COMPUTER SCIENCE

Competition-level code generation with AlphaCode

Yujia Li*†, David Choi*†, Junyoung Chung†, Nate Kushman†, Julian Schrittwieser†, Rémi Leblond†, Tom Eccles†, James Keeling†, Felix Gimeno†, Agustin Dal Lago†, Thomas Hubert†, Peter Choy†, Cyprien de Masson d'Autume†, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, Oriol Vinyals*

Programming is a powerful and ubiquitous problem-solving tool. Systems that can assist programmers or even generate programs themselves could make programming more productive and accessible. Recent transformer-based neural network models show impressive code generation abilities yet still perform poorly on more complex tasks requiring problem-solving skills, such as competitive programming problems. Here, we introduce AlphaCode, a system for code generation that achieved an average ranking in the top 54.3% in simulated evaluations on recent programming competitions on the Codeforces platform. AlphaCode solves problems by generating millions of diverse programs using specially trained transformer-based networks and then filtering and clustering those programs to a maximum of just 10 submissions. This result marks the first time an artificial intelligence system has performed competitively in programming competitions.

Automatically creating programs given a high-level description of what they should do is a long-standing task in computer science (*1*, *2*). Creating an artificial intelligence (AI) system that can solve unforeseen problems by generating code from problem descriptions is a challenge that both affords a greater understanding of problem solving and reasoning (*3*) and leads to important applications, such as improving programmer productivity (*4*) and education (*5*).

Generating code that solves a specified task requires searching in the enormous space of all possible character sequences, only a tiny portion of which corresponds to valid and correct programs. Furthermore, single character edits can completely change program behavior or even cause crashes, and each task has many valid solutions that may be drastically different. These challenges make learning to generate correct programs difficult. Most prior work has been limited to either restricted domain-specific programming languages (*6*) or short code snippets (*7*, *8*). Perhaps the best-known examples of program synthesis are Flash Fill (*6*), which synthesizes programs from input and output examples to automatically fill in data in Microsoft Excel, and code completion tools common in integrated development environments, which boost programmer productivity (*9*, *10*).

Contrasting with the conceptually more complex systems used in most of program synthesis history, recent large-scale transformer-based (*11*) language models [which achieve impressive performance generating text (*12*)] can, with minimal modification, solve simple programming problems in Python (*13*, *14*). A stripped-down version of our model performs similarly to these prior works (table S13). However, those problems consist mostly of simple task descriptions with short solutions, and solving them often amounts to translating a sequence of steps (e.g., adding together all even numbers in a list) directly into code. In contrast, generating entire programs often relies on understanding the task (e.g., win a board game), reasoning out the appropriate algorithm to solve it, and then writing the code to implement that algorithm.

Solving competitive programming problems (Fig. 1A) represents a big step forward. It requires understanding complex natural language descriptions, reasoning about previously unseen problems instead of simply memorizing code snippets, mastering a wide range of algorithms and data structures, and precisely implementing submissions that can span hundreds of lines. To evaluate these submissions (Fig. 1B), they are executed on an exhaustive suite of hidden tests and checked for execution speed and correctness on edge cases. Feedback is minimal; the submission is correct only if it has the correct output on all hidden tests, otherwise it is incorrect. Hidden tests are not visible to the submitter, who must instead write their own tests or rely on the trivial example tests for debugging. Because competitors are allowed to draw on solutions and algorithms from previous contests, challenging new problems are created for each competition. Competitive programming is very popular; events such as the International Collegiate Programming Competition (*15*) and the International Olympiad in Informatics (*16*) date back to the 1970s and are some of the most prestigious competitions in computer science, drawing hundreds of thousands of participants from around the world. Using problems that humans find challenging from battle-tested competitions provides a robust and meaningful benchmark for many aspects of intelligence.

Early work using program synthesis for competitive programming has shown that large transformer models can achieve low single-digit solve rates (*13*, *17*). In contrast, we created a code generation system named AlphaCode that manages to solve 29.6% of test set held-out competitive programming problems in a dataset we released named CodeContests, using at most 10 submissions per problem (comparable to humans). A key driver of AlphaCode's performance came from scaling the number of model samples to orders of magnitude more than previous work; the overall solve rate scaled log-linearly with the number of samples generated, even when only 10 of them were submitted, a sample scaling law similar to those found for training compute and model size (*12*).

When evaluated on simulated programming competitions hosted on the popular Codeforces platform (*18*), AlphaCode achieved an average ranking within the top 54.3% of human participants (a small, selected subset of all programmers). To the best of our knowledge, a computer system has never before achieved such a competitive level in programming competitions.

Learning system

Our system (Fig. 2) was designed to address the main challenges of competitive programming: (i) searching in the huge space of programs, (ii) access to only ~13,000 example tasks

Table 1. AlphaCode's results on Codeforces competitions. For each contest, we show the estimated percent ranking (lower is better) using simulated time and incorrect submission penalties, as well as the best and worst possible rankings using minimum and maximum time penalties, averaged over three evaluations. Percents are how many users performed better than AlphaCode. AlphaCode achieved an overall ranking in the top 54.3% averaged across the 10 contests.

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%

DeepMind, London, UK.
*Corresponding author. Email: yujiali@deepmind.com (Y.L.); davidhchoi@deepmind.com (D.C.); vinyals@deepmind.com (O.V.)
†These authors contributed equally to this work.

for training, and (iii) a restricted number of submissions per problem. Notably, scaling the amount of model samples generated greatly improved performance (Fig. 3), and therefore many aspects of our system were designed to draw samples as quickly as possible, to ensure sample diversity, and to select the best samples for submission. See supplementary materials (SM) text section C for full details of the method.

Creating solutions to problems can be seen as a sequence-to-sequence (*19*) prediction task: Given a problem description X in natural language (e.g., Fig. 1A), produce a corresponding solution Y in a programming language (e.g., Fig. 1B). This task naturally motivated an encoder-decoder transformer architecture (*11*) for AlphaCode, which models $p(Y|X)$. The architecture took as input to the encoder the problem description and metadata X as a flat sequence of tokenized (*20*) characters and sampled Y autoregressively from the decoder one token at a time until an end-of-code token was produced. The code was then ready to be compiled and run against tests.

We pretrained our models on a 715-GB snapshot of human code from GitHub (*21*), with cross-entropy next-token prediction loss. During pretraining, we randomly split code files into two parts, using the first part as input to the encoder and training the decoder to produce the second part. This pretraining learned a strong prior for human coding, enabling subsequent task-specific fine-tuning with a much smaller dataset.

We fine-tuned and evaluated models on a 2.6-GB dataset of competitive programming problems that we created and released under the name CodeContests (*22*). CodeContests includes problems, correct and incorrect human submissions, and test cases. The training set contains 13,328 problems, with an average of 922.4 submissions per problem. The validation and test sets contain 117 and 165 problems, respectively. Full sets of hidden test cases used in competitions were not available, so we included extra generated tests in CodeContests (created by mutating existing tests) to decrease the false positive rate of our model's submissions from 60% (comparable to other datasets) to 4%. We still observed that 42% of solutions were correct but algorithmically inefficient, running out of time or memory on larger test inputs, which motivated the crucial step of evaluating model submissions on the official competitive programming website for the main evaluation. The validation set of CodeContests was used as an easy-to-measure, low-variance proxy of submitting to a competition, and the test set was used only for final evaluations after all training and tuning had been completed. All data used to train the models (in pretraining and fine-tuning) appeared online before all problems in the validation and test sets, ensuring that only information that humans could have seen before a competition would have been available to the models.

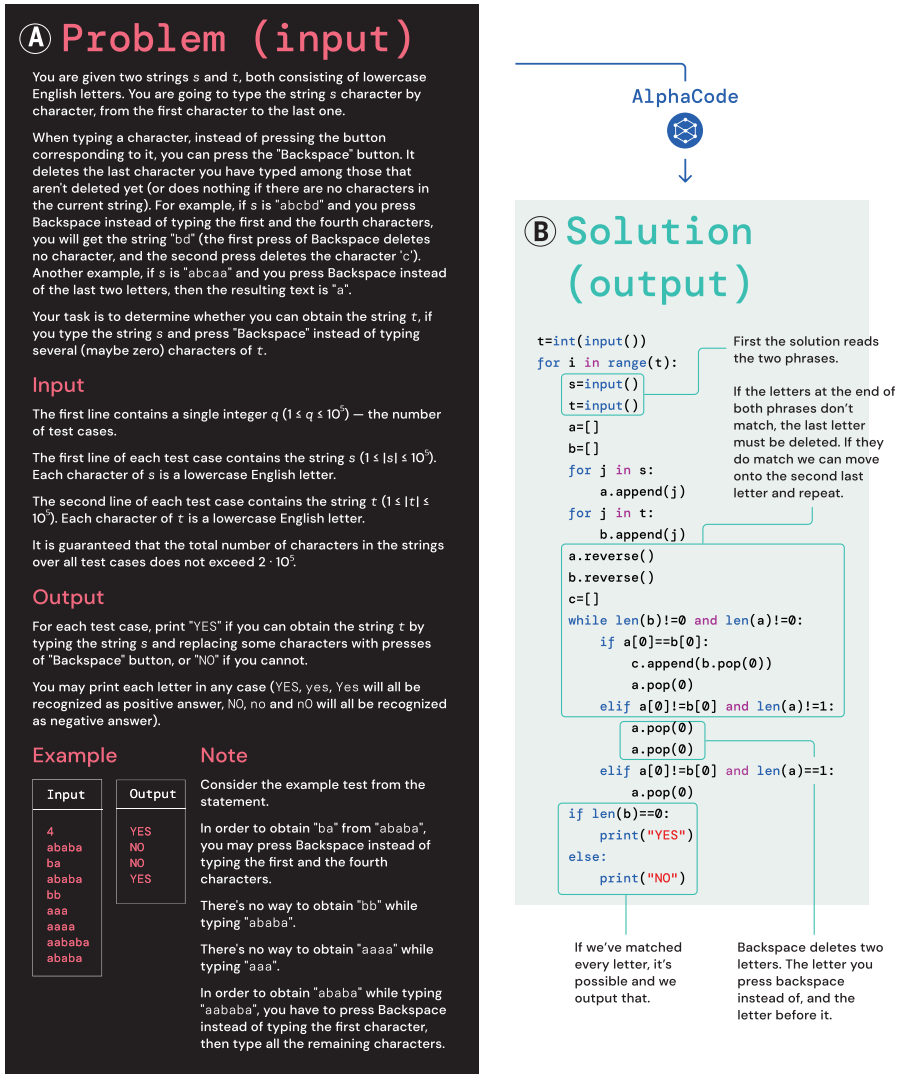

Fig. 1. Competitive programming problem statement and solution. (**A**) Problem statement of 1553D ("Backspace"; https://codeforces.com/problemset/problem/1553/D), a Codeforces problem (*18*). This is a medium-difficulty problem, with a rating of 1500, and its statement includes a public example test case. The entire statement is given to AlphaCode. (**B**) Solution to the problem statement, generated by AlphaCode. Examples of the exact formatting of problem descriptions, solutions, and what the model perceives can be found at https://alphacode.deepmind.com (*32*).

During fine-tuning, we encoded the natural language problem statement as a program comment, to make it appear more similar to files seen during pretraining (which can include extended natural language comments), and used the same next-token prediction loss. We trained a variety of models ranging from 300 million to 41 billion parameters.

Our models included the following enhancements on top of a standard transformer encoder-decoder architecture. Each improvement improved the solve rate, as we show in the evaluation section:

1) Multi-query attention: Multi-query attention (*23*) used a full set of query heads but shared key and value heads per attention block to substantially reduce memory usage and cache updates (which were bottlenecks during sampling). In combination with an encoder-decoder model instead of a decoder-only one, the change increased the sampling speed of AlphaCode more than 10-fold.

2) Masked language modeling (MLM): We included an MLM loss (*24*) on the encoder, which empirically improved model solve rates.

3) Tempering: Tempering (*25*) artificially made the training distribution sharper, producing a smoother sampling distribution (a regularization effect to prevent overfitting).

4) Value conditioning and prediction: We used value conditioning and prediction to discriminate between correct and incorrect

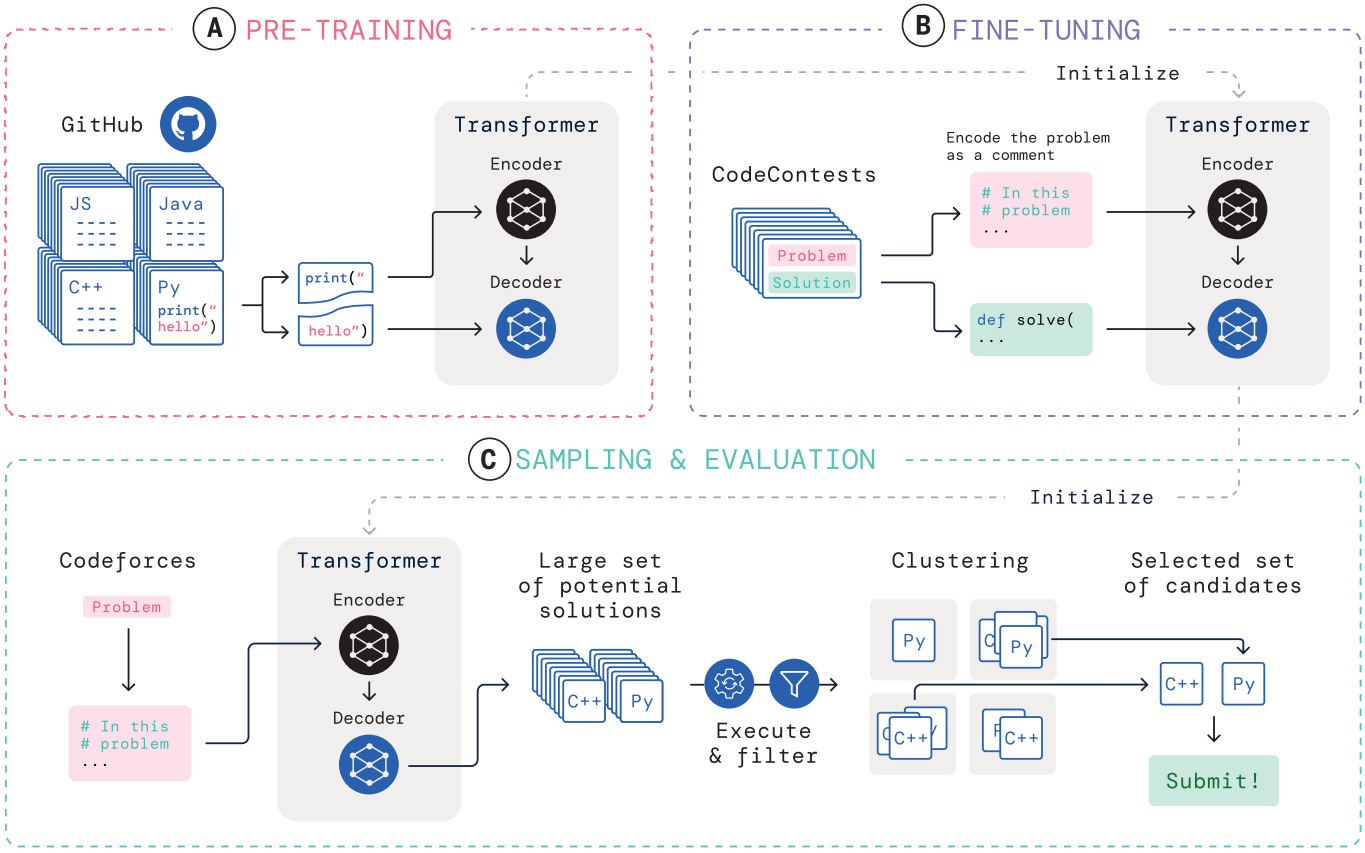

Fig. 2. Overview of AlphaCode. (**A**) In pretraining, files from GitHub are randomly split into two parts. The first part goes to the encoder as input, and the decoder is trained to produce the second part. (**B**) In fine-tuning, problem descriptions (formatted as comments) are given to the encoder, and the decoder is trained to generate the solutions. (**C**) For evaluation, AlphaCode generates many samples for each problem description, then it executes them to filter out bad samples and cluster the remaining ones before finally submitting a small set of candidates.

problem submissions contained in CodeContests, providing an additional training signal and allowing the use of incorrect submission data that might otherwise mislead the model.

5) Generation by off-policy learning from demonstrations (GOLD): The CodeContests training set contains hundreds of solutions for each problem. The standard cross-entropy next-token prediction loss would put equal weight on all solutions. A successful model, however, only needs to generate a single correct solution for each problem. To resolve this discrepancy, we adopted a variation of GOLD (*26*), an offline reinforcement learning (RL) algorithm that focuses training on the most likely solutions for each problem instead of all possible solutions.

At sampling time, the diversity of samples was important, so that the millions of samples for each problem could effectively explore the space of possible solutions. As in another publication (*27*), we ensured sample diversity by using a high temperature and conditioning samples on random metadata: problem difficulty ratings, problem tags (which indicate which techniques a solution might use), and solution programming language.

To select the 10 best samples for submission, we applied filtering and clustering to obtain a small number of candidate submissions on the basis of their program behavior. Filtering executed samples using example tests included in the problem statement and removed samples that failed those tests. This filtering removed ~99% of model samples. The possibly tens of thousands of candidate samples that remained were then clustered by executing them on inputs generated by a separate transformer model trained to do test input generation and by grouping together programs that produced the same outputs on the generated inputs. We then picked one sample from each of the 10 largest clusters for submission, approximately the most likely program behaviors from our model. Intuitively, correct programs would behave the same and form large clusters, but incorrect programs could fail in many different ways.

Evaluation

To assess the performance of AlphaCode, we evaluated it against programming competitions from the Codeforces platform (*18*). Compared with reporting the solve rate on a dataset, this evaluation avoids dataset assumptions and weaknesses that could skew results and allows us to benchmark against the best performers on this task–human competitors.

We ensembled our 41 billion (41B) and 9 billion (9B) parameter models by pooling their samples and then evaluated the ensemble on all Codeforces competitions from 1 December 2021 to 28 December 2021 with >5000 participants per contest, a total of 10 competitions that we believe are a representative sample of Codeforces contests. For each contest, we simulated running AlphaCode live, generating samples for each problem, filtering with example tests (*28*), and clustering to get candidate submissions. We submitted these candidates to the Codeforces platform and computed AlphaCode's placement in each contest (Table 1). After the first run, we repeated this procedure two more times to measure variance.

Overall, our system achieved an average ranking in the top 54.3% when limited to 10 submissions per problem, although 66% of solved problems were solved with the first submission. This performance in competitions approximately corresponds to a novice programmer with a few months to a year of training (see SM text section G). To the best of our knowledge,

Table 2. Buildup ablation for model enhancements. Effect of each additional model enhancement building up from "No enhancements"—a plain fine-tuned 1B encoder-decoder model trained with the standard next-token prediction loss. Numbers in parentheses represent 95% confidence intervals. For each setting, we fine-tuned and sampled from at least three different models from the same pretrained checkpoint.

	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)

this is the first time that a computer system has been competitive with human participants in programming competitions. 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*)]. The large resource usage required for this experiment both has environmental impacts and makes replication challenging for most research entities.

To assess our modeling decisions, we evaluated our models on the validation and test sets of CodeContests. The main CodeContests metric was 10@k: the percentage of problems solved when we take k samples from the model for each problem but can only submit 10 of them for evaluation on the hidden tests. If any of the 10 selected samples solved a problem, the problem is solved. This limit of 10 submissions mimics the upper bound of the number of submissions a human would typically use, and tracking k is important for comparisons, as performance improves as k increases.

With up to 100,000 samples per problem (10@100K), the AlphaCode 41B model solved 29.6% of problems in the CodeContests test set. Prior work on comparable datasets achieved low single-digit solve rates (*13*, *17*), and direct comparisons can be found in SM text section E.3.

Figure 3 shows how the model performance scaled on the 10@k metrics with more samples, that is, as we increased k, or with more compute. These scaling curves highlight a few notable facts about this problem domain and our models:

1) Solve rates scale log-linearly with more samples. As shown in Fig. 3A, the 10@k solve rates scaled approximately log-linearly with k, bending down slightly at high sample budgets. The fact that sampling much more than 10 still improved the 10@k solve rate shows how important it was to sufficiently explore the search space before committing to the final 10 submissions per problem. However, improving the solve rate required exponentially increasing amounts of samples, and the costs quickly became prohibitive.

2) Better models have higher slopes in the scaling curve. Figure 3A also shows that larger models tended to generate higher-quality samples, reflected as better solve rates with the same number of samples and higher slopes in this log-linear scaling curve. Because of log-linear scaling, a better model could reach the same solve rates with exponentially fewer samples than worse models. Improving model quality is an effective way to counter the exponential explosion of sample budget required for a higher solve rate.

3) Solve rates scale log-linearly with more compute. As shown in Fig. 3B, the solve rate also scaled approximately log-linearly with more training compute. Each vertical slice on the curves corresponds to one model size. Figure 3C shows how the solve rate scaled with sampling compute: Larger models required more compute per sample but eventually outperformed smaller models, as the better quality of samples from the larger models became the dominant factor for performance. Both ways of leveraging more compute demonstrate log-linear scaling.

4) Sample selection is critical to solve rate scaling. Figure 3D shows how 10@k solve rates scaled when we used different sample selection methods. With clustering and filtering, as was used in AlphaCode, taking more samples improved the solve rate. However, the combination was far from the theoretical upper bound of performance with perfect sample selection. Without filtering (bottom curve), 10 samples were randomly selected for submission, and taking more samples did not improve the solve rate.

Table 2 shows a buildup ablation of AlphaCode's enhancements, which significantly improved the solve rate on models at the 1B scale. We started from the base setting with no enhancements (beyond the multi-query attention change). We added one new setting at a time, with the final setting that corresponds to AlphaCode reported at the bottom of the table. Combining the six enhancements together increased the 10@1M solve rate from 19.6% to 28.4%, although the improvement depended on the number of samples.

Model analysis

To explore the possibility of solving problems by exploiting weaknesses in the problem structure, we analyzed the capabilities and limitations of our models. A common concern for large language models trained on large amounts of data is that they solve downstream problems by simply memorizing the training set (*30*, *31*). For competitive programming to be a good test of problem-solving ability, models need to come up with novel solutions to solve new problems. We found no evidence that our model copied core logic from the training data. Copying solutions from the training set was not sufficient to solve any problems in the unseen validation set, and the longest substrings that both human solutions and AlphaCode solutions shared with the training data were of similar lengths (fig. S10). Further manual investigation showed that core logic of model solutions necessary to solve problems could not be found in the training data.

We also analyzed the properties of model samples. Like humans, models correctly solved easier problems more often than harder problems; the 41B model solved 62.4% of validation problems rated between 800 and 1100 and 7.8% of problems rated between 2400 and 2700 (higher-rated problems are more difficult). Models had the highest solve rates for problems that deal with bitmasks, sorting, greedy algorithms, or math, but the lowest solve rates for dynamic programming, constructive algorithm, and graph problems. The former categories tend to afford shorter solutions and the latter categories afford longer ones, indicating that AlphaCode had trouble producing longer solutions.

AlphaCode was sensitive to changes in the problem descriptions, indicating that it did not exploit obvious weaknesses in the task structure. We modified a problem statement that asked for a program that can find the maximum product of two consecutive array elements. When AlphaCode generated samples for the opposite problem (minimum instead of maximum product) and a related problem (any elements instead of consecutive ones), the percentage correctness of those samples on the original problem decreased from 17.1% to 0.1% and 3.2% for the opposite and related rewordings, respectively. AlphaCode attended to important changes in problem descriptions that fundamentally changed the problem, instead of uniformly trying every related algorithm by abusing the large sample

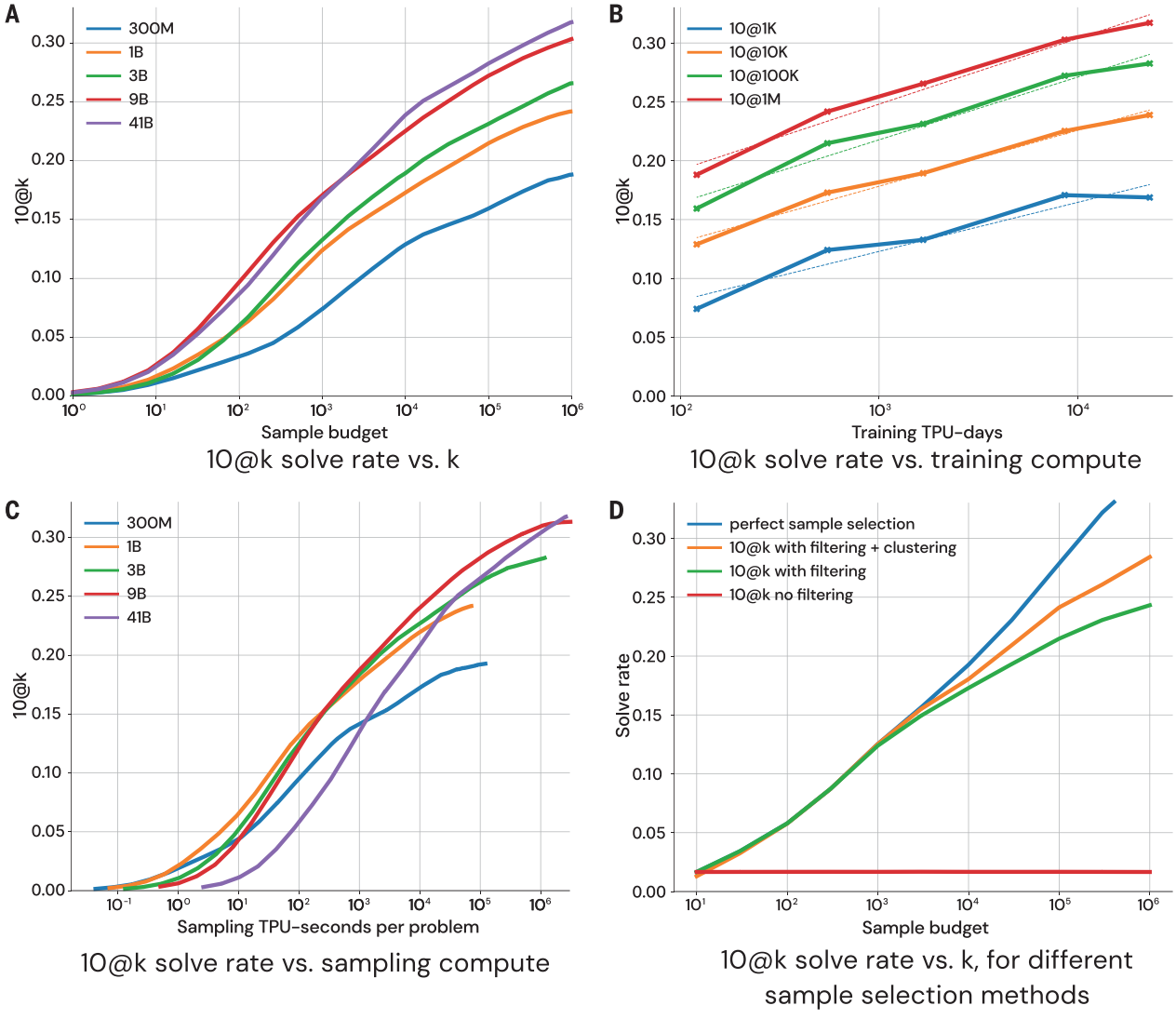

Fig. 3. Solve rate scaling. (**A**) The solve rate scaled approximately log-linearly with the number of samples, tapering off slightly in the 10@*k* setting. Larger models with more parameters had higher scaling slopes in this log-linear plot. (**B**) The solve rate scaled approximately log-linearly with the training compute when we choose near-optimal model sizes for each compute allocation. (**C**) As we increased the amount of compute we used for sampling, the optimal model size increased. (**D**) Improved sample selection methods were necessary to see solve rate scaling. In order of quality: random selection ("10@*k* no filtering"), filtering using example tests ("10@*k* with filtering"), clustering after filtering ("10@*k* with filtering + clustering"), and perfect sample selection. TPU, tensor processing unit.

budget. Conversely, alterations that fundamentally left the problem unchanged affected solve rates less, indicating that AlphaCode could mostly ignore changes that humans could also ignore. For example, introducing 30 adjacent character transposition typos decreased the 10@1024 solve rate from 13.5% to 11.3%. More analysis can be found in SM text section F.

Conclusion

Here, we present AlphaCode, a system applied to code generation for competitive programming that can generate novel solutions to unseen programming problems. When evaluated on Codeforces, AlphaCode performed roughly at the level of the median competitor. We found that massively scaling up sampling and then filtering and clustering samples to a small set, together with efficient transformer architectures to support large-scale sampling, were essential to achieving good performance. Our clean dataset and robust evaluation procedure [released as CodeContests (*22*)] also contributed substantially to guiding our research progress. We show through detailed analysis that there is no evidence that AlphaCode copied important parts of previous solutions or exploited weaknesses in the problem structure. The analysis indicates that our model was indeed able to solve problems it has never seen before, even though those problems require considerable reasoning.

Combined with careful research and engineering, the relatively simple architecture of transformers shows exceptional potential in performing reasoning necessary to solve complex problems with code. This line of work has exciting applications that can improve the productivity of programmers and make programming accessible to a new generation of developers. Our work in code generation still leaves much room for improvement, and we hope that further work will advance the fields of both programming and reasoning in AI.

REFERENCES AND NOTES

1. Z. Manna, R. J. Waldinger, *Commun. ACM* **14**, 151–165 (1971).
2. A. Church, *J. Symb. Log.* **28**, 289–290 (1963).
3. C. C. Green, in *Readings in Artificial Intelligence*, B. L. Webber, N. J. Nilsson, Eds. (Elsevier, 1981), pp. 202–222.
4. N. D. Matsakis, F. S. Klock II, *ACM SIGAda Ada Lett.* **34**, 103–104 (2014).
5. M. Resnick *et al.*, *Commun. ACM* **52**, 60–67 (2009).
6. S. Gulwani, *SIGPLAN Not.* **46**, 317–330 (2011).

7. V. Raychev, M. Vechev, E. Yahav, . *SIGPLAN Not.* **49**, 419–428 (2014).
8. M. Bruch, M. Monperrus, M. Mezini, "Learning from examples to improve code completion systems," *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '09)*, Amsterdam, Netherlands, 24 to 28 August 2009 (Association for Computing Machinery, 2009), pp. 213–222.
9. A. Hindle, E. T. Barr, Z. Su, M. Gabel, P. Devanbu, "On the naturalness of software," *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, Zurich, Switzerland, 2 to 9 June 2012 (IEEE Press, 2012), pp. 837–847.
10. A. Svyatkovskiy, S. K. Deng, S. Fu, N. Sundaresan, "IntelliCode compose: code generation using transformer," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering(ESEC/FSE '20)*, 8 to 13 November 2020 (Association for Computing Machinery, 2020), pp. 1433–1443.
11. A. Vaswani *et al*., *Adv. Neural Inf. Process. Syst.* **30**, 5998–6008 (2017).
12. T. B. Brown *et al*., arXiv:2005.14165 [cs.CL] (2020).
13. M. Chen *et al*., arXiv:2107.03374 [cs.LG] (2021).
14. J. Austin *et al*., arXiv:2108.07732 [cs.PL] (2021).
15. International Collegiate Programming Contest (ICPC), https://icpc.global/.
16. International Olympiad in Informatics (IOI), https://ioinformatics.org/.
17. D. Hendrycks *et al*., arXiv:2105.09938 [cs.SE] (2021).
18. Codeforces, https://codeforces.com/.
19. I. Sutskever, O. Vinyals, Q. V. Le, *Adv. Neural Inf. Process. Syst.* **27**, 3104–3112 (2014).
20. T. Kudo, J. Richardson, arXiv:1808.06226 [cs.CL] (2018).
21. The public GitHub dataset is hosted on Google BigQuery at https://console.cloud.google.com/marketplace/product/github/github-repos.
22. The released CodeContests dataset can be found at https://github.com/deepmind/code_contests and at https://doi.org/10.5281/zenodo.6975437.
23. N. Shazeer, arXiv:1911.02150 [cs.NE] (2019).
24. J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (Association for Computational Linguistics, 2019), pp. 4171–4186.
25. R. Dabre, A. Fujita, arXiv:2009.09372 [cs.CL] (2020).
26. R. Y. Pang, H. He, arXiv:2009.07839 [cs.CL] (2020).
27. O. Vinyals *et al*., *Nature* **575**, 350–354 (2019).
28. For problems permitting multiple correct outputs, we changed the example test outputs to be the most canonical, which gives our approach a slight advantage in the evaluation. See SM text section D for more details.
29. US Energy Information Administration, Electric Sales, Revenue, and Average Price: Summary Table T5.a: 2021 Residential Average Monthly Bill by Census Division, and State; https://www.eia.gov/electricity/sales_revenue_price/.
30. A. Ziegler, "GitHub Copilot research recitation: GitHub Copilot: Parrot or Crow? A first look at rote learning in GitHub Copilot suggestions," The GitHub Blog, 30 June 2021; https://docs.github.com/en/github/copilot/research-recitation.
31. N. Carlini *et al*., "Extracting training data from large language models," *30th USENIX Security Symposium (USENIX Security 21)*, 11 to 13 August 2021, pp. 2633–2650.
32. AlphaCode examples, explanations, and visualizations can be found at https://alphacode.deepmind.com and at https://doi.org/10.5281/zenodo.6975437.
33. Y. Li *et al*., AlphaCode data materials, version 1.0.0, Zenodo (2022); https://doi.org/10.5281/zenodo.6975437.

ACKNOWLEDGMENTS

We thank T. Cai, J. Rae, S. Borgeaud, M. Glaese, R. Ring, L. Sifre, J. Hoffman, J. Aslanides, J.-B. Lespiau, A. Mensch, E. Elsen, G. van den Driessche, and G. Irving for developing tools that we use to train large language models and for lending their expertise in model training; K. Anderson, C. Pope, and R. Foley for project management in early stages; Y. Whye Teh, C. Dyer, D. Silver, A. Barekatain, A. Zhernov, M. Overlan, and P. Veličković for research advice and assistance; K. Simonyan, C. Dyer, and D. Yogatama for reviewing the paper; L. Bennett, K. Ayoub, and J. Stanway for logistically making the project possible; S. Dathathri for analyzing our model; E. Caballero for granting permission to use Description2Code data; R. Ke for helping connect us with E. Caballero; P. Heiber for helping connect us with Codeforces; P. Mitrichev for helping connect us with Codeforces, and lending competitive programming expertise as the paper was being written; M. Mirzayanov for allowing us to evaluate on Codeforces and for lending competitive programming expertise when writing the paper; and everyone at DeepMind for their insight and support. **Funding:** All research in this paper was funded by DeepMind and Alphabet. There was no external funding. **Author contributions:** Y.L. and D.C. led the project; C.d.M.d'A., D.J.M., D.C., F.G., J.K., J.S., J.C., N.K., P.C., R.L., T.H., T.E., X.C., and Y.L. designed and implemented the learning system and algorithm; J.C. and N.K. ran the final experiments; A.D.L., D.C., F.G., J.K., J.M., J.S., J.C., N.K., P.C., R.L., T.H., T.E., and Y.L. worked on infrastructure for running and evaluating experiments; A.C., C.d.M.d'A., J.W., N.K., P.C., P.-S.H., R.L., S.G., and T.E. worked on model analysis; A.D.L., D.C., F.G., J.S., J.C., N.K., R.L., T.E., and Y.L. worked on datasets; D.C., I.B., J.C., N.K., and Y.L. worked on initial prototypes and infrastructure; D.C., J.K., J.S., J.C., N.d.F., N.K., O.V., P.C., R.L., T.E., and Y.L. wrote the paper; D.C., E.S.R., O.V., and Y.L. worked on project management; and N.d.F., O.V., K.K., and P.K. advised the project. **Competing interests:** This work was done in the course of employment at DeepMind, with no other competing financial interests. DeepMind has filed the following patent application related to this work: US63/306,043. **Data and materials availability:** The datasets used in the experiments as well as evaluation code have been made available for download on GitHub at https://github.com/deepmind/code_contests. Visualizations of the AlphaCode model, selected samples, and explanations can be found at https://alphacode.deepmind.com. Both of the above, as well as results of manual marking, can also be found in Zenodo (*33*). All other data needed to evaluate the conclusions in the paper are present in the paper or the supplementary materials.

SUPPLEMENTARY MATERIALS

science.org/doi/10.1126/science.abq1158
Supplementary Text
Figs. S1 to S29
Tables S1 to S23
References (*34*–*88*)

Submitted 29 April 2022; accepted 4 October 2022
10.1126/science.abq1158