10. COMPUTATIONAL RESOURCES

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

November 2024

Up until now we have studied the field of Computability Theory, answering questions about what problems are, what algorithms are, which problems can be solved algorithmically, which problems cannot be solved algorithmically, etc.

We now have a good understanding of which problems are decidable (or computable) and can filter out those that are not, using reductions.

In this lecture, we move on into the field of Complexity Theory. We will focus solely on decidable problems – those which admit an algorithmic solution – and we'll focus on the nature of possible solutions.

The goal of this part of the course is to move on from the classification of "solvable" vs "unsolvable" and start reasoning about *difficulty*; we will set the basis for a new classification system, around the lines of "easy" and "hard".

Our focus will remain mainly on problems! We want to study not only whether some *particular solution* is "efficient". But rather, whether some particular problem *admits an "efficient" solution*; if so, just how "efficient"?

It may come as a surprise that problems can be arbitrarily difficult. Another surprising fact is how many questions in this field are currently *open*; i.e. unsolved. We'll introduce a practically-relevant class of problems whose actual "difficulty" is currently unknown; they're believed to be "hard", but there's no proof that they aren't "easy"!

In the paragraphs above, many words appear within quotes, to emphasize that they're merely intuitive qualities, not precise terms. We now move on to develop a formal framework for studying and discussing complexity. Our main model of computation will be, as until now, the Turing machine.

1 Computational resources

When a Turing machine solves a problem, it needs both time and space to do so.

We will not judge "time" in seconds, but in *number of transitions* performed from start to finish.

For space, we will consider the *number of nonblank cells*.[†].

In our analysis, we will focus on *time*, proceeding from the idea that *time is more important than space*. This isn't just an empty nicety that you might see embroidered on a pillow; we can provide a few arguments to support it:

- **space is dependent on time**: if a TM needs to take more space (i.e. extend its tape contents by writing onto a blank cell), then it can only do so by writing during a transition. Thus each new occupied cell entails a new transition.

- **space is reusable, time is not**: once a TM has written some symbols onto its tape, it can come back later and rewrite them with something else; giving the already occupied portion of space some new meaning. But once a transition is performed, it's done: we can't reuse time.

We do not mean that space isn't important; it's just that we choose to prioritize our discussion of time complexity. We could provide you with hundreds more pages of lecture notes on space complexity, but you wouldn't have time to read them.

[†]This is actually problematic, because we always have to include the cells that contain symbols of the input; but it might not seem fair to consider that the TM "uses" that space. For example, we would like to say that a machine that treats its tape in a read-only fashion, uses no additional space. A better definition is needed and we'll provide it when we start discussing space complexity; we will use a multitape machine that has a read-only input tape.

2 Resource consumption as a function

Now that we know what computational resources are, we can develop a framework which allows us to *compare* and *classify* different Turing Machines.

Note that we don't want to discuss *pairs of the form* (M, w), where M is a TM and w a word. We want to abstract away particular inputs and talk about a machine's "general" performance.

The first idea is to consider a function $t_M : \Sigma^* \to \mathbb{N}$, fixed for a particular machine M, which maps each input w to the number of transitions performed by $M[w]$.

However this is "too general"; it allows for all sorts of time-complexity functions. But an important empirical observation is that the kinds or problems we're interested in have solutions which **take more time to run on longer inputs.**[‡]

So we can focus solely on the input length, abstracting away its contents.

This immediately seems problematic, as machines can make different number of transitions based on *input contents*. Consider a machine which checks if a binary string contains at least one "0", and its run on all 16 inputs of length 4:

- for the 8 inputs that start with 0, the machine makes one transition
- for the 4 inputs that start with 10, the machine makes two transitions
- for the 2 inputs that start with 110, the machine makes three transitions
- for 1110, the machine makes four transitions
- for 1111, the machine makes five transitions (the machine doesn't know the length of the string, so it needs to end up on the blank symbol after the input in order to conclude the input did not contain any 0s)

This machine will always make just one transition for inputs that start with a single 0, regardless of their length. But the longer the input, the longer the maximum possible prefix of 1s that a machine has to go trough, searching for a 0. In the *worst-case*, the number of transitions is one more than the length of the input (when the input is just a string of 1s).

In general, for any input length n, there are $|\Sigma|^n$ possible inputs and thus $|\Sigma|^n$ or less possible values of our transition-counting function.

But one (or more) of these values is *maximal*.

2.1 "Worst-case complexity"

We choose to focus our analysis on the "worst-case input" for a given input-length n. There are other sensible approaches (most notably *"average-case complexity"*) that offer valuable insights, but treating the worst-case is both simple and very general. It provides us with strong guarantees about the "slowness" of a machine: it can't go any slower than this.

Definition 10.1. A Turing Machine M's *running time* is a function $T_M : \mathbb{N} \to \mathbb{N}$, where:

$$T_M(n) \stackrel{\text{def}}{=} \max_{w \in \Sigma^n} (t_M(w))$$

Remember that t_M is the function that counts the number of transitions on a particular input.

Σ^n is the set of all inputs of length n over alphabet Σ.

What this definition tells us is that T_M counts, for each input length n, the maximum number of transitions taken by M. We say that: *"M runs in $T_M(n)$ time"*.

We will employ this same definition for multitape Turing Machines as well. Note that for a machine with k tapes, there are k *current symbols* and k *tape-heads* that are affected **simultaneously** by a single transition. In other words, we don't count "k steps" for a transition, just a single one.

[‡]There are also other considerations that should be taken into account for a time-complexity function. This is out of scope for our course and we will not delve into this subject, but they keyword to look up is "time constructible functions".

Our goal is to characterize and classify *problems*. At this point, it might seem deceptively easy: just take the *fastest machine* that solves the problem, and consider its running time to be the problem's difficulty.

However, as we shall see next, there is no such machine! We can always build a faster one.

3 Speeding up!

In this section we will employ a two-tape Turing Machine to illustrate why our characterization of efficiency is still too particular. We will use this observation to motivate a new level of abstraction, where we hide away constants and *"lower-order terms"*, such that we can say that $3n + 3$ is "the same" as $5n + 12$.

3.1 Two-tape palindrome check

Consider the following two-tape Turing Machine[§] that checks if a binary string is a palindrome:

1. the machine copies the input string onto the second tape, by moving both tape-heads in tandem to the right, until the first head reaches a blank
2. the second head is moved back onto the beginning of the input
3. the machine moves both heads in opposite directions (first head to the left, second head to the right), while their symbols match
4. if both heads reach a blank at the same time, the string is a palindrome and the machine accepts; if there's ever a mismatch, the string isn't a palindrome and the machine rejects[††].

What is the running time of the machine?

First, we must identify, for a given input length n, what are the worst-case inputs. The machine behaves differently on palindromes and on non-palindromes; on non-palindromes it ends its execution as soon as it finds a mismatch. But if the input is a palindrome, the machine has to go through all symbols in the comparison stage to make sure of this. Thus palindrome strings are the worst case inputs.

On a palindrome of length n, the machine makes:

- n transitions to copy the input onto the second tape
- $n+2$ transitions to move the first head back onto the first input symbol (the last of this transitions also moves the second head one cell back onto the last input symbol)
- $n + 1$ transitions to go through the whole input and transition to Y at the end

In total, $3n + 3$ transitions.

It may come as a surprise, but we can actually improve this running time and build a machine that does fewer than $3n + 3$ transitions. We will do so by increasing the number of states and the number of tape symbols.

3.2 Linear speedup

Our new machine[‡‡] will have four new tape symbols that encode pairs of binary digits (note that these can't be part of the input):

- A: 00
- B: 01
- C: 10

[§]The table representation for this machine is too large to be included here; but you can find a complete description of this machine (written in the syntax for the online simulator) in the `examples` directory.

[††]Note that the tape contents are necessarily of the same length; the only mismatch that can happen is that one of them reads a 0, while the other a 1, not blank.

[‡‡]Full description in the `examples` folder.

- D: 11

For now, we will only consider **palindromes of even length**. This keeps our analysis simple and easy to follow. At the end of this section, we will show that our conclusion also holds when palindromes of odd-length are considered; but we have to make an extra effort for our machine to work.

1. The machine will move its first head to the right, but will use three variants of the state $copy$ to memorize pairs of symbols and fill in the second tape with a string of letters that encodes the input in half the length.

   As the binary digits are copied from the first tape, they are also **erased**.

   After this copying, the first tape will be empty and the second tape will contain a half-length encoding of the original input

2. Both tape-heads are then moved in tandem to the left, copying each symbol from the second tape onto the first one.

3. The first head is reset onto the first blank symbol after the input.

4. Now the machine enters the "comparison stage", where it moves both heads in opposite directions (first to the left, second to the right), checking if the symbols underneath match (note that a B (01) matches a C (10)!).

   If it finds a mismatch, it rejects; otherwise, it accepts.

This machine is certainly more *complicated*. It uses more symbols, it has more states and its behavior is more complex. However, it **is** faster!

On an input of length n, the machine performs:

- n transitions to erase the input from the first tape and write its encoding onto the second tape
- $\frac{n}{2} + 1$ transitions to copy the encoded contents of the second tape back onto the first one
- $\frac{n}{2} + 2$ transitions to move the second head back onto the first symbol (the last of this transitions also moves the first head one cell forwards onto the first symbol)
- $\frac{n}{2} + 1$ transitions to go through the whole input and transition to Y at the end

In total, $\frac{5n}{2} + 4$ transitions.

For the empty string, this machine is slower than the first one. But for palindromes of length 2, the machines make the same number of transitions. For longer strings, it does less!

For a string with a hundred symbols, the first machine makes 303 transitions, but this machine only makes 254!

The key element of our efficiency-gain technique was to first rewrite the input in a more *space-efficient* way, such that any subsequent processing on the input will take fewer transitions.

There's nothing stopping us at halving the input! We can introduce eight new symbols in the tape alphabet, to encode three bits at the time. Or sixteen symbols to encode four bits at the time.

In general, we can employ an additional 2^c tape symbols to encode c input bits; we might need a few extra transitions to make this work, but the result will generally be faster. Whatever the original running time T_M was, we would obtain a new one, of the form: $T_{M'}(n) = aT_M(n) + b$ for some constants $a < 1$ and b. This is why this technique is called *"linear speedup"*.

However, we need $n + 1$ extra transitions in order to delete the initial input and write this compressed version of it. Thus we can only use this technique to speedup machines with a running time greater than $n + 2$.

Note 10.1. What about palindromes of odd length?

During the initial compression stage, we would be left with a "remainder" symbol; a final 0 or 1 that we would simply copy as-is on the second tape. So `100111001` would become `CBDA1`.

We would then copy it onto the first tape; the issue is that, during the next step we would have to compare the first and last symbols.

The key to solve this issue is to notice that the final symbol is merely a bit and match it with either C (10) or D (11), then **memorize** the second bit of the letter using states:

- if the first symbol was a C, transition to `compare_0`
- if the first symbol was a D, transition to `compare_1`

We would the move on to compare B with A in state `compare_0`; that means we should take the bit from the state (0) and the first bit of B (0) and compare it to A; the second bit of B tells us that we should go to state `compare_1`.

This logic is simply hardcoded in the transitions of the machine. You can find a full version of it in the `examples` folder.

While cumbersome, this technique does generalize to any c number of bits we want to compress into a single new symbol.

3.3 Linear speedup for single-tape machines

We used a two-tape machine to illustrate the concept of linear speedup because it is easier to follow and we could do it starting from a known problem.

The presence of an extra tape means that we can very easily go through the input and copy it in its compressed form on another tape as we go. On a single-tape machine, we can only write the compressed form on the same tape, at the end of the original input.

After reading two bits, we would have to cross all the input to reach the compressed form, cross that as well and add a new letter (A, B, C or D) at the end; then go back to the beginning of the input.

We can roughly approximate that for each symbol of the input, we have to cross the entire input, so we need around n^2 transitions just to rewrite our input to the compressed form. Does this effort to halve the input length pay off?

For any single-tape machine that does fewer than n^2 transitions, no. But if the machine did n^3 transitions, the sped-up version would only do around $n^2 + (\frac{n}{2})^3$ transitions.

The "classic" single-tape solution for `PALINDROME` (that we saw during the second lecture) does perform around n^2 transitions, but does so with very little overhead; so linear speedup does not help.

4 Asymptotic notations

In the previous section we showed that using linear speedup we can always improve the running time of a particular machine and get one that makes fewer transitions.

This is problematic for us, because we set out to characterize *problems* by efficiency.

We can't say that the decision problem `PALINDROME` has complexity $3n + 3$. It also has $5\frac{n}{2} + 4$ complexity and so on.

However, linear speedup has its limitations. It's right there in the name: we can only speedup a machine in a *linear* fashion, in an expression like $aT_M(n) + b$.

The key element we need to characterize problems is to be able to abstract away these linear elements; to ignore a and b when discussing complexities. For this, we turn to *asymptotic notations*, with which we will be able to express both $3n + 3$ and $5\frac{n}{2} + 4$ with the same notation: $O(n)$.

The asymptotic notations are a family of five notations that define sets of functions that share a certain *growth* characteristic.

4.1 Θ - exact bounds

The first asymptotic notation that we will study is perhaps the most intuitive one, which gives a precise rate of growth.

Definition 10.2.

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ s.t. } \forall n \geq n_0,\ c_1 f(n) \leq g(n) \leq c_2 f(n)\}$$

This definition tells us that, for a given function f, the set $\Theta(f)$ consists of functions g from naturals to naturals that *grow at the same rate as* f. But what does that mean?

Imagine that we're going through the natural numbers; starting from 0 and making our way **up** through 1, 2, 3, ... As we go up the natural number scale, for each number n we look at the value of $g(n)$ and its linear relationship with the value of $f(n)$.

We are interested in comparing the values "as we go to infinity". There might be a section in the beginning where the relation between these functions vary, but we're not interested in it: no matter how large, it is finite, thus negligible. This is the role of n_0 and of the qualifier $\forall n, n \geq n_0$: it says that we're only interested in the relationship between $f(n)$ and $g(n)$ from some point onwards.

Can we find two positive constants c_1 and c_2, such that we can "pin" $g(n)$ between $c_1 f(n)$ and $c_2 f(n)$? (Note that it's important for our constants to be positive; any negative number would be a satisfactory choice for c_1, which would make it useless in our definition). If yes, then we say: $g \in \Theta(f)$ and we can read this as "g grows at the same rate as f".

Remember how we defined two multitape TMs for solving `PALINDROME`, one with running time $3n+3$ and the other $5\frac{n}{2} + 4$.

We can find $n_0 = 2, c_1 = c_2 = 1$, such that $\forall n \geq n_0, c_1(3n+3) \leq 5\frac{n}{2} + 4 \leq c_2(3n+3)$.

So $5\frac{n}{2} + 4 \in \Theta(3n+3)$.

The reverse is also true, $3n + 3 \in \Theta(5\frac{n}{2} + 4)$.

Moreover, choosing $n_0 = 1, c_1 = \frac{1}{4}, c_2 = 1$ we can see that $3n+3 \in \Theta(n)$. Similarly, we can show that $5\frac{n}{2}+4 \in \Theta(n)$. We shall always use an asymptotic notation in its simplest term (here n) as a canonical representation of that set.

Both machines' performance can be characterized using $\Theta(n)$. Can we extend this to the problem they solve and say: "`PALINDROME` has a complexity of $\Theta(n)$"?

Not quite! We haven't proven that there isn't some *asymptotically faster* machine that solves `PALINDROME`. Sure, it can be solved by a machine with running time $\Theta(n)$, but what if it could be solved by one with running time $\Theta(\log n)$[§§]?

In order to use an exact bound to label the complexity of the problem, we should show not only that it has a solution with that time complexity, but also that no faster solution exists. This is quite difficult to achieve and we won't attempt it much at this course.

We can reduce our burden by settling instead on an *upper bound*.

4.2 O - upper bounds

Θ may be the most intuitive of the asymptotic notations, but it's likely not the best-known. That honor goes to O (read as "big-oh").

Definition 10.3.

$$O(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ s.t. } \forall n \geq n_0,\ 0 \leq g(n) \leq cf(n)\}$$

If $g \in O(f)$, we can read this as "g grows as fast as f, or slower".

$3n + 3 \in O(n)$, but also $3n + 3 \in O(n^2)$, even though $3n + 3 \notin \Theta(n^2)$. The function $3n + 3$ grows *slower* than n^2, not at the same rate.

We can now safely say that the time complexity of `PALINDROME` is $O(n)$. In the field of Complexity Theory, we model this by saying that a problem is part of a particular *"complexity class"* related to a resource, parameterized by an O notation of a Turing Machine that solves the problem. So `PALINDROME` $\in \mathbf{DTIME}(n)$ (the "D" stands for "deterministic"; for now, we will ignore what this means, but we will soon discuss *"nondeterminism"* and *nondeterministic complexity classes*).

[§§] $\log n \notin \Theta(n)$ – you will prove this at the labs.

4.3 Ω - lower bounds

Definition 10.4.

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ s.t. } \forall n \geq n_0,\ 0 \leq cf(n) \leq g(n)\}$$

4.4 o - strict upper bounds

Definition 10.5.

$$o(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \forall c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ s.t. } \forall n \geq n_0,\ 0 \leq g(n) \leq cf(n)\}$$

Note the subtle difference with O: g can be bounded from above by f **for all factors** c, no matter how small they are.

It might be the case that, as c gets lower, the further we have to go on the natural numbers, i.e. the bigger n_0 has to get.

4.5 ω - strict lower bounds

Definition 10.6.

$$\omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} \mid \forall c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \text{ s.t. } \forall n \geq n_0,\ 0 \leq cf(n) \leq g(n)\}$$

4.6 Asymptotic notations in practice

Starting from the definitions presented, we can rigorously prove many statements, short but relevant for the practice of labeling functions with a notation.

We can show that $\forall c, cf(n) \in \Theta(n)$, or that $log(n) \in o(\sqrt{n})$, or that $\forall k, n^k \in o(2^k)$.

Because of limited time at the lecture, we will consider many of these "intuitive" and take them for granted. The proofs are left as exercises for the lab.

4.7 Syntactic sugars

We will often use expressions in which we need to refer to "some" function or to "all" functions in a particular asymptotic set.

For example:

$$\forall f \in \Theta(n), \exists g \in O(n), \exists h \in o(\log n), \text{ s.t. } f(n) = g(n) + h(n)$$

We will employ an abuse of notation, common in the field and simply use the *existentially quantified* sets on the right-hand-side of the equal sign. We can also do this for *universally quantified* ones on the left-hand-side:

$$\Theta(n) = O(n) + o(\log n)$$

5 References and further reading

The linear speedup presented here is adapted from Christos Papadimitriou's textbook *"Computational Complexity"* [1, p. 32]. The book is an excellent introduction in complexity theory. Chapter 2, 3, 7, 8 and 9 touch on many of the subjects we have already discussed in the first part of the course and those that we will discuss in this second part on complexity.

The asymptotic notations have their origins in number theory and are sometimes named *"Bachmann–Landau notations"*, after mathematicians Paul Gustav Heinrich Bachmann (1837-1920) and Edmund Landau (1877-1938). Bachmannn first introduced the O notation in his 1894 book *"Analytische Zahlentheorie"* [2] ("analytical number theory"). The symbol was used as shorthand for the word "Ordnung" ("order").

The notation was adopted by Landau and extended to o in his 1909 book *"Handbuch der Lehre von der Verteilung der Primzahlen"* [3] ("Handbook of the theory of the distribution of prime numbers").

In the context of algorithm analysis, they were popularized (and redefined) by Donald Knuth. His 1976 article *"Big omicron and big omega and big theta"* [4] provides the modern definitions (as well as justifications for their usefulness) that serve as the basis for our discussion.

Bibliography

[1] Christos Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[2] Paul Bachmann. *Analytische Zahlentheorie*. BG Teubner, 1894, p. 401.

[3] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primazahlen*. Vol. 1. BG Teubner, 1909.

[4] Donald E Knuth. "Big omicron and big omega and big theta". In: *ACM Sigact News* 8.2 (1976), pp. 18–24.