17-0: **Tractable vs. Intractable**

- If a problem is *recursive*, then there exists a Turing machine that always halts, and solves it.
- However, a recursive problem may not be practically solvable
  - Problem that takes an exponential amount of time to solve is not practically solvable for large problem sizes
- Today, we will focus on problems that are practically solvable

17-1: **Language Class $\mathbf{P}$**

- 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 $\mathbf{P}$

17-2: **Language Class $\mathbf{P}$**

- $\mathbf{P}$ is the set of languages that can reasonably be decided by a computer
  - What about n^{100}, or $10^{100000}n^2$
    - Can these running times really be "reasonably" solvable
  - What about $n^{\log \log n}$
    - Not bound by any polynomial, but grows very slowly until n gets quite large

17-3: **Language Class $\mathbf{P}$**

- $\mathbf{P}$ is the set of languages/problems that can reasonably be solved by a computer
  - What about n^{100}, or $10^{100000}n^2$
  - Problems that have these kinds of running times are quite rare
  - Even a huge polynomial has a chance at being solvable for large problems if you throw enough machines at it – unlike exponential problems, where there is pretty much no hope for solving large problems

17-4: **Reachability**

- Given a Graph G, and two vertices x and y, is there a path from x to y in G?
- Note that this is a *Problem* and not a *Language*, though we can easily convert it into a language as follows:
- $L_{reachable} = \{w : w = en(g)en(x)en(y)$, there is a path from x to y in $G\}$
  - Can encode G:
    - Numbering all of the vertices
    - Give an adjacency matrix, using binary encoding of each vertex

17-5: **Reachability**

- Let $A[]$ be the adjacency matrix
    - $A[i, j] = 1$ if link from v_i to v_j

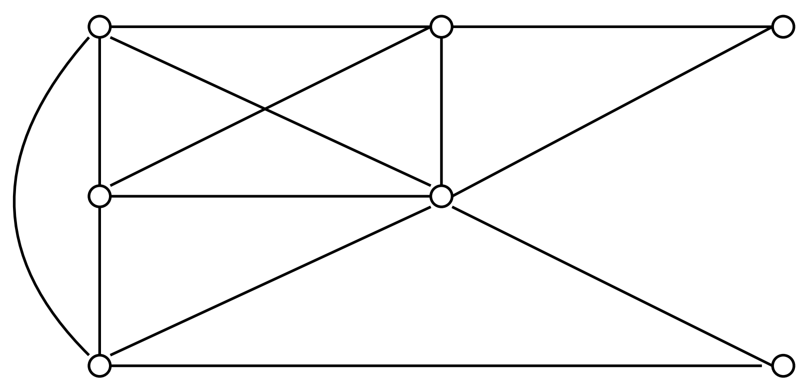
```
for (i=0; i<|V|; i++) {
  A[i,i] = 1;
  for (i=0; i < |V|; i++)
    for (j=0; j < |V|; j++)
      for (k=0; k < |V|; k++)
        if (A[i,j] && A[j,k])
          A[i,k] = 1;
}
```

17-6: **Java/C vs. Turing Machine**

- But wait ... that's Java/C code, not a Turing Machine!
- If a C program can execute in n steps, then we can simulate the C program with a Turing Machine that takes at most $p(n)$ steps, for some polynomial function p.
- We will use Java/C style pseudo-code for many of the following problems

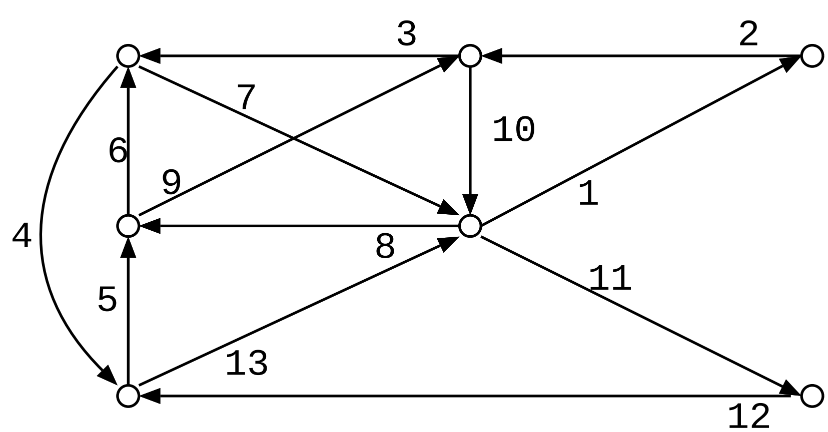
17-7: **Euler Cycles**

- Given an undirected graph G, is there a cycle that traverses every edge exactly once?



17-8: **Euler Cycles**

- Given an undirected graph G, is there a cycle that traverses every edge exactly once?



17-9: **Euler Cycles**

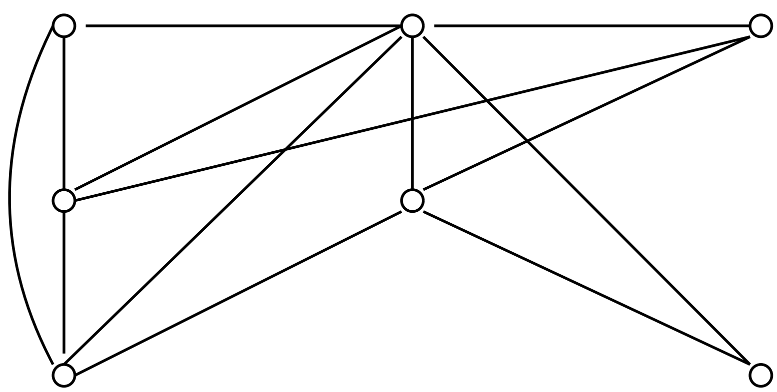
- We can determine if a graph G has an Euler cycle in polynomial time.
- A graph G has an Euler cycle if and only if:
  - G is connected
  - All vertices in G have an even # of adjacent edges

17-10: **Euler Cycles**

- Pick any vertex, start following edges (only following an edge once) until you reach a "dead end" (no untraversed edges from the current node).
- Must be back at the node you started with
  - Why?
- Pick a new node with untraversed edges, create a new cycle, and splice it in
- Repeat until all edges have been traversed

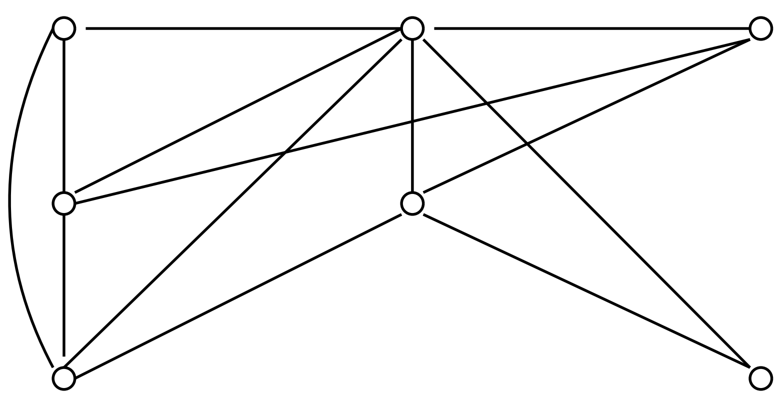
17-11: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?



17-12: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?

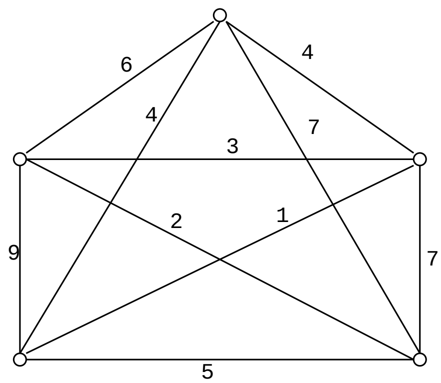


17-13: **Hamiltonian Cycles**

- Given an undirected graph G, is there a cycle that visits every vertex exactly once?
    - Very similar to the Euler Cycle problem
    - No known polynomial-time solution

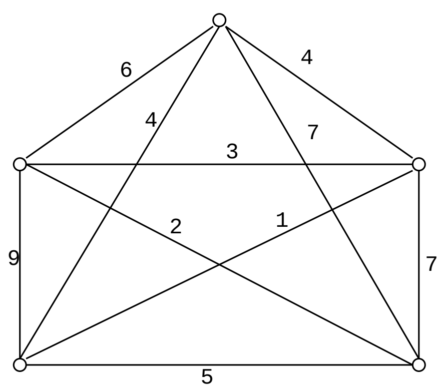
17-14: **Traveling Salesman**

- Given an undirected, completely connected graph G with weighted edges, what is the minimal length circuit that connects all of the vertices?



17-15: **Traveling Salesman**

- Given an undirected, completely connected graph G with weighted edges, what is the minimal length circuit that connects all of the vertices?



`Path Cost:18`

17-16: **Decision vs. Optimization**

- A *Decision Problem* has a yes/no answer
    - Is there a path from vertex i to vertex j in graph G?
    - Is there an Euler cycle in graph G?
    - Is there a Hamiltonian cycle in graph G?
- An *Optimization Problem* tries to find an optimal solution, from a choice of several potential solutions
    - What is the cheapest cycle in a weigted graph?

17-17: **Decision vs. Optimization**

- Given an undirected, completely connected graph G with weighted edges, what is the minimal length circuit that connects all of the vertices?
  - This is an *optimization* problem, and not a *decision* problem
  - We can easily convert it into a decision problem:
    - Given a weighted, undirected graph G, is there a cycle with cost no greater than k?

17-18: **Decision vs. Optimization**

- For every optimization problem
  - Find the lowest cost solution to a problem
- We can create a similar decision problem
  - Is there a solution under cost k?

17-19: **Decision vs. Optimization**

- If we can solve the "optimization" version of a problem in polynomial time, we can solve the "decision" version of the same problem in polynomial time.
  - Find the optimal solution, check to see if it is under the limit
- If we can solve the "decision" version of the problem, we can solve the "optimization" version of the same problem
  - Modified binary search

17-20: **Integer Partition**

- Set S of non-negative numbers $\{a_1 \ldots a_n\}$
- Is there a set $P \subseteq \{1, 2, \ldots n\}$ such that

$$\sum_{i \in P} a_i = \sum_{i \notin P} a_i$$

- Can we partition the set into two subsets, each of which has the same sum?

17-21: **Integer Partition**

- $S = \{3, 5, 7, 10, 15, 20\}$
- Can break S into:
  - $\{3, 5, 7, 15\}$
  - $\{10, 20\}$

17-22: **Integer Partition**

- $S = \{1, 4, 9, 10, 15, 27\}$
- No valid partiton
  - Sum of all numbers is 66
  - Each partition needs to sum to 34 (why?)

  - No subset of S sums to 34

17-23: **Solving Integer Partition**

- H = sum of all integers in S divided by 2
- $B(i) = \{b \leq H : b$ is the sum of some subset of $a_1 \ldots a_i\}$
  - $a_1 = 5, a_2 = 20, a_3 = 17, a_4 = 30, H = 36$
  - $B(0) = \{0\}$
  - $B(1) = \{0, 5\}$
  - $B(2) = \{0, 5, 20, 25\}$
  - $B(3) = \{0, 5, 17, 20, 22, 25\}$
  - $B(4) = \{0, 5, 17, 20, 22, 25, 30, 35\}$
- Partition iff $H \in B(n)$

17-24: **Solving Integer Partition**

- Computing $B(n)$ (inefficient):

```
B(0) = {0}
for (i = 1; i <= n; i++)
    B(i) = B(i − 1)                    (copy)
    for (j = i; j < H; j++)
        if (j − a_i) ∈ B(i − 1)
            add j to B(i)
```

(How might we make this more efficient?) 17-25: **Solving Integer Partition**

- Computing $B(n)$ (inefficient):

```
B(0) = {0}
for (i = 1; i <= n; i++)
    B(i) = B(i − 1)                    (copy)
    for (j = i; j < H; j++)
        if (j − a_i) ∈ B(i − 1)
            add j to B(i)
```

Running time: $O(nH)$. Polynomial? 17-26: **Solving Integer Partition**

- Running time: $O(nH)$.
- *Not* polynomial.
  - n integers of size $\approx 2^n$
    - n integers, each of which has $\approx n$ digits
  - $H \approx \frac{n}{2}2^n$
  - Length of input n^2

- Not the most efficient algorithm to solve the problem
- All known solutions require exponential time, however

17-27: **Unary Integer Partition**

- Given a set S of non-negative numbers $\{a_1 \ldots a_n\}$, *encoded in unary*
- Is there a set $P \subseteq \{1, 2, \ldots n\}$ such that
$$\sum_{i \in P} a_i = \sum_{i \notin P} a_i$$
- This problem can be solved in Polynomial time
- In fact, the previous algorithm will solve the problem in polynomial time!
  - How can this be?

17-28: **Unary Integer Partition**

- Given a set S of non-negative numbers $\{a_1 \ldots a_n\}$, *encoded in unary*
- Is there a set $P \subseteq \{1, 2, \ldots n\}$ such that
$$\sum_{i \in P} a_i = \sum_{i \notin P} a_i$$
- This problem can be solved in Polynomial time
  - We've made the problem description exponentially longer
  - In general, it doesn't matter how you encode a problem *as long as you don't use unary to encode numbers!*

17-29: **Satisfiability**

- A Boolean Formula in Conjunctive Normal Form (CNF) is a conjunction of disjunctions.
  - $(x_1 \vee x_2) \wedge (x_3 \vee \overline{x_2} \vee \overline{x_1}) \wedge (x_5)$
  - $(x_3 \vee x_1 \vee x_5) \wedge (x_1 \vee \overline{x_5} \vee \overline{x_3}) \wedge (x_5)$
- A Clause is a group of variables x_i (or negated variables $\overline{x_j}$) connected by ORs ($\vee$)
- A Formula is a group of clauses, connected by ANDs ($\wedge$)

17-30: **Satisfiability**

- Satisfiability Problem: Given a formula in Conjunctive Normal Form, is there a set of truth values for the variables in the formula which makes the formula true?
- $(x_1 \vee x_4) \wedge (\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
  $(\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
  - Satisfiable: x_1 = T, x_2 = F, x_3 = T, x_4 = F
- $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$
  - Not Satisfiable

17-31: **2-SAT**

- 2-SAT is a special case of the satisfiability problem, where each clause has no more than 2 variables.
- Both of the following problems are instances of 2-SAT
  - $(x_1 \vee x_4) \wedge (\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
    $(\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
  - $(x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$

17-32: **2-SAT**

- 2-SAT is in $\mathbf{P}$ – given an instance of 2-SAT, we can determine if the formula is satisfiable in polynomial time
- If a variable x_i is true:
  - Every clause that contains x_i is true.
  - For every clause of the form $(\overline{x_i} \vee x_j)$, variable x_j must be true.
  - For every clause of the form $(\overline{x_i} \vee \overline{x_j})$, variable x_j must be false.

17-33: **2-SAT**

- 2-SAT is in $\mathbf{P}$ – given an instance of 2-SAT, we can determine if the formula is satisfiable in polynomial time
- If a variable x_i is false:
  - Every clause that contains $\overline{x_i}$ is true.
  - For every clause of the form $(x_i \vee x_j)$, variable x_j must be true.
  - For every clause of the form $(x_i \vee \overline{x_j})$, variable x_j must be false.
- Once we know the truth value of a single variable, we can use this information to find the truth value of many other variables

17-34: **2-SAT**

- $(x_1 \vee x_4) \wedge (\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
  $(\overline{x_1} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
- If x_1 is true ...

17-35: **2-SAT**

- ~~$(x_1 \vee x_4)$~~$\wedge (\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
  $($~~$\overline{x_1}$~~$\vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
- If x_1 is true ...

17-36: **2-SAT**

- $(\overline{x_2} \vee x_4) \wedge (x_3 \vee x_2) \wedge$
  $(\overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_2 \vee \overline{x_4})$
- If x_1 is true
- Then x_4 must be false ...

17-37: **2-SAT**

- $(\overline{x_2} \vee$ ~~x_4~~$) \wedge (x_3 \vee x_2) \wedge$
  ~~$(\overline{x_4})$~~$\wedge(\overline{x_2} \vee \overline{x_3})\wedge$~~$(x_2 \vee \overline{x_4})$~~
- If x_1 is true
- Then x_4 must be false ...

17-38: **2-SAT**

- $(\overline{x_2}) \wedge (x_3 \vee x_2) \wedge$
  $(\overline{x_2} \vee \overline{x_3})$
- If x_1 is true
- Then x_4 must be false
- Then x_2 must be false ...

17-39: **2-SAT**

- ~~$(\overline{x_2})$~~ $\wedge(x_3 \vee$ ~~x_2~~$)\wedge$
  ~~$(\overline{x_2} \vee \overline{x_3})$~~
- If x_1 is true
- Then x_4 must be false
- Then x_2 must be false ...

17-40: **2-SAT**

- (x_3)
- If x_1 is true
- Then x_4 must be false
- Then x_2 must be false
- Then x_3 must be true ...

17-41: **2-SAT**

- ~~(x_3)~~
- If x_1 is true
- Then x_4 must be false
- Then x_2 must be false
- Then x_3 must be true
- And the formula is satisfiable

17-42: **Algorithm to solve 2-SAT**

- Pick any variable x_i. Set it to true
- Modify the formula, based on x_i being true:
  - Remove any clause that contains x_i
  - For any clause of the form $(\overline{x_i}, x_j)$, Variable x_j must be true. Recursively modify the formula based on x_j being true.
  - For any clause of the form $(\overline{x_i}, \overline{x_j})$, Variable x_j must be false. Recursively modify the formula based on x_j being false.

17-43: **Algorithm to solve 2-SAT**

- Pick any variable x_i. Set it to true
- Modify the formula, based on x_i being true:
- When you are done with the modification, one of 3 cases may occur:
  - All of the variables are set to some value, and the formula is thus satisfiable
  - Several of the clauses have been removed, leaving you with a smaller problem. Pick another variable and repeat
  - The choice of True for x_i leads to a contradiction: some variable x_j must be both true and false. In this case, restore the old formula, set x_i to false, and repeat

17-44: **Algorithm to solve 2-SAT**

- Example:
- $(x_1 \vee x_3) \wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge$
  $(\overline{x_1} \vee x_4) \wedge (x_1 \vee x_2)$
- First, we pick x_1, set it to true ...

17-45: **Algorithm to solve 2-SAT**

- Example:
- ~~$(x_1 \vee x_3)$~~ $\wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge$
  $(\overline{x_1} \vee x_4) \wedge$ ~~$(x_1 \vee x_2)$~~
- First, we pick x_1, set it to true
- Which means than x_4 must be true ...

17-46: **Algorithm to solve 2-SAT**

- Example:
- ~~$(x_1 \vee x_3)$~~ $\wedge (\overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge$
  ~~$(\overline{x_1} \vee x_4)$~~ $\wedge$ ~~$(x_1 \vee x_2)$~~
- First, we pick x_1, set it to true
- Which means than x_4 must be true ...

- And we have a smaller problem.

17-47: **Algorithm to solve 2-SAT**

- Example:
- $(\overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$
- First, we pick x_1, set it to true
- Which means than x_4 must be true
- And we have a smaller problem.
- Next, pick x_2, set it to true ...

17-48: **Algorithm to solve 2-SAT**

- Example:
- $(\cancel{\overline{x_2}} \vee x_3) \wedge (\cancel{\overline{x_2}} \vee \overline{x_3})$
- First, we pick x_1, set it to true
- Which means than x_4 must be true
- And we have a smaller problem.
- Next, pick x_2, set it to true ...

17-49: **Algorithm to solve 2-SAT**

- Example:
- $(\cancel{\overline{x_2}} \vee x_3) \wedge (\cancel{\overline{x_2}} \vee \overline{x_3})$
- First, we pick x_1, set it to true
- Which means than x_4 must be true
- And we have a smaller problem.
- Next, pick x_2, set it to true
- and x_3 must be both true and false. Whoops!

17-50: **Algorithm to solve 2-SAT**

- Example:
- $(\overline{x_2} \vee x_3) \wedge (\overline{x_2} \vee \overline{x_3})$
- First, we pick x_1, set it to true
- Which means than x_4 must be true
- And we have a smaller problem.
- Next, pick x_2, set it to true

- and x_3 must be both true and false.
- Back up, set x_2 to false ...

17-51: **Algorithm to solve 2-SAT**

- Example:
- ~~$(\overline{x_2} \vee x_3)$~~ $\wedge$ ~~$(\overline{x_2} \vee \overline{x_3})$~~
- First, we pick x_1, set it to true
- Which means than x_4 must be true
- And we have a smaller problem.
- Next, pick x_2, set it to true
- and x_3 must be both true and false.
- Back up, set x_2 to false
- And all clauses are satisfied (value of x_3 doesn't matter)

17-52: **Algorithm to solve 2-SAT**

- Example:
- $(\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_3)$
- First, we pick x_1, and set it to true

17-53: **Algorithm to solve 2-SAT**

- Example:
- $(\cancel{\overline{x_1}} \vee x_2) \wedge (\cancel{\overline{x_1}} \vee \overline{x_2}) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4}) \wedge$ ~~$(x_1 \vee x_3)$~~
- First, we pick x_1, and set it to true
- And x_2 must be both true and false. Back up ...

17-54: **Algorithm to solve 2-SAT**

- Example:
- $(\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4}) \wedge (x_1 \vee x_3)$
- First, we pick x_1, and set it to true
- And x_2 must be both true and false. Back up
- And set x_1 to be false ...

17-55: **Algorithm to solve 2-SAT**

- Example:
- ~~$(\overline{x_1} \vee x_2)$~~ $\wedge$ ~~$(\overline{x_1} \vee \overline{x_2})$~~ $\wedge (\overline{x_3} \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4}) \wedge (\cancel{x_1} \vee x_3)$

- First, we pick x_1, and set it to true
- And x_2 must be both true and false. Back up
- And set x_1 to be false
- And x_3 must be true ...

17-56: **Algorithm to solve 2-SAT**

- Example:
- $\cancel{(\overline{x_1} \vee x_2)} \wedge \cancel{(\overline{x_1} \vee \overline{x_2})} \wedge (\cancel{\overline{x_3}} \vee x_4) \wedge (\cancel{\overline{x_3}} \vee \overline{x_4}) \wedge (\cancel{x_1} \vee x_3)$
- First, we pick x_1, and set it to true
- And x_2 must be both true and false. Back up
- And set x_1 to be false
- And x_3 must be true
- And x_4 must be both true and false. No solution

17-57: **Algorithm to solve 2-SAT**

- Once we've decided to set a variable to true or false, the "marking off" phase takes a polynomial number of steps
- Each variable will be chosen to be set to true no more than once, and chosen to be set to false no more than once more than once
- Total running time is polynomial

17-58: **3-SAT**

- 3-SAT is a special case of the satisfiability problem, where each clause has no more than 3 variables.
- 3-SAT has no known polynomial solution
  - Can't really do any better than trying all possible truth assignments to all variables, and see if they work.