Does $P = NP$?

A real-life mathematical problem

- This will be a talk about the mathematics of *algorithms*. An algorithm is a set of rules and instructions used to solve a real-life problem. Often this algorithm will then be run on a computer.

- One of the most important problems here is "does P = NP?" Mathematicians and computer scientists have been working on this problem for 25 years, but have not yet solved it.

Example

- As a hypothetical example, suppose that you want to build a reverse phone directory of Sydney. (A reverse phone directory has the *phone numbers* in order, and gives the name for each number).

- Your only tools are (a) a Pentium III computer; and (b) a CD-ROM version of the Telstra White Pages. These Pages contain the names and numbers of $N = 3,000,000$, in alphabetical order of the names.

- You'd like to be done with this by dinner-time.

First try: Insertion sort

- The most obvious thing to try is to start with an empty list and insert the numbers and names in one by one in order. This is known as *insertion sort*.
- For instance, suppose we have already sorted the first 5 names in the white pages:

$$
\begin{array}{lll}
1. & 82490931 & Abandowitz, E. \\
2. & 91435124 & Abadeen, W. \\
3. & 93210946 & Abe, L. \\
4. & 94029382 & Abacan, M. \\
5. & 98371342 & Aarden, J.
\end{array}
$$

- The next entry in the White Pages is "Abraham, S. - 93948234". We would search our list for where 93948234 lies, i.e. between position 3 and 4. Then we would insert this entry in, and bump all the later names up by one:

1. 82490931 $Abandowitz, E.$
2. 91435124 $Abadeen, W.$
3. 93210946 $Abe, L.$
4. 93948234 $Abraham, S.$
5. 94029382 $Abacan, M.$
6. 98371342 $Aarden, J.$

- We then repeat this 2,999,994 more times.

How long will insertion sort take?

- Insertion sort is slow. Suppose that we've already inserted 1,000,000 numbers and names, and are just about to insert the $1,000,0001^{th}$.
- To perform the insertion, we have to (a) find the place to insert the name, then (b) bump all the names after this up by one. On the average, it will take about 500,000 searches to find the place to insert, and another 500,000 commands to bump the names. So we're looking at about 1,000,000 commands just to insert the $1,000,001^{th}$ number.
- This means that the total number of steps needed is about

  $$0+1+2+3+\ldots+2,999,999 \approx 4,500,000,000,000.$$

- On a Pentium III, this will take about 50 hours.
- One could use some fancy programming tricks to speed this up a bit, but only by a factor of 10 or so.

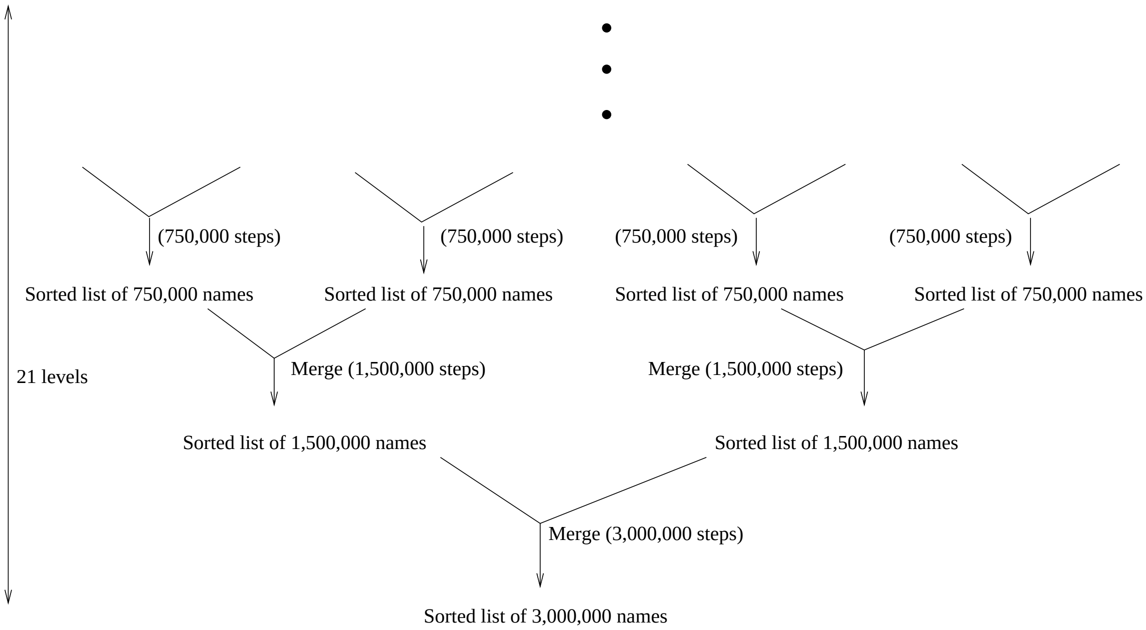
A better algorithm: Merge sort

- You could try to speed up insertion sort by programming more efficiently, or buying a faster computer. But a much cheaper thing to do is to come up with a better algorithm. One such algorithm is "Merge sort".

- Merge sort is a "divide and conquer" strategy, and works like this. Take the white pages and divide it into two equal halves ("A-M" and "N-Z"). Sort the two halves separately. Then merge the two sorted lists together.

- Of course, this begs the question of how to sort the two halves. The answer is to use Merge sort again, i.e. divide each half into equal quarters, sort each quarter separately, and then merge them together. To sort the quarters, you divide up into eighths, and so forth, until you are down to just sorting one or two names, which is very easy.

- Merging two lists of size N takes about $2N$ steps (Why?). Thus, for instance, the merging the two halves of the White Pages together would take $3,000,000$ steps.
- Since $3,000,000 \sim 2^{21}$, one would need to do about 21 levels of Merge sort:



- The total number of steps needed is therefore about
$$\approx 3,000,000 * 21 = 63,000,000.$$
- A Pentium III could do this in about 1 second (give or take a factor of 10).
- The task of sorting N objects is a "polynomial time" algorithm, because the number of steps needed is

at most a polynomial in N (N^2 for insertion sort, $N \log_2 N$ for Merge sort). The set of all polynomial time tasks is called $\mathbf{P}$.

Example: Booking examination rooms

- Suppose you are an administrator for UNSW. Your job is to assign examination times for $N = 100$ courses. The exams must be in exam week, Monday to Friday, and either in the morning or afternoon, so there are 10 time slots available.
- This seems easy: just put 10 courses in each time slot. There is, however, a catch. There are 1000 students, taking three courses each. A student can't take two different exams at the same time, so you have to avoid clashes. In other words, if student X is taking courses A, B, C, then you have to assign different time slots to A, B, C.
- It may be that there are so many clashes that a time-table is impossible. However, we would like an algorithm which will provide a workable time-table whenever one exists.

First attempt: brute force search

- One thing you can do is try all the possible exam assignments one by one. After all, once you have chosen the exam times, it's an easy matter for the computer to check each student one by one and make sure there is no clash. (This takes about $1,000$ steps).

- Unfortunately, there are a large number of possible assignments. Each course has 10 choices of time slot, and there are 100 courses, so there are 10^{100} possibilities. So the total number of computations needed could be as bad as

$$10^{100} \times 1000 = 10^{103}.$$

- On a Pentium III, this would take about 10^{82} millennia.

- This is so slow that no amount of technological improvement can help:
- 1000x improvement in speed of computer: 10^{79} millennia
- Clever programming reducing number of steps by 1000: 10^{76} millennia
- Using 100 million computers linked up via the internet: 10^{68} millennia
- Eliminating 90% of the students: 10^{67} millennia

Is there a better algorithm?

- Clearly, brute force is not the right answer to this problem. Is there a better one?
- There are partial algorithms for this problem which work 90% of the time, or only avoid clashes for 95% of the students, etc., but this is not completely satisfactory.
- The time-tabling problem is an example of a *polynomially-verifiable* problem: once you actually have a time-table, it is very quick (polynomial time) to check whether the time-table works or not, but it's very difficult to find the time-table in the first place. The class of all such problems is called **NP**.
- The **P** = **NP** problem asks: are all polynomially verifiable problems solvable in polynomial time? If this is true, then many problems of the above type (e.g. airplane scheduling, or any other matching of resources to needs) would be solvable very quickly, and this would have tremendous economic consequences.
- On the other hand, most encryption and security procedures (e.g. the encryption of credit card transactions on the internet) rely on the assumption that

$\mathbf{P} \neq \mathbf{NP}$ (because the problem of cracking an encryption code is usually **NP**).

- Despite 25 years on work on this problem, we are nowhere near a solution. To give some idea of its importance, the Clay Mathematical Institute in Boston has offered US $1 million for a proof of either $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$!
- The time-tabling problem is an example of a **NP**-*complete* problem. What this means is that if you can figure out how to solve the time-tabling problem in polynomial time, then you can solve all other **NP** problems in polynomial time (thus showing that $\mathbf{P} = \mathbf{NP}$). There are many other known examples of **NP**-complete problems.

Further reading

- The Clay Institute page for the $\mathbf{P} = \mathbf{NP}$ problem is at `http://www.claymath.org/prize_problems/p_vs_np.htm`
- The book “Computers and Intractibility, a guide to the theory of NP-completeness”, by M. Garey and D. Johnson (W.H. Freeman and Co., San Francisco, 1979) is a thorough introduction to the subject, and contains a list of over 300 **NP**-complete problems.