What is an algorithm?

- Informally an *algorithm* is a well-defined computational procedure comprising a sequence of steps for solving a particular problem.

- Donald Knuth identifies the following five characteristics of an algorithm:
  - *Input*: there are zero or more quantities which are externally supplied.
  - *Output*: at least one quantity is produced.
  - *Finiteness*: terminates after a finite number of steps for all possible inputs.
  - *Definiteness*: each step must be clear and unambiguous.
  - *Effectiveness*: every step must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper.
Computational problems

- A well-defined computational problem is a pair \( P = (I,O,R) \) such that:
  - \( I \) is a specification of the set of allowed inputs
  - \( O \) is a specification of the set of outputs
  - \( R \) is a specification of the desired relation between an input and an output.

- A problem instance is given by a specific input \( i \in I \).

- Example – the sorting problem:
  - Input: a sequence \((a_1, a_2, \ldots, a_n)\), \(a_i, 1 \leq i \leq n\), elements of an ordered set
  - Output: a permutation (reordering) \((b_1, b_2, \ldots, b_n)\) of the input sequence such that \((b_1 \leq b_2 \leq \ldots \leq b_n)\).

Insertion sort

```
INSERTION-SORT(A)
1. for j ← 2, length[A] do
2.    key ← A[j]
3.   ▷ Insert A[j] into the sorted sequence A[1...j - 1]
4.   i ← j - 1
5.   while i > 0 and A[i] > key do
7.      i ← i - 1
8.   A[i + 1] ← key
```

\(2013\)
Issues in studying algorithms I

• **Correctness**
  – An algorithm is *correct* iff for all problem instances $i \in I$ it terminates and produces the correct output $o \in O$ (i.e. the pair $(i,o)$ satisfies $R$).
  – An *incorrect* algorithm either does not terminate or terminates and produces a wrong output for at least one input.

• **Design**
  – Various design techniques that often yield good algorithms have been established: recursion, divide and conquer, dynamic programming, a.o.

Issues in studying algorithms II

• **Specification**
  – An algorithm can be specified in various ways using: natural languages, informal pseudo-codes or formal specifications, programming languages.

• **Analysis**
  – Evaluates the amount of resources (processor time, memory) required by an algorithm.

• **Testing**
  – Debugging (error correction) and profiling (deriving performance profiles of algorithms).
Pseudo-code conventions

- Indentation indicates block structure.
- Conditional and looping statements are similar to Pascal.
- The symbol ▶ starts a comment.
- Multiple assignments $i \leftarrow j \leftarrow e$ have a semantics similar to C.
- Variables are local unless otherwise is explicitly specified.
- Compound data are organized as objects or structures. For an object $x$ the field $f$ is specified as $f[x]$.
- An object variable is a reference (or pointer) to the object (similar to Java).
- Parameters are passed by value.

Algorithm analysis

- Estimates the amount of resources required by an algorithm:
  - Execution time
  - Memory
- We assume the execution model of the Random-Access Machine (RAM hereafter): one processor + memory + sequential execution.
- Execution time depends on:
  - Input size (i.e. 4 elements versus 10000 elements)
  - Input itself (the degree of sort-ness: input partially sorted). The input size is problem dependent: no.of elements, no.of bits, a.o.
- Execution time = no. of primitive operations or executed steps.
- A step should be independent of a specific computer. We assume that the time for executing a pseudo-code line on the RAM is constant.
Analysis of insertion sort

```
INSERTION-SORT(A)
1. for j ← 2,length[A] do
2.     key ← A[j]
3.     // Insert A[j] into the sorted sequence A[1...j-1]
4.     i ← j - 1
5.     while i > 0 and A[i] > key do
7.         i ← i - 1
8.     A[i + 1] ← key
```

Execution time: $T(n) = c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8 (n - 1)$

$n = \text{length}(A)$ and $t_j$ is the number of executions of the test of the while loop in line 5.

Worst, average and best cases

Let $P = \langle I, O, R \rangle$ be a problem and $A$ be an algorithm for solving $P$. Let $T(A(i))$ the execution time for executing $A$ on instance $i \in I$. Taking into account the contents of the input data we can define three cases:

i) **Worst.** $T^A_w(n) = \sup\{T(A(i)) | i \in I, |i| = n\}$

ii) **Best.** $T^A_b(n) = \inf\{T(A(i)) | i \in I, |i| = n\}$

iii) **Average.** $T^A_a(n) = \frac{\sum_{i \in I, |i|=n} T(A(i))}{|\{i \in I, |i|=n\}|}$

Most often we prefer the worst case because:

- It is an upper bound for the execution time for all inputs
- It occurs frequently
- The average case is very often as bad as the worst case
**Worst, average and best cases for insertion sort**

**Best case** = the input array is already sorted, i.e. \( t_j = 1 \) for all \( j = 2, 3, \ldots, n \). \( T_b(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \)

**Worst case** = the input array is sorted the other way around, i.e. \( t_j = j \) for all \( j = 2, 3, \ldots, n \). \( T_w(n) = c_1n + c_2(n-1) + c_4(n-1) + c_5(n(n+1)/2-1) + c_6n(n-1)/2 + c_7n(n-1)/2 + c_8(n-1) = (c_5/2 + c_6/2 + c_7/2)n^2 + (c_1 + c_2 + c_4 + c_5/2 - c_6/2 - c_7/2 + c_8)n - (c_2 + c_4 + c_5 + c_8) \)

**Average case** = we have to check half of the elements of \( A[1 \ldots j - 1] \), so \( t_j = j/2 \). \( T_a(n) \) still is a quadratic function of \( n \).

---

**Asymptotic analysis**

- In analyzing the execution time we have already made a first assumption:
  - the real cost of an instruction is independent of the underlying machine.
- Moreover, we have noticed that the execution time was a function \( an^2 + bn + c \), so the values of the constants \( c_i \) can be abstracted away as well.
  - Second assumption = execution time is a polynomial of degree 2.
- In what follows we make a third simplifying assumption:
  - We shall only be interested in the growth of the execution time. This can be expressed with the \( \Theta \) notation. Using this notation the worst case execution time of insertion sort is \( \Theta(n^2) \).
  - The growth is determined by considering only the dominant term in the expression of \( T(n) \) and then ignoring the \( a \) constant.
  - We can simply say that the execution time is quadratic.
Recursion

- Recursion = defining a concept with a definition that refers directly or indirectly to the concept itself.
- Examples:
  - A *royal person* is either a monarch or a descendant of a royal person.
  - A *Canadian citizen* is either a person born in Canada, a person that obtained the citizenship after emigrating to Canada or a child of a Canadian citizen.
  - An *arithmetic expression* is either a constant, a variable, an arithmetic expression between parentheses or two arithmetic expressions separated by an operator.
- A *recursive definition* contains non-recursive definition rules and recursive definition rules. We have simple recursion, double recursion, depending on the number of references of the defined concept in a recursive definition rule.
- In programming we have *algorithm recursion* and *data recursion*.

Recursion illustrated
Designing algorithms by divide and conquer

- An algorithm design technique that uses recursion is *divide and conquer*:
  - *Divide* the problem into smaller sub-problems.
  - *Conquer* the sub-problems by recursion if they are small.
  - *Combine* the solutions to the sub-problems into the solution of the original problem.

- We can apply divide and conquer to sorting:
  - *Divide* the input sequence into 2 sub-sequences.
  - *Conquer* the sub-sequences by sorting them recursively.
  - *Combine* the sorted sub-sequences into the final sorted sequence.

- Merging = combining 2 sorted sequences into a sorted sequence. If their total length is \( n \) then this takes \( \Theta(n) \) time.

- In *merge sort* dividing is splitting and combining is merging.
- Note that other divide and conquer sorts are possible.

**Merge sort**

\[
\text{MERGE-SORT}(A, p, r) \\
1. \quad \textbf{if} \ p < r \ \textbf{then} \\
2. \quad q \leftarrow \lfloor (p + r)/2 \rfloor \\
3. \quad \text{MERGE-SORT}(A, p, q) \\
4. \quad \text{MERGE-SORT}(A, q + 1, r) \\
5. \quad \text{MERGE}(A, p, q, r)
\]

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{if } n > 1 
\end{cases}
\]

Solving this recurrence we get \( T(n) = \Theta(n \lg n) \).

For merge sort the best, average and worst case execution time is \( \Theta(n \lg n) \).
The height of this tree is $\log n$.
The execution time at each tree level is $\Theta(n)$.
So the total execution time is $\Theta(n \log n)$.
Analysis of divide and conquer algorithms

• The execution time can be described using a recurrence which describes the overall running time on a problem of size $n$ in terms of running time on smaller inputs.

• Let $T(n)$ be the execution time on a problem of size $n$. If $n$ is sufficiently small, i.e. $n \leq c$ for some constant $c$ the straightforward solution takes a constant time $\Theta(1)$.

• Let $D(n)$ be the time to divide the problem into $a$ subproblems each of which is $1/b$ in size of the original.

• Let $C(n)$ be the time to combine the solutions to the subproblems into the solution to the original problem.

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = c \\
\alpha T(n/b) + D(n) + C(n) & \text{otherwise}
\end{cases}
\]

Growth of functions

• We can assign algorithms to complexity classes based on the asymptotic growth of their worst case execution time on the input size.

$\Theta(g(n)) = \{f(n) | \exists c_1, c_2 \text{ and } n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$

$O(g(n)) = \{f(n) | \exists c \text{ and } n_0 > 0 \text{ such that } 0 \leq f(n) \leq c g(n) \text{ for all } n \geq n_0\}$

$\Omega(g(n)) = \{f(n) | \exists c \text{ and } n_0 > 0 \text{ such that } 0 \leq c g(n) \leq f(n) \text{ for all } n \geq n_0\}$

$f(n) \in \Theta(g(n)) \iff (f(n) \in O(g(n))) \land (f(n) \in \Omega(g(n)))$

• Read chapters 1 and 2 from the textbook.
Other algorithms

- **Selection sort:** find the smallest element in $A[1...n]$, move it to the first position, find the second element in $A[2...n]$, move it to the second position, ... . Evaluate the execution time as well.
- **Binary search:** search a value $x$ in a sorted array $A[1...n]$ in time $\Theta(lgn)$.
- **Compute $x^n$ with divide and conquer** in time $\Theta(lgn)$.
- **Check if an array $A[1...n]$ contains at least two equal elements** in time $\Theta(n\lg n)$.
- **Consider an array $A[1...n]$ such that $A[i] \in \{1, \ldots, n\}$. Find an algorithm that checks non-destructively if $A$ contains at least two equal elements and that takes execution time $\Theta(n)$ and $\Theta(1)$ memory.

Brief Review of Standard C

- Our focus is Standard C (ANSI C) to assure program portability.
- C is the basis for understanding more complex languages: C++, Java, C#.
- A C program is a set of functions. A source file may contain one or more functions. A C program is built from one or more C source files.
- There is a single function called `main()`. This defines the program’s entry point – where the program’s execution starts.
- A function has:
  - a *signature* (also known as *prototype*) that states the types of arguments and the return type; it is given by a program statement called *declaration*.
  - a *definition* that defines the processing steps of the function (i.e. function body).
- The allowed prototypes of function `main()` are:
  ```c
  int main()
  int main(int argc,char *argv[])
  ```
- Variables are either:
  - *local* (i.e. they are defined at the beginning of a block – a thing that starts with { and ends with } ) or
  - *global* (i.e they are defined outside any block).
  Note that a function body is a block, so this applies to functions as well.
Control flow statements in C

- **Sequencing:**
  
  ```c
  stmt ; stmt ; ...
  ```

- **If-else:**
  
  ```c
  if ( expr )
  stmt
  else
  stmt
  ```

- **Switch:**
  
  ```c
  switch ( expr ) {
    case const-expr : stmts
    case const-expr : stmts
    ...
    default : stmts
  }
  ```

- **While:**
  
  ```c
  while ( expr )
  stmt
  ```

- **Do-while:**
  
  ```c
  do
  stmt
  while ( expr ) ;
  ```

- **For:**
  
  ```c
  for ( expr1 ; expr2 ; expr3 )
  stmt
  ```

- **Block:**
  
  ```c
  {
    stmt ;
    stmt ;
    ...
    stmt ;
  }
  ```

Data Types in C

- **Scalar types:**
  - arithmetic types
    - integral types: char, short, int, long
    - floating point types: float, double
  - pointer types: \( T^* \)

- **Aggregate types**
  - struct type
  - union type
  - array type: \( T[\] \)
  - function type: \( T() \)

- **void** type

- Note that integral types may be signed or unsigned.
- Note that arithmetic and void types are also known as basic or primitive types. They are the basic building blocks for the rest, derived types.