Natural Language Processing with Deep Learning

CS224N/Ling284

Gabriel Poesia

Lecture 15: Code Generation

Announcements

1. **Project milestone report due Thursday**
2. Reminder to watch your costs on Azure/AWS with storage!
3. Discussion on Training Large Language Models **tomorrow (Wed, Mar 1st)**
   3:30-4:20 in Skilling Auditorium; see Ed
4. Invited talk this Thursday! Attendance expected for in-person students!
   If SCPD or you can't come, you'll submit a reaction paragraph (see Ed)

Today: Code Generation

1. **Background: Program Synthesis**
2. Program Synthesis as Pragmatic Communication
3. Program Synthesis with Language Models
4. Programs as Tools For Language Models
5. Limitations and Discussion

Program Synthesis

A major long-standing challenge of AI: programs that write programs!

Program synthesizer: program that takes a specification and outputs a program that satisfies it

What specification?

- A logical formula
- Another equivalent program (e.g., a slower one)
- Input/output examples
- **Natural language description**

[Gulwani, 2010]

Program Synthesis from Logical Specifications

- When does synthesis make sense?

- When it's easier to describe *what* the program should do rather than *how* it should do it

- Unlike natural language generation, in program synthesis we can often *test* if a program satisfies the specification

- First attempt: logical formula describing what the program does

Program Synthesis Example: Sorting

- How would you logically specify a sorting algorithm?

- First attempt:
  - Suppose the algorithm takes a list A and outputs a list B.
  - Key property: B should be *sorted*
    - *For all i < length(B), B[i] <= B[i+1]*

Synthesizer

```
def sort(A):
  return [1, 2]
```

Not correct because we didn't mention to sort A and return as B

Program Synthesis Example: Sorting

- How would you logically specify a sorting algorithm?

- Oops! Second attempt:
  - Suppose the algorithm takes a list A and outputs a list B.
  - Key property #1: B should be *sorted*
    - *For all i < length(B), B[i] <= B[i+1]*
  - *Key property #2: B should be a permutation of A*
    - *length(B) = length(A)*
    - *For all B[i] there should exist some A[j] such that A[j] = B[i]*

$$
\begin{aligned}
& \forall k.\ (0 \leq k < n-1) \implies (B[k] \leq B[k+1]) && (1)\\
\land\ & \forall k\ \exists j.\ (0 \leq k < n) \implies (0 \leq j < n \land B[j] = A[k]) && (2)
\end{aligned}
$$

[Gulwani, 2010]

QuickSort algorithm, Recursive

Synthesizer

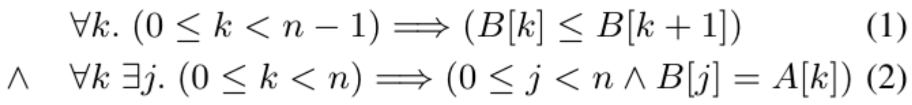
```
def sort(A):
  if len(A) <= 1: return A
  return (sort([x for x in A[1:] if x <= A[0]]) + [A[0]] +
              sort([x for x in A[1:] if x > A[0]])
```

Synthesis from logical specifications

- Note that the problem is non-trivial: the specification says very little about *how*!
  - Not just a translation problem

- But logical specifications are hard to read, hard to write, hard to check

- Often easier to just write the program?



$$\begin{aligned} & \forall k.\ (0 \leq k < n-1) \implies (B[k] \leq B[k+1]) && (1) \\ \wedge \quad & \forall k\ \exists j.\ (0 \leq k < n) \implies (0 \leq j < n \wedge B[j] = A[k]) && (2) \end{aligned}$$

instead of writing logical specifications

[Gulwani, 2010]

Synthesis from **examples**

- Another kind of specification: input/output examples.

- How do you specify sorting a list?
  - Given [3, 2, 1, 0], should return [0, 1, 2, 3]
  - Given [1, 4, 2], should return [1, 2, 4]
  - Given [9], should return [9]

  - How about:

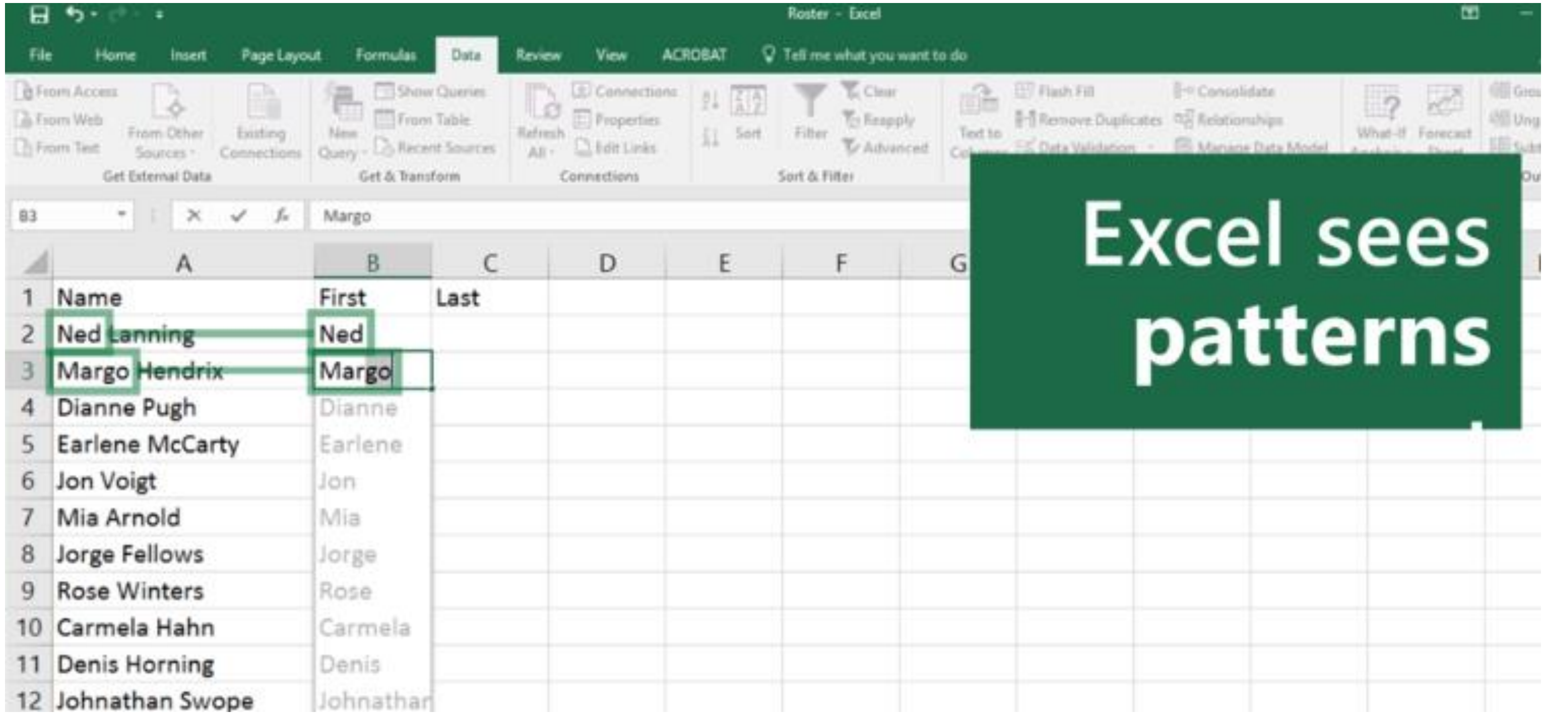
```
def sort(A):
  if len(A) == 4: return list(range(4))
  if len(A) == 3: return [1, 2, 4]
  return [9]
```

suitable only for the examples

Synthesis from examples: **FlashFill**

- Perhaps the first massively deployed program synthesizer was FlashFill, in Microsoft Excel 2013

- Often worked with 1 or 2 examples!



B3 | Margo

	A	B	C
1	Name	First	Last
2	Ned Lanning	Ned	
3	Margo Hendrix	Margo	
4	Dianne Pugh	Dianne	
5	Earlene McCarty	Earlene	
6	Jon Voigt	Jon	
7	Mia Arnold	Mia	
8	Jorge Fellows	Jorge	
9	Rose Winters	Rose	
10	Carmela Hahn	Carmela	
11	Denis Horning	Denis	
12	Johnathan Swope	Johnathan	

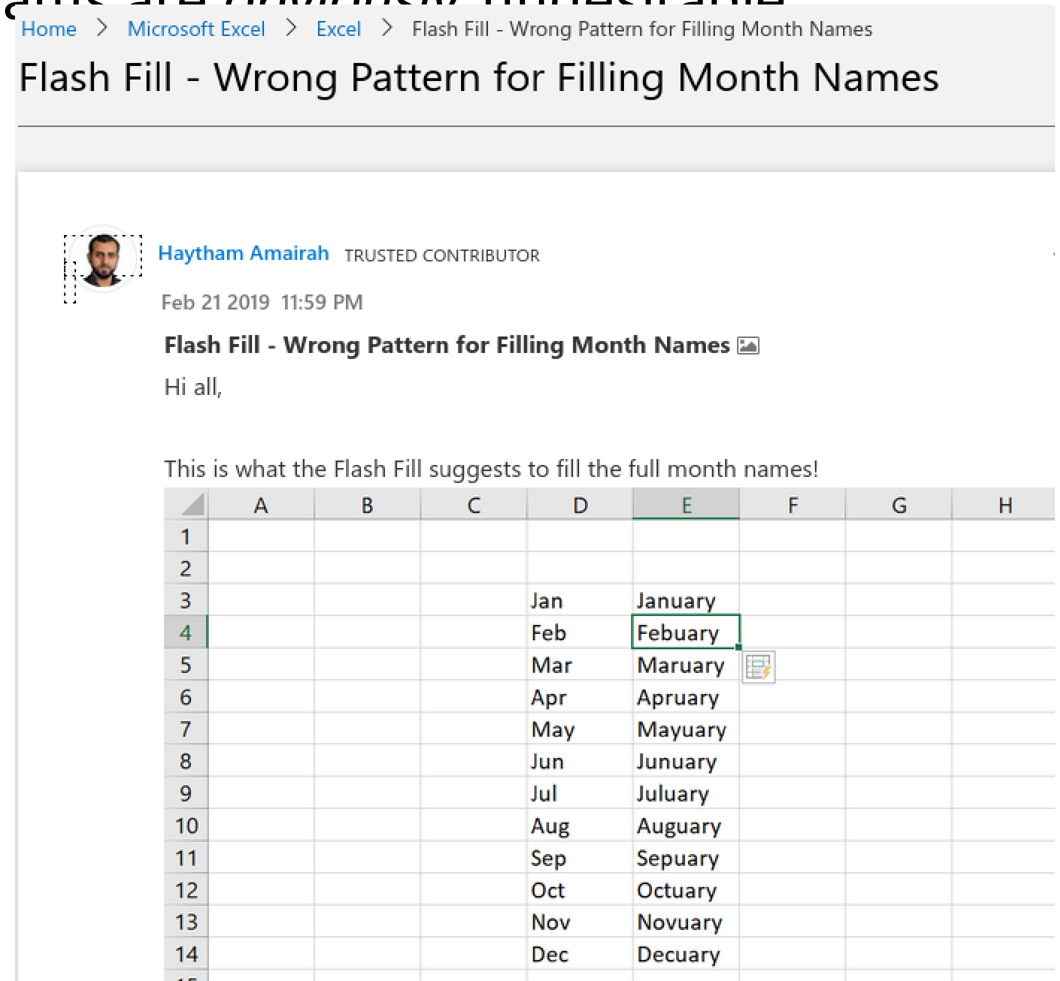
Source: https://support.microsoft.com

Synthesis from examples: **ambiguity**

- Examples are always ambiguous: infinite number of satisfying programs

- There's an implicit human preference: some programs are *obviously* undesirable
  - To you, but not to a search algorithm

- What program is this?
  - "Jan" -> "January"
  - "Feb" -> "February"



Home > Microsoft Excel > Excel > Flash Fill - Wrong Pattern for Filling Month Names

Flash Fill - Wrong Pattern for Filling Month Names

Haytham Amairah TRUSTED CONTRIBUTOR

Feb 21 2019 11:59 PM

Flash Fill - Wrong Pattern for Filling Month Names

Hi all,

This is what the Flash Fill suggests to fill the full month names!

	A	B	C	D	E	F	G	H
1								
2								
3				Jan	January			
4				Feb	Febuary			
5				Mar	Maruary			
6				Apr	Apruary			
7				May	Mayuary			
8				Jun	Junuary			
9				Jul	Juluary			
10				Aug	Auguary			
11				Sep	Sepuary			
12				Oct	Octuary			
13				Nov	Novuary			
14				Dec	Decuary			

Program Synthesis: Summary and Challenges

- A synthesizer should take a higher-level specification of a program and generate an implementation

- Many implications if we make this work: lower barrier to access programming, higher productivity, …

- But many challenges:
  - Infinite space of programs
  - Enumerative search is impractical in real-world languages (e.g., Python)
  - Simple specifications are ambiguous: how to capture human preferences?

Today: Code Generation

1. Background: Program Synthesis
2. **Program Synthesis as Pragmatic Communication**
3. Program Synthesis with Language Models
4. Programs as Tools For Language Models
5. Limitations and Discussion

Ambiguity in natural language

- Human languages are extremely ambiguous, but that's typically not a problem for communication

- In fact, ambiguity is not a bug, it's a feature (of efficiency) [Piantadosi et al, 2012]

- Example: Winograd Schema Challenge
  - "The city councilmen refused the demonstrators a permit because **they feared** violence.
  - "The city councilmen refused the demonstrators a permit because **they advocated** violence."

Syntactically same but the word they refers to different agents in the 2 sentences

- How do we do it?

Pragmatic Reasoning

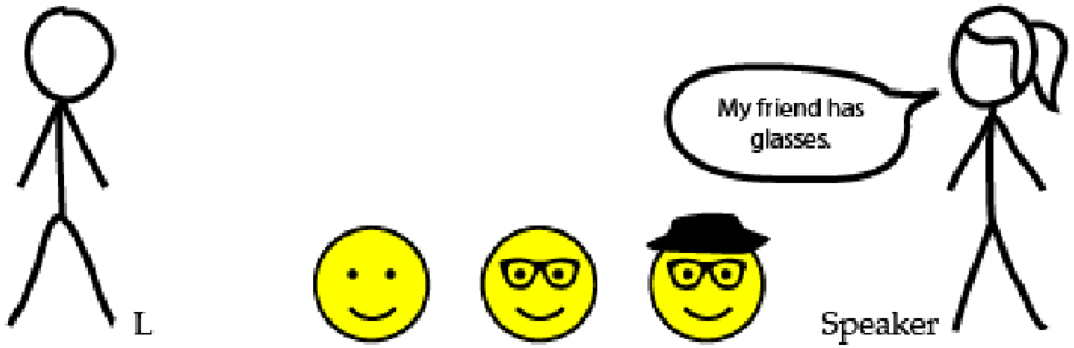
a way to resolve ambiguity

- Human communication depends on *cooperativity*: we assume our conversational partner is trying to be as informative as possible

- We can use that in context to perform *pragmatic reasoning*

- Rational Speech Act (RSA) is a Bayesian model of how we choose (spearker) or interpret (listener) an utterance u given context c by recursively reasoning about the other party

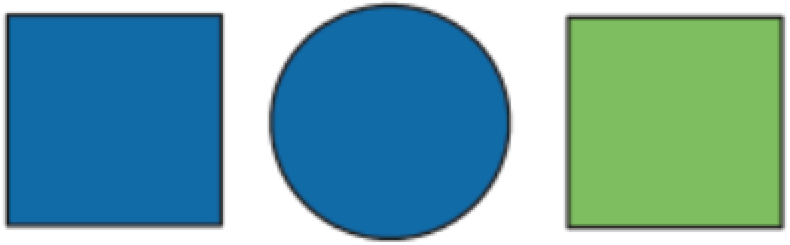
The listener assumes that the speaker is speaking about the 2nd person because the 3rd person would have been referred as the person with the hat.



[Goodman & Frank, 2016]

Rational Speech Acts

- Assume these 3 objects and the following space of utterances: {"blue", "green", "circle", "square"}



o: object

- Given an utterance u, in RSA, a *literal listener L0* would assign: $P_{L0}(o|u) \propto [[u]](o)\, P(o)$

  If squares, L0 would assign uniform probability over the squares

- A *pragmatic speaker* S1 that wants to refer to o chooses u reasoning about L0, balancing (1) how likely L0 is to infer o given u, and (2) how costly is u (e.g., length)
  $P_{S1}(u \mid s) \propto \exp(\text{alpha}((\log P_{L0}(o \mid u) - \text{Cost}(u)))$

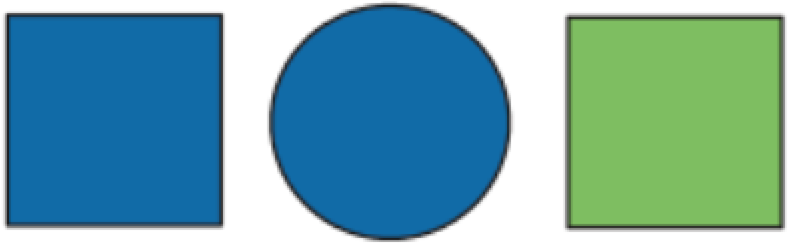
- A *pragmatic listener* L1 interprets utterance u assuming S1 is speaking:
  $P_{L1}(o \mid u) = P_{S1}(u \mid o)\, P(o)$

- We could keep recursing, but the S1-L1 pair is usually enough!

[Goodman & Frank, 2016]

Rational Speech Acts

- Assume these 3 objects and the following space of utterances: {"blue", "green", "circle", "square"}



- Suppose your speaker says: "blue".
  What object do they mean?

- Literal listener L0: puts equal probability in either blue square or blue circle

- Pragmatic listener L1: P(blue circle | blue) = 0.4, P(blue square | blue) = 0.6

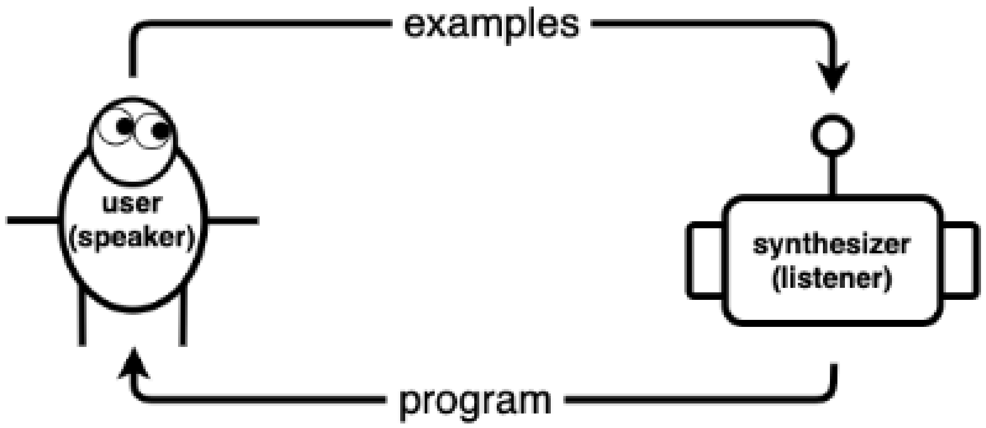
- Agrees with human judgements!

Assumption that blue circle would have been referred to as circle instead of blue

[Goodman & Frank, 2016]

Program Synthesis as Pragmatic Communication

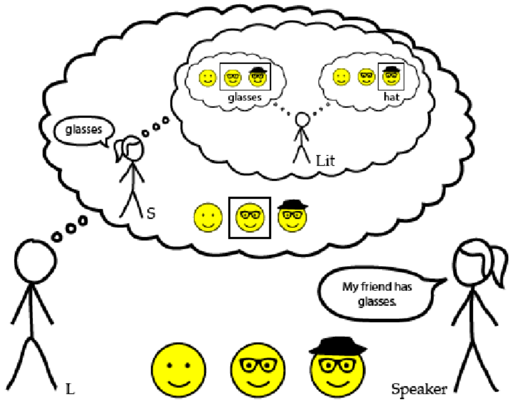
- In program synthesis, our utterances are our specification (e.g., input/output examples)



- The synthesizer is our listener, trying to infer what program we're referring to

- One way to handle ambiguity is with RSA-style reasoning!

Rational Speech Acts

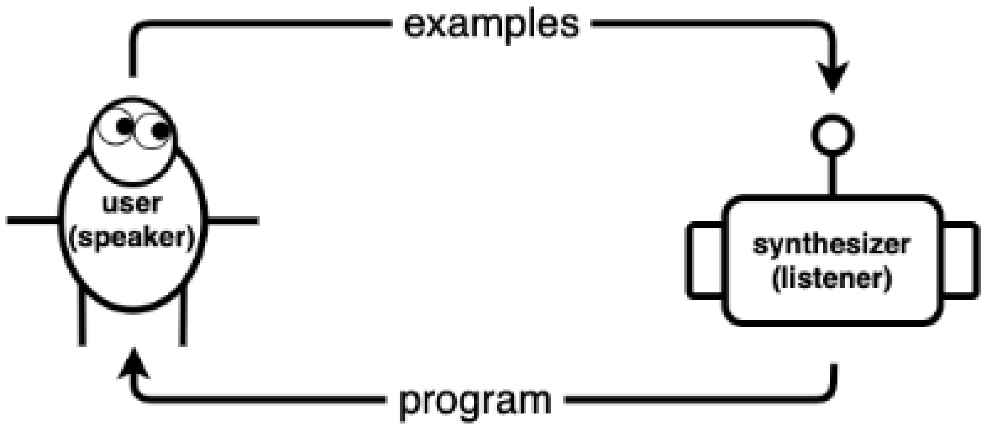


[Pu et al, 2020]

Program Synthesis as Pragmatic Communication

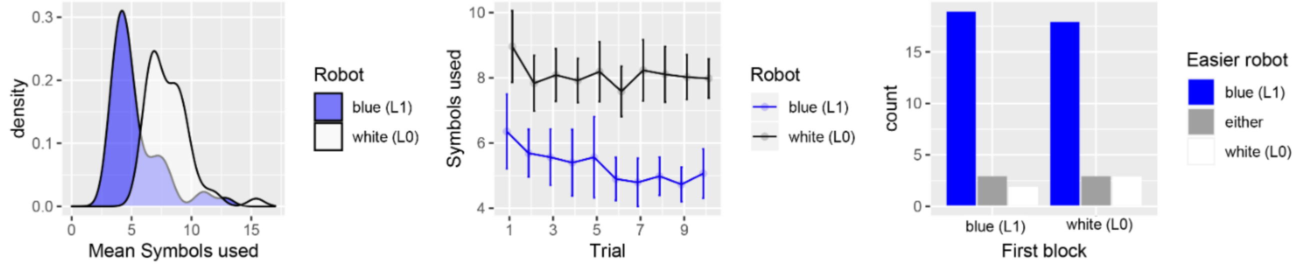
- Assuming a finite set of specifications and of programs, we can build a *meaning matrix:* *M[s][p] = 1* if program p satisfies s   0 otherwise

		h0	h1	h2	h3	h4	h5	h6	h7	h8	h9
		▨□□□	▨▨□□	▨▨▨□	▨▨▨▨	□▨□□	□▨▨□	□▨▨▨	□□▨□	□□▨▨	□□□▨
u0	✔□□□	1	1	1	1	0	0	0	0	0	0
u1	✘□□□	0	0	0	0	1	1	1	1	1	1
u2	□✔□□	0	1	1	1	1	1	1	0	0	0
u3	□✘□□	1	0	0	0	0	0	0	1	1	1
u4	□□✔□	0	0	1	1	0	1	1	1	1	0
u5	□□✘□	1	1	0	0	1	0	0	0	0	1
u6	□□□✔	0	0	0	1	0	0	1	0	1	1
u7	□□□✘	1	1	1	0	1	1	0	1	0	0



- M gives a literal synthesizer; L1 is a pragmatic synthesizer

- Human users needed less examples to get intended program with L1 (blue)



(a) (b) (c)

[Pu et al, 2020]

Program Synthesis as Pragmatic Communication

our work was of finite cases

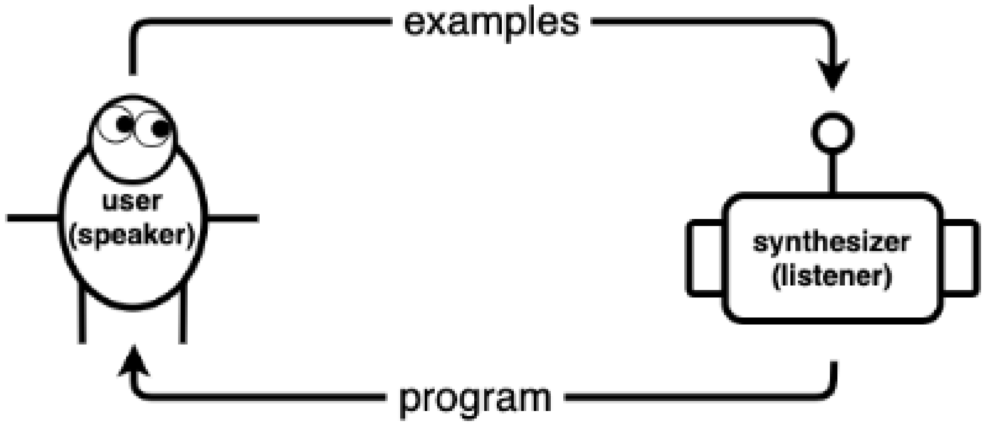
- Works well for finite program / specification spaces

- How to handle infinite programs?

challenge

- Richer specifications, like natural language?

challenge



Until now, we have been talking about search based synthesizers

[Pu et al, 2020]

Today: Code Generation

1. Background: Program Synthesis
2. Program Synthesis as Pragmatic Communication
3. **Program Synthesis with Language Models**
4. Programs as Tools For Language Models
5. Limitations and Discussion

Large language models can generate code

- Language models give us P(continuation | prefix) on Internet text data
- "Stanford University is located in the state of " -> "California" "Ohio"
- "Theodore Landon Streleski (b. 1936) is an American former graduate student in mathematics at Stanford University who" ->
  - GPT-3: "became known for his advocacy of the use of psychedelic drugs for self-exploration and healing"
  - GPT-3: "became a homeless advocate"
  - Wikipedia: "murdered his former faculty advisor"
- "The following is a Python function that, when given the list [1, 3, 2], returns [1, 2, 3]:"
  - GPT-3:
    ```
    "def sort_list(lst):
            lst.sort()
            return lst"
    ```

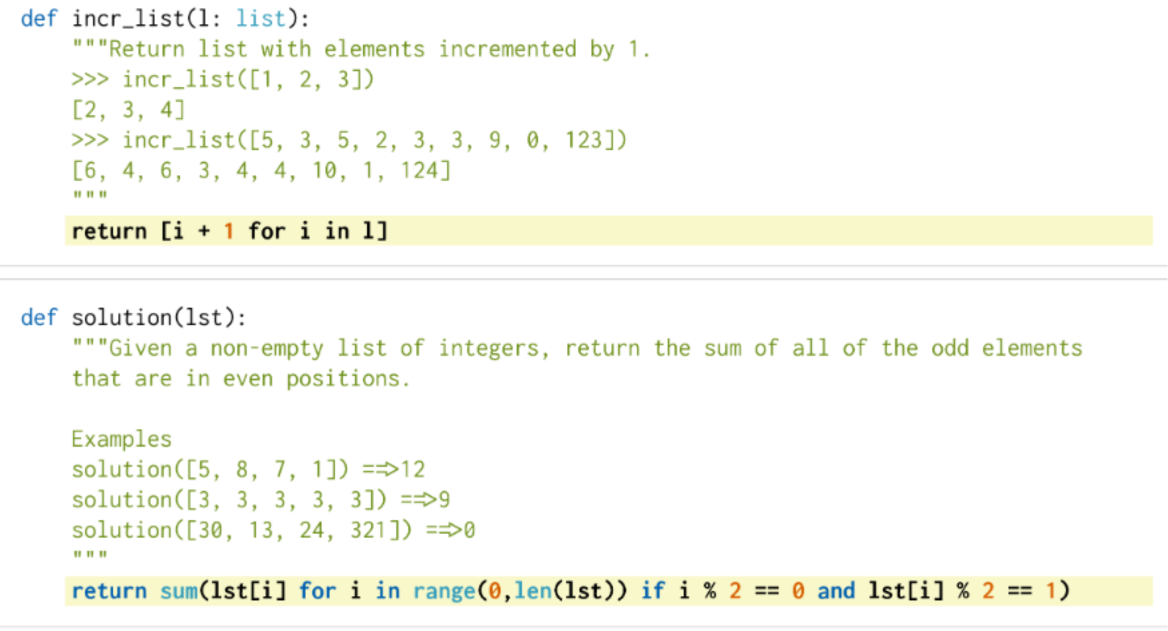
Large language models can generate code

- GPT-3 was able to implement simple Python functions from docstrings without having been explicitly trained for that
- Code is massively available as training data from open source projects (e.g., over 120M public repositories on Github)
- Idea in OpenAI Codex: train large language model on majority code data
- Codex (v1): Same architecture as GPT-3, but with 12B parameters (vs 175B)

[Chen et al, 2021]

Evaluating language models for code generation

- Synthesis challenge: given a Python docstring, generate the function implementation
- How to ensure problems were not seen during training?
- Authors introduced HumanEval, a manually created dataset of 164 problems
- Each problem has a set of hidden tests; a program is correct if it passes all hidden tests
- pass@k: probability that, out of k samples, at least one is correct

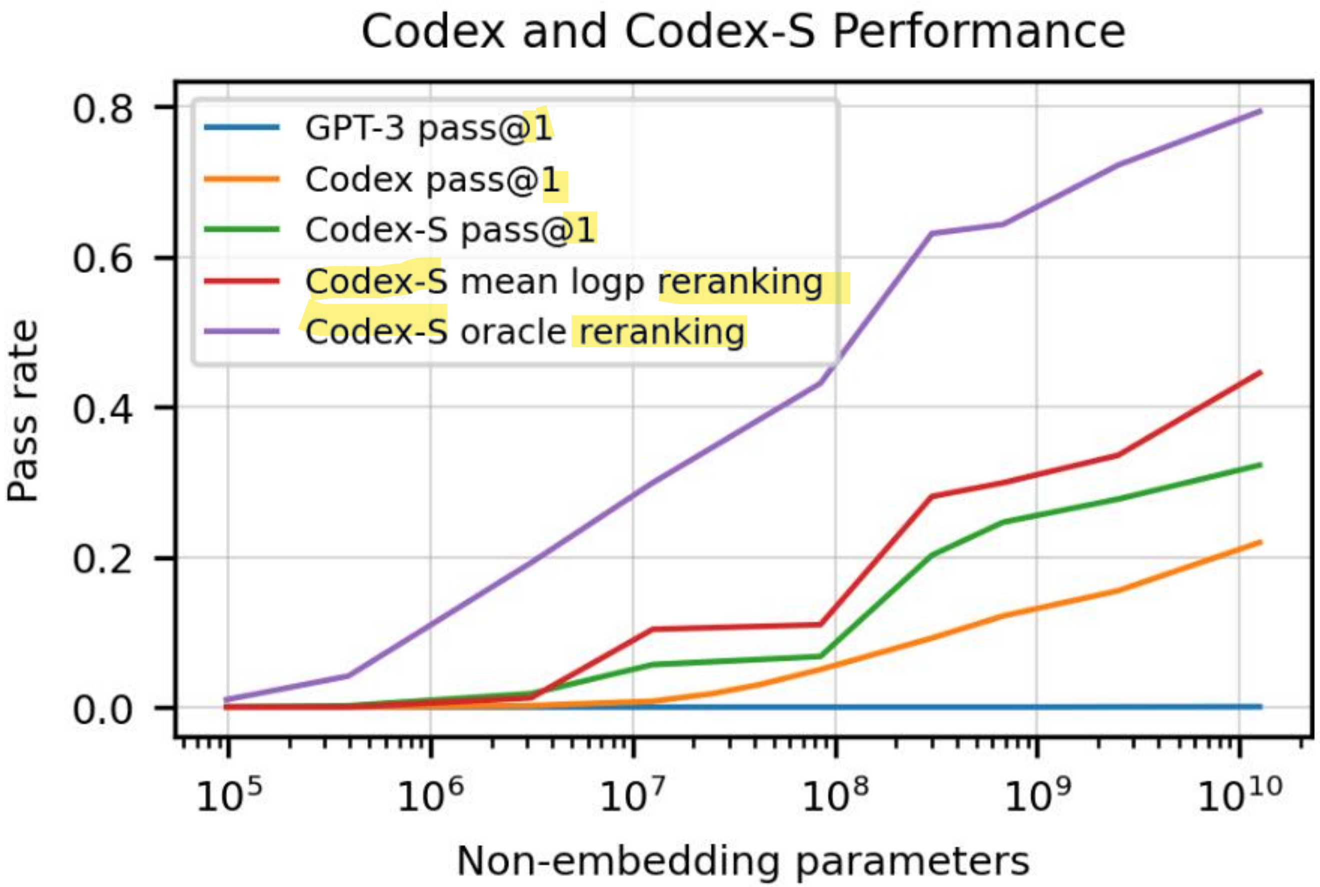


```
def incr_list(l: list):
    """Return list with elements incremented by 1.
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    """
    return [i + 1 for i in l]
```

```
def solution(lst):
    """Given a non-empty list of integers, return the sum of all of the odd elements
    that are in even positions.

    Examples
    solution([5, 8, 7, 1]) ==>12
    solution([3, 3, 3, 3, 3]) ==>9
    solution([30, 13, 24, 321]) ==>0
    """
    return sum(lst[i] for i in range(0,len(lst)) if i % 2 == 0 and lst[i] % 2 == 1)
```

[Chen et al, 2021]

Evaluating language models for code generation



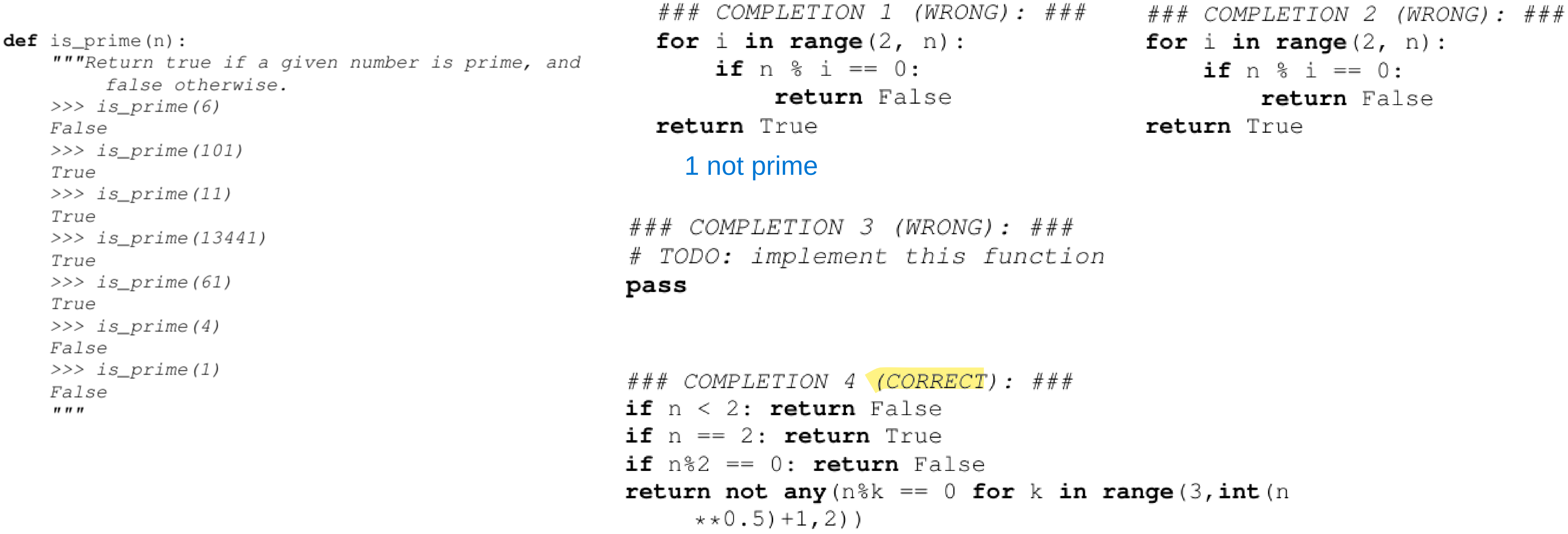
- GPT-3 fails on all problems
- Codex alone has non-trivial performance
- Fine-tuning on problems with this format of function synthesis (Codex-S) improves performance

format: docstring

- Sampling 100 programs, reranking and choosing best improves further

[Chen et al, 2021]

Evaluating language models for code generation



```
def is_prime(n):
    """Return true if a given number is prime, and
        false otherwise.
    >>> is_prime(6)
    False
    >>> is_prime(101)
    True
    >>> is_prime(11)
    True
    >>> is_prime(13441)
    True
    >>> is_prime(61)
    True
    >>> is_prime(4)
    False
    >>> is_prime(1)
    False
    """
```

```
### COMPLETION 1 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

1 not prime

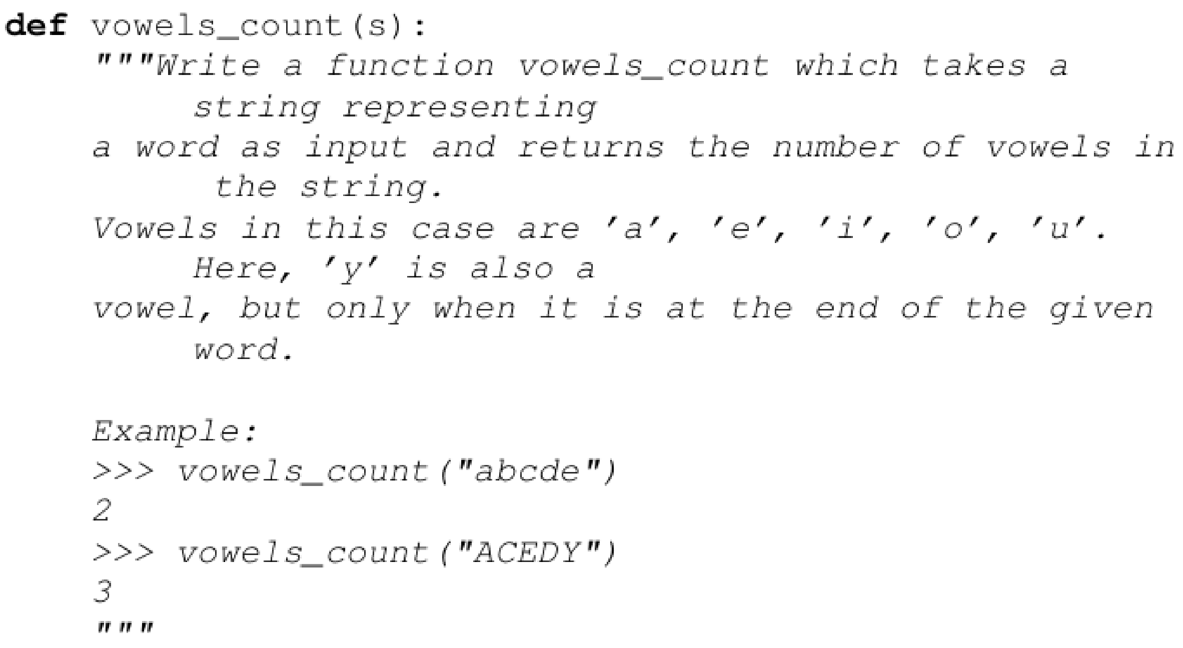
```
### COMPLETION 2 (WRONG): ###
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

```
### COMPLETION 3 (WRONG): ###
# TODO: implement this function
pass
```

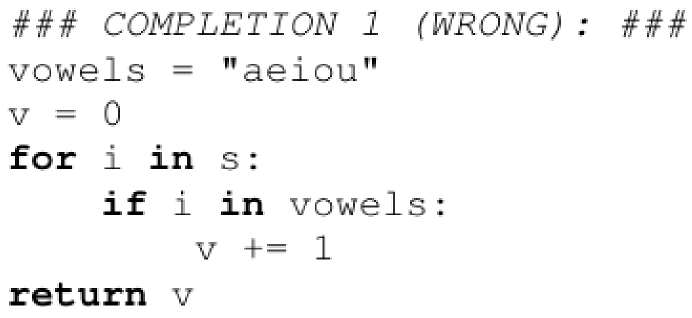
```
### COMPLETION 4 (CORRECT): ###
if n < 2: return False
if n == 2: return True
if n%2 == 0: return False
return not any(n%k == 0 for k in range(3,int(n
    **0.5)+1,2))
```

[Chen et al, 2021]

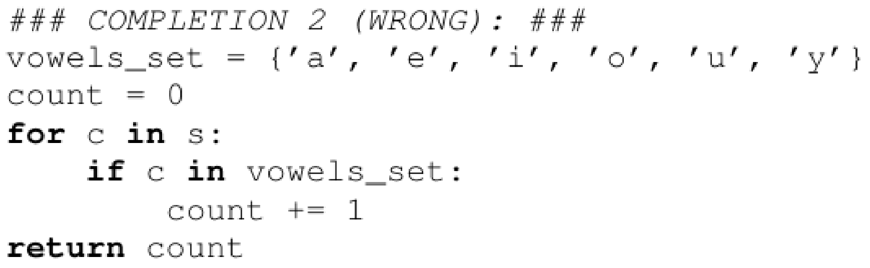
Evaluating language models for code generation



```
def vowels_count(s):
    """Write a function vowels_count which takes a
         string representing
    a word as input and returns the number of vowels in
          the string.
    Vowels in this case are 'a', 'e', 'i', 'o', 'u'.
         Here, 'y' is also a
    vowel, but only when it is at the end of the given
         word.

    Example:
    >>> vowels_count("abcde")
    2
    >>> vowels_count("ACEDY")
    3
    """
```

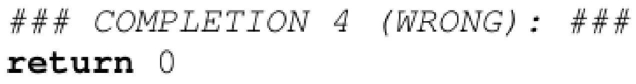


```
### COMPLETION 1 (WRONG): ###
vowels = "aeiou"
v = 0
for i in s:
    if i in vowels:
        v += 1
return v
```



```
### COMPLETION 2 (WRONG): ###
vowels_set = {'a', 'e', 'i', 'o', 'u', 'y'}
count = 0
for c in s:
    if c in vowels_set:
        count += 1
return count
```

samples are usually reasonable even if wrong



```
### COMPLETION 4 (WRONG): ###
return 0
```

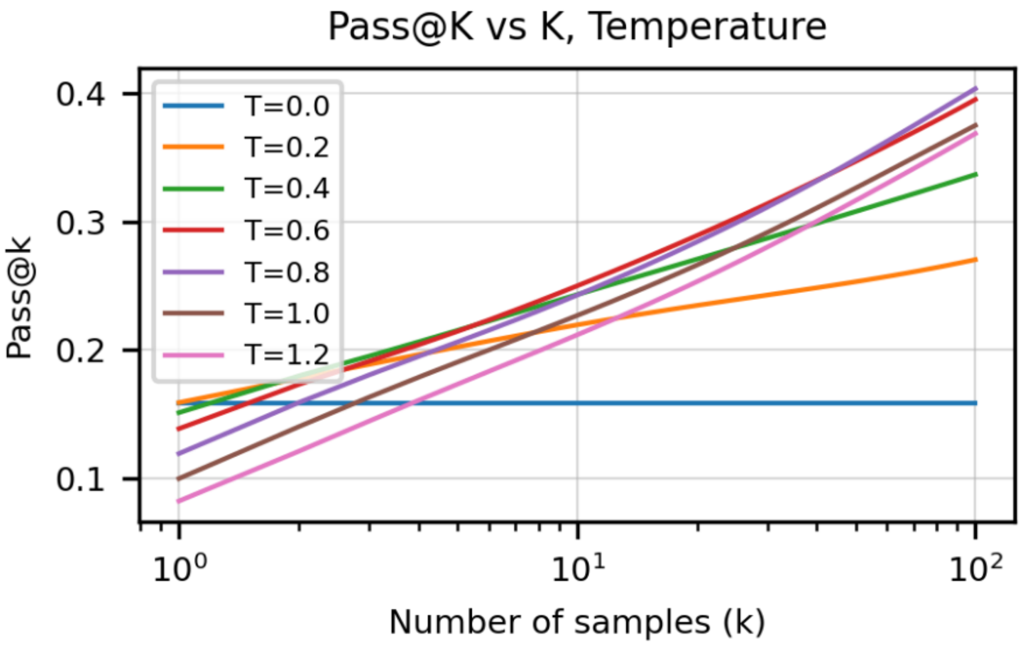
[Chen et al, 2021]

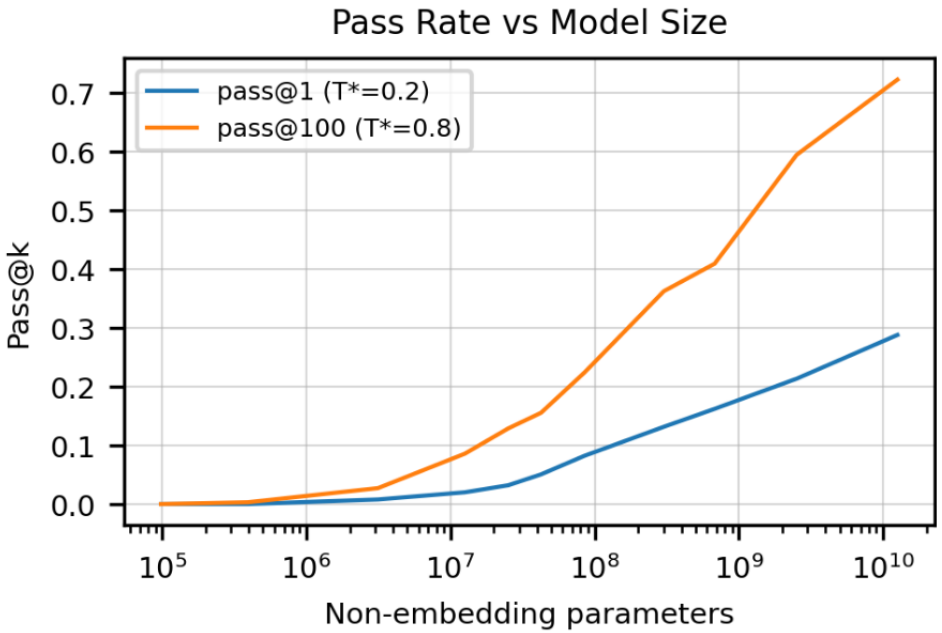
Sampling vs Temperature

- Sampling more programs increases the chance of getting one right

- Trade-off between temperature and P(correct):
  - Low temperature:
    high likelihood (higher P(correct))
    less diversity
  - Higher temperature:
    lower likelihoods
    more diversity

[Chen et al, 2021]

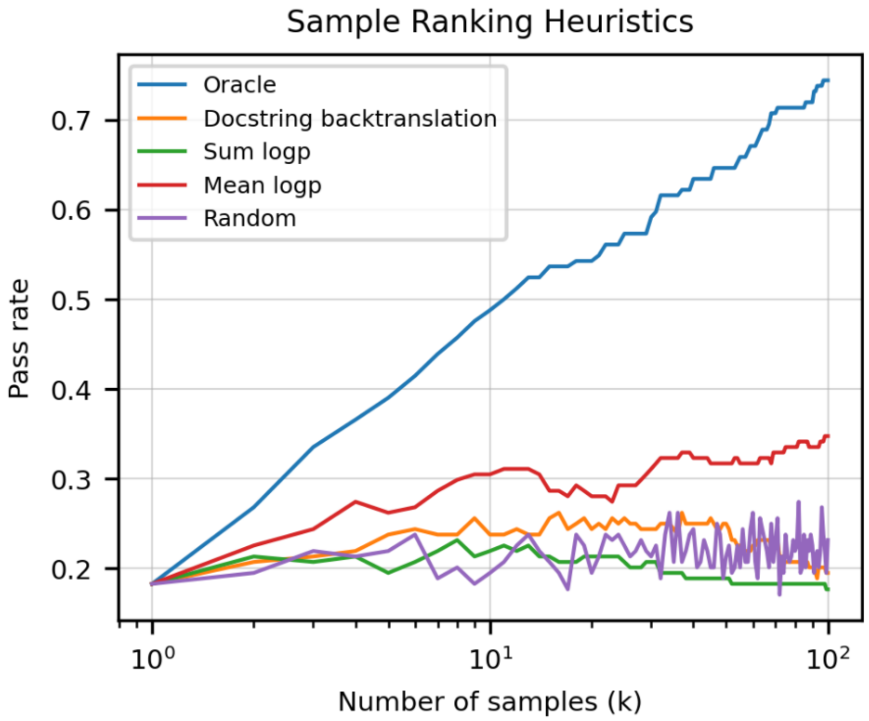




Ranking

At higher number of parameters the models are less likely to make syntactic errors, so the errors are usually the logic.

- For end-users, you don't want to present 100 choices
- Alternative #1: only sample small number of programs
- Alternative #2: sample large number of programs, but re-rank and only show top k for small k
- Oracle: run on all hidden tests and return the program that passes all, if any
- Alternatives: use model's log-probabilities to re-rank



[Chen et al, 2021]

From code to natural language

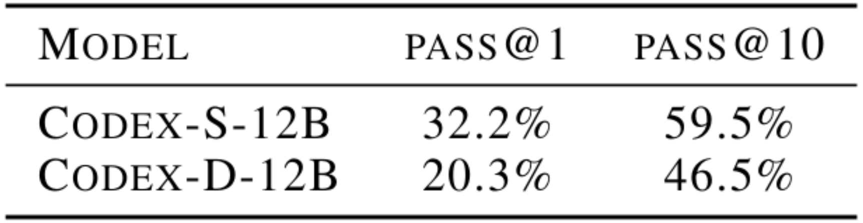
- The last tests sampled from P(code | docstring). What if we sample P(docstring | code)?

- Less frequent in training data since docstring comes before

- Can synthetically create a dataset by inverting this order and fine-tune Codex to obtain Codex-D

- Pass@K estimated by human evaluation

- Lower performance than in the other direction!
  Fine-tuning with teacher forcing did not help



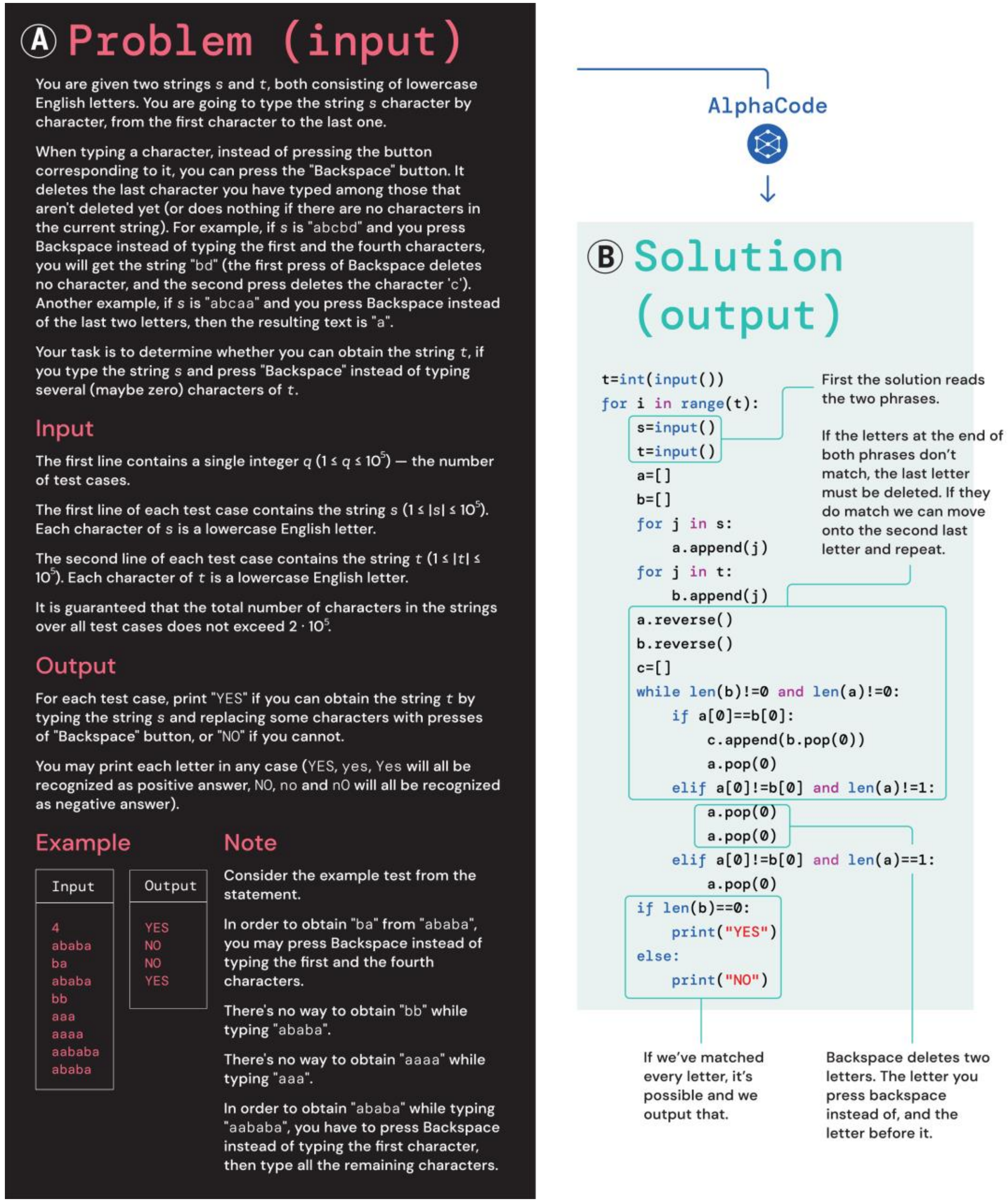
MODEL	PASS@1	PASS@10
CODEX-S-12B	32.2%	59.5%
CODEX-D-12B	20.3%	46.5%

[Chen et al, 2021]

AlphaCode

- In 2022, DeepMind published AlphaCode, a system combining & expanding these ideas to solve competitive programming problems

- Most technical design choices in AlphaCode targeted faster sampling:
  - Unlike Codex, AlphaCode used an encoder-decoder Transformer (faster to encode the problem)
  - Unlike a regular Transformer, they used multi-query attention instead of full multi-head attention blocks (several query heads but single key/value)



Ⓐ Problem (input)

You are given two strings s and t, both consisting of lowercase English letters. You are going to type the string s character by character, from the first character to the last one.

When typing a character, instead of pressing the button corresponding to it, you can press the "Backspace" button. It deletes the last character you have typed among those that aren't deleted yet (or does nothing if there are no characters in the current string). For example, if s is "abcbd" and you press Backspace instead of typing the first and the fourth characters, you will get the string "bd" (the first press of Backspace deletes no character, and the second press deletes the character 'c'). Another example, if s is "abcaa" and you press Backspace instead of the last two letters, then the resulting text is "a".

Your task is to determine whether you can obtain the string t, if you type the string s and press "Backspace" instead of typing several (maybe zero) characters of t.

Input

The first line contains a single integer q ($1 \le q \le 10^5$) — the number of test cases.

The first line of each test case contains the string s ($1 \le |s| \le 10^5$). Each character of s is a lowercase English letter.

The second line of each test case contains the string t ($1 \le |t| \le 10^5$). Each character of t is a lowercase English letter.

It is guaranteed that the total number of characters in the strings over all test cases does not exceed $2 \cdot 10^5$.

Output

For each test case, print "YES" if you can obtain the string t by typing the string s and replacing some characters with presses of "Backspace" button, or "NO" if you cannot.

You may print each letter in any case (YES, yes, Yes will all be recognized as positive answer, NO, no and nO will all be recognized as negative answer).

Example

Input

```
4
ababa
ba
ababa
bb
aaa
aaaa
aababa
ababa
```

Output

```
YES
NO
NO
YES
```

Note

Consider the example test from the statement.

In order to obtain "ba" from "ababa", you may press Backspace instead of typing the first and the fourth characters.

There's no way to obtain "bb" while typing "ababa".

There's no way to obtain "aaaa" while typing "aaa".

In order to obtain "ababa" while typing "aababa", you have to press Backspace instead of typing the first character, then type all the remaining characters.

AlphaCode

Ⓑ Solution (output)

```
t=int(input())
for i in range(t):
    s=input()
    t=input()
    a=[]
    b=[]
    for j in s:
        a.append(j)
    for j in t:
        b.append(j)
    a.reverse()
    b.reverse()
    c=[]
    while len(b)!=0 and len(a)!=0:
        if a[0]==b[0]:
            c.append(b.pop(0))
            a.pop(0)
        elif a[0]!=b[0] and len(a)!=1:
            a.pop(0)
            a.pop(0)
        elif a[0]!=b[0] and len(a)==1:
            a.pop(0)
    if len(b)==0:
        print("YES")
    else:
        print("NO")
```

First the solution reads the two phrases.

If the letters at the end of both phrases don't match, the last letter must be deleted. If they do match we can move onto the second last letter and repeat.

If we've matched every letter, it's possible and we output that.

Backspace deletes two letters. The letter you press backspace instead of, and the letter before it.

[Li et al, 2022]

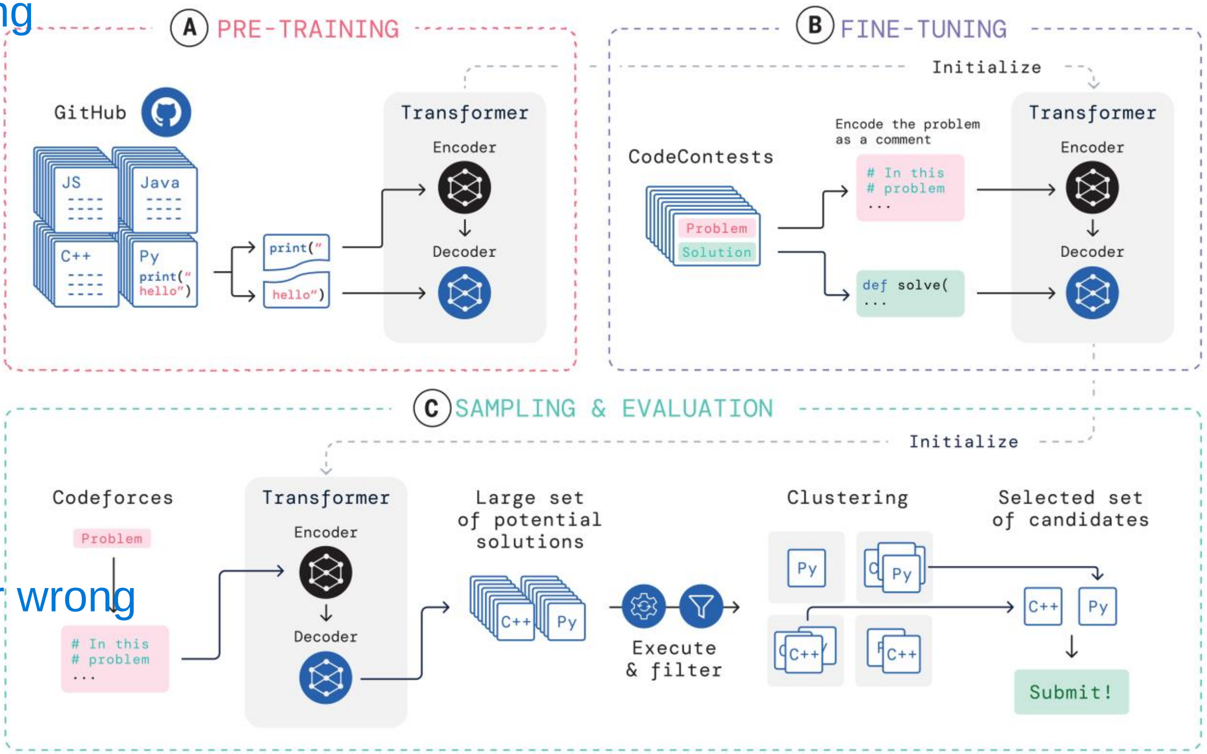
AlphaCode: Pipeline

- Pre-training: standard cross-entropy loss training on 715GB of Github code; encoder had additional MLM loss

Mass language Modeling

- Fine-tuning: human solutions to 13k competitive programming problems:
  - RL fine-tuning (GOLD); only need to learn how to produce one correct solution, instead of necessarily making *all* training solutions likely

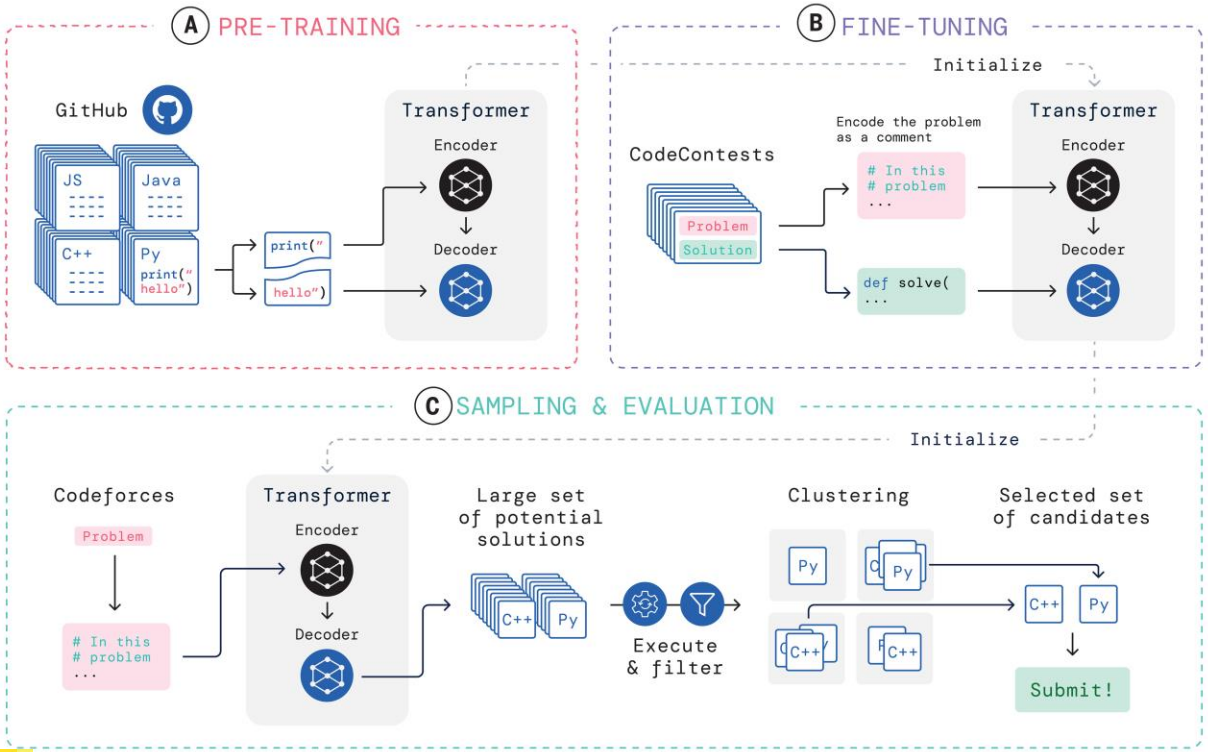
    similar to rlhf but no penalty for wrong
  - Value-conditioning: use incorrect submissions to augment training, but prepend a comment saying whether solution was accepted or not



[Li et al, 2022]

AlphaCode: Pipeline

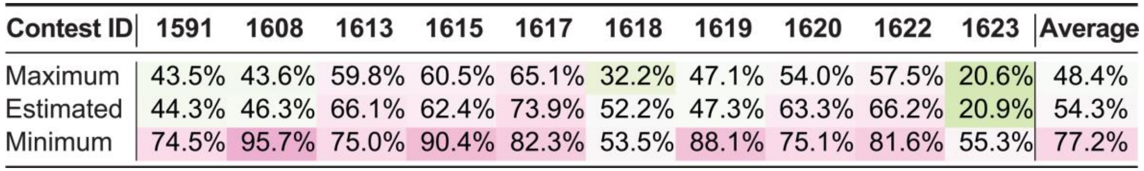
- Sampling:
  - Up to 100k samples per problem (!)
  - Want to do up to 10 submissions per problem; only assume access to a small number of public tests
  - Filtering: discard samples that fail public tests
  - Clustering: trained a separate model to generate inputs; clustered generations by behavior on those inputs; submitted one exemplar from 10 largest clusters



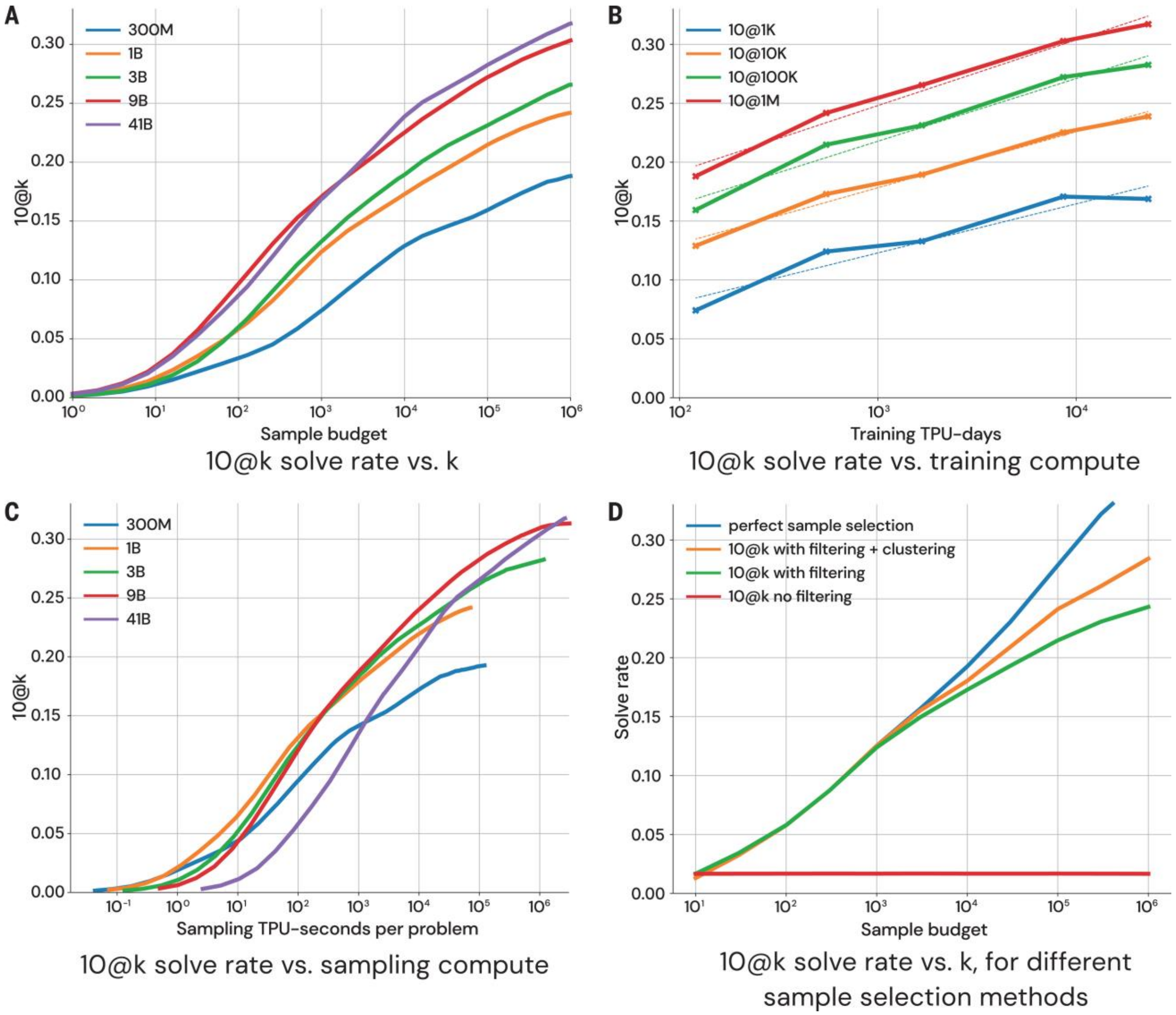
[Li et al, 2022]

AlphaCode: Results

- Log-linear scaling with sample budget (10x samples approx. +6% solve rate)
- Log-linear scaling with compute
- Non-trivial performance on several Division 2 Codeforces contests (below)
  - "approximately corresponds to a novice programmer with a few months to a year of training"



Contest ID	1591	1608	1613	1615	1617	1618	1619	1620	1622	1623	Average
Maximum	43.5%	43.6%	59.8%	60.5%	65.1%	32.2%	47.1%	54.0%	57.5%	20.6%	48.4%
Estimated	44.3%	46.3%	66.1%	62.4%	73.9%	52.2%	47.3%	63.3%	66.2%	20.9%	54.3%
Minimum	74.5%	95.7%	75.0%	90.4%	82.3%	53.5%	88.1%	75.1%	81.6%	55.3%	77.2%



10@k solve rate vs. k

10@k solve rate vs. training compute

10@k solve rate vs. sampling compute

10@k solve rate vs. k, for different sample selection methods

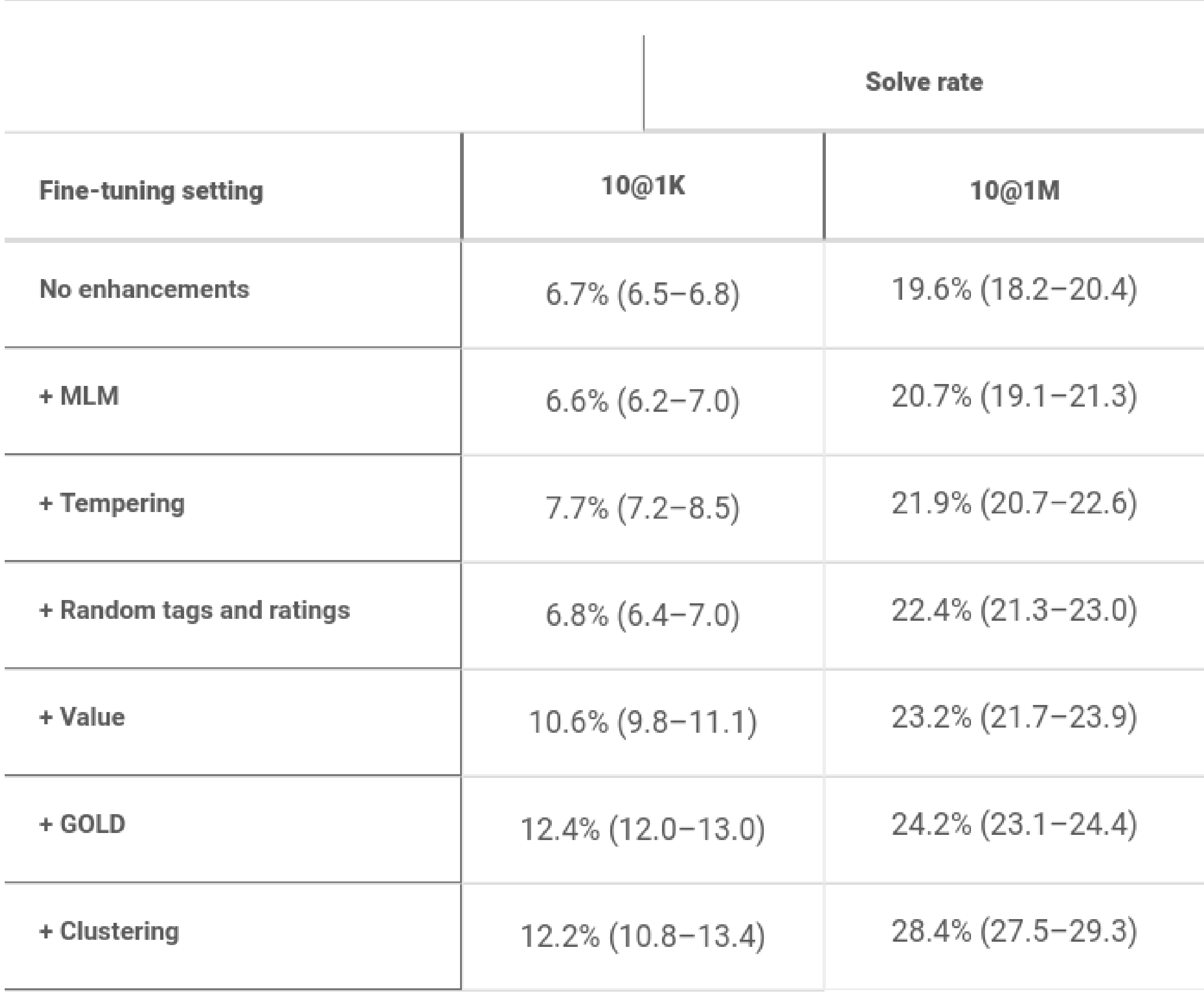
[Li et al, 2022]

AlphaCode: Results & Takeaways

- Sampling more is the largest contributor! Most of their methods don't improve things with at 1k samples

- Shows potential of simple Transformer-based synthesizers taken to scale

- But scale alone won't get this to competing in division 1 at current log-linear rates...



	Solve rate	
Fine-tuning setting	10@1K	10@1M
No enhancements	6.7% (6.5−6.8)	19.6% (18.2−20.4)
+ MLM	6.6% (6.2−7.0)	20.7% (19.1−21.3)
+ Tempering	7.7% (7.2−8.5)	21.9% (20.7−22.6)
+ Random tags and ratings	6.8% (6.4−7.0)	22.4% (21.3−23.0)
+ Value	10.6% (9.8−11.1)	23.2% (21.7−23.9)
+ GOLD	12.4% (12.0−13.0)	24.2% (23.1−24.4)
+ Clustering	12.2% (10.8−13.4)	28.4% (27.5−29.3)

[Li et al, 2022]

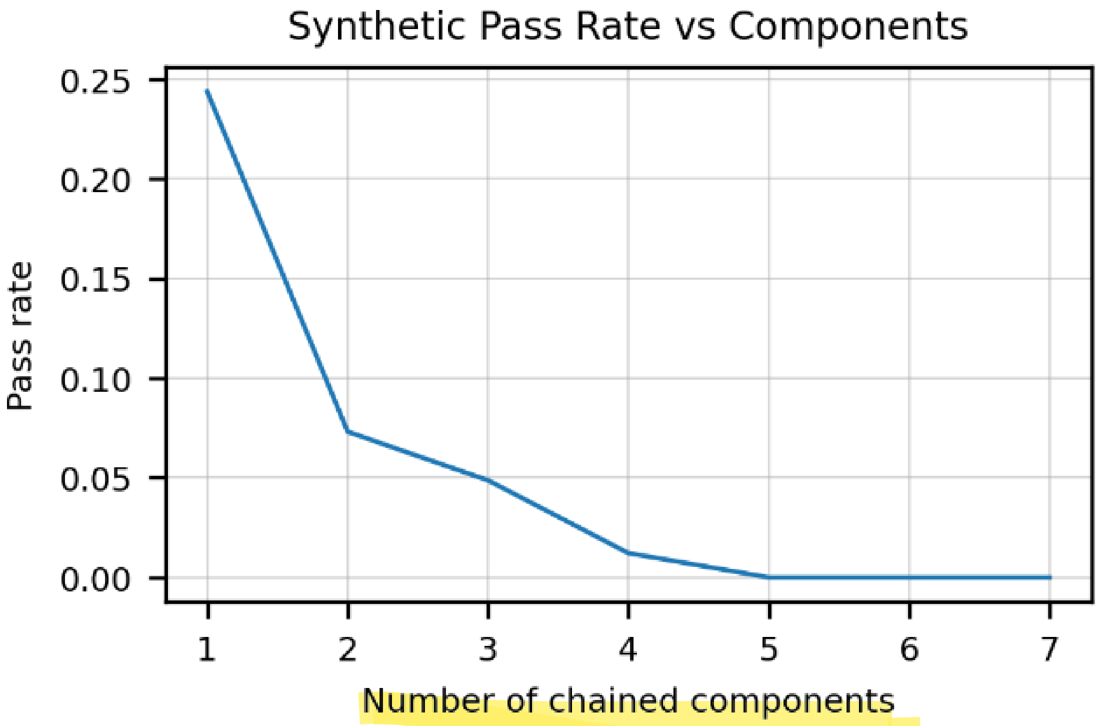
Back to Codex: A note on compositionality

- If a human can trivially solve problem X (e.g., reverse a string), and also problem Y (e.g., compute string length), the problem "do X then Y" is still trivial

- This is not necessarily the case for LMs

- Codex paper reported an experiment with synthetic tasks made by chaining simple components like example above

- Result: performance decays exponentially as components increase, even if individually they're still trivial



[Chen et al, 2021]

Code language models: takeaways

- Transformer models trained on large amounts of code have non-trivial performance in generating programs in real-world programming languages
  - These results were unimaginable just a few years ago

- Sampling, testing and filtering can get quite far
  - Although it gets expensive fast: "Training and evaluating our largest 41B model on Codeforces required a total of 2149 petaflop/s-days and 175 megawatt-hours [~16 times the average American household's yearly energy consumption (29)]."

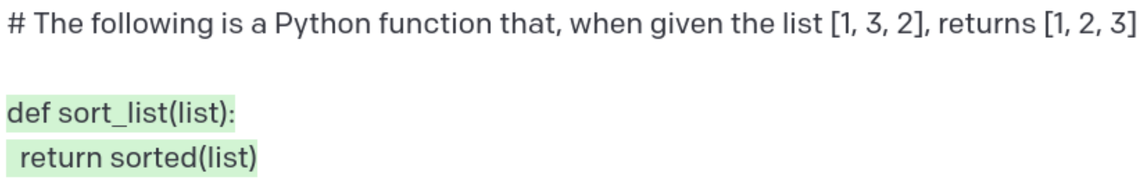
- Still, many of these experiments assume a setting that fundamentally differs from real-world programming
  - Well-defined, self-contained, short problems; extensive existing correctness & performance tests; only need standard libraries; …

- But code LMs can already be quite helpful tools if you can guide them!

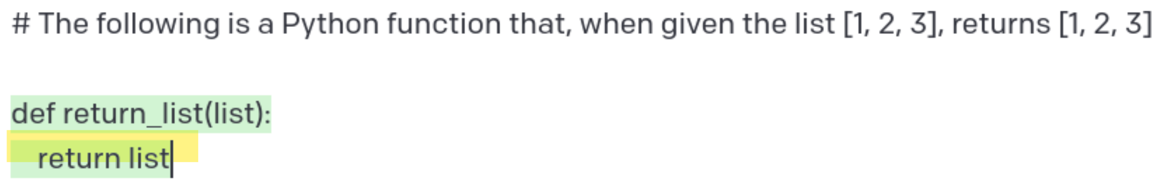
Bonus: pragmatic reasoning

- Codex does some simple pragmatic reasoning!

- What function takes [1, 3, 2] and returns [1, 2, 3]?
  - Probably sorting?
- What function takes [1, 2, 3] and returns [1, 2, 3]?
  - Probably identity?
  - Could also be sorting! But *if you wanted to make an example for sorting you'd likely give a different example*

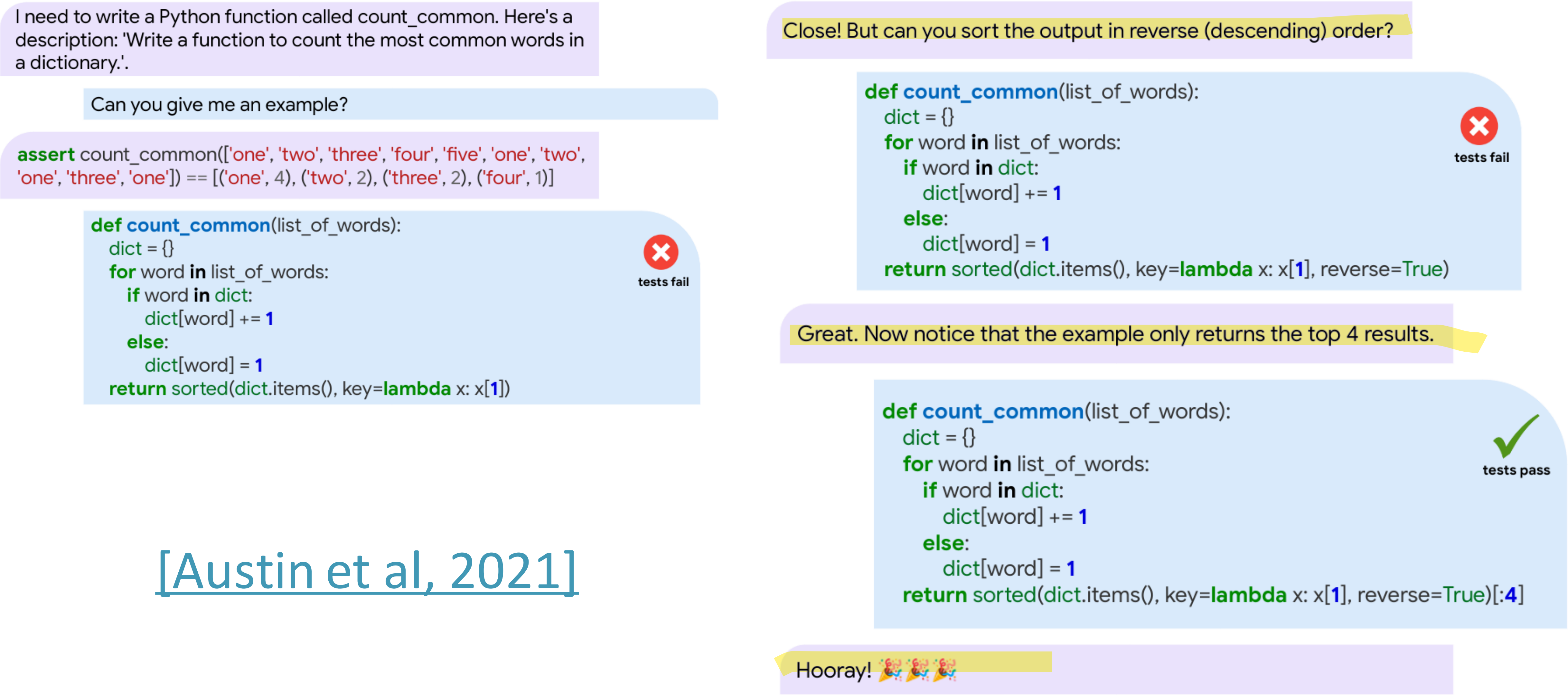


```
# The following is a Python function that, when given the list [1, 3, 2], returns [1, 2, 3]

def sort_list(list):
  return sorted(list)
```



```
# The following is a Python function that, when given the list [1, 2, 3], returns [1, 2, 3]

def return_list(list):
    return list
```

Bonus: editing code in dialogue-style



I need to write a Python function called count_common. Here's a description: 'Write a function to count the most common words in a dictionary.'.

Can you give me an example?

```
assert count_common(['one', 'two', 'three', 'four', 'five', 'one', 'two',
'one', 'three', 'one']) == [('one', 4), ('two', 2), ('three', 2), ('four', 1)]
```

```
def count_common(list_of_words):
  dict = {}
  for word in list_of_words:
    if word in dict:
      dict[word] += 1
    else:
      dict[word] = 1
  return sorted(dict.items(), key=lambda x: x[1])
```

tests fail

Close! But can you sort the output in reverse (descending) order?

```
def count_common(list_of_words):
  dict = {}
  for word in list_of_words:
    if word in dict:
      dict[word] += 1
    else:
      dict[word] = 1
  return sorted(dict.items(), key=lambda x: x[1], reverse=True)
```

tests fail

Great. Now notice that the example only returns the top 4 results.

```
def count_common(list_of_words):
  dict = {}
  for word in list_of_words:
    if word in dict:
      dict[word] += 1
    else:
      dict[word] = 1
  return sorted(dict.items(), key=lambda x: x[1], reverse=True)[:4]
```

tests pass

Hooray! 🎉🎉🎉

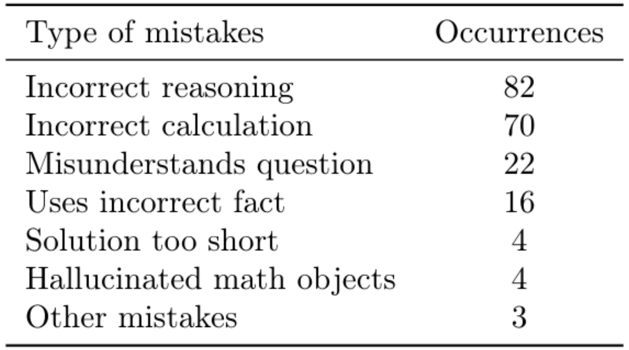
[Austin et al, 2021]

Today: Code Generation

1. Background: Program Synthesis
2. Program Synthesis as Pragmatic Communication
3. Program Synthesis with Language Models
4. **Programs as Tools For Language Models**
5. Limitations and Discussion

Programs as tools

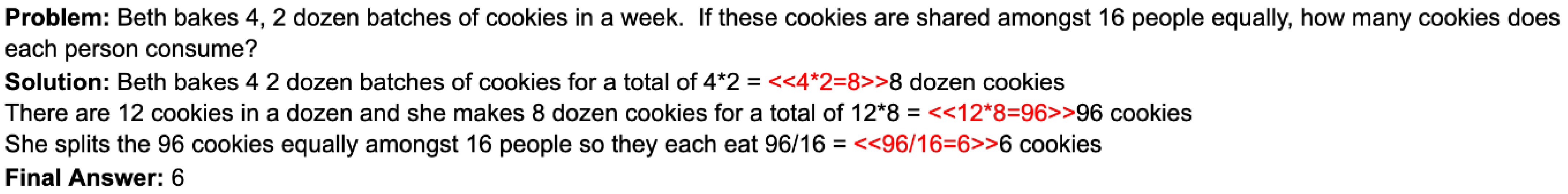
- Humans are effective in large part for our ability to create and use complex tools
  - What is 123 * 456? You'll use a calculator
  - What time is it? You'll use a clock
  - What are the 5 largest airports in the world? You'll do a Web search
  - How many shoe boxes would fit in an average car trunk? You'll possibly do several searches to get data and use a calculator, and perhaps a volume conversion table
- A language model might have the strategy of how to solve these problems, but with standard decoding it can use no external tools
- This is quite limiting! E.g. Minerva, an LLM trained on mostly math, still performs frequent calculation errors when solving math problems.



Type of mistakes	Occurrences
Incorrect reasoning	82
Incorrect calculation	70
Misunderstands question	22
Uses incorrect fact	16
Solution too short	4
Hallucinated math objects	4
Other mistakes	3

[Lewkowycz et al., 2022]

Example: Calculator



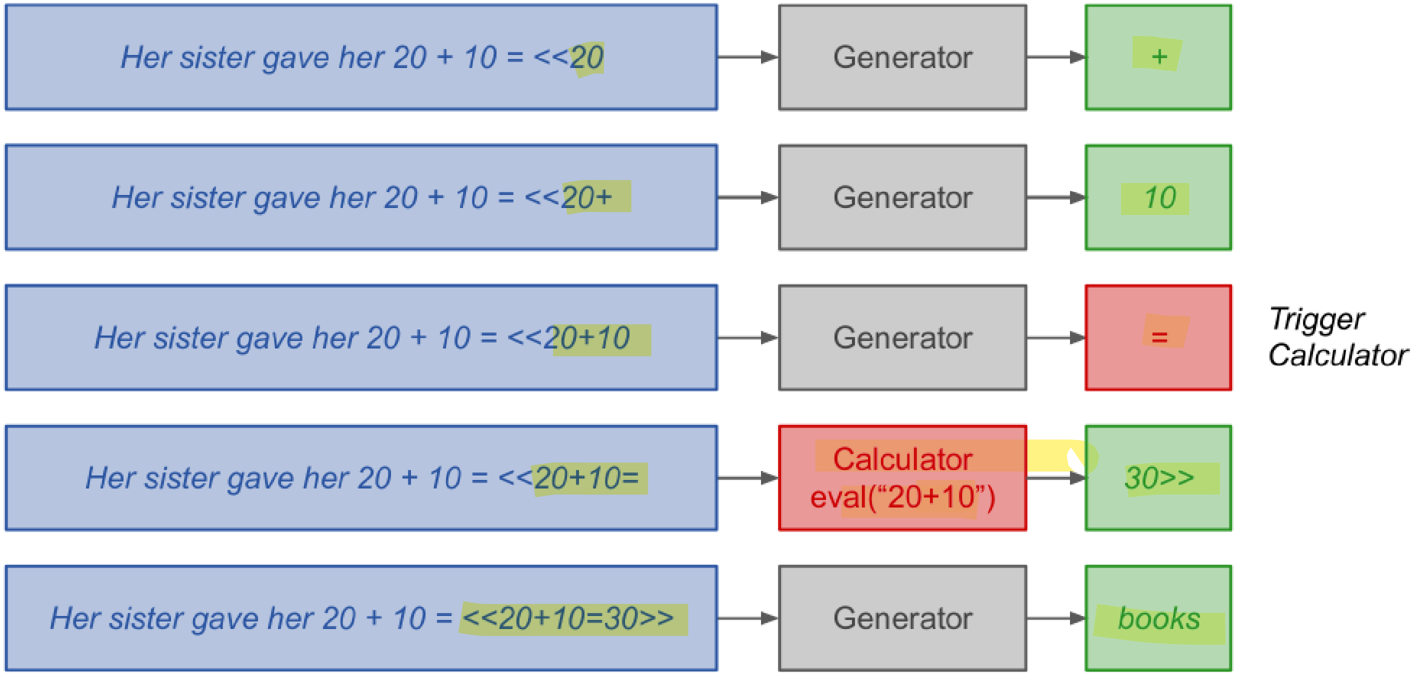
Problem: Beth bakes 4, 2 dozen batches of cookies in a week. If these cookies are shared amongst 16 people equally, how many cookies does each person consume?

Solution: Beth bakes 4 2 dozen batches of cookies for a total of 4*2 = <<4*2=8>>8 dozen cookies
There are 12 cookies in a dozen and she makes 8 dozen cookies for a total of 12*8 = <<12*8=96>>96 cookies
She splits the 96 cookies equally amongst 16 people so they each eat 96/16 = <<96/16=6>>6 cookies

Final Answer: 6

Example: Give the model access to a calculator once it outputs a equal sign

- Key idea:
  - Watch for special input token during decoding
  - When model generates that token, call external tool (e.g., calculator)
  - Paste the result in decoding sequence and keep decoding
  - Model will now generate conditioned on the tool's result!



Her sister gave her 20 + 10 = <<20 → Generator → +

Her sister gave her 20 + 10 = <<20+ → Generator → *10*

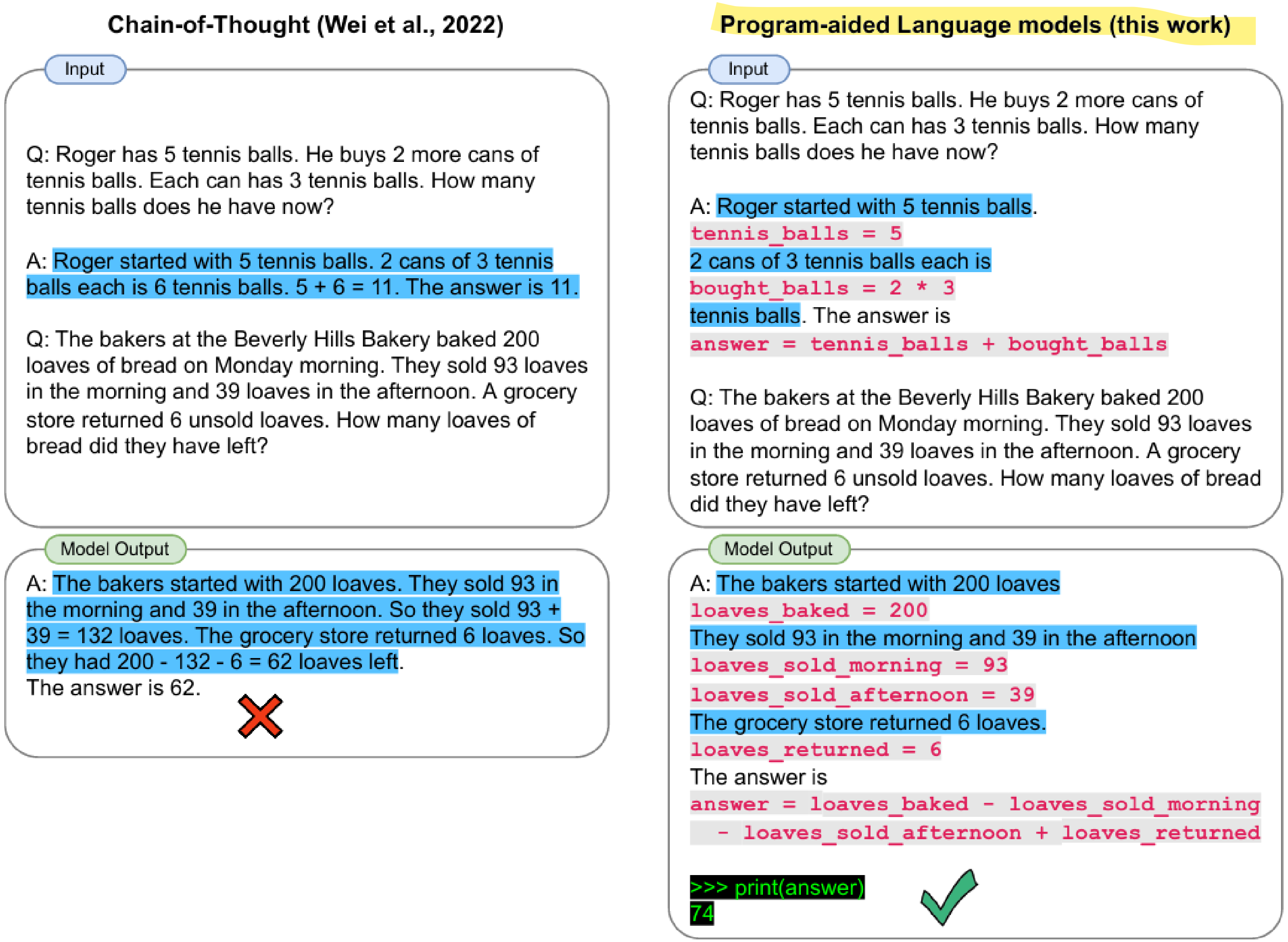
Her sister gave her 20 + 10 = <<20+10 → Generator → = *Trigger Calculator*

Her sister gave her 20 + 10 = <<20+10= → Calculator eval("20+10") → *30>>*

Her sister gave her 20 + 10 = <<20+10=30>> → Generator → *books*

[Kobbe et al, 2021]

Example: Python programs to solve reasoning problems



Chain-of-Thought (Wei et al., 2022)

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11.

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

A: The bakers started with 200 loaves. They sold 93 in the morning and 39 in the afternoon. So they sold 93 + 39 = 132 loaves. The grocery store returned 6 loaves. So they had 200 - 132 - 6 = 62 loaves left.
The answer is 62.

Program-aided Language models (this work)

Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?

A: Roger started with 5 tennis balls.

```
tennis_balls = 5
```

2 cans of 3 tennis balls each is

```
bought_balls = 2 * 3
```

tennis balls. The answer is

```
answer = tennis_balls + bought_balls
```

Q: The bakers at the Beverly Hills Bakery baked 200 loaves of bread on Monday morning. They sold 93 loaves in the morning and 39 loaves in the afternoon. A grocery store returned 6 unsold loaves. How many loaves of bread did they have left?

A: The bakers started with 200 loaves

```
loaves_baked = 200
```

They sold 93 in the morning and 39 in the afternoon

```
loaves_sold_morning = 93
loaves_sold_afternoon = 39
```

The grocery store returned 6 loaves.

```
loaves_returned = 6
```

The answer is

```
answer = loaves_baked - loaves_sold_morning
  - loaves_sold_afternoon + loaves_returned
```

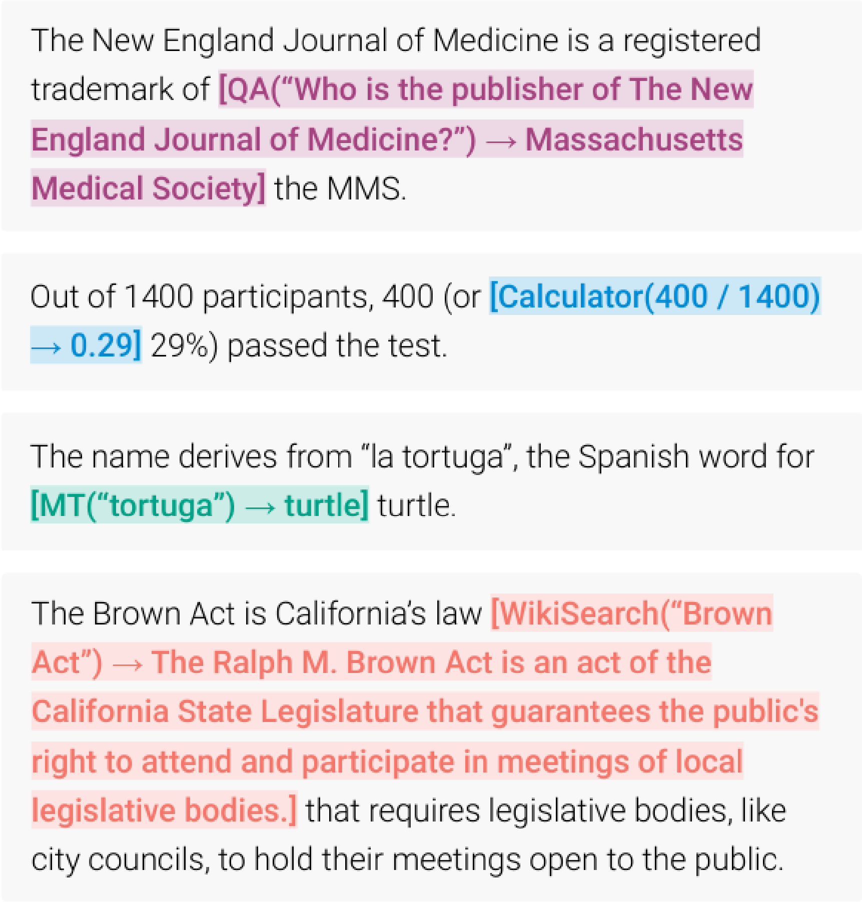
```
>>> print(answer)
74
```

Python code used to solve these problems

[Gao et al, 2022]

Example: Toolformer

- First example (calculator) trained the model to use the calculator by having annotations in the training dataset
- Second (PAL) used few-shot prompting
- Toolformer introduced a self-supervised approach to teach models to use new tools:
  - Start with a few examples of each tool and larger dataset for the task without tool use
  - Use in-context learning to insert candidate API calls in training examples
  - Call APIs, evaluate whether the result decreases perplexity of the rest of the solution
  - Fine-tune model on cases where it does
  - Result: model can now often use APIs



The New England Journal of Medicine is a registered trademark of [QA("Who is the publisher of The New England Journal of Medicine?") → Massachusetts Medical Society] the MMS.

Out of 1400 participants, 400 (or [Calculator(400 / 1400) → 0.29] 29%) passed the test.

The name derives from "la tortuga", the Spanish word for [MT("tortuga") → turtle] turtle.

The Brown Act is California's law [WikiSearch("Brown Act") → The Ralph M. Brown Act is an act of the California State Legislature that guarantees the public's right to attend and participate in meetings of local legislative bodies.] that requires legislative bodies, like city councils, to hold their meetings open to the public.

[Schick et al, 2023]

Today: Code Generation

1. Background: Program Synthesis
2. Program Synthesis as Pragmatic Communication
3. Program Synthesis with Language Models
4. Programs as Tools For Language Models
5. **Limitations and Discussion**

Limitations and Discussion

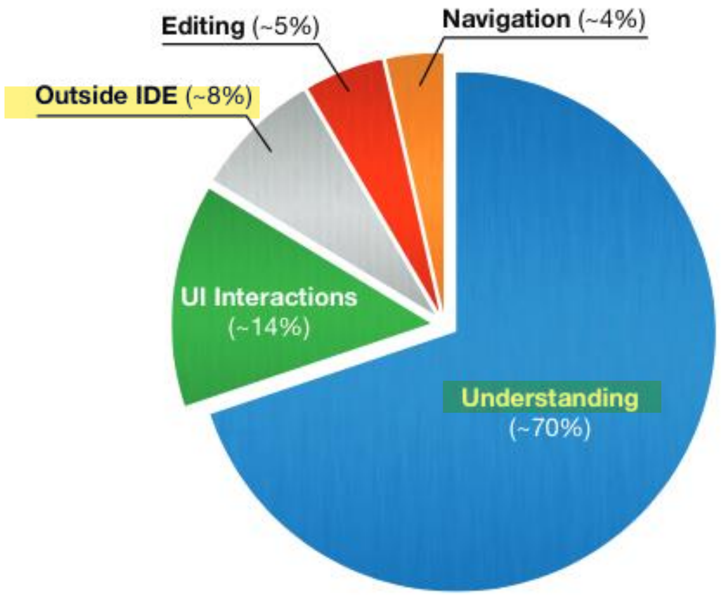
- How much of a programmer's job can current language models automate?

- Relatively very little! Most time in actual software engineering is **not** spent writing code

- Chart shows just time spent with IDE open.
  A lot more happens outside that time
  (talking, prioritizing, meetings)

- A ton of time deciding & discussing *what* to build rather than actually building it

- Even when coding, much of it is editing rather than writing new code



[Minelli et al, 2015]

Limitations and Discussion

- Debugging is very incremental and interactive:
  - Write & run your code
  - See outputs / compiler errors
  - Maybe add a breakpoint / print statement
  - Edit, repeat
- Re-sampling in Codex / AlphaCode ignores outputs/errors
  - As programs grows, probability that a full sample will solve it completely decays exponentially
- Active research in learning to find and fix errors with LMs! (e.g. [Yasunaga & Liang, 2021])
  - Still different setting from open-ended debugging that humans do

Limitations and Discussion

- Lots of public code to pre-train, but does not cover everything:
  - New or internal libraries
  - New programming languages
  - New language features

- Language models fail many tests of code understanding
  - Example: code execution. Given code and inputs, what does it output?
  - [Austin et al, 2021] found that even fine-tuned models struggle

Limitations and Discussion

- Public code repositories have lots of code with bugs
- Generated code often has functional or security bugs. Still need to understand it!
- [Perry & Srivastava et al, 2022] ran a user study where participants solved programming tasks with and without Codex
  - *"Overall, we find that participants who had access to an AI assistant based on OpenAI's codex-davinci-002 model wrote significantly less secure code than those without access"*
  - *"Additionally, participants with access to an AI assistant were more likely to believe they wrote secure code than those without access to the AI assistant."*
- General psychological phenomenon known as *Automation Bias*

Concluding remarks

- Many of these capabilities were completely out of reach until very recently

- Fascinating intersection between natural and programming languages:
  - Natural language is ambiguous, flexible, contextual
  - Programming languages are unambiguous, rigid, precise
  - Humans can bridge these two, and LMs now start to do as well

- Programs are a general representation for reasoning: formal mathematics (theorem-proving languages), legal contracts, tool use, …

- Extremely active areas of research!