FR2-0: **Halting Problem**

- Halting Machine takes as input an encoding of a Turing Machine $e(M)$ and an encoding of an input string $e(w)$, and returns "yes" if M halts on w, and "no" if M does not halt on w.
- Like writing a Java program that parses a Java function, and determines if that function halts on a specific input

FR2-1: **Halting Problem**

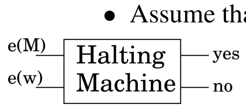
- Halting Machine takes as input an encoding of a Turing Machine $e(M)$ and an encoding of an input string $e(w)$, and returns "yes" if M halts on w, and "no" if M does not halt on w.
- Like writing a Java program that parses a Java function, and determines if that function halts on a specific input
- How might the Java version work?
  - Check for loops
  - `while (<test>) <body>`
    Use program verification techniques to see if test can ever be false, etc.

FR2-2: **Halting Problem**

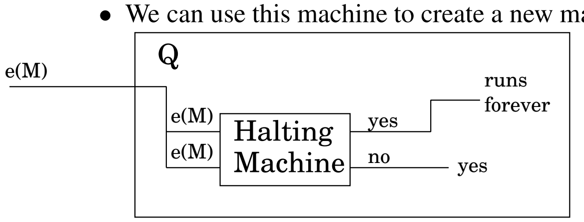
- The Halting Problem is *Undecidable*
  - There exists no Turing Machine that decides it
  - There is no Turing Machine that halts on all inputs, and always says "yes" if M halts on w, and always says "no" if M does not halt on w
- Prove Halting Problem is Undecidable by Contradiction:

FR2-3: **Halting Problem**

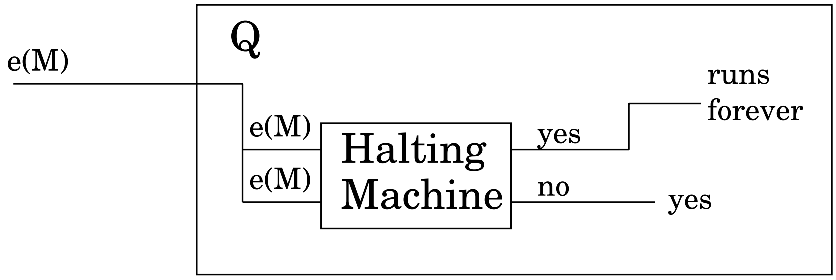
- Prove Halting Problem is Undecidable by Contradiction:
  - Assume that there is some Turing Machine that solves the halting problem.

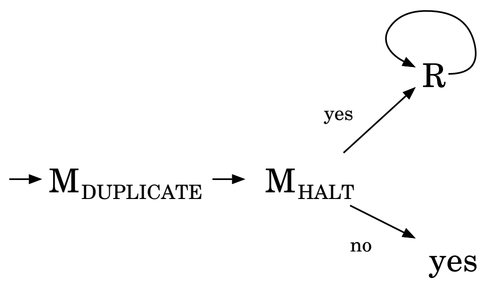


  - We can use this machine to create a new machine Q:



FR2-4: **Halting Problem**





FR2-5: **Halting Problem**

- Machine Q takes as input a Turing Machine M, and either halts, or runs forever.
- What happens if we run Q on $e(Q)$?
  - If M_{HALT} says Q should run forever on $e(Q)$, Q halts
  - If M_{HALT} says Q should halt on $e(Q)$, Q runs forever
- Q must not exist – but Q is easy to build if M_{HALT} exists, so M_{HALT} must not exist

FR2-6: **Halting Problem (Java)**

- Quick sideline: Prove that there can be no Java program that takes as input two strings, one containig source code for a Java program, and one containing an input, and determines if that program will halt when run on the given input.

```
boolean Halts(String SourceCode, String Input);
```

FR2-7: **Halting Problem (Java)**

```
boolean Halts(String SourceCode, String Input);

void Contrarian(String SourceCode) {
   if (Halts(SourceCode, SourceCode))
       while (true);
   else
       return;
}
```

FR2-8: **Halting Problem (Java)**

```
boolean Halts(String SourceCode, String Input);

void Contrarian(String SourceCode) {
   if (Halts(SourceCode, SourceCode))
       while (true);
   else
       return;
}
Contrarian("void Contrarian(String SourceCode { \
               if (Halts(SourceCode, SourceCode)) \
                ...
            } ");
```

What happens?

FR2-9: **Undecidable**

- Once we have one undecidable problem, it is (easier) to find more
- Use a reduction

FR2-10: **Reduction**

- Reduce Problem A to Problem B
  - Convert instance of Problem A to an instance of Problem B
    - Problem A: Power – x^y
    - Problem B: Multiplication – $x * y$
  - If we can solve Problem B, we can solve Problem A
  - If we can multiply two numbers, we can calculate the power x^y

FR2-11: **Reduction**

- If we can reduce Problem A to Problem B, and
- Problem A is undecidable, then:
- Problem B must also be undecidable
  - Because, if we could solve B, we could solve A

FR2-12: **Reduction**

- To prove a problem B is undecidable:
  - Start with a an instance of a known undecidable problem (like the Halting Problem)
  - Create an instance of Problem B, such that the answer to the instance of Problem B gives the answer to the undecidable problem
  - If we could solve Problem B, we could solve the halting problem . . .
  - . . . thus Problem B must be undecidable

FR2-13: **Reduction**

- Professor Shadey has given a reduction from a problem P_{new} to the Halting Problem

- Given any instance of P_{new}:
  - Create an instance of the halting problem
  - Use the solution to the halting problem to find a solution for P_{new}
- What has Professor Shadey shown?

FR2-14: **Reduction**

- Professor Shadey has given a reduction from a problem P_{new} to the Halting Problem
  - Given any instance of P_{new}:
    - Create an instance of the halting problem
    - Use the solution to the halting problem to find a solution for P_{new}
- What has Professor Shadey shown? NOTHING!

FR2-15: **More Reductions ...**

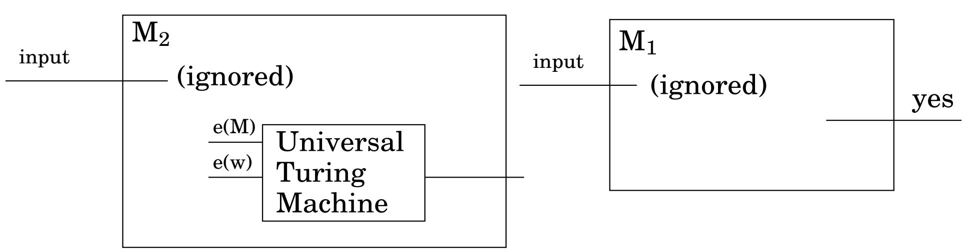
- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?

FR2-16: **More Reductions ...**

- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?
  - Start with an instance M, w of the halting problem
  - Create M_1, which accepts everything
  - Create M_2, which ignores its input, and runs M, w through the Universal Turing Machine. Accept if M halts on w.
- If M halts on w, then $L[M_2] = \Sigma^*$, and $L[M_1] = L[M_2]$
- If M does not halt on w, then $L[M_2] = \{\}$, and $L[M_1] \neq L[M_2]$

FR2-17: **More Reductions ...**

- Given two Turing Machines M_1, M_2, is $L[M_1] = L[M_2]$?



FR2-18: **More Reductions ...**

- If we had a machine M_{same} that took as input the encoding of two machines M_1 and M_2, and determined if $L[M_1] = L[M_2]$, we could solve the halting problem for any pair M, w:
  - Create a Machine that accepts everything (easy!). Encode this machine.
  - Create a Machine that first erases its input, then writes $e(M), e(w)$ on input, then runs Universal TM. Encode this machine

- Feed encoded machines into M_{same}. If M_{same} says "yes", then M halts on w, otherwise M does not halt on w

FR2-19: **Rice's Theorem**

- Determining if the language accepted by a Turing machine has any non-trivial property is undecidable
- "Non-Trivial" property means:
  - At least one recursively enumerable language has the property
  - Not all recursively enumerable languages have the property
- Example: Is the language accepted by a Turing Machine M regular?

FR2-20: **Rice's Theorem**

- Problem: Is the language defined by the Turing Machine M recursively enumerable?
  - Is this problem decidable?

FR2-21: **Rice's Theorem**

- Problem: Is the language defined by the Turing Machine M recursively enumerable?
  - Is this problem decidable? YES!
- All recursively enumerable languages are recursively enumerable.
- The question is "trivial"

FR2-22: **Rice's Theorem**

- Problem: Does the Turing Machine M accept the string w in k computational steps?
  - Is this problem decidable?

FR2-23: **Rice's Theorem**

- Problem: Does the Turing Machine M accept the string w in k computational steps?
  - Is this problem decidable? YES!
  - Problem is not language related – we're not asking a question about the language that is accepted, but about the language that is accepted *within a certain number of steps*

FR2-24: **Rice's Theorem – Proof**

- We will prove Rice's theorem by showing that, for any non-trivial property P, we can reduce the halting problem to the problem of determining if the language accepted by a Turing Machine has Property P.
- Given any Machine M, string w, and non-trivial property P, we will create a new machine M', such that either
  - $L[M']$ has property P if and only if M halts on w
  - $L[M']$ has property P if and only if M does not halt on w

FR2-25: **Rice's Theorem – Proof**

- Let P be some non-trivial property of a language.
- Two cases:
  - The empty language $\{\}$ has the property
  - The empty language $\{\}$ does not have the property

FR2-26: **Rice's Theorem – Proof**

- Properties that the empty language has:
  - Regular Languages
  - Languages that do not contain the string "aab"
  - Languages that are finite
- Properties that the empty language does not have:
  - Languages containing the string "aab"
  - Languages containing at least one string
  - Languages that are infinite

FR2-27: **Rice's Theorem – Proof**

- Let M be any Turing Machine, w be any input string, and P be any non-trivial property of a language, such that $\{\}$ has property P.
- Let L_{NP} be some recursively enumerable language that does *not* have the property P, and let M_{NP} be a Turing Machine such that $L[M_{NP}] = L_{NP}$
- We will create a machine M' such that M' has property P if and only if M does not halt on w.

FR2-28: **Rice's Theorem – Proof**

- M':
  - Save input
  - Erase input, simulate running M on w
  - Restore input
  - Simulates running M_{NP} on input

FR2-29: **Rice's Theorem – Proof**

- M':
  - Save input
  - Erase input, simulate running M on w
  - Restore input
  - Simulates running M_{NP} on input
- If M halts on w, $L[M'] = L_{NP}$, and $L[M']$ does not have property P
- If M does not halt on w, $L[M'] = \{\}$, and $L[M']$ does have property P

FR2-30: **Rice's Theorem – Proof**

- Let M be any Turing Machine, w be any input string, and P be any non-trivial property of a language, such that $\{\}$ *does not* have property P.
- Let L_{NP} be some recursively enumerable language that *does* have the property P, and let M_P be a Turing Machine such that $L[M_P] = L_P$
- We will create a machine M' such that M' has property P if and only if M does halt on w.

FR2-31: **Rice's Theorem – Proof**

- M':
  - Save input
  - Erase input, simulate running M on w
  - Restore input
  - Simulates running M_P on input

FR2-32: **Rice's Theorem – Proof**

- M':
  - Save input
  - Erase input, simulate running M on w
  - Restore input
  - Simulates running M_P on input
- If M halts on w, $L[M'] = L_P$, and $L[M']$ does have property P
- If M does not halt on w, $L[M'] = \{\}$, and $L[M']$ does not have property P

FR2-33: **Language Class P**

- A language L is polynomially decidable if there exists a polynomially bound Turing machine that decides it.
- A Turing Machine M is polynomially bound if:
  - There exists some polynomial function $p(n)$
  - For any input string w, M always halts within $p(|w|)$ steps
- The set of languages that are polynomially decidable is $\mathbf{P}$

FR2-34: **Language Class NP**

- A language L is non-deterministically polynomially decidable if there exists a polynomially bound non-deterministic Turing machine that decides it.
- A Non-Deterministic Turing Machine M is polynomially bound if:
  - There exists some polynomial function $p(n)$
  - For any input string w, M always halts within $p(|w|)$ steps, for all computational paths
- The set of languages that are non-deterministically polynomially decidable is $\mathbf{NP}$

FR2-35: **Language Class $\mathbf{NP}$**

- If a Language L is in $\mathbf{NP}$:
  - There exists a non-deterministic Turing machine M
  - M halts within $p(|w|)$ steps for all inputs w, in all computational paths
  - If $w \in L$, then there is at least one computational path for w that accepts (and potentially several that reject)
  - If $w \notin L$, then all computational paths for w reject

FR2-36: **$\mathbf{NP}$ vs $\mathbf{P}$**

- A problem is in $\mathbf{P}$ if we can *generate* a solution quickly (that is, in polynomial time
- A problem is in $\mathbf{NP}$ if we can *check* to see if a potential solution is correct quickly
  - Non-deterministically create (guess) a potential solution
  - Check to see that the solution is correct

FR2-37: **$\mathbf{NP}$ vs $\mathbf{P}$**

- All problems in $\mathbf{P}$ are also in $\mathbf{NP}$
  - That is, $\mathbf{P} \subseteq \mathbf{NP}$
  - If you can generate correct solutions, you can check if a guessed solution is correct

FR2-38: **Reduction Redux**

- Given a problem instance P, if we can
  - Create an instance of a different problem P', in polynomial time, such that the solution to P' is the same as the solution to P
  - Solve the instance P' in polynomial time
- Then we can solve P in polynomial time

FR2-39: **$\mathbf{NP}$-Complete**

- A language L is $\mathbf{NP}$-Complete if:
  - L is in $\mathbf{NP}$
  - *If* we could decide L in polynomial time, then *all* $\mathbf{NP}$ languages could be decided in polynomial time
  - That is, we could reduce *any* $\mathbf{NP}$ problem to L in polynomial time

FR2-40: **$\mathbf{NP}$-Complete**

- How do you show a problem is $\mathbf{NP}$-Complete?
  - Given *any* polynomially-bound non-deterministic Turing machine M and string w:
    - Create an instance of the problem that has a solution if and only if M accepts w

FR2-41: **$\mathbf{NP}$-Complete**

- First $\mathbf{NP}$-Complete Problem: Satisfiability (SAT)
  - Given any (possibly non-deterministic) Turing Machine M, string w, and polynomial bound $p(n)$
    - Create a boolean formula f, such that f is satisfiable if and only of M accepts w

FR2-42: **More NP-Complete Problems**

- So, if we could solve Satisfiability in Polynomial Time, we could solve *any* $\mathbf{NP}$ problem in polynomial time
  - Including factoring large numbers ...
- Satisfiability is $\mathbf{NP}$-Complete
- There are many $\mathbf{NP}$-Complete problems
  - Prove $\mathbf{NP}$-Completeness using a reduction

FR2-43: **Proving NP-Complete**

- To prove that a problem P_{new} is NP-Complete
  - Start with an instance of a known NP-Complete problem NP
  - Use this instance of NP to create an instance of P_{new}, such that the solution of P_{new} gives us a solution to the instance of NP
  - If we could solve P_{new} in polynomial time, we could solve NP in polynomial time, hence P_{new} is NP-Complete

FR2-44: **Proving NP-Complete**

- What does it mean if I could reduce a new problem to a known NP-Complete problem?

FR2-45: **Proving NP-Complete**

- What does it mean if I could reduce a new problem to a known NP-Complete problem?
  - If I could solve the NP-Complete problem quickly, I could solve the new poblem quickly

FR2-46: **Proving NP-Complete**

- What does it mean if I could reduce a new problem to a known NP-Complete problem?
  - If I could solve the NP-Complete problem quickly, I could solve the new poblem quickly
  - But if I could solve the NP-Complete problem quickly, then I could solve **any** problem quickly

FR2-47: **Proving NP-Complete**

- What does it mean if I could reduce a new problem to a known NP-Complete problem?
  - If I could solve the NP-Complete problem quickly, I could solve the new poblem quickly
  - But if I could solve the NP-Complete problem quickly, then I could solve **any** problem quickly
  - Haven't learned anything

FR2-48: **Proving NP-Complete**

- To prove P_{new} is NP-Complete:

- Need to reduce a know NP-Complete problem to P_{new}
- **Not** the other way around
- Can be confusion the first (or second) time you see it

FR2-49: **NP-Complete Problems**

- Undirected Hamilton Cycle is NP-Complete
  - How would we show this?

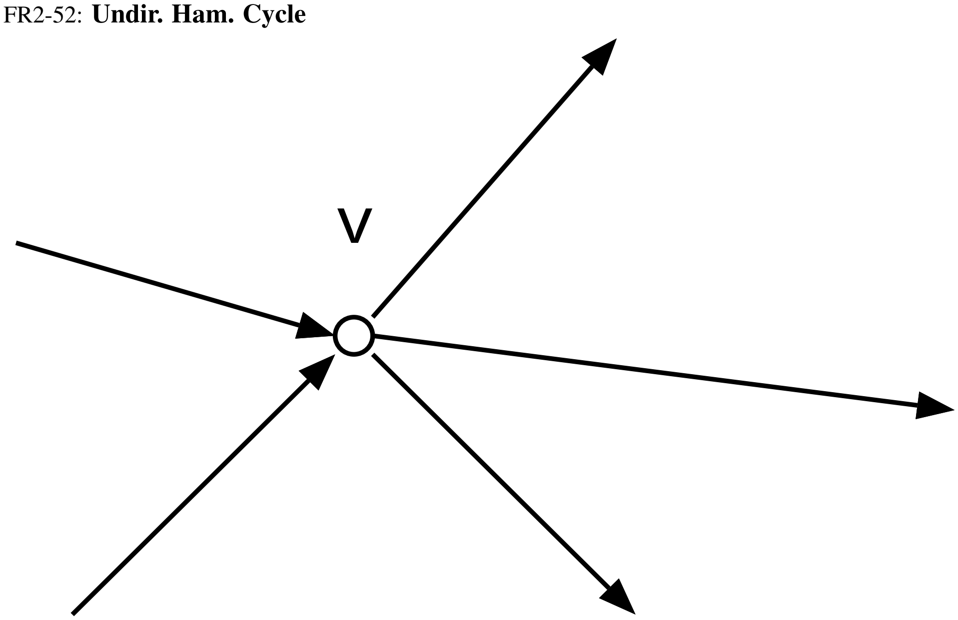
FR2-50: **NP-Complete Problems**

- Undirected Hamilton Cycle is NP-Complete
  - Start with a known NP-Complete problem
  - Reduce the NP-Complete problem to Undirected Hamilton Cycle
  - What would be a good choice, given what we've already proven NP-Complete in this class?

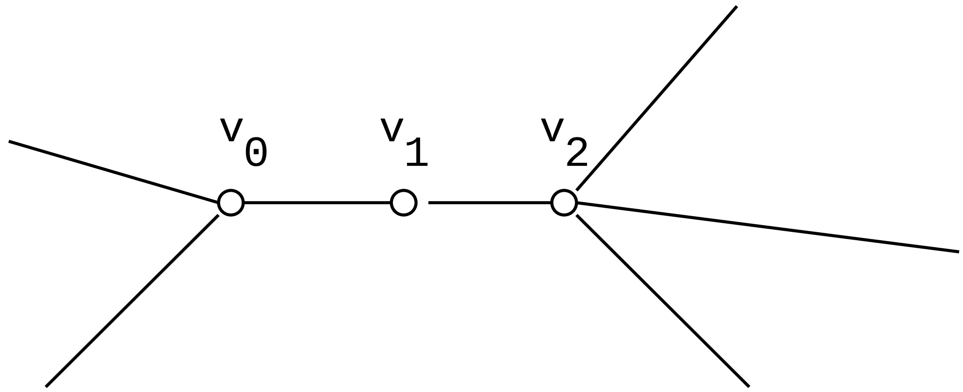
FR2-51: **NP-Complete Problems**

- Undirected Hamilton Cycle is NP-Complete
  - Reduction from Directed Hamilton Cycle
  - Given any instance of Directed Hamilton Cycle:
    - Create an insance of Undirected Hamilon Cycle
    - Show that the solution to Undirected Hamilton Cycle gives solution to Directed Hamilton Cycle

FR2-52: **Undir. Ham. Cycle**



FR2-53: **Undir. Ham. Cycle**


v0
v1
v2