09 —

Summary:

- Divide-and-conquer (DaC) and the master theorem
- DaC examples
- Dynamic programming (DP)
- DP examples

Recommended bibliography for this lecture:

- **Analysis of Algorithms**, Jeffrey J. McConnell, second edition, Jones and Bertlett Publishers, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Estruturas de Dados e Algoritmos em C**, António Adrego da Rocha, terceira edição, FCA.

Divide-and-conquer (DaC)

The **divide-and-conquer** algorithmic technique consists of solving a problem recursively using the following methodology:

Divide — If the problem is small enough, solve it directly. Otherwise, subdivide it into smaller instances (at least two, preferably of nearly equal size) of the same problem.

Conquer (recurse) — Solve each subproblem using the same algorithm.

Combine — Construct the solution of the problem by combining the solutions of the subproblems.

The Divide step can be trivial (almost nothing to do, just decide what subdivision to use), as in mergesort. The Combine step can also be trivial (nothing to do), as in quicksort. In general, both steps are nontrivial.

Let $T(n)$ be the effort required to solve a problem of size n, let $f(n)$ be the effort needed to perform steps Divide and Combine (again for a problem of size n), and let a be the number of subproblems of size n/b that are done in the Conquer step. Assigning an effort of 1 to problems of size $n \leqslant n_0$, we have

$$T(n) = \begin{cases} 1 & \text{for } n \leqslant n_0; \\ aT\left(\frac{n}{b}\right) + f(n) & \text{otherwise.} \end{cases}$$

The so-called **master theorem** provides solutions to this recursion in some cases:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = O(n^{\log_b a} \log n)$
- If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and if $af\left(\frac{n}{b}\right) \leqslant cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta\big(f(n)\big)$.

Divide-and-conquer (examples of application of the master theorem)

Mergesort is an example of a divide-and-conquer algorithm for which $f(n) = \Theta(n)$ and $a = b = 2$. The second case of the master theorem then says that $T(n) = O(n \log n)$.

In the improved multiplication method presented in lecture TP.04, $f(n) = \Theta(n)$, because additions and subtractions are $O(n)$, $a = 3$, and $b = 2$. The first case of the master theorem then says that $T(n) = O(n^{\log_2 3})$.

In the improved matrix multiplication method also presented in lecture TP.04, $f(n) = \Theta(n^2)$, because matrix additions and subtractions are $O(n^2)$, $a = 7$, and $b = 2$. The first case of the master theorem then says that $T(n) = O(n^{\log_2 7})$.

In the recursive exponentiation algorithms also presented in lecture TP.04, and in the binary search algorithm presented in lecture TP.07, $f(n) = \Theta(1)$, $a = 1$, and $b = 2$. The second case of the master theorem then says that $T(n) = O(\log n)$.

Homework: Apply the master theorem to each of the following recurrences:

$$T(n) = 2T\left(\tfrac{n}{2}\right) + n^4$$
$$T(n) = T\left(\tfrac{7n}{10}\right) + n$$
$$T(n) = 16T\left(\tfrac{n}{4}\right) + n^2$$
$$T(n) = 7T\left(\tfrac{n}{3}\right) + n^2$$
$$T(n) = 7T\left(\tfrac{n}{2}\right) + n^2$$
$$T(n) = 2T\left(\tfrac{n}{4}\right) + \sqrt{n}$$

Possible solution: In this case we have $a = 2$ — the factor before $T(\cdot)$ on the right hand side — $b = 2$ — the divisor of n inside $T(\cdot)$ — and $f(n) = n^4$. Since $\log_b a = \log_2 2 = 1$, we have $f(n) = \Theta(n^{1+3})$, that is, $\epsilon = 3$. Since $\epsilon > 0$, we are possibly in the last case of the master theorem. This will be so if and only if $af\left(\frac{n}{b}\right) \leqslant cf(n)$, for some $c < 1$. In this case $af\left(\frac{n}{b}\right) = \frac{1}{8}f(n)$, so any c larger than $\frac{1}{8}$ will do, and the third case of the master theorem can be applied. It follows that $T(n) = \Theta(n^4)$.

DaC examples (part 1a, selection)

Given an array of (unsorted) numbers, select the k-th smallest array element. For example, for $k = 0$ the smallest array element is the one selected, for k equal to the size of the array minus one the largest array element is the one selected, and for k equal to half the size of the array the median of the arrays elements is the one selected. By sorting the array in increasing order, a task that requires $O(n \log n)$ time, each selection takes $O(1)$ time.

If only one selection is needed, it is not necessary to sort the array. Using the divide-and-conquer paradigm, selection can be done in $O(n)$ average time. The following function does the job in $O(n^2)$ time (without sorting!), and so it should be used only for very small arrays:

```c
int select_v1(int *a,int n,int k)
{ // O(n^2) algorithm, could be improved to O(n log n)
  for(int i = 0;i < n;i++)
  {
    int rank = 0;
    for(int j = 0;j < n;j++)
      if(a[j] < a[i] || (a[j] == a[i] && j < i))
        rank++;
    if(rank == k)
      return a[i];
  }
  abort(); // if 0 <= k < n, this point cannot be reached
}
```

It will be used by the divide-and-conquer function of the next slide to deal with the terminal cases (i.e., the small ones). For that function, and assuming that the pivot splits the array in approximately equal-sized smaller and larger parts, we have $T(n) = T(n/2) + O(n)$, which, according to the master theorem, gives $T(n) = O(n)$.

DaC examples (part 1b, selection using DaC)

The following code solves the selection problem using the divide-and-conquer paradigm. Its main idea is the choose a pivot element at random, and to split the array into three parts: smaller than, equal to, and larger than the pivot (just like quicksort). If the desired selection lies on the smaller or larger parts, then the same problem is solved for the appropriate part (recursion!).

```
int select_v2(int *a,int n,int k)
{ // O(n) on average, at worst O(n^2)
  int i,j,pivot,n_smaller,n_larger,*aa;

  if(n <= 10)
    return select_v1(a,n,k);
  pivot = a[(int)rand() % n];                         // choose pivot at random
  for(n_smaller = n_larger = i = 0;i < n;i++)         // count number of elements
    if(a[i] < pivot)      n_smaller++;                //   smaller than the pivot
    else if(a[i] > pivot) n_larger++;                 //   larger than the pivot
  if(k >= n_smaller && k < n - n_larger) return pivot; // bingo!
  if(k < n_smaller)
  { // divide (keep only the elements smaller than the pivot) and conquer
    aa = (int *)malloc((size_t)n_smaller * sizeof(int));
    for(i = j = 0;i < n;i++) if(a[i] < pivot) aa[j++] = a[i];
    i = select_v2(aa,n_smaller,k);
  }
  else
  { // divide (keep only the elements larger than the pivot) and conquer
    aa = (int *)malloc((size_t)n_larger * sizeof(int));
    for(i = j = 0;i < n;i++) if(a[i] > pivot) aa[j++] = a[i];
    i = select_v2(aa,n_larger,k - (n - n_larger));
  }
  free(aa);
  return i;
}
```

DaC examples (part 2a, the maximum-sum subarray problem)

Given an array of numbers, the maximum-sum subarray problem consists of finding the maximum sum of consecutive elements of the array. Let $a[0], \ldots, a[n-1]$ be the elements of the array (of size n). Mathematically, the problem is to solve

$$\max_{0 \leqslant i \leqslant j < n} \sum_{k=i}^{j} a[k] \qquad \text{and} \qquad \arg\max_{0 \leqslant i \leqslant j < n} \sum_{k=i}^{j} a[k].$$

The notation $\arg\max$ denotes the argument where the maximum occurs, or one argument chosen arbitrarily if the minimum occurs in several places; in this case the argument is the pair (i, j), with $0 \leqslant i \leqslant j < n$. The following code solves this problem in $O(n^2)$ time:

```
typedef struct { int first; int last; int sum; } ret_val;

ret_val max_sum_v1(int *a,int first,int one_after_last)
{
  ret_val r = { first,first,a[first] };

  for(int i = first;i < one_after_last;i++)
    for(int sum = 0,j = i;j < one_after_last;j++)
    {
      sum += a[j];
      if(sum > r.sum)
      {
        r.first = i;
        r.last = j;
        r.sum = sum;
      }
    }
  return r;
}
```

DaC examples (part 2b, the maximum-sum subarray problem using DaC)

The maximum-sum subarray either lies completely in the first half of the array, either lies completely in the second half of the array, or in must cross the middle of the array. This last case can be solved in $O(n)$ time, so the following code solves the problem in $O(n \log n)$ time using divide-and-conquer:

```
ret_val max_sum_v2(int *a,int first,int one_after_last)
{
  int i,sum,max,middle;
  ret_val r1,r2,r3;

  if(one_after_last - first < 20)
    return max_sum_v1(a,first,one_after_last);
  else
  {
    // divide
    middle = (first + one_after_last) / 2;
    // recurse
    r1 = max_sum_v2(a,first,middle);
    r2 = max_sum_v2(a,middle,one_after_last);
    // combine
    sum = max = a[r3.first = middle - 1];
    for(i = middle - 2;i >= first;i--) if((sum += a[i]) > max) { max = sum; r3.first = i; }
    r3.sum = max; // best left half (at least one element)
    sum = max = a[r3.last = middle];
    for(i = middle + 1;i < one_after_last;i++) if((sum += a[i]) > max) { max = sum; r3.last = i; }
    r3.sum += max; // add best right half (at least one element)
  }
  if(r2.sum > r1.sum)
    r1 = r2;
  if(r3.sum > r1.sum)
    r1 = r3;
  return r1;
}
```

DaC examples (part 2c, a better solution to the maximum-sum subarray problem)

In the particular case of the maximum-sum subarray problem the divide-and-conquer programming paradigm does not produce an algorithm with the best computational complexity. The following function solves the problem in only $O(n)$ time! This is achieved by scanning the array once, keeping track of the best (largest) sum of consecutive array elements ending at the scan location:

```
ret_val max_sum_v3(int *a,int first,int one_after_last)
{
  int i,max_sum_ending_here,first_max_sum_ending_here;
  ret_val r = { first,first,a[first] };

  max_sum_ending_here = a[first_max_sum_ending_here = first];
  for(i = first + 1;i < one_after_last;i++)
  {
    if(max_sum_ending_here > 0)
      max_sum_ending_here += a[i]; // max_sum_ending_here + a[i] is better than just a[i]
    else
      max_sum_ending_here = a[first_max_sum_ending_here = i]; // just a[i] is better
    if(max_sum_ending_here > r.sum)
    {
      r.first = first_max_sum_ending_here;
      r.last = i;
      r.sum = max_sum_ending_here;
    }
  }
  return r;
}
```

This solution can be considered to be a particular case of the application of the dynamic programming paradigm (this case does not require memoization!), to be discussed in the next slides.

Dynamic programming (DP)

The dynamic programming technique solves a problem by combining the solutions of smaller problems (solved recursively). It is useful when these subproblems share subsubproblems. Time efficiency is gained, at the expense of extra storage space, by solving each subproblem only once and by storing its output in an array, or other data structure, for later reuse (this is called **memoization**).

Dynamic programming is usually used to solve optimization problems for which the solution to an order n problem can be found by combining the solutions to one or more order $n - 1$ problems. It can be applied with advantage:

- if an optimal solution to an order n problem contains within it optimal solutions to order $n - 1$ problems, which in turn contain optimal solutions to order $n - 2$ problems, and so on;
- when the solution of the subproblems share subsubproblems.

If a problem does not have these two characteristics then the dynamic programming technique either cannot be applied or it will not solve the problem efficiently.

For example, consider a network of roads connecting several cities. The problem of finding the shortest route between two cities A and B can be solved using dynamic programming, because if that route passes through city C then we can be assured that the route from A to C is the shortest possible (if that were not the case we could produce a better solution by replacing the route from A to C by the shorter route). On the other hand, the problem of finding the longest route that does not visit a city **more than once** (that constraint is not present in the smallest route problem, because there a route with a loop cannot be optimal) cannot be solved by dynamic programming, because if that route passes through city C then the route from A to C might not be the longest route between these two cities (that route may pass through a city that appears later in the longest route from A to B).

DP examples (part 1, two simple dynamic programming examples)

The Fibonacci numbers are defined by the recursion

$$F_n = F_{n-1} + F_{n-2}, \qquad n > 1,$$

with initial conditions $F_0 = 0$ and $F_1 = 1$. They can be computed as follows:

```
int F_v1(int n)
{
  return (n < 2) ? n : F_v1(n - 1) + F_v1(n - 2);
}
```

Using a dynamic programming approach, i.e., by storing and reusing previously computed results, the previous code becomes **much** faster at the expense of a small amount of memory:

```
int F_v2(int n)
{
  static int Fv[50] = { 0,1 };

  if(n > 1 && Fv[n] == 0)
    Fv[n] = F_v2(n - 1) + F_v2(n - 2);
  return Fv[n];
}
```

The same trick can be used to compute the binomial coefficients (number of combinations of n objects taken k at a time), denoted by C_k^n, which are given by

$$C_k^n = \begin{cases} \dfrac{n!}{k!(n-k)!}, & \text{if } 0 \leqslant k \leqslant n, \\ 0, & \text{otherwise (by convention).} \end{cases}$$

Using this formula directly may lead to arithmetic overflow, so one can use the following recursive formula instead (Pascal's triangle):

$$C_k^n = \begin{cases} 0, & \text{if } k < 0 \text{ or } k > n, \\ 1, & \text{if } k = 0 \text{ or } k = n, \\ C_{k-1}^{n-1} + C_k^{n-1}, & \text{otherwise.} \end{cases}$$

This gives rise to the following function:

```
int C(int n,int k)
{
  static int Cv[100][100];

  if(k < 0 || k > n)   return 0;
  if(k == 0 || k == n) return 1;
  if(Cv[n][k] == 0)
    Cv[n][k] = C(n - 1,k - 1) + C(n - 1,k);
  return Cv[n][k];
}
```

DP examples (part 2a, the maximum-sum scattered-subarray problem)

Given an array of positive numbers, the maximum-sum scattered-subarray problem consists of finding the maximum sum of elements of the array, with the restriction that adjacent array elements cannot both contribute to the sum. (Alternative problem formulation: given a line of n bottles with volumes $v_0, \ldots, v_{n-1}$, drink as much as you can, given that you cannot drink from adjacent bottles.)

This problem can be solved easily for size n if the solutions for the sizes $n-1$ and $n-2$ are known, as illustrated by the following recursive function:

```c
int max_scattered_sum_v1(int *v,int n)
{
  if(n == 1)
    return v[0];                                 // one is better than nothing
  if(n == 2)
    return (v[0] > v[1]) ? v[0] : v[1];          // choose the larger of the two
  int t1 = max_scattered_sum_v1(v,n - 2) + v[n - 1]; // use v[n - 1], cannot use v[n - 2]
  int t2 = max_scattered_sum_v1(v,n - 1);        // do not use v[n - 1], can use v[n - 2]
  return (t1 > t2) ? t1 : t2;                    // choose the larger of the two
}
```

The number of function calls done by `max_scattered_sum_v1()` is exactly the same as the number of function calls done by `F_v1()` of the previous slide. As that number grows exponentially, this function is useless for n greater than, say, 40. The next slide shows how this severe handicap can be eliminated (memoization saves the day!).

DP examples (part 2b, the maximum-sum scattered-subarray problem with DP)

To speed up max_scattered_sum_v1() enormously simply record and reuse its return value. This gives rise to the function max_scattered_sum_v2(), presented below on the left-hand side (notice that it is possible to reset the memoized data by using special values of the function arguments). In this case it is also possible to get rid of the memoized data entirely, as illustrated in max_scattered_sum_v3(), presented below on the right-hand side.

```c
int max_scattered_sum_v2(int *v,int n)
{ // O(n) on first call, O(1) on subsequent calls
  static int r[1001]; // n cannot be larger than 1000

  if(v == NULL || n <= 0 || n > 1000)
  { // invalidate memoized data
    for(int i = 0;i <= 1000;i++)
      r[i] = -1;
    return -1;
  }
  if(r[n] < 0)
  { // compute for the first time
    if(n == 1)
      r[n] = v[0];
    else if(n == 2)
      r[n] = (v[0] > v[1]) ? v[0] : v[1];
    else
    {
      int t1 = max_scattered_sum_v2(v,n - 2) + v[n - 1];
      int t2 = max_scattered_sum_v2(v,n - 1);
      r[n] = (t1 > t2) ? t1 : t2;
    }
  }
  return r[n];
}
```

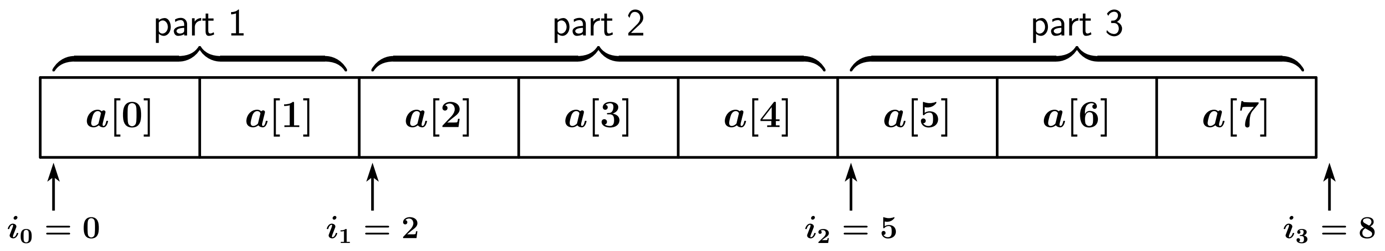
```c
int max_scattered_sum_v3(int *v,int n)
{ // always O(n) time and O(1) space
  if(n == 1)
    return v[0];
  int t1 = v[0];
  int t2 = (v[0] > v[1]) ? v[0] : v[1];
  for(int i = 2;i < n;i++)
  {
    int t3 = (t1 + v[i] > t2) ? t1 + v[i] : t2;
    t1 = t2;
    t2 = t3;
  }
  return t2;
}
```

DP examples (part 3a, best partition problem)

Given an array of $n > 0$ positive integers $a[0], a[1], \ldots, a[n-1]$, the best partition problem asks for the best way to partition these integers into $m \geqslant 1$ consecutive ranges covering the entire array so that the maximum sum of the array elements over each range is minimized. Mathematically, it asks for a set of indices $i_0, i_1, \ldots, i_m$, with $i_0 = 0$, $i_k \leqslant i_{k+1}$ for $k = 0, 1, \ldots, m-1$ and $i_m = n$, such that

$$\max_{0 \leqslant k < m} \sum_{i_k \leqslant i < i_{k+1}} a[i]$$

is minimized. By convention the sum is zero when the summation range is empty. For example, for $n = 8$ and $m = 3$ one possible partition of the array can be



Let $M[j, i]$ be the minimum of the maximum sums of the array elements $a[0], \ldots a[i-1]$ when they are subdivided into j ranges. We are interested in the value of $M[m, n]$ and in one partition that achieves it. The boundary cases gives us

$$M[1, i] = \sum_{0 \leqslant k < i} a[k], \quad 0 \leqslant i \leqslant n, \qquad \text{and} \qquad M[j, 0] = 0, \quad 2 \leqslant j \leqslant m,$$

and adding one more partition gives us (note that $\sum_{c \leqslant k < i} a[k] = M[1, i] - M[1, c]$)

$$M[j, i] = \min_{0 \leqslant c \leqslant i} \max\bigl(M[j-1, c], M[1, i] - M[1, c]\bigr), \quad 1 \leqslant i \leqslant n, 2 \leqslant j \leqslant m.$$

DP examples (part 3b, code for the best partition problem)

The recursive formula for $M[j,i]$ gives rise to an order $O(mn^2)$ algorithm to solve the best partition problem, as shown in the following code:

```
int best_partition(int *a,int n,int m,int show)
{
  int M[m + 1][n + 1]; // M[j][i] stores the best cost for the subproblem with n=i and k=j
  int D[m + 1][n + 1]; // D[j][i] records the best partition point to get to M[j][i]
  int I[m + 1];        // partition indices
  int i,j,c,best_c,cost,best_cost;

  assert(n >= 1 && m >= 1);
  //
  // boundary cases
  //
  M[1][0] = 0;
  D[1][0] = 0;
  for(i = 1;i <= n;i++)
  {
    M[1][i] = M[1][i - 1] + a[i - 1]; // array elements in one partition
    D[1][i] = 0;
  }
  for(j = 2;j <= m;j++)
  {
    M[j][0] = 0; // zero array elements
    D[j][0] = 0;
  }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
//
// apply dynamic programming to solve all subproblems
//
for(j = 2;j <= m;j++)
  for(i = 1;i <= n;i++)
  {
    best_c = 0;
    best_cost = M[1][i] - M[1][0];
    for(c = 1;c <= i;c++)
    {
      cost = (M[j-1][c] > M[1][i] - M[1][c]) ? M[j-1][c] : M[1][i] - M[1][c];
      if(cost < best_cost)
      {
        best_cost = cost;
        best_c = c;
      }
    }
    M[j][i] = best_cost;
    D[j][i] = best_c;
  }
//
// construct best partition
//
I[m] = n;
for(i = n,j = m;j > 0;j--)
  i = I[j - 1] = D[j][i]; // partition separator
assert(I[0] == 0);
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
  //
  // show solution
  //
  if(show > 1)
    for(j = 1;j <= m;j++)
      for(i = 0;i <= n;i++)
        printf("%2d[%2d]%s",M[j][i],D[j][i],(i == n) ? "\n" : " ");
  if(show > 0)
  {
    printf("<%d> ",M[m][n]);
    for(i = j = 0;i <= n;i++)
    {
      if(i == I[j])
      {
        printf("|%s",(i == n) ? "\n" : " ");
        j++;
      }
      if(i < n)
        printf("%d ",a[i]);
    }
  }
  //
  // done (we should also return the best partition, but in C that is cumbersome)
  //
  return M[m][n];
}
```

DP examples (part 3c, best partition example)

Consider the problem of subdividing the array

8 4 6 1 2 7 2 1 1 1 7 9

into 4 parts so that the largest sum of the elements of each part is as small as possible. In this case the contents of the `M[j][i]` array of the function presented in the previous slides is

$j\backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	8	12	18	19	21	28	30	31	32	33	40	49
2	0	8	8	10	11	12	16	18	18	18	18	21	28
3	0	8	8	8	8	9	10	11	12	12	12	16	18
4	0	8	8	8	8	8	9	9	10	10	10	11	**16**

and the contents of the `D[j][i]` array, useful to retrace the choices make by the algorithm and so find the partition boundaries, is

$j\backslash i$	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	**0**	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	**2**	2	2	3	3	3	4	5
3	0	0	1	2	2	2	3	4	4	5	**5**	6	8
4	0	0	1	2	2	3	4	5	5	6	6	7	**10**

From this information it is possible to extract (in bold in the `D[j][i]` array) the best partition

| 8 4 | 6 1 2 | 7 2 1 1 1 | 7 9 |

It has a maximum sum of 16 (in bold in the `M[j][i]` array).

DP examples (part 4a, smallest edit distance)

Given the strings $a[0], a[1], \ldots, a[n_a - 1]$ and $b[0], b[1], \ldots, b[n_b - 1]$, find the best sequence of insertions, deletions, and character changes, that transforms the first string into the second. The cost of an insertion or a deletion is I and the cost on a character change is X. (As insertion in $a[]$ can be considered a deletion in $b[]$, so the cost of an insertion should be the same as that of a deletion.)

Let $D[i, j]$ be the best (smallest) distance between the substrings $a[0], \ldots a[i - 1]$ and $b[0], \ldots, b[j - 1]$. We are interested in the value of $D[n_a, n_a]$, and in how one gets there in an optimal way starting from $D[0, 0]$ (two empty strings). Clearly, we have $D[i, 0] = iI$ and $D[0, j] = jI$, i.e., $D[i, j] = \max(i, j)I$ when $\min(i, j) = 0$. We also have

$$D[i,j] = \begin{cases} D[i-1, j-1], & \text{if a[i-1]=b[j-1],} \\ \min\bigl(D[i-1, j] + I, D[i, j-1] + I, D[i-1, j-1] + X\bigr), & \text{otherwise.} \end{cases}$$

This formula gives rise to the following function to partially solve the problem (just compute $D[n_a, n_b]$):

```c
int distance_v1(char *a,char *b,int i,int j,int I,int X)
{
  if(i == 0 || j == 0)
    return ((i > j) ? i : j) * I;
  int d1 = distance_v1(a,b,i - 1,j,I,X) + I;                              // one more in a[]
  int d2 = distance_v1(a,b,i,j - 1,I,X) + I;                              // one more in b[]
  int d3 = distance_v1(a,b,i - 1,j - 1,I,X) + ((a[i -1] == b[j - 1]) ? 0 : X); // one more in a[] and in b[]
  if(d1 < d2 && d1 < d3)
    return d1;
  return (d2 < d3) ? d2 : d3;
}
```

DP examples (part 4b, smallest edit distance using dynamic programming)

The code of the previous slide suffers from two problems: the distance function may be called **more than once** with the same arguments (inefficient!), and, because the C language does not allow more than one return argument (if we need that, we need to return a structure), it is not trivial to retrieve the best sequence of insertions/deletions and of character exchanges. All this is solved if the values returned by the function are cached. The resultant algorithm is a dynamic programming algorithm. It uses $O(n_a n_b)$ time and space. It the best sequence of insertions/deletions and exchanges is not needed the space requirements can be lowered to $O\big(\min(n_a, n_b)\big)$.

```
int distance_v2(char *a,char *b,int na,int nb,int I,int X)
{
  int i,j,D[na + 1][nb + 1];

  for(i = 0;i <= na;i++)  D[i][0] = i * I;
  for(j = 0;j <= nb;j++)  D[0][j] = j * I;
  for(i = 1;i <= na;i++)
    for(j = 1;j <= nb;j++)
    {
      int d1 = ((D[i - 1][j] < D[i][j - 1]) ? D[i - 1][j] : D[i][j - 1]) + I;
      int d2 = D[i - 1][j - 1] + ((a[i - 1] == b[j - 1]) ? 0 : X);
      D[i][j] = (d1 < d2) ? d1 : d2;
    }
  return D[na][nb];
}
```

If the best sequence of insertions/deletions and exchanges were needed, one would only need to start at $D[n_a, n_b]$ and attempt to reach $D[0, 0]$ in the smallest number of moves to an adjacent position (either decreasing i by 1, j by 1, or i and j by 1), as illustrated in the next slide. When called with `a[] = "destruction"` ($n_a = 11$) and `b[] = "construction"` ($n_b = 12$), `distance_v1()` is **one million times slower** than `distance_v2()`!

DP examples (part 4c, smallest edit distance example)

The following table presents the values of $D[i,j]$ (`D[i][j]` in the C function) when `a[] = "master"` and `b[] = "small edit distance"`. One one the best ways to transform one string into the other is depicted in bold.

	b[j-1]		's'	'm'	'a'	'l'	'l'	' '	'e'	'd'	'i'	't'	' '	'd'	'i'	's'	't'	'a'	'n'	'c'	'e'
a[i-1]	$i \backslash j$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
	0	**0**	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
'm'	1	1	1	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
'a'	2	2	2	2	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	12	13	14	15	16	17
's'	3	3	2	3	2	2	3	4	5	6	7	8	9	10	11	**11**	12	13	14	15	16
't'	4	4	3	3	3	3	3	4	5	6	7	7	8	9	10	11	**11**	**12**	**13**	14	15
'e'	5	5	4	4	4	4	4	4	4	5	6	7	8	9	10	11	12	12	13	**14**	14
'r'	6	6	5	5	5	5	5	5	5	5	6	7	8	9	10	11	12	13	13	14	**15**

This path was constructed by selecting, at position (i,j), its neighbor with the smallest value. Only the neighbors $(i-1,j-1)$, to the left and up, $(i-1,j)$, to the left, and $(i,j-1)$, up, need to be considered. In the case of a tie the choice of neighbor is arbitrary.

Possible final written report (Huffman encoder and decoder) — P.09 —

Summary:

- Huffman code
- Huffman encoder
- Huffman decoder
- Final remarks

Huffman code

A binary Huffman code assigns to each alphabet symbol S_i, which is assumed to occur in a data source with probability p_i, a codeword C_i composed by N_i bits. It is a variable-length code.

The Huffman code selects codewords so that the average codeword length $\sum_i p_i N_i$ is as small as possible. This is achieved by assigning codewords with more bits to symbols with low probabilities. (The exact way this can be done is described in the following slides.) The Huffman code is not unique.

Let $H = -\sum_i p_i \log_2 p_i$ be the entropy of the alphabet ($\log_2 x$ is the base-2 logarithm of x). It is known that the average codeword length of a binary Huffman code cannot is smaller than H and that it cannot be larger than or equal to $H + 1$. Furthermore, no other code with codewords with an integer number of bits can be better than a Huffman code.

For more information, consult section 2.8 of

- **Data Compression, The Complete Reference**, David Solomon, fourth edition, Springer, 2007.

Huffman encoder (part 1, code construction)

A binary Huffman code for n symbols is constructed in the following way.

- The two symbols with smallest probabilities are replaced by a virtual symbol that aglomerates the two.
- The probability of the virtual symbol is the sum of the probabilities of the two symbols.
- The codeword of the two symbols is constructed by appending $\mathbf{0}$ and $\mathbf{1}$ to the codeword of the virtual symbol.
- The problem is thus reduced to constructing a binary Huffman code for $n - 1$ symbols.
- When there is only one symbol there is nothing more to do: the codeword of the single symbol has 0 bits.

In order to do this efficiently, it is necessary to

- identify the two symbols with smallest probabilities; that will be done using a **min-heap**,
- keep track of the symbols that were merged; that will be done using a **binary tree**,
- after the construction of the so-called Huffman tree is finished, it is necessary to assign codewords to the symbols.

In our case, we also need to estimate the probabilities of each symbol, by counting the number of times they occur in the data source (a file). In order to do this efficiently, one possibility is to

- use a **hash table**, in which the key is the symbol and the value is a counter of the number of times it occurs.

Huffman encoder (part 2, data structure)

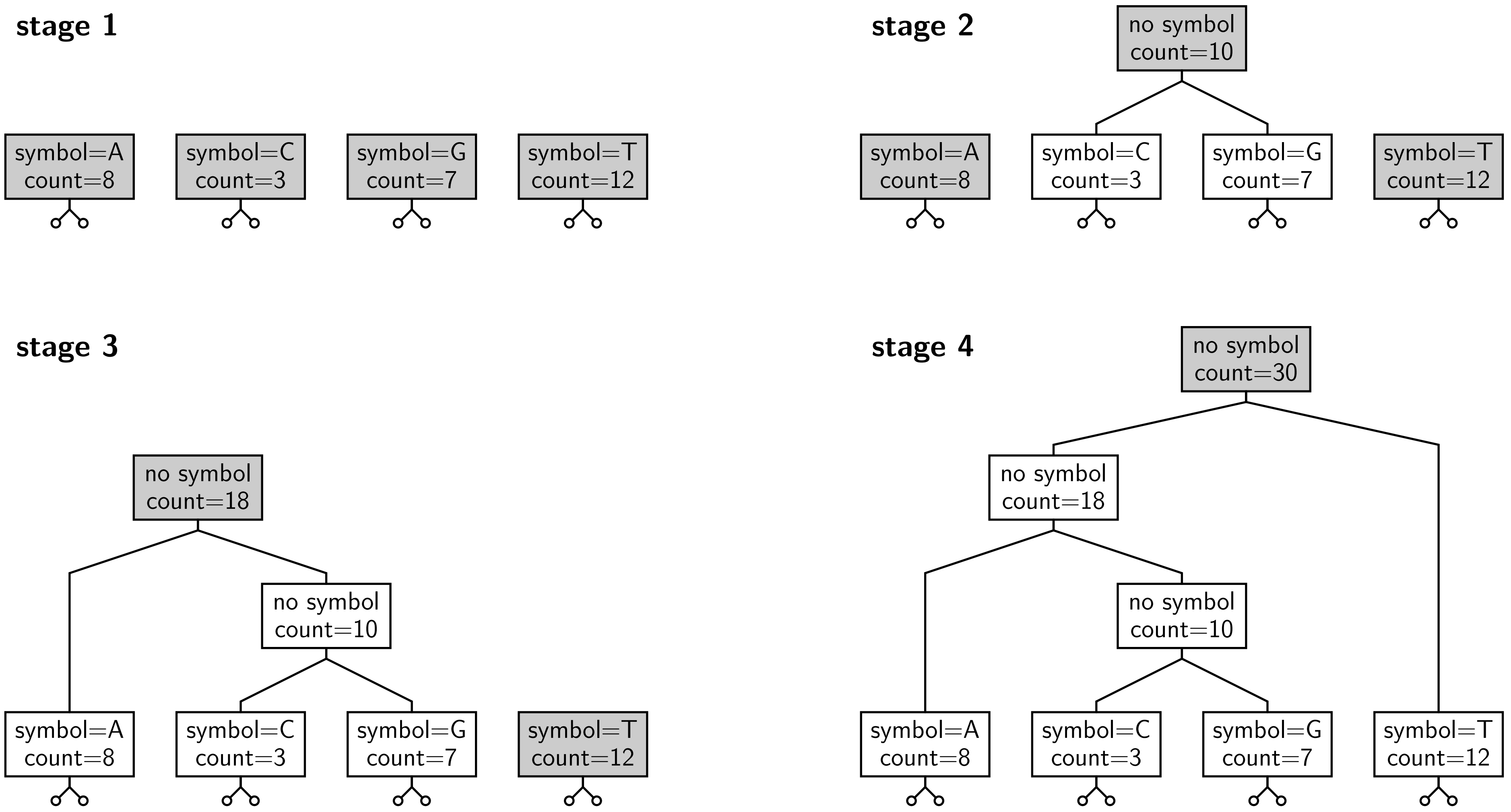
The following C data structure will allow us to do all that is necessary. (In an object-oriented programming language with an inheritance mechanism, things could be done in a different way.)

```
typedef struct node
{
  //
  // data stored in the node (symbol, symbol size (number of bytes), and number of occurrences
  //
  char symbol[MAX_SYMBOL_SIZE]; // the symbol (a word or a group of characters of other type, NOT terminated by '\0')
  int symbol_size;              // number of bytes stored in symbol[]
  int count;                    // number of occurrences of the symbol
  //
  // pointers
  //
  struct node *next;            // for a linked list (hash table with chaining)
  struct node *left;            // for the Huffman binary tree
  struct node *right;           // for the Huffman binary tree
  //
  // the Huffman code
  //
  uint64_t code;                // 64 bit unsigned integer
  int code_bits;
}
node;
```

It has fields needed by the nodes of the hash table (used for counting the number of times each symbol appears), and fields needed by the binary tree (used to construct the Huffman code). The min-heap is to be organized by the value of the `count` field. Note that the min-heap should store pointers to the nodes.

Huffman encoder (part 3, Huffman tree construction example)

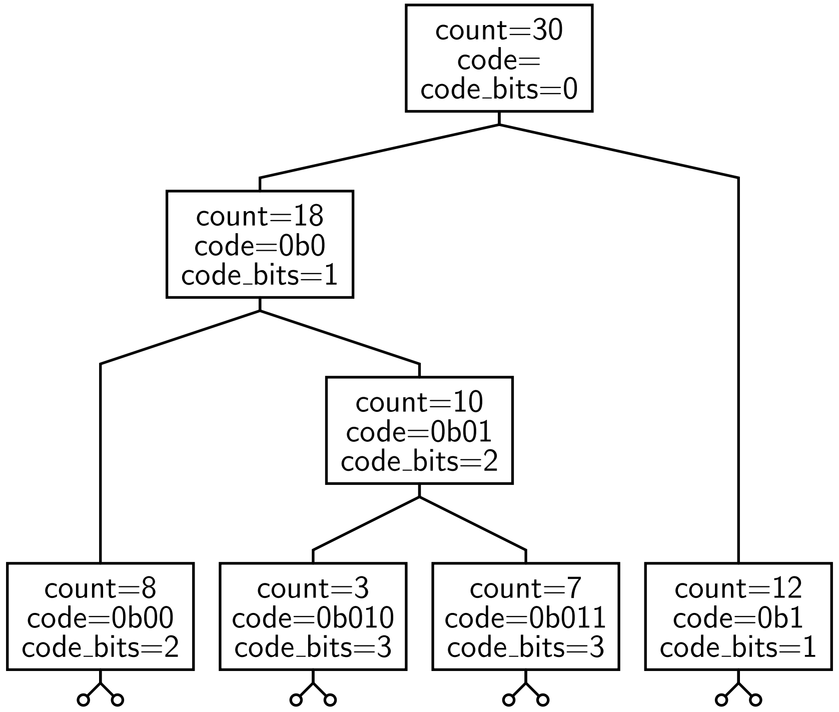
The following figure exemplifies how the Huffman tree is constructed. Nodes with a gray background are the ones currently stored in the min-heap.



To pass from one stage to the next a new node is created, its left and right pointers are initialized with nodes retrieved from the min-heap, and the new node is put back in the min-heap.

Huffman encoder (part 4, Huffman code construction example)

Given the Huffman tree, the following figure exemplifies how the Huffman code is constructed. Movement to the left corresponds to appending 0 to the code and movement to the right corresponds to appending 1 to the code (in binary!). Constants starting with `0b` are expressed in binary (the gcc compiler accepts them).

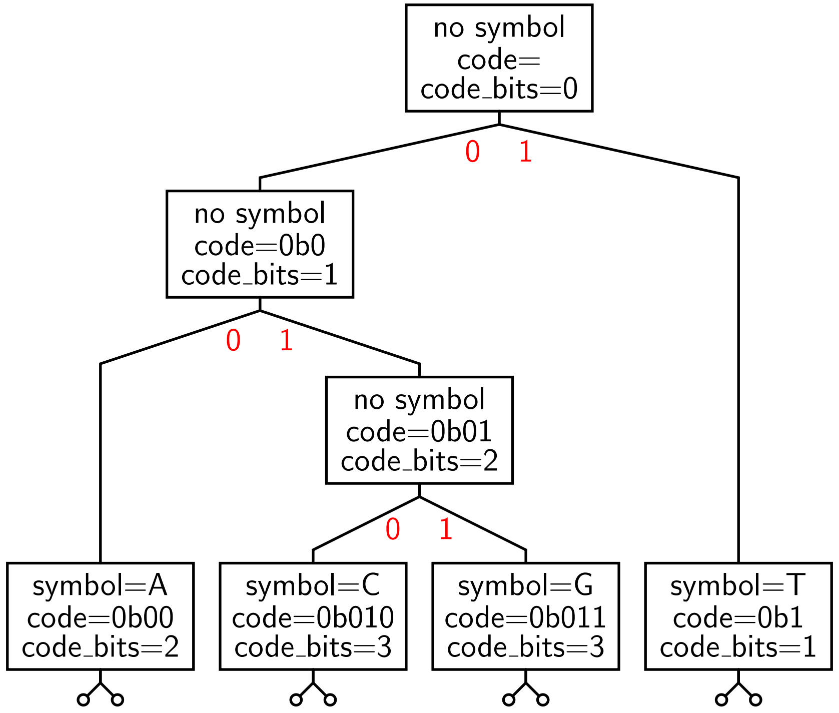


As expected, symbols with small counts get long codes, and symbols with large counts get short codes.

Huffman decoder (decoding example)

Given the Huffman tree, to decode a stream of bits one starts at the root of the tree and one moves left of right according to the bit received (0 for left and 1 for right). When a leaf is reached the corresponding symbol is written to the output stream and one returns to the root of the tree.

For example, for the following Huffman tree, the bit stream `000110101` is split into `00_011_010_1` (an underscore denotes a return to the root node), which is decoded as `AGCT`.



Final remarks

An incomplete implementation of the entire code can be found in the file `Huffman.c`; three dots (...) mark the places where code has to be completed.

It is necessary to encode the entire Huffman tree (the function `encode_Huffman_node` does this recursively). It has to match the code of the `decode_Huffman_node` function, which is provided in full.

In the written report it is necessary to explain (at the very least) how the functions `expand_binary_code`, `make_Huffman_tree`, `decode_Huffman_node`, and `encode` work, and why a special symbol was used to denote the end of encoded data.

In the written report it is necessary to compare the compression capabilities of the Huffman encoder with other compression programs (for example, `gzip`, `bzip2`, `lzma` and `xz`). Compress at least the file `SherlockHolmes.txt`. Also, experiment with other ways of splitting the input byte stream into symbols.

Finding all possibilities (part 1) — TP.10 —

Summary:

- Exhaustive search
- Depth-first search
- Breadth-first search
- Transversing a binary tree in depth-first order and in breadth-first order
- Backtracking
- Pruning
- An example: a chessboard problem

Recommended bibliography for this lecture:

- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.

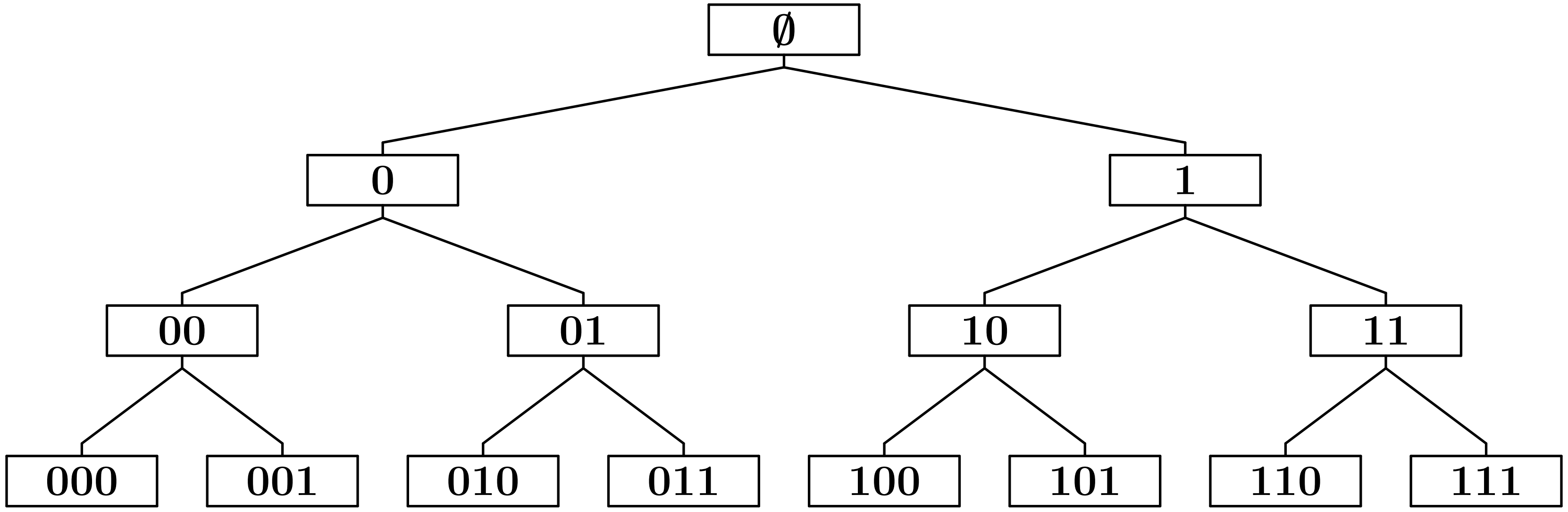
Exhaustive search

Many problems amenable to be solved by a computer require generating all possible configurations of a potential solution and either

- finding one that actually solves the problem or
- finding, among all that solve the problem, the "best one."

The potential solutions are usually constructed in an incremental way, piece by piece. When that happens it is possible to depict the generation process as building a so-called search tree; adding a node to the tree corresponds to adding another piece to the potential solution. For example, while attempting to solve a sudoku, adding a piece might be placing a specific number in a specific empty square.

We will illustrate the two main strategies of generation of a search tree using the following binary search tree:



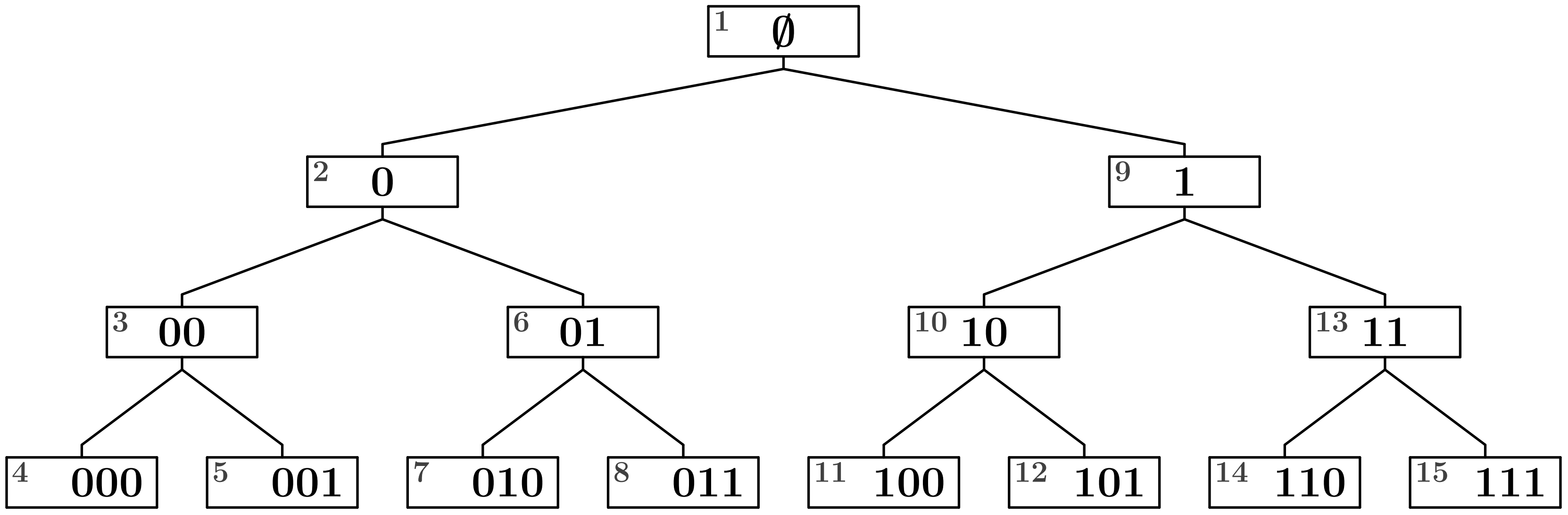
(In this case we may think that the root of the tree corresponds to an empty initial configuration and that at each level we decide to append either a 0 or a 1 to the configuration.)

Depth-first search

The way the search tree is constructed can make a big different in the execution time of the exhaustive search. For example, placing early on a piece in a position that invalidates any possibility of a solution is obviously a bad choice. Deciding how to build the search tree is usually one of the most critical tasks one faces when performing an exhaustive search.

In a **depth-first search** one tries to go as deep as possible as soon as possible. It is the most common strategy used in an exhaustive search, because it can be done quite easily (using recursion, for example), and does not require much memory to manage the search.

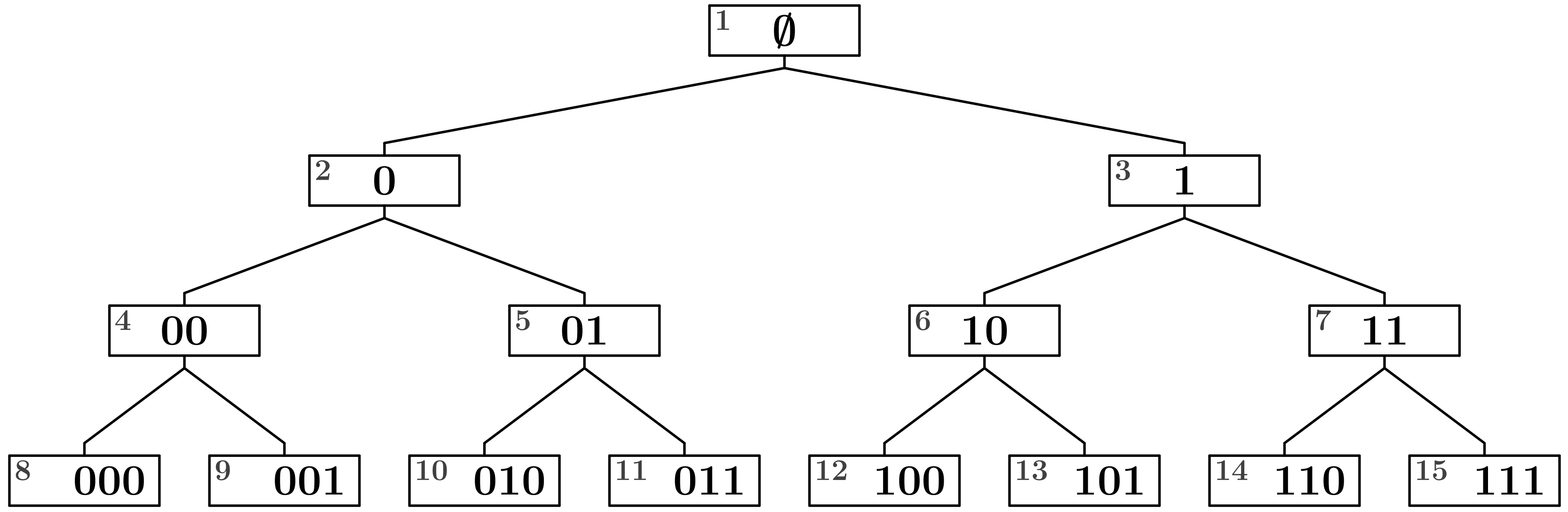
For the search tree we are using as example, in a depth-first search the nodes of the tree are visited in the following order (small numbers in gray on the upper left of each node):



Breadth-first search

In a **breadth-first search** one attempts to avoid going deeper for as long as possible, i.e., one explores the nodes of the search tree one level at a time. It is usually the strategy to use when one wants to first the shallowest solution and when the tree can be very deep. (The problem may ask, for example, for the smallest number of moves to win a game.) Its implementation is more difficult than that of a depth-first search, and it usually requires a significant amount of memory to store the search tree nodes that have not yet been expanded.

For the search tree we are using as example, in a breadth-first search the nodes of the tree are visited in the following order (small numbers in gray on the upper left of each node):



Transversing a binary tree in depth-first order and in breadth-first order

The following two non-recursive functions transverse a binary tree in depth-first and in breadth-first order. Note that they only differ in the data structure used to keep track of the nodes that have not yet been expanded! In both cases each node of the binary tree will be implemented (in C++) as follows:

```
typedef struct tree_node
{
  struct tree_node *left;   // pointer to the left branch (a sub-tree)
  struct tree_node *right;  // pointer to the right branch (a sub-tree)
  int data;                 // the data item (we use an int here, but it can be anything)
}
tree_node;
```

Here go the functions:

```
void depth_first(tree_node *root)
{
  stack<tree_node *> s;
  tree_node *n;

  s.push(root);
  while(s.isEmpty() == 0)
    if((n = s.pop()) != nullptr)
    {
      visit(n);
      s.push(n->left);
      s.push(n->right); // right is done first!
    }
}
```

```
void breadth_first(tree_node *root)
{
  queue<tree_node *> s;
  tree_node *n;

  s.enqueue(root);
  while(s.isEmpty() == 0)
    if((n = s.dequeue()) != nullptr)
    {
      visit(n);
      s.enqueue(n->left); // left is done first!
      s.enqueue(n->right);
    }
}
```

Observe their simplicity and elegance! The first one can also be easily implemented in a recursive way but that cannot be done easily for the second. The maximum queue size can be quite large!

Backtracking

When performing a depth-first search it is quite common to not be able to progress into a deeper search tree level. In that case one has to go retrace our steps and try the next possibility at a lower level (this is called **backtracking**). If the depth-first search is done via recursion, backtracking is done by returning from the function.

A backtracking depth-first search algorithm usually has the following structure:

1. [Initialize.] Initialize all relevant data structures and set the search level to 0. Go to Step 2.
2. [Try the current configuration.] If the current configuration is not acceptable, go to Step 3. Otherwise, if we have a solution to the problem, record it (and terminate the algorithm if the goal was to find one solution) and go to Step 3. Otherwise, update the relevant data structures to reflect the new data in the current configuration, increase the search level by 1, and go to Step 3.
3. [Advance.] Advance to the next configuration. If one exists, go back to Step 2. Otherwise, go to Step 4.
4. [Backtrack.] If we are at search level 0, terminate the algorithm. Otherwise, decrease the search level by 1 and undo the changes made to the relevant data structures in step 2. After that, go back to Step 3.

Depth-first search trees usually have a huge number of nodes, so any gain in speed in its implementation is welcome. Thus, although it is possible to code a depth-first search algorithm using a recursive function, for efficiency reasons it is usually preferable to do it all in the same function, using a custom made stack to store the search history. The author of this document goes one step further: he usually uses `goto` statements to jump between the different stages of the search, because it makes the code easier to understand! (The use of the more “respectable” control flow statements in algorithms of this nature usually complicates matters if one wants an efficient implementation.)

Pruning

In some problems it is possible to check if continuing to search for a solution by expanding the current search tree node (i.e., going deeper by checking the node's children) can lead to a solution, or to a better solution, of the problem. If it can be determined in advance that a (better) solution is not possible, one can immediately advance to the next configuration at the same search depth. This is called **pruning** the search tree.

If the pruning test is expensive, it may be advantageous to apply it only at some selected search tree depths, say, every tenth level.

The most successful depth-first search algorithms use pruning to reduce, sometimes dramatically, the number of search tree nodes that are visited by the program. In some difficult problems (i.e., very time consuming problems), one may even perform a non-exhaustive search by pruning the search tree with an **heuristic** (a test that keeps only promising search tree branches).

Pruning is usually performed in depth-first searches. It can also be done in a breadth-first search.

An example: a chessboard problem (part 1, problem statement and some code)

Consider the following "toy problem" (actually, not a toy problem, check problem C18 in "Unsolved Problems in Number Theory," Richard K. Guy, third edition, Springer, 2004): find the maximum number of unattacked squares when Q queens are placed on a $W \times H$ chessboard (squares with queens are considered to be attacked). This problem will be solved using depth-first search, backtracking and pruning.

We begin by declaring the data structures used to solve the problem. In general, this step requires considerable thought. In our case, a few arrays are enough (for legibility, the code uses `max_n_queen` instead of Q, `max_x` instead of W, and `max_y` instead of H):

```
#define true_max_x          10  // max_x cannot be larger than this
#define true_max_y          10  // max_y cannot be larger than this
#define true_max_n_queens   10  // max_n_queens cannot be larger than this

int board[true_max_x][true_max_y];
int queen_x[true_max_n_queens],queen_y[true_max_n_queens];
int max_x,max_y,max_n_queens,n_unattacked;

int best_queen_x[true_max_n_queens],best_queen_y[true_max_n_queens],best_n_unattacked;

void init_board(void)
{
  for(int x = 0;x < max_x;x++) for(int y = 0;y < max_y;y++) board[x][y] = 0; // unattacked
  n_unattacked = max_x * max_y;
  best_n_unattacked = 0;
}
```

The array `board[][]` will record the number of times each square is attacked by the queens, and the arrays `queen_x[]` and `queen_y[]` will record the coordinates of the queens. The variable `n_unattacked` will keep track of the number of unattacked squares. The `best_`... variables will keep track of the best solution (i.e., the one with the largest number of unattacked squares).

An example: a chessboard problem (part 2, support code)

Next, we need a way to register the effects of placing a queen in a square and a way of undoing those effects. The following two simple functions do those jobs:

```
void mark(int x,int y)
{
# define apply_mark(xx,yy)  do if(board[xx][yy]++ == 0) n_unattacked--; while(0)
  for(int i = 0 ;     i < max_x                 ; i++) apply_mark(i,y);
  for(int i = 0 ;     i < max_y                 ; i++) apply_mark(x,i);
  for(int i = 1 ; x + i < max_x && y + i < max_y ; i++) apply_mark(x + i,y + i);
  for(int i = 1 ; x - i >= 0    && y - i >= 0    ; i++) apply_mark(x - i,y - i);
  for(int i = 1 ; x + i < max_x && y - i >= 0    ; i++) apply_mark(x + i,y - i);
  for(int i = 1 ; x - i >= 0    && y + i < max_y ; i++) apply_mark(x - i,y + i);
# undef apply_mark
}

void unmark(int x,int y)
{
# define remove_mark(xx,yy) do if(--board[xx][yy] == 0) n_unattacked++; while(0)
  for(int i = 0 ;     i < max_x                 ; i++) remove_mark(i,y);
  for(int i = 0 ;     i < max_y                 ; i++) remove_mark(x,i);
  for(int i = 1 ; x + i < max_x && y + i < max_y ; i++) remove_mark(x + i,y + i);
  for(int i = 1 ; x - i >= 0    && y - i >= 0    ; i++) remove_mark(x - i,y - i);
  for(int i = 1 ; x + i < max_x && y - i >= 0    ; i++) remove_mark(x + i,y - i);
  for(int i = 1 ; x - i >= 0    && y + i < max_y ; i++) remove_mark(x - i,y + i);
# undef remove_mark
}
```

The first two `for` cycles deal with the squares attacked by a queen in the horizontal and vertical directions, the next two deal (somewhat inefficiently) with squares attacked in the 45 degree direction, and the last two deal (again, somewhat inefficiently) with squares attacked in the -45 degree direction.

An example: a chessboard problem (part 3, more support code)

Next, we need a way to record and present best solutions:

```c
void update_best_solution(void)
{
  if(n_unattacked > best_n_unattacked)
  {
    best_n_unattacked = n_unattacked;
    for(int i = 0;i < max_n_queens;i++)
    {
      best_queen_x[i] = queen_x[i];
      best_queen_y[i] = queen_y[i];
    }
  }
}

#include <stdio.h>

void show_best_solution(void)
{
  if(best_n_unattacked == 0)
    return;
  printf("%2d %2d %2d %2d ",max_x,max_y,max_n_queens,best_n_unattacked);
  for(int i = 0;i < max_n_queens;i++)
    printf(" (%d,%d)",best_queen_x[i],best_queen_y[i]);
  printf("\n");
  fflush(stdout);
}
```

An example: a chessboard problem (part 4, the depth-first code)

Now everything is in place to present the depth-first search code in all its glory:

```c
void solve(void)
{
               int x,y,n_queens;

               init_board();
               x = y = n_queens = 0;
place_queen:   mark(queen_x[n_queens] = x,queen_y[n_queens] = y);
               if(++n_queens == max_n_queens)
               {
                 update_best_solution();
                 goto backtrack;
               }
/* prune: */   if(n_unattacked <= best_n_unattacked)
               goto backtrack;
next_position: if(++y == max_y)
               {
                 y = 0;
                 if(++x == max_x)
                   goto backtrack;
               }
               goto place_queen;
backtrack:     if(--n_queens < 0)
                 goto done; // this goto can be easily avoided, the others are not that easy...
               unmark(x = queen_x[n_queens],y = queen_y[n_queens]);
               goto next_position;
done:          show_best_solution();
}
```

In this particular case the `goto` statements are used in a disciplined way and actually make the program cleaner (at least to the author). If you are not convinced, try doing it using more traditional control flow statements and without recursion (remember, recursion is in this case inefficient); `goto` statements are not always harmful!

An example: a chessboard problem (part 4, the main code)

It is now time to present the main program. It just requests solutions for all board sizes and number of queens up to the hardcoded limits defined at the beginning of the code:

```
int main(void)
{
  printf(" X  Y  Q  P  coords\n");
  printf("-- -- -- --  -----------------------------------------------------------\n");
  for(max_x = 1;max_x <= true_max_x;max_x++)
    for(max_y = max_x;max_y <= true_max_y;max_y++)
      for(max_n_queens = 1;max_n_queens <= true_max_n_queens;max_n_queens++)
        solve();
  printf("-- -- -- --  -----------------------------------------------------------\n");
  return 0;
}
```

[**Research problem:** What is the exact maximum number of unattacked squares when N queens are placed on an $N \times N$ chessboard for $N > 14$?]

Second written report work

— P.10 —

Summary:

- Empirical study of some sorting algorithms.

Finding all possibilities (part 2, two examples) — TP.11 —

Summary:

- sudoku solver (depth-first search example)
- klotski solver (breadth-first search example)

Remarks: The explanation of the two examples of the following slides follow a **bottom-up** approach: they start by the foundations (data types and low-level functions), and then they present functions that depend on the data types and functions that were previously explained, finishing with the `main()` function. The author is these slides almost always writes his programs in this way. **However**, his approach to the **planning phase** of the program is **top-down**. He usually spends a significant amount of time thinking about the data structures that will be used, and about how to subdivide the program into manageable parts. He only starts writing code when he has convinced himself that his solution will work and is reasonably efficient. While coding, when he has finished a module (a group of functions that address a particular part of the problem), we usually writes one of more test functions to make sure that that part of the program works as planed.
The complete code of these two examples can be found in the archive `P11.tgz`.

Highly recommended bibliography for this lecture:

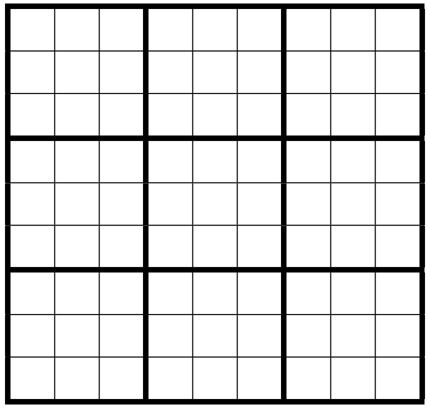
- **Dancing links**, Donald E. Knuth, 2000. [Although the dancing links method is not explained in these slides, reading this paper is definitely worth the time.]

Recommended bibliography for this lecture:

- **Programming Pearls**, Jon Bentley, second edition, Addison Wesley, 2000.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.
- **Algorithm Design**, Jon Kleinberg and Éva Tardos, Addison Wesley, 2006.

Sudoku solver (part 1, problem formulation)

In the standard **sudoku puzzle** a 9×9 array of cells in partitioned (independently) into 9 rows, into 9 columns, and into 9 smaller 3×3 arrays of cells (see figure on the right hand side). Given a set of clues (numbers already placed in the array), the objective of the sudoku puzzle is to fill the empty cells with the numbers $1, 2, \ldots, 9$ in such a way that



- each row contains only one occurrence of each number,
- each column also contains only one occurrence of each number, and
- each 3x3 smaller square also contains only one occurrence of each number.

Usually, the clues are selected so that the puzzle has only one solution (we can use that to our advantage!). In this lecture we are going to study in detail a program capable of **solving** sudoku puzzles. Is has the following characteristics:

- it can deduce cell numbers using some simple methods,
- when no deduction is possible, it can guess cell numbers (and backtrack if the guess was wrong),
- it reports the number of solutions found (0, 1, or more than 1), and
- it can be configured to use only some deduction methods, so it can be the starting point of another program capable of **generating** sudoku puzzles. (One starts with a complete array and then one removes numbers at random and one keeps checking if the puzzle can still be solved and if it still has only one solution.)

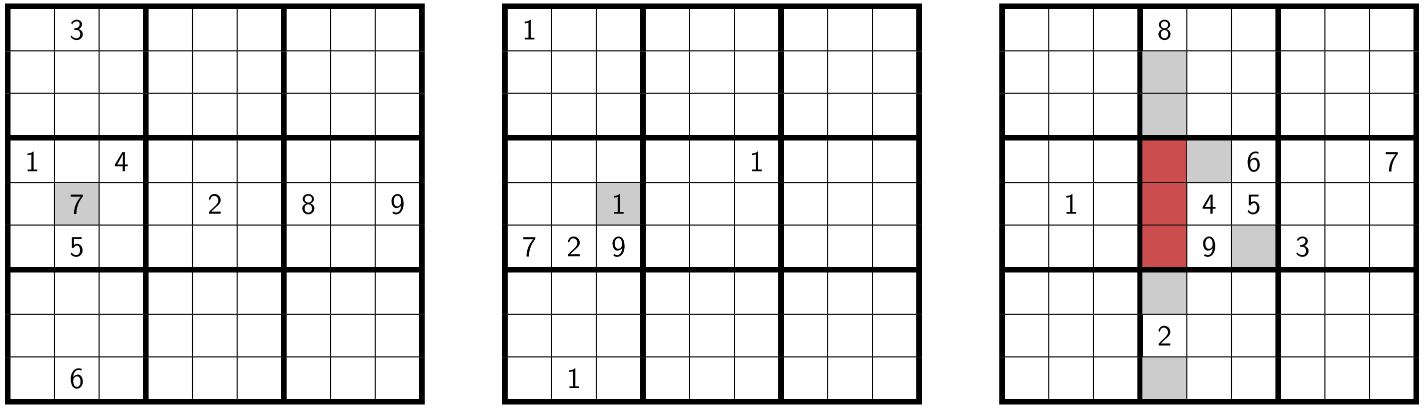
The program preforms a depth-first search because if a second guess is necessary it does it without trying first the other possible choices for the first guess.

Sudoku solver (part 2a, deduction methods)

The program will use the following deduction methods:

- **Method number 1: (elementary)** If only one number can be placed in a cell, place it. (Example on the left below.)
- **Method number 2: (elementary)** If in a region (row, column, or small 3×3 square) a number can only be placed in one cell, place it in that cell. (Example on the middle below.)
- **Method number 3: (intermediate/advanced)** If in a region there exist n cells in which only the same n numbers can be placed, then these numbers cannot be placed in the other cells of that region. Furthermore, if these cells also all belong to another region, then these numbers cannot be placed in the other cells of that other region. (Example on the right below.)

These methods should be reapplied until they do not produce any more changes in the puzzle state. The following examples illustrate how these deduction methods work. On the first two, the number inside the gray square is forced. On the last, the gray squares cannot contain the numbers in the red squares (the red squares must contain the numbers 1, 3, and 7, the other two cells of the central small square must contain the numbers 2 and 8).



Stop and think! How can these tasks be accomplished in an efficient way?

Sudoku solver (part 2b, data structures for the deduction methods)

Deduction method number 1 is implemented by keeping for each cell a list of the numbers that can still be placed there. Given that there are only 9 possible numbers for each cell, the list can be stored in a single integer (in the code this integer is called a `mask`), one bit per possible number. If bit number i of the mask is set (equal to 1) then the number $i + 1$ can be placed in the cell without violating the sudoku rules. To apply this deduction method it will be necessary to find out if a mask has only one bit set. This is accomplished by storing in the array `n_bits[]` the number of bits equal to 1 for all possible masks. If the mask has only one bit set, it will be necessary to find out that bit number. This is accomplished by storing in the array `lsb_number[]` the number of the least significant bit that is not zero (if the mask is all zeros, -1 is stored in that array).

Deduction method number 2 is implemented by providing a way to expand a mask, inserting 3 zero bits between the bits of the original mask (and three more on the left). For example, the 9-bit mask **100110110** when expanded becomes the 36-bit mask 000**1**000**0**000**0**000**1**000**1**000**0**000**1**000**1**000**0**. By adding up the expanded masks one is counting **in parallel** the number of times each digit appears in the masks. (Given that a digit can appear in all 9 masks, at least 4 bits are necessary to do this for each number.) In the program, the array `expand_mask[]` stores the expanded masks.

Deduction method number 3 uses the arrays described above and some bit manipulation tricks to do its job. In particular, code such as the following is used to identify and extract one at a time the bits set to one of a mask:

```
for(mask_copy = mask;((bit_number = lsb_number[mask_copy]) >= 0;mask_copy ^= 1 << bit_number)
{
  // do some stuff here (this is done n_bits[mask] times)
}
```

Sudoku solver (part 3, bit stuff)

The following code initializes the arrays `n_bits[]`, `lsb_number[]`, and `expand_mask[]` described in the previous slide. Note that there are $2^9 = 512$ possible masks and that the data type of the `expand_mask[]` array must be able to hold at least 36 bits.

```
static int n_bits[512];                      // number of bits set to 1
static int lsb_number[512];                  // number of the least significant bit that is set to 1
static long long expand_mask[512];           // transform each mask bit into four bits (insert three zeros)

static void init_mask_data(void)
{
  int m,i,j;

  for(m = 0;m < 512;m++)                     // for each mask ,,,
  {
    n_bits[m] = 0;                           // compute its number of bits set to one
    lsb_number[m] = -1;                      // compute also the bit number of its least significant bit set to one
    expand_mask[m] = 0ll;                    // and compute its expanded mask ...
    i = -1;
    for(j = m;j != 0;j >>= 1)
    {
      i++;
      if((j & 1) != 0)                       // is the i-th bit of m one?
      {                                      // yes!
        n_bits[m]++;                         // one more bit
        if(lsb_number[m] < 0)                // is this the first bit set to one?
          lsb_number[m] = i;                 // yes! set the least significant bit number
        expand_mask[m] |= 1ll << (4 * i);    // update the expanded mask
      }
    }
  }
}
```

Sudoku solver (part 4a, puzzle regions)

The 9×9 array of cells is partioned into rows, columns, and smaller 3×3 arrays of cells (called regions in the code). It will be necessary to known which cells belong to which regions and vice versa. That information is stored in the arrays `cell_regions[][]` and `region_cells[][]`. To apply deduction method number 3 it will also be necessary to discover if a group of cells belongs to more than one region. Sufficient information to do this is stored in the array `region_intersections[][]`.

The following code initializes the arrays mentioned above. Note that there are 81 cells and 27 regions, that each cell belongs to 3 regions, that each region has 9 cells, and that there are 54 relevant region intersections.

```c
#include <assert.h>

static int cell_regions[81][3];          // the numbers of the regions each cell belongs to
static int region_cells[27][9];          // the cells that are part of each region
static int region_intersections[54][5];  // intersection data

void init_regions(void)
{
  int c,r,n,x,y,r1,r2,i,j,C[9],ni;

  for(c = 0;c < 81;c++)                    // for each cell ...
  {
    x = c % 9;                             // cell x coordinate
    y = c / 9;                             // cell y coordinate
    cell_regions[c][0] = x;                // vertical region formed by the same x value
    cell_regions[c][1] = 9 + y;            // horizontal region formed by the same y value
    x /= 3;                                // square region x coordinate
    y /= 3;                                // square region y coordinate
    cell_regions[c][2] = 18 + 3 * y + x;   // square region
  }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
  for(r = 0;r < 27;r++)                        // for each region ...
  {
    for(n = c = 0;c < 81;c++)                  // find the cells that belong to this region ...
      if(cell_regions[c][0] == r || cell_regions[c][1] == r || cell_regions[c][2] == r)
        region_cells[r][n++] = c;
    assert(n == 9);                            // must be 9 for each region
  }
  ni = 0;                                      // count the number of valid region intersections
  for(r1 = 18;r1 < 27;r1++)                    // for all square regions ...
    for(r2 = 0;r2 < 18;r2++)                   // for all non-square regions ...
    {
      n = i = j = 0;                           // compute intersection of the two regions
      while(i < 9 && j < 9)                    // find and count the number of common elements of two sorted arrays
        if(region_cells[r1][i] == region_cells[r2][j])
        {
          C[n++] = region_cells[r1][i++];
          j++;
        }
        else if(region_cells[r1][i] < region_cells[r2][j])
          i++;
        else
          j++;
      if(n == 3)                               // if the intersection has 3 cells ...
      {
        region_intersections[ni][0] = C[0];  // record first cell number
        region_intersections[ni][1] = C[1];  // record second cell number
        region_intersections[ni][2] = C[2];  // record third cell number
        region_intersections[ni][3] = r1;    // record first region number
        region_intersections[ni][4] = r2;    // record second region number
        ni++;
      }
    }
  assert(ni == 54);                            // must be 54
}
```

Sudoku solver (part 4b, find a puzzle region)

To apply deduction method number 3 it is necessary to find if a given group of cells belong to two regions, and if so it is necessary to find the number of one of those regions given the number of the other. The following code does precisely that.

```
static int find_region(int r,int nc,int *c)
{
  int i,j,k,n;

  if(nc < 2 || nc > 3)    // only 2 or 3 cells are possible
    return -1;            // no valid region
  for(i = 0;i < 54;i++)   // for each possible intersection
  {
    n = j = k = 0;        // count the number of common elements of two sorted arrays
    while(j < 3 && k < nc)
      if(region_intersections[i][j] == c[k])
      {
        n++;
        j++;
        k++;
      }
      else if(region_intersections[i][j] < c[k])
        j++;
      else
        k++;
    assert(n != nc || region_intersections[i][3] == r || region_intersections[i][4] == r);
    if(n == nc)           // found it! return the number of the other region
      return (region_intersections[i][3] == r) ? region_intersections[i][4] : region_intersections[i][3];
  }
  return -1;              // no valid region
}
```

[**Homework:** Study how the number of common elements of two sorted arrays is counted.]

Sudoku solver (part 5a, puzzle state initialization)

The state of the sudoku puzzle is essentially the state of each one of its cells. For each cell it was decided to store the number it holds (`digit` in the code below), to store a mask (`mask`) of the possible numbers it can hold, to record the method number (`method`) used to place the number, and to use a flag (`frozen`) to simplify the implementation of the deduction method number 3. The following function initializes the puzzle state.

```
typedef struct
{
  int n_known_digits;            // number of digits already known
  struct
  {
    int digit;                   // number placed in this cell (-1 means none, 0..8 means 1..9)
    int mask;                    // bit-mask of the digit values that can be placed in this cell
    int method;                  // method number used to place a digit in this cell (-1 means none)
    int frozen;                  // if non-zero, do not make changes to the mask
  }
  cells[81];                     // the state of each of the 9x9 cells
}
state;

void init_state(state *s)
{
  int c;

  s->n_known_digits = 0;         // no known digits
  for(c = 0;c < 81;c++)          // for each cell ...
  {
    s->cells[c].digit = -1;      // no digit
    s->cells[c].mask = 0x1FF;    // all digits are possible
    s->cells[c].method = -1;     // no placement method
    s->cells[c].frozen = 0;      // not frozen
  }
}
```

Sudoku solver (part 5b, display puzzle state)

The following function outputs the current puzzle state.

```c
#include <stdio.h>

static void show_state(state *s)
{
  int c,x,y,t0,t1,t2,t4,tx;

  t0 = t1 = t2 = t4 = tx = 0;
  for(c = 0;c < 81;c++)
  {
    x = c % 9;
    y = c / 9;
    if(x == 0)
      printf("  ");
    if(s->cells[c].digit < 0)
      printf("[?]");
    else
      printf("[%d]",s->cells[c].digit + 1);
    switch(s->cells[c].method)
    {
      case  0: printf("F"); t0++; break;
      case  1: printf("1"); t1++; break;
      case  2: printf("2"); t2++; break;
      case  4: printf("g"); t4++; break;
      default: printf("?"); tx++; break;
    }
    if(x < 8)
      printf((x % 3 == 2) ? "  " : " ");
    else
      printf((y % 3 == 2) ? "\n\n" : "\n");
  }
  printf("  --- F:%d 1:%d 2:%d g:%d ?:%d\n",t0,t1,t2,t4,tx);
}
```

Sudoku solver (part 6, place a number)

The following function places a number in a given cell. It returns 0 if no solution is possible (that happens when a mask becomes 0), and returns 1 otherwise.

```c
static int place_digit(state *s,int cell,int digit,int method)
{
  int ir,r,ic,c,m;

  assert(0 <= cell && cell < 81 && digit >= 0 && digit <= 8);
  assert(s->cells[cell].digit == -1 && (s->cells[cell].mask & (1 << digit)) != 0);
  s->n_known_digits++;
  s->cells[cell].digit = digit;
  s->cells[cell].mask = 0;
  s->cells[cell].method = method;
  s->cells[cell].frozen = 1;
  m = 1 << digit;                               // digit mask
  for(ir = 0;ir < 3;ir++)                       // for each of the three cell regions ...
  {
    r = cell_regions[cell][ir];
    for(ic = 0;ic < 9;ic++)                     // for each of the 9 cells belonging to the region ...
    {
      c = region_cells[r][ic];
      if((s->cells[c].mask & m) != 0)           // if the digit can be place here ...
        if((s->cells[c].mask &= ~m) == 0)       // make that impossible, and say no solution is
          return 0;                             //   possible if the new mask is 0
    }
  }
  return 1;                                     // say that a solution is possible
}
```

Sudoku solver (part 7, parse a sudoku puzzle)

The following function extract the clues of a sudoku puzzle from an initialization string and places them on an initially empty puzzle state. It returns 0 if the initialization string is not valid, or if no solution is possible, and returns 1 otherwise.

```
static int init_sudoku(state *s,char *data)
{
  int c;

  init_state(s);
  for(c = 0;c < 81 && *data != '\0';data++)
    if(*data == ' ' || *data == '.' || *data == '?')
      c++;
    else if(*data >= '1' && *data <= '9' && place_digit(s,c++,(int)(*data) - '1',0) == 0)
      return 0;
  return (*data == '\0' && c == 81) ? 1 : 0;
}
```

Sudoku solver (part 8, deduction method number 1)

Everything is now in place to start presenting code to actually solve the sudoku puzzle. The functions that implement the number deduction methods return 0 if they did change the state of the puzzle and if because of that a solution is no longer possible, return 1 if they did not change the state of the puzzle (so a solution is still possible), and return 2 if they did change the state of the puzzle and if a solution is still possible.

The following code deals with cells that can have only one possible value.

```
static int do_method_1(state *s)
{
  int rv,c,d;

  rv = 1;
  for(c = 0;c < 81;c++)                     // for each cell ...
    if(n_bits[s->cells[c].mask] == 1)       // only one possibe digit?
    {                                       // yes!
      d = lsb_number[s->cells[c].mask];     // find digit
      if(place_digit(s,c,d,1) == 0)         // place it
        return 0;                           // say no solution is possible if place_digit() says so
      rv = 2;                               // say at least one digit was placed
    }
  return rv;
}
```

Sudoku solver (part 9, deduction method number 2)

The following code deals will regions in which a digit can only be placed in one cell.

```
static int do_method_2(state *s)
{
  int rv,r,ic,c,d;
  long long sum;

  rv = 1;
  for(r = 0;r < 27;r++)                              // for each region ...
  {
    sum = 0ll;
    for(ic = 0;ic < 9;ic++)                          // for each cell ...
    {
      c = region_cells[r][ic];
      sum += expand_mask[s->cells[c].mask];
    }
    for(d = 0;d < 9;d++)                             // for each digit ...
      if(((sum >> (4 * d)) & 15ll) == 1ull)          // if the digit appears in only one mask ...
        for(ic = 0;ic < 9;ic++)                      // find cell! For each cell ...
        {
          c = region_cells[r][ic];
          if(((s->cells[c].mask >> d) & 1) != 0)     // is this cell the one?
          {                                          // yes!
            if(place_digit(s,c,d,2) == 0)            // place the digit in the cell
              return 0;                              // say no solution is possible if place_digit() says so
            rv = 2;                                  // say at least one digit was placed
          }
        }
  }
  return rv;
}
```

Sudoku solver (part 10a, deduction method number 3, trim masks)

The following function adjusts the masks of all unfrozen cells in a region so that they no longer can hold a given set of numbers. It returns 0 is a solution becomes impossible, returns 1 if no change was made to the puzzle state, and returns 2 otherwise.

```c
static int trim_masks(state *s,int r,int mask)
{ // 0 -> no solution, 1 -> no change, 2 -> change
  int rv,ic,c,new_mask;

  rv = 1;
  if(r >= 0 && r <= 26)                         // if the region number is valid ...
    for(ic = 0;ic < 9;ic++)                     // for each cell ...
    {
      c = region_cells[r][ic];
      if(s->cells[c].frozen == 0)               // if the mask can be modified ...
      {
        new_mask = s->cells[c].mask & ~mask;    // trim mask
        if(new_mask == 0)
          return 0;                             // say no solution is possible
        if(new_mask != s->cells[c].mask)        // if the new mask is different from the old one ...
        {
          s->cells[c].mask = new_mask;          // update mask
          rv = 2;                               // and say that at least one mask changed
        }
      }
    }
  return rv;
}
```

Sudoku solver (part 10c, deduction method number 3)

The following function implements the deduction method number 3. For each region it first records all cells without numbers (that is the code presented in this page), and then for each possible subset of `nc` cells it checks if their masks force `nc` numbers to be placed in them; if so these numbers are removed from the masks of the other cells of the region and of a possible "intersecting" region (that is the code presented on the next page). For example, the last two of the following five masks $m_1 = 000000101$, $m_2 = 001000100$, $m_3 = 001000001$, $m_4 = 010000111$, and $m_5 = 011000011$ can be trimmed to $m_4 = m_5 = 010000010$ because the mask $m_1 \text{ or } m_2 \text{ or } m_3 = 001000101$ has 3 bits set to 1 and so three numbers (1, 3, and 7) must be placed in the cells corresponding to these masks. Here is its code:

```
static int do_method_3(state *s)
{
  int rv,r,ic,c,nm,M[9],C[9],nc,CC[9],i,j,cm,b,rr;

  rv = 1;
  for(r = 0;r < 27;r++)                   // for each region ...
  {
    nm = 0;                               // count the number of cells and masks that are relevant
    for(ic = 0;ic < 9;ic++)               // for each cell ...
    {
      c = region_cells[r][ic];
      if(s->cells[c].digit < 0)           // if it does not yet have a digit ...
      {
        C[nm] = c;                        // record the cell
        M[nm] = s->cells[c].mask;         // record the mask
        nm++;
      }
    }
```

— the code continues on the next slide —

— continuation of the code of the previous slide —

```
    for(i = 3;i < (1 << nm);i++)           // for each possible subset with at least 2 elements
      if(n_bits[i] >= 2)
      {
        nc = 0;                            // count the cells of the subset and combine (logical or) their masks
        for(cm = 0,j = i;(b = lsb_number[j]) >= 0;j ^= 1 << b)
        {
          CC[nc++] = C[b];                 // record the cell
          cm |= M[b];                      // combine masks
        }
        assert(n_bits[i] == nc);           // not needed, but we check this anyway
        if(nc == n_bits[cm])               // if the number of cells is equal to the number of possible digits
        {
          for(j = 0;j < nc;j++)
            s->cells[CC[j]].frozen = 1;    // freeze the cells of the subset
          switch(trim_masks(s,r,cm))       // trim all unfrozen cells of the region
          {
            case 0: return 0;              // say no solution is possible if trim_masks() says so
            case 2: rv = 2;                // say that at least one mask changed if trim_masks() says so
          }
          rr = find_region(r,nc,CC);       // find another region (if any) that has the frozen cell of the current one
          if(rr >= 0)                      // if it exists ...
            switch(trim_masks(s,rr,cm))    // trim all unfrozen cells of that region
            {
              case 0: return 0;            // say no solution is possible if trim_masks() says so
              case 2: rv = 2;              // say that at least one mask changed if trim_masks() says so
            }
          for(j = 0;j < nc;j++)
            s->cells[CC[j]].frozen = 0;    // unfreeze the cells of the subset
        }
      }
  }
  return rv;
}
```

Sudoku solver (part 11, solve sudoku)

It is now time to present the function that solves a sudoku puzzle. It returns -1 when a solution was not found, returns 0 when the puzzle does not have a solution, returns 1 when it has only one solution, and returns 2 when it has two or more solutions. It records one solution in the `sol` argument. While attempting to find a solution it does not use methods with numbers larger than `max_method` (guessing a number is method number 4). Here is its code:

```
static int solve_sudoku(state *s,state *sol,int max_method)
{
  int rv1,rv2,rv3,c,bc,nb,m,d,ns;
  state ss;

  rv1 = rv2 = rv3 = 2;
  while(s->n_known_digits < 81 && (rv1 > 1 || rv2 > 1 || rv3 > 1))  // deduce digits
  {
    rv1 = rv2 = rv3 = 1;
    if(max_method >= 1 && (rv1 = do_method_1(s)) == 0) return 0;
    if(max_method >= 2 && (rv2 = do_method_2(s)) == 0) return 0;
    if(max_method >= 3 && (rv3 = do_method_3(s)) == 0) return 0;
  }
  if(s->n_known_digits == 81) { *sol = *s; return 1; }              // solved!
  if(max_method < 4) return -1;                                     // not solved!
  for(nb = 10,bc = 0,c = 0;c < 81;c++)                              // find the "best" cell
    if(s->cells[c].digit < 0 && n_bits[s->cells[c].mask] < nb)
      nb = n_bits[s->cells[bc = c].mask];
  for(ns = 0,m = s->cells[bc].mask;ns < 2 && m != 0;m ^= 1 << d)    // try all digits
  {
    d = lsb_number[m];                                              // digit to try
    ss = *s;                                                        // clone the state
    if(place_digit(&ss,bc,d,4) != 0)                                // place digit
      ns += solve_sudoku(&ss,sol,max_method);                       // recurse if the digit placement allows solutions
  }
  return (ns > 1) ? 2 : ns;
}
```

Sudoku solver (part 12a, main function)

To exercise the sudoku solver the main function uses it to solve a few interesting puzzles:

```c
int main(void)
{
  static struct
  {
    char *data;
    char *author;
  }
  puzzles[] =
  {
    { "7.19..3.6.....6.71...1..29..3....6.99.4...8.76.8....3..19..5...57.4.....4.3..91.8","solo (trivial)"            },
    { "..462.5.....75...6.....3...........5..3284.9.6....1.8...9.4..623............7.4..","solo (unreasonable)"       },
    { "8..........36......7..9.2...5...7.......457.....1...3...1....68..85...1..9....4..","Arto Inkala"               },
    { "6....894.9....61...7..4....2..61..........2...89..2.......6...5.......3.8....16..","David Filmer"              },
    { "...8.1..........435............7.8........1...2..3....6......75..34........2..6..","McGuire, Tugemann, Civario" }
  };
  state s,sol;
  int i,ns;

  init_mask_data();
  init_regions();
  for(i = 0;i < (int)(sizeof(puzzles) / sizeof(puzzles[0]));i++)
    if(init_sudoku(&s,puzzles[i].data) != 0)
    {
      ns = solve_sudoku(&s,&sol,9);
      printf("Suduku by %s\n\n",puzzles[i].author);
      if(ns > 0)
        show_state(&sol);
      printf("  --- number of solutions: %d%s\n\n",ns,(ns == 2) ? " or more" : "");
    }
  return 0;
}
```

Sudoku solver (part 12b, main function output)

It is interesting to observe that one of the possible sudoku puzzles with the least number of clues (17) can be solved by entirely elementary methods:

```
Suduku by McGuire, Tugemann, Civario

  [2]1 [3]1 [7]1  [8]F [4]1 [1]F  [5]1 [6]2 [9]2
  [1]2 [8]2 [6]2  [7]1 [9]1 [5]2  [2]1 [4]F [3]F
  [5]F [9]1 [4]1  [3]2 [2]1 [6]1  [7]1 [1]2 [8]2
  [3]2 [1]1 [5]2  [6]2 [7]F [4]2  [8]F [9]2 [2]1
  [4]2 [6]1 [9]2  [5]1 [8]2 [2]2  [1]F [3]2 [7]1
  [7]1 [2]F [8]1  [1]1 [3]F [9]1  [4]1 [5]1 [6]1
  [6]F [4]1 [2]2  [9]2 [1]2 [8]2  [3]2 [7]F [5]F
  [8]2 [5]1 [3]F  [4]F [6]1 [7]2  [9]1 [2]2 [1]2
  [9]1 [7]1 [1]1  [2]F [5]1 [3]2  [6]F [8]1 [4]2

  --- F:17 1:34 2:30 g:0 ?:0
  --- number of solutions: 1
```

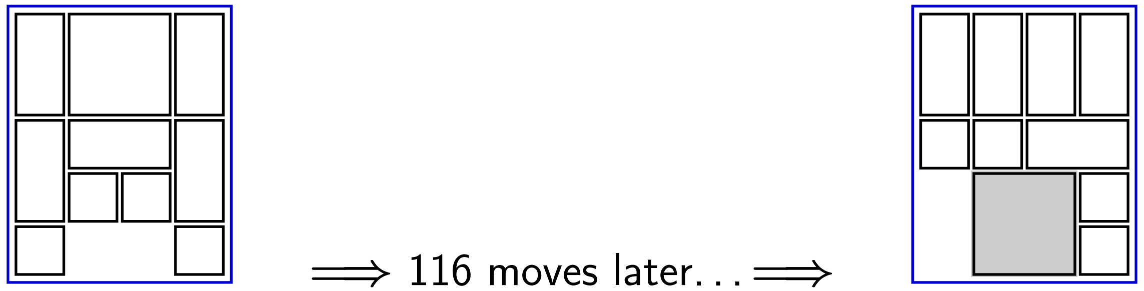
(The clues have a gray background.)

2	3	7	**8**	4	**1**	5	6	9
1	8	6	7	9	5	2	**4**	**3**
5	9	4	3	2	6	7	1	8
3	1	5	6	**7**	4	**8**	9	2
4	6	9	5	8	2	**1**	3	7
7	**2**	8	1	**3**	9	4	5	6
6	4	2	9	1	8	3	**7**	**5**
8	5	**3**	**4**	6	7	9	2	1
9	7	1	**2**	5	3	**6**	8	4

Using only the simple deduction methods used by the program, hard sudoku puzzles (Arto Inkala, David Filmer) require several guesses. A more sophisticated program would use more (and more complex) deduction methods.

Klotski solver (part 1, problem formulation)

The so-called **klotski puzzle** is a sliding block puzzle. Given an initial set of blocks placed inside an enclosure, the player has to move the blocks one at a time without lifting them (i.e., sliding them) with the goal of reaching a certain final configuration. In most cases the final configuration consists of placing one of the pieces, usually the largest one, in a certain position. The following figure presents one very popular puzzle of this kind.



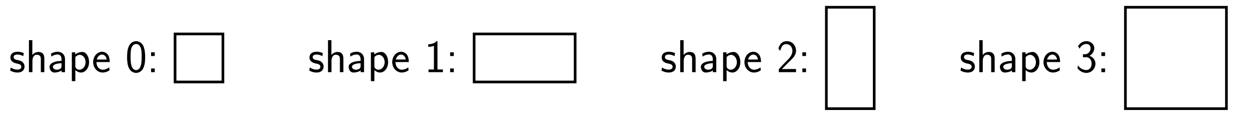
The following slides describe a program capable of solving small puzzles of this kind. Is has the following characteristics:

- the shape of the puzzle pieces (1×1, 2×1, 1×2 and 2×2) is hardwired in the code
- the puzzle enclosure shape is rectangular, with a width and a height that are at most 8,
- it reports a solution with the smallest number of moves (one move is a piece movement to an adjacent location).

The program preforms a breadth-first search because we are interested in a solution with the smallest number of moves. It starts with the initial configuration (generation 0) and produces all possible configurations obtained from it by performing one movement of one piece (this gives rise to generation 1). From that point onwards, generation n is obtained by considering all possible piece movements (to an adjacent location) for each of the generation $n-1$ configurations; the generation n configurations are the **new** ones (those not observed before). The search terminates either when a solution is found or when there are no more new configurations.

Klotski solver (part 2, fundamental data types, defines, and some global variables)

To simplify things, it was decided to make available only the following four piece shapes:



The shape number can be encoded using only two bits. It was also decided to restrict coordinates to the values $0, 1, \ldots, 7$, so that each of the two coordinates can be encoded in only 3 bits. The following declarations of data types and some macro definitions reflect these choices. (Some global variables are also declared.)

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

typedef unsigned int  u32;      // for hash table indices
typedef unsigned char u08;      // piece info: shape in bits 7..6, x coordinate in bits 5..3, y coordinate in bits 2..0

#define info(s,x,y)  (u08)(((s) << 6) | ((x) << 3) | ((y) << 0))
#define s_info(p)    (((int)(p) >> 6) & 3)
#define x_info(p)    (((int)(p) >> 3) & 7)
#define y_info(p)    (((int)(p) >> 0) & 7)

#define hash_table_size  50000u // the size of the hash table
#define max_width             8 // maximum puzzle width (HARD CODED, do not change)
#define max_height            8 // maximum puzzle height (HARD CODED, do not change)
#define max_n_pieces         16 // maximum number of pieces a puzzle can have

static u32 width;               // actual puzzle width
static u32 height;              // actual puzzle height
static u32 n_pieces;            // actual number of pieces of the puzzle
static u08 goal;                // puzzle goal (piece shape and position)
```

Klotski solver (part 3a, hash table node data type)

The puzzle configurations will be stored in a hash table. Assuming a sufficiently large hash table size, this makes checking if a configuration is new or not an $O(1)$ operation (on average).

Each hash table node stores one puzzle configuration. It has the following fields:

- the `next_hash_node` field keeps a pointer to a possible other hash table node with an equal key (our hash table will use chaining),
- the `pieces[]` field keeps the compacted information of each piece shape and coordinates,
- the `parent` field keeps a pointer to the configuration that gave rise to this one, and
- the `next_configuration` field keeps a pointer to the next hash table node of the singly-linked link that implements a queue (for the breadth-first search).

The data in the `pieces[]` field is stored in sorted order, so that there exists only one representation for a given puzzle configuration.

Given, the above description, the hash table node data type is as follows:

```
typedef struct hash_node
{
  struct hash_node *next_hash_node;     // pointer to the next hash table node with the same key
  struct hash_node *parent;             // pointer to the configuration that generated this one
  struct hash_node *next_configuration; // pointer to the next configuration to try (queue)
  u08 pieces[max_n_pieces];             // the actual configuration data
}
hash_node;
```

[**Question:** Which field is "the key"? Which field, or fields, is "the value"?

Klotski solver (part 3b, more global variables and allocation of a hash table node)

The following four variables keep all necessary information about the hash table and about the breadth-first search queue:

```c
static hash_node *hash_table[hash_table_size];         // the hash table
static hash_node *free_hash_nodes = NULL;              // linked list of tree hash table nodes
static hash_node *first_untried_configuration = NULL;  // head of the queue linked list
static hash_node *last_untried_configuration = NULL;   // tail of the queue linked list
```

The following function is used to allocate a new hash table node. To reduce memory allocation overheads (in both time and space), it allocates nodes 1000 at a time and it manages itself the free nodes (they are kept in a linked list).

```c
static hash_node *allocate_hash_node(void)
{
  hash_node *hn;
  int i;

  if(free_hash_nodes == NULL)
  {
    free_hash_nodes = (hash_node *)malloc((size_t)1000 * sizeof(hash_node));
    for(i = 0;i < 999;i++)
      free_hash_nodes[i].next_hash_node = &free_hash_nodes[i + 1];
    free_hash_nodes[i].next_hash_node = NULL;
  }
  hn = free_hash_nodes;
  free_hash_nodes = free_hash_nodes->next_hash_node;
  return hn;
}
```

Klotski solver (part 3c, hash table initialization and hash function)

The following function initializes the hash table array and the breadth-first search queue.

```c
void init_hash_table(void)
{
  u32 i;

  for(i = 0u;i < hash_table_size;i++)
    hash_table[i] = NULL;
  free_hash_nodes = NULL;
  first_untried_configuration = NULL;
  last_untried_configuration = NULL;
}
```

The next function computes the hash function of a given puzzle configuration (in canonical form, i.e., it is assumed that the `pieces[]` array is already sorted in increasing order).

```c
static u32 hash_function(const u08 *pieces)
{
  static u32 table[256];
  u32 crc,i,j;

  if(table[1] == 0u) // do we need to initialize the table[] array?
    for(i = 0u;i < 256u;i++)
      for(table[i] = i,j = 0u;j < 8u;j++)
        if(table[i] & 1u)
          table[i] = (table[i] >> 1) ^ 0xAED00022u; // "magic" constant
        else
          table[i] >>= 1;
  crc = 0xAED02016u; // initial value (chosen arbitrarily)
  for(i = 0u;i < (u32)n_pieces;i++)
    crc = (crc >> 8) ^ table[crc & 0xFFu] ^ ((u32)pieces[i] << 24);
  return crc % hash_table_size;
}
```

Klotski solver (part 3d, insert a configuration)

Now comes one of the most important functions: it checks if the configuration it is given is new or not, and if it is it inserts it in the hash table and in the breadth-first search queue.

```
static int insert_configuration(u08 *pieces,hash_node *parent)
{
  hash_node *hn; int i,j; u32 idx;

  // sort the pieces[] array using insertion sort (canonical representation!)
  for(i = 1;i < n_pieces;i++)
  {
    u08 tmp = pieces[i];
    for(j = i;j > 0 && tmp < pieces[j - 1];j--)
      pieces[j] = pieces[j - 1];
    pieces[j] = tmp;
  }
  // check if this is a new configuration
  idx = hash_function(pieces);
  for(hn = hash_table[idx];hn != NULL && memcmp(pieces,hn->pieces,n_pieces) != 0;hn = hn->next_hash_node)
    ;
  if(hn != NULL) return 0; // this configuration is already in the hash table, do nothing
  // it is a new configuration, insert it in the hash table and in the breadth-first search queue
  hn = allocate_hash_node();
  hn->next_hash_node = hash_table[idx];
  hash_table[idx] = hn;
  hn->parent = parent;
  if(first_untried_configuration == NULL)
    first_untried_configuration = last_untried_configuration = hn;
  else
    last_untried_configuration = last_untried_configuration->next_configuration = hn;
  hn->next_configuration = NULL;
  memcpy(hn->pieces,pieces,n_pieces);
  return 1;
}
```

Klotski solver (part 4a, piece map)

It is now time to arrange a way to check if a piece can be moved to an adjacent square in a relatively efficent way. This is going to be done by constructing a map of the puzzle.

```c
static char map[max_width + 2][max_height + 2];
```

The border of the map will be marked with a plus sign ('+', chosen more or less arbitrarily), an empty square will be marked with a space (' ', also chosen more or less arbitrarily), and the square, or squares, where each piece lies will be marked with the piece's number.

```c
static void init_map(u08 *pieces)
{
  int i,x,y,s;

  // initialize map (with a border)
  for(x = 0;x < width + 2;x++)
    for(y = 0;y < height + 2;y++)
      map[x][y] = (x == 0 || x == width + 1 || y == 0 || y == height + 1) ? '+' : ' ';
  // put the pieces on the map
  for(i = 0;i < n_pieces;i++)
  {
    s = s_info(pieces[i]);
    x = x_info(pieces[i]) + 1;
    y = y_info(pieces[i]) + 1;
    switch(s)
    {
      case 0: map[x][y] =                                                  (char)i; break;
      case 1: map[x][y] =                  map[x + 1][y] =                 (char)i; break;
      case 2: map[x][y] = map[x][y + 1] =                                  (char)i; break;
      case 3: map[x][y] = map[x][y + 1] = map[x + 1][y] = map[x + 1][y + 1] = (char)i; break;
    }
  }
}
```

Klotski solver (part 4b, piece map output)

The following function outputs a puzzle configuration. Each piece is represented by a lower case letter. The output could be beautified but that is not worth the trouble.

```c
static void print_map(void)
{
  int x,y;

  for(y = height;y >= 1;y--)
  {
    for(x = 1;x <= width;x++)
      putchar((map[x][y] < n_pieces) ? 'a' + map[x][y] : map[x][y]);
    putchar('\n');
  }
  putchar('\n');
}
```

Klotski solver (part 4c, attempt to move a piece)

The following function receives the index (`i`) of a piece and a movement displacement (`dx` and `dy`) and attempts to move that piece to its new position. If the new configuration solves the puzzle it returns 1. Otherwise it return 0.

```c
static int try_move(u08 *pieces,int i,int dx,int dy,hash_node *parent)
{
  u08 new_pieces[max_n_pieces];
  int x,y,s,j;

  s = s_info(pieces[i]);
  x = x_info(pieces[i]) + 1 + dx;
  y = y_info(pieces[i]) + 1 + dy;
  // can we move in this direction?
  if(                      map[x][y]         != (char)i && map[x][y]         != ' ') return 0;
  if((s == 2 || s == 3) && map[x][y + 1]     != (char)i && map[x][y + 1]     != ' ') return 0;
  if((s == 1 || s == 3) && map[x + 1][y]     != (char)i && map[x + 1][y]     != ' ') return 0;
  if(s == 3             && map[x + 1][y + 1] != (char)i && map[x + 1][y + 1] != ' ') return 0;
  // yes we can!
  for(j = 0;j < n_pieces;j++)
    new_pieces[j] = pieces[j];
  new_pieces[i] = info(s,x - 1,y - 1);
  if(insert_configuration(new_pieces,parent) == 0)
    return 0;
  for(j = 0;j < n_pieces && new_pieces[j] != goal;j++)
    ;
  return (j < n_pieces) ? 1 : 0;
}
```

Note that the functions `init_map()` and `try_move()` are the ones where the shape of the pieces is hard-coded. It is not difficult to lift that restriction. As stated before, that was not done to simplify the code (and to make it shorter).

Klotski solver (part 5, solve the puzzle)

It is now time to present the function that actually solves the puzzle. It performs a breadth-first search by removing configurations from the queue until the solution is found or until there are no more configurations to consider. When this function returns, the variable `last_untried_configuration` points to the first solution; if it is `NULL` no solution exists.

```
static void solve_puzzle(u08 *initial_configuration)
{
  hash_node *hn;
  int i;

  assert(width <= max_width && height <= max_height && n_pieces <= max_n_pieces);
  init_hash_table();
  insert_configuration(initial_configuration,NULL);
  // do a breadth-first search
  while(first_untried_configuration != NULL)
  {
    hn = first_untried_configuration;
    first_untried_configuration = first_untried_configuration->next_configuration;
    if(hn == last_untried_configuration)
      last_untried_configuration = NULL;
    init_map(hn->pieces);
    for(i = 0;i < n_pieces;i++)
    {
      if(try_move(hn->pieces,i, 1, 0,hn) != 0) return; // return early if solved
      if(try_move(hn->pieces,i,-1, 0,hn) != 0) return; // return early if solved
      if(try_move(hn->pieces,i, 0, 1,hn) != 0) return; // return early if solved
      if(try_move(hn->pieces,i, 0,-1,hn) != 0) return; // return early if solved
    }
  }
  // not solved
  assert(last_untried_configuration == NULL);
}
```

Klotski solver (part 6, main function)

We have finally reached the main function. If the puzzle has a solution, it prints it backwards (stating from the solution until the initial configuration is reached). This is done by following the `parent` pointers.

```
int main(void)
{
  u08 pieces[max_n_pieces];
  hash_node *hn;
  int i;

  width = 4;
  height = 5;
  n_pieces = 10;
  pieces[0] = info(0,0,0);
  pieces[1] = info(0,3,0);
  pieces[2] = info(2,0,1);
  pieces[3] = info(0,1,1);
  pieces[4] = info(0,2,1);
  pieces[5] = info(2,3,1);
  pieces[6] = info(1,1,2);
  pieces[7] = info(2,0,3);
  pieces[8] = info(3,1,3);
  pieces[9] = info(2,3,3);
  goal      = info(3,1,0);
  solve_puzzle(pieces);
  for(i = 0,hn = last_untried_configuration;hn != NULL;i++,hn = hn->parent)
  {
    init_map(hn->pieces);
    printf("move -%d\n",i);
    print_map();
  }
  printf("%d moves\n",i - 1);
  return 0;
}
```

Second written report work

— P.11 —

Summary:

- Empirical study of some sorting algorithms.

Graphs (part 1)
— TP.12 —

Summary:

- Introduction (definitions and examples)
- Data structures for graphs
- Graph traversal
- Connected components
- Connected components using the union-find data structure

Recommended bibliography for this lecture:

- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011

Useful external stuff:

- **Undirected graphs (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Directed graphs (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Union-find (slides)**, Sedgewick and Wayne, Princeton University, USA.

Introduction (part 1, initial definitions)

A **graph** G is a pair (V, E), where V is a set of **vertices** and E is a set of **edges**.

It is usual to consider that a vertex is a point in a plane (or in space), but that is not strictly necessary; a vertex is merely an object (a point, a word, a number, a book, a country, *et cetera*). A vertex usually has one or more properties. For example, a point usually has coordinates. The number of vertices of G is the number of elements of V and is denoted by $|V|$. It is usual to give numbers to the vertices, so that $V = \{v_1, \ldots, v_{|V|}\}$.

An edge e is a pair (v_i, v_j) of two vertices. The set of edges is a binary relation on V (the binary relation can be considered to be "connected to," so that an edge connects two vertices). It is also common to associate a property to an edge. For example, the vertices may represent cities and the edges may represent distances between pairs of cities. The number of edges of G is the number of elements of E and is denoted by $|E|$. Like the vertices, it is usual to give numbers to the edges, so that $E = \{e_1, \ldots, e_{|E|}\}$. We also have $e_i = (v_{1i}, v_{2i})$, i.e., edge e_i connects the vertices v_{1i} and v_{2i}.

A graph is **directed** (called a **digraph**) when the edges are **ordered** pairs of vertices. For directed graphs, edge e_i **departs from** (or **leaves**, or is **incident from**) vertex v_{1i}, and **arrives at** (or **enters**, or is **incident to**) vertex v_{2i}. In drawings, vertices are usually represented by dots or circles, and edges are represented by line or curve segments terminated by an arrow (sometimes the arrow is drawn in the middle of the segment instead of at its end).

A graph is **undirected** when the edges are **unordered** pairs of vertices, i.e., (e_i, e_j) is considered to be the same as (e_j, e_i). For undirected graphs, edge e_i is **incident on** both v_{1i} and v_{2i}. In drawings the edges to not have arrows.

The **degree** of a vertex of an undirected graph is the number of edges incident on it. For directed graphs the **in-degree** and **out-degree** of a vertex are the number of edges entering and leaving it, and the **degree** is the sum of the two.

Introduction (part 2, more definitions)

A **weighted** graph is one where each edge e_i has a weight w_i (a property of the edge). In an **unweighted** graph that does not happen. In drawings, the weight is written near the edge (near its arrow for directed graphs).

A **labeled** graph is one where each vertex v_i has a label l_i (a property of the vertex) that uniquely identifies it. In an **unlabeled** graph that does not happen. In drawings, the label is written either inside the vertex or near it.

A graph is **simple** if there do not exist more that one edge connecting any pair of vertices and if there do not exist self-loops, edges of the form (v, v). If that does not happen, the graph is **non-simple**.

A graph is **dense** if "is has many edges" and is **sparse** if it "has few edges." These definitions are a bit vague (on purpose). Since the number of vertex pairs of a graph with n vertices can be as high as $n(n-1)/2$, one possible characterization of a dense graph is one in which the number of edges is a sizable fraction of $n(n-1)/2$, while a sparse graph might be one in which the number of edges is a reasonably small multiple of n.

A **path** of length L starting at vertex v_s and ending in vertex v_d is a sequence $v_s, v_i, v_j, \ldots, v_d$ of $L+1$ vertices such that there exists an edge between consecutive vertices of the sequence. The length of the path is then the number of edges it contains. By convention, 0-length paths (of course with $v_d = v_s$) always exist. The path is **simple** if the sequence of vertices does not have repeated vertices. A path is a **cycle** if $L > 0$ and if $v_d = v_s$ (the cycle is simple if it does nor contain repeated vertices). For a weighted graph, the weight of a path is the sum of the weights of its edges, The weight of a cycle is defined in a similar way.

The vertices v_s and v_d are **connected** if there exists a path that starts at v_s and ends at v_d. A graph is **connected** if all its pairs of vertices are connected.

A graph is **acyclic** if it does not contain any cycles. Otherwise it is a **cyclic** graph. A **tree** is a connected, acyclic, undirected graph.

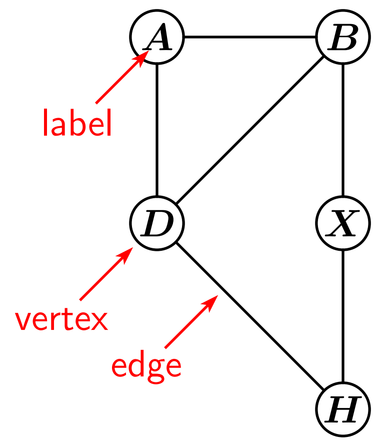
Introduction (part 3, even more definitions)

The set formed by the arrival vertices of the edges that depart from a vertex v, excluding v itself, is called the neighborhood of v. The definition can be extended to the neighborhood of a set V' of vertices: it is the set formed by the arrival vertices of the edges that depart from one of the vertices of V' and which do not arrive on a vertex of V'.

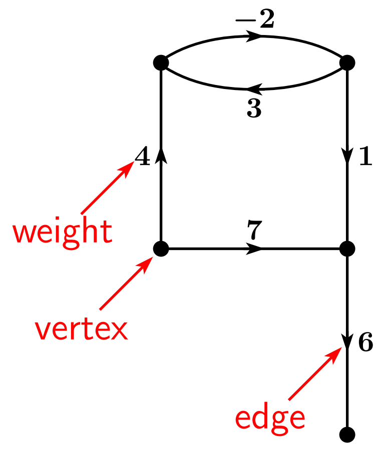
A spanning tree of a connected graph is a subgraph that is i) connected, ii) acyclic, and iii) includes all vertices. Conditions i) and ii) are the “tree” part and condition iii) is the spanning part of the name “spanning tree.”

Introduction (part 4, examples)

Undirected and labeled

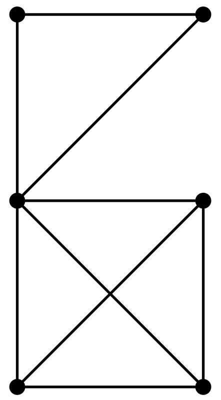


Directed and weighted

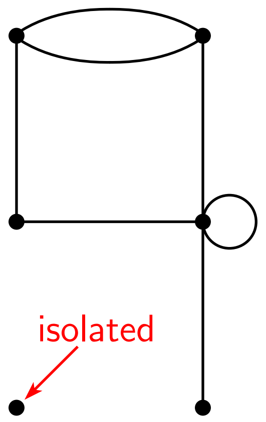


What is the degree of each vertex?

Simple

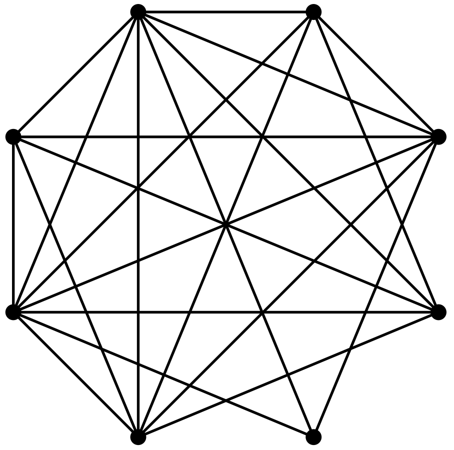


Non-simple

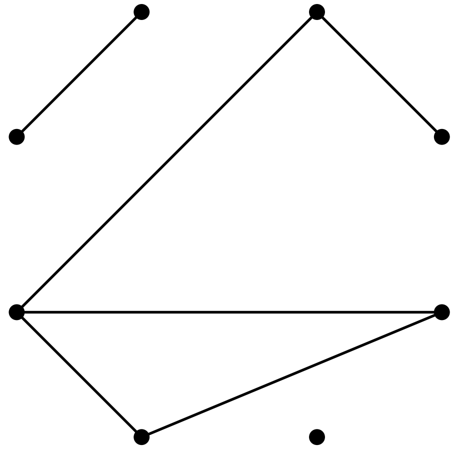


How many connected components?

Dense and connected

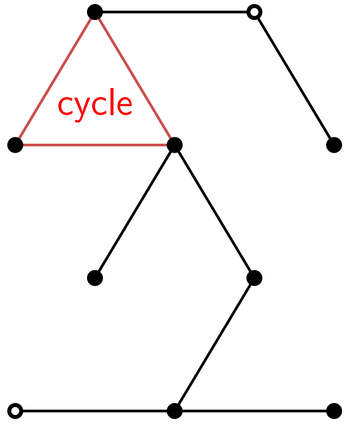


Sparse and unconnected

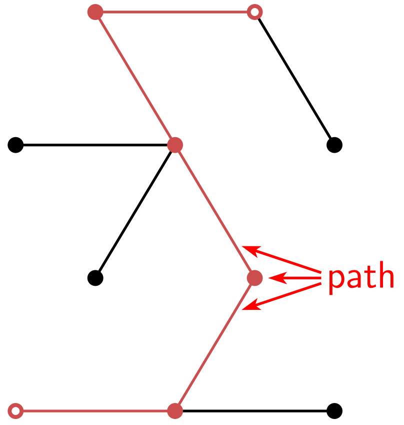


Average node degree?

Cyclic



Acyclic



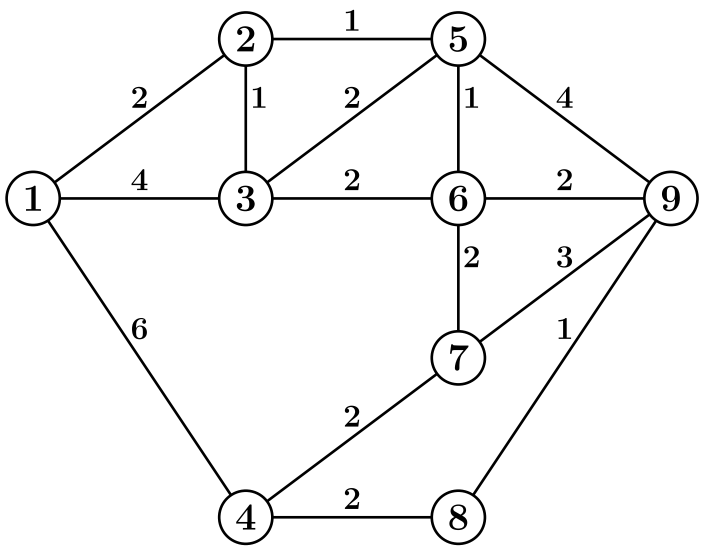
How many paths?

Data structures for graphs (part 1, adjacency matrix)

The adjacency matrix is a simple data structure (a $|V| \times |V|$ matrix) capable of storing all edge information of a simple graph (directed or undirected). It should be used when it is known that the graph has a small number of vertices, or when it is known that it is dense.

For unweighted graphs, the element in row i and column j of the adjacency matrix records the presence (1) or absence (0) of an edge starting at vertex i and ending at vertex j (for undirected graphs the adjacency matrix is a symmetric matrix). For weighted graphs it stores the weight of the edge starting at vertex i and ending at vertex j, if that edge exists, and stores an invalid weight value (for example, a NaN when the weight is a floating point number or zero if the weights are known to be positive) if it does not exist.

In the following example the adjacency matrix for the graph of the left is shown on the right (in this case the small zeros signal the absence of the edge):



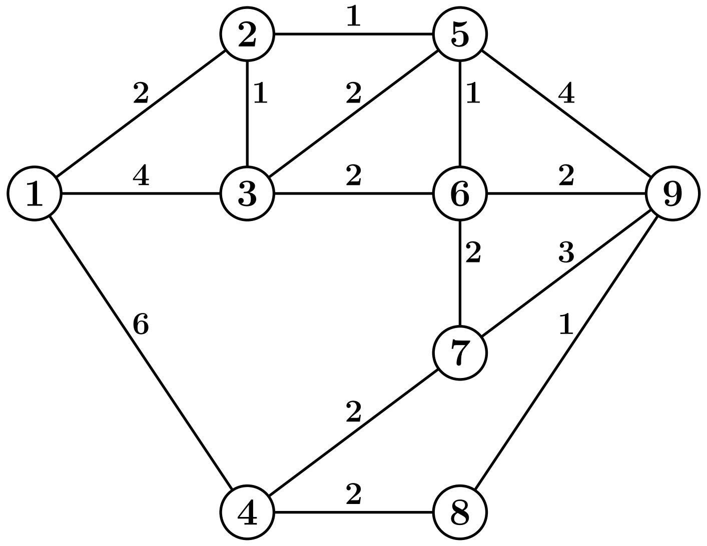
from\to	1	2	3	4	5	6	7	8	9
1	0	2	4	6	0	0	0	0	0
2	2	0	1	0	1	0	0	0	0
3	4	1	0	0	2	2	0	0	0
4	6	0	0	0	0	0	2	2	0
5	0	1	2	0	0	1	0	0	4
6	0	0	2	0	1	0	2	0	2
7	0	0	0	2	0	2	0	0	3
8	0	0	0	2	0	0	0	0	1
9	0	0	0	0	4	2	3	1	0

Data structures for graphs (part 2, adjacency lists)

The adjacency lists is another simple data structure (one linked list of edges per vertex) that is also capable of storing all edge information of a simple graph (directed or undirected). It should be used when it is known that the graph has a large number of vertices and that it is sparse (that is usually the case).

In this way to represent a graph, each vertex maintains a linked list of the edges that leave the vertex. For unweighted graphs, the nodes of the list only need to store the number of, or pointer to, the other vertex and, of course, the pointer to the next node of the linked list. For weighted graphs, they also store the weight.

In the following example the adjacency lists for the graph of the left are shown on the right (for each linked list node it is displayed first the other vertex number and then the edge weight):



1	$(4,6) \rightarrow (3,4) \rightarrow (2,2)$
2	$(5,1) \rightarrow (3,1) \rightarrow (1,2)$
3	$(6,2) \rightarrow (5,2) \rightarrow (2,1) \rightarrow (1,4)$
4	$(8,2) \rightarrow (7,2) \rightarrow (1,6)$
5	$(9,4) \rightarrow (6,1) \rightarrow (3,2) \rightarrow (2,1)$
6	$(9,2) \rightarrow (7,2) \rightarrow (5,1) \rightarrow (3,2)$
7	$(9,3) \rightarrow (6,2) \rightarrow (4,2)$
8	$(9,1) \rightarrow (4,2)$
9	$(8,1) \rightarrow (7,3) \rightarrow (6,2) \rightarrow (5,4)$

Note that the order of the edges in each list is arbitrary. Note also that for undirected graphs the size of a linked list is equal to the degree of its vertex (it is equal to the out-degree of the vertex for directed graphs).

Graph traversal (part 1, short description)

Traversing a graph is a fundamental operation. One starts at a given vertex and one moves along the edges until one visits all possible vertices. This can be done in a depth-first style or in a breadth-first style. Both are useful. The next slide explains with C code how to do this. That code uses the following data types and support functions.

```
deque *create_deque(int max_size); // deque creation
void destroy_deque(deque *dq);     // deque destruction
int get_lo(deque *dq);             // dequeue
int get_hi(deque *dq);             // pop
void put_hi(deque *dq,int v);      // push or enqueue

typedef struct edge
{
  struct edge *next; // next edge node
  int vertex_number; // vertex number
  int weight;        // edge weight
}
edge;
```

```
typedef struct vertex
{
  edge *out_edges;  // adjacency list head
  int mark;         // for graph traversals
}
vertex;

typedef struct graph
{
  vertex *vertices; // array of vertices (pointer)
  int n_vertices;   // number of vertices
}
graph;
```

To avoid visiting the same vertex more than once, a vertex is marked when it is touched for the first time (touching amounts to put the vertex in either a stack or a queue), and, later on, when it is visited and its edges are followed (to touch other vertices) the mark is given its final value (sequential visitation number). The following function is used to initialize the marks:

```
void mark_all_vertices(graph *g,int mark)
{
  for(int i = 0;i < g->n_vertices;i++)
    g->vertices[i].mark = mark;
}
```

Graph traversal (part 2, code)

Here is the code for a depth-first and a breadth-first traversal of a graph.

```
void depth_first(graph *g,int initial_vertex)
{
  deque *dq;
  edge *e;
  int i,n;

  mark_all_vertices(g,-1);
  dq = create_deque(g->n_vertices);
  put_hi(dq,initial_vertex);
  n = 0;
  while(dq->size > 0)
  {
    i = get_hi(dq); // pop
    g->vertices[i].mark = n++; // visit
    for(e = g->vertices[i].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark == -1)
      { // touch
        g->vertices[e->vertex_number].mark = 0;
        put_hi(dq,e->vertex_number); // push
      }
  }
  destroy_deque(dq);
}
```

```
void breadth_first(graph *g,int initial_vertex)
{
  deque *dq;
  edge *e;
  int i,n;

  mark_all_vertices(g,-1);
  dq = create_deque(g->n_vertices);
  put_hi(dq,initial_vertex);
  n = 0;
  while(dq->size > 0)
  {
    i = get_lo(dq); // dequeue
    g->vertices[i].mark = n++; // visit
    for(e = g->vertices[i].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark == -1)
      { // touch
        g->vertices[e->vertex_number].mark = 0;
        put_hi(dq,e->vertex_number); // enqueue
      }
  }
  destroy_deque(dq);
}
```

Notice how a small change makes a big difference in visiting order. For the graph used as example a few slides ago, starting at the vertex number $\mathbf{1}$ a depth-first traversal visits the vertices in the order $\mathbf{1, 2, 5, 6, 7, 9, 8, 3, 4}$, and a breadth-first traversal visits them in the order $\mathbf{1, 4, 3, 2, 8, 7, 6, 5, 9}$.

Connected components

Either version of the graph traversal code can be easily modified so that the new code

- finds all vertices connected to a given one,
  [Solution. The code as is already does this: vertices that at the end have a mark of -1 are not connected to the starting vertex. Marking the visited vertices with the vertex visiting order is not necessary!]
- subdivides a graph into connected components,
  [Solution. Mark initially all vertices with -1. Set a region number also to -1. Then, do the following until all vertices have a non-negative mark: Increase the region number by one; Find a vertex with a negative mark; Apply one of the algorithms to mark all vertices connected to it (when they are first touched) with the current region number.]
- determines if the graph is cyclic or acyclic.
  [Solution. Apply the method of the previous item but stop it saying the graph is cyclic if the algorithm touches a vertex that has already been touched and that has the same region number!]

The breadth-first variant can also be easily modified to find the smallest distance, in terms of number of edges traveled, between a starting vertex and all other vertices connected to it.
[Solution. Mark each touched vertex with one more than the mark of the vertex being visited.] (The depth-first variant can also do this, but in a more complex way.)

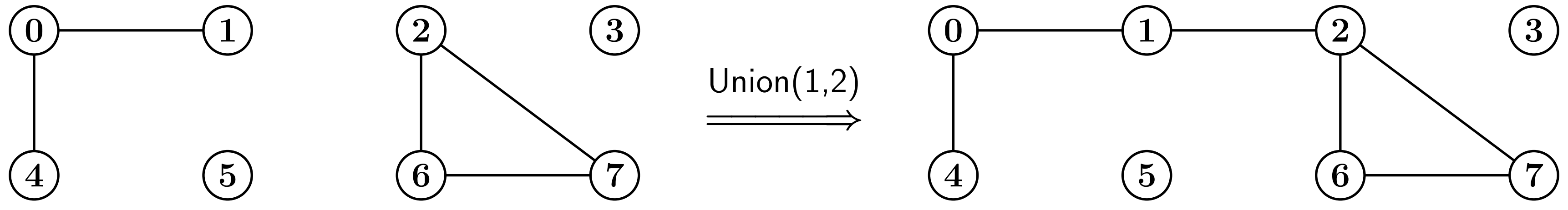
Connected components using the union-find data structure (part 1, problem)

The union-find data structure is an efficient way to keep track of the connected components of an **undirected** graph. (There are other application of this data structure.) Its main operations are

- **[Union]** replace the connected components containing vertices p and q with their union (if the vertices belong to the same component then there is nothing to to, otherwise join the two components), and
- **[Find]** given a vertex p, find the vertex q that represents the connected component that p belongs to.

To keep the connectivity information of a graph up-to-date for each edge insertion do an union operation of its incident vertices. Checking if two vertices are connected amounts to check if the representatives of their connected components are the same.

In the following example, an edge between vertices 1 and 2 is inserted in the graph. In this case the corresponding union operation joints two connected components.



Four connected components
$\{0,1,4\},\{2,6,7\},\{3\},\{5\}$

Three connected components
$\{0,1,2,4,6,7\},\{3\},\{5\}$

One of the vertices of each connected component will be the representative vertex of that component. For a graph without edges, each vertex is the representative vertex of its component (each component has only one vertex). Each union operation just makes the representative of the component of its second argument be the representative of the first.

Connected components using the union-find data structure (part 2a, code)

In order to keep track of the representative (vertex number) of each connected component and of the number of connected components, the `vertex` and `graph` structures need to be modified as follows.

```
typedef struct vertex
{
  edge *out_edges;    // adjacency list head
  int mark;           // for graph traversals
  int representative; // vertex number of representative
}
vertex;

typedef struct graph
{
  vertex *vertices;           // array of vertices (pointer)
  int n_vertices;             // number of vertices
  int n_connected_components; // number of connected components
}
graph;
```

The following extra code should be placed in the function that creates an empty graph.

```
  g->n_connected_components = n_vertices;
  for(int i = 0;i < n_vertices;i++)
    g->vertices[i].representative = i;
```

Connected components using the union-find data structure (part 2b, code)

As stated previously, the union operation amounts to change the representative of one connected region to be the representative of another connected region. Thus, in order to find the vertex number of the representative vertex of a connected region it is this necessary to follow the representative numbers until there is no change. To speed up subsequent queries, the representative numbers of all vertices that were visited in the query should be updated with the current representative number (that is called path compression). The following function does all this.

```
int find_representative(graph *g,int vertex_number)
{
  int i,j,k;

  // find (linked list done with index numbers!)
  for(i = vertex_number;i != g->vertices[i].representative;i = g->vertices[i].representative)
    ;
  // path compression
  for(j = vertex_number;j != i;j = k)
  {
    k = g->vertices[j].representative;
    g->vertices[j].representative = i;
  }
  return i;
}
```

Connected components using the union-find data structure (part 2c, code)

Finally, the function that adds an edge to the graph needs to be updated to maintain the union-find data.

```
int add_edge(graph *g,int from,int to,int weight)
{
  edge *e;

  assert(from >= 0 && from < g->n_vertices && to >= 0 && to < g->n_vertices && from != to);
  for(e = g->vertices[from].out_edges;e != NULL && e->vertex_number != to;e = e->next)
    ;
  if(e != NULL)
    return 0;
  e = create_edge();
  e->next = g->vertices[from].out_edges;
  g->vertices[from].out_edges = e;
  e->vertex_number = to;
  e->weight = weight;
```

```
  int fi = find_representative(g,from);
  int ti = find_representative(g,to);
  if(fi != ti)
  { // union
    g->vertices[ti].representative = fi;
    g->n_connected_components--;
  }
```

```
  return 1;
}
```

Possible third written report work

— P.12 —

Summary:

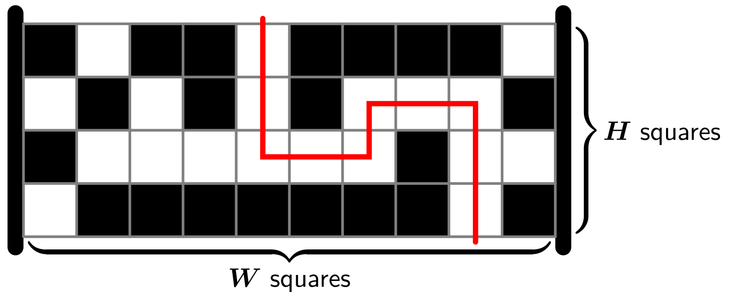
- Connectivity using union-find

Connectivity using union-find

Two regions are separated by a $W \times H$ grid made of squares ($W = 20$, $H = 10$). Each square of the grid may be either white or black. Initially, all squares are black. Black squares, chosen at random, are then changed into white ones. On average, how many black squares need to be changed into white ones until there is a path made entirely of white squares between the two regions?

Too easy? Then experiment with other grid sizes.

Suggestions: There is a simple union-find example in the `P12.tgz` archive. Strip it of all graph-related stuff (keep only the union-find stuff). When a black square is turned into white do an union with all its white neighbors (the two regions are initially white). Stop when the two regions have the same representative.

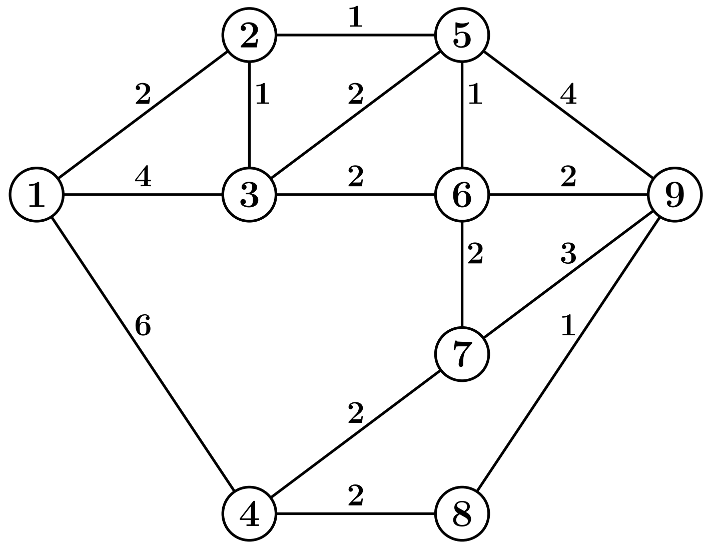


Graphs (part 2)
— TP.13 —

Summary:

- All paths
- All cycles
- Shortest path
- Minimum spanning tree

All examples of this lecture use the following "test" graph.



Recommended bibliography for this lecture:

- **The Algorithm Design Manual**, Steven S. Skiena, second edition, Springer, 2008.
- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, third edition, The MIT press, 2009.
- **Algorithms**, Robert Sedgewick and Kevin Wayne, fourth edition, Addison Wesley, 2011

Useful external stuff:

- **Shortest paths (slides)**, Sedgewick and Wayne, Princeton University, USA.
- **Minimum spanning trees (slides)**, Sedgewick and Wayne, Princeton University, USA.

All paths (part 1, problem description and solution

Problem: given two distinct vertices of a graph, find all simple paths that begin at the first given vertex (the departure vertex) and end at the second given vertex (the destination vertex).

Solution: Do a depth-first traversal starting at the departure vertex, using the vertex marks to keep track of the path being built. In particular,

- the depth-first traversal can be done using recursion (that is not possible in a breadth-first traversal),
- the depth-first traversal should backtrack when the destination vertex is reached,
- initially mark all vertices with, say, -1 (an invalid vertex number),
- do not follow an edge if it arrives at an already marked vertex, and
- when following an edge, mark the vertex it departs from with the number of the vertex it arrives at (do not forget to unmark it when backtracking!).

Marking with vertex numbers is quite useful because these marks can be used to follow each complete path from its departing vertex to its destination vertex (a linked list!).

Task: For our test graph, enumerate all paths starting at vertex 1 and ending at vertex 9. Identify the paths with the smallest and largest weights (the weight of a path is the sum of the weights of its edges).

Curiosity: In a complete undirected graph with n vertices there are $f(n)$ paths between any two vertices, where $f(1) = 0$, $f(2) = 1$, and, for $n > 2$, $f(n) = (n-2)f(n-1) + 1$. Note that $f(n) \approx e \cdot (n-2)!$

n	$f(n)$	n	$f(n)$	n	$f(n)$
1	0	5	16	9	13700
2	1	6	65	10	109601
3	2	7	326	11	986410
4	5	8	1957	12	9864101

All paths (part 2, code)

The following C functions implement one possible solution to the all paths problem posed in the previous slide.

```c
void all_paths_r(graph *g,int initial_vertex,int final_vertex,int current_vertex,int current_weight)
{
  int i;

  if(current_vertex == final_vertex)
  { // found one!
    printf("  %d",(i = initial_vertex) + 1);
    do
      printf(" -> %d",(i = g->vertices[i].mark) + 1);
    while(i != final_vertex);
    printf(" [%d]\n",current_weight);
  }
  else
    for(edge *e = g->vertices[current_vertex].out_edges;e != NULL;e = e->next)
      if(g->vertices[e->vertex_number].mark < 0)
      {
        g->vertices[current_vertex].mark = e->vertex_number;
        all_paths_r(g,initial_vertex,final_vertex,e->vertex_number,current_weight + e->weight);
        g->vertices[current_vertex].mark = -1;
      }
}

void all_paths(graph *g,int initial_vertex,int final_vertex)
{
  assert(initial_vertex != final_vertex);
  printf("all paths between vertices %d and %d\n",initial_vertex + 1,final_vertex + 1);
  mark_all_vertices(g,-1);
  all_paths_r(g,initial_vertex,final_vertex,initial_vertex,0);
}
```

All paths (part 3, example)

There are $\mathbf{33}$ paths starting at vertex $\mathbf{1}$ and ending at vertex $\mathbf{9}$:

$1 \to 4 \to 8 \to 9$ (weight 9)
$1 \to 4 \to 7 \to 9$ (weight 11)
$1 \to 4 \to 7 \to 6 \to 9$ (weight 12)
$1 \to 4 \to 7 \to 6 \to 5 \to 9$ (weight 15)
$1 \to 4 \to 7 \to 6 \to 3 \to 5 \to 9$ (weight 18, **largest**)
$1 \to 4 \to 7 \to 6 \to 3 \to 2 \to 5 \to 9$ (weight 18, **largest**)
$1 \to 3 \to 6 \to 9$ (weight 8)
$1 \to 3 \to 6 \to 7 \to 9$ (weight 11)
$1 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 13)
$1 \to 3 \to 6 \to 5 \to 9$ (weight 11)
$1 \to 3 \to 5 \to 9$ (weight 10)
$1 \to 3 \to 5 \to 6 \to 9$ (weight 9)
$1 \to 3 \to 5 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 3 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 3 \to 2 \to 5 \to 9$ (weight 10)
$1 \to 3 \to 2 \to 5 \to 6 \to 9$ (weight 9)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 2 \to 5 \to 9$ (weight 7)
$1 \to 2 \to 5 \to 6 \to 9$ (weight 6, **smallest**)
$1 \to 2 \to 5 \to 6 \to 7 \to 9$ (weight 9)
$1 \to 2 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 11)
$1 \to 2 \to 5 \to 3 \to 6 \to 9$ (weight 9)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 9$ (weight 12)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 14)
$1 \to 2 \to 3 \to 6 \to 9$ (weight 7)
$1 \to 2 \to 3 \to 6 \to 7 \to 9$ (weight 10)
$1 \to 2 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 12)
$1 \to 2 \to 3 \to 6 \to 5 \to 9$ (weight 10)
$1 \to 2 \to 3 \to 5 \to 9$ (weight 9)
$1 \to 2 \to 3 \to 5 \to 6 \to 9$ (weight 8)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 9$ (weight 11)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 4 \to 8 \to 9$ (weight 13)

The path with the smallest weight is, in this case, not one of the paths with the smallest number of edges.

All cycles

Problem: Find all simple cycles (closed paths) of a graph.

Solution: Do a depth-first traversal starting at each vertex, never move to a vertex with a number (index) smaller than that of the starting vertex, and using the vertex marks to keep track of the cycle being built. In particular,

- the depth-first traversal can be done using recursion (that is not possible in a breadth-first traversal),
- the depth-first traversal should backtrack when a cycle is found,
- initially mark all vertices with, say, -1 (an invalid vertex number),
- do not follow an edge if it arrives at an already marked vertex or if that vertex has a number (index) smaller than that of the starting vertex , and
- when following an edge, mark the vertex it departs from with the number of the vertex it arrives at (do not forget to unmark it when backtracking!).

Task: Enumerate all cycles of our test graph. Identify the cycles with the smallest and largest weights (the weight of a cycle is the sum of the weights of its edges).

Curiosity: In a complete undirected graph with n vertices there are $g(n)$ cycles, where $g(1) = g(2) = 0$, $g(3) = 1$, and, for $n > 3$, $g(n) = \frac{(n-1)(n-2)}{2} + ng(n-1) - (n-1)g(n-2)$. Note that

$$g(n) = \frac{n!}{2}\sum_{k=3}^{n}\frac{1}{k(n-k)!} \approx \frac{e}{2}\frac{(n+1)!}{n^2}.$$

n	$f(n)$	n	$f(n)$	n	$f(n)$
1	0	5	37	9	62814
2	0	6	197	10	556014
3	1	7	1172	11	5488059
4	7	8	8018	12	59740609

All cycles (part 2, code)

The following C functions implement one possible solution to the all cycles problem posed in the previous slide.

```c
void all_cycles_r(graph *g,int initial_vertex,int current_vertex,int current_weight)
{
  int i;

  for(edge *e = g->vertices[current_vertex].out_edges;e != NULL;e = e->next)
    if(e->vertex_number == initial_vertex)
    { // found one!
      if(g->vertices[initial_vertex].mark < current_vertex)
      { // make sure each cycle is reported only once (and ignore cycles with only two egdes)
        printf("  %d",(i = initial_vertex) + 1);
        do
          printf(" -> %d",(i = g->vertices[i].mark) + 1);
        while(i != current_vertex);
        printf(" -> %d [%d]\n",initial_vertex + 1,current_weight + e->weight);
      }
    }
    else if(e->vertex_number > initial_vertex && g->vertices[e->vertex_number].mark < 0)
    {
      g->vertices[current_vertex].mark = e->vertex_number;
      all_cycles_r(g,initial_vertex,e->vertex_number,current_weight + e->weight);
      g->vertices[current_vertex].mark = -1;
    }
}

void all_cycles(graph *g)
{
  printf("all cycles\n");
  mark_all_vertices(g,-1);
  for(int i = 0;i < g->n_vertices;i++)
    all_cycles_r(g,i,i,0);
}
```

All cycles (part 3, example)

$1 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 17)
$1 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 16)
$1 \to 3 \to 6 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 6 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 3 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 3 \to 2 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 3 \to 2 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 2 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 22, **largest**)
$1 \to 3 \to 2 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 3 \to 2 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 3 \to 2 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 9 \to 8 \to 4 \to 7 \to 6 \to 3 \to 1$ (weight 20)
$1 \to 2 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 5 \to 9 \to 7 \to 6 \to 3 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 5 \to 9 \to 6 \to 3 \to 1$ (weight 15)
$1 \to 2 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 15)
$1 \to 2 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 14)
$1 \to 2 \to 5 \to 6 \to 3 \to 1$ (weight 10)
$1 \to 2 \to 5 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 5 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 5 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 5 \to 3 \to 1$ (weight 9)
$1 \to 2 \to 3 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 3 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 3 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 6 \to 7 \to 4 \to 1$ (weight 15)
$1 \to 2 \to 3 \to 6 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 6 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 3 \to 5 \to 9 \to 8 \to 4 \to 1$ (weight 18)
$1 \to 2 \to 3 \to 5 \to 9 \to 7 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 3 \to 5 \to 9 \to 6 \to 7 \to 4 \to 1$ (weight 21)
$1 \to 2 \to 3 \to 5 \to 6 \to 9 \to 8 \to 4 \to 1$ (weight 17)
$1 \to 2 \to 3 \to 5 \to 6 \to 9 \to 7 \to 4 \to 1$ (weight 19)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 9 \to 8 \to 4 \to 1$ (weight 20)
$1 \to 2 \to 3 \to 5 \to 6 \to 7 \to 4 \to 1$ (weight 16)
$1 \to 2 \to 3 \to 1$ (weight 7)
$2 \to 3 \to 6 \to 9 \to 5 \to 2$ (weight 10)
$2 \to 3 \to 6 \to 7 \to 9 \to 5 \to 2$ (weight 13)
$2 \to 3 \to 6 \to 7 \to 4 \to 8 \to 9 \to 5 \to 2$ (weight 15)
$2 \to 3 \to 6 \to 5 \to 2$ (weight 5)
$2 \to 3 \to 5 \to 2$ (weight 4, **smallest**)
$3 \to 5 \to 9 \to 8 \to 4 \to 7 \to 6 \to 3$ (weight 15)
$3 \to 5 \to 9 \to 7 \to 6 \to 3$ (weight 13)
$3 \to 5 \to 9 \to 6 \to 3$ (weight 10)
$3 \to 5 \to 6 \to 3$ (weight 5)
$4 \to 7 \to 9 \to 8 \to 4$ (weight 8)
$4 \to 7 \to 6 \to 9 \to 8 \to 4$ (weight 9)
$4 \to 7 \to 6 \to 5 \to 9 \to 8 \to 4$ (weight 12)
$5 \to 6 \to 9 \to 5$ (weight 7)
$5 \to 6 \to 7 \to 9 \to 5$ (weight 10)
$6 \to 7 \to 9 \to 6$ (weight 7)
[There are 65 cycles.]

Shortest path (part 1, problem formulation)

Problem: Given a weighted directed graph, find the path with the smallest weight between a given source vertex s and a given destination vertex d (connected to s). Assume that all weights are nonnegative.

Similar problem: Given a weighted directed graph and a source vertex s, find the paths with smallest weight to all other vertices of the graph (in the same connected component). Assume that all weights are nonnegative.

Related problem: Given a weighted directed graph, find the path with the smallest weight between a given source vertex and a given destination vertex (connected to s). Assume that the weights can take positive, zero, or negative values, and that all cycles have a positive weight.

Possible solution: Use brute force. Generate all possible paths between s and d and choose one with the smallest weight. Bad idea! The number of paths that may need to be checked can be very large.

Solution: Exploit the property that if a path of smallest weight passes through vertex v then its sub-path from s to v has the smallest possible weight of all paths starting at s and ending at v. (If that were not the case then that sub-path could be replaced by a better one.)

Let $W(s,v)$ be the weight of the best path that starts at s and ends at v. Our problem is to find $W(s,d)$ and to find a path with this weight. Let $w_{p,q}$ be the weight of the edge that connects p to q (if there is no edge $w_{p,q}$ is $+\infty$). We have

$$W(s,d) = \min_{v \in V}\bigl(W(s,v) + w_{v,d}\bigr).$$

To solve the problem it is thus necessary to solve smaller problems of the same type. As explained in the next slide, it is better to organize the computation so that larger and larger problems are solved (instead of smaller and smaller as this equation implies).

Shortest path (part 2, solution [Dijkstra's algorithm])

To implement the equation at the bottom of the previous slide it is necessary to find all edges that arrive at d. That information is not stored in the graph data structures that we have been using in these slides (for each vertex only the edges that leave the vertex are stored). Although this can be solved easily, it remains the problem of how to organize the computation (in a dynamic programming style) so that cycles in the graph do not give rise to cycles in the algorithm.

It turns out that it is easier to construct the solution, one vertex at a time, starting at vertex s. Suppose that V' is a subset of V that contains s, that $W(s,p)$ is already known for all $p \in V'$, and that V' contains the $|V'|$ vertices with the smallest values of $W(s,v)$. Initially $V' = \{s\}$ and $W(s,s) = 0$. Each step of the algorithm selects an edge departing from a vertex $p \in V'$ and arriving at a vertex $q \in V - V'$ (in the neighborhood of V'), so that $W(s,p) + w_{p,q}$ is minimized. (Break ties arbitrarily.) This ensures, for the vertices p and q of the selected edge, that $W(s,q) = W(s,p) + w_{p,q}$, because since edge weights are nonnegative any other path would not be better. If q is d we are done. Otherwise, add q to V' and repeat the procedure. Best paths can be reconstructed by recording the edges that were selected (store in vertex q the corresponding p, so that paths can be traced back from d to s).

Shortest path (part 3, implementation details)

For each vertex v keep

- the weight $Z(s,v)$ of the current best path from s to v — if $v \in V'$ then $W(s,v) = Z(s,v)$,
- the number of the previous vertex, $P(v)$, of the current best path ending at the vertex.

Initially, set $Z(s,s) = 0$ and set $Z(s,v) = \infty$ for the other vertices. When a vertex p is added to V' (the first one will be s itself), for all vertices q that p is connected to check if $Z(s,q)$ is larger than $Z(s,p) + w_{p,q}$. If it is, set $Z(s,q)$ to $Z(s,p) + w_{p,q}$ and set $P(q) = p$.

The next vertex to be added to V' will be the one, not already there, with the smallest value of $W(s,v)$. To do this efficiently, the values of $W(s,v)$ can be kept in a min-heap. This is not entirely trivial to do, because it is necessary to keep track of the position where each $Z(s,v)$ value is stored in the heap. This is necessary because it may be necessary to decrease its value. It is thus also necessary, for each vertex v, to keep

- the heap position (an index) where $Z(s,v)$ is stored.

It can be shown that this algorithm takes $O\big(|E| \log |V|\big)$ time to finish.

Shortest path (part 4a, code – data structures)

To simplify the code the min heap data can be placed inside the data structures used to represent a graph (the vertex `mark` field is not needed here).

```
typedef struct edge   edge;
typedef struct vertex vertex;
typedef struct graph  graph;

struct edge
{
  edge *next;          // next edge node
  int vertex_number;   // vertex number
  int weight;          // edge weight
};

struct vertex
{
  edge *out_edges;     // adjacency list head

  vertex *trace_back;  // "follow-path-back" vertex
  int weight;          // shortest path weight
  int heap_index;      // min heap index of this vertex

};

struct graph
{
  vertex *vertices;    // the vertices
  int n_vertices;      // number of vertices

  vertex **heap;       // heap of vertex pointers
  int heap_size;       // current heap size

};
```

Shortest path (part 4b, code – update vertex data)

The following function updates $Z(s,q)$ given p and $Z(s,p)+w_{p,q}$. The function argument `from` is a pointer to the vertex p, the argument `to` is a pointer to q, and the argument `weight` is $Z(s,p)+w_{p,q}$. The function also keeps vertices in their proper places in the min-heap.

```
static void update_weight(graph *g,vertex *from,vertex *to,int weight)
{
  int i;

  if(to->trace_back != NULL && to->weight <= weight)
    return; // not better
  to->weight = weight;
  to->trace_back = from;
  if(to->heap_index > 0)
    i = to->heap_index; // already in the heap
  else
    i = ++g->heap_size; // augment heap
  for(;i > 1 && g->heap[i / 2]->weight > weight;i /= 2)
    (g->heap[i] = g->heap[i / 2])->heap_index = i;
  (g->heap[i] = to)->heap_index = i;
}
```

The `trace_back` fields are initialized to `NULL`, so that is used to deal correctly with the first update of the `to` data. The `heap_index` fields are also initialized to 0, so that is used to check if `to` is already in the min-heap (if so, it says where it is).

Shortest path (part 4c, code – get vertex with the smallest path weight)

The following function used the min-heap data to retrieve the vertex with the smallest path weight. It returns `NULL` when the min-heap is empty.

```
static vertex *get_vertex_with_min_weight(graph *g)
{
  vertex *v,*t;
  int i,j;

  if(g->heap_size == 0)
    return NULL;
  v = g->heap[1];
  t = g->heap[g->heap_size--];
  for(i = 1;2 * i <= g->heap_size;i = j)
  {
    j = (2 * i + 1 <= g->heap_size && g->heap[2 * i + 1]->weight < g->heap[2 * i]->weight) ? 2 * i + 1 : 2 * i;
    (g->heap[i] = g->heap[j])->heap_index = i;
  }
  (g->heap[i] = t)->heap_index = i;
  v->heap_index = 0; // do this last bacause t = v when the heap becomes empty
  return v;
}
```

Shortest path (part 4d, code – the algorithm)

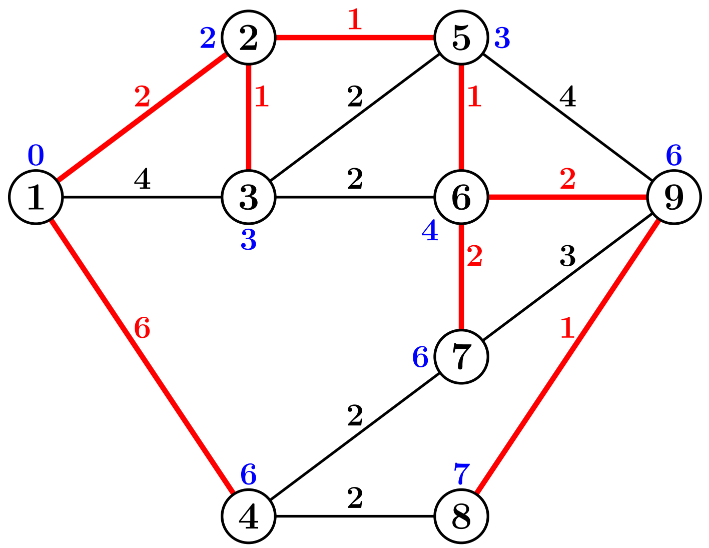
Finally, the Dijkstra algorithm. First, the vertex data and the min-heap are initialized. Then, vertices are extracted from the min-heap one at a time, all their outgoing edges are followed, that the information of each destination vertex is updated. The function computes the path of smallest weight from the starting vertex to all other vertices. It would be trivial to modify the code to stop the algorithm when the best path to a given destination vertex becomes known.

```
static void shortest_paths(graph *g,int starting_vertex)
{
  vertex *v;
  edge *e;
  int i;

  assert(starting_vertex >= 0 && starting_vertex < g->n_vertices);
  for(i = 0;i < g->n_vertices;i++)
  {
    g->vertices[i].weight = 0;
    g->vertices[i].trace_back = NULL;
    g->vertices[i].heap_index = 0;
  }
  update_weight(g,&g->vertices[starting_vertex],&g->vertices[starting_vertex],0);
  while((v = get_vertex_with_min_weight(g)) != NULL)
    for(e = v->out_edges;e != NULL;e = e->next)
      update_weight(g,v,&g->vertices[e->vertex_number],v->weight + e->weight);
}
```

The entire code can be found in the `P13.tgz` archive.

Shortest path (part 5, example)

The following figure presents the best paths from vertex 1 to all other vertices. The smallest weight of each path is shown in blue near the destination vertex. Edges belonging to at least one best path are shown in red. Note that they form a tree.



The progress of the algorithm is displayed on the table on the right. Each stage of the algorithm has two phases. In the first, one of the vertices with the smallest distance that has not yet been selected (not blue) is selected; this is signaled by a small circle placed between columns. In the second, the distances of the paths that pass through the selected vertex are updated (best current distances are presented in black, discarded distances are presented in gray). The algorithm terminates either when there are no more vertices to select or when the destination vertex has been selected.

1	0 ●	0	0	0	0	0	0	0	0	0
2	∞	∞,2 ●	2	2	2	2	2	2	2	2
3	∞	∞,4	4,3 ●	3	3	3	3	3	3	3
4	∞	∞,6	6	6	6	6	6,8 ●	6	6	6
5	∞	∞	∞,3	3,5 ●	3	3	3	3	3	3
6	∞	∞	∞	∞,5	5,4 ●	4	4	4	4	4
7	∞	∞	∞	∞	∞	∞,6 ●	6	6	6	6
8	∞	∞	∞	∞	∞	∞	∞	∞,8	8,7 ●	7
9	∞	∞	∞	∞	∞,7	7,6	6,9	6 ●	6	6

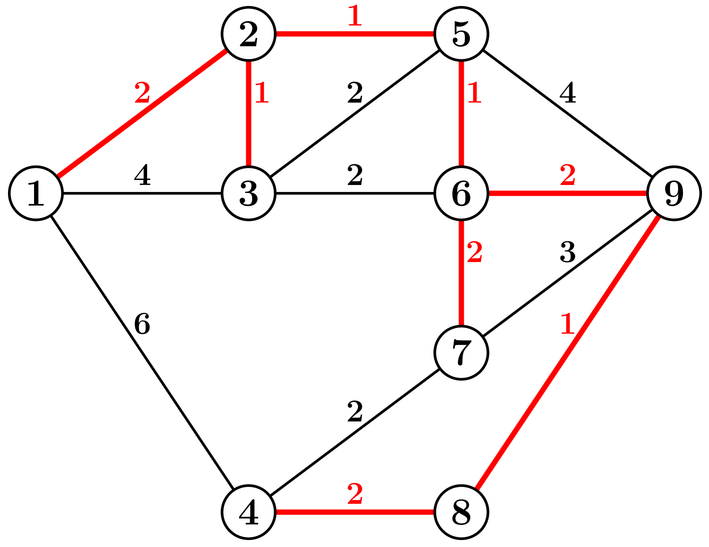
Minimum spanning tree (part 1, Kruskal's algorithm and example)

Problem: Given a weighted undirected connected graph, find the spanning tree with the smallest weight. (The weight of a tree is the sum of the weights of its edges.)

Solution [Kruskal]: Sort the edges in increasing order. While the number of accepted edges is smaller than the number of vertices minus one, get the next edge (start with the smallest). If its two endpoints belong to different connected components, accept the edge (and update the connected components information using the union-find data structure). Otherwise reject the edge.

It can be shown that this algorithm takes $O(|E| \log |E|)$ time to finish.

When the edge weights are distinct, the minimum spanning tree is unique. Otherwise, many spanning trees with the same weight can exist.



Minimum spanning tree (part 2, Prim's algorithm)

Solution [Prim]: Let V' be a set of vertices having initially only one vertex (chosen arbitrarily). Let E' be the set of edges that connect one vertex of V' to a vertex in the neighborhood of V'. Repeat the following procedure until E' becomes empty. Select the edge of E' with the smallest weight. That edge becomes part of the minimum spanning tree. Add its arriving vertex to V'.

It can be shown that this algorithm takes $O\bigl(|E|\log|V|\bigr)$ time to finish when the edge selection is done using a priority queue (min-heap).

Third written report work

— P.13 —

Summary:

- Connectivity using union-find

Some topics on computational geometry
— TP.14 —

Summary:

- Steiner trees
- Point location (grid, quad-tree, oct-tree)
- Convex hull, Delaunay triangulation, Voronoi diagram (examples)

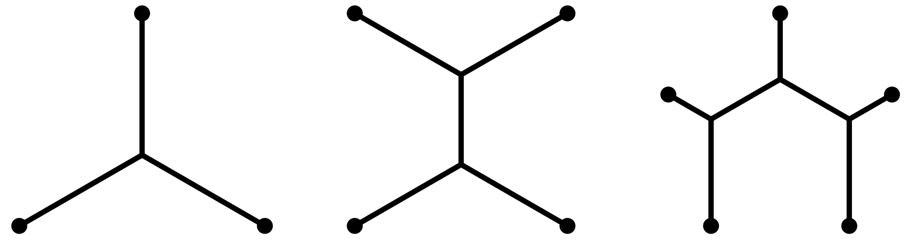
Recommended bibliography for this lecture:

- **Computational Geometry. Algorithms and Applications**, M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, second edition, Springer, 2000.
- **Computational Geometry in C**, Joseph O'Rourke, second edition, Cambridge University Press, 1998.

Steiner trees

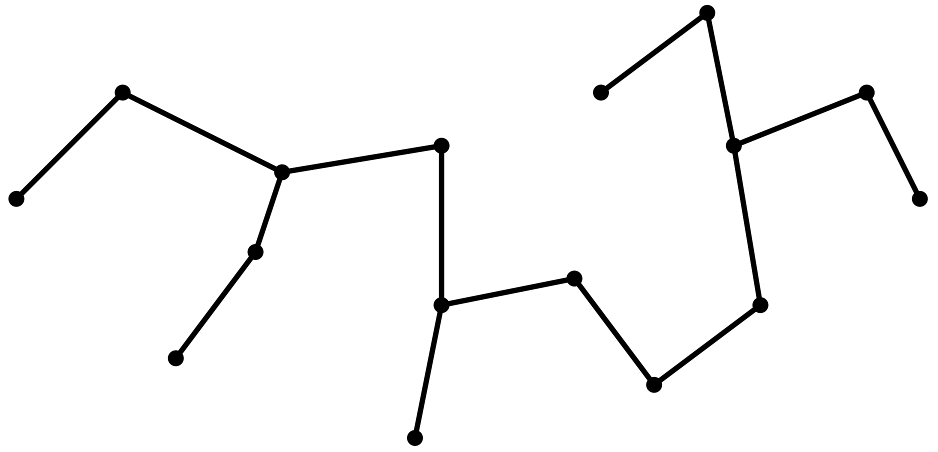
Problem: Given a set of points in the plane, find an interconnection tree with minimal total distance.

This is a very difficult problem (NP-hard).



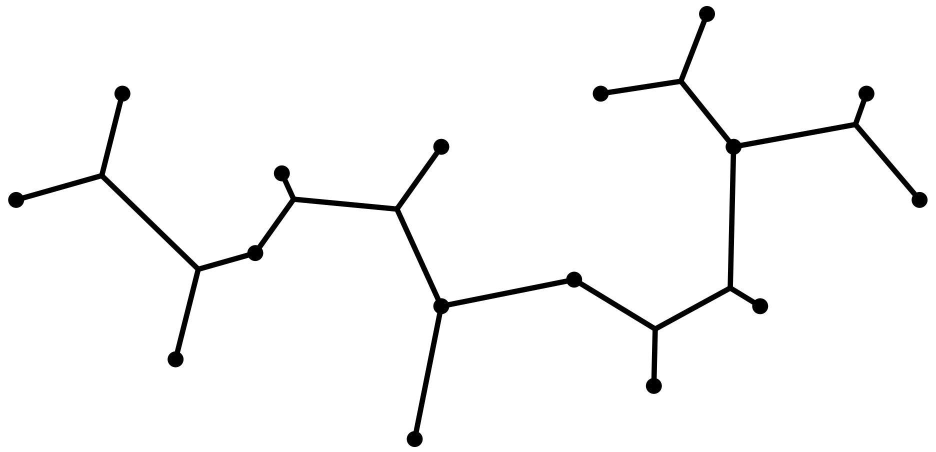
(Can be generalized to more than two dimensions.)

Minimum spanning tree



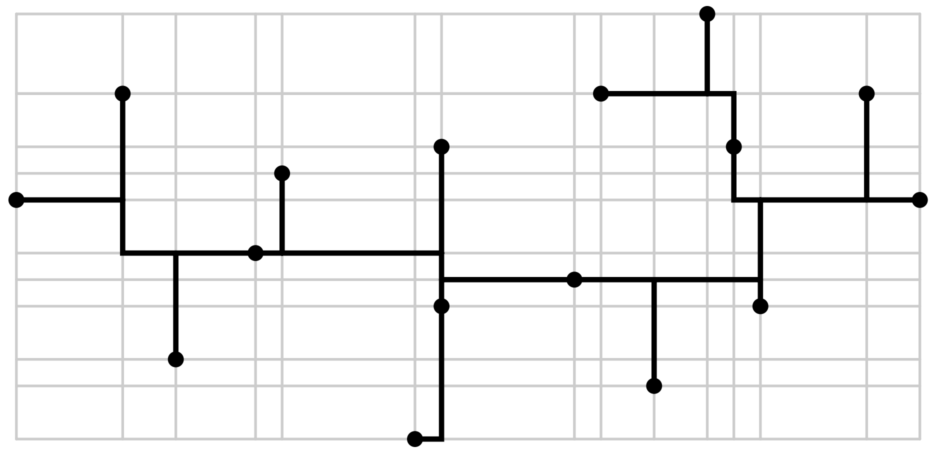
Total length of 788.472

Euclidean minimum Steiner tree



Total length of 759.996

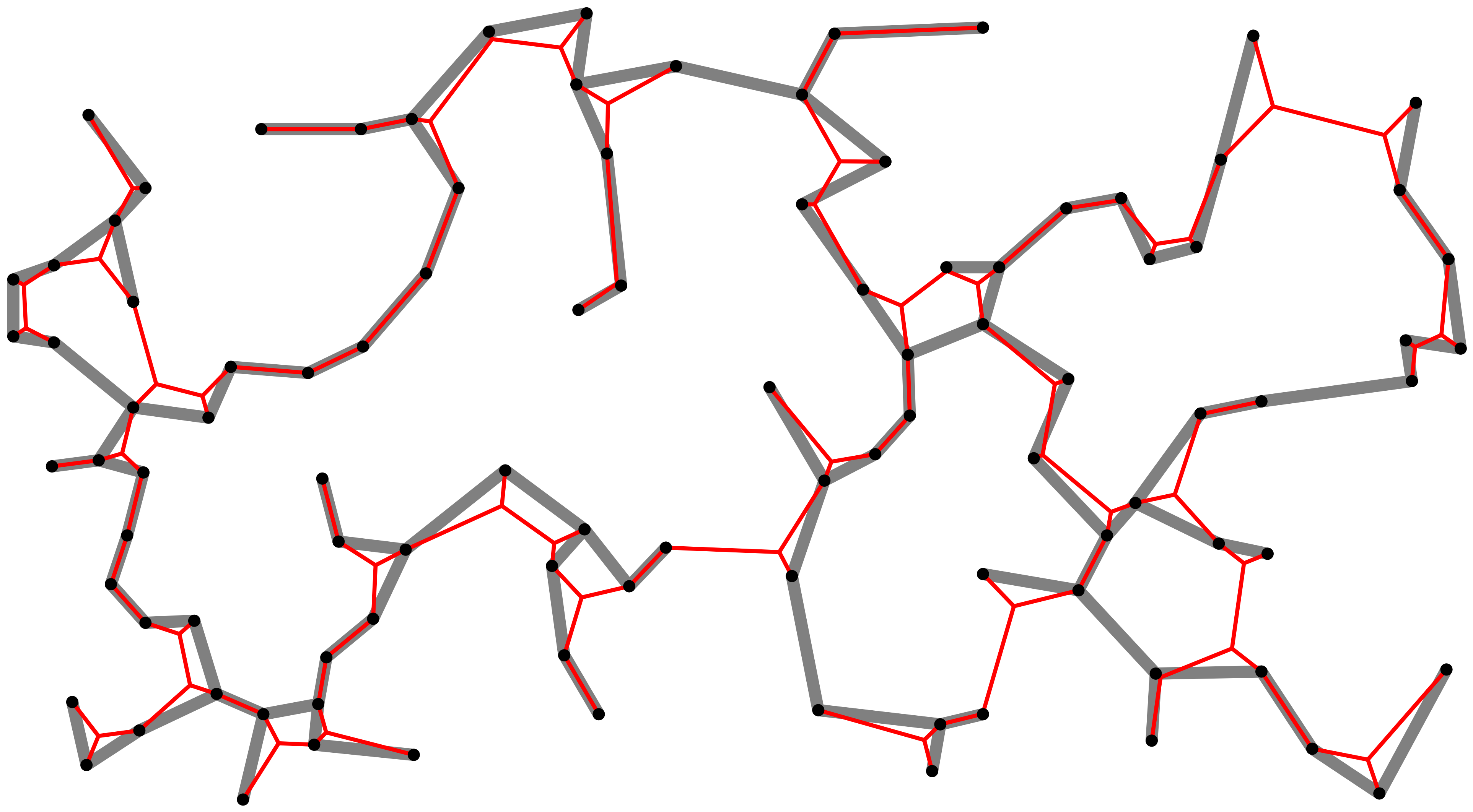
Rectilinear minimum Steiner tree



Total length of 840.000

Steiner trees ("large" example, Euclidean minimum Steiner tree)

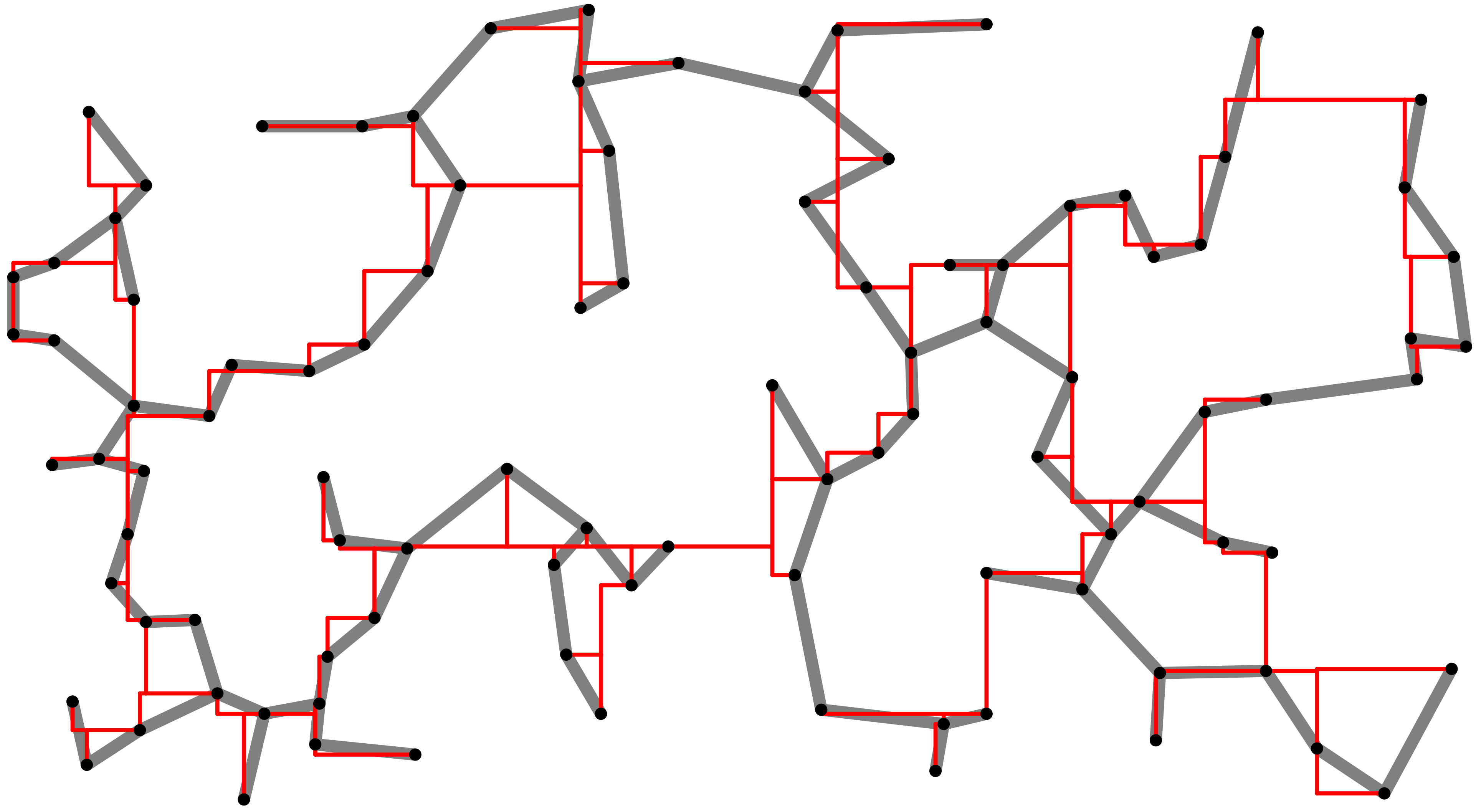
(Example made with the help of the geosteiner program)



Minimum spanning tree total length of $\mathbf{3843.871}$ Euclidean minimum Steiner tree total length of $\mathbf{3717.030}$

Steiner trees ("large" example, Rectilinear minimum Steiner tree)

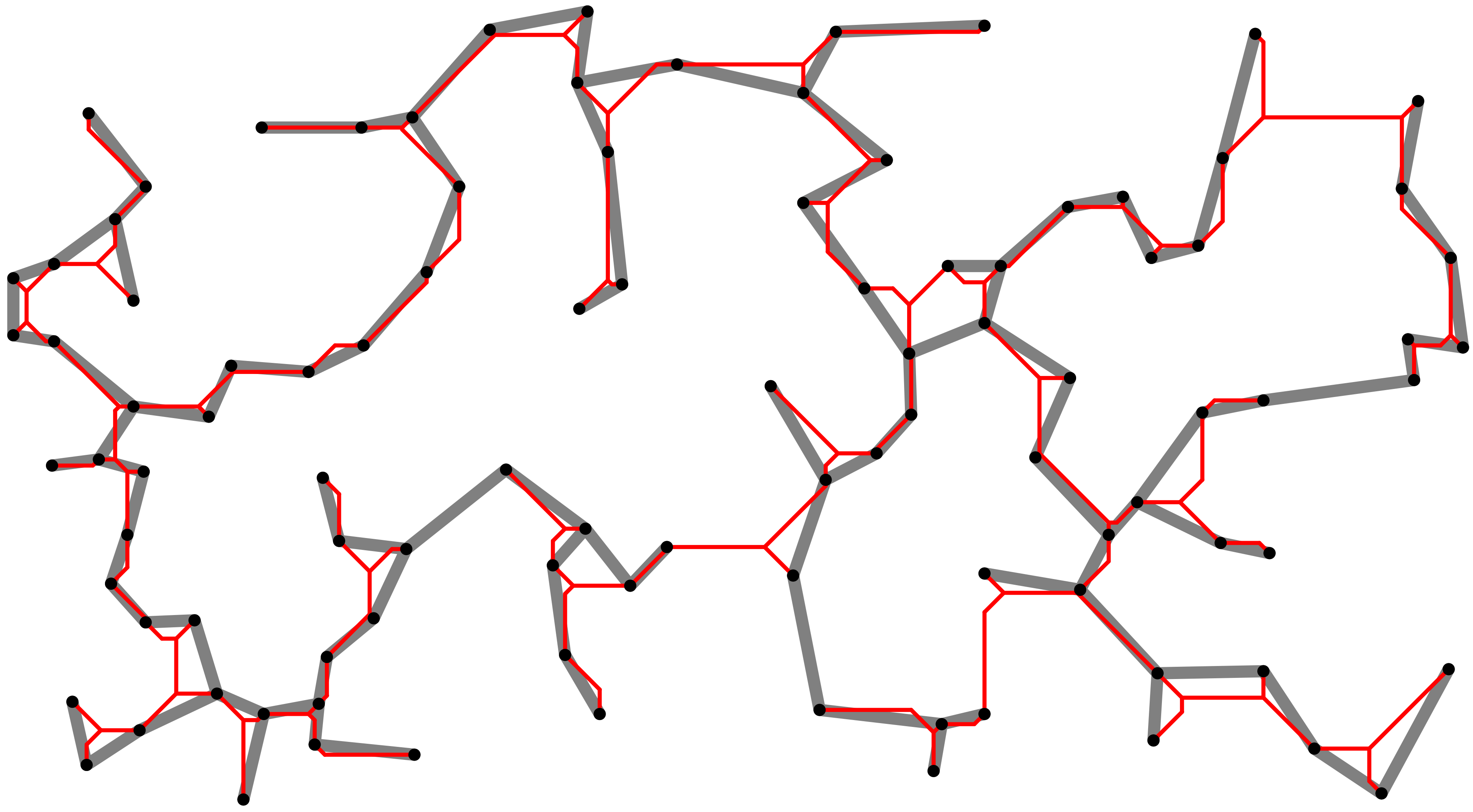
(Example made with the help of the geosteiner program)



Minimum spanning tree total length of **3843.871**   Rectilinear minimum Steiner tree total length of **4233.000**

Steiner trees (“large” example, Octilinear minimum Steiner tree)

(Example made with the help of the geosteiner program)



Minimum spanning tree total length of 3843.871 Octilinear minimum Steiner tree total length of 3847.034

Point location (part 1, problem formulation)

Problems: Given a set S of n points on a plane (or in 3-dimensional space) and another point p:

- find the point of S closer to p;
- find the k points of S closer to p; or
- find all points of S whose distance to p is smaller that d.

The obvious algorithm, trying all points of S, solves the first and third problems in $O(n)$. By sorting all distances to p the answer to the second problem can be obtained in $O(n \log n)$, irrespective of the value of k.

Is it possible to do better? After all, in the one-dimensional case a binary search solves the first problem in $O(\log n)$ (this, of source, assumes that the points are already sorted). The answer is yes. The following slides present some simple techniques that can be used to speed up point location problems.

This problem appears in many applications:

- visualization of the part of a map near a given location,
- location of the restaurant nearer to a given location,
- in a game, discovery is a given projectile is close enough to a target,
- and so on.

Point location (part 2, subdivision of space using a regular grid)

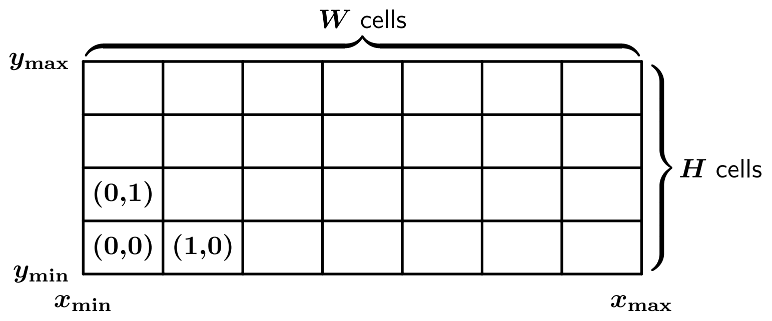
One way to speed up point location queries consists of subdividing the plane into a regular grid. Each grid cell keeps a linked list of the points that belong to that cell. For example, suppose that it is known that the x coordinates of the points satisfy $x_{\min} \leqslant x < x_{\max}$, and that the y coordinates satisfy $y_{\min} \leqslant y < y_{\max}$ (note the strict inequality in the upper bounds). Furthermore, suppose that our grid has W cells in the x direction and H cells in the y direction. The width of a cell will then be $w = (x_{\max} - x_{\min})/W$, and its height will be $y = (x_{\max} - y_{\min})/H$. A point with coordinates (x, y) will then belong to the cell with coordinates

$$\Big(\Big\lfloor \frac{x - x_{\min}}{w} \Big\rfloor, \Big\lfloor \frac{y - y_{\min}}{h} \Big\rfloor\Big).$$

Locating the nearest point would then involve looking at the list of points of the cell where the query point lies, and at the lists of nearby cells.

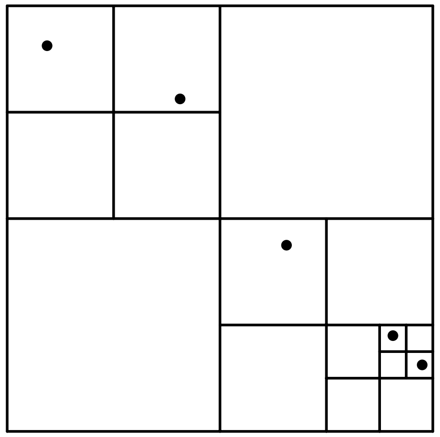
Assuming that the points are well distributed and that $wh \approx n$, finding the nearest point takes, on average and like a hash table, $O(1)$ time.

In dynamic situations, where the coordinates of the points change over time (but stay within the bounds given above), maintaining this data structure is easy: when the cell coordinates of a point change, one deletes the point from one linked list and one inserts it in another linked list. (Doubly-linked lists are better for this!)



Point location (part 3, subdivision of space using a quad-tree or oct-tree)

Another possibility consists of subdividing the space into smaller and smaller regions, in a tree-like manner, as illustrated in the following figure for the two-dimensional case.



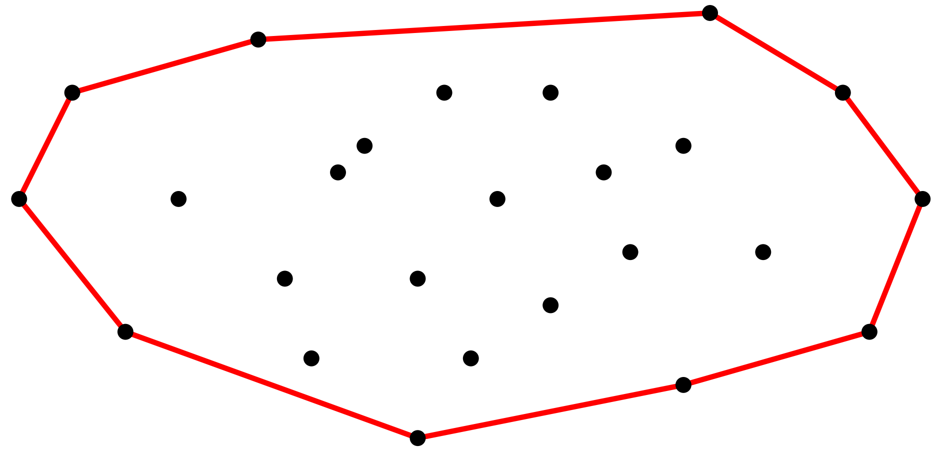
At each tree level one distributes the points that belong to a given tree branch (a rectangular region) into four subbranches (2D) or into eight subbranches (3D). The subdivision process stops when there is only one point left, in which case no further subdivision is necessary to disambiguate the points.

In two dimensions, the data structure implementing this idea is called a **quad-tree**. In three dimensions it is called an **oct-tree**.

Finding the nearest point in a quad-tree or oct-tree is not as simple as it is was using a regular space subdivision. Also, the data structure has trouble dealing with coincident points. It, however, adapts itself nicely to an arbitrary distribution of points (without using an excessive amount of memory and using a number of tree levels that is usually not too large).

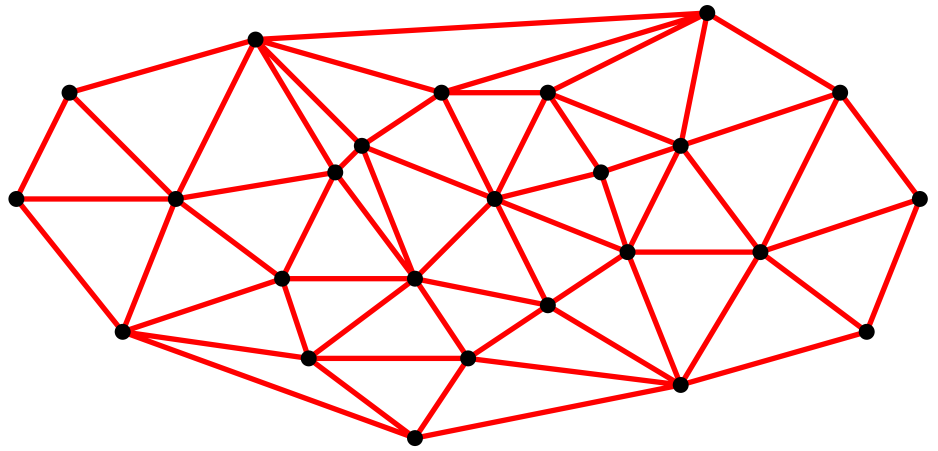
Examples (convex hull, Delaunay triangulation, Voronoi diagram)

Convex hull



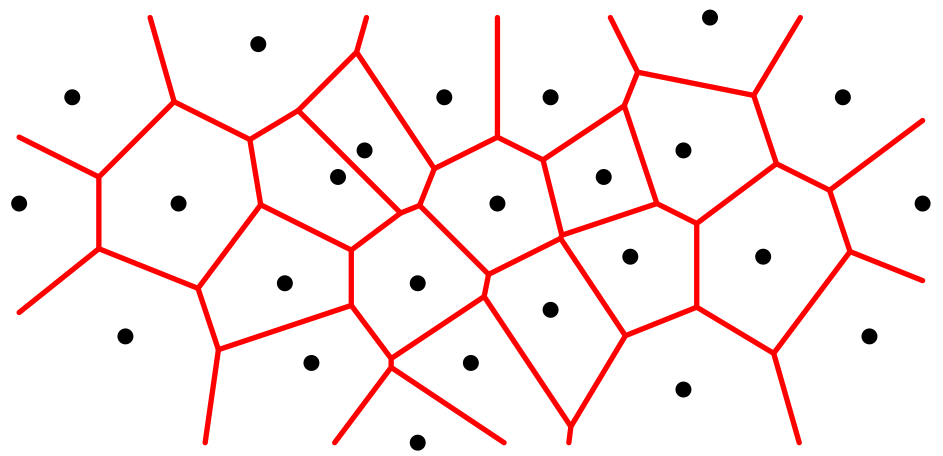
Can be computed in $O(n \log n)$ time

Delaunay triangulation



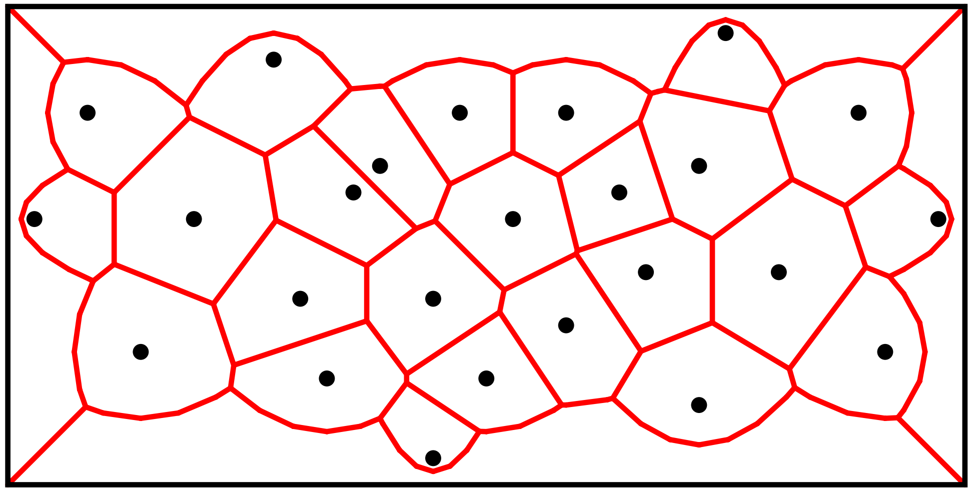
Can be computed in $O(n \log n)$ time

Voronoi diagram (points)



Can be computed in $O(n \log n)$ time

Voronoi diagram (points and line segments)



Third written report work

— P.14 —

Summary:

- Connectivity using union-find