Chapter 14

Asymptotic Complexity and The Class $\mathcal{P}$ of Problems

In the previous chapter, we have defined the time complexity of a Turing Machine as a function of the size n of input instances, where the time complexity for the size n was the maximal number of steps that the Turing Machine takes to decide any input instance of the size n. We have also seen an example of the Turing Machine M that decides the language $\{0^n1^n|n \geq 0\}$, and derived that the time complexity of M is

$$T_M(n) = \frac{n^2 + 2n}{2}.$$

We have also seen that coming up with the precise formula for time complexity of even as simple Turing Machine as M was quite fiddly and error-prone. This becomes far more complicated when we have more complex Turing Machines. We, therefore, need some simplification of this - some concept of time complexity that is not so hard to calculate and so prone to small errors in calculations. For this purpose, we will use an approximate, *asymptotic* notation for the complexity of Turing Machines.

14.1 O Notation

The asymptotic notation that we introduce in this notation has been accepted as the standard notation for asymptotic complexity of Turing Machines and computer programs in general. It gives us an estimation of the precise time complexity, taking into account only the most important factors of the time complexity and throwing away everything else. What we are interested in is just the *rate of growth* of the time complexity, i.e. how fast it increases as the instance size increases.

Definition 14.1: O Notation

Let f and g be functions on $\mathbb{N}$, $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if there exist positive integers c and n_0 such that

$$f(n) \leq cg(n),$$

for all $n \geq n_0$.

This might sound complicated, but it is generally quite a simple concept. We say that the function g is the asymptotic upper bound for function g, in that, starting from some n_0, $g(n)$ (multiplied by some constant) is larger than $f(n)$. This also means that the function g is growing at least as fast as f, in that $g(n)$ gets at least as much larger (compared to $g(n-1)$) when n increaes as does $f(n)$ (compared to $f(n-1)$). We use this notation to derive "simple functions" as the estimators of time complexity (e.g. by eliminating constants and terms that do not contribute to the 'order' of growth). Let us see some examples:

- $5n^3+2n^2+22n+6 = O(n^3)$, because $5n^3+2n^2+22n+6 \leq 6n^3$ for $n \geq 10$. However, $5n^3+2n^2+22n+6$ is not, for example, $O(n^2)$, because there does not exist a constant C such that $5n^3+2n^2+22n+6 \leq Cn^2$, for every $n \geq n_0$, for some n_0. Note also that, for obvious reasons, $n^3 = O(5n^3 + 2n^2 + 22n + 6)$, but this is not how we will use the O notation. Our goal is for the function in the O notation to be simpler than the function we are estimating, not more complex.
- $4n^2 + 3n = O(n^2)$.

- $2^n + n^7 + 6n^5 + 33 = O(2^n)$.
- $f(n) = 42$ is $O(1)$.
- $n^{1/3} + n^{1/4}(\log n)^5 + 2 \log n \log \log n = O(n^{1/3})$.

In the function for time complexity of a Turing Machine, there is usually a dominant term (the term whose value most significantly increases as the instance size increases). Using O notation, we can estimate the time complexity of the Turing Machine by this term, eliminating any constants. This gives us the tightest upper bound (the "smallest" function that bounds the time complexity from above). This then allows us to classify Turing Machines based on their time complexity into classes of Turing Machines with similar complexity.

Let us look at a few examples again:

- $5n^3 + 2n^2 + 22n + 6$ and $2n^3 + 7n$ are both $O(n^3)$. Therefore, we can estimate both functions by n^3. Of course, they are both also, for example, $O(n^{100})$, but n^3 is a much tighter bound. In fact, n^3 is the tightest bound of the form n^k. Neither functions are, for example, $O(n^2)$.
- $3n \log_2 n + 5n + 3$ and $7n \log_3 n + 3n + 2n^{1/2} + 14$ are both $O(n \log n)$. Note that we do not need a logarithm base, as $\log_a n = \frac{1}{\log_b a} \log_b n$ for any positive integers a and b. Therefore, two logarithm functions with just the different base differ by a constant factor, and we discard the constant factors anyway in our notation.

To see why O notation is good for estimating the time complexity of Turing Machines, let us assume that time complexity of some Turing Machine M is given by a function $f(n) = 5n^3 + 2n^2 + 22n + 6$ (remembering that $f(n) = 5n^3 + 2n^2 + 22n + 6 = O(n^3)$). The table below shows the values of different terms of the function $f(n)$ for different n's.

n	**6**	**22n**	**2n²**	**5n³**	**n³**	$f(n)$
1	6	22	2	5	1	36
5	6	110	50	625	125	796
10	6	220	200	5000	833	5436
20	6	440	800	40000	8000	41266
50	6	1100	5000	625000	125000	631150
100	6	2200	20000	5000000	1000000	5022300
1000	6	22000	200000	50000000	10000000	50223000

We can see that the largest term ($5n^3$) gets pretty close to the value of function $f(n)$ for large values of n and also that n^3 is of the same order of magnitude as $f(n)$ for larger values of n. This is why estimating $5n^3 + 2n^2 + 22n + 6$ as n^3 gives us a good approximation of the time complexity of M.

Looking back at our example of a Turing Machine M for deciding the language $0^n 1^n$, we have seen that the time complexity of M is $T_M(n) = \frac{n^2+2n}{2}$. We can see that $T_M(n) = O(n^2)$, and that n^2 is the tightest upper bound of the type n^k.

14.2 Classes of Time Complexity

We say that the language L belongs to the class DTIME($t(n)$) if there is a Turing Machine M that decides L, and whose time complexity is $O(t(n))$. For example, we have seen that the language $L = \{0^n 1^n | n \geq 0\}$) belongs to the class DTIME(n^2). Note that L also belongs to DTIME(n^3), DTIME(n^4), ... The fact that L belongs to DTIME(n^2) does not necessarily mean that L does not belong to, for example, DTIME(n). Note also that DTIME($t(n)$) is *a class of languages*, rather than Turing Machines. Just because some Turing Machine decides on a language in time (asymptotically) higher than $t(n)$, it does not mean that the language itself does not belong to DTIME($t(n)$). But if we find one Turing Machine M that decides on a language in time $O(t(n))$ that means that the language belongs to the class DTIME($t(n)$), but also DTIME($s(n)$) for any function s that is asymptotically greater than t.

We can also, informally, talk about the classes of decision problems related to the languages. E.g. the class of the problem "Is a given string w of the form $0^i 1^i$?".

Some common classes of time complexity are:

- DTIME(1) - the languages decided by Turing Machines that take at most some constant number of steps for each instance. We have seen, for example, that the language $L = \{w | w \in \{0,1\}^*, w \text{ starts with } 1\}$ belongs to this class, since we have seen a Turing Machine that decides whether any string belongs to this language in only one step.
- DTIME(n) - the languages decided by Turing Machines that take at most some constant number of passes over the non-blank part of the input tape. For example, the language $L = \{w | w \in \{0,1\}^*, w \text{ has an even number of 0's }\}$ belongs to this class as it can be decided by a Turing Machine that makes a single pass over the given string on a tape.
- DTIME(n^2) - the languages decided by Turing Machines that take the number of passes over the non-blank part of the input tape that is proportional to the input size. For example, $L = \{0^n 1^n | n \geq 0\}$ is decided by the Turing Machine that we have seen, and that makes $n/2$ passes over the number of cells proportional to n.
- DTIME(2^n) - the languages decided by the Turing Machines that use brute force.

To demonstrate why it is important to think of problems/languages in terms of the class of time complexity that they belong to, let us look at the following example. Suppose we have a computer that simulates a Turing Machine and that can do 10^9 (one billion) steps per second (i.e., each step takes 1ns). Then the times taken to carry out algorithms with $\log n$, n, n^2, n^3, n^5, 2^n or 3^n steps are given in the table below.

	Size n						
Time	20	30	40	50	100	1000	1000000
$\log_2 n$	4.3ns	4.9s	5.3ns	5.6ns	6.6ns	9.9ns	20ns
n	20ns	30ns	40ns	50ns	100ns	1µs	1ms
n^2	400ns	900ns	1.6µs	2.5µs	10µs	1ms	16.6m
n^3	8µs	27µs	64µs	125µs	1ms	1s	31.7y
n^5	3.2ms	24.3ms	0.1s	0.31s	10s	11.6d	10^{13}y
2^n	1ms	1s	17.9m	12.7d	10^{22}y	10^{285}y	10^{301013}y
3^n	110y	10^6y					

As we can see, for the polynomial time complexities, the time taken to do a computation rises fairly modestly as the size of the input increases, so that (especially for low degree polynomials such as n^2 or n^3) it is practical to deal with inputs of substantial size (up to the hundreds or thousands, and in some cases more). For exponential time complexities, however, there is a sharp cutoff at an input of size a few tens, beyond which the computation takes an amount of time which renders it, in practical terms, impossible.

What would have happened if the computer used is 100 or 1000 times faster? The following table shows the instance size we would be able to solve in the same time with the computers 100 and 1000 times faster.

$T(n)$	present	100× present	1000× present
n	N	100N	1000N
n^2	N	10N	31.6N
n^3	N	4.64N	10N
n^5	N	2.5N	3.98N
2^n	N	N+7	N+10
3^n	N	N+4	N+6

We can see that the instance size that we can solve increases significantly for the time complexities of n, n^2, n^3 and n^5 (obviously, for some less so than for others), but increases very slightly for the time complexities of 2^n and 3^n. This table, in conjuction with the previous one, also shows that time complexities which are polynomial (i.e., powers of n) behave very differently from those which are exponential (i.e., where n itself is the power, as in 2^n). We can see that the increases in computer speed which have occurred over the last few decades, and will no doubt continue for some time to come, have a big impact on the size of problems which can be solved with a polynomial time algorithm, for exponential algorithms the impact is quite small, and offers no prospect of being able to solve significantly larger problems in the near future.

14.3 Class $\mathcal{P}$

All the preceding discussion naturally leads us to the definition of our first key class of problems - the problems solvable in polynomial time.

Definition 14.2: Class $\mathcal{P}$

Class $\mathcal{P}$ is the class of languages that belong to DTIME(n^i) for some i. In other words,

$$\mathcal{P} = \bigcup_{i=1}^{\infty} \text{DTIME}(n^i).$$

If a language belongs to the class $\mathcal{P}$, then there is a Turing Machine that decides it and that takes at most (asymptotically) n^i steps, for some i. We will call such languages the *easy* languages (or *tractable* languages), because there exist efficient Turing Machines that decide them. Of course, not all polynomial Turing Machines can be realistically classified as efficient - e.g. a Turing Machine with time complexity n^{100} can hardly be classified as efficient. However, for many languages in $\mathcal{P}$, there exist an efficient Turing Machine that decides them (of time complexity n^i for some small i, such as 2 or 3). Also, once we find a Turing Machine of time complexity n^i for some i, such a Turing Machine can usually be improved to reduce the i in its time complexity.

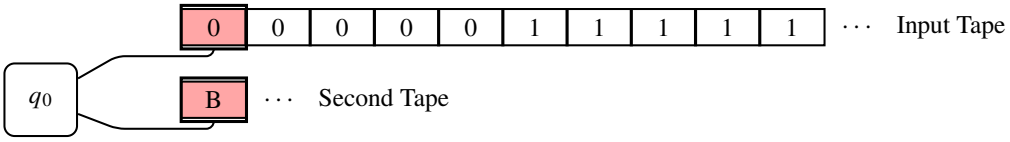
Hard or *intractable* languages would be either those that cannot be decided (undecidable languages) or those which do not have Turing Machines that decide them in polynomial time. So far, the only intractable languages that we have encountered are the undecidable ones (e.g. post correspondence problem). For some languages, we do not know of any efficient Turing Machine that decides them. But that does not mean that an efficient Turing Machine for them does not exist. Also, some inefficient solutions are still usable in practice (e.g. solutions that are efficient for most inputs, or solutions of time complexity $n^{\log \log n}$).

14.3.1 Robustness of Class $\mathcal{P}$

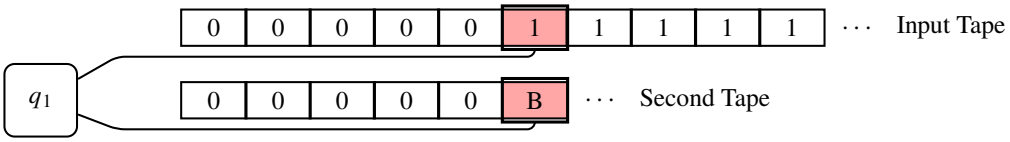
In our study of computability, it did not matter what extension of the Turing Machine model we used (e.g. multi-tape Turing Machines and Turing Machines with infinite tapes on both sides), since they all accepted the same class of languages. In the complexity theory, the situation is a bit different. So far, we have considered single-tape Turing Machines. We can ask if time complexity class of a language/problem changes if we consider multi-tape Turing Machines (or other extensions). If the time complexity of a language/problem changes, how significant the change is? Does classification of the language/problem into $\mathcal{P}$ depends on which Turing Machine model we consider?

To give some insight into this, let us come back to our running example of the language 0^n1^n. We have seen that there exists a Turing Machine that decides on this language in time $O(n^2)$. This Turing Machine can be improved by marking off every other 0 in one pass over the input string, instead of marking off just one. This would result in a Turing Machine with time complexity of $O(n \log n)$. It can be proven that this time complexity cannot be further improved using a single-tape Turing Machine. That it, it is not possible to find a Turing Machine that will decide the language 0^n1^n in asymptotically better time than $O(n \log n)$. Let us, however, consider a two-tape Turing Machine that works in the following way. The Turing Machine scans the leading 0's and copies them to the second tape. When the first 1 is encountered, the sequence of 1's is read and its length is compared with the length of the sequence of 0's on the second tape. A few configurations of this Turing Machine on an input 0000011111 are given in Figure 14.1.

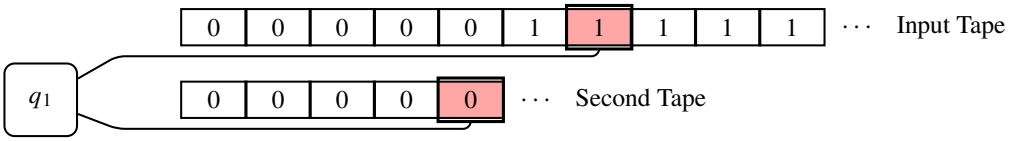
We can easily see that this two-tape Turing Machine requires only one pass over the non-empty part of the input tape, therefore its time complexity will be $O(n)$. Therefore, switching to a multi-tape Turing Machine changed the asymptotic complexity of the solution. It also put the language 0^n1^n belong to the class DTIME(n), whereas considering only single-tape Turing Machines puts the same language into the class DTIME(n^2), or DTIME($n \log n$) at best. Fortunately, while considering two-tape Turing Machine can reduce the complexity of the problem, it cannot reduce it dramatically with respect to the membership to the class $\mathcal{P}$. In other words, if a problem belongs to the class $\mathcal{P}$ when considering multitape Turing Machines, it will also belong to the class $\mathcal{P}$ when only single-tape Turing Machines are considered. The following theorem can easily be proven



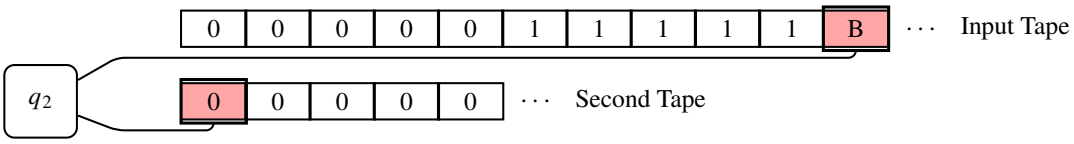
(a) Initial configuration.



(b) Configuration after copying the initial sequence of 0's on the second tape.



(c) Configuration after encountering the first 1.



(d) Configuration after the end of input is reached on the input tape.

Figure 14.1: Configurations of a Two-Tape Turing Machine for the language $L = \{0^n 1^n | n \geq 0\}$.

Theorem 14.1

If M is a multi-tape Turing Machine whose time complexity is $O(t(n))$, then there is a single-tape Turing Machine M' that accepts the same language as M and whose time complexity is $O(t^2(n))$.

Therefore, $\mathcal{P}$ is robust with respect to the number of tapes used in the Turing Machine. The same is also true for other reasonable extensions of the basic model of the Turing Machine, such as two-way infinite tape, multistate Turing Machines etc.