Not All Programming Tasks Are Made Equal: Routing LLMs for Code Generation

TJ Bai

Johns Hopkins University

tbai4@jhu.edu

Abstract

We adapt recent advances in LLM routing to the specific challenge of code generation. Our lightweight router, built on a frozen 44M parameter encoder-only language model, learns to predict which tasks can be handled by a smaller model without significant quality loss. We evaluate our approach on the MBPP programming benchmark using various pairings of open-source LLMs (1B to 70B parameters). Our results demonstrate that thoughtful routing can sometimes outperform indiscriminate use of large models, though the quality of training data and signal is important. Code is available at github.com/tjbai/llmr.

1 Introduction

Large language models (LLMs) have demonstrated remarkable capabilities in a range of language processing tasks and are increasingly integrated into human-AI collaborative systems. The rampant success of instruction-tuned chat models () provides strong evidence of LLMs' potential to accelerate human productivity. Code generation, in particular, represents an especially important domain. In contrast to *natural* language where success can be subjectively defined, code must compile, pass test cases, and produce desired downstream outcomes.

The software development industry has already rapidly adopted LLMs in human-AI systems through "copilot" tools (GitHub, 2024), AI-native development environments (Anysphere, 2024), and autonomous development agents (Cognition, 2024). As these applications become more ubiquitous, a key challenge emerges: how to meet the computational demands of the largest models while managing user experience and cost.

While the largest models can handle complex challenges, inference often requires either expensive specialized hardware or introduces latency through network calls. Meanwhile, simpler tasks can often be handled by lightweight models with minimal performance degradation. This observation motivates the design of LLM *routers* that can intelligently assign tasks between agents of varying capabilities and cost, including human developers. While our work focuses on routing between LLMs of varying sizes, it extends naturally to the human-in-the-loop setting.

Through targeted experiments on a state-of-the-art family of LLMs ranging from 1B to 70B parameters in size, we find compelling results in routing for code generation. Most notably, we obtain a statistically significant 7.3% improvement in system performance when 75% of requests are routed to an 8B model, compared to using a 70B model exclusively. While we also report our fair share of negative results, such findings have important implications for deployed human-AI systems regardless - bigger is *not* always better and care *can* be taken to balance capabilities with cost.

Preprint. Under review.

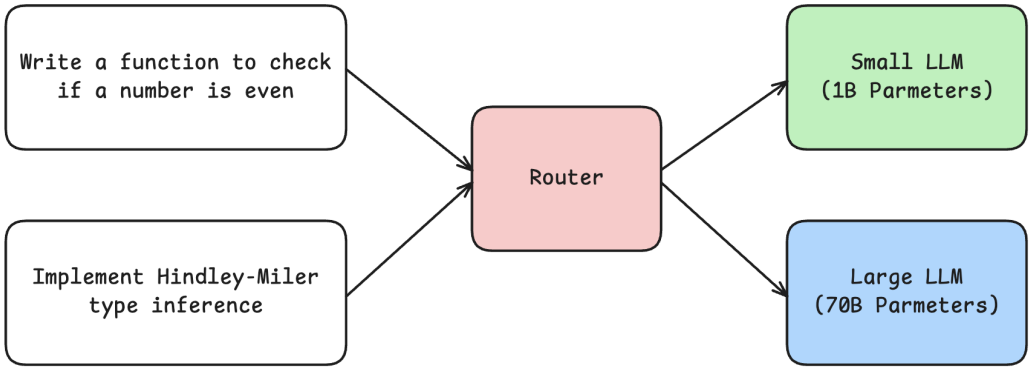


Figure 1: System architecture for our LLM routing framework.

2 Related Work

Our work builds directly on Ding et al. (2024), who introduce a hybrid inference framework using routing between LLMs of varying sizes. While they focus on general language tasks, we adapt their method specifically for code generation where test case performance provides clear evaluation metrics. The ensuing work draws on many details of their router design.

The broader context of our work intersects with several active research directions in efficient LLM deployment. While approaches like model compression, quantization, and knowledge distillation (Hinton et al., 2015; Dettmers et al., 2022; Sanh et al., 2020) aim to reduce computational costs through static optimizations, such fixed-model solutions may not suffice when serving diverse tasks with varying complexity requirements. Other work using multiple LLMs either relies on expensive model ensembles or sequential cascades (Chen et al., 2023; Wortsman et al., 2022). In contrast, our routing approach aims to minimize computational overhead by making exactly one model call per query.

3 Methodology

3.1 Problem Formulation

Consider a system that receives programming tasks $x \in X$ (e.g., "Write a function to compute the nth Fibonacci number") and produces solutions $z \in Z$ (e.g., valid Python functions). We assume access to two models:

- $S : X \to Z$, a small model with low compute cost but potentially inferior performance
- $L : X \to Z$, a larger model with higher cost but generally superior performance

To evaluate solution *quality*, we define a function $q : Z \to \mathbb{R}$. This quality function might encapsulate properties such as test case performance, code style, and solution brevity. For a given task x, we measure the quality gap between models as $H(x) = q(S(x)) - q(L(x))$.

When $H(x)$ is close to 0 or positive, the small model performs comparably to the large model, suggesting x is an "easy" task. Conversely, a large negative $H(x)$ indicates the task requires the large model's capabilities. Thus, a natural routing strategy is to model $P(H(x) \geq 0)$, in other words, the probability that S will perform at least as well as L. For a threshold τ, our router's decision rule is as follows:

$$\text{Router}(x) = \begin{cases} S(x) & \text{if } P(H(x) \geq 0) \geq \tau \\ L(x) & \text{otherwise} \end{cases} \tag{1}$$

The threshold τ controls our performance-efficiency trade-off at test time. A large τ ensures conservative routing, delegating to the small model only when the router is highly confident, while a

small τ favors the reliability of the larger model. From this perspective, the τ plays a similar role to deferral cost in learning-to-defer systems.

3.2 Training Objective

To train our model $p_w(x) = P(H(x) \geq 0)$, we can collect a data set $\{x_i, q(S(x_i)), q(L(x_i))\}_{i=1}^N$ and consider the binary cross-entropy loss with $0/1$ labels:

$$\mathcal{L}(w) = -\frac{1}{N}\sum_{i=1}^{N}\left[I_{q(S(x_i))\geq q(L(x_i))}p_w(x_i) + (1 - I_{q(S(x_i))\geq q(L(x_i))})(1 - p_w(x_i))\right] \tag{2}$$

If we instead sample M solutions for each task, noting that model outputs may be non-deterministic if sampling-based decoding schemes are used, then we can obtain a lower variance objective. For an individual task x_i, the Monte Carlo estimate of $P(H(x_i) \geq 0)$ is $y_i = \frac{1}{M}\sum_{j=1}^{M} I_{q(S(x_i)_j - Q(L(x_i)_j)}$. We can then substitute these "soft" labels into our objective:

$$\mathcal{L}(w) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i p_w(x_i) + (1 - y_i)(1 - p_w(x_i))\right] \tag{3}$$

Ding et al. (2024) notes that this objective can be problematic in practice, as we are prone to collecting a highly imbalanced dataset, due to the general inferiority of the small model to the large model. Indeed, we observe this in Figure 2. Instead, they propose modeling the shifted distribution $P(H(x_i) \geq t)$, for some $t \in \mathbb{R}$. Note that we can always tune our decision threshold τ to "counteract" this shift at test time - the augmentation only serves to provide an improved signal at train time. The shift parameter t can be computed under the heuristic that training labels should be spread as far apart as possible:

$$t^* = \text{argmax}\frac{1}{N^2}\sum_{i\neq j}|y_i(t) - y_j(t)| \tag{4}$$

4 Experiments

4.1 Setup

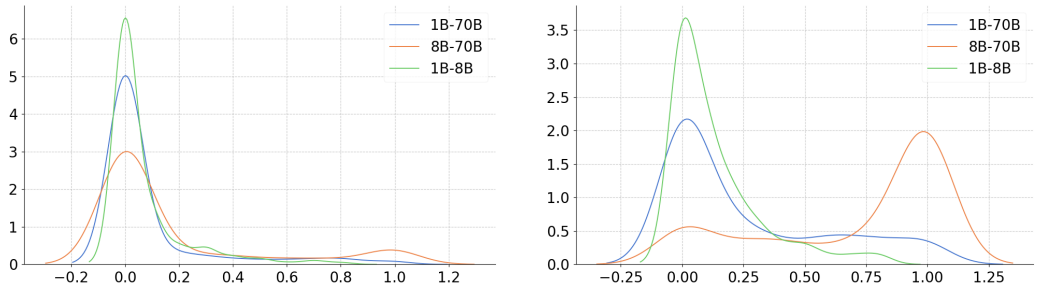
Router Design: To ensure that our routing system provides genuine efficiency gains, the router itself must be lightweight. We use DeBERTa-v3-small (He et al., 2023) as our backbone architecture, a 44M parameter model requiring less than 100MB of memory for bfloat16 inference. We train a simple 2-layer feedforward classification head (hidden dim $= 384$) with dropout regularization ($p = 0.2$).

Evaluation: We assess system performance using two key metrics. The total pass rate (TPR) measures the percentage of test cases passed at test time, while cost advantage tracks the percentage of requests routed to the smaller model. A 0% cost advantage corresponds to all-large inference, while 100% indicates all-small inference.

As baselines, we evaluate TPR at varying cost advantage percentages using a random router. We present results in terms of the performance gap - the difference between TPR at a given cost advantage and all-large inference. Intuitively, we expect this gap to widen (become more negative) as cost advantage increases, reflecting degraded performance when more requests route to the smaller model. To achieve specific cost advantage targets, we calibrate routing thresholds using empirical percentiles from the validation set.

Dataset and LLMs: We evaluate using the Mostly Basic Python Programming (MBPP) dataset (Austin et al., 2021) containing 374 training tasks and 500 test tasks, each with 3 test cases. We use the 1B, 8B, and 70B parameter models from the Llama 3 family (et al., 2024) and evaluate all possible pairings (1B-8B, 1B-70B, 8B-70B). Solutions are decoded with temperature $t=0.7$, sampling 10 times per training task for soft labels.

Although smaller than data sets in prior work, MBPP provides a realistic setting where extensive training data may be impractical - for instance, in personalized human-AI systems. To address



(a) Unilaterally giving large models "the benefit of the doubt" centers the density around 0.

(b) Prioritizing concise solutions rewards smaller models more often relative to the large model.

Figure 2: KDE-smoothed distributions of small model winrates. (a) shows results with the standard quality function while (b) the brevity-aware quality function

this data limitation, we augment our dataset by a factor of 4 using the `nlpaug` Python package. Training runs for 5 epochs, with model weights computed via exponential moving average (decay rate $\alpha = 0.999$). Empirically, these regularization measures yield measurable improvements in validation loss and Brier score.

Quality Function: For a generated solution z, we define two quality functions. First, the *standard quality function* $q(z)$ is simply the percentage of tests passed from the provided suite. This aligns with our key metric, TPR, while being easy to compute. However, this discontinuous function can be overly simplistic when the number of test cases per task is limited. For example, consider a situation where both models pass 0 out of the 3 provided test cases. While the quality gap is 0 for this task, thus favoring the small model according to Equation 3, at test time we might instead prefer to route this task to the larger, more capable model. So, in practice, we modify our objective to model the strict inequality $P(H(x) > 0)$.

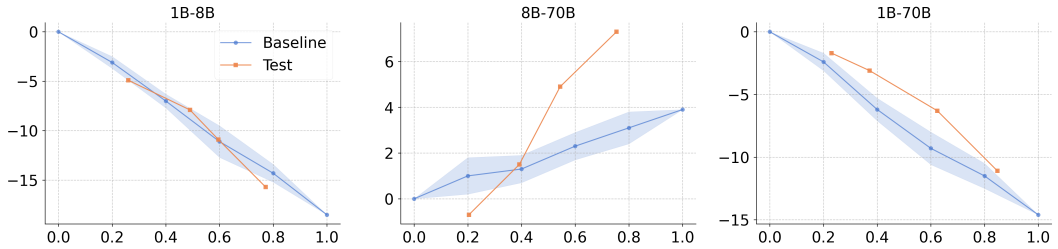
With so few test cases per task, many comparisons between models result in ties, providing limited signal for router training because these are all resolved in favor of the larger model. To address this, we introduce a *brevity-aware quality function* that breaks ties by favoring more concise solutions (i.e. less tokens). This choice follows Occam's Razor: when two solutions achieve the same test performance, the simpler (shorter) solution may indicate a better understanding of the underlying problem. While this is an imperfect proxy, it provides valuable signal for routing decisions when test results alone are ambiguous.

4.2 Baseline Performance

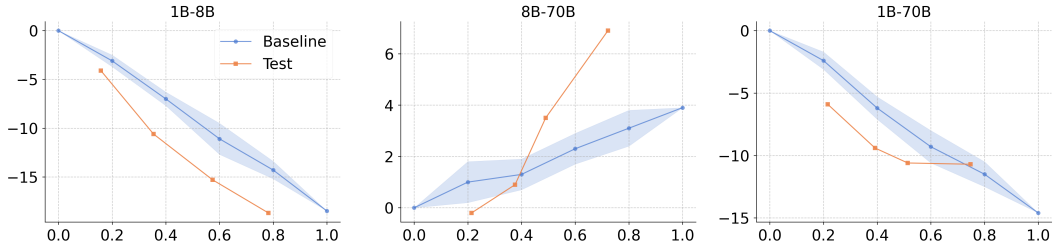
Baseline results with a random router (Table 1) challenge our prior assumption about model scaling. In the 8B-70B pairing, routing more tasks to the smaller model actually *improves* performance (+3.9% at 100% cost advantage). Given that the larger models invariably outperform on more comprehensive benchmarks, this potentially suggests that our dataset is too small/simple to differentiate model sizes. For the other pairings, as expected, increasing cost advantage degrades performance.

Pair	**Cost Advantage (%)**					
	0	20	40	60	80	100
1B-8B	63.7$_{\pm 0.0}$	60.6$_{\pm 0.6}$	56.7$_{\pm 0.7}$	52.6$_{\pm 1.6}$	49.4$_{\pm 0.9}$	45.2$_{\pm 0.0}$
8B-70B	59.8$_{\pm 0.0}$	60.8$_{\pm 0.8}$	61.1$_{\pm 0.6}$	62.1$_{\pm 0.6}$	62.9$_{\pm 0.7}$	63.7$_{\pm 0.0}$
1B-70B	59.8$_{\pm 0.0}$	57.4$_{\pm 0.7}$	53.6$_{\pm 0.9}$	50.5$_{\pm 1.3}$	48.3$_{\pm 1.0}$	45.2$_{\pm 0.0}$

Table 1: TPR with random routing. Standard deviation computed over 10 trials.



(a) Brevity-aware quality function



(b) Standard quality function

Figure 3: Comparison of performance gap in percentage points (vertical axis) and cost advantage (horizontal axis) between random baselines and learned router.

Quality	**Pair**	**Target Cost Advantage (%)**			
		20	40	60	80
Brevity-aware	1B-8B	-4.9 [26.0]	-7.9 [49.0]	-10.9 [59.6]	-15.7 [77.2]
	8B-70B	-0.7 [20.4]	+1.5 [39.2]	+4.9 [54.4]	+7.3 [75.4]
	1B-70B	-1.7 [23.0]	-3.1 [37.2]	-6.3 [62.4]	-11.1 [84.8]
Standard	1B-8B	-4.1 [15.8]	-10.6 [35.4]	-15.3 [57.4]	-18.7 [78.2]
	8B-70B	-0.2 [21.4]	+0.9 [37.6]	+3.5 [49.0]	+6.9 [72.2]
	1B-70B	-5.9 [21.6]	-9.4 [39.2]	-10.6 [51.4]	-10.7 [74.8]

Figure 4: Router performance across model pairs and quality functions. For each validation-tuned target cost advantage, we show the performance gap [actual cost advantage].

4.3 Router Performance

Our learned router shows mixed results compared to random routing baselines (Table 4, Figure 3). While we achieve modest improvements in certain settings - particularly with the 8B-70B pairing under the brevity-aware quality function - these gains are not consistent across all model combinations or routing thresholds. The router particularly struggles with the 1B-8B pairing, where it underperforms in all settings. This suggests that while our router can identify some patterns in task difficulty, its ability to make reliable routing decisions is limited.

When comparing our two quality functions, we find that the brevity-aware metric shifts the distribution of small model wins considerably, especially for the 8B model (Figure 2). This seems promising, as the brevity-aware pairings achieve superior performance gaps across the board. A brief analysis reveals that $> 50\%$ of "head-to-head" comparisons result in a tie (i.e. same number of test cases passed), with the majority of these instances coming from problems where both models produce completely incorrect solutions, thus passing 0 test cases. It appears that by introducing even slightly more training diversity with the brevity heuristic, we can materially improve system performance.

4.4 Qualitative Results

To validate whether the router is correctly assigning "easy" tasks to the small model and "hard" tasks to the large model, we manually evaluate the 5 lowest-confidence and highest-confidence tasks for each pairing, with samples provided in Appendix A. A complete record of model routing decisions on the test set can be found at github.com/tjbai/llmr.

Several clear patterns emerge: First, tasks requiring more complex data structure manipulation (nested dictionaries, tuple operations) or multi-step algorithms (heap-based sorting) consistently appear among the lowest-confidence tasks across all model pairings. In contrast, highest-confidence tasks often involve straightforward mathematical computations (finding tetrahedral numbers, calculating circle diameters) or simple numerical operations (finding quotients, minimum values). This suggests that task complexity indeed plays a role in model confidence and downstream routing decisions.

The impact of our quality function is also evident. Under the standard quality function, confidence scores are restricted to a relatively narrow band (e.g., 0.640-0.678 for 1B-8B), suggesting limited task differentiation. The brevity-aware quality function produces more pronounced confidence gaps, particularly for the 8B-70B pairing where scores range from 0.210 to 0.226, aligning with our intuition that more effective task differentiation would result in more intelligent routing decisions. Figure 2 also indicates that the dataset is slightly more balanced under the brevity-aware quality function, which would play a role in this gap.

5 Discussion

5.1 Strengths and Limitations

Our experiments paint a mixed picture of LLM routing for code generation. Most notably, we find that the relationship between model size and performance is more nuanced than previously assumed. The surprising effectiveness of the 8B model compared to its 70B counterpart - achieving up to 7.3% better performance while handling 75% of requests - challenges the "bigger is better" paradigm. This finding has immediate practical implications: identifying tasks where models with an order-of-magnitude fewer parameters excel could yield substantial efficiency gains in real-world deployments.

However, our router's inconsistent performance across different model pairings reveals important limitations. While we achieve modest improvements over random routing in specific settings (particularly the 8B-70B pair), the gains are not robust across all configurations. Several factors likely contribute to these limitations. First, our training data consists of only 374 tasks, potentially insufficient for learning reliable routing patterns. Second, with just three test cases per task, we often lack the granularity to meaningfully distinguish model performance. Third, our quality metrics may be too simplistic - while the brevity-aware function provides additional signal for routing decisions, it doesn't necessarily align with real-world code quality assessments.

5.2 Future Work

Our findings suggest several promising directions for future research. Foremost, improving the granularity and number of test cases or even identifying a richer training signal for code generation could provide more reliable routing decisions and better differentiate between models. In addition, investigating whether our results scale to larger, more-popular model pairs (>70B parameters) would be crucial to understanding routers' feasibility in practice. Exploring more sophisticated routing architectures and robust engineering tricks in the limited data regime, while maintaining practical inference costs, would further help enable reliable routing in real-world settings. These insights also naturally extend beyond model-to-model routing to better inform when and how to engage human developers in the loop, so future work could explore this connection more directly.

6 Conclusion

Our work provides the important insight that the largest models are not always necessary for achieving optimal performance. While our results show mixed success, the significant success seen in some configurations suggests that further improvements to router reliability could play a valuable role in

practical code generation systems. As these systems continue to evolve and integrate more deeply into software development, our work takes a step towards the increasingly critical goal of finding the right balance between model capability and cost.

References

Anysphere. 2024. Cursor - The AI Code Editor.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Lingjiao Chen, Matei Zaharia, and James Zou. 2023. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. ArXiv:2305.05176 [cs].

Cognition. 2024. Cognition.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. ArXiv:2208.07339 [cs].

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. 2024. Hybrid LLM: Cost-efficient and quality-aware query routing. In *The Twelfth International Conference on Learning Representations*.

Abhimanyu Dubey et al. 2024. The llama 3 herd of models.

GitHub. 2024. GitHub Copilot · Your AI pair programmer.

Pengcheng He, Jianfeng Gao, and Weizhu Chen. 2023. DeBERTaV3: Improving DeBERTa using ELECTRA-Style Pre-Training with Gradient-Disentangled Embedding Sharing. ArXiv:2111.09543 [cs].

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. ArXiv:1503.02531 [stat].

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. ArXiv:1910.01108 [cs].

Mitchell Wortsman, Gabriel Ilharco, Samir Yitzhak Gadre, Rebecca Roelofs, Raphael Gontijo-Lopes, Ari S. Morcos, Hongseok Namkoong, Ali Farhadi, Yair Carmon, Simon Kornblith, and Ludwig Schmidt. 2022. Model soups: averaging weights of multiple fine-tuned models improves accuracy without increasing inference time. ArXiv:2203.05482 [cs].

A Sample Assignments

A.1 Standard Quality Function

1B-8B

Lowest Confidence:

0.640 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm

0.642 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.645 Write a function to trim each tuple by k in the given tuple list

0.647 Write a function to perform index wise addition of tuple elements in the given two nested tuples

0.647 Write a function to maximize the given two tuples

Highest Confidence:

0.678 Write a function to find the nth tetrahedral number

0.678 Write a function to find the diameter of a circle

0.678 Write a function to calculate the nth pell number

0.678 Write a function to find the ascii value of a character

0.677 Write a function to find the nth octagonal number

8B-70B

Lowest Confidence:

0.842 Write a function to convert more than one list to nested dictionary

0.843 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.845 Write a function to find all possible combinations of the elements of a given list

0.846 Write a function to trim each tuple by k in the given tuple list

0.846 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm

Highest Confidence:

0.908 Write a function to find the diameter of a circle

0.908 Write a function to find the n-th number in newman conway sequence

0.907 Write a function to find the nth hexagonal number

0.907 Write a python function to find quotient of two numbers

0.907 Write a python function to find the sum of even factors of a number

1B-70B

Lowest Confidence:

0.555 Write a function to convert more than one list to nested dictionary

0.565 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.567 Write a function to find all possible combinations of the elements of a given list

0.577 Write a function to trim each tuple by k in the given tuple list

0.578 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm

Highest Confidence:

0.722 Write a function to find the diameter of a circle

0.718 Write a function to find the nth hexagonal number

0.717 Write a function to calculate area of a parallelogram

0.717 Write a function to find the n-th number in newman conway sequence

0.716 Write a python function to find quotient of two numbers

A.2 Brevity-aware Quality Function

1B-8B

Lowest Confidence:

0.701 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.704 Write a function to count the most common words in a dictionary

0.706 Write a function to reflect the run-length encoding from a list

0.708 Write a function to maximize the given two tuples

0.708 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm

Highest Confidence:

0.786 Write a function to find the n-th number in newman conway sequence

0.785 Write a python function to find the sum of common divisors of two given numbers

0.785 Write a function to find n'th smart number

0.785 Write a python function to find quotient of two numbers

0.784 Write a python function to find the minimum of two numbers

8B-70B

Lowest Confidence:

0.210 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.212 Write a function to generate a 3d array having each element as '*'

0.213 Write a function to remove lowercase substrings from a given string by using regex

0.213 Write a function to remove uppercase substrings from a given string by using regex

0.214 Write a function to find all possible combinations of the elements of a given list

Highest Confidence:

0.226 Write a python function to determine whether all the numbers are different from each other are not

0.226 Write a python function to count the occurence of all elements of list in a tuple

0.225 Write a function to concatenate all elements of the given list into a string

0.225 Write a function to sort the given array by using shell sort

0.225 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm

1B-70B

Lowest Confidence:

0.210 Write a function to calculate a grid of hexagon coordinates where function returns a list of lists containing 6 tuples of x,y point coordinates

0.212 Write a function to generate a 3d array having each element as '*'

0.213 Write a function to remove lowercase substrings from a given string by using regex

0.213 Write a function to remove uppercase substrings from a given string by using regex

0.214 Write a function to find all possible combinations of the elements of a given list

Highest Confidence:

0.226 Write a python function to determine whether all the numbers are different from each other are not

0.226 Write a python function to count the occurence of all elements of list in a tuple

0.225 Write a function to concatenate all elements of the given list into a string

0.225 Write a function to sort the given array by using shell sort

0.225 Write a function to merge multiple sorted inputs into a single sorted iterator using heap queue algorithm