The Hardest Problem

Brian Heinold

Efficiency

- Computer scientists (and mathematicians) are interested in how fast an algorithm runs.

Efficiency

- Computer scientists (and mathematicians) are interested in how fast an algorithm runs.
- Example: Solve $ax + b = c$.

Efficiency

- Computer scientists (and mathematicians) are interested in how fast an algorithm runs.
- Example: Solve $ax + b = c$.
- Algorithm for solution: subtract b from both sides, then divide by a.

Efficiency

- Computer scientists (and mathematicians) are interested in how fast an algorithm runs.
- Example: Solve $ax + b = c$.
- Algorithm for solution: subtract b from both sides, then divide by a.
- This is fast. This takes about the same amount of time for all reasonably-sized values of a, b, c.

Efficiency

- Computer scientists (and mathematicians) are interested in how fast an algorithm runs.
- Example: Solve $ax + b = c$.
- Algorithm for solution: subtract b from both sides, then divide by a.
- This is fast. This takes about the same amount of time for all reasonably-sized values of a, b, c.
- We call this a *constant-time algorithm* ($O(1)$).

A linear algorithm

- Example: Add up all the elements in a list.

```
total = 0
for i in range(len(L)):
    total = total + L[i]
```

A linear algorithm

- Example: Add up all the elements in a list.

```
total = 0
for i in range(len(L)):
    total = total + L[i]
```

- If n items in list, need to look at all of them.

A linear algorithm

- Example: Add up all the elements in a list.

```
total = 0
for i in range(len(L)):
    total = total + L[i]
```

- If n items in list, need to look at all of them.
- This is a *linear* ($O(n)$) algorithm.

A linear algorithm

- Example: Add up all the elements in a list.

```
total = 0
for i in range(len(L)):
    total = total + L[i]
```

- If n items in list, need to look at all of them.
- This is a *linear* ($O(n)$) algorithm.
- Note: Actual running time might be something like $14.7n + .0025$, but we don't care about the constants, just the order of growth.

A quadratic algorithm

- Example: Add up all the elements in an $n \times n$ array.

$$\begin{array}{ccccc} 2 & 3 & 5 & 8 & 3 \\ 1 & 0 & 4 & 8 & 0 \\ 6 & 6 & 3 & 9 & 1 \\ 8 & 4 & 3 & 7 & 4 \\ 3 & 1 & 5 & 8 & 5 \end{array}$$

```
for i in range(len(L)):
    for j in range(len(L[i])):
        total = total + L[i][j]
```

A quadratic algorithm

- Example: Add up all the elements in an $n \times n$ array.

$$\begin{matrix} 2 & 3 & 5 & 8 & 3 \\ 1 & 0 & 4 & 8 & 0 \\ 6 & 6 & 3 & 9 & 1 \\ 8 & 4 & 3 & 7 & 4 \\ 3 & 1 & 5 & 8 & 5 \end{matrix}$$

```
for i in range(len(L)):
    for j in range(len(L[i])):
        total = total + L[i][j]
```

- This is a *quadratic* ($O(n^2)$) algorithm.

Long division

- The familiar process from grade school:

```
      14...
18 |26532109
   -18
    85
   -72
    133
      ...
```

- Just keep repeating the process of dividing, subtracting, and bringing down the next digit.

Long division

- The familiar process from grade school:

$$
\begin{array}{rl}
 & \;14\cdots \\
18 & \big)\overline{26532109} \\
 & \underline{-18} \\
 & \;\;\;85 \\
 & \;\underline{-72} \\
 & \;\;\;133 \\
 & \;\;\;\;\;\;\cdots
\end{array}
$$

- Just keep repeating the process of dividing, subtracting, and bringing down the next digit.
- When dividing into an n-digit number, this takes n steps.

Long division

- The familiar process from grade school:

```
        14 · · ·
  18 |26532109
     -18
      85
     -72
      133
        · · ·
```

- Just keep repeating the process of dividing, subtracting, and bringing down the next digit.
- When dividing into an n-digit number, this takes n steps.
- So this is a linear ($O(n)$) algorithm.

Long division

- The familiar process from grade school:

```
       14...
  18 |26532109
     -18
      85
     -72
      133
        ...
```

- Just keep repeating the process of dividing, subtracting, and bringing down the next digit.
- When dividing into an n-digit number, this takes n steps.
- So this is a linear $(O(n))$ algorithm.
- Adding another digit to the number just adds a little more time.

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$
- If an algorithm's running time is a polynomial, we say it runs in *polynomial time*.

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$
- If an algorithm's running time is a polynomial, we say it runs in *polynomial time*.
- Common tasks that take polynomial time:
    - Grade school arithmetic

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$
- If an algorithm's running time is a polynomial, we say it runs in *polynomial time*.
- Common tasks that take polynomial time:
  - Grade school arithmetic
  - Most simple programming tasks

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$
- If an algorithm's running time is a polynomial, we say it runs in *polynomial time*.
- Common tasks that take polynomial time:
  - Grade school arithmetic
  - Most simple programming tasks
  - Sorting a list: $O(n^2)$ or better

Polynomial time

- Polynomials: 1, n, n^2, $n^3 + 3n^2 + n + 1$
- If an algorithm's running time is a polynomial, we say it runs in *polynomial time*.
- Common tasks that take polynomial time:
  - Grade school arithmetic
  - Most simple programming tasks
  - Sorting a list: $O(n^2)$ or better
  - Solving a system of n equations: $O(n^3)$

A question

- Are all problems solvable in polynomial time?

A question

- Are all problems solvable in polynomial time?
- No.

A question

- Are all problems solvable in polynomial time?
- No.
- For example, find all the subsets of $\{1, 2, 3, \ldots, n\}$.

A question

- Are all problems solvable in polynomial time?
- No.
- For example, find all the subsets of $\{1, 2, 3, \dots, n\}$.
- There are lots: $\{1, 2, 3\}$, $\{4, 5\}$, $\{7, 15, 27, 48, 76\}$, ...

A question

- Are all problems solvable in polynomial time?
- No.
- For example, find all the subsets of $\{1, 2, 3, \ldots, n\}$.
- There are lots: $\{1, 2, 3\}$, $\{4, 5\}$, $\{7, 15, 27, 48, 76\}$, $\ldots$
- There are 2^n subsets in total, so no algorithm can do better than $O(2^n)$ time.

A question

- Are all problems solvable in polynomial time?
- No.
- For example, find all the subsets of $\{1, 2, 3, \ldots, n\}$.
- There are lots: $\{1, 2, 3\}$, $\{4, 5\}$, $\{7, 15, 27, 48, 76\}$, $\ldots$
- There are 2^n subsets in total, so no algorithm can do better than $O(2^n)$ time.
- This is called *exponential time*.

A question

- Are all problems solvable in polynomial time?
- No.
- For example, find all the subsets of $\{1, 2, 3, \ldots, n\}$.
- There are lots: $\{1, 2, 3\}$, $\{4, 5\}$, $\{7, 15, 27, 48, 76\}$, $\ldots$
- There are 2^n subsets in total, so no algorithm can do better than $O(2^n)$ time.
- This is called *exponential time*.
- Adding another element doubles the amount of time.

Difference between polynomial and exponentials

- Difference between n^2 and 2^n:

  $10^2 = 100$
  $2^{10} = 1024$

Difference between polynomial and exponentials

- Difference between n^2 and 2^n:

  $10^2 = 100$
  $2^{10} = 1024$

  $100^2 = 10{,}000$
  $2^{100} = 1.27 \times 10^{30}$

Difference between polynomial and exponentials

- Difference between n^2 and 2^n:

$10^2 = 100$
$2^{10} = 1024$

$100^2 = 10{,}000$
$2^{100} = 1.27 \times 10^{30}$

$1000^2 = 1{,}000{,}000$
$2^{1000} = 1.07 \times 10^{301}$

Difference between polynomial and exponentials

- Difference between n^2 and 2^n:

  $10^2 = 100$
  $2^{10} = 1024$

  $100^2 = 10{,}000$
  $2^{100} = 1.27 \times 10^{30}$

  $1000^2 = 1{,}000{,}000$
  $2^{1000} = 1.07 \times 10^{301}$

- Sum the elements in a 1000×1000 array? No problem.

Difference between polynomial and exponentials

- Difference between n^2 and 2^n:

  $10^2 = 100$
  $2^{10} = 1024$

  $100^2 = 10{,}000$
  $2^{100} = 1.27 \times 10^{30}$

  $1000^2 = 1{,}000{,}000$
  $2^{1000} = 1.07 \times 10^{301}$

- Sum the elements in a 1000×1000 array? No problem.
- List all the subsets of $\{1, 2, \ldots, 1000\}$? No chance.

Difference between polynomial and exponentials

- Linear algorithm: if you double the problem size, you double the running time

Difference between polynomial and exponentials

- Linear algorithm: if you double the problem size, you double the running time
- Quadratic: if you double the problem size, you quadruple the running time

Difference between polynomial and exponentials

- Linear algorithm: if you double the problem size, you double the running time
- Quadratic: if you double the problem size, you quadruple the running time
- Exponential: if you add 1 to problem size, you double running the time

Sudoku

It's not easy to solve a Sudoku puzzle.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

From `http://en.wikipedia.org/wiki/Sudoku`

Sudoku

But it is easy to check a solution.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

From http://en.wikipedia.org/wiki/Sudoku

Subset sum

Given a set of integers, is it possible to find a subset of them summing to exactly 0?

$$\{-20, -12, -10, -7, -3, 4, 5, 9, 18, 25\}$$

Not too easy…

Subset sum

Given a set of integers, is it possible to find a subset of them summing to exactly 0?

$$\{-20, -12, -10, -7, -3, 4, 5, 9, 18, 25\}$$

But it is easy to verify that $\{-12, -10, -3, 25\}$ works.

Subset sum

Given a set of integers, is it possible to find a subset of them summing to exactly 0?

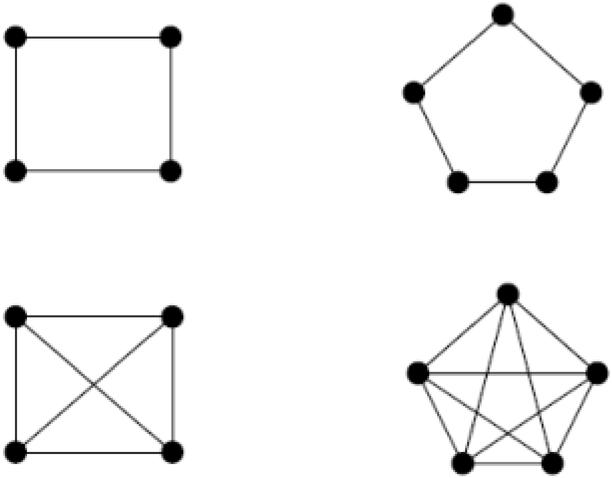
$$\{-20, -12, -10, -7, -3, 4, 5, 9, 18, 25\}$$

But it is easy to verify that $\{-12, -10, -3, 25\}$ works.

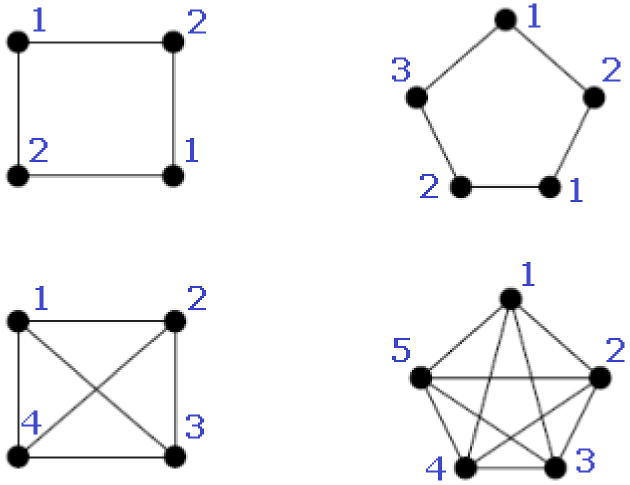
If the the set were 1000 elements long, it could be very difficult to find a solution, but still easy to check a solution (just add up 1000 numbers).

Graph coloring

Assign labels so that adjacent vertices get different labels.

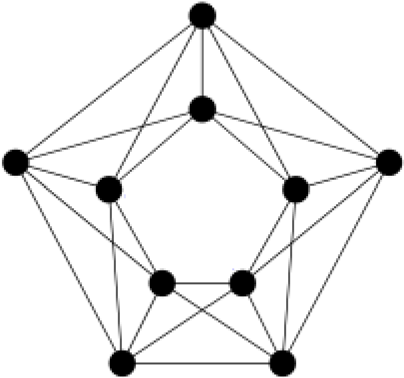


Assign labels so that adjacent vertices get different labels.



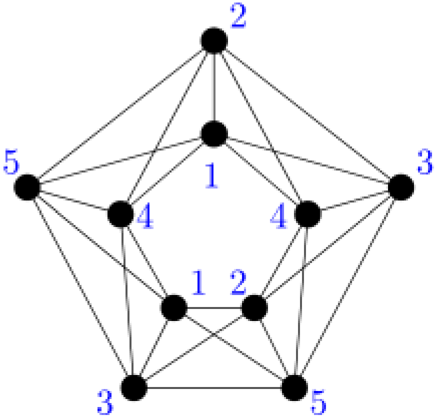
Can we color this graph with 5 colors?

Can be tricky to find a solution.



Can we color this graph with 5 colors?

But it's easy to check that a given solution works.



P and NP

- P = class of problems solvable in polynomial time
- NP = class of problems where we can verify a solution in polynomial time

P and NP

- P = class of problems solvable in polynomial time
- NP = class of problems where we can verify a solution in polynomial time
- The big problem: Does P = NP?

P and NP

- P = class of problems solvable in polynomial time
- NP = class of problems where we can verify a solution in polynomial time
- The big problem: Does P = NP?
- In other words, if we can efficiently *check* if a solution is correct, does that mean we can efficiently *solve* the problem?

- There is a whole collection of problems, considered to be the hardest ones in NP.

NP-Completeness

- There is a whole collection of problems, considered to be the hardest ones in NP.
- These are called *NP-Complete* problems.

- There is a whole collection of problems, considered to be the hardest ones in NP.
- These are called *NP-Complete* problems.
- They are of enormous practical interest.

NP-Completeness

- There is a whole collection of problems, considered to be the hardest ones in NP.
- These are called *NP-Complete* problems.
- They are of enormous practical interest.
- Here are a few...

$n \times n$ Sudoku

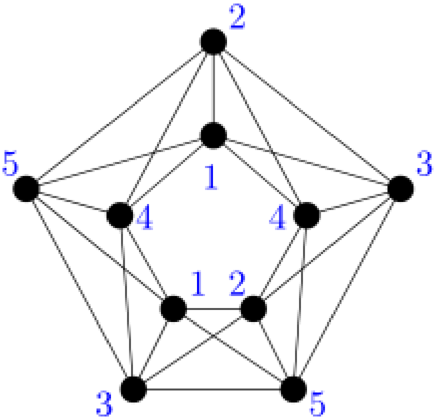
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Subset sum

Given a set of integers, is it possible to find a subset of them summing to exactly 0?

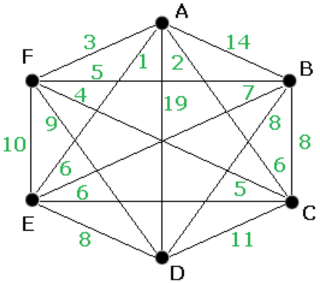
$$\{-20, -12, -10, -7, -3, 4, 5, 9, 18, 25\}$$

Graph Coloring



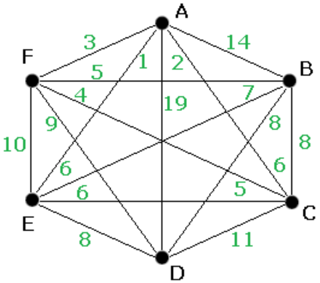
The traveling salesman problem

- Salesman needs to visit all 6 cities, needs to do so as cheaply as possible.



The traveling salesman problem

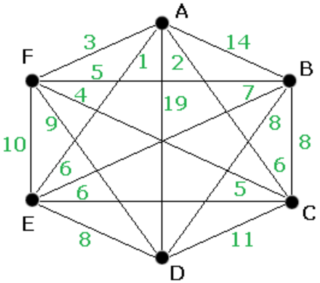
- Salesman needs to visit all 6 cities, needs to do so as cheaply as possible.



- There are $6! = 720$ possible routes.

The traveling salesman problem

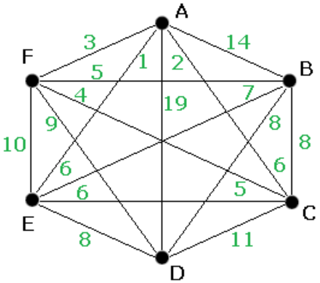
- Salesman needs to visit all 6 cities, needs to do so as cheaply as possible.



- There are $6! = 720$ possible routes.
- For n cities, to checking all possibilities is an $O(n!)$.

The traveling salesman problem

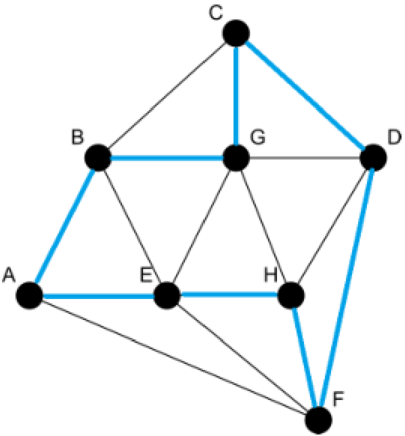
- Salesman needs to visit all 6 cities, needs to do so as cheaply as possible.



- There are $6! = 720$ possible routes.
- For n cities, to checking all possibilities is an $O(n!)$.
- Can be solved in exponential time, but no one knows if it can be solved in polynomial time.

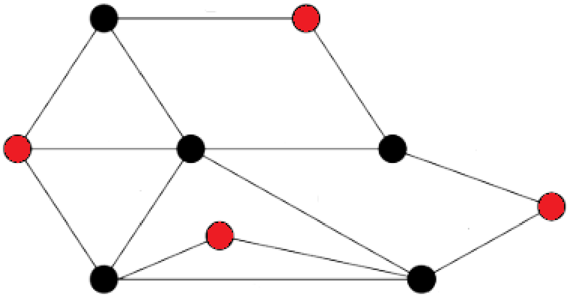
Hamiltonian cycle

Is it possible to visit each vertex exactly once and end up where you started?



Independent set

An independent set is a collection of vertices, none of which are adjacent to each other. Does there exist an independent set of a given size?



Some more problems

- Reconstructing a DNA sequence from fragments
- Ground state in the Ising model of phase transitions
- Finding Nash Equilbriums
- Optimal protein threading
- Scheduling jobs on two identical machines to finish in a given time
- Given costs, returns, and risks for a series of investments, find a strategy to minimize risk

More about NP Completeness

- There are more than 1000 known NP-complete problems.

More about NP Completeness

- There are more than 1000 known NP-complete problems.
- They all share the property that it is easy to verify a solution.

More about NP Completeness

- There are more than 1000 known NP-complete problems.
- They all share the property that it is easy to verify a solution.
- But they are all hard to solve.

More about NP Completeness

- There are more than 1000 known NP-complete problems.
- They all share the property that it is easy to verify a solution.
- But they are all hard to solve.
- Each problem in the collection *reduces* to the others.

More about NP Completeness

- There are more than 1000 known NP-complete problems.
- They all share the property that it is easy to verify a solution.
- But they are all hard to solve.
- Each problem in the collection *reduces* to the others.
- That is, a solution to any one could be used to quickly find a solution to any other one.

More about NP Completeness

- There are more than 1000 known NP-complete problems.
- They all share the property that it is easy to verify a solution.
- But they are all hard to solve.
- Each problem in the collection *reduces* to the others.
- That is, a solution to any one could be used to quickly find a solution to any other one.
- Finding a fast (polynomial-time) solution to any one of these would give a fast solution to all the others.

It's hard to predict the difficulty of a problem. Examples:

Easy vs. Hard Problems

It's hard to predict the difficulty of a problem. Examples:

- Easy: Round trip in a graph, visiting every *edge* exactly once.

  Hard: Round trip in a graph, visiting every *vertex* exactly once.

Easy vs. Hard Problems

It's hard to predict the difficulty of a problem. Examples:

- Easy: Round trip in a graph, visiting every *edge* exactly once.
  Hard: Round trip in a graph, visiting every *vertex* exactly once.
- Easy: Finding the *shortest* path between two given vertices.
  Hard: Finding the *longest* path between two given vertices.

Easy vs. Hard Problems

It's hard to predict the difficulty of a problem. Examples:

- Easy: Round trip in a graph, visiting every *edge* exactly once.
  Hard: Round trip in a graph, visiting every *vertex* exactly once.
- Easy: Finding the *shortest* path between two given vertices.
  Hard: Finding the *longest* path between two given vertices.
- Easy: Match up people into compatible *teams of 2*.
  Hard: Match up people into compatible *teams of 3*.

Easy vs. Hard Problems

It's hard to predict the difficulty of a problem. Examples:

- Easy: Round trip in a graph, visiting every *edge* exactly once.
  Hard: Round trip in a graph, visiting every *vertex* exactly once.
- Easy: Finding the *shortest* path between two given vertices.
  Hard: Finding the *longest* path between two given vertices.
- Easy: Match up people into compatible *teams of 2*.
  Hard: Match up people into compatible *teams of 3*.
- Easy: $\mathbb{R}$ solutions to systems of linear inequalities.
- Hard: *Integer* solutions to systems of linear inequalities.

Origin of the name

- “NP” does not stand for “not polynomial”.

Origin of the name

- "NP" does not stand for "not polynomial".
- A *Turing Machine* is a theoretical model for computing.

Origin of the name

- "NP" does not stand for "not polynomial".
- A *Turing Machine* is a theoretical model for computing.
- It is a (hypothetical) machine that reads a tape and changes what is written on the tape based on the current state of the machine.

Origin of the name

- “NP” does not stand for “not polynomial”.
- A *Turing Machine* is a theoretical model for computing.
- It is a (hypothetical) machine that reads a tape and changes what is written on the tape based on the current state of the machine.
- In theory, anything that can be computed seems to be able to be computed by a Turing Machine.

Origin of the name

- "NP" does not stand for "not polynomial".
- A *Turing Machine* is a theoretical model for computing.
- It is a (hypothetical) machine that reads a tape and changes what is written on the tape based on the current state of the machine.
- In theory, anything that can be computed seems to be able to be computed by a Turing Machine.
- A Nondeterministic Turing Machine extends the ordinary Turing Machine (roughly) by allowing for infinite parallelism. We can split the computation into infinitely many parallel components, though the components cannot communicate with each other.

Origin of the name

- "NP" does not stand for "not polynomial".
- A *Turing Machine* is a theoretical model for computing.
- It is a (hypothetical) machine that reads a tape and changes what is written on the tape based on the current state of the machine.
- In theory, anything that can be computed seems to be able to be computed by a Turing Machine.
- A Nondeterministic Turing Machine extends the ordinary Turing Machine (roughly) by allowing for infinite parallelism. We can split the computation into infinitely many parallel components, though the components cannot communicate with each other.
- Problems in NP are those that are computable by a **N**ondeterministic Turing Machine in **P**olynomial Time.

Origin of the name

- "NP" does not stand for "not polynomial".
- A *Turing Machine* is a theoretical model for computing.
- It is a (hypothetical) machine that reads a tape and changes what is written on the tape based on the current state of the machine.
- In theory, anything that can be computed seems to be able to be computed by a Turing Machine.
- A Nondeterministic Turing Machine extends the ordinary Turing Machine (roughly) by allowing for infinite parallelism. We can split the computation into infinitely many parallel components, though the components cannot communicate with each other.
- Problems in NP are those that are computable by a **N**ondeterministic Turing Machine in **P**olynomial Time.
- This is actually equivalent to our earlier formulation about being verifiable in polynomial time.

More about P=NP

- First posed in a famous 1956 letter from Kurt Gödel to John Von Neumann, though it wasn't phrased in the modern way until the early 1970s.

More about P=NP

- First posed in a famous 1956 letter from Kurt Gödel to John Von Neumann, though it wasn't phrased in the modern way until the early 1970s.
- Most people think that P $\neq$ NP.

More about P=NP

- First posed in a famous 1956 letter from Kurt Gödel to John Von Neumann, though it wasn't phrased in the modern way until the early 1970s.
- Most people think that P $\neq$ NP.
- Some people think that the problem may be undecidable.

More about P=NP

- First posed in a famous 1956 letter from Kurt Gödel to John Von Neumann, though it wasn't phrased in the modern way until the early 1970s.
- Most people think that P $\neq$ NP.
- Some people think that the problem may be undecidable.
- That is, it may be mathematically impossible to decide the question one way or the other.

More about P=NP

- First posed in a famous 1956 letter from Kurt Gödel to John Von Neumann, though it wasn't phrased in the modern way until the early 1970s.
- Most people think that P $\neq$ NP.
- Some people think that the problem may be undecidable.
- That is, it may be mathematically impossible to decide the question one way or the other.
- It seems really hard to prove.

A $1 Million Question

- It is one of the Clay Mathematics Institute's million dollar problems.

A $1 Million Question

- It is one of the Clay Mathematics Institute's million dollar problems.
- Lance Fortnow:

> *A person who proves $P = NP$ would walk home from the Clay Institute not with [a] $1 million check but with seven.*

A \$1 Million Question

- It is one of the Clay Mathematics Institute's million dollar problems.
- Lance Fortnow:

> *A person who proves $P = NP$ would walk home from the Clay Institute not with [a] \$1 million check but with seven.*

(Because proving things (like the other six \$1 million problems) would become easy.)

Consequences

From "A Personal View of Average-Case Complexity" by Russsell Impagliazzo:

> *"Seemingly intractable algorithmic problems would become trivial. . . Programming languages would not need to involve instructions on how the computation should be performed, Instead, one would just specify the properties that a desired output should have in relation to the input."*

Consequences

From “A Personal View of Average-Case Complexity” by Russsell Impagliazzo:

> *“Seemingly intractable algorithmic problems would become trivial. . . Programming languages would not need to involve instructions on how the computation should be performed, Instead, one would just specify the properties that a desired output should have in relation to the input.”*
>
> *“One could use an ‘Occam’s Razor’ based inductive learning algorithm to automatically train a computer to perform any task that humans can.”*

From "A Personal View of Average-Case Complexity" by Russsell Impagliazzo:

> *"Seemingly intractable algorithmic problems would become trivial. . . Programming languages would not need to involve instructions on how the computation should be performed, Instead, one would just specify the properties that a desired output should have in relation to the input."*
>
> *"One could use an 'Occam's Razor' based inductive learning algorithm to automatically train a computer to perform any task that humans can."*
>
> *"In short, as soon as a feasible algorithm for an NP-complete problem is found, the capacity of computers will become that currently depicted in science fiction."*

Consequences

Scott Aaronson:

> *"There would be no special value in creative leaps, no fundamental gap between solving a problem and recognizing the solution once its found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett."*

Consequences

Brian Heinold:

> *"I don't think I'd like to live in such a world. In fact, I think it would be pretty boring."*

Importance of the P=NP problem

Fortnow:

> *"As we solve larger and more complex problems with greater computational power and cleverer algorithms, the problems we cannot tackle begin to stand out. The theory of NP-completeness helps us understand these limitations and the P versus NP problem begins to loom large not just as an interesting theoretical question in computer science, but as* ***a basic principle that permeates all the sciences****."*

Importance of the P=NP problem

Fortnow:

> *"As we solve larger and more complex problems with greater computational power and cleverer algorithms, the problems we cannot tackle begin to stand out. The theory of NP-completeness helps us understand these limitations and the P versus NP problem begins to loom large not just as an interesting theoretical question in computer science, but as* ***a basic principle that permeates all the sciences****."*

Aaronson (refering to the other Clay Institute problems):

> *"We are after not projective algebraic varieties or zeros of the Riemann zeta function, but the* ***nature of mathematical thought itself****."*

Further Reading

If you are interested, here are some good references:

- *The Golden Ticket: P, NP and the Search for the Impossible* by Lance Fortnow. Princeton University Press, 2013.
- *The Status of the P Versus NP Problem* by Lance Fortnow, Communications of the ACM, Vol. 52 No. 9, Pages 78-86.
`http://cacm.acm.org/magazines/2009/9/38904-the-status-of-the-p-versus-np-problem/fulltext`
- *A Most Profound Math Problem* by Alexander Nazaryan.
`http://www.newyorker.com/tech/elements/a-most-profound-math-problem`
- *A Personal View of Average-Case Complexity* by Russell Impagliazzo.
`http://cseweb.ucsd.edu/ russell/average.ps`
- *Reasons to Believe* by Scott Aaronson.
`http://www.scottaaronson.com/blog/?p=122`
- *The Scientific Case for* $P \neq NP$ by Scott Aaronson.
`http://www.scottaaronson.com/blog/?p=1720`
- *Algorithms, 4th Edition* by Sedgewick & Wayne. Pages 910-921.