# DEEP LEARNING-BASED SOURCE CODE COMPLEXITY PREDICTION

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Deciding the computational complexity of algorithms is a really challenging problem even for human algorithm experts. Theoretically, the problem of deciding the computational complexity of a given program is undecidable due to the famous Halting problem. In this paper, we tackle the problem by designing a neural network that comprehends the algorithmic nature of codes and estimates the worst-case complexity. First, we construct a code dataset called the CodeComplex that consists of 4,120 Java codes submitted to programming competitions by human programmers and their complexity labels annotated by a group of algorithm experts. As far as we are aware, the CodeComplex dataset is by far the largest code dataset for the complexity prediction problem. Then, we present several baseline algorithms using the previous code understanding neural models such as CodeBERT, GraphCodeBERT, PLBART, and CodeT5. As the previous code understanding models do not work well on longer codes due to the code length limit, we propose the hierarchical Transformer architecture which takes method-level code snippets instead of whole codes and combines the method-level embeddings to the class-level embedding and ultimately to the code-level embedding. Moreover, we introduce pre-training objectives for the proposed model to induce the model to learn both the intrinsic property of the method-level codes and the relationship between the components. Lastly, we demonstrate that the proposed hierarchical architecture and pre-training objectives achieve state-of-the-art performance in terms of complexity prediction accuracy compared to the previous code understanding models.

## 1 INTRODUCTION

Worst-case computational (algorithmic) complexity indicates the longest running time $W(n)$ of an algorithm given any input of size $n$. Programmers estimate an effective upper bound on the amount of time required to complete the algorithm by analyzing its worst-case time complexity. We often describe the worst-case time complexity using the Big O notation using algebraic terms. For instance, we denote the time complexity of an algorithm that runs in constant time regardless of the size of input $n$ by $O(1)$. Similarly, we denote the linear-time or quadratic-time algorithm that requires the amount of time which is linear or quadratic in $n$ by $O(n)$ or $O(n^2)$, respectively.

While the worst-case time complexity provides us the effective indicator on the efficiency of a given algorithm or an actual implementation (a code), it is well-known that the problem of deciding the worst-case time complexity of an algorithm is undecidable (Turing, 1936). Therefore, there have been alternative tractable approaches to the problem of measuring the efficiency of an algorithm or a code using the static code analysis including the cyclomatic complexity (McCabe, 1976), afferent/efferent coupling, and the Master theorem (Bentley et al., 1980). On the other hand, many researchers also have looked at the dynamic code analysis which is basically based on the real execution of codes using many test cases in terms of analyzing complexity of codes (Burnim et al., 2009; Hutter et al., 2014; Nogueira, 2012). Dynamic code analysis is able to detect bugs and measure the execution time and space during execution time but involves steps for generating suitable test cases and actually running the code with the test cases.

With the rapid advancement of programming understanding models based on language models with large-scale code datasets, the concept of 'AI-powered (AI-assisted) programming' is inching toward

reality. Last year, OpenAI introduced GitHub Copilot powered by Codex (Askell et al., 2021) to assist human programs in integrated development environments (IDEs) by automatically generating codes from the context of the programming environment. More recently, DeepMind introduced AlphaCode (Li et al., 2022) that writes algorithmic programs from natural language specifications of logical problems from competitive programming. However, it has been also confirmed from recent studies (Austin et al., 2021; Jain et al., 2021) that the research on 'genuine' program understanding AI is still at early stage. If AI can really understands programs as human programmers do and possibly replace them in future, then it is natural that AI can also understand how to write 'algorithmically efficient' programs rather than just memorize the code patterns seen from the training dataset.

We also expect that the problem of predicting computational complexity of a code is significant for the educational purpose. Recently, there are a growing number of online coding platforms where people just log in and write programs in online web IDE interfaces for the purpose of collaborative programming, coding interview, programming competition, and especially programming education. We can imagine that AI can help novice programming learners write efficient codes for given problems by suggesting codes implementing better algorithms than current codes and further advise them with relative efficiency of the codes written by the learners compared to average or optimal implementations.

In this paper, we propose the problem of estimating worst-case time complexity using the cutting-edge deep neural network models and learning algorithms. First of all, we introduce the novel code complexity benchmark dataset called the *CodeComplex* dataset which consists of 4,517 actual program codes submitted to competitive programming platform (Codeforces[1]) with corresponding complexity classes annotated by human algorithm experts. As far as we are aware, this is by far the largest public dataset for the code complexity prediction task especially compared to the sole existing dataset of its kind, the CoRCoD dataset (Sikka et al., 2020) with 932 Java codes.

Together with the novel benchmark, we also provide several baseline algorithms with their performances for the code complexity prediction. Our baseline algorithms include the classical machine learning algorithms using the hand-crafted features and several state-of-the-art deep learning algorithms such as CodeBERT, GraphCodeBERT, PLBART and CodeT5.

## 2 THE CODECOMPLEX DATASET

Our dataset construction process owes much to the recently released dataset called the CodeContests[2], a competitive programming dataset for machine learning by DeepMind. We constructed a dataset with the codes from the CodeContests dataset that are again sourced from the coding competition platform Codeforces. Our dataset contains 4,120 codes in seven complexity classes, where there are new 500 Java source codes annotated with each complexity class. The seven complexity classes are constant ($O(1)$), linear ($O(n)$), quadratic ($O(n^2)$), cubic ($O(n^3)$), $O(\ln n)$, $O(n \ln n)$, and NP-hard. We also re-use 317 Java codes from CoRCoD as we confirmed that they also belong to the CodeContests dataset as the other 3,803 codes during the dataset creation process.

Table 1: Statistical difference between CoRCoD and CodeComplex. Numbers in parentheses imply the codes from CoRCoD.

| Class | CoRCoD | CodeComplex |
|---|---|---|
| $O(1)$ | 143 | 533 (+ 62) |
| $O(n)$ | 382 | 472 (+ 117) |
| $O(n^2)$ | 200 | 553 (+ 48) |
| $O(n^3)$ | 0 | 579 |
| $O(\ln n)$ | 54 | 576 (+ 18) |
| $O(n \ln n)$ | 150 | 518 (+ 72) |
| NP-hard | 0 | 572 |
| **Total** | 929 | 3,803 (+ 317) |

Remark that the CoRCoD (Sikka et al., 2020) is the sole previous dataset for the code complexity prediction problem. The CoRCoD provides us with five classes of codes, $O(1)$, $O(n)$, $O(n^2)$, $O(\ln n)$, and $O(n \ln n)$. However, the dataset is not well-balanced in terms of the complexity class as seen in Table 1. Moreover, the size of the dataset is not sufficiently large for recent DL-based algorithms as there are only 929 code samples in total. We expand the dataset to seven classes from five, considering the most frequently used complexity classes in algorithmic problems and each class

[1] https://codeforces.com/

[2] https://github.com/deepmind/code_contests

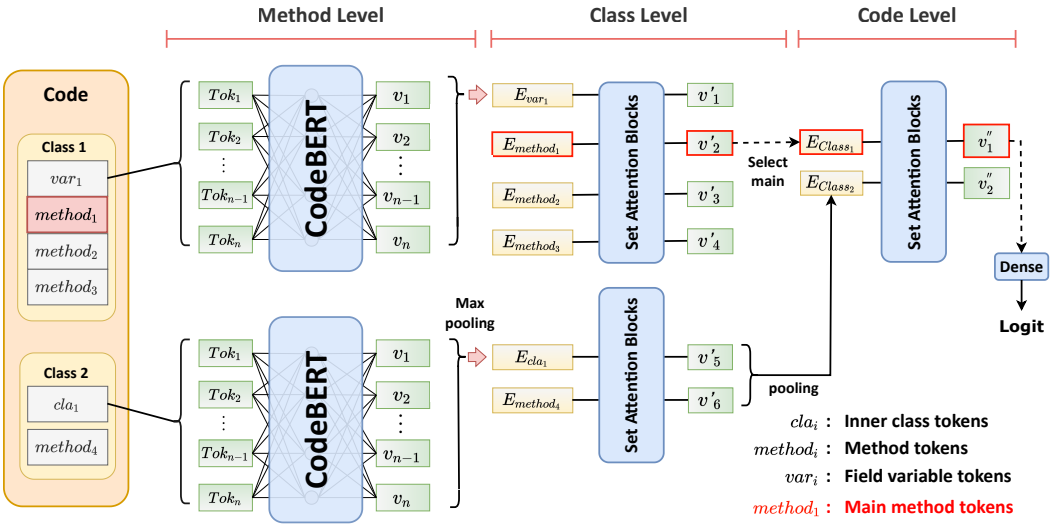



Figure 1: A graphical description of the proposed hierarchical Transformer architecture.

has at least 500 codes totaling up to 3,803 codes to boost the study in this field especially in relation to recent DL-based models.

## 3 Proposed Method

We propose a hierarchical Transformer architecture to better capture the hierarchical structure of Java codes. A Java program consists of a group of classes where each class has its own variables and methods. We can simply describe the structure of a Java code as a tree by representing the relationship between a class instance and instance members as a parent-child relationship as shown in Fig. 2. Since the previous Transformer-based language models used for understanding codes have maximum length limit (512 in CodeBERT, PLBART, CodeT5 and 256 in GraphCodeBERT), these models tend to perform worse on longer codes than the maximum length. However, it is inevitable to comprehend the entire code to analyze the complexity of the code as the complexity should be computed as a whole. As we can easily expect, a large portion of the algorithmic code dataset has codes longer than the maximum length limit. For this reason, we propose to use the pre-trained language model CodeBERT as a submodule of our hierarchical architecture so that it only deals with code snippets corresponding to a single method or a variable. Then, we aggregate the information of the entire code by processing the embeddings computed from each code snippet by the higher-level Transformer.

Together with the hierarchical architecture, we also propose *multi-level pre-training objectives* for pre-training our model. In the objectives, we aim to teach the model necessary code-related features to analyze complexity such as the nested loop depth, existence of recursion, number of parameters, call relation between methods and so on. Fig. 1 depicts our proposed hierarchical Transformer architecture.

### 3.1 Hierarchical Model Architecture

Since Java is basically an object-oriented programming language, a Java code can be interpreted in a permutation-invariant manner. For instance, the order of class or method declarations in a code does not affect its semantics. For this reason, we employ the encoder network of the Set Transformer that does not take the order of instances into account while learning the latent representations of instances.

Let $S = \{c_1, c_2, \ldots, c_{n_c}\}$ be a Java source code, where $c_i$ for $i \in [1, n_c]$ is a *class-level code snippet* for the $i$th class declaration. Note that $n_c$ is the number of classes of a code. Then, each code snippet for the $i$th class is defined as $c_i = \{m_{i,1}, m_{i,2}, \ldots, m_{i,n_{i,m}}\}$ where $m_{i,j}$ for $j \in [1, n_{i,m}]$ is

a code for the $j$th element that we call the *method-level code snippet* that could be either a method declaration, inner class declaration, field variable or so on. Note that $n_{i,m}$ is the number of method-level code snippets in the class $c_i$.

Instead of taking the entire code as input to a pre-trained programming understanding model such as CodeBERT and GraphCodeBERT, we use the method-level code snippet $m_j$ which is relatively shorter than the original code as input to the pre-trained model to obtain a latent representation $e_{m_j} \in \mathbb{R}^d$ ($d = 768$ for BERT-based models) for the snippet.

Now the problem at hand is to learn the representation of each class declaration $c_i$ that comprises information method-level code snippets from $m_1$ to $m_{n_{i,m}}$. Due to the permutation invariance of method-level code snippets, we employ the *permutation equivariant set attention blocks* from the *Set Transformer* (Lee et al., 2019) architecture. Then we take two different approaches for the class declarations depending on the existence of the `main` method as it plays the primary role in determining the computational complexity of the entire program. When there is no `main` method in the class, we apply the max pooling on the $n_{i,m}$ method-level embeddings to obtain a single class-level embedding $e_{c_i} \in \mathbb{R}^d$. Otherwise, we use the learned representation of the method-level snippet for the `main` method in the class declaration as the class-level representation.

### 3.2 Multi-Level Pre-Training Objectives

We pre-train our model in three levels: 1) token-level pre-training (MLM), 2) method-level pre-training objectives and 3) class-level pre-training objectives.

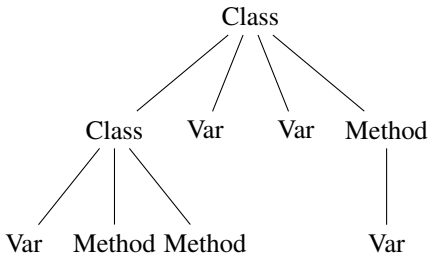



Figure 2: Example of an abstract tree representation of a Java code.

In the method-level pre-training, we further train the pre-trained programming language model such as CodeBERT, GraphCodeBERT, PLBART, and CodeT5 using our pre-training objectives for learning the better representation of codes for complexity prediction task. With MLM objective for learning token-level information, we further employ additional objectives called the *loop depth prediction*, *nubmer of parameters prediction*, *hash set existence prediction*, *hash map existence prediction*, *recursion existence prediction* and *sorting existence prediction* for learning complexity-related features of codes. We utilize MSE loss for the loop depth prediction and number of parameters prediction as the both can be any non-negative integer and binary cross-entropy loss for the next four objectives.

In the class-level pre-training, we primarily aim to pre-train the set attention blocks which takes the set of method embeddings as input and computes a single representation for a class. Here we train the set attention blocks by predicting the existence of edges between methods from a *call graph* of a program. A call graph demonstrates the control flow of a code by representing calling relationships between methods in a program. Let $G_{\mathsf{CallGraph}}(p) = (V(p), E(p))$ be a call graph of a program $p$ that consists of classes from $c_1$ to $c_{n_c}$ and each class $c_i$ consists of method-level code snippets from $m_{i,1}$ to $m_{i,n_{i,m}}$. Then, the set of nodes is defined as $V(p) = \bigcup_{i=1}^{n_c} \{m_{i,1}, \ldots, m_{i,n_{i,m}}\}$ and the set of edges is $(m_k, m_l) \in E(p)$ where $m_k, m_l \in V(p)$ and there is a reference in $m_k$ to $m_l$. Note that $\delta((m_k, m_l) \in E(p)) = 1$ if $(m_k, m_l) \in E(p)$ otherwise 0. Given two latent representations $e_{m_k}$ and $e_{m_l}$ for computed from CodeBERT (or any other submodule), we compute new representations $e'_{m_k}$ and $e'_{m_l}$ obtained from the permutation equivariant set attention blocks to augment the method-level information with the information from the other method-level code snippets. Now, let $p_{kl} = \sigma(e'_{m_k} \cdot e'_{m_l})$ be the probability of existing an edge $(m_k, m_l) \in E(p)$, which in turn implies the existence of calling relationship from $m_k$ to $m_l$ in the program $p$, where $\sigma$ is the sigmoid activation. Then, our class-level pre-training objective induced from the call graph of a program $p$ is defined as follows:

$$-\sum_{m_k, m_l \in V(p)} [\delta((m_k, m_l) \in E(p)) \log p_{kl} + (1 - \delta((m_k, m_l) \in E(p))) \log(1 - p_{kl})]. \quad (1)$$

### 3.3 DEAD CODE ELIMINATION

Since we consider codes submitted to competitive programming platform rather than well-implemented commercial codes, it is often the case that many codes contain methods and variables that are not accessed after declarations. We can remove such variables and methods while not changing the semantics of a code by statically analyzing whether or not each variable or method is accessed in the code.

Table 4 shows the performance of our model with pre-training objectives on codes where dead codes are eliminated. While it is observed that the performance gain from the model trained with pre-training objectives by using the codes after dead code elimination is marginal on average, we can also observe that the performance on longer codes $(1024, \infty]$ performance is best.

## 4 EXPERIMENTS

### 4.1 BASELINES

**ML-based (Sikka et al., 2020)** Simple ML classification algorithms such as decision tree, random forest and SVM trained with hand-crafted features such as the number of statements, variables, methods, loops, breaks, states and the existence of data structures (e.g., `HashMap` and `HashSet`) and algorithms (e.g., sorting).

**ASTNN (Zhang et al., 2019)** A neural network model that encodes the abstract syntax tree (AST) of a code. After encoding ASTs of statements with a recursive neural network called the statement encoder, it arranges statement vectors by traversing the AST in preorder and employs a bidirectional GRU for processing the sequential statement vectors.

**CodeBERT (Zhong et al., 2020)** A BERT-like pre-trained language model trained on both natural language (NL) and programming language (PL) like Python, Java, JavaScript.

**GraphCodeBERT (Guo et al., 2021)** Similar to CodeBERT but leverages *data flow* information for pre-training. Note that data flow is a graph that represents relationship between variables by analyzing where the value of each variable comes from in the entire code.

**PLBART (Ahmad et al., 2021)** A pre-trained model for program understanding and generation that uses both encoder and decoder for pre-training.

**CodeT5 (Yue Wang & Hoi, 2021)** While PLBART only treats codes simply as sequence of tokens as for NL sentences, CodeT5 relies on code-related features for pre-training such as identifier prediction and tagging.

### 4.2 EXPERIMENTAL SETUP

**Hyperparameters** For training all Transformer-based models including ours, we use the AdamW optimizer (Loshchilov & Hutter, 2019) with a learning rate scheduler using a warm-up phase with a linear decay. For fine-tuning of deep learning based code understanding models including CodeBERT, GraphCodeBERT, PLBART, and CodeT5, we used a learning rate of $2 \cdot 10^{-6}$ and weight decay of $1 \cdot 10^{-2}$. N For the proposed model, we used a learning rate of $2 \cdot 10^{-5}$ and applied different weight decay rates for the CodeBERT submodule and the set attention blocks. For the CodeBERT and set attention blocks, we used $1 \cdot 10^{-2}$ and $1 \cdot 10^{-3}$ as weight decay rate, respectively. We train all models for 15 epochs with a batch size of 6.

For the set attention blocks from the Set Transformer, we use the hidden dimension of 768, 16 inducing points, four attention heads, and four layers of attention blocks.

**Code Data Preprocessing** We first filter comments, and remove `import` and `package` statements that appear on top of codes using a regular expression as they do not affect the complexity of codes. In order to split the code into class-level and method-level code snippets while maintaining hierarchical information, we use javalang parser[3] to transform Java codes into ASTs and extract hierarchical information.

[3]https://github.com/c2nes/javalang

Table 3: Complexity prediction accuracy of classification methods for each complexity class.

| **Category** | **Method** | $O(1)$ | $O(n)$ | $O(n^2)$ | $O(n^3)$ | $O(\ln n)$ | $O(n \ln n)$ | NP-h |
|---|---|---|---|---|---|---|---|---|
| ML-based | Decision Tree | 58.7% | 15.1% | **71.4**% | 35.2% | 47.9% | 67.7% | 33.5% |
| | Random Forest | 68.1% | 18.0% | 38.4% | 25.0% | 48.6% | 68.0% | 67.9% |
| | SVM | 42.6% | 17.6% | 13.1% | 6.0% | 27.1% | 24.9% | 77.0% |
| AST-based | ASTNN | 71.4% | 26.7% | 14.2% | **49.5**% | 56.1% | 63.0% | 82.0% |
| Encoder | CodeBERT | 78.4% | 49.1% | 44.8% | 32.6% | 76.4% | 71.7% | 81.3% |
| | GraphCodeBERT | 83.0% | 40.7% | 59.3% | 11.2% | 69.5% | 73.4% | 66.0% |
| Enc-Dec | PLBART | **83.4**% | 56.5% | 45.1% | 38.2% | 75.0% | 73.1% | 88.4% |
| | CodeT5 | 77.5% | 46.0% | 29.9% | 17.3% | 75.9% | 70.2% | 83.2% |
| Ours | CodeBERT + HA | 79.3% | 44.0% | 45.1% | 32.2% | 72.2% | 77.9% | **89.7**% |
| | (+ Pretrain) | 70.0% | 57.8% | 45.6% | 39.7% | 76.5% | 78.8% | 88.1% |
| | (+ dead code elim.) | 72.9% | **61.0**% | 42.2% | 40.0% | **79.6**% | **80.0**% | 88.6% |

After then, we use the byte-pair encoding (BPE) tokenizer of RoBERTa (Liu et al., 2019) for the tokenization of the codes for the training of DL models.

**Dataset Split** We split the CodeComplex dataset in two different manners. First, we randomly split the data in 4:1 ratio. As a result, the training and test datasets contain 3,340 and 754 codes, respectively. Second, we also randomly split the data in a similar ratio but ensure that the training and test datasets have no problems in common. In other words, we randomly choose problems instead of codes and assign the entire codes for the chosen problems into the test dataset until the test dataset has around 1/5 of codes out of 4,517 codes. In case of split by problem, we perform five-fold validation and calculate the average accuracy from five iterations as the prediction accuracy tends to be highly sensitive to the choice of problems. Later, we confirm that the type of split significantly affects the performance of complexity prediction.

### 4.3 Experimental Results

Table 2 shows the experimental results of our approach with compared baselines. Note that HA is the acronym of the proposed 'hierarchical architecture'. The results show that how to split the data significantly affects the prediction performance. While the classical ML methods based on the hand-crafted features show worst performance on both random split and problem split, pre-trained models for PL exhibit much better performance on random split. This implies that the models learn the similarity of codes for the same problem instead of the operational semantics of codes for complexity prediction. In particular, our models (with/without pre-training) both achieve accuracy higher than 95% which sufficiently surpasses the performance of state-of-the-art pre-trained models.

Table 2: Complexity prediction performance on different dataset splits.

| **Method** | **Random** | **Problem** |
|---|---|---|
| Decision Tree | 53.6% | 46.9% |
| Random Forest | 61.0% | 45.3% |
| SVM | 40.7% | 24.7% |
| ASTNN | 79.3% | 51.1% |
| CodeBERT | 86.0% | 61.4% |
| GraphCodeBERT | 80.3% | 57.2% |
| PLBART | 86.3% | 63.8% |
| CodeT5 | 85.9% | 55.6% |
| CodeBERT + HA | **96.0**% | 61.3% |
| (+ Pretrain) | 95.3% | 64.2% |
| (+ Dead code) | 94.3% | **64.8**% |

**ML vs DL for Complexity Prediction** It can be readily seen that ML methods perform poorly on both splits. It should be noted that why the complexity prediction accuracy is much lower in our result than the result reported by Sikka et al. (2020). The first reason is that the CodeComplex dataset has more classes (7) than CoRCoD dataset (5). Second, we speculate that the training and test sets have soft overlap in the experiments as we confirm that there are duplicated codes in the 929 codes of the CoRCoD dataset. We find that 50 codes from 929 codes have exactly equivalent codes and two codes have 'almost equivalent' codes (which become exactly equivalent after filtering comments) within the dataset.

While ML methods do not perform well even on random split, DL models perform robustly on random split (around 90% accuracy). We can see that the performance boost compared to the case of problem split comes from the fact that DL models successfully exploit the token names from raw codes. Due to the certain amount of problem overlap in random split, DL models make use of specific token names consistently used for the same problem for predicting the same complexity class with the code seen during the training.

**Effectiveness of Pre-training Objectives** Tables 2 exhibits that our pre-training objectives are effective especially for the problem split setting. The performance gap with and without pre-training is 4.3%p for problem split while the performance slightly gets worse with pre-training for random split. Considering the fact that the performances of our model for random split are almost saturated (over 95%) with and without pre-training, we speculate that the high performance mainly relies on the similarity of code token distributions for the same problem rather than the computational similarity of codes. On the other hand, the performance boost in problem split can be regarded as more significant as it shows better understanding of semantics rather than similarity of used tokens.

**Relationship between Code Length and Accuracy** Table 4 shows the prediction performance of models on codes of different lengths. We partition the codes in the test set into four groups according to the length of sequences processed by the javalang tokenizer and calculate the prediction accuracy for each group. We can confirm the clear tendency that the prediction performance degrades as the length of code increases in every model. In order to verify the fact that a model performs better for shorter codes, we perform dead-code elimination which removes codes that do not affect the program results (unreachable or unused codes) on codes and compare the prediction accuracy. The result shows that the prediction accuracy is improved in every length group. Moreover, the result shows exactly what we expect from our hierarchical architecture design. With hierarchical architecture, the prediction performance is enhanced from vanilla CodeBERT especially in longer length group (more than 512 tokens) than in shorter length group (less than or equal to 512 tokens).

**Error Case Analysis** Table 3 and Fig. 3 show the type of errors our model more frequently makes. We can see that the model is easily confused on polynomial-time algorithms including $O(n), O(n^2)$ and $O(n^3)$. Our model makes many mispredictions for $O(n)$ codes by predicting other classes uniformly except $O(2^n)$ and is also frequently confused between quadratic and cubic algorithms.

Fig. 4 shows two failure cases where our model fails to predict the correct complexity class of codes. Fig. 4a is the case where our model predicts quadratic time complexity for the code with linear time complexity. At a glance, the code actually seems to be in $O(n^2)$ due to the nested `for` loops. However, the number of iterations is actually controlled by an integer variable `k`. Another example in Fig. 4b is also interesting. Our model predicts the complexity class as $O(n^2)$ but the actual complexity is $O(n^3)$ because the method inside the nested `for` loops named `lowestCost` runs in linear time in the size of input. From these error cases, we can deduce that our model focuses on the computational structure of a code rather than merely the token distribution of a code.

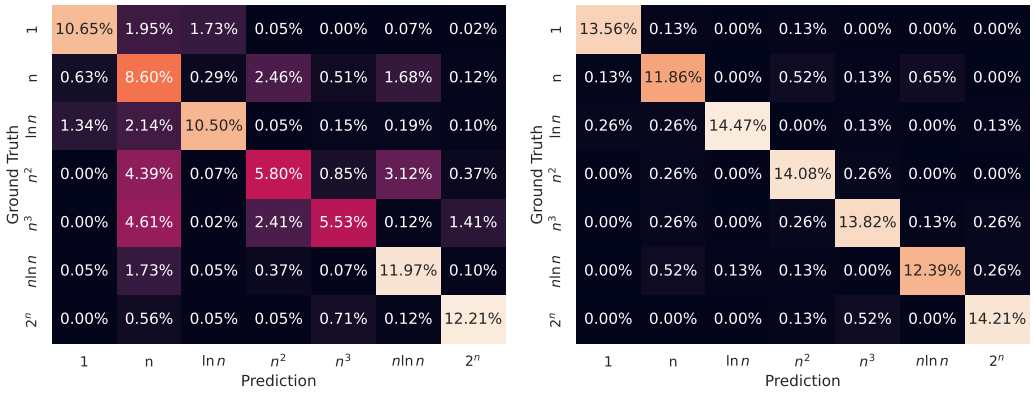
## 5 Related Work

**Analyzing Time Complexity of Programs** McCabe (1976) introduced a metric for quantitatively measuring the complexity of a program called the *cyclomatic complexity*. Intuitively, the cyclomatic complexity counts the number of linearly dependent paths. Bentley et al. (1980) presented the *Master theorem* for divide-and-conquer algorithms by describing the time complexity of an algorithm as a recurrence relation and solving the relation.

More recently, Sikka et al. (2020) studied a learning-based methods for code complexity prediction. They released a novel code dataset with 929 Java codes annotated with runtime complexities and proposed baselines of machine learning-based models with hand-engineered features. For instance, they first extracted features such as the number of loops, methods, variables, jumps, breaks, switches and the presence of special data structures or algorithms such as priority queue, hash map, hash set, and sorting functions. After then, they run the famous ML classification algorithms such as $K$-means, random forest, decision tree, SVM and so on. They also reported a similar performance by embedding the graph structure of a program's AST with a neural graph embedding framework,

Table 4: Prediction performance on different code lengths

| **Method** | **(0, 256]** | **(256, 512]** | **(512, 1024]** | **(1024, $\infty$]** |
|---|---|---|---|---|
| Decision Tree | 57.2% | 45.6% | 40.0% | 38.2% |
| Random Forest | 62.3% | 46.8% | 40.6% | 26.4% |
| SVM | 48.9% | 18.1% | 18.1% | 16.6% |
| ASTNN | 16.7% | 52.1% | 51.5% | 44.2% |
| CodeBERT | 72.4% | 62.8% | 60.7% | 48.0% |
| GraphCodeBERT | **74.6**% | 61.7% | 49.8% | 39.4% |
| PLBART | 74.3% | **65.1**% | 62.5% | 52.8% |
| CodeT5 | 69.5% | 56.5% | 52.4% | 42.4% |
| CodeBERT + HA | 70.6% | 60.7% | 62.3% | 51.3% |
| CodeBERT + HA + Pretrain | 70.3% | 63.6% | **65.9**% | 57.8% |
| CodeBERT + HA + Pretrain + Dead code elim. | 71.7% | 63.5% | 64.5% | **60.1**% |




(a) On problem split

(b) On random split

Figure 3: Confusion matrices for the predictions of the proposed model.

graph2vec (Narayanan et al., 2017). More recently, Prenner & Robbes (2021) examined the performance of the pre-trained programming language understanding model such as CodeBERT (Feng et al., 2020) for code complexity prediction and showed that the pre-trained model can be a promising alternative for the problem.

**Programming Language Understanding Models** There have been numerous studies on pre-training methods for understanding programming languages. Feng et al. (2020) proposed CodeBERT, which is RoBERTa-based model pre-trained on multiple programming languages with masked language modeling (MLM) and replaced token prediction (RTD) objectives. Guo et al. (2021) introduced GraphCodeBERT which is strengthened from CodeBERT by incorporating data flow information in the pre-training stage. Jiang et al. (2021) introduced TreeBERT, a tree-based pre-trained model that focuses on utilizing the extracted tree structure by encoding an AST as a set of composition paths. TreeBERT is trained by two novel objectives called tree masked language modeling (TMLM) and node order prediction (NOP). Rozière et al. (2021) investigated another programming langauge-oriented pre-training objective called DOBF, which is based on deobfuscation of identifier names in source code. Note that we do not use TreeBERT and DOBF as our baseline as they are mainly for code generation tasks not for classification task.

Recently, Ahmad et al. (2021) proposed PLBART (Program and Language BART), which learns the interaction between program codes and natural language descriptions by leveraging the idea of denoising autoencoder that uses a bidirectional encoder and an auto-regressing decoder. Yue Wang & Hoi (2021) introduced CodeT5, which leverages the code-specific characteristics in the pre-training

```
int n = nextInt();
int k = nextInt();
int[] a = new int[n];
for (int i = 0; i < n; i++) {
   a[i] = nextInt();
}
Set<Integer> set = new
    HashSet<Integer>();
for (int i = 0; i < a.length;
    i++) {
   set.add(a[i]);
   if (set.size() == k) {
      Set<Integer> set2 = new
          HashSet<Integer>();
      for (int j = i; j >= 0;
          j--) {
         set2.add(a[j]);
         if (set2.size() == k) {
            out.print((j + 1) +
                " " + (i + 1));
            out.close();
            return;
         }
      }
   }
}
```

(a) A code snippet from a program whose complexity is predicted as $O(n^2)$ by our model while the actual complexity is in $O(n)$.

```
public static void main(String[]
    args){
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < m;
          j++) {
         if (steps % 2 != 0) {
            out.print(-1 + " ");
         } else {
            out.print(2 *
                lowestCost(i, j,
                steps / 2) + " ");
         }
      }
   }
}
private long lowestCost(int i,
    int j, int distance) {
   long minDist = Long.MAX_VALUE;
   if (i > 0)
      minDist =
          Math.min(minDist,
          distI[i - 1][j] +
          lowestCost(i - 1, j,
          distance - 1));
   ...
   return minDist;
}
```

(b) A code snippet from a program whose complexity is predicted as $O(n^2)$ by our model while the actual complexity is in $O(n^3)$.

Figure 4: Failure examples from most frequent cases of mispredictions discovered from confusion matrix.

stage by employing the new objectives such as masked random token prediction, masked identifier prediction, and identifier prediction objectives.

## 6 Conclusions

In the light of recent studies, we have proposed to employ pre-trained models for the task of predicting algorithmic complexity of codes. In order to utilize the entire code information, we have suggested a hierarchical Transformer architecture where any pre-trained program understanding models can be used at the bottom level for understanding method-level code snippets. Then, we aggregate the method-level embeddings using permutation-invariant model Set Transformer to compute the class-level embedding. Finally, we aggregate the class-level embeddings for the final complexity prediction. In order to effectively pre-training the proposed architecture, we have proposed multi-level pre-training objectives. We have demonstrated that the proposed algorithm shows the best performance compared to the previous state-of-the-art program understanding models in complexity prediction.

In future, we plan to incorporate dynamic program analysis techniques into the proposed deep learning-based approach. For instance, we can obtain runtime information of a code by actually executing the code with proper test cases. We can also manipulate the size of test cases to see how the runtime performance changes asymptotically. In this scenario, the major challenge would be how to generate proper test cases of a given program so that we can get accurate runtime estimates for the worst-case complexity analysis.

## REPRODUCIBILITY

For reproducing the results of our paper, please find `readme.md` file from our supplementary material. The supplementary material contains 3,803 Java codes in `data` directory. We also include the five-fold splits of codes by problems for verifying the generalization of our model to unseen problems. The data files with `_d` contain codes after dead code elimination and `_r` are randomly split train and test files. One can train the baseline models and our proposed model using the commands in the `readme.md` file. Note that all the necessary data files for the entire reproduction of the experimental results can be accessed from the links provided in the file.

## REFERENCES

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2655–2668, 2021.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. *CoRR*, abs/2112.00861, 2021.

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. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021.

Jon Louis Bentley, Dorothea Haken, and James B. Saxe. A general method for solving divide-and-conquer recurrences. *SIGACT News*, 12(3):36–44, 1980.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. WISE: automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009*, pp. 463–473. IEEE, 2009.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, volume EMNLP 2020 of *Findings of ACL*, pp. 1536–1547. Association for Computational Linguistics, 2020.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event*. OpenReview.net, 2021.

Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence*, 206:79–111, 2014.

Naman Jain, Skanda Vaidyanath, Arun Shankar Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram K. Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. *CoRR*, abs/2112.02969, 2021.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. Treebert: A tree-based pre-trained model for programming language. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021*, volume 161 of *Proceedings of Machine Learning Research*, pp. 54–63. AUAI Press, 2021.

Juho Lee, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh. Set transformer: A framework for attention-based permutation-invariant neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pp. 3744–3753. PMLR, 2019.

Yujia Li, David H. 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, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*, 2019.

T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(04):308–320, 1976.

Annamalai Narayanan, Mahinthan Chandramohan, Rajasekar Venkatesan, Lihui Chen, Yang Liu, and Shantanu Jaiswal. graph2vec: Learning distributed representations of graphs. *CoRR*, abs/1707.05005, 2017.

Ana Filipa Nogueira. Predicting software complexity by means of evolutionary testing. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12*, pp. 402–405. ACM, 2012.

Julian Aron Aron Prenner and Romain Robbes. Making the most of small software engineering datasets with modern machine learning. *IEEE Transactions on Software Engineering*, 2021.

Baptiste Rozière, Marie-Anne Lachaux, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. *CoRR*, abs/2102.07492, 2021.

Jagriti Sikka, Kushal Satya, Yaman Kumar, Shagun Uppal, Rajiv Ratn Shah, and Roger Zimmermann. Learning based methods for code runtime complexity prediction. In *Advances in Information Retrieval - 42nd European Conference on IR Research, ECIR 2020, Proceedings, Part I*, volume 12035 of *Lecture Notes in Computer Science*, pp. 313–325. Springer, 2020.

Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42):230–265, 1936.

Shafiq Joty Yue Wang, Weishi Wang and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*, pp. 8696–8708, 2021.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pp. 783–794. IEEE / ACM, 2019.

Wanjun Zhong, Duyu Tang, Zhangyin Feng, Nan Duan, Ming Zhou, Ming Gong, Linjun Shou, Daxin Jiang, Jiahai Wang, and Jian Yin. Logicalfactchecker: Leveraging logical operations for fact checking with graph module network. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020*, pp. 6053–6065. Association for Computational Linguistics, 2020.

## A Overview on CodeComplex Dataset

Our dataset construction process owes much to the recently released dataset called the CodeContests[4], a competitive programming dataset for machine learning by DeepMind. We constructed a

[4]https://github.com/deepmind/code_contests

dataset with the codes from the CodeContests dataset that are again sourced from the coding competition platform Codeforces. Our dataset contains 4,120 codes in seven complexity classes, where there are new 500 Java source codes annotated with each complexity class. The seven complexity classes are constant ($O(1)$), linear ($O(n)$), quadratic ($O(n^2)$), cubic ($O(n^3)$), $O(\ln n)$, $O(n \ln n)$, and NP-hard. We also re-use 317 Java codes from CoRCoD as we confirmed that they also belong to the CodeContests dataset as the other 3803 codes during the dataset creation process.

For constructing the dataset, we asked twelve human annotators who have more than five years of programming experience and algorithmic expertise to inspect the codes manually and classified them into one of the seven complexity classes. Once each human annotator reported the initial result, we collected the annotation results and inspected them once again by assigning the initial result to two different annotators other than the initial annotator. Finally, we have collected 3803 complexity annotated codes in which there are 500 codes for each complexity class.

First, we selected several problems that are expected to belong to one of the considered complexity classes, and submitted codes for the problems from Codeforces. The submitted codes contain both correct and incorrect solutions, and they are implemented in various programming languages such as C, C++, Java, and Python. We sorted out only the correct Java codes for our dataset construction.

In the second step, before delving into the time complexity of problems, we divide the problems by the problem-solving strategy such as sorting, DP (dynamic programming), divide-and-conquer, DFS (depth-first search), BFS (breadth-first search), A*, and so on. This is because it is helpful to know the type of problem-solving strategy used to solve the problem for human annotators to analyze the time complexity, and problems solved by the same strategy tend to have similar time complexity.

Third, we uniformly assign problems and correct codes for the problems to human annotators and let them carefully examine the problem-code pairs to label the time complexity of the codes. Notice that there can be solutions with different time complexities for a problem depending on how to actually implement the solutions. We, therefore, provide specific guideline that contains instructions and precautions to annotators so that human annotators can assign correct and consistent labels to the assigned codes.

After the initial annotation process, we collect the results and assign them to different annotators to carefully cross-check the correctness of the initial annotation results. Primarily, we instruct the annotators again to carefully verify the results in accordance with the precautions provided in the annotation guideline.

### A.1 Further Details on CodeComplex Dataset Construction

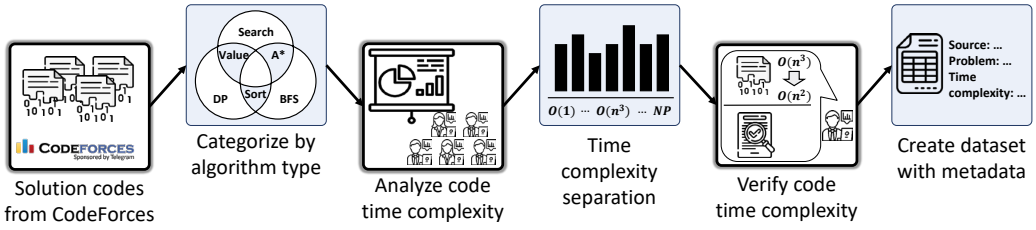



Figure 5: Overall workflow of CodeComplex dataset creation.

We gathered 128,000,000 submissions of Codeforces, where 4,086,507 codes are implemented in Java language. After discarding the incorrect codes (that do not pass all the test cases), there are 2,034,925 codes and 7,843 problems. Then the problems are split with their tags (e.g. sorting, dfs, greedy, etc) and given to the annotators with the guidelines in Section A.2. We were able to gather around 500 problems and 15,000 codes for the seven complexity classes.

As the complexity of codes for the same problem can vary depending on the implemented algorithms, it is obvious that the codes we inspect also have various complexity classes. However, we only target seven complexity classes that are the most frequently used complexity classes for algorithmic problems. Accordingly, some codes we inspect through belong to other complexity classes such as $O(n^5)$ or $O(\ln \ln n)$. We inspected around 800 problems and found out that the complexity classes of approximately 15% of the problems belong outside the chosen complexity classes. Although it is still possible that one might implement codes with complexity class that falls into the

seven complexity classes, we simply rule out the problems from our dataset to ease the annotation process.

During this process, we found out that many codes are not optimal for the given problem and some codes are too difficult to analyze due to their complex code structure. Moreover, there are many codes with a number of methods that are never used, mainly because the codes come from a coding competition platform and participants prefer just to include the methods that are frequently used in problem-solving regardless of the actual usage of the methods.

Our dataset, the CodeComplex dataset is constructed from the code instances from Codeforces platform which are recently revealed from AlphaCode. As Codeforces is a coding competition platform, our dataset consists of codes that implement various algorithms that are designed to solve algorithmically challenging problems. Our dataset offers a larger number of source codes with a broader category of complexity classes compared to the sole existing complexity prediction benchmark dataset, CoRCoD dataset (Sikka et al., 2020). In the section below, we share the detailed guideline provided to human annotators for the precise code complexity annotation process.

### A.2 GUIDELINE OF PRODUCTION

1. Check the variables that are described in the algorithm problems. Each algorithm implementation can have many variable instances and we only consider the variables that are given as inputs from the problems for calculating the time complexity.

* For convenience, we use $n$ and $m$ in the guideline to denote the input variable and $|n|$ and $|m|$ to denote the size of $n$ and $m$.

2. Considering the input variable the prior step, follow the below instruction for each input type and calculate the time complexity.
   (a) When only a number $n$ is given as an input, calculate the time complexity proportional to $n$. Do the same thing when there are two or more variables. For instance, when only $n$ is given as an input, the variable used to denote the time complexity of a code is $n$.
   (b) When a number $n$ and $m$ numeric instances are given as inputs, calculate the time complexity proportional to the one with higher complexity. For instance, when $m = n^2$, we calulate complexity of a code with $m$. If the implemented algorithm runs in $O(n^2) = O(m)$, it belongs to linear complexity class.
   (c) If the input is given as constant values, the complexity of a given code also belongs to constant class. For instance, if an algorithm problem states that exactly 3 numeric values are given as inputs, the solution code only uses constant number of operations. Therefore, the code belongs to the constant class.
3. Consider the case where the code utilizes the input constraints of the problem. When the input is given by $n \leq a$, the code can use the fixed value $a$ in the problem instead of using $n$. Mark these codes as unsuitable.
4. Consider the built-in library that the implemented algorithm is using (e.g. HashMap, sort, etc) to calculate the time complexity of an entire code. For instance, given $n$ numeric instances as inputs, when an implemented algorithm uses $O(n)$ iterations of built-in sort algorithm for $n$ numeric instances, the time complexity for the algorithm is $O(n^2 \ln n)$.
5. When the code has unreachable codes, only consider the reachable code.
6. Mark the item that does not belong to any of the 7 complexity classes.

- Listing 1 exhibits a failure example where our model predicts $O(2^n)$ for a code with $O(\ln n)$ complexity. We suspect that the primary reason is the usage of bitwise operators. When we filter the codes that use any bitwise operator at least once from our CodeComplex dataset, about 56% of the codes belong to the class $O(2^n)$, which implies NP-hardness. We find that many implementations for NP-hard problems rely on the bitwise operators as they can efficiently manage backtracking process by manipulating bit-level flags.

Listing 1: A failure example of our model (GT: $O(\ln n)$, Prediction: $O(2^n)$).

```java
public class mad {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int cura = 0, curb = 0;
      int ver;
      System.out.println("? 0 0");
      System.out.flush();
      ver = sc.nextInt();
      for (int i = 29; i >= 0; i--) {
         System.out.println("? " + (cura + (1 << i)) + " " + curb);
         System.out.flush();
         int temp1 = sc.nextInt();
         System.out.println("? " + cura + " " + (curb + (1 << i)));
         System.out.flush();
         int temp2 = sc.nextInt();
         if (temp1 != temp2) {
            if (temp2 == 1) {
               cura += (1 << i);
               curb += (1 << i);
            }
         } else {
            if (ver == 1) cura += (1 << i);
            if (ver == -1) curb += (1 << i);
            ver = temp1;
         }
      }
      System.out.println("! " + cura + " " + curb);
   }
}
```

- Listing 2 demonstrates the case when our model predicts constant time complexity $O(1)$ for a code that runs in $O(n)$ time. We suspect that our model may have ignored the existence of the `check` method which actually determines the $O(n)$ time complexity or considered the argument of `check` as constant.

Listing 2: A failure example of our model (GT: $O(n)$, Prediction: $O(1)$).

```java
public class abc {
   public static int check(StringBuilder s) {
      int countRemove = 0;
      if (!s.toString().contains("xxx")) return countRemove;
      else {
         for (int i = 1; i < s.length() - 1; i++) {
            if (s.charAt(i - 1) == 'x' && s.charAt(i) == 'x' &&
                s.charAt(i + 1) == 'x') {
               countRemove++;
            }
         }
         return countRemove;
      }
   }

   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      String s = sc.next();
      StringBuilder sb = new StringBuilder("");
      sb.append(s);
      System.out.println(check(sb));
   }
}
```

- Listing 3 shows the case where our model predicts the quadratic time complexity $O(n^2)$ when the ground-truth label is $O(n \ln n)$. We guess that our model simply translates the nested `for` loops into the quadratic time complexity. However, the outer loop is to repeat each test case and therefore should be ignored. Then, the $O(n \ln n)$ complexity can be determined by the `sort` function used right before the nested loops.

Listing 3: A failure example of our model (GT: $O(n \ln n)$, Prediction: $O(n^2)$).

```
ppublic class round111A {
   public static void main(String[] args) {
      Scanner sc = new Scanner(System.in);
      int n = sc.nextInt();
      int[] coins = new int[n];
      for (int i = 0; i < n; ++i) coins[i] = sc.nextInt();
      Arrays.sort(coins);
      int ans = (int) 1e9;
      for (int i = 1; i <= n; ++i) {
         int sum1 = 0;
         int c = 0;
         int j = n - 1;
         for (j = n - 1; j >= 0 && c < i; --j, ++c) {
            sum1 += coins[j];
         }
         int sum2 = 0;
         for (int k = 0; k <= j; ++k) sum2 += coins[k];
         if (sum1 > sum2) {
            System.out.println(i);
            return;
         }
      }
   }
}
```

- Listing 4 shows the case when our model is confused exponential complexity $O(2^n)$ with quadratic complexity $O(n^2)$. The code actually runs in exponential time in the worst-case but our model simply returns quadratic time complexity as it does not take into account the recursive nature of the method `solve`.

Listing 4: A failure example of our model (GT: $O(2^n)$, Prediction: $O(n^2)$).

```
public class D {
   static int n, KA, A;
   static int[] b;
   static int[] l;
   static double ans = 0;

   public static void main(String[] args) throws IOException {
      Scanner in = new Scanner(System.in);
      n = in.nextInt();
      KA = in.nextInt();
      A = in.nextInt();
      b = new int[n];
      l = new int[n];
      for (int i = 0; i < l.length; i++) {
         b[i] = in.nextInt();
         l[i] = in.nextInt();
      }
      dp = new double[n + 2][n + 2][n * 9999 + 2];
      go(0, KA);
      System.out.printf("%.6f\n", ans);
   }

   public static void go(int at, int k) {
```

```
        if (at == n) {
            ans = Math.max(ans, solve(0, 0, 0));
            return;
        }
        for (int i = 0; i <= k; i++) {
            if (l[at] + i * 10 <= 100) {
                l[at] += i * 10;
                go(at + 1, k - i);
                l[at] -= i * 10;
            }
        }
    }

    static double dp[][][];

    public static double solve(int at, int ok, int B) {
        if (at == n) {
            if (ok > n / 2) {
                return 1;
            } else {
                return (A * 1.0) / (A * 1.0 + B);
            }
        }
        double ret = ((l[at]) / 100.0) * solve(at + 1, ok + 1, B) + (1.0 -
            ((l[at]) / 100.0)) * solve(at + 1, ok, B + b[at]);
        return ret;
    }

}
```

- Listing 5 shows the case when our model predicts $O(\ln n)$ for a code with $O(n^2)$ complexity. It is easily seen that the `inversions` function determines the quadratic time complexity by the nested `for` loops. We suspect that somehow our model does not take into account the `inversions` function in the complexity prediction and instead focuses on the modulo (%) operator to draw the wrong conclusion that the complexity is in $O(\ln n)$.

Listing 5: A failure example of our model (GT: $O(n^2)$, Prediction: $O(\ln n)$).

```
public class maestro {
    public static long inversions(long[] arr) {
        long x = 0;
        int n = arr.length;
        for (int i = n - 2; i >= 0; i--) {
            for (int j = i + 1; j < n; j++) {
                if (arr[i] > arr[j]) {
                    x++;
                }
            }
        }
        return x;
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        long[] arr = new long[n];
        for (int i = 0; i < n; i++) arr[i] = sc.nextLong();
        long m = sc.nextLong();
        long x = inversions(arr) % 2;
        for (int i = 0; i < m; i++) {
            int l = sc.nextInt() - 1;
            int r = sc.nextInt() - 1;
            if ((r - l + 1) % 4 > 1) x = (x + 1) % 2;
```

```
            if (x == 1) System.out.println("odd");
            else System.out.println("even");
        }
    }
}
```

## B FURTHER DETAILS ON DEAD CODE ELIMINATION

In a broad sense, the dead code includes redundant code, unreachable code, oxbow code, and so on. We only focus on eliminating unreachable codes, mainly methods and classes that are declared but used nowhere in the code. In order to find such dead codes, we first parse a Java code into an AST, and discover methods and classes that do not exist in any method call, class declaration, and arguments of methods. Once we discover such unused methods and classes, we remove the codes corresponding to the declarations of these methods and classes.

Listings 6 and 7 show a running example of the dead code elimination process. From the code in Listing 6, we can obtain the code in Listing 7 by applying the dead code elimination. It is readily seen that the number of lines decreases from 211 to 101 by the elimination process. In fact, our model predicts $O(\ln n)$ and $O(1)$ for the complexity of the code before and after dead code elimination, respectively, while the actual complexity of the code is $O(1)$.

Listing 6: An example code containing many dead codes such as unused methods and variables.

```
public class Main {
    static long mod = ((long) 1e9) + 7;

    public static int gcd(int a, int b) {
        if (b == 0) return a;
        else return gcd(b, a % b);
    }

    public static long pow_mod(long x, long y) {
        long res = 1;
        x = x % mod;
        while (y > 0) {
            if ((y & 1) == 1) res = (res * x) % mod;
            y = y >> 1;
            x = (x * x) % mod;
        }
        return res;
    }

    public static int lower_bound(int[] arr, int val) {
        int lo = 0;
        int hi = arr.length - 1;
        while (lo < hi) {
            int mid = lo + ((hi - lo + 1) / 2);
            if (arr[mid] == val) {
                return mid;
            } else if (arr[mid] > val) {
                hi = mid - 1;
            } else lo = mid;
        }
        if (arr[lo] <= val) return lo;
        else return -1;
    }

    public static int upper_bound(int[] arr, int val) {
        int lo = 0;
        int hi = arr.length - 1;
        while (lo < hi) {
            int mid = lo + ((hi - lo) / 2);
```

```
            if (arr[mid] == val) {
                return mid;
            } else if (arr[mid] > val) {
                hi = mid;
                ;
            } else lo = mid + 1;
        }
        if (arr[lo] >= val) return lo;
        else return -1;
    }

    public static void main(String[] args) throws java.lang.Exception {
        Reader sn = new Reader();
        Print p = new Print();
        int n = sn.nextInt();
        while ((n--) > 0) {
            int a = sn.nextInt();
            int b = sn.nextInt();
            int small = Math.min(a, b);
            int large = Math.max(a, b);
            long steps = 0;
            while (small != 0) {
                steps += (long) (large / small);
                int large1 = small;
                small = large % small;
                large = large1;
            }
            p.printLine(Long.toString(steps));
        }
        p.close();
    }
}

class Pair implements Comparable<Pair> {
    int val;
    int in;

    Pair(int a, int b) {
        val = a;
        in = b;
    }

    @Override
    public int compareTo(Pair o) {
        if (val == o.val) return Integer.compare(in, o.in);
        else return Integer.compare(val, o.val);
    }
}

class Reader {
    final private int BUFFER_SIZE = 1 << 16;
    private DataInputStream din;
    private byte[] buffer;
    private int bufferPointer, bytesRead;

    public boolean isSpaceChar(int c) {
        return c == ' ' || c == '\n' || c == '\r' || c == '\t' || c == -1;
    }

    public Reader() {
        din = new DataInputStream(System.in);
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

```

```
    public Reader(String file_name) throws IOException {
        din = new DataInputStream(new FileInputStream(file_name));
        buffer = new byte[BUFFER_SIZE];
        bufferPointer = bytesRead = 0;
    }

    public String readLine() throws IOException {
        byte[] buf = new byte[64];
        int cnt = 0, c;
        while ((c = read()) != -1) {
            if (c == '\n') break;
            buf[cnt++] = (byte) c;
        }
        return new String(buf, 0, cnt);
    }

    public String readWord() throws IOException {
        int c = read();
        while (isSpaceChar(c)) c = read();
        StringBuilder res = new StringBuilder();
        do {
            res.appendCodePoint(c);
            c = read();
        } while (!isSpaceChar(c));
        return res.toString();
    }

    public int nextInt() throws IOException {
        int ret = 0;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (neg) return -ret;
        return ret;
    }

    public long nextLong() throws IOException {
        long ret = 0;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (neg) return -ret;
        return ret;
    }

    public double nextDouble() throws IOException {
        double ret = 0, div = 1;
        byte c = read();
        while (c <= ' ') c = read();
        boolean neg = (c == '-');
        if (neg) c = read();
        do {
            ret = ret * 10 + c - '0';
        } while ((c = read()) >= '0' && c <= '9');
        if (c == '.') {
            while ((c = read()) >= '0' && c <= '9') {
                ret += (c - '0') / (div *= 10);
```

```
            }
        }
        if (neg) return -ret;
        return ret;
    }

    private void fillBuffer() throws IOException {
        bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
        if (bytesRead == -1) buffer[0] = -1;
    }

    private byte read() throws IOException {
        if (bufferPointer == bytesRead) fillBuffer();
        return buffer[bufferPointer++];
    }

    public void close() throws IOException {
        if (din == null) return;
        din.close();
    }
}

class Print {
    private final BufferedWriter bw;

    public Print() {
        bw = new BufferedWriter(new OutputStreamWriter(System.out));
    }

    public void print(String str) throws IOException {
        bw.append(str);
    }

    public void printLine(String str) throws IOException {
        print(str);
        bw.append("\n");
    }

    public void close() throws IOException {
        bw.close();
    }
}
```

Listing 7: A code obtained from Listing 7 by eliminating dead codes.

```
public class Main {
    static long mod = ((long) 1e9 + 7);

    public static int gcd(int a, int b) {
        if ((b == 0)) return a;
        else return gcd(b, (a % b));
    }

    public static void main(String[] args) throws java.lang.Exception {
        Reader sn = new Reader();
        Print p = new Print();
        int n = sn.nextInt();
        while ((n > 0)) {
            int a = sn.nextInt();
            int b = sn.nextInt();
            int small = Math.min(a, b);
            int large = Math.max(a, b);
            long steps = 0;
            while ((small != 0)) {
```

```
            steps += (long) (large / small);
            int large1 = small;
            small = (large % small);
            large = large1;
         }
         p.printLine(Long.toString(steps));
      }
      p.close();
   }
}

class Reader {
   final private int BUFFER_SIZE = (1 << 16);
   private DataInputStream din;
   private byte[] buffer;
   private int bufferPointer, bytesRead;

   public boolean isSpaceChar(int c) {
      return (((((c == ' ') || (c == '\n')) || (c == '\r')) || (c ==
          '\t')) || (c == -1));
   }

   public Reader() {
      din = new DataInputStream(System.in);
      buffer = new byte[BUFFER_SIZE];
      bufferPointer = bytesRead = 0;
   }

   public Reader(String file_name) throws IOException {
      din = new DataInputStream(new FileInputStream(file_name));
      buffer = new byte[BUFFER_SIZE];
      bufferPointer = bytesRead = 0;
   }

   public int nextInt() throws IOException {
      int ret = 0;
      byte c = read();
      while ((c <= ' ')) c = read();
      boolean neg = (c == '-');
      if (neg) c = read();
      do {
         ret = (((ret * 10) + c) - '0');
      } while ((((c = read()) >= '0') && (c <= '9')));
      if (neg) return -ret;
      return ret;
   }

   private void fillBuffer() throws IOException {
      bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
      if ((bytesRead == -1)) buffer[0] = -1;
   }

   private byte read() throws IOException {
      if ((bufferPointer == bytesRead)) fillBuffer();
      return buffer[bufferPointer++];
   }

   public void close() throws IOException {
      if ((din == null)) return;
      din.close();
   }
}

class Print {
   final private BufferedWriter bw;
```

```
84
85      public Print() {
86          bw = new BufferedWriter(new OutputStreamWriter(System.out));
87      }
88
89      public void print(String str) throws IOException {
90          bw.append(str);
91      }
92
93      public void printLine(String str) throws IOException {
94          print(str);
95          bw.append("\n");
96      }
97
98      public void close() throws IOException {
99          bw.close();
100     }
101 }
```