Ans 1.1)

To build a MathGPT, I decided to go with a dataset which was divided into three stages with progressively increasing difficulty. This dataset design was chosen to find understand how the model performs on simple operations and what hyperparameters needs to be tweaked for extending to more complex operations. Below is information of operations in each stage

Stage1: This stage consisted of only **single-digit operations**. Problems consisting of Addition, Subtraction, Multiplication and Division of single digit numbers were added. It was made sure that subtraction does not have any negative values as well that division results are whole numbers only.

Stage2: This stage consisted of only **double-digit operations**. Problems consisting of Addition, Subtraction, Multiplication and Division of double-digit numbers were added. It was made sure that subtraction does not have any negative values as well that division results are whole numbers only. For multiplication operations in this stage, answers were up to 3-digit only.

Stage3: This stage consisted of operations consisting **parentheses**. Multiplication Problems were constructed as – (a + b) *c AND (a - b) *c. Division problems were constructed as – (a + b) /c AND (a - b) /c. Again, only whole and positive numbers were stored.

Stage1 & Stage2 datasets consisted of 40k problems each, 10k for every operation. Stage3 dataset consisted of 10k multiplication parentheses problems and 10k division parentheses problems, making a total of 100k problems in the dataset.

Out of theses, 10% problems from each set were held-out and kept in an unseen test dataset for final model validation.

Final_test_dataset description:

Total characters: 83435 Total lines: 10001 Vocabulary size: 19 Vocabulary: ["PAD", "ANS", "STOP", "LPAREN", "RPAREN","PLUS", "MINUS", "MUL", "DIV", "NUM_0","NUM_1","NUM_2","NUM_3","NUM_4", "NUM_5","NUM_6","NUM_7","NUM_8","NUM_9"]	Sample lines: 1. 5*9=45 2. 47-34=13 3. (12-9)/3=1

Ans 1.2)

Different metrics are used for training and evaluation due to the nature of the task.

Cross-Entropy Loss

As an evaluation metric, while the model was trained on different hyper parameters, the **cross-entropy loss** was monitored to check the model performance on train data and validation data. Training loss and validation loss was calculated on each evaluation interval. This loss was monitored to have a gradual decline with increasing number of iterations as the model learnt arithmetic operations and become more confident on its predictions. The difference between training and validation losses was particularly monitored to understand if the model is actually generalizing of if it is overfitting on the training data.

Exact match Accuracy

Since the model's task is to output the correct answer on the mathematical problem, an exact match of the generated output with the expected target was used to calculate the model **accuracy** on held-out test data. This made sure that even a single digit mismatch was penalized and no "close answer" was rewarded. The aim was for the model to exactly match the predicted output with the expected target.

Operation wise Accuracy

Further, to understand which operations the model was able to understand and exactly predict the answer, an operation wise accuracy was also calculated. This helped compare the model performance on multiple operations as per the given dataset.

Ans 1.3)

The model architecture design was done in mainly in two steps:

1) Pilot training for Hyperparameters Selection

The model was trained sequentially on stage1(simple), stage2(moderate) and stage3(complex) problems. Hyperparameters were tweaked to find best model that solved stage3 complexity problem. The model design was fixed at this stage.

2) Curriculum Learning

The final model was first trained and saved on stage1 dataset, these weights were reused and trained on stage2 dataset, the updated weights were again saved and reused for stage3 dataset. The final model after stage3 was used for calculating accuracy on held-out test dataset.

- Adaptations to Model Architecture

1) Tokenizer Redesign

Instead of using character level encoding, I experimented with implementing a custom symbolic tokenizer. All arithmetic components such as digits, operations, parentheses and '=' were encoded as semantic tokens. (e.g., NUM_3, PLUS, LPAREN, ANS). By using tokens as meaningful symbols, the model is encouraged to learn arithmetic rules and reasoning rather than memorizing character patterns. This choice significantly improves the learning and improves generalization to unseen expressions, especially for problems in stage3 complexity.

2) Block Size Redesign

Unlike natural language, arithmetic expressions make sense when seen altogether from start to end. Thus, to avoid expressions being cut out in between while batching, the block size was redesigned to accommodate complete expressions. This made sure that all relevant context is always available within the same block. This is redesigned block is much more aligned to the task the model is trying to solve. In blocks where small arithmetic expressions where present, a "PAD" token was added to complete the block size so that all blocks contain one arithmetic expression and are of the same size.

3) Loss Function Adaptation

The standard cross-entropy loss was modified to ignore padding tokens ("PAD") introduced for fixed-length batching. By excluding these tokens from the loss calculation, gradient updates are driven by valid symbolic answer tokens.

loss = F.cross_entropy(logits, targets,ignore_index=stoi["PAD"])

4) Argmax Distribution

Unlike natural language, arithmetic expressions have a single correct answer for a given input. Thus, the softmax sampling from the output distribution was replaced with argmax, where the most likely token is selected at each step.

idx_next = torch.argmax(logits, dim=-1, keepdim=True)

Once these adaptions were made in the model architecture, I started with pilot training to find best hyperparameters:

1) Pilot Training for Hyperparameter Selection

Model1 → Aim – Understand and Solve stage1 complexity problems.

Model	Stage1 Dataset performance	Stage2 Dataset performance
batch_size = 32 block_size = 20 max_iters = 1000 eval_interval = 200 n_embd = 128 n_head = 4 n_layer = 4 dropout = 0.1	0.837651 M parameters step 0: train loss 3.0361, val loss 3.0337, step 200: train loss 1.1256, val loss 1.1382, step 400: train loss 0.9529, val loss 0.9568, step 600: train loss 0.8291, val loss 0.8329, step 800: train loss 0.7543, val loss 0.7560, step 999: train loss 0.7212, val loss 0.7297,	0.837651 M parameters step 0: train loss 3.0076, val loss 3.0077, step 200: train loss 1.4964, val loss 1.4989, step 400: train loss 1.3977, val loss 1.3947, step 600: train loss 1.3049, val loss 1.3076, step 800: train loss 1.2490, val loss 1.2542, step 999: train loss 1.2196, val loss 1.2257,

Since stage1 dataset consisted of single digit operations only, a small batch size of 32 was chosen as the starting point. Similarly, the generated output was also of single to double digits only and so a block size of 20 was found enough as a starting point. To keep the model simple, 128 embeddings, 4 heads and 4 layers were decided. This gave us a model with 0.83M parameters. A minimal dropout rate of 0.1 was chosen for the model to generalize the data well. It was decided to slightly improve dropout if there were signs of overfitting.

As seen in the table above, this model performed well on stage1 dataset with a minimum loss of 0.72 for 1000 iterations. This shows that the model was capable of understanding the equations and predict correct answers. There was not difference between train and validation loss showing no signs of overfitting.

The same model however could not work well on stage2 dataset, where the loss did not show significant improvements even after training to 1000 iterations. This showed that the model is too small for stage2 complexity problems.

Model2 → Aim – Understand and Solve stage2 complexity problems.

Model	Stage2 Dataset performance	Stage3 Dataset performance
batch_size = 64 block_size = 32 max_iters = 2000 eval_interval = 500 n_embd = 192 n_head = 6 n_layer = 6 dropout = 0.1	2.679571 M parameters step 0: train loss 2.9527, val loss 2.9530, step 500: train loss 1.2203, val loss 1.2282, step 1000: train loss 1.1145, val loss 1.1202, step 1500: train loss 1.0361, val loss 1.0363, step 1999: train loss 0.9785, val loss 0.9842,	2.679571 M parameters step 0: train loss 2.9615, val loss 2.9631, step 500: train loss 1.2315, val loss 1.2366, step 1000: train loss 1.1313, val loss 1.1281, step 1500: train loss 1.0399, val loss 1.0445, step 1999: train loss 0.9869, val loss 0.9944,

I increased the batch size to 64 and block size to 32 since the stage2 dataset had longer problems with double to 3-digit answers. The model configuration was also increased to 192 embeddings with 6 heads and 6 layers for deeper understanding of the complex problems. Since there was no sign of overfitting, the dropout rate was kept unchanged. This new model was trained for 2000 iterations.

As you can see from the above table, compared to the first model, the new configurations were able to work much better in stage2 dataset with loss dropping just below 1.

The same model also works well with our stage3 dataset. This shows that the model is capable of understating problems with parentheses since the loss for stage3 is also around 1.

Final Model → Aim – Understand and Solve stage3 complexity problems.

Model	Stage3 Dataset performance
batch_size = 64 block_size = 32 max_iters = 5000 eval_interval = 500 n_embd = 192 n_head = 6 n_layer = 6 dropout = 0.1	2.679571 M parameters step 0: train loss 2.9615, val loss 2.9631, step 500: train loss 1.2315, val loss 1.2366, step 1000: train loss 1.1313, val loss 1.1281, step 1500: train loss 1.0399, val loss 1.0445, step 2000: train loss 0.9865, val loss 0.9941, step 2500: train loss 0.9569, val loss 0.9602, step 3000: train loss 0.9406, val loss 0.9423, step 3500: train loss 0.9157, val loss 0.9240, step 4000: train loss 0.9099, val loss 0.9075, step 4500: train loss 0.9098, val loss 0.9123, step 4999: train loss 0.8971, val loss 0.9013,

The same model was then again trained on the sateg3 dataset for more iterations. Thus, at 5000 iterations the model had a lowest loss of 0.89.

Masked Loss Experimentation

At this stage, I also experimented with a masked loss strategy.

In this, the cross-entropy loss was calculated only over the answer segment of each problem, specifically the tokens occurring after the '=' and up to the STOP token. Using this masked loss, instead of penalizing the model for predicting the LHS of the expression, we penalized only predicting the RHS which was the answer. The motivation behind this approach was to focus learning on the correctness of the generated answer, rather than on reproducing the input expression, which is already provided to the model. This method yielded a min loss of around 0.1 with no signs of overfitting.

However, this approach was not used as it prevents the model from learning the input expression. It would not understand the syntax of the arithmetic expression such operator placement and thus the model would overly depend on receiving perfectly formatted inputs.

Calculating loss over the full equation ensures that the model learns both the structure of valid arithmetic expressions and the mapping from expressions to correct outputs, leading to more robust behavior.

2) Curriculum Learning

Training on the whole dataset at once could lead to poor generalization and unstable training as the model struggles to discover fundamental arithmetic rules while simultaneously handling complex operations. For this reason, a curriculum learning strategy was employed by training the model on datasets of increasing complexity, from simple single-operation expressions (stage1) to multi-digit (stage2) and finally to fully nested expressions with parentheses (stage3). In this approach, the model was first encouraged to learn basic math, which was then reused and solve complex problems at later stages. Model weights at each stage were stored and loaded for the next stage.

Ans 1.4)

The final model after curriculum learning was evaluated for accuracy on held-out unseen dataset. A baseline that randomly picks any number of the same length was used for comparison. Below are the results.

Model Category-wise accuracy:	Random model outputs:
add: 98.80% (1980/2004) sub: 93.44% (1866/1997) mul: 71.58% (1456/2034) div: 100.00% (1965/1965) paren: 83.15% (1663/2000) Overall: **89.30%** (8930/10000) Random baseline accuracy: 5.74% (574/10000)	(16-4)/6=2 0+4=4 9-3=6 44/44=1 4*4=16 10*74=740 45*12=540 99+43=142 37-37=0 8*0=0

The baseline model gave us an accuracy of only 5.74%, while out model overall accuracy was strong of 89.30%.

Addition (98.80%)

Addition is learned almost perfectly by the model. This high performance is because of the local nature of addition, where each output digit depends primarily on the corresponding input digits and, at most, a single carry from the previous position. Such dependencies are well captured by the transformer's self-attention mechanism and autoregressive decoding. Furthermore, addition patterns are highly regular and abundant in the training data, allowing the model to generalize reliably to unseen examples.

Subtraction (93.44%)

Subtraction achieves a strong but slightly lower accuracy of 93.44%, with most errors arising from cases involving borrowing across digits. Borrowing requires maintaining an intermediate state that propagates across multiple token positions, which is more challenging for a next-token prediction model. While the transformer can attend to earlier digits, it lacks an explicit mechanism for persistent state updates, leading to occasional failures when multiple borrows are required.

Multiplication (71.58%)

Multiplication is the most challenging operation for the model, with accuracy dropping to 71.58%. Unlike addition or subtraction, multiplication requires multi-step reasoning, including the calculation of partial products and carry across digits. The model often produces numerically plausible outputs that are nevertheless incorrect, indicating reliance on learned correlations rather than execution of a true multiplication algorithm.

Division (100%)

Strangly division, achieves 100% accuracy, representing the best performance among all operation categories. This result is likely due to the restricted nature of the division dataset, which contains only exact integer divisions with limited operand ranges. Under these constraints, the model can effectively generalize simple division patterns without needing to perform complex reasoning. The result should therefore be interpreted as strong performance within a constrained domain rather than evidence of general division capability.

Parentheses (83%)

Expressions involving parentheses achieve 83.15% accuracy, reflecting partial but incomplete learning of operator precedence and expression structure. While the attention mechanism allows the model to attend to parentheses tokens, it does not consistently enforce execution order, particularly in longer expressions. Errors often occur when precedence rules conflict with left-to-right token order, indicating that the model treats expressions primarily as sequences rather than as structured trees.

Overall, the model performed well on the held-out dataset with an accuracy of 89.30%, comfortably better than the baseline model and the generated correct output on random expressions.