*10.5 TRACTABLE AND INTRACTABLE ALGORITHMS

The merge sort algorithm is much faster than the quadratic-time selection and insertion sort algorithms, but all of these algorithms are considered to be ***tractable***, meaning that they can generally be expected to finish in a "reasonable" amount of time. Any algorithm with time complexity that grows no faster than a *polynomial function* of n (e.g., n, n^2, n^5) is considered to be tractable, while algorithms with time complexities that are *exponential* functions of n are said to be ***intractable***. An intractable algorithm is essentially useless with all but the smallest inputs; even if it is correct and will eventually finish, the answer will come long after anyone who needs it is dead and gone.

Table 1 shows the execution times of five algorithms with different time complexities, on a hypothetical computer capable of executing one billion operations per second. We can see that, for n up to 30, all five algorithms complete in about a second or less. However, when $n = 50$, we start to notice a dramatic difference: while the first four algorithms still finish in a fraction of a second, the exponential-time algorithm requires 13 *days* to complete. When $n = 100$, it needs *41 trillion years*, about 3,000 times the age of the universe. And the differences only get more pronounced for larger values of n; the first four, tractable algorithms finish in a "reasonable" amount of time, even with input sizes of 1 billion[8], while the exponential-time algorithm requires an absurd amount of time. *Notice that the difference between tractable and intractable algorithms holds no matter how fast computers get;* a computer that is one billion times faster than the one used in the table can only bring the exponential-time algorithm down to 10^{275} years from 10^{284} years when $n = 1{,}000$!

The dramatic time differences in Table 1 also illustrate just how important efficient algorithms

Input size n	Logarithmic $\log_2 n$	Linear n	$n \log_2 n$	Quadratic n^2	Exponential 2^n
10	3×10^{-9} sec	10^{-8} sec	3×10^{-8} sec	10^{-7} sec	10^{-6} sec
20	4×10^{-9} sec	2×10^{-8} sec	9×10^{-8} sec	4×10^{-7} sec	0.001 sec
30	5×10^{-9} sec	3×10^{-8} sec	2×10^{-7} sec	9×10^{-7} sec	1.1 sec
50	6×10^{-9} sec	5×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	13 days
100	7×10^{-9} sec	10^{-7} sec	7×10^{-7} sec	10^{-5} sec	41 trillion yrs
1,000	10^{-8} sec	10^{-6} sec	10^{-5} sec	0.001 sec	3.5×10^{284} yrs
10,000	1.3×10^{-8} sec	10^{-5} sec	0.0001 sec	0.1 sec	6.3×10^{2993} yrs
100,000	1.7×10^{-8} sec	10^{-4} sec	0.0016 sec	10 sec	way too long
1 million	2×10^{-8} sec	0.001 sec	0.02 sec	17 min	way, way too long
1 billion	3×10^{-8} sec	1 sec	30 sec	31 yrs	really???

Table 1 A comparison of the approximate times required by algorithms with various time complexities on a computer capable of executing 1 billion steps per second.

[8] 31 years is admittedly a long time to wait, and no one would actually wait that long, but it is far shorter than 41 trillion years.

Tangent 1: Moore's Law

Moore's Law is named after Gordon Moore, a co-founder of the Intel Corporation. In 1965, he predicted that the number of fundamental elements (transistors) on computer chips would double every year, a period he revised to every 2 years in 1975. It was later noted that this would imply that the performance of computers would double every 18 months. This prediction has turned out to be remarkably accurate, although it has started to slow down a bit as we run into the limitations of current technology. We have witnessed similar exponential increases in storage and network capacity, and the density of pixels in computer displays and digital camera sensors.

Consider the original Apple IIe computer, released in 1983. The Apple IIe used a popular microprocessor called the 6502, created by MOS Technology, which had 3,510 transistors with features as small as 8 μm and ran at a clock rate of 1 MHz. Current Apple Macintosh and Windows computers use Intel Core processors. The recent Intel Core i9 (Comet Lake) 8-core processor contains billions of transistors with features as small as 14 nm, running at a maximum clock rate of 5.3 GHz. In less than 40 years, the number of transistors has increased by a factor of at least 500,000, the transistors have become 570 times smaller, and the clock rate has increased by a factor of 5,300.

are. In fact, advances in algorithm efficiency are often considerably more impactful than faster hardware. Consider the impact of improving an algorithm from quadratic to linear time complexity. On an input with n equal to 1 million, we would see execution time improve from 16.7 minutes to 1/1000 of a second, a factor of one million! According to ***Moore's Law*** (see Tangent 1), such an increase in hardware performance will not manifest itself for about 30 more years!

Hard problems

Unfortunately, there are many common, simply stated problems for which the only known algorithms have exponential time complexity. For example, suppose we have n tasks to complete, each with a time estimate, that we wish to delegate to two assistants as evenly as possible. In general, there is no known way to solve this problem that is any better than trying all possible ways to delegate the tasks and then choosing the best solution. This type of algorithm is known as an ***exhaustive search*** or *brute force* algorithm. For the task assignment problem, there are two possible ways to assign each task (to one of the two assistants), so there are

$$\underbrace{2 \cdot 2 \cdot 2 \cdot \cdots \cdot 2}_{n \text{ tasks}} = 2^n$$

possible ways to assign the n tasks. Therefore, the exhaustive search algorithm that tries all 2^n possibilities has $\mathcal{O}(2^n)$, or exponential, time complexity. Referring back to Table 1, we see that even if we only have $n = 50$ tasks, this algorithm would be of no use to us.

It turns out that there are thousands of such problems, known as the *NP-hard problems*, that, on the surface, do not seem as if they should be intractable, but, as far as we know, they are. Even more interesting, no one has actually *proven* that they are intractable. See Tangent 2 if you would like to know more about this fascinating problem.

When we cannot solve a problem exactly, one common approach is to instead use a ***heuristic***. A heuristic is a type of algorithm that does not necessarily give a correct answer, but tends to work well in practice. For example, a heuristic for the task delegation problem might

Tangent 2: Does P = NP?

The ***P=NP problem*** is the most important unresolved question in computer science. Formally, the question involves the tractability of *decision problems*, problems with a "yes" or "no" answer. For example, the decision version of the task assignment problem would ask, "Is there is an assignment of tasks that is within m minutes of being even?" Problems like the original task assignment problem that seek the value of an optimal solution are called *optimization problems*.

The "P" in "P=NP" represents the class of all tractable decision problems (for which algorithms with **P**olynomial time complexity are known). "NP" (short for "**N**ondeterministic **P**olynomial") represents the class of all decision problems for which a candidate solution can be *verified* to be correct in polynomial time. For example, in the task delegation problem, a verifier would check whether a task delegation is within m minutes of being even. (This is a much easier problem!) Because any problem that can be solved in polynomial time must have a polynomial time verifier, we know that a problem in the class P is also contained in the class NP.

The ***NP-complete*** problems are the hardest problems in NP. (NP-hard problems are usually optimization versions of NP-complete problems.) "P=NP" refers to the question of whether all of the problems in NP are also in P or have polynomial time algorithms. Since so many brilliant people have worked on this problem for over 50 years, it is *assumed* that no polynomial-time algorithms exist for NP-hard problems.

assign the tasks in order, always assigning the next task to the assistant with the least to do so far. Although this will not necessarily give the best solution, it may be "good enough" in practice.