So having stated the motivation for working on AI and the challenges, how should we actually make progress?

Given a complex real-world task, at the end of the day, we need to write some code (and possibly build some hardware too). But there is a huge chasm between the real-world task and code.

A useful paradigm for solving complex tasks is to break them up into two stages. The first stage is modeling, whereby messy real-world tasks are converted into clean formal tasks called models. The second stage is algorithms, where we find efficient ways to solve these formal tasks.

### Example

**Formal task**:
- **Input**: list \( L = [x_1, \ldots, x_n] \) and a function \( f: \mathbb{X} \rightarrow \mathbb{R} \)
- **Output**: \( k \) highest-scoring elements

**Example (\( k = 2 \))**:
\[
L: A \quad B \quad C \quad D \\
f: 3 \quad 2 \quad 7 \quad 1
\]

Two algorithms:
- Scan through to find largest, scan through again to find the second largest, etc.
- Sort \( L \) based on \( f \), return first \( k \) elements
So having stated the motivation for working on AI and the challenges, how should we actually make progress?

Given a complex real-world task, at the end of the day, we need to write some code (and possibly build some hardware too). But there is a huge chasm between the real-world task and code.

A useful paradigm for solving complex tasks is to break them up into two stages. The first stage is **modeling**, whereby messy real-world tasks are converted into clean formal tasks called models. The second stage is **algorithms**, where we find efficient ways to solve these formal tasks.

**Example** ($k = 2$):

$\text{L} : \text{A B C D}$

$\text{f} : 3 2 7 1$

Two algorithms:

• Scan through to find largest, scan through again to find the second largest, etc.

• Sort $\text{L}$ based on $\text{f}$, return first $k$ elements

When you study algorithms, you are generally given a well-defined formal task, something specified with mathematical precision, and your goal is to solve the task. A solution either solves the formal task or it doesn't, and in general, there are many possible solutions with different computational trade-offs.

As an example, suppose you wanted to find the $k$ largest elements in a list of $\text{L} = [x_1, \ldots, x_n]$ according to given a scoring function $\text{f}$ that maps each element into a real-valued score.
How?

Real-world task

• So having stated the motivation for working on AI and the challenges, how should we actually make progress?

• Given a complex real-world task, at the end of the day, we need to write some code (and possibly build some hardware too). But there is a huge chasm between the real-world task and code.

Paradigm

Real-world task

Modeling

Formal task (model)

Algorithms

Program

• A useful paradigm for solving complex tasks is to break them up into two stages. The first stage is modeling, whereby messy real-world tasks are converted into clean formal tasks called models. The second stage is algorithms, where we find efficient ways to solve these formal tasks.

• As an example, suppose you wanted to find the $k$ largest elements in a list of $L = [x_1, \ldots, x_n]$ according to given a scoring function $f$ that maps each element into a real-valued score.

• Solving a formal task involves coming up with increasingly more efficient algorithms for solving the task.

Church-Turing thesis: Program $\approx$ Turing machine

finite action control

\[ \cdots \# \# a_1 a_2 \cdots a_n \# \# \cdots \]

input & output symbols
Finite state machine (fsm)

Formally, a finite state machine (FSM) $M$ is a triple $\langle \text{Trans}, \text{Final}, Q_0 \rangle$ where:

- $\text{Trans}$ is a list of triples $[Q, X, Q_n]$ such that $M$ may, at state $Q$, see symbol $X$ and change state to $Q_n$.
- $\text{Final}$ is a list of $M$’s final (i.e., accepting) states.
- $Q_0$ is $M$’s initial state.

Example:

$\text{Trans} = \{[q_0, a, q_0], [q_0, b, q_1], [q_1, b, q_1]\}$

$\text{Final} = \{q_1\}$

$Q_0 = q_0$
A **fsm** $M$ is a triple $[\text{Trans}, \text{Final}, \text{Q0}]$ where

- **Trans** is a list of triples $[Q,X,Q_n]$ such that $M$ may, at state $Q$ seeing symbol $X$, change state to $Q_n$
- **Final** is a list of $M$’s final (i.e. accepting) states
- **Q0** is $M$’s initial state.

E.g. $\text{Trans} = [[q0,a,q0],[q0,b,q1],[q1,b,q1]]$

$\text{Final} = [q1]$

$Q0 = q0$
From strings to fsm’s

Encode strings as lists; e.g. 102 as \([1, 0, 2]\).
From strings to fsm’s

Encode strings as lists; e.g. 102 as \([1, 0, 2]\).
From strings to fsm’s

Encode strings as lists; e.g. 102 as [1,0,2].

% string2fsm(+String, ?TransitionSet, ?FinalStates)
string2fsm([], [], [q0]).
string2fsm([H|T], Trans, [Last]) :-
    mkTL(T, [H], [[q0, H, [H]]], Trans, Last).

% mkTL(+More, +LastSoFar, +TransSoFar, ?Trans, ?Last)
mkTL([], L, Trans, Trans, L).
mkTL([H|T], L, TransSoFar, Trans, Last) :-
    mkTL(T, [H|L], [[L,H,[H|L]]|TransSoFar], Trans, Last).
From strings to fsm's

Encode strings as lists; e.g. 102 as \([1, 0, 2]\).

% string2fsm(+String, ?TransitionSet, ?FinalStates)
string2fsm([], [], [q0]).
string2fsm([H|T], Trans, [Last]) :-
    mkTL(T, [H], [[q0, H, [H]]], Trans, Last).

% mkTL(+More, +LastSoFar, +TransSoFar, ?Trans, ?Last)
mkTL([], L, Trans, Trans, L).
mkTL([H|T], L, TransSoFar, Trans, Last) :-
    mkTL(T, [H|L], [[L,H,[H|L]]|TransSoFar], Trans, Last).

States as histories (in reverse)
Exercise

Define a 4-ary predicate

\[
\text{accept}(\text{+Trans}, \text{+Final}, \text{+Q0}, \text{?String})
\]

that is true exactly when \([\text{Trans}, \text{Final}, \text{Q0}]\) is a fsm that accepts \text{String} (encoded as a list).
Exercise

Define a 4-ary predicate

\[
\text{accept}(\text{+Trans}, \text{+Final}, \text{+Q0}, \text{?String})
\]

that is true exactly when \([\text{Trans}, \text{Final}, \text{Q0}]\) is a fsm that accepts \(\text{String}\) (encoded as a list).

That is, write a Prolog program to answer queries such as

\[
|?- \text{accept}([[q0,0,q1],[q0,1,q1],[q1,0,q0],[q1,1,q0]], [q1], q0, [1,0,0]).
\]

yes
Exercise

Define a 4-ary predicate

\[
\text{accept}(\text{++Trans,+Final,+Q0,?String})
\]

that is true exactly when \([\text{Trans,Final,Q0}]\) is a fsm that accepts \text{String} (encoded as a list).

That is, write a Prolog program to answer queries such as

\[
|\text{- accept}([[[q0,0,q1],[q0,1,q1],[q1,0,q0],[q1,1,q0]]],
[q1], q0, [1,0,0])\).
\]

\text{yes}

test(String) :- string2fsm(String, Trans, Final),
accept(Trans, Final, q0, String).