# Exponential Time and Polynomial Time

Let me tell you the story of the king and the sage.

The sage invented chess. The king liked the game so much that he offered the sage a reward.

"All I want is wheat", said the sage.

"Then I'll give you as much wheat as you want!", said the king.

The sage smiled and said: "I would like one grain of wheat for the first square of a chess-board, two grains for the second square, four grains for the third, and so forth, for each of the 64 squares."

The king agreed, without realizing that he had committed to delivering more wheat than was contained in his entire kingdom.

In order to fulfill his promise with respect to the $n$th square of the chess-board, the king would have to deliver $2^{n-1}$ grains of wheat. So, in order to fulfill his promise with respect to the last square on the board, the king would need $2^{63}$ $(\approx 9.2 \times 10^{18})$ grains of wheat.

That is an enormous number, much greater than the yearly production of wheat on Earth.

Notice, moreover, that the sage's method will quickly generate even larger numbers.

If the board had contained 266 squares instead of 64, the king would have committed to delivering more grains of wheat than there are atoms in the universe (about $10^{80}$, according to some estimates).

The moral of our story is that functions of the form $2^x$ can grow extremely fast.

This has important consequences related to the efficiency of computer programs. Suppose, for example, that I wish to program a computer to solve the following problem: I pick a finite set of integers, and the computer must determine whether the numbers in some non-empty subset of this set add up to zero. For instance, if I pick the set $\{104, -3, 7, 11, -2, 14, -8\}$, the computer should respond "Yes!" because the elements of $\{-3, 11, -8\}$ add up to zero.

This problem is known as the **subset sum problem.** The simplest way to write a computer program to solve it for any initial set is using a "brute force" technique: the computer generates all the subsets of the original set one by one, and determines for each one the sum of the elements. If at any point, it gets zero as a result, it outputs "Yes!"; otherwise it answers "No!".

Unless the initial set is very small, however, this method is not efficient enough to be used in practice. This is because if the initial set has $n$ elements, the brute force approach could require more than $2^n$ operations to yield a verdict. (An $n$-membered set has $2^n - 1$ non-empty subsets, and our computer may have to check them all before being in a position to yield an answer.)

We therefore face the same problem that was faced by the king of our story.

As of October 2014, the world's fastest computer was the Tianhe-2 at Sun Yat-sen University, Guangzhou, China. The Tianhe-2 is able to execute $33.86 \times 10^{15}$ operations per second. But even at that speed it could take more than seventy years to solve the subset sum problem using the brute force method when the initial set has 86 elements.

An algorithm like the brute force method, which requires up to $2^n$ operations to solve the subset sum problem for an input of size $n$, is said to be of **exponential time.**

In general, an algorithm is said to be of exponential time if, given an input of size $n$, it must perform up to $2^{p(n)}$ operations to solve the problem. (Here and below, $p(n)$ is a fixed polynomial of the form $c_0 + c_1 n^1 + c_2 n^2 + \cdots + c_k n^k$.)

Even though there are known to be more efficient methods for solving the subset sum problem than the brute force method, we do not at present know of a method that is truly efficient.

In particular, there is no known method capable of solving the problem in **polynomial time.**

In general, an algorithm is said to be of polynomial time if, given an input of size $n$, it must perform up to $p\left(n\right)$ operations to solve the problem (rather than up to $2^{p(n)}$ operations, as in the case of exponential time).

To illustrate the difference between exponential time and polynomial time consider the polynomial function $4n^2 + 31n^{17}$ and the exponential function $2^n$. For small values of $n$, the value of our polynomial function is greater than the value of our exponential function. $n = 2, 4 \cdot 2^2 + 31 \cdot 2^{17} = 4,063,248$, and $2^2 = 4$). But once $n$ gets big enough, the value of $2^n$ will always be greater than that of our polynomial function. For $n = 1000, 4 \cdot 1000^2 + 31 \cdot 1000^{17} \approx 10^{52}$, but $(2^{1000} \approx 10^{301})$.

The class of problems that can be solved using algorithms of polynomial time is usually called $P$.

This set is of considerable interest to computer scientists because any problem that can be solved using a procedure that is efficient enough to be used in practice will be in $P$.

University of Texas computer scientist Scott Aaronson visited *Paradox*! Here he is on exponential time problems.

## Video: Aaronson on When Problems are Exponential Time

proof of my question,

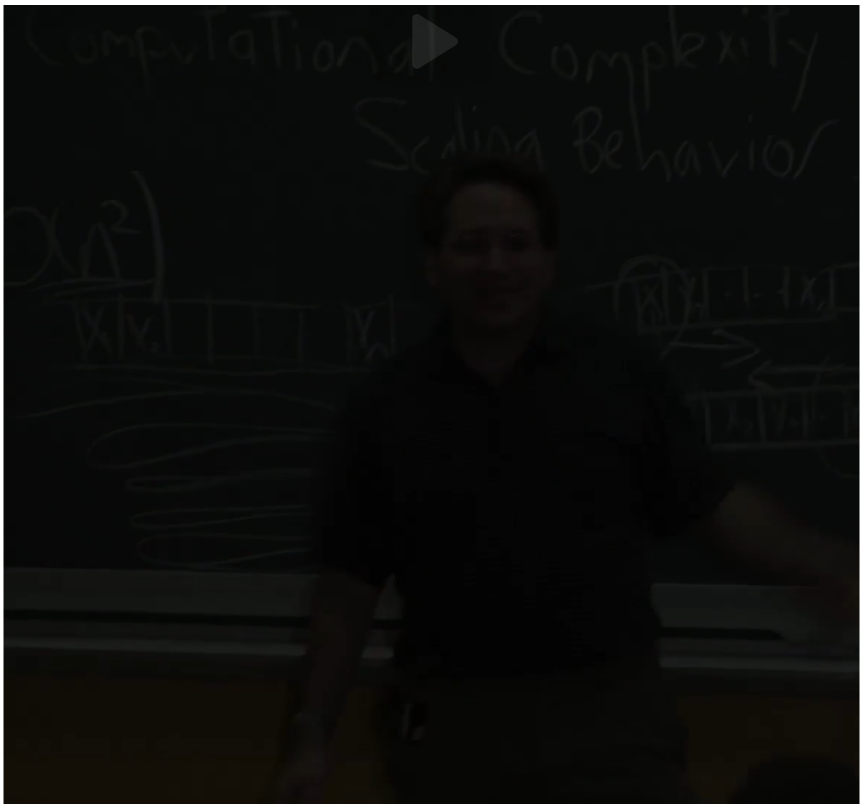
like the Riemann hypothesis.

That's certainly a program I could write.

Then the only problem is that I run it,

and then I don't see it stop before the universe goes cold,

**because this is a 2 to the n complexity.**



End of transcript. Skip to the start.

3:02 / 3:02 1.50x

## Video
Download video file

## Transcripts
Download SubRip (.srt) file
Download Text (.txt) file

## Discussion

Hide Discussion

**Topic:** Week 9 / Exponential Time and Polynomial Time

Add a Post

Show all posts by recent activity

There are no posts in this topic yet.

✖