Cardinality

Two sets have equal cardinality if there is a bijection ("1-to-1" and "onto" function) between them.

A set is countable if it is finite or has the same cardinality as the natural numbers.

Examples:

- $\Sigma^*$ is countable (think of strings as base-$|\Sigma|$ numerals)
- Even natural numbers are countable: $f(n) = 2n$
- The Rationals are countable

More cardinality facts

If $f: A \to B$ in an injective function ("1-1", but not necessarily “onto”), then

$$|A| \leq |B|$$

(Intuitive: $f$ is a bijection from $A$ to its range, which is a subset of $B$, & $B$ can’t be smaller than a subset of itself.)

Theorem (Cantor-Schroeder-Bernstein):

If $|A| \leq |B|$ and $|B| \leq |A|$ then $|A| = |B|$
The Reals are Uncountable

Suppose they were
List them in order
Define \( X \) so that its \( i \)th digit \( \neq \) \( i \)th digit of \( i \)th real
Then \( X \) is not in the list
Contradiction

A detail: avoid .000..., .9999... in \( X \)

Number of Languages in \( \Sigma^* \) is Uncountable

Suppose they were
List them in order
Define \( L \) so that \( w_i \in L \iff w_i \notin L \)
Then \( L \) is not in the list
Contradiction
\( \therefore \) the powerset of any countable set is uncountable

Are All Languages Regular?

\( \Sigma \) is finite (for any alphabet \( \Sigma \))
\( \Sigma^* \) is countably infinite
Let \( \Delta = \Sigma \cup \{\varepsilon, \emptyset, \cup, \star, (, )\} \)
\( \Delta \) is finite, so \( \Delta^* \) is also countably infinite
Every regular lang. \( R = L(x) \) for some \( x \in \Delta^* \)
\( \therefore \) the set of regular languages is countable
But the set of all languages over \( \Sigma \) (the powerset of \( \Sigma^* \)) is uncountable
\( \therefore \) non-regular languages exist!
(In fact, “most” languages are non-regular.)

The same is true for any real “programming system” I can imagine – programs are finite strings from a finite alphabet, so there are only countably many of them, yet there are uncountably many languages, so there must be some you can’t compute...

Above is somewhat unsatisfying – they exist, but what does one “look like”? What’s a concrete example?

Next few lectures give specific examples of non-regular languages. And proof techniques to show such facts – for such and such a language, none of the infinitely many DFAs correctly recognize it.
Some Examples

\[ L_3 = \{ w \text{ } w^{-1} \} \]

Intuitively, a DFA accepting \( L_3 \) must “remember” the entire left half as it crosses the middle. “Memory” = “states”. As \(|w| \to \infty\), this will overwhelm any finite memory. We make this intuition rigorous below...

\( L_3 \) is not a Regular Language

Proof: For a DFA \( M = (Q, \Sigma, \delta, q_0, F) \), suppose \( M \) ends in the same state \( q \in Q \) when reading \( x \) as it does when reading \( y \), \( x \neq y \). Then for any \( z \), either both \( xz \) and \( yz \) are in \( L(M) \) or neither is.

Let \( \Sigma = \{a, b\} \), \(|Q| = p \), and pick \( k \) so that \( 2^k > p \). Consider all \( n=2^k \) length \( k \) strings \( w_1, w_2, ..., w_n \). Consider the set of states \( M \) is in after reading each of these strings. By the Pigeon Hole Principle there must be some state \( q \in Q \) and some \( w_i \neq w_j \) such that both take \( M \) to \( q \). But then \( M \) must either accept both of \( w_iw_i \) and \( w_jw_j \) or neither. In either case, \( L(M) \neq L_3 \), since one is in \( L_3 \), but the other is not.

In pictures:

Since \( 2^k > p \), list of state names

\[ r_i = r_j, \text{ i.e.,} \]

\[ r_i = r_j \text{ (but } w_iw_i \text{, } w_jw_j \text{)} \]

\[ \text{as mentioned such} \]
$L_3 = \{ \text{ww} \mid w \in \{a,b\}^* \}$ is not regular:
Alternate Proof

Assume $L_3$ is regular. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing $L_3$. Let $p = |Q|$. Consider the $p+1$ strings $x_i = a^i b$, $0 \leq i \leq p$. Again, by the Pigeon Hole Principle, $\exists q \in Q$ and $\exists 0 \leq i < j \leq p$ s.t. $M$ reaches $q$ from $q_0$ on both $x_i$ and $x_j$. Since $M$ accepts both $x_i x_i$ and $x_j x_j$, it also accepts $x_j x_i = a^j b a^i b$. But $j > i$, so total length is odd or both $b$'s in right half. Either way, $x_j x_i \not\in L_3$, a contradiction. Hence $L_3$ is not regular.

A third way: feed $M$ many $a$'s; eventually it will loop. Say $a^i$ gets to $q$, then $a^j$ more revisits. Again, exploit this to reach a (many) contradictions.

Notes on these proofs
All versions are proof by contradiction: assume some DFA $M$ accepts $L_3$. $M$ of course has some fixed (but unknown) number of states, $p$. All versions also relied on the intuition that to accept $L_3$, you need to "remember" the left half of the string when you reach the middle, "memory" = "states", and since every DFA has only a finite number of states, you can force it to "forget" something, i.e., force it into the same state on two different strings. Then a "cut and paste" argument shows that you can replace one string with the other to form another accepted string, proving that $M$ accepts something it shouldn't.

Version 1 (slides 11-12): pick length so there are more such strings than states in $M$. Version 2 (slides 13-14): pick increasingly long strings of a simple form until the same thing happens. This argument is a little more subtle, since the string length, hence middle, changes when you do the cut-and-paste, and so you have to argue that where ever the middle falls, left half $\neq$ right half. Some cleverness in picking "long strings of a simple form" makes this possible; in this case the "$b$" in "ab" is a handy marker.

Version 3 (slide 15): Generalizing version 2, accepted strings longer than $p$ always forces $M$ around a loop. Substring defining the loop can be removed or repeated indefinitely, generating many simple variants of the initial string. Carefully choosing the initial string, you can often prove that some variants should be rejected. Again, there is some subtlety in these proofs to allow for any start point/length for the loop.

Not all proofs of non-regularity are about "left half/right half", of course, so the above isn't the whole story, but variations on these themes are widely used. Version 3 is especially versatile, and is the heart of the "pumping lemma", (next few slides).
Those who cannot remember the past are condemned to repeat it.

-- George Santayana (1905) Life of Reason

**Corollary**

Every sufficiently long input string forces a DFA around a loop.

**Proof**

Let \( p = |Q| \) and \( |w| \geq p \).

Let \( s \) be state \( M \) in after reading \( 1 \) of \( w \).

By pumping principle, \( \exists \gamma \) such that \( \gamma \in Q \).

---

**The Pumping Lemma**

For all regular languages \( L \), there is an integer \( p > 0 \) such that any string \( w \in L \) with \( |w| \geq p \) may be split into three substrings \( x, y, z \in \Sigma^* \) so that

1. \( w = xyz \)
2. \( y \neq \epsilon \)
3. \( |xy| \leq p \), and
4. \( \forall i \geq 0, xy^iz \in L \)
The Pumping Lemma

For all regular languages $L$, there is an integer $p > 0$ such that any string $w \in L$ with $|w| \geq p$ may be split into three substrings $x, y, z \in \Sigma^*$ so that

1. $w = xyz$
2. $y \neq \epsilon$
3. $|xy| \leq p$, and
4. $\forall i \geq 0, xy^iz \in L$

Proof:

$L$ is regular, so $\exists$ a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(M)$. Let $p = |Q|$. Let $w$ be any string in $L$. If $|w| < p$, the conclusion holds, vacuously. If $|w| \geq p$, let $q_0 = r_0, r_1, \ldots, r_p$ be the sequence of states entered by $M$ after reading the first $0, 1, \ldots, p$ letters of $w$. There are $p + 1$ entries in that list, but only $p$ states. So, by the Pigeon Hole Principle, $\exists i < j$ such that $r_i = r_j$. Let $x$ be the 1st $i$ letters of $w$, $y$ be letters $i + 1$ through $j$, inclusive, and let $z$ be the rest. Since $M$ accepts $w = xyz$ passing through $q$ both immediately before and immediately after $y$, it also accepts $xz, xyyz, xyyyz, \ldots$ using the $q-q-q$ loop $0, 2, 3, \ldots$ times, resp.

Key Idea: perfect squares become increasingly sparse, but PL $\Rightarrow$ at most $p$ gap between members
Idea: Pick big enough square so that gap to next is larger than the short piece the PL repeats.

$L = \{ a^n b^n | n \geq 0 \}$

Suppose $L$ is regular. By PL,

$\exists p \text{ such that } w = a^p \text{ by PL.}
\exists xyz \text{ such that } w = xyz
\quad \alpha 1y1 \leq p
xy^2z = a^p + 1y1
(p+1)^2 = p^2 + 2p + 1
p^2 + y1 \leq p^2 + 2p + 1
\therefore xy^2z \notin L$

Recall

So, by closure of regular languages under intersection, $L$ cannot be regular.

Of course, direct proof via Pumping Lemma is possible. E.g., a lot like the one for $\{a^n b^n | n \geq 0\}$. Alt way:

$L = \{ w \mid \#a(w) = \#b(w) \}$

$L \cap a^* b^* = \{ a^n b^n | n \geq 3 \}$

regular ?, regular, not regular

So, by closure of regular languages under intersection, $L$ cannot be regular.
C – the programming language – satisfies the pumping lemma, but is non-regular

```c
main(){return (((0))));}
```

If C were regular, \( \exists p \forall C \text{ programs } \exists x,y,z, \ldots \)
e.g., \( x = \varepsilon,y = \text{“m”} \): pumps nicely, giving new func names

But C is not regular

\[
L = C \cap L(\text{main() \{return(*0)*;\}})
\]

\( L \) is not regular: \( \exists p \ldots \)

Let \( w = \text{main() \{return(*0)*;\}} \)
then if \( y \in (\varepsilon,i \neq 1 \text{ gives unbalanced parens} \)
\( y \notin (\varepsilon,i \neq 1 \text{ gives an invalid prefix} \)

**Similar results possible for C++, Java, Python,**

---

**Some Algorithm Qs**

Given a string \( x \) and a regular language \( L \), how hard is it to decide these questions?

<table>
<thead>
<tr>
<th></th>
<th>( x \in L )</th>
<th>( L = \emptyset )</th>
<th>( L = \Sigma^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFA</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>NFA</td>
<td>(exercise)</td>
<td>( O(n) )</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>RegExp</td>
<td>(exercise)</td>
<td>(exercise)</td>
<td>( O(2^n) )</td>
</tr>
<tr>
<td>Java Prog</td>
<td>Undecidable – think “halting problem”</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended RegExp (( \neg ))</td>
<td>time at least ( 2^2^{\frac{2}{h}} ), where ( h &gt; \log n )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Some Algorithm Sketches

DFA/x ∈ L: read in DFA, simulate it step by step
DFA/L=∅: read DFA, build graph structure; depth-first-search to see if F is reachable from q₀; accept if not.
DFA/L=Σ*: apply DFA complement constr; do above
NFA/L=∅: like DFA/L=∅:
NFA or regexp/L=Σ*: not like DFA case;
do reexp → NFA, NFA → DFA via subset constr

“Extended” Regular Exprs

Regular languages are closed under ops other than ∪, •, *, e.g., ∩, complement, and DROP-OUT. We could add them to regexp syntax and still get only regular languages. E.g.:

aa*\((\sim((a ∪ b))^(aaa ∪ bbb) (a ∪ b)^*))\)

denotes the strings starting with 2 a’s, followed by a string not containing 3 adjacent a’s or b’s. (I think you did something like that in a homework, and it’s kind of a nuisance with plain regexp.)

Why don’t standard RegExp packages support this? The added code is minor: just the closure-under-complement construction.

But the run-time cost is ...

How much can we compute?

Visualize a fast, small computer, say:

One petaflop (10^15 ops sec^-1)
Femtometer (10^-15) in diameter (~ size of a neutron)

Buy a few: say, enough to pack the visible universe:

Radius of visible universe:
10^20 light years x π x 10^9 s/year x 3 x 10^8 m/s = 10^26 m

Volume: (10^24)^3 = 10^72 m^3

# processors: 10^72/(10^-15)^3 = 10^{123} (1 yotta-googles)

Let it run for a little while, say 10^10 years
10^10 yr x π x 10^7 s/yr x 10^15 ops/s x 10^{123} processors

= 10^{155} ops since the dawn of time
(somewhat optimistically)

Towers of twos

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>= 2</td>
</tr>
<tr>
<td>2^2</td>
<td>= 4</td>
</tr>
<tr>
<td>2^2^2</td>
<td>= 2^4 = 16</td>
</tr>
<tr>
<td>2^2^2^2</td>
<td>= 2^{16} = 65536</td>
</tr>
<tr>
<td>2^2^2^2^2</td>
<td>≈ 10^{19728}</td>
</tr>
</tbody>
</table>

Summary

There are (many) non-regular languages
Famous examples: {a^n b^n | n>0}, {#a = #b}, {ww}, {C}, {Java}
Famous ways to prove:
Diagonalization
M in same state on 2 strings it should distinguish
One stylized way: Pumping Lemma
Closure Properties
Simple algorithmic problems can be astronomically slow