TDDE25 Seminar 7: Algorithms and Computability

Victor Lagerkvist

In this lecture note we give an overview to algorithms and computability. This corresponds to Chapter 5 and Chapter 12 in the course book [1] but uses different examples and introduces three models of computation in order of increasing strength rather than just presenting Turing machines. Chapter 5 uses material from the course *Introduction to algorithms*, MIT, and Chapter 12 is partially based on slides by Christer Bäckström and Gustav Nordh from TDDD14/TDDD85.

[1] J.G. Brookshear and D. Brylow. *Computer Science: An Overview*. Pearson Education, Limited, 2018. ISBN 9781292263427

Algorithms

Assume that we are working with arrays containing natural numbers. We use the notation $[a_1, \ldots, a_n]$ to denote an array with n elements $a_1, \ldots, a_n$. For example, $[0, 1, 2]$ denotes an array with the three elements 0, 1, and 2. Additionally, let us use the notation $A[i]$ to denote the ith element in an array A. We are are then interested in the following property.

Definition 1. *Let A be an array with $n \geq 1$ natural numbers. If $n = 1$ then $A[1]$ is a peak. For $n \geq 2$ and $1 \leq i \leq n$ say that $A[i]$ is a* peak *if*

1. $A[i-1] \leq A[i], A[i] \geq A[i+1]$ *(for $2 \leq i \leq n-1$)* [2]
2. $A[1] \geq A[2]$ *(for $i = 1$)* [3]
3. $A[n] \geq A[n-1]$ *($i = n$).* [4]

[2] Put simply: an element is a peak if it not strictly smaller than its left or right neighbor...

[3] ... Unless it is the first element, where we just require that it is not smaller than the second element...

[4] ... And unless it is the last element, where we just require that it is not smaller than the element to the left.

Example 1. *Consider the array $A = [0, 3, 7, 7, 1, 10]$. It contains the peaks $A[3] = 7, A[4] = 7$, and $A[6] = 10$.*

The natural problem that we want to solve is then to find a peak in a given array (note that a peak according to Definition 1 always exists). We typically refer to such problems as *computational problems* and explicitly formulate them as follows.

Peak finder
Instance: An array A with n natural numbers ($n \geq 1$).
Output: An element $A[i]$ such that $A[i]$ is a peak ($1 \leq i \leq n$)

A Simple Solution

Now, our objective is to construct an *algorithm* which solves this problem. Here, we follow the informal definition in the course book and simply view an algorithm as a sequence of sufficiently unambiguous steps. Naturally, any correct solution in a programming language

(e.g., Python) would constitute an algorithm, but it is almost always easier to begin by outlining the main algorithmic ideas in language agnostic pseudo code.

```
Scan the array from left to right
    If the current element is a peak then return it
```

Whether this pseudo code fragment is sufficient is up to interpretation. One the one hand, it is not explicitly stated exactly how we should "scan" the array, how we should access the current element, or how to test the peak condition. On the other hand, it is clear to any programmer that scanning the array can be easily implemented in any programming language by using a loop construct (e.g., a while loop or a for loop) or via a recursive function, and that the peak condition can easily be tested by a sequence of if statements mirroring the logic in Definition 1. Thus, when presenting pseudo code we typically only require that it is precise enough to in principle be converted to a more precise solution when the need arises. However, before turning to an actual implementation — which would be straightforward in this case but in general be very time consuming compared to spewing out a few lines of pseudo code — it is a very good idea to analyze the solution and see to which extent it can be improved. The solution is clearly correct since it performs the peak test over all possible elements in the array, so all that remains to verify is whether it is sufficiently fast. Here, one is typically interested how well the algorithm performs on the *worst possible* input since we then obtain a bound valid for any possible type of input[5]. For our algorithm it is easy to see that an array of the form $[1, 2, 3, \ldots, n]$ is going to require roughly n arithmetical operations, meaning that we in the worst case always have to consider every element in the array. We sometimes say that an algorithm of this form requires *linear* time.

[5] However, in this case the so-called average case is also not difficult to compute.

A Faster Solution Using Divide and Conquer

It certainly feels like the algorithm in the previous section cannot be improved substantially. However, intuition is not always correct, and we can in fact construct a solution which is *much* faster. To achieve this we will apply the principle of *divide and conquer*: split the original problem into subproblems and solve the general problem by using the solution of one or more subproblem. This strategy is perhaps best known in the context of *binary search* where one splits a sorted input array in the middle and then decides whether to continue searching in the left or right part depending on the size of the current element. However, a very important difference in the current setting is that we do not assume that the input is sorted. Despite this, let us consider

a divide and conquer approach based on comparing the middle element to the elements to the left and right. Here, it is important to keep in mind that a peak *always* exists and that our task is simply to find it.

```
Let m = Ceil(n/2)
If A[m] < A[m - 1] then search for a peak in the left subarray
Else if A[m] < A[m+1] then search for a peak in the right subarray
Else return A[m]
```

Let us try to argue that this approach is correct. We assume that the array contains at least three elements and that a middle element exists. Then we are always going to (1) find a peak in the left subarray, (2) right subarray, or (3) conclude that the middle element is a peak. No other cases can occur since a peak always exists.

To further convince us that we have a correct solution it is in this case a good idea to make the pseudo code more precise. Here, we provide a sketch of a recursive solution, where we also handle the case when n is not an even number.

```
Function Peak(A, n):
    Let m = Ceil(n/2)
    If n =< 2 then ... // How to implement the base case?
    Let A1 = [A[1], ..., A[m-1]]
    Let A2 = [A[m+1], ..., A[n]]
    If A[m] < A[m-1] then return Peak(A1, m-1)
    Else if A[m] < A[m+1] then return Peak(A2, n-m)
    Else return A[m]
```

This code, while more precise than the previous fragment, still leaves a little bit to the imagination. For example, we do not describe exactly how the two subarrays A1 and A2 should be computed. In Python this could very simply be implemented by slicing the list via the built in colon operator.

Logarithmic Versus Linear Time

How much faster is then the divide and conquer based algorithm compared to the simplistic solution? In each application we split the current input in half and then solve either the left subtask, the right subtask, or conclude that the middle element is a peak. This implies that the number of arithmetical operations is going to be dominated by a *logarithmic* function $\log_2(n)$. This is a major improvement compared to a linear number of operations. For example, if the input array contains a million elements, then the divide and conquer algorithm is going to require at most 20 (!) iterations in the worst possible case, while the simple solution might have to go through the array element by element.

Computability

In the previous section we argued that many computational problems are best solved by high level pseudo code which outlines the main algorithmic ideas, which can then be converted into an actual solution in a specific programming language when the need arises. Thus, we ignore certain details since we know that they can be implemented in a number of straightforward ways. However, there is something unsatisfactory about this mindset. Could it be the case that it can be implemented on one type of computer but not on another? What if we do not agree an the basic assumptions we should make on our computing device? For example, should we allow randomness? Should we have a fixed size on the amount of memory that a program can use or can it (in principle) be infinite? What about quantum bits? Do we require that our computational model is completely *deterministic* or could there sometimes be multiple choices? What, exactly, should our assumptions be on the underlying computational model?

The field of computer science which study different models of computation and their relationship to each other is known as *computability theory*. For example, what can we compute with very limited memory, and what can we compute with an unbounded amount of memory? Consider e.g. the difference between a machine which recognizes a 4-digit numerical PIN code (Figure 1) and a full-blown personal computer (Figure 2). Is there a fundamental difference between these two devices?

Figure 1: A simple computer?


Figure 2: A more universal computer?

While the simple machine in Figure 1 in principle could be implemented by a small, universal computer, it is not a great leap of faith to imagine that there might exist a simpler model of representation for the PIN code machine since it at any stage only needs to read a single digit as input and proceed accordingly, without needing to care about any past attempts. For example, suppose that the correct code is "1234". Then the machine may immediately reject the current attempt if it reads a symbol outside the set $\{1, 2, 3, 4\}$, and reset the memory state. Similarly, if the machine is in its initial state, reads the symbol "1", then it may immediately reject if the next symbol is not "2".

In contrast, a universal computer certainly seems more complicated since it e.g. has access to additional memory which may be used to aid the computation. Another striking difference is that the PIN code machine — at least in principle — always terminates with a definite answer, but that it is certainly possible to write a computer program which does not terminate. We now briefly describe three

models of computation before we zoom in on the most powerful one.

A Crash Course in Formal Languages

In the theory of computation it is common to view a computational device as a machine which takes a *string* as input and returns either YES or NO depending on whether the machine accepts the string or not. The string could, for example, be a binary string consisting only of zeroes and ones, but it is sometimes convenient to allow any predefined "alphabet" to obtain a slightly more general theory. For the peak finder problem considered in the previous section, an input string could consist of a binary string describing the number of array elements, n, followed by a binary encoding of the array elements. To simplify the description we could also separate the array elements by some suitable symbol so that we know exactly where each element begins and ends. Thus, we are working in a simplified setting where machines only return YES or NO answers, but if we want to obtain e.g. a numerical answer (such as in the peak finder problem) we can simply inspect the string once the machine has halted, or the state of the internal memory of the machine.

Definition 2. *A* language *is a set of strings over a predefined alphabet.*

Our main interest in the rest of this lecture note is when a language encodes a computational problem, and the goal is then to construct a machine which answers YES if the string is included in the language and NO otherwise.

Finite Automata

A *finite automata* represents the simplest possible model of computation which is still powerful enough to result in interesting applications. This device consists of a finite number of "states" representing different stages of the computation, and for each symbol in the input string proceeds to a new state through a set of transition rules. If there are no symbols left to read and the automaton reaches a so-called *accept state* then the machine answers YES, and otherwise NO. Crucially, a finite automaton has very little memory of the past, may not modify the input string in any form, and always terminates. See Figure 3 for a visualisation of a finite automaton. The PIN code machine from Figure 1 is an example of a finite automaton.

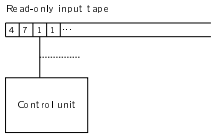

Figure 3: A visualisation of a finite automaton.

Pushdown Automata

The pushdown automaton is a generalisation of finite automata where the machine is equipped with additional memory in the form

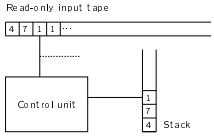

Figure 4: A visualisation of a pushdown automata.

of a stack, where symbols may be pushed and pulled. Thus, a pushdown automaton can store an unbounded amount of items, but only within the confinements of a stack (e.g., we only have access to the topmost element). For example, assume that we in the context of a parser want to be able to recognise whether a given string of parentheses is *balanced*, i.e., each left parenthesis (has a matching right parenthesis) later in the string. This language is known to *not* be recognizable by any finite automaton but can easily be recognised by a pushdown automaton by pushing and pulling in an appropriate way depending on whether the current input symbol is (or). See Figure 4 for a visualisation of a pushdown automaton.

Turing Machines

The most powerful model of computation that we consider is the *Turing machine*, named after the British mathematician Alan Turing. The difference between a Turing machine and the two previously mentioned classes of automata is that a Turing machine may not only inspect the input string, but also (1) modify it and (2) make it longer so that the Turing machine in effect also has unbounded memory (see Figure 5). This seemingly minor modification results in a significantly increased expressive strength, and a Turing machine is believed to be able to compute *everything* that can be computed, a conjecture known as the *Church-Turing thesis*. Modern computers, including smartphones and similar devices, all operate in a fashion similar to Turing machines. The main advantage of studying Turing machines instead of concrete computers is that they are much simpler to describe and reason with, which in turn makes it easier to prove mathematical properties.

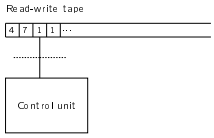

Figure 5: A visualisation of a Turing machine.

Formal Definitions

A Turing machine takes an input string as argument, to which it can both read, write, and extend in any direction if it needs additional memory. In addition the Turing machine satisfies the following specification.

1. The input string is visualized as a *tape* consisting of cells.
2. Cells on the tape either contains a symbol from the alphabet or a special *blank* symbol, which we denote by B.
3. The Turing machine has a "head" which can read, write, move left, and move right, on the tape, but does this in a sequential manner.

Those of you not familiar with magnetic tapes can simply view a magnetic tape as a doubly-linked list where each node, corresponding to a cell, knows the cell to the left and to the right, but nothing else. Hence, just as in a doubly linked list, we can move sequentially both to the left and to the right, but do not have "random access" to the memory.

4. The Turing machine cannot move any further left once it has reached the leftmost position of the tape.

5. However, the tape is effectively *unbounded* in size, meaning that the Turing machine cannot reach the (right) end of it. Conceptually, the tape at a single moment in time is always finite, but if the Turing machine reaches the rightmost cell then we "stretch" the tape by adding additional blank cells.

The formal definition of a Turing machine is then surprisingly undramatic if one is used to mathematical definitions.

Definition 3. [6] *A* Turing machine *is a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ *where:*

- Q *is the finite set of states,*
- Σ *is the finite input alphabet not containing the blank symbol* B*,*
- Γ *is the finite tape alphabet where* $B \in \Gamma$ *and* $\Sigma \subseteq \Gamma$*,*
- $\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *is the transition function, where* L *and* R *stands for "left" and "right", respectively,*
- $q_0 \in Q$ *is the start state,*
- $q_{\text{accept}} \in Q$ *is the accept state, and*
- $q_{\text{reject}} \in Q$ *is the reject state.*

[6] The formal definition is included here only for completeness. Do not worry about the details!

However, since this is an introduction to computability we are happy with merely knowing that there *exists* a concise, formal definition, and will not refer to the above definition again. A Turing machine then operates as follows.

1. The "input string" is written to the "left" of the tape.

2. The Turing machine moves its head to the first cell of the tape and begins in its initial state.

3. The Turing machine reads a single symbol at the current position of its head, writes a symbol at the cell, transitions to the state specified by the transition function, and moves its head left or right.

4. The machine accepts (immediately) if it reaches the accept state, and rejects (immediately) if it reaches the reject state, and in both cases the machine is said to *halt*.

5. If the machine does not reach an accept state or the reject state then it continues forever and is said to *loop*.

The operational semantics of a Turing machine can be formally described by *configurations*, consisting of the current state, the current position of the head, and the current content of the tape. A Turing machine then *accepts* a string if the start configuration leads to a configuration containing the accept state, and *rejects* the string if it leads to a configuration containing the reject state.

Definition 4. *A language (or computational problem) is* decidable *if there exists a Turing machine which accepts all strings in the language and rejects all strings that are not in the language.*

We sometimes also say that a language is *computable* by a Turing machine. Let us conclude this section with an example.

Example 2. *Let us construct a Turing machine which decides the language* $\{0^n1^n2^n \mid n \geq 0\}$*. Consider a Turing machine which operates according to the following description.*

1. *Scan the input from left to right and make sure it is of the form* $0^*1^*2^*$*, i.e., any number of zeroes followed by any number of ones, followed by any number of twos (if it is not, then reject).*
2. *Return the head to the left end of the tape.*
3. *If there is no* 0 *on the tape, then scan right and check that there are no* 1*'s and* 2*'s on the tape and accept (should a* 1 *or* 2 *be on the tape, then reject).*
4. *Otherwise, cross of the first* 0 *and continue to the right crossing of the first* 1 *and the first* 2 *that is found (should there be no* 1 *or no* 2 *on the tape, then reject).*
5. *Go to Step* 2*.*

When describing a Turing machine our main interest is typically *not* to describe the precise number of states or how the transition function should be defined. All that matters is that the high-level description is unambiguous and precise enough so that we in principle could describe the states and the resulting transition function.

The Church-Turing Thesis

The Turing machine seems powerful, but how strong is it, really?

Example 3. *Assume that we want to describe a Turing machine which, given a binary number on its tape, adds* 1 *to this number and accepts. The machine could then shift the input one step to the right and proceed to the last symbol of the string. If this symbol is* 0 *then it simply replaces it by* 1*. If*

this number is 1 *then it replaces it by* 0*, remembers that it has* 1 *in "carry", goes to the left, and repeats this as long as necessary.*

We do not bother with describing the exact number of states or the transition function, since it is clear that we can accomplish the procedure with a finite number of states. Even better: if we can add 1 *to a binary number, then we could certainly also compute the addition of two arbitrary binary numbers simply by repeatedly adding* 1 *to the sum. However, if we can do addition, then we can also do multiplication as repeated addition. Similarly, we could describe a Turing machine which given two binary numbers writes* 1 *if the first number is strictly smaller than the second number, and* 0 *otherwise. But if we can implement addition, multiplication, all normal arithmetical operations and relations, then we could certainly implement e.g. "for loops", and so on. We can easily continue in this fashion and describe more sophisticated constructs from programming languages and describe their implementation in term of Turing machines.*

Once we have the idea of implementing more and more complicated data structures and concepts by shuffling strings around on the tape then it becomes more and more reasonable that a Turing machine can compute anything in this manner. Naturally, if we tried to build a Turing machine following Definition 3 then it would turn out to be awfully slow in practice, but since the Turing machine is a theoretical model of computation we do not care about this deficiency. The conjecture that Turing machines can compute everything that can be computed is known as the *Church-Turing thesis*.

The "Church" part of the conjecture is named after the American mathematician Alonzo Church who discovered an equivalent notion of computability called the *lambda calculus*.

Conjecture 1. *(The Church-Turing thesis) Everything that is "computable" can be computed by a Turing machine.*

There exist several variants of the Church-Turing thesis but the above claim is good enough for our purposes. Note that the Church-Turing thesis is not a proper mathematical conjecture since we have not provided a proper definition of "computable". We will not delve deeper into the problematic nature of giving a general definition of "computation" and take a very pragmatic view: every reasonable model of computation that has been discovered thus far can be simulated by a Turing machine. In particular, no matter how we try to generalise a Turing machine (e.g., by adding more memory, or more features), the resulting machine is still not more powerful than a Turing machine. Now, observe that if we have e.g. a programming language which is expressive enough to "simulate" a given Turing machine on any given input string, then the language can (in principle) be used to solve any computable problem by simulating a suitable Turing machine.

Definition 5. *(Informal) A programming language is said to be* Turing complete *if it can simulate any Turing machine.*

This property is naturally very desirable since it implies that the programming language can be used to solve any task which we believe is computable.

On the Existence of Undecidable Problems

If Turing machines are so powerful, could it even be the case that they can compute any type of computational problem? Or does there exist problems which are not decidable by any Turing machine, and, thus, via the Church-Turing thesis, cannot be computed at all? Consider the following example which closely mirrors the actual proof of the existence of undecidable problems.

Example 4. *Assume that you have just finished writing a nice meta-interpreter for your favourite programming language. This interpreter takes a program, represented as a string, and an input string, as arguments, and returns the resulting of interpreting the given program with the given input string. Naturally, after this feat of engineering you immediately start looking for applications for your shiny new toy. Would it be possible to answer some meta questions about programs? For example, could we determine whether a program terminates or not? This would be a rather nice application since it would make debugging much easier. Hence, you decide to extend the meta-interpreter so that it returns* 1 *if a given program halts with respect to the given input string, and* 0 *if it loops. The easy part is of course if the program terminates: you then simply run the interpreter with the given input, and once it terminates you return* 1*. To handle the case when the program does not seem to terminate you implement some "clever" loop-detection scheme and are completely satisfied with the test programs that you tried. To make it user-friendly you call the function* halt *and you let it take a single argument consisting of the program in question (which contains some type of main function containing some test data).*

However, how can you be sure that halt *actually works? Could it be possible to come up with a counter example? Consider the following. We define a new function (in the same programming language)* halt$'$ *which takes a program P as argument and*

1. *calls* halt(P)*,*
2. *if the result is* 0 *then it returns* 1*, and if the result is* 1 *then it loops.*

Hence, the new program does exactly the opposite of halt*. But now, what happens if we call* halt$'$ *with* itself *as input? That is, a string encoding of the source code of* halt$'$*. We have the following possibilities.*

1. *Case* 1*:* halt *returned* 0*, meaning that* halt$'$ *loops. But according to the definition of* halt$'$ *it should return* 1*, not loop.*

2. *Case* 2*:* halt *returned* 1*, i.e., that* halt′ *does not loop. But then* halt′ *will loop, which is exactly the opposite of what* halt *reported.*

Since neither outcome is possible we conclude that the claimed properties of halt *cannot be possible, and that it cannot correctly deduce whether every given program halts or not.*

Deciding whether a program (or a Turing machine) halts on a given input is typically called the HALTING problem. The formal proof of the undecidability of the HALTING problem closely follows the intuition outlined above and can be formally proved as a mathematical theorem.

Theorem 1. *The* HALTING *problem is undecidable.*

Many problems are known to be undecidable. Curiously, essentially all non-trivial meta properties of Turing machines are undecidable. Importantly, if the Church-Turing thesis is true, then *no* other reasonable model of computation can resolve these problems either. This suggests that certain problems are inherently uncomputable.

References

J.G. Brookshear and D. Brylow. *Computer Science: An Overview*. Pearson Education, Limited, 2018. ISBN 9781292263427.