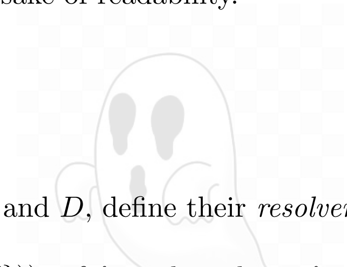
CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 17: Resolution and the Church-Turing Thesis

Harvard SEAS - Fall 2024 *Oct. 31, 2024*

1 Announcements

- Happy Halloween
- Pset7 out.
- Anurag's in person OH today Thur 11AM - 12 PM.
- These notes have been reorganized for the sake of readability.



2 Resolution

2.1 The Basic Resolution Algorithm

Definition 2.1 (resolution rule)**.** For clauses C and D, define their *resolvent* to be

$$C \diamond D = \begin{cases} \text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) & \text{if } \ell \text{ is a literal s.t. } \ell \in C \text{ and } \neg\ell \in D \\ 1 & \text{if there is no such literal } \ell \end{cases}$$

Here $C - \{\ell\}$ means remove literal ℓ from clause C, and 1 represents `true`. As noted last time, if C and D can be resolved with respect to more than one literal ℓ, then for all choices of ℓ we will have $\text{Simplify}((C - \{\ell\}) \vee (D - \{\neg\ell\})) = 1$, so $C \diamond D$ is well-defined.

In the special case where $C = \ell, D = \neg\ell$, we use our definition from Lecture 15 that empty clause is always false and obtain

$$(\ell) \diamond (\neg\ell) = \emptyset = \text{FALSE}.$$

From now on, it will be useful to view a CNF formula as just a set $\mathcal{C}$ of clauses.

Definition 2.2. Let $\mathcal{C}$ be a set of clauses over variables $x_0, \ldots, x_{n-1}$. We say that an assignment $\alpha \in \{0,1\}^n$ *satisfies* $\mathcal{C}$ if α satisfies all of the clauses in $\mathcal{C}$, or equivalently α satisfies the CNF formula

$$\varphi(x_0, \ldots, x_{n-1}) = \bigwedge_{C \in \mathcal{C}} C(x_0, \ldots, x_{n-1}).$$

The following theorem gives us a criteria to decide if a set of clauses is satisfiable. Note that resolution plays a crucial rule here.

Theorem 2.3 (Resolution Theorem)**.** *Let $\mathcal{C}$ be a set of clauses over n variables $x_0, \ldots, x_{n-1}$. Suppose that $\mathcal{C}$ is* closed under resolution*, meaning that for every $C, D \in \mathcal{C}$, we have $C \diamond D \in \mathcal{C} \cup \{1\}$. Then:*

1. $\emptyset \in \mathcal{C}$ iff $\mathcal{C}$ is unsatisfiable.

2. *If $\emptyset \notin \mathcal{C}$, then* `ExtractAssignment(`$\mathcal{C}$`)` *finds a satisfying assignment to $\mathcal{C}$ in time $O(n + k \cdot |\mathcal{C}|)$, where* `ExtractAssignment()` *is an algorithm described in Section 2.2.*

Above k is the maximum width over all clauses in $\mathcal{C}$.

(Note that the "$\cup\{1\}$" is a slight modification to the definition of closed under resolution from lecture and the previous version of these notes.)

The `ExtractAssignment(`$\mathcal{C}$`)` algorithm is described in Section 2.2. The algorithm for solving the CNF-Satisfiability is informally described as follows. We start with a set $\mathcal{C}$ of clauses from a CNF formula and keep adding resolvents until we cannot add any new ones. At this point, we know that the new set of clauses is closed under resolution. Then Theorem 2.3 tells us that if $\emptyset$ is a clause, then the formula is unsatisfiable. If $\emptyset$ is not in the set of clauses, then the `ExtractAssignment()` algorithm gives us a way to find a satisfying assignment.

A more formal version is as follows. We start with the set of clauses $C_0, C_1, \cdots, C_{m-1}$ that appear in the CNF φ, simplify all the clauses in φ and then:

1. Resolve C_0 with each of $C_1, \ldots, C_{m-1}$, adding any new clauses obtained from the resolution $C_m, C_{m+1}, \ldots$. If $\emptyset$ clause is found, return `unsatisfiable`.

2. Resolve C_1 with each of $C_2, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Step 1, again adding any new clauses. If $\emptyset$ clause is found, return `unsatisfiable`.

3. Resolve C_2 with each of $C_3, \ldots, C_{m-1}$ as well as with all of the resolvents obtained in Steps 1 and 2, again adding any new clauses. If $\emptyset$ clause is found, return `unsatisfiable`.

4. etc.

5. Run `ExtractAssignment()` on the set of all clauses and return the satisfying assignment.

In pseudo-code, the algorithm can be written as follows.

```
ResolutionInOrder($\varphi$)
Input        : A CNF formula $\varphi(x_0, \ldots, x_{n-1})$
Output       : Whether $\varphi$ is satisfiable or unsatisfiable
Let $C_0, C_1, \ldots, C_{m-1}$ be the clauses in $\varphi$, after simplifying each clause;
$i = 0$ ;                 /* clause to resolve with others in current iteration */
$f = m$ ;     /* start of 'frontier' - new resolvents from current iteration */
$g = m$ ;                                                   /* end of frontier */
while $f > i + 1$ do
    foreach $j = i + 1$ to $f - 1$ do
        $R = C_i \diamond C_j$;
        if $R = 0$ then return unsatisfiable;
        else if $R \notin \{C_0, C_1, \ldots, C_{g-1}\}$ then
            $C_g = R$;
            $g = g + 1$;
    $f = g$;
    $i = i + 1$
return ExtractAssignment(($C_0, C_1, \ldots C_{g-1}$))
```

Algorithm 1: A Resolution-based SAT Algorithm

Example: $\phi(x_0, x_1, x_2) = (\neg x_0 \vee x_1) \wedge (\neg x_1 \vee x_2) \wedge (x_0 \vee x_1 \vee x_2) \wedge (\neg x_2)$
We write out the clauses explicitly:

$$C_0 = (\neg x_0 \vee x_1)$$

$$C_1 = (\neg x_1 \vee x_2)$$

$$C_2 = (x_0 \vee x_1 \vee x_2)$$

$$C_3 = (\neg x_2)$$

We can then begin to resolve clauses:

$$C_4 = C_0 \diamond C_1 = (\neg x_0 \vee x_2)$$

$$C_5 = C_0 \diamond C_2 = (x_1 \vee x_2)$$

$$C_6 = C_0 \diamond C_3 = 1 \qquad \text{(since there is no common literal to resolve on)}$$

$$C_7 = C_1 \diamond C_2 = (x_0 \vee x_2)$$

$$C_8 = C_1 \diamond C_3 = (\neg x_1)$$

$$\xcancel{C_1 \diamond C_4 = 1} \qquad \text{(since we already have the clause 1)}$$

$$C_9 = C_1 \diamond C_5 = (x_2)$$

$$C_{10} = C_2 \diamond C_3 = (x_0 \vee x_1)$$

$$C_{11} = C_3 \diamond C_4 = (\neg x_0)$$

$$C_{12} = C_3 \diamond C_5 = (x_1)$$

$$C_{13} = C_3 \diamond C_7 = (x_0)$$

$$C_{14} = C_3 \diamond C_9 = 0$$

Therefore, $\phi(x_0, x_1, x_2)$ is `unsatisfiable`.

Example 2: $\psi(x_0, x_1, x_2, x_3) = (\neg x_0 \vee x_3) \wedge (x_0 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1) \wedge (\neg x_3)$
Note that the first four clauses correspond to the palindrome formula. When we apply resolution to the above formula, we derive $(\neg x_0) = (\neg x_0 \vee x_3) \diamond (\neg x_3)$, leaving us with the following set of clauses:

$$(\neg x_0 \vee x_3), (x_0 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1), (\neg x_3), (\neg x_0)$$

Then we get stuck and cannot derive any new clauses. Then the Resolution Algorithm says that ψ is `satisfiable`. The execution of `ExtractAssignment()` on this set of clauses is given in Section 2.2. .

2.2 Assignment Extraction

We begin by describing the `ExtractAssignment()` algorithm for finding a satisfying assignment to a set $\mathcal{C}$ of clauses that are closed under resolution and don't contain $\emptyset$. Specifically, we generate our satisfying assignment one variable v at a time in the following manner:

1. If $\mathcal{C}$ contains a singleton clause (v), then we assign $v = 1$.
2. If it contains $(\neg v)$ then assign $v = 0$.
3. If it contains neither (v) nor $(\neg v)$, then assign v arbitrarily.
4. $\mathcal{C}$ cannot contain both (v) and $(\neg v)$, because $\mathcal{C}$ is closed and does not contain 0.

Once we have assigned a variable to a value, we set that variable's value in every clause and simplify.

Formally, the algorithm works as follows:

```
1 ExtractAssignment($\mathcal{C}$)
  Input   : A closed and simplified set $\mathcal{C}$ of clauses over variables $x_0, \ldots, x_{n-1}$ such
            that $0 \notin \mathcal{C}$
  Output  : An assignment $\alpha \in \{0,1\}^n$ that satisfies all of the clauses in $\mathcal{C}$
2 foreach $i = 0, \ldots, n-1$ do
3     if $(x_i) \in \mathcal{C}$ then $\alpha_i = 1$;
4     else $\alpha_i = 0$;
5     $\mathcal{C} = \mathcal{C}|_{x_i=\alpha_i}$, meaning that we set $x_i = \alpha_i$ and then simplify all clauses (see Sec. 2.6)
6 return $\alpha$
```

Algorithm 2: Assignment extraction algorithm

Example of assignment extraction: Consider applying Algorithm 2 to the set of clauses derived from the formula in the Example 2 above:

$$(\neg x_0 \vee x_3), (x_0 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1), (\neg x_3), (\neg x_0)$$

Going through the variables in order, we set $x_0 = 0$ because we are forced to by the clause $(\neg x_0)$. After that, the clauses become:

$$(\neg 0 \vee x_3), (0 \vee \neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1), (\neg x_3), (\neg 0)$$

which simplifies to

$$(\neg x_3), (\neg x_1 \vee x_2), (\neg x_2 \vee x_1).$$

These clauses don't include (x_1) or $(\neg x_1)$, so we can set x_1 as either 0 or 1. Arbitrarily choosing $x_1 = 1$, the clauses become:

$$(\neg x_3), (\neg 1 \vee x_2), (\neg x_2 \vee 1), (\neg x_3),$$

which simplifies to

$$(\neg x_3), (x_2).$$

Then we set $x_2 = 1$, and finally set $x_3 = 0$, yielding the satisfying assignment $(0, 1, 1, 0)$.

2.3 Proof Sketches of Runtime and Correctness

Proof Sketch of Theorem 2.3. First, suppose $\mathcal{C}$ is such that $\emptyset \in \mathcal{C}$. Since $\emptyset$ is the always-false clause, no assignment can satisfy it. Hence $\mathcal{C}$ is unsatisfiable.

Conversely, if $\emptyset \notin \mathcal{C}$, then Item 2 says that `ExtractAssignment()` will find a satsifying assignment, so $\mathcal{C}$ is satisfiable. Thus we turn to sketching the proof of Item 2.

The key point is to show that after assigning a variable x_i as in `ExtractAssignment()`, set of clauses $\mathcal{C}|_{x_i=\alpha_i}$ (a) does not contain $\emptyset$, and (b) remains closed. (a) holds because of how we set v. Intuitively, (b) holds because assigning v and then resolving two resulting clauses C' and D' is equivalent to first resolving the original clauses C and D and then assigning v. We know that $C \diamond D \in \mathcal{C}$ by closure of $\mathcal{C}$, so we have $C' \diamond D'$ after assigning v. □

Now, we consider the runtime and correctness of the resolution algorithm. Let $\mathcal{C}_{\mathit{fin}}$ be the final set of clauses produced in Algorithm 1. Let k_{fin} be the maximum *width* (number of literals) among the clauses in $\mathcal{C}_{\mathit{fin}}$.

Runtime: Before analysing the runtime of the resolution algorithm, lets understand why the resolution algorithm terminates. The resolution algorithm always terminates because there are only finitely many clauses that can be generated on n variables, namely at most $3^n + 1$. (The base is 3 since for each variable we can either include it, include its negation, or not include it at all.) The $+1$ accounts for the "clause" 1.

We can now give a finer estimate of the runtime of the resolution algorithm. The algorithm performs $O(|\mathcal{C}_{\mathit{fin}}|^2)$ resolutions and the runtime of each resolution step is $O(k_{\mathit{fin}})$. By Theorem 2.3, the `ExtractAssignment()` takes time $O(n + k_{\mathit{fin}} \cdot |\mathcal{C}_{\mathit{fin}}|)$. Thus, the overall runtime is $O(n + k_{\mathit{fin}} \cdot |\mathcal{C}_{\mathit{fin}}|^2)$.

Remark: Using $k_{\mathit{fin}} \leq n$ and $|\mathcal{C}_{\mathit{fin}}| \leq 3^n$, we have a worst-case runtime $O(n \cdot 9^n)$, which is worse than exhaustive search over the 2^n satisfying assignments. However, there are cases where the estimate on $\mathcal{C}_{\mathit{fin}}$ can be significantly improved; an example is discussed in Section 2.4. SAT Solvers in practice only run the resolution partially; see Section 2.5.

Correctness: We first argue that if the resolution algorithm finishes all the resolution steps, then $\mathcal{C}_{\mathit{fin}}$

Lemma 2.4. *Consider an execution of Algorithm 1 that reaches Line 15. Then $\mathcal{C}_{\mathit{fin}}$ is closed under resolution.*

Proof. We show this by contradiction. Suppose $\mathcal{C}_{\mathit{fin}}$ is not closed. Then we have a pair of distinct clauses $C, D \in \mathcal{C}_{\mathit{fin}}$, such that $C \diamond D \notin \mathcal{C}_{\mathit{fin}} \cup \{1\}$. But $C \diamond D$ must have been added to $\mathcal{C}_{\mathit{fin}}$ in some step of the algorithm, which is a contradiction. □

The next lemma we will need is the following:

Lemma 2.5. *Let $\mathcal{C}$ be a set of clauses and let $C, D \in \mathcal{C}$. Then $\mathcal{C}$ and $\mathcal{C} \cup \{C \diamond D\}$ have the same set of satisfying assignments (if any).*

It says that adding resolvents does not change the set of satisfying assignments. Lemma 2.5 implies that if Algorithm 1 ever derives $\emptyset$, then it will correctly say that the original formula ϕ is unsatisfiable. ($\emptyset$ is an unsatisfiable clause.) On the other hand, if we never derive $\emptyset$, then Algorithm 1 will reach Line 15 and call `ExtractAssignment()` with a set of clauses $\mathcal{C}_{\mathit{fin}}$ that does not contain $\emptyset$ and which is closed under resolution (by Lemma 2.4). By Theorem 2.3, `ExtractAssignment()` will find a satisfying assignment to $\mathcal{C}_{\mathit{fin}}$, which will be a satisfying assignment to ϕ by Lemma 2.5.

2.4 Efficient algorithm for 2-SAT

Input	**:** A CNF formula φ on n variables in which each clause has width at most 2 (i.e. contains at most 2 literals)
Output	**:** An $\alpha \in \{0,1\}^n$ such that $\varphi(\alpha) = 1$, or $\perp$ if no satisfying assignment exists

Computational Problem 2-SAT

Runtime of the resolution algorithm for 2-SAT: Note that we will never create a clause of size larger than 2 (this is not true in general for larger initial clauses - why is it true for 2?). By removing 1 literal from each clause and concatenating the remainder of each clause (which has at most 1 literal), the new clause also has at most 2 literals.

Thus, in this case we have $k_{\mathit{fin}} \leq 2$ and $|\mathcal{C}_{\mathit{fin}}| = O(n^2)$, since there are only $O(n^2)$ clauses of size at most 2. So resolution runs in time $O(2 \cdot (n^2)^2) = O(n^4)$ for 2-SAT. An additional factor of n can be saved by only trying to resolve each clause with the $O(n)$ other clauses that share a variable (with opposite sign), yielding a runtime of $O(n^3)$.

Corollary 2.6. *2-SAT can be solved in time $O(n^3)$.*

In CS1240, it is shown how to obtain runtime $O(n + m)$ for 2-SAT, where m is the number of clauses, by reduction to finding strongly connected components of directed graphs. Unfortunately, just like with coloring, once we switch from $k = 2$ to $k = 3$, the best known algorithms still have exponential ($O(c^n)$) worst-case runtimes.

2.5 SAT Solvers

Enormous effort has gone into designing SAT Solvers that perform well on many real-world satisfiability instances, often but not always avoiding the worst-case exponential complexity. These methods are very related to Resolution. In some sense, they can be viewed as interleaving the `ExtractAssignment()` algorithm and Resolution steps, in the hope of quickly finding either a satisfying assignment or a proof of unsatisfiability. For example, they start by assigning a variable (say x_0) to a value $\alpha_0 = 0$. Recursing, they may discover that setting $x_0 = 0$ makes the formula unsatisfiable, in which case they backtrack and try $x_0 = 1$. But in the process of discovering the unsatisfiability of $\mathcal{C}$ with x_0 set to α_0, they may discover many new clauses (by resolution) and these can be translated to resolvents of $\mathcal{C}$ (in a manner similar to Lemma 2.9 below). These new "learned clauses" then can help improve the rest of the search. Many other heuristics are used, such as always setting a variable v as soon as a unit clause (v) or $(\neg v)$ is derived, and carefully selecting which variables and clauses to process next.

2.6 Formalizing Assignment Extraction

This section is optional reading, to give you more precision and proof details about the assignment extraction algorithm. We introduce the following notation:

Definition 2.7. For a (simplified) clause C, a variable v, and an assignment $a \in \{0,1\}$, we write $C|_{v=a}$ to be the simplification of clause C with v set to a. That is,

1. if neither v nor $\neg v$ appears in C, then $C|_{v=a} = C$,
2. if v appears in C and $a = 0$, $C|_{v=a}$ equals C with v removed,
3. if $\neg v$ appears in C and $a = 1$, $C|_{v=a}$ equals C with $\neg v$ removed,
4. if v appears in C and $a = 1$ or if $\neg v$ appears in C and $a = 0$, $C|_{v=a} = 1$.

(We do not need to address the case that both v and $\neg v$ appear in C, since we assume that all clauses are simplified.)

Definition 2.8. For a set $\mathcal{C}$ of clauses, a variable v, and an assignment $a \in \{0,1\}$, we write

$$\mathcal{C}|_{v=a} = \{C|_{v=a} : C \in \mathcal{C}\}.$$

Observe that the satisfying assignments of $\mathcal{C}|_{v=a}$ are exactly the satisfying assignments of $\mathcal{C}$ in which v is assigned a.

To analyze the correctness of `ExtractAssignment()` algorithm, we prove the following:

Lemma 2.9. *Let $\mathcal{C}$ be a set of clauses, v a variable, and $a \in \{0,1\}$ an assignment to v. If $\mathcal{C}$ is closed, then so is $\mathcal{C}|_{v=a}$.*

Proof. Let $C|_{v=a}$ and $D|_{v=a}$ be any two clauses in $\mathcal{C}|_{v=a}$, where $C \in \mathcal{C}$ and $D \in \mathcal{C}$. We need to show that $C|_{v=a} \diamond D|_{v=a} \in \mathcal{C}|_{v=a}$. By definition,

$$C|_{v=a} \diamond D|_{v=a} = \begin{cases} \text{Simplify}((C|_{v=a} - \ell\}) \vee (D|_{v=a} - \neg\ell)) & \text{if } \ell \text{ is a literal s.t. } \ell \in C|_{v=a} \text{ and } \neg\ell \in D|_{v=a} \\ 1 & \text{if there is no such literal } \ell \end{cases}$$

In the former case (where we resolve on ℓ and $\neg\ell$), we have

$$C|_{v=a} \diamond D|_{v=a} = \text{Simplify}((C|v = a - \ell\}) \vee (D|_{v=a} - \neg\ell)) = \text{Simplify}((C - \ell) \vee (D - \neg\ell))|_{v=a} = (C \diamond D)|_{v=a}.$$

That is, we could have resolved on literal ℓ first, then set $v = a$. Since $\mathcal{C}$ is closed, $C \diamond D \in \mathcal{C}$, and hence $(C \diamond D)|_{v=a} \in \mathcal{C}|_{v=a}$. In the latter case (there is no such literal ℓ), we have

$$C|_{v=a} \diamond D|_{v=a} = 1 = 1|_{v=a} \in \mathcal{C}|_{v=a}.$$

□

Lemma 2.9 implies the correctness of `ExtractAssignment(`$\mathcal{C}$`)`. It ensures (by induction) that as we assign $x_0 = \alpha_0, x_1 = \alpha_1, \ldots$, the set $\mathcal{C}$ of variables remains closed. This also implies (by induction) that we never derive the empty clause: since $\mathcal{C}$ is closed and does not contain the empty clause, it cannot contain both (x_i) and $(\neg x_i)$, so our choice of α_i ensures that $\mathcal{C}|_{x_i=\alpha_i}$ does not contain the empty clause. Item 2 of Theorem 2.3 now follows by observing that Algorithm 2 can be implemented in time $O(n + k \cdot |\mathcal{C}|)$.

3 Introduction to Limits of Computation

Thus far in CS 1200, we've focused on what algorithms can do, or what they can do efficiently. In the remainder of the course, we'll talk about what algorithms can't do, or can't do efficiently. In particular, recall Lecture 4's lemma about reductions:

Lemma 3.1. *Let Π and Γ be computational problems such that $\Pi \leq \Gamma$. Then:*

1. *If there exists an algorithm solving Γ, then there exists an algorithm solving Π.*
2. *If there does not exist an algorithm solving Π, then there does not exist an algorithm solving Γ.*
3. *If there exists an algorithm solving Γ with runtime $R(n)$, and $\Pi \leq_{T,q\times h} \Gamma$, then there exists an algorithm solving Π with runtime $O(T(n) + q(n) \cdot R(h(n)))$.*
4. *If there does not exists an algorithm solving Π with runtime $O(T(n) + q(n) \cdot R(h(n)))$, and $\Pi \leq_{T,q\times h} \Gamma$, then there does not exist an algorithm solving Γ with runtime $R(n)$.*

In the last unit of the course, we'll use the Item 2: we'll find a problem Π which we can prove is not solved by any Word-RAM algorithm, then reduce Π to other problems Γ to prove that no Word-RAM algorithm solves them.

Similarly, in the upcoming second-last unit of the course, we'll use item 4: we'll assume that the problem $\Pi = \text{SAT}$ is not solved quickly by any Word-RAM algorithm, then reduce SAT to other problems Γ to prove that no Word-RAM algorithm solves them quickly.

Before we do so, let's consider how fundamental Word-RAM is to the statements above. That is, if we prove limitations of Word-RAM programs, are those limits specific to Word-RAM or are they more general/independent of technology? Could find substantially faster algorithms by choosing a different model of computation than Word RAM, like Python or Minecraft? The answer is conjectured to be "no".

To explain why, we'll first recall our simulation arguments which state that the same problems are solvable by Word-RAM programs, Python programs, and so on.

4 The Church–Turing Thesis

Theorem 4.1 (Turing-equivalent models)**.** *If a computational problem Π is solvable in one of the following models of computation, then it is solvable in all of them:*

- *RAM programs*
- *Word-RAM programs*
- *XOR-extended RAM or Word-RAM programs*
- *%-extended RAM or Word-RAM programs*
- *Python programs*
- *OCaml programs*

- *C programs (modified to allow a variable/growing pointer size)*
- *Turing machines*
- *Lambda calculus*
- $\vdots$

Moreover, there is an algorithm (e.g. a RAM program) that can transform a program in any of these models of computation into an equivalent program in any of the others.

The Church–Turing Thesis: The above equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1. The (equivalent) models of computation in Theorem 4.1 capture our intuitive notion of an algorithm.
2. Every physically realizable computation can be simulated by one of the models in Theorem 4.1.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

Idea behind Theorem 4.1: Recall from the earlier part of the course, a theorem like this is proven via "compilers" and simulation arguments like we have seen several times, giving a procedure to transform programs from one model to another (e.g. simulating XOR-extended Word-RAMs by ordinary Word-RAMs). Like we have seen, we can write simulators for RAM programs in high-level languages like Python and OCaml, and conversely those high-level languages are compiled down to assembly code, which is essentially Word-RAM code.

Simple and elegant models: The λ calculus and Turing machines are extremely simple (even moreso than the RAM model) and mathematically elegant models of computation, coming from the work of Church and Turing, respectively, in 1936, in their attempts to formalize the concept of an algorithm (prior to, and indeed inspiring, the development of general-purpose computer technology). Turing machines are similar to the Word-RAM model, but with a fixed word size and memory access only at a pointer that moves in increments of ± 1. We won't have time to describe the lambda calculus, but it provided the foundation for future functional programming languages like OCaml, and one of the theorems in Turing's paper established the equivalence of Turing machines and the λ calculus.

Input encodings: One detail we are glossing over in Theorem 4.1 is that the different models have different ways of representing their inputs and outputs. For example, natural numbers can be represented directly in RAM programs, but in a Turing machine they need to be encoded as a string (e.g. using binary representation), and in the lambda calculus, they are represented as an operator on functions (which maps a function $f(x)$ to $f^{(n)}(x) = f(f(\cdots f(x))))$. So to be maximally precise, these models are equivalent up to the representation of input and output.