Computing basics

Ruurd Kuiper

October 29, 2009

Overview (cf Schaum Chapter 1)

Basic computing science is about using computers to do things for us. These things amount to processing data. The way a computer processes data, is by executing programs of algorithmic nature. Algorithms are recipes, in terms of a basic statements for simple, one step, transformation of data into new data, and control statements to order the steps. A programming language provides basic statements and control statements to write programs that make the computer perform such tasks - this particular issue is addressed in the lectures and exercise classes for Java programming that form the main part of this course. For this practical part the lecture notes **Object Oriented Programming – with Java, Par I**, with the accompanying exercises, by Kees Huizing and Ruurd Kuiper, are used.

Other, more conceptual issues remain that are relevant for a basic understanding of computer science. These are addressed in the present lectures and handouts, using the book **Principles of computer science**, by Carl Reynolds and Paul Tymann, Schaum's Outline Series, MacGraw-Hill, 2008.

The following issues are addressed.

1. What is a computer?
2. What can a computer (not) do?
3. What can a computer (not) do efficiently?
4. How do we get a computer to do things?
5. How do we know that a computer does (not) what we want?
6. What should we (not) use a computer for?

Chapter 1

What is a computer? (cf Schaum Chapter 2,3, 4)

A computer is a machine that can be programmed to perform a data processing task, i.e., storing and retrieving data values, and transforming data values to new data values using algorithms. A program, written in a programming language like for example Java, for a specific data processing task consists of the instructions to the computer how to perform that task; this is called software. The machinery on which the program is executed, i.e. is turned into activity, is called hardware.

1.1 Turing machines (Schaum pp 25 – 28)

A computer processes data, by executing programs of algorithmic nature. This amounts to executing basic statements, in an order prescribed by control statements. As programs are written by humans, the statements in a programming language are human-understanding oriented to a considerable extend – and also may vary among programming languages.

To understand what the essence of algorithmic data processing is, the Turing machine (TM) (introduced in 1936 by Alan Turing) is used: a conceptual computer with a fixed program and arbitrary input.

A TM consists of the following.

1. A tape, divided into cells that each contain a symbol or a blank, Δ, from a finite set. The tape is infinite to the right.
2. A head that reads and writes one symbol (possibly a Δ) in the cell under it and can move left, right or be stationary.
3. A finite number of, numbered, states and a special halt state.
4. A finite instruction table of five-tuples: (current state, current symbol, overwriting symbol, next state, direction to move).

The TM overwrites the cell the head is over and moves the head over the tape according to the instructions. A TM starts in state 1, positioned at the extreme left of the tape, and if it halts, it halts in the halt state.

The program in this version of the Turing machine is the instruction table. Programming such a Turing machine for a specific task means writing the corresponding instruction table. Note, that different input can be given by putting different values on the tape.

To bring such a Turing machine intuitively closer to a programmable computer, it is desirable to be able to write the instructions (the "program") for a specific task on the tape rather than to have to change the instruction table of the Turing machine. It is possible to define a so called "universal" Turing machine, with a fixed, "universal", instruction table, and input the instructions for a specific task on the tape. The idea is, that part of the instructions in the fixed instruction table transform the specific instructions on the tape in such a manner, that the universal Turing machine behaves just like a Turing machine that would have the instructions on the tape as its instruction table.

The feeling that a Turing machine captures the essence of computability is phrased in the Church-Turing thesis as: If something can be effectively computed, i.e., there exist a stepwise, always terminating, recipe to do the calculation for any input, than a Turing machine can be constructed that does this.

NB This can only be a thesis (a "belief" not a, provable, theorem), as the notion of effective computability is an intuitive and not a formal one.

Furthermore, a realistic programming language can do what a Turing machine can: it is Turing equivalent. To support this claim, observe that in a realistic programming language, it is possible to program a Turing machine.

The claim (following from the Church-Turing thesis) the other way around can be supported by showing that all basic operations of a language as executed by a computer can be modeled by a Turing machine.

1.2 Hardware (cf Schaum Chapter 3)

The Von Neumann architecture reflects the (Universal) Turing Machine structure: the memory corresponds to the tape, the CPU corresponds to the instruction table; the stored program corresponds to the instruction put on the tape and the interpretation rules being present in the CPU.

1.3 Software (cf Schaum Chapter 4, 6 pp 95 – 98)

The high level language basic and control statements reflect the Turing machine instructions.

The machine language basic and control statements reflect the Turing machine instructions.

Translation from high level language into machine instructions or an intermediate format (Java's byte code) is provided by the compiler.

Chapter 2

What can a computer (not) do? (cf Schaum pp 28 – 29)

As argued in chapter 1, the intuitive question"what is a computer" can, according to the Church-Turing thesis, be formalized as: a computer is equivalent to a Turing machine.

Therefore, the question"what can a computer (not) do", becomes equivalent to the question "what can a Turing machine (not) do".

Theorem 2.1 (Halting problem) *There exists no algorithm that, for all Turing machines M and all inputs i decides whether M(i) halts. (Decides means that the algorithm itself halts on any input and give a yes/ no answer.)*

Proof:

Assume there exists $H(\#M, i)$ *that for all machines and all inputs decides termination:*

$H(\#M, i)$ *returns yes if M(i) terminates and returns no if M(i) does not terminate*

(I use returns yes if M(i) terminates rather then if M(i) terminates return yes to indicate that M(i) terminates is a property rather than a guard that can be evaluated.)

define $X(\#M)$ *if* $H(\#M, \#M)$ == *yes do not terminate else (i.e., if* $H(\#M, \#M)$ == *no) terminate*

If $X(\#X)$ *terminates* $\Rightarrow$ $H(\#X, \#X)$ == *no* $\Rightarrow$ $X(\#X)$ *does not terminate: Contradiction. Else (i.e., if* $X(\#X)$ *does not terminate)* $\Rightarrow$ $H(\#X, \#X)$ == *yes* $\Rightarrow$ $X(\#X)$ *terminates: Contradiction.*

So $H(\#M, i)$ *cannot exist.*

We also discussed in how far there were "realistic" problems that could not be solved algorithmically. NB This is advanced material, meant to give some idea about what the issues are and how to approach the various problems - it is not required for the exam.

Theorem 2.2 (Tiling problem) *Whether or not a given set of tiles* T *can be used to tile the upper half of an infinite plane, starting at the bottom with a specific tile, cannot be decided algorithmically.*

Proof (sketch):

A Turing machine can be constructed that translates the tiling problem to the halting problem, thus showing that also the Tiling problem cannot be decided algorithmically.

We now take a somewhat closer, more precise look at these questions.

Definition 2.1 (Language) *A language* L *over an alphabet* Σ *is a set of words* $w \in Sigma^*$*: finite strings built from letters from* Σ*.*

Definition 2.2 (Recognizing a language) *A Turing machine M* recognizes *a language L if for each word $w \in \Sigma^*$, it returns yes and halts iff $w \in L$, and it returns no and halts or it loops iff $w \notin L$.*

Definition 2.3 (Accepting a language) *A Turing machine M* accepts *a language L if for each word $w \in \Sigma^*$, it returns yes and halts iff $w \in L$, and it returns no and halts iff $w \notin L$.*

Consider the following language $A_TM = \{\langle M, w\rangle | \text{M is a Turing machine and } M(w) = yes\}$ to see, as stated in the two following theorems, that there are languages that are recognizable but not decidable.

Theorem 2.3 (Recognizability of A_TM) *There exists an algorithm that, for all Turing machines M and all inputs i decides whether M(i) halts.*

Proof:

Built the following machine. Use the (universal) Turing machine to simulate M, run it on w, if M(w) = yes return yes, if M(w) = no return no, else (i.e., if M(w) loops), loop. This machine recognizes A_TM.

Theorem 2.4 (Undecidability of A_TM) *There exists no algorithm that, for decides A_TM.*

Proof:

You can do this yourself: it is very much analogous to the proof of the Halting problem.

There are even languages that are not recognizable.

Theorem 2.5 (Non-recognizable languages) *There are not recognizable languages.*

Proof (sketch):

A set is countable if it has the same size, i.e., can be brought into 1-1 correspondence with the natural numbers.

The set of Turing machines is countable. Because: any Turing machine can be represented as a finite string - which can be regarded as a natural number.

The set $\mathcal{B}$ of infinite binary sequences is uncountable. Because of diagonalization: Assume all binary sequences can be put next to the natural numbers. Make a new binary sequence that differs in the first digit from the first one listed, in the second digit from the second one listed etc. - this new one is not in the list. Contradiction.

The set of languages over alphabet Σ is uncountable. Because the set of all languages over an alphabet Σ has the same size as $\mathcal{B}$. Namely, each language has a unique characteristic sequence in $\mathcal{B}$, as follows. The characteristic sequence $s_1, s_2, \ldots$ for language L is build as follows:

- *All words over Σ are: $\Sigma^* = \{w_1, w_2, \ldots\}$.*
- *$s_i = 1$ iff $w_i \in L$, $s_i = 0$ iff $w_i \notin L$.*

So there are more languages than Turing machines, so there are not recognizable languages.

Remark: The complement of a language L over alphabet Σ is $Sigma^* \setminus L$.The complement of A_TM is not recognizable. (You can prove this yourself, using that A_TM is recognizable and that (show!) a language is decidable iff both it and its complement are recognizable ;-)

Chapter 3

What can a computer (not) do efficiently? (cf Schaum Chapter 2)

As explained in chapter 2, there are problems that cannot be solved algorithmically. A more refined analysis shows, that there are also problems hat cannot be solved in a practical sense, because there is no efficient algorithm. Efficiency comes in two kinds: in terms of time and in terms of space.

3.1 Time complexity (Schaum pp 17 – 25)

For time complexity, the notions of computation step and order of magnitude are crucial.

3.2 Space complexity

For space complexity, the notions of memory cell and order of magnitude are essential.

Often there is a trade-off between time- and space complexity.

Chapter 4

How do we get a computer to do things? (cf Schaum Chapter 4, pp 44 –53)

There are two answers to this question: high level programming languages and software engineering.

We only consider high level language: they enable to *partition* a programming problem.

In programming, there are many situations where something is too complex to be dealt with conveniently as a whole. The general approach of partitioning can often be applied in such situations. Here a general description of the approach is given, followed by a specific application to the case of methods.

Partitioning means dividing something complex into less complex parts that can, to some extent, be treated separately. The approach is based on two ideas. For ease of explanation, the case for two parts is considered.

The first idea is *composition*: making the whole from less complex parts that put together form the whole.

To reduce complexity, making a separate part should be simpler than making the whole directly. This requires that when making one part, not all the details of the other part should have to be considered. Similar holds for putting the parts together. The second idea in the partitioning approach is therefore to make an *abstraction* of a part: a description of a part that provides only the information necessary to make the other part. Similar for putting the parts together. That such an abstraction can be made, depends on choosing relatively independent parts.

The idea of partitioning is now applied to the method, to partition a data manipulation task into subtasks.

Applying the idea of composition, we declare and define several methods, with different names, in a class, that each contain the description of a subtask. One method is, as usual, started from `main`. This starts the processing; from this method others are started, *called*, these may call other methods, and so on. A method call takes place when control arrives at a statement that consists of the method name followed by a (often empty) pair of parentheses. In this way the parts are combined.

The process of defining methods is called “grouping and naming”: grouping of code in a method that has a name by which it can be called. This code can be used repeatedly and at different places in the code by writing a call where desired.

Applying the idea of abstraction, we write a short description of the effect of the method on the instance variables as a comment just before the method declaration.

The various methods manipulate the data stored in the instance variables of the class. These variables are accessible for all methods: it is the data manipulation that is partitioned, not the data storage.

The partitioning of data storage is done by creating more than one object from a class - and also more classes.

Chapter 5

How do we know that a computer does (not) what we want?

To start with, a formal specification of the desired behavior is needed, because otherwise it is unclear what it is that we want the computer to do.

For this, think of the remarks about abstraction in chapter 4 and the remarks about verification in the lecture notes. The behavior is described using assertions, invariants and pre-/postconditions.

Then there are three main approaches to *verify* whether or not a program satisfies a specification: testing, model checking and proving.

In testing, test runs of the program, on example inputs, are performed, and the outcomes compared to the specification. Testing can show presence of errors (in test runs), but not show general absence of errors (in all runs) if infinitely many runs exist and only finitely many runs can be tried.

Model checking is appropriate if only finitely many different states occur and a finite model of the runs can be given. Then all possible runs can be checked, and thus presence as well as general absence of errors can be shown.

Proving properties is appropriate when no finite model exists. Limitations are, that no general proof algorithm can exist (because if such an algorithm existed, it would be able to decide halting, which would contradict the Halting Problem). The situation is even worse: there is no proof system that enables, even with human proof construction, to prove all properties that can be specified and are true of a program.

Chapter 6

What should we (not) use a computer for? (cf Schaum Chapter 9)

...