Remarks on Turing Machines, Languages, and Complexity

Dimitris Diochnos

Norman, November 21, 2019

1 Different Levels of Description of a Turing Machine

When describing TMs we can have three different levels of details that we provide for their execution.

Formal Description. We define the 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ that completely describes the TM. So that the transition function δ can be defined it is perhaps best if you draw a diagram and this way also help people who are reading the description of the TM to understand in an easier way what is happening.

Implementation Description. This is a higher level description for a TM, compared to the formal description. This time we are using English prose to describe the way the TM moves its head and how data is stored on its tape. We do not give details of the transition function.

High-Level Description. We give an algorithm ignoring the implementation details. We do not have to explain how the machine manages its tape or head.

1.1 Example of a High-Level Description

Let $B = \left\{ w\#w \mid w \in \{0,1\}^* \right\}$. Give a high-level description of a TM that decides the language B.

Solution

Let $k \geqslant 1$ be the position of the input where the unique # symbol is found. For each $i \in \{1, 2, \ldots, k-1\}$, we will be matching the symbols in positions i and $k+i$. If we can do that for every $i \in \{1, 2, \ldots, k-1\}$ and no more 0 or 1 symbols remain, then we *accept*.

In any other case, we *reject*.

1.2 Example of an Implementation Description

Let $B = \left\{ w\#w \mid w \in \{0,1\}^* \right\}$. Give an implementation description of a TM that decides the language B. (This is an example from Sipser's book [6].)

Solution

M_1 = "On input string w:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, *reject*; otherwise, *accept*."

1.3 Example of a Formal Description

Let $B = \{ w\#w \mid w \in \{0,1\}^* \}$. Give a formal description of a TM that decides the language B. (This is Example 3.9 from Sipser's book [6].)

Solution

- The set of states is $Q = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_a, q_r\}$.
- The input alphabet is $\Sigma = \{0, 1, \#\}$ and the tape alphabet is $\Gamma = \{0, 1, \#, \square\}$.
- The starting state is q_1. The accept state is q_a. The reject state is q_r.
- For the transition function δ we have the diagram shown in Figure 1.

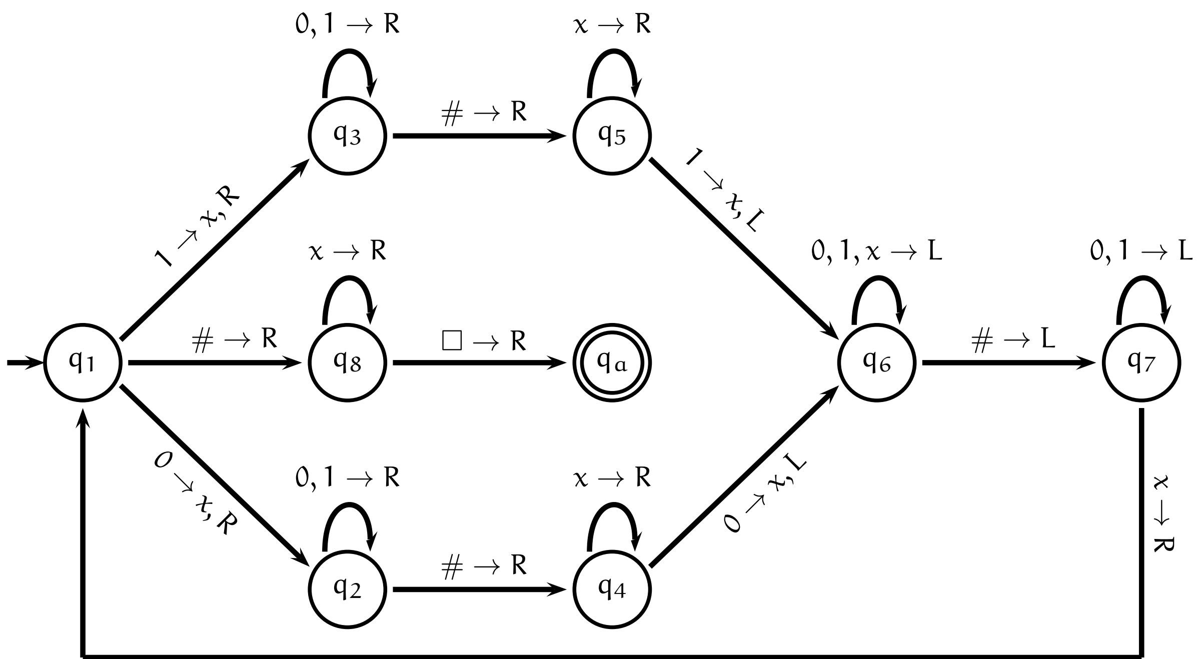

Figure 1: State diagram for a TM deciding language B.

First of all, note that we have grouped some symbols together. For example, in state q_6 we have the rule $0, 1, x \to L$. This notation means that if we read a 0 we write a 0 and we move L, or if we read a 1 we write a 1 and we move L and if we read an x we write an x and we move L. In other words, it has combined the following three rules: $0 \to L$, $1 \to L$ and $x \to L$. Similarly, for state q_2 for the rule $0, 1 \to R$ we have combined the rules $0 \to R$ and $1 \to R$.

Also, note that the diagram in Figure 1 does not explicitly mention the reject state q_r. This is a typical simplification where all the missing arrows from the diagram will be leading to the reject state. Such transitions occur implicitly whenever a state lacks an outgoing transition for a particular symbol. For example, if we are in state q_4 and we read a 1, or a #, or a $\square$, then the TM will go to the state q_r (which is not shown) and *reject*. (It does not matter how the head moves in such transitions to the reject state; say that the head is moving to the right.)

Based on the *implementation* description that we have from Section 1.2, we see that the branch with the states q_1, q_2, q_4, q_6 and q_7 corresponds to reading a 0 to the left of #, crossing off the matching 0 to the right of # and then rewinding the tape head all the way to the left until we find the last x which is to the left of #.

Similarly, the branch with the states q_1, q_3, q_5, q_6 and q_7 corresponds to matching the appropriate 1's this time, crossing them off and rewinding the head of the tape at the appropriate position.

Finally, states q_8 and q_a correspond to stage 2 from the implementation description that was given in Section 1.2.

2 A Hierarchy of Languages

With the tools that we have explored so far we have identified several inclusions among the different classes of languages that exist. In particular we know the following:

$$\text{Finite Languages} \subsetneq \text{Regular Languages} \subsetneq \text{Context-Free Languages} \subsetneq \text{Turing-Decidable Languages}\,.$$

In other words, if a certain language is finite; e.g., $L_1 = \{0, 01, 001, 011\}$, then that language is also regular, it is also context-free, it is also Turing-decidable.

Similarly, if we are given a language L and the information that it is, say, context-free, then we can deduce that L is also decidable. However, we can not deduce that L is regular unless we know something more and we can prove this claim.

In fact the set inclusions above are all proper. That is, for every class of languages that we have identified, there is at least one language that we can define that strictly belongs to that class and not to a subset class of languages; examples follow.

- The language $L_4 = \{\, 0^n1^n2^n \mid n \geqslant 0 \,\}$ is decidable but not context-free.
  (We can construct a TM that decides L_4 and it is an easy exercise to use the pumping lemma for context-free languages to show that the language is not context-free.)

- The language $L_3 = \{\, 0^n1^n \mid n \geqslant 0 \,\}$ is context-free but it is not regular.
  (We can give a grammar or construct a PDA and show that this language is context-free and then it is an easy exercise to use the pumping lemma for regular languages in order to show that the language is not regular.)

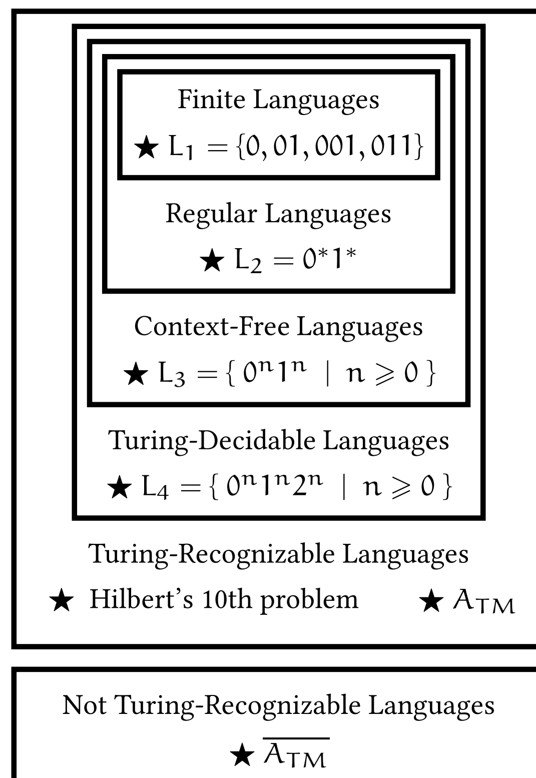

Figure 2: A view of different languages

- The language $L_2 = 0^*1^*$ is regular but clearly not finite, as is for example $L_1 = \{0, 01, 001, 011\}$ which is composed of only four different strings.
  (We can construct a DFA and show that the language is regular. The fact that the language does not have a finite number of strings is obvious by the Kleene star operators.)

All the above are shown schematically in Figure 2. Figure 2 has additional information that we have not yet discussed but we will do so below.

2.1 Decidable vs Recognizable Languages

Recall the definitions that we have for *decidable* and *recognizable* languages.

Definition 1 (Decidable Languages). *A language L is called **Turing-decidable** (or just **decidable**), if there exists a deterministic TM M, such that $L(M) = L$ and moreover M satisfies the following: on an arbitrary input string w, if $w \in L$ then M accepts, otherwise ($w \notin L$) M rejects.*

Definition 2 (Recognizable Languages). *A language L is called **Turing-recognizable** (or just **recognizable**), if there exists a deterministic TM M, such that $L(M) = L$ and moreover M satisfies the following: on an arbitrary input string w, if $w \in L$ then M accepts.*

In other words, *every decidable language is recognizable, but not the other way around.* For a recognizable but not decidable language L, if we are given a TM M that recognizes L and an input $w \notin L$, then feeding w into M may result in M never halting and reaching the reject state.

- Is this discussion meaningful? Are there languages that are recognizable but not decidable?

The answer is *yes!* We will examine this below.

2.2 Integral Roots of Univariate Polynomials

Consider the following language.

$$L_{poly,1} = \{ \langle p \rangle \mid p \text{ is a univariate polynomial in } x \text{ with integer coefficients that has an integral root} \} ,$$

where $\langle p \rangle$ is some suitable representation of the polynomial p; see Section 5 for a discussion.

2.2.1 $L_{poly,1}$ is Recognizable

There is an easy algorithm for recognizing if a given polynomial $p(x)$ has an integral root.

Algorithm 1: An algorithm that identifies an integral root of a univariate polynomial $p(x)$.

Input: a polynomial $p(x) = a_k x^k + \ldots + a_1 x + a_0$.
Output: Returns TRUE if $p(x)$ has an integral root.

```
1 for x ∈ {0, +1, −1, +2, −2, +3, −3, ...} do
2     v ← p(x);
3     if v = 0 then
4         return TRUE;
```

Note that Algorithm 1 will indeed recognize a polynomial $p(x)$ that has an integral root, because sooner or later it will find that root.

However, if the polynomial does not have an integral root, the algorithm will progressively attempt integer values for x that have larger and larger absolute value and will never halt.

In the case of univariate polynomials we can however improve our algorithm and make it stop and therefore decide if a given univariate polynomial $p(x)$ has an integral root or not.

2.2.2 $L_{poly,1}$ is Decidable

Regarding roots of univariate polynomials the following bound is due to Cauchy.

Proposition 1 (Cauchy [5]). *Let* $\rho \in \mathbb{C}$ *be a root of* $\mathbb{R}[x] \ni h(x) = \alpha_k x^k + \cdots + \alpha_1 x + \alpha_0$*. Then,*

$$|\rho| \leqslant 1 + \frac{\max\{|\alpha_0|, |\alpha_1|, \ldots, |\alpha_{k-1}|\}}{|\alpha_k|} .$$

Algorithm 2 decides the language $L_{poly,1}$.

Algorithm 2: An algorithm that decides if a given univariate polynomial has an integral root.

Input: a polynomial $p(x) = a_k x^k + \ldots + a_1 x + a_0$.
Output: Returns TRUE if $p(x)$ has an integral root, otherwise returns FALSE.

```
1 b ← 1 + ⌊ max{|α_0|,|α_1|,...,|α_{k-1}|} / |α_k| ⌋;
2 for x ∈ {−b, −b+1, ..., −1, 0, +1, ..., b−1, b} do
3     v ← p(x);
4     if v = 0 then
5         return TRUE;
6 return FALSE;
```

2.3 Hilbert's Tenth Problem

Consider the following equation in the three variables x, y, and z. Does it have an integral solution?

$$x^3 + y^3 + z^3 = 42 \tag{1}$$

It turns out that the above equation has at least one integral solution. We know the following one:

$$\begin{cases} x &= -80538738812075974 \\ y &= +80435758145817515 \\ z &= +12602123297335631 \end{cases}$$

As you can see, the three integer numbers[1] that are satisfying (1), have grown very-very large compared to the coefficients involved in (1).

Hilbert's tenth problem asked to devise an algorithm that *decides* if an arbitrary multivariate polynomial with integer coefficients has an integral root – not just for the sum of the three cubes that we see in (1).

While Hilbert posed the problem formally in 1900, it was only in 1970 that Yuri Matijasevič showed that no algorithm exists for testing whether a polynomial has integral roots [3, 4]. Matijasevič's work built on results by Martin Davis, Hilary Putnam, and Julia Robinson. Davis in [2] gives a complete account of the results that were put together in order to prove that Hilbert's tenth problem is unsolvable.

[1] In fact, the above solution was found on September 6, 2019 by Andrew Sutherland of MIT and Andrew Booker of Bristol University; http://news.mit.edu/2019/answer-life-universe-and-everything-sum-three-cubes-mathematics-0910.

3 The Church-Turing Thesis: Definition of Algorithm

The Church-Turing thesis connects what we intuitively believe is an algorithm to what is formally defined as an algorithm. Alonzo Church made this connection with what is known as λ-calculus[2]. Alan Turing made this connection with Turing machines. Both published their work in the same year (1936), the two definitions are equivalent (from the point of view of computation), and thus tribute is given to both.

In particular, we have the situation shown in Figure 3.

Intuitive notion of algorithms	equals	Formal description of a Turing machine

Figure 3: The Church-Turing thesis.

4 Complexity Theory

Complexity theory studies *decidable languages* and attempts to further refine their classification in different *complexity classes*.

This way, we have problems that belong to the complexity class P, problems that belong to the complexity class NP, problems that belong to the complexity class L, problems that belong to the complexity class NL, problems that belong to complexity class PSPACE, problems that belong to the complexity class $\mathsf{EXPTIME}$, problems that belong to the complexity class $\mathsf{EXPSPACE}$, and many-many others.[3]

In particular, regarding P, which contains all the problems (languages) that are decidable by a deterministic TM in polynomial time, we are used to discussing about further refinements. For example, a sorting algorithm A_1 that is running in time $\mathcal{O}(n\log(n))$ is, in principle, preferable to a sorting algorithm A_2 that is running in time $\mathcal{O}(n^2)$. In fact, there is a lower bound of $\Omega(n\log(n))$ for sorting algorithms that are based on comparisons - we saw this in class in the beginning of the semester. Thus A_1 is *asymptotically optimal* and in fact the running time for A_1 in this example would be $\Theta(n\log(n))$.

5 Encoding

We want to be able to encode different objects and pass them as input to TMs. In this direction, for an object O we will denote its encoding as $\langle O\rangle$.

For example, we may want to encode the graph G shown in Figure 4. Then, a reasonable encoding is the following one:

$$\langle G\rangle = (\{1,2,3,4\},\{(1,2),(2,3),(2,4),(3,4)\})\,.$$

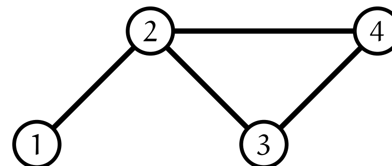

Figure 4: A graph.

In other words, G is described as a pair (a 2-tuple), where the first element of that pair is the set of vertices of G and the second element of the pair is the set of edges of G.

As another example, consider the multivariate polynomial

$$p(x,y,z) = x^3 + y^3 + z^3 - 42$$

[2] See *functional programming* for the evolution of λ-calculus.

[3] *Complexity Zoo* (https://complexityzoo.uwaterloo.ca/Complexity_Zoo) has 544 different classes of languages at the time of this writing.

which corresponds to (1). One reasonable encoding for p would be:

$$\langle p \rangle = \{(1, (3, 0, 0)), (1, (0, 3, 0)), (1, (0, 0, 3)), (-42, (0, 0, 0))\} \ .$$

In other words we represent p as a set, where each element of that set is a monomial. We denote monomials as pairs where the first element denotes the coefficient of the monomial and the second element is a triple indicating the degrees in which the different variables x, y, and z appear in the monomial.

5.1 Feeding Input into Turing Machines

Assume we want to design a TM for recognizing the language

$$L_{poly,3} = \{ \langle g \rangle \mid g \text{ is a polynomial in 3 variables with integer coefficients, that has an integral root} \} \ .$$

We can do so with the following high-level description of a TM M_3.

$M_3 =$ "On input $\langle p \rangle$, the encoding of a polynomial p:

1. Set $b \leftarrow 0$.
2. Increase b by 1.
3. Consider iteratively all the triples (x, y, z) such that the absolute value of x, y, and z is at most b. (There are $(2b + 1)^3$ such triples.) For each triple generated test if it is a root of p.
   - If some triple is a root, *accept*.
   - Otherwise, go to step 2."

In the above algorithm, where M_3 receives the input $\langle p \rangle$, M_3 first tries to determine if the input is some proper encoding of a polynomial in three variables. If this is not the case, then M_3 will reject. Such a test can be made easily. First of all, the input has to be a set of monomials. So, M_3 checks that indeed we have a set and each member of a set is a pair. Once this is established, then with another pass of the head from left to right we make sure that the first coordinate is an integer and the second coordinate of each monomial is a triple of non-negative integers. (Finally, a last pass may require that all the monomials are distinct with each other, so that our job is easier and we do not have to merge different monomials into one.) If the input passes all these tests, then it is the encoding of some polynomial in three variables.

Then we can move on with stage 1 and the rest of the execution steps dictated by M_3.

5.2 Encoding Turing Machines

Once we have a description of a TM, as for example M_3 above, then we can encode this description into $\langle M_3 \rangle$. Of course the best way to think about such an encoding $\langle M \rangle$ of a TM M, is that we are really encoding the 7-tuple that corresponds to the formal description of the Turing machine M. However, as we have discussed, we will hardly ever give such detailed information on paper and we will usually give either an implementation description, or a high-level description of the Turing machine – realizing however that, if we want to, we can indeed use such a description and obtain the most detailed description, the formal description.

Having such an encoding $\langle M \rangle$ of a Turing machine M is interesting, because we will be able to pass it as part of the input to another Turing machine M' which will be able to simulate M in a manner similar to how modern computers execute different computer programs.

Exercises

Exercise 1. *Let* $\Sigma = \{0, 1\}$. *Give a DFA* M *such that* $L(M) = L_1 = \{0, 01, 001, 011\}$.

Exercise 2. *Let* $\Sigma = \{0, 1\}$. *Do the following.*

(a) Give a DFA M *such that* $L(M) = L_2 = 0^*1^*$.

(b) Give a context-free grammar G *that generates* L_2.

(c) Give a high-level description of a Turing machine that decides L_2.

Exercise 3. *Let* $\Sigma = \{0, 1\}$. *Do the following.*

(a) Give a context-free grammar for the language $L_3 = \{\, 0^n 1^n \mid n \geqslant 0 \,\}$.

(b) Use the pumping lemma for regular languages and show that L_3 *is not a regular language.*

Exercise 4. *Let* $\Sigma = \{0, 1\}$. *Do the following.*

(a) Use the pumping lemma for context-free languages and show that $L_4 = \{\, 0^n 1^n 2^n \mid n \geqslant 0 \,\}$ *is not a context-free language.*

(b) Give an implmementation description of a Turing machine that decides L_4.

Exercise 5. *Let* $h(x) = \alpha_k x^k + \cdots + \alpha_1 x + \alpha_0$ *be a polynomial that has a real root at* $x = x_0$. *Show that,*

$$|x_0| < (k+1) \cdot \frac{\max\{|\alpha_0|, |\alpha_1|, \ldots, |\alpha_{k-1}|, |\alpha_k|\}}{|\alpha_k|}.$$

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[2] Martin Davis. Hilbert's tenth problem is unsolvable. *The American Mathematical Monthly*, 80(3):233–269, 1973.

[3] Yuri Matijasevič. Enumerable sets are diophantine (russian). In *Dokl. Akad. Nauk SSSR*, volume 191, pages 279–282. Improved English translation: Soviet Math. Doklady, 11 (1970) 354-357, 1970.

[4] Yuri Matijasevič. Diophantine representation of recursively enumerable predicates. In *Studies in Logic and the Foundations of Mathematics*, volume 63, pages 171–177. Elsevier, 1971.

[5] Maurice Mignotte. *Mathematics for computer algebra.* Springer-Verlag, New York, 1991.

[6] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, Boston, MA, 3rd edition, 2013.

A A Quick Tour on Asymptotic Complexity

A.1 Preliminaries

Definitions drawn from [1, Section 3.1, pp. 41-50].

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \text{ such that } 0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)\ \forall n \geqslant n_0\}\,. \tag{2}$$

$$\mathcal{O}(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ such that } 0 \leqslant f(n) \leqslant c g(n)\ \forall n \geqslant n_0\}\,. \tag{3}$$

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \text{ such that } 0 \leqslant c g(n) \leqslant f(n)\ \forall n \geqslant n_0\}\,. \tag{4}$$

A.2 Dropping Lower Order Terms of Polynomials

$\mathcal{O}$ Notation

Assume that $f(n) = \mathcal{O}(g(n))$, where $g(n) \in \mathbb{R}[n]$ with $g(n) = \sum_{i=0}^{k} \alpha_i n^i$, $1 \leqslant k, n \in \mathbb{N}$, $\alpha_i \in \mathbb{R}$ with $i \in \{0, 1, \ldots, k-1\}$, and $\alpha_k = 1$. By (3) there exist $c, n_0 > 0$ such that

$$0 \leqslant f(n) \leqslant c \cdot \sum_{i=0}^{k} \alpha_i n^i \quad \forall n \geqslant n_0\,. \tag{5}$$

We now want to show that $f(n) \leqslant c' \cdot n^k$, for some $c' > 0$ and $\forall n \geqslant n_1$. It is enough to satisfy

$$c' \cdot n^k \geqslant c \cdot \sum_{i=0}^{k} \alpha_i n^i \quad \forall n \geqslant n_1 \geqslant n_0\,. \tag{6}$$

However, $n^k > 0$, so dividing both sides of (6) we want to find a c' such that

$$c' \geqslant c \cdot \sum_{i=0}^{k} \alpha_i n^{i-k} = c \cdot \left[1 + \frac{\alpha_{k-1}}{n} + \frac{\alpha_{k-2}}{n^2} + \cdots + \frac{\alpha_0}{n^k}\right]\,.$$

On the other hand,

$$\begin{aligned} c \cdot \sum_{i=0}^{k} \alpha_i n^{i-k} &= \left| c \cdot \sum_{i=0}^{k} \alpha_i n^{i-k} \right| \\ &\leqslant c \cdot \sum_{i=0}^{k} |\alpha_i| n^{i-k} \\ &\leqslant c \cdot \sum_{i=0}^{k} |\alpha_i| \end{aligned}$$

It therefore follows that selecting

$$c' = c \cdot \sum_{i=0}^{k} |\alpha_i|$$

and setting $n_1 = n_0$ it holds that $f(n) \leqslant c' \cdot n^k$ for all $n \geqslant n_0$, and hence we can neglect all the lower order terms that appeared in the original polynomial $g(n)$. In other words, we can simply write down that $f(n) = \mathcal{O}(n^k)$.

Ω Notation

Let $f(n) = \Omega(g(n))$, where $g(n) \in \mathbb{R}[n]$ with $g(n) = \sum_{i=0}^{k} \alpha_i n^i$, $1 \leqslant k, n \in \mathbb{N}$, $\alpha_i \in \mathbb{R}$ with $i \in \{0, 1, \ldots, k-1\}$, and $\alpha_k = c > 0$. By (4) there exists $n_0 > 0$ such that

$$0 \leqslant \sum_{i=0}^{k} \alpha_i n^i \leqslant f(n) \quad \forall n \geqslant n_0 \,. \tag{7}$$

We now want to show that $f(n) \geqslant c' \cdot n^k$, for some $c' > 0$ and $\forall n \geqslant n_1$. First, let $n^\star$ be as the bound in Proposition 1, i.e.

$$n^\star = 1 + \frac{\max\{|\alpha_0|, |\alpha_1|, \ldots, |\alpha_{k-1}|\}}{c} \,.$$

Then, for the largest real root $\rho^\star$ of $g(n)$ it holds $\rho^\star \leqslant n^\star$, after which point $g(n)$ is strictly positive. We can therefore rewrite (7) as

$$0 < \sum_{i=0}^{k} \alpha_i n^i \leqslant f(n) \quad \forall n \geqslant \max\{n_0, n^\star + 1\} \,. \tag{8}$$

Now consider the polynomial $h(n) = \frac{c}{2} n^k$, where $c = \alpha_k$ in $g(n)$ above. Therefore the difference $d(n) = g(n) - h(n) = \frac{c}{2} n^k + \sum_{i=0}^{k-1} \alpha_i n^i$ is a polynomial of degree k, with coefficient for the largest power equal to half of the respective one in $g(n)$ and the rest of the coefficients agree with $g(n)$. We now apply Proposition 1 to $d(n)$ and it follows that the largest root $\rho^{\star\star}$ is at most

$$1 + 2 \cdot \frac{\max\{|\alpha_0|, |\alpha_1|, \ldots, |\alpha_{k-1}|\}}{c} \,.$$

Setting $n^{\star\star} = 2 + 2 \cdot \frac{\max\{|\alpha_0|, |\alpha_1|, \ldots, |\alpha_{k-1}|\}}{c} = 2n^\star$, it follows that $d(n) > 0$ for all $n \geqslant 2n^\star$. Moreover, for every $n \geqslant 2n^\star$ both $g(n)$ and $h(n)$ are strictly positive, and since $d(n)$ is also positive we have

$$0 < h(n) < g(n) \quad \forall n \geqslant 2n^\star \,.$$

As a consequence, we can rewrite (7) as

$$0 < \frac{c}{2} \cdot n^k < \sum_{i=0}^{k} \alpha_i n^i \leqslant f(n) \quad \forall n \geqslant \max\{n_0, 2n^\star\} \,. \tag{9}$$

It therefore follows that we can neglect all the lower order terms that appeared in the original polynomial $g(n)$ and hence simply write down $f(n) = \Omega\left(n^k\right)$.

A.3 The Substitution Method

The substitution method for solving recurrences consists of two steps:

1. Guess the form of the solution.

2. Use mathematical induction to find constants in the form and show that the solution works. The base case of induction is called the *boundary condition*.

The method is used both for lower bounds as well as upper bounds.

A.3.1 A Detailed Example

Consider the recurrence relation

$$\begin{cases} T(n) & = & 2 \cdot T(\lfloor n/2 \rfloor) + n \\ T(1) & = & 1 \end{cases} . \tag{10}$$

We want to show that $T(n) = \Theta(n \lg n)$.

Upper Bound. Our induction hypothesis is $T(n)$ is $\mathcal{O}(n \lg n)$ or $T(n) \leqslant cn \lg n$ for some constant c, independent of n. Assume the hypothesis holds for all $m < n$ and substitute:

$$\begin{array}{rcll} T(n) & \leqslant & 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n & \text{(Induction Hypothesis)} \\ & \leqslant & cn \lg(n/2) + n & ((\forall n \in \mathbb{N})\ (\lfloor n/2 \rfloor \leqslant n/2)) \\ & = & cn \lg n - cn \lg 2 + n & \\ & = & cn \lg n - (c-1)n & \\ & \leqslant & cn \lg n \ , & \end{array}$$

where the last inequality was obtained by requiring $-(c-1)n \leqslant 0 \Leftrightarrow c \geqslant 1$.

For the base case we would like to show

$$T(1) \leqslant c \cdot 1 \cdot \lg 1 = c \cdot 0 = 0 \ ,$$

which is impossible when $T(1) > 0$. However, we only want to show that $T(n) \leqslant cn \lg n$ for *sufficiently large* values of n; i.e. $\forall n \geqslant n_0$. Hence, we will try $n_0 > 1$. First, by (10) we have $T(2) = 4$ and $T(3) = 5$. Hence, we want to satisfy simultaneously[4]

$$\begin{cases} 4 = T(2) & \leqslant & c \cdot 2 \cdot \lg 2 \\ 5 = T(3) & \leqslant & c \cdot 3 \cdot \lg 3 \end{cases} \Longrightarrow \begin{cases} c & \geqslant & 2 \\ c & \geqslant & \frac{5}{3 \lg 3} \approx 1.052 \end{cases} \Longrightarrow c \geqslant 2 \ .$$

As a conclusion, we have proved that $T(n) \leqslant 2n \lg n$ for all $n \geqslant 2$.

Lower Bound. For the lower bound we will break the proof into two parts. First, we will give a lower bound for all the values of n which are powers of 2, and we will also show that the function $T(n)$ is a strictly increasing sequence. In the end we will give a function $g(n)$ for the lower bound such that $g(n)$ will take different values when n is a power of 2, and remain constant to the latest value until the next power of 2.

Powers of 2. We want to show $T(n) \geqslant cn \lg n$. Assume that n is a power of 2. We have:

$$\begin{array}{rcll} T(n) & \geqslant & 2(c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n & \text{(Induction Hypothesis)} \\ & = & cn \lg(n/2) + n & (n \text{ is a power of 2}) \\ & = & cn \lg n - cn \lg 2 + n & \\ & = & cn \lg n - (c-1)n & \\ & \geqslant & cn \lg n \ , & \end{array}$$

where the last inequality was obtained by requiring $-(c-1)n \geqslant 0 \Leftrightarrow c \leqslant 1$.

We also want to satisfy the boundary condition ($T(2) = 4$).

$$T(2) \geqslant c \cdot 2 \cdot \lg 2 = 2 \cdot c$$

In other words, it is enough if $c \leqslant 2$. By the requirement $c \leqslant 1$ for the induction step we choose $c = 1$.

[4] We have to check *both* $T(2)$ and $T(3)$ *simultaneously* because of the floor function in the recursive equation.

$T(n)$ Is Strictly Increasing. We will prove this claim again by induction. For the base case note that $T(1) = 1 < 4 = T(2)$. Assuming that for all $k \leqslant n$ it holds $T(k) > T(k-1)$, we want to show that $T(n+1) > T(n)$. We distinguish cases for $n+1$.

$(n+1)$ is odd: Say $n+1 = 2m+1$. Then, it holds

$$\begin{aligned} T(2m+1) &= 2T\left(\lfloor (2m+1)/2 \rfloor\right) + 2m + 1 && \text{(Definition)} \\ &= 2T(m) + 2m + 1 \\ &= T(2m) + 1 && \text{(Definition)} \\ &> T(2m)\,. \end{aligned}$$

$(n+1)$ is even: Say $n+1 = 2m$. Then, it holds

$$\begin{aligned} T(2m) &= 2T\left(\lfloor (2m)/2 \rfloor\right) + 2m && \text{(Definition)} \\ &= 2T(m) + 2m \\ &> 2T(m-1) + 2m && \text{(Induction Hypothesis)} \\ &= 2T\left(\lfloor (2m-1)/2 \rfloor\right) + (2m-1) + 1 \\ &= T(2m-1) + 1 && \text{(Definition)} \\ &> T(2m-1)\,. \end{aligned}$$

Note that the induction hypothesis is used only when $(n+1)$ is even!

Putting Everything Together. Consider the binary expansion of $n > 0$; i.e. $n = \sum_{i=0}^{\infty} b_i 2^i$. Based on the previous two paragraphs we define the function

$$g(n) = \begin{cases} n \cdot \lg n & , \quad n \text{ is a power of } 2, \\ k \cdot 2^k & , \quad \text{otherwise and } k \text{ is the maximum } i \text{ for which it holds } n = \sum_{i=0}^{\infty} b_i 2^i \end{cases} . \qquad (11)$$

In other words, $g(n)$ has "jumps" on the values that it takes when n is a power of 2, and remains constant until the next power of 2. From the previous analysis it now follows that $T(n) = \Omega\left(g(n)\right)$.

A Figure. Figure 5 presents graphically the sequence $T(n)$, as well as the functions $2 \cdot n \cdot \lg n$ and $g(n)$ which we defined for the lower bound for $n \in \{1, 2, \ldots, 65\}$. The figure also shows a looser lower bound derived from $g(n)$. Can you guess which function is this one ?

A.3.2 Wrong Approaches

Consider the recurrence

$$\begin{cases} T(n) &= T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1 \\ T(1) &= 1 \end{cases} . \qquad (12)$$

Our guess is $\mathcal{O}(n)$, so we try to show $T(n) \leqslant cn$.

$$\begin{aligned} T(n) &\leqslant c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

which does *not* imply that $T(n) \leqslant cn$, for any c.

Remark 1. *Note that this is different compared to dropping lower order terms from a polynomial as in Section A.2. The reason is that here we try to determine the order of $T(n)$ and hence we have to stick on the exact statement that we want to prove. Once this step is over and we have actually determined the order of $T(n)$, we can simplify our answer based on Section A.2 by dropping lower order terms.*

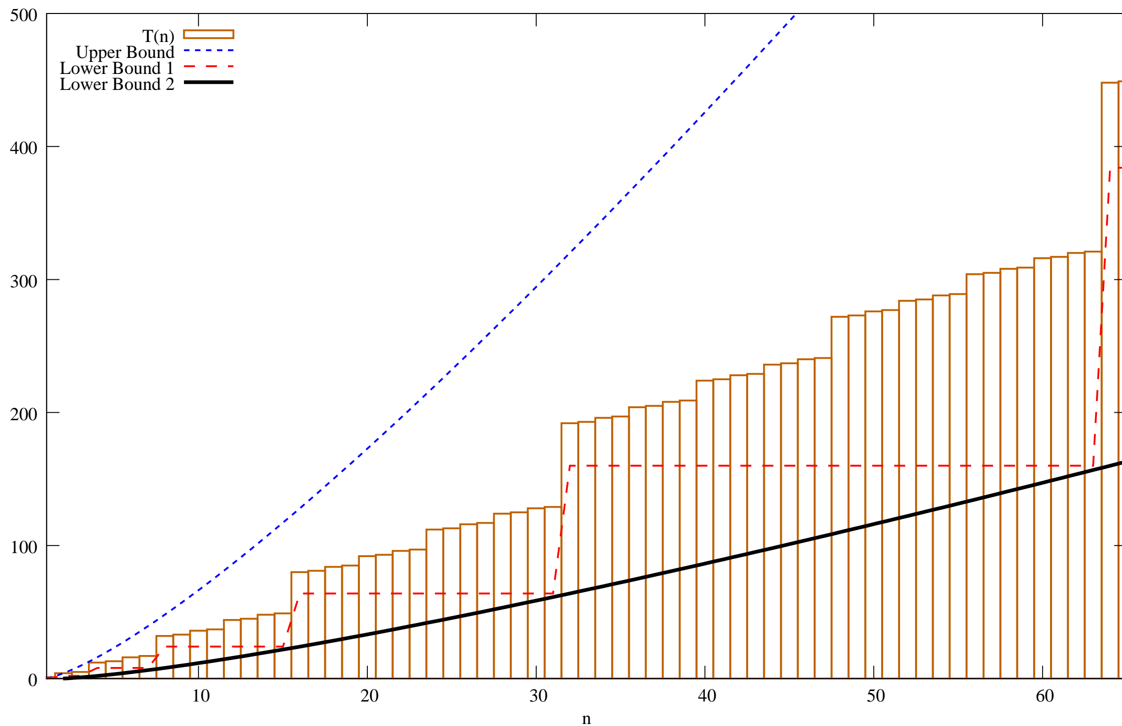

Figure 5: With boxes we can see the actual sequence $T(n)$, while with dashed lines we can see the lower and upper bound functions we have derived. The solid line gives a looser lower bound. Can you guess which function is this one ?

As another example, consider the recurrence given in (10). The following argument is wrong. We guess that the recurrence has $T(n) \leqslant cn$ and try to "prove" that it is indeed $\mathcal{O}(n)$.

$$\begin{aligned} T(n) &\leqslant 2(c\lfloor n/2 \rfloor) + n \\ &\leqslant cn + n \\ &= \mathcal{O}(n), \end{aligned}$$

since c is a constant.

Remark 2. *The error is at the last step, where again we have not proved the exact form that we wanted to; i.e.* $T(n) \leqslant cn$.