Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Análise de desempenho de Grandes Modelos de Linguagem no desenvolvimento de código em Python

Autor: Gabriel Bonifácio Perez Nunes
Orientador: Prof. Dr. Nilton Correia da Silva

Brasília, DF
2024

Gabriel Bonifácio Perez Nunes

Análise de desempenho de Grandes Modelos de Linguagem no desenvolvimento de código em Python

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Prof. Dr. Nilton Correia da Silva

Brasília, DF

2024

Gabriel Bonifácio Perez Nunes

Análise de desempenho de Grandes Modelos de Linguagem no desenvolvimento de código em Python/ Gabriel Bonifácio Perez Nunes. – Brasília, DF, 2024-

72 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Nilton Correia da Silva

Trabalho de Conclusão de Curso – Universidade de Brasília – UnB
Faculdade UnB Gama – FGA , 2024.

1. grandes modelos de linguagem. 2. problemas de programação. I. Prof. Dr. Nilton Correia da Silva. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Análise de desempenho de Grandes Modelos de Linguagem no desenvolvimento de código em Python

CDU 02:141:005.6

Gabriel Bonifácio Perez Nunes

Análise de desempenho de Grandes Modelos de Linguagem no desenvolvimento de código em Python

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 12 de julho de 2024 – Data da aprovação do trabalho:

Prof. Dr. Nilton Correia da Silva
Orientador

Prof. Dr. Fabricio Ataides Braz
Convidado 1

Ma. Aline Dayany de Lemos
Convidado 2

Brasília, DF
2024

Dedicatória.

Dedico este trabalho à minha esposa Beatriz
e aos nossos filhos José e Pedro, que estão sempre a meu lado e
fazem de mim um marido, um pai e um homem melhor a cada dia.

"Estudante: aplica-te com espírito de apóstolo aos teus livros, com a convicção íntima de que essas horas e horas são já - agora! - um sacrifício espiritual oferecido a Deus, proveitosa para a humanidade, para o teu país, para a tua alma."

(São Josemaria Escrivá, Sulco 15, 522)

Resumo

A crescente adoção da inteligência artificial no cotidiano levanta questões sobre seu impacto no trabalho humano. Este estudo investiga até que ponto modelos avançados de inteligência artificial podem se equiparar a um desenvolvedor de software na geração de códigos de programação, além de analisar a diferença de desempenho entre os modelos abordados. O foco está na avaliação da qualidade dos códigos gerados por esses grandes modelos de linguagem na tentativa de solucionar problemas de programação. A pesquisa utiliza métricas específicas para analisar a eficácia e a precisão dos códigos produzidos. Considerando a relevância e a falta de estudos detalhados nessa área da inteligência artificial para códigos de programação, o estudo examina a capacidade de modelos específicos que estão em destaque atualmente de resolver problemas de programação em Python. A análise se concentra em aspectos como exatidão, tempo de execução, uso de memória, presença de comentários e módulos, complexidade cognitiva e manutenabilidade. Os objetivos principais incluem enviar algumas questões de programação aos modelos, apresentar os resultados obtidos, analisar as métricas de desempenho e identificar quais modelos se destacam na compreensão e produção desses códigos.

Palavras-chave: grandes modelos de linguagem. inteligência artificial. problemas de programação. qualidade de software.

Abstract

The increasing integration of artificial intelligence into daily life raises questions about its impact on human work. This study explores the extent to which advanced AI models can match a software developer in generating programming codes and assesses the performance differences among the covered models. The focus lies in evaluating the code quality generated by these language models in the attempt to solve programming problems. Specific metrics are employed in the research to analyze the effectiveness and accuracy of the produced code. Considering the significance and the lack of detailed studies in this area of artificial intelligence for programming code, the investigation delves into the capabilities of specific models that are currently highlighted in solving Python programming challenges. The analysis concentrates on aspects such as accuracy, execution time, memory usage, presence of comments and modules, cognitive complexity, and maintainability. The main objectives include sending some programming questions to the models, presenting the obtained results, analyzing the performance metric, and identifying which models excel in understanding and producing these codes.

Key-words: large language models. artificial intelligence. programming problems. software quality.

Lista de ilustrações

Lista de abreviaturas e siglas

LLM	EN-US: Large Language Model / PT-BR: Grande Modelo de Linguagem
IA	EN-US: Artificial Intelligence / PT-BR: Inteligência Artificial
API	EN-US: Application Programming Interface / PT-BR: Interface de programação de aplicações
XML	EN-US: Extensible Markup Language / PT-BR: Linguagem de Marcação Extensível
GPT	EN-US: Generative Pre-trained Transformer / PT-BR: Transformador Pré-treinado Generativo
LLaMA	EN-US: Large Language Model Meta AI / PT-BR: Grande Modelo de Linguagem META IA
KPMG	Klynveld Peat Marwick Goerdeler
UnB	Universidade de Brasília
FGA	Faculdade do Gama
WA	EN-US: Wrong Answer / PT-BR: Resposta Incorreta
AC	EN-US: Accepted / PT-BR: Resposta Correta
TLE	EN-US: Time Limit Exceeded / PT-BR: Limite de tempo excedido
RE	EN-US: Runtime error / PT-BR: Erro de tempo de execução

Sumário

Parte I

Introdução

1 Introdução

A utilização da inteligência artificial no cotidiano das pessoas tem se tornado cada vez mais frequente. Segundo uma pesquisa da KPMG[1], 82% das pessoas de todo o mundo disseram já ter ouvido sobre essa tecnologia e 50% das pessoas no Brasil disseram acreditar que as empresas em que trabalham utilizam a inteligência artificial. Por um lado, isso pode ser um motivo de preocupação na hipótese de que tal tecnologia poderá substituir o trabalho humano. Fato é que 42% acreditam que isso é realmente possível, segundo a pesquisa citada. (GILLESPIE N., 2023)

O poder dessa tecnologia é realmente alto, mas fica um questionamento se os grandes modelos de linguagem são capazes de substituir uma pessoa em qualquer trabalho como, por exemplo, substituir à altura um desenvolvedor de software especialista em linguagens de programação, e entender as capacidades técnicas de programação desses modelos. O intuito desse estudo é, portanto, analisar a performance de alguns grandes modelos de linguagem e compará-los entre si.

A presente pesquisa é baseada na investigação e avaliação da qualidade de software em códigos de programação que alguns LLMs podem gerar, com foco na compreensão e aplicação de métricas específicas.

1.1 Justificativa

A popularidade e discussão em maior escala dos LLMs teve um de seus picos após o lançamento do CHAT-GPT (OpenAI)[2], em novembro de 2022. Portanto, é um tema bastante atual e ainda não existem muitos estudos cuidadosos sobre como os modelos de linguagem podem construir códigos de programação precisos e de alta qualidade.

A evolução nesse campo de estudo auxilia no desenvolvimento da área como um todo, porque se torna possível e cada vez mais viável a geração de códigos de programação de forma rápida e em alto nível, podendo contribuir para outros campos na área de inteligência artificial. Dessa forma, é possível entender a preocupação de que essa tecnologia possa substituir pessoas no mercado de trabalho.

Para conseguir entender melhor essa área, é importante considerar qual é a qualidade que esses grandes modelos de linguagem conseguem apresentar, não somente para entender sua performance e sua comparação diante de um programador experiente, mas também entre os próprios modelos existentes atualmente.

[1] <https://kpmg.com/br/pt/home.html>
[2] <https://openai.com/blog/chatgpt>

1.2 Objetivos

Este trabalho tem como objetivo principal julgar a capacidade de geração de código de alguns modelos de linguagem. Estes receberão o direcional de produzir um código para solucionar um problema de programação específico. Dessa forma, serão retiradas certas análises sobre o que cada modelo produziu de código. Serão utilizados os seguintes modelos: GPT-3.5-TURBO-1106[3], H2oGPT-32k-codellama-34b-instruct[4], Gemini[5], Gemma-7B[6] e Llama3 [7], mediante suas aplicações em exercícios de programação. Utilizando a linguagem Python, serão realizados alguns testes para entender como que cada um desses modelos escolhidos respondem aos desafios propostos.

Para entender tais qualidades de software, serão observadas as seguintes métricas (detalhadas posteriormente): exatidão do código, tempo de execução, memória consumida, presença de comentários e módulos no código, complexidade cognitiva e manutenabilidade. Com isso, estes testes contemplarão uma análise quanto ao desempenho desses códigos gerados para que seja possível avaliar e apresentar a real qualidade daquilo que foi gerado.

Para melhor julgar um modelo, serão realizadas essas tarefas:

1. ***Definição dos problemas de programação***
2. ***Envio dos problemas de programação para o modelo***
3. ***Extrair a solução proposta***
4. ***Enviar a solução proposta para o juiz online dar o veredito***
5. ***Enviar a solução proposta para o SonarCloud***
6. ***Coletar os dados e analisar o retorno que as ferramentas deram para a solução proposta***

1.3 Conteúdos dos capítulos

- Capítulo 2: Materiais e Métodos

  Neste capítulo serão abordadas as principais ferramentas a serem utilizadas e cada método que sustentará a construção deste trabalho, assim como as métricas de qualidade de software e como será utilizada a engenharia de prompt para uma melhor comunicação com os modelos.

[3] <https://platform.openai.com/docs/models/gpt-3-5>
[4] <https://huggingface.co/h2oai/h2ogpt-32k-codellama-34b-instruct>
[5] <https://deepmind.google/technologies/gemini/>
[6] <https://ai.google.dev/gemma/docs/model_card>
[7] <https://ai.meta.com/blog/meta-llama-3/>

- Capítulo 3: Metodologia

  Em metodologia, será demonstrado o que será feito com os materiais e métodos especificados no capítulo anterior. Serão definidos e especificados os problemas de programação e a construção dos testes de software, assim como um fluxograma representando os processos do trabalho.

- Capítulo 4: Resultados alcançados

  Este capítulo terá a apresentação dos resultados alcançados, com as respostas dos modelos aos problemas propostos e a avaliação das soluções apresentadas.

- Capítulo 5: Conclusão e trabalhos futuros

  Na conclusão serão recapitulados os principais resultados do trabalho, assim como a confirmação se os objetivos inicialmente traçados se relacionam com o resultado final do trabalho. Serão descritos os trabalhos futuros, e serão pontuadas quais partes do trabalho que poderiam ser mais amplamente abordadas e possíveis desafios futuros.

2 Materiais e Métodos

2.1 Introdução

Este capítulo aborda os materiais e métodos utilizados para avaliar a qualidade de software de alguns códigos de programação. Ao longo deste estudo, serão exploradas algumas das principais métricas de qualidade como exatidão, eficiência de tempo e memória, presença de comentários e funções no código, complexidade cognitiva e manutenabilidade.

Além disso, este capítulo discutirá o emprego de ferramentas como SonarCloud[1], uma ferramenta que apresenta um conjunto de análises estáticas que auxiliam na verificação e validação das métricas estabelecidas; Codeforces[2] que é outra plataforma que será abordada, sendo essencial para validar a exatidão do código, o tempo gasto na execução e a memória consumida pelo código, garantindo sua precisão e funcionamento esperado.

A partir da resposta do modelo, o código será encaminhado para a questão de programação do Codeforces que retornará a exatidão, o tempo gasto e o tamanho consumido.

De forma a auxiliar os modelos passando o comando correto, será explorada a engenharia de prompt, direcionada à otimização do processo de interação com o LLM na busca de se obter um código com uma qualidade superior.

2.2 Qualidade de Software

A qualidade de um software está objetivamente relacionada com seus requisitos. Portanto, é preciso entender as ações que o software deve executar e quais são os resultados esperados. (SALVIANO, 2020).

Em um código de programação, toda a ideia da qualidade do código se apresenta de forma mais objetiva, já que as entradas e as saídas esperadas são dados únicos e a eficiência do código pode cumprir ou não as limitações propostas pelo problema. Além disso, há um lado mais subjetivo como comentários de código, modularização e a complexidade ciclomática, mas que também pode ser representado por valores mais objetivos de forma a ser possível avaliar bem tais métricas.

Durante este trabalho, a avaliação da qualidade de código apresentado pelos modelos passará por algumas métricas específicas estabelecidas no tópico a seguir.

[1] <https://www.sonarsource.com/products/sonarcloud/>
[2] <https://codeforces.com/>

2.2.1 Métricas de Software

Na hora de selecionar as métricas que servirão de base, deve-se levar em consideração sua objetividade e se há a possibilidade de extrair algum dado claro a partir de sua análise.

Além de considerar que devem ser objetivas com o objetivo de reduzir a influência pessoal na coleta de dados, as métricas precisam ser simples de entender para facilitar na hora da verificação se as metas foram ou não alcançadas. (SALVIANO, 2020).

Nesse sentido foram selecionadas as métricas que servirão de base para as análises de dados:

- Exatidão: representa se o código atende as saídas esperadas; se o código proposto pelo modelo funciona para o problema proposto. O método de avaliação se dará na resposta do juiz online, que pode ser:
  - Correto (AC): é a única resposta que representa que a solução foi desenvolvida de forma ideal a atender o problema e passou em todos os casos de teste.
  - Resposta errada (WA): o código não apresenta o resultado correto para todos os casos de testes do problema. Um erro em uma única questão é suficiente para apresentar a solução como incorreta.
  - Erro de tempo de execução (RE): erro que costuma aparecer quando se define um vetor com menos capacidade do que o necessário para o problema. O código foi enviado com problemas no tempo de execução.
  - Limite de tempo excedido (TLE): a solução submetida levou um tempo maior que o permitido para a questão (definida no enunciado do problema de programação).
  - Limite de memória excedido (MLE): a solução submetida levou um consumo maior de memória que o permitido para a questão (definida no enunciado do problema de programação).
  - Erro de apresentação (PE): a solução aparece com um problema na hora de apresentar o resultado, podendo ser um erro na quantidade de espaços, caracteres ou símbolos na hora de apresentar o dado. Torna a solução como incorreta.

1. Problema: verificar se um código retorna a resposta correta para um problema específico.
2. Exemplo: um algoritmo para verificar se um número é primo. Se o código retorna 'verdadeiro' para números primos e 'falso' para não-primos, a exatidão é avaliada comparando a saída do código com um verificador de números primos.

- Eficiência de tempo: representa o tempo que o código demora para terminar e apresentar o resultado; se o código proposto pelo LLM consegue gastar menos tempo para rodar os testes que o limite máximo definido no enunciando do problema. O método de avaliação será em tempo gasto em milissegundos.

    1. Problema: Medir o tempo que o código leva para executar uma determinada tarefa.
    2. Exemplo: Ordenação de uma lista. Comparar o tempo que um algoritmo de ordenação leva para ordenar uma lista de tamanho específico.

- Eficiência de memória: representa a memória que o código consome para apresentar o resultado; se o código proposto pelo LLM atende os requisitos de memória do problema. O método de avaliação será de memória consumida em megabyte.

    1. Problema: avaliar a quantidade de memória utilizada por um código durante a execução.
    2. Exemplo: algoritmo para armazenar uma grande quantidade de dados. Comparar o uso de memória entre diferentes abordagens de armazenamento de dados para avaliar qual consome menos memória e não ultrapassa o limite de memória.

- Modularização: representa a quantidade de quebras do código em partes menores de forma a encapsular funções; se o código faz o uso de funções e quantas foram utilizadas. O método de avaliação será em quantidade de funções usadas no código.

    1. Problema: analisar como o código é dividido em funções e quantas são.
    2. Exemplo: identificar quantas funções ou módulos foram utilizados no código.

- Comentários de código: representa a quantidade de linhas de comentários presentes no código; O método de avaliação será em frequência no código.

    1. Problema: verificar a presença de comentários no código.
    2. Exemplo: avaliar um código para ver se contém explicações sobre o que algumas seções de código estão fazendo. Verificar se há comentários descrevendo trechos do código.

- Manutenabilidade[3]: é a métrica que avalia a facilidade ao qual o código pode ser reparável, melhorado e entendível.

    1. Problema: verificar se a manutenabilidade foi violada e se fornece algum risco para o software.

[3] <https://docs.sonarsource.com/sonarcloud/digging-deeper/metric-definitions/#maintainability>

2. Exemplo: o código do GPT para a questão 10 violou a manutenabilidade, fornecendo um alto risco pro software.

- Complexidade cognitiva: uma métrica quantitativa usada para entender o quão difícil é entender o fluxo de controle do código. Ela compreende tanto o número de caminhos através do código quanto a sua dificuldade em compreender o código como um todo.[4] O principal motivo da implementação da complexidade cognitiva se dá por uma certa ineficácia da complexidade ciclomática [5] em alguns pontos. A complexidade ciclomática é calculada baseada nos caminhos que o código percorre durante o código, somente. Por esse motivo pode, por exemplo, dar a percepção de "alarmes falsos" ao supervalorizar algumas estruturas e subvalorizar outras. (CAMPBELL, 2023)

  O SonarCloud define, por padrão, um limite permitido da complexidade cognitiva de até 15. Ao passar desse número, irá gerar uma pendência na métrica de manutenabilidade, seguido de uma orientação de refatoração do código a fim de diminuir tal complexidade. Para este trabalho, esse limite também será aplicado, de forma que essa métrica fique mais objetiva.

  1. Problema: entender a dificuldade de compreender o caminho que o código percorre.
  2. Exemplo: retornará uma métrica quantitativa que retornará o nível de dificuldade de compreensão do código. Quanto maior, mais complicado é o seu entendimento.

A tabela abaixo apresenta as métricas a serem analisadas e qual ferramenta será usada para extrair cada uma delas.

Tabela 1 – Métricas de Qualidade de Software

Nome da Métrica	**Ferramenta a ser utilizada**
Exatidão	Codeforces
Eficiência de tempo	Codeforces
Eficiência de memória	Codeforces
Modularização	SonarCloud
Comentários no código	SonarCloud
Complexidade Cognitiva	SonarCloud
Manutenabilidade	SonarCloud

[4] <https://docs.sonarsource.com/sonarcloud/digging-deeper/metric-definitions/#complexity>
[5] <https://docs.sonarsource.com/sonarcloud/digging-deeper/metric-definitions/#complexity>

2.2.2 Ferramentas

Na busca pela objetividade no momento da avaliação da qualidade dos códigos, foram adotadas algumas ferramentas voltadas para a análise de métricas. Essas ferramentas desempenham um papel fundamental neste trabalho ao oferecer métricas precisas, permitindo uma compreensão mais aprofundada da eficiência das soluções.

São ferramentas de extração de métricas de qualidade de software que se propõem a apresentar abordagens distintas para a análise de código. Abaixo serão detalhadas cada uma das ferramentas a serem utilizadas neste estudo:

- **SonarCloud**: é uma plataforma de código aberto para análise da qualidade de código com o objetivo de se realizar análises estáticas. Ela oferece um amplo conjunto de métricas para avaliar a qualidade do código, como códigos mal estruturados (o que pode ser resultado da quantidade de funções no código), quantidade de comentários no código, métricas de qualidade de software e a complexidade cognitiva. O SonarCloud suporta a linguagem de programação Python e fornece um painel de controle intuitivo para visualização das métricas de código.

  Na figura 1, é apresentado uma fração de código que contém 9 linhas comentadas e a ferramenta consegue diferenciar o que é de fato um comentário válido, um comentário vazio e um código que foi comentado:

Figura 1 – Exemplo de Código Comentado

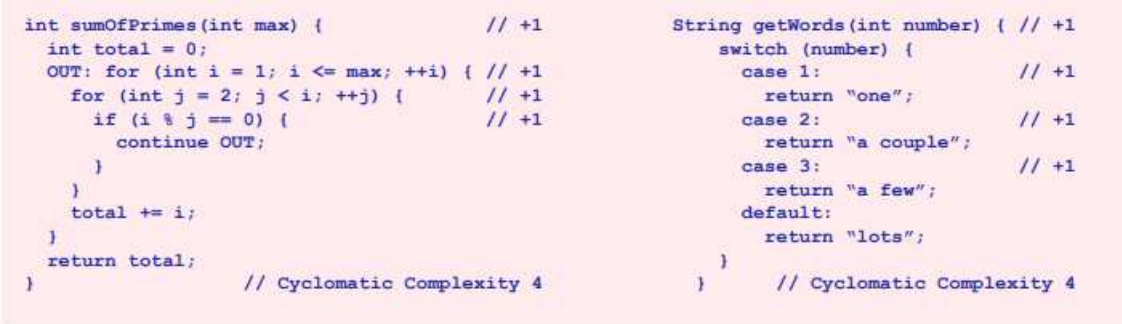
```
/**                                          +0 => empty comment line
 *                                           +0 => empty comment line
 * This is my documentation                  +1 => significant comment
 * although I don't                          +1 => significant comment
 * have much                                 +1 => significant comment
 * to say                                    +1 => significant comment
 *                                           +0 => empty comment line
 ***************************                  +0 => non-significant comment
 *                                           +0 => empty comment line
 * blabla...                                 +1 => significant comment
 */                                          +0 => empty comment line

/**                                          +0 => empty comment line
 * public String foo() {                     +1 => commented-out code
 *   System.out.println(message);            +1 => commented-out code
 *   return message;                         +1 => commented-out code
 * }                                         +1 => commented-out code
 */                                          +0 => empty comment line
```

Fonte: SonarCloud.

A figura 2 representa dois métodos que possuem a mesma complexidade ciclomática,

mas são bem diferentes em termos de entendimento de código. Por esse motivo, a complexidade cognitiva é aplicada, de forma a retornar com uma precisão maior qual é a dificuldade de se compreender um código.

Figura 2 – Exemplo de Complexidade Ciclomática



```
int sumOfPrimes(int max) {                    // +1
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) {       // +1
    for (int j = 2; j < i; ++j) {             // +1
      if (i % j == 0) {                       // +1
        continue OUT;
      }
    }
    total += i;
  }
  return total;
}                          // Cyclomatic Complexity 4
```

```
String getWords(int number) {  // +1
    switch (number) {
      case 1:                  // +1
        return "one";
      case 2:                  // +1
        return "a couple";
      case 3:                  // +1
        return "a few";
      default:
        return "lots";
    }
}      // Cyclomatic Complexity 4
```

Fonte: Cognitive Complexity Sonar Guide

- **Codeforces**: O Codeforces é uma plataforma utilizada por estudantes e profissionais independentes que desejam praticar e desenvolver suas habilidades em problemas de programação, pois atua como um juiz online fazendo a validação de códigos de programação. Ele fornecerá o resultado dos códigos propostos pelo LLM, além do tempo e tamanho que ocupou para passar nos casos de teste. (PRIVACY. . . , 2024)

- **Python**: é uma linguagem de programação de alto nível amplamente utilizada, incentivada por sua simplicidade em comparação a outras, legibilidade e uma alta quantidade de bibliotecas e módulos úteis. No contexto da análise de métricas de código, o Python é uma linguagem otimizada para desenvolver scripts ou programas que realizam medições de eficiência de tempo, consumo de memória e outras métricas do código. (CARVALHO, 2024)

- **Ollama**[6]: é uma ferramenta que torna possível a execução de alguns grandes modelos de linguagem. Com sua execução facilitada, oferece flexibilidade ao permitir a execução de modelos com diferentes configurações e tamanhos, atendendo a uma ampla gama de modelos. Dessa forma, foi utilizada nesse trabalho para o auxílio de execução de dois modelos. (THOUGHTWORKS, 2024)

- **Kaggle**[7]: uma plataforma online de ciências de dados. Foi utilizada para executar um LLM, envolvendo a utilização dos notebooks Kaggle para a execução de certos modelos. De forma a tirar proveito do poder computacional oferecido, a partir de uma execução com a ferramenta do Ollama, foi possível extrair os resultados dos modelos Gemma e Llama3. (DATACAMP, 2024)

[6] <https://ollama.com/>
[7] <https://www.kaggle.com/>

2.3 Grandes Modelos de Linguagem

Os LLMs são modelos de linguagem de grande escala que utilizam técnicas de aprendizado de máquina para gerar texto em linguagem natural. Eles operam por meio de uma arquitetura neural complexa permitindo o processamento de informações e a geração de respostas com base em um amplo conjunto de dados de treinamento. (DSA, 2023).

Quando se trata de gerar respostas para códigos de programação, apenas alguns LLMs foram especificamente treinados para escrever código — um exemplo é o CodeGPT[8] que oferece um uso personalizado e mais voltado à programação. No entanto, modelos mais conhecidos como o GPT-3.5[9] ou Gemini (antigamente conhecido Bard)[10] não possuem essa especificação voltada aos códigos de programação. No entanto, esses modelos com uma alta quantidade de parâmetros e de armazenamento, possuem a capacidade de realizar tarefas simples de codificação com base na inferência de padrões aprendidos durante o treinamento. (KUNDU, 2023).

Eles podem gerar trechos de código ao receber instruções e exemplos claros do que se espera. No entanto, é importante ressaltar que sua habilidade de escrever código é limitada e pode gerar resultados imprecisos ou inadequados para tarefas complexas ou específicas. (KOUL, 2023).

O modelo interpreta a entrada fornecida (por exemplo, uma descrição de um problema de programação) e tenta gerar uma sequência de código que se alinhe com essa descrição. Ele faz isso com base no contexto e nos padrões que aprendeu dos dados de treinamento. Se o exemplo dado se assemelha a algo que o modelo já viu em sua enorme quantidade de dados de treinamento, ele tentará gerar um código que se aproxime dessa expectativa.

Abaixo serão especificados cinco grandes modelos de linguagem que serão utilizados para a principal proposta deste trabalho que é a avaliar a qualidade de código que estes modelos conseguem extrair. Sendo estes modelos:

- GPT-3.5-TURBO-1106[11]: criado pela OpenAI, este modelo é uma versão aprimorada e com instrução guiada melhorada dos outros GPT-3.5, possui modo JSON, um retorno de até 4096 tokens de saída, entre outras melhorias. De forma geral, o OpenAI tem sido referência na criação desses modelos. A arquitetura é baseada em Transformer (RADFORD et al., 2018). Sua utilização se dará localmente por meio de uma chave de API, onde será possível estruturar um código para enviar os comandos desejados.

[8] <https://codegpt.co/>
[9] <https://platform.openai.com/docs/models>
[10] <https://gemini.google.com/app>
[11] <https://platform.openai.com/docs/models/gpt-3-5>

- Motivo da escolha do LLM: um dos modelos mais famosos atualmente, com alta capacidade de geração de textos, códigos e afins. Por ser uma ferramenta muito utilizada, o uso dela para geração de código pode se tornar uma facilidade para diversos programadores. Sua escolha, portanto, se dá nessa ampla utilização e facilidade de uso.
- Método de uso neste trabalho: via API, utilizando o código 2.1 para fazer as requisições.
- Nome identificador do modelo neste trabalho: GPT.

No código 2.1 é apresentado a solução que faz a requisição da chave API do OpenAI e operacionaliza esse envio da requisição para o modelo. Sendo o prompt o texto a ser enviado para o modelo. A estrutura e a solução proposta do código é da empresa Hashtag Treinamentos[12]. (PROGRAMAçãO, 2023).

```
1    # Importa a variavel API_KEY do arquivo senha e certas bibliotecas
2    from senha import API_KEY
3    from questoes import q1, template
4    import requests
5    import json
6
7    # Define os headers necessarios para a requisi  o
8    headers = {"Authorization": f"Bearer {API_KEY}", "Content-Type": "
     application/json"}
9
10   # URL da API OpenAI para fazer a requisicao
11   link = "https://api.openai.com/v1/chat/completions"
12
13   # Identificador do modelo a ser utilizado
14   id_modelo = "gpt-3.5-turbo-1106"
15
16   # Chama a fun   o q1() do arquivo questoes para obter um prompt
17   template = template()
18   prompt = q1()
19
20   # Concatena o prompt com o template
21   prompt = str(prompt)
22   template = str(template)
23   prompt = template + prompt
24
25   # Cria um dicionario com o modelo e as mensagens a serem processadas
     pelo modelo
26   body_mensagem = {
27       "model": id_modelo,
```

[12] <https://www.hashtagtreinamentos.com/cursos-hashtag-programacao>

```
28          "messages": [{"role": "user", "content": prompt}],
29      }
30
31      # Converte o dicionario para o formato JSON
32      body_mensagem = json.dumps(body_mensagem)
33
34      # Envia uma requisicao POST para a API OpenAI com os dados
        preparados
35      requisicao = requests.post(link, headers=headers, data=body_mensagem
        )
36
37      # Converte a resposta da requisicao
38      resposta = requisicao.json()
39
40      # Extrai e imprime a resposta gerada pelo modelo
41      print(resposta.get("choices")[0].get("message").get("content"))
```

Código 2.1 – Código para GPT-3.5

- H2oGPT-32k-codellama-34b-instruct[13]: um dos modelos do H2o.AI[14], o H2OGPT disponibiliza uma plataforma[15] para se estabelecer uma comunicação com este modelo — a plataforma será utilizada para uso do modelo em questão, dada essa viabilidade. Ele é otimizado para diálogos e é capaz de desenvolver códigos com o propósito de resolver problemas de programação. Este modelo é um clone do modelo do codeLlama-34b-instruct-hf[16], que é uma coleção de modelos de texto generativos pré-treinados, com uma grande quantidade de parâmetros, variando de 7 a 34 bilhões.

- Motivo da escolha do LLM: um dos modelos que possuem plataforma aberta para utilização, sem obrigação de uso da *API*, tornando seu uso simples e prático - de fácil utilização no dia-a-dia. Seu uso também poderá ajudar a entender a performance de um modelo que é um clone de outro.
- Método de uso neste trabalho: via plataforma do H2o.ai[17].
- Nome identificador do modelo neste trabalho: H2o.

O código 2.2 é um dos exemplos de resposta deste modelo após uma solicitação que gerasse um código para um problema de programação específico.

```
1   # Leia 2 valores inteiros e armazene-os nas variaveis A e B
2   a = int(input())
```

[13] <https://huggingface.co/h2oai/h2ogpt-32k-codellama-34b-instruct>
[14] <https://huggingface.co/h2oai>
[15] <https://codellama.h2o.ai/>
[16] <https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf>
[17] <https://codellama.h2o.ai/>

```
    b = int(input())

    # Efetue a soma de A e B e atribua o resultado a variavel X
    x = a + b

    # Imprima X seguido do final de linha
    print("X =", x)
```

Código 2.2 – Exemplo de resposta do modelo

- Gemini[18]: o Bard foi lançado originalmente como um modelo de linguagem natural para responder perguntas, gerar texto e realizar outras tarefas baseadas em linguagem. No entanto, com a evolução do projeto, ele foi renomeado para Gemini, refletindo uma expansão dos objetivos da Google. É um projeto de inteligência artificial que tem ganhado notoriedade pela sua fácil execução, similar ao do GPT.

- Motivo da escolha do LLM: modelo de uma grande empresa que vem trazendo bastante atenção recente. Após uma ressignificação no modelo (de BARD para Gemini), traz uma facilidade pro usuário ao enviar um comando, seja por texto, imagem ou voz. Tem sido mais reconhecido e pode trazer bons resultados para este estudo.
- Método de uso neste trabalho: via própria plataforma do Gemini[19].
- Nome identificador do modelo neste trabalho: Gemini.

- Gemma-7B[20], da Google DeepMind[21], empresa de inteligência artificial, é um "integrante" da família dos modelos mais leves. É inspirado no Gemini, pois foram criados da mesma tecnologia e fonte de pesquisa. (TEAM et al., 2024). De forma a executar o modelo de forma eficiente e rápida, foi utilizada a ferramenta do Ollama[22] para utilização do modelo Gemma.

Para este trabalho, será utilizado o Gemma que possui 7 bilhões de parâmetros.

- Motivo da escolha do LLM: modelo um pouco mais desconhecido, com 7 bilhões de parâmetros, mas que também integra uma grande empresa e é inspirado no modelo anterior (Gemini). A ideia é entender se um modelo com menos parâmetros pode trazer boas performances quando se trata de geração de código.
- Método de uso neste trabalho: via Ollama, com código executado pelo Kaggle.

[18] <https://deepmind.google/technologies/gemini/>
[19] <https://gemini.google.com/>
[20] <https://ai.google.dev/gemma/docs/model_card>
[21] <https://deepmind.google/>
[22] <https://ollama.com/>

- Nome identificador do modelo neste trabalho: Gemma.

No código 2.3 é apresentada a solução que faz a utilização do modelo e operacionaliza esse envio da requisição para ser executado pelo modelo, sendo OLLAMA_MODEL='gemma:7b' e {questao} o prompt + enunciado do problema.

```
!curl https://ollama.ai/install.sh | sh
!sudo apt install -y neofetch

!neofetch

OLLAMA_MODEL='gemma:7b'

import os
os.environ['OLLAMA_MODEL'] = OLLAMA_MODEL
!echo $OLLAMA_MODEL

import subprocess
import time

# Iniciar ollama como um processo em segundo plano
command = "nohup ollama serve&"

# Usar subprocess.Popen para iniciar o processo em segundo plano
process = subprocess.Popen(command,
                           shell=True,
                          stdout=subprocess.PIPE,
                          stderr=subprocess.PIPE)
print("ID do processo:", process.pid)

time.sleep(5)  # Esperar 5 segundos

llama run $OLLAMA_MODEL {questao}
```

Código 2.3 – Código para Gemma-7B

- Llama3[23], "o modelo mais capaz disponível abertamente até o momento", segundo a própria empresa fundadora, META AI[24]. Tal modelo é otimizado para casos de uso de diálogo e bate-papos, superando muitos modelos de bate-papo de código aberto disponíveis. De forma a executar o modelo de forma eficiente e rápida, foi utilizada a ferramenta do Ollama[25] para utilização do modelo Llama3.

Para este trabalho, será utilizado o Gemma que possui 8 bilhões de parâmetros.

[23] <https://ai.meta.com/blog/meta-llama-3/>
[24] <https://ai.meta.com/>
[25] <https://ollama.com/>

- Motivo da escolha do LLM: modelo da META AI[26] com uma alta performance, segundo o próprio modelo, para diálogos. O objetivo é analisar se esse desempenho também transparece no desenvolvimento de códigos em Python. Modelo com alto potencial de evolução, e portanto terá sua aplicação neste estudo.

- Método de uso neste trabalho: via Ollama, com código executado pelo Kaggle.

- Nome identificador do modelo neste trabalho: Llama3.

No código 2.3 é apresentada a solução que faz a utilização do modelo e operacionaliza esse envio da requisição para ser executado pelo modelo, sendo OLLAMA_MODEL='llama3' e {questao} o prompt + enunciado do problema.

2.4 Engenharia de Prompt

A engenharia de prompt se refere ao processo de otimização e aprimoramento da interação do usuário com um sistema, sendo no caso desse trabalho, a inteligência artificial. (LARGUESA, 2023).

Seu objetivo, portanto, é utilizar as instruções da melhor forma para extrair respostas e ações específicas dos sistemas. Esse processo será de grande importância para o trabalho para conseguir respostas precisas.

Para ilustrar melhor, é possível ver por meio de uma lista algumas das razões pelas quais a engenharia de prompt se fará importante nesta pesquisa (LARGUESA, 2023):

- Comunicação eficiente: envolve a habilidade de transmitir instruções de maneira clara aos LLMs, permitindo que ela interprete corretamente tanto o problema proposto quanto demais instruções. Isso engloba também a capacidade do modelo em responder de forma precisa, garantindo que as informações sejam repassadas corretamente.

- Mitigação da ambiguidade: refere-se à redução ou eliminação de informações de duplo sentido nos comandos fornecidos ao LLM. Isso pode ser alcançado por meio de instruções bem definidas, evitando linguagem ambígua que possa ser interpretado de maneira diferente da esperada.

- Melhoria contínua: diz respeito ao processo de aprimoramento constante do ambiente de interação. Isso envolve a identificação de áreas que precisam de aperfeiçoamento para aprimorar a eficiência e precisão das interações.

[26] <https://about.meta.com/br/>

- Convergência entre humanos e LLM: se refere à colaboração efetiva entre ambos para alcançar objetivos comuns. Isso implica em uma interação fluida, onde os sistemas de IA compreendem de forma precisa as instruções fornecidas pelos usuários, que compreendem as respostas que o LLM retornar. Harmonia entre as duas partes.

Dadas essas razões, é preciso entender de que forma deve-se apresentar as instruções para extrair as melhores respostas possíveis. Todos estes tópicos abaixo são essenciais na hora de formular um prompt eficaz. (LARGUESA, 2023).

- Ser específico: especificar mais as perguntas com o objetivo de reduzir possíveis ambiguidades.

  Exemplo: em vez de perguntar "quais os benefícios de usar o Python?", é mais recomendado perguntar "quais os benefícios de usar o Python numa análise de código envolvendo métricas de qualidade de software?".

- Definir o escopo: delimitar o escopo é uma prática importante e necessária na interação com o LLM.

  Exemplo: "aponte três métricas de qualidade de software que auxiliam na análise de um código" é mais eficaz que "quais métricas de qualidade de software auxiliam na análise de um código?".

- Dar informações detalhadas: especificar mais as informações pedindo detalhes.

  Exemplo: "como que o SonarQube avalia o meu código?" não teria uma resposta tão satisfatória quanto teria se a pergunta fosse feita assim: "como que o SonarQube avalia a complexidade ciclomática do meu código que foi desenvolvido em Python?".

- Evitar perguntas com múltiplas partes: a melhor prática é dividir uma pergunta em várias partes menores e mais focadas.

  Exemplo: Troque a pergunta "Como que a biblioteca time e a biblioteca memory-profiler operam e como eu posso desenvolver um código que possa abranger as duas?" por perguntas separadas em duas partes para obter respostas mais precisas.

- Fornecer contexto: trazer informações contextuais para ajudar o LLM a compreender o todo.

  Exemplo: "por que a modularização auxilia no tempo de execução de um código de programação?" é melhor do que "por que a modularização auxilia em um código de programação?".

- Usar delimitadores para indicar diferentes partes do texto: na ideia de enviar um problema de programação que contém a proposta do problema, especificações de

entrada e saída e exemplos, é importante que haja essa delimitação para passar o melhor comando possível.

Exemplo: “Faça um resumo do artigo abaixo que está delimitado entre tags de XML. <texto> artigo <texto>”.

Na aplicação dessas técnicas, o modelo estará bem direcionado para uma resposta mais precisa.

3 Metodologia

3.1 Introdução

O presente capítulo apresenta a metodologia adotada para julgar a qualidade de software em códigos de programação, o principal objetivo deste trabalho. O foco deste capítulo se baseia na apresentação de como se dará a metodologia que abrange a análise desde métricas como a exatidão do código até a otimização do processo de interação com os modelos.

A abordagem metodológica adotada para este estudo também contempla a análise e a definição dos problemas de programação, a construção do template inicial do comando e a representação visual desses processos por meio de um fluxograma. Esta estrutura visa não apenas demonstrar os métodos utilizados, mas também estabelecer um caminho claro para a compreensão e a replicação dos procedimentos realizados.

3.2 Escopo Inicial

De forma a melhor julgar a capacidade de geração de código dos modelos, será preciso considerar quais serão os critérios para julgar um modelo como bom ou ruim.

Por esse motivo, será necessária a definição de métricas específicas já existentes para que não haja uma subjetividade na hora de julgar. Neste trabalho, o fluxo de avaliação desses modelos não deve abrir margem para interpretação.

O caminho percorrido para definir essas métricas foi se baseando nas métricas disponíveis do SonarCloud (uma ferramenta já própria pra esse time de análise de métricas) que não tivessem uma subjetividade ou não apresentassem uma clareza quanto às suas utilizações neste trabalho.

Inicialmente, foram definidas 7 métricas para serem utilizadas no SonarCloud, sendo elas as métricas de manutenabilidade, complexidade ciclomática, complexidade cognitiva, comentários, funções, confiabilidade e segurança. No entanto, no decorrer das pesquisas do trabalho, foi entendido que as métricas de confiabilidade e segurança não apresentariam uma relevância para o trabalho, já que todos os modelos apresentavam um alto indíce de acerto nessas métricas, não se diferenciando entre eles.

Além dessas, a métrica de complexidade ciclomática foi removida, já que a complexidade cognitiva apresenta um dado bem mais concreto e confiável, por conta da forma que o SonarCloud apresenta sua complexidade no código.

Por parte do juiz online, as métricas de exatidão e tempo de execução são métricas principais quando se trata de problemas de programação. Sendo a métrica de exatidão a principal, não há a possibilidade de se analisar a eficiência de uma solução sem olhar primeiro para as respostas que ela gera com diferentes casos-teste. O tempo de execução vai trabalhar também nesse sentido, já que uma solução devagar irá representar, muito provavelmente, uma falha na geração do código. A métrica de memória consumida trabalhará na mesma ideia da de tempo, mas com muito menos ocorrências, já que ocupar toda a memória proposta seria mais difícil.

De forma a compreender isso no código, também foi levado em consideração a necessidade de se manter somente métricas exatas, sem levar em conta a interpretação do desenvolvedor ou orientador deste estudo.

É com esse propósito que foram definidas as métricas de exatidão , tempo de execução e memória consumida que serão extraídas do próprio Codeforces. Isso garante uma objetividade na hora de se avaliar: ou a solução acerta, ou erra. Ou a solução executa no tempo esperado, ou não. Sempre com uma análise de ”0 ou 1” pra descartar a possibilidade de uma avaliação humana incorreta.

Com o SonarCloud extraindo as métricas de manutenabilidade, complexidade ciclomática, comentários e funções presentes no código, também não haverá subjetividade que impacte o julgamente dos modelos.

Para melhor compreensão de como se darão esses passos, na figura 3 é apresentado um fluxograma geral que mostra cada etapa deste estudo:

O fluxograma apresenta o caminho que será usado para cada problema de programação. De forma mais detalhada:

- *Prompt* (*template* + Problema de Programação).

  O fluxo inicia com a entrada do texto contendo o template de interação com o LLM juntamente com o problema de programação específico a ser resolvido.

- LLM:

  Representa o modelo, que recebe o texto que é a junção do *template* com o problema de programação para então gerar as soluções em código.

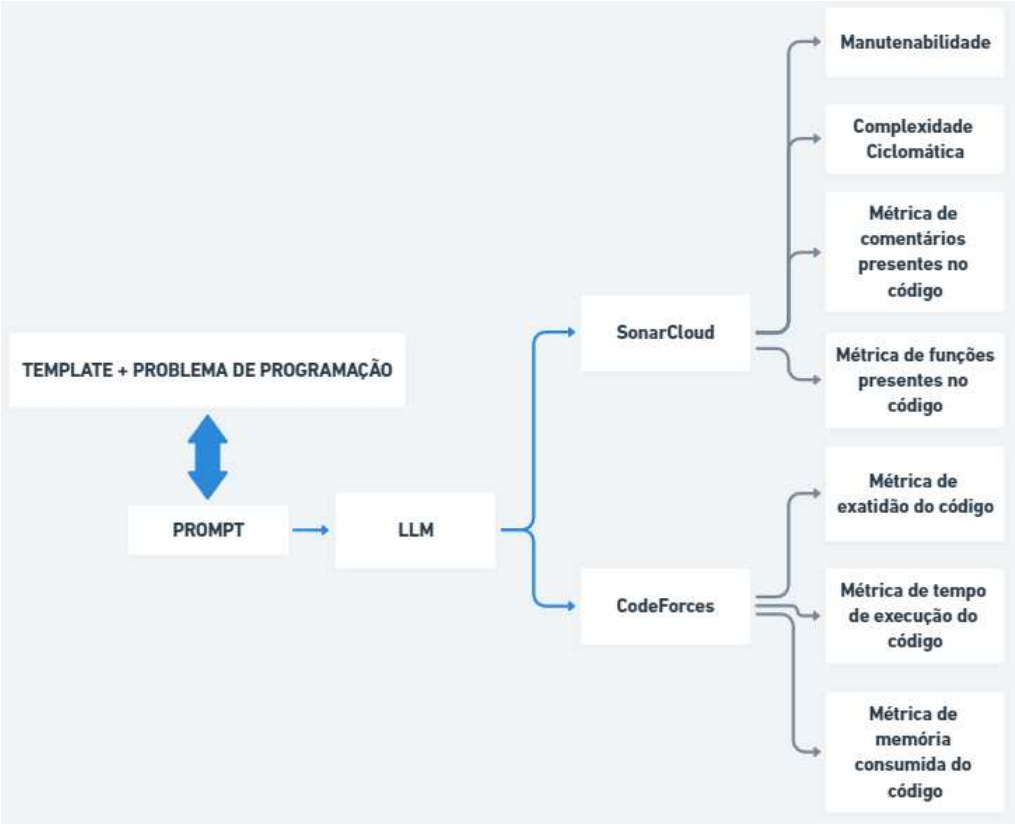
- SonarCloud:

  As soluções de código geradas pelo LLM são enviadas para o SonarCloud.

  O SonarCloud realizará análises estáticas e fornecerá métricas de qualidade, tais como:

  – Complexidade Cognitiva;

Figura 3 – Fluxograma Geral



  - Quantidade de linhas de comentários presentes no código;
  - Quantidade de módulos/funções presentes no código;
  - Manutenabilidade.

- Codeforces:

  As soluções de código geradas pelos LLMs são enviadas para o Codeforces, que executa os códigos gerados e fornece uma métrica de exatidão do código, avaliando se o código produz a saída esperada para o problema proposto, além do tempo e do tamanho que a solução levou para passar nos testes.

Portanto, esse fluxograma descreve um processo sequencial e interconectado em que o texto de entrada passa por várias etapas de avaliação utilizando certas ferramentas para a obtenção de métricas que abrangem diferentes aspectos da qualidade do código gerado pelo LLM.

3.3 Definição do prompt

Na proposta deste trabalho, é essencial para o sucesso do estudo colocar na estrutura do texto um *template* em texto dando alguns direcionais iniciais para os LLMs.

Dessa forma, será possível conduzir de forma ideal de como são esperadas as respostas desses modelos.

Considerando todas as orientações tratadas na seção Engenharia de *Prompt* do capítulo de Materiais e Métodos, este template inicial terá a seguinte estrutura:

- Convidá-lo a estar em uma posição de desenvolvedor;
- Contextualizar o objetivo da conversa;
- Apresentar características do desenvolvimento que ele deve seguir.

Ficando assim o *template*:

Considere-se na posição de um desenvolvedor de software especialista na criação de códigos de Python para resolução de problemas de programação.

Você receberá um destes problemas e deve solucioná-lo criando um código em Python, respeitando a estrutura da versão 3.9. Faça comentários dentro do código.

A primeira parte será um enunciado da questão para entender o objetivo principal dela. Além disso, serão colocados exemplos de entradas e saídas esperadas, de forma com que você crie uma solução que funcione com diferentes entradas, não somente com as que estarão nos exemplos. Crie um código que faça a requisição dessas entradas assim como a função que solucione o problema.

Sua solução precisa atender exatamente ao que o problema requisitar, sem mais nem menos entradas ou impressões.

O problema terá quatro partes: o texto inicial (delimitado por 3 aspas simples no início e 3 aspas simples no final), as especificações de entrada e saída (que estarão delimitadas entre *tags XML*), os exemplos de entrada e saída (que estarão numeradas para que consiga entender qual entrada corresponde a qual saída), e as especificações de limites de tempo e memória (que estarão abaixo dos exemplos).

Segue o problema abaixo:

No entanto, de forma a padronizar a linguagem e tornar a comunicação em um só idioma, o template será colocado em inglês, para que o direcional para os modelos não tenham ruídos de linguagem, já que o enunciado das questões estará nesse idioma em questão. Ficando, dessa forma, assim:

Consider yourself in the position of a software developer expert in creating Python codes to solve programming problems.

You will receive one of these problems and must solve it by creating a code in Python, respecting the structure of version 3.8.10. Leave comments within the code.

The first part will be the statement of the question for you to understand the main point of it. In addition, there will be examples of inputs and outputs that are expected, so you may create a solution that works with differents inputs, not only with the ones in the examples. Create a code that requests the inputs as well as the function that solves the problem.

Your solution needs to meet exactly what the problem requires, with no more or less inputs or prints.

The problem will have four parts: the initial text (delimited by 3 quotation marks single at the beginning and 3 single quotes at the end), the input and output specifications (which will be delimited between XML tags), the input and output examples (which will be numbered so you can understand which entry corresponds to which output), you'll also receive the time and memory limits specifications (which will be below examples) that must not be exceeded. Some problems may have a "Notes" section, containing some extra information to help you solving the problem.

This is the programming problem:

Após isso, tem-se a inserção do problema, juntamente da especificação das entradas e saídas, e seus exemplos, na seguinte estrutura:

- Texto inicial (delimitado por 3 aspas simples no início e 3 aspas simples no final);
- Especificações das entradas e saídas (delimitadas entre *tags XML*).
- Exemplos de entradas e saídas (especificadas com números de correspondência).
- Limite de tempo e memória (especificadas logo abaixo dos exemplos de entradas e saídas).
- Notas extras (especificadas logo abaixo dos limites de tempo e memória).

Dessa forma, atinge-se o modelo ideal para mitigar qualquer desentendimento com o LLM.

3.4 Definição dos problemas de programação

Neste estudo, será realizado um processo de avaliação comparativa do desempenho de diferentes códigos de solução para uma série de problemas de programação. A abor-

dagem adotada visou analisar e comparar as implementações propostas pelos modelos de linguagem de inteligência artificial escolhidos para este trabalho.

Para isso, foram selecionados dez problemas distintos, cada um com suas características próprias, desafiando a capacidade do modelo de linguagem em oferecer soluções precisas e eficientes para diferentes níveis de problema.

Todos os problemas foram retirados da plataforma Codeforces e estão divididos em categorias diferentes e em níveis de dificuldade diferentes. Em uma de escala de 800 a 3500, onde 800 representa um nível fácil e 3500 um nível extremamente difícil, de acordo com a divisão do próprio juiz online.

A escolha da dificuldade e da categoria da questão foi feita a partir de um desejo de misturar diferentes níveis e tipos, de forma com que a análise pudesse ser feita com variações diversas. As questões possuirão tais dificuldades e categorias:

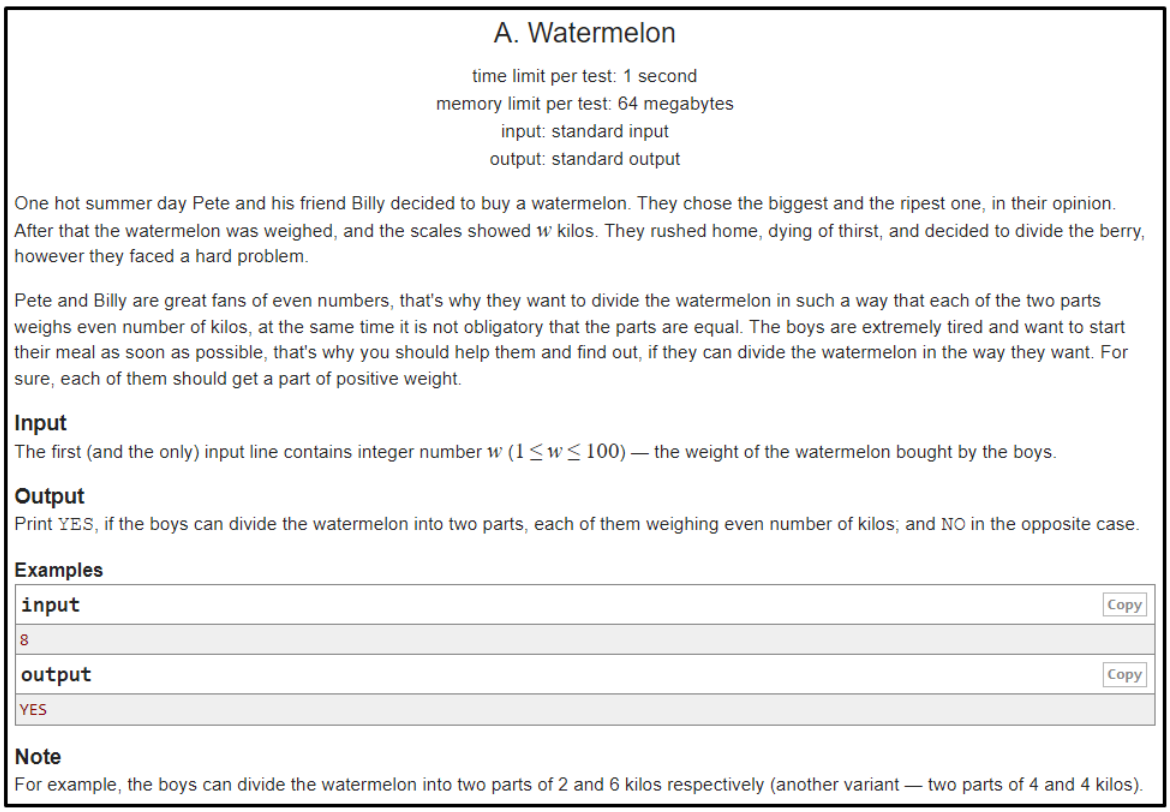
- Questão 1: Nível 800, categoria(s): ”Força Bruta” e ”Matemática”;
- Questão 2: Nível 800, categoria(s): ”Strings”;
- Questão 3: Nível 1000, categoria(s): ”Strings” e ”Implementação”;
- Questão 4: Nível 1100, categoria(s): ”Ordenação”;
- Questão 5: Nível 1200, categoria(s): ”Implementação”;
- Questão 6: Nível 1300, categoria(s): ”Implementação”, ”*Hashing*” e ”Estrutura de Dados”;
- Questão 7: Nível 1500, categoria(s): ”Programação Dinâmica”;
- Questão 8: Nível 2000, categoria(s): ”Programação Dinâmica” e ”Matemática”;
- Questão 9: Nível 2300, categoria(s): ”Estrutura de Dados” e ”Algoritmo de busca em profundidade” e ”Estrutura de Dados União-Busca” e ”Árvores”;
- Questão 10: Nível 3500, categoria(s): ”Combinatória” e ”Algoritmos Construtivos” e ”Matemática”.

A escolha das dificuldades e das categorias se deu pela importância de se analisar os modelos com diferentes níveis e categorias de problemas. É importante entender se de fato os modelos irão se sobressair com questões mais fáceis ou se isso não é algo que realmente importa. Também ajudará na compreensão do entendimento destes modelos quanto a categorias diferentes. É importante julgar se um modelo terá um desempenho superior ou inferior em questões de matemática, por exemplo.

Tabela 2 – Problemas a serem trabalhados

Identificador	**Nome**	**Número da figura**	**Nível**
01	Watermelon	Figura 4	800
02	Way Too Long Words	Figura 5	800
03	String Task	Figura 6	1000
04	Laptops	Figura 7	1100
05	Cheap Travel	Figura 8	1200
06	Registration System	Figura 9	1300
07	Boredom	Figura 10	1500
08	The Least Round Way	Figura 11	2000
09	Lomsat Gelral	Figura 12	2300
10	Omkar and Mosaic	Figura 13 e 14	3500

Figura 4 – Problema de Programação 01: Watermelon



A. Watermelon

time limit per test: 1 second
memory limit per test: 64 megabytes
input: standard input
output: standard output

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem.

Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out, if they can divide the watermelon in the way they want. For sure, each of them should get a part of positive weight.

Input
The first (and the only) input line contains integer number w ($1 \le w \le 100$) — the weight of the watermelon bought by the boys.

Output
Print YES, if the boys can divide the watermelon into two parts, each of them weighing even number of kilos; and NO in the opposite case.

Examples

input	Copy
8	
output	Copy
YES	

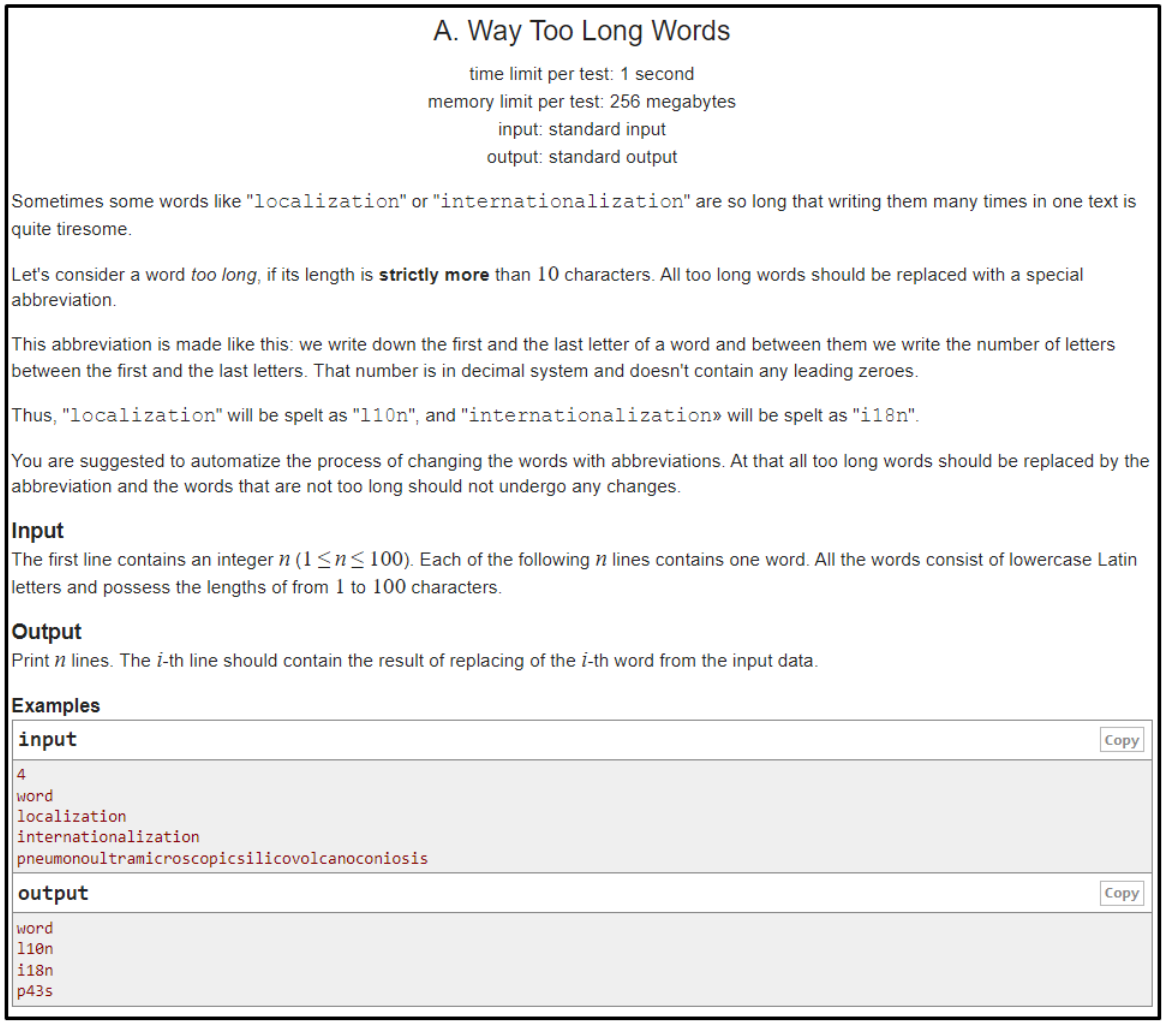
Note
For example, the boys can divide the watermelon into two parts of 2 and 6 kilos respectively (another variant — two parts of 4 and 4 kilos).

Fonte: Exercício - Codeforces

Na tabela 2 e nas figuras a partir da 4, estão representados todos os problemas a serem trabalhos neste estudo:

Com a definição dos problemas de programação, toda a estrutura está pronta para ser aplicada aos LLMs e assim enviar às ferramentas, recolher os resultados e analisar as métricas.

Figura 5 – Problema de Programação 02: Way Too Long Words



A. Way Too Long Words

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Sometimes some words like "`localization`" or "`internationalization`" are so long that writing them many times in one text is quite tiresome.

Let's consider a word *too long*, if its length is **strictly more** than 10 characters. All too long words should be replaced with a special abbreviation.

This abbreviation is made like this: we write down the first and the last letter of a word and between them we write the number of letters between the first and the last letters. That number is in decimal system and doesn't contain any leading zeroes.

Thus, "`localization`" will be spelt as "`l10n`", and "`internationalization`» will be spelt as "`i18n`".

You are suggested to automatize the process of changing the words with abbreviations. At that all too long words should be replaced by the abbreviation and the words that are not too long should not undergo any changes.

Input

The first line contains an integer n ($1 \le n \le 100$). Each of the following n lines contains one word. All the words consist of lowercase Latin letters and possess the lengths of from 1 to 100 characters.

Output

Print n lines. The i-th line should contain the result of replacing of the i-th word from the input data.

Examples

input Copy

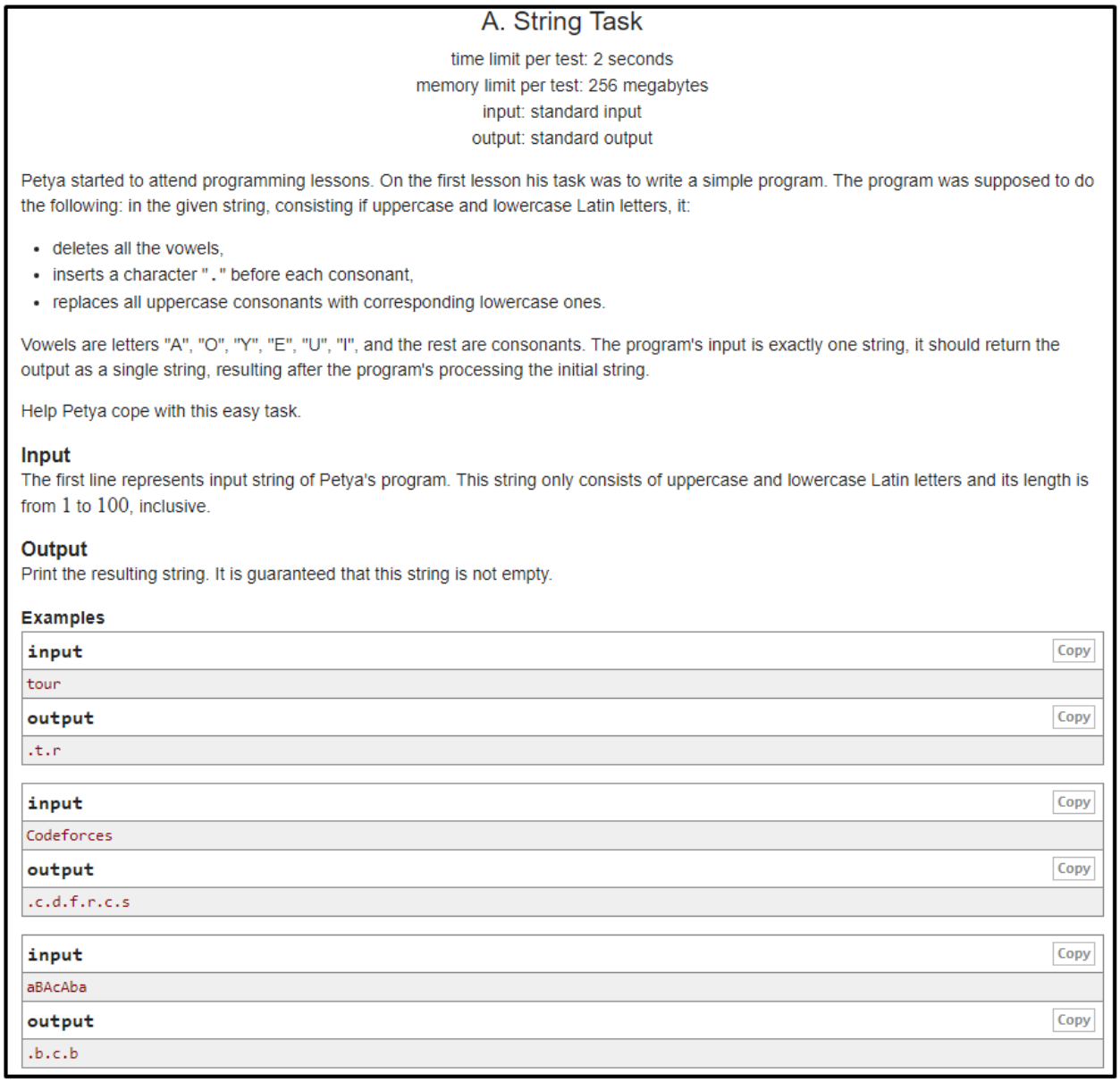
```
4
word
localization
internationalization
pneumonoultramicroscopicsilicovolcanoconiosis
```

output Copy

```
word
l10n
i18n
p43s
```

Fonte: Exercício - Codeforces

Figura 6 – Problema de Programação 03: String Task



A. String Task

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Petya started to attend programming lessons. On the first lesson his task was to write a simple program. The program was supposed to do the following: in the given string, consisting if uppercase and lowercase Latin letters, it:

- deletes all the vowels,
- inserts a character "." before each consonant,
- replaces all uppercase consonants with corresponding lowercase ones.

Vowels are letters "A", "O", "Y", "E", "U", "I", and the rest are consonants. The program's input is exactly one string, it should return the output as a single string, resulting after the program's processing the initial string.

Help Petya cope with this easy task.

Input

The first line represents input string of Petya's program. This string only consists of uppercase and lowercase Latin letters and its length is from 1 to 100, inclusive.

Output

Print the resulting string. It is guaranteed that this string is not empty.

Examples

input	Copy
tour	
output	Copy
.t.r	

input	Copy
Codeforces	
output	Copy
.c.d.f.r.c.s	

input	Copy
aBAcAba	
output	Copy
.b.c.b	

Fonte: Exercício - Codeforces

Figura 7 – Problema de Programação 04: Laptops

A. Laptops

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

One day Dima and Alex had an argument about the price and quality of laptops. Dima thinks that the more expensive a laptop is, the better it is. Alex disagrees. Alex thinks that there are two laptops, such that the price of the first laptop is less (strictly smaller) than the price of the second laptop but the quality of the first laptop is higher (strictly greater) than the quality of the second laptop.

Please, check the guess of Alex. You are given descriptions of n laptops. Determine whether two described above laptops exist.

Input

The first line contains an integer n $(1 \leq n \leq 10^5)$ — the number of laptops.

Next n lines contain two integers each, a_i and b_i $(1 \leq a_i, b_i \leq n)$, where a_i is the price of the i-th laptop, and b_i is the number that represents the quality of the i-th laptop (the larger the number is, the higher is the quality).

All a_i are distinct. All b_i are distinct.

Output

If Alex is correct, print "`Happy Alex`", otherwise print "`Poor Alex`" (without the quotes).

Examples

input	Copy
2 1 2 2 1	
output	**Copy**
Happy Alex	

Fonte: Exercício - Codeforces

Figura 8 – Problema de Programação 05: Cheap Travel

A. Cheap Travel

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Ann has recently started commuting by subway. We know that a one ride subway ticket costs a rubles. Besides, Ann found out that she can buy a special ticket for m rides (she can buy it several times). It costs b rubles. Ann did the math; she will need to use subway n times. Help Ann, tell her what is the minimum sum of money she will have to spend to make n rides?

Input

The single line contains four space-separated integers n, m, a, b ($1 \le n, m, a, b \le 1000$) — the number of rides Ann has planned, the number of rides covered by the m ride ticket, the price of a one ride ticket and the price of an m ride ticket.

Output

Print a single integer — the minimum sum in rubles that Ann will need to spend.

Examples

input	Copy
6 2 1 2	
output	Copy
6	

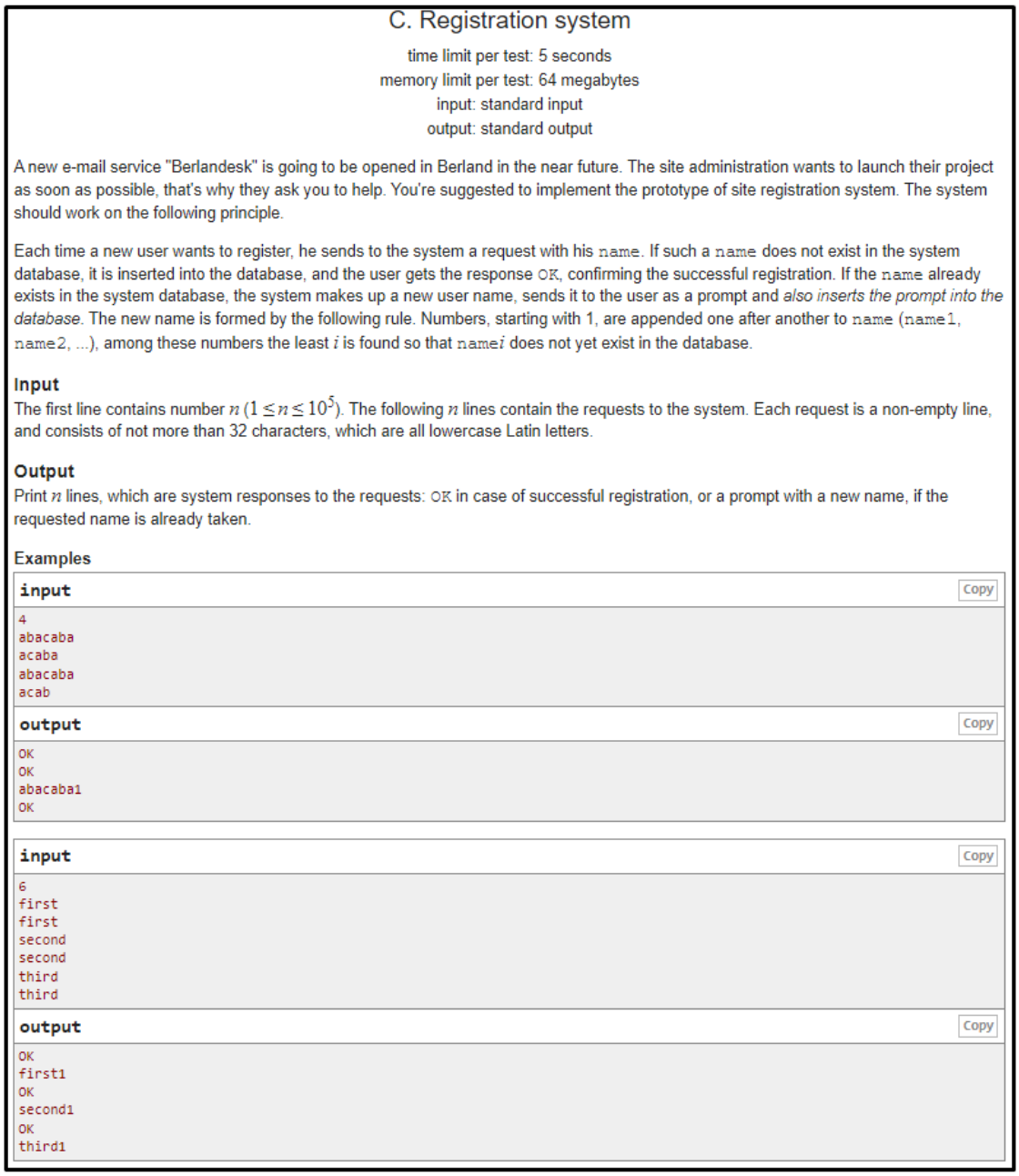
input	Copy
5 2 2 3	
output	Copy
8	

Note

In the first sample one of the optimal solutions is: each time buy a one ride ticket. There are other optimal solutions. For example, buy three m ride tickets.

Fonte: Exercício - Codeforces

Figura 9 – Problema de Programação 06: Registration system



C. Registration system

time limit per test: 5 seconds
memory limit per test: 64 megabytes
input: standard input
output: standard output

A new e-mail service "Berlandesk" is going to be opened in Berland in the near future. The site administration wants to launch their project as soon as possible, that's why they ask you to help. You're suggested to implement the prototype of site registration system. The system should work on the following principle.

Each time a new user wants to register, he sends to the system a request with his `name`. If such a `name` does not exist in the system database, it is inserted into the database, and the user gets the response `OK`, confirming the successful registration. If the `name` already exists in the system database, the system makes up a new user name, sends it to the user as a prompt and *also inserts the prompt into the database*. The new name is formed by the following rule. Numbers, starting with 1, are appended one after another to `name` (`name1`, `name2`, ...), among these numbers the least i is found so that `name`i does not yet exist in the database.

Input
The first line contains number n ($1 \le n \le 10^5$). The following n lines contain the requests to the system. Each request is a non-empty line, and consists of not more than 32 characters, which are all lowercase Latin letters.

Output
Print n lines, which are system responses to the requests: `OK` in case of successful registration, or a prompt with a new name, if the requested name is already taken.

Examples

input Copy

```
4
abacaba
acaba
abacaba
acab
```

output Copy

```
OK
OK
abacaba1
OK
```

input Copy

```
6
first
first
second
second
third
third
```

output Copy

```
OK
first1
OK
second1
OK
third1
```

Fonte: Exercício - Codeforces

Figura 10 – Problema de Programação 07: Boredom

A. Boredom

time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output

Alex doesn't like boredom. That's why whenever he gets bored, he comes up with games. One long winter evening he came up with a game and decided to play it.

Given a sequence a consisting of n integers. The player can make several steps. In a single step he can choose an element of the sequence (let's denote it a_k) and delete it, at that all elements equal to $a_k + 1$ and $a_k - 1$ also must be deleted from the sequence. That step brings a_k points to the player.

Alex is a perfectionist, so he decided to get as many points as possible. Help him.

Input
The first line contains integer n ($1 \le n \le 10^5$) that shows how many numbers are in Alex's sequence.

The second line contains n integers $a_1, a_2, ..., a_n$ ($1 \le a_i \le 10^5$).

Output
Print a single integer — the maximum number of points that Alex can earn.

Examples

input	Copy
2 1 2	
output	Copy
2	

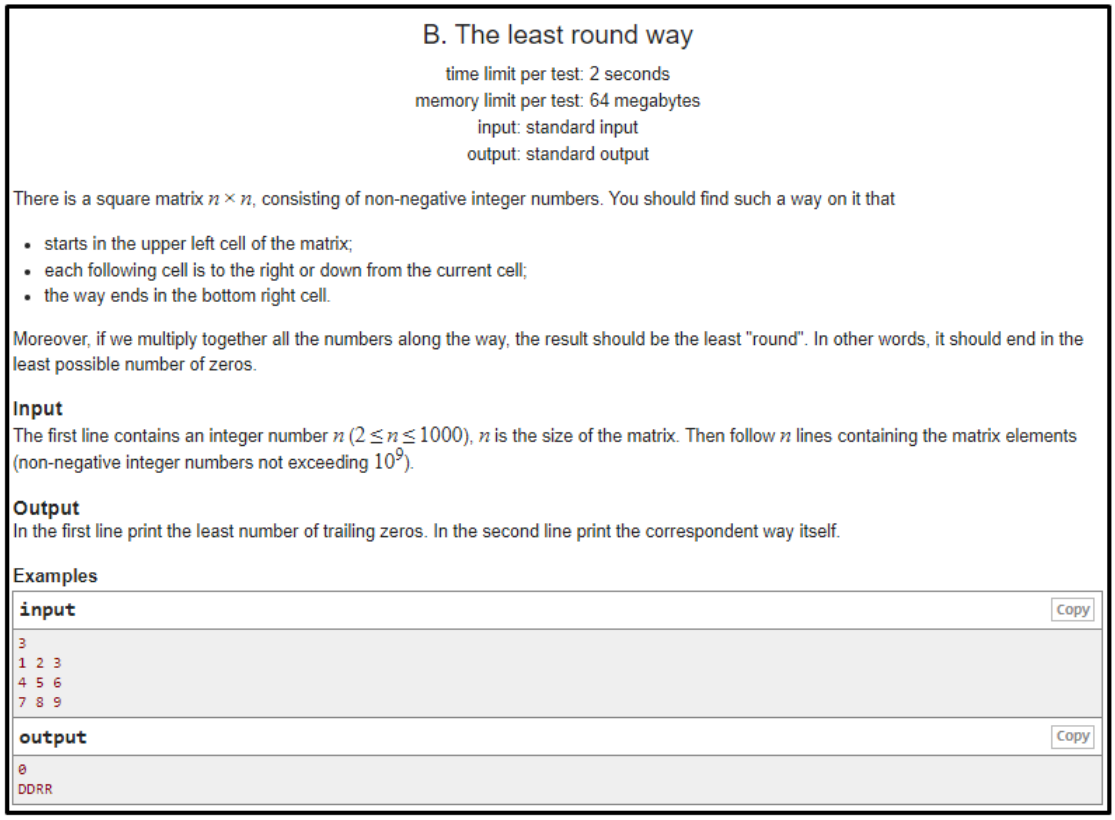
input	Copy
3 1 2 3	
output	Copy
4	

input	Copy
9 1 2 1 3 2 2 2 2 3	
output	Copy
10	

Note
Consider the third test example. At first step we need to choose any element equal to 2. After that step our sequence looks like this [2, 2, 2, 2]. Then we do 4 steps, on each step we choose any element equals to 2. In total we earn 10 points.

Fonte: Exercício - Codeforces

Figura 11 – Problema de Programação 08: The least round way



B. The least round way

time limit per test: 2 seconds
memory limit per test: 64 megabytes
input: standard input
output: standard output

There is a square matrix $n \times n$, consisting of non-negative integer numbers. You should find such a way on it that

- starts in the upper left cell of the matrix;
- each following cell is to the right or down from the current cell;
- the way ends in the bottom right cell.

Moreover, if we multiply together all the numbers along the way, the result should be the least "round". In other words, it should end in the least possible number of zeros.

Input
The first line contains an integer number n ($2 \le n \le 1000$), n is the size of the matrix. Then follow n lines containing the matrix elements (non-negative integer numbers not exceeding 10^9).

Output
In the first line print the least number of trailing zeros. In the second line print the correspondent way itself.

Examples

input Copy

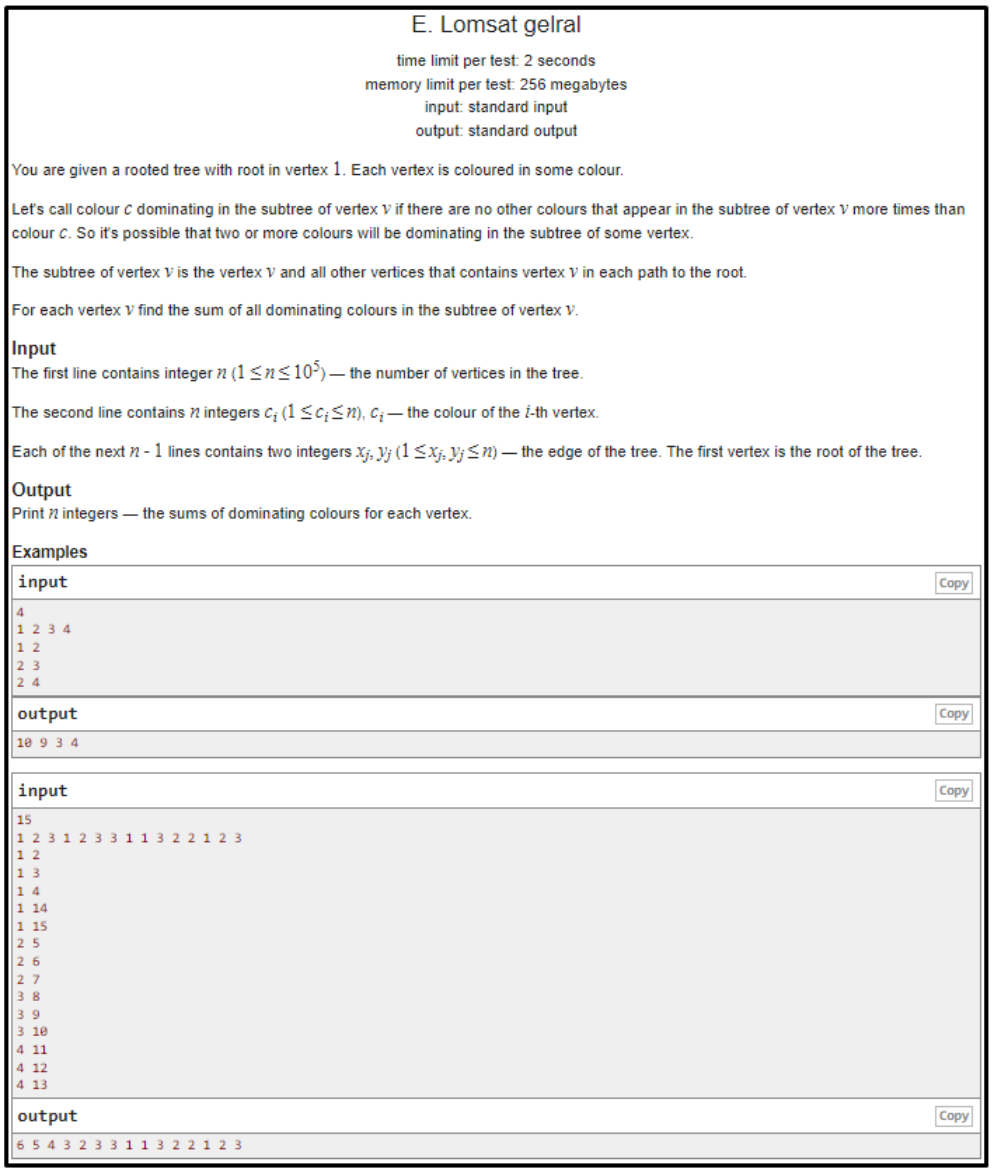
```
3
1 2 3
4 5 6
7 8 9
```

output Copy

```
0
DDRR
```

Fonte: Exercício - Codeforces

Figura 12 – Problema de Programação 09: Lomsat gelral



E. Lomsat gelral

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

You are given a rooted tree with root in vertex 1. Each vertex is coloured in some colour.

Let's call colour c dominating in the subtree of vertex v if there are no other colours that appear in the subtree of vertex v more times than colour c. So it's possible that two or more colours will be dominating in the subtree of some vertex.

The subtree of vertex v is the vertex v and all other vertices that contains vertex v in each path to the root.

For each vertex v find the sum of all dominating colours in the subtree of vertex v.

Input

The first line contains integer n ($1 \le n \le 10^5$) — the number of vertices in the tree.

The second line contains n integers c_i ($1 \le c_i \le n$), c_i — the colour of the i-th vertex.

Each of the next $n - 1$ lines contains two integers x_j, y_j ($1 \le x_j, y_j \le n$) — the edge of the tree. The first vertex is the root of the tree.

Output

Print n integers — the sums of dominating colours for each vertex.

Examples

input Copy

```
4
1 2 3 4
1 2
2 3
2 4
```

output Copy

```
10 9 3 4
```

input Copy

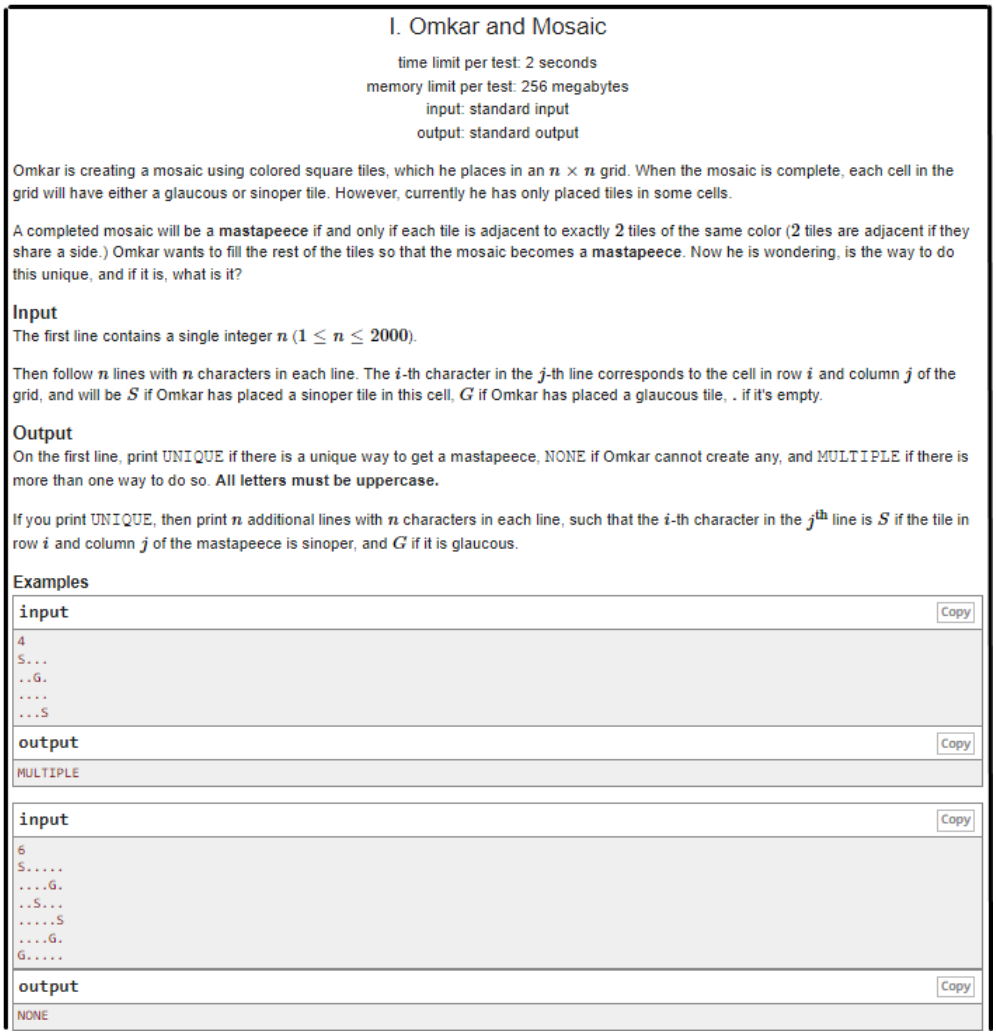
```
15
1 2 3 1 2 3 3 1 1 3 2 2 1 2 3
1 2
1 3
1 4
1 14
1 15
2 5
2 6
2 7
3 8
3 9
3 10
4 11
4 12
4 13
```

output Copy

```
6 5 4 3 2 3 3 1 1 3 2 2 1 2 3
```

Fonte: Exercício - Codeforces

Figura 13 – Problema de Programação 10: Omkar and Mosaic



I. Omkar and Mosaic

time limit per test: 2 seconds
memory limit per test: 256 megabytes
input: standard input
output: standard output

Omkar is creating a mosaic using colored square tiles, which he places in an $n \times n$ grid. When the mosaic is complete, each cell in the grid will have either a glaucous or sinoper tile. However, currently he has only placed tiles in some cells.

A completed mosaic will be a **mastapeece** if and only if each tile is adjacent to exactly 2 tiles of the same color (2 tiles are adjacent if they share a side.) Omkar wants to fill the rest of the tiles so that the mosaic becomes a **mastapeece**. Now he is wondering, is the way to do this unique, and if it is, what is it?

Input

The first line contains a single integer n ($1 \leq n \leq 2000$).

Then follow n lines with n characters in each line. The i-th character in the j-th line corresponds to the cell in row i and column j of the grid, and will be S if Omkar has placed a sinoper tile in this cell, G if Omkar has placed a glaucous tile, . if it's empty.

Output

On the first line, print `UNIQUE` if there is a unique way to get a mastapeece, `NONE` if Omkar cannot create any, and `MULTIPLE` if there is more than one way to do so. **All letters must be uppercase.**

If you print `UNIQUE`, then print n additional lines with n characters in each line, such that the i-th character in the j^{th} line is S if the tile in row i and column j of the mastapeece is sinoper, and G if it is glaucous.

Examples

input Copy

```
4
S...
..G.
....
...S
```

output Copy

```
MULTIPLE
```

input Copy

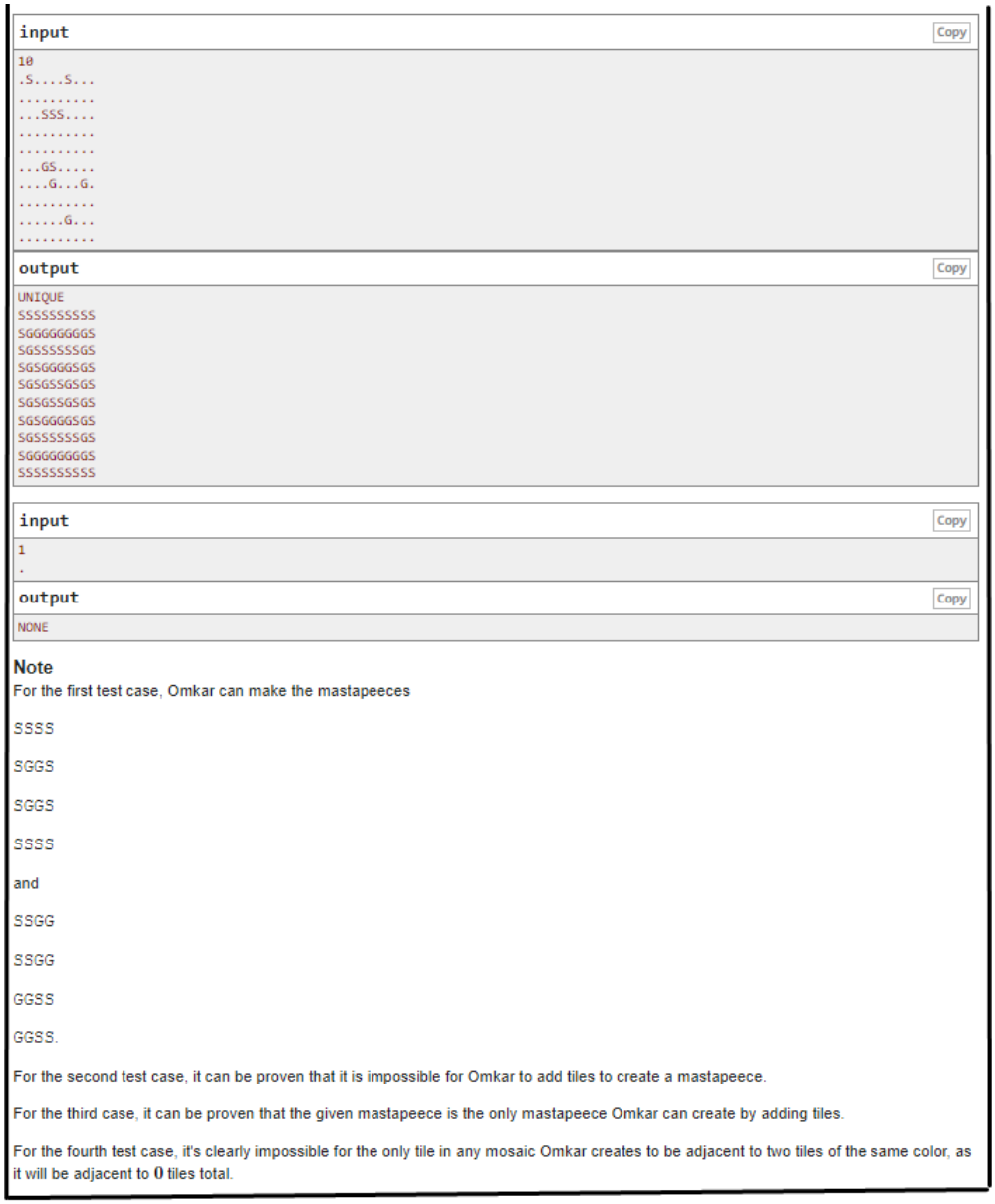
```
6
S.....
....G.
..S...
.....S
....G.
G.....
```

output Copy

```
NONE
```

Fonte: Exercício - Codeforces

Figura 14 – Problema de Programação 10 - Continuação: Omkar and Mosaic



input Copy

```
10
.S....S...
..........
...SSS....
..........
..........
...GS.....
....G...G.
..........
......G...
..........
```

output Copy

```
UNIQUE
SSSSSSSSSS
SGGGGGGGGS
SGSSSSSSGS
SGSGGGGSGS
SGSGSSGSGS
SGSGSSGSGS
SGSGGGGSGS
SGSSSSSSGS
SGGGGGGGGS
SSSSSSSSSS
```

input Copy

```
1
.
```

output Copy

```
NONE
```

Note

For the first test case, Omkar can make the mastapeeces

```
SSSS
SGGS
SGGS
SSSS
```

and

```
SSGG
SSGG
GGSS
GGSS
```

For the second test case, it can be proven that it is impossible for Omkar to add tiles to create a mastapeece.

For the third case, it can be proven that the given mastapeece is the only mastapeece Omkar can create by adding tiles.

For the fourth test case, it's clearly impossible for the only tile in any mosaic Omkar creates to be adjacent to two tiles of the same color, as it will be adjacent to **0** tiles total.

Fonte: Exercício - Codeforces

4 Análises e Resultados

4.1 Introdução

Este capítulo aborda as principais análises sobre os resultados gerados, aprofundando nas respostas dos modelos aos problemas propostos. Serão avaliadas cada uma das métricas selecionadas, de forma a entender qual modelo se superou em exatidão, eficiência de tempo e memória, presença de modularização e comentários no código, complexidade cognitiva e manutenabilidade. Dessa forma, os principais pontos de cada métrica serão destacados e analisados. As respostas dos modelos podem ser encontradas em um repositório na ferramenta do GitHub[1].

Nas seções abaixo, estarão representadas visões mais específicas para cada métrica, a fim de entender quais códigos dos modelos conseguiram atender o requisitado de forma positiva, e quais não conseguiu.

4.2 Exatidão

A exatidão é a métrica que avalia de forma objetiva se a solução é suficiente para o que pede a questão. Os resultados foram coletados e analisados de acordo com os vereditos recebidos por cada resposta de cada modelo.

Na tabela 3, temos uma análise geral dos modelos onde estão representados os vereditos de cada resposta dos modelos para cada questão de programação, a fim de apresentar quais modelos tiveram êxito nas questões, ao apresentar soluções em código corretas.

Os resultados aparentam indicar que o modelo do GPT foi o mais eficaz, com uma taxa de acerto de 60%, destacando-se aparentemente como o melhor modelo entre os cinco analisados. Ele conseguiu acertar 6 das 10 questões apresentadas, sendo as questões 1, 2, 4, 6, 7 e 8. Em segundo lugar, empatados com uma taxa de acerto de 20%, temos os modelos Llama3 e H2o, cada um acertando 2 questões, sendo essas as Questões 2 e 6 para ambos os modelos.

Os modelos Gemini e Gemma apresentaram os piores desempenhos, cada um com uma taxa de acerto de 10%, acertando apenas a Questão 2. Esses resultados sugerem que há uma diferença significativa na capacidade dos modelos em resolver problemas de programação, com o GPT demonstrando um desempenho aparentemente superior.

[1] https://github.com/

Tabela 3 – Veredito das respostas dos modelos

Questão	**Nível**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Acerto (%)**
01	800	AC	WA	WA	WA	WA	**20%**
02	800	AC	AC	AC	AC	AC	**100%**
03	1000	WA	WA	WA	WA	AC	**20%**
04	1100	AC	TLE	AC	RE	WA	**40%**
05	1200	AC	WA	WA	WA	WA	**20%**
06	1300	AC	TLE	TLE	TLE	WA	**20%**
07	1500	AC	WA	RE	WA	WA	**20%**
08	2000	RE	RE	WA	TLE	WA	**0%**
09	2300	RE	WA	RE	RE	RE	**0%**
10	3500	WA	RE	WA	WA	RE	**0%**
Acerto (%)	-	**60%**	**10%**	**20%**	**10%**	**20%**	**24%**

Ao analisar as questões em si, verificamos que a Questão 2 (Nível 800) foi a mais fácil, com uma taxa de acerto de 100%. Todos os modelos conseguiram responder corretamente a esta questão. A segunda questão mais fácil foi a Questão 4 (Nível 1100), que teve uma taxa de acerto de 40%, sendo acertada pelos modelos GPT e H2o. Em contrapartida, as Questões 8 (Nível 2000), 9 (Nível 2300) e 10 (Nível 3500) foram as mais difíceis, com nenhum dos modelos conseguindo a resposta correta, resultando em uma taxa de acerto de 0%.

Também é importante considerar que das duas questões de nível 800 (indicadas como mais fáceis) apenas uma teve êxito completo (100% de acerto) por parte dos modelos, enquanto a outra teve uma performance tão abaixo quanto questões de níveis maiores. O problema de programação 01, apesar de ser considerado fácil por se tratar de um cálculo simples a ser feito, tem uma espécie de ”pegadinha”, causando uma ilusão tanto nos modelos quanto em programadores mais experientes.

A questão fala de uma divisão de um número qualquer (entrada do usuário) pelo número 2. Caso o quociente fosse um número par, deveria imprimir ”YES”, caso não, deveria imprimir ”NO”. Dessa forma, a maioria dos modelos simplesmente fez a verificação se o número de entrada era par. Se sim, imprimiam ”YES”. Se não, imprimiam ”NO”. Por terem essa única condição, falharam em um caso-teste que quebra a solução que é quando o número de entrada era o próprio 2. Nesse caso, quando o 2 é dividido por um outro 2, o quociente é o número 1, ou seja, ímpar, tornando diversas soluções inválidas. O GPT, no entanto, faz a verificação se o número é tanto par quanto maior que 2, tendo a solução ideal para a questão e sendo o único a acertá-la.

Dessa forma, tais análises podem indicar que os modelos em sua maioria não lidam

tão bem com problemas que possam iludi-los ou que tenham casos-teste muito específicos, apesar de gerarem soluções muito próximos ao esperado.

A porcentagem de acertos para cada questão indica que não necessariamente os modelos sempre vão performar melhor em questões mais fáceis do que em questões mais difíceis, já que a questão 04 (nível 1100) teve um desempenho maior (40%) do que a questão 01 (nível 800) e questão 03 (nível 1000) que tiveram performances inferiores (ambas com 20%). No entanto, apesar dessas exceções apontadas, pode-se entender que é uma análise justa a ser feita, já que as questões consideradas mais difíceis não tiveram nenhum acerto, indicando certa dificuldade dos modelos para solucionar problemas que exijam um conhecimento maior tanto em programação de forma geral quanto aos assuntos das categorias abordadas por elas: estrutura de dados, algoritmo de busca em profundidade, árvores, entre outros.

O desempenho geral dos modelos indica que o GPT possui uma capacidade superior na resolução de problemas de programação em Python quando a métrica é a exatidão, enquanto os modelos Llama3 e H2o apresentam um desempenho intermediário, comparado aos demais. Já os modelos Gemini e Gemma indicaram ter as piores performances com seus 10% de acerto cada uma. Observa-se também que as questões de níveis mais fáceis são mais acessíveis para os modelos de linguagem, enquanto questões mais complexas representam um aumento considerável na dificuldade, com a maioria dos modelos não conseguindo resolver as questões de nível 1300 e superiores.

4.3 Eficiência de tempo

Nesta seção, discute-se a eficiência temporal dos algoritmos gerados. A análise aqui envolve compreender se o tempo de execução das soluções foi cumprido. A eficiência de tempo é vital em contextos onde há restrições rigorosas de tempo, que é o caso de problemas de programação.

A tabela 4 apresenta os resultados da eficiência de tempo dos modelos de linguagem na resolução dos problemas de programação. A análise se concentra em verificar se os códigos gerados cumpriram os limites de tempo estabelecidos para cada questão.

Esses dados mostram que os modelos GPT e Llama3 indicam ter se destacado com um desempenho perfeito, cumprindo todos os limites de tempo estabelecidos para cada questão (100% de acerto). Atrás deles, o modelo H2o também apresenta uma alta taxa de acerto (90%), falhando apenas em uma questão de maior dificuldade (de nível 1300).

Os modelos Gemini e Gemma aparentam ter tido mais dificuldades, com ambos alcançando 80% de acerto.

Nas demais questões, houve uma variação de 80% 100%, o que apresenta que

Tabela 4 – Cumpriu requisito de limite de tempo

Questão	**Nível**	**Limite de tempo (em ms)**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	1.000	Sim (154ms)	Sim (124ms)	Sim (124ms)	Sim (124ms)	Sim (154ms)	**100%**
02	800	1.000	Sim (77ms)	Sim (62ms)	Sim (77ms)	Sim (62ms)	Sim (92ms)	**100%**
03	1000	2.000	Sim (122ms)	Sim (92ms)	Sim (154ms)	Sim (186ms)	Sim (156ms)	**100%**
04	1100	1.000	Sim (328ms)	Não (1000ms)	Sim (234ms)	Sim (46ms)	Sim (93ms)	**80%**
05	1200	1.000	Sim (77ms)	Sim (62ms)	Sim (62ms)	Sim (77ms)	Sim (77ms)	**100%**
06	1300	5.000	Sim (1.186ms)	Não (5.000ms)	Não (5.000ms)	Não (5.000ms)	Sim (124ms)	**40%**
07	1500	1.000	Sim (187ms)	Sim (62ms)	Sim (46ms)	Sim (62ms)	Sim (92ms)	**100%**
08	2000	2.000	Sim (46ms)	Sim (61ms)	Sim (62ms)	Não (2.000ms)	Sim (62ms)	**80%**
09	2300	2.000	Sim (61ms)	Sim (46ms)	Sim (734ms)	Sim (61ms)	Sim (77ms)	**100%**
10	3500	2.000	Sim (61ms)	Sim (62ms)	Sim (62ms)	Sim (62ms)	Sim (46ms)	**100%**
Média	-	-	**100%**	**80%**	**90%**	**80%**	**100%**	**90%**

todos os modelos apresentaram soluções eficientes em relação ao tempo de execução, com pequenas exceções.

A questão 06, por exemplo, indica ter sido a com maior complexidade, já que falhou em 3 dos 5 modelos. Isso pode apresentar dela ser uma questão que exige de conhecimentos mais técnicos para inibir o problema do tempo e que tem casos-teste mais complexos e maiores. A questão, no caso, possui um alto limite de 5 segundos (o maior entre as demais questões), mas mesmo assim correspondeu negativamente na maioria das soluções propostas. A entrada da questão pode ser um número de até 100.000, de forma com que qualquer laço no código pode gerar uma solução lenta para casos-teste maiores.

Vale salientar que não necessariamente um modelo é superior ao outro só por não ter errado pela execução em tempo maior que o limite permitido, mas por outro motivo. Ambos falharam na métrica de exatidão, mas é importante considerar que alguns modelos podem ter absorvido a orientação do limite de tempo com mais atenção.

4.4 Eficiência de memória

A seção de eficiência de memória avalia o uso de memória pelas soluções geradas. Esta métrica é importante para garantir que os algoritmos não apenas resolvam os problemas, mas o façam de maneira eficiente em termos de recursos computacionais.

A tabela 5 avalia se os códigos gerados pelos modelos cumpriram os limites de memória estabelecidos para cada questão.

Tabela 5 – Cumpriu requisito de limite de memória

Questão	**Nível**	**Limite de memória (em kb)**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	64.000	Sim (24kb)	Sim (4kb)	Sim (32kb)	Sim (4kb)	Sim (8kb)	**100%**
02	800	256.000	Sim (20kb)	Sim (20kb)	Sim (16kb)	Sim (16kb)	Sim (12kb)	**100%**
03	1000	256.000	Sim (0kb)	Sim (0kb)	Sim (20kb)	Sim (8kb)	Sim (12kb)	**100%**
04	1100	256.000	Sim (14.328kb)	Sim (11.648kb)	Sim (15.104kb)	Sim (16kb)	Sim (0kb)	**100%**
05	1200	256.000	Sim (12kb)	Sim (4kb)	Sim (4kb)	Sim (0kb)	Sim (12kb)	**100%**
06	1300	64.000	Sim (13.996kb)	Sim (5.248kb)	Sim (5.212kb)	Sim (4.468kb)	Sim (4kb)	**100%**
07	1500	256.000	Sim (13.068kb)	Sim (4kb)	Sim (4kb)	Sim (8kb)	Sim (12kb)	**100%**
08	2000	64.000	Sim (0kb)	Sim (0kb)	Sim (4kb)	Sim (4kb)	Sim (0kb)	**100%**
09	2300	256.000	Sim (4kb)	Sim (0kb)	Sim (39.756kb)	Sim (0kb)	Sim (8kb)	**100%**
10	3500	256.000	Sim (28kb)	Sim (8kb)	Sim (16kb)	Sim (0kb)	Sim (8kb)	**100%**
Média	-	-	**100%**	**100%**	**100%**	**100%**	**100%**	**100%**

Pelo apresentado, todos os modelos aparentam ter cumprido os requisitos de memória em todas as questões, apresentando 100% de acerto.

Não houve variação entre os modelos nesta métrica, indicando que todos seriam igualmente eficazes em termos de uso de memória, assim como todas as questões teriam apresentado uma performance igual.

4.5 Modularização

A análise da modularização envolve verificar a estrutura dos códigos gerados, avaliando se eles são divididos em funções ou módulos bem definidos. A modularização facilita a manutenção e reutilização do código.

A tabela 6 avalia a modularização dos códigos gerados, verificando a presença de funções bem definidas.

Tabela 6 – Mínimo de funções no código (quantidade)

Questão	**Nível**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	Não (0)	Sim (2)	Sim (1)	Não (0)	Não (0)	**40%**
02	800	Sim (1)	Sim (2)	Sim (1)	Não (0)	Sim (1)	**80%**
03	1000	Sim (1)	Sim (1)	Sim (1)	Sim (1)	Sim (1)	**100%**
04	1100	Sim (1)	Sim (1)	Sim (1)	Não (0)	Não (0)	**60%**
05	1200	Não (0)	Sim (1)	Sim (1)	Não (0)	Não (0)	**40%**
06	1300	Não (1)	Sim (1)	Sim (1)	Não (0)	Sim (1)	**60%**
07	1500	Sim (1)	Sim (1)	Sim (1)	Não (0)	Não (0)	**60%**
08	2000	Sim (1)	Sim (1)	Sim (1)	Não (0)	Sim (1)	**80%**
09	2300	Sim (2)	Sim (2)	Sim (2)	Sim (1)	Sim (2)	**100%**
10	3500	Sim (1)	Sim (2)	Sim (4)	Sim (3)	Sim (2)	**100%**
Média	-	**70%**	**100%**	**100%**	**30%**	**60%**	**72%**

De forma inicial, os modelos Gemini e H2o tiveram uma porcentagem de 100% de utilização de funções em seus códigos, aparentemente se destacando aos demais. Por outro lado, o Gemma aparentou ter um desempenho abaixo com 30% de utilização em seus códigos.

Além dele, o Llama3 não teve muita utilização de funções em suas soluções propostas, com 60% de frequência.

A tabela também indica que questões mais difíceis (níveis de 2000-3500) tiveram uma alta frequência de modularização, sugerindo que esses modelos utilizem tais recursos em questões mais complexas, já que em problemas de programação mais simples não há tanta necessidade de se utilizarem funções — pela facilidade em se entender, manter e reutilizar o código.

A questão 01, por exemplo, só teve 40% de utilização de funções. Isso pode demonstrar a falta de funções em questões que exijam menos sua presença.

De forma geral, os modelos parecem ter utilizado a modularização com certa frequência, resultando em 72% em média.

4.6 Comentários no código

Aqui, a análise se concentra na presença e qualidade dos comentários no código. Comentários são essenciais para a compreensão e manutenção do código por outros desenvolvedores.

A tabela 7 avalia a presença dos comentários em porcentagem nos códigos gerados.

Tabela 7 – Presença de comentários no código (%)

Questão	**Nível**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	Sim (28%)	Sim (58%)	Sim (50%)	Sim (50%)	Não (0%)	**80%**
02	800	Sim (16%)	Sim (50%)	Sim (50%)	Não (0%)	Sim (18%)	**80%**
03	1000	Sim (38%)	Sim (58%)	Sim (36%)	Sim (58%)	Não (0%)	**80%**
04	1100	Sim (29%)	Sim (56%)	Sim (23%)	Não (0%)	Sim (7%)	**80%**
05	1200	Sim (50%)	Sim (62%)	Sim (56%)	Não (0%)	Sim (17%)	**80%**
06	1300	Sim (21%)	Sim (42%)	Sim (29%)	Não (0%)	Não (0%)	**60%**
07	1500	Sim (21%)	Sim (48%)	Sim (27%)	Não (0%)	Sim (33%)	**80%**
08	2000	Sim (33%)	Sim (45%)	Sim (5%)	Não (0%)	Sim (6%)	**80%**
09	2300	Sim (38%)	Sim (57%)	Sim (8%)	Não (0%)	Sim (11%)	**80%**
10	3500	Sim (8%)	Sim (41%)	Sim (13%)	Não (0%)	Sim (11%)	**80%**
Média	-	**100%**	**100%**	**100%**	**20%**	**70%**	**78%**

Pela tabela 7, três modelos aparentam ter tido uma presença de comentários em todos os problemas propostas. Era uma orientação do prompt que comentários devessem ser incluídos e os modelos GPT, Gemini e H2o indicaram ter compreendido essa orientação por completo.

Por outro lado, o modelo do Llama3 demonstrou ter comentários em 70% dos seus códigos, enquanto o Gemma indicou ter tido a pior performance nessa métrica, com apenas 20% de códigos com comentários.

A maioria das questões tiveram 80% de códigos com comentários presentes, mas a questão 06 demonstrou ter a pior porcentagem, com apenas 3 dos 5 modelos tendo códigos comentados, falhando nos modelos Gemma e Llama3.

A ideia de analisar a presença desses comentários é justamente compreender se o modelo consegue seguir uma orientação simples e se consegue explicar um código que ele mesmo gerou. Por isso, observamos que a maioria dos modelos aparentam ter tido uma boa performance nessa métrica, apesar dos 20% apresentados pelo Gemma.

4.7 Complexidade Cognitiva

Junto dessa análise de eficiência dos modelos, a tabela 8 apresenta a complexidade cognitiva que cada modelo teve em cada questão. Retomando um ponto da seção de Materiais e Métodos, o SonarCloud apresenta um limite aceitável de 15. Ao ultrapassar esse número, a ferramenta apresenta uma falha na manutenabilidade do software, orientando, então, seus ajustes nas funções afetadas.

Esta subseção avalia a complexidade cognitiva das soluções, ou seja, a facilidade com que um desenvolvedor pode entender e seguir a lógica do código. Menor complexidade cognitiva geralmente indica um código mais claro e mais fácil de manter.

Tabela 8 – Cumpre requisitos de complexidade cognitiva (quantidade)

Questão	**Nível**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	Sim (3)	Sim (3)	Sim (2)	Sim (2)	Sim (2)	**100%**
02	800	Sim (6)	Sim (4)	Sim (4)	Sim (4)	Sim (4)	**100%**
03	1000	Sim (4)	Sim (3)	Sim (3)	Sim (4)	Sim (4)	**100%**
04	1100	Sim (6)	Sim (10)	Sim (4)	Sim (3)	Sim (13)	**100%**
05	1200	Sim (0)	Sim (2)	Sim (0)	Sim (2)	Sim (4)	**100%**
06	1300	Sim (3)	Sim (5)	Sim (8)	Sim (7)	Sim (11)	**100%**
07	1500	Sim (9)	Sim (6)	Sim (9)	Sim (1)	Sim (6)	**100%**
08	2000	Sim (14)	Sim (12)	Não (25)	Sim (12)	Sim (4)	**80%**
09	2300	Sim (15)	Sim (6)	Sim (7)	Sim (9)	Sim (14)	**100%**
10	3500	Não (115)	Não (47)	Não (52)	Não (27)	Não (33)	**0%**
Média	-	**90%**	**90%**	**80%**	**90%**	**90%**	**88%**

4.8 Manutenabilidade

Finalmente, a análise da manutenabilidade examina quão fácil é modificar e atualizar o código gerado. A manutenibilidade é um indicador crucial da qualidade do software a longo prazo, já que soluções com manutenções menos frequentes podem possuir uma vida mais curta.

A tabela 9 indica que os modelos GPT e Llama3 têm o melhor desempenho, com 100% de acerto, sugerindo que esses modelos produzem códigos mais fáceis de manter.

Os modelos Gemini e H2o apresentaram desempenho semelhante, com 90% de acerto. Gemma teve o pior desempenho, com 70% de acerto, indicando possuir códigos que podem ser mais difíceis de fazer manutenções.

Tabela 9 – Cumpre requisitos de manutenabilidade (tamanho do impacto)

Questão	**Nível**	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
01	800	Sim (-)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**100%**
02	800	Sim (-)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**100%**
03	1000	Sim (-)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**100%**
04	1100	Sim (-)	Sim (-)	Sim (-)	Não (baixo)	Sim (-)	**80%**
05	1200	Sim (-)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**100%**
06	1300	Sim (-)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**100%**
07	1500	Não (médio)	Sim (-)	Sim (-)	Sim (-)	Sim (-)	**80%**
08	2000	Sim (-)	Sim (-)	Não (alto)	Sim (-)	Sim (-)	**80%**
09	2300	Sim (-)	Sim (-)	Sim (-)	Não (médio)	Sim (-)	**80%**
10	3500	Não (alto)	Não (alto)	Não (alto)	Não (médio)	Não (alto)	**0%**
Média	-	**80%**	**90%**	**80%**	**70%**	**90%**	**82%**

De todos, os modelos Gemini e Llama3 indicaram possuir o maior cuidado com a métrica de manutenabilidade. Apesar de ambos terem falhado na questão 10 com um alto impacto na métrica, desempenharam uma taxa de 90%, apresentando uma performance superior aos demais.

A questão que mais teve impacto foi a 10, o que se pode indicar é que isso foi ocasionado não somente por uma certa incapacidade dos modelos, mas também pela dificuldade do problema proposto, exigindo um volume de funções e linhas de código que podem exigir uma análise mais aprofundada para uma compreensão humana total da solução.

No entanto, todos os modelos aparentaram ter resultados positivos quanto à manutenabilidade, sendo a menor taxa de 70% do Gemma. Apesar de ser o modelo com a menor porcentagem, representou um desempenho superior na última questão, sendo o único modelo a não ter tido um alto impacto na manutenabilidade do software.

5 Conclusões e Trabalhos Futuros

5.1 Conclusão

Neste trabalho, foi realizada uma análise detalhada do desempenho de modelos de linguagem de larga escala (LLMs) na geração de códigos de programação. Utilizando diversos modelos de LLM, incluindo GPT, H2o, Gemini, Gemma-7B e Llama3, foram resolvidos problemas de programação extraídos da plataforma Codeforces. A avaliação foi feita com base em métricas específicas como exatidão do código, tempo de execução, memória consumida, presença de comentários e módulos, complexidade cognitiva e manutenabilidade.

Os resultados mostraram que, embora os LLMs sejam capazes de gerar códigos funcionais e, em muitos casos, eficientes, ainda há desafios significativos, especialmente em problemas de maior complexidade. Alguns modelos destacaram-se em termos de eficiência e qualidade do código gerado, enquanto outros apresentaram dificuldades em compreender e implementar corretamente os requisitos dos problemas mais avançados.

Os objetivos do estudo foram amplamente alcançados, pois foi possível não apenas comparar a performance dos modelos, mas também identificar suas limitações e pontos fortes. A pesquisa contribuiu para um melhor entendimento das capacidades e desafios dos LLMs na programação, oferecendo insights valiosos para futuros desenvolvimentos na área.

Na tabela 10 há a representação da porcentagem média de acerto para cada métrica trabalhada neste estudo. Uma maior porcentagem representa uma maior eficiência do modelo em atender a métrica requisitada para os problemas de programação propostos.

Dada a tabela 10 é possível tirar algumas conclusões em relação ao desempenho dos modelos em relação a cada métrica. Segue uma lista com os LLMs que tiveram melhor performance para cada métrica analisada:

- Exatidão: GPT;
- Eficiência de tempo: GPT e Llama3;
- Eficiência de memória: Todos;
- Modularização: Gemini e H2o;
- Comentários: GPT, Gemini e H2o;
- Complexidade Cognitiva: GPT, Gemini, Gemma e Llama3;

Tabela 10 – Porcentagem da eficiência de cada modelo em cada métrica

Métricas	**GPT**	**Gemini**	**H2o**	**Gemma**	**Llama3**	**Média**
Exatidão	60%	10%	20%	10%	20%	**24%**
Eficiência de Tempo	100%	80%	90%	80%	100%	**90%**
Eficiência de Memória	100%	100%	100%	100%	100%	**100%**
Modularização	70%	100%	100%	30%	60%	**72%**
Comentários	100%	100%	100%	20%	70%	**78%**
Complexidade Cognitiva	90%	90%	80%	90%	90%	**88%**
Manutenabilidade	80%	90%	80%	70%	90%	**82%**
Média	**85,7%**	**81,4%**	**81,4%**	**57,1%**	**75,7%**	**76,3%**

- Manutenabilidade: Llama3.

Em uma análise que não há a utilização de pesos para cada métrica, pode-se considerar que o GPT foi o modelo que melhor performou, já que está entre os melhores em 5 das 7 métricas analisadas.

Além disso, pode-se entender que a métrica de exatidão é a mais importante quando se trata da análise de um código voltado para a programação, já que um código correto pode ser considerado eficiente mesmo sem funções ou comentários presentes, mas não se pode dizer isso de um código errado que esteja comentado e bem modularizado.

Por essas razões, o GPT superou os demais modelos em diversas métricas quando se trata em geração de códigos para problemas de programação, mas principalmente na de exatidão, com 60% de acerto em 10 questões dos mais variados níveis.

Em contraste, modelos como Llama3 e H2o apresentaram uma taxa de acerto de 20%, enquanto Gemini e Gemma ficaram com apenas 10% na métrica de exatidão. Este último, aliás, não superou os demais modelos em nenhuma métrica, podendo ser apresentado como o com pior performance quando se trata da geração de códigos para problemas de programação.

Os objetivos inicialmente traçados foram alcançados na medida em que conseguimos identificar claramente as diferenças de desempenho entre os modelos. O GPT demonstrou uma superioridade significativa, sugerindo que está mais bem equipado para lidar com tarefas de programação complexas, ao passo que os outros modelos mostraram limitações em várias questões. Esta análise reforça a importância de escolher o modelo adequado para tarefas específicas de programação, especialmente em contextos que exigem alta precisão e eficiência.

5.2 Trabalhos Futuros

Para trabalhos futuros, várias direções podem ser exploradas para aprofundar e expandir os conhecimentos adquiridos neste estudo:

- Aprimoramento das Métricas de Avaliação: A inclusão de métricas adicionais, como a qualidade dos comentários, a estrutura do código, a aderência a boas práticas de programação e de métricas de qualidade de software adicionais, podem oferecer uma visão ainda mais completa da qualidade dos códigos gerados pelos LLMs.
- Testes com mais problemas de programação: expandir os testes para incluir uma gama mais ampla de problemas de programação, incluindo aqueles que exigem conhecimento de domínios específicos.
- Comparação com programadores humanos: realizar uma comparação direta entre os códigos gerados pelos LLMs e os desenvolvidos por programadores humanos, avaliando não apenas a eficiência e correção, mas também a criatividade e a capacidade de otimização das soluções. Além disso, fazer uma análise para compreender em qual nível (em *rating*) esses modelos teriam dentro da plataforma do Codeforces.
- Avaliação de modelos emergentes: continuar a avaliação de novos modelos de LLMs à medida que são desenvolvidos, comparando suas capacidades e identificando tendências e avanços na geração de código de programação.
- Pesos para as métricas: definir um número único que identifique a qualidade de geração de códigos dos modelos, de forma que cada métrica tenha um peso para a construção desse número final de avaliação. Com uma pontuação, a análise de quais modelos são mais eficientes seria mais rápida e eficaz.
- Insistir com o modelo: após o modelo gerar uma resposta incorreta, retornar para ele com o motivo do erro, de forma com que ele possa compreender o que causou a ineficiência na solução e devolver uma solução aprimorada, como uma segunda chance de acertar.

Ao explorar essas áreas, futuras pesquisas poderão não apenas melhorar a eficiência e a qualidade dos códigos gerados por LLMs, mas também ampliar o escopo de aplicação dessas tecnologias na indústria de software, contribuindo para o avanço contínuo da inteligência artificial e suas aplicações práticas.

Referências

CAMPBELL, G. A. *Cognitive Complexity a new way of measuring understandability*. [s.n.], 2023. Disponível em: <https://www.studocu.com/in/document/mgms-college-of-engineering/computer-science-and-engineering/cognitive-complexity-sonar-guide-2023/73011188>. Citado na página 28.

CARVALHO, C. O que é python? 2024. Disponível em: <https://www.alura.com.br/artigos/python>. Citado na página 30.

DATACAMP. O que é o kaggle? 2024. Disponível em: <https://www.datacamp.com/pt/blog/what-is-kaggle>. Citado na página 30.

DSA. *O Que São Large Language Models?* 2023. Disponível em: <https://blog.dsacademy.com.br/o-que-sao-large-language-models-llms/>. Citado na página 31.

GILLESPIE N., L. S. C. C. P. J. . A. A. *Trust in Artificial Intelligence, a global study*. 2023. Disponível em: <https://static.poder360.com.br/2023/03/Inteligencia-Artificial-2023.pdf>. Citado na página 21.

KOUL, N. *Prompt Engineering for Large Language Models*. [s.n.], 2023. Disponível em: <https://books.google.com.br/books?id=e9_jEAAAQBAJ&newbks=0&printsec=frontcover&pg=PT138&dq=all+of+large+language+models&hl=pt-BR&source=gbs_vpt_read&redir_esc=y#v=onepage&q=all%20of%20large%20language%20models&f=false>. Citado na página 31.

KUNDU, D. *Unleashing the Potential of Domain-Specific LLMs*. 2023. Disponível em: <https://www.analyticsvidhya.com/blog/2023/08/domain-specific-llms/#:~:text=A%20domain%2Dspecific%20LLM%20aims,LLM%20to%20domain%2Dspecific%20data.> Citado na página 31.

LARGUESA, R. P. *Engenharia de Prompt para Programadores*. [s.n.], 2023. Disponível em: <https://www.google.com.br/books/edition/Engenharia_de_Prompt_para_Programadores/Ipm-EAAAQBAJ?hl=pt-BR&gbpv=1&dq=engenharia+de+prompt+com+ia&printsec=frontcover>. Citado 2 vezes nas páginas 36 e 37.

PRIVACY Policy. 2024. Disponível em: <https://codeforces.com/privacy>. Citado na página 30.

PROGRAMAçãO, H. *Como Usar a API do ChatGPT*. 2023. Disponível em: <https://www.youtube.com/watch?v=ZwfZlqTzsuA>. Citado na página 32.

RADFORD, A. et al. Improving language understanding by generative pre-training. *arXiv preprint arXiv:1803.08493*, 2018. Citado na página 31.

SALVIANO, C. F. *Qualidade de Software*. [s.n.], 2020. Disponível em: <https://www.google.com.br/books/edition/Qualidade_de_software/uXr_DwAAQBAJ?hl=pt-BR&gbpv=1>. Citado 2 vezes nas páginas 25 e 26.

TEAM, T. M. G. et al. Gemma. Kaggle, 2024. Disponível em: <https://www.kaggle.com/m/3301>. Citado na página 34.

THOUGHTWORKS. Ollama. 2024. Disponível em: <https://www.thoughtworks.com/pt-br/radar/tools/ollama#:~:text=Ollama%20%C3%A9%20uma%20ferramenta%20c%C3%B3digo,com%20ferramentas%20como%20o%20Ollama.> Citado na página 30.