**Uncountable to uncomputable**

However, the set of all programs (in your favorite language) is just $\Sigma^*$, for some finite alphabet $\Sigma$.

Hence the set of all programs is countable.

Hence the set of all “computable reals” is countable.

But $\mathbb{R}$ is uncountable.

Therefore there exist “uncomputable reals”.

**Computable Boolean functions**

An equivalence between languages and (Boolean-valued) functions:

function $f: \{0,1\}^* \rightarrow \{0,1\}$ $\equiv$ subset $L \subseteq \{0,1\}^*$

$L = \{x \in \{0,1\}^* : f(x) = 1\}$

$f(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L \end{cases}$

If $L$ is decidable we call $f$ **computable**, and vice versa.
Decidable languages

Definition:
A language \( L \subseteq \Sigma^* \) is **decidable** if there is a Turing Machine \( M \) which:

1. **Halts on every input** \( x \in \Sigma^* \).
2. Accepts inputs \( x \in L \) and rejects inputs \( x \notin L \).

Such a Turing Machine is called a **decider**. It ‘decides’ the language \( L \).

We like deciders. We don’t like TM’s that sometimes loop.

Encoding different objects with strings

Fix some alphabet \( \Sigma \).
We use the \( \langle \cdot \rangle \) notation to denote the encoding of an object as a string in \( \Sigma^* \)

**Examples:**
- \( \langle M \rangle \in \Sigma^* \) is the encoding a TM \( M \)
- \( \langle D \rangle \in \Sigma^* \) is the encoding a DFA \( D \)
- \( \langle M_1, M_2 \rangle \in \Sigma^* \) is the encoding of a pair of TMs \( M_1, M_2 \)
- \( \langle M, x \rangle \in \Sigma^* \) is the encoding a pair \( M, x \), where \( M \) is a TM, and \( x \in \Sigma^* \) is an input to \( M \)

Question:
- Is every language in \( \{0,1\}^* \) decidable?
- \( \iff \) Is every function \( f : \{0,1\}^* \to \{0,1\} \) computable?

Answer:
- No! Via a simple counting argument.
  - Every TM is encodable by a finite string.
  - Therefore the set of all TM’s is countable.
  - So the subset of all decider TM’s is countable.
  - Thus the set of all decidable languages is countable.
  - But the set of all languages is uncountable.
  - \( (|\mathcal{P}\{0,1\}^*| > |\{0,1\}^*|) \)
Is it just weirdo languages that no one would care about which are *undecidable*?

**Question:**
Is it just weirdo languages that no one would care about which are *undecidable*?

**Answer (due to Turing, 1936):**
Sadly, no.
There are some very reasonable languages we’d like to compute which are *undecidable*.

---

We need to write an autograder for `isPrime`

**Does they return True on exactly the same inputs?**

<table>
<thead>
<tr>
<th>student submission</th>
<th>the correct program</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isPrime</code></td>
<td><code>isPrime</code></td>
</tr>
</tbody>
</table>

**Kosbie’s version**

returns True on exactly same inputs?

<table>
<thead>
<tr>
<th>True or False</th>
</tr>
</thead>
</table>

**Does such a program exist?**

i.e., can we solve/decide the following?

EQ = \{ (M_1, M_2) : \text{M}_1 \text{ and } \text{M}_2 \text{ are TMs s.t. } L(\text{M}_1) = L(\text{M}_2) \}
Some undecidable languages

Given two TM descriptions, \( M_1 \) and \( M_2 \), do they act the same (accept/reject/loop) on all inputs?

Given the description of an algorithm, \( M \), does it print out “HELLO WORLD”?

This C program prints out all the lyrics of *The Twelve Days Of Christmas*.

\begin{verbatim}
main(_a) char *a; { return !a; } main(79, 13) main(97, 1... main(66, 0, a+1) +a); 1, 2, 3 main(41, a); 4, main (34, 27, a) &a == 2 7 < main (2, a, "Hello\%d\%"); b=16; b=727 main(...

Some undecidable languages

Given two TM descriptions, \( M_1 \) and \( M_2 \), do they act the same (accept/reject/loop) on all inputs?

Given the description of an algorithm, \( M \), does it print out “HELLO WORLD”?

Given a TM description \( M \) and an input \( x \), does \( M \) halt on input \( x \)?

Given a TM description \( M \), does \( M \) halt when the input is a blank tape?
Some uncomputable functions

This one is called

The Halting Problem.

Given a TM description \( \langle M \rangle \) and an input \( x \),
does \( M \) halt on input \( x \)?

Turing’s Theorem:
The Halting Problem is undecidable.

The Halting Problem is Undecidable

Theorem:
Let \( \text{HALTS} \subseteq \Sigma^* \) be the language
\[ \{ \langle M, x \rangle : M \text{ is a TM which halts on input } x \} \]
Then \( \text{HALTS} \) is undecidable.

Proof:
Assume for the sake of contradiction that
\( M_{\text{HALTS}} \) is a decider TM which decides \( \text{HALTS} \).

Here is the (high level) description of another
TM called \( D \), which uses \( M_{\text{HALTS}} \) as a subroutine:

Given as input \( \langle M \rangle \), the encoding of a TM \( M \):
\( D \) executes \( M_{\text{HALTS}}(\langle M, \langle M \rangle \rangle) \).
If this call accepts, \( D \) enters an infinite loop.
If this call rejects, \( D \) halts (say, it accepts).

In other words...
\( D(\langle M \rangle) \) loops if \( M(\langle M \rangle) \) halts,
halts if \( M(\langle M \rangle) \) loops.

The Halting Problem is Undecidable

Assume \( M_{\text{HALTS}} \) is a decider TM which decides \( \text{HALTS} \).
We can use it to construct a machine \( D \) such that
\( D(\langle M \rangle) \) loops if \( M(\langle M \rangle) \) halts,
halts if \( M(\langle M \rangle) \) loops.

Time for the contradiction:

Does \( D(\langle D \rangle) \) loop or halt?

By definition, if it loops it halts and if it halts it loops.

Contradiction.
BTW: last part of proof basically the same as Cantor’s Diagonal Argument.

\[ D(\langle M \rangle) \] loops if \( M(\langle M \rangle) \) halts, halts if \( M(\langle M \rangle) \) loops

The set of all TM's is countable, so list them all:

\[
\begin{array}{cccc}
\langle M_1 \rangle & \langle M_2 \rangle & \langle M_3 \rangle & \langle M_4 \rangle & \langle M_5 \rangle & \cdots \\
M_1 & \text{halts} & \text{halts} & \text{loops} & \text{halts} & \text{loops} \\
M_2 & \text{loops} & \text{loops} & \text{loops} & \text{loops} & \\
M_3 & \text{halts} & \text{loops} & \text{halts} & \text{halts} & \\
M_4 & \text{halts} & \text{halts} & \text{halts} & \text{loops} & \\
M_5 & \text{halts} & \text{loops} & \text{loops} & \text{halts} & \\
\vdots & & & & \\
\end{array}
\]

In our proof that HALTS is undecidable, we used a hypothetical TM deciding HALTS to derive a contradiction.

Having established the undecidability of HALTS, we can show further problems to be undecidable using the powerful tool of \textit{reductions}. 

\[ D(\langle M \rangle) \text{ loops if } M(\langle M \rangle) \text{ halts, halts if } M(\langle M \rangle) \text{ loops} \]
Reductions

Using one problem as a subroutine to solve another problem.

Informally, a reduction from A to B gives a way to solve problem A using a subroutine that can solve B.

Calculating the area of a rectangle reduces to calculating its length and height.
Solving a linear system \(Ax = b\) reduces to computing the matrix inverse \(A^{-1}\).

Language A reduces to language B means (informally): “there is a method that could be used to solve A if it has available to it a subroutine for solving B.”
The reduction gives such a method.

Formally a (Turing) reduction from A to B is an oracle Turing machine that decides A when run with an oracle for B.

Notation for A reduces to B: \(A \leq_T B\) (\(T\) stands for Turing). Think,
“A is no harder than B”
“A is at least as easy as B”

Reducing language A to B

Oracle O for B

Input \(x\) with Oracle access to O

\(y_1 \in B?\) Yes/No
\(y_2 \in B?\) Yes/No

Accept if \(x \in A\)
Reject if \(x \notin A\)

Fact: Suppose \(A \leq_T B\); i.e., A reduces to B.

If B is decidable, then A is also decidable.
(can replace the assumed oracle for B with a decider for B, and the reduction can run this decider whenever it needs to ascertain membership of some string in B)

Contrapositive: if A is undecidable then so is B.
Think: “B is at least as hard as A”

Reductions are the main technique for showing undecidability.
Reductions — examples

Theorem: 
\( \text{ACCEPTS} = \{ \langle M, x \rangle : M \text{ is a TM which accepts } x \} \)

is undecidable.

Proof: We'll prove \( \text{HALTS} \) reduces to \( \text{ACCEPTS} \).

Suppose \( O_{\text{ACCEPTS}} \) is an oracle for language \( \text{ACCEPTS} \).

Then here's a description of an oracle TM deciding \( \text{HALTS} \):

“Given \( \langle M, x \rangle \), run \( O_{\text{ACCEPTS}}(\langle M, x \rangle) \). If it accepts, then accept.

Reverse the accept & reject states in \( (M) \), forming \( \langle M' \rangle \).

Run \( O_{\text{ACCEPTS}}(\langle M', x \rangle) \). If it accepts (i.e., \( M \) rejects \( x \)), then accept.

Else reject.”

Interesting observation

To prove a negative result about computation (that a certain language is undecidable),
you actual construct an algorithm — namely, the reduction.

Reductions — another example

Theorem: \( \text{EMPTY} = \{ \langle M \rangle : M \text{ accepts no strings} \} \)
is undecidable.

Proof: Let's prove \( \text{ACCEPTS} \) reduces to \( \text{EMPTY} \).

This suffices, since we just showed \( \text{ACCEPTS} \) is undecidable.

So suppose \( O_{\text{EMPTY}} \) is an oracle for language \( \text{EMPTY} \).

Here's an oracle TM with oracle access to \( O_{\text{EMPTY}} \) deciding \( \text{ACCEPTS} \):

“Given \( \langle M, x \rangle \)...

Write down the description \( (N_x) \) of a TM \( N_x \) which does the following:

"On input \( y \), check if \( y = x \).

If not, reject. If so, simulate \( M \) on \( y \)."

Then call upon the oracle \( O_{\text{EMPTY}} \) on input \( \langle N_x \rangle \) and do the opposite.”

Correctness of reduction

Code for \( N_x \):

L(\( N_x \)) is either \( \{x\} \) or \( \emptyset \)

“On input \( y \),
check if \( y = x \).
If not, reject.
If so, simulate \( M \) on \( y \).”

And L(\( N_x \)) = \( \{x\} \) precisely when \( M \) accepts \( x \),
i.e., \( \langle M, x \rangle \in \text{ACCEPTS} \)

Important:
Reduction never runs \( N_x \);
it simply writes down the description \( (N_x) \) of \( N_x \)
and probes the oracle whether \( \langle N_x \rangle \in \text{EMPTY} \)
Schematic of the reduction
\( \text{ACCEPTS} \leq_T \text{EMPTY} \)

Oracle \( O_{\text{EMPTY}} \)

\((N_x) \in \text{EMPTY} ? \) Yes/No

Oracle TM deciding \( \text{ACCEPTS} \)
1. Write down description \((N_x)\)
2. Query oracle
3. Accept if \( O_{\text{EMPTY}} \) answers No, and reject if it answers Yes

Another example:
\( \text{ACCEPTS} \leq_T \text{INFINITE} = \{ (M) : M \text{ accepts infinitely many strings} \} \)

Oracle \( O_{\text{INFINITE}} \)

\((I_x) \in \text{INFINITE} ? \) Yes/No

Oracle TM deciding \( \text{ACCEPTS} \)
1. Write down description \((I_x)\)
2. Query oracle on \((I_x)\)
3. Accept if \( O_{\text{INFINITE}} \) answers Yes, and reject if it answers No

Note: Reduction is particularly simple: a single oracle query, and we just pass on answer to that query. Called “mapping reduction”

Undecidability galore
Similar reductions can show undecidability of telling if, given an input TM \((M)\), \( L(M) \) is:

- Finite
- Regular
- Contains 251 in binary
- Decidable
- Contains a string of length more than 251
- Contains only palindromes
- Etc etc

Essentially any non-trivial property of languages

Question:
Is everything about TMs undecidable?

Answer:
No!
Some problems about TMs behavior are decidable (as on your HW)
Question:
Do all undecidable problems involve TM's?

Answer:
No!
Some very different problems are undecidable!

**Post's Correspondence Problem**

Input: A finite collection of “dominoes”, having strings written on each half.

E.g.:
```
  a  bcc
  ab  c
```

Definition: A match is a sequence of dominoes, repetitions allowed, such that top string = bottom string.

Match:
```
  a  bcc  a  bcc
  ab  c  cabc  c
```

= abccabcc
= abccabcc

**Theorem (Post, 1946):** Undecidable.
There is no algorithm solving this problem.

(More formally, PCP = \{⟨Domino Set⟩ : there’s a match\} is an undecidable language.)
Post’s Correspondence Problem
Input: A finite collection of “dominoes”, having strings written on each half.
Task: Output YES if and only if there is a match.

Theorem (Post, 1946): Undecidable.
There is no algorithm solving this problem.

Two-second proof sketch:
Given a TM M, you can make a domino set such that the only matches are execution traces of M which end in the accepting state. Hence ACCEPTS ≤_T PCP.

Wang Tiles
Input: Finite collection of “Wang Tiles” (squares) with colors on the edges. E.g.,

Task: Output YES if and only if it’s possible to make an infinite grid from copies of them, where touching sides must color-match.

Theorem (Berger, 1966): Undecidable.

Mortal Matrices
Input: Two 15 × 15 matrices of integers, A & B.
Question: Is it possible to multiply A and B together (multiple times in any order) to get the 0 matrix?


Hilbert’s 10th problem
Input: Multivariate polynomial w/ integer coeffs.
Question: Does it have an integer root?

Hilbert’s 10th problem

Input: Multivariate polynomial w/ integer coeffs.

Question: Does it have an integer root? **Undecidable.**

Question: Does it have a real root? **Decidable.**

Question: Does it have a rational root? *Not known* if it’s decidable or not.

Entscheidungsproblem

Input: A sentence in first-order logic.

$$\neg \exists n, x, y, z \in \mathbb{N}: (n \geq 3) \land (x^n + y^n = z^n)$$

Question: Is it provable?

This is undecidable.

We’ll come back to this in the lecture on Gödel’s Incompleteness Theorem

Introduction to Time Complexity

What have we done so far?

What will we do next?
### What have we done so far?

- **Introduction to the course**
  
  > "Computer science is no more about computers than astronomy is about telescopes."

- **Strings and Encodings**

- **Formalization of computation/algorithm**
  - Deterministic Finite Automata
  - Turing Machines

### What is next?

- **The study of computation**
  - Computability/Decidability
    - Most problems are **undecidable**.
    - Some very interesting problems are **undecidable**.
  
  But many interesting problems are **decidable**!

### Why is computational complexity important?

- **complexity ~ practical computability**

  - **Computational Complexity** *(Practical Computability)*
    - How do we define computational complexity?
    - What is the right level of abstraction to use?
    - How do we analyze complexity?
    - What are some interesting problems to study?
    - What can we do to better understand the complexity of problems?
      
  - **Simulations** (e.g. of physical or biological systems)
    - tremendous applications in science, engineering, medicine, …
  
  - **Optimization problems**
    - arise in essentially every industry
  
  - **Social good**
    - finding efficient ways of helping others
  
  - **Artificial intelligence**

  - **Security, privacy, cryptography**
    - applications of computationally hard problems
      
  *list goes on*
Why is computational complexity important?

1 million dollar question
(or maybe 6 million dollar question)

Goals for next 2 lectures

1. What is the right way to study complexity?
   - using the right language and level of abstraction
     - upper bounds vs lower bounds
     - polynomial time vs exponential time

2. Appreciating the power of algorithms.
   - analyzing some cool (recursive) algorithms

Guiding principles

What is the right language and level of abstraction for studying computational complexity?

What is the meaning of:
“The (asymptotic) complexity of algorithm A is \( O(n^2) \).”
Size matters

running bazillion numbers > running 2 numbers.

Running time of an algorithm depends on input length.

\[ n = \frac{\text{input length}}{\text{size}} \]

- \( n \) is usually: # bits in a binary encoding of input.
- sometimes: explicitly defined to be something else.

GREAT IDEA # 1

Running time of an algorithm is a function of \( n \).

(But what is \( n \) going to be mapped to?)

We have to be careful

Value matters

Not all inputs are created equal!

Among all inputs of length \( n \):
- some might take 2 steps
- some might take bazillion steps.

Model matters
GREAT IDEA # 2

Running time of an algorithm is a \textit{worst-case} function of \( n \).

\[ n \mapsto \text{# steps taken by the worst input of length } n \]

Why worst-case?

We are not dogmatic about it.
- Can study “average-case” (random inputs)
- Can try to look at “typical” instances.
- Can do “smoothed analysis”.

\[ \ldots \]

\textbf{BUT worst-case analysis has its advantages:}
- An ironclad guarantee.
- Hard to define “typical” instances.
- Random instances are often not representative.
- Often much easier to analyze.

We have to be careful

\textbf{Model matters}

\[ \text{PAL} = \{ x \in \{0,1\}^* : x = x^R \} \]

How many steps required to decide \( \text{PAL} \)?

\textbf{Facts:}
- \( O(n^2) \) is the best for 1-tape TMs.
- \( O(n) \) is the best for 2-tape TMs.

\textbf{Alert:} Just because the obvious/standard way to solve it on single-tape TMs takes \( O(n^2) \) time does NOT mean there isn’t a better way
- Algorithms can be clever and unexpected.
- Lower bounds are a difficult enterprise.
Model matters

$L = \{0^k 1^k : k \geq 0\}$

How many steps required to decide $L$?

Facts:

$O(n \log n)$ is the best for 1-tape TMs.

$O(n)$ is the best for 2-tape TMs.

A function in Python:

```python
def twoFingers(s):
    lo = 0
    hi = len(s)-1
    while (lo < hi):
        if (s[lo] != 0 or s[hi] != 1):
            return False
        lo += 1
        hi -= 1
    return True
```

# of steps

1 1 1 3? 4? 5? 1

Seems like $O(n)$
Model matters

\[ L = \{0^k1^k : k \geq 0\} \]

if (s[lo] != 0 or s[hi] != 1):

Initially lo = 0, hi = n-1

Does it take n steps to go from s[0] to s[n-1]?

GREAT IDEA # 3

Choice of computational model

Which model is the best model?

No such thing.

Any reasonable model should be able to simulate another reasonable model with only a polynomial increase in running time.

- For example, 1-tape TM can simulate 2-tape TM with quadratic slowdown

A function in Python:

def twoFingers(s):
    lo = 0                      1
    hi = len(s)-1               1
    while (lo < hi):
        if (s[lo] != 0 or s[hi] != 1):
            return False         1
        lo += 1                  1
        hi -= 1                  1
    return True                log n ??

# of steps

76 77 78 79
GREAT IDEA # 4

All reasonable deterministic models are polynomially equivalent.

---

The Random-Access Machine (RAM) model

Good combination of reality/simplicity.

+ , - , / , *, <, >, etc.

e.g. 245*12894 takes 1 step

memory access

e.g. A[94] takes 1 step

Actually:

Assume arithmetic operations take 1 step IF numbers are bounded by poly(n).

Unless specified otherwise, we use this model.

(more on this next lecture)

---

Putting great ideas # 1, # 2 and # 3 together

---

Which model does this correspond to?

```python
def twoFingers(s):
    lo = 0
    hi = len(s)-1
    while (lo < hi):
        if (s[lo] != 0 or s[hi] != 1):
            return False
        lo += 1
        hi -= 1
    return True
```

\( O(n) \)
Defining running time

With a specific computational model in mind:

**Definition:**
The running time of an algorithm $A$ is a function

$$T_A : \mathbb{N} \rightarrow \mathbb{N}$$

defined by

$$T_A(n) = \max_{\text{instances } I \text{ of size } n} \{ \text{# steps } A \text{ takes on } I \}$$

Write $T(n)$ when $A$ is clear from context.

Need one more level of abstraction

There is a TM that decides PALINDROME in time

$$T(n) = \frac{1}{2} n^2 + \frac{3}{2} n + 1.$$  

Analogous to

"too many significant digits".

Palindrome TM has running time

$$T(n) = \frac{1}{2} n^2 + \frac{3}{2} n + 1$$

We want to use the right level of abstraction!

The key takeaway of this $T(n)$:

it’s “quadratic”; that is, proportional to $n^2$.

This leads us to...
Great Idea #5:

Big-O notation

The CS way to compare functions:

\[
O(\cdot) \quad \Omega(\cdot) \quad \Theta(\cdot)
\]

\[\leq \quad \geq \quad =\]

Big O notation

Our notation for \( \leq \) when comparing functions.

The right level of abstraction!

"Sweet spot"

- coarse enough to suppress details like programming language, compiler, architecture,…

- sharp enough to make comparisons between different algorithmic approaches.

Informal: An upper bound that ignores constant factors and ignores small \( n \).

For \( f, g : \mathbb{N}^+ \to \mathbb{R}^+ \):

\[
f(n) = O(g(n)) \quad \text{roughly means}
\]

\[
f(n) \leq g(n) \quad \text{up to a constant factor and ignoring small } n.
\]
For \( f, g : \mathbb{N}^+ \to \mathbb{R}^+ \)

\[
f(n) = O(g(n)) \quad \text{roughly means}
\]

\[
f(n) \leq Cg(n) \quad \text{up to a constant factor and ignoring small } n.
\]

**Formal Definition:**

For \( f, g : \mathbb{N}^+ \to \mathbb{R}^+ \), we say \( f(n) = O(g(n)) \) if there exist constants \( C, n_0 > 0 \) such that for all \( n \geq n_0 \), we have \( f(n) \leq Cg(n) \).

\( C \) and \( n_0 \) cannot depend on \( n \).
Same example:

\[ f(n) = 3n^2 + 10n + 30 \quad g(n) = n^2 \]

Take \( C = 43, \quad n_0 = 1 \) When \( n \geq 1, \)

\[ f(n) = 3n^2 + 10n + 30 \leq 3n^2 + 10n^2 + 30n^2 \]

\[ = 43n^2 = 43g(n) \]

Poll

Select all that apply.

\( \log(n!) \) is:

- \( O(n) \)
- \( O(n \log n) \)
- \( O(n^2) \)
- \( O(2^n) \)

Beats me

---

Big O practice

- \( 1000n \) is \( O(n) \)
- \( 0.000001n \) is \( O(n) \)
- \( 0.1n^3 + 10^9n + 10^{10000} \) is \( O(n^3) \)
- \( n \) is \( O(2^n) \)
- \( 0.0000001n^2 \) is not \( O(n) \)
- \( n \log n \) is not \( O(n) \)

Note on notation:

People usually write \( 4n^2 + 2n = O(n^2) \)

Another valid notation: \( 4n^2 + 2n \in O(n^2) \)

Run time scaling

<table>
<thead>
<tr>
<th>Running-time:</th>
<th>Ratio:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c \cdot n ) double the input</td>
<td>( c \cdot 2n ) 2</td>
</tr>
<tr>
<td>( c \cdot n^2 ) double the input</td>
<td>( c \cdot (2n)^2 ) 4</td>
</tr>
<tr>
<td>( c \cdot n^3 ) double the input</td>
<td>( c \cdot (2n)^3 ) 8</td>
</tr>
<tr>
<td>( c \cdot n^k ) double the input</td>
<td>( c \cdot (2n)^k ) ( 2^k ) (constant)</td>
</tr>
<tr>
<td>( c \cdot 2^n ) double the input</td>
<td>( c \cdot 2^{2^n} ) ( 2^n )</td>
</tr>
</tbody>
</table>
**Common Big O classes and their names**

- **Constant:** $O(1)$
- **Logarithmic:** $O(\log n)$
- **Square-root:** $O(\sqrt{n}) = O(n^{0.5})$
- **Linear:** $O(n)$
- **Quasi-linear:** $O(n \log n)$
- **Quadratic:** $O(n^2)$
- **Polynomial:** $O(n^k)$ (for some constant $k > 0$)
- **Exponential:** $O(2^n)$ (for some constant $k > 0$)

---

### $n$ vs $2^n$

<table>
<thead>
<tr>
<th>$2^n$</th>
<th>$n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
</tr>
<tr>
<td>1024</td>
<td>10</td>
</tr>
<tr>
<td>1,048,576</td>
<td>20</td>
</tr>
<tr>
<td>1,073,741,824</td>
<td>30</td>
</tr>
<tr>
<td>1,152,921,504,606,846,976</td>
<td>60</td>
</tr>
</tbody>
</table>

---

**Exponential running time**

If your algorithm has exponential running time e.g. $\sim 2^n$

Then it is (usually) not practical.
Given a list of integers, determine if there is a subset of the integers that sum to 0.

Exhaustive Search (Brute Force Search):
Try every possible subset and see if it sums to 0.
Number of subsets is $2^n$
So running time is at least $2^n$