Overview of the Course

What is an Algorithm?

Cormen, Lieserson, Rivest [1] defines the notion of an **algorithm** as follows:

> An ***algorithm*** is any well-defined procedure that takes some well-defined value or values as *input* and produces some value or set of values as *output.* An ***algorithm*** is a set of steps that *transform* the input into the output.

From Wikipedia[1]:

> . . . an ***algorithm*** is an effective method for solving a problem using a finite sequence of instructions.

(Note from Alex: this "definition" is subject to controversy as it includes the word "effective" whose meaning needs to be further clarified.)

A separate Wikipedia page[2] declares:

> There is no generally accepted definition of *algorithm.* Over the last 200 years the definition has become more complicated and detailed as researchers have tried to pin down the term. Indeed there may be more than one type of "algorithm". But most agree that algorithm has something to do with defining generalized processes for the creation of "output" integers from other "input" integers – "input parameters" arbitrary and infinite in extent, or limited in extent but still variableby the manipulation of distinguishable symbols (counting numbers) with finite collections of rules that a person can perform with paper and pencil.

[1] http://en.wikipedia.org/wiki/Algorithm
[2] http://en.wikipedia.org/wiki/Algorithm_characterizations

The same page quotes Andrey A. Markov's description of an algorithm:

> The following three features are characteristic of algorithms and determine their role in mathematics:
>
> 1. the precision of the prescription, leaving no place to arbitrariness, and its universal comprehensibility – the definiteness of the algorithm;
> 2. the possibility of starting out with initial data, which may vary within given limits – the generality of the algorithm;
> 3. the orientation of the algorithm toward obtaining some desired result, which is indeed obtained in the end with proper initial data – the conclusiveness of the algorithm.

Donald Knuth identified *five properties* that are accepted as requirements for algorithms:

> **Finiteness.** "An algorithm must always terminate after a finite number of steps... a very finite number, a reasonable number"
>
> **Definiteness.** "Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case"
>
> **Input.** "...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects"
>
> **Output.** "...quantities which have a specified relation to the inputs"
>
> **Effectiveness.** "... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil"

Computational Models: Turing Machines

The notion of an algorithm is often associated with a notion of a **computational model**, i.e. a (*real* or *imaginary* device) that is used to perform the instructions of the algorithm to solve the specified problem.

In theoretical computer science, the main *computational model* is **Turing Machine**: an *imaginary* computational device that cosists of:

- One or more *infinite memory tapes*;
- A Read, Write or Read/Write head associated with each tape;
- A **state** selected from a finite collection;
- A **program** that specifies actions the machine has to take given the current state, and information observed by the Read/Read/Write heads on the tape(s).

Turing Machines are:

- **Imaginary.** No such devices physically exist (in fact, building a Turing Machine with an infinite tape is impossible).
- **Feature-poor.** Turing Machine programs consist of very simple actions: moving heads one position left/right, replacing current content on a tape with new content.
- **Powerful.** If a problem has an algorithms for solving it using a Turing Machine, then an algorithms for solving the problem can be found for **other traditional computational models**[3], and vice versa.
- **Convenient for theory.** Because Turing Machines are feature-poor, proving statements about correctness of algorithms and *computability* (i.e., existance of algorithms for certain problems) is easier using Turing Machines as the computational model.
- **Inconvenient in practice.** The Turing Machine model captures the nature of a computational device, but both the "hardware" and the "programming language" (i.e., the language of actions) are primitive and provide little intuition on how to actually solve problems using real-life computational models (i.e., computers).

Computational Models: Computers

In this course we study how to **design** and **implement** efficient algorithms that would work on the computing devices that are currently in use: **computers**.

Our descriptions of algorithms must be:

- **Easy-to-understand.** You must be able to observe the description of an algorithm and understand what actions are performed, and what effects these actions will have.
- **Easy-to-implement.** You must be able to create actual programs in existing programming languages[4] in a relatively straightforward manner, after having read and understood the descriptions of the algorithms.
- **Precise.** Following Knuth, the descriptions we provide should "leave no place to arbitrariness"....

To achieve this, we must select a **computational model** that is close enough to actual computers on which algorithms will be implemented, but *not as complex as a real computer is.*

Our computational model has the following features:

- **A single processor.** A "processor" is a device repsonsible for performing instructions.

[3]Except for situation when a computational model has been purposefully chosen to restrict what computations are possible on it.

[4]The course uses Java as the language of choice, but the statement is true w.r.t. any existing programming language.

- **Infinite Random-Access Memory.** The amount of memory available for any computation is not limited. Each memory cell has a unique address and can be accessed at any time.
- **High-level instruction set/programming language.** All algorithms are recorded in the language pseudocode instructions: a high-level instruction set, which abstracts low-level memory manipulation (individual cell access) and the elementary operations performed by the processor, as well as the actual processor architecture (e.g., the existance of registers).
- **Integers-only computations.** For most of the course, our algorithms will operate solely on integer numbers. Thus, `integers` will be the only data type accepted by our computational model, *except where explicitly specified.*

Pseudocode

We use the following language of instructions, which we call **pseudocode** in the rest of the course.

Constants. We only have `integer` constants. Most computations will involve *decimal numbers*, occasionally, *binary* or *hexadecimal* numeric (integer) constants will be used.

Examples: 10, -1, 34, 776, 10, 1101.

Variables. We have *simple variables* representing a single `integer` value and *array variables* representing finite sequences of `integer` values.

Simple variable names follow traditional programming language conventions: they start is a character in the Latin alphabet (`'a'...'z','A',...,'Z'`), followed by a finite sequence of Latin alphabet characters, digits and underscores.

Examples: a, AB1, FooBar, D_23.

Array variable names follow the same convention, but must be followed by an *index expression*: an expression (see below) identifying the range of array indexes, or a single array index enclosed in *square brackets.*

Examples: a[1], B[n], X[1..N].

Simple variables need not be *declared* in pseudocode programs. Array variables may be declared using a statement that specifies the name of the array variable and the range of array indexes. For example

```
B[1..2];
M[1..N];
```

are two array declarations. The first declares an array B with two elements indexed 1 and 2. The second declares and array M with elements indexed 1 through N. We expect that the value of N is known at the time of the declaration.

Multidimensional arrays may also appear. They are declared and referred to in a straightforward manner:

Declaration (2D array):

```
B[1..M][1..N];
```

Array element reference (2D array):

```
B[i][j];
```

Expressions. A constant (e.g. 10), a simple variable name (a) and a variable name followed by a single index (f[3]) are all expressions. If $E1$ and $E2$ are expressions, then the following are also expressions:

```
E1 + E2      (E1 + E2)
E1 - E2      (E1 - E2)
E1 * E2      (E1 * E2)
E1 / E2      (E1 / E2)
E1 mod E2    (E1 mod E2)
 -E1         (-E1)
```

Statements. The pseudocode programs use the standard range of statements: assignment, conditional statements, loops, procedure calls.

Assignment statement. We use two different symbols for the assignment: ← and :=. These can be used interchangably, although, typically we will use the same assignment symbol throughout entire algorithm. The syntax of the assignment statement is:

VariableName := *Expression*;

or

VariableName ← *Expression*;

The **effect** of the assignment statement is as expected: the value of an expression is computed and assigned to the variable VariableName.

Multiple assignments. The textbook uses the

VariableName1 ← VariableName2← . . . ←VariableNameN← *Expression*;

to represent:

```
VariableNameN ← Expression;
...
VariableName2 ← VariableName3;
VariableName1 ← VariableName2;
```

Boolean Expressions. Boolean expressions appear in conditional statements and loops. If $E1$ and $E2$ are two expressions, then the following are boolean expressions:

E1 =E2	(E1 = E2)
E1 < E2	(E1 < E2)
E1 > E2	(E1 > E2)
E1 ≤ E2 (or E1 <= E2)	(E1 ≤ E2) (or (E1 <= E2))
E1 ≥ E2 (or E1 >= E2)	(E1 ≥ E2) (or (E1 >= E2))
E1 ≠ E2 (or E1 ! = E2)	(E1 ≠ E2) (or (E1 ! = E2))

If $B1$ and $B2$ are two boolean expressions, then the following are also boolean expressions:

B1 and B2	(B1 and B2)
B1 or B2	(B1 or B2)
not B1	(not B1)

Boolean expressions are evaluated using **short-circuiting**.

Conditional Statements. Pseudocode has two types of conditional statements: the if-then and the if-then-else statements. Indention is commonly used to specify the scope of the if and else clauses. Alternatively endif may be used to indicate where the scope of the clauses ends:

```
if BooleanExpression then
    statements
endif
```

or

```
if BooleanExpression then
    statements
else
    statements
endif
```

Loops. We will use three types of loops in our pseudocode:

1. **Counting loops.** The for-to-do loop:

```
for Assignment to Expression do
    statements
endfor
```

Here *Assignment* is an assignment statement which initializes the *loop counter*. For loops have the increment of 1 by default. If other increments are necessary, the appropriate syntax will be introduced.

2. **Loops with precondition.** The while-do loop:

```
while BooleanExpression do
    statements
endwhile
```

3. **Loops with postcondition.** The repeat-until loop:

   ```
   repeat
       statements
   until BooleanExpression
   ```

   Notice, that the loop will continue for as long as the *BooleanExpression* evaluates to **false**.

Procedure/Function/Algorithm declarations. Procedure, function and algorithm declarations in pseudocode are straightforward:

Procedure declaration:

```
PROCEDURE Name ([Parameters])
begin
 statements
end
```

Function declaration:

```
FUNCTION Name ([Parameters])
begin
 statements
end
```

Procedure declaration:

```
ALGORITHM Name ([Parameters])
begin
 statements
end
```

Here, *Parameters* is a list of variable names (each variable name is either a simple variable or an array variable with array boundaries).

Functions and algorithms **return values**. Because pseudocode operates with `integers` only, return types for functions/algorithms are not explicitly specified. Both individual integer values and arrays of integers can be returned.

Return statement. Functions and algorithms return values using **return** statements. The syntax is:

```
return Expression
```

Occasionally, we might use a simple **return** (w/o the return values) inside procedures.

Procedure calls. A procedure call is simply a name of a procedure followed by the values of all parameters in parentheses:

```
ProcName(ParameterValues)
```

Examples.:

```
Initialize()
FindMin(A,B,C)
```

Comments. Comments in pseudocode are shown using either `//` (single line comment) or `/* ...*/` (multiline comment) notation. The textbook uses the ▷ symbol, which we may use on occasion as well.

Examples. Here are some examples of comments.

```
// Let us start the computation
A← 0   // Initialize A
/* This is a placeholder
  for the main loop */
return A    ▷ return the computed value
```

Variable scope. All variables have local scope within any procedure/function/algorithm. There are **no** global variables.

Use of semicolumns. I tend to use semicolumns (";") to separate statements in pseudocode. The textbook does not.

Algorithm example: Finding the largest number in an array

Computational Problem. Given a list of numbers, find and output the largest one.

Solution Idea. Use an array to represent the list of numbers. Scan the array and compare each number in it to the currently found maximum. If a larger number is discovered, update the maximum.

Algorithm.

```
ALGORITHM FindMax(N, A[1..N])
begin
  tmpMax← $A[1]$;
  for i ← 2 to N do
    if tmpMax< $A[i]$ then
      tmpMax← $A[i]$;
    endif
  endfor
  return tmpMax;
end
```

Correctness: Consider two cases: $N = 1$ and $N > 1$.

$N = 1$. If $N = 1$ then the **for** loop won't get executed. tmpMax will be assigned the value of $A[1]$, which will be returned. $A[1]$ is the largest element in array A.

$N > 1$. Consider the work of FindMax on inputs $N, A[1..N]$, where $N > 1$. Let $A[k]$ be the largest element of A for some $1 \leq k \leq N$. We consider two cases:

- $k = 1$. In this case, tmpMax will be assigned the value of $A[1] = A[k]$ before the **for** loop starts. Because, $A[k]$ is the largest value in A, tmpMax is greater than or equal to any of the $A[2], \ldots A[N]$, and therefore the comparison in the **if** will **always** be false. Therefore, at the end of the **for** loop, tmpMax $= A[1] = A[k]$.
- $k > 1$. Consider step $k - 1$ of the **loop** in which the current value of tmpMax is compared to $A[k]$. tmpMax can take as its value either $A[1]$ or some $A[i]$ for $i < k$ (but NOT any other values). That is, any value of tmpMax up to step $k - 1$ is *an element of the array* A. But $A[k]$ is the largest element in A, therefore, after step $k - 1$:
    1. The value of tmpMax will become $A[k]$.
    2. The value of tmpMax will not change until the end of the **for** loop.

Computational Complexity. We will measure the effort spent executing this algorithm in terms of *number of comparisons* executed during the runtime of the algorithm.

For $N > 1$, the **for** loop gets executed $N - 1$ times. On each step of the loop, the comparison in the **if** statement is made. Therefore, Algorithm FindMax will **always** use $N - 1$ comparisons.

Lower bound on complexity. The problem of finding the largest element of an array has a *lower bound on computational complexity* in our computational model (number of comparisons):

> Any algorithm computing the largest number in an array of N numbers **must** use **at least** $N - 1$ comparisons.

Proof. Consider an algorithm that solves the problem of finding the largest element of an array of size N which uses K comparisons. Consider a graph whose nodes are elements of the array and whose (directed) edges are pairs of elements that were compared by the algorithm during a specific run. The arrow goes from the smaller to the greater element. *The graph will contain K edges. If $K < N - 1$, then the graph* ***will not be connected.***

Divide-and-conquer version of FindMax. Various algorithmic problems accept multiple solutions: sometimes, even multiple "optimal" solutions. Algorithm FindMax described above, can be considered **greedy**: on each step a *locally largest number* is determined.

Another possibile approach is the *divide-and-conquer* technique, which breaks the solution of a problem into the solution of one or more problems of the same type *on smaller inputs.* The classical *divide-and-conquer* algorithm for finding the maximum in an array works as follows:

1. Divide your input array into two parts as equally as possible.
2. Find the maximum element in each of two parts.
3. Compare the two maximum elements.

This gives rise to the following **recursive** algorithm:

```
ALGORITHM FindMaxRecursive(I,J, A[I..J])
begin
  if I = J then return $A[I]$;   //base case
  max1← FindMaxRecursive(I, I+(J-I)/2, A);
  max2← FindMaxRecursive(1+I+(J-I)/2,J, A);
  if max1>max2 then return max1
  else return max2;
end
```

Note. To understand how FindMaxRecursive works, imagine a tournament played between a number of teams. Each time two teams play a game, one team advances. Eventually, two teams reach the final, and one team wins. If we replace teams with numbers, and "winning a game" with "being greater than", then we get the FindMaxRecursive algorithm.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1990, 1st Edition.